MEMORIA PRÁCTICA 1: CÁLCULO DE LA EFICIENCIA DE ALGORITMOS

David Bacas Posadas
Julián Carrión Tovar
Manuel Marín Rodríguez
Alicia Ruiz Gómez
Raúl Martínez Bustos

UNIVERSIDAD DE GRANADA

Índice

Objetivos:

- Estudiar diferentes algoritmos de ordenación para analizar la eficiencia de los mismos en función del tamaño de los datos.
- Para ello, es necesario calcular y analizar la eficiencia teórica , práctica e híbrida de algoritmos recursivos e iterativos

Procedimiento:

Eficiencia teórica:

Analizaremos los siguientes algoritmos línea a línea siguiendo las pautas que se han especificado en el guión de prácticas.

Eficiencia Práctica:

Para calcular la eficiencia práctica , ejecutaremos el algoritmo dándole valores desde el 1000 hasta el 20000.
En la terminal de nuestro PC ejecutaremos la siguiente línea:
./algoritmo <**fich_algoritmo**> <**semilla**> <**tam1**> <**tam2**> **. . .** <**tam n-1**> < **tam n**>
De manera que introduciremos como argumentos:

- **./algoritmo:** Nombre del ejecutable que hemos conseguido a partir de nuestro <algoritmo.cpp>.
- <**fich_algoritmo**>: El fichero donde se van a guardar los datos registrados sobre eficiencia.
- <**semilla**>**:** Número que ayudará a generar números pseudoaleatorios (en los ejemplos se usa 12345, nosotros también lo usaremos).
- <**tam1**> <**tam2**> **. . .** < **tam n**>: Diferentes n (tamaños) que queremos probar para nuestro algoritmo, del que se obtendrá **T**(**n**) en el <**fich_algoritmo**>**.**

Una vez tengamos los datos del ejercicio en el fichero de texto, la herramienta gnuplot nos permitirá obtener la gráfica que ilustre la ejecución del programa para así, poder comparar con mayor facilidad los diferentes tipos de eficiencia, e incluso los distintos algoritmos entre sí.

Eficiencia Híbrida

Si los datos obtenidos en la eficiencia teórica y en la práctica no se corresponden , debemos buscar un valor **K** que nos ayude a ajustar los valores de las funciones anteriores de manera que nos permita sacar conclusiones más parecidas.

La **K** se obtiene mediante la fórmula $K = Tiempo / f(n)$, siendo Tiempo los datos obtenidos en el experimento práctico y f(n) el resultado de insertar los datos en su orden de eficiencia esperado por la eficiencia teórica.

Con todos estos valores de **K**, usaremos la media aritmética para poder sacar la **K** promedio y así posteriormente usarla en la fórmula del tiempo tórico estimado , obteniendo la eficiencia híbrida

Tam.Caso (n)	***Práctica (tiempo en us)***	***f(n)***	***K= Tiempo/f(n)***	***Tiempo teórico estimado= K*f(n)***
1000				
2000				
3000				
4000				
…	…	…	…	…
20000				
			K promedio	

Conteo (Counting Sort)

Algoritmo

```
1  void countSort(int* arr, int n) {
2      // Encuentra el máximo valor en el array
3      int max = arr[0];
4      for (int i = 1; i < n; i++) {
5          if (arr[i] > max) {
6              max = arr[i];
7          }
8      }
9
10     // Crea un array de conteo con el tamaño del máximo valor + 1
11     int* count = new int[max + 1]();
12
13     // Incrementa el conteo de cada elemento en el array original
14     for (int i = 0; i < n; i++) {
15         count[arr[i]]++;
16     }
17
18     // Reasigna los valores en el array original según el conteo
19     int index = 0;
20     for (int i = 0; i <= max; i++) {
21         while (count[i] > 0) {
22             arr[index++] = i;
23             count[i]--;
24         }
25     }
26
27     delete[] count;
28 }
```

Eficiencia teórica

El algoritmo Counting Sort es un método de ordenación lineal que trabaja eficientemente para ordenar elementos. Su eficiencia la podemos medir de la siguiente manera:

Encontramos un máximo en el vector (líneas 3-8): Usamos un bucle for que recorre todo el vector una vez para encontrar el máximo valor. Dado que el bucle se ejecuta n-1 veces en el peor caso, siendo n el tamaño del vector, y que cada iteración del bucle involucra por un lado una comparación y, posteriormente si se cumple, una

asignación, siendo estas operaciones son O(1) en el peor caso. Por lo tanto, la eficiencia de este tramo es O(n).

Crear un vector auxiliar para el conteo (línea 11): Se asigna dinámicamente un nuevo vector de tamaño "máximo + 1" . Esto es una operación de eficiencia O(1).

Contar la frecuencia de cada elemento (líneas 14-16): Se recorre el vector original una vez y se incrementa el contador correspondiente en el vector de conteo. Dado que este bucle se ejecuta exactamente n veces y cada operación dentro del bucle es O(1), la complejidad de este tramo es O(n).

Reconstruir el vector original (líneas 19-24): Para esto, utilizaremos un for que recorre nuestro vector de conteo, por lo que el tamaño de nuestro bucle será max.

Dentro de nuestro for incluiremos un while que tendrá tantas iteraciones como 1 encontremos en nuestro array, que en el peor de los casos corresponderá con el max.

Dado que si el while se ejecutase max veces, el for anterior no va a ejecutar nada durante todas sus iteraciones, ya que el while no va a ejecutarse, la eficiencia en el peor de los casos que tendrá este tramo será O(max).

Para concluir con la eficiencia total del Counting Sort, realizaremos la suma de los distintos tramos, es decir O(n) + O(n) + O(max), que se puede simplificar en O(n+max).

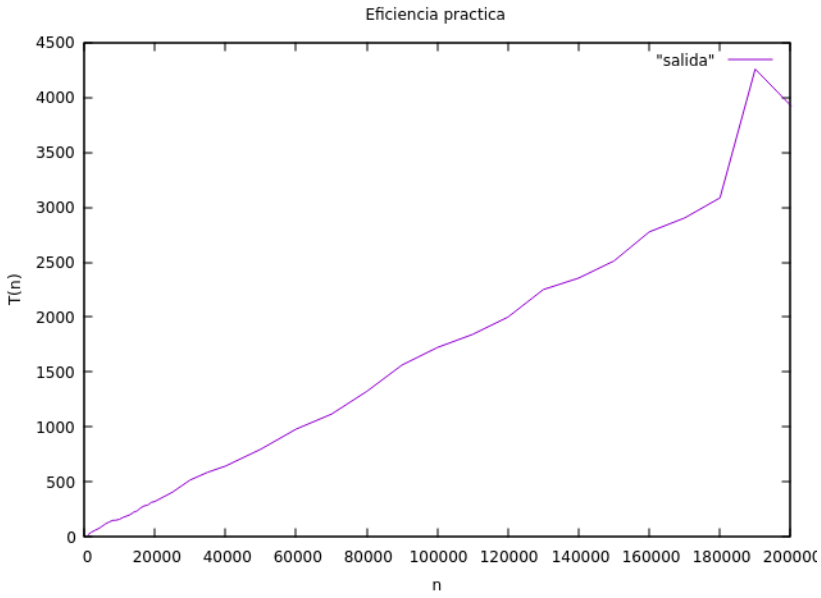
Por tanto, la eficiencia en el peor de los casos de nuestro algoritmo Counting Sort será O(n+k).

Eficiencia práctica

Al ejecutar el algoritmo dándole como tamaños desde el 1000 a 20000, los resultados son los siguientes:

1000	19	25000	408
2000	44	30000	519
3000	61	35000	590
4000	76	40000	646
5000	97	50000	799
6000	119	60000	982
7000	136	70000	1118
8000	151	80000	1325
9000	154	90000	1564
10000	162	100000	1724
11000	179	110000	1843
12000	191	120000	2000
13000	203	130000	2251
14000	226	140000	2355
15000	238	150000	2511
16000	265	160000	2775
17000	284	170000	2902
18000	291	180000	3085
19000	315	190000	4256
20000	323	200000	3925

A partir de esta tabla podemos obtener la siguiente gráfica:



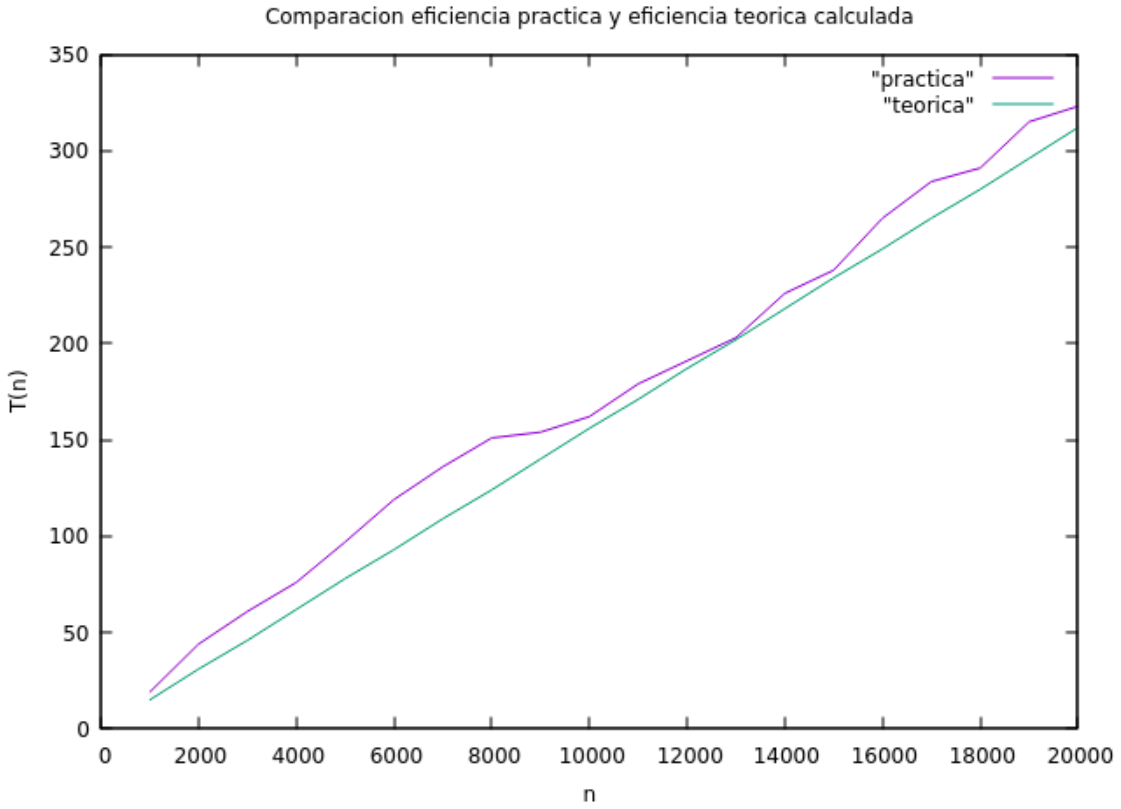
Eficiencia híbrida

A partir de lo obtenido sobre la eficiencia teórica y la eficiencia práctica, comprobaremos si ambas coinciden:

