ASYMTOTICS

$O \rightarrow g(n) = O(f(n))$ IFF $g(n) \leq c \cdot f(n)$

$\Omega \rightarrow g(n) = \Omega(f(n))$ IFF $g(n) \geq c \cdot f(n)$

$\Theta \rightarrow g(n) = \Theta(f(n))$ IFF $g(n) = O(f(n)) = \Omega(f(n))$

$\log(xy) = \log x + \log y$

$\log(\log n) = O(\log n)$

$\log(1) = \text{zero}$

$\binom{n}{6000} \rightarrow \frac{n!}{6000!(n-6000)!} = \frac{n(n-1)\cdot\ldots\cdot(n-6000)}{6000!} = \Theta(n^{6000})$

$a\log_b c = \log_b c^a$

$\log_b b^n = b^{\log_b n} = n$

$(a^b)^c = a^{bc} = (a^c)^b$

$2^{n+1} = \Theta(2^n)$ but $2^{2^{n+1}} = \Omega(2^{2^n})$ b/c $2^{2n+1} = (2^{2^n})^2$

$(\log n)^a = O(n^b)$ $\forall$ positive a & b

$(\log n)^{\log n} = \Omega(n)$

$\Theta(1) \rightarrow$ constant

$\Theta(\log n) \rightarrow$ logarithmic

$\Theta(n) \rightarrow$ linear

$\Theta(n^2) \rightarrow$ quadratic

$\Theta(n^c) \rightarrow$ polynomial

$\Theta(c^n) \rightarrow$ exponential

Stirling's Approximation: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \Theta(\sqrt{n}(\frac{n}{e})^n)$

$\log(n!) = \Theta(n \log n)$ $(\log n)^{\log \log n} = O(\log(\log n)^{\log n})$

$(\omega n)! = \Omega(n!)$ but $\log((\omega n)!) = \Theta(\log(n!))$

* Sometimes it's helpful to take log of functions but if logs are equal it doesn't mean the functions are *

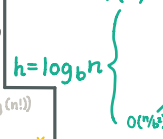
$n^{\log n}$ vs. $(\log n)^n \rightarrow (\log n)^2 = O(n \log \log n)$

Data Structure	len	get/set (@ index/by key)	append/pop (last index)	insert/pop (@ index)	in/index/remove (by item)	set/del (by key)
List	O(1)	O(1)	amortized O(1)	O(n)	O(n)	
Dictionary	O(1)	expected O(1)			↓ by key: expected O(1)	amortized & expected O(1)

Algorithm
1. Description
2. Proof of Correctness
3. Runtime Analysis

$T(n) = aT(n/b) + O(f(n))$

$h = \log_b n$



$T(n) = T(n-1) + O(1) \rightarrow T(n) = O(n) \rightarrow$ length n chain, O(1) work per node

$T(n) = T(n-1) + O(n) \rightarrow T(n) = O(n^2) \rightarrow$ length n chain, O(k) work per node @ height k

$T(n) = 2T(n-1) + O(1) \rightarrow T(n) = O(2^n) \rightarrow$ height n binary tree, O(1) work per node

$T(n) = T(2n/3) + O(1) \rightarrow T(n) = O(\log n) \rightarrow$ height $\log_{3/2} n$ chain, O(1) work per node

$T(n) = 2T(n/2) + O(1) \rightarrow T(n) = O(n) \rightarrow$ height $\log_2 n$ binary tree, O(1) work per node

$T(n) = T(n/2) + O(n) \rightarrow T(n) = O(n) \rightarrow$ length $\log_2 n$ chain, $O(2^k)$ work per node @ height k

$T(n) = 2T(n/2) + O(n) \rightarrow T(n) = O(n \log n) \rightarrow$ height $\log_2 n$ binary tree, $O(2^k)$ work per node

$T(n) = 4T(n/2) + O(n) \rightarrow T(n) = O(n^2) \rightarrow$ height $\log_2 n$ degree 4 tree, $O(2^k)$ work per node

Divide & Conquer

- find something in a set of data
- runtime: O(log n)
- ex: binary search

Peak Finding

1D → given list of n numbers find a peak that is ≥ its neighbors
- Algorithm: look @ middle element & check if peak otherwise recurse on half with larger neighbor
- Recurrence: $T(n) = T(n/2) + O(1) \rightarrow$ runtime: O(log n)

2D → given n×n matrix of numbers, find peak
- Algorithm: Take max element in window frame (Figure #1) defined as columns $0, n/2$, and $n-1$, & rows $0, n/2$, and $n-1$. Check if this max is a peak. Otherwise, the max must have a neighboring element that is greater. Recurse into quadrant with this neighbor and repeat until peak
- Recurrence: $T(n) = T(n/4) + O(n) \rightarrow$ runtime: O(n)

FIGURE #1

Partial Sorting via heaps → O(n + x log n) (x < n)

- When there's n total elements and you want the x biggest
  - build_max_heap → θ(n)
  - extract_max called x times → θ(x log n)
  - in place because original array turned into heap

Delete in heap at a given index

- swap element at i with last value in heap
- truncate array by 1 element
- call max_heapify down on recently swapped element

Binary Search Recurrence: $T(n) = T(n/2) + O(1)$

Insertion Sort Recurrence: $T(n) = T(n-1) + O(n)$

Merge Sort Recurrence: $T(n) = kT(n/k) + \Theta(n)$

Query (x, q): (for a ≤ x ≤ b)

```
if x is None: return None
if q contained in (leftendpoint(x), rightendpoint(x)): return x
if q < leftendpoint(x): return query(left(x), q)
else:
    if left(x) and M(left(x)) >= q: return query(left(x), q)
    else: return query(right(x), q)
```

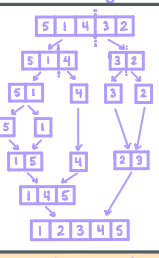
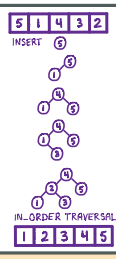
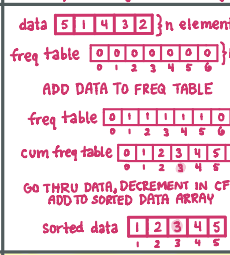
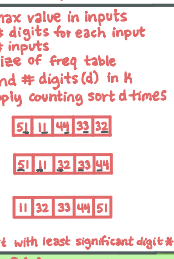
MASTER THM

Master Theorem $T(n) = aT(n/b) + f(n)$

- case #1:
  - $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$
  - $T(n) = \Theta(n^{\log_b a})$
- case #2:
  - $f(n) = O(n^{\log_b a} \log^k n)$ for $k \geq 0$
  - $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- case #3:
  - $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ and $af(n/b) < cf(n)$ for $0 < c < 1$
  - $T(n) = \Theta(f(n))$

Weak Master Theorem $T(n) = aT(n/b) + \Theta(n^c)$

- case #1: leaf heavy recurrence tree
  - $c < \log_b a$
  - $T(n) = \Theta(n^{\log_b a})$
- case #2: balanced recurrence tree
  - $c = \log_b a$
  - $T(n) = \Theta(n^c \log n)$
- case #3: root heavy recurrence tree
  - $c > \log_b a$
  - $T(n) = \Theta(n^c)$

Sort	Insertion Sort*	Selection Sort*	Mergesort*	Heapsort*	AVL Sort*	Counting Sort	Radix Sort
Description	maintain & grow subset of first i elements	maintain & grow i elements; swap min with i^{th} element	recursively sort left and right halves individually and then merge them	BUILD_MAX_HEAP(A) EXTRACT_MAX(A) called n times	insert each item into tree; perform in order traversal	elements must be $0 \leq x < k$ insert n elements into freq table with k bins, then scan thru freq table to get sorted array	better than counting sort if $n < k$
Example							
Runtime	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n+k)$	$O(n)$ if $k = O(n^c)$; $O(d(n+b))$ otherwise
In place? at most O(1) extra space	✓	✓	✗	✓	✗	✗	✗
Stable [3,2,1,2] → [1,2,2,3]	✓	✗	✓	✗	✗	✓	✓
Comments	O(nk) if k far from sorted	O(n) swaps	stable optimal comparison	low space optimal comparison	good if also need dynamic	k is size of domain of possible values	b^d is size of domain of possible values

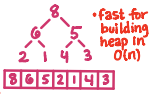
* start with least significant digit *

Data Structure	Sorted Array	Array	Dynamic Array	Binary Heap	AVL Tree	Hash Table	BST
Insert	O(n)	O(1)	Amortized O(1)	O(log n)	O(log n)	Amortized & Expected O(1)	O(h)
Delete	O(n)	O(n)	O(n)	O(n)	O(log n)	Amortized & Expected O(1)	O(h)
Min	O(1)	O(n)	O(n)	O(1)	O(1) (augment)		O(h)
Find	O(log n)	O(n)	O(n)	O(n)	O(log n)	Expected O(1)	O(h)
Successor	O(log n)	O(n)	O(n)	O(n)	O(log n)		O(h)

* = comparison sort

Max-Heap

every node's key is ≥ its children's



- fast for building heap in O(n)
- must be balanced & left justified to be a heap

8 6 5 2 1 4 3

- Max_Heapify_Up(A,i) → O(log i)
  - bubble up i^{th} element until it is in correct spot
  - assume all other elements in correct spot
  - parent(i) = ⌊i/2⌋
  - left(i) = 2i
  - right(i) = 2i+1
- Max_Heapify_Down(A,i) → O(log n − log i)
  - bubble down i^{th} element until it is in correct spot
  - assume both subtrees are heaps
  - always swap with larger of two children
- Insert(x) → O(log n)
  - append x to end of array
  - call Max_Heapify_Up(A, index of x)
- Extract_Max(A) → O(log n)
  - swap first & last elements
  - pop last element out of array
  - call Max_Heapify_Down(A,i)
  - return popped element
- Build_Max_Heap(A,i) → O(n)
  - Build from leaves to root
  - Build_Max_Heap(A, left(i)) if exists
  - Build_Max_Heap(A, right(i)) if exists
  - Max_Heapify_Down(A,i)
- HeapSort(A) → O(n log n)
  - Build_Max_Heap(A,i)
  - Extract_Max(A) called n times

good for augmentation ordering keys; range queries; find closest x keys to this key

* takes O(n log n) to build

BST → every node contains k ≥ keys in left subtree & ≤ keys in right subtree

- Minimum(T) → O(h)
  - go left until no left child
- Find(T,k) → O(h)
  - check if k = key of current node
  - if k < current key, go left, else right
  - if k not in T return None
- Successor → O(h)
  - returns node w/ next largest key
- Predecessor → O(h)
  - returns node w/ next smallest key
- Insert(T,k) → O(h)
  - run find until empty spot, then append
- Delete(T,k) → O(h)
  - run find, if node is leaf, remove
  - if 1 child, link child to node's parent
  - if 2 children swap keys of node & its successor, then remove node
- In_Order_Traversal → O(n)
  - list of all nodes ordered by key

complete = all levels filled except bottom

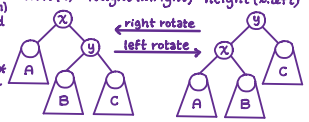
AVL → BST where every node has |skew| ≤ 1

→ balanced: h = O(log n), can be augmented

→ Maintain(x) balances tree w/ rotations

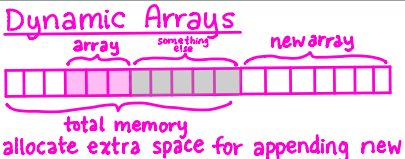
height(x) = max(height(x.left), height(x.right)) + 1

skew(x) = height(x.right) − height(x.left)



* rotations always occur @ lowest unbalanced node

Dynamic Arrays



- allocate extra space for appending new elements
- n = # elements
- m = # slots
- initialize m = 2n
- when run out of space, double size of array
- if $n \leq \frac{1}{4}m$, half size of array

Direct Access Array

- holds a unique slot for each possible k
- n = # elements in data structure
- N = # possible keys

Rolling Hashes → $h(k) = \sum_{i=0}^{r-1} (k[i] \cdot \sigma^i) \bmod m$

1 3 4 5 2 8 7 3

r

- r = size of sequence
- σ = # possible values k[i] takes on

- Given str s & t is s in t?
- Hash s
- For each substring of t: t[i:i+|s|] compare hashes
- if hashes match, check if s == t[i:i+|s|]
- b/c we assume Pr(collision) < 1/m & |s|<m, runtime is:

$O(|s| + |t|\,(1 + \frac{|s|}{m})) =$ expected $O(|s|+|t|)$

hash query string; hash O(t) substrings

Hashing

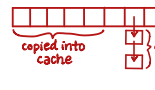
- hash function h: [0, N-1] → [0, m-1] slots in hash table
- store key K @ h(k) in array
- assumes m << N → multiple keys map to same hash value
- collision: when two keys a, b have the same hash value (h(a) = h(b))
  - resolved by chaining (linked list)
    - Find whether or not something exists or if seen before

if m = O(n) expected length of the chain is constant

Universal Hash Family

→ allows us to assume Pr(collision) ≤ 1/m

Open Addressing (cache)



Open Addressing → store collisions in different hash locations

- n (filled spaces) ≤ m (# hash spots)

Probe Sequence → $h(k,i) = (h'(k) + i) \bmod m$ (linear probing)

- check h(k,0), then h(k,1), etc.

Insert → keep checking until empty slot found

Delete → indicate variable has recently been deleted

Property checking → BST → perform traversal of tree (O(n)), then read one by one & verify $a_{i+1} > a_i$. AVL → let AVL_height denote len root to leaf path → 1. Find height(T_L) 2. Find height(T_R) 3. if |AVL_height(T_L) − AVL_height(T_R)| > 1: ↳ return NO 4. Return max{AVL_height(T_L), AVL_height(T_R)}+1

Set 0: $(\log n)^{6006} = O(n)$

4 $f_1 = n^{6006}$
2 $f_2 = 6006n$
3 $f_3 = n\log(n^{6006})$
5 $f_4 = 6006^n$
1 $f_5 = (\log n)^{6006}$

Set 1: $3^{2^n} = 2^{(\log_2 3)2^n} = O(2^{2^{n+1}})$

2 $f_1 = 8^n$
3 $f_2 = 2^{n^3}$
5 $f_3 = 2^{2^{n+1}}$
1 $f_4 = 2^{2n}$
4 $f_5 = 3^{2^n}$

Set 2: $\log((\log n)^4) = O((\log\log n)^4)$

1 $f_1 = \log((\log n)^4)$
4 $f_2 = (\log n)^{\log(n^4)}$
5 $f_3 = n^{4\log n}$
3 $f_4 = \log(4^{n^4})$
2 $f_5 = (\log\log n)^4$

Set 3: $\binom{n}{n/5} = O((5/4^{4/5})^n/\sqrt{n})$

5 $f_1 = 5(n!)$
1 or 2 $f_2 = \binom{n}{5}$
1 or 2 $f_3 = n^5$
3 $f_4 = \binom{n}{n/5}$
4 $f_5 = 5^n$

a)	b)	c)	d)
$f_1 = 2018n$	$f_1 = (\log\log n)^5$	$f_1 = 9^n$	$f_1 = 4^n$
$f_2 = n\log(n^{2018})$	$f_2 = \log(5^{n^5})$	$f_2 = 3^{3n}$	$f_2 = 4(n!)$
$f_3 = 2018^n$	$f_3 = (\log n)^{\log(n^5)}$	$f_3 = 2^{3^n}$	$f_3 = n^4$
$f_4 = (\log n)^{2018}$	$f_4 = \log((\log n)^5)$	$f_4 = 3^{3^{n+1}}$	$f_4 = \binom{n}{4}$
$f_5 = n^{2018}$	$f_5 = n^{5\log n}$	$f_5 = 3^{n^2}$	$f_5 = \binom{n}{n/4}$

Asymtotics

a. (4, 1, 2, 5, 3) This order follows from knowing that $(\log n)^a$ grows slower than n^b for all $0 < a$ and $0 < b$, and n^a grows slower than n^b for all $0 < a < b$.
b. (4, 1, 2, 3, 5) This order follows from elementary exponentiation and logarithm rules.
c. (1, 2, 5, 3, 4) This order follows after converting all the exponent bases to 3, specifically for $f_1 = 3^{2n}$ and $f_3 = 3^{(\log_3 2)3^n}$ (constant factors matter inside the exponent).
d. ({3, 4}, 5, 1, 2) This order follows from the definition of the binomial coefficient and Stirling's approximation. The tricky one is f_5 which is $O(((4/3)^{3/4})^n/\sqrt{n})$ by repeated application of Stirling and algebra. A weaker bound could also be used with Stirling applied once: $\binom{n}{k} \le n^k/k! < (ne/k)^k$ yielding $O((\sqrt[4]{4e})^n)$.

