PRÁCTICA 1:
EFICIENCIA DE ALGORITMOS

Integrantes del grupo:

Ana López Mohedano
Adrián Anguita Muñoz
Marina Jun Carranza Sánchez
Pedro Antonio Mayorgas Parejo
Rafael Jiménez Márquez

ÍNDICE

0. Explicaciones previas....3
1. Conteo (Counting Sort)....3
a) Eficiencia teórica....4
b) Eficiencia práctica y eficiencia híbrida....5
c) Comparación entre teórica e híbrida....6
2. Inserción (Insertion Sort)....7
a) Eficiencia teórica....7
b) Eficiencia práctica....8
c) Eficiencia híbrida....8
d) Comparación entre teórica e híbrida....9
3. Rápido (Quicksort)....10
a) Eficiencia teórica....11
b) Eficiencia práctica....12
c) Eficiencia híbrida....13
d) Comparación entre teórica e híbrida....13
4. Selección (Selection Sort)....14
a) Eficiencia teórica....14
b) Eficiencia práctica....15
c) Eficiencia híbrida....15
d) Comparación entre teórica e híbrida....16
5. Ordenamiento Shell (Shell sort)....17
a) Eficiencia teórica....17
b) Eficiencia práctica....18
c) Eficiencia híbrida....19
d) Comparación entre teórica e híbrida....19
6. Comparación general de algoritmos....21

0. Explicaciones previas

Cálculo de la eficiencia híbrida

Para calcular la eficiencia híbrida, se repetirá el mismo proceso en los 5 algoritmos. Habiendo calculado la eficiencia teórica; es decir, la complejidad temporal en notación O y sabiendo que saber que existe una constante oculta K para cada algoritmo, tal que el tiempo T(n) de ejecución del mismo para un tamaño de caso n es: **T(n) <= K*f(n)**.

Por lo que para calcular K despejamos la fórmula anterior **K = T(n)/f(n)**.

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo para distintos tamaños de caso produciendo valores aproximados para K. Aproximamos el valor de K como la media de todos estos valores obtenidos.

1. Conteo (Counting Sort)

Este algoritmo basa el ordenamiento en la frecuencia acumulada de las veces que aparece un elemento en el propio vector de entrada.

La frecuencia acumulada total al final del vector de frecuencias, es igual a N. Por lo que a cada elemento le corresponde de manera ordenada un lugar por la frecuencia en el vector final.

El tamaño del vector de frecuencias, está determinado por el máximo elemento que se pueda encontrar en el vector de entrada.

```
NON CUMULATIVE
1 1 1 2
CUMULATIVE
1 2 3 5
ActualCell: 4 Actual Elem Ordering 3 Sizeof Output Arr 5
ActualCell: 0 Actual Elem Ordering 0 Sizeof Output Arr 5
ActualCell: 2 Actual Elem Ordering 2 Sizeof Output Arr 5
ActualCell: 1 Actual Elem Ordering 1 Sizeof Output Arr 5
ActualCell: 3 Actual Elem Ordering 3 Sizeof Output Arr 5
INPUT VECTOR
3 1 2 0 3
ORDERED VECTOR
0 1 2 3 3
```

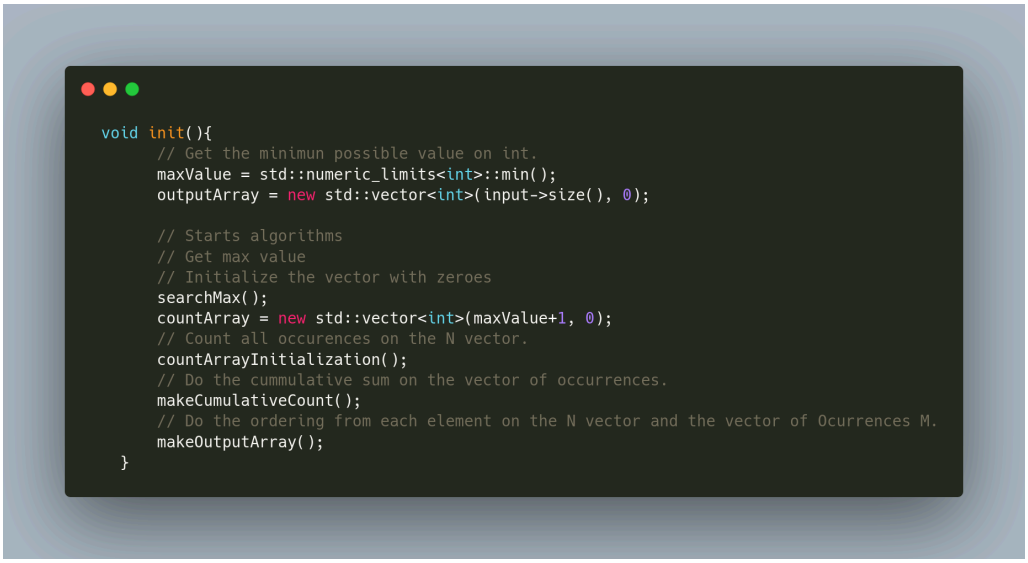
En la ejecución de ejemplo, podemos ver que dado un vector llamado INPUT VECTOR podemos derivar, que hay una frecuencia de elementos, que se contabilizan hasta 3. Luego esa frecuencia se tiene que hacer de manera acumulada. Por lo que al iterar en reverso el vector de entrada, podemos:

- Para elem -> 3

Si cogemos la frecuencia acumulada y le restamos 1 ya que los vectores empiezan en 0. Obtenemos la posición final de ordenamiento. Una vez terminado, se disminuye en 1 la frecuencia acumulada.

a) Eficiencia teórica

Análisis de la eficiencia a través del código.



```
void init(){
      // Get the minimun possible value on int.
      maxValue = std::numeric_limits<int>::min();
      outputArray = new std::vector<int>(input->size(), 0);

      // Starts algorithms
      // Get max value
      // Initialize the vector with zeroes
      searchMax();
      countArray = new std::vector<int>(maxValue+1, 0);
      // Count all occurences on the N vector.
      countArrayInitialization();
      // Do the cummulative sum on the vector of occurrences.
      makeCumulativeCount();
      // Do the ordering from each element on the N vector and the vector of Ocurrences M.
      makeOutputArray();
  }
```

searchMax

Eficiencia en el peor caso **O(N+M):** Esta función tiene que buscar el elemento máximo que exista en el vector de entrada. Una vez terminado se ejecuta la inicialización y creación del vector de conteo de tamaño M.

countArrayInitialization

O(N): Para cada elemento del vector de entrada, tenemos que contabilizarlo en el vector de conteo.

