Universidad De Las Fuerzas Armadas ESPE

Nombre: Mateo De la Cruz

Fecha: 02-12-2025

Ejercicios de búsqueda binaria

- Ejercicio 1

Problema: Asignación de Cargadores de Vehículos Eléctricos

En una ciudad, se han identificado N **puntos de estacionamiento** a lo largo de una carretera principal. Cada punto tiene una coordenada entera que indica su posición. La ciudad planea instalar M **estaciones de carga rápida** fijas en la misma carretera, también con coordenadas enteras conocidas.

Cada estación de carga cubre todos los puntos de estacionamiento que estén a una distancia **menor o igual a D** (en valor absoluto) de ella.

Se pide **determinar el valor mínimo de D** que garantiza que **todos** los puntos de estacionamiento tengan cobertura de al menos una estación de carga.

Entrada:

1. **Primera línea:** Dos enteros N (número de puntos de estacionamiento) y M (número de estaciones de carga).
2. **Segunda línea:** N enteros con las **posiciones de los puntos de estacionamiento** (ordenados no decrecientemente).
3. **Tercera línea:** M enteros con las **posiciones de las estaciones de carga** (ordenados no decrecientemente).

Salida:

Un entero **D mínimo** que garantiza la cobertura de todos los puntos de estacionamiento.

```
#include <iostream>
#include <cstdlib>

using namespace std;

long long valor_absoluto(long long n) {
  return (n < 0) ? -n : n;
}
```

```
bool verificar(long long D, const int* estacionamientos, int N, const int*
cargadores, int M) {
  if (M == 0) return (N == 0);

  int indice_cargador = 0;

  for (int i = 0; i < N; ++i) {
    long long pos_estacionamiento = estacionamientos[i];

    while (indice_cargador < M - 1) {
      long long dist_actual = valor_absoluto((long
long)cargadores[indice_cargador] - pos_estacionamiento);
      long long dist_siguiente = valor_absoluto((long
long)cargadores[indice_cargador + 1] - pos_estacionamiento);

      if (dist_siguiente <= dist_actual) {
        indice_cargador++;
      } else {
        break;
      }
    }

    long long distancia_minima = valor_absoluto((long
long)cargadores[indice_cargador] - pos_estacionamiento);

    if (distancia_minima > D) {
      return false;
    }
  }

  return true;
}

void resolver_problema() {
  int N, M;

  // Lectura de N y M
  cin >> N >> M;

  if (N < 0 || M < 0) return;

  int* estacionamientos = nullptr;
  int* cargadores = nullptr;

  // Lectura de estacionamientos
  if (N > 0) {
    estacionamientos = new (nothrow) int[N];
```

```
    if (!estacionamientos) { return; }
    for (int i = 0; i < N; ++i) {
        if (!(cin >> estacionamientos[i])) {
            delete[] estacionamientos;
            return;
        }
    }
}

// Lectura de cargadores
if (M > 0) {
    cargadores = new (nothrow) int[M];
    if (!cargadores) {
        delete[] estacionamientos;
        return;
    }
    for (int i = 0; i < M; ++i) {
        if (!(cin >> cargadores[i])) {
            delete[] estacionamientos;
            delete[] cargadores;
            return;
        }
    }
}

if (N == 0) {
    cout << 0 << endl;
    delete[] estacionamientos;
    delete[] cargadores;
    return;
}

// Rango de búsqueda para D
int min_coord = estacionamientos[0];
int max_coord = estacionamientos[N - 1];

if (M > 0) {
    if (cargadores[0] < min_coord) min_coord = cargadores[0];
    if (cargadores[M - 1] > max_coord) max_coord = cargadores[M - 1];
}

long long low = 0;
long long high = (long long)max_coord - min_coord;

long long min_D = high + 1;

// Búsqueda Binaria
```

```
    while (low <= high) {
        long long mid = low + (high - low) / 2;

        if (verificar(mid, estacionamientos, N, cargadores, M)) {
            min_D = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    if (M == 0 && N > 0) {
        // En un entorno de programación competitiva, si la solución es imposible,
        // la salida puede ser un error o un valor muy grande.
        // Asumiendo que el caso de prueba siempre tiene solución si N>0 y M>0,
        // o que N=0 si M=0. Si el caso M=0, N>0 es válido, la salida esperada varía.
        // Aquí, simplemente no se imprime un resultado válido.
        // Si el problema garantiza que siempre hay cobertura posible, esta línea no
es necesaria.
    } else {
        cout << min_D << endl;
    }

    delete[] estacionamientos;
    delete[] cargadores;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    resolver_problema();

    return 0;
}
```

- Ejercicio 2

Problema: Distribución de Tareas en Máquinas

Tienes N **tareas** y M **máquinas** idénticas. Cada tarea i requiere un tiempo Ti para ser completada por una máquina. Las tareas pueden asignarse a cualquier máquina. Una vez que una máquina comienza una tarea, no puede empezar otra hasta que la primera finalice.

