Tema VI: Búsqueda

Problema de la Búsqueda Estática. Interna y externa (métodos). Búsqueda Secuencial. Búsqueda de máximos y mínimos. Búsqueda Binaria. Búsqueda Ternaria. Búsqueda Interpolada. Búsqueda por transformación de claves (*Hassing*). Búsqueda fuerza bruta.

6.1 Problema de la Búsqueda Estática

La búsqueda estática es aquel **proceso de búsqueda efectuado sobre una estructura cuyo contenido no se altera a lo largo del tiempo**, es decir, se mantienen estáticos, de allí su nombre.

La operación de búsqueda nos permite encontrar datos que están previamente almacenados. La operación puede ser un éxito, si se localiza el elemento buscado o un fracaso en otros casos.

La búsqueda se puede realizar sobre un conjunto de datos ordenados, lo cual hace la tarea más fácil y consume menos tiempo; o se puede realizar sobre elementos desordenados, tarea más laboriosa y de mayor insumo de tiempo.

Definición: *La operación de búsqueda de un elemento X en un conjunto consiste en determinar si el elemento X pertenece al conjunto y en este caso dar su posición, o bien, determinar que el elemento X no pertenece al conjunto.*

Los métodos más usuales para la búsqueda son:

- *Búsqueda secuencial o lineal*
- *Búsqueda binaria*
- *Búsqueda por transformación de claves*

6.2 Tipos de búsqueda

- ***Búsqueda Interna*** será aquella acción que se realice sobre datos que se encuentran en la memoria principal, por ejemplo en un arreglo.

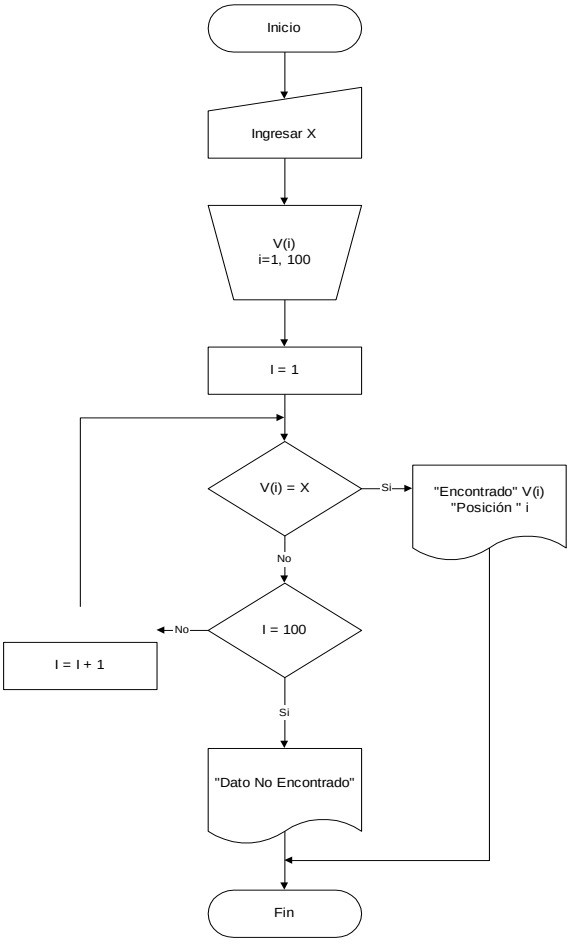
- ***Búsqueda Externa*** es cuando todos sus elementos se encuentran en memoria secundaria (archivos almacenados en dispositivos de cinta, disco, etc.-)

6.3 Búsqueda Secuencial

La búsqueda secuencial es la técnica más simple para buscar en una lista de datos. Este método consiste en recorrer una lista (o arreglo) en forma secuencial y comparar cada elemento del arreglo (o de la lista) con el valor deseado, hasta que éste se encuentre o finalice el arreglo (o la lista).

Normalmente cuando la función de búsqueda termina con éxito, es decir encontró el dato buscado, interesa conocer en qué posición fue encontrado el dato buscado. Esta idea se puede generalizar en todos los métodos de búsqueda.

La búsqueda secuencial no requiere ningún requisito para el arreglo, y por lo tanto no necesita estar ordenado.



Inicio
Ingresar X
Leer V(100)
Desde I = 1 hasta 100 **hacer**
  Si V(i) = X **entonces**
    Imprimir " Encontrado" V(i), "Posición" i
  Fin Si
Fin desde
Imprimir "Dato no encontrado"
Fin

El método es solo adecuado para listas cortas de datos.

A la hora de analizar la complejidad del método secuencial, tenemos que tener en cuenta el caso más favorable y el más desfavorable.

