CSE 202 Homework #2

[redacted]

1. **Problem 1: Nesting Boxes**

   (1) We prove the transitivity by claiming that:

   Claim 1. *Denote three d-dimensional boxes as $(x_1, x_2, \cdots, x_d)$, $(y_1, y_2, \cdots, y_d)$ and $(z_1, z_2, \cdots, z_d)$, if x can nest in y and y can nest in z, x can always nest in z.*

   Proof. Since x can nest in y, we have:

   $$x_{\pi_x(1)} < y_1, \cdots, x_{\pi_x(d)} < y_d$$

   where π_x is a permutation on $\{1, \cdots, d\}$. Since y can nest in z, we have:

   $$y_{\pi_y(1)} < z_1, \cdots, y_{\pi_y(d)} < z_d$$

   where π_y is a permutation on $\{1, \cdots, d\}$.
   As $\pi_x \circ \pi_y(i) = \pi_x(\pi_y(i))$ is also a permutation, and we have:

   $$x_{\pi_x(\pi_y(1))} < y_{\pi_y(1)} < z_1, \cdots, x_{\pi_x(\pi_y(d))} < y_{\pi_y(d)} < z_d$$

   Thus x can also nest in z, which shows that the nesting relation is transitive. $\square$

   (2) Denote two d-dimensional boxes as $(x_1, x_2, \cdots, x_d)$ and $(y_1, y_2, \cdots, y_d)$, we can design an algorithm to determine whether x can nest in y as follows:

   - Sort the dimensions of each box in ascending order, i.e., both $x_1 \leq x_2 \leq \cdots \leq x_d$ and $y_1 \leq y_2 \leq \cdots \leq y_d$ satisfy.
   - x can nest in y if and only if $\forall i \in \{1, \cdots, d\}$, $x_i < y_i$ holds.

   The correctness of this algorithm can be proved by two steps. With no loss of generality, we can always assume $y_1 \leq y_2 \leq \cdots \leq y_d$.

   - If x cannot nest in y, we can simply prove by contradiction, i.e., if we sort the dimension of x in ascending order and $\forall i \in \{1, \cdots, d\}$, $x_i < y_i$ holds, denote the permutation which matches the indices before and after sorting in x as π, then π^{-1} (i.e. the inverse of permutation π) is a valid permutation where $\forall i \in \{1, \cdots, d\}, x_{\pi^{-1}(i)} < y_i$ holds, which means x can nest in y. Thus we find a contradiction.

- If x can nest in y, denote a valid permutation π such that $\forall i \in \{1, \cdots, d\}, x_{\pi(i)} < y_i$ holds.
  - If the sequence $x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(d)}$ is already in ascending order, it just meet what is stated in our algorithm.
  - If the sequence $x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(d)}$ is not in ascending order, i.e. $\exists i$ such that $x_{\pi(i)} > x_{\pi(i+1)}$ holds. In this case, we have:

    $$\begin{cases} x_{\pi(i)} < y_i \\ x_{\pi(i+1)} < y_{i+1} \end{cases}$$

    If we swap $\pi(i)$ and $\pi(i+1)$, we can see that it is still a valid permutation:

    $$\begin{cases} x_{\pi(i+1)} < x_{\pi(i)} < y_i & \Rightarrow \quad x_{\pi(i+1)} < y_i \\ x_{\pi(i)} < y_i \leq y_{i+1} & \Rightarrow \quad x_{\pi(i)} < y_{i+1} \end{cases}$$

    By swapping $\pi(i)$ and $\pi(i+1)$, we preserve the validity. Note that the number of reverse pairs in $x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(d)}$ decreases by 1 after the swap, thus by finite swapping steps, we can always finish by obtaining a valid permutation where $x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(d)}$ is in ascending order (i.e. the number of reverse pairs is 0).

The time complexity of this algorithm is $O(d \log d)$, since the sorting takes $O(d \log d)$ time and the checking process for all (x_i, y_i) pairs takes $O(d)$ time. Thus it is efficient.

