Instituto Politécnico Nacional

Escuela Superior de Cómputo

Práctica 01: Pruebas a posteriori

Gloria Oliva Olivares Ménez
Boleta: 2020630350

Análisis de Algoritmos

Profr. Edgardo Adrián Franco Martínez

3CM15

24 de marzo, 2022

Índice.

Práctica 01: Pruebas a posteriori.

Objetivo.

Realizar un análisis de algoritmos a posteriori de los algoritmos de ordenamiento más conocidos en la computación y realizar una aproximación a sus funciones de complejidad temporal.

Planteamiento del Problema.

Con base en el archivo de entrada proporcionado que tiene hasta 10,000,000 de números diferentes; ordenar bajo los siguientes 10 métodos de ordenamiento y comparar experimentalmente las complejidades aproximadas según se indica en las actividades a reportar.

- Burbuja (Bubble Sort)
  - Burbuja Simple
  - Burbuja Optimizada 1
  - Burbuja Optimizada 2
- Inserción (Insertion Sort)
- Selección (Selection Sort)
- Shell (Shell Sort)
- Ordenamiento con árbol binario de búsqueda (Tree Sort)
- Ordenamiento por Mezcla (Merge Sort)
- Ordenamiento rápido (Quick Sort)
- Ordenamiento por montículos (Heap Sort)

Entorno de pruebas.

Debido a la edad de mi laptop, las pruebas en Linux tuve que realizarlas en otra computadora, cuya ficha técnica es:

- OS: Manjaro Linux x86_64
- CPU: Intel Celeron J1800 (2) @2.582GHz
- GPU: Intel Atom Processor Z36xxx/Z37xxx Series Graphics and Display

Sin embargo, al programar en C y probar los distintos programas, lo hice en la laptop propia:

- Nombre del dispositivo: DESKTOP-SGH89EG
- Procesador: Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz 2.00 GHz
- RAM instalada: 8.00 GB
- Tipo de Sistema: Sistema operativo de 64 bits, procesador x64

Desarrollo.

Pseudocódigos.

Burbuja Simple.

```
Algoritmo BurbujaSimple(A,n)
        para i=0 hasta n-2 hacer
                para j=0 hasta n-2 hacer
                        si (A[j]>A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                        fin si
                fin para
        fin para
fin Algoritmo
```

Burbuja Optimizada 1

```
Algoritmo BurbujaSimple(A,n)
        para i=0 hasta n-2 hacer
                para j=0 hasta (n-2)-i hacer
                        si (A[j]>A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                        fin si
                fin para
        fin para
fin Algoritmo
```

Burbuja Optimizada 2

```
Algoritmo BurbujaOptimizada(A,n)
        cambios = SI
        i=0
        mientras i<= n-2 && cambios != NO hacer
                cambios = NO
                para j=0 hasta (n-2)-i hacer
                        si(A[j] < A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                                cambios = SI
                        fin si
                fin para
                 i= i+1
        fin mientras
fin Algoritmo
```

Inserción (Insertion Sort)

```
Algoritmo Insercion(A,n)
{
       para i=0 hasta n-1 hacer
              j=i
              temp=A[i]
              mientras(j>0) && (temp<A[j-1]) hacer
                     A[j]=A[j-1]
                     j--
              fin mientras
              A[j]=temp
       fin para
fin Algoritmo
```

Selección (Selection Sort)

```
Algoritmo Seleccion(A,n)
       para k=0 hasta n-2 hacer
              p=k
              para i=k+1 hasta n-1 hacer
                     si A[i]<A[p] entonces
                            p=i
                     fin si
              fin para
              temp = A[p]
              A[p] = A[k]
              A[k] = temp
       fin para
fin Algoritmo
```

Shell (Shell Sort)

```
Algoritmo Shell(A,n)
       k = TRUNC(n/2)
       mientras k >=1 hacer
              b=1
              mientras b!=0 hacer
                     b=0
                     para i=k hasta n-1 hacer
                            si A[i-k]>A[i] entonces
                                   temp=A[i]
                                   A[i]=A[i-k]
                                   A[i-k]=temp
                                   b=b+1
                            fin si
                     fin para
              fin mientras
              k=TRUNC(k/2)
       fin mientras
fin Algoritmo
```

Ordenamiento con árbol binario de búsqueda (Tree Sort)

```
Algoritmo OrdenaConABB(A,n)

        para i=0 hasta n-1 hacer
                Insertar(ABB,A[i]);
        fin para

        GuardarRecorridoInOrden(ABB,A);

fin Algoritmo
```

Ordenamiento por Mezcla (Merge Sort)

```
Algoritmo MergeSort(A, p, r)
    si p < r entonces
        q = parteEntera((p+r)/2)
        MergeSort(A, p, q)
        MergeSort(A, q+1,r)
        Merge(A, p, q, r)
    fin si
fin Algoritmo
```

```
Algoritmo Merge(A, p, q, r)
    l=r-p+1, i=p, j=q+1
    para k=0 hasta l hacer
        si i<=q y j<=r entonces
            si A[i]<A[j] entonces
                C[k]=A[i]
                i++
            sino entonces
                C[k]=A[j]
                j++;
            fin si
        sino si i<=q entonces
            C[k]=A[i]
            i++
        sino entonces
            C[k]=A[j]
            j++
        fin si
    A[p-r]=C[]
fin Algoritmo
```

Ordenamiento rápido (Quick Sort)

```
Algoritmo QuickSort(A, p, r)
    si p < r entonces
        j = Pivot(A,p,r)
        QuickSort(A, p, j-1)
        QuickSort(A, j+1,r)
    fin si
fin Algoritmo
```

```
Algoritmo Intercambiar(A, i, j)
   temp= A[j]
   A[j]=A[i]
   A[i]=temp
fin Algoritmo
```

```
Algoritmo Pivot(A, p, r)
   piv=A[p], i=p+1, j=r
   mientras (i<j)
        mientras A[i]<= piv y i<r hacer
            i++
        mientras A[j]> piv hacer
            j--
        Intercambiar(A,i,j)
   fin mientras
   Intercambiar(A,p,j)
   regresar j
fin Algoritmo
```

Ordenamiento por montículos (Heap Sort)

```
Algoritmo HeapSort(A,n)

        para i=0 hasta n-1 hacer
                Insertar(Heap,A[i]);
        fin para

        para i=0 hasta n-1 hacer
                A[i]=Extraer(Heap);
        fin para

fin Algoritmo
```

Tabla comparativa (500,000 números)

Para esta sección se tomó el archivo de texto de 10 millones de números como entrada, solamente que, al momento de ejecutar, se solicitó que fueran 500,000 números.

Se midió el tiempo que cada algoritmo tardaba al ordenar dicha cantidad de números.

Algoritmo	Tiempo real	Tiempo CPU	Tiempo E/S	%CPU/Wall
Burbuja Simple	1110.4260130000s	1099.8872740000s	0.2467970000s	99.0731537400%
Burbuja Optimizada 1	1275.2776398659s	1275.1701260000s	0.0319990000s	99.9940785549%
Burbuja Optimizada 2	1037.585490000s	1035.0417380000s	0.2234640000s	99.0754524200%
Inserción (Insertion Sort)	159.70102810 86 s	159.698167000 0s	0.0000000000 s	99.9982084595 %
Selección (Selection Sort)	308.1264090538s	291.4265880000s	3.1297610000s	95.5959438545%
Shell (Shell Sort)	1.0166339874s	1.0166100000s	0.000s	99.9976405051%
Ordenamiento con árbol binario de búsqueda (Tree Sort)	0.802300930000s	0.57417200000s	0.0231380000s	77.3425431745%
Ordenamiento por Mezcla (Merge Sort)				
Ordenamiento rápido (Quick Sort)	0.5719251633 s	0.3554800000 s	0.0132420000 s	64.4703229864 %

Ordenamiento por montículos (Heap Sort)	0.2936809063s	0.2936140000s	0.0000600000s	99.9976483675%

Tabla 1. Comparación de tiempos entre algoritmos.

Comportamiento temporal de cada algoritmo.

Aquí se realizará un análisis para observar la relación entre el comportamiento del algoritmo y el tamaño del problema, denotado por n.

A continuación, se enlistan los diferentes valores que tendrá n:

n = [100, 1000, 5000, 10000, 50000, 100000, 200000, 400000, 600000, 800000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000].

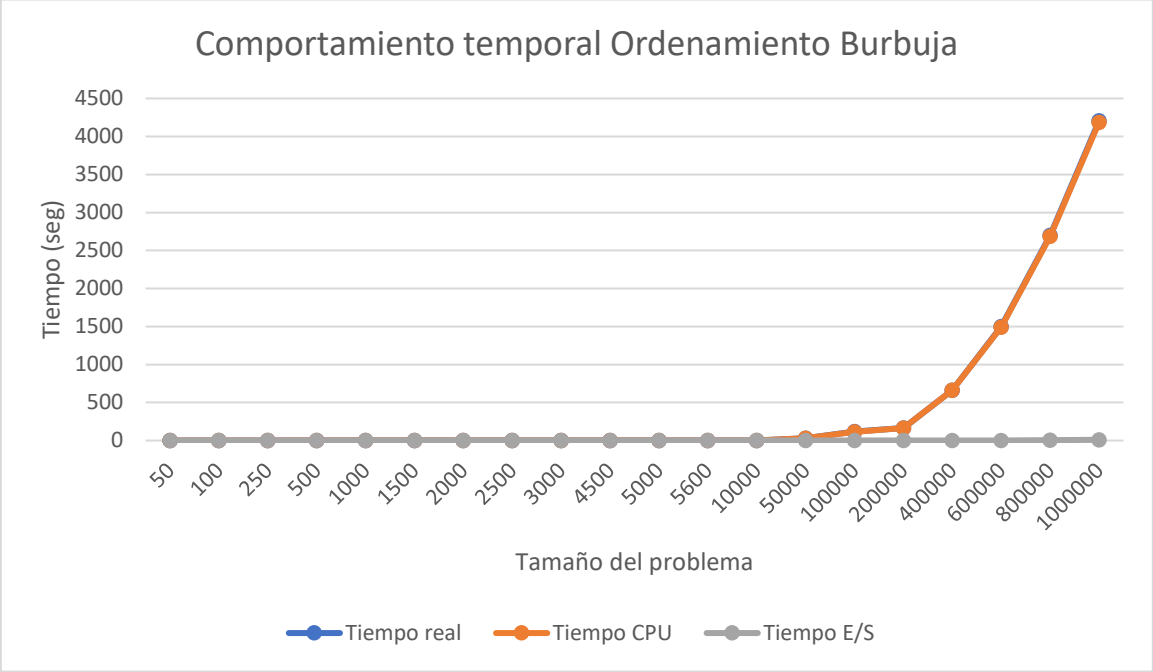
Ordenamiento Burbuja.

Tabla

Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
50	0.0000319481	0.0000280000	0.0000290000
100	0.0001199245	0.0001440000	0.0000000000
250	0.0007297993	0.0007530000	0.0000000000
500	0.0030159950	0.0030200000	0.0000000000
1000	0.0129630566	0.0121850000	0.0000000000
1500	0.0295910835	0.0268880000	0.0009790000
2000	0.0544049740	0.0452260000	0.0035680000
2500	0.0790159702	0.0772430000	0.0000000000
3000	0.1120791435	0.1096320000	0.0008150000
4500	0.2504179478	0.2447880000	0.0000000000
5000	0.3123328686	0.2964610000	0.0064930000
5600	0.5682771206	0.3742290000	0.0028230000
10000	1.2682828903	1.1820810000	0.0099580000
50000	30.4067990780	28.8568310000	0.1733150000
100000	118.3791751862	115.7556430000	0.2852610000
200000	165.6042111	165.406207	0.068306
400000	663.3611219	660.776908	0.803383
600000	1498.692925	1492.52732	2.776583
800000	2699.049169	2687.21182	5.285554
1000000	4204.313638	4184.98469	9.967584

Tabla 2. Comportamiento temporal vs. Tamaño del problema

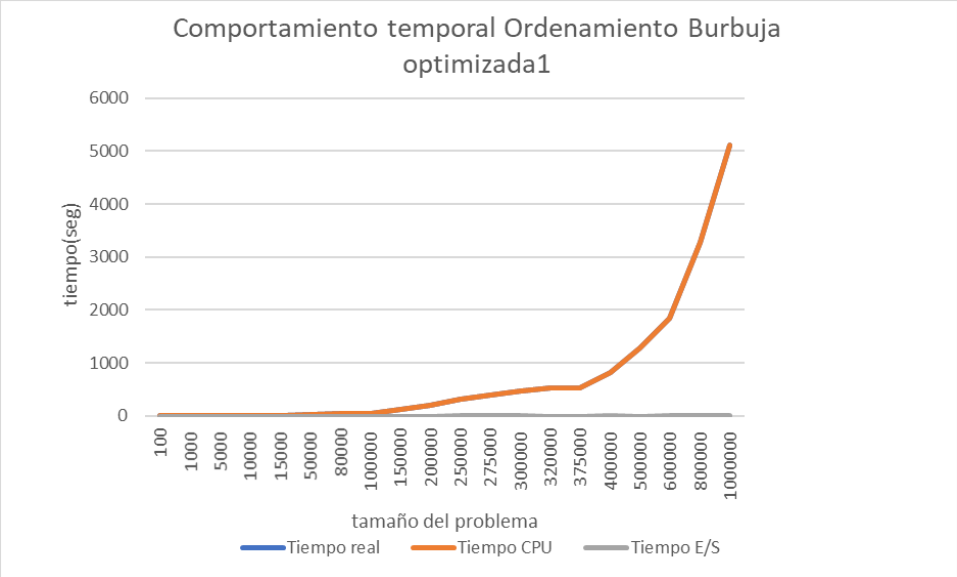
Gráfica.



