0.0.1. Algoritmos para ordenar (Sort)

Uno de los problemas que provocó una de las mas ricas discusiones en la creación de algoritmos fue la búsqueda de métodos que sirvan para realizar programas que ordenen la información. Es decir, por ejemplo ordenar de mayor a menor (o al revés), una tabla a partir de una de sus columnas. Para simplificar el problema sólo trataremos el caso de ordenar las componentes de un vector. Los métodos más simples que permiten entender la complejidad del problema son el método de la Burbuja y el de Selección. Pero a la hora hora de ordenar grandes cantidades de información son desaconsejados por su lentitud e ineficiencia.

Método de Selección

Si la información se encuentra en un vector, el método de selección se basa en la idea de tomar el primer elemento del vector y compararlo contra todos los otros elementos. Si ordenamos de mayor a menor y este primer elemento es menor que el elemento contra el cual estoy comparando lo cambio por este. Una vez hecho este cambio sigo ahora comparando mi nuevo primer elemento contra los otros elementos restantes, continuando con la idea de que cada vez se cumpla con encontrar otro menor se vuelve a realizar el cambio. Para realizar el cambio del elemento se debe usar una variable temporaria (**temp** en el programa). Una vez terminada la comparación del primer elemento tomo el segundo y así continuo el proceso de comparación contra los elementos restantes. Este ciclo termina cuando comparo el anteúltimo elemento contra el último. Con este algoritmo se va coleccionando los elementos más grandes en los primeros elementos del vector.
El tiempo que tarda este procedimiento es proporcional a $t \sim N^2$ (donde N es la cantidad de elementos a ordenar). Por lo tanto, el tiempo crece muy rápido si se aumenta la cantidad de elementos a ordenar. Por ejemplo, si duplico la cantidad de elementos a ordenar tardaré 4 veces más, pero en el caso de tener 10 veces mas elementos tardaré 100 veces más en obtener un resultado útil. No es un método aconsejable para usar en una situación real, pero tiene utilidad académica, ya que sirve para entender los procesos y problemas involucrados en los algoritmos usados para ordenar.
El programa que realiza esta tarea sería el siguiente:

```
subroutine selec(n,a)
real*4 a(10000000)

do i=1,n-1
  do j=i+1,n

    if(a(i).lt.a(j)) then
      temp=a(i)
      a(i)=a(j)
      a(j)=temp
    endif

  enddo
enddo

return
end
```

Método de la Burbuja

El método de la burbuja se basa en la idea de comparar sólo entre pares cercanos e intercambiar los valores cuando cumplen con la condición deseada (mayor o menor valor según el orden que esté buscando). Este ciclo se tiene que repetir al menos n-1 veces si n es el número de elementos a ordenar. Pero se puede programar que si no se produce ningún cambio en uno de los ciclos (esto sucede cuando el vector ya quedó ordenado) que no se continue el proceso, ahorrando tiempo de la computadora. Para testar si dejó de haber cambios, uso la variable lógica **bandera** en el programa. Esta variable toma el valor .true. mientras se siguen haciendo cambios y .false cuando ya el vector está ordenado. En este último caso el programa termina.

El programa que realiza esta tarea sería el siguiente y es relativamente sencillo:

```
subroutine bubble(n,a)
real*4 a(10000000)
logical bandera

bandera=.true.


do while(bandera)
 bandera=.false.

 do i=1,n-1

  if(a(i).gt.a(i+1)) then
   temp=a(i)
   a(i)=a(i+1)
   a(i+1)=temp
   bandera=.true.
  endif

 enddo

enddo
return
end
```

Este método va con t $\sim$ N^2 y al igual que el método de selección no es para nada bueno, por no decir que es bastante malo, en lo que hace a su eficiencia. Se tiene como única ventaja que el algoritmo termina abruptamente si encuentra el vector ordenado y por lo tanto es mejor que el método de selección. Este último no sería capaz de descubrir que el vector ya está ordenado y repetiría todos los pasos hasta el final aún con un vector ordenado.

