Introduction to Algorithms
Massachusetts Institute of Technology
Erik Demaine, Ronald L. Rivest, and Nir Shavit

December 17, 2013
6.006 Fall 2013
Final Exam Solutions

Final Exam Solutions

- Do not open this exam booklet until directed to do so. Read all the instructions on this page.
- When the exam begins, write your name on every page of this exam booklet.
- You have 180 minutes to earn 180 points. Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed three 1-page cheat sheets**. No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required unless asked for.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.
- If you need to, make any reasonable assumptions and state those assumptions.
- **Blank answers will receive approximately 25% of the credit for the problem**. If your answer shows no understanding of the problem, you will receive less than 25%. So, if you have no idea how to attack a problem, you should leave it blank.

Problem	Parts	Points	Grade	Grader
1	13	39		
2	1	5		
3	1	12		
4	8	12		
5	2	12		
6	1	16		
7	2	10		
8	1	12		
9	2	17		
10	1	15		
11	3	30		
Total		180		

Name: _____

Circle your recitation:	R01 Gurtej Ashwin **10AM**	R02 Gurtej Ashwin **11AM**	R03 Jennifer Quanquan **12PM**	R04 Jennifer Quanquan **1PM**	R05 Arvind Vlad **2PM**	R06 Arvind Vlad **3PM**	R07 Justin Sherwin **11AM**	R08 Justin Sherwin **12PM**

Problem 1. (13 parts) [39 points] **True/False and Justify**

Circle **T**rue or **F**alse for each statement below, and provide a brief justification for your answer. Your justification is worth more points than your true/false designation.

(a) T F [3 points] If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Solution: True. We have $f(n) \leq r_1 \cdot g(n) + c_1$ and $g(n) \leq r_2 \cdot h(n) + c_2$, which means $f(n) \leq r_1 \cdot (r_2 \cdot h(n) + c_2) + c_1 = r_1 \cdot r_2 \cdot h(n) + c_1 + r_1 \cdot c_2$ and $f(n) = O(h(n))$.

(b) T F [3 points] The Master Theorem can be used to solve the recurrence

$$T(n) = 2\,T(n/2) + \frac{n}{\log n}.$$

Solution: False. This equation does not satisfy any of the cases.

(c) T F [3 points] It is possible to sort any n integers between 0 and n^{100} in $O(n)$ time.

Solution: True. Radix sort with base n.

(d) T F [3 points] Given $n/2$ sorted elements $A[0 : n/2]$ and $n/2$ sorted elements $A[n/2 : n]$ in a single array $A[0 : n]$ (as in merge sort), the ***merge*** algorithm presented in lecture sorts A by moving the elements in A while using $O(1)$ additional space ('in-place').

Solution: False. The merge algorithm presented in lecture uses an additional array of size $O(n)$.

(e) T F [3 points] Quadratic probing (the open addressing scheme using hash/collision function $h(k,i) = h(k) + c_1 i + c_2 i^2$ for constants c_1, c_2) satisfies the ***uniform hashing assumption***: the probe sequence of each key k is equally likely to be any of the $m!$ permutations of the slots $\{0, 1, \ldots, m-1\}$.

Solution: False. All values of k that have the same hash $h(k)$, end up having the same probing sequence which leads to secondary clustering.

(f) T F [3 points] In a hash table maintained with cuckoo hashing, the worst-case running time to search for an element is $O(1)$.

Solution: True. There are only two places needed to be searched.

(g) T F [3 points] Consider a hash table with open addressing and linear probing. Any table resulting from insertions and deletions into an empty table can also be created just by inserting into an empty table.

Solution: False. There is an example of this in problem set 3.

(h) T F [3 points] Given a directed graph $G = (V, E)$, we can (using algorithms presented in 6.006) determine whether G has a directed cycle in $O(V + E)$ time.

Solution: True. Use DFS and look for a back edge.

(i) T F [3 points] Given a weighted directed graph $G = (V, E, w)$ with no negative-weight edges, we can (using algorithms presented in 6.006) compute the shortest-path weight from node s to node t in $O(V + E)$ time.

Solution: False. Since there might be cycles, the best algorithm we can use in this case is Dijkstra's algorithm which runs in $O(V \log V + E)$ when implemented with a Fibonacci heap.

