Algorithms: CSE 202 — Homework 2 Solutions

Problem 1: Nesting Boxes (CLRS)

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

1. Argue that the nesting relation is transitive.
2. Describe an efficient method to determine whether or not one d-dimensional box nests inside another.
3. Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k}\rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$. Express the running time of your algorithm in terms of n and d.

Solution: Nesting Boxes

1. Let A, B and C be d-dimensional boxes with dimensions $(a_, \ldots, a_d)$, $(b_1, \ldots, b_d)$ and $(c_1, \ldots, c_d)$ respectively. We use the same symbol to denote a box as well as its dimensions. To prove that the nesting is transitive, we need to show that if A can be nested in B and B can be nested in C, then A can be nested in C. Let π_A be a permutation that nests A in B, that is, $a_{\pi_A(1)} < b_1, \ldots, a_{\pi_A(d)} < b_d$. Let π_B be a permutation that nests B in C, that is, $b_{\pi_B(1)} < c_1, \ldots, b_{\pi_B(d)} < c_d$. Let $\pi'_A = \pi_A \circ \pi_B$. In other words, π' is the permutation obtained by applying π_B followed by π_A. We argue that π' lets us nest A inside C. To see that $a_{\pi'(i)} < c_i$ for $1 \le i \le d$, we observe $b_{\pi_B(i)} < c_i$ and $a_{\pi_A(\pi_B(i))} < b_{\pi_B(i)}$.

2. **Algorithm description:** Let A be the d-dimensional box which is to be nested within another d-dimensional box B. Sort the dimensions of A and B. If the length of every dimension in the sorted list of A is smaller than the length of the corresponding dimension in the sorted list of B, then A can be nested within B.

Correctness proof:

Claim 0.1. If the sorted permutation of both A and B does not satisfy the nesting requirement for each dimension, no other permutation can satisfy the sorting requirement.

Proof. Let $(a_1, \ldots, a_d)$ be the dimensions of the box A such that $a_1 \le \cdots \le a_d$. Let $(b_1, \ldots, b_d)$ be the dimensions of the box B such that $b_1 \le \cdots \le b_d$. If $a_i > b_i$ for some $1 \le i \le d$, then $a_j > b_k$ for all $i \le j \le d$ and $k \le i$. Hence any permutation that satisfies the nesting requirement must map the $d - i + 1$ elements $a_i, \ldots, a_d$ bijectively to the $d - i$ elements $b_{i+1}, \ldots, b_d$, which is not possible □

Time complexity: Sorting the d dimensions takes $\Theta(d \log d)$. An additional d comparisons are made after sorting. Hence overall time complexity is $\Theta(d \log d)$.

3. **Algorithm description:** We represent the problem as that of finding the longest path in a graph G. Each box is represented by a vertex in G. In addition, G contains the vertices S and T. We use B_i to represent the box and the corresponding vertex. The edges of G are determined as follows.

For every pair of vertices B_i and B_j, join vertex B_i to vertex B_j with a directed edge if box B_i nests in box B_j. There is a directed edge from S to each of the other vertices. Every vertex $u \ne T$ is connected to T by a directed edge from u to T. The graph G is a directed acyclic graph since a box cannot be nested inside itself. The longest path from S to T will correspond to the longest sequence of nesting boxes from $\{B_1, B_2, \ldots, B_n\}$.

We define the following subproblems for each vertex v in G.

- Let $D(v)$ be the length of the longest path that terminates at vertex v.
- Let $P(v)$ be the vertex immediately preceding v in the longest path that terminates at v.

We use the following recursive formulation to compute the functions $D(.)$ and $P(.)$.

For the unique source vertex S, we define $D(S) = 0$ and $P(S) = \textbf{null}$. For all other nodes v, we define $D(v) = \max_{u \text{ is an immediate predecessor of } v \text{ in } G} D(u) + 1$ and $P(v) = u$, the predecessor vertex for which the maximum is achieved in the recursive definition of $D(v)$.

The recursive formulation is well-defined as long as we can compute the functions $D(v)$ and $P(v)$ for each v after the predecessors of v have been computed. Since the vertices can be topologically ordered, we conclude that the recursive formulation is well-defined.

