UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AERONÁUTICA Y DEL ESPACIO

ALGORITMOS DE BÚSQUEDA Y DE ORDENACIÓN

SEARCH AND SORT ALGORITHMS

Yago Pego Martínez (yago.pego.martinez@alumnos.upm.es)
Evaristo de Vega Galindo (evaristo.devega.galindo@alumnos.upm.es)

Grupo M4_02

Índice general

1- MARCO TEÓRICO

Algoritmo y programación

Un algoritmo es un procedimiento que describe de forma inequívoca una secuencia de pasos en un orden específico con el objetivo de resolver un problema u obtener soluciones aproximadas, en la matemática, física o ingeniería.

Por otro lado, un programa o código es la traducción de un algoritmo a un lenguaje de programación adecuado para que el ordenador lo resuelva; siendo la programación el conjunto de conceptos teóricos que nos permiten diseñar, desarrollar y mantener estos programas o códigos.

Lenguaje Fortran

Definidas las bases en que se sustenta cualquier lenguaje de programación, cabe hablar de Fortran, al que hemos "traducido" nuestras intenciones algorítmicas.

Fortran (previamente FORTRAN, del inglés *The IBM Mathematical* ***For****mula* ***Tran****slating System*) es un lenguaje de programación de alto nivel que está especialmente adaptado al cálculo numérico y a la computación científica. Desde que nació en el año 1957, de la mano de John W. Backus y su equipo de programadores, para el equipo IBM 704, el lenguaje FORTRAN ha estado en uso continuo hasta la actualidad, siendo la ingeniería y la ciencia sus principales aplicaciones.

Desde el primer compilador Fortran, que aparece en el año 1957 y que tenía hasta treinta y dos sentencias (*write*, *read*, *if*, *dimension*...), hasta la última y más reciente versión: Fortran 2008 (aprobada en 2010), que facilita la interoperabilidad con el lenguaje de programación C; se han creado hasta seis versiones intermedias. Entre ellas, la que hemos venido utilizando durante el presente curso, Fortran 90, que, en su momento, renovó el lenguaje con la introducción de procedimientos recursivos (ver *quicksort*), comentarios entre líneas y acumulador de operaciones.

Gran parte de mi trabajo ha venido de ser vago. No me gustaba escribir programas y, entonces, cuando estaba trabajando con el IBM 701 (un prototipo de ordenador) escribiendo programas para calcular trayectorias de misiles, empecé a dedicarme a crear un nuevo sistema programador que facilitara escribir estos programas.

John Warner Backus, creador de Fortran

2- ALGORITMOS DE BÚSQUEDA

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.

La variante más simple del problema, y a la que nos hemos dedicado, es la búsqueda de un número en un vector.

Existen dos tipos principales de búsqueda: la búsqueda lineal o secuencial y la búsqueda binaria o dicotómica.

Como cabe esperar, en un algoritmo programado para que, una vez encontrado el valor, finalicen el bucle y el programa, el caso óptimo será cuando el primer elemento sea el buscado, y el peor caso, cuando este sea el último.

Búsqueda secuencial

En ciencias de la computación, la búsqueda secuencial o lineal es un método para encontrar un valor-objetivo en una lista. El algoritmo recorre uno a uno cada uno de los elementos hasta toparse con el buscado o, en el peor caso, hasta finalizar la búsqueda de manera insatisfactoria.

Aunque la búsqueda secuencial se trata de un algoritmo fácilmente programable, su eficiencia disminuye ante grandes series, de millones o más elementos, con respecto a la citada búsqueda dicotómica o las *hash tables*.

```
Datos de entrada:
  vec: vector en el que se desea buscar el dato
  tam: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
Puede representarse así:  vec[0...tam) o vec[0...tam-1].
  dato: elemento que se quiere buscar.

Variables
  pos: posición actual en el vector

pos = 0
while pos < tam:
  if vec[pos] == dato:
      Retorne  verdadero y/o pos,
  else:
     pos = pos + 1
Fin (while)
Retorne falso,
```

Pseudocódigo

Aparte de la habitual y simple búsqueda de un valor en un *array*, nosotros hemos decidido ampliar las aplicaciones de la búsqueda lineal a, por ejemplo, los espacios matriciales, o a la búsqueda de un valor tantas veces como esté presente.

Búsqueda de un valor en una serie.

¿Problema? En caso de que ese valor se repita una o más veces, el algoritmo solamente identificará el primero.

```
subroutine simple_search(dimension, comp, vector)
implicit none

integer, intent(in) :: dimension
real, intent(inout) :: vector(dimension)
real, intent(in) :: comp

integer :: i

do i = 1, dimension
if (comp == vector(i)) exit
enddo

if (i == dimension+1) then
write(*,*) "No hay ninguna componente con ese valor."
endif

if (i <= dimension) then
write(*, "(a31, 1x, i2)") "Ese valor esta en la posicion", i
end if

end subroutine
```

```
C:\Users\USUARIO\Documents\Universidad\Asignatur...
Introduce la dimension del vector:
7

Escribe el vector:
2  4  -1  0  3  2  1

Escribe el valor que quieres buscar:
0

 Ese valor esta en la posicion  4
```

Búsqueda de un valor en una matriz.

