Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Gestión de Datos

Métodos de Clasificación

Ing. Enrique Reinosa
Leandro R. Barbagallo
Julio 2007

Índice

Índice 2
Introducción 3
  Formalización 3
  Registros 3
  Estabilidad 4
  In situ 5
  Clasificación interna y externa 6
Teoría de la complejidad u orden de crecimiento de algoritmos 7
  Ejemplos 8
  Problemas de decisión 9
  Clases de complejidad 10
  La pregunta P=NP 10
  Intratabilidad 10
Bubble Sort 11
Selection Sort 13
Insertion Sort 15
Shell Sort 18
Merge Sort 21
  Fusión lineal de dos arrays 21
  Fusión lineal de dos arrays 22
QuickSort 28
  Estrategia de partición 29
  Mejor de los casos 31
  Peor de los casos 31
  Seleccionando el pivote 32
Comparación de los métodos 35
Bibliografía 36

Introducción

La clasificación es una operación fundamental en las ciencias de computación, por lo que es uno de los campos más estudiados y donde se pueden encontrar gran cantidad de algoritmos.

Muchos de los algoritmos más importantes requieren que los datos estén previamente ordenados para poder operar, como es el caso de la búsqueda binaria o la búsqueda de elementos duplicados.

La elección del mejor algoritmo depende de varios factores como la cantidad de elementos a ordenar, el grado de orden con el que ya vienen dados los elementos, si va a ser ordenado en memoria RAM, en cinta o en disco, etc.

Formalización

Definamos formalmente el problema de la clasificación.

Entrada: Una secuencia de números $\{a_1, a_2, \ldots, a_n\}$ $n \geq 0$

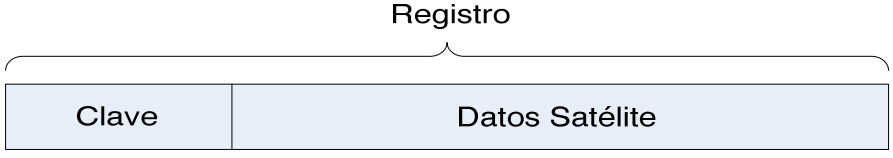
Salida: Una reclasificación $\{a'_1, a'_2, \ldots, a'_n\}$ de la secuencia inicial tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Por ejemplo una secuencia de entrada como $\{31, 41, 59, 26, 41, 58\}$ retorna la secuencia $\{26, 31, 41, 41, 58, 59\}$

La clasificación de los datos requiere al menos de dos operaciones fundamentales, la comparación de valores, es decir, el tipo de dato debe permitir determinar si un valor es menor, mayor o igual a otro y mover los valores a su posición ordenada, se debe poder reubicar los elementos.

Registros

En la práctica muy pocas veces los números a ordenar son valores aislados, cada número suele ser parte de una colección de datos llamado registro. Cada registro contiene una clave (key), que es el valor a ser ordenado, y el resto del registro contiene los datos satélites.



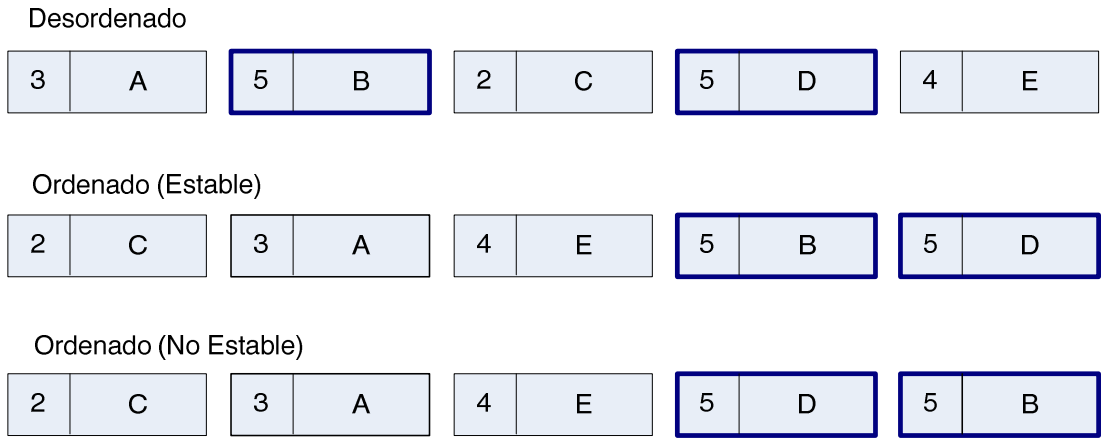
Generalmente cuando un algoritmo permuta las claves, también lo hace con los datos satélites. Si cada registro contiene grandes cantidades de datos, convendrá permutar punteros a los registros mismos para así minimizar los movimientos de datos.

El hecho de ordenar simples números o registros no es relevante para el método de clasificación en sí, por lo que generalmente se asume que la entrada simplemente consiste en números.

Estabilidad

Un ordenamiento se considera estable si mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Si se tiene dos registros A y B con la misma clave en la cual A aparece primero que B, entonces el método se considera estable cuando A aparece primero que B en el archivo ordenado.

Ejemplo



Podemos ver en la figura que el archivo desordenado tiene dos elementos con claves iguales, {5, B} y {5, D}. En el archivo ordenado mediante un método estable este orden relativo se conserva, no siendo así en el método no estable.

La mayoría de los métodos de clasificación simples son estables, pero gran parte de los métodos complejos no lo son. Se puede transformar uno inestable en estable a costa de mayor espacio en memoria o mayor tiempo de ejecución.

Una de las ventajas de los métodos estables es que permiten que un array se ordene usando claves múltiples, por ejemplo, por orden alfabético del apellido, luego el nombre, luego el documento, etc.

In situ

Los métodos in situ son los que transforman una estructura de datos usando una cantidad extra de memoria, siendo ésta pequeña y constante. Generalmente la entrada es sobrescrita por la salida a medida que se ejecuta el algoritmo. Por el contrario los algoritmos que no son in situ requieren gran cantidad de memoria extra para transformar una entrada.

Esta característica es fundamental en lo que respecta a la optimización de algoritmos, debido a que el hecho de utilizar la misma estructura disminuye los tiempos de ejecución, debido a que no se debe utilizar tiempo en crear nuevas estructuras, ni copiar elementos de un lugar a otro.

Como ejemplo veremos dos versiones de un algoritmo para invertir un array, una in situ y la otra no.

```
void invertirArray(int[] a) {

    int[] aux = new int[ a.length ];

    for( int c = 0; c < a.length ; c++) {

        aux[c] = a[a.length - c - 1];
    }

    a = aux;
}
```

```
void invertirArrayInSitu(int[] a) {

    int temp;

    for( int c = 0; c < a.length / 2; c++) {

        temp = a[c];
        a[c] = a[a.length - c - 1];
        a[a.length - c - 1] = temp;
    }
}
```

En el primer ejemplo se invierte el contenido de un array creando otro array auxiliar con el mismo tamaño que el que se quiere ordenar, esto requiere memoria extra para realizar la operación.

En el segundo ejemplo se ve un algoritmo que hace lo mismo pero a diferencia del anterior éste es in situ, donde no se requiere memoria adicional para alcanzar el objetivo.

Clasificación interna y externa

Si el archivo a ordenar cabe en memoria principal, entonces el método de clasificación es llamado método interno, por otro lado si ordenamos archivos desde un disco u otro dispositivo, se llama método de clasificación externo. La diferencia radica en que el método interno puede acceder a los elementos fácilmente y de forma aleatoria, mientras que el externo debe acceder a los elementos de forma secuencial o al menos en grandes bloques de datos.

De esta forma, dependiendo la situación y el conjunto de valores o bloques de registros a ordenar, se deberá utilizar métodos de clasificación internos o externos, considerando que para cada uno de los casos el método a aplicar será el acorde a dicho fin, sobre todo basado en el orden de crecimiento u orden de complejidad que analizaremos en el próximo apartado.

En función del método aplicado, también será diferente la concepción del algoritmo utilizado, debido que es bastante diferente una clasificación interna a una externa, por las diferencias de tiempo de acceso a los datos y los volúmenes de los mismos.

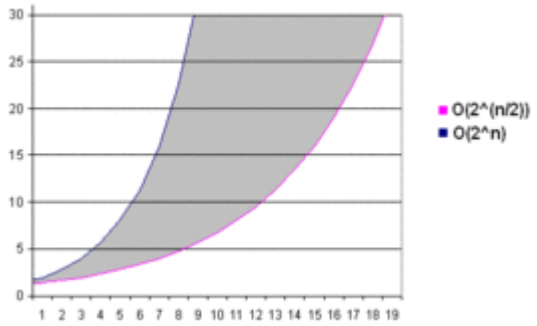
Teoría de la complejidad u orden de crecimiento de algoritmos

La teoría de la complejidad computacional es la parte de la teoría de la computación que estudia, de manera teórica, los recursos requeridos durante el cálculo para resolver un problema. Los recursos comúnmente estudiados son el tiempo (mediante una aproximación al número de pasos de ejecución de un algoritmo para resolver un problema) y el espacio (mediante una aproximación a la cantidad de memoria utilizada para resolver un problema). Se pueden estudiar igualmente otros parámetros, tales como el número de procesadores necesarios para resolver el problema en paralelo. La teoría de la complejidad difiere de la teoría de la computabilidad en que esta se ocupa de la factibilidad de expresar problemas como algoritmos efectivos sin tomar en cuenta los recursos necesarios para ello.

Los problemas que tienen una solución con orden de complejidad lineal son los problemas que se resuelven en un tiempo que se relaciona linealmente con su tamaño.

Hoy en día las máquinas resuelven problemas mediante algoritmos que tienen como máximo una complejidad o coste computacional polinómico, es decir, la relación entre el tamaño del problema y su tiempo de ejecución es polinómica. Éstos son problemas agrupados en el conjunto P. Los problemas con coste factorial o combinatorio están agrupados en NP. Estos problemas no tienen una solución práctica, es decir, una máquina no puede resolverlos en un tiempo razonable.