Cuando el elemento no se encuentra tiene que realizar las *n* comparaciones. Y en los casos en que el elemento buscado se localiza, este podrá estar en el primer lugar, en el último o en un lugar intermedio.

Entonces, al buscar un elemento en un arreglo de N componentes se harán:

- *N comparaciones* si el elemento no se localiza
- *N comparaciones* si el elemento está en la última posición
- *1 comparación* si está en el primer lugar
- *i comparaciones* si está en un lugar intermedio (posición 1 < i < N)

Podemos suponer que el número medio de comparaciones a realizar es de ***(n + 1) / 2***, que es aproximadamente igual a la mitad de los elementos de la lista.

6.4 Búsqueda de máximos y mínimos

En muchos casos, es necesario determinar el mayor o el menor valor de un conjunto de datos.

Existen diversos algoritmos para esta determinación, en la mayoría de ellos se realizan comparaciones sucesivas de todos y cada uno de los datos resguardando en una variable auxiliar el valor que resulte mayor o menor, de acuerdo a lo que se busque, de manera tal que cuando no existan mas datos para comparar, esta variable auxiliar contendrá el valor máximo o mínimo buscado.

Existen tres métodos para la resolución de este problema:

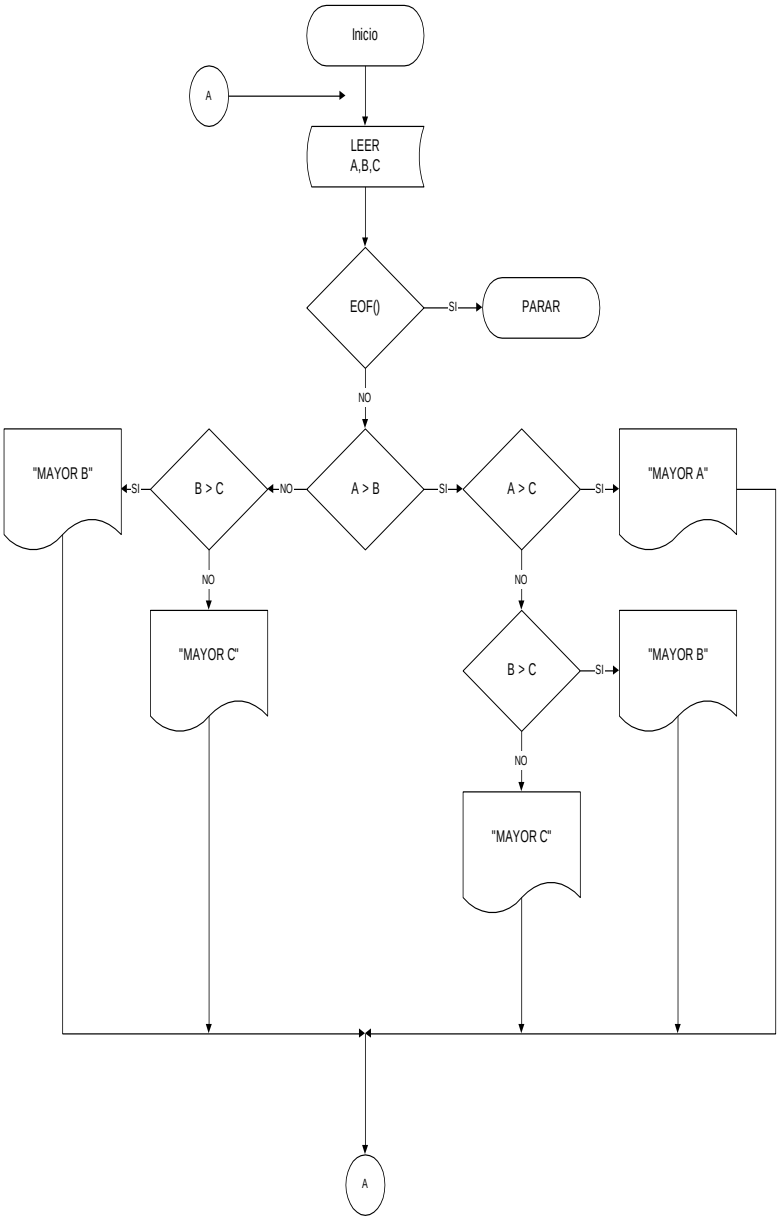
- *Ramificación del árbol*
- *Campeonato*
- *Supuesto o prepo*

6.4.1 Ramificación del árbol

Consiste en las combinaciones de comparaciones de todas las variables que intervienen.

Este método se realiza teniendo en cuenta que todos los campos deben estar simultáneamente en memoria (es del tipo de búsqueda interna).

