ALGORITMIA DE LA ORDENACIÓN EFICIENTE
Necesidad de Ordenar Eficientemente
Análisis Algorítmico

Asignatura: Algoritmos y Estructuras de Datos II
Profesor: Mgter. Oscar Adolfo Vallejos.
Alumno: Arduino Guillermo Andrés

FACENA-UNNE
2011

ABSTRACT

Es sabido que los métodos de ordenación, son materia de estudio en las universidades, ya que permiten la comprensión de la lógica de programación, como así también el análisis de la complejidad y eficiencia de los mismos. Por ello este trabajo fue desarrollado en el marco del cursado de la Asignatura "Algoritmos y Estructuras de Datos II" en el segundo cuatrimestre de la carrera de Licenciatura en Sistemas de la Información de la Facultad de Ciencias Exactas y Naturales de la Universidad Nacional del Nordeste.

Este trabajo está orientado a lograr, el estudio de los algoritmos de ordenamiento, y los grados de eficiencia de cada uno de los algoritmos tratados.

A los fines indicados se tratará de conocer los métodos desde el más simple hasta el más complejo, que se han incluido en esta investigación, para ello se describirá cada método investigado y se analizará tanto su complejidad algorítmica vista desde un punto de vista teórico, como también en las comparaciones de tiempo de ejecución, requisitos para ejecutarlos, funcionalidades y alcance.

Arribando a conclusiones, basándose en la ejecución de una aplicación, codificada en Lenguaje Pascal y que permita medir el tiempo de los métodos de ordenamiento tratados en este trabajo.

INTRODUCCIÓN

En este Trabajo de Investigación se verán diferentes métodos de ordenamiento, codificados en Lenguaje Pascal. La importancia del Ordenamiento de datos, en el contexto del cursado de la asignatura Algoritmos y Estructuras de Datos II, reside básicamente en la compresión del mismo para analizar los algoritmos a observarse, con el fin de entender la eficiencia y optimización en su uso real.

Este informe permitirá conocer más a fondo cada método distinto de ordenamiento, desde uno simple hasta el más complejo dentro del contexto pedido en la Asignatura. Se realizaran comparaciones en tiempo de ejecución, pre-requisitos de cada algoritmo, funcionalidad y alcance.

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Haciendo foco en los métodos más populares, analizando el tiempo que demora y revisando el código, escrito en Pascal, de cada algoritmo

Entonces habiendo definido que es la ordenación, restaría definir entonces a que se considera eficiencia y optimización en el ordenamiento de los datos.

La optimización de un algoritmo se obtiene cuando se hace la elección correcta de algoritmo y estructura de datos, esto hace referencia al análisis cuidadoso de su desempeño para analizar las fallas y concebir mejoras antes de llevarlos a la ejecución de los mismos en una computadora.

Es así, en el caso de vectores pequeños (valor de N muy chico), los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy

frecuente. Sin embargo, en vectores grandes (valor de N considerable) estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.
Aunque es evidente ver que tanto el tipo y tamaño de los elementos como el dispositivo en donde se encuentran almacenados pueden influir en el método que utilicemos para ordenarlos, en este tema se soluciona el caso en que los elementos son números enteros y se encuentran almacenados en un vector, como se ha explicado anteriormente.
A continuación se detallan cada uno de los métodos realizados en este trabajo.

MÉTODO DE LA BURBUJA

```
Procedure OrdenarBurbuja;
Begin
   clrscr;
   For i:=1 to va_n - 1 do →Recorre el vector
    Begin
       For j:=1 to va_n - 1 do →Recorre el vector ordenando
        Begin
            If ( vv_vector[j] > vv_vector[j+1] ) Then → Pregunta para ordenar de
menor a mayor
             Begin
                va_aux:=vv_vector[j];
                vv_vector[j]:=vv_vector[j+1];
                vv_vector[j+1]:=va_aux;
             End;
        End;
   End;
```

