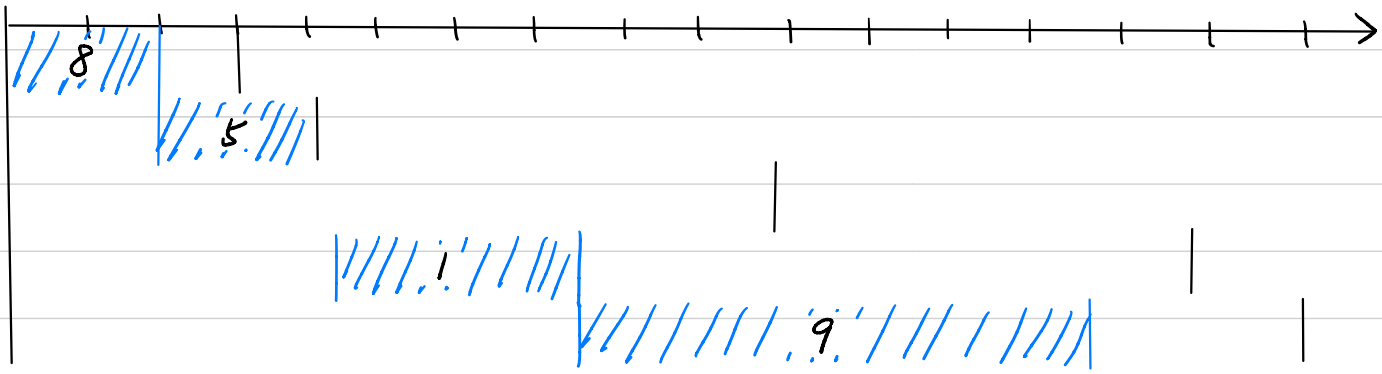
with B08902- 072, 037, 065, 022, ...

stackoverflow, wiki

Problem 5

(1)

The maximum candies I can get is $8 + 5 + 1 + 9 = 23$.



⇒ so select # 1 ~ 4 tasks will reach the max.

(2)

<Algorithm Design>

1° From Assumption 2 & 3 we know, those with smaller i will have a strictly smaller t_i, r_i. So the sequence is already sorted in r, t.

2° So, we will apply greedy algorithm to find optimal solution:

First, choose to talk to this proffesor if t_j + (previous end) $\le r_j$

Second, if yes, make a_j right next to a_{j-1} and add t_j to previous end; also increment aquired candies by 1

Third, repeat until every candidate is considered.

※ For the first candidate, (previous end) = 0 and stick a_0 at the very beginning. and for first k available, previous end = $\sum_{i}^{k} t_i$

< Proof of Correctness >

• Optimal substructure :

1° Suppose S_j is the OPT to choose best from set of $[c_i, r_i, t_i]$ from $[1, j)$, there are 2 case.

- j in OPT, removing j from OPT is optimal solution of S_{j-1} ($S_j = 1 + S_{j-1}$)
- j not in OPT, OPT is an optimal solution of S_{j-1}

• Greedy Choice : Choose the smallest r_i, t_i which $t_i + \sum_{avail\ k} t_k \le r_i$

1° Suppose we have the OPT for S_{i-1}, then for remaining choice i to N, we have to make $\sum_{avail\ k} t_k + t_{new\ choice}$ as small as possible.

2° So if $t_i + \sum_{avail} t_k \le r_i$, we will choice it and the current end time will obtain minimum and $S_i = 1 + S_{i-1}$

< Time Complexity >

1° Since we sequentially considered 1 to N, we use O(N)

2° During every i for $1 \le i \le N$, we consider whether it is legal (t_j + (previous end) $\le r_j$) and modify previous end and aquired candies if needed. All these operations are O(1)

3° So, overall we only need O(N) time. (psuedo-time complexity)

(3)

<Algorithm Design>

1° First, sort the choices (conversation) by r_i increasingly ($r_i \le r_{i+1}$)

2° Maintain a variable total time to record the total time spent to do the conversation. Also maintain a max-heap priority queue pq to record which conversation was made. Also, we need to record numbers of candy obtained.

