1 Short Questions

Part (a) (5 pts) Give an asymptotically tight solution to the following recurrence. No justification required.

\[ T(n) = T(n/4) + T(3n/4) + n^3 \quad \text{for } n \geq 4 \text{ and } T(n) = 1 \text{ for } n = 1, 2, 3. \]

**Solution:** Building out the recursion tree we find the work at level \( k \) is \( (\frac{7}{16})^k n^3 \). This results in a decreasing geometric series, which converges to \( \Theta(n^3) \). ■

Part (b) (5 pts) Let \( G = (V, E) \) be a directed graph. Given an edge \( e = (u, v) \) describe a linear time algorithm to find the shortest cycle in \( G \) that contains \( e \) or report that there is no cycle containing \( e \). The length of the cycle is simply the number of edges in it.

**Solution:** The shortest directed cycle containing \( e \) consists of \( e \) along with the shortest path \( v \rightarrow u \). Thus to find this shortest cycle (or report that none exists), we simply run breadth-first search from \( v \) to find the shortest path \( v \rightarrow u \) in \( O(V + E) \) time. ■

**Rubric:** 10 points.
- 5 points for part (a): all or nothing.
- 5 points for part (b):
  - 4 points for the algorithm. 2 points for a superlinear algorithm.
  - 1 point for time analysis
  - −1 for a minor error, −2 for a major error.
2 Directed Graphs

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 \( \alpha(s) = \min\{w(v) \mid v \text{ can reach } s \text{ in } G\} \) be the minimum weight among the weights of all nodes that \( s \) can reach in \( G \). In the figure below \( \alpha(a) = 3 \) and \( \alpha(g) = 6 \).

- List the strongly connected components in the example graph and draw the meta-graph of \( G \).

**Solution:** The strongly connected components are \( \{a, b, f\}, \{c\}, \{d, e\}, \{g\}, \{h\}, \{i\} \). The meta-graph is shown below:

- Suppose \( G \) is DAG. Describe a linear-time algorithm that computes \( \alpha(s) \) for every \( s \in V \).

**Solution:** Let \( v_1, \ldots, v_n \) be a topological ordering of the vertices of \( G \). For \( 1 \leq i \leq n \), \( \alpha(v_i) \) satisfies the recurrence

\[
\alpha(v_i) = \min\left\{ w(v_i), \min_{v_i \rightarrow v_j \in E}\{\alpha(v_j)\}\right\}.
\]

Note that when \( v_i \) is a sink, the second term is a \( \min \) over an empty set, which we interpret to be \( \infty \), so no explicit base case is necessary. Since \( \alpha(v_i) \) only depends on \( \alpha(v_j) \) where \( i < j \) (due to the topological ordering), we can memoize this into a one-dimensional array, in order from \( n \) down to 1 (i.e., in reverse topological order). We return the entire memoization structure, since we are asked to compute \( \alpha(s) \) for all \( s \in V \). The running time is \( O(V + E) \).
• Extend the algorithm in the previous part to the case of a general directed graph. If you cannot figure out the previous part, you can use it as a black box in this part.

Solution: Note that for a strongly-connected component \( S \), \( \alpha(s) \) is the same for all \( s \in S \), since the set \( rch(G, s) = \{ v \in V \mid s \text{ can reach } v \} \) is the same for all \( s \in S \). Moreover, for any other vertex \( u \), if \( u \) can reach any vertex of \( S \) than also \( u \) can reach the minimum-weight vertex of \( S \).

Thus we take the strongly-connected component metagraph \( G_{SCC} \), and for each metavertex \( \bar{v} \), we set \( w(\bar{v}) = \min_{u \in \text{comp}(\bar{v})} w(u) \), where \( \text{comp}(\bar{v}) \) is the strongly-connected component associated with \( \bar{v} \). We then run the algorithm for the previous part on \( G_{SCC} \) with these vertex weights, and then for each vertex \( u \) in \( G \), we will set \( \alpha(u) \) to be the \( \alpha \)-value found for the metavertex corresponding to the strongly-connected component containing \( u \).

Forming the metagraph \( G_{SCC} \) and computing its weights takes \( O(V + E) \) time, as does running the linear-time algorithm for the previous part, for a total of \( O(V + E) \) time.

Rubric: 10 points.

• 2 points for the first part: 1 point for the vertices (i.e. the list of strongly connected components), 1 point for the edges.

