

SHANGHAI TECH UNIVERSITY

CS240 Algorithm Design and Analysis
Spring 2024
Problem Set 2

Zhou Shouchen

StudentID: 2021533042

Due: 23:59, April 11, 2024

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

Problem 1:

Suppose you are given a set of intervals $I_1 = [s_1, t_1], \dots, I_n = [s_n, t_n]$, which can possibly overlap. Your task is to select a set of points p_1, \dots, p_m so that each interval intersects at least one of the points. Give an algorithm to minimize the number of selected points m , and prove that your algorithm is optimal.

Solution:

We can sort the intervals by the each interval's end point t_i .

And then traverse all intervals in the order we sorted:

1. If the interval is intersected by the selected points, then we do nothing.
2. If no intersection, then we add a point at the end of the interval t_i .

Proof correctness:

Suppose p_1, \dots, p_m is the set of points of our algorithm's selection.

And q_1, \dots, q_r is the set of points of the optimal selection.

Suppose $p_1 = q_1, \dots, p_t = q_r$ with the largest possible r .

And suppose the first uncovered interval in the sorted order is $I_i = [s_i, t_i]$.

And there are 3 possible situations.

1. $q_{r+1} \leq p_{r+1}$ and $q_{r+1} \notin I_i$

Then q_{r+1} is actually a wasted selection, we can just replace q_{r+1} with p_{r+1} .

2. $q_{r+1} \leq p_{r+1}$ and $q_{r+1} \in I_i$

Since $\forall k < i$, we know that the interval I_k has already intersected with p_1, \dots, p_r .

And $\forall k > i$, we know that $t_k \geq t_i$, so we can get that for all intervals I_k which is intersected with q_{r+1} , it must be intersected with p_{r+1} . So we can just replace q_{r+1} with p_{r+1} .

3. $q_{r+1} > p_{r+1}$

From our algorithm, we could know that we are setting p_{r+1} to the end of I_i , and I_i has no intersections with p_1, \dots, p_r . But since $q_{r+1} > p_{r+1}$, and $\forall k \geq r+1, q_k \geq q_{r+1}$, so the optimal solution will have no intersections with interval I_i . Which is impossible. So we always have $q_{r+1} \leq p_{r+1}$.

So above all, we can say that our solution is always equal or better than the assumed optimal solution, i.e. our algorithm will generate one of the optimal solutions.

The time consumption: Sorting the intervals takes $O(n \log n)$ time, and traverse all intervals to get the number of points to need takes $O(n)$ time, so the total time complexity is $O(n \log n)$.

The space complexity is $O(n)$ for storing the intervals.

Problem 2:

Suppose you need to perform n tasks, and you can only perform one task at a time. It takes t_i time to perform the i 'th task, and the task has an importance of $w_i > 0$. Let C_i be the completion time of the i 'th task, i.e. the time when you finish performing the task. Find an ordering of the tasks to minimize their weighted completion time, defined as $\sum_{i=1}^n w_i C_i$. Your algorithm should work in $O(n \log n)$ time, and you need to analyze the algorithm's time complexity and prove its correctness.

Solution:

We can sort the tasks by $\frac{w_i}{t_i}$ in descending order.

Proof correctness:

W.L.O.G, we can just consider the two neighboring tasks i and j , where $j = i+1$, and we now only consider the order of i, j .

In our sorting order, we have $\frac{w_i}{t_i} \geq \frac{w_j}{t_j}$, since we have $\forall i, w_i > 0$, then we can get that $w_j \cdot t_i - w_i \cdot t_j \leq 0$.

Suppose the start time of the task i is t_0 , s_0 is the weighted completion time before task i , and s_3 be the weighted completion time after task j . Then the weighted completion time for the order i, j is

$$s_1 = s_0 + w_i \cdot (t_0 + t_i) + w_j \cdot (t_0 + t_i + t_j) + s_3$$

If we swap i and j , then the weighted completion time for the order j, i is

$$s_2 = s_0 + w_j \cdot (t_0 + t_j) + w_i \cdot (t_0 + t_i + t_j) + s_3$$

and we can get that

$$s_1 - s_2 = w_j \cdot t_i - w_i \cdot t_j \leq 0$$

So we could know that with such sorting order, we can always get a smaller weighted completion time.

So the sorting order is one of the optimal one.

The time consumption: sorting takes $O(n \log n)$ time for storing, and travels all tasks to compute the weighted completion time takes $O(n)$ time, so the total time complexity is $O(n \log n)$.

The space complexity is $O(n)$ for storing each tasks' information.

Problem 3:

A thief is planning to rob houses along a street. Each house has a certain amount of money stashed, and the thief can rob any set of houses, as long as he does not rob any adjacent houses. Determine the maximum amount of money the thief can steal.

