Algoritmos de ordenación

Sebastián Gurin (Cancerbero)

Revision History

Revision 1 30 de noviembre de 2004 Revised by: Cancerbero

Sobre la licencia de este documento

Sobre la licencia del código fuente que aparece en este documento.

El código de los algoritmos que aparecen en este documento se deja libre al dominio público. Esto se deja explícito en el siguiente apartado:

Discusión sobre varios algoritmos de ordenación y sus características.

1. Introducción

En este documento se estudiará el problema de ordenar un array de elementos sobre los cuales se puede establecer una relación de orden (i.e los operadores <, >, = tienen sentido).

Los algoritmos de este documento serán escritos en C y serán intercambiables entre si; es decir, todos aceptarán los mismos parámetros: un array A de datos y un entero que representa el tamaño del array.

Si bien en todo este documento se mostrará como ordenar de forma creciente (de menor a mayor), esperamos que el lector no tenga problemas realizar las modificaciones pertinentes (que deberán ser obvias) en los algoritmos para poder ordenar de forma decreciente.

El array de entrada A tendrá elementos de tipo `Dato` los cuales podrán ser cualquier tipo de dato representable (una estructura, un objeto de una clase en particular, un número, un string, etc). Dicho array será usado al estilo C, es decir los índices de los N elementos de A serán 0, 1, 2, ..., N-1.

Se supondrá por simplicidad que los datos aceptan el uso de operadores "<" y ">". Si bien esto no será así en la práctica, no perderemos generalidad. La estrategia para establecer un orden entre los elementos dependerá de la tecnología que usemos para implementar el algoritmo de ordenación. En Appendix A se discute esto con más profundidad y para tres tecnologías de programación distintas: programación estruturada (C), programación orientada a objetos (C++) y programación de objetos (Smalltalk, Self, etc.)

También se utilizará el operador de asignación de manera especial en algunos casos. Por ejemplo, en el siguiente segmento de código, en `tmp` se almacenará una copia del iésimo elemento del array A de entrada.

```
Dato tmp;

/* ... */

tmp = A[i];
```

Junto con la descripción de cada algoritmo también se discutirá el orden de tiempo de ejecución del mismo. Se utilizará para esto la notación de ordenes de magnitud "O grande" ("big oh"), descripta en el documento "Análisis de Algoritmos" del mismo autor (ver *Referencias*). La medición de estos tiempos ha sido hecha considerando solamente la cantidad de comparaciones, asignaciónes, etc que impliquen elementos del array de datos: o sea, las cotas de tiempo sólo están en función del tamaño del conjunto de datos. Puesto que estos pueden ser arbitrariamente complejos, no se consideran los tiempos de operaciones sobre ellos. Por ejemplo, (la velocidad de un algoritmo será distinta para conjuntos de enteros que para conjuntos de reales, puesto que las operaciones sobre estos últimos por lo general son más lentas que sobre los primeros).

2. Ordenación por selección

Características.

- Su tiempo de ejecución es $O(N^2)$ para el mejor, peor y caso promedio.
- Es el más fácil de codificar de los mostrados en este documento.
- Si el array de datos es A y su tamaño es N, lo que hace el algoritmo, para cada i de [0..N-2] es intercambiar A[i] con el mínimo elemento del subarray [A[i+1], ..., A[N]].
- Dado que es muy simple de codificar, aunque no tiene los mejores tiempos de ejecución, es apropiado utilizarlo para arrays de datos relativamente pequeños.

```
void
intercambiar (Dato * A, int i, int j)
{
  Dato tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
}

void
ordenacion_seleccion (Dato * A, int N)
{
  int i, j, k;
  for (i = 0; i < N - 1; i++)
    {
      for (k = i, j = i + 1; j < N; j++)
```

```
 if (A[j] < A[k])
   k = j;
      if (k != i)
 intercambiar (A, i, k);
    }
}
```

3. Ordenación por inserción

Características.

- Es un algoritmo sencillo de entender y de codificar.
- Si el tamaño de la entrada es N, entonces el orden del tiempo de ejecución, para el peor caso es $O(N^2)$;
- Si la entrada esta "casi ordenada", el algoritmo se ejecuta mucho más rápidamente. Esta velocidad tiende a un tiempo O(N), peor caso que se cumple cuando la entrada está totalmente ordenada.
- Es por la propiedad anterior que este algoritmo, a pesar de no ser el más rápido para entradas grandes, suele usarse de la siguiente manera: Se semi ordena la entrada con algún otro algoritmo más rápido y más adecuado para entradas grandes. Luego, cuando tenemos la entrada "casi ordenada" usamos este algoritmo. La velocidad de ejecución será muy buena por dos razones: su tiempo de ejecución tiende a O(N) con entradas "casi ordenadas" (lo cual es un tiempo excelente), y la simpleza de su implementación hará que se ejecute más rápido que otros algoritmos más complejos. Esto se implementará en Section 8.

Explicación del algoritmo. Sea A el array a ordenar con N elementos. El algoritmo consiste en recorrer todo el array A comenzando desde la posición p=2 y terminando en p=N. Para cada p, se trata de ubicar en el lugar correcto el elemento A[p] en entre los elementos anteriores: A[p-1], A[p-2], ..., A[0].

Dada la posición actual p, el algoritmo se basa en que los elementos A[0], A[1], ..., A[p-1] ya están ordenados.

```
void
ordenacion_insercion (Dato * A, int N)
{
  int p, j;
```

```
  Dato tmp;
  for (p = 1; p < N; p++)
    {
      tmp = A[p];
      j = p - 1;
      while ((j >= 0) && (tmp < A[j]))
        {
          A[j + 1] = A[j];
          j--;
        }
      A[j + 1] = tmp;
    }
}
```

Establezcamos la siguiente definición para poder discutir sobre el orden del tiempo de ejecución del algoritmo y algunas de sus características.

Inversión. Si hablamos de orden creciente, decimos que en un array `A` de `n` datos, la pareja `(A[i],A[j])` es una inversión si se cumple que i<j y A[i]>A[j]. (en orden decreciente sería el caso contrario).

Es decir, una *inversión* es aquella pareja `(A[i],A[j])` que rompe con el orden de una secuencia de datos. Dada una secuencia A, si pudiéramos detectar todas sus inversiones e intercambiar los valores de las mismas obtendríamos la secuencia ordenada. Por ejemplo, considérese la secuencia `(5 3 9 7)`. Las inversiones de esta secuencia son (5,3) y (9,7). Intercambiando el 5 por el 3 y el 9 por el 7 obtenemos la secuencia ordenada `(3 5 7 9)`.