(j) T F [3 points] In a weighted graph $G = (V, E, w)$ with no cycles and all edge weights negative, we can run Dijkstra's algorithm on some graph G' to find the maximum possible weight of a path from node s to node t in G.

Solution: True. Negate the weights in w.

(k) T F [3 points] If you are told that SAT (Boolean satisfiability problem) is NP-complete, and you find a polynomial-time reduction from SAT to Hamiltonian Path, then you can conclude that Hamiltonian path is NP-hard.

Solution: True. If Hamiltonian path were not NP-hard, then the polynomial-time reduction from SAT to Hamiltonian Path would imply that SAT is also not NP-hard, which is a contradiction.

(l) T F [3 points] Given two integers a, b where $a < m$, we can (using algorithms presented in 6.006) compute $a^b \bmod m$ in $O((\lg m)^{\lg 3} \lg b)$ bit operations.

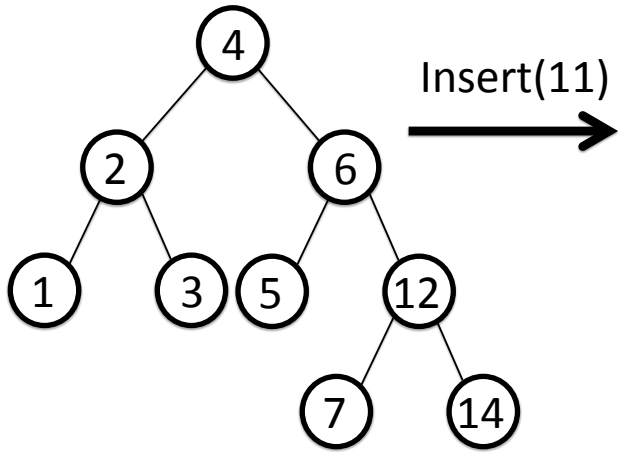
Solution: True. We can use Karatsuba and repeated squaring.

(m) T F [3 points] Given an array A of n nonzero distinct integers, we can compute in $O(n)$ time a ***stable sign sort***, that is, a permutation A' of A such that all negative integers appear before all positive integers, and all integers of the same sign appear in the same order as they did in A.

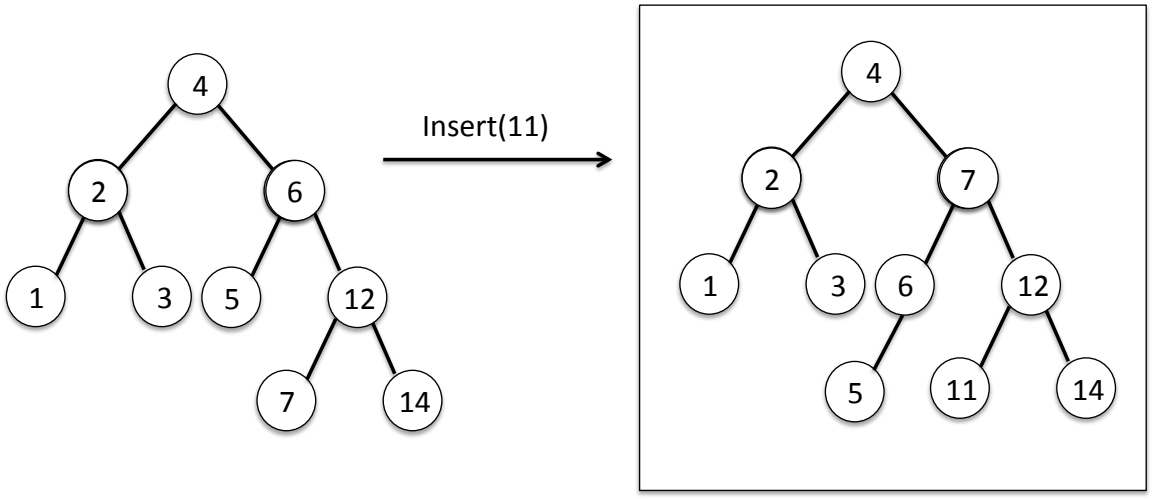
Solution: True. Use a modified counting sort where the sign is the key.

