Estructura de datos III

TP 1:

Algoritmos de ordenamiento

Autor:

Love4Programming@gmail.com

Sede:

San Isidro

Algoritmos de ordenamiento

Resumen

El siguiente trabajo desarrolla el tema de la performance en distintos algoritmos de ordenamiento y presenta una comparativa de cada uno de ellos, como también el órden de complejidad de los mismos.

Los algoritmos analizados son:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Heap sort
- Merge sort
- Quick sort

Las implementaciones de los algoritmos han sido realizadas en c++.

Indice

Introducción.

Uno de los problemas fundamentales en la ciencia de la computación es ordenar una lista de items. Existen una infinidad de métodos de ordenamiento, algunos son simples e intuitivos, como el bubble sort, y otros como son extremadamente complicados, pero producen los resultados mucho más rápido.

En este trabajo se presentan los algoritmos de ordenamiento más comunes, entre los cuales están los siguientes: Bubble sort, Heap sort, Insertion sort, Merge sort, Quick sort, Selection sort y Shell sort.

Los algoritmos de ordenamiento pueden ser divididos en dos clases de acuerdo a la complejidad de los mismos. La complejidad del algoritmo se denota según la notación Big-O.

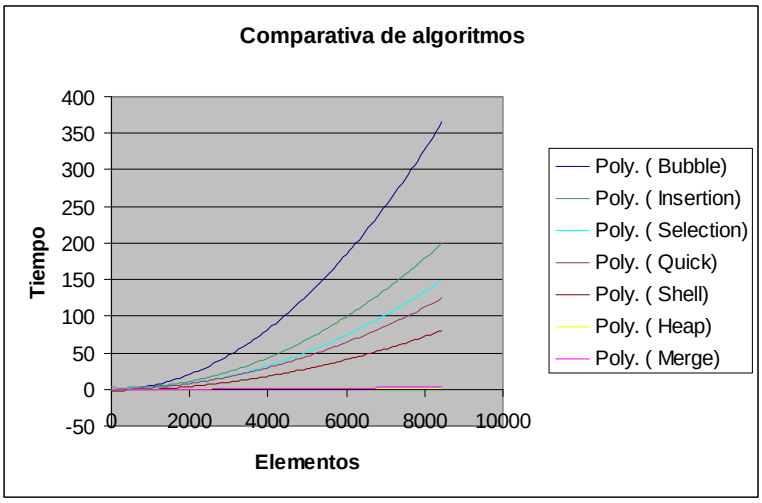
Por ejemplo, O(n) significa que el algoritmo tiene una complejidad lineal. En otras palabras, toma 10 veces más tiempo en operar un set de 100 datos que en hacerlo con un set de 10 items. Si la complejidad fuera $O(n^2)$ entonces tomaría 100 veces más tiempo en operar 100 items que en hacerlo con 10.

Adicionalmente a la compejidad de los algoritmos, la velocidad de ejecución puede cambiar de acuerdo al tipo de dato a ordenar, es por ello que es conveniente comprar los algoritmos contra datos empíricos. Éstos datos empíricos se deben elaborar tomando la media de tiempo de ejecución en un conjunto de corridas y con datos del mismo tipo.

En este trabajo se ejecutaron 10 veces cada algoritmo de ordenamiento con un set de 10000 datos de tipo entero de c++.

Se realiza también una comprativa con otros tipos de datos como ser el long y el char para todos los algoritmos y luego se los compara entre cada uno de ellos.

En el gráfico siguiente se puede ver el tiempo de ejecución del algoritmo en función de la cantidad de elementos, se puede observar que si los elementos a ordenar son pocos, menos de 1000 casi todos los tienen el mismo tiempo de respuesta, pero si la cantidad de datos aumenta los tiempos de respuesta van cambiando drásticamente entre cada uno de los ellos, y para una cantidad de datos de 8000 ya se puede determinar que el peor algoritmo es el bubble sort, mientras que el mejor es el Heap sort.



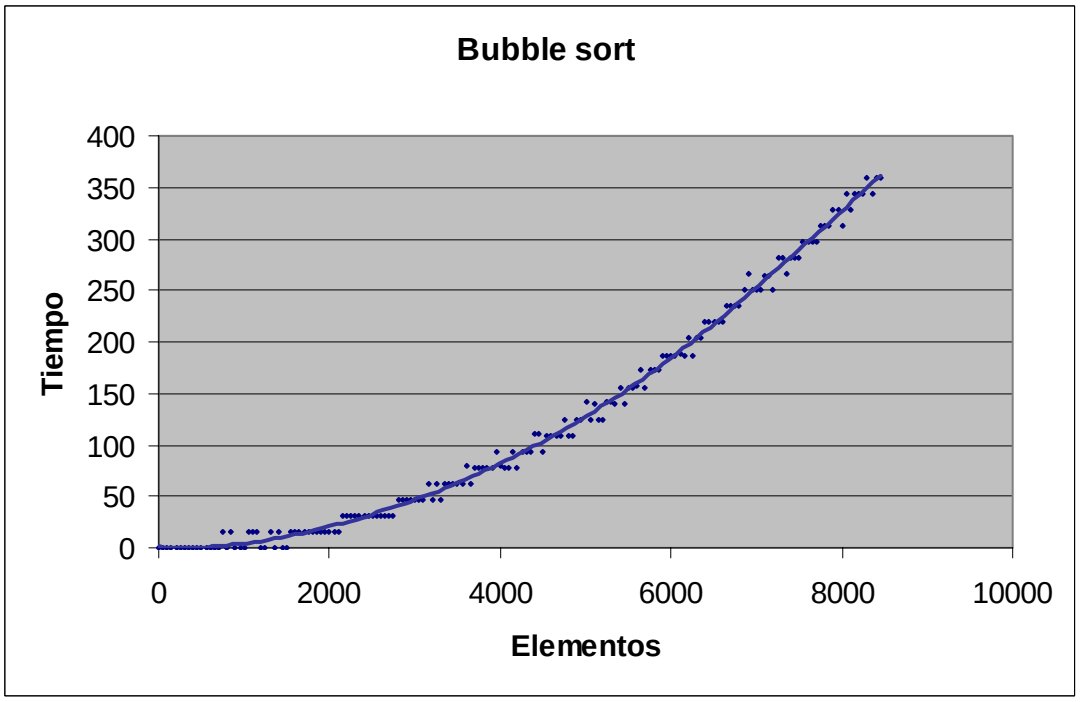
Análisis de los algoritmos

Bubble sort

Este es el método más simple y antiguo para ordenar un conjunto de datos, es también el más lento.

El algoritmo bubble sort tiene dos bucles for internos que recorren el vector comparando el elemento j-esimo-1 con el elemento con el j-esimo elemento y en caso de que este sea mayor hace un cambio de los elementos.

Al tener dos bucles internos el comportamiento es en general $O(n^2)$, y en las mejores condiciones se comporta como $O(n)$.



