ADA HW #2 - Hand-Written

Student Name: 林楷恩
Student ID: b07902075

Problem 5 - Piepie's Pie Shop

(1) $(2,13) \to (5,5) \to (3,4) \to (7,3) \to (4,2)$
the finished time of each customer is: 15, 12, 14, 20, 23, so the answer is 23.

(2) The problem can be solve by a **greedy algorithm**. At any time we are ready to make a new pie, we always choose to make the pie for the customer with the largest e_i, formally says: $e_i \geq e_j,\ \forall\ i < j$.

Thus, we just need to sort all the (p, e) pair by e in decending order, whose time complexity is $O(N\,lg\,N)$ with a efficient comparison based sorting algorithm (e.g. merge sort). Then, find the biggest finished time of all customers in $O(N)$ and we get the answer. The pseudo-code is as below.

Piepie's Pie Shop

```
input: the number of customers N, preparation time p[], eating time e[]
output: the minimum time needed for the group of customers to finished

function PiePie(N, p[], e[]):
    sort p[], e[] together by e[i] in decending order

    ans ← 0
    preparation_time ← 0
    for i from 1 to N:
        ans ← max(ans, preparation_time + e[i])
        preparation_time ← preparation_time + p[i]

    return ans
```

(3) I first define the finishing time of a sequence S of (p, e) pairs as the function:

$$cost(S) = \max_{1 \leq i \leq N} (\sum_{k=1}^{i} p_k) + e_i$$

- **Proof of greedy choice property by exchange argument**:
Assume $S^* = [(p_1^*, e_1^*), (p_2^*, e_2^*), \ldots, (p_N^*, e_N^*)]$ is an optimal solution. If e_1^* is the largest, then done. If S^* does not contain the greedy choice at the 1^{st} position, i.e. there $\exists\ j > 1$ s.t.

$$e_1^* < e_j^* \quad \text{and} \quad e_j^* = \max_{1 \leq k \leq N} e_k^*$$

Then I exchange (p_1^*, e_1^*) and (p_j^*, e_j^*) to construct a new sequence:

$$S = [(p_1, e_1), \ldots, (p_j, e_j), \ldots, (p_N, e_N)]$$
$$s.t.\ (p_1, e_1) = (p_j^*, e_j^*),\ (p_j, e_j) = (p_1^*, e_1^*) \implies e_1 > e_j \ and\ e_1 = \max_{1 \leq k \leq N} e_k$$

I will then show that $cost(S) \leq cost(S^*)$ by proving that for every $(p_x, e_x) \in S$, there exists a $(p_y^*, e_y^*) \in S^*$ such that

$$(\sum_{k=1}^{x} p_k) + e_x \ \leq\ (\sum_{k=1}^{y} p_k^*) + e_y^*$$

which implies $cost(S) \leq cost(S^*)$, I will divide S into several intervals to disuss:

(a) $x = 1$: let $y = j$, then:

$$p_1 + e_1 = p_j^* + e_j^* < (\sum_{k=i}^{j-1} p_k^*) + p_j^* + e_j^* = (\sum_{k=1}^{j} p_k^*) + e_j^*$$

(b) $2 \le x \le j-1$: let $y = j$, then:

$$(\sum_{k=1}^{x} p_k) < (\sum_{k=1}^{j} p_k) = (\sum_{k=1}^{j} p_k^*) \quad and \quad e_x = e_x^* \le e_j^*$$
$$\implies (\sum_{k=1}^{x} p_k) + e_x \;<\; (\sum_{k=1}^{j} p_k^*) + e_j^*$$

(c) $x = j$: let $y = j$, then:

$$(\sum_{k=1}^{j} p_k) + e_j = (\sum_{k=1}^{j} p_k^*) + e_j < (\sum_{k=1}^{j} p_k^* + e_1) = (\sum_{k=1}^{j} p_k^* + e_j^*)$$

(d) $j+1 \le x \le N$: both the $(\sum_{k=1}^{x} p_k)$ part and e_x are the same as S^*, so let $y = x$, then the statement is true.

