# 2019 ADA HW 2

b07902064 資工二 蔡銘軒

November 12, 2019

## Problem 5

(1) The optimal arrangement is [(2, 13), (5, 5), (3, 4), (7, 3), (4, 2)]. The minimum time needed is 23 minutes, which is the time the 5-th customer finishes his meal.

(2)

```
minimum_time():
   Read input as (p, e) pair into customer[]
   Sort all pairs such that e is non increasing.
   min_time = 0
   prefix_sum = 0
   for i from 0 to n - 1:
      prefix_sum += customer[i].p
      min_time = max(min_time, prefix_sum + customer[i].e)
   return min_time
```

In the above snippet, the complexity is:
$O(N)$ for reading input
$O(NlgN)$ for sorting the array
$O(N)$ for finding the longest time needed
So the overall time complexity is $O(N) + O(NlgN) + O(N) = O(NlgN)$

(3) The algorithm adopts a greedy approach. We'll show the greedy choice property and optimal substructure to prove the correctness of the algorithm.
**Optimal substructure** Let $I(m, (p, e))$ be the optimal answer to problem with $m$ people, and $(p, e)$ is the information of the $m$ people. Our goal is to find $I(n, (p, e))$, which can be found by:

$$I(n, (p, e)) = \min_{i=1,2,\ldots,n} (\max(p_i + e_i, p_i + I(n-1, (p, e))))$$

If not, there exists some $j$ and $I'(n-1, (p, e)) \neq I(n-1, (p, e))$ such that $\max(p_j+e_j, p_j+I'(n-1, (p, e)))$ is smaller than $\max(p_i + e_i, p_i + I(n-1, (p, e)))$, for $i = 1, 2, \ldots, n$. This indicates $I'(n-1, (p, e)) < I(n-1, (p, e))$, which contradicts the fact that $I(n-1, (p, e))$ is an optimal answer to the problem.
**Greedy choice property** We argue that, the more time a person needs to finish the food, the earlier he should be served.
Consider an arrangement $\mathcal{A} = (p_1, e_1), \ldots, (p_i, e_i), (p_{i+1}, e_{i+1}), \ldots, (p_n, e_n)$. If there's a pair such that $e_i < e_{i+1}$, we swap their positions and denote the new arrangement as $\mathcal{A}'$. We introduce the notation $\mathcal{P}$ and define $\mathcal{P}_i = p_1 + p_2 + \cdots + p_i$. Then the time the *i-th* person finish his meal can be expressed as $\mathcal{P}_i + e_i$.
The answer to $\mathcal{A}$ is $\max(I(i-1, (p, e)), \mathcal{P}_{i-1} + p_i + e_i, \mathcal{P}_{i-1} + p_i + p_{i+1} + e_{i+1}, \mathcal{P}_{i-1} + p_i + p_{i+1} + I(n - i + 1, (p, e)))$
The answer to $\mathcal{A}'$ is $\max(I(i-1, (p, e)), \mathcal{P}_{i-1} + p_{i+1} + e_{i+1}, \mathcal{P}_{i-1} + p_{i+1} + p_i + e_i, \mathcal{P}_{i-1} + p_i + p_{i+1} + I(n - i + 1, (p, e)))$
The first and the last terms in the two equations are equal, and since $e_i < e_{i+1}$, we have $\mathcal{P}_{i-1} + p_i +$

$p_{i+1} + e_{i+1} > \mathcal{P}_{i-1} + p_{i+1} + e_{i+1}$, and $\mathcal{P}_{i-1} + p_i + p_{i+1} + e_{i+1} > \mathcal{P}_{i-1} + p_{i+1} + p_i + e_i$. This shows that $\mathcal{A}'$ leads to an answer that is at least as good or better than that of $\mathcal{A}$.

(4) No, the property in (2) no longer holds. Consider the case: [(100, 1), (1, 2), (1, 3)]
The optimal solution is the following arrangement, which gives the answer 101:

- The two colors represent the two chefs.
- In case of a hard copy, different chefs are also represented by different fonts.

