This homework is due on gradescope Friday November 2nd at 11:59pm. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

Question 1 (Verifying Shortest Path Lengths, 30 points). Let $G$ be a directed, weighted graph with potentially negative edge weights, but no negative weight cycles. Let $s$ be a vertex of $G$. Using Bellman-Ford to compute the shortest path lengths from $s$ to the other vertices is somewhat slow, however verifying that the answer you have is correct is somewhat faster. Provide an algorithm that given values $d(v)$ for each vertex $v \in V$, determines whether or not $d(v)$ is the length of the shortest path from $s$ to $v$ for all $v$ (your algorithm should return a single TRUE/FALSE value, returning FALSE if there is any vertex $v$ so that $d(v)$ is not the correct shortest path length from $s$). For full credit your algorithm should run in linear time or better.

Solution 1.

Algorithm. To determine if the $d(v)$ we are given are correct, we can run a single Bellman-Ford update which iterates over $E$ and find, for all $v$, $u$, and $e = (u,v)$, $v_{\text{min}} = \min(d(u) + l(u,v))$. We then check if $d(v) = v_{\text{min}}$ (with the exception of $s$ which we just check to be equal to $0$). If not, we return FALSE. Otherwise, we need to perform an additional check.

Let $G'$ be the graph obtained from $G$ keeping only edges where $d(v) = d(u) + l(u,v)$. Verify that every other vertex with distance $< \infty$ is reachable from $s$ in $G'$ via DFS. If so, return TRUE. Otherwise return FALSE. [Note: this second test may be skipped if $G$ is guaranteed to have no zero-weight cycles.]

Correctness. Firstly, we note that if the distances are correct our algorithm must return TRUE. Since the true distances satisfy $d(v) = \min(d(u) + l(u,v))$, for $v \neq s$, it must be the case that $d(v) = v_{\text{min}}$ for all $v$. Additionally, for $v = s$, since there are no negative weight cycles, $d(s)$ must equal $0$.

To verify the second check, we note that the shortest path from $s$ to $v$ must only contain edges in $G'$. This is because as a shortest path, and subpath $s$ to some intermediate vertex $v$ must also be shortest. This means that for an edge $(u,w)$ that it uses, $d(w) = d(u) + l(u,v)$. Thus, this shortest path, gives a path from $s$ to $v$ in $G'$. Thus, our graph passes this second test.

Next, we need to verify that any graph that passes this test must have the correct distances. We verify this in two stages. First, we show that any path from $s$ to $v$ must have length at least $d(v)$, and then show that there always is a path of length at most $d(v)$.

For the first claim, we note that if we have a path $s = v_0, v_1, \ldots, v_n$, it must be the case that $l(v_i, v_{i+1}) + d(v_i) \geq d(v_{i+1})$ (by the first test), and therefore, that $l(v_i, v_{i+1}) \geq d(v_{i+1}) - d(v_i)$. This means that the length of the total path is at least $(d(v_1) - d(v_0)) + (d(v_2) - d(v_1)) + \ldots + (d(v_n) - d(v_{n-1})).$ This sum telescopes and noting that $d(s) = 0$, we have that the length of the path is at least $d(v_n)$.

On the other hand, if our graph passes the second check, if we have a path $s = v_0, v_1, \ldots, v_n = v$ from $s$ to $v$ in $G'$, the length of this path must be exactly $(d(v_1) - d(v_0)) + (d(v_2) - d(v_1)) + \ldots + (d(v_n) - d(v_{n-1})) = d(v)$. Therefore, there is a path of length $d(v)$.

Complexity. Since we iterate over every edge once to compute $v_{\text{min}}$, this step is $O(|E|)$. Since we check that $d(v) = v_{\text{min}}$ for all $v \in V$, this step is $O(|V|)$. Additionally, we compute the edges of $G'$ in linear time by iterating over the edges. Determining reachability in $G'$ can be done with a DFS which is $O(|V| + |E|)$. Therefore, the runtime of this algorithm is $O(|V| + |E|) + O(|E|) + O(|V|) = O(|V| + |E|)$.
Question 2 (The Skyline Problem, 30 points). A city’s skyline is traced out by \( n \) rectangular buildings. In particular, each building has a height \( h_i \) and a base given by an interval \([x_i, y_i]\). For our purposes we will assume that the \( x_i \) and \( y_i \) used by the \( n \) buildings are exactly the integers from 1 to \( 2n \) without any repeats. The height of the skyline at location \( \ell_i \) at a position \( i \) is given by \( \max_{x,y \leq \ell_i} h_i \). Give an algorithm to compute \( \ell_1, \ell_2, \ldots, \ell_{2n} \). For full credit, your algorithm should run in time \( O(n \log(n)) \) or better. Hint: use divide and conquer. Your subproblems may need to look slightly more complicated than your original problem.

