Algoritmos de ordenación

Reporte de los algoritmos vistos en clases

Paulina Aguirre García 1837503

Un algoritmo de ordenación nos permite ordenar los elementos de un arreglo o matriz, de la manera deseada. Para esto existen diferentes algoritmos, algunos con mayor o menor complejidad y eficacia.

Selection:

Este algoritmo consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo.

Rendimiento del algoritmo: Cada búsqueda requiere comparar todos los elementos no clasificados, de manera que el número de comparaciones C(n) no depende del orden de los términos, si no del número de términos; por lo que este algoritmo presenta un comportamiento constante independiente del orden de los datos. C(n)= n(n-1)/2. Luego la complejidad es del orden Θ(n2).

Implementación en Python:

```
1. def selection_sort(arr):
2.     nb = len(arr)
3.     for actual in range(0,nb):
4.         mas_pequeno = actual
5.         for j in range(actual+1,nb) :
6.             if arr[j] < arr[mas_pequeno] :
7.                 mas_pequeno = j
8.         if min is not actual :
9.             temp = arr[actual]
10.                 arr[actual] = arr[mas_pequeno]
11.                 arr[mas_pequeno] = temp
```

Bubble:

La ordenación de burbuja 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.

Rendimiento del algoritmo:

- Rendimiento en el caso óptimo: En el caso óptimo, con los datos ya ordenados, el algoritmo sólo efectuará n comparaciones. Por lo tanto la complejidad en el caso óptimo es en $\Theta(n)$.
- Rendimiento en el caso desfavorable: n el caso desfavorable, con los datos ordenados a la inversa, la complejidad es en $\Theta(n^2)$.
- Rendimiento en el caso medio: En el caso medio, la complejidad de este algoritmo es también en $\Theta(n^2)$

Implementación en Python:

```
1. def bubble_sort(arr):
2.     permutation = True
3.     iteración = 0
4.     while permutation == True:
5.         permutation = False
6.         iteración = iteración + 1
7.         for actual in range(0, len(arr) - iteración):
8.             if arr[actual] > arr[actual + 1]:
9.                 permutation = True
10.                     # Intercambiamos los dos elementos
11.                     arr[actual], arr[actual + 1] = \
12.                     arr[actual + 1],arr[actual]
13.         return arr
```

Quicksort:

El ordenamiento rápido es un algoritmo creado por el científico británico en computación Tony Hoare y basado en la técnica de divide y vencerás. Este algoritmo es quizá el más eficiente.

El algoritmo consiste en:

1. Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
2. 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. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
3. 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.
4. 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.

Rendimiento del algoritmo: 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(n•log n). En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$.

Implementación en Python:

```
1. def quick_sort(vector):
2.     if not vector:
3.         return []
4.     else:
5.         pivote = vector[-1]
6.         menor = [x for x in vector      if x <  pivote]
7.         mas_grande = [x for x in vector[:-1] if x >=
   pivote]
8.         return quick_sort(menor) + [pivote] +
   quick_sort(mas_grande)
```

Insertion:

La idea de este algoritmo de ordenación consiste en ir insertando un elemento de la lista o un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada, el algoritmo ira comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada.

Rendimiento del algoritmo:

- Rendimiento en el caso óptimo: En el caso óptimo, con los datos ya ordenados, el algoritmo sólo efectuara *n* comparaciones. Por lo tanto la complejidad en el caso óptimo es en Θ(*n*).
- Rendimiento en el caso desfavorable: En el caso desfavorable, con los datos ordenados a la inversa, se necesita realizar *(n-1) + (n-2) + (n-3) .. + 1* comparaciones e intercambios, o (n2-n) / 2. Por lo tanto la complejidad es en Θ(n^2).
- Rendimiento en el caso medio: En el caso medio, la complejidad de este algoritmo es también en Θ(n^2).

Implementación en Python:

```
1. def Insertion_sort(arr):
2.     for i in range(1,len(arr)):
3.         actual = arr[i]
4.         j = i
5.         #Desplazamiento de los elementos de la matriz
6.         while j>0 and arr[j-1]>actual:
7.             arr[j]=arr[j-1]
8.             j = j-1
9.         #insertar el elemento en su lugar
10.             arr[j]=actual
```

El rendimiento de cada uno de estos algoritmos varía según el arreglo a ordenar, es por eso que haremos una prueba con un arreglo aleatorio para cada algoritmo de ordenamiento.

