Editorial

Competitive Programming

Coding Club IIT Jammu

ICE BREAKER

Let's tackle the scenario where Sangeet and Nageshwar start at the same coordinate as the bounty basket. In this setup, let's denote the total distance from each bounty element to the bounty basket as T, which is twice the sum of distances from each bounty element to the bounty basket .

Now, if Sangeet chooses the i-th bounty element and Nageshwar selects the j-th bounty element (where i ≠ j), the total distance they need to travel can be calculated as follows:

1. We start with T, which already accounts for twice the distance from each bounty element to the bounty basket.
2. If Sangeet picks the i-th bounty element, we add the distance from Sangeet to the i-th bounty element and subtract the distance from the i-th bounty element to the bounty basket from T. This adjustment represents the distance saved by Sangeet picking up the i-th bounty element directly.
3. Similarly, if Nageshwar chooses the j-th bounty element, we add the distance from Nageshwar to the j-th bounty element and subtract the distance from the j-th bounty element to the bounty basket from T. This adjustment represents the distance saved by Nageshwar picking up the j-th bounty element directly.

So, the total distance they need to travel is T + distance(Sangeet, bounty element i) - distance(bounty element i, bounty basket) + distance(Nageshwar, bounty element j) - distance(bounty element j, bounty basket).

To optimize this calculation:

- We can construct two arrays: ai and bj, where ai represents the difference between the distance from Sangeet to the i-th bounty element and the distance from the bounty basket to the i-th bounty element, and bj represents the same for Nageshwar.
- We can then choose i naively and find the minimum value in bj where j ≠ i. This can be precalculated as opt1 for Nageshwar. Additionally, we find the second optimal value opt2 for Nageshwar.
- If the chosen i is equal to opt1, Nageshwar will go for opt2; otherwise, Nageshwar will choose opt1.

Lastly, we need to handle special cases where either all bounty elements are taken by Sangeet or Nageshwar, ensuring our solution covers all possibilities.

Code

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

```cpp
using namespace std;
typedef long long ll;
const int N = 1e6 + 5;

int Kx, Ky, Cx, Cy, Tx, Ty, n, x, y;
double pre[N], suff[N], add[N], all;
double dist(int x, int y, int a, int b) {
    return sqrt((ll) (x - a) * (x - a) + (ll) (y - b) * (y - b));
}

int main() {

    scanf("%d %d %d %d %d %d", &Kx, &Ky, &Cx, &Cy, &Tx, &Ty);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d %d", &x, &y);
        double added = dist(Tx, Ty, x, y);
        all += added * 2;
        suff[i] = pre[i] = dist(Cx, Cy, x, y) - added;
        add[i] = dist(Kx, Ky, x, y) - added;
    }
    pre[0] = suff[n+1] = 1e18 + 5;
    // Prefix and suffix minimum
    for (int i = 1; i <= n; i++) {
        pre[i] = min(pre[i-1], pre[i]);
    }
    for (int i = n; i >= 1; i--) {
        suff[i] = min(suff[i+1], suff[i]);
    }
    // Compute answer
    double ans = suff[1] + all;

    for (int i = 1; i <= n; i++) {
        ans = min(ans, all + min(0.0, min(pre[i-1], suff[i+1])) + add[i]);
    }

    // Output
    printf("%.12lf\n", ans);
    return 0;
}
```

***Due to a technical issue on HackerRank affecting result accuracy, all participating teams for this problem statement (Ice Breaker) will receive full points as a fair resolution.**

The Treasure Hunt

Jethalal and Babitaji, two brilliant minds at IIT Jammu, faced an intriguing puzzle. Their friend Iyer provided a challenge: find the maximum possible value of the sum of the highest subarray scores for both Jethalal and Babitaji. The twist was that they could swap puzzle pieces at any index to optimize their solutions.

To tackle this problem, a dynamic programming approach was employed. Let's delve into the editorial and explore the thought process behind this solution.

Problem Analysis:

The goal is to select two segments (one for each friend) with the maximum sum, considering the possibility of swapping elements at any index.

Dynamic Programming Solution:

1. Defining the States:

- Use dynamic programming with three parameters: dpi, f1, and f2.
- dpi represents the number of positions considered (ranging from 0 to N).
- f1 and f2 are flags indicating the state of each friend's array (before, inside, or after the chosen segment).

2. Transitions:

- Transitions are straightforward. At each step, there's a choice to take ai for the first array and bi for the second (or vice versa).
- Flags can change states, but transitions are limited to moving from the current state to subsequent states. For example, moving from "segment ended" to "inside the segment" is allowed.

3. Handling Segments:

- If the current state of a flag is "inside the segment," add the corresponding value to the dynamic programming result.

4. Optimization:

- Since elements can be swapped, simplify the implementation by ensuring ai ≥ bi.
- This ensures that whenever an index belongs to one of the two chosen segments, it's clear which element to select.

5. Time Complexity:

- Achieve an efficient solution with O(N) complexity.

Implementation:

Implementing this dynamic programming approach allows Jethalal and Babitaji to efficiently explore the solution space, considering various combinations of segments and swaps.

In conclusion, the duo can confidently approach the puzzle with a dynamic programming strategy, optimizing their chances of finding the highest possible sum. The editorial provides a clear roadmap for solving the problem and highlights the elegance of the chosen approach.

