FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1

[75.29/95.06] Teoría de Algoritmos I
Primer cuatrimestre de 2018

Grupo MinMax

Alumno	Padrón	Mail
del Mazo, Federico	100029	delmazofederico@gmail.com
Djeordjian, Esteban Pedro	100701	edjeordjian@gmail.com
Kristal, Juan Ignacio	99779	kristaljuanignacio@gmail.com
Raveszani, Nicole	101031	nraveszani@gmail.com

Índice

1. Cálculo Empírico de Tiempos de Ejecución

1.1. Complejidad Teórica

1.1.1. Selección

El ordenamiento de selección es uno de los algoritmos de ordenamiento más faciles de implementar. En cada iteracion, se halla el máximo elemento de la seccion del arreglo desordenado y se intercambian las posiciones de este valor y el ultimo elemento. Se reduce en una unidad la seccion del arreglo desordenado y al resto se lo considera ordenado.

De acuerdo a nuestra implementación[1] del ordenamiento de selección, la complejidad teóricas es:

$Tseleccion(n) = O(1+1+1+n*(1+n+(1+1+n(2)+1)+4)) = O(2n^2+9n+3) \in O(n^2)$

Esto viene de analizar linea por linea teniendo en cuenta:

- Hacer una asignación y comparar variables: $O(1)$
- Operadores aritméticos básicos: $O(1)$
- Sacar la longitud de un arreglo: $O(1)$
- Hacer un ciclo definido que itera por todo el arreglo: $O(n * f(n))$ siendo $f(n)$ lo que hay dentro del ciclo.
- Hallar el indice de un elemento: $O(n)$

Este ordenamiento itera siempre por el arreglo sin importar su disposicion inicial. Por lo tanto, su peor caso y su caso promedio son iguales al tiempo analizado: (EL PEOR CASO DE SELECCION ES UN ARREGLO ORDENADO)

$PeorTselecccion(n) \in O(2n^2+9n+3) \in O(n^2)$

$PromedioTselecccion(n) \in O(2n^2+9n+3) \in O(n^2)$

1.1.2. Inserción

Inicialmente, se tiene un solo elemento y se lo considera un conjunto ordenado. En cada iteración, se compara al siguiente elemento del conjunto ordenado con su predecesor y se intercambia en caso de ser menor a este. Se itera de esta forma hasta que la condicion anterior no se cumpla. Al igual que el de seleccion, el algoritmo consiste de dos ciclos anidados que recorren el arreglo entero, por lo tanto, el tiempo esperado sera cuadratico

Este ordenamiento usa las mismas herramientas usadas en el ordenamiento de selección. Por lo tanto, con las consideraciones anteriores tomadas, queda:

$O(1+1+1+n(3+2+2+1+n(1+3))) = O(3n^2+8n+3) \in O(n^2)$

Siendo la condicion de corte que en la iteracion el elemento actual no sea menor a lo iterado descendientemente, el peor caso del algoritmo es donde el arreglo inicial esta dispuesto de manera descendiente, de tal forma que el ciclo sea ejecutado sin cortes.

$PeorTinsercin(n) \in O(3n^2+8n+3) \in O(n^2)$

Como es claro que encontrar en un arreglo un intervalo ordenado (suficientemente grande) es mucho menos probable que no hacerlo (es decir, para un arreglo de n elementos, de las $n!$ posibles distribuciones de los elementos, solo una estara ordenada, y con un n no demasiado grande, se cumple que $n! - 1 \approx n$). Luego, en el caso promedio se tiene que el algoritmo tambien tiene orden cuadratico.

$PromedioTinsercin(n) \in O(3n^2+8n+3) \in O(n^2)$

[1] Las complejidades de los métodos nativos de las listas de Python 3 fueron sacadas de: https://wiki.python.org/moin/TimeComplexity

1.1.3. Mergesort

El algoritmo de mergesort es un ordenamiento que utiliza la tecnica de División y Conquista, y por lo tanto, para calcular su complejidad teorica va a ser utilizado el Teorema Maestro. Si el arreglo solo tiene un elemento o ninguno se considera ordenado. En otro caso, el arreglo se subdivide por la mitad creando dos nuevos arreglos. Cada uno de estos arreglos se vuelve a subdividir de forma recursiva hasta que queda un elemento en cada subarreglo. A partir de aqui, cada mitad creada se une de forma ordenada en un nuevo arreglo hasta que queda un unico arreglo ordenado.