A diferencia del anterior, este algoritmo sí dará las posiciones del valor pedido por teclado, tantas veces como aparezca, indicando fila y columna.

```
subroutine search (fila, columna, matriz)

integer, intent(in) :: fila
integer, intent(in) :: columna
real(8), intent(in) :: matriz(fila, columna)
integer :: valor
integer :: i, j

write(*,*) "Indica el valor que deseas buscar entre los elementos de la matriz:"
read(*,*) valor
write(*,*)

i = 1
do while (i <= fila)
do j = 1, columna
if (matriz(i, j) == valor) then
write(*,"(a37, 2(i2, 1x))") "El valor se encuentra en la posicion ", i, j
endif
enddo
i = i + 1
enddo

end subroutine
```

```
C:\Users\USUARIO\Documents\Universidad\Asignatur...
Introduce el numero de filas de la matriz:
4

Introduce el numero de columnas de la matriz:
3

Escribe la matriz (por filas, de arriba a abajo):
2  -2  3

1  -1  0

3   0  1

3   7 -2

Indica el valor que deseas buscar entre los elementos de la matriz:
3

El valor se encuentra en la posicion  1  3
El valor se encuentra en la posicion  3  1
El valor se encuentra en la posicion  4  1
```

Búsqueda de dos valores en una serie.

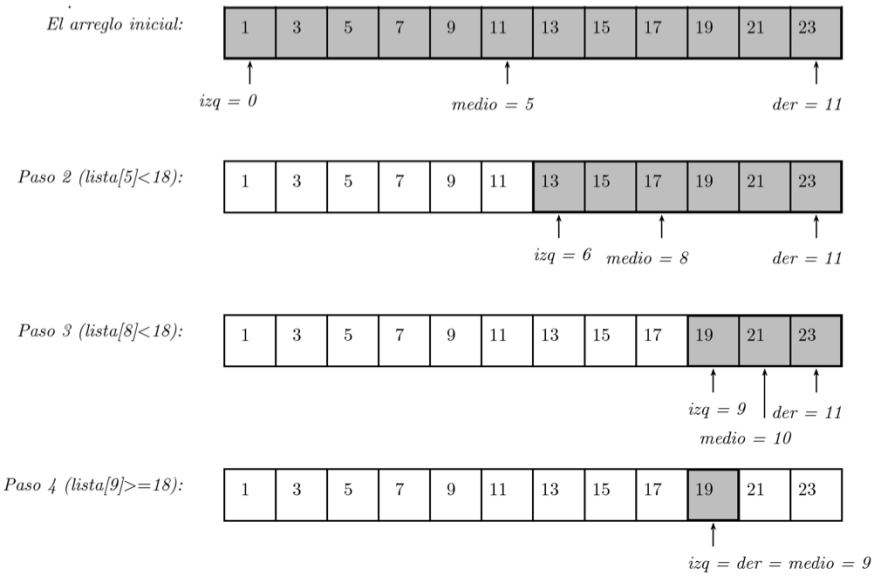
Para un vector de dimensión asignable, podemos pedir la posición de dos valores cualesquiera. Por pantalla aparecerán dichas posiciones.

```
C:\Users\USUARIO\Documents\Universidad\Asignatur...
Introduce la dimension del vector:
12

Escribe el vector:
3  -3  2  4  2  1  -1  7  7  8  7  2

Introduce el primer valor a buscar:
2

Introduce el segundo valor a buscar:
7

El primer valor buscado esta en la posicion   3
El primer valor buscado esta en la posicion   5
El primer valor buscado esta en la posicion  12

El segundo valor buscado se encuentra en la posicion   8
El segundo valor buscado se encuentra en la posicion   9
El segundo valor buscado se encuentra en la posicion  11
```

```
subroutine search(dimens,comp_1,comp_2,vector)
implicit none

integer, intent(in) :: dimens
real(8), intent(inout) :: vector(dimens)
real(8), intent(in) :: comp_1
real(8), intent(in) :: comp_2

integer :: i, j

do i = 1, dimens
if (comp_1 == vector(i)) exit
enddo

do j = 1, dimens
if (comp_2 == vector(j)) exit
enddo

if (i == dimens+1) then
write(*,*) "El primer valor buscado no existe entre las componentes del vector."
end if

if (j == dimens+1) then
write(*,*) "El segundo valor buscado no existe entre las componentes del vector."
end if

if (i <= dimens) then
write(*,"(a35, 2x, i2)") "El primer valor esta en la posicion", i
endif

if (j <= dimens) then
write(*,"(a36, 2x, i2)") "El segundo valor esta en la posicion", j
endif

end subroutine
```

Búsqueda binaria

En ciencias de la computación, la búsqueda binaria, también conocida como búsqueda logarítmica o búsqueda del medio intervalo (*half-interval search*), es un algoritmo que permite localizar un valor en un *array* previamente ordenado.



La búsqueda binaria se fundamenta en la comparación del valor buscado y un valor pivote (variable), y las sucesivas subdivisiones del intervalo.

La búsqueda logarítmica es además una sencilla aplicación del algoritmo "divide y vencerás" (en inglés, *divide and conquer*).

Al igual que en la búsqueda secuencial, en el mejor caso posible, solamente se necesitará una operación para dar con el valor-objetivo. En este caso, pivote y valor pedido coincidirán en el primer momento. En el caso contrario, en el peor comportamiento del algoritmo, el número de operaciones se reducirá considerablemente, al orden de *log n* operaciones, donde *n* es el número de elementos del *array*.

Para el correcto funcionamiento de este algoritmo, se deberán cumplir los tres siguientes requisitos:

- Se ha de conocer la dimensión del vector.
- Se ha de conocer el valor a buscar.
- El vector ha de estar ordenado, antes de iniciar la búsqueda.