MÉTODO DE SELECCIÓN

```
Procedure OrdenarSeleccion;
Begin
   clrscr;
  For i:=1 to va_n - 1 do →Recorre n-1 pasadas el vector
   Begin
      va_menor:=vv_vector[i];
      k:=i;
      For j:=i+1 to va_n do →Recorre el vector ordenando
      Begin
          If (vv_vector[j] < va_menor ) Then
           Begin
              va_menor:=vv_vector[j]; →  Obtiene el menor valor
              k:=j;
           End;
      End;
      vv_vector[k]:=vv_vector[i];
      vv_vector[i]:=va_menor; → Sitúa el elemento más pequeño en vv_vector[i]
   End;
```

```
End;
```

MÉTODO POR INSERCIÓN

```
Procedure OrdenarBaraja;
Begin
    clrscr;
    For i:=2 to va_n do →Recorre el vector
    Begin
        va_aux:=vv_vector[i];
        j:=i-1;
        While ((vv_vector[j] > va_aux) and (j>1)) do → Explora las sub-listas
        Begin
            vv_vector[j+1] := vv_vector[j];
            j:=j-1; → Se mueve buscando la posición correcta de inserción
        End;
        if (vv_vector[j] > va_aux) then → Realiza el Intercambio Insertando el
elemento
        Begin
            vv_vector[j+1] := vv_vector[j];
            vv_vector[j]:=va_aux;
        End
          Else
            vv_vector[j+1]:=va_aux;
End;
```

MÉTODO SHELL

```
Procedure Shell;
var
   va_cambios,va_aux: Integer;
   i,va_salto:longint;
Begin
 clrscr;
 va_salto:=va_n; → La variable salto toma el valor de la longitud del vector
While not(va_salto=0) Do
  Begin
   va_cambios:=1;
   While not(va_cambios=0) Do  → Si no hay cambios
    Begin
     va_cambios:=0;
     For i:= (va_salto+1) to va_n do      → Recorre el vector
      Begin
       If (vv_vector[i-va_salto]>vv_vector[i]) Then →Inicia el ordenamiento
        Begin
         va_aux:=vv_vector[i];
         vv_vector[i]:=vv_vector[i-va_salto];
         vv_vector[i-va_salto]:=va_aux;
        inc(va_cambios); → Activa la bandera
```

```
      End;
    End;
  End;
va_salto:= va_salto div 2; → Se divide el vector original en dos
  va_salto:= va_salto  div 2; →vuelve a calcular el salto
 End;
End;
```

MÉTODO QUICK SORT

```
Procedure Quick_Sort(VAR  vv_vector:array of integer ; izquierda,derecha:longint);
Var
pivot,i,j :longint;
va_aux :integer;
Begin
i:=izquierda;
j:=derecha;
pivot:=vv_vector[(i+j) DIV 2]; → Selecciona el Pivot
While i<=j Do
Begin
        While vv_vector[i]<pivot Do inc(i); → Recorre a la derecha
        While vv_vector[j]>pivot Do dec(j); → Recorre a la izquierda
        IF ( i <= j) THEN → Comienza a ordenar de acuerdo al pivot
         Begin
            va_aux:=vv_vector[i];
            vv_vector[i]:=vv_vector[j];
             vv_vector[j]:=va_aux;
             inc(i);
             dec(j);
        End;
End;
If (derecha> i ) Then
Quick_Sort(vv_vector,i,derecha); → ordena recursivamente los elementos ubicados a
la derecha
If (izquierda < j ) Then
Quick_Sort(vv_vector,izquierda,j); → ordena recursivamente los elementos ubicados
a la Izquierda
End;
Procedure QuickS;
Begin
clrscr;
Quick_Sort(vv_vector,0,va_n-1); → Llama al procedimiento recursivamente hasta
ordenar el vector.
End;
```