Problem 2. AVL Practice [5 points]

Suppose you have the AVL tree shown below. Draw the AVL tree resulting from inserting 11 into the tree. There is no need to show intermediate work: we will be looking only at the final tree configuration.



Solution:



Problem 3. Dynamic Programming vs. Shortest Path in DAG (6 parts) [12 points]

At one time, Prof. Dynamic thought that every dynamic program could be viewed as finding the shortest path from some node s to some node t in the directed acyclic graph (DAG) where the vertices represent subproblems and edges represent dependencies between subproblems. But this is not true.

Which of the following dynamic programs from lecture can be solved by computing shortest paths in the subproblem dependency DAG (with appropriately weighted edges)? Circle "can" or "cannot" in each case. You do not need to justify your answer.

(a) Fibonacci numbers	[can]	[cannot]
(b) text justification	[can]	cannot
(c) parenthesizing / matrix multiplication	can	[cannot]
(d) edit distance	[can]	cannot
(e) knapsack	[can]	cannot
(f) piano/guitar fingering	[can]	cannot

Most dynamic programs are simply computing shortest paths in the subproblem dependency DAG. The primary exception seen in this class is parenthesizing, which requires computing two subproblem solutions instead of just one, so the resulting structure is more of a tree than a path. Fibonacci numbers also intuitively combine two subproblem solutions, so we accepted the intended "cannot" solution, but with care in setting the edge weights, they can be solved with shortest paths, so we also accepted "can".

Problem 4. Chaining vs. Linear Probing [12 points]

Profs. Sauron and Saruman disagree on how to best organize a hash table storing their records on the n people of Middle Earth. They both construct a table with $m = 2n$ slots, and use the hash function $h(k) = k \bmod m$. Prof. Sauron insists that chaining is a better conflict resolution strategy, while Prof. Saruman insists that linear probing is better.

For each of the following sequences, compute the total asymptotic running time (using Θ notation) for inserting all n items into each of the two hash tables. For chaining, assume that inserting into a chain of length k costs $\Theta(k)$ time (to check for duplicates).

Sequence	**Chaining**	**Linear Probing**
$1, 2, 3, \ldots, n$	$\Theta(n)$	$\Theta(n)$
$m, 2m, 3m, \ldots, n\,m$	$\Theta(n^2)$	$\Theta(n^2)$
$1, 2, \ldots, \frac{n}{2}, m+1, m+2, \ldots, m+\frac{n}{2}$	$\Theta(n)$	$\Theta(n^2)$
$1, m+1, 2, m+2, \ldots, \frac{n}{2}, m+\frac{n}{2}$	$\Theta(n)$	$\Theta(n^2)$

Some students misread the problem, not noticing that they were asked for the *total* running time for n insertions, so their answers were too low by a factor of n. A few points of partial credit were given if all of the answers were correct except for a scale factor of n.

Problem 5. Failure to Launch (2 parts) [12 points]

Suppose you have a hash table with n items in m slots maintained with open addressing with the uniform hashing assumption: the probe sequence of a key is equally likely to be any of the $m!$ permutations of the slots $\{0, 1, \ldots, m-1\}$.

(a) [6 points] If you insert one more element, what is the probability that the first probe fails, i.e., finds an occupied slot?

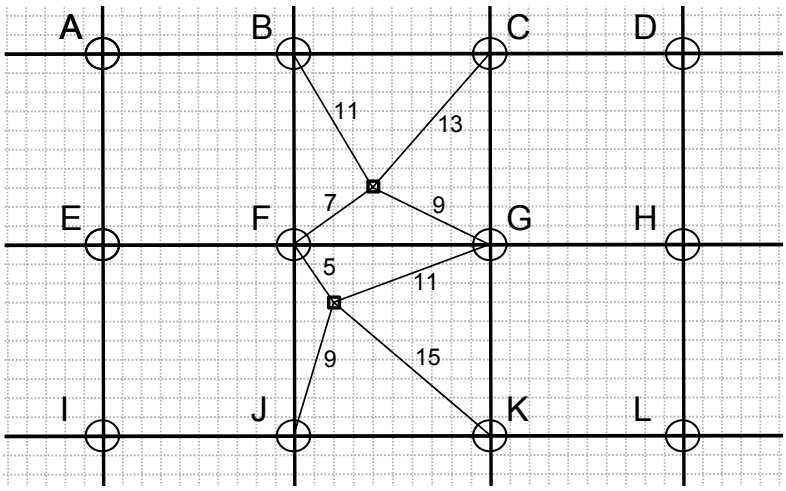
Solution: $\frac{n}{m}$

(b) [6 points] What is the probability that the second probe finds an empty slot **given** that the first probe failed?

