1 Short Questions (10 pts)

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

$T(n) = T(n/4) + T(3n/4) + n^3$ for $n \geq 4$ and $T(n) = 1$ for $n = 1, 2, 3$.

Solution: We can bound $T(n)$ from below by $T_1(n) = 2T(n/4) + n^3$ and from above by $T_2(n) = 2T(3n/4) + n^3$. In the recursion tree of $T_1(n)$, there are 2^ℓ nodes at the ℓ-th level, each with size $(1/4)^\ell n$. Thus,

$$T_1(n) = \sum_\ell ((1/4)^\ell n)^3 2^\ell = n^3 \sum_\ell (2/4^3)^\ell$$

. This is a decreasing geometric series, dominated by its first term n^3. So $T_1(n) = \Theta(n^3)$. Similarly, $T_2(n) = n^3 \sum_\ell (2 \cdot 3^3/4^3)^\ell = \Theta(n^3)$. Therefore, $T(n) = \Theta(n^3)$. ■

Rubric: 5 points. 0 points for an $O(n^3 \log n)$ answer.

Part (b) (5 pts) Let $G = (V, E)$ be a directed graph. Given an edge $e = (u, v)$ describe a linear time algorithm to decide if there is a cycle in G that contains e.

Solution: Edge e is in a cycle iff u is also reachable from v, which can be checked using DFS in linear time. ■

Rubric: Totally 5 points. 3 points for identifying the solution as checking reachability from v to u. 2 points for a linear time algorithm.

2 Removing Outliers (10 pts)

It is common to compute final course grades (for example in CS 374) by ignoring the outliers — the top 5% (who are too smart for their own good) and the bottom 5% (who never show up). More precisely, suppose A is an n element *unsorted* array of total scores where $A[i]$ is the score of person i. Describe an $O(n)$ time algorithm that given A computes the average student score by ignoring the top 5% of scores and the bottom 5% of scores. Assume for simplicity that n is a multiple of 100 and that all numbers in A are distinct. Note that sorting A will easily solve the problem but will take $\Omega(n \log n)$ time.

Solution: Run QUICKSELECT twice to find elements with rank $\frac{5n}{100}$ and rank $\frac{95n}{100}$ in $A[1, \cdots, n]$ in linear time. Denote the two elements selected are A_{High} and A_{Low}, respectively. Then, compare all elements in $A[1, \cdots, n]$ with A_{High} and A_{Low} to identify all elements with rank between $\frac{5n}{100}$ and $\frac{95n}{100}$, then store them into a new array $A'[1, \cdots, m]$, where $m = 0.9n$. Calculate the average of $A'[1, \cdots, m]$.

The pseudocode for the above algorithm is as follows.

```
AvgWithoutOutliers(A[1, ..., n]):
  A_High = QuickSelect(5n/100)
  A_Low = QuickSelect(95n/100)
  for every element A[i]
      if A[i] < A_High and A[i] > A_low
          append A[i] into A'[1, ..., m]
  Avg = Average(A'[1, ..., m])
  return Avg
```

It is easy to see that QUICKSELECT takes $O(n)$ time. Loop and average both take $O(n)$ time. Thus, total runtime is $O(n)$. ■

Rubric: 10 points in total.

- 5 points for correct algorithm.
  Either pseudo code or clear English expression.
- 5 points for correct runtime analysis.
  3 points for corrected $O(n \log n)$ or slower algorithms

3 Burgers

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 at 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 signed a contract with Shampoo-Banana that it will open k restaurants. Opening a restaurant at location L_i costs c_i dollars. To avoid restaurants being too close to each other McKing has decided that no two open restaurants can be less than D meters apart. Describe an algorithm that McKing can use to figure out the minimum cost needed to open k restaurants while satisfying the distance constraints. If no feasible solution exists your algorithm should output ∞.

Solution: Let $F(i,l)$ denote the minimum cost needed to open ℓ restaurants while satisfying the distance constraints using locationa $L_i, \cdots, L_n$.

$$F(i,l) = \begin{cases} 0 & l = 0 \\ \infty & i > n \\ \min\{c_i + F(next(i), l-1), F(i+1,l)\} & \text{otherwise} \end{cases}$$

where $next(i) = \min\{j : m_j \geq m_i + D\}$ is the next available location that is at least D distance away from the current location at L_i. As we have shown in HW5 Q3, all $next(i), \forall i = 1, \cdots, n$ can be calculated in $O(n)$ time.

The initial call to the recurrence will be $F(1,k)$.