MÉTODO COUNTING SORT

```
Procedure Counting_Sort (Var vv_vector: Array of Integer; min, max: longint);
```

```
Var
Contador: Array of Integer;   →Vector auxiliar
i, j, z : longint;
Begin
clrscr;
SetLength(Contador, max-min);  → Minimo y Maximo del vector auxiliar
 For i := 0 to (max-min) do
  Contador[i] := 0;
  For i := 0 to (va_n-1) do
  Contador[ vv_vector[i] - min ] := Contador[ vv_vector[i] - min ] + 1;  → Recorre el
vector y obtiene la lista en forma ordenada
  z := 0;
   For i := min to max do
   For j := 0 to (Contador[i - min] - 1) do
   Begin
      vv_vector[z] := i;
       z := z + 1
   End;
End;
```

COMPLEJIDAD ALGORÍTMICA

En este apartado se intentará ver la complejidad algorítmica de cada método desde el punto de vista teórico, no se pretende hacer un análisis exhaustivo pero si pretender mostrar como es la complejidad en cada método anteriormente expuesto.

Método Burbuja

Al algoritmo de la burbuja, para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones:

$$\mathbf{c(n)} = \frac{(\mathbf{n^2 - n})}{\mathbf{2}}$$

Esto es, el número de comparaciones **c(n)** no depende del orden de los términos, si no del número de términos. $\theta\big(\mathbf{c(n)}\big) = \mathbf{n^2}$

Por lo tanto la cota ajustada asintótica del número de comparaciones pertenece al orden de n cuadrado.

Cuando una lista ya está ordenada, 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.

Método Selección

Al algoritmo de ordenamiento por selección, para ordenar un vector de **n** términos, tiene que realizar siempre el mismo número de comparaciones:

$$\mathbf{c(n)} = \frac{(\mathbf{n^2 - n})}{\mathbf{2}}$$

Esto es, el número de comparaciones **c(n)** no depende del orden de los términos, si no del número de términos.

$\theta\big(\mathbf{c(n)}\big) = \mathbf{n^2}$

Por lo tanto la cota ajustada asintótica del número de comparaciones pertenece al orden de n cuadrado

Método Inserción

En el peor caso este método toma tiempo O(n2). Un punto interesante es que el mejor caso, ocurre cuando el arreglo se encuentra ordenado, en ese caso el tiempo es O(n). Requiere $O(n^2)$ operaciones para ordenar una lista de n elementos.
El algoritmo se aprovecha del posible orden parcial que pueda existir entre los elementos. Es decir, si un elemento está "cerca" de su posición definitiva, el algoritmo hace menos intercambios. El caso extremo es que el array esté totalmente ordenado... en cuyo caso, el algoritmo nunca entra en el bucle interior y por eso su complejidad en este caso el mejor es $\Omega(n)$. En este caso, la cota inferior de la complejidad temporal baja hasta una función lineal.

Método Shell

Con esto, el método de ordenación por Incrementos consiste en hacer *h*-ordenaciones de *a* para valores de *h* decreciendo hasta llegar a uno. Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso, esto es mejor que las $O(n^2)$ comparaciones requeridas por algoritmos simples pero peor que el óptimo $O(n \log n)$.
El Shell Sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez

El algoritmo Shell Sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga saltos mayores hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

Método Quick Sort

El procedimiento *Quick Sort* descarta la filosofía de caso mejor, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del vector, sino de la elección del pivote.
Dado que Quick-Sort particiona el arreglo en dos y el procedimiento de partición tarda tiempo lineal con respecto al largo total del arreglo, podemos decir que el tiempo que le toma a Quick-Sort está regido por la relación de recurrencia

$T(n) = T(p) + T(n - p) + c \cdot n$

$T(1) = c$

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 sub-listas de igual tamaño. En este caso, el orden de complejidad del algoritmo es **$O(n \cdot \log n)$**.
- 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)**.

Método Counting Sort

En el ordenamiento por conteo se asume que :

- Los n elementos de entrada son enteros en el rango 0 y k.
- Si k = O(n), el algoritmo corre en tiempo O(n).

Cuál es el costo de ejecución del algoritmo Counting Sort?
Tiempo de ejecución: |O{(zk}) +|O{(zn}) +|O{(zk}) +|O{(zn}) = O(k + n)
Si k = O(n), entonces tiempo de ejecución es O(n)
Una importante propiedad del algoritmo Counting Sort es la Estabilidad.
Este método tiene la restricción de que sólo puede ser aplicado en elementos en un intervalo de a lo más *k* elementos, típicamente en números naturales en el rango de 1 a *k*. Su cota de tiempo es de O(*n* + *k*).