Tam.Caso (n)	*Práctica (tiempo en us)*	*f(n)*	*K= Tiempo/f(n)*	*Tiempo teórico estimado= K*f(n)*
1000	19	1000 + max	19/(1000+max)	15,6
2000	44	2000 + max	44/(2000+max)	31,2
3000	61	3000 + max	61/(3000+max)	46,8
4000	76	4000 + max	76/(4000+max)	62,4
5000	97	5000 + max	97/(5000+max)	78
6000	119	6000 + max	119/(6000+max)	93,6
7000	136	7000 + max	136/(7000+max)	109,2
8000	151	8000 + max	151/(8000+max)	124,8
9000	154	9000 + max	154/(9000+max)	140,4
10000	162	10000 + max	162/(10000+max)	156
11000	179	11000 + max	179/(11000+max)	171,6
12000	191	12000 + max	191/(12000+max)	187,2
13000	203	13000 + max	203/(13000+max)	202,8
14000	226	14000 + max	226/(14000+max)	218,4
15000	238	15000 + max	238/(15000+max)	234
16000	265	16000 + max	265/(16000+max)	249,6
17000	284	17000 + max	284/(17000+max)	265,2
18000	291	18000 + max	291/(18000+max)	280,8
19000	315	19000 + max	315/(19000+max)	296,4
20000	323	20000 + max	323/(20000+max)	312
			K promedio	***0.0156***

La K promedio ha sido calculada utilizando la pendiente de la recta de regresión hecha con excel a través de los datos prácticos obtenidos.
Puesto que la n toma valores que tienden a infinito, para calcular el tiempo teórico estimado hemos obviado el máximo, ya que terminaría siendo despreciable.

Así, para ver la comparación de las eficiencias teórica calculada y práctica, hemos realizado la siguiente gráfica:



Como se puede observar en la gráfica, ambos cálculos de la eficiencia nos han representado un algoritmo similar, por lo que podemos concluir con que la eficiencia del algoritmo Counting Sort es O(n+k).

Inserción (Insertion Sort)

Algoritmo

```
1 void insertionSort(int* arr, int n) {
2     for (int i = 1; i < n; i++) {
3         int key = arr[i];
4         int j = i - 1;
5 
6         // Mover los elementos del array que son mayores que key a una posición adelante
7         while (j >= 0 && arr[j] > key) {
8             arr[j + 1] = arr[j];
9             j--;
10        }
11 
12        arr[j + 1] = key;
13    }
14 }
```

Eficiencia teórica

El algoritmo resuelve el problema de ordenación de un subvector **arr**, el cuál contiene **n** componentes útiles.

Variable o variables de las que dependen el tamaño del caso: El tamaño del problema depende de un único parámetro **n** , que es la longitud del subvector a ordenar.

Primero analizaremos la parte más interna de la función , en este caso se corresponde con las líneas 8 y 9 de la función. La línea 8 se corresponde con la asignación de los valores del array. En la línea 9 se decrementa el valor j. Ambas operaciones se corresponden con el orden de eficiencia **O(1)**.

Dichas operaciones se encuentran dentro de un bucle "**while**", el cuál se ejecuta i - 1 veces y abarca las líneas 7-10. Por tanto, su eficiencia es del orden **O (1).**

Podemos observar que todo el código de la función se encuentra dentro de un bucle "**for**", incluyendo el bucle que hemos analizado anteriormente, veamos qué sucede en las demás líneas de código.

Las líneas 3,4 y 12 realizan diferentes asignaciones:

- En la línea 3 se le asigna a la variable de tipo entero key la posición i-ésima del array.
- En la línea 4, a j se le asigna el valor i-1.
- En la línea 12 , al elemento j+1 del vector se le asigna el valor de key.
-

Todas estas asignaciones son simples por lo que su eficiencia es **O(1).**

Por la regla de la suma podemos decir que la eficiencia del cuerpo del bucle es **O(n),** puesto que es la mayor de todas las que hasta ahora se nos han presentado.

El bucle exterior es un bucle "**for**" clásico que va desde 1 hasta **n** , por lo que se ejecuta **n** veces y su eficiencia es del orden **O(n)**.

Como ya hemos comentado, la eficiencia del cuerpo del bucle es O(n) y la del bucle también es **O(n),** aplicando la regla del producto, es fácil concluir que la eficiencia del algoritmo es **O**(n^2)

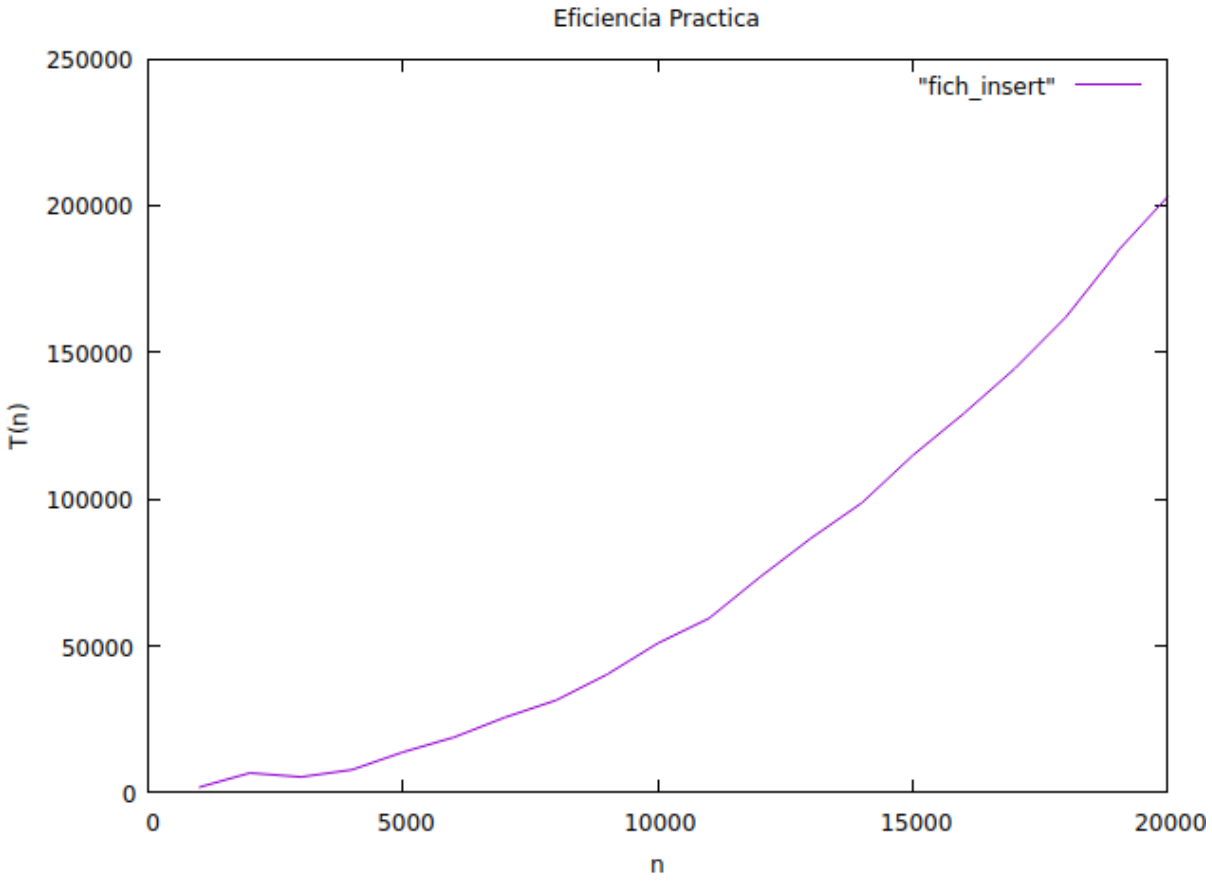
Eficiencia práctica

Al ejecutar el algoritmo dándole valores desde el 1000 a 20000, los resultados son los siguientes:

Tam.Caso	*Práctica (tiempo en us)*
1000	1906
2000	6705
3000	5338
4000	7832
5000	13802
6000	18899

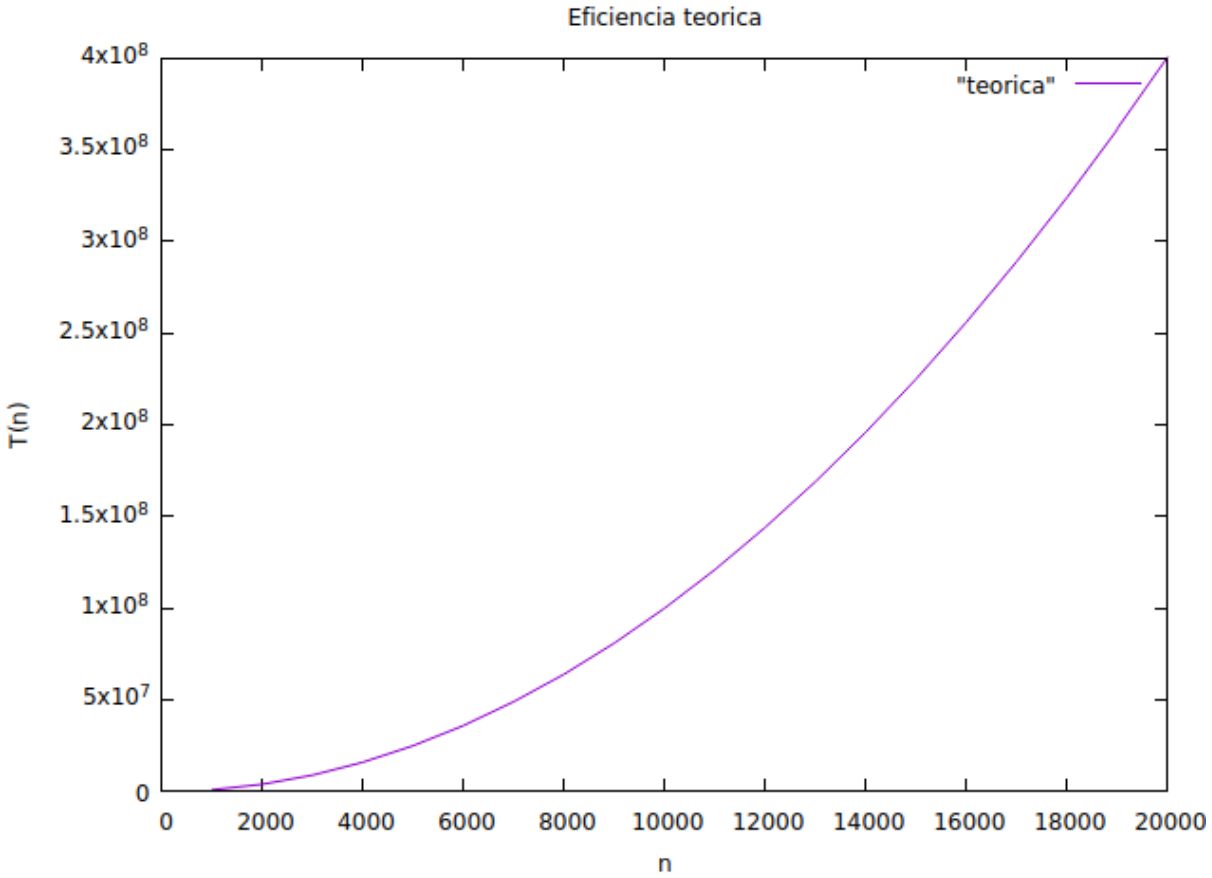
7000	25634
8000	31494
9000	40248
10000	50945
11000	59369
12000	73393
13000	86660
14000	97722
15000	114907
16000	129188
17000	144489
18000	161999
19000	183822
20000	202701

