

Ejercicio 1: Se cuenta con un archivo directo llamado ALUMNOS.DAT con información de alumnos de una universidad. El archivo se abre para lectura y se indexa por medio de su clave primaria (#legajo). El índice se mantiene en memoria principal. Se ofrece la posibilidad al usuario de buscar alumnos por legajo. La búsqueda binaria se aplica sobre el índice y se accede en forma directa (seek) al archivo para recuperar el registro. Se asume que el archivo está ordenado de manera ascendente por su clave primaria (#legajo). Si éste no fuese el caso, el índice debería ordenarse antes de aplicar una búsqueda binaria sobre él. Aclaración: no se provee el archivo ALUMNOS.DAT (debido a que es binario). Deberá ser generado antes de la ejecución de este programa).

1. ANÁLISIS DEL PROBLEMA

Tenemos:

- Un archivo binario ALUMNOS.DAT, donde cada registro guarda datos de alumnos.
- Cada alumno tiene un legajo (ID) que es la clave primaria.
- El archivo está ordenado por legajo.
- Antes de buscar, cargamos un índice en memoria: un vector con todos los legajos y sus posiciones en el archivo.
- El usuario ingresa un legajo y hacemos:

Búsqueda binaria sobre el índice

Esto es rápido:

$O(\log n)$

Si se encuentra, usamos:

Acceso directo a archivo (seekg)

Para leer el registro de la posición correcta.

¿Qué contiene el índice?

Cada entrada tiene:

legajo – posición en el archivo

Ejemplo:

Legajo Offset en archivo

100	0
150	1
220	2

Si busco 150 → está en offset 1 → “ir al registro 1 del archivo”.

Estructura del archivo (asumida)

```
struct Alumno {  
    int legajo;  
    char nombre[30];  
    int edad;  
};
```

2. CÓDIGO C++

Incluye:

- creación del archivo (para que funcione)
- generación del índice
- búsqueda binaria
- lectura directa desde archivo

```
#include <bits/stdc++.h>  
using namespace std;  
  
struct Alumno {  
    int legajo;  
    char nombre[30];  
    int edad;  
};  
  
int main() {  
    // ----- 1. Crear archivo (solo primera vez) -----  
    {  
        ofstream f("ALUMNOS.DAT", ios::binary);  
        Alumno a[] = {  
            {100, "Juan", 20},  
            {150, "Maria", 22},  
            {220, "Pedro", 19},  
            {300, "Lucia", 21}  
        };  
        for (auto &x : a) f.write((char*)&x, sizeof(Alumno));  
    }  
  
    // ----- 2. Cargar índice en memoria -----  
    vector<int> indice;  
    ifstream f("ALUMNOS.DAT", ios::binary);  
  
    f.seekg(0, ios::end);  
    int n = f.tellg() / sizeof(Alumno);  
    f.seekg(0);  
  
    for (int i = 0; i < n; i++) {  
        Alumno a;  
        f.read((char*)&a, sizeof(Alumno));  
        indice.push_back(a.legajo);  
    }  
  
    // ----- 3. Búsqueda binaria en el índice -----  
    int buscado;  
    cout << "Legajo a buscar: ";  
    cin >> buscado;
```

```

        int pos = lower_bound(indice.begin(), indice.end(), buscado) -
indice.begin();

        if (pos == n || indice[pos] != buscado) {
            cout << "No encontrado\n";
            return 0;
        }

        // ----- 4. Acceso directo al archivo -----
f.clear();
f.seekg(pos * sizeof(Alumno));

Alumno a;
f.read((char*)&a, sizeof(Alumno));

cout << "ENCONTRADO:\n";
cout << "Legajo: " << a.legajo << "\n";
cout << "Nombre: " << a.nombre << "\n";
cout << "Edad:    " << a.edad << "\n";

return 0;
}

```

Ejercicio 2: Localice un entero en un arreglo usando una versión recursiva del algoritmo de búsqueda binaria. El número que se debe buscar (clave) se debe ingresar como argumento por línea de comandos. El arreglo de datos debe estar previamente ordenado (requisito de la búsqueda binaria).

Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

1. ANÁLISIS DEL PROBLEMA

Debemos:

Tener un arreglo ordenado.

