Cite:

B08902072 黃國銘 B08902127 林歆凱 B08902022 徐敬能 B08902047 汪昊新, B08902063 陳羿穎 B08202029 楊欣翰 B08902009 陳若婕 B08902038 翁浩永, B08902075 林耘平

ADA Homework 2 Hand-written

5. Trick-or-Treating

(1)

The best solution of the given case is to visit the first, second, third and forth professor in order in order to maximize the candies we got. In this case, we can receive 8 + 5 + 1 + 9 = 23 candies. To prove it's the optimal solution, I used contradiction method. If this isn't the optimal solution, then the fifth professor must be included, however, if he/she is included, then we must visit him/her first (he/she rest at 10) and take 10 units of time. Therefore, we won't be able to visit the first and second professor, as a result, 23 candies is the optimal solution.

(2)

I start from the first professor whose rest time and chat time are the shortest, and go through all professors in increasing order. If the current time + the professor's chat time <= the professor's rest time, then I chat with that professor and get the candies. Current time += the professor's chat time. As a go through all professors one time and do O(1) calculation, it has a O(n) time complexity.

I define the optimal substructure first. Let N_i means the OPT having i professors. If i is in OPT, than we have $N_{(i-1)} = N_i - 1$, else, we have $N_{(i-1)} = N_i$. My goal is to find N_n.

To prove the greedy choice won't be worse than the optimal solution. I use contradiction method. Define the greedy choice as if we found N_{i-1}, and the i_{th} professor is still legal, then we put i into the optimal solution set and $N_i = N_{i-1} + 1$. If we don't choose i_{th} professor, and I choose any j greater than i to visit. We can see that for all j, the ending time of choosing j must be later than visiting i_{th} and the j_{th} professor rest later than i_{th} as j > I. In this case, choosing i is always the best choice. We can see that as long as we choose the current legal professor, we reach an solution that is not worse than the optimal solution. QED

(3)

Step 1:

Use quick sort algorithm to sort the professor by r_i in increasing order. It takes O(n*log(n)) time.

Step 2:

Maintain an answer priority queue that keep the one that has biggest t_i on the top. For i from 0 to n, if current time + t_i <= r_i, put it into the priority queue. Else if current time + t_i > r_i, compare t_i and answer.top().t_i. If t_i > answer.top().t_i, continue, else pop the top element and push the i_{th} professor into the answer queue.

For each professor, we need to do at most one push and one pop, which take O(1) and O(log(n)) time, and there are n professors, so we need O(n*log(n)) time to complete this operation.

Time Complexity:

Sum up two steps, it takes O(n * log(n)) + O(n * log(n)) = O(n * log(n)) time.

Proof of Correctness:

For each professor, if the time is still legal, just go and chat with the professor and you will maximize the candies you can get till now! Else, I compare him/her with the top of the priority queue, who has the biggest t_i on the top. So if you have lower time cost than the biggest one, replace it in the queue must be the best choice you can do. While I start from the base case and do the best choice in every step, I will reach the best answer.

(4)

Step 1:

Use quick sort algorithm to sort the professor by R_i in increasing order. It takes O(n*log(n)) time.

Step 2:

Build a N * M dp_table. In each coordinate dp_table[i][j], save the value of the least time to get j candies with first i professors. I go through the whole table once to fill in the value and find the maximum possible answer. For every coordinate I go through, I consider three conditions.

First, if j > sum(C_0 : C_i), set the value to INT_MAX which indicates it's impossible to reach. => O(1) time

Second, if j < C_i, set the value to dp_table[i - 1][j], which means the current professor will give me too more candies that exceed the number this level should have. => O(1) time

Third, general cases, compute min(dp_table[i - 1][j], dp_table[i - 1][j – C_j] + T_i), which means visit the i_{th} professor or not. However, if both > R_i, set the value to INT_MAX which means it's impossible. => O(1) time

While I'm computing through the table, I check the value of each coordinate, if j > m_max and dp_table[i][j] <= R_i, replace m_max with j. After I find the maximum coordinate, back trace the whole dp_table and find which professor sequence to go.

