ENTREGA DEL REPORTE FINAL

Año Sabático: 1/septiembre/2023 al 31/agosto/2024

Docente: Luz Elena Cortez Galván

lcortez@tectijuana.edu.mx

Algoritmos de ordenamiento

Contenido

2

1. Introducción a los Algoritmos Computacionales.

¿Qué es un algoritmo informático?

Un algoritmo informático es un conjunto de instrucciones definidas, ordenadas y acotadas para resolver un problema, realizar un cálculo o desarrollar una tarea. Es decir, un algoritmo es un procedimiento paso a paso para conseguir un fin. A partir de un estado e información iniciales, se siguen una serie de pasos ordenados para llegar a la solución de una situación.

En programación, un algoritmo supone el paso previo a ponerse a escribir el código. Primero debemos encontrar la forma de obtener la solución al problema (definir el algoritmo informático), para luego, a través del código, poder indicarle a la máquina qué acciones queremos que lleve a cabo. De este modo, un programa informático no sería más que un conjunto de algoritmos ordenados y codificados en un lenguaje de programación para poder ser ejecutados en un ordenador.

No obstante, los algoritmos no son algo exclusivo de los ámbitos de las matemáticas, la lógica y la computación. Utilizamos numerosos algoritmos para resolver problemas en nuestra vida cotidiana. Algunos de los ejemplos más habituales son los manuales de instrucciones o las recetas de cocina.

Partes de un algoritmo informático

Las tres partes de un algoritmo son:

I. **Input (entrada).** Información que damos al algoritmo con la que va a trabajar para ofrecer la solución esperada.
II. **Proceso.** Conjunto de pasos para que, a partir de los datos de entrada, llegue a la solución de la situación.
III. **Output (salida).** Resultados, a partir de la transformación de los valores de entrada durante el proceso.

De este modo, un algoritmo informático parte de un estado inicial y de unos valores de entrada, sigue una serie de pasos sucesivos y llega a un estado final en el que ha obtenido una solución.

Características de los algoritmos

Asimismo, los algoritmos presentan una serie de características comunes. Son:

Precisos. Objetivos, sin ambigüedad.

Ordenados. Presentan una secuencia clara y precisa para poder llegar a la solución.

Finitos. Contienen un número determinado de pasos.

Concretos. Ofrecen una solución determinada para la situación o problema planteados.

Definidos. El mismo algoritmo debe dar el mismo resultado al recibir la misma entrada.

Tipos de algoritmos

Existen diversas clasificaciones de algoritmos, en función de diferentes criterios. Según su sistema de signos (cómo describen los pasos a seguir), se distingue entre **algoritmos cuantitativos y cualitativos**, si lo hacen a través de cálculos matemáticos o secuencias lógicas. Asimismo, si requieren o no el empleo de un ordenador para su resolución, se clasifican en **computacionales y no computacionales**.

Pero, si nos fijamos en su función (qué hace) y su estrategia para llegar a la solución (cómo lo hace), encontramos muchos más tipos de algoritmos.

Destacamos los siguientes **cinco tipos de algoritmos informáticos**; y en este libro nos centraremos en los algoritmos de ordenamiento y búsqueda.

Algoritmos de búsqueda.

Los **algoritmos de búsqueda** localizan uno o varios elementos que presenten una serie de propiedades dentro de una estructura de datos.

Existen diversos tipos de búsquedas, entre las que sobresalen:

- **Búsqueda secuencial.** En la que se compara el elemento a localizar con cada elemento del conjunto hasta encontrarlo o hasta que hayamos comparado todos.
- **Búsqueda binaria.** En un conjunto de elementos ordenados, hace una comparación con el elemento ubicado en el medio y, si no son iguales, continúa la búsqueda en la mitad donde puede estar. Y así sucesivamente en intervalos cada vez más pequeños de elementos.

Algoritmos de ordenamiento.

Los **algoritmos de ordenamiento** reorganizan los elementos de un listado según una relación de orden. Las más habituales son el orden numérico y el orden lexicográfico. Un orden eficiente optimiza el uso de algoritmos como los de búsqueda y facilitan la consecución de resultados legibles por personas y no solo máquinas.

Algunos **algoritmos de ordenamiento** son:

- **Ordenamiento de burbuja.** Compara cada elemento de la lista a ordenar con el siguiente e intercambia su posición si no están en el orden adecuado. Se revisa varias veces toda la lista hasta que no se necesiten más intercambios.
- **Ordenamiento por selección.** Vamos colocando el elemento más pequeño disponible en cada una de las posiciones de la lista de forma consecutiva.
- **Ordenamiento rápido.** Elegimos un elemento del conjunto (pivote) y reubicamos el resto a cada uno de sus lados, en función de si son mayores o menores que el elemento que estamos tomando como referencia. Repetimos el procedimiento en cada subconjunto.

Programación dinámica.

La **programación dinámica** es un método de resolución de problemas en el que dividimos un problema complejo en subproblemas y calculamos y almacenamos sus soluciones, para que no haga falta volver a calcularlas más adelante para llegar a la solución del problema. La programación dinámica reduce el tiempo de ejecución de un algoritmo al optimizar la recursión.

Eso sí, para poder aplicarse a un problema, éste debe tener subestructuras óptimas y subproblemas superpuestos. Es decir, que en él se puedan usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto y que el problema se pueda dividir en subproblemas que se reutilizan para ofrecer el resultado global.

Algunos casos en los que se utiliza la **programación dinámica** son:

- **La serie de Fibonacci.** Sucesión de números que comienza con “0” y “1” y, a partir de ellos, cada número es resultado de la suma de los dos que le preceden. La relación de recurrencia la define.
- **Problema de la mochila.**

Algoritmos voraces.

Los **algoritmos voraces** consisten en una estrategia de búsqueda que sigue una heurística en la que se elige la mejor opción en cada paso local con el objetivo de llegar a una solución general óptima. Es decir, en cada paso del proceso escogen el mejor elemento (elemento prometedor) y comprueban que pueda formar parte de una solución global factible. Normalmente se utilizan para resolver problemas de optimización.

Ejemplos de **algoritmos voraces**

En ocasiones, estos algoritmos no encuentran la solución global óptima, ya que al tomar una decisión solo tienen en cuenta la información de las decisiones que han tomado hasta el momento y no las futuras que puede adoptar. Algunos casos en los que los algoritmos voraces alcanzan soluciones óptimas son:

- **Problema de la mochila fraccional (KP).** Disponemos de una colección de objetos (cada uno de ellos con un valor y un peso asociados) y debemos determinar cuáles colocar en la mochila para lograr transportar el valor máximo sin superar el peso que puede soportar.
- **Algoritmo de Dijkstra.** Utilizado para determinar el camino más corto desde un vértice origen hasta los demás vértices de un grafo, que tiene pesos en cada arista.
- **Codificación Huffman.** Método de compresión de datos sin perder información, que analiza la frecuencia de aparición de caracteres de un mensaje y les asigna un código de longitud variable. Cuanto mayor sea la frecuencia le corresponderá un código más corto.

Algoritmos probabilísticos.

Un **algoritmo probabilístico** basa su resultado en una técnica que usa una fuente de aleatoriedad como parte de su lógica. Mediante un muestreo aleatorio de la entrada llega a una solución que puede no ser totalmente óptima, pero que es adecuada para el problema planteado.

Se utiliza en situaciones con limitaciones de tiempo o memoria y cuando se puede aceptar una buena solución de media, ya que a partir de los mismos datos se pueden obtener soluciones diferentes y algunas erróneas. Para que sea más probable ofrecer una solución correcta, se repite el algoritmo varias veces con diferentes submuestras aleatorias y se comparan los resultados.

Existen **dos tipos principales de algoritmos probabilísticos**:

- **Algoritmo de Montecarlo.** Dependiendo de la entrada, hay una pequeña probabilidad de que no acierte o no llegue a una solución. Se puede reducir la probabilidad de error aumentando el tiempo de cálculo.
- **Algoritmo de Las Vegas.** Se ejecuta en un periodo de tiempo concreto. Si encuentra una solución en ese tiempo ésta será correcta, pero es posible que el tiempo se agote y no encuentre ninguna solución.

Conclusión.

Como hemos visto, un algoritmo informático no es más que un conjunto de instrucciones para conseguir un fin.

Los algoritmos están muy presentes en el ámbito de la informática, pero también en nuestra vida cotidiana.

Existen numerosos tipos y ejemplos de algoritmos y, dependiendo de la situación en que nos encontremos, unos u otros nos ayudarán a llegar a la solución que necesitemos.

2 - Algoritmos de Ordenamiento.

Los datos de un vector, una lista o un archivo están ordenados cuando cada elemento ocupa el lugar que le corresponde según su valor, un dato clave o un criterio. Por ejemplo, si cada elemento de un vector diferente del primero, es mayor que los anteriores, se dice que está ordenado ascendentemente, mientras que, si cada elemento diferente del primero, es menor que los anteriores, está ordenado de forma descendente.

En un pequeño conjunto de datos el orden en que estos se presenten puede ser irrelevante, pero cuando la cantidad aumenta el orden toma importancia. ¿Qué tan útil podría ser un diccionario si no estuviera ordenado? ¿O el directorio telefónico? Sara Baase y Allen Van Gelder (Algoritmos computacionales. Introducción al análisis y diseño) mencionan que el ordenamiento de datos fue una de las principales preocupaciones de las ciencias de la computación, prueba de ello es la cantidad de métodos de ordenamiento que se han diseñado. En este libro se explican los algoritmos más utilizados para este propósito.

En computación y matemáticas un **algoritmo de ordenamiento** es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico.

Los métodos de ordenación **se clasifican en dos categorías**:

a) Ordenación interna (de arreglos).

b) Ordenación externa (de archivos).

2.1 – Algoritmos de Ordenamiento Internos.

La ordenación interna o de arreglos, recibe este nombre ya que los elementos o componentes del arreglo a ordenar se encuentran almacenados en la memoria principal de la computadora.

Los **métodos de ordenación interna** a su vez se clasifican en:

- **a)** Métodos directos (n^2).
- **b)** Métodos logarítmicos (n * log n).

Los **métodos directos**, son los más simples y fáciles de entender, tienen la característica de que sus programas son cortos, de sencilla elaboración y comprensión. Son eficientes cuando se trata de una cantidad de datos pequeña, pero son ineficientes cuando el número de elementos a ordenar es grande.

Los métodos directos se subdividen en:

- Ordenación por intercambio. (Burbuja, Burbuja con señal, Shaker sort)
- Ordenación por inserción. (Directa, Binaria)
- Ordenación por selección. (Directa)

Los **métodos logarítmicos**, son más complejos, difíciles de entender, pero requieren de menos comparaciones y movimientos para ordenar sus elementos. Aunque su elaboración y comprensión resulte más sofisticada y abstracta, son eficientes en el ordenamiento de grandes cantidades de datos.

Algunos de los métodos logarítmicos más utilizados son:

- Ordenación por el método ShellSort.
- Ordenación por el método QuickSort.
- Ordenación por el método HeapSort.

Debe mencionarse que una buena medida de eficiencia entre los distintos métodos lo representa el tiempo de ejecución del algoritmo, y éste depende fundamentalmente del número de comparaciones y movimientos que se realicen entre los elementos.

Como **conclusión** puede decirse que cuando N es pequeño deben utilizarse métodos directos, y cuando N es de medio a grande deben emplearse métodos logarítmicos.

2.2 – Algoritmos de Ordenamiento Externos.

La **ordenación externa o de archivos**, recibe este nombre ya que los elementos se encuentran almacenados en un archivo, el cual se almacena en un dispositivo de almacenamiento secundario o externo.

Los algoritmos de ordenación externa son necesarios cuando los datos que se quiere ordenar no caben en la memoria principal (RAM) de la computadora y por tal motivo se encuentran almacenados en un dispositivo secundario externo (el disco duro, cinta, memoria USB, etc.).

La mayoría de estos algoritmos utilizan la técnica de divide y vencerás y la intercalación de archivos, para aplicar el ordenamiento.

Entre los algoritmos de ordenamiento externo más importantes se encuentran:

➔ **Intercalación:**

- ➢ Simple
- ➢ Binaria.
- ➢ Natural.
- ➢ Balanceada.
- ➢ Polifásica.
- ➢ Cascada.

➔ **Merge:**

- ➢ Mezcla Directa.
- ➢ Mezcla Equilibrada.

A continuación se muestran los **algoritmos de ordenamiento** agrupados según su estabilidad, tomando en cuenta la complejidad computacional:

Estables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento de burbuja	Bubble sort	O(n^2)	O(1)	Intercambio
Ordenamiento de burbuja bidireccional	Cocktail sort	O(n^2)	O(1)	Intercambio
Ordenamiento por inserción	Insertion sort	O(n^2)	O(1)	Inserción
Ordenamiento por casilleros	Bucket sort	O(n)	O(n)	No comparativo
Ordenamiento por cuentas	Counting sort	O(n+k)	O(n+k)	No comparativo
Ordenamiento por mezcla	Merge sort	O(n log n)	O(n)	Mezcla
Ordenamiento con árbol binario	Binary tree sort	O(n log n)	O(n)	Inserción
Ordenamiento Radix	Radix sort	O(nk)	O n)	No comparativo
	Gnome sort	O(n^2)	O(1)	

Inestables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento Shell	Shell sort	O($n^{1.25}$)	O(1)	Inserción
	Comb sort	O(n log n)	O(1)	Intercambio
Ordenamiento por selección	Selection sort	O(n^2)	O(1)	Selección
Ordenamiento por montículos	Heap sort	O(n log n)	O(1)	Selección
Ordenamiento rápido	Quick sort	Promedio: O(n log n), peor caso: O(n^2)	O(log n)	Partición
Cuestionables, imprácticos				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
	Bogo sort	O($n \times n!$), peor caso: no termina		

3 – Clasificación de los Algoritmos de Ordenamiento.

3.1 Algoritmos de inserción.

En este tipo de algoritmo los elementos que van a ser ordenados son considerados uno a la vez. Cada elemento es "insertado" en la posición apropiada con respecto al resto de los elementos ya ordenados.

Entre estos algoritmos se encuentran el de Inserción Directa y el ShellSort.

3.1.1 Inserción Directa.

El método de ordenación por **inserción directa** es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que también se conozca con el nombre de método de la baraja.

La idea central de este algoritmo consiste en insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada. Este proceso se repite desde el segundo elemento, hasta el n-ésimo elemento.

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento A[0] se considera ordenado es decir, la lista inicial consta de un elemento.

2. Se inserta A[1] en la posición correcta delante o detrás de A[0], dependiendo de que sea menor o mayor.

3. Por cada bucle o iteración i (desde i = 2 hasta n) se explora la sublista A[i-1] . . A[0] buscando la posición correcta de inserción; a la vez se mueve hacia abajo (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar A[i], para dejar vacía esa posición.

4. Insertar el elemento en la posición correcta.

El método InserciónDirecta tiene dos argumentos, el arreglo A[] que se va a ordenar ascendentemente y N que es el número de elementos que contiene el arreglo.

Algoritmo:

```
InserciónDirecta (A, N)
{El algoritmo ordena los elementos del arreglo utilizando el método de inserción directa, donde A es el nombre del arreglo, y N el número total de elementos que contiene el arreglo}
{I, AUX y K son variables de tipo entero}

1. Repetir con I desde 2 hasta N
        Hacer AUX ← A[I] y K ← I-1
1.1 Repetir mientras (K ≥ 1) y (AUX < A[K])
        Hacer A[K+1] ← A[K] y K ← K-1
1.2 {Fin del ciclo del paso 1.1}
        Hacer A[K+1] ← AUX
2. {Fin del ciclo del paso 1}
```

Ejemplo 3.1.1

Supóngase que se desea ordenar los siguientes números del arreglo A utilizando el método de inserción directa.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

A[2] < A[1]	(67 < 15)	no hay intercambio, el arreglo sigue igual

SEGUNDA PASADA

A[3] < A[2]	(08 < 67)	si hay intercambio
A[2] < A[1]	(08 < 15)	si hay intercambio

A: **08 15 67** 16 44 27 12 35

TERCERA PASADA

A[4] < A[3]	(16 < 67)	si hay intercambio
A[3] < A[2]	(16 < 15)	no hay intercambio

A: **08 15 16 67** 44 27 12 35

Obsérvese que una vez que se determina la posición correcta del elemento se interrumpen las comparaciones. Por ejemplo, en el caso anterior no se realizó la comparación de A[2] < A[1], porque en la pasada anterior ya se habían determinado las posiciones correctas de esos elementos.

CUARTA PASADA

A[5] < A[4]	(44 < 67)	si hay intercambio
A[4] < A[3]	(44 < 16)	no hay intercambio

A: **08 15 16 44 67** 27 12 35

QUINTA PASADA

A[6] < A[5]	(27 < 67)	si hay intercambio
A[5] < A[4]	(27 < 44)	si hay intercambio
A[4] < A[3]	(27 < 16)	no hay intercambio

A: **08 15 16 27 44 67** 12 35

SEXTA PASADA

A[7] < A[6]	(12 < 67)	si hay intercambio
A[6] < A[5]	(12 < 44)	si hay intercambio
A[5] < A[4]	(12 < 27)	si hay intercambio
A[4] < A[3]	(12 < 16)	si hay intercambio
A[3] < A[2]	(12 < 15)	si hay intercambio

A: **08 12 15 16 27 44 67** 35

SEPTIMA PASADA

A[8] < A[7]	(35 < 67)	si hay intercambio
A[7] < A[6]	(35 < 44)	si hay intercambio
A[6] < A[5]	(35 < 27)	no hay intercambio

A: **08 12 15 16 27 35 44 67** **ARREGLO ORDENADO**

3.1.2 ShellSort.

El método de **Shell** es una versión mejorada del método de inserción directa. Recibe ese nombre en honor de su autor Donald L. Shell quien lo propuso en 1959. Este método también se conoce con el nombre de inserción con incrementos decrecientes. Shell propone que las comparaciones entre elementos se efectúen con saltos de mayor tamaño, pero con incrementos decrecientes, así, los elementos quedaran ordenados en el arreglo más rápidamente.

En el algoritmo se desarrollan los siguientes procesos:

1. Este método ordena subarreglos separados del arreglo original; los subarreglos serán separados por H unidades, donde los valores del arreglo H se llamarán incrementos. El requisito para escoger los valores de H es que deben ser solamente números primos, para eficientar el ordenamiento.
2. Después de que los primeros H subarreglos han sido ordenados, se escogerá un nuevo valor de H, y el arreglo será de nuevo partido entre el nuevo conjunto de subarreglos.
3. Cada uno de estos procesos hará intercambios y comparaciones entre los elementos del arreglo; y se repetirá de nuevo el proceso hasta utilizar el valor más pequeño de H en la secuencia, que deberá ser 1. Al principio del ordenamiento se deberán establecer los incrementos.

Algoritmo:

ShellSort (A, N)
{El algoritmo ordena los elementos del arreglo utilizando el método Shell, donde A es el nombre del arreglo, N el número total de elementos que contiene el arreglo A, y V el arreglo de los incrementos decrecientes}
{M, H, J e I son variables de tipo entero, K es del tipo de elementos del arreglo}

1. Inicializar el vector V de incrementos.
2. *Repetir* el paso 3 para M ← número de incrementos hasta 1, disminuyendo en una unidad.
2.1 Hacer H ←V[M]
3. *Repetir* el paso 4 para J ← H+1 hasta N
3.1 Hacer I ← J-H y K ← A[J].
4. *Repetir* mientras (I > 0) y (K ≤ A[I]) efectuar comparaciones e intercambios si es necesario.
4.1 Hacer A[I+H]← A[I] e I ← I-H, si no se cumple
4.2 Hacer A[I+H]← K
5. {Fin del ciclo del paso 2}

Ejemplo 3.1.2

Supóngase que se desea ordenar los siguientes números del arreglo A utilizando el método de Shell.

A: 15 67 08 16 44 27 12 35

V: N/2=8/2=4 entonces serán los números primos entre 1 y 4.

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

K=A[4] ≤ A[1]	(16 ≤ 15)	no hay intercambio
K=A[5] ≤ A[2]	(44 ≤ 67)	si hay intercambio
K=A[6] ≤ A[3]	(27 ≤ 08)	no hay intercambio
K=A[7] ≤ A[4]	(12 ≤ 16)	si hay intercambio
K=A[7] ≤ A[1]	(12 ≤ 15)	si hay intercambio
K=A[8] ≤ A[5]	(35 ≤ 67)	si hay intercambio
K=A[8] ≤ A[2]	(35 ≤ 44)	si hay intercambio

A: **12 35** 08 **15** 44 27 **16 67**

SEGUNDA PASADA

K=A[3] ≤ A[1]	(08 ≤ 12)	si hay intercambio
K=A[4] ≤ A[2]	(15 ≤ 35)	si hay intercambio
K=A[5] ≤ A[3]	(44 ≤ 12)	no hay intercambio
K=A[6] ≤ A[4]	(27 ≤ 35)	si hay intercambio
K=A[7] ≤ A[5]	(16 ≤ 44)	si hay intercambio
K=A[8] ≤ A[6]	(67 ≤ 35)	no hay intercambio

A: **08 15 12 27 16 35 44 67**

TERCERA PASADA

K=A[2] ≤ A[1]	(15 ≤ 08)	no hay intercambio
K=A[3] ≤ A[2]	(12 ≤ 15)	si hay intercambio
K=A[4] ≤ A[3]	(27 ≤ 15)	no hay intercambio
K=A[5] ≤ A[4]	(16 ≤ 27)	si hay intercambio
K=A[6] ≤ A[5]	(35 ≤ 27)	no hay intercambio

K=A[7] ≤ A[6]	(44 ≤ 35)	no hay intercambio
K=A[8] ≤ A[7]	(67 ≤ 44)	no hay intercambio

A: 08 12 15 16 27 35 44 67 **ARREGLO ORDENADO**

3.2 Algoritmos de intercambio.

En este tipo de algoritmos se toman los elementos de dos en dos, se comparan y se “intercambian” si no están en el orden adecuado. Este proceso se repite hasta que se ha analizado todo el conjunto de elementos y ya no hay intercambios.

Entre estos algoritmos se encuentran el de la Burbuja (BubbleSort) y el QuickSort.

3.2.1 Burbuja.

El ordenamiento de la **burbuja** (BubbleSort en inglés) también conocido como método de intercambio directo, es un sencillo algoritmo de ordenamiento por su fácil comprensión y programación. Pero es preciso señalar que es probablemente el método más ineficiente.

Este método puede trabajar de dos maneras diferentes, ordenando los elementos ascendente o descendentemente según como se desee.

La idea básica de este algoritmo consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados. Se realizan (N-1) pasadas, transportando en cada una de las mismas el menor o mayor elemento (según sea el caso) a su posición ideal. Al final de las (N-1) pasadas los elementos del arreglo estarán ordenados.

Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas “burbujas".

Las etapas del algoritmo son:

1. En la pasada 1 se comparan elementos adyacentes

(A[1], A[2]), (A[2], A[3]), (A[3], A[4]),...(A[N-1], A[N])

Se realizan n-1 comparaciones, por cada pareja (A[i], A[i+1]) se intercambian los valores si A[i+1] < A[i].

Al final de la pasada, el elemento mayor de la lista está situado en A[N].

2. En la pasada 2 se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en A[N-1].

3. El proceso termina con la pasada N-1, en la que el elemento más pequeño se almacena en A[1] .

El método Burbuja () implementa el algoritmo de ordenación de la burbuja. Tiene dos argumentos, el arreglo A[] que se va a ordenar ascendentemente, y el número de elementos N del arreglo.

Algoritmo:

Burbuja (A, N)
{El algoritmo ordena ascendentemente los elementos del arreglo utilizando el método de la burbuja. Transporta en cada pasada el elemento más grande hacia la parte derecha del arreglo. A es el nombre del arreglo, y N el número total de elementos que contiene el arreglo}
{K, L y TEMP son variables de tipo entero}

1. *Repetir* con K desde 1 hasta N-1
1.1 *Repetir* con L desde 1 hasta N-K
1.1.1 Verificar si es necesario realizar un intercambio
Si* A[L] > A[L+1] *entonces
Hacer TEMP◄— A[L], A[L]◄— A[L+1] y A[L+1]◄— TEMP
1.1.2 {Fin del condicional del paso 1.1.1}
1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Ejemplo 3.2.1

Supóngase que se desea ordenar los siguientes números del arreglo A transportando en cada pasada el mayor elemento hacia la parte derecha del arreglo.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

A[1] > A[2]	(15 > 67)	no hay intercambio
A[2] > A[3]	(67 > 08)	si hay intercambio
A[3] > A[4]	(67 > 16)	si hay intercambio
A[4] > A[5]	(67 > 44)	si hay intercambio
A[5] > A[6]	(67 > 27)	si hay intercambio
A[6] > A[7]	(67 > 12)	si hay intercambio
A[7] > A[8]	(67 > 35)	si hay intercambio

A: 15 08 16 44 27 12 35 **67**

Obsérvese que el elemento más grande, en este caso el 67, fue situado en la última posición del arreglo.

SEGUNDA PASADA

A[1] > A[2]	(15 > 08)	si hay intercambio
A[2] > A[3]	(15 > 16)	no hay intercambio
A[3] > A[4]	(16 > 44)	no hay intercambio
A[4] > A[5]	(44 > 27)	si hay intercambio
A[5] > A[6]	(44 > 27)	si hay intercambio
A[6] > A[7]	(44 > 35)	si hay intercambio

A: 08 15 16 27 12 35 **44 67**

El segundo elemento más grande del arreglo, en este caso 44, fue situado en la penúltima posición del arreglo. Mismo proceso que se repetirá en las siguientes pasadas.

TERCERA PASADA

A[1] > A[2]	(08 > 15)	no hay intercambio
A[2] > A[3]	(15 > 16)	no hay intercambio
A[3] > A[4]	(16 > 27)	no hay intercambio
A[4] > A[5]	(27 > 12)	si hay intercambio
A[5] > A[6]	(27 > 35)	no hay intercambio

A: 08 15 16 12 27 **35 44 67**

CUARTA PASADA

A[1] > A[2]	(08 > 15)	no hay intercambio
A[2] > A[3]	(15 > 16)	no hay intercambio
A[3] > A[4]	(16 > 12)	si hay intercambio
A[4] > A[5]	(16 > 27)	no hay intercambio

A: 08 15 12 16 **27 35 44 67**

QUINTA PASADA

A[1] > A[2]	(08 > 15)	no hay intercambio
A[2] > A[3]	(15 > 12)	si hay intercambio

A[3] > A[4] (15 > 16) no hay intercambio

A: 08 12 15 **16 27 35 44 67**

SEXTA PASADA

A[1] > A[2] (08 > 12) no hay intercambio

A[2] > A[3] (12 > 15) no hay intercambio

A: 08 12 **15 16 27 35 44 67**

SEPTIMA PASADA

A[1] > A[2] (08 > 12) no hay intercambio

A: 08 12 15 16 27 35 44 67 **ARREGLO ORDENADO**

3.2.2 QuickSort.

El método de ordenación **quicksort** es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es también conocido con el nombre de método rápido y de ordenación por partición. Este método es una mejora sustancial del método de intercambio directo y recibe su nombre quicksort (rápido) por la velocidad con que ordena los elementos del arreglo. Su autor C. A. Hoare lo bautizo así.

La idea central del algoritmo consiste en lo siguiente:

1. Elegir un elemento X de la lista de elementos a ordenar, al que llamaremos **pivote**.
2. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupará exactamente el lugar que le corresponderá en la lista ordenada.
3. La lista original queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento.
5. El proceso termina cuando todos los elementos se encuentran en su posición correcta en el arreglo.

El quicksort es considerado como el mejor algoritmo de ordenamiento debido a su importante ventaja en términos de eficiencia, es por ello que produce el método más efectivo y mayormente usado de ordenamiento para listas de cualquier tamaño.

Ejemplo 3.2.2

Supóngase que se desea ordenar los elementos que se encuentran en el arreglo A utilizando el método quicksort.

A: 15 67 08 16 44 27 12 35

Se seleccionara A[1] como el **pivote**, por lo tanto **X = 15**.

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

Se empieza a recorrer el arreglo de **derecha a izquierda** comparando si los elementos son mayores o iguales a X. Si un elemento no cumple con esta condición, se intercambian los mismos y almacenamos en una variable auxiliar el elemento intercambiado (acotando el arreglo por la derecha).

A[8] ≥ X (35 ≥ 15) no hay intercambio

A[7] ≥ X (12 ≥ 15) si hay intercambio

A: **12** 67 08 16 44 27 **15** 35

Se inicia nuevamente el recorrido, pero ahora de **izquierda a derecha**, comparando si los elementos son menores o iguales a X. Si un elemento no cumple con esta condición, entonces se intercambian los mismos y se almacena en otra variable auxiliar el elemento intercambiado (acotando el arreglo por la izquierda).

A[2] ≤ X (67 ≤ 15) si hay intercambio

A: **12** **15** 08 16 44 27 **67** 35

Se repiten los pasos anteriores hasta que el elemento X encuentra su posición correcta en el arreglo.

SEGUNDA PASADA

Recorrido de derecha a izquierda.

A[6] ≥ X (27 ≥ 15) no hay intercambio

A[5] ≥ X (44 ≥ 15) no hay intercambio

A[4] ≥ X (16 ≥ 15) no hay intercambio

A[3] ≥ X (08 ≥ 15) si hay intercambio

A: **12** **08** **15** 16 44 27 **67** 35

Como el recorrido de izquierda a derecha debería iniciarse en la misma posición donde se encuentra el elemento X, el proceso se termina ya que el elemento X se encuentra en su posición correcta. Ahora se procederá a seleccionar un nuevo valor para X, y se repetirá el mismo proceso hasta que todos los elementos del arreglo se encuentren en la posición correcta. Esto se puede realizar de manera **recursiva** o **iterativa**.

