GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023-2024

UNIVERSIDAD
DE GRANADA

ALGORÍTMICA

Práctica 1 - Análisis de eficiencia

Miguel Martinez Azor

Ángel Rodríguez Faya

Alejandro Botaro Crespo

Alberto Parejo Bellido

Alejandro Ocaña Sánchez

12/03/2024

ÍNDICE1. Cálculo de la eficiencia teórica

1.1- Algoritmo Counting Sort

Un algoritmo de Counting Sort es un algoritmo de ordenación basado en el conteo del número de elementos distintos de un vector. Su funcionamiento es el siguiente:

-Se recorre el vector buscando el elemento máximo y mínimo.

-Se crea un vector auxiliar desde la componente menor a la componente mayor.

-Se recorre el vector de nuevo y se va anotando las ocurrencias de cada elemento que está en el vector.

-Por último se recorre el vector auxiliar y se genera un vector con los elementos y sus ocurrencias ordenadas.

Este algoritmo depende del tamaño del vector a ordenar y del número de ocurrencias de cada elemento del vector. Lo bueno de este algoritmo es que su eficiencia es independiente de lo ordenado o desordenado que este el vector inicial ya que la ordenación se realiza de manera muy "secuencial" realizando varios pasos que organizan el vector. Quizás se podría decir que el peor caso que se podría dar sería que que un vector de n componentes en el cual no se repite ninguna componente ya que se crearán varias estructuras de datos innecesarias como por ejemplo el vector auxiliar, o que el valor de elemento máximo - mínimo sea grande. Por lo tanto se podría decir que este algoritmo tendría una eficacia de O(n) ya que solo se recorre cada vector una vez, pero realmente en el peor de los casos su eficiencia será O(n+k), siendo k la diferencia entre el máximo elemento y el mínimo.

```cpp
void countingSort(vector<int>& arr) {
    // Encontrar el valor máximo en el vector
    int max_val = arr[0];
    for (int i = 1; i < arr.size(); ++i) {  // O(n)
        if (arr[i] > max_val) {  // O(1)
            max_val = arr[i];  // O(1)
        }
    }

    // Crear un vector de conteo y inicializarlo con ceros
    vector<int> count(max_val + 1, 0);  // O(1)

    // Contar la frecuencia de cada elemento en el vector original
    for (int i = 0; i < arr.size(); ++i) {  // O(n)
        count[arr[i]]++;  // O(1)
    }

    // Actualizar el vector de conteo para almacenar la posición real de cada elemento
    for (int i = 1; i <= max_val; ++i) {  // O(k)
        count[i] += count[i - 1];  // O(1)
    }

    // Crear un vector de salida del mismo tamaño que el vector original
    vector<int> output(arr.size());  // O(n)

    // Ordenar los elementos en el vector de salida usando el vector de conteo
    for (int i = arr.size() - 1; i >= 0; --i) {  // O(n)
        output[count[arr[i]] - 1] = arr[i];  // O(1)
        count[arr[i]]--;  // O(1)
    }

    // Copiar los elementos ordenados al vector original
    for (int i = 0; i < arr.size(); ++i) {  // O(n)
        arr[i] = output[i];  // O(1)
    }
}
```

Analizando el funcionamiento del algoritmo se puede determinar su eficiencia teórica:

- A la inicialización de valores y vectores le corresponde un orden de complejidad O(1).
- El primer bucle que recorre el vector inicial buscando el elemento máximo y mínimo tendría una complejidad de O(n) ya que es necesario recorrer todo el vector para comprobar que elemento es el mayor.
- Una vez que se conoce el rango en el que se encuentran todas las componentes del vector, se crea el vector auxiliar (O(1)) y se recorre rellenando las componentes desde el valor mínimo hasta el valor máximo. El vector inicial puede tener por ejemplo 10 componentes que se corresponde con n pero el vector auxiliar podría ser que el valor mínimo del vector sea 10 y mayor 50 por ejemplo, entonces este bucle recorre el vector 50 - 10 = 40 componentes, valor al que llamamos k ya que es diferente al número de componentes inicial del vector. De ahí que sea una complejidad O(k).
- Se crea un vector final el cual representará el resultado de la ordenación (O(1)) y se recorre rellenándolo con los elementos ordenados con respecto al vector inicial. Por lo tanto será O(n).
- Finalmente se vuelve a recorrer el vector inicial y se van copiando los valores correctos y ordenados, donde le corresponde un complejidad de O(n).

Concluimos aplicando la regla del máximo que los órdenes de eficiencia serán O(n) cuando se refiere al vector inicial y O(k) cuando se utiliza el vector auxiliar, por lo tanto el orden de eficiencia como se ha indicado anteriormente se corresponde con O(n+k) aunque a efectos prácticos se podría decir que es O(n).

1.2- Algoritmo Insertion Sort

El algoritmo de ordenación por inserción es un algoritmo de fácil aplicación que permite ordenar un vector v con un determinado tamaño de elementos útiles, que vendrá dado por la variable n.

Su funcionamiento consiste en recorrer el vector, seleccionando en cada iteración un valor como clave y comparándolo con el resto e insertándose en el lugar correspondiente.

El tamaño del caso depende de la variable n, que representa el número de elementos en el vector que se va a ordenar.

Su código en C++ es el siguiente:

```
74  void insertionSort(int *v, int n) {
75      for (int i = 1; i < n; ++i) {
76          int clave = v[i];
77          int j = i - 1;
78
79          while (j >= 0 && v[j] > clave) {
80              v[j + 1] = v[j];
81              j = j - 1;
82          }
83
84          v[j + 1] = clave;
85      }
86  }
```

A continuación, realizaremos el análisis para obtener la eficiencia teórica de este algoritmo.

Comenzamos por las líneas 80 y 81, que corresponden con la parte más interna de la función. Tenemos dos accesos, dos asignaciones básicas y una operación aritmética. Todas son operaciones elementales, por lo que serían del orden (O(1)), y siguiendo la regla del máximo, esta secuencia de operaciones también es (O(1)). Esto significa que este trozo de código es constante cuando lo acotamos asintóticamente (para n → infinito), lo que significa que el tiempo de ejecución no cambia independientemente del tamaño del vector v.

Ahora nos vamos al bucle "while" que se encuentra desde la línea 79 hasta la 82. La condición que tiene dentro será del orden (O(1)), puesto que hay dos comparaciones simples y un acceso a un vector, y como hemos visto antes, son operaciones elementales. Dentro del bucle ya lo hemos calculado anteriormente, que también sería (O(1)). Sólo nos falta ver cuántas veces se ejecutará el bucle en el peor de los casos, que será siempre que se cumpla la condición que tiene. Esa condición se cumplirá siempre y cuando el vector esté desordenado, por lo que podemos observar que el bucle se ejecutará i veces en el peor de los casos. Por lo tanto, la eficiencia del bucle "while" es O(i).
Vamos a seguir revisando el código y cuando lleguemos al bucle "for" de la línea 75, haremos este bucle "while" para darnos cuenta de una cosa.