This implies $cost(S) \le cost(S^*)$. Sequence S, which contains the greedy choice, is at least as good as the sequence S^*, which does not contain the choice. Thus, it has the **greedy choice property**.

- **Proof of optimal substructure by contradiction**:

  Assume S is a optimal solution in the form of $[\ (p_1, e_1), (p_2, e_2), \ldots, (p_N, e_N)\ ]$.
  Suppose $S \setminus (p_1, e_1)$ is not optimal, then there exists S' such that $cost(S') < cost(S \setminus (p_1, e_1))$

  (a) $p_1 + e_1 < cost(S)$:

$$\begin{aligned}
& cost(S \setminus (p_1, e_1)) = cost(S) - p_1 \\
\implies & cost(S') < cost(S) - p_1 \\
\implies & cost(S' \cup (p_1, e_1)) = max(p_1 + e_1, cost(S') + p_1) \\
\because\ & p_1 + e_1 < cost(S) \quad \text{and} \quad cost(S') + p_1 < cost(S) \\
\implies & cost(S' \cup (p_1, e_1)) < cost(S) \\
\implies & contradiction
\end{aligned}$$

  (b) $p_1 + e_1 = cost(S)$:

$$\begin{aligned}
\because\ & \max_{1 \le i \le N}((\sum_{k=1}^{i} p_k) + e_i) = p_1 + e_1 \quad \therefore \max_{2 \le i \le N}((\sum_{k=1}^{i} p_k) + e_i) \le p_1 + e_1 \\
\implies & \max_{2 \le i \le N}((\sum_{k=2}^{i} p_k) + e_i) = \max_{2 \le i \le N}((\sum_{k=1}^{i} p_k) + e_i) - p_1 \le e_1 = cost(S) - p_1 \\
\implies & cost(S \setminus (p_1, e_1)) = \max_{2 \le i \le N}((\sum_{k=2}^{i} p_k) + e_i) \le cost(S) - p_1 \\
\implies & cost(S') < cost(S \setminus (p_1, e_1)) \le cost(S) - p_1 \\
\implies & cost(S' \cup (p_1, e_1)) = max(p_1 + e_1, cost(S') + p_1) \\
\because\ & p_1 + e_1 = cost(S) \quad \text{and} \quad cost(S') + p_1 < cost(S) \\
\implies & cost(S' \cup (p_1, e_1)) \le cost(S) \\
\implies & contradiction
\end{aligned}$$

By the **greedy choice property** and **optimal structure**, the algorithm is proved to be correct.

(4) The same strategy will not work, here is a counterexample: [$(1, 10), (2, 8), (5, 6), (1, 4)$]

- **larger e first (same algorithm):**
  - piepie00: (1, 10) → (5, 6)
  - piepie01: (2, 8) → (1, 4)

  $\Longrightarrow$ finishing time = 12
- **better arrangement:**
  - piepie00: (1, 10) → (2, 8)
  - piepie01: (5, 6) → (1, 4)

  $\Longrightarrow$ finishing time = 11

(5) First, I claim that for an optimal solution contructed by above algorithm, if we killed one of the elements, the sequence of the rest elements is still an optimal solution. Because based on the algorithm, the eating time of an optimal sequence is always in non-increasing order, and removing an element will not affect the property.

Thus, I just try removing all of the elements and choose the minimum finishing time I can get. The details are described in the pseudo-code:

Piepie's Pie Shop - CHIDORI

```
input: the number of customers N, preparation time p[], eating time e[]
output: the minimum time needed for the group of customers to finished

function PieCHIDORI(N, p[], e[]):
    // pack p[], e[], and index of them together ...O(N)
    S ← [ (p, e, idx) for idx from 1 to N ]
    // sort S by e in decending order ...O(NlgN)
    sort S by e in decending order

    // preprocess prefix sum of S[].p ...O(N)
    prefix_sum[0] ← 0
    for i from 1 to N:
        prefix_sum[i] ← prefix_sum[i-1] + S[i].p

    // preprocess the max finishing time of S[1...i] for i = 1 to N ...O(N)
    prefix_max[0] ← -1
    for i from 1 to N:
        prefix_max[i] ← max(prefix_max[i-1], prefix_sum[i] + S[i].e)

    // preprocess the max finishing time of S[i...N] for i = N to 1 ...O(N)
    suffix_max[N+1] = -1
    for i from N to 1:
        suffix_max[i] ← max(suffix_max[i+1], prefix_sum[i] + S[i].e)

    // enumerate all possible killed customers and find minimum value ...O(N)
    killed ← -1
    ans ← 0
    for i from 1 to N:
        val ← max(prefix_max[i-1], suffix_max[i+1] - S[i].p)
        if val < ans:
            ans ← val
            killed ← S[i].idx

    return (killed, ans)
```