- Para cada elemento se accede a su correspondiente celda en el vector de conteo y se hace un incremento. 1

makeCumulativeCounts

O(M): Sumamos cada elemento del vector de conteo, haciendo una suma acumulativa. Esa suma acumulativa, indicará en el futuro qué posición le corresponderá a cada elemento en el vector final.

makeOutputArray

O(N) Aquí es donde “ordenamos”, a partir de los vectores de entrada y el de suma acumulativa.

Para cada elemento del vector de entrada N:

- Obtenemos el elemento 1.
- Accedemos en el vector de sumas acumulativas 1.

- En el vector de salida del mismo tamaño N, accedemos a la posición que le corresponde calculada a partir del vector de sumas acumulativas y guardamos el elemento ya ordenado. 1
- Disminuimos en uno el vector de conteo con la suma acumulativa. Para indicar la siguiente posición del otro elemento que está contigua.

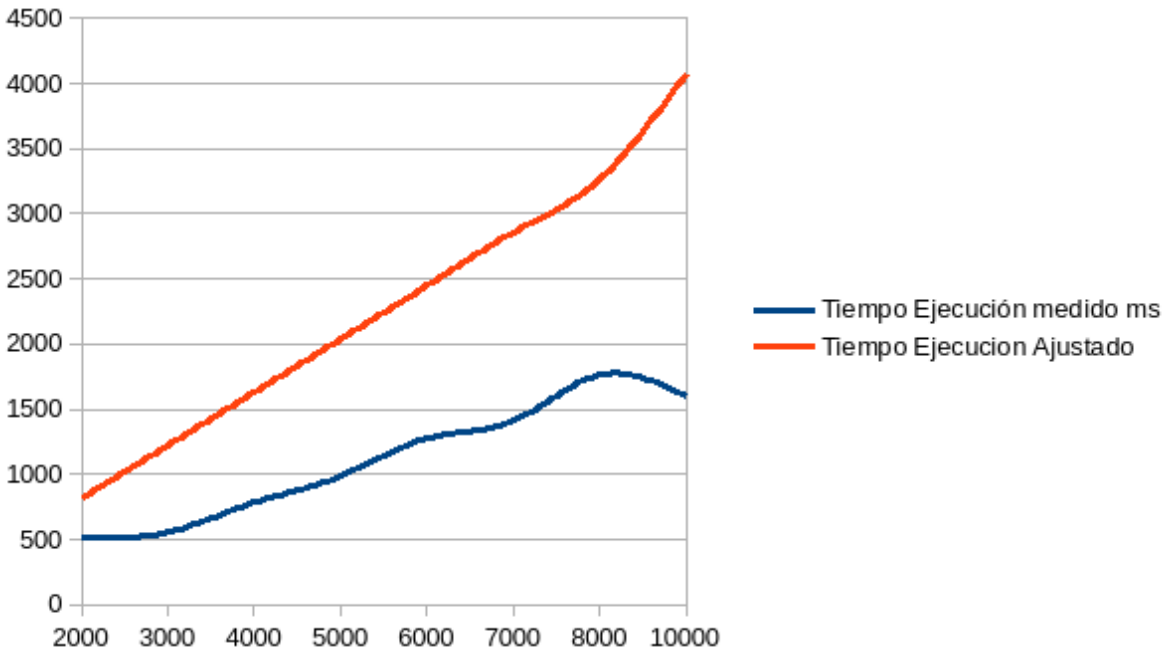
```
void makeOutputArray(){
    unsigned int actualCellOutput = -1;
    // Counting from reverse from the input
    for (auto it=input->rbegin(); it!=input->rend();it++){
        // Accessing to the array of occurrences cummulated, and we get the position -1 because
        // the cummulative frecuency marks the final possition ordering.
        actualCellOutput = countArray->operator[](*it)-1;
        // Writes to the output with the position precalculated.
        outputArray->operator[](actualCellOutput) = *it;
        //Upgrade the occurrences for the next, minus one ocurrence.
        countArray->operator[](*it)--;
    }
}
```

Por lo tanto la eficiencia big O, es la máxima eficiencia de todo el código, que en este caso es O(n+k). Es decir, tiene complejidad de lineal **O(n)**.

b) Eficiencia práctica y eficiencia híbrida

Tam Caso	Tiempo Ejecución medido (ms)	Tiempo Ejecucion Ajustado
2000	514	816.507322125
3000	559	1224.863072125
4000	785	1633.01464425
5000	990	2041.574572125
6000	1280	2449.72614425
7000	1413	2858.286072125
8000	1766	3266.641822125
10000	1599	4083.353322125
MEAN K	0.204177875	

c) Comparación entre teórica e híbrida



2. Inserción (Insertion Sort)

a) Eficiencia teórica

Variable o variables de las que dependen el tamaño del caso: El tamaño del problema depende del número de componentes del subvector a ordenar. Por tanto, n=posFin-PosIni+1.

Es un algoritmo secuencial. Tenemos el caso base y el caso general:

- ***Caso base:*** Cuando n<=1. El for evalúa la condición, y no se cumple desde un principio (ya que si n=1, posIni y posFin serían 0), por lo que no se mete en el for y termina la ejecución. Por tanto, el caso base es **O(1).**
- ***Caso general:*** Cuando n>1. En este caso
  - El bucle exterior recorre el vector desde posIni + 1 hasta posFin inclusive, lo que implica recorrer n elementos en el peor caso (cuando el vector está totalmente desordenado), donde n = posFin - posIni + . Entonces el bucle exterior tiene **O(n)**.
  - Dentro del bucle exterior, tenemos dos asignaciones simples, cada una con un **O(1)**, y una después del bucle while que también es **O(1)**.
  - El bucle interior se ejecuta en el peor de los casos n veces, por lo que tendría un **O(n)**. El peor de los casos sería que para colocar cada valor tuviera que recorrer todo el vector, que todos los valores desde i hasta posIni fueran mayores que el que estamos evaluando. Por lo tanto, tiene que recorrer todo el vector hacia atrás hasta llegar a posIni, o hasta encontrar un valor que sea menor. En resumen, en cada iteración se desplaza hacia atrás hasta la posición posIni (como máximo) para encontrar la ubicación correcta del elemento actual. Las asignaciones del bucle interior son simples, de **O(1)** cada una.
  - **<u>Conclusión:</u>** Al tener un for con O(n), que anida a un bucle while con O(n), tendremos **$O(n^2)$** para todo el algoritmo.

```
 9
10  void InsertionSort(int *v, int posIni, int posFin) {
11      for (int i = posIni + 1; i <= posFin; ++i) {
12          int valor = v[i];
13          int j = i;
14          while (j > posIni && v[j - 1] > valor) {
15              v[j] = v[j - 1];
16              --j;
17          }
18          v[j] = valor;
19      }
20  }
21
```