Bien, en las líneas 76, 77 y 84 tenemos más operaciones elementales (declaraciones, asignaciones, accesos a elementos del vector, etc), que todas ellas recordemos que tenían una eficiencia en el peor de los casos de (O(1)).

En el bucle "for" de la línea 75 tenemos que: la declaración de i, su inicialización, la comprobación y la actualización, tienen todas una eficiencia de (O(1)). Por lo que hay que ver qué eficiencia se ejecutará el bucle en el peor de los casos, que será desde i = 1, hasta n-1.

Cómo el bucle "while" y el bucle "for" están anidados, tenemos que hacer la regla del producto. Esta regla decía que, si tenemos dos trozos de códigos dependientes con tiempos de ejecución T1(n) y T2(n) (siendo los tiempos de ejecución del bucle for y while respectivamente) y el orden de la eficiencia de los dos sería O(f(n)) y O(g(n)) respectivamente, nos quedaría que el código total tendrá la siguiente eficiencia = T1(n) * T2(n) = O(f(n) * g(n)).

Pues si calculamos sus órdenes siguiendo dicha regla, y dándole la vuelta al bucle "while" para que nos sea más fácil verlo, dónde el bucle se ejecutará desde j = 0 hasta i – 1, qué es lo mismo (pero visto de forma más fácil) nos quedaría lo siguiente:

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} = \sum_{i=1}^{n-1} \text{i} = 1 + 2 + 3 + \ldots + \text{n-1 (*P. aritmética)} = n * \frac{(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

** Cómo es una progresión aritmética, todo esto sumaría: el último (n-1) más el primero (1) más uno (+1), que daría en total: n. Eso multiplicado por el número de elementos que hay en total (n-1) entre 2. Esto es de lo que quería que nos diésemos cuenta cuando estábamos en la parte del bucle while.*

Nos quedaría que **el algoritmo Insertion Sort tiene una eficiencia en el peor de sus casos de O(n²)**. Cabe destacar, que en el mejor de los casos (el vector esté ya ordenado) tendrá una eficiencia de O(n), a comparación de otros algoritmos de ordenación como el de burbuja o el de selección (Selection Sort), que en estos siempre será de O(n²). En el caso del Insertion Sort se debe a que, en caso de que ya esté ordenado, sólo realizará el bucle “for”, pero no se ejecutará en absoluto el bucle “while”, por lo que el tiempo de ejecución es proporcional al número de elementos.

1.3- Algoritmo Quick Sort

Quicksort es un algoritmo de ordenación que sigue el famoso paradigma de divide y vencerás. Su funcionamiento es el siguiente:

1. Selección de un pivote, elemento del array.
2. División del array en dos subarrays, uno con los elementos menores que el pivote y el segundo con los mayores.
3. Recursivamente se dividen los subarrays en subarrays para ordenar el array.

El algoritmo de Quicksort tardará más o menos dependiendo de la elección del pivote y del tamaño del array. En el mejor caso, el pivote será el elemento medio del vector, es decir, aquel elemento que deje al hacer la partición la misma cantidad de elementos tanto a su izquierda como a su derecha. Por otro lado, el peor caso, es donde el pivote es el menor o mayor elemento del array.

Analizando el algoritmo, nos encontramos que inicializar variables tiene un costo de O(1), en el Quicksort utilizado en está práctica se ha utilizado la selección de pivote, donde este será el elemento más a la derecha del array. Por ello está operación es O(1). La función partición compara cada elemento con el pivote y realiza un swap en el caso de que sea necesario. La función partición tiene complejidad O(n).
Por último nos metemos en el algoritmo quicksort que se aplica recursivamente en los sub-arrays que se van generando. el peor caso caso como se comentaba anteriormente era escoger el mayor o el menor elemento del array recursivamente, es decir, que nos toparemos con un subarray n-1, que conlleva un tiempo O(n) en la partición, en la recursión el tiempo de ejecución finalmente sería O(n²) en el peor caso.

No obstante, Quicksort tiene una complejidad promedia mucho mejor O(n*log(n)). Ya que lo común sería que los sub-arrays estuviesen más equilibrados, desde el punto de vista analitico esto lleva a log(n) niveles de recursión , donde cada uno tendría un tiempo de ejecución O(n).

En conclusión, la partición es lo que marca la eficiencia del algoritmo, siendo el promedio O(n*log(n)) y el peor O(n²).

```c
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
int partition(int v[], int izq, int der) {

  // Seleciono elemento al final como pivot
  int pivot = v[der];

  // puntero elemto mayo
  int i = (izq - 1);

  // comparo con el pivot
  for (int j = izq; j < der; j++) {
    if (v[j] <= pivot) {

      // Si se encuentra un elemento menor que piv intercambio con el mayor apuntado por i
      i++;

      swap(&v[i], &v[j]);
    }
  }

  // swap pivot con el mayor en i
  swap(&v[i + 1], &v[der]);

  // devuelvo el punto de particion
  return (i + 1);
}
void quicksort(int v[], int izq, int der) {
       if (izq < der) {

    // encontrar el pivote que tenga los elementos menores a la izq y los mayores a la der
    int pi = partition(v, izq, der);

    // llamada recursiva hacia la izq
    quicksort(v, izq, pi - 1);

    // llamada recursiva hacia la der
    quicksort(v, pi + 1, der);
  }
}
```

Ecuación en recurrencias de Quicksort:

1. Caso Base: Cuando el tamaño del array es 1 o menos, el array ya está ordenado, tiempo de ejecución O(1).
2. Caso General: Cuando el tamaño del array mayor que 1:

   Es decir, los casos hablados anteriormente donde se elegía un pivote, y se realizan las llamadas recursivas a izquierda y a derecha de este.

