6.006 Final Exam Review Session

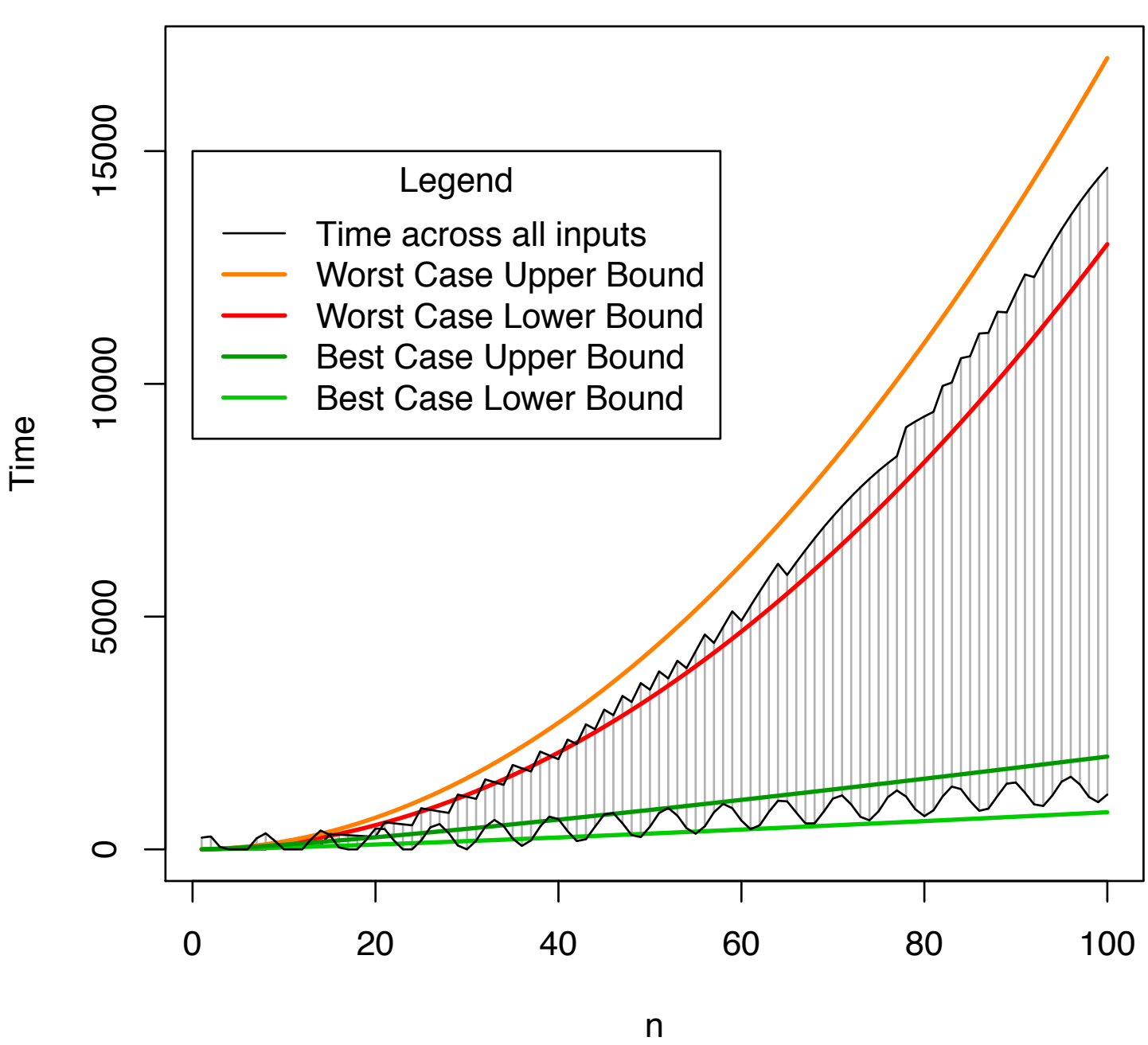
Michael Coulombe
Sabrina Liu
Srijon Mukerjee

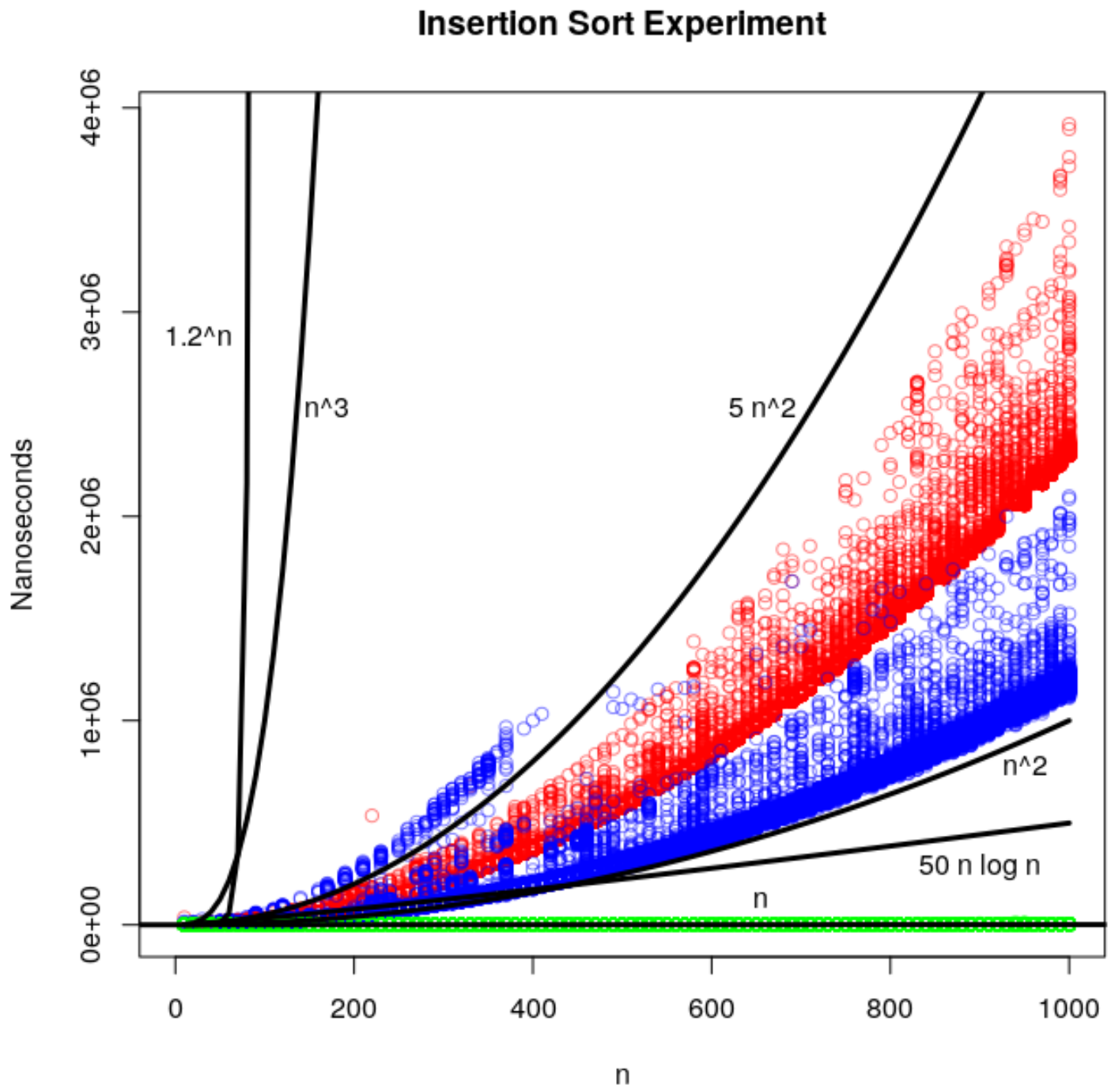
May 17, 2020

Announcements

"Some 6.006 TAs have kindly offered to staff some office hours before the final. The normal OHs Zoom Room will have at least one TA on staff on Sunday, Monday, and Tuesday."