(a) [10 points] **Invariant:** Given an $n \times m$ corner-odd array that is strictly larger than 2×2, show there always exists a strict subset of the array that is also corner-odd. Being strictly larger than 2×2 implies some dimension of the array is greater than 2. Without loss of generality, assume width is the larger dimension. Consider the top and bottom elements x and y of a middle column of the array. Their sum is either even or odd. Because the array is corner-odd, the sum of either the left two corners or the right two corners of the array is odd, with the sum of the other two being even. Thus, x and y together with either the two left or two right corners form a subarray that is also corner-odd **(b) Minimal:** Show that every corner-odd array contains a 2×2 corner-odd subarray. Proof by contradition. Suppose there exists a corner-odd array that contains no 2×2 corner-odd subarray, and let A be a smallest such array (with the fewest Since A is corner-odd it cannot itself be 2×2, so it must have a 2×2 array as a strict subset. By part (a), there exists a strictly smaller subset of A that is also corner-odd, contradicting the minimality of A.
(c) Algorithm: Describe an efficient, recursive algorithm that returns a smallest corner-odd subarray from a given an input corner-odd array. Divide the input corner-odd array in half along its larger dimension and identify the subarray half which is also corner-odd, which exists by (a). Repeat on the corner-odd subarray until the subarray is 2×2, and return it
(d) Analysis: Write and solve a recurrence relation for your algorithm. Your runtime should be asymptotically faster than $O(nm)$ Let $N = nm$. Then the recurrence relation is $T(N) = T(N/2) + O(1)$. The solution to this recurence is $T(N) = O(\log N) = O(\log(nm))$

Consider an algorithm UNBALANCEDSORT, identical to MERGESORT except that, instead of dividing an array of size n into two arrays of size $n/2$ to recursively sort, we divide into two arrays with sizes roughly $n/3$ and $2n/3$. For simplicity, assume that n is always divisible by divisors
(a) Recurrence: Write down a recurrence relation for UNBALANCEDSORT. Assume that merging takes $\Theta(n)$ time. $T(n) = T(n/3) + T(2n/3) + \Theta(n)$
(b) Analysis: Show that the solution to your recurrence relation is $\Theta(n\log n)$ The root does no more than cn work for some positive constant c. Then the second level does no more than $c(n/3)+c(2n/3) = cn(1/3+2/3) = cn$ work, the third level no more than $c(n/9)+c(2n/9)+c(2n/9)+c(4n/9) = cn(1/3+2/3)^2 = cn$ and so on. In fact, each level requires at most cn work, though may do significantly less work near the bottom of the unbalanced tree. The longest root to leaf path has height $\log_{3/2} n = O(\log n)$, leading to a $O(n\log n)$ upper bound. The work is also lower bounded by $\Omega(n\log n)$, because the subtree above and including level $h = \log_{3/1} n$ is a complete binary tree with height h, for which each level does at least $c'n$ for some positive constant c'. **(c) Generalization:** Suppose that UNBALANCEDSORT is instead divided into two arrays of size $n/4$ and $3n/4$. How does the asymptotic runtime of the algorithm change? What about div Assume that merging still takes $\Theta(n)$ time, and int $1 < a$? If we divide into $n/4$ and $3n/4$, each level i requires at most $cn(1/4+3/4)^i = O(n)$ work, while the height of the tree is $\log_{4/3} n$ which is $O(\log n)$. In fact for division into n/a and $(1-a)n/a$, each level requires at most $cn(1/a+(a-1)/a)^i = O(n)$ work, while the height of the tree is $\log_{a/(a-1)} n$ which is $O(\log n)$ for any constant $a > 0$. So the asymptotic runtime is the same for both cases.
(d) Limitation: Now suppose that UNBALANCEDSORT divided into two arrays of size a and $n-a$ for some positive constant integer a. How does the asymptotic runtime of the algorithm change? the amount of work done at the root (level 0) is cn, while the work done at the next level is $ca + c(n-a) = cn$. Then the work done at level $i+1$ is $c(n-ia)$. Then the total runtime is:

$$cn+\sum_{i=0}^{n/a} c(n-ia) = cn+cn\left(\frac{n}{a}+1\right)-ca\frac{(n/a)(n/a+1)}{2} = \frac{c}{a}\left(n^2-\frac{n^2}{2a}+2an-\frac{a^2}{2}\right) = \Theta(n^2)$$

(a) For each array, state whether it is a max heap, a min heap, or neither.
$a_0 = [7,6,4,1,5,2,3]$ Max heap $a_1 = [2,4,6,5,1,7,3]$ Neither $a_2 = [6,7,3,1,5,2,4]$ Neither
$a_3 = [1,2,5,3,4,7,6]$ Min heap $a_4 = [7,3,6,1,5,2,4]$ Neither
(b) Draw the complete binary tree representation for array a_1 turn it into a min heap

```
    Fix A[6]                 Fix A[4]                Fix A[1]
  ______2______            ______2______           ____(2)____            ____1____
__4__      _(6)_  =>   _(4)_       __3__  =>  _(1)_       __3__  =>  __2__      __3__
5    1    7   (3)      5  (1)     7    6      5    4      7    6     5    4     7    6
```

(c) What are the minimum and maximum number of elements that can be contained in a heap with height h? a heap has the minimum number of elements when only a single element is at the bottom level. There are 2^h elements on the bottom level of a complete binary tree, so the minimum number of elements is $(2^{h+1}-1) - 2^h + 1 = 2^h$.

FredEx

(a) Throughout the day he receives shipments to his warehouse, each shipment contained in identical regulation-sized packaging. As packages arrive, scanners read each package's ID, address, and the value of its contents. Each day, the trucking company sends two trucks. One is a regular truck The other truck is shared with other companies, so Fred only knows how many packages it will be able to carry when it arrives. store his inventory information in a AVL tree Inserting and deleting from the inventory directly carries over from analogous operations To return a list of the most valuable k packages, simply find the package with maximum value in $O(\log n)$ time, and then repeatedly walk backward The path together with all the nodes to its right constitutes a binary tree $O(k+\log n)$ nodes which can be traversed in $O(k+\log n)$ time.

servers

(b) fixed number of servers, each with a fixed amount of bandwidth. Many times per second, some customer makes a request for a fixed amount of bandwidth for a fixed amount of time AVL sorted by current available bandwidth. the server can remove the record from the tree in $O(\log n)$, update the record in $O(1)$, and then reinsert into the tree in $O(\log n)$. To find k servers to handle a request needing bk bandwidth, it suffices to find out if the k servers with the highest available bandwidth can each accept an increase by b. To return the list of the k servers with the highest available bandwidth, find the server with the most available bandwidth in $O(\log n)$ time, and repeatedly walk backward in the tree in a reverse-order traversal through the $k-1$ previous record

(a) Count the number of repeated numbers in an array of n integers. For each number, search a hash table for that number. If the search is successful, add one to a count of repeats. If the search is unsuccessful, insert the number into the hash table. This algorithm runs in expected $O(n)$ time because of $O(1)$ expected hash table find and insertion.
(b) Sort a set of n integers, where each integer x in the set is guaranteed to satisfy $6005n^3 < x < 6006n^3$. Transform the set of integers by subtracting $6005n^3$ from each, yielding integers between 0 and n^3. Then we can view each as a three digit base-n number, and apply radix sort to sort in worst case $O(n)$ time. **(c)** Given a set of n integers, determine whether any pair of them sum to a given value k. Insert each integer into a hash table. Then search in the hash table for the difference between k and each integer. If the difference is found, then you will have found two integers that sum to k. This algorithm runs in expected $O(n)$ time because of $O(1)$ expected hash table find and insertion. **(d)** Let $\mathcal{W}$ be the set of English words. Given a long encoded message from your friend containing $n = \Omega(|\mathcal{W}|)$ words decode the message. Use a hash table to store all words in $\mathcal{W}$ along with their ciphered versions, keyed by the ciphered version. Then for each word in the encrypted message, search in the hash table for the word and replace with the associated word. This algorithm runs in expected $O(n)$ time because of $O(1)$ expected hash table find and insertion.

(10, 13, 9, 10, 13, 5, 2, 13, 5, 10, 9, 13, 10, 9, 13, 2, 5, 13, 2, 67, 23, 1, 2, 10, 1, 2, 1, 10, 2, 1)
Write down the list of satisfying triples contained in the sequence, as well as their associated characters, ordered by initial occurrence. What is the secret message?
The list of satisfying triples is below, yielding the secret message 'fun'.

Triple	Remainder	Letter	Initial Occurrence
(13,10,9)	5	f	13
(5,13,2)	20	u	18
(10,2,1)	13	n	29

Bob has started to send longer messages, and Alice is tired of decoding his messages by hand. Design an expected $O(n)$ time algorithm to decode a sequence of n integers from Bob, and output his secret message
Solution: Scan each consecutive triple from Bob's sequence, and use a hash table to keep track of the triples seen so far. When processing a triple of integers (a, b, c), check whether a, b and c are unique. If they are not unique, then (a, b, c) cannot be a satisfying triple. It they are unique, check whether (a, b, c) is stored in the hash table. If so, we've seen this triple before, so this triple will not complete an initial occurrence of a satisfying triple. Otherwise, if (a, b, c) is not in the hash table, add the newly seen ordered triple to the hash table. Then check whether the five permutations of the triple, (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a), all exist in the hash table. If all five permutations exist in the hash table, this means that we have previously encountered all the other permutations, followed by the first occurrence of (a, b, c). This is the definition of a satisfying triple, so append the triple's decoded character to an output string. Each hash table insert and find operation takes expected $O(1)$ time; each triple requires at most six finds and one insertion, so can be processed in expected $O(1)$ time. Thus, to process all $O(n)$ consecutive triples of Bob's sequence can be done in expected $O(n)$ time. Write the Python function decode_message(sequence)

* first question *

```
def decode_message(sequence):
    triples = set()
    out = ''
    for i in range(len(sequence) - 2):
        a, b, c = sequence[i:i + 3]
        if a != b and a != c and b != c:
            if (a, b, c) not in triples:
                triples.add((a,b,c))
                if (    (a, c, b) in triples and
                        (b, a, c) in triples and
                        (b, c, a) in triples and
                        (c, a, b) in triples and
                    out += convert_to_char(a + b + c)
    return out
```

Recurrence	Solution
T(n)=T(n−1)+O(1)	T(n)=O(n)
T(n)=T(n−1)+O(n)	T(n)=O(n²)
T(n)=2T(n−1)+O(1)	T(n)=O(2ⁿ)
T(n)=T(2n/3)+O(1)	T(n)=O(log n)
T(n)=2T(n/2)+O(1)	T(n)=O(n)
T(n)=T(n/2)+O(n)	T(n)=O(n)
T(n)=2T(n/2)+O(n)	T(n)=O(n log n)
T(n)=4T(n/2)+O(n)	T(n)=O(n²)
T(n)=3T(n/2)+O(n)	T(n)=O(n^{log₂3})
T(n)=2T(n/3)+O(n)	T(n)=O(n)
T(n)=4T(n/2)+O(n²)	T(n)=O(n² log n)
T(n)=3T(n/3)+O(n)	T(n)=O(n log n)

	α	1/4	1/2	3/4	9/10
Linear Probing	successful	1.2	1.5	3.0	5.5
	unsuccessful	1.4	2.5	8.5	50.5
Double Hashing	successful	1.2	1.4	1.8	2.6
	unsuccessful	1.3	2.0	4.0	10.0

this will be in the cache → O(1) :)
this may be in memory → takes longer :(

α = load factor = # data points / # slots in hashtable = n/m = how full hash table can get before capacity is automatically increased
↳ expected # collisions w/ chaining is O(1+n/m) = O(1+α)

(a) we modify the algorithm to repeatedly divide solve a recurrence relation to upper bound the running time of this unbalanced binary search. Does the modified binary search still run in $O(\log n)$ time? master theorem case 2, since $f(n) = \Theta(n^{\log_{c/(c+1)} 1}) = \Theta(n^0) = \Theta(1)$.

Unbalanced / Search
$$T(n) = \max\left(T\left(\frac{n}{c+1}\right), T\left(\frac{c}{c+1}n\right)\right) + O(1) < T\left(\frac{c}{c+1}n\right) + O(1) = O(\log n)$$

recurrence

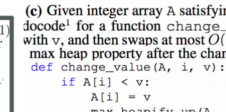
(b) Solving recurrences: Derive solutions to the following recurrences
$T(n) = 3T(\frac{n}{2}) + O(n)$ $T(n) = O(n^{\log_2 3})$ by master theorem case 1 since $f(n) = O(n^{\log_2 3-\varepsilon})$
$T(n) = 2T(\frac{n}{3}) + O(n)$ $T(n) = O(n)$ by master theorem case 3 since $f(n) = \Omega(n^{\log_3 2+\varepsilon})$
$T(n) = 4T(\frac{n}{2}) + \Theta(n^2)$ $T(n) = \Theta(n^2\log n)$ by master theorem case 2, since $f(n) = O(n^{\log_2 4}) = \Theta(n^2)$
$T(n) = 3T(\frac{n}{3}) + \Theta(n)$ $T(n) = \Theta(n\log n)$ by master theorem case 2, since $f(n) = O(n^{\log_3 3}) = \Theta(n)$
$T(n) = T(n-5) + O(\log n)$ (n)—(n−5)—(n−5i)—(1) So total work is $O(n\log n)$,
The 'tree' is simply a chain where the node at height i does $O(\log 5i)$ work.

