CS 180 Homework 4 Solutions

Question 1

You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

a. Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke `status_check`, subject to the requirement that `status_check` is invoked at least once during each sensitive process P.

Solution:

Algorithm:
- Run the greedy interval scheduling algorithm.
- For each selected interval, schedule a `status_check` at its finish time.

Runtime Analysis: Greedy interval scheduling takes $O(n \log n)$ time, and looping through to schedule `status_checks` will take $O(n)$ time since there's at most n intervals, totaling $O(n \log n)$ time.

Proof of Correctness:
- Let S be the set of non-overlapping intervals selected by interval scheduling.
- First, we prove that our algorithm is correct.
  - Suppose, for contradiction, there exists some process x that doesn't contain a `status_check` after running our algorithm.
  - Note that $x \notin S$ otherwise a `status_check` would've been scheduled for it.
  - x must overlap with at least one interval in S, otherwise $S \cup \{x\}$ would be a larger set of non-overlapping intervals, which violates the optimality of our greedy interval scheduling algorithm proven in lecture.
  - Let y be the interval in S with the earliest finish time that overlaps with x.
  - x must start before y finishes, otherwise they wouldn't overlap.
  - x must finish after y finishes, otherwise our greedy interval scheduling algorithm would've considered x first, and selected it over y (x couldn't have been eliminated already since y is the first overlapping interval).
  - This means that x will contain the `status_check` at the finish time of y.

- Thus, we have a contradiction, so there doesn't exist a process without a `status_check` within it.

- Next, we prove that our algorithm is optimal.
  - Suppose, for contradiction, there exists a smaller set of `status_check` times that still covers every single process (meaning its size is less than $|S|$).
  - This means that there must exist a `status_check` at some time t that covers at least two intervals in S. Call these intervals a and b.
  - a and b must overlap, since they both cover time t.
  - However, a and b are both in S, which consists of non-overlapping intervals.
  - Thus, we have a contradiction, so no smaller set of `status_checks` exists.

b. While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of k sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke `status_check` at least k times: no one invocation of `status_check` can handle more than one of these processes.

   This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that k^* is the largest value of k such that one can find a set of k sensitive processes with no two ever running at the same time. Is it the case that there must be a set of k^* times at which you can run `status_check` so that some invocation occurs during the execution of each sensitive process? (In other words, the kind of argument in the previous paragraph is really the only thing forcing you to need a lot of invocations of `status_check`.) Decide whether you think this claim is true or false, and give a proof or a counterexample.

Solution: This claim is true. Our algorithm schedules exactly one `status_check` for each non-overlapping interval returned by the interval scheduling algorithm, which is guaranteed to return the maximum number of non-overlapping intervals. This is exactly what k^* is defined as.

Question 2

Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They've researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays. They've found an excellent travel website that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the website answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The website guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + l_e$, where l_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the website to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the website are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the website (based on a specific edge e and a time t) as taking a single computational step.

Solution:

Algorithm: This can be solved by slightly modifying Dijkstra's.

- Let the start vertex be s and the destination vertex be d
- Initialize a map $\textsc{Dist}(s) = 0$ and $\textsc{Dist}(v) = \infty$ for all vertices $v \neq s$
- Initialize a map $\textsc{Pred}(v) = \text{null}$ for all vertices v
- Initialize a priority queue of vertices by distance
- While there are unfinalized nodes:
  - Pop the closest node from the priority queue; call this node u
  - Let t, the current time, be $\textsc{Dist}(u)$
  - For all neighbors v of u:
    - Let t' be $f_e(t)$, where e is the uv edge
    - If $t' < \textsc{Dist}(v)$: Set $\textsc{Dist}(v) = t'$ and $\textsc{Pred}(v) = u$, and update v's distance in the priority queue accordingly
  - Mark u as finalized
- Follow the linked list of predecessors back from d until reaching s
- Reverse this order and return it as the shortest path

Runtime Analysis: Every single edge can cause one binary heap update, and other operations in the loop are $O(1)$, leading to a total runtime of $O(e \log e)$.

Proof of Correctness:

- First, we observe that since $f_e(t)$ is monotonically increasing, it's never optimal to wait before taking a path, and since $f_e(t) \geq t$ for all edges e and times t, it's never optimal to return to a vertex.
- We prove inductively that after finalizing n vertices, our algorithm has found the correct shortest paths to all finalized vertices.
- **Base Case:** For $n = 1$, we always finalize the start node, which must have distance 0 from itself. Thus, our algorithm is correct.
- **Inductive Hypothesis:** We assume that for $n = k$, our algorithm has correctly found the shortest path to the first k vertices to be finalized.
- **Inductive Case:** We want to show that for iteration $n = k + 1$, our algorithm is still correct for all finalized vertices. Since our inductive hypothesis assumes the first k vertices to be finalized are correct, we just need to show our algorithm is correct for vertex $k + 1$.
  - Let v be the vertex that our algorithm chooses to finalize next.
  - Suppose, for contradiction, the path our algorithm chose isn't the shortest and the shortest path to v goes through some other non-finalized vertex.
  - Let x be the first unfinalized vertex that this path goes through, and let t' be our arrival time at x if we follow this path.
  - We observe that $t' = \textsc{Dist}(x)$, since $\textsc{Dist}(x)$ is the shortest distance to x via a path through only finalized nodes (because of our inductive hypothesis), and t' is the arrival time at x via one such path.
  - We also observe that $\textsc{Dist}(x) \geq \textsc{Dist}(v)$, otherwise our algorithm would've chosen to finalize x before v.
  - Thus, $t' \geq \textsc{Dist}(v)$, and since $f_e(t) \geq t$ for all e and t, by following the rest of the edges along the path means the final arrival time will be at least as much as $\textsc{Dist}(v)$ as well.
  - This means that our algorithm's path is at least as short, which is a contradiction.
  - Thus, our algorithm found the shortest path to vertex $k + 1$.
- We've proven that our algorithm finds the shortest path to all nodes, so it must find the shortest path to our one specific destination node as well.

Question 3

You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, ..., n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

They've written the following simple program to compute F_j for all j:

```
For j = 1, 2, ..., n
  Initialize F_j to 0
  For i = 1, 2, ..., n
    If i < j then
      Add Cq_iq_j / (j-i)^2 to F_j
    Else if i > j then
      Add -Cq_iq_j / (j-i)^2 to F_j
    Endif
  Endfor
  Output F_j
Endfor
```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Solution:

Algorithm:

- Construct an array $K = \left[-\frac{1}{(n-1)^2}, ..., -\frac{1}{4}, -1, 0, 1, \frac{1}{4}, ..., \frac{1}{(n-1)^2}\right]$
- Construct an array $Q = [q_1, q_2, ..., q_n]$
- Compute the convolution $V = K * Q$ (only keeping elements where K and Q fully overlap)
- Compute a new array $V'[i] = C \cdot V[i] \cdot Q[i]$ for all $i \in \{1, ..., n\}$
- Return V'

Runtime Analysis: K has $2n - 1$ elements, and each one can be computed in $O(1)$ time, so constructing K takes $O(n)$ time. Computing $V = K * Q$ will take $O(n \log n)$ when using an FFT-based convolution. Elementwise multiplication will take $O(n)$ time. This totals $O(n \log n)$ time.

Proof of Correctness:

- Let i be the index (one-indexed) of the element we're computing in V. Note that q_i will be multiplied by 0 in the convolution.
- For all $j < i$, q_j will be multiplied by elements in K to the right of the 0, which is of the form $\frac{1}{(j-i)^2}$ because of how K was constructed.
- For all $j > i$, q_j will be multiplied by elements in K to the left of the 0, which is of the form $-\frac{1}{(j-i)^2}$ because of how K was constructed.
- This means that $V_i = \sum_{i<j} \frac{q_j}{(j-i)^2} - \sum_{i>j} \frac{q_j}{(j-i)^2}$
- Because of the elementwise multiplication, we have that $V_i' = Cq_iV_i$
- Expanding this, we get $V_i' = V_i = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$
- This is exactly what we wanted to compute.

Question 4

Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

Solution:

Algorithm:

- Initialize a variable `cur` to the root node
- Repeatedly:
  - If `cur` is a leaf node, return `cur`
  - Otherwise, probe `cur` and its 2 children
  - If the children are both greater than `cur`, return `cur`
  - Otherwise, set `cur` to a child less than it (if both children are less then arbitrarily pick one)

Runtime Analysis: The height of a complete binary tree is $\lceil \log n \rceil$. This algorithm can at most probe along a path from the root to a leaf and the siblings of the nodes along that path, so it will probe at most $2\lceil \log n \rceil$ nodes, resulting in $O(\log n)$ probes.

Proof of Correctness:

- We claim that, during all iterations, `cur` is less than its parent if it has one.
- We prove this via casework:
  - **Case 1:** `cur` hasn't been updated after initialization. Our claim is true since `cur` has no parent due to being the root node.
  - **Case 2:** `cur` was updated during the loop. We only set `cur` to a child that's less than its parent, so the new `cur` must also be less than its parent.
- If `cur` is a leaf node, since we've proven that it's has smaller x_v than its parent, and it has no children, it must be a local minimum.
- If `cur` is less than both its children, since we've proven that it's less than its parent, it must be a local minimum.
- We've now proven that if our algorithm terminates, then it must have correctly returned a local minimum.
- Our algorithm will also always terminate, since it must find a leaf node eventually due to only being able to traverse away from the root node.
- Thus, our algorithm is always correct.