El algoritmo de ordenación por inserción lo que hace es recorrer todo el array detectando inversiones (la segunda condición del `while()`) y corrigiéndolas (una por una en cada iteracción del `while()`). Dado que existe esta relación entre el algoritmo de ordenación y el número de inversiones, calculando el número medio de inversiones en la secuencia de datos se pueden obtener cotas precisas sobre el tiempo de ejecución del algoritmo.

Puede demostrarse fácilmente que le número medio de inversiones en un array de n elementos es $n(n-1)/4 = O(n^2)$.

Podemos decir entonces que el algoritmo de ordenación por inserción (y en realidad cualquier algoritmo de ordenación que intercambie sólo elementos adyacentes), requiere un tiempo de hasta orden n^2 en promedio.

4. Ordenación de Shell (ShellSort)[1]

Características.

- A diferencia del algoritmo de ordenación por inserción, este algoritmo intercambia elementos distantes. Es por esto que puede deshacer más de una inversión en cada intercambio, hecho del cual nos aprovechamos para ganar velocidad.
- La velocidad del algoritmo dependerá de una secuencia de valores (llamados incrementos, de los cuales hablaremos más abajo) con los cuales trabaja utilizándolos como distancias entre elementos a intercambiar. Veremos que con algunas secuencias podremos obtener ordenes de tiempo de ejecución en el peor caso de $O(n^2)$, O(n^(3/2)) y O(n^(4/3)).
- Se considera la ordenación de Shell como el algoritmo más adecuado para ordenar entradas de datos moderadamente grandes (decenas de millares de elementos) ya que su velocidad, si bien no es la mejor de todos los algoritmos, es aceptable en la práctica y su implementación (código) es relativamente sencillo.

Antes que nada definiremos el concepto de k-ordenación.

Secuencia k-ordenada. Decimos que una secuencia A de n elementos está k-ordenada (siendo k un natural) si, para todo i de [0,n] se cumple que los elementos A[i + hk] están ordenados, siendo h un entero y siempre que el índice i+hk esté en [0,n].

El algoritmo de ordenación de Shell lo que hace en realidad es tomar una secuencia de incrementos h_1, h_2, ..., h_p y en la etapa k-esima realizar una h_k-ordenación de los datos. La única condición que debe respetar la secuencia de incrementos es h_p=1 de modo tal que en la última etapa se hace una 1-ordenación (o sea una ordenación normal).

¿Cómo hacemos para k-ordenar un array A de n elementos? Dado que la ordenación por inserción lo que hace es una 1-ordenación, usaremos el mismo algoritmo para k-ordenar, pero comparando sólo elementos k-distanciados. Más detalladamente, para cada i de [k+1, n] intercambiamos (si hay que hacerlo) A[i] con alguno de los elementos anteriores a i con distancia múltiplo de k (es decir, intercambiamos A[i] con con alguno de los elementos A[i-k], A[i-2k], ...).

Una propiedad importante de implementar una k-ordenación de este modo es que si k>p entonces realizar una p-ordenación luego de una k-ordenación conserva el k-orden.

El lector podría estar preguntándose por qué k-ordenar varias veces si al final terminamos haciendo una 1-ordenación utilizando ordenación por inserción. ¿Por qué no hacer simplemente una 1-ordenación? Recordemos que la ordenación por inserción

trabaja mucho más rápido si la secuencia de datos está "casi ordenada". Así, luego de efectuar varias k-ordenaciones, en la última etapa, la 1-ordenación tiene un array "casi ordenado", por lo que el algoritmo se ejecuta más rápidamente.

Podemos decir entonces que la ordenación de Shell, es en realidad una ordenación por inserción pero a la cual se le pasa el array "casi ordenado". Esta "casi ordenación" se efectúa mediante h_k-ordenaciones para algunos h_k.

Como el lector podrá intuir, la velocidad del algoritmo dependerá de esta secuencia de incrementos h_k y una buena secuencia de incrementos constará de naturales primos relativos entre sí.

Secuencias de incrementos usadas comúnmente.

- La propuesta por Shell: n/2, n/4, ..., n/(2^k), ..., 1. Ventajas: es fácil de calcular. Desventajas: no optimiza mucho la velocidad del algoritmo puesto que los incrementos tienen factores comunes (no son primos relativos). El tiempo de ejecución promedio con esta secuencia de incrementos será $O(n^2)$.
- La propuesta por Hibbard: 2^k-1, ..., 7, 3, 1; donde k se elije de modo tal que $2^k-1 < n/2$ y $2^{k+1}-1 > n/2$. Aunque no se ha podido demostrar formalmente (sólo por medio de simulaciones), utilizar esta secuencia de incrementos hace que el algoritmo de Shell tenga un tiempo de ejecución promedio $O(n^{5/4})$. Una cota para el peor caso (que se puede demostrar utilizando la teoría de números y combinatoria avanzada) es $O(n^{3/2})$.
- La propuesta por Sedgewick: $4^k-3\cdot 2^k+1$, ..., 19, 5, 1; con la cual se pueden obtener tiempos de ejecución $O(n^{4/3})$ para el peor caso y $O(n^{7/6})$ para el caso promedio.

A continuación damos el algoritmo formal de la ordenación de Shell, en el cual utilizaremos los incrementos propuestos por Shell por simplicidad en el código:

```
void
ordenacion_shell (Dato * A, int N)
{
  int incr = N / 2, p, j;
  Dato tmp;
  do
    {
      for (p = incr + 1; p < N; p++)
 {
   tmp = A[p];
   j = p - incr;
   while ((j >= 0) && (tmp < A[j]))
            {
              A[j + incr] = A[j];
```

```
                j -= incr;
              }
    A[j + incr] = tmp;
  }
       incr /= 2;
     }
   while (incr > 0);
}
```

Observar que dentro del `do { ... } while()`, el algoritmo es análogo a la ordenación por inserción, salvo que se compara `tmp` con los elementos anteriores `incr`-distanciados.

5. Ordenación por montículos (Heapsort)

Características.

- Es un algoritmo que se construye utilizando las propiedades de los montículos binarios.
- El orden de ejecución para el peor caso es O(N·log(N)), siendo N el tamaño de la entrada.
- Aunque teóricamente es más rápido que los algoritmos de ordenación vistos hasta aquí, en la práctica es más lento que el algoritmo de ordenación de Shell utilizando la secuencia de incrementos de Sedgewick.

