CS 577 Homework 4

Jiaxin Li
Sunjun Gu

10/04/2020

1 Question 1

Algorithm 1 SmallestLength

Input: 2 Arrays A[1,..., m], B[1,...,n] representing two binary sequences a and b of length m and n respectively.
Output: The smallest length of a sequence **c** such that both a and b are sub-sequences of c.

```
0: procedure SmallestLength(A, B)
1: m← length of A
2: n← length of B
3: Length[1, ..., n + 1][1, ..., m + 1] ← a new 2-d array with size (n + 1) × (m + 1)
4: for j = n + 1 → 1 do
5:    for i = m + 1 → 1 do
6:       if j = n + 1 then
7:          Length[i][n + 1] ← m − i + 1
8:       else if i = m + 1 then
9:          Length[m + 1][j] ← n − j + 1
10:      else
11:         if A[i] == B[j] then
12:            Length[i][j] ← Min(1 + Length[i + 1][j + 1], 1 + Length[i][j + 1], 1 + Length[i + 1][j])
13:         else
14:            Length[i][j] ← Min(1 + Length[i][j + 1], 1 + Length[i + 1][j])
15:         end if
16:      end if
17:   end for
18: end for
19: return Length[1][1]
19: end procedure=0
```

Proof of Correctness :

Here we use **bottom-up** implemetation of this algorithm.
The proposition here **SmallestLength(**$A[i, ...n], B[j, ...m]$**)** is that given two binary sequences A and B of lengths m and n respectively, the SmallestLength algorithm could correctly output the smallest length of a sequence C such that both A and B are subsequences of C.
Firstly find the recurrence relation between neighboring states in the matrix.
Recurrence Relation
$Length[i][n + 1] \leftarrow m - i + 1$, where $j = n + 1$
$Length[m + 1][j] \leftarrow n - j + 1$, where $i = m + 1$
$Length[i][j] \leftarrow Min(1+Length[i+1][j+1], 1+Length[i][j+1], 1+Length[i+1][j])$ where A[i]=B[j]
$Length[i][j] \leftarrow Min(1 + Length[i][j + 1], 1 + Length[i + 1][j])$ where A[i]≠B[j]

Base case : The input are (A[m+1,m], B[j,...,n]) or (A[i,...,m], B[n+1,n])When one of the binary sequences (Suppose it is A) is empty (As we defined on hw before, A[m+1,m] is empty string), The common sequence **c** is just the other binary sequence (Suppose it is B). So the length of common sequence is just the length of B. The length of B[j,...,n] equaling to $(n - j + 1)$. Therefore line

8∼9 is correct. The same can be obtained when B is empty, C needs to contain at least contain A[i,..,m] as the subsequence. Therefore, the smallest length of C is length (A[i,...,m]) equaling to $(m - i + 1)$. Therefore line 6∼7 is correct. So, the base case is correct!

Inductive Step : Suppose SmallestLength(A, B) can give for all pairs of sequences $(A[i, ..., m], B[j, ..., n]), (A[i-1, ..., m], B[j, ..., n]), (A[i, ..., m], B[j-1, ..., n])$ where $i \leq m+1$, $j \leq n+1$.
To prove the correctness of $SmallestLength(A, B)$ where A[i-1,...,m],B[j-1,...,n]:

Because $i \leq m+1$, $j \leq n+1 \rightarrow 0 < i-1 \leq m$, $0 < j-1 \leq n$.$Line1 \sim line5$ and $Line11 \sim line14$ will be executed.
For the first letter of the common sequence **c**, there are 2 choice, if A[i-1] and B[j-1] are not the same:
(1) **c** has the same letter with A[i-1] at the first position, the problem is reduced to get the length of minimum common sequence of A[i,...,m], B[j-1,...,n] and add 1 to that value (Length[i][j-1] + 1, which is correct under inductive assumption).
(2) **c** has the same letter with B[j-1] at the first position, the problem is reduced to get the length of minimum common sequence of A[i-1,..., m], B[j,...,n] and add 1 to that value (Length[i-1][j] +1, which is correct under inductive assumption).
The smaller one will be the value of Length[i-1][j-1]
If the A[i-1] and B[j-1] are the same: then there is one more choice, which is reduce this problem to get the length of minimum common seuquence of A[i,...m] and B[j,...,n] and add 1 to that value,Length[i][j]+1 which is correct under inductive assumption
In such case, the smallest value among 3 cases will be the value of Length[i-1][j-1].
Therefore, by mathematical inductive principle, the proposition is correct.
And in the algorithm all of values that are supposed to be correct are calculated before length[i-1][j-1].

Proof of Time and Space Complexity :

Line 1∼3 take constant time.

Line 4∼18, run in a for loop from n+1 to 1,the outer loop takes n+1 times to move from the empty sequence B(end of sequence) to the entire B(beginning of sequence);Then inside the for loop, run in another inner loop from m+1 to 1 takes m+1 times from the empty sequence A to the entire A. In the inner for loop, line 6-9, check the situation then store the value(because one sequence is empty, the smallest length of C is the left length of the non-empty sequence) into array only take constant time, and line 11-12, when A,B's first letter is the same, we have three next steps(pick the same one,and A,B all decrease one letter;pick the same one, but A or B decrease one letter, the other no change) available to be taken. We choose the minimum of the three steps length where the three values Length[i+ 1][j+ 1],Length[i][j+ 1],Length[i+ 1][j] are all taken from the prior completed array.Therefore, these steps take constant time. Line 14 is when A,B's first letter is not the same, we have two next steps(pick the A or B's first letter,and A or B decrease one letter, the other no change) available to be taken. We choose the minimum of the two steps length where the two values Length[i][j+ 1],Length[i+ 1][j] are all taken from the prior completed array.These steps take constant time. Inside the inner loop, total work take constant time. Therefore, The outer and inner for loop take (n+1)×(m+1)O(1) = O(n·m+n+m+1) = O(n·m).And the return value also taken from the 2-d array in constant time. All the work time is O(1) + O(n·m) = (n·m)

Combine the above analysis, the algorithm we designed solves the smallest length dynamic programming problem in $O(n \cdot m)$ time.

For space complexity, because we create a 2-d array with size (n+1)×(m+1),store and take values from this array. With ignorance of the O(1) space for m,n, the total space is O(n·m)

2 Question 3

2.1 3(a)

Algorithm 2 Maxinterferencee

Input: A n × n matrix of each pair of antennas' interferencee intensity s, where s_ij is the interferencee intensity between antennas i and j in the same location ($1 \leq i \leq j \leq n$)

Output: The maximum interferencee values for the subinstances consisting of antennas i through j for all $1 \leq i \leq j \leq n$ on one location($k = 1$)

```
0: procedure MAXINTERFERENCEE(s)
1: interferencee[1, ..., n][1, ..., n] ← a new 2-d array with dimension n · n, initialized with -Inf
2: for j = 1, ..., n do
3:    for i = j, ..., 1 do
4:       if i = j then
5:          interferencee[i][j] ← 0
6:       else
7:          interferencee[i][j] ← MAX(s_ij,interferencee[i + 1][j],interferencee[i][j − 1])
8:       end if
9:    end for
10: end for
11: return interferencee
11: end procedure=0
```

Proof of Correctness :

We used a **bottom-up** implementation of this algorithm, which returns an n by n matrix $interference$, where $interference[i][j]$ is the maximum interference value for the sub-instance consisting of antennas i through j ($1 \leq i \leq j \leq n$). To fill this table, we firstly find the recurrence relation between neighboring states in the matrix.