(3) Assume $n \geq 1$. Denote the dimensions of B_i as $B_{i,1}, \cdots, B_{i,d}$, we can design an algorithm to find the longest subsequence:

- We first sort the dimensions of every B_i in ascending order, such that $B_{i,1} \leq B_{i,2} \leq \cdots \leq B_{i,d}$ holds.
- We then sort all the B_i with regard to any order, as long as the order satisfies that if $\forall x \in \{1, \cdots, d\}$, $B_{i,x} < B_{j,x}$ holds, B_i must appear before B_j in the sorted result. A possible order is just the "lexicographical" order on lists of the same length. We use $B_i << B_j$ to indicate that B_i can nest in B_j.
- Denote $f[i]$ as the length of longest sequence that ends with B_i, $g[i]$ as the index of the previous box in the longest sequence that ends with B_i (if $f[i] = 1$, $g[i]$ will be $-\infty$). We can use dynamic programming to find $g[i]$:

  $$g[i] = \begin{cases} \arg \max\limits_{1 \leq j < i, B_j << B_i} (f[j]), & \exists B_j \text{ such that } B_j << B_i \\ -\infty, & \text{otherwise} \end{cases}$$

  and $f[i]$:

  $$f[i] = \begin{cases} f[g[i]] + 1, & g[i] \neq -\infty \\ 1, & g[i] = -\infty \end{cases}$$

  by iterating i through $\{1, 2, \cdots, n\}$.

- After the dynamic programming, let $p = \arg\max_{1 \le i \le n}(f[i])$ and we start a loop. We put p into the answer sequence and update p as $g[p]$. We exit the loop when p becomes $-\infty$. By reversing the answer sequence, we can obtain the longest sequence of boxes.

The correctness of the dynamic the programming can be proved directly. For each B_i, in order to find the longest sequence that ends with B_i, we can enumerate all the B_j that can nest in B_i as the second-to-last box in the sequence. By ensuring that "if $B_j << B_i$, B_j must appear before B_i" after sorting, all the B_j that can nest in B_i will appear before B_i. Thus it is sufficient to iterate j through $\{1, \cdots, i-1\}$ to find all the valid B_j that can be the second-to-last box in the sequence. As we would like to find the longest sequence, we pick the B_j that has the maximum value of $f[j]$. If no valid B_j is found, B_i will form a sequence of length 1 itself.

After the dynamic programming, we find the longest sequence that ends with B_p. By checking $g[p], g[g[p]], \cdots$, we can obtain the second-to-last, third-to-last box, $\cdots$ in the longest sequence. Thus after the loop ends when $p = -\infty$, we obtain the longest sequence in reverse order.

The time complexity is made up of four parts:

- The first sorting takes $O(nd \log d)$ time.
- The second sorting takes $O(nd \log n)$ time.
- The dynamic programming takes $O(n^2 d)$ time. The iteration of i and j both take $O(n)$ time. As the dimensions in every B_i and B_j is already sorted, the nest checking only takes $O(d)$ time instead of $O(d \log d)$.
- The answer sequence construction takes $O(n)$ time.

Thus the total time complexity is $O(nd \log d + n^2 d)$, it is efficient.

2. **Problem 2: Classes and rooms**

Input and output clarification

The input is an array C of size m and an array R of size n, where $m, n \geq 1$. Denote the elements as $C[1], \cdots, C[m]$ and $R[1], \cdots, R[n]$. All the elements are positive integers.

The output is a list of pairs. Each pair (c_i, r_i) in the list indicates that a class of size c_i is assigned to a classroom of size r_i. The sum of all the r_i entries is as small as possible. If there are no valid assignments, the list will be empty.

High-level description

We first sort C and R by ascending order such that:

$$C[1] \leq C[2] \leq \cdots \leq C[m]$$

and

$$R[1] \leq R[2] \leq \cdots \leq R[n]$$

satisfy. We then use two pointers $i = j = 1$ to assign the classes with classrooms, where i and j point to a class and a classroom, respectively. The assignment process is executed in a loop:

- If $i = m + 1$, it means all the classes are assigned with classrooms. We return the answer list.
- If $j = n + 1$, it means all the classrooms are used or ignored, but there are still classes that are not assigned with classrooms. We return an empty list.
- Otherwise, we try to assign class $C[i]$ to classroom $R[j]$. If $C[i] \leq R[j]$, the assignment is valid, we put $(C[i], R[j])$ into the answer list and increase both i and j by 1. If $C[i] > R[j]$, the capacity of the classroom is not sufficient, we ignore it and increase j by 1.