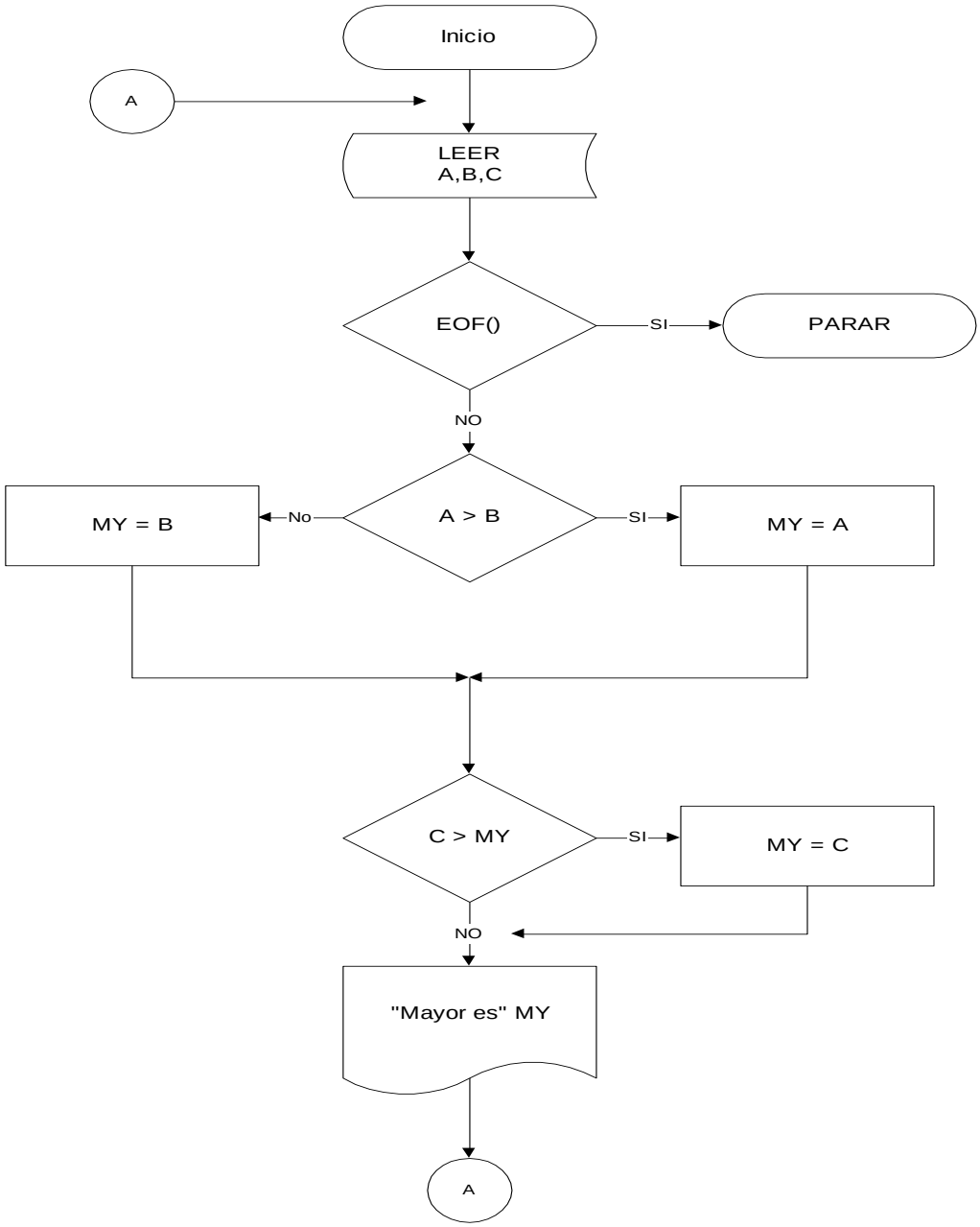
Este método es intuitivo y natural.



Comenzar
Mientras no EOF
Leer A,B,C
Si A>B
Entonces
Si A>C
Entonces
Imprimir “Mayor “A
Si_no
Si B>C
Entonces
Imprimir “Mayor “B
Si_no
Imprimir “Mayor “C
Fin_si
Fin_si
Si_no
Si B>C
Entonces
Imprimir “Mayor “B
Si_no
Imprimir “Mayor “C
Fin_si
Fin_si
Fin_mientras
Parar