**[(100, 1)**, *(1, 3)*, *(1, 2)*]

However, adopting the idea in (2) gives 102, which is not optimal:

**[(1, 3)**, *(1, 2)*, **(100, 1)]**

(5)

```
minimum_time():
   Read input as (p, e) pair into customer[], starting at index 1 for convenience
   Sort all pairs such that e is non increasing.
   Create finish_time[], left_max[] and right_max[]
   prefix_sum = 0
   for i from 1 to n:
      prefix_sum += customer[i].p
      finish_time[i] = prefix_sum + customer[i].e
   left_max[0] = right_max[n + 1] = -INF
   for i from 1 to n:
      left_max[i] = max(left_max[i - 1], finish_time[i])
   for i from to 1:
      right_max[i] = max(right_max[i + 1], finish_time[i])
   answer = 0
   all_max = INF
   for i from 1 to n:
      current_max = max(left_max[i - 1], right_max[i + 1] - customer[i].p)
      if current_max < all_max:
         all_max = current_max
         answer = i
   return answer
```

We use the notation $\mathcal{P}$ defined in (3). After sorting, the answer to the original problem can be expressed as

$$\max_{i=1,2,\ldots,n} \mathcal{P}_i + e_i.$$

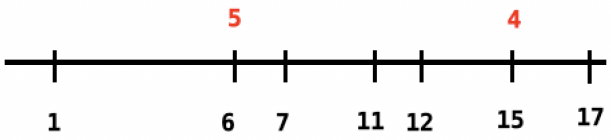
After removing a customer, the remaining customers are still sorted non-increasingly according to $e_i$ to produce the optimal solution. Note that if we remove the *i-th* customer, for $j = 1, 2, \ldots, i-1$, $\mathcal{P}_j + e_j$ is untouched; for $j = i+1, i+2, \ldots, n$, their finish time is changed to $\mathcal{P}_j - p_i + e_j$, so the answer to the problem after deleting the *i-th* person is:

$$\max(\max_{j=1,2,\ldots,i-1} \mathcal{P}_j + e_j, \max_{j=i+1,\ldots,n} \mathcal{P}_j - p_i + e_j)$$

Our final answer is the $i$ that produces the minimum among these values
As for the time complexity, we briefly explain it as follows:
$O(N)$ for reading input
$O(NlgN)$ for sorting the array
$O(N)$ for calculating prefix sum, maximum from left and right
$O(N)$ for finding the least time needed among every removal
So the overall time complexity is $O(NlgN)$

# Problem 6

(1) One of the optimal plans is as follows:




The students in classes located at $x = 1, 7, 11$ can go to the diner with $d = 5$, and the others can go to the diner with $d = 4$.

(2)(3) Let $x = \{x_1, x_2, \ldots, x_n\}$ and $d = \{d_1, d_2, \ldots, d_m\}$. Suppose $x_j$ is the leftmost class that is not covered by any diner yet, and we pick $d_i$ to cover $x_j$. We note that in order to use as few diners as possible, $d_i$ should be placed to the right of $x_j$ and as far away from $x_j$ as possible. That is, $d_i$ should be placed at $x_j + d_i$.
It is obvious that if we wish to cover all classes with the fewest diners, each diner should cover as many classes as possible. We briefly explain the reason that $d_i$ should be placed at $x_j + d_i$.
Let $d_i$ be placed at some $v \in [x_j - d_i, x_j + d_i]$, so it can at least cover $x_j$, which is our intention. We know that $x_j$ is the leftmost class that is not covered yet, so $[v - d_i, x_j)$ does not cover any uncovered class. The range that does cover some classes is $[x_j, v + d_i]$, and the range is expanded as $v$ moves to the right, until it reaches its upper bound, which is $x_j + d_i$. This shows that $d_i$ should be placed at $x_j + d_i$. The following algorithm adopts this idea.

```
diner():
   read class location into x[] and diner info to d[]
   cur_point = -INF //the rightmost point that is covered by some diner
   cur_diner = 0 // the pending diner, using 0-based index
   ans = 0
   for i from 0 to n - 1:
      if x[i] > cur_point: // not covered yet
         if cur_diner == m:
            return -1 // used all the diners but can't cover all classes
         ans += 1
         cur_point = x[i] + 2 * d[cur_diner]
         cur_diner += 1
   return ans
```

The above algorithm goes through each $x_i$ exactly one time, and each $d_i$ is visited at most one time, possibly not visited if not needed. So the time complexity is $O(N + M)$.