It take O(N * M) time to build the table as the table has size of N * M and I've to do O(1) operations in each coordinate.

Time Complexity:

It takes O(N * log(N)) + O(N * M) time to implement this algorithm. As M >= N, O(N * log(N)) < O(N * M), it's an O(N * M) algorithm.

Proof of Correctness:

Optimal substructure:

```
dp_table[i][j] = if (j > sum(C_0 : C_i))    => INT_MAX
                    if (j < C_i)             => dp_table[i - 1][j]
                   else                      => min(dp_table[i - 1][j], dp_table[i - 1][j – C_j] + T_i)
```

In the base case, it does have the optimal choice, in every cases, I consider all possible condition and find out that it's a local optimal choice. As a result, I can reach the optimal solution through this algorithm.

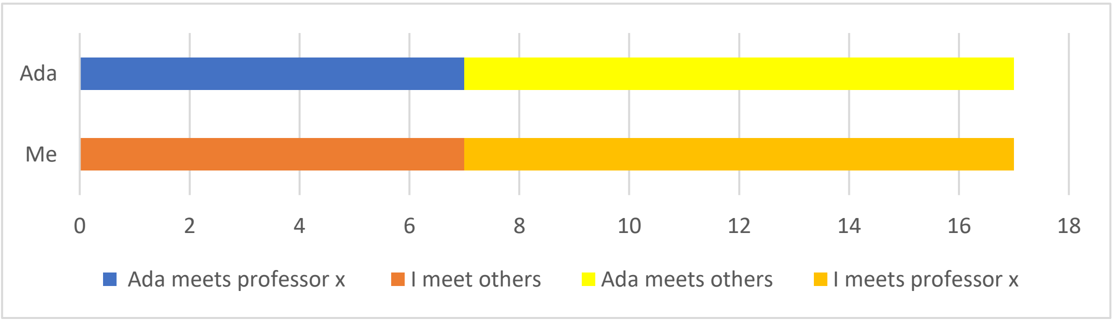
(5)

A = {0, 17, 1, 4, 9}, B = {21, 0, 34, 35, 37} The minimum ending time is 40.
It's trivial that this is the optimal solution, as the sum of f is 40. Ada needs to visit all five professors, which cause he/she needs to spend at least 40 units of time to visit all. Or maybe Ada can try just leave some professor along. :) :)

(6)

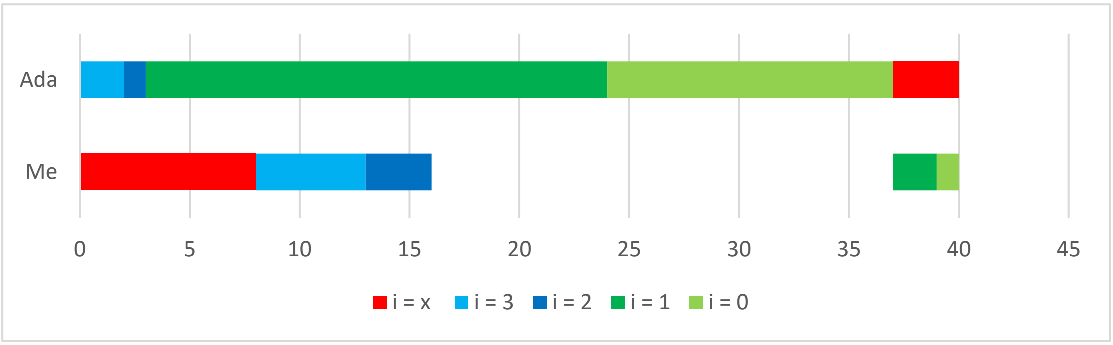
First, I prove $f_x + g_x$ is the minimum ending time and can't be earlier. As both of Ada and me need to visit that professor and we can't go simultaneously, it takes us at least $f_x + g_x$ time to visit that professor.