El algoritmo de ordenación por el método quicksort en su forma recursiva, para realizar más rápido el ordenamiento, es el siguiente:

Algoritmo:

QuicksortRecursivo (A, N)
{El algoritmo ordena los elementos del arreglo utilizando el método quicksort, de manera recursiva. A es el nombre del arreglo, y N el número total de elementos que contiene el arreglo}

1. Llamar al algoritmo ReduceQRecursivo con 1 y N.

Obsérvese que el algoritmo **quicksortrecursivo** requiere para su funcionamiento de otro algoritmo que se presenta a continuación, a este método se la llama pasando como argumentos los índices que la delimitan 0 y n (índice inferior y superior):

```
ReduceQRecursivo (INI, FIN)
{INI y FIN representan las posiciones del extremo izquierdo y derecho respectivamente, del conjunto de elementos a ordenar}
{IZQ, DER, POS y AUX son variables de tipo entero, BAND es una variable de tipo booleano}

1. Hacer IZQ ← INI, DER ← FIN, POS ← INI y BAND ← VERDADERO.
2. Repetir mientras (BAND = VERDADERO)
        Hacer BAND ← FALSO
     2.1 Repetir mientras(A[POS] ≤ A[DER]) y (POS ≠ DER)
       Hacer DER ← DER – 1
     2.2 {Fin del ciclo del paso 2.1}
     2.3 Si POS ≠ DER entonces
       Hacer AUX ← A[POS], A[POS] ← A[DER],
       A[DER] ← AUX y POS ← DER
     2.3.1 Repetir mientras (A[POS] ≥ A[IZQ]) y (POS ≠ IZQ)
       Hacer IZQ ← IZQ + 1
     2.3.2 {Fin del ciclo del paso 2.3.1}
     2.3.3 Si POS ≠ IZQ entonces
       Hacer BAND ← VERDADERO], AUX ← A[POS],
       A[POS] ← A[IZQ], A[IZQ] ← AUX y POS ← IZQ
     2.3.4 {Fin de la condicional del paso 2.3.3}
     2.4 {Fin de la condicional del paso 2.3}
3. {Fin del ciclo del paso 2}
4. Si (POS – 1) > INI entonces
Regresar a ReduceQRecursivo con INI y (POS – 1) {Llamada Recursiva}
5. {Fin de la condicional del paso 4}
6. Si FIN > (POS + 1) entonces
Regresar a ReduceQRecursivo con (POS + 1) y FIN {Llamada Recursiva}
7. {Fin de la condicional del paso 6}
```

Una gran mejora en el funcionamiento del método del quicksort puede producirse si el primer elemento del arreglo seleccionado como pivote, se encuentra en la mitad o muy próximo a la mitad del mismo. Es decir, se permutan los elementos del arreglo de tal forma que para el elemento X, todos los elementos que se encuentren a su izquierda desde A[1] hasta A[i], donde i es igual a ((N/2) – 1), sean menores o iguales a él y todos los elementos que se encuentren a su derecha desde A[i + 1] hasta A[N] sean mayores o iguales a él.

3.3 Algoritmos de selección.

En este tipo de algoritmos se "selecciona" o se busca el elemento más pequeño (o más grande) de todo el conjunto de elementos y se coloca en su posición adecuada. Este proceso se repite para el resto de los elementos hasta que todos son analizados.

Entre estos algoritmos se encuentra el de Selección Directa y el HeapSort.

3.3.1 Selección Directa.

El método de ordenación por **selección directa** es sencillo, comprensible y de fácil programación, pero no es recomendable utilizarlo cuando el número de elementos del arreglo es medio o grande. La idea básica de este algoritmo consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se coloca en la segunda posición. El proceso se repite de una forma similar hasta que todos los elementos del arreglo hayan sido ordenados.

El método se basa en los siguientes principios:

1. Seleccionar el menor elemento del arreglo.
2. Intercambiar dicho elemento con el primero.
3. Repetir los pasos anteriores con los (N-1), (N-2) elementos, y así sucesivamente hasta que solo quede el elemento mayor.

Algoritmo:

```
SeleccionDirecta (A, N)
{El algoritmo ordena los elementos del arreglo utilizando el método de selección directa. A es el nombre del arreglo, y N el número total de elementos que contiene el arreglo}
{I, MENOR, J y K son variables de tipo entero}

1. Repetir con I desde 1 hasta N-1
        Hacer MENOR ← A[I] y K ← I
        1.1 Repetir con J desde I + 1 hasta N
                1.1.1 Si A[J] < MENOR entonces
                        Hacer MENOR ← A[J] y K ← J
                1.1.2 {Fin del condicional del paso 1.1.1}
        1.2 {Fin del ciclo del paso 1.1}
        Hacer A[K] ← A[I] y A[I] ← MENOR
2. {Fin del ciclo del paso 1}
```

Ejemplo 3.3.1

Supóngase que se desea ordenar los siguientes números del arreglo A utilizando el método de selección directa.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

Se realiza la siguiente asignación: MENOR ← A[1] (**15**)

A[2] < MENOR	(67 < 15)	no se cumple la condición
A[3] < MENOR	(08 < 15)	si se cumple la condición, hay intercambio
		MENOR ← A[3] (**08**)
A[4] < MENOR	(16 < 08)	no se cumple la condición
A[5] < MENOR	(44 < 08)	no se cumple la condición
A[6] < MENOR	(27 < 08)	no se cumple la condición
A[7] < MENOR	(12 < 08)	no se cumple la condición
A[8] < MENOR	(35 < 08)	no se cumple la condición

A: **08** 67 15 16 44 27 12 35

Obsérvese que el menor elemento del arreglo A[3](08), se intercambió con el elemento de la primer posición A[1](15).

SEGUNDA PASADA

Se realiza la siguiente asignación: MENOR ← A[2] (**67**)

A[3] < MENOR	(15 < 67)	si se cumple la condición, hay intercambio
		MENOR ← A[3] (**15**)
A[4] < MENOR	(16 < 15)	no se cumple la condición
A[5] < MENOR	(44 < 15)	no se cumple la condición
A[6] < MENOR	(27 < 15)	no se cumple la condición
A[7] < MENOR	(12 < 15)	si se cumple la condición, hay intercambio
		MENOR ← A[7] (**12**)
A[8] < MENOR	(35 < 12)	no se cumple la condición

A: **08** **12** 15 16 44 27 67 35

Obsérvese que el segundo elemento más pequeño del arreglo A[7](12), se intercambió con el elemento de la segunda posición A[2](67).

TERCERA PASADA

Se realiza la siguiente asignación: MENOR◄— A[3] (**15**)

A[4] < MENOR (16 < 15) no se cumple la condición

A[5] < MENOR (44 < 15) no se cumple la condición

A[6] < MENOR (27 < 15) no se cumple la condición

A[7] < MENOR (67 < 15) no se cumple la condición

A[8] < MENOR (35 < 15) no se cumple la condición

A: 08 12 15 16 44 27 67 35

En esta pasada no hubo intercambios, el tercer elemento más pequeño del arreglo A[3](15) ya se encontraba en su posición correcta.

CUARTA PASADA

Se realiza la siguiente asignación: MENOR◄— A[4] (**16**)

A[5] < MENOR (44 < 16) no se cumple la condición

A[6] < MENOR (27 < 16) no se cumple la condición

A[7] < MENOR (67 < 16) no se cumple la condición

A[8] < MENOR (35 < 16) no se cumple la condición

A: 08 12 15 16 44 27 67 35

En esta pasada tampoco hubo intercambios, el cuarto elemento más pequeño del arreglo A[4](16) ya se encontraba en su posición correcta.

QUINTA PASADA

Se realiza la siguiente asignación: MENOR◄— A[5] (**44**)

A[6] < MENOR (27 < 44) si se cumple la condición, hay intercambio

MENOR ◄— A[6] (**27**)

A[7] < MENOR (67 < 27) no se cumple la condición

A[8] < MENOR (35 < 27) no se cumple la condición

A: 08 12 15 16 27 44 67 35

Obsérvese que el quinto elemento más pequeño del arreglo A[6](27), se intercambió con el elemento de la quinta posición A[5](44).

SEXTA PASADA

Se realiza la siguiente asignación: MENOR◄— A[6] (**44**)

A[7] < MENOR (67 < 44) no se cumple la condición

A[8] < MENOR (35 < 44) si se cumple la condición, hay intercambio

MENOR ◄— A[8] (**35**)

A: 08 12 15 16 27 35 67 44

Obsérvese que el sexto elemento más pequeño del arreglo A[8](35), se intercambió con el elemento de la sexta posición A[6](44).

SEPTIMA PASADA

Se realiza la siguiente asignación: MENOR◄— A[7] (**67**)

A[8] < MENOR (44 < 67) si se cumple la condición, hay intercambio

MENOR ◄— A[8] (**44**)

A: 08 12 15 16 27 35 44 67 ARREGLO ORDENADO

Obsérvese que el séptimo elemento más pequeño del arreglo A[8](44), se intercambió con el elemento de la séptima posición A[7](67).

3.3.2 HeapSort.

El método de ordenación **heapsort** es también conocido con el nombre de montículo. Su nombre se debe a su autor J. W. Williams, quien lo bautizo así. Es el más eficiente de los métodos de ordenación que trabajan con árboles.

Este ordenamiento utiliza un **árbol binario de N nodos**, tal que el contenido de cada nodo es menor o igual al contenido de su nodo padre. La raíz del árbol o primer elemento del resultante, debe ser el elemento más grande del grupo original. En el ordenamiento solo se reserva un nodo para cada uno de los elementos del arreglo original, por lo que el arreglo se utiliza como un espacio de trabajo para el ordenamiento, de tal manera que el espacio adicional requerido es únicamente para las variables auxiliares.

La idea central del algoritmo consiste en lo siguiente:

1. Construir un montículo a partir del arreglo original.
2. Eliminar la raíz del montículo, en forma repetida, al momento de ordenar los elementos del árbol.

INSERCION DE UN ELEMENTO EN UN MONTICULO

La inserción de un elemento en el montículo se lleva a cabo de la siguiente manera:

1. Se inserta el elemento en la primera posición disponible.
2. Se verifica si su valor es mayor que el de su padre. Si se cumple esta condición, entonces se efectúa un intercambio. Si no se cumple la condición, entonces el algoritmo se detiene y el elemento queda ubicado en su posición correcta en ese momento en el montículo.

Ejemplo 3.3.2 a)

Supóngase que se desean insertar los siguientes números de un arreglo, en un montículo que se encuentra vacío.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan serán las siguientes:

PRIMERA PASADA

A[2] > A[1] (67 > 15) si hay intercambio

A: **67** **15** 08 16 44 27 12 35

SEGUNDA PASADA

A[3] > A[1] (08 > 67) no hay intercambio

A: 67 15 **08** 16 44 27 12 35

TERCERA PASADA

A[4] > A[2] (16 > 15) si hay intercambio

A: 67 16 08 **15** 44 27 12 35

A[4] > A[1] (16 > 67) no hay intercambio

CUARTA PASADA

A[5] > A[2] (44 > 16) si hay intercambio

A: 67 44 08 15 **16** 27 12 35

A[5] > A[1] (44 > 67) no hay intercambio

QUINTA PASADA

A[6] > A[3] (27 > 08) si hay intercambio

A: 67 44 27 15 16 **08** 12 35

A[6] > A[1] (27 > 67) no hay intercambio

SEXTA PASADA

A[7] > A[3] (12 > 27) no hay intercambio

A: 67 44 27 15 16 08 **12** 35

SEPTIMA PASADA

A[8] > A[4] (35 > 15) si hay intercambio

A: 67 44 27 35 16 08 12 **15**

A[8] > A[2] (35 > 44) no hay intercambio

MONTICULO RESULTANTE

A: 67 44 27 35 16 08 12 15

El algoritmo para insertar elementos en un montículo es el siguiente:

Algoritmo:

InsertaMontículo (A, N)
{El algoritmo inserta los elementos en un montículo almacenados en un arreglo. A es el nombre del arreglo, y N el número total de elementos que contiene el arreglo}
{I, K y AUX son variables de tipo entero, BAND es una variable de tipo booleano}

1. *Repetir* con I desde 2 hasta N
Hacer K ← I y BAND ← VERDADERO
1.1 *Repetir* mientras (K > 1) y (BAND = VERDADERO)
Hacer BAND ← FALSO
1.1.1 *Si* A[K] > A[parte entera(K entre 2)] *entonces*
Hacer AUX← A[parte entera(K entre 2)],
A[parte entera(K entre 2)] ← A[K],
A[K] ← AUX,
K ← parte entera(K entre 2) y
BAND ← VERDADERO
1.1.2 {Fin del condicional del paso 1.1.1}
1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

ELIMINACION DE UN MONTICULO

El proceso para obtener los elementos ordenados se efectúa eliminando la raíz del montículo en forma repetida. Ahora bien, los pasos necesarios para lograr la eliminación de la raíz del montículo son los siguientes:

1. Se remplaza la raíz con el elemento que ocupa la última posición del montículo.
2. Se verifica si el valor de la raíz es menor que el valor más grande de sus hijos. Si se cumple la condición, entonces se efectúa el intercambio. Si no se cumple la condición, entonces el algoritmo se detiene y el elemento queda ubicado en su posición correcta en el montículo en ese momento.
   Este paso se aplica de manera recursiva, de arriba hacia abajo.

Ejemplo 3.3.2 b)

Supóngase que se desea eliminar la raíz del montículo creado anteriormente.

PRIMERA ELIMINACION DE LA RAIZ

Se intercambia la raíz (67), con el elemento que ocupa la última posición del montículo A[8] (15).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (44 < 27) no se cumple la condición

Si(AUX < MAYOR) (15 < 44) si se cumple la condición, hay intercambios

Si (MAYOR < A[5]) (35 < 16) no se cumple la condición

Si(AUX < MAYOR) (15 < 35) si se cumple la condición, hay intercambios

A[1] = 44, A[2] = 35, A[4] = 15

A: 44 35 27 **15** 16 08 12 **67**

SEGUNDA ELIMINACION DE LA RAIZ

Se intercambia la raíz (44), con el elemento que ocupa la posición A[7] del montículo (12).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (35 < 27) no se cumple la condición

Si(AUX < MAYOR) (12 < 35) si se cumple la condición, hay intercambio

Si (MAYOR < A[5]) (15 < 16) si se cumple la condición, hay intercambio

Si(AUX < MAYOR) (12 < 16) si se cumple la condición, hay intercambio

A[1] = 35, A[2] = 16, A[5] = 12

A: 35 16 27 15 **12** 08 **44 67**

TERCERA ELIMINACION DE LA RAIZ

Se intercambia la raíz (35), con el elemento que ocupa la posición A[6] del montículo (08).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (16 < 27) si se cumple la condición, hay intercambio

Si(AUX < MAYOR) (08 < 27) si se cumple la condición, hay intercambio

A[1] = 27, A[3] = 08

A: 27 16 **08** 15 12 **35 44 67**

CUARTA ELIMINACION DE LA RAIZ

Se intercambia la raíz (27), con el elemento que ocupa la posición A[5] del montículo (12).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (16 < 08) no se cumple la condición

Si(AUX < MAYOR) (12 < 16) si se cumple la condición, hay intercambio

Si (MAYOR < A[3]) (15 < 08) no se cumple la condición

Si(AUX < MAYOR) (12 < 15) si se cumple la condición, hay intercambio

A[1] = 16, A[2] = 15, A[4] = 12

A: 16 15 08 **12 27 35 44 67**

QUINTA ELIMINACION DE LA RAIZ

Se intercambia la raíz (16), con el elemento que ocupa la posición A[4] del montículo (12).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (15 < 08) no se cumple la condición

Si(AUX < MAYOR) (12 < 15) si se cumple la condición, hay intercambio

A[1] = 15, A[2] = 12

A: 15 12 08 **16 27 35 44 67**

SEXTA ELIMINACION DE LA RAIZ

Se intercambia la raíz (15), con el elemento que ocupa la posición A[3] del montículo (08).

Las comparaciones que se realizan son las siguientes:

Si (MAYOR < A[3]) (12 < 08) no se cumple la condición

Si(AUX < MAYOR) (08 < 12) si se cumple la condición, hay intercambio

A[1] = 12, A[2] = 08

A: 12 08 15 16 27 35 44 67

SEPTIMA ELIMINACION DE LA RAIZ

Se intercambia la raíz (12), con el elemento que ocupa la posición A[2] del montículo (08).

Las asignaciones que se realizan son las siguientes:

A[2] = 12, A[1] = 08

A: 08 12 15 16 27 35 44 67 ARREGLO ORDENADO

Es de observar que, al eliminar repetidamente la raíz del montículo, el arreglo quedo ordenado. Aquí se introduce la descripción formal del algoritmo de eliminar sucesivamente la raíz del montículo.

Algoritmo:

```
EliminaMontículo (A, N)
{El algoritmo elimina la raíz del montículo en forma repetida. A es el nombre
del arreglo, y N el número total de elementos que contiene el arreglo}
{I, AUX, IZQ, DER, K y AP son variables de tipo entero}

1. Repetir con I desde N hasta 2
        Hacer AUX ← A[I], A[I] ← A[1], IZQ ← 2, DER ← 3 y K ← 1
        1.1 Repetir mientras (IZQ < I)
                Hacer MAYOR ← A[IZQ] y AP ← IZQ
        1.1.1 Si (MAYOR < A[DER] y (DER ≠ I) entonces
                Hacer MAYOR ← A[DER] y AP ← DER
        1.1.2 {Fin del condicional del paso 1.1.1}
        1.1.3 Si AUX < MAYOR entonces
                Hacer A[K] ← A[AP]
        1.1.4 {Fin del condicional del paso 1.1.3}
                Hacer K ← AP, IZQ ← K*2 y DER ← IZQ+1
        1.2 {Fin del ciclo del paso 1.1}
        Hacer A[K] ← AUX
2. {Fin del ciclo del paso 1}
```

Luego de haber presentado los algoritmos para insertar y eliminar elementos de un montículo, es posible analizar el proceso de ordenación de arreglos por medio de este método de la siguiente manera, la cual consta de dos partes:

1. Construir el montículo a partir del arreglo original.
2. Eliminar repetidamente la raíz del montículo creado.

El algoritmo general sería el siguiente:

Algoritmo:

```
MONTICULO (A, N)
{El algoritmo ordena los elementos del arreglo utilizando el método del
montículo. A es el nombre del arreglo, y N el número total de elementos que
contiene el arreglo}

1. Llamar al algoritmo InsertaMontículo con A y N.
2. Llamar al algoritmo EliminaMontículo con A y N.
```

3.4 Algoritmos de intercalación.

El proceso de ordenar los datos almacenados en varios archivos se conoce con el nombre de **fusión** o **mezcla**, entendiendo por este concepto la intercalación o combinación de dos o más secuencias en una única secuencia ordenada.

Por **intercalación** de archivos se entiende la unión o fusión de dos o más archivos ordenados, de acuerdo con un determinado campo clave, en un solo archivo.

3.4.1 Intercalación Simple.

Este procedimiento supone la existencia de dos archivos F1 y F2 que contienen N y M elementos respectivamente, los intercala y produce un tercer archivo llamado F3, que contendrá N + M elementos. Para que funcione este algoritmo los archivos F1 y F2 deben estar previamente ordenados utilizando algún método de ordenamiento.

Ejemplo 3.4.1

Supóngase que se tienen dos archivos, F1 y F2, previamente ordenados de acuerdo a su campo clave, en orden ascendente:

F1: 06 09 18 20 35

F2: 10 16 25 28 66 82 87

Debe producirse entonces el archivo F3 ordenado, como resultado de la unión de F1 y F2. En cada proceso, solo pueden ser accesadas directamente dos claves; una del archivo F1 y otra del archivo F2. Las comparaciones que se realizan para producir el archivo F3 son las siguientes:

(06 < 10) si se cumple la condición.

Entonces se escribe 06 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F1(09).

F3: 06

(09 < 10) si se cumple la condición.

Entonces se escribe 09 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F1(18).

F3: 06 09

(18 < 10) no se cumple la condición.

Entonces se escribe 10 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F2(16).

F3: 06 09 10

(18 < 16) no se cumple la condición.

Entonces se escribe 16 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F2(25).

F3: 06 09 10 16

(18 < 25) si se cumple la condición.

Entonces se escribe 18 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F1(20).

F3: 06 09 10 16 18

(20 < 25) si se cumple la condición.

Entonces se escribe 20 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F1(35).

F3: 06 09 10 16 18 20

(35 < 25) no se cumple la condición.

Entonces se escribe 25 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F2(28).

F3: 06 09 10 16 18 20 25

(35 < 28) no se cumple la condición.

Entonces se escribe 28 en el archivo de salida **F3** y se vuelve a leer la siguiente clave de F2(66).

F3: 06 09 10 16 18 20 25 28

(35 < 66) si se cumple la condición.

Entonces se escribe 35 en el archivo de salida.

F3: 06 09 10 16 18 20 25 28 35

El proceso se detiene cuando en uno u otro archivo se detecta el fin de archivo, en tal caso solo se tendrán que transcribir las claves del archivo no vacío al archivo de salida F3.

El resultado final de la intercalación entre los archivos F1 y F2 es la siguiente:

F3: 06 09 10 16 18 20 25 28 35 66 82 87

Analicemos ahora el algoritmo de intercalación de archivos

Algoritmo:

```
INTERCALACION (F1, F2, F3)
{El algoritmo intercala los elementos de dos archivos F1 y F2 previamente ordenados, y almacena en el archivo F3}
{R1 y R2 son variables de tipo entero}

1. Abrir los archivos F1 y F2 para lectura.
2. Abrir el archivo F3 para escritura.
3. Leer R1 de F1 y R2 de F2
        {R1 y R2 son las primeras claves de F1 y F2 respectivamente}
4. Repetir mientras (no sea el fin de archivo de F1) y (no sea el fin de archivo de F2)
        4.1 Si R1 < R2 entonces
                Escribir R1 en F3
                Leer R1 de F1
        si no
                Escribir R2 en F3
                Leer R2 de F2
        4.2 {Fin del condicional del paso 4.1}
5. {Fin del ciclo del paso 4}
6. Repetir mientras (no sea el fin de archivo de F1) o (no sea el fin de archivo de F2)
        6.1 Si (fin de archivo de F1) entonces
                6.1.1 Repetir mientras (no sea el fin de archivo de F2)
                        Leer R2 de F2
                        Escribir R2 en F3
                6.1.2 {Fin del ciclo del paso 6.1.1}
        6.2 si no
                6.2.1 Repetir mientras (no sea el fin de archivo de F1)
                        Leer R1 de F1
                        Escribir R1 en F3
                6.2.2 {Fin del ciclo del paso 6.2.1}
7. {Fin del ciclo del paso 6}
8. Cerrar los archivos F1, F2 y F3.
```

Pre condiciones de los archivos de entrada:

- ✓ Los archivos de entrada contienen registros homogéneos
- ✓ Los archivos de entrada deben estar ordenados en base a la misma llave

Post condiciones del archivo de salida:

- ➢ El archivo de salida quedará ordenado por la misma llave que los archivos de entrada
- ➢ El archivo de salida contendrá todos los registros de los archivos de entrada

Existen varias formas de manejar la fase de intercalación de un archivo. Entre las que encontramos los siguientes métodos:

•Intercalación binaria

El algoritmo de intercalación simple requiere una exploración o búsqueda secuencial para localizar la posición de un elemento en la sublista ordenada, si en lugar de considerar una búsqueda secuencial se realizara una búsqueda binaria se mejoraría considerablemente el algoritmo y se aumentaría la velocidad de ejecución. Este algoritmo utiliza la técnica de dividir y conquistar, por lo tanto, divide al vector y toma un elemento pivote y compara contra él los elementos del vector dividido.

•Intercalación natural

Una intercalación que manipula dos archivos de entrada a la vez se llama intercalación de noble vía; una intercalación que manipula M archivos de entrada a la vez se llama intercalación de M-vías. M es referido como el grado de la intercalación. Una intercalación natural de M vías se define como una intercalación con M archivos de entrada y solo uno de salida.

•Intercalación balanceada

Una intercalación natural de M-vías usa M+1 archivos, mientras que una intercalación balanceada de M-vías usa 2M archivos. En una intercalación balanceada, los datos son removidos una y otra vez entre un número igual de archivos de entrada y salida.

•Intercalación polifásica

Un tipo de intercalación desbalanceada es la intercalación de polifase, en la cual se utiliza un número constante de archivos de entrada. Una intercalación de polifase de M-vías usa 2M – 1 archivos de entrada.

•Intercalación de cascada

Una intercalación de cascada de gradoM usa 2M-1 archivos, después 2M – 2, luego 2M – 3, y así sucesivamente hasta 2M – m, es decir, hasta llegar al uso de 2 archivos de entrada como una intercalación polifásica, la de cascada descansa sobre una buena distribución inicial de la sublistas por la fase de ordenamiento interno.

Cada paso de intercalación comienza con 2M – 1 archivos de entrada a uno de salida. Cuando el archivo se vacía se convierte en un archivo de salida y el de salida anterior es apartado. Ahora 2M – 2 archivos son intercalados ene l nuevo archivo de salida. Cuando un archivo de entrada se vacía se vuelve archivo de salida y el archivo de salida anterior es retirado temporalmente. Eventualmente todos los archivos de entrada serán vaciados. Al terminar el paso de intercalación, cada registro habrá sido procesado una sola vez.

Después, comienza el siguiente paso de intercalación. De nuevo se intercalan 2M – 1 entradas después 2M – 2 y así sucesivamente hasta completar todas las fases de la intercalación.

3.5 Algoritmos por mezcla.

La ordenación de archivos se lleva a cabo cuando el volumen de los datos a tratar es demasiado grande y no caben en la memoria principal de la computadora.

La principal desventaja de esta ordenación es el tiempo de ejecución, debido a las sucesivas operaciones de entrada y salida.

Los dos métodos de ordenación externa más importantes son los basados en la mezcla directa y en la mezcla equilibrada.

Esta clasificación es llamada **mezcla** porque combina dos secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

Fueron desarrollados en 1945 por John Von Neumann. Aplican la técnica divide y vencerás, dividiendo la secuencia de datos en dos subsecuencias hasta que las subsecuencias tengan un único elemento, luego se ordenan mezclando dos subsecuencias ordenadas en una secuencia ordenada, en forma sucesiva hasta obtener una secuencia única ya ordenada. Si n = 1 solo hay un elemento por ordenar, sino se hace una ordenación de mezcla de la primera mitad con la segunda mitad. Las dos mitades se ordenan de igual forma.

3.5.1 Mezcla Directa.

El método de ordenación por mezcla directa es probablemente el más utilizado de los dos, por su fácil comprensión.

La idea central de este algoritmo consiste en la realización sucesiva de una partición y una fusión que produce secuencias ordenadas de longitud cada vez mayor. En la primera pasada la partición es de longitud 1 y la fusión o mezcla produce secuencias ordenadas de longitud 2. En la segunda pasada la partición es de longitud 2 y la fusión o mezcla produce secuencias ordenadas de longitud 4. Este proceso se repite duplicando el tamaño de las secuencias, hasta que la longitud de la secuencia para la partición sea mayor o igual que el número de elementos del archivo original.

Ejemplo 3.5.1

Supóngase que se desean ordenar las claves del archivo F. Para realizar tal actividad se utilizarán dos archivos auxiliares a los que se denominara F1 y F2.

F: 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

En total el archivo original contiene 15 elementos.

PRIMERA PASADA

Partición en secuencias de longitud 1.

F1: 09’ 14’ 29’ 31’ 04’ 13’ 72’ 61’

F2: 75’ 68’ 17’ 25’ 05’ 18’ 46’

Fusión en secuencias de longitud 2.

F: 09 75’ 14 68’ 17 29’ 25 31’ 04 05’ 13 18’ 46 72’ 61’

SEGUNDA PASADA

Partición en secuencias de longitud 2.

F1: 09 75’ 17 29’ 04 05’ 46 72’

F2: 14 68’ 25 31’ 13 18’ 61’

Fusión en secuencias de longitud 4.

F: 09 14 68 75’ 17 25 29 31’ 04 05 13 18’ 46 61 72’

TERCERA PASADA

Partición en secuencias de longitud 4.

F1: 09 14 68 75’ 04 05 13 18’

F2: 17 25 29 31’ 46 61 72

Fusión en secuencias de longitud 8.

F: 09 14 17 25 29 31 68 75’ 04 05 13 18 46 61 72’

CUARTA PASADA

Partición en secuencias de longitud 8.

F1: 09 14 17 25 29 31 68 75’

F2: 04 05 13 18 46 61 72’

Fusión en secuencias de longitud 16.

F: 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75

En este momento ya se encuentran ordenadas las claves del archivo y el algoritmo ya no continua porque la siguiente longitud de la secuencia de la partición seria 16 que sería mayor al número de elementos del archivo original.

La descripción formal del algoritmo de mezcal directa seria la siguiente.

Algoritmo:

```
MEZCLADIRECTA (F, F1, F2, N)
{El algoritmo ordena los elementos del archivo F por el método de mezcla directa. Utiliza dos archivos auxiliares F1 y F2. N el número total de elementos que contiene el archivo F}
{PART es una variable de tipo entero}

1. Hacer PART◄— 1
2. Repetir mientras (PART < N)
        Llamar al algoritmo PARTICIONA con F, F1, F2 y PART.
        Llamar al algoritmo FUSIONA con F, F1, F2 y PART.
        Hacer PART◄— PART * 2
3. {Fin del ciclo del paso 2}
```

Obsérvese que el algoritmo **mezcladirecta** requiere para su funcionamiento de dos algoritmos auxiliares, que se presentan a continuación.

```
PARTICIONA (F, F1, F2, PART)
{El algoritmo particiona el archivo F en dos archivos auxiliares, F1 y F2. PART es la longitud de la partición que se va realizar}
{K, L y R son variables de tipo entero}

1. Abrir el archivo F para lectura.
2. Abrir los archivos F1 y F2 para escritura.
3. Repetir mientras (no sea el fin de archivo de F)
        Hacer K◄— 0
        3.1 Repetir mientras (K < PART) y (no sea el fin de archivo de F)
                Leer R de F
                Escribir R en F1
                Hacer K◄— K + 1
        3.2 {Fin del ciclo del paso 3.1}
        Hacer L◄— 0
        3.3 Repetir mientras (L < PART) y (no sea el fin de archivo de F)
                Leer R de F
                Escribir R en F2
                Hacer L◄— L + 1
        3.4 {Fin del ciclo del paso 3.3}
4. {Fin del ciclo del paso 3}
```

```
FUSIONA (F, F1, F2, PART)
{El algoritmo fusiona los archivos auxiliares F1 y F2 en el archivo F. PART es
la longitud de la partición que se realizó}
{K, L, R1 y R2 son variables de tipo entero. B1 y B2 son variables de tipo
booleano}

1. Abrir el archivo F para escritura.
2. Abrir los archivos F1 y F2 para lectura.
3. Hacer B1← VERDADERO y B2← VERDADERO
4. Si (no es el fin de archivo de F1) entonces
        Leer R1 de F1
        Hacer B1← FALSO
5. {Fin del condicional del paso 4}
6. Si (no es el fin de archivo de F2) entonces
        Leer R2 de F2
        Hacer B2← FALSO
7. {Fin del condicional del paso 6}
8. Repetir mientras ((no sea el fin de archivo de F1) o (B1 = FALSO)) y
                    ((no sea el fin de archivo de F2) o (B2 = FALSO))
        Hacer K← 0 y L← 0
8.1 Repetir mientras ((K < PART) y (B1 = FALSO)) y ((L < PART) y (B2 = FALSO))
        8.1.1 Si R1 ≤ R2 entonces
                Escribir R1 en F
                Hacer K← K + 1 y B1← VERDADERO
                8.1.1.1 Si (no es el fin de archivo de F1) entonces
                        Leer R1 de F1
                        Hacer B1← FALSO
                8.1.1.2 {Fin del ciclo del paso 8.1.1.1}
                si no
                Escribir R2 en F
                Hacer L← L + 1 y B2 ←VERDADERO
                8.1.1.3 Si (no es el fin de archivo de F2) entonces
                        Leer R2 de F2
                        Hacer B2← FALSO
                8.1.1.4 {Fin del ciclo del paso 8.1.1.3}
        8.1.2 {Fin del condicional del paso 8.1.1}
8.2 {Fin del ciclo del paso 8.1}
8.3 Si K < PART entonces
        8.3.1 Repetir mientras (K < PART) y (B1← FALSO)
                Escribir R1 en F
                Hacer K← K + 1 y B1← VERDADERO
                8.3.1.1 Si (no es el fin de archivo de F1) entonces
                        Leer R1 de F1
                        Hacer B1← FALSO
                8.3.1.2 {Fin del condicional del paso 8.3.1.1}
        8.3.2 {Fin del ciclo del paso 8.3.1}
```

```
8.4 {Fin del condicional del paso 8.3}
8.5 Si L < PART entonces
        8.5.1 Repetir mientras (L < PART) y (B2 ←FALSO)
                Escribir R2 en F
                Hacer L← L + 1 y B2 ←VERDADERO
                8.5.1.1 Si (no es el fin de archivo de F2) entonces
                        Leer R2 de F2
                        Hacer B2← FALSO
                8.5.1.2 {Fin del condicional del paso 8.5.1.1}
        8.5.2 {Fin del ciclo del paso 8.5.1}
8.6 {Fin del condicional del paso 8.5}
9. {Fin del ciclo del paso 8}
10. Si B1 = FALSO entonces
        Escribir R1 en F
11. {Fin del condicional del paso 10}
12. Si B2 = FALSO entonces
        Escribir R2 en F
13. {Fin del condicional del paso 12}
14. Repetir mientras (no sea el fin de archivo de F1)
        Leer R1 de F1
        Escribir R1 en F
15. {Fin del ciclo del paso 14}
16. Repetir mientras (no sea el fin de archivo de F2)
        Leer R2 de F2
        Escribir R2 en F
17. {Fin del ciclo del paso 16}
```

3.5.2 Mezcla Equilibrada.

El método de ordenación por **mezcla equilibrada**, conocido también con el nombre de mezcla natural, es una optimización del método de mezcla directa.