Solution: Probability second probe fails given first probe failure is: $1 - \frac{n-1}{m-1} = \frac{m-n}{m-1}$. Some students didn't notice the word "empty" in this second problem part. Some students had an answer that was off by a factor of n/m, since they didn't understand that the question was asking for a conditional probability.

Problem 6. I'm Hungry [16 points]

You are the mayor of the infinitely large city Aleph Town, a 2D plane with infinitely many ***restaurants***: for all integers i, j, there is a restaurant at coordinates $(10\,i, 10\,j)$. In addition, there are n ***houses*** at integer points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Each restaurant $(10\,i, 10\,j)$ serves customers from all houses (x_i, y_i) within Manhattan distance 10: $|10\,i - x_i| + |10\,j - y_i| \leq 10$. Describe and analyze an $O(n)$-time algorithm to find a restaurant serving the most houses.



In the example above, the ovals represent the restaurants in this portion of the grid, the two squares represent houses, and the numbers on edges denote the Manhattan distances between the corresponding house–restaurant pairs. In this example, Restaurant F serves the most houses, as it is the only restaurant serving both houses on the grid.

Solution: We will build a hash table from restaurant coordinates to number of houses served. First, we define a function to get the restaurant coordinates which serve a particular house:

$$\begin{aligned}
&\text{REACHABLERESTAURANTS}(x, y) = \\
&\{\Big(10(\lfloor\frac{x}{10}\rfloor + i), 10(\lfloor\frac{y}{10}\rfloor + j)\Big) \\
&\text{ if } |10(\lfloor\frac{x}{10}\rfloor + i) - x| + |10(\lfloor\frac{y}{10}\rfloor + j) - y| \leq 10, \\
&\text{for } i, j \in \{-1, 0, 1\}\}
\end{aligned}$$

A common mistake here was to forget that a house could exist on top of a restaurant, so we need to actually check 9 locations, rather than just 4. This function will run in $O(1)$ time, since we do a constant number of comparisons.

We can then build our hash table by iterating over all n houses, and for each restaurant coordinate returned by REACHABLERESTAURANTS, either insert it into the hash table with value 1, if it doesn't exist, or find it in the hash table and increment its value. This loop takes (expected) $O(n)$ time, because we perform $O(n)$ hash table operations.

Finally, we iterate through all the restaurants in our hash table, keeping track of the maximum restaurant and returning its coordinates at the end. Because there are at most 5 restaurants reachable from each house, there are at most $5 * n$ keys to iterate through, giving us a runtime of $O(n)$. So, in total our runtime is $O(n)$.

Problem 7. Ternary Search (2 parts) [10 points]

Consider the following algorithm for searching for an element x in a sorted array $A[i:k]$. Instead of breaking the array in half and searching in one of the halves, like binary search, it breaks the array into thirds and searches in one of the thirds. (For clarity, we assume that $n = 3^\ell$ for an integer $\ell = \log_3 n$.)

```
TERNARY-SEARCH(x, A, i, j)
 1  // Assumption: A[i] ≤ x < A[j]
 2  if j − i ≤ 1:
 3      return i
 4  p = (2/3)i + (1/3)j
 5  q = (1/3)i + (2/3)j
 6  if x < A[p]:
 7      return TERNARY-SEARCH(x, A, i, p)
 8  elseif A[p] ≤ x < A[q]:
 9      return TERNARY-SEARCH(x, A, p, q)
10  elseif x ≥ A[q]:
11      return TERNARY-SEARCH(x, A, q, j)
```

(a) [5 points] Write down a recurrence for $T(n)$, the exact number of calls to TERNARY-SEARCH when starting with TERNARY-SEARCH$(x, A, 0, n)$ (counting the initial call).

Solution:

$$\begin{aligned} T(n) &= T(n/3) + 1 \\ T(1) &= 1 \end{aligned}$$

(b) [5 points] Solve for $T(n)$ **exactly**. (For partial credit, write the solution using asymptotic Θ notation.)

