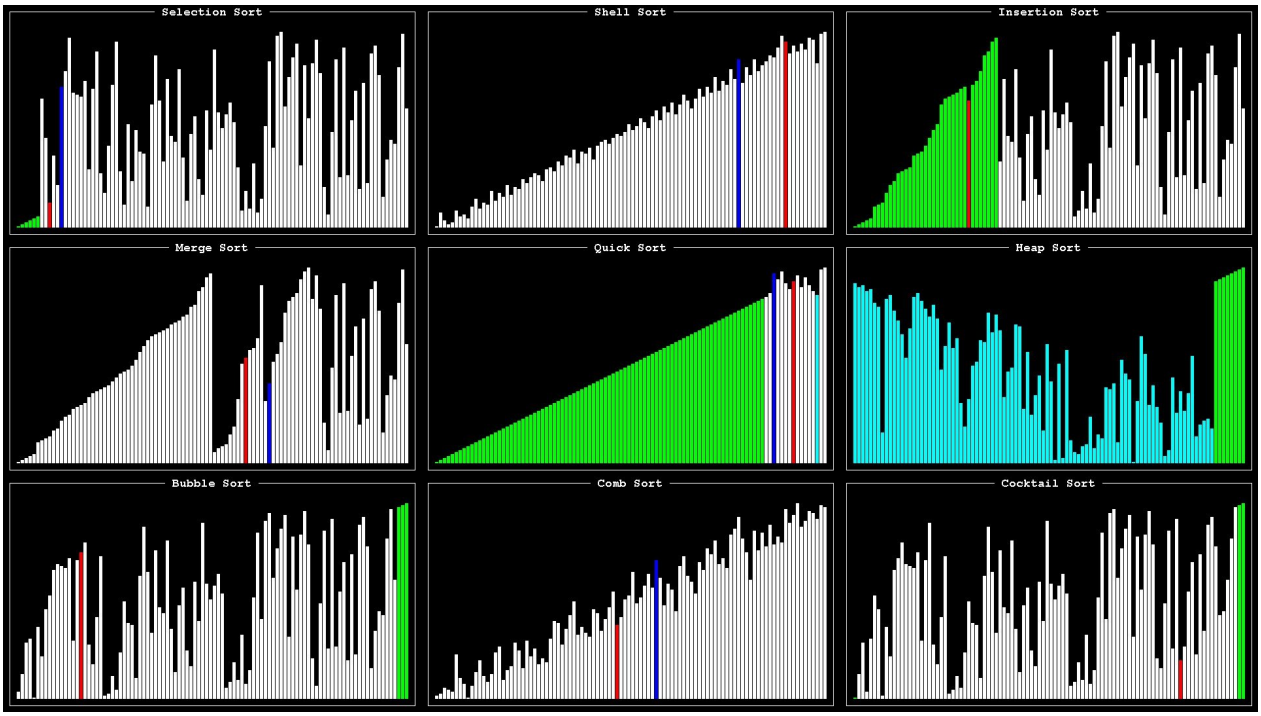
Análisis de Algoritmos de Ordenación

Alberto Robles Enciso

Febrero de 2018



Resumen

En este documento se va a analizar tres algoritmos de ordenación para determinar el orden computacional teóricamente y experimentalmente.

Índice

1. Introducción

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos. [1]

2. Clasificación

Los algoritmos de ordenamiento se pueden clasificar en las siguientes maneras:

- La más común es clasificar según el lugar donde se realice la ordenación
  - Algoritmos de ordenamiento interno: en la memoria del ordenador.
  - Algoritmos de ordenamiento externo: en un lugar externo como un disco duro.
- Por el tiempo que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:
  - Algoritmos de ordenación natural: Tarda lo mínimo posible cuando la entrada está ordenada.
  - Algoritmos de ordenación no natural: Tarda lo mínimo posible cuando la entrada está inversamente ordenada.
- Por estabilidad: un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si una lista ordenada por fecha se reordena en orden alfabético con un algoritmo estable, todos los elementos cuya clave alfabética sea la misma quedarán en orden de fecha. Otro caso sería cuando no interesan las mayúsculas y minúsculas, pero se quiere que si una clave aBC estaba antes que AbC, en el resultado ambas claves aparezcan juntas y en el orden original: aBC, AbC. Cuando los elementos son indistinguibles (porque cada elemento se ordena por la clave completa) la estabilidad no interesa. Los algoritmos de ordenamiento que no son estables se pueden implementar para que sí lo sean. Una manera de hacer esto es modificar artificialmente la clave de ordenamiento de modo que la posición original en la lista participe del ordenamiento en caso de coincidencia.