Recurrence Relation $interference[i][j] = 0$, where $i = j$

$interference[i][j] = Max(s_{ij}, interference[i + 1][j], interference[i][j - 1])$, where $i < j$

Proposition The proposition here P(d) is recurrence relation above is correct, where $d = j - i$.

Base case:

When $d = j - i = 0$, there is only 1 antennas in the location, so the interference must be 0. So $interference[i][j] = 0$ is correct.

Inductive Step:

Suppose P(k), $0 \leq k \leq n - 1$ is correct, which means when $j - i = k$, $interference[i][j]$ is the correct max interference between i-th antennas and j-th antennas.

To prove the correctness of P(k+1):

Because

$$\bigcup_{i \leq x \leq y \leq j} s_{xy} = \bigcup_{i+1 \leq x \leq y \leq j} s_{xy} \cup \bigcup_{i \leq x \leq y \leq j-1} s_{xy} \cup s_{ij}$$

$$Max(\bigcup_{i \leq x \leq y \leq j} s_{xy}) = Max(\bigcup_{i+1 \leq x \leq y \leq j} s_{xy} \cup \bigcup_{i \leq x \leq y \leq j-1} s_{xy} \cup s_{ij})$$

$$Max(\bigcup_{i \leq x \leq y \leq j} s_{xy}) = Max(Max(\bigcup_{i+1 \leq x \leq y \leq j} s_{xy}), Max(\bigcup_{i \leq x \leq y \leq j-1} s_{xy}), s_{ij})$$

The formula above is equivalent to $Interference[i][j] = Max(s_{ij}, Interference[i+1][j], Interferencee[i][j-1])]$

$i - j = k + 1 \Rightarrow j - (i + 1) = k, (j - 1) - i = k$

Under the assumption P(k) is correct, $Interference[i+1][j]$ and $Interference[i][j-1]$ have correct value, so $Interference[i][j]$ will be correct.

Therefore $P(k) \Rightarrow P(k + 1)$

By mathematical inductive principle, P(d) is correct. Because the algorithm used a bottom-up implementation, the last thing to consider is the order of filling the states in the Interference matrix.

By enumerate j from 1 to n and i from j to i, Interference[i+1][j] and Interference[i][j-1] both have been filled before Interference[i][j]. So overall the algorithm is correct.
Proof of Time and Space Complexity :

Line 1: create and initialize a 2-d array with dimension n·n, it takes O(n^2) time.

Line 2∼10: run in a for loop from 1 to n,the outer loop takes n times;Then inside the for loop, run in another inner for loop from the outer for loop value to 1 and takes j times which at most is n times. Line 4-5, check whether i=j that is only one antenna, if so no interference, store 0 into interference[i][j] and takes constant time. Line 7, find and store the maximum of s_{ij}, Interference[i+1][j],Interference[i][j−1]. The s_{ij} is given as output and Interference[i+1][j],Interference[i][j−1] could be got from the prior completed array value. Therefore, it works on constant time. Therefore, the outer and inner for loop totally take $(1 + 2 + 3, ..., +n) \times O(1) = O(n^2)$ time.

Line 11: return the whole interference array, and through it, we can find any max interference from i to j antennas with constant time.

Combine all the work time, O(1)+O(n^2)=O(n^2), therefore the algorithm runs in $O(n^2)$ time. For space complexity, because we create a 2-d array with dimension n×n,store and get values from this array. The total space is O(n·n)

2.2 3(b)