El algoritmo de Insertion Sort ordena una lista de elementos mediante la inserción secuencial de cada elemento en su lugar adecuado.
El algoritmo comienza con el segundo elemento del array. Este elemento se considera trivialmente ordenado, ya que un solo elemento siempre está ordenado. Se selecciona el siguiente elemento no ordenado, y lo comparamos con los elementos anteriores en el array, desde la posición actual hacia atrás. Se mueven los elementos mayores hacia delante en el array mientras el elemento seleccionado sea menor que el elemento en esa posición. Esto abre espacio para el elemento seleccionado. Una vez que se encuentra la posición correcta para el elemento seleccionado, lo insertamos en esa posición. Repetiremos este proceso para cada elemento no ordenado del array, avanzando gradualmente hacia la derecha.

b) Eficiencia práctica

Se han tomado tamaños de caso n en el intervalo [1000, 10000]:

Tam. Caso	Tiempo (us)
1000	638
2000	2545
3000	4866
4000	8141
5000	13214
6000	19468
7000	26281
8000	38498
9000	41743
10000	51257

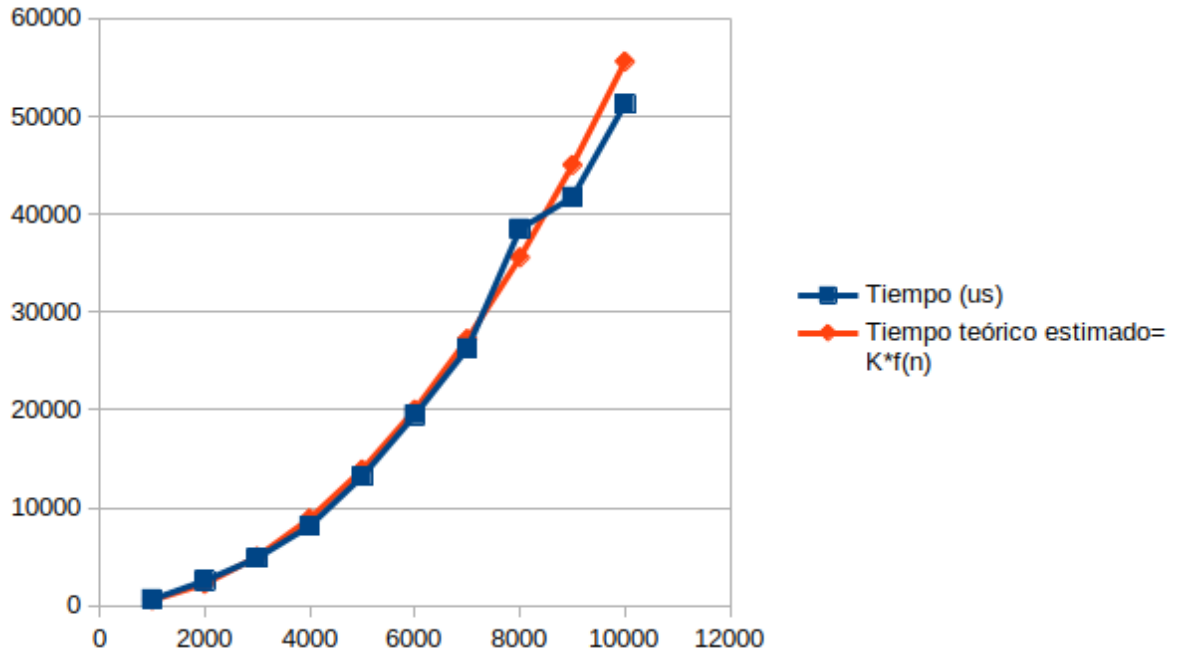
c) Eficiencia híbrida

Aproximaremos el valor final de K como la media de todos estos valores. Nuestra K en este caso sería 0,00055588608.

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	638	0,000638	555,88608122323
2000	2545	0,00063625	2223,54432489292
3000	4866	0,00054066667	5002,97473100907
4000	8141	0,0005088125	8894,17729957168
5000	13214	0,00052856	13897,1520305808
6000	19468	0,00054077778	20011,8989240363
7000	26281	0,00053634694	27238,4179799383
8000	38498	0,00060153125	35576,7091982867
9000	41743	0,00051534568	45026,7725790816
10000	51257	0,00051257	55588,608122323
	K promedio:	0,00055588608	

d) Comparación entre teórica e híbrida

Aquí podemos ver la comparación de las gráficas entre el tiempo teórico estimado con la constante oculta calculada anteriormente (naranja), y el tiempo real medido (azul). Podemos comprobar que el orden O(f(n)) calculado con una constante oculta, normalmente será mayor que el tiempo de ejecución real del algoritmo, ya que el tiempo teórico indicará el peor de los casos. Ambas funciones siguen la forma de la función f(n), siendo esta una función cuadrática. Podemos observar que en el tamaño de problema n=8000, el tiempo medido es mayor que el estimado, esto puede pasar por distintos factores, por implementación, por interferencias del sistema (como pueden ser otros procesos, la CPU, la memoria, etc.), por el entorno de ejecución (sistema operativo, la configuración del hardware, etc.), o incluso por errores de medición del tiempo.



3. Rápido (Quicksort)

El método Quicksort ordena un vector de elementos.

Este método es una mejora del método de intercambio directo, su nombre viene dado por la velocidad con la que ordena los elementos del array está basado en la técnica de divide y vencerás.

```
8
9  int dividir(int *array, int inicio, int fin)
10 {
11   int izq;
12   int der;
13   int pibote;
14   int temp;
15
16   pibote = array[inicio];
17   izq = inicio;
18   der = fin;
19
20   //Mientras no se cruzen los índices
21   while (izq < der){
22     while (array[der] > pibote){
23       der--;
24     }
25
26     while ((izq < der) && (array[izq] <= pibote)){
27       izq++;
28     }
29
30     // Si todavia no se cruzan los indices seguimos intercambiando
31     if(izq < der){
32       temp= array[izq];
33       array[izq] = array[der];
34       array[der] = temp;
35     }
36   }
37
38   //Los indices ya se han cruzado, ponemos el pivote en el lugar que le corresponde
39   temp = array[der];
40   array[der] = array[inicio];
41   array[inicio] = temp;
42
43   //La nueva posición del pivote
44   return der;
45 }
46
```

```
48
49 void quicksort(int *array, int inicio, int fin)
50 {
51   int pivote;
52   if(inicio < fin)
53   {
54     pivote = dividir(array, inicio, fin );
55     quicksort( array, inicio, pivote - 1 );//ordeno la lista de los menores
56     quicksort( array, pivote + 1, fin );//ordeno la lista de los mayores
57   }
58 }
59
```

a) Eficiencia teórica

Variable o variables de las que dependen el tamaño del caso: El tamaño del problema viene dado por el número de elementos del vector a ordenar por lo tanto sería **"n"** que es igual a la longitud del vector a ordenar.

