UNIVERSIDAD DE GRANADA

Practica 1
Cálculo de la eficiencia de algoritmos

Joaquín Sergio García Ibáñez
Juan Navarro Maldonado

1.-Introducción

Esta práctica consiste en el análisis de eficiencia de distintos algoritmos. Para ello dividiremos el estudio de los tres algoritmos en varias partes:

- **Análisis de la eficiencia teórica:** Consiste en analizar el tiempo de ejecución en el peor caso de dichos algoritmos.
- **Análisis de la eficiencia práctica:** Probaremos la ejecución de los algoritmos en nuestros ordenadores midiendo el tiempo de ejecución para distintos tamaños de caso.
- **Análisis de la eficiencia híbrida:** A partir del análisis de la eficiencia real, procederemos a hallar la constante K.

2.- Diseño de los algoritmos

2.1.- Algoritmos Iterativos

2.1.1.- Descripción del problema

En este apartado nos pide que diseñemos un algoritmo iterativo para resolver un problema planteado, este consiste en eliminar las componentes repetidas de un vector dado y devolver el mismo vector sin las componentes repetidas, nos piden que resolvamos este problema con dos casos distintos:

- Cuando las componentes del vector original están desordenadas
- Cuando las componentes del vector original están ordenadas.

Para cada caso usaremos un algoritmo distinto.

2.1.2.- Descripción de la solución planteada

Para la resolución de este problema hemos decidido basarnos en la plantilla proporcionada en la práctica, la cual hemos modificado para que se ajuste a los requerimientos del problema, ya que nos permite poder generar un vector aleatorio con el número de componentes que deseamos y así facilitarnos la tarea a la hora de calcular las eficiencias prácticas e híbridas para los distintos casos.

También, para tener resultados más precisos a la hora de comparar ambos algoritmos, hemos usado un algoritmo de ordenación para ordenar el vector original antes de ejecutar el segundo algoritmo propuesto y así tener los mismos componentes en ambas ejecuciones del código.

2.1.3.- Código algoritmo para un vector desordenado

```
1. void eliminaComponentesDesordenado2(int *v, int &tam){
2.     int tam_copia = 0;
3.     int v_copia[tam];
4.     int k= 0;
5.     int i = 1;
6.     bool terminado = false;
7.     int aux = 0;
8.     bool repetido = false;
9.     bool copiaRealizada = false;
10.    bool encontrado = false;
11.
12.    v_copia[0] = v[0];
13.    tam_copia++;
14.    //cout << "Bucles anidados " << endl;
15.    while (!terminado){
16.        while(!encontrado){
17.
18.            //cout << "\t\t Comparando v[]" << v[i] << " con v_copia[]
   " << v_copia[k] << endl;
19.            if(v[i] == v_copia[k]){
20.                //cout << "\tSon iguales No copia v = " << v[i] << " k
   " << k << endl;
21.                v[i] = v[i+1];
22.                encontrado = true;
23.            }else{
24.                //cout << " Iteracion k valor:" << k << endl;
25.                k++;
26.            }
27.
28.            if( k >= tam_copia && !encontrado){
29.                //cout << "Finalizada la busqueda en el vector copia
   ... copiando " << v[i] << endl;
30.                v_copia[k] = v[i];
31.                //cout << "Copiado numero " << v[i] << endl;
32.                tam_copia++;
33.                encontrado = true;
34.
35.            }
36.        }
37.        if(i == tam-1){
38.            //cout << "Terminado i:" << i  << endl;
39.            terminado = true;
40.        }
41.        else{
42.            encontrado = false;
43.            k=0;
44.            //cout << "Incremente i "<< endl;
45.            i++;
46.        }
47.    }
48.
49.    //Copiamos v_copia
50.    for(int i = 0; i < tam_copia; i++)
51.        v[i] = v_copia[i];
```

```
52.
53.     tam = tam_copia;
54.
55. }
56.
```

Como podemos observar este algoritmo se basa en copiar los valores que aparecen por primera vez en el vector original en un vector auxiliar para finalmente, cuando se recorra el vector original, se copie el vector auxiliar en el vector original.

Este algoritmo nos servirá tanto para un vector desordenado como para uno ordenado, pero para un vector ordenado podemos simplificar la tarea con el siguiente código.

2.1.4.-Código algoritmo para un vector ordenado

```
1. void eliminaComponentesOrdenados(int *v, int &tam){
2.     bool ordenado = false;
3.     int i = 0;
4.     int aux_tam = tam;
5.     int aux_int = -1;
6.     int aux_i = i;
7.
8.     while (!ordenado){
9.         if (v[i] == v[i+1]){
10.            for(int k = i; k<tam; k++)
11.                v[k] = v[k+1];
12.            aux_tam--;
13.        }else
14.            i++;
15.
16.        if(i == aux_tam-1)
17.            ordenado = true;
18.    }
19.    tam = aux_tam;
20. }
```

Como vemos este código es más sencillo, simplemente se basa en hacer un borrado lógico de las componentes de un vector, simplemente si el siguiente elemento al que estamos apuntando es igual, copiamos dicho elemento en la posición actual y decrementamos el tamaño del vector.

Este algoritmo no serviría para un vector desordenado, ya que solamente elimina las componentes repetidas que están en posiciones consecutivas.

