Práctica 1: Análisis de eficiencia de algoritmos

Manuel Jesús Núñez Ruiz
Mario Genol Morales
Miguel Pedregosa Pérez

Índice

1. Análisis del algoritmo de búsqueda binaria

1.1. El algoritmo

```
int BuscarBinario(double *v, const  int ini, const int fin, const double x){

        int centro;                                                    O(1)
        if(ini > fin) return -1;                                       O(1)

        centro = (ini+fin)/2;                                          O(1)
        if(v[centro] == x) return centro;                              O(1)
        if(v[centro]>x) return BuscarBinario(v, ini, centro-1, x);     O(1)+O(T(n/2))
        return BuscarBinario(v, centro+1, fin, x);                     O(T(n/2))
}
```

1.2. Eficiencia teórica

1.2.1. El peor de los casos y el mejor

$\text{Max}\{O(1), O(1), O(1), O(1), O(1+T(n/2)), O(T(n/2))\} = O(1+T(n/2))$

$$T(n) \begin{cases} 1 & \text{caso base} \\ 1+T(n/2) & \text{caso general} \end{cases}$$

$$T(n) = 1 + T(n/2) \qquad n = 2^m$$
$$T(2^m) = 1 + T(2^{m-1})$$

Parte Homogénea:

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

Parte no Homogénea

$1 = b_1^m \cdot q_1(m) \Rightarrow b_1 = 1$ y $q_1(m) = 1$ con grado $d_1 = 0$

$$p(x) = (x-1)^2$$

$$t_m = c_1 \cdot 1^m + c_2 \cdot m \cdot 1^m$$
$$m = \log_2(n)$$

$$t_n = c_1 \cdot 1^{\log(n)} + c_2 \cdot \log_2(n) \cdot 1^{\log(n)}$$

Este algoritmo es **$O(\log_2(n))$** y **$\Omega(1)$**.

1.2.2. El caso medio

Ya que la eficiencia en caso medio dependerá del dominio y cardinalidad de los elementos del vector, comenzamos por calcular el caso medio del algoritmo aplicado a un vector de 2^{k-1} elementos (correspondiente a un árbol completo) en el que sabemos que el elemento buscado está.

Si representamos el vector como un árbol binario de búsqueda, vemos que el número de "pasos" que llevará encontrarlo es igual a la profundidad del nivel del árbol en el que se encuentra. Así, el número de pasos medio para encontrar el elemento será la media ponderada de la ordinalidad de los niveles.

Sabemos que la probabilidad de que es elemento esté en la profundidad d es $\frac{2^d}{N}$, ya que hay 2^d elementos de profundidad d. Así, obtenemos el resultado mediante la media ponderada:

$$T(N) = \sum_{i=1}^{lg(N+1)} \frac{2^i}{n} \cdot i = \frac{1}{N} \cdot \sum_{i=1}^{lg(N+1)} 2^i \cdot i$$

Y desarrollamos:

$$T(N) = \frac{1}{N} \cdot \sum_{i=1}^{lg(N+1)} 2^i \cdot i = \frac{1}{N} \cdot \left(2 + (lg(N) - 2) \cdot 2^{lg(N)}\right) = \frac{1}{N} \cdot (2 + N \cdot lg(N) - 2 \cdot N) = \frac{2}{N} + lg(N) - 2$$

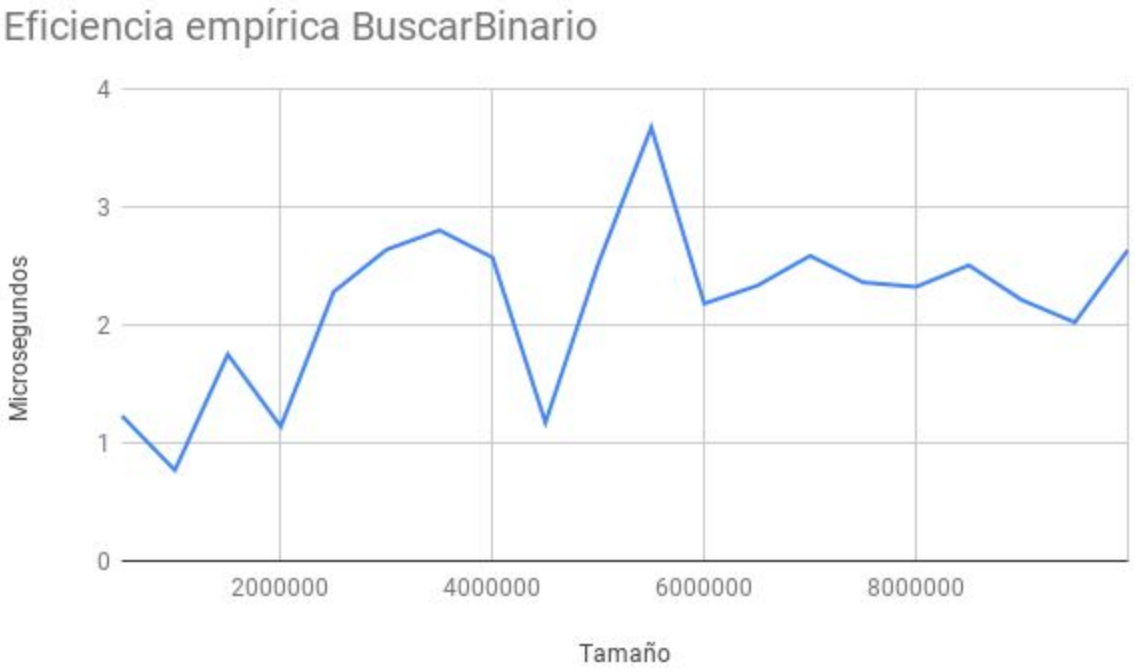
Así, vemos que, en el caso medio con N grande, el elemento está en el antepenúltimo nivel y que el caso medio, al igual que el caso peor, es O(lg(N)).

