Homework #2 Solution

Contact TAs: `ada-ta@csie.ntu.edu.tw`

Problem 1 - Line of Battle (Programming) (10 points)

Solution

(Note that in this document, a lot of edge cases are omitted for brevity.)

A simple $\mathcal{O}(n^3)$ ($\mathcal{O}(n)$ states, $\mathcal{O}(n^2)$ transition) dynamic programming algorithm can be formulated as follows. Let d_i be the number of combinations of fleets formed by ships in the segment $[1, i]$. We have the transition

$$d_i = \sum_{j<i, [j+1,i] \text{ is a valid fleet}} d_j$$

We can easily improve the transition to $\mathcal{O}(n)$ by preprocessing the prefix sum of the team awareness values, enabling us to decide whether a fleet is valid in $\mathcal{O}(1)$ time.

In order to improve the overall time complexity to $\mathcal{O}(n \log n)$, we need a few observations. Firstly, notice that for each ship l, the set of possible r such that $[l, r]$ forms a valid fleet is a segment. For example, in the following configuration, $[1, 1], [1, 2], [1, 3], [1, 4]$ all form valid fleets.

31111
11111

Therefore, for each ship l, we can find its corresponding aforementioned segment via binary search in $\mathcal{O}(\log n)$ time.

On the other hand, the recurrence we previously used does not have such properties, i.e., for each i, the j's do not necessarily form a segment. We thus have to change from a "pull" approach to a "push" approach:

```
for l in {1..n}:
  r <- the largest r s.t. [l, r] is valid -- O(log n)
  add dp[l] to every element in dp[l + 1 .. r] -- (*) O(?)
```

It remains to process $(*)$ effectively. Notice that given an sequence of numbers a and an arbitrary d, if we increase a_l by d and decrease a_{r+1} by d, and let b and b' be the prefix sum of a before and after the changes respectively, then $b'_i = \begin{cases} b_i + d & \text{if } l \le i \le r \\ b_i & \text{otherwise} \end{cases}$.

Therefore, we can effectively achieve $(*)$ in $\mathcal{O}(1)$ as follows:

```
for l in {1..n}:
  r <- the largest r s.t. [l, r] is valid -- O(log n)
  prefix_sum <- prefix_sum + dp_diff[l]
  dp_diff[l + 1] <- dp_diff[l + 1] + prefix_sum
  dp_diff[r + 1] <- dp_diff[r + 1] - prefix_sum
output prefix_sum as the answer
```

Implementation Details

- The binary search can be implemented via `upper_bound` since $lead[l] \le team[r] - team[l] \iff team[l] + lead[l] \le team[r]$.
- Notice the difference between `upper_bound` and `lower_bound`.
- The answer may get very large (for example, when all team values equal 0); hence you need to do the arithmetic under modulo $10^9 + 7$ instead of only doing the modulo at the end.
- The `%` operator in C/C++[1] truncates the quotient towards zero, and may return a negative number if the dividend is negative. Therefore, while doing subtraction, you need to write $(a - b + p)\%p$ instead of just $(a - b)\%p$.

Reference Implementation

```
// Template omitted for brevity
#include <algorithm>
#include <iostream>
#include <numeric>

constexpr int kN = 2000005, kMod = int(1E9 + 7);

int lead[kN], max_fleet[kN], diff[kN];
int64_t team[kN];

int main() {
  int n = ADA::Init();
  for (int i = 0; i < n; i++) lead[i] = ADA::GetLeadership();
  for (int i = 0; i < n; i++) team[i] = ADA::GetTeamValue();
  std::partial_sum(team, team + n, team);
  for (int i = 0; i < n; i++) {
    max_fleet[i] =
        int(std::upper_bound(team + i, team + n, team[i] + lead[i]) - team);
  }
  int64_t dp_i = 1;
  diff[1] = -1;
  for (int i = 0; i < n; i++) {
    dp_i = (dp_i + diff[i]) % kMod;
    diff[i + 1] = (diff[i + 1] + dp_i) % kMod;
    diff[max_fleet[i] + 1] = (diff[max_fleet[i] + 1] - dp_i + kMod) % kMod;
  }
  dp_i = (dp_i + diff[n]) % kMod;
  std::cout << dp_i << '\n';
}
```

[1]Technically, the behavior is implementation-defined before C++11, though most compiler implementations still do this.

Problem 2 - Chunithm (Programming) (15 points)

Solution

We can solve this problem with dynamic programming. The following is the definition:

$D_{i,j}$ = the minimum cost when:

- Your are at the i-th second. $(0 \le i < N)$
- The hand which is not used to press the keyboard is at j-th note. $(0 \le i < M)$

Since the left hand and the right hand are equivalent, the definition of D does not designate which hand should be used. Whether you use left hand or right hand to press the keyboard, the answer will be the same.