2.2.- Algoritmos Recursivos

2.2.1.- Algoritmo de Hanoi

```
1. void hanoi(int M, int i, int j){
2.     if(M>0){
3.         hanoi(M-1, i, 6-i-j);
4.         cout << i << " -> " << j << endl;
5.         hanoi(M-1, 6-i-j,j);
6.     }
7. }
```

Se trata de resolver el problema de las torres de Hanoi, en el que se tienen 3 barras y hay que mover M anillos de la primera barra a la segunda. En cada movimiento solo se puede mover un anillo y ningún anillo de mayor tamaño puede estar encima de otro de menos tamaño, se trata de un algoritmo recursivo en el que se hace dos llamadas a el mismo de tamaño M-1.

2.2.2.- Algoritmo HeapSort

```
1. void HeapSort(int *v, int n){
2.     double *apo=new double[n];
3.     int tamapo=0;
4.
5.     for(int i=0; i<n; i++){
6.         apo[tamapo]= v[i];
7.         tamapo++;
8.         insertarEnPos(apo, tamapo); 0(LOG(N))
9.     }
10.
11.    for (int i = 0; i<n; i++){
12.        v[i] = apo[0];
13.        tamapo--;
14.        apo[0]=apo[tamapo];
15.        reestructurarRaiz(apo, 0, tamapo);   0(LOG(N))
16.
17.    }
18.
19.    delete [] apo;
20.}
```

```
21.void reestructurarRaiz(double *apo, int pos, int tamapo){
22.    int minhijo;
23.     if (2*pos+1 < tamapo){
24.         minhijo=2*pos+1;
25.         if((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1]))
    minhijo++;
26.
27.         if (apo[pos]>apo[minhijo]){
28.             double tmp = apo[pos];
29.             apo[pos]=apo[minhijo];
30.             apo[minhijo] = tmp;
31.             reestructurarRaiz(apo, minhijo, tamapo);
32.         }
33.     }
34.}
```

```
    void insertaEnPos(double *apo, int pos){
1.     int idx = pos-1;
2.     int padre;
3.
4.     if (idx > 0){
5.         padre=(idx-2)/2;
6.     }else{
7.         padre=(idx-1)/2;
8.     }
9.
10.if (apo[padre] > apo[idx]) {
11.      double tmp=apo[idx];
12.      apo[idx]=apo[padre];
13.      apo[padre]=tmp;
14.      insertaEnPos(apo, padre+1);
15.}

    }
```

HeapSort es un algoritmo de ordenamiento no recursivo. Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo(heap), y luego extraer el nodo que queda como nodo raíz del montículo(cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Este algoritmo está compuesto por 2 llamadas a unas funciones externas a ella.

3.- Cálculo de la eficiencia teórica

3.1.- Algoritmos Iterativos

3.1.1.- Algoritmo EliminaComponentesDesordenados:

El siguiente algoritmo elimina los componentes repetidos en un vector desordenado.

Caso peor:

El peor caso de este algoritmo se da cuando no hay ningún elemento repetido en el vector original; ya que tiene que hacer las máximas iteraciones posibles el cual se da iterando el vector hasta su máximo número de componentes.

La eficiencia en el peor caso se calcula de forma que el algoritmo está compuesto de 9 sentencias atómicas de eficiencia $O(1)$ y después entraría en un bucle while que al darse que no había ningún elemento repetido en el vector original tendría que hacer n iteraciones en el bucle while para comprobar si ha terminado, (siendo n el tamaño del vector original) y n iteraciones más en el bucle while para comprobar si ese elemento se ha comprobado ya y en caso contrario meterlo en el vector, por lo tanto para calcular la eficiencia del bucle while seria:

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- El bloque de sentencias tiene eficiencia O(1).

Una vez hecho esto usamos la fórmula y nos da:

$$O(O(1) + n * (O(1) + O(1))) = O(n)$$

las demás sentencias del bucle while serían sentencias de eficiencia $O(1)$, por lo tanto para calcular la eficiencia del bucle while seria:

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- El bloque de sentencias tiene eficiencia O(n).

Una vez hecho esto usamos la fórmula y nos da:

$$O(O(1) + n * (O(1) + O(n))) = O(n^2)$$

Como las demás sentencias son $O(1)$ y el bucle for para copiar los vectores es:

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- La actualización es de eficiencia O(1).
- La inicialización es de eficiencia O(1).

Una vez hecho esto usamos la fórmula y nos da:

$O(O(1) + O(1) + n\,(O(1) + O(1) + (O(1))) = O(n)$, aplicamos la regla de la suma y obtenemos que el algoritmo tiene eficiencia $O(n^2)$

Caso mejor:

El mejor caso para este algoritmo se da cuando el vector original tiene el mismo elemento repetido, por lo tanto tendria las mismas sentencias $\Omega(1)$, pero entraria al bucle y se ejecutaria n veces pero el bucle for al solo haber 1 mismo elemento repetido n veces solo se ejecutaria 1 sola vez y se convertiria en $\Omega(1)$ y por lo tanto:

- La evaluación de la condición es de eficiencia $\Omega(1)$.
- El bucle se ejecuta n veces.
- El bloque de sentencias tiene eficiencia $\Omega(1)$.

Finalmente sabemos que el algoritmo en el mejor caso es $\Omega(n)$

3.1.2.- Algoritmo EliminaComponentesOrdenados:

Caso peor:

El peor caso para este algoritmo se da cuando el vector original tiene el mismo elemento repetido; ya que tiene que hacer las máximas iteraciones posibles el cual se da iterando el vector hasta su máximo número de componentes.