La prueba consiste en:

Teniendo el código de cada algoritmo

1. Hacer un código para crear un arreglo aleatorio.
2. Copiar ese arreglo cuatro veces (uno para cada algoritmo).
3. En cada algoritmo obtener el número de operaciones que realiza para el acomodo del arreglo.
4. Mostrar las operaciones en manera de comparación.
5. Una gráfica de los resultados obtenidos.

Código:

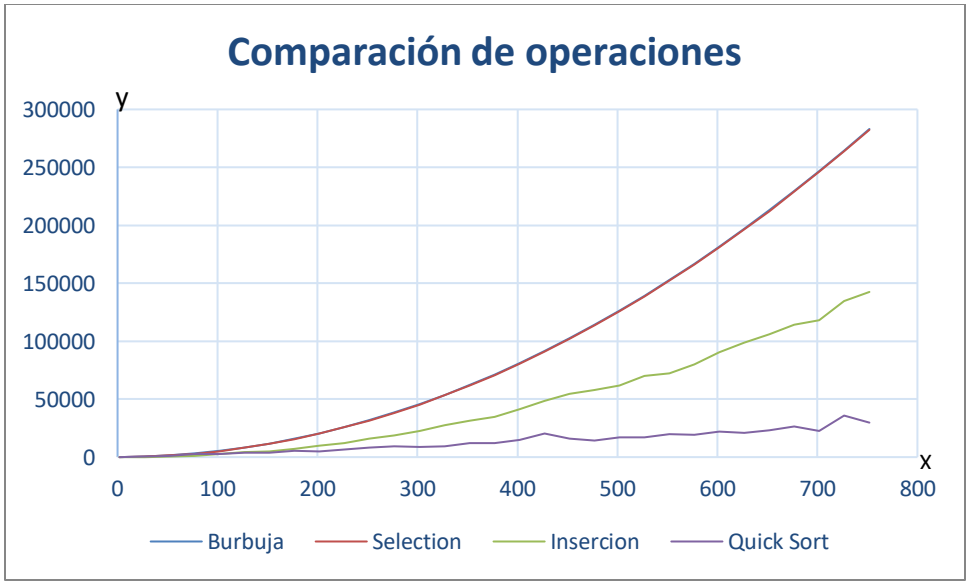
https://github.com/PaulinaAguirre/1837503MC/blob/master/Codigo_graficas.py

Tabla de comparación:

Longitud	Burbuja	Selection	Insercion	Quick Sort
2	3	1	1	10
27	378	351	194	386
52	1378	1326	747	1162
77	3003	2926	1304	2666
102	5253	5151	2757	2523
127	8128	8001	4469	3603
152	11628	11476	5142	3874
177	15753	15576	7004	5728
202	20503	20301	9882	5137
227	25878	25651	12123	6697
252	31878	31626	15941	8156
277	38503	38226	18750	9377
302	45753	45451	22483	8936
327	53628	53301	27720	9526
352	62128	61776	31362	12344
377	71253	70876	34624	12223
402	81003	80601	41325	14618
427	91378	90951	48283	20120
452	102378	101926	54518	15790
477	114003	113526	58012	14375
502	126253	125751	61914	17202
527	139128	138601	70162	16800
552	152628	152076	72071	19945
577	166753	166176	80208	19197
602	181503	180901	90686	21886
627	196878	196251	98834	20920
652	212878	212226	105920	22883
677	229503	228826	114242	26367
702	246753	246051	118011	22490
727	264628	263901	134705	35850
752	283128	282376	142515	29826
777	302253	301476	150051	26054
802	322003	321201	169160	28462
827	342378	341551	165289	28928
852	363378	362526	187437	33894

877	385003	384126	195667	33911
902	407253	406351	200817	36037
927	430128	429201	219467	37697
952	453628	452676	221528	35704
977	477753	476776	240132	36782
1002	502503	501501	250016	37136

Grafica obtenida:



Donde el eje x es la longitud del arreglo
y el eje y el número de operaciones.

De la gráfica podemos observar que algoritmos hacen más operaciones cuando el arreglo es más grande.

Para esta prueba se usó un arreglo de 1002 elementos.

Se podría concluir para este caso (el arreglo aleatorio) que Quick Sort es el algoritmo de ordenación que más conviene ya que hace menos operaciones inclusive con arreglos de más elementos.

De la gráfica también es posible observar que algunos de los ordenamientos tienen un crecimiento exponencial n^2. (Como Burbuja y Selection)