Problem1:

1 Suppose we have three d-dimensional boxes, X, Y and Z that X nested within Y and Y nested within Z. Therefore there exists a permutation, we name it $\pi 1$, on *(1,2,…,d)* such that $x_{\pi1(1)} < y_1, x_{\pi1(2)} < y_2, \ldots, x_{\pi1(d)} < y_d$. And there exists a permutation $\pi 2$, on *(1,2,…,d)* such that $y_{\pi2(1)} < z_1, y_{\pi2(2)} < z_2, \ldots, y_{\pi2(d)} < z_d$. Let $\pi 3 = \pi 1 \cdot \pi 2$, we have $x_{\pi3(i)} = x_{\pi2(\pi1(i))} < y_{\pi2(i)} < z_i$, for all $1 \le i \le d$. Thus X is nested in Z which shows that the nesting relation is transitive.

2 **HLD:** If we have two d-dimensional boxes X and Y, we will first sort all elements in each box in non-decreasing order, that is, we change the original $X = (x_1, x_2, \ldots, x_d)$ to $(x'_1, x'_2, \ldots, x'_d)$ where $x'_i \le x'_j$ for all $i \le j$. We do the same operation to Y.

Then we compare x'_i with y'_i for all $1 \le i \le d$. If for all $1 \le i \le d$, we have $x'_i < y'_i$, we can say that X nested within Y or if for all $1 \le i \le d$, we have $x'_i > y'_i$, then Y nested within X. In other situations, there is not nesting relation between the two boxes.

Proof of Correctness: Suppose we have two d-dimensional boxes X and Y that are sorted in non-decreasing order, we already have $x'_i < y'_i \ for 1 \le i < k \ (k \le d)$. If $x'_k > y'_k$, X cannot nest within Y because $x'_j \ge x'_k > y'_k$ for all $k \le j \le d$. We can only find dimensions in Y that is larger than x'_k between k and d. Let's assume $x'_k < y'_r$ (r >k) and $\pi(r) = k$. There will be a dimension y'_k that has no dimension in X to match since all dimensions in X between k and d is larger than x'_k and thus larger than y'_k. There is no permutation π that could make $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

Complexity Analysis: The sorting process can be computed in *O(dlogd)*. After sorting, we compare sorted dimensions in X and Y one by one which takes *O(d)*. So the entire time complexity is *O(dlogd+d)=O(dlogd)*.

3 **HLD: Sort dimensions in every box and find all boxes that $B_i (1 \le i \le d)$ nests within.** We first sort the dimensions in every box for further determine if one box nests in another. We already proved that the nesting relation is transitive. If box B_i nests inside B_j and B_j nests in B_k, then B_i nests within B_k. If we want to find the longest sequence $< B_{i_1}, B_{i_2}, \ldots, B_{i_k} >$, we need to know all the boxes that B_i $(1 \le i \le d)$

nested in. We could use a directed graph to find the sequence(path). The graph should be directed because if B_i nests in B_j, the path can only go from B_i to B_j and cannot be reversed.

Build graph. We build a graph G=(V, E) with $v=\{s, v_1, v_2, \ldots, v_n, t\}$ where v_i denotes B_i and *s* denotes source, *t* denotes terminal. For a box B_i, we find all the boxes that B_i nests in through the method we come up in part 2. If B_i nests in B_j, we add an edge between v_i and v_j, let $(v_i, v_j) \in E$ and assign it with a weight $w=1$. We also add two edges (s, v_i) and (v_i, t) for each vertex with weight equals to 0. Vertices *s* and *t* are used as entrance and exit for our path(sequence). After adding all the edges, we build up the graph.

Find longest path. We define two array $d[n]$ and $\pi[n]$ to denote the distance (max weight) between vertex *s* and v_i and record the preceding vertex of v_i respectively. We initialized $d[v_i]=1$ and $\pi[v_i]=s$ for $1 \le i \le d$. And then we continuously update the maximum distance between *s* and v_i through the following algorithm and record the preceding vertex. In order to ensure the correctness, vertices $v_1, v_2, \ldots, v_n$ has already been sorted based on the first dimension of the corresponding box.

```
for each vertex v_i in V:
    check all v_j that (v_i, v_j) ∈ E;
        if d[v_j]<d[v_i]+w[v_i, v_j]:
            d[v_j]=d[v_i]+w[v_i, v_j]
            π[v_j]=v_i
```

Through the algorithm above, we could find the longest path with the maximum weight between s and t. The vertices within the path is the longest sequence of nesting boxes.