Los algoritmos se distinguen por las siguientes características:

- Complejidad computacional (peor caso, caso promedio y mejor caso) en términos de n, el tamaño de la lista o arreglo. Para esto se usa el concepto de orden de una función y se usa la notación $O(n)$. El mejor comportamiento para ordenar (si no se aprovecha la estructura de las claves) es $O(nlogn)$. Los algoritmos más simples son cuadráticos, es decir $O(n^2)$. Los algoritmos que aprovechan la estructura de las claves de ordenamiento (p. ej. bucket sort) pueden ordenar en $O(kn)$ donde k es el tamaño del espacio de claves. Como dicho tamaño es conocido a priori, se puede decir que estos algoritmos tienen un desempeño lineal, es decir $O(n)$.
- Uso de memoria y otros recursos computacionales. También se usa la notación $O(n)$.

3. Algoritmos de Ordenacion

3.1. Ordenamiento de burbuja

La Ordenación de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas"(se puede ver un ejemplo de su funcionamiento en la figura 1). También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillo de implementar. [3]

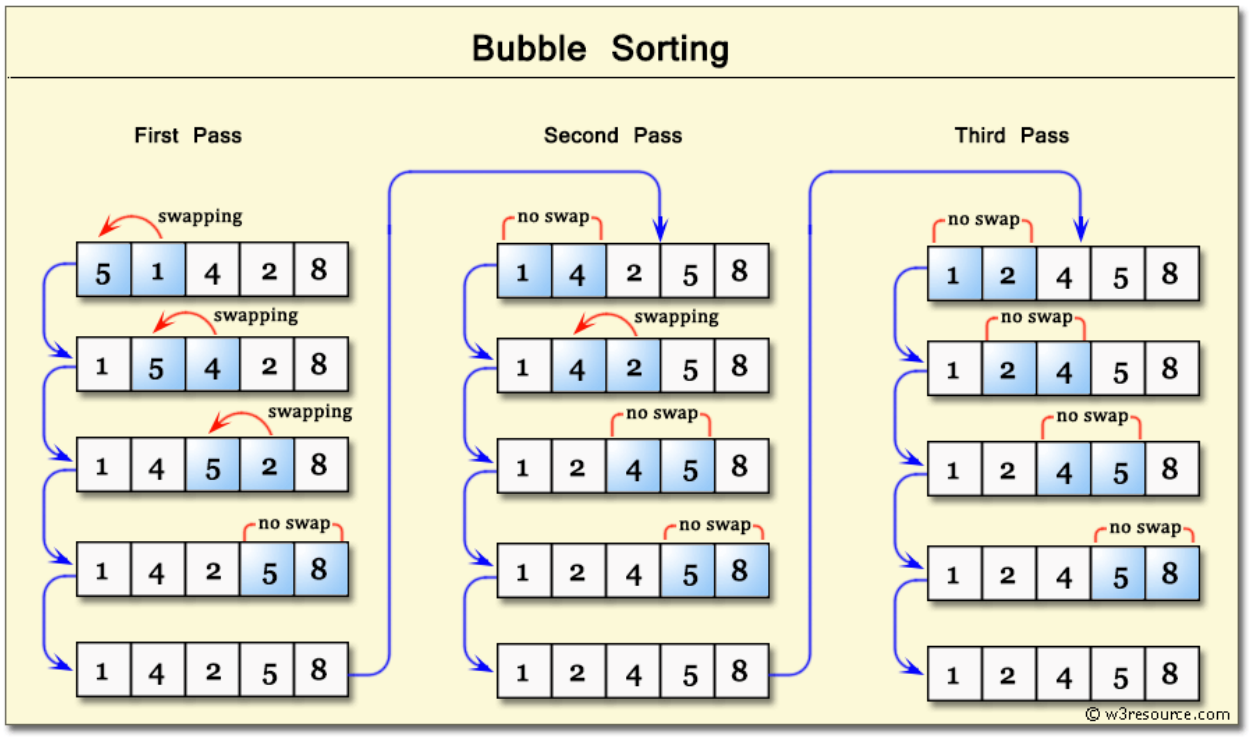


Figura 1: Ejemplo de funcionamiento del algoritmo Bubble Short

3.1.1. Descripción

Una manera simple de expresar el Bubble Sort en pseudocódigo es indicado en la figura 2.

Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso de n términos numerados del 0 al n-1; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta n-i, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja son dos términos de la lista seguidos, j y j+1, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Esta comparación se repite en el centro de los dos bucles, dando lugar a la postre a una lista ordenada. Puede verse que el número de repeticiones solo depende de n y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Esta es una característica de este algoritmo. Luego veremos una variante que evita este inconveniente.

```
procedimiento DeLaBurbuja (a_0, a_1, a_2, ..., a_(n-1))
    para i ← 2 hasta n hacer
        para j ← 0 hasta n − i hacer
            si a_(j) > a_(j+1) entonces
                aux ← a_(j)
                a_(j) ← a_(j+1)
                a_(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

Figura 2: Pseudocódigo del algoritmo Bubble Short

Para comprender el funcionamiento, veamos un ejemplo sencillo:

Tenemos una lista de números que hay que ordenar:

$a = \{55, 86, 48, 16, 82\}$

Podemos ver que la lista que tiene cinco términos, luego:

$n = 5$

El índice i hará un recorrido de 2 hasta n:

para $i \leftarrow 2$ *hasta* n *hacer*

que en este caso será de 2 a 5. Para cada uno de los valores de i, j tomará sucesivamente los valores de 0 hasta n-i:

para $j \leftarrow 0$ *hasta* $n - i$ *hacer*

Para cada valor de j, obtenido en ese orden, se compara el valor del índice j con el siguiente:

si $a_{(j)} > a_{(j+1)}$ *entonces*

Si el término j es mayor que el término j+1, los valores se permutan, en caso contrario se continúa con la iteración.

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	
a_4	82	82	82	$\rightarrow 82$	86
a_3	16	16	$\rightarrow 16$	$\rightarrow 86$	82
a_2	48	$\rightarrow 48$	$\rightarrow 86$	16	16
a_1	$\rightarrow 86$	$\rightarrow 86$	48	48	48
a_0	$\rightarrow 55$	55	55	55	55

Para el caso del ejemplo, tenemos que:

$n = 5$

Para la primera iteración del primer bucle:

$i = 2$

y j tomará los valores de 0 hasta 3:

para $j \leftarrow 0$ *hasta* 3 *hacer*

Cuando j vale 0, se comparan a_0 a_1, el 55 y el 86, dado que 55 ¡86, no se permuta el orden.

Ahora j vale 1 y se comparan a_1 a_2 el 86 y el 48. Como 86 ¿48, se permutan, dando lugar a una nueva lista.

Se repite el proceso hasta que j valga 3, dando lugar a una lista parcialmente ordenada. Podemos ver que el término de mayor valor está en el lugar más alto.

Ahora i vale 3, y j hará un recorrido de 0 a 2.

	$j=0$	$j=1$	$j=2$	
a_4	86	86	86	86
a_3	82	82	$\rightarrow 82$	82
a_2	16	$\rightarrow 16$	$\rightarrow 55$	55
a_1	$\rightarrow 48$	$\rightarrow 55$	16	16
a_0	$\rightarrow 55$	48	48	48

Primero j vale 0, se comparan a_0 a_1, el 55 y el 48. Como 55 ¿48 se permutan dando lugar a la nueva lista.

Para j = 1 se compara el 55 con el 16 y se cambian de orden.

Para j = 2 se compara el 55 y el 82 y se dejan como están, finalizando el bucle con una lista mejor ordenada. Puede verse que los dos valores más altos ya ocupan su lugar. No se ha realizado ninguna comparación con el término cuarto, dado que ya se sabe que después del primer ciclo es el mayor de la lista.

