CS324: Welcome, Overview, Intro to Algorithms, Insertion Sort

January 13, 2020

To do before next class:

- Read the syllabus and calendar for the course (posted to Moodle)
- Read CLRS chapters 1 and 2, section 3.1
- Complete HW0 [looks like a quiz on moodle, but it really is HW0]

Today's learning outcomes:

- Understand the computer and pseudo-code model for describing algorithms
- Read a formal description of a problem and algorithm (insertion sort)
- Prove an algorithm is correct

Course Overview

Tammy's Teaching Philosophy

- Practice, practice, practice
- We'll practice in class and you will practice via homework (work with others and get a variety of ideas)

Key Points About This Course

- Moves quickly – no programming in this course; we will program in pseudocode
- Calendar will be updated on Moodle if we get ahead or behind
- In-class activities and quizzes – practice; may have some interview-style questions and programming contest questions
- 4 midterm exams
- 1 final exam
- 2 homework assignments per exam
    - 2 free late days (2 24-hour grace late submissions or 1 48-hour grace late submission)

Student Outcomes

- Analyze time-complexity of algorithms
- Analyze design paradigms and trade-offs of algorithms
- Execute (sometimes clever) algorithms and determine their efficiency
- Prove correctness of algorithms
- Prove that a problem is NP-complete (determine if a problem is intractable)

What is your goal for the course? ____

Why is studying algorithms important?

- An important part of the design process – just as important as choosing hardware, programming language, storage/data structures/database model, user experience and interfaces

- More tools in your toolbox
- Helpful for technical interviews and your professional life
- Documenting problem-solving process; communicating ideas to others
- Proving a solution is fast (or not fast)

Activity: How might we analyze algorithms? (Discuss in small group)

1.

2.

3.

4.

Where do we find applications of algorithms?

Prime number detection -> Cryptography and security

Matrix multiplication -> Computer graphics

Scheduling -> Compilers, Operating Systems, Distributed Systems, Software Engineering

Network-Flow algorithms -> Distribution systems (electricity)

Major focus of our analysis will be about time:

Given an algorithm, what is the worst-case running time as a function of the size of the input?
Given an algorithm, what is the average-case running time as a function of the size of the input?
Given a problem, what is the running time of the best (fastest) algorithm?

Complete notecards for course (see moodle for questions)

Machine Model and Code in this Course

We will write algorithms in pseudo-code in this course (this will be our programming language to communicate our ideas)

Machine model:

- RAM
- One operation at a time
- +, -, >, >=, ==, <, <= take constant time (not quite true when operands are large)
- Reading or writing to memory takes constant time (not quite true if data is stored in secondary memory)
- Arrays, structs, strings are treated as multiple memory locations

Pseudo-code for CS 324	**Meaning**
`for i = 1 to n`	For loop executing for i initialized to 1 and incrementing up to and including i set to n
`while i > 0`	While loop executing when i is greater than 0
`repeat ____ until ____`	Repeat statements after repeat while condition after until is true
`v`	Variable v
`A[i]`	Element in array (list) A at position i, lists are usually indexed starting at index 1
`=`	Assignment operator
`i = j = e`	Assign j to e and assign i to j
+, -, *, /, %	Arithmetic operators; assume floating point arithmetic for +, -, *, /
`if and if else`	Conditionals
`>, <, ==, !=, <=, >=`	Inequality and equality operators
Indentation	Indicates blocking – no use of { } to indicate blocks
`x.f`	Member f of object (struct) x
`x = y` (as objects)	Assignment of one object to another is done via pointers, so after x is assigned to y, both objects refer to the same object
`NIL`	Null
Functions/Procedures are written in all caps	Function/Procedure includes parameters (w/o types) and parameters are passed by value (called procedure receives own copy of the parameters and changes are not seen by the calling procedure); when objects/arrays are passed, they are passed by reference (pointers to object) so changes made to elements of these compound data types are seen by the calling procedure
`return`	Return statement; exits function
`and`	Boolean && (short circuits, so if first expression is false, the second one is not evaluated)
`or`	Boolean || (short circuits, so if the first expression is true, the second one is not evaluated)
`error`	Indicates an error occurred
`//`	Comments

Let's look at an example of a familiar algorithm for sorting: insertion sort

Formal problem statement
Input: A sequence of numbers $<a_1, a_2, a_3, \dots a_n>$
Output: A permutation (reordering) $<a'_1, a'_2, a'_3, \dots a'_n>$ such that $a'_1 <= a'_2 <= a'_3 \dots <= a'_n$.

```
INSERTION-SORT(A)
1   for j = 2 to A.length                //A.length is the length of the array A
2       key = A[j]
3       // Insert A[j] into sorted sequence A[1..j-1]
4       i = j-1
5       while i > 0 and A[i] > key
6           A[i+1] = A[i]
7           i = i - 1
8       A[i+1] = key
```

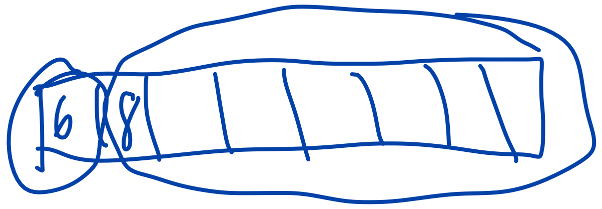
Demo with people and cards

Practice: Follow the insertion-sort algorithm with the following input:

A = [10, 4, 8, 16, 3, 7]

j = 2	key = 4	i = 1	A = 4, 10, $\underline{8}$, 16, 3, 7
j = 3	key = 8	i = 2	4, 8, 10, $\underline{16}$, 3, 7
j = 4	key = 16	i = 3	4, 8, 10, 16, $\underline{3}$, 7
j = 5	key = 3	i = 4	3, 4, 8, 10, 16, $\underline{7}$
j = 6	key = 7	i = 5	3, 4, 7, 8, 10, 16

Another implementation of insertion-sort using recursion



```
INSERTION-SORT-R(A)
1    if A == NIL return NIL
2    else return INSERT(A.head, INSERTION-SORT-R(A.tail))  //A.head is the first element of A
                                                           //A.tail is all but the first element of A

INSERT(key, A)
1    if A == NIL return [key]                  // [key] is list containing key
2    else if key <= A.head return [key]+A      // [key]+A means concatenate list of key with A
3    else return [A.head] + INSERT(key, A.tail)
```

Does this work?

ISR will stand for INSERTION-SORT-R and IN will stand for INSERT.

```
ISR(<10, 4, 8, 16, 3, 7>)
-> IN(10, ISR(<4, 8, 16, 3, 7>))
-> IN(10, IN(4, ISR(<8, 16, 3, 7>)))
-> IN(10, IN(4, IN(8, ISR(<16, 3, 7>))))
-> IN(10, IN(4, IN(8, IN(16, ISR(<3, 7>)))))
-> IN(10, IN(4, IN(8, IN(16, IN(3, ISR(<7>))))))
-> IN(10, IN(4, IN(8, IN(16, IN(3, IN(7, ISR(<NIL>)))))))

-> IN(10, IN(4, IN(8, IN(16, IN(3, IN(7, NIL))))))
-> IN(10, IN(4, IN(8, IN(16, IN(3, <7>)))))
-> IN(10, IN(4, IN(8, IN(16, <3, 7>))))
-> IN(10, IN(4, IN(8, <3> + IN(16, <7>))))
-> IN(10, IN(4, IN(8, <3> + <7> + IN(16, NIL))))
-> IN(10, IN(4, IN(8, <3, 7, 16>)))
-> IN(10, IN(4, <3> + IN(8, <7, 16>)))
-> IN(10, IN(4, <3> + <7> + IN(8, <16>)))
-> IN(10, IN(4, <3, 7, 8, 16>))
-> IN(10, <3> + IN(4, <7, 8, 16>))
-> IN(10, <3, 4, 7, 8, 16>)
-> <3> + IN(10, <4, 7, 8, 16>)
-> <3> + <4> + IN(10, <7, 8, 16>)
-> <3, 4> + <7> + IN(10, <8, 16>)
-> <3, 4, 7> + <8> + IN(10, <16>)
-> <3, 4, 7, 8, 10, 16>
```

How would we prove that the iterative insertion sort is correct? (For any input, the returned array is in sorted order)

How would we prove that the recursive version of insertion sort is correct?

First, the proof of correctness for the iterative version:

```
INSERTION-SORT(A)
1    for j = 2 to A.length                  //A.length is the length of the array A
2        key = A[j]
3        // Insert A[j] into sorted sequence A[1..j-1]
4        i = j-1
5        while i > 0 and A[i] > key
6            A[i+1] = A[i]
7            i = i - 1
8        A[i+1] = key
```

Proof:

We will prove that that at the end of the insertion-sort procedure, A is sorted by showing a loop invariant holds. The loop invariant refers to the for loop on line 1.

Loop invariant: A[1..j-1] is a sorted permutation of the original array A

To prove that the loop invariant holds, we must show that it is true *initially* before the loop executes, the invariant is maintained in the next execution -- *maintenance*, and that the invariant is true at the end of the loop – *termination.*

Loop invariant initialization (before): base case

Consider the state prior to the loop executing. Assume j = 2. Then, A[1..1] is just a list with a single element. Therefore, A[1] is itself a sorted list.

Loop invariant maintenance (iterative step): induction step

Suppose A[1..j-1] is sorted (induction hypothesis). The body of the for loop moves A[j-1], A[j-2], A[j-3] and so on one position to the right until it finds the location for A[j] (lines 4 to 7). The algorithm inserts A[j] into the proper position. Thus, at the end of the iteration, A[1…j] is now a sorted permutation prior to incrementing j for the next iteration.

Loop invariant termination (after): conclusion

The condition for causing the loop to terminate is that j > A.length. Because each loop iteration increments j by 1, j must be A.length+1 at this time. Therefore, the subarray A[1…(A.length+1)-1] is sorted by the loop invariant. This is the original array A[1..A.length].

Second, the proof of correctness for the iterative version:

```
INSERTION-SORT-R(A)
1       if A == NIL return NIL
2       else return INSERT(A.head, INSERTION-SORT-R(A.tail))  //A.head is the first element of A
                                                              //A.tail is all but the first element of A

INSERT(key, A)
1       if A == NIL return [key]                   // [key] is list containing key
2       else if key <= A.head return [key]+A       // [key]+A means concatenate list of key with A
3       else return [A.head] + INSERT(key, A.tail)
```

Proof:
We will prove that each function produces a correct solution and that the function terminates.

INSERT

First, let's prove INSERT satisfies the property: if input A of N items is sorted, then INSERT(key, A) of N+1 items is sorted. Since the function is recursive, mathematical induction can be applied directly.

B.C.

Case 1: Assume A is NIL so N=0. INSERT will return the array [key]. Since it is a single-item list, it is sorted with 1 item. [base case]

B.C.

Case 2: Assume A is not NIL with length N and key <= A.head. Since key <= A.head and A is sorted, then key is less than or equal to all elements in A. Thus [key] + A is a sorted list of N+1 items. [base case]

INDUCTION STEP

Case 3: Assume A is not NIL, A has N items, and key > A.head. The length of A.tail is N-1. Therefore, we can apply mathematical induction to assume that INSERT(key, A.tail) returns a sorted list of N-1+1 items in {key} U {A.tail}. Since A.head < key, appending A.head to the front of INSERT(key, A.tail) creates a sorted list of N+1 items.

In all three cases, INSERT(key, A) returns a sorted list of {key} U A. Therefore, the INSERT function is correct.

Second, let's prove INSERTION-SORT-R works correctly y satisfying the property: for any input A of N items, INSERTION-SORT-R(A) returns a permutation of A that is sorted.

B.C.

Case 1: Assume A is NIL with N=0. INSERTION-SORT-R returns NIL, which is a permutation of NIL that is sorted.

INDUCTION

Case 2: Assume A is not NIL with length N. Because A.tail has length N-1, we can apply the induction hypothesis and assume INSERTION-SORT-R(A.tail) returns a sorted permutation of A.tail. Earlier, we proved that INSERT(key, A) returns a sorted list of {key} U A if A is sorted. A.tail is sorted and A.head is a key that can be inserted. Therefore, INSERT(A.head, INSERTION-SORT-R(A.tail)) returns a sorted list of {A.head} U A.tail containing N total elements.

In both cases, INSERTION-SORT-R returns a sorted permutation of its input A.

Practice: Complete the pseudocode for linear search and prove its correctness with a loop invariant.

Input: List A of N items, key
Output: Lowest index i such that A[i] == key or the special value NIL if key is not found in A

Pseudocode:

LINEAR-SEARCH(A, key)

```
1    i = NIL
2    for j = 1 to A.length
3        if A[j] == key
4
5            return j
6

     return NIL
```

Proof of Correctness:

We will prove that a loop invariant holds prior to the loop body, on each iteration of the loop body, and at the termination of the loop.

Loop invariant: There is no index k < j such that A[k] == key.

Loop initialization: j = 1. No elements in A at positions less than 1.

Loop maintenance: Only way for loop to continue is if A[j] != key. ∴ A[1...j] is not equal to key.

Loop termination:

2 cases:
loop terminates early
loop terminates w/ NIL

Next time: We will look at the *complexity* of insertion-sort (how long does it take to execute given N items?).

CS324: Algorithm Paradigms, Insertion Sort Analysis, Running Time, Asymptotic Notation

January 15, 2020

To do before next class:

- Review mathematical induction
- Read CLRS Appendix A
- Start looking at HW1 – you can do most problems (it includes merge sort which is in the textbook and should be review from data structures; you should already know how it executes)

Today's learning outcomes:

- Analyze the running time of an algorithm (iterative insertion sort)
- Understand best case, worst case, and average case analysis
- Identify examples of algorithms for different paradigms
- Define key terms with regard to analysis of problems and algorithms
- Understand theta, O, omega, little o, and little omega notation

---

Review insertion sort below

INSERTION-SORT(A)

```
1   for j = 2 to A.length                  //A.length is the length of the array A
2       key = A[j]
3       // Insert A[j] into sorted sequence A[1..j-1]
4       i = j-1
5       while i > 0 and A[i] > key
6           A[i+1] = A[i]
7           i = i - 1
8       A[i+1] = key
```

Do you have any questions about how it works or why it works?

Now, how would we determine how long it takes to execute?

Some details about how we count in this course:

- Loops setting N different values will take N+1 tests (to do the final condition check for something like i = 0; i < n; i++)
- T_j denotes the number of times for the jth iteration of a loop
- Comments do not cost any time

```
INSERTION-SORT(A)                                               cost    times
1   for j = 2 to A.length                                       $c_1$   n
2       key = A[j]                                              $c_2$   n-1
3       // Insert A[j] into sorted sequence A[1..j-1]           0       n-1
4       i = j-1                                                 $c_4$   n-1
5       while i > 0 and A[i] > key                              $c_5$   $\Sigma_{j=2 \text{ to } n} t_j$
6           A[i+1] = A[i]                                       $c_6$   $\Sigma_{j=2 \text{ to } n} (t_j - 1)$
7           i = i – 1                                           $c_7$   $\Sigma_{j=2 \text{ to } n} (t_j - 1)$
8       A[i+1] = key                                            $c_8$   n-1
```

$c_1 \times n$

$c_2 \times (n-1)$

The total running time T(n) for insertion sort is:

$$T(n) = \underline{c_1 n} + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

What is the best case (what does the array A look like) for insertion sort? (Hint: minimize the t_j terms)

What is the worst case (what does the array A look like) for insertion sort? (Hint: maximize the t_j terms)

What is the average case (what does the array A look like) for insertion sort?

Best case: array already in sorted order, no shifting of elements in while loop, so $t_j = 1$.

Runtime is:

$$T(n) = c_1 n + c_2(n-1) + \; c_4(n-1) + \; c_5(n-1) + \; 0 + 0 + c_8(n-1)$$

Therefore, the best-case runtime of insertion sort is linear.

Worst case: array is in reverse sorted order, so every element needs to be shifted in while loop, so $t_j = j$.
$\underline{\sum_{j=2}^{n} j}$ is $\frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1)$ is $\frac{n(n-1)}{2}$

Replacing these values in the runtime gives:

$$T(n) = c_1 n + c_2(n-1) + \; c_4(n-1) + \; c_5\left(\underline{\frac{n(n+1)}{2} - 1}\right) + \; c_6\left(\underline{\frac{n(n-1)}{2}}\right) + c_7\left(\underline{\frac{n(n-1)}{2}}\right) + c_8(n-1)$$

There, the worst-case runtime of insertion sort is quadratic (n^2).

Average case: array is random for average case, so on average about half the elements in A[1..j-1] are less than A[j] and about half are greater than A[j]. So, t_j is about j/2. The sum $\sum_{j=2}^{n}(j\,/2)$ is still quadratic, so the average case of insertion sort is quadratic (n^2).

Which case do we generally analyze with respect to algorithms? Worst-case, unless otherwise specified; Note that average case analysis is generally more difficult to do.

Algorithm Paradigms

Incremental – solve slightly smaller problem and incorporate additional piece

Insertion sort

Divide and conquer – divide problem into approximately equal subproblems and combine solutions

Merge sort

Greedy – find locally-optimal solutions and combine them

Fractional knapsack

Randomized – use randomization to get high probability of efficient or correct solution

Median-finding

Dynamic programming – re-use saved subproblem solutions

String alignment

Many of the ACM programming competition problems use dynamic programming

Transformation – transform problem with already known solution

NP-complete problems such as Clique, Vertex Cover

VanDeGrift's dissertation

Key Terms

Problem: statement of what computation is supposed to do, often detailing the input and output (declarative – what)

Algorithm: sequence of steps that solves a problem, transforming the input into the desired output (procedural – how)

Input size: the number of items in the input, often referred to as n or N

Complexity of an algorithm: how long it takes an algorithm to run as a function of its input size

Complexity of a problem: how long it takes the FASTEST of all algorithms for solving the problem

problem {

Upper bound: Generally based on the fastest known algorithm for the problem at the time

Lower bound: Generally based on the mathematical proof (for example, search has a lower bound of N-time since every item must be scanned)

Goals of CS theoreticians: Decrease problem's upper bound by finding better algorithms; Increase problem's lower bound by finding better mathematical proofs; When the upper bound is equal to the lower bound, we know the complexity of the problem.

decision tree for comparisons is size NlgN: it is $\Omega(n\lg n)$

Problem	Lower Bound	Upper Bound
Sorting (# comparisons)	Ω (nlgn)	O(nlgn)
Finding median	Ω (n)	O(n)
Matrix multiplication	Ω (n^2)	O($n^{2.376}$)
Graph coloring	Ω (n)	O(2^n)

Pop Quiz:

Is *quicksort* a problem or an algorithm? Algorithm

Is *returning the number of occurrences of a key in an array* a problem or an algorithm? Problem

Is *returning a route from UP to PSU* a problem or an algorithm? Problem

Which is more difficult? Analyzing the complexity of an algorithm or the complexity of a problem? usually a problem

Which is larger? The upper bound for a problem or the lower bound for a problem? upper bound

Formalization of notation

We have different kinds of notation to specify bounds. You have likely seen big-O in CS 305 as a way to specify how a function grows asymptotically (proportional to N, N^2, lgN, etc.). Now, in this course, we will use the appropriate notation for the appropriate bound.

Notation	How to say it	What it means	Another way to think about it
$\theta(N^2)$	Big-theta or Theta	Grows proportional to N^2	Runtime is bounded above and below by N^2 (trapped)
$O(N^2)$	Big-O ("big oh")	Grows no faster than proportional to N^2	Runtime has upper bound of N^2
$\Omega(N^2)$	Big-Omega	Grows no slower than proportional to N^2	Runtime has lower bound of N^2
$o(N^2)$	Little-o ("little oh")	Grows slower than proportional to N^2	Runtime has strict upper bound of N^2
$\omega(N^2)$	Little-Omega	Grows faster than proportional to N^2	Runtime has strict lower bound of N^2

Check: Is a linear-time algorithm O(N^2)? ___yes___

Check: Is a linear-time algorithm θ(N^2)? ___no___

Check: Is a linear-time algorithm Ω(N^2)? ___no___

Check: is a linear-time algorithm o(N)? ___no___

	Θ	O	Ω	o	ω
best	N	N			
worst	~~A~~ N^2	~~A~~ N^2	~~A~~		
avg	N^2	N^2			

Mathematical definitions of these bounds:

THETA

$\Theta(g(n))$ = {f(n) | there exists positive constants c_1, c_2, and n_0 such that $0 <= \underline{c_1 g(n)} <= f(n) \underline{<=} c_2 g(n)$ for all $n >= n_0$}

O

$O(g(n))$ = {f(n) | there exist positive constants c_2 and n_0 such that $0 <= f(n) <= \underline{c_2 g(n)}$ for all $n \underline{>=} n_0$}

OMEGA

$\Omega(g(n))$ = {f(n) | there exist positive constants c_1 and n_0 such that $0 <= \underline{c}_1 g(n) <= f(n)$ for all $n >= n_0$}

LITTLE-o

$o(g(n))$ = {f(n) | for any positive constant $c_2 > 0$, there exists a constant $n_0 > 0$ such that $0 <= f(n) \underline{<} c_2 g(n)$ for all $n >= n_0$}

LITTLE-OMEGA

$\Omega(g(n))$ = {f(n) | for any positive constant $c_1 > = 0$, there exists a constant $n_0 > 0$ such that $0 <= c_1 g(n) \underline{<} f(n)$ for all $n >= n_0$}

Check: What are the differences between O and OMEGA?

Check: What are the differences between OMEGA and LITTLE-OMEGA?

Check: What are the differences between THETA and O?

Practice:

Is $2n^2 + \theta(n) = \theta(n^2)$? Yes, no matter how the function is chosen on the left of the =, we can choose a function on the right of the = to make the equation valid.

Is $2n^2 + \underline{3n + 6} = 2n^2 + \theta(n)$? Yes

Is $2n^2 + 3n + 6 = \theta(n^2)$? Yes

Is $2n^2 + 3n + 6 = \theta(n^3)$? no

Is $2n^2 + 3n + 6 = O(n^3)$? yes

If $f(n) = \theta(g(n))$, must $g(n) = \theta(f(n))$? yes

If $f(n) = O(g(n))$, must $g(n) = \Omega(f(n))$? yes

CS324: Induction

January 17, 2020

To do before next class:

- Remember, Monday is a holiday so we do not have class on January 20
- Read CLRS chapter 3.2 and 9.2
- Continue working on HW1

Today's learning outcomes:

- Review the roadmap of mathematical induction
- Practice applying induction to prove statements

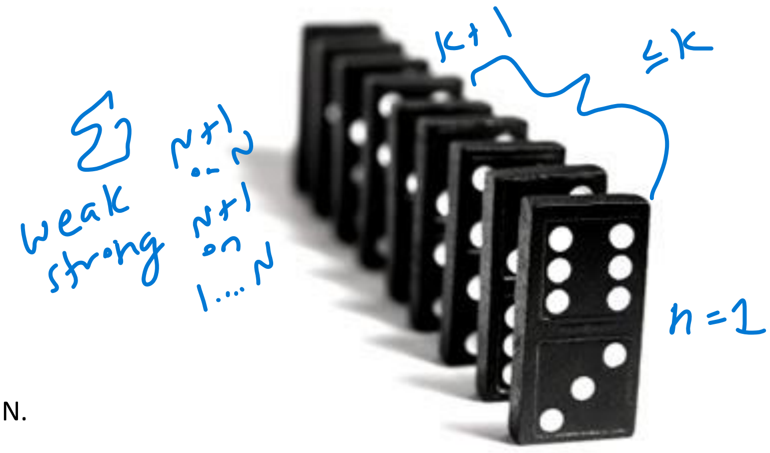
What do you remember about induction from MTH 311?

base cases

$N \to N+1$

I.S. I.H.

conclusion



Basic idea:
Think of Dominos

Show truth for a base case (typically N = 1)

Show that if statement is true for N-1, it is true for N.

-This is a hypothesis of weak induction

-Strong induction is assuming it is true for N=1, 2, … N-1 and showing it is true for N. This version is needed for many recursive data structures that could split sub-problems into smaller groups than just the size N-1.

I like to think of induction as unwinding the recursion. The base case in induction is the base case of recursion. The recursive call assumes the function solves the problem for a smaller input and this is like the induction step.

Structure of induction proofs often includes four parts:

1. Proof of the base case(s)
2. Induction hypothesis
3. Induction step
4. Conclusion

Example of an induction proof

Statement: The sum of numbers 0, 1, 2, …n is less than or equal to n^2 for all n >= 0.

Mathematically, we can write this as:
Statement S: $\underline{\sum_{k=0}^{n}} k \leq n^2$ for all integers of n >= 0

Proof: We will prove Statement S by induction on n.

Base case:
Consider n = 0. The sum of 0 is equal to 0, which is less than or equal to 0^2. Therefore, Statement S holds for n=0.

✱ strong induction

Induction hypothesis:
Assume that for all integers m such that $\underline{0 <= m} \boxed{<} n$, $\sum_{k=0}^{m} k \leq m^2$.

Induction step:
Consider the sum $\sum_{k=0}^{n} k$ for n > 0. We can split the sum into parts, splitting out the highest order term from the rest of the summation as follows:

$$\underline{\sum_{k=0}^{n}} k = [n + \boxed{\left(\sum_{k=0}^{n-1} k\right)}]$$

We can apply the induction hypothesis to the sum from 0 to (n-1) since (n-1) is smaller than n. Thus, the sum on the right-hand side is equal to $(n-1)^2$. Thus, we have:

$$\sum_{k=0}^{n} k \overset{\leq}{=} [n + \underline{(n-1)^2}]$$

With algebra, we can simply the right-hand side to:

$$= n + n^2 - 2n + 1$$

$$= n^2 \underline{- n + 1}$$

Since n >0, (-n + 1) is at most 0. For n=1, (-n + 1) is 0. For n larger than 1, (-n + 1) is negative. Therefore, we have the value n^2 plus a 0 or negative term, which has to be smaller than n^2:

$$\leq n^2$$

Conclusion:
We have proved by mathematical induction that for all non-negative integers of n, $\sum_{k=0}^{n} k \leq n^2$.

Now, why did that proof work?

Assume Billy wants to know that 0+1+2+3+4+5+6 <= 6^2. We could just do the math directly. If Billy wants to know it works for all values of n, we can't really write an infinite number of mathematical statements. Instead, it is like dominos. We proved it worked for 0. Then, it works for 1 due to the induction step. Then, it works for 2 due to the induction step. Etc.

Practice: In a small group, prove the following statement is true via mathematical induction:

Statement T: 3^n – 1 is a multiple of 2 for all positive integers n $3^n - 2$

Proof: We will prove Statement T by induction on n.

Base case: Consider $n=1$. $3^1 - 1 = 2$. 2 is a multiple of 2, so T is true for $n=1$.

Induction hypothesis: Assume $3^m - 1$ is a multiple of 2 for $1 \le m < n$.

Induction step:

Consider $3^n - 1$ for $n > 1$.

$$= 3 \cdot 3^{n-1} - 1$$

$$= 3 \cdot (2d+1) - 1 \quad // \text{ I. H.} \quad // \; 2d+1 \text{ is odd since } 3^{n-1} - 1 \text{ is even}$$

$$= 6d + 3 - 1$$

$$= 6d + 2$$

$$= 2\underbrace{(3d+1)}_{\text{is an integer}} \quad // \; d \text{ is an int}$$

So $3^n - 1$ is a multiple of 2.

Conclusion:

By M.I. on n $3^n - 1$ is a multiple of 2 for integers $n \ge 1$.

Another example with a recurrence
A recurrence is a recursive function. We will see more about recurrences in a few lectures. You have seen a recurrence in the form a Fibonacci numbers.

Let's define the following recurrence:

$T(n) = 0$ $\quad n=0$
$T(n) = T(n-1) + 2n – 1$ $\quad n>0$

Let's see how it grows:

n	T(n)
0	0
1	0 + 2*1 – 1 = 1
2	1 + 2*2 – 1 = 4
3	4 + 2*3 – 1 = 9
4	You do this one 16
5	You do this one 25

What is the pattern?

$T(n) = n^2$.

Let's prove that the recurrence $T(n) = n^2$ for integers $n >= 0$.

Proof:
We will proceed with mathematical induction to prove that $T(n) = n^2$ for integers $n >= 0$.

Base case:
Consider n=0. By definition, T(n) = 0 for n=0, and $0 = 0^2$. Thus, the statement is true for n=0.

Induction hypothesis:
Assume that for all integers m such that $0 <= m < n$, $T(m) = m^2$.

Induction step:
Consider T(n):

$$\begin{aligned} T(n) &= \underline{T(n-1)} + 2n – 1 \\ &= \underline{(n-1)}^2 + 2n – 1 && \text{// by induction hypothesis since } (n-1) < n \\ &= (n^2 – 2n + 1) + 2n - 1 && \text{// algebra} \\ &= n^2 && \text{// simplification} \end{aligned}$$

Conclusion:
We have proved by mathematical induction that for all non-negative integers of n, $T(n) = n^2$.

Practice: In a small group, prove the following statement is true via mathematical induction:

T(n) = 2 n=1 – B.C.
T(n) = 2T(n-1) n>1 – R.C

Prove that $T(n) = 2^n$ for all positive integers n.

Proof:

Base case: $n=1$. $2^1 = 2$ so it works for the base case.

Induction hypothesis: Assume that for integers m, $1 \le m < n$, $T(m) = 2^m$.

Induction step: Consider $T(n)$. for $n>1$.

$$T(n) = 2\underbrace{T(n-1)}_{}, \qquad \text{I.H.}$$

$$T(n) = 2 \cdot 2^{n-1}$$
$$= 2^n$$

Conclusion:

By M.I. on n, $T(n) = 2^n$ for integers $n \ge 1$.

Note, sometimes induction proofs use inequalities instead of equality in the statement. Let's try one:

Statement: $4^{n-1} > n^2$ for integers n >= 3.

Prove this by mathematical induction:

base case:
Conrider $n=3$. $4^{(3-1)} = 4^2 = 16$. $16 > 9 = 3^2$. Thus, it holds for the base case

induction hypothesis:
Assume for $3 \le m < n$, $4^{m-1} > m^2$ where m is an integer.

induction step:
Consider 4^{n-1} where $n \ge 4$.

$$\begin{aligned} 4^{n-1} &= 4 \cdot 4^{n-2} \\ &> 4 \cdot (n-1)^2 \quad \text{via induction hypothesis} \\ &= 4(n^2-2n+1) \\ &= 4n^2 - 8n + 4 \\ &= n^2 + \underbrace{(3n^2 - 8n + 4)}_{>0 \text{ when } n \ge 4} \\ &= n^2 + k \quad \text{where } k > 0 \\ &> n^2 \end{aligned}$$

conclusion:
By M.I. on n, $4^{n-1} > n^2$ for $n \ge 3$ where n is an integer.

CS324: More Math Foundations: Summations

January 24, 2020

To do before next class:

- HW2 due Wed, Feb 5 (9:15am to Moodle) read ch. 4

Today's learning outcomes:

- Review summations and properties
- Practice writing, reading, and applying summations

What is a sum?

3 + 5
2 + 24

What is a summation?

1+2+3+4+5 //this one is a finite sum(mation)
2+4+6+8+10+… //this one is an infinite sum(mation)

What does a summation in formal notation look like?

$$\sum_{k=1}^{n} a_k = a_1 + a_2 + a_3 + \ldots + a_n$$

$$\sum_{1\le k\le n} a_k$$

$$\textstyle\sum_{1\le k\le n} a_k$$

Are the above summations equivalent? yes

Are the above summations finite or infinite? finite

Infinite sum would look like:

$$\sum_{k=1}^{\infty} a_k$$ rare

$$\sum_{k\ge 1} a_k$$ common

Why would we use summations in CS?

- Adding steps for algorithm analysis
- Finding upper and lower bounds for runtimes

Example:

- We have already seen the analysis for insertion sort where we added up the number of steps.
  - In the worst case, shifting in insertion sort takes 1+2+3+4+…+n steps which is a summation.

Some sums have **closed-form** solutions, so we can get rid of the giant sigma. To derive closed-form solutions, we often need to do some algebra. Hopefully, summations are review for you from Calculus 2.

Check your memory:
What is the closed-form solution for the following sum:

1 + 2 + 3 + 4 + … + N = $\sum_{1 \le k \le N} k$

Hint: Add the first term to the last, second term to second to last, etc.

$\frac{n(n+1)}{2}$

Operations on Summations
A summation can be split or combined across terms:

$$\sum_{0 \le k \le N} (3k^2 + 8k - 10) = \sum_{0 \le k \le N} 3k^2 + \sum_{0 \le k \le N} 8k - \sum_{0 \le k \le N} 10$$

Distribution of multiplication over addition (pull out constants that do not depend on index variable):

$$\sum_{0 \le k \le N} 3k^2 + \sum_{0 \le k \le N} 8k - \sum_{0 \le k \le N} 10 = 3\sum_{0 \le k \le N} k^2 + 8\sum_{0 \le k \le N} k - \sum_{0 \le k \le N} 10$$

Note, if the sum does not include an index variable, we can just multiply it by the number of components (watch the index to get the number of components correct):

$$\sum_{0 \le k \le N} 10 = 10\sum_{0 \le k \le N} 1 = 10(N+1)$$

Change of Index Variable
Sometimes, a summation is easier to reason about if we change the index variable. For example, the original sum may start at 1 and it is easier to write starting with ~~1~~ 0:

$$\sum_{0<k<N}(k-1)^2x^k = \sum_{0\le k<N-1} k^2 \underline{x^{k+1}} = x\sum_{0\le k<N-1}k^2x^k$$

Study the above identity to ensure you see the steps.

Split into even/odd parts of a infinite sum

$$\sum_{k>0}(k+1)x^k = \sum_{k>0}\Big(\underbrace{\big((2k-1)+1\big)}_{\text{odd}}x^{2k-1} + \underbrace{(2k+1)}_{\text{even}}x^{2k} = \sum_{k>0}2kx^{2k-1} + \sum_{k>0}(2k+1)x^{2k}$$

2k-1 are odd numbers
2k are even numbers

$$= \left(\frac{2}{x}\right)\sum_{k>0}kx^{2k} + 2\sum_{k>0}kx^{2k} + \sum_{k>0}x^{2k}$$

$$x^{2k-1} = \frac{x^{2k}}{x}$$

What steps were applied?

Now, we can combine the first two summations because they have common parts:

$$= \left(\frac{2}{x}+2\right)\sum_{k>0}kx^{2k} + \sum_{k>0}x^{2k}$$

Practice 1: Are the following two sums equivalent? yes

$$\sum_{k>0}k^2x^k \quad = x + \sum_{k>1}k^2x^k$$

$$x + \sum_{k>1}k^2x^k$$

Practice 2: Can we reverse the order of summations? For example, are the two sums below equivalent? yes

$$\sum_{\cancel{k>0}\; n\ge 0}\ \sum_{0<k<n}\frac{1}{n^2+k^3}$$

$$\int\int$$

(n,k) look at pairs (n,k)

(n,k)	(n,k)
(2,1)	(2,1)
(3,1)	(3,1)
(3,2)	(4,1)
(4,1)	⋮
(4,2)	(3,2)
⋮	(4,2)
	⋮
	(4,3)
	(5,3)

$$\sum_{0<k<n} \sum_{\substack{\cancel{k>0} \\ k\geq 0}} \frac{1}{n^2+k^3}$$

This is like reversing the integrals in vector calculus.

Practice 3: Are the following sums equivalent? If so, try to do the algebra to get from one to the other. yes

$$\sum_{n\geq 0}\sum_{k\geq 0} x^{n+k}$$

$$\left(\sum_{n\geq 0} x^n\right)^2$$

Hints: 1) exponent identity, 2) factor out term not reliant on index variables, 3) products are independent, 4) rename index variable

$$\sum_{n\geq 0}\sum_{k\geq 0} x^{n+k} = \sum_{n\geq 0}\sum_{k\geq 0} x^n x^k = \left(\sum_{n\geq 0} x^n\right)\underbrace{\left(\sum_{k\geq 0} x^k\right)}_{\sum_{k\geq n} x^n}$$

Common Summations that You Should Know

Arithmetic series:

$$\sum_{0\leq k\leq n} k$$

$$\sum_{0\leq k<n} C_0 + C_1 k = \left(\frac{n}{2}\right)\left(2C_0 + C_1(n-1)\right) = C_0 n + \left(\frac{C_1 n(n-1)}{2}\right)$$

Geometric series (for |x| < 1):

$$\sum_{n\geq 0} x^n = \left(\frac{1}{1-x}\right)$$

General geometric series (substitute -x for x and -1 for r to get the above identity), based on the Taylor series of $(1+x)^r$.

$$\sum_{n\geq 0}\left(\frac{r(r-1)\dots(r-n+1)}{n!}\right)x^n = (1+x)^r$$

$\binom{n}{k} =$

n choose k

Binomial expansion: (note that the parentheses mean the "choose" operator)

$$\sum_{0\leq k\leq n}\binom{m+k}{k} = \binom{m}{0} + \binom{m+1}{1} + \dots + \binom{m+n}{n} = \binom{m+n+1}{n}$$

$$\sum_{n\geq 0}\binom{k+n}{n}x^n = \frac{1}{(1-x)^{k+1}}$$

$$\sum_{0\leq k\leq n}\binom{k}{m} = \binom{0}{m} + \binom{1}{m} + \dots + \binom{n}{m} = \binom{n+1}{m+1}$$

$$\sum_{\underline{k\geq 0}}\binom{n}{k}x^k y^{n-k} = \sum_{\underline{0\leq k\leq n}}\binom{n}{k}x^k y^{n-k} = (x+y)^n$$

Practice 4: Why is the finite sum (immediately above) equivalent to the infinite sum (immediately above)?

$\binom{4}{5} = ?$
$= 0$

Practice 5: Use the last sum under binomial expansion to determine the sum when x and y both equal 1.

$2^n = |\wp(\{1,2,\dots,n\})|$

Using Infinite Sums to Find Closed Form Solution for Finite Sum

Sometimes, we can simplify a finite sum by subtracting the infinite part of the sum that does not include the index variable. Assume |x| < 1.

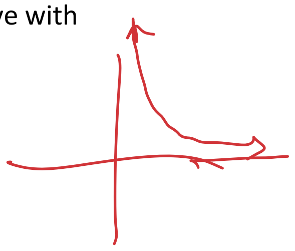
$$\sum_{0\leq k\leq n} x^k = \sum_{k\geq 0} x^k - \sum_{k>n} x^k \checkmark = \left(\sum_{k\geq 0} x^k\right) - \left(x^{n+1}\sum_{k\geq 0} x^k\right) = \left(\frac{1}{1-x}\right) - (x^{n+1})\left(\frac{1}{1-x}\right) = \frac{1-x^{n+1}}{1-x}$$

Check what happened:
1. Subtracted infinite part
2. Change of variable and factor out x^(n+1)
3. Use known closed form solution for geometric series

Be careful – some sums are divergent! They will add to infinity or negative infinity.

We can also approximate sums using integrals (see pages 1154-1155 in textbook):
If we have a monotonically increasing or a monotonically decreasing function, we can approximate sums using integrals. This is similar to how you learned calculus – estimating the area under the curve with rectangles.

$f(x) = \frac{1}{x}$

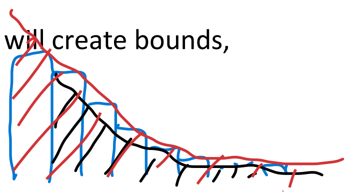


Let's look at the sum:

$$\sum_{1 \le k \le n} \left(\frac{1}{k}\right)$$

If we think of this as the function f(x) = 1/x, is f(x) *monotonically increasing* or *monotonically decreasing*?

Now, consider the integral from 1 to n+1 of f(x) and the integral from 0 to n. These will create bounds, but you need to know that the function is monotonic.



$$\int_1^{n+1} \left(\frac{1}{x}\right) dx \le \sum_{1 \le k \le n} \left(\frac{1}{k}\right) \le \int_0^{n} \left(\frac{1}{x}\right) dx$$

blue is the sum area
black is under curve from 1 to n+1
red is area under curve from 0 to n

Because the right integral starts at 0 and 1/x is undefined at 0, we can alter the sum to be 1 + the sum from 2 to n as indices:

$$\int_1^{n+1} \left(\frac{1}{x}\right) dx \le \sum_{1 \le k \le n} \left(\frac{1}{k}\right) = 1 + \sum_{2 \le k \le n} \left(\frac{1}{k}\right) \le 1 + \int_1^{n} \left(\frac{1}{x}\right) dx$$

We know from integrating (1/x)dx, this yields ln(n+1) on the left integral. The right integral evaluates to ln(n). So, we have the following bounds for the summation:

$$\ln(n+1) \le \sum_{1 \le k \le n} \left(\frac{1}{k}\right) \le 1 + \ln(n)$$

Inequalities can be helpful to bound terms and prove bounds.

Example: Prove that $\sum_{1 \le k \le n} k^{2.5} = \Omega(n^{3.5})$

For simplicity, let's call the summation T(n).

T(n) =

$$\sum_{1 \le k < \lceil \frac{n}{2} \rceil} k^{2.5} + \sum_{\lceil \frac{n}{2} \rceil \le k \le n} k^{2.5} \ge \sum_{\lceil \frac{n}{2} \rceil \le k \le n} k^{2.5} \ge \sum_{\lceil \frac{n}{2} \rceil \le k \le n} \left\lceil \frac{n}{2} \right\rceil^{2.5} = \left\lceil \frac{n}{2} \right\rceil^{2.5} \sum_{\lceil \frac{n}{2} \rceil \le k \le n} 1 \ge \left(\frac{n}{2}\right)^{2.5} \left(\frac{n-1}{2}\right)$$
$$= \frac{n^{3.5} - n^{2.5}}{2^{3.5}}$$

Now, the dominant term is n to the 3.5 and we have a lower bound of n to the 3.5.

CS324: Solving Recurrences, Master Method

January 27, 2020

To do before next class:

- HW2 due Wed, Feb 5 (9:15am to Moodle)
- Read chapter 4 (if not done earlier), 2.3, Appendix D.1

Today's learning outcomes:

- Review recurrence formation
- Proving runtime of recurrence functions via induction and a recursion tree
- Apply the master method for recurrences in certain formation

---

Often times we analyze algorithms (especially those that are recursive) by determining a recurrence function for the runtime.

lucky ☺

If we are lucky, we get an exact solution, such as $\sum_{1\le k\le n} k = \frac{n(n+1)}{2}$. This is Θ($n^2$).

But, most of the time, we just want to know the big-O or big-theta runtime of a recurrence.

There are <u>three</u> general techniques for solving recurrences:

1. Substitution/Induction (we have seen induction on recurrences for closed-form solutions already)
2. Recursion Tree
3. Master Method (only works for recurrences <u>in a specific form</u>)

Substitution/Induction Example To Prove Big-O

The recurrence for insertion sort looks like:

$O(n^2)$

$$T(n) = T(n-1) + n$$

Note: often times, we do not write the base case of the recurrence since asymptotic analysis concerns n becoming large. Base cases are done in constant work, so the recursive part of a recurrence dominates the runtime.

Prove that the (closed-form) solution of T(n) is O(n^2).

What do we need to do?

Since this concerns big-O, we need to find a constant C such that T(n) $\underline{<=}$ Cn^2 for n>=n_0. Then we prove by induction that the inequality holds.

<u>Proof:</u>

Let C = max(1, T(1)). Let n_0 = 1.

Base case:

Consider n=1. Since C = max(1, T(1)), T(1) <= C*1*1.

Induction hypothesis:

Assume that for integers m such that 1<=m<n, T(m) <= $\boxed{C}$$n^2$. Cm^2.

Induction step:

Consider T(n):

$$T(n) = T(n-1) + n$$

$$\le C(n-1)^2 + n \qquad \textit{by induction hypothesis}$$

$$= C(n^2 - 2n + 1) + n$$

$$= Cn^2 - 2Cn + C + n$$

$$= Cn^2 + \underbrace{\underline{(1-2C)}n}_{\le 0} + C$$

algebra math

$$\le Cn^2 \qquad \textit{since } (1-2C)n \textit{ is less than } C \textit{ for } n \ge 1 \textit{ and } C \ge 1$$

Conclusion:

By mathematical induction, T(n) is O(n^2).

Practice Example with Big-O: In small groups, complete the proof below to show that the solution to the following recurrence is O(lgn). Note that since the base case is not given, we will assume the base case is T(1) = 1.

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Proof:

We will show that T(n) <= 3lgn – 1. Note that it this case that n=1 does not adhere to the inequality, but since this is a big-O analysis, we just need to show it works for n>=n_0. Also, note that we are subtracting 1 from 3lgn, so if we show this inequality holds then we know that T(n) <= 3lgn.

Conjecture: T(n) <= 3lgn – 1 for n>=2. //Note: C=3 and n_0=2.

Base case:

Consider n = 2. T(2) = T(1)+1 = 2. 3*lg2 – 1 = 2. 2 <= 2, so it holds for n_0 = 2.

Induction hypothesis:

Assume for all integers m such that 2<=m<n, T(m) <= 3lg(m) – 1.

Induction step:

You complete this part:

Consider T(n):

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$\leq \left[3\lg\left(\frac{n}{2}\right) - 1\right] + 1$$

$$= 3\lg\left(\frac{n}{2}\right)$$

$$= 3\lg n - 3\lg 2$$

$$= 3\lg n - 3 \quad < 3\lg n - 1$$

Conclusion:

By mathematical induction, T(n) <= 3lgn-1 which implies that T(~~N~~ n) is O(lgn).

For which algorithm would the T(n) recurrence above be appropriate?

Merge sort
Linear search
[✓ Binary search]
Selection sort

Is T(n) $\Omega(\lg n)$?

yes.

Example To Prove Big-Omega:

Complete the proof below to show that the solution to the following recurrence is Ω(nlgn). Note that since the base case is not given, we will assume the base case is T(1) = 1.

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$

merge sort

Proof:

We will show that T(n) >= Cnlgn. ~~Note that it this case that n=1 does not adhere to the inequality, but since this is a big-O analysis, we just need to show it works for n>=n~~$_0$~~.~~

Conjecture: T(n) >= Cnlgn where C = 1/3. Note that we will keep the proof in terms of C and then show that C = 1/3 works for the induction step.

Base case:

Consider n = 1. T(1) = 1. C*1*lg1 = 0. Since 1 > 0, the base case holds.

Induction hypothesis:

Assume for all integers m such that 1<=m<n, T(m) >= C~~n~~lg~~n~~ where C = 1/3.

m Cmlgm

Induction step:

Let's complete this part:

Consider T(n):

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$

note: $\frac{n}{\frac{n}{n-1}} = n-1$

$\lg(n-1) = \lg(n) - \lg\left(\frac{n}{n-1}\right)$

$$\geq 2 \cdot C \cdot \lfloor \tfrac{n}{2} \rfloor \cdot \lg \lfloor \tfrac{n}{2} \rfloor + n$$
$$\geq 2 \cdot C\left(\tfrac{n-1}{2}\right) \cdot \lg\left(\tfrac{n-1}{2}\right) + n$$
$$= C(n-1) \cdot \lg\left(\tfrac{n-1}{2}\right) + n$$
$$= c(n-1)[\lg(n-1) - 1] + n$$
$$= [cn - c][\lg(n-1) - 1] + n$$
$$= cn[\lg(n-1)] - cn - c\lg(n-1) + c + n$$

$$= cn\left[\lg(n-1) - 1 + \tfrac{1}{c}\right] - c[\lg(n-1) - 1]$$
$$= cn\left[\lg(n) - \lg\left(\tfrac{n}{n-1}\right) - 1 + \tfrac{1}{c}\right]$$
$$- cn\left[\frac{\lg(n-1)}{n} - \frac{1}{n}\right]$$

Conclusion:

By mathematical induction on n>=1, T(n) >= (1/3)nlgn which implies that T(N) is Ω(nlgn).

$$= cn\left[\lg(n) - \underbrace{\lg\left(\tfrac{n}{n-1}\right)}_{0 \leq 1} - 1 + \tfrac{1}{c} - \underbrace{\frac{\lg(n-1) - 1}{n}}_{0 \leq 1}\right]$$

$\geq cn[\lg(n) - 1 - 1 + \frac{1}{c} - 1]$

$= cn[\lg(n) - 3 + \frac{1}{c}]$

$= cn\lg(n)$ when $c = \frac{1}{3}$

Check for understanding:

1. Does the constant C in an induction proof for asymptotic analysis need to be the same value throughout the proof? yes
2. Can you try the induction proof with a not-yet-known value for C to determine what it needs to be? yes
3. If the proof is big-omega, will the proof include T(n) <= C*runtime or T(n) >= C*runtime?
4. If the proof is big-theta, do you need two induction proofs, one for big-O and one for big-omega? yes

Practice with Big-O
The following recurrence is the runtime for the median-finding algorithm in chapter 9. Show the solution to the recurrence is O(n).

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n)$$

Base case: Since this is asymptotic analysis, we do not need to prove a base case since we are analyzing values of n away from 0.

☆ *(You can do this from now on for any big-O, big-theta, big-omega asymptotic proof. Note that if the proof asks for a closed form such at T(n) = n², then you must include the base case.)*

Induction hypothesis:
T(k) <= Ck for all k less than n where C is a constant.

Induction step:

Consider T(n):

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + Dn$$

Apply induction hypothesis: $\leq C \cdot \frac{n}{5} + C \cdot \frac{7n}{10} + Dn$

$= \frac{2cn}{10} + \frac{7cn}{10} + Dn$

Let C = 10*D. $= \frac{9cn}{10} + Dn \quad = \frac{9cn}{10} + \frac{c}{10}n$

Conclusion: By mathematical induction, T(n) = O(n).

$= \frac{10cn}{10} = cn$

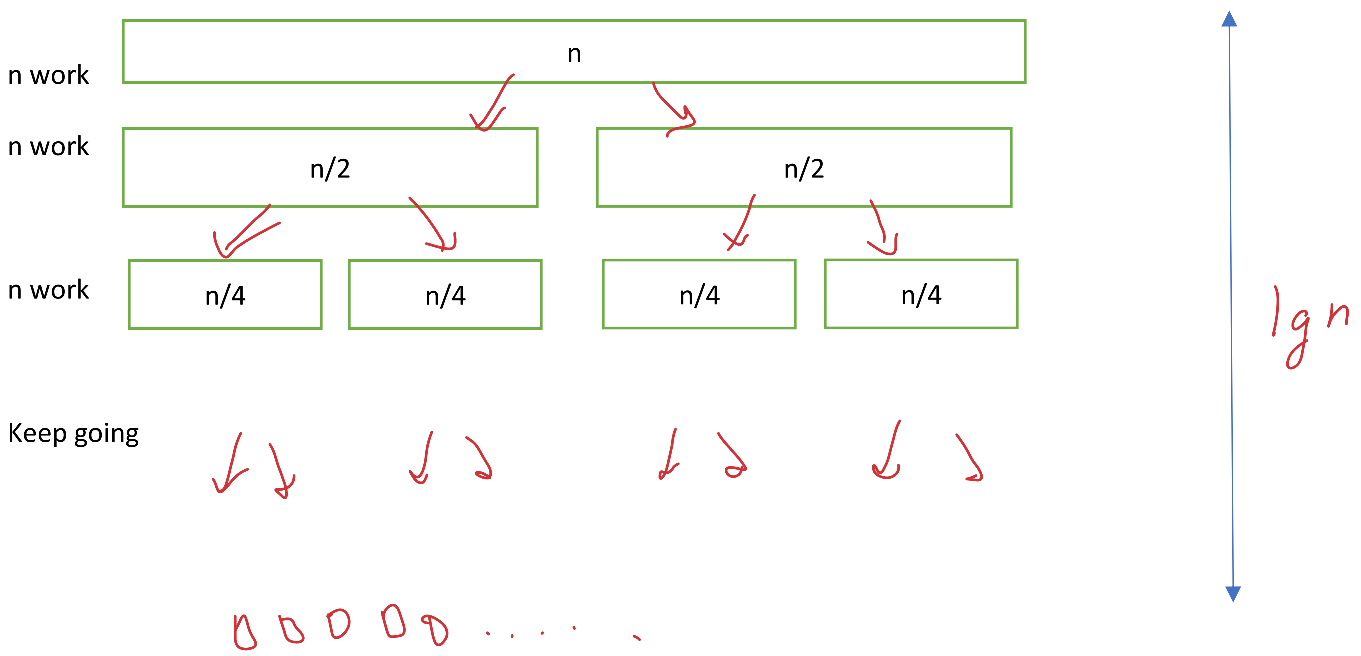
Some reminders on induction:

- Remember, you can assume n is greater than or equal to n_0 and n_0 can be set large, so the inequalities work.
- Sometimes, you want to subtract a value to make the induction work:
    - Example: Suppose you want to show that T(n) = 5T(n/2) + n = O($n^{\lg 5}$). Doing this straightforward of having the I.H. be T(k) <= $Ck^{\lg 5}$ won't work because we end up with an extra n term that we cannot get rid of in the math.
    - Instead, we can make the I.H. T(k) <= $Ck^{\lg 5}$ – Dk (for some constant D > 0). This will make the math work out so we can choose D to be >= 2/3. And, it still shows that T(n) is O($n^{\lg 5}$).

Recursion Tree Method

Let's look at merge sort. We can think of the work as the size of the recursion tree.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



How tall is the tree? lgn // there are that many splits

How much work at each level? n // copying into subarrays and merging

Master Method

Good news! We have a short-cut for solving recurrences when the recurrence divides a problem into a fixed number of equal-sized subproblems. The recurrence must have the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Let's digest this:

- The solution split into "a" problems of size "n/b".
- f(n) is the work in addition to the split
- n/b could have a floor or ceiling and we can treat it as (n/b) for the master method

Check: does the merge sort recurrence below satisfy the form for use of the master method:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

How many sub-problems? 2

What is the size of the sub-problems? $\frac{n}{2}$

What is f(n)? n

The proof of the master method is in the book in case you are interested. For this course, you do not need to reproduce the proof.

Check: does the following recurrence satisfy the form for use of the master method? no

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$$

Cases for the master theorem:

$n^{\log_b a} > f(n)$

$n^{\log_b a} = f(n)$

$n^{\log_b a} < f(n)$

1. If $f(n) = \omega(n^{\log_b a})$, then the complexity of T(n) is $\theta(f(n))$.
2. If $f(n) = \theta(n^{\log_b a})$, then the complexity of T(n) is $\theta(f(n)\log n)$.
3. If $f(n) = o(n^{\log_b a})$, then the complexity of T(n) is $\theta(n^{\log_b a})$.

Let's digest this:

1. We always look at $n^{\log_b a}$ and compare it to $f(n)$. One of the following cases (lower bound, theta, or upper bound must apply).
2. If one of f(n) or $n^{\log_b a}$ dominates, that is the complexity of the recurrence.
3. If f(n) and $n^{\log_b a}$ are equivalent asymptotically, then it is the common complexity with an additional factor of log(n).

Practice with the Master Method on given recurrences:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

T(n)	a	b	$n^{\log_b a}$	f(n)	Which dominates?	Θ
$3T\left(\frac{n}{2}\right) + n$	3	2	$n^\wedge(\log_2 3)$	n	$n^{\log_b a}$	$n^{\log_2 3}$
$3T\left(\frac{n}{2}\right) + n^2$	3	2	$n^\wedge(\log_2 3)$	n^2	n^2	n^2
$4T\left(\frac{n}{2}\right) + n^2$	4	2	n^2	n^2	equal	$n^2 \log n$
$T\left(\frac{n}{2}\right) + 1$	1	2	$n^{\log_2 1} = 1$	1	equal	$\log n$
$4T\left(\frac{n}{4}\right) + \log(n)$	4	4	$n^{\log_4 4} = n$	$\log n$	n	n
$24T\left(\frac{n}{5}\right) + n$	24	5	$n^{\log_5 24}$	n	$n^{\log_5 24}$	$n^{\log_5 24}$
$25T\left(\frac{n}{5}\right) + n^2$	25	5	$n^{\log_5 25} = n^2$	n^2	equal	$n^2 \log n$
$25T\left(\frac{n}{5}\right) + n^3 + n^{2.5}$	25	5	$n^{\log_5 25} = n^2$	n^3	n^3	n^3
$T\left(\frac{n}{20}\right) + 1$	1	20	$n^{\log_{20} 1} = n^0 = 1$	1	equal	$\log n$

Questions on the master method?

CS324: Matrix Multiplication and Strassen

January ~~29~~ 31 2020

To do before next class:

- HW2 due Wed, Feb 5 (9:15am to Moodle)
- Read chapter 27

Today's learning outcomes:

- Review divide-and-conquer algorithms
- Review matrix multiplication
  - Recurrence for regular divide-and-conquer matrix multiplication
- Strassen's algorithm for matrix multiplication
  - Recurrence for Strassen's algorithm

Divide-and-Conquer Algorithms split a problem into subproblems, which are each recursively solved.

Divide: divide into subproblems that are the same problem as the original, only with smaller input; typically, the smaller subproblems are approximately the same size

Conquer: conquer the subproblems by solving them recursively

Combine: combine the subproblem-solutions into a solution for the original program

What are algorithms that you have already seen the use a divide-and-conquer strategy?

merge sort $\Theta(n \lg n)$

quicksort $O(n^2)$

Do you know the big-theta or big-O runtimes of these algorithms?

One such problem for which we can use a divide-and-conquer strategy is **Matrix Multiplication**.

First, we will review how to multiply a matrix by a vector:

$$\overset{3\times3}{\begin{bmatrix}3 & 1 & -4\\ 8 & 0 & 2\\ -2 & 1 & -1\end{bmatrix}}\overset{3\times1}{\begin{bmatrix}5\\ 3\\ -3\end{bmatrix}} = \overset{3\times1}{\begin{bmatrix}?\\ ?\\ ?\end{bmatrix}} \begin{matrix}30\\ 34\\ -4\end{matrix}$$

(3 x 5) + (1 x 3) + (-4 x -3) = 15+3+12 = 30 // top value is inner-product of top row with vector

(8 x 5) + (0 x 3) + (2 x -3) = 40+0+(-6) = 34 // middle value

(-2 x 5) + (1 x 3) + (-1 x -3) = -10 + 3 +3 = -4 //bottom value

Review 1: Can we multiply a 3 x 3 matrix by a 2 x 2 matrix? NO

We can multiply any n x m (n rows, m columns) matrix by an m x p (m rows, p columns) matrix.