5/17-5/19 @2-4pm and @8-10pm (eastern)

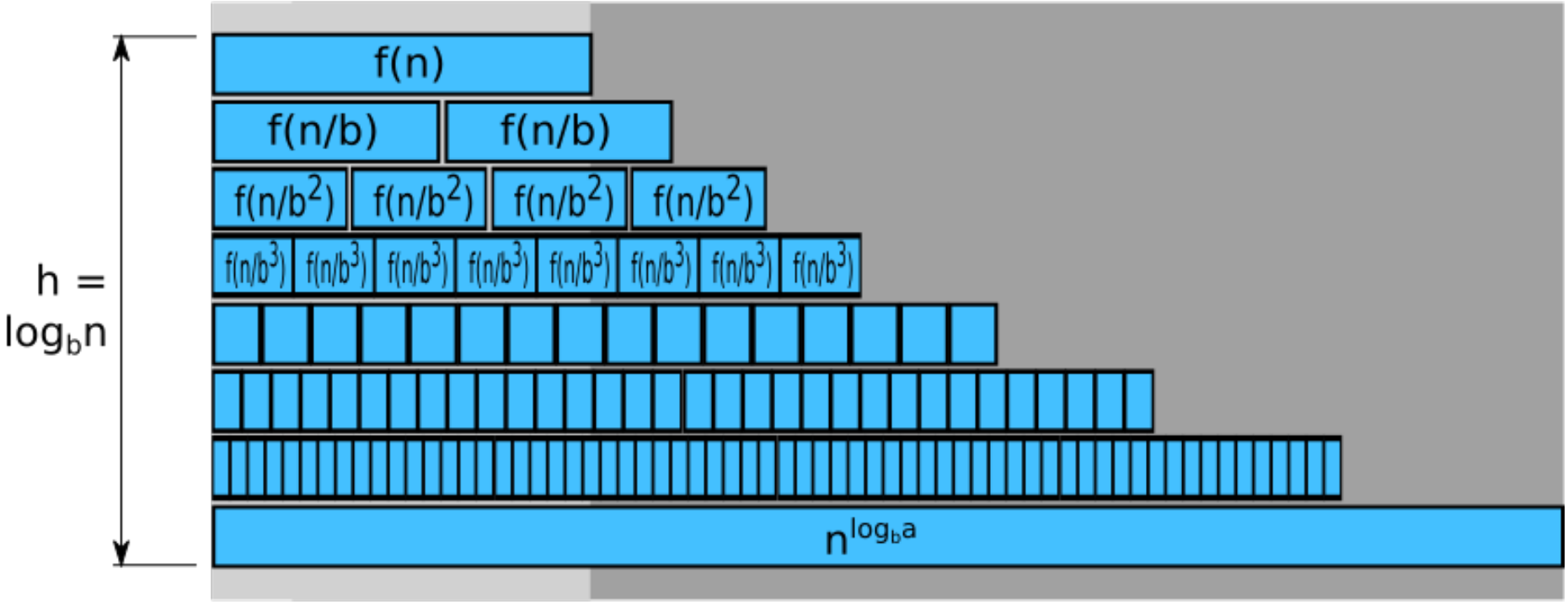

Run Time Graph
Legend
Time across all inputs
Worst Case Upper Bound
Worst Case Lower Bound
Best Case Upper Bound
Best Case Lower Bound
Time
0
5000
10000
15000
0
20
40
60
80
100
n


Insertion Sort Experiment
Nanoseconds
0e+00
1e+06
2e+06
3e+06
4e+06
0
200
400
600
800
1000
n
1.2^n
n^3
5 n^2
n^2
50 n log n
n

Master's Theorem

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

Case 1



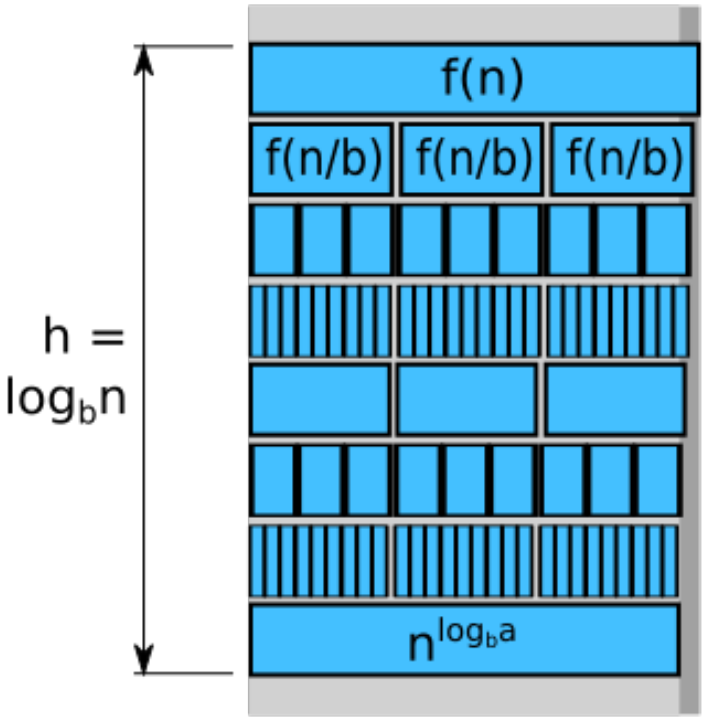
$$f(n) = O(n^{\log_b a} / n^{\varepsilon})$$

$$T(n) = \Theta(n^{\log_b a})$$

Master's Theorem

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

Case 2



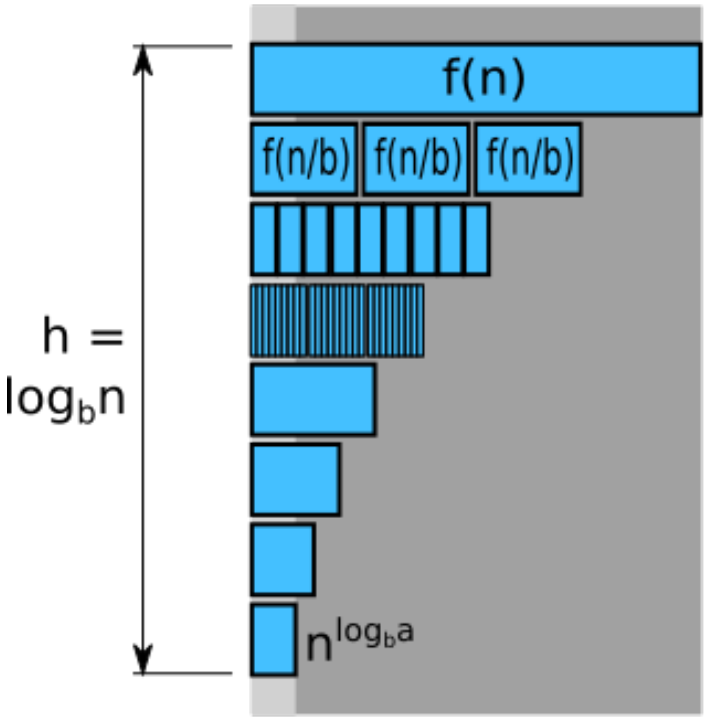
$$f(n) = \Theta(n^{\log_b a} \times \log^{\varepsilon} n)$$

$$\begin{aligned} T(n) &= \Theta(f(n) \times \log n) \\ &= \Theta(n^{\log_b a} \log^{\varepsilon+1} n) \end{aligned}$$