Es un algoritmo recursivo por lo que vamos a llamar T(n) al tiempo de ejecución que tarda el algoritmo "Quicksort" en resolver un problema en el cual el tamaño del mismo es **"n"**. Existen 2 casos principales un caso general y un caso base.

-Caso Base: si n <= 1 el vector ya está ordenado por lo que consideraríamos que es O(1). Esto es porque al ser el tamaño menor o igual a 1 lo que hace es en la función dividir comparar el tamaño con 1 y retornar por lo que al ser estas operaciones constantes la eficiencia es O(1).

-Caso General: Sucede cuando n > 1 cuando esto sucede el algoritmo hace lo siguiente:

1. Se elige un elemento que actuará como pivote: La elección del pivote suele ser O(1) pero pueden haber casos especiales en los cuales la elección del pivote no sea O(1) y sea peor como por ejemplo recorrer la lista de antemano para saber que elemento ocupará la posición central para elegirlo esto puede hacerse en O(n) e incluso en el peor de los casos puede llegar a ser O(n*log n).

2.Partimos el vector en 2 subvectores uno con los elementos mayores que el pivote y otro con los elementos menores: Recorremos el vector desde inicio+1 hasta fin usando izq y der intercambiando los elementos que son menores que el pivote con los que son mayores por lo que esto tardaría O(n) en promedio.

3. Ordenamos los dos subvectores de manera recursiva: Realizamos dos llamadas recursivas a la función Quicksort para ordenar los 2 subvectores que obtuvimos anteriormente por lo que el tamaño del problema sería n/2 y siendo T(n) el tiempo que tarda el algoritmo en resolver el problema de tamaño n por lo que la llamada recursiva tendrá un tiempo de ejecución de T(n/2).

La siguiente llamada recursiva en la que se resuelve el problema desde pivote + 1 hasta fin podemos aproximar su tamaño a n/2 por lo que el tiempo de ejecución sería de T(n/2).

Por lo que la función Quicksort tiene un tiempo de ejecución de T(n) = O(1) + O(n) + 2T(n/2).

A continuación, vamos a resolver la ecuación de recurrencias:

T(n) = 2*T(n/2) + n

Tenemos que realizar un cambio de variable n = 2^c , c = log2(n)

Por lo que la ecuación quedaría de la siguiente manera T(2^c) = 2*T(2^c-1) + 2^c

Una vez tenemos esa ecuación hacemos el siguiente cambio Tc = 2*Tc-1 + 2^c

Y ahora lo pasamos todo a un lado Tc – 2*Tc-1 = 2^c

Sacamos el polinomio de la parte homogénea p(x) = x – 2

Y ahora vamos a calcular la parte no homogénea: 2^c = b1^c * q1(c)

Por lo que deducimos que b1 = 2 y q1(c) = 1 y d1 = 0 este es el grado

Una vez tenemos eso formamos el polinomio completo p(x) = (x-2) * (x-b1)^d1+1

Por lo que quedaría como p(x) = (x-2) * (x-2) = (x-2)^2

Con eso sacamos la ecuación Tc = C10 * 2^c + C11 * 2^c *c

Y deshacemos el cambio para obtener la ecuación final

T(n) = C10 * 2^log2(n) + C11 * 2^log2(n) * log2(n) = C10 * n + C11 * n * log2(n)

Aplicamos la regla del máximo por lo que obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es O(n*log(n)) para el caso general o promedio.

Pero a esto podemos añadirle una excepción dado que en el peor caso es decir que si siempre elegimos como pivote el extremo tanto menor como mayor el algoritmo se comporta de manera parecida al de inserción con un tiempo de ejecución de O(n^2) y otro elemento que puede afectar al rendimiento del algoritmo es la presencia de elementos repetidos.

b) Eficiencia práctica

Se han tomado tamaños de caso *n* en el intervalo [10000, 500000]:

Tam. Caso	Tiempo (us)
10000	2872
20000	2295
30000	5041
40000	6220
50000	6171
60000	8347
70000	9140
80000	10452
90000	12355
100000	12868
110000	15391
120000	15534
150000	20798
200000	28487
300000	42714
400000	58710
500000	72491

c) Eficiencia híbrida

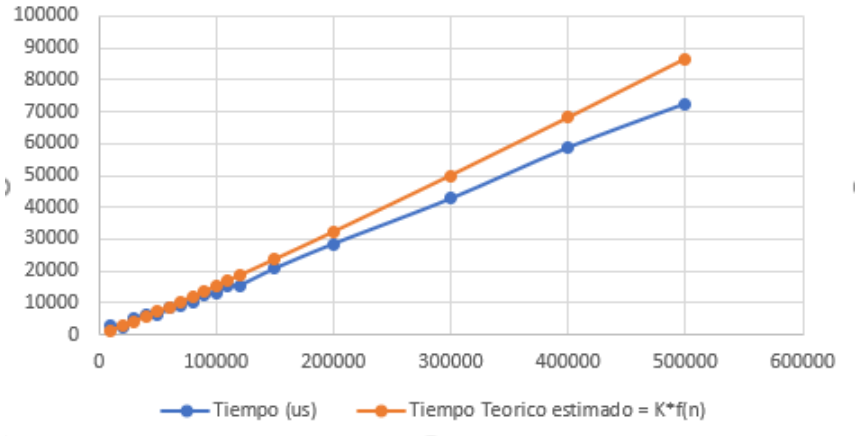
Para calcular la K lo haremos como hemos visto en los anteriores, despejando la K:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo Teorico estimado = K*f(n)
10000	2872	0,0718	1215,892166
20000	2295	0,026679656	2614,794339
30000	5041	0,037531557	4082,772496
40000	6220	0,033789216	5595,608692
50000	6171	0,02626533	7141,801023
60000	8347	0,029115166	8714,575012
70000	9140	0,026949182	10309,45432
80000	10452	0,026646462	11923,25741
90000	12355	0,027709136	13553,60548
100000	12868	0,025736	15198,65208
110000	15391	0,027753875	16856,92215
120000	15534	0,025486391	18527,21007
150000	20798	0,026787266	23600,88305
200000	28487	0,026869307	32227,40423
300000	42714	0,025995408	49946,91621
400000	58710	0,026200184	68115,00859
500000	72491	0,025440036	86616,66231
	K Promedio:	0,030397304	

d) Comparación entre teórica e híbrida

En el siguiente gráfico el eje vertical representa el tiempo de ejecución y el horizontal el tamaño del caso (n).