(a) Prove that every nonempty two-dimensional array contains a peak. Because the integers are **well-ordered**, there exists a maximal integer p among the $n \times n$ integers that is at least as large as all the others. In particular, p is at least as large as its eight neighbors, so p is a peak.
(b) Consider a two-dimensional array A and rectangular sub-array $B \subseteq A$. Prove that if sub-array B contains an element that is greater than or equal to every element of A directly outside and adjacent to B, then B contains a peak. the only way for p to **not** be a peak in A is if it were adjacent to a larger integer outside of B. Since B contains an integer (witness) that is greater than or equal to every element of A directly outside and adjacent to B, then p, being at least as large as the witness, will also be greater than or equal to the elements outside and adjacent to B. Thus p is a peak.
(c) Given an $n \times m$ array with $n \ge m$, divide it by splitting the longer dimension, n, in half. Show how to identify a witness in one of the halves to certify that it contains a peak in $O(m)$ time. Consider the set of $O(m)$ elements comprising the two rows or columns of elements adjacent to the line separating the two halves of the array. A maximal integer p among those elements will again be at least as large as any array element outside and adjacent to the half containing p. Thus p a witness certifying a peak in the half **(d)** Our algorithm will divide the array in half by splitting its longer dimension, identify a witness in one of the halves, and then recursively search within the half containing the witness. Write and solve a recurrence relation based on this algorithm. The input size of this problem is $\Theta(nm)$. A recurrence relation for this algorithm is $T(nm) = T(nm/2) + O(\min(n, m))$. Here, we note that $\min(n, m) = O(\sqrt{nm}) = O(\max(n, m))$, so by letting $N = nm$, we get the recurrence relation $T(N) = T(N/2) + O(\sqrt{N})$. This recurrence relation evaluates to $T(nm) = T(N) = O(\sqrt{N}) = O(\max(n, m))$ by case 3 of the master theorem, since $f(N) = \Omega(N^{\log_2 1}) = \Omega(1)$. When $m = O(n)$, this algorithm is $O(\sqrt{nm})$, which is **sub-linear** in the input $O(nm)$.

inversions

(b) Give an $O(n^2)$ algorithm to count the number of inversions in an array. There are only $\binom{n}{2} = O(n^2)$ pairs (i, j) where i is less than j. Then an $O(n^2)$ brute force solution would loop through each pair and check in constant time whether it is an inversion. **(c)** Given array A whose left and right halves, A[:n/2] and A[n/2:], are each sorted, give a linear time algorithm to count the number of inversions in A. There are no inversions in the first half or second half as they are sorted. Thus the only inversions occur between some element i in the first half and element j in the second half. Scan left pointer i and right pointer j across each half of the array element by element in order from left to right, similar to the merge procedure in merge sort. During the scan, we maintain the invariant that all inversions involving elements $j' < j$ in the second half of the array have been counted. This is exhaustively true at the start when j points to the left-most element of the right half of the array. Now assume that we have counted all inversions involving $j' < j$. If $A[i] < A[j]$, element i does not form an inversion with j or any element to j's right, so we increment i and maintain the invariant. Alternatively, if $A[i] > A[j]$, then element i forms an inversion with i and every element to i's right in the first half of the array. We then add this number to a count and increment j, thus preserving the invariant. Because we increment a pointer over each element of the array exactly once, performing constant work each time, this algorithm runs in linear time.
(d) Give an $O(n\log n)$ algorithm to count the number of inversions in an array. We augment the merge sort algorithm by separating the array into halves, sorting and counting inversions in each half inductively, counting inversions across the halves using the counting algorithm described in the previous part, completing the sort by merging them as in merge sort, and returning the sum of the three counts Correctness is argued in the section below. This algorithm remains $O(n\log n)$ because the counting function and the merge step can both be done in linear time, leading to the same asymptotic running time as merge sort.
(e) Prove that your $O(n\log n)$ algorithm is correct, We prove that the algorithm returns the correct number of inversions **and** sorts the array by inducting on the size of the array. When the array contains only one element which is sorted, it cannot participate in an inversion, and the algorithm correctly returns a count of zero. Now consider an array of length n, assuming that the claim is true for the algorithm run on arrays of length $< n$. Each inversion exists between two elements i and j, which can either both be in the left half, both in the right half, or one is in each. Sorting and counting inversions in each half correctly counts the first two types of inversion by induction. Then the counting algorithm and merging sorts correctly counts the inversions across the halves by part (c), so summing these three correctly counts all three types of inversions, completing the proof.

(b)



The heap's largest key A The heap's second largest key B, C
The heap's third largest key B, C, D, E, F, G
The heap's smallest key D, E, F, G

heap ordering & change_val

(c) Given integer array A satisfying the max heap property, index i, and value v, write pseudocode[1] for a function change_value(A, i, v) which replaces the value at A[i] with v, and then swaps at most $O(\log n)$ pairs of elements to ensure the array maintains the max heap property after the change. The running time in either case is proportional to the $O(\log n)$ height of the tree, since we might swap up or down the entire height.

```
def change_value(A, i, v):
    if A[i] < v:
        A[i] = v
        max_heapify_up(A, len(A), i)
    if A[i] > v:
        A[i] = v
        max_heapify_down(A, len(A), i)
```

(a) Given an array of cow friend locations, describe an $O(F\log F)$ time algorithm to return Bessie's guest list. Use any $O(F\log F)$ sorting algorithm to sort the list of points by their distance to the origin, and then tell Bessie the G smallest elements by reading from the left. The total time is $O(F\log F + G) = O(F\log F)$ **(b)** Describe an **in-place** $O(F + G\log F)$ time algorithm to return Bessie's guest list. Use build_min_heap This takes $O(F)$ time and $O(1)$ extra space. Now, run extract_min G times and tell Bessie the last G elements of A in reverse order.
(c) Describe a $O(F\log G)$ time algorithm to return Bessie's guest list using at most $O(G)$ extra space, provided only read access to the by maintaining a **max** heap of the *smallest* G elements seen so far, as we scan through all F elements, we can find the minimum G elements $O(F\log G)$ time. Once the scan is complete, the heap contents can be sorted (via multiple calls to extract_max) and the now-sorted array can be returned.

Solution: We assume the number of possible characters k to be small compared to the number of Kerberos usernames, $k \ll n$. A Kerberos username will be similar to another when it is different only by a single character. Suppose a similar pair differs at character i. If we sort the usernames character-by-character, but prioritize character i last below all others, similar usernames differing at character i will be adjacent to each other in the sort as all other characters in a similar pair are the same by definition. We can perform such a sort in **worst-case** $O(n+k)$ time, by applying radix sort with a counting sort subroutine on the letters, prioritizing character i last. Then scan through the sorted list comparing adjacent usernames. If two adjacent usernames are identical except at character i, then they are similar. To keep track of whether usernames have been found to be similar, append the character 0 to the end of each username. If a scan finds a pair that are similar, change their appended character to 1. We repeat this process eight times, each time prioritizing character $i \in \{0, \ldots, 7\}$ last. After all eight radix sorts and scans have been complete, loop through the sorted list in linear time and append to an output list any username for which a 1 is appended. We perform a constant number of radix sorts and linear scans, so the entire algorithm takes **worst-case** linear time. QUIZ 1 KERB QUESTION →

Given a max heap, it's possible to find the ith largest element in O(1) time when i is a constant (ie 3rd)

DS Data structures

(a) Insert integer keys (1, 6, 2, 4, 7, 8, 3) in order into a hash table using hash function $h(k) = (11 * k + 5) \bmod 7$. Collisions should be resolved via chaining [4|6| |3| |7|2], 1, 8
(b) Given an array A containing n integers, describe an expected $O(n^2)$ time algorithm to determine whether two pairs of distinct integers from A have the same sum, i.e. there exists $\{a, b, c, d\} \subseteq A$ where $a + b = c + d$. Pre-process to find duplicates in the input by inserting each array element into a hash table, but for each integer value, store a list of array elements having the same value. Each insertion can be performed in expected amortized constant time. If the lists associated with two distinct integer values each contain two or more array elements, we will can construct a satisfying quadruple by choosing two elements from each, so return True. Alternatively, exactly one integer value list contains two or more array elements.– If the list contains four or more array elements, four of them comprise a satisfying quadruple, so return True. – Otherwise, at most three array elements have the same value. Run the initial algorithm as described above. Now, if a pair sum collision occurs, either the collision represents a satisfying quadruple, or if the array elements overlap, it is because they are of the form (a, b) and (a', b) If $a' + b = c + d$ for some other pair (c, d) for which $c \ne a$ and $d \ne a$, then $a + b$ also equals $c + d$, and we can safely ignore pair (a', b). The pre-processing step to find duplicates takes expected $O(n)$ time to discover which case above holds, while expected $O(n^2)$ will be used in the last case, leading to an expected $O(n^2)$ time algorithm. **(c)** Design a **double-ended addressable queue** data structure to store a sequence of items that supports **worst-case** constant time array indexing to the ith item in the stored sequence, as well as dynamic operations: insertion and removal store the queued items in the middle of an array rather than the front, leaving linear number of extra slots at both the beginning and end whenever rebuilding occurs, guaranteeing that linear time rebuilding only occurs once every $\Omega(n)$ operations. A linear number of operations between expensive linear time rebuilds ensures that each dynamic operation takes at most amortized $O(1)$ time. To support array indexing in constant time, we maintain the index location i of the left-most item in the array and the number of items n stored in the array. To access the jth item stored in the queue sequence using zero indexing, confirm that $i + j < n$ and return the item at index $i + j$ of the array container
(d) set B of books where the text from book $b \in B$ contains w_b words Describe an expected $O(p + \sum_{b\in B} w_b)$ time algorithm to determine which book contains the inspiring passage. Compute a rolling hash of characters contained in passage p in $O(p)$ time, since each word contains at most a constant number of characters. For each book apply Rabin-Karp to check if p is a substring of b in expected $O(w_b)$ time, by comparing each length p substring of b from a close rolling hash in amortized expected $O(1)$. Doing this for all books then yields expected running time $O(p + \sum_{b\in B} w_b)$ as desired.

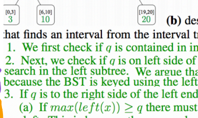
5

(a) Given an HB daily log containing n entries, describe a worst-case $O(n^2)$ algorithm to determine the set of distinct portfolios still associated with active trades at the end of the day Process each line of the log from top to bottom while maintaining a list (dynamic array) storing active portfolio names. For each plus line $(+, C, P)$ scan all $O(n)$ lines below it and check if the line $(-, C)$ appears in $O(1)$ time per line. If $(-, C)$ appears in any subsequent line, the plus line was canceled during the day, so skip it and process the next plus line. If $(-, C)$ does **not** appear in any subsequent we must add it to the active portfolio list. Search through the list of active portfolio names for P in $O(n)$ time; if P does not appear, append P to the end of the dynamic array in amortized $O(1)$ (but worst-case $O(n)$) time. The algorithm processes all $O(n)$ lines, and each line takes at most worst-case $O(n)$ to process, so the algorithm runs in worst-case $O(n^2)$ time. Describe an algorithm that produces a list of distinct portfolios associated with active trades which runs in expected $O(n)$ time. Store a hash table of hash tables The top level hash table maps each counterparty to a hash table maintaining the set of portfolios associated with the counterparty. To process plus line $(+, C, P)$, add P to the hash table associated with C in expected amortized $O(1)$ time per insertion. To process a minus line $(-, C)$, replace the hash table associated with C with an empty hash table, also in expected constant time. The resulting size of the data structure will be worst-case $O(n)$ because each portfolio in a counterparty hash table uniquely corresponds to a line of the log, and we can enforce that each hash table maintain a constant load factor. To return the list of portfolios at the end of the day, construct a new output hash table to store distinct portfolios. Scan all counterparty hash tables in worst-case $O(n)$ time, adding each portfolio found to the output portfolio hash table in expected amortized $O(1)$ time per insertion. Then return all portfolios contained in the output portfolio hash table via a linear time scan this runs in expected $O(n)$

(a) Given a list of n 6.006 students and their semester schedules, describe a worst-case linear time algorithm to generate a list of MIT classes containing more than m 6.006 students. Insert each class of each student into a single array. Sort the array using radix sort with a counting sort subroutine, sorting the array by each of the 8 letter positions identifying a course After the sort concludes, the same classes taken by different students will appear next to each other in the sorted array. Scan the array, keeping a count of the number of students for each class. If the count is larger than m, add the class to an output list Counting sort takes $O(n+36)$ time to sort each character (26 characters and 10 digits), while radix takes $O(8(n+36))$. The final scan through the array looks at each class at most once. also linear time
(b) The mobile app should store a local copy of the current inventory in a database, dynamically displayed to a user as a table, with one product per row and one property per column To display the desired sorted orderings of the products we radix sort, sorting each property using an appropriately applicable stable sorting algorithm, in reverse order of property column order. Assuming the number of products is more than 10,000, using counting sort can sort products by price and category in linear time. By contrast, stock and name likely represent keys that are not bounded if either stock or name contain an unbounded number of characters, a stable optimal $O(n\log n)$ comparison sort like merge sort would be a reasonable choice. **(a)** There is a very large number of flights to choose from. Help your TA sort the flights by price. We can leverage the fact that prices have numerical values with finite numerical precision. Assume that flight prices are between \$0 and \$10,000 with a resolution of cents, so at most 7 base-10 digits. Use radix sort to sort each digit using counting sort in $7 \times O(n)$ time.
(b) At various times through out the day, the manager wants to know statistics about the orders that have been fulfilled or canceled (e.g. mean, median, minimum, maximum). maintain the transactions in sorted order using an AVL sort, augmenting each node with any relevant stat
(c) a couple of assignments (far fewer than the total number of problem sets) are out of place. Insertion sort works very well when the array is mostly sorted, running in $O(ne)$, where e is the number of errors. Thus, re-sort the problem sets using insertion sort.
(d) though it would be difficult to put a numerical value on it. Help choose the best set of talks Since we only need the first couple of elements in the sorted array, we can use heap sort to forgo doing extra work. By building a max heap in $O(n)$ time, removing the k best talks will only take an additional $O(k\log n)$ time, where k is the number of talks your TA can attend

4

Show that in fact, in-order traversal requires at most $O(n)$ time. An in-order traversal walks around the tree touching each edge at most twice. Because a tree has one fewer edge than the number of nodes and the in-order traversal sequence will cross each edge at most twice, the traversal will be linear in the number of nodes
(a) For the interval tree in the figure above, compute the value of $M(x)$ for each node x in the tree Argue that you can augment any interval tree with these values in linear time.

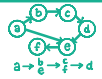


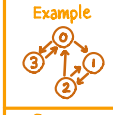
By computing the value of $M(x)$ in level order (from the lowest level to the root) computing the maximum using pre-computed lower values of $M(x)$ is simply a comparison between three values, which can be done in constant time per node, leading to linear time overall. You can think of this approach as computing $M(x)$ via a bottom up dynamic program.
(b) describe and give pseudocode for an algorithm query(x,q) that finds an interval from the interval tree rooted at x that contains q (or None), and runs in $O(h)$
1. We first check if q is contained in interval stored in node x – if it is we are done.
2. Next, we check if q is on left side of the left endpoint of interval in node x - if it is we recursively search in the left subtree. We argue that q cannot be contained in any interval in the right subtree because the BST is keyed using the left endpoint of the interval, and the tree satisfies the BST property
3. If q is to the right side of the left endpoint of the interval in x, we have two cases:
(a) If $max(left(x)) \ge q$ there must be a segment in the left subtree containing q and we recurse left. This is because the max value represents the rightmost point of *some* interval in the left subtree; endpoints of all nodes in the subtree are all less than q thanks to the BST property. Therefore, there is an interval corresponding to some node in the left subtree that contains q.
(b) If $max(left(x)) < q$ there is no segment in the left subtree containing q and we recurse right. All the intervals in the left subtree are such that the left and right endpoints are less than q. We do not know if the right subtree has an interval that contains q, but we know that the left subtree does not.