Solution:

$$T(n) = 1 + \ell = 1 + \log_3 n$$

The $1+$ accounts for the initial call, i.e., the base case of $T(1) = 1$.

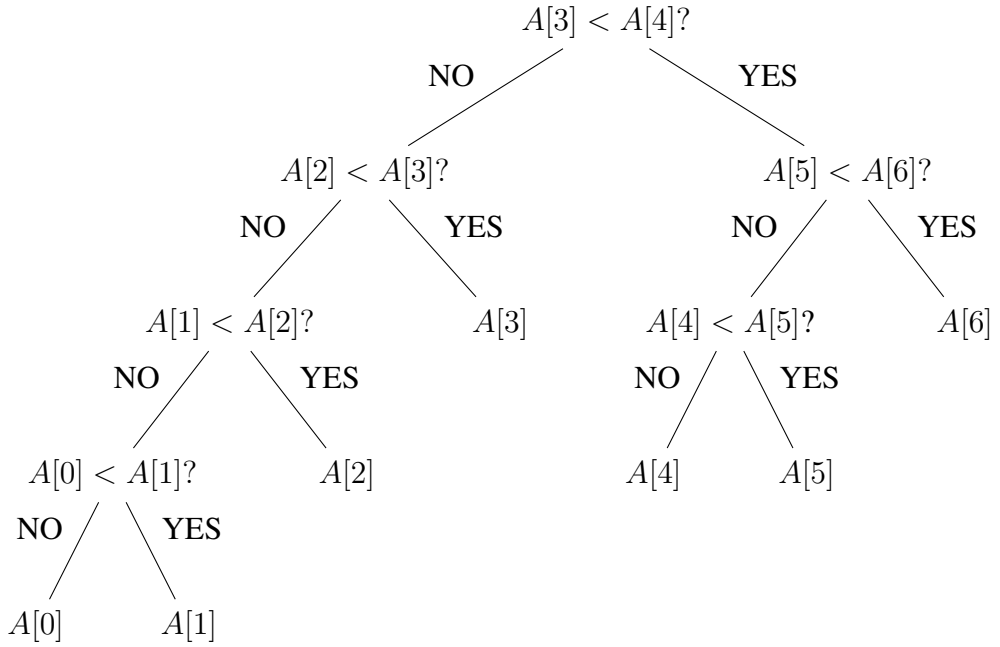
Problem 8. Blind Mountain Climbing with a Transporter [12 points]

Suppose you are given an array A of n distinct numbers $A[0], A[1], \ldots, A[n-1]$, and define $A[-1] = A[n] = -\infty$. Assume you must work in the comparison model, i.e., all you are allowed to do with the numbers is compare them (with $<$, $>$, $=$). Prove that, in the worst case, it takes $\Omega(\lg n)$ comparisons to find a peak, i.e., an index i such that $A[i-1] < A[i] > A[i+1]$.

Solution: The key to this problem is **proving the lower bound in the worst case**. Some students mistook this as showing an algorithm that solves the problem, but that is not what the question is asking.

One correct method is to use a decision tree as stated in lecture. The internal nodes are the binary decisions and the leaves represent the possible outcomes. To receive full credit, students had to demonstrate knowledge of how to set up the decision tree.

There are several ways to construct the tree. One possibility for $n = 7$ made using the divide and conquer 1D peak finding algorithm as taught in lecture is shown below. (Briefly stated, the algorithm asks you to look at the middle element, compare it with its two sides and if it is a peak, return it. Otherwise, recurse on the side that has an element that is bigger than the middle element.)



There are n total leaves in the tree representing the n possible outputs of the peak finding algorithm. The height of any binary tree with n leaves is at least $\lg n$. Therefore, the lower bound of the worst case peak-finding scenario is $\Omega(\lg n)$ since in the worst case, you need to make as many comparisons as the height of the tree. A common mistake in constructing the tree is confusing the decision tree with the recursion tree from the 1D peak finding divide and conquer recurrence equation.

