ADA HW2

B07902126 資工二 謝宗儒

PB5

(1) Arrangement : (2, 13), (5, 5), (3, 4), (7, 3), (4, 2)

minimum time needed : 23 minutes

(2) merge-sort the pair-sequence with the "e" descending

```
Int time=0, eat_up=0, sorted-pair-sequence(p, e)[N]
for(i=1; i<=N; i++){
    time += pi
    eat_up = max(eat_up,   time + ei)
}
total time needed =eat_up.
```

merge-sort cost O(N lg N) and the for loop cost O(N), so this algorithm cost O(N lg N) in total to find total time needed.

(3)This algorithm use greedy skill

Greedy choice : first do the task needed most time to eat

Assume there is no OPT including this greedy choice

Since the sum of the cost "p" is not related and most less than the total time needed, and also all "e" are independent to each other. The total time needed is actually influenced by the arrangement of "e".

If OPT process ai as the i-th task to do, we can switch ak that k > i and ai and make that become OPT' (greedy choice). Since, ek > ei,

Case1 : if k is not the last one finish in OPT and k is not the last one finish in OPT'

(i is either not the last one in both OPT and OPT')

total time needed(OPT') = total time needed(OPT)

➔ OPT' is as good as OPT, so OPT' is an optimal solution containing greedy choice.

Case2 : if k is the last one finish in OPT and k is the last one finish in OPT'

Since OPT' do ak earlier than OPT

total time needed(OPT') <= total time needed(OPT)

➔ OPT' is better than OPT, the property is proved by contradiction

Case3 : if k is the last one finish in OPT and k is not the last one finish in OPT'

Total time needed(OPT') <= total time needed(OPT)

➔ OPT' is better than OPT, the property is proved by contradiction

With the 3 cases, we prove that OPT' is equal or better than OPT, so we prove this problem has greedy-choice property and also prove the correctness of this algorithm.

(4)No, take (2, 12), (3, 11), (15, 1) for example, if use the method in (2), Piepie00 will do (2, 12) while Piepie01 do (3,11) at first, and then piepie00 will do (15, 1). Total time needed = 18 minutes.
However, if Piepie00 do (2, 12) and (3, 11) while Piepie01 do (15, 1).
Total time needed = 16 minutes.

⇨ Using the method in (2) won't perform the best in this situation.

(5) merge-sort the pair-sequence with the "e" descending (cost O(N lg N))

```
Int time=0, eat_up=0, maxP=0,maxP_idx, end[2][N],sorted-pair-sequence(p, e)[N]
Int kill_P_time , kill_E_time, target
for(i=1; i<=N; i++){
    if(maxP < pi)
        maxP = max(maxP, pi)
        maxP_idx = i
    time += pi
    end[0][i]=i      //keep the index
    end[1][i]= time + ei      //keep end time of each client
    if(eat_up < time + ei)
        eat_up = time + ei
}    //(cost O(N))
merge-sort end[1][] with end time of each client and also switch end[0][] to match
find the index of the one last eat up "EL_idx" and the second latest end time
"sEL_time".    //(cost O(N lg N))
if(maxP_idx == EL_idx)        //the one with largest p is the latest one
    target = EL_idx;
else{
    kill_P_time = eat_up - maxP        //if kill the one with largest P
    kill_E_time = sEL_time – p[EL]        //if kill the one last eat up
    if(kill_P_time < kill_E_time)
        target = maxP_idx
    else
        target = EL_idx
} //(cost O(1))
target is the one to kill.    (total cost = O(N lg N)+ O(N)+O(N lg N)+ O(1) = O(N lg N))
```

This algorithm also first use greedy skill to decide the order
Greedy choice : first do the task needed most time to eat

With the 3 cases mentioned in (3), we prove that OPT’ is equal or better than OPT, so we prove this problem has greedy-choice property.
To minimize the time needed, Piepie must kill the one who influences the time needed the most. There are 3 possibilities :
Case 1 : kill the one needed the most time to prepare.

Prepare time will influence when the rest client start to eat. The last one eat up might just start to eat late, so kill the one needed the most time to prepare might decrease the time needed more.

Kill any other with pi < pmax will save pi minutes which is less than pmax, so it won’t be better than kill the one with pmax (temporarily ignore collision of case1 and case2).

Case 2 : kill the last one eat up

Kill the last one eat up will obviously decrease the time needed. Kill the last one eat up will save (plast + elast - e2last) minutes, but kill any other not the last will only save 0 or pi minutes, (temporarily ignore collision of case1 and case2).

For all save 0 minutes situation or pi < (plast + elast - e2last), kill the last one eat up is better.

(all pi > (plast + elast - e2last) are collisions of case1 and case2)

Case 3 : kill the one needed the most time to prepare and also the last one eat up.