Ordenamiento por Burbuja Optimizada 1

Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
100	0.0000448227s	0.0000510000s	0.0000000000s
1000	0.0032470226 s	0.0032540000 s	0.0000000000 s
5000	0.0947909355 s	0.0947990000 s	0.0000000000 s
10000	0.4267220497 s	0.4266970000 s	0.0000000000 s
15000	1.0256671906 s	1.0256450000 s	0.0000000000 s
50000	12.5068790913 s	12.5059200000 s	0.0000000000 s
80000	32.5662269592 s	32.5613540000 s	0.0002920000 s
100000	50.9534118176 s	50.9522070000 s	0.0001150000 s
150000	114.9378070831 s	114.9278900000 s	0.0000000000 s
200000	204.0861520767 s	204.0761480000 s	0.0000000000 s
250000	319.1513929367 s	319.0961840000 s	0.0239980000 s
275000	387.1039428711 s	387.0661930000 s	0.0081130000 s

300000	459.4268858433 s	459.3956740000 s	0.0080000000 s
320000	522.7856340408 s	522.7507320000 s	0.0000000000 s
375000			
400000	816.9416761398 s	816.8745540000 s	0.0039990000 s
500000	1275.2776398659s	1275.1701260000s	0.0319990000s
600000	1838.0762288570 s	1837.7865610000 s	0.1360390000 s
800000	3270.3919088840 s	3270.0245120000 s	0.1400070000 s
1000000	5111.2584860325 s	5110.8097400000 s	0.1359720000 s



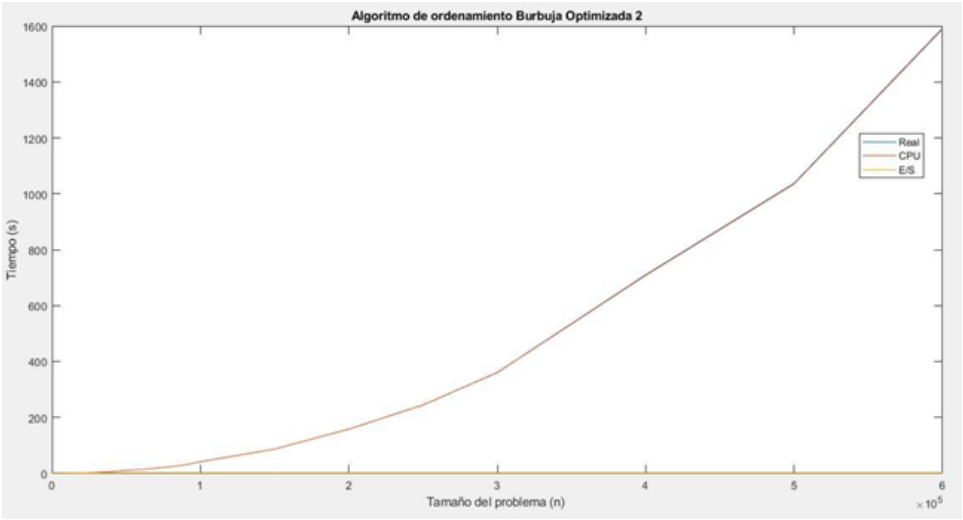
Ordenamiento por Burbuja Optimizada 2

Tabla

Tamaño de problema n	**Tiempo (seg) Burbuja Optimizada 2**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
100	3.89E-05	55000	0.0000234100
1,000	2.64E-03	4280000	0.0000000000
5,000	9.47E-02	102167000	0.0000000000
10,000	0.325105906	396149000	0.0000000000
50,000	11.75101113	13133221000	0.0000000000
100,000	41.35668302	54553557000	0.00234410000

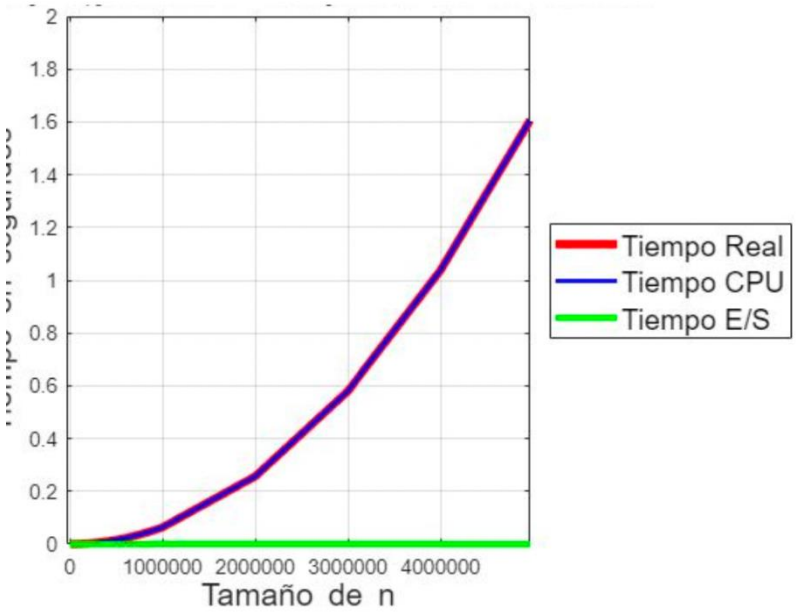
200,000	174.9560308	2.22547E+11	0.00000000
400,000	703.224402	1.04E+12	0.0000000000
500,000	703.224402	1.55E+12	0.00045250000
600,000	1475.37679	2.62E+12	0.0234450000
800,000	-	-	0
1,000,000	-	-	0
2,000,000	-	-	0
3,000,000	-	-	0
4,000,000	-	-	0
5,000,000	-	-	0
6,000,000	-	-	0
7,000,000	-	-	0
8,000,000	-	-	0
9,000,000	-	-	0

Gráfica



Ordenamiento por Inserción.

Gráfica.

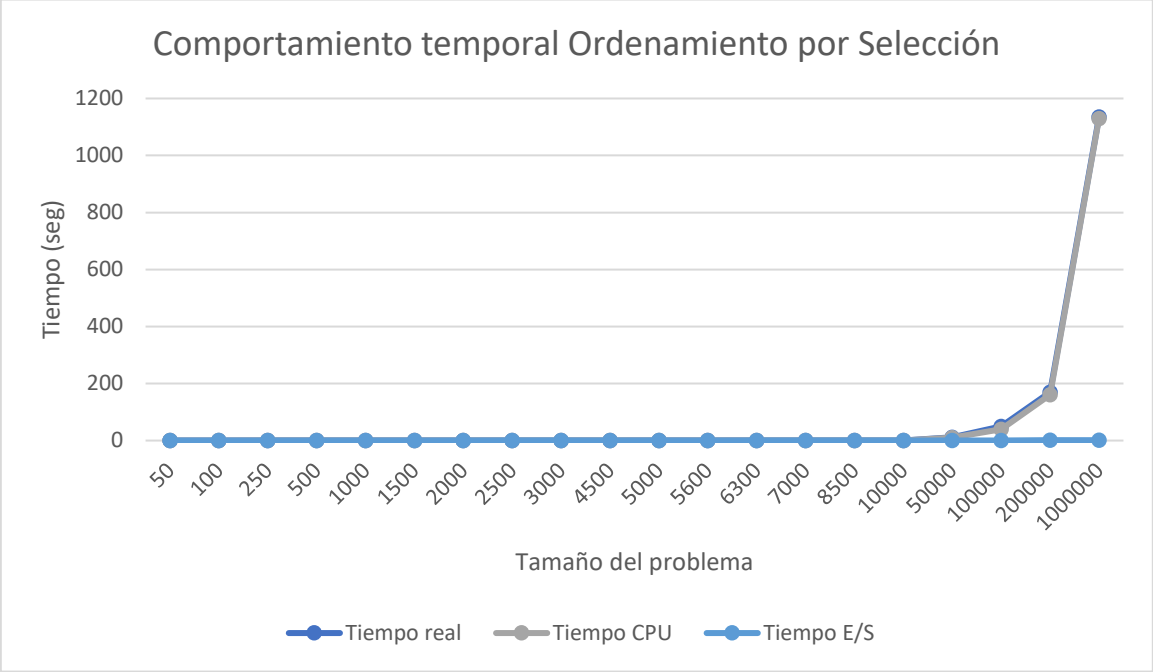


Ordenamiento por Selección.

Tabla

Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
50	0.0000929832	0.0000000000	0.0001220000
100	0.0001440048	0.0001750000	0.0000000000
250	0.0004088879	0.0004380000	0.0000000000
500	0.0012459755	0.0012740000	0.0000000000
1000	0.0048668385	0.0045170000	0.0000000000
1500	0.0109550953	0.0097970000	0.0000000000
2000	0.0192849636	0.0174740000	0.0000000000
2500	0.0302648544	0.0227820000	0.0035010000
3000	0.0423450470	0.0363580000	0.0014150000
4500	0.0976541042	0.0828860000	0.0000000000
5000	0.1199071407	0.0987260000	0.0031920000
5600	0.1581840515	0.1275800000	0.0000000000
6300	0.1705269814	0.1559550000	0.0048110000
7000	0.2020819187	0.1953610000	0.0017690000
8500	0.3772180080	0.2888910000	0.0033310000
10000	0.4592590332	0.3962610000	0.0075500000
50000	11.2790470123	9.9705700000	0.0745030000
100000	49.6225831509	40.1024660000	0.4078840000
200000	169.3458108902	160.4359560000	0.8371280000
1000000	1134.656388	1129.84297	1.333869

Gráfica.



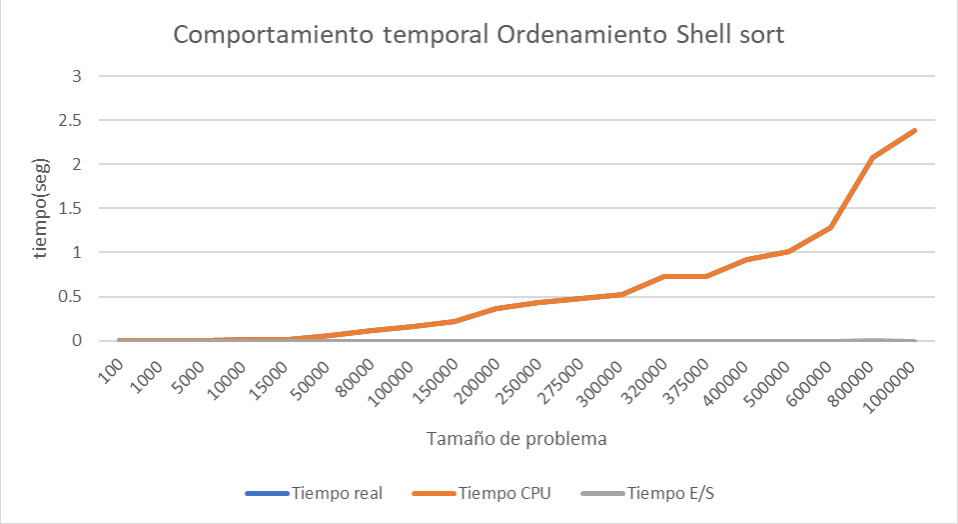
Ordenamiento de Shell

Tabla

Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
100	0.0000209808 s	0.0000270000 s	0.0000000000 s
1000	0.0003900528 s	0.0003950000 s	0.0000000000 s
5000	0.0036859512 s	0.0036920000 s	0.0000000000 s
10000	0.0103111267 s	0.0103160000 s	0.0000000000 s
15000	0.0148639679 s	0.0148550000 s	0.0000000000 s
50000	0.0624139309 s	0.0624160000 s	0.0000000000 s
80000	0.1157460213 s	0.1157530000 s	0.0000000000 s
100000	0.1585288048 s	0.1585300000 s	0.0000000000 s
150000	0.2112650871 s	0.2112710000 s	0.0000000000 s
200000	0.3602030277 s	0.3602000000 s	0.0000000000 s
250000	0.4273800850 s	0.4272400000 s	0.0000000000 s

