Problem 1: Nesting boxes (CLRS)

A d-dimensional box with dimensions $(x_1, x_2, ..., x_d)$ nests within another box with dimensions $(y_1, y_2, ..., y_d)$ if there exists a permutation π on $1, 2, ..., d$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, ..., x_{\pi(d)} < y_d$.

1.1: Argue that the nesting relation is transitive.

Claim 1. *Let A, B, C be three d-dimensional boxes with dimensions $(a_1, a_2, ..., a_d)$, $(b_1, b_2, ..., b_d)$, $(c_1, c_2, ..., c_d)$ respectively. If box A nests inside box B and box B nests inside box C, then box A nests inside box C.*

Proof. Let $\pi_{A,B}$ be a permutation such that $a_{\pi_{A,B}(1)} < b_1, a_{\pi_{A,B}(2)} < b_2, ..., a_{\pi_{A,B}(d)} < b_d$. Let $\pi_{B,C}$ be a permutation such that $b_{\pi_{B,C}(1)} < c_1, b_{\pi_{B,C}(2)} < c_2, ..., b_{\pi_{B,C}(d)} < c_d$.

Let $1 \leq i \leq d$, $j = \pi_{B,C}(i)$, we can create a permutation $\pi_{A,C}$ where $\pi_{A,C}(i) = \pi_{A,B}(j)$. We have $a_{\pi_{A,C}(i)} = a_{\pi_{A,B}(j)} < b_j = b_{\pi_{B,C}(i)} < c_i$, thus according to definition, A nests inside C. □

1.2: Describe an efficient method to determine whether or not one d-dimensional box nests inside another.

Algorithm:

Let the two boxes be A and B, for each of them, sort the dimensions in ascending order, then for i from 1 to d, check whether $a_i < b_i$. If there exists i such that $a_i \geq b_i$, then A cannot nest inside B, otherwise A nests inside B.

Proof of correctness:

Claim 2. *$\exists\ i,\ a_i \geq b_i. \iff$ A cannot nest inside B.*

Proof. $\Rightarrow$: If $a_i \geq b_i$, and the dimensions are sorted in ascending order, then for all indexes $j > i$, $a_j \geq b_i$, meaning none of those dimensions in A can nest inside b_i. Apparently, they cannot nest inside $b_k (1 \leq k < i)$ either. Since there are a total number of $d - i + 1$ dimensions and only $d - i$ possible remaining positions to nest in, it is guaranteed that not all pf them can be nested. Hence A cannot nest inside B.

$\Leftarrow$: We prove the contrapositive. If $\forall i, a_i < b_i$, then a permutation can be established by simply pairing all a_i with b_i, thus by definition A nests inside B. □

Time complexity:

We sort the two arrays, which takes $O(dlogd)$ time, then we compare two arrays by indexes, which takes $O(d)$. So the total time complexity is $O(dlogd)$.

1.3: Suppose that you are given a set of n d-dimensional boxes $B_1, B_2, ..., B_n$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, ..., B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, ..., k-1$. Express the running time of your algorithm in terms of n and d.

Algorithm:

Using the algorithm described in the previous question, we first sort the dimensions for all n boxes, then we determine whether for any two boxes there is a nesting relationship, i.e., B_i can nest inside B_j or B_j can nest inside B_i. After this, we get a directed graph $G = (V, E)$ where an edge (u, v) means box B_u nests inside box B_v. G is acyclic because of Claim 1. We then need to determine what is the longest path in G as that will be the longest sequence of nesting relationships.

We can do this by topologically sorting the nodes. Specifically, we maintain an in-degree count for all the nodes. We iteratively process nodes with an in-degree of 0 (assume it's u): subtract one from the in-degrees of all nodes that u points to, and if a node v reaches an in-degree of 0, we mark its predecessor as u and append v to the list for next iteration. After processing all the nodes we should reach the tail of a longest path, and we can recover the complete node sequences based on predecessors.