We prove that $D(v)$ is the length of the longest path that terminates at v by induction on the index of v in the topological order. Since S is the unique source vertex, it is the first vertex in the topological order. The length of the longest path that terminates at S is zero. Indeed $D(S) = 0$ which establishes the correctness for the base case.

Let v be any vertex other than the source vertex. Assume that $D(u)$ is indeed the length of the longest path for all vertices u preceding v in the topological order. We will prove that $D(v)$ is the length of the longest path that terminates at v. Consider the longest path L that terminates at v. Since v is not a source vertex, there must be at least one predecessor vertex for v. Let u be the vertex that immediately precedes v in L. By inductional hypothesis, $D(u)$ is the length of the longest path that terminates at u so the length of L is $1 + D(u)$. Since $D(v)$ is the maximum of $1 + D(u')$ over all immediate predecessors u' of v, we conclude that $D(v)$ is indeeed the length of the longest path that terminates at v.

$D(T)$ is the length of the longest path in G. We can trace back the path from T to S to get the longest path in G from S to D.

Algorithm 1 Nesting Boxes

```
Input:
Output:
 1: Construct graph $G = (V, E)$
 2: for each $u$ in $V$ do
 3:     $dist[u] \leftarrow -\infty$
 4:     $prev[u] \leftarrow nil$
 5: end for
 6: $dist[S] \leftarrow 0$
 7: Topologically sort $V$
 8: for each $v$ in $V$ do
 9:     for each $u$ which is a predecessor of $u$ do
10:         if $D[v] < D[u] + 1$ then
11:             $D[v] = D[u] + 1$
12:             $P[v] = u$
13:         end if
14:     end for
15: end for
16: current $= P[T]$
17: while current $\neq S$ do
18:     print current
19:     current $\leftarrow P[$current$]$
20: end while
```

Time Complexity: The time for sorting the dimensions for each of the n nodes is $\Theta(nd \log d)$. The time for constructing the graph is $\Theta(n^2 d)$ since it requires n^2 comparison of d dimensions each. The algorithm to find the longest path takes $\Theta(|V| + |E|)$. Since there are a total of n nodes and $\Theta(n^2)$ edges in G, finding the longest path takes $\Theta(n^2)$. Hence the overall complexity is $\Theta(n^2 d + nd \log d)$.

Problem 2: Classes and rooms

You are given a list of classes C and a list of classrooms R. Each class c has a positive enrollment $E(c)$ and each room r has a positive integer capacity $S(r)$. You want to assign each class a room in a way that minimizes the total sizes (capacities) of rooms used. However, the capacity of the room assigned to a class must be at least the enrollment of the class. You cannot assign two classes to the same room. Design an efficient algorithm for assigning classes to rooms and prove the correctness of your algorithm.

Solution:

An assignment α is a mapping that maps every class to a distinct room. An assignment α is valid if it is a 1-1 mapping and if for every class c, the enrollment of c is less or equal to the capacity of the room $\alpha(c)$. An assignment is optimal if it is valid and the sum of the capacities of the assigned rooms is minimized over all valid assignments.

For our algorithm, we will sort the classes in order of increasing enrollment and the rooms in increasing capacity. Then, for each class from smallest to largest, we assign it the smallest possible room, discarding all rooms with smaller capacity. If we run out of rooms before classes, then we output $\perp$, indicating there is no possible assignment. This takes $O(n \log n + m \log m)$ time, where n is the number of classes and m the number of rooms.

Observe that if the algorithm outputs an assignment, then the assignment is a valid assignment since the algorithm assigns a room to a class only if the room is not assigned to any other class and the capacity of the room is greater or equal to the enrollment of the class.

Now suppose there is a valid assignment to the problem; we shall show our algorithm outputs an optimal assignment. Enumerate the classes $c_1, \ldots, c_n$ in order of increasing enrollment, and the rooms $r_1, \ldots, r_m$ in order of increasing size. We will use an exchange argument to prove that the greedy solution is correct. Let $G = [(c_1, r_{g_1}), \ldots, (c_n, r_{g_n})]$ be our greedy assignment, and let G_i be the partial assignment for the first i classes. We also use G_0 to denote an empty assignment. We say that an optimal assignment $O = [(c_1, r_{o_1}), \ldots, (c_n, r_{o_n})]$ extends G_i if $o_j = g_j$ for $j \le i$. We show by induction that for all $0 \le i \le n$, there exists an optimal assignment that extends G_i. Since there is a valid assignment for the problem, there exists an optimal assignment that extends G_0. This proves the base case.

