EJERCICIOS DE BÚSQUEDA BINARIA Y HASH

Nombre: Andino Herrera Kevin Mauricio

NRC: 29852

EJERCICIOS DE BÚSQUEDA BINARIA

Problema: El leñador Mirko necesita cortar M metros de madera. Tiene una máquina que funciona así: Mirko establece un parámetro de altura H (en metros), y la máquina levanta una cuchilla gigante a esa altura y corta todas las partes de árboles más altas que H. Mirko se lleva las partes cortadas.

Por ejemplo, si hay árboles de alturas 20, 15, 10 y 17 metros, y Mirko pone la cuchilla a 15 metros, las alturas restantes serán 15, 15, 10 y 15 metros, mientras que Mirko obtiene 5 metros del primer árbol y 2 metros del cuarto (7 metros en total).

Mirko es ecológico y no quiere cortar más madera de la necesaria. Quiere poner su cuchilla lo más alta posible que aún le permita cortar al menos M metros de madera.

Entrada:

- Primera línea: N (número de árboles, $1 \leq N \leq 1{,}000{,}000$) y M (metros requeridos, $1 \leq M \leq 2{,}000{,}000{,}000$)
- Segunda línea: N enteros positivos menores que 1,000,000,000 (alturas de los árboles)

Salida: La altura máxima requerida.

Ejemplo:

Input:

4 7

20 15 10 17

Output:

15

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

template<typename T>
class DynamicArray {
private:
   T* data;
   int capacity;
   int size;
```

```
    void resize() {
        capacity *= 2;
        T* newData = new T[capacity];
        for(int i = 0; i < size; i++) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
    }

public:
    DynamicArray() : capacity(10), size(0) {
        data = new T[capacity];
    }

    ~DynamicArray() {
        delete[] data;
    }

    void push_back(T value) {
        if(size >= capacity) resize();
        data[size++] = value;
    }

    T& operator[](int index) {
        return data[index];
    }

    int getSize() const { return size; }
    T* getData() { return data; }
};

// Función para verificar si podemos obtener al menos M metros con altura H
auto canCut = [](long long* trees, int n, long long h, long long m) -> bool {
    long long total = 0;
    for(int i = 0; i < n; i++) {
        if(trees[i] > h) {
            total += (trees[i] - h);
            if(total >= m) return true; // Optimización temprana
        }
    }
    return total >= m;
};

int main() {
    int n;
    long long m;
```

```
    cin >> n >> m;

    long long* trees = new long long[n];
    long long maxHeight = 0;

    for(int i = 0; i < n; i++) {
        cin >> trees[i];
        maxHeight = max(maxHeight, trees[i]);
    }

    // Binary search sobre la respuesta
    long long left = 0, right = maxHeight;
    long long answer = 0;

    while(left <= right) {
        long long mid = left + (right - left) / 2;

        if(canCut(trees, n, mid, m)) {
            answer = mid; // Esta altura funciona
            left = mid + 1; // Intentamos una altura mayor
        } else {
            right = mid - 1; // Necesitamos una altura menor
        }
    }

    cout << answer << endl;

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

Problema: Dado un arreglo de enteros nums y un entero k, dividir nums en k subarreglos no vacíos de manera que la suma más grande de cualquier subarreglo sea minimizada.

Entrada:

- Un arreglo nums de enteros no negativos
- Un entero m (número de subarreglos)

Salida: La suma máxima minimizada.

Ejemplo:

Input: nums = [7,2,5,10,8], m = 2

Output: 18

Explicación:

La mejor forma es dividirlo en [7,2,5] y [10,8],

donde la suma más grande entre los dos subarreglos es 18.

Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $0 \leq \text{nums}[i] \leq 10^6$
- $1 \leq m \leq \min(50, \text{nums.length})$

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

// Función para verificar si podemos dividir el array en m subarrays
// con suma máxima <= maxSum
auto canSplit = [](int* nums, int n, int m, long long maxSum) -> bool {
    int subarrays = 1;
    long long currentSum = 0;

    for(int i = 0; i < n; i++) {
        if(currentSum + nums[i] > maxSum) {
            // Necesitamos un nuevo subarray
            subarrays++;
            currentSum = nums[i];

            if(subarrays > m) return false;
        } else {
            currentSum += nums[i];
        }
    }

    return true;
};

int splitArray(int* nums, int n, int m) {
    // El rango de búsqueda es [max(nums), sum(nums)]
    long long left = 0, right = 0;

    for(int i = 0; i < n; i++) {
        left = max(left, (long long)nums[i]); // Máximo elemento
        right += nums[i]; // Suma total
    }

    long long answer = right;