3

(a) Silliam maintains an inventory of available tour packages. store tour package information in a AVL Tree sorted first by duration, then by price. Inserting and deleting tours can be performed by normal AVL insert and delete using a duration-price tuple as a key in the tree, along with any other relevant tour information. To respond to a query, search for a key having matching duration, but price less than or equal to the customer's bud-get in $O(\log n)$ time. This can be done using normal BST find to a leaf node, which will either contain a package at or just below budget (or just above budget). Once such a leaf node is identified, call a predecessor (analogous to successor) operation four (or five) times to find the relevant cheaper packages for return. If a shorter duration tour is reached before identifying five packages (or if the left-most node is reached), then return however many were found. Since predecessor also runs in $O(\log n)$ time and is run only a constant number of times, the query operation takes $O(\log n)$ time.
(b) Their base is at the center of a large, perfectly-circular island space biologist teams journey to the coast to study and tag coastal aliens. and identify the twenty closest tagged aliens to their location. Have the base store the n locations of aliens in a AVL Tree, sorted by coastline angle relative to, say, North. When an alien is tagged, insert its angular position using normal AVL insert in $O(\log n)$ time. When an alien teleportation log is recorded, find the alien in the tree, and if the alien is there, remove it from the AVL tree, change its position, and re-insert, also in $O(\log n)$ time. To find the k closest aliens to a queried location, find the location in the tree in $O(\log n)$ time, and then repeatedly find the next or previous alien in the order based on which is closer to the queried location, using successor and predecessor. Angles are cyclicly ordered so if the minimum or maximum angle contained in the tree is reached, we modify successor and predecessor to wrap around. Because the successor and predecessor calls are consecutive, all of them together will cost no more than $O(k\log n)$. A tighter upper bound is $O(k+\log n)$, but since $k = 20$ is constant, both are logarithmic, so we will accept either bound.

pairs (x_i, y_i) of observed inputs and outputs. algorithm will accept two inputs: yesterday's temperature x and a confidence interval w. The algorithm will return a prediction for today's temperature y^* by taking the average temperature from all example outputs y_i from the wide set of sample inputs x_i within w degree of x (i.e. $|x - x_i| < w$).
(b) Prove that at most **one** node of a BST can have left and right sub-trees that each contain keys both inside and outside a given range. If you know that every node in a sub-tree is within range, can you think of a way to pre-compute information to store at each node to shortcut computation? Suppose for contradiction there exist two nodes A and B for which both left and right sub-trees contain keys both inside and outside a given range. First, the key a at node A is in range or else by the BST property, no key would be in range in one of the left or right sub-trees, and similarly for the key b at node B. Second, the minimum key a^- and maximum key a^+ in A's sub-tree are both out of range or else no key would be out of range in the left or right sub-tree respectively. If we have A with key a where the nodes underneath A go from a^- to a^+ where a^- and a^+ are to the left and right of the given range $[l, r]$, with $a^- < l$, and $a^+ > r$, then the A subtree needs to contain all nodes in the entire BST that are in the range $[l, r]$. To see this, assume that A is not the root (the root contains all nodes) and that A is the left child of its parent, then the parent node and nodes in the right subtree of the parent will all have keys that are $> a^+ > r$, i.e., outside the $[l, r]$ range. If A is the right child of the parent then the parent node and nodes in the left subtree of the parent will all have keys that are $< a^- < l$. Since b is also in range, B must also be in A's sub-tree. But by the same argument, A must be in B's sub-tree. Both cannot be true, a contradiction, completing the proof.
(c) Describe how Cole can augment his binary search tree to efficiently add examples and calculate predictions, each in logarithmic time At each node, we store additional information about its sub-tree to shortcut computation: sub-tree min and max to make range querying faster, the number of keys stored in the sub-tree, and the sum total of y values stored in the sub-tree. To add an example, insert as in normal AVL, and while maintaining height on ancestor nodes, we can maintain the additional information in the AVL update method by computing min, max, sum, and count from the same sub-tree properties of its children. To make a prediction, run find to find the highest node that is within the queried range. Then for each child, recursively find the count and sum of keys contained in its sub-tree that are in the queried range. By part (a), at most one node makes two recursive calls over the height of the tree, while all other nodes make zero or one. Thus only a logarithmic number of nodes are touched, with constant work done at each, leading to $O(\log n)$ running time.
(x, y) = (predicted temp, today's temp) = (input, output)

	BFS	DFS	TOPOSORT RELAXATION	DIJKSTRA	BELLMAN FORD
path length	shortest	any path	shortest (or longest)	shortest (or longest)	shortest (or longest)
edge weights	equal weights	ignores weights	+ & −	+ only	+ & −
cycles	OK	OK	DAGs only	OK	OK
runtime	O(V+E)	O(V+E)	O(V+E)	O(V log V+E)	O(V·E)
example	 a→b→e→f→d	 a→e→f→b→c→d	 relax edges in topological order start→end	 a b c d e f [0 ∞ ∞ ∞ ∞ ∞] [0 5 ∞ ∞ ∞ 2] [0 5 10 ∞ ∞ 2] [0 5 10 ∞ ∞ 2] [0 5 10 11 ∞ 2] [0 5 10 11 15 2]	 calculated by relaxing edges, getting confirmed SP to a node with SP using 1 edge, then 2 edges, etc.
applications	• find all connected components • testing graph for bipartiteness • Queue	• topological sort order • find connected components (& strongly connected) • find articulation points • (single) cycle detection ↳back edges • stack	• scheduling with prerequisites • shortest/longest DAG path* * AKA DAG-SP *	• travel time from point A to point B	• making a profit by trading currency • identifying a neg weight cycle exists

	Adjacency List	Adjacency Matrix	Adjacency Set
insert edge	O(1) ✓	O(1) ✓	O(1) ✓
delete edge	O(out-degree(v)) ✗	O(1) ✓	O(1) ✓
find edge	O(out-degree(v)) ✗	O(1) ✓	O(1) ✓
iterate through neighbors	O(out-degree(v)) ✓	O(V) ✗	O(out-degree(v)) ✓
Example	 0 [[1,3], 1 [2], 2 [0,1], 3 [0]]	 0 1 2 3 0 [0 1 0 1] 1 [0 0 1 0] 2 [1 0 0 0] 3 [1 0 0 0]	{0:{1,3}, 1:{2}, 2:{0,1}, 3:{0}}
Space	Θ(V+E)	Θ(V²)	Θ(E)

GRAPHS

- graph: G=(V,E) (V: set of vertices, E: set of edges)
- undirected edge: {u,v} ∈ E
- directed edge: (u,v) ∈ E
- degree(v) = # edges adjacent to vertex v
- $\sum_{v \in V} \text{degree}(v) = 2|E|$
- connected: there is a path between every pair of vertices (treat as undirected)
- strongly connected: theres a path between all distinct u & v (only for directed graphs)

BFS

- to compute level set L_{i+1}, include all neighbors of L that aren't already in a level
- can discover SP from s to all other nodes

Pseudocode

```
visited set/array/dictionary = {S: None}
current_level = [s]
next_level = []
while current_level is not empty:
    for node in current_level:
        for neighbor in outgoing(node):
            if neighbor not in visited:
                next_level.append(neighbor)
                visited[neighbor] = node
    current_level = next_level
```

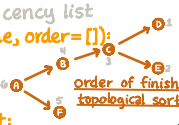
(dict can be used for parent pointers)

DFS

- does not give SP, just finds any path to the given end node

Pseudocode

```
# A is an adjacency list
def dfs(A, start, parent=None, order=[]):
    if parent is None:
        parent = {start: None}
    for neighbor in A[s]:
        if neighbor not in parent:
            parent[neighbor] = start
            dfs(A, neighbor, parent, order)
    order.append(start) # allows us to get topo order
    return parent, order
```



Edge Classification

- tree edge → edge used by DFS to discover a new node
  - ↳ tree edges compose a tree
- back edge → edge from a descendant to an ancestor
- forward edge → edge from an ancestor to a descendant
- cross edge → neither are ancestors of the other

* an undirected graph only has tree edges & back edges *
* a graph has a cycle IFF it has a back edge *
* if there is a cycle then there is no topological sort possible *

RELAXATION

Definitions

- w(u,v) = weight of edge (u,v)
- π = (v₁,...,vₖ) = some path from v₁ to vₖ
- $\delta_s(v)$ = weight of the shortest path from s to v
- d[v] = length of SP from s to v found so far in our algorithm
  - ↳ initially: d[s] = 0, d[v] = ∞ for all other v
  - ↳ by end of algorithm: d[v] = $\delta_s(v)$

Relaxation

```
for an edge (u,v):
    if d[v] > d[u] + w(u,v):
        d[v] = d[u] + w(u,v)
```

→ if there is a shorter path possible we replace our current SP with this new shortest path

Path Relaxation Property

For a shortest path π = (s,...,v) from s to v, if we relax each edge in the path, in path order, d[v] = δ_s[v]

```
General_relax(A, w, s):
    d = [∞ for v in A]
    parent = {s: None}
    d[s] = 0
    while there exists nontrivial relaxations:
        pick some edge (u,v)
        if d[v] > d[u] + w(u,v):
            d[v] = d[u] + w(u,v)      } relax(u,v)
            parent[v] = u
    return d, parent
```

Triangle Inequality: $\delta_s(v) \le \delta_s(u) + w(u,v)$

Optimal Substructure: Consider a shortest path from v_i to v_k: π = (v_i,...,v_k) then all subpaths v_i to v_j are also shortest paths from v_i to v_j

- in a connected graph, V = O(E)
- # tree edges = V−1
- BFS → queue
- DFS → stack
- Dijkstra → priority queue

Path Relaxation Property

1) If there are no negative weight cycles then no SP uses ≥|V| edges
2) If we relax every edge and repeat |V|−1 times all paths of length ≤ |V|−1 will be relaxed in path order

Pseudocode

```
def Bellman_Ford(A, weights, start):
    d = [∞ for _ in A]
    d[s] = 0
    parent = {s: None}
    V = len(A)  # V = # of vertices
    for i in range(V-1):                 # construct SP in ≤ V-1 rounds
        for u in range(V):
            for v in A[u]:
                relax(A, weights, d, parent, u, v)   # relax edge from u to v
    for u in range(V):                   # check for negative weight cycle
        for v in A[u]:
            if d[v] > d[u] + w(u,v):
                raise Exception("neg weight cycle")
    return d, parent
```

BELLMAN-FORD

DIJKSTRA

```
def Dijkstra(A, weights, s):
    d = [float('inf') for _ in A]
    d[s] = 0
    parent = {s: None}
    Q = PriorityQueue()
    for node in A:
        Q.insert(node, d[node])
    for _ in range(V):
        node = Q.extract_min()
        for neighbor in neighbors(node):
            relax(A, weights, d, parent, node, neighbor)
            Q.decrease_key(neighbor, d[neighbor])
    return d, parent
```

Overall Runtime: V × insert + V × extract_min + E × decrease_key

- Q = Binary Heap (faster if graph is sparse and o(E)=O(V)) → Insert = O(log V), Extract_Min = O(log V), Decrease_Key = O(log V), Overall = O((V+E) log V)
- Q = Direct Access Array (faster if graph is dense and very connected) → Insert = O(1), Extract_Min = O(V), Decrease_Key = O(1), Overall = O(V²+E)
- Q = Fibonacci Heap (fastest asymptotic runtime) → Insert = O(1), Extract_Min = O(log V), Decrease_Key = O(1), Overall = O(V log V + E)

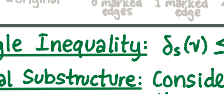
SP summary

General Technique for SP Algorithms

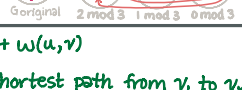
sometimes it's possible to use a faster algorithm by modifying the graph

- specify which node to start at, identify nodes & edges
- sometimes its helpful to add an auxilary node
- to find longest path, negate edges
- phrase solution as a SP problem
- Shortest path from some start node(s) to some end node(s) optimizing some quantity (↳ modify edge weights or relaxation step) with some constraint/requirement (↳ graph copying: each copy of the graph represents a state & state transitions are connections between graph copies)

Find SP s to t using ≤2 marked edges



Find SP from s to t using a multiple of 3 edges (* starts end ∞ in 0 and 3 layers)



Decidability

Since # possible problems > # possible programs, some problems don't have programs & thus are undecidable

- R → decidable problems
- EXP → problems decidable in exponential time → $2^{n^{O(1)}}$
- P → problems decidable in polynomial time → $n^{O(1)}$
- NP → set of decision problems that are verifiable
  - ↳ verifiable → there exists a certificate in which we can verify is valid if the solution to problem is true

Problem	Cerificate (aka a possible solution)
SP	A path with weight ≤ d
Knapsack	A set of items with total value ≥ V and capacity ≤ C
non-trivial decision problem	NP → no certificate

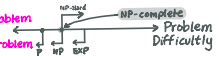
- P = NP?
  - ↳ Are problems that are verifiable in polynomial time also solvable in polynomial time?
- P ⊊ EXP ⊊ R
- P ⊊ NP ⊊ EXP ⊊ R

Reductions: To solve problem A, convert it to problem B that you already can solve efficiently

- If A ≤ B ⇒ A ≤ B (B is at least as hard as A)

NP-Hard → all other problems in NP can be reduced to this problem in polynomial time
↳ the problem is at least as hard as the hardest problem (or any problem) in NP

How to show problem is NP-Hard
ⓐ reduce every problem in NP to the given problem
OR
ⓑ reduce an NP-Hard problem into the given problem
↳ NP-Hard → A ⇒ NP-Hard ≤ A



How to show problem is NP-Complete
- NP-complete = NP-Hard & in NP

B = unknown
A = NP-Hard or NP-complete

NP Complete Problems: 3SAT, Hamiltonian Path, Knapsack, Clique, 3 coloring

if B can be used to solve A, we reduced A to B, & B is at least as hard as A
we convert a problem we know is hard into a problem you want to show is hard → want to show B is NP-Hard → if we can show how to solve A in poly time using [a black box to solve B in polytime] then B is NP-Hard (aka this blackbox can solve A & any other NP problem in poly time

TSR/DAG-SP

- relax each node's outgoing edges in topological order

```
DAG_shortest_path(A, w, s):
    topo_order = topo_order(A, s)   # DFS O(V+E)
    d = [∞ for v in A]
    parent = {s: None}
    d[s] = 0
    for u in topo_order:
        for v in A[u]:              } relax outgoing edges O(V+E)
            relax(u,v)
    return d, parent
```

AVL COUNT

SELECT AVL AUGMENTATION

Design an augmented AVL tree which supports the following operations in $O(\log n)$ time.
COUNT-GREATER-THAN(x): Return the number of values greater than x.
COUNT-LESS-THAN(x): Return the number of values less than x.
Solution: We augment the AVL tree such that *node.count* stores the number of nodes in the subtree of x. This satisfies the following relation: *node.count* = *node.left.count* + 1 + *node.right.count*. Any time a node is affected, we iteratively update *node*, *node.parent*, *node.parent.parent*, ... until the root is updated. Therefore, updating one node (inserting or deleting it) takes $O(\log n)$. Anytime we INSERT or DELETE, we update any affected nodes in $O(\log n)$. Note that anytime there's a rotation in the tree, we update the relevant counts: if a node has new children, its count is updated, as well as the counts of its parents augmentation preserves the runtime of all the AVL tree operations.
Now, to implement COUNT-GREATER-THAN, we may FIND the node, updating counts as we go. We start at the root, *node*: if *node.value* < x, we continue to *node.right*. If *node.value* > x, we add *node.right.count* to our running count, and continue to *node.left*. Finally, if *node.value* = x, we terminate our algorithm. Because we are performing $O(1)$ operations at each node, and we are traversing through $O(h) = O(\log n)$ nodes, COUNT-GREATER-THAN runs in $O(\log n)$ time. The implementation of COUNT-LESS-THAN is analogous.
Using part (a), design a data structure to implement INSERT and COUNT-OUTLIERS.
Solution: We use the augmented AVL tree from part (a). In addition, we keep track of n, the number of nodes in the tree (and update it accordinely durine deletions and insertions) Using the SELECT algorithm from the AVL recitation notes we may find Q_1 = SELECT($\lfloor n/4 \rfloor$) and Q_3 = SELECT($\lfloor 3n/4 \rfloor$). Then, we return COUNT-LESS-THAN($Q_1 - 1.5(Q_3 - Q_1)$) + COUNT-GREATER-THAN($Q_3 + 1.5(Q_3 - Q_1)$), which takes $O(\log n)$ time in total.

