Homework #2

Solution Released Time: 10/25/2022 22:30
Blue Correction Date: 10/20/2022 12:00
Red Correction Date: 10/14/2022 22:00
Due Time: 2022/10/25 14:20
Contact TAs: `ada-ta@csie.ntu.edu.tw`

Instructions and Announcements

- There are **four programming problems** and **two hand-written problems**.
- **Programming.** The judge system is located at `https://ada-judge.csie.ntu.edu.tw`. Please login and submit your code for the programming problems (i.e., those containing "Programming" in the problem title) by the deadline. NO LATE SUBMISSION IS ALLOWED.
- **Hand-written.** For other problems (also known as the "hand-written problems"), you should upload your answer to **Gradescope** as demonstrated in class. For each sub-problem, please label (on Gradescope) the corresponding pages where your work shows up. NO LATE SUBMISSION IS ALLOWED.
- **Collaboration policy.** Discussions with others are strongly encouraged. However, you should write down your solutions **in your own words**. In addition, for **each and every** problem you have to specify the references (e.g., the Internet URL you consulted with or the people you discussed with) on the first page of your solution to that problem. You may get zero point due to the lack of references.
- **Tips for programming problems.** Since the input files for some programming problems may be large, please add
  - `std::ios_base::sync_with_stdio(false);`
  - `std::cin.tie(nullptr);`

  to the beginning of the main function if you are using **`std::cin`**.

Problem 1 - Sleeping Stages (Programming) (10 points)

Problem Description

Wonling is a legendary gifted Mahjong girl and a super talented student, who receives the Academic Excellence Prize every semester. To keep a clear mindset during all quizzes and exams, she schedules her sleeping time very well.

For a single day, she has N minutes to sleep and she likes to partition her sleeping time into K sleeping stages, each of which is at least one-minute long. Today, she watched a horror movie before sleeping, so today's k-th minute in her sleeping is cursed by a potential scariness value a_k ($\forall k \in [1..N]$).

Also, Wonling will have exactly one nightmare for each sleeping stage tonight. Formally speaking, for each sleeping stage starting from the ℓ-th minute and ending at the r-th minute, there is a nightmare starting from the i-th minute and ending at the j-th minute such that:

- $\ell \le i \le j \le r$
- This nightmare contributes a discomfort value $(r - \ell + 1) \cdot \sum_{k=i}^{j} a_k$ to Wonling.

Even if Wonling is a gifted girl, she cannot control the time when a nightmare occurs. Thus, for each sleeping stage, she considers the worst-case discomfort value $\max_{\ell \le i \le j \le r} \left((r - \ell + 1) \cdot \sum_{k=i}^{j} a_k \right)$. She wants to partition those N minutes of sleeping time into K sleeping stages such that the summation of the worst-case discomfort values for all sleeping stages is **minimized**. Can you help her?

Input

For each testcase, there are two lines. The first line contains two integers N and K, representing how many minutes Wonling can sleep and the number of sleeping stages she would like. The second line contains N integers $a_1, a_2, \cdots, a_N$ representing the potential scariness values.

In other words, the input is presented in the following format:

```
N K
a_1 ... a_N
```

- $1 \le N \le 1000$
- $1 \le K \le \min(N, 500)$
- $-10^6 \le a_k \le 10^6$

Output

For each testcase, output the minimal summation of the worst-case discomfort values.

Test Group 0 (0 %)

- Sample input

Test Group 1 (20 %)

- $K = 2$

Test Group 2 (20 %)

- $1 \le N \le 100$
- $1 \le K \le \min(N, 20)$

Test Group 3 (60 %)

- No additional constraint

Sample Input 1

```
5 2
5 -2 3 -1 4
```

Sample Output 1

```
26
```

Sample Input 2

```
10 3
1 -5 3 -4 3 -5 9 -6 -3 6
```

Sample Output 2

```
45
```

Sample Input 3

```
20 4
3 -1 -4 1 -5 9 -2 6 -5 3
5 -8 9 -7 9 3 -2 3 8 -4
```

Sample Output 3

```
187
```

Notes

- The Sample Input 3 only contains two lines; the second and third lines should be in the same line as below.

  ```
  20 4
  3 -1 -4 1 -5 9 -2 6 -5 3 5 -8 9 -7 9 3 -2 3 8 -4
  ```

  It is reformatted into three lines in this document simply because a single line containing 20 numbers is too long to fit in the page margin.
- In the Sample Input 1, $[5, -2, 3]$ and $[-1, 4]$ is a possible partition to achieve the output value 26.

Solution

Test Group 1

This can be done by maintaining the maximum subarray from two sides of the array.

Test Group 2

Test Group 3 without calculate all possible maximum subarray first.

Test Group 3

Let `dp[i][j]` be the minimum value that we can achieve in the first `i` items with `j` cuts,

`dp[i][j] = min(dp[k][j-1] + maximum_subarray(k + 1, i))` for `k` from `0` to `i-1`

If we calculate the value of all possible `maximum_subarray(k + 1, i)` first, the total time complexity for this problem can be $O(N^2K)$.

Sample Code

```cpp
#include <bits/stdc++.h>
using namespace std;

const long long MAXN = 1000;
const long long MAXK = 200;
const long long MAXC = 1000000;
const long long INF = 1e18;
long long N, K;
long long a[MAXN+1];
long long dp[MAXN+1][MAXK+1];

signed main(){
  ios_base::sync_with_stdio(0), cin.tie(0);
  cin >> N >> K;
  for (int i = 1; i <= N; i++)
    cin >> a[i], dp[i][0] = INF;
  for (int k = 1; k <= K; k++) {
    dp[0][k] = INF;
    for (int r = 1; r <= N; r++) {
      dp[r][k] = INF;
      int cur_max = -INF, cur_sum = 0;
      for (int l = r; l >= 1; l--) {
        cur_sum = cur_sum + a[l];
        cur_max = max(cur_max, cur_sum);
        cur_sum = max(0LL, cur_sum);
        dp[r][k] = min(dp[r][k], dp[l-1][k-1] + cur_max * (r - l + 1));
      }
    }
  }
  cout << dp[N][K] << '\n';
}
```

Test Data

You may download the test data for this problem here.