    // Binary search sobre la respuesta
```

```
    while(left <= right) {
        long long mid = left + (right - left) / 2;

        if(canSplit(nums, n, m, mid)) {
            answer = mid; // Esta suma máxima funciona
            right = mid - 1; // Intentamos una suma menor
        } else {
            left = mid + 1; // Necesitamos una suma mayor
        }
    }

    return answer;
}

int main() {
    int n, m;
    cout << "Enter array size and number of subarrays: ";
    cin >> n >> m;

    int* nums = new int[n];
    cout << "Enter array elements: ";
    for(int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    int result = splitArray(nums, n, m);
    cout << "Minimized largest sum: " << result << endl;

    delete[] nums;
    return 0;
}

// Ejemplo de uso:
// Input: n=5, m=2, nums=[7,2,5,10,8]
// Output: 18
// Explicación: [7,2,5] y [10,8], máximo = 18
```

Ejercicios con HASH

Problema 1: Sistema de Detección de Plagio en Ensayos

Contexto: Una universidad ha implementado un sistema automatizado para detectar plagio en ensayos estudiantiles. El sistema debe comparar el ensayo de un estudiante contra una base de datos de ensayos previos para encontrar si existe alguna sección copiada. Para considerarse plagio, debe haber una coincidencia de al menos K palabras consecutivas idénticas.

Problema: Dado el ensayo del estudiante S y N ensayos en la base de datos D_1, D_2, ..., D_n, junto con un umbral K, determinar:

1. Si existe plagio (alguna coincidencia de K o más palabras consecutivas)
2. La longitud máxima de texto copiado encontrado
3. El índice del ensayo de donde se copió la sección más larga

Entrada:

- Primera línea: Tres enteros N, M, K
  - N = número de ensayos en la base de datos ($1 \leq N \leq 100$)
  - M = número de palabras en el ensayo del estudiante ($1 \leq M \leq 10{,}000$)
  - K = umbral mínimo de palabras para considerar plagio ($1 \leq K \leq M$)
- Segunda línea: M palabras separadas por espacio (ensayo del estudiante S)
- Siguientes N líneas: Cada línea contiene un ensayo de la base de datos (cada uno con hasta 10,000 palabras)

Salida:

- Primera línea: "PLAGIO" o "ORIGINAL"
- Si hay plagio:
  - Segunda línea: La longitud máxima de coincidencia encontrada
  - Tercera línea: El índice del ensayo (1-indexed) de donde se copió más texto

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

const long long MOD = 1e9 + 9;
const long long P = 31;

template<typename T>
class HashSet {
private:
   struct Node {
      T value;
      Node* next;
      Node(T v) : value(v), next(nullptr) {}
   };

   Node** buckets;
   int capacity;
   int size;

   int hashFunction(T value) {
      return ((long long)value % capacity + capacity) % capacity;
   }
```

```
public:
    HashSet(int cap = 10007) : capacity(cap), size(0) {
        buckets = new Node*[capacity];
        for(int i = 0; i < capacity; i++) {
            buckets[i] = nullptr;
        }
    }

    ~HashSet() {
        for(int i = 0; i < capacity; i++) {
            Node* current = buckets[i];
            while(current) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
        }
        delete[] buckets;
    }

    void insert(T value) {
        int idx = hashFunction(value);
        Node* current = buckets[idx];

        while(current) {
            if(current->value == value) return;
            current = current->next;
        }

        Node* newNode = new Node(value);
        newNode->next = buckets[idx];
        buckets[idx] = newNode;
        size++;
    }

    bool contains(T value) {
        int idx = hashFunction(value);
        Node* current = buckets[idx];

        while(current) {
            if(current->value == value) return true;
            current = current->next;
        }
        return false;
    }
```

```
    void clear() {
        for(int i = 0; i < capacity; i++) {
            Node* current = buckets[i];
            while(current) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
            buckets[i] = nullptr;
        }
        size = 0;
    }
};

// Precalcular potencias de P
long long* precomputePowers(int maxLen) {
    long long* powers = new long long[maxLen + 1];
    powers[0] = 1;
    for(int i = 1; i <= maxLen; i++) {
        powers[i] = (powers[i-1] * P) % MOD;
    }
    return powers;
}

// Calcular hash de una secuencia de palabras
long long computeHash(int* words, int start, int length, long long* powers) {
    long long hash = 0;
    for(int i = 0; i < length; i++) {
        hash = (hash + ((long long)words[start + i] * powers[i]) % MOD) % MOD;
    }
    return hash;
}

// Verificar si existe coincidencia de longitud len
bool hasMatch(int* student, int studentLen, int** database, int* dbLengths, int
n, int len, long long* powers) {
    HashSet<long long> studentHashes;

    // Insertar todos los hashes del ensayo del estudiante
    for(int i = 0; i <= studentLen - len; i++) {
        long long hash = computeHash(student, i, len, powers);
        studentHashes.insert(hash);
    }

    // Verificar contra cada ensayo de la base de datos
    for(int essay = 0; essay < n; essay++) {
        for(int i = 0; i <= dbLengths[essay] - len; i++) {
```

```
            long long hash = computeHash(database[essay], i, len, powers);
            if(studentHashes.contains(hash)) {
                return true;
            }
        }
    }

    return false;
}

int main() {
    int n, m, k;
    cin >> n >> m >> k;

    // Leer ensayo del estudiante
    int* student = new int[m];
    for(int i = 0; i < m; i++) {
        cin >> student[i];
    }

    // Leer ensayos de la base de datos
    int** database = new int*[n];
    int* dbLengths = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> dbLengths[i];
        database[i] = new int[dbLengths[i]];
        for(int j = 0; j < dbLengths[i]; j++) {
            cin >> database[i][j];
        }
    }

    // Precalcular potencias
    long long* powers = precomputePowers(m);

    // Búsqueda binaria sobre la longitud de coincidencia
    int left = k, right = m;
    int maxMatch = 0;

    while(left <= right) {
        int mid = left + (right - left) / 2;

        if(hasMatch(student, m, database, dbLengths, n, mid, powers)) {
            maxMatch = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
```

```
        }
    }

    if(maxMatch >= k) {
        cout << "PLAGIO" << endl;
        cout << maxMatch << endl;

        // Encontrar de qué ensayo se copió más
        // (simplificado - buscar el primer match de longitud maxMatch)
        for(int essay = 0; essay < n; essay++) {
            bool found = false;
            for(int i = 0; i <= dbLengths[essay] - maxMatch && !found; i++) {
                long long dbHash = computeHash(database[essay], i, maxMatch,
powers);
                for(int j = 0; j <= m - maxMatch && !found; j++) {
                    long long stuHash = computeHash(student, j, maxMatch, powers);
                    if(dbHash == stuHash) {
                        cout << essay + 1 << endl;
                        found = true;
                    }
                }
            }
            if(found) break;
        }
    } else {
        cout << "ORIGINAL" << endl;
    }

    // Liberar memoria
    delete[] student;
    delete[] powers;
    delete[] dbLengths;
    for(int i = 0; i < n; i++) {
        delete[] database[i];
    }
    delete[] database;

    return 0;
}
```

