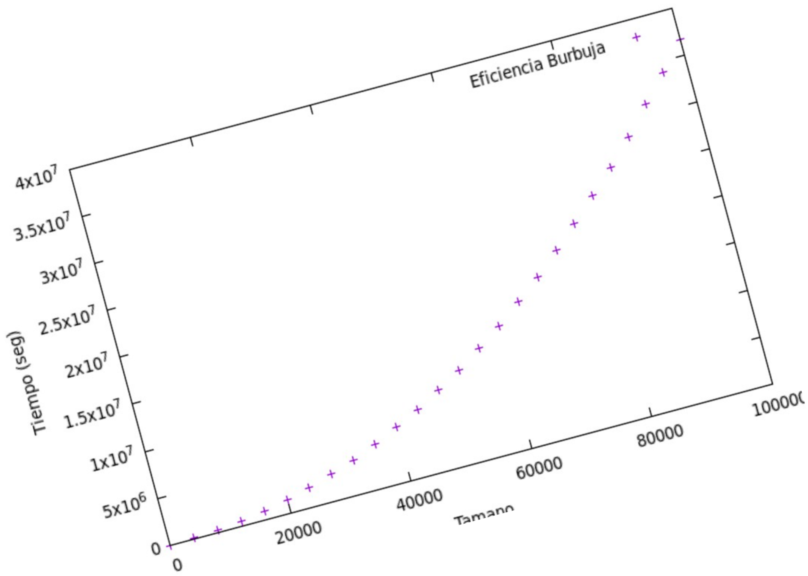
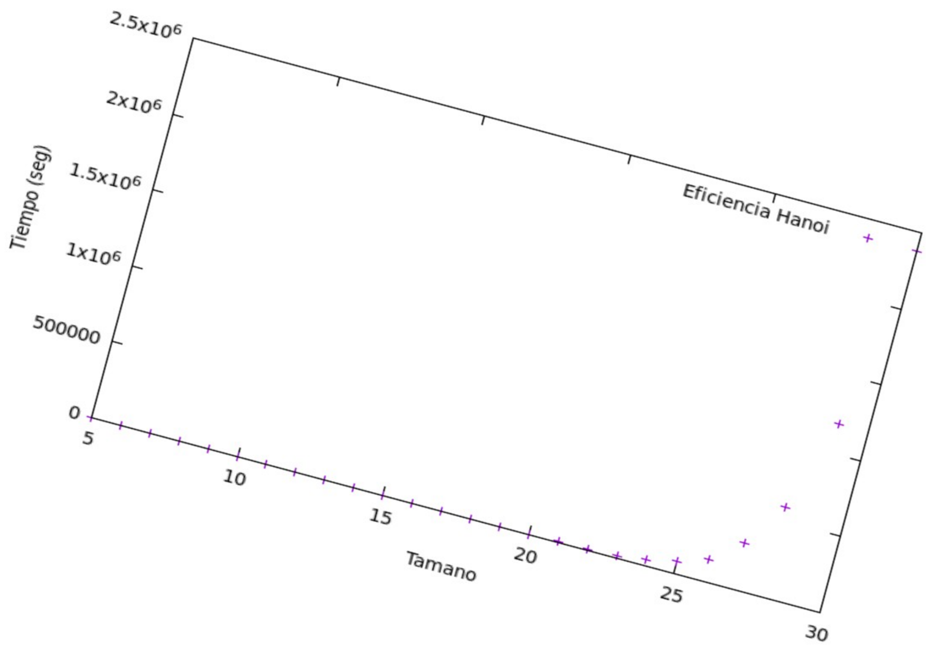
Asignatura: Algorítmica
Nombre: Juan Manuel
Apellidos: Mateos Pérez
Curso: Doble Grado Informática y Matemáticas