Master's Theorem

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

Case 3



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

$$T(n) = \Theta(f(n))$$

When you can't use Master's Theorem (Fall '11)

Example

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) = ???$$

1. By definition of $\Theta(n)$, must exist k, k' such that for large n:
$$kn \leq f(n) \leq k'n$$
2. Expand the recursion or draw tree (similar for $\leq$):
$$\begin{aligned} T(n) &\geq T(n/3) + T(2n/3) + kn \\ &\geq T(n/9) + 2T(2n/9) + T(4n/9) + 2 \times kn \\ &\geq T(n/27) + \ldots + T(8n/27) + 3 \times kn \\ &\geq T(n/3^i) + \ldots + T(n(2/3)^i) + i \times kn \end{aligned}$$
3. $\Omega(\log_3 n)$ levels with $\Omega(n)$ work $\implies T(n) = \Omega(n \log n)$
4. $O(\log_{3/2} n)$ levels with $O(n)$ work $\Rightarrow T(n) = O(n \log n)$

Sort by Number (Fall '18)

Describe an algorithm to sort n **pairs of integers** $(a_1, b_1), \ldots, (a_n, b_n)$ by increasing $f(a_i, b_i)$ value. If your algorithms use division, they should only use **integer** division.

- $f(a, b) = a/b$
  all input pairs (a_i, b_i) satisfy $0 \leq a_i < n$ and $0 < b_i < m < n$
  show how to sort pairs in $O(n \log m)$ time.

Solution:

1. Tuple Sort with (stable) Counting Sort, keyed on a_i then b_i.
2. Use a binary heap to perform a many-way merge on the $O(m)$ groups with same denominator using comparison:
   $(a_i, b_i) < (a_j, b_j) \iff a_i b_j < a_j b_i$
3. Tuple sorting takes $O(n)$ time, $O(n)$ heap insertions and deletions taking $O(\log m)$ time $\implies$ $O(n \log m)$ total time.

True or False? (Spring '18)

T F Given an array of n integers representing a binary min-heap, one can find and extract the maximum integer in the array in $O(\log n)$ time.

False. The maximum element could be in any leaf of the heap, and a binary heap on n nodes contains at least $\Omega(n)$ leaves.

True or False? (Spring '18)

T F Given an array of n key-value pairs, one can construct a hash table mapping keys to values in **expected** $O(n)$ time by inserting normally into the hash table one at a time, where collisions are resolved via chaining. If the pairs have unique keys, it is possible to construct the same hash table in **worst-case** $O(n)$ time.

True. Because keys are unique, we do not have to check for duplicate keys in chain during insertion, so each insert can take worst-case $O(1)$ time.

True or False? (Spring '18)

T F Any binary search tree on n nodes can be transformed into an AVL tree using $O(\log n)$ rotations.

False. Since any rotation changes height of any node by at most a constant, a chain of n nodes would require at least $\Omega(n - \log n)$ rotations.

Cooking Online (Fall '17)

(a) Chulia Jild wants to make a responsive website where people can find and share recipes. Let R represent the number of recipes listed on the website at any given time. Each recipe has a name and contains a constant length description of ingredients, cooking instructions, and unit serving price for the meal. Describe in detail a database that supports each of the following operations:

- `Add_Recipe`: add a recipe to the database in $O(\log R)$ time.
- `Find_Recipe`: return a recipe given its name in $O(1)$ time.
- `Similar_Price`: given a recipe name, return a list of 10 recipes closest in price to the given recipe in $O(\log R)$ time.

Construct a Set AVL tree with recipes stored at nodes, sorted by price. Additionally, store a hash table mapping a recipe name to its respective node in the tree. Use `pred/succ` to search for similar prices.

Graphs

6.006 Spring 2020 Morning Review Session

Checklist

- Full description of your graph:
  - What are the vertices?
  - What are the edges?
  - What are the edge weights (are there edge weights?)
  - How many edges (O-notation)
  - How many vertices (O-notation)
  - What is your source? What is your destination / are your destinations?
  - What type of graph is it? (i.e. DAG, positive weight, etc.) How do you know?
- Which algorithm(s) are you going to apply to the graph? Why?
- Runtime

NOT sssp graph things

- Connectivity / Connected Components
  - Full-DFS/Full-BFS in O(V + E) time works for **undirected** graphs
- Cycle detection
  - DFS finds cycles BUT only tells you if there is ≥ 1 cycle in the graph, will **not** find all
  - Bellman-Ford detects negative weight cycles **if they are reachable** from the source node
- Topological order
  - DFS only
- Reachability
  - DFS/BFS from source
- APSP
  - Johnson's in O(V^2 log(V) + VE)

SSSP Algorithms

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$

DFS

- SSSP: Not guaranteed to find shortest path **unless** graph is a DAG (DAG Relax)
- O(E)
- Can detect if a graph is a DAG or not
- reverse finishing order == topological order
- Does **not** count cycles
- Corrections to misconceptions from Exam 2:
  - Cannot be used to turn any graph into a DAG
  - Cannot be used to distinguish positive versus negative cycles

BFS

- Only finds shortest paths in unweighted graphs
- O(E)
- Corrections to misconceptions from Exam 2:
    - Cannot be used to turn any graph into a DAG

Dijkstra

- SSSP: non-negative weights (0 or positive)
- O(Vlog(V) + E)
- Traverses edges in order of the minimum path discovered so far
- Corrections to misconceptions from Exam 2:
  - You **can** use the potential function idea from Johnson's without running Bellman-Ford from a supernode for ASPS - just need to find the right one (Bellham's Fjord)

Bellman-Ford

- SSSP: generic graphs
- O(VE)
- Two ways of thinking about this one:
  - graph duplication (V times) and then DAG Relax
  - V-1 rounds of E relaxations (inductive proof)
- Detects shortest path at step V in either case - if there is a shortest path with V edges, it has a cycle somewhere.
- Marks nodes **reachable** from a negative weight cycle as -∞
- Corrections to misconceptions from Exam 2:
  - Does **not** count negative weight cycles alone
  - Only detects negative weight cycles reachable from a source - may need a supernode to find all of them

Johnson’s

- APSP for generic graphs
- O(V^2 log(V) + VE) time
- Not covered much on Quiz 2 (fair game for final)
- Add a supernode, use Bellman-Ford to calculate weights from supernode, and use as potential function in re-weighting edges for V rounds of Dijkstra’s
- Same runtime as Floyd-Warshall in dense graphs (E = O(V^2)) but better in sparse graphs (E = O(V))

Other techniques

- Graph duplication (ONLY way to store state other than shortest path information)
  - How are the states connected?
- Super node (possibly multiple starting points?)
  - How is it connected to the graph? Weighted/unweighted, directed/undirected?
- Maximal path?
  - Negate edge weights - **but** can introduce negative weight cycles
- Cannot use vertex weights -> convert to edge weights somehow

True/False

(i) **T** **F** Given a weighted directed graph $G = (V, E)$ and a source $s \in V$, if G has a negative-weight cycle somewhere, then the Bellman-Ford algorithm will necessarily compute an incorrect result for the distance from s to some vertex v in the graph.

