Homework 2

Wenjun Zhang

A53218995

Problem 1: Nesting Boxes (CLRS)

Question 1

Let x, y and z be three d-dimensional boxes where x nests with y and y nests within z, then x, y, z satisfy the relations below:

$$x_{\pi_1(1)} < y_1, x_{\pi_1(2)} < y_2, \dots, x_{\pi_1(d)} < y_d$$

$$y_{\pi_2(1)} < z_1, y_{\pi_1(2)} < z_2, \dots, y_{\pi_1(d)} < z_d$$

Then we can derive the relation as below:

$$x_{\pi_1(\pi_2(1))} < z_1, x_{\pi_1(\pi_2(2))} < z_2, \dots, x_{\pi_1(\pi_2(d))} < z_d$$

Therefore, there exists a permutation π_3 on {1, 2, …, d} such that $x_{\pi_3(1)} < z_1, x_{\pi_3} < z_2, \dots, x_{\pi_3} < z_d$, where $\pi_3(i) = \pi_1(\pi_2(i))$. So the nesting relation is transitive.

Question 2

Algorithm description:

We are asked to determine whether a d-dimensional box nests inside another. We can assume the two boxes are x and y and the algorithm is to check whether x nests within y.

We first sort the two boxes x and y individually in non-decreasing order. If $y_i < x_i$. We switch the position of x and y to check is y nests within x.

We return true if the following equation is satisfied

$$x_i < y_i \qquad \text{for } 1 \le i \le d$$

Otherwise, we return false.

Proof of correctness:

To argue the correctness of the algorithm, we first prove that sorting will not change whether a box nests within another. Since sorting only changes the order of the elements in the box, it does not add or delete an element, so sorting will not affect if a box nests within another box.

Then we prove by induction that $x_i < y_i$ is a must for any *i* to ensure *x* nests within *y*. If the size of *x* and *y* is 1, then $x_1 < y_1$ can mean that *x* nests within *y*. Then we assume the length of *x* and *y* is n. For any $i < n(n \ge 2)$, $x_i < y_i$ but $x_n \ge y_n$. It's

obvious that there will be no such k that satisfies $x_n < y_k$ for $k \in [1, n]$, since y is sorted in non-decreasing order. So $x_n \geq y_n$ disobey the rule that x nests within y and we prove the correctness of the algorithm.

Time complexity:

The sort algorithm takes $O(dlogd)$ and then we only need to traverse the box once, so the time complexity for this algorithm is

$$T(n) = O(dlogd) + O(d) = O(dlogd)$$

Question 3:

Algorithm description:

We are given a set of n-dimensional boxes $\{B_1, B_2, \dots, B_n\}$ and we are asked to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \dots, k-1$.

To begin with, we first sort each box B_i individually in non-decreasing order so as to determine whether a box nests within another. Then we can assume a box B_i to be a vertex V_i, and if a box B_i nests within another box B_j, we add a directed edge (V_i, V_j) to the edge set *E*. So this problem now becomes finding the longest path in an unweighted DAG *G = (V, E)*.

To find the longest path in an unweighted DAG, we assign a new node S into the vertex set *V*, where S is connected to all the vertexes. After constructing the graph, we need to get the Topological order for all the nodes.

Next, we can use the Dynamic Programming to solve this problem. We define DP(V) is the longest path from S to V, with base case DP(S) = 0 and the recurrence is as below:

$$DP(V) = max_{(U,V)\in E}\{DP(U) + 1\}$$

where U are all nodes that have a directed edge to V and are in front of V in Topological sorting order.

We finally return $max_{U \in V} DP(U)$ as the output.

Proof of correctness:

To prove the correctness of the algorithm, we first prove that the graph we construct is a DAG. Since from question 1 we have proved that the nesting relation is transitive, if there is any circle in the graph, a tail of the nesting relation must nest within a head of this relation, which will be a contradiction.

Next, we prove that the longest path in this DAG represents the longest sequence we want in this question. Since we use vertex V_i to represents a box B_i in this question, a directed edge means a nesting relation, so a longest path $\{V_{i_1}, V_{i_2}, \dots, V_{i_k}\}$ means exactly the same as the longest sequence $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \dots, k-1$.

Finally, we prove that the Dynamic Programming method can return the longest path in this DAG. We first assume a starting node S so DP(S) = 0. From the recurrence

$$DP(V) = max_{(U,V)\in E}\{DP(U) + 1\}$$

Since we traverse the nodes in the vertex set in Topological order, if U is in front of V in Topological sorting order and have a directed edge to V, DP(U) will have a value, and the recurrence will consider all the candidates that have a directed edge to V and return the maximum. So DP(V) will be the longest path to V, and the largest value of all DP(V) will be the longest path in the DAG.