Second, I prove that we can always find a solution of $f_x + g_x$ ending time. As $f_x + g_x$ max(sum(f_1 : f_n), sum(g_1 : g_n)) >= sum(f_1 : f_n), g_x > sum(f_1 : f_n) – f_x. This states that the time Ada chats with professor x must be sufficient enough for me to chat with all other professors except x. As a result, we can arrange our schedule as I illustrate in the chart below. QED



(7)

As the hint says, pick the max (from 0 to n, min (f_i, g_i)) and set the index of it to x. Put f_x on the most left of the chart and g_x on the most right of the chart, split all other professors into two part. One is those $f_i > g_i$, another one is others. Arrange those $f_i > g_i$ in the left of the chart and the others start from the opposite site of the chart. Let's take (5) for example, we can see that x = 4 and split data into [0, 1] and [2, 3] two set. Put these two set in the left (blue) and right (green), respectively. At the end, push two part together as the chart below.



Time complexity:

For all data I just go through only once and do O(1) operation, therefore, it's an O(N) time complexity algorithm.

Proof of Correctness:

As we choose x from max (from 0 to n, min (f_i, g_i)), f_x and g_x are greater than every min(f_i, g_i) for all other i. Accordingly, overlap can't occurs in any case as there I put a time period with x on either side first. Moreover, it's also the optimal solution as it always takes at least max(sum(f_0 : f_n), sum(g_0 : g_n)) time to finish meeting all professors. Therefore, it's a possible way to find optimal solution.

6. String Problems
(1)
First, compare the length of two string, it takes O(1) time.

If the difference of their length >= 2, break and return false. => O(1) time

If the difference is 1, set two pointers for both string and a dif_count, start from the first char and go through them one by one, compare the pair of chars pointers point at, respectively (S_1[ptr1] and S_2[ptr2]). If they're different, hold the pointer of shorter string and add the dif_count by one, otherwise, add both pointers by 1. If dif_count >= 2 at any time, break the loop and return false, however, if both pointer reach the end of string and dif_count is still 1, return true. => O(n) time.

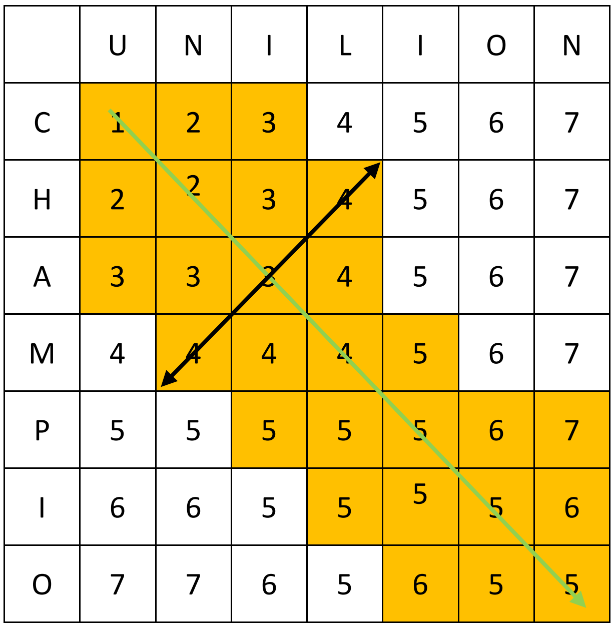
If the difference is 0, set a dif_count, traverse through both string at the same time and compare the pair of char at the same position (e.g. S_1[i] and S_2[i]). If they aren't same, add dif_count by one. If dif_count >= 2 at any time, break the loop and return false, however, if it reach the end of string and dif_count is still 1, return true. => O(n) time.

We can see that it takes O(1) * max(O(n), O(n), O(1)) time which equals to O(n) time.

Prove of correctness:
1. Difference >= 2, trivial, k == 1, it's possible only if 1 >= 2. : (
2. Difference == 1, if dif_count == 1 at the end, then we can just add the one missing char into the shortest string. However, if dif == 2, it means there must be more than one action to do. Then it's false.
3. Difference == 0, if dif_count == 1 || dif_count == 0, just replace the char that is different or do nothing, however, if dif_count >= 2, then we need to do more than one actions. Then it's false.