True/False

(i) **T** **F** Given a weighted directed graph $G = (V, E)$ and a source $s \in V$, if G has a negative-weight cycle somewhere, then the Bellman-Ford algorithm will necessarily compute an incorrect result for the distance from s to some vertex v in the graph.

Solution: False. The negative-weight cycle has to be reachable from s.

True/False

(j) **T F** It is possible for a directed acyclic graph with n vertices and $(n-2)$ edges to have a unique topological order.

True/False

(j) T F It is possible for a directed acyclic graph with n vertices and $(n-2)$ edges to have a unique topological order.

Solution: False. Such a graph cannot be connected. Suppose we divide the graph into sets C_1 and C_2 that are disconnected from each other. Then a topological order of the vertices in C_1, followed by a topological order of the vertices in C_2, is a valid topological order of the entire graph; however, it is also possible to list the vertices in C_2 before those in C_1.

True/False

(g) **T** **F** If a graph contains no negative-weight cycles, there exists an ordering of edges such that Bellman-Ford computes shortest paths by relaxing each edge at most once.

True/False

(g) **T** **F** If a graph contains no negative-weight cycles, there exists an ordering of edges such that Bellman-Ford computes shortest paths by relaxing each edge at most once.

Solution: **True**, if there are no negative-weight cycles, there exists a directed shortest path tree. Relaxing in a topological sort order of that tree will only relax each edge once.

True/False

(l) **T** **F** If Johnson's algorithm always produces the correct all-pairs shortest-path weights, then G must have at least one edge with positive weight.

True/False

(l) **T** **F** If Johnson's algorithm always produces the correct all-pairs shortest-path weights, then G must have at least one edge with positive weight.

Solution: False. It could be a DAG with all negative-weight edges.

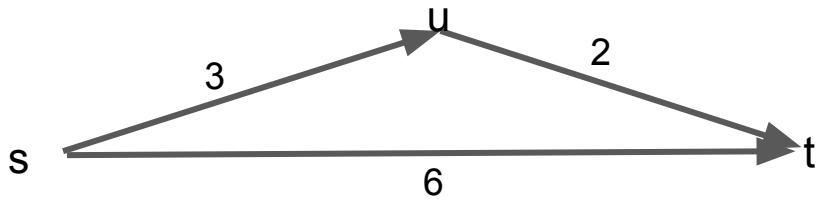
Fall 2015 Quiz 2 Problem 1d

(d) **T** **F** We can use Dijkstra's algorithm on graphs with negative edge weights by adding some large constant C to every edge, running Dijkstra's algorithm and then subtracting C from every edge in the result.

Fall 2015 Quiz 2 Problem 1d

(d) **T** **F** We can use Dijkstra's algorithm on graphs with negative edge weights by adding some large constant C to every edge, running Dijkstra's algorithm and then subtracting C from every edge in the result.

Solution: False. For example, the shortest s-t path for the graph on the left is $\langle s, u, t \rangle$. However, after adding $C = 4$ to the weights, we obtain the graph on the right, whose shortest s-t path is $\langle s, t \rangle$.



Fall 2007 Quiz 2 Problem 1d

(c) If the DFS finishing time $f[u] > f[v]$ for two vertices u and v in a directed graph G, and u and v are in the same DFS tree in the DFS forest, then u is an ancestor of v in the depth first tree.

True **False**

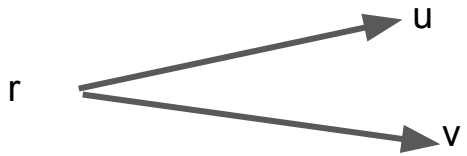
Explain:

Fall 2007 Quiz 2 Problem 1d

(c) If the DFS finishing time $f[u] > f[v]$ for two vertices u and v in a directed graph G, and u and v are in the same DFS tree in the DFS forest, then u is an ancestor of v in the depth first tree.

True **False**

Explain:



Solution: **False.** In a graph with three nodes, r u and v, with edges (r, u) and (r, v), and r is the starting point for the DFS, u and v are siblings in the DFS tree, neither as the ancestor of the other.

2008 Fall Quiz 2 Problem 1f

(f) T F If a topological sort exists for the vertices in a directed graph, then a DFS on the graph will produce no back edges.
Explain:

2008 Fall Quiz 2 Problem 1f

(f) T F If a topological sort exists for the vertices in a directed graph, then a DFS on the graph will produce no back edges.
Explain:

Solution: **True.** Both parts of the statement hold if and only if the graph is acyclic.

FR

Problem 5. [20 points] **Algorithmic Adventure Time!** (3 parts)

Finn and Jake are trying to travel through the wonderful country of Dijkstan, which consists of ***cities*** $V = \{v_1, v_2, \ldots, v_{|V|}\}$ and one-way ***flights*** $E = \{e_1, e_2, \ldots, e_{|E|}\}$ connecting ordered pairs of cities. Each flight $e \in E$ has a positive integer ***time*** $t(e)$ and positive integer ***cost*** $c(e)$ required to traverse it (measured in minutes and dollars, respectively).

You may assume that there exists a path between every pair of cities, and that there is at most one flight between any ordered pair of cities.

Answer the following questions to help Finn and Jake plan our their schedules. In the following parts, partial credit will be awarded for correct solutions, but the amount of credit depends on the efficiency of your algorithm. **Clearly specify your running time.**

(a) [5 points] Finn wants to travel from city s to city t using the path p that minimizes $a \cdot t(p) + b \cdot c(p)$, where $t(p)$ and $c(p)$ are the total time and cost of the edges (flights) along path p, respectively, and a and b are positive real numbers. Design an $O(V \log V + E)$-time algorithm to compute the optimal such route.

FR

- What are the vertices?
  - Cities - O(V)
- What are the edges?
  - Flights - directed (**one-way**) - O(E)
- What are the edge weights (are there edge weights?)
  - The cost we want to minimize: a * t(p) + b * c(p)
- What is your source? What is your destination / are your destinations?
  - s and t - givens
- What type of graph is it? (i.e. DAG, positive weight, etc.) How do you know?
  - Positive weight!
- Which algorithm(s) are you going to apply to the graph? Why?
  - Dijkstra's because of positive weight
- Runtime: O(V log(V) + E)

FR

Problem 5. [20 points] **Algorithmic Adventure Time!** (3 parts)

Finn and Jake are trying to travel through the wonderful country of Dijkstan, which consists of ***cities*** $V = \{v_1, v_2, \ldots, v_{|V|}\}$ and one-way ***flights*** $E = \{e_1, e_2, \ldots, e_{|E|}\}$ connecting ordered pairs of cities. Each flight $e \in E$ has a positive integer ***time*** $t(e)$ and positive integer ***cost*** $c(e)$ required to traverse it (measured in minutes and dollars, respectively).

You may assume that there exists a path between every pair of cities, and that there is at most one flight between any ordered pair of cities.

Answer the following questions to help Finn and Jake plan our their schedules. In the following parts, partial credit will be awarded for correct solutions, but the amount of credit depends on the efficiency of your algorithm. **Clearly specify your running time.**

(a) [5 points] Finn wants to travel from city s to city t using the path p that minimizes $a \cdot t(p) + b \cdot c(p)$, where $t(p)$ and $c(p)$ are the total time and cost of the edges (flights) along path p, respectively, and a and b are positive real numbers. Design an $O(V \log V + E)$-time algorithm to compute the optimal such route.