Proof of correctness:

Claim 3. *The algorithm above gives the longest sequence of nesting relationships.*

Proof. We prove by contradiction. Let's assume our algorithm returns $\langle B_{i_1}, B_{i_2}, ..., B_{i_k} \rangle$, and the longest sequence is $\langle B_{s_1}, B_{s_2}, ..., B_{s_q} \rangle$ $(q > k)$. Clearly B_{s_1} has an in-degree of 0, otherwise the sequence could be made longer by considering B_{s_1} 's predecessors. After the first iteration, both B_{s_1} and B_{i_1} are removed from G, now again both B_{s_2} and B_{i_2} have an in-degree of 0 and they will be removed in the next iteration. Generally, we see that a pair (B_{s_j}, B_{i_j}) will be removed in the same iteration. Hence after k iterations there still will be nodes $\langle B_{s_{k+1}}, ..., B_{s_q} \rangle$ which are not processed so the algorithm will continue to process them instead of terminating. This is contradictory to our assumption. □

Time complexity:

Sorting n arrays of dimensions takes $O(ndlogd)$ time. We need to compare all pairs of boxes so that is $O(n^2 d)$. Topological sort takes $O(n^2)$ time because in worst cases $|E| = O(|V|^2)$ and we have to traverse each edge once. Thus, the total time is $O(nd \max(logd, n))$.

Problem 2: Classes and rooms

Input: Class enrollment list $E = (e_1, e_2, ..., e_n)$, where e_i represents the enrollment of class i, $1 \leq i \leq n$. Room capacity list $S = (s_1, s_2, ..., s_m)$, where s_j represents capacity of room j, $1 \leq j \leq m$.

Output: Assignment list $A = (a_1, a_2, ..., a_n)$, $1 \leq a_i \leq m$. Each a_i is unique, $\forall i, s_{a_i} \geq e_i$ and $\sum_{a_i} s_{a_i}$ is minimized. If no such assignment exists, output *None*.

Algorithm:

We assume $m \geq n$, otherwise there won't be an assignment to fit all classes and we can just return *None*. First we sort both E and S in ascending order. Let the order be p and q respectively. Starting with the first element in sorted E and the first element in sorted S. If $e_{p(1)} \leq s_{q(1)}$ then we assign room $q(1)$ to class $p(1)$, increase both p and q indexes by 1, and compare the next pair. Otherwise, we only increase the index of q by 1 and compare the next pair (e.g., compare $e_{p(1)}$ with $s_{q(2)}$). After assigning all classes we return the assignment. If we exhaust the room list before assigning all classes, we will return *None*.

Proof of correctness:

Claim 1. *If the algorithm returns an assignment, it will have the minimum $\sum_{a_i} s_{a_i}$ among all possible assignments.*

Proof. We prove by induction that for all k $(1 \leq k \leq n)$, the algorithm finds the assignment that has the minimum $\sum_{a_{p(i)}} s_{a_{p(i)}}$ $(1 \leq i \leq k)$ for class $p(1)$ through $p(k)$.

Base case: Because lists are sorted in ascending order, when scanning from left to right, the first room that has a size big enough to hold class $p(1)$ will be the smallest one among all rooms that can hold class $p(1)$. So for $k = 1$ the assignment is optimal.

Inductive step: If the assignment for class $p(1)$ through $p(i)$ $(1 \leq i \leq k)$ is optimal for all i, we argue that according to the algorithm, the assignment for $p(1)$ through $p(k+1)$ is also optimal.

Let the room we assigned to $p(k)$ be $q(k')$. There are two cases:

1. $e_{p(k+1)} > s_{q(k')}$. In this case we can only choose from room $q(k'+1)$ to $q(m)$. As we scan from left to right, the first one that can hold class $p(k+1)$ is the optimal choice. This is the same as what the algorithm does.

2. $e_{p(k+1)} \leq s_{q(k')}$. In this case the algorithm chooses room $q(k'+1)$.

   For any room $q(j)$ $(1 \leq j \leq k')$, if it is unassigned, then we have $s_{q(j)} < s_{q(k')}$ because the algorithm won't skip a room and choose the one behind it which has the same capacity. We will also have $s_{q(j)} < e_{p(k)}$ because according to the fact that we already have an optimal assignment for class $p(1)$ through $p(k)$, we won't be able to assign $q(j)$ to $p(k)$ to reduce total capacity. Since $e_{p(k+1)} \geq e_{p(k)}$, we won't be able to assign $q(j)$ to $p(k+1)$ either.

If $q(j)$ is already assigned to $p(i)$, we can potentially re-assign $q(j)$ to $p(k+1)$. To see why this won't produce a smaller total capacity, we first argue that if $e_{p(k+1)} \leq s_{q(j)}$, then room $q(j)$ through $q(k')$ must have all been assigned. This is for the same reason that the algorithm won't skip rooms as discussed above. Since they are all assigned, after replacing $p(i)$ with class $p(k+1)$, we have to allocate $p(i)$ again. We can swap $p(i)$ with any other classes whenever possible but that won't solve the problem that there will always be an extra class that we have to re-assign. If we search to the left, we find that the analysis we did above for $p(k+1)$ applies also to $p(i)$, that is, we either find a free room with capacity less than $e_{p(i)}$, or we cannot find a free room at all. Thus we can only resort to searching to the right, in which case room $q(k'+1)$ is the best available choice.