Another common mistake is to argue that using Case 2 of the Master Theorem on the recurrence equation, $T(n) = T(n/2) + \Theta(1)$ gives $T(n) = \Theta(\lg n)$ and by definition, $T(n) = \Omega(\lg n)$. The

problem with this approach is that it assumes that the presented algorithm is the most efficient algorithm that exists for peak finding. One cannot make this assumption in general because we often do not know which algorithm is most efficient until we have proven a lower bound. And so citing the Master Theorem for a particular algorithm does not prove that in the worst case, the number of comparisons is $\Omega(\lg n)$.

Some students used an interesting approach by making the assumption that the number of comparisons in the worst case is $o(\lg n)$ and proving by contradiction that this cannot be true because it violates the lower bound for comparison sorts, $\Omega(n \lg n)$. However, the mistake that students made was assuming that repeatedly running a peak finding algorithm and removing the peak each time while appending this peak to a new list gives a sorted list. This procedure does not return a sorted list for every possible input (e.g. consider $[10, 2, 3, 4, 5]$) because the peak finding problem asks to find a local rather than a global peak. Extra processing needs to be done besides running "find peak" n times. Therefore, the argument does not automatically follow through without proving that this extra processing takes $o(\lg n)$ for each iteration of "find peak."

Problem 9. Binary Search as a DP (2 parts) [17 points]

Prof. Bottoms-Up has discovered a new way to do binary search: dynamic programming! To search for a number x in a sorted array $A[0:n]$, he uses the following dynamic program:

1. **Subproblems:** For all $0 \le i \le j \le n$, $\mathrm{BS}(i,j)$ = the index of x (or its predecessor) in the interval $A[i:j]$.
2. **Guessing:** None.
3. **Recurrence:**

```
BS(i, j)
1  m = ⌊(i+j)/2⌋.
2  if j − i ≤ 1:
3      return i
4  elseif x < A[m]:
5      return BS(i, m)
6  else return BS(m, j)
```

4. **Ordering Loop:**

```
1  for d = 1, 2, ..., n:
2      for i = 0, 1, ..., n − d:
3          solve BS(i, i + d)
```

5. **Original problem:** $\mathrm{BS}(0,n)$

(a) [7 points] If Prof. Bottoms-Up implements this dynamic program in a bottom-up style (expanding the **Ordering Loop** with the **Recurrence**, but replacing recursive calls with table lookups), what would be the resulting asymptotic running time?

Solution: $\Theta(n^2)$. The described bottom-up implementation is

```
1  for d = 1, 2, ..., n:
2      for i = 0, 1, ..., n − d:
3          j = i + d
4          m = ⌊(i+j)/2⌋.
5          if j − i ≤ 1:
6              BS(i, j) = i
7          elseif x < A[m]:
8              BS(i, j) = BS(i, m)
9          else BS(i, j) = BS(m, j)
```

This code has

$$\sum_{d=1}^{n}(n-d+1) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$$

subproblems/iterations (and traverses all of them), and spends $\Theta(1)$ time per subproblem, so the running time is $\Theta(n^2)$.

(b) [10 points] If Prof. Bottoms-Up implements this dynamic program in a top-down recursive style (implementing the **Recurrence** as a recursive algorithm with memoization), what would be the resulting asymptotic running time?

Solution: The recursive algorithm visits only $1 + \lg n$ subproblems, *exactly* as in binary search: each subproblem calls only one other, of half the size. In fact, memoization does nothing here, as no subproblem solution ever needs to be re-used.

So Prof. Bottoms-Up effectively has another way of thinking of (efficient) binary search: dynamic programming!

Problem 10. [15 points] **Must-See T.V.**

Suppose you are given a weighted directed graph $G = (V, E, w)$ with nonnegative edge weights, and you are given a source node $s \in V$ and a destination node $t \in V$. In addition, there are two ***red*** edges e_1, e_2 that *must* be traversed at least once each, although the order doesn't matter.

Give an efficient algorithm to find the weight of a shortest path from s to t that contains both of the red edges (or ∞ if there is no such path).

Solution: This is a graph transformation problem. Call the two red edges $A = (u_A, v_A)$ and $B = (u_B, v_B)$. We create a new graph, G', containing four (unmodified) copies of G: $G_\emptyset, G_A, G_B$ and G_{AB}. The subscript of G indicates which edges we have already traversed. To enforce this property, we add edges from u_A in $G_\emptyset$ to v_A in G_A, u_A in G_B to v_A in G_{AB}, u_B in $G_\emptyset$ to v_B in G_B and u_B in G_A to v_B in G_{AB}. These additions represent copies of edge A going from $G_\emptyset, G_B$ to G_A, G_{AB} respectively, so that the only way to get to a copy of G with A in its subscript is to follow a copy of A, and similarly for B.

