ESTRUCTURA DE DATOS
PARCIAL No. 2

ACTIVIDAD

Nombre:	Alex López
Nivel:	Segundo Parcial
NRC:	29852
Asignatura:	Estructura de Datos

Universidad De Las Fuerzas Armadas
Parcial 2

Ejercicios de Búsqueda Binaria

Ejercicio 1. Cortando Cables (Maximizar un valor)

El Problema: Tienes `N` cables de diferentes longitudes. Necesitas obtener `K` piezas de cable que tengan exactamente la misma longitud. ¿Cuál es la **longitud máxima** posible para esas piezas? (Se pueden descartar los trozos sobrantes, pero no se pueden unir trozos).

Entrada: `N` (cables), `K` (piezas necesarias). Luego `N` enteros (longitudes). **Salida:** La longitud máxima entera.

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

using namespace std;

bool check(long long* cables, int n, int k, long long len) {
    if (len == 0) return true;
    int pedazos = 0;
    for (int i = 0; i < n; i++) {
        pedazos += (*(cables + i) / len);
    }
    return pedazos >= k;
}

int main() {

    // ios::sync_with_stdio(false); cin.tie(nullptr);

    int N, K;


    cout << "Ingrese la cantidad de cables (N) y los pedazos deseados (K): ";
    if (!(cin >> N >> K)) return 0;

    long long* cables = new long long[N];
    long long maxLen = 0;

    cout << "Ingrese las " << N << " longitudes de los cables: ";
    for (int i = 0; i < N; i++) {
```

```
        cin >> *(cables + i);
        if (*(cables + i) > maxLen) maxLen = *(cables + i);
    }

    long long low = 1, high = maxLen;
    long long ans = 0;

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

        if (check(cables, N, K, mid)) {
            ans = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    cout << "\nLa longitud maxima posible por pedazo es: " << ans << "\n";
    delete[] cables;
    return 0;
}
```

Ejercicio 2. Koko comiendo bananas (Minimizar velocidad)

El Problema:Hay N montones de bananas. El montón i-ésimo tiene P_i bananas. Tienes H horas para comer todas. Cada hora puedes elegir un montón y comer K bananas de él. Si el montón tiene menos de K, te lo comes todo y esperas a la siguiente hora.¿Cuál es la velocidad mínima K (bananas/hora) tal que puedas terminar todo en H horas?Entrada: N (montones), H (horas). Luego N enteros.Salida: El entero K mínimo.

```
#include <iostream>
#include <cmath> // Para ceil si quisieras, pero lo haremos manual

using namespace std;

// Función Check: Si comemos 'vel' bananas por hora, ¿terminamos en 'h' horas?
bool puedeTerminar(int* pilas, int n, int h, int vel) {
    long long horasNecesarias = 0;
    for (int i = 0; i < n; i++) {
        int bananas = *(pilas + i);

        // Fórmula equivalente a ceil(bananas / vel) usando enteros:
        // (a + b - 1) / b es el techo de a/b
        horasNecesarias += (bananas + vel - 1) / vel;
    }
```

```
    return horasNecesarias <= h;
}

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

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

    int* pilas = new int[N];
    int maxPila = 0;

    for (int i = 0; i < N; i++) {
        cin >> *(pilas + i);
        if (*(pilas + i) > maxPila) maxPila = *(pilas + i);
    }

    // Búsqueda: La velocidad mínima es 1, la máxima es comer el montón más grande de una vez
    int low = 1, high = maxPila;
    int ans = high;

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

        if (puedeTerminar(pilas, N, H, mid)) {
            ans = mid;      // Velocidad válida, intentemos hacerlo más lento (optimizar)
            high = mid - 1;
        } else {
            low = mid + 1;  // Demasiado lento, se nos acaba el tiempo
        }
    }

    cout << ans << "\n";
    delete[] pilas;
    return 0;
}
```

Ejercicio 3. Leñador Ecológico (Suma con umbral)

El Problema: Un leñador necesita `M` metros de madera. Tiene `N` árboles en fila de distintas alturas. Él ajusta su sierra a una altura `H`. Corta todos los árboles; las partes por encima de `H` caen y se las lleva. Las partes por debajo de `H` quedan plantadas. ¿Cuál es la **altura máxima `H`** a la que puede configurar la sierra para obtener *al menos* `M` metros de madera (para no cortar de más innecesariamente)?

Entrada: `N` (árboles), `M` (madera necesaria). `N` alturas. **Salida:** La altura máxima `H`.