Como funciona

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

Primera pasada:
{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:
{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.

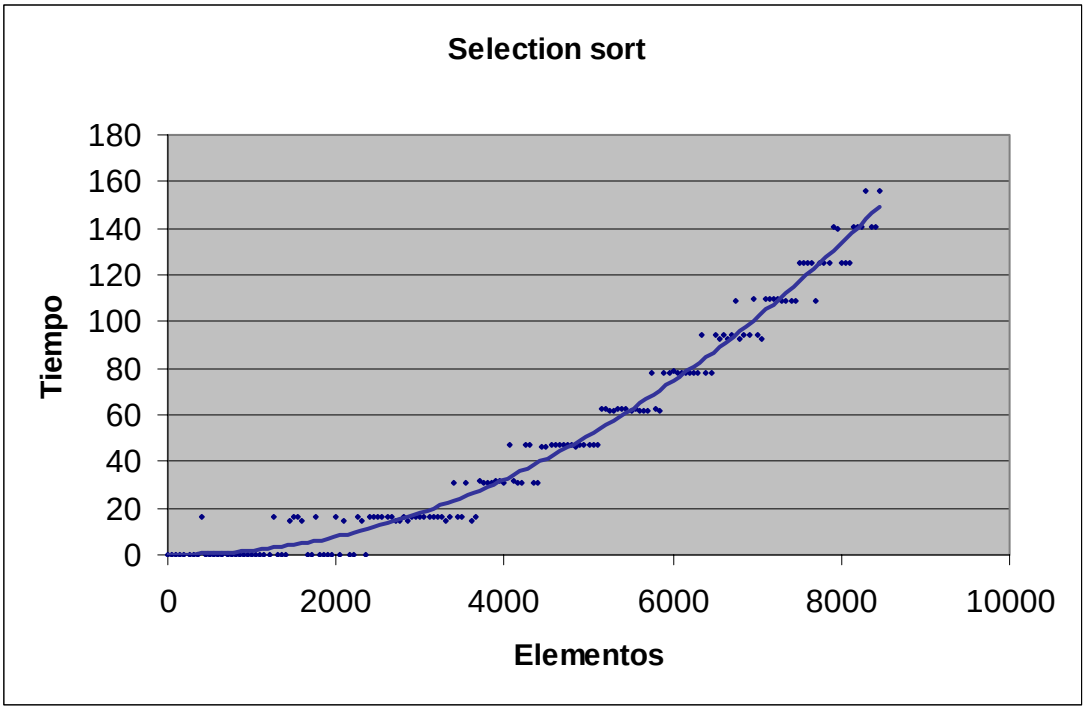
Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Código fuente

```
void bubbleSort (int numbers[], int array_size)
{
  int i, j, temp;
  for (i = (array_size - 1); i >= 0; i--){
    for (j = 1; j <= i; j++){
      if (numbers[j-1] > numbers[j]){
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

Selection sort

Este algoritmo trabaja seleccionando el item más pequeño a ser ordenado que aún esta en la lista, y luego haciendo un intercambio con el elemento en la siguiente posición. La complejidad de este algoritmo es $O(n^2)$.



Como funciona.

Este método consiste en buscar el elemento más pequeño del array y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segudo lugar, y así sucesivamente hasta colocar el último elemento. Por ejemplo, si tenemos el array {40,21,4,9,10,35}, los pasos a seguir son:

```
{4,21,40,9,10,35} <-- Se coloca el 4, el más pequeño, en primera
posición : se cambia el 4 por el 40.
{4,9,40,21,10,35} <-- Se coloca el 9, en segunda posición: se cambia el
9 por el 21.
{4,9,10,21,40,35} <-- Se coloca el 10, en tercera posición: se cambia
el 10 por el 40.
{4,9,10,21,40,35} <-- Se coloca el 21, en tercera posición: ya está
colocado.
{4,9,10,21,35,40} <-- Se coloca el 35, en tercera posición: se cambia
el 35 por el 40.
```

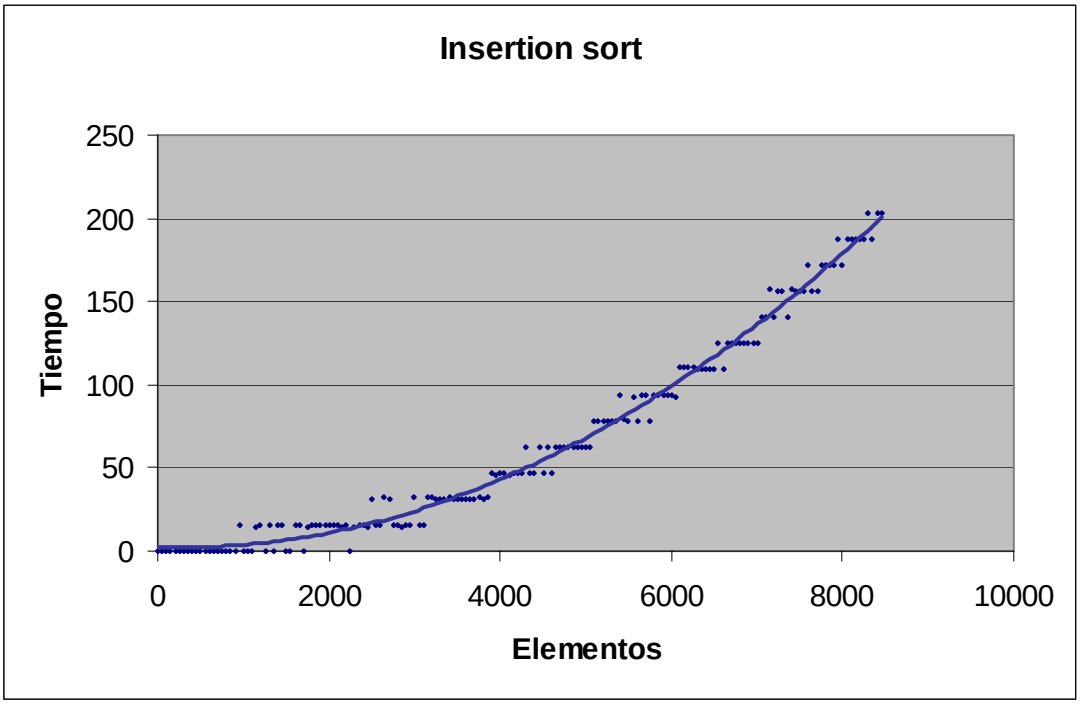
Código fuente.