DP

1) Choose Subproblems
   Ex: fib(n) = n^{th} fibonacci #
2) Relate subproblems using recursion
   Ex: fib(0) = 1, fib(1) = 1, fib(n) = fib(n−1) + fib(n−2)
3) Show that the subproblems form a DAG (ie acyclic)
   ↳ show that each subproblem depends only on "smaller" subproblems
4) Evaluate the subproblems & Original problem (use parent pointers?)
   ↳ we can use bottom-up (↳ iterative) or top-down (↳ recursive) approach to evaluate (both are equally valid)

memoization

Ex: Bottom Up Approach:

```
def fib(n):
    sub = [None for i in range(n+1)]
    sub[0] = sub[1] = 1
    for i in range(2, n+1):
        sub[i] = sub[i-1] + sub[i-2]
    return sub[n]
```

Ex: Top-Down Approach

```
def fib(n, sub=None):
    if sub is None:
        sub = [None for i in range(n+1)]
    if sub[n] != None:
        return sub[n]
    if n ≤ 1:
        sub[n] = n
        return sub[n]
    sub[n] = fib(n-1, sub) + fib(n-2, sub)
    return sub[n]
```

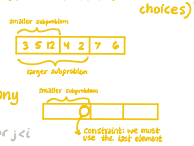
5) Analyze Runtime
   ↳ Runtime = # subproblems × work per subproblem
   Ex: For above examples, runtime = O(n) × O(1) = O(n)

Example: Fibonacci
$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

DP Problem Solving Tips

1) Think about how the problem is making a sequence of choices/selections
2) Think about how your first selection reduces the overall search space (total sequence = first choice + remaining choices)
   ↳ helps you figure out what your subproblem search space is
3) Think about what subproblem to use
   ↳ always start with the obvious one
   ↳ larger subproblem must be able to utilize any solution to smaller subproblem
   ↳ Constrain the smaller subproblem so you can decide whether you can use it for any solution to the smaller subproblem
   ex: subproblem: X[i] = longest increasing subsequence ending @ element i; X[i] → X[j] for j<i
4) Think about base cases
   ↳ Search space becomes constant size



Describe a linear time algorithm to find a flight itinerary with the fewest flights, all of them being short, or to determine that no such itinerary exists. Construct a directed graph G with one vertex per airport and a directed edge from one airport to another exactly when there exists a short route between them. This graph takes $O(a+r)$ time to build, containing a vertices and at most r edges. Run BFS on this graph, starting at the home airport. If BFS finds the destination airport, return the path found, which by BFS will be a shortest path. Otherwise no route exists using only short flights, so return that. BFS runs in $O(|V|+|E|)$, so total runtime is $O(a+r)$ as desired.

Coffee: Mr. G loves to drink coffee, but unfortunately not all flights serve coffee. Describe a linear time algorithm to find a flight itinerary with the fewest flights that do not serve coffee, or to determine that no such itinerary exists. Construct a directed graph G with one vertex per airport and a directed edge from one airport to another exactly when there exists a route between them. This graph has size $O(a+r)$ and can be constructed in as much time. Run a modified BFS on the graph, starting at the home city. When processing an airport v in the frontier, run BFS locally starting at v, exploring only edges representing routes that serve coffee, adding all newly explored airports to the frontier, and adding parent pointers to the original BFS tree. If the modified BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. To see that this algorithm returns an itinerary with the fewest flights that do not serve coffee, observe that all airports reachable with exactly k flights not serving coffee are processed consecutively with this algorithm, so if the destination city is found for the first time, the path is guaranteed to be a path with fewest flights not serving coffee. Each edge is traversed at most once, so the modified BFS also runs in $O(a+r)$ as desired.

Jon wants to make sure his itinerary is *lasagna dense*: for any airport on his itinerary, at least one of the next, current, or previous airports on the itinerary contains an Italian restaurant. Let d be the maximum number of routes out of any airport. Describe a $O(d^3a)$ time algorithm to find a lasagna dense flight itinerary with the fewest number of flights, or to determine that no such itinerary exists. You may assume that the home and destination airports both contain Italian restaurants. For each airport $v \in V$ containing an Italian restaurant, run BFS from v, but only searching airports within three flights of v. Store a list P_v of all $O(d^3)$ paths of flights starting from airport v with length at most 3 that ends at another Italian restaurant. Each BFS can be performed in $O(d^3)$ time for, each of the at most a airports containing an Italian restaurant. Then construct a new graph G' in the following way. Add a vertex for each airport containing an Italian restaurant. Then for each vertex, add a new path from v to u for each path in P_v that ends in airport u. Run BFS on G' starting at the home airport, searching for the destination airport. If BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. This graph G' has at most $a + 2d^3a$ vertices and at most $3d^3a$ edges, so can be constructed and searched with BFS in $O(d^3a + a)$ time. **Faster Solution**: construct a new graph G' containing $3a$ vertices with three vertices for each airport: $(a,0)$, $(a,1)$, and $(a,2)$. Let node (a,i) represent landing in airport i having last been at an airport with an Italian restaurant j airports before. For all edges (v,u) in G, add edges to G' in the following way. If u does not $(v,1)$ to $(u,2)$. Alternatively, if u contains an Italian restaurant, add an edge to $(u,0)$ from (v,i) for $j \in [0,2]$. Let s be the starting airport and t be the destination airport. Run BFS on G', beginning at $(s,0)$ and ending at $(t,0)$. If BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. The runtime for constructing the graph and running BFS is $O(da+a)$, as there exists $3a$ vertices and at most $3da$ edges in G'.

Describe a $O(c(a+r+1))$ time algorithm to find a flight itinerary with the fewest flights containing no two consecutive flights on the same airline, or to determine that no such itinerary exists. Construct a graph G on ca vertices, one for each airport/airline pair (a_i, c_j). Connect an edge from (a_i, c_j) to (a_k, c_l) if there is a flight from airport a_i to airport a_j on airline c_l, and $c_j \neq c_l$. Add an additional vertex v representing the home airport and connect it to each vertex (v, c_j) for $j \in [1,c]$. This graph has $ca+1$ vertices and $(c-1)r + c$ edges and can be constructed in $O(c(a+r+1))$ time. Run BFS starting from vertex v, searching for any vertex associated with the destination airport. If BFS finds the destination airport, return the path found. Otherwise, no route exists. This algorithm is correct because any edge traversed corresponds to a valid route, and no two consecutive edges are flights on the same airline. BFS runs in $O(|V|+|E|)$, so total runtime is $O(c(a+r+1))$

A graph is *connected* if there is a path between every pair of vertices. A graph is *simple* if every edge connects different vertices, and no two distinct edges exist between the same pair of vertices. What is the maximum and minimum number of edges in a connected simple undirected graph on $|V|$ vertices? What about for simple undirected graphs that are not connected? What about connected undirected graphs that are not simple?

If an undirected graph is connected, edges must form at minimum a spanning tree of the vertices. If it is simple and connected, the most edges occurs in a complete graph, with an edge between every pair of vertices. If the graph is simple but not connected, no edge is necessary. However, the maximum number of edges is less than the connected case as the graph must contain at least two distinct connected components, and is maximized when one of the components is a single vertex. If the graph is connected but not simple, a spanning tree must exist, in addition to one more edge to make the graph non-simple; but maximum number of edges is unbounded.

Connected	Simple	Min	Max
Y	Y	$\|V\|-1$	$\|V\|(\|V\|-1)/2$
N	Y	0	$(\|V\|-1)(\|V\|-2)/2$
Y	N	$\|V\|$	∞
N	N	0	∞

Gill realizes that his code is quickly able to explore vertices of the graph that are within half the diameter from a given configuration, but it takes a long time to explore vertices that are more than half the diameter away. Let d be the maximum degree of the Pocket Cube graph, and let w be its diameter. Describe an algorithm to find a shortest sequence of moves to solve a given Pocket Cube configuration by visiting no more than $2d^{\lfloor w/2 \rfloor}$ configurations. Run BFS from both the query configuration and the solved configuration, but alternating exploring frontiers from each. Store parent pointers to the configuration preceded by each explored configuration. After exploring each frontier, check if a configuration in the new frontier has been explored by the other BFS. If it has been explored by the other BFS, construct the path from the query configuration to the solved configuration through the overlapping node, by following parent pointers from one BFS to the query configuration, and following parent pointers from the other BFS to the solved configuration. Because we alternate exploring frontiers, the lengths of the paths found by each BFS differ by at most one, so each has length at most $\lfloor w/2 \rfloor$. Further, a frontier at distance i from its source contains at most d^i, so the number of configurations explored is at most $d^{\lfloor w/2 \rfloor}$ for each of the two BFSs, as desired.

ComCat wants to direct traffic in the network along routes that maximize catwidth. Given a pair of servers, describe an algorithm to compute a route with the largest catwidth in order to direct cat videos between them. Construct graph G on the servers, with an undirected edge from server s_i to s_j with weight b when s_i and s_j have a connection between them with bandwidth w. Let s_1 and s_2 be the queried pair of servers. We run Dijkstra from s_1 to compute paths that maximize catwidth by modifying the relaxation step. Modify δ to represent a lower bound of catwidth, initializing $\delta[s_1]$ to ∞, and $\delta[v]$ to -1 for all other $v \in V$. Store a max priority queue of servers, with server s keyed by $\delta[s]$. The maximum catwidth from u to v will either be the catwidth of an edge from u, or the minimum between the maximum catwidth from u to a neighbor $v_n \in Adj[u]$ and the bandwidth b of the edge from (v_n, v). We modify the relaxation step of Dijkstra to increase the lower bounds δ to satisfy this relation, removing servers from the priority queue when it has maximum δ from among servers not yet processed. Because we have only changed the relaxation step of Dijkstra which may still be computed in $O(1)$ time the running time of Dijkstra depends on the data structure for the priority queue. Using a Fibonacci Heap acheives $O(|V|\log|V| + |E|)$

Assume that the number of servers, i.e. asymptotically quadratic in the number of servers, i.e. asymptotically quadratic in the number of servers. Can you simplify the implementation of data structures used by your algorithm based on the assumed server density? A $O(|V|\log|V|+|E|)$ runtime is acheivable by implementing the priority queue in Dijkstra with a Fibonacci heap. When $|E| = O(|V|^2)$, and thus running time is quadratic. Diikstra also runs in $O(|V|^2)$ using an array as your priority queue. Note that using a binary heap for the priority queue would result in a $O(|V|^2 \log|V|)$ running time, which is asymptotically slower than using an array.

Currency Exchange: Travelwhy is a currency exchange company that buys and sells currencies at different prices, with a possibly different fixed rate everyday. Travelwhy is concerned that the daily prices automatically generated by their computer might allow someone to make money off of them, simply by buying and selling the right sequence of currencies on their exchange. Describe an algorithm to check if there's a way to make money on their exchange on a given day. We note that the logarithm of a product of exchange rates is equal to the sum of the logarithms of the rates. Construct graph G with vertices on currencies, with a directed edge from currancy a to currency b with weight $-\log r$ when there is a conversion from a to b at rate r. Thus, if there is a negative weight cycle in G, there will be a cyclic sequence of conversion rates whose product is positive. Here's an algorithm: run $|V|$ rounds of Bellman-Ford from an arbitrary currency c in the graph in $O(|V||E|)$ time. If any edge is relaxed in the final round, a negative cycle exists, thus there is a way to make money on the exchange. However This algorithm assumes that all currencies are convertible to every other currency on the exchange; otherwise shortest paths from c may be infinite, and we may never find a negative cycle not reachable from c. It is a reasonable assumption but you don't need the graph to be connected to solve the problem in $O(|V||E|)$ time. Assume that not all vertices are reachable from c. Run BFS from c to mark all vertices V' and edges E' reachable from c, and run Bellman-Ford on the subgraph (V', E') to find any negative weight cycles within it. This takes $O(|V'||E'|)$ time. Remove V' and E' from the graph and recurse on the remaining subgraph, running BFS then Bellman-Ford from some remaining vertex, checking for negative weight cycles. Do this repeatedly until all vertices and edges in the graph have been processed. Notice that each edge is searched by BFS at most once, and relaxed by Bellman-Ford at most $V' = O(|V|)$ times, so this algorithm also runs in $O(|V||E|)$ time.

Harriet's informant has given her details on various terrorists and who they take orders from. A terrorist's *influence* is the set of terrorists he can give orders to, directly or indirectly; a terrorist will never take orders from someone within their influence, but may take orders directly from multiple people. A *terrorist cell leader* is a terrorist who takes orders from no one; a leader's influence is called a *terrorist cell*. Based on her informant's information, Harriet needs to generate a list of terrorist cells, and for each cell, construct a list of the cell's members so that a terrorist within the influence of another appears lower down on the list. Describe an algorithm to help Harriet generate these lists as quickly as you can. Construct a directed graph G with a vertex associated with each terrorist, and an edge from terrorist a to terrorist b if b takes orders from a. Because a terrorist does not take orders from someone within their influence, G is acyclic. Identify cell leaders by looping over vertices, and for each check whether any incident edge terminates at the vertex. This identification process loops over vertices and evaluates each edge at most twice, so leaders can be identified using this algorithm in $O(|V|+|E|)$ time. Then for each cell leader, run BFS to find the subgraph of vertices within their influence. Then, topologically sorting vertices within the same cell produces an ordering for which a terrorist within the influence of another occurs later in the sort. The influence of each cell may contain $O(V)$ vertices and $O(E)$ edges, and there are at most $O(V)$ cells, so these lists can be generated in $O(|V|(|V|+|E|))$ by the above procedure.

Harriet's informant has been discovered by a terrorist cell. He's been captured and is being held, chained in an elaborate system of underground tunnels. Fortunately, the guards do not patrol the tunnels at night, so Harriet has been flown in to rescue him. Unfortunately, Harriet has no information about the tunnels or where in the tunnels her informant is being held, except that she will be able to reach him from the tunnel's only entrance. Inside the tunnels, there is a lantern hanging at each intersection to light the way. Time is short, but Harriet thinks there will be enough time to scratch a single arrow on the ground at each intersection. Explore the tunnels using depth first search. When entering an intersection containing no markings, draw an arrow pointing to the tunnel from which you entered. Then search counterclockwise, continuing to the next unexplored tunnel. Harriet must also take care to remember if she is actively searching or backtracking the tunnel she is currently in. If she enters an already arrowed intersection and she is actively searching, she backtracks along the same tunnel she came in. If she enters an already arrowed intersection and she is backtracking, she should explore the next counterclockwise tunnel. If an arrow points to the next counterclockwise tunnel, then backtrack along that tunnel. This is standard DFS from a local perspective, so takes time linear in the number of tunnels plus the number of intersections.