275000	0.4821269512 s	0.4815910000 s	0.0000000000 s
300000	0.5253558159 s	0.5253520000 s	0.0000000000 s
320000	0.7270920277 s	0.7270760000 s	0.0000000000 s
375000	0.7273640633 s	0.7273570000 s	0.0000000000 s
400000	0.9162020683 s	0.9161690000 s	0.0000000000 s
500000	1.0166339874 s	1.0166100000 s	0.0000000000 s
600000	1.2895908356 s	1.2895670000 s	0.0000000000 s
800000	2.0744960308 s	2.0743340000 s	0.0000190000 s
1000000	2.3817708492 s	2.3817010000 s	0.0000000000 s

Gráfica



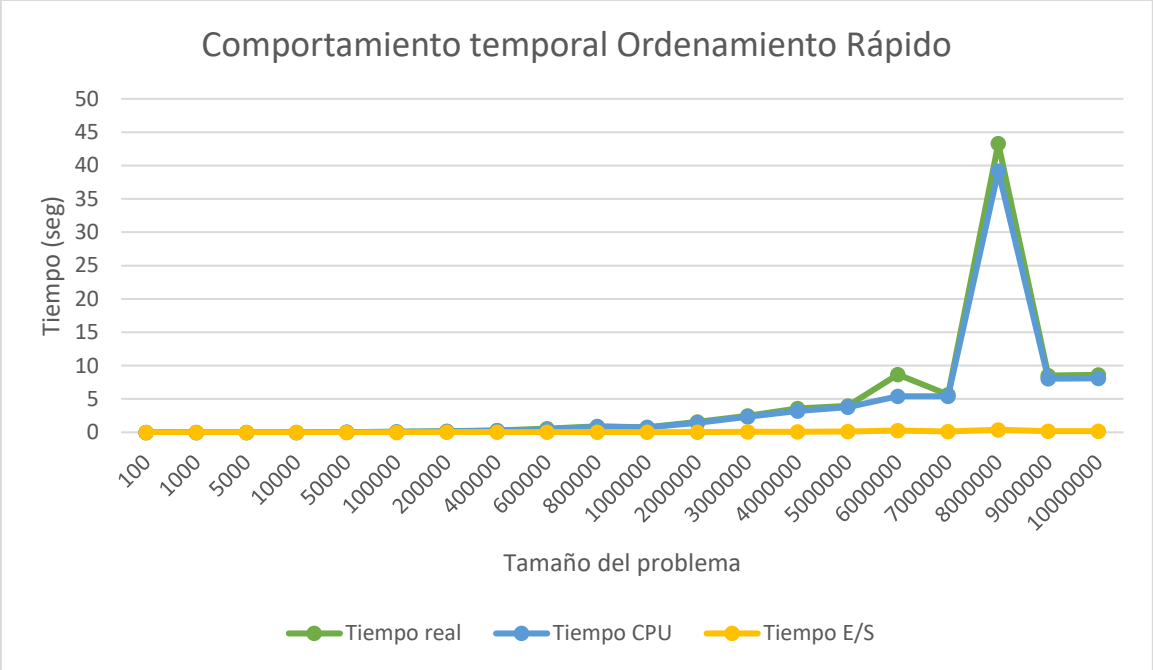
Ordenamiento Rápido

Tabla

Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
100	0.0002069473	0.0002030000	0.0000000000
1000	0.0005970001	0.0006230000	0.0000000000
5000	0.0038108826	0.0037570000	0.0000000000
10000	0.0058128834	0.0048740000	0.0007350000
50000	0.0348770618	0.0307100000	0.0032540000
100000	0.0717039108	0.0638880000	0.0000000000
200000	0.1953759193	0.1247400000	0.0110800000

400000	0.2961168289	0.2582720000	0.0097120000
600000	0.5487711430	0.4181980000	0.0189580000
800000	0.8963379860	0.8302040000	0.0283830000
1000000	0.7724988461	0.7305810000	0.0227160000
2000000	1.5501439571	1.4168660000	0.0543210000
3000000	2.4557430744	2.3383130000	0.0645170000
4000000	3.5675950050	3.2044590000	0.0923250000
5000000	3.9616911411	3.7631430000	0.1183650000
6000000	8.6657218933	5.4241130000	0.2590580000
7000000	5.6700332165	5.4118610000	0.1350400000
8000000	43.2867641449	39.1925800000	0.3886960000
9000000	8.5369639397	8.0255930000	0.1558100000
10000000	8.6005420685	8.1107650000	0.1806300000

Gráfica



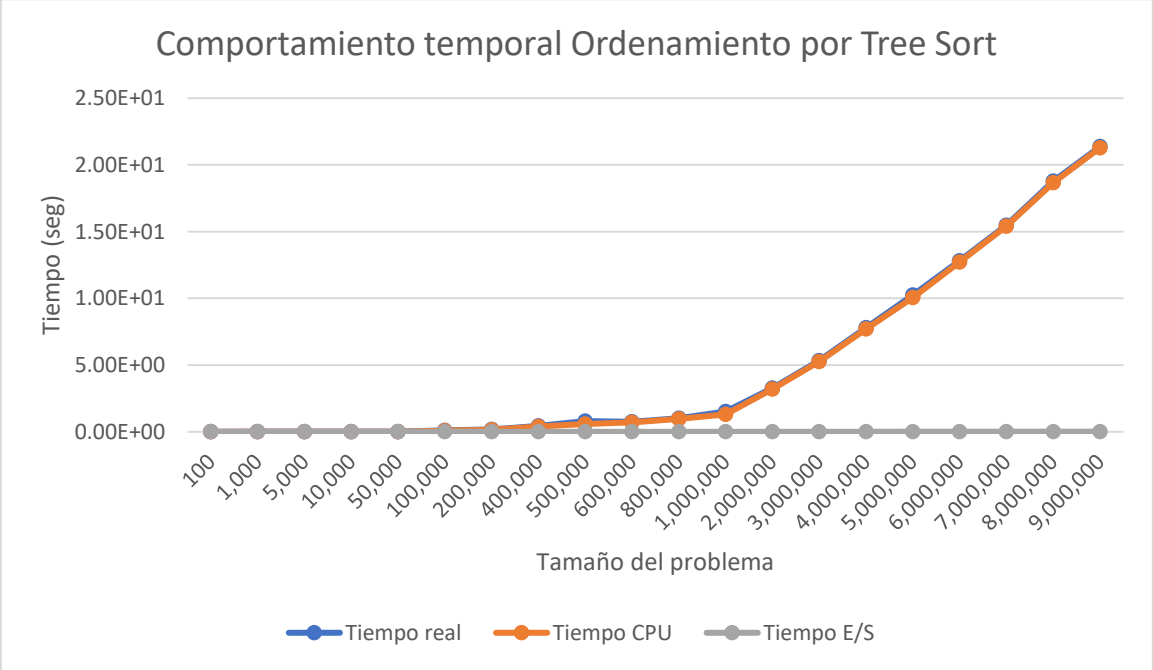
Ordenamiento por Tree Sort:

Tabla

Tamaño de problema n	**Tiempo (seg) Algoritmo Tree Sort**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
100	1.31E-05	1.70E-05	0.0000000000
1,000	2.03E-04	2.03E-04	0.0000000000
5,000	1.35E-03	1.35E-03	0.0000000000
10,000	2.34E-03	2.35E-03	0.0000000000

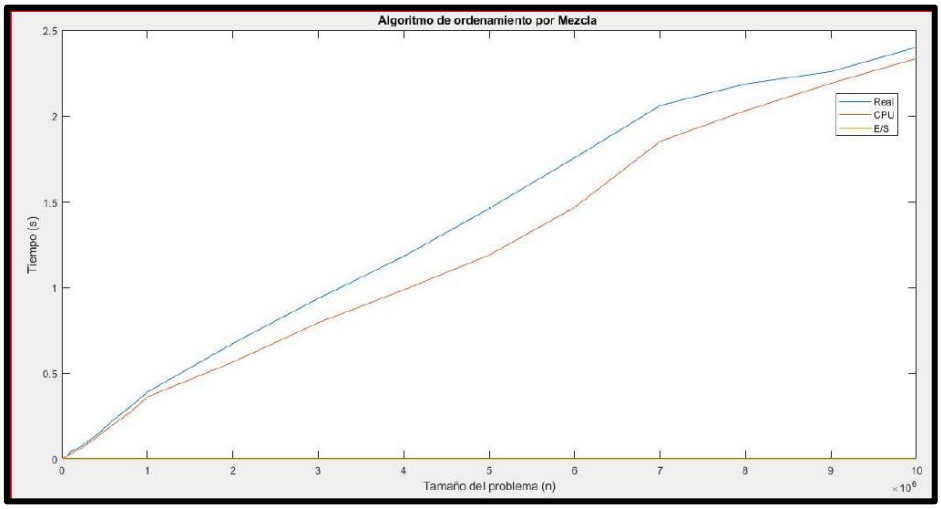
50,000	0.01905489	1.90E-02	0.0000000000
100,000	0.11226511	0.072086	0.0000000000
200,000	0.172807932	0.149724	0.0026740000
400,000	0.438128948	0.39228	0.0000000000
500,000	0.80230093	0.574172	0.0000000000
600,000	0.763154984	0.711087	0.0000000000
800,000	1.018180847	9.60E-01	0.0000000000
1,000,000	1.508898974	1.29972	0.0000000000
2,000,000	3.259153843	3.19E+00	0.0000000000
3,000,000	5.332906961	5.259874	0.0000000000
4,000,000	7.801563025	7.695969	0.0000000000
5,000,000	10.255265	10.056551	0.00093423
6,000,000	12.81993198	12.708296	0.00100432
7,000,000	15.47936606	15.415762	0.00113434
8,000,000	18.78603005	18.660815	0.00123343
9,000,000	21.3665452	21.290105	0.00231232

Gráfica.



Ordenamiento Merge.

Gráfica.



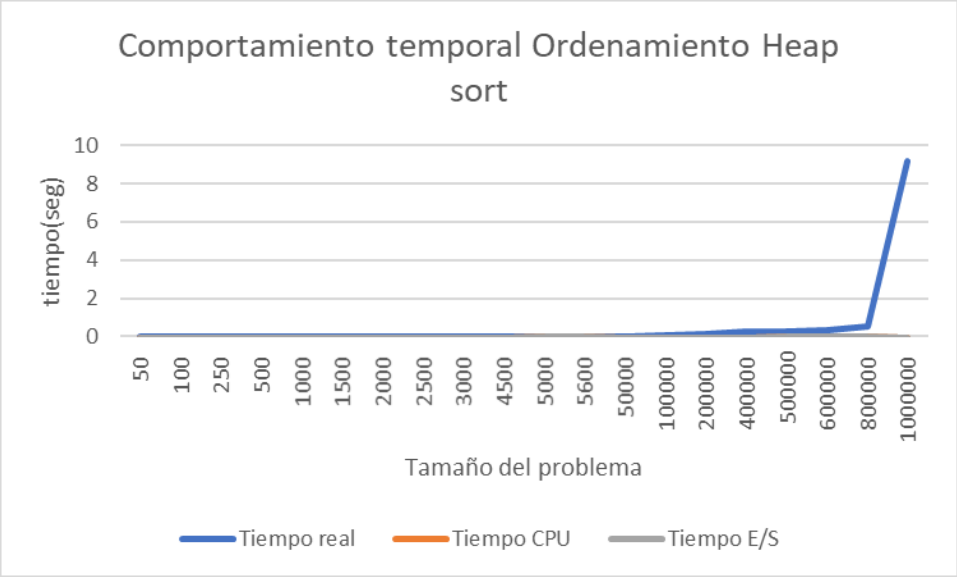
Ordenamiento Heap sort

Tabla

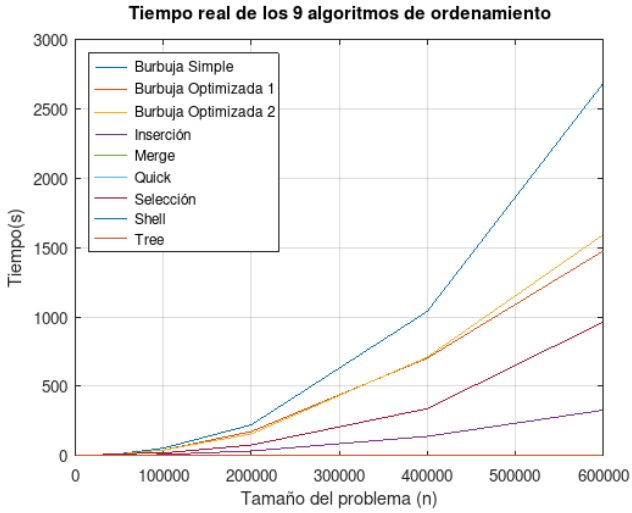
Tamaño de problema n	**Tiempo (seg)**		
	Tiempo real	*Tiempo CPU*	*Tiempo E/S*
50	0.0000128746 s	0.0000190000 s	0.0000000000 s
100	0.0000278950 s	0.0000340000 s	0.0000000000 s
250	0.0000591278 s	0.0000650000 s	0.0000000000 s
500	0.0001289845 s	0.0001340000 s	0.0000000000 s
1000	0.0042130947 s	0.0042180000 s	0.0000000000 s
1500	0.0004768372 s	0.0004810000 s	0.0000000000 s
2000	0.0006270409 s	0.0006310000 s	0.0000000000 s
2500	0.0009357929 s	0.0009420000 s	0.0000000000 s
3000	0.0011031628 s	0.0011080000 s	0.0000000000 s
4500	0.0017290115 s	0.0017340000 s	0.0000000000 s
5000	0.0017678738 s	0.0000000000 s	0.0017720000 s
5600	0.0020208359 s	0.0000000000 s	0.0020200000 s
50000	0.0231769085 s	0.0231820000 s	0.0000000000 s
100000	0.0502049923 s	0.0502110000 s	0.0000000000 s

