演算法 Homework#2 20191017
山下夏輝（Yamashita Natsuki）
R08922160
Reference: 陳建宏（d08944003）, TA

Problem5

(1)

(2,13), (5,5), (3,4), (7,3), (4,2)
The minimum time: 23

(2)

Pseudo code

```
Min_T(customer[], N)
      Sort(customer[])          // sort by e
      dp[0] = 0
      For i to N
         dp[i] = dp[i-1] + p[i]
      For i to N
            If e[i] < dp[N] – dp[i]+ e[N]
                  case_1 = dp[N] + e[N]
            Else
                  temp = dp[N] + e[i] – (dp[N] – dp[i]+ e[N])
                  case_2 = Max(temp, dp[N] + e[i] – (dp[N] – dp[i]+ e[N]))
      return Max(case_1, case_2)
```

The minimum time: 23

The short- time first. T() is the minimum time that customers take from when coming into the shop until leaving. If none of e is bigger than $\Sigma_{1 \text{ to } n}(p) + (e_n)$, it its case 1. If not, it is case 2.

Case 1

$T = \Sigma_{1ton}(p) + (e_n)$

Case 2

$T = \Sigma_{1ton} (p) + (e_i) – \{\Sigma_{iton}(p) + (e_n)\}$

Also, the time complexity of Sort function is O(nlogn) and the time complexity of for loops are O(n) each.

(3)

Goal:

dp[N] + a
//dp[N] is the sum of the value of all p
//a is the minimam time of the after finishing all preparation

Optimal Substructure:

Suppose OPT is an Optimal solution dp[N] + a after sorting all elements by e, there 2 cases.If none of e_i is bigger than $\Sigma_{i \text{ to } n}(p) + (e_n)$, it its pattern 1. If not, it is pattern 2.

Case 1

a is $e_{n.}$

Case 2

a is $(e_i) - \{\Sigma_{iton}(p) + (e_n)\}$.

Also dp[i] is in dp[N] when dp[N] + a is OPT.

Greedy Choice

The larger e first.

Correctness:

The reason why the greedy algorithm is not worse than OPT' is below.

In case 1

OPT': After sorting all elements by e, you substitute e_n to e_i. $(0<i<N)$

When $e_n = e_i$,

OPT = OPT'

When $e_n < e_i$,

OPT < OPT'

In case 2

// e_i is bigger than $\Sigma_{i \text{ to } n}(p) + (e_n)$

OPT': After sorting all elements by e, you substitute e_i to e_j. $(i<j)$

OPT< OPT'

Because of $(e_i) - \{\Sigma_{\mathbf{j} \text{ to } n}(p) + (e_n)\} > (e_j) - \{\Sigma_{\mathbf{i} \text{ to } n}(p) + (e_n)\}$

It is because of $\{\Sigma_{\mathbf{i} \text{ to } n}(p) + (e_n)\} > \{\Sigma_{\mathbf{j} \text{ to } n}(p) + (e_n)\}$

So the correctness of greedy algorithm holds.

(4)

This algorithm perform the best only when each of piepie00 and piepie01 is alternatively assigned the pies which is ordered by e. For example, Piepie00 makes odd number of pies, and piepie01 does even number of pies.

The reasons are below.

In case 1

1. e_n and e_{n-1} is the minimum and 2nd minimum value each.
   -> each of a in dp[N] + a is minimum
2. The nearest value of dp[N]/2 is the minimum value in each parsons when you assign pies two people so that assignment of pie to two people should be alternatively done.
   -> each of dp[N] in dp[N] + a is minimum

So, each of dp[N] + a is minimum.

In case 2

Because of the condition of 2 of in case,

OPT': After sorting all elements by e, you substitute e_i to e_j. $(i<j)$

OPT< OPT'

Because of $(e_i) - \{\Sigma_{\mathbf{j} \text{ to } n}(p) + (e_n)\} > (e_j) - \{\Sigma_{\mathbf{i} \text{ to } n}(p) + (e_n)\}$

It is because of $\{\Sigma_{\mathbf{i} \text{ to } n}(p) + (e_n)\} > \{\Sigma_{\mathbf{j} \text{ to } n}(p) + (e_n)\}$

So, each of dp[N] + a is minimum.