Tomar un número clave desde la línea de comandos.

Implementar búsqueda binaria recursiva.

Compararla con la versión iterativa.

¿Qué hace la búsqueda binaria?

Divide el arreglo en mitades y descarta la mitad donde la clave no puede estar.

2. VERSIÓN RECURSIVA (solicitada)

```

int binariaRec(int a[], int ini, int fin, int clave) {

    if (ini > fin) return -1;

    int mid = (ini + fin) / 2;

```

```

    if (a[mid] == clave) return mid;

    if (clave < a[mid])

        return binariaRec(a, ini, mid - 1, clave);

    else

        return binariaRec(a, mid + 1, fin, clave);

}

```

3. VERSIÓN ITERATIVA (para comparar)

```

int binariaIt(int a[], int n, int clave) {

    int ini = 0, fin = n - 1;

    while (ini <= fin) {

        int mid = (ini + fin) / 2;

        if (a[mid] == clave) return mid;

        if (clave < a[mid]) fin = mid - 1;

        else ini = mid + 1;

    }

    return -1;
}

```

4. PROGRAMA COMPLETO

Se ejecuta con:

./programa 25

```

#include <iostream>

using namespace std;

int binariaRec(int a[], int ini, int fin, int clave) {

    if (ini > fin) return -1;

    int mid = (ini + fin) / 2;

```

```

    if (a[mid] == clave) return mid;
    if (clave < a[mid]) return binariaRec(a, ini, mid-1, clave);
    return binariaRec(a, mid+1, fin, clave);
}

int binariaIt(int a[], int n, int clave) {
    int ini = 0, fin = n-1;
    while (ini <= fin) {
        int mid = (ini + fin)/2;
        if (a[mid] == clave) return mid;
        if (clave < a[mid]) fin = mid-1;
        else ini = mid+1;
    }
    return -1;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        cout << "Debe ingresar la clave.\n";
        return 0;
    }

    int clave = atoi(argv[1]);
    int a[] = {3, 8, 12, 17, 25, 30, 45};
    int n = sizeof(a)/sizeof(a[0]);

    int r1 = binariaRec(a, 0, n-1, clave);
    int r2 = binariaIt(a, n, clave);
}

```

```

cout << "Recursiva: " << r1 << "\n";
cout << "Iterativa: " << r2 << "\n";
}

```

Ejercicio 3: Diseñe una variación de Búsqueda Binaria (algoritmo 1.4) que efectúe sólo una comparación binaria (es decir, la comparación devuelve un resultado booleano) de K con un elemento del arreglo cada vez que se invoca la función. Pueden hacerse comparaciones adicionales con variables de intervalo. Analice la corrección de su procedimiento. Sugerencia: ¿Cuándo deberá ser de igualdad (==) la única comparación que se hace?

1. ANÁLISIS DEL PROBLEMA

Debemos diseñar una variación de Búsqueda Binaria con esta restricción importante:

Solo se permite UNA comparación binaria entre K y un elemento del arreglo:

$A[\text{mid}] == K$

Se prohíbe usar:

$A[\text{mid}] < K$

$A[\text{mid}] > K$

Sí se permiten comparaciones con variables de intervalo (low, high, mid) y con la clave K.

Entonces, el gran problema es:

En la búsqueda binaria normal, se decide:

¿Voy a la izquierda?

¿Voy a la derecha?

Comparando:

si $A[\text{mid}] < K \Rightarrow$ derecha

si $A[\text{mid}] > K \Rightarrow$ izquierda

Aquí eso NO está permitido.

¿Cómo resolverlo? (Idea del enunciado)

Solo podemos usar:

if ($A[\text{mid}] == K$)

Entonces, ¿cómo decidir izquierda/derecha?

Comparando K con mid

NO con el arreglo, pero sí con el índice.

Para que esto funcione, el arreglo debe estar compuesto de valores conocidos y crecientes, donde:

A[i]=*io una función creciente conocida*
A[i] = i \quad o una función creciente conocida
A[i]=*io una función creciente conocida*

Ejemplo válido:

0, 1, 2, 3, 4, 5...