Presentación



Un problema dado puede verse como un conjunto de preguntas relacionadas, donde cada pregunta se representa por una cadena de caracteres de tamaño finito. Por ejemplo, el problema factorización entera se describe como: Dado un entero escrito en notación binaria, retornar todos los factores primos de ese número. Una pregunta sobre un entero específico se llama una *instancia*, por ejemplo, "Encontrar los factores primos del número 15" es una instancia del problema factorización entera.

La complejidad en tiempo de un problema es el número de pasos que toma resolver una instancia de un problema, a partir del tamaño de la entrada utilizando el algoritmo más eficiente a disposición. Intuitivamente, si se toma una instancia con entrada de longitud n que puede resolverse en n^2 pasos, se dice que ese problema tiene una complejidad en tiempo de n^2. Por supuesto, el número exacto de pasos depende de la máquina en la que se implementa, del lenguaje utilizado y de otros factores. Para no tener que hablar del costo exacto de un cálculo se utiliza la notación O. Cuando un problema tiene costo en tiempo $O(n^2)$ en una configuración de computador y lenguaje

dado este costo será el mismo en todos los computadores, de manera que esta notación generaliza la noción de coste independientemente del equipo utilizado.

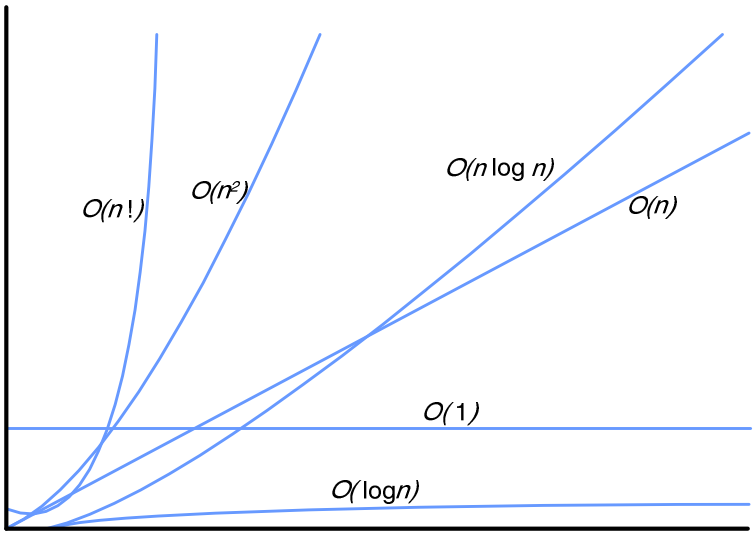
Ejemplos

Extraer cualquier elemento de un vector. La indexación en un vector o array, lleva el mismo tiempo sea cual fuere el índice que se quiera buscar. Por tanto es una operación de complejidad constante *O(1)*.

Buscar en un diccionario tiene complejidad logarítmica. Se puede iniciar la búsqueda de una palabra por la mitad del diccionario. Inmediatamente se sabe si se ha encontrado la palabra o, en el caso contrario, en cuál de las dos mitades hay que repetir el proceso (es un proceso recursivo) hasta llegar al resultado. En cada (sub)búsqueda el problema (las páginas en las que la palabra puede estar) se ha reducido a la mitad, lo que se corresponde con la función logarítmica. Este procedimiento de búsqueda (conocido como búsqueda binaria) en una estructura ordenada tiene complejidad logarítmica *O(ln n)*.

El proceso más común para ordenar un conjunto de elementos tiene complejidad cuadrática. El procedimiento consiste en crear una colección vacía de elementos. A ella se añade, en orden, el menor elemento del conjunto original que aún no haya sido elegido, lo que implica hacer un recorrido completo del conjunto original (*O(n)*, siendo *n* el número de elementos del conjunto). Este recorrido sobre el conjunto original se realiza hasta que todos sus elementos están en la secuencia de resultado. Se puede ver que hay que hacer *n* selecciones (se ordena todo el conjunto) cada una con un coste *n* de ejecución: el procedimiento es de orden cuadrático *O(n²)*. Hay que aclarar que hay diversos algoritmos de ordenación con mejores resultados.

Expresión	Nombre
$O(1)$	constante
$O(\log n)$	logarítmico
$O(n)$	lineal
$O(n \log n)$	n log n
$O(n^2)$	cuadrático
$O(n^3)$	cúbico
$O(n^c)$	polinómico
$O(c^n)$	exponencial
$O(n!)$	factorial



Comparación de los diferentes órdenes de complejidad

La siguiente tabla muestra una comparación entre las distintas funciones que demuestra como la complejidad de las funciones afecta el tiempo de ejecución de un algoritmo.

Se asume que al algoritmo le toma exactamente un segundo para procesar cada elemento.

complejidad	16 elementos	32 elementos	64 elementos	128 elementos
$O(\log n)$	4 segundos	5 segundos	6 segundos	7 segundos
$O(n)$	16 segundos	32 segundos	64 segundos	128 segundos
$O(n \log n)$	64 segundos	160 segundos	384 segundos	896 segundos
$O(n^2)$	256 segundos	17 minutos	68 minutos	273 minutos
$O(n^3)$	68 minutos	546 minutos	73 horas	24 días
$O(2^n)$	18 horas	136 años	500.000 milenios	-

Problemas de decisión

La mayor parte de los problemas en teoría de la complejidad tienen que ver con los problemas de decisión, que corresponden a poder dar una respuesta positiva o negativa a un problema dado. Por ejemplo, el problema *ES-PRIMO* se puede describir como: *Dado un entero, responder si ese número es primo o no*. Un problema de decisión es equivalente a un lenguaje formal, que es un conjunto de palabras de longitud finita en un lenguaje dado. Para un problema de decisión dado, el lenguaje equivalente es el conjunto de entradas para el cual la respuesta es positiva.

Los problemas de decisión son importantes porque casi todo problema puede ser transformado en un problema de decisión. Por ejemplo el problema *CONTIENE-FACTORES* descrito como: Dados dos enteros n y k, decidir si n tiene algún factor menor que k. Si se puede resolver *CONTIENE-FACTORES* con una cierta cantidad de recursos, su solución se puede utilizar para resolver *FACTORIZAR* con los mismos recursos, realizando una búsqueda binaria sobre k hasta encontrar el más pequeño factor de n, luego se divide ese factor y se repite el proceso hasta encontrar todos los factores.

En teoría de la complejidad, generalmente se distingue entre soluciones positivas o negativas. Por ejemplo, el conjunto NP se define como el conjunto de los problemas en donde las respuestas positivas pueden ser verificadas muy rápidamente (es decir, en tiempo polinómico). El conjunto Co-P es el conjunto de problemas donde las respuestas negativas pueden ser verificadas rápidamente. El prefijo "Co" abrevia "complemento". El complemento de un problema es aquel en donde las respuestas positivas y negativas están intercambiadas, como entre *ES-COMPUESTO* y *ES-PRIMO*.

Un resultado importante en teoría de la complejidad es el hecho de que independientemente de la dificultad de un problema (es decir de cuántos recursos de espacio y tiempo necesita), siempre habrá problemas más difíciles. Esto lo determina en el caso de los costes en tiempo el teorema de la jerarquía temporal. De éste se deriva también un teorema similar con respecto al espacio.

Clases de complejidad

Los problemas de decisión se clasifican en conjuntos de complejidad comparable llamados clases de complejidad.

La clase de complejidad P es el conjunto de los problemas de decisión que pueden ser resueltos en una máquina determinista en tiempo polinómico, lo que corresponde intuitivamente a problemas que pueden ser resueltos aún en el peor de sus casos.

La clase de complejidad NP es el conjunto de los problemas de decisión que pueden ser resueltos por una máquina no determinista en tiempo polinómico. Esta clase contiene muchos problemas que se desean resolver en la práctica, incluyendo el problema de satisfactibilidad booleana y el problema del viajante, un camino Hamiltoniano para recorrer todos los vértices una sola vez. Todos los problemas de esta clase tienen la propiedad de que su solución puede ser verificada efectivamente.

La pregunta P=NP

El saber si las clases P y NP son iguales es el más importante problema abierto en Computación teórica. Incluso hay un premio de un millón de dólares para quien lo resuelva.

Preguntas como esta motivan la introducción de los conceptos de *hard* (difícil) y *completo*. Un conjunto X de problemas es hard con respecto a un conjunto de problemas Y('Y' pertenecientes a NP) si X > Y o X=Y, es decir Y se puede escribir como un conjunto de soluciones de los problemas X. En palabras simples, Y es "más sencillo" que X. El término sencillo se define precisamente en cada caso. El conjunto hard más importante es NP-hard. El conjunto X es completo para Y si es hard para Y y es también un subconjunto de Y. El conjunto completo más importante es NP-completo. En otras palabras, los problemas del conjunto NP-completo tienen la característica de que, si se llega a encontrar una solución en tiempo P para algún miembro del conjunto (cualquiera de los problemas de NP-completo), entonces de hecho existe una solución en tiempo P para **todos** los problemas de NP-completo.

Intratabilidad

Los problemas que pueden ser resueltos en teoría, pero no en práctica, se llaman *intratables*. Qué se puede y qué no en la práctica es un tema debatible, pero en general sólo los problemas que tienen soluciones de tiempos polinomiales son solubles para más que unos cuantos valores. Entre los problemas intratables se incluyen los de EXPTIME-completo. Si NP no es igual a P, entonces todos los problemas de NP-completo son también intratables.

Para ver por qué las soluciones de tiempo exponencial no son útiles en la práctica, se puede considerar un problema que requiera 2^n operaciones para su resolución (n es el tamaño de la fuente de información). Para una fuente de información relativamente pequeña, n=100, y asumiendo que una computadora puede llevar a cabo 10^{10} (10 giga) operaciones por segundo, una solución llevaría cerca de $4*10^{12}$ años para completarse, mucho más tiempo que la actual edad del universo.