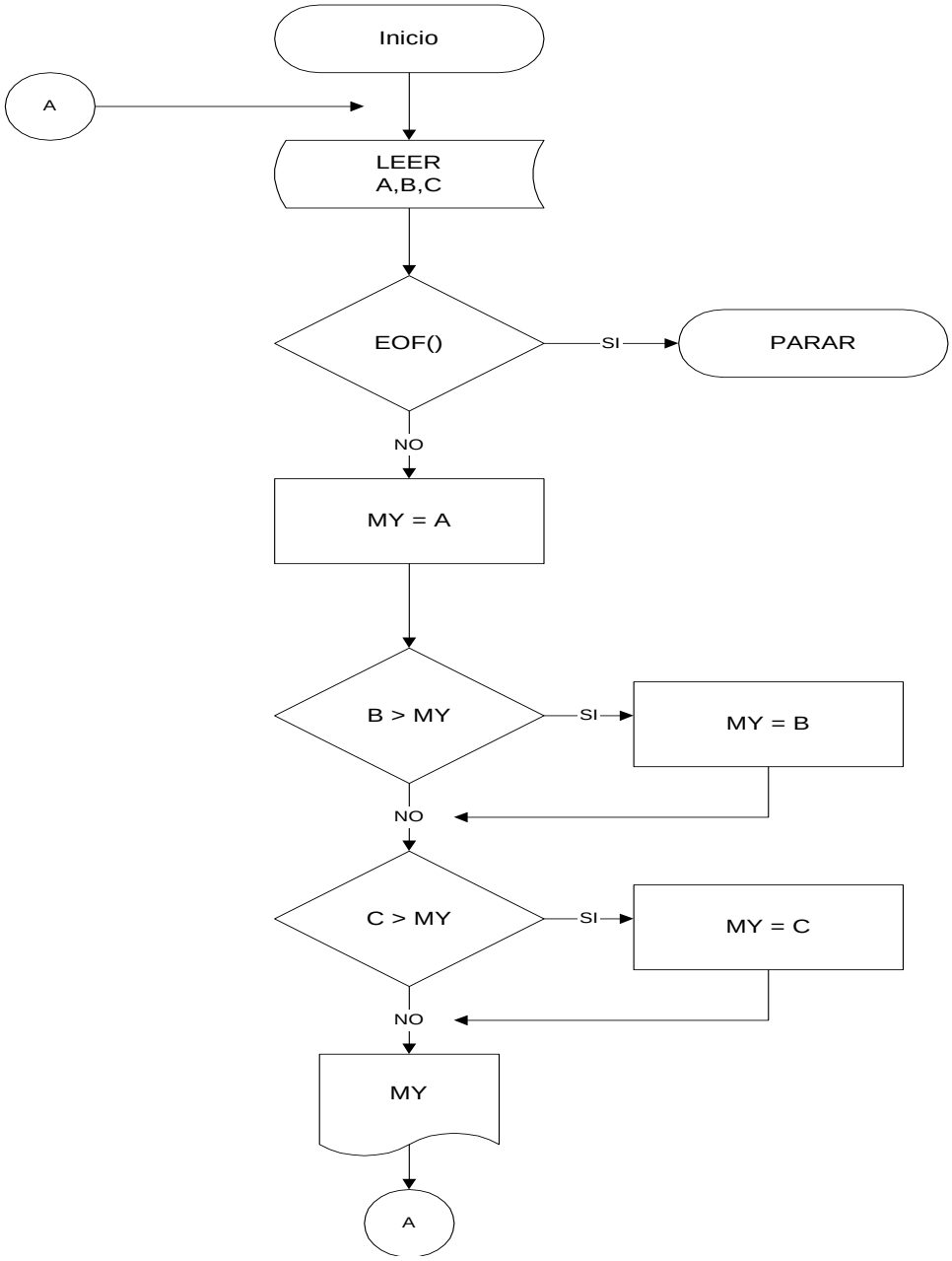
6.4.2 Campeonato

Consiste en la comparación **de a pares** de todas las variables que intervienen. En este método los campos también deben estar simultáneamente en memoria.



6.4.3 Supuesto o Prepo:

Es el que más utilizaremos a lo largo del curso y consiste en suponer que una de las variables que existen en memoria, en el mismo momento, es mayor o menor de todas, y luego se realiza las comparaciones sucesivas con las restantes. Este método se adapta para los algoritmos de *búsqueda externa* (los campos no están simultáneamente en memoria, sino que ingresan registro a registro).


