

Tema 8:

Algoritmos de ordenación y búsqueda

Objetivos: en este tema se presentan algoritmos que permiten buscar un elemento dentro de una colección y ordenar una colección en base a algún criterio (el valor de un número, orden alfabético...). Ambas operaciones son empleadas profusamente en computación, de ahí su gran importancia, y se apoyan la una es la otra, por lo que las hemos agrupado bajo un mismo tema.



CEU

Índice

Índice	2
1 Introducción.....	4
2 Complejidad computacional y algoritmos	5
2.1 Complejidad computacional de las estructuras básicas de programación.....	7
2.1.1 Sentencias sencillas	7
2.1.2 Secuencia de sentencias.....	8
2.1.3 Selección.....	8
2.1.4 Bucles	8
2.2 Limitaciones del análisis O.....	10
3 Algoritmos de búsqueda	10
3.1 Búsqueda secuencial.....	10
3.1.1 Búsqueda sin centinela	11
3.1.2 Búsqueda con centinela	12
3.2 Búsqueda binaria o dicotómica	13
4 Ordenación	15
4.1 Método de la burbuja.....	15
4.1.1 Burbuja mejorada	17
4.2 Método de selección	18

4.3	Método de inserción	20
4.4	Quicksort	22
4.4.1	Análisis del rendimiento de Quicksort	25
5	Ejercicios	27

1 Introducción

La ordenación es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cálculos que tiene que realizar un programa son más eficientes si los datos sobre los que operan están ordenados. Uno de los tipos de cálculo que más se benefician de operar sobre un conjunto de datos ordenados es la búsqueda de un dato: encontrar el número de teléfono de una persona en un listín telefónico es una tarea muy simple y rápida si conocemos su nombre, ya que los listines telefónicos se encuentran ordenados alfabéticamente. Encontrar a que usuario corresponde un número de teléfono dado, sin embargo, es una tarea prácticamente inabordable. Valga esto a modo de ejemplo de cómo el disponer de una colección de datos ordenados simplifica la búsqueda de información entre ellos, de ahí la importancia de las tareas de ordenación y búsqueda y la relación que existe entre ellas.

Las operaciones de ordenación y búsqueda suelen clasificarse en dos tipos:

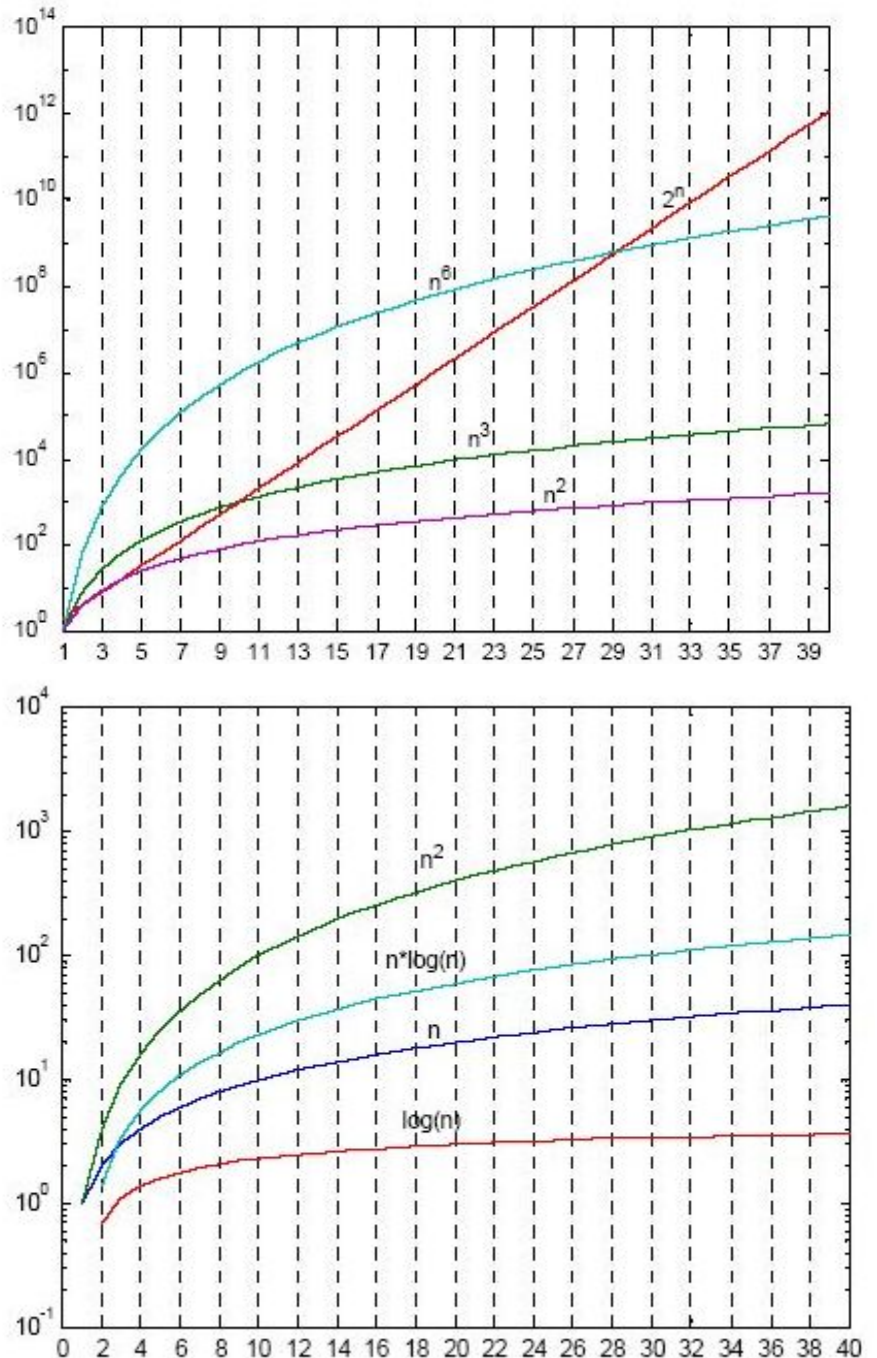
- **Internas:** todos los datos a procesar residen en la memoria principal.
- **Externas:** los otros procesos residen en un dispositivo de almacenamiento masivo, de acceso lento. Su volumen es demasiado elevado para trasladarlos todos a la memoria principal, lo cual fuerza realizar numerosos accesos al dispositivo de almacenamiento masivo.