Then, the transition is defined as follows:

$$D_{i,j} = \min(D_{i-1,t} + \min(\text{dis}(j,t) + \text{dis}(a_i, a_{i-1}), \text{dis}(j, a_{i-1}) + \text{dis}(a_i, t))) \text{ for } 0 \le t < M$$

$\text{dis}(x, y)$ is the cost of moving a hand from position x to y, which defined as:

$$\text{dis}(x, y) = \max(|x - y| - k, 0)$$

Since we have a table D of size $N \times M$ and the transition runs in $O(M)$ (checking all the t), the total time complexity is $O(N \times M^2)$.

Improvement

To get the total points of the problem, we need to make it run in $O(N \times M)$. Here is the idea:

When you update the table D, try to solve all $D_{i,j}$ with $0 \le j < M$ together. And consider the following cases:

1. Two hands move from (a_{i-1}, t) to (j, a_i): In this case, we can calculate the cost of either hand of $D_{i,j}$ with $0 \le j < M$ in $O(M)$. The left hand moves from the same position (a_{i-1}) to a different position (j); the right hand moves from a different position (t) to the same position (a_i). We can calculate the cost of both of them and add them together.
2. Two hands move from (a_{i-1}, t) to (a_i, j): In this case, calculating the cost of left hand (the one on a_i) is simple. However, calculating the cost of right is much more complicated. We need to consider the following two sub-cases:
   (a) $|j - t| \ge K$: You can consider $t < j$ and $t > j$ separately (they are equivalent). After you already know the answer of $D_{i,j}$, you can reuse the previous result and calculate $D_{i,j+1}$ in $O(1)$. The formula is $D_{i,j+1} = \min(D_{i,j} + 1, D_{i-1,j-K} + \text{dis}(a_i, a_{i-1}))$.
   (b) $|j-t| \le K$: In this case, the cost of moving right hand is 0. However, to find from where the right hand should move is the most difficult part. We need to a technique called "monotone queue optimization" ("單調隊列優化" in Chinese).

Reference Implementation

```
#include <iostream>
#define MAXN 100000
#define MAXM 300
#define INF 100000000
using namespace std;
int a[MAXN];
int dp[2][MAXM] = {};
int dq[MAXM], ql, qr;
inline void qclear() { ql = qr = 0; }
inline int qsize() { return qr - ql; }
inline void qpush_back(int v) { dq[qr++] = v; }
inline int qfront() { return dq[ql]; }
inline int qback() { return dq[qr-1]; }
inline void qpop_front() { ql++; }
inline void qpop_back() { qr--; }
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, m, k;
    cin >> n >> m >> k;
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    for (int i = 1; i < n; ++i)
    {
        for (int j = 0; j < m; ++j)
            dp[i&1][j] = INF;
        // case 2b
        int cost_aij = max(0, abs(a[i] - a[i-1]) - k);
        qclear();
        for (int j = 0; j < m; ++j)
        {
            if (qsize() && j - qfront() > k) qpop_front();
            while (qsize() && dp[i-1&1][qback()] > dp[i-1&1][j]) qpop_back();
            qpush_back(j);
            dp[i&1][j] = min(dp[i&1][j], dp[i-1&1][qfront()] + cost_aij);
        }
        qclear();
        for (int j = m-1; j >= 0; --j)
        {
            if (qsize() && qfront() - j > k) qpop_front();
            while (qsize() && dp[i-1&1][qback()] > dp[i-1&1][j]) qpop_back();
            qpush_back(j);
            dp[i&1][j] = min(dp[i&1][j], dp[i-1&1][qfront()] + cost_aij);
        }
        // case 2a
        int best = INF;
        for (int j = k; j < m; ++j)
        {
            best = min(best + 1, dp[i-1&1][j-k]);
            dp[i&1][j] = min(dp[i&1][j], best + cost_aij);
        }
        best = INF;
        for (int j = m-k-1; j >= 0; --j)
        {
            best = min(best + 1, dp[i-1&1][j+k]);
            dp[i&1][j] = min(dp[i&1][j], best + cost_aij);
        }
        // case 1
        best = INF;
        for (int j = 0; j < m; ++j)
            best = min(best, dp[i-1&1][j] + max(0, abs(a[i] - j) - k));
        for (int j = 0; j < m; ++j)
            dp[i&1][j] = min(dp[i&1][j], max(0, abs(j - a[i-1]) - k) + best);
    }
    // for (int i = 0; i < n; ++i)
    // {
    //     for (int j = 0; j < m; ++j)
    //         cout << dp[i&1][j] << " \n"[j == m-1];
    // }
    int ans = INF;
    for (int j = 0; j < m; ++j)
        ans = min(ans, dp[n-1&1][j]);
    cout << ans << endl;
}
```

Problem 3 - Pokémon GO (Programming) (15 points)

Solution

First of all, let's determine the optimal order of any subset of K Pokémon. If there are some Pokémon in this subset with attack power $A_i = 0$, then it's obviously optimal to put them at the beginning. Therefore, without loss of generality, we may assume that all the Pokémon in the subset have non-zero attack powers.