Como vimos anteriormente aunque la eficiencia teórica sea O(n*log(n))en los casos prácticos reales se suelen obtener mejores resultados esto lo comprobamos gracias a la gráfica donde vemos que a partir de un tamaño de n grande vemos que los tiempos que hemos medido son mejores que los que se estiman teóricamente.



4. Selección (Selection Sort)

El algoritmo de selección ha sido extraído de wikipedia, siguiendo el enlace proporcionado en el guión de la práctica. El código es el siguiente:

```
73
74  void seleccion(int* v, int n){
75      int menor;
76      int aux;
77
78      for (int i = 0; i < n - 1; i++)
79      {
80          menor = i;
81          for (int j = i + 1; j < n; j++)
82          {
83              if (v[j] < v[menor])
84              {
85                  menor = j;
86              }
87          }
88          aux = v[i];
89          v[i] = v[menor];
90          v[menor] = aux;
91      }
92  }
```

a) Eficiencia teórica

Vamos a calcular la eficiencia teórica de este algoritmo. Para ello, vemos que la variable de la que depende el tamaño del caso es un único parámetro n, que es la longitud del vector a ordenar.

Comenzamos analizando la función desde la parte más interna. Las líneas 83-86 contienen un if en el que se hace una comparación entre dos componentes de un vector y si resulta cierta la condición se hace una asignación, siendo tipos de operaciones básicas y, por tanto, O(1).

Si analizamos el resto del bucle for de las líneas 81-87 vemos que su inicialización es O(1) por ser una operación elemental, al igual que la comprobación y la actualización. En el peor de los casos, el bucle se ejecuta n-1 veces (cuando i=0, al principio del otro bucle for). Su orden de eficiencia será (n-1)*(O(comprobación) + O(actualización) + O(sentencias del bucle)) que, como todas son O(1), aplicando la regla del máximo la eficiencia es O(n-1), que equivale a O(n).

Ahora analizamos el bucle for que contiene al analizado anteriormente. Todas las sentencias exceptuando el bucle for analizado anteriormente son operaciones elementales y, por tanto, son O(1). La secuencia de todas estas operaciones también es O(1), aplicando la regla del máximo.

Vemos que la inicialización de este bucle es también O(1) por ser una operación elemental, al igual que la comprobación y la actualización. En el peor de los casos, vemos que el bucle se ejecuta también n-1 veces, por tanto, su orden de eficiencia será, aplicando la regla del máximo, O(n-1), que equivale a O(n).

De este modo, sabemos que el bucle for se ejecutará un máximo de n-1 veces y el bucle for interno igualmente n-1 veces. El resto de operaciones de la función son operaciones elementales, por lo que la eficiencia de este algoritmo es O(n-1)*O(n-1), que equivale a $O(n^2)$.

Concluimos, por tanto, que la eficiencia del algoritmo de selección es de orden $O(n^2)$.

b) Eficiencia práctica

Para el cálculo de la eficiencia práctica hemos utilizado un n desde 1000 hasta 10000 incrementando en cada ejecución el n en 1000 debido al orden $O(n^2)$ que presenta este algoritmo de ordenación. El resultado obtenido ha sido el siguiente:

```
º cuatrimestre/ALG/ALGP1$ ./Seleccion.bin salida.txt 12345 1000 2000 30
00 4000 5000 6000 7000 8000 9000 10000
n=1000 1551 us
n=2000 5297 us
n=3000 11731 us
n=4000 20591 us
n=5000 31138 us
n=6000 44417 us
n=7000 60549 us
n=8000 78864 us
n=9000 104379 us
n=10000 122881 us
```

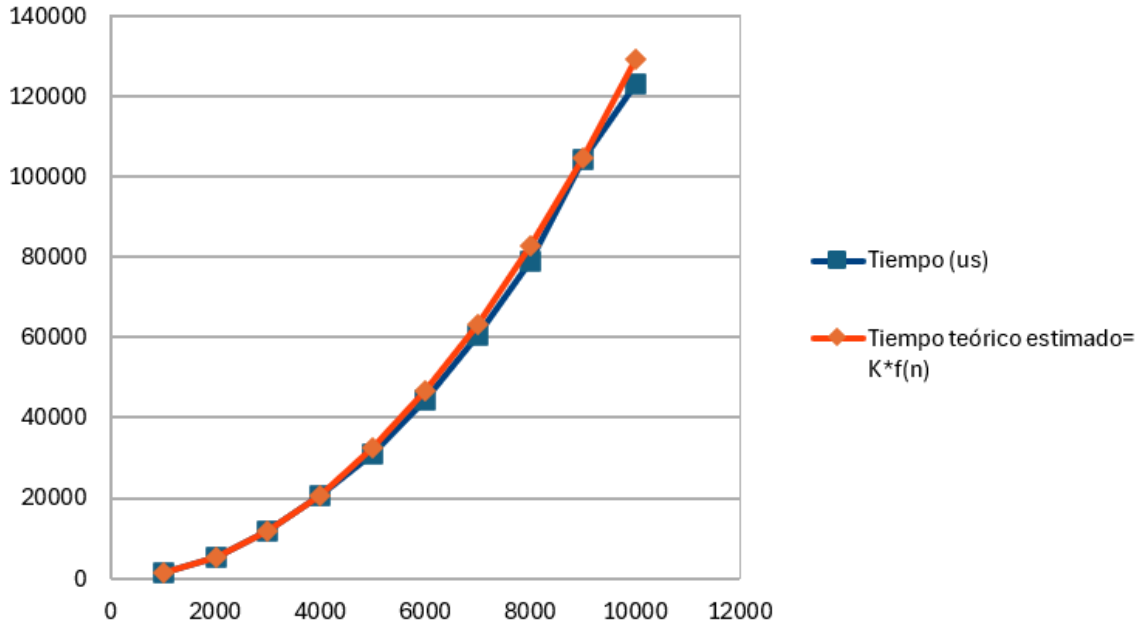
c) Eficiencia híbrida

Aproximamos el valor final de K como la media de todos los valores, obteniendo como resultado la siguiente tabla:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	1551	0,001551	1293,034101
2000	5297	0,00132425	5172,136403
3000	11731	0,001303444	11637,30691
4000	20591	0,001286938	20688,54561
5000	31138	0,00124552	32325,85252
6000	44417	0,001233806	46549,22763
7000	60549	0,001235694	63358,67094
8000	78864	0,00123225	82754,18245
9000	104379	0,00128863	104735,7622
10000	122881	0,00122881	129303,4101
	K promedio	0,001293034	

d) Comparación entre teórica e híbrida

Una vez calculado el orden de eficiencia del algoritmo de selección, el valor de la constante oculta en promedio (K) calculado, a continuación comprobaremos experimentalmente que el orden O(f(n)) calculado con una constante oculta superior al K calculado siempre será mayor que el tiempo de ejecución real del algoritmo por lo que hemos conseguido acotar el tiempo de ejecución del mismo y estimar, en el futuro, cuánto tardará en ejecutarse para otros tamaños de casos en el peor de los casos. Se muestra la siguiente gráfica generada en la hoja de cálculo mostrando ambas curvas:



5. Ordenamiento Shell (Shell sort)

a) Eficiencia teórica

- El tamaño del problema es ***n***, el número de elementos a ordenar o dicho en otras palabras, el tamaño del vector desordenado.
- **Caso base**: si n<=1, el vector tiene un solo elemento y se considera que ya está ordenado, por lo que tendrá una complejidad de **O(1)**.
- **Caso general**: si n>1, se asume que está desordenado y puede llegar a tener una complejidad de **O(n^2)** en el peor de los casos [ver *Figura 1* de esta página].

Breve explicación del algoritmo, siendo `*v` el vector de tamaño n a ordenar:

- Mientras la brecha (`gap`), inicialmente `n/2`, sea mayor que 0:
  1) Recorre el vector, comparando cada elemento en posición `i` (`i`=0,1,...,n-1) con el separado por la brecha (`i+gap`).
     a) Si `v[i] > v[i+gap]`, intercambia las posiciones de los elementos.
     b) Si no, se consideran ordenados y se pasa a la siguiente comparación.
  2) Cuando llega al final del vector, recalcula el `gap` (`gap/2`) y vuelve al paso (1).

```
int nextGap(int gap) {
    if (gap <= 1) return 0;          } O(1)
    return (gap + 1) / 2;            }
}

void shellSort(int *v, int n) {  ← O(n^2)
    int gap = n / 2;  ← O(1)

    while (gap > 0) {  ← O(n^2)
        // Se realiza un ordenamiento por inserción para cada elemento con el gap
        for (int i = gap; i < n; i++) {  ← O(n)
            int temp = v[i];             } O(1)
            int j;                       }
            for (j = i; j >= gap && v[j - gap] > temp; j -= gap) {  ← O(n)
                v[j] = v[j - gap];  ← O(1)
            }
            v[j] = temp;  ← O(1)
        }
        // Se reduce el gap para la sig. iteración
        gap = nextGap(gap);  ← O(1)
    }
}
```

Figura 1: *Análisis de la eficiencia del código basado en el de Wikipedia.*

Por tanto, habiendo analizado la eficiencia del código anterior, se tiene que:

- La función `nextGap` tiene una complejidad temporal de **O(1)**, ya que solo realiza una operación aritmética y una comparación. Esta va calculando el valor de la brecha siguiendo la secuencia básica de Shell (`n/k`, `k=2,4,6…`).

- ➢ La función `shellSort` tiene eficiencia teórica de **$O(n^2)$** en el peor de los casos, que sucedería si para cada gap calculado tuviera que intercambiar todas las parejas que compara porque están desordenadas (n*n operaciones).

Cabe destacar la dificultad para llegar a un consenso en cuanto a la eficiencia teórica de Shell, ya que depende principalmente de los intervalos que se utilicen (el de Shell, Knuth, Hibbard, Pratt, Sedgewick…). No obstante, el caso medio suele ser de **$\theta(n \cdot \log_2(n)) \approx \theta(n^{1.25})$**, debido a que el número de comparaciones e intercambios tiende a reducirse a medida que el gap va disminuyendo. Con el análisis híbrido de los apartados siguientes se podrá observar esta mejora en la eficiencia.

En conclusión, la eficiencia del algoritmo de ordenamiento Shell depende de la secuencia de brechas utilizada. Con la secuencia de brechas de Shell, el algoritmo puede tener hasta $O(n \cdot \log(n))$, lo que lo hace más eficiente que el ordenamiento por inserción para listas grandes. Sin embargo, en el peor de los casos, la complejidad temporal puede ser $(O(n^2))$, similar al de inserción.

b) Eficiencia práctica

Se han tomado tamaños de caso *n* en el intervalo [2000, 50000]:

Tam. Caso (n)	Tiempo (us)
2000	490
4000	720
6000	1278
8000	2072
10000	2064
12000	2474
14000	3648
16000	4238
18000	4017
20000	5459
22000	6170
24000	6831
26000	6761
28000	7998
30000	7195
32000	10046
34000	9160
36000	9220
38000	11608
40000	14210
42000	13048
44000	13942
46000	14187
48000	15780
50000	15239

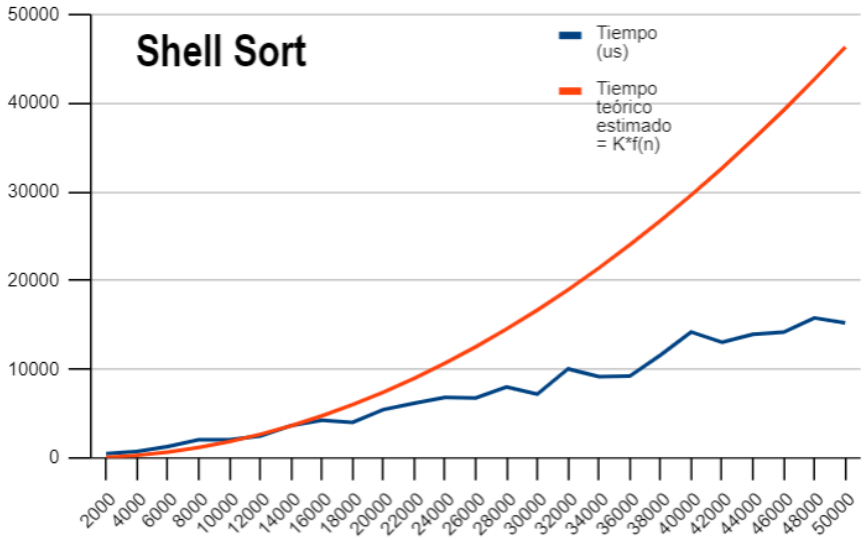
c) Eficiencia híbrida

Siendo $f(n)=n^2$ y usando K promedio para la expresión K*f(n):