Problem 2 - Big Chef DanBin (Programming) (10 points)

Problem Description

"I'm Pasta" is one of the most famous restaurants in Alley 118. Today, several groups of people would like to have spaghetti there for dinner. DanBin, the chef of I'm Pasta, wants to go home earlier to watch his favorite VTuber's live streaming. Hence, he plans to arrange the order of dishes such that people can leave the restaurant as early as possible.

The j-th customer from the i-th group orders a dish that needs $a_{i,j}$ minutes for DanBin to prepare and $b_{i,j}$ minutes for this customer to finish. In order to keep the high quality of dishes, DanBin can only prepare one dish at a time, but several people can, of course, have their own dishes at the same time.

According to the regulations in I'm Pasta, DanBin has to prepare dishes for each group one by one. That is, DanBin has to prepare dishes for all customers in the same group consecutively. Besides, all people in the same group will leave the restaurant together when all of them have finished their own dishes.

Can you tell DanBin the minimum **sum** of all customers' time leaving the restaurant if he prepares the dishes in the optimal order?

Input

The first line contains an integer N, indicating the number of groups. Then, N lines follow. The i-th of these lines contains several integers, the first of which is the size of the i-th group, m_i, then the rest $2 \times m_i$ numbers are $a_{i,1}, b_{i,1}, a_{i,2}, b_{i,2}, \ldots, a_{i,m_i}, b_{i,m_i}$, representing the needed time for preparing and enjoying the dishes.

- $1 \leq N \leq 10^5$
- $1 \leq m_i \leq 10^5$
- $1 \leq \sum\limits_{i=1}^{n} m_i \leq 10^5$
- $1 \leq a_j, b_j \leq 10^8$

In other words, the input is presented in the following format:

$$
\begin{array}{l}
N \\
m_1\ a_{1,1}\ b_{1,1}\ \ldots\ a_{1,m_1}\ b_{1,m_1} \\
\vdots \\
m_N\ a_{N,1}\ b_{N,1}\ \ldots\ a_{N,m_N}\ b_{1,m_N}
\end{array}
$$

Output

Output the the minimum **sum** (in minutes) of all customers' time leaving I'm Pasta.

Test Group 0 (0 %)

- Sample input

Test Group 1 (30 %)

- $N = 1$

Test Group 2 (30 %)

- $m_i = 1$

Test Group 3 (40 %)

- No additional constraints

Sample Input 1

```
1
3 4 2 5 8 9 2
```

Sample Output 1

```
60
```

Sample Input 2

```
3
1 4 2
1 5 8
1 9 2
```

Sample Output 2

```
43
```

Sample Input 3

```
3
3 9 2 5 10 1 7
4 7 5 5 1 4 7 2 1
3 10 6 8 5 8 5
```

Sample Output 3

```
373
```

Notes

- In **Sample 1**, preparing the dishes for customer 2, then customer 1 and finally customer 3 would be the best order. These people will leave I'm Pasta in 20 minutes, so the answer is $20 + 20 + 20 = 60$ minutes.
- In **Sample 2**, preparing the dishes for customer 1, then customer 2 and finally customer 3 would be the best order. The first group of people will leave I'm Pasta in 6 minutes, the second group of people will leave I'm Pasta in 17 minutes, and the third group of people will leave I'm Pasta in 20 minutes. As a result, the answer is $6 + 17 + 20 = 43$ minutes.

Solution

Test Group 1

The goal is to minimize the time of the last customer who leaves the store, which is s classic problem and can be solved by sorting the customers with decreasing order of their eating time. This strategy can be proved by swapping two elements that violate the rule and finding out it is a better or equal solution.

Test Group 2

The time a customer leaves the store is his eating time + the sum of cooking time of all previous customers. Based on rearrangement inequality, the answer is to sort the customers with increasing order of their cooking time.

Test Group 3

Combine the previous solution.

Because customers in a group will leave simultaneously, we can handle each group independently as Test Group 1 and treat the whole group as a customer. Therefore, it's the same as Test Group 2 except that each customer has their weight, so the order is decided $\frac{cook\ time}{number\ of\ customers\ in\ the\ group}$ by rather than just cooking time. The proof is the same as Test Group 2, or you can also prove it by swapping two elements that violate the rule and finding out it is a better or equal solution.

Sample Code

```cpp
#include <bits/stdc++.h>
#define int long long
#define pi pair<int, int>
#define pii pair<pi, int>
#define F first
#define S second
using namespace std;
const int N = 1e5 + 10;
inline const bool cmp1(const pi x, const pi y) {
  return x.S > y.S;
}
pi gen(vector<pi> &v) {
  sort(v.begin(), v.end(), cmp1);
  pi ans = {0, 0};
  for (auto &a:v) {
    ans.F += a.F;
    ans.S = max(ans.S, ans.F + a.S);
  }
  ans.S -= ans.F;
  return ans;
}
inline const bool cmp2(const pii x, const pii y) {
  return x.F.F * y.S < y.F.F * x.S;
}
int solve(vector<pii> &v) {
  sort(v.begin(), v.end(), cmp2);
  int ans = 0, t = 0;
  for (auto &a:v) {
    t += a.F.F;
    ans += a.S * (t + a.F.S);
  }
  return ans;
}
vector<pi> g;
vector<pii> q;
signed main () {
  int n, m, a, b;
  cin >> n;
  for (int i = 0; i < n; i++) {
    g.resize(0);
    cin >> m;
    for (int j = 0; j < m; j++) {
      cin >> a >> b;
      g.push_back({a, b});
    }
    q.push_back({gen(g), m});
  }
  cout << solve(q) << '\n';
}
```

Test Data

You may download the test data for this problem here.

Problem 3 - Three-Mouthed Sheep and the String (Programming) (15 points)

Problem Description

Three-mouthed sheep is a strange animal in the ADA kingdom. One day, it receives a string S with N characters as a gift. As a strange animal, it wants to make the string as strange as possible, and it thinks that a string A is stranger than a string B if and only if A is lexicographically smaller than B, which is also a strange definition. A string A is defined as lexicographically smaller than a string B if one of the following conditions holds:

- A is a proper prefix of B; that is, A is a prefix of B and $A \neq B$.
- In the first different characters between A and B, the character in A has a smaller ASCII code than the one in B.

For example, `"abc"` is stranger than `"bac"` because ASCII(`'a'`) $= 97 <$ ASCII(`'b'`) $= 98$.