La idea central de este algoritmo consiste en realizar las particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias de tamaño fijo previamente determinadas. Luego realiza la fusión de las secuencias ordenadas, alternativamente sobre dos archivos. Aplicando estas acciones en forma repetida se logrará que el archivo original quede ordenado.

Para la realización de este proceso de ordenación se necesitarán cuatro archivos. El archivo original F y tres archivos auxiliares a los que se denominarán F1, F2 y F3. De estos archivos, dos serán considerados de entrada y dos de salida; esto alternativamente con el objeto de realizar la fusión-partición. El proceso termina cuando en la realización de una fusión-partición el segundo archivo quede vacío.

Ejemplo 3.5.2

Supóngase que se desean ordenar las claves del archivo F utilizando el método de mezcla equilibrada. Para realizar tal actividad se utilizarán tres archivos auxiliares a los que se denominará F1, F2 y F3.

F: 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

En total el archivo original contiene 15 elementos.

PARTICION INICIAL

F2: 09 75’ 29’ 25’ 46 61’

F2: 14 68’ 17 31’ 04 05 13 18 72’

PRIMERA FUSION-PARTICION

F: 09 14 68 75’ 04 05 13 18 25 46 61 72’

F1: 17 29 31’

SEGUNDA FUSION-PARTICION

F2: 09 14 17 29 31 68 75

F3: 04 05 13 18 25 46 61 72’

TERCERA FUSION-PARTICION

F: 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75

F1:

Obsérvese que al realizar la tercera fusión-partición el segundo archivo queda vacío, por lo que puede afirmarse que el archivo original ya se encuentra ordenado.

A continuación, se vera la descripción formal del algoritmo de mezcal equilibrada.

Algoritmo:

```
MEZCLAEQUILIBRADA (F, F1, F2, F3)
{El algoritmo ordena los elementos del archivo F por el método de mezcla equilibrada. Utiliza tres archivos auxiliares F1, F2 y F3}
{BAND es una variable de tipo booleano}

1. Llamar al algoritmo ParticiónInicial con F, F2 y F3.
2. Hacer BAND← VERDADERO
3. Repetir
        3.1 Si BAND = VERDADERO entonces
                Llamar al algoritmo PARTICIONFUSION con F2, F3, F y F1.
                Hacer BAND← FALSO
        si no
                Llamar al algoritmo PARTICIONFUSION con F, F1, F2 y F3.
                Hacer BAND ←VERDADERO
        3.2 {Fin del condicional del paso 3.1}
4. Hasta que (F1 = VACIO) o (F3 = VACIO)
```

Obsérvese que el algoritmo **mezclaequilibrada** requiere para su funcionamiento de dos algoritmos auxiliares, que se presentan a continuación.

```
PARTICIONINICIAL (F, F2, F3)
{El algoritmo produce la partición inicial del archivo F en dos archivos auxiliares, F2 y F3}
{AUX y R son variables de tipo entero. B es una variable de tipo booleano}

1. Abrir el archivo F para lectura.
2. Abrir los archivos F2 y F3 para escritura.
3. Leer R de F.
4. Escribir R en F2.
5. Hacer B← VERDADERO y AUX← R
6. Repetir mientras (no sea el fin de archivo de F)
        Leer R de F
        6.1 Si R ≥ AUX entonces
                Hacer AUX← R
                6.1.1 Si B = VERDADERO entonces
```

```
                    Escribir R en F2
                si no Escribir R en F3
                6.1.2 {Fin del condicional del paso 6.1.1}
                si no Hacer AUX← R
                6.1.3 Si B = VERDADERO entonces
                    Escribir R en F3
                    Hacer B← FALSO
                Si no Escribir R en F2
                    Hacer B← VERDADERO
                6.1.4 {Fin del condicional del paso 6.1.3}
        6.2 {Fin del condicional del paso 6.1}
7. {Fin del ciclo del paso 6}
```

```
PARTICIONFUSION (FA, FB, FC, FD)
{El algoritmo produce la partición y la fusión de los archivos FA y FB, en los
archivos FC y FD}
{R1, R2 y AUX son variables de tipo entero. B, DELE1 y DELE2 son variables
de tipo booleano}

1. Abrir los archivos FA y FB para lectura.
2. Abrir los archivos FC y FD para escritura.
3. Hacer B← VERDADERO
4. Leer R1 de FA y R2 de FB.
5. Si R1 < R2 entonces
        Hacer AUX← R1
Si no Hacer AUX← R2
6. {Fin del condicional del paso 5}
7. Hacer DELE1← FALSO y DELE2← FALSO
8. Repetir mientras ((no sea el fin de archivo de FA) o (DELE1 ≠ VERDADERO))
y ((no sea el fin de archivo de FB) o (DELE2 ≠ VERDADERO))
        8.1 Si DELE1 = VERDADERO entonces
                Leer R1 de FA
                Hacer DELE1← FALSO
        8.2 {Fin del condicional del paso 8.1}
        8.3 Si DELE2 = VERDADERO entonces
                Leer R2 de FB
                Hacer DELE2← FALSO
        8.4 {Fin del condicional del paso 8.3}
        8.5 Si R1 < R2 entonces
                8.5.1 Si R1 ≥ AUX entonces
                Llamar al algoritmo AYUDA1 con AUX, R1, FC, FD y B
                Hacer DELE1← VERDADERO
                Si no
                        8.5.1.1 Si R2 ≥ AUX entonces
                        Llamar al algoritmo AYUDA1 con AUX, R2, FC, FD y B
```

```
                    Hacer DELE2◄─VERDADERO
                    Si no
                    Llamar al algoritmo AYUDA2 con AUX, R1, FC, FD y B
                    Hacer DELE1◄─ VERDADERO
                    8.5.1.2 {Fin del condicional del paso 8.5.1.1}
               8.5.2 {Fin del condicional del paso 8.5.1}
               si no
               8.5.3 Si R2 ≥ AUX entonces
               Llamar al algoritmo AYUDA1 con AUX, R2, FC, FD y B
               Hacer DELE2◄─ VERDADERO
               Si no
                    8.5.3.1 Si R1 ≥ AUX entonces
                    Llamar al algoritmo AYUDA1 con AUX, R1, FC, FD y B
                    Hacer DELE1◄─VERDADERO
                    Si no
                    Llamar al algoritmo AYUDA2 con AUX, R2, FC, FD y B
                    Hacer DELE2◄─ VERDADERO
                    8.5.3.2 {Fin del condicional del paso 8.5.3.1}
               8.5.4 {Fin del condicional del paso 8.5.3}
          8.6 {Fin del condicional del paso 8.5}
9. {Fin del ciclo del paso 8}
10. Si (DELE1 = VERDADERO) y (es el fin de archivo de FA) entonces
          Llamar al algoritmo AYUDA3 con R2, FB, FC, FD y B
11. {Fin del condicional del paso 10}
12. Si (DELE2 = VERDADERO) y (es el fin de archivo de FB) entonces
          Llamar al algoritmo AYUDA3 con R1, FA, FC, FD y B
13. {Fin del condicional del paso 12}
```

A su vez el algoritmo **particionfusión** hace uso también de tres algoritmos auxiliares: **ayuda1**, **ayuda2** y **ayuda3**. Esto con el fin de darle mayor modularidad y claridad al mismo. A continuación, se presentan estos algoritmos auxiliares.

```
AYUDA1 (AUX, R, FC, FD, B)
{El algoritmo escribe el elemento R en el archivo activo: FC o FD}

1. Hacer AUX◄─ R
2. Si B = VERDADERO entonces
          Escribir R en FC
          si no
          Escribir R en FD
3. {Fin del condicional del paso2}
```

```
AYUDA2 (AUX, R, FC, FD, B)
{El algoritmo escribe el elemento R en el archivo desactivado y luego activa al mismo}

1. Hacer AUX◄— R
2. Si B = VERDADERO entonces
        Escribir R en FD
        Hacer B◄— FALSO
        si no
        Escribir R en FC
        Hacer B◄— VERDADERO
3. {Fin del condicional del paso2}
```

```
AYUDA3 (R, F, FC, FD, B)
{El algoritmo escribe el elemento R y los elementos de F pendientes de tratar en el archivo FC o FD}

1. Si R ≥ AUX entonces
        Llamar al algoritmo AYUDA1 con R, FC, FD y B
        si no
        Llamar al algoritmo AYUDA2 con R, FC, FD y B
2. {Fin del condicional del paso 1}
3. Repetir mientras (no sea el fin de archivo de F)
        Leer R de F
        3.1 Si R ≥ AUX entonces
        Llamar al algoritmo AYUDA1 con AUX, R, FC, FD y B
        si no
        Llamar al algoritmo AYUDA2 con AUX, R, FC, FD y B
        3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso3}
```

3.6 Algoritmo RadixSort.

Este método se puede considerar como una generalización de la clasificación por urnas. Aprovecha la estrategia de la forma más antigua de clasificación manual, consiste en hacer diversos montones de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra si la ordenación fuera alfabética) en la misma posición; estos montones se recogen en orden ascendente y se reparte de nuevo en montones según el siguiente dígito de la clave a clasificar.

La idea clave de la ordenación **radixsort** (también llamada por residuos) es clasificar por urnas primero respecto al dígito de menor peso (menos significativo) d_k, después concatenar las urnas, clasificar de nuevo respecto al siguiente dígito d_{k-1}, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo d_l, en ese momento la secuencia estará ordenada. La concatenación de las urnas consiste en enlazar el final de una con el frente de la siguiente.

Ejemplo 3.6

Para centrarnos en lo que estamos diciendo, supóngase que tenemos que ordenar cierto número de fichas identificadas por tres dígitos:

345, 721, 425, 572, 836, 467, 672, 194, 365, 216, 891, 746, 431, 834, 247, 529, 389

Primeramente, atendiendo el dígito de menor peso (**unidades**) se reparten las fichas en montones del 0 al 9, y se colocan de acuerdo al dígito con el que coincidan; las urnas de números cuyo dígito es 0, 3 y 8 quedan vacías:

	431				365	746			
	891	672		834	425	216	247		389
	721	572		194	345	836	467		529
0	**1**	**2**	3	**4**	**5**	**6**	**7**	8	**9**

Tomando los montones en orden, la secuencia de fichas queda así:

721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 216, 746, 467, 247, 529, 389

De esta secuencia podemos decir que esta ordenada con respecto al dígito de menor peso, que son las unidades.

Continuando, distribuimos la secuencia de fichas en montones con respecto al segundo dígito (**decenas**), ahora las urnas de números cuyo dígito es 0 y 5 son las que quedan vacías:

		5**2**9	8**3**6	2**4**7					
		4**2**5	8**3**4	7**4**6		4**6**7	6**7**2		1**9**4
	2**1**6	7**2**1	4**3**1	3**4**5		3**6**5	5**7**2	3**8**9	8**9**1
0	**1**	**2**	**3**	**4**	5	**6**	**7**	**8**	**9**

Tomando nuevamente los montones en orden, la secuencia de fichas queda de la siguiente manera:

216, 721, 425, 529, 431, 834, 836, 345, 746, 247, 365, 467, 572, 672, 389, 891, 194

Esta secuencia de fichas ya la tenemos ordenada con respecto a los dos últimos dígitos, es decir, respecto a las decenas.

Por último, se distribuyen de nuevo las fichas con respecto al tercer dígito (**centenas**), y las urnas de números cuyo dígito es 0 y 9 quedarán vacías:

			389	**4**67				**8**91	
		247	**3**65	**4**31	**5**72		**7**46	**8**36	
	194	**2**16	**3**45	**4**25	**5**29	**6**72	**7**21	**8**34	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9

Finalmente, tomando los montones en orden, la secuencia de fichas queda ya ordenada:

194, 216, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721, 746, 834, 836, 891

Las urnas se representan mediante un vector de listas enlazadas. En el caso de que la clave respecto a la que se ordena sea un entero, se tendrán 10 urnas, numeradas de 0 a 9. Las listas tienen una realización dinámica, cada lista se mantiene con dos punteros, uno al frente y otro al final de la lista, así el añadir un nuevo registro es inmediato ya que se enlaza por el final, de igual forma, concatenar las urnas consistirá en enlazar el final de una con el frente de la siguiente.

Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que represente a la letra a hasta la z.

Para el caso de que clave sea entera, en primer lugar, se determina el máximo número de dígitos que puede tener la clave. En un bucle de tantas iteraciones como

máximo de dígitos se realizan las acciones de distribuir por urnas los registros, concatenar...

La distribución por urnas exige obtener el dígito del campo clave que se encuentra en la posición definida por el bucle externo, dicho dígito será el índice de la urna.

A continuación, se vera la descripción formal del algoritmo de radix sort.

Algoritmo:

```
RadixSort (X, M)
{El algoritmo ordena los elementos del arreglo utilizando el método de radix
sort. X es el nombre del arreglo de números, y M el número de dígitos que
contiene cada elemento del arreglo}
{rear y front son arreglos de tipo entero, node es una lista enlazada}
{first, i, k, p, y, expon, q, j, r son variables de tipo entero}

1. Repetir con i desde 1 hasta X.Length -1
        Hacer node[i].info← X[i]
        y node[i].next← i + 1
2. {Fin del ciclo del paso 1}
Hacer node[X.Length].info← X[X.Length],
node[X.Length].next ←-1 y first← 1
3. Repetir con k desde 1 hasta M
        3.1 Repetir con i desde 0 hasta 9
                Hacer rear[i] ←-1;
        3.2 {Fin del ciclo del paso 3.1}
        3.3 Repetir con i desde 0 hasta 10
                Hacer front[i] ←-1;
        3.4 {Fin del ciclo del paso 3.3}
        3.5 Repetir mientras (first ≠ -1)
                Hacer p← first,
                first← node[first].next,
                y ← node[p].info,
                expon ← 1,
                3.5.1 Repetir con i desde 1 hasta k
                        Hacer expon ← expon * 10
                3.5.2 {Fin del ciclo del paso 3.5.1}
                j← y / expon % 10;
                q← rear[j],
                3.5.3 Si q = -1 entonces
                        front[j]← p
                si no node[q].next ← p
                3.5.4 {Fin del condicional del paso 3.5.3}
                rear[j]← p
```

```
        3.6 {Fin del ciclo del paso 3.5}
        Hacer j← 0
        3.7 Repetir mientras (j ≤ 9) y (front[j] = -1)
                Hacer j← j + 1
        3.8 {Fin del ciclo del paso 3.7}
        Hacer first ← front[j] y p← j
        3.9 Repetir mientras (j ≤ 9)
                Hacer r← j + 1
                3.9.1 Repetir mientras (r ≤ 9) y (front[r] = -1)
                        Hacer r ← r + 1
                3.9.2 {Fin del ciclo del paso 3.9.2}
                3.9.3 Si r ≤ 9 entonces
                        p ← r,
                        node[rear[j]].next ← front[r]
                3.9.4 {Fin del condicional del paso 3.9.3}
                Hacer j ← r
        3.10 {Fin del ciclo del paso 3.9}
        Hacer node[rear[p]].next← -1
4. {Fin del ciclo del paso 3}
5. Repetir con i desde 1 hasta X.Length
        Hacer X[i]← node[first].info,
        first← node[first].next
6. {Fin del ciclo del paso 5}
```

4 – Complejidad de los Algoritmos de Ordenamiento.

Los algoritmos de ordenamiento han sido fuente de gran interés e investigación desde el inicio de la computación, aunque su resolución es relativamente simple a lo largo de la historia se ha diseñado numerosas técnicas para lograr el algoritmo más eficiente que logre ordenar una lista de la manera más rápida y eficiente.

La gran variedad y cantidad de algoritmos de ordenamiento diferentes los hace un buen tema para el aprendizaje de cualquier lenguaje de programación. Para programar uno de estos algoritmos es necesario aplicar conceptos de arreglos, operaciones de comparación y operaciones aritméticas.

A continuación, se listan algunas de las características por las cuales pueden clasificarse los algoritmos de búsqueda

COMPLEJIDAD COMPUTACIONAL

La complejidad computacional es el mejor, promedio y peor comportamiento que tiene un determinado algoritmo dependiendo del tamaño de la lista. La cantidad de pasos a realizar dependerá en muchos casos del nivel de desorden inicial de la lista, por lo que son necesarios estos tres valores para comparar el rendimiento de los diferentes algoritmos.

Para describir la complejidad computacional de un algoritmo de ordenamiento para una lista de tamaño n se utiliza **la notación *O()*** que indica la cantidad de operaciones necesarias para finalizar el algoritmo correctamente. Por ejemplo, *O(n)* significa que el algoritmo necesita tantos pasos como elementos en la lista para finalizar.

USO DE MEMORIA

Aunque la mayoría de los algoritmos no utilizan más memoria que un espacio extra que el ocupado por la lista a ordenar algunos requieren espacio adicional.

Aquellos algoritmos que solo requieren un espacio adicional de memoria *O(1)* son denominados algoritmos in situ o in-place. Una definición más amplia de este término incluye también aquellos que ocupan una memoria igual a *O(log(n))*

ESTABILIDAD

Cuando los elementos de la lista a ordenar tienen varias características, pero solo se utiliza una de ellas para ordenar pueden darse dos tipos de ordenamiento dependiendo del algoritmo. En el caso que el algoritmo mantenga el orden relativo que tenía la lista original se tratara de un algoritmo estable. En este otro caso, en cambio, al ordenar es posible que el orden relativo se modifique, en este caso será un algoritmo inestable.

La estabilidad del algoritmo de ordenamiento tendrá relevancia dependiendo de la naturaleza de la lista, en el caso de que la lista tenga una sola característica o que no existan elementos con las mismas características entonces el resultado de un algoritmo estable no variará con el de uno inestable.

METODO

El método es el proceso elegido para lograr el ordenamiento de la lista, entre ellos puede nombrarse:

- Inserción.
- Intercambio.
- Selección.
- Unión.
- Mezcla.
- Particionado.

SERIAL/PARALELO

Aunque la mayoría de los algoritmos presentados estén diseñados para operaciones en serie existen otros que aprovechan las ventajas del procesamiento paralelo. Estos permiten realizar operaciones en paralelo sobre la lista permitiendo reducir el tiempo requerido.

COMPARATIVOS/NO COMPARATIVOS

Se les llama algoritmos comparativos a aquellos que comparan de dos elementos a la vez utilizando un operador de comparación. Otros tipos de algoritmos como los de ordenamiento entero utiliza operaciones aritméticas sobre las claves para obtener el ordenamiento.

Los métodos de ordenación para su estudio se suelen dividir en dos grandes grupos:

4.1 - Los métodos Directos o Simples: Inserción (o inserción directa), Burbuja, Selección Directa, y ShellSort, en dónde el último es una extensión al método de inserción, siendo más rápido.

a) Análisis de eficiencia del método de Inserción Directa

El número **mínimo** de comparaciones y movimientos entre claves se produce cuando los elementos del arreglo ya están ordenados.

El número **máximo** de comparaciones y movimientos entre claves se produce cuando los elementos del arreglo están en orden inverso.

El número **promedio** de comparaciones y movimientos entre las claves se da cuando los elementos aparecen en el arreglo en forma aleatoria.

Así, por ejemplo, si se tiene que ordenar un arreglo que contiene 500 elementos:

a) Si el arreglo se encuentra ordenado serán necesarias
   499 comparaciones.
   0 movimientos de elementos.
b) Si los elementos del arreglo se encuentran dispuestos en forma aleatoria se realizarán
   62,624 comparaciones, en promedio.
   62,375 movimientos de elementos, en promedio.
c) Si los elementos del arreglo se encuentran en orden inverso serán necesarias
   124,750 comparaciones.
   124,750 movimientos de elementos.

El tiempo necesario para ejecutar el algoritmo de inserción directa es $O(n^2)$, donde n es el número de elementos del arreglo. La complejidad del algoritmo es cuadrática $O(n^2)$, debido a que todo el proceso se controla con dos bucles anidados que en el peor de los casos realizan N-1 iteraciones.

Análisis del algoritmo.

•**Estabilidad:** Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto, es estable.

•**Requerimientos de Memoria:** Una variable adicional para realizar los intercambios.

•**Tiempo de Ejecución:** Para una lista de n elementos el ciclo externo se ejecuta n-1 veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración,

2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad $O(n^2)$.

Ventajas:

1.Fácil implementación.

2.Requerimientos mínimos de memoria.

Desventajas:

1.Lento.

2.Realiza numerosas comparaciones.

Es un algoritmo lento, pero puede ser de utilidad para listas que están semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

b) Análisis de eficiencia del método de la Burbuja

El número de comparaciones en el método de la burbuja es fácilmente contabilizable; en la primera pasada realizamos (N-1) comparaciones, en la segunda pasada (N-2) comparaciones, en la tercera pasada (N-3) comparaciones y así sucesivamente hasta llegar a 2 y 1 comparaciones entre las claves, siendo N el número de elementos del arreglo.

Respecto al número de movimientos entre los elementos del arreglo, estos dependen de si el arreglo esta ordenado, desordenado o en orden inverso.

Así, por ejemplo, si se tiene que ordenarse un arreglo que contiene 500 elementos:

a) Si el arreglo se encuentra ordenado serán necesarias
   124,750 comparaciones.
   0 movimientos de elementos.
b) Si los elementos del arreglo se encuentran dispuestos en forma aleatoria se realizarán
   124,750 comparaciones.
   187,125 movimientos de elementos.
c) Si los elementos del arreglo se encuentran en orden inverso serán necesarias
   124,750 comparaciones.
   374,250 movimientos de elementos.

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja?

Esto dependerá de la versión utilizada. En la versión más simple se hacen n-1 pasadas y n-1 comparaciones en cada pasada. Por consiguiente, el número de comparaciones es (n-1) * (n-1) = n2 - 2n + 1, es decir la complejidad es ***O(n^2)***.

Si se tienen en cuenta las versiones mejoradas del algoritmo, entonces se tendrá una eficiencia diferente para cada algoritmo. En el mejor de los casos, la ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y por tanto su complejidad es ***O(n)***. En el caso peor se requieren (n-i-1) comparaciones y (n-i-1) intercambios.

La ordenación completa requiere n(n – 1)/2 comparaciones y un número similar de intercambios. Por lo tanto, la complejidad para el caso peor es ***O(n^2)*** comparaciones y ***O(n^2)*** intercambios.

Análisis del algoritmo.

•**Estabilidad:** Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto, es estable.

•**Requerimientos de Memoria:** Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.

•**Tiempo de Ejecución:** El ciclo interno se ejecuta n veces para una lista de n elementos. El ciclo externo también se ejecuta n veces. Es decir, la complejidad es n * n = *O(n^2)*. El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo *O(n^2)*.

Ventajas:

1.Fácil implementación y programación.

2.No requiere memoria adicional, trabaja in situ.

3. Es fácil de comprender, es el más extendido para efectos de enseñanza.

4. Es bastante sencillo

5. En un código reducido se realiza el ordenamiento

6. Es eficaz

7. Su bucle interno es extremadamente corto.

Desventajas:

1.Muy lento.

2.Realiza numerosas comparaciones.

3.Realiza numerosos intercambios.

4.Este algoritmo es uno de los más pobres en rendimiento.

5. Es uno de los menos eficientes y por ello, normalmente, se aprende su técnica, pero no se utiliza.

6. Consume bastante tiempo de computadora

7. Requiere muchas lecturas/escrituras en memoria

A pesar de que el ordenamiento de la burbuja es uno de los algoritmos más sencillos de implementar, su orden *O(n2)* lo hace muy ineficiente para usar en listas que no tengan un número reducido de elementos. Incluso entre los algoritmos de ordenamiento de orden *O(n2)*, otros procedimientos como el ordenamiento por inserción son considerados más eficientes. Dada su simplicidad, el ordenamiento de burbuja es utilizado para introducir el concepto de algoritmo, o de algoritmo de ordenamiento para estudiantes de ciencias de la computación. El ordenamiento de burbuja es asintóticamente equivalente, en tiempos de ejecución con el ordenamiento por inserción en el peor de los casos, pero ambos algoritmos difieren principalmente en la cantidad de intercambios que son necesarios realizar.

c) Análisis de eficiencia del método de Selección Directa

El análisis de este método es relativamente simple. Debe tenerse en cuenta que el número de comparaciones entre elementos es independiente de la disposición inicial de los mismos en el arreglo. En la primera pasada se realizan (N-1) comparaciones, en la segunda pasada (N-2) comparaciones y así sucesivamente hasta 2 y 1 comparaciones, en la penúltima y ultima pasada respectivamente.

Respecto al número de intercambios, siempre será N-1 a excepción de que se tenga incorporado en el algoritmo alguna técnica para prevenir el intercambio de un elemento consigo mismo.

Así, por ejemplo, si se tiene que ordenarse un arreglo que contiene 500 elementos:

Siempre se efectuarán 124 750 comparaciones y 499 movimientos de elementos.

El análisis del algoritmo de selección es muy claro, ya que requiere un número fijo de comparaciones e intercambios, que sólo dependen del tamaño de la lista o arreglo y no de la distribución inicial de los datos.

El tiempo de ejecución del algoritmo es proporcional a n^2, es $\boldsymbol{O(n^2)}$, aun cuando es más rápido que otros métodos directos.

Análisis del algoritmo.

Análisis del Costo Computacional

El ciclo externo se ejecuta n veces para una lista de n elementos, o sea que para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones. c(n)= (n2-n)/2 Cada búsqueda requiere comparar todos los elementos no clasificados, de manera que el número de comparaciones c(n) no depende del orden de los términos, si no del número de términos; por lo que este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad es del orden ***n2***.

Estabilidad.

Puede que exista algo de discrepancia en cuanto a si es o no estable este algoritmo, pero en realidad esta implementación parece ser bastante estable. Se puede verificar esto ordenando un conjunto de datos que tenga un par de ellos con la misma clave. Se vera claramente que el orden relativo entre ellos es conservado.

Ventajas:

1.Es fácil su implementación.

2.No requiere memoria adicional.

3.Realiza pocos intercambios.

4.Tiene un rendimiento constante, pues existe poca diferencia entre el peor y el mejor caso.

Desventajas:

1.Es lento y poco eficiente cuando se usa en listas grandes o medianas.

2.Realiza numerosas comparaciones.

d) Análisis de eficiencia del método ShellSort

Aunque este método es ineficiente para grandes cantidades de información, es uno de los algoritmos más rápidos para ordenar pequeñas cantidades de elementos.

Otra ventaja de este algoritmo es que requiere pequeñas cantidades de memoria.

Su tiempo promedio de ejecución es $O(n^{7/6})$ y en el peor de los casos $O(n^{4/3})$.

Si bien Shell no es el algoritmo más eficiente para ordenar arreglos, comparado con la complejidad *O(n log n)* de los algoritmos Quicksort, Mergesort y Heapsort, es un algoritmo mucho más fácil de programar.

Su simplicidad radica en que deriva del algoritmo más simple para ordenar, el Insert Sort. Además, su complejidad promedio en tiempo de $O(n^{9/8})$, y su complejidad en espacio de $O(n)$, que lo hacen un buen candidato para resolver el problema de ordenamiento en conjuntos de menos de 100,000 elementos.

Ventajas:

1.No requiere memoria adicional.

2.Mejor rendimiento que el método de Inserción clásico

3. Es un algoritmo muy simple teniendo un tiempo de ejecución aceptable.

4. Es uno de los algoritmos más rápidos.

5. Es de fácil implementación.

Desventajas:

1.Implementación algo confusa.

2.Realiza numerosas comparaciones e intercambios.

3. Su complejidad es difícil de calcular y depende mucho de la secuencia de incrementos que utilice.

4. Shell Sort es un algoritmo no estable porque se puede perder el orden relativo inicial con facilidad.

5. Es menos eficiente que los métodos Merge, HeapSort y QuickSort.

4.2 - Los métodos Indirectos o Complejos: el QuickSort (ordenación rápida), el HeapSort (montículo), MergeSort (ordenación por mezcla), y el RadixSort

a) Análisis de eficiencia del método QuickSort

El algoritmo del quicksort es el método más rápido de ordenación interna que existe en la actualidad. Diversos estudios realizados sobre el comportamiento del mismo demuestran que si se escoge en cada pasada el elemento que ocupa la posición central del conjunto de datos a analizar, el número de pasadas necesarias para ordenarlo es del orden de ***log n***, con respecto al número de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada realizara (n-1) comparaciones, en la segunda pasada realizará (n-1)/2 comparaciones pero en dos conjuntos diferentes, en la tercera pasada realizará (n-1)/4 comparaciones pero en cuatro conjuntos diferentes, y así sucesivamente.

El análisis general de la eficiencia de quicksort es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que n (número de elementos de la lista) es una potencia de 2, $n = 2k$ ($k = \log_2 n$).

Además, supongamos que el pivote es el elemento central de cada lista, de modo que quicksort divide la sublista en dos sublistas aproximadamente iguales.

En la primera exploración o recorrido se hacen n-1 comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño n/2. En la siguiente fase, el proceso de cada sublista requiere aproximadamente n/2 comparaciones. Las comparaciones totales de esta fase son 2(n/2) = n. La siguiente fase procesa cuatro sublistas que requieren un total de 4(n/4) comparaciones, etc. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$n + 2(n/2) + 4(n/4) + ... + n(n/n) = n + n + ... + n$

$= n * k = n * \log_2 n$

Para una lista normal la complejidad de quicksort es $0(n\log_2 n)$. El caso ideal que se ha examinado se realiza realmente cuando la lista (el arreglo) está ordenado en orden ascendente. En este caso el pivote es precisamente el centro de cada sublista.

Si el array está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada y el algoritmo tiene la complejidad ***0(nlog$_2$n)***.

El escenario del caso peor de quicksort ocurre cuando el pivote cae en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene n-1 elementos. En el siguiente recorrido, la sublista mayor requiere n-1 comparaciones y produce una sublista de n-2 elementos, etc.