La eficiencia en el peor caso se calcula de forma que el algoritmo está compuesto de 9 sentencias atómicas de eficiencia $O(1)$ y después entraría en un bucle while que al darse que no había ningún elemento repetido en el vector original tendría que hacer n iteraciones en el bucle while para comprobar si ha terminado, (siendo n el tamaño del vector original) y n iteraciones más en el bucle for para comprobar si ese elemento se ha comprobado ya y en caso contrario meterlo en el vector, por lo tanto para calcular la eficiencia del bucle for seria:

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- La actualización es de eficiencia O(1).
- La inicialización es de eficiencia O(1).

Una vez hecho esto usamos la fórmula y nos da:

$$O(O(1) + O(1) + n\,(O(1) + O(1) + (O(1))) = O(n)$$

las demás sentencias del bucle while serían sentencias de eficiencia $O(1)$, por lo tanto para calcular la eficiencia del bucle while seria:

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- El bloque de sentencias tiene eficiencia O(n).

Una vez hecho esto usamos la fórmula y nos da:

$$O(O(1) + n * (O(1) + O(n))) = O(n^2)$$

Como las demás sentencias son $O(1)$, aplicamos la regla de la suma y obtenemos que el algoritmo tiene eficiencia $O(n^2)$

Caso mejor:

El mejor caso de este algoritmo se da cuando no hay ningún elemento repetido en el vector original, por lo tanto tendría las mismas sentencias $\Omega(1)$, pero entraría al bucle y se ejecutaria n veces pero el bucle for al solo haber 1 mismo elemento repetido n veces solo se ejecutaria 1 sola vez y se convertiria en $\Omega(1)$ y por lo tanto:

- La evaluación de la condición es de eficiencia $\Omega(1)$.
- El bucle se ejecuta n veces.
- El bloque de sentencias tiene eficiencia $\Omega(1)$.

Finalmente sabemos que el algoritmo en el mejor caso es $\Omega(n)$

3.2.- Algoritmos Recursivos

3.2.1.- Algoritmo ReestructuraRaiz:

El siguiente algoritmo también inserta un vector dado una posición y en este caso también se pasa por parámetro el tamaño del vector.

Variables de las que depende el tamaño del caso peor:

Para obtener el tamaño de caso debemos de analizar el código de nuestra segunda función llamada ReestructurarRaiz.

Podemos observar que esta función empieza con una sentencia de declaración de eficiencia O(1) y posteriormente nos encontramos con un condicional if que lo llamaremos if padre el cual para obtener su eficiencia primero debemos de analizar lo que contiene e ir de dentro hacia afuera. Si nos vamos a lo más profundo tenemos un if llamémosle if 3 el cual tiene 3 sentencias de asignación de eficiencia O(1) y posteriormente cuenta con una llamada recursiva la cual el tamaño viene dado por la variable minhijo la cual compone el tamaño del caso para nuestro problema la cual va a aumentarse cada vez que se ejecuta el algoritmo 2*pos +1 eso hará que se reduzca a la mitad el número de llamadas recursivas por que ya que en la condición tenemos que minhijo < tamapo reduciremos las llamadas recursivas a tan solamente la mitad. y llamariamos + 1 veces a minhijo, por lo tanto nos encontraríamos con una Recurrencia Lineal No Homogénea de la manera:

$$T(n) = T(n/2) + 1$$

+1 por que en la función recursiva podríamos ver que la variable minhijo sería usada +1 veces.

Para resolver esta recurrencia lineal vamos a usar el método de cambio de variable donde $n = 2^m$; quedaría de la siguiente manera $T(2^m) = T(2^m/2) + 1$; que simplificando se quedaría en $T(2^m) = T(2^{m-1}) + 1$ donde separamos la Parte Homogénea de la No Homogénea.

Parte Homogénea:

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

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

Obtenemos: $PH(x) = x - 1$

Parte No Homogénea:

$1 = b_1^{\ m} * q_1(n)$; Donde vemos que

$b_1 = 1$ y $q_1(n) = 1$ con grado $d_1 = 0$, sacando el polinomio correspondiente:

$p(x) = (x-1) * (x-b_1)\hat{}d_1 + 1 = (x-1) * (x-1) = (x-1)^2$

Y obtenemos que la raíz 1 es 1 y la multiplicidad es 2, entonces obtenemos

$T(2^m) = c_{10}1^{\ m} + c_{11}1^{\ m}m$

Deshacemos cambio de variable: $m = log_2 n$

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

Con lo cual la eficiencia sería O($log_2 n$)

Finalmente tenemos que calcular la eficiencia de todo el algoritmo el cual va a ser O($log_2 n$) ya que hemos calculado las anteriores eficiencias y son O(1).

3.2.2.- Algoritmo InsertaEnPos:

El siguiente algoritmo inserta en una posición dada un vector dado por parámetro. Podemos ver que a la función InsertarEnPos se le pasa como parámetros un vector de tipo double y una posición pasada como entero.

Variables de las que depende el tamaño del caso peor:

Para obtener el tamaño de caso debemos de analizar el código de nuestra primera función llamada insertarEnPos, a esta función se le pasan dos parámetros un vector de tipo doble y una posición de tipo entero.

Podemos observar que las dos primeras sentencias se componen de una asignación de O(1) y una declaración de O(1) también.