Solution: Weight each edge e by $w(e) = a \cdot t(e) + b \cdot c(e)$. Because every edge weight is positive, we can run Dijkstra's algorithm to find the shortest path from s to t, which runs in $O(V \log V + E)$.

FR

Problem 4-2. [20 points] **Zootopia**

Judy has just arrived at the Zootopia train station. She rents a car and wants to get from the train station to the police department. There are n junctions in Zootopia connected by m bidirectional edges (which is either a street, or a bridge, etc). We know that it is possible to get from any junction to any other junction. For each edge $e_i = (u_i, v_i)$, she knows w_i, the number of minutes it takes to go from u_i to v_i. Moreover, at every vertex v, there is a traffic light that is red for $r_v > 0$ minutes and then green for $g_v > 0$ minutes starting at time 0. We know that the cycle length is the same for all traffic lights (i.e., $g_v + r_v = T$) and therefore all traffic lights become red again in time T.

Moreover, while driving across edge e_i, her car uses d_i gallons of fuel per minute, and while stopping at vertex v, her car uses s_v gallons of fuel per minute.

You are given all the information about Zootopia, that is all the vertices along with the *integer* values g_v, r_v and s_v, and the structure of all the edges along with the *integer* values w_i and d_i. Give an algorithm that, in time $O(mT + nT \log(nT))$, finds the minimum amount of fuel her car needs to get from the train station (vertex 1) to the police department (vertex n).

FR

- What are the vertices?
  - Junctions, duplicated for each time modulo T - O(nT)
- What are the edges?
  - Undirected edges between junctions, change state when time increases - O(mT)
  - Edges don’t move junctions on red light - i.e. only increases time - O(nT)
- What are the edge weights (are there edge weights?)
  - d_i when moving, s_i when stopped
- What is your source? What is your destination / are your destinations?
  - Source: vertex 1, time 0 || Destination: vertex n, time t (all t)
- What type of graph is it? (i.e. DAG, positive weight, etc.) How do you know?
  - Non-negative! (can’t use negative gas)
- Which algorithm(s) are you going to apply to the graph? Why?
  - Dijkstra’s because of non-negative weights
- Runtime: O(E + V log(V)) = O(nT + mT + (nT) log(nT)) = O(mT + (nT) log(nT))

FR

Problem 4-2. [20 points] **Zootopia**

Judy has just arrived at the Zootopia train station. She rents a car and wants to get from the train station to the police department. There are n junctions in Zootopia connected by m bidirectional edges (which is either a street, or a bridge, etc). We know that it is possible to get from any junction to any other junction. For each edge $e_i = (u_i, v_i)$, she knows w_i, the number of minutes it takes to go from u_i to v_i. Moreover, at every vertex v, there is a traffic light that is red for $r_v > 0$ minutes and then green for $g_v > 0$ minutes starting at time 0. We know that the cycle length is the same for all traffic lights (i.e., $g_v + r_v = T$) and therefore all traffic lights become red again in time T.

Moreover, while driving across edge e_i, her car uses d_i gallons of fuel per minute, and while stopping at vertex v, her car uses s_v gallons of fuel per minute.

You are given all the information about Zootopia, that is all the vertices along with the *integer* values g_v, r_v and s_v, and the structure of all the edges along with the *integer* values w_i and d_i. Give an algorithm that, in time $O(mT + nT \log(nT))$, finds the minimum amount of fuel her car needs to get from the train station (vertex 1) to the police department (vertex n).

Solution: We build a graph $G' = (V', E')$ with nT vertices and $O(mT + nT)$ edges as follows. $V' = \{(v, t) | v \in V, 0 \leq t < T\}$ is the set of vertices where (v, t) corresponds to being in vertex v at time t' where $(t' \mod T) = t$. The edge set $E' = E'_1 \cup E'_2$ contains two types of weighted edges, where the weights correspond to the amount of fuel.

- First if there is an edge $e_i = (u_i, v_i)$ in the input graph G, then for all values of t such that $r_v \leq t < T$, we put an edge from (u, t) to $(v, (t + w_i) \mod T)$ with weight $w_i d_i$. This type of edges model traversing a street.
- Second, for each vertex $(v, t) \in V'$, we put an edge from (v, t) to $(v, (t + 1) \mod T)$ with weight equal to s_v. This type of edges model stopping at a junction for one minute.

We then use Dijskstra on this graph with the source vertex $(1, 0)$ to get to vertices (n, t) for all values of $t < T$ and we take the minimum value over all these T vertices. We use Fibonacci heap in the Dijskstra to get running time equal to $O(mT + nT \log(nT))$.

FR

Problem 9. Star Power! [20 points]

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which (along with making you temporarily invincible) lets you traverse the edge of your choice for free. In other words, you may change the weight of any one edge to zero.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, given that you may set one edge weight to zero. Analyze your algorithm's running time. For full credit, your algorithm should have a running time of $O(E + V \lg V)$, but partial credit will be awarded for slower solutions.

FR

- What are the vertices?
  - Regular vertices but duplicates, one for each state - 2V
- What are the edges?
  - Three versions of each edge, in states and across states (using Star) - 3E
- What are the edge weights (are there edge weights?)
  - Normal if within states, 0 if crossing from state 0 to state 1 (using Star the one time)
- What is your source? What is your destination / are your destinations?
  - s_0 and t_1 - it is **never** optimal to **not** use Star, as original weights are positive
- What type of graph is it? (i.e. DAG, positive weight, etc.) How do you know?
  - Non-negative!
- Which algorithm(s) are you going to apply to the graph? Why?
  - Dijkstra's because of non-negative weights
- Runtime: O(V log(V) + E)

FR

Solution 1: Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path.

The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra: $\Theta(E + V \lg V)$.

Solution 2: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, v_0 and v_1, in G'. The v_0 vertices represent being at v in G and having not yet used the Star. The v_1 vertices represent being at v, and having already used the Star.

For each edge $(u, v) \in E$, create three edges in G': (u_0, v_0) and (u_1, v_1) with the original weight $w(u, v)$, and (u_0, v_1) with a weight of zero. An edge, (u_0, v_1), crossing from the "0" component to the "1" component of the graph represents using the Star to traverse (u, v).

Now, run Dijkstra's algorithm with source s_0 and destination t_1. Drop the subscripts from the resulting path to recover the desired answer. (Observe that, because all edge weights are positive, it is always optimal to use the Star somewhere.)

G' has $2V$ vertices and $3E$ edges. Thus, constructing G' takes $\Theta(V+E)$ time, and running Dijkstra on G' takes $\Theta(E + V \lg V)$ time. Thus the total running time is $\Theta(E + V \lg V)$.

Fall 2017 PSet 7 Problem 7-2c

(c) [10 points] **Currency Exchange:** Travelwhy is a currency exchange company that buys and sells currencies at different prices, with a possibly different fixed rate every day. For instance, today they might purchase $1 US Dollar from you and give you €0.8 Euros in return, though they might also purchase €1 Euro from you in exchange for only $1.10 US Dollars. In this case, we would say the exchange rate from US Dollars to Euros is 0.8, while the exchange rate from Euros to US Dollars is 1.1. 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.