No obstante, esto es un análisis de un caso particular, aunque es bastante fácil la generalización. Para un caso cualquiera definiremos: probabilidad de que el elemento esté en el vector (P_a, mediante un análisis del dominio y cardinalidad de los elementos), y el número de elementos que faltan para completar el último nivel del árbol equivalente (t, que es igual a la solución de la ecuación $N + t = 2^{techo(lg(N))}$). Así, T(N) (que seguirá siendo O(lg(N)) generalizado queda de la siguiente manera:

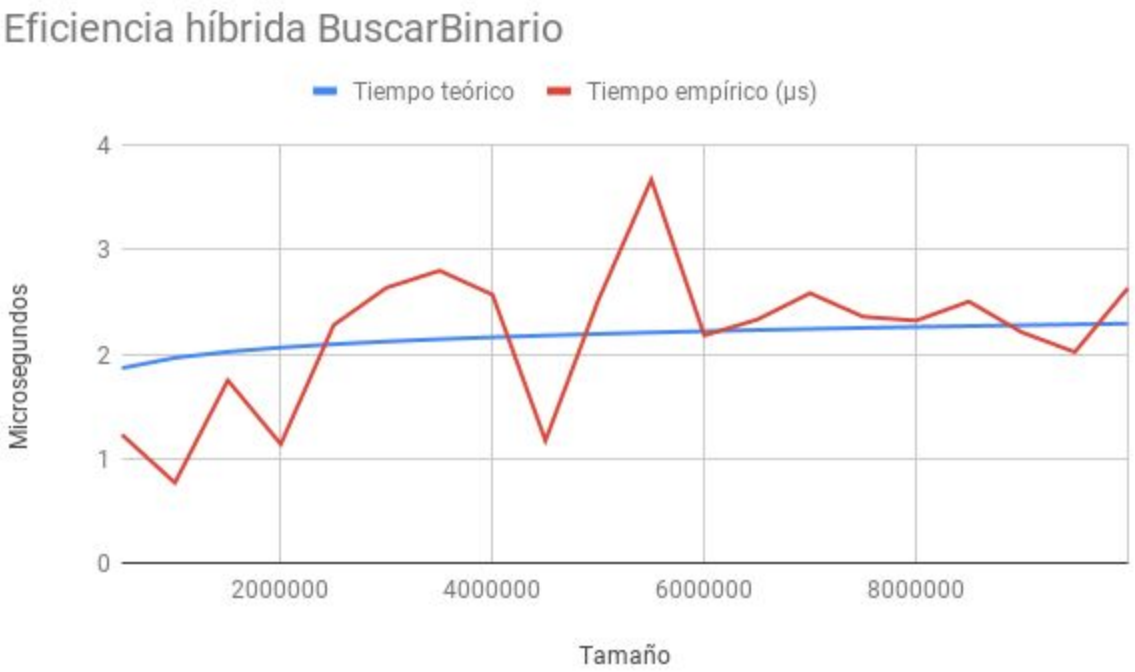
$$T(N) = \left(P_a - \frac{t}{N}\right) \cdot \left(\frac{2}{N} + lg(N) - 2\right) + \left(1 - P_a + \frac{T}{N}\right) \cdot techo(lg(N))$$

1.3. Eficiencia empírica

Buscar_Binario n	Tiempo empírico
500000	1230
1000000	769
1500000	1751
2000000	1139
2500000	2281
3000000	2639
3500000	2802
4000000	2572
4500000	1173
5000000	2521
5500000	3674
6000000	2181
6500000	2333
7000000	2586
7500000	2361
8000000	2323
8500000	2506
9000000	2211
9500000	2022
10000000	2633



1.4. Eficiencia híbrida



Como podemos ver en la gráfica anterior, la gráfica de la eficiencia empírica del algoritmo no se ajusta muy bien a la gráfica del logaritmo. Este es debido a las constantes ocultas del algoritmo de búsqueda binario.