Review 2: Suppose we multiply a 4 x 2 matrix by a 2 x 3 matrix. What is the size of the product matrix?

yes 4x3

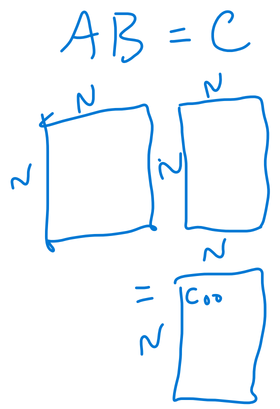
Practice: Let's try multiplying a 4 x 4 matrix with a 4 x 4 matrix. Complete the missing entries of the product matrix. The entry for the second row of matrix 1 and the third row of matrix 2 is done for you.

$$\begin{bmatrix}3 & 1 & -4 & 5\\ 8 & 0 & -1 & -3\\ 1 & 1 & 1 & 1\\ 0 & 0 & 1 & -1\end{bmatrix}\begin{bmatrix}7 & 4 & 16 & -7\\ -3 & -2 & 13 & -8\\ 10 & -5 & -9 & 12\\ 6 & 11 & 2 & 19\end{bmatrix} = \begin{bmatrix}8 & 85 & 107 & 18\\ 28 & 4 & 131 & -125\\ 20 & 8 & 22 & 16\\ 4 & -16 & -11 & -7\end{bmatrix}$$

$$21 - 3 - 40 + 30 = 8$$

Questions about how to multiply matrices?

Suppose we are multiplying A x B = C, where A, B, and C are N x N matrices. We can express the entry for each position as:



$$C_{ij} = \sum_{1 \le k \le N} A_{ik} B_{kj}$$

How many multiplications are needed for each sigma sum, in terms of N? ___N___

How many additions are needed for each sigma sum, in terms of N? ___$N-1$___

How many total C_{ij} terms are there, in terms of N? ___N^2___

Thus, the straightforward iterative solution with a triply nested loop runs in time Θ(___N^3___).

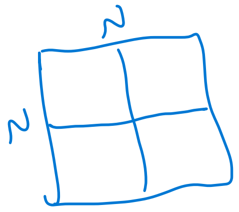
NOTE: For matrix multiplication, we typically assume a square matrix. If not, we pad the matrix with zeros to make it square. The square matrix has dimensions N x N, so one side of the matrix has N elements. In other problems, computer scientists use N as the size of the input. You can see that the data for the two matrices has size 2(N x N). This is the bizarre case of CS folks since they normally use N as the size of the input. Just remember in matrix multiplication, the N refers to the dimension of one side.

Thinking Recursively
We can think of matrix multiplication as dividing the matrix into subproblems like this:

$$\begin{bmatrix} \begin{bmatrix} 3 & 1 \\ 8 & 0 \end{bmatrix} & \begin{bmatrix} -4 & 5 \\ -1 & -3 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} 7 & 4 \\ -3 & -2 \end{bmatrix} & \begin{bmatrix} 16 & -7 \\ 13 & -8 \end{bmatrix} \\ \begin{bmatrix} 10 & -5 \\ 6 & 11 \end{bmatrix} & \begin{bmatrix} -9 & 12 \\ 2 & 19 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \quad \end{bmatrix} & \begin{bmatrix} 131 \end{bmatrix} \\ \begin{bmatrix} \quad \end{bmatrix} & \begin{bmatrix} \quad \end{bmatrix} \end{bmatrix}$$

We can then matrix-multiply 2 x 2 matrices and combine them by adding the 2 x 2 matrices into a 4 x 4 matrix.

In general, we can divide N x N matrix multiplication into several (N/2) x (N/2) matrix multiplications. If N is odd, we can just pad it with an extra row of zeros and an extra column of zeros.



The only tricky bit here is to figure out how to combine the half-sized matrices together to get the right values of the full matrix product.

$$\overset{A}{\begin{bmatrix} \overset{A}{\begin{bmatrix} 3 & 1 \\ 8 & 0 \end{bmatrix}} & \overset{B}{\begin{bmatrix} -4 & 5 \\ -1 & -3 \end{bmatrix}} \\ \underset{C}{\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}} & \underset{D}{\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}} \end{bmatrix}} \overset{B}{\begin{bmatrix} \overset{E}{\begin{bmatrix} 7 & 4 \\ -3 & -2 \end{bmatrix}} & \overset{F}{\begin{bmatrix} 16 & -7 \\ 13 & -8 \end{bmatrix}} \\ \underset{G}{\begin{bmatrix} 10 & -5 \\ 6 & 11 \end{bmatrix}} & \underset{H}{\begin{bmatrix} -9 & 12 \\ 2 & 19 \end{bmatrix}} \end{bmatrix}} = \overset{C}{\begin{bmatrix} \overset{U}{\begin{bmatrix} 8 & 85 \\ 28 & 4 \end{bmatrix}} & \begin{bmatrix} 107 & 18 \\ 131 & -125 \end{bmatrix} \\ \begin{bmatrix} 20 & 8 \\ 4 & -16 \end{bmatrix} & \begin{bmatrix} 22 & 16 \\ -11 & -7 \end{bmatrix} \end{bmatrix}}$$

$$\overset{A}{\begin{bmatrix} 3 & 1 \\ 8 & 0 \end{bmatrix}} \overset{E}{\begin{bmatrix} 7 & 4 \\ -3 & -2 \end{bmatrix}} = \begin{bmatrix} 18 & 10 \\ 56 & 32 \end{bmatrix}$$

$$\overset{B}{\begin{bmatrix} -4 & 5 \\ -1 & -3 \end{bmatrix}} \overset{G}{\begin{bmatrix} 10 & -5 \\ 6 & 11 \end{bmatrix}} = \begin{bmatrix} -10 & 75 \\ -28 & -28 \end{bmatrix}$$

$$\begin{bmatrix} 18 & 10 \\ 56 & 32 \end{bmatrix} + \begin{bmatrix} -10 & 75 \\ -28 & -28 \end{bmatrix} = \begin{bmatrix} 8 & 85 \\ 28 & 4 \end{bmatrix}$$

Let’s now add these results together:

$$\begin{bmatrix} 8 & 85 \\ 28 & 4 \end{bmatrix}$$

So, the top-left of matrix A multiplied by the top-left of matrix B plus the top-right of matrix A multiplied by the bottom-left of matrix B equals the top-left of matrix C.

In general, suppose each of the letters below represents an (N/2) x (N/2) matrix. Then the resulting matrix can be derived by adding two multiplied matrices.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} R & S \\ T & U \end{bmatrix}$$

The following matrix multiplications and additions give us R, S, T, and U:

R = AE + BG // we did this one above with the example
S = AF + BH
T = CE + DG
U = CF + DH

8 multiplications
4 additions

If we create a recursive divide-and-conquer algorithm with this approach, how much work is done?

8	(N/2) x (N/2) matrix multiplications
4	(N/2) x (N/2) matrix additions

It takes KxK additions to add two KxK matrices //work for the combine step

Therefore, the recurrence for the recursive matrix multiplication algorithm is:

$$T(n) = \underline{8T\left(\left\lceil \frac{N}{2} \right\rceil\right)} + 4\underline{\left(\frac{N}{2}\right)^2} = 8T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \theta(N^2)$$

Review: Does this recurrence satisfy the format for the Master Method? yes

Therefore, the solution for this recurrence is:

$$T(N) = \theta\left(N^{\lg 8}\right) = \theta(N^3)$$

Yikes, did we gain anything from the iterative, triply nested loop version? no

Strassen's Algorithm
Here comes Strassen (year 1969). Strassen was VERY clever in determining how all those subproblem matrix multiplications can be done in 7 multiplications and 18 additions instead of 8 multiplications and 4 additions.

Does that really help us? It seems like we are doing more work. However, the multiplication piece is the dominant work in the recurrence and 7 multiplications is better than 8!!!

Let's look at our submatrices again:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} R & S \\ T & U \end{bmatrix}$$

The following matrix multiplications and additions give us R, S, T, and U:

$$\left.\begin{aligned} R &= AE + BG \\ S &= AF + BH \\ T &= CE + DG \\ U &= CF + DH \end{aligned}\right\}\ \text{straitforward recursion}$$

Strassen's sub-computations, each labeled P_i. These will form the basis of getting to R, S, T, and U.

P1 = A(F-H) // 1 multiplication, 1 addition
P2 = (A+B)H // 1 multiplication, 1 addition
P3 = (C+D)E // 1 multiplication, 1 addition
P4 = D(G-E) // 1 multiplication, 1 addition
P5 = (A+D)(E+H) // 1 multiplication, 2 additions
P6 = (B-D)(G+H) // 1 multiplication, 2 additions
P7 = (A-C)(E+F) // 1 multiplication, 2 additions

// So far, we have 7 multiplications and 10 additions

$$\text{final matrix}\left\{\begin{aligned} R &= P5 + P4 - P2 + P6 && = AE+AH+DE+DH+DG-DE-AH-BH+BG+BH-DG-DH \quad ☆ \\ S &= P1 + P2 && = AF-AH+AH+BH \\ T &= P3 + P4 && = CE+DE+DG-DE \\ U &= P5 + P1 - P3 - P7 && = AE+AH+DE+DH+AF-AH-CE-DE-AE-AF+CE+CF \end{aligned}\right.$$

// This is another 8 additions

// We have a total of 7 multiplications and 18 additions

What is the recurrence for Strassen's Matrix Multiplication?

$$T(N) = 7T\left(\left\lceil\frac{N}{2}\right\rceil\right) + 18\left(\left\lceil\frac{N}{2}\right\rceil\right)^2 = 7T\left(\left\lceil\frac{N}{2}\right\rceil\right) + \theta(N^2)$$

Using the Master Method, we get:

$$N^{\log_2 7} \text{ vs } N^2$$

$$T(N) = \theta\left(N^{lg7}\right) \approx \theta(N^{2.81})$$

So, we have done better! Well, Strassen did better.

Some Takeaways:

- Before Strassen discovered this factorization, people may have thought that the lower bound of the matrix multiplication problem is $\Omega(N^3)$ since there are NxN entries that each need N multiplications. But, Strassen created a faster algorithm. This illustrates that computing lower bounds for problems is difficult.
- There is a large constant factor in Strassen (those extra additions), so in practice, the ordinary divide-and-conquer algorithm should be used when N < 45.
- Many matrices are sparse (lots of zeros), so there are faster algorithms for multiplication.
- Strassen's algorithm requires addition and subtraction, in addition to multiplication. Therefore, the values/operations must be a ring (in algebraic terms). For you, this means that addition must have an inverse operation, namely subtraction. For most purposes, the matrices you will come across will use numbers and the ordinary *, +, and – operations.
- Right now, matrix multiplication has an upper bound of about $O(N^{2.376})$ and the mathematical lower bound is $\Omega(N^2)$. We must enter data in for all N x N entries in the product matrix.

Practice: Use Strassen's intermediate Pi values and combinations to compute the product.

$$\begin{bmatrix} \overset{A}{\begin{bmatrix}3&1\\4&0\end{bmatrix}} & \overset{B}{\begin{bmatrix}0&-1\\-1&-3\end{bmatrix}} \\ \overset{C}{\begin{bmatrix}1&-1\\0&0\end{bmatrix}} & \overset{D}{\begin{bmatrix}1&1\\1&-1\end{bmatrix}} \end{bmatrix} \begin{bmatrix} \overset{E}{\begin{bmatrix}-1&4\\2&5\end{bmatrix}} & \overset{F}{\begin{bmatrix}3&-2\\0&1\end{bmatrix}} \\ \overset{G}{\begin{bmatrix}4&-1\\-4&5\end{bmatrix}} & \overset{H}{\begin{bmatrix}2&3\\2&1\end{bmatrix}} \end{bmatrix} = \begin{bmatrix} \overset{R}{\begin{bmatrix}\ &\ \\ \ &\ \end{bmatrix}} & \overset{S}{\begin{bmatrix}\ &\ \\ \ &\ \end{bmatrix}} \\ \overset{T}{\begin{bmatrix}\ &\ \\ \ &\ \end{bmatrix}} & \overset{U}{\begin{bmatrix}\ &\ \\ \ &\ \end{bmatrix}} \end{bmatrix}$$

Here's the answer for reference:

$$: \begin{bmatrix} 3 & 12 & 7 & -6 \\ 4 & 2 & 4 & -14 \\ -3 & 3 & 7 & 1 \\ 8 & 6 & 0 & 2 \end{bmatrix}$$

Use Strassen's:

P1 = A(F-H) = $\begin{bmatrix}3&1\\4&0\end{bmatrix}\begin{bmatrix}1&-5\\-2&0\end{bmatrix} = \begin{bmatrix}1&-15\\4&-20\end{bmatrix}$

P2 = (A+B)H $\begin{bmatrix}3&0\\3&-3\end{bmatrix}\begin{bmatrix}2&3\\2&1\end{bmatrix} = \begin{bmatrix}6&9\\0&6\end{bmatrix}$

P3 = (C+D)E $\begin{bmatrix}2&0\\1&-1\end{bmatrix}\begin{bmatrix}-1&4\\2&5\end{bmatrix} = \begin{bmatrix}-2&8\\-3&-1\end{bmatrix}$

P4 = D(G-E) $\begin{bmatrix}1&1\\1&-1\end{bmatrix}\begin{bmatrix}5&-5\\-6&0\end{bmatrix} = \begin{bmatrix}-1&-5\\11&-5\end{bmatrix}$

P5 = (A+D)(E+H) $\begin{bmatrix}4&2\\5&-1\end{bmatrix}\begin{bmatrix}1&7\\4&6\end{bmatrix} = \begin{bmatrix}12&40\\1&29\end{bmatrix}$

P6 = (B-D)(G+H) $\begin{bmatrix}-1&-2\\-2&-2\end{bmatrix}\begin{bmatrix}6&2\\-2&6\end{bmatrix} = \begin{bmatrix}-2&-14\\-8&-16\end{bmatrix}$

P7 = (A-C)(E+F) $\begin{bmatrix}2&2\\4&0\end{bmatrix}\begin{bmatrix}2&2\\2&6\end{bmatrix} = \begin{bmatrix}8&16\\8&8\end{bmatrix}$

R = P5 + P4 – P2 + P6 $\begin{bmatrix}3&12\\4&2\end{bmatrix}$

S = P1 + P2 $\begin{bmatrix}7&-6\\4&-14\end{bmatrix}$

T = P3 + P4 $\begin{bmatrix}-3&3\\8&6\end{bmatrix}$

U = P5 + P1 – P3 – P7 $\begin{bmatrix}7&1\\0&2\end{bmatrix}$

CS324: Analyzing Parallel Algorithms

February 3, 2020

To do before next class:

- HW2 due Wed, Feb 5 (9:15am to Moodle)
- Read Appendix B
- Study for exam on Fri, Feb 7 (see Moodle for practice exam and solutions)

Today's learning outcomes:

- Understand parallel algorithm pseudocode
- Apply algorithm analysis in terms of work and span to parallel algorithms
- Understand limitations of parallelization

Activity 1: Everyone think of a number between 1 and 10,000. Write it down and keep it secret.

______

1. Each of you is an element of an array. How can we determine if the array contains the value X?

2. Each of you is an element in an array. How can we use parallelism to determine if the array contains the value X?

3. What are some reasons why we would want to use parallelism (threads) in our code/algorithms?

OSes

graphics

security

4. Suppose we have 30 processors to determine if a 30-element array contains X. Can we get linear speedup from just doing linear search on a single processor? (For example, could I get N/N speedup?) Why or why not?

we cannot

If not, what are some of the costs for why we cannot achieve linear speedup?

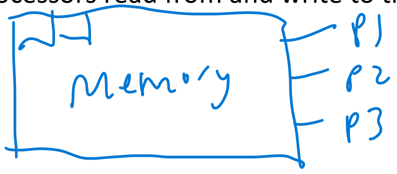
shared memory

thread spawning

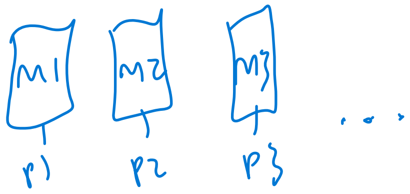
thread termination

Parallel Programming Models

1. Shared-memory architecture: all processors read from and write to the same (shared memory)
    Term: threads



2. Distributed-memory architecture: each processor has its own memory and data is shared through messaging
    Term: processes



Which do we use in this analysis course? We will assume <u>shared-memory</u> architecture. If you want to learn more about parallel computing, take the elective CS 436.

Using Parallelism to Find if Array A Contains X

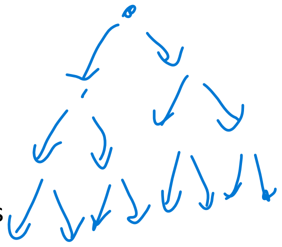
```
FIND_P(A, X):
1       Found = FALSE
2       Spawn A.length threads, each with unique ID number in {1, …, A.length}
3               If A[ID] == X
4                       Found = TRUE
5       Wait for threads to complete
6       Return Found
```

What about the analysis of this?

1 O(1)

2 ?? Well, it could take O(N) time to spawn N threads

?? We can actually spawn N threads in O(lgN) time be having one thread spawn 2 threads



3 ?? O(1) unless there is contention to read shared memory

4 ?? O(1) unless there is a queue of threads waiting to get access to this shared Found variable if all array positions contain X

5 ?? Could take up to O(N) time if Found is being accessed by every thread

?? Hardware to detect that all threads have completed

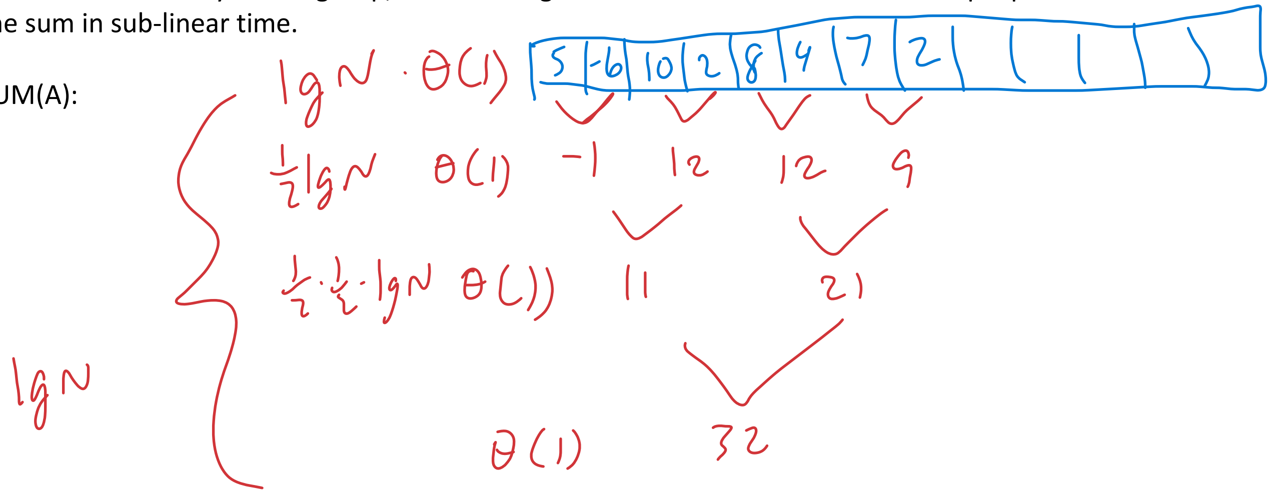
6 O(1)

Yikes, it is not as easy as it seems… this could actually take O(N) time which is the same time as ordinary linear search.

We will make some assumptions to make the analysis more consistent in a bit. For now, though, let's try to think in a parallel fashion.

Activity 2: You are given an array A of N numbers. You have an infinite number of processors and each processor can do a binary addition (add X + Y) in constant time. You want to determine the sum of all numbers in the array A. In a group, create an algorithm that uses threads and multiple processors to find the sum in sub-linear time.

SUM(A):



What is the running time analysis of this algorithm? (You can just explain it in words or a picture for now).

lg N

Additional keywords in our parallel pseudocode:

- **Parallel**: parallelize loop (includes creating the threads and executing loop iterations simultaneously in different threads)
    - Be careful – there could be race conditions and data dependencies
- **Spawn**: Spawns a new thread and runs it while main thread (or thread that spawned it) continues to run
- **Sync**: Wait for all threads that have been spawned to complete

Suppose we look at the recursive definition of FIB for calculating Fibonacci numbers. Note that this is a terrible way to do this (much better in a loop) but it will help us see how a recursive function can get parallelized and how we can compute metrics on the running time.

```
FIB_R(n):
    if n <= 1
        return n
    else
        x = FIB_R(n-1)
        y = FIB_R(n-2)
        return (x + y)
```

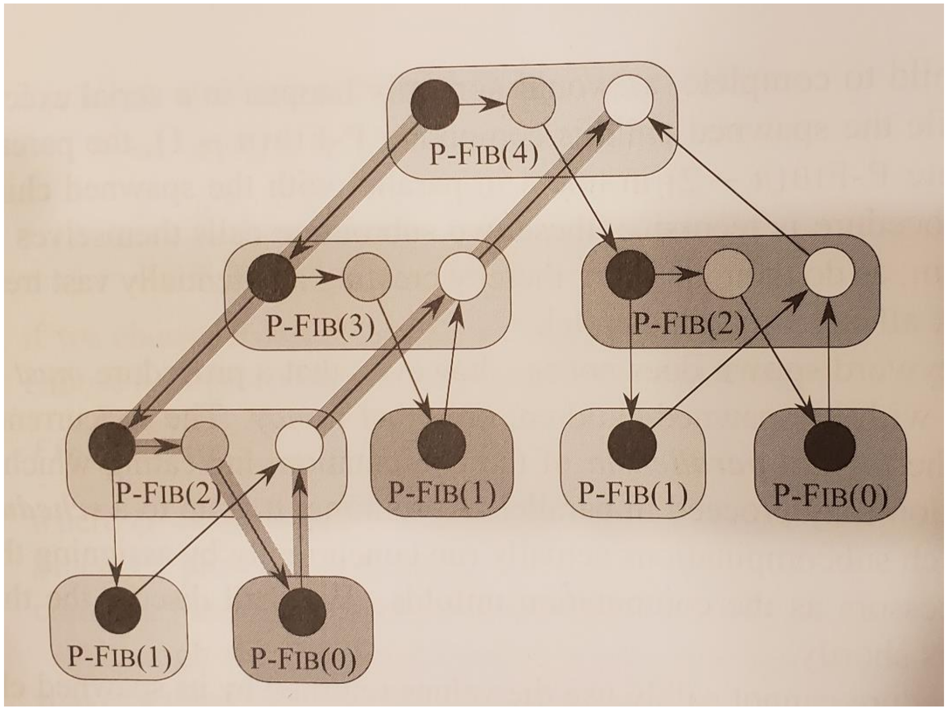
a. What is the recurrence for this non-parallel version? $T(n) = T(n-1) + T(n-2) + 1$

Turns out that we can solve this inductively and the solution is the golden ratio raised to the n'th power.

b. Where can we parallelize this FIB_R algorithm?

```
P-FIB_R(n):
1     if n <= 1
2          return n
3     else
4          x = spawn P-FIB_R(n-1)
5          y = P-FIB_R(n-2)
6          sync
7          return (x + y)
```

As a graph, here is how the computation proceeds:



● up to spawn

● lines 5 and 6

○ line 7

Figure 27.2 from CLRS: strand is a group of instructions, denoted by circles; they are grouped into procedures, donated by rounded rectangles. Spawn edges and call edges point downward. Continuation edges point to the right. Return edges point upward.

How many circles are there for P-FIB(4)? 17

If we assume each circle take unit time to complete, the **work** is equal to the number of circles.

How many circles are on the critical path of this DAG? 8

The **span** is the time to complete along the critical path.

The ratio of work / span is the **parallelism** of the computation.

What is the parallelism for P-FIB(4)? $\frac{17}{8}$ ~ 2.125

Word	**Notation**	**Meaning**
	P	Number of processors (concurrent maximum threads)
Work	T_1	The runtime of the algorithm on a single processor (no parallelization)
Span	T_∞	The runtime of the algorithm on an infinite set of processors; length of critical path
	T_P	The runtime of the algorithm on P processors
Parallelism	$\frac{T_1}{T_\infty}$	Average amount of work that can be performed in parallel with infinite processes; Maximum possible speedup
Speedup	$\frac{T_1}{T_P}$	How many times faster is the computation on P processors versus 1 processor
Slackness	$\frac{T_1}{PT_\infty}$	Factor by which the parallelism of the computation exceeds the number of processors in the machine

Check 1: Can the speedup be smaller than the parallelism? no

Check 2: Can the span be smaller than the work? yes

Check 3: Can T_P be smaller than the span? no

Check 4: Can T_P be smaller than (T_1 / P)? no

Check 3 is referred to as the **span law**.
Check 4 is referred to as the **work law**.

Check 5: What sorting routine would have a better runtime when using parallelism?

Insertion sort
Selection sort
Bubble sort
[✓ Merge sort]

Analyzing Parallel Algorithms

Some assumptions to make our analysis a bit easier:

- Parallel cost is never worse than the work (it could be in real life if the cost to set up the parallelism is large)
- We assume the thread scheduler is greedy and spawns threads immediately
- When two threads are running in parallel, they run together and we use the maximum of their runtimes up to the sync instead of the sum of the two runtimes.
- We can spawn N concurrent threads in logn time (main spawns A and B, A spawns C and D while B spawns E and F, and so on).

Parallel Merge Sort

```
Mergesort(A):
      <A1, A2> = split(A)
      x = spawn Mergesort(A1)
      y = Mergesort(A2)
      sync
      return merge(x, y)
```

Assume functions for split and merge exist. Split copies A into two arrays, roughly each of A.length/2, in linear time. Merge combines x and y together in linear time.

Draw process:

$$\begin{tikzcd}
\text{split} \arrow[d] \arrow[r] & \text{MS}(A1) \arrow[d] \\
\text{MS}(A2) \arrow[r] & \text{merge}
\end{tikzcd}$$

What is the work (T_1) of Mergesort? Remember, work is the cost of just running on a single processor.

$$T_1(n) = \theta(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \theta(n)$$
$$= 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \qquad = \Theta(n \lg n)$$

What is the span ($T\infty$) of Mergesort? Remember, span uses the max of the runtime of threads operating in parallel.

$$T_\infty(n) = \Theta(n) + \max\left(T\left(\frac{n}{2}\right), T\left(\frac{n}{2}\right)\right) + \Theta(n)$$
$$= T\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n) \text{ by M.M.} \qquad n^{\log_2 1} = n^0 = 1$$

Analyzing Parallel For Loops

First, a loop can only be parallelized if there are no race conditions (different results can happen if multiple threads access the same data in different orders!!).

BE CAREFUL when looking at loops that have the parallel keyword.

Assume A is a one-dimensional array of size N.

```
parallel for i = 1 to N
      A[i] = i*i
```

1. Can this loop be parallelized? ___yes___

2. What would be the span? ___$\Theta(\lg n)$___

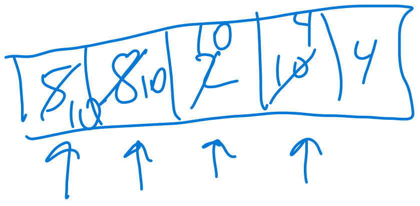
Again, assume A is a one-dimensional array of size N.

```
parallel for i = 1 to N-1
      A[i] = A[i+1]
```

* not good

3. Can this loop be parallelized? ___no___

4. What would be the span? ___$\Theta(n)$___



initialization of C

Parallel Matrix Multiplication

```
P-MAT_MULT(A, B, C):
1     parallel for i = 1 to N
2           parallel for j = 1 to N
3                 C[i][j] = 0
4     parallel for i = 1 to N
5           parallel for j = 1 to N
6                 for k = 1 to N
7                       C[i][j] = C[i][j] + A[i][k]*B[k][j]
```

cannot be parallel

T_1 = work

Let's analyze the work (one processor).

$T_{1,1}(n) = work\ of\ line\ 1\ array\ with\ dimensions\ n\ x\ n$ $= N \cdot T_{1,2}(n)$

$T_{1,2}(n) = N \cdot T_{1,3}(n)$

$T_{1,3}(n) = \Theta(1)$

$T_{1,4}(n) = N \cdot T_{1,5}(n)$

$T_{1,5}(n) = N \cdot T_{1,6}(n)$

$T_{1,6}(n) = N \cdot T_{1,7}(n)$

$T_{1,7}(n) = \Theta(1)$

$T_{1,1}(n) + T_{1,4}(n)$ = total time

$\Theta(n^2) + \Theta(n^3) = \Theta(n^3)$

Let's analyze the span (infinite processors).

$T_{\infty,1}(n) = work\ of\ line\ 1\ array\ with\ dimensions\ n\ x\ n$ $= \Theta(\log n) + T_{\infty,2}(n)$

$T_{\infty,2}(n) = \Theta(\log n) + T_{\infty,3}(n)$

$T_{\infty,3}(n) = \Theta(1)$

$T_{\infty,4}(n) = \Theta(\log n) + T_{\infty,5}(n)$

$T_{\infty,5}(n) = \Theta(\log n) + T_{\infty,6}(n)$

$T_{\infty,6}(n) = N \cdot T_{\infty,7}(n)$

$T_{\infty,7}(n) = \Theta(1)$

$\underline{T_{\infty,1}(n)} + \underline{T_{\infty,4}(n)}$

$= \underline{\Theta(\log n) + \Theta(\log n) + \Theta(1)}$

$+ \underline{\Theta(\log n) + \Theta(\log n) +}$

$\underline{N \cdot \Theta(1)}$

$= \Theta(N)$

Some reminders about limitations:

- We cannot spawn more threads than we have processors available; most of the analysis in this course assumes lots of processors.
- Remember we can spawn N threads in logN time.
- Be careful when analyzing parallel for loops – those with race conditions (using common data) cannot be parallelized if the order in which the threads operate could lead to different results.
- We assume a shared memory model. In the distributed memory model, there would be additional costs for sending messages.
- Synchronization has a cost that we assume is Θ(1).

CS324: Sets, Functions, Relations, Counting, Probability

February 5, 2020

To do before next class:

- HW3 assigned
- Study for exam on Fri, Feb 7 (see Moodle for practice exam and solutions)
- Read chapters 5 and 6 before Mon, Feb 10

Today's learning outcomes:

- Review discrete math
- Review probability

Today's lecture material should all be review, so you will be working through problems to review the material. If you have questions and/or clarifications, please ask.

Sets (review from MTH 311)

Notation review:
$\mathbb{Z}$ = integers
$\mathbb{N}$ = natural numbers (some mathematicians include 0 and some do not include 0)
$\mathbb{R}$ = real numbers
$\mathbb{C}$ = complex numbers

1. Is -3 in $\mathbb{Z}$? [✓ Yes] No
2. Is -3 in $\mathbb{N}$? Yes [✓ No]
3. Is 3.25 in $\mathbb{R}$? [✓ Yes] No
4. Is 3 + 25i in $\mathbb{R}$? Yes [✓ No]

5. Suppose S = {n ∈ $\mathbb{Z}$ | 4n + 3 < 100}. Is 25 in S? Yes [✓ No]

6. How do we write the empty set? $\emptyset$ $\{\}$

Suppose A and B are sets:

7. How do we write set **union** of A and B? $A \cup B$

8. How do we write set **intersection** of A and B? $A \cap B$

9. How do we write set **difference** (elements in A but not in B)? $A - B$

10. Suppose we have a universal set U consisting of all elements in the domain under consideration. For example, the universal set might be the set of integers. Suppose S = {n ∈ $\mathbb{Z}$ | 2d = n for some integer d}. What is the **complement** of S? odd integers

11. What is the complement of the universal set? $\emptyset$

12. What is $S \cap \bar{s}$? $\emptyset$

13. What is $\emptyset \cap S$? $\emptyset$

14. What is $\emptyset \cup S$? S

15. Does $\overline{A \cup B} = \bar{A} \cap \bar{B}$? yes

16. Does $\overline{A \cap B} = \bar{A} \cup \bar{B}$? yes

17. Suppose A and B are *disjoint*, what does that mean about the intersection?
$\emptyset$

18. Suppose A = {1, 2, 3, 4, 5}. What is |A| (**cardinality** of A)? 5

19. Suppose A is $\mathbb{Z}$. What is |A|? ∞ - countable

20. Suppose A is $\mathbb{R}$. What is |A|? ∞ - uncountable

21. An ordered pair (or ordered tuple) is written as x. Suppose A x B = C. Suppose A = {2, 4, 6}, and B = {1, 3}. Write the elements of A x B here:

$\{(2,1), (2,3), (4,1), (4,3), (6,1), (6,3)\}$

22. Suppose A is the set of even integers. Is A a **subset** of the integers? yes

23. Suppose A = {a, b, c}. Write the elements of the power set of A below: $|\mathcal{P}(A)| = 2^3 = 8$

$\{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$

Relations and Functions (review from MTH 311)

A **relation** on a set is a subset of the set's ordered pairs. For example, the relation "less than" on the set {0, 1, 2, 3} is the set:

{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}

Properties of a relation R:
Reflexive: (a, a) is in R for all a
Symmetric: (a, b) is in R iff (b, a) is in R for all a and for all b in R
Transitive: (a, b) and (b, c) in R implies that (a, c) is in R for all a, b, c in R

Connected: at least one of (a, b) or (b, a) is in R for all a, b in R
Equivalence relation: is reflexive, symmetric, and transitive (creates equivalence classes)
Partial order: is reflexive, antisymmetric, and transitive
Total order: is a partial order and is connected

24. Circle the properties for which the relation <= obeys over integers:

[✓ Reflexive]
Symmetric
[✓ Transitive]
[✓ Connected]
Equivalence
[✓ Partial order]
[✓ Total order]

25. Circle the properties for which the relation > obeys over integers:

Reflexive
Symmetric
[✓ Transitive]
[✓ Connected]
Equivalence
Partial order
Total order

A function from a set A into the set B is a binary relation on AxB for which each element of A appears exactly once. We often write f:A -> B or f(a) = b.

26. What is an *onto* function? for all $b \in B$, $\exists a \mid f(a) = b$
This is also called a **surjection**.

27. What is a *one-to-one* function? if $a_1 \neq a_2$ and $a_1, a_2 \in A$, $f(a_1) \neq f(a_2)$
This is also called an **injection.**

A function that is both a surjection and an injection is called a **bijection**.

28. Is the function f(n) = 2n surjective where the domain is the set of integers and the range is the set of integers? no

29. Give an example of a function from the integers to the integers that is a bijection:

F(n) = n

Counting (review from MTH 311 and EGR 361)

A lot of what we do in computer science is count things. We might count the possible ways an array's contents can be ordered. We might count how many distinct binary trees there are with N nodes. We might count how many ways we can re-arrange assembly instructions to send them to a processor. We might count how many ways we can choose a random sample of 10 test cases from 200 test cases.

NcK is **N choose K**, meaning how many ways can we choose K items from a set of N items.

$$\binom{N}{K} = \frac{N!}{K!\,(N - K!)}$$

30. How many ways are there to choose 10 balls from a set of 15 balls? You can leave your answer with factorials if you want. ______________________ $\binom{15}{10} = \frac{15!}{10!\,5!}$

A **permutation** is an ordering of elements within a set. If the set has N elements, there are N! permutations.

31. Suppose we have 20 students in the class. How many different ways can the students line up single-file? ____20!____

Here are some useful bounds for the choose operation:

$$\left(\frac{N}{K}\right)^K \le \binom{N}{K} \le \left(\frac{eN}{K}\right)^K$$

Probability (review from EGR 361)

Classic probability revolves around determining the probability of an event (outcome) taking place. The sample space includes all possible outcomes. Let's look at a 6-sided die. There are 6 outcomes {1, 2, 3, 4, 5, 6}.

32. What is the probability of rolling a 6 with one die? ____$\frac{1}{6}$____

33. What is the probability of rolling an odd number with one die (sum of the probabilities of the 3 outcomes)? _________________ $\frac{1}{2}$

Events A and B are independent if the Pr(A ∩ B) = Pr(A) * Pr(B). In other words one of them occurring does not impact the chance of the other occurring.

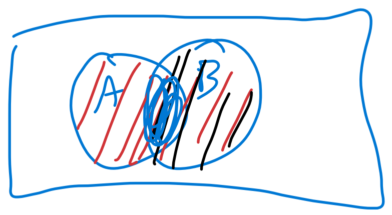
34. Are the following events independent or not? Roll a 3 with the first die and roll a 5 with the second die.

35. Are the following events independent or not? Roll a 5 with the first die and roll an odd number with the first die?

Conditional probability is the probability of B happening given that A happens. We can use this in Bayes Theorem to calculate the intersection of the probabilities.

Pr(A ∩ B) = Pr(A)*Pr(B|A) = Pr(B)*Pr(A|B)

Axioms of Probability



1. Axiom 1: sum of all outcomes is equal to 1.
2. Axiom 2: 0 <= P(A) <= 1 for any event A.
3. Pr(A U B) = Pr(A) + Pr(B) – Pr(A ∩ B). // Draw Venn Diagram to convince yourself

Random variables represent the outcome of a probabilistic event. Its value is not fixed, but has a range of values with certain probabilities.

Let's use dice. Let X represent the value of a die.

36. What is P(X>3)? $\frac{1}{2}$

37. What is P(X >= 3)? $\frac{2}{3}$

The **mean** is the expected value of a random variable – the average of the values that it will have. It does not mean a value that we expect (as in the most probable outcome). It really is a weighted average.

$$E[X] = \sum x * \Pr(x = X)$$

Note that for continuous probability distributions, we would integrate x*f(x).

What is the expected value of one dice roll?

E[X] = 1*Pr(x=1) + 2*Pr(x=2) + 3*Pr(x=3) + 4*Pr(x=4) + 5*Pr(x=5) + 6*Pr(x=6)

Since the Pr(x=1) = (1/6) and so do the rest, we get:

E[X] = 1*(1/6) + 2*(1/6) + 3*(1/6) + 4*(1/6) + 5*(1/6) + 6*(1/6) = 3.5

38. Suppose we roll two dice. What is the expected value of the sum of the two dice? 7.0

The **standard deviation** of a random variable gives a measure of how much the random variable deviates from the mean. If the standard deviation is 0, then the random variable always takes on the mean (expected) value. **Variance** is the square of the standard deviation and can be calculated in two, equivalent ways:

$$Var[X] = E[(X - E[X])^2]$$

$$Var[X] = E[X^2] - (E[X])^2$$

} equivalent

The second formulation has fewer differences that must be calculated, so it is usually faster to use the second formulation. The standard deviation is the square root of the variance, so that the units for standard deviation align with the units of the random variable.

Let’s see what the variance of a die roll is:

Var[X] = E[X^2] – (E[X])^2 = [(1 + 4 + 9 + 16 + 25 + 36)*(1/6)] – (3.5)*(3.5) = 2.916667.

Thus, the standard deviation is the square root of 2.916667 which is 1.7078.

Theorems:
When two random variables are independent, the variance of the sum is the sum of the variance.

When two random variables are independent, the expected value of the sum is the sum of the expected values.

More Practice:

39. Suppose I have an array of N integers. I randomly choose 5 of the N. How many different ways can I choose 5 of N integers?

$$\binom{N}{5}$$

40. What is the probability that the chosen 5 will be in sorted order?

$$\frac{1}{5!}$$

41. What is the probability that one of the chosen 5 is the largest element in the array?

$$\frac{5}{N}$$

42. What is the probability that a 5-card hand from a regular 52-card deck has cards all of the same suit?

$$1 * \left(\frac{12}{51}\right) * \left(\frac{11}{50}\right) * \left(\frac{10}{49}\right) * \left(\frac{9}{48}\right) = .002$$

CS324: Probability Distributions

February 10, 2020

To do before next class:

- HW3 due Wednesday, Feb 19
- Read chapter 7 before Wednesday

Today's learning outcomes:

- Review probability distributions, both discrete and continuous

---

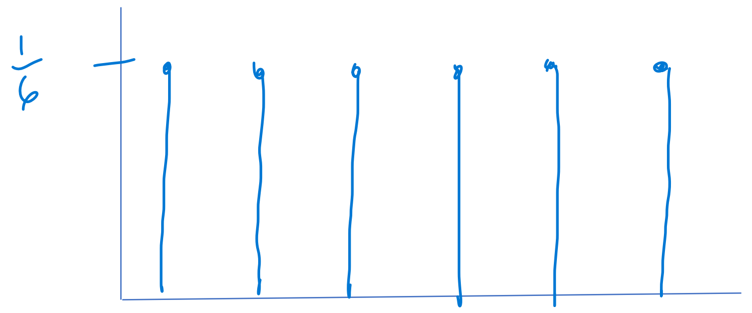
1. What is a probability distribution? graph of likely outcomes

2. What do you remember about being true about probability distributions?

$\sum$ outcomes $= 1$

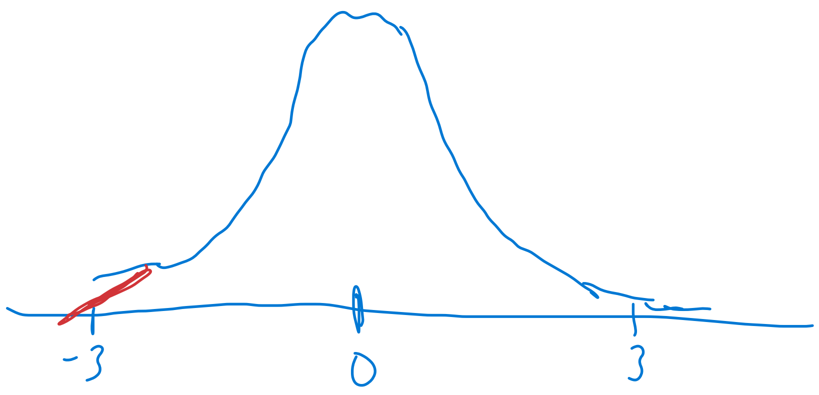
$0 \le P(\text{event}) \le 1$

3. What does the probability distribution look like for rolling dice?



1 2 3 4 5 6

4. Draw the standard normal distribution below.



Axioms:

1. Does the sum of the probabilities for all events equal 1? yes

2. Does 0 <= P(x) <= 1 for all x in S, where S is the set of all events? yes

5. What is the difference between discrete and continuous probability distributions?

Discrete: events can be numbered (counted); each event has a probability associated with it.

Example: The probability that the coin lands "heads" is 0.5.
Example: The probability for rolling the 1 on a 6-sided die is 1/6.
Example: The probability that the sum of 3 dice rolls is 3 is 1/216.

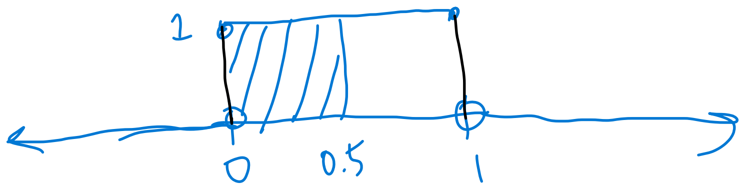
Continuous: events cannot be numbered, but map to the real numbers; events are defined by a probability density function f(x). Integral of f(x)dx from negative infinity to infinity is equal to 1. We no longer think of the probability of x=5, but now think of the probability of x being in a range, such as Pr(4 <=x<=5).

Example:

$$f(x) = 1, \quad 0 \le x \le 1$$
$$f(x) = 0, \quad x < 0 \; or \; x > 1$$

What is the Pr(0<=x<= 0.5) for this probability density function? 0.5

Draw this:



Examples of distributions

Discrete

Uniform	Binomial
Poisson	Geometric

Continuous

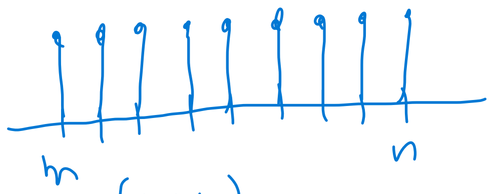
Uniform	Normal
Exponential	Approximate Binomial

We will now look at each of these distributions and why they are useful for modeling problems.

Uniform (discrete)
Each event has equal probability.

$\Pr(X = k) = \frac{1}{n-m+1}$ for m<=k<=n, Else 0

Draw it:



What is the mean? $\frac{(n+m)}{2}$

What is the variance? $\frac{(\overset{n}{\cancel{m}}-\overset{m}{\cancel{n}}-1)^2-1}{12}$

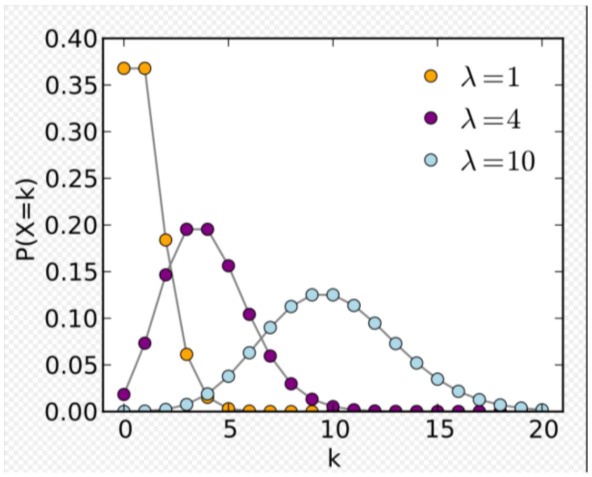
What is an example situation for which the discrete uniform distribution is an appropriate model?

dice
random element in array

Poisson (discrete)
How many occurrences of an event in one time unit where event happens randomly at a rate of λ per time unit.

$\Pr(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$ For k=0, 1, 2, 3, …

Draw it:



What is the mean? λ

What is the variance? λ

What is an example situation for which the Poisson distribution is an appropriate model?

Example: There are 200 errors in 500 pages of code and the errors follow a Poisson process. What is the probability that a given page has exactly three errors?

$$\lambda = \frac{200}{500} = 0.4\, errors\ per\ page$$

We want to find the probability that X = 3 using the Poisson distribution:

$$P(X = 3, \lambda = 0.4) = \frac{e^{-0.4}0.4^3}{3!} = 0.0072$$

So, there's a less than 1% chance that there are exactly 3 errors on the page. Does this make sense given there are 200 errors in 500 pages?

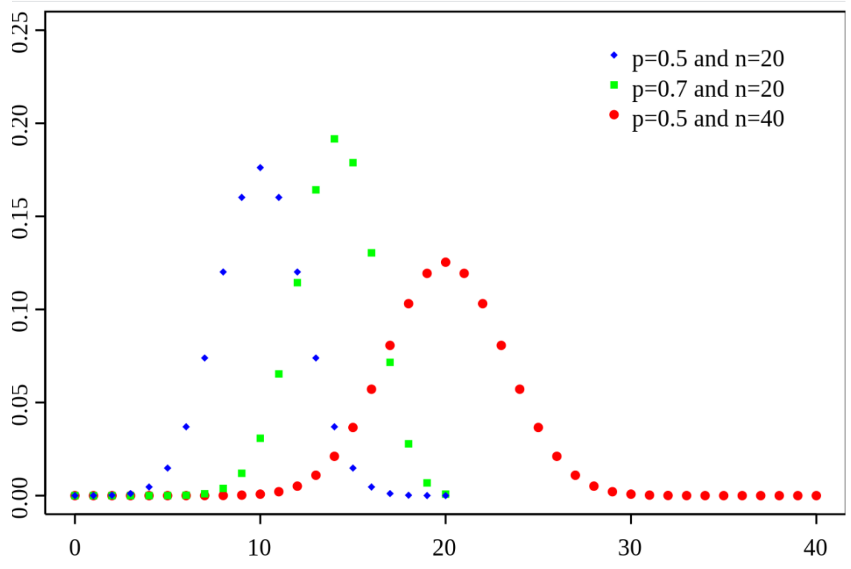
Binomial (discrete)
N events occur in succession, each with a success-probability of p and a failure probability of (1-p). This is called binomial since there are exactly two outcomes.

$\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$ for k = 0, 1, 2, 3, …

p = prob of success

$0 \le p \le 1$

Draw it:



What is the mean? np

What is the variance? np(1-p)

What is an example situation for which the Binomial distribution is an appropriate model?

Example: What is the probability that out of 100 coin flips, exactly 52 land heads?

Example: Suppose Harry's free throw percentage is 0.8. What is the probability that he makes 9 of 9 free throws?

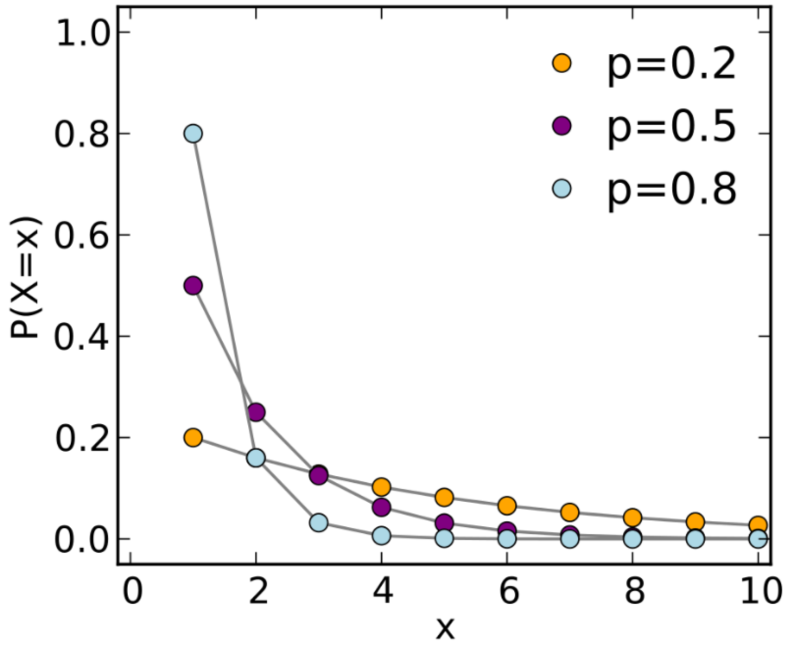
$$P(X=9) = \binom{9}{9} \underset{p}{(.8)^9} \underset{(1-p)}{(.2)^0} = 1 * (.8)^9 * 1 = .1342$$

Geometric (discrete)

Probability that the first success of a sequence of Bernoulli trials (each with success probability p) is the kth one.

$\Pr(X=k) = p(1-p)^{k-1}$ for k>0, else 0

Draw it:



What is the mean? 1/p

What is the variance? (1-p) / (p^2)

What is an example situation for which the Geometric distribution is an appropriate model?

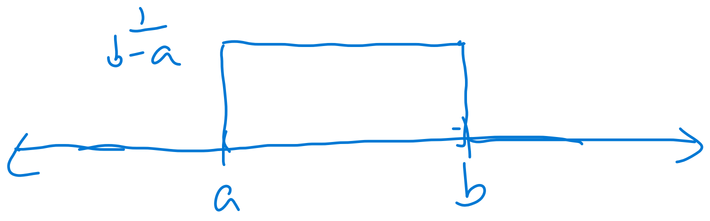
Example: What is the probability that it takes 4 flips to get first heads?

$$\Pr(X=4) = 0.5(1-0.5)^3 = 0.0625$$

Uniform
~~Discrete~~ (continuous)
In a range from a to b, f(x) is equal.

$f(x) = \frac{1}{b-a}$ for a<=x<=b; else 0

Draw it:

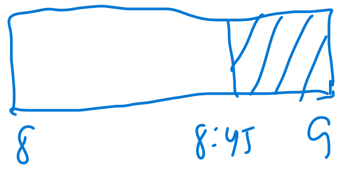


What is the mean? $\frac{a+b}{2}$

What is the variance? $(b-a)^2 / 12$

What is an example situation for which the continuous uniform distribution is an appropriate model?
Example: What is the probability that Jake will come into the office between 8:45 and 9:00am, assuming it is equally likely that Jake comes in with equal probability between 8 and 9 am?

Answer: 0.25
If we model the hour from 0 to 1, so f(x) = 1 for 0<=x<=1. Then we take the integral of 1dx from .75 to 1.



Exponential (continuous)
Time until a next random occurrence where they occur at the rate of λ per time unit. Note this is related to the Poisson (discrete) distribution. Poisson asks how many occur versus the exponential is when does the next occur.

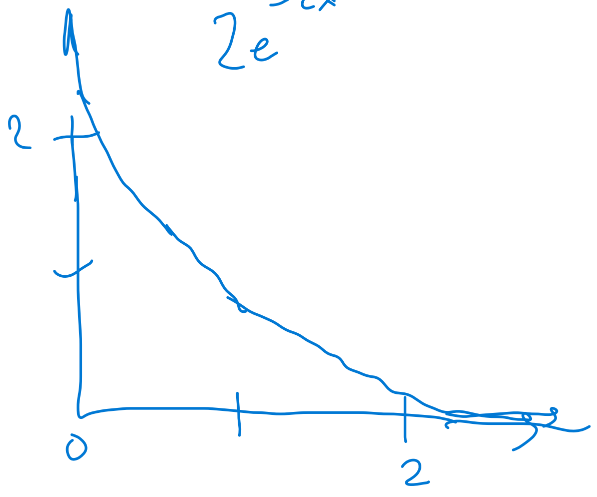
$f(x) = \lambda e^{-\lambda x}$ for x>= 0; else 0

What is the mean? 1/λ

What is the variance? $1/\lambda^2$

Draw it for lambda = 2:

$2e^{-2x}$



What is an example situation for which the exponential distribution is an appropriate model?

Example: When will the next transistor fail given that they fail at a rate of 1 per day.

Example: The call times to customer support is exponentially distributed with a mean time between calls of 12 minutes. What is the probability that there are no calls within a 30-minute interval?

Let X denote the time until the first call. X is exponential and λ = 1/E(X). So λ = 1/12 calls/minute. We need to determine the probability that X is greater than 30 (no calls within 30-minute interval):

$$P(X > 30) = \int_{30}^{\infty} \frac{1}{12}\left(e^{\frac{-x}{12}}\right) dx = -e^{\frac{-x}{12}}\big|_{30}^{\infty} = 0 - (-e^{-2.5}) = e^{-2.5} = 0.0821$$

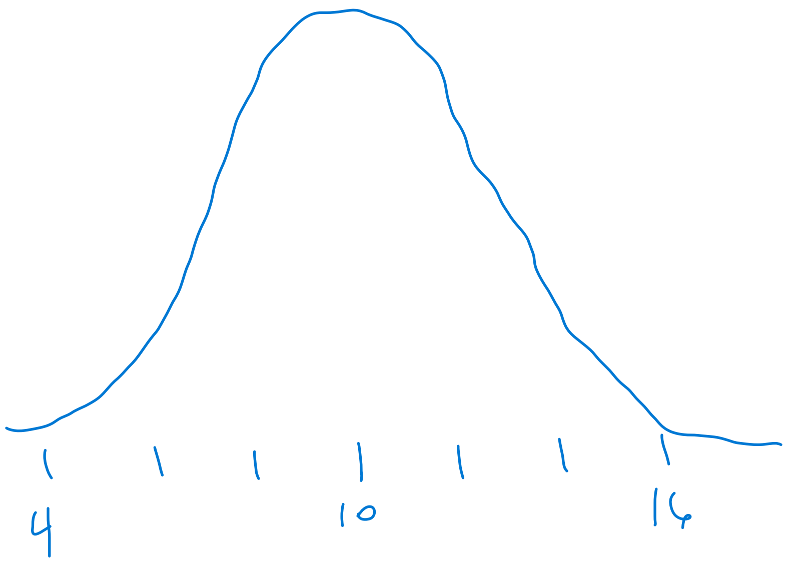
Normal (continuous)

This is the bell curve distribution for which you are likely familiar. It can model the distribution of naturally occurring phenomena, such as the height of people in the US.

$$f(x) = \frac{e^{\frac{-(x-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}$$ with mean μ and standard deviation σ

If we convert to standard normal, this distribution has mean 0 and standard deviation of 1. This is really ugly to integrate, so there are tables to approximate the probabilities after converting to standard normal.

Draw the normal distribution with mean 10 and standard deviation of 2:



The mean and standard deviation are embedded in the function definition.

What is an example situation for which the normal distribution is an appropriate model?

Example: A robot fills juice bottles such that the mean fill is 10 ounces with a standard deviation of 0.05 ounces. What is the probability that a bottle has fewer than 9.9 ounces?

You likely did lots of these types of problems in EGR 361.

Approximate Binomial (continuous)

The binomial distribution works for calculating probabilities when N is small. However, in asymptotic analysis, we analyze for large N. It turns out that the discrete binomial distribution can be approximated by the standard normal (mean 0, standard deviation of 1) distribution for large N.

p is the probability of success for a single event

$$\mu = np$$
$$\sigma^2 = np(1-p)$$

$$Z = \frac{X - np}{\sqrt{(np(1-p)}}$$

What is an example situation for which the approximate binomial distribution is an appropriate model?

Example: What is the probability of getting at least 60 heads in 100 coin flips?

Solve for Z:

$$Z = \frac{60 - 100 * .05}{\sqrt{100(.05)(.05)}}$$

$$Z = 2$$

When looking where 2 is on the standard normal distribution, we find that 2 corresponds to probability 0.9772. Since we are looking for the probability >= 60, we want to area of the curve to the right of 2. This is 1 – 0.9772 = 0.0228.

Why is probability important in the analysis of algorithms?

- Testing -- finding bugs
- Probability of getting a collision in a size N hash table with M items?
- Probability of hash table chains reaching a length of 10?
- Expected time for quicksort with randomly-arranged data
- Expected depth of binary search tree with random insertions
- And others…

Let's Practice Using Distributions:

1. Suppose a manufacturing process has 100 customer orders to fill. Each order requires one component that is purchased from a supplier. However, 2% of the components are identified as defective and the components can be assumed to be independent.

   a) If the manufacturer stocks 100 components, what is the probability that the 100 orders can be filled without reordering components?

   binomial

   $$Pr(X=0) = \binom{100}{0} \cdot (.02)^0 \cdot (.98)^{100}$$

   $$= 1 \cdot 1 \cdot .98^{100}$$

   $$= .133$$

   b) If the manufacturer stocks 102 components, what is the probability that the 100 orders can be filled without reordering?

   c) If the manufacturer stocks 105 components, what is the probability that the 100 orders can be filled without reordering?