0.0.2. Otros Métodos

Hay otros métodos que son mejores. Veremos algunos de ellos, pero no con profundidad. Para el lector con mayor interés en el tema se puede consultar el libro Numerical Recipes (ya nombrado anteriormente). Este libro se puede consultar aquí. El texto consta de explicaciones de los algoritmos primero en forma teórica y luego práctica, en la cual se incluye la subrutina del método escrita

en Fortran. En este libro hay mucha información detallada con explicaciones y evaluaciones de la eficiencia de los métodos que nombraremos y otros mas.

Inserción Directa (Straight Insertion)

Este método viene del modo más simple de ordenar un mazo de cartas. Se toma la segunda baraja y se la compara con la primera, si hay que cambiarla, se la cambia. Se toma la tercera y se compara con la segunda y luego con la primera. Se sigue así hasta la última carta. Hay que notar que no es necesario comparar contra todas las cartas, solo hasta que se encuentre el lugar que le corresponde, lo que ahorra tiempo de computación. Este método va con N^2 (en el caso extremo) y se aconseja su uso para muestras pequeñas ($\sim N < 20$). Se lo utiliza como apoyo de otros algoritmos para ordenar subgrupos parciales de un grupo mayor, porque para pocos elementos es eficiente y rápido.

Método de las cáscaras (Shell´s Method)

Esta es en si una variante del método anterior. La idea es dividir la muestra en grupos de a dos, ordenar estos elementos en el grupo de a dos usando inserción directa. Y luego juntar en ahora grupos de 4 elementos ordenar y juntar en grupos del doble de elementos y repetir el proceso. Se puede mostrar que en promedio que en este método el tiempo va como $t \sim N^{1.25}$ al menos para $N < 60000$. El truco de este algoritmo es que se puede ordenar muy rápido grupos de pocos elementos y si junto grupos que ya están algo ordenados el algoritmo tiene pocas veces que cambiar elementos de lugar, es decir aumento la probabilidad de que un elemento ya esté en su lugar. Y por ello en promedios el método es más rápido que N^2.

Quicksort

Este método es el más rápido conocido para muestras muy grandes que se necesita ordenar, la idea es que la muestra se parte en dos pedazos y un elemento en particular (el a) es el seleccionado para separar estas dos particiones. Los elementos son chequeados en pares dejando en una muestra los elementos $\leq$ a la izquierda y en la otra los $\geq$ a la derecha, por lo tanto a queda en su lugar. El proceso se vuelve a repetir con la partición de la izquierda y luego con la de la derecha. Para realizar toda esta tarea el algoritmo requiere algo más de memoria, pero es extremadamente eficiente. Este método es algo mejor que otro método conocido como Heapsort que también muestra un excelente desempeño para grandes valores de N.

0.0.3. Probando métodos

Para probar los distintos métodos haremos un programa que ordene un vector de elementos al azar y mediremos los tiempo de cada algoritmo y su implementación. Pero hay que señalar un punto importante, los distintos métodos, tienen distintos resultado según las características de los datos de entrada. Puede haber datos desordenados al azar (como lo que vamos a usar en el ejemplo), datos con cierto grado de ordenamiento y datos que estén completamente invertidos en su distribución, es decir, quiero ordenar de menor a mayor pero los datos originales están ya ordenados de mayor a menor (o parcialmente ordenados). La eficiencia de los algoritmos puede en si cam-

Figura 1. Comparación como animación de distintos tipos de algoritmos para ordenar. Algunos los hemos detallado en el apunte, y otros no.

biar bastante según esta distribución de datos de entrada.

Para esto utilizaremos las rutinas del método de selección y el de la burbuja que hemos visto, y las demás implementaciones serán del Numerical Recipes. La tabla 1 muestra los tiempos que tarda en hacer los cálculos cada uno de los algoritmos descriptos sobre la misma muestra. En la figura 1 pueden verse una comparación de distintos métodos como animación en tiempo real de ellos.