Breve repaso de las propiedades de los montículos binarios (heaps)

Recordemos que un montículo Max es un árbol binario completo cuyos elementos están ordenados del siguiente modo: para cada subárbol se cumple que la raíz es mayor que ambos hijos. Si el montículo fuera Min, la raíz de cada subárbol tiene que cumplir con ser menor que sus hijos.

Recordamos que, si bien un montículo se define como un árbol, para representar éste se utiliza un array de datos, en el que se acceden a padres e hijos utilizando las siguientes transformaciones sobre sus índices. Si el montículo está almacenado en el array A, el padre de A[i] es A[i/2] (truncando hacia abajo), el hijo izquierdo de A[i] es A[2*i] y el hijo derecho de A[i] es A[2*i+1].

> Al insertar o eliminar elementos de un montículo, hay que cuidar de no destruir la propiedad de orden del montículo. Lo que se hace generalmente es construir rutinas de filtrado (que pueden ser ascendentes o descendentes) que tomen un elemento del montículo (el elemento que viola la propiedad de orden) y lo muevan vérticalmente por el árbol hasta encontrar una posición en la cual se respete el orden entre los elementos del montículo.
>
> Tanto la inserción como la eliminación (`eliminar_min` o `eliminar_max` según sea un montículo Min o Max respectivamente), de un elemento en un montículo se realizan en un tiempo O(log(N)), peor caso (y esto se debe al orden entre sus elementos y a la característica de árbol binario completo).

Estrategia general del algoritmo. A grandes razgos el algoritmo de ordenación por montículos consiste en meter todos los elementos del array de datos en un montículo MAX, y luego realizar N veces `eliminar_max()`. De este modo, la secuencia de elementos eliminados nos será entregada en ordes decreciente.

Implementación práctica del algoritmo. Existen dos razones por las que la estrategia general debe ser refinada: el uso de un tad auxiliar (montículo binario) lo cual podría implicar demasiado código para un simple algoritmo de ordenación, y la necesidad de memoria adicional para el montículo y para una posible copia del array. Estas cosas también implican un gasto en velocidad.

Lo que se hace es reducir el código al máximo reduciéndose lo más posible la abstracción a un montículo. Primero que nada, dado el array de datos, este no es copiado a un montículo (insertando cada elemento). Lo que se hace es, a cada elemento de la mitad superior del array (posiciones 0,1,...,N/2) se le aplica un filtrado descendente (se "baja" el elemento por el árbol binario hasta que tenga dos hijos que cumplan con el orden del montículo. Esto bastará para hacer que el array cumpla con ser un montíclo binario.

Notamos también que al hacer un eliminar_max() se elimina el primer elemento del array, se libera un lugar a lo último de este. Podemos usar esto para no tener que hacer una copia del array para meter las eliminaciónes sucesivas. Lo que hacemos es meter la salida de eliminar_max() luego del último elemento del montículo. Esto hace que luego de N-1 eliminar_max(), el array quede ordenado de menor a mayor.

Para ahorrar código queremos lograr insertar y eliminar_max() con una sola rutina de filtrado descendente. Ya explicamos cómo hacer que el array de datos A preserve el orden de los elementos de un montículo. Lo que hacemos para ordenarlo usando sólo la rutina de filtrado descendente es un eliminar_max() intercambiando el primer elemento del array por el último del montículo y filtrar el nuevo primer elemento hasta que se respete el orden Max. Quizá esto quede más claro mirando el código.

```
void
filtrado_desc (Dato * A, int i, int N)
{
  /* queremos que se respete el orden MAX del montículo */
  Dato tmp = A[i];
  int hijo = 2 * i;

  if ((hijo < N) && (A[hijo + 1] > A[hijo]))
    hijo++;

  while ((hijo <= N) && (tmp < A[hijo]))
    {
      /* elijo bien el hijo */
      if ((hijo < N) && (A[hijo + 1] > A[hijo]))
 hijo++;
      A[i] = A[hijo];
      i = hijo;
      hijo = 2 * i;
    }
  A[i] = tmp;
}


void
intercambiar (Dato * A, int i, int j)
{
  Dato tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
}


void
heapsort (Dato * A, int N)
{
  int i;

  /* meto los datos en el montículo (ordeno) */
  for (i = N / 2; i >= 0; i--)
    filtrado_desc (A, i, N);

  /* saco los datos y los meto al final para obtener el array ordenado */
  for (i = N - 1; i > 0; i--)
    {
```

```
      intercambiar (A, 0, i);
      filtrado_desc (A, 0, i - 1);
    }
}
```

6. Ordenación por Intercalación (mergesort)

Características.

- Es un algoritmo recursivo con un número de comparaciones mínimo. El tiempo de ejecución promedio es O(N log(N)).
- Su desventaja es que trabaja sobre un array auxiliar lo cual tiene dos consecuencias: uso de memoria extra y trabajo extra consumido en las copias entre arreglos (aunque es un trabajo de tiempo lineal).
- Es una aplicación clásica de la estrategia para resolución de algoritmos "divide y vencerás". Esta estrategia plantea el hecho de que un problema puede ser dividido en varios subproblemas y una vez resueltos estos se puede proceder a unir las soluciones para formar la solución del problema general. La solución de los subproblemas más pequeños se realiza de la misma manera: es decir, se van resolviendo problemas cada vez más pequeños (hasta encontrarnos con un caso base: problema no divisible con solución tribial).

En cada recursión se toma un array de elementos desordenados. Se lo divide en dos mitades, se aplica la recursión en cada una de estas y luego (dado que al finalizar estas recursiones tenemos las dos mitades ordenadas) se *intercalan* ambas para obtener el array ordenado.

Intercalar. Es la operación que le da el nombre a este algoritmo. La intercalación toma dos secuencias (arrays) de elementos y a partir de estas construye una tercera secuencia que contiene todos los elementos de estas en orden.

Implementación de la intercalación. Sean dos arrays ordenados A y B (cuyas longitudes pueden ser distintas). Sea el array C tal que |C|>=|A|+|B| (tiene capacidad para almacenar todos los elementos de ambas listas A y B). Sean los contadores ap, bp y cp con valor inicial 0 (se ponen al inicio de sus arreglos respectivos). Lo que hace `intercalar` es copiar el menor de A[ap] y B[bp] en C[cp] y avanzar los contadores apropiados. Cuando se agota cualquier lista de entrada (A o B), los datos que quedan en la otra lista se copian en C.