2. Messages arrive to a computer server according to a Poisson distribution with a mean rate of 10 messages per hour.

a) What is the probability that three messages will arrive in an hour?

$\lambda = 10$ messages/hour

$k = 3$

$$Pr(X=3) = \frac{e^{-10} \cdot 10^{3}}{3!} = .00757$$

b) What is the probability that six messages arrive in 30 minutes?

Hint: remember to update the mean for this unit of time.

c) Do these probabilities make sense?

CS324: Heaps and Heapsort

February 12, 2020

To do before next class:

- HW3 due Wednesday, Feb 19
- Read chapter 7 before Wednesday (should have also read chapter 6 before today)

Today's learning outcomes:

- Understand the heap data structure and the heap property
- Insert into a heap
- Heapify
- Remove smallest/largest from heap
- Sort using a heap

Once again, what is the runtime of merge sort of N elements? $\Theta(N \lg N)$

Does merge sort do its merge in place? No, needs another array for merged items to be copied into.

Does insertion sort do its sorting in place? Yes.

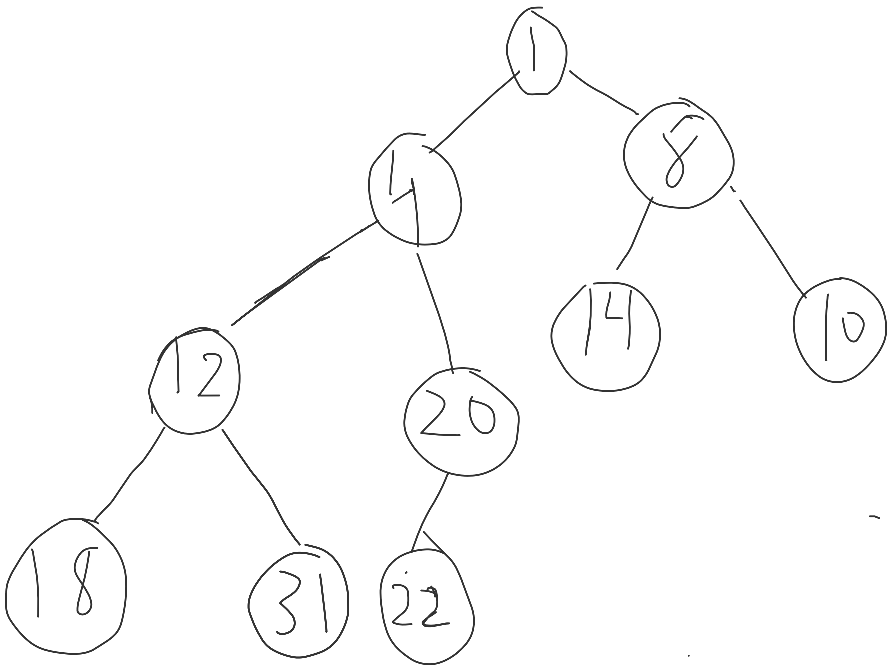
Well, heapsort gives us the best of both worlds:

- NlgN runtime
- No additional array for sorting; everything can be done in a single array

Note: heap for heapsort is a data structure. This is not to be confused with heap storage in Java or C.

What is a heap?

A heap is a nearly complete binary tree. Each element (key) is one node in the binary tree. Look at the heap below and answer the questions that follow.



1. Is the heap a binary search tree? no

2. If not, what do you see that violates the binary search tree property? 14 < 8 ?

3. What assertion can you make about the arrangement of nodes in a heap?

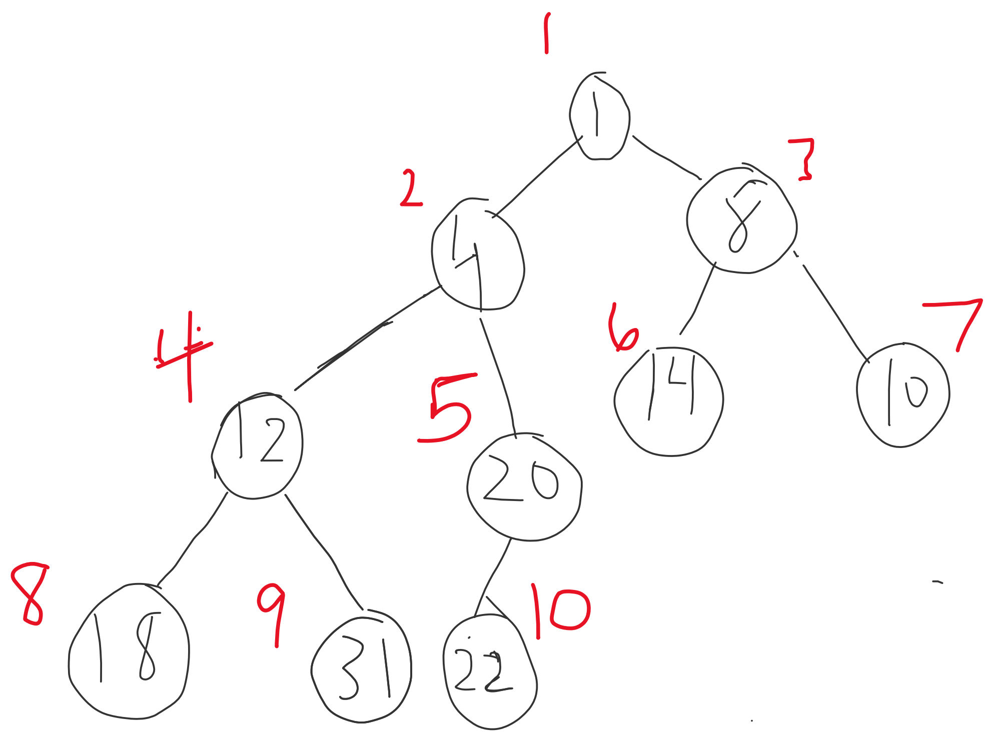
parent $\leq$ children

4. What do you notice about the leaves? generally largest, left-justified

5. Where is the minimum element (key) in the heap? root

6. Where are the largest elements? leaves

Now, we will put the heap into an array with indices 1 to 10, where the root is at position 1. The indices are in red.



Array: [1, 4, 8, 12, 20, 14, 10, 18, 31, 22]

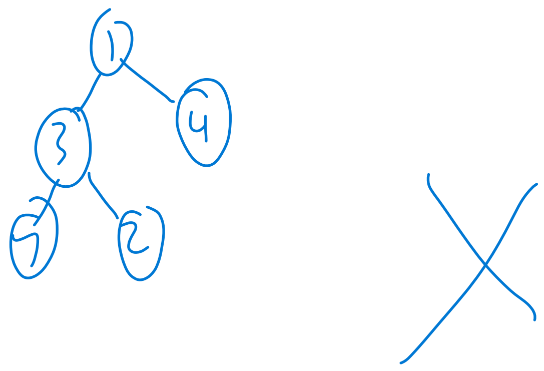
7. What is the relationship between the parent and its children in terms of the array indices:

Parent index is i

Left child index is: 2i

Right child index is: 2i + 1

8. Draw a heap that is NOT valid below.



We have been making heaps that satisfy the MIN-HEAP property. Each parent is less than or equal to its children. The textbook creates heaps with the MAX-HEAP property, so you have two examples to look at. Throughout lecture, we will stick to min heaps. Both types of heaps can be used for sorting (increasing or decreasing).

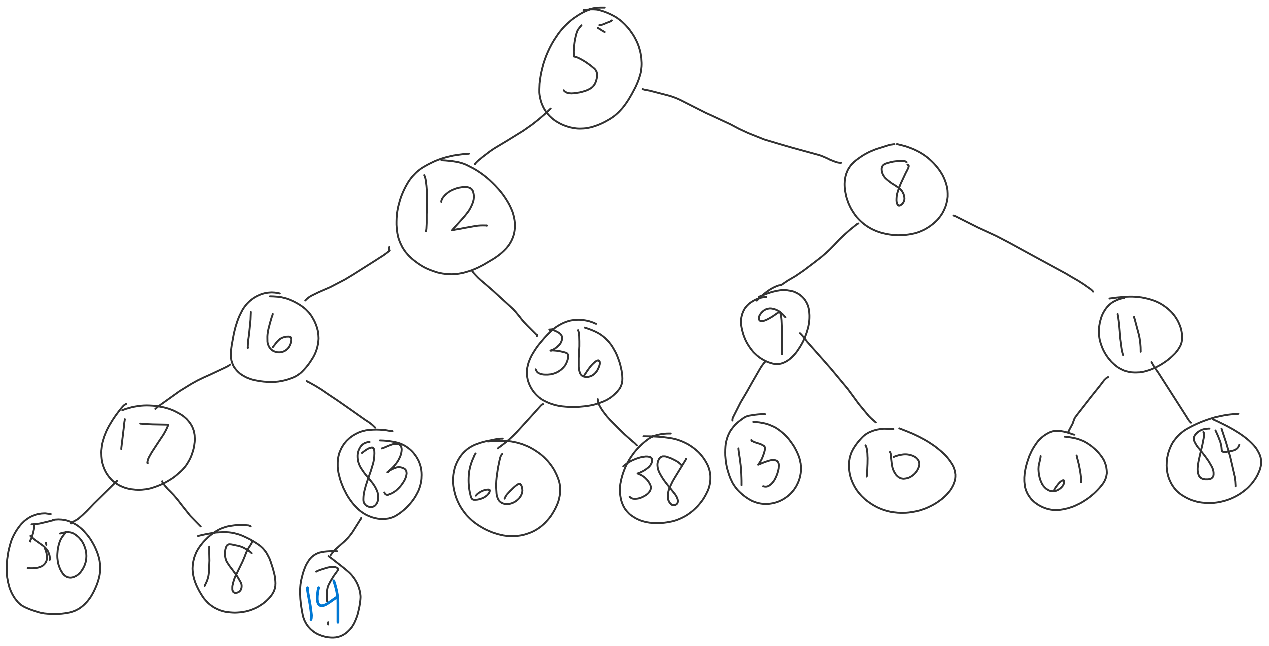
A heap acts as a priority queue:

- We can **insert** elements in random order
- We can **"heapify"** (turning tree into a heap)
- We can **remove** elements in priority order from a heap, which puts them in sorted order

So, in the next few pages, we will focus on the three algorithms. We will consider the heap as a tree in our examples. Note that the textbook includes all the pseudocode using the array representation, so you can look there for the details.

Inserting into a heap

Suppose we have the following min-heap. The next slot in the array representation is where the ? is located. Note that heaps are always filled from the left on the bottom level.

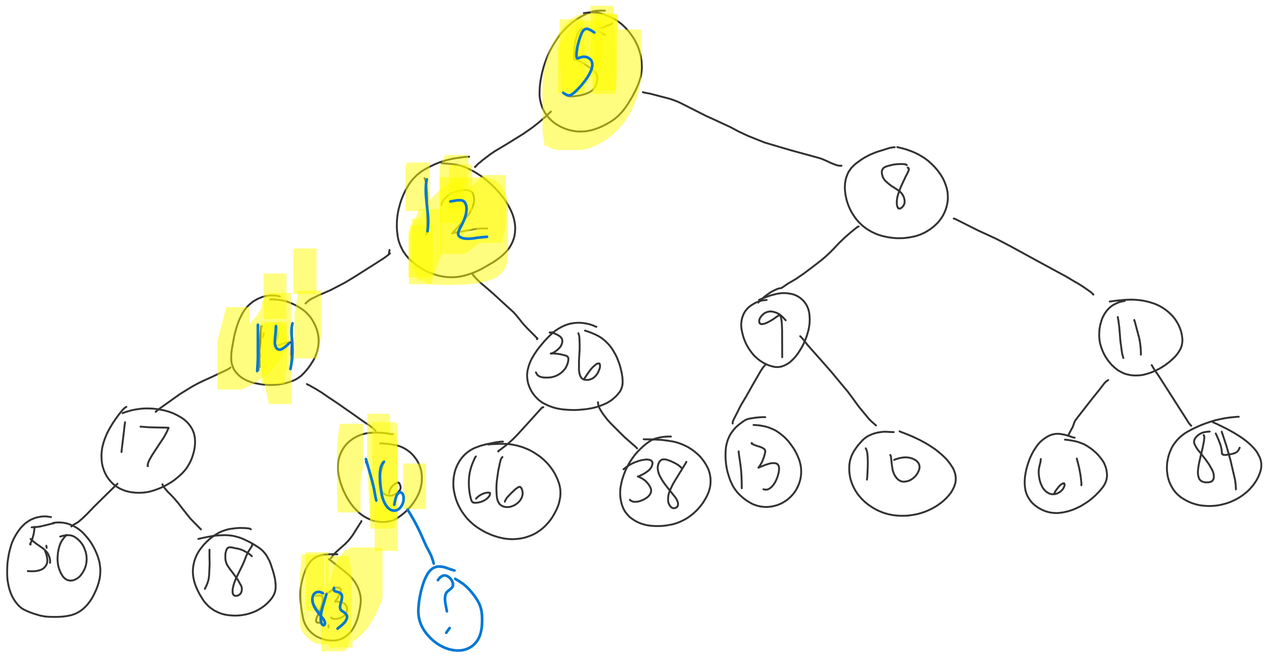


Let's say we want to insert 14 into the heap. What should we do?

Well, if we put it in the ? location, will the heap property be satisfied? ___no__________

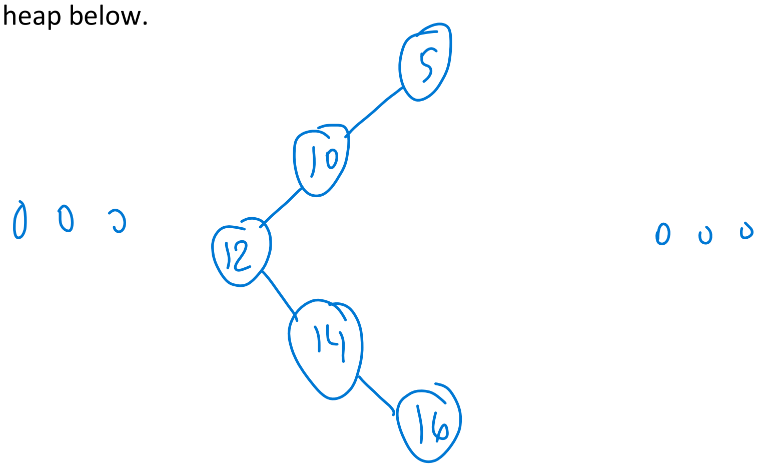
So, we put it in the ? location and then push it up toward root into it is in the right location. What path is the ? node on? <5, 12, 16, 83, ?>

14 would be between 12 and 16 if this is just an ordinary linked list, so that's where we will put it.

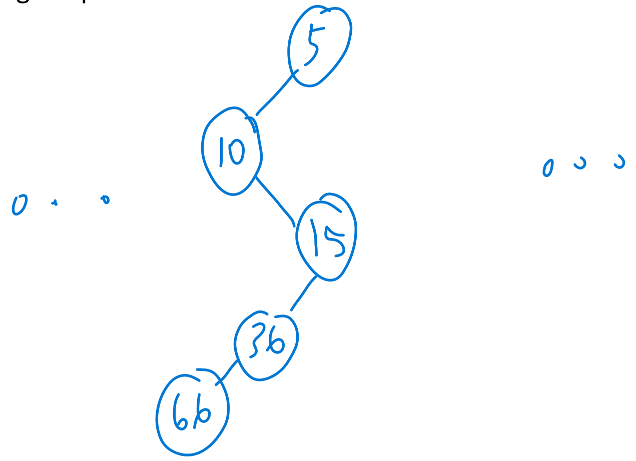


Is this still a heap? ____yes____

Practice: Insert 10 into the heap above (which is next empty child, which path to use, where to insert?). Draw resulting heap below.



Practice: Insert 15 into the just modified heap (which is next empty child, which path to use, where to insert?). Draw resulting heap below.

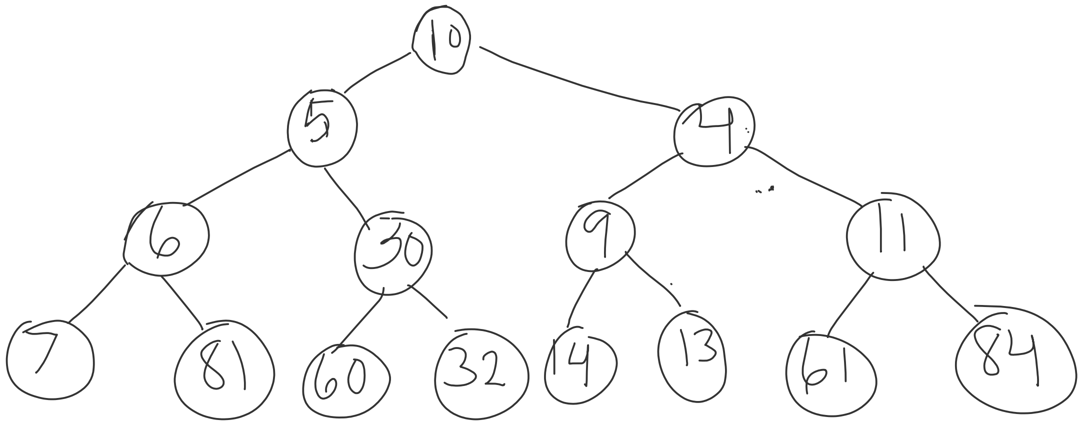


What is the runtime of HEAP-INSERT? ____O(lg N)____

Min-Heapify (turning a bad heap into a good heap)

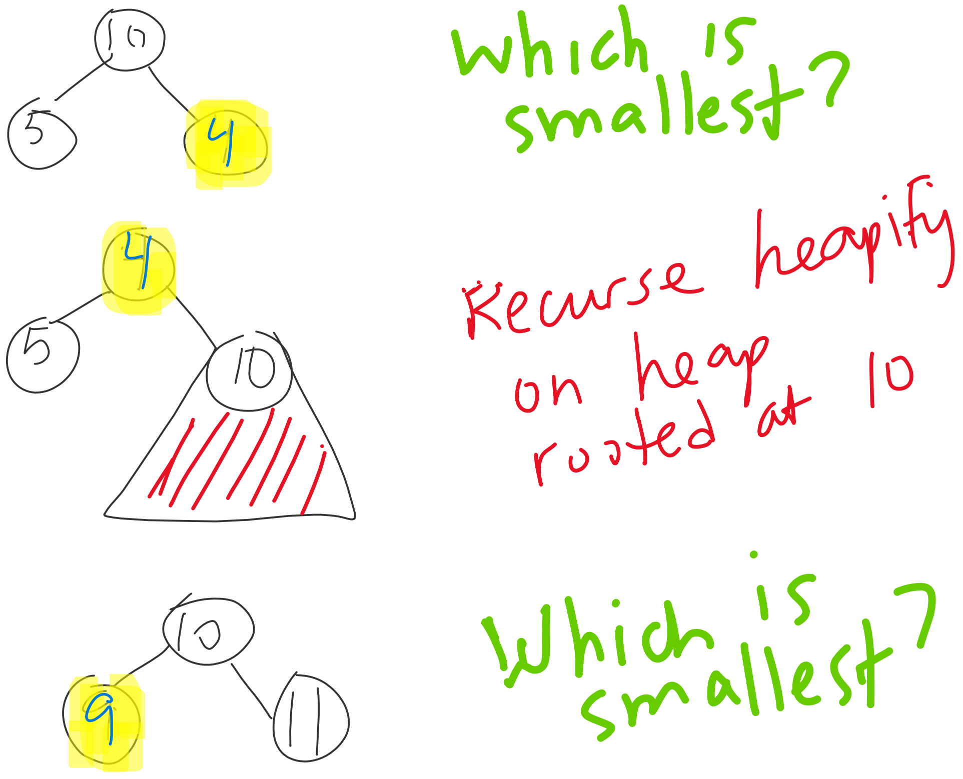
Idea: if an element (key) is misplaced in the heap, keep moving it down until min-heap property is satisfied.

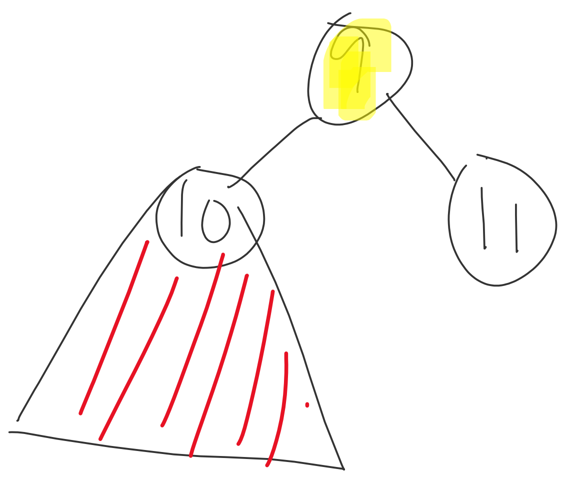
Let's look at a bad heap. What element(s) is/are out of place? ____10____



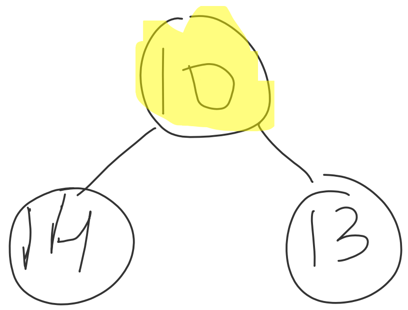
How can we turn the bad heap into a good heap?

Start at bad element and move element downward until it is less than or equal to both children. This is a recursive process.





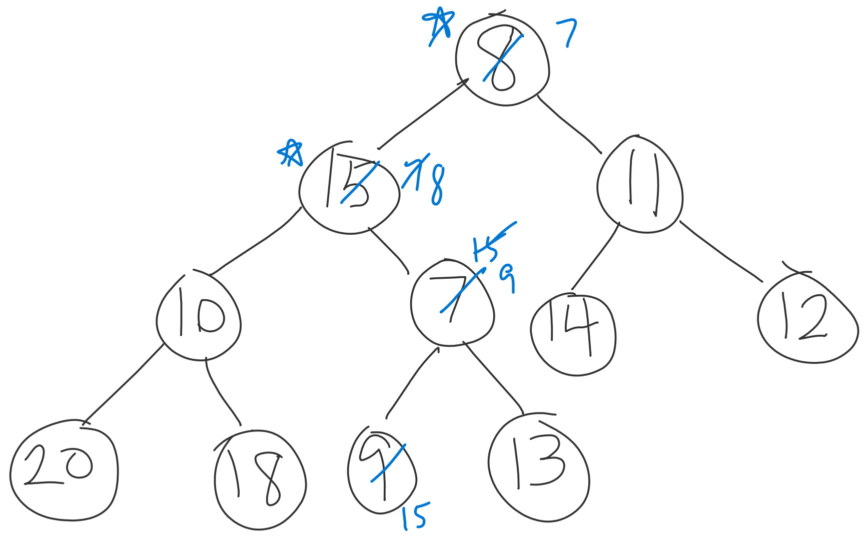
Recurse here



Which is smallest?

Recursion stops ✗

Practice: Find the "bad" element in the heap and apply the MIN-HEAPIFY procedure from that location down until recursion stops.



What is the runtime of the MIN-HEAPIFY procedure?

What is the longest path in the worst case? lgN

How much work is done at each stop along the path? Θ(1)

Total runtime: O(lgN)

We can write the MIN-HEAPIFY runtime as a recurrence as well. Each recursive step goes to the left child or right child, which means the recursive step is done on a heap of size n/2.

$$T(n) = \theta(1) + T\left(\frac{n}{2}\right)$$

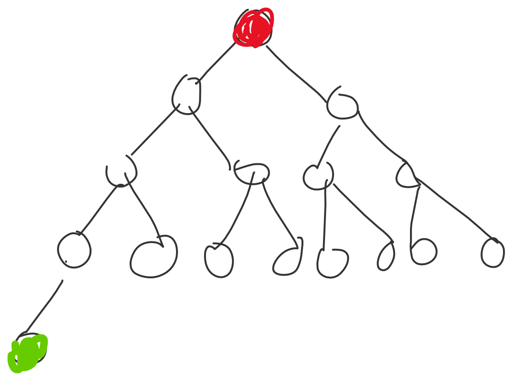
Using the master method, what is the runtime of the recurrence? Θ(lgN)

Removing an element from the heap

Where is the easiest to remove an element (key) from the heap? leaf

Unfortunately, leaves are not where the hightest priority item is located. The highest priority item is at the root. We need to remove the root element and then reconstruct the heap. Fortunately, the MIN-HEAPIFY procedure will come in handy once again.

What should we do?



- Remove root data
- Rightmost leaf data goes to root
- Heapify from root

What is the runtime of delete from heap? O(lgN)

Sorting – we can finally sort!!!

Input: Assume A is an unsorted array.

Heapsort(A):

Step 1: Start heap with initial array elements [initially likely a bad heap]

Step 2: Heapify from the lowest level and work up to top

Step 3: While heap is not empty, remove root from heap and insert into sorted list L

What is the upper bound runtime of heapsort?

Step 1: Θ(1) if array is already stored

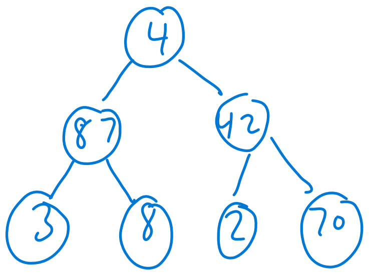
Step 2: N items to heapify, each one calls MIN-HEAPIFY which takes in worst-case O(lgN) time
O(NlgN)

Step 3: N items to remove, each one calls MIN-HEAPIFY which takes in worst-case O(lgN) time
O(NlgN)

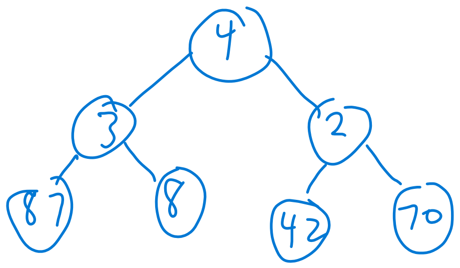
Lower bound is NlgN as well. We could get lucky if A is already sorted. That would be no recursive calls to MIN-HEAPIFY on the heapify step (since it is already a heap). However, the removal would keep putting the largest element at the root on the first swap and trickling the heapify all the way to the leaf which take lgN time per element.

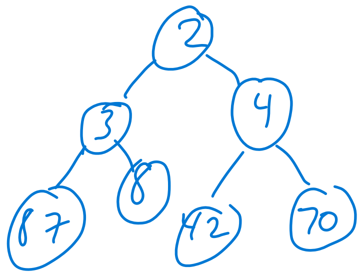
Practice: Create a bad heap with the array elements below.

1 2 3 4 5 6 7
[4, 87, 42, 3, 8, 2, 70]

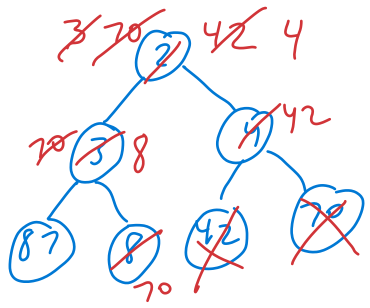


Starting with the lowest level, run MIN-HEAPIFY and show the resulting good heap.





With the good heap, now remove each root and heapify for each step.



2 3

How are heaps useful in practice?

Sorts in place since heap data structure is embedded in array indices (no extra memory required)

Implementation of priority queues when elements can elevate priority

Example: Assume there is a priority queue waiting to use the processor. Each process has a waiting number. For simplicity, let's assume they are [10, 20, 30, 40, 50, …]. Let's say process with priority 60 now gets elevated to priority 15. It's easy to move it up the heap, taking lgN time instead of just a regular array that might involve doing N shifts.

FYI: many other types of heaps (binomial, fibonacci) in the book if you are interested. They do not have the nice underlying array representation but some can merge heaps in lgN time instead of N time.

Warning: the book uses max-heaps (largest value in root), so just be aware when referencing those heaps. We did min-heaps today in lecture to see the opposite inequality among items. Both can work for sorting. If done in place, the max-heaps will eventially sort items in increasing order. If done in place, the min-heaps will eventually sort items in decreasing order. If done in place, the "removed" elements in the heap get put in the back of the array first and the heap is stored in the front part of the array.

CS324: Quicksort

February 14, 2020

To do before next class:

- HW3 due Wednesday, Feb 19
- Read chapter 8 before Monday

Today's learning outcomes:

- Review the quicksort algorithm, partitioning, and using a pivot
- Analyze quicksort (best, worst, average-case)
- Variants of quicksort

Quicksort relies on partitioning a list using a *pivot*. After partitioning, the items in the list to the left of the pivot are less than the pivot and items to the right of the pivot are greater than the pivot. Then, the two sublists are recursively quicksorted. Quicksort sorts in place, meaning that the array L can be used for the "working" storage while we get the data in order. Another advantage of quicksort (like merge sort) is that it can be parallelized.

Here is the pseudocode for one version of quicksort (partition is slightly different than textbook):

```
Quicksort(List L, int low, int high):
      if(low < high):
            part = Partition(L, low, high)
            Quicksort(L, low, part-1)   // recursive call
            Quicksort(L, part+1, high)  // recursive call

Partition(List L, int low, int high):
      pivot = L[low]
      lastSmall = low
      for i = (low + 1) to high:
            if(L[i] < pivot)
                  lastSmall++
                  Swap(L, lastSmall, i)
      Swap(L, low, lastSmall)
      return lastSmall       // pivot location

Swap(List L, int a, int b):
      tmp = L[a]
      L[a] = L[b]
      L[b] = tmp
```

Check for understanding: What can we parallelize about quicksort?

second call to QS

Example: Assume L has the following data, low is 1 and high is 10.

36 10 54 14 83 25 60 72 44 31

pivot = 36

What does the array look like after the call to Partition?

31 10 14 25 36 54 60 72 44 83

In what location did the pivot end up? 5

What can you say about the elements to the left of 36? <

What can you say about the elements to the right of 36? >

Quicksort is then recursively called on the following two sublists:

31 10 14 25 36 54 60 72 44 83

After each of these is partitioned, we get:

25 10 14 31 36 44 54 72 60 83

What are the elements of L during the rest of the recursive calls?

14 10 25 44 60 72 83

10 14 72 83

Do you understand quicksort?

See the textbook (page 173) for the proof of correctness of the loop invariants for quicksort.

How long does quicksort take to run?

Hmmm, it depends.

Worst-Case

1. Provide a list L of 10 items that is the worst input for quicksort: 1, 2, 3, 4, ... 10

2. How many recursive calls are made on this list? $2 \cdot 9$

3. What is the runtime of Partition for a list of length N? $\Theta(N)$

4. Draw out the recursive calls for this worst-case input list of length 10:

$$\begin{tikzcd}
 & QS(1,10) \arrow[dl, dash] \arrow[dr, dash] & & & \\
QS(1,0) \arrow[d, phantom, "\times"] & & QS(2,10) \arrow[dl, dash] \arrow[dr, dash] & & \\
{} & QS(2,1) & & QS(3,10) \arrow[d, dash, dotted] & \\
 & & & \vdots \arrow[dr, phantom] & \\
 & & & & QS(10,10)
\end{tikzcd}$$

The recurrence for the worst-case looks like:

$$T(n) = \overset{\text{partition}}{(n-1)} + \underline{T(n-1)} + \underline{T(0)} \qquad (n-1) + (n-2) + \ldots 1 + 0$$

$$T(0) = 0$$

If we un-roll the recurrence, what do we get for a runtime? $\Theta(N^2)$

How does this compare to insertion sort? same

Best-Case

5. Where would the pivot end up after the call to Partition in the *best case*? middle

6. Suppose that we get lucky every time with the pivot location after each call to Partition. This means the recursive calls to Quicksort operate on an array of size (n/2) or just under and a second array of size (n/2) or just under. What does this recurrence look like?

$$T(n) = (n-1) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = (n-1) + 2T\left(\frac{n}{2}\right)$$

We can use the Master Method and we get a runtime of:

$$\theta(\overset{T(n)}{\cancel{n}}) = \Theta(N \lg N) = \Theta(N \log N)$$

OK, so the best-case is asymptotically the same as merge sort.

We are in a situation where the best-case and worst-case have different trends. What about the average case? Will that be closer to the best-case or the worst-case?

Good-Case (worst split is 90/10 at each level)
Let's suppose we can bound how bad the split can be based on the pivot at each level of the recursion. Suppose we can say that at-worst, at least 10% of the array data is to the left of the pivot and at least 10% of the array data is the right of the pivot.

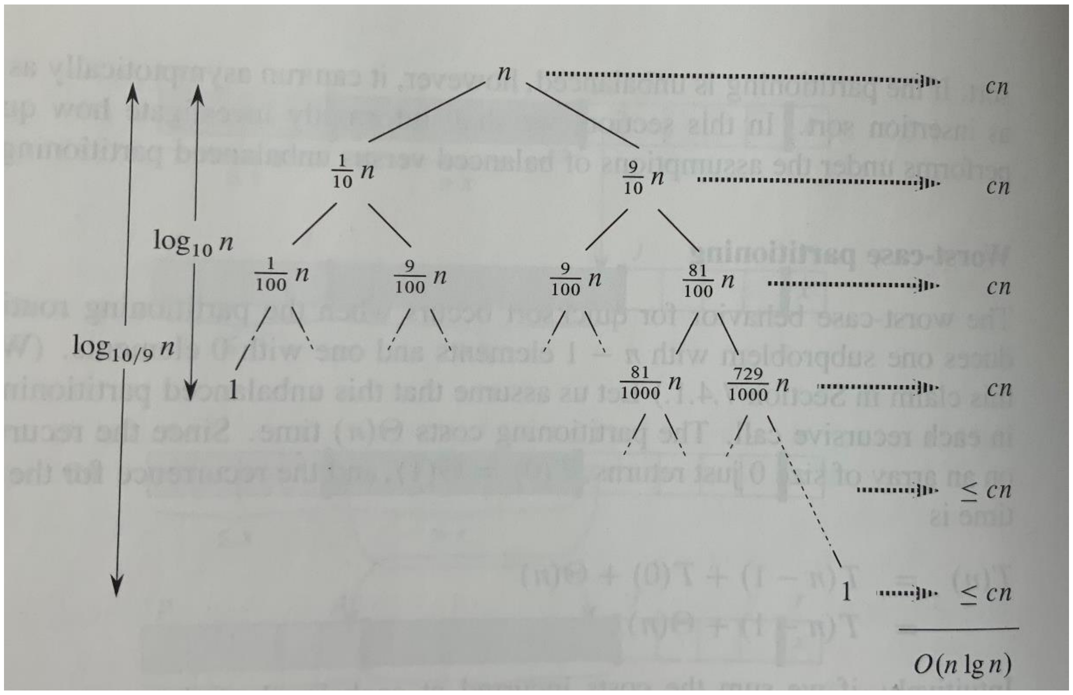


Figure 7.4 from CLRS – 10/90 split for quicksort

The left branch of the recursive call includes 1/10 of the data. The right branch includes 9/10 of the data. So the left-most path has height log_10(n). The right-most path as height log_(10/9) (n). Each level has some constant c times n work to doe the partitioning. So, we get a runtime of O(nlgn).

Note: this is not a formal proof (by picture), but the picture helped us guess the runtime. We would need to prove the following recurrence is O(nlgn) through induction. Assume c in the picture above is equal to 1. Note to keep things simple in the recurrence n is used instead of (n-1) below. Also the split of 9/10 and 1/10 are kept to keep things simple. (left as exercise for the student)

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n$$

It turns out that any bounded-split (999/1 or 99999/1 or 999999 to 1) results in a recursion tree with a logarithmic height.

Average-Case
Let's think about probability now. Assume we call quicksort on an array of n items. What is the probability that the pivot is the smallest element in the array? _____________ $\frac{1}{n}$

What is the probability that the pivot is the biggest element in the array? _______________ $\frac{1}{n}$

What is the probability that the pivot is the middle element in the array? _______________ $\frac{1}{n}$

OK, we have a (1/n) chance for each pivot. (1/n) chance we get a bad pivot. The recurrence looks like:

$$T(n) = (n-1) + \frac{1}{n} \sum_{0 \le k < n} (T(k) + T(n-k-1))$$

$$T(0) = T(1) = 0$$

Let's find an upper bound for this recurrence.

$$T(n) = (n-1) + \left(\frac{1}{n}\right) \sum_{0 \le k < n} T(k) + \left(\frac{1}{n}\right) \sum_{0 \le k < n} T(n-k-1)$$

(split sum into two parts)

We can convert the second sum to go from k from (n-1) to 0, so we can do a change of index for the second sum.

$$T(n) = (n-1) + \left(\frac{1}{n}\right) \sum_{0 \le k < n} T(k) + \left(\frac{1}{n}\right) \sum_{0 \le k < n} T(k)$$

Now we have two sums that are the same, so we can combine them.

$$T(n) = (n-1) + \left(\frac{2}{n}\right) \sum_{0 \le k < n} T(k)$$

Now, I have a recurrence that is more manageable and can prove via induction that the solution to T(n) is O(nlgn).

Proof: T(n) is O(nln(n)).

Base case: T(0) = 0 and T(1) = 0. 0 <= any work done.

Induction hypothesis: Assume that for 0 <= m < n, **T(m) <= C*m*ln(m)** for some constant C.
Note: We are using ln here instead of lg to make the math easier in the induction step. We know that ln and lg are the same asymptotically since they are just a constant factor different.

Induction step:

Consider T(n):

$$T(n) = (n-1) + \left(\frac{2}{n}\right) \sum_{0 \le k < n} T(k)$$

$$T(n) \le (n-1) + \left(\frac{2}{n}\right) \sum_{0 \le k < n} Ckln(k)$$

Note that the induction hypothesis is substituted for T(k) since each of the T(k) values use k from 0 to n. Note that T(0) = 0 and T(1) = 0, so we can remove them from the index for the summation.

$$T(n) \le (n-1) + \left(\frac{2}{n}\right) \sum_{2 \le k < n} Ckln(k)$$

We can now factor out C from the sum:

$$T(n) \le (n-1) + \left(\frac{2C}{n}\right) \sum_{2 \le k < n} kln(k)$$

Now, we need to evaluate the sum. Is kln(k) monotonically increasing for k>=1? __yes__

We can use our approximation by integrals technique, so let's just focus on the sum:

$$\sum_{2 \le k < n} kln(k) < \int_2^n xln(x)dx$$

What is the integral of xln(x)? (Hint: integration by parts, u = ln x, dv = x, du = (1/x), v = $x^2/2$)

$$\int_2^n xln(x)dx = \left[\frac{x^2}{2} lnx - \frac{x^2}{4} + C\right] \Bigg|_2^n$$

With bounds from 2 to n.

This gives us:

$$\int_2^n xln(x)dx = (n^2 * \frac{\ln(n)}{2} - \frac{n^2}{4} - 2\ln(2) + 1)$$

Now, we can plug this in for an upper bound of the sum:

$$T(n) \le \underline{(n-1)} + \left(\frac{2C}{n}\right)(n^2 * \frac{\ln(n)}{2} - \frac{n^2}{4} \underline{- 2\ln(2) + 1})$$

Things underlined are negative, so we can remove them for the inequality:

$$T(n) < n + \left(\frac{2C}{n}\right)\left(\frac{n^2 \ln(n)}{2} - \frac{n^2}{4}\right)$$

Recombine terms:

$$= n + \left(\frac{C}{2n}\right)(2n^2 \ln(n) - n^2)$$

$$= Cnln(n) - \frac{Cn}{2} + n$$

If we choose C=2, we get the subtracted term to be at least the size of n, so we have:

$$T(n) \leq 2nln(n)$$

Conclusion: By induction on n, we have shown that

$$T(n) \leq O(nln(n))$$

Now, how do we get this in the form of nlg(n)?

Recall that:

$$\log_b x = \frac{\ln(x)}{\ln(b)}$$

So:

$$\lg x = \frac{\ln(x)}{\ln(2)}$$

Thus, we have:

$$T(n) \leq 2n * \ln(2) * \lg(n) \sim 1.3863\lg(n)$$

Note: the textbook does this analysis using random variables (an indicator variable to determine if the ith value is compared to the jth value over the entire quicksort algorithm). You should look at this proof approach as well.

Some questions:

1. We know merge sort can run in 1.0nlgn comparisons and quicksort in the average case runs in 1.39lgn comparisons. Why advantage does quicksort have over merge sort? memory
   $\uparrow n$

2. Suppose we quicksort an array of N elements, all with the same value. Will this end up being closer to nlgn or n^2 with the pseudocode given? n^2

Some Variations of Quicksort

Randomized quicksort – do not choose the pivot as the low or high element in the partition range; instead, randomly (uniform distribution) choose a pivot location.

What is the average case runtime for randomized quicksort? $\Theta(N \lg N)$

Quicksort on random input – pre-shuffle input so the probability of each permutation is equally likely.

Median-of-3 quicksort – take low, medium, and high elements in the partition range and choose the median as these three values as the pivot. This guarantees that you do not pick the worst pivot each time.

What is the runtime for an already sorted array for median-of-3 quicksort? $\Theta(N \lg N)$

It turns out that the recurrence for median-of-3 can be solved and shown that the constant in front of nlg(n) is 1.18826. So, it is about 19% worse than the best-case.

Here is the recurrence:

$$T(n) = \frac{8}{3} + (n-1) + \sum_{0 \le k < n} \Pr(medianof3\ belongs\ at\ k)(T(k) + T(n-k-1))$$

Where does the 8/3 come from?

- When we select 3 elements from the array, the number of comparisons we need to make to determine the median of 3 elements is:
  - 2 (with probability 1/3) if we get them in sorted order of min, medium, max or max, medium, min
  - 3 (with probability 2/3) if we get them in order of max, min, medium or min, max, medium or medium, min, max or medium, max min
  - Thus, the average is 2*(1/3) + 3*(2/3) = 8/3.

Where does the (n-1) come from?

- Work of partition, just like our other recurrences

What is the probability that the medianof3 belongs at position k?

$$\frac{k(n-k-1)}{\binom{n}{3}}$$

Numerator: number of ways to choose an element that belongs in position less than k times the number of ways to choose an element that belongs in position greater than k.

Denominator: number of ways to choose 3 elements from n

CS324: Radix-Sort and Bucket-Sort

February 17, 2020

To do before next class:

- HW3 due Wednesday, Feb 19
- Coming up: Exam 2 on Wed, Feb 26
- Coming up: HW4 due Monday, Feb 24 (part A is a programming assignment)
- Read chapter 11 before Wednesday

Today's learning outcomes:

- Lower bound of comparison sorts
- Radix sort – sorting by place-value
- Bucket sort – putting items into buckets and then sort each bucket; aim to get small buckets

---

So far, we have looked at comparison sorts, where we use operations like <, <=, >, >= to compare two elements:

- Insertion sort
- Mergesort
- Heapsort
- Quicksort

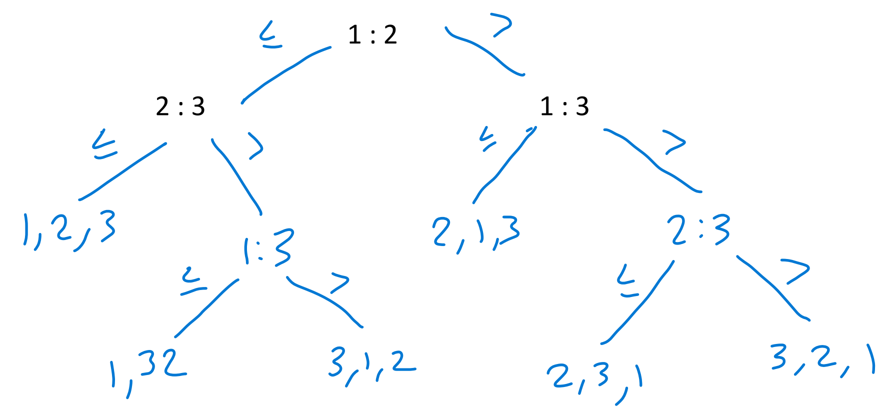
Comparison sorts mean that we can sort anything that has a comparator operator defined (for example, if we define a new struct and a comparison operator, we can compare two instances of that new data type).

What is the lower bound for comparison sorting? Hmm, let's think about how many permutations we can have of N elements.

If I want to sort an array of size 3, how many different permutations are there? 6

Let's draw the decision tree:

....

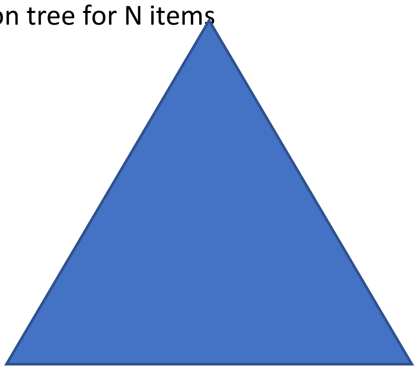


How many leaves do we have? ____6____

How many comparisons do we have in the longest branch? ____3____

OK, let's generalize this.

Decision tree for N items



How many leaves (permutations)? ___N!___

Suppose the decision tree has height H. How many leaves are in a binary tree with height H?

$$n! \leq 2^H$$

$$\underbrace{\lg(n!)}_{} \leq H \rightarrow \Theta(n\lg n)$$

Since lg(n!) is Θ(nlgn), we have the height is big-omega of nlgn. Therefore, the number of comparisons needed for sorting has a lower bound of nlgn.

$$\textit{Comparison sorting is } \Omega(nlgn)$$

Let's turn our attention to sorting where we know something about our data. Can we sort faster than nlgn? ____yes____

What properties of data can we take advantage of? digits

a - z

Radix Sort

Data assumption: data has digits/places and each digit/symbol is comparable

For example, 1358 is a 4-digit number.

For example, "algorithms" is a 10-letter word.

```
RADIX-SORT(A, d):                //d is number of places in largest/longest element
1      for i = 1 to d            //1 is right-most place
2            Stable sort A on digit i   // stable sort means we keep elements in order
                                        // like physical cards
```

Let's see it in practice.

Sort the following numbers using Radix-Sort:

Input:
275 ←
125 ←
032
116
103
250

stable

1. Right-most digit:
250
032
103
275
125
116

2. 10s-digit:
103
116
125
032
250
275

3. 100s-digit:
032
103
116
125
250
275

Practice: Use the other handout to create cards. Sort the cards with the numbers on them using Radix-Sort.

Sort the cards with the words on them using Radix-Sort.

See the textbook for the correctness of Radix-Sort. Induction on the column being sorted.

What is the runtime of Radix-Sort? Assume d is the width of widest input. Assume N is the number of items. Assume k is the amount of digits (10 for base-ten numbers, 26 for lower-case words).

How many iterations do we do given d, N, and k? d

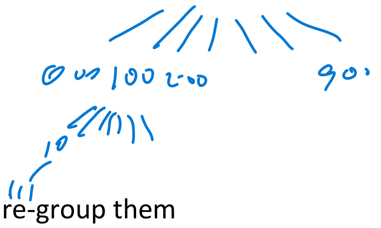
How many items do we process each iteration given d, N, and k? N

How many digits do we have (boxes we would need to do a stable sort)? k

Total runtime: $\Theta(d(N+k))$

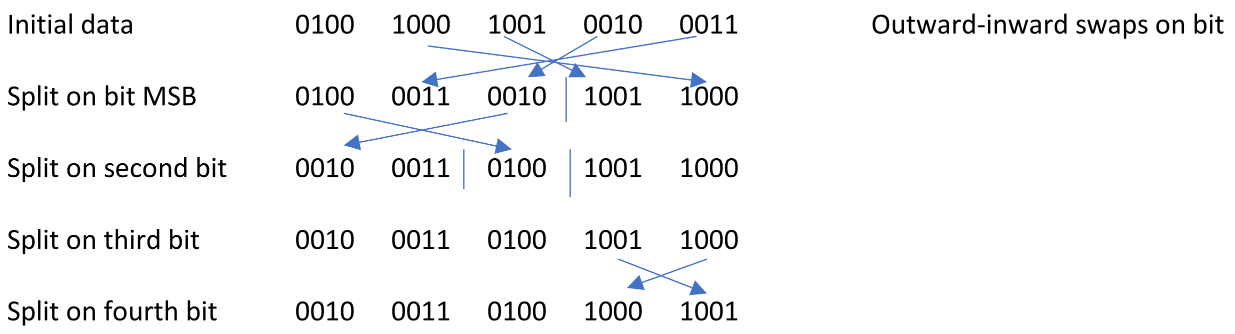
Note: if N dominates k, then we get: $\Theta(dN)$

Note: if d is constant, then we get: $\Theta(N)$ // note that for numbers, d is lg(N) for binary
// note that for numbers, d is log(N) for decimal
// linear runtime only when N dominates d



Variants of Radix-Sort:

- We can also sort by most-significant place order if we keep track of sub-lists and re-group them back together in the correct order.
- We can do a Radix-exchange sort over bit positions. See below for an example.



Initial data	0100	1000	1001	0010	0011	Outward-inward swaps on bit
Split on bit MSB	0100	0011	0010	1001	1000	
Split on second bit	0010	0011	0100	1001	1000	
Split on third bit	0010	0011	0100	1001	1000	
Split on fourth bit	0010	0011	0100	1000	1001	

Comments on Radix Sorts:

- Radix-exchange can be done in place. The other two require extra temporary storage.
- If N does not dominate k and d, then the linear runtime is misleading.
- Need to be able to extract “digits” from data to be usable. Not appropriate for any new struct that simply has a comparison operator defined.
- Behavior on floating point numbers (with large number of bits) and long strings can be quite poor.

Bucket-Sort

Demo with cups and slips of paper.

Idea: put data in a bucket (cup). Then, insertion sort each bucket. Re-group in order of the buckets.

To keep things simple, we will assume we are sorting N items into N buckets.

```
BUCKET-SORT(A)      //textbook assumes A contains data in range [0, 1)
1       N = A.length
2       B = new array[0..(N-1)]
3       for i = 0 to (N-1):
4               B[i] = {}     //empty list
5       for i = 1 to N:
6               B[floor(N*A[i])] = B[floor(N*A[i])] + {A[i]}  // concatenate to list
7       for i = 0 to (N-1):
8               Sort B[i] with insertion sort
9       Concatenate B[0], B[1], B[2], …B[N-1].
10      A = B
```

What is the runtime of bucket-sort?

How many times does loop at line 3 execute? ___N___ What is work at line 4? ___$\Theta(1)$___

How many times does loop at line 5 execute? ___N___ What is work at line 6? ___?___ $|B[i]|$

How many times does loop at line 7 execute? ___N___ What is work at line 8? ___$|B[i]|^2$___

What is work of line 9? ___N___

The only thing that is a bit tricky is line 6 and line 8. We need to know how many items we can expect in each bucket.

Assume the data is **uniformly distributed** over [0, 1). What is the expected number of items per bucket? ____1____

What is the probability that the item will go into bucket 0? ____________ $\frac{1}{N}$

OK, now we can utilize the Binomial distribution to see the probability that we get a lot of items in one bucket. For simplicity, let's just consider bucket 0.

Assume we have 100 buckets. Each item has probability of 1/100 of going into bucket 0. We can then say the "success" probability is 1/100 of going into bucket 0. We can say the "failure" probability is 99/100 of going into bucket 0.

Let's use the binomial distribution now to see what the probability is of 1 item going into bucket 0 and 999 items going into other buckets.

$$\Pr(X = 1) = \underline{\binom{100}{1}}\ \underset{p}{(.01)^1}\underset{(1-p)}{(.99)^{99}} = .3697$$

Let's see what the probability is of 2 items going into bucket 0 and 998 items going into other buckets.

$$\Pr(X = 2) = \binom{100}{2}(.01)^2(.99)^{98} = .1849$$

Let's see what the probability is of 3 items going into bucket 0 and 997 items going into other buckets.

$$\Pr(X = 3) = \binom{100}{3}(.01)^3(.99)^{97} = .0610$$

So, it's getting smaller and smaller. Even to get 5 items in bucket 0 is a probability of 0.00289.

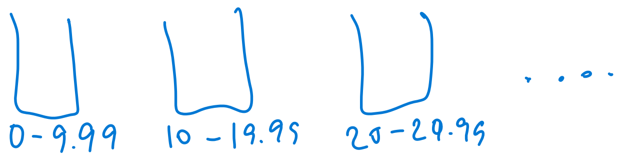
Therefore, with N buckets and N items that are uniformly distributed, it is rare to get more than 5 items in a bucket.

So, we can now look at the runtime again. Lines 6 and 8 can be done in constant work (even though line 8 is insertion sorting – but it is insertion sorting at most 5 items in 99.8% of the time). Therefore, the runtime of bucket-sort is Θ(n).

What do we do if the data is not from the interval [0, 1)?

Ideas?

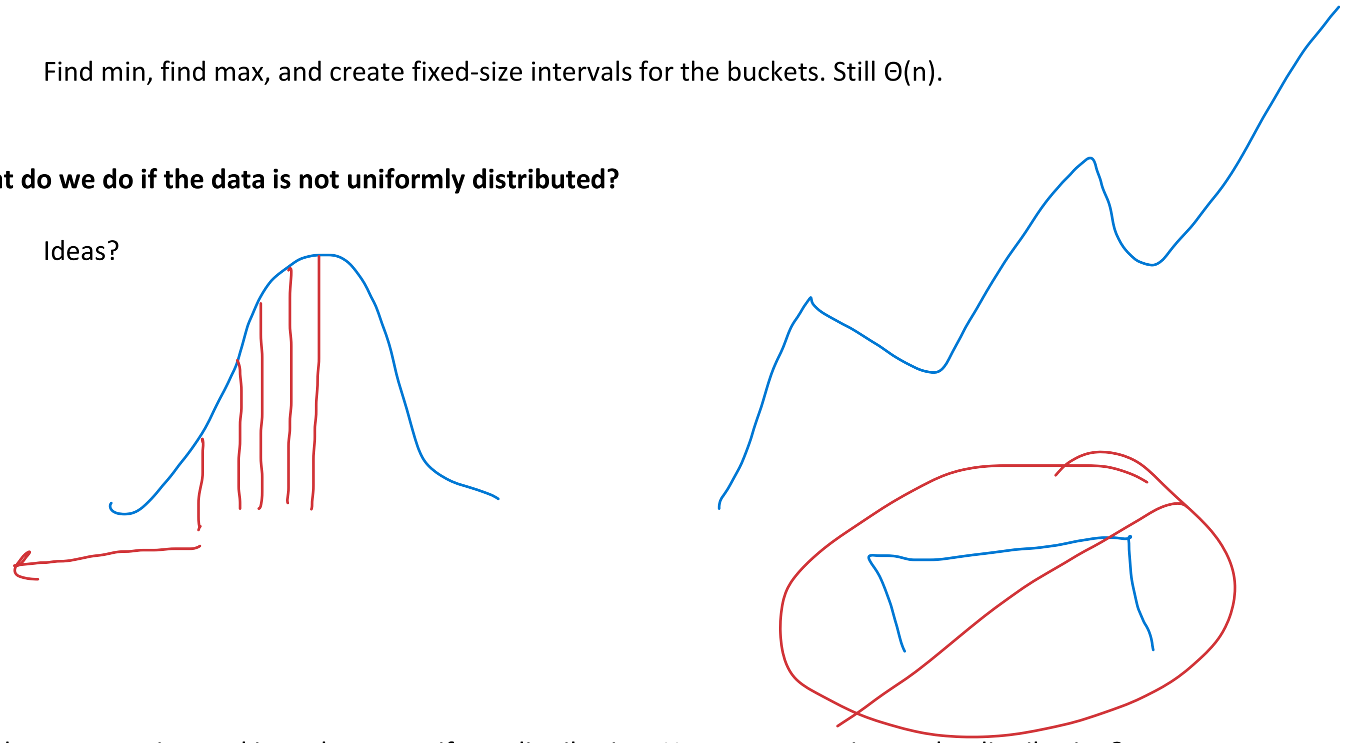
[0, 100)
10 buckets



Find min, find max, and create fixed-size intervals for the buckets. Still Θ(n).

What do we do if the data is not uniformly distributed?

Ideas?



Need to turn any interval into close to uniform distribution. How can we estimate the distribution?

Take a fixed sample S (like 98 random items, assuming N is much larger) and sort them.
Find min of all N items.
Find max of all N items.
Let’s say there are 10000 buckets.
Put sorted numbers on every 100th bucket and interpolate in-between buckets uniformly in order to decide where to place elements. When placing element E, determine which two items in S sandwiches E and then interpolate between these two values.

What is the runtime of all this? $\underline{\quad \Theta(n) \quad}$

We have now seen how to sort in linear time. But, remember, we did not do any direct element to element comparisons for radix sort and bucket sort. We had to know something about the data itself.

CS324: Hashing

February 19, 2020

To do before next class:

- Coming up: Exam 2 on Wed, Feb 26
- Coming up: HW4 due Monday, Feb 24 (part A is a programming assignment)
- Read chapter 12 before Friday

Today's learning outcomes:

- Review hashing, hash tables, and functions
- Review collision resolution (chaining, open-address)
- Analysis of hash table operations

Hash tables should be review from CS305: Data Structures. We will quickly review what a hash table is.

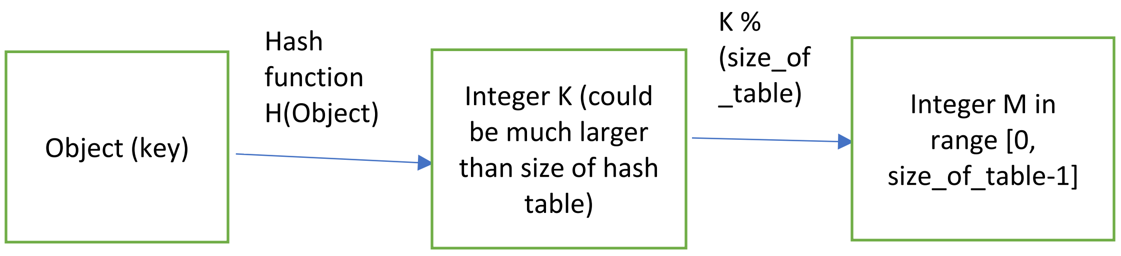
A hash table implements the Dictionary ADT

- Insert
- Find
- Delete

Do you ever use a **dictionary** type in a programming language? ___yes___ (likely a hash table)

If you want to do queries such as "Return the object just less than the 'corn'", is a hash table a good idea? ____________________

Hash tables are not good choices for keeping data in order.



