"CS 374" Fall 2015 — Midterm 2 Solutions

1 Short Questions (15 pts)

(a) (5 pts) Give an asymptotically tight solution to the following recurrence.

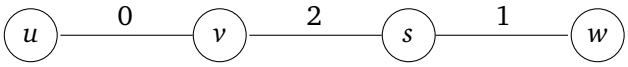
$$T(n) = T(7n/8) + T(n/8) + n \qquad \text{for } n > 8 \text{ and } T(n) = 1 \text{ for } 1 \leq n \leq 8.$$

Solution: $\Theta(n \log n)$. ■

> **Rubric: 5 points** for a correct answer.

(b) (5 pts) Let $G = (V, E)$ be an undirected graph with non-negative edge weights given by $w(e)$, $e \in E$. Consider the following generalization of the MST problem. The input consists of G, a node $s \in V$ and an integer k. The goal is to find a minimum weight tree that contains s and exactly k nodes (including s). Prove via a counter example that Prim's algorithm may fail in general for values of k other than 1 or n. You should specify a small graph with edge weights, s and k and show the optimum solution and the output of running Prim's algorithm when stopped with k nodes.

Solution: Let $k = 3$. Consider the following graph:



Prim's algorithm will take $\{s, w\}$ and then $\{s, v\}$ with weight 3, whereas the optimal solution is $\{s, v\}$ and $\{v, u\}$ with weight 2. ■

> **Rubric:** Out of 5 points, we allocate:
> - **+1 point** for showing a graph with s marked and k specified.
> - **+2 points** for describing the optimum solution.
> - **+2 points** for describing the output for modified Prim's algorithm.

(c) (5 pts) Suppose you are given an array A of n numbers and are told that it has very few distinct numbers, say k. Describe a $O(n \log k)$ algorithm that given A and k decides whether A has at most k distinct numbers or more than k.

Solution: A self-balancing binary search tree (such as Red-Black or AVL) is a data structure where finding whether a number exists and inserting a number can both be done in $O(\log |T|)$ time, where $|T|$ is the number of elements in the tree. We must make sure that the tree size is $O(k)$, so that each operation can be performed in $O(\log k)$ time.

Algorithm:

(1) $T \leftarrow$ a self-balancing BST.

(2) $count \leftarrow 0$.

(3) For $i \leftarrow 1$ to n:

  (a) Find $A[i]$ in T.

  (b) If $A[i]$ is not found, insert $A[i]$ in T, and increment $count$.

  (c) If $count > k$, return "more than k".

(4) Return "at most k".

The algorithm maintains a self-balancing BST that contains at most $k+1$ elements, so finding and inserting takes $O(\log k)$ time. So the overall running time is $O(n \log k)$. ■

Rubric: Out of 5 points, your score will be:
- **5 points max** for an $O(n \log k)$ algorithm.
- **3 points max** for an $O(nk)$ algorithm.
- **1 points max** for an $O(n \log n)$ algorithm.

Your score may go down for several reasons, including but not limited to:
- using inappropriate data structures or claiming incorrect running time for those data structures
- allowing the data structure to become large, so that the running time per operation is $O(\log n)$ or $O(n)$ rather than $O(\log k)$ or $O(k)$. (This is the only exception for having a running time slower than what it could have been.)
- not determining the final answer "more than k" versus "at most k"

2 Incomparable Pairs (15 pts)

Let $G = (V, E)$ be a directed graph. We say that a pair of nodes $u, v \in V$ are incomparable if neither u can reach v nor v can reach u.

(a) (6 pts) Describe a linear time algorithm that given $G = (V, E)$ and a node $u \in V$ finds all nodes $v \in V$ such that u, v are incomparable.

Solution: Here is a linear time algorithm.

Algorithm:

(1) Run a DFS on G from u to find all nodes reachable from u; call this set of nodes X.

