PSET-3

fatimamahmood13

March 2024

1 Q1

2 Q2

2a Since F cannot have any repeated letter, we know that there exists at most one value of j where $s_i = f_j$. We will create a lookup table so we can easily locate the value of j where $s_i = f_j$. Since F has m letters, this costs us O(m) time. Also, the lookup table will operate as such:
lookup[x]=j where x=f_j. If no such letter exists, then lookup will simply return -1.
We know that we will have to lookup at most n times because S contains n letters and we only want to lookup at the point where $s_i = f_j$ so this costs us O(n) time. $del[j]$ denotes the minimum deletions needed for all possible prefixes of S (up to the current character s[i]) to avoid the subsequence of F ending at character f[j] Now, lets establish our algorithm.
We will iterate over the characters of S so starting from i=1 we will use the lookup table, lookup[s[i]]=j to find where in F is s[i] located. If j=-1 ($i.e$ that letter does not exist in F) we will move onto the next character in S $i.e$ i+1. Otherwise, $del[j]$ = min($del[j-1]$, $del[j]$+1). Why?
If the lookup is successful $i.e$ j¿-1, that means $s_i = f_j$, so we would want to know the number of deletions that occurred for prefixes of S up to the previous character (s[i-1]) to avoid the subsequence of F ending at the previous character `f(j-1)` and the number of deletions for the current prefix (+1 to account for the deletion of s_i). We would then want to find the minimum of these 2 values and that will give us our $del[j]$.
Our final value will be given by $del[m]$ (as F has m characters). To get $del[m]$, the time complexity is O(m+n) as it takes us O(m) to create the lookup table and at most O(n) lookups.

2b String S=s_1,s_2,....s_n that is to say String S consists of characters s_1,s_2,..s_n. String F=f_1,f_2,...f_m that is to say string F consists of characters f_1,f_2,...f_m. Let s[1..i] denote prefix of string S and f[1...j] denote prefix of string F. $del[i][j]$ will denote the minimum deletions needed from the prefix s[1..i] of S to prevent the subsequence f[1..j] of F from existing. There are two possible cases for calculating $del[i][j]$:
1. $s_i \neq f_j$. This means that characters s_i and f_j are not the same. We can say that in this case s[1...i] does not contain f[1...j] is equivalent to the statement s[1...i-1] does not contain f[1...j]. Why? Since we know s_i and f_j are not the same, we know that s_i has no impact on f[1..j] as including this character doesn't affect whether the earlier characters of F (up to j) already exist within S. In simpler terms, adding a non-matching character at the end can't create the subsequence we're trying to avoid if it wasn't there before. In this case, the value in $del[i-1][j]$ will be written into $del[i][j]$.

2. $s_i = f_j$. This means that characters s_i and f_j are the same. We have two options here. We either keep or delete s_i. If we keep s_i, we need to make sure that the remaining String S $i.e$ s[1...i-1] does not contain f[1...j-1]. If that is the case, the deletion will be $del[i-1][j-1]$. If we delete s_i , we only care about

s[1..i-1][j] so our deletions will be $del[i-1][j]+1$ to account for the deletion of s_i.
So for case 2, $del[i][j] = \min(del[i-1][j], del[i-1][j]+1)$

3 Q3

We want to minimize the total cost *i.e* the sum of the distances from each gate to its closest shop. We will be using a bottom up approach. To do this, we will define F(i,s*). This represents the total cost of the best solution for the interval a_1, a_2, ..a_i when we open s* shops within these i locations. Important to note that all the i locations are assigned shops. If we iteratively fill our table F(i,s*) for all possible values ($1 \leq i \leq n$, $1 \leq$ s* $\leq$ s), we can calculate the total cost for the whole airport *i.e* T(i,s).
First, we have to find out how to calculate T(i,s*). Since we are arranging the shops on a straight line along an interval we can imagine there is a "left" and "right" side of this interval. The "rightmost" shop or the shop with the highest position value in the interval will cater to some range of gates starting from some point (lets say j+1) till i (last gate). This means that the remaining gates *i.e* s*-1 gates will cater to the range of gates from 1 till j.
There is also some cost for placing a shop in the interval. We will define this cost to be $Cost(j+1, i)$ which is the cost of placing ONE shop in the interval j+1, ... i. Now if we combine both of these ideas we can calculate T(i,s*):
T(i,s*) $= min_{j<i}$ (T(j, s*-1) + $Cost(j+1, i)$)
Basically, the T(j, s*-1) gives us the minimum cost from previously calculated sub problem of placing s*-1 shops in the left part of the a_1, a_2, ..a_i interval. The $Cost(j+1, i)$ gives us the cost of placing a shop in the "rightmost" location (j+1) and catering the remaining gates in the interval j+1,...i.
Our base case is T(i,1) which gives us the situation where we only need to place 1 (s*=1) shop for the entire interval a_1, a_2, ..a_i so $Cost(1, i)$ will give us the cost of placing 1 shop in the interval to cater to all the gates.
Now, we need to figure out an optimal value for j which is basically our break-point. Intuitively, to find the minimum cost, our shop should be placed at approximately the median of gate positions in the interval because if the gates are evenly distributed, then the distance from the gates to the shop on either side will be halved. So lets say we have some interval a_m, ..a_n. The cost $Cost(p, q)$ will be calculated in time O(q-p+1). How?
Finding the median takes us a constant O(1) time. We will iterate over all the gate positions and note that for each gate on the left side of the median, the distance to the shop is basically the the difference between the current gate position and median position. This is similar for the gates on the right side of the median. If you perform the calculations (do it yourself its not that hard), we will get time complexity O(q-p+1) for calculating $Cost(p, q)$.
Next, we prove that the time complexity will be O(n^3). There are total n gates so there can be (n*(n+1) /2) unique intervals. As we noted, time complexity for calculating $Cost(p, q)$ will be O(q-p+1) and in the worst case it will be O(n) (p=1 , q=n). Thus, calculating all $Cost(p, q)$ takes us O((n*(n+1) /2 * n) which is O(n^3).

4 Q4

4a

4b

4c