Universidad de La Laguna
Escuela Técnica Superior de Ingeniería Informática

Selection Sort y Heap Sort

Informe - Diseño y Análisis de algoritmos

ETSII - ULL

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

Repositorio Github

Selection Sort

Selection Sort es un algoritmo de ordenación. No pertenece al grupo de los más eficientes ya que su complejidad es de $\Theta(n^2)$. Pero tiene la ventaja de que su implementación es bastante rápida y sencilla y requiere el menor número de operaciones de swap posibles con una complejidad espacial de $\Theta(1)$. Su funcionamiento básico se describe de la siguiente forma (iterativamente). Y más adelante un ejemplo de pseudocódigo del algoritmo.

1. Situarnos en la posición cero.
2. Buscar el mínimo elemento entre nuestra posición y el final.
3. Intercambiarlo por el de la posición actual.
4. Avanzar nuestra posición un elemento.
5. Volver al paso 2.

En cuanto al tipo de array a ordenar, no existe ningún tipo de aceleración o decremento de la velocidad si el array es completamente aleatorio, casi ordenado o con pocos valores únicos.

Pseudocódigo

```
1:    selectionSort(A):
2:          for i = 0 to A.size
3:                min = selectMin(A, i, A.size)
4:                swap(A[min], A[i]);
5:          return

6:    selectMin(A, start, end):
7:          min = start;
8:          for i = start to end
9:                if A[i] < A[min]
10:                     min = i
11:         return min

12:   swap(a, b):
13:         aux = a
14:         a   = b
15:         b   = aux
16:         return
```

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

Repositorio Github

En el pseudocódigo podemos ver que se sigue de forma sencilla los pasos nombrados anteriormente se puede observar además que el algoritmo es correcto, esto es, siempre se adecua al comportamiento especificado, para el Selection Sort puede ser comprobado de forma bastante sencilla afirmando que se cumple la propiedad de invariabilidad del bucle para ambos. Esta propiedad nos indica que una proposición sobre un bucle es cierta antes y después de cada iteración. En este caso: *"Al comienzo de cada iteración del bucle for, que está indexado por i (elemento actual), los elementos en A[0 … i − 1] están ordenados y los elementos en A[i ... max] (desordenados) son mayores o iguales a los elementos en A[0 … i - 1]".* Esta propiedad se cumple según los siguientes principios y confirma que el algoritmo es correcto:

- **Inicialización**: Es verdadera antes de la primera iteración. A[0] está ordenado, por ser sólo un elemento.
- **Mantenimiento**: Si es verdadera antes de la primera iteración, seguirá siendo verdadera antes de la siguiente iteración ya que cuando *i* avance un elemento, se habrá asegurado que el elemento anterior sea el menor de los siguientes (Pues es el que hemos seleccionado).
- **Finalización**: Una vez finaliza el bucle, *i* a recorrido todos los elementos y contiene el valor *n.* Conteniendo entonces *A[0 … n - 1]* los elementos ordenados, es decir, todos los elementos.

Análisis de complejidad

Código	Repeticiones	Coste	Explicación
selectionSort(A)			
`selectionSort(A):`	1	c_1	Llamada a la función.
`  for i = 0 to A.size`	$2n$	c_2	Comparación del bucle y asignación o incremento por cada iteración.
`    min = selectMin(A, i, A.size)`	$n(1 + t_1)$	c_3	Asignación y tiempo de ejecución de *selectMin* por cada iteración.
`    swap(A[min], A[i]);`	$n\ t_2$	c_4	Ejecución de *swap* por cada iteración.
`  return`	1	c_5	
selectMin(A, start, end)			
`selectMin(A, start, end):`	1	c_6	Llamada a la función.
`  min = start;`	1	c_7	Asignación.