Inicio
A
LEER
A,B,C
EOF()
SI
PARAR
NO
MY = A
B > MY
SI
MY = B
NO
C > MY
SI
MY = C
NO
MY
A

Ejemplo: Búsqueda del mayor elemento en un vector

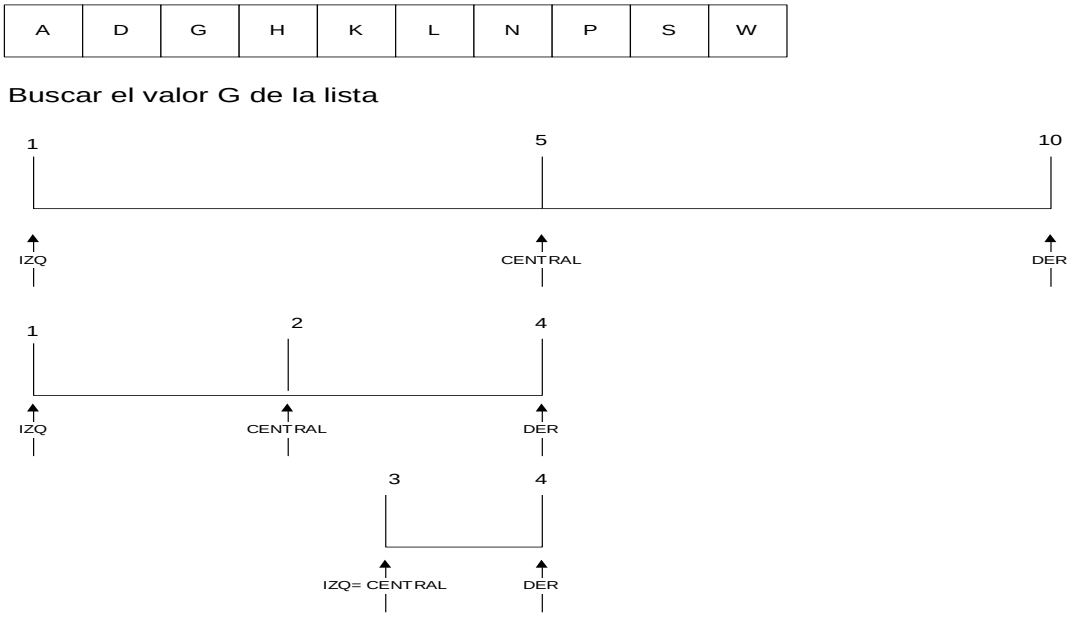
```
#include <stdio.h>
void main(){
    int vector[10];
    int i;
    int mayor=0;

   for (i=0; i<10; i++){
       printf( "ingrese 10 números enteros. Numero %d\n",
    i+1);
       scanf("%d", &vector[i]);
}

for (i=0; i<10; i++){
    if (vector[i]> mayor){
    mayor=vector[i];
    }
}
printf("El mayor es %d\n", mayor);
}
```

6.5 Búsqueda Binaria

Es válido exclusivamente para datos ordenados y consiste en comparar en primer lugar con la componente central de la lista, y si no es igual al valor buscado se reduce el intervalo de búsqueda a la mitad derecha o izquierda según donde pueda encontrarse el valor a buscar.

El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda queda anulado.

Este mecanismo es muy eficaz para buscar un elemento cualquiera que esté en una lista ordenada, y recibe el nombre de **Búsqueda Binaria o Dicotómica** cuya resolución se base en el algoritmo de divisiones sucesivas en mitades.



Con cada iteración del método el espacio de búsqueda se reduce a la mitad, por lo tanto el número de comparaciones disminuye considerablemente en cada iteración. Esta disminución es más significativa cuanto mayor sea el número de elementos de la lista.

En Pseudocódigo:

```
Comenzar
Leer X
    Ingresar V(I) I = 1,100
    ULTIMO = 100
    PRIMERO  = 1
    Hasta PRIMERO <= ULTIMO
            CENTRAL = [(PRIMERO + ULTIMO) / 2]
            Si V(CENTRAL) = X
                    Entonces
                            Imprimir “Registro encontrado “ V(CENTRAL)
                            Parar
            Fin_si
            Si V(CENTRAL) > X
                    Entonces
                            ULTIMO = CENTRAL – 1
                    Si_no
                            PRIMERO = CENTRAL + 1
            Fin_si
    Fin_Mientras
    Imprimir “Registro no encontrado”
Parar
```

```
En C:
#include<stdio.h>
int main() {
     int
A[20]={21,32,43,54,65,76,87,98,109,110,211,212,313,314,415,41
6,417,518,519,620};
     Int inf,sup,mit,dato,n=20;
     printf("dame un dato a buscarn: ");
     scanf("%d",&dato);
     inf=0;
     sup=n;
     while (inf<=sup) {
          mit=(inf+sup)/2;
          if (A[mit]==dato) {
          printf("dato %d encontrado posicion %d
\n",dato,mit);
          break;
          }
        if (A[mit]>dato) {
           sup=mit;
                  mit=(inf+sup)/2;
                   }
                if (A[mit]<dato)  {
                   inf=mit;
                   mit=(inf+sup)/2;
                 }
        }
  return 0;
}
```

Igual que en el método secuencial la complejidad del método se va a medir por los casos extremos que puedan presentarse en el proceso de búsqueda.

