# Respostas da Maratona

Ordenar 1

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

int main()
{
    int n, diff, a;
    cin >> n >> diff;  // Lê os valores de N e K (diff).

    int ans = 0;

    int arr[n];  // Declaração de array de tamanho fixo com
    variável, permitido em C++ mas menos eficiente que
    `vector<int>`.

    int i = 0;
    for (i = 0; i < n; i++)  // Lê os N elementos e armazena no
    array.
    {
        cin >> a;
        arr[i] = a;
    }

    sort(arr, arr + n);  // Ordena o array para permitir busca
    binária.

    for (int j = 0; j < i; j++)  // Itera sobre os elementos do
    array.
    {
        if (binary_search(arr, arr + n, arr[j] + diff)) ans++;  //
        Procura `arr[j] + diff` no array.
    }
```

```
    cout << ans << endl;  // Imprime a contagem de pares
    encontrados.
    return 0;
}
```

Ordenar 2

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

long long int i, j; // Variáveis globais desnecessárias, podem ser
removidas.

int main()
{
    long long int t;
    cin >> t;  // Lê o número de casos de teste.

    while (t--) // Loop para processar cada caso de teste.
    {
        long long int n;
        cin >> n;  // Lê o número de contas bancárias.
        cin.ignore(100, '\n');  // Limpa o buffer para evitar
        problemas com getline().

        map<string, long long int> m;  // Usa um map para armalong
        long intenar e contar a frequência de cada conta.
        map<string, long long int>::iterator it;

        while (n--)  // Lê todas as contas bancárias.
        {
            string s;
            getline(cin, s);  // Lê a linha completa da conta
            bancária.
            m[s]++;  // Incrementa a contagem da conta no map.
        }

        // Percorre o map ordenado e imprime as contas com suas
        frequências.
        for (it = m.begin(); it != m.end(); it++)
        {
            cout << it->first << " " << it->second << endl;
        }
    }

    return 0;
}
```

Ordenar 3

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

int main() {
    int N;
    cin >> N;  // Lê o número de elementos.

    vector<int> arr(N);  // Declara um vetor para armazenar os
    números.
    for (int i = 0; i < N; i++) cin >> arr[i];  // Lê os N
    elementos do vetor.

    sort(arr.begin(), arr.end());  // Ordena o vetor em ordem
    crescente.

    int ans = 1;  // Pelo menos um número distinto existe.

    for (int i = 1; i < N; i++) {
        /*
         * Se o número atual for diferente do anterior,
         * significa que encontramos um novo valor distinto.
         * Incrementamos a contagem.
         */
        ans += (arr[i] != arr[i - 1]);
    }

    cout << ans << endl;  // Imprime o número de valores distintos.
    return 0;
}
```

Ordenar 4

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

vector<int> collectingnumbersII(int n, int m, vector<int>& values,
                                vector<vector<int>>& swaps)
{
    // Tornar o array 1-indexed (inserindo um valor fictício na
    posição 0).
    values.insert(values.begin(), 0);

    vector<int> res;  // Vetor para armazenar os resultados após
    cada swap.
    vector<int> position(n + 1);  // Vetor que armazena a posição
    de cada valor.

    // Armazena a posição de cada número no array original.
    for (int i = 1; i <= n; i++)
        position[values[i]] = i;

    // Calcula o número inicial de rodadas
    int count = 1;
    for (int i = 1; i < n; i++)
        // Se o número `i` aparece depois do número `i+1`, então há
        uma inversão.
        count += (position[i] > position[i + 1]);

    // Conjunto para armazenar os pares de valores cujas posições
    serão atualizadas.
    set<pair<int, int>> updatedPairs;

    // Processa cada operação de troca.
    for (int i = 0; i < m; i++) {
        int l = swaps[i][0], r = swaps[i][1];  // Índices a serem
        trocados.

        // Adiciona ao conjunto os pares de valores que podem ser
        afetados pela troca.
        if (values[l] + 1 <= n)
            updatedPairs.insert({values[l], values[l] + 1});
        if (values[l] - 1 >= 1)
            updatedPairs.insert({values[l] - 1, values[l]});
```

```
        if (values[r] + 1 <= n)
            updatedPairs.insert({values[r], values[r] + 1});
        if (values[r] - 1 >= 1)
            updatedPairs.insert({values[r] - 1, values[r]});

        // Atualiza a contagem antes da troca, removendo inversões
        que podem mudar.
        for (auto swapped : updatedPairs)
            count -= position[swapped.first] >
            position[swapped.second];

        // Realiza a troca no array.
        swap(values[l], values[r]);

        // Atualiza as posições dos valores trocados.
        position[values[l]] = l;
        position[values[r]] = r;

        // Recalcula a contagem após a troca, adicionando novas
        inversões.
        for (auto swapped : updatedPairs)
            count += position[swapped.first] >
            position[swapped.second];

        // Armazena o resultado da contagem após a troca.
        res.push_back(count);

        // Limpa o conjunto para a próxima operação.
        updatedPairs.clear();
    }
    return res;  // Retorna os resultados.
}

// Função principal
int main()
{
    int n = 5, m = 3;  // Tamanho do array e número de operações.
    vector<int> values = { 4, 2, 1, 5, 3 };  // Array inicial.
    vector<vector<int>> swaps = { {2, 3}, {1, 5}, {2, 3} };  //
    Lista de trocas.

    // Chama a função e armazena os resultados.
```

```
    vector<int> res = collectingnumbersII(n, m, values, swaps);

    // Imprime o número de rodadas após cada troca.
    for (auto i : res)
        cout << i << "\n";

    return 0;
}
```