Proof of correctness

First we show that our algorithm will always terminate. If $i \neq m + 1$ and $j \neq n + 1$ when inside the loop, regardless of the relationship between $C[i]$ and $R[j]$, we always increase j by 1. As the initial value of j is 1, thus after n loop steps, we will reach $j = n + 1$ and return an empty list. If we reach $i = m + 1$ before $j = n + 1$, we will return the answer list.

Then we show that our algorithm will generate an assignment plan that minimizes the total capacity of rooms used. We can prove by giving any valid assignment plan, "transforming" it to the plan generated by our algorithm, during which the total capacity never increases.

With no loss of generality, we assume $C[1] \leq C[2] \leq \cdots \leq C[m]$ and $R[1] \leq R[2] \leq \cdots \leq R[n]$. Denote the optimal assignment plan generated by our algorithm as:

$$opt_1, opt_2, \cdots, opt_m$$

where opt_i means we assign class i with classroom $opt[i]$. $C[i] \leq R[opt_i]$ must holds. Similarly, we pick any assignment plan (other than our optimal plan) denoted as:

$$cand_1, cand_2, \cdots, cand_m$$

Denote k as the index of the first different entry between opt and $cand$, i.e., $opt_i = cand_i$ for any $1 \leq i < k$ and $opt_k \neq cand_k$. We consider three cases:

- If $opt_k < cand_k$ and the classroom opt_k is not used in $cand$, we can replace $cand_k$ by opt_k and decrease the total capacity by $R[cand_k] - R[opt_k]$.
- If $opt_k < cand_k$ and the classroom opt_k is used in $cand$ (say $cand_x$), x must be strictly larger than k (because for all indices that $< k$, opt and $cand$ entries are just the same). We can observe that:

  $$\begin{cases} C[k] \leq R[opt_k] = R[cand_x] & \Rightarrow \quad C[k] \leq R[cand_x] \\ C[x] \leq R[cand_x] = R[opt_k] \leq R[cand_k] & \Rightarrow \quad C[x] \leq R[cand_k] \end{cases}$$

  Thus we can swap $cand_k$ and $cand_x$ and the total capacity keeps the same.
- If $opt_k > cand_k$, we will show that this case is impossible. According to the definition of k, $cand_k$ should not appear in $opt_1, \cdots, opt_k$. Denote opt_x as the first entry that $opt_x > cand_k$ holds, then we will meet $j = cand_k$ before $j = opt_x$ in the loop when $i = x$. As $C[x] \leq C[k] \leq R[cand_k]$ holds, our algorithm will never assign class x to classroom opt_x based on the fact that $cand_k$ is also valid and smaller than opt_x. Thus the case $opt_k > cand_k$ never appears.

By contradiction, if there exists an assignment plan which has strictly less total capacity than our optimal plan, we can always transform from that plan to our optimal plan step by step as above, during which the total capacity will not increase. Thus we will have one optimal plan with two different capacities, which is impossible.

In addition, if there is no valid assignment plan, our algorithm will definitely return an empty list, since we can never find m pairs.

Time complexity

The time complexity is made up of two parts:

- The sorting takes $O(m \log m + n \log n)$ time.
- The loop takes $O(n)$ time, since we increase j by 1 in every loop step.

Thus the total time complexity is $O(m \log m + n \log n)$, it is efficient.

3. **Problem 3: Business Plan**

Input and output clarification

The input are two integers C_0, k and two arrays c, p of size n, where $n \geq 1$. Denote the elements as $c[1], \cdots, c[n]$ and $p[1], \cdots, p[n]$. All the elements are positive integers.

The output is a list of pairs. Each pair (c'_i, p'_i) in the list indicates a project. The company can start the projects one by one in order in the list to obtain the maximum amount of capital. The length of the list is no larger than k.

High-level description

We first sort the projects by ascending order of c such that:

$$c_1 \leq c_2 \leq \cdots \leq c_n$$

The array p is adjusted by the new order of c, such that every (c_i, p_i) is a project in the original arrays.

We then use a loop which executes no more than k times to find the maximum final amount of capital. Denote a pointer *ptr* initialized as 1 and a maximum heap *pq* initialized as empty. During each step of the loop:

- We first put all the projects that can be started into the heap. In order to do this, we keep pushing pairs (c_{ptr}, p_{ptr}) into the heap and increasing *ptr* by 1, until $c_{ptr} \leq C_0$ does not hold or *ptr* is out of range (i.e. $ptr > n$).
- We then pop out the maximum value (c_{best}, p_{best}) from the heap (the comparison in the heap is just the p entry) and update C_0 as $C_0 \leftarrow C_0 + p_{best}$. We also put (c_{best}, p_{best}) into the answer list. If the heap is empty that no value can be popped out, we exit the loop immediately since no project that can be started.