```
void selectionSort(int numbers[], int array_size)
{
  int i, j;
  int min, temp;

  for (i = 0; i < array_size-1; i++){
    min = i;
    for (j = i+1; j < array_size; j++){
      if (numbers[j] < numbers[min])
        min = j;
```

```
        }
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
      }
    }
```

Insertion sort

El insertion sort trabaja insertando el item en su lugar correspondiente al final de la lista. Como el bubble sort, éste algoritmo se comporta como $O(n^2)$, pero a pesar de tener el misma complejidad, este algoritmo es casi el doble más eficiente que el bubble sort.



Como funciona.

En este método lo que se hace es tener una sublista ordenada de elementos del array e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada. Para el ejemplo {40,21,4,9,10,35}, se tiene:

{40,21,4,9,10,35} <-- La primera sublista ordenada es {40}.

Insertamos el 21:

{40,40,4,9,10,35} <-- aux=21;

{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:

{21,40,40,9,10,35} <-- aux=4;

{21,21,40,9,10,35} <-- aux=4;

{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:

{4,21,40,40,10,35} <-- aux=9;

{4,21,21,40,10,35} <-- aux=9;

{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:

{4,9,21,40,40,35} <-- aux=10;

{4,9,21,21,40,35} <-- aux=10;

{4,9,10,21,40,35} <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:

{4,9,10,21,40,40} <-- aux=35;

{4,9,10,21,35,40} <-- El array está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en O(n2). En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(n).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a O(n2).

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

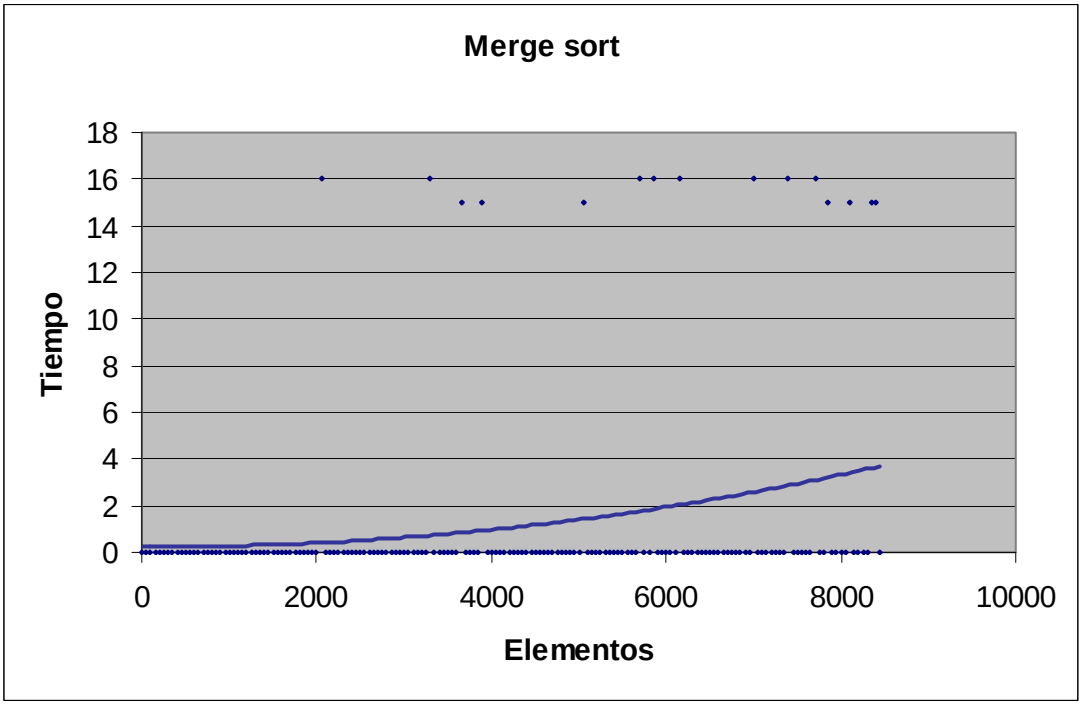
Código fuente.

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i=1; i < array_size; i++){
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))    {
      numbers[j] = numbers[j-1];
      j = j - 1;
    }
    numbers[j] = index;
  }
}
```

Merge sort

El merge sort divide la lista a ser ordenada en dos mitades iguales y las pone en arrays separadas. Cada array es ordenado recursivamente, y luego se juntan en el array final. Este algoritmo tiene un comportamiento de $O(n \log n)$.

Implementaciones elementales del merge sort hacen uso de tres arrays, uno para cada mitad y uno para almacenar el resultado final. El algoritmo aqui presentado mezcla “in situ” los dos arrays para presentar el resultado en uno de ellos, por lo cual solo utiliza dos arrays.



Como funciona.

Consta de dos partes, una parte de intercalación de listas y otra de divide y vencerás.

- Primera parte: ¿Cómo intercalar dos listas ordenadas en una sola lista ordenada de forma eficiente?

Suponemos que se tienen estas dos listas de enteros ordenadas ascendentemente:

lista 1: 1 -> 3 -> 5 -> 6 -> 8 -> 9
lista 2: 0 -> 2 -> 6 -> 7 -> 10

Tras mezclarlas queda:

lista: 0 -> 1 -> 2 -> 3 -> 5 -> 6 -> 6 -> 7 -> 8 -> 9 -> 10

Esto se puede realizar mediante un único recorrido de cada lista, mediante dos punteros que recorren cada una. En el ejemplo anterior se insertan en este orden -salvo los dos 6 que puede variar según la implementación-: 0 (lista 2), el 1 (lista 1), el 2 (lista 2), el 3, 5 y 6 (lista 1), el 6 y 7 (lista 2), el 8 y 9 (lista 1), y por llegar al final de la lista 1, se introduce directamente todo lo que quede de la lista 2, que es el

10.

- Segunda parte: divide y vencerás. Se separa la lista original en dos trozos del mismo tamaño (salvo listas de longitud impar) que se ordenan recursivamente, y una vez ordenados se fusionan obteniendo una lista ordenada. Como todo algoritmo basado en divide y vencerás tiene un caso base y un caso recursivo.

* Caso base: cuando la lista tiene 1 ó 0 elementos (0 se da si se trata de ordenar una lista vacía). Se devuelve la lista tal cual está.

* Caso recursivo: cuando la longitud de la lista es de al menos 2 elementos. Se divide la lista en dos trozos del mismo tamaño que se ordenan recursivamente. Una vez ordenado cada trozo, se fusionan y se devuelve la lista resultante.

El esquema es el siguiente:

```
Ordenar(lista L)
inicio
  si tamaño de L es 1 o 0 entonces
    devolver L
  si tamaño de L es >= 2 entonces
    separar L en dos trozos: L1 y L2.
    L1 = Ordenar(L1)
    L2 = Ordenar(L2)
    L = Fusionar(L1, L2)
    devolver L
fin
```

El algoritmo funciona y termina porque llega un momento en el que se obtienen listas de 2 ó 3 elementos que se dividen en dos listas de un elemento (1+1=2) y en dos listas de uno y dos elementos (1+2=3, la lista de 2 elementos se volverá a dividir), respectivamente. Por tanto se vuelve siempre de la recursión con listas ordenadas (pues tienen a lo sumo un elemento) que hacen que el algoritmo de fusión reciba siempre listas ordenadas.