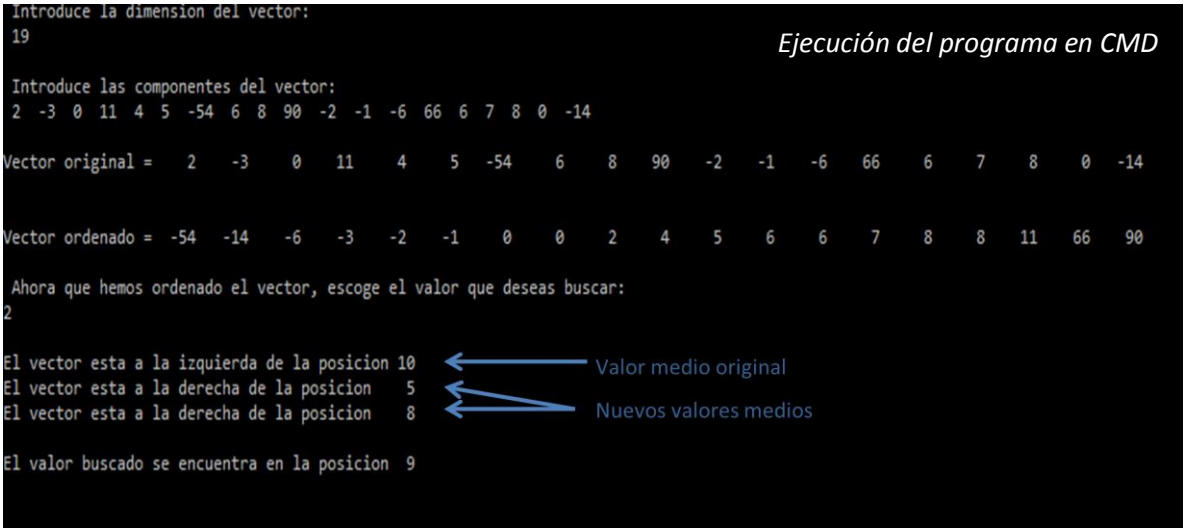
En el programa que hemos creado se cumplen estas tres condiciones: es el propio usuario el que se encarga de asignar al vector una dimensión determinada y de escoger el valor a buscar. En cuanto a la ordenación, se ha implementado una subrutina, que precede a la de búsqueda, de ordenamiento por selección (cualquier método para ordenar es válido).

Subrutina "binary_search" en Fortran90

```
subroutine binary_search(dimension, vector, valor)

integer, intent(in) :: dimension, valor
integer, intent(inout) :: vector(dimension)
integer :: i, k, sup, inf
real :: n

n = dimension
n = n/2

i = dimension/2
if (n > i .and. n < (i+1)) then
i = i + 1
endif

sup = dimension
inf = 0

do while (valor /= vector(i))
    if (valor > vector(i)) then
    inf = i
    n = inf + sup
    n = n/2
    i = (inf + sup)/2
        if (n > i .and. n < (i+1)) then
        i = i + 1
        endif
    sup = sup
    endif
    if (valor < vector(i)) then
    sup = i
    n = inf + sup
    n = n/2
    i = (inf + sup)/2
        if (n > i .and. n < (i+1)) then
        i = i + 1
        endif
    inf = inf
    endif
enddo

write(*,"(a45, (i2, 2x))") "El valor buscado se encuentra en la posicion ", i

end subroutine
```

```
busquedaBinaria(elemento, arreglo[]):
    izquierda=0, derecha=longitud(arreglo)-1
    mientras izquierda<=derecha:
        mitad = (izquierda+derecha)/2
        si arreglo[mitad]==elemento
            retornar mitad
        si arreglo[mitad]>elemento:
            derecha = mitad-1
        si arreglo[mitad]<elemento:
            izquierda = mitad+1
    fin mientras
    retornar -1
```

Pseudocódigo



Ejecución del programa en CMD

Como curiosidad, cabe decir que la búsqueda binaria fue mencionada por primera vez en 1946 por John Mauchly, como parte de las conferencias de Moore School, las primeras de la historia relacionadas con la programación. Además, y aunque la búsqueda binaria no es, a priori, excesivamente difícil de programar, sorprende saber que durante más de veinte años no fue detectado un error de desbordamiento (*overflow error*) en el libro "Programming pearls", de Jon Bentley.

Desde entonces, se han creado distintas variaciones del algoritmo de búsqueda binaria original. Como explicar cada uno de ellos, superaría lo necesariamente abordable en el presente trabajo, solamente los nombraremos: la búsqueda límite (*boundary search*), la búsqueda de Fibonacci, la búsqueda exponencial, la búsqueda con interpolación (*interpolation search*), y el algoritmo *fractional cascading*.

Comparativa

	< 1 000 000	10 000 000	100 000 000	200 000 000	300 000 000	400 000 000	450 000 000*
Búsqueda secuencial	≈ 0 s	0,015625 s	0,296875 s	0,8125 s	1,125 s	1,390625 s	1,515625 s

*La unidad central de procesamiento no nos ha permitido trabajar con dimensiones del orden de 10^9. La búsqueda binaria no aparece en la tabla anterior, porque el tiempo de ejecución para todas las pruebas realizadas (hasta el límite) se aproxima a cero.

Se podría concluir con que, para vectores de muy altas dimensiones, sí es conveniente hacer uso del algoritmo de búsqueda binaria frente al de búsqueda lineal. No necesariamente ha de ser así para vectores de dimensión menor, pues no sería correcto decir que la búsqueda secuencial es mucho menos eficiente.

3- ALGORITMOS DE ORDENACIÓN

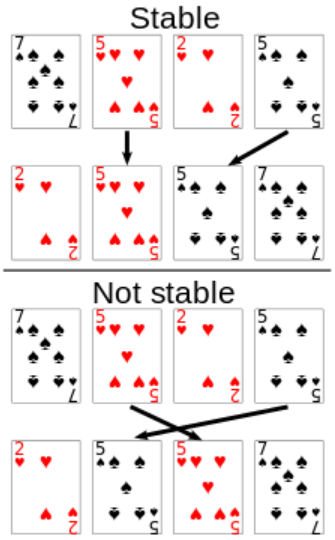
Un algoritmo de ordenación es aquel que pone elementos de una lista o vector en una secuencia dada por una relación de orden, de modo que el resultado de salida es una permutación o transformación de la serie original ordenada de la manera requerida.