200000	0.1052291393 s	0.1052280000 s	0.0000000000 s
400000	0.2743258476 s	0.2743310000 s	0.0000000000 s
500000	0.2936809063 s	0.2936140000 s	0.0000600000 s
600000	0.3577418327 s	0.3576910000 s	0.0000410000 s
800000	0.5010538101 s	0.5009080000 s	0.0001350000 s
1000000	9.1885988712 s	9.1879820000 s	0.0000050000 s

Gráfica



Gráfica comparativa del tiempo real del comportamiento de los Algoritmos.
Con base en las gráficas anteriores, se va a realizar una gráfica donde se conjunte el tiempo real de cada uno de los algoritmos para compararlos entre sí.

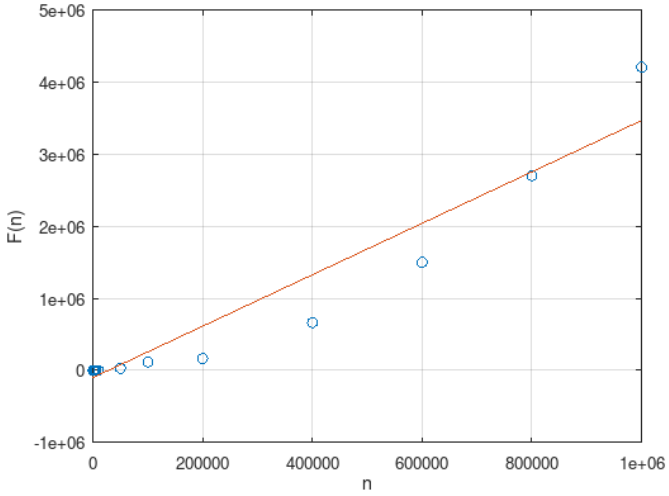


Aproximación de la función del comportamiento temporal.

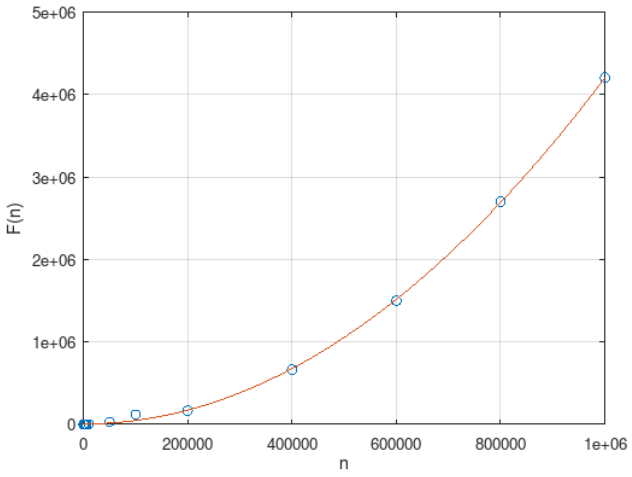
Aquí se verán las gráficas que muestran una aproximación de la función del comportamiento temporal de cada Algoritmo. Se hará de forma polinomial con distintos grados (1,2,3 y 6) para ver con cual se asemeja a los datos de ejecución; si con ninguno lo hace, entonces se procederá a hacer una aproximación logarítmica.

Burbuja Simple

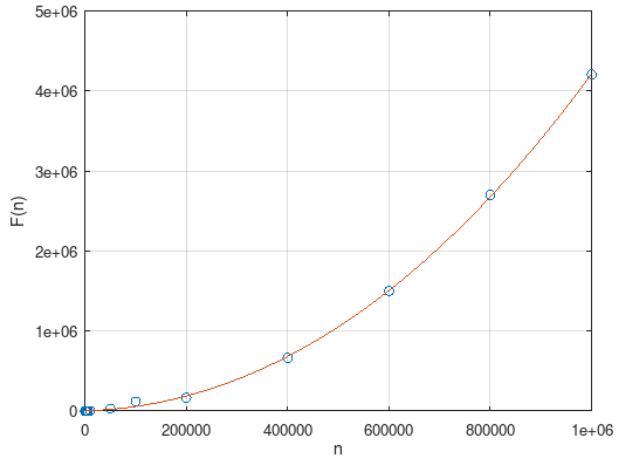
Grado 1.



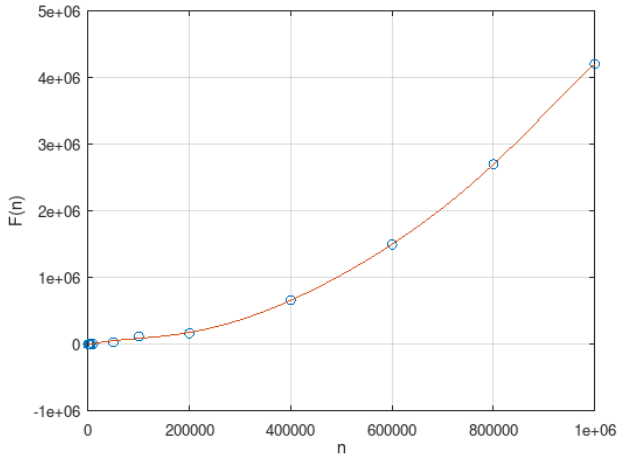
Grado 2.



Grado 3.

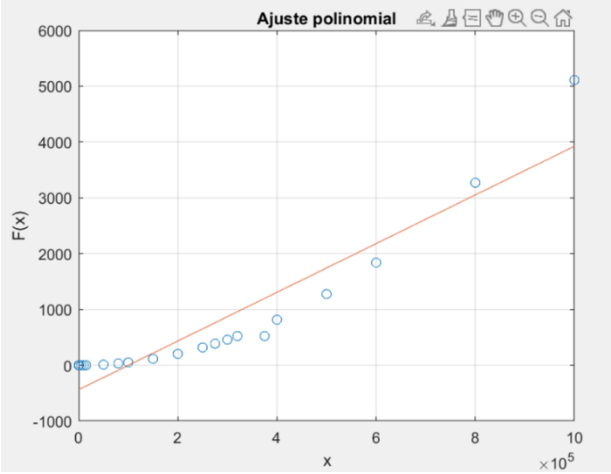


Grado 6.

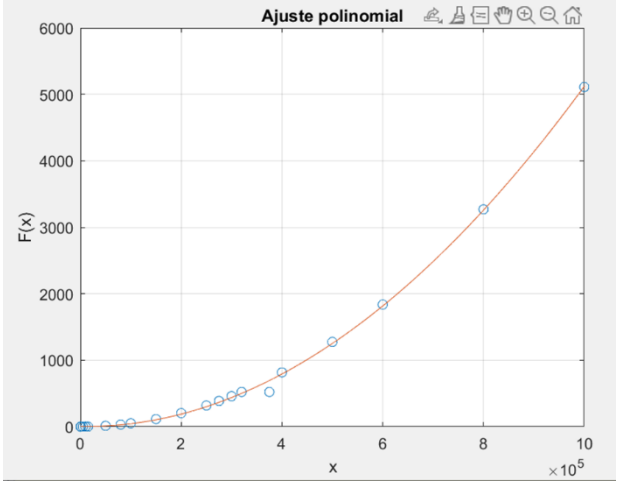


Burbuja Optimizada 1

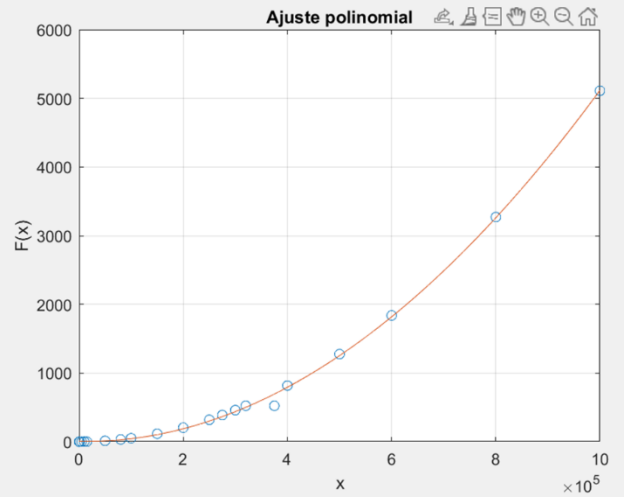
Grado 1



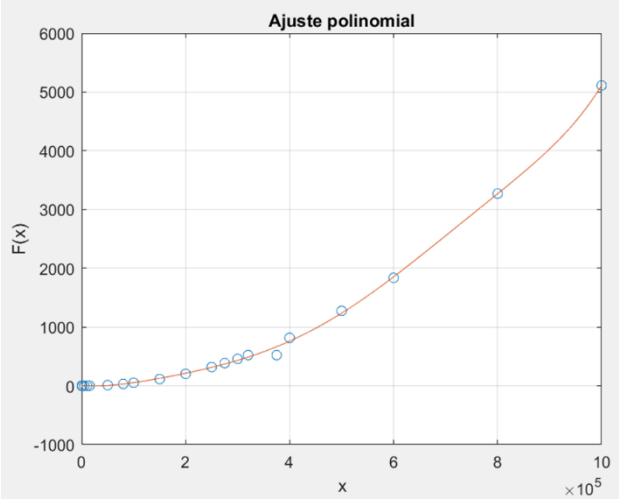
Grado 2



Grado3

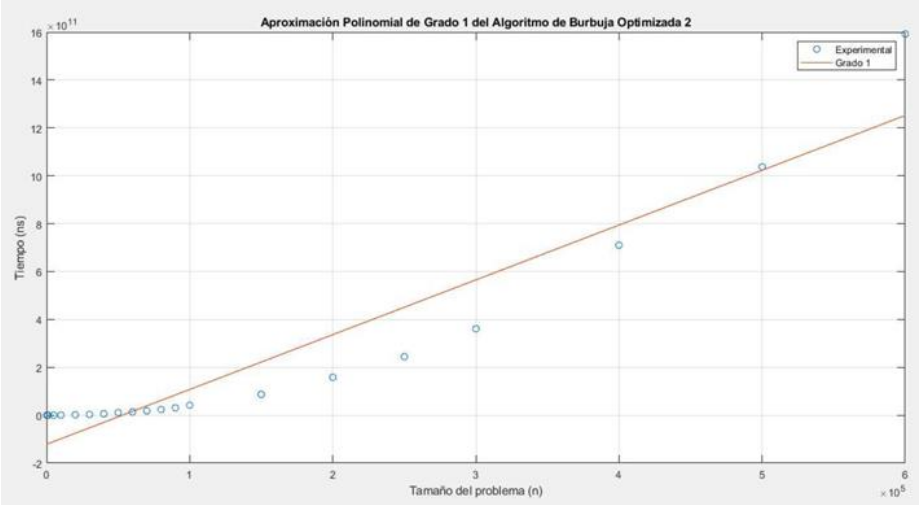


Grado6

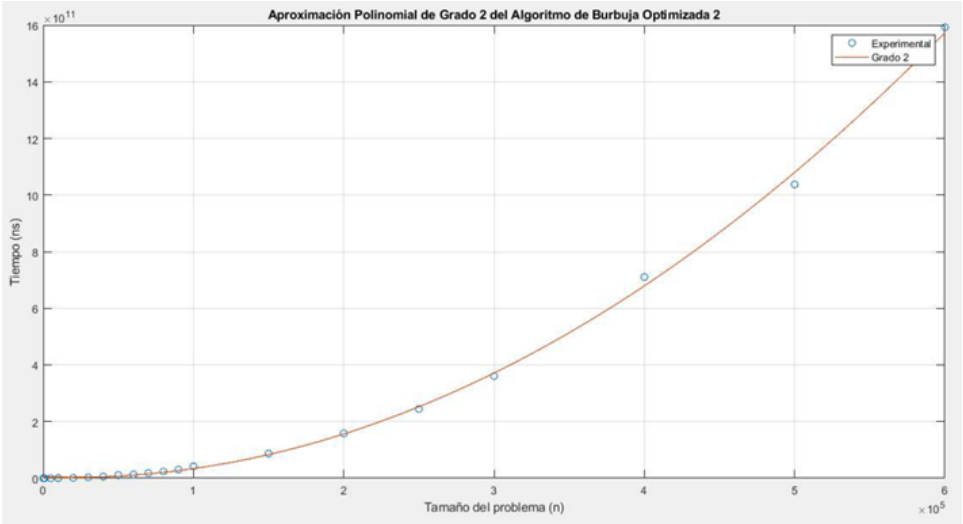


Burbuja optimizada 2.

