Nombre: Campo Vilela Natasha Stefania

NRC: 29852

EJERCICIOS DE BÚSQUEDA BINARIA

Problema: Cobertura de Wifi en el Pasillo

En una universidad hay N salones ubicados a lo largo de un pasillo recto, cada uno con una coordenada entera que indica su posición. Se instalan M routers wifi fijos en el mismo pasillo, también con coordenadas enteras conocidas. Cada router cubre todos los salones que estén a una distancia menor o igual a r (en valor absoluto). Se pide determinar el valor mínimo de r que garantiza que todos los salones tengan cobertura de al menos un router.

Entrada:

Primera línea: N y M.

Segunda línea: N enteros con las posiciones de los salones (ordenados no decrecientemente).

Tercera línea: M enteros con las posiciones de los routers (ordenados no decrecientemente).

Salida:

Un entero r mínimo que cubre todos los salones.

```
#include <iostream>
#include <algorithm>
#include <stdlib.h>   // llabs

using namespace std;

bool check(long long *rooms, int N, long long *routers, int M, long long r) {
   int i = 0; // índice de salas
   int j = 0; // índice de routers

   while (i < N && j < M) {//N salas y M routers
      // Muevo el router mientras esté demasiado a la izquierda
      while (j < M && routers[j] + r < rooms[i]) {
         j++;
      }
      if (j == M) return false; // ya no hay routers

      // Si este router cubre la sala actual, paso a la siguiente sala
      if (llabs(routers[j] - rooms[i]) <= r) {
         i++;
      } else {
         // router demasiado a la derecha, y no hay otro más a la izquierda que la cubra
         return false;
      }
   }
   return (i == N); // true si cubrimos todas las salas
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int N, M;
    if (!(cin >> N >> M)) return 0;

    long long *rooms   = new long long[N];
    long long *routers = new long long[M];

    for (int i = 0; i < N; i++) cin >> rooms[i];
    for (int i = 0; i < M; i++) cin >> routers[i];

    // Por seguridad (aunque el enunciado dice que vienen ordenados)
    sort(rooms, rooms + N);
    sort(routers, routers + M);

    long long low = 0;
    long long high = 0;

    // high: cota superior segura
    long long d1 = llabs(rooms[N-1]   - routers[0]);   // salón más a la derecha vs router más a la
izquierda
    long long d2 = llabs(routers[M-1] - rooms[0]);     // router más a la derecha vs salón más a la
izquierda
    high = (d1 > d2) ? d1 : d2;

    long long ans = high;

    while (low <= high) {
        long long mid = (low + high) / 2;
        if (check(rooms, N, routers, M, mid)) {
            ans = mid;        // este r funciona, intentamos uno menor
            high = mid - 1;
        } else {
            low = mid + 1;    // este r no alcanza, probamos más grande
        }
    }
    cout << ans << "\n";

    delete[] rooms;
    delete[] routers;

    return 0;
}
```

Problema: Faros en la Costa

Una costa recta se modela como una recta numérica 1D. Hay N pueblos costeros ubicados en posiciones enteras sobre esa recta. El gobierno instalará M faros (ya están posicionados), cada faro ilumina un intervalo simétrico de longitud 2r alrededor de su posición. Un pueblo se considera

“seguro” si está dentro del alcance de al menos un faro. Tu tarea es calcular el radio mínimo r de iluminación de los faros para que todos los pueblos sean seguros.

Entrada:

Primera línea: N y M.

Segunda línea: N posiciones de pueblos (ordenadas).

Tercera línea: M posiciones de faros (ordenadas).

Salida:

El valor mínimo de r que hace que todos los pueblos estén iluminados.

```
#include <iostream>
#include <algorithm>
#include <stdlib.h>   // llabs
using namespace std;

// Verifica si con radio r todos los pueblos quedan cubiertos por algún faro.
bool canCover(long long *towns, int N, long long *lighthouses, int M, long long r) {
    int i = 0; // índice pueblos
    int j = 0; // índice faros

    // Recorremos mientras queden pueblos y faros
    while (i < N && j < M) {
        // Avanzar faros que están demasiado a la izquierda para cubrir towns[i]
        while (j < M && lighthouses[j] + r < towns[i]) {
            j++;
        }
        if (j == M) break; // ya no hay faros para cubrir este pueblo

        // Si el faro j cubre towns[i], pasamos al siguiente pueblo
        if (llabs(lighthouses[j] - towns[i]) <= r) {
            i++;
        } else {
            // faro demasiado a la derecha para towns[i], y no hay otro más a la izquierda
            break;
        }
    }
    // Si i == N, todos los pueblos (0..N-1) fueron cubiertos
    return (i == N);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int N, M;
    cin >> N >> M;   // leer cantidad de pueblos y faros

    long long *towns = new long long[N];
```

```
    long long *lighthouses = new long long[M];

    // leer posiciones de pueblos
    for (int i = 0; i < N; i++) {
        cin >> towns[i];
    }

    // leer posiciones de faros
    for (int i = 0; i < M; i++) {
        cin >> lighthouses[i];
    }

    // Por seguridad ordenamos (aunque el enunciado dice que ya vienen ordenados)
    sort(towns, towns + N);
    sort(lighthouses, lighthouses + M);

    // Búsqueda binaria sobre el radio r
    long long low = 0;

    // Cota alta: máximo de las distancias extremas
    long long d1 = llabs(towns[N - 1] - lighthouses[0]);
    long long d2 = llabs(lighthouses[M - 1] - towns[0]);
    long long high = (d1 > d2) ? d1 : d2;

    long long ans = high;

    while (low <= high) {
        long long mid = (low + high) / 2; // r candidato

        if (canCover(towns, N, lighthouses, M, mid)) {
            ans = mid;        // mid funciona, probamos radios más pequeños
            high = mid - 1;
        } else {
            low = mid + 1;    // mid no alcanza, necesitamos radio más grande
        }
    }

    cout << ans << "\n";

    delete[] towns;
    delete[] lighthouses;
    return 0;
}
```

Problema: Megacity

Una ciudad está en el origen, y alrededor hay N puntos con cierta "población". Quieres encontrar el radio mínimo r que hay que "expandir" desde el origen para que la suma de poblaciones de todos los puntos dentro del círculo de radio r sea al menos S. Se calcula la distancia de cada punto al origen, se ordenan por distancia, se hace un prefijo de poblaciones, y se busca el menor r tal que la población acumulada ≥ S, usando búsqueda binaria o un simple recorrido.

Entrada:

Primera línea: N y S

N = número de puntos

S = población objetivo

Siguientes N líneas: x,y,p

(x,y) coordenadas del punto

P población aportada por ese punto

Salida:

Radio mínimo r tal que la suma de poblaciones de puntos con distancia <= r es >= S

Si no se puede llegar a S, puedes imprimir -1

El valor mínimo de r que hace que todos los pueblos estén iluminados.

```
#include <iostream>
#include <algorithm>
#include <math.h>    // sqrt
using namespace std;

// Vamos a guardar distancias y poblaciones en arreglos con punteros.

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int N;
    long long S;
    if (!(cin >> N >> S)) return 0;

    // Si ya somos megaciudad en el origen sin puntos, podría considerarse r=0
    // pero normalmente se asume que S > 0 y se necesitan puntos.

    double *dist = new double[N];      // distancia desde el origen
    long long *pop = new long long[N]; // población de cada punto

    long long totalPop = 0;

    for (int i = 0; i < N; i++) {
```

```
    long long x, y, p;
    cin >> x >> y >> p;
    totalPop += p;
    dist[i] = sqrt((double)x * x + (double)y * y);
    pop[i] = p;
}

// Si ni sumando todas las poblaciones llegamos a S, no hay solución.
if (totalPop < S) {
    cout << -1 << "\n";
    delete[] dist;
    delete[] pop;
    return 0;
}

// Ordenamos los puntos por distancia al origen
int *idx = new int[N];
for (int i = 0; i < N; i++) idx[i] = i;

sort(idx, idx + N, [&](int a, int b){
    return dist[a] < dist[b];
});

double *distSorted = new double[N];
long long *popSorted = new long long[N];

for (int i = 0; i < N; i++) {
    distSorted[i] = dist[idx[i]];
    popSorted[i] = pop[idx[i]];
}

delete[] dist;
delete[] pop;
delete[] idx;

// Prefijo de poblaciones
long long *prefix = new long long[N];
long long sum = 0;
for (int i = 0; i < N; i++) {
    sum += popSorted[i];
    prefix[i] = sum;
}

// Buscamos el índice mínimo i tal que prefix[i] >= S
int pos = -1;
for (int i = 0; i < N; i++) {
    if (prefix[i] >= S) {
        pos = i;
        break;
    }
}
```

```
    // Como ya verificamos totalPop >= S, pos siempre debe ser != -1
    double answer = distSorted[pos];

    cout.setf(ios::fixed);
    cout.precision(7);
    cout << answer << "\n";

    delete[] distSorted;
    delete[] popSorted;
    delete[] prefix;

    return 0;
}
```