Las relaciones de orden más utilizadas son las de tipo lexicográfico y numérico. En las segundas nos hemos centrado.

La importancia de estos algoritmos reside en la necesidad de optimizar el uso de otros algoritmos (búsqueda, fusión, etc.) que requieren listas ordenadas para una ejecución rápida.

La búsqueda de la eficiencia máxima para estos programas ha sido una constante desde los inicios de la computación. Por poner un ejemplo, se ha estado trabajando con el ordenamiento de burbuja (*bubblesort,* en inglés) desde el año 1956; al contrario, recientemente, en 2004, se publicó por primera vez el ordenamiento de biblioteca.

Una forma de clasificar los distintos tipos de algoritmos de ordenación es según su estabilidad. Que un ordenamiento sea estable quiere decir que este mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si tenemos una baraja de cartas con un determinado orden original, en casos en que el valor de dos cartas sea el mismo (de distinto palo), en un algoritmo estable, ante esta "igualdad", se tendrán en cuenta las posiciones en la serie original, conservándose ese orden.



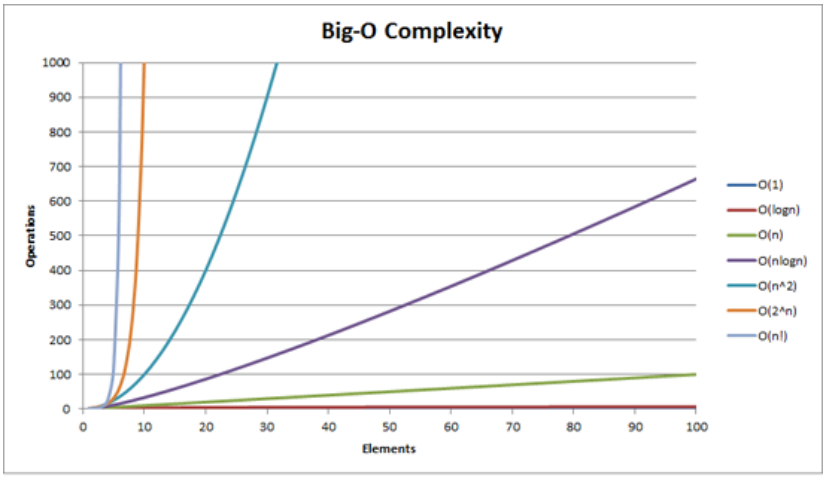
En cualquier caso, la mayoría de los algoritmos inestables pueden ser artificialmente modificados y convertidos en estables, implementando que ante la igualdad numérica se priorice la posición inicial. No obstante, esta actualización puede requerir tiempo y espacio adicional.

Sin embargo, la inestabilidad deja de ser un problema cuando trabajamos con datos numéricos (ya sean enteros o reales), pues la única clave de ordenación es la propia relación numérica.

Algoritmos estables

Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento de burbuja	*Bubblesort*	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento de burbuja bidireccional	*Cocktail sort*	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento por inserción	*Insertion sort*	$O(n^2)$ ("en el peor de los casos")	$O(1)$	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
	Pigeonhole sort	$O(n+k)$	$O(k)$	Distribución
Ordenamiento Radix	*Radix sort*	$O(nk)$	$O(n)$	No comparativo
	Distribution sort	$O(n^3)$ ("versión recursiva")	$O(n^2)$	Distribución
	Gnome sort	$O(n^2)$	$O(1)$	Intercambio

Antes de pasar a explicar los distintos algoritmos cabe explicar qué es la complejidad computacional (tercera columna). Para definir la eficiencia de un algoritmo de ordenamiento para una lista de n elementos, se realizan estudios experimentales para determinar los comportamientos peor, promedio y mejor del algoritmo. Así, para algunos ordenamientos sencillos, como el de inserción, el mejor comportamiento se dará cuando la lista esté ya previamente ordenada; en ese caso, la complejidad será $O(n)$: n comparaciones necesarias. En resumen, la complejidad computacional está relacionada directamente con el número de comparaciones requeridas.



De entre los distintos algoritmos estables citados en la tabla de la página anterior, nosotros hemos trabajado con el ordenamiento de burbuja y el ordenamiento por inserción.

