

Estructura de datos

**Unidad V**

Métodos de ordenamiento: Ordenamiento Shell Sort

**Horario:** 14:00-15:00

**Profesor:** Luis Bernardo Ballesteros Barradas

Equipo 5:

**Integrantes:**

García Rivera Oscar Iván

Martínez Solís Karla Beatriz

Romero Enríquez Eduardo Daniel

**INDICE**

Introducción 3

Ordenamiento Shell sort 4

1era Secuencia de intervalos (Gonnet y Baeza-Yates) 9

2da Secuencia de intervalos (Steven Pigeon) 10

Características 13

Conclusión 15

Bibliografía 16

**INTRODUCCIÓN**

El estudio de algoritmos de ordenamiento tiene una gran importancia dentro de la Ciencia de la Computación, pues una buena cantidad de los procesos realizados por medios computacionales requieren que sus datos estén ordenados. Además, el hecho de almacenar los datos de manera ordenada permite implementar algoritmos de búsqueda muy rápidos (por ejemplo: búsqueda binaria). Esta y muchas otras razones de fin práctico impulsaron el estudio y la búsqueda de algoritmos de ordenamiento eficientes. Desde los comienzos del uso de computadores se conocían algoritmos que resolvían el problema en tiempo cuadrático respecto del tamaño del problema, pero eran rutinas muy sencillas y lentas. El algoritmo de ordenamiento Shellsort fue publicado en 1959 por Donald L. Shell, y fue uno de los primeros en romper la barrera del orden cuadrático, aunque esto en realidad se probó un par de años después de su publicación.

El objetivo de este estudio es demostrar empíricamente que implementar Shellsort con series de pasos dependientes del tamaño del arreglo puede llegar a ser mucho más eficiente que con las series clásicas, las cuales son independientes del tamaño del arreglo, pero hay que aplicar una optimización sencilla para obtener buenos resultados: todos los pasos de la serie deben ser números impares. Además, este estudio muestra que dada la simplicidad de programación del algoritmo, su buen peor caso y caso promedio, y su ejecución “in place”, es decir, no necesita espacio adicional para realizar el ordenamiento del arreglo, lo hacen un buen candidato para resolver el problema de ordenamiento cuando la cantidad de elementos a ordenar no es muy grande (menos de 100.000 lementos).

**Ordenamiento Shell. (Shell sort)**

Este algoritmo de ordenación fue creado por **Donald Shell**, el algoritmo se denomina Shell en honor a su inventor. El algoritmo se parece al algoritmo de ordenación por inserción. En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda de uno en uno, pero con el algoritmo de Shell la comparación se hace con intervalos mayores a uno, logrando con ello que la ordenación sea más rápida. Generalmente se toma como intervalo inicial n div 2, siendo n la cantidad de elementos de la lista a ordenar, luego se reduce los intervalos a la mitad hasta que el intervalo llegue a ser uno. Cuando la ordenación de la lista se hace con un intervalo de 1 el algoritmo se comporta como el algoritmo de inserción, pero con la ventaja de que al tener una lista casi ordenada, debido a los ordenamientos por intervalos anteriores, el ordenamiento se hará más rápido.

Veremos un ejemplo ordenando de menor a mayor (ascendentemente) la siguiente lista de números:

7, 3, 10, 1, 9, 8, 4

La lista tiene 7 elementos (de 0 a 6), con lo cual obtendremos un intervalo inicial de 3, división entera de 7 entre 2 (7 div 2). Desde el elemento 3 se ordena la lista por inserción, hacia la izquierda tomando los elementos de 3 en 3, y así hasta terminar de recorrerla lista.

**1er recorrido**: intervalo 3, resultado de la división entera de 7 entre 2.

|  |  |  |
| --- | --- | --- |
| Desde el elemento 3. Elementos a ordenar: 7, 1 |  | Se colocan los elementos ordenados por inserción |
|  |  |  |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **7** |  |  | **1** |  |  |  | | 7 | 3 | 10 | 1 | 9 | 8 | 4 | |  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **1** | 3 | 10 | **7** | 9 | 8 | 4 | |

|  |  |  |
| --- | --- | --- |
| Desde el elemento 4. Elementos a ordenar: 3, 9 |  | Se colocan los elementos ordenados por inserción |
|  |  |  |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | **3** |  |  | **9** |  |  | | 1 | 3 | 10 | 7 | 9 | 8 | 4 | |  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | **3** | 10 | 7 | **9** | 8 | 4 | |

|  |  |  |
| --- | --- | --- |
| Desde el elemento 5. Elementos a ordenar: 10, 8 |  | Se colocan los elementos ordenados por inserción |
|  |  |  |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  | **10** |  |  | **8** |  | | 1 | 3 | 10 | 7 | 9 | 8 | 4 | |  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 3 | **8** | 7 | 9 | **10** | 4 | |

|  |  |  |
| --- | --- | --- |
| Desde el elemento 6. Elementos a ordenar: 1, 7, 4 |  | Se colocan los elementos ordenados por inserción |
|  |  |  |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **1** |  |  | **7** |  |  | **4** | | 1 | 3 | 8 | 7 | 9 | 10 | 4 | |  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **1** | 3 | 8 | **4** | 9 | 10 | **7** | |