Esperamos que al leer el código, el lector entienda los detalles menores tanto de la rutina recursiva del algoritmo de recursión como de la rutina `intercala()`.

```
void ord_intercalacion (Dato * A, Dato * tmp, int izq, int der);
/* Función recursiva!  A es el array de datos. tmp debe ser un
   array de tamaño mayor o igual a A. izq y der son los extremos
   del subarreglo sobre el cual trabajaremos en esta
   recursión. */


void intercalar (Dato * A, Dato * tmp, int izq, int centro, int der);
/* lo que hace esta rutina es intercalar las particiones
   [A[izq], ..., A[centro-1] ] y [ A[centro], ..., A[der] ]
   (que deberían estar ya ordenadas) en el subarray
   [ tmp[izq], ..., tmp[der] ] y luego copiar los elementos
   nuevamente a A, en el subarray [ A[izq], ..., A[der] ] */


void
mergesort (Dato * A, int N)
{
  Dato *tmp = crear (N); /* creamos un array auxiliar del mismo
       tamaño que A */
  ord_intercalacion (A, tmp, 0, N - 1);
}


void
ord_intercalacion (Dato * A, Dato * tmp, int izq, int der)
{
  if (izq < der)
    /* este if comprueba el caso base que es cuando la partición
       pasada no tiene elementos. */
    {
      /* dividimos a la mitad el subarray [A[izq],...,A[der]] */
      int centro = (izq + der) / 2;

      /* aplicamos la recursión en ambas mitades */
      ord_intercalacion (A, tmp, izq, centro);
      ord_intercalacion (A, tmp, centro + 1, der);

      /* a este punto ambas mitades deberían estar ordenadas por
         lo que las intercalamos para unirlas en una sola
         secuencia ordenada. */
```

```
      intercalar (A, tmp, izq, centro + 1, der);
    }
}


void
intercalar (Dato * A, Dato * tmp, int izq, int centro, int der)
{
  /* mis particiones serán [izq,...,centro-1] y
     [centro,...,der] */

  /* contadores para la primera mitad, la segunda y para la
     intercalacion respectivamente. */
  int ap = izq, bp = centro, cp = izq;

  while ((ap < centro) && (bp <= der))
    {
      if (A[ap] <= A[bp])
 {
   tmp[cp] = A[ap];
   ap++;
 }
      else
 {
   tmp[cp] = A[bp];
   bp++;
 }
      cp++;
    }

  /* terminamos de intercalar, ahora metemos los elementos
     restantes de la lista que aún no ha terminado de ser
     procesada. */
  while (ap < centro)
    {
      tmp[cp] = A[ap];
      cp++;
      ap++;
    }
  while (bp <= der)
    {
      tmp[cp] = A[bp];
      cp++;
      bp++;
```

```
    }

  /* ahora que tenemos la intercalación finalizada en tmp, la
     pasamos a A */
  for (ap = izq; ap <= der; ap++)
    A[ap] = tmp[ap];
}
```

Observar como la función principal `mergesort()` solamente es un manejador de la función `ord_intercalacion()` en la cual se realiza todo el trabajo recursivamente.

Si bien el algoritmo puede parecer largo, es más fácil (desde nuestro punto de vista) entenderlo que otros algoritmos más cortos pero más complejos como por ejemplo la ordenación de shell. La única parte difícil es entender cómo funciona la recursión ya que el algoritmo intercalar es bastante fácil.

7. Ordenación rápida (quicksort)

Características.

- Es el algoritmo de ordenación más rápido (en la práctica) conocido. Su tiempo de ejecución promedio es O(N log(N)).
- Para el peor caso tiene un tiempo $O(N^2)$, pero si se codifica correctamente las situaciones en las que sucede el peor caso pueden hacerse altamente improbables.
- EN la práctica, el hecho de que sea más rápido que los demás algoritmos de ordenación con el mismo tiempo promedio O(N log(N)) está dado por un ciclo interno muy ajustado (pocas operaciones).
- Al igual que el algoritmo de ordenación por intercalación, el algoritmo de ordenación rápida es fruto de la técnica de resolución de algoritmos "divide y vencerás". Como ya se explicó, la técnica de divide y vencerás se basa en, en cada recursión, dividir el problema en subproblemas más pequeños, resolverlos cada uno por separado (aplicando la misma técnica) y unir las soluciones.

Se explica a continuación, a grandes razgos, qué se hace en cada recursión:

1. Si el array de datos es vacío o tiene un sólo elemento no hacemos nada y salimos de la recursión.

2. Elegimos un elemento v (llamado *pivote*) del array de datos.
3. Particionamos el array de datos A en tres arrays:
   - A_1={todos los elementos de A - {v} que sean menores o iguales que v}
   - A_2={v}
   - A_3={todos los elementos de A - {v} que sean mayores o iguales que v}
4. Aplicamos la recursión sobre A_1 y A_3[2]
5. Realizamos el último paso de "divide y vencerás" que es unir todas las soluciones para que formen el array A ordenado. Dado que a este punto A_1 y A_3 están ya ordenados, concatenamos A_1, A_2 y A_3.

Si bien este algoritmo puede parecer sencillo, hay que implementar 2 y 3 de modo de favorecer al máximo la velocidad del algoritmo. Es por esto que discutiremos a parte cómo realizar estas tareas.

7.1. La elección del pivote

Se discutirá a continuación cuál es el mejor método para implementar 2. Es importante discutir este punto porque un mecanismo erróneo puede estropear la velocidad del algoritmo (de hecho puede hacer que el algoritmo quede muy lento).

Sin lugar a dudas la eficiencia de este algoritmo dependerá de cómo elegimos el pivote v ya que este determina las particiones del array de datos.

> **Important:** Lo que queremos lograr es elegir el pivote v tal que las particiones A_1 y A_3que se realizan en 3 sean de tamaños (cantidad de elementos) lo más iguales posible.

Al lector se le puede ocurrir la siguiente estrategia: dado que el array de datos está desordenado, se podría elegir el primer elemento del array como pivote. Esto funcionaría bien si los datos estuvieran uniformemente distribuidos en el array. Sin embargo, no podemos suponer esto: al algoritmo se le podría estar pasando un array "casi ordenado", y elegir el primer elemento como pivote resultaría, en este caso, desastroso ya que $|A_1|$ sería muy pequeño comparado con $|A_3|$. Por la misma razón, elegir el último elemento del array como pivote también es una mala idea.