Time complexity:

To sort a single box will take $O(dlogd)$, so sorting n boxes takes $O(ndlogd)$. To determine all boxes that whether a sorted box nests within another takes $O(d)$, constructing a graph needs to check all the edges so it will take $O(dn^2)$. Topological sort takes $O(|V| + |E|) = O(n + n^2) = O(n^2)$. So the total time complexity is:

$$T(n) = O(ndlogd) + O(dn^2) + O(n^2) = O(ndlogd + dn^2)$$

Problem 2: Business plan

Algorithm description:

We are given n possible projects and we are asked to provide a business plan that maximize the final amount of capital $C_{k'}$.

We can use the priority queue and sorting to solve this problem. We first make each c_i and its corresponding p_i a pair $\{c_i, p_i\}$. Then we sort all the pairs in the order of non-decreasing c_i. Suppose the sorted pairs is P.

We use a priority queue to store the pairs, where the priority queue makes the pair with lower c_i stays ahead. We initialize by pushing all pairs in P whose $c_i \leq C_0$, and we memorize the position of the last pushed in pairs.

Each time we pop the top pair $\{c_i, p_i\}$ of the priority queue Q, and it is the project we are going to start, so we can push this project in into the output. Then we also make the current capital

$$C_j = C_{j-1} + p_i \quad for\ 1 \leq j \leq k' - 1$$

Next, we push the rest pairs in P whose $c \leq C_j$ into the priority queue Q. And we go back to popping the top pair of Q again.

The algorithm is ended when the times of popping reaches k' or the priority queue is empty and we cannot pop any pair any more.

Proof of correctness:

This algorithm is essentially a Greedy algorithm. We assume $G = \{g_1, g_2, \dots, g_j\}$ is a solution generated from the Greedy algorithm, and $O = \{o_1, o_2, \dots, o_l\}$ is an arbitrary feasible solution.

We first check that if G is empty, since the algorithm is based on sorted pairs, there will be no project with $c_i \leq C_0$, and O is empty too.

Next, we can assume that until the index $i - 1 (i \geq 1)$, the two solution G and O are the same. According to our greedy algorithm, we have $p_{g_i} \geq p_{o_i}$, and thus $C_{g_i} > C_{o_i}$ which means the greedy solution G can allow more projects to be started except for the one that has been already completed. For the next step, solution G can start project o_i if p_{o_i} is the largest among the projects that can be start, or choose another new projects that has the largest profit. On the other side, solution O can select g_i if it has the largest profit, or choose another from the available projects set. However, the project set of O will be smaller than that of solution G. So swapping o_i with g_i will not decrease the profit of solution O.

If we continue swapping, we can finally eliminate all difference between G and O in a polynomial number of steps without worsening the quality of the solution. Thus, the greedy solution is as good as any optimal solution, and hence is optimal itself.

Time complexity:

We first construct the and sort the pairs, which takes $O(n)$ and $O(nlogn)$. Then we traverse the pairs no more than once, and push into or pop from priority queue of the pairs, each takes $O(logn)$, so the worst case is to push and pop each pair twice, and takes $O(nlogn)$. Therefore, the total time complexity is:

$$T(n) = O(n) + O(nlogn) = O(nlogn)$$

Problem 3: Scheduling to minimize average completion time (CLRS)

Question 1

Algorithm description:

We are given a set of tasks and we are asked to schedule the tasks non-preemptively so as to minimize the average completion time. We use the Greedy algorithm, which is to first make tasks and their corresponding process time pairs. Then we sort the pairs in the order of non-decreasing process time. The tasks order in the sorted pairs are the schedule we want.

Proof of correctness:

Let $G = \{g_1, g_2, \dots, g_n\}$ be the solution generated by the Greedy algorithm, and let $O = \{o_1, o_2, \dots, o_n\}$ be an arbitrary feasible solution. Since the number of tasks is certain, the minimum average completion time refers to the minimum sum of completion time.

We assume there are two consecutive tasks o_i and o_{i+1} in O in a different order than they are in G, and anything else in front of these two tasks are the same in G and O. Let's assume the start time of o_i in O is T_1.