Consider some arrangement of the Pokémon $P = \{(A_1, B_1), (A_2, B_2), \ldots, (A_K, B_K)\}$, with $A_i \neq 0$ for all $1 \leq i \leq K$, if there exists $1 \leq s < K$ such that

$$B_s \times A_{s+1} < B_{s+1} \times A_s \iff \frac{B_s}{A_s} < \frac{B_{s+1}}{A_{s+1}}$$

then the arrangement $P' = \{(A_1, B_1), \ldots, (A_{s+1}, B_{s+1}), (A_s, B_s), \ldots, (A_K, B_K)\}$ always result in higher damage.

To prove the claim, observe that,

$$\text{Damage}(P) = \sum_{i=1}^{K} \left(A_i \times (\sum_{j=1}^{i-1} B_j) \right)$$

$$\begin{aligned}\text{Damage}(P') &= \sum_{i=1}^{s-1} \left(A_i \times (\sum_{j=1}^{i-1} B_j) \right) + A_{s+1} \times (\sum_{j=1}^{s-1} B_j) + A_s \times (B_{s+1} + \sum_{j=1}^{s-1} B_j) + \sum_{i=s+2}^{K} \left(A_i \times (\sum_{j=1}^{i-1} B_j) \right) \\ &= \text{Damage}(P) - A_{s+1} \times B_s + A_s \times B_{s+1} > \text{Damage}(P)\end{aligned}$$

Therefore, let $\text{OPT} = \{(A_1, B_1), (A_2, B_2), \ldots, (A_K, B_K)\}$ be the optimal arrangement of the K Pokémon, then

$$\frac{B_1}{A_1} \geq \frac{B_2}{A_2} \geq \cdots \geq \frac{B_K}{A_K}$$

must hold, otherwise there must exist an adjacent pair which violates the condition. By swapping the pair we get a better arrangement, which contradicts with the optimality of OPT. This implies that we can sort the Pokémon decreasingly with respect to the ratio of the potential value and the attack power to obtain the optimal arrangement. This gives the solution to the first subtask ($N = K$).

To solve the cases when $K < N$, let $f(i, j, k)$ be the maximum damage one can achieved by selecting j Pokémon among the first i Pokémon, such that the sum of the potential value of the selected Pokémon is k. Then

$$f(i,j,k) = \begin{cases} -\infty, & \text{if } i = 0, j \neq 0, k \neq 0 \\ 0, & \text{if } i = 0, j = 0, k = 0 \\ f(i-1, j, k), & \text{if } j = 0 \text{ or } k < B_i \\ \max(f(i-1, j, k), f(i-1, j-1, k - B_i) + (k - B_i) \times A_i), & \text{otherwise} \end{cases}$$

The initial state is $f(0, 0, 0) = 0$, which corresponds to the state where we choose 0 Pokémon out of the first 0 Pokémon, with the sum of their potential value being 0. The transition

$$f(i, j, k) \leftarrow f(i-1, j, k)$$

corresponds to ”not choosing” the i-th Pokémon, while the transition

$$f(i, j, k) \leftarrow f(i - 1, j - 1, k - B_i) + (k - B_i) \times A_i$$

corresponds to ”choosing” the i-th Pokémon and dealing $(k - B_i) \times A_i$ damage to the opponent. The final answer to the problem is

$$\max_{0 \le p \le \sum_{i=1}^{N} B_i} f(N, K, p)$$

The time-complexity and the space-complexity of this dynamic programming approach is $O(N \times K \times NK) = O(N^4)$, which may be insufficient to pass the memory limit of 65536KB when $N = 100$. To reduce the space complexity, notice that the calculation of $f(i, j, k)$ only requires $f(i - 1, *, *)$, implying that we can use the similar technique of ”rolling” the DP table as in MiniHW4 and get a $O(K \times NK) = O(N^3)$ space-complexity.

Reference Implementation

```cpp
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

constexpr int kInf = int(1E9);

int main() {
  size_t n, k;
  std::cin >> n >> k;
  std::vector<std::pair<int, int>> p(n);
  for (size_t i = 0; i < n; ++i) std::cin >> p[i].first >> p[i].second;
  std::sort(p.begin(), p.end(), [](const auto &a, const auto &b) {
    if (!a.first && !b.first) return a.second < b.second;
    if (!a.first || !b.first) return !a.first;
    return a.second * b.first > b.second * a.first;
  });

  std::vector<std::vector<int>> prv_damage(
      k + 1, std::vector<int>(k * 100 + 1, -kInf));
  prv_damage[0][0] = 0;

  for (size_t i = 0; i < n; ++i) {
    std::vector<std::vector<int>> cur_damage(
        k + 1, std::vector<int>(k * 100 + 1, -kInf));
    for (size_t j = 0; j <= k; ++j) {
      for (size_t s = 0; s <= k * 100; ++s) {
        cur_damage[j][s] = prv_damage[j][s];
        if (j > 0 && s >= p[i].second) {
          int damage = (s - p[i].second) * p[i].first;
          cur_damage[j][s] = std::max(
              cur_damage[j][s], prv_damage[j - 1][s - p[i].second] + damage);
        }
      }
    }
    prv_damage.swap(cur_damage);
  }
  int ans = 0;
  for (size_t i = 0; i <= k * 100; ++i) ans = std::max(ans, prv_damage[k][i]);
  std::cout << ans << std::endl;
  return 0;
}
```