• 5 points for the second part
  – 2 points max for a super-linear algorithm.
  – Scaled Dynamic programming rubric if appropriate
  – Otherwise, 4 points for the algorithm, 1 point for time analysis. —1 for minor errors, —2 for major errors.

• 3 points for the last part: 2 points for the algorithm, 1 point for time analysis. —1 for minor errors.
3 Many points further away

Let \( P = \{p_1, p_2, \ldots, p_n\} \) be \( n \) points in the 2-dimensional plane where each point \( p_i \) is specified as a tuple \((x_i, y_i)\): \( x_i \) is the \( x \)-coordinate value and \( y_i \) is the \( y \)-coordinate value for \( p_i \). Given two points \( p = (a, b) \) and \( q = (c, d) \) in the plane, the \( L_2 \) distance (also called the Euclidean distance) between \( p \) and \( q \) is defined as \( \sqrt{|a - c|^2 + |b - d|^2} \) (note that distance is always non-negative). Given the set \( P \) of \( n \) points and a point \( q = (a, b) \), describe an \( O(n) \) time algorithm to find a point \( p_i \in P \) such that there are at least \( n/5 \) points in \( P \) that are at least as far away from \( q \) as \( p_i \). An \( O(n \log n) \) time algorithm will get you half the points (you may want to think about the slower algorithm first).

**Solution (If it looks like Selection, and quacks like Selection...):** We need to return a point in \( P \) whose distance from \( q \) is at most the \( n/5 \)-th largest (i.e., \( 4n/5 \)-th smallest) distance from \( q \). We do this by computing all of the distances from \( q \) to points in \( P \), letting \( r \) be the distance of rank \( 4n/5 \), and returning some point whose distance from \( q \) is smaller than \( r \). Note that since distances are non-negative, it is equivalent to work with squared distances to avoid needing to take square roots.

```python
THRESHOLDPOINT(P[1..n], q = (a, b)):
    allocate array A of length n
    for i ← 1 to n
        A[i] ← (x_i - a)^2 + (y_i - b)^2
    r ← SELECT(A[1..n], 4n/5)
    for i ← 1 to n
        if A[i] ≤ r
            return p_i
```

Using a linear-time selection algorithm (as shown in lecture), the algorithm takes \( O(n) \) **time**. □

**Solution (...there might be a cleverer solution):** The closest point to \( q \) trivially satisfies the requirement. The following algorithm scans through the list and finds this element in \( O(n) \) **time**:

```python
CLOSESTPOINT(P[1..n], q = (a, b)):
    minDist ← ∞
    closest ← Null
    for i ← 1 to n
        dist ← (x_i - a)^2 + (y_i - b)^2
        if dist < minDist
            minDist ← dist
            closest ← p_i
    return closest
```

□

**Rubric:** 10 points.

- 8 points for the algorithm:
  - 5 points max for an \( O(n \log n) \) algorithm.
  - 3 points max for a correct \( \omega(n \log n) \) algorithm.
  - −1 for minor errors, −2 for major errors.
- 2 points for the time analysis.
4 Division

Let \(x\) and \(y\) be two positive integers with at most \(n\) digits each. Let \(x \div y\) denote the maximum integer \(k\) such that \(yk \leq x\). For instance \(341873418723478137 \div 234334234324747 = 1458\). Describe an algorithm to compute \(x \div y\) in time polynomial in \(n\) and justify its running time. You can assume that adding and subtracting (and hence comparison) of \(n\) digit numbers takes \(O(n)\) time and that multiplication of two \(n\) digit numbers takes \(M(n)\) time where \(M(n)\) is at most \(O(n^2)\). Your running time can be expressed in terms of \(M(n)\). For this problem you should assume that single digit operations take \(O(1)\) time but not general arithmetic operations; if you use such operations the time should be accounted for. You can assume that \(x\) and \(y\) are in binary if it helps you.