Solution:

Let $dp(i, 0)$ be the maximum amount of money the thief can steal from the first i houses without robbing the i -th house, and $dp(i, 1)$ be the maximum amount of money the thief can steal from the first i houses with robbing the i -th house.

For the initial condition, we have $dp(1, 0) = 0$ and $dp(1, 1) = a_1$. As if the thief do not steal from the first house, he would get no money, and if he steals from the first house, he would get a_1 money.

And other states are set to be 0.

Then we have the following recursive formula:

If the thief does not steal from the i -th house, he can decide whether to steal from the $(i - 1)$ -th house or not. So he just need to make the optimal choice, i.e. $dp(i, 0) = \max\{dp(i - 1, 0), dp(i - 1, 1)\}$.

If the thief steals from the i -th house, he must not steal from the $(i - 1)$ -th house. i.e. $dp(i, 1) = dp(i - 1, 0) + a_i$.

Finally, the maximum amount of money the thief can steal is $\max\{dp(n, 0), dp(n, 1)\}$. As he can choose to steal from the n -th house or not. And the final decision is the maximum amount of money he can steal from the first n houses.

The pseudo-code is shown in Algorithm 1. For loop in the pseudo-code in form '**for** $i \leftarrow 1$ **to** n ' represents $i = 1, 2, \dots, n$ sequentially.

Algorithm 1 Maximum stolen money

```
1: Input: Number of house  $n$ , each house's money number  $a_1, a_2, \dots, a_n$ .
2: Output: The maximum number of money could steal  $ans$ .
3:  $dp(1, 0) \leftarrow 0, dp(1, 1) \leftarrow a_1$ 
4: for  $i \leftarrow 2$  to  $n$  do
5:    $dp(i, 0) \leftarrow \max\{dp(i - 1, 0), dp(i - 1, 1)\}$ 
6:    $dp(i, 1) \leftarrow dp(i - 1, 0) + a_i$ 
7: end for
8:  $ans \leftarrow \max\{dp(n, 0), dp(n, 1)\}$ 
9: return  $ans$ 
```

The time complexity is $O(n)$.
The space complexity is $O(n)$.
Where n is the number of houses.

Problem 4:

Given three sequences L_1, L_2 and L , design an efficient algorithm to check if L_1 and L_2 can be interleaved to produce L . For example, the sequences $L_1 = \text{aabb}$ and $L_2 = \text{cba}$ can be interleaved into sequence $L = \text{acabbab}$, but L_1 and L_2 cannot be interleaved into sequence $L = \text{aaabbbbc}$. Analyze the time and space complexity of your algorithm as a function of the lengths of L_1 and L_2 .

Solution:

Let $dp(i, j)$ be a boolean value, which means whether the first i characters of L_1 and the first j characters of L_2 can be interleaved to produce the first $i + j$ characters of L .

The initial condition is $dp(0, 0) = 1$, as if L_1 and L_2 are both empty, then they can be interleaved to produce an empty sequence.

And set $dp(i, 0) = 1$ if $L_1[1 : i]$ can be interleaved to produce $L[1 : i]$, otherwise $dp(i, 0) = 0$.

Similarly, set $dp(0, j) = 1$ if $L_2[1 : j]$ can be interleaved to produce $L[1 : j]$, otherwise $dp(0, j) = 0$.

And other state are set to be 0.

Then we have the following recursive formula:

1. If $L_1[i] == L[i + j]$, then $dp(i, j) \leftarrow dp(i, j) \vee dp(i - 1, j)$.

As if $L_1[i]$ is the same as $L[i + j]$, then if $L_1[1 : i - 1]$ and $L_2[1 : j]$ can be interleaved to produce $L[1 : i + j - 1]$, then $L_1[1 : i]$ and $L_2[1 : j]$ can be interleaved to produce $L[1 : i + j]$.

2. If $L_2[j] == L[i + j]$, then $dp(i, j) \leftarrow dp(i, j) \vee dp(i, j - 1)$.

As if $L_2[j]$ is the same as $L[i + j]$, then if $L_1[1 : i]$ and $L_2[1 : j - 1]$ can be interleaved to produce $L[1 : i + j - 1]$, then $L_1[1 : i]$ and $L_2[1 : j]$ can be interleaved to produce $L[1 : i + j]$.

The above two conditions can hold at the same time. If they all hold, the two situations should be all considered.

Finally, the answer is $dp(|L_1|, |L_2|)$. As if the first $|L_1|$ characters of L_1 and the first $|L_2|$ characters of L_2 can be interleaved to produce the first $|L_1| + |L_2|$ characters of L , then L_1 and L_2 can be interleaved to produce L .

The pseudo-code is shown in Algorithm 2.

For loop in the pseudo-code in form ‘**for** $i \leftarrow 1$ **to** n ’ represents $i = 1, 2, \dots, n$ sequentially.