Bubble Sort

Bubble Sort, o método de la burbuja o de intercambio directo es uno de los métodos más simples y elementales, pero también uno de los más lentos y poco recomendables.

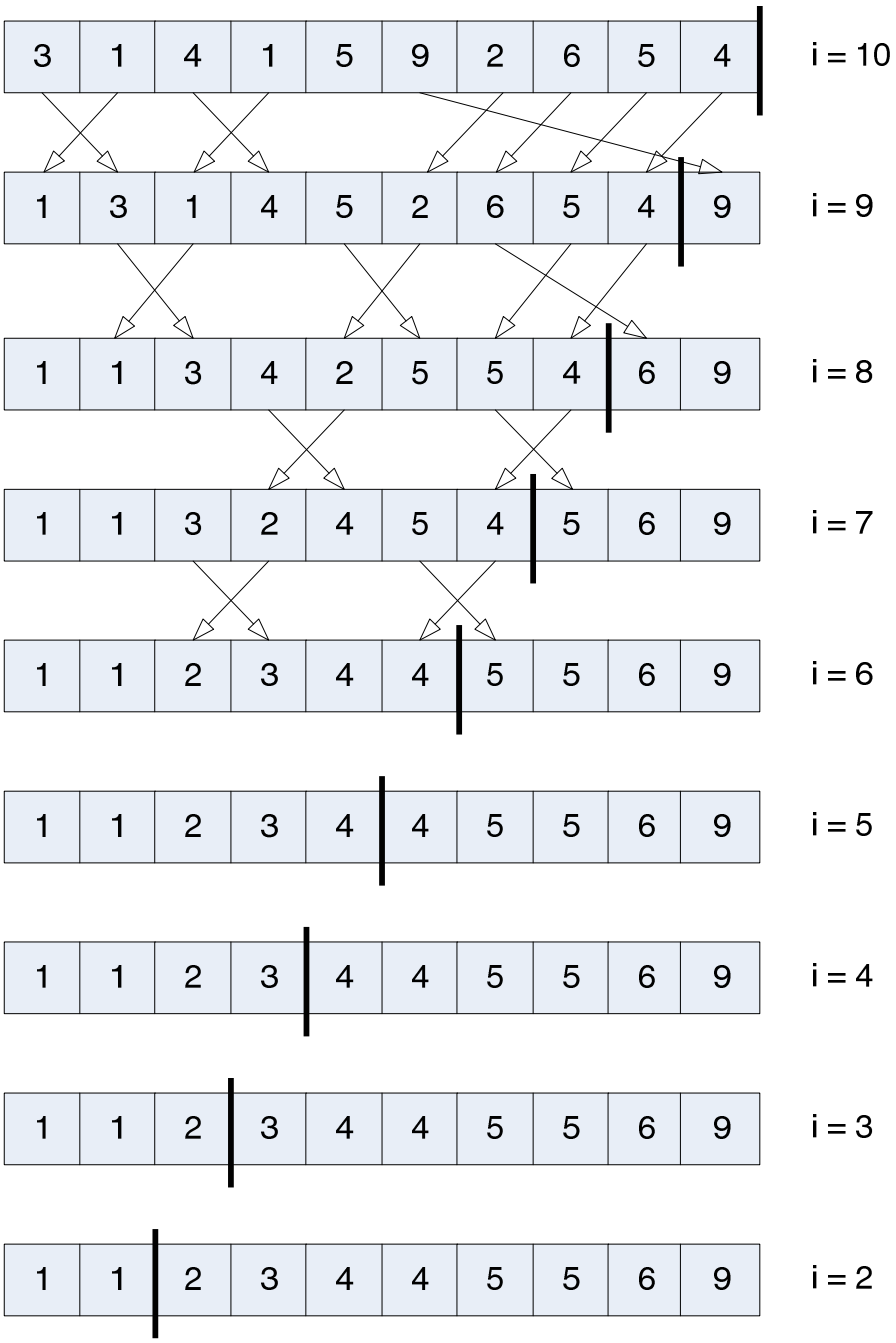
Consiste en hacer $N-1$ pasadas sobre los datos, donde en cada paso, los elementos adyacentes son comparados e intercambiados si es necesario.

Durante la primer pasada el elemento más grande va "burbujeando" a la última posición del array. En general, luego de K pasadas por el array, los últimos K elementos del array se consideran bien ordenados por lo que no se los tendrá en cuenta.

A veces no hace falta hacer $N-1$ pasadas sobre el array para que éste quede ordenado, sino que puede quedar ordenado antes de terminar todas las pasadas.

El tiempo de ejecución del Bubble Sort en el peor de los casos es de $O(n^2)$, y ocurre cuando el array viene en orden inverso. Sin embargo hay un caso en el que el Bubble Sort puede ordenar en tiempo lineal, y es cuando el array esta previamente ordenado, resultando en un tiempo de ejecución de $O(n)$.

```
void bubbleSort(int[] a) {

    for (int i = a.length ; i > 1 ; i--) {

        for (int j = 0; j < i - 1 ; j++)

            if ( a[j + 1] < a[j]) {
                intercambiar(a, j + 1, j);
            }

        }
    }
}

void intercambiar( int[] a, int posicion1, int posicion2) {

    int aux;

    aux = a[posicion1];
    a[posicion1] = a[posicion2];
    a[posicion2] = aux;
}
```



En la figura se puede ver el alcance del bucle externo delimitado con una línea gruesa. Del lado derecho de esa línea los elementos ya están en su posición final.

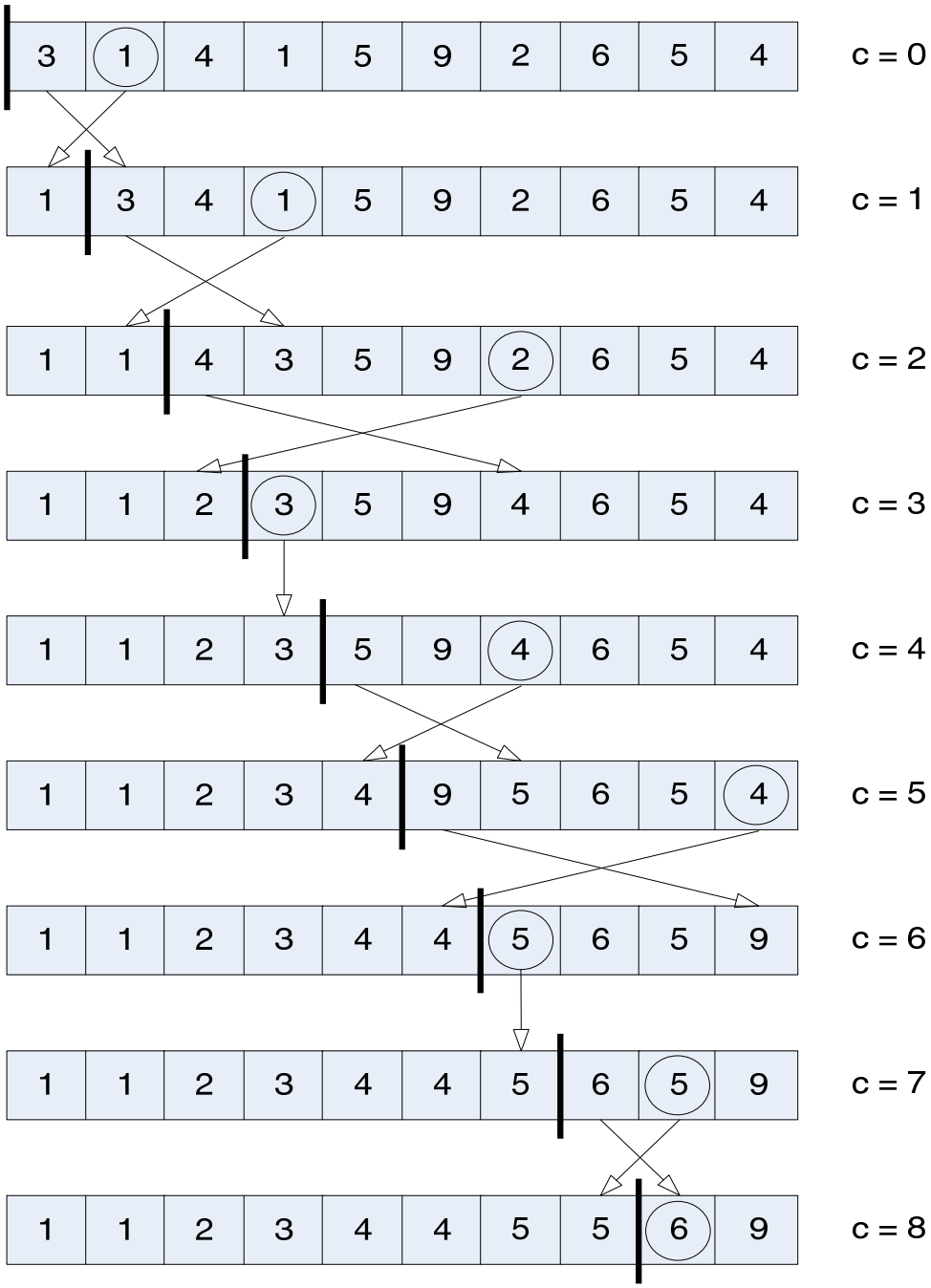
Selection Sort

Selection Sort u Clasificación por Selección, es otro de los métodos elementales, que es necesario conocer a fin de tratar luego los más complejos. Es más rápido que Bubble Sort, y no es mucho más complejo.

Comienza buscando el elemento más pequeño del array y se lo intercambia con el que esta en la primera posición, luego se busca el segundo elemento más pequeño y se lo coloca en la segunda posición. Se continua con este proceso hasta que todo el array este ordenado.