Problem 4 - Weigh Anchor (Programming) (15 points)

Solution

First of all, without loss of generality, let's assume that $s_1 \leq s_2 \leq s_3$. Then,

$$s_1 + s_2 + s_3 \geq s_2 + s_3 \geq s_1 + s_3 \geq s_1 + s_2, s_3 \geq s_2 \geq s_1$$

Obviously, the answer is `-1` if and only if there exists an enemy with strength $> s_1 + s_2 + s_3$.

By the inequality above, we know that there might exist some enemies with strength $> s_2 + s_3$. In this case, the only combination capable of defeating them is (s_1, s_2, s_3), so it's optimal to use then to fight those enemies.

Then, after eliminating the enemies described above, there might exist some enemies with strength $> s_1 + s_3$. In this case both (s_2, s_3) and (s_1, s_2, s_3) are capable of defeating them, but choosing (s_2, s_3) is undoubtedly better since s_1 gets a chance to defeat another enemy (if there's any). Therefore, the optimal strategy here is to use (s_2, s_3) to fight those enemies, and use s_1 to fight the strongest enemy which it has the ability to defeat.

Similar argument can be applied when there are some enemies with strength $> \max(s_1 + s_2, s_3)$. The optimal strategy is to use (s_1, s_3) to fight them, and use s_2 to fight the strongest enemy which it has the ability to defeat.

If $s_1 + s_2 > s_3$, we should use the same strategy again to eliminate all enemies with strength $> s_3$.

Now we've come to the situation when all remaining enemies have strength $\leq s_3$, i.e., s_3 is capable of defeating all the enemies. The optimal strategy here is to repeatedly use s_3 to defeat the strongest enemy, and use s_1 and s_2 separately if there exists an enemy which s_2 is capable of defeating (otherwise use $s_1 + s_2$ to fight the strongest enemy they can defeat).

To prove that, consider one of the optimal strategy OPT. If at some point there exists an enemy with strength $\leq s_2$ and OPT chooses to use $s_1 + s_2$ to fight it, then by replacing this action with $(s_1), (s_2), (s_3)$ we obtain another strategy OPT' with $\text{Round}(\text{OPT}') \leq \text{Round}(\text{OPT})$ (since s_1 gets an extra chance to fight some enemy). The optimality of OPT implies $\text{Round}(\text{OPT}') = \text{Round}(\text{OPT})$.

To simulate the idea above, we need to

- Know the strength of the strongest one among all remaining enemies (finding the maximum value).
- Know the strength of the strongest enemy which some combination can defeat (finding the maximum possible value less than or equal to some particular value).
- Kill an enemy (deleting an element).

Any type of balanced binary search tree will support all the functionalities required in this problem, and `std::multiset` is one of them. There also exists an alternative solution which solves the problem by counting the number of enemies in each "category" (the enemy falls into the category of the weakest combination which can defeat it) and doing some complicated case-by-case analyses.

Reference Implementation

```
#include <algorithm>
#include <iostream>
#include <set>
```

```cpp
int main() {
  int n, a[3];
  std::cin >> n >> a[0] >> a[1] >> a[2];
  std::sort(a, a + 3);
  std::multiset<int> s;
  for (int i = 0; i < n; ++i) {
    int x; std::cin >> x;
    s.insert(x);
  }
  if (*s.rbegin() > a[0] + a[1] + a[2]) {
    std::cout << -1 << std::endl;
    return 0;
  }
  int ans = 0;
  while (!s.empty() && *s.rbegin() > a[1] + a[2]) {
    // use s1 + s2 + s3
    s.erase(std::prev(s.end()));
    ans += 1;
  }
  while (!s.empty() && *s.rbegin() > a[0] + a[2]) {
    // use s2 + s3 and s1
    s.erase(std::prev(s.end()));
    auto it = s.upper_bound(a[0]);
    if (it != s.begin()) s.erase(std::prev(it));
    ans += 1;
  }
  while (!s.empty() && *s.rbegin() > std::max(a[0] + a[1], a[2])) {
    // use s1 + s3 and s2
    s.erase(std::prev(s.end()));
    auto it = s.upper_bound(a[1]);
    if (it != s.begin()) s.erase(std::prev(it));
    ans += 1;
  }
  if (a[0] + a[1] > a[2]) {
    while (!s.empty() && *s.rbegin() > a[2]) {
      // use s1 + s2 and s3
      s.erase(std::prev(s.end()));
      auto it = s.upper_bound(a[2]);
      if (it != s.begin()) s.erase(std::prev(it));
      ans += 1;
    }
  }
  while (!s.empty()) {
    s.erase(std::prev(s.end()));
    auto it = s.upper_bound(a[1]);
    if (it != s.begin()) {
      // use s1, s2 and s3
      s.erase(std::prev(it));
      it = s.upper_bound(a[0]);
      if (it != s.begin()) s.erase(std::prev(it));
    } else {
      // use s1 + s2 and s3
      it = s.upper_bound(a[0] + a[1]);
      if (it != s.begin()) s.erase(std::prev(it));
    }
    ans += 1;
  }
  std::cout << ans << std::endl;
  return 0;
}
```