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

`for i = start to end`	$2n$	c_8	Comparación del bucle y asignación o incremento por cada iteración.
`if A[i] < A[min]`	n	c_9	Comparación por cada iteración.
`min = i`	$n,\ 0$	c_{10}	Asignación en el peor caso (Será el que usemos) por cada iteración, nada en el mejor.
`return min`	1	c_{11}	
swap(a, b)			
`swap(a, b)`	1	c_{12}	Llamada a la función.
`aux = a`	1	c_{13}	Asignación.
`a = b`	1	c_{14}	Asignación.
`b = aux`	1	c_{15}	Asignación.
`return`	1	c_{16}	

Para poder calcular la complejidad final de la función **selectionSort(A)** vemos que depende de los valores t_1 y t_2 que se corresponde al tiempo de ejecución de las funciones **selectMin(A, start, end)** y **swap(a, b)** respectivamente. Debemos calcular estos valores primero.

$$t_1 = c_6 + c_7 + 2nc_8 + nc_9 + nc_{10} + c_{11}$$
$$t_1 = n(2c_8 + c_9 + c_{10}) + (c_6 + c_7 + c_{11})$$
$$t_2 = (c_{12} + c_{13} + c_{14} + c_{15} + c_{16})$$

A continuación podemos calcular la complejidad total, es decir la de la función **selectionSort(A).**

$$T(n) = c_1 + 2nc_2 + (1 + t_1)nc_3 + t_2nc_4 + c_5$$
$$T(n) = c_1 + 2nc_2 + (1 + (2c_8 + c_9 + c_{10})n + (c_6 + c_7 + c_{11}))nc_3 + t_2nc_4 + c_5$$
$$T(n) = c_1 + 2nc_2 + nc_3 + (2c_8 + c_9 + c_{10})n^2c_3 + (c_6 + c_7 + c_{11})nc_3 + t_2nc_4 + c_5$$
$$T(n) = n^2(2c_8c_3 + c_9c_3 + c_{10}c_3) + n\,(t_2c_4 + 2c_2 + c_6c_3 + c_7c_3 + c_{11}c_3 + c_3) + c_1 + c_5$$
$$T(n) = n^2a + n\,b + c,\ \ (t_2\ es\ una\ suma\ de\ constantes)$$

Podemos afirmar entonces que $T(n)\ \varepsilon\ \Theta(n^2)$ de acuerdo a nuestra afirmación inicial.

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

Heap Sort

HeapSort es un algoritmo de ordenación fácil de implementar. El funcionamiento del HeapSort se basa en construir un árbol binario utilizando el propio array y a partir de ahí recolocar los hijos si estos fueran mayores que los padres de esta forma obtendremos un árbol binario ordenado en la propia estructura del array. Es decir, el propio array estará ordenado.

Pertenece al grupo de los más eficientes ya que su complejidad es de $\Theta(nlog(n))$ pero no es **estable**. El primer bucle de **heapSort(A)** con complejidad $\Theta(n)$, pone el vector en orden. El segundo bucle en $\Theta(n\ log(n))$, extrae repetidamente el máximo y restaura el orden.

La función **sink(A, i)** se implementa de modo iterativo para que la complejidad espacial sea de $\Theta(1)$ ya que trabajamos sobre el propio array. La implementación recursiva por otro lado require $\Theta(logn)$ en complejidad (espacial) ya que utilizamos la pila de llamadas.

En cuanto al tipo de array a ordenar, en el caso casi clasificado, la fase del primer bucle destruye el orden original. En el caso inverso, la fase del primer bucle es lo más rápida posible, ya que el vector comienza en orden, pero entonces la fase del segundo bucle es típica. En el caso de algunas claves únicas, hay alguna aceleración.

