CSE 202 **Fall 2017**

Final

Jimmy Ye A12405090

Problem 1: Sequence matching

Let T and P be respectively two sequences $t_1, ..., t_n$ and $p_1, ..., p_k$ of characters such that $k \leq n$. The characters range over a finite alphabet Σ. With each position of T, we associate a nonnegative value, $v(i)$. Find a matching subsequence of T for the pattern P that maximizes the sum of values. That is, find the sequence of indices $1 \leq i_1 < i_2 < ... < i_k \leq n$ such that for all $1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{j=1}^{k} v(j)$ is maximized. Design an efficient algorithm for this problem, prove its correctness and analyze its complexity.

Algorithm:

We define a variant of the standard dynamic programming approach to longest common subsequence. Let $Q(i, j)$ be the maximum total value from a match of the pattern $p_j, ..., p_k$ with a subsequence of $t_i, ..., t_n$. If there is no match, we define $Q(i, j) = -\infty$, since the values $v(i)$ of each node are nonnegative. Additionally, we allow the i and j parameters to be $n + 1$ and $k + 1$, respectively, for convenience in defining the base case. The final value we are interested in maximizing is $Q(1, 1)$. We define $Q(i, j)$ as:

$$Q(i,j) = \begin{cases} 0 & i \leq n+1, j = k+1 \\ -\infty & i = n+1, j \leq k \\ Q(i+1, j) & i \leq n, j \leq k, t_i \neq p_j \\ \max(v(i) + Q(i+1, j+1), Q(i+1, j)) & i \leq n, j \leq k, t_i = p_j \end{cases}$$

Only in the fourth case do we match a character of T with a character from P. To keep track of which characters are matching, we define $R(i, j)$ to be 1 if the value of $Q(i, j)$ is the left term of the max in the fourth case, and 0 in all other cases. Then to recover a subsequence starting from any $R(i, j)$, if $R(i, j)$ is 1 we append i to the subsequence and go to $R(i + 1, j + 1)$, and otherwise we go to $R(i + 1, j)$ without appending.

We compute Q (and likewise R) from the base cases up to $Q(1, 1)$.

Once we have computed $Q(1, 1)$ and $R(1, 1)$, if $Q(1, 1) = -\infty$ we return an empty sequence to indicate lack of a match, otherwise we return the sequence of indices obtained from starting at $R(1, 1)$ and proceeding as described above.

Time Complexity:

The total time complexity is $O(nk)$, since $1 \leq i \leq n+1$, $1 \leq j \leq k+1$, $Q(i, j)$ and $R(i, j)$ each take constant time to compute for one entry, and iterating through R takes $O(n + k)$ time which is dominated by $O(nk)$.

Correctness:

In the base case where we have matched all the characters of P with a subsequence of T, the remaining value is 0. In the base case where we have some characters of P remaining, but we have exhausted T, we have failed to find a matching subsequence so the appropriate value is $-\infty$. Any matching subsequence will have total value ≥ 0, so $Q(1,1) = -\infty$ iff there are no matching subsequences overall. So the algorithm handles the base cases of Q correctly.

In the recursive cases, we have no choice if $t_i \neq p_j$, we must advance the t-index i to $i+1$. If $t_i \neq p_j$, then there is a choice of whether to match i and j, so we take the max over both choices; if we choose to match it contributes $v(i)$ and we also need to advance the p-index, otherwise we only advance the t-index. So the algorithm handles the recursive cases of Q correctly.

R is simply a binary table computed in additional constant time for each entry of Q which is 1 if there was a match at i, j and 0 otherwise. The algorithm's iteration through R produces the sequence of matches corresponding to a given $Q(i,j)$, so the algorithm correctly produces the matching subsequence with maximum total value.

Problem 2: Covering points with gain

Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line where the point x_i has a gain $w_i \geq 0$ and a set of closed intervals $[s_i, f_i]$ for $1 \leq i \leq d$ where the cost of the i-th interval is $c_i < 0$, select a set of closed intervals so that the sum of the gains of the points covered by the selected intervals less the sum of the costs of the selected intervals is maximized. Design an efficient algorithm, prove its correctness and determine its complexity.

Algorithm:

First, sort the x_i in nondescending order, and sort the intervals in nondescending order of their s_i.

Next sort the intervals in nondescending order of their f_i. Let $g(i)$ be the interval j with minimum f_j such that $s_j \geq s_i$ and $f_j > f_i$. (We let $g(d) = d + 1$ to handle base cases later.) We can compute this in a linear scan of the f_i-sorted list of intervals, keeping a stack containing their indices in the s_i-sorted list, popping all elements with lesser s_i every time we push.