Struct 1

```c
#include <stdio.h>

typedef struct {
    int values[200];  // Vetor de números
    int size;         // Quantidade de números
} Numbers;

// Função para contar quantas vezes todos os números podem ser
divididos por 2
int count_operations(Numbers *nums) {
    int count = 0;

    while (1) {
        for (int i = 0; i < nums->size; i++) {
            if (nums->values[i] % 2 != 0) {
                return count; // Se algum número for ímpar, retorna
                o total de operações
            }
            nums->values[i] /= 2; // Divide todos por 2
        }
        count++; // Incrementa o número de operações realizadas
    }
}

int main() {
    Numbers nums;

    // Leitura da entrada
    scanf("%d", &nums.size);
    for (int i = 0; i < nums.size; i++) {
        scanf("%d", &nums.values[i]);
    }

    // Calcula e imprime o número máximo de operações
    printf("%d\n", count_operations(&nums));

    return 0;
}
```

Struct 2

```c
#include <stdio.h>
#include <stdlib.h>

// Estrutura para representar um ponto no plano com tempo associado
typedef struct {
    int t;  // Tempo (momento em que o ponto é visitado)
    int x;  // Coordenada X
    int y;  // Coordenada Y
} Point;

// Função para calcular a distância de Manhattan entre dois pontos
// A distância de Manhattan é a soma das diferenças absolutas das
coordenadas X e Y
int manhattan(int x1, int y1, int x2, int y2) {
    return abs(x2 - x1) + abs(y2 - y1);
}

int main() {
    int N;
    // Lê o número de pontos a serem verificados
    scanf("%d", &N);

    // Aloca memória dinamicamente para armazenar os pontos
    // Cada ponto contém tempo (t) e coordenadas (x, y)
    Point *points = (Point *)malloc(N * sizeof(Point));

    // Lê os valores de tempo e coordenadas para cada ponto
    for (int i = 0; i < N; i++) {
        scanf("%d %d %d", &points[i].t, &points[i].x,
        &points[i].y);
    }

    // Variável para verificar se o caminho é possível
    // Inicialmente, assume-se que é possível (1 = verdadeiro)
    int feasible = 1;

    // Verifica a viabilidade do trajeto da origem (0, 0) no tempo
    t=0 até o primeiro ponto
    int prev_t = 0;  // Tempo inicial (origem)
    int prev_x = 0;  // Coordenada X inicial (origem)
    int prev_y = 0;  // Coordenada Y inicial (origem)
```

```
// Calcula a distância de Manhattan até o primeiro ponto
int dist = manhattan(prev_x, prev_y, points[0].x, points[0].y);

// Calcula o tempo disponível para chegar ao primeiro ponto
int time = points[0].t - prev_t;

// Verifica se o tempo disponível é suficiente para cobrir a
distância
// Além disso, verifica se a paridade do tempo restante é
válida
// (o tempo excedente deve ser par para permitir idas e vindas)
if (time < dist || (time - dist) % 2 != 0) {
    feasible = 0;  // Caminho impossível
}

// Verifica a viabilidade entre pontos consecutivos
// Só continua se o trajeto até o momento for possível
for (int i = 0; i < N - 1 && feasible; i++) {
    // Atualiza as variáveis para o ponto atual
    prev_t = points[i].t;
    prev_x = points[i].x;
    prev_y = points[i].y;

    // Calcula a distância de Manhattan até o próximo ponto
    dist = manhattan(prev_x, prev_y, points[i + 1].x, points[i
    + 1].y);

    // Calcula o tempo disponível para chegar ao próximo ponto
    time = points[i + 1].t - prev_t;

    // Verifica se o tempo disponível é suficiente e se a
    paridade é válida
    if (time < dist || (time - dist) % 2 != 0) {
        feasible = 0;  // Caminho impossível
        break;  // Sai do loop, pois já foi detectado que é
        impossível
    }
}

// Imprime o resultado: "Yes" se possível, "No" se impossível
printf("%s\n", feasible ? "Yes" : "No");
```

```
    // Libera a memória alocada para os pontos
    free(points);

    return 0;
}
```