Proof of Correctness:

We already prove that the nesting relation is transitive. Therefore in the graph, if there is a path between v_i and v_j, then v_i must nest within v_j.

Claim 0.1. The directed graph is acyclic.

If there are two boxes X and Y that X nests in Y, we sort the d dimensions of X and Y in non-decreasing order. Then for each dimension, $x_i<y_i$ for $1 \le i \le d$. Y cannot nest in X because there is no dimension in X satisfying $x_i>y_d$. In the directed graph, if $(v_i, v_j) \in E$, then $(v_j, v_i) \notin E$. If there is a directed path between v_k and v_j, then v_k must nest in v_j because the nesting relation is transitive and thus $(v_j, v_k) \notin E$. Hence the directed graph is acyclic.

Claim 0.2. The algorithm we used to find the longest path can guarantee we find the longest correct sequence.

Assume we have three boxes X, Y and Z. X nests in Y and Y nests in Z. Then the corresponding vertices v_1, v_2 and v_3 satisfy that $(v_1, v_2) \in E$ and $(v_1, v_3) \in E$ and $(v_2, v_3) \in E$. The longest path between v_1 and v_3 is v_1->v_2->v_3. Since we sort v_i in an ascending order, then v_3 will be optimized after v_2 is optimized. We could make sure that the current vertex v_k will get the longest path between *s* and v_k since all its preceding vertices have obtained the longest path before updating its distance.

Complexity analysis:

We have proved that sorting a d-dimensional box needs *O(dlogd)* time. We have to do sorting for all *n* boxes which needs *O(ndlogd)* time.

When building graph, we have to determine if the current box nests in the following boxes, each comparison takes *O(d)* time. The total determination time need $O(n^2d)$ time because there are *n* boxes that need to compare with the following at most *n-1* boxes.

When finding the longest path, we do computation and updating for each vertex with its following at most *n-1* vertices which takes $O(n^2)$ time.

Hence the total time complexity is $O(ndlogd+n^2d)$.

Problem2:

HLD: We use Greedy algorithm to pick k' distinct projects.

We first sort the *n* projects in decreasing order according to the profit *p*. Define an array *visited[n]* to denote if project i_j has been done. The array is initialized as all false.

For the base case, we have initial capital C_0, then we search the sorted projects from the one with the largest profit to the one with the least profit. If $c_j < C_0$, then project i_j will be selected because we could gain the maximum profit from the projects that we could execute. When selecting i_j, we set *visited[j]* to be true to show i_j has been completed.

After completing project i_j, we have accumulated capital $C_j = C_0 + \sum_{h=1}^{h=j} p_{ih}$. To find the next project, we still search the sorted projects. In this process, we check if project i_k has been completed (visited[k]==true) and if c_k<C_j. If the visited[k]=false and c_k<C_j, then next project is found. If for all uncompleted projects, we have c_k>C_j, it means that we could not do any projects later since our capital is not sufficient to start the uncompleted projects. If we have completed k distinct projects or we could not find any projects whose c_j is lower than C_j, we return our business plan.

Proof of Correctness:

Claim 0.1. After completing j projects, the accumulated capital we gained is the largest.

Assume that we have capital C_{j-1} after we finished $j-1$ projects and then we have some projects that are not been done and they are also sorted by their profit. If there are two projects i_j and i_k with profit $p_j > p_k$, we tend to choose i_j since we could gain more profit. But if c_j<C_{j-1}, then i_j could not be executed and i_j could not be executed in between the $j-1$ projects because $C_r < C_{j-1} < c_j$ for $1 \leq r < j-1$. Hence through searching the sorted projects, we will always find the uncompleted and "can-do" projects with the maximum profit to guarantee the current accumulated capital is the largest.