Let $h(i)$ be the index j of the rightmost point in interval i. Compute this by sorting all interval endpoints f_i and points x_j in one list, then scanning from left to right, maintaining the rightmost point x_j seen, and when we encounter f_i, set $h_i = j$.

We use a dynamic programming approach, and we define the following recurrence: Let $T(k, l)$ be the maximum value (gains less costs) which can be obtained from the intervals $[s_i, f_i]$ for $k \leq i \leq d$ and the points $x_l, ..., x_d$ (as first stated, points sorted nondescending and intervals sorted by nondescending s_i). The final result desired is $T(1, 1)$.

We compute $T(k, l) = \max(T(g(k), h(k) + 1) - c_k + \sum_{j=l}^{h(k)} w_j, T(i + 1, l))$. That is, if we choose the k-th interval, then we reduce to the subproblem where the leftmost interval covers some space to the right of the k-th interval, accumulating the cost of the k-th interval and newly covered points, and if we do not choose the k-th interval, we simply advance to the next interval. In the base case, if $k = d + 1$ or $l = n + 1$, then $T(k, l) = 0$, since we have either exhausted all intervals or all points.

Let $S(k, l) = 1$ if $T(k, l)$ is maximized by the left term, otherwise $S(k, l) = 0$ when $T(k, l)$ is maximized by the right term.

After computing $T(1, 1)$ (and therefore $S(1, 1)$), return the set of intervals found by starting at $S(1, 1)$, adding interval k if $S(k, l) = 1$, and advancing to the next appropriate $S(k', l')$ according to the same recurrence as T, until either $k = d + 1$ or $l = n + 1$.

Time Complexity:

The total time complexity is $O((n + d) \log(n + d) + dn^2)$: The sorts and linear scans are dominated by the sort of the combined list of interval endpoints and points. Each call to T takes at most $O(n)$ time to sum the gains of the newly covered points and there are $O(dn)$ entries of T since $1 \leq k \leq d + 1$ and $1 \leq l \leq n + 1$.

Correctness:

$h(i)$ is computed correctly since we directly visit all interval endpoints and points and assign the rightmost before f_i to $h(i)$. $g(i)$ is computed correctly by finding the Next Greater-s_i Element in an f_i-sorted list: since we pop all i where $s_i < s_j$, by transitivity we will always pop s_i exactly on the first element j to the right (i.e. interval with least f_i) with $s_i < s_j$.

In the base case, there are either no points or no intervals, so the total net value is 0, so the algorithm's base cases are correct.

In the recursive case, the algorithm must consider the two possibilities of selecting and not selecting the k-th interval. If it is not selected, simply advancing the interval index is sufficient, so the algorithm handles this correctly. By construction, $g(k)$ is the leftmost-finishing interval which covers space to the right of the k-th interval, and $h(k) + 1$ is the leftmost point to the right of the k-th interval. Thus the algorithm correctly advances the interval and point indices, and correctly accumulates the value by subtracting the new cost and adding the new gains.

Since T correctly maximizes the value, and S reconstructs the choices made in T, the algorithm correctly selects a set of intervals maximizing value.

Problem 3: Round-robin tournament

Assume that a round-robin tournament is played among n players. That is, each player plays once against all $n-1$ other players. There are no draws and the results of all games are given in a matrix A. Entry $A(i,j)$ is 1 if player P_i beat player P_j and 0 otherwise. It is not possible in general to sort the players, since P_i may beat P_j, P_j may beat P_k, and P_k may beat P_i. In other words, the results are not necessarily transitive. We are interested in a weak sorting as follows. Design an algorithm to arrange the players in an order $P_{\sigma(1)}, P_{\sigma(2)}, ..., P_{\sigma(n)}$ such that $P_{\sigma(1)}$ beat $P_{\sigma(2)}$, $P_{\sigma(2)}$ beat $P_{\sigma(3)}$, ..., $P_{\sigma(n-1)}$ beat $P_{\sigma(n)}$, where σ is a permutation function on the integers $\{1, 2, ..., n\}$. In other words, $A(\sigma(1), \sigma(2)) = A(\sigma(2), \sigma(3)) = ... = A(\sigma(n-1), \sigma(n)) = 1$. The worst-case running time of the algorithm should be $O(n \log_2 n)$. Any entry in the matrix can be accessed in constant time.

Algorithm:

We represent the players using their indices, and begin with the sequence $S = [1, 2, ..., n]$. We will sort the players using ascending merge-sort on S where our comparison function is as follows: player index i compares as "less than" player index j if $A(i,j) = 1$. After computing $S' = MergeSort(S)$, we return the sorted sequence of player indices S', as a representation of σ where $\sigma(i) = S'_i$.