**Solution:** Note that \(1 \leq x \div y \leq x\), so we can binary search over this space for the correct number. To divide by two, we use a bit-shift operation \(\text{shift}\), that can be implemented to run in \(O(n)\) time by simply copying bits over one by one. (We thus assume that \(x\) and \(y\) are given in binary.)

\[
\text{DivideViaDivideAndConquer}(x, y):
\]

\[
\begin{align*}
\ell &\leftarrow 1 \\
r &\leftarrow x \\
\text{while} \ \ell < r \\
\quad \text{mid} &\leftarrow \text{shift}(r + \ell) \left\ll \lfloor \frac{r + \ell}{2} \rfloor \right) \\
\quad \text{if} \ \text{mid} \cdot y > x \\
\quad \quad r &\leftarrow \text{mid} - 1 \\
\quad \text{else} \ \text{mid} \cdot y \leq x \\
\quad \quad \ell &\leftarrow \text{mid} \\
\text{return} \ \ell 
\end{align*}
\]

In each iteration of the while loop, we perform at most two additions (\(O(n)\) each), one shift (\(O(n)\)), one multiplication (\(M(n)\)), and one comparison (\(O(n)\)), for a total of \(O(n + M(n))\) time per iteration. The number of iterations is \(O(\log_2(x)) = O(n)\), for a total running time of \(O(n^2 + nM(n))\).

**Rubric:** 10 points.

- 8 points for the algorithm:
  - 6 points max for \(\omega(n^2 + nM(n))\) time algorithm.
  - 2 points max for a algorithm that is exponential in \(n\) (e.g., “linear” scan)
  - −1 for a minor errors, −2 for major errors. A small number of off-by-ones collectively count as one minor error.
- 2 points for the time analysis.
5 Fire Stations

A long straight country road can be modeled as a line starting at 0. The road has \( n \) houses at locations \( x_1 < x_2 < \ldots < x_n \) on the line. The city wants to build fire stations along the road such that every house is within distance \( D \) from some fire station. Fire stations cannot be built at arbitrary locations. The city has figured \( m \) potential locations on the road at \( y_1 < y_2 < \ldots < y_m \). For simplicity assume that all the \( x \) and \( y \) values are distinct. The cost of building a fire station at location \( y_j \) is \( c_j \). Describe an efficient algorithm that minimizes the total cost of building the fire stations with the constraint that each house is within a distance \( D \) of some fire station. Note that it is relatively easy to check whether a feasible solution exists.

Solution:

- We will add sentinel fire stations at locations \( y_0 < x_1 \) and \( y_{m+1} = \infty \), each of cost \( \infty \), and a sentinel house at location \( x_{n+1} = \infty \). Let \( \text{MinFire}(i, j) \) denote the minimum cost to cover the houses at locations \( x_j, \ldots, x_n \) using potential fire stations at locations \( y_i, \ldots, y_m \).
- We want to return \( \text{MinFire}(0, 1) \).
- For \( 0 \leq i \leq m \), let \( \text{Right}(i) = \min \{ j \mid x_j > y_i + D \} \) be the index of the first house to the right of \( y_i \) that is not covered by \( y_i \).

Then \( \text{MinFire} \) satisfies the recurrence

\[
\text{MinFire}(i, j) = \begin{cases} 
0 & \text{if } j > n \\
\infty & \text{if } x_j < y_i - D \\
\min \left\{ c_i + \text{MinFire}(i+1, \text{Right}(i)), \text{MinFire}(i+1, j) \right\} & \text{otherwise}
\end{cases}
\]

(These cases correspond to: not needing to cover any more houses; the \( j \)-th house cannot be covered with the remaining stations; and choosing whether or not to open the station at \( y_i \).)

- \( \text{MinFire}(i, j) \) depends on entries of the form \( (i+1, k) \). We memoize into a two-dimensional array, for \( i \) in descending order in the outer loop and for \( j \) in any order in the inner loop.

- \( \text{Right}(i) \) can be precomputed for \( 0 \leq i \leq m \) as follows:

\[
\text{KNOWYOURRIGHTS}(x, y):
\]

\[
j \leftarrow 1 \\
\text{for } i \leftarrow 0 \text{ to } m \\
\quad \text{while } x_j < y_i + D \\
\quad \quad j \leftarrow j + 1 \\
\quad \text{Right}[i] \leftarrow j \\
\text{return Right}
\]

The running time of the precomputation is \( O(m + n) \) since within the nested loops, \( j \) cannot be incremented more than \( n \) times.

With this precomputation filling each entry of the memoization structure takes \( O(1) \) time, for a total of \( O(mn) \) time.