(3)

I will implement the method of edit distance. However, in order to save time, I'll just implement the diagonal stripe of width 2k - 1 in the n * n array. Take s1 = "UNILION", s2 = "CHAMPIO', and k = 2 for example. I only need to maintain the orange part of the dynamic programming table. To implement this method, I used two pointers, one goes from the upper-left corner to the bottom-right corner, like the green arrow. Another one goes in the vertical direction, like the black arrow. For each coordinate dp_table[i][j] reached, if S_1[i] == S_2[j], its value equals to dp_table[I - 1][j – 1]. Else, check its left, or upper coordinate is in the stripe, if not, just ignore it in the following comparison, its value equals to min(dp_table[i - 1][j], dp_table[i – 1][j – 1], dp_table[i][j - 1]) + 1. In the end, we take the value of dp_table[s1.length() - 1][s2.length() - 1] and compare it with k, if dp_table[s1.length() - 1][s2.length() - 1] <= k, return true, else, return false.



	U	N	I	L	I	O	N
C	1	2	3	4	5	6	7
H	2	2	3	4	5	6	7
A	3	3	3	4	5	6	7
M	4	4	4	4	5	6	7
P	5	5	5	5	5	6	7
I	6	6	5	5	5	5	6
O	7	7	6	5	6	5	5

Time complexity:

This algorithm calculates n rows that not wider than (2k - 1) columns. Moreover, it takes O(1) time calculate each coordinate in the stripe. Therefore, it takes O(n * (2k - 1)) * O(1) = O(n * k) time.

Prove of correctness:

The prove of correctness is same as the one in normal edit distance problem. The only difference is that I ignore some coordinates that | i – j | > k, it obvious that if | i – j | > k, it takes at least k + 1 operations (add or delete) to reach the length, as a result, it won't affect the correct answer though we don't consider it.

(4)

S = “accdefdc”, D(s) = 2 + 0 + 1 + 1 + 1 + 2 + 1 = 8

(6)

Consider merging two string together. Build two N * N dp_tables, the first one saves the minimum cost of all strings that end with char from S_1, the second table save the minimum cost that all strings that end with char from S_2. For every coordinate in both dp_table[i][j], it indicates the cost of combination of S_1.substr(0, i) and S_2.substr(0, j). I go through both dp_table. For each coordinate dp_table$_1$[i][j], assign the value to min(dp_table$_1$[i - 1][j] + Dis(S_1[i], S_1[i -1]), dp_table$_2$[i - 1][j] + Dis(S_1[i], S_2[j])). In other hand, dp_table$_2$[i][j], assign the value to min(dp_table$_2$[i][j - 1] + Dis(S_2[j], S_2[j -1]), dp_table$_2$[i][j - 1] + Dis(S_2[j], S_1[i])), just like the two tables below, set S_1 = “AC”, S_2 = “AD”, the left one is dp_table$_1$ and dp_table$_2$.

In the end, compare the value of right-bottom corner, choose the smaller one and do backtrack with the same method I build the table.

	A	D
A	AA 0	AAD 3
C	ACA 4	AADC 4

	A	D
A	AA 0	ADA 6
C	AAC 2	AACD 3

Time complexity:

Build each table takes N * N time, for each coordinate, it takes O(1) time to assign the value. As a result, it is an 2 * O(n * n) = O(n * n) algorithm.

Proof Correctness:

Optimal Substructure:

The base case is trivial. We can see that dp_table$_1$[i][j] = min(dp_table$_1$[i - 1][j] + Dis(S_1[i], S_1[i -1]), dp_table$_2$[i - 1][j] + Dis(S_1[i], S_2[j])) and dp_table$_2$[i][j] = min(dp_table$_2$[i][j - 1] + Dis(S_2[j], S_2[j -1]), dp_table$_2$[i][j - 1] + Dis(S_2[j], S_1[i])), We can see that either if any dp_table$_1$[i][j] is an optimal solution, dp_table$_1$[i – 1][j] will also be an optimal solution, it’s same as the dp_table$_2$.