**2do recorrido**: intervalo 1, resultado de la división entera de 3 entre 2.

La lista se ordena con un intervalo de 1, es decir desde el elemento 1 hasta el último elemento de la lista. En esta parte elalgoritmo se comporta como en el algoritmo de inserción y la lista se ordena haciendo menos intercambios.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 7 | 8 | 9 | 10 |

Los recorridos terminan cuando el intervalo sea mayor que 0. Ahora veamos la primera implementación del algoritmo con el siguiente procedimiento:

|  |  |
| --- | --- |
|  | **PROCEDURE** **Ordenar**(A:LNaturales);  **VAR** i,j:longint;  aux:qword;  intv:longword; *//intervalo*  **BEGIN**  intv:=**length**(A) **DIV** **2**; *//intervalo inicial*  **WHILE** intv>**0** **DO** *// Mientras intervalo >0*  **BEGIN**  *//algoritmo de inserción, por intervalos*  **FOR** i:=intv **TO** **high**(A) **DO** *// Desde el intervalo hasta el final*  **BEGIN**  aux:=A[i];  j:=i-intv;  **WHILE** ((j>=**0**) **AND** (auxA<[j])) **DO**  **BEGIN**  A[j+intv]:=A[j];  j:=j-intv  **END**;  A[j+intv]:=aux  **END**;  *//nuevo intervalo*  intv:=intv **DIV** **2**  **END**;  **END**; |
|  | *Código fuente 5: Primer algortimo de ordenación Shell.* |

Tal como se puede observar el bucle while es el que controla la cantidad de intervalos a ordenar, y el resto viene a ser el mismoalgoritmo de inserción adaptado para que se usen los intervalos que se van obteniendo. La variable j que es la clave de estealgoritmo se va disminuyebdo de intervalo en intervalo (intv), en vez de hacerlo de 1 en 1 como en el algoritmo de inserción.

Este algoritmo también se puede adaptar para que no use la evaluación booleana en cortocircuito, a continuación el procedimiento correspondiente:

|  |  |
| --- | --- |
|  | **PROCEDURE** **Ordenar**(A:LNaturales);  **VAR** i,j:longword;  aux:qword;  intv:longword; *//intervalo*  IniLista:Boolean;  **BEGIN**  intv:=**length**(A) **DIV** **2**; *//intervalo inicial*  **WHILE** intv>**0** **DO** *// Mientras intervalo >0*  **BEGIN**  *//algoritmo de inserción, por intervalos*  **FOR** i:=intv **TO** **high**(A) **DO** *// Desde el intervalo hasta el final*  **BEGIN**  j:=i-intv;  aux:=A[i];  IniLista:=**FALSE**;  **WHILE** (**NOT** IniLista) **AND** (aux<A[j]) **DO**  **BEGIN**  A[j+intv]:=A[j];  **IF** j>=intv **THEN** j:=j-intv **ELSE** IniLista:=**TRUE**  **END**;  **IF** IniLista **THEN** A[j]:=aux  **ELSE** A[j+intv]:=aux  **END**;  *//nuevo intervalo*  intv:=intv **DIV** **2**  **END**;  **END**; |
|  | *Código fuente 6: Segundo algoritmo Shell, sin el uso de de evaluación booleana en cortocircuito.* |

Este algoritmo de Ordenamiento Shell, tiene otras variantes, que se diferencian en la generación de la secuencia de intervalos. Originalmente la secuencia de intervalos es ir dividiendo con 2 los intervalos hasta llegar a 1, tomando como primer intervalo la longitud de la lista, pero cualquier secuencia de intervalos funcionará con este algoritmo siempre y cuando esta termine en 1. A continuación dos secuencias de intervalos que ofrecen un mejor rendimiento y de fácil implementación:

**1era Secuencia de intervalos (Gonnet y Baeza-Yates)**

Esta secuencia de intervalos consiste en tomar la longitud de la lista multiplicarlo por 5 y después hacer una división entera con 11, para el siguiente intervalo, tomar el anterior multiplicarlo por 5 y hacer una división entera con 11, y así sucesivamente hasta llegar a 1. Sólo se debe tener en cuenta que cuando el intervalo es 2 este debe cambiarse siempre a 1.Por ejemplo si tenemos una lista de 100 elementos a ordenar, los intervalos serían. 45, 20, 9, 4, 1; pero en el caso de 150 elementos los intervalos serán: 68, 30, 13, 5, 1. Como puede observar del 5 se salto al 1, ignorando el 2 que es resultado de (5\*5) div 11. Esto es por lo siguiente: Si intentáramos obtener el intervalo considerando el 2 obtendríamos 68, 30, 13, 5, 2, 0; y eso estaría muy mal porque la secuencia de intervalos no termina en 1 sino en cero, es por eso que se debe cambiar el 2 por el 1, sólo cuando este aparezca. Ejemplo:

|  |  |
| --- | --- |
|  | **PROCEDURE** **Ordenar**(A:LNaturales);  **VAR** i,j:longint;  aux:qword;  intv:longword; *// intervalo*  **BEGIN**  intv:=(**length**(A)\***5**) **DIV** **11**; *//intervalo inicial*  **WHILE** intv>**0** **DO** *// Mientras intervalo > 0*  **BEGIN**  *//algoritmo de inserción, por intervalos*  **FOR** i:=intv **TO** **high**(A) **DO** *// Desde el intervalo hasta el final*  **BEGIN**  j:=i;  aux:=A[i];  **WHILE** ((j>=intv) **AND** (auxA<[j-intv])) **DO**  **BEGIN**  A[j]:=A[j-intv];  j:=j-intv  **END**;  A[j]:=aux  **END**;    *// nuevo intervalo*  intv:=(intv \* **5**) **DIV** **11** ;  **IF** intv=**2** **THEN** intv:=**1**  **END**;  **END**; |
|  | *Código fuente 7: Algoritmo Shell, Gonnet y Baeza-Yates.* |

**2da Secuencia de intervalos (Steven Pigeon)**

Esta secuencia de intervalos está basado en la siguiente secuencia matemática: an=1+en-2, en donde, e es el número de Euler y n es cualquier número entero positivo mayor que 0, es decir n=1, 2 , 3, 4, 5, 6 ... ∞. A todos los valores de esta secuencia matemática se debe redondear antes de obtener la parte entera, que se usará comointervalo. Para una lista de 150 elementos los intervalos con dos cifras decimales sin redondear serían: 1.36, 2.00, 3.71, 8.38, 21.08, 55.59, 49.41, redondeando obtendremos: 1, 2, 4, 8, 21, 56, 149. El ordenamiento shell ordena la lista usando los intervalos desde el mayor hasta 1, por lo que se necesita obtener el intervalo an que sea menor que la longitud de la lista, para ello despejamos n de la siguiente ecuación: a=1+en-2, en donde a es la longitud de la lista, con lo que obtenemos la siguienteecuacion: n=ln(a-1)+2, (ln=logaritmo neperiano) el valor devuelto por la ecuación, debe ser truncado para obtener un valor entero que nos permite obtener el intervalo an menor a la longitud de la lista. Hay que tener presente que si la lista tiene longitud de 1 entonces al obtener el valor de n, se intentará obtener el ln(0), esto producirá un error, que se puede evitar con una estructura de control **if-then-else** al inicio del procedimiento. En donde se evita ordenar la listas cuando este es de un sólo elemento.

|  |  |
| --- | --- |
|  | **PROCEDURE** **OrdenarD**(A:LNaturales);  **VAR** i,j:longint;  aux:qword;  intv:longword; *// intervalo*  n:longword;  **BEGIN**    *//intervalo inicial*  intv:=**0**;  **IF** **length**(A)>**1** **THEN** *// Debe ser mayor que 1*  **BEGIN**  n:=**trunc**(**ln**(**length**(A)-**1**)+**2**);  intv:=**trunc**((**1**+**exp**(n**-2**))+**0.5**);  **END**;    **WHILE** intv>**0** **DO** *// Mientras intervalo > 0*  **BEGIN**  *//algoritmo de inserción, por intervalos*  **FOR** i:=intv **TO** **high**(A) **DO**  **BEGIN**  j:=i;  aux:=A[i];  **WHILE** ((j>=intv) **AND** (auxA[j-intv])) **DO**  **BEGIN**  A[j]:=A[j-intv];  j:=j-intv  **END**;  A[j]:=aux *//A[j+inc]:=aux*  **END**;    *//nuevo intervalo*  n-=**1**;  **IF** n>**0** **THEN** intv:=**trunc**((**1**+**exp**(n**-2**))+**0.5**)  **ELSE** intv:=**0**  **END**;  **END**; |
|  | *Código fuente 8: Algoritmo Shell, Steven Pigeon.* |

En este algoritmo se puede observar que se comprueba que la longitud de la lista sea mayor que 1, en caso no lo sea la variableintv tendrá el valor 0, evitando ordenar la lista. Cada vez que se obtiene el nuevo intervalo se debe asegurar que n no llegue a valer 0, ya que si obtenemos un valor 0 para n, obtendremos un intervalo de 1, y si se sigue disminuyendo n obtendríamos un error de desbordamiento de punto flotante.