Caso general sacamos la ecuación:

1. El tiempo necesario para realizar la partición del array es proporcional al tamaño del array, por lo que lo representamos como n.
2. Al hacer la partición obtenemos dos sub-arrays de tamaño aproximadamente n/2 cada uno.
3. Se realizan dos llamadas recursivas en cada uno de los sub-arrays.
4. El tiempo de ejecución total para ordenar el array de tamaño T(n) es la suma del tiempo de la partición más el tiempo de ejecución de las dos llamadas recursivas en los sub-arrays

Por lo tanto, podemos expresar el tiempo total de ejecución T(n) como la suma del tiempo de la partición y el tiempo de ejecución de las dos llamadas recursivas en los sub-arrays, lo que lleva a la ecuación de recurrencia:

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

Resuelvo:

$n=2^c$

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

Obtenemos la ecuación lineal no homogénea:

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

Resolvemos la parte homogénea:

T(2^c)-2T(2^{c-1})=0

x(2^c)-2x(2^{c-1})=0

x(2^{c-1})*(x-2)=0

Ph(x)=(x-2)

Resolvemos la parte no homogénea tal que obtengamos

2^c=$b1^m q1(m)$

$b1 = 2$ y $q1(m) = 1$, donde el grado del polinomio es $d1 = 0$

Luego nuestro polinomio característico es= $(x - 2)\ (x - 2)$ =$(x - 2)^2$

r=1 , con valor R1=2 y M1=2

Aplicamos la fórmula de la ecuación característica

T(2^c)=$C10 * 2^c + C11 * 2^c * c$

Deshacemos cambio:

T(n)=$C10 * n + C11 * n * log(n)$

El orden de la eficiencia en el caso medio es O(n*log(n))

Ahora el peor caso, que ya hemos comentado con anterioridad como sería, obtenemos que

T(n)=T(n-1)+O(n)

siendo O(n) el tiempo de ejecución para la partición en el peor caso y T(n-1) el tiempo de ejecución de Quicksort en el subarreglo más grande después de la partición

Luego si n= k+1

T(k+1) =T(k) + O(k+1)

Asumo que T(k)=O(k^2)

T(k+1)<= ak^2 + b(k+1) siendo a y b constantes positivas

T(k+1)<= ak^2 + bk+b

Luego ya podemos concluir que para una k lo suficientemente grande

T(n)= O(n^2)

1.4- Algoritmo Selection Sort

El algoritmo Selection sort es un algoritmo de ordenación sobre un vector de tamaño n que funciona de la siguiente manera:

1. Se busca el elemento mínimo del vector.
2. Se intercambia con el elemento que está en la primera posición.
3. Se busca el siguiente mínimo y se intercambia con el elemento en la segunda posición y así sucesivamente hasta la posición n-2.

La variable de la que depende el tamaño del caso es n , que es el tamaño del vector que se quiere ordenar.

comenzando a analizar el funcionamiento interno del algoritmo de la línea 9 a la 12(el condicional if) y lo que hay en su interior tienen eficiencia O(1) y están dentro del condicional for de la línea 7, que la primera que entra al bucle realiza n-1 iteraciones y cada vez que sale y vuelve a entrar hace una iteración menos hasta llegar a la última que solo ejecuta 1 vez. Por lo tanto tiene eficiencia O(n-i-1). En las líneas 14,15,16 son operaciones para intercambiar los valores de posiciones que son operaciones elementales y tiene eficiencia O(1). Por último en la línea 4 está el primer bucle for que avanza en 1 en cada iteración y siempre va a hacer n-1 iteraciones, por lo que tiene eficiencia O(n). Por lo tanto, la eficiencia del Algoritmo sería O((n-1)*(n-i.1)), Lo que sería una eficiencia O(n^2).

```
void ordenSeleccion(int *v, int n){
    int min;
    int aux;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (v[j] < v[min])
            {
                min = j;
            }
        }
        aux = v[i];
        v[i] = v[min];
        v[min] = aux;
```

1.5- Algoritmo Shell Sort

El algoritmo Shell Sort es un algoritmo de ordenación de listas o vectores el cual utiliza un algoritmo de Inserción internamente con la diferencia de que en lugar de ir comparando las posiciones del vector a ordenar las comparaciones que realiza son de elementos que están separados por un espacio de varias posiciones y además de comportarse mejor que inserción cuando la lista está ligeramente desordenada.

Realizar un análisis de eficiencia en el peor de los casos de este algoritmo es una tarea un tanto compleja ya que influye mucho la elección de los intervalos en los que se “parte” el vector debido a que si por ejemplo se realiza una división en intervalos de n/2 el algoritmo tendría una complejidad en el peor de los casos de O(n^2), pero puede darse el caso de realizar una división diferente y que se produzcan eficiencias de O(n^k). Este hecho a día de hoy permanece como una pregunta no resuelta y es que encontrando una división correcta para un sub caso concreto podrían producirse eficiencias de orden O(n log n), caso que se presta al estudio a día de hoy.

Un ejemplo de ordenación Shell utilizando divisiones de n/2 podría ser:

-División en intervalos: Se comienza dividiendo el vector inicial en intervalos de la mitad y se agrupan en ambas mitades.

[8, 3, 7, 2, 5, 4, 6, 1] -> [8, 5], [3, 4], [7, 6], [2, 1]

-Ordenación de intervalos: Cada subconjunto de los intervalos formados se ordenan independientemente.

[8, 5] -> [5, 8]
[3, 4] -> [3, 4]
[7, 6] -> [6, 7]
[2, 1] -> [1, 2]

-Reducción del intervalo: se vuelve a reducir el intervalo a la mitad y se repite el proceso anterior:

[5, 6, 8, 7], [3, 1, 4, 2] -> [5, 6, 8, 7] -> [5, 6, 7, 8]
[3, 1, 4, 2] -> [1, 2, 3, 4]

-Finalmente se vuelve a dividir y se aplica un Insertion Sort normal:

[5, 1, 6, 2, 7, 3, 8, 4] -> [1, 2, 3, 4, 5, 6, 7, 8]

En este caso el código de las mediciones que se ha utilizado el cual divide el vector en la mitad con una eficiencia como ya se ha explicado anteriormente de $O(n^2)$ es:

```
void shellSort(vector<int>& arr) {
    int n = arr.size(); //-->O(1)
    //Elecicon de intervalo que se ira reducionedo
    for (int gap = n/2; gap > 0; gap /= 2) {      //--> O(log n) se va reducciendo el intervalo
        //insercion con los intervalos
        for (int i = gap; i < n; i++) { // --> O(n) todo el vector
            int temp = arr[i]; //--->O(1)
            int j;
            //mover los elementos hasta encontrar la posicion correcta
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {//O(n) tantas veces como elementos
                arr[j] = arr[j - gap]; //-->O(1)
            }
            //isertar el elemento finalmente
            arr[j] = temp; //-->O(1)
        }
    }
}
```

-Primer bucle O(log n) debido a que en cada paso se va reduciendo el intervalo a resolver.
-Segundo bucle O(n) debido a que se va a comprobar tantas componentes como elementos tenga el vector.
-Tercer bucle O(n) debido a que el número de iteraciones es proporcional a la distancia entre i y gap y en el peor caso será el tamaño n

Finalmente O(log n) * O(n) * O(n) = O(n² * log n) = O(n²)

2. Cálculo Constante K

2.1- Algoritmo Counting Sort

En este algoritmo de Counting Sort las mediciones que se han hecho realizado una ejecución del algoritmo en varios tamaños de casos son las siguientes:

Tam. Caso	Tiempo (us)
50000	3567
60000	4326
70000	4514
80000	5793
90000	5201
100000	5451
110000	5711
120000	7223
130000	6821
140000	7297
150000	8850

Para poder realizar una comparación y comprobar si el tiempo de ejecución real en un PC (datos que muestra la tabla anterior) y el tiempo teórico que debería tener este algoritmo es necesario realizar un cálculo de ajuste por medio del cálculo de K denominada constante oculta.
Partiendo del orden de eficiencia teórico calculado en el apartado anterior que indica cuando el tamaño de caso tiende a infinito, el tiempo de ejecución del algoritmo se denomina T(n) y está acotado por el orden de eficiencia O(f(n)) que en este caso para un algoritmo de conteo como este se corresponde con O(n+k), pero se ha cogido O(n) para realizar los cálculos ya que la k va ligada a los componentes que haya dentro del vector.