(5)

Pseudo code

```
Kill()
    P = Max_p(customer[])
    Return p
Min_T(customer[], N)
    Sort(customer[], sorted[])        // sort by e
    dp[0] = 0
    For i to N
        dp[i] = dp[i-1] + p[i]
    For i to N
        If e[i] < dp[N] – dp[i]+ e[N]
            case_1 = dp[N] + e[N]
        Else
            temp = dp[N] + e[i] – (dp[N] – dp[i]+ e[N])
            case_2 = Max(temp, dp[N] + e[i] – (dp[N] – dp[i]+ e[N]))
            If temp != Max(temp, dp[N] + e[i] – (dp[N] – dp[i]+ e[N]))
                influence_answer = e[i] – (dp[N] – dp[i]+ e[N])
                influence_answer_index = i
    If case_1 == Max(case_1, case_2)
        Return case_1 – Max(Kill(), e[N])
    Else
        Return case_2 – Max(Kill(), influence_answer + p[influence_answer_index])
```

The biggest value of time which influence the total time in optimal should be "kill"ed. The optimal answer is indicated by below(task2).

Case 1

$T = \Sigma_{1ton}(p) + (e_n)$

Case 2

$T = \Sigma_{1ton}\,(p) + (e_i) - \{\Sigma_{iton}(p) + (e_n)\}$

Then according to the definition of cases, in case 1 the biggest value of time which influence the total time in optimal is the biggest p or e[N]. So the bigger one should subtracted from the total time after Kill() function finds the biggest p.

Same as case 1, in case 2 the biggest value of time which influence the total time in optimal is the biggest p or $(p_{i)}) + (e_i) - \{\Sigma_{iton}(p) + (e_n)\}$. So the bigger one should subtracted from the total time after Kill() function finds the biggest p.

Also, the time complexity of Kill() is O(nlogn) so the entire algorithm runs in O(nlogn).

Problem6

(1)

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
class	o						o				o	o					o
diner						5									4		

The number of diner: 2

(2)

We start from x[i] and d[j] and iterating them. Pointer memorizes the position of the most right side value which d[j] is covering. Once pointer is updated, 1 is added to counter, which means that one more diner is needed for covering the class. Also one x[i] (i < N) is smaller than d[j], we do not need to consider it. In every for loop, we check which x[i] has not been smaller than d[j].

The time complexity is O(N).

Pseudo code

```
Min_num()
    i = 1
    j = 1
    Pointer = 0
    cnt = 0
    For i to N
        If pointer < x[i]
            Pointer = x[i] + (d[j]*2)
            j ++
            cnt ++
    Return cnt
```

(3)

(4)

Problem7

[Task2]

We think there are 2 route, one is up route which is from the lowest point to the highest point and down route(which is from the highest point to the start point).We try to find minimum time route which is satisfied that every points is in one of 2 two route.

We assume that if the route with Y[N+1] is in up route, the route with Y[N+2] is in down route. Then the answer we need to get is below with Dynamic Programing. dp[[up,down]] indicates the minimum distance of the up -> every points under up or down -> down.

Min(dp[[N+2, N+1]] + Distance(N+1, N+2) , dp[[N+1, N+2]] + Distance(N+1, N+2)

In recursive case, we find which point is connected to N+2 when it would be the minimum path. The recursive case is below and also the picture for your understanding is attached.

dp[[k,k-1]] + g[[k,i]] – g[[0,k-1]] + Distance(k-1, i+1)