En cuanto a código se introducen nuevos metodos:

- `Append` y `pop`: $O(1)$
- `Extend`: $O(k)$ siendo k la cantidad de elementos agregados

También es de notar que este es el primer algoritmo donde usamos llamadas recursivas, por su naturaleza de división y conquista.

Aplicando el Teorema Maestro:

$Tmergesort(n) = a.Tmergesort(\frac{n}{b}) + f(n^c)$

La cantidad de llamadas recursivas (a) son dos. Cada llamada recursiva divide la entrada inicial a la mitad (b=2). Por último, el tiempo de las llamadas no recursivas es

$O(1 + 1 + 1 + 4 + 1 + 2 + n(1 + 1 + 1) + 1 + \frac{n}{2} + 1) = O(\frac{9n}{2} + 11)$

(asumiendo un n lo suficientemente grande como para que se requieran intercalar aproximadamente n elementos la mayoría de las veces), por lo que d=1. Se tiene entonces:

$Tmergesort(n) = 2.Tmergesort(\frac{n}{2}) + f(n)$

Como $(a = b^c)$ $(2 = 2^1)$, se tiene que:

$Tmergesort(n) \in O(n \log n)$

Texto sobre peor tiempo y tiempo promedio de mergesort

$PeorTmergesort(n) \in O(n \log n))$

$PromedioTmergesort(n) \in O(n \log n))$

1.1.4. Quicksort

Al igual que mergesort, quicksort utiliza la tecnica de División y Conquista. Se elige un elemento del arreglo como pivote (en nuestra implementacion se elige al primero) y a partir del pivote se arman tres arreglos: la de todos los elementos menores al pivote, la conformada solo por el pivote y la de todos los elementos mayores al pivote. Con esto se obtiene un orden relativo al pivote. El mecanismo antes descripto se repite recursivamente para cada parte. El arreglo final se obtiene concatenando las sublistas ordenadas. Al igual que en Mergesort, siendo División y Conquista la técnica utlilizada, se usa el Teorema Maestro para calcular su complejidad.

Considerando las complejidades analizadas hasta ahora (en código no hay nada nuevo) y aplicando el Teorema Maestro se tiene:

$Tquicksort(n) = a.Tquicksort(\frac{n}{b}) + f(n^c)$

La cantidad de llamadas recursivas (a) son 2. Para analizar b, se debe considerar dos casos. Por un lado, en el caso promedio, b será igual a 2, ya que en un arreglo arbitrariamente grande, es mucho más probable tomar en cualquier posición a un elemento alejado de las cotas superiores e inferiores que muy próximo a ellas, por lo que la recursión dividirá el problema en "dos partes iguales" (probablemente nunca sean exactamente iguales, pero con un n arbitrariamente grande, las diferencias que puedan existir son despreciables en tanto el pivote elegido no esté lo suficientemente próximo a una cota superior o inferior). El tiempo de las llamadas no recursivas es $O(4n + 4)$, por lo que $c = 1$

Para este caso, como $(a = b^c)$ $(2 = 2^1)$, se tiene que:

$PromedioTquicksort(n) \in O(n \log n))$

Pero existe un peor caso en donde el pivote elegido es una el menor o el mayor elemento del arreglo, dividiendo el arreglo en dos partes de 1 y $n - 1$ elementos respectivamente. Por lo tanto, usamos como peor caso un arreglo ordenado descendientemente y tomando de pivote el primer

elemento del arreglo. Para este peor caso se tiene que b=1, ya los elementos quedan "a la derecha o a la izquierda" del pivote. Para este caso, donde no se puede aplicar el teorema del maestro (porque no hay división y conquista propiamente dicha) queda:
$PeorTquicksort(n) = O(n(4n+4)) = O(4n^2+4n) \in O(n^2)$

1.1.5. Heapsort

En el heapsort se utiliza la estructura de cola de prioridad (heap). Se construye un heap de minimos con los elementos del arreglo. Al ser la raiz el menor elemento, se extrae la raiz y se la guarda en la primera posicion del arreglo. De esta forma, se extrae en sucesivas iteraciones la raiz y el elemento es guardado en la siguiente posicion hasta vaciar el heap.
Los tiempos del heap de minimos implementado son:

- `Heapify`: $O(x)$
- `Upheap`: $O(x)$
- `Downheap`: $O(x)$
- `Heappop`: $O(x)$
- `Heapush`: $O(x)$

Considerando todo esto queda:
$O(n+n+1+1+1+n(1+\log n) = O(3n+3+n\log n) \in O(n\log n)$
Texto sobre peor tiempo y tiempo promedio de heapsort
$PeorTheapsort(n) \in O(3n+3+n\log n) \in O(n\log n)$
$PromedioTheapsort(n) \in O(3n+3+n\log n) \in O(n\log n)$

1.2. Comparación

1.2.1. Peores tiempos

En orden ascendiente de eficiencia:

$PeorTinsercion(n) \in O(4n^2+9n+2) \in O(n^2)$
$PeorTselecccion(n) \in O(3n^2+8n+3) \in O(n^2)$
$PeorTquicksort(n) \in O(4n^2+4n) \in O(n^2)$
$PeorTheapsort(n) \in O(3n+3+n\log n) \in O(n\log n)$
$PeorTmergesort(n) \in O(n\log n)) \in O(n\log n)$

1.2.2. Tiempos promedio

En orden ascendiente de eficiencia:

$PromedioTinsercion(n) \in O(4n^2+9n+2) \in O(n^2)$
$PromedioTselecccion(n) \in O(3n^2+8n+3) \in O(n^2)$
$PromedioTheapsort(n) \in O(3n+3+n\log n) \in O(n\log n)$
$PromedioTmergesort(n) \in O(n\log n)) \in O(n\log n)$
$PromedioTquicksort(n) \in O(n\log n)) \in O(n\log n)$

Para poder terminar de ubicar en la escala a quicksort y mergesort, se pone el foco en la complejidad de su parte no recursiva: por parte de quicksort es $O(4n+4)$, y en el caso de mergesort es $O(\frac{9n}{2}+11)$. Es decir, la diferencia fundamental en ambos casos es que en el peor caso quicksort será mucho menos eficiente.

1.3. Peores sets

1.3.1. Selección

Como se ve en el análisis, el comportamiento de este algoritmo no depende de las característicos del arreglo. En término más específicos, el algoritmo de selección no es un algoritmo "adaptativo": su conducta no se ve afectada por ninguna característica del array, por lo cual realizará el mismo número de comparación entre elementos tanto en el peor caso, en el caso promedio y en el mejor caso (considerando una longitud de arreglo arbitrariamente grande).(EL PEOR CASO DE SELECCION ES ARREGLO ORDENADO)

1.3.2. Inserción

Este algoritmo sí presenta un peor caso: cuando el arreglo de entrada se encuentra ordenado de forma descendente (y no al revés, según la implementación dada). El bucle interno del algoritmo deberá comparar cada elemento hasta llegar a la primera posición del conjunto, con lo cual realizará el máximo número de comparaciones posibles entre elementos. Esto también está contemplado dentro de su comportamiento en promedio: se destaca el caso en que, si el arreglo estuviera ordenado al revés, el algoritmo terminaría en un orden de O(n) (siendo para este caso trivial, mejor que cualquier ordenamiento).

1.3.3. Mergesort

Como se analizo con el Teorema Maestro, Mergesort es de orden $O(n \log n)$, por lo que no presenta un peor caso para ningun set en general.

1.3.4. Quicksort

Quicksort presenta un caso en que su comportamiento resulta cuadrático: cuando todos (o casi todos) los elementos son menores que el pivote seleccionado. En otras palabras, el pivote es una cota del set. En nuestra implementación, Quicksort elige siempre al primer elemento del arreglo como pivote, por lo que un set que cumpla con estas características es uno ordenado de forma descendente.

1.3.5. Heapsort

Para este algoritmo, el numero de comparaciones entre los elementos del arreglo puede variar en poca proporción dependiendo orden en que se presentan los mismos (según su ubicación en el heap). Aún así, tanto el caso promedio como el peor, su tiempo de ejecución será $O(nlogn)$, lo cual significa que asintóticamente no hay diferencia entre ambos, aunque pueden diferir en un valor constante.