Lo cual nos deja la siguiente gráfica generada con gnuplot:



Eficiencia híbrida

Somos conscientes de que la eficiencia práctica y la teórica no se corresponden, simplemente con sus gráficas podemos ver que:

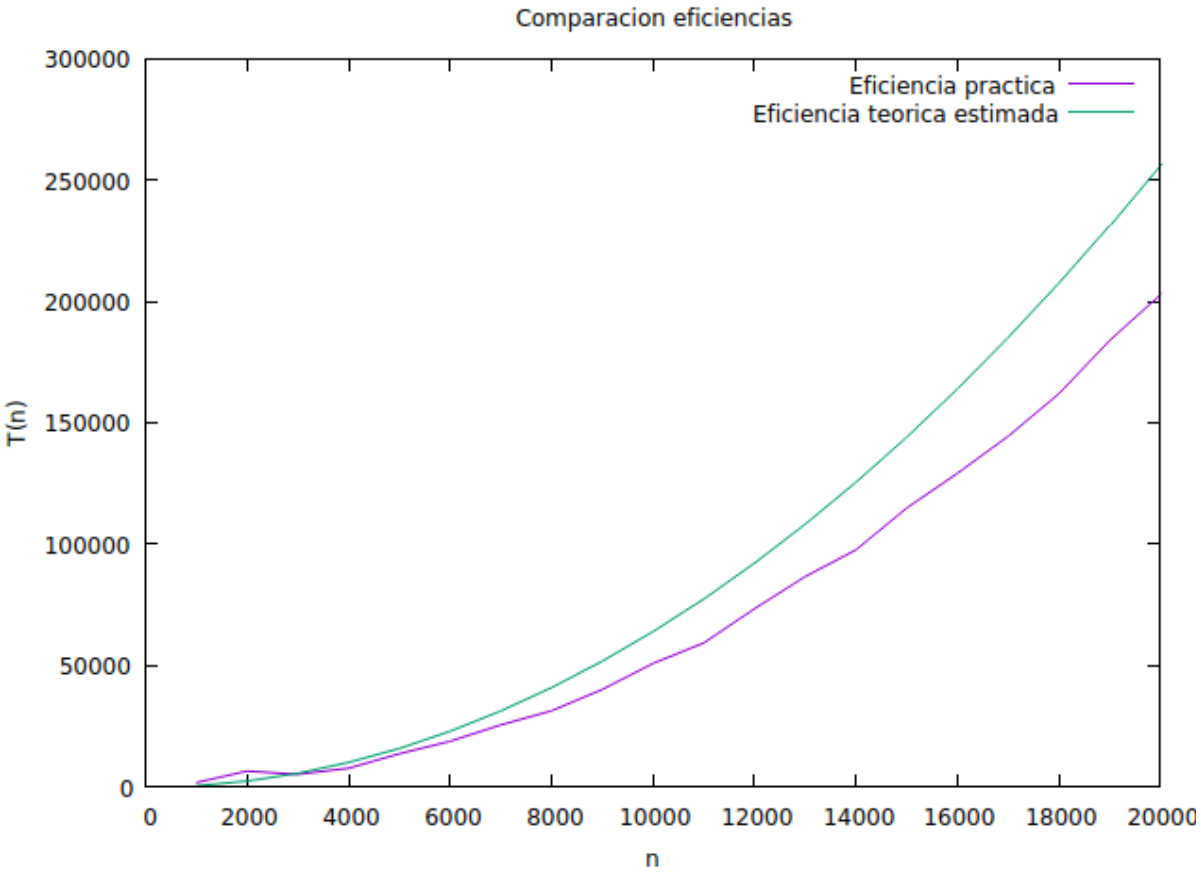


Sin embargo, hagamos los cálculos para poder realizar la comparación entre la eficiencia práctica y la teórica estimada, viendo si esta variación entre ambas es realmente notable o no. Calculamos los diferentes valores necesarios para nuestra estimación:

Tam.Caso (n)	*Práctica (tiempo en us)*	*f(n)*	*K= Tiempo/f(n)*	*Tiempo teórico estimado= K*f(n)*
1000	1906	1000000	0,001906	640,3215
2000	6705	4000000	0,00167625	2561,286
3000	5338	9000000	0,0005931111111	5762,8935
4000	7832	16000000	0,0004895	10245,144
5000	13802	25000000	0,00055208	16008,0375
6000	18899	36000000	0,0005249722222	23051,574
7000	25634	49000000	0,0005231428571	31375,7535
8000	31494	64000000	0,00049209375	40980,576

9000	40248	81000000	0,0004968888889	51866,0415
10000	50945	100000000	0,00050945	64032,15
11000	59369	121000000	0,0004906528926	77478,90151
12000	73393	144000000	0,0005096736111	92206,29601
13000	86660	169000000	0,0005127810651	108214,3335
14000	97722	196000000	0,0004985816327	125503,014
15000	114907	225000000	0,0005106977778	144072,3375
16000	129188	256000000	0,000504640625	163922,304
17000	144489	289000000	0,0004999619377	185052,9135
18000	161999	324000000	0,0004999969136	207464,166
19000	183822	361000000	0,0005092022161	231156,0615
20000	202701	400000000	0,0005067525	256128,6
			K promedio	0,000640321594

Una vez calculados, mediante gnuplot representamos gráficamente ambas funciones en una sola gráfica:



Como se puede comprobar hay un poco de variación entre la eficiencia práctica y teórica estimada, pies vemos que para que un mismo n, la teórica tarda más tiempo, mientras que la práctica tarda menos. Por lo tanto, nuestra estimación sobre la eficiencia teórica es más lenta que la velocidad real del algoritmo en la práctica.

Sin embargo, no hay que perder de vista lo crucial aquí, y es que aunque haya un ligero desplazamiento entre ambas, se puede ver como perfectamente ambas describen la gráfica de una parábola, confirmando así que nuestras consideraciones y estimaciones eran ciertas, pues finalmente el algoritmo tiene y describe gráficamente una eficiencia $O(n^2)$.

Rápido (Quick Sort)

Algoritmo

```
1 void swap(int* a, int* b) {
2     int temp = *a;
3     *a = *b;
4     *b = temp;
5 }
6
7 int partition(int* arr, int low, int high) {
8     int pivot = arr[high]; // Tomar el último elemento como pivote
9     int i = (low - 1); // Índice del menor elemento
10
11    for (int j = low; j <= high - 1; j++) {
12        // Si el elemento actual es menor que el pivote
13        if (arr[j] < pivot) {
14            i++; // Incrementar el índice del menor elemento
15            swap(&arr[i], &arr[j]);
16        }
17    }
18    swap(&arr[i + 1], &arr[high]);
19    return (i + 1);
20 }
21
22 void quickSort(int* arr, int low, int high) {
23    if (low < high) {
24        int pi = partition(arr, low, high); // Índice de partición
25
26        // Ordenar los elementos antes y después de la partición
27        quickSort(arr, low, pi - 1);
28        quickSort(arr, pi + 1, high);
29    }
30 }
```

Eficiencia teórica

El algoritmo Quick Sort es un método de ordenación que trabaja eficientemente para ordenar elementos. Su eficiencia la podemos medir de la siguiente manera:
Para comenzar, nombraremos como T(n) al tiempo de ejecución que tarda el algoritmo “QuickSort” en resolver un problema de tamaño n.
Función auxiliar swap (líneas 1-5): Esta función se encarga de intercambiar 2 valores dentro de un array de enteros. Esta función contiene 3 asignaciones, es decir 3 operaciones elementales, por lo que podemos decir que su eficiencia es O(1).

Búsqueda de pivote (líneas 7-20): Tomamos como pivote inicial el último elemento de nuestro array, y lo mismo para el menor elemento. Ambas son funciones elementales (O(1)). Posteriormente, de las líneas 11-17 aparece el for que se encargará de ordenar los elementos en nuestro conjunto. En el caso de recorrer el array completo, este va a tener eficiencia O(n). Conforme vayamos realizando las distintas llamadas a la propia función QuickSort, partition ya no recorrerá el array entero, pero tendremos varias llamadas a esta función, que en conjunto, sí que lo harán, por lo que podemos simplificar la eficiencia de partition durante toda la ejecución de nuestra función como O(n).

Función QuickSort (líneas 22-29): Aquí encontramos una llamada a partition, que hemos concluido anteriormente que puede simplificarse como O(n). Posteriormente, encontramos dos llamadas recursivas a la propia función QuickSort. Ambas llamadas tendrán como parámetro una mitad del array, tomando como centro el pivote calculado anteriormente. Por esto, podemos decir que cada recurrencia tiene eficiencia O(n/2).

Así, nos quedaría una ecuación de recurrencia con la siguiente forma: T(n) = 2 * t(n/2) + n

De esta manera podríamos resolver la ecuación de recurrencia de la siguiente manera:

Tomaremos el cambio de variable $n = 2^m$. De esta forma, la ecuación se nos quedaría de la siguiente forma: $T(2^m) = 2T(2^{m-1}) + 2^m$. Ahora resolvemos está ecuación: $T(2^m) - 2T(2^{m-1}) = 2^m$

Resolvemos la parte homogénea:

- $T(2^m) - 2T(2^{m-1}) = 0$

- $x^m - 2x^{m-1} = 0$

- $(x-2)x^{m-1}=0$

De aquí obtenemos que la parte homogénea del polinomio característico vale :

- $P_H(x) = (x-2)$

Ahora resolvemos la parte no homogénea:

- $2^m = b^m * q(m)$

Al resolverla obtenemos que b=2 y q=1, siendo el grado del polinomio 0, por lo que el polinomio característico nos queda tal que:

- $P(x) = P_H(x)(x-b)^{d+1} = (x-2)^2$

Por tanto tenemos como única raíz 2, con multiplicidad 2.