Object (key)
Hash function H(Object)
Integer K (could be much larger than size of hash table)
K % (size_of_table)
Integer M in range [0, size_of_table-1]

Then, Object is stored at T[M]. We will deal with collisions later.

Example: Assume the hash function from a string to the value K is given below.

```
unsigned int hash(char *key) {
        unsigned int rtnVal = 3253;
        char *p;
        for (p = key; *p != '\0'; p++) {
                rtnVal *= 28277;
                rtnVal += *p * 2749;
        }
        return rtnVal;
}
```

So, hash("a") is:

```
        rtnVal = 3253
        rtnVal = 3253 * 28277            // 91985081
        rtnVal = 91985081 + 97*2749      //note: 'a' in ASCII is 97
        rtnVal = 92251734
```

Assume we have a hash table of size 10. We want to insert the following items:

- "coconut"
- "milk"
- "apple"

STEP 1: We'll use the hash function listed above to calculate the hash values for these keys:

- "coconut" (hash value = 2104178476)
- "milk" (hash value = 461110994)
- "apple" (hash value = 3515030035)

STEP 2: We'll map the hash values to the size for this hash table (% 10):

- "coconut" (location = 6)
- "milk" (location = 4)
- "apple" (location = 5)

eggs

	corn		orange	"milk"	"apple"	"coconut"			
0	1	2	3	4	5	6	7	8	9

eggs

1. Now, insert "orange" into this hash table. Its hash value is 3410197053. Insert this key into the table above.

2. Insert "corn". Its hash value is 1347376851. Insert this key into the table above.

3. Insert "eggs". Its hash value is 881505635. Insert this key into the table above.

4. What should we do?

The book describes several options. Chaining and open-address are common ways to deal with collisions. We will look at chaining in this lecture. See the textbook for open-addressing – finding empty cells with other probing techniques. Note that deleting items in open-address hash tables needs care in defining some indicator value that something was deleted so other lookups to the same hash value K keep searching after finding a deleted cell.

Questions on hash functions

1. Suppose we insert "apple" into our hash table and the hash function returns 12345. Suppose we want to find "apple". What should the hash function return? 12345

2. Now, suppose we want to find "Apple" in the same table. What should the hash function return? depends

3. Should a hash function distribute keys *uniformly* over the range of the function? yes

4. Should a hash function be fast to compute? yes

5. Should a hash function map near keys (such as "corn" and "corny") to very different hash values? yes

6. Should there be a relationship between the key and the hash function? return value H

The book has several numerically-based hash functions (ascii value of chars, division method using mod, multiplication of bits). Another class of has functions does a bunch of bit manipulations based on the actual storage of the object (key) and then do a table lookup.

For example, a hash function on a 32-bit key might be:

- Take bits 0, 3, 7, 9, 12, 15, 17, and 28 of the key and append them. XOR value into key.
- Pick another set of bits of the key and append them. XOR value into key.
- Pick another set… (keep going so that each bit position is chosen at least once)
- Hash value is the final value after last XOR.

With the hash table example above, we get a collision in box 5. The box already has data in it when we want to insert "eggs".

We can resolve collisions through chaining. This is similar to bucket-sort, where we end up with a linked list of items within the same bucket. Chaining is the same process – we could end up with a linked list of items within the same location in the hash table.

Now, let's look at the process for **insert** and to keep things simple, we will assume the hash function's range is the same as the hash table size.

```
Insert(T, key):
        K = hash(key)
        Insert key into head of linked list at T[K]

Find(T, key):
        K = hash(key)
        Search linked list at T[K] for key
        If key is found
                Return key
        Else
                Return NIL

Delete(T, key):
        K = hash(key)
        Search linked list at T[K] for key
        If key is found
                Delete linked list node and repair
```

Let's analyze each of these functions in the average (expected) case.

M = number of buckets (hash table size, also likely the range of the hash function)
N = number of keys currently in hash table

With the assumption that keys are uniformly distributed, what is the expected number of keys in each bucket?_______________ $\frac{N}{M}$

Find – key is not in hash table
The expected number of elements compared is the size of the linked list: _______________ $\frac{N}{M}$

Find – key is in hash table
Compare key with each item in bucket until we find it. The average case utilizes the fact that the item is in any of the linked list positions with equal probability. The book (page 260) has a nice derivation of this. The expected number of elements compared is $1 + \frac{N-1}{2M}$. We will do a high-level analysis of this below.

Insert – key is not in hash table (and you know it is not already there)
Where does the element go in the linked list? ____front____
Expected number of elements compared is: ____0____

Insert – key is not in hash table (but we do not yet know that)
We need to do a find (key not located) followed by an insertion.
Expected number of elements compared is: _______________ $\frac{N}{M}$

Insert – key is already in hash table (but we still need to discover this)
We need to do a find (key located).
Expected number of elements compared is: __________________ $1 + \frac{(N-1)}{2M}$

Delete – key is not there
We need to look through entire linked list.
Expected number of elements compared is: ___________________ $\frac{N}{M}$

Delete – key is there
We need to do a find (key located), followed by linked-list pointer repair, which is constant time.
Expected number of elements compared is: __________________ $1 + \frac{(N-1)}{2M}$

Why is the expected number of comparisons for a find when the key is there, the following?

$$1 + \frac{N-1}{2M}$$

The 1 is guaranteed since we know the item is in the linked list, so we have at least 1 comparison.

What is the probability of all other items in our hash table being anywhere? ______________ ☆ $\frac{N-1}{M}$

N-1 other elements in table
M slots (buckets)

For a linked list of length K, we do at least 1 comparison and at most K comparisons when searching. So, our average case is (K+1)/2. The expected length of the list, from above, is:

$$K = \left(\left(\frac{N-1}{M}\right) + \overset{\swarrow \text{ we know it's there}}{1}\right)$$

With this value for K, we have the expected number of comparisons as:

$$\frac{K+1}{2} = \frac{\overbrace{\left(\left(\frac{N-1}{M}\right) + 1\right)}^{K} + 1}{2}$$

With algebra, this becomes:

$$\left(\frac{N-1}{2M}\right) + 1$$

What happens when N/M (load factor) becomes large?

What this means is that our expected lengths of our linked lists get longer and longer.
Our expected runtime for find, insert, and delete get longer and longer.
It could get as much as Θ(N), which is no better than just a single linked list!!!

1. What should we do when N overwhelms M?

increase size of hash table

re-map all keys

~~*~~ may need new hash function

de-allocate original

2. How much time will doing all this take?

$\Theta(M)$

M = # buckets

$\Theta(N)$

$\Theta(M)$

$\Theta(N+M)$

3. How many times do we need to do this as the number of keys in the table increase?

?

depends on size of dictionary

Here is where **amortized** analysis comes in. I think of amortized analysis as a "bank". You put pennies into the bank as you do each operation, so that you have enough money saved to complete one large operation when you need to complete it. The idea is that for each insert into the hash table, we put three pennies into the bank. When the hash table gets to a certain load factor (let's say 10), we dump the pennies out of the bank to pay for the large cost of creating a larger table and re-hashing all our keys.

Aside: garbage-collected languages like Java do this with memory allocation. Most memory allocations are quick since there is space available. However, when there's a lot of memory that must be garbage-collected, that garbage-collection is a large cost, but the cost is proportional to the amount of memory already allocated.

Instead of money, we can think of putting "time" into the bank.

Suppose insert takes 1 second.
We insert 999 keys into the hash table.
The 1000th key would make the load factor too high, so we allocate new memory and re-insert all the data, which takes 1000 seconds.

From an observer's perspective, this is: 999 fast operations and 1 slow operation. However, the whiole thing took about 2000 seconds, so we could think of each insert taking 2 seconds when spread over the entire set of 1000 insertions.

Example of amortized analysis

Assume a hash table of size 10 buckets, initially empty.
Assume a load factor of 10.
Assume hash table doubles in size each time (this is necessary for this amortized analysis to be linear time)

	Bank Deposits (3 cents per insertion)	**Bank Withdrawals**	**Balance**
99 insertions	297		297
1 insertion	3		300
Allocate 20 buckets, move 100 items, deallocate old table		100	200
99 insertions	297		497
1 insertion	3		500
Allocate 40 buckets, move 200 items, deallocate old table		200	300
199 insertions	597		897
1 insertion	3		900
Allocate 80 buckets, move 400 items, deallocate old table		400	500

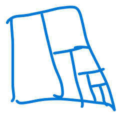
We keep enough money in the bank for each re-sizing of the hash table. Note that we ignored finds and deletes, which only helps us in the amortized analysis – that's more operations that we could use to pay the bank when we need to make a withdrawal for the costly operation.

An actual proof of amortized analysis would be by induction on the "bank having a non-negative balance" and each induction step is a doubling of the hash table size.

Another way to look at this:

Suppose there are N elements in our hash table in the end. That means there were the following re-sizes up to the value of N:

$$\underline{N} + \underline{\frac{N}{2}} + \underline{\frac{N}{4}} + \underline{\frac{N}{8}} + \cdots = N\overset{2}{\left(\underbrace{\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots}\right)} = N * 2 \quad + 2$$



Due to the geometric series (1/2) + (1/4) + (1/8) + …. that you showed in a prior homework is = 2. Thus, we have 2N re-sizes over time and N inserts for a total of 3N.

Thus, amortized analysis of hash table operations is Θ(1).

FYI – see **universal hash functions** in the textbook – interesting way to make good hash functions that scatter, even if adversary has discovered the hash function and sets keys to make the buckets not uniformly distributed.

CS324: Binary Search Trees, AVL Trees, and Treaps

February 21, 2020

To do before next class:

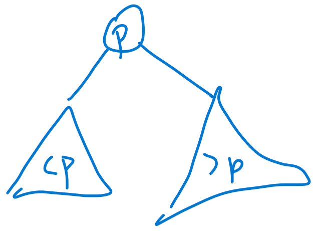
- HW4 due Monday, Feb 24 (or Sunday at noon if you want part B returned before the exam)
- Read chapter 13 before Monday
- Exam 2 is Wednesday, Feb 26 (can use 1 crib sheet, both sides and a calculator on exam)

Today's learning outcomes:

- Review binary search trees data structure
- AVL trees or balanced binary search trees
- Analyze the expected depth of a node in a randomly-built BST

---

1. Binary search trees should be review from CS 305. What is a binary search tree?



What operations can one do with a BST (a BST implements the Dictionary ADT)?

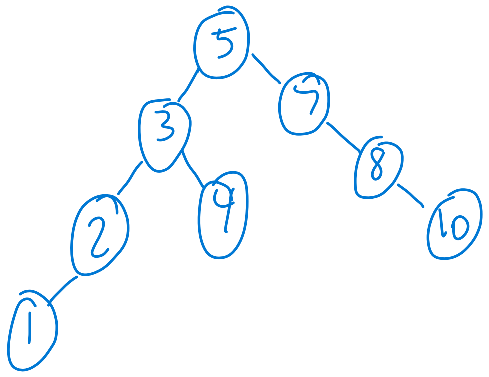
- Insert
- Delete
- Find
- *Note: these are the same functions as a hash table*

What kinds of questions can you ask when using a BST that you can't when using a hash table?

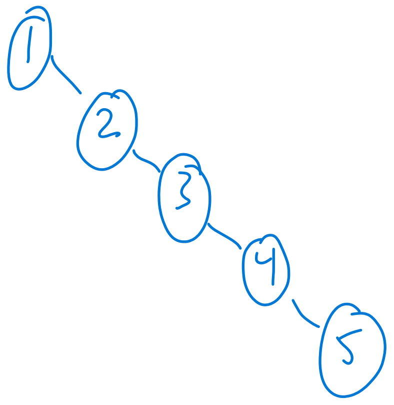
< 100 ?   # are between 2 and 10

2. Draw the BST that results from inserting the following keys (in the order given) into the tree:

[5, 7, 3, 2, 1, 8, 10, 4]

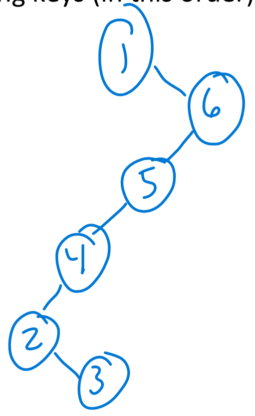


3. Draw a really unbalanced BST



4. Draw the BST from inserting the following keys (in this order):

[1, 6, 5, 4, 2, 3]



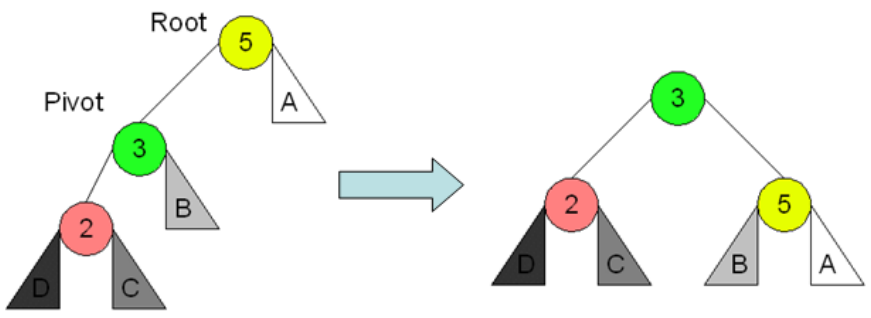
5. How can BSTs become unbalanced?

- Deletions can shorten branches in an imbalanced way
- Insertions of sorted data
- Insertions along long branches

AVL Trees (self-balancing BSTs)

Note: we can rotate a BST when it becomes unbalanced (height of right-child and height of left-child differ by more than 1). These are called **AVL trees** or **self-balancing binary search trees**.

a) Here's how we can balance when the right-child is longer than the left-child:

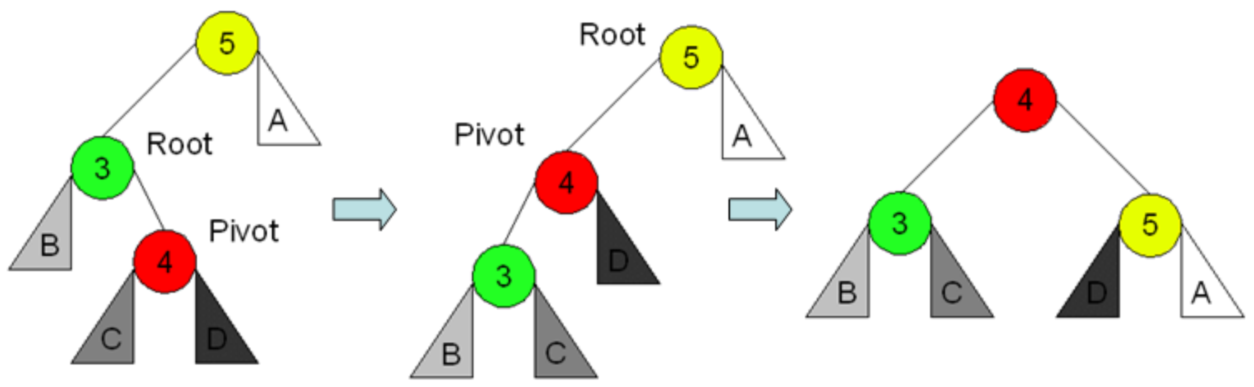


Left-Left Case: Right Rotation

Figure by Daisy Tang

b) The Right-Right Case is a mirror image of this one.

c) The Left-Right case is as follows:



Left-Right Case: Left-Rotation and then Right-Rotation

Figure by Daisy Tang.

d) The fourth case is a Right-Left Case where we do a right-rotation and then a left-rotation.

Those are the four cases for balancing BSTs.

6. What is the runtime for **inserting** into a balanced BST with size N? $\underline{\Theta(\lg N)}$

7. What is the runtime for **deleting** from a balanced BST with size N? $\underline{\Theta(\lg N)}$

8. What is the runtime for **finding** a key in a balanced BST with size N? $\underline{O(\lg N)}$

9. What is the runtime for **balancing** a BST with size N? $\underline{\Theta(\lg N)}$ for calculating depths + $\Theta(1)$ for pointer updates

---

Now, we will make the assumption that we are just using regular binary search trees (no balancing). We are going to ask the question:

What is the average node depth in a randomly-built BST? This will tells us the average-case runtime for find, insert, and delete.

Check: What does randomly-built mean?

First, we will do the analysis of the node depth of the smallest key (S) in the BST to get some intuition about the analysis. Assume the BST has N items (after N random insertions).

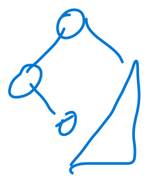
D(N) is the expected depth of S.

10. What is the probability that the smallest key is the root? $\frac{1}{N}$, depth of S? 0

11. What is the probability that 2nd smallest is the root? $\frac{1}{N}$, depth of S? $D(1)+1$

12. What is the probability that the 3rd smallest is the root? $\frac{1}{N}$, depth of S? $D(2)+1$

13. What is the probability that the largest is the root? $\frac{1}{N}$, depth of S? $D(N-1)+1$



Thus,

S is root

$$D(N) = \cancel{\frac{0}{N}} + \underline{\frac{1}{N}\sum_{0<k<N}(1+D(k))}$$

$$D(1) = 0$$

Eliminating 0-term and splitting sum, we get:

$$D(N) = \frac{1}{N}\underline{\sum_{0<k<N}1} \quad + \quad \frac{1}{N}\sum_{0<k<N}D(k)$$

$$D(N) = \frac{N-1}{N} + \frac{1}{N}\sum_{0<k<N}D(k)$$

Now, what is the asymptotic analysis of D(N)?

We'll show that D(N) is O(lgN).

Proof:

Base case: Consider D(1). Since D(1) = 0, 0 <= 0 == C*lg1.

Induction hypothesis: Assume for 1 <= m < n:

$$D(m) <= C * lg(m)$$

Induction step: Consider D(n):

$$D(n) = \frac{N-1}{N} + \frac{1}{N}\sum_{0<k<N}\underline{D(k)}$$

Applying the induction hypothesis for D(k), we get:

$$D(n) \leq \frac{N-1}{N} + \frac{1}{N}\sum_{0<k<N}C * \lg(k)$$

I.H.

Let’s split the sum into two regions, one in the lower half and one in the upper half:

$$\le \frac{N-1}{N} + \frac{1}{N}\sum_{0<k\le\frac{N}{2}} C * \underline{\lg(k)} + \frac{1}{N}\sum_{\frac{N}{2}<k<N} C * \underline{\lg(k)}$$

Now, we can bound each term in the sum by N/2 for each k in the left sum and N for each k in the right sum.

$$\le \frac{N-1}{N} + \frac{1}{N}\underline{\sum_{0<k\le\frac{N}{2}}} C * \underline{\lg\left(\frac{N}{2}\right)} + \frac{1}{N}\sum_{\frac{N}{2}<k<N} C * \underline{\lg(N)}$$

Now, we can factor out the summation terms since they do not depend on k.

$$\le \frac{N-1}{N} + \frac{1}{N}\underline{\left(\frac{N}{2}\right)} C\,\underline{\lg\left(\frac{N}{2}\right)} + \frac{1}{N}\underline{\left(\frac{N}{2}\right)} C\,\underline{\lg(N)}$$

$$= 1-(\frac{1}{N}) + \left(\frac{1}{2}\right) C\lg\left(\frac{N}{2}\right) + \left(\frac{1}{2}\right) C\lg(N)$$

$$= 1-(\frac{1}{N}) + \left(\frac{1}{2}\right) C[\lg\left(\frac{N}{2}\right) + \lg(N)]$$

$$= 1-(\frac{1}{N}) + \left(\frac{1}{2}\right) C[\lg(N) + \lg(N) - \lg(2)]$$

$$= 1-(\frac{1}{N}) + \left(\frac{1}{2}\right) C[2\lg(N) - \lg(2)]$$

$$= 1\boxed{-(\frac{1}{N})} + C\lg(N) - \frac{C}{2}$$

Take out the –(1/N) term since this is a <= proof.

$$\le 1 + C\lg(N) - \frac{C}{2}$$

$$= Clg(N) + (1-\frac{C}{2})$$

Thus, to get the (1 – C/2) term to be less than or equal to 0, we can set C to be 2.

Conclusion: By mathematical induction, D(N) is O(lgN).

Therefore, the expected depth of the smallest key in the BST is O(lgN). Thus, the runtime for insert, find, and delete is O(lgN) for a a randomly-built BST.

The expected depth of any node in the randomly-built BST can also be shown to be O(lgN) with a similar proof. The recurrence is:

$$D(N) \le \frac{2}{N} \sum_{\frac{N}{2} \le k < N} (1 + D(k))$$

There are two halves of the tree, each with the maximum-node side size of N/2 up to (N-1).

Treaps

We looked at randomly-build BSTs and balanced BSTs (AVL trees) today. There are many other trees. Another tree data structure that you may want to research is called a treap. A treap has one more word of information per node, which is simply a random number that is generated and stored into the node. When a new node has a random number that has higher priority than the root, we replace the current root with the new node and rip the tree apart so all items to the left of the new node go to the left and all items to the right of the new node go to the right.

Suppose we have the following treap and we insert 27 with a lower priority than the current root 14:

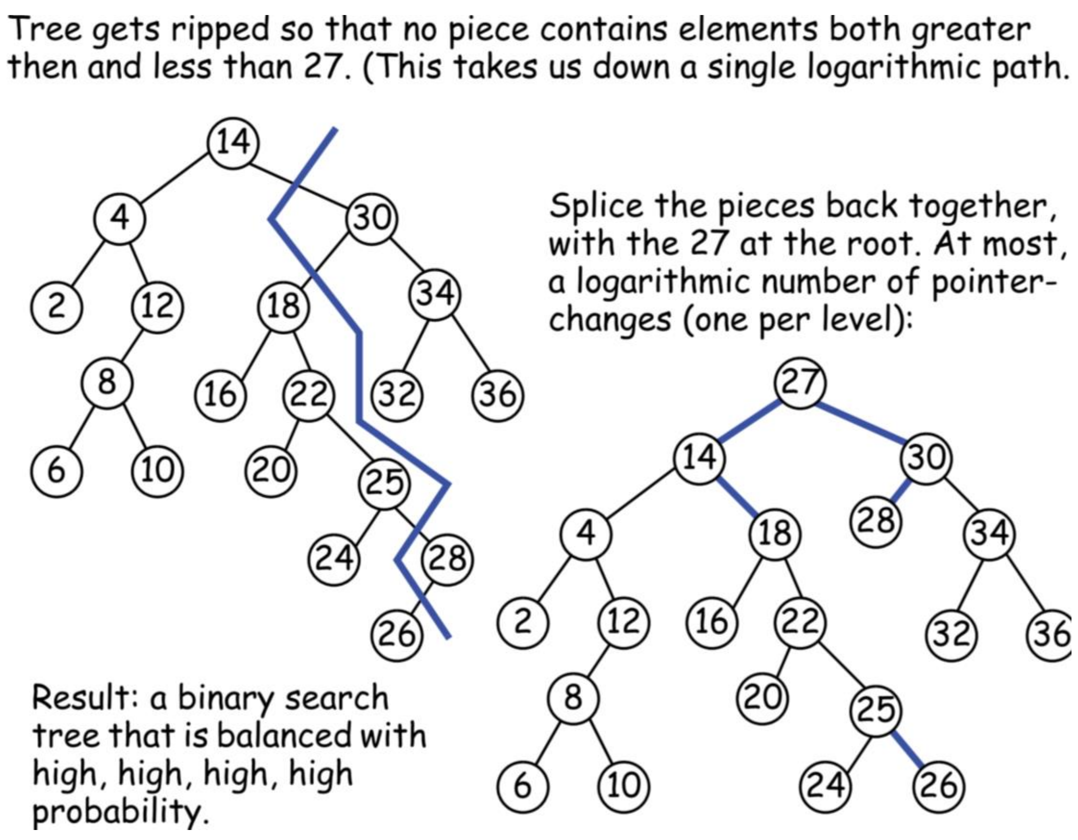

Tree gets ripped so that no piece contains elements both greater then and less than 27. (This takes us down a single logarithmic path.
Splice the pieces back together, with the 27 at the root. At most, a logarithmic number of pointer-changes (one per level):
Result: a binary search tree that is balanced with high, high, high, high probability.
14
4
30
2
12
18
34
8
16
22
32
36
6
10
20
25
24
28
26
27
14
30
4
18
28
34
2
12
16
22
32
36
8
20
25
6
10
24
26

Figure courtesy of Dr. Vegdahl

CS324: Red-Black Trees (BSTs with colored nodes)

February 28, 2020

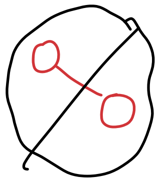
To do before next class:

- HW5 assigned (due Wed, Mar 18)
- Read chapter 15 before Monday, March 9
- Enjoy spring break!!

Today's learning outcomes:

- Learn properties of red-black trees
- Analyze operations (insert, delete, rotate)
- Repair red-black trees after insertions and deletions

A red-black tree is a binary search tree with one additional bit of information per node — its color (1=red, 0=black).



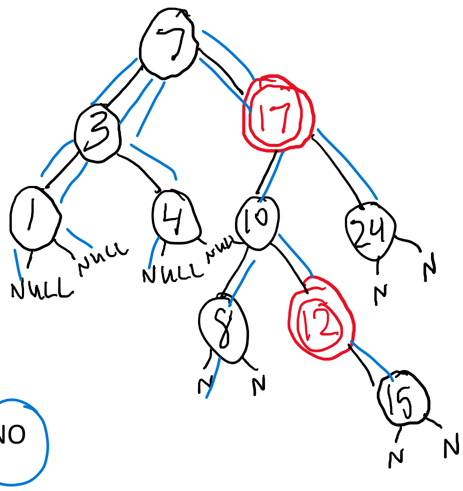
Color properties:

- Root node is always black
- A red node cannot be the parent or child of another red node (from root to leaf, cannot have two or more adjacent red notes along path)
- All the leaves are null trees and there are the same number of black nodes along all paths from the root to the leaves
- All the null leaves are colored black (note that we usually do not include these in the drawings to keep the drawings less cluttered)

Check for understanding:

1. Which of the following trees are legal red-black trees?

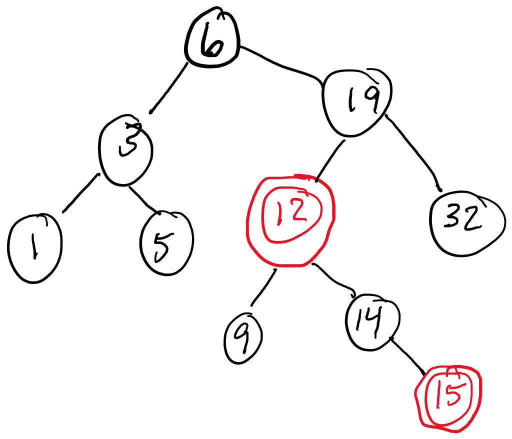
Tree A:



Legal? YES [✓ NO]

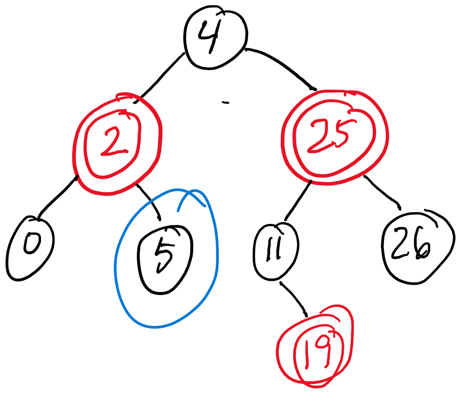
7-17-24 only 2 black nodes

Tree B



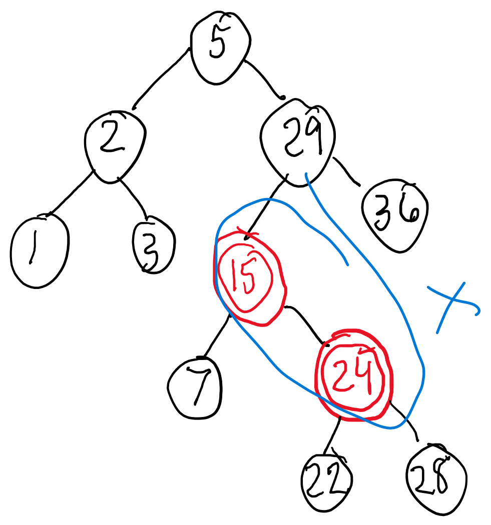
Legal? [✓ YES] NO

Tree C



Legal? YES [✓ NO]

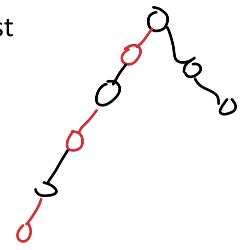
Tree D



Legal? YES [✓ NO]

2. What can you say about the length of the longest branch versus the length of the shortest branch for a red-black tree?

2x in length



3. Suppose the red-black tree contains N nodes. What is the maximum depth of the tree? 2·lgN

4. In terms of O-notation, what is the maximum depth of the tree? O(lgN)

5. Complete the chart:

Operation	Does it change the tree structure?	Runtime?
Find	No	O(lgN)
Insert (successful)	yes	Θ(lgN)
Insert (unsuccessful since key is already there)	No	O(lgN)
Delete (successful)	yes	Θ(lgN)
Delete (unsuccessful since key is not in tree)	No	Θ(lgN)

Note: the operations that do not change the structure of the tree operate just like a regular BST.

We need to handle the cases where the tree structure changes (insert successful, delete successful) in order to preserve the red-black tree color properties. Both of these may prompt rotations and recoloring of nodes.

INSERTION OF NEW KEY

6. Where do new keys get successfully inserted in a red-black tree? leaves

7. What color should the new node be? red

   a. If it is black, we will certainly make its path have a greater # of black nodes.
   b. If it is red, we may have a parent/child both be red.

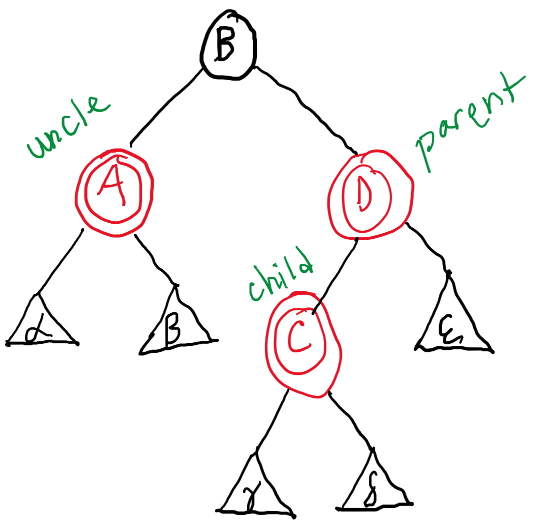
After we insert new node (as red) just above a NULL leaf, we need to check its parent's color. If its parent's color is black, what do we do? we are done!

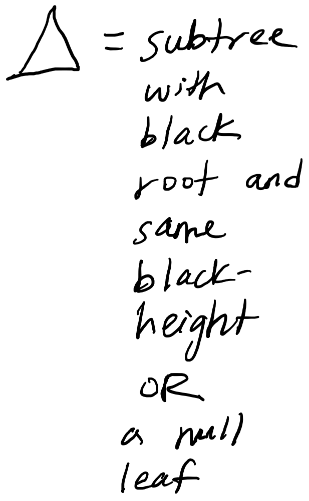
If its parent's color is red, what do we do? recolor / balance

6 cases to eliminate red-red ancestry (we will look at 3 cases, since the other 3 are mirror-images of these 3).

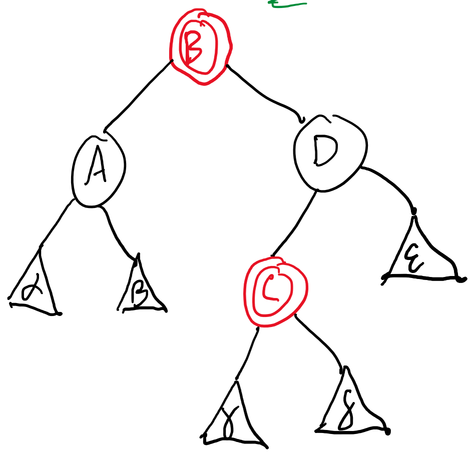
RED-RED ELIMINATION

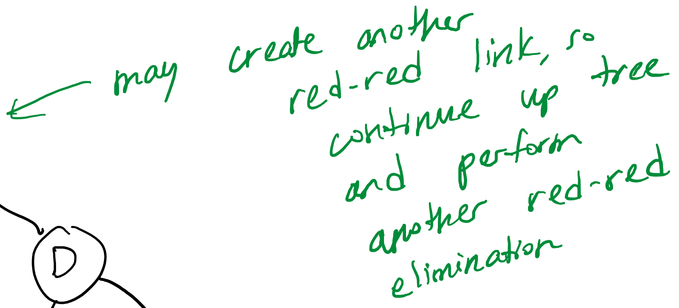
CASE A: uncle is red (re-color)





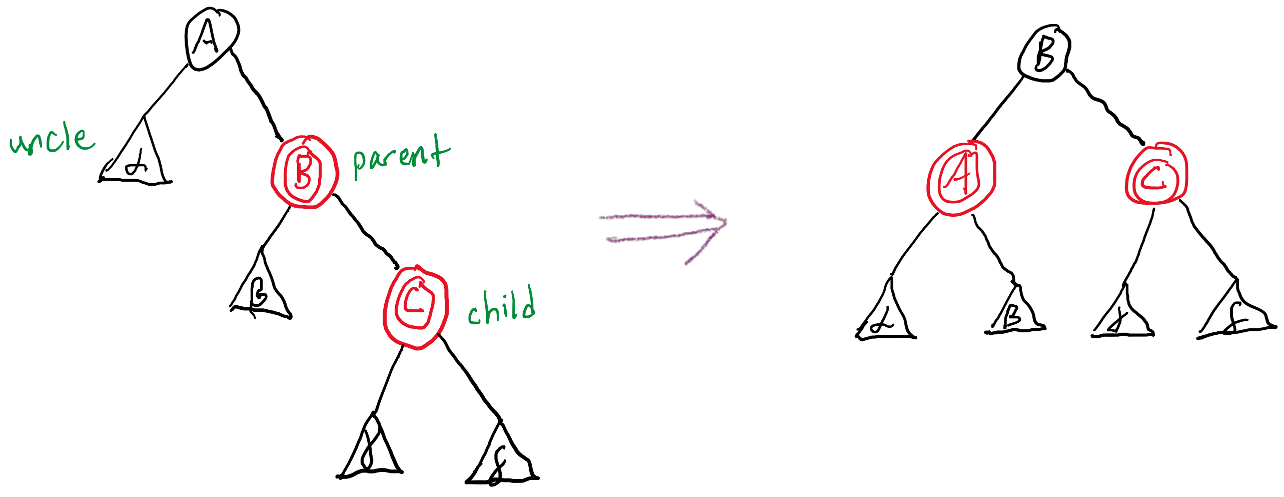
= subtree with black root and same black-height OR a null leaf




may create another red-red link, so continue up tree and perform another red-red elimination

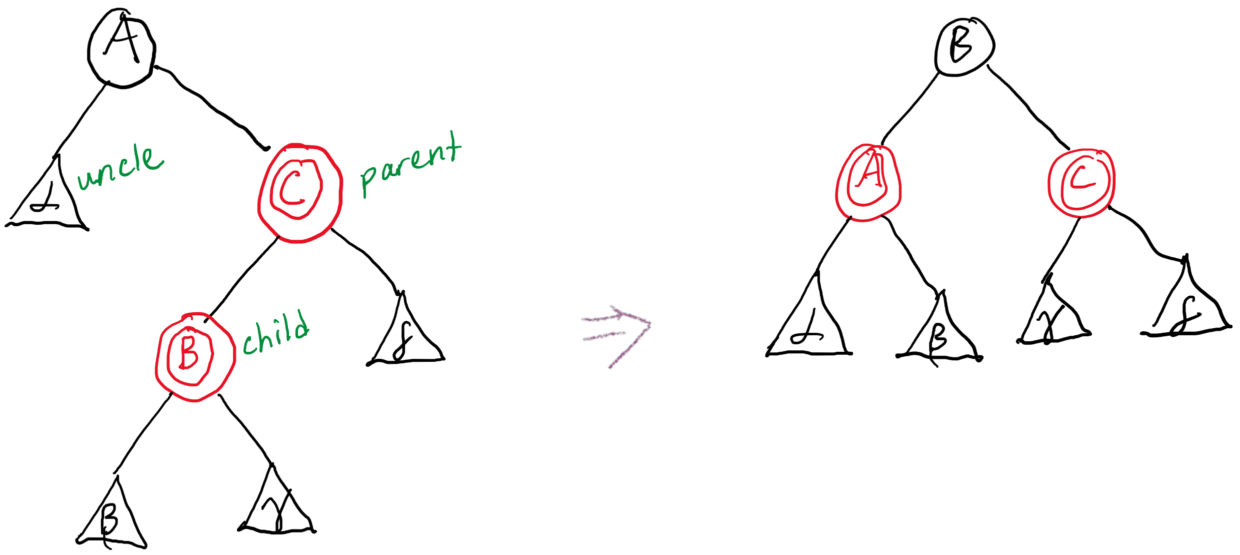
What changed in the tree? just colors

CASE B: uncle is black or null; no jig in path to node (do a single rotation)



What changed in the tree?

CASE C: uncle is black or null; jig in path to node (do a double rotation)



What changed in the tree?

The other 3 cases are mirror-images of these cases.

8. Do any of the insertions/rotations violate the BST ordering properties? no

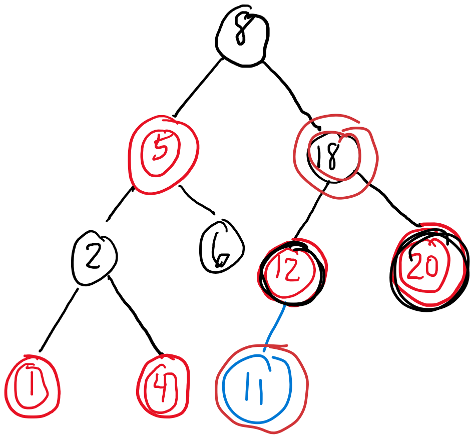
9. Do any of the insertions/rotations create a red-red node path (except possibly CASE A, but the updates will continue upward toward root)? no

10. Do any of the insertions/rotations change the black-height along any of the paths from root to null sub trees? no

11. How much work is done to update the trees? constant per level

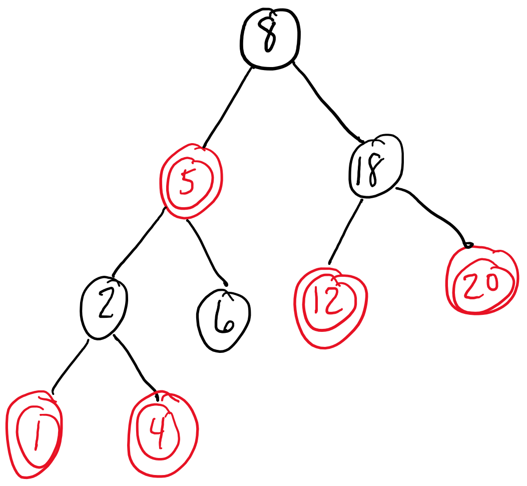
Practice with insertions:

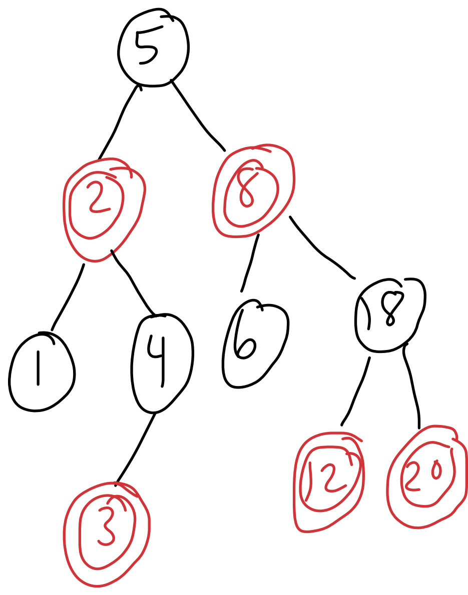
Insert new key 11 into the following red-black tree:



FINAL

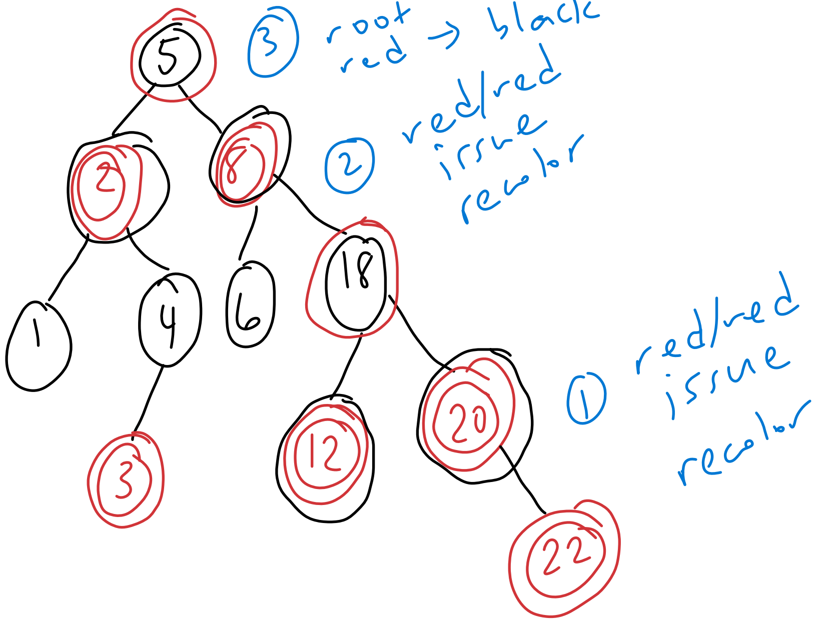
Insert new key 3 into the following red-black tree:

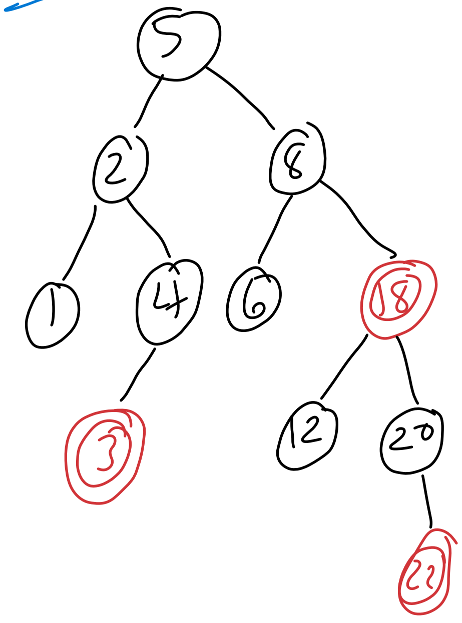




Keep going with tree above and insert new key 22.

FINAL





DELETING A KEY

What key location is the easiest to delete? leaf

What color at this location is the easiest to delete? red

What if we have just one key in the tree? null tree

So, we really need to consider the following situations:

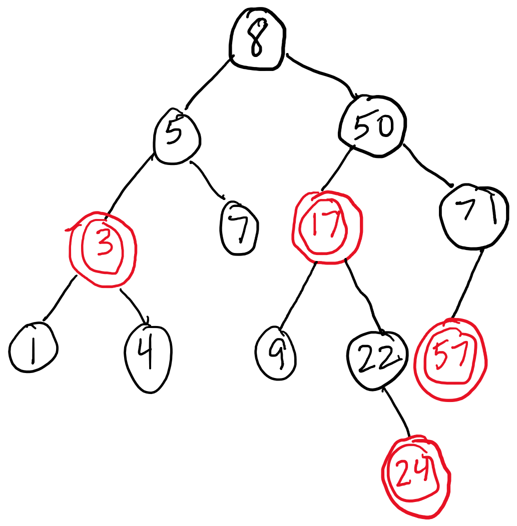
Key location	Color	Task
Leaf	Red	Just delete it (black-height property is maintained)
Leaf	Black	Apply transformation below to maintain black-height property
Semi-Leaf (1 null child, 1 non-null child)	Red	Cannot happen (prove on HW)
Semi-Leaf	Black	It's child must be red, otherwise it would violate the black-height property; Delete black node and recolor its red child to black
Interior	Red or Black	Swap with in-order successor, so it becomes a leaf; then apply one of the above cases

So, we will delete the node on the way down and rebalance on the way up in the tree.

Consider the following red-black tree. Suppose we want to delete 9. What happens to the black-height property?______________ decreases by one

Suppose we want to delete 50. What is the in-order successor of 50? _______ 57

Suppose we want to delete 8. What is the in-order successor of 8? ________ 9



DELETION TRANSFORMATIONS

These apply when we have a short black-height branch.

Flow-chart:

- Start at parent of the “short” node.
- Move down the sibling subtree.
- Look for the first red node.
    - If parent is red, do DELETE transformation #1
    - If sibling is red, do DELETE transformation #2
    - If either nephew is red, do DELETE transformation #3a or #3b (depending on which is red; if both are red, can apply either one)
    - Else, do DELETE transformation #4

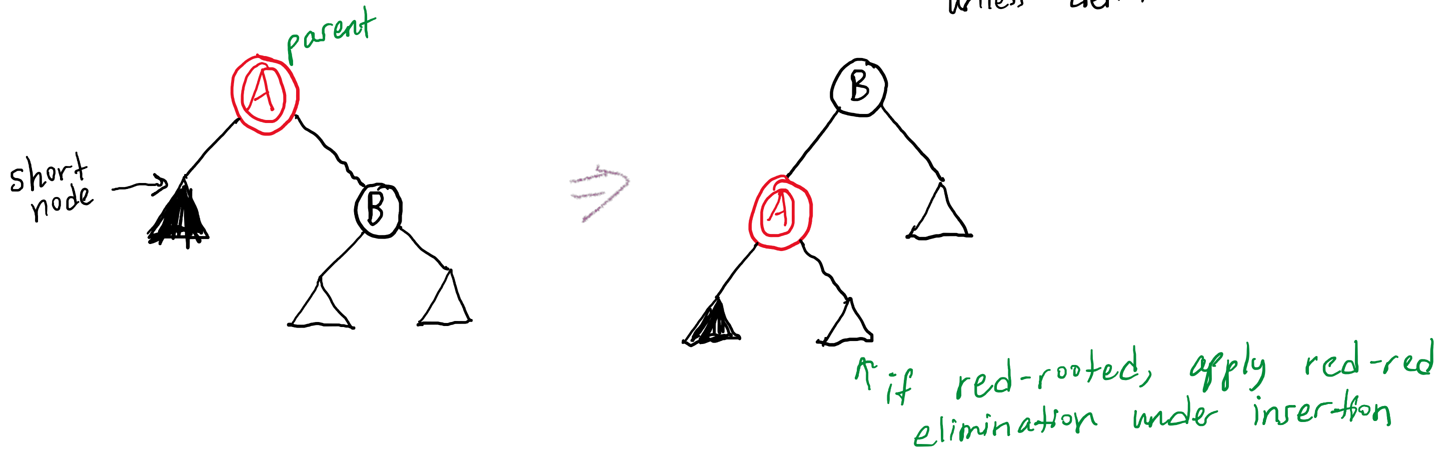
All these transformation pictures show the shortness being on the left. If the shortness is on the right, the pictures are a mirror-image.

▲ = null or black rooted subtree

△ = any null or non-null subtree

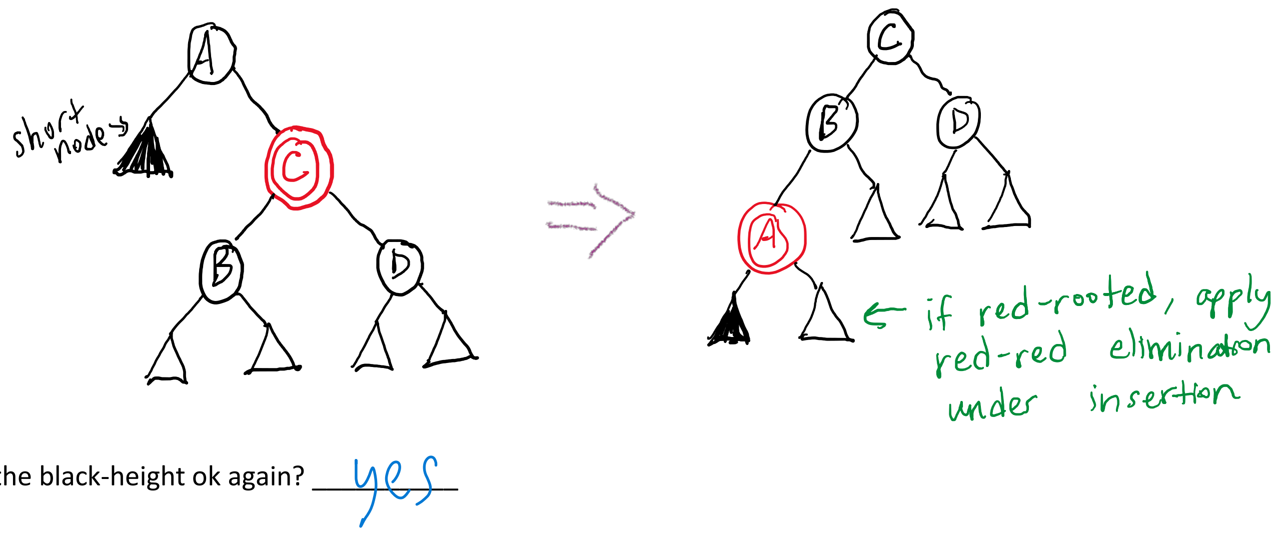
All triangles have same black-height unless denoted short node

#1: parent is red



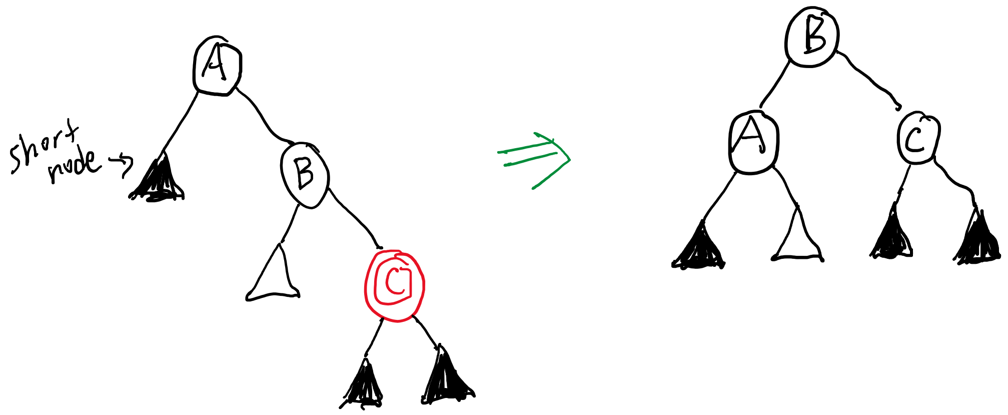
Is the black-height ok again? yes

#2: sibling is red



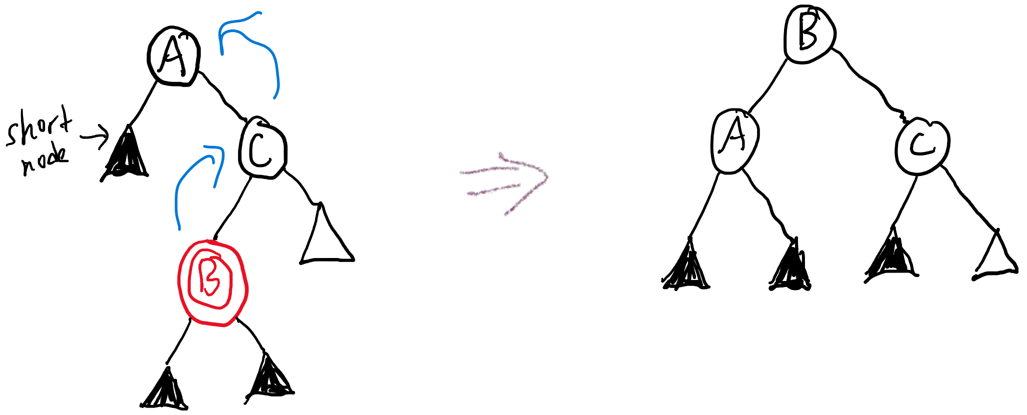
Is the black-height ok again? yes

#3a: right nephew is red



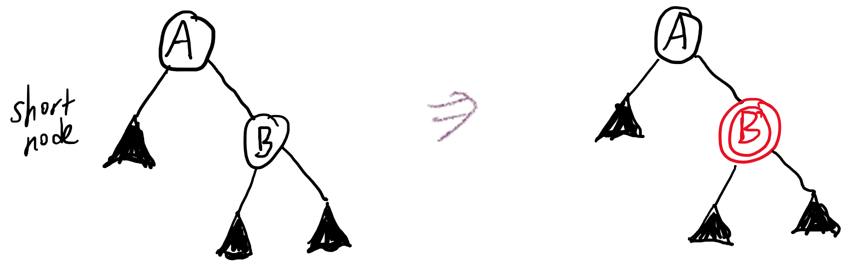
Is the black-height ok again? yes

#3b: left nephew is red



Is the black-height ok again? yes

#4: else (parent, sibling, and nephews are black)



propagate shortness transformation up tree from A

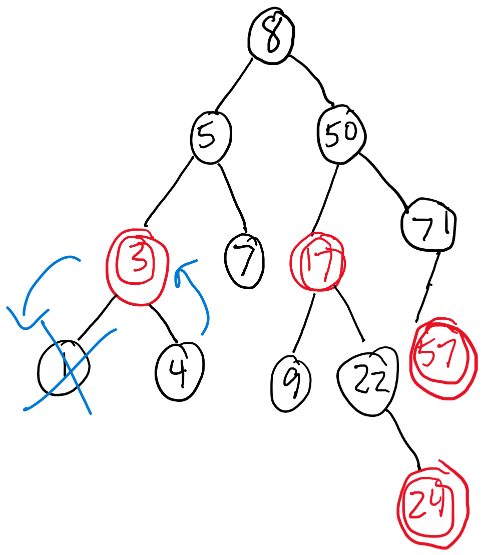
Note that this part of the tree has equal black-height but we reduced it by one, so that is why we need to propogate upward

How much work did we do for each of these transformations at each level? $\Theta(1)$

Note that the book does these transformations differently, so you have two models for red-black trees. The transformations in this lecture have fewer cases (thanks to Dr. Vegdahl).