Se incluye un ejemplo explicativo donde cada sublista lleva una etiqueta identificativa.

Dada: 3 -> 2 -> 1 -> 6 -> 9 -> 0 -> 7 -> 4 -> 3 -> 8 (lista original)

se divide en:

3 -> 2 -> 1 -> 6 -> 9 (lista 1)

0 -> 7 -> 4 -> 3 -> 8 (lista 2)

se ordena recursivamente cada lista:

3 -> 2 -> 1 -> 6 -> 9 (lista 1)

se divide en:

3 -> 2 -> 1 (lista 11)

6 -> 9 (lista 12)

se ordena recursivamente cada lista:

3 -> 2 -> 1 (lista 11)

se divide en:

3 -> 2 (lista 111)

1 (lista 112)

se ordena recursivamente cada lista:

3 -> 2 (lista 111)

se divide en:

3 (lista 1111, que no se divide, caso base). Se devuelve 3

2 (lista 1112, que no se divide, caso base). Se devuelve 2

se fusionan 1111-1112 y queda:

2 -> 3. Se devuelve 2 -> 3

1 (lista 112)

1 (lista 1121, que no se divide, caso base). Se devuelve 1

se fusionan 111-112 y queda:

1 -> 2 -> 3 (lista 11). Se devuelve 1 -> 2 -> 3

6 -> 9 (lista 12)

se divide en:

6 (lista 121, que no se divide, caso base). Se devuelve 6

9 (lista 122, que no se divide, caso base). Se devuelve 9

se fusionan 121-122 y queda:

6 -> 9 (lista 12). Se devuelve 6 -> 9

se fusionan 11-12 y queda:

1 -> 2 -> 3 -> 6 -> 9. Se devuelve 1 -> 2 -> 3 -> 6 -> 9

0 -> 7 -> 4 -> 3 -> 8 (lista 2)

tras repetir el mismo procedimiento se devuelve 0 -> 3 -> 4 -> 7 -> 8

se fusionan 1-2 y queda:

· 0 -> 1 -> 2 -> 3 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9, que se devuelve y se termina.

- Segunda parte: divide y vencerás. Se separa la lista original en dos trozos del mismo tamaño (salvo listas de longitud impar) que se ordenan recursivamente, y una vez ordenados se fusionan obteniendo una lista ordenada. Como todo algoritmo basado en divide y vencerás tiene un caso base y un caso recursivo.

* Caso base: cuando la lista tiene 1 ó 0 elementos (0 se da si se trata de ordenar una lista vacía). Se devuelve la lista tal cual está.

* Caso recursivo: cuando la longitud de la lista es de al menos 2 elementos. Se divide la lista en dos trozos del mismo tamaño que se ordenan recursivamente. Una vez ordenado cada trozo, se fusionan y se devuelve la lista resultante.

Código fuente.

```
void mergeSort(int numbers[], int temp[], int array_size)
{
  m_sort(numbers, temp, 0, array_size - 1);
}

void m_sort(int numbers[], int temp[], int left, int right)
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, mid+1, right);

    merge(numbers, temp, left, mid+1, right);
  }
}

void merge(int numbers[], int temp[], int left, int mid, int right)
{
  int i, left_end, num_elements, tmp_pos;

  left_end = mid - 1;
  tmp_pos = left;
  num_elements = right - left + 1;

  while ((left <= left_end) && (mid <= right))
  {
    if (numbers[left] <= numbers[mid])
    {
      temp[tmp_pos] = numbers[left];
      tmp_pos = tmp_pos + 1;
      left = left +1;
    }
    else
    {
      temp[tmp_pos] = numbers[mid];
      tmp_pos = tmp_pos + 1;
      mid = mid + 1;
    }
  }

  while (left <= left_end)
  {
    temp[tmp_pos] = numbers[left];
```

```
        left = left + 1;
        tmp_pos = tmp_pos + 1;
      }
      while (mid <= right)
      {
        temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;
      }

      for (i=0; i <= num_elements; i++)
      {
        numbers[right] = temp[right];
        right = right - 1;
      }
    }
```

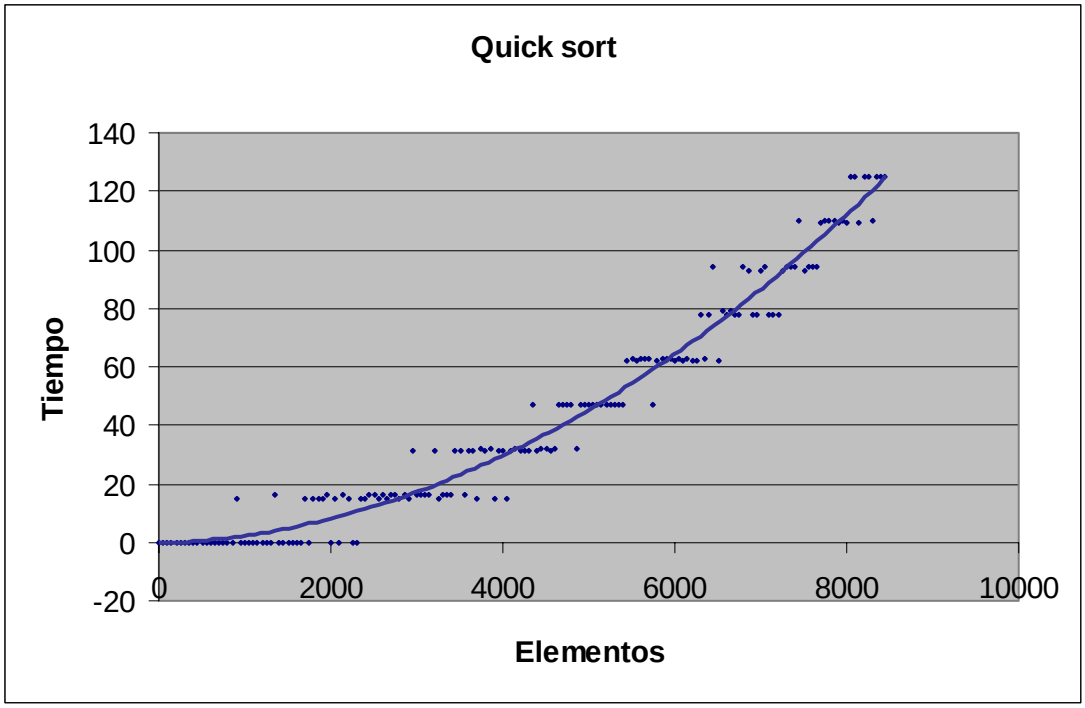
Quick sort

El Quick sort es un algoritmo del estilo divide y venceras. Es bastante más rápido que el merge sort.

El algoritmo de recursion consiste en una serie de cuatro pasos:

1. Si hay menos de un elemento a ser ordenado retorna inmediatamente.
2. Tomar un elemento del vector que sirve como “Pivot”
3. Dividir el array en dos partes, una con los elementos mayores y una con los elementos menores al pivot.
4. Repetir recursivamente el algoritmo para las dos mitades del array original.