El algoritmo consiste en comparaciones sucesivas de dos términos consecutivos ascendiendo de abajo a arriba en cada iteración, como la ascensión de las burbujas de aire en el agua, de ahí el nombre del procedimiento. En la primera iteración el recorrido ha sido completo, en el segundo se ha dejado él último término, al tener ya el mayor de los valores, en los sucesivos sé ira dejando de realizar las últimas comparaciones, como se puede ver.

	$j=0$	$j=1$	
a_4	86	86	86
a_3	82	82	82
a_2	55	$\rightarrow 55$	55
a_1	$\rightarrow 16$	$\rightarrow 48$	48
a_0	$\rightarrow 48$	16	16

Ahora ya i vale 4 y j recorrerá los valores de 0 a 1.

Cuando j vale 0, se comparan a_0 a_1, esto es, el 48 y el 16. Dado que 48 es mayor que 16 se permutan los valores, dando lugar a una lista algo más ordenada que la anterior. Desde esta nueva ordenación, j pasa a valer 1, con lo que se comparan los términos a 1 a 2 a_1 a_2 el 48 y el 55 que quedan en el mismo orden.

En este caso la burbuja ha ascendido menos que en los casos anteriores, y la lista está ya ordenada, pero el algoritmo tendrá que completarse, realizando una última iteración.

Hay que tener en cuenta que el bucle realiza un número fijo de repeticiones y para finalizar tendrán que completarse, aun en el caso extremo, de que la lista estuviera previamente ordenada.

Por último i vale 5 y j solo puede vale 0, con lo que sólo se realizará una comparación de a_0 a_1 el 16 y el 48, que ya están ordenados y se dejan igual.

	$j=0$	
a_4	86	86
a_3	82	82
a_2	55	55
a_1	$\rightarrow 48$	48
a_0	$\rightarrow 16$	16

Los bucles finalizan y también el procedimiento, dejando la lista ordenada.

Una variante que finaliza en caso de que la lista esté ordenada, puede ser la siguiente: como en el ejemplo anterior, empleando un centinela ordenado, que detecta que no se ha modificado la lista en un recorrido de la burbuja, y que por tanto la lista ya está ordenada, finalizando inmediatamente.

3.1.2. Análisis

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

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

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

$$\Theta(c(n)) = n^2$$

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

El número de intercambios i(n), que hay que realizar depende del orden de los términos y podemos diferenciar, el caso mejor, si el vector está previamente ordenado, y el caso peor, si el vector está ordenado en orden inverso:

$$\Theta(i(n)) = ?$$

Por lo que no se puede determinar una cota ajustada asintótica del número de intercambios, dado que éste dependerá del orden del vector en cuestión.

Facilmente se puede demostrar que tanto el rendimiento en el caso mas desfavorable como en el óptimo es el mismo ya que se ve obligado a recorrer todo el vector (aunque el orden de los intercambios será constante).

3.1.3. Estudio experimental

Verificación empírica del análisis, se usará el codigo de ejemplo de la siguiente sección para analizar los tiempos en funcion del tamaño de entrada (eliminando toda funcionalidad de escritura en pantalla para agilizar el proceso).

Haciendo uso de una entrada de numeros aleatorios se prueban los casos de 10, 100, 1000, 10000, 20000, 40000, 60000, 80000, 100000, 200000 y 300000 elementos. Tras medir los tiempos se deduce experimentalmente (se aprecia en la figura 3) que el orden de este algoritmo en tiempo es $O(n^2)$.

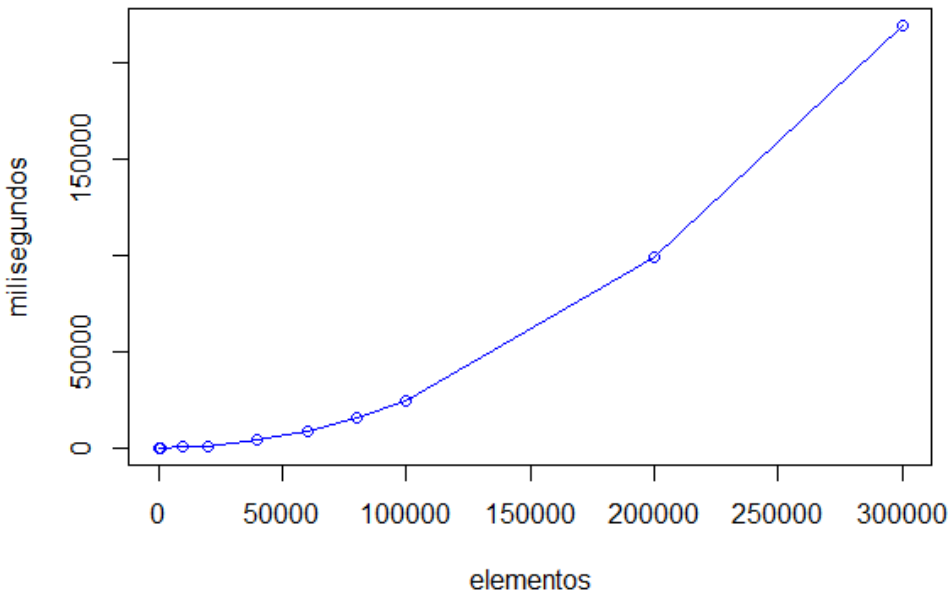


Figura 3: Tiempos en función del número de elementos usando el algoritmo Bubble Short

3.1.4. Ejemplo en C++

– bubbleSortCode.cpp –