Problem 5 - Piepie's Pie Shop (Hand-Written) (16 points)

Solution

(1)

Arrange those customers as followed will reach the minimum time needed.

[(2, 13), (5, 5), (3, 4), (7, 3), (4, 2)]

- (2, 13): Piepie starts to make his pie at $t = 0$ and ends at $t = 2$. The customer has his pie starting at $t = 2$ and ends at $t = 15$.
- (5, 5): Piepie starts to make his pie at $t = 2$ and ends at $t = 7$. The customer has his pie starting at $t = 7$ and ends at $t = 12$.
- (3, 4): Piepie starts to make his pie at $t = 7$ and ends at $t = 10$. The customer has his pie starting at $t = 10$ and ends at $t = 14$.
- (7, 3): Piepie starts to make his pie at $t = 10$ and ends at $t = 17$. The customer has his pie starting at $t = 17$ and ends at $t = 20$.
- (4, 2): Piepie starts to make his pie at $t = 17$ and ends at $t = 21$. The customer has his pie starting at $t = 21$ and ends at $t = 23$.

The total time needed is 23 units of time.

(2)

```
# Arguments:
# - A : a list of pairs. Each pair is consisted of preparation time and
#       eating time. The indices are 0-based.
# - N : an integer indicating the number of customers.
def arrangeCustomers(A, N):
    # Arrange the customers
    sort A according to the eating time, in descending order
    # There are several sorting algorithms runing in O(nlgn) time.
    # Merge sort is an example.

    # Calculate time needed
    cookTime = 0
    totalTime = 0
    for i in [0, N):
        cookTime += A[i].p
        totalTime = max(totalTime, cookTime + A[i].e)
    return totalTime
```

(3)

Our greedy choice is **"longest eating time first"**. Formally, for any $1 \le i < N$, $e_i \ge e_{i+1}$.

Proof by Exchange Argument Assume that there is an optimal strategy OPT that exists some i such that $e_i < e_{i+1}$. Define f_i be the time that the i-th customer finishes his pie and and s_i be the time that Piepie starts to prepare the i-th customer's pie, under OPT.
Suppose the strategy OPT' be the strategy that we only exchange i-th customer with $(i+1)$-th customer in OPT. Define f'_i be the time that the i-th customer finishes his pie and and s'_i be the time that Piepie starts to prepare the i-th customer's pie, under OPT'.

Some observations:

(a) For $j \le i$, $s_j = s'_j$ since s_j is the sum of the preparation time of each pie made before j.

(b) By (a), for $j < i$, $f_i = f'_i$ holds.

(c) For $j > i+1$, $s_j = s'_j$ since s_j is the sum of the preparation time of each pie made before j.

(d) Similarly, by (c), for $j > i+1$, $f_j = f'_j$

The time to finish serving a group of customers is determined by the customer with largest f_k. Now, we discuss the customer k in the following four cases.

(1) $k < i$: By observation (b), $f_k = f'_k$, the total time needed does not change.

(2) $k = i$ is impossible. Given that $e_i < e_{i+1}$ as we assumed, we can easily have

$$f_i = s_i + p_i + e_i < s_i + p_i + p_{i+1} + e_{i+1} = f_{i+1}$$

(3) $k = i+1$

- Consider the i-th person in OPT'.

$$\begin{aligned} f'_i &= s'_i + p'_i + e'_i \\ &= s_i + p_{i+1} + e_{i+1} \\ &< s_i + p_i + p_{i+1} + e_{i+1} = f_k \end{aligned}$$

- Consider the $(i+1)$-th person in OPT'.

$$\begin{aligned} f'_{i+1} &= s'_{i+1} + p'_{i+1} + e'_{i+1} \\ &= s_i + p_{i+1} + p_i + e_i \\ &< s_i + p_i + p_{i+1} + e_{i+1} = f_k \end{aligned}$$

We can see that both f'_i and f'_{i+1} are less than f_k.

(4) $k > i+1$: By observation (d), $f_k = f'_k$, the total time needed does not change.

Finally, we can see that total time needed in OPT' is no more than OPT. Thus, we can keep exchanging until the customers are sorted by eating time in descending order.