El número total de comparaciones es: n + n-1 + n-2 +...+ 2 = (n-1)(n+2)/2

Entonces la complejidad en el peor caso sería ***0(n2)***.

En general el algoritmo de ordenación tiene como complejidad media ***0(nlog$_2$n)*** siendo posiblemente el algoritmo más rápido en la actualidad.

Análisis del algoritmo.

•**Estabilidad:** No es estable.

•**Requerimientos de Memoria:** No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.

•**Tiempo de Ejecución:**

Caso promedio. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto, la complejidad se define en forma recurrente como:

f(1) = 1

f(n) = n + 2 f(n/2)

La forma cerrada de esta expresión es:

f(n) = n log$_2$n

Es decir, la complejidad es *O(n log$_2$n)*.

El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a *O(n^2)*.

Ventajas:

1.Muy rápido

2.No requiere memoria adicional durante la ejecución (in-place processing).

3. Requiere de pocos recursos en comparación a otros métodos de ordenamiento.

4. En la mayoría de los casos, se requiere aproximadamente N log N operaciones.

5. Su ciclo interno es extremadamente corto.

6. Este es un algoritmo que puedes utilizar en la vida real. Es muy eficiente. En general será la mejor opción.

Desventajas:

1.Implementación un poco más complicada.

2.La recursividad (utiliza muchos recursos).

3.Mucha diferencia entre el peor y el mejor caso.

4. Se complica la implementación si la recursión no es posible.

5. Un simple error en la implementación puede pasar sin detección, lo que provocaría un rendimiento pésimo.

6. No es útil para aplicaciones de entrada dinámica, donde se requiere reordenar una lista de elementos con nuevos valores.

7. Se pierde el orden relativo de elementos idénticos.

b) Análisis de eficiencia del método HeapSort

El análisis del método del montículo (heapsort), como en general de los métodos logarítmicos, es complejo. Debe tenerse en cuenta tanto la fase de construcción del montículo como la fase donde se elimina repetidamente la raíz del mismo, para finalmente obtener el arreglo ordenado.

A diferencia de lo que pudiera pensarse, ya que en la fase de construcción del montículo los elementos mayores se cargan hacia la izquierda y en la fase de eliminación de la raíz los elementos mayores se cargan hacia la derecha, este es un método muy rápido, sobre todo para valores grandes de N.

Los estudios realizados al respecto demuestran que el tiempo de ejecución del algoritmo en ambas fases es de ***O(n * log n)***. Aunque el método del montículo, puede ser un poco más lento que el quicksort, es el único que garantiza que aun en el peor caso su tiempo de ejecución es proporcional a ***(n * log n)***, ***O(n * log n)***. Recuérdese que el tiempo de ejecución del método quicksort, en el peor caso, es proporcional a n^2, $\boldsymbol{O(n^2)}$.

Ventajas:

Su desempeño es en promedio tan bueno como el Quicksort y se comporta mejor que este último en los peores casos.

Desventajas:

Aunque el Heapsort tiene un mejor desempeño general que cualquier otro método presentado de clasificación interna, es bastante complejo de programar.

c) Análisis de eficiencia del método MergeSort

MergeSort es un ordenamiento estable, paraleliza mejor, y es más eficiente manejando medios secuenciales de acceso lento. MergeSort es a menudo la mejor opción para ordenar una lista enlazada: en esta situación es relativamente fácil implementar MergeSort de manera que sólo requiera *O(1)* espacio extra, y el mal rendimiento de las listas enlazadas ante el acceso aleatorio hace que otros algoritmos como quicksort den un bajo rendimiento, y para otros como heapsort sea algo imposible. El algoritmo de MergeSort es un algoritmo de ordenamiento que presenta algunas propiedades interesantes. En primer lugar, el orden del algoritmo es (n log n), y esto ocurre tanto en el peor caso, como en el mejor caso, ya que el tiempo que insume el algoritmo no depende de la disposición inicial de los elementos. En segundo lugar, es un algoritmo que requiere de un espacio extra de almacenamiento para poder funcionar.

La complejidad de tiempo en el peor de los casos de la ordenación por fusión es ***O(n log(n))***, dónde n es el tamaño de la entrada.

La relación de recurrencia es: T(n) = 2T(n/2) + cn = O(n log(n))

El espacio auxiliar requerido por el algoritmo de clasificación por fusión es *O(n)* para la pila de llamadas.

Análisis del algoritmo.

- Es Estable.
- Memoria Auxiliar ***O(n)***.
- No ordena en el lugar.
- Su complejidad es ***O(n log n)***.

Ventajas:

1.Método de ordenamiento estable mientras la función de mezcla sea implementada correctamente.

2.Muy eficiente cuando la cantidad de registros a acomodar es de índice bajo, en caso contrario gasta el doble del espacio que ocupan inicialmente los datos.

3.Efectivo para conjunto de datos a los que se puede acceder secuencialmente (arreglos, vectores, etc.)

Desventajas:

1.Principal desventaja: está definido recursivamente.

2.Si se deseara implementarlo no recursivamente se tendría que emplear una pila y se requeriría un espacio adicional de memoria para almacenarla.

d) Análisis de eficiencia del método RadixSort

El vector de registros *X* tiene *n* elementos. Al ser el campo clave entero el número de urnas es *d* = 10. Además, el número de dígitos de que consta el campo clave va a ser *K*. Con estas premisas y teniendo en cuenta los bucles anidados de que consta el algoritmo, tenemos que el tiempo de ejecución es ***O(K * n + K * d)***.

Es muy rápido en comparación con otros métodos de ordenación, ya que posee una complejidad por lo general lineal ***O(Kn)***. Ya que procesa sus elementos de forma individual, y no compara elementos del arreglo.

La idea clave del método es clasificar por urnas, primero respecto al dígito de menor peso (menos significativo) d1, después concatenar las urnas, clasificar de nuevo respecto al siguiente dígito d2, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo dn. En ese momento la secuencia estará ordenada.

Ventajas

1.Es estable, preservando el orden de elementos iguales.

2.Funciona en un tiempo lineal, en comparación de varios otros métodos de ordenamiento. El tiempo de ordenar cada elemento es constante, ya que no se hacen comparaciones entre elementos.

3. Es particularmente eficiente cuando se tratan con grandes grupos de números cortos.

Desventajas

1.No funciona tan bien cuando los números son muy largos, ya que el total de tiempo es proporcional a la longitud del número más grande y al número de elementos a ordenar.

5 – Conclusiones.

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

• Tener un mayor conocimiento de los métodos de ordenamiento es fundamental para resolver problemas en menos tiempo, mejorar el razonamiento y análisis lógico y ser más creativos. Cuanto más dominemos estos algoritmos más sabios seremos en el mundo de la programación ya que entenderemos lo que realmente hay detrás de muchos programas informáticos.

• El ordenamiento de Burbuja tiene una complejidad $O(n^2)$. En este algoritmo el número de repeticiones depende de n (términos del vector) y no del orden de los términos, esto significa que, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Cuando una lista ya está ordenada, a diferencia de otros métodos que pasan por la lista una vez y encontraran que no hay necesidad de intercambiar las posiciones de los elementos, el método de ordenación por burbuja está forzado a pasar por dichas comparaciones, lo que hace que su complejidad sea cuadrática en el mejor de los casos. Esto lo cataloga como el algoritmo más ineficiente que existe, aunque para muchos programadores sea el más sencillo de implementar.

El algoritmo ShellSort es un algoritmo de ordenación interna muy sencillo pero muy ingenioso, basado en comparaciones e intercambios, y con unos resultados radicalmente mejores que los que se pueden obtener con el método de la burbuja.

• Luego de analizarse el algoritmo QuickSort se pudo comprobar que es uno de los mejores métodos de ordenación ya que a pesar de no ser tan sencillo su código tampoco es tan complicado, resultando ser un algoritmo con una estructura elegante y con buena eficiencia. Con este método queda claro que en muchas ocasiones es mejor dividir para un óptimo desarrollo.

6 – Ejercicios.

ORDENACION INTERNA.

1. En un arreglo se guardan los apellidos de N alumnos. Aplique el método de la **burbuja** para ordenar el arreglo en forma ascendente, de tal manera que: $Ap_1 \leq Ap_2 \leq \ldots \leq Ap_n$
2. Resuelva el problema 1 aplicando el método de **inserción directa**.
3. Resuelva el problema 1 aplicando el método de **selección directa**.
4. Compare el tiempo de ejecución de los ejercicios 1, 2 y 3 para distintos valores de N, y documente los resultados obtenidos.
5. Dado un arreglo de N números enteros generados aleatoriamente, ordénelo en forma descendente aplicando el método de **shellsort**.
6. Resuelva el problema 5 aplicando el método de **quicksort**.
7. Resuelva el problema 5 aplicando el método de **heapsort**.
8. Compare el tiempo de ejecución de los ejercicios 5, 6 y 7 para distintos valores de N, y documente los resultados obtenidos.
9. Se tienen 3 arreglos paralelos. El primero de ellos almacena las matrículas de N alumnos; el segundo los nombres completos de los N alumnos; y el tercero las calificaciones finales obtenidas de los N alumnos en cierta materia. Los elementos de los arreglos se corresponden.
   a) Aplique el método de **inserción directa** para ordenar los arreglos, de tal manera que queden ordenados ascendentemente por la matrícula del alumno.
   b) Aplique el método **quicksort** para ordenar los arreglos, de tal manera que queden ordenados en forma descendente por la calificación final obtenida por el alumno en la materia.
   c) En ambos casos presente los resultados obtenidos.
10. En cierta empresa se maneja una lista de precios de los N artículos que se venden. De cada artículo se tiene la siguiente información:

- Clave del artículo.
- Nombre del artículo.
- Precio del artículo.

   a) Ordene el arreglo en forma ascendente según el campo “clave del artículo”. Aplique el método **selección directa**.
   b) Ordene el arreglo en forma descendente según el campo “precio del artículo”. Aplique el método **heapsort**.

11. Escriba un programa que genere un vector de 1000 números aleatorios del rango 1 al 500. Realice la ordenación del vector ascendentemente por el método **radixsort**.

- Imprimir el arreglo ordenado.
- Calcule y documente el tiempo empleado en la ordenación.

12. La fecha de nacimiento de una persona está representada por el registro: día = 1..31; mes = 01..12; año = 2000..2024. Escribir un programa que tenga como entrada el nombre completo y la fecha de nacimiento de los alumnos de una institución. Obtener como salida el listado de los alumnos en el orden de nacimiento utilizando el método de ordenamiento **radixsort**. Calcular y desplegar su tiempo de ejecución.

ORDENACION EXTERNA.

13. Se tienen dos archivos A1 y A2 con información sobre los recitales efectuados en un teatro en los últimos dos años. Cada registro de los archivos tiene los siguientes campos:

- Nombre del cantante u orquesta que ofreció el recital.
- Fecha de la presentación.

Los dos archivos están ordenados en forma ascendente según el primer campo. Escriba un programa que **intercale** los 2 archivos, formando el archivo final RECITALES.

14. Se tienen dos archivos ordenados con los nombres de los estudiantes de una escuela. No se han actualizado de la misma forma los dos archivos, habiéndose dado de alta algunos alumnos en un archivo, pero no en el otro. Escriba un programa que obtenga un tercer archivo, también ordenado, **intercalando** la información de los dos archivos existentes. (No deben quedar elementos repetidos en el archivo final.)
15. En el archivo EMPLEADOS se tiene la información sobre los empleados de una empresa, Los datos almacenados por cada empleado son:

- Nombre completo.
- Antigüedad.
- Categoría.
- Sueldo.

Ordene el archivo según el campo “nombre”.

a) Aplique el método **mezcla directa**.
b) Aplique el método **mezcla equilibrada**.
Compare el tiempo de ejecución de las soluciones a y b.

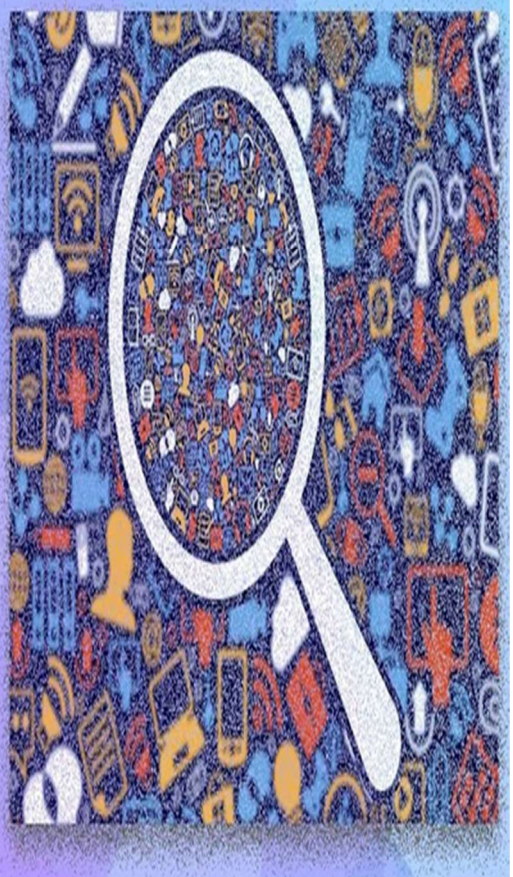
16. Se tiene un archivo con la siguiente información sobre los huéspedes de un hotel:

- Número de habitación.
- Nombre del huésped.
- Fecha de llegada.

Ordene el archivo según el campo “número de habitación”.

a) Aplique el método **mezcla directa**.
b) Aplique el método **mezcla equilibrada**.
Compare el tiempo de ejecución de las soluciones a y b.

¿QUÉ ES UN ALGORITMO DE BÚSQUEDA?



7 - Algoritmos de Búsqueda.

Este capítulo se dedica al estudio de una de las operaciones más importantes en el procesamiento de información: **la búsqueda**. Esta operación se utiliza básicamente para recuperar datos que se habían almacenado con anticipación. El resultado puede ser de éxito si se encuentra la información deseada, o de fracaso, en caso contrario.

La búsqueda ocupa una parte importante de nuestra vida. Prácticamente todo el tiempo estamos buscando algo. El mundo en que se vive hoy día es desarrollado, automatizado, y la información representa un elemento de vital importancia. Es fundamental estar informados y, por lo tanto, buscar y recuperar información. Por ejemplo, se buscan números telefónicos en un directorio, ofertas laborales en un periódico, libros en una biblioteca, etcétera.

En los ejemplos mencionados, la búsqueda se realiza, generalmente, sobre elementos que están ordenados. Los directorios telefónicos están organizados alfabéticamente. las ofertas laborales están ordenadas por tipo de trabajo y los libros de una biblioteca están clasificados por tema. Sin embargo, puede suceder que la búsqueda se realice sobre una colección de elementos no ordenados. Por ejemplo, cuando se busca la localización de una ciudad dentro de un mapa.

Puede concluirse entonces, que la operación de búsqueda se puede llevar a cabo sobre elementos ordenados o desordenados. Cabe destacar que la búsqueda es más fácil y ocupa menos tiempo cuando los datos se encuentran ordenados.

Todo lo mencionado anteriormente es aplicable a la búsqueda de datos en estructuras de información. Los métodos de búsqueda se pueden clasificar en internos y externos, según la ubicación de los datos sobre los cuales se realizará la búsqueda. Se denomina **búsqueda interna** cuando todos los elementos se encuentran en la memoria principal de la computadora; por ejemplo, almacenados en arreglos, listas ligadas o árboles. Y se denomina **búsqueda externa** si los elementos están almacenados en memoria secundaria; es decir, si hubiera archivos en dispositivos como cintas y discos magnéticos.

7.1 Búsqueda interna.

La búsqueda interna trabaja con elementos que se encuentran almacenados en la memoria principal de la máquina. Éstos pueden estar en estructuras estáticas como arreglos, o estructuras dinámicas como listas ligadas y árboles.

Los métodos de búsqueda interna más importantes son:

- Secuencial o lineal.
- Binaria.
- Por transformación de claves.
- Árboles de búsqueda

7.1.1 Secuencial o lineal.

La **búsqueda secuencial** consiste en revisar elemento tras elemento hasta encontrar el dato buscado, o llegar al final del conjunto de datos disponible.

Primero se tratará sobre la búsqueda secuencial en arreglos, y luego en listas enlazadas. En el primer caso, se debe distinguir entre arreglos ordenados y desordenados

La búsqueda secuencial en arreglos desordenados consiste, básicamente, en recorrer el arreglo de izquierda a derecha hasta que se encuentre el elemento buscado o se termine el arreglo, lo que ocurra primero. Normalmente cuando una función de búsqueda concluye con éxito, interesa conocer en qué posición fue hallado el elemento que se estaba buscando. Esta idea se puede generalizar para todos los métodos de búsqueda.

A continuación, se presenta el algoritmo de búsqueda secuencial en arreglos desordenados.

A) <u>Algoritmo Secuencial Desordenado:</u>

```
SecuencialDesordenado (V, N, X)
{El algoritmo busca secuencialmente el elemento X en el arreglo desordenado
llamado V, y N es el número total de elementos que contiene el arreglo}
{I es una variable de tipo entero, BANDERA es una variable de tipo booleano}

1. Hacer I ← 1 y BANDERA ← FALSO
2. Repetir mientras (I ≤ N) y (BANDERA = FALSO)
        2.1 Si (V[I] = X) entonces
                Hacer BANDERA ← VERDADERO
                si no
                Hacer I ← I + 1
        2.2 {Fin de la condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (BANDERA = VERDADERO) entonces
        Escribir "El elemento X está en la posición: I"
        si no
        Escribir "El elemento X no está en el arreglo"
5. {Fin de la condicional del paso 4}
```

Son dos los posibles resultados que se pueden obtener al aplicar este algoritmo: la posición en la que encontró el elemento, o un mensaje de fracaso si el elemento no se halla en el arreglo. Si hubiera dos o más ocurrencias del mismo valor, se

encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias del dato buscado, utilizando una variable auxiliar que sirva de contador.

Este algoritmo también recibe el nombre de **secuencial con bandera** por utilizar la variable auxiliar booleana en la condición de parada del ciclo. El empleo de esta variable evita seguir buscando una vez que el dato ha sido encontrado, con la ventaja de que disminuye el número de comparaciones a realizar, y por lo tanto aumenta la eficiencia del algoritmo.

Ejemplo 7.1.1 a)

Considere que se han almacenado diez números enteros en el arreglo V, y que el dato a buscar es 78.

V: 18 14 23 12 78 56 08 10 21 45

N = 10 y **X** = 78

Las operaciones que se realizarán serán las siguientes:

Hacer I = 1 , BANDERA = FALSO

Repetir mientras (1 ≤ 10) y (BANDERA = FALSO)

A[1] = X	(18 = 78)	FALSO, I = 1+1
A[2] = X	(14 = 78)	FALSO, I = 2+1
A[3] = X	(23 = 78)	FALSO, I = 3+1
A[4] = X	(12 = 78)	FALSO, I = 4+1
A[5] = X	(78 = 78)	CIERTO, BANDERA = VERDADERO, **sale del ciclo**

Si VERDADERO = VERDADERO, entonces

Escribir "El elemento 78 está en la posición: 5"

Ahora suponga que el dato buscado es el 28, en este caso se compara con todos los números guardados en el arreglo y como no es igual a ninguno de ellos, la operación de búsqueda fracasa cuando se llega al último valor.

N = 10 y **X** = 28

Las operaciones que se realizarán serán las siguientes:

Hacer I = 1 , BANDERA = FALSO

Repetir mientras (1 ≤ 10) y (BANDERA = FALSO)

A[1] = X	(18 = 28)	FALSO, I = 1+1
A[2] = X	(14 = 28)	FALSO, I = 2+1
A[3] = X	(23 = 28)	FALSO, I = 3+1
A[4] = X	(12 = 28)	FALSO, I = 4+1
A[5] = X	(78 = 28)	FALSO, I = 5+1
A[6] = X	(56 = 28)	FALSO, I = 6+1
A[7] = X	(08 = 28)	FALSO, I = 7+1
A[8] = X	(10 = 28)	FALSO, I = 8+1
A[9] = X	(21 = 28)	FALSO, I = 9+1
A[10] = X	(45 = 28)	FALSO, I = 10+1, (11 ≤ 10) NO se cumple, **sale del ciclo**

Si no **Escribir "El elemento 28 no está en el arreglo"**

A continuación, se presenta una variante de este algoritmo, pero **utilizando recursividad**, en lugar de iteratividad.

```
SecuencialDesordenadoRecursivo (V, N, X, I)
{Este algoritmo busca secuencialmente, y de forma recursiva, al elemento X
en el arreglo unidimensional desordenado llamado V, el número total de
elementos que contiene el arreglo es N}
{I es un parámetro de tipo entero que inicia su recorrido en la posición 1}

1. Si (I > N) entonces
        Escribir "El elemento X no se encuentra en el arreglo"
        si no
        1.1 Si (V[I] = X) entonces
                Escribir "El elemento X se encuentra en la posición: I"
                si no
                Regresar a SecuencialDesordenadoRecursivo (V, N, X, I + 1)
        1.2 {Fin de la condicional del paso 1.1}
2. {Fin de la condicional del paso 1}
```

La búsqueda secuencial en arreglos ordenados es similar al caso anterior. Sin embargo, el orden entre los elementos del arreglo permite incluir una nueva condición que hace más eficiente al proceso. A continuación, analicemos el algoritmo de **búsqueda secuencial en arreglos ordenados**.

B) Algoritmo Secuencial Ordenado:

```
SecuencialOrdenado (V, N, X)
{El algoritmo busca secuencialmente el elemento X en el arreglo ordenado
llamado V, N es el número total de elementos que contiene el arreglo.
El arreglo V esta ordenado de forma ascendente: V[1] ≤ V[2] ≤ … ≤ V[N]}
{I es una variable de tipo entero, BANDERA es una variable de tipo booleano}

1. Hacer I ← 1 y BANDERA ← FALSO
2. Repetir mientras (I ≤ N) y (BANDERA = FALSO) y (X ≥ V[I])
        2.1 (Si X = V[I]) entonces
                Hacer BANDERA ← VERDADERO
                si no
                 Hacer I ← I + 1
        2.2 {Fin de la condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (BANDERA = VERDADERO) entonces
        Escribir "El elemento X está en la posición: I"
        si no
        Escribir "El elemento X no está en el arreglo"
5. {Fin de la condicional del paso 4}
```

Como el arreglo está ordenado, se establece una nueva condición para controlar la búsqueda: el elemento a encontrar debe ser mayor o igual que el elemento del arreglo con el que se está comparando en ese momento, para poder proseguir con la búsqueda. Cuando el ciclo se interrumpe, se evalúa cuál de las condiciones es falsa. Si (I > N), o si se comparó el elemento con un valor mayor a sí mismo (X < V[I]), entonces se está ante un caso de fracaso: el elemento no está en el arreglo. Por el contrario, si X = V[I] entonces se encontró al elemento en el arreglo, y desplegará el mensaje con la posición en donde se encuentra.

Ejemplo 7.1.1 b)

Considere que se han almacenado diez números enteros en el arreglo V, el cual fue ordenado previamente, y que el dato a buscar es 12.

V: 08 10 12 14 18 21 23 45 56 78

N = 10 y **X** = 12

Las operaciones que se realizarán serán las siguientes:

Hacer I = 1 , BANDERA = FALSO

Repetir mientras (1 ≤ 10) y (BANDERA = FALSO) y (12 ≥ V[I])

X = A[1]	(12 = 08)	FALSO, I = 1+1
X = A[2]	(12 = 10)	FALSO, I = 2+1
X = A[3]	(12 = 12)	CIERTO, BANDERA = VERDADERO,
		sale del ciclo

Si VERDADERO = VERDADERO, entonces

Escribir "El elemento 12 está en la posición: 3"

Ahora suponga nuevamente que el dato buscado es el 28, en este caso se compara con todos los números guardados en el arreglo mientras el número a buscar sea menor o igual, que el almacenado en el arreglo en la posición dada, y la operación de búsqueda fracasa cuando ya no se cumpla la condición.

N = 10 y **X** = 28

Las operaciones que se realizarán serán las siguientes:

Hacer I = 1 , BANDERA = FALSO

Repetir mientras (1 ≤ 10) y (BANDERA = FALSO) y (28 ≥ V[I])

X = A[1]	(28 = 08)	FALSO, I = 1+1
X = A[2]	(28 = 10)	FALSO, I = 2+1
X = A[3]	(28 = 12)	FALSO, I = 3+1
X = A[4]	(28 = 14)	FALSO, I = 4+1
X = A[5]	(28 = 18)	FALSO, I = 5+1
X = A[6]	(28 = 21)	FALSO, I = 6+1
X = A[7]	(28 = 23)	FALSO, I = 7+1, (28 ≥ 45) NO se cumple
		sale del ciclo

Si no **Escribir "El elemento 28 no está en el arreglo"**

A continuación, se presenta el algoritmo de búsqueda secuencial en arreglos ordenados, pero **en forma recursiva**.

```
SecuencialOrdenadoRecursivo (V, N, X, I)
{Este algoritmo busca en forma secuencial y recursiva al elemento X en el
arreglo unidimensional ordenado llamado V, el número total de elementos que
contiene el arreglo es N.
El arreglo V esta ordenado de forma ascendente: V[1] ≤ V[2] ≤ … ≤ V[N]}
{I es un parámetro de tipo entero que inicia su recorrido con el valor de 0}

1. Si (I ≤ N) y (X ≥ V[I]) entonces
        Llamar a SecuencialOrdenadoRecursivo (V, N, X, I + 1)
        si no
        1.1 Si (I > N) o (X < V[I]) entonces
                Escribir "El elemento X no se encuentra en el arreglo"
                si no
                Escribir "El elemento X se encuentra en la posición: I"
        1.2 {Fin de la condicional del paso 1.1}
2. {Fin de la condicional del paso 1}
```

El método de búsqueda secuencial también se puede aplicar a listas ligadas. Consiste en recorrer la lista nodo tras nodo, hasta encontrar al elemento buscado (éxito), o hasta que lleguemos al final de la lista (fracaso). La lista, como en el caso de arreglos, se puede encontrar ordenada o desordenada. El orden en el cual se puede recorrer la lista depende de sus características; puede ser simplemente ligada, doblemente ligada, y además ser circular. En este capítulo se presentará el caso de **búsqueda secuencial en listas simplemente ligadas**, no circulares y desordenadas. El lector, con los conocimientos que tiene sobre listas y búsqueda, podrá implementar fácilmente los otros algoritmos.

C) Algoritmo Secuencial con Listas:

```
SecuencialListaDesordenada (P, X)
{Este algoritmo busca en forma secuencial al elemento X en una lista simplemente ligada desordenada. P es un apuntador al primer nodo de la lista. INFO y LIGA son los campos de cada nodo}
{Q es una variable de tipo apuntador, BANDERA es una variable de tipo booleano}

1. Hacer Q ←P y BANDERA← FALSO
2. Repetir mientras (Q ≠ NULL) y (BANDERA = FALSO)
        2.1 Si (Q^.INFO = X) entonces
                Hacer BANDERA← VERDADERO
                si no
                Hacer Q← Q^.LIGA
        2.2 {Fin de la condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (BANDERA = VERDADERO) entonces
        Escribir "El elemento X si está en la Lista"
        si no
        Escribir "El elemento X no está en la Lista"
5. {Fin de la condicional del paso 4}
```

Si la lista estuviera ordenada, debería modificarse el algoritmo, incluyendo una condición similar a la que se escribió en el algoritmo de secuencial ordenado, esto último con el objetivo de disminuir el número de comparaciones que se realizan.

A continuación, se presenta **la variante recursiva** de este algoritmo de búsqueda secuencial en listas simplemente ligadas desordenadas.

```
SecuencialListaDesordenadaRecursivo (P, X)
{Este algoritmo busca en forma secuencial y recursiva el elemento X en una lista simplemente ligada desordenada. P es un apuntador al primer nodo de la lista. INFO y LIGA son los campos de cada nodo}

1. Si (P ≠ NULL) y (P^.INFO ≠ X) entonces
        Regresar a SecuencialListaDesordenadaRecursivo (P^.LIGA, X)
        si no
        1.1 Si (P = NULL) entonces
                Escribir "El elemento X no está en la Lista"
                si no
                Escribir "El elemento X si está en la Lista"
        1.2 {Fin de la condicional del paso 1.1}
2. {Fin de la condicional del paso 1}
```

7.1.2 Binaria.

La **búsqueda binaria** consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. En el caso de que no fueran iguales se redefinen los extremos del intervalo, según el elemento central sea mayor o menor que el elemento buscado, disminuyendo de esta forma el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado, o cuando el intervalo de búsqueda se anula, es vacío.

El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas, ya que no podríamos retroceder para establecer intervalos de búsqueda, ni con arreglos desordenados.

Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de comparaciones a realizar disminuye notablemente. Esta disminución resulta significativa cuanto más grande sea el tamaño del arreglo.

A continuación, se presenta el **algoritmo de búsqueda binaria**.

```
Binaria (V, N, X)
{El algoritmo busca el elemento X en el arreglo ordenado llamado V, N es el
número total de elementos que contiene el arreglo.}
{IZQ, CEN y DER son variables de tipo entero, BANDERA es una variable de
tipo booleano}

1. Hacer IZQ ← 1, DER ← N y BANDERA ← FALSO
2. Repetir mientras (IZQ ≤ DER) y (BANDERA = FALSO)
        Hacer CEN ← PARTE ENTERA ((IZQ + DER) / 2)
        2.1 Si (X = V[CEN]) entonces
                Hacer BANDERA ← VERDADERO
                si no {Redefinir el intervalo de búsqueda}
                2.1.1 Si (X > V[CEN]) entonces
                 Hacer IZQ ← CEN + 1
                si no
                Hacer DER ← CEN – 1
                2.1.2 {Fin de la condicional del paso 2.1.1}
        2.2 {Fin de la condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (BANDERA = VERDADERO) entonces
        Escribir "El elemento X está en la posición: CEN"
        si no
        Escribir "El elemento X no está en el arreglo"
5. {Fin de la condicional del paso 4}
```

Analicemos ahora un ejemplo para ilustrar el funcionamiento de este algoritmo.

Ejemplo 7.1.2

Sea V un arreglo unidimensional de números enteros, ordenado de manera ascendente, como se muestra en la siguiente figura 9.3.

101	215	325	410	502	507	600	610	612	670

En la siguiente tabla se presenta el seguimiento del algoritmo de búsqueda binaria para (**X = 325**). El valor **SI** se encuentra en el arreglo.

PASO	BANDERA	IZQ	DER	CEN	X=V[CEN]	X>V[CEN]
1	FALSO	1	10	5	325=502? NO	325>502? NO
2	FALSO	1	4	2	325=215? NO	325>215? SI
3	FALSO	3	4	3	325=325? SI	
4	VERDADERO					

Como ya no se cumple la condición **(BANDERA = FALSO)** entonces, sale del ciclo y despliega el mensaje:

Escribir "El elemento 325 está en la posición: 3"

La siguiente tabla, por otra parte, se muestra nuevamente el seguimiento del algoritmo de búsqueda binaria, ahora para (**X = 615**). Valor que **NO** se encuentra en el arreglo.