Say an optimal assignment O extends G_{i-1}, but does not extend G_i; we show that there is some other optimal assignment O' which does extend G_i. We do this by analyzing two cases:

Case 1: O does not use r_{g_i}. In this case, we just replace r_{o_i} with r_{g_i} to get O'. Since O extended G_{i-1}, we have that $S(r_{o_i}) \ge S(r_{g_i})$, since r_{o_i} would not have been used in G_{i-1} and r_{g_i} is a smallest capacity room that fits c_i. So the total capacity of rooms in O' is at most that of O.

Case 2: O does use r_{g_i} for some $c_{i'}$. Since O extended G_{i-1}, we have $i' > i$. We claim that replacing the pairs (c_i, r_{o_i}) and $(c_{i'}, r_{g_i})$ with (c_i, r_{g_i}) and $(c_{i'}, r_{o_i})$ to obtain O' gives us a valid assignment; since we do not change the set of rooms used between O and O', O' remains an optimal assignment. To prove O' is a valid assignment, since r_{o_i} was not chosen by the greedy solution (and since it wasn't amongst the first $i-1$ rooms chosen), we have that $S(r_{o_i}) \ge S(r_{g_i})$. So $S(r_{o_i}) \ge E(c_{i'})$, and so pairing r_{o_i} with $c_{i'}$ is valid.

Having proven that for each i there is an optimal assignment extending G_i, we note that an optimal assignment extending G_n means there is an optimal solution using the same pairs as $G_n = G$, and so G is an optimal assignment.

Problem 3: Business plan

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project i and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \ldots, i_{k'}$ with $k' \leq k$. Your *accumulated capital* after completing the project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project i_{j+1} after completing the first j projects, i.e., $C_j \geq c_{i_{j+1}}$ for each $j = 0, \ldots, k'-1$. You want to maximize the final amount of capital, $C_{k'}$.

Solution: Business plan

Algorithm: We select, start and complete projects in a sequence. The next project is selected after the previous project has been completed. Assume the projects $s_1, s_2, \ldots, s_j$ have already been selected, started and completed. Let C_j denote the available capital after the project s_j has been completed and before the next project is selected. A project l is *eligible* at time j if it has not been selected so far and its capital requirement $c_l \leq C_j$. If $j \leq k-1$, select the next project using the following greedy strategy. Among all eligible projects, select the project with the maximum profit. If there is no such project, stop the selection process.

We maintain a heap of all eligible projects ordered by profit so that the project with the maximum profit is at the top of the heap. After completing a project, if the heap is not empty, we select an eligible project with the maximum profit, remove it from the heap and continue. Observe that C_j is increasing with j as profits are nonnegative. As projects get completed, additional projects may become eligible in which case we insert them into the heap. A project never leaves the heap unless it is selected for execution. The heap can be managed with $O(n \lg n)$ time.

Let S^g be the sequence of projects selected by the greedy strategy. We show that S^g is an optimal sequence of projects. We use induction on the number of projects to prove the correctness. In particular, we show that replacing any other choice with the greedy choice in the sequence will produce at least as a good a solution as the optimal solution.

For a feasible sequence of projects $T = t_1, \ldots, t_l$, let value$(T) = C_0 + p_{t_1} + \cdots + p_{t_l}$ denote the capital after all the projects in the sequence have been completed. For an empty sequence T, we define its value as value$(T) = C_0$.

Claim 0.2. The greedy strategy produces an optimal solution on every instance.

Proof. We obtain a contradiction by assuming that there is an instance on which the greedy strategy fails to produce an optimal solution.

Let n be the smallest integer such that there exists an instance of size n for which the greedy strategy does not produce an optimal solution. We argue that $n \geq 2$. If $n = 1$ there is only one feasible solution. If the initial capital is large enough, we select and execute the project. Otherwise, no project is selected. Hence the greedy solution is optimal if there is only one project.

Let I be an instance of size n such that the greedy algorithm fails to produce an optimal solution on I. We are given $k, C_0, c_1, \ldots, c_n, p_1, \ldots, p_n$. Let $S^g = s_1, s_2, \ldots, s_{k'}$ be the greedy solution for I for some $0 \leq k' \leq k$ where s_i is the i-th project selected by the algorithm. If $k' = 0$, S^g is an empty sequence. Observe that the sequence S^g satisfies the following property: for all $1 \leq j < l$, s_{j+1} is the project with the largest profit among all eligible projects at the completion of the first j projects in S^g.