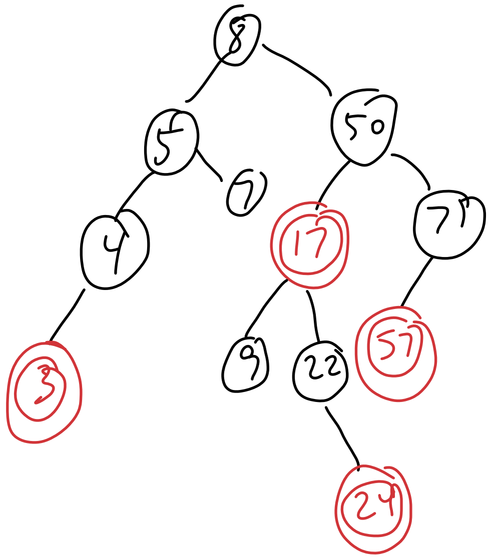
Practice:

1. Delete key 1 from the following tree:

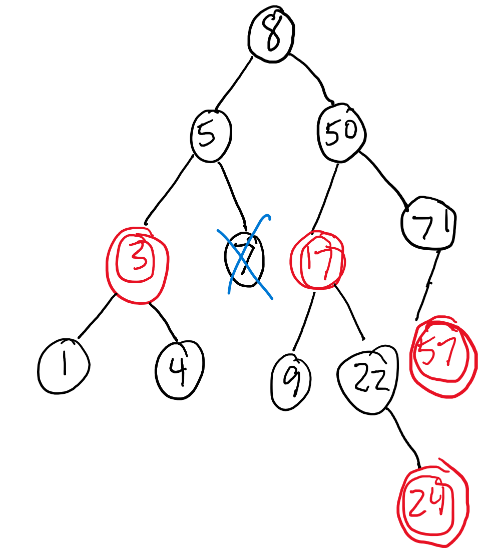


Note which case you are using.

case 1

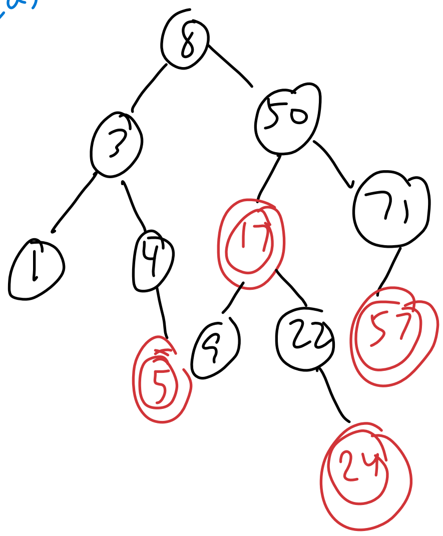


2. Delete 7 from the same tree:

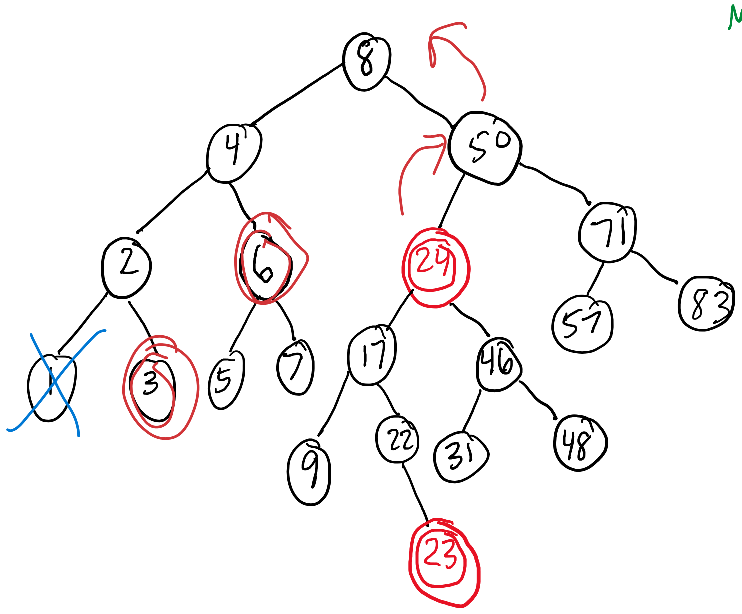


Note which case you are using.

case 2



3. Delete 1 from the following tree:

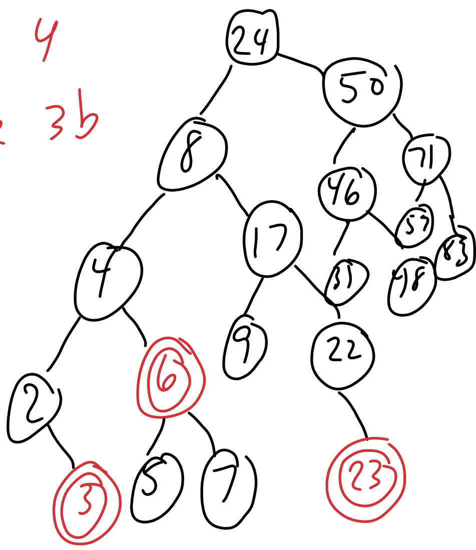


May need to propagate.

case 4

case 4

case 3b



Comparison to AVL (balanced BSTs)

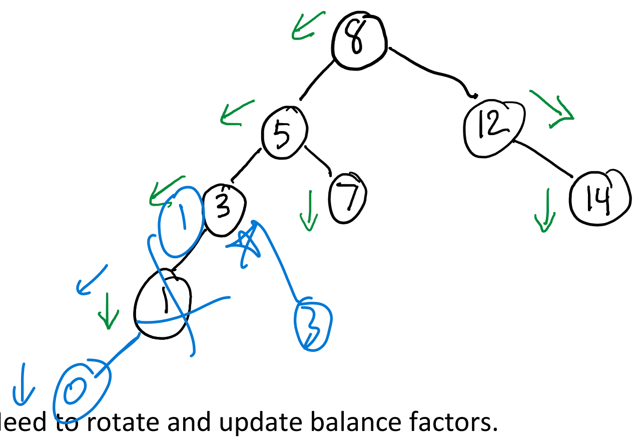
- AVL trees stay much more balanced, so lookups are generally a bit faster (but still lgN time) since the branch is not as deep.
- Rebalancing AVL trees happens more often, so insert and delete are a bit slower.
- AVL trees require 2 bits of information per node (1, 0, or -1 to store the balance factor) instead of 1 bit per node for the color of red-black trees.

We looked at AVL trees in the prior lecture and saw how to rotate when things get out of balance. Let's annotate the following tree with arrows to show the balance factor.

Down arrow = balanced

Left arrow = left side is longer

Right arrow = right side is longer



Insert 0 to the tree. Need to rotate and update balance factors.

CS324: Memoization and Dynamic Programming (part 1): Stone game and Matrix-chain multiplication

March 9, 2020

To do before next class:

- HW5 due Wed, Mar 18

Today's learning outcomes:

- Understand memoization process for recursive functions
- Know the properties of problems for which dynamic programming can be applied
- Understand how memoization (and therefore, dynamic programming) can be applied to a game and matrix-chain multiplication

---

We are going to start by playing a 2-person game.

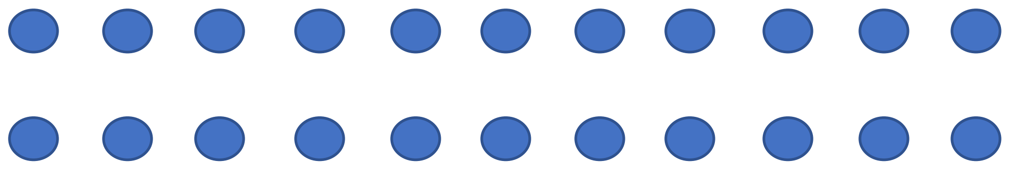
Example 1: Stones Game (based on the game Nim)

Find a partner. Choose one partner's sheet to play this game. Pictured below are 20 stones.

On your turn, you may cross out **one** or **two** stones. The player who crosses out the **last** stone **loses** the game. The player whose birthday is soonest goes first. Use player 1's sheet for the first game.

Player 1: ________________________________

Player 2: ________________________________



Play the game a second time with player 2 going first on player 2's sheet.

How could you determine the winner if you know the starting number of stones and the first player to play?

Here is a recursive algorithm for determining the number of stones to cross out to force a win. If there is no way to force a win, the algorithm returns 0.

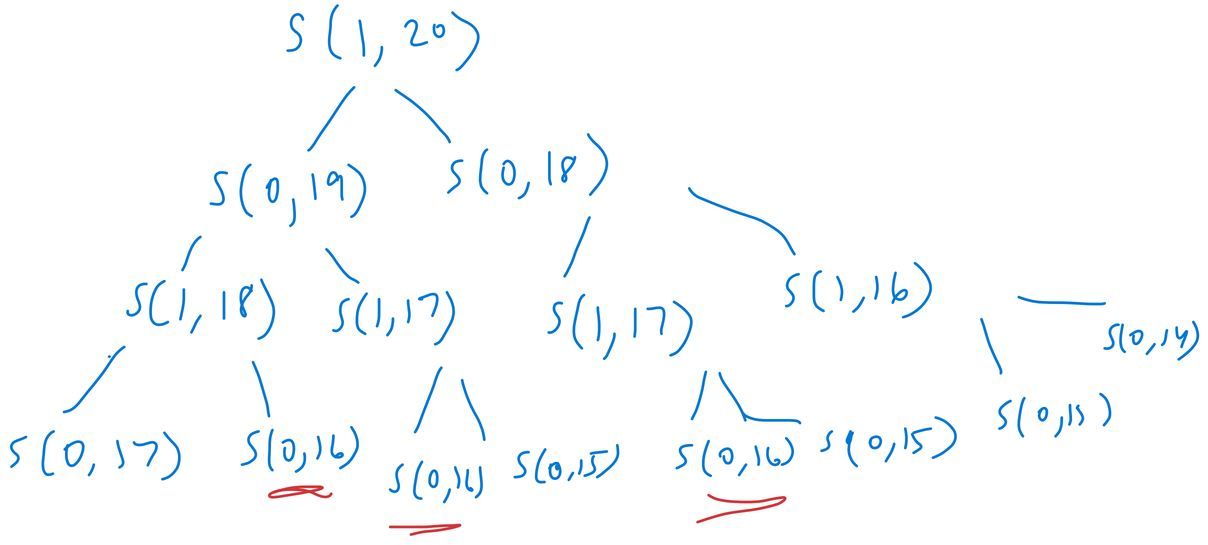
```
STONEWINNER(player, count):
      if count == 1
            return 0      // I lose since there is 1 stone left
      else if count == 2
            return 1      // I win by taking 1 stone, opponent must take last stone
★     else if STONEWINNER(opponent(player), count-1) == 0
            return 1      // I win by taking 1 stone, forcing opponent to lose
★     else if STONEWINNER(opponent(player), count-2) == 0
            return 2      // I win by taking 2 stones, forcing opponent to lose
      else
            return 0      // I cannot force a win
```

What is the recurrence for this recursive algorithm?

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

Does this look like Fibonacci? Yikes. What is the runtime of recursive Fibonacci? 2^n

Let's try drawing the recursion tree for a 20-stone game:



We end up having lots of nodes in the recursion tree with the same argument, just like we did with Fibonacci. What can we do instead of recomputing the sub-problem each time?

SOLUTION: Memoization

Memoization is a fancy word that means we record the answer to each recursive-call. Why? So we can just do a look-up instead of recomputing a copy of a function call. Hash-table to the rescue. We insert the value (player, count, result) into the hash table with the key being (player, count). For this updated version of the algorithm, we assume the hash table has the functions: lookup and insert. If lookup does not find the key, we assume it returns -1. If lookup finds the key, we assume it returns the result (one of 0, 1, or 2).

```
STONEWINNER_M(player, count):
        temp = lookup(player, count)      // hash table lookup
        if temp >=0
                return temp
        if count == 1
                result = 0    // I lose since there is 1 stone left
        else if count == 2
                result = 1    // I win by taking 1 stone, opponent must take last stone
        else if STONEWINNER_M(opponent(player), count-1) == 0
                result = 1    // I win by taking 1 stone, forcing opponent to lose
        else if STONEWINNER_M(opponent(player), count-2) == 0
                result = 2    // I win by taking 2 stones, forcing opponent to lose
        else
                result = 0    // I cannot force a win
        insert(player, count, result)
        return result
```

Now, we are guaranteed to not launch a recursive call tree if we have already computing the result.

Well, this was a lot of work for a constant-time solution for this stone game. But, it illustrates how memoization works.

How could you write the STONEWINNER algorithm such that the solution is constant-time?

count % 3 == 0

Example 2: Matrix-Chain Multiplication

Earlier in the course, we saw different ways to multiply two matrices together: iterative, recursive, and Strassen's. Let's do a little review of matrix multiplication in terms of matrix dimensions.

$$5\times 3 \quad \begin{bmatrix} 2 & 4 & 1 \\ 3 & 5 & 7 \\ 1 & 2 & 2 \\ -4 & 0 & 3 \\ 2 & 4 & 3 \end{bmatrix} \times \underset{3\times 4}{\begin{bmatrix} 1 & 2 & 4 & 2 \\ 5 & 3 & 0 & 2 \\ 0 & 1 & -3 & -2 \end{bmatrix}} = \begin{bmatrix} 0 & & & \\ & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

1. Given the matrices above, what is the size (in dimensions) of the resulting matrix? 5 x 4

2. How many (non-Strassen) multiplications do we need to do to get the final matrix product? 60 = 5 x 3 x 4

3. How many (non-Strassen) additions do we need to do to get the final matrix product? 5 x 2 x 4

OK, so multiplications dominate, so we will just focus the cost on that operator.

Suppose we are multiplying four matrices together: A x B x C x D.

4. Is matrix multiplication associative? yes

(Can we multiply (A x B) x (C x D) to get the product or any other parenthesized version?)

How many different ways can you put parentheses around these matrices A x B x C x D? Write them below.

A x (B x C) x D

((A x B) x C) x D

⋮

total of 5 ways

Now, suppose we are multiplying 3 matrices together (A, B, C).

- A is 10 x 5
- B is 5 x 21
- C is 21 x 8

A x B x C

(A x B) x C

A x (B x C)

How many multiplications are done if we do (A x B) x C?

(A x B) = 10 x 5 x 21 = 1050 multiplications with a matrix of size 10 x 21

((A x B) x C) = 10 x 21 x 8 = 1680 multiplications with a matrix of size 10 x 8

Total of 2730 multiplications

How many multiplications are done if we do A x (B x C)?

(B x C) = 5 x 21 x 8 = 840 w/ matrix 5 x 8

A x (B x C) = 10 x 5 x 8 = 400

Total of 1240 multiplications

Which one is the better approach (better == fewer multiplications)? $A \times (B \times C)$

OK, but how bad does this chaining get in terms of ways to chain the parentheses?

Suppose I have ten matrices. Put some set of parentheses around them so that the order is clear.

$$A\ x\ (B\ x\ (C\ x\ (D\ x\ E))\ x\ ((F\ x\ G)\ x\ ((H\ x\ I)\ x\ J)))$$

It is likely that everyone in the room made a different set of parentheses.

It turns out that the number of ways to parenthesize a matrix chain of N matrices is exponential with respect to N (lower bound is 2^N).

P = number of ways to parenthesize

$$P(1) = 1$$

$$P(n) = \sum_{1 \le k \le n-1} \underline{P(k)}\,\underline{P(n-k)}$$

$P(1) \cdot P(1) = 1$

$P(1)P(2) + P(2) \cdot P(1) =$
$1 + 1 = 2$

$P(1)P(3) + P(2)P(2) + P(3)P(1) =$
$2 + 1 + 2 = 5$

Let's fill in the chart for small values of N:

N	P(N)
1	1
2	1
3	2
4	5
5	14
6	42

11 58786

This grows according to the Catalan numbers (similar to Fibonacci, but worse in terms of fast growth!).

$$C_n = \left(\frac{1}{n+1}\right)\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

Yikes, so trying every possible way to put parens around a matrix chain even becomes very bad at N = 11. At 11, it is 58786.

Memoization to the rescue!!

Here is the recursive solution that does not save solutions. Assume arr contains the dimensions of the matrices, so arr is one longer than the number of matrices in the chain. If arr has:

[5, 3, 6, 4, …] then matrix A has dimensions 5 x 3, matrix B has dimensions 3 x 6, matrix C has dimensions 6 x 4, etc.

```
MINMULT(lowindex, highindex, arr):
    if lowindex >= highindex
        return 0            // we are done
    bestCost = INFINITY
    for i from lowindex to highindex-1:
        temp = MINMULT(lowindex, i, arr) + MINMULT(i, highindex, arr) +
            arr[lowindex]*arr[i]*arr[highindex]
        if temp < bestCost
            bestCost = temp
    return bestCost
```

We will be doing lots of repeated recursive calls with common sets of indices. Let's memoize this by saving results of function calls.

Here is the memoized version. The key will be (lowindex, highindex, arr) and the result will be the best cost for that chain.

```
MINMULT_M(lowindex, highindex, arr):
    temp = lookup(lowindex, highindex, arr)    //lookup in hash table
    if temp >= 0
        return temp
    if lowindex >= highindex
        result = 0          // we are done
    else
        bestCost = INFINITY
        for i from lowindex to highindex-1:
            temp = MINMULT_M(lowindex, i, arr) + MINMULT_M(i, highindex, arr)+
            arr[lowindex]*arr[i]*arr[highindex]
            if temp < bestCost
                bestCost = temp
        result = bestCost
        insert(lowindex, highindex, arr, result)  //insert into hash table
  ←     return result
```

Are we doing better than exponential running time? Let's see.

How many non-memoized calls will we have at most for indices 1 through N over the entire recursion? Think of a table with row being lowindex and col being highindex.

___________ n^2

How much work is done per function call? ____ $\Theta(1)$ ____

Total runtime of memoized version: ____ $\Theta(n^2)$ ____

Let's do an example where we unwind the recursive calls to see how it works:

startindex
lastindex

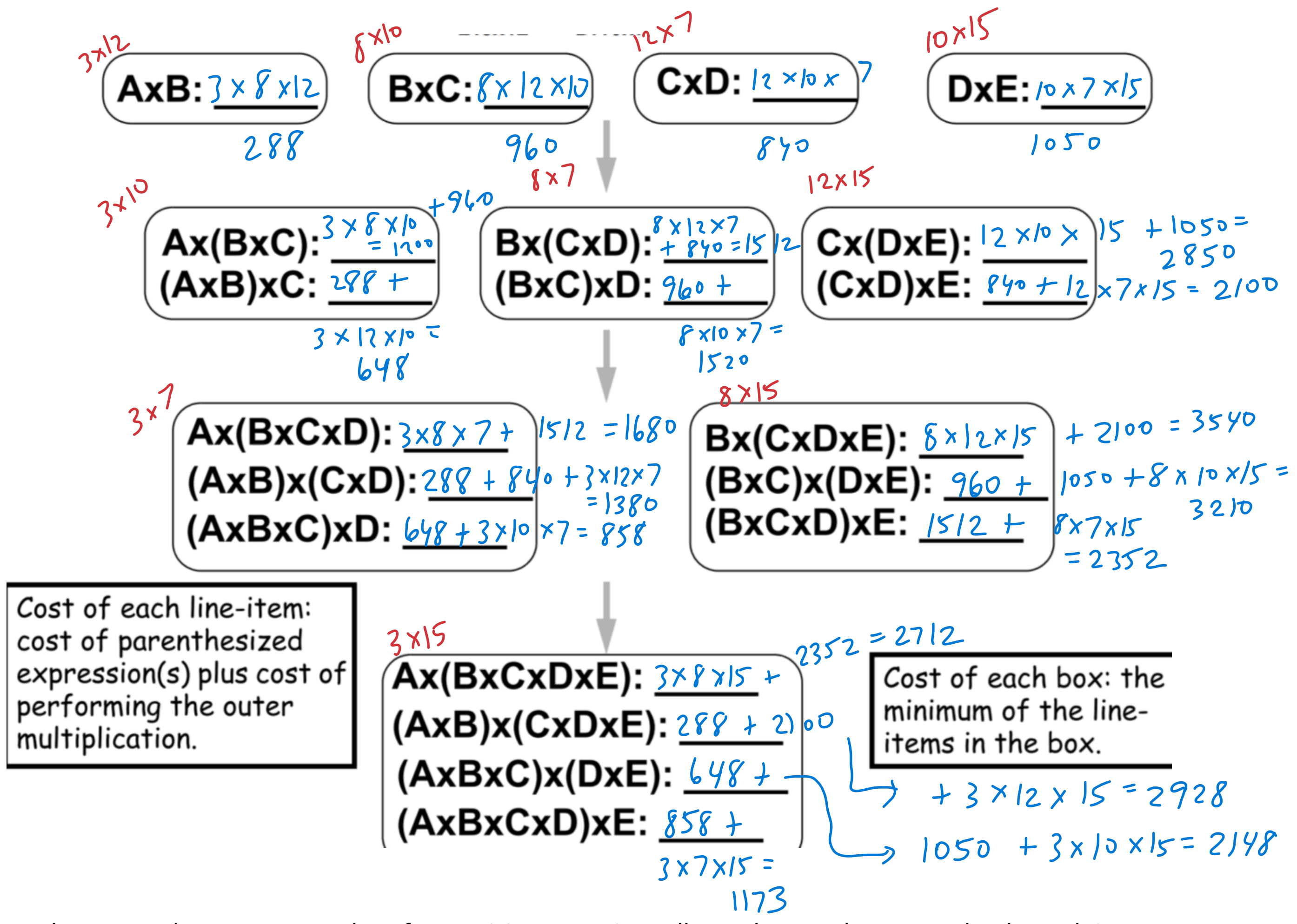
1,2	2,3
1,3	2,4
1,4	2,5
1,5	⋮
1..n	2...n

• matrix dimensions

• calculations

A is a 3x8 matrix
B is a 8x12 matrix
C is a 12x10 matrix
D is a 10x7 matrix
E is a 7x15 matrix

Therefore, arr = [3, 8, 12, 10, 7, 15]. Example courtesy of Dr. Vegdahl.


3x12
AxB: 3 x 8 x12
288
8x10
BxC: 8x 12 x10
960
12x7
CxD: 12 x10 x 7
840
10x15
DxE: 10 x 7 x15
1050
3x10
Ax(BxC): 3 x 8 x 10 = 1200 +960
(AxB)xC: 288 +
3 x 12 x10 = 648
8x7
Bx(CxD): 8 x 12 x 7 + 840 = 1512
(BxC)xD: 960 +
8 x10 x 7 = 1520
12x15
Cx(DxE): 12 x10 x 15 + 1050 = 2850
(CxD)xE: 840 + 12 x 7 x 15 = 2100
3x7
Ax(BxCxD): 3x8 x 7 + 1512 = 1680
(AxB)x(CxD): 288 + 840 + 3x12x7 = 1380
(AxBxC)xD: 648 + 3x10 x7 = 858
8x15
Bx(CxDxE): 8 x 12 x 15 + 2100 = 3540
(BxC)x(DxE): 960 + 1050 + 8 x 10 x 15 = 3210
(BxCxD)xE: 1512 + 8x7x15 = 2352
3x15
Ax(BxCxDxE): 3x8 x15 + 2352 = 2712
(AxB)x(CxDxE): 288 + 2100 + 3 x 12 x 15 = 2928
(AxBxC)x(DxE): 648 + 1050 + 3 x 10 x 15 = 2148
(AxBxCxD)xE: 858 + 3 x7x15 = 1173

Cost of each line-item: cost of parenthesized expression(s) plus cost of performing the outer multiplication.

Cost of each box: the minimum of the line-items in the box.

We have now done two examples of memoizing recursive calls, so that we do not need to keep doing the same work.

1173 is min

Bottom-Up Memoization: Dynamic Programming

Dynamic programming takes a bottom-up approach to filling in a table of values that define optimal solutions to sub-problems. There are two properties that a problem must have in order to employ dynamic programming:

1. **Optimal substructure**: an optimal solution to the subproblems can easily be combined to form an optimal solution for the entire problem.
2. **Overlapping subproblems**: subproblems (and subsubproblems and subsubsubproblems etc.) must result in lots of repetitions of the same problems. Otherwise, memoization or filling in a table does not help us much.

Note: dynamic programming is similar to divide-and-conquer in that recursion is employed to solve subproblems. However, divide-and-conquer algorithms split a problem into *independent* subproblems, where the solutions can then be combined. Dynamic programming algorithms have overlapping subproblems in which the same subproblem solution is computed over and over and we save time by storing solutions to subproblems.

Check for understanding:

1. Can dynamic programming be employed for minimizing matrix-chain multiplications?
    a. Does it have optimal substructure? [✓ YES] NO
    b. Does it have overlapping subproblems? [✓ YES] NO

2. Can dynamic programming be employed for minimal graph coloring of N nodes (graph coloring is a solution in which each node is assigned a color and adjacent nodes have different colors, optimal graph coloring is finding the minimum number of colors needed to color all the nodes in the graph)?
    a. Does it have optimal substructure? YES [✓ NO]
    b. Does it have overlapping subproblems? YES [✓ NO]

3. Can dynamic programming be employed for sorting an array? merge sort
    a. Does it have optimal substructure? [✓ YES] NO
    b. Does it have overlapping subproblems? YES [✓ NO]

4. Can dynamic programming be employed for finding the lonest common substring in two strings?
    a. Does it have optimal substructure? [✓ YES] NO
    b. Does it have overlapping subproblems? [✓ YES] NO

5. Can dynamic programming be employed for finding the longest common subsequence in two strings?
    a. Does it have optimal substructure? [✓ YES] NO
    b. Does it have overlapping subproblems? [✓ YES] NO

6. Can dynamic programming be employed for finding the maximum-value items to be placed in a maximum-weight knapsack (each item has a weight and a value), this is the 0-1 knapsack problem?
    a. Does it have optimal substructure? [✓ YES] NO
    b. Does it have overlapping subproblems? [✓ YES] NO

We will focus on longest common subsequence in the next lecture. You will implement the dynamic programming solution for the 0-1 knapsack problem in the following lecture.

CS324: Dynamic Programming (part 2): Manhattan Graph and Longest Common Subsequence

March 11, 2020

To do before next class:

- HW5 due Wed, Mar 18
- Read lecture (lab) for March 13 prior to class; you will work on the lab during class on Friday

---

Today's learning outcomes:

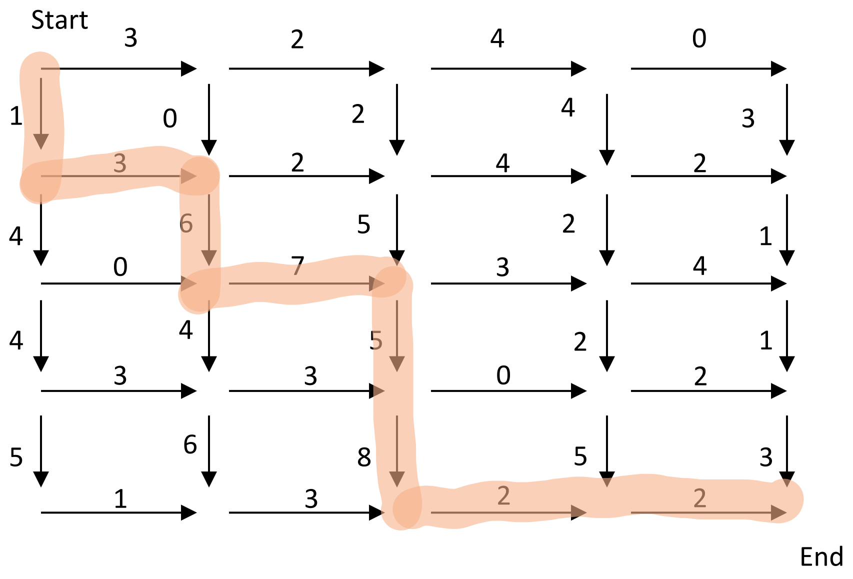
- Understand the Manhattan and LCS problems
- Apply dynamic programming (bottom-up filling in tables) to find optimal solutions

---

Example 1: Manhattan Problem

Manhattan Problem: Find the highest total weight path in a weighted graph (DAG on grid) from a start node to an end node.

Graph is like blocks in the city – can travel east and south. Each edge on the block has a weight (cost).



1. Find the path with the highest total weight from Start to End and highlight it.

2. Does this problem have optimal substructure? [✓ YES] NO

3. Does this problem have overlapping subproblems? [✓ YES] NO

Let's label each vertex as [i][j] where Start is at position [0][0] and the one to the right of start is position [0][1]. The max total weight from the start to vertex at [i][j] is stored as in table as v[i][j].

We'll create a table that contains the highest scoring path to each of the vertices:

V: Table of Costs from Start to each Vertex (First row completed):

0	3	5	9	9

The top row is simply the cost along the top horizontal path.

V: Table of Costs from Start to each Vertex (Left column and first row completed):

0	3	5	9	9
1	4 ←	7 ↑	13 ↑	15 ←
5	10 ↑	17 ←	20 ←	24 ←
9	14 ↑	22 ↑	22 ← ↑	25 ↑
14	20 ↑	30 ↑	32 ←	34 ←

The left column is simply the cost along the left vertical path.

Now, to fill in the rest of the table:

$$v[i][j] = \max \begin{cases} v[i-1][j] + \text{weight of edge from } v[i-1][j] \text{ to } v[i][j] & \text{// from top} \\ v[i][j-1] + \text{weight of edge from } v[i][j-1] \text{ to } v[i][j] & \text{// from left} \end{cases}$$

4. Final Table (complete the entries above in the table using the above rule)

Manhattan Algorithm:
Input: G is a Manhattan graph of n nodes wide and m nodes tall with start node at [0][0] and end node at [m-1][n-1], weights are stored for each edge

```
MANHATTAN(G):
    v[0][0] = 0
    for j = 1 to (n-1): // fill in top row
        v[0][j] = v[0][j-1] + weight([0][j-1] to [0][j])
    for i = 1 to (m-1): // fill in left column
        v[i][0] = v[i-1][0] + weight([i-1][0] to [i][0])
    for i = 1 to (m-1): // fill in middle cells
        for j = 1 to (n-1):
            v[i][j] =   max  { v[i-1][j] + weight([i-1][j] to [i][j])
                             { v[i][j-1] + weight([i][j-1] to [i][j])

    return v[n-1][m-1]  // highest scoring path
```

5. What is the running time of the Manhattan Algorithm assuming the graph is n nodes wide and m nodes tall?

Θ(___n · m___)

Example 2: Longest Common Subsequence

Suppose we have the following two sequences of uppercase letters (order matters here):

ABIGCATINTHEHAT
THEBIGMONSTERTRUCK

1. A **subsequence** of the top sequence is GANTH. Provide another subsequence of the top sequence: ___BHT___

2. A **common subsequence** of two sequences is a subsequence of both sequences. Can you find a **common subsequence** of the two sequences above: ___TIN___

3. The **longest common subsequence** of two sequences is a subsequence whose length is maximal of all possible common subsequences. Can you find the LCS of the two sequences above? ___BIGNTET___

4. Underline the common subsequence BIGNTT in the strings below:

A<u>BIG</u>CATI<u>N</u>THEHA<u>T</u>

THE<u>BIG</u>MO<u>N</u>S<u>T</u>ER<u>T</u>RUCK

Problem Definition: Given two strings S and T, what is the longest common subsequence between the two strings?

(Application: In biology, this is important if we want to find the longest sequence of DNA that is exactly the same and in the same order in two DNA strings.)

Assume `S = S[1…q]` //note: using 1-indexing for strings
Assume `T = T[1…p]`

Let's think about the cases if we want to solve `LCS_length` recursively:

- `if S is empty or T is empty, return 0`
- `i is last index of S; j is last index of T` return 0
- `if S[i] == T[j], return 1 + LCS_length(S[1..i-1], T[1..j-1]) // remove one char`
- `if S[i] != T[j], return max(LCS_length(S[1..i-1], T[1..j]), LCS_length(S[1..i], T[1..j-1]) // remove just one char from either string`

Note: this solution returns the <u>length</u> of the LCS. If we want to return the actual LCS, then we can update the recursive solution to return the empty string for the base case, append a char to the longest LCS in the second case, and return the longer of the two strings in the third case.

5. Does this problem have optimal substructure? [✓ YES] NO

6. Does this problem have overlapping subproblems? [✓ YES] NO

7. Do we solve lots of the same subproblems over and over? [✓ YES] NO

We can use dynamic programming. We need to build a table and think about how to construct the LCS bottom-up. We will use two tables to keep track of things. Table `c` will keep track of the length of the LCS. Table `d` will keep track of directions, so we can reconstruct the LCS. Note that this algorithm is slightly different than the textbook.

```
LCS_DP(S[1..q], T[1..p]):
    Create 2D tables c and d with q+1 rows and p+1 columns
    c[0][0] = 0                 // no alignment
    d[0][0] = "F"               // start location of LCS
    for i from 1 to q           // fill in left column
          c[i][0] = 0
          d[i][0] = "F"
    for j from 1 to p           // fill top row
          c[0][j] = 0
          d[0][j] = "F"
    for i from 1 to q           // fill in table left to right, top to bottom
          for j from 1 to p
                if T[i] == S[j] // have a match
                      c[i][j] = c[i-1][j-1] + 1
                      d[i][j] = "D"             // diagonal
                else if c[i-1][j] >= c[i][j-1]  // from above
                      c[i][j] = c[i-1][j]
                      d[i][j] = "T"             // top
                else                            // from left
                      c[i][j] = c[i][j-1]
                      d[i][j] = "L"             // left
    return c[q][p]
```

This will return the length of the longest LCS.

Practice:
S = "CGACAG"
T = "AGATCAC"

Use the dynamic programming algorithm to complete the tables c and d below.

Table 3: Complete the cost table for LCS (strings are not actually stored in the tables, but are there to help you process tables)

		String T	A ✓	G	A	T	C	A	C
0	String S	0	0	0	0	0	0	0	0
1	C ✓	0	0	0	0	0	1	1	1
2	G	0	0	1	1	1	1	1	1
3	A	0	1	1	2	2	2	2	2
4	C	0	1	1	2	2	3	3	3
5	A	0	1	1	2	2	3	4	4
6	G	0	1	2	2	2	3	4	4 ✩

D
T
L

✩ LCS length in this box

Table 4: Complete the direction table

		A	G	A	T	C	A	C
	F	F	F	F	F	F	F	F
C	F	T	T	T	T	D	L	D
G	F	T	D	L	L	T	T	T
A	F	D	T	D	L	L	D	L
C	F	T	T	T	T	D	L	D
A	F	D	T	D	T	T	D	L
G	F	T	D	T	T	T	T	T

8. What is the length of the longest LCS? ____4____

9. How do we get the LCS from the direction table? Start at d[q][p] and follow directions until you get to a box with the value "F". Whenever you see a "D", append the corresponding char from S or T to the front the growing string. The "T" and "L" directions just move you through the table.

LCS = ""

GACA

10. What is the running time of the LCS dynamic programming algorithm to complete the tables with a string S of q chars and string T of p chars?

Θ($p \cdot q$)

11. How much storage space did we use for the dynamic programming algorithm? $(p+1)(q+1)$ boxes per table

12. What is the running time to construct the LCS given table d? Θ($p + q$)

13. If we just want to know the length of the LCS (and not construct it), how many rows of the table do we need to keep at any point in time? 2 prev row current row

Some dynamic programming tips:

- Be sure that the problem has **optimal substructure** and **overlapping subproblems** that will be re-computing many times.
- You can always think about solving the problem recursively first.
- Then, unroll the recursion to determine how you could fill in a table in a bottom-up fashion.
- Memoization works, too (putting already computing results in a hash table), but the constant factor is usually lower for dynamic programming solutions (values are already in the table, so no hash lookups are necessary). If the problem needs to solve all subproblems (aka, a completely filled table) to get the optimal solution, then the dynamic programming table will be completely filled anyway and will be faster than memoization.
- These are popular technical interview questions, so if you see that you are solving a problem recursively which will recursively call the same subproblems over and over, you may want to turn it into a dynamic programming solution. It is likely the recursive version is an exponential time solution and the dynamic programming version is a poly time solution.
- Dynamic programming problems are popular programming competition problems.
- DP is a wonderful tool to have in your toolbox for algorithm design.

CS324: Practice with Dynamic Programming (0-1 Knapsack)

March 13, 2020

To do before next class:

- HW5 due Wed, Mar 18
- This lab due Fri, Mar 20 if not finished in class
- Read CLRS chapters 16 and 22 before Monday

Today's learning outcomes:

- Recognize optimal sub-structure and overlapping problems in a recursive solution
- Convert a recursive solution to dynamic programming solution
- Implement recursive and dynamic programming solutions to the 0-1 knapsack problem
- Analyze recursive and dynamic programming solutions

Note: Tammy is away at a conference for this class session, so you will complete this lab during the regular class period.

You may work on this lab with a partner, if you wish.

Due: Friday, March 20
Submit your code to moodle and submit an electronic write-up of this lab to the separate link on moodle. If you work with a partner, just one partner needs to submit to moodle.

Name(s): ____

This lab is based on a popular interview coding question, so this may come in handy during your technical interviews. It is also a common type of question you may see on the ACM-ICPC contest.

Your basic task is to design and implement a solution to the 0-1 knapsack problem. The lab will help guide you through the design and implementation. You may implement the solution in any programming language.

Problem: Given a list of items (each with a weight and value) and given a max-weight for the knapsack, what is the maximum *total value* of items such that the *total weight* of the items is less than or equal to the max-weight of the knapsack?

To keep this simple, we will assume the weights and values are integers.

For example, if we have three items (w = weight, v = value):

Item 0: (w 10, v 60)
Item 1: (w 20, v 100)
Item 2: (w 30, v 120)

and the knapsack has a max-weight of 50, answer the following questions:

1. Can we put all three items in the knapsack? YES [✓ NO]

2. Can we put item 0 and item 1 in the knapsack? [✓ YES] NO

3. Which set of items maximizes the total value with total weight <= max-weight?
   - a. 0 and 2
   - b. 0 and 1
   - [✓ c. 1 and 2]

Now, let's think about how we can solve this problem recursively. For each item, we have two choices: add it to the knapsack or leave it out of the knapsack. We can return the maximum of these two choices along with a recursive call to the rest of the items. So, that's it. In the pseudocode below, note that the lists will be 0-indexed, since the programming language you choose is likely 0-indexed. [For most of the course, we have been writing pseudo-code with 1-indexed lists.]

Parameters:

```
w = max-weight of knapsack
weights = weights of items in the list
vals = values of items in the list
len = length of weights and vals
index = index within the list
```

```
KNAPSACK_R(w, weights, vals, len, index):
1       If index >= len
2               Return 0
3       If w - weights[index] < 0
4               Return KNAPSACK_R(w, weights, vals, len, index+1)
5       Return Max(KNAPSACK_R(w - weights[index], weights, values, len, index+1) + values[index],
6                   KNAPSACK_R(w, weights, values, len, index+1))
```

In order to start the recursive process at index 0, we can define KNAPSACK as follows:

```
KNAPSACK(w, weights, vals, len):
1       // Error-check that w and len are >= 0
2       // Error-check that weights and vals are the same length with len
3       // Error-check that weights[i] and values[i] are >= 0
4       // If something is in error, return 0
5       Return KNAPSACK_R(w, weights, vals, len, 0)
```

4. What does line 5 in KNAPSACK_R represent?
   - [✓ a. Including the item in the knapsack]
   - b. Excluding the item in the knapsack

5. Why do we make the line 5 recursive call to (w – weights[index]) and also add values[index] to the returned result?

_________To account for the weight added to the knapsack and the overall value of the items_

6. What does the base case in line 3 of KNAPSACK_R represent?

______Weight of item is too large to be added to knapsack_________________________________

Implement these two functions. Again, remember that the pseudo-code uses 0-indexed lists. Also, if your programming language includes length as a member of the list data structure, you do not need to pass len into the functions.

Implement a main function that creates the following items and max-weight:

Item 0: (w 1, value 6)
Item 1: (w 2, value 10)
Item 2: (w 3, value 12)
Item 3: (w 1, value 7)

Max-weight = 5

Then, test your recursive solution on this set of items.

7. What is the best set of items of this set? ___0, 2, 3________________

8. What does running your code return for the maximum value (be sure to print the total in main, so you can see it)? ___25_____

9. Suppose there are N items to process. What is the big-theta runtime of the recursive version? (Hint: how many different possible sets of items can you put in the knapsack, power set comes in handy)

__$\Theta(2^N_)$___________

Let's see if we can turn this recursive version into a dynamic programming solution. Remember that dynamic programming solutions are possible if two conditions are met:

1. Does the problem have optimal sub-structure?
2. Does the problem have overlapping sub-problems?

Both of these questions are "yes" for the 0-1 knapsack problem, so we can turn the recursive solution into a bottom-up solution that uses dynamic programming. So, we need to figure out how to build this up from the bottom of the recursion. If we have 0 items, then we have a solution of 0 total value for any max-weight. If we have a max-weight of 0, we have a solution of 0 total value for any number of items. We will build a 2D table where the max-weight is represented as column indices and the item-number is represented as row indices.

Suppose we have the following items and max-weight:

Item 0: (w 1, value 6)
Item 1: (w 2, value 10)
Item 2: (w 3, value 12)
Item 3: (w 1, value 7)

Max-weight = 5

Then, we build a table with 6 columns (1 + max-weight) and 5 rows (1 + number of items). The bold numbers are just the indices – they are not stored in the table themselves.

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

The top row of 0's represents putting no items in the knapsack across max-weights from 0 to 5.
The left column of 0's represents putting more and more items in the knapsack with max-weight of 0.

How do we fill in the table? `i` is the row index and `j` is the column index.

10. What if the weight of item `i-1` has weight greater than `j`? Can we add it to the knapsack?

YES [✓ NO]

In that case, we just copy the value in the row above in the table, keeping what is already in the knapsack and not adding any weight into the knapsack.

What if the weight of item `i-1` has weight <= `j`? We have two choices as before – do not add the item to the knapsack or add the item to the knapsack. Just as before, if we do not add the item to the knapsack, we can use the table entry from above. If we add the item to the knapsack, we use the table entry from the `row above` and the column `j-weights[i-1]` and add it to `vals[i-1]`. We choose the max of these two values (see algorithm below).

So, our dynamic programming solution is as follows:

```
KNAPSACK_DP(w, weights, vals, len):
        T[len+1][w+1] is a new table of ints
        Initialize row 0 and col 0 of T to 0
        For i = 1 to len:
                For j = 1 to w:
                        If weights[i-1] > j
                                T[i][j] = T[i-1][j]
                        Else
                                T[i][j] = max(T[i-1][j], T[i-1][j-weights[i-1]] + vals[i-1])
        Return T[len][w]
```

11. Execute the dynamic programming solution to fill in the table below on the set of 4 items from above with max-weight of 5.

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22
4	0	7	13	17	23	25

Implement the dynamic programming solution and call it on the same data you already have in main. Print out both the recursive solution and the dynamic programming solution, as follows:

```
Total recursive: 25
Total dp: 25
```

12. What does your dynamic programming solution return? ____25________

Add code to the dynamic programming function after the table is filled in to print out the table. Include your table data for the four items with max-weight of 5.

13. Put your results below and ensure they match what you hand-executed above.

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22
4	0	7	13	17	23	25

14. What is the runtime of the dynamic programming version of the knapsack solution? (Hint: how big is the table and how much work do you do to fill in each entry?)

Θ(___N*W_____________)

Now, comment out the code that prints the table. Create a larger set of items (20 to 30) and a higher max-weight in your main function. Be sure that your recursive and dynamic programming solutions return the same values for your larger set of items.

15. Do a timing experiment (use unix `time` and record user time) where you just run the recursive version.

a. What is the time? _______varies________________

Now, time just the run of the dynamic programming solution.

b. What is the time? _______varies, but should be shorter________________

16. Now, we improved the runtime with dynamic programming. How much space (memory) is used for N items and W as the max-weight in order to achieve this improved runtime? _____(N+1)*(W+1) ints____
Ok if they said N*W ints, since that is the space complexity

You are done. Submit your code and answers to this write-up in an electronic file to moodle or on paper to Tammy (one per pair if you worked with partner).

CS324: Greedy Algorithms (part 1): Activity Selection

March 16, 2020 (moved to March 18, 2020 due to University class suspension)

To do before next class:

- HW5 due Wed, Mar 18
- This lab due Fri, Mar 20 if not finished on Mar 13
- Read CLRS chapter 23 before Monday

Today's learning outcomes:

- Understand greedy algorithm properties
- Understand the difference between a greedy algorithm that solves a problem optimally and a greedy heuristic that gives a non-optimal solution
- Apply the greedy strategy to the activity selection problem
- Apply the greedy strategy to the fractional knapsack problem

---

Activity Selection Problem

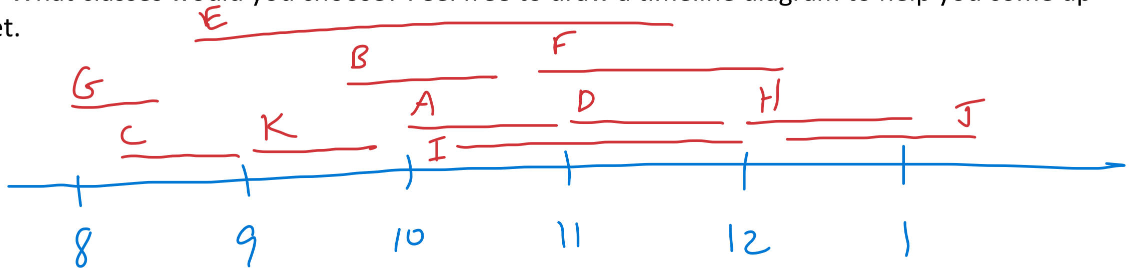
Try it:

Let's say you are trying to schedule the most classes in a single classroom (this reminds me of my days as Associate Dean…). The following contains a list of classes with start and end times. A class can start and the same time as an end time of a previous class (in other words – this is not as realistic for actual humans to enter/exit the room, but it could work on a single processor, for example).

Which classes would you choose to **maximize** the total number that can be assigned to the classroom?

Class	Start	End
A	10:00am	11:00am
B	9:45am	10:30am
C	8:15am	9:00am
D	11:00am	11:55am
E	8:45am	11:45am
F	10:45am	12:15pm
G	8:00am	8:30am
H	12:00pm	1:00pm
I	10:15am	12:00pm
J	12:15pm	1:30pm
K	9:00am	9:45am

Practice: What classes would you choose? Feel free to draw a timeline diagram to help you come up with a set.



In general, what algorithm/strategy could you use to (hopefully) maximize the number of classes that can be put in the classroom?

- shortest duration
- end first
- start first

Let's try this:

```
1.    sort the activities by earliest start time in list L.
2.    F = finish(L[1]); S = {L[1]}
3.    for i = 2 to N:  //N = L.length
4.          if start(L[i]) >= F
5.                S = S U {L[i]}
6.                F = finish(L[i])
```

So, we add the activity to the set if its start time is after the most recent activity selected.

Apply this to our problem input:

	Class	Start	End
6	A	10:00am	11:00am
5	B	9:45am	10:30am
2	C	8:15am	9:00am
9	D	11:00am	11:55am
3	E	8:45am	11:45am
8	F	10:45am	12:15pm
1	G	8:00am	8:30am
10	H	12:00pm	1:00pm
7	I	10:15am	12:00pm
11	J	12:15pm	1:30pm
4	K	9:00am	9:45am

F = ~~8:30~~ ~~11:45~~ 1:00

What's the sorted order by start time?

G, ~~C~~, E, ~~K~~, ~~B~~, ~~A~~, ~~I~~, ~~F~~, ~~D~~, H, ~~J~~

S (set of activities chosen) = G, E, H

Now, let's make a one change and order the activities by earliest **finish** time instead.

G, ~~C~~, K, B, ~~A~~, ~~E~~, D, ~~I~~, ~~F~~, H, ~~J~~

F = ~~8:30~~ ~~9:45~~ ~~10:30~~ ~~11:55~~ 1:00

S = G, K, B, D, H

Discuss: Does this second strategy (sorted by earliest finish time) create an optimal solution? Why or why not?

It does

How would we prove that this is optimal?

greedy choice

1a. Usually by contradiction, to show that the "greedy choice" must be in an optimal solution.
Assume the greedy choice is not in the final set and show that it can be substituted for another choice in an optimal solution

1b. Sometimes by induction to show that the "greedy choice" is optimal for a problem of size N given solution from problem of size K < N

optimal substructure

2. Show that the problem exhibits optimal substructure
Optimal solutions to sub-problems can be combined to create optimal solution for the problem (same as property for dynamic programming)
See textbook page 416 for the proof of optimal substructure for activity selection

We also need to state clearly the theorem we are proving. This can be tricky to formulate – a bit like loop invariants. Here is the theorem for the greedy choice property:

Theorem: Let S_k be a non-empty ~~sub~~-problem with k activities $\{a_1, a_2, a_3, a_4, \ldots a_k\}$ and assume activity a_m has the minimal (earliest) finish time of the activities in S_k. Then, a_m is included in the maximum-sized subset A of schedule-able activities in S_k.

Proof: Suppose we have S_k, a set of k activities and an optimal solution of schedule-able subset of activities called A. Let a_m be the activity in S_k that has the earliest finish time (if there is a tie, choose a_m to be the one with the latest start time).

Case 1: a_m is in A. We are done, since a_m is in the set of maximal-sized activities A.

Case 2: a_m is not in A. Let a_j be the activity with the earliest finish time in A. Since a_m is not equal to a_j, we know the finish time for a_m is less than the finish time for a_j. Thus, we can substitute a_m for a_j in A, since there are no activities in A that would start before a_j's finish time, meaning there are no activities that would start before A_m's finish time. Let $A' = A - \{a_j\} \cup \{a_m\}$. The size of A' is equal to the size of A. Therefore, a_m is in a maximum-sized subset of schedule-able activities.

■

How would we prove a greedy strategy is not optimal?
This is usually easier. Just find a counter-example.
Find one example that is produced by the greedy strategy and show there is another solution that is better.

We actually already did this with the example above. We can use the set of class activities above to show that the ordering by ***earliest*** start time is not optimal.

3 activities
5 activities

```
Activity_Selection_G(L):
1.    sort the activities by earliest finish time in list L.
2.    F = finish(L[1]); S = {L[1]}
3.    for i = 2 to N:  // N = L.length
4.          if start(L[i]) >= F
5.                S = S U {L[i]}
6.                F = finish(L[i])
```

What is the runtime of this activity selection algorithm by each step?

Line 1. $\Theta(N)$ // radix sort

Line 2. $\Theta(1)$

Line 3. $\Theta(N)$

Total: Θ(______N______)

Some variations of activity selection:

- More than one resource (classroom, processor, channel, conference room)
- Maximize total activity time (usage) versus number of activities
- Activities have weights and we want to maximize the total weight of activities (perhaps customers pay a premium for use of the resource)

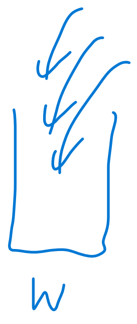
CS324: Greedy Algorithms (part 2): Fractional Knapsack

March 16, 2020 (moved to March 18, 2020 due to University class suspension)

Fractional Knapsack Problem

You programmed the 0-1 knapsack problem (recursively and using dynamic programming).

Now, let's look at this problem.

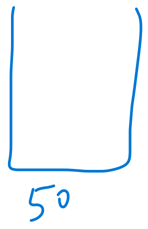


Input: You have a knapsack that can hold up to some maximum weight. Items can be taken in a fractional way, so now think of items as bulk rice, bulk flour, and bulk candy. Each of the items has a weight and value, and fractions of the items can be taken up to the entire amount available.

Output: Choose fractional items to maximize the value of the knapsack.

Let's try it:

Item 1: weight 10, value 60
Item 2: weight 20, value 100
Item 3: weight 30, value 120



Max-weight: 50

1. Which items (including fractional part) would you select?

100% Item 2
100% Item 3 $\longrightarrow$ 220 ☆ optimal for 0-1

100% Item 1
100% Item 2 $\longrightarrow$ 60 + 100 + 80 = 240
67% Item 3

Think about it: How long would a brute-force solution take to solve this problem?

$2^N \times N$ $\quad\quad$ $\Theta(2^N)$

Think about it: What greedy strategy could you use to choose your knapsack items?

density = value / weight

```
        FRACTIONAL_KNAPSACK_G(W, L);  // W is max-weight; L is a list of items
θ(n)        for i from 1 to L.length:
                L[i].density = L[i].value / L[i].weight        * calculate densities
θ(nlgn)     sort L by decreasing density
θ(1)        Total = W;  S = nil
θ(n)        for i from 1 to L.length:       * && Total > 0
                if L[i].weight <= Total:   // take all of the item
                    S = S U {100% of L[i]}
                    Total = Total - L[i].weight
                else if L[i].weight > Total:     // take as much as you can of this item
                    S = S U {100*(Total/L[i].weight)% of L[i]}
                    Total = 0
            return S
```

Think about it: What is the runtime of this algorithm? $\Theta(n \lg n)$

That was much better than the brute-force try-all-combinations strategy. Whew. But, does that give us an optimal solution? Let's look at the greedy choice of the highest-density item.

greedy choice

Theorem: The highest density item is in the optimal solution.

Proof: Suppose S is an optimal solution to the fractional knapsack of a set K of items that could be put in the knapsack. Let k_d be the highest-density item in set K. Suppose k_d is not in S. Find the highest-density object in S and call that s_d. Consider set S' = S – {s_d} U {k_d}, where we replace as much weight of s_d with k_d as we can. If there are 10 pounds of s_d and only 5 pounds of k_d available, then replace 5 pounds of s_d with 5 pounds of k_d. Since k_d has the highest density, k_d has larger density than s_d. Therefore, S' has a larger value than S. S cannot be the optimal solution. Therefore, by contradiction, k_d must be in the optimal solution at its maximum amount.

We also need to show that fractional knapsack exhibits optimal substructure, but hopefully you can see that if you have solutions for fractional knapsack for two smaller knapsacks, you can combine these together to get a solution for the single knapsack.

Some last notes on greedy algorithms:

- Need to determine if problem has optimal substructure and exhibits the greedy choice property
  - Remember – greedy choice is that the greedy decision must be in an optimal solution
- Usually the strategy is to figure out the greedy choice, sort the data by best choice, and then decide to add item to set given other constraints
- Warning – some greedy choice elements may change priority as earlier decisions are made. For example, if you are dealing with a graph and your order is most-neighbors, your list order may change as you remove nodes and process them, because it may affect the number of neighbors of other nodes. Not to fear – heaps to the rescue to use as a priority queue.
- Backtracking algorithms can be thought of as greedy – make a choice and keep going until you reach a point where that choice proves to not be good, then back up and try another path. Real greedy algorithms do not backtrack. You commit to the choice at each stage.
- **Greedy Algorithm** = greedy strategy that produces an **optimal** solution.
- **Greedy Heuristic** = greedy strategy that produces a solution (but it may not be best). Sometimes, our problems are so difficult in terms of runtime that we need to use a greedy heuristic. We go with fast over perfect.

CS324: Graph Algorithms (Lecture Part A)

March 20, 2020

Announcements:

- HW6 due Mon, Mar 23; part B is due Sunday, Mar 22 at noon if you want it graded before Mar 25
- Next exam Wed, Mar 25
- Read CLRS chapter 24 before Monday

Today's learning outcomes:

- Review graph representations (should have seen this in CS 305)
- Review breadth-first-search, depth-first-search, and topological sort (should have seen this in CS 305)
- Review minimum spanning trees, Prim, and Kruskal (should have seen this in CS 305)
- Prove that MSTs have the greedy choice property
- Analyze the runtime of graph algorithms

Graphs

Directions: You will work through these questions to quickly refresh the graph algorithms you learned in CS 305. Since we are now on-line, you are welcome to work through these problems with other people in either section of the course in any way that you can – through phone calls, chat, Microsoft Teams. Some of the problems will ask that you supply the answers to Moodle as the "check-in" for today's lecture. Those questions will indicate "Submit answer to Moodle".

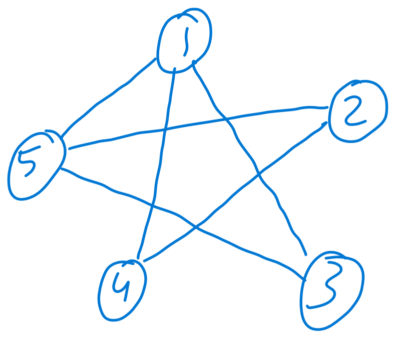
Recall that a graph in computer science contains vertices (typically denoted by set V) and edges (typically denoted by set E).

1. What is the difference between a directed graph and an undirected graph?

directed has arrows, undirected has links

2. Suppose this is the graph in written as vertices and edges. Draw the graph these sets represent below.

V = {1, 2, 3, 4, 5}
E = {{1, 3}, {3, 5}, {2, 4}, {1, 4}, {1, 5}, {2, 5}}



3. Is your graph in question 2 directed or undirected? **ANSWER IN MOODLE**

a. directed
[✓ b. undirected]

4. Complete the matrix representation of the graph in question 2 below (see page 590 of textbook):

	1	2	3	4	5
1	0	0	1	1	1
2	0	0	0	1	1
3	1	0	0	0	1
4	1	1	0	0	0
5	1	1	1	0	0

5. Complete the adjacency-list representation for the graph in 2 below (see page 590 of textbook):

$$\begin{tikzcd}
1 \quad \boxed{} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4} \arrow[r] & \boxed{5 \mid /} \\
2 \quad \boxed{} \arrow[r] & \boxed{4} \arrow[r] & \boxed{5 \mid /} & \\
3 \quad \boxed{} \arrow[r] & \boxed{1} \arrow[r] & \boxed{5 \mid /} & \\
4 \quad \boxed{} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2 \mid /} & \\
5 \quad \boxed{} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3 \mid /}
\end{tikzcd}$$

6. Graphs are an important tool for modeling problems in computer science. They are particularly useful right now in our current situation of trying to stay socially distant from other people. We could use a graph to model spread of disease. The last paper I read about the coronavirus said that scientists believe the "infection rate" is just over 2. That means one person infects, on average, two other people. If we model relationships as a graph (people as nodes and people-in-contact as edges), we can start modeling and simulating the spread of the virus. I write this not to scare you more about coronavirus. I write this to inspire you as computer scientists. The tools you are learning in this course will prepare you to solve the world's most difficult problems. What is another application/problem where using a graph is an important tool?

ANSWER IN MOODLE

answers vary

Graph Searches

7. You should already know how breadth-first search works. I think of BFS as a slow slime that oozes from the start vertex to find the other vertices. Review the BFS algorithm on page 595 of the textbook. We use a queue to store nodes. The first node we put in the queue is the start node (denoted s in the

textbook). All nodes start out as white. Follow the example in figure 22.3 of the textbook. What is the order of the nodes that are explored in BFS?

ANSWER IN MOODLE

s w r t x v u y

8. You should already know how depth-first search works. I think of DFS as navigating a maze – exploring down one long path until we reach a dead-end, then back-tracking to the previous node, exploring again, then back-tracking, etc. See the algorithm on page 604 of the textbook. DFS uses this notion of a stamp-time. Each node has a **discovery time** when it is first explored and a **finish time** when it is done being explored. White nodes have not yet been explored. Grey nodes are in process of being explored. Black nodes are finished being explored. Look at figure 22.4 in the textbook. The search starts at node u with a start time of 1. We also keep track of our "parent" node in the discovery. This is denoted as the field pi for the vertices. The figure shows back edges, cross edges, and forward edges as well. During the search, why is there a cross-edge from node w to node y?

y is black when w is visited

9. What is the runtime of breadth-first search, given V is the number of vertices and E is the number edges? Think about how many vertices are put into the queue and total lengths of the adjacency lists.

ANSWER IN MOODLE

O(V + E)

10. What is the runtime of depth-first search, given V and E? Think about how many times DFS-VISIT is called on each vertex and how much work is done per call based on the number of edges.

ANSWER IN MOODLE

Θ(V + E)

11. If we have a DAG (directed acyclic graph), we can do a topological sort. A topological sort is an ordering of the vertices such that if (a, b) is an edge in E, then a must come before b in the ordering (but they do not need to be adjacent in the ordering). Describe a problem in which doing a topological sort would be useful:

- Scheduling tasks for a project
- prerequisites for courses

answers can vary

12. It is easy to extend depth-first search to solve the topological sort problem. Here's what you do:

```
TOPOLOGICAL_SORT(G):
1      L = nil // empty linked list
2      DFS(G) to compute finish times for each vertex
3      As each vertex v is finished, insert v into L
4      Return L // this is the topological sort order
```

What is the runtime of topological sort? Θ(V + E)

13. Let's practice finding a topological sort of a DAG. Here is a DAG below. Assume depth-first search processes nodes in alphabetical or numerical order when there is a tie. Thus, DFS will start with node m in the graph below.