(2) Run a DFS on G^{rev} from u to find all nodes reachable from u, i.e., the nodes that can reach u in G. Call this set of nodes Y.

(3) Output $V \setminus (X \cup Y)$.

DFS and constructing G^{rev} can both be done in $O(m+n)$ time. The set union and intersection can be done in $O(n)$ time. Thus the overall running time is $\boldsymbol{O(m+n)}$, or linear in the size of the input. ■

Rubric: Out of 6 points, your score will be:
- **6 points max** for an $O(m+n)$ algorithm.
- **4 points max** for an $O(n^2+m)$ algorithm.
- **3 points max** for an $O(n(m+n))$ algorithm.

Your score may go down for several reasons, including but not limited to:
- not stating which graph WFS is being run in (when more than one is defined)
- not stating where search/DFS/BFS/WFS/explore calls start from
- using $rch(u)$ as an algorithm and/or not stating how to compute it
- deleting vertices between WFS calls

(b) (7 pts) Describe a *linear-time* algorithm that given a DAG G checks whether there is any incomparable pair, and if there is, outputs one of them.

Solution: Here is a linear time algorithm.

First, note that a DAG has an incomparable pair if and only if it does not have a unique topological sort, which is equivalent to it not having a Hamiltonian path.

Algorithm:

(1) Perform a topological sort on G, ordering the vertices from v_1 to v_n.

(2) Scan through the vertices and find the first i such that $(v_i, v_{i+1}) \notin E$.

(3) Output v_i and v_{i+1} or else output "not found."

Topological sort can be done in linear time, then scanning through the vertex list takes linear time. So the overall running time is linear.

Alternatively, the following solution:

Algorithm: Modify Kahn's algorithm for topological sort to check at each step whether there are two or more source vertices. (This algorithm works by finding all sources initially, and then removing one source vertex and each outgoing edge incident to it until every vertex has been reached.) Each pair of source vertices at any step are incomparable, so return the first such pair. If at every step there is exactly one source vertex, return "not found". ■

Rubric: A correct solution gets 7 points. Other solutions are classified as follows:
- ► **5 points max** for doing topological sort but then doing $O(n^2)$ work.
- ► **4 points max** for doing topological sort but then failing to check all adjacent pairs in the order.
- ► **3 points max** for doing topological sort and then searching from a single vertex only.
- ► **3 points max** for finding running solution from part a) on every vertex.
- ► **2 points max** for checking each pair separately, or something similar.
- ► **2 points max** for just checking whether there are multiple sources or sinks.
- ► **1 point max** for algorithms that find pairs that might not be incomparable

Additionally, a solution that mentions topological sort will get at least **1 point**. We will deduct **1 point** for not returning the pair that's been found.

(c) (2 pts) Assuming a linear-time algorithm for DAGs from the previous part describe a linear-time algorithm that given an arbitrary directed graph G checks whether there is any incomparable pair, and if there is, outputs one of them.

Solution: Here is a linear time algorithm.

Algorithm:

(1) Construct G^{SCC}.

(2) Run the algorithm from the previous part on G^{SCC} to obtain two SCCs C_1 and C_2 or "not found."

(3) If found, output some $u \in V(C_1)$ and some $v \in V(C_2)$. Else report "not found" again.

Constructing G^{SCC} can be done in linear time, then the algorithm takes time linear in the size of G^{SCC}. Finally, converting the output takes constant time. So the overall running time is linear. ■

Rubric: Solutions will get **2 points** for using the metagraph and specifying that you can take any pair of vertices from the incomparable SCCs. Solutions will get **1.5 points** for using the metagraph but not saying what to do with the result of part b) on the metagraph. Solutions will get at most **1 point** for anything else.

