Algorítmica (2018-2019)
Doble Grado en Ingeniería Informática Y Matemáticas
Universidad de Granada

Memoria Práctica 1
Grupo Azul

Irene Huertas González, Javier Alcántara García y César Muñoz Reinoso

20 de marzo de 2019

Índice

1. Máquinas empleadas

Antes de empezar con la comparación de eficiencia es necesario ver qué máquinas se han empleado para esta:

1. HP Pavilion, 2 año:
   - **Procesador:** Intel Core i7, 6ª generación.
   - **Memoria RAM:** 12,0 GB.
   - **Tipo de sistema:** Sistema operativo de 64 bits, procesador x64.
2. Dell XPS 15, 0.5 años:
   - **Procesador:** Intel Core i7, 7ª generación.
   - **Memoria RAM:** 8,0 GB.
   - **Tipo de sistema:** Sistema operativo de 64bits, procesador x64.
3. Xiaomi Mi NoteBook Pro, 1.5 años:
   - **Procesador:** Intel Core i7, 8ª generación.
   - **Memoria RAM:** 16,0 GB.
   - **Tipo de sistema:** Sistema operativo de 64bits, procesador x64.

2. Diferencia de eficiencia entre máquinas

Ahora pasamos a analizar las diferencias de la ejecución de los algoritmos entre las computadoras Sony Vaio, HP Paviliony Xiaomi Mi Notebook Bro. Tras realizar la práctica individual, ya tenemos todos los datos necesarios para llevar a cabo la comparación. Estos son los resultados obtenidos:

2.1. Algoritmo 1

2.1.1. Eficiencia teórica

Hemos consensuado la eficiencia teórica de el algoritmo 1 con codigo:

```
int pivotar (double *v , const int ini , const int fin) {
  double pivote = v[ini], aux ;
  int i = ini + 1, j = fin ;

  while (i<=j){
    while (v[i] < pivote && i <= j) i++;
    while (v[j] >= pivote && j >= i) j--;

    if (i<j){
      aux = v[i];
      v[i] = v[j];
      v[j] = aux;
    }
  }

  if (j > ini) {
    v[ini] = v[j] ;
    v[j] = pivote ;
  }
  return j ;
}
```

Consideramos el peor caso: que el pivote corresponda a un extremo. El vector está casi ordenado

$$T(n) = 2 + 3 + 1 + \sum_{i=1}^{i>j}(\sum_{i=ini}^{n-t}(6) + \sum_{j=fin}^{t}(6) + max(7, 0+1)) + max(6, 0) + 1 =$$

$$6 + \sum_{i}^{i>j}(\sum_{i}^{n-t}(6) + \sum_{j}^{t}(6) + 8) + 7 = 6 + \sum_{i}^{i>j}(6(n-t-i) + 6(t-j) + 8) + 7)$$

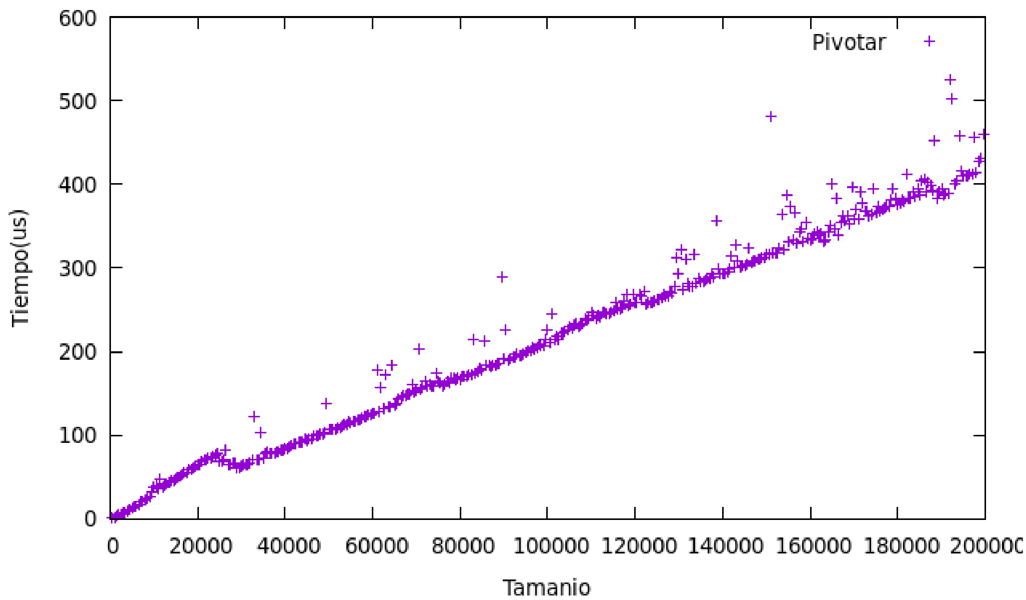
$$T(n) = 6 + 6(n-t-i) + 6(t-j) + 15$$

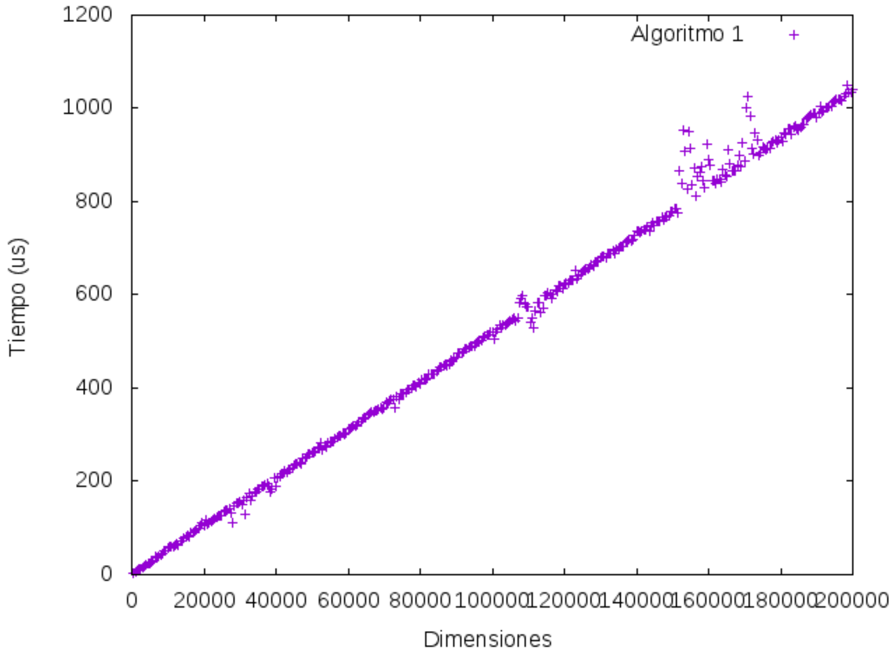
En las dos primeras líneas se realizan operaciones elementales, orden O(1) hasta que se llega al primer while. Al ponernos en el peor caso, este primer while sólo se ejecutará un número muy pequeño de veces, por no decir una sola vez, ya que el vector estará casi ordenado y el pivote será un extremo o cercano a uno.

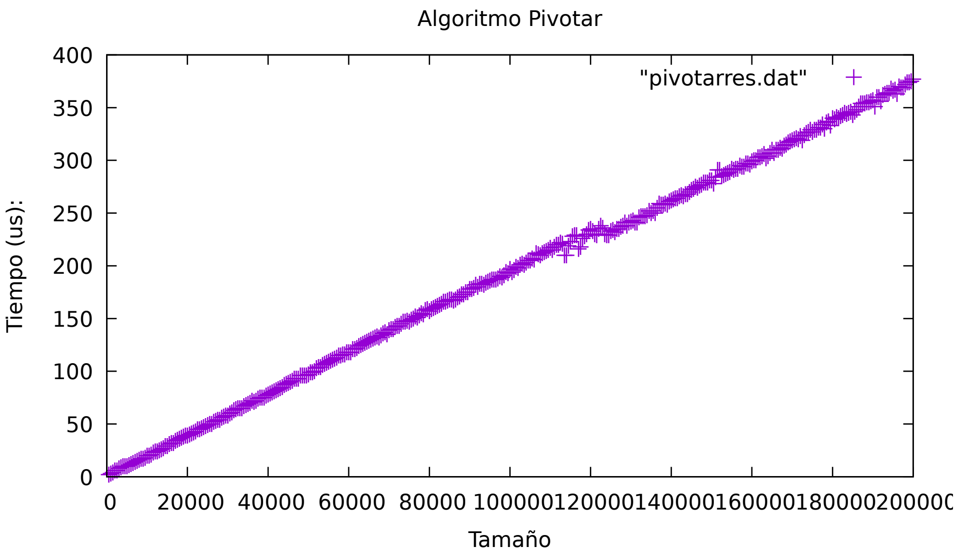
Dentro del while nos encontramos dos bucles while sin anidar. Como el pivote se quedaría cercano al extremo, en vez de que se nos particione el vector en dos listas de n/2, se nos parte en una de n-t y otra de tamaño t. t tendrá valores muy bajos (0, 1 o como mucho 2 para que se cumpla la condición del peor caso).

Lo de max(7,0) está porque para la eficiencia teórica ante un caso de if-else, nos interesa el que tiene más carga, es decir, mayor número de operaciones elementales. Como no hay else se pone un 0 en las OE del else. Lo mismo ocurre con el if que está fuera del while. Concluimos que el orden de eficiencia es $O(n)$, eficiencia lineal