2. Análisis del algoritmo heap sort

2.1. El algoritmo

```
1 void heapsort( int T[] , int num_elem )
2 {
3       int i;                                          O(1)
4       for( i = num_elem / 2 ; i >= 0 ; i--)           O(n·log2(n))
5               reajustar (T, num elem , i);
6       for( i = num_elem - 1 ; i >= 1 ; i--)           O(n·log2(n))
7       {
8               int aux = T[0];                         O(1)
9               T[0] = T[i];                            O(1)
10              T[i] = aux;                             O(1)
11              reajustar(T, i , 0);                    O(log2(n))
12      }
13 }
```

```
2 void reajustar ( int T[] , int num_elem , int k )
3 {
4       int j;                                                          O(1)
5       int v;                                                          O(1)
6       v = T[k];                                                       O(1)
7       bool esAPO = false;                                             O(1)
8       while( ( k < num_elem / 2 ) && ! esAPO)                         O(log2(n))
9       {
10              j = k + k + 1;                                          O(1)
11              if( ( j < ( num_elem − 1 ) ) && (T [ j ] < T [ j + 1 ]))        O(1)
12                      j++;                                            O(1)
13              if( v >= T [ j ] )                                      O(1)
14                      esAPO = true;                                   O(1)
15              T[k] = T[j];                                            O(1)
16              k = j;                                                  O(1)
17      }
18      T[k] = v;                                                       O(1)
19 }
```

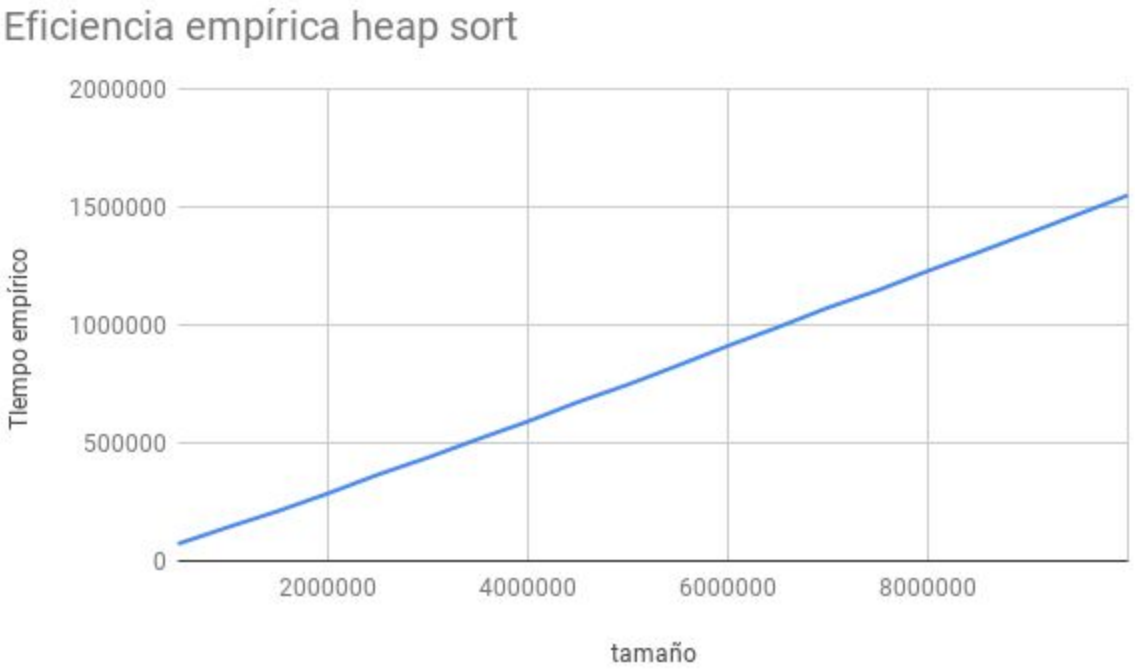
2.2. Eficiencia teórica

$\text{Max}\{O(1), O(2 \cdot n \cdot \log_2(n))\} = O(2 \cdot n \cdot \log_2(n)) = O(n \cdot \log_2(n))$