La eficiencia de este algoritmo es impactama mayormente por la selección del elemento que sera tomado como pivot. En el peor caso, el comportamiento es de la forma $O(n^2)$, y ocurre cuando la lista esta ordenada. Si el elemento utilizado como pivot se selecciona de forma random, el comportamiento se parece a $O(n \log n)$.



Como funciona.

Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Normalmente se toma como pivote el primer elemento de array, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.Por ejemplo, para dividir el array

{21,40,4,9,10,35}, los pasos serían:

{21,40,4,9,10,35} <-- se toma como pivote el 21. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote.

Se intercambian:{21,10,4,9,40,35} <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:

{9,10,4,21,40,35} <-- Ahora tenemos dividido el array en dos arrays más pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.

Código fuente.

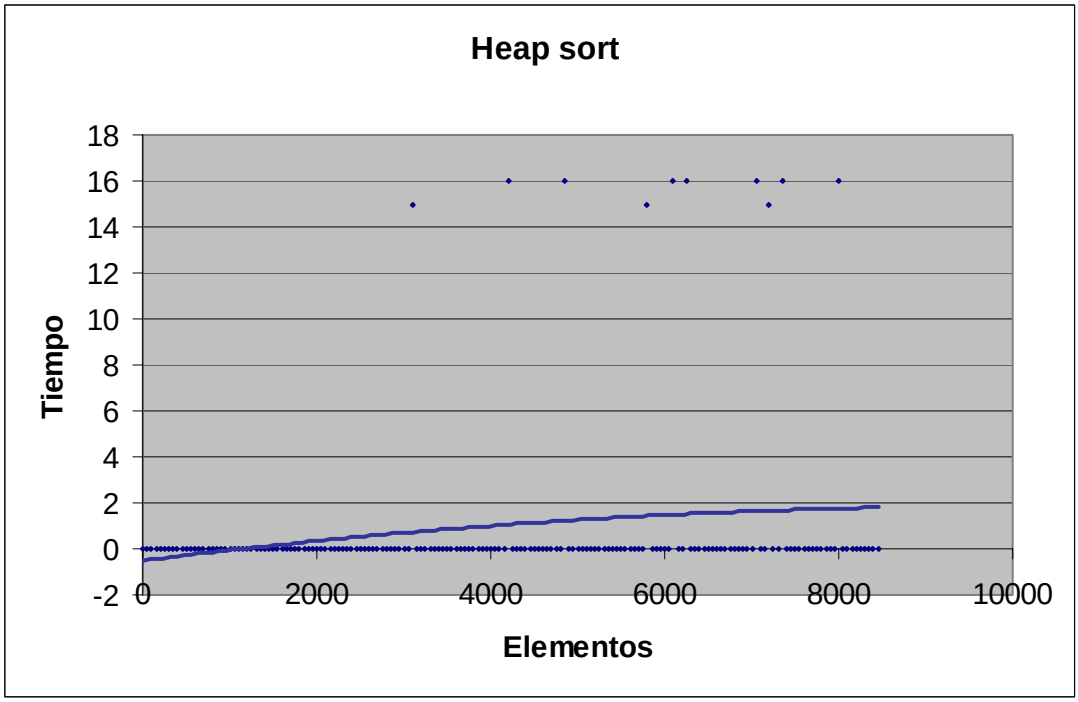
```
void quickSort(int numbers[], int array_size)
```

```
{
  q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

Heap Sort

El heap sort tiene un comportamiento del tipo O(n log n), pero a diferencia de los algoritmos merge y quick sort este no requiere recursión masiva o múltiples arrays para trabajar.

El heap sort comienza construyendo un heap del set de datos, y luego remueve los items más grandes poniendolos al final del array ordenado. Luego de eliminar los items más grandes, reconstruye el heap y remueve los elementos más grandes nuevamente. Esto lo repite hasta que no queden elementos en el heap y el heap de items ordenados este lleno.



Como funciona.

No es propiamente un método de ordenación, consiste en la unión de dos arrays ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arrays de izquierda a derecha e ir tomando el menor de los dos elementos, de forma que sólo aumenta el contador del array del que sale el elemento siguiente para el array-suma. Si quisiéramos sumar los arrays {1,2,4} y {3,5,6}, los pasos serían:

Inicialmente: i1=0, i2=0, is=0.
Primer elemento: mínimo entre 1 y 3 = 1. Suma={1}. i1=1, i2=0, is=1.
Segundo elemento: mínimo entre 2 y 3 = 2. Suma={1,2}. i1=2, i2=0, is=2.
Tercer elemento: mínimo entre 4 y 3 = 3. Suma={1,2,3}. i1=2, i2=1, is=3.
Cuarto elemento: mínimo entre 4 y 5 = 4. Suma={1,2,3,4}. i1=3, i2=1, is=4.
Como no quedan elementos del primer array, basta con poner los elementos que quedan del segundo array en la suma:

Código fuente.

```
void heapSort(int numbers[], int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
```

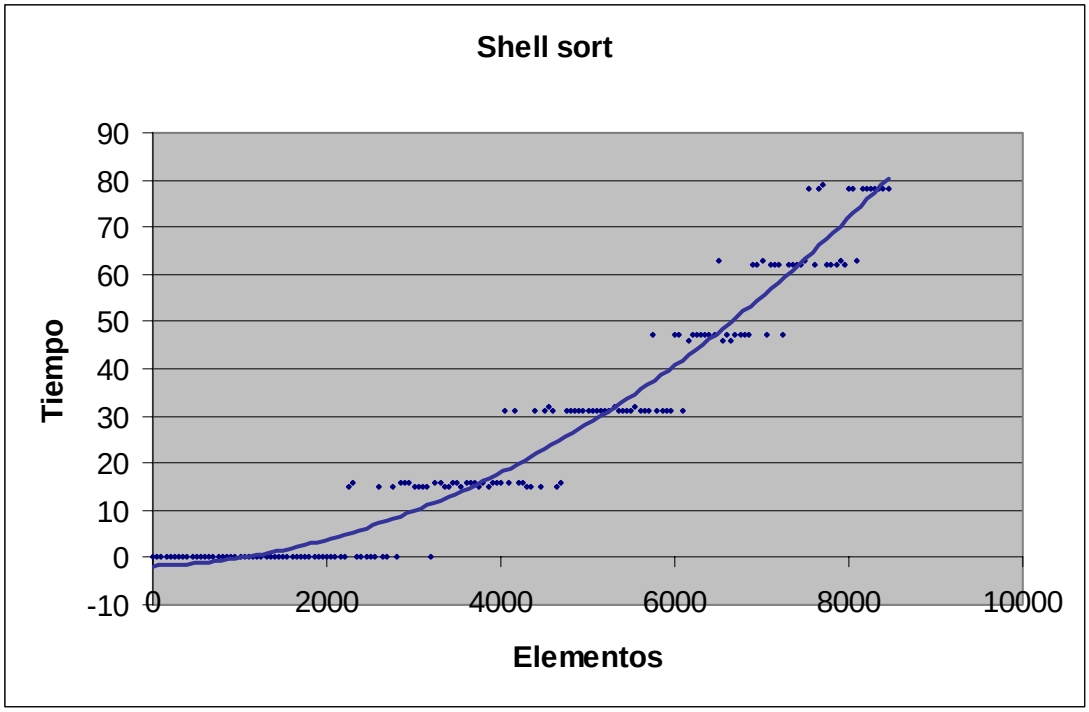
```
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}