Solution 2.

Algorithm. Let \( B \) be the initial set of building such that \( b_i = (h_i, [x_i, y_i]) \) for \( b_i \in B \) and \( L \) be a table indexed from 1 to \( 2n \) that will contain the value for \( \ell_i \) at position \( i \).

We recursively run an algorithm \( \text{Heights}(x, y, S) \) which takes endpoints \( x \) and \( y \) and a set \( S \) of buildings.

1. If \( x = y \), we store the height of the tallest building in \( S \) at \( L[x] \).

2. Else let \( z = \lfloor \frac{x+y}{2} \rfloor \) and \( S' \) be the set of buildings in \( S \) that have an endpoint in \([x, z]\) and the tallest building in \( S \) that contains the entire interval \([x, z]\). Let \( S'' \) be the corresponding set for the interval \([z+1, y]\). We then call the algorithm on both halves:

\[
\text{Heights}(x, z, S'), \text{Heights}(z+1, y, S'')
\]

We initiate the algorithm by calling \( \text{Heights}(1, 2n, B) \). After the algorithm terminates, \( L \) will contain the correct values for \( \ell_1, \ell_2, \ldots, \ell_{2n} \).

Correctness. We can see that the algorithm splits every problem into smaller subproblems until we hit our base case of \( x = y \). Due to the way we built \( S \), at this point \( |S| = 1 \) and \( S \) only contains the building with the tallest height covering \([x, y]\). We also see that the algorithm covers the entire range 1 to \( 2n \), and will thus correctly compute \( \ell_1, \ell_2, \ldots, \ell_{2n} \).

Complexity. Our runtime complexity is \( O(n \log(n)) \).

Our algorithm splits a problem of size \( n \) into two problems of size roughly \( n/2 \) at each level, giving us \( \log(n) \) levels. Notice that any point in the algorithm, \( |S| = O(y-x+1) \), as all but one building in \( S \) will have an endpoint in \([x, y]\). This implies that our input sizes for our base case of \( x = y \) will be \( O(1) \) and thus take \( O(1) \) time to process.

Because we iterate over \( |S| \) elements to build \( S' \), \( S'' \) this implies we do \( O(n) \) at each level of the recursion. Thus, the recurrence relation for the algorithm is \( T(n) = 2T(n/2) + O(1) \). If we use the Master Theorem, then \( a = 2, b = 2, d = 1 \) (note that \( a = b^d \)) which implies our runtime is \( O(n^d \log(n)) = O(n \log(n)) \)

Question 3 (Densest Interval, 30 points). Given a set \( S \) of \( n \) real numbers and a positive real number \( L \), give an algorithm to compute an interval of length at most \( L \) that contains as many elements of \( S \) as possible. For full credit, your algorithm should run in time \( O(n \log(n)) \) or better.

Solution 3.

Algorithm. First, we will sort all the numbers in \( O(n \log(n)) \). Let \( A = \{a_1, a_2, \ldots, a_n\} \) be the array of numbers that we obtain after the sorting, i.e. \( a_1 < a_2 < \ldots < a_n \). Now, we will solve the problem with the two-pointer technique:

```
left = 1
x = y = 0
for right = 1 to n:
    while a[right] - a[left] > L:
        left++
    if a[right] - a[left] > y - x:
        x = a[left]
        y = a[right]
return x, y
```
We have pointers left and right. The left pointer initially equals 1. We iterate the values for the right pointer and if at any step the absolute difference between the elements to which our pointers point is greater than L, then we increase the value of the left pointer. We increase the value of the left pointer until the distance becomes less than L. At each step of the loop we also try to update the optimal answer x, y.

Correctness. Let’s call the elements to which our left and right pointers point to left and right elements respectively.

First, if the array doesn’t contain any numbers, then our algorithm will return the interval [0, 0]. This is correct, because any interval of any length will contain zero elements from empty array. Let’s show that it is impossible to have left pointer until we have |l| ≤ 0.