```
#include<iostream>
using namespace std;
int main() {
   int num, aux;
   int comparaciones = 0;
   int intercambios = 0;
   int* arreglo;
   cout << "Cuantos numeros seran: ";
   cin >> num;
   arreglo = new int[num];
   cout << endl << "***CAPTURA DE NUMEROS***" << endl;

   for(int x = 0; x < num; x++) {
      cout << "Ingresa el numero " << x << " de la serie: ";
      cin >> arreglo[x];
      cout << endl;
   }
   cout << "***MUESTRA DE NUMEROS***" << endl;

   for(int y = 0; y < num; y++) {
      cout << "Numero " << y << ".- " << arreglo[y] << endl;
   }

   for(int z = 1; z < num; ++z) {
      for(int v = 0; v < (num - z); ++v) {
         comparaciones++;
         if(arreglo[v] > arreglo[v+1]){
            aux = arreglo[v];
            arreglo[v] = arreglo[v + 1];
            arreglo[v + 1] = aux;
            intercambios++;
         }
      }
    }

   cout << "***NUMEROS ARREGLADOS***" << endl;

   for(int w = 0; w < num; w++) {
      cout << "Numero " << w << ".- " << arreglo[w] << endl;
   }

   cout << "Numero de comparaciones: " << comparaciones << endl;
   cout << "Numero de intercambios: " << intercambios << endl;

   delete[] arreglo;
   return 0;
}
```

3.2. Ordenamiento por selección

El ordenamiento por selección (Selection Sort en inglés) es un algoritmo de ordenamiento que requiere (n^2) operaciones para ordenar una lista de n elementos. La gran ventaja de este algoritmo es su simplicidad. [2]

3.2.1. Descripción del algoritmo

Este algoritmo funciona de forma muy intuitiva, en esencia hace lo siguiente:

- Buscar el mínimo elemento de la lista
- Intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista
- Intercambiarlo con el segundo

Y en general:

- Buscar el mínimo elemento entre una posición i y el final de la lista
- Intercambiar el mínimo con el elemento de la posición i

De esta manera se puede escribir el siguiente pseudocódigo para ordenar una lista de n elementos indexados desde el 1:

```
para i=0 hasta n-1
        mínimo = i;
        para j=i+1 hasta n
                si lista[j] ¡lista[mínimo] entonces
                        mínimo = j /* (!) */
                fin si
        fin para
        intercambiar(lista[i], lista[mínimo])
fin para
```

Un ejemplo gráfico de este algoritmo se muestra en la figura 4.

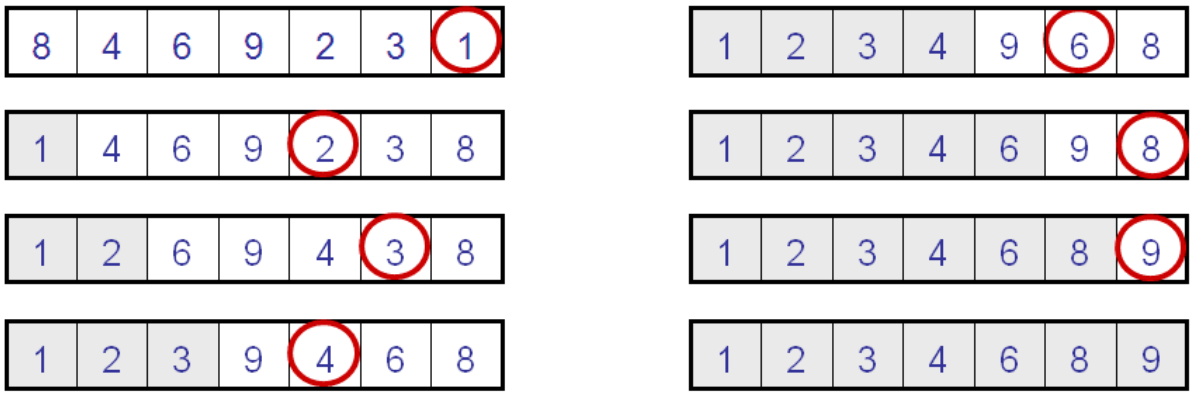


Figura 4: Ejemplo de funcionamiento del algoritmo de selección

3.2.2. Análisis

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

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

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

$$\Theta(c(n)) = n^2$$

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

El número de intercambios i(n), también es fijo, téngase en cuenta que la instrucción: **intercambiar(lista[i], lista[mínimo])** siempre se ejecuta, aun cuando i= mínimo, lo que da lugar:

$$i(n) = n$$

Sea cual sea el vector, y el orden de sus términos, lo que implica en todos los casos un coste lineal:

$$\Theta(i(n)) = n$$

La cota ajustada asintótica del número de intercambios es lineal, del orden de n.

Asimismo, la fórmula que representa el rendimiento del algoritmo, viene dada por la función:

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

3.2.3. Estudio experimental

Verificación empírica del análisis, se usará el codigo de ejemplo de la siguiente sección para analizar los tiempos en funcion del tamaño de entrada (eliminando toda funcionalidad de escritura en pantalla para agilizar el proceso y usando los tamaños del algoritmo anterior).

Tras medir los tiempos se deduce experimentalmente (se aprecia en la figura 5) que el orden de este algoritmo en tiempo es $O(n^2)$, aunque es ligueramente más óptimo que el algoritmo anterior ya que aun siendo un crecimiento exponencial es menor en escala (se dice que su orden exacto es menor, esto se puede apreciar en el apartado de análisis anterior comparándolo con el del algoritmo Bubble Sort).

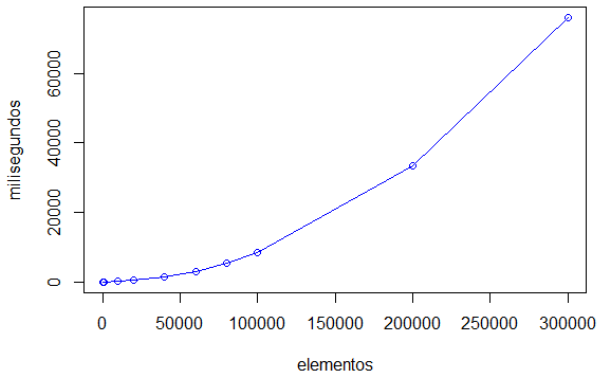


Figura 5: Tiempos en función del número de elementos usando el algoritmo SelectionSort

3.2.4. Ejemplo en C++

– selectionSort.cpp –

```
#include<iostream>

using namespace std;

int main() {
   int num, aux;
   int* arreglo;
   cout << "Cuantos numeros seran: ";
   cin >> num;
   arreglo = new int[num];

   for(int x = 0; x < num; x++) {
      cout << "Ingresa el numero " << x << " de la serie: ";
      cin >> arreglo[x];
      cout << endl;
   }

        int pos_min;
        for (int i=0; i < num-1; i++) {
            pos_min = i;
                // Busca el minumo
                for (int j=i+1; j < num; j++) {
                        if (arreglo[j] < arreglo[pos_min])
                                pos_min=j;
                }
                // Intercambia
                if (pos_min != i) {
                        aux = arreglo[i];
                        arreglo[i] = arreglo[pos_min];
                        arreglo[pos_min] = aux;
                }
        }

   cout << "***NUMEROS ARREGLADOS***" << endl;

   for(int w = 0; w < num; w++) {
      cout << "Numero " << w << ".- " << arreglo[w] << endl;
   }