Proof of Correctness:
Sub-problems:
The main idea used here is to split the whole problem into 2 sub-problems: Divide n antennas into 2 parts, the first part is put in the first k-1 locations, and the second part is put in the last location. To be noticed, each part could be empty set given the restriction in the question.
Recurrence Relation:
Firstly, here are the clarifications of the terms that I will use bellow:
Let F(i,j,k) be the minimum sum of interference between i-th antennas and j-th antennas when these antennas are put into k locations when $i \leq j$. When $i > j$, F(i,j,k) is 0, which means the subset is an empty set. The main recurrence relation used is:
$F(1, j, k) = Min(F(1, \theta, k - 1) + F(\theta + 1, j, 1))$, where $0 \leq \theta \leq j$
In such text, the problem is asking for F(1,n,k).
Proposition:
The proposition used here is that $P(d)$ The algorithm can return correct values for all $F(1, j, d)$, where $1 \leq j \leq n$
Base Case:
When d = 1, the algorithm directly returns the corresponding value calculated by 3(a)'s algorithm ($F(1, j, 1) = Interference[1][j]$, where $1 \leq j \leq n$).
Inductive Step:
Suppose P(k) $1 \leq k < n$ is correct, which means the algorithm can returns correct values for $F(1, j, k)$, where $1 \leq j \leq n$.
To prove the correctness of P(k+1):
For each F(1,j,k+1):
When split n antennas into 2 parts, $\theta - th$ is the last one in the first one. The first part goes to first k-1 locations, the second part goes to the last location
For antennas 1 to j, there are j+1 split method (1) When $\theta = 0$, which means we put all of the antennas in the second part. The interference of the first part (empty) will be zero, so the problem is reduced to putting all of the antennas into the second part (1 location). So one possible value of F(1,j,k+1) is **F(1,j,1)**.
(2) When $\theta = j$, which means we put all of the antennas in the first part. The interference of the second part (empty) will be zero, so the problem is reduced to puuting all of the antennas into the first part (k-1) locations. So one possible value of F(1,j,k+1) is **F(1,j, k)**, which can be calculated correctly by this algorithm under the assumption.
(3) When $1 \leq \theta < j$, which means both parts are not empty. The recurrence relation $possiblevalue = F(1, \theta, k) + F(\theta + 1, j, 1)$ applies. Given P(k) is true, we can get the correct values of $F(1, \theta, k)$

Algorithm 3 Mininterferencee

Input: A n × n matrix of each pair of antennas' interferencee intensity S, where S_{ij} is the interferencee intensity between antennas i and j in the same location ($1 \leq i \leq j \leq n$); Number of locations k, which $1 \leq k < n$

Output: the minimum total interferencee min_i.

0: **procedure** MININTERFERENCEE(S, k)
1: $MaxInterfernce \leftarrow$ Maxinterferencee(S) * Use the algorithm in 3(a)
2: **if** $k == 1$ **then**
3: **return** $Maxinterference[1][n]$
4: **else**
5: $Previous \leftarrow Maxinterference[1, 1 \rightarrow n]$ *The first row of matrix Mainterference
6: $Current \leftarrow$ A new array of length n
*When the location number is 2 to k-1, we need to calculate n min values to calculate the next layer's values
7: **for** t = 2,...,k-1 **do**
8: **for** $j = 1 \rightarrow n$ **do**
9: **for** $\theta = 0 \rightarrow j$ **do**
10: $Values[0, ..., j] \leftarrow$ a new 1-d array with length j+1
11: **if** $\theta = 0$ **then**
12: $Values[\theta] \leftarrow Maxinterference[1][j]$
13: **else if** $\theta == j$ **then**
14: $Values[\theta] \leftarrow Previous[j]$
15: **else if** $1 \leq \theta < j$ **then**
16: $Values[\theta] \leftarrow Previous[\theta] + Maxinterference[\theta + 1][j]$
17: **end if**
18: **end for**
19: $Current[j] = Min(Values[0, ..., j])$
20: **end for**
21: $Previous \leftarrow Current$
22: $Current \leftarrow$ A new array of length n
23: **end for**
* But for the k-th layer, only 1 min value need to be calculated, because there is no next layer!
24: $Values[0, ...n] \leftarrow$ a new 1-d array with length n+1
25: **for** $\theta = 0 \rightarrow n$ **do**
26: **if** $\theta = 0$ **then**
27: $Values[\theta] \leftarrow Maxinterference[1][n]$
28: **else if** $\theta == n$ **then**
29: $Values[\theta] \leftarrow Previous[n]$
30: **else if** $1 \leq \theta < n$ **then**
31: $Values[\theta] \leftarrow Previous[\theta] + Maxinterference[\theta + 1, n]$
32: **end if**
33: **end for**
34: $min_i \leftarrow Min(Values[0, ..., n])$
35: **return** min_i
36: **end if**
36: **end procedure**=0