We have pointers O total numbers of steps of the inner loop is O(n). The outer loop has exactly n steps of the outer loop. The left pointer initially equals 1, can only increase, and doesn’t exceed n.

For any non-empty array our algorithm only returns valid answers, i.e. intervals whose length is no greater than L. This is guaranteed by the inner loop which increases the left pointer while the distance between the left and the right elements is greater than L. The left pointer will also never exceed the right pointer as the distance from the element to itself is zero and L is a positive number.

Now, let [a′, b′] be the optimal answer. This range includes at least one element from A. Let x_l′, x_r′ be the smallest and the largest elements from A that belong to [a′, b′] (they may be the same number). As |x_l′ − x_r′| ≤ |a′ − b′|, any number x_i from A, such that x_l′ ≤ x_i ≤ x_r′, belongs to the optimal answer. Thus, the number of elements in the optimal answer [a′, b′] is the same as the number of elements that belong to [x_l′, x_r′].

In order to show that our algorithm works, we will prove that the algorithm finds the interval [x_l′, x_r′] at some point, i.e. left = l and right = r′. At some point we will have right = r′. We know that left ≤ right at any moment in our algorithm. Let’s show that it is impossible to have l′ < left and r′ = right. Indeed, if l′ < left, this means that at some point when we had l′ ≤ left and right ≤ r′, we had to increase the value of left. But this is not possible, because for any two elements in the interval [x_l′, x_r′] the distance between them is no greater than L. Thus, we can only have left ≤ l′ and right = r′. By the definition of the optimal answer, we also have that |x_l′ − x_r′| > L for all left < l′ and r′ = right. Thus, we will increase the left pointer until we have left = l′ and right = r′. As we only consider valid intervals in the algorithm, this will be the best interval that we can find with our algorithm.

Complexity. We sort elements in O(n log(n)). The outer loop has exactly n steps. Each step consist of an inner loop and an If statement. If statement takes O(1) for each step. The total number of steps the inner loop takes for all steps of the outer loop is determined by the number of times the left pointer increases for all steps of the outer loop. The left pointer initially equals 1, can only increase, and doesn’t exceed n. Thus, the total numbers of steps of the inner loop is O(n). Thus, we have O(n log(n)) + O(n) + O(n) = O(n log(n)).

Question 4 (Divide an Conquer Runtimes, 10 points). For each of the following, give the asymptotic runtime of the following divide and conquer algorithms:

(a) An algorithm that splits a problem of size n into five problems of size n/3 and does O(n^{3/2}) work combining the answers.

(b) An algorithm that splits a problem of size n into eight problems of size n/2 and does O(n^3) work combining the answers.

(c) An algorithm that splits a problem of size n into three problems of size 2n/3 and does O(n^3) work combining the answers.

(d) An algorithm that splits a problem of size n into one problem of size 99n/100 and does O(1) work combining the answers.

(e) An algorithm that splits a problem of size n into two problems of size 2n and does O(n) work combining the answers.

Solution 4. By the Master Theorem:

(a) T(n) = 5T(n/3) + O(n^{3/2}) = O(n^{3/2})

Proof: a = 5, b = 3, d = 3/2

(a = 5) < (5.2 ≈ 3^{3/2} = b^d) \implies O(n^d) = O(n^{3/2})
(b) \( T(n) = 8T(n/2) + O(n^3) = O(n^3 \log(n)) \)
Proof: \( a = 8, b = 2, d = 3 \)
\((a = 8) = (8 = 2^3 = b^d) \implies O(n^d \log(n)) = O(n^3 \log(n))\)

(c) \( T(n) = 3T(2n/3) + O(n^3) = O(n^3) \)
Proof: \( a = 3, b = 3/2, d = 3 \)
\((a = 3) < (3.372 = 3/2^3 = b^d) \implies O(n^d) = O(n^3)\)

(d) \( T(n) = T(99n/100) + O(1) = O(\log(n)) \)
Proof: \( a = 1, b = 100/99, d = 0 \)
\((a = 1) = (1 = 100/99^0 = b^d) \implies O(n^d \log(n)) = O(n^0 \log(n)) = O(\log(n))\)

(e) \( T(n) = 2T(2n) + O(n) = \text{undefined [this “algorithm” will never terminate]} \)
Proof: \( a = 2, b = 1/2, d = 1 \)
But a divide and conquer algorithm will only make progress when \( b > 1 \).

**Question 5** (Extra credit, 1 point). *Approximately how much time did you spend on this homework?*