Ordenamiento de burbuja

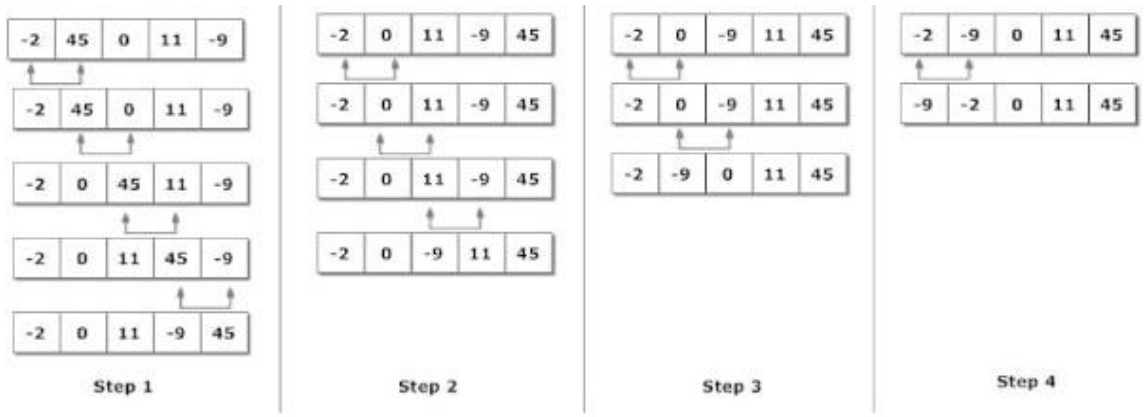


Figure: Working of Bubble sort algorithm

El ordenamiento de burbuja, o *bubblesort*, es un algoritmo sencillo que funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si el orden no es el correcto. Es necesario revisar varias veces la lista hasta que no se necesiten más intercambios.

```
procedimiento DeLaBurbuja (a(0), a(1), a(2), ..., a(n-1))
    para i ← 1 hasta n hacer
        para j ← 0 hasta n − i hacer
            si a(j) > a(j+1) entonces
                aux ← a(j)
                a(j) ← a(j+1)
                a(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

Pseudocódigo

El también conocido como método de intercambio directo tiene una complejidad computacional óptima de O(n) y un promedio y peor caso de $O(n^2)$.

```
Introduce la dimension del vector:
6

Escribe el vector:

2 -2 1 9 7 -4

Vector original =    2   -2    1    9    7   -4

        -2          2          1          9          7          -4
        -2          1          2          9          7          -4
        -2          1          2          7          9          -4
        -2          1          2          7          -4         9
        -2          1          2          -4         7          9
        -2          1          -4         2          7          9
        -2          -4         1          2          7          9
        -4          -2         1          2          7          9

Vector ordenado =   -4   -2    1    2    7    9
```

Ejecución del programa en CMD

```
subroutine bubblesort(dimension, vector)
integer :: i, j, k
real :: swap
integer, intent(in) :: dimension
real, intent(inout) :: vector(dimension)

do j = 1, dimension-1
    do k = 2, dimension
    if (vector(j) > vector(k)) then
        do i = 1, dimension-1
        if (vector(i) > vector(i+1)) then
        swap = vector(i)
        vector(i) = vector(i+1)
        vector(i+1) = swap
        endif
        enddo
    endif
    enddo
enddo

end subroutine
```

Subrutina "bubblesort" en Fortran90

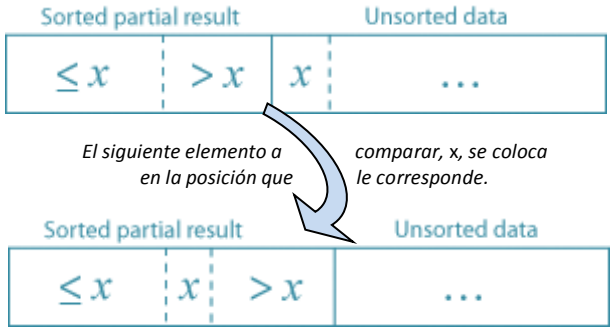
Ordenamiento por inserción

El ordenamiento por inserción (*insertion sort*, en inglés) es una manera muy natural de ordenar para un ser humano. Es comparable a ordenar un mazo de cartas numéricamente.

El algoritmo consiste en reordenar una serie de dimensión indefinida, de modo que se comparan todos los elementos, de izquierda a derecha, colocando el i-ésimo en la posición que le corresponde respecto a los elementos anteriores, que ya ocuparán una posición ordenada.

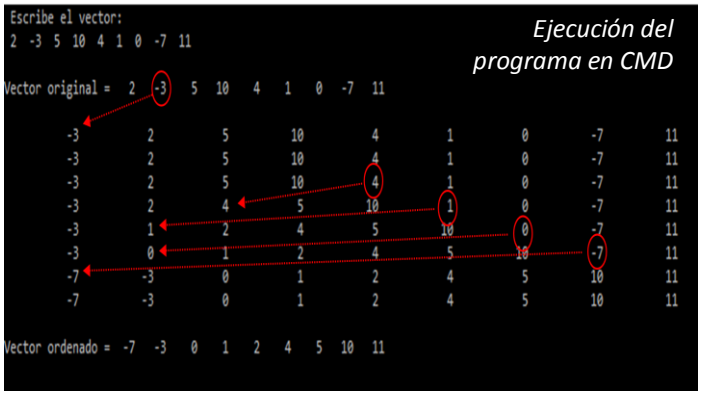
```
for i ← 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
end for
```

Pseudocódigo




```
subroutine insertionsort(dimension, vector)

integer :: i, j
integer :: swap
integer, intent(in) :: dimension
real, intent(inout) :: vector(dimension)

do i = 2, dimension
j = i - 1
swap = vector(i)
    do while (j >= 1 .and. vector(j) > swap)
    vector(j+1) = vector(j)
    j = j - 1
    enddo
vector(j+1) = swap
enddo