3° Sequentially check 1 to N and at last, we will get maximum candies.

→ if total time + $t_i \le r_i$, push t_i into the pq and candies + 1 and total time + t_i

→ if total time + $t_i > r_i$: ① $t_i <$ pq.top() ⇒ total + t_i - pq.top() then push t_i into pq and pop pq.top()

↳ then if total time $\le r_1$ ⇒ candies + 1

② $t_i \ge$ pq.top, do nothing.

<Time Complexity>

1° We sort all conversation with ascending r_i takes $O(N\log N)$

2° Then we sequentially check 1 to N. The condition check need $O(1)$ while push and pop into/out of priority queue takes $O(\log N)$. So it overall takes $O(N\log N)$

3° So for sorting and counting, we use $O(2N\log N) = O(N\log N)$ (psuedo)

< Proof of Correctness >

- Optimal Substructure

  1° Suppose S_j is the OPT to choose best from set of $[c_i, r_i, t_i]$ from $[1, j)$, there are 2 case.

  - j in OPT, removing j from OPT is optimal solution of S_{j-1} ($S_j = 1 + S_{j-1}$)
  - j not in OPT, OPT is an optimal solution of S_{j-1}

- Greedy choice: Try to add/replace the smallest t_i into the solution set

  1° Suppose we have OPT for S_j and consider $(j+1)$

  2° if previous total time + $t_{j+1} \leq r_{j+1}$, we can simply add it to the solution set and $S_{j+1} = S_j + 1$

  if not, we can replace the largest perious time with t_{j+1} if t_{j+1} is smaller. This makes ending time earlier and hopefully we can add more to solution set since we have more spared time interval. So $S_{j+1} = S_j$ if new total time $> r_i$, else $S_{j+1} = S_j + 1$. but total time is smaller or equal.

(4) < Algorithm design >

1° Construct a $M+1$ array to record the earliest "total time" to obtain m candies. And initialize $1 \sim M$ candies with infinite and 0 candy with zero. $End = \{0, inf, inf, \cdots, inf\}$

2° Sequentially run through all choices (conversation) and for every choice i, sequentially check c_i to M. Also, maintain a M boolean array to record whether did we get to this numbers of candies by including choice i. (call it $used[M] = \{0, \cdots 0, 0\}$

3° if not $used[j]$ and $End[j - c[i]] + t[i] < r[i]$

update $End[j]$ to $End[j - c[i]] + t[i]$ and set $used[j]$ to 1

else, don't change. (for $j \in [c[i], M]$)

4° After checking all choices (conversation), find the largest non-infinite number and its index will be the maximum amount of candies we can get.

$$\begin{cases} \text{Initialize } End = \{0, inf, inf \cdots inf\} \\ Ans = \arg \max_{1 \le j \le i} End[j] \neq inf \\ End[j] = \min_{c[j] \le i \le M} (End[j], End[i] + t[j]) \text{ if } \begin{cases} used[i] = 0 \\ End[i] + t[j] < r[j] \end{cases} \end{cases}$$

↳ update $used[i] \leftarrow 1$

<Proof of Correctness>

- Optimal substructure :

  1° Suppose S_j is the OPT to choose best from set of $[c_i, r_i, t_i]$ from $[1, j]$, there are $M - c[j]$ cases

  - $c[j]$ candies, if not used[j] and $End[0] + t[j] \le r[j]$
  - $c[j] + 1$ candies, if not used[j] and $End[1] + t[j] \le r[j]$

    ⋮

  - M candies, if not used[j] and $End[M - c[j]] + t[j] \le r[j]$

- Overlapping Subproblem.

  1° End[0] is checked N time, End[1] is checked N-1 time, ..., left is trivial.

- Correctness

  1° for $j = 1$, $End[c[j]] = t_j \Rightarrow c[j]$ is the maximum for $j = 1$.