PASO	BANDERA	IZQ	DER	CEN	X=V[CEN]	X>V[CEN]
1	FALSO	1	10	5	615=502? NO	615>502? SI
2	FALSO	6	10	8	615=610? NO	615>610? SI
3	FALSO	9	10	9	615=612? NO	615>612? SI
4	FALSO	10	10	10	615=670? NO	615>670? SNO
5	FALSO	10	9			

Como ya no se cumple la condición **(IZQ ≤ DER)** porque **(10 ≤ 9)** entonces, sale del ciclo y despliega el mensaje:

Escribir "El elemento 615 no está en el arreglo"

A continuación, se presenta el algoritmo de **búsqueda binaria recursivo**.

```
BinariaRecursivo (V, IZQ, DER, X)
{Este algoritmo busca al elemento X en el arreglo unidimensional ordenado
llamado V, de N componentes en total, IZQ ingresa inicialmente al algoritmo
con el valor de 1, DER por otra parte ingresa con el valor de N}
{CEN es una variable de tipo entero}

1. Si (IZQ > DER) entonces
        Escribir "El elemento X no se encuentra en el arreglo"
        si no
        Hacer CEN ← PARTE ENTERA ((IZQ + DER) / 2)
        1.1 Si (X = V[CEN]) entonces
                Escribir "El elemento X se encuentra en la posición: CEN"
                si no {Redefinir el intervalo de búsqueda}
                1.1.1 Si (X > V[CEN]) entonces
                Regresar a BinariaRecursivo (V, CEN + 1, DER, X)
                si no
                Regresar a BinariaRecursivo (V, IZQ, CEN – 1, X)
                1.1.2 {Fin de la condicional del paso 1.1.1}
        1.2 {Fin de la condicional del paso 1.1}
2. {Fin de la condicional del paso 1}
```

7.1.3 Por transformación de claves.

Los dos métodos analizados anteriormente permiten encontrar un elemento en un arreglo. En ambos casos el tiempo de búsqueda es proporcional a su número de componentes. Es decir, a mayor número de elementos se debe realizar mayor número de comparaciones. Se mencionó además que, si bien el método de búsqueda binaria es más eficiente que el secuencial, existe la restricción de que el arreglo se debe encontrar ordenado.

Esta sección se dedica a un nuevo método de búsqueda. Este método, conocido como transformación de claves o hash, permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta con la ventaja de que el tiempo de búsqueda es independiente del número de componentes del arreglo.

Supongamos que se tiene una colección de datos, cada uno de ellos identificado por una clave. Es claro que resulta atractivo tener acceso a ellos de manera directa; es decir, sin recorrer algunos datos antes de localizar al buscado. El método por transformación de claves permite realizar justamente esta actividad; es decir, localizar el dato en forma directa. El método trabaja utilizando una función que convierte una clave dada en una dirección —**índice**— dentro del arreglo.

dirección ← H (clave)

La función hash aplicada a la clave genera un índice del arreglo que permite acceder **directamente** al elemento. El caso más trivial se presenta cuando las claves con números enteros consecutivos.

Supongamos que se desea almacenar la información relacionada con 100 alumnos cuyas matrículas son números del 1 al 100. En este caso conviene definir un arreglo de 100 elementos con índices numéricos comprendidos entre los valores l y 100. Los datos de cada alumno ocuparán la posición del arreglo que se corresponda con el número de la matrícula; de esta manera se podrá acceder directamente a la información de cada alumno.

Consideremos ahora que se desea almacenar la información de 100 empleados. La clave de cada empleado corresponde al número de su seguro social. Si la clave está formada por 11 dígitos, resulta por completo ineficiente definir un arreglo con 99 999 999 999 elementos para almacenar solamente los datos de los 100 empleados. Utilizar un arreglo tan grande asegura la posibilidad de acceder directamente a sus elementos; sin embargo, el costo en memoria resulta tanto ridículo como excesivo. Siempre es importante equilibrar el costo del espacio de memoria con el costo por tiempo de búsqueda.

Cuando se tienen claves que no se corresponden con índices (por ejemplo, por ser alfanuméricas), o cuando las claves representen valores numéricos muy grandes o no se corresponden con los índices de los arreglos, se utilizará una función hash que permita transformar la clave para obtener una dirección apropiada. Esta función hash debe ser simple de calcular y asignar direcciones de la manera más uniforme posible. Es decir, debe generar posiciones diferentes dadas dos claves también diferentes. Si esto último no ocurre ($H(K_1) = d$, $H(K_2) = d$ y $K_1 \neq K_2$), hay una **colisión**, que se define como la asignación de una misma dirección a dos o más claves distintas.

En este contexto, para trabajar con este método de búsqueda se debe seleccionar previamente:

- Una función hash que sea fácil de calcular y distribuya uniformemente las claves.
- Un método para resolver colisiones. Si éstas se presentan, se contará con algún método que genere posiciones alternativas.

Estos dos casos se tratarán en forma separada.

Como ya se mencionó, seleccionar una buena función hash es muy importante, pero es difícil encontrarla. Básicamente porque no existen reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en su distribución. Realizar un análisis de las principales características de las claves siempre ayuda en la elección de una función de este tipo.

A continuación, se explican algunas de las funciones hash más utilizadas.

1) Función modulo (por división).

La función hash por módulo o división consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Supongamos, que se tiene un arreglo de N elementos, y K es la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:

$$H(k) = (K \bmod N) + 1$$

En la fórmula se observa que al residuo de la división se le suma 1, esto último con el objetivo de obtener un valor comprendido entre 1 y N.

Para lograr mayor uniformidad en la distribución, es importante que N sea un numero primo o divisible entre muy pocos números. Por lo tanto, si N no es un número primo, se debe considerar el valor primo más cercano al valor de N.

En el siguiente ejemplo se presenta un caso de función hash por módulo.

Ejemplo 7.1.3 a)

Supongamos que N = 100 es el tamaño del arreglo, y las direcciones que se desea asignar a los elementos (al guardarlos o recuperarlos) son los números del 1 al 100. Consideremos además que K_1 = 7259 y K_2 = 9359 son dos claves a las que se deben asignar posiciones en el arreglo. Si aplicamos la fórmula anterior con N =100, para calcular las direcciones correspondientes a K_1, y K_2, obtenemos:

$H(K_1)$ = (7259 mod 100) + 1 = 60

$H(K_2)$ = (9359 mod 100) + 1 = 60

Como $H(K_1)$ es igual a $H(K_2)$ y K_1 es distinto de K_2, se está ante una **colisión** que se debe resolver porque a los dos elementos le correspondería la misma dirección.

Observemos, sin embargo, que, si aplicáramos la fórmula con el número primo más cercano a N, el resultado cambiaría:

$H(K_1)$ = (7259 mod 97) + 1 = 82

$H(K_2)$ = (9359 mod 97) + 1 = 48

Con **N = 97** se ha eliminado la colisión.

2) Función cuadrado.

La función hash cuadrado consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos que se debe considerar se encuentra determinado por el rango del índice. Sea K la clave del dato a buscar, la función hash cuadrado queda definida, entonces, por la siguiente fórmula:

H(k) = dígitos centrales (K^2) + 1

La suma de una unidad a los dígitos centrales es útil para obtener un valor comprendido entre 1 y N.

En el ejemplo siguiente se presenta un caso de función hash cuadrado.

Ejemplo 7.1.3 b)

Sea N = 100 el tamaño del arreglo, y sus direcciones los números comprendidos entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula anterior para calcular las direcciones correspondientes para K_1 y K_2:

K_1^2 = 52 693 081

K_2^2 = 87 590 881

$H(K_1)$ = dígitos centrales (526 93 081) + 1 = 94

$H(K_2)$ = dígitos centrales (875 90 881) + 1 = 91

Como el rango de índices en nuestro ejemplo varía de 1 a 100, se toman solamente los dos dígitos centrales del cuadrado de las claves.

3) Función plegamiento.

La función hash por plegamiento consiste en dividir la clave en partes, tomando igual número de dígitos, aunque la última puede tener menos, y operar con ellas, asignando como dirección los dígitos menos significativos. La operación entre las partes se puede realizar por medio de sumas o multiplicaciones. Sea K la clave del dato a buscar. K está formada por los dígitos d_1, d_2, ..., d_n. La función hash por plegamiento queda definida por la siguiente fórmula:

$$H(k) = \text{dígmensig}((d_1 \ldots d_i) + (d_i + 1 \ldots d_j) + \ldots + (d_1 \ldots d_n)) + 1$$

El operador que aparece en la fórmula operando las partes de la clave es el de suma, pero, como ya se aclaró, puede ser el de la multiplicación. En este contexto, la suma de una unidad a los dígitos menos significativos —**dígmensig**— es para obtener un valor comprendido entre 1 y N.

En el ejemplo siguiente se presenta un caso de función hash por plegamiento.

Ejemplo 7.1.3 c)

Sea N = 100 el tamaño del arreglo, y las direcciones que deben tomar sus elementos los números comprendidos entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula anterior para calcular las direcciones correspondientes para K_1 y K_2.

$H(K_1)$ = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32

$H(K_2)$ = dígmensig (93 + 59) + 1 = dígmensig (152) + 1 = 53

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

4) Función truncamiento.

La función hash por truncamiento consiste en tomar algunos dígitos de la clave y formar con ellos una dirección. Este método es de los más sencillos, pero es también de los que ofrecen menos uniformidad en la distribución de las claves.

Sea K la clave del dato a buscar. K está formada por los dígitos d_1, d_2, ..., d_n. La función hash por truncamiento se representa con la siguiente fórmula:

H(K) = elegirdígitos (d_1, d_2 ... d_n) + 1

La elección de los dígitos es arbitraria. Se podrían tomar los de las posiciones impares o de las pares. Luego se podrían unir de izquierda a derecha o de derecha a izquierda. La suma de una unidad a los dígitos seleccionados es útil para obtener un valor entre 1 y 100.

En el ejemplo siguiente se muestra un caso de función hash por truncamiento.

Ejemplo 7.1.3 d)

Sea N = 100 el tamaño del arreglo, y las direcciones de sus elementos los números entre y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula anterior para calcular las direcciones correspondientes para K_1 y K_2.

H(K_1) = elegirdígitos (7259) + 1 = 75 + 1 = 76

H(K_2) = elegirdígitos (9359) + 1 = 95 + 1 = 96

En este ejemplo se toman el primero y tercer números de la clave y se unen de izquierda a derecha.

Es importante destacar que en todos los casos anteriores se presentaron ejemplos de claves numéricas. Sin embargo, en la práctica las claves pueden ser alfabéticas o alfanuméricas. En general, cuando aparecen letras en las claves se suele asociar a cada una un entero con el propósito de convertirlas en numéricas.

A	B	C	D	. Z
01	02	03	04	... 27

Si, por ejemplo, la clave fuera **ADA**, su equivalente numérico sería **010401**. Si hubiera combinación de letras y números, se procedería de la misma manera. Por ejemplo. dada una clave **Z4F21**, su equivalente numérico sería **2740621**. Otra alternativa sería tomar el valor decimal asociado para cada carácter según el código ASCII. Una vez obtenida la clave en su forma numérica, se puede utilizar normalmente cualesquiera de las funciones antes mencionadas.

Solución de colisiones.

La elección de un método adecuado para resolver colisiones es tan importante como la elección de una buena función hash. Cuando ésta obtiene una misma dirección para dos claves diferentes, se está ante una colisión. Normalmente, cualquiera que sea el método elegido resulta costoso tratar las colisiones. Es por ello que se debe hacer un esfuerzo importante para encontrar la función que ofrezca la mayor uniformidad en la distribución de las claves.

La manera más natural de resolver el problema de las colisiones es reservar una casilla por clave; es decir, aquellas que se correspondan una a una con las posiciones del arreglo. Pero, como ya se mencionó, esta solución puede tener un alto costo en memoria; por lo tanto, se deben analizar otras alternativas que permitan equilibrar el uso de memoria con el tiempo de búsqueda.

En adelante se estudiarán algunos de los métodos más utilizados para resolver colisiones, que se pueden clasificar en:

- **Reasignación.**
- **Arreglos anidados.**
- **Encadenamiento.**

A) Reasignación.

Existen varios métodos que trabajan bajo el principio de comparación y reasignación de elementos. A continuación, se analizarán tres de ellos:

- **Prueba lineal.**
- **Prueba cuadrática.**
- **Doble dirección hash.**

1. Prueba lineal.

El método de prueba lineal consiste en que una vez que se detecta la colisión, se recorre el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o cuando se encuentra una posición vacía. El arreglo se trata como una estructura circular: el siguiente elemento después del último es el primero.

A continuación, se expone el algoritmo de solución de colisiones por medio de la prueba lineal.

Algoritmo PruebaLineal:

```
PruebaLineal (V, N, K)
{El algoritmo busca el dato K en el arreglo llamado V de N elementos. Resuelve
el problema de las colisiones por medio de la prueba lineal.}
{D y DX son variables de tipo entero.}

1. Hacer D← H(K) {Generar dirección}
2. Si V[D] = K entonces
        Escribir "El elemento K está en la posición: D"
        si no
        Hacer DX← D + 1
        2.1 Repetir mientras (DX ≤ N) y (V[DX] ≠ K) y (V[DX] ≠ VACIO) y (DX≠D)
        Hacer DX← DX + 1
                2.1.1 Si (DX = N + 1) entonces
                        Hacer DX← 1
                2.1.2 {Fin de la condicional del paso 2.1.1}
        2.2 {Fin del ciclo del paso 2.1}
        2.3 Si (V[DX] = K) entonces
                Escribir "El elemento K está en la posición: DX"
                si no
                Escribir "El elemento K no está en el arreglo"
        2.4 {Fin de la condicional del paso 2.3}
3. {Fin de la condicional del paso 2}
```

La cuarta condición del ciclo del punto 2.1, **(DX ≠ D)**, es para evitar caer en un ciclo infinito, ya que regresaría a su punto de partida, si el arreglo estuviera lleno y el elemento a buscar no se encontrará en él.

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo podrían permanecer vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial, perdiendo así las ventajas del método hash.

Ejemplo 7.1.3 e)

Sea V un arreglo unidimensional de 10 elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 fueron asignadas según la función hash:

H(K) = (K mod 10) + 1

En la siguiente figura se aprecia el estado de arreglo (a) y la tabla con H(K) para cada clave (b). Solución de colisiones por la prueba lineal.

a)

1) 80
2)
3)
4) 43
5) 54
6) 25
7) 56
8) 35
9) 13
10) 104

Arreglo V

b)

K	*H(K)*
25	6
43	4
56	7
35	6->8
54	5
13	4->9
80	1
104	5->10

Tabla H(K)

En la siguiente tabla se presenta el seguimiento de las variables importantes del algoritmo para el caso del ejemplo anterior. Con la solución de colisiones por la prueba lineal para **K = 35**.

D	DX
6	7
	8

Al aplicar la función hash a la clave 35, se obtiene una dirección (D = 6). Sin embargo, en esa posición no se encuentra el elemento buscado, por lo que se comienza a recorrer secuencialmente el arreglo a partir de la posición (DX = 7).

La búsqueda concluye al encontrar el valor deseado en la posición 8.

En la siguiente tabla se presenta ahora el seguimiento de las variables importantes del algoritmo, pero ahora para un caso más complejo del ejemplo anterior. Con la solución de colisiones por la prueba lineal para **K = 13**.

D	DX
4	5
	6
	7
	8
	9

2. Prueba cuadrática.

El método de la prueba cuadrática es similar al anterior. La diferencia consiste en que aquí las direcciones alternativas se generarán como D + 1, D + 4, D + 9, ..., D + i^2 en vez de D + 1, D + 2, ..., D + i. Esta variación permite una mejor distribución de las claves que colisionan.

A continuación, se presenta el algoritmo de solución de colisiones por medio de la prueba cuadrática.

Algoritmo PruebaCuadrática:

```
PruebaCuadrática (V, N, K)
{El algoritmo busca el dato K en el arreglo llamado V de N elementos. Resuelve
el problema de las colisiones por medio de la prueba cuadrática.}
{D, DX e I son variables de tipo entero.}

1. Hacer D← H(K) {Generar dirección}
2. Si V[D] = K entonces
        Escribir "El elemento K está en la posición: D"
        si no
        Hacer I← 1 y DX← D + I²
        2.1 Repetir mientras (V[DX] ≠ K) y (V[DX] ≠ VACIO)
        Hacer I← I + 1 y DX← DX + I²
                2.1.1 Si (DX > N) entonces
                        Hacer I← 0, DX ← 1 y D ← 1
                2.1.2 {Fin de la condicional del paso 2.1.1}
        2.2 {Fin del ciclo del paso 2.1},
        2.3 Si (V[DX] = K) entonces
                Escribir "El elemento K está en la posición: DX"
                si no
                Escribir "El elemento K no está en el arreglo"
        2.4 {Fin de la condicional del paso 2.3}
3. {Fin de la condicional del paso 2}
```

La principal desventaja de este método es que pueden quedar casillas del arreglo sin visitar. Además, como los valores de las direcciones varían en I^2 unidades, resulta difícil determinar una condición general para detener el ciclo del punto 2.1.

Este problema podría solucionarse empleando una variable auxiliar, cuyos valores dirijan el recorrido del arreglo de tal manera que garantice que serán visitadas todas las casillas del arreglo.

A continuación, se presenta un ejemplo que ilustra el funcionamiento del algoritmo de la **prueba cuadrática**.

Ejemplo 7.1.3 f)

Sea V un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80, 104 y 55 se asignaron según la función hash:

H(K) = (K mod 10) + 1

En la siguiente figura se presenta el estado de arreglo (a) y la tabla con H(K) para cada clave (b). Solución de colisiones por la prueba cuadrática.

a)

1) 80
2) 55
3)
4) 43
5) 54
6) 25
7) 56
8) 13
9) 104
10) 35

Arreglo V

b)

K	*H(K)*
25	6
43	4
56	7
35	6->10
54	5
13	4->8
80	1
104	5->9
55	6

Tabla H(K)

En la siguiente tabla se presenta el seguimiento de las variables importantes del algoritmo para el caso del ejemplo anterior. Con la solución de colisiones por la prueba cuadrática para **K = 35**.

D	I	DX
6	1	7
	2	10

Al aplicar la función hash a la clave 35, se obtiene una dirección (D = 6). Sin embargo, en esa posición no se encuentra el elemento buscado. Se calcula posteriormente DX, como la suma de D + I^2, obteniéndose la dirección (DX = 7).

El algoritmo de búsqueda concluye al encontrar el valor deseado en la posición 10 del arreglo.

En la siguiente tabla se presenta ahora el seguimiento de las variables importantes del algoritmo, pero ahora para un caso más complejo del ejemplo anterior. Con la solución de colisiones por la prueba cuadrática para **K = 55**.

D	I	DX
6	1	7
	2	10
	3	15
1	0	1
	1	2

3. Doble dirección hash.

El método de doble dirección hash consiste en que una vez que se detecta la colisión, se genera otra dirección aplicando la misma función hash a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o cuando se encuentra una posición vacía.

D H(K)

D' H(D)

D" H(D')

La función hash que se aplica a las direcciones puede o no ser la misma que originalmente se aplicó a la clave; podría ser cualquier otra. No existe ningún estudio que precise cuál es la mejor función que se debe utilizar en el cálculo de las direcciones sucesivas.

Analicemos ahora el algoritmo de solución de colisiones por medio del método de la doble dirección hash.

Algoritmo DobleDirección:

```
DobleDirección (V, N, K)
{El algoritmo busca el dato K en el arreglo llamado V de N elementos. Resuelve
el problema de las colisiones por medio de la doble dirección hash.}
{D y DX son variables de tipo entero.}

1. Hacer D← H(K) {Generar dirección}
2. Si V[D] = K entonces
        Escribir "El elemento K está en la posición: D"
        si no
        Hacer DX← H'(D)
        2.1 Repetir mientras (DX ≤ N) y (V[DX] ≠ K) y (V[DX] ≠ VACIO) y (DX≠D)
        Hacer DX← H'(DX)
        2.2 {Fin del ciclo del paso 2.1}
        2.3 Si (V[DX] = K) entonces
                Escribir "El elemento K está en la posición: DX"
                si no
                Escribir "El elemento K no está en el arreglo"
```

2.4 {Fin de la condicional del paso 2.3}
3. {Fin de la condicional del paso 2}

A continuación, se presenta un ejemplo que ilustra el funcionamiento del algoritmo de la **doble dirección hash**.

Ejemplo 7.1.3 g)

Sea V un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 se asignaron según la función hash:

$$H(K) = (K \bmod 10) + 1$$

Además, se definió una función H' para calcular direcciones alternativas en caso de haber colisión:

$$H'(D) = ((D + 1) \bmod 10) + 1$$

En la siguiente figura se presenta el estado de arreglo (a) y la tabla con H(K) para cada clave (b), y H'(D). Solución de colisiones por la doble dirección hash.

a)

1) 80
2)
3)
4) 43
5) 54
6) 25
7) 56
8) 35
9) 104
10) 13

Arreglo V

b)

K	H(K)	H'(D)	H'(D')	H'(D'')
25	6			
43	4			
56	7			
35	6	8		
54	5			
13	4	6	8	10
80	1			
104	5	7	9	

Tabla H(K)

En la siguiente tabla se presenta el seguimiento de las variables importantes del algoritmo para el caso del ejemplo anterior. Con la solución de colisiones por la prueba doble dirección hash para **K = 13**.

D	DX
4	6
	8
	10

Al aplicar la función hash (H) a la clave 13, se obtiene una dirección (D = 4). Sin embargo, en esa posición no se encuentra el elemento buscado. Posteriormente se calcula H' reiteradamente, generando direcciones hasta localizar el valor deseado. En este ejemplo fue preciso aplicar tres veces la función H' obteniéndose las direcciones 6, 8 y 10, en la que finalmente se localiza el dato buscado.

B) Arreglos anidados.

El método de arreglos anidados consiste en que cada elemento del arreglo tenga otro arreglo, en el cual se almacenen los elementos que colisionan. Si bien la solución parece ser sencilla, es claro que resulta ineficiente. Al trabajar con arreglos se depende del espacio que se haya asignado a éstos. Lo cual conduce a un nuevo problema difícil de solucionar: elegir un tamaño adecuado de arreglo que permita un equilibrio entre el costo de memoria y el número de valores colisionados que pudiera almacenar.

Analicemos un ejemplo de arreglos anidados.

Ejemplo 7.1.3 h)

Sea V un arreglo unidimensional de diez elementos. Los elementos con claves: 25, 43, 56, 35, 54, 13, 80 y 104 fueron almacenados utilizando la función hash:

H(K) = (K mod 10) + 1

En la siguiente figura se presenta el estado del arreglo anidado (a) y la tabla con H(K) para cada clave (b). Solución de colisiones con arreglos anidados.

a)

1) 80		
2)		
3)		
4) 43	13	
5) 54	10 4	
6) 25	35	
7) 56		
8)		
9)		
10)		

Arreglo V anidado

b)

K	*H(K)*
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

Tabla con H(K)

C) Encadenamiento.

El método de encadenamiento consiste en que cada elemento del arreglo tenga un apuntador a una lista ligada, la cual se irá generando y almacenará los valores que colisionan. Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones que se presenten, se podrán resolver sin inconvenientes.

Como desventajas del método de encadenamiento se menciona el hecho de que ocupa espacio adicional al de la tabla y que exige el manejo de listas ligadas. Además, si las listas crecen demasiado se perderá la facilidad de acceso directo del método hash.

A continuación, se presenta el algoritmo de solución de colisiones por encadenamiento.

Algoritmo Encadenamiento:

```
Encadenamiento (V, N, K)
{El algoritmo busca el dato con clave K en el arreglo unidimensional llamado
V de N elementos. Resuelve el problema de las colisiones por medio de
encadenamiento en listas ligadas. SIG e INFO son los campos de cada nodo
de la lista.}
{D es una variable de tipo entero. Q es una variable tipo puntero.}

1. Hacer D← H(K) {Generar dirección}
2. Si V[D] = K entonces
        Escribir "El elemento K está en la posición: D"
        si no
        Hacer Q ← V[D].SIG {Apuntador a la lista}
        2.1 Repetir mientras (Q ≠ VACIO) y (Q^.INFO ≠ K)
        Hacer Q ← Q^.SIG
        2.2 {Fin del ciclo del paso 2.1}
        2.3 Si (Q = VACIO) entonces
                Escribir "El elemento K NO está en la lista"
                si no
                Escribir "El elemento K SI está en la lista"
        2.4 {Fin de la condicional del paso 2.3}
3. {Fin de la condicional del paso 2}
```

El funcionamiento de este algoritmo queda más claro con el siguiente ejemplo.

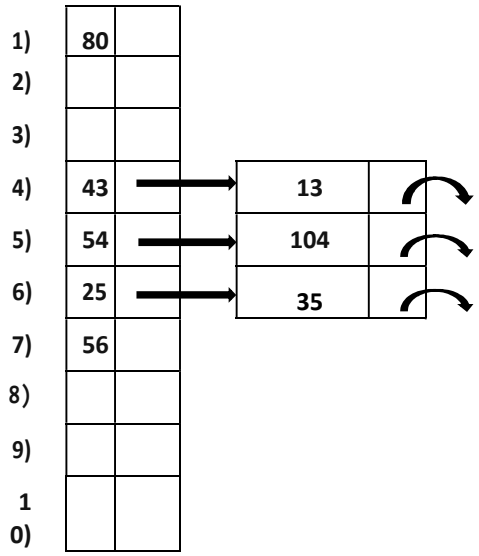
Ejemplo 7.1.3 i)

Sea V un arreglo unidimensional de diez elementos. Los elementos con claves 25. 43, 56, 35, 54, 13, 80 y 104 fueron almacenados en el arreglo unidimensional V utilizando la siguiente función hash:

H(K) = (K mod 10) + 1

En la siguiente figura se presenta el estado del arreglo V (a) y la tabla con H(K) para cada clave (b). Solución de colisiones por encadenamiento.

a)



b)

K	H(K)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

Arreglo V **Lista ligada** **Tabla con H(K)**

Una vez detectada la colisión en una cierta posición del arreglo, deberá recorrerse la lista asociada a ella hasta encontrar el elemento buscado o hasta llegar al final de la lista.

7.1.4 Árboles de búsqueda.

En ciencias de la computación, un árbol de búsqueda es una estructura de datos de tipo árbol utilizado para localizar llaves concretas dentro de un conjunto. Para que un árbol pueda funcionar como árbol de búsqueda en cada nodo tiene que cumplirse que su llave tiene que ser más grande que cualquier llave contenida en su subárbol izquierdo y menor que cualquier llave contenida en su subárbol derecho.

La ventaja de los árboles de búsqueda es su eficiencia en el tiempo de búsqueda, dado que el árbol está razonablemente balanceado, lo que quiere decir que todas las hojas se encuentran a profundidades similares. Existen varias estructuras de datos de tipo árbol de búsqueda, varias de las cuales también permiten la inserción y eliminación eficiente de elementos, operaciones que tienen que mantener el equilibrio del árbol.

Los árboles de búsqueda a menudo son utilizados para implementar vectores asociativos. El algoritmo del árbol de búsqueda utiliza la llave del par llave-valor para encontrar una ubicación, y entonces la aplicación almacena el par entero en esa ubicación.

Árbol Binario de Búsqueda

Un árbol binario de búsqueda es una estructura de datos basada en nodos donde cada nodo contiene una llave y dos subárboles, el izquierdo y el derecho. Para todos los nodos, las llaves de los nodos pertenecientes a su subárbol izquierdo deben ser menores que la llave del nodo, y las llaves de los nodos pertenecientes a su subárbol derecho deben ser mayores que la llave del nodo. Estos subárboles deben calificar también como árboles de búsqueda binarios.

La complejidad temporal del algoritmo de búsqueda en un árbol binario de búsqueda es la altura del propio árbol, la cual es menor que ***O(log n)*** para un árbol que contiene n elementos.

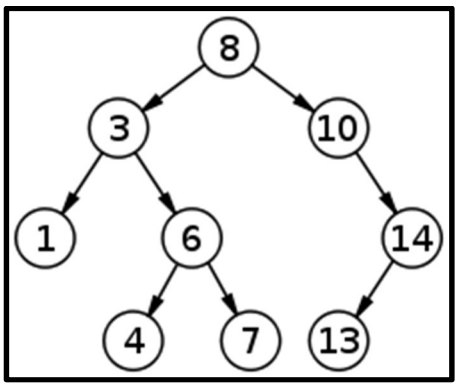


Figura 7.1.4 Árbol binario de búsqueda de tamaño 9, de profundidad 3, con raíz 8 y hojas 1, 4, 7 y 13.

El árbol binario de búsqueda es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación. Comparando esta estructura con otras, pueden observarse ciertas ventajas. Por ejemplo, en un arreglo es posible localizar datos eficientemente si los mismos se encuentran ordenados, pero las operaciones de inserción y eliminación resultan costosas. En las listas, las operaciones de inserción y eliminación se pueden llevar a cabo con facilidad, sin embargo, la búsqueda es una operación bastante costosa que incluso nos puede llevar a recorrer todos los elementos de ella para localizar uno en particular.

Algoritmo de Búsqueda:

```
BÚSQUEDA (NODO, INFOR)
{Este algoritmo localiza un nodo en un árbol binario de búsqueda. NODO es
una variable de tipo puntero que apunta a la raíz del árbol. INFOR es una
variable de tipo entero que contiene la información que se desea localizar en
el árbol. Cabe aclarar que la primera vez la variable NODO no puede ser vacía.}

1. Si (INFOR < NODO^INFO) entonces
        1.1 Si (NODO^IZQ = NIL) entonces
                Escribir "El Nodo no se encuentra en el árbol"
                si no
                Regresar a BÚSQUEDA con NODO^IZQ e INFOR
                {Llamada recursiva}
        1.2 {Fin de la condicional del paso 1.1}
        si no
        1.3 Si (INFOR > NODO^INFO) entonces
                1.3.1 Si (NODO^DER = NIL) entonces
                        Escribir "El Nodo no se encuentra en el árbol."
                        si no
                        Regresar a BÚSQUEDA con NODO^DER e INFOR
                        {Llamada recursiva}
                1.3.2 {Fin de la condicional del paso 1.3.1}
                si no
                Escribir "El Nodo si se encuentra en el árbol."
        1.4 {Fin de la condicional del paso 1.3}
2. {Fin de la condicional del paso 1}
```

Ejemplo 7.1.4

Supóngase que se desea **localizar las claves 4 y 11** en el árbol binario de búsqueda de la figura 7.1.4. En las siguientes tablas se presentan los pasos (P), número de comparaciones (C) y las preguntas con las acciones necesarias para localizar las claves.