Obviously, with the two cases above, kill this one will minimize the time needed.

If case 3 exist, kill the one.
If case 3 doesn’t exist, to compare killing the one in case 1 and the one in case 2 which is better (deal with collision of case1 and case2) will find the best target to kill.
Therefore, we prove the correctness.

PB6

(1)One of the best plan is to put the mobile diner with d=5 at x=12 and put the diner with d=3 at x=1.

(2)

```
Given x[1 to N] and M mobile diner with d*
Int range = -1, count=0;
for(i=1; i<=N; i++){
    if(x[i] > range ){          //put a mobile diner at x=x[i]+(d*)
        count++
        range = x[i]+2*(d*)
    }
```

```
}
```

This algorithm cost O(N) belong to O(N+M)

If I want to use the least mobile diners, diners' range should better not overlap. By linear searching from the first class, whenever meet a class is not covered, put an available mobile diner and cover the class with left end of its range and update the cover range. By doing so, we can make sure that we get the most out of the diner because it extends as much as possible to the right (can't cover more classes) while cover the class we meet with its left end of range.

(3)Given x[1 to N] and d[1 to M]

```
Int range = -1, count=0;
for(i=1; i<=N; i++){
        if(x[i] > range ){                //put a mobile diner at x=x[i]+d[i]
                count++
                range = x[i]+2*d[i]
        }
}
```

This algorithm cost O(N) belong to O(N+M)

If I want to use the least mobile diners, diners' range should better not overlap.

Greedy choice : whenever put a diner, cover a class with left end of its range.

Therefore, whenever put a mobile diner, we can regard the rest not covered part and other available diner as its subproblem. Prove it has optimal substructure.

By linear searching from the first class, whenever meet a class is not covered, greedy put an available mobile diner with smallest index and update the cover range. By this greedy choice, we can make sure that we get the most out of the diner because it extends as much as possible to the right (can't cover more classes) while cover the class we meet with its left end of range.

If ai of OPT is not a greedy choice ak, we can use ak to replace ai and make OPT become OPT'. Since the algorithm starts from the left, whenever meet a class "C" not cover, the classes on the left of C must have been dealt with. Therefore, the range on the left of C which can extend to the right would be wasted. It's obviously that ak will cover equal or more class not covered than ai. Therefore, OPT' is better than OPT. Prove it has greedy property.

PB7.

(1)

Since time cost of pa->pb = pb->pa, we can see part2 as another road from p0 to pN. Let DP[i][j] = the minimum time cost of part 1 with p0 to pi and part2 (inverse) with p0 to pj. If the next point to decide to put part1 or part2 is pk, we can update DP by comparing which decision will lead to less time cost.

```
DP[N+1][N+1]
//initialize (cost O (N^2))
for i=0 to N
        for j= 0 to N
                DP[i][j]=inf
DP[0][0]=0

//DP    (cost O(N^2))
i=1, j=0
for i= 1 to N-2
        for j=0 to i-1
                k=i+1
                DP[i][k] = min(DP[i][j]+ f(j, k), DP[i][k])
                DP[k][j] = min(DP[i][j]+ f(i, k), DP[k][j])

//find answer from DP (cost O(1))
Minimum_time = inf
Minimum_time = min(DP[N-1][j] + f(i, N), f(j, N) ,    DP[i][N-1] + f(i, N), f(j, N) )
return Minimum_time
```

This algorithm will totally cost time with O(N^2).

(2)

Since time cost of pa->pb = pb->pa, we can see part2 as another road from p0 to pN. Let DP[i][j] = the minimum time cost of part 1 with p0 to pi and part2 (inverse) with p0 to pj. If the next point to decide to put part1 or part2 is pk, we can update DP by comparing which decision will lead to less time cost.

```
DP[2][N+1]
//initialize    (cost O (N))
for i=0 to N{
        DP[0][i]=inf
        DP[1][i]=inf
}
DP[0][0]=0, T=0
//DP     (cost O (N^2))
for i = 1 to N-1{
        T++
        for j = 0 to i-1{
                DP[T%2][ j ] = min(DP[(T-1)%2][ j ] + f(i, i-1), DP[T%2][ j ] )
                DP[T%2][ i- 1 ] min(DP[(T-1)%2][ j ] + f(i, j), DP[T%2][ i-1 ] )
        }
}
//find answer from DP      (cost O(N))
Minimum_time = DP[T%2][ 0 ] + f(N-1, N) + f(0, N)
for j = 1 to N-1
        Minimum_time = min(Minimum_time, DP[T%2][ j ] + f(N-1, N) + f(j, N))

return Minimum_time
```

This algorithm will totally cost time with O(N^2) and space with O(N).