PRÁCTICA 1 INDIVIDUAL





LOS VALORES DE MI ORDENADOR SON LOS SIGUIENTES:

```
Arquitectura:                            x86_64
modo(s) de operación de las CPUs:        32-bit, 64-bit
Orden de los bytes:                      Little Endian
CPU(s):                                  8
Lista de la(s) CPU(s) en línea:          0-7
Hilo(s) de procesamiento por núcleo:     2
Núcleo(s) por «socket»:                  4
«Socket(s)»                              1
Modo(s) NUMA:                            1
ID de fabricante:                        GenuineIntel
Familia de CPU:                          6
Modelo:                                  158
Nombre del modelo:                       Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
```

ALGORITMO NÚMERO 1:

```
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;
}
```

Claramente, las dos primeras sentencias tienen una eficiencia de O(1).
Entre los 3 bucles while obtenemos una eficiencia de O(n), ya que los dos interiores van a ir acercando los valores de i y j progresivamente hasta hacerlos iguales.
Los if, que simplemente realizan un intercambio, tienen orden O(1), esto hace que, finalmente diremos que el programa tiene una eficiencia de O(n).

ALGORITMO NÚMERO 2:

Todas las sentencias hasta la linea 7 tienen eficiencia igual a O(1).
El while de la linea 8 tiene una eficiencia de O(log2 n) ya que si nos fijamos divide el puntero v en dos mitades y busca “elem” en esos elementos restantes (que son la mitad de los anteriores).

Por tanto la eficiencia de este algoritmo es O(log n).

```
1  int Busqueda (int *v, int n, int elem) {
2
3    int inicio, fin, centro;
4
5    inicio= 0;
6    fin= n-1;
7    centro= (inicio+fin)/2;
8    while ((inicio<=fin) && (v[centro] != elem)) {
9
10    if (elem<v[centro])
11     fin= centro -1;
12    else
13     inicio= centro+1;
14     centro= (inicio+fin)/2;
15 }
16
17   if (inicio>fin)
18    return -1;
19
20   return centro;
21 }
```

ALGORITMO NÚMERO 3:

```
1
2 void EliminaRepetidos(double original[], int & nOriginal) {
3
4    int i, j, k;
5
6    // Pasamos por cada componente de original
7    for (i= 0; i<nOriginal; i++) {
8
9     // Buscamos valor repetido de original[i]
10    // desde original[i+1] hasta el final
11    j= i+1;
12    do {
13
14      if (original[j] == original[i]) {
15
16        // Desplazamos todas las componentes desde j+1
17        // hasta el final, una componente a la izquierda
18        for (k= j+1; k<nOriginal; k++)
19         original[k-1]= original[k];
20
21        // Como hemos eliminado una componente, reducimos
22        // el numero de componentes utiles
23        nOriginal--;
24      } else // Si el valor no esta repetido, pasamos al siguiente j
25        j++;
26    } while (j<nOriginal);
27
28   } // FIN del primer for
```

Este ejercicio puede parecer que tiene una eficiencia cúbica ya que encontramos 3 bucles for anidados y todos que llegan hasta n, lo que sucede es que encontramos dependencia entre ellos por que varían los límites de los bucles.

Sea n el número de componentes repetidas del vector, en el bucle de las lineas 12-24 se realizan n-j-1 operaciones con eficiencia O(1), esto es, eficiencia O(n). Como *a posteriori* se ejecuta menos de n veces, la eficiencia total es $O(n^2)$.