(4)

Disprove The following example is a counter example.

[(3, 8), (3, 8), (10, 4)]

If they follow the same strategy as subproblem (2). Both piepie00 and piepie01 will make a (3, 8) pie at the beginning, and piepie00 makes the last (10, 4) at last. Total time needed is

$$T = \max(3 + 8, 3 + 8, 3 + 10 + 4) = 17$$

However, if piepie00 makes (3, 8) at the beginning and makes another (3, 8) at $t = 3$ while piepie01 makes (10, 4), the total time needed would be

$$T' = \max(3, 8, 3 + 3 + 8, 10 + 4) = 14$$

Thus, following the strategy in subproblem (2) might not reach the optimal outcome.

(5)

A Fake Solution If we apply the strategy in subproblem (2) and kill the one with largest f_k, this is definitely wrong. An counterexample is shown below

[(1000, 10), (8, 6)]

You will kill (8, 6); however, killing (1000, 10) will decreate the total time needed to 14.

Observation

(a) No matter who we kill, the remaining $N - 1$ customers are still in the same order or else the time needed might not be optimized.

(b) If we kill some i-th person, the finish time for customer j can be divided into two classes:

- $j < i$: The finishing time will not change.
- $i < j$: The finishing time would be
$$f'_j = f_j - p_i$$

Brute-force Based on observation (a), we can firstly sort all the customers by eating time in descending order as we did in subproblem (2). Also, by observation (b), we can have time needed if we kill i-th customer be

$$\begin{aligned} t'_i &= \max(f_0, f_1, ..., f_{i-1}, f_{i+1} - p_i, f_{i+2} - p_i, ..., f_{N-1} - p_i) \\ &= \max(\max(f_0, f_1, ..., f_{i-1}), \max(f_{i+1}, f_{i+2}, ..., f_{N-1}) - p_i) \end{aligned}$$

Although determining each i needs a $O(N)$ to calculate maximum numbers, we can build a prefix maximum and suffix maximum to access those maximum numbers in $O(1)$ time.

Problem 6 - Mobile Diners (Hand-Written) (14 points)

Solution

Assume the smaller x-coordinate lies on the left and larger-x coordinate lies on the right.

(1)

- Place the first mobile diner at $x = 4$ and it can serve classrooms between $x = 1$ to $x = 7$. The students in first two classes are satisfied.
- Place the third mobile diner at $x = 14$ and it can serve classrooms between $x = 9$ to $x = 19$. The students in last three classes are satisfied.

(2)

Goto (3)

(3)

Starting from left and iterate over all classrooms. For each classroom i,

- If i is served, we skip it.
- Otherwise, we assign the first unused mobile diner j to serve the classroom. We will place the diner j at $x = x_i + d_j$.

Algorithm

```
# Arguments:
# - C : a list of integers indicating x coordinate for each classroom. The indices
are 1-based.
# - N : an integer indicating the number of classrooms
# - D : a list of integers indicating delicious rate for each mobile diners. The
indices are also 1-based.
# - M : an integer indicating the number of mobile diners
def arrange(C, N, D, M):
    rightBoundary = C[1] - 1
    k = 0
    # O(N) time in total
    for i in [1, N]:
        # O(1) time for updating the boundary
        if C[i] > rightBoundary:
            if k < M:
                k += 1
                rightBoundary = C[i] + 2 * D[k]
            else:
                return INF # Impossible
    return k
```

Prove of Correctness

Claim 1 There are no two different mobile diners serving one classroom.
Suppose there is an optimal strategy OPT that some mobile diner j and $j+1$ serving the same classroom i. If we move $j+1$ a little bit right to make diner it just cannot serve i and acquire another OPT'. We can observe that every classroom is satisfied and the number of mobile diners needed in OPT' is no more that that in OPT.

Claim 2 For each mobile diner j, there is a corresponding classroom i that i is on the left boundary of j.
Suppose there is an optimal strategy OPT that there are some classroom i and mobile diner j such that i is the leftmost classroom served by j but i is not on the boundary of j. If we move j a little bit right to make i exactly on the left boundary of j and acquire another OPT'. We can see that all classrooms are still satisfied and the number of mobile diners needed in OPT' is no more that that in OPT.

(4)

We can solve the task with Dynamic Programming.
Unlike what we do in subproblem (2) and (3) that we tried to align the left boundary of a mobile diner to some classroom, I'm going to align the right boundary to a classroom.

Algorithm First of all, we need to design the DP table

$$dp[n][m] := \begin{array}{l}\text{the minimum number of mobile diners used to satisfy}\\ \text{the first } n \text{ classrooms with only the first } m \text{ classrooms}\\ \text{such that the } n\text{-th classroom lies on the right}\\ \text{boundary of the last used mobile diner}\end{array}$$

The size of the DP table is $O(N \times M)$.

