Universidad de Granada

ALGORÍTMICA

Práctica 1

Práctica 1 - Cálculo de la eficiencia de algoritmos.

INTEGRANTES
GRUPO B3

Miguel Rodríguez Ayllón
Alberto Cámara Ortiz
Cristóbal Pérez Simón
José Vera Castillo
Pablo Rejón Camacho

ÍNDICE

Preámbulo

Para cada ejecución, hemos utilizado la semilla de números aleatorios 12345, buscando el "*fair play"* entre las diferentes ejecuciones de los algoritmos.

Para el cálculo de la constante oculta en cada apartado, hemos utilizado la siguiente fórmula:

$$K = \frac{T(n)}{f(n)}$$

siendo:
K → *Constante oculta.*
T(n) → *Resultado del tiempo experimental. (**µs**)*
f(n) → *Eficiencia teórica de la función*

Para el cálculo del tiempo teórico estimado necesitamos el $K_{promedio}$ de forma que se obtiene de la siguiente manera: Tiempo teórico estimado = $K_{promedio}$ * f(n).

Para analizar la eficiencia práctica de los algoritmos hemos adaptado el código proporcionado por el profesorado, de manera que cada programa de prueba utiliza la base proporcionada en el material de prácticas empleando variaciones en caso de que los parámetros de las funciones no se ajusten a las implementaciones encontradas.

1 - Conteo (Counting Sort).

```cpp
vector<int> countSort(vector<int>& inputArray)
{
    int N = inputArray.size();

    // Finding the maximum element of array inputArray[].
    int M = 0;

    for (int i = 0; i < N; i++)
        M = max(M, inputArray[i]);

    // Initializing countArray[] with 0
    vector<int> countArray(M + 1, 0);

    // Mapping each element of inputArray[] as an index
    // of countArray[] array

    for (int i = 0; i < N; i++)
        countArray[inputArray[i]]++;

    // Calculating prefix sum at every index
    // of array countArray[]
    for (int i = 1; i <= M; i++)
        countArray[i] += countArray[i - 1];

    // Creating outputArray[] from countArray[] array
    vector<int> outputArray(N);

    for (int i = N - 1; i >= 0; i--) {
        outputArray[countArray[inputArray[i]] - 1] = inputArray[i];
        countArray[inputArray[i]]--;
    }

    return outputArray;
}
```

Referencia de: https://www.geeksforgeeks.org/counting-sort/

Este algoritmo resuelve el problema de ordenación de un vector pasado como parámetro no empleando para ello ningún mecanismo de comparación entre sus componentes.
Es particularmente eficiente cuando el rango de los valores de entrada es más pequeño que el número de elementos del vector, y básicamente consiste en contar con qué frecuencia se repite cada elemento del vector de entrada y posicionarlo en el vector resultado de forma ordenada.

1.1 Análisis de la eficiencia teórica

Variable o variables de las que dependen el tamaño del caso: El tamaño del problema depende de dos valores. El primero es el tamaño del vector de entrada (N) y el segundo es el máximo valor de ese vector (M).

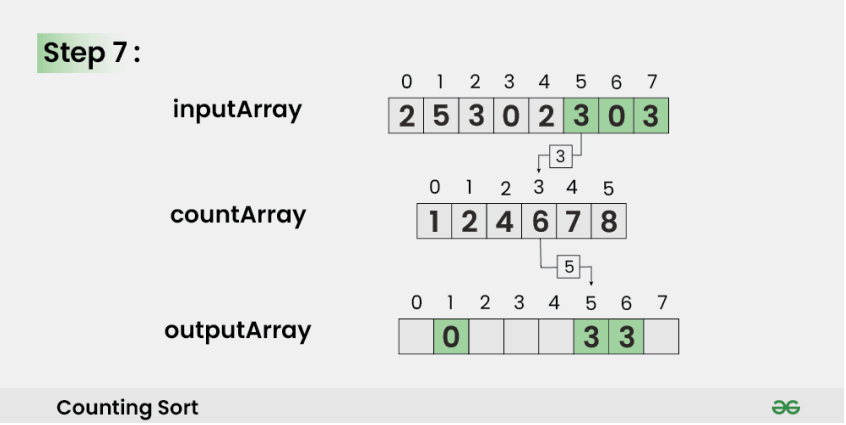
Comenzamos calculando cuál es ese valor M comparando todos los elementos del vector de entrada. Esta comparación en sí es una operación simple, pero como lo estamos calculando para todo el vector, hay que hacerlo en un bucle y por tanto, este primer paso tiene una eficiencia de **O(n)**.

El siguiente paso es crear un vector (que será el vector auxiliar) con “M+1” elementos inicializados a 0. Este será el vector en el que vamos a almacenar la frecuencia con la que se repite cada elemento del vector. Como esta operación también se hace en un bucle, seguiremos empleando una eficiencia de **O(n)**.

A continuación, para poder realizar el paso anterior empleamos la instrucción “countArray[InputArray[i]]++” siendo “countArray” el vector auxiliar y “InputArray” el vector de entrada pasado como argumento a la función.
Esta operación sigue teniendo una eficiencia de **O(n)** puesto que también la hacemos en un bucle.

Seguidamente, este vector auxiliar vuelve a modificarse, de forma que ahora en cada elemento se almacena la suma acumulada del elemento en cuestión con el elemento anterior. Esta operación también se hace en un bucle, que depende directamente de la variable M, así que también requiere de una eficiencia de **O(n)**.

Finalmente, crearemos el vector resultado, en el que estarán almacenados todos los elementos del vector original ordenados.
Esta operación consiste en ir recorriendo el vector original desde el final y buscar en qué posición se encuentra ese elemento en el vector auxiliar una vez que este ya contiene el sumatorio de todos los valores. Vamos a coger un caso concreto del problema, por ejemplo i=5:



Como vemos, el valor correspondiente a i=5 es el valor 3, buscamos por tanto cuál es el valor almacenado en countArray[3] y a ese valor que llamaremos k le restamos 1 porque el vector de salida tiene el mismo número de elementos que el vector original. Ya lo que nos queda es almacenar en outputArray[k-1] el contenido de inputArray[i] y decrementar el valor almacenado en “countArray”.

Esta última operación también se hace un un bucle que recorre el vector de origen completo, por tanto, tendrá una eficiencia de **O(n)**.

En conclusión, hemos visto que todas las operaciones del algoritmo son operaciones simples, pero están incluidas en varios bucles que actúan de forma secuencial.

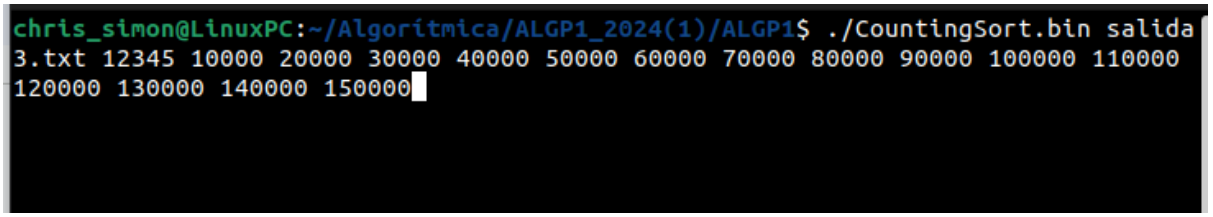
Aplicando la regla que nos dice que el orden de eficiencia de un conjunto de instrucciones será igual al máximo de la suma de los órdenes de eficiencia de cada instrucción tenemos que :

- Hay cuatro bucles que tienen un orden de eficiencia **O(n**) cada uno.
- Por tanto, el orden de eficiencia total será: máx{O(f_1(n), O(f_2(n), O(f_3(n), O(f_4(n)} = **O(n)** siendo f_n el orden de eficiencia de cada bucle.