Problema 2: Verificación de Integridad de Archivos Distribuidos

Contexto: Una empresa de almacenamiento en la nube divide archivos grandes en bloques para distribuirlos en múltiples servidores. Cada bloque debe tener un hash único para verificar su integridad. El sistema necesita detectar bloques duplicados para optimizar el almacenamiento (deduplicación) y también verificar si un archivo ha sido modificado comparando los hashes de sus bloques.

Problema: Dado un archivo original dividido en bloques de tamaño B bytes y un archivo potencialmente modificado, determinar:

1. Cuántos bloques son idénticos entre ambos archivos
2. La posición del primer bloque modificado (si existe)
3. Cuántos bloques únicos existen en el archivo original (para deduplicación)

Entrada:

- Primera línea: Dos enteros N y B
  - N = longitud del archivo en bytes ($1 \leq N \leq 1{,}000{,}000$)
  - B = tamaño del bloque en bytes ($1 \leq B \leq 1{,}000$)
- Segunda línea: String de N caracteres (archivo original)
- Tercera línea: Entero M (longitud del archivo modificado, $1 \leq M \leq 1{,}000{,}000$)
- Cuarta línea: String de M caracteres (archivo potencialmente modificado)

Salida:

- Primera línea: Número de bloques idénticos entre ambos archivos
- Segunda línea: Posición (1-indexed) del primer bloque modificado, o -1 si son idénticos
- Tercera línea: Número de bloques únicos en el archivo original

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

const long long MOD = 1e9 + 9;
const long long P = 257; // Para caracteres ASCII

template<typename T>
class HashSet {
private:
   struct Node {
      T value;
      Node* next;
      Node(T v) : value(v), next(nullptr) {}
   };