Ejemplos NO válidos (no funciona la técnica):

10, 14, 20, 21, 30

7, 8, 19, 110, 500

Porque si A[mid] ≠ K, no podemos saber hacia dónde movernos sin mirar su valor.

REGLA DEL ALGORITMO

Única comparación permitida con el arreglo:

A[mid] == K

Decisión izquierda/derecha:

solo comparando

K con mid.

2. CÓDIGO C++

Arreglo asumido: A[i] = i, estrictamente creciente.

```
#include <iostream>
```

```
using namespace std;
```

```
// Búsqueda binaria con solo 1 comparación con el arreglo
```

```
int busquedaRestrictiva(int A[], int low, int high, int K) {
```

```
    if (low > high) return -1;
```

```
    int mid = (low + high) / 2;
```

```

// ÚNICA comparación permitida con elementos del arreglo

if (A[mid] == K)

    return mid;

// Decisión usando SOLO índices (permitido)

if (K < mid)

    return busquedaRestrictiva(A, low, mid - 1, K);

else

    return busquedaRestrictiva(A, mid + 1, high, K);

}

int main() {

int A[] = {0,1,2,3,4,5,6,7,8,9};

int n = sizeof(A)/sizeof(A[0]);

int K;

cout << "Ingrese K: ";

cin >> K;

int pos = busquedaRestrictiva(A, 0, n-1, K);

if (pos == -1) cout << "No encontrado\n";

else cout << "Encontrado en posicion: " << pos << "\n";

}

```

4. ANÁLISIS DE CORRECCIÓN

Por qué funciona

Sea el arreglo:

$$A[i] = i \quad A[i] = i$$

Entonces:

Si $A[\text{mid}] == K \rightarrow \text{encontrado.}$

Si $A[\text{mid}] != K:$

Si $K < \text{mid}$, entonces forzosamente $K < A[\text{mid}]$, así que debe estar a la izquierda.

Si $K > \text{mid}$, entonces forzosamente $K > A[\text{mid}]$, así que debe estar a la derecha.

Esto permite reemplazar la comparación:

$$K < A[\text{mid}]$$

por:

$$K < \text{mid}$$

sin romper la corrección.

¿Cuándo NO funciona?

Si el arreglo NO cumple $A[i] = i$ o no es creciente conocido.

Ejemplo:

5, 10, 15, 20

Si $K = 17$, $\text{mid} = 1$, $A[\text{mid}] = 10$:

$$A[\text{mid}] != 17$$

$\hat{K} < A[\text{mid}]$? NO puedo preguntar.

$\hat{K} < \text{mid} = 1$? Falso, pero 17 no está a la derecha del índice 1, sino a la derecha de $A[\text{mid}] = 10$.

El algoritmo fallaría.

CONCLUSIÓN FINAL

Solo se puede diseñar una búsqueda binaria con una sola comparación == por llamada si:

El arreglo cumple $A[i] = i$ o es una función estrictamente creciente conocida.

La decisión izquierda/derecha se hace solo con índices, no con datos.

Su corrección depende de esta estructura.

Ejercicio 4: ¿Cómo podría modificar Búsqueda Binaria (¿algoritmo 1.4) para eliminar trabajo innecesario si se tiene la certeza de que K está en el arreglo? Dibuje un árbol de decisión para el algoritmo modificado con $n = 7$. Efectúe análisis de comportamiento promedio y de peor caso. (Para el promedio, puede suponerse que $n = 2 - 1$ para alguna k .)

1. ANÁLISIS DEL ALGORITMO

En la búsqueda binaria normal se realizan varias comparaciones en cada iteración:

1. Comparar si $A[mid] == K$.
2. Comparar si $A[mid] < K$ para decidir si se va a la izquierda o derecha.
3. Verificar la condición del bucle $low \leq high$.

Cuando se sabe de antemano que K está en el arreglo, no es necesario comprobar si la búsqueda ha fallado (porque no fallará). Por lo tanto, la verificación $low \leq high$ no es necesaria. También si $A[mid]$ no es menor que K, entonces necesariamente K está en la izquierda y no hay que evaluar igualdad explícitamente en ese caso.

