Codeforces Round 995 (Div. 3) — Editorial

By **BledDest**, 11 hours ago, translation,

2051D - Counting Pairs

Idea: **fcspartakm**

Tutorial

2051D - Counting Pairs

There is a common trick in problems of the form "count something on segment $[l, r]$": calculate the answer for $[0, r]$, and then subtract the answer for $[0, l - 1]$. We can use this trick in our problem as follows: calculate the number of pairs i, j such that the sum of all other elements is less than $y + 1$, and subtract the number of pairs such that the sum is less than x.

Now we need to solve the following problem: given an array and an integer x, calculate the number of ways to choose i, j ($1 \le i < j \le n$) so that the sum of all elements, except for a_i and a_j, is less than x.

Naive solution (iterate on the pair, calculate the sum of remaining elements) works in $O(n^3)$. It can be improved to $O(n^2)$ if, instead of calculating the sum of remaining elements in $O(n)$, we do it in $O(1)$: if we remove a_i and a_j, the remaining elements sum up to $s - a_i - a_j$, where s is the sum of all elements.

However, $O(n^2)$ is still too slow. For every i, let's try to calculate the number of elements j which "match" it faster. If we sort the array, the answer won't change; but in a sorted array, for every i, all possible values of j form a suffix of the array (if $s - a_i - a_j < x$ and $a_{j+1} \ge a_j$, then $s - a_i - a_{j+1} < x$).

So, for every i, let's find the minimum j' such that $s - a_i - a_{j'} < x$; all $j \ge j'$ are possible "matches" for i. This can be done with two pointers method: when we decrease i, the index j' won't decrease.

Unfortunately, this method has an issue. We need to calculate only pairs where $i < j$, but this method doesn't maintain this constraint. However, this issue can be easily resolved.

First, let's get rid of pairs where $i = j$. To do so, simply calculate the number of indices i such that $s - 2a_i < x$.

Then, let's get rid of pairs where $i > j$. For every such pair, there is a pair with $i < j$ where these two indices are swapped (and vice versa), so we just need to divide the number of pairs by 2.

Now we have a solution working in $O(n \log n)$ for each test case. Instead of two pointers, you can use binary search, the complexity will be the same.

Solution (BledDest)

```
def calcLessThanX(a, x):
    n = len(a)
    s = sum(a)
    j = 0
    ans = 0

    for i in range(n-1, -1, -1):
        while j < n and s - a[i] - a[j] >= x:
            j += 1
        ans += (n - j)

    for i in range(n):
        if s - a[i] - a[i] < x:
            ans -= 1

    return ans // 2

for _ in range(int(input())):
```

```
    n, x, y = map(int, input().split())
    a = list(map(int, input().split()))

    a = sorted(a)
    print(calcLessThanX(a, y+1) - calcLessThanX(a, x))
```

2051E - Best Price

Idea: **BledDest**

Tutorial

2051E - Best Price

First, let's design a solution in $O(n^2)$. We can solve the problem in $O(n \cdot maxb_i)$, if we iterate on the price p we use, and for every price, calculate the number of trees bought and the number of negative reviews. However, we don't need to check every possible price from 1 to $maxb_i$: let's instead check every integer in the union of a and b (or check every a_i, and then check every b_i).

Why is it always optimal? Suppose some integer price p which is not present in the union of a and b is optimal. Then, if we use $p + 1$ instead of p, the status of each customer will be the same, but we will get more money for each tree we sell. So, it is enough to check the elements of a and the elements of b as possible prices.

This works in $O(n^2)$, we need to speed it up. I will explain two different methods that allow to check every price faster.

Event processing (or sweep line):

Shortly, we process all possible prices in ascending order, and when we go from one price to the next, we update the customers which no longer want to buy a tree with a new price, and the customers which will leave a negative review if the price is increased.

One of the ways to implement it is as follows. For every customer, create two "events" of the type "when price exceeds a_i, the customer will leave a negative review" and "when price exceeds b_i, the customer will no longer buy a tree and leave a negative review". These events can be implemented as pairs of integers $(a_i, 1)$ and $(b_i, 2)$.

Then, we can sort the events and process them from left to right in sorted order, maintaining the number of trees and negative reviews. When we process the event with price p, the change it makes will come into effect only when the price exceeds p, so we **should first update the answer, then apply the change from the event**. Furthermore, all events with the same price value should be processed at the same time (so if there are multiple events with the same price value, you don't update the answer after processing only several of them). All of this is a bit complicated to implement, that's why I would like to show you an

Alternative approach:

For every price p, we need to calculate two values:

- the number of trees bought, i.e. the number of customers i such that $b_i \geq p$;
- the number of negative reviews, i.e. the number of customers i such that $a_i < p \leq b_i$.