Fall 2017 PSet 7 Problem 7-2c

Solution:

The problem is asking whether there exists a sequence of successive exchange rates whose product is more than one. 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 there's a subtlety here. 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 for this problem to assume connectedness (so you will receive full credit for simply applying Bellman-Ford), 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.

Fall 2018 PSet 7 Problem 7-4

Problem 7-4. [12 points] **Escape to the Surface**

After solving PS6, Lelda has successfully reached the underground cave where Zink was being held, and now they must escape via one of the above-ground cave entrances. As before, some caves are marked **dangerous**, where Zink and Lelda will need to fight enemies in order to pass. With their combined fighting abilities, Zink and Lelda can win any fight without losing a single heart, but Zink's sword will take damage during the scuffle.[3] If ever Zink's sword takes a total of k damage, it will break. Beetle's map indicates the strength of enemies at any cave, from which they can estimate how much damage will be dealt to Zink's sword in that cave. To maintain morale, they decide only to traverse tunnels that make **vertical upward progress** to the surface, i.e. they will only traverse a tunnel from cave a to cave b if cave b has strictly higher elevation than cave a (Beetle's map also indicates cave elevations). Describe an efficient algorithm to determine whether Lelda and Zink can escape to the surface always traveling up, without ever breaking Zink's sword.

[3] Lelda uses a bow and arrow which does not take damage.

Fall 2018 PSet 7 Problem 7-4

Solution: Construct a graph G where each of the C caves corresponds to a vertex, having a directed edge for each of the T tunnels, adding an edge from cave a to cave b if: there is a tunnel from a to b and the elevation of cave b is strictly higher then the elevation of cave a, weighted by the amount of damage that would be done to Zink's sword at cave a. In addition, add a single vertex t, and a directed edge from each above-ground cave entrance to t, also weighted by the amount of damage that would be done at the corresponding cave entrance. Let vertex s correspond to the cave where Lelda and Zink begin. Then Lelda and Zink can escape to the surface without breaking Zink's sword if and only if a minimum weight path from s to t in G has weight less than k. G is acyclic because edges only connect to strictly higher caves, so compute $k > \delta(s, t)$ via DAG shortest paths in $O(C + T)$ time, relaxing edges from vertices in a topological sort order.

2009 Fall Quiz 2

Problem 5. Road Network [15 points]

Consider a road network modelled as a weighted undirected graph G with positive edge weights where edges represent roads connecting cities in G. However some roads are known to be very rough, and while traversing from city s to t we never want to take a route that takes more than a single rough road. Assume a boolean attribute $r[e]$ for each edge e which indicates if e is rough or not. Give an efficient algorithm to compute the shortest distance between two cities s and t that doesn't traverse more than a single rough road. (Hint: Transform G and use a standard shortest path algorithm as a black-box.)

2009 Fall Quiz 2

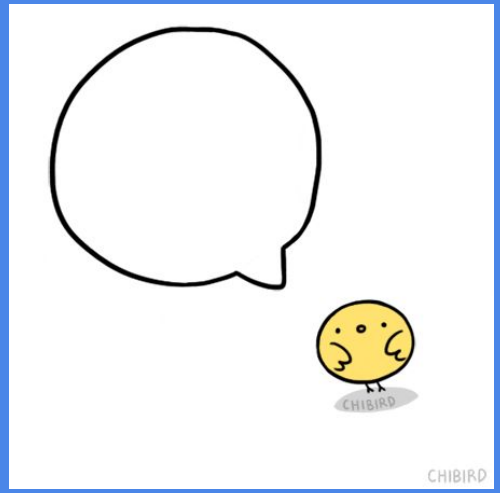
Solution: Transform the graph G to get the graph $G' = (V', E')$ in the following way:

For every vertex $v \in V(G)$, create two vertices v_r and v_s in G'. For every smooth edge $(u, v) \in E(G)$, create edges (u_r, v_r) and (u_s, v_s) in G'. For every rough edge $(u, v) \in E(G)$, create edges (u_s, v_r) and (v_s, u_r) in G'. Now we can run Dijkstra's shortest path algorithm to compute the shortest paths from s_s to t_s and from s_s to t_r and select the minimum of the two. Creating G' from G takes $O(E + V)$ time and has $2V$ vertices and $2E$ edges. Running Dijkstra takes $O(2E + 2V \log 2V)$, i.e. $O(E + V \log V)$ time using fibonacci heaps. Alternatively, we can also add 0 weight edges from u_s to u_r for all $u \in V(G)$ and run Dijsktra once from s_s to t_r which still has the same asymptotic runtime.

The rough vertices $v_r \in V(G')$ model the scenario when one rough edge has been traversed during the path. From construction we can guarantee that any path in G' can have atmost one rough edge. As soon as a rough edge is traversed, from the smooth vertices region we reach the rough vertices region and now there are no outgoing rough edges from this region, only smooth edges can be traversed from this point onwards.

Dynamic Programming

6.006 Final Review



Subproblems

- suffix/prefix/substring/combinations
- possibly include “state vars” as inputs
- DEFINE ALL VARIABLES YOU USE
- Define valid ranges for variables
  - Optional, but will help for relate, base cases, and time complexity
- Your subproblem *should not* define a problem of constant size
  - Don’t do:
    - x(i) is the maximum possible value at element i
  - Do
    - x(i) is the maximum possible sum of values for elements A[i:]

Prefixes

suffixes!

Relate

- What are the possible cases? What are you “guessing”?
  - Each “case” should be a part of your formula
- If you’re using “OR”, YOU CANNOT DO ARITHMETIC ON YOUR BOOLEAN
  - Don’t do
    - x(i) = OR({1+x(i-1), 1+x(i-2)})
  - Do
    - x(i) = max({1+x(i-1) if f(i-1), 1+x(i-2) if f(i-2), -∞})
      - f(i) is some function that tells you if that case is possible, optional (dep on q)
- Only use variables that are in your subproblem input or have been globally defined

Topological Sort

- “Subproblems depend only on strictly <increasing/decreasing> <vars/combination of vars>, so acyclic”
  - ^literally that’s it
  - if you have a + in the inputs of the subproblem in your relate step, increasing vars; if you have a -, decreasing
    - if you have multiple vars and one’s plus and one’s minus, pick the one that always has a +/-
    - if all vars sometimes stay the same, do a formula combination of vars that’s strictly increasing/decreasing (usually adding/subtracting) or define a successive ordering

Base cases

- smallest case possible / case with the fewest number of elements/no element
- Check your relate step-all cases accessible from your relate step should have a base case
- If you have an OR/”if possible” question-should have a set of base case for True/if possible and another set for False/±∞

Original Problem

- REREAD THE Q and make sure you're answering it
- Include "Use Parent Pointers" if you also need the path to your subproblem
- Could possibly be the max/min across multiple possibilities

Time Complexity

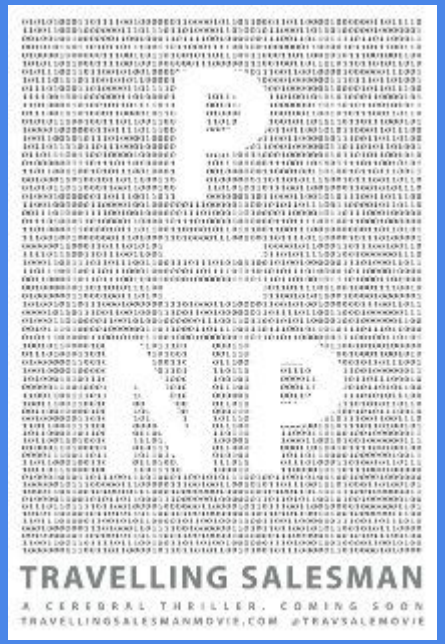
- # subproblems = product of number of options for each input
- (# subproblems) * (time/subproblem) + (time of original problem)
- If time's too long, maybe:
  - DP the arithmetic in relate?
  - Reduce the number of inputs?
    - Calculate one input from the others
    - Approach from 1 direction rather than 2
  - Limit the number of cases in relate?
    - Ex. if you know the next "break" in a longest increasing subseq style problem is at most k away, only k cases