  2° Suppose $j = k$ this algorithm holds

  3° for $j = k+1$, check if we can get to $c[j+1] \sim M$ candies but sequentially examine all. If $k+1$ is in Solution set S_i, it will make the biggest index not inf, else if it is not in Solution set, it won't get to the largest index and $S_{i+1} = S_i$

  4° So we know, the solution whether equal maximum index get from current or it will be the same as previous Solution set. QED.

< Time Complexity >

1° We sequentially considered 1 to N

2° During every i for $1 \le i \le N$, we sequentially check $c[i]$ to M and update some constant.

3° So, overall we only need $O(MN)$ time. (psuedo-time complexity)

(5) The minimum ending time is $13 + 21 + 1 + 2 + 3 = 40$.

$a = [18, 16, 13, 8, 0]$ $b = [0, 13, 34, 35, 37]$

(6) 1° Without loss of generality, suppose $\Sigma f \ge \Sigma g$

$\Rightarrow f_x + g_x \ge \sum_{i}^{N} f_i \Rightarrow g_x \ge f_1 + f_2 + \cdots + f_{x-1} + f_{x+1} + \cdots + f_N$.

also, $f_x + g_x \ge \sum_{i}^{n} g_i \Rightarrow f_x \ge g_1 + g_2 + \cdots + g_{x-1} + g_{x+1} \cdots + g_N$

2° So we can place the sequence as following and remaining f_i can fit into A with remaining g_i into B.

(f_x, g_x cannot overlap)

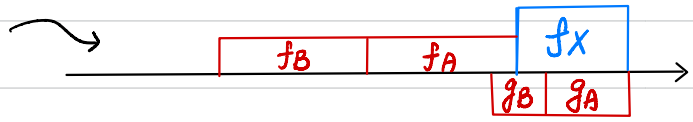
3° From 1°, 2°, the minimum time is $f_x + g_x$. QED

(7) <Algorithm Design>

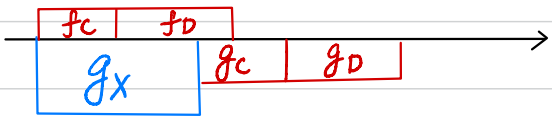
1° If $(f_x + g_x) > \max(\Sigma f_i, \Sigma g_i)$, we can construct a, b as (6), first place f_x, g_x as shown and arbitrary place f_i, g_i into the space.

2° First we devide all pairs (f_i, g_i) into two group with the first group $f_i > g_i$ and second $f_i \leq g_i$. Then also, find the index x which $\min(f_x, g_x) = \max_{i=1}^{N}(\min(f_i, g_i))$ which means the greatest of the smaller one in all pairs.

3° After seperate them into 2 groups and find the index x that makes fulfil the condition, then for those $f_i > g_i$, place block as following:



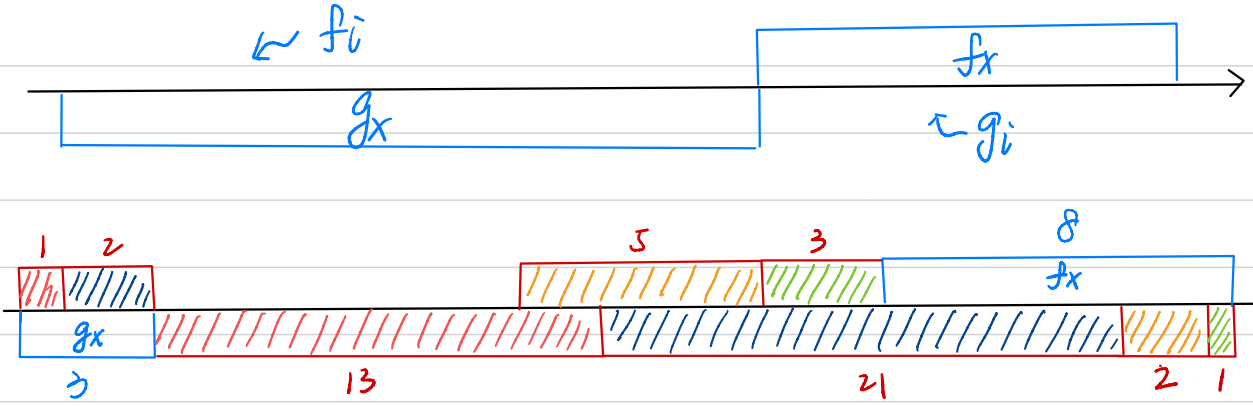
And for those $g_i \geq f_i$, place as following:



4° Implementation is trivial, you can take in arbitrary sequence in the two group and the minimum time is $\max(\Sigma f_i, \Sigma g_i)$. So without lost of generality, suppose $\Sigma g \geq \Sigma f$, construct b and put g_x first and those in group $(g_i > f_i)$ and those in group $(f_i \geq g_i)$.

And for a, we need to maintain the same sequence (ignore f_x, g_x) except there might be gap between two group, so you might want to put those in group ($g_i > f_i$) from time = 0, and place backward from time Σg_i with f_x first and follow those in other group ($f_i \geq g_i$) in reverse order.

5° So we can construct visit plan a and b. And also, minimum time might be $f_x + g_x$ if $(f_x + g_x) > \max(\Sigma f, \Sigma g)$ and $\max(\Sigma f, \Sigma g)$ otherwise.



<Proof of Correctness>

1° For $(f_x + g_x) > \max(\Sigma f, \Sigma g)$, see (6) plz.

2° We can split it into 2 groups and focus on the group $f_i \geq g_i$.

$\because \min(f_x, g_x) = \max_{i=1}^{n}(\min(f_i, g_i))$ $\therefore f_x \geq$ all g_i in that group.

3° So if we place f_x first, we can disjoint f_i and g_i and we won't spend time waiting.

4° Do the same with group $f_i < g_i$, and we stick two components together and let one end stick together (two, maybe). Thus, we can obtain minimum visiting time and the plan a, b.

<Time Complexity>

1° To seperate f_i, g_i into two groups and find f_x, g_x, we spend $O(N)$

2° To place the N groups to correct place, we need $O(1)$ for each operation, and $O(N)$ for all.

3° To sticky two together and output the plan a, plan b. we also need $O(2N) = O(N)$

4° So, overall we need $O(N)$ time.

Problem 6.

(1) < Algorithm Design >

1° First, if $||S_1| - |S_2|| > 1$, then return false.

2° Maintain two pointer ptr1 pointing to the start of s1 and ptr2 points to the start of s2. Also, record how many edit did we make.

3° Run a while loop when both ptr1 and ptr2 isn't at the end of string S_1, S_2. There is mainly two case.

→ if $S_1[ptr1] = S_2[ptr2]$, increment both ptr1 and ptr2 by 1

→ if not equal, increment edit by 1 and return false if edit > 1

then if ptr1 = ptr2, increment both by 1 (replace),

else increment the smaller one (ptr1 or ptr2) by 1 (delete, insert).

4° After we get out of loop, check if there is any character left. If there are any character left (ptr1 != $|S_1|-1$, ptr2 != $|S_2|-1$), simply add the difference to edit and if edit is greater than 1, return false.

< Proof of Correctness >

1° This algorithm apply two pointer technique. And for $s1\ (0 \sim ptr1)$, we can only match $S2\ (0 \sim ptr2)$, $S2\ (0 \sim ptr2-1)$, $S2\ (0 \sim ptr2+1)$. And we can only take one operation. So, we simply take out prefix of S_1 and S_2 (first few match character) and perform replace / insert / delete on that character. After that, the postfix of S_1, S_2 should match.

2°

if $O = \Delta$, then $S_1 = S_2$

if $O \neq \Delta$, replace

O or Δ is empty, insert / delete

< Time Complexity >

1° Since we sequentially run through S_1 and S_2. and increment both pointer if $S_1[ptr_1] = S_2[ptr_2]$ or when we are performing replace and increment shorter string's ptr while performing insert / delete, we use $O(Max(|S_1|, |S_2|)) = O(N)$

2° Also, increment variable within each iteration only takes $O(1)$

3° So, overall we only need $O(N)$ time.