En este tema nos centraremos en las operaciones internas. Un factor clave en cualquier algoritmo de búsqueda u ordenación es su complejidad computacional, y el cómo esta depende del número de datos a procesar. Habitualmente, cuando nos enfrentamos una tarea de ordenación o búsqueda con un ordenador el volumen de datos de entrada será enorme y es importante contar con algoritmos que no degraden considerablemente su rendimiento con el tamaño del conjunto de datos. Por ello, antes de abordar algoritmos de ordenación y búsqueda realizaremos una pequeña introducción al cálculo de la complejidad computacional de un algoritmo.

Para estudiar la complejidad computacional de los algoritmos se toma por cierto el **principio de invarianza**: dos implementaciones distintas del mismo algoritmo no difieren en su eficiencia más que en una constante multiplicativa; esto es, si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ unidades de tiempo, donde n es el tamaño del vector de entrada, entonces existe un $c > 0$ tal que $t_1(n) < c \cdot t_2(n)$. Este principio permite concluir que un cambio en la máquina donde se ejecuta un algoritmo proporciona una mejora

de un factor constante, mientras que las mejoras dependientes del número de datos que procesa el algoritmo deberán de venir dadas por cambios en el propio algoritmo.

Hay dependencias en el tamaño n del volumen de datos a procesar muy frecuentes: tiempo logarítmico ($c \cdot \log(n)$), tiempo lineal ($c \cdot n$), tiempo cuadrático ($c \cdot n^2$), tiempo polinomial ($c \cdot n^k$), y tiempo exponencial (c^n). Estas dependencias están ordenadas de menor a mayor, siempre que se consideren valores de n suficientemente grandes.



El cálculo de la eficiencia de un algoritmo se basa en contar el número de operaciones elementales que realiza. Por operación elemental se entiende operación cuyo tiempo de ejecución es constante y depende únicamente de la implementación como, por ejemplo, sumas, restas, productos, divisiones, módulo, operaciones lógicas, operaciones de comparación, etcétera. Estas operaciones no deben depender del volumen de datos manejados por el algoritmo, y sólo nos interesa el número de operaciones, no cuánto consume cada una de ellas. La diferencia en los tiempos de ejecución entre las distintas operaciones quedan incluidos en la constante multiplicativa.

Para representar la eficiencia de un algoritmo suele emplearse la notación "del orden de" ($O(\dots)$). Diremos que una función $t(n)$ está en el orden de $f(n)$ si existe una constante c y un umbral n_0 tal es que:

$$\forall n \geq n_0 \quad t(n) \leq c \cdot f(n)$$

Textualmente esto se representa por "t es del orden de f" ($t=O(f)$). Esta frase indica que si t es el tiempo de la implementación de un algoritmo podemos considerar que dicho tiempo está en el orden de f . Además, por el principio de invarianza, cualquier implementación de dicho algoritmo también será de orden f . La notación $t=O(f)$ suele usarse aunque $f(n)$ sea menor que cero para algún n , o incluso cuando no esté definida para algún valor de n ; lo importante es el comportamiento de $f(n)$ para valores grandes de n .

2.1 Complejidad computacional de las estructuras básicas de programación

En esta sección estudiaremos cuál es la complejidad computacional de las estructuras básicas que permiten construir algoritmos.