Si contamos con un buen generador de números aleatorios (siempre que estos cumplan con una distribución uniforme), una solución sería elegir el pivote al azar entre los

elementos del array de datos. Por lo general, esta estrategia es segura ya que es improbable que un pivote al azar de una partición mala. Sin embargo, la elección de números aleatorios es un proceso costoso e incrementa el tiempo de ejecución del algoritmo. mecanismo

Una buena estrategia: madiana de tres elementos. . Una estrategia para elegir el pivote que siempre funcionará es elegir como pivote la mediana de los elementos de la izquierda, la derecha y el centro, es decir, si el array de datos es A, la mediana del conjunto { A[0], A[N-1], A[(N-1)/2] }. Por ejemplo, si nuestro array de datos fuera 6 8 1 3 10 2 9, el pivote es la mediana de { 6, 9, 3 } o sea 6. Esto determinará las particiones A_1={1, 3, 2}, A_2={ 6 } y A_3={ 8, 10, 9 }.

Es importante tener en cuenta que esta estrategia de elección del pivote no dará la mejor elección[3]: simplemente nos asegura que, en promedio, la elección del pivote es buena.

7.2. La estrategia de partición

Sea A el array de datos con N elementos. Se podría implementar muy fácilmente la partición utilizando arrays auxiliares. Sin embargo no podemos permitirnos este uso de memoria auxiliar ni el tiempo que lleva almacenar los datos en estos arrays auxiliares.

Trataremos de reordenar los elementos de A de forma tal que los elementos que aparecen antes del pivote sean menores o iguales a él, y los que aparecen después del pivote sean mayores o iguales a él. De este modo nos ahorraremos el paso 5 y no necesitaremos utilizar memoria auxiliar.

Una vez que hemos elegido el pivote v, intercambiamos éste con el último elemento de A. Tomamos dos contadores: i que apunte al primer elemento de A y j que apunte al penúltimo elemento de A (el último es el pivote). Mientras i<j incrementamos i hasta encontrar un elemento mayor o igual al pivote y decrementamos j hasta encontrar un elemento menor o igual al pivote. (Notar que cuando i y j se han detenido, i apunta a un elemento grande y j apunta a un elemento pequeño). Siempre que se siga cumpliendo i<j, intercambiamos A[i] con A[j]. Seguimos incrementando i y decrementando j y seguimos intercambiando hasta que i>=j. Cuando esto sucede (i y j se cruzan), i está apuntando al primer elemento de lo que sería la partición A_3. Queremos que el pivote quede en el medio de las dos particiones por lo que intercambiamos A[i] con el pivote (A[N-1]).

Tal vez el lector esté interesado en la siguiente discusion: ¿Por qué parar los contadores i o j cuando nos encontramos con un elemento igual al pivote? Póngase el caso límite en el que todos los datos sean iguales[4]. Si i se detuviera pero j no cuando se encuentra con un elemento igual al pivote, tendríamos que al final de algoritmo todos los

elementos quedan en la partición A_3. Si ninguno de los contadores i o j se detuviera al encontrarse con un elemento igual al pivote, se incrementaría i hasta la penúltima posición y esto creará particiones muy desiguales.

Cabe decir que esta es sólo una de las varias formas de particionar el array de datos. Sin embargo, alentamos al lector de seguir esta técnica ya que es muy fácil hacerlo mal o hacerlo sin eficiencia (lo que aruinará la velocidad del algoritmo).

7.3. Implementación

Primero que nada, tratamos de optimizar lo más que podamos el algoritmo de elección del pivote. Dado que debemos comparar tres elementos para hallar la mediana, podemos utilizar esto para ya ir ordenando el array de datos. Al hallar la mediana aprovechamos para ordenar los elementos A[izq]<=A[(izq+der)/2]<=A[der].

Aunque el algoritmo descripto es correcto, haremos una pequeña variación para optimizarlo aún más: una vez hallado el pivote lo intercambiamos por el penúltimo y no por el último elemento del array. Dado que estamos seguros de que A[izq]<=pivote y que A[der-1]=pivote, tenemos que A[izq] y A[der-1] son centinelas para que los contadores i y j paren (no tendremos que realizar comparaciones extra para verificar que i y j no sobrepasen los límites del array). A continuación presentamos el algoritmo completo. Sabemos que se podría haber dividido más la función principal para un mejor entendimiento. Sin embargo, el uso de varias funciones distintas puede enlentecer el algoritmo cuando se copian las variables que se pasan como parámetros a las funciones.

```
void ord_rapida (Dato * A, int izq, int der);

void
quicksort (Dato * A, int N)
{
  ord_rapida (A, 0, N - 1);
}

void
ord_rapida (Dato * A, int izq, int der)
     /* se trabaja en el subarray [A[izq],...,A[der]] */
{
  if (der - izq > 1) /* caso base de la recursión */
    {
      {  /* elegimos el pivote y lo ponemos en A[der-1] */
 int centro = div2 (izq + der);
 if (A[izq] > A[centro])
```

```
      intercambiar (A, izq, centro);
    if (A[izq] > A[der])
      intercambiar (A, izq, der);
    if (A[centro] > A[der])
      intercambiar (A, centro, der);
    intercambiar (A, centro, der - 1);
         }

         {  /* "particionamos" */
    int i = izq, j = der - 1;
    Dato pivote = A[der - 1];
    do
      {
        do
          i++;
        while (A[i] < pivote);
        do
          j--;
        while (A[j] > pivote);
        intercambiar (A, i, j);
      }
    while (j > i);

    /* deshacemos el último intercambio el cual se efectuó
       sin cumplirse i<j */
    intercambiar (A, i, j);

    /* ponemos el pivote en el medio de ambas particiones */
    intercambiar (A, i, der - 1);

    /* aplicamos la recursión en las particiones halladas */
    ord_rapida (A, izq, i - 1);
    ord_rapida (A, i + 1, der);
         }
       }
  }
```