Después tenemos un condicional if que se compone por otros dos condicionales if-else por lo tanto la eficiencia del if padre sera el Maximo entre la eficiencia del if-else hijo y la evaluación de su condición, como podemos ver dentro del if hijo nos encontramos con una sentencia de asignación que es O(1) y su evaluacion de condicion es de O(1) y en el condicional else tenemos otra sentencia de asignación que es O(1), por lo tanto el orden de eficiencia del if hijo es el máximo de los órdenes de eficiencia de la sentencia del condicional else, la eficiencia del condicional if y la evaluación de la condición del condicional if es decir, MAX(O(1), O(1), O(1)) es decir la eficiencia en todo ese bucle if padre sera el maximo de la evaluación de la condición del condicional if padre y el bloque de sentencias de dentro que hemos visto antes era O(1), por lo tanto todo ese bloque if padre será MAX(O(1), O(1)).

Después nos encontramos con otro condicional if donde podemos ver que la evaluación de la condición de su condicional es O(1) y su bloque de sentencias vemos que consta de 3 sentencias de asignación la cual las 3 son de eficiencia O(1) y el máximo de esas 3 sentencias vemos que es O(1), luego nos encontramos con que hace una llamada recursiva , donde podemos ver que el tamaño del problema viene dado por la variable padre por lo tanto nuestra n = padre, ya sabremos el tamaño del problema que nos preguntamos al principio del análisis del algoritmo y como podemos ver en el condicional 1 la variable padre decrementará su tamaño a la mitad, por lo tanto nos encontraríamos con una Recurrencia Lineal No Homogénea de la manera:

$$T(n) = T(n/2) + 1$$

+1 por que en la función recursiva podríamos ver que la variable padre sería usada +1 veces.

Para resolver esta recurrencia lineal vamos a usar el método de cambio de variable donde $n = 2^m$; quedaría de la siguiente manera $T(2^m) = T(2^m/2) + 1$; que simplificando se quedaría en $T(2^m) = T(2^{m-1}) + 1$ donde separamos la Parte Homogénea de la No Homogénea.

Parte Homogénea:

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

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

Obtenemos: $PH(x) = x - 1$

Parte No Homogénea:

$1 = b_1^m * q_1(n)$; Donde vemos que

$b_1 = 1$ y $q_1(n) = 1$ con grado $d_1 = 0$, sacando el polinomio correspondiente:

$p(x) = (x-1) * (x-b_1)^{\wedge}d_1 + 1 = (x-1) * (x-1) = (x-1)^2$

Y obtenemos que la raíz 1 es 1 y la multiplicidad es 2, entonces obtenemos

$T(2^m) = c_{10}1^m + c_{11}1^m m$

Deshacemos cambio de variable: $m = log_2 n$

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

Con lo cual la eficiencia sería O($log_2 n$)

Finalmente tenemos que calcular la eficiencia de todo el algoritmo el cual va a ser O($log_2 n$) ya que hemos calculado las anteriores eficiencias y son O(1)

3.2.3.- Algoritmo HeapSort:

Podemos observar que en la función de HeapSort, se llaman a las funciones insertarEnPos y reestructurarRaiz; las cuales hemos calculado su eficiencia teórica con anterioridad.

Variables de las que depende el tamaño del caso peor:

Para obtener el tamaño del caso peor debemos de analizar el código del algoritmo HeapSort.

Lo primero que podemos observar es que las dos primeras líneas son una declaración y una asignación por lo que se tratan de O(1).

Si seguimos analizando el código podemos encontrarnos con el primero de dos bucles for. Ambos bucles van a tener n (n = tamaño del vector) iteraciones, por lo que podemos determinar n como nuestro tamaño de caso peor. Para determinar el orden de eficiencia del primer bucle habrá que seguir la fórmula para las sentencias repetitivas for la cual sabemos

El bloque de sentencias es de eficiencia $O(log_2(n))$ ya que tenemos 3 sentencias de asignación y la llamada recursiva que habíamos calculado su orden de eficiencia anteriormente y su orden era $O(log_2(n))$ con lo cual todo el bloque de sentencias es de eficiencia $O(log_2(n))$**.**

- La evaluación de la condición es de eficiencia O(1).
- El bucle se ejecuta n veces.
- La actualización es de eficiencia O(1).
- La inicialización es de eficiencia O(1).

Una vez hecho esto usamos la fórmula y nos da:

O(O(1)+O(1)+N*(O(1)+$O(log_2(n))$+(O(1)))=$O(n * log_2(n))$

Para el segundo bloque for tiene la misma eficiencia que el primer bloque for solo varia que en el bloque de sentencias del 2 for hay una sentencia más de asignacion de eficiencia O(1) la cual no repercute en nada y el orden de eficiencia del 2 for seria también $O(n * log_2(n))$ y finalmente nos encontramos con una sentencia de asignación de eficiencia O(1) con lo cual podemos decir que nuestro algoritmo finalmente tiene eficiencia $O(n * log_2(n))$**.**

3.2.4.- Algoritmo Hanoi:

Se hacen 2 llamadas recursivas de tamaño n-1.