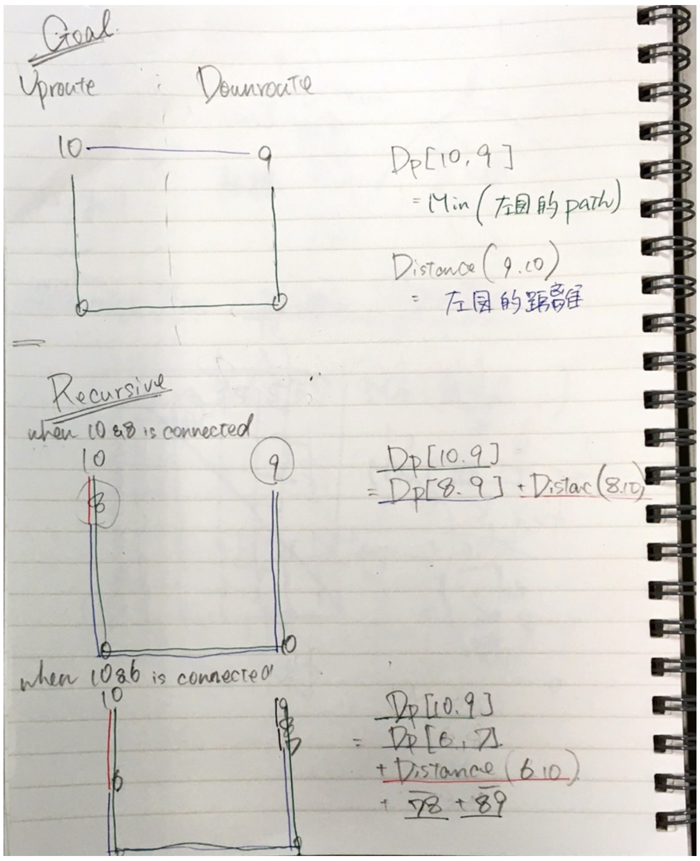
DEFINITION

i = N+1

0 < k ≤ i

g[[Y[s],Y[t]]] = the sum of the distance through every points from Y[s] to Y[t]

In addition to it, this is O(N^2) time complexity and O(N) space complexty.



Pseudo code

```
Min_time(A[])
    Sort(A[], Y[])                // order by the value of y from lower to higher and set in Y[]
    Distance(d,e)        // calculate the distance of 2 points. d,e is the index of Y[]
    Preprocessing(Y[])           // calculate the sum of the distance through every points from Y[0]
                                 to Y[i]  and set in g[]
```

```
         For i to N+2
              g[[0,i]] = g[[0,i-1]] + Distance(i-1, i)

dp[[0,1]] = Distance(0,1)       // base case
dp[[1,0]] = Distance(1,0)       // base case

temp = infinity max
for( i = 1;  i < N+1; i++)
         for ( k = 1 ; k <= i ; k ++)         //Y[N+1]( 2nd highest point in Y) is in th left side route
              temp = min(temp, dp[[k,k-1]] + g[[k,i]] – g[[0,k-1]] + Distance(k-1, i+1))
          dp[[i,i+1]] = temp

         for ( k = 1 ; k <= i; k ++)          //Y[N+1]( 2nd highest point in Y) is in the right side route
              temp = min(temp, dp[[k-1,k]] + g[[k,i]] – g[[0,k-1]]  + Distance(k-1, i+1))
         dp[i+1][i] =temp

return Min(dp[[N+2, N+1]] + Distance(N+1, N+2) , dp[[N+1, N+2]] + Distance(N+1, N+2)
```

[Task 4]

In addition to Task2, we need to consider the color. the inside of the recursive cases which are in “m++” for loop restrict the condition which Collee cases need to satisfy.

Same as Task2, this is $O(N^2)$ time complexity because the time complexity of “m++” for loop is $O(2^7)$. The space complexity is O(N)

Pseudo code

```
Min_time(A[])
   // order by the value of y from lower to higher and set in Y[]
   Sort(A[], Y[])
   // calculate the distance of 2 points. d,e is the index of Y[]
   Distance(d,e)
   // calculate the sum of the distance through every points from Y[0] to Y[i]  and set in g[]
   Preprocessing(Y[])
         For i to N+2
             g[[0,i]] = g[[0,i-1]] + Distance(i-1, i)

   dp[[0,1, SET={ R,O,Y,G,B,I,Y }]] = Distance(0,1)      // base case
   dp[[1,0, SET={ R,O,Y,G,B,I,Y }]] = Distance(1,0)      // base case
   for( i = 1;  i < N+2; i++)
         temp = infinity max
         for ( k = 1 ; k <= i ; k++)         //Y[N+1]( 2nd highest point in Y) is in the left side path
              for(m = k, m <= i, m++)                //delete $c_k$ to $c_i$ from the left side route
                   dp[[k,k-1,SET]] =dp[[k,k-1,SET-$c_m$]]
              temp =  min(temp, dp[[k,k-1,SET]] + g[[k,i]] – g[[0,k-1]] + Distance(k-1, i+1)
         dp[[i,i+1,SET]] = temp

         temp = infinity max
         for ( k = 1 ; k <= i; k ++)         //Y[N+1]( 2nd highest point in Y) is in the right side route
              for(m = k, m <= i, m++))               //delete $c_k$ to $c_i$ from the right side route
                   dp[[k-1,k,SET]] =dp[[k-1,k,SET- $c_m$]]
              temp = min(temp, dp[[k-1,k,SET]] + g[[k,i]] – g[[0,k-1]] + Distance(k-1, i+1))
         dp[i+1,i,SET] =temp
   return Min(dp[[N+2, N+1,SET]] + Distance(N+1, N+2) , dp[[N+1, N+2,SET]] + Distance(N+1,
N+2)
```