- $T(2^m)=\sum_{i=1}^{r}\sum_{j=0}^{Mi-1} c_{ij} R_j^m m^j = c_{10}\,2^m + c_{11}2^m m$

Ahora, deshacemos el cambio de variable:

- $T(n)= c_{10}\,n + c_{11}\,n\log_2(n)$

Así pues, obteniendo el máximo de la ecuación anterior, finalmente obtenemos que el algoritmo de ordenación QuickSort tiene una eficiencia en el peor de los casos ***O(nlog(n)).***

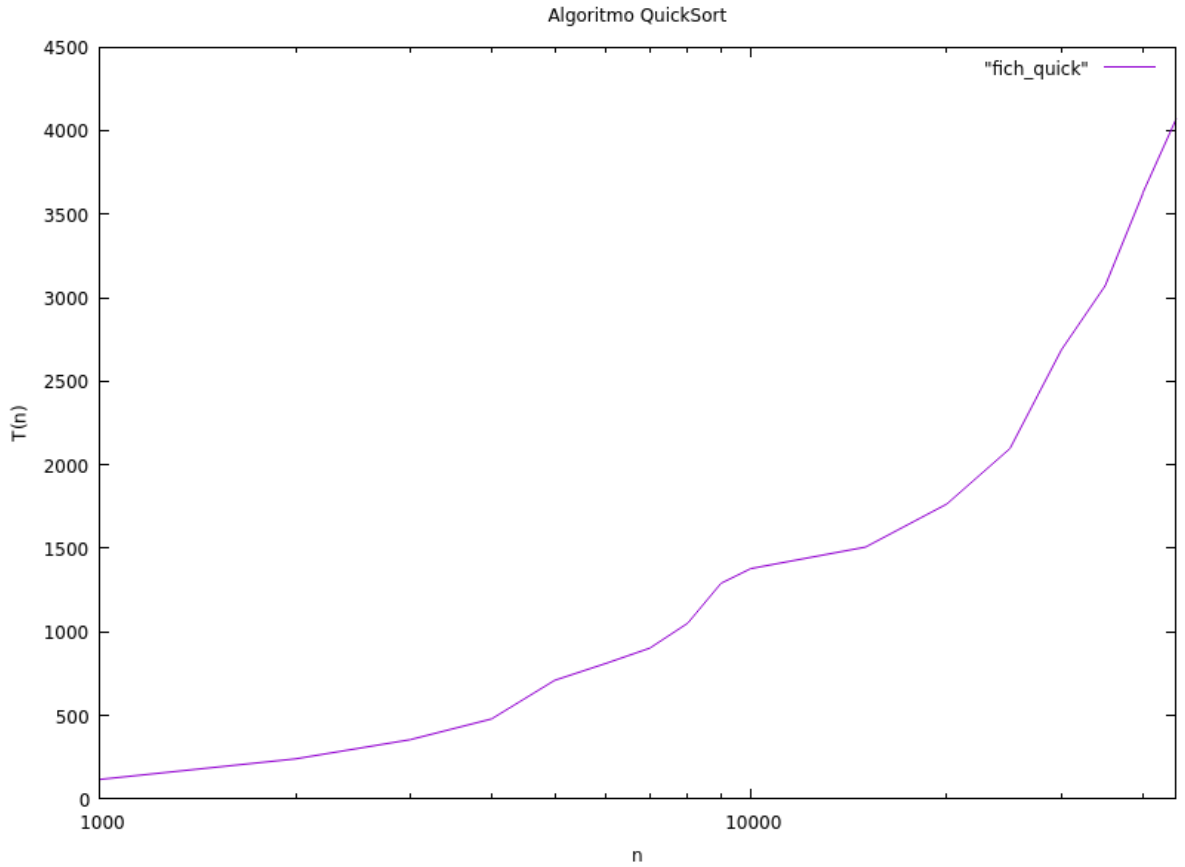
Eficiencia práctica

Al ejecutar el algoritmo con valores de n desde 1000 hasta 200000 obtenemos los siguientes resultados:

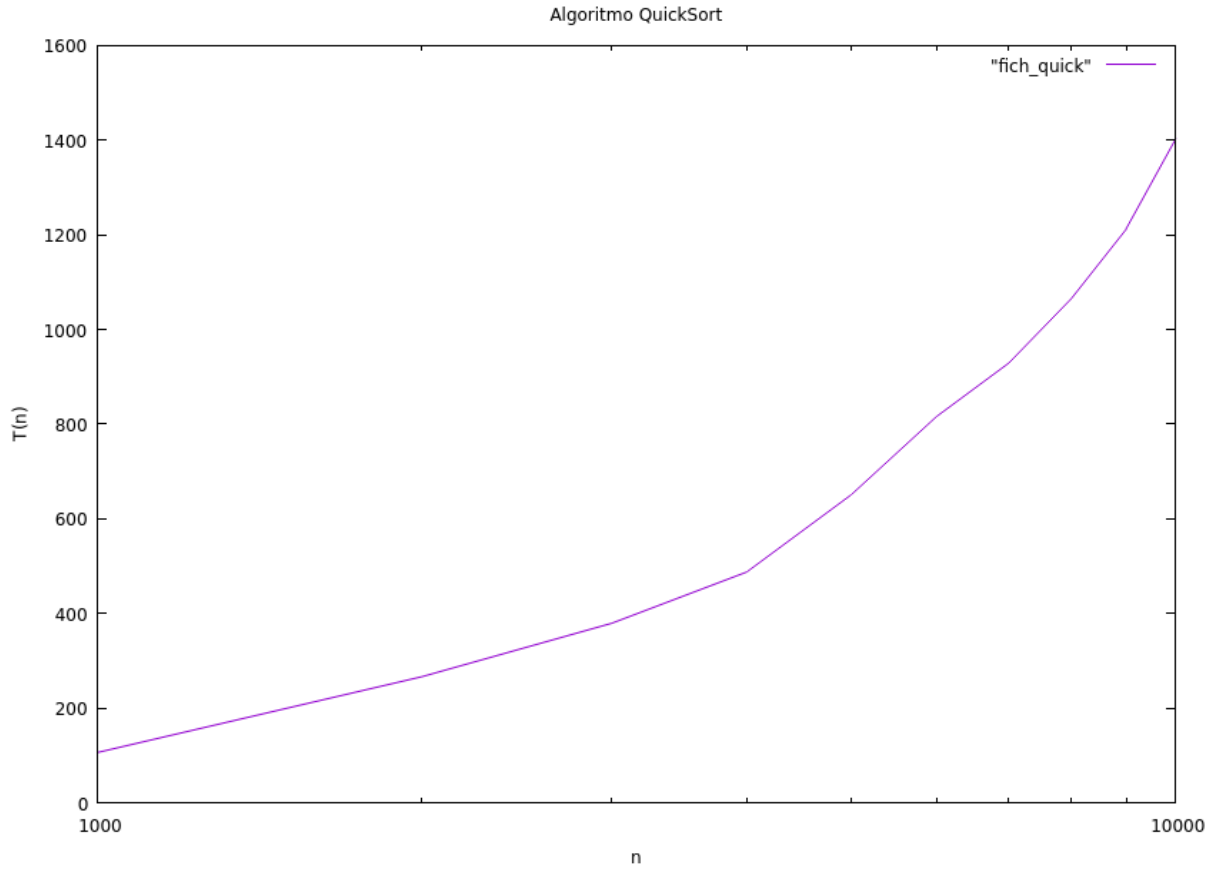
1000	117	50000	6379
2000	241	60000	5907
3000	355	70000	6897
4000	479	80000	8244
5000	710	90000	8804
6000	812	100000	10105

7000	903	110000	10788
8000	1052	120000	12140
9000	1291	130000	12830
10000	1379	140000	15396
15000	1507	150000	16882
20000	1765	160000	18442
25000	2097	170000	19319
30000	2688	180000	19748
35000	3071	190000	20752
40000	3623	200000	21563
45000	4065	x	

Y a partir de estos datos podemos obtener la siguiente gráfica:



Otra prueba de ejecución más acertada:



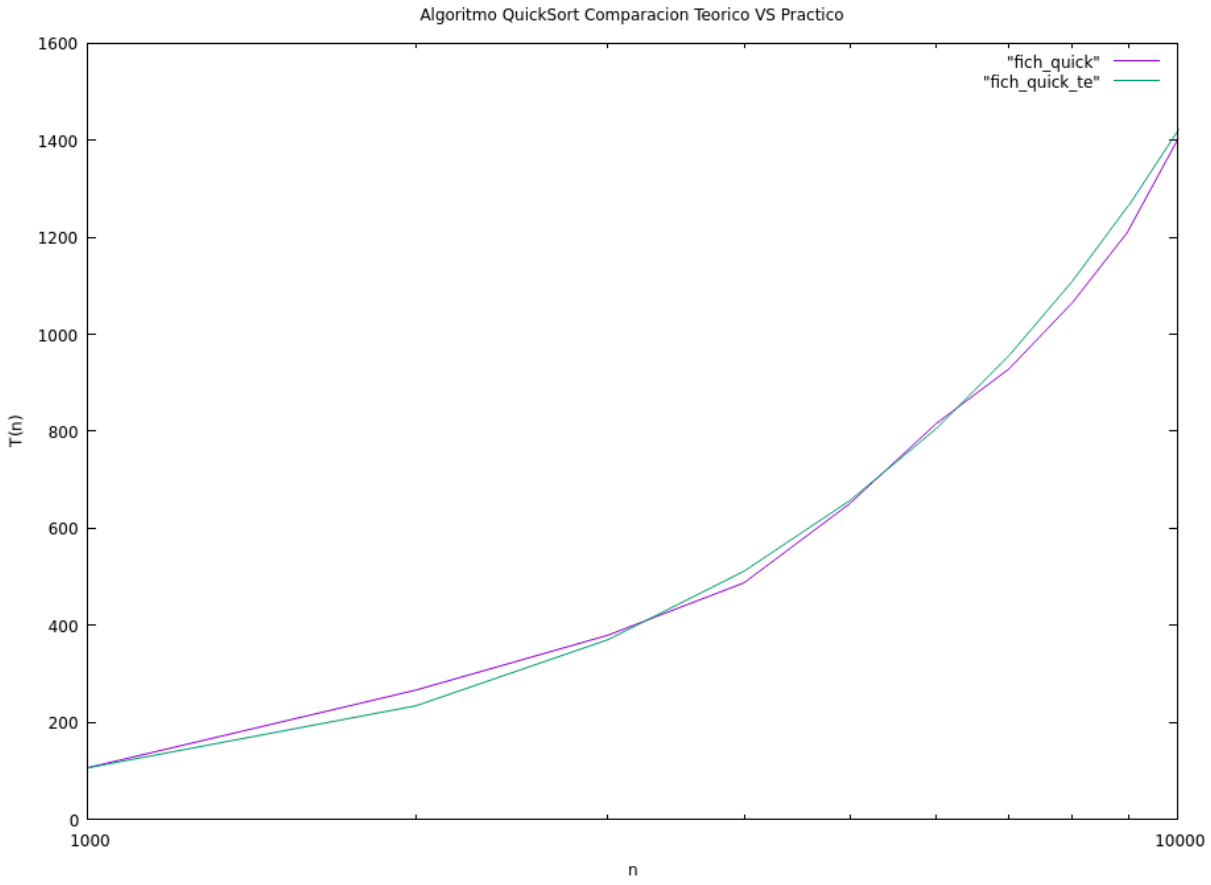
Eficiencia híbrida

A partir de lo obtenido tanto en la eficiencia teórica como en la práctica, comprobaremos si ambas coinciden:

Tam.Caso (n)	*Práctica (tiempo en us)*	*f(n)*	*K= Tiempo/f(n)*	*Tiempo teórico estimado= K*f(n)*
1000	107	6907,75	0.01548985	106.591802
2000	267	15201,80	0.01756371	234.57526
3000	380	24019,1	0.01582074	370.632861
4000	488	33176,2	0.01470934	511.933833
5000	651	42585,9	0.01528675	657.132613
6000	816	52197.1	0.01563305	805.440691

7000	929	61975,6	0.01498977	956.330334
8000	1065	71897,6	0.01481273	1109.43429
9000	1212	81944,8	0.01479044	1264.47018
10000	1401	92103,4	0.01521116	1421.22505
			K promedio	0.01543076

Así pues, vamos a comparar la eficiencia práctica y teórica en la siguiente gráfica



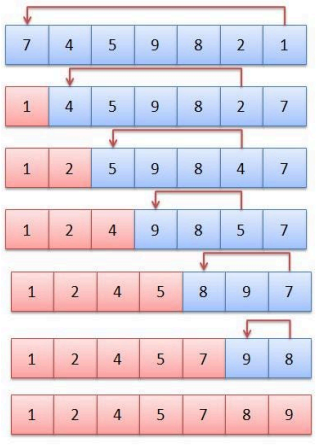
Selección (Selection Sort)

Algoritmo

```
1 void selectionSort(int* arr, int n) {
2     for (int i = 0; i < n - 1; i++) {
3         // Encontrar el índice del mínimo elemento en el subarray no ordenado
4         int min_idx = i;
5         for (int j = i + 1; j < n; j++) {
6             if (arr[j] < arr[min_idx]) {
7                 min_idx = j;
8             }
9         }
10
11        // Intercambiar el mínimo elemento con el primer elemento no ordenado
12        if (min_idx != i) {
13            int temp = arr[i];
14            arr[i] = arr[min_idx];
15            arr[min_idx] = temp;
16        }
17    }
18 }
```

Eficiencia teórica

Antes de empezar a estudiar el algoritmo en profundidad, primero veamos cómo se encuentra implementado este.



El "Selection sort" (u ordenación por selección) es un método de ordenación. La manera de la que funciona le da ese nombre, pues intenta encontrar el menor de los elementos desordenados del array e intercambiarlo con el que está en primera posición de los ordenados.

De esta manera, busca constantemente ese nuevo mínimo que queda en el array y lo pone al principio de los no ordenados.Así resuelve el problema de ordenación de un vector (o puntero) "arr", con un total de n componentes útiles.

Variable o variables de las que dependen el tamaño del caso: El tamaño del problema depende de un único parámetro, n, que es la longitud del vector o array a ordenar.

Vamos a comenzar por la parte más interna del código. La línea 13 contiene una asignación de variable, igualando min_idx a j, siendo esta una operación elemental, por lo tanto O(I).

Al encontrarse esta asignación (min_idx=j) dentro de un if, tenemos que analizar esta sentenecia condicional detenidamente. No existe un else acompañando a este if, así que suponemos que el peor caso para la función se dará cuando se cumpla esa condición. Asumimos que en el peor de los casos (array está ordenado de manera descendente), esta condición se cumplirá siempre (que el valor del array de la posición j (o i +1) sea menor que la de la posición min_idx (o i), guardando así como mínimo el valor de la posición j, la cual es la menor de ambas).

La evaluación de la condición se corresponde con una comparación "<" y dos accesos a componentes del array. Todas son operaciones elementales y orden de eficiencia O(I). Consecuentemente, el "if" tiene una eficiencia igual al orden de eficiencia de comprobar la condición + orden de eficiencia sentencias de dentro de la estructura. Vemos que que ambas son O(I)), por lo que aplicando la regla del máximo, la eficiencia entre las líneas 12-14 es O(I).

El bucle "for" de la línea 11 (el cuál abarca hasta la línea 15, incluyendo dentro de si mismo la estructura condicional tratada anteriormente [12-14]): Su inicialización es O(1) por ser una operación elemental (int j=i+1), al igual que la comprobación (j < n) y la actualización (j++).

En este bucle,

```
for (int j = i + 1; j < n; j++) {
    if (arr[j] < arr[min_idx]) {
        min_idx = j;
    }
}
```

Su orden de eficiencia será (n)*(O(comprobación) + O(actualización) + O(sentencias del bucle)) que, como todas son O(1), aplicando la regla del máximo la eficiencia es O(n). Como consecuencia, este bucle que va desde la linea 11 a la 15, tiene una eficiencia de O(n) [lineal].

A continuación, a la misma altura de nuestro "for" de la línea 11, nos encontramos sobre el bucle una declaración y asignación (int min_idx = i;), y debajo de este otro"if".

Al estar ambas al mismo nivel que nuestro bucle "for", vamos a analizar primero la declaración e inicialización de un tipo entero, para poder leer nuestro código preferentemente de arriba a abajo.

```
int min_idx = i;
```

Como se contempla, se trata de una declaración e inicialización de variable de tipo entero. (línea 10)

Por lo tanto, estamos ante una operación elemental, entonces esta operación es O(I).

Dentro del condicional if, podemos ver tres líneas de código, (19-21). Si nos fijamos, son: nuna declaración e inicialización de variable tipo entero, y las otras son asignaciones a valores del array de otros valores del mismo.Todas las operaciones de dentro del condicional son operaciones elementales, siendo la eficiencia O(I).

Al no haber un else para el"if", suponemos que el peor caso se dará si se cumple la condición (nuevo valor mínimo sea diferente al valor de i actual, (min_idx != i)), y que se dará siempre.

La evaluación de la condición se corresponde con la comparación "<" y dos accesos a valores enteros. Todas son operaciones elementales y tienen un orden de eficiencia O(1).

```
if (min_idx != i)
```

De este modo, el "if" completo tiene una eficiencia igual al orden de eficiencia de comprobar la condición + orden de eficiencia de las sentencias de dentro de la estructura.

Ambos son O(1), por lo que aplicando la regla del máximo, la eficiencia del algoritmo entre las líneas 87-92 (inclusive) es O(1).Proseguimos como hemos dicho de dentro hacia fuera. Al aplicar la regla del máximo de la eficiencia (9-22), vemos que la eficiencia teórica de este bloque será O(n), ya que es la eficiencia más alta que encontramos entre las tres estructuras que tiene dentro.

Por lo tanto, en el peor caso, el número total de comparaciones que realiza el algoritmo de selección es:

(n - 1) + **(n - 2)** + **(n - 3)** + **...** + **2** + **1** = **(n** * **(n - 1)) / 2**

Resulta curioso ver que se forma esta sumatoria, debemos aclarar que debido a esto el algoritmo llega a ser un $O(n^2)$ "más eficiente", porque a partir de una parte de la gráfica subirá cada vez más lento hacia arriba (a diferencia de los algoritmos $O(n^2)$ comunes), lo que nos dará una constante k más chica finalmente.

Esto da como resultado una complejidad cuadrática O(n^2) en el peor caso.

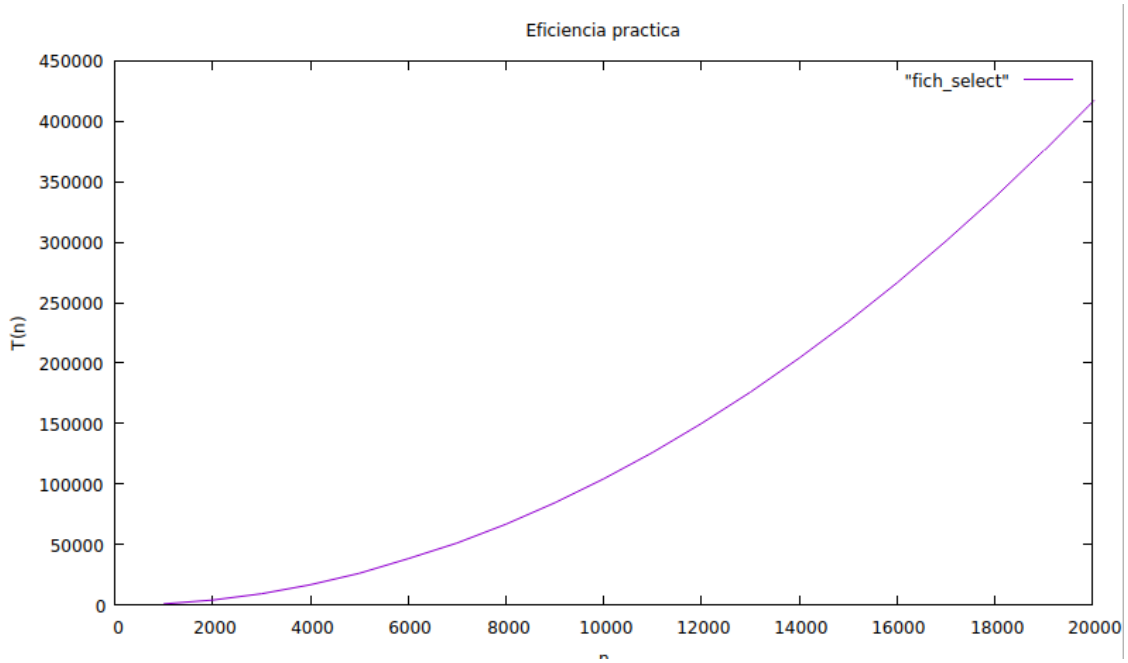
Por lo tanto, la eficiencia de nuestro algoritmo será de $O(n^2)$, ya que nos encontramos con que O(n) del "for" más exterior se multiplicará con O(n) de la estructura de código anterior, dando así la eficiencia de este algoritmo $O(n*n) = O(n^2)$.