Harriet needs to recommend to her superiors a list of candidate terrorists for capture to help break up the cells. Cell leaders are very difficult to capture, so she has been asked to prioritize non-leader terrorists whose individual capture would decrease the size of a cell leader's influence by more than one. Assume that only one terrorist from the list will be pursued at this time. Describe an algorithm to help Harriet identify a list of candidate targets for capture as quickly as you can. Construct graph G with each vertex labeled with the cells in which the terrorist associated with the vertex appears, and each cell leader containing the number of terrorists within their influence. The following brute force algorithm can identify candidate terrorists in $O(|V|(|V|+|E|))$ time. For each terrorist t and for every cell leader c, remove t from G and run BFS from c on the new graph to count the number of nodes reachable from c in the modified graph. If c's influence is exactly one less than c's influence in G, then t is not a candidate; otherwise c is a candidate. This brute force algorithm can be slow, but it gets the job done.

A more sophisticated algorithm can identify candidates in $O(|V|(|V|+|E|))$ time. We notice that a non-leader t is a candidate (i.e. individual capture would decrease the size of the influence of leader c by more than one) if and only if t has an outgoing edge to a vertex v, whose only incoming edge from a vertex within c's influence is vertex t. The reverse implication is trivial: removing t from the graph would disconnect v from c because t was the only vertex within c's influence with an edge to v. Alternatively, suppose t's capture decreases the size of the influence of leader c by more than one. Let V be the nonempty set of t's influence that would be disconnected from c upon the removal of t, and let v be the first $v \in V$ occuring in a topological sort of G. We argue that v's only parent contained in c's influence is t. Suppose for contradiction that v has another parent p within c's influence that is not t. Vertex p must be in V or else v would not be disconnected by removal of t. But then p would come before v in a topological order, and we have reached a contradiction.

Downhill Skiing: Warbler is a ski resort bidding to be the location for next year's downhill ski competition. The resort contains numerous downhill trails weaving down the mountain with each trail reachable from the lodge at the peak. Warbler has collected trail reviews from customers, and has compiled an average rating for each trail between -5 and 5. The steering committee for the competition has scheduled a visit to evaluate the resort. They will have time for one evaluation downhill ski tour starting at the peak. For the tour, Warbler wants to choose a path of downhill trails beginning at the peak that maximizes the sum of trail ratings along the path. Describe an algorithm to quickly find such a path. Construct directed graph $G = (V, E)$ where vertices are locations where downhill ski trails start, end, or diverge, and edges represent downhill ski trails. Each route is downhill, so G will be acyclic. Weight each edge as the negative average rating of the trail associated with the edge. Use DFS to topologically sort the intersections. Initialize each vertex v with an upper bound on the minimum weight path $\delta(v)$ from the peak, i.e. 0 for the peak and $-\infty$ for all other vertices. Relax edges in order based on the topological sort order of their strating vertex. This algorithm computes shortest weighted paths in $O(|V|+|E|)$ time.

Minimal Destination Paths: Let $G = (V, E, w)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, containing negative weights but no negative weight cycles. A **destination path** to vertex $v \in V$ is a path with smallest weight starting from any vertex $u \in V$ that terminates at vertex v. Note that vertex u may be v, resulting in a 0 weight destination path. A destination path is **minimal** if its weight is less than or equal to the weight of any other destination path terminating at v. Describe an algorithm to compute the weight of a minimal destination path for every vertex in V, in $O(|V||E|)$ time. Add vertex s to G to construct a new graph G', connecting to every vertex in V with a 0 weight edge. Run Bellman-Ford to find the weight of a shortest path from s to each vertex in $O(|V||E|)$. The weight of a shortest path from s to vertex v is identical to the weight of the minimum destination path to v. If it were not, then path between them of smaller weight could be used to construct a smaller path satisfying the other.

Driving Options: Describe an algorithm to find both the shortest and second shortest routes between two locations. Assume that every road supports traffic in both directions. Construct undirected graph $G = (V, E)$ where vertices are connections between verticies and edges are roads. Weight each road based on the length of the road, which will be a positive number. Because weights are positive, we may apply Dijkstra to compute the shortest path between a pair of cities in $O(|V|\log|V| + |E|)$ time using a Fibonacci Heap for the priority queue. If we assume the road network is planar (which is reasonable for a road network), the number of edges will be linear in the number of vertices $|E| = O(|V|)$, so you may choose to use a binary heap to implement the priority queue to achieve a $O(|V|\log|V|)$ runtime bound. **Paths:** To find a second shortest path use a brute force algorithm: remove an edge contained in the shortest path from the graph, and find the shortest path in the new graph using Dijkstra. Doing this for each edge in the shortest path containing at most $|V|-1$ edges yields a $O(|V|(|V|\log|V|+|E|))$ time. If we store parent pointers during Dijkstra, we can follow them back to the source to return a requested path. **Walks:** we can compute second shortest walks in $O(|V|\log|V|+|E|)$ by performing one pass of a modified Dijkstra on the graph. In addition to maintaining δ representing upperbounds on shortest paths, we also maintain δ_2 representing upperbounds on second shortest walks. In our priority queue, we will store pairs (v, i) for each $v \in V$, and for i equal to 1 and 2. Pair $(v, 1)$ will be keyed by $\delta(v)$ while $(v, 2)$ will be keyed by $\delta_2(v)$. Initialize both $\delta(v)$ and $\delta_2(v)$ to ∞ for each $v \in V$, except for the source vertex s where we initialize both $\delta(s)$ and $\delta_2(s)$ to 0. As with Dijkstra, we will repeatedly remove and process pairs with minimum key. To process a pair $(u, 1)$, for each edge (u, v) outgoing from u, relax $\delta(v)$ normally as in Dijkstra, replace $\delta(v)$ with $\delta(u) + w(u, v)$ if it is smaller, or replace $\delta_2(v)$ with $\delta(u) + w(u, v)$ if it is strictly between $\delta(v)$ and $\delta_2(v)$. Also, if $\delta(v)$ was updated, replace $\delta_2(u)$ with the old value of $\delta(v)$. Similarly, to process a pair $(u, 2)$, for each edge (u, v) outgoing from u, replace $\delta_2(v)$ with the smaller of $\delta_2(u) + w(u, v)$ and $\delta(u) + w(u, v)$ that is also larger than $\delta(v)$ and smaller than $\delta_2(v)$. When updating δ and δ_2 values, also update the corresponding keys in the priority queue. This algorithm finds shortest path lengths δ identically to Dijkstra, but also finds second shortest walk lengths δ_2 because it is last updated exactly second time that Dijkstra reaches the vertex from a shortest or second shortest walk. To return the walks, we also store parent pointers, we can determine the next parent on the second shortest walk by comparing whether $\delta_2(v)$ equals $w(\pi(v), v) + \delta(\pi(v))$ or $w(\pi(v), v) + \delta_2(\pi(v))$.

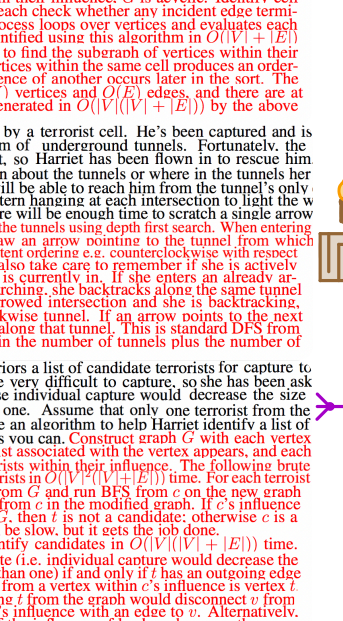
[10 points] **Party Separation:** Haddy Ceron has gotten into trouble with some of her, shall we say, difficult friends: her friends do not get along with each other. She would like to invite them all to a party, but that would cause too much drama. Haddy decides instead to throw two separate parties, one on Friday and one on Saturday night. She wants to invite each of her friends to exactly one of the parties, but not to both. Haddy is specifically worried about conflicting pairs of friends who intensely dislike each other. Given a list of conflicting pairs among her friends, describe a linear time algorithm to determine whether Haddy can assign friends to parties, without assigning any conflicting pair of friends to the same party.

Solution: Construct a graph on Haddy's friends with an edge between friend a and friend b if they are a conflicting pair. We must determine whether this graph is **bipartite**: every vertex can be assigned one of two colors, either red or blue, such that no edge connects to vertices of the same color. Pick an arbitrary vertex $v \in G$ and color it blue. Run a breadth-first search from v and color red all vertices reachable from v in one step. Continue the breadth-first search, coloring nodes in even levels blue and odd levels red. Then loop through the edges of the graph. If there exists an edge connecting two vertices a and b having the same color, the shortest paths from v to r and b, together with the edge, form a cycle with odd length. This cycle is not two-colorable, so friends in this cycle cannot be assigned to parties without some conflicting pair being invited to the same party. Otherwise, if no edge between same colored vertices exists, then invite red friends on Friday and blue friends on Saturday, free from drama!

[10 points] The **separation between two vertices** in an undirected graph is the length of the shortest path between them. The **separation of an undirected graph** is the maximum separation between any two vertices in the graph. The **maximum separation of a vertex** in an undirected graph is the maximum separation between the vertex and any other vertex in the graph. A vertex is a **centroid** of an undirected graph if its maximum separation is minimum across all vertices. Let G be a **connected** undirected graph, with d the separation of G and c the maximum separation of a centroid of G. Prove that $c \leq d \leq 2c$.

Solution: Clearly since d is the maximum separation between any two vertices, and the centroid is a vertex minimizing its maximum separation, then $c \leq d$. To show $d \leq 2c$, let v be a centroid of G. For any pair of vertices a and b, there is a path of length at least c from a to v, and a path of length at least c from v to b. Thus there exists a path from a to b of length at least $2c$, so the separation between a and b cannot be larger or else the path from a to v to b would be shorter. So every pair of vertices has a path between them with length at least $2c$, so the minimum path length between any pair (d) cannot be greater than this number.

Label edges in graphs G_1, G_2, and G_3 as either tree (T), back (B), forward (F), or cross (C), using a CCW DFS.



List vertices in a topological sort or argue that a topological sort does not exist.

G_1: [1, 3, 4, 2]

G_2: Undirected graph contains back edges, thus cycles: no topological sort.

G_3: [1, 2, 5, 3, 8, 9, 6, 4, 7]

- Case 1: If Louis enters a room for which all the doors in the room are open, then Louis followed a tree edge into a new room. Tell Louis to close the door he entered through (leaving it unlocked for now) then choose an open door to pass through into another room. Closed doors will represent parent pointers in DFS tree.
- Case 2: If Louis enters a room containing a closed door and does not lock the door he entered through, then Louis followed a back edge into a room visited previously. Tell Louis to go back via the hallway he just came from, locking both doors on his way out.
- Case 3: If Louis enters a room containing a closed door and locks the door he entered through, then Louis has just backtracked along a tree or back edge. If there is an open door, tell Louis to go through it, continuing the exploration. Otherwise, all other doors are locked, so tell Louis go through through the closed door hallway, locking its doors behind him.

