Ejercicio búsqueda binaria – Distancia Mínima

Enunciado

Se desea desarrollar un programa que permita encontrar la distancia mínima entre un valor entero X ingresado por el usuario y cualquier número perteneciente a un arreglo también ingresado por el usuario.

La distancia se calcula como:

$$|\text{número} - X|$$

Dado que el arreglo puede venir desordenado, primero debe ordenarse.
Posteriormente, se debe aplicar búsqueda binaria para encontrar el número más cercano a X sin recorrer todo el arreglo.

El programa debe mostrar mensajes claros para el usuario:

- Ingrese el tamaño del arreglo
- Ingrese el valor a encontrar
- Ingrese los elementos del arreglo
- Resultado final: distancia mínima

Análisis

Para resolver el problema se siguen los siguientes pasos:

1. Lectura del tamaño del arreglo, del valor X y de cada uno de los elementos.
2. El arreglo ingresado puede venir en cualquier orden, por lo que el primer paso fundamental es ordenarlo.
   Esto es importante porque la función `lower_bound()` solo funciona con arreglos ordenados.
3. Una vez ordenado el arreglo, se utiliza búsqueda binaria para encontrar la primera posición donde podría insertarse el número X sin romper el orden.
4. A partir de esta posición, se identifican dos posibles números cercanos:
   - El número en la posición entregada por `lower_bound` (candidato a la derecha).
   - El número inmediatamente anterior (candidato a la izquierda).
5. Se calcula la distancia absoluta de ambos candidatos con respecto a X y se escoge el valor mínimo.
6. Finalmente, se muestra la distancia mínima encontrada.

Este método es eficiente porque:

- Ordenar toma O(n log n)
- La búsqueda binaria toma O(log n)
- El cálculo de distancias toma O(1)

Codificación en C++

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

int main() {

    int cantidad, valorBuscado;

    cout << "Ingrese el tamano del arreglo: ";
    cin >> cantidad;

    cout << "Ingrese el valor a encontrar: ";
    cin >> valorBuscado;

    vector<int> numeros(cantidad);

    cout << "Ingrese los elementos del arreglo:" << endl;
    for (int i = 0; i < cantidad; i++) {
        cout << "Elemento " << i + 1 << ": ";
        cin >> numeros[i];
    }

    // Ordenar el arreglo (muy importante para usar lower_bound)
    sort(numeros.begin(), numeros.end());

    // Búsqueda binaria para encontrar el número más cercano
    auto posicion = lower_bound(numeros.begin(), numeros.end(),
valorBuscado);

    int distanciaMinima = INT_MAX;

    // Candidato a la derecha
    if (posicion != numeros.end())
        distanciaMinima = min(distanciaMinima, abs(*posicion -
valorBuscado));

    // Candidato a la izquierda
    if (posicion != numeros.begin()) {
        posicion--;
        distanciaMinima = min(distanciaMinima, abs(*posicion -
valorBuscado));
    }

    cout << "\n---------------------------\n";
    cout << "La distancia minima es: " << distanciaMinima << endl;
    cout << "---------------------------\n";

    return 0;
}
```

Ejercicio búsqueda binaria – “Buscar el Primer Elemento Mayor o Igual que X”

Enunciado

Dado un arreglo de números enteros ordenado de menor a mayor, y un valor X, se desea encontrar:

La posición del primer número en el arreglo que sea mayor o igual que X.

Si no existe ningún número mayor o igual que X, se debe indicar que no existe.

El programa debe:

1. Solicitar el tamaño del arreglo.
2. Solicitar sus elementos (el usuario ya los ingresa ordenados).
3. Solicitar el valor X.
4. Aplicar búsqueda binaria para encontrar la primera posición cuyo valor cumpla:

$$arreglo[i] \geq X$$

5. Mostrar la posición (1-based) y el valor encontrado.

Análisis

Este problema requiere una variante de la búsqueda binaria llamada:

"binarizar para encontrar la primera aparición que cumple una condición"

La idea es:

- Buscar un punto medio.
- Si `arreglo[mid]` es mayor o igual a X:
  - Es candidato válido, pero podría haber otro más a la izquierda → mover la búsqueda hacia la izquierda.
- Si `arreglo[mid]` es menor que X:
  - No sirve → mover la búsqueda hacia la derecha.

Complejidad:

- O(log n) gracias a la búsqueda binaria.

Codificación en C++

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

int main() {

    int cantidad;
    cout << "Ingrese el tamano del arreglo: ";
    cin >> cantidad;

    vector<int> numeros(cantidad);

    cout << "Ingrese los elementos del arreglo (en orden ascendente):"
<< endl;
    for (int i = 0; i < cantidad; i++) {
        cout << "Elemento " << i + 1 << ": ";
        cin >> numeros[i];
    }

    int buscado;
    cout << "Ingrese el valor X a buscar: ";
    cin >> buscado;

    int inicio = 0, fin = cantidad - 1;
    int posicion = -1; // -1 significa "no encontrado"

    while (inicio <= fin) {
        int medio = (inicio + fin) / 2;

        if (numeros[medio] >= buscado) {
            posicion = medio;   // posible candidato
            fin = medio - 1;    // buscar más a la izquierda
        } else {
            inicio = medio + 1; // buscar a la derecha
        }
    }

    if (posicion == -1) {
        cout << "\nNo existe ningun numero mayor o igual que " <<
buscado << endl;
    } else {
        cout << "\nEl primer numero mayor o igual que " << buscado <<
" es: "
             << numeros[posicion] << endl;
        cout << "Posicion: " << (posicion + 1) << endl; // posición
humana
    }

    return 0;
}
```

Ejercicio búsqueda binaria – “Cortar barras de metal en longitudes iguales”

Enunciado

Tienes N barras de metal, cada una con una longitud dada.
Deseas cortar estas barras para obtener exactamente K piezas, todas con la misma longitud entera.