□

Claim 2. *If the algorithm returns None, there is no assignment that has a unique mapping and in the same time satisfies* $\forall i, s_{a_i} \geq e_i$.

Proof. If $m < n$, it's trivial to see that no unique mapping exists.

If $m \geq n$, let the first class that is not assigned be $p(i)$. There are two cases:

1. $e_{p(i)} > s_m$. The largest room cannot hold $p(i)$, so there isn't an assignment.
2. $e_{p(i)} \leq s_m$. In this case s_m must have been assigned to $p(i-1)$. And we won't be able to fit class $p(i)$ in. The proof of this is identical to the proof of the second case in Claim 1 where $e_{p(k+1)} \leq s_{q(k')}$.

□

Time complexity:

Determining whether $m \geq n$ can be done by counting in $O(m)$ time. In the case that $m \geq n$, we first sort two lists, which takes $O(mlogm)$ time, then we compare sorted lists pair-wisely in one pass which takes $O(m)$ time in worst case. Hence the total time complexity is $O(mlogm)$.

Problem 3: Business plan

Input: Capital list $M = (c_1, ..., c_i, ..., c_n)$. Profit list $P = (p_1, ..., p_i, ..., p_n)$. Initial capital C_0. Maximum number of projects k.

Output: A list of up to k distinct projects, $i_1, ..., i_{k'}$. Accumulated capital after completing the project i_j is $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy $C_j \geq c_{i_{j+1}}$ for each $j = 0, ..., k' - 1$. And the final amount of capital, $C_{k'}$, is maximized.

Algorithm:

Sort projects w.r.t. c_i in ascending order. Initialize a priority queue based on profit (larger profit in the front). Linearly scan from left to right, enqueue all projects that have $c_i \leq C_0$, stop when $c_i > C_0$. Dequeue one project, consider it done and append it to the output list. We then update our accumulated capital to include the last profit. And we enqueue new projects with a reachable capital requirement. We iterate until we have done k projects or we cannot find a project whose minimal capital requirement is reachable within our accumulated capital.

Proof of correctness:

Claim 1. *The algorithm returns a sequence of projects where $C_{k'}$ is maximized.*

Proof. We prove by exchanging the projects in optimal solution with our solution and show that the optimal solution will not degrade, thus our solution is the optimal solution.

Assume an optimal solution, $(i_1^*, ..., i_{k''}^*)$ exists that has a final accumulated capital of $C_{k''}^* \geq C_{k'}$. Let the first project that differs in two sequences be i_j and i_j^*. Since $C_{j-1} = C_{j-1}^*$, the total number of projects that has a budget under C_{j-1} is the same for two solutions when choosing the jth project. For our solution, that subset of projects is consisted of all projects in the queue plus the projects we already selected. Among them, we select the one in queue with the maximum profit. So we can safely conclude that $p_{i_j} \geq p_{i_j^*}$, and $C_j \geq C_j^*$. If we replace i_j^* with i_j, it will not affect subsequent choices since it will not shrink the set of projects to choose from when selecting the $(j+1)$th project. The only special scenario is that i_j was already chosen in the optimal solution, say $i_j = i_h^*$, $j < h \leq k''$, in that case we swap i_j^* with i_h^*, i.e., with i_j, thus also maintaining $C_{k''}^*$ unchanged. Either way, the exchange iteration can proceed all the way to the end and cannot degrade the performance of optimal solution.

At the end, we argue that k'' cannot be smaller that k' because otherwise it will have a smaller final capital. It cannot be larger either because according to the algorithm, we would have done k projects already or no project with a permissible budget exists and we have to terminate. Hence $k'' = k'$ and our solution is the optimal solution. □

Time complexity:

Sorting takes $O(nlogn)$ time. Afterwards, at most n projects are enqueued and k ($k \leq n$) projects are dequeued. Either operation in a priority queue takes $O(logn)$ time. Thus the total time complexity for the algorithm is $O(nlogn)$.