Time Complexity:

The comparison function is constant time, so the worst-case running time is the same as that of merge-sort: $O(n \log_2 n)$.

Correctness:

Consider the $Merge$ procedure: $Merge$ is correct if the $Merge(A, B)$ procedure returns a correctly ordered sequence of player indices given correctly ordered sequences A and B.

Let $A = a_1, a_2, ..., a_k$, $B = b_1, b_2, ..., b_l$, and let C be the output sequence. Towards induction we assume that C is correctly ordered and we are comparing a_i and b_j as candidates to be appended after c, the last element of C.

Consider the possible values of c. Since $Merge$ advances either the A-index or B-index at each step, and it only appends the winning player to C, either a_{i-1} beat b_j so $c = a_{i-1}$, or b_{j-1} beat a_i so $c = b_{j-1}$. Assume without loss of generality that the first case is true (the argument is identical swapping A and B). Then since a_{i-1} beat both a_i and b_j, regardless of whether a_i beat b_j or not, C will be correctly ordered after appending one more element.

If we are not comparing an a_i and b_j, i.e. we have exhausted one of the sequences, then the same argument (less some cases) shows that the first element of the remaining input sequence can be appended to C while preserving the ordering, and the rest of the elements likewise by the correct ordering of A and B.

Since the empty sequence is trivially correctly ordered, by induction, C will be correctly ordered after all elements of A and B have been appended. So $Merge$ returns a correctly ordered sequence given correctly ordered sequences. By the correctness of merge-sort, the final result S' will be a permutation of $\{1, 2, ..., n\}$ such that S'_i beat S'_{i+1} for all $1 \leq i < n$.

Problem 5: Center selection with producers and consumers

Consider the following variation of the center selection problem. You are given a set V of n vertices and a distance function $d(.,.)$ that satisfies the non-negativity, symmetry and triangle inequality properties. However, the vertices are partitioned into two categories, producers and consumers. Let $P \subseteq V$ denote the set of vertices corresponding to the producers and $Q = V - P$ denote the set of vertices corresponding to the consumers. You are also given an integer $k \geq 1$. The objective is to select a set C of k producers so that $\max_{q \in Q} d(q, C)$ is minimized, that is, the maximum distance from a supplier to a consumer is minimized. Provide a 3-approximation algorithm for the problem. Prove its correctness, bound its approximation ratio and determine its complexity.

For a set of points R and a point s, we define $d(s, R) = \min_{r \in R} d(s, r)$. You can assume that the distance between two vertices can be computed in constant time.

Algorithm:

We define a greedy algorithm as follows: Initially, choose an arbitrary consumer q_0, and select the producer p_0 closest to q. Find the consumer q' which is farthest away from any currently selected producer, and select the corresponding unselected producer p' which is the closest to q'. Repeat this until k producers have been selected, and return the set of k selections.

Time Complexity:

The total time complexity is $O(kn)$, since there are k iterations and in each iteration:

- Finding the maximum-distance consumer takes $O(n)$ time: for each consumer, we only need $O(1)$ time to take the min of its previous distance and its distance to the most recently selected producer.
- Finding the closest producer to that consumer takes $O(n)$ time: a linear scan through $O(n)$ producers to find the min distance.

Correctness: The algorithm iterates k times and selects an unselected producer in each iteration, so it correctly returns a set of k producers.

Ratio:

Let OPT be an optimal set of k selections, and let $r_{OPT} = \max_{q \in Q} d(q, OPT)$. Let G be the set of k selections returned by our greedy algorithm, and let $r_G = \max_{q \in Q} d(q, G)$.

Consider the selection g_i at iteration i of the greedy algorithm. g_i is the either the closest producer to some consumer q, or the closest producer to q has already been selected, and g_i is merely the closest unselected producer. If the latter case is true, this implies the worst-distance consumer is already as close as it can be to a producer, so the greedy algorithm has already found an optimal solution using $< k$ producers.

So we may assume that g_i is the closest producer to some consumer q.

Let $B(p, r)$ be the set of all vertices (consumers or producers) x such that $d(x, p) \leq r$. By construction, $q \in B(p, r_{OPT})$ for some $p \in OPT$. Since g_i is the closest producer to q, $d(g_i, q) \leq d(q, p) \leq r_{OPT}$. By triangle inequality, $d(g_i, p) \leq d(g_i, p) + d(p, q) \leq 2\ r_{OPT}$. Applying triangle inequality one more time for an arbitrary $q' \in B(p, r_{OPT})$, we get $d(g_i, q') \leq d(g_i, p) + d(p, q) \leq 3\ r_{OPT}$. We refer to this as g_i *covering* the optimal producer p.