Tam. Caso (n)	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
2000	490	0,0001225	74,11664188
4000	720	0,000045	296,4665675
6000	1278	0,0000355	667,0497769
8000	2072	0,000032375	1185,86627
10000	2064	0,00002064	1852,916047
12000	2474	0,00001718055556	2668,199108
14000	3648	0,0000186122449	3631,715452
16000	4238	0,0000165546875	4743,46508
18000	4017	0,00001239814815	6003,447992
20000	5459	0,0000136475	7411,664188
22000	6170	0,00001274793388	8968,113667
24000	6831	0,000011859375	10672,79643
26000	6761	0,00001000147929	12525,71248
28000	7998	0,00001020153061	14526,86181
30000	7195	0,000007994444444	16676,24442
32000	10046	0,000009810546875	18973,86032
34000	9160	0,000007923875433	21419,7095
36000	9220	0,000007114197531	24013,79197
38000	11608	0,000008038781163	26756,10772
40000	14210	0,00000888125	29646,65675
42000	13048	0,000007396825397	32685,43907
44000	13942	0,000007201446281	35872,45467
46000	14187	0,00000670463138	39207,70355
48000	15780	0,000006848958333	42691,18572
50000	15239	0,0000060956	46322,90117
	K promedio	0,00001852916047	

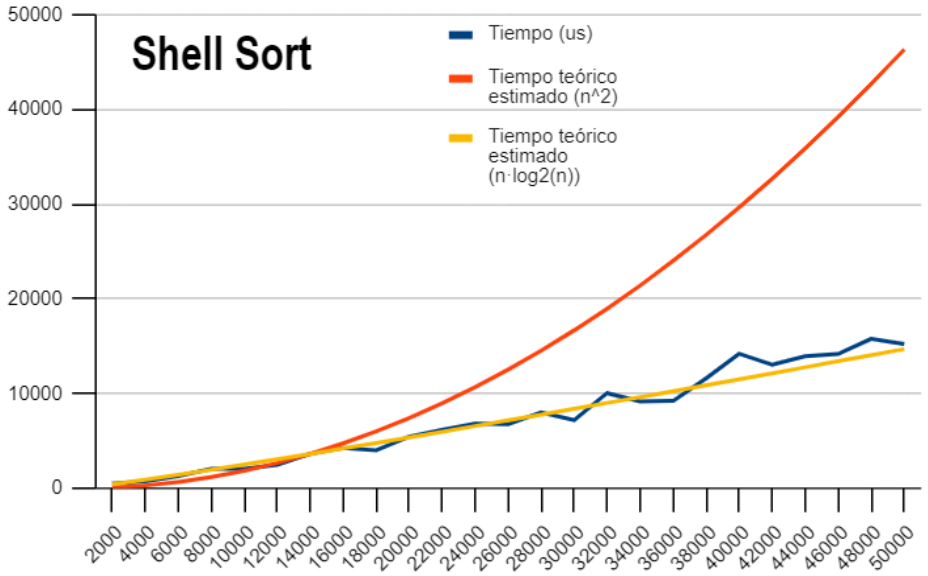
d) Comparación entre teórica e híbrida

En el siguiente gráfico de líneas, el eje vertical representa el tiempo de ejecución y el horizontal, el tamaño del caso (n). La línea azul son los tiempos medidos durante el análisis práctico y la roja, los calculados teóricamente.

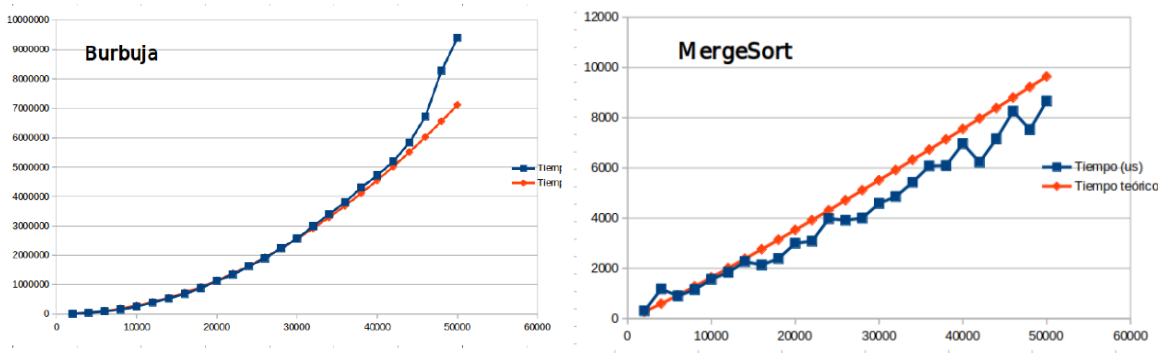


Como se explicó anteriormente, aunque la eficiencia teórica sea **$O(n^2)$**, como en el algoritmo de burbuja y el de inserción, en casos prácticos y reales se suelen obtener mejores resultados. Este hecho se puede comprobar en la gráfica de líneas de arriba, donde se observa que a partir de un tamaño de *n* considerable (≥16000), los tiempos medidos de forma práctica son mejores que los estimados teóricamente.

Es más, los resultados obtenidos en este caso se corresponden con la complejidad del caso promedio **$\theta(n \cdot \log_2(n))$**, lo que puede comprobarse si se traza una línea (la amarilla) que represente los tiempos teóricos estimados: K*f(n) donde f(n)=n*$\log_2$(n).

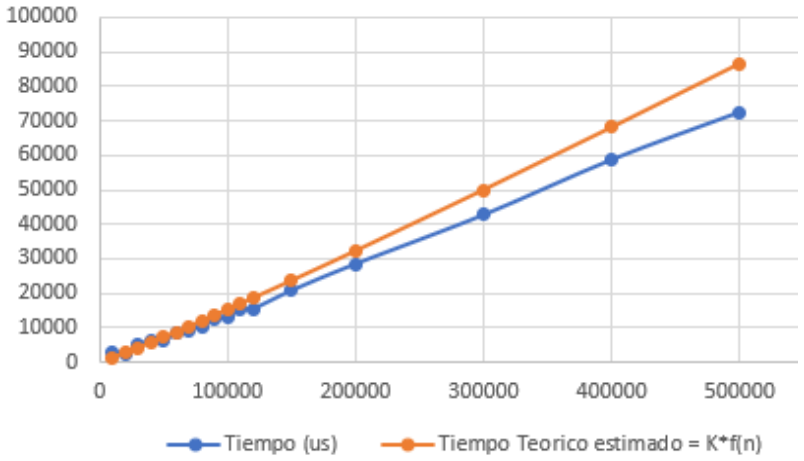


Dado que el algoritmo de burbuja tiene una complejidad de O(n^2) y Mergesort de O(n·$\log_2$(n)), podríamos inferir que Shell se sitúa entre los dos en cuanto al más eficiente. Pues aunque para muchos casos Shell tiene una eficiencia similar a la de Mergesort, no siempre garantiza una eficiencia de O(n·$\log_2$(n)); es decir, a medida que el tamaño aumenta, Mergesort tiende a presenta mejores tiempos (es dominado asintóticamente por Shell).