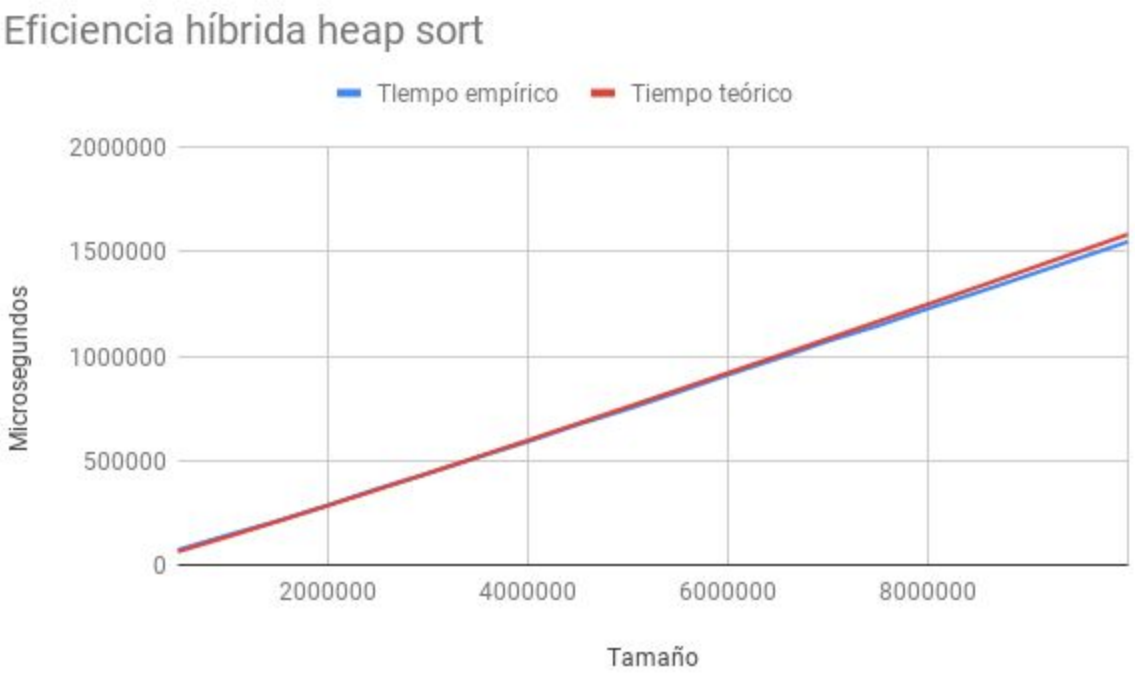
La eficiencia de este algoritmo es **$O(n \cdot \log_2(n))$** y **$\Omega(n \cdot \log_2(n))$**

2.3. Eficiencia empírica

heap_sort n	Tiempo empírico Heap Sort
500000	72386
1000000	143268
1500000	211513
2000000	286127
2500000	365959
3000000	438247
3500000	516180
4000000	591224
4500000	673410
5000000	747358
5500000	828127
6000000	911436
6500000	990140
7000000	1073066
7500000	1146447
8000000	1228861
8500000	1306721
9000000	1387179
9500000	1468324
10000000	1549203



2.4. Eficiencia híbrida



La gráfica del heap sort se ajusta bastante bien a la función de $n \cdot \log_2(n)$, por lo que parece que apenas tiene constantes ocultas que empeoren la eficiencia del algoritmo.

3. Análisis del algoritmo merge sort

3.1 El algoritmo

```
static void insercion_lims(int T[], int inicial, int final){
  int i, j;                                   O(1)
  int aux;                                    O(1)
  for (i = inicial + 1; i < final; i++) {     O(n)
    j = i;                                    O(1)
    while ((T[j] < T[j-1]) && (j > 0)) {      O(n)
      aux = T[j];                             O(1)
      T[j] = T[j-1];                          O(1)
      T[j-1] = aux;                           O(1)
      j--;                                    O(1)
    };
  };
}

static void fusion(int T[], int inicial, int final, int U[], int V[])
{
  int j = 0;                                  O(1)
  int k = 0;                                  O(1)
  for (int i = inicial; i < final; i++)       O(n)
    {
      if (U[j] < V[k]) {                      O(1)
          T[i] = U[j];                        O(1)
          j++;                                O(1)
      } else{                                 O(1)
          T[i] = V[k];                        O(1)
          k++;                                O(1)
      };
    };
}


static void mergesort_lims(int T[], int inicial, int final)
```

```
{
 if (final - inicial < UMBRAL_MS)          O(1)
   {
    insercion_lims(T, inicial, final);     O(n^2)
   } else {                                O(1)
    int k = (final - inicial)/2;           O(1)

    int * U = new int [k - inicial + 1];   O(1)
    assert(U);                             O(1)
    int l, l2;                             O(1)
    for (l = 0, l2 = inicial; l < k; l++, l2++)  O(n/2) = O(n)
      U[l] = T[l2];                        O(1)
    U[l] = INT_MAX;                        O(1)

    int * V = new int [final - k + 1];     O(1)
    assert(V);                             O(1)
    for (l = 0, l2 = k; l < final - k; l++, l2++) O(n/2) = O(n)
      V[l] = T[l2];                        O(1)
    V[l] = INT_MAX;                        O(1)

    mergesort_lims(U, 0, k);               T(n/2)
    mergesort_lims(V, 0, final - k);       T(n/2)
    fusion(T, inicial, final, U, V);       O(n)
    delete [] U;                           O(1)
    delete [] V;                           O(1)
   };
}
```