3 Shortest Walk (10 pts)

Let $G = (V, E)$ be directed graph with non-negative edge lengths where $\ell(e)$ denotes the length of edge $e \in E$. Let $X \subset V$ and $Y \subset V$. Assume that $X \cap Y = \emptyset$. Moreover, assume that s and t are not elements of $X \cup Y$. Describe an efficient algorithm that finds the length of a shortest walk from s to t that visits at least one vertex of X and at least one vertex of Y. Note that the order in which you visit them is not important. A faster algorithm will get you more points.

Solution (Graph Modeling): The idea is as follows:

Create four copies of the graph, $G_{ij} = (V_{ij}, E_{ij})$ for $i \in \{0,1\}$, $j \in \{0,1\}$, where $V_{ij} = \{\langle v, i, j\rangle \mid v \in V\}$ and $E_{ij} = \{(\langle u, i, j\rangle, \langle v, i, j\rangle) \mid (u, v) \in E\}$. Intuitively, $G_{i,j}$ represents having visited *at least* i grocery stores and *at least* j gas stations.

Next, we connect the copies in the following way: If $(u, v) \in E$ and $v \in X$, then we add the edges $(\langle u, 0, j\rangle, \langle u, 1, j\rangle)$ for $j \in \{0,1\}$. Similarly, if $(u, v) \in E$ and $v \in Y$, then we add the edges $(\langle u, i, 0\rangle, \langle u, i, 1\rangle)$ for $i \in \{0,1\}$. Thus, moving from $V_{0,j}$ to $V_{1,j}$ represents visiting a grocery store, and moving from $V_{i,0}$ to $V_{i,1}$ represents visiting a gas station.

The distance from $\langle s, 0, 0\rangle$ to $\langle v, i, j\rangle$ is the length of the shortest walk from s to v that visits *at least* i grocery stores and *at least* j gas stations.

Algorithm:

(1) Construct $\tilde{G} = (\tilde{V}, \tilde{E})$ such that $\tilde{V} = V \times \{0,1\} \times \{0,1\}$ and $\tilde{E} = E_0 \cup E_1 \cup E_2$, where

$$E_0 = \{(\langle u, i, j\rangle, \langle v, i, j\rangle) : (u, v) \in E, i, j \in \{0,1\}\}$$
$$E_1 = \{(\langle u, 0, j\rangle, \langle v, 1, j\rangle) : (u, v) \in E, v \in X, j \in \{0,1\}\}$$
$$E_2 = \{(\langle u, i, 0\rangle, \langle v, i, 1\rangle) : (u, v) \in E, v \in Y, i \in \{0,1\}\}$$

and the length of $(\langle u, i_1, j_1\rangle, \langle v, i_2, j_2\rangle)$ in $\tilde{G}$ is given by $\ell(u, v)$.

(2) Run Dijkstra's algorithm on $\tilde{G}$ from $\langle s, 0, 0\rangle$ to $\langle t, 1, 1\rangle$.

Running time: Note that $|\tilde{V}| = 4|V|$ and $|\tilde{E}| \leq 3|E|$. Step (1) takes $O(4n + 3m) = O(m + n)$ and Step (2) takes $O(4m + 3n \log 3n) = O(m + n \log n)$. So the overall running time is $\boldsymbol{O(m + n \log n)}$. ■

Solution (Combining multiple shortest paths): The idea is to find the distance from s to all grocery stores and the distance from all gas stations to t, and use this information to find the shortest path from s to a grocery store to a gas station to t: we create a vertex s^* with edges (s^*, x) with length $dist(s, x)$ for all $x \in X$, and similarly a vertex t^* with edges (y, t^*) with length $dist(y, t)$ for all $y \in Y$. Then $dist(s^*, t^*)$ will be equal to the length of the shortest s-t walk passing through a grocery store and then a gas station. We can repeat the process for passing through a gas station first, and take the minimum of the two. Formally:

Algorithm:

(1) Run Dijkstra's algorithm on G starting at s and record $dist_G(s, x)$ for all $x \in X$, and $dist_G(s, y)$ for all $y \in Y$.