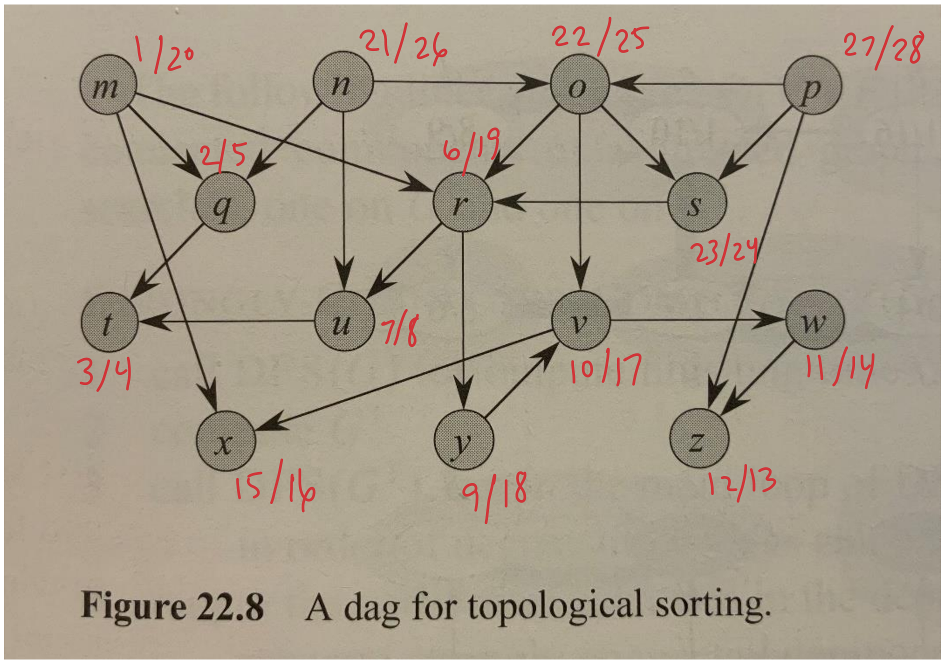


Figure 22.8 A dag for topological sorting.

p n o s m r y v x w z u q t

Run depth-first-search on the vertices, starting with m. As each vertex is finished, insert it into the linked list. What is the topological sort order of the vertices?

ANSWER IN MOODLE

____

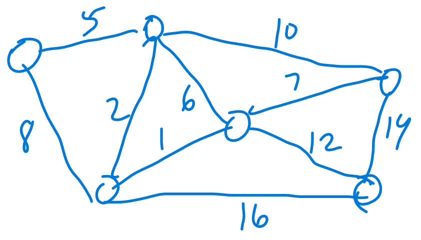
The next set of notes will review minimum spanning trees, Prim's algorithm, and Kruskal's algorithm. This material is part of today's lecture as well.

CS324: Minimum Spanning Trees (Lecture Part B)

March 20, 2020

See the textbook or view the videos from CS 305 to review Prim's Algorithm and Kruskal's Algorithm for finding the minimum spanning tree of an undirected graph G.

Why is finding a minimum spanning tree useful?



Since this is an analysis of algorithms course, we are going to analyze the runtime of Prim's as well as show that the greedy strategy produces an optimal solution.

1. Is Prim's Algorithm greedy? [✓ YES] NO

2. Is Kruskal's Algorithm greedy? [✓ YES] NO

Analysis of Prim

Recall that Prim's Algorithm grows the tree from a particular root vertex and uses a min-priority queue (aka, a min-heap) to store the vertices.

Here's the algorithm from the textbook.

initializing

```
MST-PRIM(G, w, r)    // r is the root from where we are growing the tree
1       for each u in G.V
2               u.key = INF
3               u.parent = NIL
4       r.key = 0
5       Q = G.V                 // put all vertices in min-priority Q (heap)
6       while Q != NIL
7               u = EXTRACT_MIN(Q)
8               for each v in G.Adj[u]
9                       if v in Q and w(u, v) < v.key
10                              v.parent = u
11                              v.key = w(u, v)       ← elevating priority
```

Let's look at the lines in groups:

a. What is the runtime of lines 1 through 3? O(V)

b. What is the runtime of line 4? O(1)

c. What is the runtime of line 5 (how long to build-heap?)? O(V)

Now, let's analyze the while-loop more holistically.

VlgV {
d. How many vertices will be extracted from Q? V

e. How long does it take for each extract-min operation? lgV
}

ElgV {
f. How many overall edges do we examine? E

g. How long does it take to set v's key? (this involves a decrease-key operation)? lgV
}

The total runtime is O(VlgV + ElgV). O(VlgV) for the extract-mins on V vertices and O(Elgv) for the decrease-keys for each edge. Since the graph will have more edges than vertices (otherwise, finding the spanning tree is not interesting), the total runtime is O(ElgV).

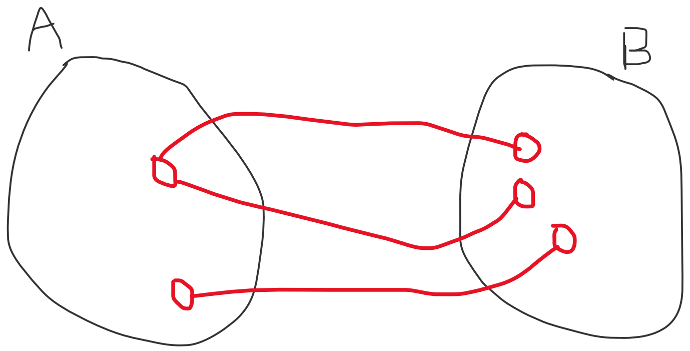
Note: there is a faster heap (Fibonacci heap) that can decrease this to O(E + VlgV).

You can read about the runtime of Kruskal's algorithm in the textbook on page 633. Note that the runtime depends on the data structure used to handle the disjoint sets to determine if an edge creates a cycle. Since we did not focus on the disjoint-set data structure earlier this semester, I will leave this for you to read as a "for your information".

Greedy Choice

Now, we need to also know that the MST problem has the greedy choice property.

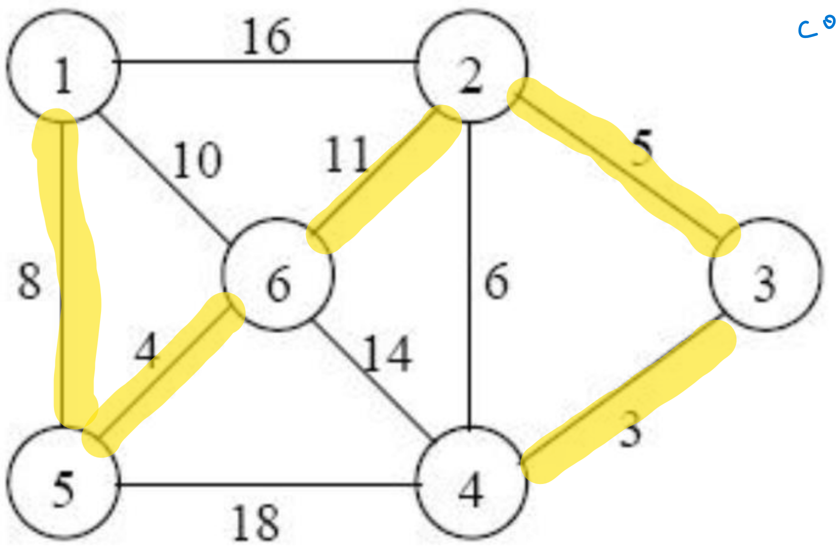
Theorem: If A and B are a partition of the nodes in graph G, then a minimum-weight edge that connects node in A with a node in B must be in the MST for G.



Thus, in the figure above, the edge that spans the sets A and B with the minimum cost must be in the MST.

Proof by contradiction: Assume e is the minimal-weight edge that connects any node in A with any node in B, where A and B are disjoint sets of vertices of G. Let's assume that a minimum spanning tree T of G does not contain edge e. T must contain one of the edges from set A to B. Let's call it d. Let T' be T with d removed and e added. T' must have a total cost that is smaller than T. This is a contradiction since we assumed T was a minimum spanning tree. Therefore, the MST must contain edge e.

Practice: Use Kruskal's Algorithm to determine the minimum spanning tree of the following graph:



Sort edges

costs:

3 ✓
4 ✓
5 ✓
6 × cycle
8 ✓
10 × cycle
11 ✓
14 × cycle
16 ×
18 ×

SUBMIT TO MOODLE

Submit to moodle the list of edges that are part of the MST.

If you have, run Prim's Algorithm on the same graph to determine the MST.

CS324: Graph Path Search

March 23, 2020

Announcements:

- HW6 due before class today
- Exam on Wednesday
- Read CLRS chapter 34 before Friday

Today's learning outcomes:

- Review the shortest path search problem
- Review and apply Dijkstra's Algorithm to single-source shortest path search to other vertices
- Apply Bellman-Ford Algorithm to single-source shortest path search to other vertices
- Compare the two approaches

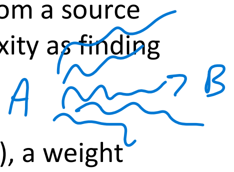
Suppose you are creating a navigation system and need to find the best route from UP to Dallas, Oregon. How could you model this as a graph problem? What are the vertices? What are the edges?

intersections roads

What metric would you use for "best" route? shortest mileage, fastest speed (time to destination)
consistent speed

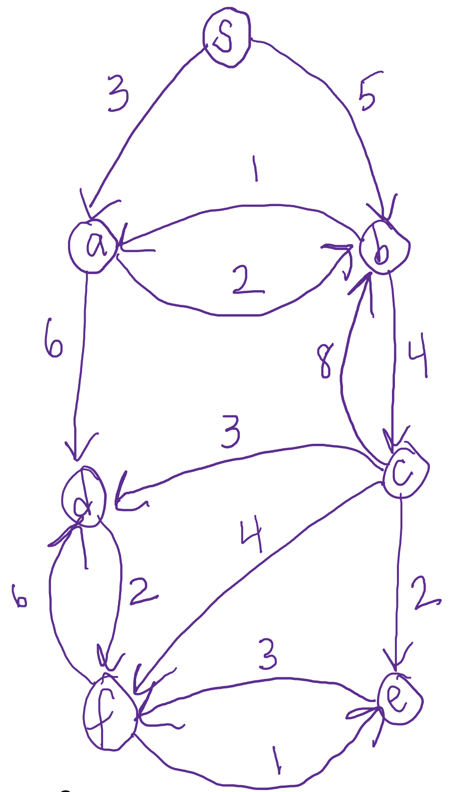
What values would you put on the edges in this graph problem? miles

This problem of finding the shortest path from one vertex to another arises all the time in problems that can be modeled as graphs. We will look at two algorithms for finding the shortest path from a source node to all other vertices in the graph. It turn out that this problem has the same complexity as finding the shortest path from a source node to a particular destination node.



Single-Source Shortest Path Problem definition: Given a graph G (directed or undirected), a weight function that maps each edge to a real number, and a source node s, what are the shortest paths and costs to other vertices in G?

First, let's look at a graph and see if we can figure it ourselves.



What is the shortest path from s to e?

11 $s \to b \to c \to e$ $s \to a \to b \to c \to e$

What is the shortest path from a to f?

8 $a \to d \to f$

Now, suppose we update the graph to have negative weights instead of just positive weights. Are negative weights ok? Try updating the weight from c to e to be -2 instead of 2. Can you still solve the problem?

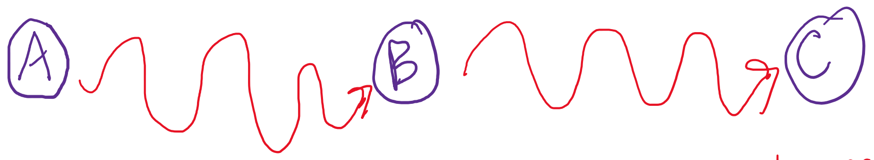
$s \to e$: 7
$a \to f$: 7

Now, suppose we update the weight of (e,f) to be -3. Can you still solve the problem?

OK, so we will need to handle cycles with negative total cost.

Let's take a step back and look at the problem again. How many total paths could there be from a vertex S to a vertex D in a graph? __________ $2^{|V|}$

Optimal substructure to the rescue! It turns out that the shortest path from A to C that passes through B must have a the shortest path from A to B and from B to C.



combine both paths to get best path

So, if we can solve sub problems, we can combine them for a solution to the original graph. We have actually already seen a restricted graph version of this problem — the Manhattan problem, where the graph consists of edges leaving vertices going east and south.

Now, let's take a step back and talk about weights and the total cost of the path. The weights on edges are real numbers. What do we mean by "shortest" path in this problem? The classic case is that the cost is the sum of the weights of the edges in the path. However, there may be other cost metrics, such as:

- Minimize product of edge costs (assuming weights are at least 1)
- Minimize maximum edge cost
- Minimize minimum edge cost

We just need the total cost operation to be associative and for weights {x, y, z}, x <= y implies x op z <= y op z.

∅

So, a metric like minimize the average edge cost will not work for these algorithms. (Choose route from Portland to Dallas, OR such that the variance in traffic speed is minimal.)

We will stick to the total cost being the <u>sum of weights for</u> the edges on the path.

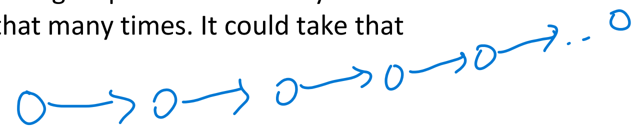
Relaxation, Generic, and Bellman-Ford

These algorithms use a process called relaxation. All this means is if we find a better way, we update the cost and the predecessor node along the path.

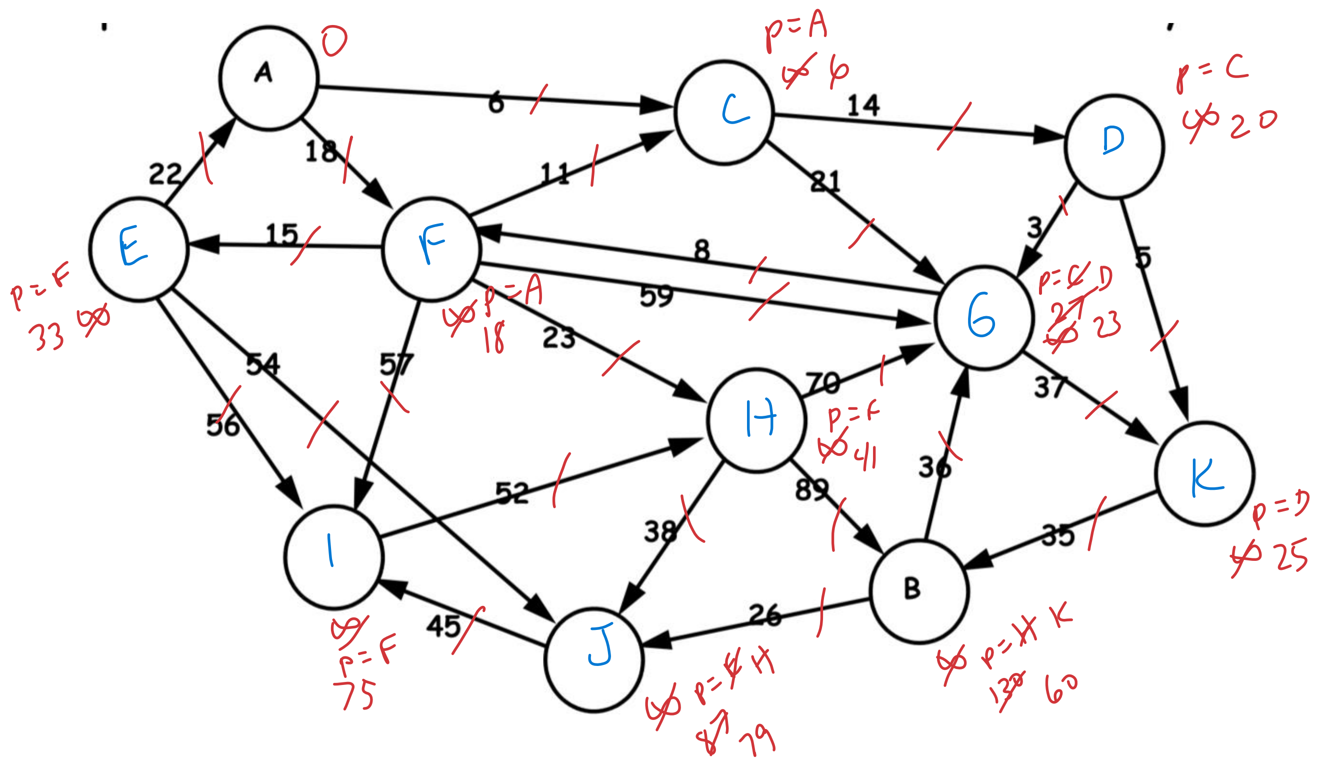
```
RELAX(u, v, w):              //u and v are vertices, w is the weight function
      if v.d > u.d  + w(u, v)
            v.d = u.d + w(u, v)          // found a better way
            v.p = u                      // update parent/predecessor along path
```

These algorithms also initialize the vertices such that v.d is infinity for all vertices and v.p is NIL for all vertices. Then, s.v is set to 0 where s is the source vertex. This is called INITIALIZE_SINGLE_SOURCE(G, s).

Our first algorithm, which we will call GENERIC, relaxes each edge |V|-1 times. So, by the end of the process, each vertex has the total cost of the best path stored. The longest path from s to any node with |V| vertices has length |V|-1, so that's why we do the relaxation that many times. It could take that long to set the total cost.



```
GENERIC(G, w, s):
1       INITIALIZE_SINGLE_SOURCE(G, s)
2       for i = 1 to |G.V|-1
3               for each edge (u, v) in G.E:
4                       RELAX(u, v, w)
```



Let's try it on the graph above and calculate the shortest-cost paths from A to the other nodes.

Put the v.d value inside each circle.

Run the relaxation for all edges 10 times. could stop early if there are no changes

How do we reconstruct the best path from A to B? We figure it out in reverse order, starting with B and accessing its p value, going to the p value node, accessing its p value, etc. until we get to node A.

60

A C D K B

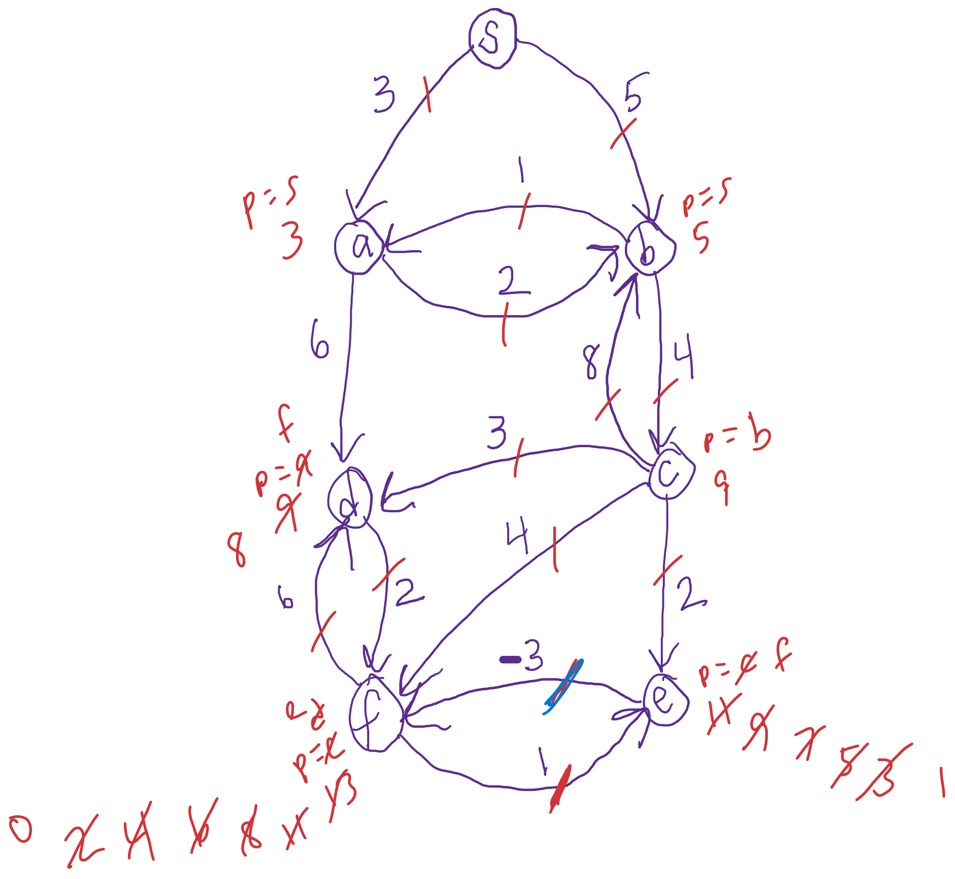
OK, so we have one algorithm for getting the best cost path to the other nodes. How long does GENERIC take? ___O(VE)___

O(V)
+ O(VE) = O(VE)

Well, this worked on graphs that have non-negative weights. The Bellman-Ford extends the generic algorithm in that it does GENERIC. Then, it checks for negative-weight cycles. If it finds a negative weight cycle, it returns FALSE.

```
BELLMAN_FORD(G, w, s):
        GENERIC(G, w, s)
        for each edge (u, v) in G.E
                if v.d > u.d + w(u, v)
                return FALSE
        return TRUE
```

Let's try running it on this graph from s to e with a negative-weight cycle:



f.d = 0

e.d = 2 + -3

= -2

1. How long does the Bellman-Ford algorithm take to run, given V vertices and E edges? O(VE) O(VE) + O(E)

UPLOAD ANSWER TO MOODLE

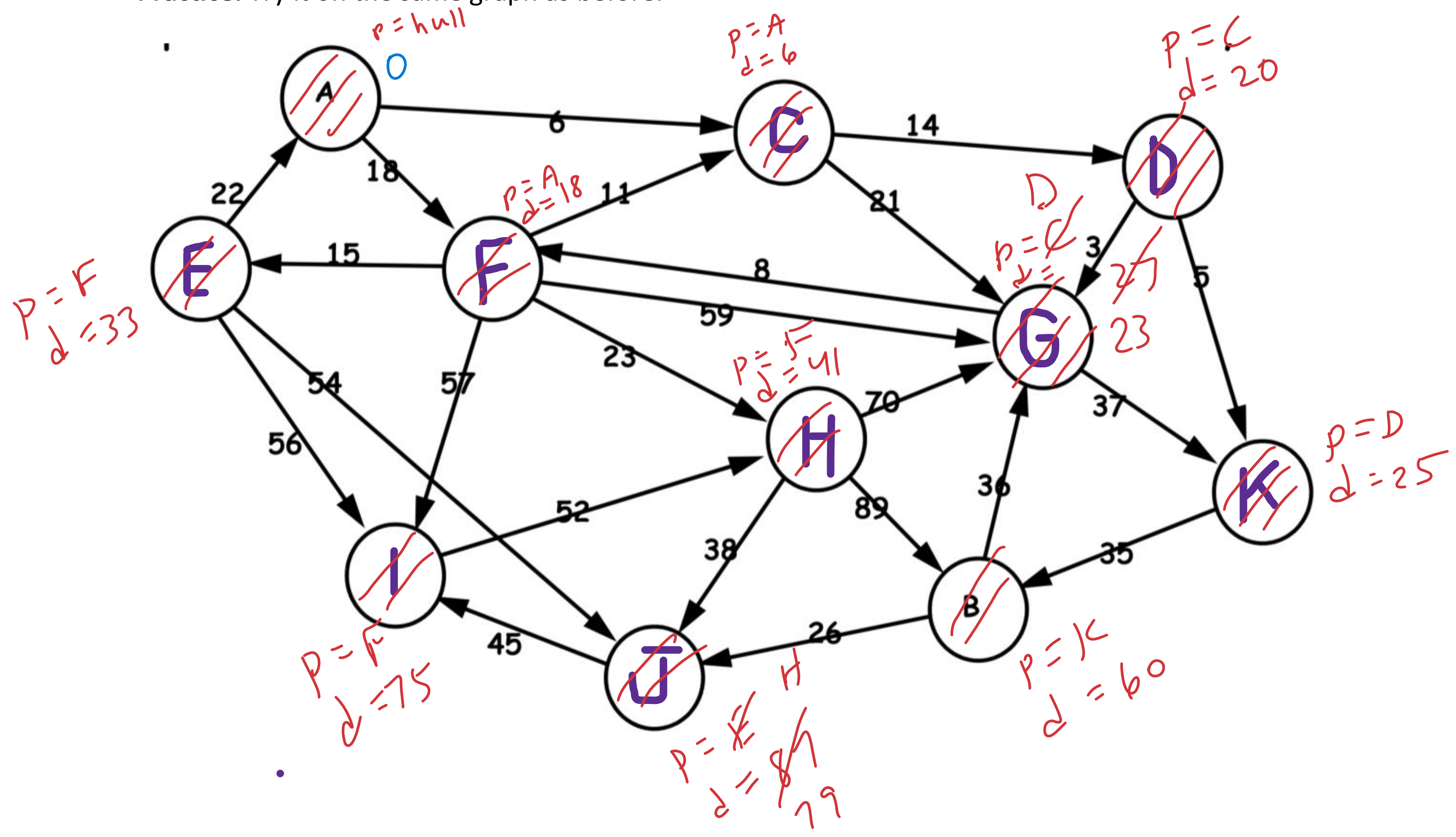
See the textbook section 24.2 for how to turn this idea of relaxation into finding critical paths in directed, acyclic graphs. If the relaxation happens in topological sort order, then we find the critical path in O(V+E) time.

Dijkstra

This algorithm is likely familiar from data structures. It uses relaxation, just like Bellman-Ford. But, it does the vertex processing in a certain order to lower the number of relaxations we need to do. It uses a priority queue (binary min-heap is what we saw earlier in the course) to extract the next node to process.

```
DIJKSTRA(G, w, s)
    INITIALIZE_SINGLE_SOURCE(G, s)
    S = NIL
    Q = G.V                 // put all vertices in heap
    while Q is not NIL
        u = EXTRACT_MIN(Q)   // get smallest u.d from min-heap
        S = S U {u}
        for each vertex v in G.Adj[u]
            RELAX(u, v, w)       // may decrease key
```

Practice: Try it on the same graph as before.



2. What order are the vertices removed from the queue (min-heap)? A, C, F, D, G, K, E, H, B, I, J

UPLOAD ANSWER TO MOODLE

Now, how long does DIJKSTRA take to run?

- What is the cost of EXTRACT_MIN? O(lgV) // to move right-most leaf to root and heapify
- What is the cost to put all vertices in the heap? O(V) // all but source have cost of infinity, so // just put them in the array with source // at position 0 and others following
- How many total edges across all loops do we need to process? O(E) // for all queue extractions
- How many total decrease-key calls do we have? O(ElgV)

Total: O($\underline{\text{VlgV + ElgV}}$)

Since the number of edges is greater than the number of vertices, this is O($\underline{\text{ElgV}}$).

O(V)
O(V)
O(VlgV)
O(ElgV)

Comparing the two algorithms:

3. Which algorithm can report negative-weight cycles?
   a. Generic
   [✓ b.] Bellman-Ford
   c. Dijkstra

   UPLOAD ANSWER TO MOODLE

4. Which algorithm is fastest for non-negative-weight cycle graphs?
   a. Generic
   b. Bellman-Ford
   [✓ c.] Dijkstra

   UPLOAD ANSWER TO MOODLE

I encourage you to practice running the algorithms on graphs, so you are familiar with their execution.

CS324 Part A: Welfare Check

March 27, 2020

Announcements:

- HW7 assigned (you can finish most of it now) and due in one week
- Keep reading CLRS chapter 34

Today's learning outcomes:

- Check-in about how things are going
- Understand complexity class P (tractable) and provide examples in P
- Understand complexity class NP (intractable) and provide examples in NP
- Prove a problem is in NP

Note: During the Teams lectures on Friday, I am happy to answer any questions you have about the exam you took on Wednesday.

Welfare Check

Most of you that I see on Teams during class time seem to be doing ok. Given your response rate for homework assignments and the in-class activities, you seem to be doing ok. I know you are stressed. I know this is not what you expected when the semester started. We are all learning. With the transition and Moodle issues during the past week, I am not going to charge anyone any late days used on HW5, the knapsack lab, or HW 6. You still get two more late days to use (although there are just two more assignments remaining this semester). I want to check-in with you. Please answer the questions on Moodle.

Tammy's answers

1. On a scale of 1 to 5 (1 = not well, 3 = ok, 5 = awesome), rate your current ~~sense of~~ overall sense of being right now.

2. I have access to the Internet that allows me to do my schoolwork. ([✓ yes] or no)

3. If you answered no to #2, please describe where you are and what you need in terms of infrastructure.

4. I am keeping in touch with friends and classmates who are helping me with this course. (yes or no) N/A

5. My main life concern right now is: balance w/ work and family – teaching online is much more work (can be anything)

6. What is helping me most in this course is: getting to hear from students on Teams

7. What I need more help with in this course is: hearing from more students

This is a stressful and uncertain time right now. I want you think about someone who is important in your life that you are not physically self-distancing with right now. Please call them or send them a text to check in with them and tell them you care about them.

8. I have contacted someone important in my life to check-in with them. ([✓ yes] or no)

CS324 Part B: Class P and Class NP

March 27, 2020

What is fast, anyway?

We are now moving into the part of the course where we classify problems as tractable or intractable. Problems that are **tractable** can be solved in polynomial time with regard to their input size N. **Intractable** problems are those that are super-polynomial.

2

$N \lg N$

What is polynomial-time? Anything that can be solved in O(N^k) where k >= 0.

N^2

N^3

N^{1000}

Problems that have polynomial-time algorithms are in the **complexity class P**.

N^{5000}

Class P

1. What problems have we seen so far are in the complexity class P?

Sorting

Graph searches

M·M.

LCS $\rightarrow$ poly

2. What running times would not be poly-time?

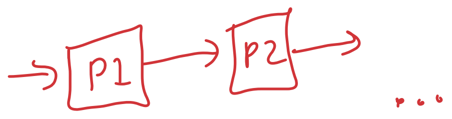
2^N $N!$ N^N

We do have to be somewhat careful about the model we use for computing. We have to be able to solve problems in polynomial time on a deterministic computer (an ordinary machine).

3. How do we show a problem is in complexity class P?

What we have done for much of the semester – show there exists an algorithm that solves the problem in O(N^k) time.

4. What if we have two problems (P1 and P2) in class P. Suppose the answer to P1 is fed in as input to P2:



Solution = P2(P1(input)).

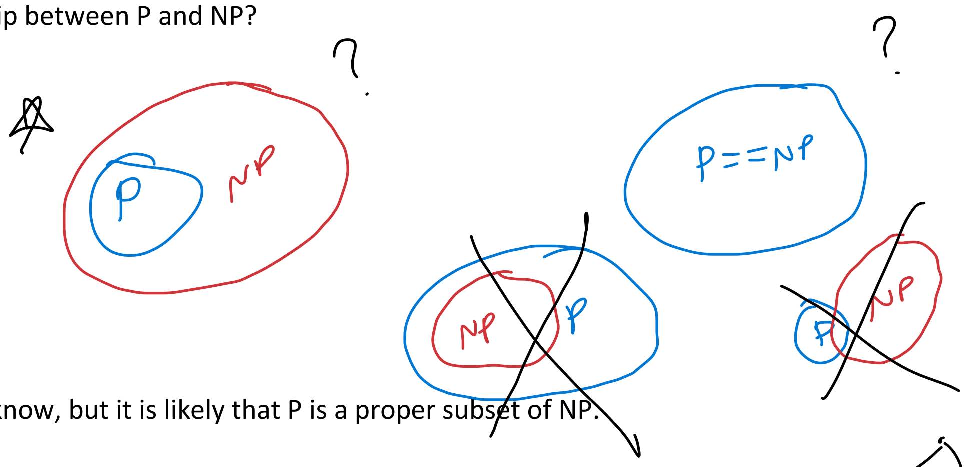
Is this composition of problems in P? ____yes____

Class NP Nondeterministic Polynomial Time

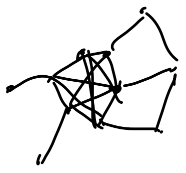
Some problems need a nondeterministic computational model to be solved in polynomial time. The set of problems that can be solved in polynomial time on a nondeterministic computer are in complexity class NP.

Another way to view this is if we have the right guess (nondeterminism makes the guess), we can **verify** the guess in polynomial time. guess → and – check

4. What is the relationship between P and NP?



We don't really know, but it is likely that P is a proper subset of NP.



5. What kinds of problems are in NP?
- ✓ Clique – does a graph have a clique of size-k?
- ✓ Graph coloring – can we color a graph with k colors?
- Theorem proving – can an conjecture be proved given a set of axioms and rules of interference?
- Crossword puzzle construction – can we put a list of words in MxN grid of white/black squares?
- Sudoku – given a sudoku puzzle will filled-in entries, can the rest of the puzzle be solved?
- Hamiltonian path – is there a Hamiltonian path (path goes through every vertex) of a graph?
- Hamiltonian cycle – is there a Hamiltonian cycle (cycle that goes through every vertex) of a graph?

Things to think about regarding NP problems:
- If you are given a solution for an instance of a problem, it is straightforward and poly-time to check the constraints.
- It may not be clear how to come up with an efficient solution in the first place.
- Try all possibilities is a likely way to come up with a solution.

Example of proving a problem is in NP.

yes/no

Formal Decision Problem: Does a graph G have a Hamiltonian cycle?

Language Formulation: HAM = {<G>| G is a Hamiltonian graph}

This is a set of graphs that are Hamiltonian

Proof:
Assume G is a graph. Assume c, the certificate solution, is an ordering of vertices c = (v_1, v_2, … v_n). We can verify that c is a solution as follows:

N-1 +1

1. Check that all vertices in c are in G.
2. Check that all vertices in G are in c.
3. For each pair of vertices in c, v_i and v_{i+1}, check that there is an edge with these vertices as endpoints.
4. Check that (v_n, v_1) is an edge in G.

We also need to show that this verification process takes polynomial time. For this verification, we will assume the graph is represented as an adjacency matrix.

1. $O(N^2)$

2. $O(N^2)$

3. $\rceil O(N)$

4.

Since HAM can be verified in polynomial time, it is in complexity class NP.

6. What is the runtime? ______$O(N^2)$______
SUBMIT TO MOODLE

Some interesting things to think about:

Tractable (in P)	In NP (likely intractable)
Shortest Path (Dijkstra)	Longest Simple Path
Eulerian Tour (cycle traversing all edges of a graph)	Hamiltonian Cycle (cycle traversing all vertices of a graph)
Boolean formula with 2 variables per clause is satisfiable	Boolean formula with ~~2~~ 3 variables per clause is satisfiable

$(x \vee y) \wedge (!x \vee y) \wedge (x \vee z) \ldots$ $(x \vee y \vee !z) \wedge (x \vee y \vee z) \ldots$

Your Turn to Practice: Show that graph coloring is in complexity class NP.

Formal Decision Problem: Can vertices of graph G be colored with K different colors such that no two adjacent (connected) nodes are colored with the same color. *This is used for coloring countries on maps.*

Language Formulation: GRAPH_COLOR = {(G,K) | G is a graph and vertices can be colored using up to K different colors such that no two adjacent vertices have the same color}

Proof (complete on your own):

Hint: think about what the certificate would include and how you would verify the certificate as a solution. Then, be sure to present the runtime of the verification.

Assume G is a graph and K is a positive integer. Assume a certificate c is a list of vertices and the corresponding colors. We can verify that c is a certificate as follows:

1. Ensure all vertices in c are in G.
2. Ensure all vertices in G are in c.
3. For each edge (u,v) in G:
   a. Check that u's color is different than v's

7. What is your certificate to the problem? a list of vertices and the corresponding colors
SUBMIT TO MOODLE

We also need to check this in polynomial time.

1. $O(N^2)$
2. $O(N^2)$
3. $\underbrace{O(N)}_{|E| \text{ edges}} \cdot \left[\overbrace{O(N) + O(N)}^{\text{two lookups in } c, \text{ one per vertex}}\right] = O(N^2)$

Total runtime is $O(N^2)$, so the problem is in NP.

CS324: Problem Transformations and Circuit-SAT

March 30, 2020

Announcements:

- HW7 due Friday (submit to Moodle)
- Keep reading CLRS chapter 34

Today's learning outcomes:

- Understand problem transformations (mapping one problem to another with an algorithm)
- Relating problems through complexity
- Why transformations are useful
- NP-complete definition
- Define the Circuit-SAT problem
- Circuit-SAT is NP-complete (and will be our "base" problem from which we prove other problems are NP-complete)

---

1. Suppose you want to find the median element of a list L (L is unsorted). Suppose there exists an API that you can use that contains the following functions: min, max, sort, and reverse. How could you use one or more of these functions to find the median?

$\boxed{\begin{array}{l}\text{min}\\ \text{max}\\ \text{sort}\\ \text{reverse}\end{array}}$

Sort(L)

Find middle element $= L[L.length/2 + 1]$

if L.length is odd

or $\dfrac{L[L.length/2] + L[L.length/2+1]}{2}$

You do transformations *all the time* when you code! Or perhaps you write all your code from scratch (no built-in libraries, no use of other's code). I assume you use other code and you spend more of your thinking time about "how can I transform my problem in order to use someone else's code". Being able to transform one problem to another is a very important skill in CS. Keep this algorithmic principle in your toolbox as they come in really handy.

This is called a *reduction* or a *transformation*. I will stick to the word transformation in these lectures, but you should know that other computer scientists call these reductions.

OK, let's go back to median-finding. We reduced this problem to sorting. It turns out there is a linear-time algorithm for finding the median that does not use sorting (so that one should be used in practice), but this example gives us some intuition about problem complexity.

2. Can the **sorting problem** be asymptotically faster than the **median-finding problem**? Given your transformation above, why or why not?

YES [✓ NO]

UPLOAD TO MOODLE

Note, we are referring to the asymptotic complexity of problems here (not the complexity of individual algorithms).

So, we know that median-finding is no harder than sorting in terms of complexity.

If we write out the runtimes for the problems, we get:

$$T_M(N) = T_S(N) + \theta(1) + \theta(1)$$

$$T_M(N) \leq T_S(N) + C_0$$

$$P1 \xrightarrow{f(n)} P2$$

In general, if we have a problem P1 and we transform it into problem P2 and the transformation runs in f(n) time, then:

$$T_{P1}(N) \leq f(n) + T_{P2}(N)$$

Relating Problems to One Another

Suppose we have problems A, B, and C. We will denote algorithms with X, Y, W, and Z.

What if we have the following?

$$\begin{tikzcd} A \arrow[r, bend left, "X \quad \theta(N)"] & B \arrow[l, bend left, "Y \quad \theta(N)"] \end{tikzcd}$$

3. What does that tell you about the complexity of problem A and problem B?

[✓ a. They are the same]
b. A is harder than B
c. B is harder than A

UPLOAD TO MOODLE

Now suppose we have the following relationships. All algorithms run in O(N) time.

$$\begin{tikzcd} A \arrow[r, bend left, "x"] & B \arrow[l, bend left, "y"] \arrow[r, bend left, "w"] & C \arrow[l, bend left, "z"] \end{tikzcd}$$

4. What does that tell you about the complexity of problems A, B, and C?

[✓ a. They are the same]
b. A is harder than B and B is harder than C
c. C is harder than B and B is harder than A

OK, so we have limited ourselves to linear transformations in the examples above. What if we now let ourselves have transformations that are polynomial-time?

5. Suppose A maps to B with a poly-time algorithm and B ~~runs~~ has complexity in O(N) time. What can you say about the complexity of A? ← updated since video

[✓ a. A is poly-time]
b. A is O(N) time
c. A is O(1) time

$$\begin{tikzcd} A \arrow[r, bend left, "O(N^k) \ast"] & B \end{tikzcd}$$

$O(N^k + N)$ ← cost of B $O(N) \ast$

↑ cost of transformation

UPLOAD TO MOODLE

6. Suppose we have A maps to B in O(N^5)-time and B maps to C in O(N^25) time. Do we have a polynomial time transformation from A to C?

[✓ YES]   NO

$$\begin{tikzcd} A \arrow[r, "O(N^5)"] \arrow[rr, bend right] & B \arrow[r, "O(N^{25})"] & C \end{tikzcd}$$

The nice thing about polynomials is that we can add, multiply, and compose transformations.

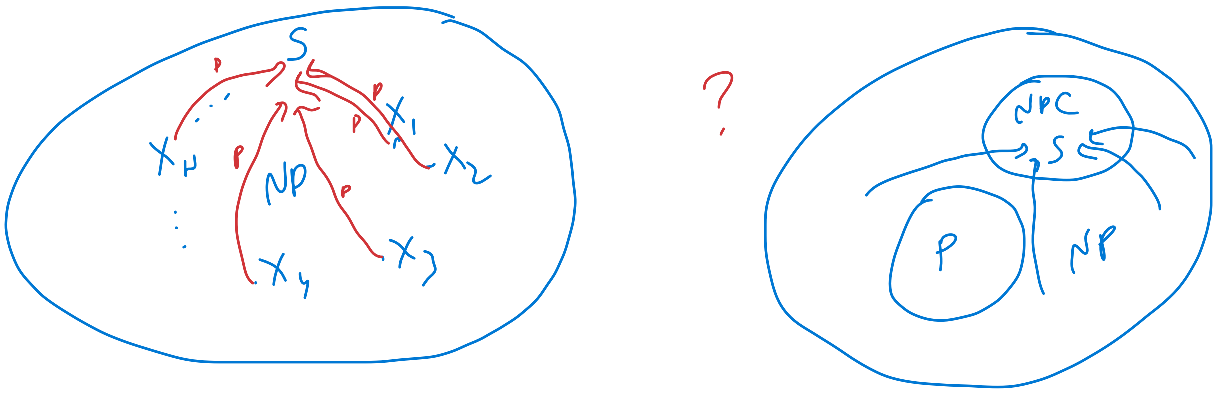
Is the sum of two polynomials a polynomial? yes

Is the product of two polynomials a polynomial? yes

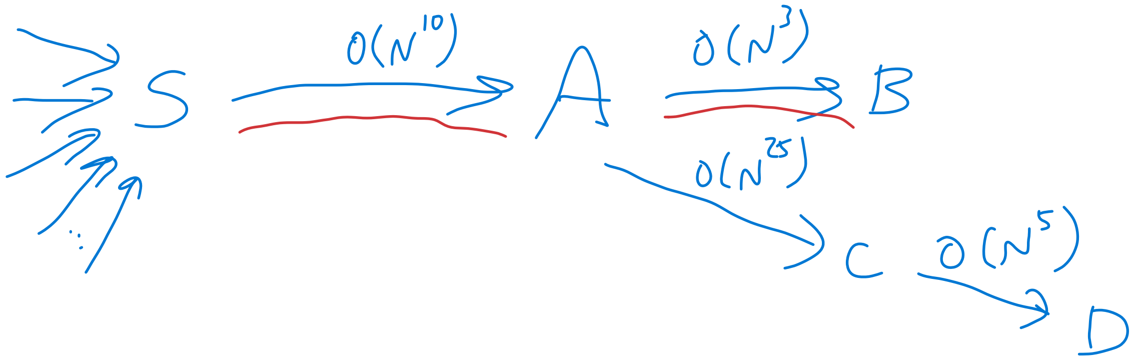
Is the composition of poly-time functions still run in polynomial time? yes

Why is this useful?

- Designing transformations allows you to "pass the buck" on doing work. Perhaps if someone has solved B and you want to solve A, the mapping from A to B is easier than solving A directly.
- Allows us to define a class or problems within NP that are very special. These are called **NP-complete** problems.
- If S is NP-complete, then all other problems in NP can be mapped to S in poly time. Say what? That would be an (infinite?) number of transformations. Yes, that's right.



It does seem wild that we can have all these transformations to a single problem S. Once we have one special problem S, how could we show other problems are NP-complete? (This is actually easier…)



$$\begin{tikzcd}
S \arrow[r, "O(N^{10})"] & A \arrow[r, "O(N^3)"] \arrow[dr, "O(N^{25})"'] & B & \\
 & & C \arrow[r, "O(N^5)"] & D
\end{tikzcd}$$

If we have S → A → B and both mappings are poly-time and S is NP-complete, are A and B NP-complete?
[✓ Yes]

if A and B are in NP

We need to find one of these special S problems. That seems difficult, doesn't it? Well, Cook to the rescue. Cook proved that any problem in NP can be transformed in poly time to the Boolean formula satisfiability problem (SAT).

- We prove SAT is NP-complete in CS 357 very formally, so I am skipping it here since you have already taken CS 357 or may take it in the future. The jist is that we can model any algorithm/program as a series of configurations of the computer state (memory + program counter) and those configurations can be modeled as a Boolean formula.

CIRCUIT-SAT

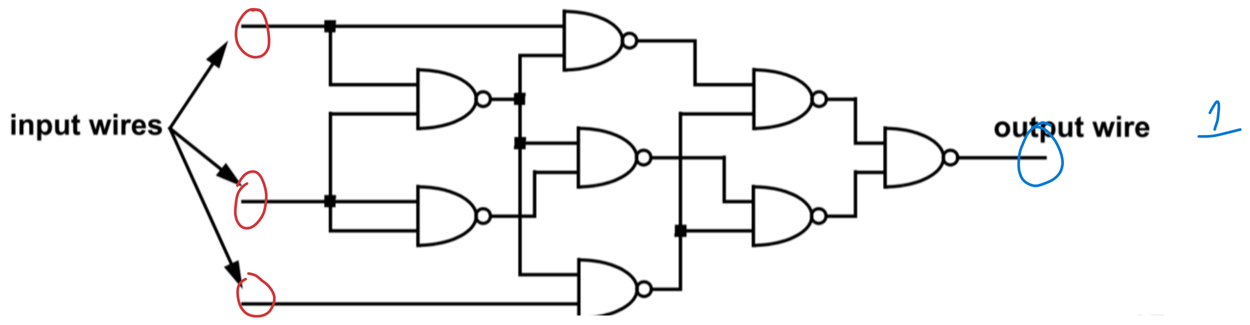
We are going to look at a related problem – CIRCUIT-SAT. The only real difference is that CIRCUIT-SAT uses a circuit-model with input wires and an output wire.

Theorem: Any NP problem can be transformed in polynomial time to the CIRCUIT-SAT problem. NP-hard

Note: the textbook allows circuits with AND, OR, and NOT gates. We are just going to use NAND gates for our circuits, since it is a universal gate (all other gates can be modeled with one or more NAND gates).

Formal Definition of CIRCUIT-SAT with only NANDs: Given 2-input NAND gates and wires, is there a setting of input wires that results in the output wire being a 1?

Picture of a NAND-circuit instance:

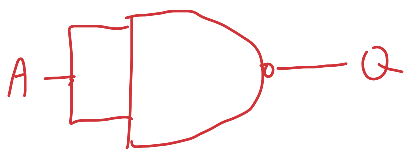


Truth-table for NAND (not and)

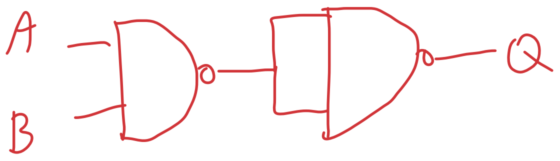
A	B	A NAND B
0	0	1
0	1	1
1	0	1
~~0~~ 1	~~0~~ 1	0

Here are practice problems to show that NAND is the only gate we need in a circuit.

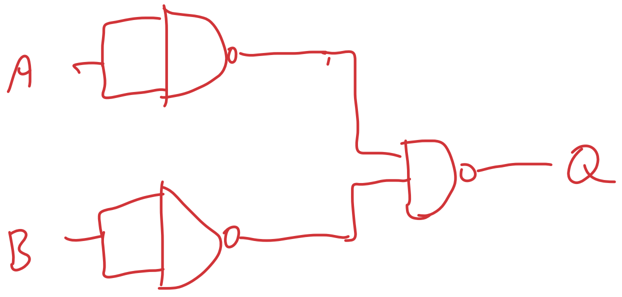
Practice 1: Create a NOT using one NAND gate.



Practice 2: Create AND using two NAND gates.



Practice 3: Create OR using three NAND gates.



Now that we have a good understanding of the CIRCUIT-SAT problem, how could you solve it brute-force?

combinations of input wire settings

How many different input settings do we need to consider? 2^N

7. If we had a certificate of the input settings, could we verify if the circuit's output wire is 1 in polynomial time?

[✓ YES] NO in NP

UPLOAD TO MOODLE

This is the basis of the proof that CIRCUIT-SAT is in complexity class NP (given a certificate and the instance, we can verify it in polynomial time)

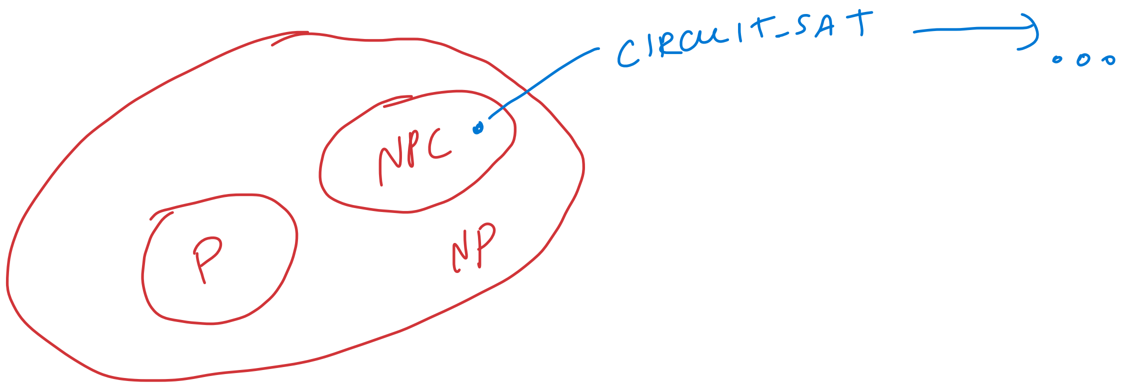
Now, for the harder part to show that CIRCUIT-SAT is NP-complete, we also need to show it is NP-hard (all other NP problems poly time reduce to CIRCUIT-SAT). A more detailed description about modeling a computational process with circuitry is given in the textbook, so this is just a high-level sketch.

1. Any digital device can be implemented with NAND gates and wires (CS 333)

2. A constant amount of computation can be modeled with a constant number of NAND gates.

3. A polynomial amount of computation can be modeled with a polynomial number of NAND gates.

4. Nondeterminism can be injected with "free" input wires from one circuit configuration to the next.



CS324: Circuit-SAT -> SAT; Circuit-SAT -> Graph-Coloring

April 1, 2020

Announcements:

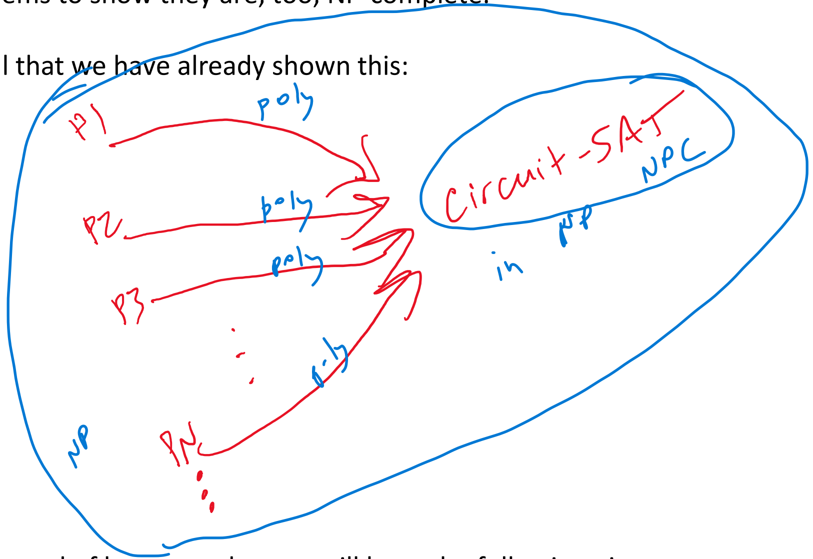
- HW7 due Friday (submit to Moodle)
- Keep reading CLRS chapter 34

Today's learning outcomes:

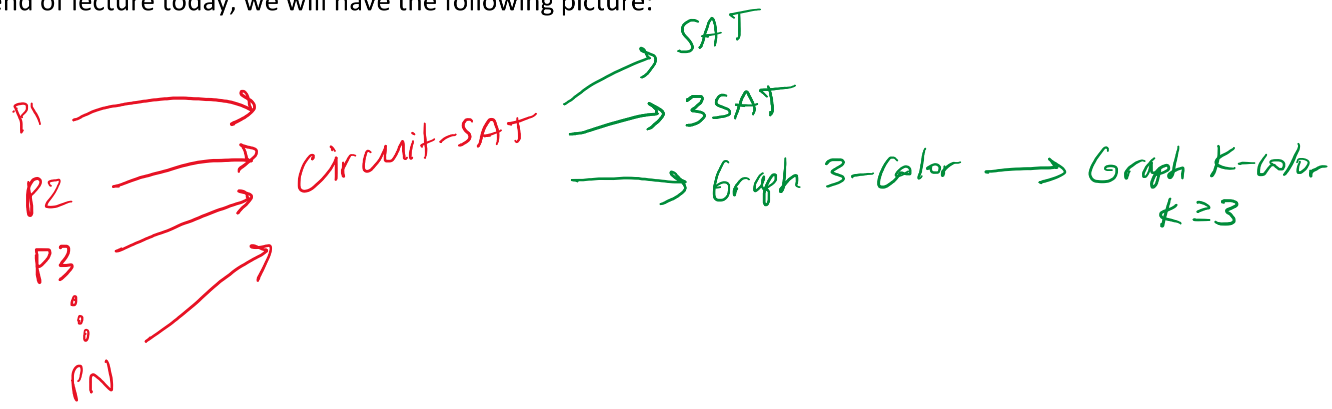
- Review Circuit-SAT instances
- Transform Circuit-SAT instances to SAT instances in poly time
    - By-product: Transformation also works for 3SAT
- Transform Circuit-SAT instances to Graph 3-Color instances in poly time
- Understand that Graph 2-Color is in P
- Graph K-Color where K>=3 is NP-complete

Today we will look at transformations from Circuit-SAT (our first NP-complete problem) to other problems to show they are, too, NP-complete.

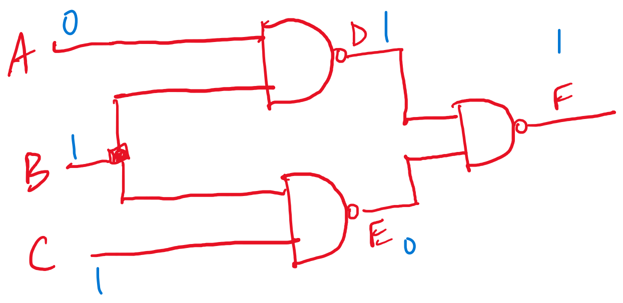
Recall that we have already shown this:



By the end of lecture today, we will have the following picture:



Practice:
Let's recall the Circuit-SAT with NANDs problem. Here is an instance of the problem:



Can you find input-wire settings that make the circuit output wire be 1? What are those settings?

A = 0

B = 1

C = 1

Now, we are going to transform Circuit-SAT into Boolean formula satisfiability (SAT).

$$\underset{\text{NANDs}}{\text{Circuit-SAT}} \longrightarrow \underset{\text{Boolean formula}}{\text{SAT}}$$

In general, to show a problem P1 is NP-complete, we must do two things:

✓ 1. Show P1 is in the complexity class NP by showing a certificate can be verified in polynomial time.

✓ 2. Show there is a poly time transformation from an already proven NP-complete problem to P1.

SAT (Boolean Formula Satisfiability)

Input: Boolean expression of AND, OR, NOT, parens, and Boolean variables.

Output: 1 if there is a satisfying assignment of T/F to variables such that the entire expression is true; 0 if there is no satisfying assignment

Here is an example:

$\swarrow$ CS

((A && B) || (C && (B ~~&&~~ || !C) && !A)) && (!B || !A)

$$\underline{((A \wedge B) \vee \underline{(C \wedge (B \, \cancel{\wedge} \overset{\vee}{} \neg C) \wedge \neg A)})} \wedge \underline{(\neg B \vee \neg A)}$$

$\swarrow$ math

$\wedge$ = and
$\vee$ = or
$\neg$ = not

1. Can you find a setting of A, B, and C such that the entire expression is true?

A: false
B: true
C: true

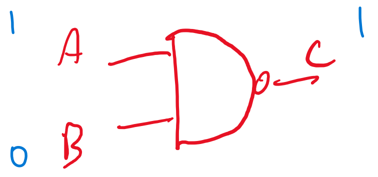
UPLOAD ANSWER TO MOODLE

2. Can you find a Boolean expression involving one variable that is not satisfiable?

$A \wedge \neg A$

UPLOAD ANSWER TO MOODLE

OK, hopefully you understand the SAT problem now. How can we transform Circuit-SAT to SAT? We need to figure out how to convert each NAND gate and wires to a Boolean expression. Well, it is straightforward since C depends on A and B as follows:

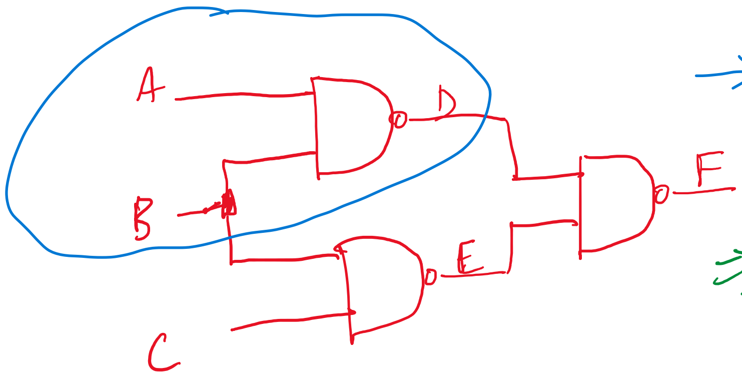


$\Rightarrow$

$$(A \vee C) \wedge (B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

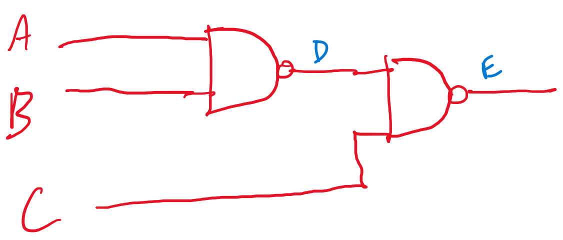
Let’s check that they are equivalent. Try a few settings to the input wires.