Tabla 1. Tiempo que se tarda en ordenar un vector según el método

Algoritmo	10^5 números segundos	10^6 números segundos
Burbuja	36.07	3678.71
Selección	25.95	2547.27
Inserción Directa	6.56	646.6
Cáscaras	0.02	0.3
Heapsort	0.02	0.19
Quicksort	0.01	0.12

Tabla 2. Resultados obtenidos para una muestra de números al azar, la cual se la ordena con cada algoritmo y se le toma el tiempo. Cada una de estas implementaciones recibió exactamente la misma muestra. Note que los dos primeros métodos van N^2 y que al multiplicar por 10 la muestra, estoy tardando 100 veces más en correr el algoritmo (10^2 = 100).

0.0.4. Ordenar pares ordenados

Si tengo pares ordenados (X_i,Y_i) y por ejemplo los ordeno por los valores X_i. Es necesario que los pares ordenados no se destruyan o sea que cada X_i sigan asociados a su Y_i. Para resolver este problema deberé agregar a la subrutina no importa cual sea el método que cuando cambio el elemento X_i cambie también el Y_i, de esta manera:

```
⋮
IF(algo) then
temp=X(i)
X(i)=X(i+1)
X(i+1)=temp
temp=Y(i)
Y(i)=Y(i+1)
Y(i+1)=temp
⋮
```

El “algo” que aparece en el IF dependerá del algoritmo a utilizar.

0.0.5. Ordenar sin modificar el vector inicial

Una posibilidad es ordenar pero sin modificar el vector inicial y sin copiarlo a otro vector. Para hacer esta tarea creamos un vector de índices, donde el primer elemento tiene un 1, el segundo un 2 y así hasta los N valores que sean necesarios.
Si A es el vector a ordenar e I es vector de índices, puede referirse a los elementos de A como A(I(j)), es decir cada elemento de A, será el elemento i(j). Entonces en vez de ordenar el vector A, modifico los índices del vector I y de esta manera puedo mantener la configuración inicial de A y también la forma ordenada como A(i(j)).

Es decir haríamos:

```
⋮
IF(algo) then
temp=I(j)
I(j)=I(j+1)
I(j+1)=temp
⋮
```

Y el vector A no lo tocamos, sólo modificamos el vector de índices.

0.0.6. Subrutinas del sistema - getenv(), getargs() y systems()

Estas son llamadas a subrutinas del propio sistema operativo y son muy útiles.

GETARG() permite ingresar datos para las variables en el mismo renglón que se llama al programa, mientras que GETENV() permite obtener parámetros del ambiente del usuario (ejemplo, su nombre, directorio en el que está, etc).
Para GETARG() tenemos que dar como argumentos el número de la variable que voy a leer en la línea de programa y una variable de caracter donde quedará escrita esa variable.
En cambio para para GETENV() deberé indicar el nombre de la variable de ambiente que leo y una variable de caracter donde quedarán escritos esos datos.

Ejemplo:

```
program prueba
character name*32, argument*32

call getenv('USER',name)
call getarg(1,argument)

write(6,*) 'Username=',name
write(6,*) 'Argument=',argument
end
```

Y si compilo y luego corro el programa así:
./prueba 22
obtengo como resultado:

Username=carlos
Argument=22

CALL SYSTEM()

Esta llamada permite dar órdenes al sistema operativo con los comandos de texto como si fuesen escritos en una terminal. Saber usar esta llamada tiene muchas ventajas. Normalmente existen más de 3000 comandos en una instalación simple de LINUX. y muchos cubren la mayoría de los necesidades domésticas, como transformar archivos de formatos, etc. En resumen, un programa Fortran puede darle órdenes al sistema operativo.

Por ejemplo, si quiero que mi programa lea del directorio los archivos que terminen en ".dat" para después procesarlos haría:

```
⋮
CALL SYSTEM('ls *.dat > mis_archivos.txt')
OPEN(45, file='mis_archivos.txt')
⋮
```

Entonces en mis_archivos.dat tendría la información requerida.