After the loop, we return the answer list.

Proof of correctness

First we show that out algorithm will always generate a valid project chosen plan. As we our loop executes no more than k times, we will always return a list of length no more than k. During each step of the loop, all the projects in the heap have c value no more than C_0 and we pop out a project if it is started, thus we will always start valid projects and never start a same project more than once.

Then we show that our algorithm will generate a project chosen plan that maximize the final amount of capital. We can prove by giving any valid project chosen plan, "transforming" it to the plan generated by our algorithm, during which the amount of capital never decreases.

With no loss of generality, we assume $c_1 \leq c_2 \leq \cdots \leq c_n$ and every (c_i, p_i) is a project. Denote the indices of projects chosen in the optimal plan generated by our algorithm as:

$$opt_1, opt_2, \cdots, opt_{k_0}$$

where $k_0 \leq k$ is the number of projects chosen in the optimal plan. Similarly, we pick any arbitrary project chosen plan (other than our optimal plan) denoted as:

$$cand_1, cand_2, \cdots, cand_{k_1}$$

where $k_1 \leq k$ is the number of projects chosen in the picked plan. k_1 may not be equal to k_0.

Denote u as the index of the first different entry between *opt* and *cand*, i.e., $opt_u = cand_u$ for any $1 \leq i < u$ and $opt_u \neq cand_u$. We consider five cases:

- If $p_{opt_u} \geq p_{cand_u}$ and the project opt_u is not used in *cand*, we can replace $cand_u$ by opt_u and increase the total amount of capital by $p_{opt_u} - p_{cand_u}$.
- If $p_{opt_u} \geq p_{cand_u}$ and the project opt_u is used in *cand* (say $cand_x$), x must be strictly larger than u (because for all indices that $< u$, *opt* and *cand* entries are just the same). We can focus on whether we could swap $cand_u$ and $cand_x$. As the only restriction of starting a project is the total amount of capital, $\forall i < u$ and $\forall i > x$, the total amount of capital before starting (c_{cand_i}, p_{cand_i}) does not change after swap. For the other indices, if we swap $cand_u$ and $cand_x$:
  - The new $cand_u$ is the old $cand_x$. Since all the first $u - 1$ entries of *opt* and *cand* are the same, when $cand_x$ is swapped to $cand_u$:

    $$c_{cand_x} = c_{opt_u} \leq C_0 + \sum_{i=1}^{u-1} p_{cand_i}$$

    satisfies. Thus the old $cand_x$ can start after swapping.
  - $\forall u < i < x$, before swapping it satisfies:

    $$c_{cand_i} \leq C_0 + \sum_{j=1}^{i-1} p_{cand_i}$$

    After swapping, the p_{cand_u} entry on the right of $\leq$ is replaced with p_{cand_x}. Since $p_{cand_x} = p_{opt_u} \geq p_{cand_u}$, the $\leq$ relationship will be kept.
  - The new $cand_x$ is the old $cand_u$. There is always no problem to "delay" a project since the total amount of capital always increase. Thus the old $cand_u$ can start after swapping.

  It shows that the plan is still valid after swapping $cand_u$ and $cand_x$, and the final amount of capital keeps the same.

- If $p_{opt_u} < p_{cand_u}$, we will show that this case is impossible. The pointer *ptr* ensures that all the projects that have c value no more than $C_0 + \sum_{i=1}^{u-1} p_{opt_u} = C_0 + \sum_{i=1}^{u-1} p_{cand_u}$ are pushed into the heap, and the maximum heap ensures that the project that has the maximum p value is selected as opt_u. Thus no valid projects can have p value strictly larger than p_{opt_u}.
- If u is undefined and $k_0 > k_1$ (i.e. all the first k_1 entries of *opt* and *cand* are the same), we can simply add opt_{k_0+1} to the end of *cand* and increase the total amount of capital by $p_{cand_{k_0+1}}$.
- If u is undefined and $k_0 < k_1$ (i.e. all the first k_0 entries of *opt* and *cand* are the same), we will show that this case is also impossible. As $cand_{k_0+1}$ is never used in *opt* and can be started after the first k_0 *cand* entries (the same as the first k_0 *opt* entries), it must still inside the heap when we are taking the $(k_0 + 1)$-th step of the loop, thus our algorithm will pop out a project (maybe $cand_{k_0+1}$ or not) and *opt* should have at least $k_0 + 1$ entries.