Then it suffices to show that all optimal producers $p \in OPT$ are covered by at least one g_i in this way. By the above, every g_i covers at least one optimal producer p. We show that at any iteration of the greedy algorithm, one of these two cases must hold: either we have already satisfied $r_G \leq 3\ r_{OPT}$, or one new optimal producer is covered by g_i.

If the first case is true, we are already done. Then assume we are in the second case: there are consumers which are at least 3 r_{OPT} away from all greedily selected producers. Then the max-distance consumer q is more than 3 r_{OPT} away from previous greedy selections, and therefore there must be a corresponding uncovered optimal producer p. Then the greedy algorithm will select the closest producer to q, this selection g_i will cover p, and we are done.

Thus in every case, after k selections, $d(q, G) \leq 3\ r_{OPT}$ for all $q \in Q$, so $\dfrac{r_G}{r_{OPT}} \leq 3$.

Problem 6: Approximation algorithm

Consider the following optimization problem: Given n integers $c_1, c_2, ..., c_n$, find a partition of $\{1, 2, ..., n\}$ into two subsets S_1 and S_2 that minimizes the quantity $\max(\sum_{i \in S_1} c_i \sum_{i \in S_2} c_i)$. Consider the following heuristic for some fixed integer k:

1. Choose the k largest c_i's
2. Find the optimal partition of these k integers
3. Complete this into a partition of $\{1, 2, ..., n\}$ by considering each of the remaining c_i's and adding it to the partition which at the time has the smallest sum.

Prove that this algorithm with $O(2^k + n)$ time complexity outputs a sum that is within $(k+3)/(k+1)$ fraction of the optimal output. (Note: you can assume that c_i's are non-negative integers.)

Proof:

Let $U = \{1, 2, ..., n\}$ and K be the set of indices corresponding to the k largest c_i's. Let $C = \sum_{i \in \{1,2,...,n\}} c_i$ be the sum of the given integers.

The algorithm has $O(2^k + n)$ time complexity: We can calculate the sum of values of a partition in constant time per partition by a straightforward recurrence with an accumulator parameter: let $Q(i, a) = \min(Q(next(i), a), Q(next(i), a + c_i))$, and in the base case let $Q(end, a) = \max(a, C - a)$ (where *end* denotes a terminal index and *next* gives the next c_i). There are $O(2^k)$ calls to Q and each takes $O(1)$ time, so with additional $O(1)$ tracking of the argmin at each call, we can compute the optimal partition in $O(2^k)$. Iterating through U and greedily assigning each remaining index takes $O(1)$ time per index and therefore $O(n)$ time overall.

Clearly if $k = n$ we achieve the optimal solution, so we may assume that $k < n$.

Let $(S_G, S_{G'})$ be the partition given by the heuristic algorithm, and let $G = \sum_{i \in S_G} c_i$, $G' = C - G$ (WLOG assume $G \geq G'$) be the corresponding values of the partition.

Towards contradiction, assume that $G > \dfrac{k+3}{k+1} OPT \geq \dfrac{k+3}{k+1}\left(\dfrac{C}{2}\right)$. Then $G' \leq \dfrac{k-1}{k+1}\left(\dfrac{C}{2}\right)$.

Consider the last integer c_z added to S_G. Note that by the above, $G - G' \geq \frac{2}{k+1}C$, and therefore (since we can only add nonnegative integers to each set) $c_z \geq G - G' \geq \frac{2}{k+1}C$.

We refer to the optimal partition of the largest k integers as the optimal phase and the greedy completion as the greedy phase. There are two cases: either c_z was added to S_G during the optimal phase, or during the greedy phase.

In the first case, since the last element added to S_G was during the optimal phase, the greedy phase only added elements to $S_{G'}$. However, this implies the heuristic achieves an optimal solution (which is a contradiction): The sum of the smallest $n-k$ elements is smaller than the partition gap after the optimal phase, but the gap of any other partition of the largest k elements can only be larger, which would still be larger than the sum of the smallest $n-k$ elements. Thus no partition can improve on the one given by the heuristic in this case.

But if c_z was added to S_G during the greedy phase, then since the k largest elements must all be greater than c_z, their sum (plus c_z) must be $\geq 2C > C$, which is a contradiction.

Thus by contradiction $G \leq \dfrac{k+3}{k+1} OPT$ as desired.