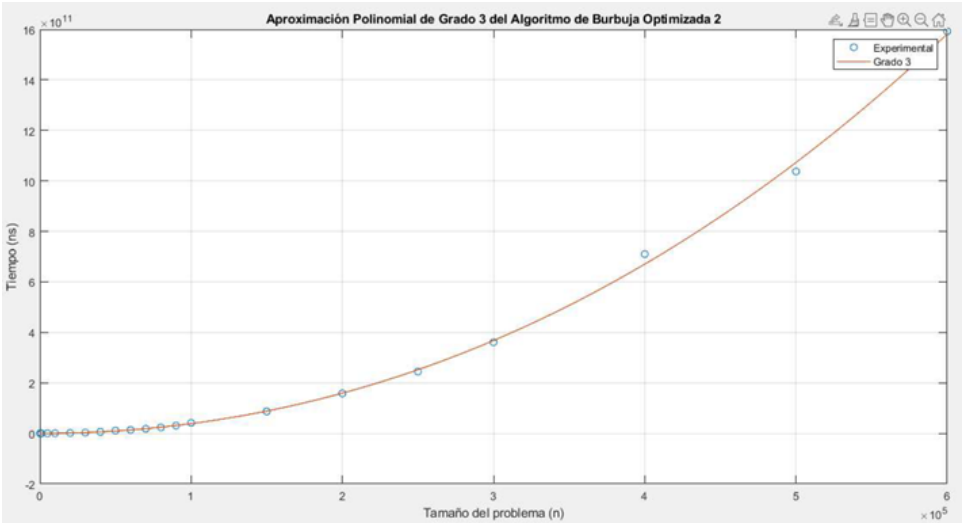
Grado 1.



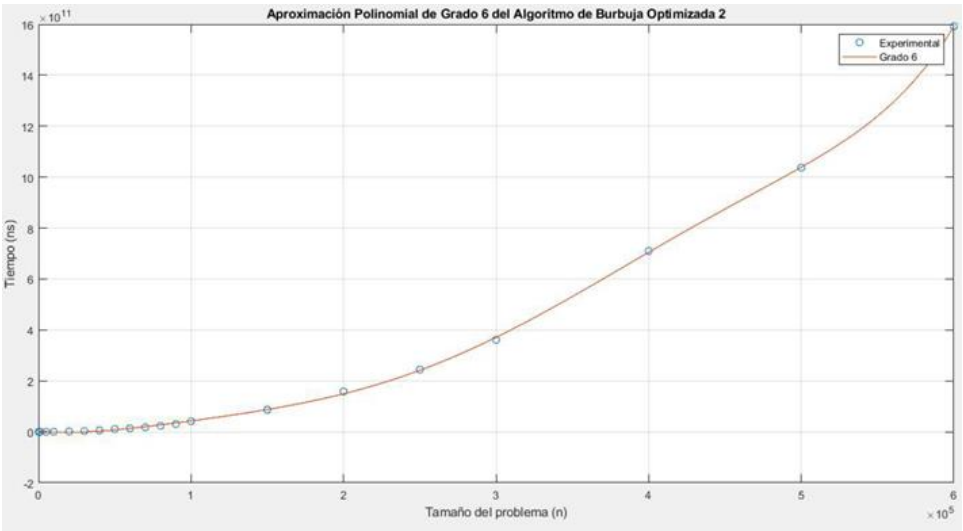
Grado 2:



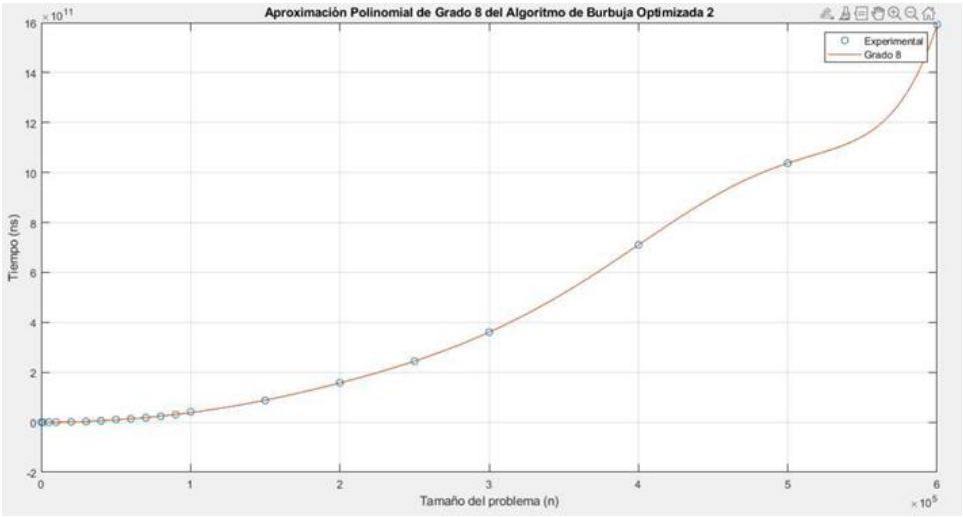
Grado 3.



Grado 6.

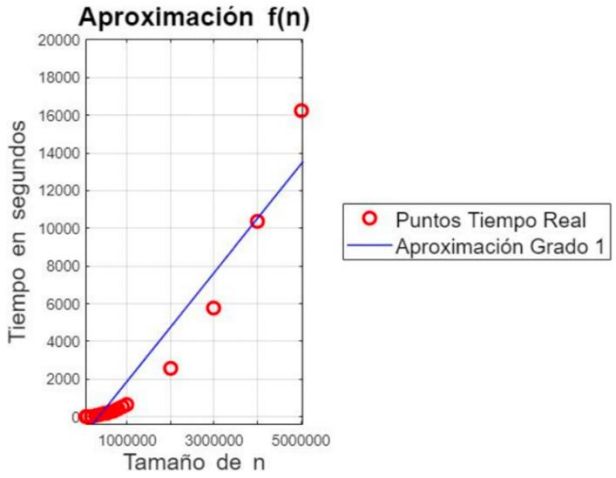


Grado 8.

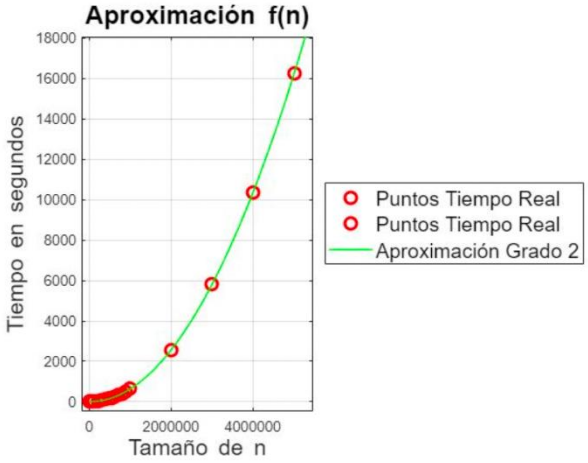


Inserción.

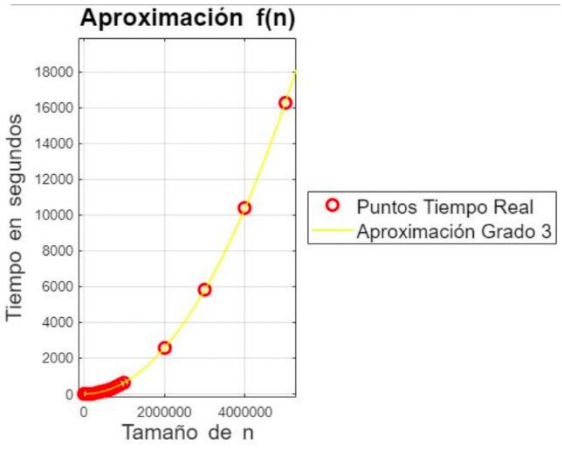
Grado 1.



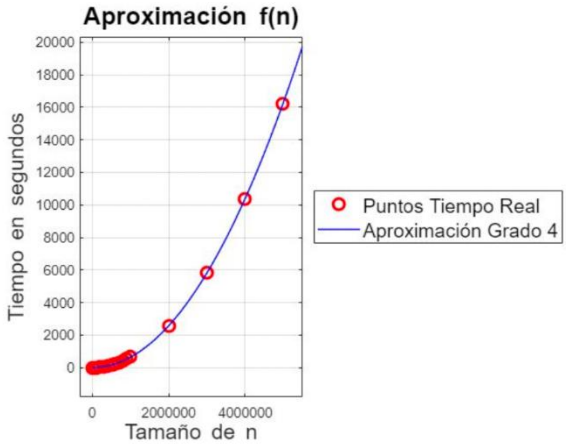
Grado 2.



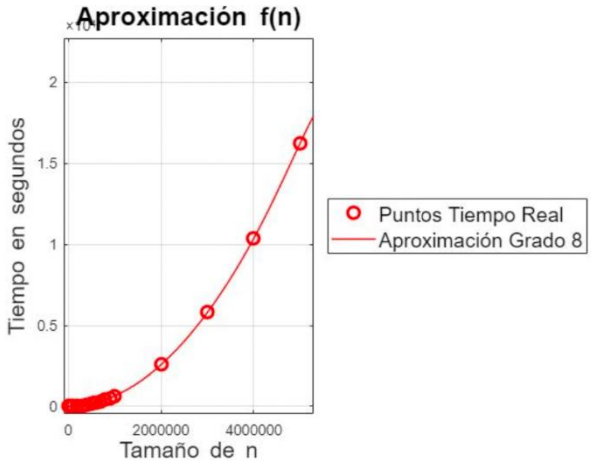
Grado 3.



Grado 4.

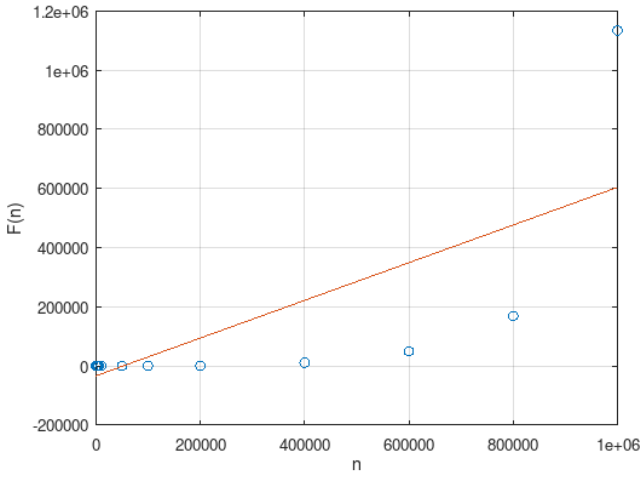


Grado 8.

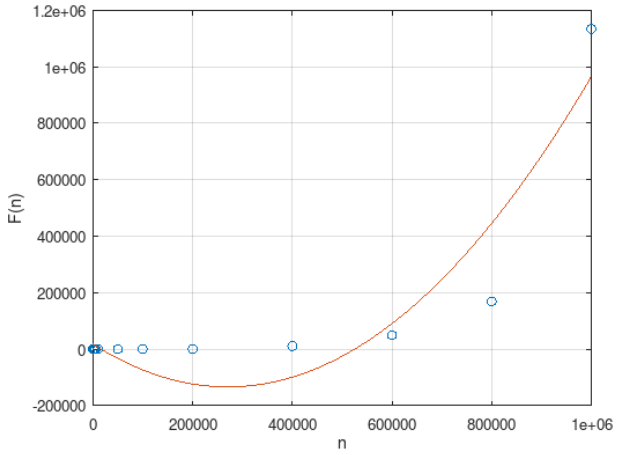


Selección.