Variables de las que depende el tamaño del caso peor:

La variable de la que depende el caso peor es que el número de anillos sea mucho y por lo tanto se hagan llamadas de tamaño M-1 llamaremos n a ese tamaño. Por lo tanto nos encontramos con una Recurrencia Lineal No Homogénea de la forma:

$T(n) = 2T(n-1) + 1$ para resolver esta recurrencia pasamos a la izquierda la parte homogénea: $T(n) - 2T(n-1) = 1$. Ahora resolvemos la parte homogénea y después la parte no homogénea

-Parte Homogénea:

$T(n) - 2T(n-1) = 0$; $x - 2 = 0$; $PH(x) = x - 2$

-Parte No Homogénea:

$1 = b_1^{\,n} * q_1(n)$; Donde vemos que

$b_1 = 1$ y $q_1(n) = 1$ con grado $d_1 = 0$, sacando el polinomio característico correspondiente la ecuación :

$p(x) = (x-2) * (x-b_1)^{d_1} + 1 = (x-1) * (x-1) = (x-2) * (x-1)$

Y obtenemos 2 raices:

$R_1 = 2;\quad M_1 = 1$

$R_2 = 1;\quad M_2 = 1$

Obtenemos finalmente:

$T(n) = c_{10}2^n * n^0 + c_{20}1^n * n^0$

Con lo cual la eficiencia sería $O(2^n)$ y por lo tanto la eficiencia del algoritmo de hanoi es $O(2^n)$

3.3.- Comparación eficiencia teórica entre algoritmo HeapSort y MergeSort

Hemos visto en clase de teoría que la eficiencia del algoritmo de ordenación MergeSort es $O(n * log_2(n))$. Como hemos calculado antes el algoritmo de ordenación HeapSort es $O(n * log_2(n))$, podemos ver que a nivel teórico ambos tienen la misma eficiencia luego comprobaremos a nivel práctico su eficiencia.

4.-Cálculo de la eficiencia práctica

Para el cálculo de la eficiencia práctica de nuestros algoritmos debemos de ejecutar nuestro código con los distintos tamaños, en nuestro caso, hemos decidido que estos serán 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 y 100000.

A continuación, presentamos los resultados obtenidos tras realizar las pruebas.

4.1.- Algoritmos Iterativos

4.1.1.- Algoritmo EliminaComponentesDesordenados

Tamaño de caso	Tiempo real (us)
10000	30509
20000	115289
30000	246262
40000	437180
50000	686285
60000	978417
70000	1325097
80000	1746368
90000	2227656
100000	2728950

4.1.2.- Algoritmo EliminaComponentesOrdenados

Tamaño de caso	Tiempo real (us)
10000	19369
20000	73924
30000	160641
40000	281779
50000	439768
60000	628223
70000	863030
80000	1133663
90000	1424065
100000	1755935

4.2.- Algoritmos Recursivos

4.2.1.- HeapSort

Tamaño de caso	Tiempo real (us)
10000	1357
20000	3164
30000	4818
40000	6166
50000	7845
60000	9754
70000	11312
80000	14735
90000	16247
100000	17868

4.2.2.- MergeSort

Tamaño de caso	Tiempo real (us)
10000	911
20000	2179
30000	2443
40000	3648
50000	3985
60000	5081
70000	5702
80000	6976
90000	6882
100000	8794

5.-Cálculo de la eficiencia híbrida

Una vez realizada la eficiencia práctica podemos proceder al análisis de la eficiencia híbrida, obteniendo los resultados que se mostraran a continuación.

Para calcular la constante oculta debemos de despejar K en la siguiente fórmula:

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

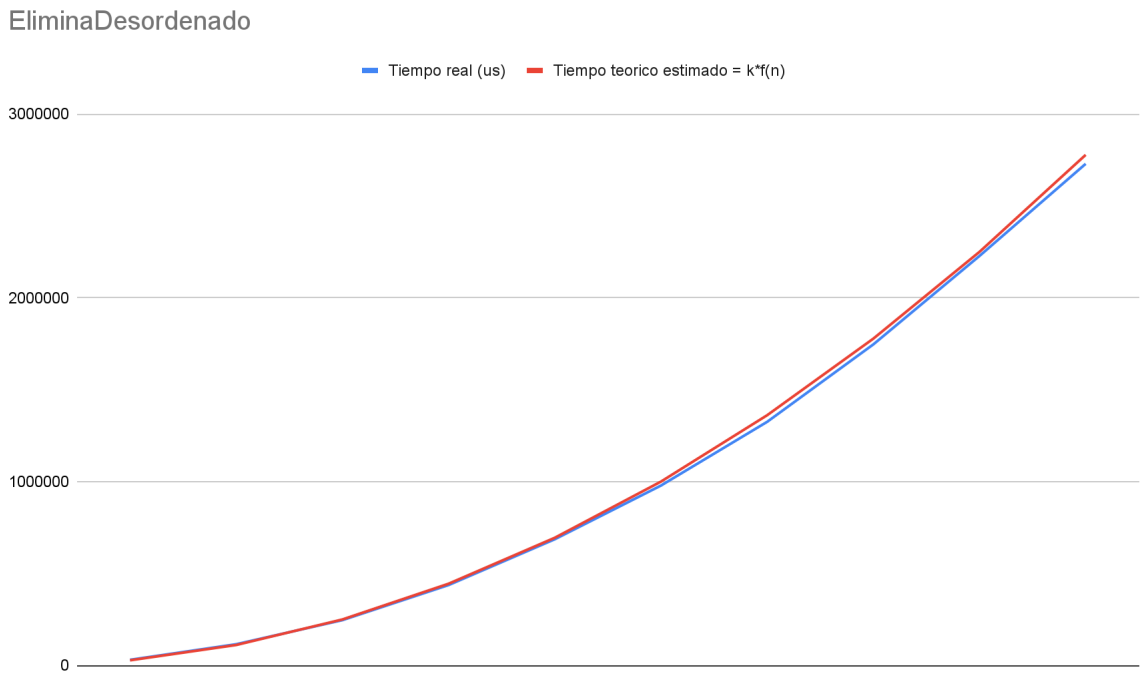
Este cálculo, lo realizaremos para todas las ejecuciones del mismo algoritmo para distintos tamaños de caso, para finalmente concluir en que nuestra K sería la media de todos los valores que hemos obtenido y se mostrará junto con los resultados obtenidos de cada algoritmo.