Let's try a bigger example of a NAND circuit and transform it into an equivalent Boolean expression.



$\rightarrow$ $((A \vee D) \wedge (B \vee D) \wedge (\neg A \vee \neg B \vee \neg D)) \wedge$
$((B \vee E) \wedge (C \vee E) \wedge (\neg B \vee \neg C \vee \neg E)) \wedge$
$((D \vee F) \wedge (E \vee F) \wedge (\neg D \vee \neg E \vee \neg F)) \wedge$
$\Rightarrow$ F

Practice: Your turn to practice turning a circuit into a Boolean expression.



$((A \vee D) \wedge (B \vee D) \wedge (\neg A \vee \neg B \vee \neg D)) \wedge$
$((D \vee E) \wedge (C \vee E) \wedge (\neg D \vee \neg C \vee \neg E)) \wedge$
$\Rightarrow$ E

Now, is this a polynomial time transformation (reduction) from Circuit-SAT to SAT? Suppose there are W wires and N NAND gates in Circuit-SAT.

How many Boolean variables do we have in the SAT instance? ___W___

How many Clauses do we have in the SAT instance? ___$3 \cdot N$___

We have a polynomial time transformation!! Is SAT in NP? That's easy to show since we can have a certificate of each variable and its truth value. Verifying that this certificate gives us a satisfying assignment takes O(N) time given N is the length of the Boolean formula. Therefore, SAT is NP-complete.

3SAT (Boolean Formula Satisfiability in 3-CNF Form)

3SAT is a special restriction of Boolean formula formats:

- It is in conjunctive normal form (up to 3 variables per clause, each clause only has ORs, clauses are combined via ANDs).

A good

Ok: (A || D || !B) && (B || !D || E) && (!A || !B || !D)

Not ok: (A || D || !B) && ($\underbrace{\text{B || !D || E || C}}_{4}$) Why?

Not ok: (A || D && !B) Why?

Not ok: (A || B || C) || (D || E || F) Why?

We can use the same transformation from Circuit-SAT to 3SAT since the Boolean expression we created has at most 3 variables per clause, each clause has ORs within it, and the clauses are combined via ANDs.

So, now we have the following picture:

$$\begin{tikzcd}
P1 \arrow[dr] & & SAT \\
P2 \arrow[r] & \text{circuit-SAT} \arrow[ur] \arrow[dr] & \\
P3 \arrow[ur] & & 3SAT \\
PN \arrow[uur] & &
\end{tikzcd}$$

NP-complete

✓ Circuit-SAT

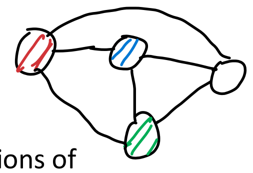
✓ SAT

✓ 3SAT

The nice thing about 3SAT is that its format is not as "wild" as regular SAT and in many cases, it is easier to transform 3SAT to new problems rather than SAT to new problems.

$C_1 \wedge C_2 \wedge C_3 \wedge \cdots C_N$

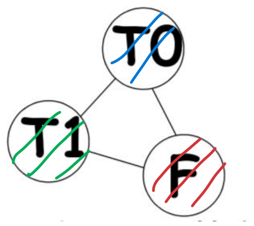
Graph 3-Color (Can we color a graph such that each vertex has a different color than its adjacent vertices? We can use at most 3 distinct colors.)



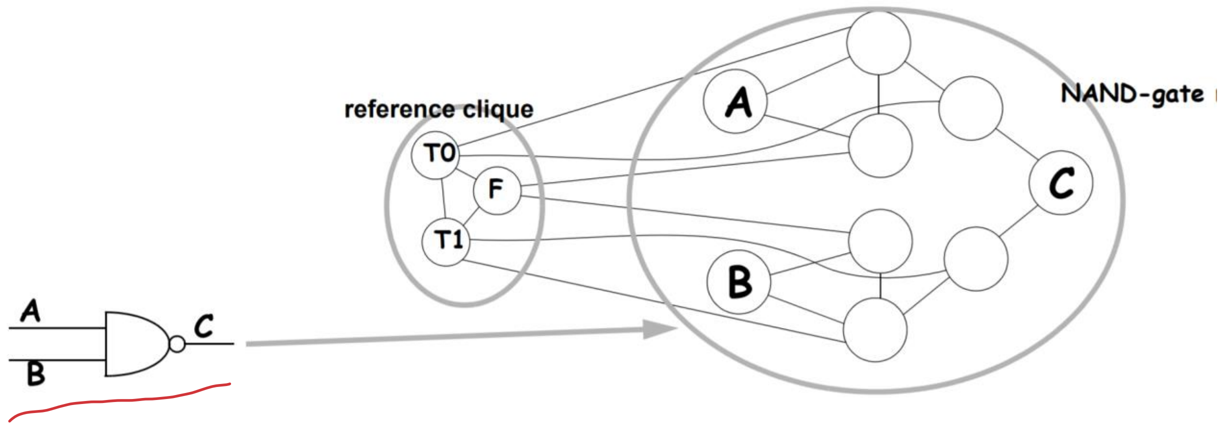
We are now going to transform Circuit-SAT to Graph 3-Color (thanks to Dr. Vegdahl for the transformation). Note that the textbook goes from 3SAT to Graph 3-Color, as to many other versions of this proof. We will go straight from Circuit-SAT.

We need to convert each NAND gate and wires to a graph.

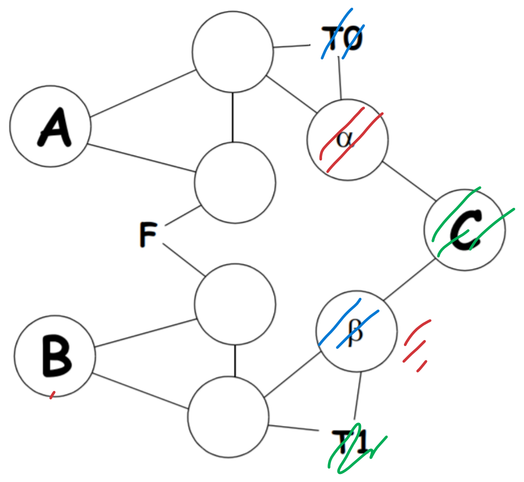
We create a 3-clique that gets put into the graph. The nodes are labeled T0, T1, and F to represent truth values. Because this is a 3-clique, each vertex must have a unique color in the graph.



Now, for each NAND gate, we create the following edges and vertices:



To keep the diagram more simple, we will draw it like this:



Note that T0, T1, and F are still connected via the 3-clique, but we don't draw the lines between them. Otherwise, all the other edges are there.

Does this model a NAND-gate?

T0, T1, and F are distinct colors.

Suppose input A is T0 or T1 (true), what is the node marked alpha forced to be? T1

Suppose input A is F (false), what is the node marked alpha forced to be? F

Suppose input B is T0 or T1 (true), what is the node marked beta forced to be? T0

Suppose input B is F (false), what is the node marked beta forced to be? F

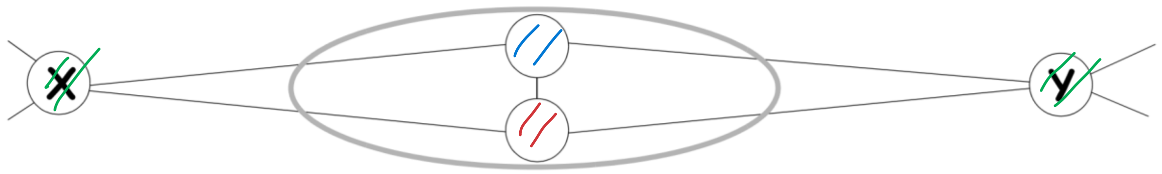
So, if A and B are both colored with T0 or T1, what is the node marked C forced to be? F

UPLOAD TO MOODLE

So, if at least one of A or B is false, one of alpha or beta is forced to be false, so C is? T1 or T0

Thus, this gadget models the NAND gate. If the NAND gate has input wire settings to make the output wire be 1, then the graph is 3-colorable.

OK, so that gives us the basic widget. Now, we need to combine the widgets to model circuits with more than one gate. We will create a connector to each NAND like this:



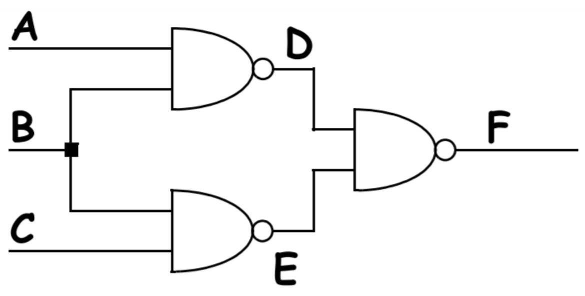
Must X and Y be the same color in a 3-color graph? yes

UPLOAD TO MOODLE

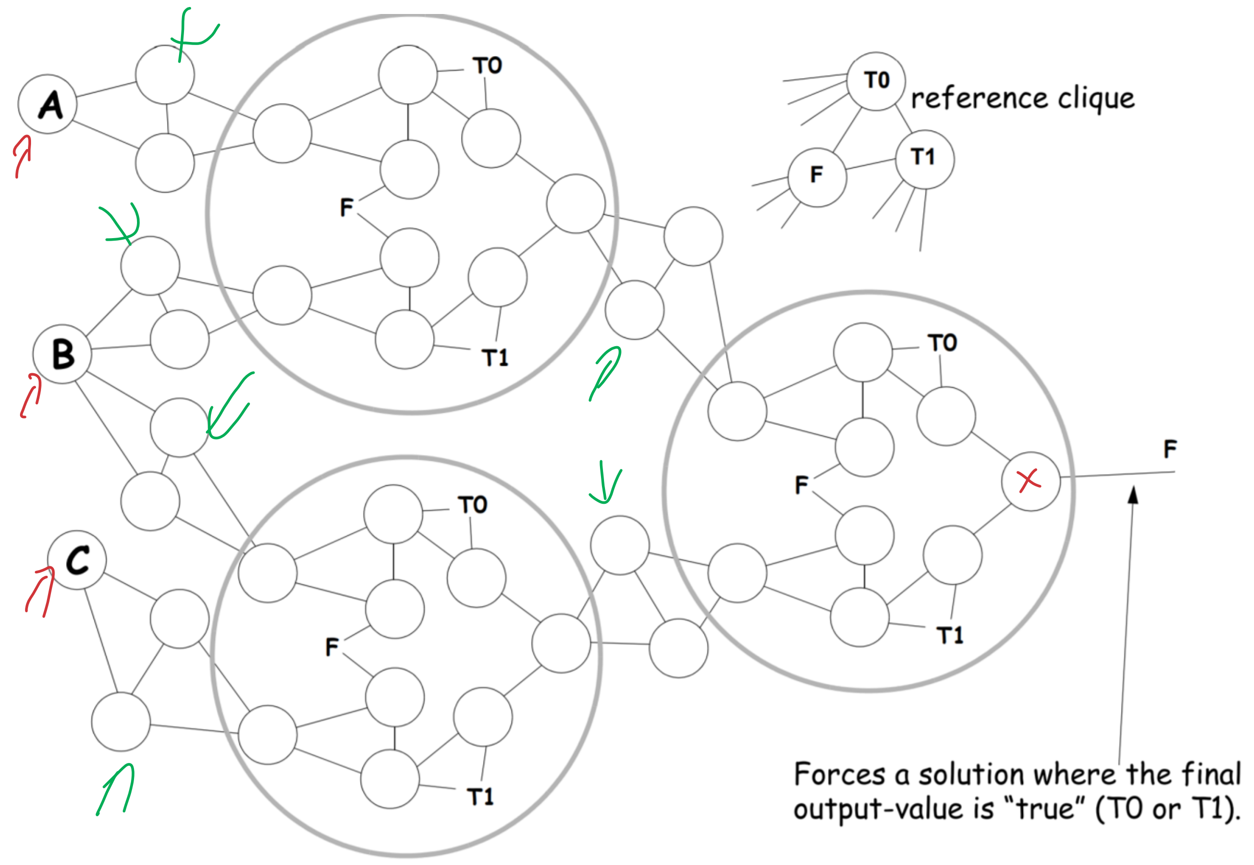
So, this forces any setting of X to be the same setting as Y.

Now, to put this all together:

Here's our input circuit:



Here's our graph:


A
B
C
T0
T1
F
reference clique
Forces a solution where the final output-value is "true" (T0 or T1).

Now, is this transformation done in polynomial time? We have a constant number of vertices and edges that we create for each NAND gate, so all the larger circles take a total of Θ(N) time to make. We have Θ(N) connectors that simulate each wire. We have one vertex per input wire. Thus, the total transformation takes Θ(N) time.

So, that was the "hard" part of the proof – transforming Circuit-SAT to Graph 3-Color in polynomial time. We showed that Graph 3-Color is in NP during a previous lecture. Those two things together show that Graph 3-Color is NP-complete.

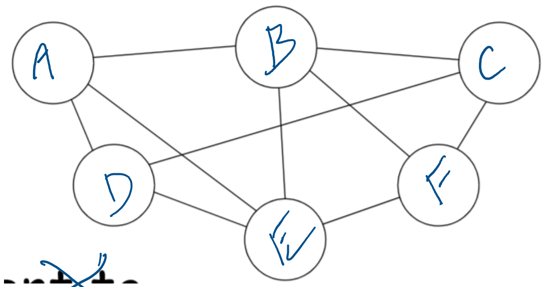
Graph 2-Color is in P

What if we have just two colors to use for a graph? This is actually solvable in polynomial-time by looking for any odd-length cycles. If we have one of those, we would be forced to color two adjacent nodes the same color.

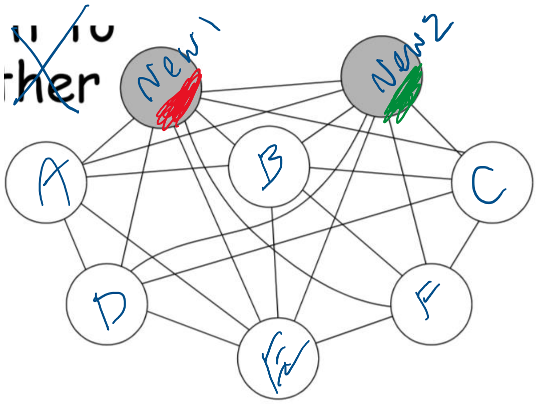
Graph 4-Color, Graph 5-color, Graph K-color is NP-Complete

We can transform Graph 3-Color to Graph 4-Color as follows. We add a constant number of new nodes (for 4-color, we would add 1 more node that is connected to every other node in a graph – forcing the new node to be colored its own distinct color not used in the original graph).

Example: if the input graph for 3-color is this:



Then, the transformation to Graph 5-color would be this:



The new grey nodes would each be colored a new and different color, forcing the original 3-color part of the graph to still be colored in 3 colors.

So, to conclude, we have the following picture now:

$$\begin{tikzcd}
 & & \text{SAT} \\
P1 \arrow[r] & \text{Circuit SAT} \arrow[ur] \arrow[r] \arrow[dr] & \text{3SAT} \\
P2 \arrow[ur] & & \text{Graph 3-color} \arrow[r] & \text{Graph K-color for } K \geq 4 \\
P3 \arrow[uur] \\
\vdots \\
PN \arrow[uuuur]
\end{tikzcd}$$

CS324: Circuit-SAT -> Vertex Cover; Vertex Cover -> Clique; Other NP-complete Problems

April 3, 2020

Announcements:

- HW 8 is our last one (it is due Wed, April 15 or Tues, April 14 at noon if you want it returned before the fourth midterm exam)
- Read CLRS 35.1 and 35.2 before Monday

Today's learning outcomes:

- Review set of NP-complete languages we have seen so far
- Transform Circuit-SAT to Vertex Cover
- Transform Vertex Cover to Clique
- Overview of other NP-complete problems

Let's review where we are:

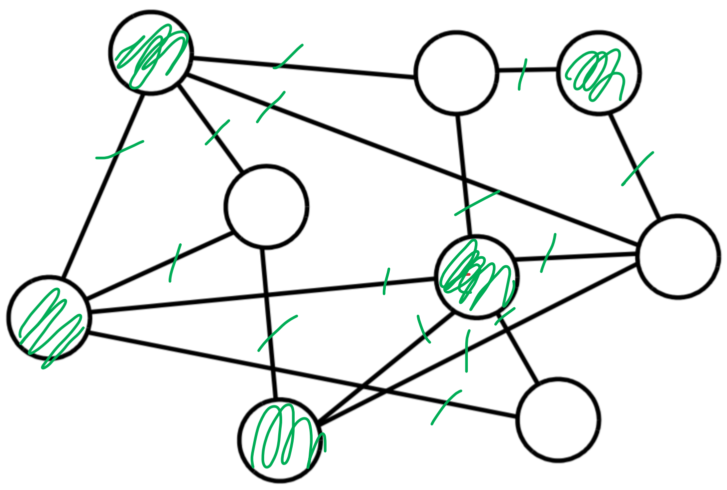
All problems in NP --→ Circuit-SAT

$$\begin{tikzcd} & \text{SAT} & & \\ \text{Circuit-SAT} \arrow[ru] \arrow[r] \arrow[rd] & \text{3SAT} & & \\ & \text{Graph 3-color} \arrow[r] & \text{Graph } K\text{-color} & K \geq 4 \end{tikzcd}$$

Vertex Cover Problem

Given a graph, find a subset of vertices V' such that every edge (u, v) has either u in V' or v in V' or both u and v in V'. Every edge is "covered" by a vertex in the subset.

Let's try an example: Find a cover in the following graph:



OK, so now we understand the problem. Usually, this is posed as an optimization problem – find the smallest such cover. The decision version of the problem is:

Decision version: Can a graph be covered by marking K nodes?

Formal Vertex Cover:

Input: A undirected graph G and a positive integer K

Output: Yes if all of G's edges can be covered by a set of vertices of size K or smaller; No, otherwise

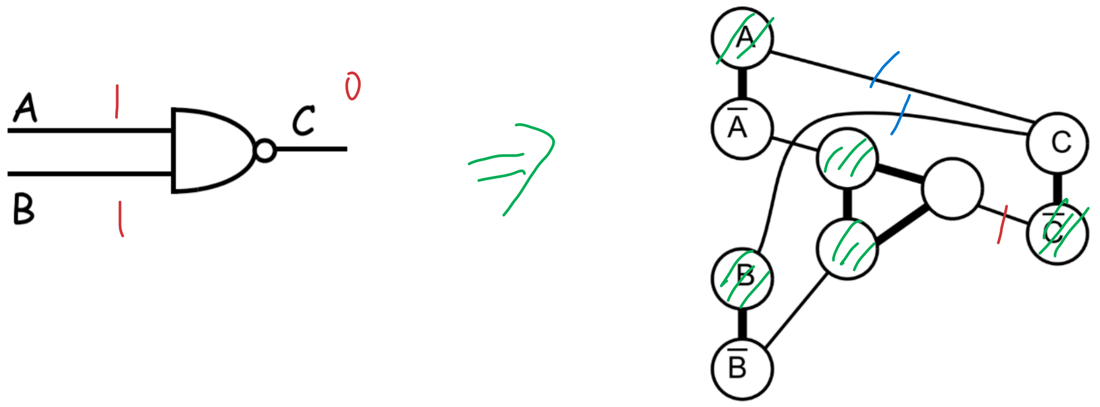
$P \xrightarrow{\text{poly}} VC$

$VC \in NP$

We want to show that Vertex Cover is NP-complete. We could make a poly time transformation from 3SAT, SAT, Circuit-SAT, or Graph 3-Color. In CS 357, we do the transformation from 3SAT. Note that the textbook does the transformation from Clique to Vertex Cover. So, you will see this proof done in three ways:

- Here: Circuit-SAT -> Vertex Cover
- CS 357: 3SAT -> Vertex Cover
- Textbook: 3SAT -> Clique -> Vertex Cover

OK, so our job is to convert NAND gates and wires into a graph (edges, vertices) and a number K. On the surface, this seems difficult. Here's how we convert a NAND into a graph gadget:



Let's see what happened. Each wire gets turned into a barbell (two vertices with one edge, with one vertex being the opposite label as the other). Each negated vertex is connected to one node in the triangle gadget. Then, A is connected to C and B is connected to C.

1. What would K be? _____5________

How many nodes would need to be in the covering of the gadget?

Two for triangle

One for each barbell; total of 3 barbells

ANSWER IN MOODLE

2. Why does this work?

The bold edges in the gadget can be covered by 5 vertices. (choose 2 of the 3 triangle nodes, choose one of A or $\bar{A}$, choose one of B or $\bar{B}$, choose one of C or $\bar{C}$. So, we need to make sure the cover can also cover the non-bolded edges.

If A is 0 and B is 0, does C need to be in the cover? yes

If A is 0 and B is 1, does C need to be in the cover? yes

If A is 1 and B is 0, does C need to be in the cover? yes

ANSWER IN MOODLE

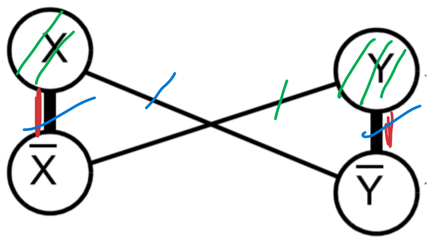
If A is 1 and B is 1, does C complement need to be in the cover? yes

Thus, the graph has a 5-cover if and only if A, B, and C are consistent with the NAND operation.

Now, we need to convert several NAND gates and wires into a single graph G and integer K.

K is the easier part: K is 5N where N is the number of NAND gates in the circuit.

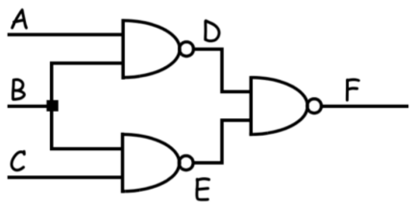
To connect the gates (output of one NAND gate becomes input of another), we use this gadget to connect wire X to Y:



3. What nodes can be put into a cover? $\{X, y\}$ $\{\bar{X}, \bar{y}\}$

Now we can put this all together into one large transformation. Let's transform the following 3-gate circuit into a graph G and integer K.

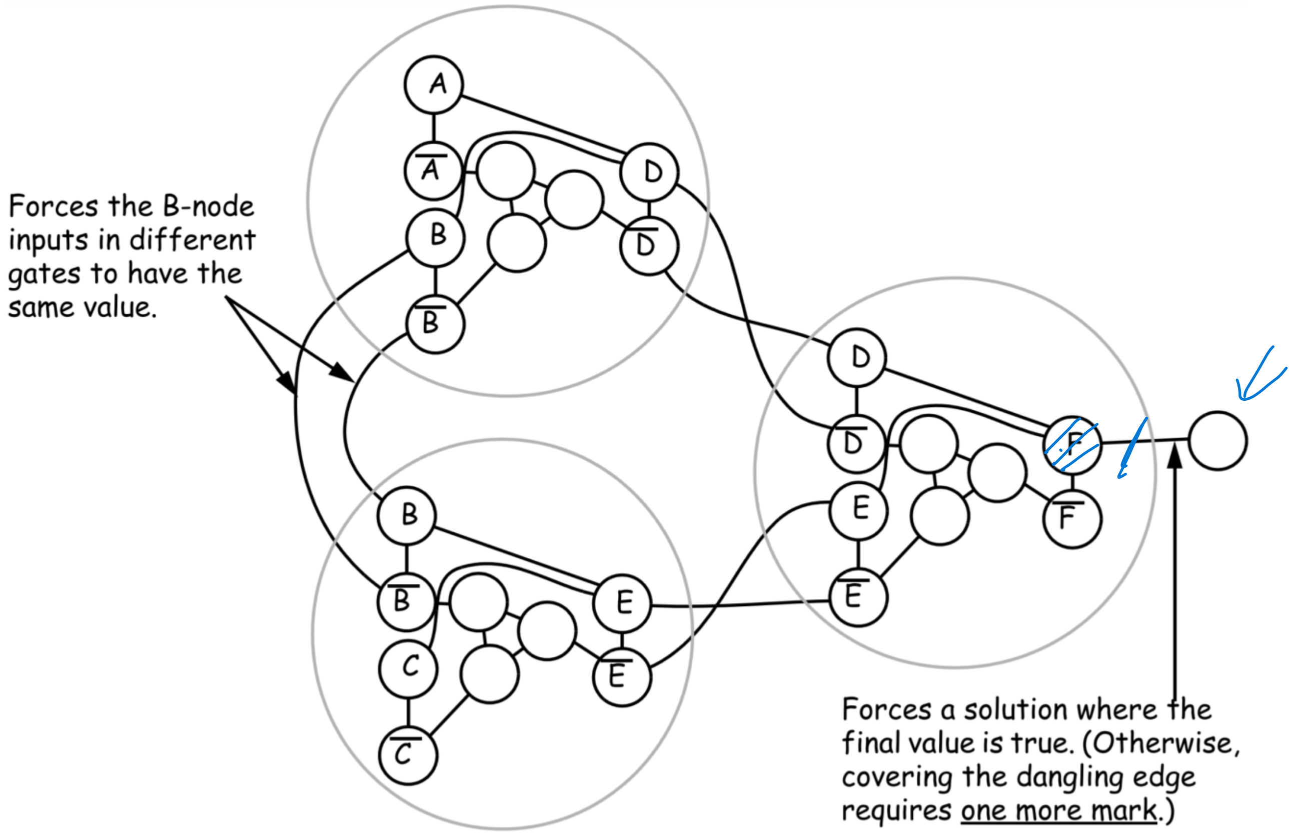
Circuit-SAT Instance:



Vertex Cover Instance:

K = 5*3 = 15

G =


Forces the B-node inputs in different gates to have the same value.
A
$\overline{A}$
B
$\overline{B}$
D
$\overline{D}$
B
$\overline{B}$
C
$\overline{C}$
E
$\overline{E}$
D
$\overline{D}$
E
$\overline{E}$
F
$\overline{F}$
Forces a solution where the final value is true. (Otherwise, covering the dangling edge requires one more mark.)

4. How long does this transformation take (is it poly time)?
   Each NAND creates a constant-sized graph gadget
   Each wire creates constant number of edges
   Final output wire creates a single edge and vertex

   Total runtime: Θ(N)

$\text{Circuit} \rightarrow_p \text{VC}$

In order to complete the NP-completeness proof, we also need to show that Vertex Cover is in NP. This part is easier to show that we can verify a solution in polynomial time:

$\text{VC} \in \text{NP}$

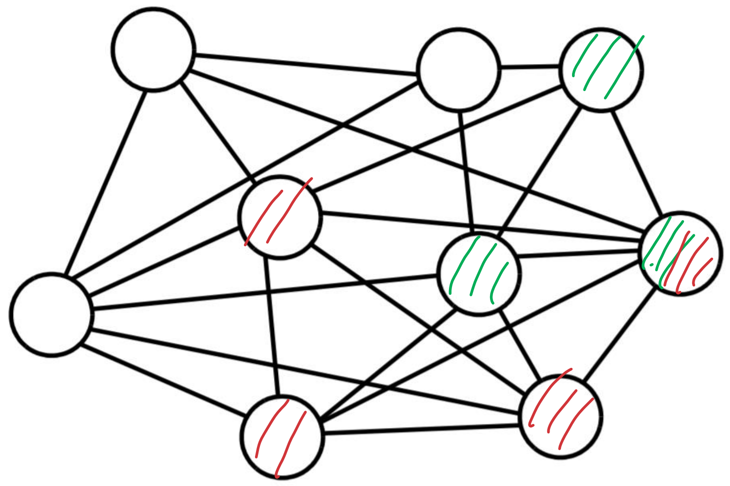
Given a graph G=(V, E), and integer K, and a certificate c of vertices:
1. Ensure that c is a subset of V ✓
2. Ensure that size of c == K ✓
3. Verify that all edges (u, v) in E that u is in c or v is in c ✓

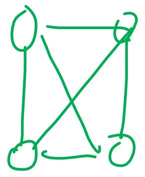
The verification takes Θ(|c|*|V|) + Θ(|c|) + Θ(|E|*|c|) = Θ(N^2) since the size of the certificate is less than or equal to the size of V.

That concludes the proof that Vertex Cover is NP-complete.

Clique

A clique within an undirected graph is a subset of vertices where every vertex in that subset is connected to each other. Can you find a 3-clique in the graph below? Can you find a 4-clique in the graph below?



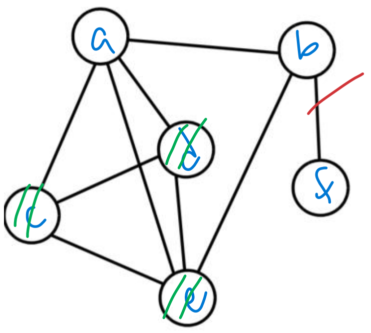


The formal decision-version of the problem is: Given a graph G and a positive integer K, does G contain a K-clique?

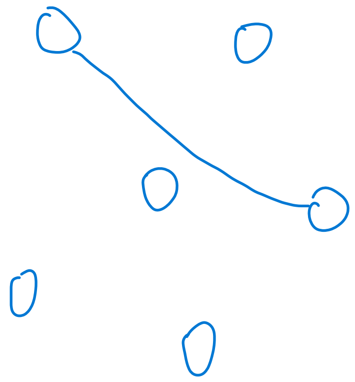
It turns out that Clique and Vertex Cover are just different statements of the same problem. We need to take small detour to understand complements of a graph.

The **complement** of an undirected graph is a graph that has exactly the same set of nodes but wherever there was an edge, there is no edge and wherever there was no edge, there is an edge.

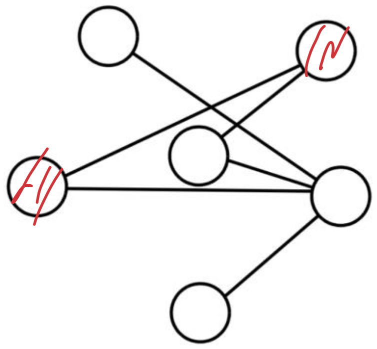
Suppose we have the following graph G:



Can you draw the complement G^C?



Here is the complement G^C:



Think about it: If G has a vertex-cover of size K, what does that tell you about a clique size for the complement of G?

N - K

ANSWER IN MOODLE

Think about it: If the complement of G has a clique of size N-K, what does that tell you about a vertex cover size for G?

K

If there is a set of K nodes that covers all the edges of G, then none of the remaining nodes have connections between them --- this forms the clique in the complement of size N-K.

Look at G above. Can you find a vertex-cover of size 4 in G? Can you find a clique of size 2 in G complement? yes

Can you find a vertex-cover of size 3 in G? Can you find a clique of size 3in G complement? no

So, we have the following transformation from a Vertex-Cover instance to Clique instance:

Given (G, K) as a vertex-cover instance, produce (G^C, N-K) where N is the number of vertices of G.

Does the transformation run in polynomial time? Going through all pairs of vertices of G to produce or remove an edge takes $\Theta(N^2)$ time. Calculating N-K takes $\Theta(N)$ time to count vertices. Thus, the transformation is poly time.

Is Clique in NP? This the easier task.

Given an undirected graph G=(V, E), an integer K, and a certificate c containing the vertices of the clique:

1. Ensure that c is a subset of V ✓
2. Ensure that c has size K ✓
3. Verify that for all pairs of vertices in c, there exists a corresponding edge in G. ✓

Total time: $\underline{\Theta(|c|*|V|)} + \underline{\Theta(|c|)} + \underline{\Theta(|\overset{c}{V}|*|\overset{c}{V}|*|E|)}$, which is a total of $\Theta(N^3)$.

What does our picture look like now?

$$\begin{tikzcd}
 & & & & \text{Graph 3-color w/ degree} \leq 4 \\
 & & \text{SAT} & & \text{Planar Graph 3-color} \\
{} \arrow[rd] & & \text{3SAT} \arrow[r] & \text{SS} & \text{Graph K-color} \\
{} \arrow[r] & \text{Circuit-SAT} \arrow[ruu] \arrow[ru] \arrow[r] \arrow[rd] & \text{Graph 3-color} \arrow[rruuu] \arrow[rruu] \arrow[rru] & & \\
{} \arrow[ru] & & \text{VC} \arrow[r] \arrow[rd] & \text{Clique} \arrow[r] \arrow[rd] & \text{Subgraph Iso} \\
 & & & \text{Ham Circuit} \arrow[d] & \text{PCS} \\
 & & & \text{TSP} &
\end{tikzcd}$$

Other Problems

Subgraph Isomorphism is NP-Complete (transformation from Clique).

Precedence $\overset{\text{PCS}}{\text{Constrained}}$ Scheduling is NP-Complete (transformation from Clique).

Tasks, each taking one time-unit

N processors

Partial order as DAG for task ordering

Can all tasks be scheduled and completed within T time units?

Planar Graph 3-Color is NP-Complete (transformation from Graph 3-Color)

Graph 3-Color with all vertex-degrees <= 4 (transformation from Graph 3-Color)

Hamiltonian Circuit is NP-Complete (see textbook, transformation from Vertex Cover)

Traveling Salesman is NP-Complete (see textbook, transformation from Hamiltonian $\overset{\text{Circuit}}{\text{Cycle}}$)

Subset-Sum is NP-Complete (see textbook, transformation from 3SAT)

There are MANY others and we could spend an entire course on NP-Complete problems.

Practical Advice:

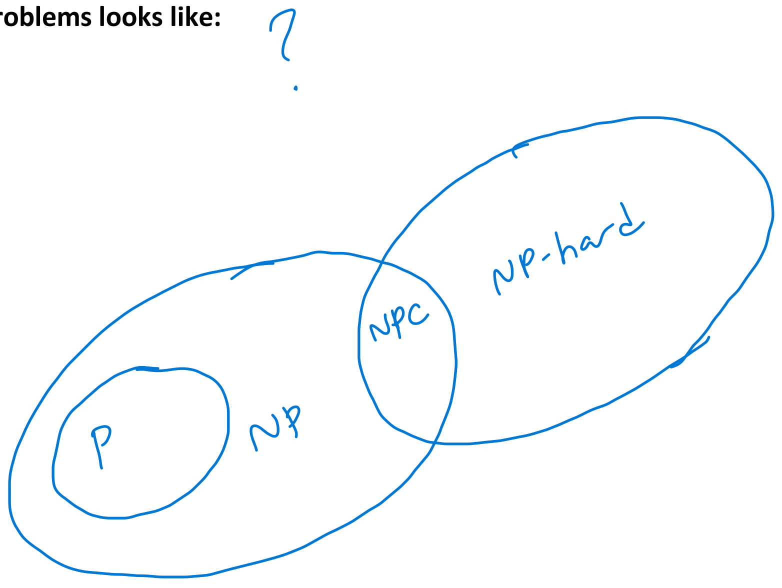
If you run into a problem that looks NP-complete, try to relate it to one of the known NP-complete problems. Your problem may be a small variation of these and can be transformed trivially. Be sure to do the proof in the correct direction (already proved NP-complete problem TO your problem).

If you create a proof for a new problem, have someone check it.

You can also try restricting inputs to your problem and turn it into something tractable. For example, we know that Graph 3-Color is NP-complete, but Graph 2-Color can be solved in polynomial time.

Give up on finding an optimal solution with brute-force and get a "good enough" approximate solution in polynomial time. Approximation algorithms is the topic of the next lecture.

What we think the complexity picture of problems looks like:

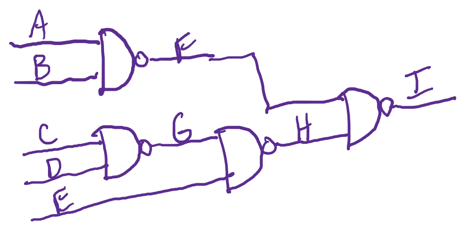


CS324: Practice With Transformations

April 3, 2020

Here are some example problems to help you practice applying NP-complete transformations from lecture.

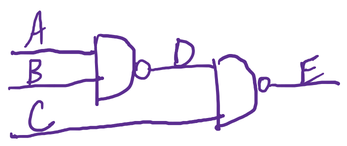
1. Convert the following circuit-SAT instance into a SAT instance



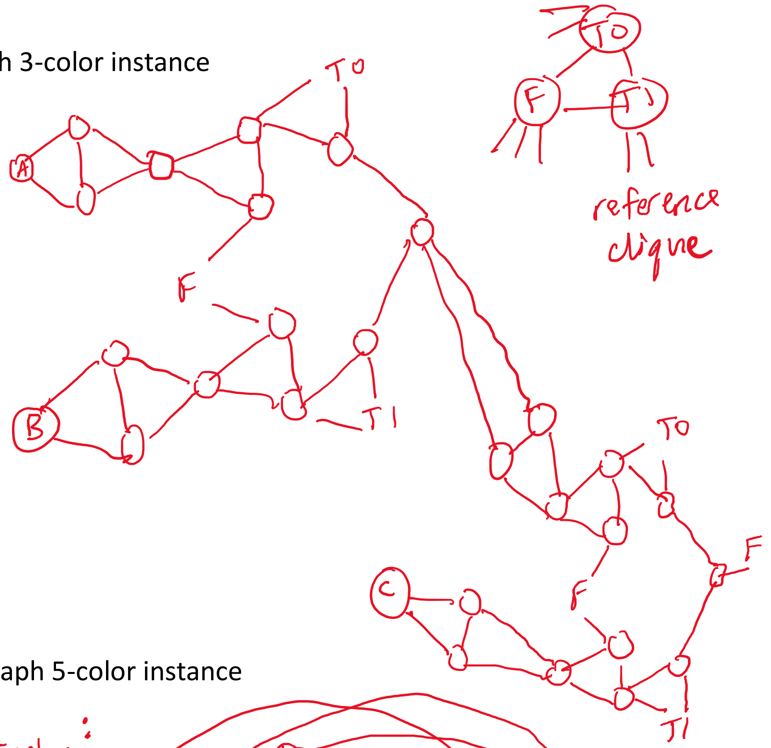
SAT:

$((A \vee F) \wedge (B \vee F) \wedge (\neg A \vee \neg B \vee \neg F)) \wedge$
$((C \vee G) \wedge (D \vee G) \wedge (\neg C \vee \neg D \vee \neg G)) \wedge$
$((G \vee H) \wedge (E \vee H) \wedge (\neg G \vee \neg E \vee \neg H)) \wedge$
$((F \vee I) \wedge (H \vee I) \wedge (\neg F \vee \neg H \vee \neg I)) \wedge$
I

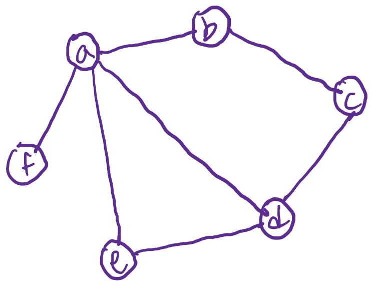
2. Convert the following circuit-SAT instance into a graph 3-color instance



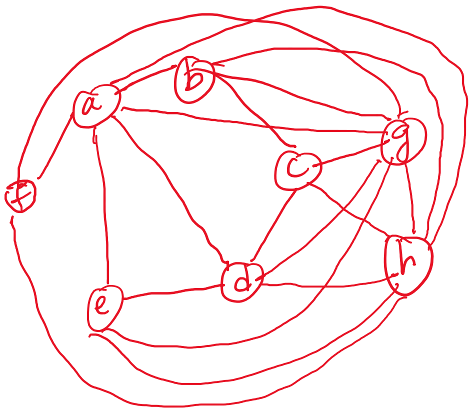
G:



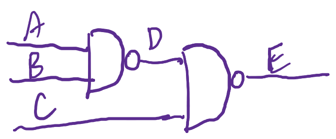
3. Convert the following graph 3-color instance into a graph 5-color instance



G_{5color}:

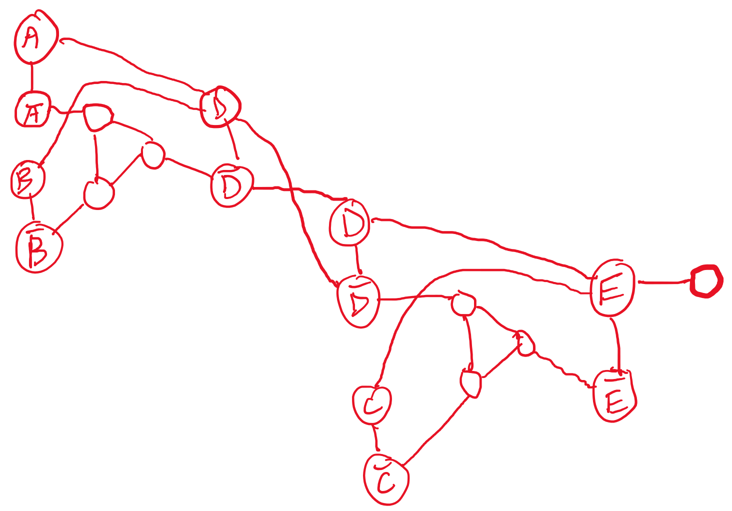


4. Convert the following circuit-SAT instance into a vertex-cover instance. Remember vertex-cover includes both a graph and an integer K.



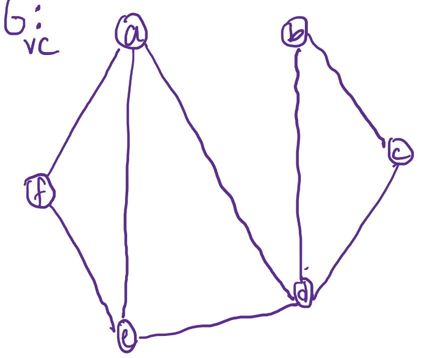
$K = 5 \times 2 = 10$

G:



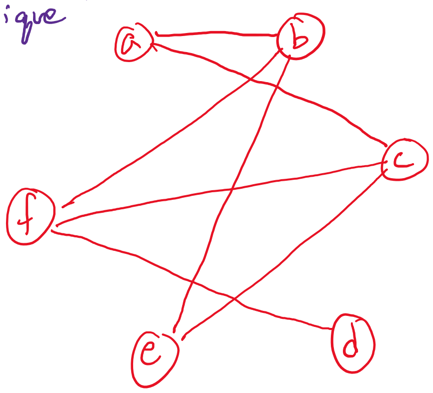
5. Convert the following vertex-cover instance into a clique instance. Remember that clique includes a graph and an integer K.

G_{vc}:



$K_{vc} = 4$

G_{clique}:



$K_{clique} = 2$

CS324: Approximation Algorithms; Vertex-Cover and Traveling Salesman

April 6, 2020

Announcements:

- HW 8 assigned (it is due Wed, April 15 or Tues, April 14 at noon if you want it returned before the fourth midterm exam)
- Skim CLRS ch5 again (review probability and probabilistic algorithms) and read CLRS ch9 before Wednesday

---

Today's learning outcomes:

- ✓ Define the meaning of an approximation algorithm
- ✓ Apply approximation to the Vertex Cover problem
- ✓ Apply approximation to the Traveling Salesman problem

In the past week we looked at several NP-complete problems. It is likely that these problems do not have polynomial-time algorithms (that is if P really does not equal NP) that can return optimal solutions.

Recall that we have a few ways to handle NP-complete problems:
✓ 1. If the problem instances are small, we just go with brute-force search that takes exponential time
✓ 2. If we can restrict the instances (such as graph 3-color to graph 2-color), we may be able to find polynomial-time algorithms.
✓ 3. We can also try to create fast *approximation* algorithms that give us answers that are "good enough". The "good enough" is usually a measure of how close we are to the optimal solution. In practice, we may not need the very best solution – just something we know is 2x the best solution's cost or 3x the best solution's cost. This is called *near-optimality*.

Today's lecture is about approximation algorithms.

First, a definition:

p(n)-approximation: means that the algorithm achieves an approximation ratio of p(n).

Let's assume the approximation algorithm gives us a solution with cost C'. Let's assume the optimal solution has a cost of C.

For problems where we optimize a **maximum**, 0 < C' <= C, so the approximation ratio p(n) is:

$$p(n) = \frac{C \leftarrow \text{optimal}}{C'} \qquad \text{// maximum optimization problems}$$

$\nwarrow$ approximation

For problems where we optimize a minimum, 0 < C <= C', so the approximation ration p(n) is:

$$p(n) = \frac{C' \leftarrow \text{approximation}}{C} \qquad \text{// minimum optimization problems}$$

$\nwarrow$ optimal

1. What are the possible ranges of p(n)?

   a. 0 <= p(n) <= 1

   [✓ b.] 1 <= p(n)

   SUBMIT ANSWER TO MOODLE

An approximation scheme for an optimization problem is an approximation algorithm that takes both the instance of the problem as input PLUS ε > 0 (epsilon) which tells us how far way we are willing to be from optimal. The algorithm is a (1 + ε)-approximation algorithm. So, for a fixed epsilon, the algorithm runs in polynomial time with respect the size N of the problem instance.

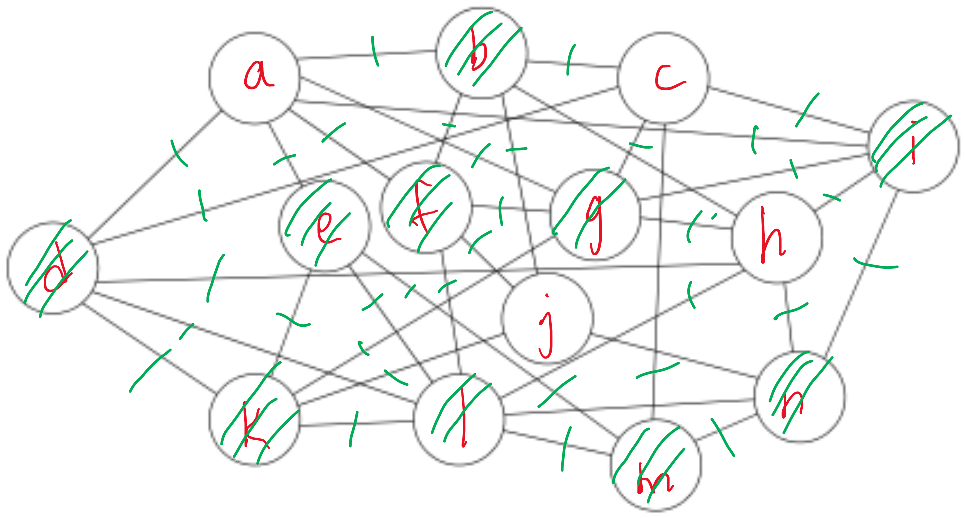
$n^{2/\varepsilon}$

Note: sometimes these approximation algorithms have polynomials that quickly become big as epsilon gets smaller. For example, the runtime might be n raised to (2/ε).

Some approximation schemes are fully polynomial-time if the runtime of the algorithm is polynomial with respect to both n and (1/ε). Any constant-factor decrease in ε results in a constant-factor increase in runtime.

Vertex Cover Approximation Algorithm

Recall the vertex cover problem: given an undirected graph G, what is the smallest number of vertices that "cover" all edges?



```
APPROX-VERTEX-COVER(G)
1     C = NIL        // C is the set of vertices for the cover
2     E' = G.E
3     while E' is not NIL
4           let (u, v) be an arbitrary edge in E' // random pick
5           C = C U {u} U {v}
6           Remove from E' every edge incident on either u or v
7     return C
```

$C = \emptyset$

E'

Let's first see how long this algorithm takes. Suppose the graph G is represented using adjacency lists.

1. $\Theta(1)$

2. $\Theta(|E|)$

3. How long is the loop? $|E| \cdot |E|$ $O(V+E)$

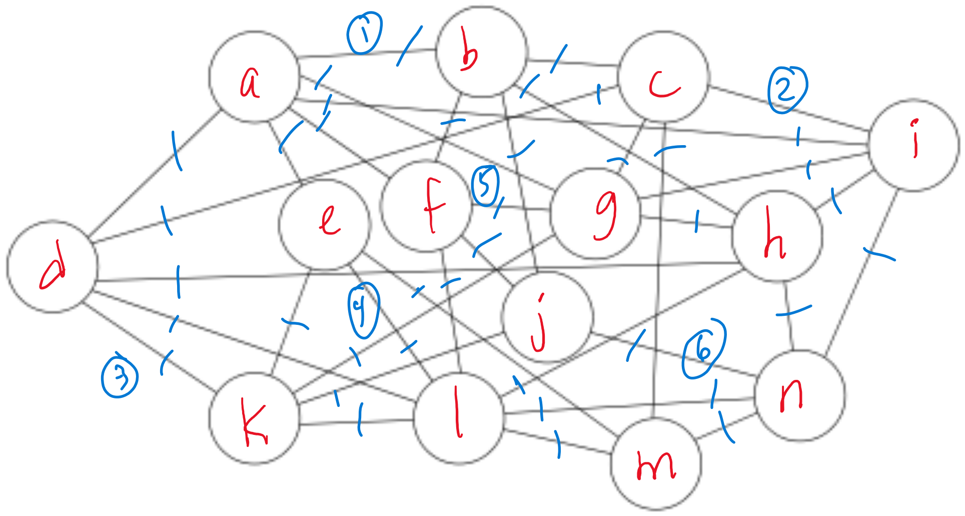
7. $\Theta(1)$

Total: $O(N^2)$

OK, so it runs in polynomial-time with respect to the size of the graph. Note that this does not take an epsilon as input.

Practice: Let's run the approximation algorithm on the graph. Since the algorithm uses randomization, the edge choice is random each time. Once could run this thing several times and return the best of the runs. This is a common practice with any algorithm that uses random choices.

2. Everyone apply the algorithm with a random order of edges. Put your vertex cover below.



a, b, c, i, d, k, e, l, f, g, j, n

Approximate Cover: a, b, c, i, d, k, e, l, f, g, j, n **SUBMIT ANSWER TO MOODLE**

Think about it: How close did we get to the optimal cover?

We have a 2-approximation algorithm. We get a vertex cover that is at most twice the size of the optimal vertex cover. Why?

Proof:

C

C'

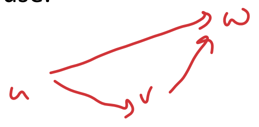
Assume C is the optimal vertex cover. Assume C' is the vertex cover produced by the approximation algorithm. Let E' be the set of edges chosen by the approximation algorithm. The optimal solution C must include at least one vertex from each edge in E' (otherwise, it would not be a cover). No two edges in E' share an endpoint since once an edge is picked in line 4, all other edges that are incident on its endpoints are removed from E' in line 6. Therefore, no two edges in E' are covered by the same vertex in C. |C| >= |E'|. By the approximation algorithm, we choose C' such that it is exactly twice the size of E' (two vertices go in for each edge chosen). Thus, |C'| = 2|E'|. So, we have the relationship |C'| <= 2|C|.

Traveling Salesman

Now, we look at another problem that is NP-complete. While we did not prove this one formally in lecture, it is proved in the textbook in chapter 34 as a transformation from Hamiltonian-circuit.

Here is the formal problem: Given a complete undirected graph G = (V, E) that has a non-negative integer cost c(u, v) associated with each edge, we find a Hamiltonian cycle (a tour) of G with minimum cost.

It is called traveling salesman (salesperson) because one could think of the vertices as houses and edges with travel costs and the salesperson wants to minimize the cost of visiting every house.



Note: we will consider graphs that satisfy the **triangle inequality**:

c(u, w) <= c(u, v) + c(v, w)

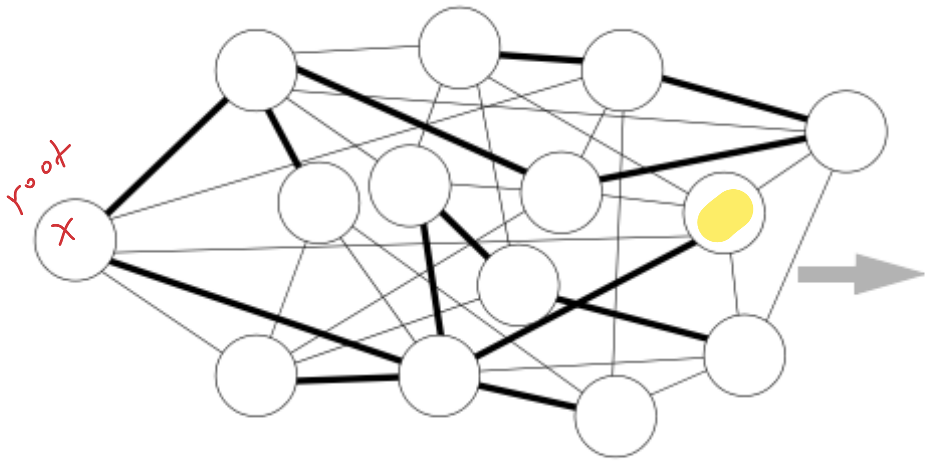
This means that the direct path from u to w is always at least as good as going along a path that detours through v. This assumption is quite natural for graphs where we want to find a good tour. Plus, it turns out that we cannot find a poly-time approximation algorithm with a constant approximation ratio for graphs that do not satisfy the triangle inequality unless P == NP.

Think about it: How might you find a path through the graph that visits all the cities if we allow ourselves to visit a city more than once (no longer a tour)?

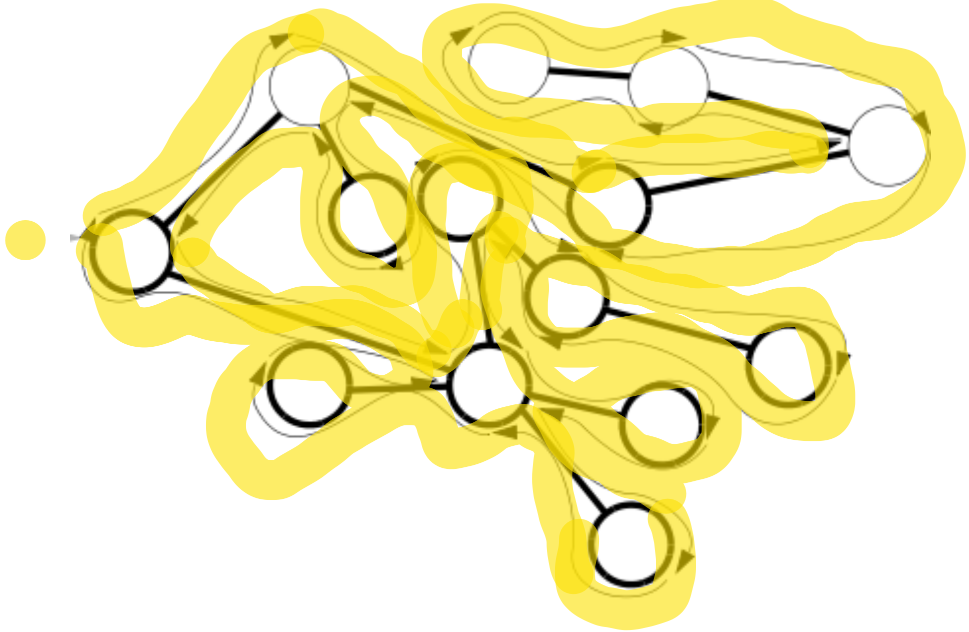
```
APPROX-TSP(G, c)     // G is complete undirected graph; c is the cost function
1       Select vertex r of G.V     // this will be the root vertex of the MST
2       T = MST using PRIM(G, c, r)
3       H = preorder walk of vertices of T, according to when they are first visited
4       Return the order of vertices H for the path
```

So, we can combine some algorithms that we have seen before: Prim to create a minimum spanning tree and then walk that tree in preorder fashion. Remember that preorder visits node first and then recursively calls preorder on the children. In practice, this turns into depth-first-search on the minimum spanning tree.

Example: actual costs are omitted to reduce clutter on graph; bold lines are the minimum spanning tree from left-most vertex.



Walk the minimum-spanning tree



3. How long does the APPROX-TSP take? $\Theta(N \lg N)$

Hint: Prim is the dominant part since walking the tree has twice the edges of the tree

SUBMIT ANSWER TO MOODLE

4. How far are we away from the optimal solution? 2-approximation algorithm

SUBMIT ANSWER TO MOODLE

Proof idea (full proof is in the textbook):

H

$c(T) \leq c(H)$

Let H be the optimal tour of edges. We can create a spanning tree (not necessarily minimum) by removing one edge from H. Thus, c(T) <= c(H) where c is the total cost of edges in the set. The full walk of T traverses each edge twice. Let's call that walk of edges W. Then, c(W) = 2c(T), by definition. Since c(T) <= c(H), c(W) <= 2c(H). The walk W is not a tour, but we can we can remove stops on the walk that we have visited before and the triangle inequality ensures that a tour generated from a walk has less than or equal cost of the walk.

Note: there are better ways strategies to do the TSP tour approximation, but this one is nice to see – it's simple, involves MSTs which we have seen before, and gives us a 2-approximation.