Para dicho cálculo se cuenta con la siguiente fórmula:

$$T(n) <= K * f(n)$$

donde f(n) es en este caso O(n). Despejando la K de la fórmula (constante oculta) obtenemos que K = T(n) / f(n), por lo tanto por ejemplo para el para el primer caso representado en la tabla donde el tamaño del caso es 50000 y el tiempo que se le ha calculado a dicho caso es 3567 ns obtenemos la siguiente K:

$$K = T(n) / f(n) \rightarrow K = 3567 / 50000 \rightarrow K = 0{,}07134$$

Realizando esta cuenta para el resto de tamaños del caso obtendremos una K promedio que nos servirá más adelante para calcular el tiempo teórico estimado. Cabe destacar que este cálculo puede no ser el correcto al 100% por el hecho que se ha comentado anteriormente de que el orden de eficiencia correcto sería O(n+k) pero en el infinito esta variación será mínima de ahí el O(n) y además esta k en el orden de eficiencia sería muy difícil de hallar.

2.2- Algoritmo Insertion Sort

Los tiempos que se han obtenido en nuestro PC al ejecutar el algoritmo de Insertion Sort para los siguientes tamaños del caso son:

Tam. Caso	Tiempo (us)
1000	1605
2000	1490
3000	2827
4000	4553
5000	5615
6000	8378
7000	11357
8000	14161
9000	17939
10000	22660

De igual manera que el algoritmo anterior, calculamos la constante K con la siguiente fórmula, pero teniendo en cuenta que este algoritmo tiene una eficiencia en el peor de los casos de $O(n^2)$, es decir, $f(n) = O(n^2)$. Para el primer tamaño del caso (n = 1000) tenemos que:

$$K = T(n) / f(n) \rightarrow K = 1605 / (1000)^2 \rightarrow K = 0{,}001605$$

Si realizamos esta operación para el resto de tamaños(n), obtendremos el valores aproximados para para K, y aproximamos el valor final de K como la media de todos estos valores. Este valor nos servirá para calcular el tiempo teórico estimado.

2.3- Algoritmo Quick Sort

En el algoritmo de Quicksort las mediciones que se han obtenido al realizar la ejecución del algoritmo en varios tamaños de casos son las siguientes:

Tam. Caso	Tiempo(us)
1000	147
10000	1042
20000	2390
30000	3621
40000	9706
50000	12265
60000	7786
70000	8792
80000	11898
90000	16295
100000	19892

Para el cálculo de la constante oculta, K
Se ha tenido en cuenta la eficiencia Teórica de este algoritmo, la cual depende de n, siendo O(n*log(n)) la promedio y O(n^2) la peor, debido a que el peor caso se aleja mucho del caso real, he utilizado O(n*log(n)) también, siendo este el que mejor se ajusta a la realidad.
f(n)=O(n^2) -> T(n) / f(n) ->K=Tiempo(us)/Tam. Caso*Tam. Caso
f(n)=O(n*log(n)) ->T(n) / f(n) ->K= Tiempo(us)/Tam. Caso*log(Tam. Caso)

2.4- Algoritmo Selection Sort

En el Selection Sort los tiempos obtenidos al realizar las mediciones del algoritmo en un PC son:

Tam. Caso	Tiempo (us)
1000	1555
2000	2140
3000	3250
4000	5683
5000	8833
6000	12681
7000	17373
8000	23052
9000	28288
10000	34847

Para realizar la comprobación y comparar la eficiencia teórica y la práctica necesitamos calcular la constante oculta,K.
Teniendo en cuenta la eficiencia teórica de este algoritmo anteriormente calculado y que dependía de n, su tiempo de ejecución es T(n) y está acotado por la eficiencia O(f(n)) , donde f(n) es O(n^2) y se usaría la fórmula ya antes mencionada:

$$T(n)<=K*f(n)$$

Si sustituimos por los datos en la tabla obtendremos la k por cada tam de caso.
Por ejemplo en 3000 de tam de caso habría 3250 de T y la K sería:

$$K=3250/3000^2 =0,0003611111111$$

Calculando el resto de K podríamos obtener el valor de la K promedio que nos va a ayudar a comparar la eficiencia práctica y teórica.

2.5- Algoritmo Shell Sort

En el caso de Este algoritmo de Shell Sort los tiempos que se han obtenido al realizar las mediciones en un PC son los siguientes:

Tam. Caso	Tiempo (us)
50000	27387
60000	25072
70000	30312
80000	38345
90000	39862
100000	46985
110000	50863
120000	56043
130000	62765
140000	68987
150000	69760

Hay que destacar que se han cogido tamaños más grandes debido a que para tamaños muy pequeños el algoritmo ordenaba muy rápido y se ha considerado que para que fuera más ilustrativo de esta manera sería mejor. Esto viene a decirnos que para tamaños de caso pequeños el algoritmo es considerablemente rapido (por ejemplo para tamaños de caso 1000 y 5000, el tiempo medido ha sido 299 y 1987 ns respectivamente) pero conforme va aumentando el tamaño el tiempo va creciendo exponencialmente y de ahí que su eficiencia teórica en el peor caso calculada anteriormente sea $O(n^2)$.
Siguiendo la dinámica para el cálculo de la constante oculta K, en este caso sería el siguiente:

$f(n) \rightarrow O(n^2)$, entonces tenemos que $T(n) <= K^* O(n^2)$

despejando la K quedaria que $K = T(n)/O(n^2)$ y por ejemplo para el primera tamaño de caso 50000 quedaria que:

$K=27387/50000^2 = 0{,}0000109548$

Siguiendo la misma dinámica y realizando el cálculo de K para todos los tamaños obtendremos una K promedio que es necesaria para el cálculo del tiempo teórico estimado K*f(n).

