Universidad de Granada

Algoritmos de Ordenación Básicos

Algorítmica

Lukas Häring García 2º D

Tabla de contenidos

Inserción

0.1 Código

El algoritmo has sido sacado de:
https://www.geeksforgeeks.org/insertion-sort/

```
1  void insertionSort(int arr[], int n)
2  {
3    int i, key, j;
4    for (i = 1; i < n; i++){ <-- 1
5      key = arr[i];
6      j = i-1;
7
8      /* Move elements of arr[0..i-1], that are
9      greater than key, to one position ahead
10     of their current position */
11     while (j >= 0 && arr[j] > key){ <-- 2
12       arr[j+1] = arr[j];
13       j = j-1;
14     }
15     arr[j+1] = key;
16   }
17 }
```

0.2 Eficiencia

0.2.1 Teórica

En el código de la página anterior se han señalado las dos líneas más importantes, las demás pueden considerarse un orden de eficiencia O(1), veamos según cada punto, que orden tendrá:

1. En este caso es un bucle *for*, este será recorrido siempre, por lo que su eficiencia es **O(n)**.

2. El bucle while está anidado al anterior, por lo que aquí dependerá de las condiciones que hacen que entre o no, vamos a diferencias dos casos.

   (a) **El peor caso**. El vector se encuentra ordenado de forma inverso al que se quiere ordenar, provocando que cada iteración deba buscar en todo el tamaño menos 1, por lo que la eficiencia es $O(n^2)$, ya que es anidado.

   (b) **El mejor caso**. Ahora bien, si el vector está ordenado, entonces nunca entrará en el bucle, por lo tanto, es como si no existiera y su eficiencia será $\Omega(n)$.

0.2.2 Empírica

Realizando el algoritmo de ordenación por inserción desde 0 dando pasos de 100 hasta 10000 elementos y realizando un ajuste con gnuplot, calculando las variables ocultas, obtenemos una eficiencias de:
Para el peor caso:

$$f(n) = 8.03432 \cdot 10^{-11} \cdot n^2 - 2.50858 \cdot 10^{-9} \cdot n + 1.72744 \cdot 10^{-5} \in O(n^2)$$

Para el mejor caso:

$$g(n) = 7.85323 \cdot 10^{-10} \cdot n - 3.02854 \cdot 10^{-7} \in \Omega(n)$$

0.3 Gráficas

0.3.1 Teórica

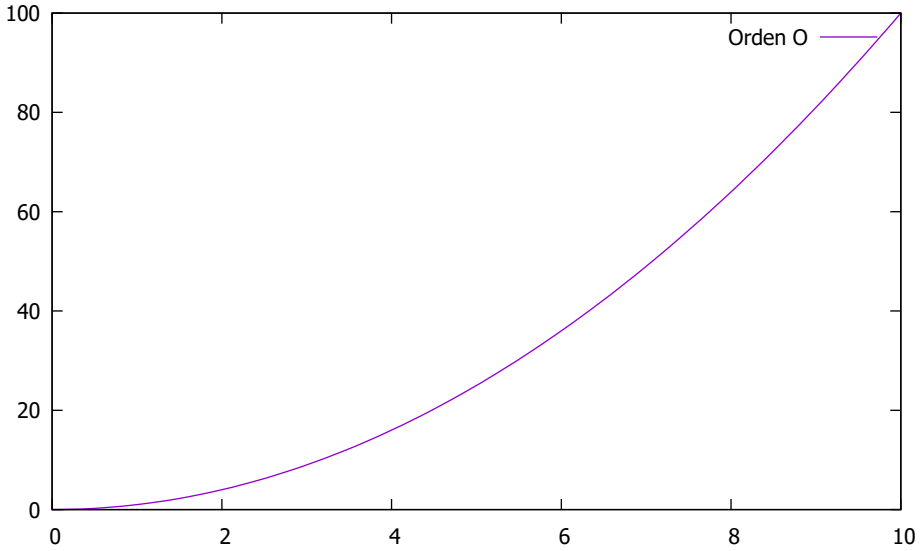


Figure 1: Orden de eficiencia teórico de big O.