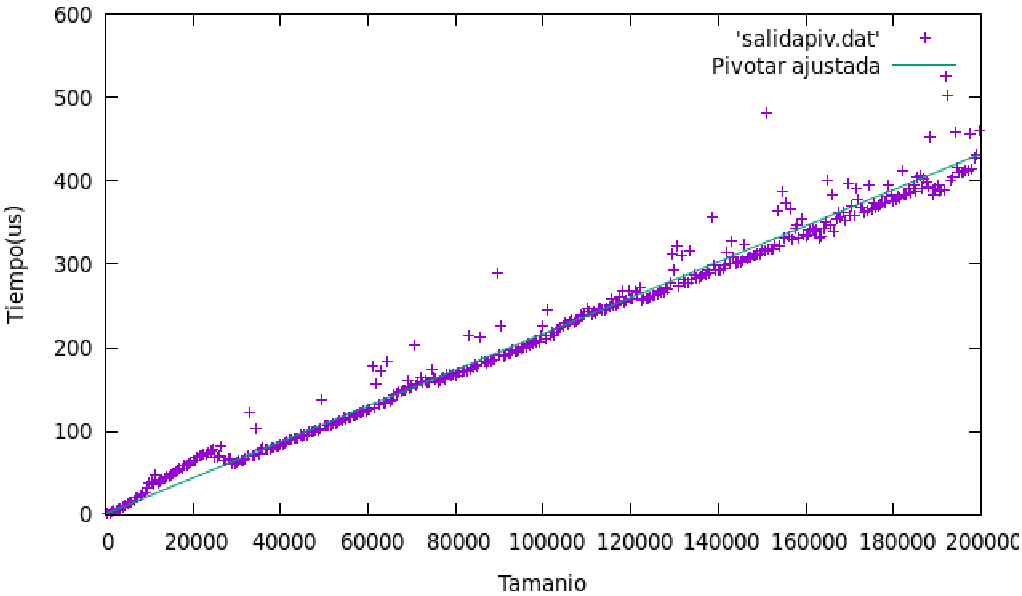
2.1.2. Eficiencia empírica

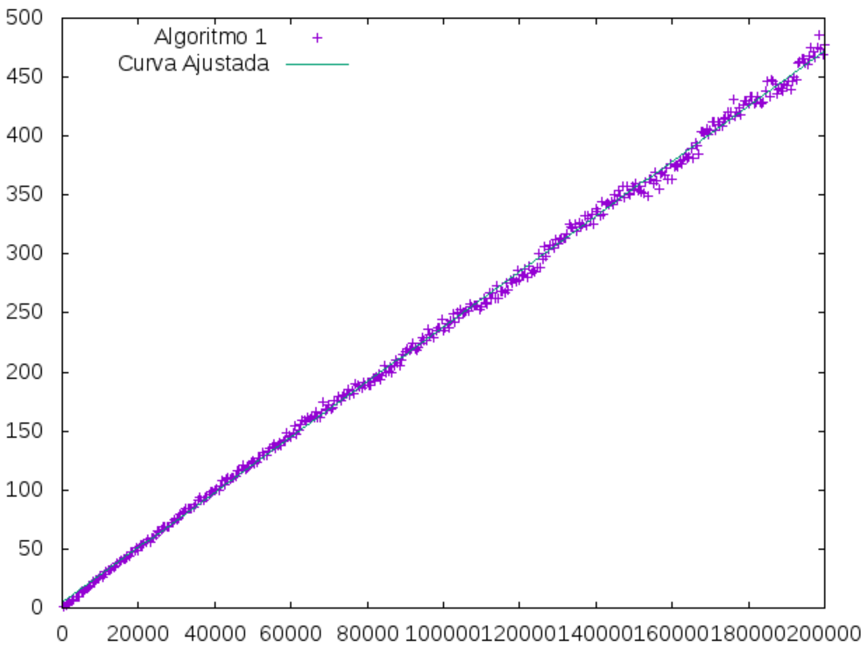


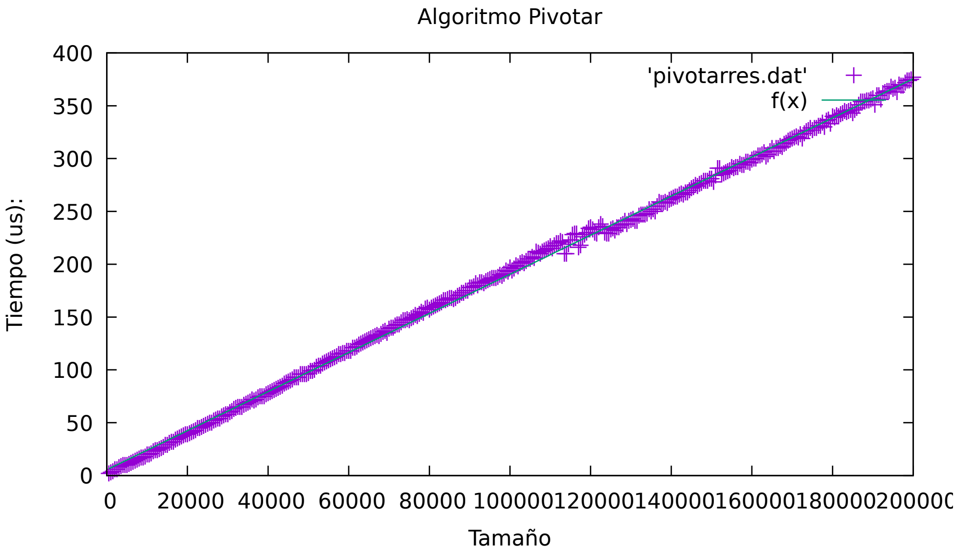

Algoritmo 1
Tiempo (us)
Dimensiones
0
200
400
600
800
1000
1200
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000


Algoritmo Pivotar
"pivotarres.dat"
Tiempo (us):
Tamaño
0
50
100
150
200
250
300
350
400
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000

2.1.3. Eficiencia híbrida







2.2. Algoritmo 2

Para el algoritmo 2 hemos tenido la siguiente eficiencia teórica:

```
int Busqueda (int *v , int n , int elem) {

  int inicio , fin , centro;

  inicio = 0;
  fin = n-1;
  centro = (inicio + fin)/2;
  while ((inicio <= fin) && (v[centro] != elem)){
    if (elem < v[centro])
      fin = centro - 1;
    else
      inicio = centro + 1;

    centro = (inicio + fin)/2;
  }
  if (inicio > fin)
    return - 1;


  return centro;
}
```

eor caso: Que no se encuentre el elemento que buscamos.

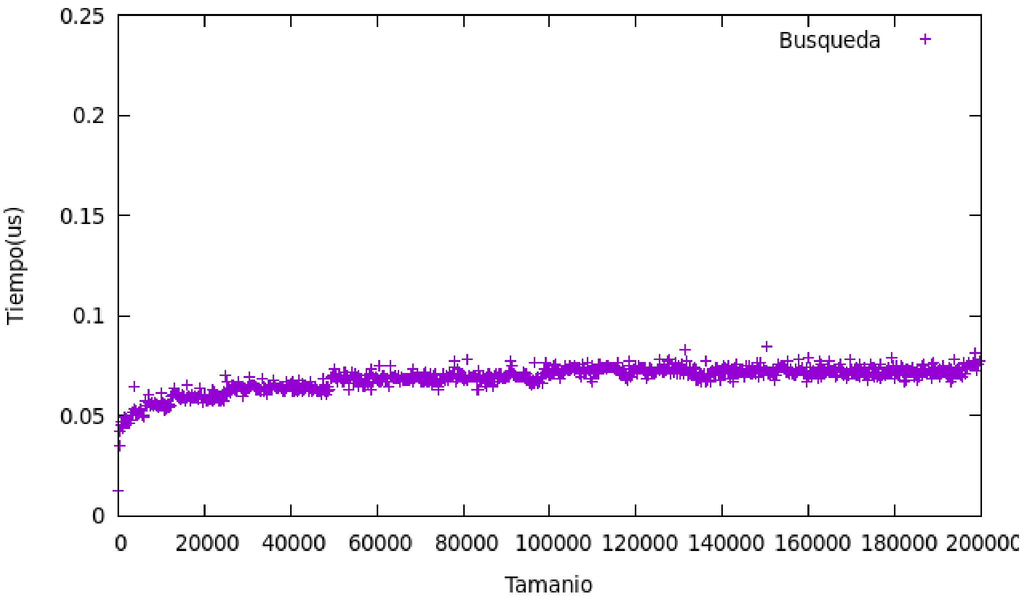
$$T(n) = 6+4+\sum_{ini,fin}^{ini>fin \quad encontrado}(max(4,2)+3+4)+max(2,0)+1 = 10+\sum_{0}^{n/2}(4+7)+3 = 10+11\frac{n}{2}+3$$

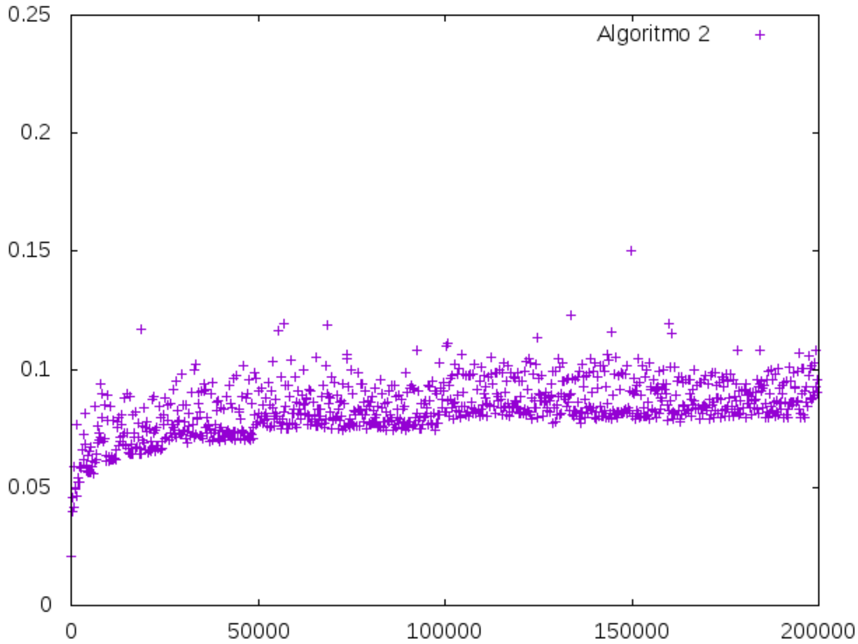
En este algoritmo el caso peor se da cuando se particiona el vector de n elementos por la mitad hasta que no es posible hacerlo más y no se encuentra el elemento, en total log2(n) veces

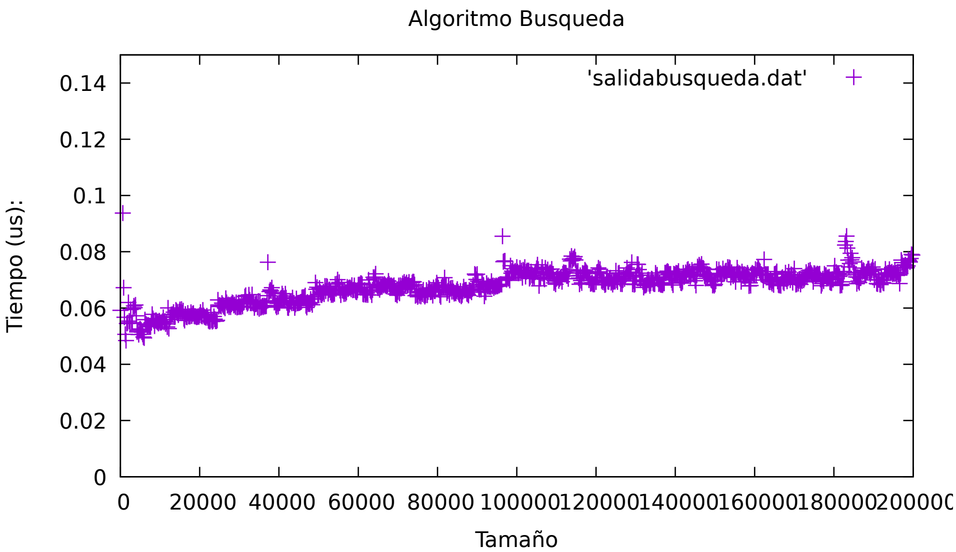
Hasta el while todo son operaciones elementales de orden O(1). Dentro del while encontramos caso de if-else que se solventa con lo de coger el que tenga más operaciones elementales.

El orden de eficiencia es $O(log(n))$

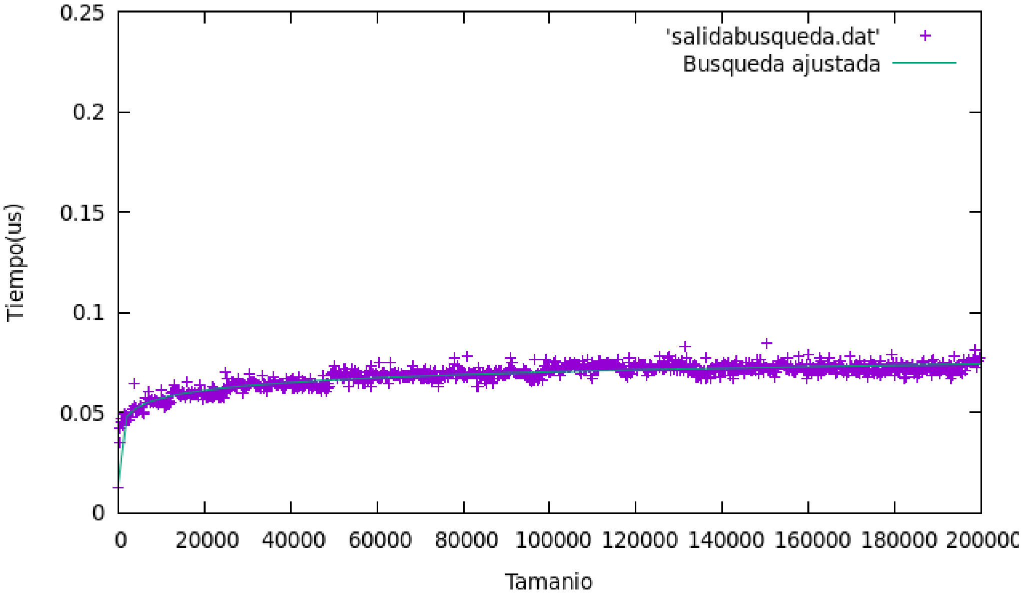
2.2.1. Eficiencia empírica

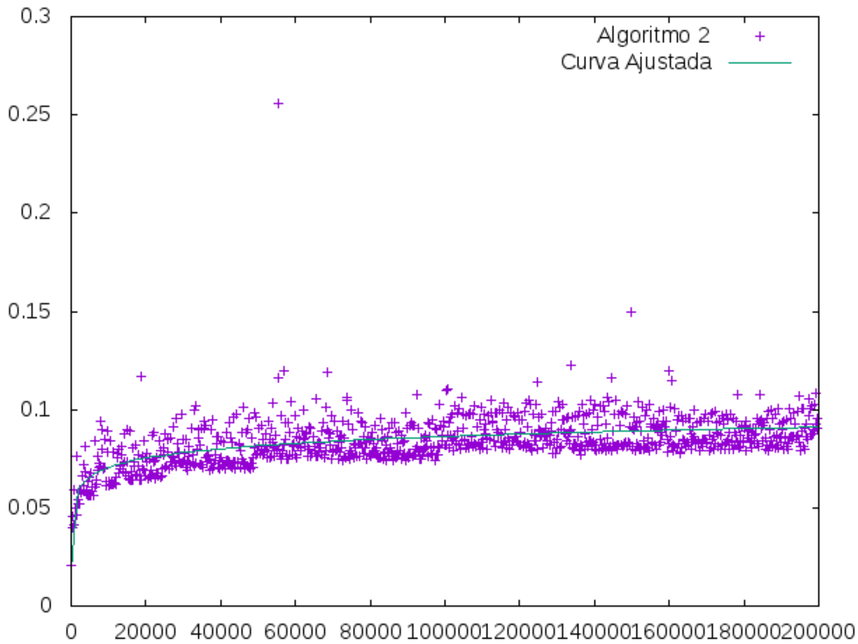


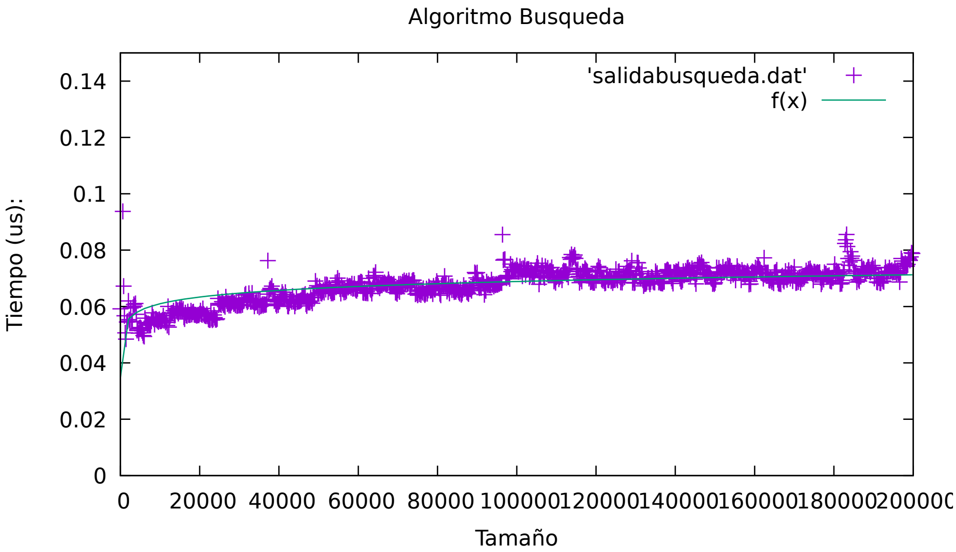

0.25
0.2
0.15
0.1
0.05
0
Algoritmo 2
0
50000
100000
150000
200000


Algoritmo Busqueda
'salidabusqueda.dat'
Tiempo (us):
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
Tamaño