3.2. Eficiencia teórica

$$T(n) \begin{cases} n^2 & n < UMBRAL\_MS \\ 11+3n+2T(n/2) & n >= UMBRAL\_MS \end{cases}$$

$$T(n)= 11 + 3n + 2T(n/2) \qquad n = 2^m$$
$$T(2^m) = 11 + 3*2^m + 2*T(2^{m-1})$$

Parte Homogénea:

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

$$p_H(x) = x-2$$

Parte no Homogénea:

$$2^m*3 = b_1^m \cdot q_1(m) \Rightarrow b_1 = 2 \text{ y } q_1(m) = 3 \text{ con grado } d_1 = 0$$

$$p(x) = (x-2)^2$$

$$t_m = c_1 \cdot 2^m + c_2 \cdot 2^m \cdot m$$

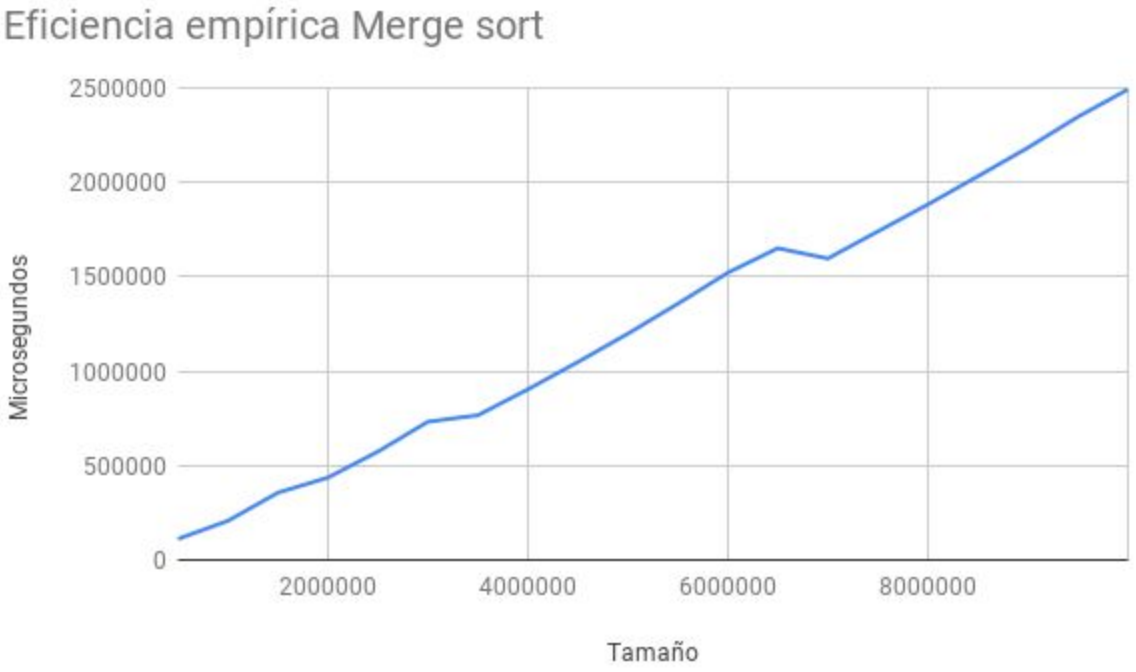
$$m = \log_2(n)$$

$$t_n = c_1 \cdot n + c_2 \cdot n \cdot \log_2(n)$$

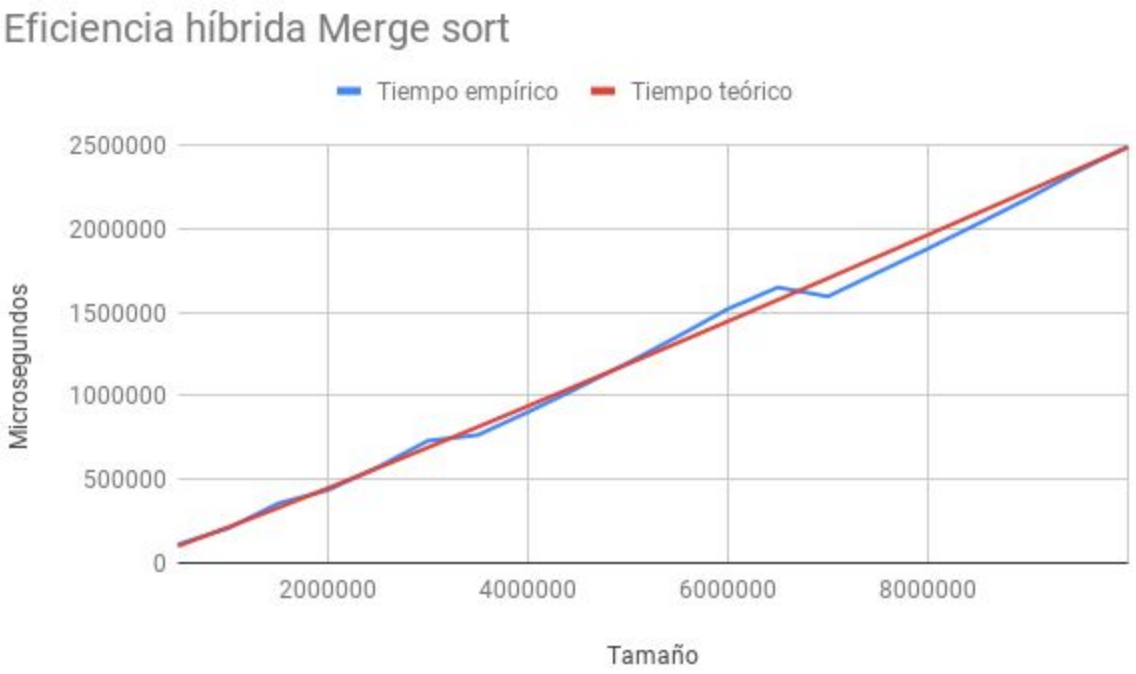
Este algoritmo es **$O(n \cdot \log_2(n))$**.