(pseudo)polynomial

- Polynomial time: Based on the *size* of the inputs
- Pseudopolynomial time: Based on explicit *values* of constants that define the problem

P/NP

- P: I can solve this in polynomial time
  - Basically every alg we taught you is in P
- EXP: I can solve this in exponential time
- R: I can solve this in finite time
- NP: I can verify the answer in polynomial time
- NP-hard: At least as hard as anything in NP
- NP-complete: NP and NP-hard
  - If P=NP, NP-complete is in P!
- $P \subset EXP \subset R$
- Either $P = NP$ or $P \subset NP$; either $NP = EXP$ or $NP \subset EXP$



TRUE/FALSE

(j) **T** **F** [5 points] The following problem is in NP: given a weighted directed graph $G = (V, E, w)$, vertices $s, t \in G$, and a number ℓ, decide whether there is a **simple** path in G from s to t of weight $\geq \ell$.

(j) T F [5 points] The following problem is in NP: given a weighted directed graph $G = (V, E, w)$, vertices $s, t \in G$, and a number ℓ, decide whether there is a **simple** path in G from s to t of weight $\geq \ell$.

Given a path from s to t, you could check if it's simple and has weight >= l by following the path, checking for duplicate vertices, and keeping track of the path weight in O(n) time.

(j) **T** **F** [3 points] Let G be an undirected graph with arbitrary (possibly negative) edge weights. Assume P $\neq$ NP. True or false: the decision problem of determining whether G has a negative-weight cycle is NP-hard.

(j) T **F** [3 points] Let G be an undirected graph with arbitrary (possibly negative) edge weights. Assume P $\neq$ NP. True or false: the decision problem of determining whether G has a negative-weight cycle is NP-hard.

We can determine if G has a negative weight cycle in polynomial time using Bellman Ford. If P != NP, then Bellman Ford is not NP-hard. (Note: if P=NP, then *technically* Bellman Ford is NP-hard, because NP-hard would just mean polynomial then).

(i) **T** **F** Suppose an algorithm runs in $\Theta(ns)$ time for any input array A of n positive integers where $\max(A) = s$. Then the algorithm runs in polynomial time.

(i) T **F** Suppose an algorithm runs in $\Theta(ns)$ time for any input array A of n positive integers where $\max(A) = s$. Then the algorithm runs in polynomial time.

The factor of s makes it pseudopolynomial

(j) **T** **F** If some NP-complete problem is EXP-hard, then P $\neq$ NP.

(j) T F If some NP-complete problem is EXP-hard, then P $\neq$ NP.

P != EXP, so if NP = EXP, P != NP.

Practice Qs

Optimizing Santa FA12 Final

Saturday, May 16, 2020 8:22 AM

Problem 12. Optimizing Santa [15 points]

Santa is given an ordered sequence of n children $[c_1, \ldots, c_n]$, and an ordered sequence of n toys $[t_1, \ldots, t_n]$. For each $1 \leq i \leq n$ and $1 \leq j \leq n$, the non-negative quantity h_{ij} gives the happiness that child c_i will obtain upon receiving toy t_j. Santa wishes to assign toys to children in order to maximize the total aggregate happiness:

$$H = \sum_{i=1..n} \sum_{j=1..n} x_{ij} h_{ij}$$

where $x_{ij} = 1$ if the child c_i receives the toy t_j, and $x_{ij} = 0$ otherwise. Each child may receive at most one toy, and each toy may be given to at most one child. Note that if a child receives no toy, the child obtains happiness 0.

Unfortunately, due to time constraints, Santa is only allowed to give away at most q toys, where $q \leq n$ is a known positive integer.

Additionally, due to sleigh-packing rules, Santa's toy assignment must obey the following *no-crossing constraint*: for $i < k$, if c_i receives t_j and c_k receives t_ℓ, then it must be that $j < \ell$.

(a) [10 points]

The maximum aggregate happiness H can be computed using dynamic programming. Let $H[i, j, k]$ denote the maximum aggregate happiness that is obtainable if at most k children (where $1 \leq k \leq q$) from the set $\{c_1, \ldots, c_i\}$ are assigned toys from the set $\{t_1, \ldots, t_j\}$, while obeying the *no-crossing constraint*. Write a recurrence relation for $H[i, j, k]$. Include base cases.

S:

Why $H(i,j,k)$?

2 lists, so need pointer for each (i,j)

capacity constraint for the toys, so need to keep track of space left.

For $H(i,j,k)$ → I either assign child i to toy j, or I don't.
If I don't, I either don't assign the child a toy, or I don't assign the toy to a child.

R:

$$H(i,j,k) = \max \begin{cases} h_{ij} + H(i-1, j-1, k-1) & \text{giving toy } j \text{ to kid } i \\ H(i-1, j, k) & \text{kid doesn't get toy} \\ H(i, j-1, k) & \text{toy doesn't get assigned} \end{cases}$$

can't use kid i anymore; can't use toy j anymore; put a toy in bag

T:

Subproblems depend on strictly decreasing $i+j$, so acyclic.
→ both i & j are changing, but neither are guaranteed to change alone.

B: I can have no more kids, no more toys, or no more room in my sleigh

$H(0, j, k) = 0$ (j, k can be anything)
$H(i, 0, k) = 0$ (i, k can be anything)
$H(i, j, 0) = 0$ (i, j can be anything)

O: $H(n, n, q)$ all the kids, toys, & capacity

Use parent pointers to find kid-toy assignment.

T: # subproblems $= O(n) \cdot O(n) \cdot O(q) = O(n^2 q)$

time/subproblem $= O(1)$

original $= O(1)$

total $= O(n^2 q) \cdot O(1) + O(1) = O(n^2 q)$

Palindromes FA14 Final

Saturday, May 16, 2020 8:21 AM

Problem 6. Palindromes [20 points]

Given a string $x = x_1, ..., x_n$, design an efficient algorithm to find the minimum number of characters that need to be inserted to make it a palindrome (recall that a palindrome is a string such as "racecar" the reads the same backwards). Analyze the running time of your algorithm and justify its correctness.

For example, when $x =$ "ab3bd", we need to insert two characters (one "d" and one "a"), to get either the of the palindromes "dab3bad" or "adb3bda".

Similar to edit distance, but on 1 string:

need index for start, index for end.

S: $s(i,j) = \overset{\min}{\#}$ inserts necessary to make substring $x[i:j+1]$ into a palindrome

R: if the end characters match, we can move on.

if we don't match them, we either insert char x_j at the beginning or x_i at the end to get a match.

$$s(i,j) = \min \begin{cases} s(i+1,j-1) & \text{if } x_i = x_j \quad \text{chars match, no inserts} \\ 1 + s(i,j-1) & \text{if } x_i \neq x_j \quad \text{insert } x_j \text{ at beginning, done w/ } x_j \\ 1 + s(i+1,j) & \text{if } x_i \neq x_j \quad \text{insert } x_i \text{ at beginning, done w/ } x_i \end{cases}$$

T: Substring length always gets smaller; substring length $= j - i$

Subproblems depend on strictly decreasing $j-i$, so acyclic

B: if there are only 2 chars left, R is a bit different

$s(i,i+1) = 0$ if $x_i = x_{i+1}$, else 1

$s(i,i) = 0$ ← including in case input is 1 letter.