(4)

```
diner():
   x[n] stores the locations of classes, 0-based index
   d[m] stores the information of diners, 0-based index
   distance[n][m] stores the first class the m-th diner cannot cover if it is placed at x[n]
       + d[m]
   dp[n][m] stores the answer to each subproblem
   for i from 0 to m - 1: //building the distance table in O(n*m) time
      right_end = 0
      for j from 0 to n - 1:
         while right_end < n and x[right_end] <= x[j] + 2 * d[i]
            right_end += 1
         distance[j][i] = right_end
   for i from n - 1 to 0:
```

```
    for j from m - 1 to 0:
      if j == m - 1 and distance[i][j] != n: // only one diner but cannot cover classes
          from i to n - 1
        dp[i][j] = INF
      else if distance[i][j] == n: //using this diner can cover all the classes from i to
          n - 1
        dp[i][j] = 1
      else:
        dp[i][j] = min(dp[i][j + 1], 1 + dp[distance[i][j]][j + 1])
  return dp[0][0]
```

The above algorithm adopts dynamic programming, we'll show its optimal substructure and overlapping subproblems.
First we define $\mathcal{D}(i,j)$ to be the problem where we consider classes $i, i+1, \ldots, n-1$, and we have diners $j, j+1, \ldots, m-1$. The optimal answer to the problem is stored in $dp[i][j]$.
**Optimal substructure** When we consider $\mathcal{D}(i,j)$, there are only two possibilities. In the first case, we choose to use diner $j$ to cover class $i$ and place it at $x[i]+d[j]$(the reason to do so is discussed in (3)). Then we need to solve the subproblem $\mathcal{D}(i', j+1)$, where $i'$ is the class that cannot be covered by diner $j$. For obvious reason, we don't need to consider classes covered by diner $j$. This situation gives $dp[i][j] = 1 + dp[i'][j+1]$. Suppose the opposite that this is not the optimal answer when we choose diner $j$, then there exists a better answer to $\mathcal{D}(i', j+1)$, which contradicts the fact that $dp[i'][j+1]$ is the optimal answer to $\mathcal{D}(i', j+1)$. The second case, where we choose not to use diner $j$, is similar, only the subproblem is $\mathcal{D}(i, j+1)$. Taking the minimum of the two gives the answer to $\mathcal{D}(i,j)$.
**Overlapping subproblems** Consider $\mathcal{D}(i,j)$. If class $i+1$, but not class $i+2$, is covered by diner $j$ when we place it to cover class $i$, we then have to solve $\mathcal{D}(i+2, j+1)$ in order to solve $\mathcal{D}(i,j)$. It is possible when choosing diner $j$ to cover class $i+1$ in $\mathcal{D}(i+1, j)$, diner $j$ is not able to cover class $i+2$, and we have to solve $\mathcal{D}(i+2, j+1)$ again. In this case $\mathcal{D}(i+2, j+1)$ is an overlapping subproblem, and there are plenty more.
As for the time complexity, there are mainly two loops in the algorithm. The second loop, which is used to fill the array $dp$, obviously runs in $O(MN)$. We need a closer look on the first loop. It seems that it runs in $O(N^2M)$ because of the extra while loop, which is used to determine the the rightmost point diner $j$ can reach. But note that as $j$ increases, this point is also moving rightwards, never leftwards. So when the inner for loop ends, it visits each $x[j]$ at most once, resembling a sliding window whose both ends move in the same direction. The time complexity of the first loop is thus $O(2NM) = O(NM)$, and the overall time complexity is $O(NM) + O(NM) = O(NM)$.