2.1.1 Sentencias sencillas

Nos referimos a las sentencias de asignación, comparación, operaciones lógicas y matemáticas. Este tipo de sentencias tiene un tiempo de ejecución constante que no depende del tamaño del conjunto de datos de entrada, siendo su complejidad $O(1)$.

2.1.2 Secuencia de sentencias

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales; el caso de una secuencia de sentencias sencillas la complejidad será: $O(1) + O(1) + \dots + O(1) = k * O(1) = O(1)$.

2.1.3 Selección

La evaluación de la condición suele ser de $O(1)$, complejidad a sumar con la mayor complejidad computacional posible de las distintas ramas de ejecución, bien en la rama IF, o bien en la rama ELSE. En decisiones múltiples (ELSE IF, SWITCH CASE), se tomará la rama cuya complejidad computacional es superior.

2.1.4 Bucles

Los bucles son la estructura de control de flujo que suele determinar la complejidad computacional del algoritmo, ya que en ellos se realiza un mayor número de operaciones.

En los bucles con contador podemos distinguir dos casos: que el tamaño del conjunto de datos n forme parte de los límites del bucle o que no. Si la condición de salida del bucle es independiente de n entonces la repetición sólo introduce una constante multiplicativa:

```
| for (int i= 0; i < K; i++) { O (1) }
```

la complejidad será $K * O(1) = O(1)$.

Cuando el número de iteraciones a realizar depende del tamaño de datos a procesar la complejidad computacional del bucle incrementará con el tamaño de los datos de entrada:

```
| for (int i= 0; i < n; i++) { O (1) }
```

la complejidad será $n * O(1) = O(n)$.

```
| for (int i= 0; i < n; i++) {
    |     for (int j= 0; j < n; j++) {
    |         O (1)
    |     }
    | }
```


tendremos $n * n * O(1) = O(n^2)$.

```
for (int i= 0; i < n; i++) {
    for (int j= 0; j < i; j++) {
        O(1)
    }
}
```

En este caso el bucle exterior se realiza n veces, mientras que el interior se realiza 1, 2, 3, ..., n veces respectivamente. En total $1 + 2 + 3 + \dots + n = n*(1+n)/2 = O(n^2)$.

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
c= 1;
while (c < n) {
    O(1)
    c= 2*c;
}
```

El valor inicial de "c" es 1, siendo 2^k al cabo de k iteraciones. El número de iteraciones es tal que $2^k \geq n \rightarrow k = \log_2(n)$ (el entero inmediato superior) y, por tanto, la complejidad del bucle es $O(\log n)$.

```
c= n;
while (c > 1) {
    O(1)
    c= c / 2;
}
```

Razonando de un modo similar al caso anterior, obtenemos un orden $O(\log n)$.

```
for (int i= 0; i < n; i++) {
    c= i;
    while (c > 0) {
        O(1)
        c= c/2;
    }
}
```

En este caso tenemos un bucle interno de orden $O(\log n)$ que se ejecuta n veces, luego el conjunto es de orden $O(n \log n)$.

2.2 Limitaciones del análisis O

El análisis O no es adecuado para pequeñas cantidades de datos, ya que las simplificaciones que se realizan en el cálculo de la complejidad del algoritmo se apoyan en la suposición de que n es un número grande.

Las constantes grandes pueden entrar a juego en los algoritmos excesivamente complejos; en esto también influye el hecho de que el análisis no tiene en cuenta que la constante asociada a una operación simple, con un acceso memoria, es muy inferior a la constante asociada a un acceso a disco.

Por último, el análisis supone que contamos con una memoria infinita y no mide el impacto de este recurso en la eficiencia del algoritmo.

3 Algoritmos de búsqueda

Un problema de búsqueda puede enunciarse del siguiente modo: dado un conjunto de elementos CB (Conjunto Búsqueda) de un cierto tipo determinar si un elemento ("dato") se encuentra en el conjunto o no.

Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos: si se encuentran ordenados o si se ignora su disposición o se sabe que están al azar. También depende de si los datos a ordenar pueden ser accedidos de modo aleatorio o deben ser accedidos de modo secuencial.

3.1 Búsqueda secuencial

Es el algoritmo de búsqueda más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio. Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

Supondremos que los datos están almacenados en un array y se asumirá acceso secuencial. Se pueden considerar dos variantes del método: con y sin centinela.

3.1.1 Búsqueda sin centinela

El algoritmo simplemente recorre el array comparando cada elemento con el dato que se está buscando:

```
/*
*ejemplo8_1.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int i, dato;

    srand((unsigned int)time(NULL));
    for(i = 0; i < TAM; i++)
        CB[i] = (int)(rand() % 100);
    imprimeCB(CB);
    dato = (int)(rand() % 100);
    printf("Dato a buscar %d\n",dato);

    i=0;
    while ((CB[i]!=dato) && (i<TAM))
        i++;

    if (CB[i]==dato) printf("Posicion %d\n",i);
    else printf("Elemento no esta en el array");
}
```

La complejidad del algoritmo medida en número de iteraciones en el mejor caso será 1, y se corresponderá con aquella situación en la cual el elemento a buscar está en la primera posición del array. El peor caso la complejidad será TAM y sucederá cuando el elemento

buscar esté en la última posición del array. El promedio será $(TAM+1)/2$. El orden de complejidad es lineal ($O(TAM)$). Cada iteración necesita una suma, dos comparaciones y un AND lógico.

Consideremos un ejemplo: buscar 8 en el siguiente conjunto de datos:

```
0 9 5 5 8 4 6 0 4 9
- - - - -
Posicion 5
```

3.1.2 Búsqueda con centinela

Si tuviésemos la seguridad de que el elemento buscado está en el conjunto, nos evitaría controlar si se supera el límite superior. Para tener esa certeza, se almacena un elemento adicional (centinela), que coincidirá con el elemento buscado y que se situará en la última posición del array de datos. De esta forma se asegura que encontraremos el elemento buscado.

```
/*
*ejemplo8_2.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM+1];
    int i, dato;

    srand((unsigned int)time(NULL));
    for(i = 0; i < TAM; i++)
        CB[i] = (int)(rand() % 100);
    imprimeCB(CB);
    dato = (int)(rand() % 100);
    CB[i] = dato;
```

```

printf("Dato a buscar %d\n",dato);

i=0;
while (CB[i]!=dato)
    i++;

if (CB[i]==dato) printf("Posicion %d\n",i);
else printf("Elemento no esta en el array");
}

```

Ahora sólo se realiza una suma y una única comparación (se ahorra una comparación y un AND). El algoritmo es más eficiente.

3.2 *Búsqueda binaria o dicotómica*

Es un método muy eficiente, pero tiene varios prerequisites:

- El conjunto de búsqueda está ordenado.
- Se dispone de acceso aleatorio.

Este algoritmo compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior, respectivamente, al elemento central, y así sucesivamente.

```

/*
*ejemplo8_3.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int ini=0,fin=TAM-1,mitad,dato,i;

    srand((unsigned int)time(NULL));

```

```

for(i = 0; i < TAM; i++)
    CB[i] = (int)(rand() % 100);
imprimeCB(CB);
dato = (int)(rand() % 100);
CB[i] = dato;
printf("Dato a buscar %d\n",dato);

mitad=(ini+fin)/2;
while ((ini<=fin)&&(CB[mitad]!=dato))
{
    if (dato < CB[mitad])
        fin=mitad-1;
    else ini=mitad+1;
    mitad=(ini+fin)/2;
}

if (dato==CB[mitad]) printf("Posicion %d\n", mitad);
else printf("Elemento no esta en el array");
getch();
}

```

En el caso más favorable (el dato es el elemento mitad) se realiza 1 iteración. En el caso más desfavorable, el número de iteraciones es el menor entero K que verifica $2^K \geq TAM$. Esto es, el orden de complejidad es $O(\log_2(TAM))$.

Veamos un ejemplo; tenemos que buscar el número 8 en la siguiente conjunto de datos:

1 2 2 2 4 4 5 6 6 9	(ini,fin)	mitad
-	(0 ,9)	4
1 2 2 2 4 4 5 6 6 9	-	(5 ,9)
-	(8 ,9)	8
1 2 2 2 4 4 5 6 6 9	-	(9 ,9)
-	(9 ,8)	9
No se encontro....	(9 ,8)	

Busquemos el mismo valor en otro conjunto de datos:

0 1 1 2 2 6 7 7 8 9	(ini,fin)	mitad
-	(0 ,9)	4
0 1 1 2 2 6 7 7 8 9	-	(5 ,9)
-	(8 ,9)	8
Posicion 8		

A continuación mostramos una tabla que compara la eficiencia de la búsqueda binaria con la búsqueda secuencial. Se comprueba como para conjuntos de datos grandes la búsqueda binaria sigue siendo eficiente mientras que la secuencial se va degradando.

Número de elementos examinados (peor caso)		
Tamaño del array	Búsqueda binaria	Búsqueda secuencial
1	1	1
10	4	10
1000	11	1000
5000	14	5000
100000	18	100000
1000000	21	1000000

4 Ordenación

La ordenación o clasificación es un proceso de organizar un conjunto de datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o el orden alfabético para datos compuestos por caracteres. Los algoritmos de ordenación permutan los elementos del conjunto de datos hasta conseguir dicho orden. Para ello se basan en dos operaciones básicas: la comparación y el intercambio.

Existen muchos algoritmos de ordenación con diferentes ventajas e inconvenientes; en este tema veremos los más comunes.

4.1 Método de la burbuja

Se basa en recorrer el array ("realizar una pasada") un cierto número de veces, comparando pares de valores que ocupan posiciones adyacentes (0-1,1-2, ...). Si ambos datos no están ordenados, se intercambian. Esta operación se repite $n-1$ veces, siendo n el tamaño del conjunto de datos de entrada. Al final de la última pasada el elemento mayor estará en la última posición; en la segunda, el segundo elemento llegará a la penúltima, y así sucesivamente.

Su nombre se debe a que el elemento cuyo valor es mayor sube a la posición final del array, al igual que las burbujas de aire en un depósito suben a la parte superior. Para ello debe realizar un recorrido paso a paso desde su posición inicial hasta la posición final del array.