Now we can simply run Dijkstra's algorithm on G' with s' being the copy of s in $G_\emptyset$ and t' being the copy of t in G_{AB}. Then the resulting path must traverse both A and B as desired. This approach has running time equal to that of Dijsktra's algorithm (i.e. $O((V + E) \log V)$) and is therefore efficient.

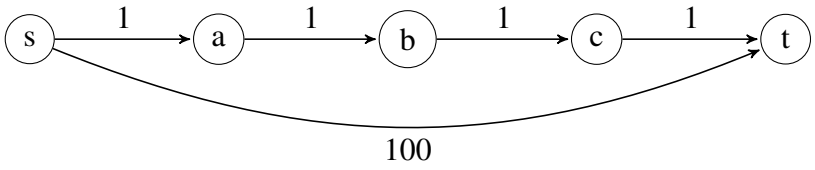
Problem 11. Flying Home for Winter Break (3 parts) [30 points]

You've just acquired the weighted graph $G = (V, E, w)$ representing the airline route network: vertices represent cities, edges represent flight routes, and edge weights represent ticket prices. (Sadly, all edge weights are positive.) You'd like to travel via a directed path from $s =$ Boston $\in V$ to $t =$ Springfield $\in V$ of minimum total price, but you're only willing to take 3 flights. Your goal is to compute, in $O(V + E)$ time, a minimum-cost path from s to t among paths using at most 3 edges. (Assume that there is such a path.)

(a) [10 points] Your first approach is to run 3 iterations of Bellman-Ford. That is, you start with distance estimates $d[v] = \infty$ except for $d[s] = 0$; then you relax all edges in E; then you relax all edges in E; and then you relax all edges in E. Give a counterexample where $d[t]$ won't then contain the weight of a minimum-cost path from s to t using at most 3 edges.

Solution: The relaxation order matters! Consider the following graph, and suppose that we run 3 iterations of Bellman-Ford, first relaxing $s \to a$, then $a \to b$, then $b \to c$, then $c \to t$, and finally $s \to t$ at each iteration. After one iteration, we will have $d[t] = 4$, the cost of the 4-edge path $s \to a \to b \to c \to t$. After two more iterations, we will still have $d[t] = 4$.

Thus $d[t] \neq 100$ after 3 iterations of Bellman-Ford, where 100 is the length of $s \to t$, the minimum-cost path from s to t that uses at most 3 edges.



(b) [10 points] Your second approach is to run BFS to locate all vertices reachable from s by a path of at most 3 edges, remove all other vertices and their incident edges, and then run Dijkstra on the resulting subgraph. Give a counterexample where $d[t]$ won't then contain the weight of a minimum-cost path from s to t using at most 3 edges.

Solution: The couterexample from (a) works here too. Each vertex is reachable from s by a path of at most 3 edges (for instance, c is exactly 3 edges away), so no vertices or edges are removed. Upon running Dijkstra on the resulting subgraph, we will have $d[t] = 4$ again.

(c) [10 points] Give an efficient algorithm to compute the weight of a minimum-cost path from s to t using at most 3 edges. For full credit, your algorithm should run in $O(V + E)$ time.

Solution: We present two solutions: one using dynamic programming, and one using a graph transformation. We then discuss common incorrect approaches and their counterexamples.

First Solution: We may find the weight of a minimum-cost path from s to t using at most 3 edges using a slightly modified version of the DP algorithm for shortest paths. The subproblems are $\delta_k(s, v)$, the weights of the shortest paths from s to v using at most k edges. When computing $\delta_k(s, v)$, we guess the last edge (u, v) in the path. This leads to the following recurrence:

$$\delta_k(s, v) = \min_{(u,v)\in E}\{\delta_{k-1}(s, u) + w(u, v)\},$$

with base case

$$\delta_0(s, s) = 0, \qquad \delta_0(s, v) = \infty \quad (v \neq s).$$

The original problem is $\delta_3(s, t)$.