Secondly, we define the recurrence relation.

$$dp[n][m] = \begin{cases} 0 & \text{if } n = 0 \\ \infty & \text{if } n > 0 \text{ and } m = 0 \\ \min \begin{cases} dp[n'][m-1] + 1 & \text{Select the } m\text{-th mobile diner} \\ dp[n][m-1] & \text{Discard the } m\text{-th mobile diner} \end{cases} \end{cases}$$

1. When $n = 0$, we don't have to place any mobile diners.
2. When $n > 0$ and $m = 0$, it's impossible to place mobile diners to satisfy all the students.
3. For the rest situations, we can consider two cases:
   - **Select the m-th mobile diner.** As mentioned above, we'll align the right boundary of the mobile car to the n-th classroom. In this part, $n' < n$ is the largest index of classroom is not covered by the mobile diner.
   - **Discard the m-th mobile diner.**

The time complexity for one calculating one recurrence takes $O(N)$ time if we calculate n' in naive way. Thus, to overcome the bottleneck, we can build another table so that we can access n' in constant time. An implementation is provided below:

```
# Arguments are same as subproblem (3)
def arrange(C, N, D, M):
    # Preprocess n'
    left = an empty two dimension array
    for j in [1, M]:
        r = 1
        for l in [0, N]:
            while r <= N and C[r] <= C[i + 1] + 2 * D[j]:
                left[r][j] = l
                r += 1
            if r > N:
                break

    # Define the recurrence relation
    dp = an empty two dimension array
    def DP(n, m):
        if dp[n][m] is not empty:
            return dp[n][m]
        elif n == 0:
            return dp[n][m] = 0
        elif m == 0:
            return dp[n][m] = INF
        else:
            return dp[n][m] = max(DP(left[n][m], m - 1) + 1,
                                  DP(n, m - 1))

    return DP(N, M)
```

In this way, the preprocessing stage takes $O(NM)$ time. As for DP part, there are total $O(NM)$ states and we need $O(1)$ time to calculate each state. Thus, total time complexity is

$$O(NM) + O(NM \times 1) = O(NM)$$

Prove of DP Properties

Optimal Substructure For $dp[n][m]$, if m-th mobile diner is not selected, it comes directly from $dp[n][m-1]$. If m-th mobile diner is selected, by claim (1) in subproblem (3), we can see that we should place it at some n' that is not covered by m.

Overlapping Subproblems From the recurrence relation, we can see that there might be multiple n-s having the same $n' = left[n][m]$ accessing the same $dp[n'][m-1]$.

Problem 7 - Rainbow Rarity Rally (Hand-Written) (20 points)

Definition 0.1. Sequence: $\{a_i\}_{i=l}^{r}$ denotes a sequence $\{a_l, a_{l+1}, \cdots, a_{r-1}, a_r\}$.

$\text{rev}(\{a_i\}_{i=l}^{r}) := \{a_r, a_{r-1}, \cdots, a_{l+1}, a_l\}$

$\{a_i\}_{i=1}^{n} + \{b_j\}_{j=1}^{m} := \{a_1, a_2, \cdots, a_{n-1}, a_n, b_1, b_2, \cdots, b_{m-1}, b_m\}$

Task 1, 2

Intuition

Let's find some properties of a route first.

Property 0.1. A route is an integer sequence a of length $N+2$. $a_0 = a_{N+1} = 0$ and each $i \in \{1, 2, \cdots, N\}$ appears in a exactly once.
$\Leftrightarrow a_0 = a_{N+1} = 0, \{a_i\}_{i=1}^{N}$ is a permutation of $\{i\}_{i=1}^{N}$.

Property 0.2. Let q be the index of N in a. $\forall i \in [0, q-1], y_{a_i} < y_{a_{i+1}}$. $\forall i \in [q, N], y_{a_i} > y_{a_{i+1}}$.

Since $y_i < y_{i+1} \forall i \in \{0, 1, \cdots, N-1\}$,

$(\forall i \in [0, q-1], y_{a_i} < y_{a_{i+1}}) \Leftrightarrow (\forall i \in [0, q-1], a_i < a_{i+1})$

$(\forall i \in [q, N], y_{a_i} > y_{a_{i+1}}) \Leftrightarrow (\forall i \in [q, N], a_i > a_{i+1})$.

Let $A = \{a_i\}_{i=0}^{q-1}$, $B = \{a_i\}_{i=q}^{N+1}$. A is an increasing sequence and B is an decreasing sequence. $A \setminus \{0\}, B \setminus \{0\}$ form a partition of the set $\{1, 2, \cdots, N\}$. Let B' be the reverse of B, it's also an increasing sequence.

Let 2^S be the set of all subsets of S. $\{A, B\} \in 2^S$ is a partition of S if $A \cap B = \emptyset$ and $A \cup B = S$. (We only consider partitions consist of two (possibly empty) subsets here). Let $P(S)$ be the set of all partitions of S.

Let $S(N) = \{1, 2, \cdots, N-1, N\}$. S_i denotes the k-th smallest number in the set S. We have a bijection between $P(S(N))$ and the set of all routes.