Because the three-mouthed sheep is strange, other animals in the ADA kingdom don't want it to move and only allow the three-mouthed sheep to perform at most K operations. In each operation, it can swap any two adjacent characters in A. What is the strangest string it can obtain from A if it operates optimally?

Input

The first line is the string S in the length N the three-mouthed sheep initially receives. The second line contains an integer K.

In other words, the input is presented in the following format:

```
S
K
```

- $1 \leq N \leq 5 \times 10^5$
- $0 \leq K \leq \frac{N(N-1)}{2}$
- The string contains only digits and Latin alphabets (including both lowercase and uppercase).

Output

Output the strangest string that the three-mouthed Sheep can achieve by swapping any two adjacent characters for at most K times.

Test Group 0 (0 %)

- Sample input

Test Group 1 (10 %)

- $1 \leq N \leq 10$

Test Group 2 (20 %)

- $1 \leq N \leq 1000$

Test Group 3 (30 %)

- S only contains `'a'`, `'b'`

Test Group 4 (40 %)

- No additional constraint

Sample Input 1

```
dcxxxa
1
```

Sample Output 1

```
cdxxxa
```

Sample Input 2

```
dcxxxa
5
```

Sample Output 2

```
adcxxx
```

Sample Input 3

```
Aa210
5
```

Sample Output 3

```
0A2a1
```

Notes

- The intended solution does not involve any advanced data structure like binary indexed tree, segment tree, treap, etc. If you insist on using these data structures in your code, the TA reserves the right to decline your debugging request.

Solution

Test Group 1

In this test group, you can use any brute force method to run through all permutations of the string, and check which permutation can be reached by using no more than K steps.

Test Group 2

Use greedy algorithm to solve this test group. In each step, choose the smallest character in the string that can be moved to the front in remain times. The total time complexity is $O(N^2)$.

Test Group 3

This test group may help you solve this problem.

Test Group 4

Let c_i denote the number of characters in the string in front of the i-th position that is strictly larger than the character at the i-th position. When a character is moved to the front, because it is the smallest character that can be moved, all characters that are between the front and the original position of the moved character are strictly larger than it, which means that c_i will decrease by 1 if and only if the character at the i-position is smaller than the moved character. The total time complexity is $O(CN)$.

Sample Code

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <deque>
#include <string.h>
#define int long long
using namespace std;
char all[500005];
char ans[500005];
deque < int > where[105];
int tag[105];
int con[105];
int cha[500];
char cha2[500];
signed main()
{
    //freopen("0-01.in","rt",stdin);
  int N,K,i,j,k,now=0,tt;
  scanf("%s",all);
  scanf("%lld",&K);
  N=strlen(all);
  for(i='0';i<='9';i++)
  {
    cha[i]=now;
    cha2[now]=i;
    now++;
  }
  for(i='A';i<='Z';i++)
  {
    cha[i]=now;
    cha2[now]=i;
    now++;
  }
  for(i='a';i<='z';i++)
  {
    cha[i]=now;
    cha2[now]=i;
    now++;
  }
  for(i=0;i<N;i++)
  {
    tt=0;
    for(j=cha[all[i]]+1;j<now;j++) tt+=con[j];
    where[cha[all[i]]].push_back(tt);
    con[cha[all[i]]]++;
  }
  for(i=0;i<N;i++)
  {
    for(j=0;j<now;j++)
    {
      if(!where[j].empty()&&K-(where[j].front()-tag[j])>=0)
      {
        ans[i]=cha2[j];
        K-=(where[j].front()-tag[j]);
        where[j].pop_front();
        for(k=0;k<j;k++) tag[k]++;
        break;
      }
    }
  }
  ans[N]='\0';
  printf("%s\n",ans);
  return 0;
}
```

Test Data

You may download the test data for this problem here.

Problem 4 - Omelet and the LEGO® Tower Vol. 2 (Programming) (15 points)

Problem Description

Recall that Omelet loves playing with LEGO® bricks. This time, he comes up with a new way to play. He places N tower bases on the playground and builds towers on them in a mysterious way.

More formally, there are initially N tower bases on the ground without any bricks on top of them. Then, he will perform the following operation at most K times. Each operation consists of two steps:

1. Pick two numbers, $\ell, r\,(1 \le \ell \le r \le N)$.
2. For all i satisfying $\ell \le i \le r$, put a brick on the top of the i-th tower.

After Omelet finishes building towers, he takes a photo of the playground with all N towers in it. Now he's wondering how many different photos he can collect. Note that all bricks are indistinguishable and two photos are different if and only if there exists at least one $i\,(1 \le i \le N)$ such that the i-th tower has different heights in two photos.

Since the answer might be very large, we only want to know the answer modulo M.

Input

The input contains one line with three numbers N, K, M.

In other words, the input is presented in the following format:

```
N K M
```

- $1 \le N \le 300$
- $1 \le K \le 300$
- $1 \le M \le 10^9 + 7$

Output

Output a number in a line that represents the answer.

Let X be the number all different photos. Then you should output F, where $0 \le F < M$ and $F \equiv X \pmod{M}$.

Test Group 0 (0 %)

- Sample input

Test Group 1 (5 %)

- $K = 1$

Test Group 2 (10 %)

- $1 \le K \le 3$

Test Group 3 (50 %)

- $1 \le K \le 50$

Test Group 4 (35 %)

- No additional constraints

Sample Input 1