GRAPHS

2 Chronicles 15:7

	Subproblem	relation	DAG	base cases	Solution *memoization*	runtime
Longest Increasing Subsequence (not necessarily contiguous)	$x[i]$ = length of longest subsequence for subarray a_0 to a_i, ending in a_i P	$x[i] = \max\{x[j]+1 \mid j \in [0,i-1], a_j < a_i\} \cup \{1\}$	subproblems $x[i]$ only depend on strictly smaller i	$x[0] = 1$	$\max\{x[i]\}$ for $i \in [0,n-1]$ *store parent pointers* to reconstruct line breaks	$O(n)$ subproblems $O(n)$ work per subproblem $O(n^2)$ overall runtime
Text Justification	$x[i]$ = minimum badness sum of formatting the words from w_i to w_{n-1} S	$x[i] = \min_{j \in [i,n-1]}(b(i,j) + x[j+1])$	subproblems $x[i]$ only depend on strictly larger i	$x[n] = 0$ (badness of justifying zero words is zero)	$x[0]$ *store parent pointers* to reconstruct line breaks	$O(n)$ subproblems $O(n)$ work per subproblem $O(n^2)$ overall runtime
Alternating Coin Game	$x(i,j)$ = maximum value winnable for player moving when # coins $j-i+1$ is even, starting from the contiguous subsequence of coins from i to j C	player must choose coin i or coin j. Even player will seek to maximize $x(i,j)$ & odd player will seek to minimize $x(i,j)$. $x(i,j) = \begin{cases} \max(v(i)+x(i+1,j), v(j)+x(i,j+1)) & j-i+1 \text{ even} \\ \min(x(i+1,j), x(i,j-1)) & j-i+1 \text{ odd} \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller $j-i$ ↳solve in order of increasing $j-i$	$x(i,i) = 0$ (even player never gains value from last coin)	if $x(1,n) \geq \frac{\text{sum(values)}}{2}$ play on even turns, odd turns otherwise	$O(n^2)$ subproblems $O(1)$ work per subproblem $O(n^2)$ overall runtime
Edit Distance	$x(i,j)$ = minimum number of edits to transform prefix up to $A(i)$ to prefix up to $B(i)$ P	$x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } A(i) = B(i) \\ 1 + \min(\underbrace{x(i-1,j)}_{\text{deletion}}, \underbrace{x(i,j-1)}_{\text{insertion}}, \underbrace{x(i-1,j-1)}_{\text{replacement}}) & \text{otherwise} \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller i & j	$x(i,0) = i$ $x(0,j) = j$ (need all insertions or all deletions)	$x(|A|,|B|)$ *store parent pointers* to reconstruct all edits	$O(n^2)$ subproblems $O(1)$ work per subproblem $O(n^2)$ overall runtime
Box Separation	$x(i)$ = minimum weight required to separate boxes 1 to i	we must move either box i or the one before $x(i) = \min\begin{cases} \text{weight}[i] + x(i-1) \\ \text{weight}[i-1] + x(i-2) \end{cases}$	subproblems $x(i)$ only depend on strictly smaller i	$x(0) = 0$ $x(1) = 0$	$x(n)$ where n is the total number of boxes	$O(n)$ subproblems $O(1)$ work per subproblem $O(n)$ overall runtime
Contiguous Subarray Sum (1D)	$x(i)$ = sum of largest contiguous subarray from 1 to i, ending at i	$x(i) = \max(x(i-1), 0) + A[i]$	subproblems $x(i)$ only depend on strictly smaller i solve bottom up	$x(0) = 0$	$x(n)$ where n is the total length of the array	$O(n)$ subproblems $O(1)$ work per subproblem $O(n)$ overall runtime
Contiguous Subarray Sum (2D)	$x(i,j)$ = sum of largest contiguous subarray with bottom left corner A[i][i] & top right corner A[i][j]	$x(i,j) = x(i-1,j) + x(i,j-1) - x(i-1,j-1) + A[i][j]$	subproblems $x(i,j)$ only depend on strictly smaller $i+j$	$x(i,0) = 0$ $x(0,j) = 0$	$x(n,m)$ for a $n \times m$ matrix	$O(nm)$ subproblems $O(1)$ work per subproblem $O(nm)$ overall runtime
Largest Black Square	$x(i,j)$ = height of largest black square in subarray with corners A[i][i] & A[i][j]	$x(i,j) = \begin{cases} 0 \text{ if } A[i][j] \text{ is } 0 \\ \min(x(i-1,j), x(i,j-1), x(i-1,j-1)) + 1 \text{ otherwise} \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller $i+j$	$x(i,0) = 0$ $x(0,j) = 0$	$[\max(x(i,j))]^2$ for $i,j \in [1,n]$	$O(n^2)$ subproblems $O(1)$ work per subproblem $O(n^2)$ overall runtime
Choosing Prizes	$x(i)$ = sum of best prize values from 1 to i when you choose prize i	$x(i) = \max\begin{cases} x(i-1) \\ A[i] + x(i-2) \end{cases}$	subproblems $x[i]$ only depend on strictly smaller i	$x(k) = 0$ for $k \leq 0$ no values if no prizes	$x(n)$ store parent pointers	$O(n)$ subproblems $O(1)$ work per subproblem $O(n)$ overall runtime
Subset Sum	$x(i,j)$ = True if can make sum j using items 1 to i, False otherwise	$x(i,j) = \text{OR}\begin{cases} x(i-1,j-A[i]) & \text{if } j \geq A[i] \\ x(i-1,j) & \text{always} \end{cases}$ for $i \in [0,n], j \in [0,S]$	subproblems $x(i,j)$ only depend on strictly smaller i	$x(i,0) = T$ for $i \in [0,n]$ $x(0,j) = F$ for $j \in [1,S]$	$x(n,S)$ n = # integers; S = sum	$O(nS)$ subproblems $O(1)$ work per subproblem $O(nS)$ overall runtime
0-1 Knapsack	$x(i,j)$ = maximum value by packing size j knapsack with items 1 to i	$x(i,j) = \begin{cases} \max(x(i-1,j-S_i)+V_i, x(i-1,j)) & \text{↳ if } j \geq S_i \\ x(i-1,j) & \text{otherwise} \end{cases}$ where $i \in [0,n]$ & $j \in [0,S]$	subproblems $x(i,j)$ only depend on strictly smaller i	$x(i,0) = 0$ for $i \in [0,n]$ $x(0,j) = 0$ for $j \in [0,S]$	$x(n,S)$ n = # items; S = size of bag store parent pointers	$O(nS)$ subproblems $O(1)$ work per subproblem $O(nS)$ overall runtime
Unbounded Knapsack	$x(j)$ = max value by packing knapsack of size j using provided items	$x(j) = \max(0, \max_{\text{where } j \in [0,S]}\{V_i + x(j-S_i) \mid i \in [1,n], S_i \leq j\})$	subproblems $x(j)$ only depend on strictly smaller i	$x(0) = 0$ no space to pack	$x(S)$ S = space in knapsack	$O(S)$ subproblems $O(n)$ work per subproblem $O(nS)$ overall runtime
Professor Devkusha	$x(i,j)$ = max coin value achievable if it is professors turn to divide c_i to c_j $m(i,j)$ = max coin value from c_i to c_j	$x(i,j) = \min_{k \in [i,j-1]}\left(\max\begin{cases} m(i,k)+x(k+1,j) \\ x(i,k)+m(k+1,j) \end{cases}\right)$ where $m(i,j) = \max(m(i,j-1), c_j)$	subproblems $x(i,j)$ & $m(i,j)$ only depend on strictly smaller $j-i$	$x(i,i) = 0$ $m(i,i) = c_i$	$x(1,n) \geq k$	$O(n^2)$ subproblems $O(n)$ work per subproblem $O(n^3)$ overall runtime
DNA Debacle	$x(i,j,k,\ell)$ = True if can match k seq. chars from A[i:] and ℓ from B[j:] to all chars in C[:k+ℓ], False otherwise	$x(i,j,k,\ell) = \text{OR}\begin{cases} x(i-1,j,k-1,\ell) \text{ if } A[i-1]=C[i-1] \\ x(i,j-1,k-1,\ell) \text{ if } B[j-1]=C[i-1] \\ x(i-1,j,k,\ell) \text{ always} \\ x(i,j-1,k,\ell) \text{ always} \end{cases}$	subproblems $x(i,j,k,\ell)$ depend on strictly smaller $i+j+k+\ell$	$x(i,j,0,0) = \text{True}$ $x(i,j,k,\ell) = \text{False}$ for $i < \max(k,0)$ $j < \max(\ell,0)$	$x(n,n,n/2,n/2)$	$O(n^4)$ subproblems $O(1)$ work per subproblem $O(n^4)$ overall runtime
Election Votes	$x(i,j)$ = min population to win exactly j electoral counts from districts 1 to i	$x(i,j) = \begin{cases} \min(x(i-1,j), \text{LP}[x(i-1,j-c_i)+v_i]) & \text{↳ if } j \geq c_i \\ x(i-1,j) & \text{otherwise} \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller i	$x(i,0) = 0$ for $i \in [0,n]$ $x(0,j) = \infty$ for $j \in [1,c]$	$x(n,c)$ or none if $x(n,c) = \infty$	$O(nc)$ subproblems $O(1)$ work per subproblem $O(nc)$ overall runtime
Wire wafer	$x(i,j)$ = max # of matchings in the interval p_i to p_j where $1 \leq i < j \leq n$	$x(i,j) = \max\begin{cases} x(i+1,j) \\ \max_{t \in [i+2,j] \text{ s.t. } a_t = a_i}(1 + x(i+1,t-1) + x(t+1,j)) \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller $j-i$	$x(i,j) = 0$ for $j-i \in \{1,0\}$	$\max(x(2,n), \max_{t \in [3,n-1] \text{ s.t. } a_t = a_1}(1 + x(2,t-1) + x(t+1,n)))$	$O(n^2)$ subproblems $O(n)$ work per subproblem $O(n^3)$ overall runtime
Longest Common Subsequence	$x(i,j)$ = length of longest common subsequence of prefixes A[:i], B[:j]	$x(i,j) = \begin{cases} x(i-1,j-1)+1 & \text{if } A[i] = B[i] \\ \max(x(i-1,j), x(i,j-1)) & \text{else} \end{cases}$	subproblems $x(i,j)$ only depend on strictly smaller $i+j$	$x(i,0) = 0$ $x(0,j) = 0$	$x(|A|,|B|)$ where A & B are strings	$O(\|A\|\|B\|)$ subproblems $O(1)$ work per subproblem $O(\|A\|\|B\|)$ overall runtime
Rod Cutting	$x(i)$ = max value obtainable by cutting rod of length i	$x(i) = \max\{x(i-j)+v(j) \mid j \in [1,i]\}$	subproblems $x(i)$ only depend on strictly smaller i	$x(0) = 0$	$x(n)$ where n is original length	$O(n)$ subproblems $O(n)$ work per subproblem $O(n^2)$ overall runtime
Arithmetic Parenthesization	$x(i,j,-1)$ = min parenthesized evaluation of subsequence from integer i to j	$x(i,j,+1) = \max_{k \in [i,j-1]\ s_1,s_2 \in \{+1,-1\}}\{O_k(x(i,k,s_1), x(k+1,j,s_2))\}$ $x(i,j,-1) = \min_{k \in [i,j-1]\ s_1,s_2 \in \{+1,-1\}}\{O_k(x(i,k,s_1), x(k+1,j,s_2))\}$	subproblems $x(i,j,s)$ only depend on strictly smaller $j-1$	$x(i,i,s) = a_i$	$x(1,n,+1)$ store 2 parent pointers	$O(n^2)$ subproblems $O(n)$ work per subproblem $O(n^3)$ overall runtime
Egg Drop	$x(f,e)$ = min # of drops to check any sequence of f floors using e eggs	$x(f,e) = 1 + \min\{\max(x(i-1,e-1), x(f-i,e)) \mid i \in [1,f]\}$	subproblems $x(f,e)$ only depend on strictly smaller f	$x(0,e) = 0$ $x(f,1) = f$	$x(n,k)$ n is # floors & k is # eggs	$O(nk)$ subproblems $O(n)$ work per subproblem $O(n^2k)$ overall runtime

[15 points] Design an algorithm to determine whether Chad can make money on the exchange, and if so, provide a sequence of currencies from US dollars to US dollars for which the sequence of conversions makes money. **Hint:** Products can be transformed into sums using logarithms.

Solution: Let $r(i,j)$ represent the conversion rate from currency i to currency j. Our goal is to find a sequence of currency exchanges $(v_0, \ldots, v_k)$ such that $\prod r(v_{i-1}, v_i) > 1$. We can convert this optimization to finding a negative sum by taking a negative logarithm of both sides: $\sum_{i=1}^{k} -\log(r(v_{i-1}, v_i)) < 0$.

Construct a graph with a vertex for each of the n currencies on the exchange. Add a directed edge from vertex u to vertex v weighted by $-\log(r(u,v))$. Assuming there is an exchange rate between every pair of currencies, there will be $n(n-1)$ edges in the graph. Run Bellman-Ford from the vertex associates with US dollars to find a negative weight cycle of weight $-c$. If the vertex associated with US dollars is on the cycle, return the cycle. Otherwise, convert from dollars to an arbitrary vertex v on the cycle and back to dollars. This single loop has weight d. Compute k such that $d - kc < 0$. Then return the single loop with k repetitions of the negative weight cycle in between, representing a conversion sequence that will make money. This algorithm takes $O(n \times n^2)$ time to find the negative weight cycle using Bellman-Ford, but the output sequence may be arbitrarily large depending on the value of k; however an implicit representation of the sequence could be returned in $O(n^3)$ time.

[15 points] **Longer Shortest Paths:** Given a connected undirected graph $G = (V, E)$ with positive edge weights, describe algorithms to find a minimum weight path (if one exists) between two specified vertices that uses: (1) **exactly** k edges in $O(k|E|)$ time, and (2) **at least** k edges in $O(k|E| + |V| \log |V|)$ time.

Solution: Let s and t be the two specified vertices. For part (1), to find a shortest path from s to v using exactly k vertices, we construct a graph on $(k+1)|V|$ vertices: (v,i) for each $v \in V$ and $i \in [0,k]$. Connect a directed edge from vertex $(u, i-1)$ to (v,i) for each edge $(u,v) \in E$ and for every $i \in [1,k]$. Since traversing an edge adds one to the second index of any vertex, this graph is acyclic and a path from vertex $(s,0)$ to a vertex (v,k) will represent a (possibly non-simple) path traversing exactly k edges from s to t in the original graph. To find such a path of minimum weight, relax edges from vertices in a topological sort order in $O((k+1)|V| + k|E|) = O(k|E|)$ time, since $|V| = O(|E|)$ in a connected graph, resulting in shortest path weight $\delta_s(v)_k$ for a path from s to v using exactly k edges.

For part (2), we take the original graph and add an auxiliary node a, with a directed edge (a,v) to each vertex $v \in V$, weighted by $\delta_s(v)_k$, the shortest path weight from s to each vertex using exactly k edges. A minimum weight path π from s to t using at least k edges will have as a prefix a minimum weight path using exactly k edges ending at some vertex u. Further, the (possibly empty) subset of the path π from u to t cannot have cycles (as they would have positive weight and could be removed), so finding a shortest path from a to t in this new graph will correspond to a minimum weight path using at least k edges. This graph has only positive weights, so we can apply Dijkstra in $O(|V| \log |V| + |E|)$ time (using a Fibonacci Heap to implement its queue). Constructing the graph takes $O(k|E|)$ time using the method from part (1), so this algorithm takes $O(k|E| + |V| \log |V|)$.

k & >k

[10 points] Given a list of classes you've already taken, describe a linear time algorithm to determine the earliest semester you can take 19.006, assuming you can take as many classes as you want in any semester.

Solution: Construct a graph with a vertex for each of the n classes at MIT. Add a directed edge from class a to class b if class a is a prerequisite to class b and you have NOT yet taken class a. Let m be the number of such prerequisites. Then add auxiliary vertex s and connect it to every class. We assume that this graph is acyclic (that no cycle of prerequisite dependencies exists), or else no one would be allowed to take a class in that cycle. Then the length of the longest path from s to class 19.006 in this graph, will be the minimum number of semesters you need to take.

For each class/vertex v in the graph, we will compute the minimum number of semesters $m(v)$ needed to take that class starting from node s. Let $N(v)$ be the set of prerequisites to v. Since v cannot be taken before any prerequisite, then $m(v) = 1 + \max_{u \in N(v)} m(u)$. Compute $m(v)$ in topological sort order of the vertices from s, with $m(s) = 0$. All of these can be computed in $O(n+m)$ time as each edge appears in at most one prerequisite list $N(v)$.

Alternatively, one can assign each edge of the graph a weight of -1 and run topological sort relaxation to find a path with lowest weight from s to 19.006 in $O(n+m)$ time. The negative weight of this path will then be the number of semesters required. Note that we can negate weights to find longest paths because the graph is acyclic. This strategy would fail if the graph contained positive weight cycles.

[10 points] Unfortunately, being able to take as many classes as you want isn't realistic, and your adviser is upset at you for proposing such a schedule. You've been given a credit limit of only 1 class per semester. Since you don't want to pay tuition indefinitely, design an algorithm, that returns a valid schedule for every semester that minimizes the number of semesters you're here.

Solution: First, we identify set of classes that must be taken prior to 19.006, and then we schedule those classes in a valid order. Begin with the same graph constructed for part (a), but remove any vertices from the graph not reachable from 19.006 following perquisite edges in the opposite direct. We can identify such vertices by running breadth- or depth-first search from 19.006 in $O(n+m)$ time, traversing edges only in the backward direction. Then, compute a topological sort order on the classes using DFS, and then schedule classes in the order of the topological sort, also in $O(n+m)$ time.

[10 points] **Shortest Solve:** Given a board configuration, describe an $O((nm+1)!)$ time algorithm to output a shortest sequence of moves that solves the board, if such a sequence of moves exists.

Solution: Perform a breadth-first search from a solved configuration of the board, producing parent pointers and expanding a frontier of configurations until the input configuration is found (can alternatively search from the input configuration until the solved configuration is found). When the target is found, follow parent pointers to produce a sequence of moves for return. If the search completes before finding the target configuration, the board is not solvable. Because breath-first search finds shortest paths in an unweighted graph, this algorithm will find a shortest sequence of moves. Breadth-first search runs in linear time with respect to graph size. By part (a), there are at most $(nm)!$ vertices in the graph. Each vertex has constant out-degree, so the number of edges is also $O((nm)!)$. By part (b), each configuration takes $O(nm)$ time to compute, so the breadth-first search takes $O((nm)! \cdot nm) = O((nm+1)!)$ time.

[10 points] **Rover Scheduling:** NASA scientists need to plan a task schedule for the Mars rover. Each task requires a certain amount of time to complete. One task may be required to complete prior to another. For example, before transmitting soil sample data to Earth, the samples must have been collected, and the antennae battery must have been charged. Given a list of rover tasks including their durations and dependencies, describe an algorithm to compute the shortest time to complete all tasks.