Los algoritmos mencionados anteriormente ordenan la listas de menor a mayor (ascendente), en caso quisiéramos hacerlo de mayor a menor (Descendente), sólo basta con cambiar la condición, aux<A[j-intv] por aux>A[j-intv].

**CARACTERÍSTICAS**

* se trata de un algoritmo de **ordenación interna**. Al igual que cualquier otro de ordenación interna (los datos están en memoria principal) podría utilizarse para ordenación externa (en memoria secundaria) siempre y cuando se disponga de acceso aleatorio, pero el algoritmo no fue ideado para eso.
* Se basa en comparaciones e intercambios.
* Necesita que el tiempo de acceso a cualquier dato sea constante (es decir, fue ideado para trabajar con arrays, arrays de referencias o punteros, etc...). Ojo a otras estructuras, como listas enlazadas, etc... ya que en ese caso, el tiempo de acceso a un elemento no es constante, depende de la posición del elemento.
* **No es estable**: dados dos elementos que al compararlos sean "iguales" -es decir, que pueden ir indistintamente en la misma posición, no mantienen necesariamente el orden relativo inicial entre ellos.
* El estudio de su complejidad no es trivial, sino todo lo contrario. La implementación original de Shell tiene una complejidad en el peor caso de O(n2), aunque en un caso promedio o en casos típicos comprobados empíricamente, los resultados son mucho mejores que con la burbuja, selección directa o inserción directa, cuya complejidad en el peor caso también es del orden de O(n2).
* Sin embargo, optimizaciones posteriores han logrado reducir esa cota... Por ejemplo, con la optimización de [Robert Sedgewick](http://en.wikipedia.org/wiki/Robert_Sedgewick_(computer_scientist))(external link) se llega a O(n4/3), y con la propuesta por [Vaughan Pratt](http://en.wikipedia.org/wiki/Vaughan_Pratt" \t "_blank)(external link) se llega al orden de O(n log2n).
* En 1992, [Greg Plaxton, Bjorn Poonen y Torsten Suel](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.1393)(external link) prueban que es posible incluso rebajar aún más esas cotas.
* En cierto modo, puede considerarse una ampliación del algortimo de [inserción directa](http://latecladeescape.com/t/inserci%C3%B3n+directa), con lo cual, conviene tenerlo claro antes de meterse con el de Shell.
* **No** es un algoritmo **en-línea**[**?**](http://latecladeescape.com/t/tiki-editpage.php?page=algoritmo+on-line).

***VENTAJAS:***

• Eficiente para conjuntos de elementos medianos (típicamente menores a 1000 elementos)

* No requiere memoria adicional.
* Mejor rendimiento que el método de Inserción clásico

• ***DESVENTAJAS:***

• Complejo para analizar y no se acerca a la eficiencia de merge, heap y quick sorts.

* Implementación algo confusa.
* Realiza numerosas comparaciones e intercambios.

**CONCLUSIÓN**

Si bien Shellsort no es el algoritmo más eficiente para ordenar arreglos, comparado con la complejidad ))ln(\*( *nnO* de los algoritmos Quicksort, Mergesort y Heapsort, es un algoritmo mucho más fácil de programar. Su simplicidad radica en que deriva del algoritmo más simple para ordenar, Insert Sort. Además, su complejidad promedio en tiempo de *nO* ocupando la serie complejidad en espacio de )(*nO* , debido a que opera “in place”, lo hacen un buen candidato para resolver el problema de ordenamiento en conjuntos de menos de 100.000 elementos.

Es vital para la eficiencia del algoritmo que todos los elementos de la serie de pasos sean números impares, para lo cual basta con restarle 1 al paso si éste es par. Con esta pequeña modificación se reduce el tiempo promedio de ejecución y su varianza. Además, el estudio demuestra empíricamente que algunas series dependientes del tamaño del arreglo reducen el tiempo de ejecución del algoritmo con respecto a las series clásicas. Sin embargo, aún no se sabe con certeza cuál es la eficiencia real del algoritmo, y es muy posible que existan series de pasos que reduzcan los tiempos de ejecución obtenidos

en los tests descritos.

**BIBLIOGRAFÍA**

<http://webcache.googleusercontent.com/search?q=cache:tkXpY4x7atgJ:www.cimat.mx/~alram/comp_algo/clase15.pdf+&cd=6&hl=es&ct=clnk&gl=mx>

<http://elvex.ugr.es/decsai/algorithms/slides/problems/Sorting.pdf>

<https://sites.google.com/site/estdatjiq/home/unidad-v>

<http://estructura-de-datos-itsav.blogspot.mx/2012/03/613-shell-sort-ordenacion.html>

<http://estructuras-de-datos.wikispaces.com/ordenamiento+shell+sort>