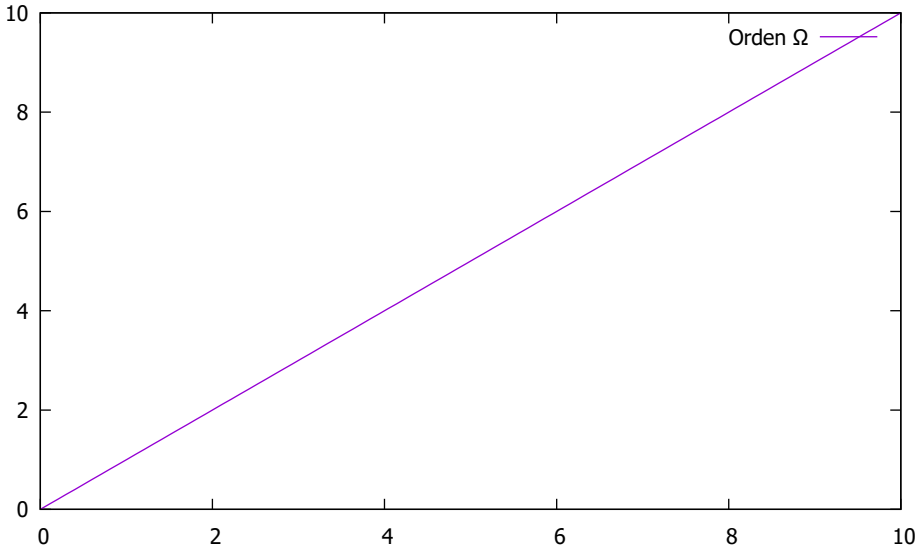


Figure 2: Orden de eficiencia teórico de Ω.

0.3.2 Empírica

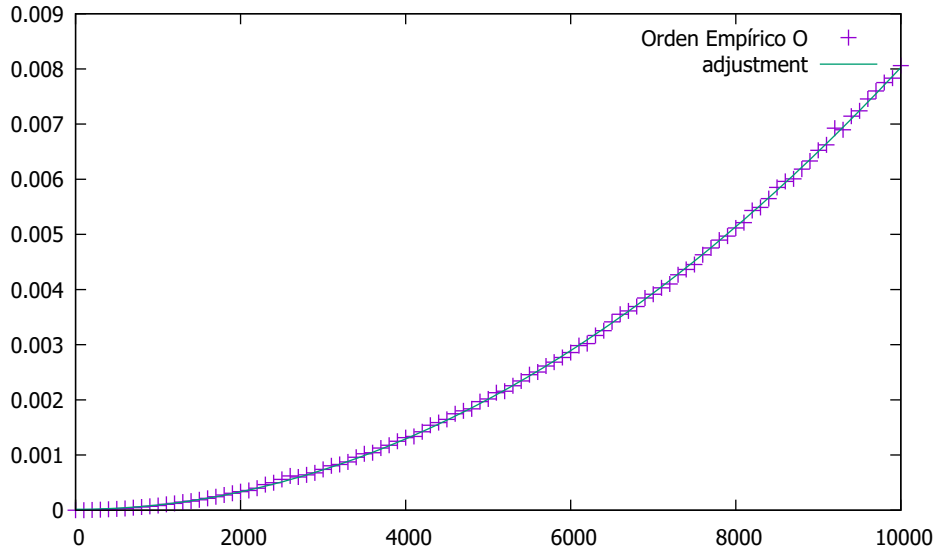


Figure 3: Orden de eficiencia empírico de big O.