Claim 0.2. The picking process could stop at loop j, $1 \leq j \leq k$.

We already proved that after completing j projects, the accumulated capital we gained is the largest. When we pick the $(j+1)$th project, if we could not find a project whose required capital is less than <C_j, the loop will stop. Or if we could compete *k* projects, the loop will stop since the question's requirement is to pick up to *k* distinct projects.

Complexity Analysis:

Sorting the projects according to their profit needs $O(nlogn)$ time. Then it takes $O(n)$ time to pick the next project. We need to pick *k* times at most. Hence the picking time is $O(nk)$. The total time complexity is $O(nk + nlogn)$

Problem3:

HLD: We solve this problem in Greedy algorithm. First, we sort the degree adaptability p_i in decreasing order and obtain P=$\{p_n, p_{n-1}, \dots, p_1\}(p_j \geq p_{j-1})$ and q_i in acceding order Q=$\{q_1, q_2, \dots, q_n\}(q_{j-1} \leq q_j)$.

For the base case, we will find a coworker for worker *n* with degree adaptability p_n. Firstly, we compute y=p_n/x. Then we search Q from left to right and find the minimum element q_i with $q_i \geq y$. Define a variable $minIndex$ to record the index of q_i.

Then for element $p_i(n > i \geq 1)$ in P, we compute y=p_i/x. We search the coworker of $i-th$ worker by traversing Q from $minIndex$ $(not\ included)$ to right and find the minimum element q_j with $q_j \geq y$. Then worker *i* from Ur and worker *j* from Nena will form a pair. Update $minIndex$ with the index of q_j.

We do such process to elements in P from left to right until we could not find an element in Q that satisfies $q_j \geq y$ or we have found coworkers for all workers in Ur, that is, for all $p_i\ in\ P$, there exist a distinct q_j that satisfies $p_i q_j \geq x$. Then we output the partner pairs we have got.

Proof of Correctness:

For the base case, we compute y=p_n/x and find the minimum element q_j with $q_j \geq y$. Hence for any element $q_k(1 \leq k < j)$, $p_n q_k < x$. There should not be any element in P that can pair with q_k since p_n is the largest in P.

Assume that p_j pairs with q_r, since $p_j \geq p_{j-1}$ in P, the element that could pair with p_{j-1} should be on the right side of q_r in Q (if existed). The reason is that q_r is the leftmost unmatched element that satisfies $p_j q_r \geq x$. For any unmatched element $q_{unmatched}$ in Q whose index is lower than q_r, $q_{unmatched}p_{j-1} < q_{unmatched}p_j < x$. Thus we only need to check the element from q_r $(not\ included)$ which is the reason we define the variable $minIndex$.

We could find the leftmost unmatched element q_r that satisfies $p_j q_r \geq x$. Assume the output match of our algorithm for p_j and p_{j-1} is

$$p_j -> q_r \text{ and } p_{j-1} -> q_{r+1}.$$

We could also match p_j with $q_{r+1}, \dots, q_n$. Let's assume p_j pair with q_{r+1}. If $p_{j-1}q_r < x$, then p_{j-1} must match with element on the right side of q_{r+1}. q_r will be left with no one to match. Any element that could match q_r can match $q_k(r < k \leq$

n) since Q is in acceding order. The number of formed pairs within (q_r, q_n) should be no less than that within (q_{r+1}, q_n). Therefore, we should pair $p_j \text{−>} q_r$ to obtain more possible team.

Complexity Analysis:

For each element in P, we do division to find $y = x/p_i$. Then we do for-loop to find an unmatched q_j that satisfies $p_i q_r > x$. Since there are two loops, the overall time complexity is $O(n^2)$.

Problem 4:

1. **HLD: Find the edges that are contained in every graph.** In graph G_0, we use *Adjacency List* to store the adjacent nodes of every node v_i. For each graph, we define an adjacency list and get all b adjacency lists. For graph $G_0(V, E_0)$ and $G_1(V, E_1)$, if $(v_i, v_j) \in E_0$ and $(v_i, v_j) \notin E_1$, we delete the link between v_i and v_j in Graph G_0 and vise versa. By doing this, we delete the edges which do not exist in both graphs. After checking G_0 and G_1, we use the intersection of E_0 and E_1 to do intersection with E_2 and so on. After checking all b graphs, we get the edges that exist in all the graphs, let's assume the intersectant edges as E'.