The first one can be calculated in $O(\log n)$ with binary search, if we sort the array b. The second one is a bit trickier.

Let's calculate it as follows: take the number of trees bought, and then subtract the number of trees bought without a negative review (which is the number of customers i such that $a_i \geq p$). If we sort both arrays a and b, this value can also be processed in $O(\log n)$ with binary search. So, we spend $O(\log n)$ time to check one possible price, and the number of different prices we have to check is up to $2n$, so this solution works in $O(n \log n)$.

Solution (Neon)

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  ios::sync_with_stdio(false); cin.tie(0);
  int t;
  cin >> t;
```

```
  while (t--) {
    int n, k;
    cin >> n >> k;
    vector<int> a(n), b(n);
    for (auto &x : a) cin >> x;
    for (auto &x : b) cin >> x;
    vector<pair<int, int>> ev;
    for (int i = 0; i < n; ++i) {
      ev.emplace_back(a[i], 1);
      ev.emplace_back(b[i], 2);
    }
    sort(ev.begin(), ev.end());
    long long ans = 0;
    int cnt = n, bad = 0;
    for (int i = 0; i < 2 * n;) {
      auto [x, y] = ev[i];
      if (bad <= k) ans = max(ans, x * 1LL * cnt);
      while (i < 2 * n && ev[i].first == x) {
        bad += (ev[i].second == 1);
        bad -= (ev[i].second == 2);
        cnt -= (ev[i].second == 2);
        ++i;
      }
    }
    cout << ans << '\n';
  }
}
```

2051F - Joker

Idea: **BledDest**

Tutorial

2051F - Joker

Let's represent the positions where the joker can be as a set of non-overlapping segments $[l_1, r_1]$, $[l_2, r_2]$, ... Let's consider what happens to the segment $[l, r]$ after applying the i-th operation:

- if $a_i < l$, the possible positions segment becomes $[l - 1, r]$ (since moving the a_i-th card to the front does not change the joker's positions, while moving it to the back shifts the positions up by 1);
- if $a_i > r$, the possible positions segment becomes $[l, r + 1]$ (since moving the a_i-th card to the front shifts the positions down by 1, while moving it to the back does not change the joker's positions);
- if $l \le a_i \le r$, let's consider 3 subsegments where the joker can be located:
  - positions from the subsegment $[l, a_i - 1]$ moves to $[l, a_i]$ (similarly to the case $a_i > r$);
  - positions from the subsegment $[a_i + 1, r]$ moves to $[a_i, r]$ (similarly to the case $a_i < l$);
  - the joker from position a_i moves to one of two positions: 1 or n.

  Thus, in this case, the segment $[l, r]$ remains, but we need to add two new segments ($[1, 1]$ and $[n, n]$) to the set.

Note that when $l = r = a_i$, the current segment disappears.

At first glance, it seems that this solution works in $O(nq)$, since the number of segments can be $O(n)$, and we need to update each of them. However, it is not difficult to notice that there cannot be more than 3 segments. Specifically: the initial segment $[m, m]$, which expands to the left and right, the segment $[1, 1]$, which expands only to the right, and the segment $[n, n]$, which expands only to the left.

Solution (Neon)

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int t;
  cin >> t;
  while (t--) {
    int n, m, q;
    cin >> n >> m >> q;
    vector<pair<int, int>> segs({{1, -q}, {m, m}, {n + q + 1, n}});
    while (q--) {
      int x;
      cin >> x;
      bool ins = false;
      for (auto& [l, r] : segs) {
        if (x < l) l = max(1, l - 1);
        else if (x > r) r = min(n, r + 1);
        else {
          ins = true;
          if (l == r) l = n + q, r = -q;
        }
      }
      if (ins) {
        segs[0] = {1, max(segs[0].second, 1)};
        segs[2] = {min(segs[2].first, n), n};
      }
      int lf = 0, rg = -1, ans = 0;
      for (auto [l, r] : segs) {
        if (l > r) continue;
        if (l > rg) {
          ans += max(0, rg - lf + 1);
          lf = l; rg = r;
        }
        rg = max(rg, r);
      }
      ans += max(0, rg - lf + 1);
      cout << ans << ' ';
    }
    cout << '\n';
  }
}
```

2051G - Snakes

Idea: **BledDest**

Tutorial

2051G - Snakes

Note that when you place snakes on the strip in some order, they form some permutation. And when you fix that permutation, you can place them greedily.

