Introduction to Algorithms: Second Assignment

Liu Chengjun, Student ID: 0490303961

Problem 1:

For Algorithm A, present general pseudo-code:

Function AlgorithmA($A(n)$)------ $T(n)$

-If $n=1$

-Solve A ------ $T(1)$

-Return solved A

-Else

-Divide A into A_1 till A_5 with size $n/2$

-For i from 1 to 5

-Solve A_i to get S_i ------ $T(n/2)$

-Combine from S_1 to S_5 ------ $O(n)$ (linear time)

-Return solved A

The time of Algorithm A follows: $T(n)=O(n)+5T(n/2)$

By Master Theorem, $O(n)$ is linear, $1<\log_2 5$ as $O(n)=O(n^{(\log_2 5)-\varepsilon})$ where $\varepsilon>0$.

So $T(n)=O(n^{\log_2 5})$ between $O(n^2)$ and $O(n^3)$.

The running time is estimated to be $O(n^3)$.

For Algorithm B, present general pseudo-code:

Function AlgorithmB($B(n)$)------ $T(n)$

-If $n=1$

-Solve B ------ $T(1)$

-Return solved B

-Else

-Divide B into B_1, B_2 with size $n-1$

-Solve B_1 to get S_1 ------ $T(n-1)$

-Solve B_2 to get S_2 ------ $T(n-1)$

-Combine S_1 and S_2 ------ $O(1)$ (constant time)

-Return combined B

The time of Algorithm B follows: $T(n) = O(1) + 2T(n-1)$

So $T(n) = O(1) + 2T(n-1) = 3O(1) + 4T(n-2) = ... = (2^{n-1} - 1)O(1) + 2^{n-1}T(1) = O(2^n)$

So the running time is estimated to be exponential $O(2^n)$.

For Algorithm C, present pseudo-code:

Function AlgorithmC($C(n)$)------$T(n)$

-If $n = 1$

-Solve C ------ $T(1)$

-Return C

-Else

-Divide C into C_1 till C_9 with size $n/3$

-For i from 1 to 9

-Solve C_i subdivision to get S_i subdivision------$T(n-1)$

-Combine from S_1 to S_9 ------ $O(n^2)$ (quadratic time)

-Return solved C

The time of Algorithm C follows: $T(n) = O(n^2) + 9T(n/3)$

By Master Theorem, $O(n^2)$ is quadratic, $2 = \log_3 9$ as $O(n^2) = O(n^{\log_3 9})$ where $\varepsilon > 0$.

So $T(n) = O(n^{\log_3 9} \log n) = O(n^2 \log n)$ between $O(n^2)$ and $O(n^3)$.

The running time is estimated to be $O(n^2 \log n)$.

Problem 2:

Theoretical discussion:

By Greedy Algorithm, the global optimization is reached based on optimization of local circumstances.

For already selected customer sequence $s_1, s_2 ... s_m$ and remained customer sequence $k_1, k_2 ... k_n$, to choose one from remained customer sequence and bind it with already selected customer sequence to optimize new selected customer sequence of size $m+1$, the optimized solution is to choose customer of minimum time from remained customer sequence, which follows the Greedy Algorithm.

So in global environment, it actually performs the same as sorting method of selection sorting by choosing the minimum, place it one by one and record the corresponding customer number.

That means by using fast sorting method and moving customer numbers the same as movements of sorted time, the algorithm can reach optimization.

Pseudo-code:

Given two vectors of customer numbers and spending time corresponding to each other: $C(n)$ and $S(n)$

Function MinWaitingTime($C(n), S(n)$)

-If $n <= 1$

- Return $C(n)$ %Only one in C(n)

-Else

- $x = S[n]$ %seeking reference at the end of array S(n)

- $i = 0$ %counting from in, tracking the final place that reference x should be

-For j from 1 to $n-1$

-If $S[j] < x$ %if finding the spending time smaller than x

- $i = i+1$

-Swap $S[i]$ with $S[j]$ %swap it to the left

- Swap $C[i]$ with $C[j]$ %moving the corresponding index together

-Swap $S[i+1]$ with $S[n]$ %insert the reference

-Swap $C[i+1]$ with $C[n]$ %insert the index of reference together

- MinWaitingTime($C[1,i], S[1,i]$) %split and sort lower half

- MinWaitingTime($C[i+1,n], S[i+1,n]$) %sort upper half

Return $C(n)$

The pseudo-code above simulates Quick Sort by binding the movement of customer number and spending time. Because the movements of elements ($C(n), S(n)$) are doubled compared with Quick Sort only moving $S(n)$, the running time is estimated to be $2 \times O(n \log n) = O(n \log n)$. ($T(n) = T(n-q) + T(q) + (n-1) \times 3+2$)

The average waiting time of customers will be the minimum of $\sum_{i=1}^{n} (n-i)t_i$, where sequence $t_1 \sim t_n$ should be in ascending order. The reason why it is minimum is given by recursion from dual inequality $1 \times t_{max} + 2 \times t_{min} < 1 \times t_{min} + 2 \times t_{max}$, further proving it applies to multiple elements.