```

/*
*ejemplo8_4.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int e, i, auxiliar;

    srand((unsigned int)time(NULL));

    for(e = 0; e < TAM; e++)
        CB[e] = (int)(rand() % 100);

    printf( "Antes de ordenar\-----\n");
    imprimeCB(CB);

    for(e = 0; e < TAM; e++)
        for(i = 0; i < TAM-1-e; i++)
            if(CB[i] > CB[i+1]) {
                auxiliar = CB[i+1];
                CB[i+1] = CB[i];
                CB[i] = auxiliar;
            }

    printf( "\nDespués de ordenar\n-----\n");
    imprimeCB(CB);
}

```

Veamos un ejemplo:

| 4 0 6 5 7 7 0 2 9 7


```

0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9

```

La complejidad computacional de éste algoritmo $O(TAM^2)$.

4.1.1 Burbuja mejorada

Existe una forma muy obvia para mejorar el algoritmo de la burbuja. Basta con tener en cuenta la posibilidad de que el conjunto esté ordenado en algún paso intermedio. Si el bucle interno no necesita realizar ningún intercambio en alguna pasada, el conjunto estará ya ordenado.

```

/*
*ejemplo8_5.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int e, i, auxiliar, intercambio;

    srand((unsigned int)time(NULL));

    for(e = 0; e < TAM; e++)
        CB[e] = (int)(rand() % 100);

    printf( "Antes de ordenar\n-----\n");
    imprimeCB(CB);

```

```

for(e = 0; e < TAM; e++){
    intercambio = 0;
    for(i = 0; i < TAM-1-e; i++){
        if(CB[i] > CB[i+1]) {
            auxiliar = CB[i+1];
            CB[i+1] = CB[i];
            CB[i] = auxiliar;
            intercambio =1;
        }
    }
    if (intercambio==0){
        printf( "\nPara en la iteración %d\n",e);
        break;
    }
}

printf( "\nDespués de ordenar\n-----\n");
imprimeCB(CB);
}

```

Veamos el mismo ejemplo que el caso anterior procesado mediante el algoritmo mejorado:

```

4 0 6 5 7 7 0 2 9 7
0 4 5 6 7 0 2 7 7 9
0 4 5 6 0 2 7 7 7 9
0 4 5 0 2 6 7 7 7 9
0 4 0 2 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9
0 0 2 4 5 6 7 7 7 9

```

En el mejor caso (si ya está ordenado) realiza TAM-1 comparaciones. En el peor caso (el elemento menor estaba situado al fin del array) se necesitan las mismas pasadas que antes y el orden es TAM². En el caso medio el orden es proporcional a TAM²/2. Obsérvese que éste algoritmo tiene una complejidad computacional en el peor caso igual al de la burbuja simple, aunque en término medio es aproximadamente dos veces más eficiente (requiere la mitad de tiempo para ejecutarse).

4.2 Método de selección

Este método considera que el array está formado por 2 partes: una parte ordenada (la izquierda) que estará vacía al principio y al final comprende todo el array; y una parte desordenada (la derecha) que al principio comprende todo el array y al final estará vacía. El algoritmo toma elementos de la parte derecha y los coloca en la parte izquierda;

empieza por el menor elemento de la parte desordenada y lo intercambia con el que ocupa su posición en la parte ordenada. Así, en la primera iteración se busca el menor elemento y se intercambia con el que ocupa la posición 0; en la segunda, se busca el menor elemento entre la posición 1 y el final y se intercambia con el elemento en la posición 1. De esta manera las dos primeras posiciones del array están ordenadas y contienen los dos elementos menores dentro del array. Este proceso continúa hasta ordenar todos los elementos del array.

