COMP 3711 – Design and Analysis of Algorithms
2024 Fall Semester – Written Assignment 3
Distributed: 9:00 on October 28, 2024
Due: 23:59 on November 11, 2024

Your solution should contain
(i) your name, (ii) your student ID #, and (iii) your email address
at the top of its first page.

Some Notes:

- Please write clearly and briefly. In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.
- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page. ***You must acknowledge individuals who assisted you, or sources where you found solutions.*** Failure to do so will be considered plagiarism.
- The term *Documented Pseudocode* means that your pseudocode must contain documentation, i.e., comments, inside the pseudocode, briefly explaining what each part does.
- Many questions ask you to explain things, e.g., what an algorithm is doing, why it is correct, etc. To receive full points, the explanation must also be *understandable* as well as correct.
- Submit a SOFTCOPY of your assignment to Canvas by the deadline. If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.

1. (25 points) You are given the locations of n buildings $B_1, \ldots, B_n$ on the real line. Each location is a coordinate (i.e., a real number). Each building B_i has a WIFI signal receiver with a range of $r_i > 0$. That is, if we place a signal tower at distance r_i or less from B_i, then B_i gets WIFI. Note that the signal receivers of different buildings may have different ranges.

   Describe a greedy algorithm that places the smallest number of signal towers so that every building gets WIFI. Explain the correctness of your algorithm. Derive the running time of your algorithm.

   Solution: The algorithm follows these steps:

   (a) **Sort the Buildings:** First, sort the buildings based on their rightmost coverage point $x_i + r_i$.

   (b) **Place Towers:** Initialize an empty list for towers. Iterate through the sorted list of buildings:

   - For the current building B_i:
     - If it is not already covered by the last placed tower, place a new tower at the rightmost point of the current building's coverage, which is $x_i + r_i$.
     - Move to the next building that is not covered by the last placed tower.

   (c) Repeat until all buildings are covered.

   The correctness of the greedy algorithm can be justified as follows:

   - Placing the tower at the rightmost coverage point of the current building maximizes the range of the tower and covers as many subsequent buildings as possible.
   - By moving to the next uncovered building after placing a tower, we ensure that we do not miss any building that requires coverage.

   We can prove the optimality of our algorithm using this argument:

   Consider that the solution in this approach is shown as T and it contains $T = (t_1, t_2, \cdots, t_k)$ which are the positions of k towers. And assume the optimal solution is shown as $T^* = (t_1^*, t_2^*, \cdots, t_{k'}^*)$. Note that as T^* is the optimal solutions we must have $k' \leq k$.

   Now consider the smallest index j such that $t_i \neq t_j^*$ or in other words j is the first place that the solutions are different. Which means for any $i < j$ we have $t_i = t_i^*$. Now consider the sorted list of the buildings based on their rightmost coverage point. Assume B_m is the first uncovered building in that list. We know that $t_j^* < x_m + r_m$ otherwise B_m wont be covered by any signal tower. Also we know that $t_j = x_m + r_m$. Now we can see that if we move t_j^* to the position of t_j, the new modified T^* still covers all the buildings. So this means the solution T can not work worse than T^* and in the worst case it gives an answer that is equal to T^*.

- **Sorting:** The initial sorting of buildings based on their rightmost coveragte point takes $O(n \log n)$.
- **Iterating through buildings:** The subsequent iteration through the buildings takes $O(n)$.

Thus, the overall time complexity of the algorithm is:

$$O(n \log n)$$

2. (25 pts) Given an array $A[1..n]$ of positive integers and an integer $m \leq n$, we want to split A into at most m non-empty contiguous subarrays so that the largest sum among these subarrays is minimized. Design an algorithm based on a greedy strategy to solve this problem. Analyze the running time of your algorithm. Explain the correctness of your algorithm.

   For example, if the array is $A = [7, 2, 5, 10, 8]$ and $m = 2$, there are several ways to split A: $([7, 2, 4, 10, 8], \emptyset)$, $([7], [2, 5, 10, 8])$, $([7, 2], [5, 10, 8])$, $([7, 2, 5], [10, 8])$, and $([7, 2, 5, 10], [8])$. The best way is $([7, 2, 5], [10, 8])$ because the maximum sum is $10 + 8 = 18$ which is the minimum among all possible ways of splitting.

   Hint: You need to invoke a greedy algorithm multiple times. How many times?

   Solution:

   There are several solutions to this problem. We first argue one subquestion that will be used in all of them!

   The question is as follows:

   Consider having an array $A = \{a_1, a_2, \cdots, a_n\}$. The question is if its possible to split this array into at most m contiguous subarrays such that the largest sum among these subarrays is at most k.