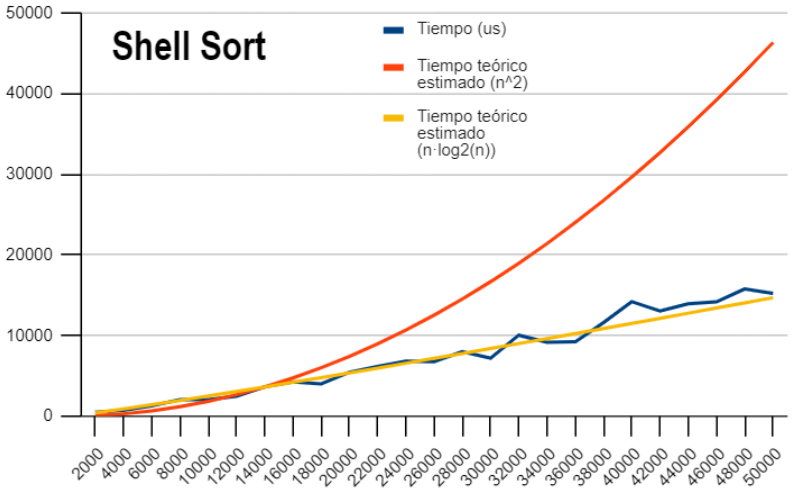


Con respecto al algoritmo de inserción sobre el que está basado, puesto que Shell es una "versión mejorada" gracias a las brechas que consiguen reducir el número de operaciones en la mayoría de casos, es evidente que Shell tiene una eficiencia mayor (o igual en el peor de los casos) que su predecesor.

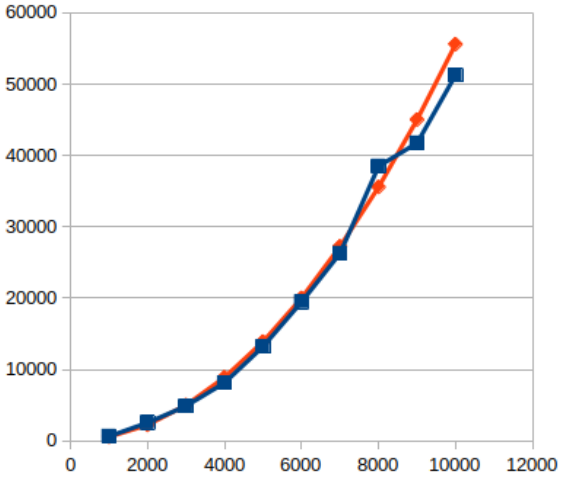
6. Comparación general de algoritmos



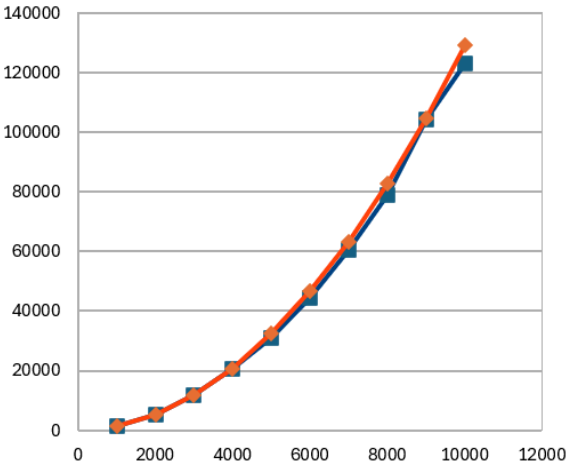
Quicksort: O(n·log(n))



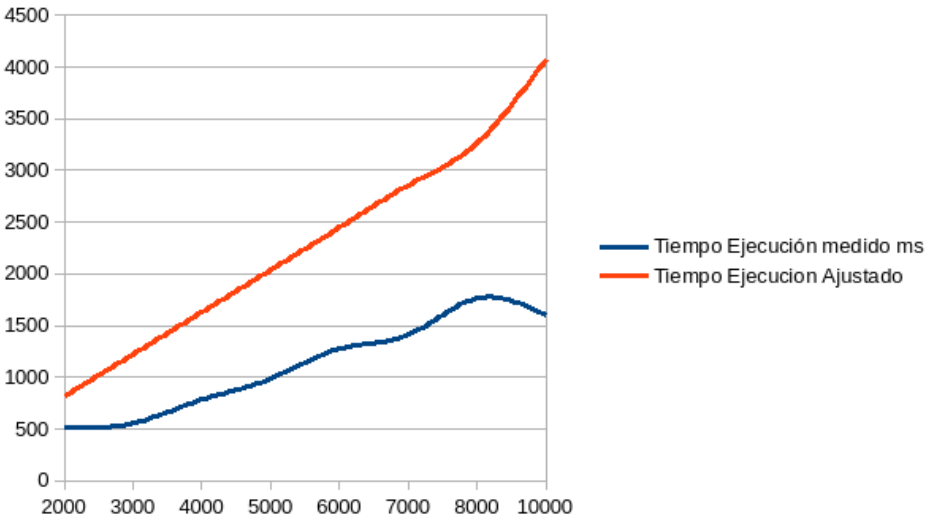
Shell Sort: Teórico: O(n²); Práctico: O(n·$\log_2$(n))



Inserción: O(n²)



Selección: O(n²)



Counting Sort: O(n+k)

Tras el estudio de la eficiencia de estos cinco algoritmos de ordenación, vamos a comparar cuáles son los mejores y cuales ofrecen peores resultados.

Empezando por los que tienen un orden de eficiencia mayor, nos encontramos con los algoritmos de **inserción** y de **selección** cuyo orden de eficiencia teórico es **$O(n^2)$**. Sin embargo, si vemos los resultados obtenidos en el cálculo de la eficiencia práctica e híbrida, vemos que la constante del algoritmo de inserción es mucho menor que la del algoritmo de selección por lo que podemos decir que el algoritmo de inserción es más eficiente que el de selección.

Seguidamente, nos encontramos con **Quicksort** y **Shell Sort** (como ya hemos visto tiene un orden teórico de $O(n^2)$ aunque se comporta en la práctica como $O(n \cdot \log_2(n))$) que tienen un orden de eficiencia de **$O(n \cdot \log_2(n))$**. Podemos ver que el más rápido es el Quicksort por lo que podemos decir que el mejor algoritmo de los cinco es el Quicksort.

Por último, tenemos el Counting Sort que tiene una eficiencia de orden **O(n+k)**, que es lineal y por tanto, menor que los anteriores.