Localización de la clave 4 (INFOR ← 4)

P	C	Preguntas y Acciones
1	1	¿Es 4 < 8?
	2	Si. ¿Es el subárbol izquierdo de 8(3) = NIL? No. Entonces regresamos a BÚSQUEDA con el subárbol izq. de 8(3) e INFOR
2	3	¿Es 4 < 3?
	4	No. ¿Es 4 > 3?
	5	Si. ¿Es el subárbol derecho de 3(6) = NIL? No. Entonces regresamos a BÚSQUEDA con el subárbol der. de 3(6) e INFOR
3	6	¿Es 4 < 6?
	7	Si. ¿Es el subárbol izquierdo de 6(4) = NIL? No. Entonces regresamos a BÚSQUEDA con el subárbol izq. de 6(4) e INFOR
4	8	¿Es 4 < 4?
	9	No. ¿Es 4 > 4? No. Entonces escribir "El Nodo si se encuentra en el árbol." ÉXITO

Localización de la clave 11 (INFOR ← 11)

P	C	Preguntas y Acciones
1	1	¿Es 11 < 8?
	2	No. ¿Es 11 > 10?
	3	Si. ¿Es el subárbol derecho de 10(14) = NIL? No. Entonces regresamos a BÚSQUEDA con el subárbol der. de 10(14) e INFOR
2	4	¿Es 11 < 14?
	5	Si. ¿Es el subárbol izquierdo de 14(13) = NIL? No. Entonces regresamos a BÚSQUEDA con el subárbol izq. de 14(13) e INFOR
3	6	¿Es 11 < 13?
	7	Si. ¿Es el subárbol izquierdo de 13(NIL) = NIL? Si. Entonces escribir "El Nodo no se encuentra en el árbol." FRACASO

7.2 Búsqueda externa.

En la sección anterior se estudiaron las técnicas de búsqueda que son aplicables cuando la información reside en la memoria principal de la computadora. En particular, se analizó la operación de búsqueda en estructuras estáticas (arreglos) y dinámicas (listas y árboles) de información. Sin embargo, existen casos en los cuales no se puede manejar toda la información en memoria principal, sino que es necesario trabajar con información almacenada en archivos. Este tipo de búsqueda se denomina **búsqueda externa**.

Los archivos se usan normalmente cuando el volumen de datos es significativo. o cuando la aplicación exige la permanencia de los datos, aun después de que ésta se termine de ejecutar. Como los archivos se encuentran almacenados en dispositivos periféricos (cintas, discos, etc.), las operaciones de escritura y lectura de datos tienen un alto costo en cuanto a tiempo, por los accesos a estos periféricos.

Para disminuir el tiempo de acceso es muy importante optimizar las operaciones de búsqueda, inserción y eliminación en archivos. Una forma de hacerlo es trabajar con archivos ordenados.

A continuación, se describen y analizan algunos de los métodos más utilizados en búsqueda externa.

7.2.1 Secuencial o lineal.

Los archivos secuenciales son aquellos cuyos componentes o registros ocupan posiciones relativas consecutivas. Todo componente o registro de un archivo tiene generalmente un campo que lo identifica, llamado campo clave. Éste se encuentra formado por un conjunto de caracteres o dígitos. Además, ocupa la misma posición relativa en todos los registros de un mismo archivo. Algunos ejemplos de campos clave son el número de cliente (archivo de clientes), el número de contribuyente (archivo de hacienda), la matrícula de un alumno (archivo de alumnos), el número de empleado (archivo de empleados), etc. Puede suceder que la clave de un registro esté formada por más de un campo del mismo. Por ejemplo, en un sistema de inventarios cada pieza se podría identificar por un campo que haga referencia al departamento al cual pertenece, y otro campo para la pieza en sí.

Enseguida se describen algunos métodos de búsqueda en archivos secuenciales.

Búsqueda secuencial

El método de búsqueda secuencial consiste en recorrer el archivo comparando la clave buscada con la clave del registro en curso. El recorrido lineal del archivo termina cuando se encuentra el elemento, o bien, cuando se alcanza el final del archivo. Se pueden presentar algunas variantes dentro de este método, dependiendo sobre todo de si el archivo esta ordenado o desordenado.

A continuación, se detalla el algoritmo de búsqueda lineal en un archivo secuencial desordenado.

A) Algoritmo Archivo Secuencial Desordenado:

```
ArchivoSecuencialDesordenado (FA, K)
{Este algoritmo busca secuencialmente en el archivo FA, un registro con clave K}
{BANDERA es una variable de tipo booleano, R es una variable de tipo registro}

1. Abrir el archivo FA para lectura
2. Hacer BANDERA◄— FALSO
3. Repetir mientras (no sea el fin de archivo de FA) y (BANDERA = FALSO)
        Leer R de FA
        3.1 Si (R.CLAVE = K) entonces
                Escribir "El elemento K está en el archivo."
                Hacer BANDERA◄— VERDADERO
        3.2 {Fin de la condicional del paso 3.1}
```

```
4. {Fin del ciclo del paso 3}
5. Si (BANDERA = FALSO) entonces
        Escribir "El elemento K no está en el archivo."
6. {Fin de la condicional del paso 5}
```

Como podrá observarse, este algoritmo es similar al 7.1.1. En general, tiene las mismas características que el método secuencial en arreglos desordenados.

El algoritmo de búsqueda en archivos ordenados se estudiará considerando, en particular, archivos ordenados ascendentemente.

B) Algoritmo Archivo Secuencial Ordenado:

```
ArchivoSecuencialOrdenado (FA, K)
{Este algoritmo busca secuencialmente en el archivo ordenado FA, un registro
con clave K}
{BANDERA es una variable de tipo booleano, R es una variable de tipo
registro}

1. Abrir el archivo FA para lectura
2. Hacer BANDERA◄— FALSO
2. Repetir mientras (no sea el fin de archivo de FA) y (BANDERA = FALSO)
        Leer R de FA
        3.1 Si (R.CLAVE ≥ K) entonces
                Hacer BANDERA◄— VERDADERO
        3.2 {Fin de la condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Si (R.CLAVE = K) entonces
        Escribir "El elemento K está en el archivo."
        si no
        Escribir "El elemento K no está en el archivo."
6. {Fin de la condicional del paso 5}
```

La diferencia entre este algoritmo y el anterior consiste en que la búsqueda también se detiene cuando la clave de R es mayor que K. Esto último se debe a que, si el archivo está ordenado, ya no se encontrará el registro con clave K entre los registros aún no visitados.

Búsqueda secuencial mediante bloques

La búsqueda secuencial mediante bloques consiste en tomar bloques de registros en vez de registros aislados. Un **bloque** es un conjunto de registros. Su tamaño es arbitrario y depende del número de elementos del archivo. Generalmente se define el tamaño del bloque igual a √N, donde N el número de registros del archivo (la demostración de por qué es √N será presentada más adelante). El archivo debe estar ordenado. La búsqueda se realiza al comparar la clave en cuestión con el último registro de cada bloque. Si la clave resulta menor, entonces se busca en forma secuencial a través de los registros salteados en el bloque. En caso contrario se continúa con el siguiente bloque. En promedio el número de comparaciones requeridas para encontrar un valor dado será igual a √N.

A continuación, se presenta un algoritmo de búsqueda secuencial usando bloques.

C) Algoritmo Archivo Secuencial con Bloques:

```
ArchivoSecuencialBloques (FA, N, K)
{Este algoritmo busca secuencialmente en el archivo ordenado FA de N elementos, un registro con clave K}
{I y TB son variables de tipo entero, BANDERA es una variable de tipo booleano, R es una variable de tipo registro}

1. Abrir el archivo FA para lectura
2. Hacer BANDERA◄— FALSO y I◄— 1
3. Hacer TB ◄— parte entera (sqrt(N)) {Calcula el tamaño del bloque como la raíz cuadrada de N}
4. Repetir mientras (TB * I ≤ N) y (BANDERA = FALSO)
        Leer R de FA en la posición TB * I
        4.1 Si (R.CLAVE ≥ K) entonces
                Hacer BANDERA◄— VERDADERO
                si no
                Hacer I◄— I + 1
        4.2 {Fin de la condicional del paso 4.1}
5. {Fin del ciclo del paso 4}
6. Si (BANDERA = VERDADERO) entonces
        6.1 Si (R.CLAVE = K) entonces
                Escribir "El elemento K está en el archivo."
                si no
                Realizar búsqueda secuencial en los registros salteados:
                del registro (TB * (I – 1) + 1) al registro (TB * I – 1)
                Reposicionar el puntero del archivo, y aplicar el algoritmo
                7.2.1 b) para ejecutar la búsqueda elemento por elemento
        6.2 {Fin de la condicional del paso 6.1}
        si no {Si TB no es múltiplo de N, quedaron elementos sin revisar}
```

Realizar búsqueda secuencial en los registros comprendidos
Entre (TB * (I – 1) + 1) y N
7. {Fin de la condicional del paso 6}

En este algoritmo se lee el último registro de cada bloque, y de la comparación del elemento buscado con él, se decide cómo continuar con la búsqueda. El siguiente ejemplo ilustra mejor el funcionamiento de este algoritmo.

Ejemplo 7.2.1 c)

Sea FA un archivo ordenado de 20 registros. Los registros ocupan posiciones consecutivas con direcciones relativas del 1 al 20. Las claves de los registros almacenados en FA son: 204, 311, 409, 415, 439, 450, 502, 507, 600, 623, 679, 680, 691, 692, 695, 698, 730, 850, 870, 889. Dado que se conoce N, se calcula el tamaño del bloque de la siguiente manera:

$$TB = \sqrt{20} \approx 4$$

La siguiente tabla presenta el seguimiento del algoritmo para la búsqueda **K = 623**.

PASO	I	REGISTRO LEÍDO	COMPARA	BANDERA
1	1	415	415 ≥ 623?	F
2	2	507	507 ≥ 623?	F
3	3	680	608 ≥ 623?	V
4		600	600 = 623?	
5		623	623 = 623?	

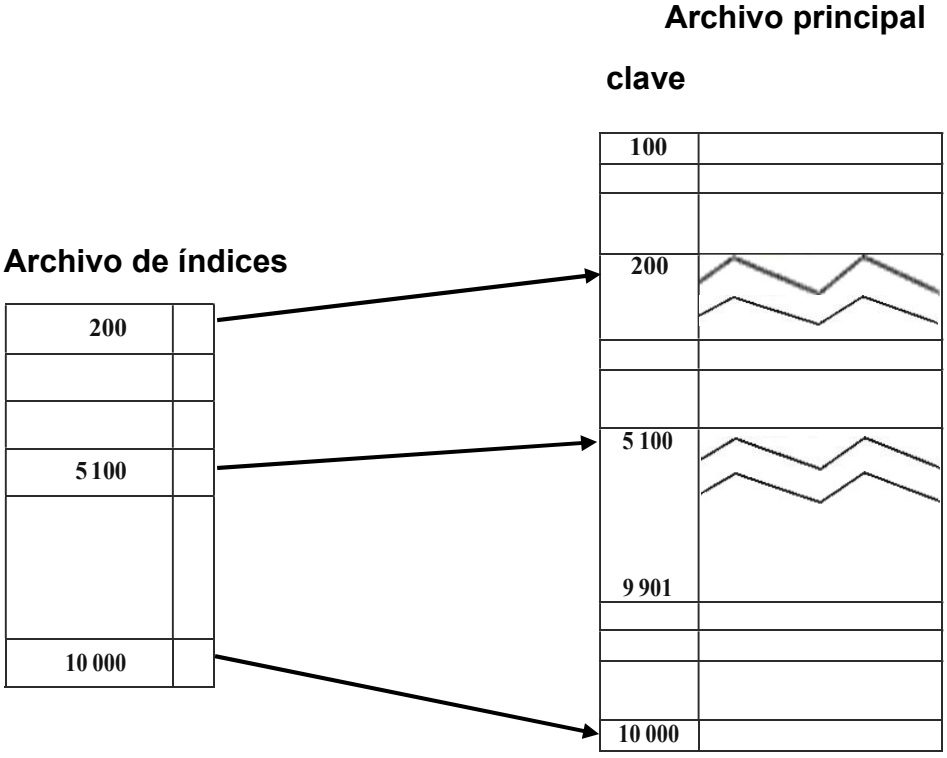
En la columna REGISTRO LEÍDO aparece el último registro del bloque, pasos 1, 2 y 3. En los pasos 4 y 5 es el valor leído durante la búsqueda secuencial a través de los registros salteados. En el paso 3, cuando se cumple la condición de que R.CLAVE ≥ K, entonces se comienza la búsqueda secuencial a partir del elemento (TB * (I - 1) + 1), en este caso el 600, hasta que se encuentra el valor deseado o hasta el elemento (TB * I - 1). Observe que en el paso 5 se encuentra el registro buscado (**ÉXITO**).

Búsqueda secuencial con índices

El método de búsqueda secuencial con índices trabaja con bloques y con archivos de índices. En el archivo de índices se almacenan las claves que hacen referencia a cada bloque y la dirección de los bloques en el archivo. La búsqueda de un elemento comienza recorriendo el archivo de índices, comparando las claves allí almacenadas con la clave del elemento en cuestión. Una vez que se determina el bloque en el cual se puede encontrar el registro buscado, se continúa la búsqueda ahora recorriendo secuencialmente dicho bloque.

La desventaja de este método es que requiere más espacio de memoria, ya que se trabaja con dos archivos: el principal, en el cual se almacenan los registros, y el de índices. Una forma de acelerar el proceso de búsqueda consiste en mantener en memoria principal el archivo de índices.

En la siguiente figura se presenta un esquema de un archivo con su correspondiente archivo de índices.



El archivo de índices se recorre secuencialmente hasta encontrar la clave que sea mayor o igual a la clave buscada. Cuando esto último suceda, se tomará la dirección del bloque apuntado por dicha clave y se aplicará búsqueda secuencial (algoritmo 7.2.1) en dicho bloque.

Determinación del tamaño del bloque

El tamaño del bloque se debe elegir de tal forma que permita reducir el número de comparaciones. Sea N el número de registros en el archivo y TB el tamaño del bloque. La probabilidad de encontrar un registro en un bloque es igual para todos los bloques; por lo tanto, el número medio de bloques examinados será:

$$\sum_{i=1}^{N/TB}\left(i * \frac{1}{(N f\ TB)}\right) \quad \frac{N f TB + 1}{2} \qquad (1)$$

Donde 1/(N/TB) representa la probabilidad de encontrar un registro en un bloque.

Considere, además, que todos los registros tienen la misma probabilidad de ser el bus- cado; por lo tanto, el número medio de registros examinados será:

$$\sum_{i=0}^{TB-1}\left(i * \frac{1}{TB}\right) = \frac{TB-1}{2} \qquad (2)$$

Donde 1/TB es la probabilidad de que el registro examinado sea el buscado.

Se suman las expresiones 1 y 2 para obtener el número total medio de comparaciones (TC) que se deben hacer para encontrar un elemento en el archivo.

$$TC = ((N/TB) + 1)/2 + (TB - 1)/2$$

Operando se obtiene:

$$TC = N/(2 * TB) + TB/2 \qquad (3)$$

Al minimizar TC se podrá determinar cuál es el tamaño adecuado para definir los bloques; es decir, el problema se reduce a encontrar un valor tal para TB que minimice el valor de TC.

$$\frac{d(TC)}{d(TB)} \quad \frac{-N}{2(TB)^2} + \frac{1}{2} \qquad (4)$$

Se iguala a cero la expresión 4 y se hacen las operaciones:

$$\frac{-N}{2(TB)^2} + \frac{1}{2} = 0 \qquad \frac{N}{2(TB)^2} = \frac{1}{2} \qquad (5)$$

De la expresión 5 se puede afirmar que el valor de TB que minimiza a TC es:

$$\boxed{TB = \sqrt{N}}$$

Los archivos de índices, por otra parte, se pueden definir a distintos niveles; es decir, se pueden definir índices de índices. Si bien este tipo de organización optimiza el tiempo de búsqueda, tiene el inconveniente de que ocupa mucho espacio de almacenamiento.

7.2.2 Binaria.

El principio que rige el método de búsqueda binaria en la búsqueda externa es el mismo que se explicó en búsqueda binaria interna, sección 7.1.2 de este libro.

El archivo debe estar ordenado y se debe conocer el número de elementos del mismo (N), para poder aplicar este método.

El lector puede desarrollarlo fácilmente, ya que conoce el método de búsqueda binaria en memoria interna.

Cabe destacar que un gran inconveniente de la búsqueda binaria externa es que requiere accesos a diferentes posiciones del dispositivo periférico en el cual está almacenado el archivo; ello produce un alto costo en tiempo de acceso, que hace impráctica esta búsqueda

7.2.3 Transformación de claves (hash).

El método de búsqueda externa por transformación de claves tiene básicamente las mismas características que el presentado en la sección 7.1.3.

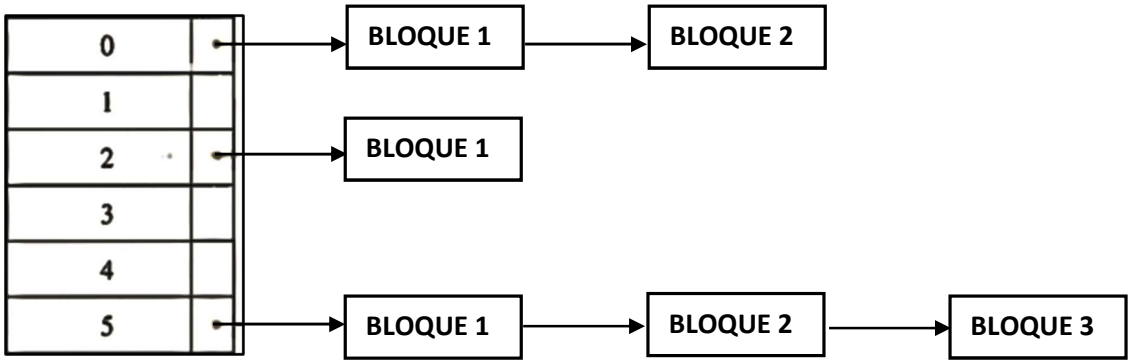
Los archivos normalmente se encuentran organizados en áreas llamadas cubetas. Éstas se encuentran formadas por cero, uno o más bloques de registros. Por lo tanto, la función hash, aplicada a una clave, dará como resultado un valor que hace referencia a una cubeta en la cual se puede encontrar el registro buscado.

Tal como se mencionó en búsqueda interna, la elección de una adecuada función hash y de un método para resolver colisiones es fundamental para lograr mayor eficiencia en la búsqueda.

Antes de presentar algunas funciones hash se hará un comentario sobre las colisiones. Los bloques contienen un número fijo de registros. Con respecto a las cubetas, no se establece un límite en cuanto al número de bloques que pueden almacenar. Esta característica permite solucionar, al menos parcialmente, el problema de las colisiones. Sin embargo, si el tamaño de las cubetas crece considerablemente, se perderán las ventajas propias de este método. Es decir, si el número de bloques que se deben recorrer en una cubeta es grande, el tiempo necesario para ello será significativo; por lo tanto, ya no se contará con la ventaja del acceso directo que caracteriza al método por transformación de claves.

En la siguiente figura se presenta una estructura de archivo organizado en cubetas, las que a su vez están formadas por bloques.

Directorio de cubetas



Archivo organizado con cubetas de bloques

Como se muestra en la figura, cada cubeta puede tener un apuntador a un bloque. Si una cubeta tiene dos o más bloques se establecen ligas entre ellos. Dada la clave de un registro buscado, se aplicará una función hash, la cual dará como resultado un número de cubeta. Una vez localizada ésta, habrá que recorrer sus bloques hasta encontrar el registro, o llegar a un bloque con puntero nulo, lo cual indicará que no existen otros bloques.

Es importante elegir una función hash que distribuya las claves en forma homogénea a través de las cubetas, de manera que se evite la concentración de numerosas claves en una cubeta mientras otras permanecen vacías.

A continuación, se presentan algunas de las funciones hash más comunes.

Funciones hash

Una función hash se puede definir como una transformación de clave a una dirección. Al aplicar una función hash a una clave se obtiene el número de cubeta en la cual se puede encontrar el registro con dicha clave.

La función debe transformar las claves para que la dirección resultante sea un número comprendido entre los posibles valores de las cubetas. Por ejemplo, si se tienen 10 000 cubetas numeradas de 0 al 9999, las direcciones producidas por la función deben ser valores comprendidos entre 0 y 9999. Si las claves fueran alfabéticas o alfanuméricas, primero deberán convertirse a numéricas, tratando de no perder información, para luego ser transformadas en una dirección. Es importante que la función distribuya homogéneamente las claves entre los números de cubetas disponibles.

Las funciones **módulo(1)**, **cuadrado(2)**, **plegamiento(3)** y **truncamiento(4)** presentadas anteriormente para búsqueda interna son válidas también para

búsqueda externa. Otra función que se puede utilizar para el cálculo de direcciones es la de conversión de bases, aunque no proporciona mayor homogeneidad en la distribución. De todas las citadas, la función módulo es, sin embargo, la que ofrece mayor uniformidad.

5) Función conversión de bases.

La conversión de bases consiste en modificar de manera arbitraria la base de la clave obteniendo un número que corresponda a una cubeta. Si el número de dígitos del valor resultante excede el orden de las direcciones, entonces se suprimirán los dígitos más. significativos

Ejemplo 7.2.3

Supongamos que se tienen 100 cubetas, cada una de ellas referenciada por un número entero comprendido entre 1 y 100. Sea **K = 7259** la clave del registro que se busca. Se elige el 9 como base a la cual se convierte la clave.

H(7259) = dígmensig ($7*9^3 + 2*9^2 + 5*9^1 + 9*9^0$)

H(7259) = dígmensig(5319) = 19

Se toma entonces como dirección el 19 y los dígitos más significativos, 5 y 3, se suprimen.

Solución de colisiones

Como se mencionó anteriormente cuando se trató búsqueda interna, uno de los aspectos que siempre se deben de considerar en el método por transformación de claves es la solución de colisiones. Cuando dos o más elementos con distintas claves tienen una misma dirección, se origina una colisión.

Para evitar las colisiones se debe elegir un tamaño adecuado de cubetas y de bloques. Con respecto a las cubetas, si se definen muy pequeñas el número de colisiones aumenta, mientras que si se definen muy grandes se pierde eficiencia en cuanto a espacio de almacenamiento. Además, si se necesitara copiar una cubeta en memoria principal y ésta fuera muy grande, ocasionaría problemas por falta de espacio. Otro inconveniente que se presenta en el caso de cubetas muy grandes es que se requiere mucho tiempo para recorrerlas.

Con respecto al tamaño de los bloques, es importante considerar la capacidad de éstos para almacenar registros. Un bloque puede almacenar uno, dos o más registros. Normalmente los tamaños de las cubetas y los bloques dependen de las capacidades del equipo con el que se esté trabajando.

Cabe destacar que utilizando una estructura como la que anteriormente se definió, no se tendría problemas de colisiones, debido principalmente a que por más que la cubeta esté ocupada, es posible seguir enlazando tantos bloques como fueran necesarios. Este esquema de solución se corresponde con el presentado en búsqueda interna, bajo el nombre de encadenamiento. Sin embargo, no siempre es posible definir una estructura de este tipo.

7.2.4 Búsqueda dinámica por transformación de claves.

La principal característica del hashing dinámico es su dinamismo para variar el número de cubetas en función de su densidad de ocupación. Se comienza a trabajar con un número determinado de cubetas, y a medida que éstas se van llenando se asignan nuevas cubetas al archivo.

Existen básicamente dos formas de trabajar con el hashing dinámico:

- Por medio de expansiones totales.
- Por medio de expansiones parciales.

a) Expansiones totales.

El método de expansiones totales es probablemente el más utilizado. Consiste en duplicar el número de cubetas en la medida en que éstas superan la densidad de ocupación previamente establecida. Así, por ejemplo, si el número inicial de cubetas es N y se hace una expansión total, el valor resultante (nuevo número de cubetas) será 2N. Si se hace una segunda expansión total, se tendrá N y así sucesivamente.

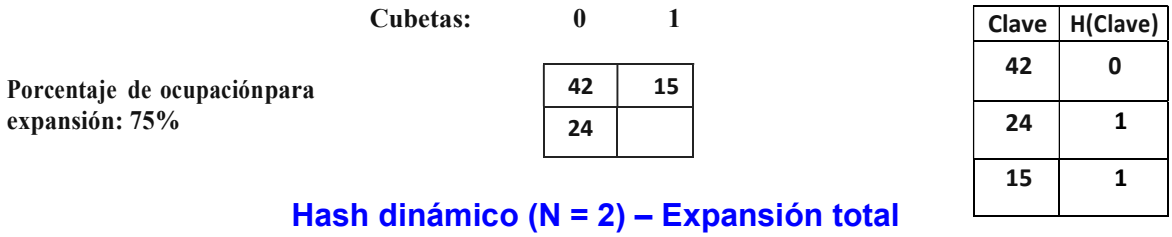
El dinamismo de este método también se da en sentido contrario; es decir, que a medida que la densidad de ocupación de las cubetas disminuye, se reduce el número de éstas. Así, se gana flexibilidad en cuanto a que se pueden incrementar los espacios de almacenamiento, pero también se pueden reducir si la demanda de espacio así lo indica.

Ejemplo 7.2.4 a)

Supongamos que se tiene un archivo organizado en dos cubetas [N = 2), y se ha fijado una densidad de ocupación de 80%. La densidad de ocupación se calcula como el cociente entre el número de registros ocupados y el de registros disponibles. Cada cubeta tiene dos registros, y la función hash que transforma claves en direcciones se define de la siguiente manera:

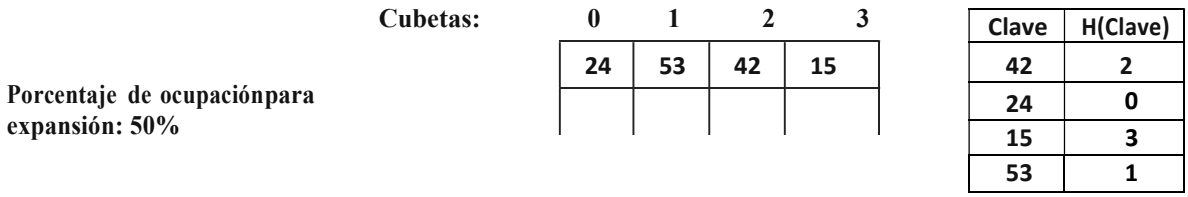
H (clave) = clave MOD Número de cubetas

Los valores 42, 24, 15 y 53 son las claves de los registros que se desea almacenar. Inicialmente el archivo está vacío. En la siguiente figura se presenta un esquema de cómo quedan las cubetas, después de insertar las tres primeras claves.



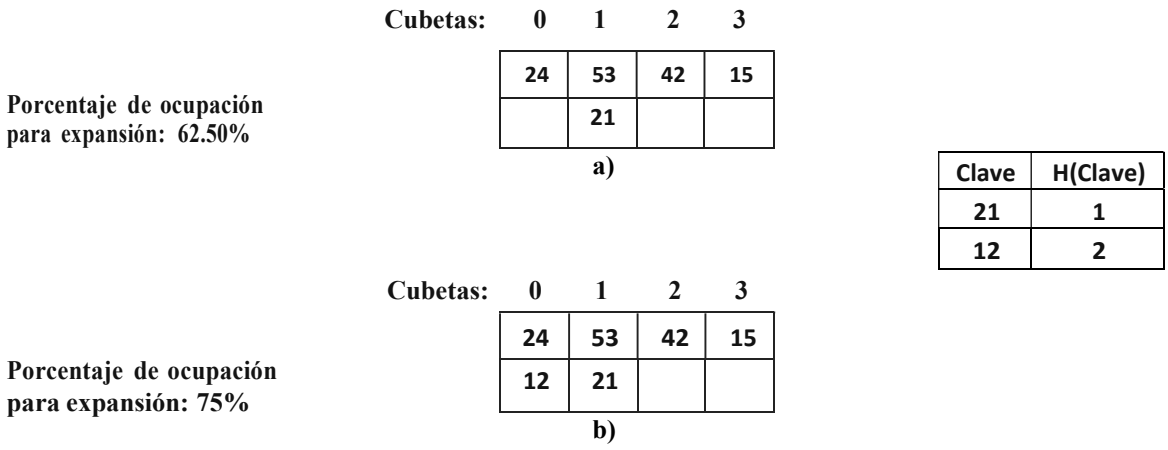
Hash dinámico (N = 2) – Expansión total

Cuando se quiere insertar la clave 53, se supera la densidad de ocupación establecida, ya que se alcanzaría 100% de llenado. Por lo tanto, se deben expandir y reasignar los registros considerando ahora que el número de cubetas es igual a 2*N.



Hash dinámico (N = 2) – Expansión total

Supongamos ahora que se desea incorporar los registros con claves 21, 12, 14, 18, 49, 128, 22, 23 y 67 en este orden. El resultado, después de insertar las dos primeras claves, se puede observar en la siguiente figura.



Hash dinámico (N = 4) – Expansión total.

a) Luego de insertar 21. b) Luego de insertar 12.

Cuando se inserta el registro con clave 14, la densidad de ocupación supera el 80% fijado. Se vuelven, entonces, a expandir y a reasignar los registros almacenados [figura a)], y luego se continúa con la inserción del resto de los elementos. La figura b) presenta el estado de las cubetas luego de realizar otras inserciones.

Cubetas: 0 1 2 3 4 5 6 7

24		42		12	53	14	15
					21		

a)

Porcentaje de ocupaciónpara expansión: 43.75%

Cubetas: 0 1 2 3 4 5 6 7

24	49	42		12	53	14	l5
128		18			21	22	23

b)

Porcentaje de ocupaciónpara expansión: 68.75%

Clave	H(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

Hash dinámico (N = 8) – Expansión total.

a) Luego de insertar 14. b) Luego de insertar 18, 49, 128, 22 y 23.

Cuando se inserta la última clave, 67, se supera nuevamente la densidad de ocupación y hay que volver a expandir las cubetas. Por lo tanto, ahora N será igual a 16.

Cubetas: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

128	49	18	67		53	22	23	24		42		12		14	15
					21										

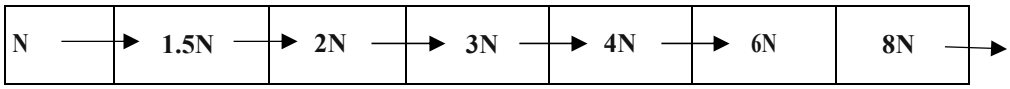
Clave	H(Clave)
42	10
24	8
15	15
53	5
21	5
12	12
14	14
18	2
49	1
128	0
22	6
23	7
67	3

Hash dinámico (N = 16) – Expansión total.

Es importante señalar que en este método también se pueden producir colisiones, las cuales podrían tratarse según alguno de los esquemas propuestos anteriormente. Por ejemplo, si en el caso anterior luego de insertar los registros con claves 24 y 128 se tratara de agregar el registro con clave 192, se produciría una colisión, ya que la cubeta 0 está llena.

b) Expansiones parciales.

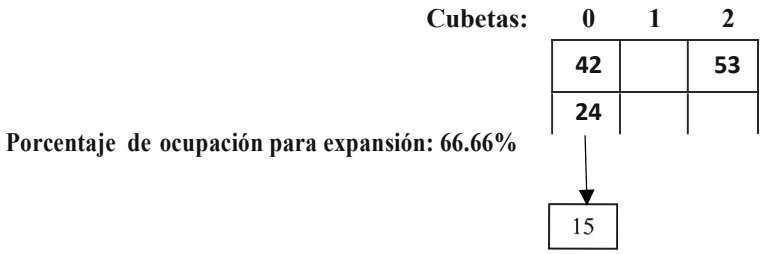
El método de las expansiones parciales consiste en incrementar en 50% el número de cubetas, haciendo de esta forma que dos expansiones parciales equivalgan a una total. Así, por ejemplo, si el número inicial de cubetas es N, y se hace una expansión parcial, el valor resultante será 1.5N. Si se hacen otras expansiones parciales se tendrá 2N, luego 3N, y así sucesivamente.