   What we can do is to have a greedy approach. We start from the beginning of the array, we pick numbers as long as the summation does not exceed k. Then, if picking the next number cause an issue, then we have a new sub-array and we start picking numbers again such that their sum does not exceed k. We continue this process until we cover all the numbers. If we have at most m sub-arrays then we can say that it is POSSIBLE. If we can not do it with at most m sub-arrays then we say IMPOSSIBLE.

   In order to prove that this approach is optimal we use the following argument.

   Assume the optimal solution is shown as $T^* = A_1^*, \cdots, A_k^*$ and the greedy solution is shown as $T = A_1, \cdots, A_{k'}$.

   Assume the first index for the subarrays that the subarrays are different is j. So it means $A_j^* \neq A_j$. Now we know that $A_j^* \subset A_j$. Now note that we can easily move some elements from A_{j+1}^* or more into A_j^* to get to A_j without crossing the threshold of the total sum. So as you can see the number of partitions in greedy solution is either equal to the optimal solution or even smaller. So this means that If the optimal solution outputs POSSIBLE then the greedy one also outputs POSSIBLE. So we will not see a case such that the greedy algorithm outputs IMPOSSIBLE while the optimal solution is POSSIBLE.

   - **Solution 1:** We can consider all possible contiguous subarrays in this array. Note that we have $\binom{n}{2} = O(n^2)$ possible contiguous subarrays (because we need to pick two index i, j $i < j$ that maps to subarray $A[i..j]$). Now we calculate the sum of each subarary (S_{ij}) which takes $O(n)$ for each subarray.

Then we have to check if its possible to split A into at most m contiguous subarrays such that the sum of each subarray is at most S_{ij}. The answer to some of them is POSSIBLE and to some of them is IMPOSSIBLE. The last step is to find the "best" option which in this case is the smallest possible S_{ij} that gives us POSSIBLE.

Time complexity: There are $O(n^2)$ different contiguous subarrays. To calculate the sum of each it takes $O(n) \cdot O(n^2) = O(n^3)$.

For each S_{ij} check if its possible to split A into at most m subarrays such that the sum of each subarray is at most S_{ij}. $O(n) \cdot O(n^2) = O(n^3)$

Now we need to find the minimum possible S_{ij} that gives us POSSIBLE. This can be done in $O(n^2)$.

The total complexity is : $O(n^3) + O(n^3) + O(n^2) = O(n^3)$.

- **Solution 2:** We use the same idea as before, but instead of checking all S_{ij} to see if it gives POSSIBLE or IMPOSSIBLE and then find the minimum value, we sort them first which takes $O(n^2 \log(n^2)) = O(n^2 \log n)$. Now we use binary search to find the smallest possible S_{ij} that gives us POSSIBLE.

  Time complexity: There are $O(n^2)$ different contiguous subarrays. To calculate the sum of each it takes $O(n) \cdot O(n^2) = O(n^3)$.

  We sort them which takes $O(n^2 \log n)$.

  Then we use binary search to find the smallest possible value that gives POSSIBLE which takes $O(n) \cdot O(\log(n^2)) = O(n \log n)$.

  The total complexity is : $O(n^3) + O(n^2 \log n) + O(\log n) = O(n^3)$.

- **Solution 3:** We know that the the smallest possible value for sum of subarrays has to be less than *sum* (the sum of all the numbers in the array) and more than *biggest* (the largest element). We can use binary search in this range to find the smallest possible value that gives us POSSIBLE.

  Time complexity: Then we use binary search to find the smallest possible value that gives POSSIBLE in range $(biggest, sum)$ which takes $O(n) \cdot O(\log(sum - biggest)) = O(n(\log(sum - biggest))$.

3. (25 points) Consider a two-dimensional array $A[1..n, 1..n]$ of distinct integers. We want to find the *longest increasing path* in A. A sequence of entries $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ is a path in A if and only if every two consecutive entries share a common index and the other indices differ by 1, that is, for all k,

   - either $i_k = i_{k+1}$ and $j_{k+1} \in \{j_k - 1, j_k + 1\}$, or
   - $j_k = j_{k+1}$ and $i_{k+1} \in \{i_k - 1, i_k + 1\}$.

   A path in A is increasing $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ if and only if $A[i_k, j_k] < A[i_{i+1}, j_{k+1}]$. The length of a path is the number of entries in it.

   Design a dynamic programming algorithm to find the longest increasing path in A. Your algorithm needs to output the maximum length as well as the indices of the array entries in the path. Note that there is no restriction on where the longest increasing path may start or end. Define and explain your notations. Define and explain your recurrence and boundary conditions. Write your algorithm in pseudo-code. Derive the running time of your algorithm.