Without loss of generality, $N \in A$.

$$\{A, B\} \longleftrightarrow \{0, A_1, A_2, \cdots, A_{|A|-1}, A_{|A|} = N, B_{|B|}, B_{|B|-1}, \cdots_2, B_1, 0\}$$

Let $L = \{L_1 = 0, L_2, \cdots, L_{|L|} = N\}, R = \{R_1 = 0, R_2, \cdots, R_{|R|}\}$ be two increasing sequence such that $\{\{L_i\}_{i=2}^{|L|}, \{R_j\}_{j=2}^{|R|}\}$ is a partition of $S(N)$.

Let $g(L, R) = \sum_{i=1}^{|L|-1} f(L_i, L_{i+1}) + \sum_{j=1}^{|R|-1} f(R_j, R_{j+1})$.

The smallest amount of time required to finish the race is $\min\{g(L, R) + f(L_{|L|}, R_{|R|}) \mid \text{all possible } L, R\} = \min\{g(L, R) + f(N, R_{|R|}) \mid \text{all possible } L, R\}$.

If we can find the smallest $g(L, R)$ among all possible L, R such that $N \in L$ and $R_{|R|} = j$ for each j, we can answer the question. This gives us the intuition of the definition of subproblems.

Dynamic Programming Algorithm

Without loss of generality, $i > j$.

$$S(i,j) := \{(L = \{0\} + A, R = \{0\} + B) \mid A_{|A|} = i, B_{|B|} = j, \{A, B\} \in P(S(i))\}$$

$$DP(i,j) := \min\{g(L,R) \mid (L,R) \in S(i,j)\}$$

By taking out a term from $g(L,R)$ we, get

$$DP(i,j) = \min\{f(L_{|L|-1}, i) + g(L \setminus \{i\}, R) \mid (L,R) \in S(i,j)\}$$

Case 0. Base case: $DP(1,0) = f(1,0)$

Case 1. If $i - j = 1$, then $L_{|L|-1} \in \{0, 1, \cdots, j-1\}$.

$$\begin{aligned} DP(i,j) &= \min\{\min\{f(k,i) + g(R,L') \mid (R,L') \in S(j,k)\} \mid k \in \{0,1,\cdots,j-1\}\} \\ &= \min\{f(k,i) + DP(j,k) \mid k \in \{0,1,\cdots,j-1\}\} \end{aligned}$$

Case 2. If $i - j > 1$, then $L_{|L|-1} = i - 1$.(Otherwise, $i - 1 \in R \Rightarrow j = i - 1 \Rightarrow i - j = 1 \rightarrow\leftarrow$)

$$\begin{aligned} DP(i,j) &= \min\{f(i-1,i) + g(L',R) \mid (L',R) \in S(i-1,j)\} \\ &= f(i,i-1) + \min\{g(L',R) \mid (L',R) \in S(i-1,j)\} \\ &= f(i,i-1) + DP(i-1,j) \end{aligned}$$

Notice that, if $i-1$ is still larger than j, then

$$DP(i,j) = f(i,i-1) + f(i-1,i-2) + DP(i-2,j)$$

In conclusion, if $i - j > 1$, then

$$DP(i,j) = DP(j+1,j) + \sum_{k=j+1}^{i-1} f(k+1,k)$$

Since we can precalculate the prefix sum of $f(k+1,k)$, we only have to store $DP(j+1,j)$ for each j in our algorithm.

Pseudocode

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int N, C;
vector<int> x, y, c;
vector<int> pf;
// pf[i] =  f(0, 1) + f(1, 2) + ... + f(i - 1, i)
vector<int> d;
```

```cpp
// d[j] = DP(j + 1, j)
int f(int i, int j)
{
    return (x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]);
}
int DP(int i, int j) // i > j
{
    return d[j] + pf[i] - pf[j + 1];
}
int main()
{
    cin >> N >> C;
    const int inf = 2 * C * C * (N + 1);
    x = y = c = pf = vector<int>(N + 1, 0);
    for (int i = 0; i <= N; ++i)
        cin >> x[i] >> y[i];
    for (int i = 1; i <= N; ++i)
        cin >> c[i];
    pf[0] = 0;
    for (int i = 1; i <= N; ++i)
        pf[i] = pf[i - 1] + f(i - 1, i);
    d = vector<int>(N, inf);
    d[0] = f(1, 0); // base case: DP(1, 0) = f(1, 0)
    for (int j = 1; j < N; ++j) {
        int i = j + 1;
        for (int k = 0; k < j; ++k) // calculate DP(j + 1, j)
            d[j] = min(d[j], f(k , i) + DP(j, k));
    }
    int ans = inf;
    for (int j = 0; j < N; ++j)
        ans = min(ans, DP(N, j) + f(j, N));
    cout << ans << endl;
}
```

Task 3, 4

The solution is still under construction. Sorry for the inconvenience.