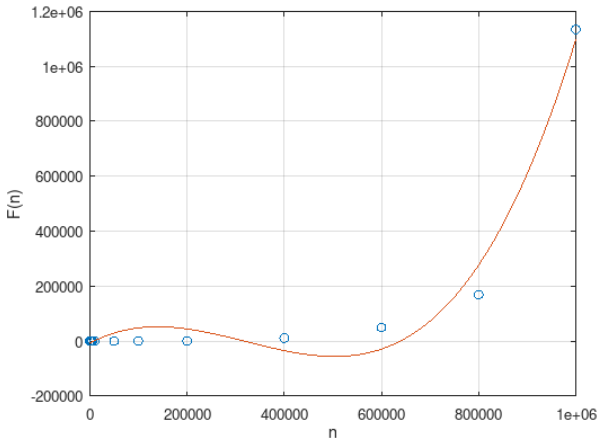
Grado 1.



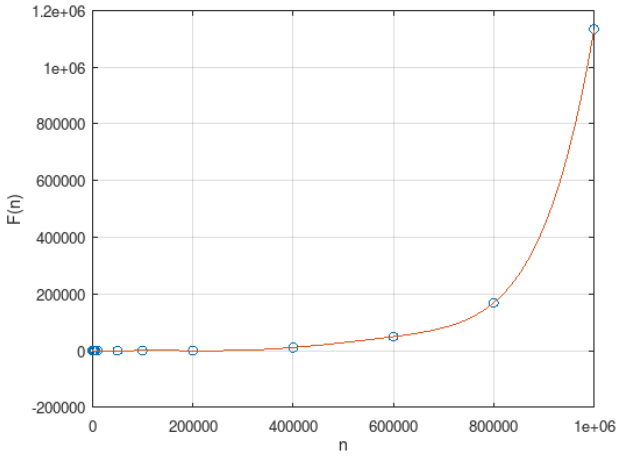
Grado 2.



Grado 3.

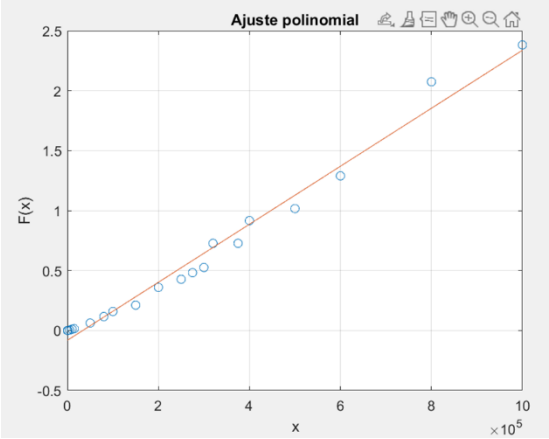


Grado 6.

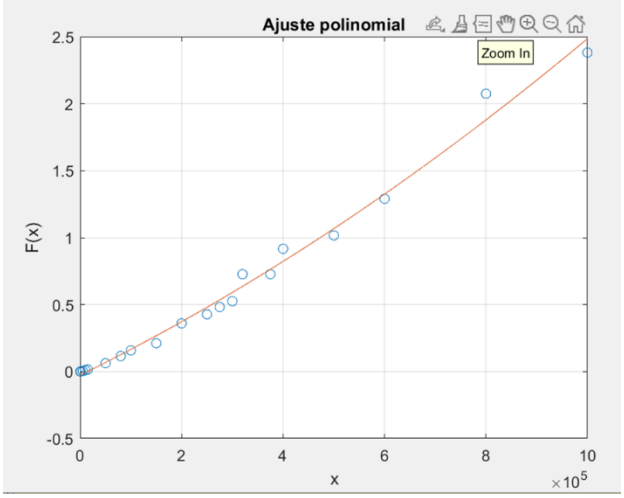


Shell

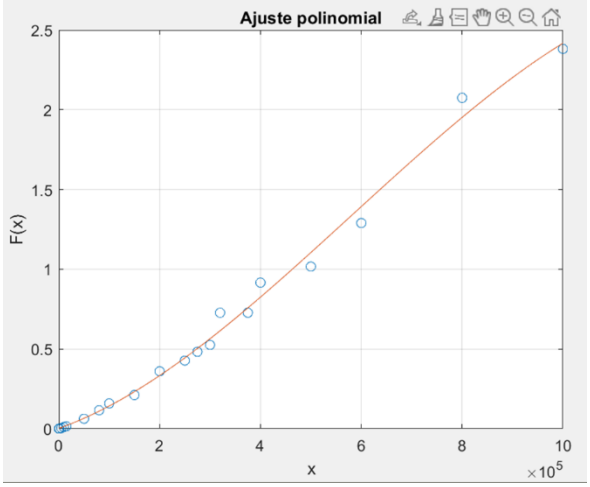
Grado1



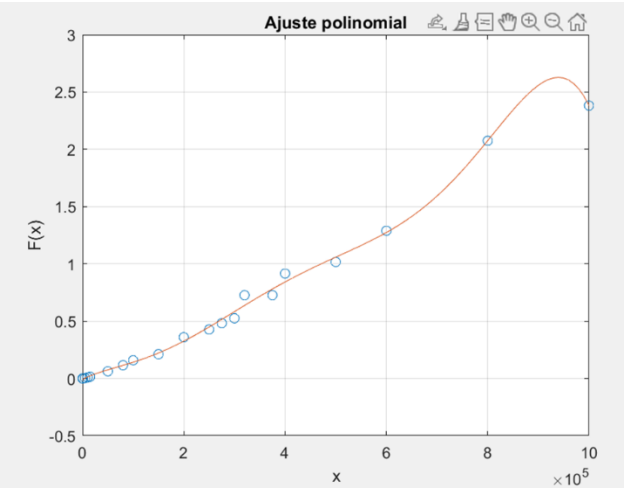
Grado2



Grado3

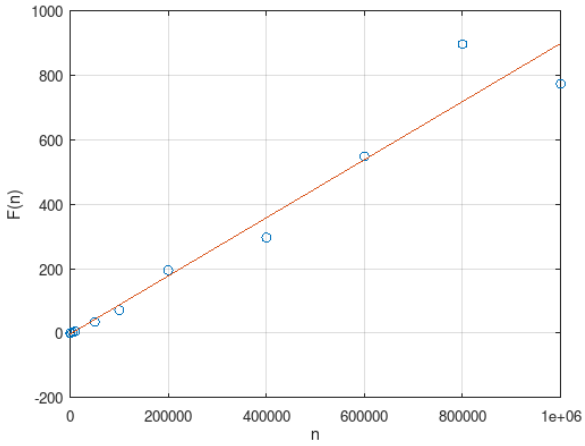


Grado6

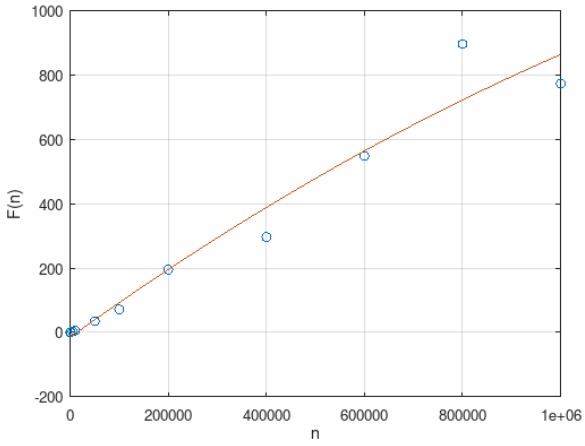


Rápido.

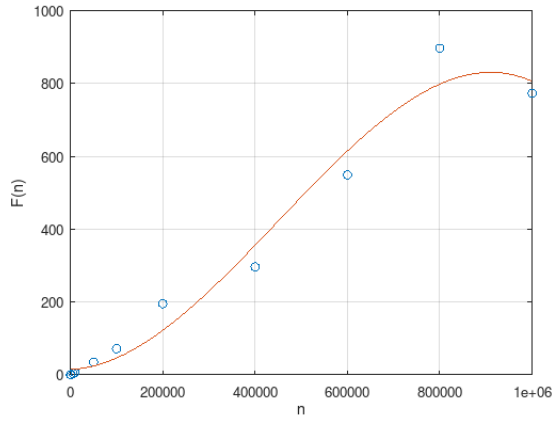
Grado 1.



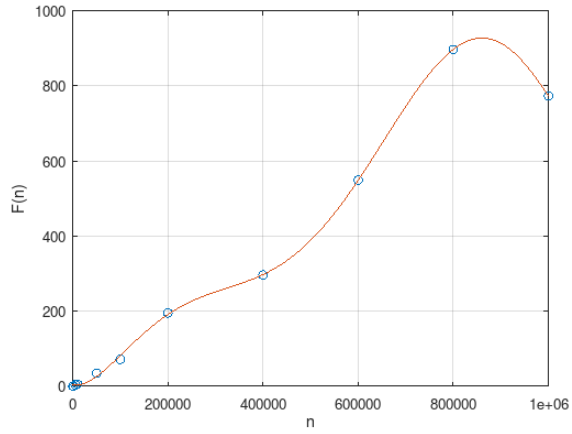
Grado 2.



Grado 3.

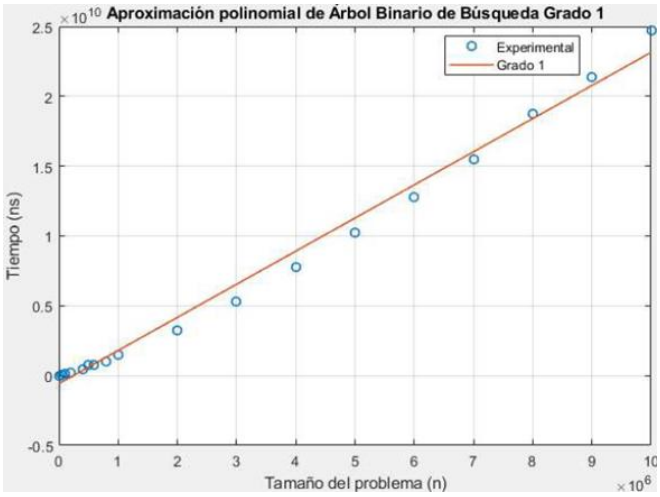


Grado 6.

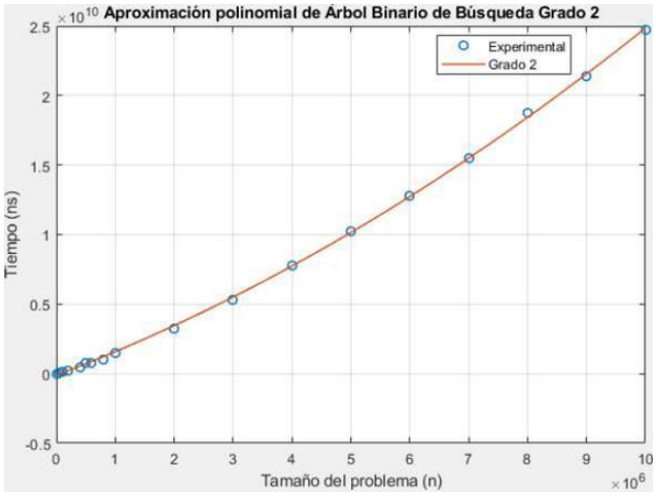


Tree Sort.

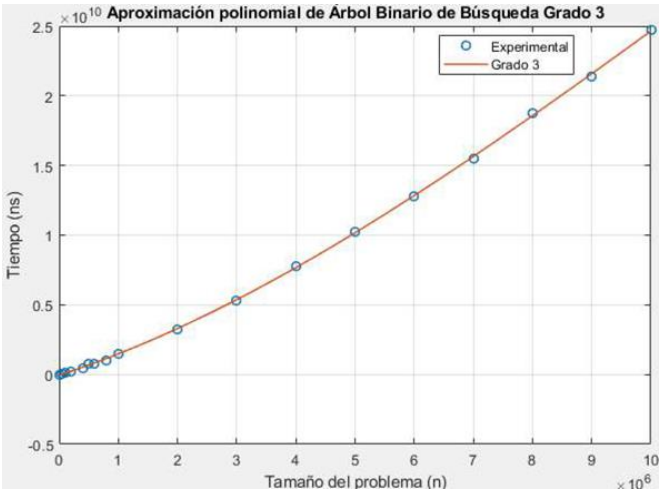
Grado 1.