El caso más favorable se dará cuando el primer elemento central es el buscado, en cuyo caso se hará una sola comparación. El caso más desfavorable se dará cuando el elemento buscado no está en las sublistas, en este caso se harán en forma aproximada $\log_2(n)$ comparaciones, ya que en cada ciclo de comparaciones el número de elementos se reduce a la mitad, factor de 2. Por lo tanto, el número medio de comparaciones que se realizarán con este método es de: **$(1 + \log_2(n)) / 2$**

Si comparamos las fórmulas dadas en ambos métodos (ver apartado 5.1.1. Método secuencial), resulta que para el mismo valor de N el método binario es más eficiente que el método secuencial; además la diferencia es más significativa cuanto más crece N.

6.6 Búsqueda Ternaria

Una búsqueda ternaria es un algoritmo de búsqueda que utiliza una estrategia de "*divide y vencerás*" para aislar un valor particular. Es similar a una búsqueda binaria, pero divide la estructura de datos de búsqueda en tres partes en lugar de dos.

Los algoritmos de divide y vencerás funcionan de forma recursiva. A través de operaciones repetitivas, el algoritmo reduce el campo de búsqueda (es decir, la estructura de datos de búsqueda) para aislar el valor de búsqueda. En una búsqueda ternaria, el algoritmo divide el campo de búsqueda en tercios y aísla el valor mínimo o máximo de dos de esos tercios. Trabajando recursivamente, el algoritmo puede aislar el valor de búsqueda si existe. Por ejemplo, de 30 nodos finales disponibles, una búsqueda ternaria de primer orden reduciría el campo de 30 a 10, y una búsqueda de segundo nivel lo reduciría aún más de 10 a 3 o 4.

En Búsqueda ternaria, dividimos nuestra array en tres partes (*tomando dos a la mitad*) y descartar dos tercios de nuestro espacio de búsqueda en cada iteración. A primera vista, parece que la búsqueda ternaria podría ser más rápida que la búsqueda binaria debido a su complejidad de tiempo en una entrada que contiene n los artículos deben ser $O(\log_3 n)$, que es menor que la complejidad temporal de la búsqueda binaria $O(\log_2 n)$.

En C:

```
#include <stdio.h>

// Algoritmo de búsqueda ternario para devolver la posición // de
objetivo `x` en una array dada `A` de tamaño `n`

int TernarySearch(int arr[], int n, int x)
{
    int low = 0, high = n - 1;

    while (low <= high)
    {
        int left_mid = low + (high - low) / 3;
        int right_mid = high - (high - low) / 3;
```

```
        // int left_mid = (2*low + high)/3;
        // int right_mid = (low + 2*high)/3;

        if (arr[left_mid] == x) {
            return left_mid;
        }
        else if (arr[right_mid] == x) {
            return right_mid;
        }
        else if (arr[left_mid] > x) {
            high = left_mid - 1;
        }
        else if (arr[right_mid] < x) {
            low = right_mid + 1;
        }
        else {
            low = left_mid + 1, high = right_mid - 1;
        }
    }

    return -1;
}

int main(void)
{
    int A[] = { 2, 5, 6, 8, 9, 10 };
    int target = 6;

    int n = sizeof(A) / sizeof(A[0]);
    int index = TernarySearch(A, n, target);

    if (index != -1) {
        printf("Element found at index %d", index);
    }
    else {
        printf("Element not found in the array");
    }

    return 0;
}
```

6.7 Búsqueda Interpolada

La búsqueda por interpolación es un algoritmo similar a búsqueda binaria para buscar un valor objetivo dado en una *array* ordenada. *Es similar a cómo los humanos buscan en una guía telefónica un nombre en particular, el valor objetivo por el cual se ordenan las entradas del libro.*.

Sabemos que la búsqueda binaria siempre elige la mitad del espacio de búsqueda restante, descartando una mitad u otra según el resultado de la comparación entre el valor medio y el valor objetivo. El espacio de búsqueda restante se reduce a la parte anterior o posterior a la posición intermedia.

En comparación, en cada paso de búsqueda, la búsqueda de interpolación calcula en qué espacio de búsqueda restante podría estar presente el objetivo, en función de los valores alto y bajo del espacio de búsqueda y el valor del objetivo. El valor encontrado en esta posición estimada se compara luego con el valor objetivo. Si no es igual, el espacio de búsqueda restante se reduce a la parte anterior o posterior a la posición estimada según la comparación. Este método solo funcionará si los cálculos sobre el tamaño de las diferencias entre los valores objetivo son sensatos.