8. Utilizando múltiples algoritmos

En Section 3 prometimos usar el algoritmo de ordenación por inserción para terminar de ordenar un array de datos "casi ordenado". En esta sección examinaremos este caso realizando lo siguiente: Modificamos ligeramente el algoritmo de ordenación rápida para que ordene particiones de hasta CORTE elementos, donde CORTE es un entero previamente determinado en función del tamaño de la entrada. Mostramos la implementación esperando que se entienda el cometido de las ligeras modificaciones:

```
#include <math.h>
static int CORTE;
void ord_rapida (Dato * A, int izq, int der);

void
quicksort (Dato * A, int N)
{
  CORTE=log(N)*log(N);
  ord_rapida (A, 0, N - 1);
  if(CORTE>1)
    ordenacion_insercion(A, N);
}

void
ord_rapida (Dato * A, int izq, int der)
     /* se trabaja en el subarray [A[izq],...,A[der]] */
{
  if (der - izq > CORTE) /* caso base de la recursión */
    {
      {  /* elegimos el pivote y lo ponemos en A[der-1] */
 int centro = div2 (izq + der);
 if (A[izq] > A[centro])
   intercambiar (A, izq, centro);
 if (A[izq] > A[der])
   intercambiar (A, izq, der);
 if (A[centro] > A[der])
   intercambiar (A, centro, der);
 intercambiar (A, centro, der - 1);
      }

      {    /* "particionamos" */
 int i = izq, j = der - 1;
 Dato pivote = A[der - 1];
 do
   {
```

```
      do
        i++;
      while (A[i] < pivote);
      do
        j--;
      while (A[j] > pivote);
      intercambiar (A, i, j);
    }
 while (j > i);

 /* deshacemos el último intercambio el cual se efectuó
    sin cumplirse i < j */
 intercambiar (A, i, j);

 /* ponemos el pivote en el medio de ambas particiones */
 intercambiar (A, i, der - 1);

 /* aplicamos la recursión en las particiones halladas */
 ord_rapida (A, izq, i - 1);
 ord_rapida (A, i + 1, der);
      }
    }
}
```

Como puede verse sólo se ha modificado una línea de la función `ord_rapida()` pero se ha modificado sustancialmente la función `quicksort()`. En esta se determina el valor correcto de la variable CORTE y se aplica, luego de ord_rapida() la función de ordenación por inserción. Esta última se ejecutará muy rápido cuanto "mejor ordenado" halla quedado el array de datos.

Hemos propuesto CORTE=$\log(N)^2$ basados sólo en observaciones experimentales. Sin embargo, hemos observado un incremento de velocidad bastante bueno con este corte.

9. Velocidades observadas

A continuación se tabulan y grafican los tiempos de ejecución de algunos algoritmos vistos hasta aquí para entradas de tamaño 10, 100, 1000, ..., 10000000.

> **Note:** Las gráficas que siguen no son muy buenas. Sin embargo esperamos que sirvan para ver las diferencias

Figure 1. Comparación de los tiempos de ejecución de varios algoritmos de ordenación