1.4. Tiempos de ejecución

Se calcularon los promedios de cada tiempo para cada cantidad de elementos y cada algoritmo:

Cuadro 1: Tiempos promedio sobre 10 sets aleatorios

Promedios	50 Elementos	100 Elementos	500 Elementos	1000 Elementos	2000 Elementos	3000 Elementos	4000 Elementos	5000 Elementos	7500 Elementos	10000 Elementos
heapSort	0.0001918	0.0004454	0.0031486	0.0071124	0.0155359	0.0244543	0.0338884	0.0436388	0.0693577	0.0948624
insertionSort	0.0001648	0.0005625	0.0131826	0.0514842	0.1984840	0.4477631	0.7776322	1.2132667	2.6847850	4.8147374
mergeSort	0.0001562	0.0003355	0.0022104	0.0047541	0.0107222	0.0173661	0.0247102	0.0328206	0.0544240	0.0787522
quickSort	0.0000776	0.0001754	0.0010672	0.0022888	0.0049410	0.0076537	0.0106931	0.0135835	0.0211154	0.0289411
selectionSort	0.0001518	0.0004635	0.0098650	0.0401897	0.1618426	0.3629149	0.6486356	1.0275165	2.3131156	4.1221336

GRAFICOS

Es claro que el caso de orden solo afecta descendente en quicksort e insertionsort. Se ve en la comparación que para pocos elementos el comportamiento de los ordenamientos es similar. Pasando los mil elementos se ve como quicksort e insertsort empeoran notablemente. Para el set

mayor, se ve que insertionSort tarda cuatro veces más, y quicksort tarda más de 200 veces lo que tardaba en promedio. El resto de los ordenamientos no se ve afectado notablemente. En el gráfico es más que notorio como quicksort tomó un orden cuadrático al igual que los ordenamientos por inserción y selección.

En el siguiente gráfico se muestra la comparación explícita del insertion sort: y a continuación el análogo de quicksort

TABLA DE PEORES SETS

GRAFICOS DE PEORES SETS

Las tablas enteras con todas las ejecuciones pueden verse en el anexo al final.

1.5. Comparación con valores teóricos

De acuerdo a las complejidades analizadas para el peor caso, se predice que los algoritmos de selección inserción y quicksort tardarán mucho más en ordenar arreglos que heapsort o mergesort. Esto se cumple, con la salvedad que en el análisis teórico para el peor caso se preveía que el ordenamiento por selección tendría peor desempeño que quicksort, lo cual no ocurrió, por ser el set especialmente diseñado para que quicksort se comporte de forma poco eficiente.

Con respecto al caso promedio, los tiempos se cumplieron de forma exactamente coherente con el orden de eficiencia predicho (en orden ascendente) en el análisis teórico: inserción, selección, heapsort, mergesort y quicksort. En síntesis, excepto por el peor caso de quicksort que no fue considerado de forma explícita, el análisis teórico se condice con los valores visualizados en la práctica.

2. Algoritmo Gale-Shapley

2.1. Algoritmo

El algoritmo de asignación propiamente dicho se reduce a la función homónima:

```
1 codigocodigocodigo
```

2.2. Demostraciones

2.2.1. Tiempo polinómico

Para facilitar la notación de está sección, J será la cantidad de jugadores y E la cantidad de equipos. Para la justificación detallada de cada sentencia del algoritmo, ver los comentarios de la complejidad en el código.

En el algoritmo el primer método que se llama es jugadoresEquiposGS. En caso en que se especifique la creación de los archivos .rtf, el método generarArchivos tiene una complejidad es de orden O(E*J), ya que por cada equipo se debe crear un archivo de longitud igual a la cantidad de jugadores, y lo análogo ocurre a la recíproca.

La carga de archivos con el método cargaDeArchivos naturalmente también de orden O(J*E) ya que por cada jugador que se quiera crear, se debe recorrer un archivo cuya longitud es igual a la cantidad de equipos, y viceversa.