For completeness, we include the iterative pseudocode on the next page.
This Burning Desire \((x, y, c)\):
\[
\begin{align*}
\text{Right} & \gets \text{KnowYourRights}(x, y) \\
\text{for } i & \gets m + 1 \text{ down to } 0 \\
\text{for } j & \gets n + 1 \text{ down to } 1 \\
& \quad \text{if } j > n \\
& \qquad \text{MinFire}[i, j] \gets 0 \\
& \quad \text{else if } x_j < y_i - D \\
& \qquad \text{MinFire}[i, j] \gets \infty \\
& \quad \text{else} \\
& \qquad \text{MinFire}[i, j] \gets \min \{c_i + \text{MinFire}[i + 1, \text{Right}[i]], \text{MinFire}[i + 1, j]\}
\end{align*}
\]
return \(\text{MinFire}[0, 1]\)

**Rubric:** 10 points. Standard Dynamic programming rubric. No penalty for slower polynomial time algorithms.
6 Shortest Paths

Let $G = (V,E)$ be a directed graph with non-negative edge lengths; $\ell(e)$ denotes the length of edge $e$. Suppose you have computed the shortest path distance from $s$ to $t$ and are not too happy about it. You have the ability to add one edge to the graph $G$ to reduce the shortest path distance but you have to choose this edge from a given list of edges $E' = \{e_1 = (u_1,v_1), e_2 = (u_2,v_2), \ldots, e_k = (u_k,v_k)\}$ where each of these edges also has its length $\ell(e_i)$ specified to you. Design an algorithm that finds the best edge to add so that the resulting graph has the smallest shortest path distance from each of these edges also has its length $\ell(e_i)$ specified to you. 

Solution (Graph Modeling): We will create two copies of the graph, one representing not having taken one of the edges in $E'$, and one representing having taken of them.

- $V^* := V \times \{\text{No}, \text{Yes}\}$
- $E^* := \{(u,a) \to (v,a) \mid u \to v \in E, a \in \{\text{No}, \text{Yes}\}\} \cup \{(u,\text{No}) \to (v,\text{Yes}) \mid u \to v \in E'\}$. For all edges, $\ell^*((u,a) \to (v,b)) = \ell(u \to v)$.
- We need to find the shortest path from $(s,\text{No})$ to $(t,\text{Yes})$, and extract the edge in $E^*$ corresponding to some $e_i \in E'$ that this path passes through.
- Since all edge lengths are non-negative, we will run Dijkstra’s algorithm from $(s,\text{No})$.
- Building $G^* = (V^*,E^*)$ by brute force takes $O(V^* + E^*) = O(V + E + k)$ time, and running Dijkstra’s algorithm on $G^*$ takes $O(E^* + V^* \log V^*) = O(k + E + V \log V)$ time. ■

Solution (Combining Shortest Paths): For each $e_i = u_i \to v_i \in E'$, the length of the shortest path going through $e_i$ has length $\text{dist}(s,u_i) + \ell(e_i) + \text{dist}(v_i,t)$. Thus if we compute $\text{dist}(s,v)$ and $\text{dist}(v,t)$ for all $v \in V$, we can then iterate over $E'$ to find the edge that minimizes the relevant length.

\[\text{WrinkleInTime}(G, E'):\]

\begin{itemize}
  \item compute $\text{dist}(s,u)$ for all $u$ via Dijkstra’s algorithm
  \item compute $\text{dist}(v,t)$ for all $v$ via Dijkstra’s algorithm on $G^\text{rev}$
  \item return $\arg\min_{e \in E'} \{\text{dist}(s,u_i) + \ell(e_i) + \text{dist}(v_i,t)\}$
\end{itemize}

Finding the correct edge takes $O(k)$ time, so overall the algorithm takes $O(k + E + V \log V)$ time. ■

Rubric: 10 points.

- No penalty for citing Dijkstra’s algorithm as $O(E \log V)$ (and thus getting $O(k \log V)$ plus Dijkstra’s).
- 8 points max for an $O(kE \log V)$ time algorithm (e.g., overly large graph construction if using graph reduction, or making $O(k)$ calls to Dijkstra’s algorithm if combining multiple shortest path computations).
- 6 points max for a polynomial time algorithm slower than $O(kE \log V)$.
- For a graph reduction solution, use the Standard graph reduction rubric.
- For a combining multiple shortest path computations solution:
  - 4 points for finding the relevant shortest paths computations
  - 4 points for combining them in the correct ways
  - 2 point for time analysis in terms of the input parameters
  - $-1$ for each minor error, $-2$ for each major error