(3)

<Algorithm Design>

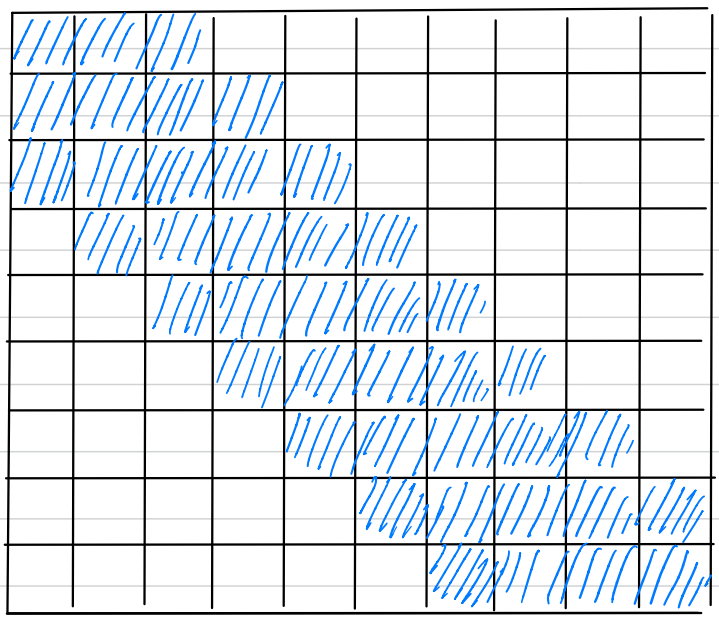
1° Maintain a $(|S_1|+1) \times (|S_2|+1)$ map and initialize $dp[0][i] = dp[i][0]$ with i.

2° Follow the transition function as below to fill the table.

$$dp[i][j] = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ dp[i-1][j-1], & \text{if } S_1[i] = S_2[j] \\ 1 + \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]), & \text{otherwise.} \end{cases}$$

But for every row i, we only need to fill column $(i-k)$ to $(i+k)$ if it is legal. Also when $j = (i-k)$, we are not allowed to come from $dp[i][i-k-1]$ and at $j = (i+k)$, we are not allowed to come from $dp[i-1][i+k]$.

3° So after filling up the table, we simply check whether $dp[|S_1|][|S_2|] \le k$.



(fill the ▨)

ex: k = 2

< Proof of Correctness >

- Optimal Substructure

  1° Suppose $S(i,j)$ is the OPT solution of the minimum cost to convert $S_1[0 \cdots i]$ to $S_2[0 \cdots j]$, there are mainly 2 cases.

  - if $S_1[j] = S_2[i]$, we can make $S(i,j) \leftarrow S(i-1, j-1)$
  - if not equal, we choose the minimum among these
    1. remove $S_1[i]$ from $S_1[0 \cdots i]$, thus $S(i,j) \leftarrow 1 + S(i-1, j)$
    2. insert $S_2[j]$ to $S_1[0 \cdots i-1]$, thus $S(i,j) \leftarrow 1 + S(i, j-1)$
    3. replace $S_1[i]$ with $S_2[j]$, thus $S(i,j) \leftarrow 1 + S(i-1, j-1)$.

  $\therefore S(i,j)$ choose from the minimum of these three.

- Overlapping Subproblem

  1° $S(i,j)$ might be used when at $(i+1, j)$, $(i, j+1)$, $(i+1, j+1)$

- Correctness

  1° for base case $dp[0][i]$, $dp[i][0]$, trivial

  2° Suppose this holds for $dp[i-1][j]$, $dp[i-1][j-1]$, $dp[i][j-1]$

  3° For (i,j), if $S_1[i] = S_2[j]$, then $dp[i][j] = dp[i-1][j-1]$

  else, we can either insert, remove, or replace.

  Thus we choose minimum between them three and add 1 for the cost of operation. So we can obtain best $dp[i][j]$

4° We only need to consider $S_{i,i-k} \sim S_{i,i+k}$ since it will obviously take more if two substring's length is different in more than K. ($S_{i,j}$ will at least take $j-i$ operations).