Let $S^g_1 = s_2, \ldots, s_{k'}$.

Let $T = t_1, \ldots, t_{k''}$ be an optimal sequence of projects. Without loss of generality, assume that the projects in T are ordered so that project t_{j+1} is the project with the maximum profit among all projects which are eligible at the time the first j projects in the sequence T are completed. Let $T_1 = t_2, \ldots, t_{k''}$.

Case I — $s_1 = t_1$: Consider the instance I' of size $n-1$ obtained from I by deleting the project s_1 and setting its initial capital to $C_1 = C_0 + p_{s_1}$ and the number of projects to be performed to $k-1$. Observe that the greedy strategy produces the solution S^g_1 on I' (explain why), which by assumption, is an optimal solution. Also, we have that T_1 is a feasible solution for I' and value(T) = value(T_1) (explain why). We get

$$\begin{aligned}\text{value}(S^g) &= C_0 + p_{s_1} + p_{s_2} + \cdots + p_{s'_k} \\ &= \text{value}(S^g_1) \\ &\geq \text{value}(T_1) \\ &= \text{value}(T)\end{aligned}$$

which shows that S^g is indeed an optimal solution which leads to a contradiction.

Case II — $s_1 \neq t_1$:

Let $T' = s_1, t_2, \ldots, t_{k''}$ if s_1 is not in the sequence T. Otherwise, let T' be the sequence obtained from T by swapping t_1 with s_1. In either case, T' is a feasible solution for I (explain why). Moreover value$(T') \geq$ value(T) since $p_{s_1} \geq p_{t_1}$. This implies that T' is an optimal solution.

Now we are in the situation where the first choices of S^g and the optimal solution T' coincide which leads to a contradiction to the fact that S^g is the smallest instance on which the greedy strategy fails.

□

Complexity; The algorithms runs in time $O(n \lg n)$ since the total time required to manage the heap is $O(n \lg n)$.

Problem 4: Shortest wireless path sequence (KT 6.14)

A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

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 $i = 0, 1, 2, \ldots, b$, 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 $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. 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 $P_0, P_1, \ldots, P_b$ 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*$(P_0, P_1, \ldots, P_b)$ 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 $P_0, P_1, \ldots, P_b$ to be

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

1. Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.
2. Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

Solution: Shortest wireless path sequence (KT 6.14)

1. Let $G_i = (V, E_i), i = 0, \dots, b$, be a sequence of graphs and let $s, t \in V$. If there is a single path P from s to t in each of the G_i, then to find a shortest one we can perform breadth first search (BFS) on $G = (V, \bigcap_{i=0}^{b} E_i)$. Computing G can be done in time $O((|V| + \sum |E_i|)$ and BFS takes time $O(|E|+|V|)$ where $|E| = \bigcap_{i=0}^{b} E_i$. Hence the total runtime is bounded by $O((|V| + \sum |E_i|) = O(b \cdot n^2)$

2. To find a min cost sequence $P_0, \dots, P_b$, notice that in any optimal sequence, if i is the *last breakpoint*, i.e., the last index where the path changes, then P_b (with no further breakpoints) is the shortest path for the graph sequence $G_i, \dots, G_b$; and $P_0, \dots, P_{i-1}$ is the optimal sequence of paths for $G_0, \dots, G_{i-1}$ (with potentially additional breakpoints). We turn this into an algorithm below.

   Let $D_{i,j}$ be the length of a shortest path from s to t in $(V, \bigcap_{k=i}^{j} E_k)$. We can compute $D_{i,j}$ for each i, j as follows.

Algorithm 2 Algorithm

```
for $i \leftarrow 0, \dots, b$ do
    $E' \leftarrow E_i$    // $E'$ will store $\cap_{k=i}^{j} E_k$
    $D_{i,i} \leftarrow$ BFS distance from $s$ to $t$ in $(V, E')$
    for $j \leftarrow i+1, \dots, b$ do
        $E'' \leftarrow \varnothing$
        for each $e \in E_j$ do
            if $e \in E'$, add $e$ to $E''$
        end for
        $E' \leftarrow E''$
        $D_{i,j} \leftarrow$ BFS distance from $s$ to $t$ in $(V, E')$
    end for
end for
```

This takes time $O(b^2 \sum(|E_i| + |V|)) = O(b^3 n^2)$.