In other words, when you know in what order you'll place snakes, it's always optimal to place them as close to each other as possible. Since the bigger the initial distance — the bigger the resulting distance of the farthest snake (or the bigger the final score). We can even calculate that final score precisely: it's equal to
$1 + \texttt{sum of distances} + \texttt{number of times the last snake enlarges}$.

So, we can solve the task in two steps. First, let's calculate $\text{minDist}[i][j]$ — the minimum possible distance between snakes i and j if we plan to place snake j right after snake i. Suppose the initial *gap* between these snakes is x. Let's skim through all events:

- each time the i-th snake enlarges, our gap decreases, or $x' = x - 1$.
- each time the j-th snake shrinks, our gap increases, or $x' = x + 1$.
- if at any moment x' becomes negative, then we lose. In other words, we needed bigger initial x.

We can rephrase what happens more formally: for each event i let $e_i = 1$ if x increases, $e_i = -1$ if x decreases or 0 otherwise. Then after the i-th event the current gap will be equal to $x' = x + \sum_{j=1}^{i} e_j$.

The following inequality should hold for each i: $x + \sum_{j=1}^{i} e_j \ge 0$ or $x \ge -\sum_{j=1}^{i} e_j$. So, if we will find the minimum $\min\limits_{1 \le i \le q} \sum_{j=1}^{i} e_j$ then we can set the initial distance to this minimum gap plus one, or $\text{minDist}[i][j] = -\min\limits_{1 \le i \le q} \sum_{j=1}^{i} e_j + 1$.

Now we know the minimum distances between neighboring snakes, so we can find the optimal order. Let's do it with bitmask dp $d[mask][lst]$, since all we need to know in each state is the set of already placed snakes $mask$ and the last snake lst. Transitions are straightforward: let's just choose the next snake to place and place it at distance minDist.

The initial states are $d[2^i][i] = 1$ for each i. The answer is $\min\limits_{1 \le i \le n} d[2^n - 1][i] + \texttt{number of times snake } i \texttt{ enlarges}$, i. e. we just choose the last snake.

The time complexity is $O(n^2 q)$ for the first part (or $O(nq)$ if written more optimally) plus $O(2^n n^2)$ for the second part.

Solution (adedalic)

```cpp
#include<bits/stdc++.h>

using namespace std;

#define fore(i, l, r) for(int i = int(l); i < int(r); i++)
#define sz(a) int((a).size())

const int INF = int(1e9);

int n, q;
vector<int> id, ch;

bool read() {
    if (!(cin >> n >> q))
        return false;
    id.resize(q);
    ch.resize(q);
    fore (i, 0, q) {
        char c;
        cin >> id[i] >> c;
        id[i]--;
        ch[i] = c == '+' ? 1 : -1;
    }
    return true;
}

int getDist(int s, int t) {
    int pSum = 0, cMin = 0;
    fore (e, 0, q) {
        if (id[e] == t)
            pSum += ch[e] < 0;
        if (id[e] == s)
            pSum -= ch[e] > 0;
        cMin = min(cMin, pSum);
    }
```

```
    return -cMin + 1;
}

inline void solve() {
    vector<vector<int>> minDist(n, vector<int>(n, INF));

    fore (i, 0, n) fore (j, 0, n)
        minDist[i][j] = getDist(i, j);

    vector<int> len(n, 0);
    fore (e, 0, q)
        len[id[e]] += ch[e] > 0;

    vector< vector<int> > d(1 << n, vector<int>(n, INF));
    fore (i, 0, n)
        d[1 << i][i] = 1;

    fore (mask, 1, 1 << n) fore (lst, 0, n) {
        if (d[mask][lst] == INF)
            continue;
        fore (nxt, 0, n) {
            if ((mask >> nxt) & 1)
                continue;
            int nmask = mask | (1 << nxt);
            d[nmask][nxt] = min(d[nmask][nxt], d[mask][lst] + minDist[lst][nxt]);
        }
    }
    int ans = INF;
    fore (lst, 0, n)
        ans = min(ans, d[(1 << n) - 1][lst] + len[lst]);
    cout << ans << endl;
}

int main() {
#ifdef _DEBUG
    freopen("input.txt", "r", stdin);
    int tt = clock();
#endif
    ios_base::sync_with_stdio(false);

    if(read()) {
        solve();

#ifdef _DEBUG
        cerr << "TIME = " << clock() - tt << endl;
        tt = clock();
#endif
    }
    return 0;
}
```