end subroutine
```

Subrutina "insertionsort" en Fortran90



Ejecución del programa en CMD

Como se venía diciendo antes, el mínimo número de comparaciones que va a realizar este método de ordenación va a ser igual al número de elementos de la serie. En casos no particulares, la inserción presentará una complejidad de $O(n^3)$.

De entre los algoritmos de ordenación con un código más fácil de programar, sean también el de burbuja y el de selección, el método por inserción es el más rápido.

Algoritmos 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	*Heapsort*	$O(n \log n)$	O(1)	Selección
	Smoothsort	$O(n \log n)$	O(1)	Selección
Ordenamiento rápido	*Quicksort*	promedio: $O(n \log n)$ peor: $O(n^2)$	O(log *n*)	Partición
	Several unique sort	promedio: $O(n\ u)$ peor: $O(n^2)$ u = número único de registros		

De entre los distintos algoritmos inestables citados en la tabla anterior, nosotros hemos trabajado con el ordenamiento por selección, el ordenamiento rápido y el ordenamiento *Shell*.

Ordenamiento por selección

El ordenamiento por selección (*selection sort*, en inglés) es un método de ordenación que se fundamenta en el reposicionamiento de los elementos de una lista a partir de la búsqueda del valor mínimo, que se colocará en la posición primera.

Como algunos ordenamientos anteriores, este también tiene siempre una complejidad computacional de $O(n^2)$ —independiente de la posición inicial de los elementos—, que, aunque resulta ineficiente para *arrays* de cientos de miles de componentes, supera ligeramente en tiempo necesario a los ordenamientos de burbuja y gnomo (*gnome sort*).

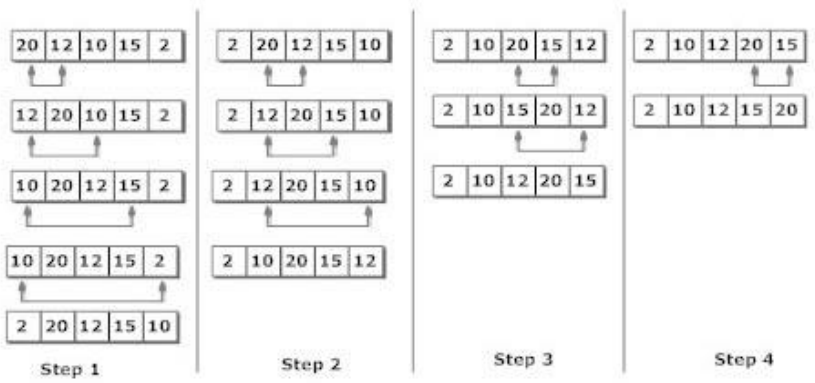


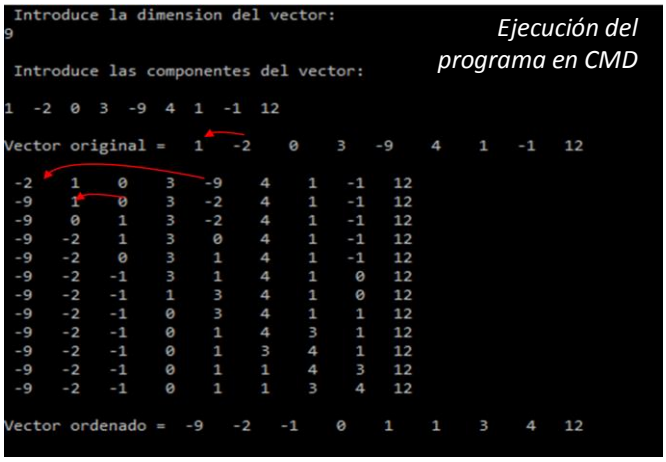
Figure: Selection Sort

```
para i=1 hasta n-1
    mínimo = i;
    para j=i+1 hasta n
        si lista[j] < lista[mínimo] entonces
            mínimo = j /* (!) */
        fin si
    fin para
    intercambiar(lista[i], lista[mínimo])
fin para
```

Pseudocódigo

```
subroutine selectionsort(dimension, vector)

integer, intent(in) :: dimension
real, intent(inout) :: vector(dimension)
real :: swap
integer :: i, j

do i = 1, dimension-1
    do j = i+1, dimension
        if (vector(i) > vector(j)) then
            swap = vector(i)
            vector(i) = vector(j)
            vector(j) = swap
        endif
    enddo
enddo

end subroutine
```

Subrutina "selectionsort" en Fortran90



Ejecución del programa en CMD

Una versión mejorada y más eficiente del ordenamiento por selección es el ordenamiento por montículos (ver tabla), descubierto en 1964 por J. W. J. Williams, que se acerca en rapidez al *quicksort*.

Ordenamiento rápido

El ordenamiento rápido, también conocido como *quicksort* o *partition-exchange sort)*, es un método sistemático de colocar los elementos de un *array* en un determinado orden. Es considerado a menudo el método más rápido —de ahí su nombre—, aunque en el peor caso, pierde la ventaja frente a otros como el *merge sort* o *heapsort*.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- 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 ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(n \log n)$.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del *array*, y el *array* que le pasamos está ordenado, siempre va a generar a su izquierda un *array* vacío, lo que es ineficiente.
- En el caso promedio, el orden es $O(n \log n)$.

No es de extrañar, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot

        do
            j := j - 1
        while A[j] > pivot

        if i >= j then
            return j

        swap A[i] with A[j]
```

Pseudocódigo

```
recursive subroutine quicksort(A)
  real, intent(in out), dimension(:) :: A
  integer :: iq

  if(size(A) > 1) then
     call partition(A, iq)
     call quicksort(A(:iq-1))
     call quicksort(A(iq:))
  endif
end subroutine quicksort

subroutine partition(A, marker)
  real, intent(in out), dimension(:) :: A
  integer, intent(out) :: marker
  integer :: i, j
  real :: temp
  real :: x
  x = A(1)
  i= 0
  j= size(A) + 1

  do
     j = j-1
     do
        if (A(j) <= x) exit
        j = j-1
     end do
     i = i+1
     do
        if (A(i) >= x) exit
        i = i+1
     end do
     if (i < j) then
        temp = A(i)
        A(i) = A(j)
        A(j) = temp
     elseif (i == j) then
        marker = i+1
        return
     else
        marker = i
        return
     endif
  end do

end subroutine partition
```