Algorithm 2 Interleaved the object sequence

```

1: Input: Two sequences  $L_1, L_2$  and  $L$ .
2: Output: Whether  $L_1$  and  $L_2$  can be interleaved to produce  $L$ .
3:  $dp(0, 0) \leftarrow 1$ 
4: for  $i \leftarrow 1$  to  $|L_1|$  do
5:   if  $L_1[i] == L[i]$  then
6:      $dp(i, 0) \leftarrow 1$ 
7:   end if
8: end for
9: for  $j \leftarrow 1$  to  $|L_2|$  do
10:  if  $L_2[j] == L[j]$  then
11:     $dp(0, j) \leftarrow 1$ 
12:  end if
13: end for
14: for  $i \leftarrow 1$  to  $|L_1|$  do
15:  for  $j \leftarrow 1$  to  $|L_2|$  do
16:     $dp(i, j) \leftarrow 0$ 
17:    if  $L_1[i] == L[j]$  then
18:       $dp(i, j) \leftarrow dp(i, j) \vee dp(i - 1, j)$ 
19:    end if
20:    if  $L_2[j] == L[j]$  then
21:       $dp(i, j) \leftarrow dp(i, j) \vee dp(i, j - 1)$ 
22:    end if
23:  end for
24: end for
25: return  $dp(|L_1|, |L_2|)$ 

```

The time complexity is $O(nm)$.

The space complexity is $O(nm)$.

If we use the scrolling array method to optimize, the space complexity can be reduced to $O(n + m)$.

Where n is the length of L_1 , m is the length of L_2 .

Problem 5:

Suppose you are given a propositional logic formula containing only the terms \wedge , \vee , T and F, without any parentheses. You want to find out how many different ways there are to correctly parenthesize the formula so that the resulting formula evaluates to true. For example, the formula $T \vee F \vee T \wedge F$ can be correctly parenthesized in 5 ways:

$$\begin{aligned} & (T \vee (F \vee (T \wedge F))) \\ & (T \vee ((F \vee T) \wedge F)) \\ & ((T \vee F) \vee (T \wedge F)) \\ & (((T \vee F) \vee T) \wedge F) \\ & ((T \vee (F \vee T)) \wedge F) \end{aligned}$$

Of these, 3 evaluate to true: $((T \vee F) \vee (T \wedge F))$, $(T \vee ((F \vee T) \wedge F))$ and $(T \vee (F \vee (T \wedge F)))$.

Give a dynamic programming algorithm to solve this problem. Describe your algorithm, including a clear statement of your dynamic programming equation, show that it is correct, and prove its running time.

Solution:

Let $dp(i, j, val)$ be the number of ways to parenthesize the formula a_i, a_{i+1}, \dots, a_j so that the resulting formula evaluates to val .

Let $operator(k)$ be the operator between a_k and a_{k+1} .

Where a_i, a_{i+1}, \dots, a_j is a subsequence of the original formula.

The initial condition is set to be $dp(i, i, a_i) = 1$, and $dp(i, i, \neg a_i) = 0$. And other state are set to be 0.

So we have the following recursive formula: For an interval $[i, j]$ ($i \neq j$), then it could be separated into two parts $[i, k]$ and $[k + 1, j]$, where $k \in [i, j - 1]$.

Then we have the following recursive formula: If we want the formula in the interval $[i, j]$ to evaluate to 0, then we have the following cases:

- If $operator(k) == \wedge$, then the formula in the interval $[i, j]$ evaluates to 0 if and only if at least one interval $[i, k]$ or $[k + 1, j]$ evaluates to 0.
- If $operator(k) == \vee$, then the formula in the interval $[i, j]$ evaluates to 0 if and only if the formula in the interval $[i, k]$ evaluates to 0 and the formula in the interval $[k + 1, j]$ evaluates to 0.

If we want the formula in the interval $[i, j]$ to evaluate to 1, then we have the following cases:

- If $\text{operator}(k) == \wedge$, then the formula in the interval $[i, j]$ evaluates to 1 if and only if the formula in the interval $[i, k]$ evaluates to 1 and the formula in the interval $[k + 1, j]$ evaluates to 1.
- If $\text{operator}(k) == \vee$, then the formula in the interval $[i, j]$ evaluates to 1 if and only if at least one interval $[i, k]$ or $[k + 1, j]$ evaluates to 1.