Next, for $0 \le i \le b$, let C_i be the cost of an optimal sequence for $G_0, \dots, G_i$. Let B_i be the index position of the latest breakpoint. The cost of the optimal sequence of paths C_j is then either the cost of the of shortest path with no breakpoints $D_{0,j}$ or the cost of the shortest sequence of paths up to index B_j, followed by a path with no more breakpoints.

More formally, for $j \in [0, b]$,

$$C_j = \min_{i \in [0,j]} C_{i-1} + (j - i + 1) D_{i,j} + K.$$

$$B_j = \arg\min_{i \in [0,j]} C_{i-1} + (j - i + 1) D_{i,j} + K.$$

In the base case, $C_0 = D_{0,0}$ and $B_0 = 0$. We define $C_{-1} = -K$ such that $C_j = j * D_{0,j}$ if $B_j = 0$, i.e. in the case that the optimal sequence of paths does not contain a breakpoint.

Correctness follows immediately from our assertion that the optimal sequence of paths either contains no breakpoints, in which case we pick the shortest path in the intersection of the graphs, or there must be a last breakpoint B_j. Then the cost of the shortest sequence of paths consists of the cost of the shortest sequence of paths from indices 0 to B_{j-1}, plus the cost of the shortest path from with no breakpoints from indices B_j to j times the number of indices in that range, plus the cost of one breakpoint.

Finally, the total runtime consists of computing first all values $D_{i,j}$ in time $O(b^3 n^2)$ and then computing all values C_j. Since C_j is computed from $C_0, \dots, C_{j-1}$ in time $O(b)$, the total time of the algorithm is dominated by the time to compute all values $D_{i,j}$, i.e. $O(b^3 n^2)$.

Problem 5: Untangling signal superposition (KT 6.19)

You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: “untangling” a superposition of two known signals. Specifically, they’re picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they’re hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. 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. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters $1, 2, 5, 7, 8, 9$ form 101101–a repetition of x–and the remaining characters $3, 4, 6$ form 000–a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we’re listening to: We want to make sure s “unravels” into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Solution: Untangling signal superposition (KT 6.19)

We first repeat the definitions stated in the problem. 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. Equivalently, x' is repetition of x if $x' = x^k u$ for some $k \geq 0$ where u is a prefix of x.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.)

For each prefix $u \neq x$ of x, each prefix $v \neq y$ of y, and $1 \leq i \leq n$, let

$$T(u,v,i) = \begin{cases} \text{True} & \text{if } s_i \dots s_n \text{ can be partitioned into } s' \text{ and } s'' \\ & \text{such that } us' \text{ is a repetition of } x \text{ and } vs'' \text{ is a repetition of } y. \\ \text{False} & \text{otherwise} \end{cases}$$

This definition creates $|x||y|n$ subproblems. The following recursive formulation shows how to compute T.

For each prefix $u \neq x$ of x and prefix $v \neq y$ of y, the subproblems $T(u, v, n)$ will be our base cases. The base cases are computed by the following formula.

$$T(u,v,n) = \begin{cases} \text{True} & \text{if either } us_n \text{ is a prefix of } x \text{ or } vs_n \text{ is a prefix of } y \\ \text{False} & \text{otherwise} \end{cases}$$

Let $1 \leq i < n$. Also let $u \neq x$ and $v \neq y$ be prefixes of x and y respectively. Let $u' = us_i$ if $us_i \neq x$ and $u' = \varepsilon$ otherwise. Also let $v' = vs_i$ if $vs_i \neq y$ and $v' = \varepsilon$ otherwise. The following recursion computes $T(u, v, i)$.

$$T(u,v,i) = \begin{cases} \text{True} & \text{if } us_i \text{ is a prefix of } x \text{ and } T(u', v, i+1) \text{ is True} \\ \text{True} & \text{if } vs_i \text{ is a prefix of } y \text{ and } T(u, v', i+1) \text{ is True} \\ \text{False} & \text{otherwise.} \end{cases}$$

We can compute each of the subproblems in constant time with careful implementation. Please work out the details of the implementation. The total time complexity is thus $O(|x||y|n)$.