1.2 Análisis de la eficiencia práctica

A continuación se detalla el método de ejecución incluyendo los distintos tamaños del vector que se van a analizar.

El tiempo obtenido es el tiempo que tarda el algoritmo en resolver el problema de ordenación calculado de forma práctica.



Tamaño del vector	Tiempo (µs)
10000	1108
20000	759
30000	1197
40000	1600
50000	2127
60000	2516
70000	2871
80000	3645
90000	3921
100000	4086
110000	4855
120000	4712
130000	5322

1.3. Análisis de la eficiencia híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado anterior se corresponde con el orden de eficiencia calculado en el primer apartado.
Para ello, hemos de calcular la constante oculta.

1.3.1. Cálculo de la constante oculta

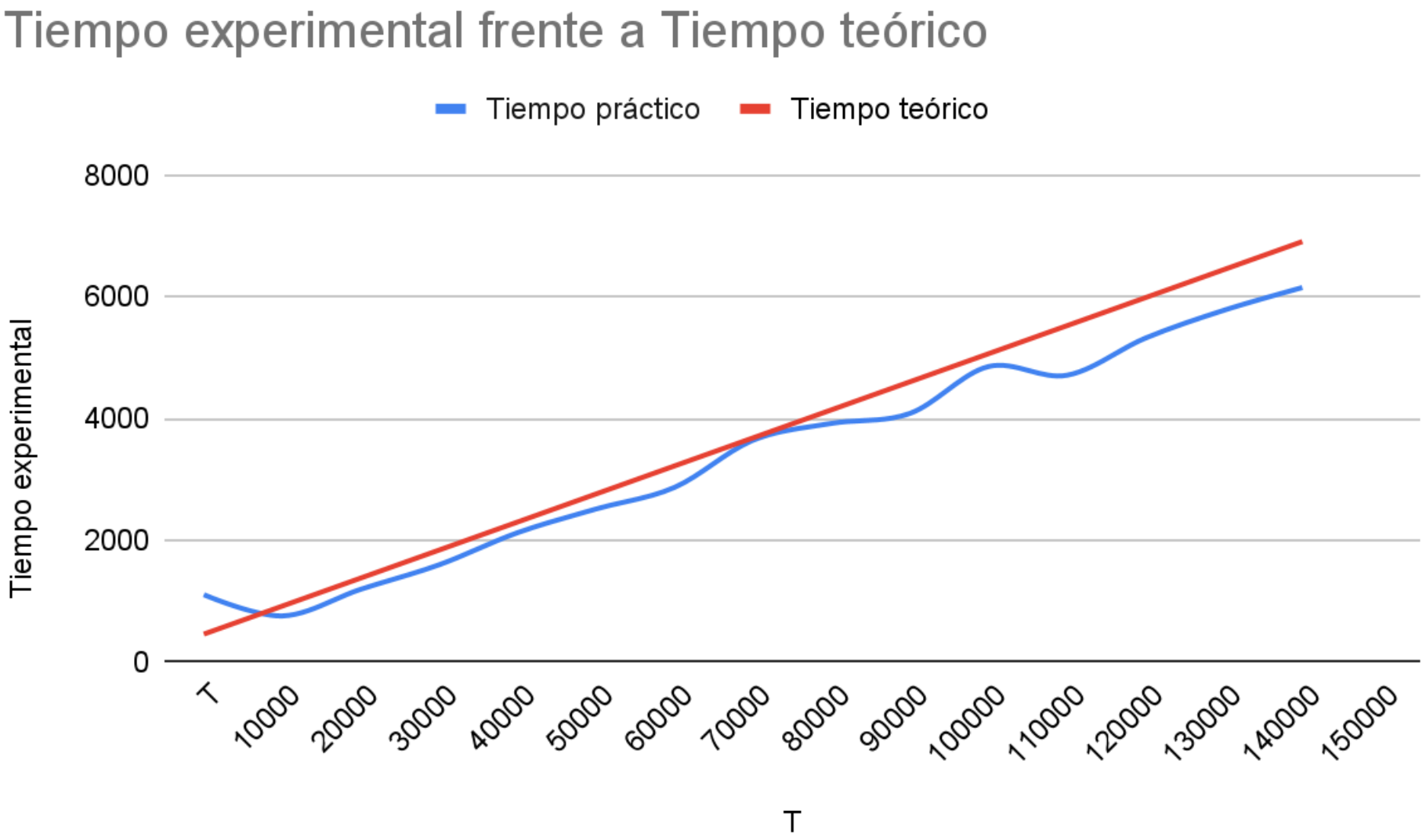
La obtención de la constante oculta K se ha detallado en el preámbulo.
Una vez obtenido este valor para todos los tamaños de caso, calcularemos el promedio, que será utilizado para obtener el tiempo estimado teórico.

Tamaño de caso	Tiempo experimental T(n)	f(n)	K
10000	1108	10000	0,1108
20000	759	20000	0,03795
30000	1197	30000	0,0399
40000	1600	40000	0,04
50000	2127	50000	0,04254
60000	2516	60000	0,04193333333
70000	2871	70000	0,04101428571
80000	3645	80000	0,0455625
90000	3921	90000	0,04356666667
100000	4086	100000	0,04086
110000	4855	110000	0,04413636364
120000	4712	120000	0,03926666667
130000	5322	130000	0,04093846154
140000	5779	140000	0,04127857143
150000	6152	150000	0,04101333333

$$K_{promedio} = 0{,}04680525212$$

1.3.2. Comparación gráfica entre tiempo teórico estimado y el tiempo experimental

Tiempo experimental (µs)	Tiempo teórico estimado (µs)
1108	460,5067882
759	921,0135764
1197	1381,520365
1600	1842,027153
2127	2302,533941
2516	2763,040729
2871	3223,547517
3645	3684,054306
3921	4144,561094
4086	4605,067882
4855	5065,57467
4712	5526,081459
5322	5986,588247
5779	6447,095035
6152	6907,601823
1108	460,5067882
759	921,0135764


Tiempo experimental frente a Tiempo teórico
Tiempo práctico
Tiempo teórico
8000
6000
4000
2000
0
Tiempo experimental
T
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
T