A continuación, se presenta un ejemplo de hash dinámico con expansiones parciales

Ejemplo 7.2.4 b)

Retomando el ejemplo 7.2.4. Supongamos que hasta el momento se han almacenado los registros con claves 42, 24 y 15. Cuando se quiere insertar el registro con clave 53, el número de registros supera el máximo permitido ya que la densidad de ocupación supera 80%; por tal razón se realiza una expansión parcial. La siguiente figura muestra el estado de las cubetas luego de expandir y reasignar los registros.



Clave	H(Clave)
42	0
24	0
15	0
53	2

Hash dinámico (N = 3) – Expansión parcial.

Observe que en este caso el valor de N no fue muy adecuado para distribuir uniformemente los registros a través de las cubetas. En la cubeta 0 se tiene una colisión, mientras que la cubeta 1 permanece vacía.

Supongamos ahora que se desea incorporar los registros con claves: 21, 12, 14, 18, 49, 128, 22 y 23, en ese orden. Al insertar el registro con clave 21 se supera la densidad de ocupación, por lo que se deben expandir nuevamente las cubetas y reasignar los registros. Se inserta a continuación el registro con clave 12, como se ve en la siguiente figura.

Cubetas	0	1	2	3
	24	53	42	15
	12	21		

Porcentaje de ocupación para expansión: 75%

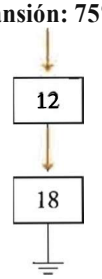
Clave	H(Clave)
42	2
24	0
15	3
53	1
21	1
12	0

Hash dinámico (N = 4) – Expansión parcial.

Al insertar el registro con clave 14, otra vez se supera el porcentaje de ocupación permitido. Se vuelven a expandir las cubetas y a reasignar los registros. El resultado final, luego de insertar todas las claves, se muestra en las siguientes figuras.

Cubetas:	0	1	2	3	4	5
	42	49	14	15		53
	24			21		

Porcentaje de ocupación para expansión: 75%



Clave	H(Clave)
42	0
24	0
15	3
53	5
21	3
12	0
14	2
18	0
49	1

a)

Cubetas:	0	1	2	3	4	5	6	7
	24	49	42		12	53	14	15
	128		18			21	22	23

Porcentaje de ocupación para expansión: 75%

Clave	H(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

b)

Hash dinámico (N = 8) – Expansión parcial.

a) Luego de expandir e insertar los registros con claves 14, 18 y 49.

b) Luego de expandir e insertar los registros con claves 128, 22 y 23.

8 – Complejidad de los Algoritmos de Búsqueda.

La complejidad es un factor involucrado en un proceso complejo. Con respecto a los algoritmos y las estructuras de datos; esto puede ser por tiempo o espacio (en disco) necesario para realizar una tarea específica como buscar, clasificar o acceder a los datos en una estructura de datos determinada. La eficiencia del rendimiento de una tarea depende del número de operaciones requeridas para completarla.

Sacar la basura puede requerir 3 pasos (atar, sacar y tirar en el basurero). Sacar la basura puede ser simple, pero si la vas a sacar después de una larga semana, es posible que no puedas completar la tarea debido a la falta de espacio en el basurero.

Aspirar una habitación puede requerir muchos pasos repetitivos (encender la aspiradora, aspirar repetidamente los espacios de la habitación y apagarla). Mientras más grande sea la habitación, más tiempo llevará aspirar la habitación.

Entonces, existe una relación entre el número de operaciones realizadas y el número de elementos operados. Tener mucha basura (elementos) requiere sacarla muchas veces. Esto puede generar un problema de complejidad espacial (space complexity). Tener una gran cantidad de metros cuadrados (elementos) requiere aspirar muchas veces. Esto puede dar lugar a un problema de complejidad en tiempo (time complexity).

Ya sea que esté sacando la basura o aspirando una habitación, podría decirse que el orden de la operación (O) incrementa exactamente con el número de elementos involucrados (n). Si tengo 1 bolsa de basura, tengo que sacar la basura una vez. Si tengo 2 bolsas de basura, tengo que hacer la misma tarea dos veces, asumiendo que físicamente no puedo levantar más de una bolsa a la vez. Entonces, la Big-O de estas tareas es O = función(n) u O(n). Esto es una complejidad lineal (linear complexity) (una línea recta entre 1 operación: 1 elemento). Entonces, 30 operaciones equivalen a 30 elementos.

Esto es similar a lo que sucede cuando se consideran algoritmos y estructuras de datos

Búsqueda lineal

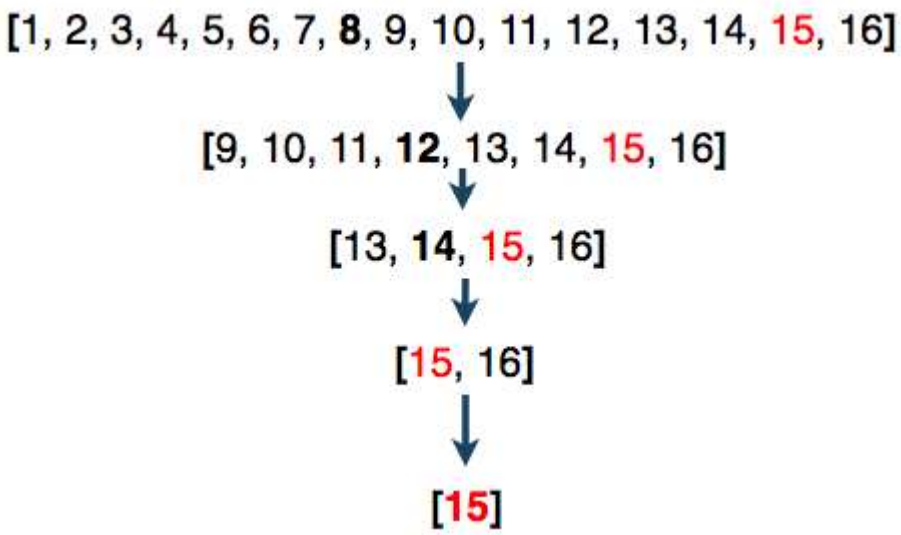
El mejor caso para buscar un elemento en una lista ordenada, es uno después del otro. Es una constante O(1), asumiendo que es el primer elemento en tu lista. Por lo tanto, si el elemento que estás buscando siempre aparece primero en la lista, independientemente del tamaño de la lista, encontrarás el elemento inmediatamente. La complejidad de la búsqueda es constante con el tamaño de la lista. El peor caso de este tipo de búsqueda es de complejidad lineal u O(n). En otras palabras, para n elementos, tengo que buscar n veces antes de encontrar mi elemento, por eso es una búsqueda lineal.

Búsqueda binaria

Para búsqueda binaria, el mejor caso es O(1), esto significa que el elemento que buscas se encuentra en el centro. El peor caso es log base 2 de n:

$$\log_2 n = \text{exponent}$$

Logaritmo o log es la manera de expresar un exponente a una base dada. Entonces, si hay 16 elementos (n = 16), esto tomará, en el peor caso, 4 pasos para encontrar el número 15 (exponente = 4).



o simplemente: O(log n)

$$\log_2 16 = 4$$

8.1 Comparación de los algoritmos.

A menudo dispondremos de más de un algoritmo para resolver un problema dado, ¿con cuál nos quedamos?, ¿cual nos dará la solución más óptima?

Para tomar una decisión, se debe realizar un análisis.

Uso de los recursos.

- ✓ **Computacionales:**
- Tiempo de ejecución
- Espacio en memoria

- ✓ **No computacionales:**
- Esfuerzo de desarrollo (análisis, diseño & implementación)

Factores que influyen en el uso de los recursos.

- ➢ **Recursos computacionales:**
- Diseño del algoritmo.
- Complejidad del problema (p.ej. tamaño de las entradas).
- Hardware (arquitectura, MHz, MBs…).
- Calidad del código.
- Calidad del compilador.

- ➢ **Recursos no computacionales:**
- Complejidad del algoritmo.
- Disponibilidad de bibliotecas reutilizables.

8.1.1 Análisis de la búsqueda secuencial:

El número de comparaciones es uno de los factores más importantes que se utilizan para determinar la complejidad de los métodos de búsqueda.

Para analizar la complejidad de la búsqueda secuencial, se deben establecer los casos más favorable y más desfavorable que se presenten. Al buscar, por ejemplo, un elemento en un arreglo unidimensional desordenado de N componentes, puede suceder que ese valor no se encuentre; por lo tanto, se harán N comparaciones al recorrer todo el arreglo. Por otra parte, si el elemento se encuentra en el arreglo, éste puede estar en la primera posición, en la última o en alguna intermedia. Si es el primero caso, se hará una comparación; si se trata del último, se harán N comparaciones; y si se encuentra en la posición i (1 < i < N), entonces se realizarán i comparaciones.

Ahora bien, el número de comparaciones que se llevan a cabo si trabajamos con arreglos ordenados, será el mismo que para desordenados, siempre y cuando el elemento se encuentre en el arreglo. Si no fuera éste el caso, entonces el número de comparaciones disminuirá sensiblemente en arreglos ordenados, siempre que el valor buscado esté comprendido entre el primero y el último elemento del arreglo.

Por otra parte, el número de comparaciones en la búsqueda secuencial en listas simplemente ligadas es el mismo que para arreglos. El orden entre los nodos de la lista influye en el número de comparaciones de igual manera que el orden entre los componentes del arreglo.

En las fórmulas siguientes, se presentan los números mínimo, mediano y máximo de comparaciones que se ejecutan cuando se trabaja con la búsqueda secuencial.

$$C_{mín} = 1 \qquad C_{med} = (1+n) / 2 \qquad C_{máx} = N$$

La siguiente tabla presenta, para distintos valores de N, los números mínimo, mediano y máximo de comparaciones que se requieren para buscar secuencialmente un elemento en un arreglo o lista ligada.

N	Cmín	Cmed	Cmáx
10	1	5.5	10
100	1	50.5	100
500	1	250.5	500
1000	1	500.5	1000
10000	1	5000.5	10000

Tabla 7.1.1 Complejidad del método de búsqueda secuencial

8.1.2 Análisis de la búsqueda binaria:

Para analizar la complejidad del método de búsqueda binaria es necesario establecer los casos más favorables y desfavorables que se pudieran presentar en el proceso de búsqueda. El primero sucede cuando el elemento buscado es el central, en dicho caso se hará una sola comparación; el segundo sucede cuando el elemento no se encuentra en el arreglo; entonces se harán aproximadamente $\log_2(n)$ comparaciones, ya que con cada comparación el número de elementos en los cuales se debe buscar se reduce en un factor de 2. (No olvidar que N es el número de elementos del arreglo).

De esta forma, se determinan los números mínimo, medio y máximo de comparaciones a realizar cuando se utiliza el algoritmo de búsqueda binaria.

$$C_{mín} = 1 \qquad C_{med} = (1+\log_2(N)) / 2 \qquad C_{máx} = \log_2(N)$$

En la siguiente tabla se presentan, para distintos valores de N, los números mínimo, medio y máximo de comparaciones requeridas para buscar un elemento en un arreglo. aplicando el método de búsqueda binaria.

N	Cmín	Cmed	Cmáx
10	1	2.5	4
100	1	4	7
500	1	5	9
1000	1	5.5	10
10000	1	7.5	14

Tabla 7.1.2 Complejidad del método de búsqueda binaria

Si se comparan los valores de la tabla 7.1.1 con los de la tabla 7.1.2, resulta claro que el método de búsqueda binaria es más eficiente que el método de búsqueda secuencial. Además, la diferencia se hace más significativa conforme más grande sea el tamaño del arreglo. Sin embargo, no hay que olvidar que el método de búsqueda binaria trabaja solamente con arreglos ordenados; por lo tanto, si el arreglo estuviera desordenado antes de emplear este método, aquél debería ordenarse.

Cabe destacar, sin embargo, que la ordenación de un arreglo también implica comparaciones y movimientos de elementos, así que si se va a realizar sólo una búsqueda sobre un arreglo desordenado conviene utilizar el método secuencial. En cambio, si se realizan búsquedas en forma continua, puede resultar más conveniente ordenarlo para poder aplicar el método de búsqueda binaria en él.

8.1.3 Análisis del método por transformación de claves:

Para analizar la complejidad de este método es necesario realizar varios cálculos probabilísticos, que no se estudiarán aquí. La dificultad del análisis se debe principalmente a que no sólo interviene la función hash sino también el método utilizado para resolver las colisiones. Por lo tanto, se debería analizar cada una de las posibles combinaciones que se pudieran presentar.

Sea **λ** (lambda minúscula) el factor de ocupación de un arreglo, definido como M/N, donde M es el número de elementos en el arreglo y N es el tamaño del mismo. Según Lipschutz, la probabilidad de llevar a cabo una búsqueda con éxito (S) y otra sin éxito (Z), quedan determinadas por las siguientes fórmulas:

$S(\lambda) = \frac{(1 + 1 / (1 - \lambda))}{2}$	$Z(\lambda) = \frac{(1 + 1 / (1 - \lambda)^2)}{2}$
a) Búsqueda con éxito.	b) Búsqueda sin éxito.

Cabe aclarar que estas fórmulas son válidas solamente en caso de funciones hash con el método lineal de solución de colisiones.

8.1.3 Análisis del árbol binario de búsqueda:

La búsqueda en un árbol binario de búsqueda consiste en acceder a la raíz del árbol, si el elemento a localizar coincide con este la búsqueda ha concluido con éxito, si el elemento es menor se busca en el subárbol izquierdo y si es mayor en el derecho. Si se alcanza un nodo hoja y el elemento no ha sido encontrado es que no existe en el árbol.

Cabe destacar que la búsqueda en este tipo de árboles es muy eficiente, representa una función logarítmica. El máximo número de comparaciones que necesitaríamos para saber si un elemento se encuentra en un árbol binario de búsqueda estaría entre $[\log_2(N+1)]$ y N, siendo N el número de nodos.

La búsqueda de un elemento en un ABB (Árbol Binario de Búsqueda) se puede realizar de dos formas, iterativa o recursiva.

Tipos de árboles binarios de búsqueda

Hay varios tipos de árboles binarios de búsqueda. Los árboles AVL, árbol rojo-negro, son árboles autobalanceables. Los árboles biselados son árboles también autobalanceables con la propiedad de que los elementos accedidos recientemente se accederán más rápido en posteriores accesos. En el montículo, como en todos los árboles binarios de búsqueda, cada nodo padre tiene un valor mayor que sus hijos y además es completo, esto es cuando todos los niveles están llenos con excepción del último que puede no estarlo. Por último, en los montículos con prioridad cada nodo mantiene una prioridad y siempre un nodo padre tendrá una prioridad mayor a la de su hijo.

Los árboles de búsqueda balanceados permiten realizar las operaciones de búsqueda, inserción y borrado con complejidad ***O(log(n))***.

Comparación de rendimiento

D. A. Heger (2004) realiza una comparación entre los diferentes tipos de árboles binarios de búsqueda para encontrar que tipo nos daría el mejor rendimiento para cada caso. Los montículos se encuentran como el tipo de árbol binario de búsqueda que mejor resultado promedio da, mientras que los árboles rojo-negro los que menor rendimiento medio nos aporta.

8.2 Principio de invarianza.

Sea $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n.

Teóricamente $T(n)$ indica el número de instrucciones ejecutadas por una computadora idealizada.

- se deben buscar medidas simples y abstractas independientes de la computadora utilizada
- acotar la diferencia entre las distintas implementaciones de un mismo algoritmo

A continuación, se formula el **principio de invarianza:**

Dado un algoritmo y dos de sus implementaciones I_1 e I_2 con tiempos $T_1(n)$ y $T_2(n)$ respectivamente.

El principio afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

Con esto, podemos definir que un algoritmo tarda un tiempo del orden $T(n)$ si existe un constante real $c > 0$ y una implementación I del algoritmo que tarda menos que $cT(n)$, para todo n tamaño de la entrada.

Dos factores a tener muy en cuenta son la constante multiplicativa y el n_0 para los que se verifican las condiciones, pues si bien a priori un algoritmo de orden cuadrático es mejor que uno de orden cúbico, en el caso de tener dos algoritmos cuyos tiempos de ejecución son 10^6n^2 y $5n^3$ el primero sólo será mejor que el segundo para tamaños de la entrada superiores a 200,000.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, datos que se encuentren ya ordenados, a los datos que haya que ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta.

Así suelen estudiarse tres casos para un mismo algoritmo:

- Peor caso
- Mejor caso
- Caso promedio

El mejor caso corresponde a la traza (secuencia de instrucciones) del algoritmo que realiza menos instrucciones.

El peor caso corresponde a la traza (secuencia de instrucciones) del algoritmo que realiza más instrucciones.

El caso promedio corresponde a la traza (secuencia de instrucciones) del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la

variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de entrada dado, con las probabilidades de que estas ocurran para esa entrada.

Es muy importante destacar que esos casos corresponden a un tamaño de la entrada dado, puesto que es un error común confundir el caso mejor con el que menos instrucciones realiza, en cualquier caso, y por lo tanto contabilizar las instrucciones que hace para $n = 1$.

A la hora de medir el tiempo, siempre lo haremos en función del número de operaciones elementales que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa. También es posible distinguir entre los tiempos de ejecución de las diferentes operaciones elementales, lo cual es necesario a veces por las características específicas del ordenador (por ejemplo, se podría considerar que las operaciones + y ÷ presentan complejidades diferentes debido a su implementación). Sin embargo, en este texto tendremos en cuenta, a menos que se indique lo contrario, todas las operaciones elementales del lenguaje, y supondremos que sus tiempos de ejecución son todos iguales.

Reglas generales para el cálculo del número de OE (operaciones elementales)

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

- Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el principio de invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

- El tiempo de ejecución de la sentencia “CASE C OF v1: S1|v2: S2|...|vn: Sn END;” es $T = T(C) + \max\{T(S_1), T(S_2), ..., T(S_n)\}$. Obsérvese que $T(C)$ incluye el tiempo de comparación con $v_1, v_2, ..., v_n$.
- El tiempo de ejecución de la sentencia “IF C THEN S1 ELSE S2 END;” es $T = T(C) + \max\{T(S1), T(S2)\}$.
- El tiempo de ejecución de un bucle de sentencias “WHILE C DO S END;” es $T = T(C) + (\text{nº iteraciones}) * (T(S) + T(C))$. Obsérvese que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.
- Para calcular el tiempo de ejecución del resto de sentencias iterativas (*FOR, REPEAT, LOOP*) basta expresarlas como un bucle *WHILE*.
- El tiempo de ejecución de una llamada a un procedimiento o función *F*(*P1, P2,..., Pn*) es 1 (por la llamada), más el tiempo de evaluación de los parámetros *P1, P2,..., Pn*, más el tiempo que tarde en ejecutarse *F*, esto es, $T = 1 + T(P_1) + T(P_2) + ... + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia.

8.3 Eficiencia.

Existen numerosos enfoques a la hora de resolver un problema. ¿Cómo elegir el más adecuado entre ellos?

Entre las líneas de acción fundamentales en el diseño se suelen plantear dos objetivos, a veces conflictivos y contradictorios entre sí:

1. Diseñar un algoritmo que sea fácil de entender, codificar y depurar.
2. Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadora.

Si el problema es sencillo o no hay que resolver muchos casos se podría elegir el más “fácil”.

Si el problema es complejo o existen muchos casos para resolver, habría que elegir “el algoritmo que menos recursos utilice”.

Los recursos más importantes son el tiempo de ejecución y el espacio de almacenamiento. Generalmente, el más importante es el tiempo.

Al hablar de la eficiencia de un algoritmo nos referiremos a lo “rápido” que se ejecuta. La eficiencia de un algoritmo dependerá, en general, del “tamaño” de los datos de entrada.

A la hora de estudiar la eficiencia de distintos algoritmos es necesario distinguir problemas de ejemplares.

Un **problema** sería una “proposición dirigida a averiguar el modo de obtener un resultado cuando ciertos datos son conocidos.”

Un **ejemplar**, en cambio, es un “individuo de una especie o género.”

Ejemplos de problemas serían: “multiplicar dos números enteros” o “resolver una ecuación de segundo grado”.

Ejemplares de dichos problemas serían: (425,7) o (1,1,2). Es decir: 425 *·7 y x2+x+2.

Un problema dado puede tener un número infinito de ejemplares (multiplicar dos números enteros) o finito (jugar una partida de ajedrez).

Un algoritmo que afirma resolver un problema debe resolver todos los ejemplares del mismo. Si hay un solo ejemplar que no sea resuelto por el algoritmo el algoritmo no es correcto.

No todos los ejemplares de un problema son iguales. En el caso de la “multiplicación de dos números enteros” no es lo mismo multiplicar (a y b ≠ 0) que a *·0.

Al no ser todos los ejemplares iguales, la eficiencia del algoritmo puede variar en función del ejemplar o del grupo de ejemplares.

A la hora de analizar un algoritmo es necesario saber que pueden darse tres tipos de ejemplares o casos:

Caso mejor: se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente; por ejemplo: multiplicar un número por cero, insertar en una lista vacía, ordenar un vector que ya está ordenado, etc.

Caso peor: se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente. Ejemplos: insertar al final de una lista, ordenar un vector que está ordenado en orden inverso, etc.

Caso medio: se trata del resto de ejemplares del problema. Por ejemplo: multiplicar dos números enteros distintos de cero, insertar en una lista, pero que no sea el principio ni el final, ordenar un vector que no está ordenado ni en orden directo ni inverso, etc. Es el caso que más nos debería de importar, puesto que será el más habitual, sin embargo, no siempre se puede calcular (habría que saber cuáles son las probabilidades de los distintos ejemplares).

Una **operación elemental** es aquella cuyo tiempo de ejecución tiene una cota superior constante que sólo depende de su implementación (por ejemplo: el ordenador o el lenguaje de programación utilizado).

Las operaciones elementales son consideradas de coste unitario. A la hora de analizar un algoritmo importarán, por tanto, el número de operaciones elementales que precisa.

La decisión de determinar que una operación determinada es de coste unitario dependerá de los ejemplares del problema que la utilice. Por ejemplo, en la mayor parte de los casos la suma se considera de coste unitario puesto que al operar con los números que se manejan habitualmente en un ordenador los tiempos que emplea la suma son similares; sin embargo, en caso de manejar números muy grandes la suma no tendrá un coste unitario puesto que tardará más cuanto más grandes sean los números a sumar.

El tamaño de un ejemplar sería el número de bits necesarios para representarlo en el ordenador. Sin embargo, no es necesario llegar a ese nivel de detalle; basta con determinar algún parámetro que nos permita dirimir entre dos ejemplares cuál es el “mayor”. Por ejemplo:

Si tenemos un algoritmo que trabaja con vectores parece obvio que cuantos más componentes tiene un vector “mayor” será; así pues, en este caso el tamaño será el número de componentes.

Si escribimos un algoritmo que permite sumar números de cualquier número de cifras el tamaño de los números dependerá del número de cifras que contengan.

Un algoritmo para multiplicar matrices considerará como tamaño al producto del número de filas por el número de columnas.

En resumen, para cada problema es necesario analizar la naturaleza de los datos para determinar qué parámetro define el tamaño de los ejemplares.

La **eficiencia de los algoritmos** siempre vendrá dada en función de dichos tamaños:

- Algoritmos con vectores en función del número de componentes.
- Algoritmos con matrices en función del número de filas y columnas.
- Etc.

Sabemos que nos interesa saber qué algoritmo utiliza más eficientemente los recursos, en general, el tiempo de ejecución.

Esta eficiencia en tiempo depende del tamaño de los datos de entrada que vendrá dado por algún parámetro que define dichos datos (número de componentes, filas y columnas, etc.)

La eficiencia además de depender del tamaño de los ejemplares depende de la naturaleza de los mismos: caso mejor, caso peor y caso medio.

Para calcular la eficiencia es posible considerar sólo el número de veces que se debe ejecutar una operación elemental pues éstas tendrán un coste unitario.

¿En qué “unidad” mediremos esta eficiencia?

Podríamos pensar en utilizar segundos, sin embargo, el tiempo de ejecución depende del ordenador utilizado...

La solución está en el principio de invarianza, dicho principio afirma que dos implementaciones distintas del mismo algoritmo no diferirán en su eficiencia más que en una constante multiplicativa.

Por ejemplo, si un algoritmo se compila en un Pentium III 1MHz y en un 486 tendríamos dos implementaciones distintas del mismo algoritmo siendo más eficiente la primera. Si al ejecutar este algoritmo en el Pentium III para un problema dado tarda 3 segundos y en el 486 para el mismo problema tarda 30 segundos, el

principio de invarianza nos dice que un problema que en el Pentium III requiera 1 minuto necesitaría en el 486 aproximadamente 10.

De esta forma, no habrá unidad para medir la eficiencia, nos limitaremos a decir que el tiempo de ejecución de un algoritmo será *T(n)*, es decir, una función del tamaño de los ejemplares; en unas máquinas el tiempo de ejecución será a *·*T(n)* y en otras será b *·*T(n)*.

Sabemos que podemos definir la eficiencia de un algoritmo como una función T(n). A la hora de analizar un algoritmo nos interesa, principalmente, la forma en que se comporta el algoritmo al aumentar el tamaño de los datos; es decir, cómo aumenta su tiempo de ejecución.

Esto se conoce como eficiencia asintótica de un algoritmo y nos permitirá comparar distintos algoritmos puesto que deberíamos elegir aquellos que se comportarán mejor al crecer los datos.

La notación asintótica se describe por medio de una función cuyo dominio es el conjunto de números naturales, N.

Las notaciones asintóticas estudian el comportamiento del algoritmo cuando el tamaño de las entradas es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes.

Orden de eficiencia

Un algoritmo tiene un tiempo de ejecución de orden ***f(n)***, para una función dada *f*, si existe una constante positiva *c* y una implementación del algoritmo capaz de resolver cada caso del problema en un tiempo acotado superiormente por *c* *·*f (n)*, donde *n* es el tamaño del problema considerado.

8.4 Notaciones asintóticas

Las cotas de complejidad, también llamadas **medidas asintóticas** sirven para clasificar funciones de tal forma que podamos compararlas.

Las medidas asintóticas permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute.

Existen diversas notaciones asintóticas para medir la complejidad, las tres cotas de complejidad más comunes son:

1) **La cota superior, que se expresa mediante la notación O (o grande).**
2) **La cota inferior, que se expresa mediante la notación Ω (omega grande).**
3) **La cota ajustada asintótica, expresada con la notación θ (theta grande).**

Todas se basan en el peor caso.

8.4.1 Notación O Grande.

El número de sentencias ejecutadas en una función para *n* datos está en función del número de elementos y se expresa por la formula *f(n)*. Aunque la ecuación para obtener una función puede ser compleja, el factor dominante que se debe considerar para determinar el orden de magnitud del resultado es el denominado factor de eficiencia. Por consiguiente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que determina la magnitud. Este factor se define como **"O grande"**, que representa "está en el orden de" y se expresa como ***O(n)***, es decir, "está en el orden de *n*".

La notación *O* indica la cota superior del tiempo de ejecución de un algoritmo o programa. Así, en lugar de decir que un algoritmo emplea un tiempo de *4n-1* en procesar un arreglo de longitud *n*, se dirá que emplea un tiempo *O(n)* que se lee "*O grande de n*", o bien "*O de n*" y que informalmente significa "algunos tiempos constantes de *n*".

Descripción de tiempos de ejecución con la notación O.

Sea *T(n)* el tiempo de ejecución de un programa, medido como una función de la entrada de tamaño *n*. Se dice que *"T(n) es O(g(n))"* si *g(n)* acota superiormente a *T(n)*. De modo más riguroso, *T(n)* es *O(g(n))* si existe un entero n_0 y una constante $c > 0$ tal que para todos los enteros $n \geq n_0$ se tiene que $T(n) \leq cg(n)$.

Ejemplo 8.4.1 a)

Dada la función $f(n) = n^3 + 3n + 1$, encontrar su "O grande" (complejidad asintótica)

Para valores de $n \geq 1$ se puede demostrar que:

$f(n) = n^3 + 3n + 1 \leq n^3 + 3n^3 + 1n^3 = 5n^3$

Escogiendo la constante $c = 5$ y $n_0 = 1$ se satisface la desigualdad $f(n) \leq 5n^3$. Entonces se puede asegurar que: ***f(n) = O(n3)***

Ahora bien, también se puede asegurar que $f(n) = O(n^4)$ y que $f(n) = O(n^5)$, y así sucesivamente con potencias mayores de *n*. Sin embargo, lo que realmente interesa es la cota superior más ajustada que informa de la tasa de crecimiento de la función con respecto a *n*.

Una función f(x) puede estar acotada superiormente por un numero indefinido de funciones a partir de ciertos valores x_0:

$$f(x) \leq g(x) \leq h(x) \leq k(x) \ldots$$

La complejidad asintótica de la función f(x) se considera que es la cota superior más ajustada: ***f(x) = O(g(x))***

Determinar la notación O grande.

La notación *O grande* se puede obtener de f(n) utilizando los siguientes pasos:

1. En cada término, establecer el coeficiente del término en 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor.

Ejemplo 8.4.1 b)

Calcular la función *O grande* de eficiencia de la siguiente función:

$f(n) = 1/2n\ (n + 1) = 1/2n^2 + 1/2n$

1. Se eliminan todos los coeficientes y se obtiene:

$$n^2 + n$$

2. Se eliminan los factores más pequeños:

$$n^2$$

3. La notación *O grande* correspondiente es:

$$\mathbf{O(f(n)) = O(n^2)}$$

Ejemplo 8.4.1 c)

Calcular la función *O grande* de eficiencia de la siguiente función:

$f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n^1 + a_0 n^0$

1. Se eliminan todos los coeficientes y se obtiene:

$$n^k + n^{k-1} + \ldots + n^2 + n + 1$$

2. Se eliminan los factores más pequeños y el termino de exponente es el primero: n^k
3. La notación O grande correspondiente es:

$$\mathbf{O(f(n)) = O(n^k)}$$

Los diferentes tipos de complejidad.

Existen diferentes tipos de complejidad, lo que se desea de un algoritmo es que su complejidad sea la menor posible. A continuación, se presentan diferentes tipos de complejidad (las más comunes) ordenadas de menor a mayor.

***O(1)* Complejidad constante.** Es la más deseada. Por ejemplo, es la complejidad que se presenta en secuencias de instrucciones sin repeticiones o ciclos.

***O(log n)* Complejidad logarítmica.** Puede presentarse en algoritmos iterativos y recursivos).

***O(n)* Complejidad lineal.-** Es buena y bastante usual. Suele aparecer en un ciclo principal simple cuando la complejidad de las operaciones interiores es constante.