Solution: Construct a graph with a vertex for each task, and a directed edge from task a to task b if a must be completed prior to b, weighted by the **negation** of the time required to complete task a. Further, add vertices s and t, with a directed edge from s to every other vertex in the graph and a directed edge from every vertex in the graph to t, each with weight 0. If there are n tasks and m dependencies, this graph has size $O(n+m)$. Use depth-first search to check whether this graph contains and cycles; if it does, then no task on the cycle can be first, so no plan can be constructed. Otherwise, finding the shortest path from s to t will correspond to the longest time to complete any sequence of tasks, with all other tasks being performed in parallel. Since the graph is acyclic, relax edges in topological sort order to compute the path of minimum weight in $O(n+m)$ linear time, which is as faster than other methods.

[5 points] An *articulation point* of a connected undirected graph is any vertex that, if removed from the graph would make the graph no longer connected. For example, if only one path of vertices connects vertices u and v in a graph, then every vertex in the path between u and v would be an articulation node. Given an undirected graph, describe a $O(|V||E|)$ time algorithm to output a list of articulation points in a graph. **Challenge:** listing all articulation points of an undirected graph can be accomplished in $O(|V|+|E|)$ time.

Solution: For each vertex, check if removing it would disconnect the graph. The check can be done by running either BFS or DFS on any remaining vertex of the graph with the selected vertex removed. The running time is $O(|V|(|V|+|E|))$. Since the graph is connected, $|E| = \Omega(|V|)$, so the running time simplifies to $O(|V||E|)$.

The challenge finds articulation points in linear time. Run depth-first search from an arbitrary root vertex r. During the DFS, label each graph edge as a directed tree or back edge, and label each vertex v with its depth $d(v)$ in the DFS tree. Then for each vertex v, compute the lowest depth $D(v)$ of any vertex reachable from v via a (possible empty) path of tree edges followed by at most one back edge. If $T(v)$ and $B(v)$ are the sets of out-going tree and back edges from v respectively, then

$$D(v) = \min(\{D(u) | \, u \in T(v)\} \cup \{d(u) | \, u \in B(v)\}).$$

Because each edge is labeled out-going from at most one vertex, these $D(v)$ values can be computed in $O(|E|)$ time by computing in reverse topological sort order of the vertices. Any vertex v that is not the root of the DFS tree, will be an articulation point if and only if none of its children in the DFS tree can reach a vertex with depth lower than $d(v)$ (a rigorous proof of this statement is omitted). So, for each non-root vertex v with DFS tree children $C(v)$, v is an articulation point if and only if $d(v) \geq \min_{u \in C(v)} D(u)$. Again, all such evaluations can be computed in $O(|E|)$ time since each vertex appears as a child of at most one vertex, and each such relation corresponds to an edge of the graph. As a special case, the root of the DFS tree will be an articulation point only when it has more than one child in the DFS tree. Since $|E| = \Omega(|V|)$, this algorithm runs in $O(|E|)$ time.

[10 points] **Ground Transportation:** Tim the Beaver needs to get from MIT to a recruiting event in San Francisco, but the airlines no longer allow beavers to fly. Tim cannot drive, so he decides to travel to his destination via bus and train. Tim vastly prefers trains over buses, and wants his trip to be **bus-sparse**: the trip should not contain a transfer from a bus to another bus, only from a bus to a train, a train to a bus, or a train to a train. Given a map of bus and train routes across the country, describe a linear time algorithm to find a bus-sparse itinerary containing the fewest vehicle transfers, or tell Tim that no bus-sparse itinerary exists to his destination.

Solution: Construct a graph having two vertices per transit station v: vertex v_b representing arrival at the station by bus, and vertex v_t representing arrival at the station by train. For each train route from station u to station v, add directed edges (u_t, v_t) and (u_b, v_t) to the graph, and for each bus route from station u to station v, add directed edge (u_t, v_b) (not (u_b, v_b) as such an edge would mean taking two buses in a row). Let s be the starting station and t be the ending station. Use breadth-first search to determine whether a path exists from s_b or s_t to t_b or t_t. If there is, return the itinerary associated with the path; otherwise return that no such itinerary exists. Let n be the number of stations and m be the number of routes. The graph has two vertices for each station and at most two edges for each route, so the size of the graph is $O(n+m)$. Running breadth first search twice (once from s_b and once from s_t) takes $O(n+m)$ time, so this algorithm will answer TIM's query in linear time. (Note that running BFS once from s_t is sufficient since s_t is connected to all vertices that s_b is connected to.)

[10 points] **Purchasing Parts:** Faylee Krye is the mechanic on board a Sirefly-class spaceship named Ferenity. While docked at a spaceport, she heads into town to purchase six parts for the ship: a power converter, a flux capacitor, a hyper drive, an ion shield generator, a binary motivator, and an air freshener. Faylee has downloaded a map of the town including the location of the parts distributors and time estimates to travel long any particular road in town (you may assume that the number of parts distributors is large). Looking down the list of distributors, Faylee observes that each store sells at most one of the parts, so she will need to visit six different stores to buy everything. Assume that the time spent in any store is negligible compared to transit time. Describe an algorithm to find a route for Faylee to travel from the spaceport and back, purchasing all the needed spaceship parts in the minimum amount of time.

Solution: Let n be the number of stores and road intersections, let m be the number of the roads between stores and intersection in the town, and let s be location of the spaceport. Construct a graph on $2^6 \times n$ vertices of the form $(b_1, b_2, b_3, b_4, b_5, b_6, v)$, denoting that Faylee arrives at store or intersection v while either possessing or not possessing item i, depending on whether boolean b_i is True (1) or False (0) respectively. For every directed road (u, v) between stores or intersections u and v, add many directed edges weighted by the time to traverse the road. Specifically add an edge from vertex $(b_1, b_2, b_3, b_4, b_5, b_6, u)$ to vertex $(b_1, b_2, b_3, b_4, b_5, b_6, v)$ (Faylee did not buy anything at u), one edge for every assignment of the b_is; and if u is a store selling a desired part, say part 1 without loss of generality, add a directed edge from vertex $(0, b_2, b_3, b_4, b_5, b_6, u)$ to vertex $(1, b_2, b_3, b_4, b_5, b_6, v)$, one edge for every assignment of the remaining b_is. This graph has $2^6 n$ vertices and $(2^6 + 2^5)m$ edges, so the graph has size $O(n+m)$ if n and m are large compared to 2^6. A route that corresponds to purchasing all six items and returning home will be a path from vertex $(0, 0, 0, 0, 0, 0, s)$ to vertex $(1, 1, 1, 1, 1, 1, s)$ in the graph. Under the assumption that it takes positive time to traverse any road, we can run Dijkstra to find the shortest such path in $O(m+n \log n)$ time (using a Fibonacci Heap). Dijkstra is faster than Bellman-Ford, and the graph may have cycles and many weights, so linear time methods do not apply.

[5 points] Prove that for any connected undirected graph G on n vertices and any positive integer $k < n$, either G has some path of length at least k, or G has fewer than $k \cdot n$ edges.

Solution: Note that this problem statement is ambiguous as to whether the desired path must be **simple**. If non-simple paths are allowed, the following argument holds. If there is a cycle in the graph, then there is a path of length n, and if there is not, the undirected graph is a tree, and has at most $n-1$ edges.

If non-simple paths are disallowed, the following argument holds. Run depth-first search from some node in the graph and consider the longest path in the DFS tree. If it has length at least k, we are done. Otherwise, since G is undirected, there are no cross edges with respect to the DFS tree, so each edge of the graph connects an ancestor and a descendant in the DFS tree. This bounds the number of edges in the graph by counting the total number of ancestors of each descendant; but if the longest path is shorter than k, each descendant has at most $k-1$ ancestors. So there can be at most $(k-1)n$ edges.

[10 points] **Specific Improvement:** For many graphs, single source breadth-first search can be done faster by making the search **bidirectional**: perform a breadth-first search separately from s and t, alternating the exploration of level i from s, then level i from t, then level $i+1$ from s, etc., until the frontier levels from the two searches intersect, returning the concatenation of the two shortest paths found[1]. Consider the family of unweighted undirected graphs having bounded degree b, where a shortest path from vertices s to t is known to contain d edges. Describe a graph from this family for which breadth-first search from s takes at least $\Omega((b-1)^d)$ time, while bidirectional breadth-first search from s and t takes at most $O(b^{d/2})$ time.

Solution: Consider a tree rooted at s with lower branching factor $b-1$ and depth d. Let t be any leaf of the tree, so that there is a unique path from s to t. Let the vertices on this path be labeled $s = v_0$ through $t = v_d$. Regular breath-first search from s would explore the entire tree, which has size $\Omega((b-1)^d)$, so breath-first search will take at least that long to touch all nodes. On the other hand, a bidirectional search would terminate a breadth-first search from s at vertex $v_{d/2}$, exploring a subtree of depth $d/2$ of size $O((b-1)^{d/2})$. The search from t will also explore out $d/2$ levels before reaching node $v_{d/2}$. Each node at level $i > 0$ in the search from t has one parent and at most $b-1$ unexplored neighbors to add to level $i+1$, so the search from t also touches at most $O((b-1)^{d/2})$ nodes. Thus bidirectional search runs in at most $O((b-1)^{d/2}) = O(b^{d/2})$ time.

$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ with $\binom{n}{k} = 1$ when $k = 0$ or $k = n$. Help Pascal calculate his binomial but his broken calculator and notebook.

The recursive definition of a binomial coefficient is a natural dynamic program:

1. Subproblems: $p(i,j)$ represents $\binom{i}{j}$.
2. Relation: $p(i,j) = p(i-1, j-1) + p(i-1, j)$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $n+k$, so acyclic
   - Base case: $p(i,0) = p(i,i) = 1$ for any $i \in [0, n]$
4. Solution: Calculating $\binom{n}{k}$ is simply evaluating $p(n,k)$
5. Algorithm:
   - Use notepad to memoize results of smaller subproblems, either top-down or bottom-up
   - Running Time: At most $(n+1)(k+1)$ subproblem solutions written on notepad, $O(1)$ work per subproblem using the calculator, so $O(nk)$ time

Ally wants to spend her gift certificate by purchasing the fewest possible items at the store, allowing items to be purchased more than once. Given a price list of items in the store, describe an efficient algorithm to return a shortest shopping list to Ally. Assume there are n items, and let $c(i)$ be the price of item i, where items are indexed in an arbitrary linear order

1. Subproblems: $p(j)$ is the smallest number of items that can be chosen whose respective prices sum to j (i.e. filling a j dollar gift certificate).
2. Relation:
   - Either a minimizing subset uses item i or it does not.
   - Similar to Knapsack, where 'size' is the cost to fill, and the 'price' to minimize is the same for all items (1 per item used).
   - $p(j) = \min_{i \in [1,n]} p(j - c(i)) + 1$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller j, so acyclic
   - Base case: zero or negative gift certificate $p(0) = 0$, $p(j) = \infty$ for $j < 0$
4. Solution:
   - $p(d)$ represents filling a d dollar gift certificate using the fewest items
   - Shopping list is constructable by storing parent pointers to minimizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $d+1$ subproblems each taking $O(n)$ time, so $O(nd)$ time

Let A be the n characters of the person's DNA, and let B be the k characters of the disease substring

1. Subproblems: $p(i,j)$ is the minimum distance between any substring of $A[:i]$, and disease substring $B[:j]$.
2. Relation: Minimize over five local options
   (a) If last characters match, match them and recurse on smaller problem. Otherwise:
   (b) Replace $A[i]$ with $B[j]$ and match them at a cost of one modification
   (c) Insert $B[j]$ to the end of $A[:i]$ and match them at a cost of one modification
   (d) If we've already started matching, delete $A[i]$ and recurse at cost one
   (e) If we haven't started matching, find best match in smaller substring

$$p(i,j) = \min(\begin{array}{ll} p(i-1, j-1) \text{ if } A[i] = B[j], & (a) \\ p(i-1, j-1) + 1, & (b) \\ p(i, j-1) + 1, & (c) \\ p(i-1, j) + 1 \text{ if } j < k, & (d) \\ p(i-1, j) \text{ if } j = k) & (e) \end{array}$$

3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $i+j$, so acyclic
   - Base case: Done if no disease to match, $p(i,0) = 0$
   - Base case: Negative number of characters impossible, $p(i,j) = \infty$ for negative i or j
4. Solution:
   - $p(n,k)$ represents the minimum distance match between B and any substring of A
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(nk)$ subproblems each taking $O(1)$ time, so $O(nk)$ time

Assume there are n students, with $a(i)$ being the dodge ball ability of the ith student in the initial line up.

1. Subproblems: $p(i,j)$ and $q(i,j)$ are the maximum ability score you can achieve, when either you or your opponent choose next respectively, from among students remaining in a line up from student i to j, where $i \leq j$.
2. Relation:
   - You will maximize ability sum, while your opponent will minimize.
   - $p(i,j) = \max(q(i+1, j) + a(i), q(i, j-1) + a(j))$
   - $q(i,j) = \min(p(i+1, j), p(i, j-1))$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $j - i$, so acyclic
   - Base case: Only one choice if only one student, $p(i,i) = a(i), q(i,i) = 0$
4. Solution:
   - $p(1,n)$ represents a maximal score for you when choosing first
   - Optimal choices are constructable by storing parent pointers to maximizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(n^2)$ subproblems each taking $O(1)$ time, so $O(n^2)$ time

We will guess a pair of edges to bring together first and recurse on the remaining unmatched subsets. Let $c(i) \in \{Blue, Red\}$ be the color of edge i.

1. Subproblems: $p(i,j)$ is the maximum number of matchings from among any valid folding of the contiguous subset of edges from the ith to the jth, matching only within the subset.
2. Relation:
   - Guess the edge k that matches to edge i (could be any other edge from $i+1$ to j)
   - $p(i,j) = \max_{k \in [i+1, j]} p(i+1, k-1) + p(k+1, j) + (1 \text{ if } c(i) = c(k), \text{ and } 0 \text{ otherwise})$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $j - i$, so acyclic
   - Number of segments to match is given by $j - i + 1$
   - Base case: If odd or negative number of segments, valid folding impossible, so $p(i,j) = -\infty$ when $j - i + 1$ is odd or less than zero.
   - Base case: If exactly zero segments, no matchings, so $p(i,j) = 0$ when $j - i + 1 = 0$
4. Solution:
   - $p(1,n)$ represents the maximum number of matchings for the whole chain
   - Optimal matching is constructable by storing parent pointers to maximizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(n^2)$ subproblems each taking $O(n)$ time, so $O(n^3)$ time

Describe an efficient algorithm to calculate an assignment of each day to either 'buy', 'sell', or 'none' would have maximized his revenue during those days. Let $c(i)$ be the price of FredEx stock on day i

1. Subproblems: $p(i,j)$ maximum revenue trading only in the first i days while holding j shares at the end of day i (j can be either 0 or s).
2. Relation:
   - To maximize revenue in first i days, either Seorge sell shares on day i or not
   - If sell on day i, must be holding s shares to sell, so $j = s$
   - To buy on day i, must not be holding any shares, so $j = 0$
   - $p(i,0) = \max(p(i-1,0), p(i-1,s) + sc(i))$
   - $p(i,s) = \max(p(i-1,s), p(i-1,0) - sc(i))$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller i, so acyclic
   - Base case: First day, can buy or not buy, so $p(1,s) = -sc(1), p(1,0) = 0$
4. Solution:
   - $p(d,0)$ represents the maximum return in d days (holding shares at end is suboptimal)
   - Order of trades constructable by storing parent pointers to maximizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $2d$ subproblems each taking $O(1)$ time, so $O(d)$ time