2 - Inserción (Insertion Sort).

```
// Function to sort an array using
// insertion sort
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of their
        // current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Referencia de: https://www.geeksforgeeks.org/insertion-sort/

2.1 Análisis de la eficiencia teórica.

Este algoritmo resuelve el problema de ordenación de un vector con un total de “n” componentes útiles.

Variable o variables de las que dependen el tamaño del caso: El tamaño depende del parámetro entero n, que corresponde a su vez con la longitud del vector a ordenar.

En primer lugar, nos encontramos con tres operaciones elementales, que son las declaraciones de las 3 variables enteras a utilizar. Estas operaciones cuentan con un orden de eficiencia de **O(1)**.

A continuación, nos encontramos dos bucles anidados, el primero de ellos es el que recorre el vector completo.

La asignación, comparación y actualización de este bucle son de eficiencia **O(1)** y, por ello, su orden de eficiencia será (n-1)*(O(comprobación)+O(actualización)+O(sentencias bucle1)).

Con estos datos, podemos llegar a la conclusión de que el orden de eficiencia de este bucle, aplicando la regla del máximo, será:

(n-1)*(O(1)+O(1)+O(sentencias bucle “for”))

Y, por ello, deberemos de estudiar las sentencias que contiene internamente para poder determinar con exactitud el orden de eficiencia de éste.

Una vez analizamos las sentencias que contiene el bucle “for”, nos encontramos con dos líneas con una operación elemental de asignación cada una, “key = arr[i]” y “j = i+1”, cuyo orden de eficiencia es **O(1)** en ambas.

Seguido, nos encontramos un bucle “while”, que irá iterando mientras sus dos condiciones sean ciertas. Ambas condiciones son **O(1)** y, para poder evaluar el peor de los casos, debemos asumir que al bucle se entrará todas las veces posibles.

Para ello, será necesario utilizar como parámetro un vector desordenado u ordenado de manera descendente. Esto conlleva que la segunda condición sea casi siempre cierta y el bucle “while” se ejecuta tantas veces como el número de iteración (variable i) en el que nos encontremos, llegando a la conclusión de que cuenta con una eficiencia **O(i)**.

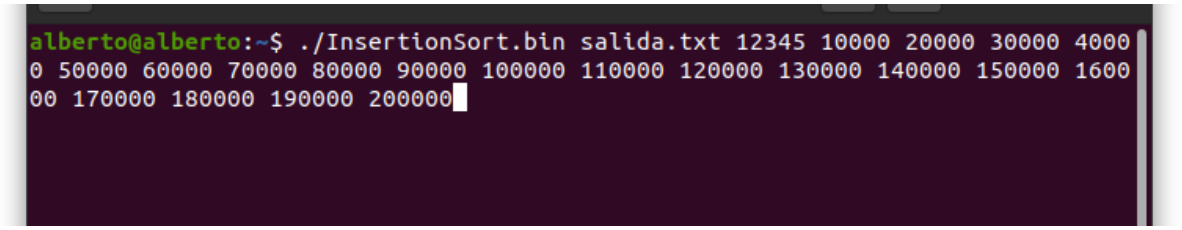
Además, nos encontramos de nuevo dos operaciones elementales, que son dos operaciones de asignación simples cuya eficiencia es **O(1)**. Es por ello que aplicando la regla del máximo, el orden de eficiencia del bucle “while” será **O(n)**.

Una vez finalizado este bucle “while”, y aún dentro del bucle “for”, nos encontramos una operación de asignación para ajustar el valor de “key” dentro del vector y dejar éste ordenado de manera creciente. Esta operación es **O(1)**.

Por lo tanto, la eficiencia de las sentencias encontradas dentro del bucle “for” será, aplicando la regla del máximo, de **O(n)**, dando como resultado final un orden de eficiencia cuadrático **O(n^2)**.

2.2 Análisis de la eficiencia práctica.

Dado que este algoritmo tiene un orden de eficiencia de **O(n²)**, no podemos usar tamaños muy grandes debido a que el tiempo de ejecución se dispararía y el programa podría entrar en un bloqueo (temporal).



Tamaño del vector	Tiempo (µs)
10000	67578
20000	243111
30000	536108
40000	956958
50000	1504963
60000	2186900
70000	2910956
80000	3805208
90000	4911218
100000	6621182
110000	7683671
120000	10186198
130000	11398453
140000	12458095
150000	13965480
160000	17632337
170000	17884617
180000	19753026
190000	22195323
200000	27206061

2.3 Análisis de la eficiencia híbrida.

2.3.1 Cálculo de la constante oculta.

Para el cálculo de la eficiencia híbrida, es necesario realizar el cálculo de la/s constante/s ocultas.
Para ello, elegiremos diferentes tamaños de casos y compararemos los valores obtenidos durante el estudio de la eficiencia teórica y práctica.

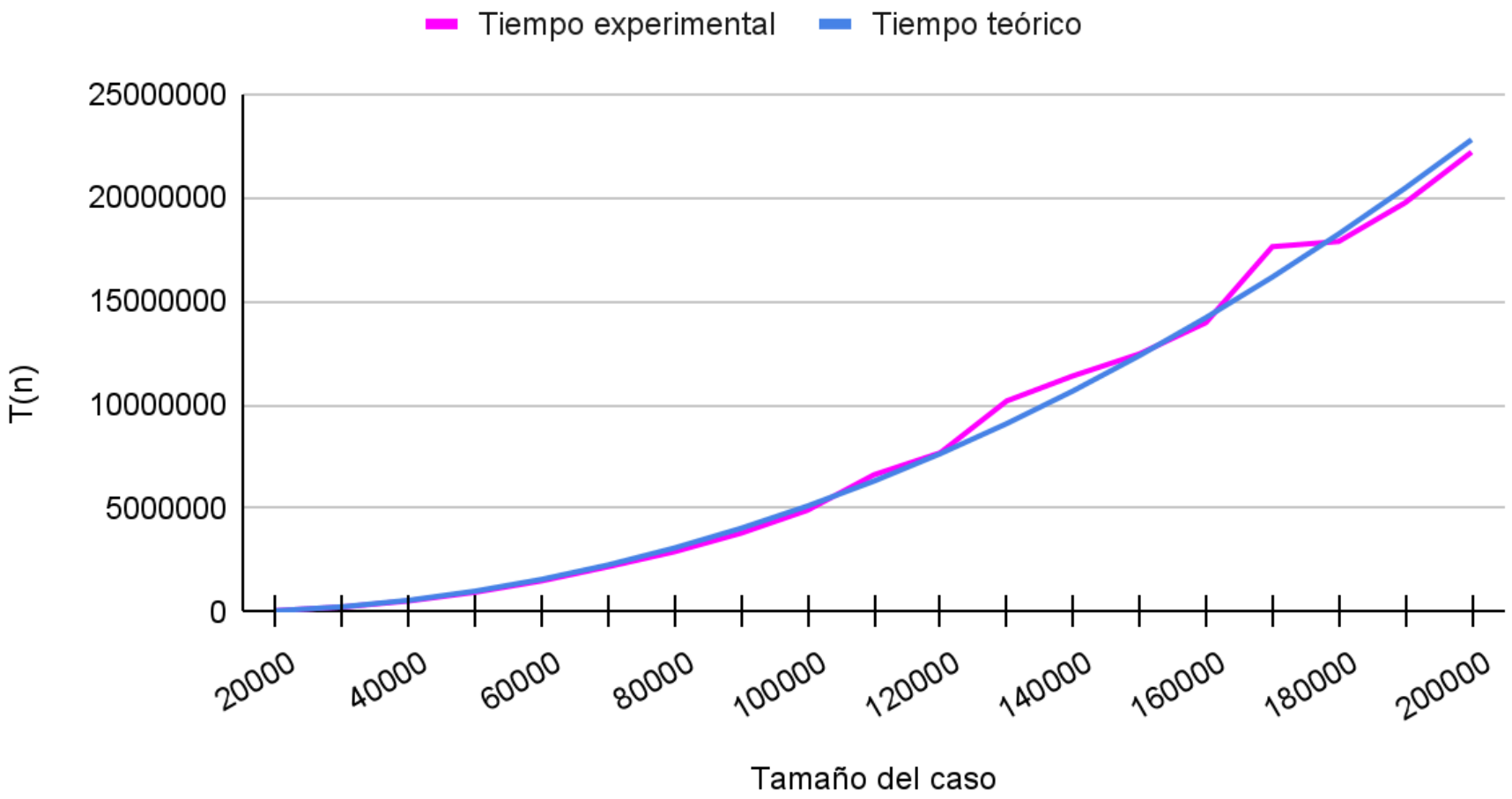
Tamaño de caso	Tiempo experimental T(n)	f(n)	K
10000	67578	100000000	0,00067578
20000	243111	400000000	0,000607777
30000	536108	900000000	0,000595676
40000	956958	1600000000	0,000598099
50000	1504963	2500000000	0,000601985
60000	2186900	3600000000	0,000607472
70000	2910956	4900000000	0,000594073
80000	3805208	6400000000	0,000594564
90000	4911218	8100000000	0,000606323
100000	6621182	10000000000	0,000662118
110000	7683671	12100000000	0,000635014
120000	10186198	14400000000	0,000707375
130000	11398453	16900000000	0,000674465
140000	12458095	19600000000	0,000635617
150000	13965480	22500000000	0,000620688
160000	17632337	25600000000	0,000688763
170000	17884617	28900000000	0,000618845
180000	19753026	32400000000	0,000609661
190000	22195323	36100000000	0,000614829
200000	27206061	40000000000	0,000680152

$$K_{promedio} = 0{,}000631464$$

2.3.2 Comparación gráfica entre tiempo teórico estimado y el tiempo experimental.

Tiempo experimental (µs)	Tiempo teórico estimado (µs)
67578	63146,38
243111	252585,52
536108	568317,42
956958	1010342,08
1504963	1578659,5
2186900	2273269,68
2910956	3094172,62
3805208	4041368,32
4911218	5114856,78
6621182	6314638
7683671	7640711,98
10186198	9093078,72
11398453	10671738,22
12458095	12376690,48
13965480	14207935,5

Tiempo Teórico vs Tiempo Práctico



3 - Rápido (Quick sort).