The sum of completion time of tasks o_i and o_{i+1} in O is:

$$c_{o_i} + c_{o_{i+1}} = T_1 + p_{o_i} + T_1 + p_{o_i} + p_{o_{i+1}}$$

And the sum of completion time of tasks o_{i+1} and o_i in G is:

$$c_{g_i} + c_{g_{i+1}} = T_1 + p_{o_{i+1}} + T_1 + p_{o_{i+1}} + p_{o_i}$$

Since from the greedy algorithm, we know that $p_{o_{i+1}} \leq p_{o_i}$, swapping tasks o_i and o_{i+1} in O will not increase the sum of completion time. So if we continue swapping, we can eliminate all differences between O and G in a polynomial number of steps without worsening the quality of the solution. Thus, the greedy solution is as good as any optimal solution, and hence is optimal itself.

Time complexity:

The algorithm requires to sort the tasks in the order of non-decreasing processing time, so the time complexity for this problem is

$$T(n) = O(nlogn)$$

Question 2

Algorithm description:

We now consider the tasks with release time and task can be implemented preemptively. We can still develop an algorithm based on Greedy with some additional implementation.

We first make pairs of tasks and their corresponding process time and release time. Then we sorted the pairs in the order of non-decreasing release time. Let's assume the pairs $\{a_i, p_i, r_i\}$ $for\ 1 \leq i \leq n$ are well sorted. We also use a priority queue to store pairs, which makes pairs with smaller process time stays in front.

We use an integer T to represent the current time and initialize with $T = 0$, and we also first push pair $\{a_1, p_1, r_1\}$ into the priority queu.

We pop the top pair of the priority queue as the task we are currently doing, when a new release time comes, we first decide whether the process time of the new pair is smaller than the rest of the process time of the current task. If the process time of the new pair is smaller, we change the process time of the current task to be the rest of its process time, and push the pair of it into the priority. Then we use the new pair as the task we are going to do. Otherwise, we continue doing the current task and push the new pair into the priority queue.

The algorithm ends when we traverse all pairs and the priority queue is empty.

Proof of correctness:

We have proven that if we schedule task non-preemptively, we should do the tasks with smaller process time first so as to achieve minimum average completion time.

We now have a priority queue which store pairs with smaller process time in the front, so the tasks popped from the priority will be exactly the one that we should be doing to achieve minimum completion time.

At each time when we reach a new release time of pair $\{a_j, p_j, r_j\}$, there are one more task that we can start. We can consider the task we are currently doing is $\{a_i, p_i', r_i\}$, where p_i' is the rest of the process time. Since by the definition of the priority queue, the tasks in the priority queue are impossible to have smaller process time than the task we are currently doing, we can only compare the rest of the process time p_i' with p_j of the new coming task. Then as what we proved in the last Question, if the $p_j < p_i'$, then by switching to do the new coming task a_j will not increase the average

completion time. If $p_i' \leq p_j$, then sticking to do the current task a_i will not increase the average completion time.

Time complexity:

The algorithm first makes pairs of the tasks, process time and release time, which takes $O(n)$. Then we need sort the pairs and it takes $O(nlogn)$. Finally we need to push in or pop from the priority queue, each operation takes $O(logn)$ and each time a new release time comes, we at most need to push and pop each for once. Therefore, the total time complexity of the algorithm is

$$T(n) = O(n) + O(nlogn) = O(nlogn)$$

Problem 4: Shortest wireless path sequence (KT 6.14)

Question 1

Algorithm description:

We are given a set of known graphs $\{G_1, G_2, \dots, G_b\}$ and we are asked to choose a single P that is shortest and is an s-t path in each of the graph.

Since all the graphs in $\{G_1, G_2, \dots, G_b\}$ are already known, we can build a graph $H = (V, E')$ where H involves all the nodes in G_i and the edges set of it E' involves all the s-t edges that appear in each edge set E_i. Then we apply breadth-first search to find the shortest path in H.

Proof of correctness:

The algorithm is actually pretty straightforward, since we are asked to find a path P that appears in each of the graph, the graph we build H should include all the possible candidates for the result. Then we apply BFS in the graph H and we can find the shortest path, and that will be the result that meets the requirements.

Time complexity:

To build the graph H will take $O\left(|V| + \sum_{i=1}^{b}|E_i|\right) = O(n + bn^2)$ and BFS takes $O(|V| + |E'|)$. Therefore, the total time complexity of the algorithm is

$$T(n) = O(bn^2)$$

Question 2

Algorithm description:

We are asked to provide a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \dots, P_b$ of minimum cost. We can solve this problem by applying Dynamic Programming method.

We first define that E_{ij} is the edge set that involves all the s-t edges that appear in each edge set from $\{E_i, \dots, E_j\}$. Then we can build the graph $G_{ij} = (V, E_{ij})$. We can compute the length of shortest path in each G_{ij} based on the algorithm mentioned in Question 1 and mark it as $P(i, j)$, and $P(i, j) = \infty$ specifically if no such path exists.