Las condiciones iníciales son las siguientes:

- El archivo donde buscar ya está en memoria, dispuesto en slots contiguos.
- Los registros del archivo los denominaremos: $R_1, R_2, R_3, ..., R_i, ..., R_n$ donde 1=< i =< n.
- Existen en memoria “n” slots, donde en cada uno de ellos se dispone un registro del archivo donde buscar.
- Todos los registros tienen una clave K_i, donde i es 1 =< i =< n.
- Los registros están **ordenados** respecto de la clave mencionada.
- La búsqueda será hecha en los mismos slots donde están los registros.

La idea de la búsqueda interpolada es arriesgar una proporcionalidad basándose en los valores de las siguientes claves:

Clave más baja = Kl (l de low)
Clave de búsqueda = K
Clave de interpolación = Ki
Clave más alta = Ku (u de up)
donde el índice de la clave de interpolación es :

*i = ((K - Kl) / (Ku - Kl)) * n = factor de interpolación * n*

donde :

Kl = (low)
Ku = (up)
K = clave de búsqueda
n es la cantidad de claves presentes en el archivo o cantidad de registros

Observar que si K = Kl ------> factor de interpolación = (Kl - Kl) / (Ku - Kl) = 0 / (Ku - Kl) = 0; donde para un archivo de 70 registros i = 0 * 70 = 0
Observar que si K = Ku ------> factor de interpolación = (Ku - Kl) / (Ku - Kl) = 1; donde para un archivo de 70 registros i = 1 * 70 = 70

Claves ----> Kl Ku

|<----------->|

Factor ----> 0 1
Registro --> 1 n
Indice ----> 0 n-1

Establecido el índice de la clave de interpolación, queda fijado el registro Ri, de clave Ki, y determinados los subarchivos menor y mayor.

R1 Ri Rn

...----------.....--------.....--------.....
| Kl | | | Ki | | Ku | n slots
...----------.....--------.....--------.....

|<----------->| |<----------->|
subarchivo menor subarchivo mayor

|<-------- Ki - Kl ---------->|

|<---------- Ku - Kl ---------->|

- Si K = Ki, la clave fué hallada.
  Si K < Ki, se repite el procedimiento con el subarchivo menor.
  Si K > Ki, se repite el procedimiento con el subarchivo mayor.

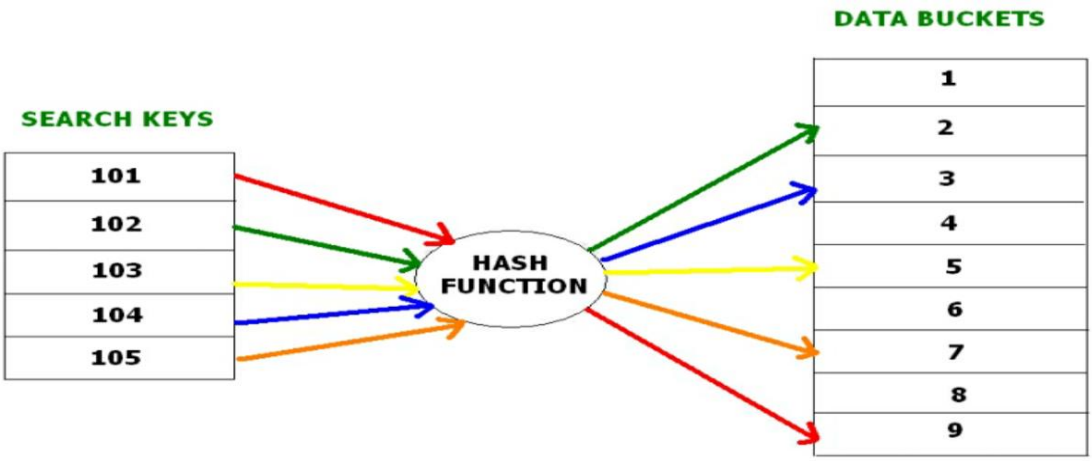
6.8 Búsqueda por transformación de claves (*Hassing*).

Es un método de búsqueda que aumenta la velocidad de búsqueda, pero que no requiere que los elementos estén ordenados.

Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada función *hash*.

La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente. (números demasiado grandes, función es inservible).

La función de *hash* ideal debería ser biyectiva, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que puedes no saber el número de elementos a almacenar.



Por todo lo mencionado, para trabajar con este método de búsqueda debe elegirse previamente:

- Una función hash que se fácil de calcular y que distribuya uniformemente las claves.