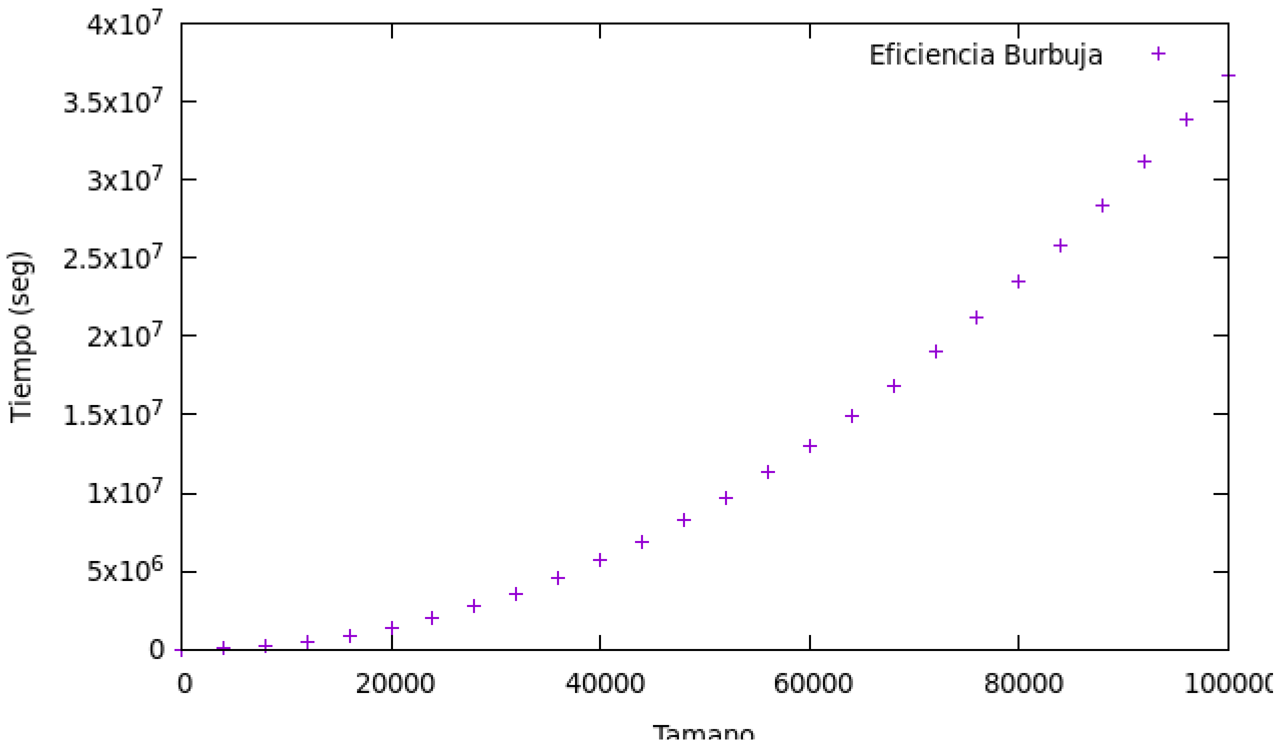
ESTUDIO EMPÍRICO DEL ALGORITMO DE BURBUJA:

Voy a ejecutar 10 veces el programa de orden_burbuja.cpp para comprobar y estudiar su eficiencia empírica. Para ello voy a introducir los mismos valores (columna izquierda) y voy a hacer la media de 25 ejecuciones, obteniendo lo siguiente:

Valor Tiempo
1 0
4000 50403.400000000001
8000 207496.60000000001
12000 485296.70000000001
16000 868404.30000000005
20000 1391952.7

24000 2009767.2
28000 2756549.1000000001
32000 3606939
36000 4603238.7999999998
40000 5726022.0999999996
44000 6938070.0999999996
48000 8294102.2999999998
52000 9722269.5999999996
56000 11298313.300000001
60000 13044836.300000001
64000 14883183.9
68000 16830861.800000001
72000 18977516.600000001
76000 21145872.600000001
80000 23507065.800000001
84000 25865241.100000001
88000 28396701
92000 31144964.300000001
96000 33815353.799999997
100000 36734749.100000001

Y esto nos da los siguientes puntos:

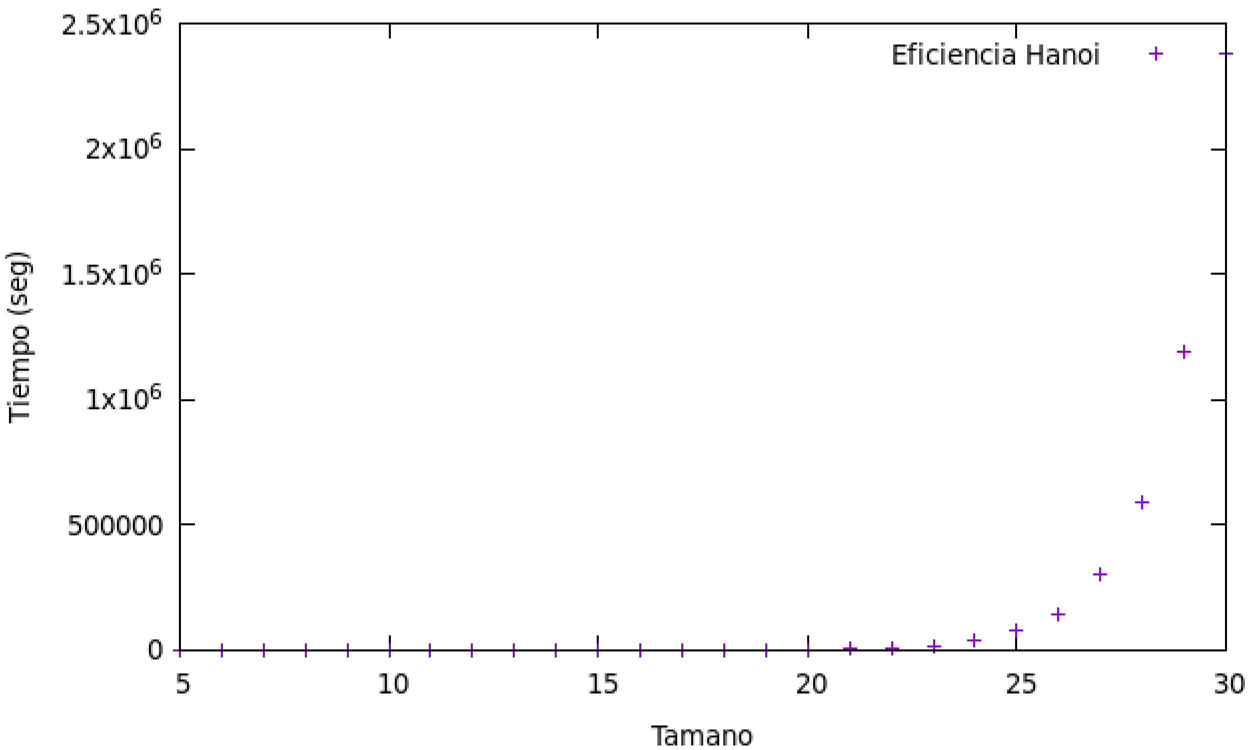


ESTUDIO EMPÍRICO DEL ALGORITMO DE HANOI:

Voy a ejecutar 10 veces el programa de hanoi.cpp para comprobar y estudiar su eficiencia empírica. Para ello voy a introducir los mismos valores (columna izquierda) y voy a hacer la media de 25 ejecuciones, obteniendo lo siguiente:

5 0
6 0
7 0
8 0
9 1
10 2
11 4
12 8
13 17
14 35
15 70
16 140
17 282
18 573
19 1207
20 2476
21 4659
22 9308
23 18398
24 37466
25 75340
26 146305
27 300669
28 587379
29 1186052
30 2382712