```
2 1 100
```

Sample Output 1

```
4
```

Sample Input 2

```
2 2 100
```

Sample Output 2

```
9
```

Sample Input 3

```
228 244 1000000007
```

Sample Output 3

```
502498633
```

Notes

- In the first sample testcase, all possible photos are $(0,0),(0,1),(1,0),(1,1)$.
- In the second sample testcase, all possible photos are $(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)$.

Hints

Here are two techniques that may provide insights to this problem.

Difference Array & Prefix Sum

Let's consider two arrays, A with the length N and B with the length $N+1$, where all entries are 0 initially. If we define the entries of A and B with

$$\begin{cases} B[1] = A[1] \\ B[i] = A[i] - A[i-1], \quad \forall 1 < i \leq N \end{cases}$$

Then, you may notice that increasing $A[l..r]$ by one is equivalent to simply increasing $B[l]$ by one and decreasing $B[r+1]$ by one. As for restoring the entries in A from B, we can add up a prefix of B. Formally, it can be shown that the equation below is equivalent to the definition above.

$$A[i] = \sum_{k=1}^{i} B[k], \quad \forall 1 \leq i \leq N$$

These techniques are called difference array and prefix sum.

Modular Operation

- Avoid redundant modular operations since modular operations can be slow. For example, if you want to find $a+b$ with $0 \leq a, b < M$ holds, you should use `(a + b >= M ? a + b - M : a + b)` instead of `(a + b) % M`.

Reminder

You might not have space for 300^3 integers. To save memory usage, you may want to overwrite some data which will no longer be used in the rest of your algorithms.

Solution

The problem can be simplified as following.

Given a 0 sequence A of length N, you can do the following operation at most K times.

- choose $l, r (1 \leq l \leq r \leq N)$ and increase each of $A_i, i \in [l, r]$ by one.

Count the number of different sequence you can create.

By the hint provided above, you know that it is equivalent to the following problem.

Given a 0 sequence B of length $N + 1$, you can do the following operation at most K times.

- choose $l, r (1 \leq l \leq r \leq N)$. Then increase B_l by one, and decrease B_{r+1} by one.

Count the number of different sequence you can create.

Let's just focus on what properties the sequence B must have after at most K operations.

1. $\sum_i B_i = 0$,since doing operation doesn't change the sum in B.
2. $\sum_{j \leq i} B_j \geq 0$ for all i ,since $\sum_{j \leq i} B_j$ is equal to A_i, it must be non-negative.
3. $\sum_i \max(0, B_i) \leq K$,since after one operation, the sum of positive element in B is increased by at most one.

Proving that all sequences that satisfy above conditions can also be generated from above operation is left as an exercise.

Let $f(Q, W, E)$ be the number of sequence B that satisfy the following.

1. $|B| = Q$
2. $\sum_{j \leq i} B_j \geq 0$ for all i
3. $\sum_i B_i = E$
4. $\sum_i \max(0, B_i) = W$

We know that the answer to the original problem is $\sum_{0 \leq i \leq K} f(N + 1, i, 0)$

By enumerating the last element of B, We know that

$$F(Q, W, E) = \sum_{0 \leq i} F(Q - 1, W - i, E - i) + \sum_{-1 \geq i} F(Q - 1, W, E - i)$$

Which is fast enough for subtask 3. To get full score, you can optimize it by the prefix sum technique.

Construct G, H by following.

$$G(Q, W, E) = G(Q, W - 1, E - 1) + F(Q, W, E)$$

$$H(Q, W, E) = H(Q, W, E - 1) + F(Q, W, E)$$

Then $F(Q, W, E) = G(Q - 1, W, E) + H(Q - 1, W, E) - F(Q - 1, W, E)$

By precalculating $G(Q - 1), H(Q - 1)$, we can calculate $F(Q)$ in $O(K^2)$, which is fast enough for perfect score.

Sample Code

```cpp

#include <bits/stdc++.h>
using namespace std;

int32_t main() {
  ios_base::sync_with_stdio(0), cin.tie(0);

  int N, K, M;

  cin >> N >> K >> M;

  function<void(int&,int)> add = [&](int &a, int b) {
    a = (a + b) % M;
  };

  vector<vector<int>> dp(K + 1, vector<int>(K + 1));

  dp[0][0] = 1;

  for (int i = 0;i <= N;++i) {
    auto tmp1 = dp, tmp2 = dp;
    for (int j = 1;j <= K;++j)
      for (int k = 1;k <= K;++k)
        add(tmp1[j][k], tmp1[j-1][k-1]);
    for (int j = 1;j <= K;++j)
      for (int k = K;k >= 1;--k)
        add(tmp2[j][k-1], tmp2[j][k]);
    for (int j = 0;j <= K;++j)
      for (int k = 0;k <= K;++k) {
        dp[j][k] = (0ll + M + tmp1[j][k] + tmp2[j][k] - dp[j][k]) % M;
      }
  }

  int res = 0;
  for (int k = 0;k <= K;++k)
    add(res, dp[k][0]);