    A. Restas sucesivas
    B. Aritmética modular
    C. Mitad del cuadrado
    D. Truncamiento
    E. Plegamiento
    F. Tratamiento de colisiones

- Un método para resolver colisiones. Si estas se presentan se debe contar con algún método que genere posiciones alternativas.

La función de *hash* depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas.

A. Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

```
1998-000        -->        0        =        1998000-1998000
1998-001        -->        1        =        1998001-1998000
1998-002        -->        2        =        1998002-1998000
   ...
1998-399        -->        399      =        1998399-1998000
1999-000        -->        400      =        1999000-1998000+400
   ...
yyyy-nnn --> N = yyyynnn-1998000+(400*(yyyy-1998))
```

B. Aritmética modular

Este método convierte la clave a un entero, se divide por el tamaño del rango del índice y toma el resto como resultado.

La función que se utiliza es el MOD(módulo o resto de la división entera).

$$H(x)= x \text{ MOD } m$$

Donde m es el tamaño del arreglo. La mejor elección de los módulos son los números primos.

Un vector T tiene cien posiciones (0..100). Se tiene que las claves de búsqueda de los elementos de la tabla son enteros positivos. La función de conversión H debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango de (0..100)

$$H(x)= x \text{ MOD } m$$

Si clave=234661234 MOD 101 = 56 234661234 MOD 101 = 56

C. Mitad del cuadrado

Este método consiste en calcular el cuadrado de la clave x.

La función de conversión se define como:

H(x)=c

Donde c se obtiene eliminando dígitos a ambos lados de x2.

Ejemplo:
Una empresa tiene ochenta empleados y cada uno de ellos tiene un número de identificación de cuatro dígitos y el conjunto de direcciones de memoria varía en el rango de 0 a 100. Se pide calcular las direcciones que se obtendrán al aplicar la función de conversión por la mitad del cuadrado de los números empleados:

x –> 4205, 7148, 3350, etc.

x2 –> 17682025 ➔ 82 (dirección de memoria) = H(x)

x2 -> 51093904 => 93 (dirección de memoria) = H(x)

x2 -> 11222500 => 22 (dirección de memoria) = H(x)

D. Truncamiento

Ignora parte de la clave y se utiliza la parte restante directamente como índice (considerando campos no numéricos y sus códigos numéricos.

Ejemplo: Se tiene claves de tipo entero de 8 dígitos y para la tabla de transformación tiene mil posiciones, entonces para la dirección(índice) se considera: el primer, segundo y quinto dígito de derecha forman la función de conversión.

clave:72588495 –> h(clave)=728

E. Plegamiento

Consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión. Por ejemplo, si dividimos los números de 8 cifras en 3, 3 y 2 cifras y se suman, dará otro número de tres cifras (y si no, se toman las tres últimas cifras):

13000000 --> 130 =130+000+00
12345678 --> 657 =123+456+78
71140205 --> 118 --> 1118 =711+402+05
13602499 --> 259 =136+024+99
25000009 --> 259 =250+000+09

F. Tratamiento de colisiones

Cuando el índice correspondiente a un elemento ya está ocupado, se le asigna el primer índice libre a partir de esa posición. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.

También se pueden reservar unos cuantos lugares al final del *array* para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.

Lo más efectivo es crear una lista enlazada (punteros). Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del *array*, ya que se pueden añadir nodos dinámicamente a la lista.

6.9 Búsqueda fuerza bruta.

Búsqueda por fuerza bruta, **búsqueda combinatoria**, **búsqueda exhaustiva** o simplemente **fuerza bruta**, es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de fuerza bruta para encontrar el divisor de un número natural *n* consistiría en enumerar todos los enteros desde 1 hasta *n*, chequeando si cada uno de ellos divide *n* sin generar resto.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema. Por el contrario, la búsqueda por fuerza bruta se usa habitualmente cuando el número de soluciones candidatas no es elevado, o bien cuando éste puede reducirse previamente usando algún otro método heurístico.

Este método sólo se usa cuando el número de posibilidades a evitar es lo suficientemente grande y que contando con el tiempo de ejecución del código necesario para implementarlo, reduzca ampliamente el tiempo necesario para encontrar el resultado esperado.

Características

- Es el algoritmo más simple posible.
- Consiste en probar todas las posibles posiciones del patrón en el texto.
- Requiere espacio constante.
- Realiza siempre saltos de un carácter.
- Compara de izquierda a derecha.
- Realiza la búsqueda del patrón en un tiempo O(mn).
- Realiza 2^n comparaciones previstas de los caracteres del texto.