El número de comparaciones por iteración se reduce porque ya no hay que verificar que el rango siga siendo válido. Las decisiones se basan únicamente en comparar $A[mid]$ con K.

Complejidad: el algoritmo sigue siendo $\log n$, pero con menos comparaciones por nivel.

2. ALGORITMO MODIFICADO

Idea:

Mientras no se encuentre K, comparar $A[mid]$ con K.

Si $A[mid] < K \rightarrow$ moverse a la derecha.

Si $A[mid] > K \rightarrow$ moverse a la izquierda.

Como K está en el arreglo, necesariamente se llegará a $A[mid] == K$.

3. CÓDIGO C++

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

int busquedaAsumida(int A[], int n, int K) {
    int low = 0, high = n - 1;

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

        if (A[mid] == K) return mid;

        if (A[mid] < K) low = mid + 1;
        else high = mid - 1;
    }
}
```

4. ÁRBOL DE DECISIÓN PARA $n = 7$

Arreglo ordenado de 7 elementos (índices 0 a 6):



Primer mid: índice 3

Segundo nivel: 1 y 5

Tercer nivel: 0, 2, 4, 6

Como se sabe que K está en el arreglo, todas las hojas son posibles soluciones, no hay casos de "no encontrado".

5. ANÁLISIS DEL COMPORTAMIENTO

Peor caso:

El elemento buscado está en el nivel más profundo del árbol.

Profundidad = $\log_2(7) \approx 3$

Peor caso = $O(\log n)$

Promedio:

Suponiendo $n = 2^k - 1$ para que el árbol sea completo.

La profundidad promedio es aproximadamente:

$\log_2(n+1) - 2$

El promedio es un poco menor que el peor caso porque hay más nodos en niveles intermedios que en los niveles más profundos.

Ejercicio 5: Se cuenta con una lista de Pokémons, de cada uno de estos se sabe su nombre, número y tipo (solo considerar uno el principal), con los cuales deberá resolver las siguientes tareas: determinar si existe el Pokémon Cobalion y mostrar toda su información.

ANÁLISIS

Se tiene una lista de Pokémons donde cada Pokémon posee:

1. nombre
2. número en la Pokédex
3. tipo principal

La tarea es determinar si existe el Pokémon llamado Cobalion y, en caso afirmativo, mostrar toda su información.

Hay dos posibles formas de búsqueda:

- a) Búsqueda lineal si la lista no está ordenada.
- b) Búsqueda binaria si está ordenada por nombre.

Como no se especifica que la lista esté ordenada, la solución más segura es la búsqueda lineal, recorriendo elemento por elemento hasta encontrar Cobalion.

Complejidad:

$O(n)$, donde n es la cantidad de Pokémons.

CÓDIGO C++ (VERSIÓN CORTA)

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

struct Pokemon {
    string nombre;
    int numero;
    string tipo;
};

int main() {
    vector<Pokemon> pokedex = {
        {"Pikachu", 25, "Electrico"}, 
        {"Cobalion", 638, "Acero"}, 
        {"Charmander", 4, "Fuego"} 
    };

    bool encontrado = false;

    for (auto &p : pokedex) {
        if (p.nombre == "Cobalion") {
            cout << "Nombre: " << p.nombre << endl;
            cout << "Numero: " << p.numero << endl;
            cout << "Tipo: " << p.tipo << endl;
            encontrado = true;
            break;
        }
    }

    if (!encontrado)
        cout << "Cobalion no esta en la lista" << endl;
}

return 0;
}
```

EJEMPLO DE FUNCIONAMIENTO

Lista de Pokémons:

Pikachu (25, Eléctrico)

Cobalion (638, Acero)

Charmander (4, Fuego)

Se recorre la lista:

1. Pikachu → no coincide
2. Cobalion → coincide

Salida esperada:

Nombre: Cobalion

Numero: 638

Tipo: Acero

Ejercicio 6: Se dispone de la lista de superhéroes y villanos de la saga de Marvel Cinematic Universe (MCU) de los que contamos con la información de nombre del personaje y año de la primera película en la que apareció; a partir de estos resolver las siguientes actividades: indicar quien fue el primer y el último personaje en aparecer en una película sin realizar un recorrido de la lista (podrían ser más de uno tanto el primero como el último).