Pseudocódigo

```
1:    heapSort(A):
3:          for i = (A.size/2) to 1
4:                sink(A, i)
6:          for i = A.size to 2
7:                swap(A[1], A[i]) // Misma función que en Selection Sort
8:          sink(A, i)
9:          return

10:   sink(A, i):
11:         x = A[i-1]
12:         while 2i <= A.size
13:               left = 2i
14:               right = 2i+1
15:               if (left == A.size) or (A[left] > A[right])
16:                     max = left
17:               else
18:                     max = right
19:               if (A[max] <= x)
```

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

```
20:                     break
21:                else
22:                     swap(A[i], A[max])
23:                     i = max
24:          return
```

Análisis de complejidad

Código	Repeticiones	Coste	Explicación
heapSort(A)			
`heapSort(A):`	1	c_1	Llamada a la función.
`for i = (A.size/2) to 1`	$2(n/2)$	c_2	Comparación del bucle y asignación o decremento por cada iteración.
`sink(A, i)`	$n\ t_1$	c_3	Ejecución de *sink* para cada iteración.
`for i = A.size to 2`	$2(n-1)$	c_4	Comparación del bucle y asignación o decremento por cada iteración.
`swap(A[1], A[i])`	$n\ t_2$	c_5	Ejecución de *swap* para cada iteración.
`sink(A, i)`	$n\ t_1$	c_6	Ejecución de *sink* para cada iteración.
`return`	1	c_7	
sink(A, i)			
`sink(A, i):`	1	c_8	Llamada a la función.
`x = A[i-1]`	1	c_9	Asignación.
`while 2i <= A.size`	$2logn$	c_{10}	Comparación y multiplicación para cada iteración. Se suceden *logn* veces **como caso peor** cuando hay que mover el nodo un nivel por cada uno de los niveles del árbol binario del heap.En un mejor caso, sale del bucle a través del *break* siguiente.
`left = 2i`	$2logn$	c_{11}	Asignación y multiplicación por iteración.
`right = 2i+1`	$3logn$	c_{12}	Asignación, multiplicación y suma por iteración.

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

`if (left == A.size) or (A[left] > A[right])`	$2logn$	c_{13}	Dos comparaciones por iteración.
`max = left`	$logn,\ 0$	c_{14}	Asignación por iteración en el caso de que entre tras la comprobación. Sin embargo se tomará como que siempre se cumple la condición necesaria.
`else`	-	-	-
`max = right`	$logn,\ 0$	c_{15}	Asignación por iteración en el caso de que entre tras la comprobación. Sin embargo se tomará como que nunca se cumple la condición necesaria..
`if (A[max] <= x)`	$logn$	c_{16}	Comparación por cada iteración.
`break`	1	c_{17}	En el caso de que no sea el peor caso, el break detendrá el bucle. Se tomará como que nunca es cierto.
`else`	-	-	-
`swap(A[i], A[max])`	t_2logn	c_{18}	Ejecución de swap por cada iteración. Se tomará como que la condición necesaria siempre se cumple.
`i = max`	$logn$	c_{19}	Asignación por cada iteración.Se tomará como que la condición necesaria siempre se cumple.
`return`	1	c_{20}	

Para poder calcular la complejidad final de la función **heapSort(A)** vemos que depende de los valores t_1 y t_2 que se corresponde al tiempo de ejecución de las funciones **sink(A, i)** y **swap(a, b)** respectivamente. Debemos calcular estos valores primero. (t_2 tiene el mismo valor que en el apartado anterior.

$$t_1 = c_8 + c_9 + 2log(n)c_{10} + 2log(n)c_{11} + 3lognc_{12} + 2log(n)c_{13} + log(n)c_{14} + log(n)c_{16} + t_2log(n)c_{18} + log(n)c_{19} + c_{20}$$

$$t_1 = log(n)K_1 + K_2$$

A continuación podemos calcular la complejidad total, es decir la de la función **heapSort(A).**