and $F(\theta+1, j, 1)$ So overall, we get j+1 possible values for F(1,j,k+1), And its values of all of the possible distributions of those antennnas. So by Min() function, we can get the smallest value among those j+1 values.
For all j which belongs to [1,n], the above proof is true. Therefore, P(k+1) is correct given P(k) is correct.
By mathematical inductive principle, P(d) is true.
Given the algorithm can return correct values of all $F(1, j, k)$, where $1 \leq j \leq n$, it can also return correct value of F(1,n,k), which is the goal of this problem.

Proof of Time Complexity :

Line 1: call the algorithm Maxinterferencee(S) and store the 2-d array on MaxInterfernce with $O(n^2)$ time.

Line 2-3: if only put antennas on one location(we can see it as the first location), it is the same as 3(a) algorithm, return the value of Maxinterference[1, n]. Total work takes constant time.

Line 4-36: for the worst situation, that is we need to put antennas on different locations. Create two length n array Previous and Current with O(n) time. The Previous array stores Maxinterference[1,1:n], Current is the length n new array.

Line 7-23, use the first t = 2 to k-1 for loop to talk about different locations situation with (k-2) loops; then for the second j = 1 to n for loop talk about different number of antennas situations put on this location with n loops ; then for the third $\theta = 0$ to j for loop talk about how the number of antennas distributed on this location and the before location with j loops. From the Line 10 to 17, we create a new i-d array Values with length j+1. By the recurrence relation, if θ=0 that is all the antennas is on the t-th location, get the Maxinterference[1][j] and store the max interference on Values[θ];

if θ=j that is all the antennas is not on the t-th location, but store on the previous location,therefore get the Previous[j] that is the previous location and store the max interference on Values[θ];

if $1 \leq \theta < j$, that is the j antennas both stores on the t-th location from $\theta+1$ to j antennas and the previous location for 1 to θ, therefore, the interference is Previous[θ]+Maxinterference[θ+1, j]. All this procedures on the third loop is about check situation, get value by index from known array and store value. So the work takes constant time. Therefore, the third loop takes $j \cdot O(1) = O(j)$ and worst O(n) time

After the third loop, we need to find the minimum interference of antennas distribution and store it on the Current[j] on the second loop,it also takes O(j) time and worst O(n) time. Therefore, all the second loop takes worst $n(O(n) + O(n)) = O(n^2)$ time.

After the second loop, we get the minimum interference for the t-th location, when go to the next location, the t-th location becomes previous lication, therefore, we need to update the Previous and Current arraies with worst O(n) time.And inside the first loop, total work is $O(n^2) + O(n) = O(n^2)$ time. Therefore, all the first for loop takes $(k-2) \cdot O(n^2) = O(kn^2)$time.

Line 24-34,for the total k locations, create a 1-d array Values to store the interference. When k-th location has different nummber of antennas from 0 to n(for loop from line 25 to 33), if all put on the k-th location, it is Maxinterference[1][n];if all put on the previous location, it is Previous[n]; if the antennas are both on k-th location and previous location, it is Previous[θ] +Maxinterference[θ+ 1, n]. And all these work is constant time. So the for loop taks O(n) time. And find the returning minimum interference of different distribution of antennas takes O(n) time.

Conbine all the work together, it taks $O(1)+O(n)+O(kn^2) = O(kn^2)$ time.So the overall time complexity of algorithm is $O(kn^2)$