  cout << res << '\n';
}
```

Test Data

You may download the test data for this problem here.

Problem 5 - Generals.io (Hand-Written) (25 points)

Note: In this problem, you are not allowed to write pseudocode. Please explain your algorithm in words.

As a general of CSIE kingdom, you have to defend your country from invasion. Specifically, you anticipate N waves of attack from the enemies, and the i-th wave will arrive on the day T_i. As a part of the defense measures, some military contractors proposed M weapon construction plans. The i-th construction can be done on the day t_i and costs p_i dollars. We need exactly one weapon to successfully defend against one wave of attack. Also, after each wave of attack, the used weapon will be destroyed and is no longer usable under the heavy gunfire.

Now, your goal is to minimize the cost for constructions while keeping the country safe from all hostility. (T_i and t_i are all distinct. $\langle T_i \rangle$ is in an ascending order. Also, you can assume that there exists at least one way to successfully defend against all attacks, which leads to $N \in O(M)$.)

Solution

We can view this set of problems as, for each attack, choosing the corresponding weapon to defend such that the total cost is minimized.

Also, for convenience, attacks will be represented as their arrival time T_i, and plans will be represented as their (time, cost) pair (t_i, p_i).

(a) (2 points)

Given $N = 4,\ M = 7,\ T = [9, 21, 28, 32],\ t = [10, 3, 16, 39, 25, 31, 5],\ p = [4, 5, 6, 1, 2, 7, 3]$, find the optimal **choice**.

Solution

We can choose the 1st, 2nd, 5th and the 7th plans, i.e., $(t_i, p_i) = (10, 4), (3, 5), (25, 2), (5, 3)$.

(b) (7 points)

Design an $O(M \log M)$ greedy algorithm to find the minimum cost and prove its correctness (including the greedy-choice property) and time complexity.

Solution

Concept

Greedy Strategy: starting from the first wave, always choose the cheapest weapon available (hasn't been chosen & can be constructed before the current wave).

Algorithm

1. Sort T_i and (t_i, p_i) together by their date in ascending order.
2. Enumerate from the beginning. If encounters a plan, push its cost p_i into a min-heap. If encounters an attack, take the smallest element from the min-heap.
3. The answer will be the summation over all chosen costs.

Time Complexity

We evaluate the time complexity for each step:

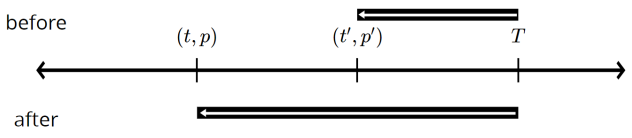
1. Sorting both plans and attacks together costs $O((N+M)\log(N+M)) \overset{N\in O(M)}{=} O(M\log M)$
2. push/pop once on heap for each element. $O(N+M) \times O(\log(N+M)) = O(M\log M)$
3. Summation over N elements. $O(N)$

$O(M\log M) + O(M\log M) + O(N) = O(M\log M)$ in total.

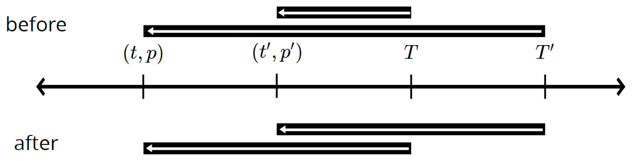
Correctness

We want to prove that for each optimal solution OPT', the solution OPT generated by our algorithm is no worse than OPT'.

- Find chronologically the first attack T such that OPT choose differently from OPT'.
- Assume OPT choose plan (t,p) while OPT' choose plan (t',p'). $t < T$ and $t' < T$ as constrained. $p \le p'$ since p is the minimal available element according to the algorithm.
- If (t,p) is not chosen in OPT':
  - In OPT', replace the choice of (t',p') with (t,p). Since (t,p) is not chosen in OPT' originally and $t < T$ still holds, such replacement does not violate the rules.
  - $cost(OPT'') = cost(OPT') - p' + p \le cost(OPT') \qquad (p \le p')$



- Otherwise, (t,p) is chosen in OPT':
  - Assume OPT' choose (t,p) at attack T'. $T < T'$ since OPT and OPT' have the same choices before T. Thus, $t < T < T'$ and $t' < T < T'$.
  - In OPT', replace the choice for attack T with (t,p) and the choice for attack T' with (t',p').
  - $cost(OPT'') = cost(OPT')$ because it is just swapping.



- In both cases, we can change OPT' such that every choice at and before attack T follows OPT, while the cost does not increase.
- Repeat the process, then OPT' can be transformed into OPT in $O(N)$ iterations.

Thus, $\forall OPT' \in$ optimal_solutions, $cost(OPT) \le cost(OPT')$.

$\implies OPT \in$ optimal_solutions.

Sadly, you find out that your country is too small to accommodate so many constructions. Thus, the number of weapons that are ready for service must not exceed K at any time.

(c) (2 points)

Given $N = 5$, $M = 9$, $K = 2$, $T = [9, 21, 31, 39, 45]$, $t = [25, 10, 16, 43, 50, 5, 28, 32, 3]$, $p = [9, 1, 2, 8, 7, 3, 6, 5, 4]$, find the optimal **choice**.

Solution

We can choose the 2nd, 3rd, 6th, 7th and the 8th plans, i.e., $(t_i, p_i) = (10, 1), (16, 2), (5, 3), (28, 6), (32, 5)$.

(d) (7 points)

Design an $O(MK + M \log M)$ dynamic-programming algorithm to find the minimum cost and prove its correctness and time complexity.

Solution

Concept

$DP(i, j)$ represents the minimum cost on the i-th day(event) with j items in possession. If the i-th event is a plan, transition can be taken from $DP(i-1, j-1)$ and $DP(i-1, j)$. If the j-th event is an attack, transition can be taken from $DP(i-1, j+1)$.

Algorithm

1. Sort T_i and (t_i, p_i) together by their date in ascending order. The i-th event $evt(i)$ means the i-th item in the sorted array.
2. $$DP(i,j) = \begin{cases} 0 & , i = 0 \wedge j = 0 \\ \infty & , j > K \vee j < 0 \vee i \le 0 \\ \min(DP(i-1,j), DP(i-1,j-1) + evt(i).p) & , evt(i) \text{ is a plan} \\ DP(i-1, j+1), & , evt(i) \text{ is an attack} \end{cases}$$
3. The answer will be $DP(N + M, 0)$.

Time Complexity

- As usual, sorting costs $O(M \log M)$.
- With recurrence relation 1 and 2, DP table is restricted to $(i, j) \in [1, N + M] \times [1, K]$. ($O((N + M)K)$ states).
- recurrence relation 3 and 4 both take $O(1)$ to compute.
- Time complexity is $O(M \log M) + O((N + M)K) \times O(1) = O(M \log M + MK)$.

Correctness

Recurrence relation: Consider all the possibilities at i-th event with j weapons ready. If $evt(i)$ is an attack, the only choice is to spend a weapon to defend from it, which corresponds to the subproblem at $i-1$-th event with $j+1$ weapons. If $evt(i)$ is a plan, we can either choose to purchase it or not, which correspond to subproblem at $i-1$-th event with $j-1$ weapons and subproblem at $i-1$-th event with j weapons respectively. Recurrence relation 3 and 4 cover all the above cases.