3. Tiempo real vs Tiempo Teórico

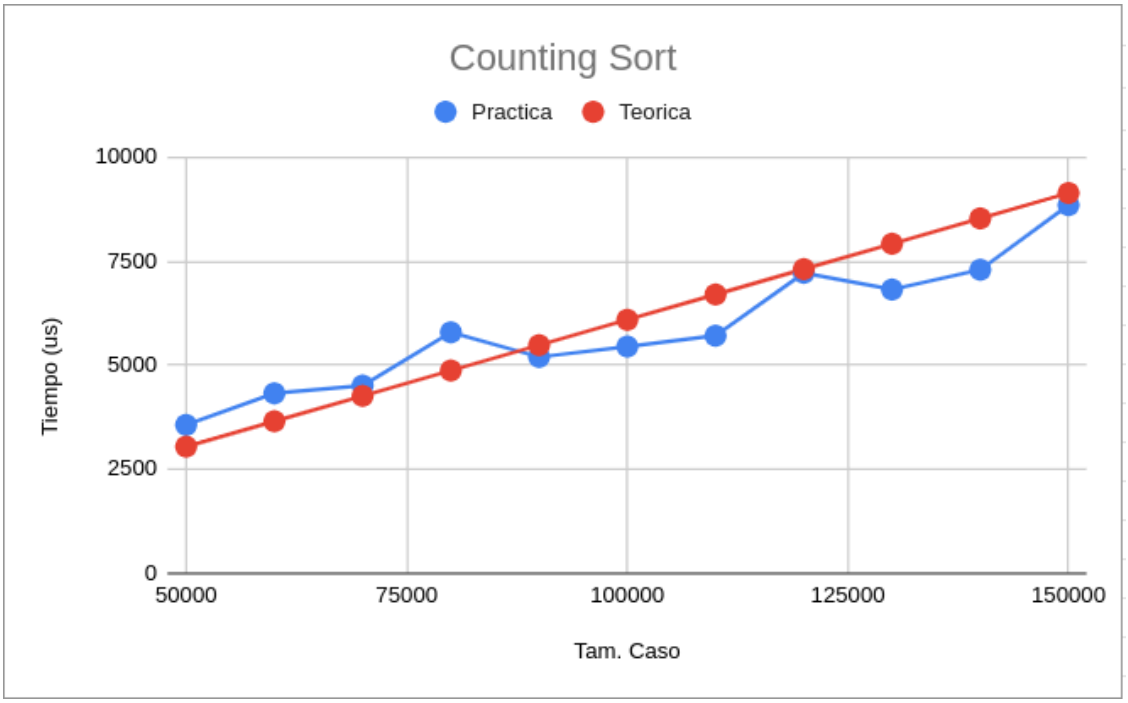
3.1- Algoritmo Counting Sort

En esta tabla definitiva se muestra los valores que se han calculado para hallar el tiempo teórico estimado, el cual se calcula por medio de la eficiencia híbrida hallada en apartados anteriores por medio de la constante oculta K:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
50000	3567	0,07134	3046,688055
60000	4326	0,0721	3656,025666
70000	4514	0,06448571429	4265,363277
80000	5793	0,0724125	4874,700888
90000	5201	0,05778888889	5484,038499
100000	5451	0,05451	6093,37611
110000	5711	0,05191818182	6702,713721
120000	7223	0,06019166667	7312,051332
130000	6821	0,05246923077	7921,388943
140000	7297	0,05212142857	8530,726554
150000	8850	0,059	9140,064165
	K promedio:	0,0609337611	

El tiempo teórico estimado quedaría como K * f(n), siendo K la K promedio que se ha obtenido realizando el cálculo para todos los tamaños de caso y f(n) como se ha indicado anteriormente O(n).
Como se ha indicado en el apartado anterior correspondiente a este algoritmo la K promedio que se obtiene puede variar muy ligeramente debido a que el orden de eficiencia real de este algoritmo es O(n+K) pero a efectos prácticos el haber elegido O(n) nos vale.

Siguiendo esta Tabla, se ha realizado una gráfica ilustrativa a modo de realizar una comparación lo más ajustada posible entre el tiempo real y el tiempo teórico de este algoritmo:



Se puede observar que aunque no se haya elegido el orden de eficiencia correcto que se ajuste al 100%, O(n+k), una complejidad O(n) se ajusta bastante bien la eficiencia práctica con la teórica salvo por algunos pico que pueden generarse por este factor o por algún ruido generado al medir los tiempos con el PC, factor que es inevitable. En líneas generales, ambas rectas siguen una tendencia lineal y que si se pudiera medir esa K que corresponde a valores arbitrarios dentro del propio código fuente en el orden de O(n+k) probablemente se ajustará aún más.
En líneas generales se puede considerar que este algoritmo es muy rápido en un amplio rango de tamaños de casos y consideramos que puede verse afectado su rendimiento para tamaños muy grandes.

3.2- Algoritmo Insertion Sort

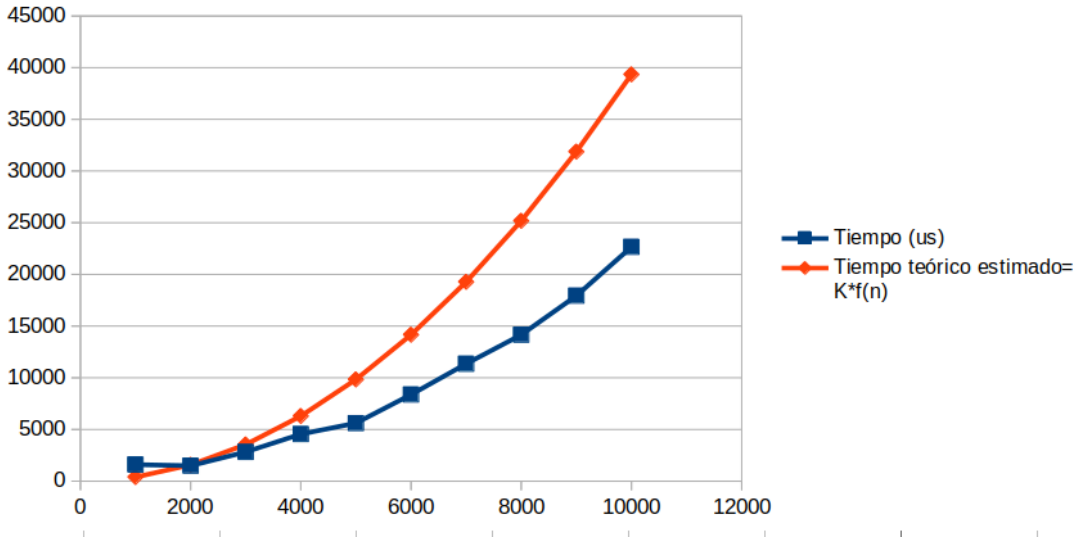
En esta tabla se puede observar los valores que hemos obtenido al calcular el tiempo teórico estimado para el algoritmo Insertion Sort y el valor de la constante oculta K.