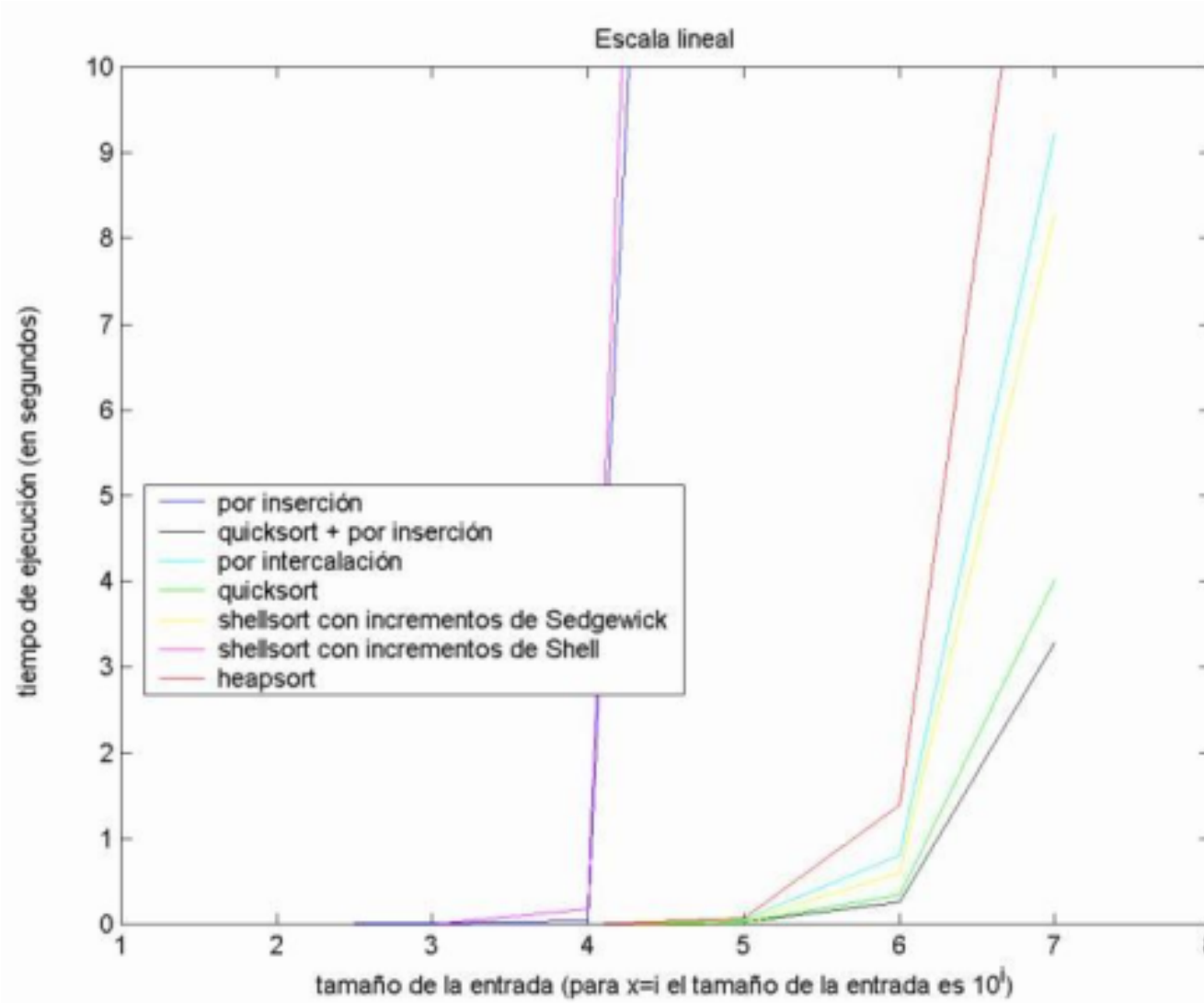
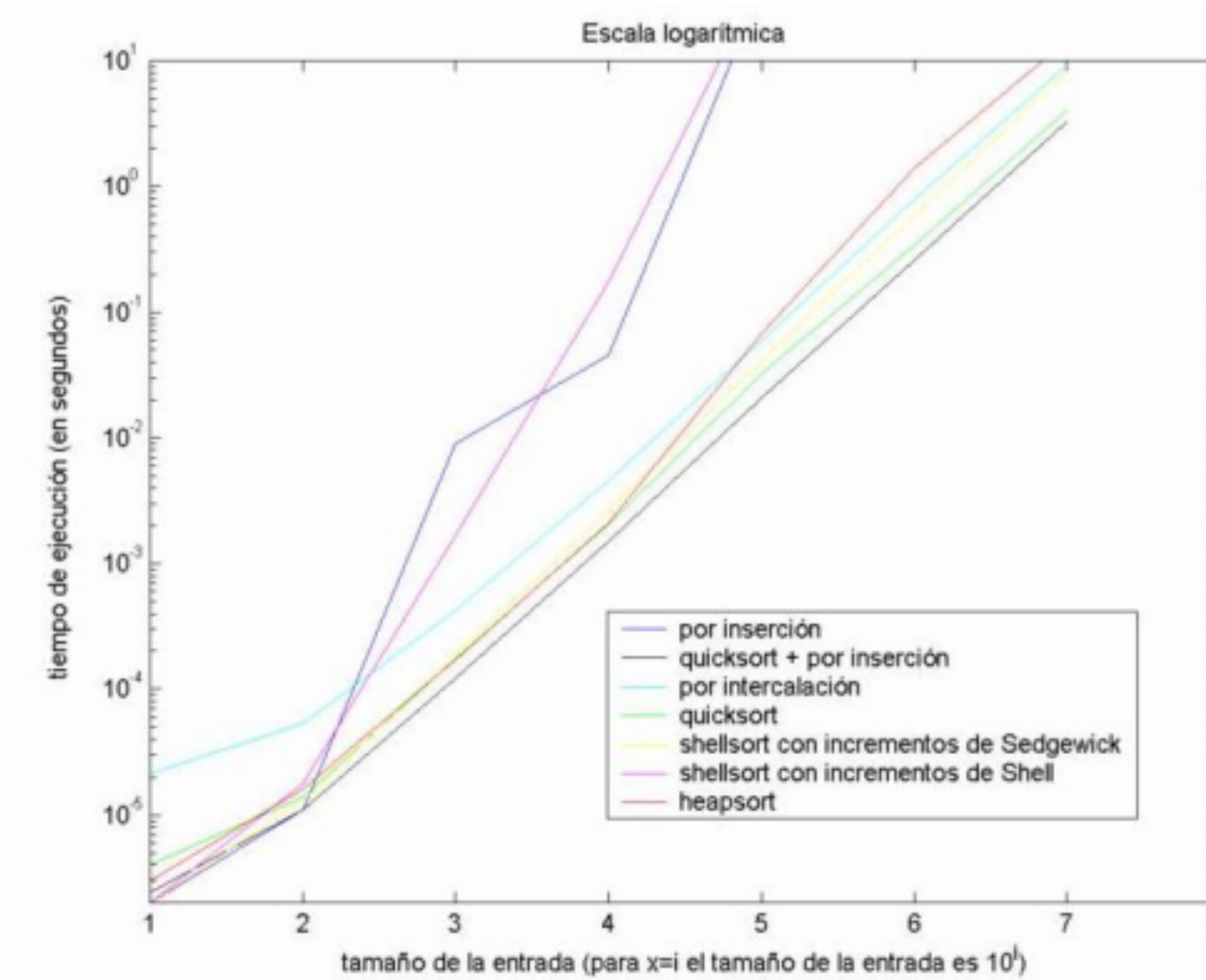


Figure 2. Comparación de los tiempos de ejecución de varios algoritmos de

ordenación



10. Ordenando elementos de gran tamaño: Ordenación indirecta

A este punto queremos advertir al lector sobre las dificultades de ordenar objetos grandes. Con objeto grande nos referimos a objetos en los que el tiempo que demora

hacer una copia en memoria del mismo es considerable.

Hasta ahora veníamos trabajando en nuestro código con el tipo Dato como tipo de objeto a ordenar. Asignábamos los datos con el operador de asignación, como si fueran enteros o caracteres. Sin embargo, si nuestros objetos son relativamente grandes, no podemos darnos el lujo de trabajar directamente sobre ellos. Es decir, en el algoritmo de ordenación, debemos evitar tocar los objetos en sí y sólo trabajar con apuntadores.

¿Cómo hacemos esto? En lugar de trabajar con un array de objetos (datos) trabajar con un array de punteros a los objetos (direcciones de los datos). Los objetos no se moverán de la memoria: moveremos las variables que contienen su dirección. Dado que un puntero es un tipo de dato pequeño, hacer esto será muycho más eficiente que trabajar con todo el objeto en sí.

A esta técnica se la conoce como *ordenación indirecta* de los datos.

11. Cotas inferiores generales para algoritmos de ordenación: Árboles de decisión[TERMINAR]

Como hemos podido ver, los algoritmos descriptos en este documento solamente efectúan comparaciones entre los elementos de un array de datos para ordenarlos. ¿A qué nos referimos con esto? Nos referimos a que no trabajan con información adicional sobre los datos y que sólo puedan realizar comparaciones entre dos de ellos para contruir el array ordenado. Dichas comparaciones pueden definirse como preguntas con dos posibles respuestas: SI o NO. Por ejemplo, A[i]>A[j] sólo puede tener dos valores lógicos: "true" o "false".

Nos complace informarle que existe una estructura con la cual se puede representar algoritmos de este tipo: *un árbol de decisión*. No es más que un árbol binario cuyos nodos representan las posibles soluciones considerando las deciciones tomadas hasta ese punto y cuyas aristas representan deciciones. Dado que a la raíz del árbol no llega ninguna arista (no hemos tomado ninguna decisión en la raíz) odas las ordenaciones (permutaciones)

A. Estableciendo una relación de orden sobre

los datos.