Eficiencia práctica

Al ejecutar el algoritmo dándole tamaños desde el 1000 a 20000, los resultados son los siguientes:

Tam.Caso	***Práctica (tiempo en us)***
1000	1115
2000	4324
3000	8836
4000	11388
5000	17614
6000	25215
7000	34246
8000	44666
9000	56525
10000	69641
11000	84230
12000	100139
13000	117447
14000	136186
15000	156233
16000	177631
17000	200459
18000	224641
19000	250414
20000	277464

Lo cual nos deja con la siguiente gráfica de T(n) en función de n, generada con gnuplot:



Eficiencia híbrida

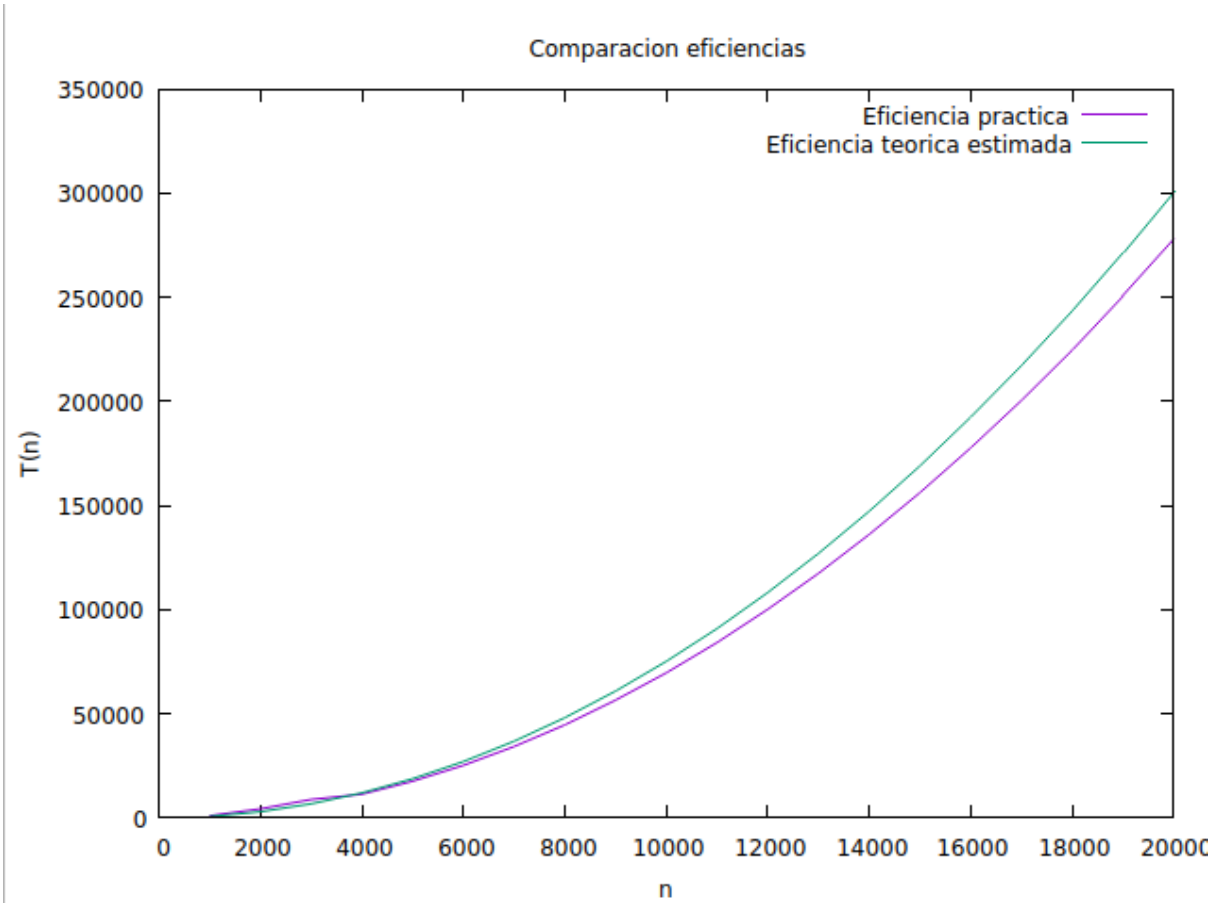
Afortunadamente, podemos ver como la eficiencia teórica y práctica de nuestro algoritmo son similares, ya que estimamos la teórica como $O(n^2)$, y la práctica ha descrito exactamente el trazado de una parábola. (cuadrática).

Veamos entonces ahora la comparación entre la eficiencia teórica estimada y práctica, basándonos en los cálculos adyacentes para poder estimarla:

Tam.Caso (n)	***Práctica (tiempo en us)***	***f(n)***	***K= Tiempo/f(n)***	***Tiempo teórico estimado= K*f(n)***
1000	1115	1000000	0,001115	751,469826751,469826
2000	4324	4000000	0,001081	3005,879313005,87931
3000	8836	9000000	0,00098178	6763,228446763,22844
4000	11388	16000000	0,00071175	12023,517212023,5172
5000	17614	25000000	0,00070456	18786,745718786,7457

6000	25215	36000000	0,00070042	27052,913827052,9138
7000	34246	49000000	0,0006989	36822,021536822,0215
8000	44666	64000000	0,00069791	48094,068948094,0689
9000	56525	81000000	0,00069784	60869,055960869,0559
10000	69641	100000000	0,00069641	75146,982675146,9826
11000	84230	121000000	0,00069612	90927,84990927,849
12000	100139	144000000	0,00069541	108211,655108211,655
13000	117447	169000000	0,00069495	126998,401126998,401
14000	136186	196000000	0,00069483	147288,086147288,086
15000	156233	225000000	0,00069437	169080,711169080,711
16000	177631	256000000	0,00069387	192376,276192376,276
17000	200459	289000000	0,00069363	217174,78217174,78
18000	224641	324000000	0,00069334	243476,224243476,224
19000	250414	361000000	0,00069367	271280,607271280,607
20000	277464	400000000	0,00069366	300587,931300587,931
			K promedio	0,000751470,00075147

Finalmente, nos disponemos a hacer nuestra gráfica de comparación de eficiencia práctica y eficiencia teórica estimada:



Observando esta gráfica, podemos confirmar que efectivamente la eficiencia práctica de nuestro algoritmo "Selection sort" o algoritmo por selección, coincide con la eficiencia teórica estimada que hemos obtenido, confirmando así que este algoritmo tiene una eficiencia de $O(n^2)$, siendo este uno de los mejores algoritmos de orden cuadrático que podemos usar para resolver el problema planteado.

Ordenamiento Shell (Shell Sort)

Algoritmo

```
1 void shellSort(int arr[], int n) {
2     for (int gap = n / 2; gap > 0; gap /= 2) {
3         for (int i = gap; i < n; i++) {
4             int temp = arr[i];
5             int j = i;
6             while (j >= gap && arr[j - gap] > temp) {
7                 arr[j] = arr[j - gap];
8                 j -= gap;
9             }
10            arr[j] = temp;
11        }
12    }
13 }
```

Eficiencia teórica

La implementación original de este mecanismo es **O(n^2)** en el peor caso (ya que viene del de inserción), pero con la última implementación puede llegar a tener un rendimiento de **O(n log^2 n)** aún estando en el peor caso que no llega al caso óptimo que sería **O(n log n)**. De aquí que analizar su tiempo de ejecución es bastante complicado.

El Shell sort está basado en dos observaciones: el ordenamiento por inserción es eficiente cuando la entrada está casi ordenada y es ineficiente en general debido a sus movimientos de valores de a uno. Shell sort mejora esto al comparar elementos separados por un espacio de varias posiciones, permitiendo que los elementos avancen hacia su posición esperada en pasos más grandes. Los pasos sobre los datos se hacen con espacios cada vez más pequeños, y el último paso es un simple ordenamiento por inserción. Para entonces, los datos del vector están casi ordenados.

El algoritmo se encarga de solventar el problema de ordenar un vector dado "**arr**", que a su vez tiene "**n**" componentes o números.
Variables de las que depende el tamaño.- En este algoritmo solo hay un único parámetro del que depende el problema, es "**n**" que es la longitud del vector que queremos ordenar.

Para analizarlo vamos a ir por pasos, empezaremos de dentro a fuera, es decir, por la parte interna para llegar a la más externa, no sin antes hacer una aclaración para simplificar la explicación, me refiero a dejar claro que en las líneas 4-5-7-8-10 contienen accesos a posiciones de un array (arr[j] = ...) y asignaciones de tipos básicos (**j** = **i, j** -= **gap**). Todas estas son operaciones elementales y por tanto son O(1). Esto lo usaremos más tarde.

Ahora nos centraremos en la parte más interna que nos lleva al **bucle** "**while**"**:**
Este bucle "while" se ejecuta para mover los elementos del arreglo hacia la derecha hasta que se encuentra la posición correctaon O(1) porque lo hemos demostrado antes, así que solo nos fijamos en el bu para insertar el elemento "temp". Sabemos que las sentencias del interior scle en sí.
En el peor caso, este bucle "while" se ejecutará en cada iteración del bucle externo para un valor de "i", cuando el elemento "temp" tenga que ser insertado al principio del subarreglo ordenado. En este caso, el bucle "while" podría tener que mover "i/gap" elementos hacia la derecha antes de encontrar la posición adecuada para insertar "temp".
En el mejor caso, cuando el subarreglo ordenado esté completamente ordenado y no sea necesario mover elementos, este bucle no se ejecutará en absoluto.
Así nos queda que la eficiencia varía según el orden y puede oscilar entre O(1) en el mejor caso y O(n/gap) en el peor caso. El factor "gap" también influye, ya que determina la cantidad de elementos que se deben mover en cada iteración.

Análisis de los **bucle "for" externo e interno:**
Empezaremos por el interno. Este itera sobre el arreglo desde la posición "gap" hasta la última posición (n - 1). Cada iteración de este bucle selecciona un elemento del arreglo (arr[i]) para ser insertado en su posición correcta dentro del subarreglo ordenado.
La eficiencia de estos bucles depende de la cantidad de iteraciones que realiza, en el peor caso, el bucle interno for realiza "n-gap" iteraciones. ahora podemos ver cómo en la actualización (cada iteración) del bucle externo, el número total de iteraciones

que se ejecuta el bucle interno es O(n*log n). Esto (la actualización del bucle) va a ser lo que cambie la eficiencia que teníamos hasta ahora.