También se adjuntará una comparación gráfica de cada algoritmo del tiempo real frente al tiempo estimado.

5.1.- Algoritmos Iterativos

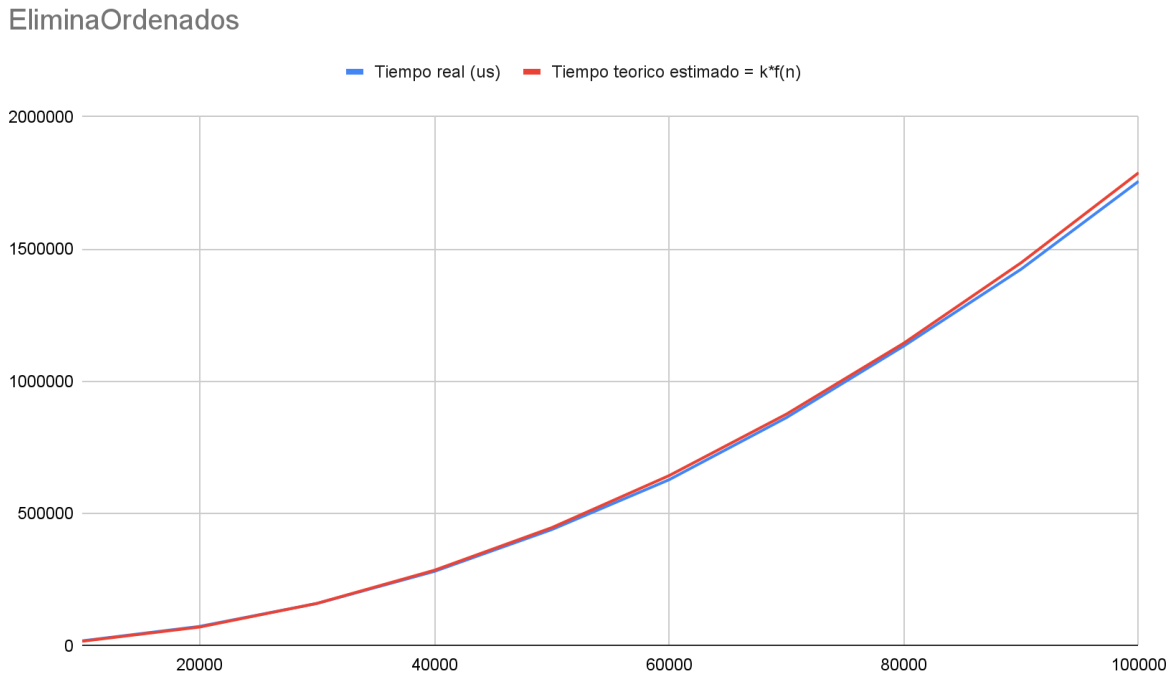
5.1.1.- Algoritmo EliminaComponentesDesordenados

Tamaño de caso	Tiempo real (us)	K=Tiempo/f(n)	Tiempo teorico estimado = k*f(n)	
10000	30509	0,00030509	27777	
20000	115289	0,00028822	111107	
30000	246262	0,00027362	249991	
40000	437180	0,00027324	444429	
50000	686285	0,00027451	694421	
60000	978417	0,00027178	999966	
70000	1325097	0,00027043	1361065	
80000	1746368	0,00027287	1777717	
90000	2227656	0,00027502	2249923	
100000	2728950	0,00027290	2777683	
Desordenado				
	k_promedio	0,00027777		



5.1.2.- Algoritmo EliminaComponentesOrdenados

Tamaño de caso	Tiempo real (us)	K=Tiempo/f(n)	Tiempo teorico estimado = k*f(n)
10000	19369	0,00019369	17881,82873
20000	73924	0,00018481	71527,31492
30000	160641	0,00017849	160936,4586
40000	281779	0,000176111875	286109,2597
50000	439768	0,0001759072	447045,7182
60000	628223	0,000174506388	643745,8342
70000	863030	0,000176128571	876209,6077
80000	1133663	0,000177134843	1144437,039
90000	1424065	0,000175810493	1448428,127
100000	1755935	0,0001755935	1788182,873
Ordenado			
	k_promedio	0,000178818287	