Correctness of the algorithm follows from the following claim.

Claim 0.3. For all $1 \leq i \leq n$, for each prefix $u \neq x$ of x and for each prefix $v \neq y$ of y, the recursive formulation correctly computes $T(u, v, i)$ (as per the definition).

We prove the claim by (reverse) induction on i.

$i = n$ is our base case. Let $u \neq x$ and $v \neq y$ are arbitrary prefixes of x and y respectively. There are two ways of partitioning s_n into two strings s' and s'': either $s' = s_n$ and $s'' = \varepsilon$. or $s' = \varepsilon$ and $s'' = s_n$. Consider the case $s' = s_n$ and $s'' = \varepsilon$. Since $u \neq x$, $us' = us_n$ is prefix of x if and only if us' is a repetition of x. Since v is a prefix of y, $vs'' = v$ is a repetition of y. Therefore, $T(u, v, n)$ is correctly computed by the formula (for the base case). The proof for the other partition is similar.

Let $1 \leq i < n$. For the induction hypothesis, assume that the recursive formulation correctly computes $T(u, v, j)$ for $n \geq j > i$ and for all prefixes $u' \neq x$ of x and all prefixes $v' \neq y$ of y. Let $u \neq x$ be a prefix of x and let $v \neq y$ be a prefix of y. We will show that $T(u, v, i)$ is computed correctly (as per its definition) by the recursive formulation.

For the rest of the discussion, let $u' = us_i$ if $us_i \neq x$ and $u' = \varepsilon$ otherwise. Also let $v' = vs_i$ if $vs_i \neq y$ and $v' = \varepsilon$ otherwise.

Assume that $T(u, v, i)$ is True, that is, $s_1 \ldots s_n$ can be partitioned into s' and s'' such that us' is a repetition of x and vs'' is a repetition of y. There are two cases to consider, either $s' = s_i t'$ and $s'' = t''$ for some t' and t'' or $s' = t'$ and $s'' = s_i t''$ for some t' and t''. In both cases, $s_{i+1} \ldots s_n$ can be partitioned into t' and t''. In the first case, since $us' = us_i t'$ is a repetition of x, u is a prefix of x, and $u \neq x$, it follows that us_i is a prefix of x.

We now show that $T(u', v, i + 1)$ is True as per the definition. Let $u' = us_i$. Clearly, $u't' = us_i t'$ is a repetition of x since $s' = s_i t'$. Let $u' = \varepsilon$ in which case $us_i = x$. Again, since $us' = us_i t' = xt'$ is a repetition of x, it follows that $u't'$ is a repetition of x. Clearly, $vt'' = vs''$ is a repetition of y. This shows that $T(u', v, i+1)$ is True as per the definition. By induction hypothesis, $T(u'.v.i+1)$ is computed correctly. Therefore, the recursive formulation sets the value of $T(u, v, i)$ to True as required. The other case can be argued similarly.

Now assume that $T(u, v, i)$ is False. In other words, there is no partition of $s_i \ldots s_n$ into s' and s'' such that us' is repetition of x and vs'' is a repetition of y. If $T(u, v, i)$ were to be set to True by the recursive formulation, then either us_i is a prefix of x and $T(u', v, i+1)$ is True or vs_i is a prefix of y and $T(u, v', i+1)$ is True. Consider the first case. Since $T(u', v, i + 1)$ is True and it is correctly computed (by induction hypothesis), there exists a partition of $s_{i+1} \ldots s_n$ into t' and t'' such that $u't'$ is a repetition of x and vt'' is a repetition of y. It follows that $s_i \ldots s_n$ can be partitioned into $s_i t'$ and t'' such that $u't'$ is a repetition of x and $vt''$is a repetition of y. To see this, assume that $u' = \varepsilon$ in which case we have $us_i = x$. Since $u't' = t'$ is a repetition of x, we have $us_i t' = xt'$ is a repetition of x since $T(u', v, i + 1)$ is True. If $u' = us_i$, then $us_i t' = u't'$ is a repetition of x. This implies that $T(u, v, i)$ is True which leads to a contradiction. The other case can be argued similarly. In both cases, we get a contradiction and therefore $T(u, v, i)$ is set to False by the recursive formulation.