A more basic way simulating finding the minimum one by one like Greedy Algorithm is as follow:

Function MinWaitingTime($C(n), S(n)$)

Order=Array(n)

RemainSn= $S(n)$

RemainCn=C(n)

```
for i from 1 to n:
    Order[i]=RemainSn(FindMinimum(RemainSn))    %finding the index of minimum in the remained array
    RemainSn=RemainSn(1:Order[i]-1)+RemainSn(Order[i]+1:n) %keep remaining array
    RemainCn=RemainCn(1:Order[i]-1)+RemainCn(Order[i]+1:n) %keep remaining array of index
Return Order


Function j=FindMinimum(RemainSn)
j=1
Minimum=RemainSn(1) %starting point to compare
For i from 2 to n:
    if RemainSn(i)<Minimum %if smaller than reference
        Minimum=RemainSn(i) %make the compared the minimum
        j=I %record the index of minimum
Return j %return the index of minimum
```

Obviously the running time of the algorithm is 1+2+…+n=$O(n^2)$

Problem 3:

The Algorithm for finding consecutive LCS is similar to finding LCS.

By running all combinations of comparisons of substrings the same as finding LCS($m \times n$ situations, that is $O(mn)$), the algorithm can be completed.

To find the length of LCS in every sub-situation, the length should follow such recursive rule:

$$C_{i,j}\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ C_{i-1,j-1}+1 & \text{if } i,j>0\ ,\ X_i=Y_j \text{ and } X_{i-1}=Y_{j-1} \\ \max\{C_{i,j-1},C_{i-1,j}\} & \text{else} \end{cases}$$

A new rule is that we regard each combination (i, j) as a vector of two elements $C_{i,j}^{\max}$ and $C_{i,j}$. Recursive rule changes the $C_{i,j}$element of a vector and inherit $C_{i,j}^{\max}$ element. When $X_i = Y_j$ but $X_{i-1} \neq Y_{j-1}$, reset $C_{i,j}$ to be 1 in the vector (i, j) . When$C_{i,j} > C_{i,j}^{\max}$, let $C_{i,j}^{\max}$=$C_{i,j}$.

For example, applying the sequences given in assignment:

	A	G	C	T	A
A	1,1	1,1	1,1	1,1	1,1
C	1,1	1,1	1,1	1,1	1,1
T	1,1	1,1	1,1	2,2	2,2

A	1,1	1,1	1,1	2,2	3,3
T	1,1	1,1	1,1	2,2	3,3
G	1,1	1,1	1,1	2,2	3,3

The pseudo-code of algorithm is presented as follow:

Function ConsecutiveLCS($X(m), Y(n)$) %two string array of X of m size and Y of n size

$C = Array(m+1, n+1)$, with all elements to be 0 %recording the LCS still counting the length

$D = Array(m+1, n+1)$, with all elements to be 0 %recording the maximum LCS up to now

For i from 2 to $m+1$

For j from 2 to $n+1$

If $X_i = Y_j$ and $X_{i-1} = Y_{j-1}$ %if X=Y and consecutive with the extending LCS in the front

$C(i, j) = C(i-1, j-1)+1$ %adding the counting length of maximum

$D(i, j) = D(i-1, j-1)$ %copying the maximum from i-1, j-1

If $D(i, j) < C(i, j)$ %if the counting LCS exceeds the maximum record

$D(i, j) = C(i, j)$ %renew the maximum record

Else if $X_i = Y_j$ and $X_{i-1} \neq Y_{j-1}$ %if X=Y but disconnected to previous LCS

$C(i, j) = 1$ %restart counting at 1

$D(i, j) = 1$ %usually as 1 when starting to count first LCS

If $D(i-1, j) > D(i, j-1)$ & $D(i-1, j) > 1$ %if the maximum on i-1,j is larger

$D(i, j) = D(i-1, j)$ %copy the record

Else if $D(i, j-1) \leq D(i-1, j)$ & $D(i, j-1) > 1$ %if the maximum on I,j-1 is larger

$D(i, j) = D(i, j-1)$ %copy the record

Else if $X_{i-1} \neq Y_{j-1}$ and $D(i-1, j) > D(i, j-1)$ %X unequal to Y, the counting breaks.

$C(i, j) = 0$ %restart counting

$D(i, j) = D(i-1, j)$ %copying maximum record

Else

$C(i, j) = 0$ %restart counting

$D(i, j) = D(i, j-1)$ %copying maximum record

Return $C(m+1, n+1)$

The running time is roughly $T(\text{if loop}) \times m \times n = O(mn)$, fitting the requirement.