Question 5

Find an algorithm to solve the following problem:

You are given a 2D grid `image`[][], where each `image`$[i][j]$ represents the color of a pixel in the image. Also provided is a zero-indexed coordinate (s_r, s_c) representing the starting pixel (row and column) and a new color value `newColor`.

Your task is to perform a flood fill starting from the pixel (s_r, s_c), changing its color and the color of all connected pixels that have the same original color. Two pixels are considered connected if they are adjacent horizontally or vertically (not diagonally) and have the same original color.

Example:

Input:

`image` $= [[1,1,1,0],$
$[0,1,1,1],$
$[1,0,1,1]]$
$(s_r, s_c) = (1,2)$
`newColor` $= 2$

Output:

$[[2,2,2,0],$
$[0,2,2,2],$
$[1,0,2,2]]$

Solution:

Algorithm:

- Let x be the color of the pixel at (s_r, s_c)
- Construct a graph where the vertices are every pixel with color x and the edges connect every adjacent pixel
- Run BFS starting at (s_r, s_c) and track every visited vertex
- Update all visited vertices to `newColor`

Runtime Analysis: If the grid is $n \times m$ there are at most nm vertices and $4nm$ edges, so graph construction and BFS will run in $O(nm)$. Recoloring the visited vertices will also take $O(nm)$. Thus, the total runtime is $O(nm)$.

Proof of Correctness:

- We claim that the connected component containing the pixel at (s_r, s_c) is exactly the set of connected pixels that we want to replace.
  - Since only pixels with color x are vertices, there cannot be a pixel of the wrong color in the component.
  - A patch of same-color pixels won't ever need to cross a pixel of a different color, so ignoring pixels of different colors is fine.

    - Two pixels are connected iff you can travel between them with only horizontal and vertical moves, which is exactly what the edges in our graph are.
- Since we know from lecture that BFS will find every vertex in the connected component and nothing more, our algorithm will find the correct set of pixels to replace.

Question 6

Find an algorithm to solve the following problem:

Given an $n \times n$ matrix, where every row and column is sorted in increasing order and a value x, decide whether x is in the matrix.

Example:

Input:
$x = 62, \texttt{mat}[][] = [[3, 30, 38],$
$[20, 52, 54],$
$[35, 60, 69]]$

Output: `false`

Explanation: 62 is not present in the matrix.

Input:
$x = 30, \texttt{mat}[][] = [[3, 30],$
$[20, 52]]$

Output: `true`

Explanation: $\texttt{mat}[0][1]$ is equal to 30.

Solution:

Algorithm:
- Initialize a row and column (r, c) to the top-right corner of the matrix
- While (r, c) is still in the bounds of the matrix:
  - If $x = \texttt{mat}[r][c]$: Return `true`
  - If $x > \texttt{mat}[r][c]$: Move r down one row
  - If $x < \texttt{mat}[r][c]$: Move c left one column
- Return `false`

Runtime Analysis: Since r and c can only move in one direction, they can only move n times at most before exiting the bounds of the matrix, meaning the loop can run at most $2n$ times. This results in a runtime of $O(n)$.

Proof of Correctness:
- We prove inductively that at every iteration n of the loop, if x is in the matrix it cannot be in a row above r or a column to the right of c.
- **Base Case:** When $n = 0$, (r, c) is the top-right corner, so there are no rows above r or columns to the right of c, meaning the base case is correct.
- **Inductive Hypothesis:** We assume that, for iteration $n = k$, x cannot be above r or to the right of c.
- **Inductive Case:** We want to show that after the next iteration $n = k + 1$, x cannot be above our new value of r or to the right of our new value of c.

    - If $x = \texttt{mat}[r][c]$ then returning `true` is correct since the element is in the matrix.
    - If $x > \texttt{mat}[r][c]$ then x cannot possibly be in row r, since everything to the left in the current row is even smaller, and everything to the right has been ruled out by our inductive hypothesis.
        - Everything with row $< r$ has also been ruled out by our inductive hypothesis.
        - Thus, we've proven everything above row $r + 1$ has been ruled out, meaning that it's fine for our algorithm to increment r.
    - If $x < \texttt{mat}[r][c]$ then x cannot possibly be in column c, since everything below in the current column is even larger, and everything above has been ruled out by our inductive hypothesis.
        - Everything with column $> c$ has also been ruled out by our inductive hypothesis.
        - Thus, we've proven everything to the right of column $c - 1$ has been ruled out, meaning that it's fine for our algorithm to decrement c.
    - We've proven our algorithm's correctness for every case, thus the inductive case is also correct.
- If our algorithm returns `false`, that means that r has moved out the bottom of the matrix of c has moved out the left, which, by our inductive proof, would mean that x cannot be in the matrix
- Thus, if our algorithm must be correct whenever it returns `false`
- Our algorithm can only return `true` when it's found the element exactly, which means it must also be correct whenever it returns `true`
- We've proven our algorithms correctness in all cases, so it's always correct.