If the i^{th} element is deleted, the maximum finishing time of those prior to it does not change, while those after it will be substracted by $p[i]$, so the maximum will be substracted by $p[i]$, too. Thus, if we have $\max\limits_{1 \le j \le i-1}((\sum_{k=1}^{j} p_k) + e_j)$ (`prefix_max[]`) and $\max\limits_{i+1 \le j \le N}((\sum_{k=1}^{j} p_k) + e_j)$ (`suffix_max[]`), then $cost(S \setminus (p_i, e_i))$ can be computed in $O(1)$ time by the following method:

$$cost(S \setminus (p_i, e_i)) = \max(\max_{1 \le j \le i-1}((\sum_{k=1}^{j} p_k) + e_j), \max_{i+1 \le j \le N}((\sum_{k=1}^{j} p_k) + e_j) - p_i)$$

Both `prefix_max[]` and `suffix_max[]` can be preprocessed in $O(N)$, because if we have `prefix_sum[]` of preparation time, the finishing time of each elements can be computed in $O(1)$ by simply adding it $S[i].e$. Then we iterate from 1 to N and update the minimum finishing time we can get in $O(N)$ because each value can be obtained in $O(1)$.

Therefore, the overall complexity of CHIDORI is $O(NlgN) + O(N) = O(NlgN)$

Problem 6 - Mobile Diners

(1) Place 3 at $x = 4$ such that it is reachable for classrooms at 1 and 7. Place 5 at $x = 16$ such that it is reachable for classrooms at 11, 12, and 17. The cost is 2 mobile diners.

(2) (I just answer (3))

(3) Because the constraints, I can only choose the mobile diner d_m where m is the lowest available index, place it at a position such that it is reachable for the smallest x_i which does not have any mobile diner to go, and maximize the number of x_j where $j > i$ that can go to this mobile diner. This position is $x_i + d_m$ obviously, since the range $[x_i,\ x_i + 2 \times d_m]$ cover x_i and stretches to the right as far as possible, it is the best position.

Mobile Diners (I)

```
input: b_1, b_2, ..., b_M , x_1, x_2, ..., x_N
output: minimum number of mobile diners required

function MD_restricted(b[], x[]):
    i ← 1
    ans ← 0

    for m from 1 to M:
        ans ← ans + 1
        right_bound ← x[i] + 2 * b[m]
        while i ≤ N and x[i] ≤ right_bound
            i ← i + 1
        if i > N:
            break

    return ans
```

m starts from 1 and strictly increases by 1 every iteration, since $m \le M$, this part is $O(M)$, while i starts from 1 and at least increases by 1, by $i \le N + 1$, this part is $O(N)$. Thus the overall time complexity is $O(N + M)$.

(4) Let $MD(m, i)$ be the **minimum number of mobile diners required to make $x_i \dots x_N$ be able to have meal using only d_k where $k >= m$**, and this function can be recursively defined as:

Let $next(p)$ be the smallest i such that $x_i > p$

$$MD(m,i) = \begin{cases} \infty \text{ (invalid)} & if\ m > M\ and\ i \le N \\ 0 \text{ (valid)} & if\ i > N \\ min(1 + MD(m+1, next(x_i + 2 \times d_m)), MD(m+1, i)) & if\ m \le M\ and\ i \le N \end{cases}$$

By the definition, the dynamic programming procedure is obvious:

Mobile Diners (II)

```
input: b_1, b_2, ..., b_M , x_1, x_2, ..., x_N
output: minimum number of mobile diners required

function MD(b[], x[]):
    // preprocess next[][]
    for m from 1 to M:
        for i from 1 to N:
            p ← next[m][i-1] if i > 1 else 1
            while p ≤ N and x[p] ≤ x[i] + 2 * b[m]
                p ← p + 1
            next[m][i] ← p

    // initialize base cases
    for i from 1 to N:
        dp[M + 1][i] ← ∞
    dp[M + 1][N + 1] ← 0

    // dynamic programming by bottom-up approach
    for m from M to 1:
```

```
    for i from N to 1:
        dp[m][i] ← min(1 + dp[m + 1][ next[m][i] ], dp[m + 1][i])

return dp[1][1]
```

- **Proof of correctness by showing optimal substructure**

Suppose OPT is an optimal solution to $MD(m, i)$:

- case 1: d_m is in OPT

$$\begin{aligned}&\text{but } OPT \setminus \{d_m\} \text{ is not optimal to } MD(m+1, next(x_i + 2 \times d_m))\\ \implies &\exists\ |OPT'| < |OPT \setminus \{d_m\}| = |OPT| - 1\\ \implies &|OPT' \cup \{d_m\}| = |OPT'| + 1 < |OPT \setminus \{d_m\}| + 1 = |OPT|\\ \implies &OPT' \cup \{d_m\} \text{ is a better solution to } MD(m, i) \text{ than } OPT\\ \implies &\textit{contradiction}\end{aligned}$$

- case 2: d_m is not in OPT

$$\begin{aligned}&\text{If } OPT' \text{ is a better solution to } MD(m+1, i)\\ \implies &OPT' \text{ is also a better solution to } MD(m, i)\\ \implies &\textit{contradiction}\end{aligned}$$

Therefore, the optimal solution of $MD(m, i)$ can be computed with optimal solution to subproblems.

- **Proof of time complexity**

(a) **preprocessing next(p)**: Though the domain of this function is $\mathbb{R}$, the arguments passed to it are always in $\{x_i + 2 \times d_m \mid 1 \leq i \leq N,\ 1 \leq m \leq M\}$ which has at most NM different values. Thus, we can store the outputs of this function in a 2-D array `next[][]`. However, the computation of single `next[m][i]` is still $O(N)$, so we should take use of the monotonic increasing property of x. By this property, `next[m][i]` $\geq$ `next[m][i-1]`, so for `next[m][i]`, we can start searching from `next[m][i]`. Then, when computing `next[m][1]...next[m][N]`, we just need to go through x once, with time complexity $O(N)$. Thus, the overall complexity of this part is $O(NM)$

(b) **initializing base cases**: assigning values to the $(M+1)^{th}$ row takes $O(N)$ time complexity.

(c) **filling table by bottom-up approach**: the table' size is $N \times M$, and for every cell we can compute its value in $O(1)$ by simply comparing two values. Therefore, the time complexity of this part is $O(NM)$.

Combining the 3 parts, the time complexity of this algorithm is:

$$O(NM) + O(N) + O(NM) = O(NM)$$

Problem 7 - Rainbow Rarity Rally

Reference: 謝宗晅、蔡銘軒、陳威翰

(a) (Task 1 & 2)

Since we can invert a route's part 1 and part 2 by reversing the direction of each move, I regard a route as 2 contestants fly from p_0 to p_N simultaneously without passing through the same points except p_0 and p_N. Let $R(i,j)$ be the smallest amount of time required for 2 contestants such that one of them stops at p_i, while another stops at p_j. Because a route should pass all points, if $j < i-1$, then the one stops at p_i must pass through $p_{j+1} \dots p_i$. Then, when considering p_{i+1}, we have 2 cases:

- i go to $i+1$: R(i, j) + f(i, i + 1) → R(i + 1, j)
- j go to $i+1$: R(i, j) + f(j, i + 1) → R(i + 1, i)

Let the base case be $R(1,0) = f(i,j)$, then we can fill the table from lower i to higher i.