```
4       int partition(int arr[],int low,int high){
5               //choose the pivot
6
7               int pivot=arr[high];
8               //Index of smaller element and Indicate
9               //the right position of pivot found so far
10              int i=(low-1);
11
12              for(int j=low;j<=high;j++)
13              {
14                      //If current element is smaller than the pivot
15                      if(arr[j]<pivot)
16                      {
17                              //Increment index of smaller element
18                              i++;
19                              swap(arr[i],arr[j]);
20                      }
21              }
22              swap(arr[i+1],arr[high]);
23              return (i+1);
24      }
25
26      // The Quicksort function Implement
27
28      void quickSort(int arr[],int low,int high)
29      {
30              // when low is less than high
31              if(low<high)
32              {
33                      // pi is the partition return index of pivot
34
35                      int pi=partition(arr,low,high);
36
37                      //Recursion Call
38                      //smaller element than pivot goes left and
39                      //higher element goes right
40                      quickSort(arr,low,pi-1);
41                      quickSort(arr,pi+1,high);
42              }
43      }
```

Referencia de: https://www.geeksforgeeks.org/quick-sort/?ref=shm

El algoritmo resuelve el problema de ordenación de un vector v, con un total de ”n” componentes útiles. Para ello se va escogiendo aleatoriamente un pivote (en este caso el último elemento del vector) y se sitúan a la izquierda de este los elementos menores que él y a la derecha los mayores. Después, se vuelve a aplicar el algoritmo sobre los subvectores resultantes de la partición.

3.1 Análisis de la eficiencia teórica

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 a ordenar. Por tanto, n=posFin-posIni+1.

Es un algoritmo recursivo. Por tanto, llamaremos T(n) al tiempo de ejecución que tarda el algoritmo “QuickSort” en resolver un problema de tamaño “n”. Según el algoritmo, existen dos casos (uno general y un caso base):

- **Caso base:** Cuando low >= high. Es decir, cuando solo tenemos un elemento para analizar en el vector (n=1). Entra al “if” de la línea 31 y termina. Sólo se realiza una comparación. Según las reglas de análisis de eficiencia, en este caso base de la recurrencia el tiempo de ejecución es **O(1)**.

- **Caso general:** Es el que interesa. Se da cuando low < high, es decir, cuando n>1 . En este caso:

    - El algoritmo escoge con la función partition() un pivote a partir del cuál dividirá el vector en dos subvectores (línea 35). Esta función tiene un bucle “for” entre las líneas 12 y 21 que siempre se ejecuta n veces, siendo n el número de componentes del vector o subvector sobre el que se está aplicando el algoritmo por lo que podemos asumir que la eficiencia de ese bucle es de orden **O(n)**.

    - Tanto dentro como fuera del bucle, el resto de líneas dentro de la función son de orden **O(1)** por lo que podemos asumir que la llamada a la función partition es de orden **O(n)**.

    - Después, la función quicksort hace una llamada recursiva en la línea 40 para resolver uno de los subvectores de tamaño aproximado (n/2), el que va desde el punto mínimo hasta el anterior al pivote. De esta forma el mínimo sigue siendo el que es (low) y el anterior al pivote se convierte en el nuevo máximo (pi-1=high) para la siguiente llamada recursiva.
      Esta llamada tendrá un tiempo de ejecución aproximado de **T(n/2)**.

    - Justo después hace otra llamada recursiva para el otro subvector. Esta vez entre el siguiente al pivote y el máximo. De esta forma el siguiente al pivote se convierte en el nuevo mínimo (pi+1= low) y el máximo sigue siendo el que es (high) para la nueva llamada recursiva. Esta llamada tendrá un tiempo de ejecución aproximado de **T(n/2)**.

Por tanto, en el caso general el T(n) aproximado puede expresarse como T(n) = n + 2T(n/2) por la llamada a la función “partition” de **O(n)** y las dos llamadas recursivas a quickSort que tardan T(n/2) aproximadamente.
Para obtener el orden de eficiencia de quickSort hay que resolver la ecuación de recurrencias de la función.

3.1.1 Resolución de la ecuación de recurrencias T(n) = n + 2T(n/2)

El primer paso para resolver esta ecuación es realizar un cambio de variable para facilitar la resolución. Este cambio de variable será $n= 2^m$. De esta forma nos quedaría:

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

Desplazando las T's a la izquierda obtenemos una ecuación lineal no homogénea de la siguiente forma:

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

Parte homogénea de la ecuación:

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

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

Parte homogénea del polinomio característico: $P_m(x) = (x - 2)$

Parte no homogénea de la ecuación:

Para resolver la "parte no homogénea", debemos conseguir un escalar b_1 y un polinomio $q_1(m)$, de modo que nos queda:

$$2^m = b_1{}^m q_1(m)$$

Es fácil haciendo $b_1=2$ y $q_1(m)=1$, donde el grado del polinomio es $d_1=0$ El polinomio característico se obtiene como:

$$P(x) = P_H(x)(x-b_1)^{d1+1} = (x-2)(x-2) = (x-2)^2$$

De este modo, tenemos una única raíz con valor $R_1 = 2$ y multiplicidad $M_1= 2$ Aplicando la fórmula de la ecuación característica, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

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

Ahora deshacemos el cambio de variable del principio para volver al espacio de tiempos inicial:

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

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo es **O(nlog(n))**.

3.2 Análisis de la eficiencia práctica

Dado que este algoritmo de ordenación es de los más rápidos que hay hemos decidido probarlo para tamaños de caso mayores ya que realizarlo con tamaños de caso no lo

suficientemente grandes podría resultar en tiempos de ejecución menores al microsegundo. Además, utilizar valores más grandes hará que los resultados obtenidos sean más evidentes. Los tiempos obtenidos para 20 tamaños de caso distintos son los siguientes:

```
./quicksort tiemposquick.txt 12345 1000000 2000000 3000000 4000000
5000000 6000000 7000000 8000000 9000000 10000000 11000000 12000000
13000000 14000000 15000000 16000000 17000000 18000000 19000000
20000000
```

Tamaño del vector	Tiempo (µs)
1000000	192748
2000000	416320
3000000	606162
4000000	817610
5000000	1029056
6000000	1251576
7000000	1499204
8000000	1697904
9000000	1954101
10000000	2157455
11000000	2413498
12000000	2630412
13000000	2908111
14000000	3126433
15000000	3311231
16000000	3593502
170000	3889391
180000	4068072
190000	4286316
200000	4502798

3.3. Análisis de la eficiencia híbrida

En este apartado comprobaremos que los tiempos obtenidos de forma práctica se corresponden con el orden de complejidad que hemos calculado de forma teórica. Es decir, comparamos el comportamiento de la gráfica que se obtiene con los valores experimentales con la que se obtiene para esos mismos valores si utilizamos la función de complejidad teórica del algoritmo.

3.3.1. Cálculo de la constante oculta

A partir de la tabla anterior obtenemos K para cada caso y calculamos un valor promedio.

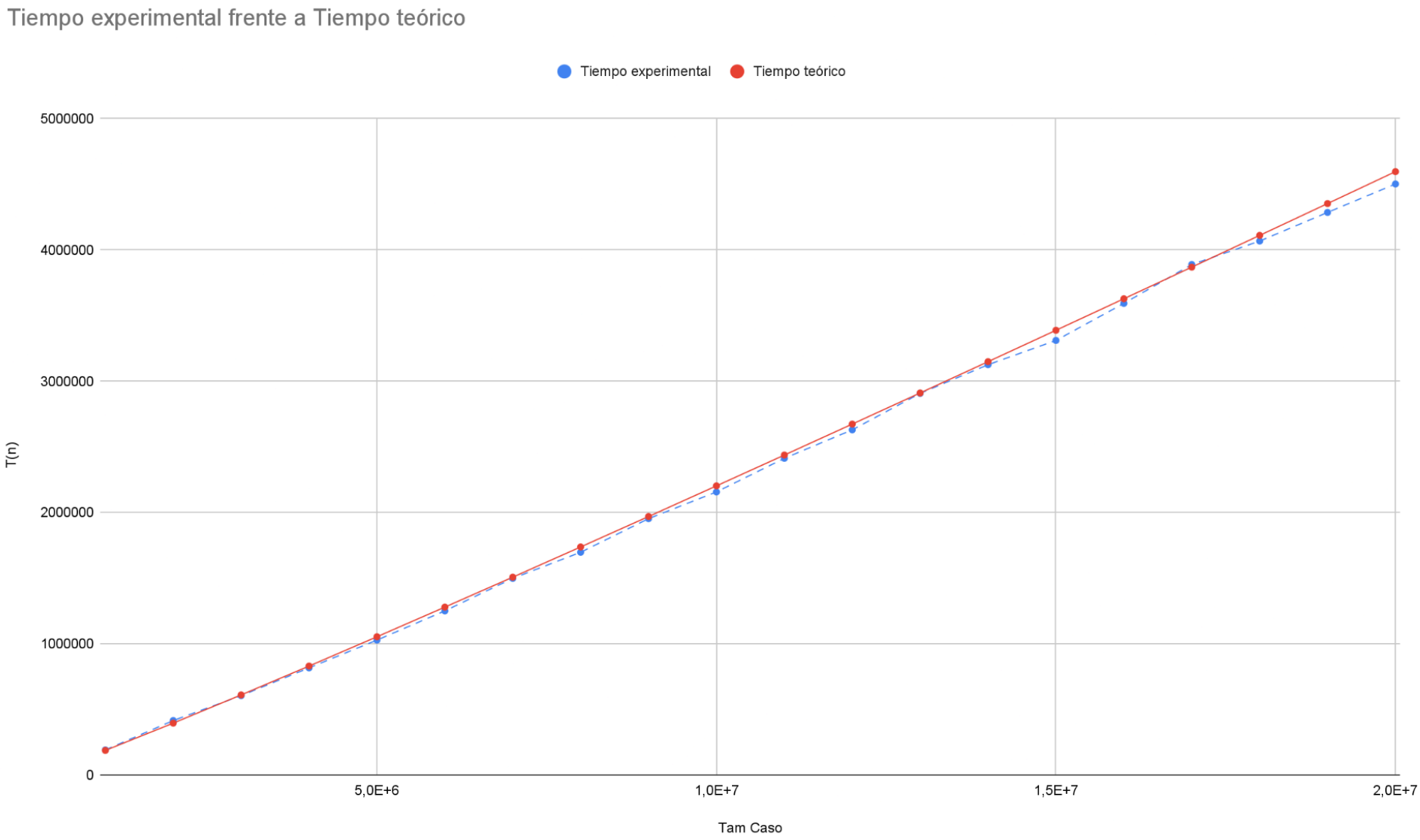
Tamaño de caso n	Tiempo experimental T(n)	f(n)	K
1.000.000	192748	6000000,00	0,009843680858
2.000.000	416320	12602059,99	0,01014941077
3.000.000	606162	19431363,76	0,01014715826
4.000.000	817610	26408239,97	0,009209162987
5.000.000	1029056	33494850,02	0,009653187314
6.000.000	1251576	40668907,50	0,009801071048
7.000.000	1499204	47915686,28	0,009534930093
8.000.000	1697904	55224719,90	0,00950570247
9.000.000	1954101	62588182,58	0,009368906494
10.000.000	2157455	70000000,00	0,01008945412
11.000.000	2413498	77455319,54	0,01245230563
12.000.000	2630412	84950174,95	0,009594170105
13.000.000	2908111	92481263,58	0,009647883877
14.000.000	3126433	100045792,5	0,00932558297
15.000.000	3311231	107641368,9	0,009425646976
16.000.000	3593502	115265919,7	0,00947201005
17.000.000	3889391	122917631,7	0,01113998253
18.000.000	4068072	130594905,1	0,009361144935
19.000.000	4286316	138296318,4	0,0094204507
20.000.000	4502798	146020599,9	0,009418001755

$$K_{promedio} = 0{,}03148069897$$

3.3.2 Comparación gráfica de órdenes de eficiencia

Tiempo experimental (µs)	Tiempo teórico estimado (µs)
192748	188884,1938
416320	396721,657
606162	611712,9132
817610	831349,8527
1029056	1054441,291
1251576	1280285,635
1499204	1508419,296
1697904	1738512,783
1954101	1970319,735
2157455	2203648,928
2413498	2438347,598
2630412	2674290,885
2908111	2911374,819
3126433	3149511,477
3311231	3388625,531
3593502	3628651,72
3889391	3869532,961
4068072	4111218,894
4286316	4353664,769
4502798	4596830,549

Como podemos observar en la gráfica, el tiempo que se ha obtenido con las ejecuciones se corresponde bastante bien con el obtenido de forma teórica por lo que podemos concluir que el cálculo de K ha sido exitoso. Además podemos apreciar que el comportamiento de la función es efectivamente el de una función **O(nlog(n))**.



4 - Selección (Selection Sort).