Tu objetivo es asignar todas las N tareas a las M máquinas de tal manera que el **tiempo máximo de finalización** (es decir, el momento en que la última máquina termina su última tarea) sea **mínimo**.

Se pide determinar el **tiempo mínimo** en el que todas las tareas pueden ser completadas.

Entrada:

1. **Primera línea:** Dos enteros N (número de tareas) y M (número de máquinas).
2. **Segunda línea:** N enteros con los **tiempos** Ti que requiere cada tarea.

Salida:

Un entero que representa el **tiempo mínimo** de finalización para todas las tareas.

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

using namespace std;

bool verificar(long long T, const int* tiempos_tareas, int N, int M) {
  if (M <= 0) return false;
  if (N == 0) return true;

  int maquinas_necesarias = 1;
  long long tiempo_acumulado_maquina = 0;

  for (int i = 0; i < N; ++i) {
    long long tiempo_tarea_actual = tiempos_tareas[i];

    if (tiempo_tarea_actual > T) {
      return false;
    }

    if (tiempo_acumulado_maquina + tiempo_tarea_actual <= T) {
      tiempo_acumulado_maquina += tiempo_tarea_actual;
    } else {
      maquinas_necesarias++;
      tiempo_acumulado_maquina = tiempo_tarea_actual;
    }
  }

  return maquinas_necesarias <= M;
```

```
}

void resolver_problema() {
  int N, M;

  if (!(cin >> N >> M)) return;

  if (N < 0 || M < 0) return;

  int* tiempos_tareas = nullptr;
  long long suma_tiempos = 0;
  long long tarea_maxima = 0;

  if (N > 0) {
    tiempos_tareas = new (nothrow) int[N];
    if (!tiempos_tareas) return;

    for (int i = 0; i < N; ++i) {
      if (!(cin >> tiempos_tareas[i])) {
        delete[] tiempos_tareas;
        return;
      }
      suma_tiempos += tiempos_tareas[i];
      tarea_maxima = std::max(tarea_maxima, (long long)tiempos_tareas[i]);
    }
  }

  if (N == 0) {
    cout << 0 << endl;
    delete[] tiempos_tareas;
    return;
  }

  long long low = tarea_maxima;
  long long high = suma_tiempos;
  long long min_tiempo_finalizacion = suma_tiempos;

  // Búsqueda Binaria
  while (low <= high) {
    long long mid = low + (high - low) / 2;

    if (verificar(mid, tiempos_tareas, N, M)) {
      min_tiempo_finalizacion = mid;
      high = mid - 1;
    } else {
      low = mid + 1;
    }
```

```
    }

    cout << min_tiempo_finalizacion << endl;

    delete[] tiempos_tareas;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    resolver_problema();