Debido a que la mayoría de los elementos se mueven a lo sumo una vez, resulta muy bueno para ordenar archivos que tienen registros muy grandes y claves muy pequeñas.

```
void selectionSort(int[] a) {

    int minPos;

    for( int c = 0 ; c < a.length - 1 ; c++ ) {

        minPos = c;

        for( int j = c + 1; j < a.length ; j++) {

            if( a[j] < a[minPos] ) {
                minPos = j;
            }
        }

        intercambiar(a, minPos, c);
    }
}
```



En la figura se puede que en el primer paso del algoritmo, luego de buscar en todo el array, se encuentra que el numero 1 es el menor de los elementos del array y ya que es el elemento más pequeño le corresponde la primera posición.

Insertion Sort

El Insertion Sort, u clasificación por inserción es el último de los métodos elementales que veremos. Es eficiente para ordenar arrays que tienen pocos elementos y están semiordenados y, en la mayoría de los casos, es más rápido que Selection Sort y Bubble Sort.

Este método se basa en la idea del ordenamiento parcial, en el cual hay un marcador que apunta a una posición donde a su izquierda se considera que están los elementos parcialmente ordenados, es decir ordenados entre ellos pero no necesariamente en sus posiciones finales.

El algoritmo comienza eligiendo el elemento marcado para poder insertarlo en su lugar apropiado en el grupo parcialmente ordenado, para eso sacamos temporalmente al elemento marcado y movemos los restantes elementos hacia la derecha. Nos detenemos cuando el elemento a ser cambiado es más pequeño que el elemento marcado, entonces ahí se intercambian el elemento que esta en esa posición con la del elemento marcado.

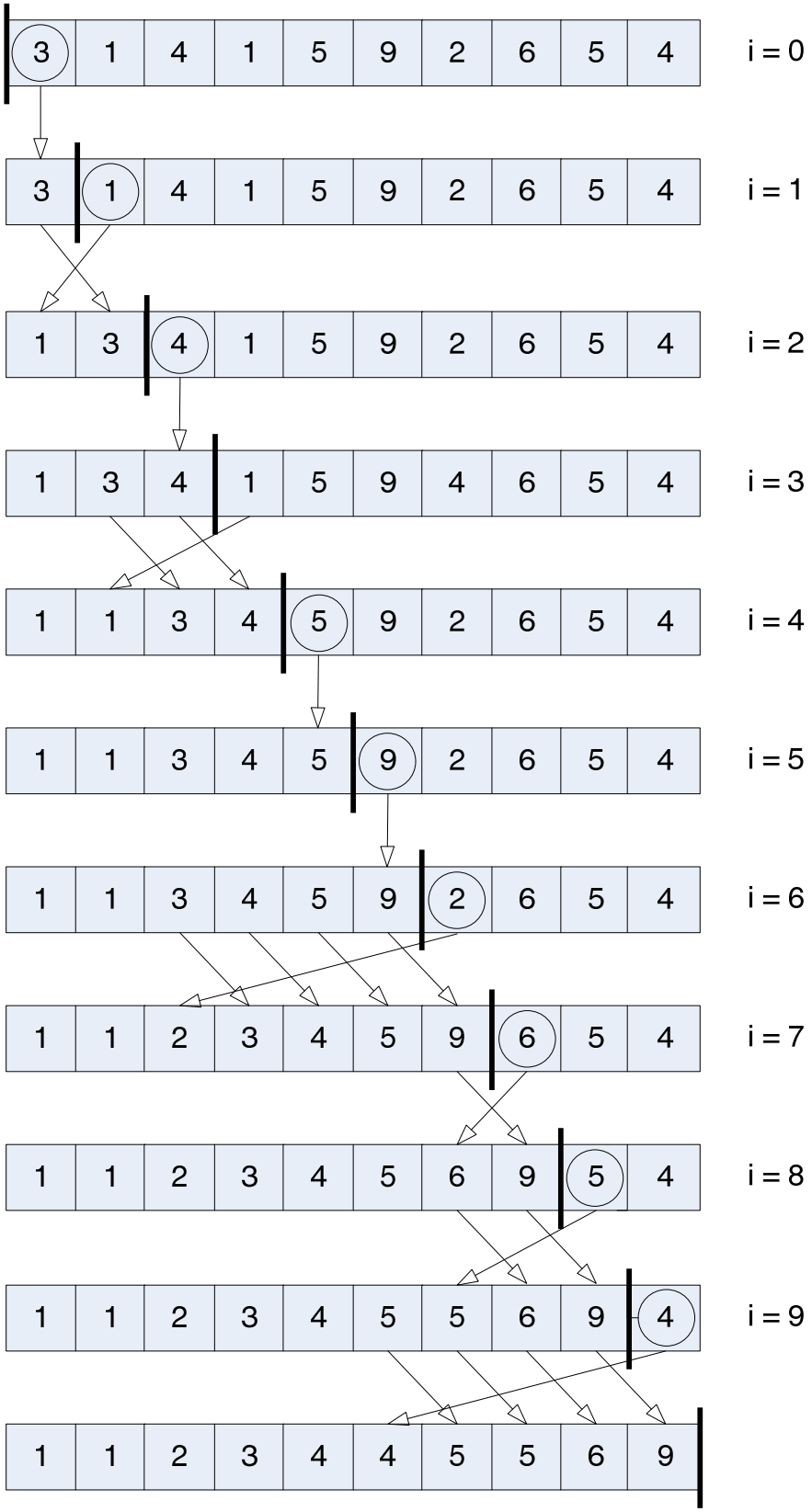
El tiempo de ejecución es de $O(n^2)$ y es alcanzable si el array viene ordenado en orden inverso.

```
void insertionSort(int[] a) {

        for( int i = 1 ; i < a.length ; i++) {

            int j = i;
            int marcado = a[i];

            while( j > 0 && a[j-1] > marcado) {
                a[j] = a[j - 1];
                j--;
            }

            a[j] = marcado;
        }
}
```

Si el array está ordenado, el bucle interno no se ejecuta, tomando un tiempo $O(n)$.

Hay una variante llamada Binary Insertion Sort, donde se tiene en cuenta que la clasificación por inserción usa una búsqueda lineal para encontrar la posición del elemento marcado. Binary Insertion Sort propone usar una búsqueda binaria dentro del grupo de elementos de la izquierda, el cual ya se encuentra ordenado. Con esta modificación requiere solo $O(\log n)$ comparaciones para hallar su posición.

```
void binaryInsertionSort(int[] a) {

    for( int i = 1 ; i < a.length ; i++) {

        int marcado = a[i];
        int izq = 0;
        int der = i;

        while( izq < der ) {

            int mitad = ( izq + der ) / 2;

            if ( a[mitad] < marcado )
                izq = mitad + 1;
            else
                der = mitad;
            }

            for ( int j = i; j > izq; j--) {
                intercambiar( a, j - 1 , j );
            }
        }
    }
}
```


3 1 4 1 5 9 2 6 5 4
i = 0
3 1 4 1 5 9 2 6 5 4
i = 1
1 3 4 1 5 9 2 6 5 4
i = 2
1 3 4 1 5 9 4 6 5 4
i = 3
1 1 3 4 5 9 2 6 5 4
i = 4
1 1 3 4 5 9 2 6 5 4
i = 5
1 1 3 4 5 9 2 6 5 4
i = 6
1 1 2 3 4 5 9 6 5 4
i = 7
1 1 2 3 4 5 6 9 5 4
i = 8
1 1 2 3 4 5 5 6 9 4
i = 9
1 1 2 3 4 4 5 5 6 9

Shell Sort

Es un método de clasificación que mejora de forma notablemente al método de inserción. Su nombre se debe a Donald Shell quien lo desarrolló en 1959.

Shell notó que el Insertion Sort era lento por la cantidad de veces que tenía que intercambiar elementos adyacentes, por lo que hizo una modificación ganando velocidad al intercambiar elementos que estén muy distantes. Por ejemplo si hay un elemento muy pequeño muy a la derecha, justo en el lugar donde tendrían que estar los elementos más grandes, para moverlo hacia izquierda se necesitaría hacer cerca de N copias para llegar a la posición indicada.

No todos los ítems deben ser movidos N espacios, pero en promedio deben moverse $N/2$ lugares. Por lo tanto lleva N veces $N/2$ cambios de lugar, resultando en $N^2/2$ copias. Por lo cual el tiempo de ejecución es de $O(n^2)$.

La idea de Shell fue evitar la gran cantidad de movimientos, para eso primero compara los elementos que más lejanos y luego va comparando elementos más cercanos para finalmente realizar un Insertion Sort.

Para lograr esto se utiliza una secuencia $H_1, H_2, \ldots, H_t$ denominada secuencia de incrementos. Es importante remarcar que cualquier secuencia es válida siempre que $H_1 = 1$, es decir que termine realizando un ordenamiento por inserción. Algunas secuencias son mejores que otras, pero todas logran ordenar el array.

Después de ordenar usando la secuencia h_k, se puede afirmar que $a[i] \leq a[i + h_k]$ para todo i que este dentro del array. En otras palabras, todos los elementos separados por una distancia de h_k estarán ordenados y por lo tanto se dice que esta h-ordenado.

Original	3	1	4	1	5	9	2	6	5	4
Después de 5-ordenación	3	1	4	1	4	9	2	6	5	5
Después de 3-ordenación	1	1	4	2	4	5	3	6	9	5
Después de 1-ordenación	1	1	2	3	4	4	5	5	6	9

Shell Sort después de cada paso luego de la secuencia de incrementos (1,3,5).

Una forma de implementar esto es aplicar Insertion Sort dentro de cada subvector h_k independiente. Cuando el $h_k = 1$ el método de ordenamiento es igual a un Insertion Sort. Esto se puede lograr modificándolo, para que incremente o decremente de a h unidades.

Ahora debemos encontrar que secuencia de incremento usar. Actualmente no se puede afirmar que exista una secuencia óptima. Cada una es probada en la práctica y se estudian los resultados que arrojan.

Una primera aproximación sugerida originalmente por Shell fue la de usar un incremento un medio menor que el anterior. Si bien fue una mejora al Insertion Sort, esta secuencia no terminó siendo la mejor, ya que en algunos casos se terminaba teniendo un tiempo de ejecución de $O(n^2)$.