```
/*
*ejemplo8_6.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int e,i,PosMenor,aux;

    srand((unsigned int)time(NULL));

    for(e = 0; e < TAM; e++)
        CB[e] = (int)(rand() % 100);

    printf( "Antes de ordenar\n-----\n");
    imprimeCB(CB);

    for (e=0; e<(TAM-1) ; e++) {
        PosMenor=e;
        for (i=e+1;i<TAM;i++)
            if (CB[i]<CB[PosMenor])
                PosMenor=i;

        aux=CB[e];
        CB[e]=CB[PosMenor];
        CB[PosMenor]=aux;
    }
}
```

```

    printf( "\nDespués de ordenar\n-----\n" );
    imprimeCB(CB);
}

```

En cada pasada se coloca un elemento en su lugar, y la variable *e* marca donde empezar la búsqueda en la parte desordenada, que será secuencial si no tenemos más información. La búsqueda del siguiente elemento menor comienza suponiendo que dicho elemento es *e*. Se comprueba la hipótesis comparándolo con cada uno de los restantes. Si se encuentra uno menor, se intercambia.

Veamos un ejemplo de la ejecución de éste algoritmo:

	1	6	8	5	9	3	0	3	7
0		6	8	5	9	3	1	3	7
0	1		8	5	9	3	6	3	7
0	1	3		5	9	8	6	3	7
0	1	3	3		9	8	6	5	7
0	1	3	3	5		8	6	9	7
0	1	3	3	5	6		8	9	7
0	1	3	3	5	6	7		9	8
0	1	3	3	5	6	7	8		9

El número de comparaciones que realiza este algoritmo es independiente de la ordenación inicial. El bucle interno hace TAM-1 comparaciones la primera vez, TAM-2 la segunda,..., y 1 la última. El bucle externo hace TAM-1 búsquedas. El total de comparaciones es $(TAM^2 - TAM)/2$. Por tanto el orden de complejidad es cuadrático ($O(TAM^2)$).

4.3 Método de inserción

Se utiliza un método similar al anterior, tomando un elemento de la parte no ordenada para colocarlo en su lugar en la parte ordenada. El primer elemento del array (CB[0]) se considerado ordenado (la lista inicial consta de un elemento). A continuación se inserta el segundo elemento (CB[1]) en la posición correcta (delante o detrás de CB[0]) dependiendo de que sea menor o mayor que CB[0]. Repetimos esta operación sucesivamente de tal modo que se va colocando cada elemento en la posición correcta. El proceso se repetirá TAM-1 veces.

Para colocar el dato en su lugar, se debe encontrar la posición que le corresponde en la parte ordenada y hacerle un hueco de forma que se pueda insertar. Para encontrar la posición se puede hacer una búsqueda secuencial desde el principio del conjunto hasta

encontrar un elemento mayor que el dado. Para hacer el hueco hay que desplazar los elementos pertinentes una posición a la derecha.

```

/*
*ejemplo8_7.c
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int e,i,k,temp;

    srand((unsigned int)time(NULL));

    for(e = 0; e < TAM; e++)
        CB[e] = (int)(rand() % 100);

    printf( "Antes de ordenar\n-----\n");
    imprimeCB(CB);

    for (e=1;e<TAM;e++){
        temp=CB[e];
        i=0;
        while (CB[i]<=temp)
            i++;
        if (i<e)
        {
            for (k=e;k>i;k--)
                CB[k]=CB[k-1];
            CB[i]=temp;
        }
    }

    printf( "\nDespués de ordenar\n-----\n");
    imprimeCB(CB);
}

```

Veamos un ejemplo de la ejecución de este algoritmo:

4	1	6	9	1	0	2	9	8	4	
1	4		6	9	1	0	2	9	8	4
1	4	6		9	1	0	2	9	8	4
1	4	6	9		1	0	2	9	8	4
1	1	4	6	9		0	2	9	8	4
0	1	1	4	6	9		2	9	8	4
0	1	1	2	4	6	9		9	8	4
0	1	1	2	4	6	9	9		8	4
0	1	1	2	4	6	8	9	9		4
0	1	1	2	4	4	6	8	9	9	

El orden de complejidad de este algoritmo es cuadrático ($O(TAM^2)$).

4.4 Quicksort

El método de ordenación rápida (Quicksort) para ordenar los elementos del array se basa en el hecho de que es más rápido y fácil ordenar dos listas pequeñas que una lista grande. Su nombre se debe a que este método, en general, puede ordenar una lista de datos mucho más rápido que cualquier otro método de la bibliografía.

El método se basa en la estrategia típica de "divide y vencerás". El array a ordenar se divide en dos partes: una contendrá todos los valores menores o iguales a un cierto valor (que se suele denominar pivote) y la otra con los valores mayores que dicho valor. El primer paso es dividir el array original en dos subarrays y un valor que sirve de separación, esto es, el pivote. Así, el array se dividirá en tres partes:

- La parte izquierda, que contendrá valores inferiores o iguales al pivote.
- El pivote.
- La parte derecha, que contiene valores superiores o iguales al pivote.

Inicialmente, las partes izquierda y derecha no estarán ordenadas, excepto en el caso de que estén compuestas por un único elemento. Consideremos, por ejemplo, la lista de valores:

28	21	37	23	19	14	26
----	----	----	----	----	----	----

elegimos como pivote del 23. Recorremos el array desde la izquierda y buscamos un elemento mayor que 23 (encontramos el 28). A continuación, recorremos el array en

sentido descendente empezando por el extremo derecho y buscamos un valor menor que 23 (encontramos el 14). Se intercambian estos los valores y se produce la lista:

```
| 14      21      37      23      19      28      26
```

se sigue recorriendo el array por la izquierda y se encuentra otro número que es mayor que 23: el 37; continuamos al recorrido por la izquierda y encontramos otro valor menor que 23: 19. Volvemos a cambiar sus posiciones:

```
| 14      21      19      23      37      28      26
```

en este punto todos los valores que están a la izquierda del pivote son menores que él, y todos los que están a su derecha son mayores. Ninguno de los dos subconjuntos de valores está ordenado. En este momento procesamos cada uno de los dos subconjuntos de valores del mismo modo que el inicial: elegimos un valor de pivote y lo ordenamos de tal modo que todos los que sean mayores que el pivote estén a la derecha y todos los que sean menores a la izquierda. Así, si cogemos el primer conjunto de datos:

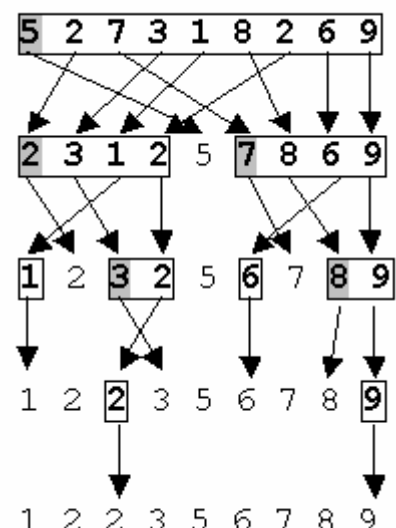
```
| 14      21      19
```

y tomamos como pivote el 14 obtenemos:

```
| 14      19      21
```

del mismo modo, procesamos el otro conjunto de datos. Cuando el tamaño de los conjuntos de datos que se sitúan a la izquierda y a la derecha del valor tomado como pivote es 0 o 1 habremos terminado el procedimiento de ordenación. La corrección de este algoritmo se basa en los siguientes hechos:

- El subconjunto de elementos menores que el pivote se ordenará correctamente, en virtud del proceso recursivo.
- El mayor elemento en el subconjunto de elementos menores no es mayor que el pivote, debido a cómo se realiza la partición.
- El menor elemento en el subconjunto de elementos mayores no es menor que el



pivote, debido a cómo se realiza la partición.

- El subconjunto de elementos mayores se ordena correctamente, en virtud del proceso recursivo.

```

/*
 *ejemplo8_7.c
 */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TAM 100

void quickSort( int[], int, int);
int partition( int[], int, int);

void imprimeCB(int *CB) {
    int i;
    for(i = 0; i < TAM-1; i++) {
        printf( "%d, ", CB[i]);
    }
    printf( "%d\n", CB[i]);
}

int main() {
    int CB[TAM];
    int e;
    srand((unsigned int)time(NULL));

    for(e = 0; e < TAM; e++)
        CB[e] = (int)(rand() % 100);

    printf( "Antes de ordenar\n-----\n");
    imprimeCB(CB);

    quickSort( CB, 0, TAM-1);

    printf( "\nDespués de ordenar\n-----\n");
    imprimeCB(CB);
}

void quickSort( int CB[], int izquierda, int derecha){
    int indice_pivote;
    if( izquierda < derecha ) {
        indice_pivote = partition( CB, izquierda, derecha);
        quickSort( CB, izquierda, indice_pivote-1);
        quickSort( CB, indice_pivote+1, derecha);
    }
}

int partition( int CB[], int izquierda, int derecha) {

```



```
int pivote, i, j, tmp;
pivote = CB[izquierda];
i = izquierda; j = derecha;