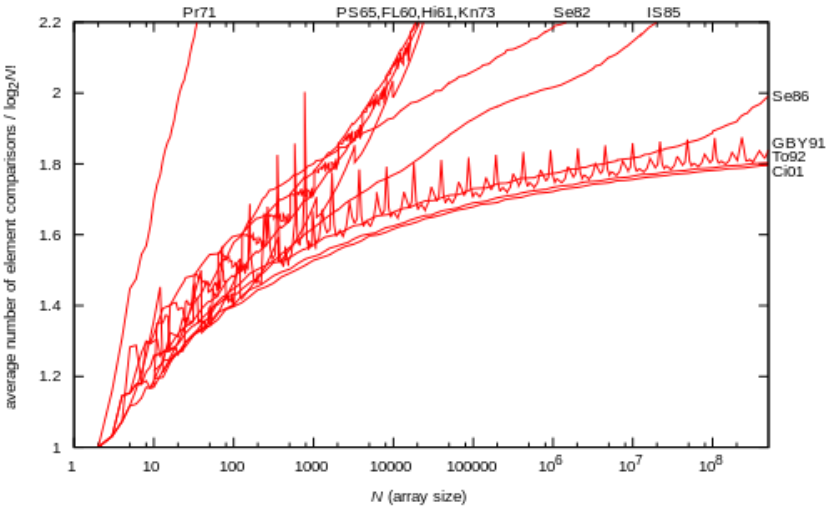
Subrutina recursiva "quicksort".

Una subrutina recursiva pretende, a través de su repetición, que el problema se reduzca a su caso base.

Ordenamiento Shell

Denominado así en honor a su creador, el estadounidense Donald Shell, el ordenamiento homónimo es una generalización del método de inserción. Respecto a este, *Shellsort* mejora al anterior comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Como consecuencia, el último paso del ordenamiento Shell es una simple ordenación por inserción, pero, para entonces, ya está garantizado que los elementos del vector estén casi ordenados.

Mejora en eficiencia los algoritmos más simples: burbuja, selección e inserción. Sin embargo, resulta más difícil de programar y, en la mayor parte de los casos, no alcanza la velocidad de ejecución de otros métodos, como el *quicksort* o el *merge sort*.



La tabla de la izquierda relaciona el número de elementos de un *array* con el número promedio de comparaciones necesarias para el ordenamiento Shell.

```
subroutine shellsort(vector)

  implicit  none
  integer :: i, j, increment
  real :: temp
  real, intent(inout) :: vector(:)

  increment = size(vector)/2
  do while (increment > 0)
     do i = increment+1, size(vector)
        j = i
        temp = vector(i)
        do while (j >= increment+1 .AND. vector(j-increment) > temp)
           vector(j) = vector(j-increment)
           j = j - increment
        enddo
        vector(j) = temp
     enddo
     if (increment == 2) then
     increment = 1
     else
        increment = increment * 5 / 11
     end if
  enddo

end subroutine
```

Subrutina "shellsort" en Fortran90

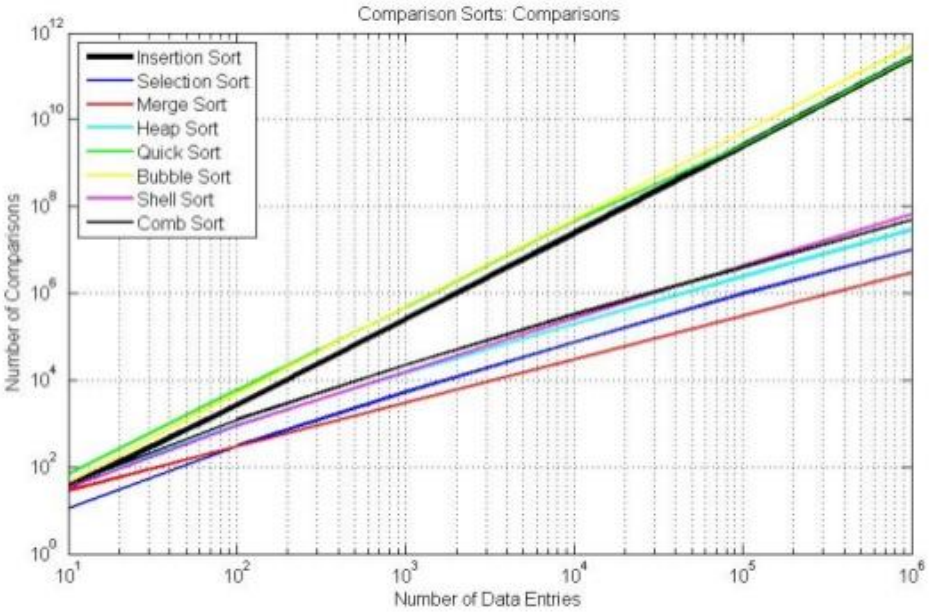
En cuanto a su complejidad computacional, el ordenamiento Shell alcanza un $O(n \log n)$ para su mejor comportamiento, y $O(n^{1.25})$ ó $O(n \log^2 n)$ de promedio.

El tiempo de ejecución del ordenamiento Shell está estrechamente relacionado con la longitud de los saltos. Todavía hoy, determinar la complejidad del algoritmo sigue siendo un problema sin cerrar.