```
1       // Function for Selection sort
2       void selectionSort(int arr[], int n)
3       {
4               int i, j, min_idx;
5
6               // One by one move boundary of
7               // unsorted subarray
8               for (i = 0; i < n - 1; i++) {
9
10                      // Find the minimum element in
11                      // unsorted array
12                      min_idx = i;
13                      for (j = i + 1; j < n; j++) {
14                              if (arr[j] < arr[min_idx])
15                                      min_idx = j;
16                      }
17
18                      // Swap the found minimum element
19                      // with the first element
20                      if (min_idx != i)
21                              swap(arr[min_idx], arr[i]);
22              }
23      }
```

Referencia de: https://www.geeksforgeeks.org/selection-sort/?ref=shm

Este algoritmo resuelve el problema de ordenación de un vector con un total de “n” componentes útiles. Para ello comprueba el menor elemento del vector y si encuentra un valor menor que este los intercambia dejando así el vector ordenado.

4.1 Análisis de la eficiencia teórica

Variable o variables de las que dependen el tamaño del caso: El tamaño depende del parámetro entero n, que corresponde a su vez con la longitud del vector a ordenar.

- En primer lugar, en la línea 4 nos encontramos una declaración de variables que podemos considerar como tres líneas de eficiencia de orden **O(1)** cada una.

- En la línea 8 nos encontramos con un bucle “for” anidado que analizaremos desde dentro hacia afuera. En la línea 14 hay un “if” dentro del cuál se realiza una asignación si se cumple la comparación. Si el valor de la posición j es menor que el valor de la posición donde está el mínimo actual, el valor almacenado en la posición j pasa a ser el mínimo valor. Si consideramos el peor de los casos, esto ocurrirá todas las veces que se ejecute el bucle. Dado que tanto la comparación del “if” como la asignación de dentro de este son de orden constante **O(1)** podemos afirmar que todo lo que se encuentra dentro del bucle tiene un orden de **O(1)**.

- Ahora bien, ¿cuántas veces se ejecuta dicho bucle? Como podemos observar, parte del siguiente elemento donde esté la i situada en ese momento (j = i+1). Es decir, el

bucle interno iterará cada vez menos veces a medida que el bucle exterior avance. Sin embargo, en el peor de los casos este bucle se ejecutará n-1 veces por lo que aplicando la regla del máximo: (n-1)*(O(comprobación) + O(actualización) + O(sentencias del bucle)) obtenemos que el orden del bucle interno es de O(n-1) que podemos asumir como **O(n)**.

- Aparte del bucle interno, dentro del primer bucle también hay una asignación de **O(1)** que se realiza siempre (línea 12) y otro "if" donde se realiza el intercambio de valores en caso de que el elemento que estemos analizando en ese momento en el bucle no sea el mínimo (líneas 20 y 21). Asumimos que la función "swap" es de orden **O(1)** ya que sólo contiene operaciones elementales y la comprobación del "if" también lo es por lo que el "if" también es de orden **O(1)**.

- Por tanto,aplicando la regla del máximo, obtenemos que dentro del bucle externo el orden de eficiencia es de **O(n)**.

- Como lo que hay dentro del bucle es de orden **O(n)** y el bucle se ejecuta "n" veces tenemos que el bucle anidado tiene un orden de eficiencia **O(n^2)**.

- Aplicando la regla del máximo entre la línea 4 de **O(1)** y el bucle anidado de **O(n^2)** obtenemos que el orden de eficiencia del algoritmo entero es de **O(n^2)**.

4.2 Análisis de la eficiencia práctica

Los tiempos obtenidos para 15 tamaños de caso son los siguientes:

```
./selection tiempos_select.txt 12345 10000 20000 30000 40000 50000
60000 70000 80000 90000 100000 110000 120000 130000 140000 150000
```

Tamaño del vector	Tiempo (µs)	Tiempo(s)
10000	134000	0,134
20000	602146	0,602146
30000	1157209	1,157209
40000	2061964	2,061964
50000	3200835	3,200835
60000	4623469	4,623469
70000	6282596	6,282596
80000	8200406	8,200406
90000	10374502	10,374502
100000	12814406	12,814406
110000	15474790	15,47479
120000	18921664	18,921664
130000	22419029	22,419029
140000	26016016	26,016016
150000	30032832	30,032832

4.3. Análisis de la eficiencia híbrida

En este apartado comprobaremos que los tiempos obtenidos de forma práctica se corresponden con el orden de complejidad que hemos calculado de forma teórica. Es decir, comparamos el comportamiento de la gráfica que se obtiene con los valores experimentales con la que se obtiene para los mismos valores si utilizamos la función de complejidad teórica del algoritmo.

4.3.1 Cálculo de la constante oculta

A partir de la tabla anterior obtenemos K para cada caso y calculamos un valor promedio.

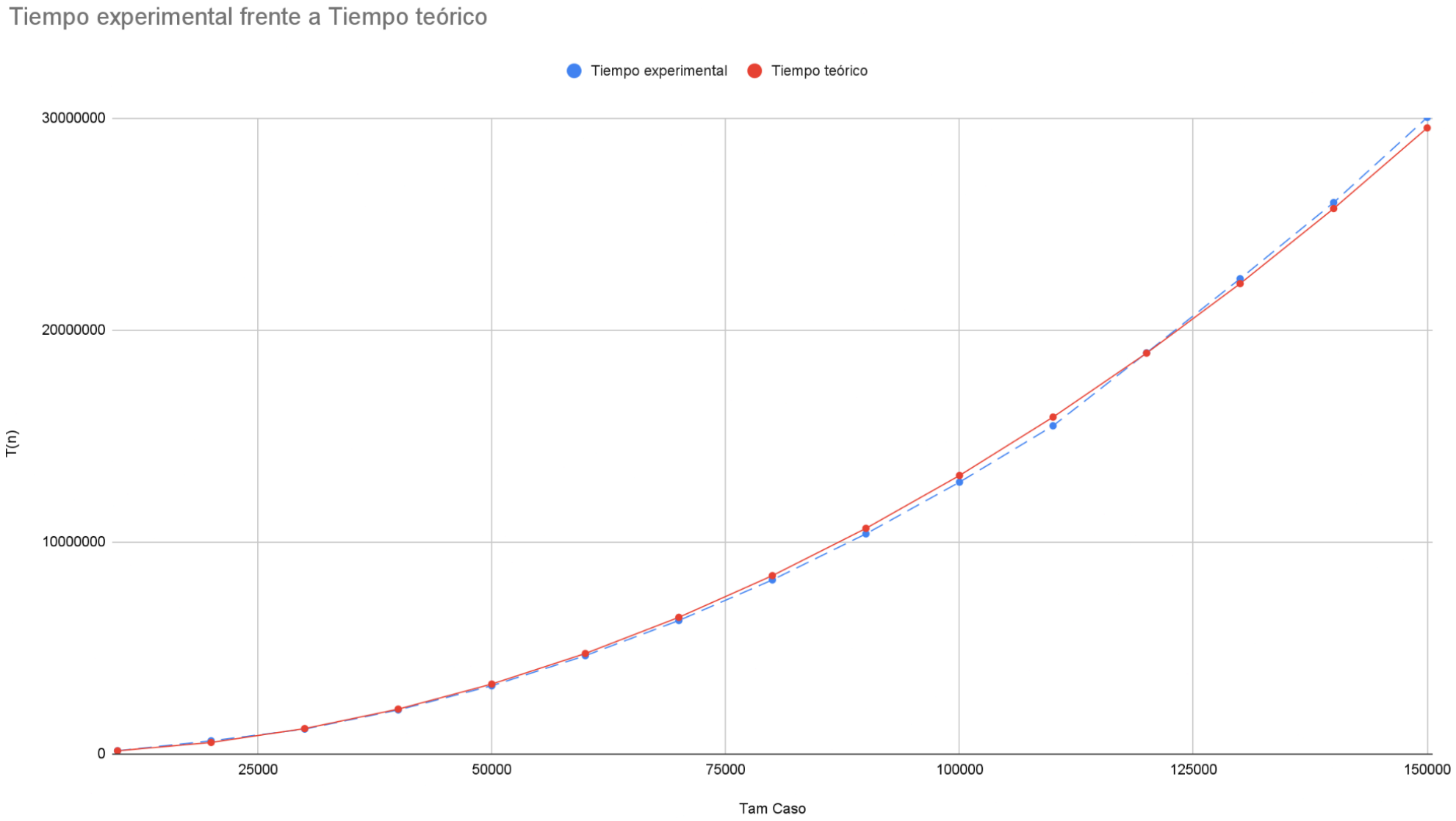
Tamaño de caso	Tiempo experimental T(n)	f(n)	K
10000	134000	100000000,0	0,00134
20000	602146	400000000,0	0,001505365
30000	1157209	900000000,0	0,001285787778
40000	2061964	1600000000,0	0,0012887275
50000	3200835	2500000000,0	0,001280334
60000	4623469	3600000000,0	0,001284296944
70000	6282596	4900000000,0	0,001282162449
80000	8200406	6400000000,0	0,001281313438
90000	10374502	8100000000,0	0,001280802716
100000	12814406	10000000000,0	0,0012814406
110000	15474790	12100000000,0	0,001278908264
120000	18921664	14400000000,0	0,001314004444
130000	22419029	16900000000,0	0,001326569763
140000	26016016	19600000000,0	0,001327347755
150000	30032832	22500000000,0	0,001334792533

$$K_{promedio} = 0{,}001312790212$$

4.3.2 Comparación gráfica de órdenes de eficiencia

Tiempo experimental (µs)	Tiempo teórico estimado (µs)
134000	131279,0212
602146	525116,0849
1157209	1181511,191
2061964	2100464,34
3200835	3281975,531
4623469	4726044,764
6282596	6432672,041
8200406	8401857,359
10374502	10633600,72
12814406	13127902,12
15474790	15884761,57
18921664	18904179,06
22419029	22186154,59
26016016	25730688,16
30032832	29537779,78

Como podemos observar en la gráfica generada el tiempo que se ha obtenido con las ejecuciones realizadas se corresponde bastante bien con el obtenido de forma teórica lo que quiere decir que la constante K se ha calculado correctamente. Además, se puede apreciar la tendencia exponencial de la función lo cual tiene sentido dado que estamos analizando un algoritmo de orden cuadrático **O(n^2)**.



5 - Ordenamiento Shell (Shell Sort).