(Task 2)

```
input: N points denoted by x-coordinates x[], y-coordinates y[], and their
    colors c[]
output: the smallest amount of time required to finish the race

function f(i, j):
    return (x[i] - x[j])^2 + (y[i] - y[j])^2

function RRR():
    initialize dp[] to ∞
    // base case
    dp[0] ← f(0, 1)

    for i from 1 to N - 1:
        initialize dp_tmp[] to ∞
        for j from 0 to i - 1:
            update dp_tmp[i] with dp[j] + f(j, i+1)
            update dp_tmp[j] with dp[j] + f(i, i+1)
        copy dp_tmp[] to dp[]

    ans ← ∞
    // because the one at p_j has not reach p_N yet, f(i, N) should be added.
    for i from 0 to N - 1:
        update ans with dp[i] + f(i, N)
    return ans
```

Proof of correctness:

I will prove it by showing the optimal substructure.

- Case 1:

  Assume OPT is an optimal solution to $R(i+1,j)$, but $OPT \setminus \{p_{i+1}\}$ is not optimal to $R(i,j)$
  $\Longrightarrow \exists\ OPT'\ s.t.\ |OPT'| < |OPT \setminus \{p_{i+1}\}|$
  $\Longrightarrow |OPT' \cup \{p_{i+1}\}| = |OPT'| + f(i,i+1) < |OPT \setminus \{p_{i+1}\}| + f(i,i+1) = |OPT|$
  $\Longrightarrow OPT' \cup \{p_{i+1}\}$ is a better than OPT
  $\Longrightarrow$ *contradiction*

- Case 2:

  Assume OPT is an optimal solution to $R(i+1,i)$, but $OPT \setminus \{p_{i+1}\}$ is not optimal to $R(i,j)$
  $\Longrightarrow \exists\ OPT'\ s.t.\ |OPT'| < |OPT \setminus \{p_{i+1}\}|$
  $\Longrightarrow |OPT' \cup \{p_{i+1}\}| = |OPT'| + f(j,i+1) < |OPT \setminus \{p_{i+1}\}| + f(j,i+1) = |OPT|$
  $\Longrightarrow OPT' \cup \{p_{i+1}\}$ is a better than OPT
  $\Longrightarrow$ *contradiction*

Proof of complexity:

- **time complexity:**
  i. initialization: $O(N)$

  ii. for i from 1 to $N-1$: $O(N)$ times
    * initialize `dp_tmp[]` to ∞: $O(N)$
    * $O(1)$ transition $\forall j \in \{0, 1, \ldots, i-1\}$: $O(1) \times i = O(N)$
    * copy `dp_tmp[]` to `dp[]` (size=N): $O(N)$

    $\Longrightarrow$ **overall**: $O(N) \times O(N) = O(N^2)$

  iii. go through dp[ ] (size=N) to find the minimum: $O(N)$

  $\Longrightarrow$ Sum up i. ii. iii., the overall time complexity is $O(N) + O(N^2) + O(N) = O(N^2)$ #

- **space complexity:** Because when we complete the transition from $R(i, j)$ to $R(i+1, j)$, we no longer need the value $R(i, j)$ anymore, so we can discard these values and use the same block of memory to store the new values. Therefore, we just need 2 array of size N, whose space complexity is $O(N)$.

(b) (Task 3 & 4)

For this problem, I add two new parameters to the previous definition. Let $R(i, j, x, S)$ be the smallest amount of time required for 2 contestants such that one of them stops at p_i, while another stops at p_j. Additionally, x indicates which contestant S is associated with. If $x = 0$, then S is the set of colors that the one stops at p_j has collected, while if $x = 1$, then S is the set of colors that the one stops at p_i has collected. By the definition, the transition becomes:

* $x = 0$
  - Base Case: $R(1, 0, 0, \emptyset) = f(0, 1)$
  - j go to $i+1$: $R(i, j, 0, S) + f(j, i+1) \to R(i+1, i, 1, S \cup \{c_{i+1}\})$
  - i go to $i+1$: $R(i, j, 0, S) + f(i, i+1) \to R(i+1, j, 0, S)$
* $x = 1$
  - Base Case: $R(1, 0, 1, \{c_1\}) = f(0, 1)$
  - j go to $i+1$: $R(i, j, 1, S) + f(j, i+1) \to R(i+1, i, 0, S)$
  - i go to $i+1$: $R(i, j, 1, S) + f(i, i+1) \to R(i+1, j, 1, S \cup \{c_{i+1}\})$