2.2.2. Eficiencia híbrida






Algoritmo Busqueda
'salidabusqueda.dat'
f(x)
Tiempo (us):
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
Tamaño

2.3. Algoritmo 3

2.3.1. Eficiencia teórica

```
void EliminaRepetidos(double original[], int nOriginal) {
  int i, j, k;
  for (i = 0; i < nOriginal; i++) {
    j = i + 1;
    do{
      if(original[j] == original[i]){
        for (k = j + 1; k < nOriginal; k++)
          original[k - 1] = original[k];
        nOriginal--;
      }else
      j++;
    } while (j < nOriginal) ;
  }
}
```

El peor caso se da cuando no hay repetidos.

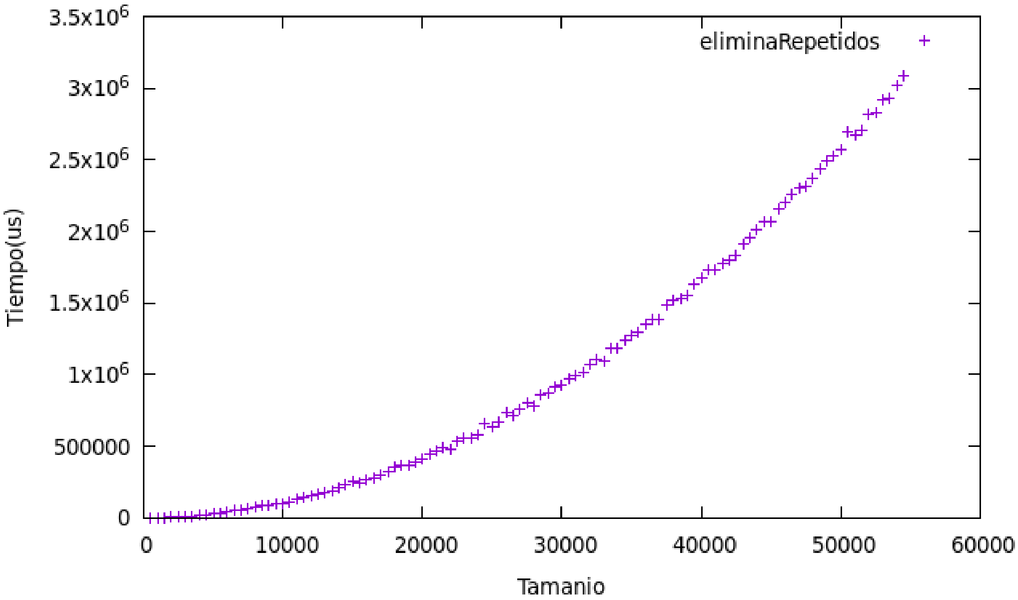
$$T(n) = \sum_{i=0}^{nOriginal} (2 + \sum_{i+1}^{nOriginal} (max((3 + \sum_{i+2}^{nOriginal} 8+1), 1)+1)) = \sum_{i=0}^{n}(2+\sum_{i+1}^{n}(1)+1) =$$

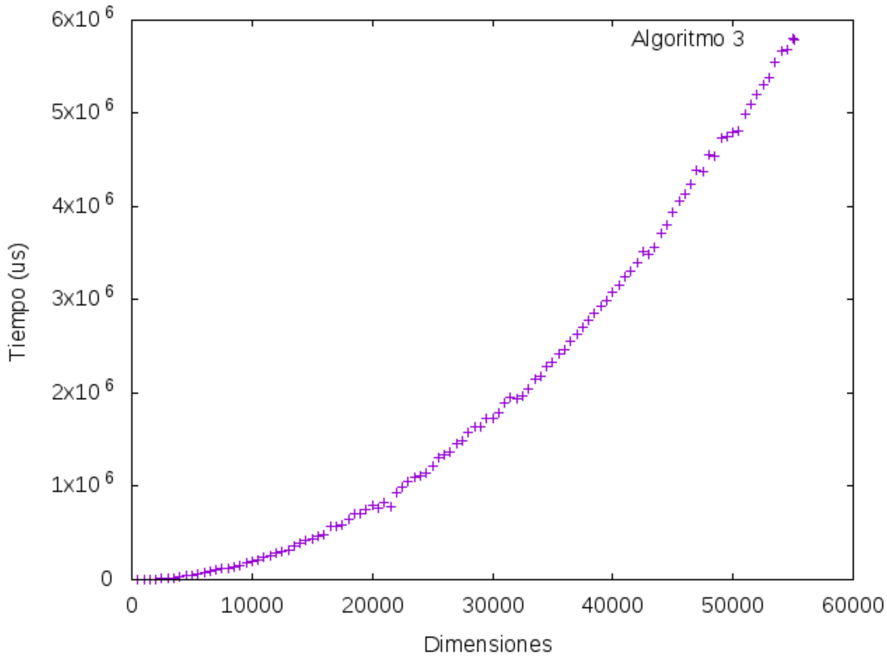
$$T(n) = \sum_{i=0}^{n}(2 + 1(n-1) + 1) = n(3 + (n-1)) = n^2 - n + 3$$

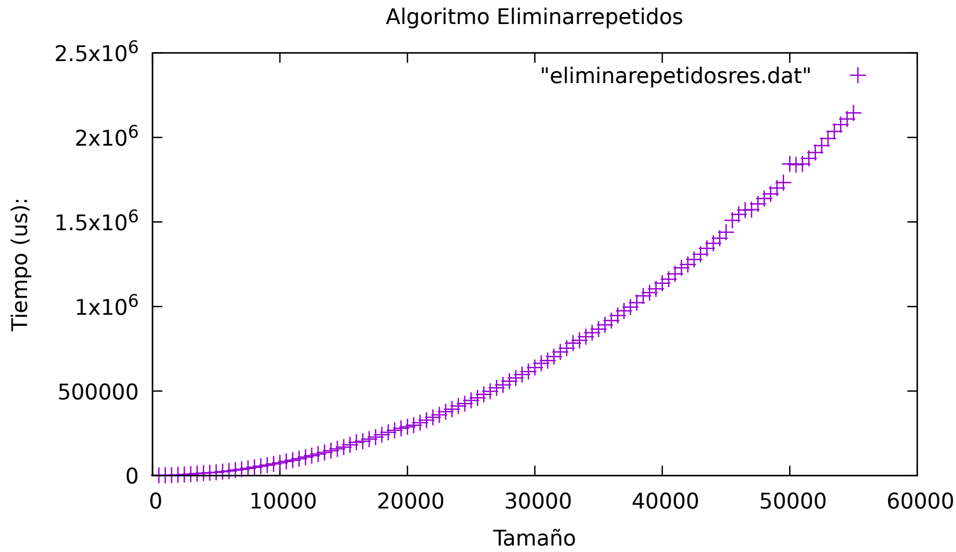
La peculiaridad del peor caso en este algoritmo se encuentra dentro del do-while ya que nunca llega a entrar en el if que contiene la tercera sumatoria al no haber elementos repetidos. Por lo tanto en el max nos quedamos con 1 que es la operacion elemental que se lleva a cabo en el else.

Se nos queda asi un orden de eficiencia cuadratico$O(n^2)$

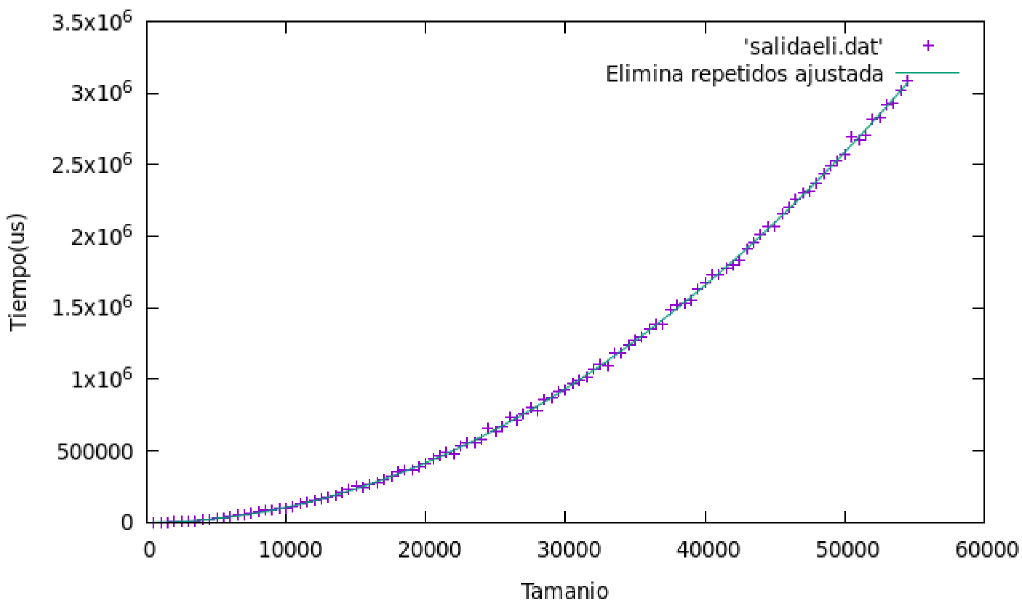
2.3.2. Eficiencia empírica

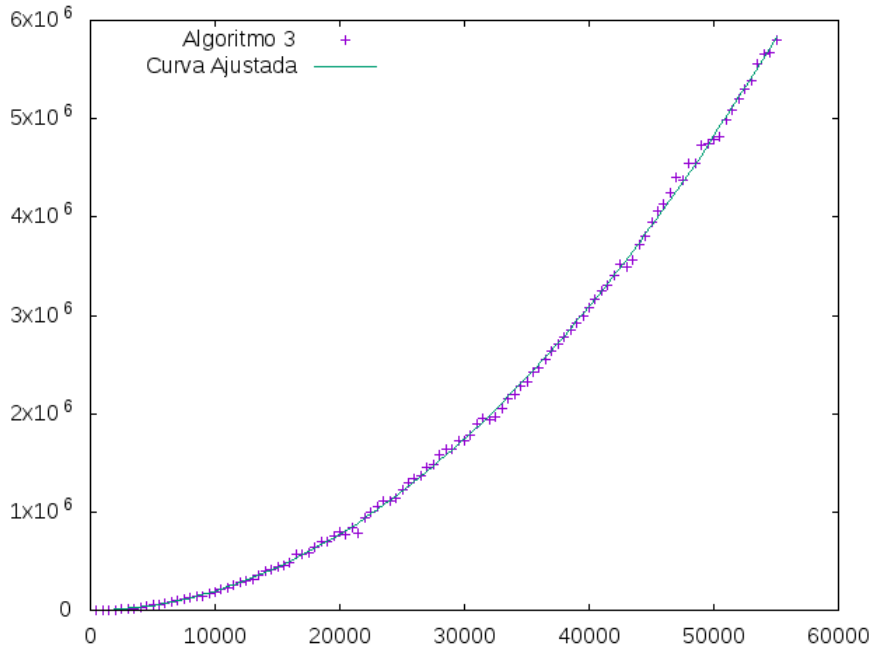




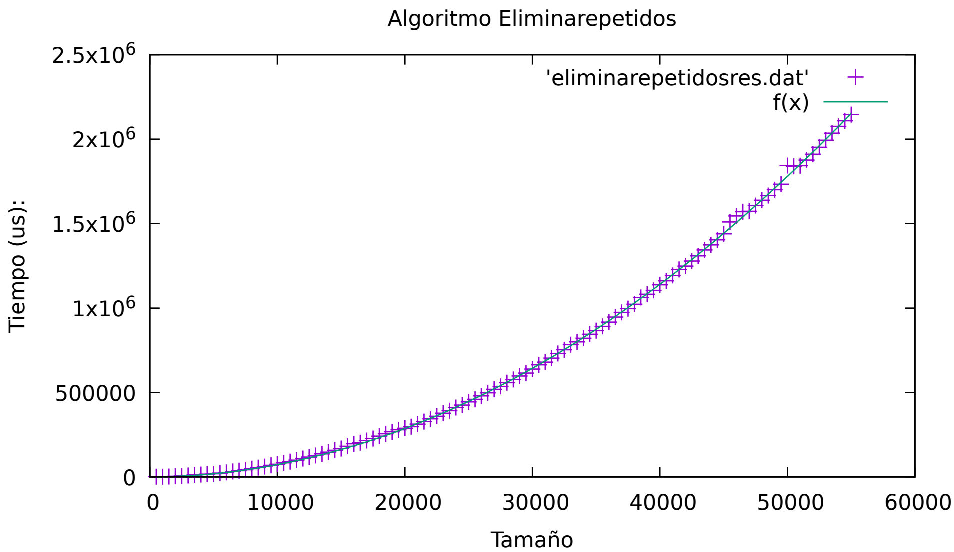


2.3.3. Eficiencia híbrida





Algoritmo 3
Curva Ajustada
6x10 6
5x10 6
4x10 6
3x10 6
2x10 6
1x10 6
0
0
10000
20000
30000
40000
50000
60000