```
/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location
            arr[j] = temp;
        }
    }
    return 0;
}
```

Referencia de: https://www.geeksforgeeks.org/shellsort/?ref=shm

5.1 Análisis de la eficiencia teórica

Este algoritmo resuelve el problema de ordenación de un vector con un total de “n” componentes útiles.

Variable o variables de las que dependen el tamaño del caso: El tamaño depende del parámetro entero n, que corresponde a su vez con la longitud del vector a ordenar y también depende de la elección de la secuencia de pasos.

Este algoritmo cuenta con tres bucles anidados, en primer lugar nos encontramos un “for”.

La asignación, comparación y actualización de este bucle son de eficiencia **O(1)**, luego la eficiencia será **O(i(n) + g (n) + h(n) · (g (n) + f (n) + a(n))** siendo:

- f(n) Eficiencia del bloque de sentencias.
- g(n) Eficiencia de comprobar la condición.
- h(n) Número de veces que se ejecuta el bucle.
- a(n) Eficiencia de la actualización.
- i(n) Eficiencia de la inicialización

Este algoritmo debido a la reducción exponencial del “gap” se repite “log(n)” veces y teniendo en cuenta que i(n), g(n) y a(n) son **O(1)**. Nos queda la siguiente expresión:

O(1 + 1 + log(n) · (1 + f (n) + 1))

Deberemos estudiar las sentencias que contienen el bucle “for” para poder determinar con exactitud el orden de eficiencia de éste.
Analizando el segundo bucle “for” nos damos cuenta de que al igual que el anterior i(n), g(n) y a(n) son **O(1)**, pero este se repite “n-gap” veces. Luego la expresión queda:

O(1 + 1 + (n-gap)· (1 + f (n) + 1))

Este bucle contiene 4 operaciones elementales que son **O(1)** y un bucle interno.
Analizando el tercer “for” somos capaces de ver que al igual que el anterior i(n), g(n) y a(n) son **O(1)**. La eficiencia de este bucle al igual que el primero depende del “gap” elegido. En el peor de los casos se repite n/gap veces, y como gap está en función de n sería **O(1)**.

Por tanto la expresión queda:

O(1 + 1 + 1· (1 + f (n) + 1))

Este “for” contiene 1 operación elemental así que la eficiencia sería **O(1)**.

Sabiendo esto podemos deducir la eficiencia del bloque de sentencias del segundo “for”, la cual sería **O(1)**. Y cómo esto se repite “n-gap” veces (n para mayor simplicidad), la eficiencia del segundo “for” sería **O(n)**. Por lo tanto el primer “for” tendría una eficiencia en un caso promedio de **O(n*log(n))**.

5.2 Análisis de la eficiencia práctica

A continuación se detalla el método de ejecución incluyendo los distintos tamaños del vector que se van a analizar.

El tiempo obtenido es el tiempo que tarda el algoritmo en resolver el problema de ordenación calculado de forma práctica.

Tamaño del vector	Tiempo (µs)
10000	1889
20000	3086
30000	4596
40000	6695
50000	9017
60000	9584
70000	11801
80000	14132
90000	16225
100000	19447
110000	19078
120000	22475
130000	25551
140000	27547
150000	30097
160000	35086
170000	36103
180000	35866
190000	37347
200000	40899

5.3 Análisis de la eficiencia híbrida.

En esta sección, evaluaremos la correspondencia entre la eficiencia práctica medida en la sección 5.2 y la eficiencia teórica calculada en la sección 5.1 . Partimos de la definición del orden de eficiencia, que establece que cuando el tamaño del caso tiende a infinito, el tiempo de ejecución del algoritmo T(n) está limitado por la función de orden de eficiencia O(f(n)), lo que se expresa como T(n) <= K·f(n), donde K es una constante real positiva, conocida como la constante oculta.

5.3.1. Cálculo de la constante oculta

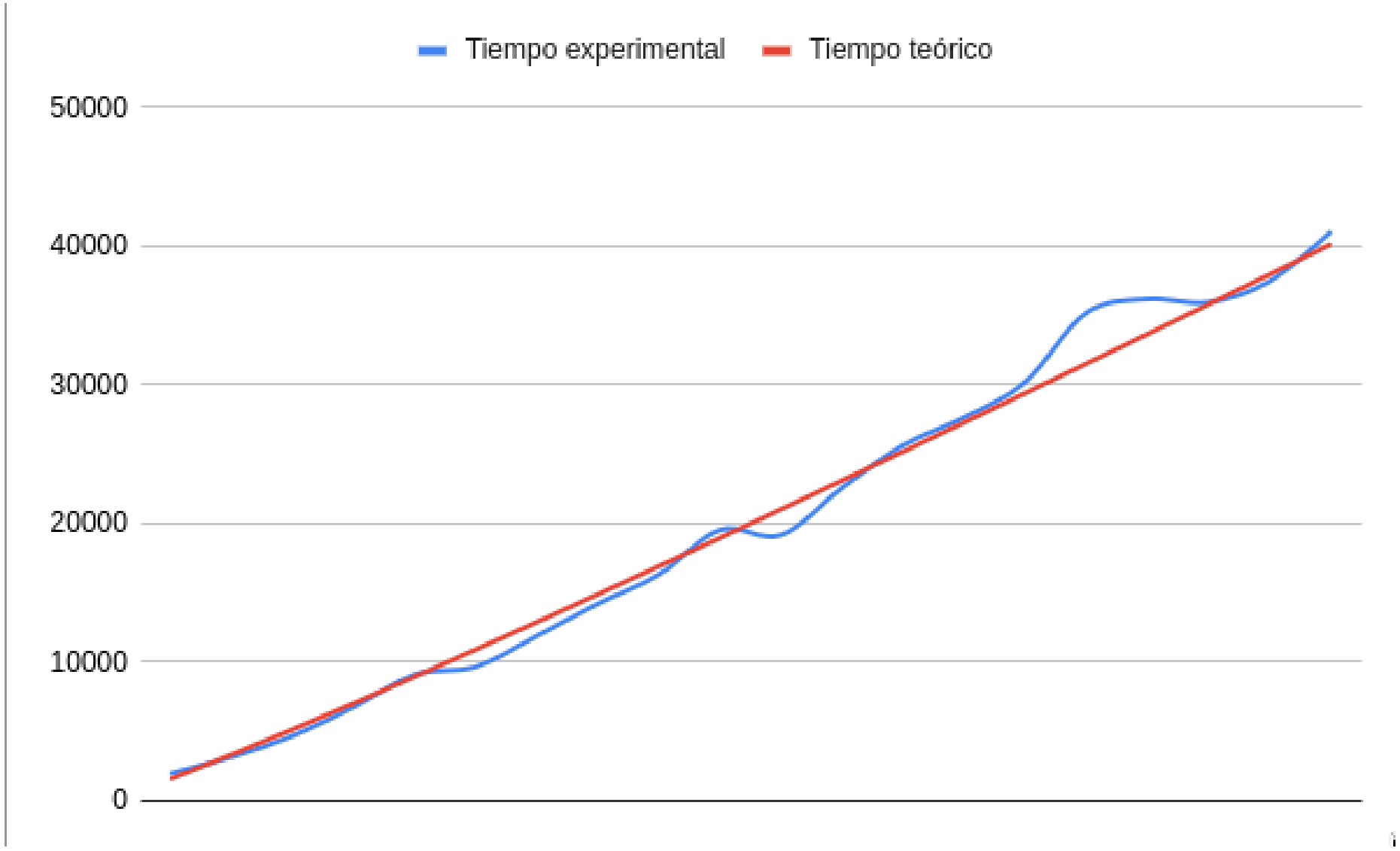
Se calculará el valor de K para todas las ejecuciones del mismo algoritmo con diferentes tamaños de caso, lo que generará valores aproximados para K. El valor final de K se aproxima a la media de todos estos valores. Las gráficas siguientes ilustran un ejemplo del cálculo de este valor, basado en los resultados previamente obtenidos para los algoritmos de ordenación por burbuja y MergeSort en secciones anteriores. En el caso del algoritmo de burbuja, donde f(n) = n^2, y para el algoritmo de MergeSort, donde f(n) = n*log(n). El valor final de K será el promedio obtenido de todas las ejecuciones.

T	Tiempo experimental T(n)	k	f(n)
10000	1889	0,01421614155	132877,1238
20000	3086	0,01079948951	285754,2476
30000	4596	0,01030076978	446180,2464
40000	6695	0,01094833523	611508,4952
50000	9017	0,01155311683	780482,0237
60000	9584	0,01006341619	952360,4928
70000	11801	0,0104743715	1126654,711
80000	14132	0,01084559918	1303016,99
90000	16225	0,0109540497	1481187,364
100000	19447	0,01170826065	1660964,047
110000	19078	0,01035617558	1842185,84
120000	22475	0,01110029488	2024720,986
130000	25551	0,01156960173	2208459,773
140000	27547	0,01151000357	2393309,422
150000	30097	0,01166916543	2579190,446
160000	35086	0,01268458748	2766033,981
170000	36103	0,01222264441	2953779,788
180000	35866	0,01141366104	3142374,729
190000	37347	0,01120935187	3331771,58
200000	40899	0,0116126732	3521928,095

$K_{promedio}$ = 0,01136058547

5.3.2. Comparación gráfica entre tiempo teórico estimado y el tiempo experimental

Tiempo experimental (µs)	Tiempo teórico estimado (µs)
1889	1509,561921
3086	3246,335552
4596	5068,868822
6695	6947,094522
9017	8866,732734
9584	10819,37277
11801	12799,45714
14132	14803,03588
16225	16827,15564
19447	18869,52402
19078	20928,30968
22475	23002,0158
25551	25089,39599
27547	27189,39624
30097	29301,1135


Tiempo experimental
Tiempo teórico
50000
40000
30000
20000
10000
0