void siftDown(int numbers[], int root, int bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}
```

Shell sort

Inventado en 1959, éste algoritmo es el más eficiente de los del tipo $O(n^2)$. Pero el Shell es también el más complicado de los algoritmos de este tipo.

El algoritmo realiza multiples pases a través de la lista, y en cada pasada ordena un numero igual de items. El tamaño del set de datos (también llamado distancia o intervalo) a ser ordenado va creciendo a medida que el algoritmo recorre el array hasta que finalmente el set esta compuesto por todo el array en si mismo.

El tamaño del set de datos usado tiene un impacto significativo en la eficiencia del algoritmo. Algunas implementaciones de este algoritmo tienen una función que permite calcular el tamaño óptimo del set de datos para un array determinado.



Como funciona

Es una mejora del método de inserción directa, utilizado cuando el array tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el array {40,21,4,9,10,35} mediante el método de Shell serían:

Salto=3:
  Primera pasada:
  {9,21,4,40,10,35} <-- se intercambian el 40 y el 9.
  {9,10,4,40,21,35} <-- se intercambian el 21 y el 10.
Salto=1:
  Primera pasada:
  {9,4,10,40,21,35} <-- se intercambian el 10 y el 4.
  {9,4,10,21,40,35} <-- se intercambian el 40 y el 21.
  {9,4,10,21,35,40} <-- se intercambian el 35 y el 40.
  Segunda pasada:
  {4,9,10,21,35,40} <-- se intercambian el 4 y el 9.

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

Código fuente

```
void shellSort(int numbers[], int array_size)
        {
         int i, j, increment, temp;

         increment = 3;
         while (increment > 0)
         {
           for (i=0; i < array_size; i++)
           {
             j = i;
             temp = numbers[i];
             while ((j >= increment) && (numbers[j-increment] > temp))
             {
               numbers[j] = numbers[j - increment];
               j = j - increment;
             }
             numbers[j] = temp;
           }
           if (increment/2 != 0)
             increment = increment/2;
           else if (increment == 1)
             increment = 0;
           else
             increment = 1;
         }
        }
```

Conclusiones

Los algoritmos comunes de ordenamiento pueden dividirse en dos clases, según su orden de complejidad. Por un lado están los **algoritmos de complejidad cuadrática 0(n^2),** entre los cuales se incluyen los de burbujeo, de inserción, de selección y el denominado shellsort, y, por otro lado están los de **complejidad 0(n * log(n)),** entre los cuales se incluyen los algoritmos heapsort, mergesort y quicksort, De más está decir que los algoritmos pertenecientes al segundo grupo son en general más veloces que los pertenecientes al primero.

Más allá de su complejidad algorítmica, la eficiencia de los distintos algoritmos de ordenamiento puede compararse utilizando datos empíricos, dado que la velocidad de u proceso de ordenamiento varía enormemente según las características del conjunto de datos a ordena, para obtener resultados empíricos precisos se debe realizar un gran número de ejecuciones de cada algoritmo sobre conjuntos de datos aleatorios, y luego, promediar los tiempos de ejecución para obtener una idea fiel del rendimiento.

Por otro lado no todos los algoritmos se comportan igual ante conjuntos de datos con características particulares. Por ejemplo, si se requiere un algoritmo para mantener el orden en una lista “casi” ordenada –es decir, una lista en la que relativamente pocos elementos se encuentran desordenados-, el algoritmo de burbujeo pude estar entre los mas eficientes, aunque es más complejo. Distinta es la situación cuando la lista a ordenar aparece con la mayoría de sus elementos desordenados. A su vez, el orden de complejidad de los algoritmos no refleja su eficiencia a la hora de ordenar conjuntos de pocos elementos. Por eso es que el orden de complejidad de un algoritmo refleja su eficiencia en un sentido asintótico.

Apéndice

Apéndice 1 - Código fuente del Trabajo Práctico

```
#include "stdio.h"
#include "conio.h"
#include "time.h"
#include "stdlib.h"
#include "iostream.h"
#include "fstream.h"
#include "string.h"

#define elements 10000

//Algoritmos de ordenamiento implementados
void bubbleSort(int *, int);
void heapSort(int *, int);
void siftDown(int *, int , int );
void mergeSort(int *, int *, int );
void m_sort(int *, int *, int , int );
void merge(int *, int *, int , int , int );
void selectionSort(int *, int);
void shellSort(int *, int);
void insertionSort(int *, int);
void quickSort(int *, int);
void q_sort(int *, int, int);

//Funciones adicionales desarrolladas
void genNumbers (int *, int);
void genNumbersInv (int *, int);
void genNumbersSor (int *, int);
void cpyNumbers (int *, int *, int);
void printNumbers (int *, int);

int main(int argc, char* argv[])
{
        int numbers[elements], aux_numbers[elements], corrida=1, maxcorrida;
        int temp[elements], option_algorithm;
        int option, actual_element=1, step=50;
        char fileref[5],filename[9];

        clock_t tBubble1, tBubble2, tHeap1, tHeap2, tInsertion1, tInsertion2;
        clock_t tMerge1, tMerge2, tQuick1, tQuick2, tSelection1, tSelection2;
        clock_t tShell1, tShell2;

        cout << "Demo - Sorting algorithms" << endl;
        cout << "Number of elements: "<< elements << endl;
        cout << "Step 1 - Select the sort algorithms to run" << endl;
        cout << "        1) Bubble sort" << endl;
        cout << "        2) Heap sort" << endl;
        cout << "        3) Insertion sort" << endl;
        cout << "        4) Merge sort" << endl;
```