Algoritmo Eliminarepetidos
'eliminarepetidosres.dat'
f(x)
Tiempo (us):
2.5x10^6
2x10^6
1.5x10^6
1x10^6
500000
0
0
10000
20000
30000
40000
50000
60000
Tamaño

2.4. Algoritmo 4

2.4.1. Eficiencia teórica

```
int BuscarBinario(double *v, const int ini, const int fin, const double
  int centro;

  if(ini>fin) return -1;

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

}
```

El peor caso se da cuando el elemento a buscar no se encuentra en el vector, es decir, cuando tras dividir los elementos por analizar nos quedemos con un número menor a 1.

- n <= 1

$$T(n) = 1$$

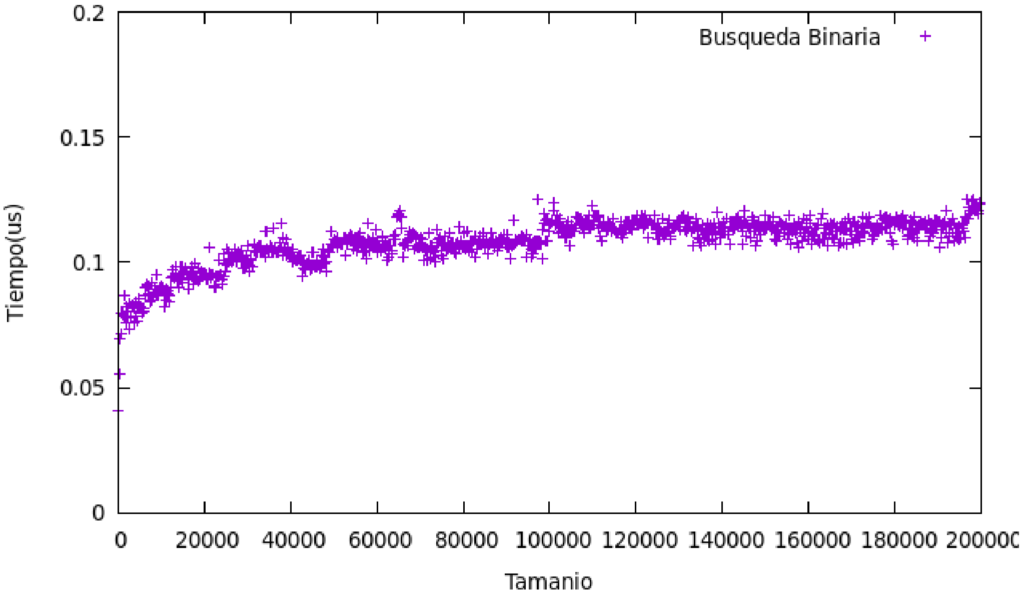
- n >1

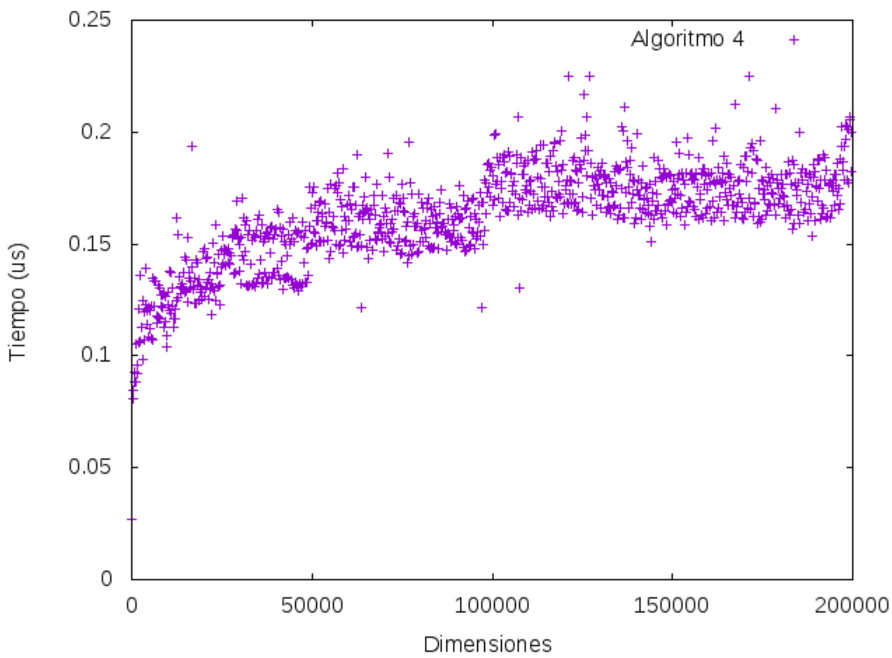
$$T(n) = i * (7 + T(n/2))$$

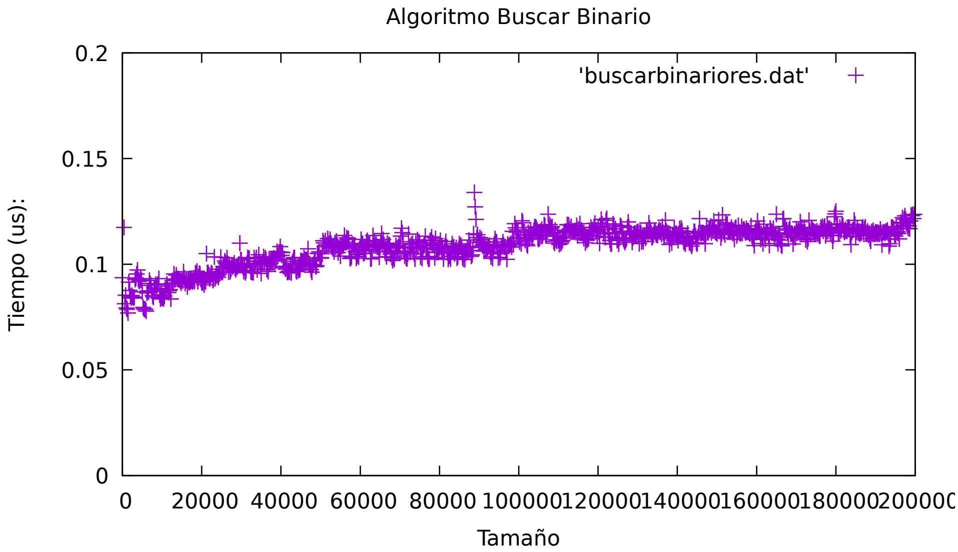
  siendo i el número de iteraciones

Es por esto que el algoritmo de búsqueda binaria tiene una complejidad de orden logarítmico (O(log n)).

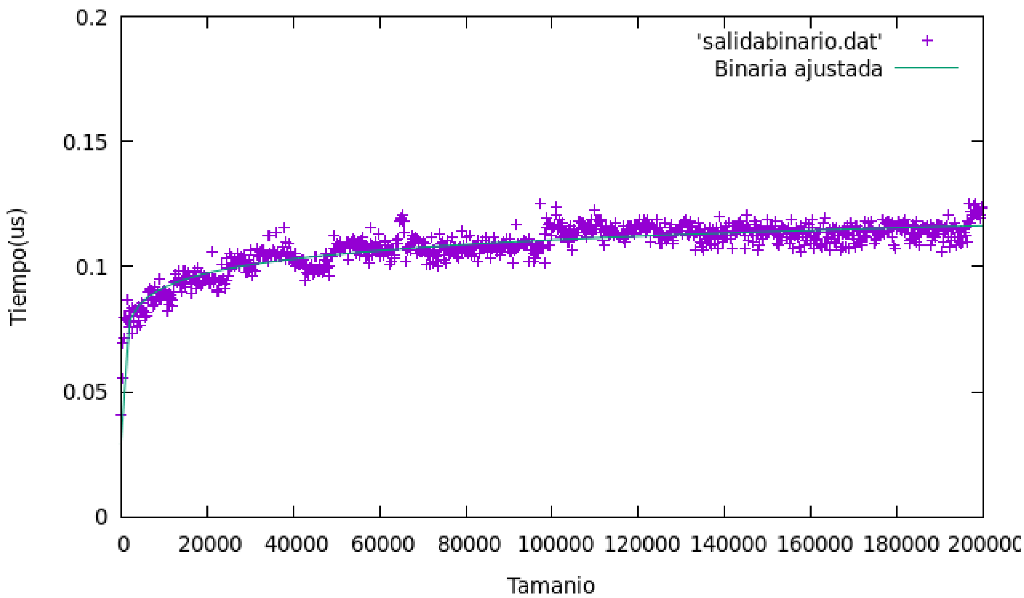
2.4.2. Eficiencia empírica

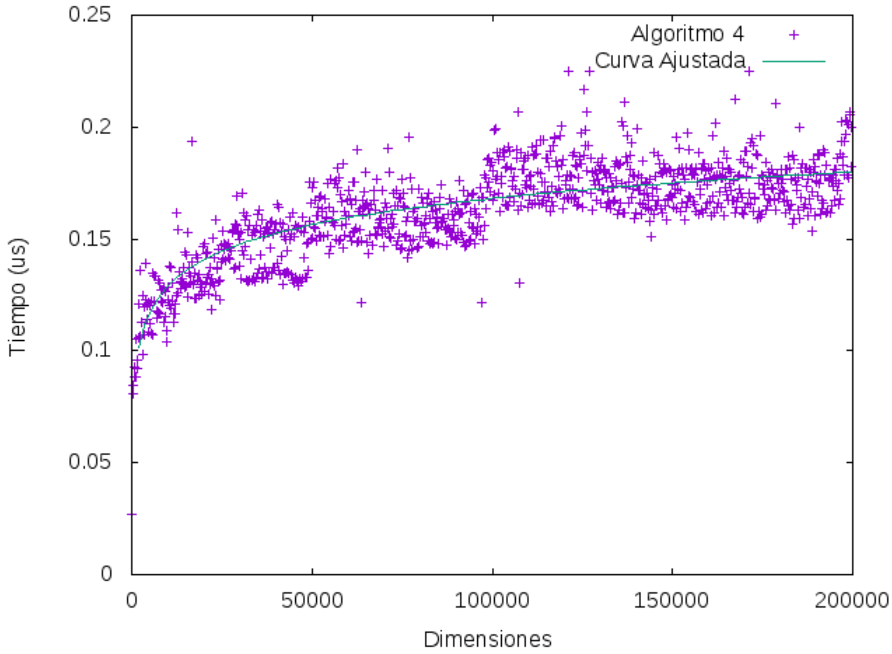


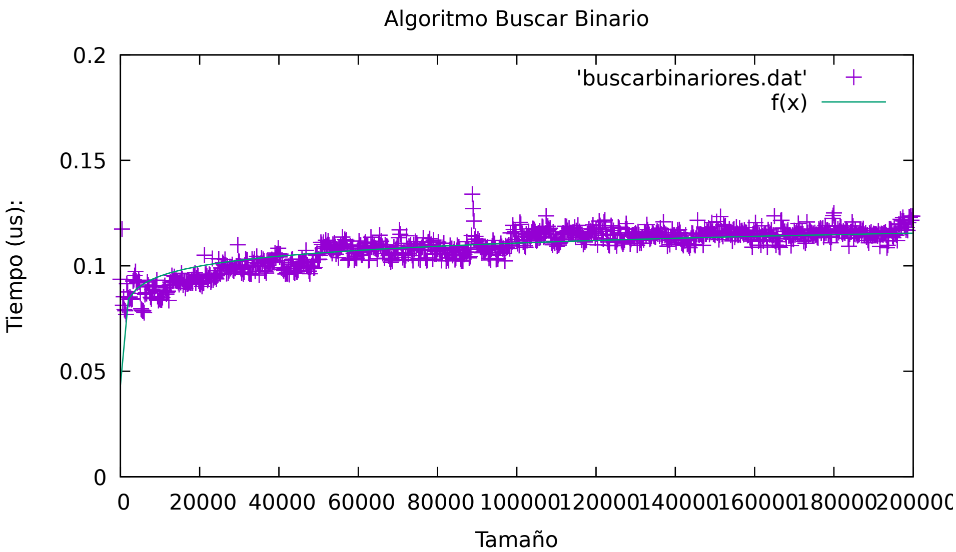




2.4.3. Eficiencia híbrida




Algoritmo 4
Curva Ajustada
Tiempo (us)
Dimensiones
0.25
0.2
0.15
0.1
0.05
0
0
50000
100000
150000
200000