Problem 4: Shortest wireless path sequence (KT 6.14)

Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on, to an edge set E_b. For $i = 0, 1, 2, ..., b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse", it would look precisely like the sequence of graphs $G_0, G_1, G_2, ..., G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now consider two particular nodes $s, t \in V$. For an $s - t$ path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and we denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, ..., P_b$ so that for each i, P_i is an $s - t$ path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes*–points at which the identity of the path switches. Formally, we define $changes(P_0, P_1, ..., P_b)$ to be the number of indices $i (0 \leq i \leq b - 1)$ for which $P_i \neq P_{i+1}$.

Fix a constant $K > 0$. We define the cost of the sequence of paths $P_0, P_1, ..., P_b$ to be

$$cost(P_0, P_1, ..., P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot changes(P_0, P_1, ..., P_b).$$

4.1: Suppose it is possible to choose a single path P that is an $s - t$ path in each of the graphs $G_0, G_1, ..., G_b$. Give a polynomial-time algorithm to find the shortest such path.

Algorithm:

We take the intersection of all edge sets, $E_I = \cap_{i=0}^{b} E_i$, and form a graph $G_I = (V, E_I)$. Starting at s, we do a Breadth First Search in G_I to find the shortest path to t and return it.

Proof of correctness:

Claim 1. *If there exists a path P that is a $s-t$ path in each of the graphs $G_0, G_1, ..., G_b$, the above algorithm returns the shortest such path.*

Proof. If P is in $G_0, G_1, ..., G_b$, then all edges in P must be present in all graphs, therefore P must be in G_I. Therefore if we do a BFS in G_I starting from s, we are guaranteed to find t and by property of BFS we find the shortest $s - t$ path among all P. □

Time complexity:

Taking the intersection of edges for two graphs can cost $O(|V|^2)$ in the worst case since we have to compare every edge. And we do this b times so it's $O(b|V|^2)$. BFS in G_I also takes $O(|V|^2)$ in the worst case. Hence the total time complexity is $O((b + 1)|V|^2)$. Note that b can be 0.

4.2: Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, ..., P_b$ of minimum cost, where P_i is an $s-t$ path in G_i for $i = 0, 1, ..., b$.

Algorithm:

We solve this by Dynamic Programming.

DP definition: Let $c(j)$ be the minimum cost of the sequence of paths $P_0, P_1, ..., P_j$ $(0 \le j \le b)$. And let π_j be the sequence that has minimum cost.

Recurrence formulation: We denote the shortest common path we find for graph $G_i, G_{i+1}, ..., G_j$ using last question's algorithm as $P^*_{i,j}$ $(0 \le i \le j)$. If $P^*_{i,j}$ doesn't exist, we set $\ell(P^*_{i,j}) = +\infty$. For convenience, we define $c(-1) = -K$, and $\pi_{-1} = \emptyset$. We have:

$$c(j) = \min_{0 \le i \le j} [c(i-1) + (j-i+1) \times \ell(P^*_{i,j}) + K].$$

Once we have computed $c(j)$, we also know the change-point i as *argmin*, and we append $j-i+1$ copies of $P^*_{i,j}$ to π_{i-1} to form π_j. After we iterate j from 0 to b, we return π_b as the answer.

Proof of correctness:

Claim 2. *The above algorithm correctly computes $c(j)$ as the minimum cost of the sequence of paths $P_0, P_1, ..., P_j$ $(0 \le j \le b)$.*

Proof. We prove this by induction.

Base case: $c(0) = \ell(P^*_{0,0})$, which is essentially the shortest $s-t$ path length in G_0. Since there is no changes in paths, this is the minimum cost.

Inductive step: If all $c(i)$ $(0 \le i < j)$ are correctly computed, we argue that $c(j)$ is correctly computed too. Because the optimal sequence of $P_0, P_1, ..., P_j$ either has $P_i = P_{i+1} = ... = P_j$ and $P_{i-1} \ne P_i$ for some i from 1 to j, or has no change-points. In the first case, the cost of the sequence is split into three parts: the cost for sequence up to P_{i-1} which is exactly $c(i-1)$, $j-i+1$ times the length of P_j, and a constant K. The optimal sequence must have minimum $\ell(P_j)$, hence $P_j = P^*_{i,j}$. In the second case, the cost is simply $j+1$ times $\ell(P^*_{0,j})$. By considering all cases and all possibilities of i and taking the minimum of them, we are guaranteed to find the optimal sequence and the minimum cost $c(j)$. □

Time complexity:

For every j, as we iterate i backwards from j to 0, we iteratively build the intersection set of E_j through E_0. For every iteration, we compute the intersection of two edge sets, we do this at most j times. We also compute $\ell(P^*_{i,j})$ at most $j+1$ times. So for every j, the time we take to compute $c(j)$ is $O((j+1)|V|^2)$. Now as we add them up, we get $O((b+1)^2|V|^2)$ total time complexity. Note that b can be 0.