We define $DP[i]$ to be the minimum cost of the paths $P_i, P_{i+1}, \dots, P_b$, and define j to be the first index where $P_j \neq P_{j+1}$. The base cases we assume are $DP[b] = P(b, b)$ and $DP[b+1] = -K$. Then the recurrence is as below:

$$DP[i] = min_{1 \le j \le b}\{(j - i + 1) * P(i, j) + DP[j + 1] + K\}$$

Finally DP[0] is the minimum cost of the paths $P_0, P_{i+1}, \dots, P_b$.

Proof of correctness:

We define *DP*[i] as the minimum cost of the paths $P_i, P_{i+1}, \ldots, P_b$. There are two base cases, $DP[b] = P(b, b)$ means that if we choose the graph G_b, then the minimum cost will be the shortest path in the last graph G_b, and $DP[b+1] = -K$ is just set to make the computation convenient.

Next, we are going to prove the recurrence

$$DP[i] = min_{1 \le j \le b}\{(j - i + 1) * P(i, j) + K + DP[j + 1]\}$$

where *j* is the first index where $P_j \neq P_{j+1}$.

From our definition, $P_k = P_{k+1}$ for $i \le k < j$, and since $P_j \neq P_{j+1}$, the cost should be added by *K*, and for the cost caused by the paths from P_{j+1} to P_b, we can add a term $DP[j + 1]$, since it's the minimum cost of $P_{j+1}, P_{j+2}, \ldots, P_b$.

Since *DP*[i] is a minimizer and we set j from 1 to b and find the minimum cost, *DP*[i] will have considered all the candidates, from assuming the first path P_i is different from the next path P_{i+1} to assuming all the paths are the same, which is $P(i, b)$. Therefore, this Dynamic Programming algorithm can return the minimum cost from P_i to P_b.

Time complexity:

The algorithm from question 1 takes $O(bn^2)$. And we compute *DP*[i] from *b* to 0 with each time computing the minimum from *i* to *b*, so the total time complexity is:

$$T(n) = O(b^3 n^2)$$

Problem 5: Selling shares of stock (KT 6.25)

Algorithm description:

We are given a set of prices $\{p_1, .., p_n\}$ and the function $f(\cdot)$ and we are asked to provide the best way to sell x shares of stock by day *n*. We can solve this problem by applying 2-dimensional Dynamic Programming method.

We first define *DP*[*i*][*j*] as the maximum profit of selling *i* shares from day *j*. And the base cases are the profits that we sell the rest shares of stock in the last day, which can be expressed as below:

$$DP[i][n] = p_n \cdot i - f(i) \cdot i \qquad \text{for } 0 \le i \le x$$

The recurrence is

$$DP[i][j] = max_{0 \le s \le i}\{p_j s - f(i) \cdot i + DP[i - s][j + 1]\}$$

We start the recursion from *j* = n-1 to 1and for each *j*, compute the value of $DP[i][j]$ from *x* to 0. And finally we return $DP[x][1]$ as the output.

Proof of correctness:

We first define *DP*[*i*][*j*] as the maximum profit of selling *i* shares from day *j*, and we define the last column as the base cases. Because on the last day *n*, we need to sell all the remaining shares of stock so as to ensure the maximum profit, the equation of the base case

$$DP[i][n] = p_n \cdot i - f(i) \cdot i \qquad \text{for } 0 \le i \le x$$

is reasonable.

Next, we are going to prove the recurrence

$$DP[i][j] = max_{0 \le s \le i}\{p_j s - f(s) \cdot i + DP[i - s][j + 1]\}$$

On day *j*, we are planning to sell *i* shares of stock. If we sell *s* shares on day *j*, we will have $p_j s$ profit for the *s* shares minus $f(s) \cdot i$, which is the loss caused by the large sales of *s* shares. The reason why this term is not $f(s) \cdot s$ is that the price drop will continue to affect the following sales. After selling *i* shares on day *j*, the maximum profit we can gain from day *j* + 1 is $DP[i - s][j + 1]$ by definition.

Since *DP*[*i*][*j*] is a maximizer, and we set *s* from 0 to *i*, *DP*[*i*][*j*] will have considered all the candidates and thus be the maximum profit of selling *i* shares from day *j*. *DP*[*x*][1] means selling x shares of stock from day 1, so it's the result we want.

Time complexity:

We first build base cases for this DP algorithm, which takes $O(x)$. Then we compute the value of each $DP[i][j]$ each can at worst takes x comparisons, so it takes $O(x^2 n)$. Therefore, the total time complexity of this algorithm is

$$T(n) = O(x^2 n)$$