Thus, we don't need to construct whose column is out of $[i-k, i+k]$ since it's obviously greater than k.

5° From 1° ~ 4° we know, dp[i][j] will be minimum and can compare it with k to check if it's possible.

<Time Complexity>

1° For initialization on the dp map (dp[0][i], dp[i][0]), we spend O(N)

2° We sequentially run through $1 \sim |S_1|$ and during each iteration, and fill up $i-k \sim i+k$. So it take $O(2NK) = O(NK)$ to go through the dp map.

3° Also, for each comparison and updating value, we spend O(1).

4° So overall, we spend O(NK) time. (psuedo time)

(4) $S = accdefdc$, $D(S) = 8$

(6)

<Algorithm Design>

1° Construct a $(|S_1|+1) \times (|S_2|+1) \times 5$ dp map and initialize $dp[0][0][0] = dp[0][0][1]$ with 0.

For $dp[i][j][k]$, we mean that we construct minimum S from $S_1[0 \cdots i]$ and $S_2[0 \cdots j]$ and $k=0$ means we end with $S_1[i]$ while $k=1$ means $S_2[j]$

2° Iterate and follow the transition function. and ignore illegal case.

→ if $S_1[i] != S_2[j]$

$$dp[i][j][0] = \begin{cases} 0, & \text{if } i=j=0 \\ \min \begin{cases} dp[i-1][j][0] + Dis(S_1[i-1], S_1[i]) \\ dp[i-1][j][1] + Dis(S_2[j], S_1[i]) \end{cases} \end{cases}$$

$$dp[i][j][1] = \begin{cases} 0, & \text{if } i=j=0 \\ \min \begin{cases} dp[i][j-1][0] + Dis(S_1[i], S_2[j]) \\ dp[i][j-1][1] + Dis(S_2[j-1], S_2[j]) \end{cases} \end{cases}$$

→ else if $S_1[i] = S_2[j]$

$$dp[i][j][0] = \begin{cases} 0, & \text{if } i=j=0 \\ \min \begin{cases} dp[i-1][j][0] + Dis(S_1[i-1], S_1[i]) \\ dp[i-1][j][1] + Dis(S_2[j], S_1[i]) \\ dp[i][j-1][0] + Dis(S_1[i], S_2[j]) \\ dp[i][j-1][1] + Dis(S_2[j-1], S_2[j]) \end{cases} \end{cases}$$

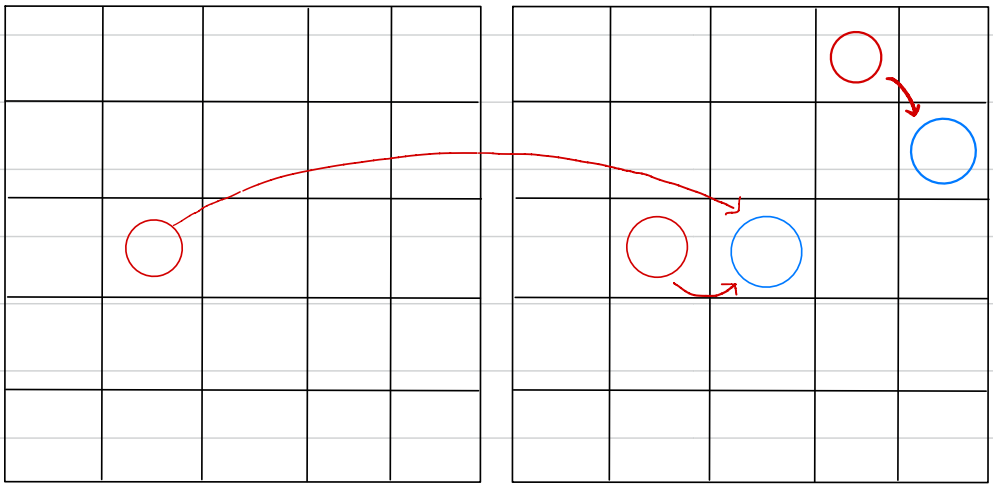
$$\begin{vmatrix} dp[i-1][j-1][0] + Dis(S_1[i], S_2[j]) \\ dp[i-1][j-1][1] + Dis(S_1[i], S_2[j]) \end{vmatrix}$$

And, take out of bound as 0 and those impossible case as ∞.