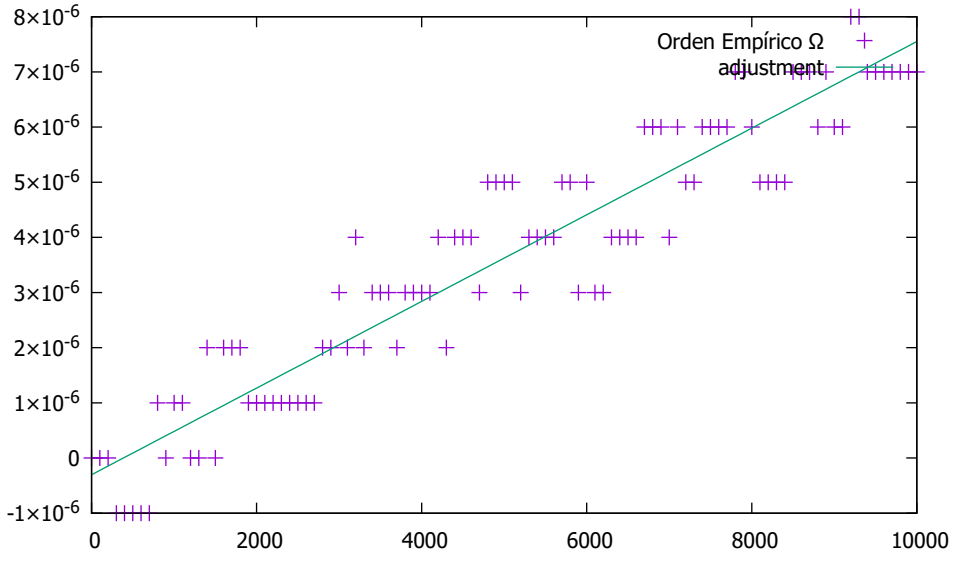


Figure 4: Orden de eficiencia empírico de Ω.

Selección

1.1 Código

El algoritmo has sido sacado de:
https://www.geeksforgeeks.org/selection-sort/

```
1  void swap(int *xp, int *yp){
2    int temp = *xp;
3    *xp = *yp;
4    *yp = temp;
5  }
6
7  void selectionSort(int arr[], int n){
8    int i, j, min_idx;
9
10   // One by one move boundary of unsorted subarray
11   for (i = 0; i < n-1; i++){ <-- 1
12     // Find the minimum element in unsorted array
13     min_idx = i;
14     for (j = i+1; j < n; j++) <-- 2
15       if (arr[j] < arr[min_idx])
16         min_idx = j;
17
18     // Swap the found minimum element with the first element
19     swap(&arr[min_idx], &arr[i]);
20   }
21 }
```

1.2 Eficiencia

1.2.1 Teórica

Su eficiencia es muy trivial, asumiendo que todo lo que no son "punteros" tienen eficiencia 1, viendo dichos punteros, observamos que son dos bucles anidados donde no hay restricción, por lo que la eficiencia es claramente $\Theta(n^2)$

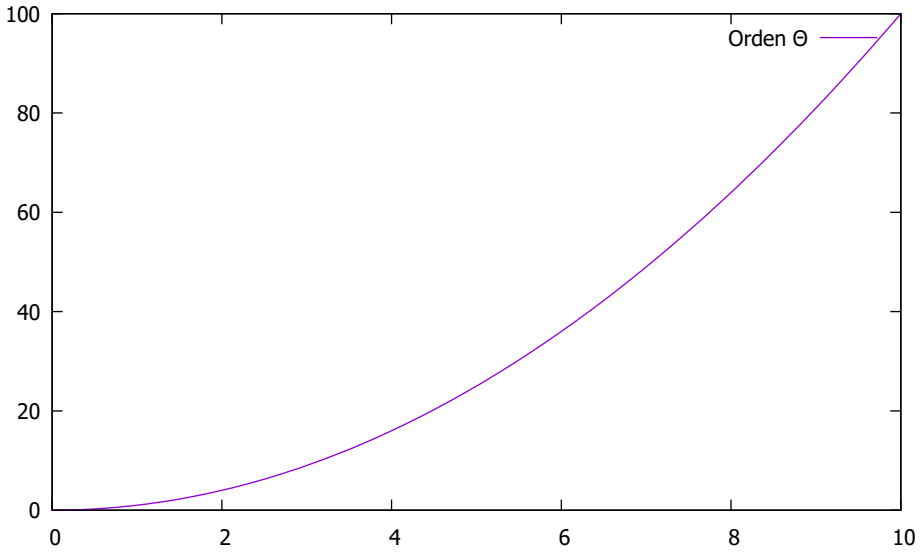
1.2.2 Empírica

Realizando el algoritmo de ordenación por selección desde 0 dando pasos de 100 hasta 10000 elementos y realizando un ajuste con gnuplot, calculando las variables ocultas, obtenemos una eficiencias de:

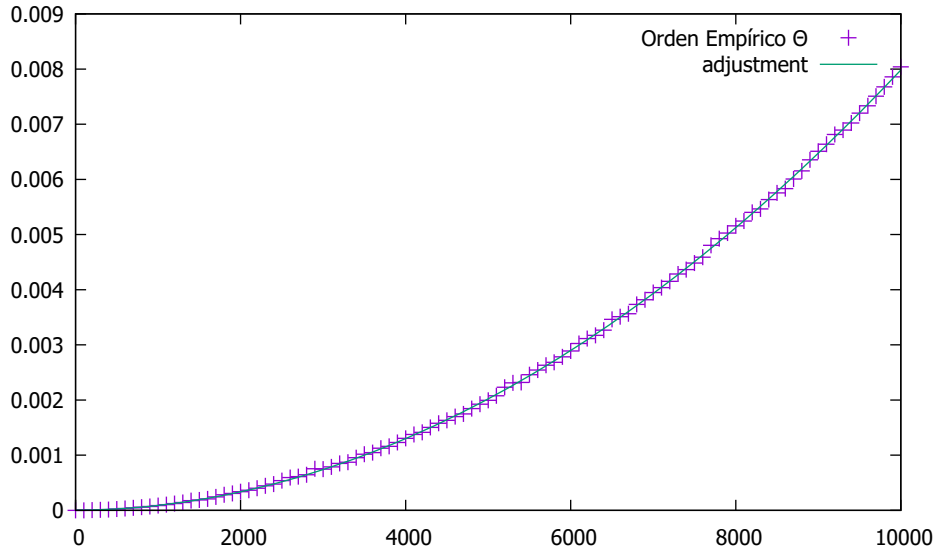
$$f(n) = 7.88641 \cdot 10^{-11} \cdot n^2 + 9.44417 \cdot 10^{-9} \cdot n + 2.31858 \cdot 10^{-6} \in \Theta(n^2)$$

1.3 Gráficas

1.3.1 Teórica



1.3.2 Empírica



Burbuja

2.1 Código

El algoritmo has sido sacado de:
https://www.geeksforgeeks.org/bubble-sort/

```
1  void swap(int *xp, int *yp){
2    int temp = *xp;
3    *xp = *yp;
4    *yp = temp;
5  }
6
7  // An optimized version of Bubble Sort
8  void bubbleSort(int arr[], int n){
9    int i, j;
10   bool swapped;
11   for (i = 0; i < n-1; i++){ <-- 1
12     swapped = false;
13     for (j = 0; j < n-i-1; j++){ <-- 2
14       if (arr[j] > arr[j+1]){
15         swap(&arr[j], &arr[j+1]);
16         swapped = true;
17       }
18     }
19
20     // IF no two elements were swapped by inner loop, then break
21     if (swapped == false){ <-- 3
22       break;
23     }
24   }
25 }
```

2.2 Eficiencia

2.2.1 Teórica

Este algoritmo es más complejo de ver su eficiencia, vamos analizando cada uno de los punteros:

1. Este bucle es ejecutado según lo que ocurra en 3, por lo que si en el tercer puntero se entra, este bucle se romperá, veamos los casos:
   (a) **El peor caso**. Que nunca pase la condición, es decir que al menos en su iteración del puntero dos haya realizado un swap, por tanto la eficiencia es $O(n^2)$.
   (b) **El mejor caso**. Si no ha realizado ningún "swap" en el bucle del puntero 2, este romperá el bucle del puntero 1, esto ocurre cuando el vector ya está ordenado por lo que no necesitaría realizar ningún intercambio, por lo que la eficiencia es $\Omega(n)$.
   (c) Es siempre ejecuta, su eficiencia es $O(n)$.
   (d) Si ocurre, entonces el bucle principal se rompe y se habrá ejecutado el del puntero 2 (razón por la eficiencia en el mejor caso).