Tu objetivo es:

Encontrar la longitud máxima entera L que puedes obtener, de forma que sea posible cortar al menos K piezas iguales.

Importante:

- No se puede unir barras.
- Solo se puede cortar.
- Si una barra mide 10 y L = 3, se obtienen 3 piezas (3 + 3 + 3).
- Si una barra mide 5 y L = 3, se obtiene solo 1 pieza.

Análisis

No podemos probar todas las longitudes posibles manualmente porque sería muy lento.
Pero sí podemos notar algo importante:

- Si una longitud L funciona (permite obtener K piezas),
  entonces cualquier longitud menor también funciona.
- Si una longitud L no funciona, entonces ninguna mayor funciona.

Esto significa que la función de factibilidad es monótona, ideal para aplicar:

Búsqueda binaria sobre la longitud L.

¿Cómo se verifica una longitud L?

Para cada barra:

$$piezas += \left[\frac{longitud\_barra}{L}\right]$$

Si el total ≥ K → L es válida.

¿Qué buscamos?

El mayor L posible que sea válido.
Por eso:

- Si L funciona → buscar valores más grandes
- Si L no funciona → buscar más pequeños

Codificación en C++

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

int main() {

    int cantidadBarras, piezasNecesarias;

    cout << "Ingrese la cantidad de barras: ";
    cin >> cantidadBarras;

    cout << "Ingrese la cantidad de piezas que se necesitan: ";
    cin >> piezasNecesarias;

    vector<int> barras(cantidadBarras);

    cout << "Ingrese la longitud de cada barra:" << endl;
    for (int i = 0; i < cantidadBarras; i++) {
        cout << "Barra " << i + 1 << ": ";
        cin >> barras[i];
    }

    int inicio = 1;
    int fin = *max_element(barras.begin(), barras.end());
    int mejorLongitud = 0;

    while (inicio <= fin) {
        int mitad = (inicio + fin) / 2;

        long long piezas = 0;
        for (int longitud : barras)
            piezas += longitud / mitad;

        if (piezas >= piezasNecesarias) {
            mejorLongitud = mitad;
            inicio = mitad - 1;
            inicio = mitad + 1;  // intentamos una L mas grande
        } else {
            fin = mitad - 1;     // L es muy grande, intentamos una
menor
        }
    }

    cout << "\nLa maxima longitud entera posible es: "
         << mejorLongitud << endl;

    return 0;
}
```

Ejercicio de Hash – “Detectar números repetidos”

Enunciado

Cree un programa que:

1. Pida al usuario cuántos números va a ingresar.
2. El usuario ingresa los números uno por uno.
3. El programa debe usar una tabla hash (unordered_map) para determinar:
    - Cuántos números están repetidos.
    - Mostrar una lista de los números que se repiten.

Debe mostrar:

- Los números repetidos
- Cuántas veces aparecen
- Si no hay repetidos, indicarlo

Análisis

Para resolver este ejercicio de forma eficiente se utiliza una tabla hash.

- La estructura `unordered_map<int, int>` permite almacenar cada número como clave y su cantidad de apariciones como valor.
- Cada vez que se lee un número, se incrementa su contador.
- Al finalizar la lectura, se recorre el mapa para mostrar todos los números cuya frecuencia sea mayor o igual a 2.

Complejidad:

- Insertar en un `unordered_map` es operación O(1) promedio.
- Recorrer todas las entradas también es eficiente.
- La complejidad total es O(n) en promedio, mucho más rápido que un método tradicional que usaría doble bucle $O(n^2)$.

Esta técnica es muy usada en problemas donde se necesita saber si hay elementos repetidos o cuántas veces se repiten.

Codificación en C++

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

int main() {

    int cantidad;

    cout << "Ingrese la cantidad de numeros: ";
    cin >> cantidad;

    unordered_map<int, int> contador;

    cout << "Ingrese los numeros:" << endl;

    for (int i = 0; i < cantidad; i++) {
        int num;
        cout << "Numero " << i + 1 << ": ";
        cin >> num;
        contador[num]++;  // aumenta su frecuencia en el hash
    }

    cout << "\n-----------------------------\n";
    cout << "NUMEROS REPETIDOS\n";
    cout << "-----------------------------\n";

    bool hayRepetidos = false;

    for (auto &par : contador) {
        if (par.second >= 2) {
            cout << "Numero " << par.first
                 << " aparece " << par.second << " veces." << endl;
            hayRepetidos = true;
        }
    }

    if (!hayRepetidos) {
        cout << "No existen numeros repetidos." << endl;
    }

    return 0;
}
```