Esto lo hemos determinado con los siguientes cálculos:

El bucle externo divide "gap" por 2 cada iteración, "gap" empieza con "b/2".

El bucle interno itera desde "gap" hasta "n" por lo que el bucle interno hace (n-gap) iteraciones.

Si nos vamos a la segunda iteración del bucle externo "gap" se vuelve a reducir a la mitad (n-gap)/2) de iteraciones.

Así continuamente ((n-gap)/4 …) hasta que "gap" es 1.

Para calcular el número total de iteraciones a lo largo de todas las iteraciones del bucle externo, sumamos todas las del interno en cada una de las iteraciones del externo:

Total de iteraciones del bucle interno = (n-gap) + (n-gap/2) + (n-gap/4) + ... + (n-1)

Esto se asimila a una serie geométrica:

Total … ≈ (n-n/2) + (n-n/4) + (n-n/8) +...+ (n-1) ≈ n + n/2 + n/4 + n/8 +...+ 1

Esta es una serie geométrica con razón 1/2 y aproximadamente logarítmica de base 2.

Por lo tanto, la suma total de esta serie es O(n*log n).

Entonces, el número total de iteraciones del bucle interno a lo largo de todas las iteraciones del bucle externo es O(n*log n) en total.

Después es evidente descubrir que el bucle externo se ejecuta O(log n) veces, ya que el valor de "gap" se divide en 2 cada iteración. Aquí lo detallo:

En la primera iteración: gap = n / 2.

En la segunda iteración: gap = (n / 2) / 2 = n / 4.

En la tercera iteración: gap = ((n / 2) / 2) / 2 = n / 8.

Y así sucesivamente...

Así llega a ser 1 cuando gap es menor que 1. Por lo tanto, el número total de iteraciones del bucle externo es O(log n), ya que el número de iteraciones es logarítmico con respecto al valor inicial de gap, que es proporcional a n.

Con todo esto solo queda determinar la eficiencia total, en este algoritmo (Shell Sort) la eficiencia es el producto de la eficiencia del bucle externo y la del bucle interno:

Eficiencia total = **Eficiencia del bucle externo** * **Eficiencia del bucle interno**

= **O(log n)** * **O(n log n)** = **O($n \log^2 n$)**

Por lo tanto, la eficiencia teórica del algoritmo Shell Sort completo es $\mathbf{O(n^*log^2(n))}$ aunque no debemos olvidar que es un método de ordenación muy complejo y según los datos que le demos puede ser o muy eficiente o no.

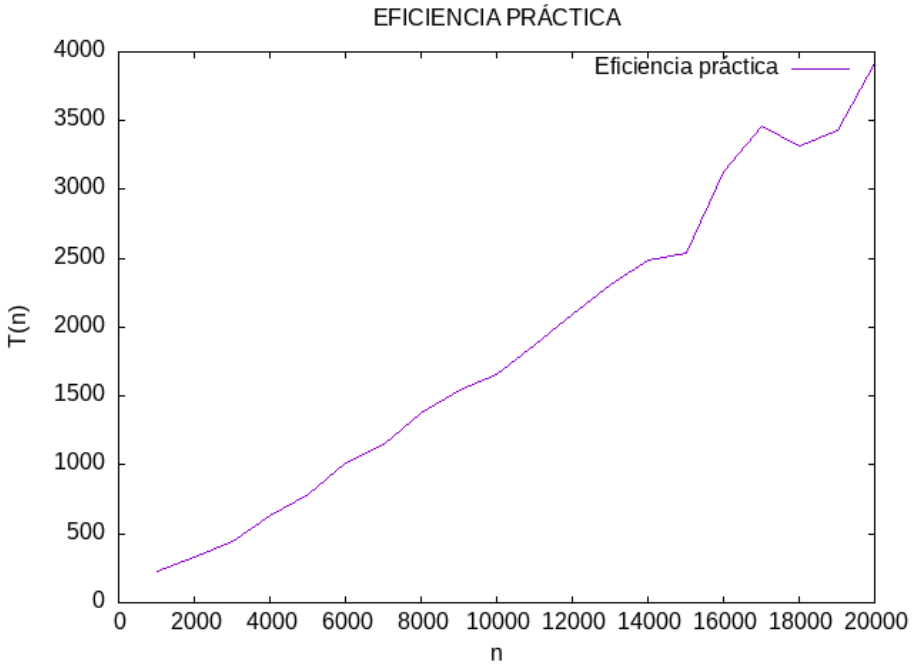
Cómo hemos comentado al inicio, también se dice que este algoritmo solía ser $\mathbf{O(n^2)}$**,** pero se acerca más a lo mencionado anteriormente aunque ambas cosas se aceptan al ser un método tan complejo. Además estudiando el mecanismo nos hemos dado cuenta que en casos normales puede acercarse a $\mathbf{O(n^*log\ n)}$**.**

Eficiencia práctica

Al ejecutar el algoritmo dándole valores desde el 1000 al 20000, los resultados son los siguientes:

Tam.Caso	***Práctica (tiempo en us)***
1000	233
2000	331
3000	448
4000	630
5000	788
6000	1019
7000	1149
8000	1376
9000	1545
10000	1657
11000	1877
12000	2089
13000	2306
14000	2490
15000	2539
16000	3128
17000	3458
18000	3318
19000	3426
20000	3919

Lo cual nos deja la siguiente gráfica generada con gnuplot:

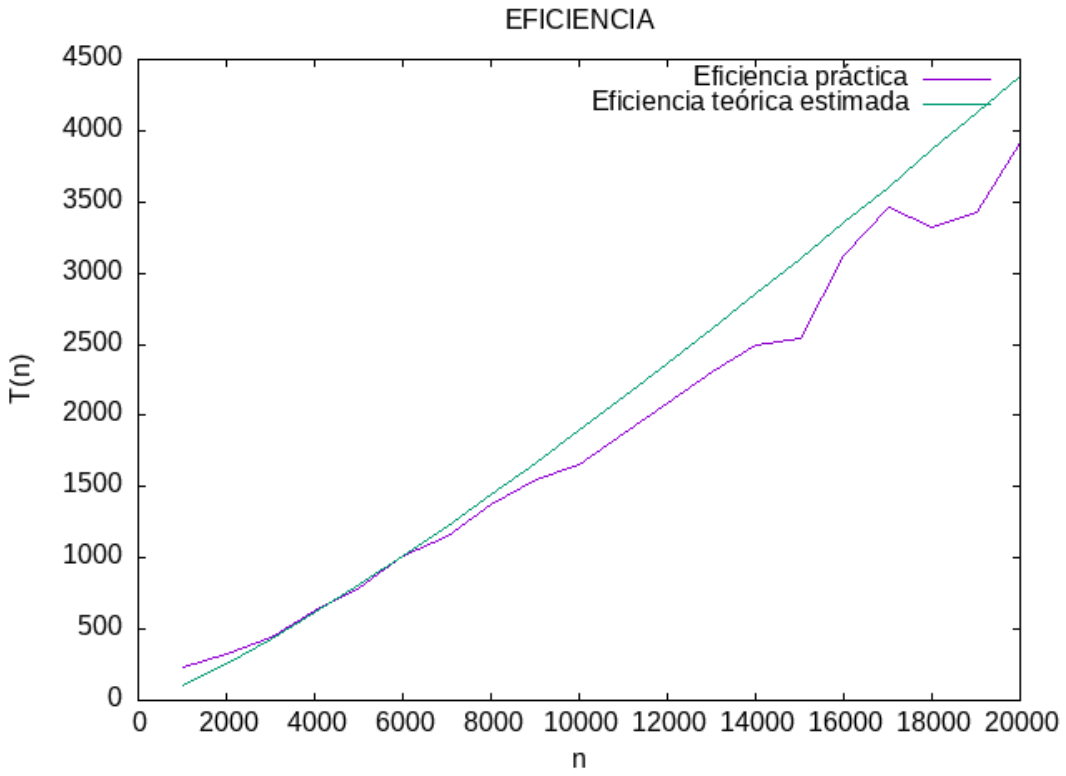


Como vemos, aunque es cierto que la gráfica hace unos giros extraños, si continuáramos dibujándola llegaríamos a ver algo parecido a la gráfica de la eficiencia que tenemos $O(n*\log^2(n))$. Estos pequeños saltos pueden venir de nuestra computadora, lo comprobaremos con la teórica más adelante.

Eficiencia híbrida

Tam.Caso (n)	***Práctica (tiempo en us)***	***f(n)***	***K= Tiempo/f(n)***	***Tiempo teórico estimado= K*f(n)***
1000	233	9000	0,0258888888888889	106,740976154355
2000	331	2179,59	0,0151879464336118	258,47443681393
3000	448	36271,11	0,0123514256315465	430,179377611591
4000	630	51899,34	0,0121388815859667	615,531857518305
5000	788	68411,89	0,0115184646565514	811,372500276868
6000	1019	85646,56	0,0118977339595356	1015,77750548702
7000	1149	103493,45	0,0111021516005565	1227,44357244136
8000	1376	121872,89	0,0112904517341493	1445,42571257751
9000	1545	140724,30	0,0109789137453217	1669,00551349831
10000	1657	160000	0,01035625	1897,61735385519
11000	1877	179661,40	0,0104474303698523	2130,80372822129
12000	2089	199676,63	0,010461915051421	2368,18655614172

13000	2306	220018,88	0,0104809183102828	2609,44788450508
14000	2490	240665,28	0,0103463196727112	2854,31641429583
15000	2539	261596,07	0,00970580318850311	3102,55779961203
16000	3128	282793,99	0,0110610551508862	3353,96747975683
17000	3458	304243,86	0,0113658823453151	3608,36526493698
18000	3318	325932,19	0,0101800315048836	3865,59116803264
19000	3426	347846,91	0,00984916022115592	4125,50214201466
20000	3919	369977,18	0,0105925451807596	4387,96948871332
			K promedio	0,01186010846159



Podemos ver que las gráficas quieren parecerse aunque es cierto que difieren un poco, esto a su vez puede ser porque la teórica la hemos calculado como el peor de los casos posibles mientras que en la práctica vemos como casi todo el rato para los mismos valores tiene un tiempo menor, esto nos sugiere que en este caso, el mecanismo no llegaría al peor de los casos aunque para tamaños pequeños (1000-3000) vemos como la práctica adquiere unos tiempos superiores. No hemos seguido calculando tiempos superiores pero la inercia nos hace suponer que para valores superiores la práctica seguirá estando por debajo de la teórica.