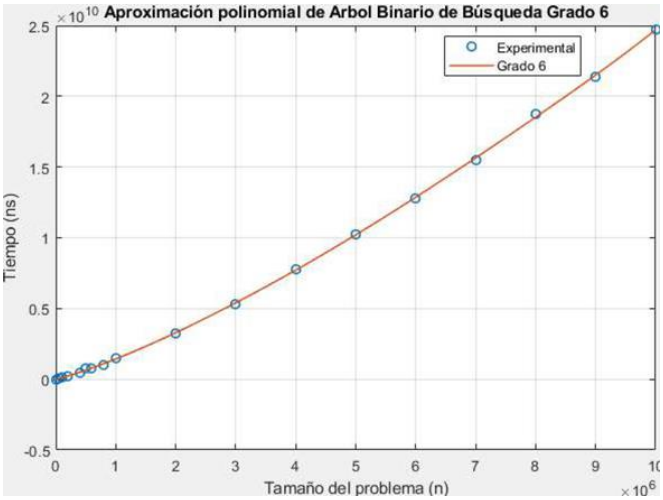
Grado 2:



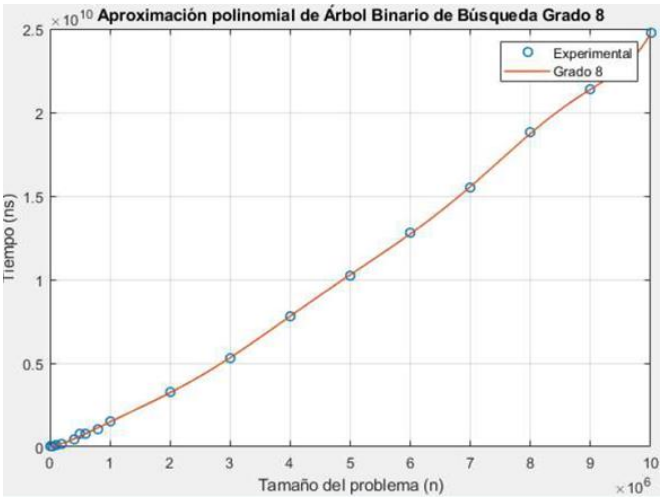
Grado 3



Grado 6

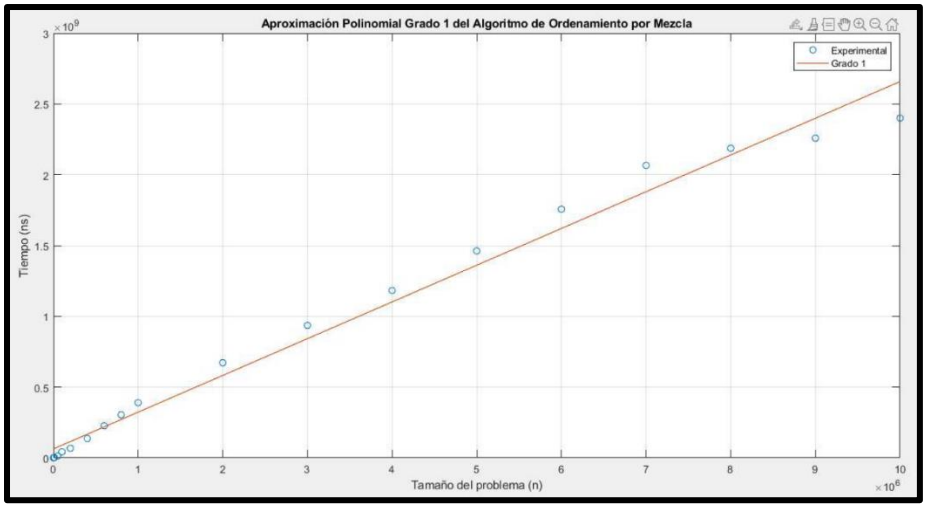


Grado 8

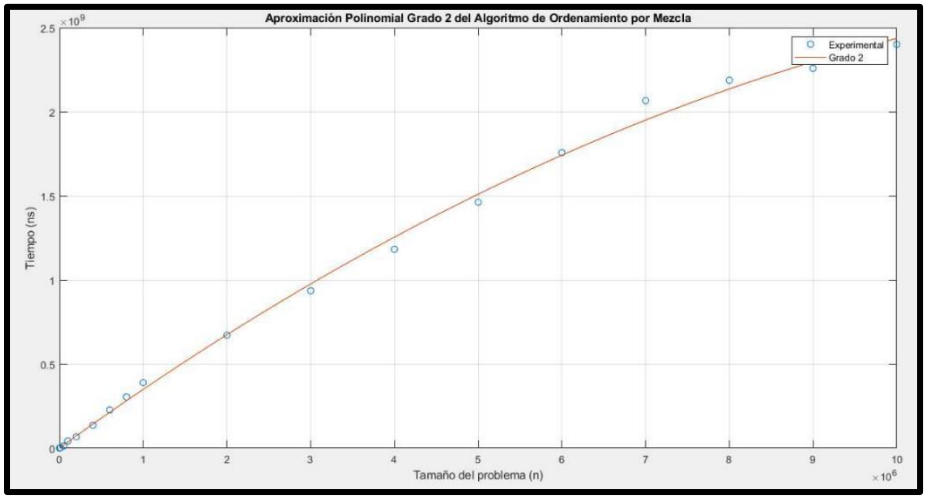


Merge Sort

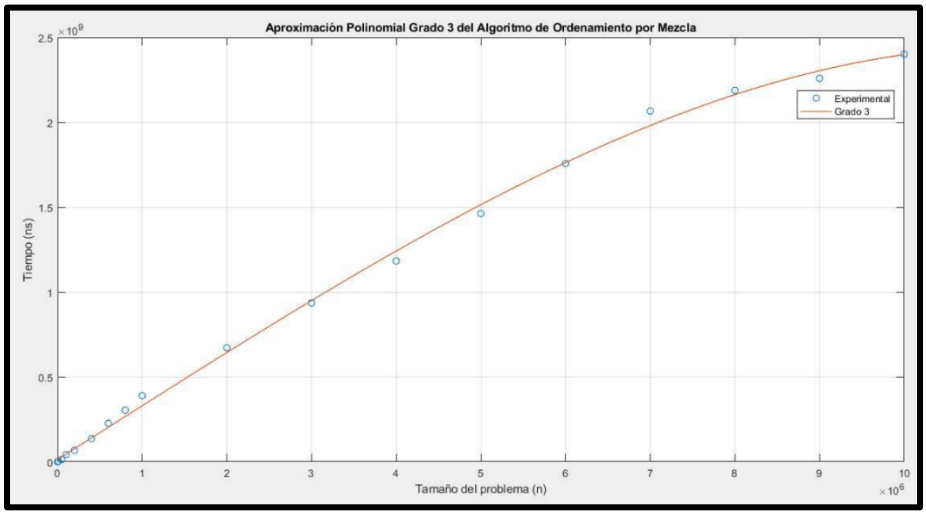
Grado 1:



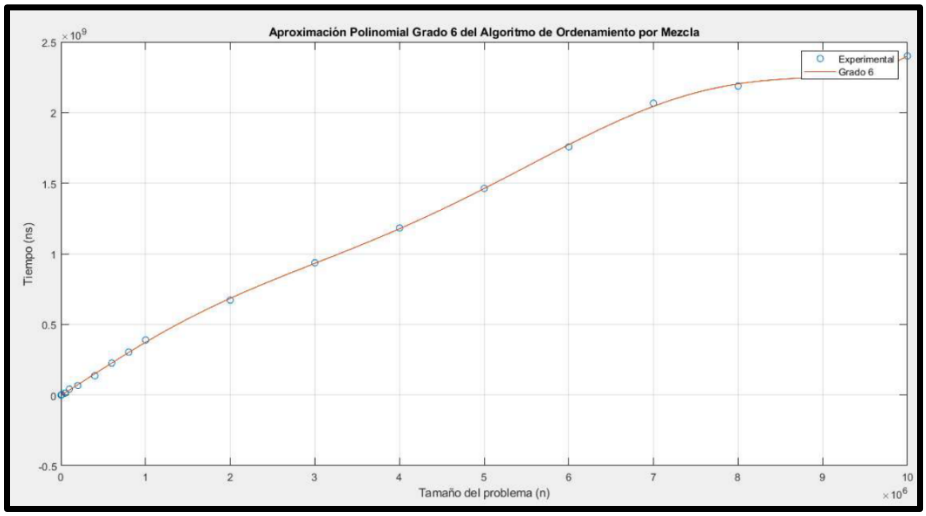
Grado 2:



Grado 3:

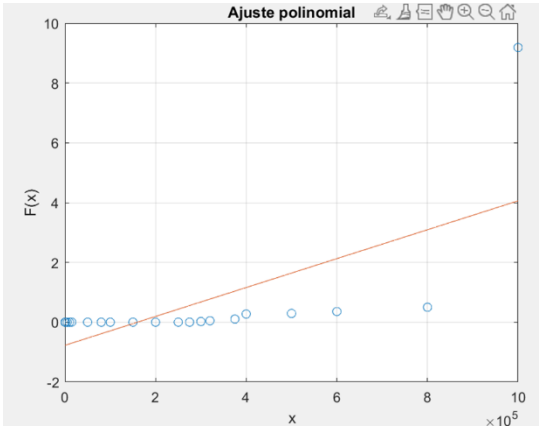


Grado 6:

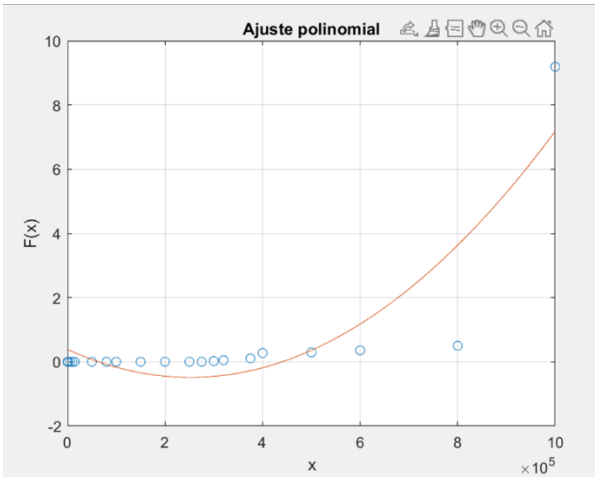


Heap sort

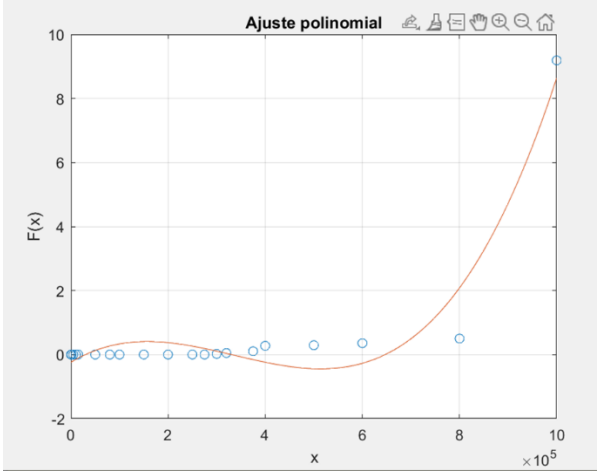
Grado1



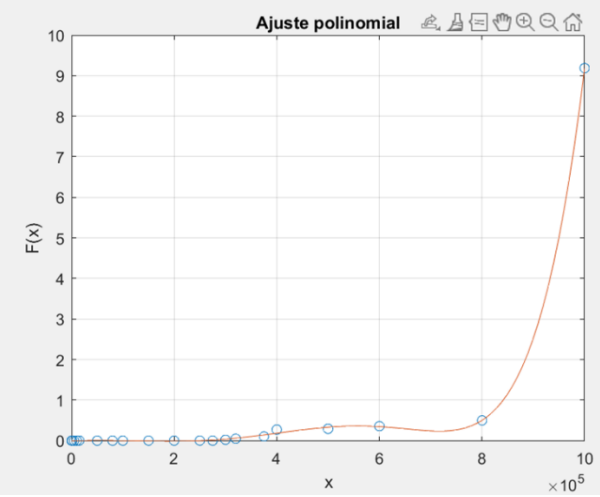
Grado2



Grado3

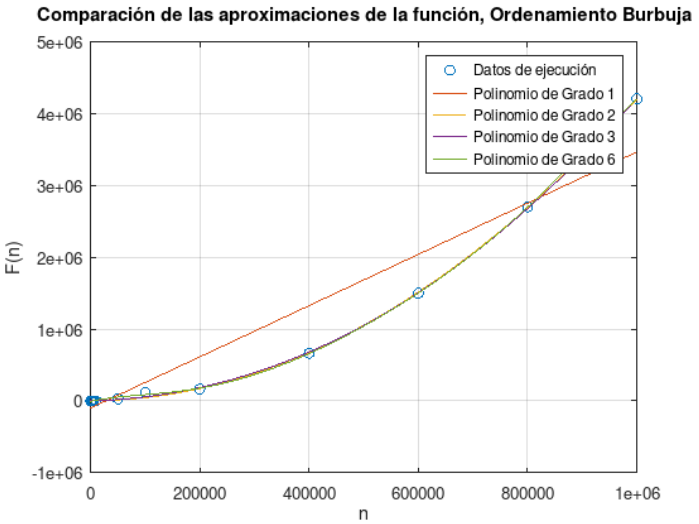


Grado6



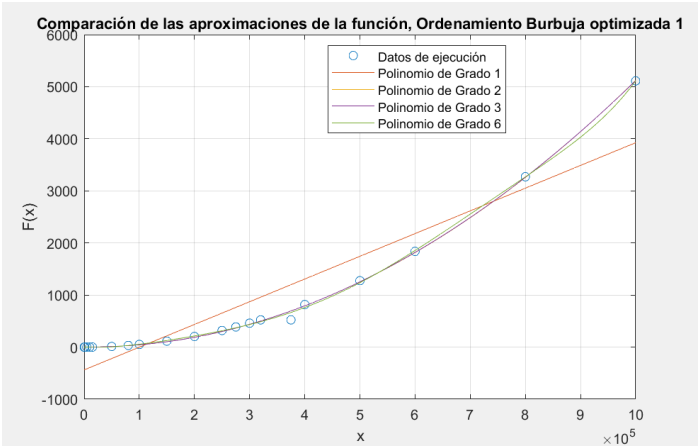
Comparativa de las aproximaciones de la función de complejidad temporal.
En esta sección se conjuntarán en una sola gráfica para cada Algoritmo los 4 grados (1,2,3 y 6), que fungen como aproximaciones de la función del comportamiento temporal, para poder compararlos.