Algoritmo Buscar Binario
'buscarbinariores.dat'
f(x)
Tiempo (us):
Tamaño
0.2
0.15
0.1
0.05
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000

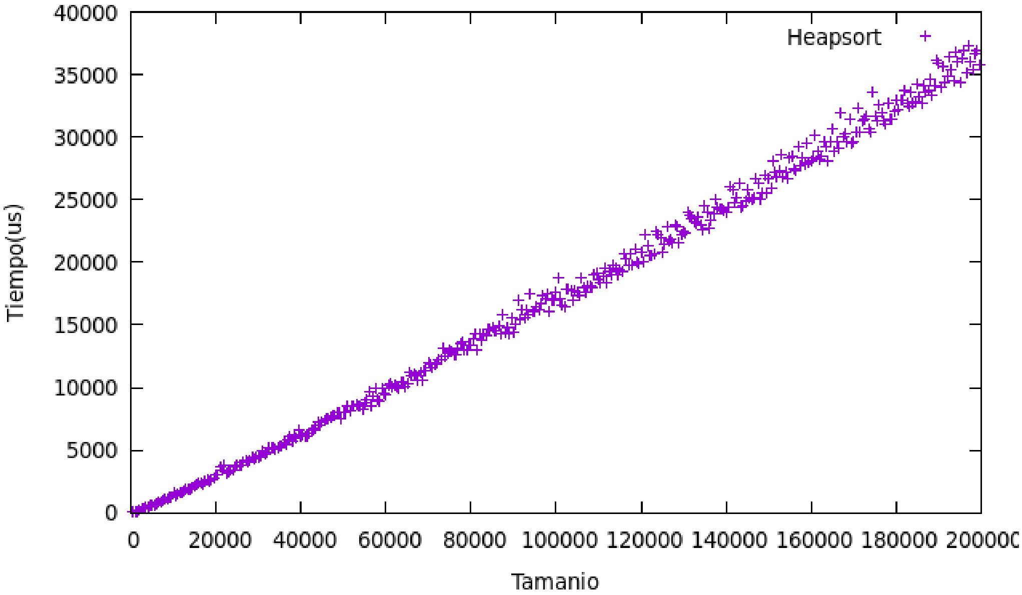
2.5. Algoritmo 5

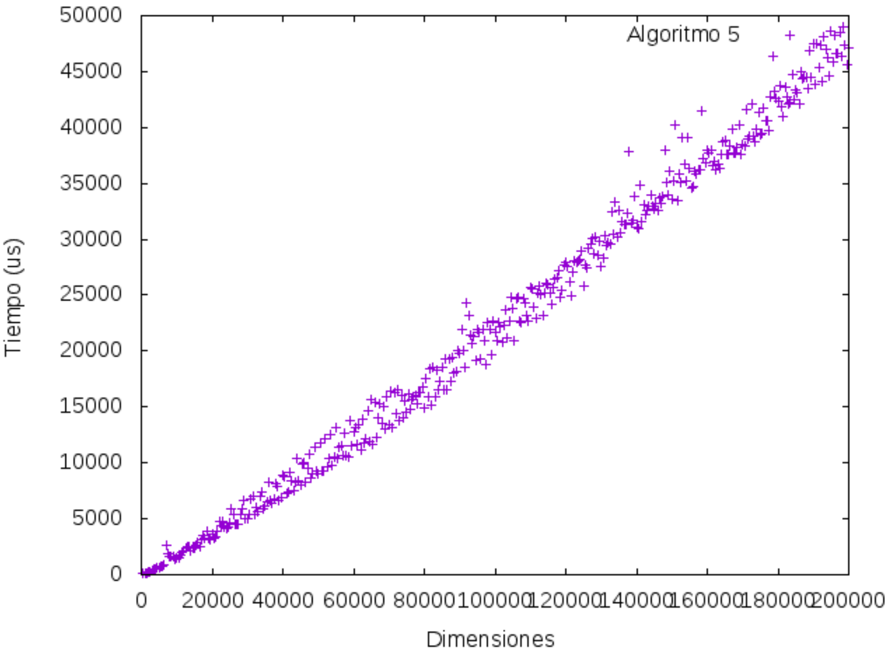
2.5.1. Eficiencia teórica

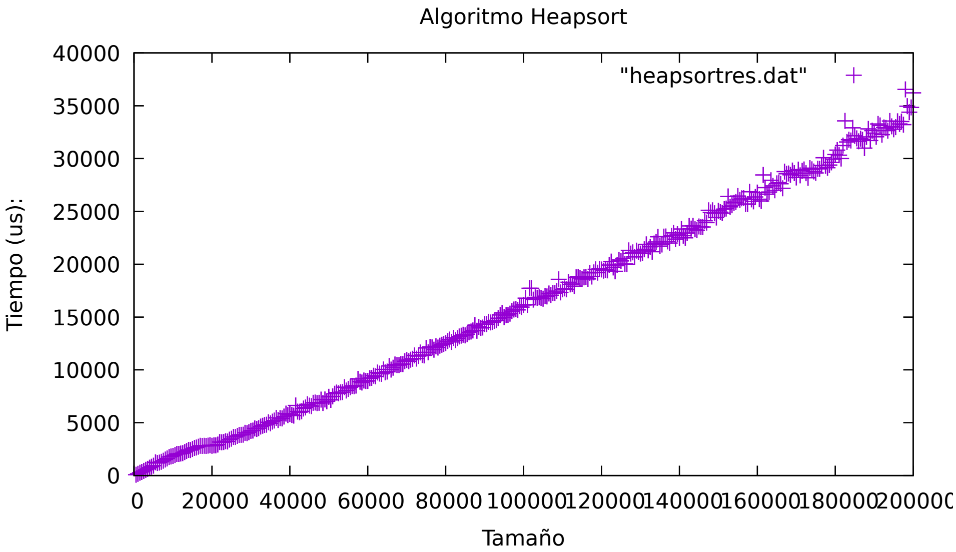
```
void reajustar (int T[], int num_elem , int k){
  int j;
  int v;
  v = T[k];
  bool esAPO = false;
  while ( ( k < num_elem / 2 ) && !esAPO){
    j = k + k + 1;
    if ((j < (num_elem - 1)) && (T[j] < T[j + 1]))
      j++;
    if(v >= T[j])
      esAPO = true;
    T[k] = T[j];
    k = j;
  }
  T[k] = v;
}

void heapsort (int T[] , int num_elem){
  int i;
  for( i = num_elem / 2 ; i >= 0 ; i--)
  reajustar(T, num_elem, i);
  for ( i = num_elem - 1 ; i >= 1 ; i--){
    int aux = T[0];
    T[0] = T[i];
    T[i] = aux;
    reajustar(T, i , 0);
  }
}
```

Orden O(nlogn)

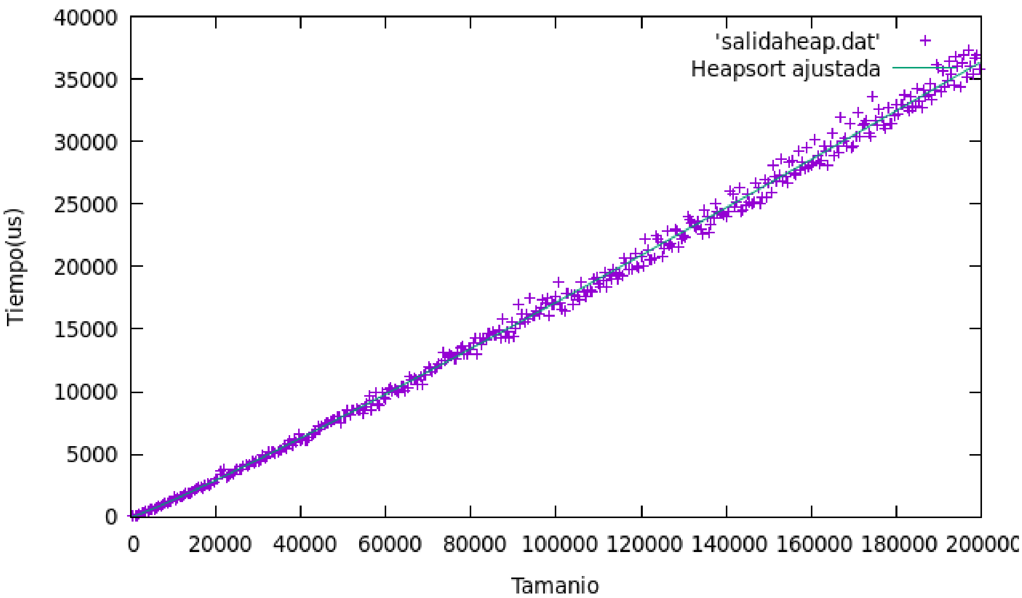
2.5.2. Eficiencia empírica

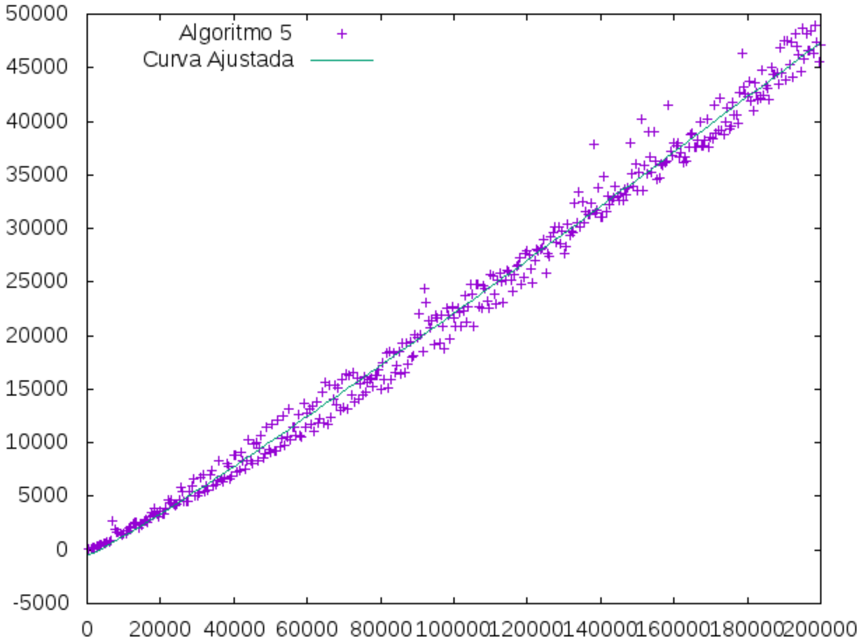




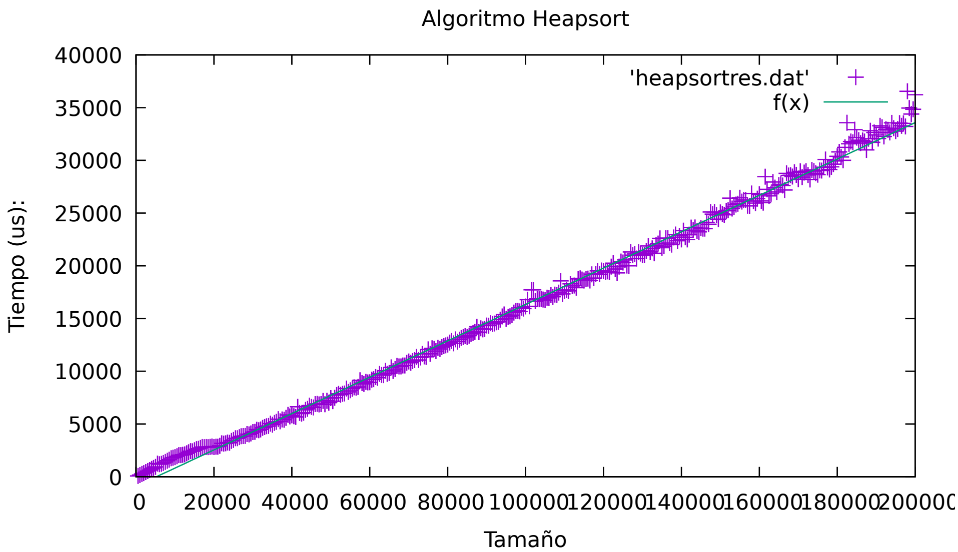


2.5.3. Eficiencia híbrida





Algoritmo 5
Curva Ajustada
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
-5000
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000



Algoritmo Heapsort
'heapsortres.dat'
f(x)
Tiempo (us):
Tamaño
40000
35000
30000
25000
20000
15000
10000
5000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