The naive way to memoize is to store into a two-dimensional array $F[1, \cdots, n+1; 0, \cdots, k]$. Space for the array is $O(nk)$.

There are $O(nk)$ distinct subproblems. If $next$ array is computed in $O(n)$ time then each call to $next(i)$ takes $O(1)$ time. This requires an extra $O(n)$ space to hold the array. Thus, the total runtime is $O(nk)$ and total space is $O(nk)$.

Without the preprocessing $next$, finding $next(i)$ will incur a time cost of $O(\log n)$ (for binary search) or $O(n)$ (for a linear scan of the array) for each subproblem. The total running time will then be $O(nk \log n)$ or $O(n^2 k)$.

Alternatively, notice that a subproblem in the column indexed by l only depends on subproblems in the same column or the previous column indexed by $l-1$. Therefore, only two columns instead of k columns need to be memoized. Such an approach uses $O(n)$ space. We memoize this in an array F[i] in decreasing values i (from $n+1$ to 1).

■

Rubric: Standard DP rubric. 10 points total. Furthermore:
- **0 points** for solving the problem without using constraint k.
- **5 points max** for no recurrence/pseudo code but correct description and idea presented.
- **8 points max** for a correct but slower solution(scaled accordingly).

4 Minimum weight

Let $G = (V, E)$ be a directed graph. Each vertex $v \in V$ has a weight $w(v)$ associated with it. Given a vertex $s \in V$ let $\alpha(s) = \min\{w(v) \mid s \text{ can reach } v \text{ in } G\}$ be the minimum weight among the weights of all nodes that s can reach in G. In the figure below $\alpha(a) = 3$ and $\alpha(g) = 6$.

- Describe a linear-time algorithm that given a directed acyclic graph (DAG) G and a weight $w(v)$ for each $v \in V$, computes $\alpha(s)$ for *every* $s \in V$.

  Solution: We find a topological ordering of G in linear time. Then we process the vertices, in the reverse topological order. For each vertex v, we compute the following recurrence:

  $$\alpha(v) = \min\{w_v\} \cup \{\alpha(u) \mid (v, u) \in E(G)\}$$

  We will store the values of this function in a one-dimensional array of length $|V(G)|$. The following psuedo-code describes this Algorithm.

```
MinWeightDag(G, w):
  Find a topological ordering T of vertices of G.
  Let α be an array of length |V(G)|.
  For i ← |V(G)| ... 1:
      v = T[i]
      α[v] = w[v]
      For each u ∈ N⁺_G(v):
          If α[v] < w[u]:
              α[v] = w[u]
  return α
```

  Observe that since we process vertices in reverse topological order, the value of α for all children of the vertex v is computed before we process v. To see the correctness of this recurrence, note that if the the vertex with least weight reachable from v is not v, then it is reached from v through one of its children. Since every vertex is processed once and every edge is considered only once, the second phase of the Algorithm runs in linear time as well. ■

  Rubric: 6 points:

  - 5 points for the Algorithm.
    - −4 If α is computed for only one vertex, gets 1 point.
    - −3 If uses one DFS for each vertex, gets 2 points.
  - 1 point for proof of linear runtime.
  - −1 for lack of clarity.
  - −1 for minor errors.
  - Non-linear runtime gets at most 2.

- Extend the algorithm in the previous part to the case of a general directed graph. *Hint:* How would you solve the problem if G is strongly connected?

  Solution: Find G_{SCC} in linear time. Define $w' : V(G_{SCC}) \to \mathbb{R}$ by setting the least weight of vertices in each strongly connected component to the vertex corresponding to that

in G_{SCC}. More precisely, for every strongly connected component C of G, we define $w'(C) = \min\{w_v \mid v \in V(C)\}$. Run the Algorithm of the previous part on G_{SCC}. For each vertex in a strongly connected component, assign the value of α computed for its strongly connected component. The following psuedo-code implements this Algorithm:

```
MinWeightDigraph(G, w):
  Let G_SCC be the strongly connected component meta-graph of G.
  Let w' and α' be two arrays of length |V(G_SCC)|.
  For each component C ∈ V(G_SCC):
      w'[C] = min{w[v] | v ∈ V(C)}
  α' = MinWeightDag(G_SCC, w')
  For C ∈ V(G_SCC):
      For vertex v ∈ V(C):
          α[v] = α'[C]
  return α
```

The runtime of each phase is linear. To see the correctness for this Algorithm, observe that every vertex in a strongly connected component has the same reachability and that if a vertex v can reach a vertex u, then v can also reach any vertex in the strongly connected component containing u. ■

Rubric: 4 points.