La asignación, donde se genera el matching establece, tiene como condición de corte que no haya más vacantes en los equipos. Considerando esto, el hecho de que a continuación en el algoritmo se itere por cada equipo, y que se verifique para cada uno si todos tiene vacantes, es solo una especificación para lograr cubrir todas las vacantes, por lo que no agregan un orden diferente: simplemente es la forma de implementar que el algoritmo avance a partir de que se cubren distintas vacantes en distintas equipos. Esto implica que cada equipo deberá completar todos sus lugares libres, y como hay igual cantidad de jugadores que de vacantes, el peor caso se considera cuando cada equipo debe preguntarle a todos los jugadores si desean ocupar una vacante: esto es O(J). Como hay E equipos, el orden de la asignación también es O(J*E).

Finalmente, almacenar el archivo de la asignación tiene un orden línea O(J+E). Esto se debe a que por cada equipo, se guarda su número seguido de los números de jugadores que tiene. La cantidad de jugadores que tiene un equipo es igual a sus vacantes (ver demostración en el próximo item de que cada equipo llena todas sus vacantes) y como la cantidad de vacantes totales es igual al número de jugadores, se tiene que se guardó un archivo con J + E números.

El orden cuadrático del algoritmo, entonces, es evidente, ya que es el peor caso de complejidad a lo largo de las funciones del algoritmo. Como O(J*E) es un orden polinomial, el algoritmo es de orden polinomial.

2.2.2. Matching estable

Para está demostración, se simplificará la estructura original de la solución implementada. Considerando el primer método llamado "jugadoresEquiposGS", se excluyen las características de la generación de los archivos .'prf', su carga, y el correspondiente almacenamiento del archivo donde se guarda la asignación estable. Entonces, el método se reduce a la asignación propiamente dicha.

Por una cuestión de comodidad, se puede traducir esto a un psuedcódigo basado en la documentación del método de asignación: "'Mientras haya un equipo con vacantes."' "'El equipo actual ofrece una vacante a su jugador favorito actual"' "'Si el jugador esta libre, acepta la vacante"' "'Si no esta libre, pero prefiere más al equipo actual, acepta la vacantey el otro equipo pierde a ese jugador."'

Como hay igual cantidad de jugadores que de vacantes totales, se ve que todos los equipos podrán cubrir todos su puestos. Supongamos que hubiera un equipo .ª"que ya ha .ºfrecido una vacante.ª todos los jugadores de acuerdo con su favoritismo, y ninguno la hubiera aceptado. Esto es decir que todos los jugadores ya han aceptado una vacante en un equipo previamente. Pero esto es una contradicción, porque existe la misma cantidad de jugadores que de vacantes totales, y por hipótesis hay una (al menos) vacante que .ª"no puede cubrir. Justamente, la contradicción viene de suponer que un equipo no ha podido satisfacer una de sus vacantes ofreciéndole a todos los jugadores, por lo que esto es falso. Con esto se ve además que la máxima cantidad de "proposiciones"que un equipo puede hacer es igual a la cantidad de vacantes disponibles en total, es decir, con la cantidad de jugadores.

Ahora se demostrar que una vez que cada equipo haya cubierto sus vacantes, no habrá parejas conflictivas (inestabilidades). La misma podría darse entre (j1,a) y (j2,b), donde j1 es un jugador que quiere estar en el equipo b, j2 es un jugador que quiere estar en el equipo a, a es un equipo que prefiere más a j2 sobre j1 y b es un equipo que prefiere más a j1 sobre j2.

Supongamos que a .elige primero". Si j1 aceptó la vacante en a, es porque j1 estaba libre, o bien, porque prefería estar en a que en otro equipo c (no necesariamente c = b).

Supongamos que era porque estaba solo. Luego b prefiere más a j2 que a j1, porque le ofreció su vacante antes. Si esto no fuera así, b le hubiera ofrecido antes su vacante a j1, y el mismo se hubiera cambiado de equipo, pero esto no sucedió. En este caso se llega a un absurdo, porque por hipótesis b prefiere más a j1 (en otro caso la pareja no es conflictiva y no hay inestabilidad).

Entonces debemos suponer que j1 aceptó la vacante porque estaba en otro equipo c. Luego, se llega al mismo absurdo anterior, porque al fin y al cabo b termina con j2.

