Ejercicios de Búsqueda Binaria

1. **Ejercicio de Cables (Búsqueda Binaria sobre la Respuesta - División)**

Problema: Cortando Cables
Eres un electricista y tienes N cables de diferentes longitudes. Necesitas obtener exactamente K piezas de cable, todas de la misma longitud entera. Puedes cortar los cables originales, pero no unirlos. Se pide determinar la longitud máxima entera L que permite obtener al menos K piezas.
Entrada:
Primera línea: N y K.
Segunda línea: N enteros con las longitudes de los cables.
Salida:
Un entero con la longitud máxima posible (0 si no es posible).

Código C++:

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

// Clase para gestionar los datos con punteros
class GestorCables {
public:
    long long* cables;
    int N;

    GestorCables(int n) {
        N = n;
        cables = new long long[n];
    }

    ~GestorCables() {
        delete[] cables;
    }

    void leerDatos() {
        for (int i = 0; i < N; i++) {
            cin >> *(cables + i);
        }
    }
```

```
    long long obtenerMaximo() {
        long long max_val = 0;
        for (int i = 0; i < N; i++) {
            if (*(cables + i) > max_val) max_val = *(cables + i);
        }
        return max_val;
    }
};

// Función de verificación: ¿Podemos sacar K piezas de tamaño 'longitud'?
bool esPosible(GestorCables* gestor, int K, long long longitud) {
    if (longitud == 0) return true;
    long long piezas = 0;
    for (int i = 0; i < gestor->N; i++) {
        piezas += (*(gestor->cables + i) / longitud);
    }
    return piezas >= K;
}

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

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

    GestorCables gestor(N);
    gestor.leerDatos();

    // Búsqueda Binaria sobre la Respuesta
    long long low = 1;
    long long high = gestor.obtenerMaximo();
    long long ans = 0;

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

        if (esPosible(&gestor, K, mid)) {
            ans = mid;
```

```
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    cout << ans << "
";
    return 0;
}
```

2. **Nuevo Ejercicio de Búsqueda Binaria (Estilo: Partición / Acumulación)**

Problema: Capacidad del Barco de Carga
Una cinta transportadora tiene N paquetes que deben ser enviados en un barco dentro de D días. El paquete i tiene un peso específico. Los paquetes deben cargarse en el barco en el orden en que llegan (no se puede reordenar). El barco tiene una capacidad de carga máxima C. Cada día, cargamos paquetes hasta que el peso total supere C, momento en el cual el barco zarpa y vuelve al día siguiente. Determina la capacidad mínima C del barco para poder transportar todos los paquetes en D días o menos.
Entrada:
Primera línea: N (paquetes) y D (días).
Segunda línea: N enteros representando el peso de cada paquete.
Salida:
Un entero representando la capacidad mínima necesaria.

Código C++:

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

// Clase contenedora
class CintaTransportadora {
public:
    int* pesos;
    int N;

    CintaTransportadora(int n) {
        N = n;
        pesos = new int[n];
```

```
    }

    ~CintaTransportadora() {
        delete[] pesos;
    }

    // Retorna el peso máximo individual (limite inferior de búsqueda)
    // y la suma total (limite superior)
    void obtenerLimites(long long &max_p, long long &sum_p) {
        max_p = 0;
        sum_p = 0;
        for(int i = 0; i < N; i++) {
            if (*(pesos + i) > max_p) max_p = *(pesos + i);
            sum_p += *(pesos + i);
        }
    }
};

// Función Greedy: Verifica si es posible enviar todo en 'dias_limite'
// usando un barco con 'capacidad'
bool verificarCapacidad(CintaTransportadora* cinta, int dias_limite, long long
capacidad) {
    int dias_usados = 1;
    long long carga_actual = 0;

    for (int i = 0; i < cinta->N; i++) {
        int peso_paquete = *(cinta->pesos + i);

        if (carga_actual + peso_paquete > capacidad) {
            dias_usados++;
            carga_actual = peso_paquete;
        } else {
            carga_actual += peso_paquete;
        }
    }
    return dias_usados <= dias_limite;
}

int main() {
    ios::sync_with_stdio(false);
```

```
    cin.tie(nullptr);

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

    CintaTransportadora cinta(N);
    for(int i=0; i<N; i++) {
        cin >> *(cinta.pesos + i);
    }

    long long low, high;
    cinta.obtenerLimites(low, high);
    long long ans = high;

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

        if (verificarCapacidad(&cinta, D, mid)) {
            ans = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    cout << ans << "
";
    return 0;
}
```

Ejercicios de Búsqueda HASH

3. **Nuevo Ejercicio con HASH (Estilo: Búsqueda de Complementos / Two Sum)**

Problema: Detectando la Pareja Oculta (Two Sum)

Se te entrega una lista de N números enteros únicos y un número objetivo S. Tu tarea es encontrar si existen dos números en la lista que sumados den exactamente S. Para hacerlo eficiente, debes implementar una Tabla Hash manual para almacenar los números visitados y verificar si el "complemento" (S - actual) ya existe en la tabla.

Entrada:

Primera línea: N (cantidad de números) y S (suma objetivo).
Segunda línea: N enteros.
Salida: Imprime los dos números encontrados num1 num2 (donde num1 < num2). Si no existe tal pareja, imprime "No existe".

Código C++:

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

const int TAM_HASH = 200003; // Un número primo grande para reducir colisiones

class NodoHash {
public:
    long long valor;
    bool ocupado;

    NodoHash() {
        ocupado = false;
        valor = 0;
    }
};

class TablaHash {
public:
    NodoHash* tabla;

    TablaHash() {
        tabla = new NodoHash[TAM_HASH];
    }

    ~TablaHash() {
        delete[] tabla;
    }

    int funcionHash(long long k) {
        long long val = (k < 0) ? -k : k;
        return val % TAM_HASH;
    }
```

```
    void insertar(long long n) {
        int pos = funcionHash(n);
        while ((tabla + pos)->ocupado) {
            pos = (pos + 1) % TAM_HASH;
        }
        (tabla + pos)->valor = n;
        (tabla + pos)->ocupado = true;
    }

    bool existe(long long n) {
        int pos = funcionHash(n);
        int inicio = pos;

        while ((tabla + pos)->ocupado) {
            if ((tabla + pos)->valor == n) {
                return true;
            }
            pos = (pos + 1) % TAM_HASH;
            if (pos == inicio) break;
        }
        return false;
    }
};

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

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

    TablaHash memoria;
    long long* entrada = new long long[N];
    bool encontrado = false;

    for (int i = 0; i < N; i++) {
        cin >> *(entrada + i);
    }
```

```
    for (int i = 0; i < N; i++) {
        long long actual = *(entrada + i);
        long long complemento = S - actual;

        if (memoria.existe(complemento)) {
            if (complemento < actual) cout << complemento << " " << actual << "
";
            else cout << actual << " " << complemento << "
";

            encontrado = true;
            break;
        }

        memoria.insertar(actual);
    }

    if (!encontrado) {
        cout << "No existe
";
    }

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