By contradiction, if there exists a projects chosen plan which has strictly more final amount of capital than our optimal plan, we can always transform from that plan to our optimal plan step by step as above, during which the final amount of capital will not decrease. Thus we will have one optimal plan with two different final amount of capital, which is impossible.

Time complexity

The time complexity is made up of three parts:

- The sorting takes $O(n \log n)$ time.
- The loop takes $O(k)$ time.
- The operations of the maximum heap takes $O(n \log n)$ time. In the worst case, all the projects are pushed into the heap (takes $O(n \log n)$ time) and k projects are popped out from the heap (takes $O(k \log n)$ time). $k \leq n$ indicates that the total time complexity of the heap is $O(n \log n)$.

Thus the total time complexity is $O(n \log n)$.

4. **Problem 4: Shortest wireless path sequence**

(1) We first use a hash set S to store all the edges in E_0.

We then iterate i through $1, 2, \cdots, b$. During each iteration, we use an empty hash set S' to store all the edges in both E_i and S (we can do this by iterating through all the edges in E_i and check whether the edge is in S). After S' is obtained, we update S by S'.

After the whole iteration, we create a graph $G = (V, S)$ and try to find the shortest path from s to t in G. We can use a breadth-first search with a FIFO queue. Specifically, we first put s into the queue. During each step of the breadth-first search, we pop out a node u from the queue and iterate through all its neighbors in the graph. For each neighbor v, if it has not been pushed (we can use a hash set to store all the nodes that have been pushed), we push v into the queue and mark the previous node of v as u. After the breadth-first search ends, we put t, the previous node of t, the previous node of the previous node of t, $\cdots$ into the answer path until reaching s. The reverse of the answer path is just the shortest s-t path.

The correctness of this algorithm can be proved straightforward. After the iteration, S is the intersection of $E_0, E_1, \cdots, E_b$, which indicates the edges we can choose for the path that exists in all $G_0, G_1, \cdots, G_b$. The breadth-first search ensures that we can find the shortest path from the starting node s to any nodes in the graph, and the record of previous node can help recover the specific path from t back to s.

The time complexity of this algorithm is $O(|V| + \sum_{i=0}^{b} |E_i|)$. During the whole iteration, we check every edge in all the E_i once and take $O(\sum_{i=0}^{b} |E_i|)$ time. The graph $G = (V, S)$ will have number of edges no more than $\min_{i=0}^{b} |E_i|$, so the breadth-first search takes $O(|V| + \min_{i=0}^{b} |E_i|)$ time and the s-t path generation takes $O(|V|)$ time. Thus the total time complexity is $O(|V| + \sum_{i=0}^{b} |E_i|)$.

(2) We first encapsulate the algorithm described in (1). Denote $\text{getLength}(l, r)$ as the length of the shortest s-t path in the graphs $G_l, G_{l+1}, \cdots, G_r$. If no valid path exists, the result will be -1.

Then we can use dynamic programming to solve this problem. Denote f_i as the minimum cost of $cost(P_0, P_1, \cdots, P_i)$, the initial value of f_i is ∞. We consider two types of transition equations.

- We would like all the $i + 1$ graphs to have the same path. We call $l = \text{getLength}(0, i)$, if $l \neq -1$, we update f_i by:

$$f_i \leftarrow \min\left\{f_i, l \times (i + 1)\right\}$$

- We would like the graphs $G_{j+1}, \cdots, G_i$ to have the same path, where $0 \leq j < i$. For every valid j, we call $l = \text{getLength}(j + 1, i)$, if $l \neq -1$, we update f_i by:

$$f_i \leftarrow \min\left\{f_i, f_j + l \times (i - j) + K\right\}$$

Besides f_i, we also use g_i to store where the minimum value of f_i comes from. If f_i comes from the first type, we have $g_i = -1$. If f_i comes from the second type, we have $g_i = j$.