Otra aproximación, basada puramente en la experimentación, consiste en dividir cada intervalo por la constante 2,2 en vez de por 2, lo cual consigue un tiempo de ejecución promedio debajo de $O(n^{5/4})$. Por ejemplo para $N = 100$ se obtiene la siguiente secuencia: 45, 20, 9, 4, 1.

Una de las secuencias que arroja mejores resultados es la secuencia de Knuth, se obtiene con la siguiente expresión recursiva: $H = 3H + 1$. Cuando el valor inicial de $H = 1$ Por ejemplo: 1, 4, 13, 40, 121, 364, 1093,…

En el algoritmo de ordenamiento esta secuencia se aplica de la siguiente forma. Primero se determina cual va a ser el h inicial. Se usa la formula $H = 3H + 1$. El proceso termina cuando se encuentra un h más grande que el tamaño del array. Por ejemplo para un array de 1000 elementos, el 7mo elemento (1093) es muy grande, por lo que se usa el 6^{to} (364). De ahí en más se usa la inversa de la formula indicada anteriormente $H = (h - 1)/3$. Cada vez en que iteremos el bucle exterior se disminuye el intervalo usando la formula inversa, y nos detenemos cuando el array fue 1-Ordenado.

Si bien no hay un consenso sobre la eficiencia del Shell Sort, se considera que este varia entre $O(n^{3/2})$ y $O(n^{7/6})$.

```
void shellSortKnuth(int[] a) {

    int h = 1;
    while(h <= a.length/9)
        h = h*3 + 1;

    while ( h > 0 ) {

        for (int i = h; i < a.length; i += h) {

            int j = i;
            int marcado = a[i];

            while( j > 0 && a[j - h] > marcado) {
                a[j] = a[j - h];
                j -= h;
            }

            a[j] = marcado;
        }

        h = h /3;
    }
}
```

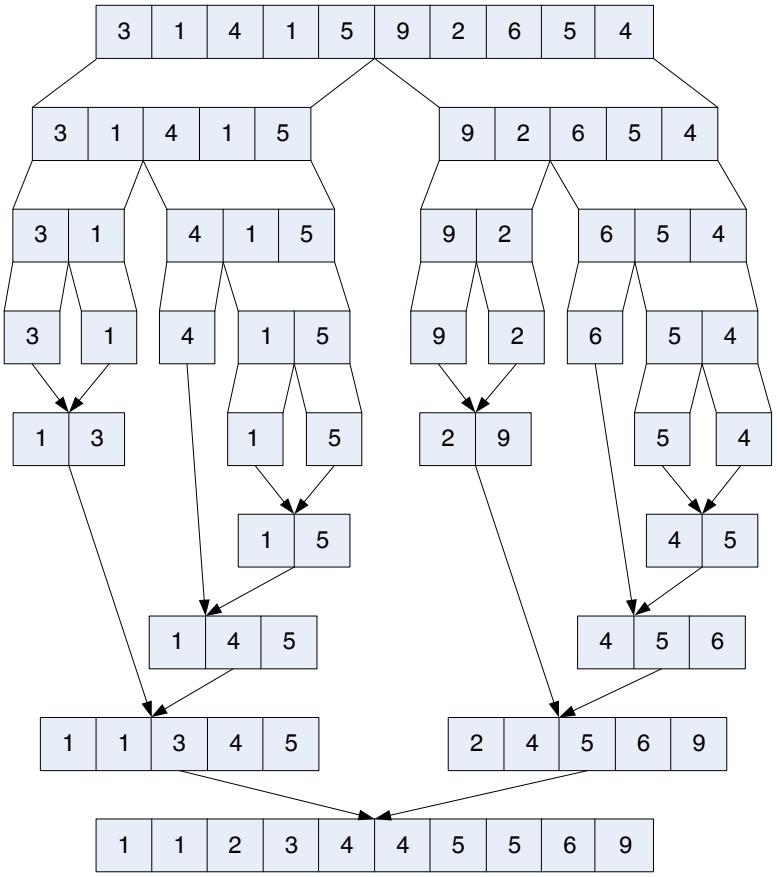
Aquí se muestra un ejemplo de una implementación de Shell Sort, usando la secuencia de incrementos de Knuth.

Merge Sort

Merge Sort, u clasificación por fusión, fue desarrollado por John Von Neumann en el año 1945. Es un algoritmo recursivo que utiliza la técnica de divide y vencerás para obtener un tiempo de ejecución $O(n \log n)$ sin importar cual sea la entrada. Se basa en la fusión de dos o más secuencias ordenadas en una única secuencia ordenada. Una de las desventajas de este algoritmo es que requiere de memoria extra proporcional a la cantidad de elementos del array. Es un algoritmo a considerar si estamos buscando velocidad, estabilidad, donde no se tolera un 'peor de los casos' y además disponemos de memoria extra. Algo que hace más atractivo a Merge Sort es que suele acceder de forma secuencial a los elementos y es de gran utilidad para ordenar en ambientes donde solo se dispone de acceso secuencial a los registros.

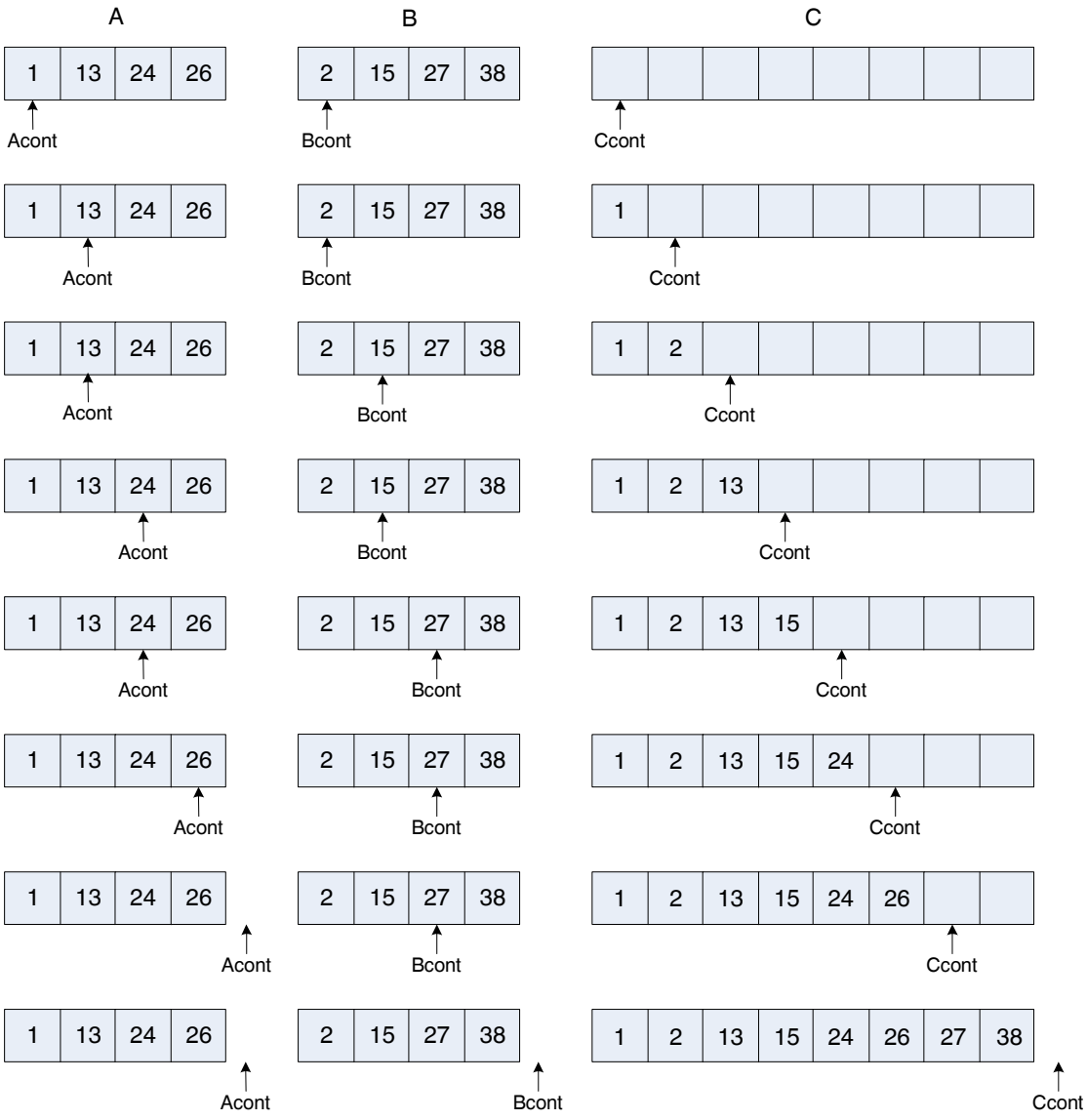
El algoritmo tiene como caso base una secuencia con exactamente un elemento en ella. Y ya que esa secuencia esta ordenada, no hay nada que hacer. Por lo tanto para ordenar una secuencia $n > 1$ elementos se deben seguir los siguientes pasos:

1- Dividir la secuencia en dos subsecuencias más pequeñas.
2- Ordenar recursivamente las dos subsecuencias
3- Fusionar las subsecuencias ordenadas para obtener el resultado final.