```
        cout << "       5) Quick sort" << endl;
        cout << "       6) Selection sort" << endl;
        cout << "       7) Shell sort" << endl;
        cout << "       8) Bubble, Insertion and Selection sort" <<endl;
        cout << "       9) Heap, Merge, Quick and Sbell sort" << endl;
        cout << "       10) All the sort algothims" << endl;
        cout << " your option?: ";
        cin >> option_algorithm;

        cout << "Step 2 - Select vector generator" << endl;
        cout << "       1) Random" << endl;
        cout << "       2) Sorted" << endl;
        cout << "       3) Worst" << endl;
        cout << " your option?: ";
        cin >> option;

        cout << "Step 3 - Select the number of times to execute" << endl;
        cout << " your option?: ";
        cin >> maxcorrida;

        cout << "Step 4 - Procesing" << endl;
        while (corrida <= maxcorrida){
                itoa(corrida,fileref,10);
                strcpy (filename,"");
                strcpy(filename,"Sort");
                strcat(filename,fileref);
                strcat(filename,".csv");
                ofstream SortFile(filename, ios::out);
                cout << "Step 4.1 - Writing file " << filename << endl;
                SortFile << "Elementos, Bubble, Heap, Merge, Selection, Quick, Shell,
Insertion" << endl;
           while (actual_element <= elements){
                   switch (option) {
                        case 1:
                                genNumbers (numbers,actual_element);
                        break;
                        case 2:
                                genNumbersSor (numbers,actual_element);
                        break;
                        case 3:
                                genNumbersInv (numbers,actual_element);
                        break;
                        default:
                                cout << "Option not valid";
                    return 1;
                        };

                   tBubble1 = tBubble2 = clock();
                   if ((option_algorithm == 1)|| (option_algorithm == 8) ||
(option_algorithm == 10) )
                   {
```

```
                cpyNumbers(numbers,aux_numbers,actual_element);
                tBubble1= clock();
                bubbleSort (aux_numbers,actual_element);
                tBubble2 = clock();
        }
        SortFile << actual_element << "," << (tBubble2-tBubble1);

        tHeap1 = tHeap2 = clock();
        if ((option_algorithm == 2)|| (option_algorithm == 9) ||
(option_algorithm == 10))
        {
                cpyNumbers(numbers,aux_numbers,actual_element);
                tHeap1= clock();
                heapSort (aux_numbers,actual_element);
                tHeap2 = clock();
        }
        SortFile << "," << (tHeap2-tHeap1);

        tMerge1 = tMerge2 = clock();
            if ((option_algorithm == 4)|| (option_algorithm == 9) ||
(option_algorithm == 10) )
            {
                cpyNumbers(numbers,aux_numbers,actual_element);
                tMerge1= clock();
                mergeSort (aux_numbers,temp,actual_element);
                tMerge2 =clock();
            }
            SortFile <<  "," << (tMerge2-tMerge1);

            tSelection1 = tSelection2= clock();
            if ((option_algorithm == 6)|| (option_algorithm == 8) ||
(option_algorithm == 10))
            {
                cpyNumbers(numbers,aux_numbers,actual_element);
                tSelection1= clock();
                selectionSort (aux_numbers,actual_element);
                tSelection2 = clock();
            }
            SortFile <<  "," << (tSelection2-tSelection1);

            tQuick1 = tQuick2 = clock();
            if ((option_algorithm == 5)|| (option_algorithm == 9) ||
(option_algorithm == 10))
            {
                cpyNumbers(numbers,aux_numbers,actual_element);
                tQuick1= clock();
                quickSort (aux_numbers,actual_element);
                tQuick2 = clock();
            }
            SortFile << "," << (tQuick2-tQuick1);
```

```
                    tShell1 = tShell2 = clock();
                    if ((option_algorithm == 7)|| (option_algorithm == 9) ||
(option_algorithm == 10))
                    {
                            cpyNumbers(numbers,aux_numbers,actual_element);
                            tShell1= clock();
                            shellSort (aux_numbers,actual_element);
                            tShell2 = clock();
                    }
                    SortFile << "," << (tShell2-tShell1);

                    tInsertion1 = tInsertion2 = clock();
                    if ((option_algorithm == 3)|| (option_algorithm == 8) ||
(option_algorithm == 10))
                    {
                            cpyNumbers(numbers,aux_numbers,actual_element);
                            tInsertion1= clock();
                            insertionSort (aux_numbers,actual_element);
                            tInsertion2 = clock();
                    }
                    SortFile <<  "," << (tInsertion2-tInsertion1) << endl;

            actual_element += step;
            }
            SortFile.close();
            actual_element=1;
            corrida ++;
          }
          cout << "Step 5 - Process completed" << endl;
        return 0;
}


void cpyNumbers (int numbers[], int aux_numbers[], int array_size)
// Esta funcion copia los numeros de un vector pasado como parametro a otro.
// Se esta usando para que los datos a ordenar sean los mismos evaluando
// los distintos algoritmos.
{
 int i;
 for (i = 0; i <= (array_size - 1); i++)
 {
            aux_numbers[i]=numbers[i];
 }
}

void genNumbersInv (int * numbers, int array_size)
// Esta funcion genera un vector con numeros completamente desordenados
{
 int i;

 for (i = 0; i <= (array_size - 1); i++)
```

```
 {
                numbers[i]=array_size -i;
 }
}

void genNumbersSor (int * numbers, int array_size)
// Esta funcion genera un vector con numeros completamente ordenados
{
 int i;

 for (i = 0; i <= (array_size - 1); i++)
 {
                numbers[i]=i;
 }
}

void printNumbers (int numbers[], int array_size)
{
 int i, j, temp;

 for (i = 0; i <= (array_size - 1); i++)
 {
                cout << numbers[i] << " ";
 }
 cout << endl;
}

void genNumbers (int numbers[], int array_size)
{
 int i, j, temp;

 for (i = 0; i <= (array_size - 1); i++)
 {
                numbers[i]=rand()%100;
 }
}

void bubbleSort(int numbers[], int array_size)
{
 int i, j, temp;

 for (i = (array_size - 1); i >= 0; i--)
 {
  for (j = 1; j <= i; j++)
  {
   if (numbers[j-1] > numbers[j])
   {
    temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
   }
```

```
    }
  }
}


void heapSort(int numbers[], int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}


void siftDown(int numbers[], int root, int bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}

void mergeSort(int numbers[], int temp[], int array_size)
{
  m_sort(numbers, temp, 0, array_size - 1);
```

```
}


void m_sort(int numbers[], int temp[], int left, int right)
{
 int mid;

 if (right > left)
 {
  mid = (right + left) / 2;
  m_sort(numbers, temp, left, mid);
  m_sort(numbers, temp, mid+1, right);

  merge(numbers, temp, left, mid+1, right);
 }
}

void merge(int numbers[], int temp[], int left, int mid, int right)
{
 int i, left_end, num_elements, tmp_pos;

 left_end = mid - 1;
 tmp_pos = left;
 num_elements = right - left + 1;

 while ((left <= left_end) && (mid <= right))
 {
  if (numbers[left] <= numbers[mid])
  {
   temp[tmp_pos] = numbers[left];
   tmp_pos = tmp_pos + 1;
   left = left +1;
  }
  else
  {
   temp[tmp_pos] = numbers[mid];
   tmp_pos = tmp_pos + 1;
   mid = mid + 1;
  }
 }

 while (left <= left_end)
 {
  temp[tmp_pos] = numbers[left];
  left = left + 1;
  tmp_pos = tmp_pos + 1;
 }
 while (mid <= right)
 {
  temp[tmp_pos] = numbers[mid];
  mid = mid + 1;
```