2.6. Burbuja

2.6.1. Eficiencia teórica

```
void OrdenaBurbuja(int *v, int n) {

        int i, j, aux;
        bool haycambios= true;

        i= 0;
        while (haycambios) {

         haycambios=false; // Suponemos vector ya ordenado
         for (j= n-1; j>i; j--) { // Recorremos vector de final a i

                 if (v[j-1]>v[j]) { // Dos elementos consecutivos mal or
                  aux= v[j]; // Los intercambiamos
                  v[j]= v[j-1];
                  v[j-1]= aux;
                  haycambios= true; // Al intercambiar, hay cambio
                 }
         }
        }
}
```

El peor caso se da cuando el vector esta ordenado a la inversa

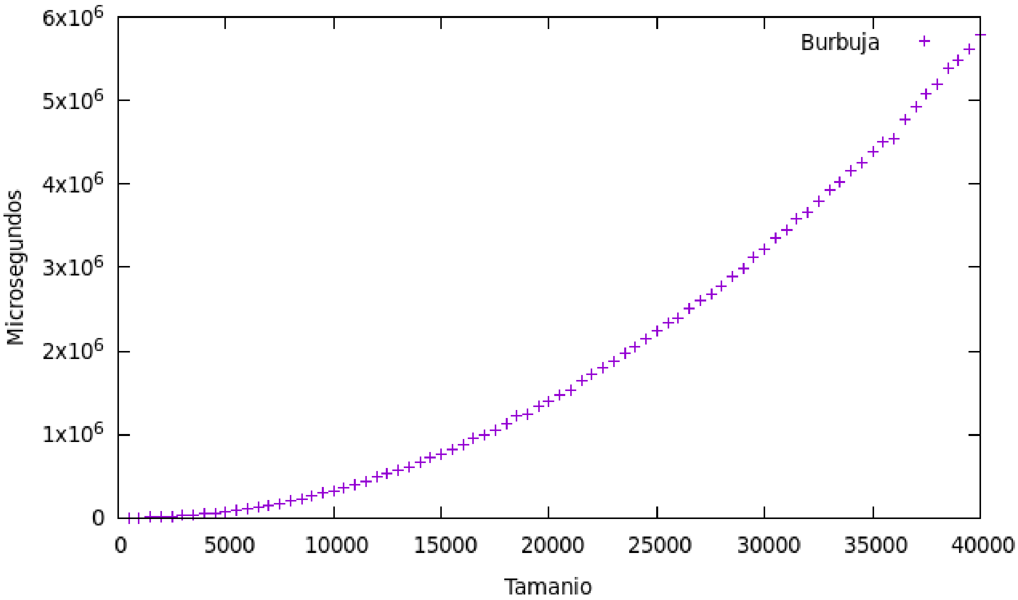
$$T(n) = 2+\sum_{0}^{n/2}(1+\sum_{j=n-1}^{j<=i}(max(13,0)+3)) = 2+\sum_{0}^{n/2}(1+a(n-1)) = 2+\frac{n}{2}*a(n-1)$$

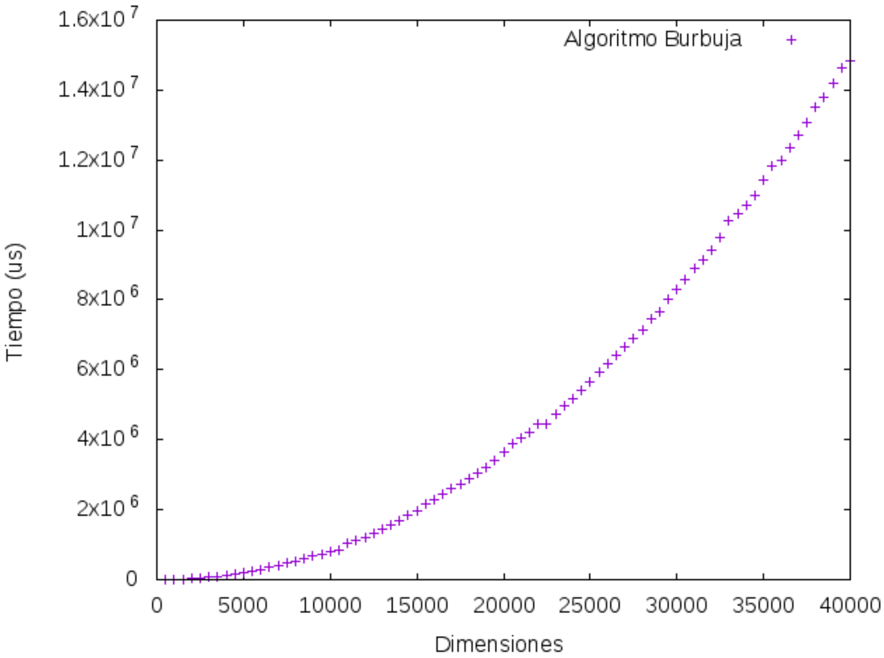
$$T(n) = 2 + a(\frac{n^2-n}{2})$$

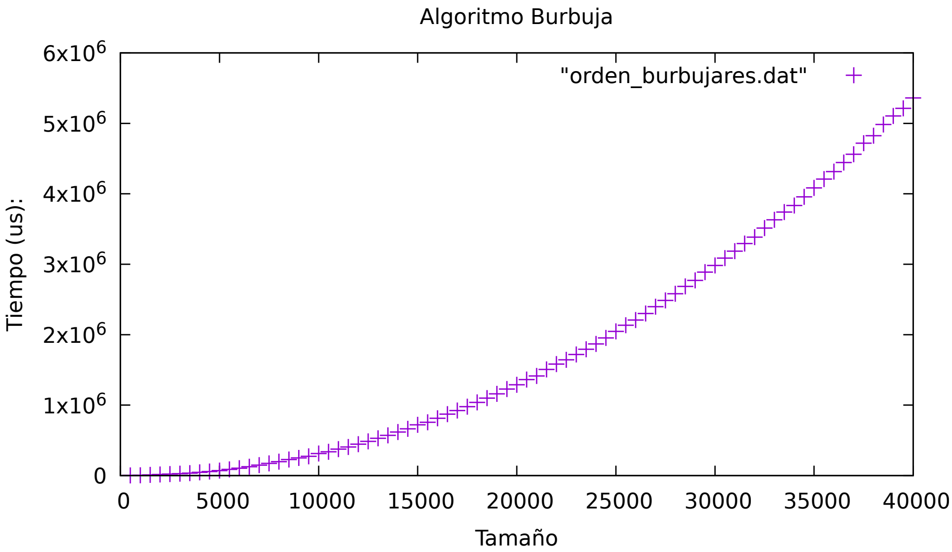
*a es el numero de operaciones elementales que se ejecutan al intercambiar valores

La eficiencia es de (O(n^2))

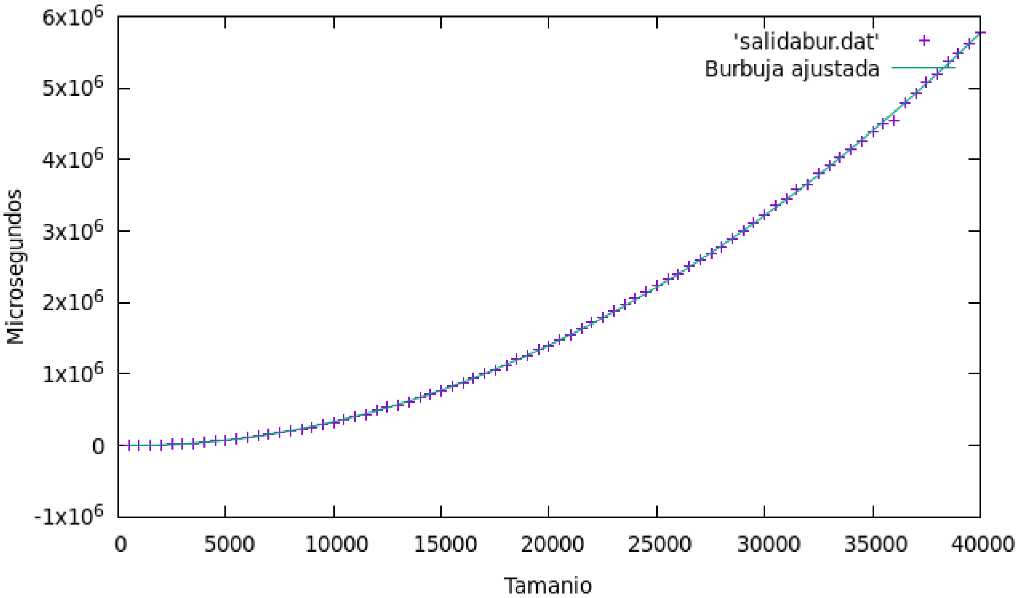
2.6.2. Eficiencia empírica

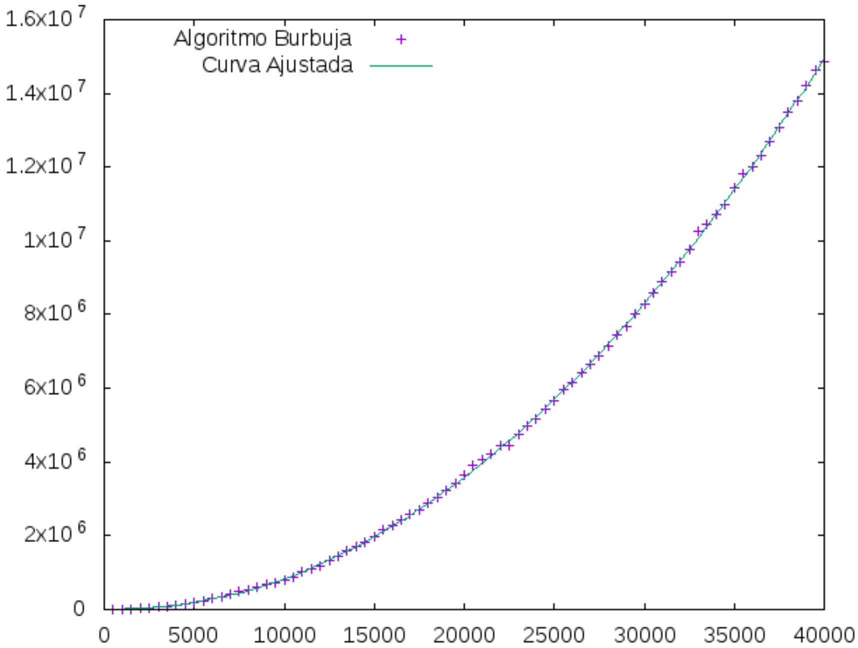


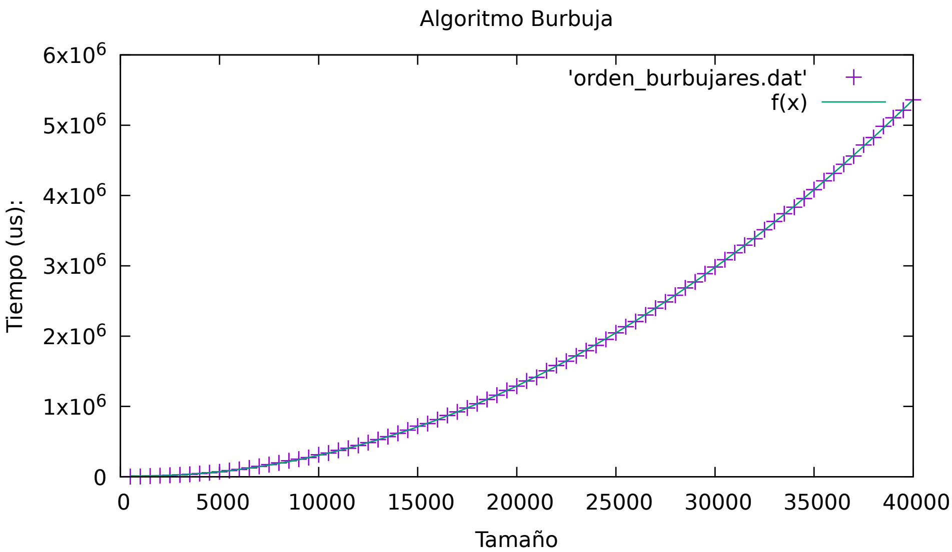




2.6.3. Eficiencia híbrida




Algoritmo Burbuja
Curva Ajustada
1.6x10^7
1.4x10^7
1.2x10^7
1x10^7
8x10^6
6x10^6
4x10^6
2x10^6
0
0
5000
10000
15000
20000
25000
30000
35000
40000


Algoritmo Burbuja
'orden_burbujares.dat'
f(x)
Tiempo (us):
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
0
5000
10000
15000
20000
25000
30000
35000
40000
Tamaño