Fusión lineal de dos arrays

El algoritmo de fusión básico utiliza dos arrays de entrada, *A* y *B* y produce un array de salida *C*, utilizando tres contadores, *Acont*, *Bcont* y *CCont*, inicializados en cero. En cada paso el menor de los elementos de *A*[*Acont*] y *B*[*Bcont*] se copia en la próxima posición de *C*, actualizándose los contadores apropiados. Cuando uno de los arrays de entrada se acaba, el resto del otro vector se copia en C.



```
void mergeSort(int[] a) {

    int tempArray[] = new int[a.length];

    mergeSortInternal(a,tempArray, 0, a.length - 1 );
}

void mergeSortInternal(int[] a, int[] tempArray, int izq, int der)
{

    if( izq < der ) {

        int centro = (izq + der) / 2;

        mergeSortInternal(a, tempArray, izq, centro);
        mergeSortInternal(a, tempArray, centro + 1, der);
        fusionar(a, tempArray, izq, centro + 1, der);
    }

}

void fusionar(int[] a, int[] tempArray, int izq, int der, int fin)
{

    int finIzq = der - 1;
    int posAux = izq;
    int numElementos = fin - izq + 1;

    while( izq <= finIzq && der <= fin )
        if( a[izq] < a[der] )
            tempArray[ posAux++ ] = a[ izq++ ];
        else
            tempArray[ posAux++ ] = a[ der++ ];


    while( izq <= finIzq )
        tempArray[ posAux++ ] = a[ izq++ ];

    while( der <= fin )
        tempArray[ posAux++ ] = a[ der++ ];

    for (int i = 0 ; i < numElementos ; i++) {
        a[fin] = tempArray[fin];
        fin--;
    }
}
```

Heap Sort

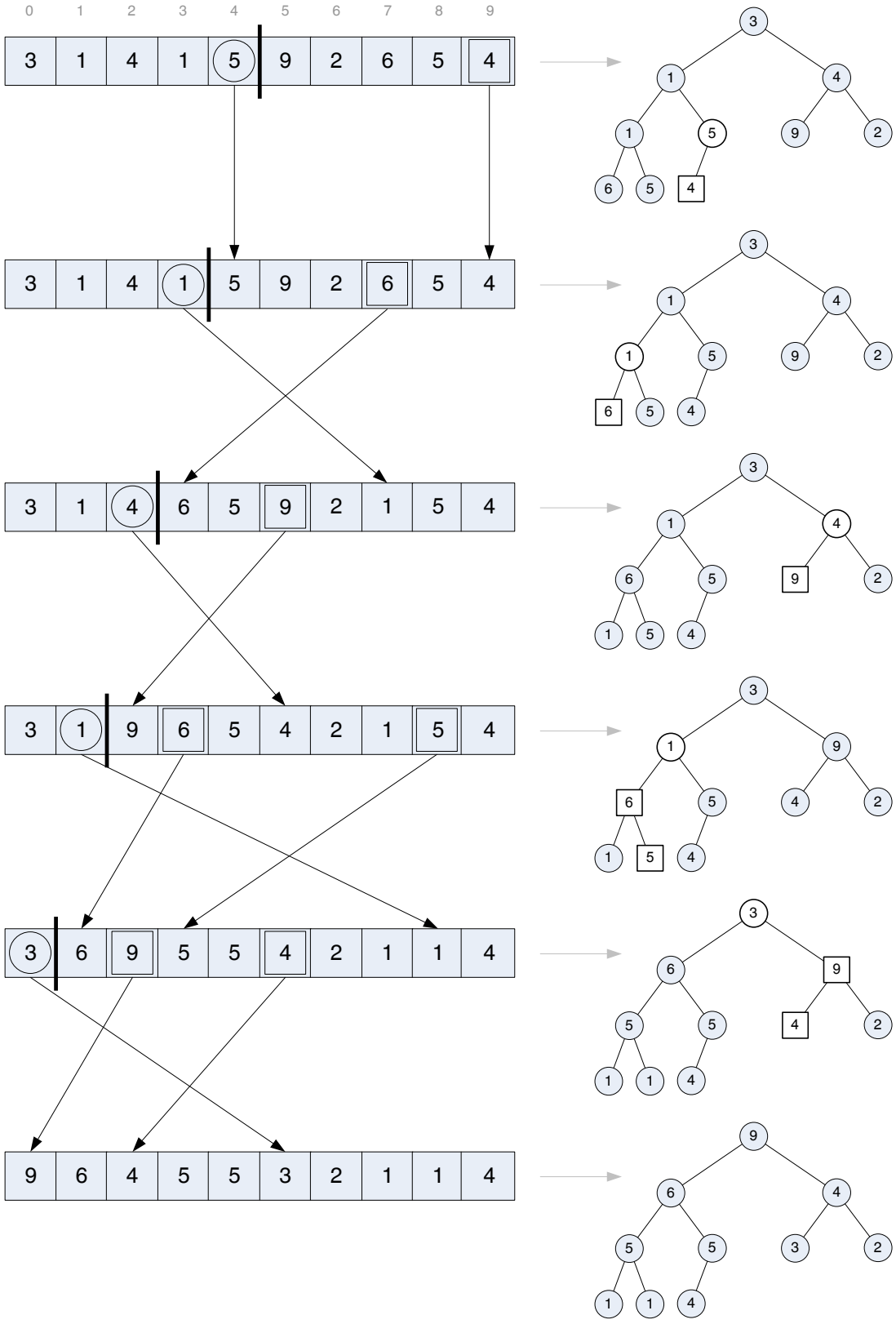
Heap Sort, u clasificación por montículo (heap), se basa en una estructura de datos llamada montículo binario (ver apunte de árboles), que es una de las formas de implementar una cola de prioridad. Más precisamente un montículo binario es un árbol binario completo representado mediante un array, en el cual cada nodo satisface la condición de montículo. Para que se cumpla la condición de montículo la clave de cada nodo debe ser mayor (o igual) a las claves de sus hijos, si es que tiene. Esto implica que la clave más grande está en la raíz. Debemos recordar que en un árbol binario completo representado como un array, tenemos la raíz en la posición 0 y los hijos del nodo de la posición i se encuentran en las posiciones $2i+1$ y $2i+2$ y el padre esta en la posición $(i-1)/2$.

Este algoritmo tiene un tiempo de ejecución que nunca supera $O(n \log n)$ y no requiere espacio de memoria adicional (in situ). Es generalmente se usa en sistemas embebidos con restricciones de tiempo real, o en sistemas en donde la seguridad es un factor importante.

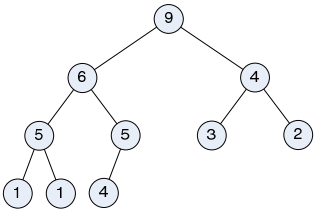
El algoritmo de Heap Sort consiste de dos fases.

En la primera fase, el array desordenado es transformado en un montículo desde abajo hacia arriba, es decir que empezamos desde las hojas y continuamos hacia la raíz. Cada nodo es la raíz de un submontículo que cumple con la condición de montículo, excepto posiblemente en su raíz. Para eso intercambiamos la raíz con el hijo más grande, este proceso continúa hacia abajo, hasta que llegamos a una posición en donde la condición de montículo se cumple, o hasta que llegamos a una hoja. Este proceso de "filtrado descendente" comienza en $(N/2)-1$, ya que es esa la posición donde hay un nodo con al menos un hijo.

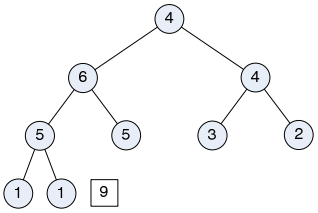
En la segunda fase, una vez construido el montículo, se continúa con la parte de clasificación. El vector ordenado se construye seleccionando el elemento más grande, quitándolo del montículo y colocándolo en la secuencia ordenada. El elemento más grande del montículo se encuentra en la raíz, y siempre está en la posición 0 del array. Se intercambia repetidamente el elemento más grande del montículo por el próximo elemento de la posición en la secuencia ordenada. Después de cada intercambio, hay un nuevo valor en la raíz del montículo y el nuevo valor es filtrado hacia abajo hasta su posición correcta en el montículo, para que vuelva a cumplir con la condición de montículo.