2.2.2 Empírica

Realizando el algoritmo de ordenación por burbuja desde 0 dando pasos de 100 hasta 10000 elementos y realizando un ajuste con gnuplot, calculando las variables ocultas, obtenemos una eficiencias de:
Para el peor caso:

$$f(n) = 1.19523 \cdot 10^{-9} \cdot n^2 - 2.64497 \cdot 10^{-6} \cdot n + 0.00183111 \in O(n^2)$$

Para el mejor caso:

$$g(n) = 4.58241 \cdot 10^{-10} \cdot n - 1.6424 \cdot 10^{-7} \in \Omega(n)$$

2.3 Gráficas

2.3.1 Teórica

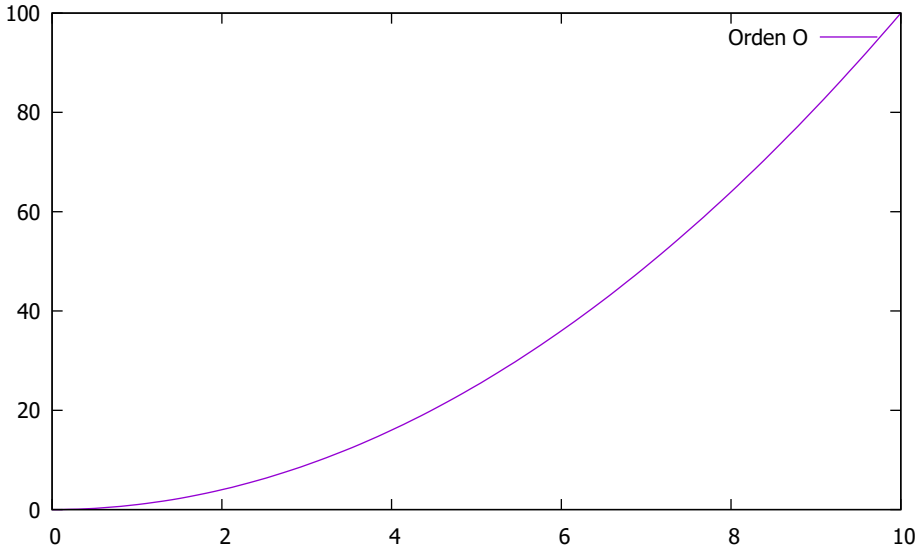


Figure 2.5: Orden de eficiencia teórico de big O.

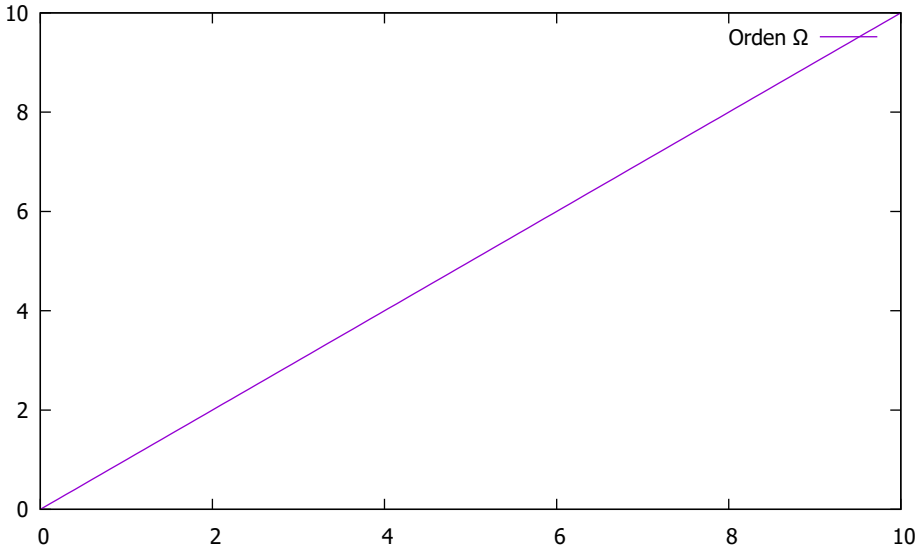


Figure 2.6: Orden de eficiencia teórico de Ω.