3.3. Eficiencia empírica

mergesort n	Tiempo empírico Merge Sort
500000	111626
1000000	207512
1500000	356533
2000000	436535
2500000	574079
3000000	732271
3500000	766512
4000000	903908
4500000	1048690
5000000	1198423
5500000	1357239
6000000	1521754
6500000	1651078
7000000	1596643
7500000	1740074
8000000	1882683
8500000	2031964
9000000	2183227
9500000	2346543
10000000	2491872



3.4. Eficiencia híbrida



Esta gráfica no se ajusta tan bien a la función de $O(n \cdot \log_2(n))$, hay veces que incluso supera la gráfica del tiempo empírico a la gráfica de la función O. Esto se debe a como se ha indicado antes, a las constantes ocultas.

4. Eficiencia del algoritmo de las torres de hanoi

4.1 El algoritmo

```
void hanoi (int M, int i, int j)
{
 if (M > 0)                                  O(1)
   {
    hanoi(M-1, i, 6-i-j);                    T(n-1)
    cout << i << " -> " << j << endl;        O(1)
    hanoi (M-1, 6-i-j, j);                   T(n-1)
 }
}
```

4.2 Eficiencia teórica

$$T(n) \begin{cases} 1 & \text{caso base} \\ 2+2T(n-1) & \text{caso general} \end{cases}$$

Parte Homogénea:

$$T(n) - 2*T(n-1) = 0$$

$$p_H(x) = x-2$$

Parte no Homogénea:

$2 = b_1^m \cdot q_1(m) \Rightarrow b_1 = 1$ y $q_1(m) = 3$ con grado $d_1 = 0$

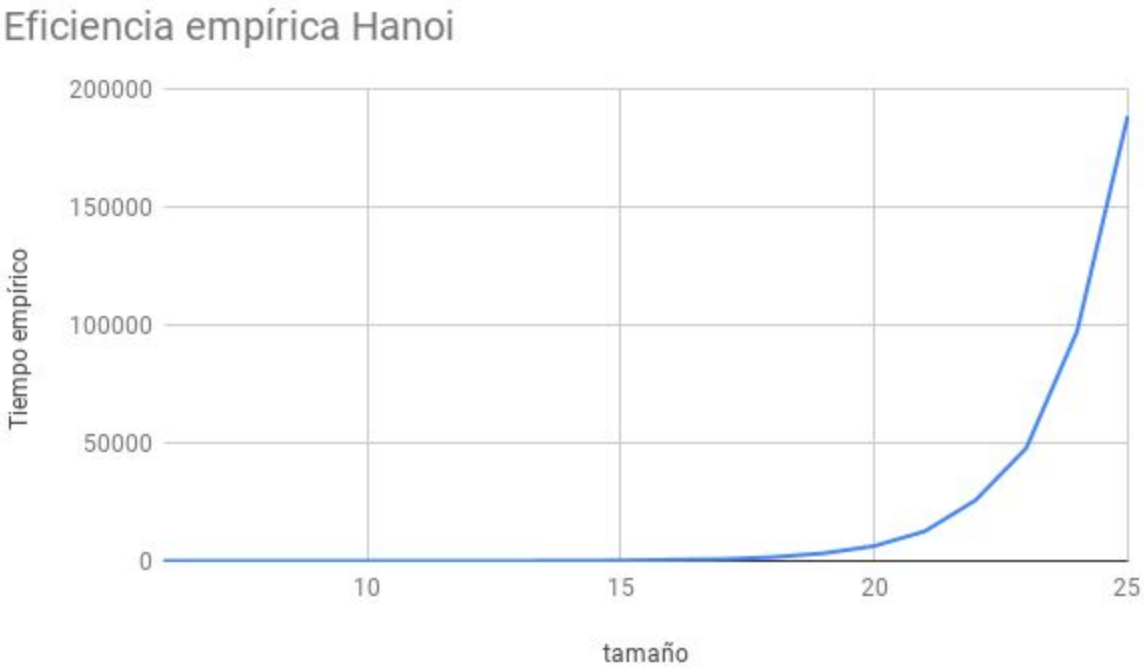
$$p(x) = (x-2)\cdot(x-1)$$

$$t_n = 2^n \cdot c_1 + 1^n \cdot c_2$$

Este algoritmo es **$O(2^n)$** y **$\Omega(1)$**.