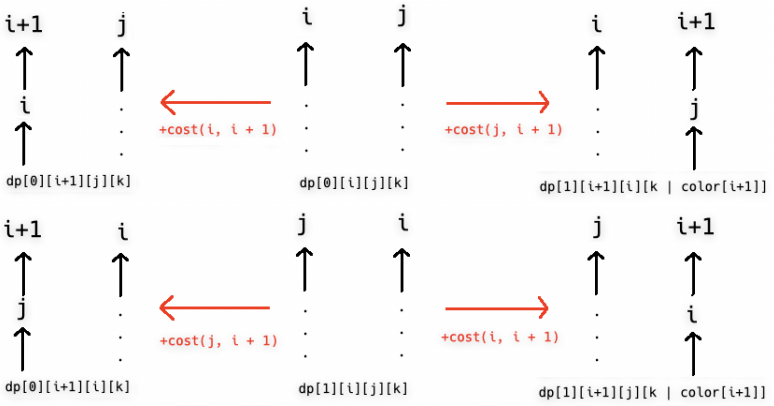
# Problem 7

(1)(2)(3)(4)

```
rainbow():
  candy[N] stores the x-y coordinate and the color of each candy //candies are from 1 to n
  dp[2][2][N][1 << 8] stores the answer to each subproblem
  Initialize array dp to INF
  The cost(i, j) function used below returns the cost to go from candy i to candy j in O(1)
  dp[0][1][0][1 << candy[1].color] = dp[0][1][1][1 << candy[1].color] = 0 //base case
  for i from 1 to n - 1:
    for j from 0 to i - 1:
      for k from 0 to 1 << 8:
        if dp[0][i % 2][j][k] != INF //transition explained below
              dp[0][(i + 1) % 2][j][k] = min(dp[0][(i + 1) % 2][j][k], dp[0][i %
                  2][j][k] + cost(i, i + 1));
              dp[1][(i + 1) % 2][i][k | (1 << candy[i + 1].color)] = min(dp[1][(i + 1) %
                  2][i][k | (1 << candy[i + 1].color)], dp[0][i % 2][j][k] + cost(j, i
                  + 1));
        if dp[1][i % 2][j][k] != INF //transition explained below
```

```
            dp[1][(i + 1) % 2][j][k | (1 << candy[i + 1].color)] = min(dp[1][(i + 1) %
                2][j][k | (1 << candy[i + 1].color)], dp[1][i % 2][j][k] + cost(i, i +
                1));
            dp[0][(i + 1) % 2][i][k] = min(dp[0][(i + 1) % 2][i][k], dp[1][i % 2][j][k] +
                cost(j, i + 1));
ans = INF
for j from 0 to n - 2:
  for k from 0 to 1 << 8:
     if k | (1 << candy[n].color) == (1 << 8) - 2: //if adding the last candy, we have
         all the colors
        ans = min(ans, min(dp[0][(n - 1) % 2][j][k], dp[1][(n - 1) % 2][j][k]) + cost(n,
            j)) + cost(n - 1, n); //manually add the last candy to the two paths
get_path(n, path[][][]) //function described in bonus part
  return ans //the shortest time required
```

We find two paths such that they both start with the first(lowest) candy and end with the last(highest) candy. Except for the two candies, each candy should appear in either path exactly once, with at least one path containing seven different colors.
Let $k$ be the 8-bit bitmask (colors are from 1 to 7, bit 0 unused) that records the colors we have collected so far, and we let $dp[0][i][j][k]$ for $j = 1, 2, \ldots, i-1$ be the optimal answer to the problem where $i$ indicates the candy we consider, i.e. candy $1, 2, \ldots, i$, $k$ is the bitmask of colors collected in the first path, and $j$ is the candy on the top of the first path. $dp[1][i][j][k]$ is similar, except $j$ stands for the candy on the top of the second path. When adding candy $i+1$, we have the following transitions:




We loop until $n-1$ and manually add candy $n$ to both paths outside the loop. Finally, we find the minimum of $dp[0][n][j][k]$ and $dp[1][n][j][k]$, where $k$ has bits from 1 to 7, indicating all the colors are collected. Now we look at the two properties of dynammic programming:
**Optimal substructure** The problem where we need to assign candies $1, 2, \ldots, i$ to either path, can be approached by discussing the candies that is on the top of the paths. Clearly, candy $i$ is one of those candies. We let candy $j$ be the other one, then we have:

- Candy $j$ is in the first path, which corresponds to $dp[0][i][j][k]$
- Candy $j$ is in the second path, which corresponds to $dp[1][i][j][k]$. For transition, from the figure above, we have:

$$dp[0][i][j][k] = \begin{cases} dp[0][i-1][j][k] + cost(i-1, i) & if\ j\ \neq i-1 \\ \min\limits_{j'=1,2,\ldots,i-2}(dp[1][i-1][j'][k] + cost(j', i)) & if\ j = i-1 \end{cases}$$

Suppose this does not give the optimal answer, then we have some $dp'[0][i-1][j][k] < dp[0][i-1][j][k]$ or $dp'[1][i-1][j'][k] < dp[1][i-1][j'][k]$, which contradicts the fact that $dp[0][i-1][j][k]$ and $dp[1][i-1][j'][k]$

are the optimal answers. The transition for $dp[1][i][j][k]$ as follows, and the discussion is similar.