2.3.2 Empírica

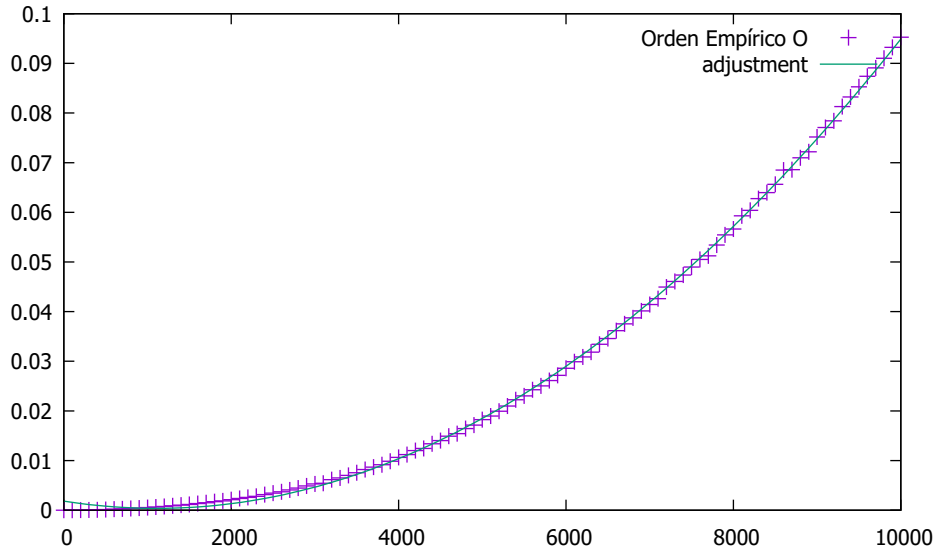


Figure 2.7: Orden de eficiencia empírico de big O.

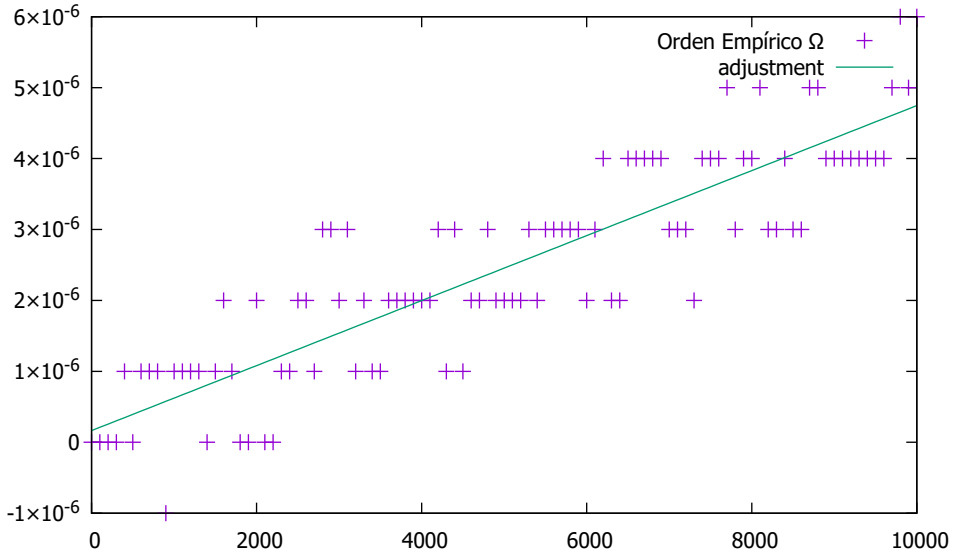


Figure 2.8: Orden de eficiencia empírico de Ω.

Especificaciones

1. Windows 10.0.14393
2. Procesador Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz, 3504 Mhz
3. 6 procesadores principales.
4. 12 procesadores lógicos.
5. Memoria física instalada (RAM) 8,00 GB x 2
6. Compilador MinGW.