   delete[] arreglo;
   return 0;
}
```

3.3. Ordenamiento Quicksort

El ordenamiento rápido (quicksort en inglés) es un algoritmo creado por el científico británico en computación C. A. R. Hoare. El algoritmo trabaja de la siguiente forma (se aprecia en la figura 6):

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

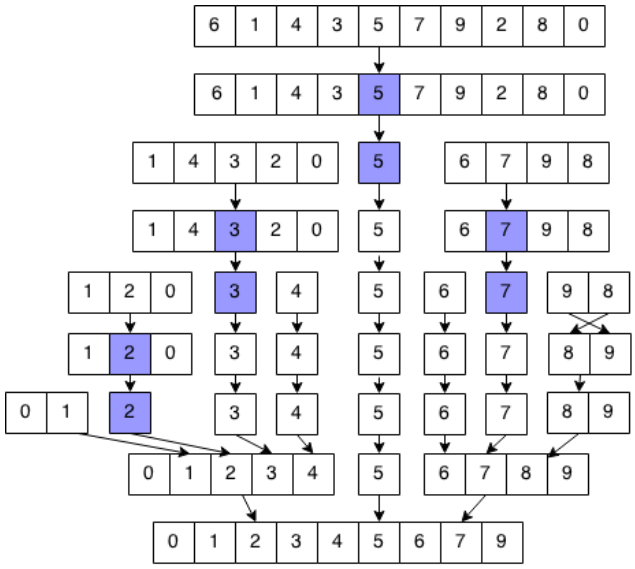


Figura 6: Ejemplo de funcionamiento del algoritmo Quick Sort

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(nlogn)$.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.
- En el caso promedio, el orden es $O(nlogn)$.

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

3.3.1. Demostración de un caso particular

Supongamos que el número de elementos a ordenar es una potencia de dos, es decir, $n = 2^k$ para algún natural k. Inmediatamente $k = log_2(n)$, donde k es el número de divisiones que realizará el algoritmo.

En la primera fase del algoritmo habrá n comparaciones. En la segunda fase el algoritmo instanciará dos sublistas de tamaño aproximadamente n/2. El número total de comparaciones de estas dos sublistas es: 2(n/2) = n. En la tercera fase el algoritmo procesará 4 sublistas más, por tanto el número total de comparaciones en esta fase es 4(n/4) = n.

En conclusión, el número total de comparaciones que hace el algoritmo es:

$n + n + n + ... + n = kn$, donde $k = log_2(n)$, por tanto el Orden de Complejidad del algoritmo en el mejor de los casos es $O(nlog_2(n))$.

3.3.2. Técnicas de elección del pivote

El algoritmo básico del método Quicksort consiste en tomar cualquier elemento de la lista al cual denominaremos como pivote, dependiendo de la partición en que se elija, el algoritmo será más o menos eficiente.

- Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección «a ciegas» siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en $O(n)$ y asegura que hasta en el peor de los casos, el algoritmo sea $O(nlog(n))$. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

3.3.3. Técnicas de reposicionamiento

Una idea preliminar para ubicar el pivote en su posición final sería contar la cantidad de elementos menores que él, y colocarlo un lugar más arriba, moviendo luego todos esos elementos menores que él a su izquierda, para que pueda aplicarse la recursividad.

Existe, no obstante, un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos índice izquierdo, y j, al que llamaremos índice derecho. El algoritmo es el siguiente:

- Recorrer la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
- Cuando lista[i] sea mayor que el pivote y lista[j] sea menor, se intercambian los elementos en esas posiciones.
- Repetir esto hasta que se crucen los índices.
- El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

3.3.4. Estudio experimental

Verificación empírica del análisis, se usará el codigo de ejemplo de la siguiente sección para analizar los tiempos en funcion del tamaño de entrada (eliminando toda funcionalidad de escritura en pantalla para agilizar el proceso).

Haciendo uso de una entrada de numeros aleatorios se prueban los casos de 10, 100, 1000, 10000, 20000, 40000, 60000, 80000, 100000, 200000 y 300000 elementos. Tras medir los tiempos se deduce experimentalmente (se aprecia en la figura 7) que el orden de este algoritmo en tiempo es casi $O(n)$ en este tamaño de problemas, para tamaño mas grandes se apreciara como la grafica tiende a un orden $O(nlog(n))$.

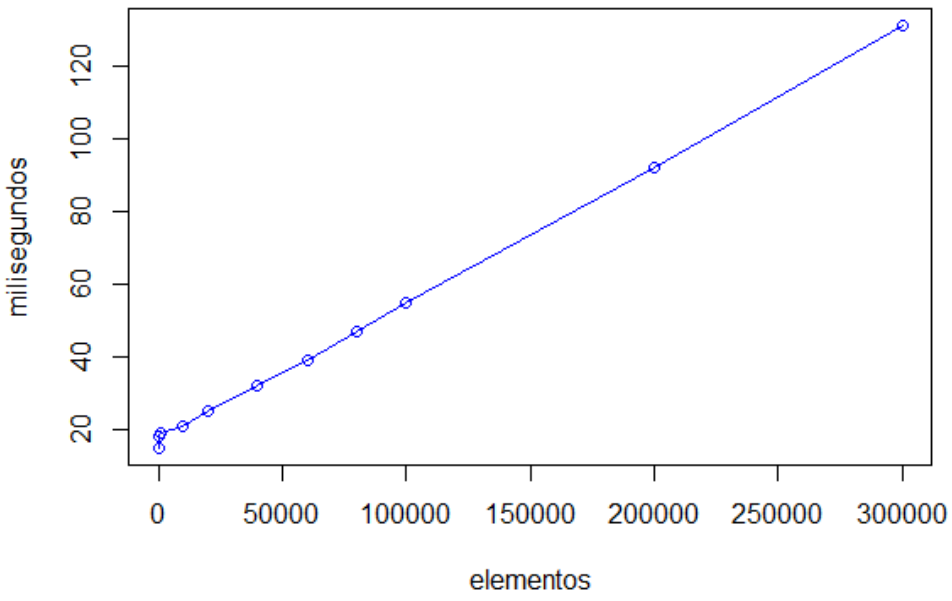


Figura 7: Tiempos en función del número de elementos usando el algoritmo QuickSort

3.3.5. Ejemplo en C++

– quicksort.cpp –