$S(i,i) = 0$ ← including in case input is 1 letter.

O: $S(1,n)$ (whole string)

T: # subproblems = $O(n) \cdot O(n) = O(n^2)$

work/subproblem = $O(1)$

original problem = $O(1)$

total = $O(n^2) \cdot O(1) + O(1) = O(n^2)$

On Solving Dynamic Programming Problems on the 6.006 Final

Srijon Mukherjee

MIT - Department of Electrical Engineering and Computer Science

General Advice

- Follow SRT BOT format.
- Even if you have a slower solution, write it down. You will get significant partial credit depending on how inefficient it is compared to official solution.
- Carefully read what you want to do: find min/max, count number of possibilities, or check if solution exists.
- Be wary of off by one errors.
- Make sure that you do not switch between prefix and suffix.

Subproblem

- For sequence/set/string based problems (like LIS, LCS, Alternating Coins etc), first try prefixes/suffixes.
- Usually if one works, the other one does as well. Stick to the one you prefer.
- Move on to substrings if you end up recursing on things that are not prefixes/suffixes.
- For graph problem, the vertex is usually part of the subproblem.
- Add additional constraints as appropriate (parameters or boundary constraints) to get the relation to work.
- Clearly define the subproblem:
  - Make sure it is clear what each parameter means (indicate whether prefix/suffix/substring and explain the constraints).
  - State what the value of the subproblem is (including what it should be if there is no solution).
  - Example: $x(i, j, c) =$ The maximum possible tastiness considering the contiguous subarray from indices i to j (both inclusive) while consuming c toppings. $-\infty$ if it is not possible to do so.

Relation

- Make sure that all cases are accounted for.
- Ensure that you only recurse on valid subproblems. For instance, if you recurse on $x(i-1,j)$, add the condition that $i > 0$ (can alternatively handle in base case).
- For prefixes/suffixes/substrings, almost always want to recurse on smaller prefixes/suffixes/substrings.
- Often good idea to consider what can happen to the element(s) at the edge(s).
- Alternatively, think of what happens in one "step".

Topological Order

- Either mention the order in which you solve the subproblems (e.g. Solve in increasing order of $j - i$) or mention what subproblems you recurse on (e.g. The relation only depends on subproblems with smaller $j - i$)
- These are always opposite.
- If you use a parameter (or a function of the parameters), it must be **strictly** increasing/decreasing.
- Might be multiple possibilities. Mentioning any one is fine.
- Example: $x(i, j) = \max(x(i + 1, j - 1), x(i - 1, j + 1))$
  Solution: No order.
- Example: $x(i, j) = \max(x(i + 1, j), x(i, j + 1))$
  Solution: Solve in decreasing $i + j$ or depends on higher $i + j$.
- Example: $x(i, j) = \min(x(i - 1, j), x(i - 1, j + 2))$
  Solution: Solve in increasing i or depends on lower i.
- Example: $x(i, j) = \min(x(i, j - 1), x(i - 1, j + 1))$
  Solution: Solve in increasing $2i + j$ or depends on lower $2i + j$.

Base Case

- Ensure base case will always be reached.
- Check consistency with topological order:
  - Usually the first ones to be solved in topological order.
  - Alternatively, ones you will reach if you keep recursing.
- Full relation does not work for base cases.
- Example: $x(i, j, k) = \max(x(i - 10, j + 2, k), x(i - 9, j + 3, k - 1))$
  Solution: $x(i, j, k) = \text{something}$ when $k = 0$ or $i \leq 9$ or $j > m - 3$ (assuming 0-indexing and taking range of j to be $[0, m]$).

Original Problem

- Usually single subproblem or a combination (usually min, max, sum) of some subset of subproblems.
- Check consistency with topological order:
  - Usually the last ones to be solved in topological order.
  - The one “opposite” to the base case.
- State use of parent pointers if you need to reproduce solution.

Time

- Count number of subproblems and amount of time per subproblem.
- Number of subproblems can usually just be found by multiplying the number of possible values of each parameter.
- Work per subproblem depends on number of cases in relation (for instance, linear if you have to loop over (almost) all values of a parameter).
- Total work usually just the product. Sometimes need to sum the work over all subproblems if the work varies significantly (for example, while analyzing DAG relaxation).
- Calculate size of input (in number of words). Sequences and sets usually contribute size proportional to the number of elements whereas parameters etc are single words.
- Check if runtime polynomial in size (and not just the values in the input).

Problem

- Given a sequence of integers A with the i^{th} element denoted by a_i for $1 \leq i \leq n$, find an increasing subsequence of A of length at least k (for $k \leq n$) with minimum sum.

Problem

- Given a sequence of integers A with the i^{th} element denoted by a_i for $1 \leq i \leq n$, find an increasing subsequence of A of length at least k (for $k \leq n$) with minimum sum.
  S - Use suffixes (prefixes also work) with additional constraints. $x(i, j) =$ Minimum sum of any increasing subsequence starting at i with exactly j elements. ∞ is no such subsequence exists.
  R - What is the next element to be included?
  $x(i, j) = a_i + \min(\{x(l, j-1) | i < l \leq n \text{ and } a_l > a_i\} \cup \{\infty\})$
  T - Solve in decreasing order of i or depends on subproblems with greater i. Alternatively, solve in increasing order of j or depends on subproblems with lower j.
  B - Subproblem undefined when $j < 1$ as it at least includes a_i. Will eventually reach $j = 1$. $x(i, 1) = a_i$
  O - Take minimum of $x(i, j)$ over all i and all $j \geq k$. Use parent pointers from max to get subsequence. No solution if all $x(i, j)$ for $j \geq k$ are ∞.
  T - $O(n^2)$ subproblems and $O(n)$ work per subproblem giving $O(n^3)$ (can do better using AVL trees). Size of input is $\Theta(n)$ and thus this is strongly polynomial.

Bonus Problem

- Given a sequences of opening and closing brackets (either round or square) with the i^{th} bracket having value v_i, find a balanced subsequence with maximum value.

Bonus Problem

- Given a sequences of opening and closing brackets (either round or square) with the i^{th} bracket having value v_i, find a balanced subsequence with maximum value.
  Use substrings and relate by trying to match the first bracket. Topological order is increasing size of substrings. Base case is empty substring whereas original problem is original sequence. Runtime is $O(n^2) \times O(n) = O(n^3)$ which is strongly polynomial.

Another Bonus Problem

- Given a knapsack of size S and n items with positives sizes s_i and values v_i with $V = \sum_{i=1}^{n} v_i$, give an $O(nV)$ algorithm to find the maximum total value you can fit inside the knapsack.

Another Bonus Problem

- Given a knapsack of size S and n items with positives sizes s_i and values v_i with $V = \sum_{i=1}^{n} v_i$, give an $O(nV)$ algorithm to find the maximum total value you can fit inside the knapsack.
  Use $x(i, j) =$ Minimum size needed to store some subset of the first i items with value j.

Good luck!

I will be sticking around for questions.
Otherwise, feel free to come to OH at 2-4 pm and 8-10 pm tomorrow, on Monday, and/or on Tuesday in case you have additional questions.