After we transition to $R(N, j, x, S)$, the answer will be the minimum among two sets:

$$\left\{ R(N, j, x, S) + f(j, N) \mid x = 0 \text{ and } S \cup \{c_N\} = \{1, 2, 3, 4, 5, 6, 7\} \right\}$$

and

$$\left\{ R(N, j, x, S) + f(j, N) \mid x = 1 \text{ and } S = \{1, 2, 3, 4, 5, 6, 7\} \right\}$$

Moreover, to boost up set union operation and indexing, I take use of bitset to implement set operation. since there are only 7 colors, so the value of bitmask is up to $2^7 - 1 = 127$.

The detailed pseudo-code is provided in the next page.

(Task 4)

```
input: N points denoted by x−coordinates x[], y−coordinates y[], and their
    colors c[]
output: the smallest amount of time required to finish the race

function f(i, j):
    return (x[i] − x[j])^2 + (y[i] − y[j])^2

function RRR():
    initialize dp[][][] to ∞

    // base cases
    dp[0][0][0] ← f(0, 1)
    dp[0][1][1 << (c[1] − 1)] ← f(0, 1)

    for i from 1 to N − 1:
        initialize dp_tmp[][][] to ∞

        // x = 0
        for S from 0 to 127:
            for j from 0 to i − 1:
                update dp_tmp[i][1][S | (1 << (c[i+1] − 1))] with dp[j][0][S] +
                    f(j, i+1)
                update dp_tmp[j][0][S] with dp[j][0][S] + f(i, i+1)

        // x = 1
        for S from 0 to 127:
            for j from 0 to i − 1:
                update dp_tmp[i][0][S] with dp[j][1][S] + f(j, i+1)
                update dp_tmp[j][1][S | (1 << (c[i+1] − 1))] with dp[j][1][S] +
                    f(i, i+1)

        copy dp_tmp[][][] to dp[][][]

    ans ← ∞

    // because the one at p_j has not included the color of p_N
    for i from 0 to N − 1:
        for S from 0 to 127:
            if (S | (1 << (c[N] − 1)) = 127:
                update ans with dp[i][0][S] + f(i, N)

    for i from 0 to N − 1:
        update ans with dp[i][1][127] + f(i, N)

    return ans
```

Proof of correctness:

Compared with (Task 1 & 2), the optimal substructure property stays the same, because the new parameters has no effect on the values. It just divides the cases of $R(i, j)$ into several classes of different color sets. So the same method of proof will work properly.

Proof of complexity:

- **time complexity:** Becuase the additional 2 parameters are all constants, so it does not affect time complexity at all. Below is the very similar analysis:
  i. initialization: $2 \times 128 \times N = O(N)$

  ii. for i from 1 to $N - 1$: $O(N)$ times
    * initialize `dp_tmp[][][]` to ∞: $2 \times 128 \times N = O(N)$
    * $O(1)$ transition $\forall j \in \{0, 1, \ldots, i-1\}$ $\forall x \in \{0, 1\}$ $\forall S \in \{0, 1, \ldots, 127\}$: $i \times 2 \times 128 \leq 256 \times N = O(N)$
    * copy `dp_tmp[]` to `dp[]` (size=N): $O(N)$

  $\Longrightarrow$ **overall**: $O(N) \times O(N) = O(N^2)$

iii. go through dp[ ][ ][ ] (size=$2 \times 128 \times N$) to find the minimum: $2 \times 128 \times N = O(N)$

$\Longrightarrow$ Sum up i. ii. iii., the overall time complexity is $O(N) + O(N^2) + O(N) = O(N^2)$ #

- **space complexity:** Similar to time complexity, because $128 \times 2 \times O(N) = O(N)$, the complexity is still $O(N)$.