Code :

```
#include <iostream>

#include <vector>

#include <fstream>

#include <iomanip>

// Check for division by zero

using namespace std;

double abs(double number){

    if(number < 0){

        number *= -1 ;

    }

    return number ;

}
```

```
void print(std::vector<std::vector<double>> &A) {
    int n = A.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n + 1; j++) {
            std::cout << A[i][j] << "\t";
            if (j == n - 1) {
                std::cout << "| ";
            }
        }
        std::cout << "\n";
    }
    std::cout << std::endl;
}
// Function to perform Gaussian elimination
void gaussianElimination(std::vector<std::vector<double>> &A) {
    int numRows = A.size();
    int numCols = A[0].size();
    double g = 0;
    double h = 0;
    for (int i = 0; i < numRows; ++i) {
        // Find the pivot row (maximum absolute value in the current column)
        int pivotRow = i;
        for (int k = i + 1; k < numRows; ++k) {
            g = ((float)A[k][i]) ;
            h = ((float)A[pivotRow][i]) ;
            if (abs(g) > abs(h)) {

                pivotRow = k;

            }

        }

        // Swap the current row with the pivot row if needed

        if (pivotRow != i) {

            swap(A[i], A[pivotRow]);

        }

        // Make the diagonal element 1

        double divisor = A[i][i];

        // Check for division by zero

if (abs(divisor) < 1e-10) {

    // cerr << "Error: Division by zero encountered. No unique solution exists.\n";

    return;

}

        for (int j = i; j < numCols; ++j) {

            A[i][j] /= divisor;

        }

        // Make other rows' elements in the current column zero

        for (int k = 0; k < numRows; ++k) {

            if (k != i) {

                double factor = A[k][i];

                for (int j = i; j < numCols; ++j) {

                    A[k][j] -= factor * A[i][j];

                }
            }
        }
    }
}
int main() {
    int n, g;
    // // Input the number of equations and variables
    cin >> n;
    Cin >> g;
    // Input coefficients for each equation
    std::vector<std::vector<double>> A(n, std::vector<double>(g + 1));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= g; ++j) {
            Cin >> A[i][j];
        }
    }
    // Solve linear equations using Gaussian elimination
    gaussianElimination(A);
    // Display the solution
    double sum =0 ;
    for (int i = 0; i < n; ++i) {
        sum +=  A[i][g] ;
    }
    Cout << std::fixed << setprecision(4)<<sum<<endl;
    return 0;
```

```
}
```

NP_Hard Problem

Sangeet, with his profound interest and proficiency in mathematics, always seeks out challenging problems. Sumit, his mathematics professor, aware of Sangeet's penchant for difficult math challenges, decided to present him with a particularly intriguing problem.

In this scenario, Sangeet is provided with a list of n numbers, accompanied by two integers a and b. Each element in the list adheres to the condition 1≤qi≤a.

The task is to select two integers, qi and qj, at a time, where 1 ≤ i < j ≤ n. Subsequently, append the maximum integer that divides both chosen numbers to the end of the list and eliminate qi and qj from the list. After precisely b operations, the objective is to determine the highest possible sum of the remaining elements in the list.

To address the repeated elements in the problem, it's essential to note the following strategies:

1. Dealing with Repeated Elements:

A strategy for handling repeated elements is available. Assuming all elements in a are pairwise distinct facilitates the problem-solving process.

2. Optimal Strategy:

Always perform operations on the minimum element.

3. Optimizing the Sequence:

Optimal approach involves choosing k + 1 elements, deleting them, and adding the maximum among them to the sequence.

4. Sorting Misconception:

Although sorting the sequence may suggest choosing a prefix as optimal, it's incorrect. There is a need to fix the strategy.

Solution Approach :

Firstly, consider the scenario where all elements in a are pairwise distinct. The problem can be reframed as dividing the sequence into n k groups to maximize the sum of the maximum element in each group.

Lemma 1:

When k > 0, the group to which the minimum element of the original sequence belongs satisfies Sa > 1. The proof involves showing that selecting a group with Sz > 1 enhances the answer.

Lemma 2:

When k > 0, only one group S exists such that Sz > 1. The proof involves iteratively removing elements of Sa from the sequence, adding gcd(Sa) to the sequence, and decrementing k until k = 0.

Handling Repeated Elements:

For repeated elements, the optimal strategy is to merge them with the same element, as they only decrease the answer by x. This aspect is independent of the previous part, and enumerating the number of operations for repeated elements suffices.

In conclusion, the solution involves enumerating gcd(Sa) to solve it in O(n + m ln m).

Code:

```
// LUOGU_RID: 106504518

#include<bits/stdc++.h>

using namespace std;

const int maxn=1000005;

int T,n,m,bs,k;

int a[maxn],vis[maxn],b[maxn];

long long ans,sum;

long long sumb[maxn];

int main(){

    scanf("%d",&T);

    while(T--){

        scanf("%d%d%d",&n,&m,&k),ans=sum=bs=0;
        for(int i=1;i<=n;i++){
            scanf("%d",&a[i]),sum+=a[i];
            if(vis[a[i]])
                b[++bs]=a[i];
            vis[a[i]]=1;
        }
        sort(b+1,b+1+bs);
        for(int i=1;i<=bs;i++)
            sumb[i]=sumb[i-1]+b[i];
        if(k<=bs)
            ans=sum-sumb[k];
        for(int i=1;i<=m;i++){
            int tot=0;
            long long now=i;
            for(int j=i;j<=m;j+=i)
                if(vis[j]){
                    tot++,now-=j;
                    if(tot<=k+1&&k+1-tot<=bs)
                        ans=max(ans,sum-sumb[k+1-tot]+now);
                }
        }
        printf("%lld\n",ans);
        for(int i=1;i<=m;i++)
            vis[i]=0;
    }
```

```
    return 0;

}
```