(2) Run Dijkstra's algorithm on G^{rev} starting at t and record $dist_G(x, t) = dist_{G^{\text{rev}}}(t, x)$ for all $x \in X$, and $dist_G(y, t) = dist_{G^{\text{rev}}}(t, y)$ for all $y \in Y$.

(3) Create $G_X = (V_X, E_X)$ where $V_X = V \cup \{s_X^*, t_X^*\}$ and $E_X = E \cup \{(s_X^*, x) \mid x \in X\} \cup \{(y, t_X^*) \mid y \in Y\}$ where $\ell(s_X^*, x) = dist_G(s, x)$ for all $x \in X$ and $\ell(y, t_X^*) = dist_G(y, t)$ for all $y \in Y$.

(4) Create $G_Y = (V_Y, E_Y)$ where $V_X = V \cup \{s_Y^*, t_Y^*\}$ and $E_Y = E \cup \{(s_Y^*, y) \mid y \in Y\} \cup \{(x, t_Y^*) \mid x \in X\}$ where $\ell(s_Y^*, y) = dist_G(s, y)$ for all $y \in Y$ and $\ell(x, t_Y^*) = dist_G(x, t)$ for all $x \in X$.

(5) $d_X \leftarrow$ the output of Dijkstra's algorithm on G_X from s_X^* to t_X^*.

(6) $d_Y \leftarrow$ the output of Dijkstra's algorithm on G_Y from s_Y^* to t_Y^*.

(7) Output $\min\{d_X, d_Y\}$.

Constructing G^{rev}, G_X, and G_Y all take $O(m)$ time. We run Dijkstra's algorithm a total of four times, so the overall running time is $\boldsymbol{O(m + n \log n)}$. ■

Rubric: Out of 10 points, your score will be:
- **10 points max** for an $O(m + n \log n)$ or $O(m \log n)$ algorithm.
- **8.5 points max** for an $O(mn)$ algorithm.
- **7 points max** for a polynomial time algorithm slower than $O(mn)$.
- **0 points** for a non-polynomial time algorithm.

Your score may go down for the following major errors:
- −5 points if the solution does not visit both an X vertex and a Y vertex.
- −5 points if the walk is not a shortest walk.
- −3 points if the solution visits both an X vertex and a Y vertex, but necessarily visits an X vertex before a Y vertex, or Y before X.

If the solution is otherwise almost correct:
- −1 points if there are minor errors.
- −2 points if there are more significant omissions.

4 Closest Points (10 pts)

Let $P = \{p_1, p_2, \ldots, p_n\}$ be n points on the real line given by their coordinates $x_1, x_2, \ldots, x_n$; these are given in an unsorted array $A[1..n]$. Let p be another point with coordinate z. Given P, p, their coordinates, and an integer k where $1 \leq k \leq n$ describe an efficient algorithm that outputs the k closest points in P to p. Note the distance between p_i and p is given by $|x_i - z|$. For full credit your algorithm should run in $O(n)$ time.

Solution: The idea is to use MoM5Select to find the kth closest point from p in linear time, say p_k. Then we can iterate through the array once to find k closest points by comparing with the distance between p_k and p.

We need to output an array of k closest points, not just distances; thus, for each point in P, we will create a tuple consisting of the point and its distance from p. Then, we will apply our algorithm on top of the array of these tuples.

Algorithm:

(1) Compute $B[1..n]$ where $B[i] = \langle A[i], |A[i] - z| \rangle$. We will use $B[i]_1$ and $B[i]_2$ to denote the first and second element of the tuple $B[i]$, respectively.

(2) Use MoM5Select on B for index k according to the pairs' second element. Let $B[i]$ be the entry returned by MoM5Select.

(3) Partition B such that for all $i \in \{1..n\}$, if $i \leq k$ then $B[i]_2 \leq B[k]_2$; otherwise $B[i]_2 \geq B[k]_2$.