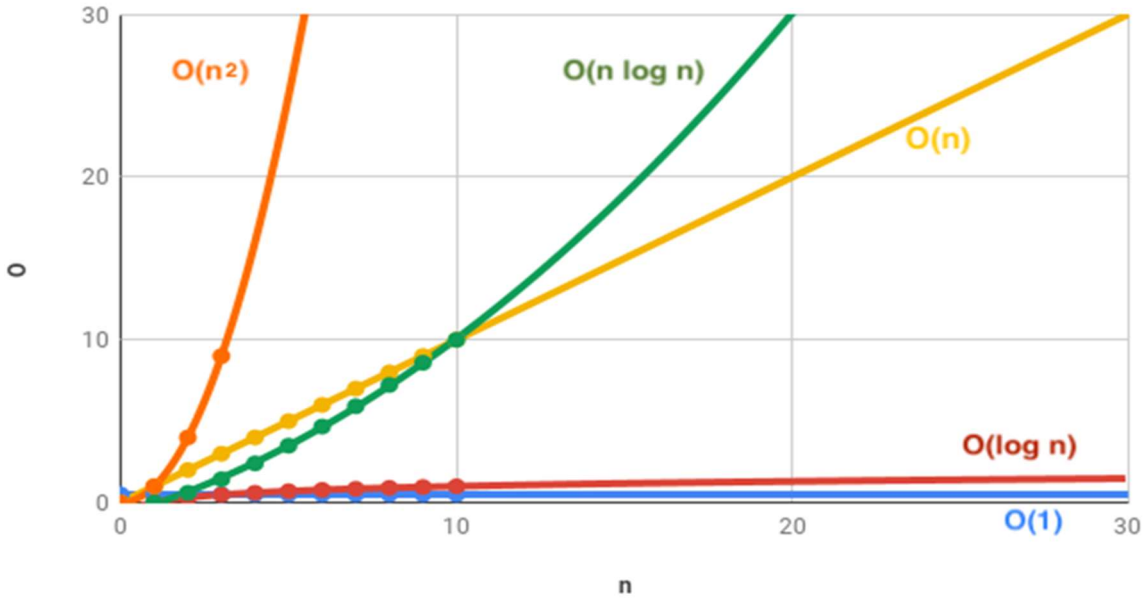
***O(n log n)* Complejidad n·log·n.-** Se considera buena. Aparece en algunos algoritmos de ordenamiento.

***O(n^2)* Complejidad cuadrática.** Aparece en ciclos anidados, por ejemplo, ordenación por burbuja. También en algunas recursiones dobles.

***O(n3)* Complejidad cúbica.** En ciclos o en algunas recursiones triples. El algoritmo se vuelve lento para valores grandes de n.

***O(n^k)* Complejidad polinomial.** Para (k ∈ N, k>3) mientras más crece k, el algoritmo se vuelve más lento.

***O(C^n)* Complejidad exponencial.** Cuando n es grande, la solución de estos algoritmos es prohibitivamente costosa. Por ejemplo, problemas de explosión combinatoria.



8.5 Calculo de la eficiencia de un algoritmo.

Nos interesa saber qué algoritmo utiliza más eficientemente los recursos, en general, el tiempo de ejecución.

Esta eficiencia en tiempo depende del tamaño de los datos de entrada que vendrá dado por algún parámetro que define dichos datos (número de componentes, filas y columnas, etc.).

La eficiencia además de depender del tamaño de los ejemplares depende de la naturaleza de los mismos: caso mejor, caso peor y caso medio.

Para calcular la eficiencia es posible considerar sólo el número de veces que se debe ejecutar una operación elemental pues éstas tendrán un coste unitario.

¿En qué "unidad" mediremos esta eficiencia? Podríamos pensar en utilizar segundos, sin embargo, el tiempo de ejecución depende del ordenador utilizado.

Un buen algoritmo es correcto, pero un gran algoritmo es correcto y además eficiente. El algoritmo más eficiente es aquel que toma el mínimo tiempo de ejecución y uso de memoria posibles, y todavía produce una respuesta correcta.

Contar las operaciones

Una forma de medir la eficiencia de un algoritmo es contar cuántas operaciones necesita para encontrar la respuesta con diferentes tamaños de la entrada.

Reglas de cálculo de la eficiencia

1. Sentencias simples
2. Bloques de sentencias
3. Sentencias condicionales
4. Bucles
5. Llamadas a funciones
6. Funciones recursivas

1. Sentencias simples:

Consideraremos que cualquier sentencia simple (lectura, escritura, asignación, etc.).

Va consumir un tiempo constante, ***O(1)***, salvo que contenga una llamada a una función.

2. Bloques de sentencias:

Tiempo total de ejecución = Suma de los tiempos de ejecución de cada una de las sentencias del bloque.

Orden de eficiencia = Máximo de los órdenes de eficiencia de cada una de las sentencias del bloque.

3. Sentencias condicionales:

El tiempo de ejecución de una sentencia condicional es el máximo del tiempo del bloque if y del tiempo del bloque else.

Si el bloque if es *O(f(n))* y el bloque else es *O(g(n))*, la sentencia condicional será ***O(max{f(n), g(n)})***.

4. Bucles:

Tiempo de ejecución = Suma de los tiempos invertidos en cada iteración.

En el tiempo de cada iteración se ha de incluir el tiempo de ejecución del cuerpo del bucle y también el asociado a la evaluación de la condición (y, en su caso, la actualización del contador).

Si todas las iteraciones son iguales, el tiempo total de ejecución del bucle será el producto del número de iteraciones por el tiempo empleado en cada iteración.

5. Llamadas a funciones:

Si una determinada función P tiene una eficiencia de *O(f(n))*, cualquier llamada a P es de orden ***O(f(n))***.

Las asignaciones con diversas llamadas a función deben sumar las cotas de tiempo de ejecución de cada llamada.

La misma consideración es aplicable a las condiciones de bucles y sentencias condicionales.

6. Funciones recursivas:

Las funciones de tiempo asociadas a funciones recursivas son también recursivas.

p.ej. $T(n) = T(n-1) + f(n)$

Para analizar el tiempo de ejecución de un algoritmo recursivo, le asociamos una función de eficiencia desconocida, *T(n)*, y la estimamos a partir de *T(k)* para distintos valores de *k* (menores que *n*).

Ejemplo 8.5 a) calcular el factorial de un número.

```
int fact(int n)
{
if (n <= 1)
return 1;
else
return (n * fact(n – 1));
}
```

Los bloques if y else son operaciones elementales, por lo que su tiempo de ejecución es *O(1)*.

T(n) = 1 + T(n - 1)

= 1 + (1 + T(n-2)) = 2 + T(n-2)

= 2 + (1 + T(n-3)) = 3 + T(n-3)

…

= i + T(n-i)

…

= (n-1) + T(n – (n-1)) = (n-1) + 1

Por tanto, ***T(n) es O(n)***

La implementación recursiva de calcular el factorial de un número, es de orden lineal.

Ejemplo 8.5 b) calcular la suma de dos números con ciclos.

Paso 1

```
para i de 1 a n
        para j de 1 a n
                escribir i + j          O(1) La complejidad de esta sentencia es constante
                siguiente j
        siguiente i
```

Paso 2

```
para i de 1 a n
        para j de 1 a n
                escribir i + j          O(n·1) = O(n) La complejidad del ciclo es el producto
                                                      del número de ejecuciones X la complejidad
                                                      de la/s sentencia/s ejecutada/s.
                siguiente j
        siguiente i
```

Paso 3

```
para i de 1 a n
        para j de 1 a n
                escribir i + j          O(n·n) = O(n²) La complejidad del ciclo es el producto
                                                       del número de ejecuciones X la complejidad
                                                       de la/s sentencia/s ejecutada/s.
                siguiente j
        siguiente i
```

Por tanto, ***T(n) es O(n²)***

La implementación del cálculo de operaciones con ciclos, es de orden cuadrático.

8.6 Resolución de recurrencias.

La palabra “recursividad” no aparece en el diccionario de la Real Academia. La palabra castellana que se emplea para tratar de lo que se va a hablar en este capítulo es la de **Recurrencia**. El diccionario de la RAE define así esta palabra:

1. f. Cualidad de recurrente. 2. f. Mat. Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes.

Aquí emplearemos ambas palabras de forma indistinta. La recursividad está presente en muchos sistemas del mundo real. Y por eso, porque muchos problemas que queremos afrontar tienen una esencia recursiva, la recursividad es una herramienta conveniente y muy útil para diseñar modelos informáticos de la realidad que deseamos modelizar. Se dice que un sistema es recursivo cuando está parcial o completamente definido en términos de sí mismo.

En general, se trata de una técnica de programación que permite crear instrucciones que se repitan un número n de veces, evitando el uso de estructuras de datos repetitivas.

Hay muchos problemas matemáticos que se resuelven mediante técnicas recursivas, por ejemplo, los siguientes:

El **algoritmo de Euclides** es recurrente. El máximo común divisor de dos números m y n ($mcd(m,n)$) se puede definir como el máximo común divisor del segundo (de n) y del resto de dividir el primero con el segundo: $mcd\ m,n = mcd(n,m\ mod\ n)$.

La secuencia se debe parar cuando se alcanza un valor de segundo entero igual a cero: de lo contrario en la siguiente vuelta se realizaría una división por cero.

Él **cálculo del factorial**: podemos definir el factorial de un entero positivo n ($n!$) como el producto de n con el factorial de $n – 1$: $n! = n \cdot (n – 1)$.

De nuevo esta definición deberá tener un límite: el momento en el que se llega al valor cero: el factorial de cero es, por definición, igual a uno.

La búsqueda de un **término de la serie de Fibonacci**: Podemos definir la serie de Fibonacci como aquella cuyo término n es igual a la suma de los términos ($n – 1$) y ($n – 2$): $Fibn = Fibn-1 + Fibn-2$.

Nuevamente se tiene el límite cuando $n \leq 2$ donde el valor del término es la unidad.

Cálculo de una **fila del triángulo de Tartaglia**. Es conocida la propiedad del triángulo de Tartaglia según la cual, si calculamos el valor de cualquier elemento situado en el primer o último lugar de cada fila, el valor de ese elemento es igual a uno; y el valor de cualquier otro elemento del triángulo resulta igual a la suma de los elementos de su fila anterior que están encima de la posición que deseamos calcular. Es decir:

$t_{i,0}$=1; $t_{i,i}$=1; $t_{i,\,j(j\neq 0, j\neq i)}$= $t_{i-1,j-1}$+ $t_{i-1,j}$.

donde $t_{i,j}$= (i/j), donde siempre se verifica que 0≤j≤i.

Y así tenemos que para calcular una fila del triángulo de Tartaglia no es necesario acudir al cálculo de binomios ni de factoriales, y basta con conocer la fila anterior, que se calcula si se conoce la fila anterior, que se calcula si se conoce la fila anterior… hasta llegar a las dos primeras filas, que están formadas todo por valores uno.

Tenemos, por tanto, y de nuevo, un camino recurrente hecho a base de simples sumas para encontrar la solución que antes habíamos buscado mediante el cálculo de tres factoriales, dos productos y un cociente para cada elemento.

En todos los casos hemos visto un camino para definir un procedimiento recurrente que llega a la solución deseada.

Técnicas de resolución de recurrencias

Cuando analizamos algoritmos recursivos es útil describir la función de coste como una recurrencia.

Una **recurrencia** es una ecuación que describe una función en términos del propio valor de la función para argumentos más cercanos a algún caso básico, para el que la función está definida explícitamente.

Resolver una recurrencia significa encontrar la expresión explícita que define la función recurrente.

Generalmente, nos bastara con clasificar la función recurrente en una notación asintótica.

Existen diferentes técnicas para la resolución de recurrencias, a continuación, se realizará un análisis de ellas.

8.6.1 Método de sustitución.

Es el método más simple y sencillo

- Se va evaluando la recurrencia para ciertos valores.
- Se deduce, a partir del comportamiento mostrado, una ecuación que represente el comportamiento de la recurrencia.
- Se generaliza la solución, encontrando un patrón.
- Se demuestra que la ecuación, efectivamente, resuelve a la recurrencia.

Ejemplo – Ordenamiento de Selección Recursivo para ordenar una lista L de tamaño n.

Si $n > 1$:

Paso 1. Buscar el menor elemento de L

Paso 2. Intercambiar el menor elemento con el primer elemento de L

Paso 3. Ordenar usando el Ordenamiento de Selección Recursivo la sublista de tamaño $n - 1$ que resulta de ignorar el primer elemento de L.

Cantidad de intercambios que se hacen en el algoritmo:

$f(1) = 0$

$f(n) = 1 + f(n - 1),\ n \geq 1$

Usando sustitución y generalización:

$f(n) = 1 + f(n - 1)$

$f(n) = 1 + 1 + f(n - 2) = 2 + f(n - 2)$

$f(n) = 1 + 1 + 1 + f(n - 3) = 3 + f(n - 3)$

…

$f(n) = i + f(n - i)$

Usando la condición inicial: $n - i = 1 \rightarrow i = n - 1$

Por lo tanto: $f(n) = n - 1 + f(1) = n - 1 + 0 = n - 1$

$$\boldsymbol{f(n) = n - 1}$$

8.6.2 Árbol de recursividad.

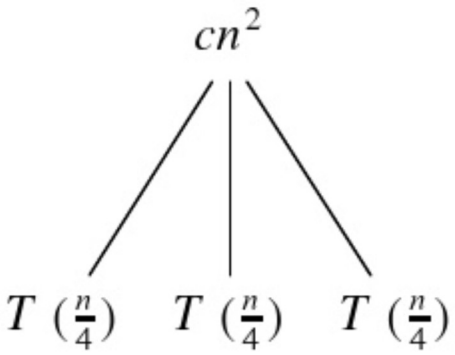
En un ´árbol recursivo, cada nodo representa el costo de un subproblema dentro del proceso recursivo. Se suma el costo de cada nivel del árbol para obtener el costo por nivel y luego se suman los costos por niveles.

Ejemplo –usando la siguiente recurrencia:

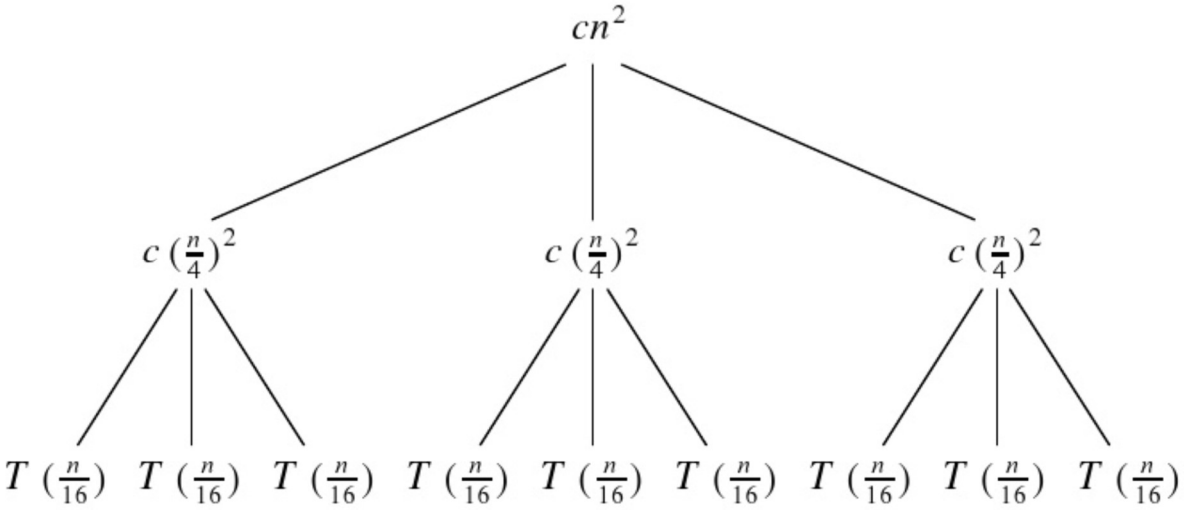
$$T(1) = c_1$$

$$T(n) = 3T(n=4) + cn^2$$

Hacemos un árbol que en su raíz tiene el costo para el *n* inicial, con ramas indicando cada una de las llamadas recursivas.



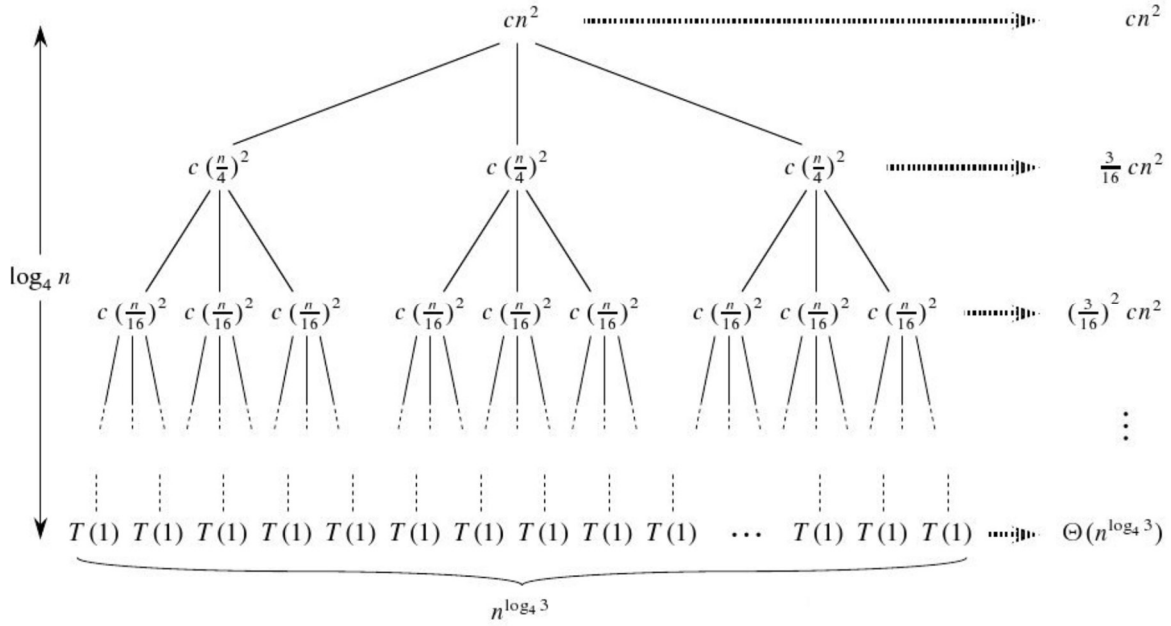
Las ramas correspondientes a las llamadas recursivas se expanden, ahora tienen el costo correspondiente a cada llamada y generan nuevas hojas.



Se sigue expandiendo el árbol hasta llegar a un caso base.

En cada nivel se suma el total de operaciones por nivel.

El costo total es la suma de todos los niveles.



La suma obtenida se manipula algebraicamente para llegar al resultado:

$$T(n) = 3T(n=4) + cn^2 \in O(n^2)$$

8.6.3 Expansión de recurrencias.

Método algebraico equivalente al árbol de recursividad.

Ejemplo – Factorial de un número.

Factorial(n)

1 if n = 0 then

2 Factorial ← 1;

3 else

4 Factorial ← n × Factorial(n − 1);

endif

Asumiendo que la multiplicación es de orden constante, uno puede fácilmente expandir la recurrencia.

El método de expansión de recurrencia consiste en utilizar la recurrencia misma para sustituir *T(n)* por su valor en términos de *T(m)*, donde $m < n$, iterativamente hasta encontrar una condición de frontera *T(r)* que tendría un valor constante. En la ecuación recurrente se ponen todos los términos con *T* en el lado izquierdo de la ecuación y se realiza la expansión. Al sumar todas las ecuaciones obtenemos una expresión para *f (n)* en término de *n* y algunas constantes. A esta expresión se le denomina forma cerrada para *T(n)*.

Expansión de recurrencia (para operación multiplicación ***O(1)***)

$$T(n) - T(n-1) = 1$$

$$T(n-1) - T(n-2) = 1$$

$$T(n-2) - T(n-3) = 1$$

...

$$T(2) - T(1) = 1$$

$$T(1) - T(0) = 1$$

$$T(0) = 0$$

Sumando todas las ecuaciones, nos queda: ***f(n)* = 1 + 1 + 1 + 1... + 1 = *n***

8.6.4 Método de la ecuación característica.

Resolución de recurrencias mediante el método de la ecuación característica:

1. Se obtiene la ecuación característica asociada a la recurrencia que describe el tiempo de ejecución *T(n)*.

2. Se calculan las k raíces del polinomio característico.

3. Se obtiene la solución a partir de las raíces del polinomio característico.

4. Se determinan las constantes a partir de las k condiciones iniciales.

Ejemplo – Secuencia Fibonacci.

$f(0) = 0$

$f(1) = 1$

$f(n) = f(n - 1) + f(n - 2)$

La recurrencia es $f(n) - f(n - 1) - f(n - 2) = 0$

Su polinomio característico es $x^2 - x - 1$

Las raíces son $r1 = (1+\sqrt{5})/2$ y $r2 = (1-\sqrt{5})/2$

La solución general es $f(n) = c_1 r_1^n + c_2 r_2^n$

$$0 = f(0) = c_1 + c_2 \qquad 1 = f(1) = r_1 c_1 + r_2 c_2$$

$$\left.\begin{array}{l} c_1 + c_2 = 0 \\ r_1 c_1 + r_2 c_2 = 1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} c_1 = 1 / \sqrt{5} \\ c_2 = -1 / \sqrt{5} \end{array}\right.$$

Por lo tanto, $\boldsymbol{f(n) = 1 / \sqrt{5} \left[\left(1 + \sqrt{5} / 2 \right)^n - \left(1 - \sqrt{5} / 2 \right)^n \right]}$

9 – Ejercicios.

BÚSQUEDA INTERNA.

1. Escriba un programa para **búsqueda secuencial** en un arreglo de N números enteros generados aleatoriamente. Desplegando la posición del elemento si éste se encontró, en caso contrario desplegar el mensaje adecuado.
2. Dado un arreglo que contiene los nombres de N maestros, escriba un programa que localice un nombre dado en el arreglo. Si lo encuentra debe desplegar como resultado la posición en la que lo encontró. En caso contrario debe desplegar un mensaje adecuado. Aplique el método de **búsqueda secuencial desordenado**.
3. Dado un arreglo de N componentes que contienen la siguiente información:
   - Nombre del alumno.
   - Promedio.
   - Número de materias aprobadas.

   Escriba un programa que lea el nombre de un alumno y regrese como resultado el promedio y el numero de materias aprobadas por dicho alumno. Si el nombre dado no esta en el arreglo, despliegue un mensaje adecuado.
   Considere que el arreglo está ordenado y aplique la **búsqueda secuencial ordenado**.
4. Escriba un programa para **búsqueda secuencial en listas enlazadas desordenadas** que contenga el nombre de N personas. Si el elemento se encuentra en la lista, indique el número de nodo en el cual se encontró. En caso contrario, despliegue el mensaje adecuado.
5. Escriba un programa para **búsqueda secuencial en listas enlazadas ordenadas** que contenga N números de control ordenados de manera descendente. Si el elemento se encuentra en la lista, indique el número de nodo en el cual se encontró. En caso contrario, despliegue el mensaje adecuado.
6. Escribir dos programas de **búsqueda binaria** para arreglos de N números enteros generados aleatoriamente, ordenarlos:
   a) De manera ascendente.
   b) De manera descendente.

   Seleccionar un método de ordenamiento diferente para cada uno de los programas. Si localiza el número, debe desplegar la posición en la que lo encontró. En caso contrario debe desplegar un mensaje adecuado.

7. Resuelva el problema 3 utilizando el algoritmo de **búsqueda binaria**. Seleccionar el método de ordenamiento para el programa.
8. Dado que se quiere almacenar los registros con claves: 23, 42, 5, 66, 14, 43, 59, 81, 37, 49, 28, 55, 94, 80 y 64 en un arreglo de 20 elementos, defina una **función hash** que distribuya los registros en el arreglo. Si hubiera colisiones, resuélvalas aplicando el método de la **prueba lineal**.
   Aplique el método de transformación de claves para buscar los registros, si localiza el número, debe desplegar la posición en la que lo encontró. En caso contrario debe desplegar un mensaje adecuado.
9. De un grupo de N alumnos se tienen los siguientes datos:
   - Matrícula: valor entero comprendido entre 1000 y 4999
   - Nombre: cadena de caracteres.
   - Dirección: cadena de caracteres.

   El campo clave es la matrícula. Los N registros han sido almacenados en un arreglo, aplicando la siguiente **función hash:**

   H(clave) = dígitos centrales(clave^2) + 1

   Las colisiones han sido tratadas con el método de **doble dirección hash**.
   Escriba un programa utilizando el método de transformación de claves que lea la matrícula de un alumno y regrese como resultado el nombre y dirección del mismo. En caso de no encontrarlo despliegue un mensaje adecuado.
10. Se quiere almacenar en un arreglo los siguientes datos de N personas:
    - Clave del contribuyente: alfanumérico de longitud 6.
    - Nombre: cadena de caracteres.
    - Dirección: cadena de caracteres.
    - Saldo: número real.

    Defina una **función hash** para almacenar los datos mencionados y utilice el método de **encadenamiento** para resolver las colisiones.
    Aplique en el programa el método de transformación de claves que lea la clave del contribuyente y regrese como resultado todos los datos del mismo. En caso de no encontrarlo despliegue un mensaje adecuado.
11. Dados los 12 signos del zodiaco: capricornio, acuario, piscis, aries, tauro, géminis, cáncer, leo, virgo, libra, escorpión y sagitario.
    a) Escriba un programa para almacenarlos en una estructura árbol.
    b) Aplique el método **árbol binario de búsqueda** para los signos del zodiaco almacenados y despliegue un mensaje adecuado.
12. Almacene el nombre de los planetas en el sistema solar de acuerdo al siguiente orden: Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano, Neptuno y Plutón, en una estructura árbol. Posteriormente utilice el método **árbol binario de búsqueda** para los nombres de planetas almacenados y despliegue un mensaje adecuado.

BÚSQUEDA EXTERNA.

13. Se han almacenado en un archivo secuencial los siguientes datos de los empleados de una empresa:
    - ✓ Nombre.
    - ✓ Registro Federal de Contribuyentes.
    - ✓ Fecha de ingreso.
    - ✓ Sueldo.

    Escriba un programa para **buscar secuencialmente** y desplegar los datos de un empleado, dando su nombre como entrada.
14. Se desea crear un archivo secuencial con la información sobre los pinos mexicanos que existen. Cada registro contiene los siguientes datos:
    - ❖ Nombre del pino.
    - ❖ Tipo de hojas.
    - ❖ Tipo de cono.
    - ❖ Lugar donde se encuentra.

    Escriba un programa para **buscar secuencialmente** y desplegar los datos de un pino, dando su nombre como entrada.
15. Se tiene un archivo con registros que almacenan información sobre clientes de una sucursal bancaria. Los datos que se manejan por cada cliente son los siguientes:
    - Nombre del titular.
    - Número de cuenta.
    - Tipo de cuenta.
    - Movimientos de la cuenta
    - Saldo

    Escriba un programa aplicando la **búsqueda binaria** y desplegar los datos del cliente, dando su nombre como entrada.
16. Defina una **función hash** que permita almacenar y posteriormente recuperar los elementos de la tabla periódica de los elementos químicos en un archivo. La clave estará dada por el nombre de los elementos. Resuelva las colisiones utilizando un área independiente para almacenar los elementos colisionados.
17. Utilice la función hash definida en el ejercicio anterior para insertar y eliminar los elementos que se indican a continuación:
    Insertar: sodio, oro, osmio, litio, boro, cobre, plata y radio.
    Eliminar: oro, osmio, boro, cobre y plata.
    El número de cubetas es dos (N = 2) y cada cubeta tiene dos registros. La densidad de ocupación permitida es 80%; en caso de superar este porcentaje se aplicarán **expansiones totales**.
    a) Dibuje un esquema de la organización después de insertar los elementos osmio y plata; y luego de eliminar oro, boro y plata.

b) Mencione que claves originaron que el numero de cubetas se expandiera o redujera.

18. Sea N = 2 el número de cubetas. Cada cubeta tiene dos registros y se establece una densidad de ocupación permitida del 85%. Una vez superada esta densidad, se aplicarán **expansiones parciales**.
H(clave) = clave MOD N
Claves a insertar: 36 11 48 06 75 65 38 88 23 14 12
a) Dibuje un esquema de la organización después de insertar los elementos: 06 38 23 y 12.
b) Diga que claves originaron que el número de cubetas se expandiera.

10 – Bibliografía.

Cairó O., Guardati S. 2006. Estructura de Datos. (3ra. ed.) México. Editorial McGraw-Hill. ISBN: 970-10-5908-5.

Joyanes L., Zahonero I. 2004. Algoritmos y Estructuras de Datos. (1era ed.) España. Editorial McGraw-Hill. ISBN: 978-84-481-4077-9

Guardati S. 2007. Estructura de Datos Orientada a Objetos: Algoritmos con C++. (1era ed.) México. Editorial Pearson Prentice Hall. ISBN: 970-26-0792-2.

Joyanes L., Sánchez L. Zahonero I. 2007. Estructura de Datos en C++. (1era ed.) España. Editorial: McGraw-Hill. ISBN: 978-84-481-5645-9

Joyanes L., Zahonero I. 2008. Estructura de Datos en Java. (1era ed.) España. Editorial: McGraw-Hill. ISBN: 978-84-481-5631-2.

Gómez M., Cervantes J. 2014. Introducción al Análisis y al Diseño de Algoritmos (1era ed.) México. Editorial: UAM, Unidad Cuajimalpa, División de Ciencias Naturales e Ingeniería, Departamento de Matemáticas Aplicadas y Sistemas. ISBN: 978-607-28-0225-4

https://profile.es/blog/que-es-un-algoritmo-informatico/#Partes-de-un-algoritmo-informatico

https://github.com/ServioTRC/AlgoritmosComputacionales/blob/master/Baase%20Sara%20Y%20Van%20Gelder%20Allen%20%20Algoritmos%20Computacionales%20(3%20Ed)

https://es.wikipedia.org/wiki/Algoritmo de ordenamiento

https://www.udb.edu.sv/udbfiles/recursosguias/informaticaingenieria/programación-iv/2019/ii/guia-3

https://platzi.com/tutoriales/1832-ordenamiento/9229-algoritmos-de-ordenamiento

https://EficienciadeAlgoritmosOrdenamientoyBúsqueda:httpswww.mheducation.esbcvguidecapitulo8448198441

https://www.freecodecamp.org/espanol/news/la-complejidad-de-los-algoritmos-simples-y-las-estructuras-de-datos/

https://www.cs.buap.mx/~asanchez/Ada/introduccionADA.pdf

http://di002.edv.uniovi.es/~dani/asignaturas/transparencias-leccion13.PDF

http://www.lcc.uma.es/~av/Libro/CAP1.pdf

https://www.cs.buap.mx/~asanchez/Ada/introduccionADA.pdf

https://es.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/measuring-an-algorithms-efficiency

https://ocw.bib.upct.es/pluginfile.php/7820/mod_resource/content/1/085_112_capitulo_6_RECURRENCIA.pdf

https://ocw.ehu.eus/pluginfile.php/40172/mod_resource/content/1/diseno_alg/contenidos/resolucion-de-recurrencias.pdf

https://ccc.inaoep.mx/~carranza/docs/alg/s3.pdf

https://www.kramirez.net/Discretas/Material/Presentaciones/RelacionesRecurrencia_MSG.pdf