   Do $\boldsymbol{BFS}$ to find the shortest path. We begin at s to do breadth-first search. BFS could ensure that when we encounter t, the path between s and t is the shortest.

   Proof of Correctness:

   Claim 0.1. The shortest path between s and t only contains edges that exist in all of the graphs.

   If there is an edge (v_i, v_j) that only exists in some of the graphs but not exist in, let's say, Graph G_k, then the expected $s - t$ path cannot have the edge (v_i, v_j) since graph G_k do not have a $s - t$ path that included (v_i, v_j). That's why we do intersection to find the edges that exist in all the graphs. We could use these intersectant edges to find a $s - t$ path whose edges must be included in all graphs.

   Claim 0.2. BFS will find the shortest $s - t$ path from the intersectant edges.

   We start from node s. BFS will firstly find all the nodes that are one edge away from s. After searching all the one-edge-away nodes, BFS will find all the unvisited two-edge away nodes. BFS will always traverse all the nodes that are k edges away before searching for the $k+1$ edge-away nodes. So when the searching comes to t, it is assured that the length between s and t is the shortest.

Complexity Analysis:

Assume there are V nodes and at most E edges in each graph, by comparing the edges in all b graphs, we need $O(b(|V|+|E|))$ time. BFS will take $O(|V|+|E|)$ time since every node will be visited only once. Hence the total time complexity is $O(b(|V|+|E|))$.

2. **HLD:** We define the length of the shortest $s-t$ path we found in part 1 is L. We use dynamic programming algorithm to solve this question. We define an array dp[b+1] where dp[i] is the minimum $cost(P_0, P_1, \dots, P_i)$ from G_0 to G_i. We initialized dp[i]=L*(i+1) because it begins at P_0.

dp[0] is the shortest length of $s-t$ path computed using BFS. We could find the shortest $s-t$ path in continuous graphs G_i to G_j by the same method mentioned in part 1. We denote the length of this $s-t$ path as $L(i,j)$.

dp[j]=min(dp[j], dp[i-1]+K-(j-i+1)* $L(i,j)$) for all $1 \le i \le j$.

And then we find the minimum cost of a sequence of paths $P_0, P_1, \dots, P_i$. We return dp[b] as the result.

Proof of Correctness:

If the $s-t$ path we find in each graph is the one we got in part 1, then the cost is L*(b+1) since there in no changes. We initialized *dp[i]* as L*(i+1) since the minimum $cost(P_0, P_1, \dots, P_i)$ will no more than L*(i+1).

For the base case, if there is only one graph G_0, then the lowest cost is the minimum length of $s-t$ path in G_0, that is L, so *dp[0]=L.*

Assume we already find the minimum cost for all *dp[i]* $(1 \le i < j)$ when we try to find the minimum cost *dp[j]* from G_0 to G_j. If there is no change among graph G_i and G_j and there is a change between G_i and G_{i-1}, we find the shortest $s-t$ path with length $L(i,j)$ between graph G_i and G_j. We should note that $L(i,j)$ is smaller or at least equals to L since the intersectant edges among G_i and G_j are no less than that among all graphs and through these edges, we may find a shorter path between s and t. So for the case that no changes among graph G_i and

G_j, we get the cost is *dp[i-1]+K-(j-i+1)** $L(i,j)$ where K denotes that there is a change between G_i and G_{i-1}. We may figure out all the case for $1 \leq i \leq j$ and get the minimum $cost(P_0, P_1, \ldots, P_j)$: *dp[j]*.

Complexity Analysis: For each *dp[j],* we take $O(d(|V| + |E|))$ to find the smallest path among graph G_i and G_j and compare j-1 times to update *dp[j]*. So the total time for each *dp[j]* is $O(b^2(|V| + |E|))$. The entire time complexity is $O(b^3(|V| + |E|))$.