Comparación Empírica de los Tiempos de Ejecución

En el lapso de tiempo comprendido entre el 08/09/2011 y el 19/11/2011; en el marco de la Asignatura "Algoritmos y Estructuras de Datos II" de la Carrera Licenciatura en Sistemas de la Información, dependiente de la Facultad de Ciencias Exactas y Naturales de la UNNE. Se realizó el proyecto de una aplicación codificada en Lenguaje Pascal, que comprendiera los métodos de ordenación: Burbuja, Selección, Inserción (Baraja), Shell Sort, Quick Sort y Counting Sort; teniendo en cuenta su medición en segundos.

Método utilizado

Se realizó la carga del vector en forma aleatoria, procediéndose a la ordenación del mismo, repitiéndose el proceso hasta obtener una muestra de 10 pruebas. Luego de esto se verificó los tiempos obtenidos, obteniéndose el mejor tiempo de ejecución, el peor y el tiempo promedio; correspondiéndose esto con el caso peor, el caso mejor y el caso promedio, ya que al ser al azar la carga del vector, se comprueba que cuando consigue el mejor tiempo es cuando el vector esta totalmente ordenado, y en los otros casos el vector ha estado parcialmente desordenado y totalmente desordenado.

Prueba de Eficiencia en tiempo y por cantidad de elementos a ordenar

256 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor	512 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor
Burbuja	0,2180	0,2189	0,2190	Burbuja	0,4530	0,4562	0,4690
Selección	0,2030	0,2140	0,2340	Selección	0,4370	0,4376	0,4380
Inserción	0,2030	0,2094	0,2190	Inserción	0,4210	0,4313	0,4380
Shell Sort	0,1250	0,1250	0,1250	Shell Sort	0,1410	0,1410	0,1410
2048 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor	4096 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor
Burbuja	1,8590	1,8795	1,9060	Burbuja	3,7650	3,8047	3,8600
Selección	1,7340	1,7861	1,8290	Selección	3,5460	3,6314	4,0160
Inserción	1,7340	1,7563	1,7660	Inserción	3,5000	3,5248	3,5470

Shell Sort	0,1720	0,1720	0,1720	Shell Sort	0,1870	0,1875	0,1880
16384 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor	65536 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor
Burbuja	16,7340	17,0360	17,1250	Burbuja	84,6720	92,6376	98,5000
Selección	13,6560	14,7827	16,6720	Selección	59,4680	63,7532	65,0160
Inserción	14,2810	14,3766	14,5000	Inserción	59,2810	61,0328	64,6100
Shell Sort	0,2180	0,2236	0,2350	Shell Sort	0,5310	0,5703	0,6250
250000 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor	500000 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor
Quick Sort	0,0310	0,0313	0,0320	Quick Sort	0,0620	0,0623	0,0630
Counting Sort	0,0000 *	0,0000*	0,0000*	Counting Sort	0,0000*	0,0000*	0,0000*
750000 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor	1000000 ELEMENTOS	Tiempo Mejor	Tiempo Promedio	Tiempo Peor
Quick Sort	0,0930	0,0935	0,0940	Quick Sort	0,1245	0,1376	0,1570
Counting Sort	0,0150	0,0152	0,0160	Counting Sort	0,0150	0,0157	0,0160

*: Limitación encontrada por la rapidez del método para ser tomado el tiempo por Pascal con función Gettickcount

CONCLUSIÓN

Según lo observado en la experimentación se puede concluir que, en las condiciones anteriormente mencionadas, el método de Burbujas es aquél con mayor tiempo de ejecución a medida que el valor de N se incrementa, con lo cuál se ve que es el más ineficiente de todos, pero cabría cuestionarse aquí porque es el método más utilizado, y la respuesta estaría en la fácil comprensión del mismo, tanto en la codificación como en su forma de ordenación. Si bien con los métodos de Inserción y Selección se nota una mejora en los tiempos de ejecución, es el método Shell quien presenta el mejor rendimiento debido a que es una modificación de los métodos de Inserción y Selección, ya que realiza "saltos grandes" que permiten ordenaciones múltiples. Esto se ve claramente demostrado en las pruebas de tiempo.