Una vez creado el paquete de datos que representará los objetos que queremos ordenar, debemos establecer una relación de orden entre estos objetos. Por ejemplo, nuestros datos podrían ser personas (representadas en un registro/estructura/clase/etc) y podríamos querer ordenarlas por su nombre, alfabéticamente. A continuación explicamos tres estrategias para definir y usar una relación de orden.

A.1. Paso de funciones por parámetro

En un lenguaje relativamente de bajo nivel como C, nos parece que la forma más proplija de establecer una relación de orden entre los objetos de un tipo de dato, es pasando funciones de comparación por parámetro. Supongamos que queremos ordenar personas deacuerdo a su edad en orden alfabético. Así, nuestros datos podrían ser del tipo:

```
struct Persona_
{
   int edad, ci;
   char *nombre;
   Ocupacion *oc;
};

typedef struct Persona_ *Persona;
```

Lo que necesitamos ahora es declarar un tipo de función: *comparadora de Persona*. Queremos que todas las funciones comparadoras de personas acepten dos personas A y B como parámetros y devuelvan un entero negativo si A<B, cero si A=B y un entero positivo si A>B. El prototipo de dicha función podrá declararse así:

```
int (*PersonaComp)(Persona, Persona);
```

De ahora en más el tipo de dato `PersonaComp` representa a las funciones que aceptan dos parámetros `Persona` como entrada y devuelve un entero. Nuestros algoritmos de ordenación deberán su declaración por:

```
void sort(Persona *datos, int n,
          int (*PersonaComp)(Persona, Persona));
```

Como tercer parámetro el usuario deberá ingresar el nombre de una función de comparación de tipo PersonaComp.

Con esto, estamos obligando al usuario de nuestro algoritmo de ordenación que establezaca de antemano un orden entre los objetos a ordenar mediante una función de tipo PersonaComp. Una vez hecho esto el usuario sólo tendrá que pasar el nombre de dicha función comparadora como tercer parámetro de `sort()`.

Si la ordenación fuera por nombre y en orden alfabético, una buena función de comparación sería:

```
#include <string.h>
int comp(Persona A, Persona B)
{
   return strcmp(A->nombre, B->nombre);
}
```

Note: Notar como `comp()` es del tipo `PersonaComp()`.

A.2. Sobrecarga de operadores

En un lenguaje ma abstracto que C, como C++, tendremos la posibilidad de usar sobrecarga de operadores y con esto mantener un nivel más alto de abstracción que en C ya que no se usarán funciones de ningún tipo. Siguiendo con el ejemplo de ordenar personas alfabéticamente según su nombre, definamos la clase persona:

```
class Persona
{
   int edad, ci;
   char *nombre;
   Ocupacion *oc;
   public:
        // aquí deberían definirse funciones miembro de acceso,
        // constructores, destructores y el comportamiento deseado
        // para una persona...
};
```

Queremos definir el significado del operador "<" entre personas. Supongamos que A y B son dos personas. Entonces queremos que A<B devuelva true si y sólo si el nombre de A es menor (alfabéticamente) que el de B. Definimos este operador de la siguiente forma

```
#include <string.h>
bool operator < (Persona A, Persona B)
{
   if(strcmp(A.nombre, B.nombre)<0)
      return true;
   else
      return false;
}
```

De la misma forma se podrá definir el significado de los demás operadores de orden de C++ como ">", "==", ">=" y "<=". Una vez definidos, dentro de las funciones de ordenación se podrá utilizar implícitamente el operador, por ejemplo, si p1 y p2 son dos opbjetos Persona:

```
if ( p1 < p2 )
   printf("%s está primero que %s", p1.nombre, p2.nombre);
```

A.3. Definición de métodos de orden

En la programación de objetos (Smalltalk, Self, etc), dado que todas las acciones son realizadas únicamente mediante mensajes desde un objeto hacia otro, podemos establecer una relación de orden definiendo métodos de comparación entre dos objetos de la misma clase. Siguiente con el ejemplo de la clase Persona, en Smalltalk podríamos definirla como

```
Object subclass: #Persona
    instanceVariableNames: ’edad ci nombre ocupacion’
    classVariableNames: ”
    poolDictionaries: ”
```

Si tenemos cuidado de hacer que la variable de instancia `nombre` sea un String, simplemente agregando el siguiente método a Persona tendremos un mensaje de comparación (orden alfabético entre los nombres) entre personas:

```
< otraPersona
    ^((self nombre) < (otraPersona nombre))
```

B. GNU Free Documentation License

Se hace referencia al sitio http://www.gnu.org/copyleft/ para obtener una copia de la *GNU Free Documentation License* bajo la cual se libera este documento.

Referencias

Estructuras de Datos y Algoritmos, Mark Allen Weiss, Addison-Wesley Iberoamericana, 1995, ISBN 0-201-62571-7.

Fundamentos de Algoritmia, G. Brassard y P. Bratley, Prentice Hall, 1998, 84-89660-00-X.

Estructuras de Datos y Algoritmos, Aho, Hopcrotf y Ullman, Addison Wesley Iberoamericana, 1988, 0-201-64024-4.

Técnicas de Diseño de Algoritmos, Rosa Guerequeta and Antonio Vallecillo , Servicio de Publicaciones de la Universidad de Málaga. 1998, 84-7496-666-3, Segunda Edición: Mayo 2000.

Cómo programar en C++, Deitel y Deitel, Prentice Hall, 1999, 970-17-0254-9.

Análisis de Algoritmos, Sebastián Gurin (Cancerbero), 2004.

The C programming Language, By Brian W. Kernighan y Dennis M. Ritchie, Prentice-Hall, 1998, 0-13-110362-8.

Notes

1. Recibe este nombre por su creador Donald Shell.
2. ¿por qué no la aplicamos sobre A_2? por 1.

3. Para saber cuál es la mejor elección deberíamos recorrer todo el array de datos y buscar la mediana de sus elementos lo cual es muy costoso en velocidad.
4. El lector podría estarse preguntando ¿pero qué sentido tiene ordenar datos iguales? Sin embargo, si por ejemplo tuvieramos que ordenar un millón de datos dentro de los cuales halla 5000 datos iguales, dada la naturaleza recursiva de la ordenación rápida, tarde o temprano, se llamará el algoritmo para ordenar solamente estos 5000 datos iguales. De este modo es realmente importante asegurarnos de que para este caso extremo el algoritmo funciones correctamente y sea eficiente.