We now analyze the running time of our DP algorithm. The time taken to compute $\delta_k(s, v)$ given $\delta_{k-1}(s, \cdot)$ is $\Theta(1 + \text{indegree}(v))$. Thus, the time taken to compute $\delta_k(s, v)$ for every vertex $v \in V$ is

$$\sum_{v\in V} \Theta(1 + \text{indegree}(v)) = \Theta(V + E).$$

Finally, the time taken to compute $\delta_1(s, \cdot)$, $\delta_2(s, \cdot)$, and $\delta_3(s, \cdot)$ so that we may know $\delta_3(s, t)$ is $\Theta(3 \cdot (V + E)) = \Theta(V + E)$ as desired.

Note: This is equivalent to running 3 iterations of Bellman-Ford with *synchronous* edge-relaxations – that is, where we use only the $\delta[u]$ of the previous iteration to compute the $\delta[v]$ of the next.

Second Solution: We may find the weight of a minimum-cost path from s to t using at most 3 edges using a graph transformation. Create a graph $G' = (V', E', w')$ as follows: first, create four copies V_0, V_1, V_2, and V_3 of the vertices V. For each weighted edge $(u, v) \in E$, create corresponding weighted edges (u_0, v_1), (u_1, v_2), and (u_2, v_3). In this way, vertices in V_i lead into vertices in V_{i+1}, and each path from V_0 to V_k represents a path in G with exactly k edges. Furthermore, the resulting graph is a DAG with $4V$ vertices and $3E$ edges.

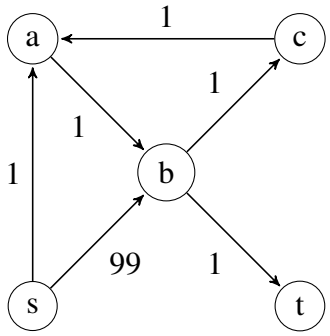
The weight that we seek is

$$\min\{\delta(s_0, t_1), \delta(s_0, t_2), \delta(s_0, t_3)\}.$$

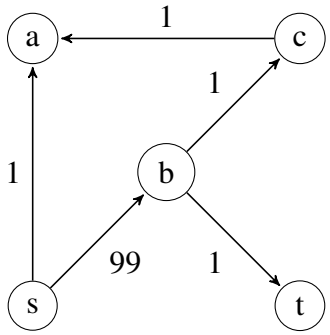
To find the $\delta(s_0, t_i)$, we use SSSP in a DAG. We find a topological sort of G' using DFS, and then relax edges according to this topological sort. This takes time $O(4V + 3E) = O(V + E)$, and the overall running time is again $O(V + E)$ as desired.

Note: We really only need $s_0 \in V_0$ and $t_3 \in V_3$; the other vertices in V_0 and in V_3 are extraneous. Furthermore, the topological sort is quite simple and does not need to be computed.

Incorrect Approach: Running DFS and deleting back-edges to remove cycles does not lead to a correct solution. To see why, consider the following graph.

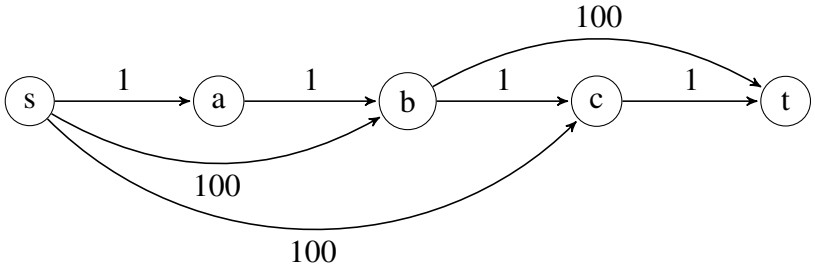


The minimum-cost path from s to t that uses at most 3 edges is $s \to a \to b \to t$, with weight 3. However, if we explore $s \to b \to c \to a \to b$ during DFS, we will delete the back-edge $a \to b$, leading to the following DAG:



Unfortunately, the minimum-cost path from s to t that uses at most 3 edges is now $s \to b \to t$, with weight 100.

Incorrect Approach: Running DFS and removing all *edges* not lying on a path of at most 3 edges from s does not lead to a correct solution. To see why, consider the following graph:



No edges are deleted in BFS, and the shortest path from s to t has 4 edges.