Base case: We can't have more than K weapons or less than 0 weapon (negative weapons means failing to defend). Also, event's count starts from 1. Recurrence relation 2 sets these invalid states to ∞ so as not to consider them. However, $DP(0, 0)$ is specially set to 0 as the initial state: no weapon is purchased or spent at the very beginning.

(e) (bonus 5 points)

Design an $O(M \log M)$ greedy algorithm to find the minimum cost and prove its correctness (including the greedy-choice property) and time complexity.

Solution

Algorithm

Similar to problem 5-b, for each attack, we always choose the cheapest weapon available at the moment. By "available", it means that the weapon hasn't been chosen & can be constructed before the current wave & choosing it won't violate the land-size restriction at any point. To achieve this goal, we can adopt the following methods in addition to solution 5-b:

1. Use a segment tree τ with range increment and range query for maximum. Whenever we pop an element from the min-heap, find the maximum count *cnt* between the chosen weapon's construction time t_i and the attack time T_i with τ.

   If $cnt < k$, such choice doesn't violate the restriction, so we can increase $[t_i, T_i]$ in τ by 1 and continue the process.

   If $cnt \geq k$, such choice violates the restriction, so we simply drop it and consider the next minimum element in the min-heap.
2. Use self-balancing binary search tree τ instead of min-heap. Besides, after each event, we pop the maximum element from τ until the size of τ is less than or equal to K.

Both of the above methods produce the same results, which will be proven at last.

Segment tree may be more straight forward: always choose the minimum available weapon. However, we should avoid using such advanced data structure (right?), so method 2 is more preferable.

Time Complexity

In additon to solution 5-b, we perform (range query + range increment in segment tree) or (insert + delete in BST) for each element (weapon), both of which cost $O(\log M)$. So the additional time cost will be $O(M) \times O(\log M) = O(M \log M)$.

$O(M \log M + M \log M) = O(M \log M)$ in total.

Correctness

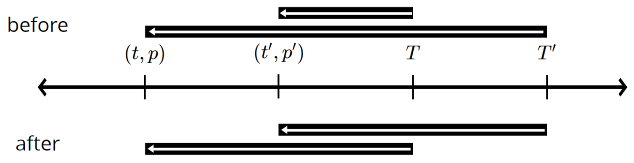
(coverage means occupation of time slots)

The proof is also similar to the proof for problem 5-b:

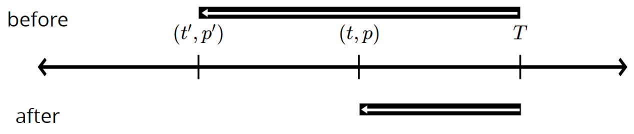
We want to show that for each optimal solution OPT', the solution OPT generated by our algorithm is no worse than OPT'.

- Find chronologically the first attack T such that OPT choose differently from OPT'.
- Assume OPT choose plan (t, p) while OPT' choose plan (t', p'). $t < T$ and $t' < T$ as constrained. $p \leq p'$ since p is the minimal available element according to the algorithm. This time, we divide the problem into 3 cases, where the first two are very similar to problem 5-b's proof.
  - (t, p) is chosen by some attack T' in OPT':
    Swap OPT''s choice for T and T', i.e., T choose (t, p) and T' choose (t', p').
    If $t' < t < T < T'$, the coverage changes from $[t', T] + [t, T']$ to $[t', T'] + [t, T]$
    If $t < t' < T < T'$, the coverage changes from $[t', T] + [t, T']$ to $[t', T'] + [t, T]$
    In both cases, the coverage remains the same. Thus, no violations under the above

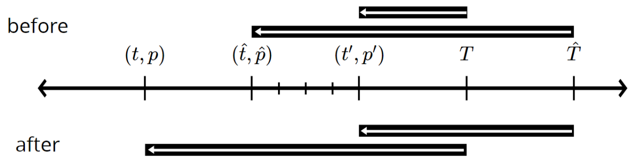
change.
$cost(OPT'') = cost(OPT')$ since it's just swapping.



- (t,p) is not chosen in OPT', and $t' < t$:
  Replace T's choice with (t,p).
  $t' < t < T$, so the coverage changes from $[t',T]$ to $[t,T]$, which means the coverage for $[t',t]$ is reduced by one.
  Reduction in coverage doesn't cause violation, so the above change is valid.
  $cost(OPT'') = cost(OPT') - p' + p \le cost(OPT') \quad (p \le p')$



- (t,p) is not chosen in OPT', and $t < t'$: (this is the most complicated and different one)
  If we replace T's choice by (t,p), the coverage for $[t,t']$ will increase by 1, which may violate the land-side restriction. So, we need to do more than just a replacement.
  Consider the item $(\hat{t},\hat{p})$ immediately chosen after (t,p) by $\hat{T}$ (no other weapons are chosen between t and $\hat{t}$).
  If $(\hat{t},\hat{p}) = (t',p')$, then the coverage increment for $[t,t']$ doesn't violate the rules, since no weapons are chosen within this range. As a result, replacing T's original choice with (t,p) works in this case.
  If $(\hat{t},\hat{p}) \ne (t',p')$, we reassign as the following: $(t < \hat{t} < t' < T < \hat{T})$
    T's choice changes from (t',p') to (t,p).
    $\hat{T}$'s choice changes from $(\hat{t},\hat{p})$ to (t',p').
  $cost(OPT'') = cost(OPT') - \hat{p} + p \le cost(OPT') \quad (p \le \hat{p})$ as p is the minimum available choice.
  The coverage changes from $[t',T] + [\hat{t},\hat{T}]$ to $[t,T] + [t',\hat{T}]$, which means the coverage for $[t,\hat{t}]$ is increased by one.
  Again, the coverage increment for $[t,t']$ doesn't violate the rules, since no weapons are chosen within this range. Thus, the above change is vaild.



- In all cases, we can change OPT' such that every choice at and before attack T follows OPT, while the cost does not increase.
- Repeat the process, then OPT' can be transformed into OPT in $O(N)$ iterations.

Thus, $\forall OPT' \in$ optimal_solutions, $cost(OPT) \le cost(OPT')$.

$\implies OPT \in$ optimal_solutions.

Lastly, we will prove that method 2 in the algorithm section follows the policy that we always choose the cheapest weapon available.

1. **Every weapon in τ must be available at the moment.**

   At time T, if choosing some weapon (p, t) from τ cause constraint violation, there must exists a t' such that, originally, the number of weapons in possession at time t' is K, and $t \le t' \le T$. According to our policy, these K weapons must be cheaper than p. Thus, consider τ at t', these K weapons must be in the τ, and so must (p, t). However, τ contains $K + 1$ weapons in this case and (p, t) is the maximum among τ, so (p, t) must be popped, which contradicts that (p, t) is chosen at time T.

   Thus, choosing any weapon from τ must not cause constraint violation.

2. **Weapons dropped by method 2 must not be chosen under the policy.**

   If there exists a weapon $(p, \_)$ dropped at time t and is chosen under the policy. Consider τ at time t, there are at least K weapons cheaper than p, and they are all available (proven above). According to our policy, these K weapons must be chosen before p, so at time t, the number of weapons in possession is at least K. As a result, weapon $(p, \_)$ is no longer available, a contradiction.