Luego, debe ser que b .eligió primero". Si j2 aceptó la vacante en b, es porque j2 estaba libre, o bien, porque prefería estar en b que en otro equipo d (no necesariamente d = b o d = c). En cualquiera de los dos casos se llega a un absurdo similar al anterior. Todas los caminos a los que conduce la hipótesis "hay una inestabilidad en la asignación final"hacen llegar a una contradicción, por lo que queda probado que la asignación es estable.

No existe posibilidad que quede un equipo con alguna vacante libre, o un jugador sólo, como se probó al principio.

2.3. Ejecución

En el anexo junto con el código del algoritmo se ve la posibilidad que brinda de generar los archivos prf. El resultado de una de las ejecuciones fue la siguiente asignación:

```
1: 150 120 119 11 22 175 28 15 78 130
2: 182 200 8 191 44 36 60 134 17 159
3: 189 34 1 138 142 45 190 100 197 187
4: 88 9 132 46 62 91 188 103 145 70
5: 48 20 178 27 168 97 30 176 124 98
6: 125 129 25 86 180 165 106 5 177 12
7: 155 158 31 54 195 93 146 29 24 170
8: 164 63 110 127 109 140 96 160 149 143
9: 154 169 198 161 61 99 84 179 33 151
10: 173 69 35 117 163 65 112 118 104 43
11: 166 76 58 137 185 18 77 94 2 186
12: 51 4 171 199 87 162 19 181 85 80
13: 49 40 55 101 26 123 131 92 114 72
14: 193 115 39 37 135 6 172 38 141 174
15: 10 147 167 73 156 14 133 57 74 152
16: 126 192 136 148 71 23 52 153 111 32
17: 116 196 67 82 64 21 59 68 107 113
18: 105 56 90 66 47 183 7 75 108 16
19: 41 81 102 184 3 122 194 79 157 139
20: 42 83 13 128 53 50 144 89 121 95
```

A. Ejecución y compilación

Todo el trabajo fue codificado en Python 3.6.1.

A.1. Ordenamientos: Ejecución general

El programa principal es ejecutado desde el archivo main, especificando por linea de comando el caso a ejecutar, entre Randomz "Peor", refiriendose a los sets por donde se ejecutaran los ordenamientos. La ejecución consiste de iterar sobre los sets generados previamente y en cada uno ejecutar cada ordenamiento estudiado sobre distintas cantidades del mismo elemento. Este archivo genera un 'SalidaRandom.csv' con las estadisticas necesarias para analizar.

```
>>> python main.py Random
>>> python main.py Peor
```

A.2. Generación de sets aleatorios

El script generarSetRandom.py permite generar los sets aleatorios de n numeros (en este caso, n=10000) en un rango determinado. Ejemplo de ejecución:

```
>>> from generarSetRandom import generarSet
>>> generarSet("set01.txt", 1, 1000000, 10000)
```

A.3. Pruebas de ordenamientos

Para verificar que los ordenamientos anden correctamente se realizaron varias pruebas unitarias que verfiquen distintos comportamientos desde los casos triviales hasta los casos borde y demás. Todas las pruebas son iguales, e incluyen asserts con ordenamientos aleatorios.

```
>>> python3 heapTest.py -v
```

A.4. Medición de tiempos

Los tiempos de ejecución fueron calculados con el modulo `time` de Python, en particular su función `process_time` [2].

A.5. Stable matching

Para generar un matching estable, se debe importar el archivo matchinGS.py y ejecutar la siguiente línea: matchingGS('ruta de los archivos de jugadores y su correspondiente nombre sin número', 'extensión de los archivos jugadores', 'ruta de los archivos de equipos y su correspondiente nombre sin número', 'extensión de los archivos de equipos', 'nombre completo del archivo donde se guardará la asignación', cantidad de jugadores, cantidad de equipos, cantidad de vacantes por equipo, 'y' si se quieren generar los archivos o cualquier otro caracter en otro caso)

Por ejemplo, para cumplir con la consigna se requieren los siguientes comandos:

```
>>> from matchingGS import matchingGS
>>> matchingGS('archivos/jugador_', '.prf', 'archivos/equipo_', '.prf', 'asignacion.txt', 200, 20
```