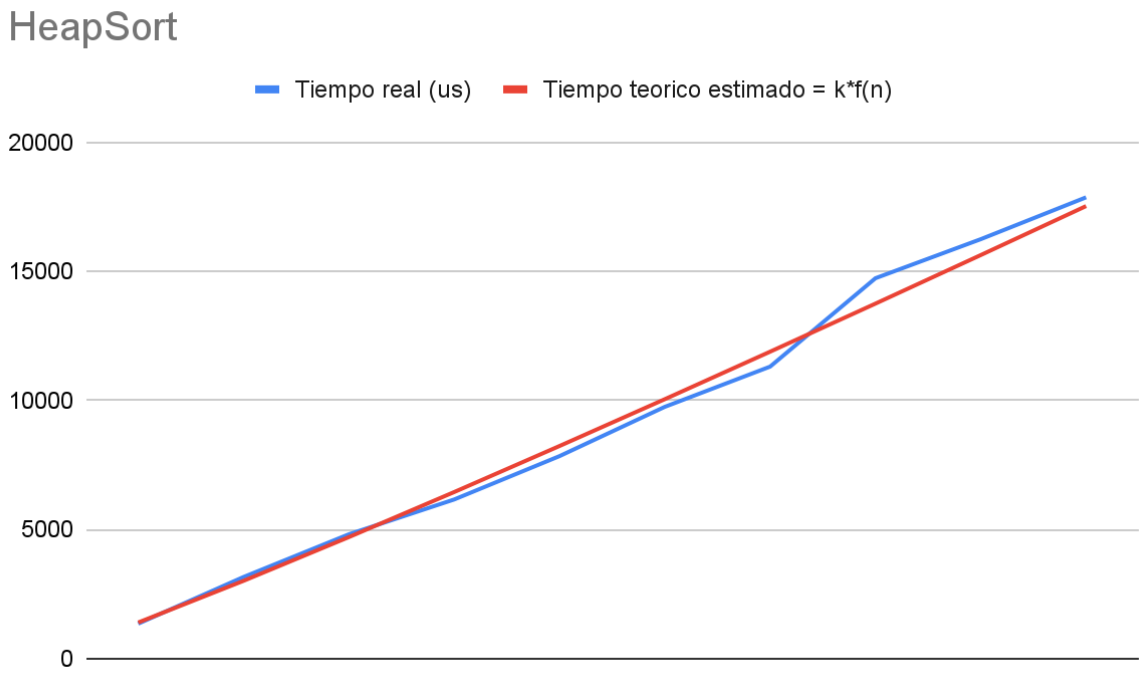
5.1.3.- Comparación Desordenados y Ordenados

Podemos apreciar observando la gráfica que los algoritmos son bastante diferentes ya que el algoritmo de eliminar componentes con el vector desordenado crece más rápido (tarde mas cuando to mayor es el numero de elementos) que el algoritmo de eliminar componentes con el vector ordenado por lo tanto el algoritmo mas eficiente es este ultimo.

5.2.- Algoritmos Recursivos

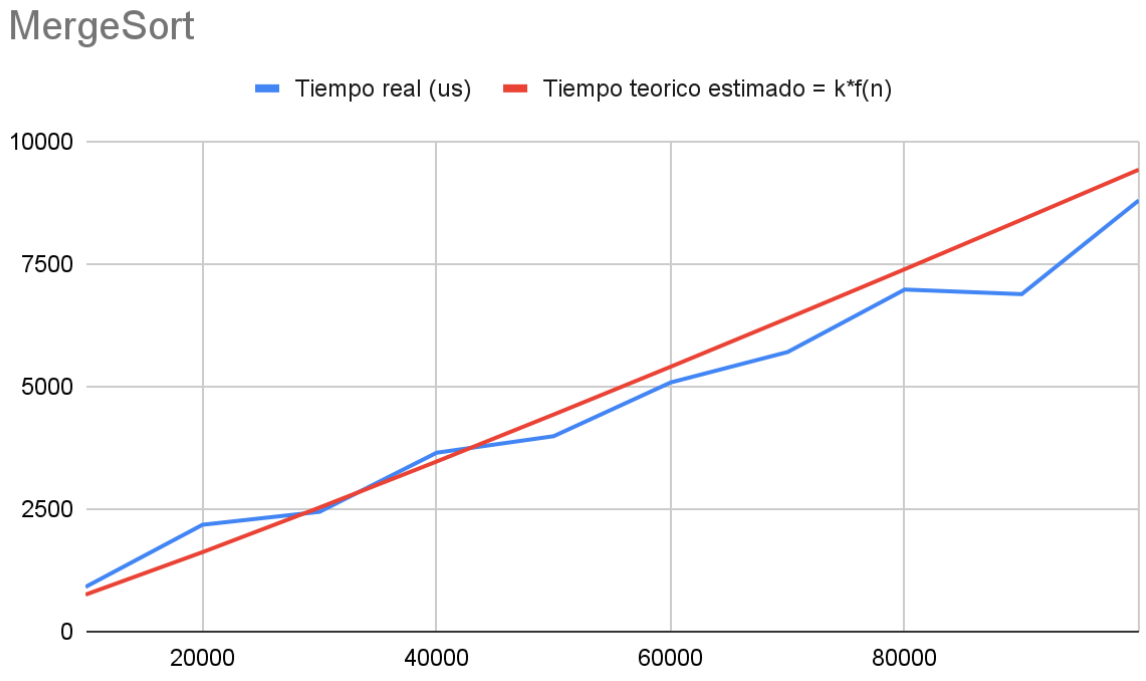
5.2.1.- HeapSort

Tamaño de caso	Tiempo real (us)	K=Tiempo/f(n)	Tiempo teorico estimado = k*f(n)
10000	1357	0,0102124426	1402,320158
20000	3164	0,01107245133	3015,710531
30000	4818	0,01079832655	4708,768038
40000	6166	0,01008326139	6453,561493
50000	7845	0,0100514807	8236,825447
60000	9754	0,01024192002	10050,74672
70000	11312	0,01004034323	11890,16263
80000	14735	0,01130837135	13751,40385
90000	16247	0,01096890265	15631,72681
100000	17868	0,01075760793	17529,00197
Heapsort			
	k_promedio	0,01055351077	