Also, $dp[i][j][2]$, $dp[i][j][3]$, $dp[i][j][4]$ to point to previous position.

3° After filling the table, minimum $D(S) = \min(dp[|S_1|][|S_2|][0], dp[|S_1|][|S_2|][1])$

And we can backtrack using the $[][][3, 4, 5]$ to acquire S. (trivial)



< Proof of Correctness >

- Overlapping Subproblem

  1° $dp[i][j][k]$ might be used at constructing $S(i+1)(j)$, $S(i)(j+1)$

- Optimal Substructure.

  1° Suppose $dp[i][j][0]$ is the optimal solution ending with $S_1[i]$, then there are two cases, and $dp[i-1][j][0]$, $dp[i-1][j][1]$ and $dp[i-1][j-1][0]$, $dp[i-1][j-1][1]$ is OPT

→ if $S_1[i] \mathrel{!}= S_2[j]$, then

· if S for $(0 \sim i-1)(j)$ end with $S_1[i-1]$,

$D_0 = dp[i-1][j][0] + Dis(S_1[i-1], S_1[i])$

· if S for $(0 \sim i-1)(j)$ end with $S_2[j]$,

$D_1 = dp[i-1][j][1] + Dis(S_2[j], S_1[i])$

$\therefore dp[i][j][0] = \min(D_0, D_1)$

→ if $S_1[i] = S_2[j]$, then

$dp[i][j][0] = \min($ $dp[i-1][j][0] + Dis(S_1[i-1], S_1[i])$,

left ← $dp[i-1][j][1] + Dis(S_2[j], S_1[i])$,

$dp[i][j-1][0] + Dis(S_1[i], S_1[i-1])$,

up ← $dp[i][j-1][1] + Dis(S_2[j-1], S_1[i])$,

$dp[i-1][j-1][0] + Dis(S_1[i-1], S_1[i])$,

upper left ← $dp[i-1][j-1][1] + Dis(S_2[j-1], S_1[i]))$

Thus, $dp[i][j][0]$ is OPT for ending $S_1[i]$ while $dp[i][j][1]$ is OPT for $S_2[j]$.

2° It is same for $dp[i][j][1]$ with S for $(0 \sim i)(0 \sim j)$ with the ending character is $S_2[j]$ (trivial)

3° Thus the optimal solution for this subproblem is minimum of the two ⇒ optimal substructure exists

- Correctness

1° When $i=1, j=1$, base case, trivial

2° Suppose it holds for $dp[i-1][j][0]$, $dp[i-1][j][1]$,

$dp[i][j-1][0]$. $dp[i][j-1][1]$

3° Consider $S(0\sim i)(0\sim j)$ with two cases. First, consider end with $S_1[i]$

① If we end with $S_1[i]$ and previous end with $S_1[i-1]$, we need

$dp[i-1][j][0] + Dis(S_1[i-1], S_1[i])$ to get to this state.

② If we end with $S_1[i]$ and previous end with $S_2[j]$, we need

$dp[i-1][j][1] + Dis(S_2[j], S_1[i])$ to get to this state.

⇒ Thus we choose $\min(①, ②)$ for $dp[i][j][0]$ to construct $S(0\sim i)(0\sim j)$

4° For $S(0\sim i)(0\sim j)$ end with $S_2[j]$, it is the same (trivial).

So we can get minimum $D(S)$ for Σ ending with $S_1[i]$ and $S_2[j]$

And the total minimum $D(S)$ is the smaller one of those two. QED.

< Time Complexity >

1° We sequentially run through $1\sim|S_1|$ and within each

iteration, we run through $1\sim|S_2|$. So we spend $O(|S_1|\times|S_2|)=O(N^2)$

2° For comparison and update of variable, we only need $O(1)$

3° So overall we need $O(N^2)$