B. Tablas de ejecución

B.1. Tiempos de ejecución sobre sets aleatorios

B.2. Tiempos de ejecución sobre sets de peores casos

[2] https://docs.python.org/3/library/time.html#process_time

Cuadro 2: Tiempos de ejecución sobre 10 sets aleatorios

50 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0001990	0.0001907	0.0001891	0.0001938	0.0001934	0.0001917	0.0001918	0.0001895	0.0001913	0.0001882
insertionSort	0.0001892	0.0001371	0.0001660	0.0001588	0.0001855	0.0001692	0.0001687	0.0001649	0.0001399	0.0001701
mergeSort	0.0001559	0.0001691	0.0001497	0.0001473	0.0001871	0.0001535	0.0001553	0.0001475	0.0001494	0.0001474
quickSort	0.0000930	0.0000783	0.0000656	0.0000742	0.0000859	0.0000790	0.0000683	0.0000764	0.0000835	0.0000715
selectionSort	0.0001585	0.0001462	0.0001457	0.0001572	0.0001770	0.0001532	0.0001471	0.0001412	0.0001484	0.0001431

100 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0004628	0.0004357	0.0004416	0.0004395	0.0004721	0.0004456	0.0004398	0.0004393	0.0004376	0.0004396
insertionSort	0.0006016	0.0004875	0.0005864	0.0005397	0.0005859	0.0005434	0.0005937	0.0005453	0.0005518	0.0005897
mergeSort	0.0004126	0.0003231	0.0003208	0.0003193	0.0003264	0.0003165	0.0003248	0.0003278	0.0003216	0.0003626
quickSort	0.0002314	0.0001618	0.0001489	0.0001659	0.0002118	0.0001675	0.0001430	0.0002106	0.0001622	0.0001512
selectionSort	0.0004638	0.0004911	0.0004886	0.0004427	0.0004483	0.0004572	0.0004580	0.0004540	0.0004462	0.0004848

500 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0030533	0.0030936	0.0032225	0.0030782	0.0038497	0.0030170	0.0030266	0.0030229	0.0030469	0.0030654
insertionSort	0.0136415	0.0124605	0.0127208	0.0127717	0.0132443	0.0131570	0.0133048	0.0137052	0.0131902	0.0136295
mergeSort	0.0022618	0.0022861	0.0022894	0.0021574	0.0021868	0.0021266	0.0021218	0.0021060	0.0022879	0.0022802
quickSort	0.0011439	0.0010552	0.0010782	0.0010665	0.0010709	0.0010688	0.0010897	0.0010095	0.0009802	0.0011091
selectionSort	0.0097367	0.0097802	0.0096571	0.0098280	0.0099636	0.0099066	0.0101287	0.0097905	0.0100458	0.0098125

1000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0067792	0.0070043	0.0068986	0.0069345	0.0072190	0.0073125	0.0081850	0.0068475	0.0070931	0.0068505
insertionSort	0.0557468	0.0530118	0.0504737	0.0531597	0.0493994	0.0491072	0.0500725	0.0509751	0.0517923	0.0511030
mergeSort	0.0047706	0.0046603	0.0047385	0.0048127	0.0047386	0.0050988	0.0047663	0.0046766	0.0046974	0.0045815
quickSort	0.0024357	0.0022442	0.0021556	0.0022674	0.0023423	0.0024573	0.0023988	0.0022838	0.0021472	0.0021557
selectionSort	0.0392514	0.0398083	0.0408062	0.0390289	0.0409778	0.0399334	0.0419789	0.0391787	0.0403418	0.0405913

2000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0151945	0.0158168	0.0157971	0.0155492	0.0158204	0.0156259	0.0154056	0.0153501	0.0154657	0.0153334
insertionSort	0.2046429	0.2006181	0.2024749	0.2056349	0.1898632	0.1968394	0.1917384	0.1970580	0.1982147	0.1977560
mergeSort	0.0104069	0.0110417	0.0106184	0.0104917	0.0111254	0.0107095	0.0108427	0.0105588	0.0108963	0.0105309
quickSort	0.0052252	0.0050108	0.0047761	0.0049240	0.0050046	0.0054048	0.0046839	0.0048581	0.0048070	0.0047157
selectionSort	0.1591686	0.1619187	0.1596146	0.1594604	0.1612478	0.1636533	0.1664229	0.1593405	0.1594487	0.1681502