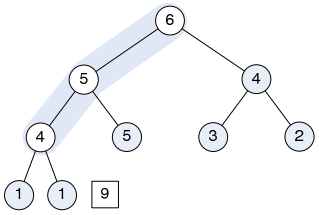
Fase de construcción del montículo



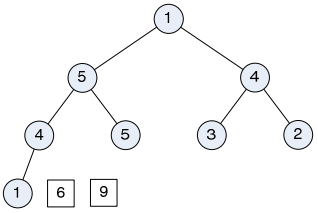
1



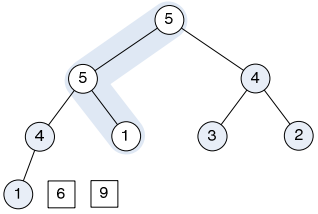
2



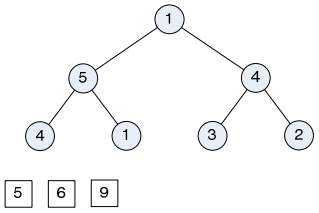
3



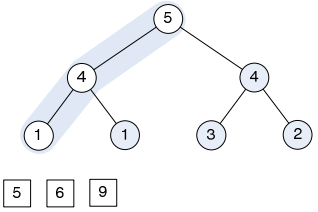
4



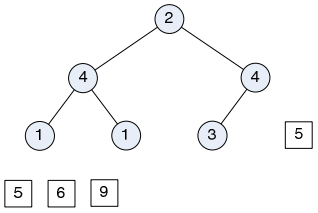
5



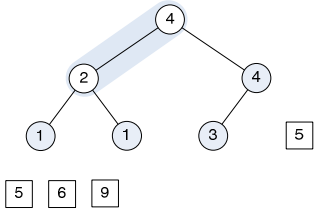
6



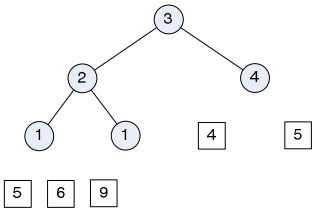
7



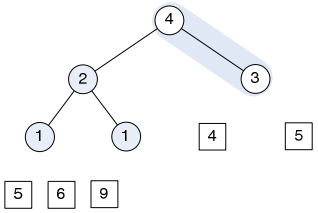
8



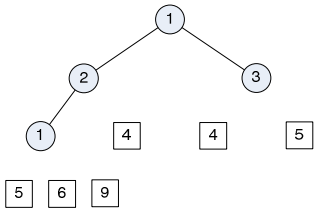
9



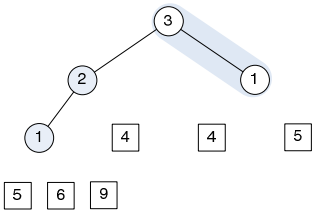
10



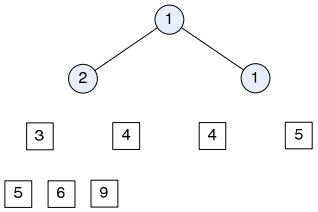
11



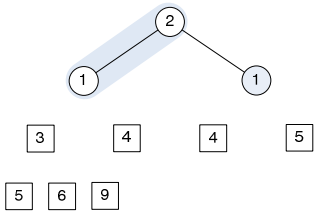
12



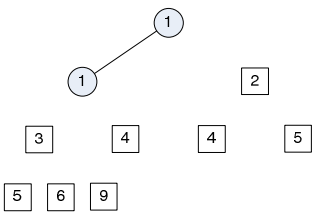
13



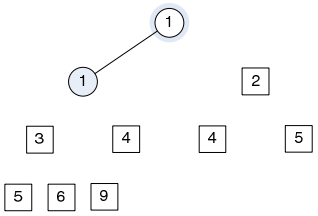
14



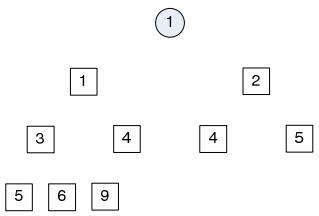
15



16



17



18

Fase de ordenación

```
void heapsort(int[] a) {

    int ultimaPosHeap = a.length;

    construirMonticulo(a, ultimaPosHeap);

    while ( ultimaPosHeap > 1 ) {

        ultimaPosHeap--;
        intercambiar(a, 0, ultimaPosHeap);
        filtradoDescendente(a, 0, ultimaPosHeap);
    }
}

void construirMonticulo(int[] a, int n) {

    for (int i = n/2 - 1 ; i >= 0 ; i--)
        filtradoDescendente(a, i, n);
}

void filtradoDescendente(int[] a, int posNodo, int ultimaPosHeap) {

    int posNodoHijo = 2*posNodo + 1;

    while ( posNodoHijo < ultimaPosHeap ) {

        if ( posNodoHijo + 1 < ultimaPosHeap )
            if (a[ posNodoHijo + 1 ] > a[ posNodoHijo ] )
                posNodoHijo++;

        if ( a[ posNodo ] >= a[ posNodoHijo ] )
            return;

        intercambiar(a, posNodo, posNodoHijo);

        posNodo = posNodoHijo;
        posNodoHijo = 2*posNodo + 1;
    }
}
```

QuickSort

Es el algoritmo usado más ampliamente en la actualidad y el más estudiado. Su popularidad se debe a su relativa facilidad de implementación y a su poco consumo de recursos. Fue desarrollada por C.A.R. Hoare en 1960.

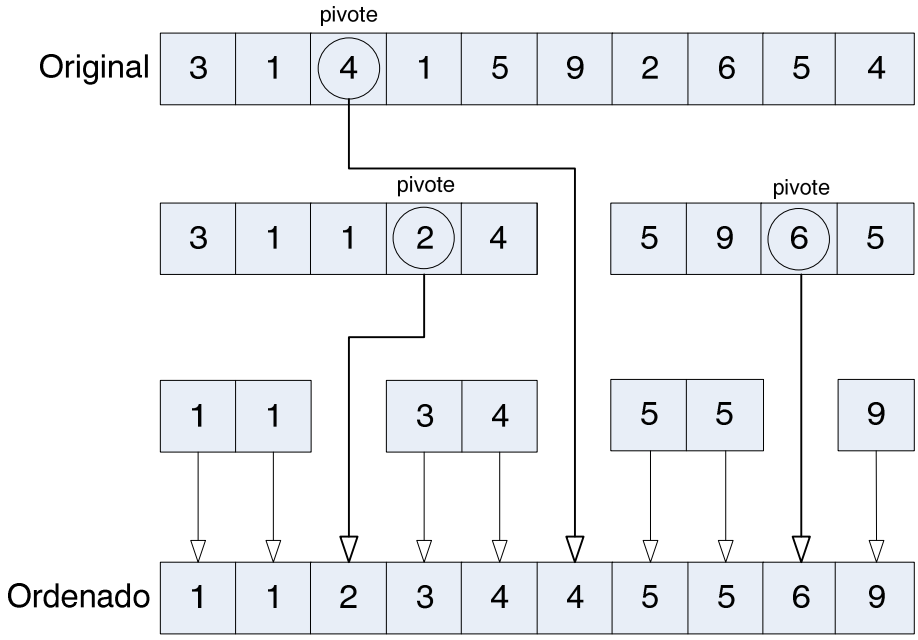
Es el algoritmo que mejor responde en la mayoría de los casos, con un tiempo promedio de $O(n \log n)$ y $O(n^2)$ en el peor de los casos.

Es recursivo pero se usan versiones iterativas para mejorar su rendimiento, también es in-situ, ya que usa solo una pila auxiliar.

El QuickSort está basado en la idea de divide y vencerás, en el cual un problema se soluciona dividiéndolo en dos o más subproblemas, resolviendo recursivamente cada uno de ellos para luego juntar sus soluciones para obtener la solución del original.

El algoritmo básico de QuickSort(S) consiste en los siguientes cuatro pasos:

1. Si el número de elementos de S es 0 o 1, terminar.
2. Elegir un elemento *v del* conjunto S, llamado pivote.
3. Particionar S en dos grupos, donde $I = \{ x \in S - \{v\} \mid x \leq v \}$ y $D = \{ x \in S - \{v\} \mid x \geq v \}$
4. Devolver el resultado de QuickSort(I), seguido de v y seguido del resultado de QuickSort(D)



En el ejemplo se elige un pivote cualquiera, en este caso el 4, luego se agrupan los elementos que son menores que el pivote en el subgrupo izquierdo, y los mayores en el derecho. Luego se aplica QuickSort a los subgrupos resultantes. Es importante remarcar que la posición donde va el pivote luego de realizar la partición, es su posición definitiva en el array ordenado.

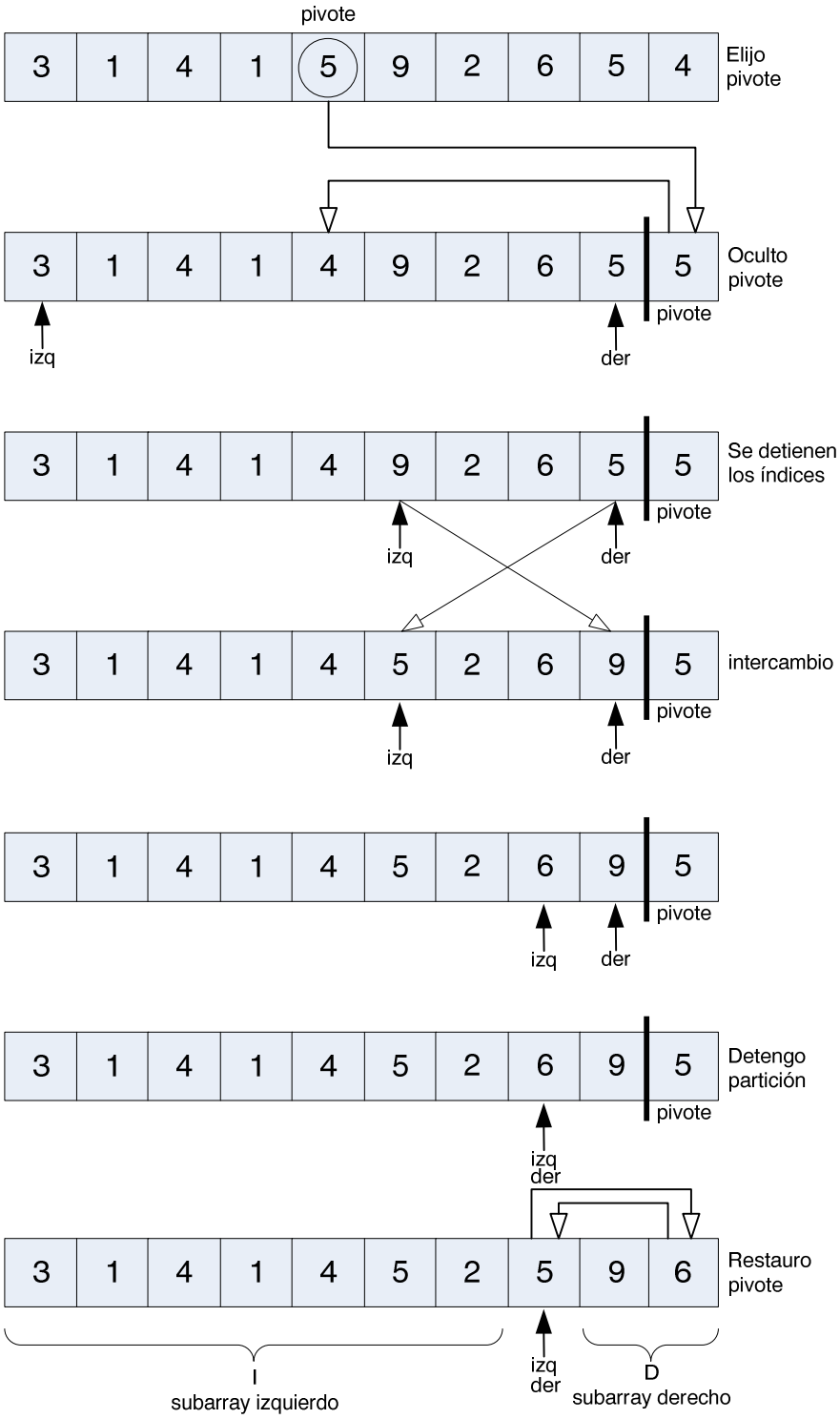
Estrategia de partición

Existen varias estrategias para particionar el array, una de las más usadas por ser in-situ y por su eficiencia es la siguiente. Se elige un pivote y se lo intercambia con el último elemento. Se tienen dos índices, *izq* que se mueve hacia la derecha y *der* que se mueve hacia la izquierda. El índice *izq* explora el array de derecha a izquierda y se detiene cuando encuentra un elemento mayor o igual al pivote. El índice *der* hace lo mismo en sentido contrario y se detiene al encontrar un elemento menor o igual al pivote. Los dos elementos en los que se detiene el proceso están mal ubicados y por lo tanto se los intercambia.