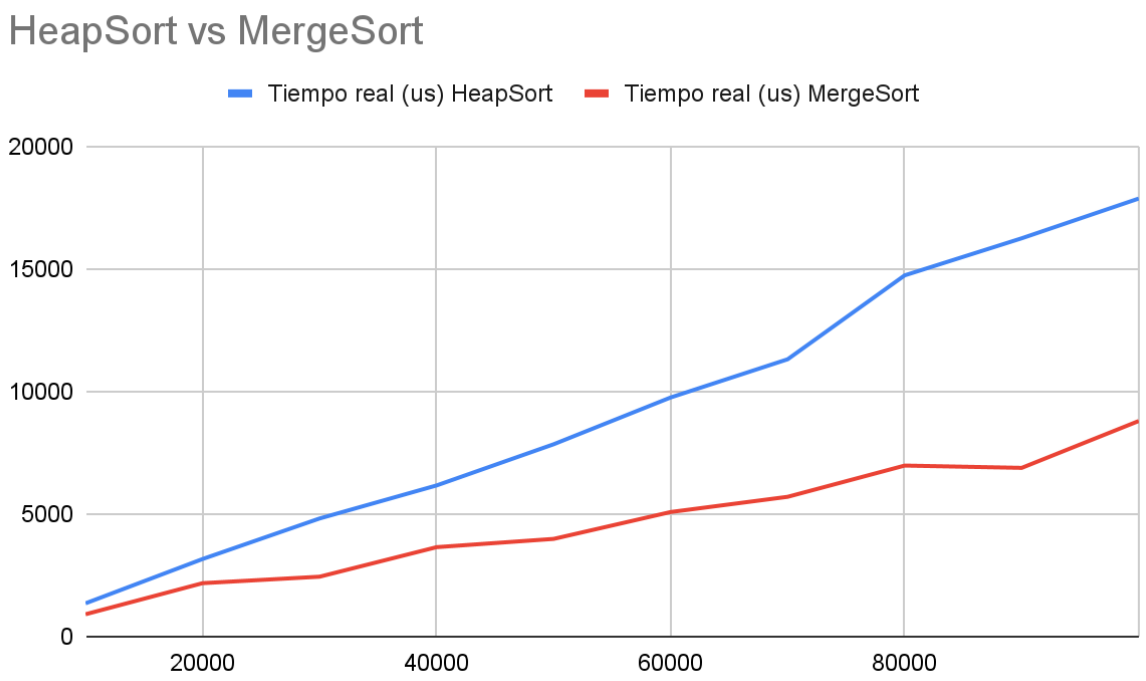


5.2.2.- MergeSort

Tamaño de caso	Tiempo real (us)	K=Tiempo/f(n)	Tiempo teorico estimado = k*f(n)
10000	911	0,006855958151	753,6635506
20000	2179	0,007625433457	1620,764769
30000	2443	0,005475365661	2530,682326
40000	3648	0,005965575342	3468,404873
50000	3985	0,005105819069	4426,803022
60000	5081	0,005335164613	5401,677655
70000	5702	0,00506100045	6390,254136
80000	6976	0,005353729116	7390,560417
90000	6882	0,00464627242	8401,122001
100000	8794	0,005294515564	9420,794383
Mergesort			
	k_promedio	0,005671883384	



5.2.3.- Comparación MergeSort y HeapSort



Podemos apreciar observando la gráfica que los algoritmos son parecidos aunque el MergeSort es siempre más rápido que el HeapSort, conforme vamos teniendo un tamaño de caso más grande apreciamos que la diferencia entre MergeSort y HeapSort es mayor y podemos ver claramente que el algoritmo MergeSort es más rápido.

6.- Ejecución del Código

En este apartado explicaremos cómo ejecutar nuestro código, así como también porque hemos decidido optar por esta resolución del problema.

Hemos decidido modificar el fichero que se nos proporcionaba al inicio de la práctica e incluir los distintos algoritmos y ejercicios que nos piden en la resolución de la práctica, ya que hemos decidido que sería la forma más sencilla de probar todos los códigos sin necesidad de compilar uno a uno los distintos algoritmos.

Lo primero es compilar el fichero Practica1.cpp con :

g++ -o <nombre_del_ejecutable> Practica1.cpp

Una vez compilado el ejecutable procedemos a ejecutarlo con:

./<nombre_ejec> <fichero_salida> <semilla> <tam_caso1> … <tam_caso_n>

Hemos decidido dejar la forma del main original que se proporcionaba de ejemplo debido a que con la semilla generadora de números aleatorios podíamos probar los diferentes algoritmos con el mismo vector generado aleatoriamente, de esta forma nos aseguramos de que los casos en los que se ejecutan los algoritmos son idénticos y los resultados son más fiables.

Cuando ejecutemos el código, se nos abrirá un menú simple en el que podemos elegir los distintos algoritmos que hemos utilizado a lo largo de la práctica. Cabe destacar que hay dos casos particulares, el algoritmo de eliminarComponentesOrdenados que hacemos uso del algoritmo MergeSort para ordenar el vector de números aleatorios(pero no contabilizamos el tiempo que tarda MergeSort en ordenar el vector original) y el algoritmo de Hanoi que nos pedirá que introduzcamos más valores adicionales.

Por último en el fichero de salida van a aparecer los datos más significativos de la ejecución del código, como el tamaño de caso y el tiempo de ejecucion, asi como tambien en el caso de los algoritmos de ordenación y eliminar repetidos si el tamaño de caso es menor que 100 podremos visualizar la salida del vector modificado.