2.7. Mergesort

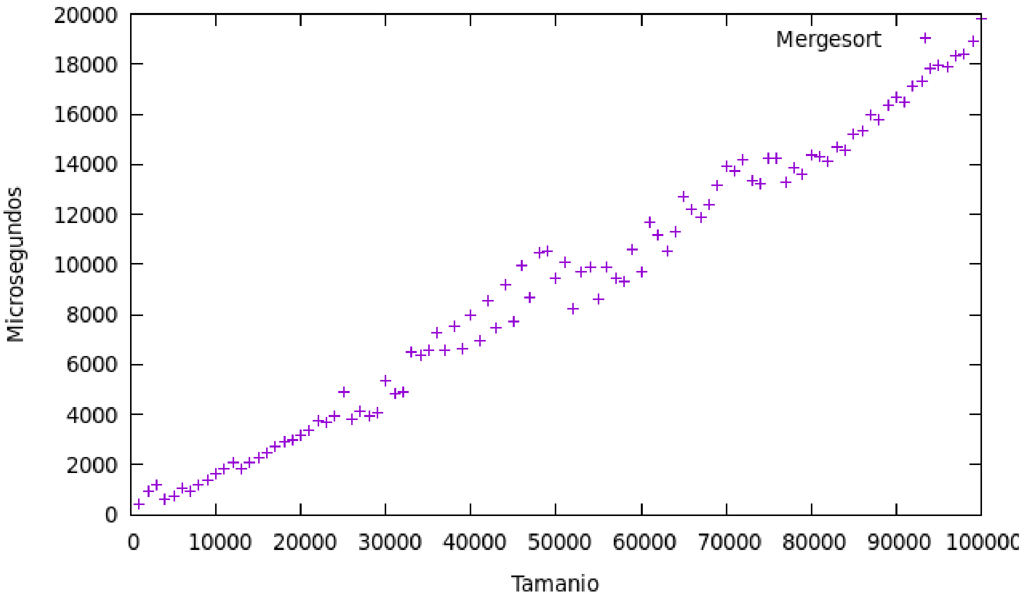
2.7.1. Eficiencia teórica

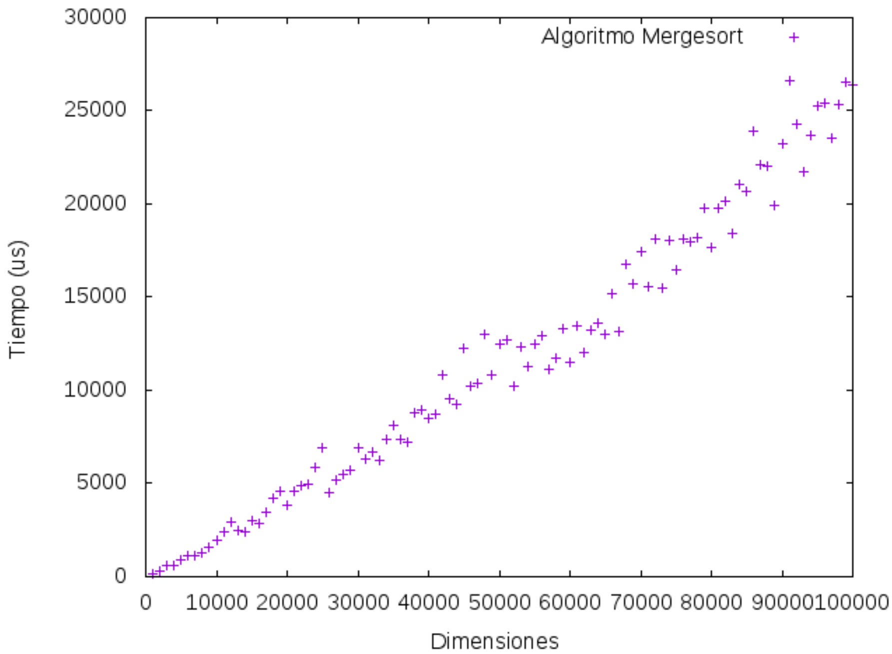
```
inline static void insercion(int T[], int num_elem)
{
  insercion_lims(T, 0, num_elem);
}


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


const int UMBRAL_MS = 100;

void mergesort(int T[], int num_elem)
{
  mergesort_lims(T, 0, num_elem);
}

static void mergesort_lims(int T[], int inicial, int final)
{
  if (final - inicial < UMBRAL_MS)
    {
      insercion_lims(T, inicial, final);
    } else {
      int k = (final - inicial)/2;

      int * U = new int [k - inicial + 1];
      assert(U);
```