3000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0248542	0.0245927	0.0246410	0.0244733	0.0241362	0.0240608	0.0248536	0.0244547	0.0240737	0.0244031
insertionSort	0.4642353	0.4359567	0.4520979	0.4604680	0.4363386	0.4307528	0.4336375	0.4531719	0.4739809	0.4369914
mergeSort	0.0175188	0.0174770	0.0176094	0.0170248	0.0173747	0.0174046	0.0174765	0.0171947	0.0172623	0.0173179
quickSort	0.0081988	0.0078375	0.0073443	0.0076117	0.0080461	0.0079049	0.0071992	0.0076201	0.0074061	0.0073687
selectionSort	0.3607497	0.3639719	0.3580664	0.3629982	0.3609123	0.3635284	0.3722692	0.3614427	0.3647209	0.3604892

4000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0346565	0.0336802	0.0339677	0.0340717	0.0336008	0.0335720	0.0339500	0.0333064	0.0346048	0.0334737
insertionSort	0.7642057	0.7622381	0.7893028	0.8062393	0.7972879	0.7578662	0.7728094	0.7724688	0.7829292	0.7709742
mergeSort	0.0267617	0.0245633	0.0243198	0.0249443	0.0243686	0.0246409	0.0246596	0.0242289	0.0243485	0.0242665
quickSort	0.0114064	0.0120993	0.0102182	0.0103307	0.0106509	0.0105468	0.0099800	0.0104413	0.0107408	0.0103171
selectionSort	0.6733503	0.6439389	0.6497549	0.6419530	0.6407484	0.6357381	0.6603671	0.6480176	0.6498467	0.6426408

5000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0434337	0.0438778	0.0438421	0.0439822	0.0440419	0.0435336	0.0434554	0.0436433	0.0435107	0.0430669
insertionSort	1.2046899	1.2067181	1.2318115	1.2126116	1.2713901	1.2186838	1.2074905	1.1798034	1.1919579	1.2075106
mergeSort	0.0334444	0.0321602	0.0321446	0.0328516	0.0330439	0.0323602	0.0322786	0.0320510	0.0334991	0.0343719
quickSort	0.0155413	0.0134631	0.0130190	0.0133885	0.0141153	0.0138595	0.0127920	0.0134561	0.0133596	0.0128409
selectionSort	1.0469931	1.0154521	1.0209151	1.0145462	1.0357843	1.0230973	1.0330363	1.0267096	1.0410148	1.0176163

7500 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0761486	0.0690381	0.0677706	0.0684095	0.0693999	0.0695108	0.0689707	0.0679884	0.0679949	0.0683460
insertionSort	2.7435652	2.7176744	2.6794948	2.6554472	2.6782472	2.7283597	2.6667099	2.6543722	2.7092949	2.6146844
mergeSort	0.0356103	0.0340695	0.0330913	0.0342600	0.0358728	0.0365334	0.0343427	0.0339535	0.0334051	0.0331010
quickSort	0.0230795	0.0208836	0.0197867	0.0208041	0.0225761	0.0219401	0.0202990	0.0211298	0.0203927	0.0202629
selectionSort	2.3154659	2.3086698	2.3049995	2.2768136	2.3440388	2.3188534	2.3363819	2.3072892	2.2852925	2.3333513

10000 Elementos	SetRandom01.txt	SetRandom02.txt	SetRandom03.txt	SetRandom04.txt	SetRandom05.txt	SetRandom06.txt	SetRandom07.txt	SetRandom08.txt	SetRandom09.txt	SetRandom10.txt
heapSort	0.0962717	0.0939746	0.0953125	0.0946152	0.0952729	0.0962848	0.0934928	0.0938169	0.0944631	0.0951193
insertionSort	4.7843860	4.9401893	4.8736038	4.9175287	4.6675018	5.2088326	4.7089758	4.6241228	4.7480252	4.6742076
mergeSort	0.0788291	0.0790973	0.0782618	0.0793668	0.0797673	0.0811491	0.0776949	0.0772860	0.0782807	0.0777895
quickSort	0.0302665	0.0293813	0.0280447	0.0284348	0.0300433	0.0303997	0.0278232	0.0289522	0.0280421	0.0280232
selectionSort	4.1073481	4.1029277	4.0807135	4.1329075	4.1677506	4.1426343	4.1079860	4.1103651	4.1249414	4.1437618