X return composite

3. Did we need all four trials in the above example to determine that 121 is composite?

a. Yes

[✓ b.] No, we only needed 3

SUBMIT ANSWER TO MOODLE

4. Assume you want to know if 89 is composite with Miller-Rabin. You run witness 4 times. How certain are you that 89 is prime?

a. 75%

b. 85%

[✓ c.] 99.6%

d. 100%

$1 - \frac{1}{2^8} = 99.6\%$

SUBMIT ANSWER TO MOODLE

CS324: Probabilistic Algorithms: Median-Finding, Selection, Primality Testing

April 8, 2020

Announcements:

- HW 8 due Wed, April 15 or Tues, April 14 at noon if you want it returned before the fourth midterm exam
- Fourth midterm on Friday, April 17
- Read CLRS chapter 31.1 – 31.3 (should be review about primes, divisors, mod)
- Read CLRS chapter 31.4 – 31.8 for next week's lectures
- No class on Friday due to Good Friday

Today's learning outcomes:

- Review randomness
- See a clever Median-Finding algorithm that runs in linear time
- Expand Median-Finding to probabilistic selection
- Determine if a number is prime with high probability

Last time we saw approximation algorithms to give us something good enough quickly. So, we got a near-correct solution with certainty. Here are some other goals that probabilistic algorithms can give us:

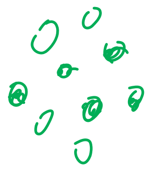
✓ Goal 1: Get a correct solution with certainty quickly (poly-time algorithms)
✓ Goal 2: Get a correct solution with certainly maybe not quickly (brute-force search exponential-time algorithms)
✓ Goal 3: Get a near-correct solution with certainty quickly (approximation algorithms)
✓ Goal 4: Get a correct solution with certainty, probably quickly (**probabilistic algorithms**)
Goal 5: Get a high-probability correct solution quickly (**probabilistic algorithms**)

We have already seen algorithms that fall into Goals 1 through 4. Here are some examples:

Goal 1: Dijkstra's Algorithm for Single-Source Shortest Path, Merge Sort
Goal 2: Try all possible colorings of a graph 3-color problem instance
Goal 3: Approximate Vertex Cover, Approximate Traveling Salesman
Goal 4: Quicksort with pivot as median-of-3 random items (very rare that the pivot will be bad for all splits); We will see find with randomization today ←
Goal 5: We will see Miller-Rabin primality testing today ←

Probabilistic algorithms use randomization. We might apply randomization to **reduce the chance of getting poor worst-case performance** and get better average-case performance. This is exactly what we did with quicksort by choosing a random pivot rather than the value at the low index. Or better yet – the pivot was chosen as the median-of-3 randomly chosen values, so we were guaranteed not to choose the min or the max of the unsorted array.

We can apply randomization to generate a bunch of "random" solution candidates and choose the best of those to continue the next iteration for choosing good solution candidates. Algorithms that use this technique are often called **monte carlo** algorithms. One specific example class of algorithms is called genetic algorithms where solutions are randomly generated and "good" solutions are combined to create good solutions. These types of algorithms are often used in prediction modeling, economics, and artificial intelligence.

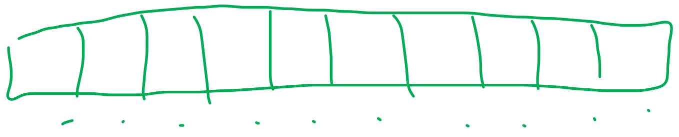


Note: probabilistic algorithms required random number generation. The libraries you use create pseudo-random numbers that begin with a seed and using some arithmetic to create the "random" number. The nice thing about having seeds is that you can debug your programs more easily by getting the same series of random numbers. Note that they are not truly random, though. A Geiger counter could give us true randomness (a capstone team I advised several years ago made a random number generator based on radiation!).

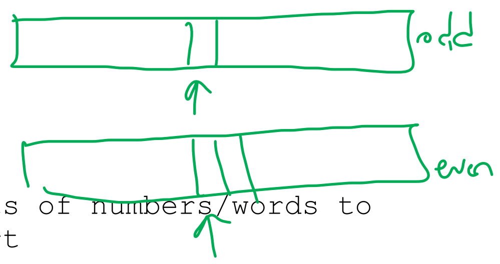
Finding the kth smallest item using median-of-median

First, we will look at how we can find the median element of an unsorted list. While this algorithm does not use randomness, we will modify it to be probabilistic.

Think about it: (This is CS203-level) Given a list L of unsorted items, how can you find the minimum element? How can you find the maximum element? How long does this take? $\Theta(N)$



Think about it: Now, you are given a list L of unsorted items. How do you find the median element? For the purpose of this problem, we will define the median as the middle element of an odd-length list L or the smaller of the two middle elements of an even-length list L.



Natural approach:
```
1. Merge sort L        // cannot assume L consists of numbers/words to
                       // use bucket or radix sort
2. If L.length is odd, return L[(1+L.length)/2]
3. If L. length is even, return L[L.length/2]
```

1. What is the runtime of the natural approach? $\Theta(N \lg N)$
   SUBMIT ANSWER TO MOODLE

Can we do better than this? We would need to eliminate the sorting step.

Think about it: Can you find the median element in L without sorting? yes

Think about it: Can you find the kth-smallest item in L without sorting? yes

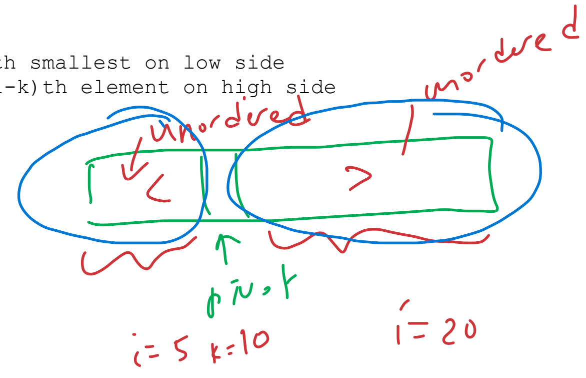
This algorithm may not seem obvious and I just want you to get a sense of how it works. This work was done by five computer scientists: Manuel Blum, Robert Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan. Four of them are Turing-award winners and one was a professor at Stanford. So, do not worry that you could not invent this yourself. The idea is to recursively find the median of medians of sublists that are 5 elements in size.

To keep things simple, we will assume the median is at L.length/2. For even-length lists with 1-indexing, this works. For odd-length lists, we would need to add one to the length before dividing by 2. This is a small detail, so just assume the median is at L.length/2.

Also, this algorithm is more general than just finding the median. It can find the ith smallest item in the list. So, we can run it to find the median, the 106th smallest, the 10th smallest, etc.

$1 \le i \le L.length$

```
Find(L, i):    // return ith smallest item in L
        If L.length < 5
                Sort L
                Return L[i]
        Divide L into 5-element sublists. If there are leftovers, final list has < 5 elements.
        For each sublist, sort the 5 elements to get the median (middle element)
        Put the median elements into list M
        Pivot = Find(M, M.length/2)          // get median of medians for good Pivot
        Partition L around Pivot              // same function as Quicksort
        // We now have a "low side" and a "high side" on either side of the Pivot
        // We make a recursive call on the appropriate side, given index i and index of Pivot
        K = index of Pivot
        If K == i, return Pivot       // we got lucky
        Else if i < K, return Find(L[1..(K-1)], i)    // find ith smallest on low side
        Else, return Find(L[(K+1)..L.length], i-K)    // find (i-k)th element on high side
```



Let's see how this works on an example. Suppose we are trying to find the 6th smallest item in this list.

L =

10	5	2	8	23	11	4	7	20	17	6	13	19	1	15

i = 6

2 5 $\underline{8}$ 10 23   4 7 $\underline{11}$ 17 20   1 6 $\underline{13}$ 15 19

1. Break list into five sublists: sec red

2. Find median of each sublist: 8 $\underline{11}$ 13

3. Find the median-of-medians. Since this has fewer than 5 elements, we just sort and return the middle one.

11

4. This median-of-median gets used as the pivot: 11

5. Partition list around pivot (swap pivot and last element first, so we can use `Partition` from book)

10	5	2	8	23	15	4	7	20	17	6	13	19	1	11

Now, execute Partition (same as Quicksort Partition):

Just before final swap of 11 into place:

10	5	2	8	4	7	6	1	20	17	23	13	19	15	11

After final swap of 11 into place:

10	5	2	8	4	7	6	1	11	17	23	13	19	15	20

9

6. Now that we have the Pivot as 11 and we know this is at index 9, we see that we were not lucky. Which way do we go? To the left. We call find on the following list with i = 6.

10	5	2	8	4	7	6	1

7. Since the list has size greater than or equal to 5, we break into sublists.

8. The median of the first 5 elements is 5 and the median of the last 3 is 6. So we choose the left of the two median, which is 5 to use as the pivot.

9. Move 5 to end, swapping with 1:

10	1	2	8	4	7	6	5

10. Now run Partition, this is before final swap of pivot into place:

1	2	4	8	10	7	6	5

2

After swapping Pivot into place:

1	2	4	5	10	7	6	8

4

11. Now, we get the index of the Pivot, which is 4. 4 is less than 6, so we call Find on the right side of 5 with index value set to 2:

10	7	6	8

12. Because our list has size less than 5, we sort it and return the second smallest, which is 7.

Return 7. Was 7 our 6th smallest item? ___yes___

Practice on your own: Use the same initial array, but find the 10th smallest item instead.

same process as above, 11 is pivot → put at position 9

rightside [17, 23, 13, 19, 15, 20] is list w/ finding index 1

medians: 17, 20 17 is pivot

13 15 20 23

~~20~~, ~~23~~, ~~13~~, 19, ~~15~~, ~~17~~ ← Run partition

17 20

17 is at position 3, so use leftside [13, 15] with index 1

length is less than 5, so return 13

What is the runtime of Find?

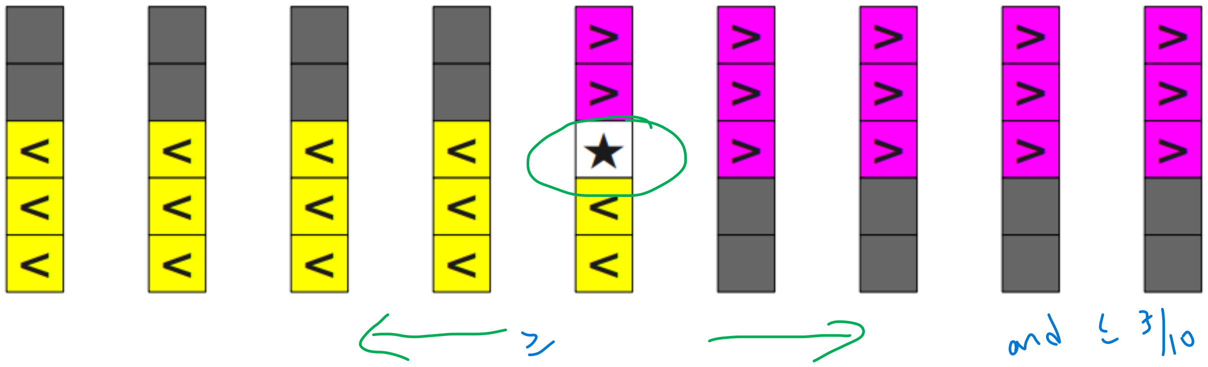
```
Find(L, i):    // return ith smallest item in L
1       If L.length < 5
2               Sort L
3               Return L[i]
4       Divide L into 5-element sublists. If there are leftovers, final list has < 5 elements.
5       For each sublist, sort the 5 elements to get the median (middle element)
6       Put the median elements into list M
7       Pivot = Find(M, M.length/2)          // get median of medians for good Pivot
8       Partition L around Pivot             // same function as Quicksort
        // We now have a "low side" and a "high side" on either side of the Pivot
        // We make a recursive call on the appropriate side, given index i and index of Pivot
9       K = index of Pivot
10      If K == i, return Pivot        // we got lucky
11      Else if i < K, return Find(L[1..(K-1)], i)    // find ith smallest on low side
12      Else, return Find(L[(K+1)..L.length], i-K)    // find (i-k)th element on high side
```

1-3:	Θ(1)	// limited to sorting 5 items
4:	Θ(N)	
5:	Θ(N)	// sorting 5-element lists is constant time per list
6:	Θ(N)	
7:	T(N/5)	// time to execute algorithm on problem of 1/5 the size the original list
8:	Θ(N)	// runtime of Partition
9:	Θ(N)	// go through array to find Pivot's index
10:	Θ(1)	
11-12:	T(S)	// what is the size of S? – depends on how good that pivot is

How good is the median-of-medians pivot?

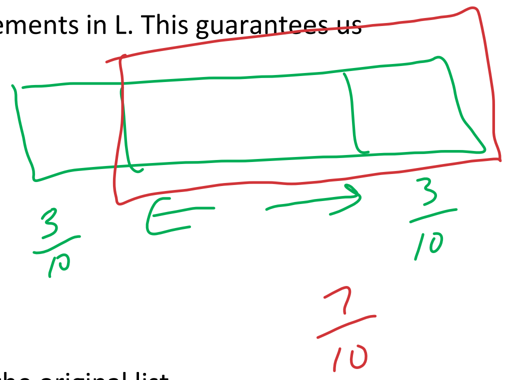
on left-side

Well, the figure below shows a star for the median-of-medians. Each block below represents a 5-element sublist. The star element is greater than 3/5 of the elements (shown in yellow) because we know those medians are smaller than the star. The star element is less than 3/5 of the elements (shown in pink) because we know those medians are greater than the star. on right-side



and $\leq$ 3/10 of all elements in L.

So, that means our median-of-medians ~~is in the middle~~ 3/10 of all the elements in L. This guarantees us a "good" pivot, so we eliminate 3/10 of the list for the recursive call.



Now, we have all we need to do the analysis of Find:

Runtime of Find:

1-3:	Θ(1)	// limited to sorting 5 items
4:	Θ(N)	
5:	Θ(N)	// sorting 5-element lists is constant time per list
6:	Θ(N)	
7:	T(N/5)	// time to execute algorithm on problem of 1/5 the size the original list
8:	Θ(N)	// runtime of Partition
9:	Θ(N)	// go through array to find Pivot's index
10:	Θ(1)	
→ 11-12:	T(7N/10)	// we eliminate 3/10 of the list with the good Pivot

$$\text{Total runtime: } T(N) = \underline{T\left(\frac{N}{5}\right)} + \underline{T\left(\frac{7N}{10}\right)} + \underline{\Theta(N)}$$

$$T(N) = \Theta(N)$$

Note: the textbook analysis is more careful with ceilings and an additional constant, but the above recurrence gives us a runtime that is asymptotically the same as the one in the book. We can then use the substitution/induction method to prove the recurrence is O(N) – remember induction?. Or you can see that if we combine the subproblems into solving one subproblem of size [N/5 + 7N/10] = 9N/10, we get something that the master method can solve.

Wow, we can find ANY ith-smallest element of a list in LINEAR time!!!! Pretty cool.

Using Randomization Instead of Median-of-Medians

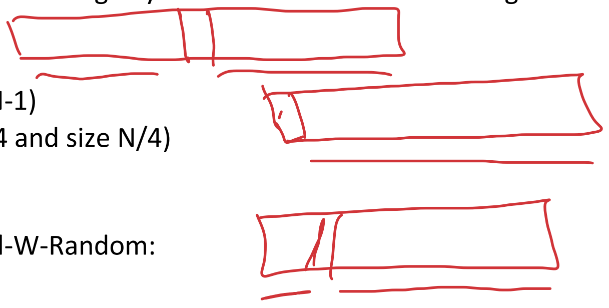
We can tweak the Find algorithm above and instead of finding the median-of-medians, we just choose a random element of the list.

```
Find-W-Random (L, i):  // return ith smallest item in L
    If L.length < 5
        Sort L
        Return L[i]
    Pivot = Random element from L[i]       // randomly select Pivot
    Partition L around Pivot               // same function as Quicksort
    // We now have a "low side" and a "high side" on either side of the Pivot
    // We make a recursive call on the appropriate side, given index i and index of Pivot
    K = index of Pivot
    If K == i, return Pivot        // we got lucky
    Else if i < K, return Find-W-Random (L[1..(K-1)], i) // find ith smallest on low side
    Else, return Find-W-Random (L[(K+1)..L.length], i-K) // find (i-k)th element on high side
```

Now, what is the expected runtime of this version?

Well, on average, where will the pivot be? any location in array

So, we have a 1/N chance of the pivot being at any element. On average, we will split the list into two parts of size 3N/4 and N/4. The textbook shows this analysis on pages 218 and 219 if you want to see the math. Intuitively, we have the same probability of selecting any element. Since we are cutting the list into two parts:



- Best-case: split evenly (size N/2 for each side)
- Worst-case: split with terrible pivot (size $\overset{0}{1}$ and size N-1)
- Average-case: split halfway between these (size 3N/4 and size N/4)

This gives us the following recurrence for the runtime of Find-W-Random:

$$T(N) = T\left(\frac{3N}{4}\right) + \Theta(N)$$

Via the Master Method, this has an asymptotic solution of Θ(N).

Therefore, the expected runtime (using randomness) of Find is linear. This is much easier to code than the median-of-median approach and we should, on average, get linear runtime.

$T(N) = T(N-1) + \Theta(N)$

2. What is the worst-case runtime of Find (using randomness)? $O(N^2)$

SUBMIT ANSWER TO MOODLE

Miller-Rabin Probabilistic Algorithm for Determining if a Number is Prime

Prime numbers are an important to cryptographic algorithms. One of the algorithms requires producing large "random" prime numbers. We will look at more cryptography next week. For today, we will focus on determining if a given number is prime.

Think about it: How would you determine if a number N is prime?

$1\ 2\ 3\ 4\ 5\ 6 \cdot \frac{N}{2} \cdot\ \ N$

$1\ 2\ 3\ 4\ 5 \ldots \sqrt{N}$

Straightforward: try all possible divisors from 1 to N/2.

A little faster: try all possible divisors from 1 to sqrt(N).

If N is REALLY big, we may not have the time/resources to try all possible divisors. Since this is the probabilistic algorithms lecture, we will use randomization to give us a solution with high probability.

This algorithm uses a **witness** function, which takes two parameters: a random number **a** between 1 and N-1 and N. If it returns false, N may be either prime or composite. If it returns true, N is definitely composite.

It turns out that the way the random number **a** relates to N using modular exponentiation gives us information about N being prime or not. At least 75% of the random selections for the a will return true for WITNESS if N is composite. The number theory for the math is beyond the scope of this course, but just know that this algorithm makes use of this fact below.

$1 \le A \le N-1$

```
WITNESS(~~a~~ A, N):      // assume N is > 2 and odd (otherwise, it is not prime)
    Write N = 1 + $2^s d$, where s is a positive integer and d is an odd integer
    Return false if any of the following is true:
        $A^d$ mod N = 1
        $A^d$ mod N = N-1
        $A^{2d}$ mod N = N-1
        $A^{4d}$ mod N = N-1
        $A^{8d}$ mod N = N-1
        …
        $A^{(N-1)/2}$ mod N = N-1
    Return true
```

```
MILLER-RABIN(N, t): // t is number of trials for witness
    For i = 1 to t
        a = RANDOM(1, N-1)
        if WITNESS(a, N)
            return composite
    return prime
```

Remember dice-rolling? Let's analyze the probability of us getting the correct answer from MILLER-RABIN.

Refresher: What is the probability of rolling a 1 on a fair die? $\frac{1}{6}$

What is the probability of rolling a 1 following by rolling a 1? $\frac{1}{6} \times \frac{1}{6}$

What is the probability of rolling three 1's in a row? $\frac{1}{6} \times \frac{1}{6} \times \frac{1}{6}$

OK, you get the idea.

Let's see how this idea applies to MILLER-RABIN. Recall that the witness will return true if the number is composite. So, each time witness returns false, we are more and more sure that the number N is prime.

Here's the probability if N is composite, we get false from witness in the ith iteration:

Iteration i	Probability of getting false
1	1/4
2	1/16
3	1/64
N	1/(2^(2n))

$\frac{1}{2^{2n}}$

Thus, if we want to be certain to 0.9999999999999999999 that our number is prime, we need to call witness 35 times.

This is an example of an algorithm that gives us a high-probability correct answer quickly.

Example:

Is 121 composite?

Let's assume t=4 and our randomly generated a values are 27, 40, 19, and 16.

121 = 1 + 2^Sd, so S is 3 and d is 15.

$120 = 2^s d$ $s = 3$ $d = 15$

Our powers within witness are: 15, 30, and 60

Check these values:

$a = 27$

$27^{15} \bmod 121 = ? = 1$ // return false

$a = 40$

$40^{15} \bmod 121 = ? = 120$ // return false

$a = 19$

$19^{15} \bmod 121 = 109$

$19^{30} \bmod 121 = 23$ // return true

$19^{60} \bmod 121 = 45$

X return composite

3. Did we need all four trials in the above example to determine that 121 is composite?

a. Yes
[✓ b.] No, we only needed 3

SUBMIT ANSWER TO MOODLE

4. Assume you want to know if 89 is composite with Miller-Rabin. You run witness 4 times. How certain are you that 89 is prime?

a. 75%
b. 85%
[✓ c.] 99.6%
d. 100%

$1 - \frac{1}{2^8} = 99.6\%$

SUBMIT ANSWER TO MOODLE

CS324: Cryptographic Algorithms, RSA, Repeated Exponentiation

April 13, 2020

Announcements:

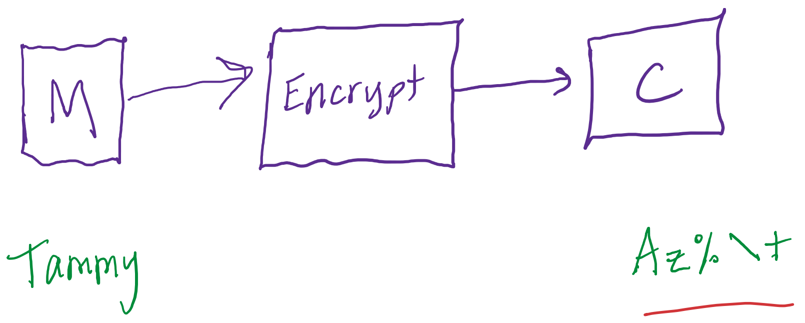
- HW 8 due Wed, April 15 or Tues, April 14 at noon if you want it returned before the fourth midterm exam
- Fourth midterm on Friday, April 17
- Review during class on Wed, April 15

Today's learning outcomes:

- Overview of encryption and decryption
- Describe key schemes
- Understand RSA algorithm for public-key scheme
- Use repeated exponentiation and Fermat's Theorem to compute X^Y mod N

Some basics with cryptography:

Plaintext: actual message M
Ciphertext: encrypted message (hopefully, looks like garbage) C



A simple scheme that you may have used to write notes in code to your friends. Assume English letters are used in the message. An encryption scheme would be moving K letters to the right, with wrap-around from Z to A.

K = 2
M = I LOVE CS

1. What is C? K NQXG EU

UPLOAD ANSWER TO MOODLE

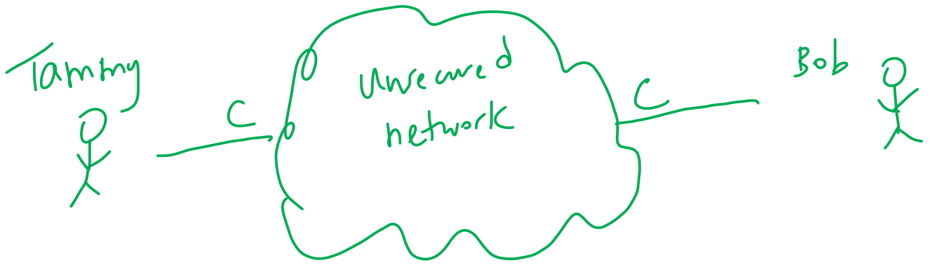
How would we decrypt C? K NQXG EU

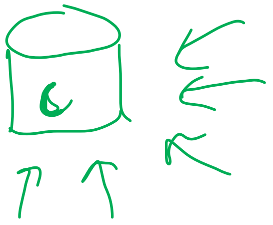
M = I LOVE CS

In general, we have the following:
Decrypt(Encrypt(M, key_e), key_d) = M

$$\underline{M} \xrightarrow{E_k} \underline{C} \xrightarrow{D_k} \underline{M}$$

Why is encryption useful? (Hint: networks and databases)

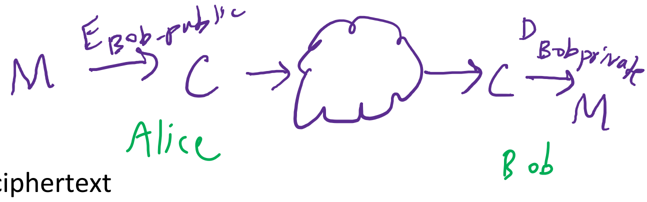




In general, there are two main schemes for encrypting and decrypting data:

Secret Key: key is secret; only sender and receiver know key; key is used for encryption and decryption (see example above)

Public Key: public key is published and used to encrypt plaintext
private key is secret and used to decrypt ciphertext



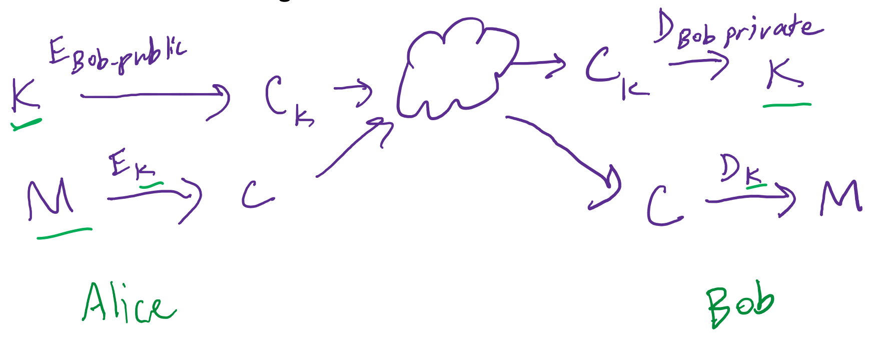
Suppose Alice wants to send an encrypted message to Bob.

1. Alice looks up Bob's public key
2. Alice encrypts plaintext with Bob's public key to create ciphertext
3. Ciphertext is sent to Bob
4. Bob decrypts ciphertext with private key to create plaintext
5. Bob reads the message

Public key systems can be slow in terms of computation, so we can combine the schemes.

Suppose Alice wants to send a **large** encrypted message to Bob.

1. Alice looks up Bob's public key
2. Alice generates a random secret key K (which is short)
3. Alice encrypts K with Bob's public key
4. Alice sends encrypted K to Bob
5. Meanwhile, Alice encrypts message with secret key K (which is faster) and sends ciphertext to Bob
6. Bob decrypts secret key K with his private key
7. Bob decrypts ciphertext using secret key K
8. Bob reads the message



RSA Public Key System

We will look at one algorithm to create public and private keys. It is called RSA, named after its inventors: Rivest/Shamir/Adleman. Note that Rivest is one of the authors of the textbook.

The mathematical premise for this algorithm is that it is difficult to factor a large number N that is a product of two prime numbers.

$$N = p_1 \cdot p_2$$

<u>Here is how we can create keys:</u>

1. Choose large random prime numbers p and q. Note that Miller-Rabin from the last lecture can be used to find (with high confidence) prime numbers.

2. n = pq

3. Φ = (p-1) (q-1)

3. Select small integer e that is relatively prime to Φ.

4. Find integer d in [1, …, Φ-1] such that de = 1 mod Φ.

5. Publish public key P = (e, n).

6. Keep private key S = (d, n).

To encrypt plaintext:
If the message M (in bits) is longer than n, break it into chunks with size <= n and encrypt each chunk separately. M can be interpreted as a number in binary. To create ciphertext:

$$C = \underline{M^e mod\, n}$$

To decrypt ciphertext:
We can get the plaintext M' from C as follows:

$$M' = \underline{C^d mod\, n}$$

If there are no bit transmission errors of the ciphertext, then M should equal M'. The math behind why this works is in the textbook. I expect that you can apply this algorithm and not prove why it works.

Example:

Assume p = 41 and q = 67.

n = (41)(67) = 2747

Φ = (40)(66) = 2640

Choose e = 7. Since 7 is prime and does not divide 2640 evenly, we know 7 and 2640 are relatively prime.

Now, we need to find d. Recall that 7d = 1 mod 2640. So, we try 6 possible candidates:

- 2640*1 + 1 = 2641 // 2641 mod 7 = 2
- 2640*2 + 1 = 5281 // 5281 mod 7 = 3
- 2640*3 + 1 = 7921 // 7921 mod 7 = 4
- 2640*4 + 1 = 10561 // 10561 mod 7 = 5
- 2640*5 + 1 = 13201 // 13201 mod 7 = 6
- 2640*6 + 1 = 15841 // 15841 mod 7 = 0

Note: d cannot be 0

So, we found that 15841 is divisible by 7.

d = (15841) / 7 = 2263

$2263 \cdot 7 \equiv 1 \mod 2640$

$d \cdot e \equiv 1 \mod \phi$

We have found our keys:
Public key P = (7, 2747)
Secret key S = (2263, 2747)

Let's see if they work for encrypting and decrypting. Assume M = 1234.

$$1234^7 \; mod\; 2747 = 517$$

Thus, the ciphertext is 517. Let's decrypt 517:

$$517^{2263} \, mod \; 2747 = 1234$$

That's good news – we got our original message back. But, look at that exponent of 2263. That's quite large. In practice, these exponents can get really large.

If we just try to do 517^2263, we may run out of space on a calculator or computer. Or it might overflow.

Repeated Squaring / Exponentiation

We can use the technique of repeating squaring to reduce our work. First, let's look at an example with ordinary integers without any mods.

Assume we want to calculate:

$$5^{21} = \overbrace{5 \times 5 \times 5 \times 5 \times \ldots 5}^{20}$$

We can find all powers of 5 that are powers of 2, up to the exponent value:

$$5^1 = 5$$
$$5^2 = (5^1)^2 = 25$$
$$5^4 = (5^2)^2 = (25^2) = 625$$
$$5^8 = (5^4)^2 = (625^2) = 390625$$
$$5^{16} = (5^8)^2 = (390625^2) = \underline{152587890625}$$

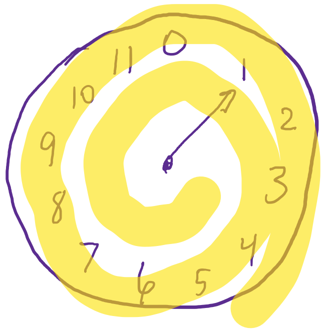
We know from math that:

$$5^{21} = \underline{5^{16}} * \underline{5^4} * \underline{5^1}$$

$$= 152587890625 * 625 * 5 = 476837158203125$$

Perhaps that did not save us much work, but it will save us from numbers getting really when we use mods.

Modulus is like a clock – it keeps wrapping around:



examples

25 mod 7 = 4

32 mod 5 = 2

21 mod 4 = 1

We can put that to use with the following identities:

$$(\underline{(a \bmod p)} + \underline{(b \bmod p)}) \bmod p = \underline{(a + b)} \bmod p$$

$$(\underline{(a \bmod p)} \times \underline{(b \bmod p)}) \bmod p = \underline{(a \times b)} \bmod p$$ ☆

It also works for subtraction. Unfortunately, it does not work for division.

Let's try one.

250 mod 7 = $((\overset{4}{\underline{25}} \bmod 7) * (\overset{3}{\underline{10}} \bmod 7)) \bmod 7 = \overset{12}{4*3} \bmod 7$ = 12 mod 7 = 5

2. You try one. Use the multiplicative property about mod to solve 250^2 mod 7: ___4___

(250 mod 7)(250 mod 7) mod 7
= (5 x 5) mod 7
= 25 mod 7
= 4

UPLOAD ANSWER TO MOODLE

Repeated Squaring with Mod

OK, now we have the tools to make repeated squaring easier.

Let's try it:

$$15^{32} \bmod 19 = \underline{\ \ ?\ \ }$$ 17

Calculate the repeated squares:

$15^1 = 15 \bmod 19 = 15$
$15^2 = 225 \bmod 19 = 16$
$15^4 = 16^2 \bmod 19 = 256 \bmod 19 = 9$
$15^8 = 9^2 \bmod 19 = 81 \bmod 19 = 5$
$15^{16} = 5^2 \bmod 19 = 25 \bmod 19 = 6$
$15^{32} = 6^2 \bmod 19 = 36 \bmod 19 = 17$

3. Suppose we are taking mod 19 as in our calculation above. How large can our intermediate result get? $0 \ldots 18$
Hint: what can mod 19 return? We can square this number.

18^2

UPLOAD ANSWER TO MOODLE

OK, so now we have a faster way to calculate things in the form:

$$\underline{X^Y mod\ N}$$

Which comes up in the RSA public key scheme.

prime

Optimization if N is prime

Due to Fermat's Theorem, we can optimize the calculation further if N is prime.

Theorem: if p is prime:

$$a^p = a\ mod\ p$$

Further if a > 0 and p does not divide a:

$$\underline{a^{p-1} = 1\ mod\ p}$$

We can use this result to get exponents cycled back around to values that are much smaller.

Example:

$$3^{1000}\ mod\ 23 = \_\_\_\_\_\_\_\_$$

Well, we could just do our regular exponentiation as above. But, because 23 is prime, we can use the second version of Fermat's Theorem above.

$$3^{1000} mod\ 23 = \underline{3^{(22*45)}3^{10}}\ mod\ 23$$

$22 \times 45 = 990$

$3^{990} \cdot 3^{10} = 3^{1000}$

We know that 3^{22} is equal to 1, so this reduces to: $1 * 3^{10}\ mod\ 23$

$$1 * 3^{10}\ mod\ 23$$

$3^{22 \times 45}\ mod\ 23 = 1$

We can re-write this shortcut as:

$$3^{\underline{(1000\ mod\ 22)}}\ mod\ 23 = 3^{10}\ mod\ 23$$

Now, we can use repeated squaring to get 3^{10} mod 23:

$3^1 = 3 \bmod 23 = 3$
$3^2 = 9 \bmod 23 = 9$
$3^4 = 9^2 \bmod 23 = 81 \bmod 23 = 12$
$3^8 = 12^2 \bmod 23 = 144 \bmod 23 = 6$

$3^8 \cdot 3^2 = 6 \times 9 = 54 \bmod 23$
$= 8$

$$\begin{array}{r} \overset{4}{\cancel{8}}14 \\ -46 \\ \hline 8 \end{array} \qquad \begin{array}{r} 23 \\ 23 \\ \hline 46 \\ 23 \\ \hline 69 \\ 23 \\ \hline 92 \end{array} \qquad \begin{array}{r} \overset{7}{\cancel{8}}1 \\ -69 \\ \hline 12 \end{array}$$

Practice: Use Fermat's Theorem to optimize the following calculation:

15^{32} *mod* 19 = 17

$= 15^{(32 \bmod 18)} \bmod 19 = 15^{14} \bmod 19$

$15^1 = 15 \bmod 19 = 15$
$15^2 = 15^2 \bmod 19 = 225 \bmod 19 =$
$= 16$
$15^4 = 16^2 \bmod 19 = 256 \bmod 19$
$= 9$

Next time: We will look at an algorithm to create secret keys.

$15^8 = 9^2 \bmod 19 = 81 \bmod 19$
$= 5$

$15^{14} \bmod 19 = (15^8 \times 15^4 \times 15^2) \bmod 19$
$= (5 \times 9 \times 16) \bmod 19$
$= (45 \bmod 19) \times (16 \bmod 19) \bmod 19$
$= (7 \times 16) \bmod 19$
$= 17$

// same answer as earlier in notes

CS324: One-Time Pads, Feistal Ciphers, Other Ways to Encode Secret Information

April 15, 2020

Announcements:

- Fourth midterm during class this Friday, April 17
- Monday and Wednesday next week – join MS Teams during the class time for an in-class activity to practice technical interviews; each day is worth 15 points of HW points
- Final exam for Section A is Tuesday, April 28 from 1:30 to 3:30pm
- Final exam for Section B is Monday, April 27 from 1:30 to 3:30pm

---

Today's learning outcomes:

- Review secret-key scheme for encryption
- Understand one-time pads and bit XOR
- See process of Feistal Ciphers
- See other ways to encode secret information

Review: In the last lecture, we looked at RSA which is an algorithm to create public and private keys. A second way to encrypt and decrypt information is through the use of a shared, secret key.

Now the model is: two or more trustworthy entities share a secret key. The secret key is used to encrypt and decrypt messages.

One-Time Pad

The strongest method for scrambling data is the use a one-time pad. The "key" is shared and kept secret between the two people. The key is also the same bit-length as the message, so a message that is very long will need a long key.

How to encrypt:

Message (plaintext):

0	1	1	1	0	1	0	0

Secret shared key:

1	0	1	0	0	1	1	0

Bitwise-XOR Message with Key to get Ciphertext:

1	1	0	1	0	0	1	0

How to decrypt:

Ciphertext:

1	1	0	1	0	0	1	0

Secret shared key:

1	0	1	0	0	1	1	0

Bitwise-XOR Ciphertext with Key to get Message:

0	1	1	1	0	1	0	0

Advantage: Ciphertext will be scrambled and completely secure as long as no one else has the secret key

Disadvantage: Key is as long as message

Disadvantage: A key can only be used once (so Alice and Bob would need to have some way to get the shared key or generate shared keys algorithmically, like google authentication codes keep getting generated)

Why can we not use the same key twice?

Two encrypted messages could be XORed, which gives us the result of two unencrypted messages

Example of why it is a one-time pad:

Suppose M1 = 0001 and M2 = 1001. Suppose the same secret key K = 1101 is used on both messages.

M1 0001
Key 1101

1100 = C1

M2 1001
Key 1101

0100 = C2

M1 0001
M2 1001

1000

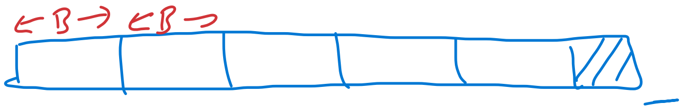
C1 1100
C2 0100

1000

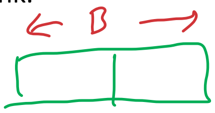
So, if someone gets both ciphertexts, they could derive some information about the two messages without even having the secret key.

Feistal Ciphers

This technique was developed by Horst Feistal and is the basis for many popular encryption protocols such as the Data Encryption Standard (DES) and Blowfish. More modern encryption protocols use this foundational idea of a cipher.



Idea: It is a block cipher, so data is chunked into fixed-length blocks and encrypted per chunk.



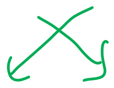
Idea: Inner loop XORs one half of the block with the hash of the other block.

- Size of hash-result should be half the size of the block
- Hash function typically uses the encryption key as one of its input parameters

Idea: Encryption loop alternates roles between two halves – each half is hashed several times

Idea: Decryption is done by running algorithm backwards, reversing roles of left and right

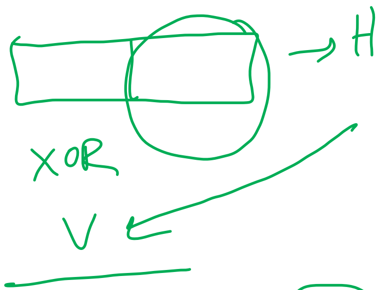
- XOR is self-cancelling: (A XOR B) XOR B == A



Idea: Encryption is done over multiple rounds

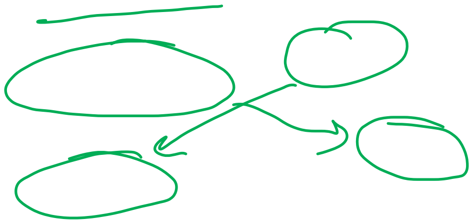
Each round:

1. Hash* right-half block with encryption key; get hash-value V whose size is the same as the right-half of the data.
2. XOR hash-value V with left-half block
3. Swap left and right half blocks



*hash function is usually different for each round

Here is a very small example, so you can see the process.



Data: 8-bit block [10110101]
Key: 0110
Hash function (this one is a poor choice – just for the example):

```
H(a, b):
      X = a + b
      X = 3 * X
      X = Adds overflow bits to low-order bits of result
      Return X
```

Example of H:
H(0011, 1110) =

X = 0011 + 1110, so X = 10001
X = 3 * X, so X = 110011
Overflow bits are 11, so we add to the number 0011 to get 0110
Return 0110

M: 1011|0101

OK, here we go. Let’s assume 6 rounds.

key

Left Bits	Right Bits	H(0110, Right)
1011	0101	0011
0011 XOR = 1000		
0101	1000	1100
1100 XOR = 1001		
1000	1001	1111
1111 XOR = 0111		
1001	0111	1001
1001 XOR = 0000		
0111	0000	0011
0011 XOR = 0100		
0000	0100	1111
1111 XOR = 1111		
0100	1111	

Note: this hash function H would typically change on each round, but to keep the example simple, it is the same one as described above.

1. So, the encrypted ciphertext is: 01001111

UPLOAD TO MOODLE

6 rounds

Now, let’s decrypt our ciphertext by reversing the roles of left and right.

key

Right Bits	Left Bits	H(0110, Left)
1111	0100	1111
1111 XOR = 0000		
0100	0000	0011
0011 XOR = 0111		
0000	0111	1001
1001 XOR = 1001		
0111	1001	1111
1111 XOR = 1000		
1001	1000	1100
1100 XOR = 0101		
1000	0101	0011
0011 XOR = 1011		
0101	1101	

Thus, the original message is: 11010101 // note the left and right

Your turn to practice: Encrypt the same message with the same hash function, but with key 1010 instead. Use 6 rounds.

Here is the list of the hash-table values for key 1010, so you do not need to compute this yourself.

hash(1010,value)	
value	result
0000	1111
0001	0011
0010	0110
0011	1001
0100	1100
0101	1111 C
0110	0011
0111	0110
1000	1001
1001	1100
1010	1111
1011	0010
1100	0110
1101	1001
1110	1100
1111	1111

Left Bits	Right Bits	H(1010, Right)
1011	0101	1111
1111 XoR = 0100		
0101	0100	1100
1100 XoR = 1001		
0100	1001	1100
1100 XoR = 1000		
1001	1000	1001
1001 XoR = 0000		
1000	0000	1111
1111 XoR = 0111		
0000	0111	0110
0110 XoR = 0110		
0111	0110	

2. What is the encrypted result? 0111 0110

UPLOAD TO MOODLE

Design goals for hash functions for ciphers:

- Fast computation
- Not reversible (cannot go from hash-value back to message easily)
- Unlikely to have two different messages with same hash-value
- Return at least 80 bits

WARNING: Do not try to attempt to create your own cipher/encryption system. Use well-established protocols. The topics of these lectures is to give you an understanding of how the algorithms are developed.

Other Ways to Encode Secret Information

People have been "talking" in code/secret for a long time – way before computers.

Steganography – hiding messages in plain sight

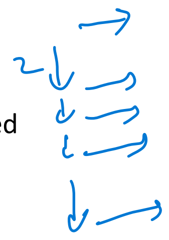
Here are some practices of hiding messages in plain sight:

- Shave hair, write message on bald head, grow hair, shave hair to deliver the message
- Invisible ink on paper
- Knitting a message in morse code in a scarf
- Encoding a message in a picture by length of grass blades

3. Can you think of other ways to hide information in plain sight?

UPLOAD TO MOODLE

In the computer world, it is somewhat easy to embed secret messages in ordinary files, especially audio/video files. One can also embed a secret in the timing of releasing information – bits represented by gap between messages.



In a digital photo:

- Put message in lowest bit of every red value per pixel – no one will notice if the red component is off by 1 in the image.

RGB
0110111

Some advantages: no one thinks to look at these files for hidden messages

Disadvantage: much less space efficient

Digital Signatures – Can "sign" documents/files to ensure that the author is who they state they are

We can use RSA in reverse to sign a plaintext message M:

$$\begin{tikzcd} M \arrow[r, "D_{private}"] & G \arrow[r, "E_{public}"] & M \end{tikzcd}$$

1. Decrypt M with my private key – this produces "garbage". Send "garbage" to Alice.
2. Alice uses my public key to encrypt "garbage". If it comes out as the plaintext message M, Alice knows that I signed it.

If we really want to be secure, I could encrypt the garbage with Alice's public key. Alice would then decrypt garbage with her private key and then encrypt the result using my public key.

$$\begin{tikzcd} M \arrow[r, "D_{Bob private}"] & G \arrow[r, "E_{Alice\_public}"] & C \arrow[r] & C \arrow[r, "D_{Alice\_private}"] & G \arrow[r, "E_{Bob\_public}"] & M \end{tikzcd}$$

Conclusions

Again, do not design/implement your own security protocols. Use well-established libraries and standards that have been tested.

If this is interesting to you, consider taking the Cryptography course from the Math department and the Computer Security elective course in the CS program. You may also want to take Number Theory from the Math department, as this is the basis for designing key systems and good hash functions.

CS324: Technical Interview Practice

April 20 and 22, 2020

Announcements:

- This Friday is our last lecture; bring review questions to Teams on Friday
- Final exam for Section A (MWF 9:15) is Tuesday, April 28 from 1:30 to 3:30pm
- Final exam for Section B (MWF 12:30) is Monday, April 27 from 1:30 to 3:30pm

---

Today's learning outcomes:

- Practice communication skills through interview practice
- Explain your thinking
- Use skills from this course to solve problems

Activity: You will be divided into pairs (or a group of 3) and each take a turn of being the interviewer and interviewee. You will not be graded on your answers to the questions, but you will earn 15 homework points for completing this activity and submitting the information below.

What to submit:

- Microsoft Teams video inside your private channel (be sure to click on the "Record" meeting option when you start your video call). Include a link to the video in your channel – it should automatically save the video recording there, but please double check that the link is there.
- Complete the Moodle "quiz" reflection questions before midnight.

Find out whose birthday is coming up the soonest. That person is Person A. Order yourselves as A, B (and C, if in a 3-person group).

Person A: __________________

Person B: __________________

Person C (if group of 3): ___________________

Two-person groups: Each interview should last 20 – 25 minutes, so do not ask a new question after the 20-minute mark.

Three-person groups: Each interview should last 15 – 20 minutes, so do not ask a new question after the 15-minute mark.

As the interviewer (or observer): Make notes about what you think the person did well in terms of presenting their ideas. As the interviewee, pretend this is a formal video technical interview and offer your thoughts and ideas as you would in an interview. You can assist by asking the interviewee to explain their thinking as they are solving the problem.

As the interviewee: Try not to peek at the questions you will be asked, so it is a more authentic experience. Tammy already gave you some practice questions for coding challenges this semester (sorting, 0-1 knapsack, set partition). The interview questions for this activity are more in the "explain your thinking" category instead of coding up a solution. Where appropriate though, you may use your whiteboard option (go to Share->Whiteboard during your MS Teams meeting) to diagram and show calculations.

At the end: Thank your groupmates and answer the reflection questions on the Moodle quiz.

Some of the interview questions come from the book, *Cracking the Coding Interview*. Other good interview resources include HackerRank.com, leetcode.com, and geeksforgeeks.com.

CS324: Technical Interview Practice

April 20, 2020

Questions for Interviewer A interviewing Person B:

1. Tell me about a time you had a persuade a group of people to make a big change.

2. Suppose you have a fixed-size array of size N in your code. It has become full and you need to add more data. What size will you use to create a new array for the copied data and empty cells? Why did you choose that size?

3. Suppose you have two sorted arrays, A1 and A2. Design a linear-time algorithm to find the number of elements in common. Provide an analysis as to why your algorithm runs in linear time.

4. Suppose you are given 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. You have a scale that provides exact measurement and you can use the scale exactly once. How do you find the heavy bottle?

5. Describe the implementation of a function to return all subsets (aka the power set) of a set {0, 1, 2, 3, …, k}.

CS324: Technical Interview Practice

April 22, 2020

Questions for Interviewer B interviewing Person A (or C for 3-person group):

1. Describe a time when you used randomization to improve the efficiency of a coding project or problem.

2. Describe how you have taken a recursive solution and turned it into an iterative solution.

3. Describe an algorithm to determine if a binary tree is balanced. A balanced tree is a tree where the two subtrees of any node never differ by more than one.

4. You have a 5-quart jug, a 3-quart jug, and an unlimited supply of water (but no measuring cups). How would you get exactly 4-quarts of water? They are milk jugs, so filling to “halfway” up or 80% up will not work.

5. You are given an unsorted array of size N with all the numbers 1 through N+1 appearing exactly once, except for one of those numbers is missing. Design an algorithm to find the missing number in O(N) time and O(1) extra space.

CS324: Technical Interview Practice

April 20, 2020

Questions for Interviewer C interviewing Person A (for 3-person group).

1. Describe a project in which you used randomness to improve its efficiency.

2. You are given two arrays, A1 and A2, that contain integers. Describe an algorithm to compute the pair of values (with one value from each array) that has the smallest non-negative difference.

3. Design an algorithm to shuffle a deck of 52 cards. It must be a perfect shuffle – all 52! permutations of the deck need to be equally likely. Assume you have access to a perfect random number generator.

4. Design an algorithm to count the total number of 2s that appear in all numbers between 0 and N, inclusive. For example, the number 22 has two 2s. The number 12 has one 2.

5. Suppose you are given an array of N distinct integers. Design an algorithm to find triplets (subsets of 3 elements) such that the sum is zero. The straightforward approach will take time O(N^3). Design an algorithm that is faster than N^3.

CS324: Technical Interview Practice

April 20 and 22, 2020

Announcements:

- This Friday is our last lecture; bring review questions to Teams on Friday
- Final exam for Section A (MWF 9:15) is Tuesday, April 28 from 1:30 to 3:30pm
- Final exam for Section B (MWF 12:30) is Monday, April 27 from 1:30 to 3:30pm

---

Today's learning outcomes:

- Practice communication skills through interview practice
- Explain your thinking
- Use skills from this course to solve problems

Activity: You will be divided into pairs (or a group of 3) and each take a turn of being the interviewer and interviewee. You will not be graded on your answers to the questions, but you will earn 15 homework points for completing this activity and submitting the information below.

What to submit:

- Microsoft Teams video inside your private channel (be sure to click on the "Record" meeting option when you start your video call). Include a link to the video in your channel – it should automatically save the video recording there, but please double check that the link is there.
- Complete the Moodle "quiz" reflection questions before midnight.

Find out whose birthday is coming up the soonest. That person is Person A. Order yourselves as A, B (and C, if in a 3-person group).

Person A: ____________________

Person B: ____________________

Person C (if group of 3): ____________________

Two-person groups: Each interview should last 20 – 25 minutes, so do not ask a new question after the 20-minute mark.

Three-person groups: Each interview should last 15 – 20 minutes, so do not ask a new question after the 15-minute mark.

As the interviewer (or observer): Make notes about what you think the person did well in terms of presenting their ideas. As the interviewee, pretend this is a formal video technical interview and offer your thoughts and ideas as you would in an interview. You can assist by asking the interviewee to explain their thinking as they are solving the problem.

As the interviewee: Try not to peek at the questions you will be asked, so it is a more authentic experience. Tammy already gave you some practice questions for coding challenges this semester (sorting, 0-1 knapsack, set partition). The interview questions for this activity are more in the “explain your thinking” category instead of coding up a solution. Where appropriate though, you may use your whiteboard option (go to Share->Whiteboard during your MS Teams meeting) to diagram and show calculations.

At the end: Thank your groupmates and answer the reflection questions on the Moodle quiz.

Some of the interview questions come from the book, *Cracking the Coding Interview*. Other good interview resources include HackerRank.com, leetcode.com, and geeksforgeeks.com.

CS324: Technical Interview Practice

April 22, 2020

Questions for Interviewer A interviewing Person B:

1. Describe a time when you used hashing to improve the efficiency of finding items in a coding project.

2. Describe how you have developed a recursive solution to a problem.

3. Given an increasing-order sorted array with unique integers, design an algorithm to create a binary search tree with minimal height.

4. There are three ants located at the three points of a triangle. What is the probability of collision (between 2 or 3 ants) if they start walking on the sides of the triangle? Assume the ants randomly choose directions with equal probability and they walk at the same speed.

5. Design an algorithm to find the smallest K numbers in an unsorted array of size N. What is the runtime?

CS324: Technical Interview Practice

April 22, 2020

Questions for Interviewer B interviewing Person A (or C for 3-person group):

1. Describe a time when you used randomization to improve the efficiency of a coding project or problem.

2. Describe how you have taken a recursive solution and turned it into an iterative solution.

3. Describe an algorithm to determine if a binary tree is balanced. A balanced tree is a tree where the two subtrees of any node never differ by more than one.

4. You have a 5-quart jug, a 3-quart jug, and an unlimited supply of water (but no measuring cups). How would you get exactly 4-quarts of water? They are milk jugs, so filling to "halfway" up or 80% up will not work.

5. You are given an array of size N with all the numbers 1 through N+1 appearing exactly once, except for one of those numbers is missing. Design an algorithm to find the missing number in O(N) time and O(1) space.

CS324: Technical Interview Practice

April 22, 2020

Questions for Interviewer C interviewing Person A (for 3-person group):

1. Describe a time when you used recursion over iteration in a coding project. Why did you make that choice?

2. Describe a problem in which using a tree is a natural data structure.

3. Design an algorithm to find the next in-order successor of a node in a red-black tree. You may assume each node has links to its children and a link to its parent.

4. Design an algorithm to determine if a string has all unique characters and you are not allowed to use additional data structures other than the array in which the string is stored.

5. You have access to an already built binary search tree. Design an algorithm to return a random key from the binary search tree.

CS324: Reflection on the course

April 24, 2020

Announcements:

- Final exam for Section A (MWF 9:15) is Tuesday, April 28 from 1:30 to 3:30pm
- Final exam for Section B (MWF 12:30) is Monday, April 27 from 1:30 to 3:30pm
- The final exam will be available on Moodle and your solutions submitted to Moodle, just like exam 4.

Today's learning outcomes:

- Be proud of what you have learned
- Summarize the learning outcomes

I usually spend part of the last lecture of a course reviewing the student learning outcomes of the course. Here are the student learning outcomes from the syllabus:

At the end of the course, students should be able to:

- Analyze time-complexity of algorithms
- Analyze design paradigms and trade-offs of algorithms
- Execute (sometimes clever) algorithms and determine their efficiency
- Prove correctness of algorithms
- Prove that a problem is NP-complete (determine if a problem is intractable)

Take a few moments and reflect upon these skills that you have developed in the course.

1. List an algorithm for which we proved the time-complexity. Insertion Sort

2. List at least three algorithm design paradigms and list an algorithm for each paradigm. One example of an algorithm paradigm is divide-and-conquer. Another is probabilistic algorithms.

   UPLOAD TO MOODLE

3. Name one "clever" algorithm that we studied this semester and why do you think it is clever?

   UPLOAD TO MOODLE

4. How do we prove correctness of iterative algorithms (those with loops)? loop invariants

5. How do we prove correctness of recursive algorithms? strong induction

6. Name at least three NP-complete problems that we studied. Circuit-SAT (nand)
Graph 3-color
VC
Clique

7. What two things do you need to prove to show a problem is NP-complete?

a. in NP (poly time verifier)

b. NP-hard NPC $\xrightarrow{P}$ new one

8. Suppose there is a poly-time transformation from Circuit-SAT to Problem A. What can you deduce about Problem A?

Circuit-SAT $\rightarrow_P$ A

NP-hard

Take-aways

I hope you are proud of what you have accomplished in this course. Even though you may not remember the exact details of all the algorithms in the years ahead, what I hope you take away is a systematic way of thinking about classifying algorithms, a systematic way to determine the runtime of algorithms, and a systematic way of thinking/proving that an algorithm (aka, your code) is correct. You should now have many skills in your "algorithms" toolbox to use throughout your professional career.

9. What are your take-aways from the course?

UPLOAD TO MOODLE

10. What skills are you most proud of developing in this course?

UPLOAD TO MOODLE

What We Covered

First, I hope you feel that this course was valuable. We covered many techniques and algorithms and integrated much from the prerequisite courses.

$$\begin{tikzcd}
\text{MTH311} \arrow[drr] & & \\
\text{EGR361} \arrow[rr] & & 324 \\
\text{CS305} \arrow[urr] & &
\end{tikzcd}$$

What We Did Not Cover

We did not cover the entire textbook, but we cam^{e} close.

We did not cover Section 5 – Advanced Data Structures (special heaps and trees). I think you can learn that material if and when you need it given the skills you have acquired this semester.

We did not cover all of graph algorithms. One extension of single-source shortest-path algorithms is all-pairs shortest-path algorithms where all source-destination pair path costs are calculated. Check out the Floyd-Warshall algorithm if you are interested. Another type of graph problem that comes up in many contexts are called flow networks. If we had more time, this would be the next topic I would put into the course. You can think of flow networks as pipes and junctions, so goods/water can flow between nodes with certain capacities.

We did not cover numerical methods such as linear programming (optimizing linear functions given constraints). If you are interested, check out the simplex algorithm. In this same topic realm, we did not cover Newton's method for root-finding (finding roots of functions) or the FFT for multiplying polynomials in Θ(NlgN) time.

We did not cover specific string matching algorithms (pattern matching, shifting). This topic gets much emphasis in CS 357 with finite automata, and many of you have already taken that course.

We did not cover computational geometry, which is important if you go on to work with computer graphics and computational physics (video games). These types of problems also come up frequently in the ACM programming contest.

Whew – we covered everything else in the textbook and this is the seminal textbook for this course. Again, be proud of how far you have come.

Final Exam Procedure

The final exam will be similar in format to other exams in the course. The final exam will be posted to Moodle for you to take during your final exam time slot. You will upload the completed exam back to Moodle when you are finished.

You may use during the final exam:

- **Four** two-sided crib sheets during the final exam
- Calculator or scientific calculator
- Red-black tree summary sheet posted to Moodle
- Moodle course site, accessing only the exam file link and the exam submission link during the exam session
- A phone or tablet for only taking photos of work on paper to embed in completed exam file
- The exam files

All other aids during the final exam are off limits. You are on your honor to follow the exam rules.

Prior to the final exam, you should upload the four crib sheets that you plan to use during the test.

You may enter answers directly into the Microsoft Word file, embed images in the Word file, annotate the pdf file with a tablet, or take photos of your papers and stitch them together. A .docx or .pdf file is the preferred file format.

You may ask questions through chats in MS Teams if you have questions during the final exam session.

Good luck on this exam and your other projects and final exams. I suggest you re-write some or all of your crib sheets so that the material becomes "fresh" in your mind again. But, you are allowed to just re-use your four crib sheets from the prior exams if you wish.

My Own Reflections

- Enjoyed this course with you
- My first time teaching it, so thank you for your patience
- ☆ Added more applied work in the course – coding portions of two homework assignments, knapsack in-class lab, and technical interview practice
- I wish we could have ended in person
- Good luck to those of you who are graduating

Course Evaluations

Please complete the on-line course evaluations. See Moodle link. Please reflect on the entire course (in-person and on-line) as you prepare your ratings and comments. Positive remarks would be especially helpful feedback. Being professional in your comments is appreciated. Plus, you will be asked to provide feedback about colleagues and managers throughout your career.