$$T(n) = c_1 + 2(n/2)c_2 + t_1nc_3 + 2(n-1)c_4 + t_2nc_5 + t_1nc_6 + c_7$$

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

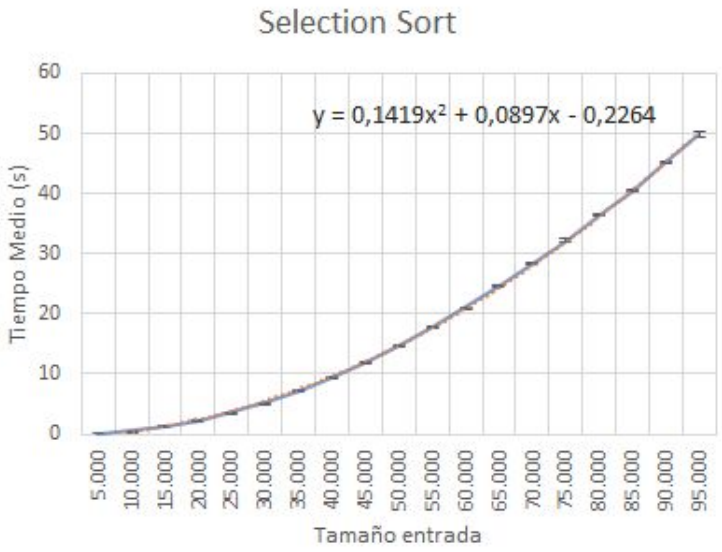
$$T(n) = c_1 + nc_2 + (log(n)K_1 + K_2)nc_3 + 2(n-1)c_4 + t_2nc_5 + (log(n)K_1 + K_2)nc_6 + c_7$$
$$T(n) = a\ n\ log(n) + b\ n + c$$

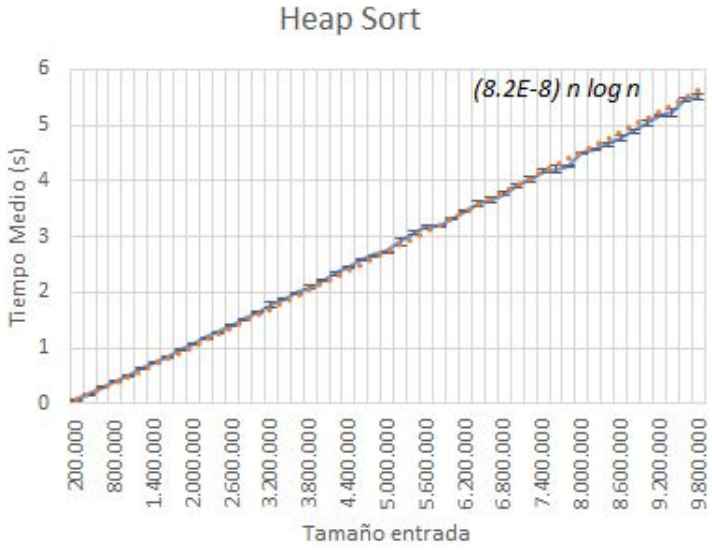
Podemos afirmar entonces que $T(n)\ \varepsilon\ \Theta(n\ log(n))$ de acuerdo a nuestra afirmación inicial.

Comparación

Podemos ver en las siguientes gráficas la comprobación experimental de los tiempos de cada uno de los algoritmos. Se puede observar que ambos efectivamente se ajustan a la curva generada por su complejidad. Se puede ver la gráfica de los tiempos del algoritmo en azul y una recta de regresión punteada en color naranja con su ecuación. Ambas coinciden de forma casi perfecta.

Heap Sort en el caso mostrado en el que se emplea un array de entrada con números colocados aleatoriamente avanza a una velocidad mucho mayor que el Selection Sort. Sucede igual para el resto de casos en los que el array se puede encontrar casi ordenado, o con pocos elementos únicos.





***Nota**: Ambos gráficos se pueden ver a mayor tamaño en el repositorio de la práctica.*

Jorge Sierra Acosta
Ruymán Rodríguez Martín
Airam Manuel Navas Simón

Repositorio Github