ANÁLISIS

Se tiene una lista de personajes del MCU, cada uno con:

1. nombre
2. año de su primera aparición en una película

La tarea pide:

Determinar quién fue el primero y quién fue el último en aparecer, sin recorrer la lista para buscarlos.

Esto significa que:

La lista ya debe venir ordenada por año de aparición.

Si la lista está ordenada de forma ascendente:

El o los primeros personajes estarán al inicio de la lista.

El o los últimos personajes estarán al final de la lista.

Pero puede haber varios personajes con el mismo año inicial o final.

Por eso, una vez que se conoce el primer año y último año, se toman todos los personajes que coincidan con esos años.

Complejidad:

Se asume que no se busca el mínimo ni el máximo recorriendo todo, porque ya está ordenado. Solo se revisan consecutivos hasta que el año cambia.

CÓDIGO C++ (VERSIÓN CORTA)

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

struct Personaje {
    string nombre;
    int anio;
};

int main() {
    vector<Personaje> mcu = {
        {"Iron Man", 2008},
        {"Hulk", 2008},
        {"Black Widow", 2010},
        {"Thor", 2011},
    };
}
```

```

        {"Captain Marvel", 2019}
    };

    int primer = mcu.front().anio;
    int ultimo = mcu.back().anio;

    cout << "Primeros personajes:\n";
    for (auto &p : mcu)
        if (p.anio == primer)
            cout << p.nombre << " (" << p.anio << ") \n";

    cout << "Ultimos personajes:\n";
    for (auto &p : mcu)
        if (p.anio == ultimo)
            cout << p.nombre << " (" << p.anio << ") \n";

    return 0;
}

```

EJEMPLO DE FUNCIONAMIENTO

Lista (ya ordenada por año):

Iron Man – 2008

Hulk – 2008

Black Widow – 2010

Thor – 2011

Captain Marvel – 2019

Primer año = 2008

Último año = 2019

Resultado:

Primeros personajes:

Iron Man (2008)

Hulk (2008)

Ultimos personajes:

Captain Marvel (2019)

Ejercicio 7: Se dispone de una lista de películas con la siguiente información: título, año de estreno, recaudación y valoración del público (de 1 a 5), los cuales debemos procesar contemplando las siguientes tareas: determinar si la película “Avengers: Infinity War” está en la lista y mostrar toda su información; indicar en qué posición se encuentra la película “Star Wars: The Return of Jedi”.

ANÁLISIS

Se tiene una lista de películas donde cada película posee:

1. título
2. año de estreno
3. recaudación

4. valoración del público (1 a 5)

Tareas:

- Determinar si la película "Avengers: Infinity War" está en la lista y mostrar toda su información.
- Indicar en qué posición se encuentra "Star Wars: The Return of Jedi".

Dado que la lista no necesariamente está ordenada, se utiliza búsqueda lineal.

Cada elemento se compara con el título buscado.

La complejidad es O(n).

CÓDIGO C++ (VERSIÓN CORTA)

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

struct Pelicula {
    string titulo;
    int anio;
    long long recaudacion;
    int valoracion;
};

int main() {
    vector<Pelicula> lista = {
        {"Avengers: Endgame", 2019, 2797800564, 5},
        {"Avengers: Infinity War", 2018, 2048359754, 5},
        {"Star Wars: The Return of Jedi", 1983, 475000000, 4},
        {"Inception", 2010, 829895144, 5}
    };

    bool estaInfinity = false;

    for (auto &p : lista) {
        if (p.titulo == "Avengers: Infinity War") {
            cout << "Pelicula encontrada:\n";
            cout << "Titulo: " << p.titulo << endl;
            cout << "Anio: " << p.anio << endl;
            cout << "Recaudacion: " << p.recaudacion << endl;
            cout << "Valoracion: " << p.valoracion << endl;
            estaInfinity = true;
            break;
        }
    }

    if (!estaInfinity)
        cout << "Avengers: Infinity War no esta en la lista\n";

    int posicion = -1;
    for (int i = 0; i < lista.size(); i++) {
        if (lista[i].titulo == "Star Wars: The Return of Jedi") {
            posicion = i;
            break;
        }
    }

    if (posicion != -1)
```

```

        cout << "Posicion de Star Wars: The Return of Jedi: " <<
posicion << endl;
else
    cout << "Star Wars: The Return of Jedi no esta en la lista\n";
return 0;
}

```

EJEMPLO DE FUNCIONAMIENTO

La lista contiene:

Avengers: Endgame (2019)
 Avengers: Infinity War (2018)
 Star Wars: The Return of Jedi (1983)
 Inception (2010)

Búsqueda:

Avengers: Infinity War se encuentra en la segunda posición, se imprime toda su información.