Ejercicio con HASH

```
#include <iostream>
#include <string>
using namespace std;

const int TAM = 10;  // Tamaño de la tabla hash

// Estructura para guardar nombre y cédula
struct Persona {
    string nombre;
    string cedula;
    int hashNombre;
    int hashCedula;
    bool ocupado = false;
};

// Tabla hash
Persona tabla[TAM];

// Función hash general (sirve para nombre y cedula)
int funcionHash(string texto) {
    int suma = 0;
    for (char c : texto) {
        suma += c;    // Suma los códigos ASCII
    }
    return suma % TAM;
}

// Insertar una persona usando hash de la cédula
void insertar(string nombre, string cedula) {

    int hashName = funcionHash(nombre);
    int hashID   = funcionHash(cedula);
    int pos = hashID;   // posición según cédula
```

```
    // Resolución de colisiones lineal
    while (tabla[pos].ocupado) {
        pos = (pos + 1) % TAM;
    }

    tabla[pos].nombre = nombre;
    tabla[pos].cedula = cedula;
    tabla[pos].hashNombre = hashName;
    tabla[pos].hashCedula = hashID;
    tabla[pos].ocupado = true;

    cout << "\nGuardado en la posicion: " << pos << endl;
    cout << "HASH del NOMBRE: " << hashName << endl;
    cout << "HASH de la CEDULA: " << hashID << endl;
}

// Buscar una persona por cédula
void buscar(string cedula) {
    int pos = funcionHash(cedula);
    int inicio = pos;

    while (tabla[pos].ocupado) {
        if (tabla[pos].cedula == cedula) {
            cout << "\n*** Persona encontrada ***" << endl;
            cout << "Nombre: " << tabla[pos].nombre << endl;
            cout << "Cedula: " << tabla[pos].cedula << endl;
            cout << "Hash Nombre: " << tabla[pos].hashNombre << endl;
            cout << "Hash Cedula: " << tabla[pos].hashCedula << endl;
            return;
        }
        pos = (pos + 1) % TAM;
        if (pos == inicio) break;  // dio la vuelta
    }

    cout << "\nLa cedula NO está en la tabla." << endl;
}

// Mostrar tabla hash completa
void mostrarTabla() {
    cout << "\n\n===== TABLA HASH =====\n";
    for (int i = 0; i < TAM; i++) {
        cout << "[" << i << "] ";
        if (tabla[i].ocupado) {
            cout << tabla[i].nombre
                 << " | Ced: " << tabla[i].cedula
                 << " | HN: " << tabla[i].hashNombre
                 << " | HC: " << tabla[i].hashCedula;
        } else {
            cout << "(vacio)";
        }
```

```
        cout << endl;
    }
}

int main() {
    int opc;
    string nombre, cedula;

    do {
        cout << "\n--- MENU ---" << endl;
        cout << "1. Insertar persona" << endl;
        cout << "2. Buscar por cedula" << endl;
        cout << "3. Mostrar tabla hash" << endl;
        cout << "4. Salir" << endl;
        cout << "Opcion: ";
        cin >> opc;

        switch (opc) {
            case 1:
                cout << "Ingrese nombre: ";
                cin >> nombre;
                cout << "Ingrese cedula: ";
                cin >> cedula;
                insertar(nombre, cedula);
                break;

            case 2:
                cout << "Ingrese cedula a buscar: ";
                cin >> cedula;
                buscar(cedula);
                break;

            case 3:
                mostrarTabla();
                break;
        }

    } while (opc != 4);

    return 0;
}
```