Problem 5: Untangling signal superposition (KT 6.19)

Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Algorithm:

We solve this by Dynamic Programming.

DP definition: Let the length of string s be n, length of x be m_x, and length of y be m_y. Let $dp(i, p_x, p_y)$ be a Boolean value, it is *True* if and only if the prefix of s of length i can be partitioned into s' and s'', where s' equals to x^k concatenated with a prefix of length p_x of x, and s'' equals to y^l concatenated with a prefix of length p_y of y ($0 \leq i \leq n, 0 \leq p_x \leq m_x - 1, 0 \leq p_y \leq m_y - 1, 0 \leq k, 0 \leq l$).

Recurrence formulation: For convenience, we define a function *lastIndex(p)* which takes in an index number and returns the index before it. If p_x or p_y is greater than 0, this function will just return $p_x - 1$ or $p_y - 1$. Otherwise, it returns the largest index in x or y, i.e., $m_x - 1$ for x and $m_y - 1$ for y. Assume all strings are 0-indexed and let $string[p]$ denote the $(p+1)$th digit in string.

We initialize $dp(0, 0, 0) = True$ and $dp(0, p_x, p_j) = False$ if $p_x \neq 0$ or $p_j \neq 0$. For all other i,

$$\begin{aligned} dp(i, p_x, p_y) = &\big(dp(i-1, lastIndex(p_x), p_y) \ AND \ s[i-1] = x[lastIndex(p_x)]\big) \ OR \\ &\big(dp(i-1, p_x, lastIndex(p_y)) \ AND \ s[i-1] = y[lastIndex(p_y)]\big). \end{aligned}$$

We iterate i from 1 to n, in every iteration, we compute $dp(i, p_x, p_y)$ for all combinations of p_x and p_y. At the end we take OR operation of all $dp(n, p_x, p_y)$ and return the result.

Proof of correctness:

Claim 1. *$\forall(i, p_x, p_y)$, the above algorithm correctly computes $dp(i, p_x, p_y)$.*

Proof. We prove this by induction.

Base case: When $i = 0$, the prefix of s is an empty string, thus it can only be partitioned into two empty strings, so $dp(0, 0, 0)$ should be $True$ and $dp(0, p_x, p_j) = False$ if $p_x \neq 0$ or $p_j \neq 0$.

Inductive step: If for all (p_x, p_y), $dp(i-1, p_x, p_y)$ is correctly computed, we argue that for all (p_x, p_y), $dp(i, p_x, p_y)$ is correctly computed too. Because for the ith digit in s, we have two options:

- We append it to s' and form a prefix of x of length p_x. To be valid, it requires the original s' to have a prefix of length $lastIndex(p_x)$ at the tail. Also the $(lastIndex(p_x)+1)$th digit in x has to match the ith digit in s. And, the prefix of s of length $i-1$ must be an interleaving of x and y. Specifically, s'' should also have a prefix of y of length p_y at the tail, since it

is unchanged. This constitutes the first part of the Boolean expression in our DP recursive equation.

- We append it to s'' and form a prefix of y of length p_y. Analysis is identical. This constitutes the second part of the Boolean expression after OR in our DP recursive equation.

By taking OR of the outcomes of those two options, we know we will have a successful partition if at least one option is valid. Otherwise, it's guaranteed there doesn't exist such a partition since we cannot assign the ith digit in s. Hence by definition $dp(i, p_x, p_y)$ is correctly computed. □

Claim 2. *The above algorithm returns True if and only if s is an interleaving of x and y.*

Proof. By Claim 1, we know all $dp(n, p_x, p_y)$ are correctly computed, hence if there exists a partition of s that satisfies the requirement, one of $dp(n, p_x, p_y)$ must be *True*, so taking OR of them will give *True*, and vice versa. □

Time complexity:

We iterate i from 1 to n, in every iteration, we compute $dp(i, p_x, p_y)$ for all (p_x, p_y), each computation takes $O(1)$ time. So total time complexity is $O(m_x m_y n)$.