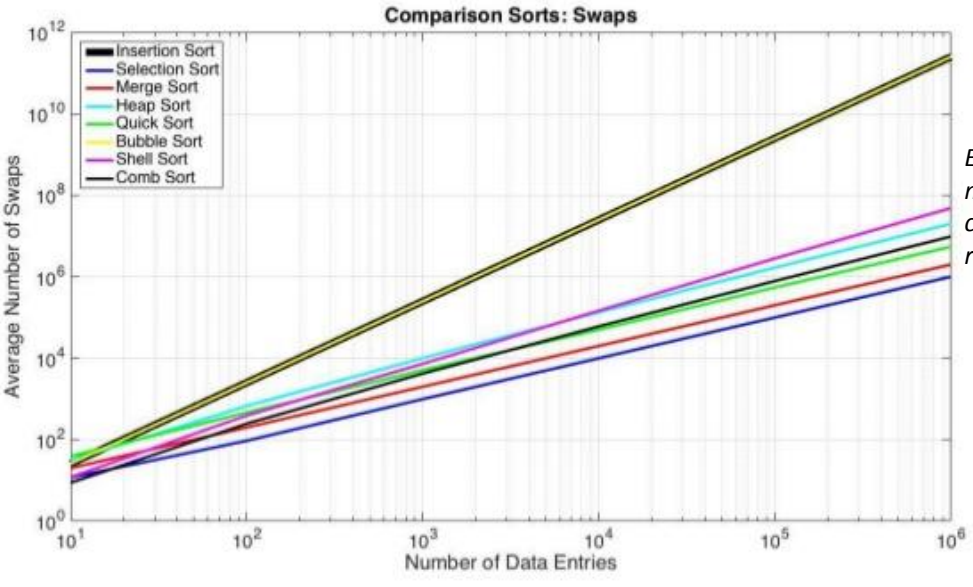
Comparativa

Como se vino sugiriendo en los apartados anteriores, existe una enorme diferencia entre los algoritmos de ordenación presentados, en cuanto al tiempo de ejecución. Además, estas diferencias se ven compensadas con la mayor o menor dificultad de programar que presenta cada método.

	10	100	1 000	10 000	100 000	1 000 000	10 000 000	100 000 000
Bubblesort	≈ 0	≈ 0	≈ 0	0.234375	27.765625	2803.71875	–	–
Insertion sort	≈ 0	≈ 0	≈ 0	0.0625	6.390625	655.703125	–	–
Selection sort	≈ 0	≈ 0	≈ 0	0.28125	27.140625	2720.15625	–	–
Quicksort	≈ 0	≈ 0	≈ 0	≈ 0	0.015625	0.1875	2.21875	23.359375
Shell sort	≈ 0	≈ 0	≈ 0	≈ 0	0.015625	0.3125	3.71875	42.9375



La tabla adjunta relaciona el número de elementos del array *con el número de comparaciones requeridas.*



Esta segunda tabla relaciona el número de elementos del array *con el número de intercambios realizados.*

Podemos sacar conclusiones de los datos anteriores y decir que:

- el ordenamiento rápido, aunque precisa de una alta cifra de comparaciones, solamente necesitará intercambiar elementos, hablando logarítmicamente, en la mitad de las ocasiones.
- los ordenamientos Shell y por mezcla (*merge sort*), que son de los más rápidos y eficientes, logran una relación relativamente constante entre el número de comparaciones e intercambios.
- no se puede afirmar que el tiempo de ejecución sea proporcional al número de operaciones dadas. Véase el caso del ordenamiento por selección.

4- CONCLUSIÓN

Noches en vela, tiempos de ejecución eternos —tan eternos que la paciencia no fue suficiente, hablamos de horas y horas—, subrutinas —de burbuja, inserción, rápidas y de Shell también, que no es concha en inglés, que es el señor Shell que inventó el algoritmo—, módulos, música al trabajar —buena y no tan buena, que te anima a seguir trabajando o te induce el sueño sin que tú puedas evitarlo—, partidas de póker entre medias y un larguísimo etcétera.

Todo esto es y ha sido parte del presente trabajo de Informática que nos disponemos a entregar en las próximas horas. Quizá la organización no fue la mejor, pero escribir la conclusión, después de leer el resto del documento, finalizado, resulta al menos satisfactorio.

Muchas horas se han dedicado a este proyecto y, ahora, podemos decir que estamos orgullosos con el resultado final. Se podría decir que la compilación y ejecución del programa han terminado con éxito.

Expandir nuestros conocimientos en cuanto a las ramas algorítmicas de la búsqueda y ordenación no solo nos ha ayudado a lo propio, sino que además nos ha proporcionado una biblioteca permanente a la que podremos acudir en cualquier momento en que nuestras ansias programadoras nos conduzcan a hacer uso de alguna de estas nuevas subrutinas que entre carpetas de nuestro ordenador se encuentran.

Agradecer por último al señor Zamecnik el encomendarnos la presente tarea y permitirnos proseguir por nuestro pequeño camino programador, a veces plagado de baches, pero siempre poblado de ganas e ilusión puestas en el siguiente reto.

5- BIBLIOGRAFÍA

Informática I: Fortran 90(Septiembre, 2010)
Fortran 95: programación multicapa para la simulación de sistemas físicos
Apuntes de clase de Informática, 2016
Hito 1: evolución de Fortran
https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda
https://en.wikipedia.org/wiki/Linear_search
https://en.wikipedia.org/wiki/Binary_search_algorithm
https://mnanjari.wordpress.com/algoritmos-de-busqueda-binaria/
https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento
https://en.wikipedia.org/wiki/Sorting_algorithm#Stability
https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
https://es.wikipedia.org/wiki/Ordenamiento_por_inserci%C3%B3n
http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort#Fortran
https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n
http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap08/sorting.f90
https://es.wikipedia.org/wiki/Quicksort
http://rosettacode.org/wiki/Sorting_algorithms/Quicksort
http://www.fortran.com/qsort_c.f95
https://en.wikipedia.org/wiki/Shellsort
http://rosettacode.org/wiki/Sorting_algorithms/Shell_sort#Fortran
https://www.quora.com/What-is-the-fastest-sorting-algorithm
https://www.slideshare.net/GrahamJohnson21/sorting-algorithms-presentation-70352257