Star Wars: The Return of Jedi está en la tercera posición (índice 2).

Salida esperada:

Pelicula encontrada:

Titulo: Avengers: Infinity War
 Anio: 2018
 Recaudacion: 2048359754
 Valoracion: 5

Posicion de Star Wars: The Return of Jedi: 2

Ejercicio 8: Encontrar el algoritmo de búsqueda binaria para encontrar un elemento K en una lista de elementos X, X, ..., X, previamente clasificados en orden ascendente.

El array o vector X se supone ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres. Las variables BAJO, CENTRAL, ALTO indican los límites inferior, central y superior del intervalo de búsqueda.

algoritmo busqueda_binaria

//declaraciones

inicio

//llenar (X,N)

//ordenar (X,N)

leer(K)

//inicializar variables

```

BAJO← 1
ALTO← N
CENTRAL ent ((BAJO + ALTO) / 2)
mientras (BAJO < ALTO) y (X[CENTRAL] <> K) hacer
    si K=X[CENTRAL] entonces
        ALTO
        si_no
            CENTRAL 1
        -
        BAJO CENTRAL + 1
        fin_si
        CENTRAL←ent ((BAJO + ALTO) / 2)
        fin_mientras
        si K = X(CENTRAL) entonces
            escribir('Valor encontrado en', CENTRAL)
        si_no
            escribir('Valor no encontrado')
        fin_si
    fin

```

ANÁLISIS

La búsqueda binaria sirve para encontrar un valor K dentro de un arreglo X ordenado de forma ascendente.

Se mantiene un intervalo de búsqueda definido por:
 BAJO → límite inferior
 ALTO → límite superior
 CENTRAL → posición media del intervalo

En cada paso se compara K con X[CENTRAL]:
 Si K es igual, se encontró.

Si K es menor, se busca en la mitad izquierda (ALTO = CENTRAL - 1).
Si K es mayor, se busca en la mitad derecha (BAJO = CENTRAL + 1).

Complejidad: O(log n).

Requisito: el arreglo debe estar ordenado previamente.

ALGORITMO EN PSEUDOCÓDIGO (CORREGIDO)

```
algoritmo busqueda_binaria
leer K
BAJO ← 1
ALTO ← N
CENTRAL ← ent((BAJO + ALTO) / 2)

mientras (BAJO <= ALTO) y (X[CENTRAL] <> K) hacer
    si K < X[CENTRAL] entonces
        ALTO ← CENTRAL - 1
    si_no
        BAJO ← CENTRAL + 1
    fin_si
    CENTRAL ← ent((BAJO + ALTO) / 2)
fin_mientras

si K = X[CENTRAL] entonces
    escribir("Valor encontrado en ", CENTRAL)
si_no
    escribir("Valor no encontrado")
fin_si

fin_algoritmo
```

VERSIÓN C++ CORTA

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

int busquedaBinaria(const vector<int>& X, int K) {
    int bajo = 0, alto = X.size() - 1;

    while (bajo <= alto) {
        int central = (bajo + alto) / 2;

        if (X[central] == K) return central;
        if (K < X[central]) alto = central - 1;
        else bajo = central + 1;
    }
    return -1;
}

int main() {
    vector<int> X = {3, 7, 12, 20, 25, 31, 40};
    int K;
    cin >> K;
```