```
   tmp_pos = tmp_pos + 1;
  }

  for (i=0; i <= num_elements; i++)
  {
   numbers[right] = temp[right];
   right = right - 1;
  }
}

void selectionSort(int numbers[], int array_size)
{
 int i, j;
 int min, temp;

 for (i = 0; i < array_size-1; i++)
 {
  min = i;
  for (j = i+1; j < array_size; j++)
  {
   if (numbers[j] < numbers[min])
    min = j;
  }
  temp = numbers[i];
  numbers[i] = numbers[min];
  numbers[min] = temp;
 }
}

void shellSort(int numbers[], int array_size)
{
 int i, j, increment, temp;

 increment = 3;
 while (increment > 0)
 {
  for (i=0; i < array_size; i++)
  {
   j = i;
   temp = numbers[i];
   while ((j >= increment) && (numbers[j-increment] > temp))
   {
    numbers[j] = numbers[j - increment];
    j = j - increment;
   }
   numbers[j] = temp;
  }
  if (increment/2 != 0)
   increment = increment/2;
  else if (increment == 1)
   increment = 0;
```

```
    else
      increment = 1;
  }
}
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i=1; i < array_size; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
      numbers[j] = numbers[j-1];
      j = j - 1;
    }
    numbers[j] = index;
  }
}


void quickSort(int numbers[], int array_size)
{
  q_sort(numbers, 0, array_size - 1);
}


void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;
  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
```

```
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

Apendice 2 - Análisis de Algoritmos

Analizar un algoritmo significa determinar la cantidad de recursos necesarios para ejecutarlo (como por ejemplo, tiempo y almacenamiento). La mayoría de los algoritmos están diseñados para trabajar con datos de entrada de longitud arbitraria. Usualmente, la eficiencia o complejidad de un algoritmo se establece como una función que relaciona el tamaño del conjunto de datos de entrada con la cantidad de pasos o unidades de almacenamiento requeridos para ejecutar el algoritmo.

La mejor técnica para diferenciar la eficiencia de los algoritmos es el estudio de los órdenes de complejidad. El orden de complejidad se expresa generalmente en término de la cantidad de datos a procesar por el programa, cantidad que denominamos N.

La cantidad de tiempo de procesamiento de un algoritmo por lo general está en función de N, y puede expresarse según los casos típicos (promedio) de ese N, o bien según casos extremos no deseables (peor caso).

En el análisis teórico de los algoritmos, es común estimar su complejidad en sentido **asintótico**, es decir considerar valores de N tan grandes que la formula de la complejidad del algoritmo queda reducida a sólo sus miembros más relevantes. Esto es lo que hacen notaciones como la **"Big-O"**

Notación Big-O

La notación **Big-O** se utiliza para expresar la complejidad de un código. En un problema de tamaño N (por ejemplo N ítems a ordenar):

- Un método constante en el tiempo es de "orden 1": O(1)
- Un método que varía linealmente en el tiempo es de "orden N": O(N)
- Un método que varia cuadráticamente en el tiempo es de "orden N al cuadrado": $O(N^2)$

Las expresiones Big-O no tienen constantes o términos de orden bajo. Esto se debe a que cuando N es muy grande, las constantes y los términos mas bajos no existen (un método constante será más rápido que uno lineal y este será más rápido que uno cuadrático).

Definición formal:

La función T(N) es O(F(N)) si existe una constante c y para valores de N mayores que un valor n_0:

T(N) <= c * F(N)

La idea es que T(N) es la **exacta** complejidad de un método o algoritmo como una función del problema de tamaño N, y F(N) es el limite superior de la complejidad (por ejemplo, el tiempo, espacio o lo que sea de un problema de tamaño N no será peor que F(N)). En la practica se busca el menor F(N).

Por ejemplo, si consideramos $T(N) = 3 * N^2 + 5$. Podemos ver que T(N) es $O(N^2)$ eligiendo $c = 4$ y $n_0 = 2$. Esto es porque para todos los valores de N mayores que 2, se cumple que:

$3 * N^2 + 5 <= 4 * N^2$

T(N) **no** es O(N), Sin importar los valores que se elijan para la constante c y el valor de n_0, Siempre se puede buscar un valor de N mayor que n_0 tal que $3 * N^2 + 5$ es mayor que $c * N$.

Como Determinar las Complejidades

¿Cómo determinar el tiempo de ejecución de un código?, Depende del tipo de instrucciones utilizadas.

1. Secuencia de instrucciones
2. instrucción 1;
3. instrucción 2;
4. ...
5. instrucción k;

(Nota: Este código es una secuencia de exactamente k instrucciones)

El tiempo total es la suma de los tiempos de cada instrucción:

Tiempo total = tiempo (instrucción 1) + tiempo (instrucción 2) +... + tiempo (instrucción k)

Si cada instrucción es "simple" (solo involucra operaciones básicas) entonces el tiempo de cada instrucción es constante y el tiempo total también es constante: O(1). En los siguientes ejemplos se asume que las instrucciones son simples, a menos que se exprese lo contrario.

2. Instrucciones condicionales (if-then-else)

```
7.  if (condición) {
8.      secuencia de instrucciones 1
9.  }
10. else {
11.     secuencia de instrucciones 2
12. }
```

Aquí, se ejecutara la secuencia 1 o la secuencia 2. Entonces, el peor caso es el mas lento de los dos: max(tiempo(secuencia 1), tiempo(secuencia 2)). Por ejemplo, si la secuencia 1 es O(N) y la secuencia 2 es O(1) el peor caso para todo el código condicional debe ser O(N).

3. Ciclos

```
14.     for (i = 0; i < N; i++) {
15.         secuencia de instrucciones
16.     }
```

El ciclo se ejecuta N veces, entonces la secuencia de instrucciones también se ejecuta N veces. Como asumimos que las instrucciones son O(1), El tiempo total para el ciclo es N * O(1), lo cual seria O(N).

4. Ciclos anidados

```
18.     for (i = 0; i < N; i++) {
19.         for (j = 0; j < M; j++) {
20.             secuencia de instrucciones
21.         }
22.     }
```

El ciclo exterior se ejecuta N veces. Por cada una de esas ejecuciones, el ciclo interno se ejecuta M veces. Como resultado, las instrucciones en el ciclo interno se ejecutan un total de N * M veces. En este caso, la complejidad es de O(N * M). En el común de los casos, donde la condición de fin del ciclo interior es que j < N, igual al ciclo exterior, la complejidad total para los dos ciclos es de $O(N^2)$.

Apendice 3 - Referencias

1. http://www.cs.wisc.edu/~hasti/cs367-common/notes/COMPLEXITY.html
2. http://www.ncsu.edu/labwrite/res/gt/gt-reg-home.html
3. http://en.wikipedia.org/wiki/Category:Sort_algorithms
4. http://www.algoritmia.net/articles.php?id=31