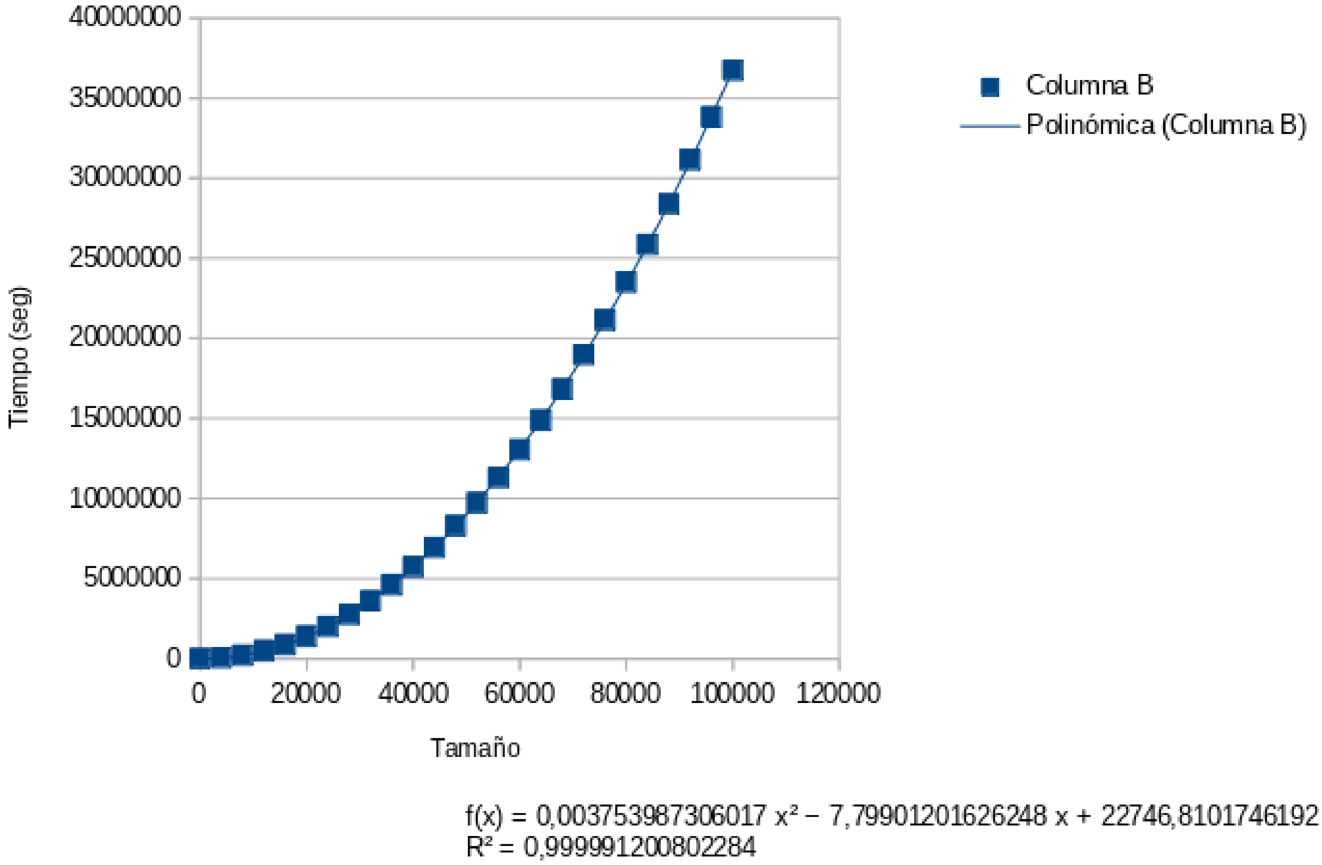
Y esto nos da los siguientes puntos:



ESTUDIO HÍBRIDO DEL ALGORITMO DE BURBUJA:

Del estudio teórico, sabemos que la eficiencia exacta de este algoritmo es la siguiente:

$(a/2)*n^2 – (3a/2)*n + a \in O(n^2)$. Diremos por tanto que el método de ordenación es de orden $O(n^2)$. Comprobemos que obtenemos los mismo resultados con el estudio empírico:



CONCLUSIÓN:

Como podemos ver, hemos obtenido un resultado que a simple vista diríamos que se “asemeja” a una parábola pero gracias al coeficiente de correlación podemos afirmar que es muy semejante a esta. Por tanto, podemos comprobar que los resultado teóricos obtenidos concuerdan con los resultados experimentales e híbridos.