4.3. Eficiencia empírica

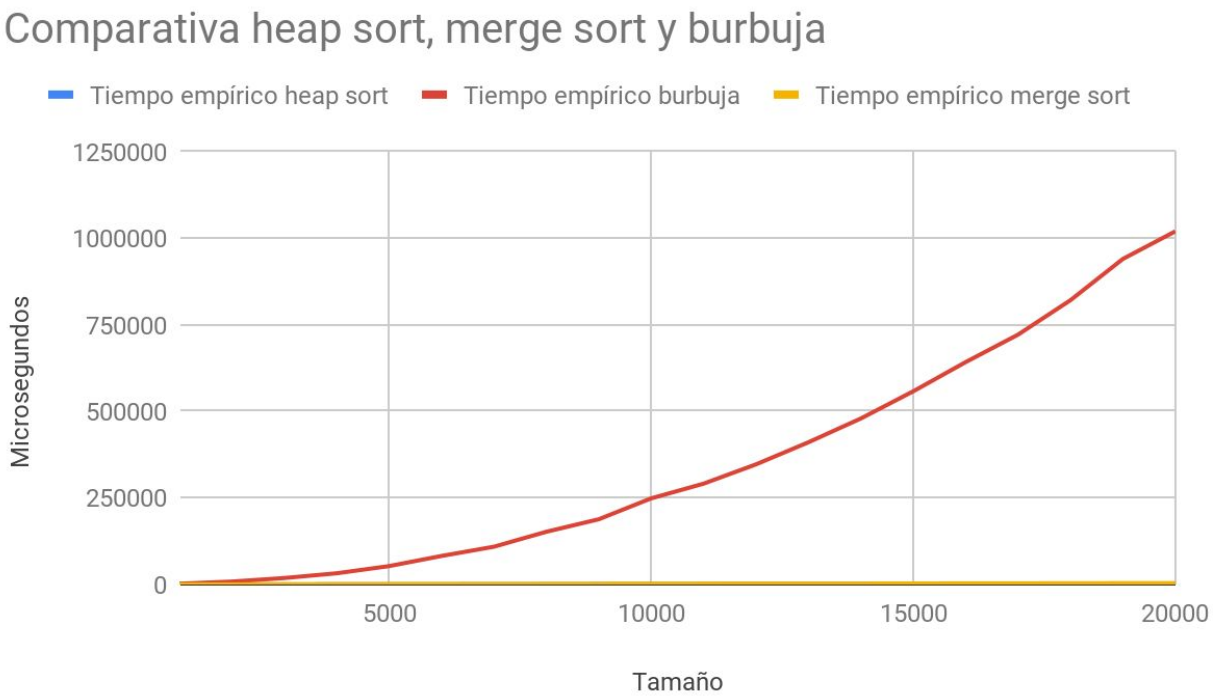
Hanoi n	Tiempo empírico
6	0
7	1
8	1
9	3
10	6
11	12
12	23
13	47
14	99
15	198
16	525
17	743
18	1628
19	3247
20	6325
21	12583
22	25750
23	47830
24	97271
25	188502