```
#include <iostream>

#include <algorithm>



using namespace std;



// Check: Si cortamos a altura 'h', ¿obtenemos al menos 'maderaRequerida'?

bool maderaSuficiente(long long* arboles, int n, long long maderaRequerida, long long h) {

    long long maderaObtenida = 0;

    for (int i = 0; i < n; i++) {

        if (*(arboles + i) > h) {

            maderaObtenida += (*(arboles + i) - h);

        }

    }

    return maderaObtenida >= maderaRequerida;

}



int main() {
```

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


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


long long* arboles = new long long[N];
long long maxH = 0;


for (int i = 0; i < N; i++) {
    cin >> *(arboles + i);
    if (*(arboles + i) > maxH) maxH = *(arboles + i);
}


// min: 0, max: el árbol más alto
long long low = 0, high = maxH;
long long ans = 0;


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


    if (maderaSuficiente(arboles, N, M, mid)) {
```

```
            ans = mid;      // Conseguimos la madera, probemos subir la sierra (cortar menos
tronco)

            low = mid + 1;

        } else {

            high = mid - 1; // No llegamos a la madera necesaria, hay que bajar la sierra

        }

    }



    cout << ans << "\n";

    delete[] arboles;

    return 0;

}
```

Ejercicio 4. Asignación de Trabajadores para Máxima Productividad

Enunciado: Tienes `N` trabajadores y `M` fábricas. Tienes un arreglo `P` donde `Pj` es la capacidad de producción total de la fábrica `j`. La productividad de un trabajador en una fábrica se define como: `Capacidad de la fábrica / Número de trabajadores en esa fábrica` (división entera). Debes asignar a **todos** los `N` trabajadores a las fábricas de tal forma que se maximice la **productividad mínima garantizada** `K`. Es decir, queremos encontrar el mayor valor `K` tal que sea posible acomodar a los `N` trabajadores asegurando que ninguno tenga una productividad menor a `K`.

Entrada:

- `N` (trabajadores) y `M` (fábricas).
- `M` enteros representando las capacidades `Pj`.

Salida:

- El valor máximo entero `K`

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

using namespace std;

// Función Check:
// ¿Es posible mantener una productividad mínima de 'prodObjetivo' para TODOS los trabajadores?
// Lógica: Para cada fábrica, calculamos cuántos trabajadores caben allí
// sin que la productividad baje de 'prodObjetivo'.
bool esPosible(int* capacidades, int m, int n, int prodObjetivo) {
    if (prodObjetivo == 0) return true; // Evitar división por 0

    long long totalTrabajadoresQueCaben = 0;

    for (int i = 0; i < m; i++) {
        // Acceso por puntero: *(capacidades + i)
        // Fórmula: cupos = Capacidad / ProductividadObjetivo
        // Ejemplo: Si Capacidad=100 y queremos Prod=25, caben 4 trabajadores.
        totalTrabajadoresQueCaben += (*(capacidades + i) / prodObjetivo);
    }

    // Si la suma de cupos es mayor o igual a los trabajadores que tenemos (N), es posible.
    return totalTrabajadoresQueCaben >= n;
}

int main() {
    // 1. Configuración para inputs rápidos (opcional pero recomendada)
    // Recuerda que si usas esto, no verás mensajes en consola hasta el final.
    // Para pruebas manuales con texto, puedes comentar la siguiente línea:
    ios::sync_with_stdio(false); cin.tie(nullptr);

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

    int* capacidades = new int[M];
    int maxCapacidad = 0;

    for (int i = 0; i < M; i++) {
        cin >> *(capacidades + i);
        if (*(capacidades + i) > maxCapacidad) {
            maxCapacidad = *(capacidades + i);
        }
    }

    // Búsqueda Binaria sobre la PRODUCTIVIDAD (K)
```

```
    // Rango: De 1 hasta la capacidad más grande (caso 1 trabajador en la mejor fábrica)
    int low = 1;
    int high = maxCapacidad;
    int ans = 0;

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

        if (esPosible(capacidades, M, N, mid)) {
            ans = mid;      // Es posible esta productividad, intentemos buscar una mayor (mejor)
            low = mid + 1;
        } else {
            high = mid - 1; // Demasiado ambicioso, no caben los trabajadores, bajemos la
exigencia
        }
    }

    cout << ans << "\n";

    delete[] capacidades;
    return 0;
}
```