(4) Output all $B[i]_1$ for $1 \leq i \leq k$.

Running Time:

Calculating $B[1..n]$ takes $O(n)$ time, since the calculation of $|A[i] - z|$ takes constant time. MoM5Select takes $O(n)$ time, and partitioning B requires scanning B once, which also takes $O(n)$ time. Finally, outputting $B[i]_1$ for $1 \leq i \leq k$ takes $O(n)$ time since $k \leq n$. Therefore, the overall running time is $O(n)$. ■

Rubric: Out of 10 points, your score will be:
- **10 points max** for an $O(n)$ algorithm
- **7 points max** for an $O(n + k \log n)$ algorithm (based on heap).
- **6 points max** for an $O(n \log n)$ algorithm (based on sorting).
- **4 points max** for an $O(n^2)$ or $O(nk)$ algorithm (variants of brute-force, DP or running k times of MoM5Select).
- **3 points max** for an algorithm based on hash tables
- **0 points** for an incorrect algorithm

Additionally,
- −1 point for minor mistakes (e.g., wrong running time analysis)
- −2 points for major mistakes (e.g., using MoM5Select to "sort").

5 Burgers (10 pts)

The McKing chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where L_i is distance m_i meters from the start of Red street. Assume that the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McKing has collected some data indicating that opening a restaurant at location L_i will yield a profit of p_i independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McKing locations should be D or more meters apart. Describe an algorithm that McKing can use to figure out the maximum profit it can obtain by opening restaurants while satisfying the city's zoning law.

Solution: Use dynamic programming.

$P(i)$ is the maximum profit ignoring the first $i-1$ locations. Each recursive call in $P(i)$ takes the *max* of either selecting the current restaurant and its profit and then making the recursive call to the next available restaurant that is at least D distance away, or skips the current restaurant completely and makes a recursive call to the next restaurant:

$$P(i) = \begin{cases} 0 & i > n \\ \max\{p_i + P(next(i)), P(i+1)\} & \text{otherwise} \end{cases}$$

where $next(i) = \min\{j : m_j \geq m_i + D\}$ is the next available restaurant that is at least D distance away from the current restaurant at i.

Evaluation order: from $i \leftarrow n$ down to 1. The initial call to the recurrence will be $P(1)$.

Side note: *next* can be preprocessed in $O(n)$ time as follows:
(1) $i \leftarrow 1, j \leftarrow 1$
(2) While $i \leq n$:
  (a) While $j \leq n$ and $m_j < m_i + D$, increment j.
  (b) $next(i) \leftarrow j$.
  (c) Increment i.

There are $O(n)$ distinct subproblems. If *next* was preprocessed, each subproblem takes a constant amount of $O(1)$ time, so the running time is $O(n)$. Without the preprocessing step, there will be $O(n)$ work for each sub-problem and the runtime will be $O(n^2)$. ■

Rubric: Standard dynamic programming rubric.

The English description requirement was relaxed and −2 points if the description is missing from a fairly obvious correct solution. No partial credit given if no English description is provided as it is impossible to gauge what the approach was trying to accomplish.

We did not penalize for not preprocessing *next*. However, if without preprocessing *next* the running time is incorrectly reported as $O(n)$, then the students lose 1 point for reporting incorrect running time.

Why does Greedy not work?

Some "obvious" greedy heuristics might be "choose the leftmost feasible location" and "choose the most profitable feasible location". The following is a counterexample:

$$m = [1, 2, 5, 8]$$
$$p = [1, 9, 10, 9]$$
$$D = 5$$

If we use the "leftmost feasible location" heuristic, we would first take L_1 with profit 1, then both L_2 and L_3 become infeasible, so we have to take L_4 with profit 9, for a total profit of 10.

If we use the "most profitable feasible location" heuristic, we would take L_3 for profit 10, and then all locations become infeasible.

However, the optimal solution would be to take L_2 and L_4, for a total profit of 18.