Continuando de esta forma se tiene la seguridad que los elementos que están a la izquierda del índice *izq* son menores que el pivote y que los situados a la derecha de *der* son mayores.

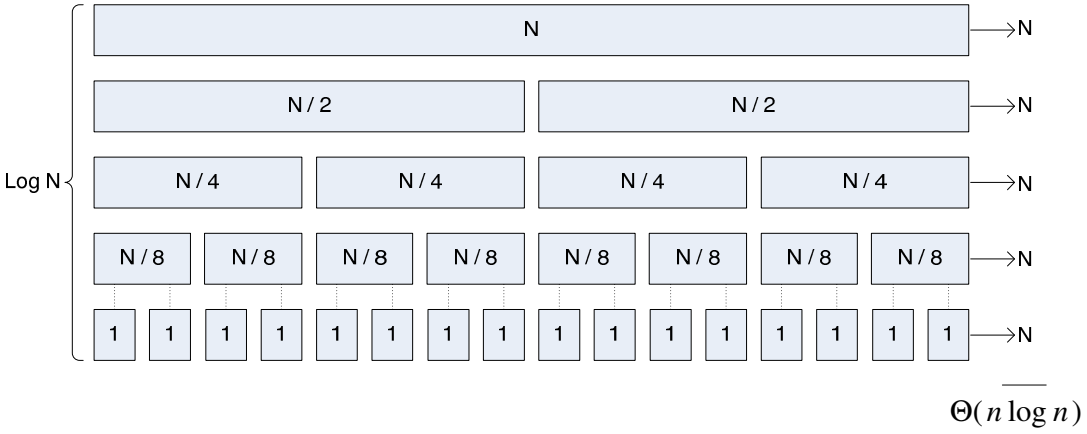
Cuando los índices se encuentran el proceso se detiene, y se intercambia el ultimo elemento con el elemento al que apunta el índice *izq*. Ahora el array esta listo para llamar a QuickSort con los dos subconjuntos resultantes, es decir que se llamará a QuickSort(I) y a QuickSort(D).

Vale la pena observar que el algoritmo no necesita memoria auxiliar más allá de los índices, y que cada elemento es comparado con el pivote exactamente una vez.


pivote
3 1 4 1 5 9 2 6 5 4
Elijo pivote
3 1 4 1 4 9 2 6 5 5
Oculto pivote
pivote
izq
der
3 1 4 1 4 9 2 6 5 5
Se detienen los índices
pivote
izq
der
3 1 4 1 4 5 2 6 9 5
intercambio
pivote
izq
der
3 1 4 1 4 5 2 6 9 5
pivote
izq
der
3 1 4 1 4 5 2 6 9 5
Detengo partición
pivote
izq
der
3 1 4 1 4 5 2 5 9 6
Restauro pivote
izq
der
I
subarray izquierdo
D
subarray derecho

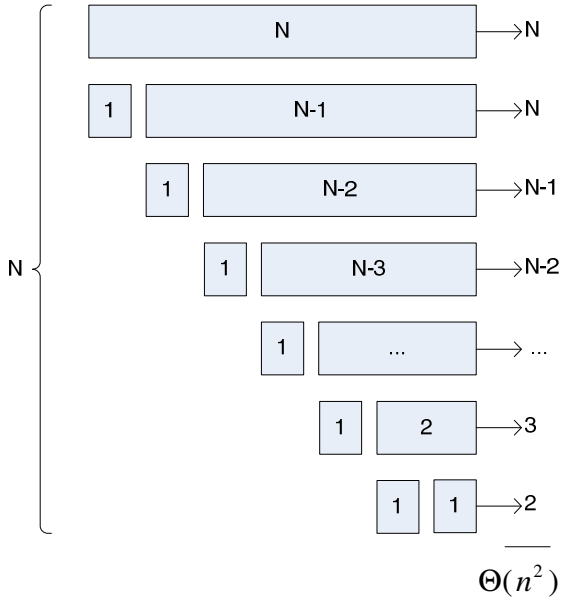
Mejor de los casos

El mejor de los casos se obtiene cuando el proceso de particionamiento determina dos subsecuencias con cantidad de elementos iguales, es decir de $N/2$. Este caso produce un tiempo de ejecución cercano a $O(n \log n)$.



Peor de los casos

El peor de los casos se da cuando el proceso de particionamiento produce una subsecuencia con $N-1$ elementos y otra con un solo elemento. Si esto ocurre en cada paso del algoritmo, dado que el particionamiento cuesta un tiempo N, el tiempo de ejecución es de $\Theta(n^2)$.



Seleccionando el pivote

El objetivo central de una buena elección de pivote es evitar que ocurra el peor de los casos. Si el array a ordenar esta conformado por una permutación aleatoria, es casi seguro que el tiempo de ejecución para ordenarlo sea cercano al caso medio. Pero si estamos frente a una entrada degenerada, como en el caso en el que los datos ya estén ordenados o que todos los elementos sean iguales, nos pueden llevar a tiempos de ejecución malos.

Una mala elección seria elegir el primer elemento como pivote, que si bien sería aceptable en una entrada aleatoria nos llevaría al peor caso si la entrada esta ya ordenada o en orden inverso. Esto produciría a una mala partición y además se repetiría recursivamente este problema.

Una mejor aproximación sería elegir el elemento central como pivote, es decir en la posición $(izq + der)/2$ del array.

Otra alternativa bastante popular es la de utilizar la partición con mediana de tres, que intenta elegir un pivote mejor que el central. La mediana de un grupo de N elementos es el $N/2$-ésimo menor elemento. La mejor elección del pivote es la mediana ya que garantiza una división uniforme de los elementos. Para obtenerla se realizan muestreos, donde se toman un subconjunto de los elementos a ordenar y se busca su mediana. Se ha probado que una muestra de tres proporciona una mejora en el caso promedio del QuickSort y además simplifica el código. Los elementos que se usan para la muestra son el primero, el central, el último. Este método de partición hace altamente improbable que ocurra el peor caso, ya que los tres elementos elegidos deberían estar entre los más grandes o entre los más pequeños del array.

El uso de la partición con mediana de tres, una implementación no recursiva y un tratamiento para casos en los que el vector pequeño puede llevar a una mejora de hasta un 30% respecto del QuickSort recursivo directo.

```
void quickSort(int[] a) {

    quickSortInterno(a, 0, a.length - 1);
}

void quickSortInterno(int[] a, int izq, int der) {

    if( der <= izq)
        return;

    int pivotPos = particionar( a, izq, der);

    quickSortInterno(a, izq, pivotPos - 1);
    quickSortInterno(a, pivotPos + 1, der);
}

int particionar(int[] a, int izq, int der) {

    int medio = (izq + der) / 2;
    int i = izq - 1;
    int j = der;

    if( a[medio] < a[izq] )
        intercambiar(a, medio, izq);
    if( a[der] < a[izq] )
        intercambiar(a, der, izq);
    if( a[der] < a[medio] )
        intercambiar(a, medio, der);

    int pivot = a[medio];

    intercambiar(a, medio, der);

    while(true) {

        while( a[++i] < pivot);

        while( a[--j] > pivot)
            if( j == i)
                break;

        if( i >= j )
            break;

        intercambiar(a, i, j);
    }

    intercambiar(a, i, der);
    return i;
}
```

Otros métodos no tratados aquí

Los métodos que hemos visto hasta aquí fueron definidos en base a las operaciones básicas de comparar e intercambiar. Sin embargo hay muchos casos en los que las claves de los elementos a ordenar pueden considerarse como números de un intervalo finito. Los métodos que aprovechan las propiedades numéricas se llaman ordenaciones por residuos. Estos métodos no solo comparan las claves, sino que además procesan y compraran fragmentos de ellas. Por ejemplo si tenemos un array que sabemos de antemano que el conjunto de valores del array pertenece a un universo pequeño y constante, podemos utilizar un método llamado Bucket Sort (u ordenamiento por casilleros), de modo semejante, si tenemos un universo de elementos que pueden ser representados por un una pequeña y acotada cantidad de bits, entonces podríamos usar el Radix Sort.

Finalmente deberíamos mencionar la existencia de clasificación en paralelo, donde el array a ordenar se encuentra distribuido a lo largo de procesadores independientes y en paralelo.

Comparación de los métodos

Nombre	Mejor caso	Caso medio	Peor caso	Estable	Comentarios
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Si	El más lento de todos. Uso pedagógico.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Si	Apto si queremos que consumir siempre la misma cantidad de tiempo.
Insertion Sort	$O(n)$	$O(n)$	$O(n^2)$	Si	Conveniente cuando el array esta casi ordenado.
Shell Sort	$O(n^{5/4})$	$O(n^{3/2})$	$O(n^2)$	No	Dependiente de la secuencia de incrementos.
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Si	Adecuado para trabajos en paralelo. Requiere memoria extra.
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	El método acotado en el tiempo muy utilizado para grandes volúmenes de datos
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	El más rápido en la práctica. Implementado en gran cantidad de sistemas.

Bibliografía

Robert Lafore, Data Structures & Algorithms in Java, Sams, 1998.

Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in Java, 1998.

Robert Sedgewick, Algorithms in Java, Addison Wesley, 2002.

Thomas H. Cormen, Introduction to Algorithms (Second Edition), MIT Press, 2001.

Mark A. Weiss, Data Structures and Problem Solving Using C++, Pearson, 2003.