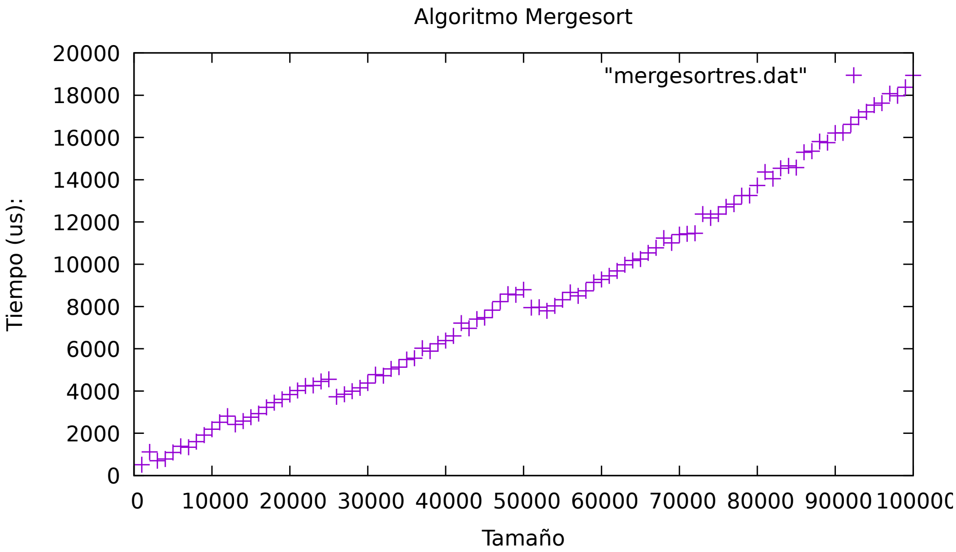
```
      int l, l2;
      for (l = 0, l2 = inicial; l < k; l++, l2++)
        U[l] = T[l2];
      U[l] = INT_MAX;

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

      mergesort_lims(U, 0, k);
      mergesort_lims(V, 0, final - k);
      fusion(T, inicial, final, U, V);
      delete [] U;
      delete [] V;
    };
}


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

Orden O(nlogn)

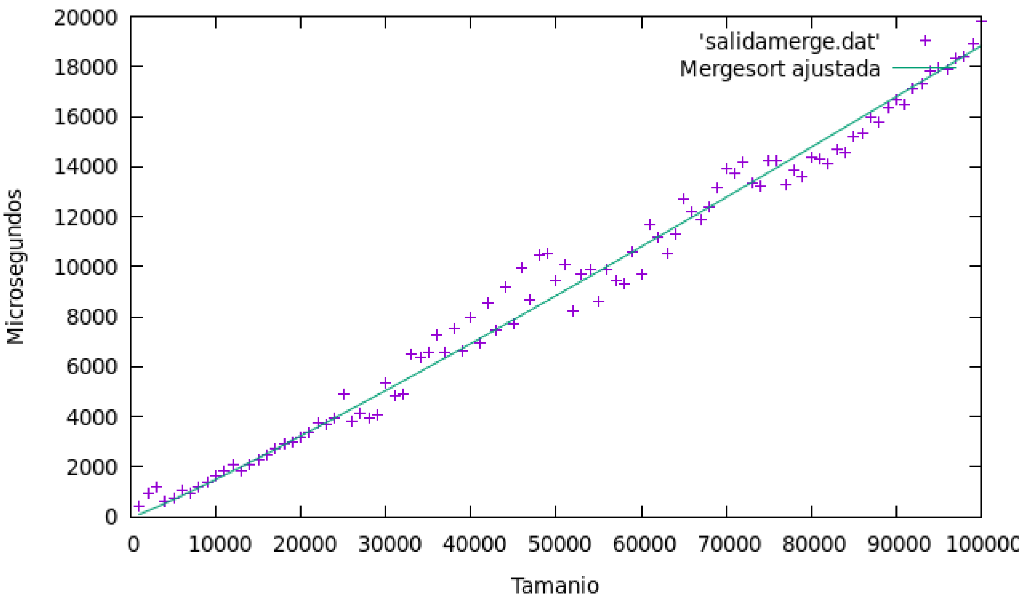
2.7.2. Eficiencia empírica

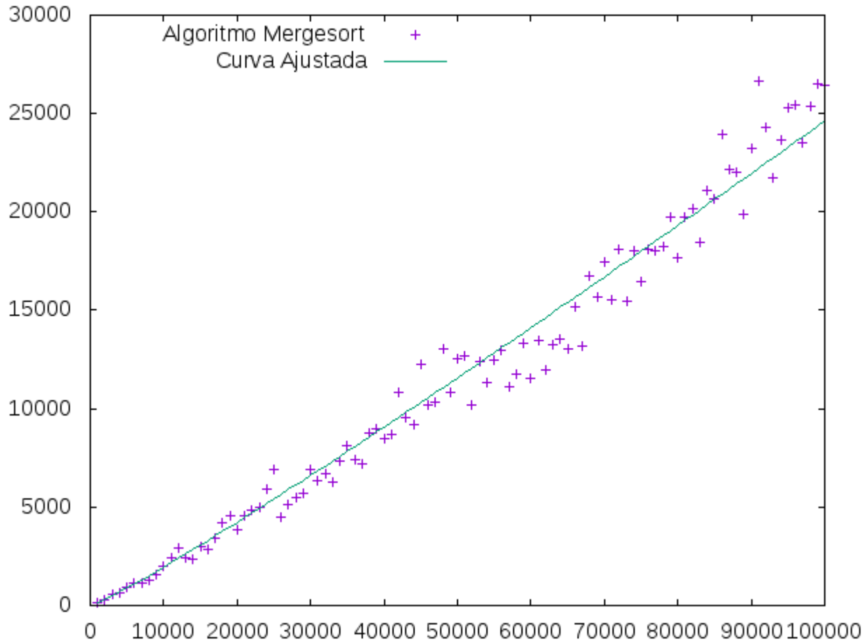




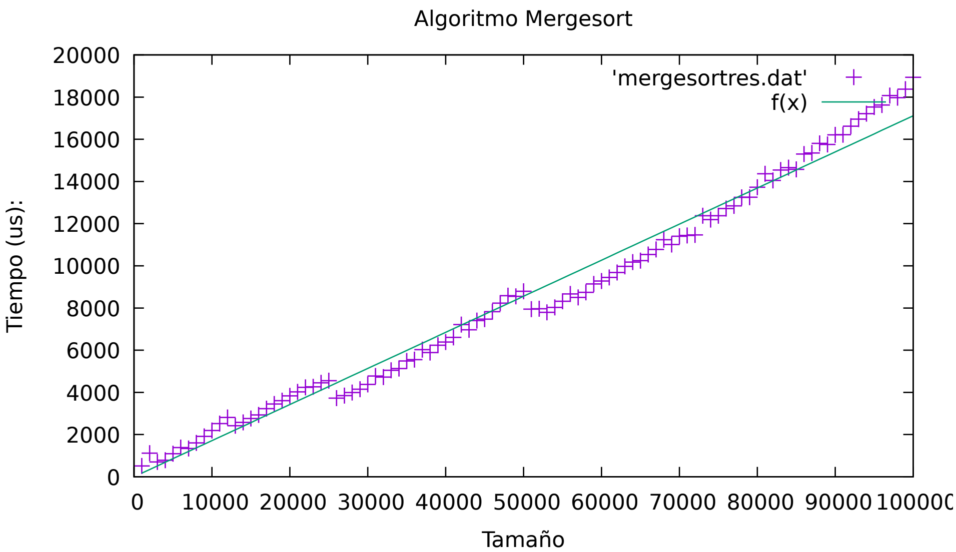


2.7.3. Eficiencia híbrida





Algoritmo Mergesort
Curva Ajustada
30000
25000
20000
15000
10000
5000
0
0 10000 20000 30000 40000 50000 60000 70000 80000 90000 100000



Algoritmo Mergesort
'mergesortres.dat'
f(x)
Tiempo (us):
Tamaño
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0 10000 20000 30000 40000 50000 60000 70000 80000 90000 100000

2.8. Hanoi

2.8.1. Eficiencia teórica

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

Total de movimientos $T(n) = 2T(n-1) + 1$

- n >1 $2T(n-1) + 1$
- $n = 1$ 1

$$T(n) = 2T(n-1) + 1 \qquad n > 1$$

$$= 2T(n-2) + 1 = 2^2T(n-2) + 2 + 1 \qquad n > 2$$

$$T(n) = 2^3T(n-3) + 2^2 + 2 + 1 \qquad n > 3$$

Y para n>i

$$T(n) = 2^iT(n-i) + (2^{i-1} + ... + 2^2 + 2 + 1)$$

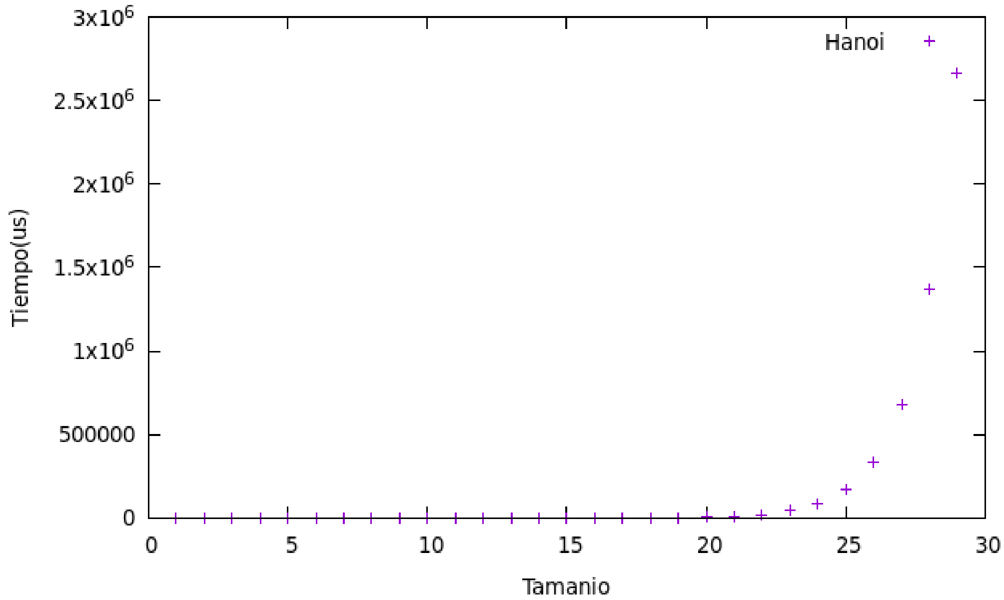
Para i=n-1

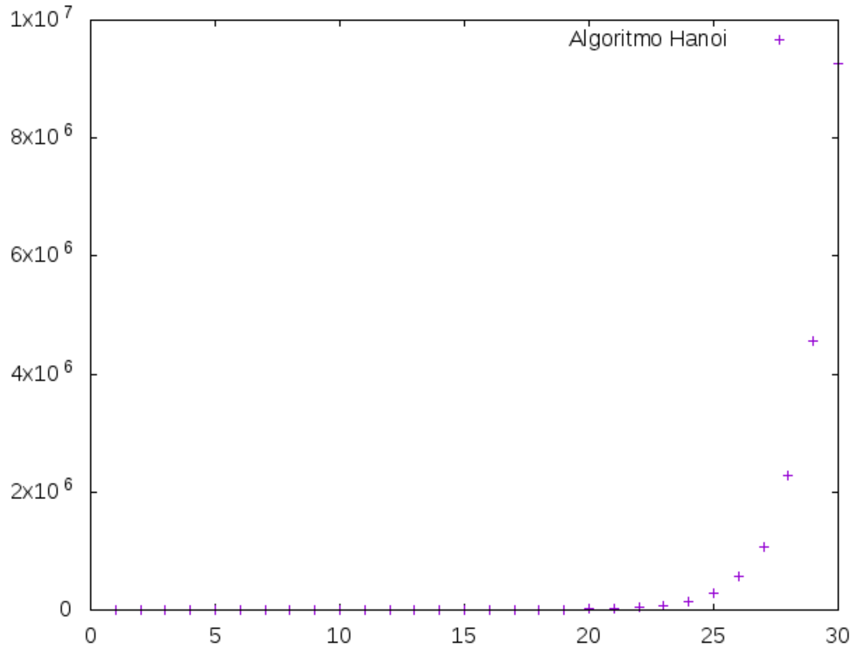
$$T(n) = 2^{n-1}T(1) + (2^{n-2} + ... + 2^2 + 2 + 1)$$

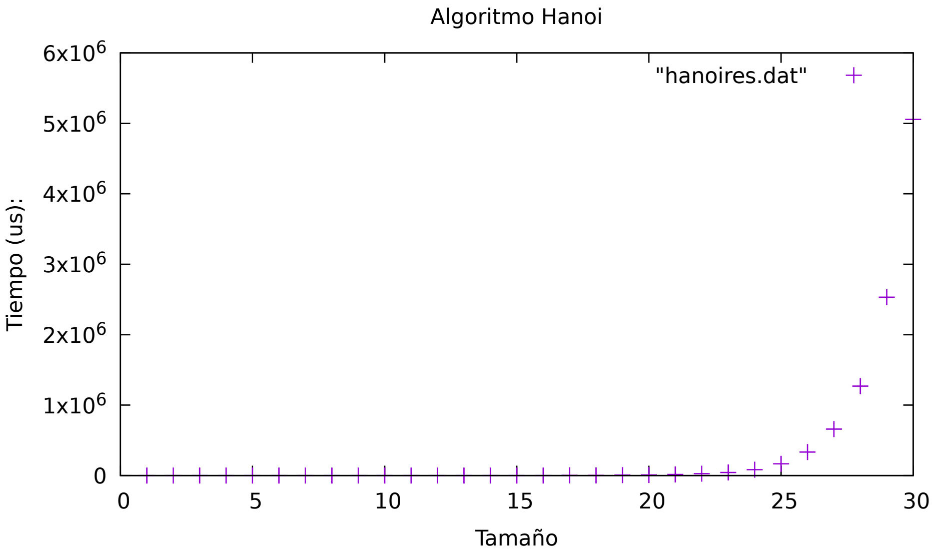
$$T(n) = \frac{2^{n-1} * 2 - 1}{2 - 1} = 2^n - 1$$

El orden de eficiencia es (O(2^n))

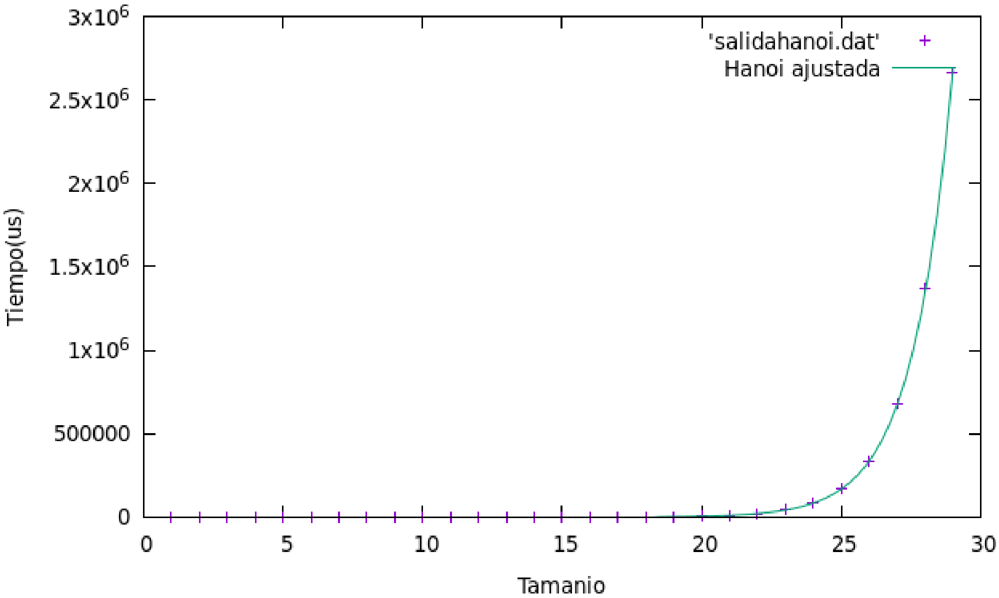
2.8.2. Eficiencia empírica

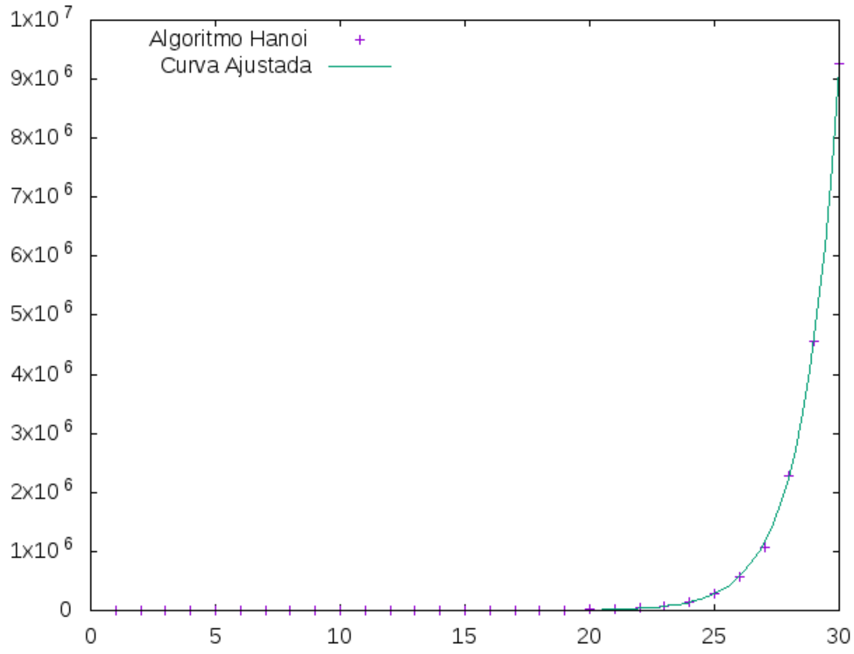


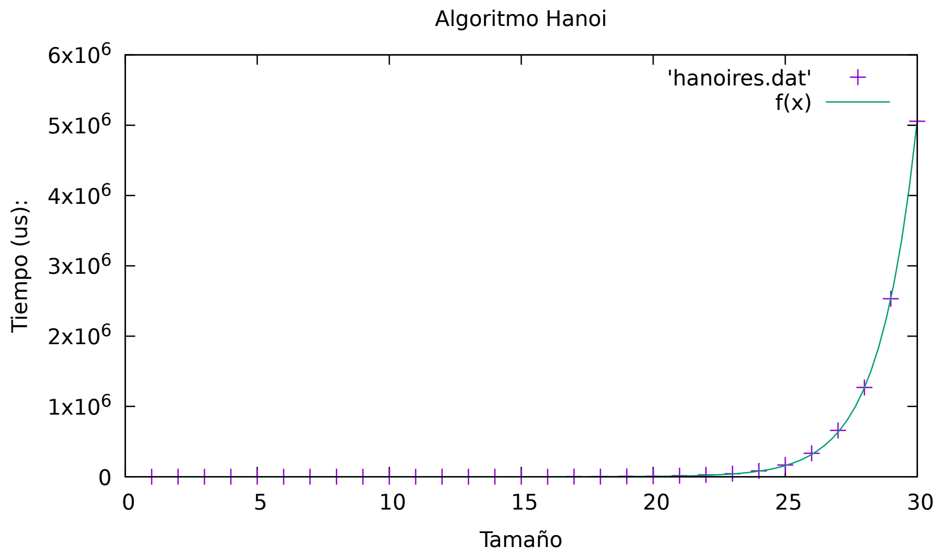




2.8.3. Eficiencia híbrida




Algoritmo Hanoi
Curva Ajustada
1x10 7
9x10 6
8x10 6
7x10 6
6x10 6
5x10 6
4x10 6
3x10 6
2x10 6
1x10 6
0
0
5
10
15
20
25
30


Algoritmo Hanoi
'hanoires.dat'
f(x)
Tiempo (us):
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
0
5
10
15
20
25
30
Tamaño

3. Comparación de constantes K

IRENE

alg1: 0.0065
alg2: 0.02685741
alg3: 0.004957
alg4: 0.04651423
alg5: 0.2245189
burbuja: 0.003512
mergesort: 0.6547
hanoi: 0.0641523

JAVIER

alg1: 0.0058
alg2: 0.016234929
alg3: 0.0023218503
alg4: 0.0323592918
alg5: 0.1091367311
burbuja: 0.0046406056
mergesort: 0.06282032
hanoi: 0.016113281

CESAR

alg1: 0.0051
alg2: 0.02547895
alg3: 0.004884
alg4: 0.04030698
alg5: 0.22781818
burbuja: 0.00236641
mergesort: 0.558471
hanoi: 0.062551

Tras analizar estas constantes, podemos ver que los ordenadores de César y de Irene son ligeramente mejores que el de Javier, tal y como vimos al inicio de la práctica, por ello tienen unas constantes (cotas superiores) mayores.

4. Conclusión

Con esta parte de la práctica 1 hemos llegado a la conclusión de que la eficiencia a la hora de ejecutar un algoritmo depende mucho de la máquina que se esté empleando. También hemos visto que a la hora de compilar.

Por otro lado, mirando ahora nuestros resultados podemos afirmar que:

1. El Principio de invarianza nos dice que los tiempos de ejecucion de un mismo algoritmo solo difieren en cuanto a constantes, cosa que hemos podido comprobar empiricamente y que hemos plasmado en los gráficos.

2. Un código ejecutado en distintas máquinas siempre va a tener resultados de ejecución distintos. En nuestro caso, si un ordenador es mucho más viejo que el otro, va a poseer componentes que se han quedado anticuados y poco a poco se han mejorado, por lo tanto le llevará más tiempo y esfuerzo realizar una tarea que al otro ordenador más actual.