After the dynamic programming finishes, we can generate the answer paths which result in the minimum cost. We also encapsulate the algorithm described in (1) as getPath(l, r), which returns the detailed shortest s-t path in the graphs $G_l, G_{l+1}, \cdots, G_r$.

- We initialize $r = b$.
- We call getPath$(g_r + 1, r)$, which is the answer path of $P_{g_r+1}, \cdots, P_r$. We then update r by $r \leftarrow g_r$. If $r \neq -1$, we keep this path generation process.

After the generation above, we will obtain $P_0, P_1, \cdots, P_b$ with minimum cost f_b.

The correctness of this algorithm can be proved by two steps. We first prove the correctness of the dynamic programming, then prove the correctness of the answer path generation.

Claim 2. *f_i stores the minimum cost of cost$(P_0, P_1, \cdots, P_i)$.*

Proof. Denote j ($j \in [-1, i)$) as the smallest index that $G_{j+1}, \cdots, G_i$ have the exactly same path to obtain the optimal *cost*$(P_0, P_1, \cdots, P_i)$. The shortest length of this path can be obtained by $l =$ getLength$(j+1, i)$, and it contributes $l \times (i-j)$ to the cost. We consider two cases:

- If $j = -1$, the total cost is just $l \times (i - j) = l \times (i + 1)$.
- If $j > -1$, the total cost should also include the lengths of paths in $G_0, G_1, \cdots, G_j$ and an extra entry of K because we change the path from G_j to G_j+1. As the optimal cost of *cost*$(P_0, P_1, \cdots, P_j)$ is just f_j, the optimal total cost should be $f_j + l \times (i - j) + K$.

The two cases above is just the same as the transition equation in our dynamic programming. Thus we can obtain the minimum cost of *cost*$(P_0, P_1, \cdots, P_i)$ in f_i. □

Claim 3. *The last generation step in our algorithm can produce the correct $P_0, P_1, \cdots, P_b$.*

Proof. For any r during the generation step, the minimum cost requires $G_{g_r+1}, \cdots, G_r$ to have the same path. We can call getPath$(g_r + 1, r)$ to generate this path and assign it to $P_{g_r+1}, \cdots, P_r$. We then generate the rest $P_0, \cdots, P_{g_r}$ "recursively" by updating r as $r \leftarrow g_r$ if $g_r \neq -1$. Since $g_r < r$ always holds, the generation step will terminate when $g_r = -1$ and we finally get $P_0, P_1, \cdots, P_b$. □

By calling getLength and getPath with no optimization, the time complexity is made up of two parts:

- The dynamic programming takes $O(b^3|V|^2)$ time. Both i and j takes $O(b)$ time to iterate, and the call of getPath takes $O(b|V|^2)$ time (every $|E|$ is bounded by $|V|^2$).
- The generation step takes $O(b|V|^2)$ time. All the intervals (l, r) used to call getPath do not intersect and their union is just $[0, b]$, thus the time complexity equals calling getPath$(0, b)$ which is $O(|V| + \sum_{i=0}^{b} |E_i|) \in O(b|V|^2)$.

Thus the total time complexity is $O(b^3|V|^2)$. But we can optimize the order of the getLength calls for any fixed i. By iterating j in decreasing order, we can re-use the union of the edge sets, because every time we decrease j by 1, we can simply use E_{j+1} to update the hash set instead of re-calculating the union of $E_{j+1}, \cdots, E_i$. In this way, we can decrease the time complexity of dynamic programming as well as the whole algorithm from $O(b^3|V|^2)$ to $O(b^2|V|^2)$.

5. **Problem 5: Untangling signal superposition**

   Input and output clarification

   The input are three 01 strings s, x and y. We assume all the three strings are not empty, but we claim that an empty string can be a repetition of any strings.

   The output is a boolean value indicating whether s is an interleaving of x and y. By our claim, if the output is true, we can partition s into two subsequences s' and s'', where s' is a repetition of x and s'' is a repetition of y, as well as either s' or s'' can be empty (they cannot be both empty since s is not empty).

   High-level description

   We can use dynamic programming to solve this problem. Denote $f(i, p_1, p_2)$ as a boolean value, indicating whether we can partition the first i characters of s into s' and s'', so that s' is a repetition of x and s'' is a repetition of y, as well as $|s'| \equiv p_1 \pmod{|x|}$ and $|s''| \equiv p_2 \pmod{|y|}$. The domain of i, p_1, p_2 are $[0, |s|], [1, |x|], [1, |y|]$, respectively.