   Node** buckets;
   int capacity;
   int size;

   int hashFunction(T value) {
      return ((long long)value % capacity + capacity) % capacity;
   }

public:
```

```
    HashSet(int cap = 10007) : capacity(cap), size(0) {
        buckets = new Node*[capacity];
        for(int i = 0; i < capacity; i++) {
            buckets[i] = nullptr;
        }
    }

    ~HashSet() {
        for(int i = 0; i < capacity; i++) {
            Node* current = buckets[i];
            while(current) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
        }
        delete[] buckets;
    }

    void insert(T value) {
        int idx = hashFunction(value);
        Node* current = buckets[idx];

        while(current) {
            if(current->value == value) return;
            current = current->next;
        }

        Node* newNode = new Node(value);
        newNode->next = buckets[idx];
        buckets[idx] = newNode;
        size++;
    }

    bool contains(T value) {
        int idx = hashFunction(value);
        Node* current = buckets[idx];

        while(current) {
            if(current->value == value) return true;
            current = current->next;
        }
        return false;
    }

    int getSize() const { return size; }
};
```

```
// Precalcular potencias de P
long long* precomputePowers(int maxLen) {
   long long* powers = new long long[maxLen + 1];
   powers[0] = 1;
   for(int i = 1; i <= maxLen; i++) {
      powers[i] = (powers[i-1] * P) % MOD;
   }
   return powers;
}

// Calcular hash de un bloque usando rolling hash
long long computeBlockHash(const char* str, int start, int blockSize, long long*
powers) {
   long long hash = 0;
   for(int i = 0; i < blockSize; i++) {
      hash = (hash + ((long long)(unsigned char)str[start + i] * powers[i]) % MOD)
% MOD;
   }
   return hash;
}

int main() {
   int n, b;
   cin >> n >> b;
   cin.ignore(); // Ignorar el newline después de los números

   // Leer archivo original
   char* original = new char[n + 1];
   cin.getline(original, n + 1);
   int originalLen = strlen(original);

   // Leer archivo modificado
   int m;
   cin >> m;
   cin.ignore();
   char* modified = new char[m + 1];
   cin.getline(modified, m + 1);
   int modifiedLen = strlen(modified);

   // Precalcular potencias
   long long* powers = precomputePowers(b);

   // Calcular número de bloques
   int originalBlocks = (originalLen + b - 1) / b;
   int modifiedBlocks = (modifiedLen + b - 1) / b;
```

```
    // Calcular hashes de bloques originales
    long long* originalHashes = new long long[originalBlocks];
    HashSet<long long> uniqueHashes;

    for(int i = 0; i < originalBlocks; i++) {
        int start = i * b;
        int size = min(b, originalLen - start);
        originalHashes[i] = computeBlockHash(original, start, size, powers);
        uniqueHashes.insert(originalHashes[i]);
    }

    // Calcular hashes de bloques modificados y comparar
    int identicalBlocks = 0;
    int firstModified = -1;
    int minBlocks = min(originalBlocks, modifiedBlocks);

    for(int i = 0; i < minBlocks; i++) {
        int start = i * b;
        int size = min(b, modifiedLen - start);
        long long modifiedHash = computeBlockHash(modified, start, size, powers);

        if(i < originalBlocks && modifiedHash == originalHashes[i]) {
            identicalBlocks++;
        } else if(firstModified == -1) {
            firstModified = i + 1; // 1-indexed
        }
    }

    // Si uno de los archivos tiene más bloques, el primer bloque extra es
modificado
    if(originalBlocks != modifiedBlocks && firstModified == -1) {
        firstModified = minBlocks + 1;
    }

    // Output
    cout << identicalBlocks << endl;
    cout << firstModified << endl;
    cout << uniqueHashes.getSize() << endl;

    // Liberar memoria
    delete[] original;
    delete[] modified;
    delete[] powers;
    delete[] originalHashes;

    return 0;
}
```

```
/* Ejemplo de uso:
Input:
20 5
ABCDEABCDEFGHIJKLMNO
20
ABCDEABCDEXGHIJKLMNO

Output:
3
3
4

Explicación:
Bloques originales: ABCDE, ABCDE, FGHIJ, KLMNO
Bloques modificados: ABCDE, ABCDE, XGHIJ, KLMNO
Idénticos: 3 (posiciones 1, 2, 4)
Primer modificado: 3
Únicos: 3 (ABCDE se cuenta una vez, FGHIJ, KLMNO)
*/
```