   Thus, dropped weapons must not be chosen under the policy.

From 1 and 2, we know that everytime we choose the minimum element from τ, it is truly the minimum available element at the moment, since only unusable elements are dropped and thus considering only weapons in τ suffice.

Clever as you are, you come up with the idea of requisitioning private residence to resolve the land-size restriction. Now, you can construct as many weapons as you want, but you have to pay the residents 1 dollar per day per weapon as long as the weapon is ready for service.

(f) (7 points)

Design an algorithm to find the minimum cost that runs as fast as possible and prove its correctness and time complexity.

Solution

Concept

Simply add the time cost to each weapon so that we can apply 5-b to solve it.

Algorithm

1. Decrease each weapon's cost by its completion time ($p'_i = p_i - t_i$).
2. Apply 5-b's greedy algorithm to find the minimum cost C
3. The answer will be $C + \sum_{i \in [1,N]} T_i$

Time Complexity

1. $O(M)$
2. $O(M \log M)$
3. $O(N)$

$O(M) + O(M \log M) + O(N) = O(M \log M)$ in total.

Correctness

Assume for attack T_i, we choose weapon $(t_{f(i)}, p_{f(i)})$. Then, the total cost will be

$$\sum_i (p_{f(i)} + (T_i - t_{f(i)})) = \sum_i (p_{f(i)} - t_{f(i)}) + \sum_i T_i$$

Since $\sum\limits_i T_i$ is irrelevant to the choice function f, we only have to minimize $\sum\limits_i (p_{f(i)} - t_{f(i)})$. In problem 5-b, we search for f to minimize $\sum\limits_i p_{f(i)}$ so we can apply 5-b's greedy algorithm under the transformation $(t'_j, p'_j) = (t_j, p_j - t_j)$ to minimize $\sum\limits_i p'_{f(i)} = \sum\limits_i (p_{f(i)} - t_{f(i)})$.

Problem 6 - $(N+1)$-Legged Race (Hand-Written) (25 points)

The ADA Kingdom is about to hold an $(n+1)$-legged race! Everyone in the ADA Kingdom has to participate in this race, and each person has his/her own velocity v_i.

To compete in the race, all participants form several teams. There is **no limit** to the number of people in a team. That is, you can form a team with only one person or even all people in the kingdom. The **speed difference of a group**: δ, is defined as the speed of the fastest person in that group minus the speed of the slowest person in that group (i.e., for a group g, $\delta(g) = \max_{i \in g} v_i - \min_{i \in g} v_i$). Note that a team consisting of only one person has a speed difference of 0. The **total speed difference** is the sum of the speed differences in all groups (i.e., $\sum_g \delta(g)$). You can assume that $v_i \neq v_j$ if $i \neq j$. (i.e. There are no two people having the same velocity.)

Please solve each subproblem based on the above premises. Also, you can assume that basic integer operations ($+$, $-$, $\times$, $\div$) can be done in $O(1)$ time and you don't need to worry about integer overflow.

In all subproblems, We recommend you to explain the algorithm in words. If you want to answer it in pseudo code, make sure it is correct, readable (comment or explain if needed), and **less than 35 lines**.

(a) (4 points)

The king of the ADA kingdom wants to separate the participants into teams for the $(n+1)$-legged race, but he does not know how many groups to divide the people into.

Please design an algorithm which runs in $O(N \log N)$ time to calculate the **minimum total speed difference** for every i from 1 to N, if he separates all people into i groups. Briefly explain the correctness and time complexity of your algorithm.

Solution

First, sort the array v in ascending order.

Let $dif[i] = v[i+1] - v[i]$ for $1 \leq i \leq n-1$, sort the array dif in ascending order.

Let $ans[i]$ = the minimum total speed difference if we separate all people into i groups. Note that $ans[n] = 0$, $ans[i] = ans[i+1] + dif[n-i]$ for $1 \leq i < n$.

Explanation: It is always better if you sort the v array in advance and group people with small velocity difference together. If there is i, j, k such that $i < j < k$ and i and k are in the same group but j is not, then we can move j from its original group into that group, the answer won't be worse.

In the sorted array, if i is in the same group with $i+1$, then $v[i+1] - v[i]$ will be added to the answer. We can thus view $ans[i]$ as : selecting $n-i$ element in the dif array and minimize their sum. Selection will not affect each other. Therefore, selecting the smallest element in dif will be optimal.

The time complexity is bounded by sort, therefore $O(N \log N)$

(b) (5 points)

The king wants to divide all participants into **two unordered groups**: A and B, and conducts an $(n+1)$-legged race such that the total speed difference of these two groups is **not greater than** a given K (i.e., $\delta(A) + \delta(B) \leq K$).

Please design an algorithm that runs in $O(N \log N)$ to calculate how many different ways he can divide the participants. Note that two ways are different if and only if there exists two people a, b such that they are in the same group in one way but in different ones in another. Briefly explain the correctness and time complexity of your algorithm.

Solution

First, sort the array v in ascending order. We will use i to denote the person with ith smallest velocity.

Note that only the person with the highest velocity and the person with the lowest velocity in the group affects the total speed difference.

No matter how we separate people into groups, n will always be the person with the highest velocity in one group, and 1 will always be the person with the lowest velocity in one group.

If $1 < i < j < n$ and i is the person with the lowest velocity in one group, j is the person with the highest velocity in one group, the person k for $1 < k < i$ can only lie in the same group with 1, and the person k for $j < k < n$ can only lie in the same group with n. The person k for $i < k < j$ can choose between two groups arbitrarily. 1 can be paired with either j or n. Therefore, for given i, j, there are 2^{i-j} possibilities if $v[n] + v[j] - v[i] - v[1] \leq K$.

For every i, we can use binary search or two pointer method to find the largest j that meets the constraint : $v[n] + v[j] - v[i] - v[1] \leq K$. If $j > i$, then we can add the answer with $2^1 + 2^2 + \cdots 2^{i-j}$ which is $2^{i-j+1} - 2$. Otherwise, add nothing to answer.