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	1605	0,001605	393,460610433988
2000	1490	0,0003725	1573,84244173595
3000	2827	0,0003141111111111	3541,1454939059
4000	4553	0,0002845625	6295,36976694381
5000	5615	0,0002246	9836,51526084971
6000	8378	0,0002327222222222	14164,5819756236
7000	11357	0,0002317755102041	19279,5699112654
8000	14161	0,000221265625	25181,4790677753
9000	17939	0,0002214691358025	31870,3094451531
10000	22660	0,0002266	39346,0610433988
	K promedio:	0,000393460610434	

Lo primero que puede llamarnos más la atención es que la constante oculta K promedio es bastante más pequeña que, por ejemplo, el algoritmo Counting Sort que hemos visto antes. Esto es debido a que el Insertion Sort es de orden cuadrático ($O(n^2)$) y el Counting Sort es de orden lineal ($O(n)$). Por tanto, al realizar la fórmula que vimos para calcular K, que recordemos que era:

$$K = T(n) / f(n)$$

Donde T(n) era el tiempo de ejecución que obtenemos para cada tamaño (n) y $f(n) = n^2$, que es inversamente proporcional a K. Es por este motivo por el que la constante oculta en promedio es más pequeña.



La gráfica representa el tiempo de ejecución real y el tiempo teórico estimado del algoritmo de Insertion Sort para diferentes tamaños de entrada.

- El eje X representa el tamaño del caso de entrada (“Tam. Caso”), que varía de 1000 a 10000 en incrementos de 1000.
- El eje Y representa el tiempo en microsegundos (us).

La gráfica tiene dos líneas:

- La línea azul representa el “Tiempo (us)”, que es el tiempo real de ejecución del algoritmo para cada tamaño de entrada.
- La línea roja representa el “Tiempo teórico estimado= K*f(n)”, que es el tiempo de ejecución teórico estimado para cada tamaño de entrada.

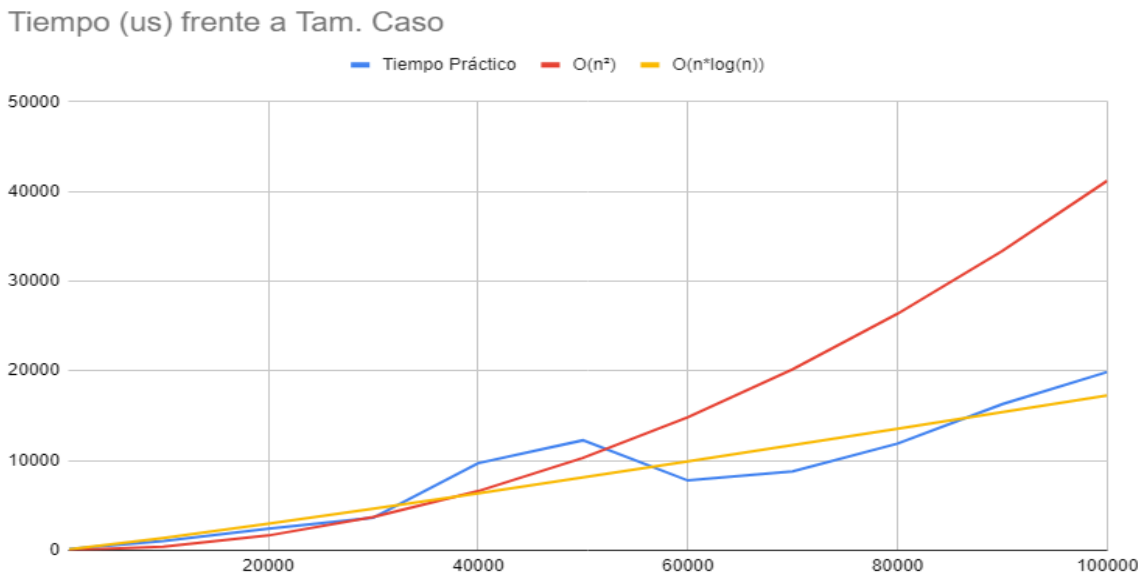
La diferencia entre las dos líneas indica que el tiempo teórico estimado es mayor que el tiempo real de ejecución. Esto puede deberse a varios factores, como la eficiencia del código, las condiciones de la máquina en la que ejecutamos el código, o las características específicas del conjunto de datos que estamos utilizando. También debemos de tener en cuenta que este algoritmo, como ya hemos mencionado, tiene una eficiencia en el peor de los casos de $O(n^2)$, pero en caso promedio tiene $O(n)$. Esto es, el tiempo teórico se calcula como $f(n) = n^2$, pero si cuando ejecutamos el algoritmo y el vector pasado como parámetro está en una gran parte ordenado, el algoritmo tendrá mejor eficiencia. Por eso el tiempo práctico es mejor que el tiempo teórico.

3.3- Algoritmo Quick Sort

Ahora pasamos a mostrar la tabla definitiva para el algoritmo Quicksort donde tenemos todos los datos necesarios para poder hacer la comparación de eficiencia teórica y eficiencia práctica. En las columnas 3-4 podemos ver la complejidad O(n²) que recordemos es el peor caso y en las columnas 5-6 la complejidad O(n*log(n)), caso promedio.

Tam. Caso	Tiempo(us)	k=Tiempo/f(n)	Tiempo teórico estimado=k*f(n)	k=Tiempo/h(n)	Tiempo Teórico estimado promedio = K*h(n)
1000	147	0,000147	4,120763772	0,049	103,6399704
10000	1042	0,00001042	412,0763772	0,02605	1381,866272
20000	2390	0,000005975	1648,305509	0,02778404246	2971,724144
30000	3621	0,00000402333	3708,687395	0,02695928771	4640,087144
40000	9706	0,00000606625	6593,222035	0,05272638785	6359,431485
50000	12265	0,000004906	10301,90943	0,05220292953	8116,685204
60000	7786	0,00000216277	14834,74958	0,02715834218	9904,149085
70000	8792	0,00000179428	20191,74248	0,02592310805	11716,73574
80000	11898	0,00000185906	26372,88814	0,03033291259	13550,82937
90000	16295	0,00000201172	33378,18655	0,03654555771	15403,72641
100000	19892	0,0000019892	41207,63772	0,039784	17273,3284
	K promedio:	0,00000412076		0,03454665681	

Pasamos a una comparación gráfica:



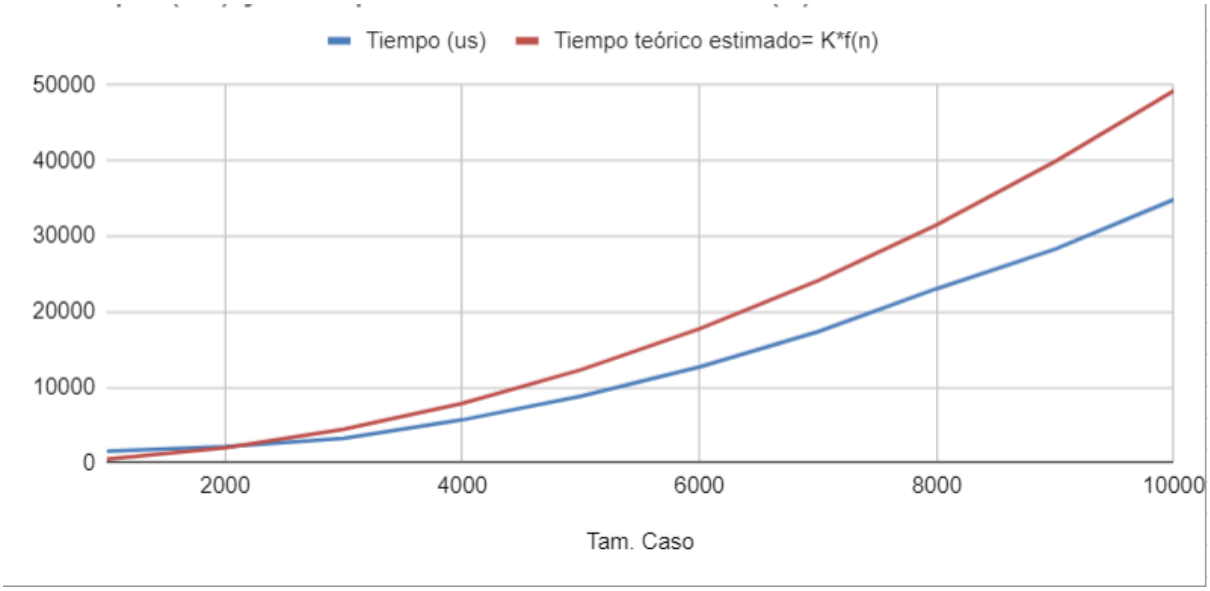
Analizando la gráfica podemos encontrar cierta similitud entre el camino que sigue el caso general del algoritmo y el tiempo práctico, mientras que el peor caso se aleja de las muestras obtenidas, a pesar de que en ciertas muestras hemos obtenido un peor resultado, lo normal en este algoritmo es que al elegir un pivote los subarrays generados estén más o menos equilibrados, no obstante como hemos podido comprobar existen ocasiones donde el algoritmo sea un poco más lento.

3.4- Algoritmo Selection Sort

Para poder comparar el tiempo teórico estimado hemos tomado la constante oculta(K) calculando la K promedio se puede obtener para cada tamaño de caso como se muestra en la siguiente tabla:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	1555	0,001555	492,4311699
2000	2140	0,000535	1969,72468
3000	3250	0,0003611111111	4431,880529
4000	5683	0,0003551875	7878,898719
5000	8833	0,00035332	12310,77925
6000	12681	0,00035225	17727,52212
7000	17373	0,0003545510204	24129,12733
8000	23052	0,0003601875	31515,59488
9000	28288	0,0003492345679	39886,92477
10000	34847	0,00034847	49243,11699
	K promedio:	0,0004924311699	

Este algoritmo presenta una complejidad $O(n^2)$, con la fórmula K*f(n) y comparándolo en una gráfica con el tiempo práctico se puede apreciar que el tiempo teórico estimado es bastante acertado y parecido al práctico y aunque el práctico está por debajo en tiempos eso puede resultar por unos mejores casos de modo aleatorio, pero crecen de manera muy similar.



3.5- Algoritmo Shell Sort

A continuación se muestra la tabla definitiva para el algoritmo de ordenación Shell donde se muestra todos los elementos necesarios para realizar una comparación entre eficiencia teorica y practica:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
50000	27387	0,0000109548	13761,39666
60000	25072	0,000006964444444	19816,4112
70000	30312	0,000006186122449	26972,33746
80000	38345	0,00000599140625	35229,17546
90000	39862	0,000004921234568	44586,92519
100000	46985	0,0000046985	55045,58665
110000	50863	0,000004203553719	66605,15985
120000	56043	0,000003891875	79265,64478
130000	62765	0,000003713905325	93027,04144
140000	68987	0,000003519744898	107889,3498
150000	69760	0,000003100444444	123852,57
	K promedio:	0,000005504558665	

Como se ha visto anteriormente este algoritmo presenta una complejidad de $O(n^2)$. El tiempo teórico estimado quedaría como K * f(n), siendo K la K promedio que se ha obtenido realizando el cálculo para todos los tamaños de caso y f(n) como se ha indicado anteriormente $O(n^2)$.

Cabe destacar que el tiempo practico para este algoritmo, como ya se ha visto en apartados anteriores, va ligado a la elección del intervalo que se elija en cada uno de los pasos que realiza el algoritmo internamente y como veremos a continuación en la gráfica comparativa entre el tiempo teórico y el tiempo práctico se puede apreciar esa variación en función de la ejecución realizada.

Podemos observar como en líneas generales sí que crece aproximadamente de manera parecida a como lo haría teóricamente aunque en tamaños grandes se distancie, fenómeno que puede ser debido a lo comentado anteriormente con respecto al funcionamiento interno del algoritmo incluso puede influir también el PC en el que se han hecho las mediciones generando algo de ruido, de ahí los picos que se producen.

4. Comparación de órdenes de eficiencia

Tras analizar cada uno de los algoritmos por individual y realizar un estudio de su eficiencia teórica y práctica a continuación se muestra una comparación entre todos ellos, por un lado comparando su eficiencia real (práctica) medida en pc y por otro la eficiencia ideal que deberían tener (teórica).

Para todos ellos se han seleccionado los mismos tamaños de caso para que la comparación no salga dispareja y poder así comprobar cuál de ellos es más o menos rápido en función del incremento del número de componentes a ordenar.

En el caso del cálculo del tiempo medido con el pc, el algoritmo más lento en realizar una ordenación es el Insertion Sort cuando el número de componentes del vector es relativamente pequeño, seguido de Selection Sort, Shell Sort, Quick Sort y el más rápido en este caso sería Counting Sort. Esto es debido a que el algoritmo de inserción en el peor caso tiene un orden cuadrático por lo que crecerá más rápido y el Counting sort tiene un orden lineal por lo que crecerá más progresivo. Cabe destacar que son tiempos medidos con pc por lo que no necesariamente tiene el porqué cumplirse este orden ya que se está hablando de un hipotético peor caso.