   Solution: We will consider all paths of lengths from 1 to n^2. We will build up from smaller paths to larger paths using the following recurrence: $DP[i_1, j_1, i_2, j_2, r]$ is $True$ if there is a path of length r from cell i_1, j_1 to cell i_2, j_2. For $r = 1$, $DP[i_1, j_1, i_2, j_2, r] = true$ if $i_1 == i_2$ and $j_1 == j_2$, otherwise it is $False$. For $r > 1$ $DP[i_1, j_1, i_2, j_2, r] = true$ if there is a cell i_3, j_3 such that the cell is either vertically or horizontally adjacent to i_2, j_2, $A[i_3, j_3] < A[i_2, j_2]$ and $DP[i_1, j_1, i_3, j_3, r-1]$ is *true*.

   Note that we first fill this 5D array when $r = 1$. Then, for filling any cell with $r = 2$ we only need to check at most 4 cells with $r = 1$ which we already calculated before. In general, for filling any cell with $r = k$ we need to consider at most 4 cells with $r = k - 1$ and all these values are already available. The solution is is the largest r that contains a True in the table.

   For running time:

   $O(n \cdot n \cdot n \cdot n \cdot n2) = O(n^6)$

4. (25 points) Let T be a rooted full binary tree of n nodes (not necessarily balanced). Each node v of T is given a positive weight $w(v)$. The depth $d(v)$ of v is the number of edges between v and the root of T. An *ancestor* of v is either v itself or a node u that can be reached from v by following parent pointers.

   Let $k \leq n$ be a given positive integer. You are asked to mark exactly k nodes of T as depots such that:

   - Every node v in T has a depot as its ancestor. Among the depots that are ancestors of v, the one with the largest depth is the *nearest depot* of v. We denote it by t_v. It is possible that $t_v = v$.
   - The sum $\sum_{v \in T} w(v) \cdot (d(v) - d(t_v))$ is minimized.

   Design a dynamic programming algorithm for this problem. You only need to output the minimized sum. You need to derive and explain your recurrence and boundary conditions. Analyze the running time of your algorithm.

   Hint: Depending on the tree height and k, you may need many subproblems for a subtree if the subtree root is not a depot.

   Solution: Define $\text{DP}[v, j, d_{rel}]$ as the minimum cost to mark exactly j depots in the subtree T_v rooted at node v while the distance between v and its nearest ancestor depot but not itself is d_{rel}.

   There are two cases:

   - **Case 1:** We mark v as depot. In this case one of the j depots are used. So there are only $j-1$ depots left that we need to distribute among the left child subtree and right child subtree so that the cost is minimized. At the same time, d_{rel} for both of its children will be set to 1 as now their distance with the nearest depot is 1 now. Also we need to check how much cost node v contributes. We can easily see that it contributes 0 cost as the nearest depot to v is itself. So we can see that if we mark v as depot then:

     $$Cost_{Yes} = 0 + \min_{j_1+j_2=j-1} (\text{DP}[\text{left}(v), j_1, 1] + \text{DP}[\text{right}(v), j_2, 1])$$

   - **Case 2:** We **do not** mark v as depot. In this case we need distribute all j depots among the left child subtree and right child subtree . At the same time, d_{rel} for both of its children will be set to $d_{rel}+1$ now because now the nearest depot to those children is 1 + the nearest depot to their parent. Also we need to check how much cost node v contributes. We can easily see that it contributes $w(v) \times d_{rel}$ cost as the nearest depot to v has distance d_{rel}. So we can see that if we do not mark v as depot then:

     $$Cost_{No} = w(v) \times d_{rel} + \min_{j_1+j_2=j} (\text{DP}[\text{left}(v), j_1, d_{rel}+1] + \text{DP}[\text{right}(v), j_2, d_{rel}+1])$$

The answer will be: DP$[(v), j] = min(Cost_{yes}, Cost_{no})$. Also note that for root node we consider it as depot and we need to distribute $k-1$ depots among the left subtree and right subtree using the same idea.

For the boundry conditions:

- If v is a leaf node and $j = 1$, then DP$[v, j, d_{rel}] = 0$. because the distance of that node to the first depot is 0 (as that leaf node itself is a depot)
- If v is a leaf node and $j = 0$, then DP$[v, j, d_{rel}] = w(v) \times d_{rel}$.

For the time complexity we have an array of size $n \times k \times n$ and for filling each one we need to go through $O(k)$ cases to find the most optimized way to distribute the remaining depots. So the time complexity is $O(n^2 \times k^2)$.