- −2 for not correctly generating DAG from the digraph.
- −2 for wrong translation of w and α between G and G_{SCC}.
- −1 for lack of clarity.
- −1 for minor errors.

5 Unique Shortest Path

Let $G = (V, E)$ be a directed graph with non-negative edge lengths; $\ell(e)$ denotes the length of edge e. Given two nodes $s, t \in V$ describe an efficient algorithm that checks whether there is a *unique* shortest path from s to t in G. By uniqueness we mean that there is exactly one path P from s to t whose length is equal to the shortest path distance from s to t. For simplicity assume that s can reach t in G. Slower algorithms get fewer points.

Solution: Given a directed graph $G = (V, E)$ with non-negative edge lengths $\ell(e)$ for every edge $e \in E$, and two nodes $s, t \in V$, let P_{st} be one shortest path from s to t in G. As given in the question, we assume for simplicity that there is at least one (shortest) path from s to t. For any two nodes $u, v \in V$, let $d(u, v)$ be the length of a shortest path from u to v in G.

We will call an edge $(u, v) \in E$ *bad* if

1. (u, v) is not on the path P_{st} and
2. $d(s, u) + \ell((u, v)) + d(v, t) = d(s, t)$.

The first condition states that the edge (u, v) is not used by P_{st}. The second condition states that there is a shortest path from s to t using the edge (u, v).

It is easy to see that P_{st} is the unique shortest path from s to t if and only if there are no bad edges in G. We can detect bad edges as follows. We first find a single shortest st-path P_{st} and compute the distances $d(s, u)$ for every node $u \in V$ by running Dijkstra's algorithm once; note that the nodes and edges of P_{st} can be identified by traversing through the edges of the resulting shortest path tree in reverse from t to s and marking those edges and nodes belonging to the path. We then compute the distances $d(u, t)$, for every node $u \in V$ by running Dijkstra's algorithm on G^{REV}, where G^{REV} is obtained from G by reversing every edge. We now have enough information to detect if a given edge is bad in $O(1)$ time by checking the two conditions above. As mentioned above, the first condition (membership of P_{st}) can be checked in constant time after marking all the edges and/or nodes in P_{st}. The second condition can be checked since we have all the relevant shortest path distance information easily accessible from the two runs of Dijkstra's above. We can therefore loop over all the edges to find any bad edges.

The pseudocode for the above algorithm is as follows.

```
DetectUniquePaths(G, s, t):
  Run Dijkstra's on G to obtain P_st and distances from s
  Get G^REV by reversing G
  Run Dijkstra's on G^REV to obtain distances to t in the original graph G
  Mark every edge on the path P_st
  for every edge (u, v) not on the path P_st:
      if d(s, u) + ℓ((u, v)) + d(v, t) = d(s, t):
          return FALSE
  return TRUE
```

Note that we do not have to check every edge; we can in fact just check the edges that are incident to nodes on P_{st} (why?). But, for simplicity, we will leave out this optimization (the runtime is asymptotically the same).

The two runs of Dijkstra take $O(|E| + |V| \log |V|)$ time each. Reversing the graph takes $O(|E| + |V|)$ time. Marking every edge on P_{st} takes $O(|E|)$ time. The for loop to detect bad edges considers every edge at most once and therefore takes a total of $O(|E|)$ time. Thus, the total runtime of the algorithm is $O(|E| + |V| \log |V|)$.

A simpler but less efficient solution is to find a single shortest st-path P_{st} using Dijkstra's algorithm. Then for every edge e in P_{st} do the following: remove e from G, recompute the shortest path distance using Dijkstra's, and check if it is equal to the original distance. Since we run Dijkstra's at most once for every edge in P_{st}, the runtime for this algorithm is $O(|E|(|E| + |V| \log |V|))$. However, note that the number of edges in P_{st} is upper bounded by the number of nodes in P_{st} minus 1. Thus, the runtime for this algorithm is $O(|V||E| + |V|^2 \log |V|)$. ■

Rubric: 10 points total.

- Score scaled to maximum of 8 points for the $O(|V||E| + |V|^2 \log |V|)$ solution.
  - -1 for runtime analysis of $O(|E|^2 + |E||V| \log |V|)$ for the same algorithm.
- Score scaled to maximum of 5 points for solutions with overall runtime worse than $O(|E|^2 + |E||V| \log |V|)$.
- 3 points for checking the proper distances to identify if a second shortest s-t path exists.
- 4 points for properly looping over all necessary edges to ensure *all* potential shortest paths aside from P_{st} are considered.
- 3 points for proper runtime analysis.