$$dp[1][i][j][k] = \begin{cases} dp[1][i-1][j][k'] + cost(i-1,i) & if\ j\ \neq i-1 \\ \min\limits_{j'=1,2,\ldots,i-2}(dp[0][i-1][j'][k'] + cost(j',i)) & if\ j = i-1 \end{cases} \text{ such that } k'|(1 << candy[i].color) = k$$

**Overlapping subproblem** From a recursive view, both $dp[1][i][i-1][k]$ and $dp[0][i][j][k']$ involves the subproblem $dp[0][i-1][j][k']$ for $j \neq i-1$ and $k'|(1 << candy[i]) = k$, so $dp[0][i-1][j][k']$ is an overlapping subproblem, and there are plenty more.
As for the time complexity, initializing $dp$ to $\infty$ is $O(N^2)$, and there are mainly two loops. The first loop contains three layer, the outer two layer gives $O(N^2)$, the inner layer is constant $2^8$, and all the operations inside the loops are $O(1)$, so we have $O(2^8N^2) = O(N^2)$. Similarly, the second loop gives $O(2^8N) = O(N)$, so the overall time complextiy is $O(N^2)$
For the space complexity, we note that in order to update $dp[0][i][j][k]$ and $dp[1][i][j][k]$, we only need the information in $dp[0][i-1][j][k]$ and $dp[1][i-1][j][k]$. So declaring the array $dp[2][2][N][2^8]$ is sufficient to find the answer. Other arrays and variables are $O(N)$, so the overall space complexity is $O(2 \cdot 2 \cdot 2^8N) + O(N) = O(N)$

- **Bonus** In the above snippet, we can add an array $path[2][N][k]$, which stores $dp[0][i][i-1][k]$ and $dp[1][i][i-1][k]$ for $i = 1, 2, \ldots, n$ and $k$ is the bitmask. We observe that, for $dp[0][i][j][k]$, it can only be obtained from $dp[0][i-1][j][k]$ for $j < i-1$, the same holds for $dp[1][i][i-1][k]$. For $dp[0][i][i-1][k]$, it can be obtained from $dp[1][i-1][j][k]$ for $j < i-1$, and for $j < i-2$, each $dp[1][i-1][j][k]$ can only be obtained from $dp[1][i-2][j][k]$. As the recursion goes on, it turns out, we only need to record $dp[0][i][i-1][k]$ and $dp[1][i][i-1][k]$ to find out how we reach certain state.
  In order to retrieve the path in $O(N^2)$, we need to calculate the prefix of cost, i.e. $[cost(1,2), cost(1,2)+cost(2,3), \ldots]$ and the prefix sum of color count for each color. Preprocessing can be done in $O(N)$ time and $O(N)$ space.
  The procedure for retrieving the path is as follows:
  - Suppose answer is at $dp[s][i][j][k]$, $s = 0, 1$
  - if $j \neq i-1$, descend to $dp[s][j+1][j][k']$, where $dp[s][j+1][j][k'] = dp[s][i][j][k] - cost(j+1 \ldots i-1)$ and $k'|(colors\ in\ path\ from\ j+1\ to\ i-1) = k$, $cost(j+1 \ldots i-1)$ stands for $cost(j+1, j+2) + \ldots, +cost(i-2, i-1)$. All the information can be obtained from preprocessed prefix in $O(1)$
    if $j = i-1$, descend to $dp[s'][j'+1][j'][k']$, where $s' = 1-s, dp[s'][j'+1][j'][k'] = dp[s][i][j][k] - cost(i,j') - cost(j'+1 \ldots i-1)$ and $k'|(1 << candy[i-1].color) = k$.
  - repeat the process until we descend to 1. This way, we obtain the sequence of one path. The other path is simply formed by the numbers not presented.
    All the information we need is stored in $O(N)$ space, and searching for state to descend to is $O(N)$, with at most $N$ search. So the time complexity is $O(N^2)$.

# Reference

## Problem 5

- B07902075 林楷恩/ B07902132 陳威翰

## Problem 6

- B07902028 林鶴哲

## Problem 7

- B07902028 林鶴哲/ B07902132 陳威翰/ B07902075 林楷恩/ B07902141 林庭風