```
#include<iostream>
using namespace std;
// Funcion para dividir el array y hacer los intercambios
int divide(int *array, int start, int end) {
    int left, right, pivot, temp;
    pivot = array[start];
    left = start;
    right = end;
    // Mientras no se cruzen los indices
    while (left < right) {
        while (array[right] > pivot)
            right--;
        while ((left < right) && (array[left] <= pivot))
            left++;
        // Si todavia no se cruzan los indices seguimos intercambiando
        if (left < right) {
            temp = array[left];
            array[left] = array[right];
            array[right] = temp;
        }
    }
    // Los indices ya se han cruzado, ponemos el pivot en el lugar que le corresponde
    temp = array[right];
    array[right] = array[start];
    array[start] = temp;
    // La nueva óposicin del pivot
    return right;
}
// Funcion recursiva para hacer el ordenamiento
void quicksort(int *array, int start, int end) {
    int pivot;
    if (start < end) {
        pivot = divide(array, start, end);
        // Ordeno la lista de los menores
        quicksort(array, start, pivot - 1);
        // Ordeno la lista de los mayores
        quicksort(array, pivot + 1, end);
    }
}
int main() {
   int num, aux;
   int* arreglo;
   cin >> num;
   arreglo = new int[num];
   for(int x = 0; x < num; x++)
      cin >> arreglo[x];
   quicksort(arreglo, 0, num - 1);
   for(int w = 0; w < num; w++)
      cout << "Numero " << w << ".- " << arreglo[w] << endl;
   delete[] arreglo;
   return 0;
}
```

4. Conclusiones

Se aprecia como el mejor algoritmo de ordenacion es QuickSort, aunque este tiene la limitación de no tener un orden exacto, el promedio como se indica anteriormente es $O(nlog(n))$ pero en el peor caso asciende a un $O(n^2)$.

Los dos primeros algoritmos, aunque tienen un orden $O(n^2)$ son más sencillos de explicar para un público no experto en algoritmia. Además el algoritmo QuickSort normalmente viene ya implementado en la mayoría de librerías estándar de muchos lenguajes de programación por lo que su uso suele ser fácil.

Referencias

[1] Wikipedia contributors. Algoritmos de ordenamiento — Wikipedia, the free encyclopedia, 2004. [Online; accessed 26-Feb-2018].

[2] Wikipedia contributors. Ordenamiento por selección — Wikipedia, the free encyclopedia, 2004. [Online; accessed 26-Feb-2018].

[3] Wikipedia contributors. Ordenamiento de burbuja — Wikipedia, the free encyclopedia, 2005. [Online; accessed 26-Feb-2018].