   Define a helper function $\text{prev}(x, n)$ as:

$$\text{prev}(x, n) = \begin{cases} x - 1, & x \neq 1 \\ n, & x = 1 \end{cases}$$

   $f(i, p_1, p_2)$ is True if and only if at least one of two following cases hold:

   - $f(i-1, \text{prev}(p_1, |x|), p_2)$ is True and $s[i] = x[p_1]$.
   - $f(i-1, p_1, \text{prev}(p_2, |y|))$ is True and $s[i] = y[p_2]$.

   At the beginning, all the $f(i, p_1, p_2)$ entries except $f(0, |x|, |y|) = \text{True}$ are initialized as False. We then use a triple nested loop to iterate i, p_1, p_2 through $[1, |s|], [1, |x|], [1, |y|]$ in ascending order to find the value of every $f(i, p_1, p_2)$ entry.

   After the dynamic programming finishes, we return true if there is an entry $f(|s|, p_1, p_2)$ that has a true value. Otherwise we return false.

   Proof of correctness

   Claim 4. *$f(i, p_1, p_2)$ shows whether we can partition the first i characters of s into s' and s'', so that s' is a repetition of x and s'' is a repetition of y, as well as $|s'| \equiv p_1 \pmod{|x|}$ and $|s''| \equiv p_2 \pmod{|y|}$.*

   Proof. We can proof the correctness by induction.

   - **Base case**. When $i = 0$, we can only partition an empty string into two empty strings s' and s'', i.e., $|s'| = 0 \equiv |x| \pmod{|x|}$ and $|s''| = 0 \equiv |y| \pmod{|y|}$. As an empty string is a repetition of any strings, we have $f(0, |x|, |y|) = \text{True}$. All the other $f(0, p_1, p_2)$ entries are false.

- **Induction step**. Assume all the $f(i-1, p_1, p_2)$ entries are correct. In order to calculate $f(i, p_1, p_2)$, we can have two choices:
  - If $s[i]$ is partitioned into s', we must have $s[i] = x[p_1]$. The reason is that s' is a prefix of some x^k and the length of s' is equivalent to p_1 modulo $|x|$. Besides that, the first $i-1$ characters of s should be partitioned into two strings, one of which is s' but removing the last character, and the other is just s''. This can be obtained by $f(i-1, p_1-1, p_2)$. In order to make $p_1 - 1$ in its domain, we replace this entry by $f(i-1, \text{prev}(p_1, x), p_2)$. Thus, in this case, we must have both $s[i] = x[p_1]$ and $f(i-1, \text{prev}(p_1, x), p_2) = \text{True}$.
  - If $s[i]$ is partitioned into s'', we must have $s[i] = y[p_2]$. The reason is that s'' is a prefix of some y^k and the length of s'' is equivalent to p_2 modulo $|y|$. Besides that, the first $i-1$ characters of s should be partitioned into two strings, one of which is just s', and the other is s'' but removing the last character. This can be obtained by $f(i-1, p_1, p_2-1)$. In order to make $p_2 - 1$ in its domain, we replace this entry by $f(i-1, p_1, \text{prev}(p_2, y))$. Thus, in this case, we must have both $s[i] = y[p_2]$ and $f(i-1, p_1, \text{prev}(p_2, y)) = \text{True}$.

  If any of the two cases satisfy, we can make a valid partition and assign $f(i, p_1, p_2) = \text{True}$. Otherwise $f(i, p_1, p_2)$ must be False.

□

Thus, after the dynamic programming finishes, we can check whether $f(|s|, p_1, p_2) = \text{True}$ for any $p_1 \in [1, |x|]$ and $p_2 \in [1, |y|]$. A True entry indicates that s can be partitioned in to two subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y, as well as $|s'| \equiv p_1 \pmod{|x|}$ and $|s''| \equiv p_2 \pmod{|y|}$. If all the entries are false, we cannot find a valid partition then.

Time complexity

The triple nested loop to iterate i, p_1, p_2 takes $O(|s||x||y|)$ time and every $f(i, p_1, p_2)$ entry takes constant time to compute. Thus the total time complexity is $O(|s||x||y|)$.