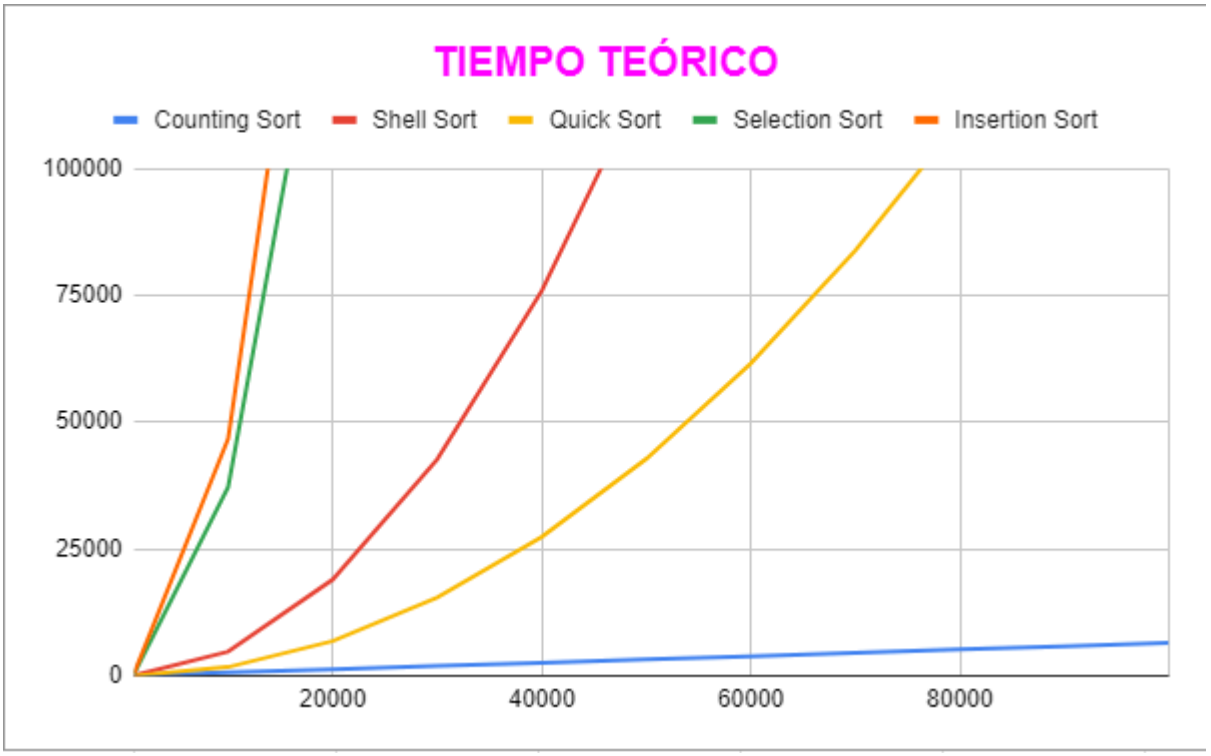
Para tamaños de caso grandes sigue más o menos la misma dinámica salvo que el algoritmo de selección es más lento que el de inserción pero a pesar de que teóricamente deberían ser iguales puede que al realizar la ejecución en el pc hubiese un vector más complejo de ordenar que otro.

Tiempo practico									
Counting Sort		Shell Sort		Quick Sort		Insertion Sort		Selection Sort	
Tam. Caso	Tiempo (us)	Tam. Caso	Tiempo (us)	Tam. Caso	Tiempo (us)	Tam. Caso	Tiempo (us)	Tam. Caso	Tiempo (us)
1000	81	1000	340	1000	147	1000	1854	1000	1684
10000	689	10000	4747	10000	1042	10000	22681	10000	35168
20000	1405	20000	10594	20000	2390	20000	88355	20000	139303
30000	1944	30000	14290	30000	3621	30000	208798	30000	310633
40000	2589	40000	17638	40000	9706	40000	353744	40000	552267
50000	3177	50000	21149	50000	12265	50000	565695	50000	874145
60000	3732	60000	25098	60000	7786	60000	806120	60000	1242256
70000	4310	70000	35014	70000	8792	70000	1087560	70000	1684619
80000	4437	80000	36887	80000	11898	80000	1412827	80000	2199470
90000	4790	90000	41482	90000	16295	90000	1802740	90000	2782125
100000	5267	100000	48305	100000	19892	100000	2217345	100000	3434434

En cuanto al orden de eficiencia teórico podemos ver que para tamaños de caso relativamente pequeños en el peor de los casos aplicando su complejidad a cada algoritmo, la clasificación de algoritmos de más lento a más rápido es: Selección > Inserción > Counting > Shell > Quick.

En el caso de tamaños de caso más grandes seguiría la misma dinámica de los algoritmos de selección e inserción como los más lentos por su $O(n^2)$ e irán seguidos de Shell Sort > Quick Sort > Counting Sort.

Tiempo teórico									
Counting Sort		Shell Sort		Quick Sort		Insertion Sort		Selection Sort	
Tam. Caso	Tiempo teorico	Tam. Caso	Tiempo teorico	Tam. Caso	Tiempo teorico	Tam. Caso	Tiempo teorico	Tam. Caso	Tiempo teorico
1000	64,56711508	1000	47,4318337	1000	17,10978525	1000	371,998703	1000	467,5785336
10000	645,6711508	10000	4743,18337	10000	1710,978525	10000	37199,8703	10000	46757,85336
20000	1291,342302	20000	18972,73348	20000	6843,914099	20000	148799,4812	20000	187031,4134
30000	1937,013452	30000	42688,65033	30000	15398,80672	30000	334798,8327	30000	420820,6802
40000	2582,684603	40000	75890,93393	40000	27375,6564	40000	595197,9248	40000	748125,6537
50000	3228,355754	50000	118579,5843	50000	42774,46312	50000	929996,7575	50000	1168946,334
60000	3874,026905	60000	170754,6013	60000	61595,22689	60000	1339195,331	60000	1683282,721
70000	4519,698056	70000	232415,9852	70000	83837,94771	70000	1822793,645	70000	2291134,815
80000	5165,369206	80000	303563,7357	80000	109502,6256	80000	2380791,699	80000	2992502,615
90000	5811,040357	90000	384197,853	90000	138589,2605	90000	3013189,494	90000	3787386,122
100000	6456,711508	100000	474318,337	100000	171097,8525	100000	3719987,03	100000	4675785,336



5. Conclusión.

En conclusión, los datos que hemos obtenido del tiempo práctico y teórico, hemos podido ver que no se ajusta del todo ya que aunque en el teórico diga que en el peor de los casos nos vamos a encontrar cierta situación, el práctico te va a demostrar que en un caso en concreto no está tan mal como se interpreta en la teoría. Por ello recomendamos lo siguiente:
Para conjuntos de datos grandes y diversos con valores desconocidos, Quicksort suele ser una mejor opción debido a su complejidad O(n*log(n)). Sin embargo, si se trabaja con datos donde el rango de valores es grande y conocido, siempre y cuando no te importe utilizar memoria adicional, el counting sort puede ser más eficiente, recordemos su complejidad O(n+k).
Por otro lado, para conjuntos de datos pequeños, Counting sort va a ser más eficiente en general. No nos olvidemos de Insertion y Selection sort que a pesar de ser menos eficientes son opciones más simples, para aquellos casos donde se premie más la simplicidad que la eficiencia. Por último, Shell sort puede ser una buena opción para tamaños intermedios, ofreciendo una mejora sobre el algoritmo de inserción básico.