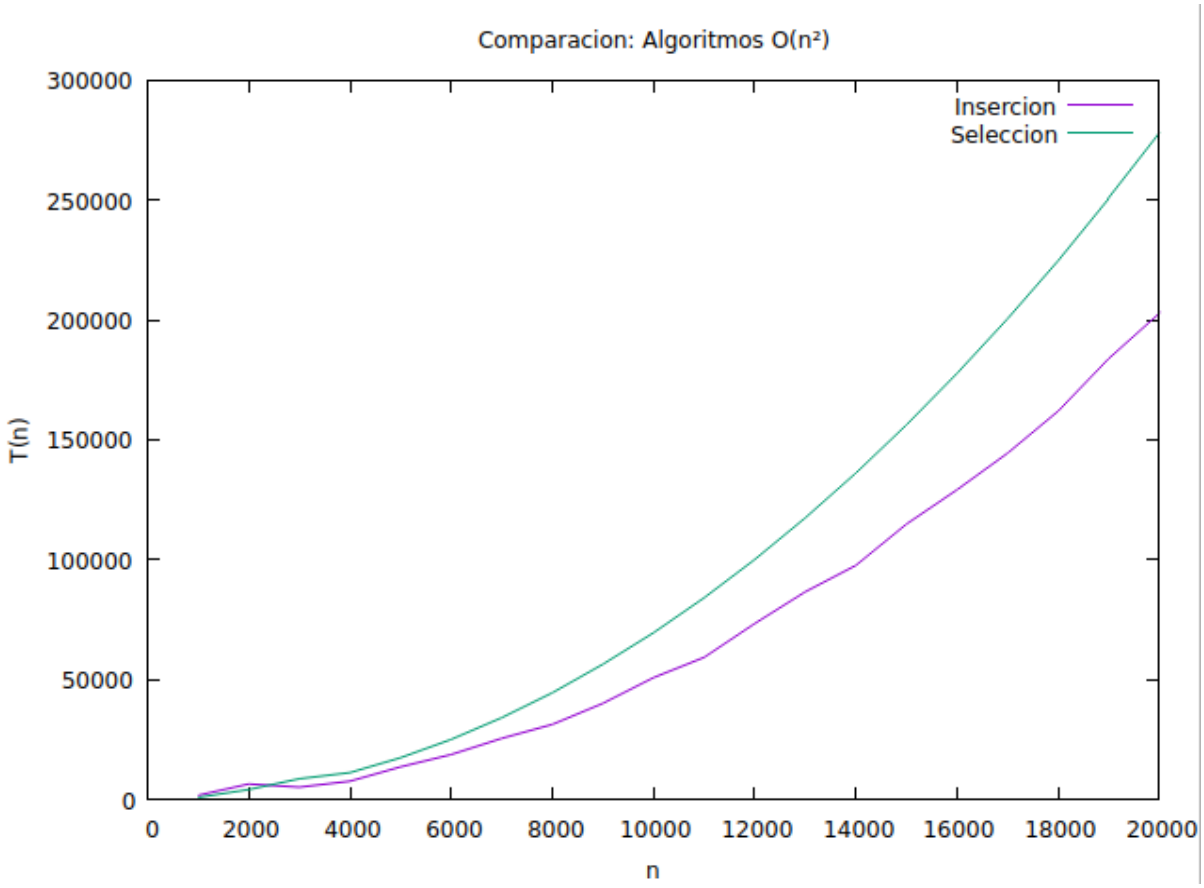
Comparación final entre algoritmos:

Tras la completa realización del estudio, debemos comparar los algoritmos entre sí y usar las conclusiones sacadas en cada uno para definir en qué situación sería óptimo usar cada uno de ellos, señalando así los más eficientes y rápidos.

Clasificaremos por lo tanto, en dos grupos; los algoritmos de orden cuadrático (menos eficientes), y los algoritmos de menor orden (o más eficiente):

Comparación: Algoritmos orden O(n²)

En este grupo, encontraremos los algoritmos de inserción y selección, y compararemos la eficiencia práctica y teórica de estos.
En la teoría, ambos son algoritmos con eficiencia de orden O(n²), veamos en la práctica:

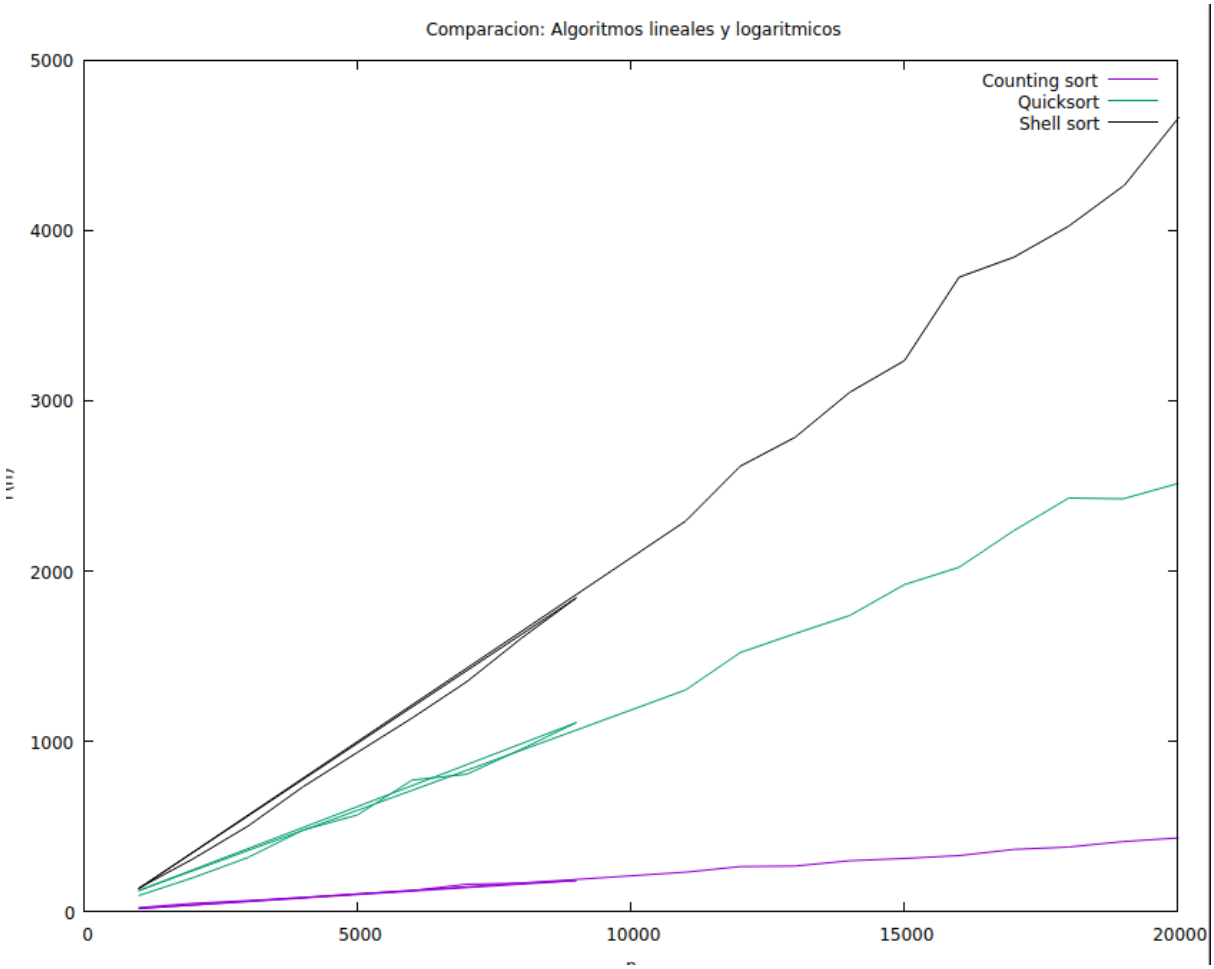


Como se puede ver, en la práctica ha salido que nuestro algoritmo de selección es más lento que el de inserción. Sin embargo, la conclusión a extraer de aquí es que

ambos algorítmicos son tremendamente lentos para conjuntos de datos muy grandes, (menos eficientes que quicksort o heap sort, por ejemplo) PERO son extremadamente eficientes para conjuntos de de datos pequeños o casi ordenados.
En conclusión, estos algoritmos no son inútiles por ser $O(n^2)$, sino que serán bastante lentos para conjuntos de datos grandes pero muy rápidos y recomendables para conjuntos de datos pequeños o casi ordenados (comparado con el resto de algoritmos, como los log que empiezan más alto T(n) al mismo nivel de n que las cuadráticas al principio).

Comparación: Algoritmos logarítmicos y lineales

Aquí pondremos en comparación los algoritmos quicksort O(n•log(n)), shell sort O(n•log(n)^2) y counting sort O(n+k). Veremos cuál de estos es el más rápido a partir de los T(n) obtenidos en función de n.



Aquí se puede comprobar cada una de las eficiencias prácticas. Como se puede ver hemos aplicado valores muy pequeños, donde el shell sort parece bastante ineficiente comparado con el quicksort y el counting sort. Cabe destacar que es más eficiente en este caso el counting sort respecto del quicksort, ya que el quicksort como se aprecia es O(n*log(n)), mientras que el counting sort es lineal O(n).

Por lo tanto, respecto a este último apartado, podemos decir que el algoritmo más eficiente de estos tres, sería el counting sort. Sin embargo, esto sería si hablamos de estos pequeños tamaños de grupos de datos. En cambio, si hablásemos de velocidad y tratamiento de conjuntos de datos muy grandes, posiblemente el algoritmo más rápido de los 3 fuese quicksort, suponiendo la elección inteligente del pivote y su correcta implementación.

En conclusión, podemos decir que si hay unos algoritmos más eficientes y rápidos que otros, pero lo importante es saber cuando utilizar cada uno de ellos, pues puede ser el quicksort que es rapidísimo pero tengamos un grupo muy pequeño de datos, lo cual lo haga ineficiente.Al contrario, podemos querer usar el counting short para la ordenación de un vector ya que vemos que este es lineal, pero si lo hacemos en un vector con una enorme cantidad de datos, quizás no sería el método más adecuado para ordenar este vector. Por lo tanto, deberíamos intentar “no matar moscas a cañonazos”, no usar un quicksort para un vector desordenado pequeño, ni el counting sort para uno grande.

Finalmente, pensamos que lo más importante es conocer los algoritmos y saber usarlos, más que intentar usar siempre los que sean más eficientes, ya que quizás un algoritmo a sea más ineficiente que uno b, pero debido a su implementación pueda servirnos más en determinado momento.