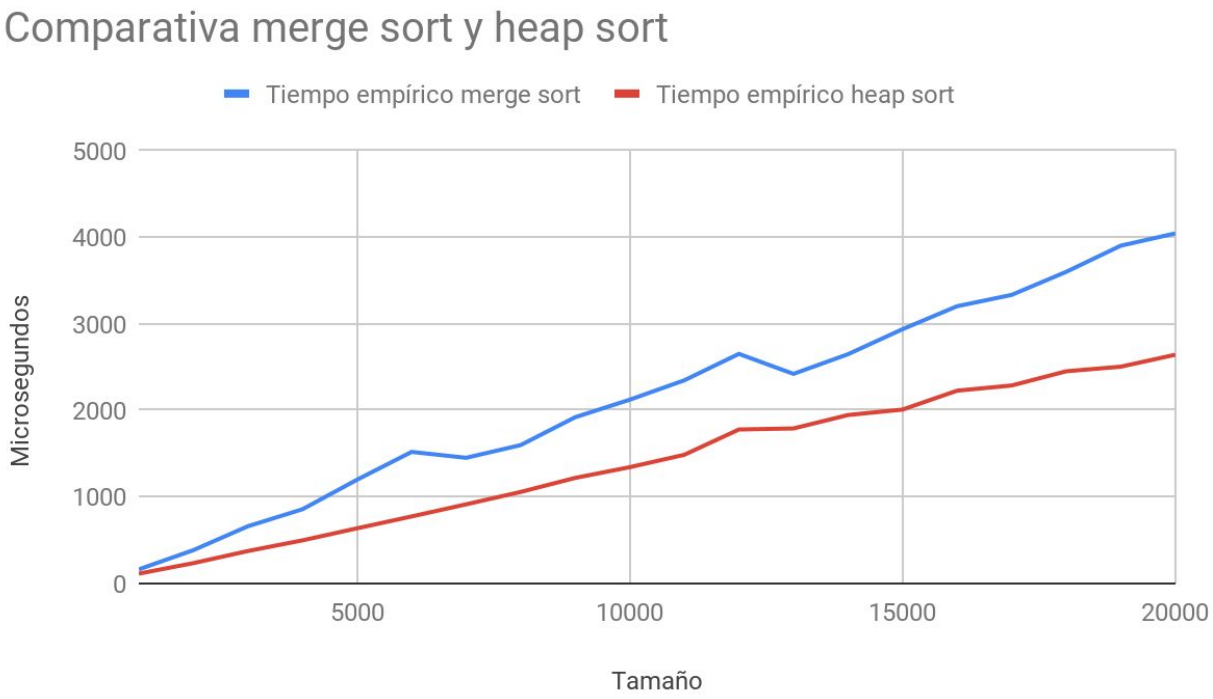
4.4. Eficiencia híbrida

Como podemos observar en la gráfica del tiempo empírico del algoritmo que resuelve el problema de las torres de Hanoi se ajusta muy bien a la gráfica de $O(2^n)$.

5. Comparativa entre los algoritmos merge sort, heap sort y burbuja (algoritmos de ordenación)



Como podemos observar en la gráfica de comparación de tiempos empíricos de los algoritmos de heap sort, merge sort y burbuja de aquí arriba, claramente el burbuja es el peor algoritmo. Esto concuerda con la teoría, pues el burbuja es $O(n^2)$ mientras que los otros dos son $O(n \cdot \log_2(n))$.



Entre el merge sort y el heap sort podemos observar que el heap sort es mejor, aunque los dos sean $O(n \cdot \log_2(n))$, esto es debido a que las constantes ocultas del heap sort son más bajas que las del merge sort.

6. Scripts usados

Script usado para BuscarBinario, heapsort y Hanoi:

```
1. #!/bin/sh
2.
3. lista="buscar_binario heap_sort"
4. cpp=".cpp"
5. arch=resultados_grupal.txt
6.
7.
8. echo "" > $arch
9. i="1"
10.
11.for p in $lista
12.do
13.        if [ ! -f ./$p ]; then
14.                g++ -o $p $p$cpp -std=gnu++0x
```

```
        fi
        if [ -f ./$p ]; then
                echo $p >> $arch
                while [ $i -le 20 ]
                do
                        n=$(( $i * 500000 ))
                        ./$p $n >> $arch
                        i=$(( $i + 1 ))
                        echo "Hecho $p $n"
                done
                i="1"
                echo "Hecho $p \n"
        fi
done

if [ ! -f ./hanoi ]; then
        g++ -o hanoi hanoi.cpp -std=gnu++0x
fi
if [ -f ./hanoi ]; then
        echo hanoi >> $arch
        while [ $i -le 20 ]
        do
                n=$(( $i + 5 ))
                ./hanoi $n >> $arch
                i=$(( $i + 1 ))
                echo "Hecho hanoi $n"
        done
fi
```

Script usado para merge sort y burbuja:

```
#!/bin/sh

lista="mergesort"

cpp=".cpp"

arch=resultados_mergeburb.txt


echo "" > $arch
i="1"

for p in $lista
do
        if [ ! -f ./$p ]; then
                g++ -o $p $p$cpp -std=gnu++0x
        fi
```

```
16.         if [ -f ./$p ]; then
17.                 echo $p >> $arch
18.                 while [ $i -le 20 ]
19.                 do
20.                         n=$(( $i * 500000 ))
21.                         ./$p $n >> $arch
22.                         i=$(( $i + 1 ))
23.                         echo "Hecho $p $n"
24.                 done
25.                 i="1"
26.                 echo "Hecho $p \n"
27.         fi
28.done
29.
30.if [ ! -f ./burbuja ]; then
31.         g++ -o burbuja burbuja.cpp -std=gnu++0x
32.fi
33.if [ -f ./burbuja ]; then
34.         echo burbuja >> $arch
35.         while [ $i -le 20 ]
36.         do
37.                 n=$(( $i * 1000 ))
38.                 ./burbuja $n >> $arch
39.                 i=$(( $i + 1 ))
40.                 echo "Hecho burbuja $n"
41.         done
42.fi
```