Ahora bien todo cambia cuando hablamos de los métodos Quick Sort y Counting Sort, claramente se observa que ambos en prueba de tiempo son altamente eficientes y rápidos en comparación al resto. Pero hay que hacer una salvedad, dándose cuenta la propia limitación que posee el Counting Sort de que solamente ordena números enteros positivos, y lo favorable que es el Quick Sort que puede ordenar cualquier elemento que integre la lista o el vector. El otro tema a tener en cuenta es que para obtener mediciones en tiempo fiables con estos métodos debemos hablar de un valor de N considerable, es decir, vectores con una carga de elementos superior a los 250000.

Por último, el tema de la codificación, si bien se puede ver que el más fácil de codificar es el Burbuja y en apariencia el más difícil es el Quick Sort, la conceptualización en ambos ayuda a entenderlos a la hora de llevarlos a lenguaje Pascal, por lo que se puede decir que en realidad el mas dificultoso a la hora de

concretar el código ha sido el método Shell, ya que hay que comprender que Salto debe asumir el valor de N, de lo contrario no se producirán todas la iteraciones necesarias, y en el mejor de los casos si este error se produjera el vector sería ordenado sólo en la primera mitad y el resto quedaría desordenado.

BIBLIOGRAFÍA

- http://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento.Fecha de Consulta: 20/11/2011
- http://es.wikipedia.org/wiki/Ordenamiento_por_cuentas. Fecha de Consulta: 20/11/2011
- Jorge Pérez **Ph**. D. Student Department of Computer Science. Pontificia Universidad Católica de Chile http://ing.utalca.cl/~jperez/ae/documentos/ordenacion.pdf. Fecha de Consulta: 20/11/2011
- ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA CAPITULO 6 **h**ttp://novella.mhhe.com/sites/dl/free/844814077x/619434/A06.pdf. Fecha de Consulta: 3/10/2011
- Julio César López. Ordenamiento en Tiempo Lineal. http://ocw.univalle.edu.co/ocw/ingenieria-de-sistemas-telematica-y-afines/fundamentos-de-analisis-y-diseno-de-algoritmos/material/ordenamiento2.pdf Fecha de Consulta: 19/11/2011
- Teoría de Algoritmos *Tema 1: Análisis de Eficiencia de Algoritmo*. *Departamento de Informática Universidad de Jaén*. http://wwwdi.ujaen.es/asignaturas/T_Algoritmos/documentos/TAlg_Tema_I_%20Analisis%20de%20Eficiencia%20de%20Algoritmos.pdf. Fecha de Consulta: 18/11/2011
- Arturo Díaz Pérez. Análisis y Complejidad de Algoritmos. http://es.scribd.com/doc/12397882/Metodos-de-to. Fecha de Consulta: 3/10/2011
- Javier Gutiérrez, Michael González. Parte II: Estructuras de Datos y Algoritmos. UNIVERSIDAD DE CANTABRIA FACULTAD DE CIENCIAS. http://www.ctr.unican.es/asignaturas/lan/algoritmos-2en1.pdf Fecha de Consulta: 03/11/2011
- Luis E. Vargas Azcona. Problemas y Algoritmos. http://lobishomen.files.wordpress.com/2011/01/libropre3.pdf. Fecha de Consulta: 15/11/2011
- Fernando A. Lagos B. Algoritmos de Ordenamiento Informe de Ordenamiento. Año 2007. http://blog.zerial.org/ficheros/Informe_Ordenamiento.pdf. Fecha de Consulta: 03/10/2011
- Prof. Liliana Caputo. Presentación Número 12- Lenguajes Formales- Lógica y Matemática Computacional- Lic. en Sistemas de la Información- FACENA UNNE. Año 2010