while( 1){
    while( CB[i] <= pivote && i <= derecha ) ++i;
    while( CB[j] > pivote ) --j;
    if( i >= j ) break;
    tmp = CB[i]; CB[i] = CB[j]; CB[j] = tmp;
}
tmp = CB[izquierda]; CB[izquierda] = CB[j]; CB[j] = tmp;
return j;
}
```

4.4.1 Análisis del rendimiento de Quicksort

El mejor caso para Quicksort se presenta cuando el pivote divide al conjunto en dos subconjuntos de igual tamaño. En este caso tendremos dos llamadas recursivas con un tamaño igual a la mitad del original y el tiempo de ejecución es $O(TAM \cdot \log(TAM^2))$.

Ya que los subconjuntos de igual tamaño son los mejores para Quicksort, es de esperar que los de muy distinto tamaño sean los peores y, efectivamente, así es. Suponiendo, por ejemplo, que el elemento que se toma como pivote es el menor del conjunto el subconjunto de la izquierda (elementos menores) estará vacío, y el de la derecha (elementos mayores) contendrá a todos los elementos menos al pivote. Poniéndonos en el peor de los casos, aquél en el que siempre obtenemos uno de los dos subconjuntos vacíos y el otro contiene $n-1$ elementos, la complejidad del algoritmo sería $O(TAM^2)$.

En el caso medio, si el algoritmo es implementado cuidadosamente y los subconjuntos de elementos generados en cada partición contienen aproximadamente el mismo número de elementos, puede demostrarse que el tiempo de ejecución es $O(TAM \cdot \log(TAM^2))$. Para conseguir esta implementación cuidadosa es crucial determinar adecuadamente el elemento pivote.

Una elección muy popular para el elemento pivote, que puede degradar notablemente el rendimiento del algoritmo, es emplear el primer elemento de la izquierda. Este pivote es aceptable si la entrada es completamente aleatoria, pero si la entrada ya está ordenada, o está ordenada en orden inverso, este pivote proporciona la peor división posible. Una elección más razonable es elegir como elemento pivote el elemento central de del array.

Lo ideal, sería elegir el valor mediana del array; esto es, aquel valor que ordenando en orden decreciente o creciente la lista de elementos quedaría justo en el medio (por tanto, este es el pivote ideal). Sin embargo, el cálculo de la mediana de una lista de elementos es excesivamente costosa para incorporarse al algoritmo; esto lleva calcular aproximaciones tomando un subconjunto de los elementos del array. En muchas ocasiones se emplea la "partición con la mediana de tres", en esta partición se emplea como pivote la mediana de los elementos primero, último y punto medio del array.

Otro factor que afecta de modo crítico la eficiencia del algoritmo es qué hacer con aquellos datos que son iguales al pivote. Si, por ejemplo, sistemáticamente colocamos los datos que son mayores que el pivote en el subconjunto de la derecha esto puede llevarnos a realizar particiones desproporcionadas cuando un dato se repite un número excesivo de veces. El caso extremo será cuando todos los elementos de la lista a particionar sean iguales al pivote. En este hipotético caso la partición sería la peor de las posibles: una lista estaría vacía y la otra contendría todos los elementos.

A simple vista, podría parecer absurdo que un programa informático tuviera que ordenar una lista de valores iguales. Sin embargo, no lo es tanto: supongamos que tenemos que ordenar una lista con un millón de datos enteros comprendidos entre el 0 y 999. Si realizamos la ordenación mediante Quicksort y suponemos una distribución uniforme de los datos llegará un momento en el que, tras varias llamadas recursivas, nuestro problema consista en ordenar 1000 listas de, aproximadamente, unos 1000 valores cada una de ellas. Estos valores serán en su mayor parte iguales: habrá una lista con todo 0s, una con todo 1s,..., y una con todo 999s. Llegado este momento, es obvia la importancia de que nuestra implementación de Quicksort gestione adecuadamente conjuntos de datos iguales.

La mejor estrategia en este caso es intercambiar los elementos iguales que estén a ambos lados del pivote: si al buscar un elemento menor que el pivote en el subconjunto de la izquierda encontramos un elemento igual al pivote lo intercambiaremos por un elemento mayor o igual que el pivote del subconjunto de la derecha. A pesar del aparente derroche de operaciones que, en caso de que ambos elementos intercambiados sean iguales, no hace nada en la práctica, es mejor gestionar esta situación de este modo y no añadir código adicional que cheque la ocurrencia de esta situación y que, inevitablemente, penalizaría la eficiencia del algoritmo.

5 Ejercicios

1. Optimizar la implementación del algoritmo quicksort dada en los apuntes según las indicaciones del apartado anterior.
2. Dado un fichero de texto (donde no aparece la letra ñ) ordena sus líneas alfabéticamente mediante el método de la burbuja, selección y quicksort, generando en los tres casos un archivo nuevo que contenga el texto ordenado.
3. Dado un fichero de texto realizar un programa que permita buscar en él una palabra dada. Las palabras no tienen por qué estar ordenadas alfabéticamente y ninguna de ellas contiene la letra ñ.
4. Modificar el programa anterior para que emplee una búsqueda binaria.
5. Dado un vector de N elementos (que se deberán inicializar aleatoriamente) eliminar todos sus elementos repetidos. Emplear para ello una variante del método de selección.
6. Modificar el programa 37 del tema 7 (o, preferiblemente, el 46) para que el listado de usuarios se realice ordenándolos según su nombre.
7. Modificar el programa anterior para que permita realizar búsquedas por nombre suponiendo que los registros de empleados no están ordenados.
8. Modificar el programa anterior para que realice las búsquedas mediante el método de búsqueda binaria.
9. Modificar el programa anterior para añadir opciones de ordenar por fecha de ingreso en la empresa y por salario. Usa dos métodos diferentes entre sí y diferentes al empleado para ordenar por nombre.