    return 0;
}
```

- Ejercicio 3

Problema: Asignación de Capacidad de Almacenamiento

Tienes N **archivos** que necesitan ser almacenados. El tamaño de cada archivo i es Si. Deseas distribuir estos archivos en M **discos duros** idénticos, donde cada disco tiene una capacidad máxima C.

Tu objetivo es determinar la **mínima capacidad C** requerida para los discos duros de modo que todos los N archivos puedan ser almacenados utilizando exactamente M discos.

Se pide determinar el valor entero **mínimo de C**.

Entrada:

1. **Primera línea:** Dos enteros N (número de archivos) y M (número de discos).
2. **Segunda línea:** N enteros con los **tamaños** Si de cada archivo.

Salida:

Un entero que representa la **capacidad mínima C** de los discos.

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

using namespace std;

/**
```

```
 * @brief Verifica si una capacidad C es suficiente para almacenar todos los
archivos
 * utilizando como máximo M discos.
 */
bool verificar(long long C, const int* tamanos_archivos, int N, int M) {
  if (M <= 0 && N > 0) return false;
  if (N == 0) return true;

  int discos_necesarios = 1;
  long long capacidad_usada_actual = 0;

  for (int i = 0; i < N; ++i) {
    long long tamano_actual = tamanos_archivos[i];

    // Si un solo archivo excede la capacidad C, es imposible.
    if (tamano_actual > C) {
      return false;
    }

    if (capacidad_usada_actual + tamano_actual <= C) {
      // El archivo cabe en el disco actual.
      capacidad_usada_actual += tamano_actual;
    } else {
      // El archivo no cabe en el disco actual, se necesita un nuevo disco.
      discos_necesarios++;
      capacidad_usada_actual = tamano_actual;
    }
  }

  return discos_necesarios <= M;
}

void resolver_problema() {
  int N, M;

  if (!(cin >> N >> M)) return;

  if (N < 0 || M < 0) return;

  int* tamanos_archivos = nullptr;
  long long suma_tamanos = 0;
  long long tamano_maximo = 0;

  if (N > 0) {
    tamanos_archivos = new (nothrow) int[N];
    if (!tamanos_archivos) return;
```

```
    for (int i = 0; i < N; ++i) {
      if (!(cin >> tamanos_archivos[i])) {
        delete[] tamanos_archivos;
        return;
      }
      suma_tamanos += tamanos_archivos[i];
      tamano_maximo = std::max(tamano_maximo, (long
long)tamanos_archivos[i]);
    }
  }

  if (N == 0) {
    cout << 0 << endl;
    delete[] tamanos_archivos;
    return;
  }

  // Rango de búsqueda para C (capacidad mínima)
  long long low = tamano_maximo;
  long long high = suma_tamanos;
  long long min_capacidad = suma_tamanos;

  // Búsqueda Binaria
  while (low <= high) {
    long long mid = low + (high - low) / 2;

    if (verificar(mid, tamanos_archivos, N, M)) {
      // La capacidad 'mid' funciona. Guardamos y buscamos una capacidad
menor.
      min_capacidad = mid;
      high = mid - 1;
    } else {
      // La capacidad 'mid' es muy pequeña, necesitamos más.
      low = mid + 1;
    }
  }

  cout << min_capacidad << endl;

  delete[] tamanos_archivos;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  resolver_problema();
```

```
  return 0;
}
```

- Ejercicio 4

Problema: Optimización de Productividad en Fábricas

Tienes N **trabajadores** y M **fábricas**. Cada fábrica j tiene una capacidad máxima de producción Pj por hora. El objetivo es asignar a cada uno de los N trabajadores a exactamente una fábrica de tal manera que la **productividad por trabajador** sea la misma en todas las fábricas.

La productividad de una fábrica se calcula como su capacidad de producción Pj dividida por el número de trabajadores asignados a ella.

Se pide determinar el valor **máximo de productividad** K que se puede alcanzar en todas las fábricas.

Entrada:

1. **Primera línea:** Dos enteros N (número de trabajadores) y M (número de fábricas).
2. **Segunda línea:** M enteros con las **capacidades de producción** Pj de cada fábrica.

Salida:

Un número real que representa el **máximo valor de productividad** K.

```
#include <iostream>
#include <cstdlib>
#include <algorithm>
#include <cmath>

using namespace std;

bool verificar(double K, const int* capacidades, int M, int N) {
  if (K <= 0) return true;

  long long trabajadores_totales_asignables = 0;

  for (int i = 0; i < M; ++i) {
    long long trabajadores_asignables = floor((double)capacidades[i] / K);
    trabajadores_totales_asignables += trabajadores_asignables;
  }
```

```
  return trabajadores_totales_asignables >= N;
}

void resolver_problema() {
  int N, M;

  if (!(cin >> N >> M)) return;

  if (N < 0 || M < 0) return;

  int* capacidades = nullptr;
  double capacidad_maxima = 0.0;

  if (M > 0) {
    capacidades = new (nothrow) int[M];
    if (!capacidades) return;

    for (int i = 0; i < M; ++i) {
      if (!(cin >> capacidades[i])) {
        delete[] capacidades;
        return;
      }
      capacidad_maxima = max(capacidad_maxima, (double)capacidades[i]);
    }
  }

  if (N == 0 || M == 0) {
    cout.precision(10);
    cout << fixed << 0.0 << endl;
    delete[] capacidades;
    return;
  }

  double low = 0.0;
  double high = capacidad_maxima;
  double max_productividad = 0.0;

  // Búsqueda Binaria para números reales
  for (int iter = 0; iter < 100; ++iter) {
    double mid = low + (high - low) / 2.0;

    if (verificar(mid, capacidades, M, N)) {
      max_productividad = mid;
      low = mid;
    } else {
      high = mid;
```

```
      }
    }

    cout.precision(10);
    cout << fixed << max_productividad << endl;

    delete[] capacidades;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    resolver_problema();

    return 0;
}
```