```

int pos = busquedaBinaria(X, K);

if (pos != -1) cout << "Valor encontrado en " << pos;
else cout << "Valor no encontrado";
}

```

EJEMPLO DE FUNCIONAMIENTO

Arreglo:
[3, 7, 12, 20, 25, 31, 40]

Entrada:
20

Pasos:
bajo=0, alto=6 → central=3 → X[3]=20 → encontrado

Salida:
Valor encontrado en 3

Ejercicio 9: Un vector T tiene cien posiciones, 0.100. Supongamos que las claves de búsqueda de los elementos de la tabla son enteros positivos (por ejemplo, número del DNI).

Una función de conversión h debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango 0.100, esto es, h es una función tal que para un entero positivo x .

$$h(x) = n,$$

donde n es entero en el rango 0.100

El método del módulo, tomando 101, será

$$h(x) = x \bmod 101$$

Si se tiene el DNI número 234661234, por ejemplo, se tendrá la posición 56:

$$234661234 \bmod 101 = 56$$

ANÁLISIS

Se tiene una tabla o vector T de tamaño 101, índices desde 0 hasta 100. Cada elemento que se quiere almacenar en la tabla tiene una clave numérica positiva (por ejemplo, un DNI).

Como el DNI puede ser muy grande, necesitamos una función de conversión $h(x)$ que transforme cualquier clave x en un número entre 0 y 100.

El método más usado en hashing para esto es el método del módulo:
 $h(x) = x \bmod 101$

Esto garantiza que el resultado siempre estará entre 0 y 100.

Ejemplo:

$x = 234661234$

$h(x) = 234661234 \bmod 101 = 56$

Por lo tanto, el elemento se guarda en $T[56]$.

Explicación del porqué:

Si dividimos 234661234 entre 101, el residuo siempre será un entero entre 0 y 100.

Ese residuo se toma como dirección o posición en la tabla de dispersión.

CÓDIGO C++ (VERSIÓN CORTA)

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

int hashModulo(int x) {
    return x % 101;
}

int main() {
    int dni;
    cin >> dni;
    int pos = hashModulo(dni);
    cout << "La posicion es " << pos;
    return 0;
}
```

EJEMPLO DE FUNCIONAMIENTO

Entrada:

234661234

Cálculo:

$234661234 \bmod 101 = 56$

Salida:

La posicion es 56

Ejercicio 10: Un entero de ocho dígitos se puede dividir en grupos de tres, tres y dos dígitos, los grupos se suman juntos y se truncan si es necesario para que estén en el rango adecuado de índices.

Por consiguiente, si la clave es:

62538194

y el número de direcciones es 100, la función de conversión será

$$625+381+94=1100$$

que se truncará a 100 y que será la dirección deseada.

ANÁLISIS

Se tiene un entero de ocho dígitos.

La técnica propuesta consiste en:

1. Dividir el número en grupos de:
 - o los primeros 3 dígitos
 - o los siguientes 3 dígitos
 - o los últimos 2 dígitos
2. Sumar los tres grupos
3. Si la suma excede el número de direcciones disponibles (por ejemplo 100), se trunca al rango permitido de índices.

Ejemplo de clave:

62538194

División:

625 | 381 | 94

Suma:

$$625 + 381 + 94 = 1100$$

Como el número de direcciones es 100, se trunca a:

$$1100 \bmod 100 = 0$$

o

simplemente se limita a 100 si la tabla tiene índices 0–100.

En el enunciado:

1100 se trunca a 100, y esa será la dirección.

Este método pertenece a las funciones de hashing llamadas "métodos basados en suma de bloques".

CÓDIGO C++ (VERSIÓN CORTA)

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

int hashGrupos(int clave, int maxDir) {
    int g1 = clave / 1000000;           // primeros 3 dígitos
    int g2 = (clave / 1000) % 1000;     // siguientes 3 dígitos
    int g3 = clave % 100;              // últimos 2 dígitos

    int suma = g1 + g2 + g3;
    if (suma > maxDir) suma = maxDir; // truncar
    return suma;
}

int main() {
    int clave = 62538194;
    int pos = hashGrupos(clave, 100);
```

```
    cout << "Direccion: " << pos;  
    return 0;  
}
```

EJEMPLO DE FUNCIONAMIENTO

Entrada:

62538194

Cálculos:

g1 = 625

g2 = 381

g3 = 94

suma = $625 + 381 + 94 = 1100$

truncado = 100

Salida:

Direccion: 100