Then we consider cases for $i = j$ and $i = j + 1$. In these cases, there are only one choice for the group a person belongs to. We can verify if it meets the constraint.

The time complexity will be bounded by sort and two-pointer/binary search, which is $O(N \log N)$.

(c) (1 point)

Please list **all different ways** for grouping **four people** into any number of groups without further restriction.

You can use i to denote the i-th person, e.g., $\{\{1, 2, 3, 4\}\}, \{\{1\}, \{2\}, \{3\}, \{4\}\}, \ldots$

Solution

There are total 15 ways to group 4 people.

Separating into 1 group :

$$\{\{1, 2, 3, 4\}\}$$

Separating into 2 groups :

$$\{\{1\}, \{2, 3, 4\}\}, \{\{2\}, \{1, 3, 4\}\}, \{\{3\}, \{1, 2, 4\}\}, \{\{4\}, \{1, 2, 3\}\},$$
$$\{\{1, 2\}, \{3, 4\}\}, \{\{1, 3\}, \{2, 4\}\}, \{\{1, 4\}, \{2, 3\}\}$$

Separating into 3 groups :

$$\{\{1, 2\}, \{3\}, \{4\}\}, \{\{1, 3\}, \{2\}, \{4\}\}, \{\{1, 4\}, \{2\}, \{3\}\},$$
$$\{\{2, 3\}, \{1\}, \{4\}\}, \{\{2, 4\}, \{1\}, \{3\}\}, \{\{3, 4\}, \{1\}, \{2\}\}$$

Separating into 4 groups :

$$\{\{1\}, \{2\}, \{3\}, \{4\}\}$$

(d) (7 points)

Design an algorithm using **dynamic programming** with time complexity $O(N^2)$ to solve the following problem:

Without any other restriction, **how many ways** are there to group N people into any number of groups? We recommend writing your solution in the following form:

1. Define the **subproblem** of dynamic programming and define **each variable** in it. (2 points)
2. Write down the **recurrence relationship** between subproblems. (2 points)
3. Briefly explain the **correctness** and both **time** and **space** complexity of your algorithm. (3 points)

Solution

Let $dp[i][j]$ be the number of possibilities to arrange first i people into j groups. The recurrence relationship is:

$$dp[i][j] = \begin{cases} 0, \text{for i} < \text{j} \\ 1, \text{for i} = \text{j} \\ dp[i-1][j] \times j + dp[i-1][j-1], \text{otherwise} \end{cases}$$

The answer will be

$$\sum_{i=1}^{n} dp[n][i]$$

Note that for a possible group formation with i people and j groups, we can either add the $(i+1)$th person into one of the existing groups or open a new group with only the $(i+1)$th person.

We have $O(N^2)$ subproblems, and solving each of them takes $O(1)$. Therefore, the space complexity is $O(N^2)$, while the time complexity is $O(N^2) \times O(1) = O(N^2)$

(e) (1 point)

There are four people with their velocity $1, 2, 3, 5$, respectively. Assuming the total speed difference cannot be greater than **3**, list **all different ways** of grouping them into any number of groups.

You can use a person's velocity to represent this person, e.g., $\{\{1, 2\}, \{3, 5\}\}, \{\{1\}, \{2\}, \{3\}, \{5\}\}, \ldots$

Solution

There are total 9 ways to group $1, 2, 3, 5$ under the constraint.

Separating into 2 groups :

$$\{\{1\},\{2,3,5\}\},\{\{5\},\{1,2,3\}\},\{\{1,2\},\{3,5\}\}$$

Separating into 3 groups :

$$\{\{1,2\},\{3\},\{5\}\},\{\{1,3\},\{2\},\{5\}\},$$
$$\{\{2,3\},\{1\},\{5\}\},\{\{2,5\},\{1\},\{3\}\},\{\{3,5\},\{1\},\{2\}\}$$

Separating into 4 groups :

$$\{\{1\},\{2\},\{3\},\{5\}\}$$

(f) (7 points)

Design an algorithm using **dynamic programming** with time complexity $O(N^2K)$ to solve the following problem:

The king limits the total speed difference to be not greater than K. Given the velocity v_i of every person, **how many ways** are there to group N people into any number of groups? We recommend write your solution in the form below:

1. Define the **subproblem** of dynamic programming and define **each variable** in it. (2 points)
2. Write down the **recurrence relationship** between subproblems. (2 points)
3. Briefly explain the **correctness** and both **time and space** complexity of your algorithm. (3 points)

Solution

Note that we can't define the subproblem of dynamic programming as it was in (d), since we can't calculate the total speed difference.

After defining $dp[i][j][k]$ as: Using first i people to form j groups with total speed difference k, you found that it is not possible to find a recurrence relationship between them.

Therefore, we need a new way to define a subproblem. Let's consider another way to solve problem (d).

Let's define an "open group" as : a group that needs at least one more person to join it. A "close group" is a group that we cannot add more people into it.

Let $dp[i][j]$ be the number of possibilities to arrange first i people into several groups with j group still open. The recurrence relationship is:

$$dp[i][j]=\begin{cases}1,\text{if } i=j=0\\0,\text{if } i<j \text{ or } i<0\\dp[i-1][j]\times(j+1)+dp[i-1][j-1]+dp[i-1][j+1]\times(j+1),\text{otherwise}\end{cases}$$

$dp[n][0]$ will be the answer to (d)

If there are j groups still open and we want to deal with a new person : i, there are a few cases to handle :

1. Joining person i into one of the j groups without closing the group. There are now j open groups in total.
2. Joining person i into one of the j groups and close the group. There are now $j-1$ open groups in total.
3. Open a group with i alone. There are now $j+1$ open groups in total.
4. Open a group with i alone and close it immediately. There are now j open groups in total.

Summing up these observations, we get the recurrence relationships mentioned above.

We try to add a dimension to the problem to solve (f):

Let $dp[i][j][k]$ = the number of possibilities to arrange first i people into several groups with j group still open and current total group difference is k. Let $d_i = v[i] - v[i-1]$. For $i \geq 2$, the recurrence relationship is:

$$dp[i][j][k] = \begin{cases} 0, \text{if } i < j \text{ or } k < 0 \\ dp[i-1][j][k - d_i \times j] \times (j+1) + \\ dp[i-1][j-1][k - d_i \times (j-1)] + \\ dp[i-1][j+1][k - d_i \times (j+1)] \times (j+1), \text{otherwise} \end{cases}$$

With base case : $dp[1][0][0] = dp[1][1][0] = 1, dp[1][j][k] = 0$ for all other j and k

$\sum_{i=0}^{k} dp[n][0][i]$ will be the answer.

Note that if there are still j groups open and we add the element in sorted order from small to big, d_i, which is $v[i] - v[i-1]$ will be added to the total group difference for j times since every open group needs a person with velocity not smaller than $v[i]$.