So the dynamic programming equation is:

$$\begin{aligned}
 dp(i, j, 0) &= \sum_{k=i}^{j-1} \begin{cases} dp(i, k, 0) * dp(k+1, j, 0) \\ + dp(i, k, 0) * dp(k+1, j, 1) \\ + dp(i, k, 1) * dp(k+1, j, 0) & \text{if } \text{operator}(k) == \wedge \\ dp(i, k, 0) * dp(k+1, j, 0) & \text{if } \text{operator}(k) == \vee \end{cases} \\
 dp(i, j, 1) &= \sum_{k=i}^{j-1} \begin{cases} dp(i, k, 0) * dp(k+1, j, 1) \\ + dp(i, k, 1) * dp(k+1, j, 1) \\ + dp(i, k, 1) * dp(k+1, j, 0) & \text{if } \text{operator}(k) == \vee \\ dp(i, k, 1) * dp(k+1, j, 1) & \text{if } \text{operator}(k) == \wedge \end{cases}
 \end{aligned} \tag{1}$$

Finally, the answer is $dp(1, n, 1)$. As we want the whole formula to evaluate to 1.

The pseudo-code is shown in Algorithm 3.

For loop in the pseudo-code in form ‘**for** $i \leftarrow 1$ **to** n ’ represents $i = 1, 2, \dots, n$ sequentially.

The time complexity is $O(n^3)$.

The space complexity is $O(n^2)$.

Where n is the number of propositions.

Algorithm 3 Number of True Statements

```
1: Input: Number of propositions  $n$ , propositions  $a_1, a_2, \dots, a_n$ , operators  
    $b_1, b_2, \dots, b_{n-1}$   
2: Output: The number of True Statements  
3: for  $i \leftarrow 1$  to  $n$  do  
4:    $dp(i, i, a_i) \leftarrow 1$   
5:    $dp(i, i, \neg a_i) \leftarrow 0$   
6: end for  
7: for  $i \leftarrow 1$  to  $n$  do  
8:   for  $j \leftarrow i + 1$  to  $n$  do  
9:     for  $k \leftarrow i$  to  $j - 1$  do  
10:      if  $b_i == \wedge$  then  
11:         $dp(i, j, 0) \leftarrow dp(i, k, 0) * dp(k + 1, j, 0)$   
           $+ dp(i, k, 0) * dp(k + 1, j, 1)$   
           $+ dp(i, k, 1) * dp(k + 1, j, 0)$   
12:         $dp(i, j, 1) \leftarrow dp(i, k, 1) * dp(k + 1, j, 1)$   
13:      else  
14:         $dp(i, j, 0) \leftarrow dp(i, k, 0) * dp(k + 1, j, 0)$   
15:         $dp(i, j, 1) \leftarrow dp(i, k, 1) * dp(k + 1, j, 1)$   
           $+ dp(i, k, 0) * dp(k + 1, j, 1)$   
           $+ dp(i, k, 1) * dp(k + 1, j, 1)$   
16:      end if  
17:    end for  
18:  end for  
19: end for  
20: return  $dp(1, n, 1)$ 
```

Problem 6:

Consider a weighted directed graph G with n vertices and m edges, where each edge (i, j) has a positive integer weight $w_{i,j}$. A walk is a sequence of not necessarily distinct vertices v_1, v_2, \dots, v_k , such that any two consecutive vertices v_i, v_{i+1} are connected by an edge. The length of the walk is the sum of the weights of the edges in the walk. Design an algorithm to find the number of different walks from a vertex s to another vertex t which have length exactly L , and analyze the time complexity of your algorithm.

Solution:

Let $dp(x, val)$ to be the number of different walks from s to x with length val .

Where x is a vertex in the graph.

The initial condition is set to be $dp(s, 0) = 1$, and $dp(x, t) = 0, \forall (x, t) \neq (s, 0)$.

Then we have the following recursive formula:

For each edge $(i, j) \in E$, then the number of different walks from s to j with length val is the sum of the number of different walks from s to i with length $val - w_{i,j}$.

So we have the following recursive formula:

$$dp(j, val) = dp(j, val) + dp(i, val - w_{i,j})$$

Then we just need to take every edge into consideration, and update the

The pseudo-code is shown in Algorithm 4.

For loop in the pseudo-code in form ‘**for** $i \leftarrow 1$ **to** n ’ represents $i = 1, 2, \dots, n$ sequentially.

Algorithm 4 Number of different walks

```
1: Input: Graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges with weight  $w_{i,j}$ 
2: Output: The number of different walks from  $s$  to  $t$  with length  $L$ 
3:  $dp(s, 0) \leftarrow 1$ 
4:  $dp(x, t) \leftarrow 0, \forall (x, t) \neq (s, 0)$ 
5: for  $val \leftarrow 1$  to  $L$  do
6:   for  $(i, j) \in E$  do
7:      $dp(j, val) \leftarrow dp(j, val) + dp(i, val - w_{i,j})$ 
8:   end for
9: end for
10: return  $dp(t, L)$ 
```

The time complexity is $O(mL)$.

The space complexity is $O(nL + m)$.

Where n is the number of vertices, m is the number of edges, L is the length of the walk.