Struct 3

```cpp
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;

// Estrutura que representa um nó da árvore de segmentos
persistente
struct tree {
  long long val;       // Valor armazenado no nó
  tree *left, *right; // Ponteiros para os filhos esquerdo e
  direito

  // Construtor da estrutura tree
  tree(long long _val = 0, tree* _left = NULL, tree* _right =
  NULL) {
    val = _val;        // Inicializa o valor do nó
    left = _left;      // Inicializa o ponteiro esquerdo
    right = _right;    // Inicializa o ponteiro direito
  }

  // Função para construir a árvore de segmentos
  // L e R representam o intervalo coberto por este nó
  void build_tree(int L, int R) {
    if(L == R) return;  // Caso base: folha da árvore

    int mid = (L + R)/2; // Calcula o ponto médio do intervalo
    // Cria subárvore esquerda para o intervalo [L, mid]
    left = new tree; left->build_tree(L, mid);
    // Cria subárvore direita para o intervalo [mid+1, R]
    right = new tree; right->build_tree(mid + 1, R);
  }

  // Função para atualizar valores na árvore de segmentos
  // Cria uma nova versão da árvore (persistência)
  // qL e qR definem o intervalo de atualização, v é o valor a ser
  adicionado
  tree* update(int L, int R, int qL, int qR, long long v) {
    // Se o intervalo atual não intersecta o intervalo de
    atualização
    if(L > qR || R < qL) return this;
```

```
    // Se o intervalo atual está completamente contido no intervalo
    de atualização
    if(qL <= L && R <= qR) {
      // Cria um novo nó com o valor atualizado
      tree* new_node = new tree(val, left, right);
      new_node->val += v;  // Atualiza o valor
      return new_node;
    }

    // Caso parcial: cria novo nó e atualiza recursivamente
    tree* new_node = new tree(val);
    int mid = (L + R)/2;
    // Atualiza subárvore esquerda
    new_node->left = left->update(L, mid, qL, qR, v);
    // Atualiza subárvore direita
    new_node->right = right->update(mid + 1, R, qL, qR, v);

    return new_node;
  }

  // Função para consultar o valor acumulado em uma posição x
  long long query(int L, int R, int x) {
    if(L == R) return val;  // Caso base: folha da árvore

    int mid = (L + R)/2;
    if(x <= mid) {  // Se x está na subárvore esquerda
      return val + left->query(L, mid, x);
    }
    else {  // Se x está na subárvore direita
      return val + right->query(mid + 1, R, x);
    }
  }
};

// Arrays e vetores globais
long long P[300001];        // Armazena os valores P[i]
tree* root[300001];         // Armazena as raízes das versões da
árvore
vector<int> I[300001];      // Armazena os índices para cada
elemento
```

```
int main() {
  int N, M; scanf("%d %d", &N, &M);  // Lê N e M

  // Lê os índices e armazena em I[t]
  for(int i = 1; i <= M; i++) {
    int t; scanf("%d", &t);
    I[t].push_back(i);
  }

  // Lê os valores P[i]
  for(int i = 1; i <= N; i++) scanf("%lld", &P[i]);

  // Inicializa a árvore de segmentos vazia
  root[0] = new tree; root[0]->build_tree(1, M);

  int K; scanf("%d", &K);  // Lê o número de atualizações
  // Processa K atualizações
  for(int i = 1; i <= K; i++) {
    int L, R, A; scanf("%d %d %d", &L, &R, &A);  // Lê os
    parâmetros da atualização
    // Trata caso em que o intervalo cruza o limite (circular)
    if(L > R) {
      root[i] = root[i - 1]->update(1, M, L, M, A);  // Atualiza
      [L, M]
      root[i] = root[i]->update(1, M, 1, R, A);      // Atualiza
      [1, R]
    }
    else {
      root[i] = root[i - 1]->update(1, M, L, R, A);  // Atualiza
      [L, R]
    }
  }

  // Para cada elemento i de 1 a N
  for(int i = 1; i <= N; i++) {
    int L = 1, R = K + 1;  // Inicializa busca binária
    while(L < R) {  // Busca binária para encontrar menor versão
    válida
      int mid = (L + R)/2;

      long long samples = 0;  // Contador de amostras
      // Verifica todos os índices relevantes para i
```

```
      for(int j = 0; j < I[i].size(); j++) {
        samples += root[mid]->query(1, M, I[i][j]);  // Consulta
        valor acumulado
        if(samples >= P[i]) break;  // Para se já atingiu o valor
        necessário
      }
      if(samples >= P[i]) {  // Se encontrou valor suficiente
        R = mid;
      }
      else {  // Se precisa de mais atualizações
        L = mid + 1;
      }
    }
    if(L == K + 1) printf("NIE\n");  // Se não encontrou solução
    else printf("%d\n", L);          // Imprime menor versão válida
  }
}
```