Burbuja.



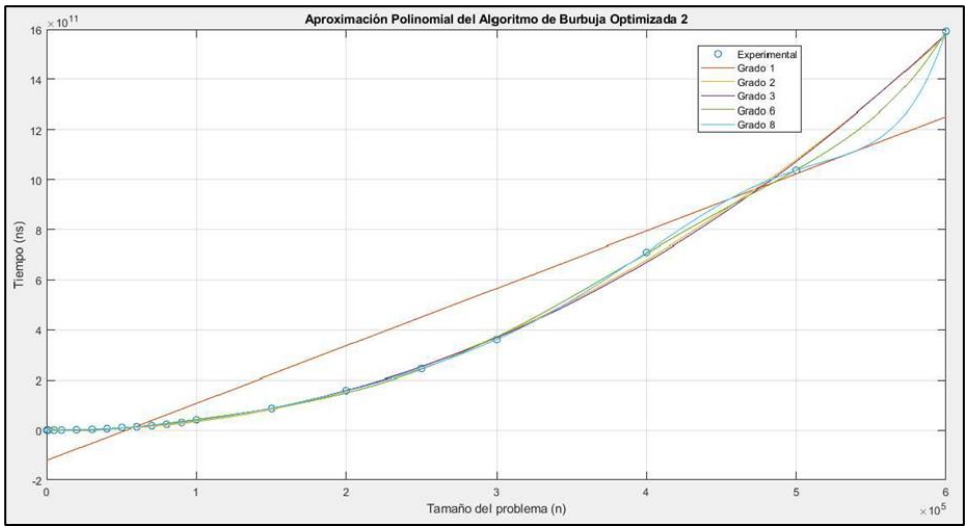
En este primer algoritmo, vemos que el grado 1 es simplemente una recta, no se acopla para nada a los datos de ejecución. Sin embargo, en los polinomios de grado 2 a 6, ya se hace una curvatura que se ajusta a lo que necesitamos, así que se decidió tomar el de grado 2.

Burbuja optimizada 1.



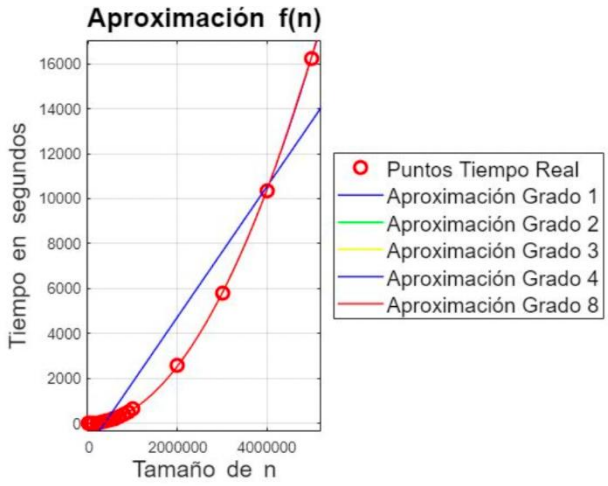
En este caso curre algo muy parecido que el algoritmo de burbuja ya que el grado uno es una recta y los polinomios que le siguen ya se ajustan de una mejor manera a lo que necesitamos por lo que tomamos el grado 2 nuevamente.

Burbuja Optimizada 2.



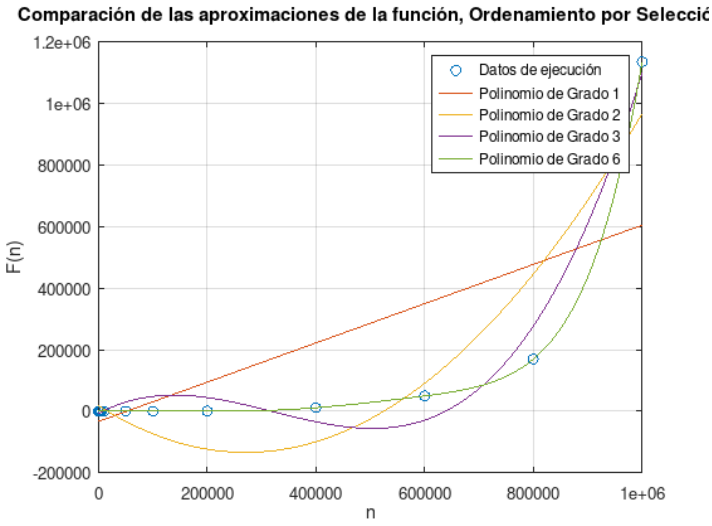
En este ordenamiento se decidió tomar la gráfica con aproximación lineal de grado 8, debido a que con las gráficas inferiores no pasaba por casi ningún punto, por otro lado, al tomar la gráfica de grado 6 se pudo observar que la gráfica empezaba desde antes del cero por lo tanto se descartó al no haber tiempo negativos.

Inserción.



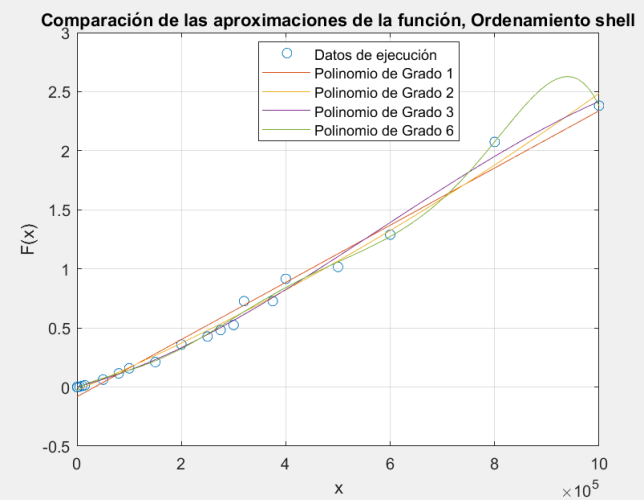
Podemos observar que en el intervalo de 0 a 5000000 todas las aproximaciones se comportan de manera monótona creciente, sin embargo para valores más grandes no todas cumplen esta condición (en específico, la aproximación grado 4 y 8), es por ello que la curva que más se acerca a todos los valores de nuestros puntos reales es la aproximación con grado 3.

Selección.



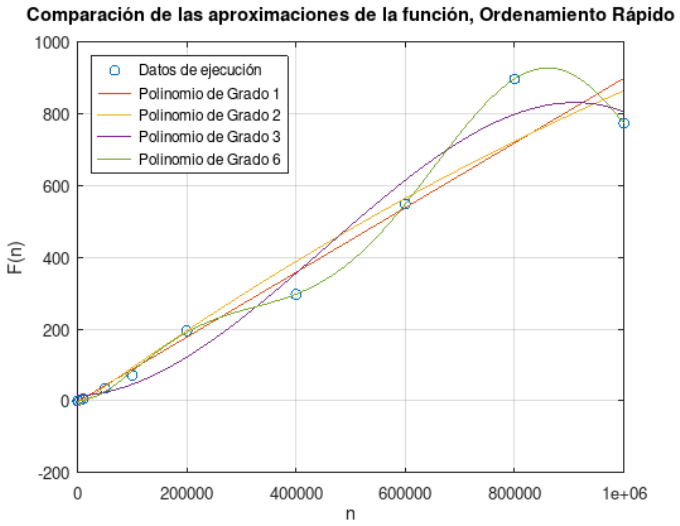
En este algoritmo se ve una mayor diferencia entre los grados. En el grado uno solo vemos una recta, en el 2 más curvatura, pero no toca ningún punto, al igual que el 3. Sin embargo, si vemos el grado 6, este toca todos los puntos de los datos de ejecución, por lo tanto, se elegirá ese.

Shell



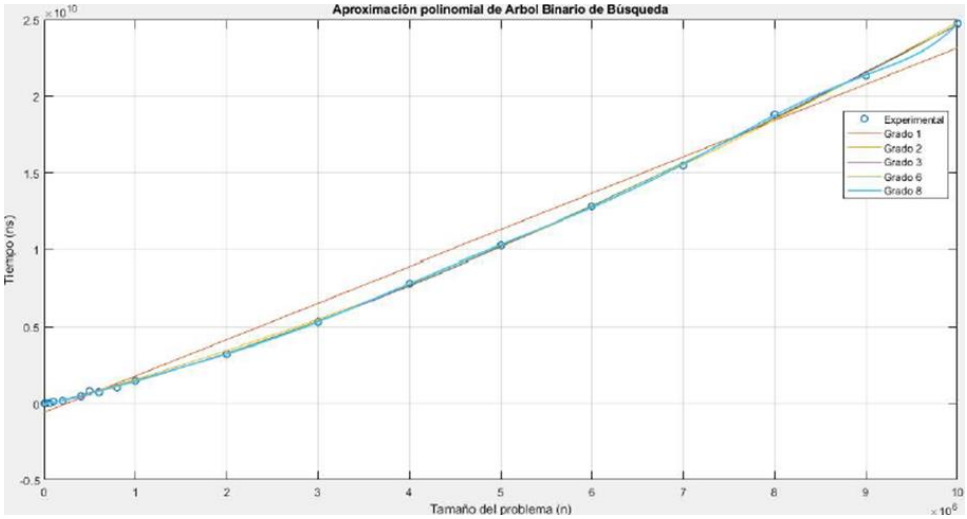
Podemos ver en este caso que el grado 6 esta toca la mayoría de los puntos de los datos de ejecución por lo que se elegirá este.

Rápido.



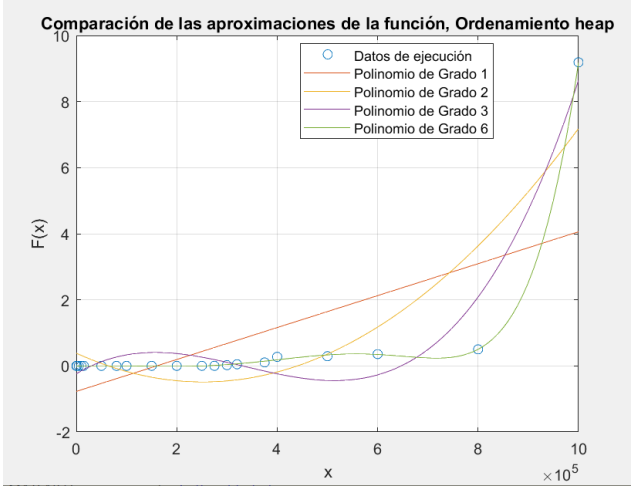
En las gráficas de este algoritmo, las gráficas con aproximación polinomial de grado 1 y 2 son sólo líneas rectas, en el 3 hay un poco más de curvatura alcanzando a tocar los primeros puntos, pero para poder realmente dar una buena aproximación, se deben tocar todos los que se marcan en los datos de ejecución. Así que el elegido es el de grado 6, ya que este si pasa por todos.

Tree Sort.



Para el caso del algoritmo de ordenamiento tree sort se decidió una función polinómica de grado 8 debido a que en la gráfica de grado 1 esta es una línea completamente recta creciente, por lo tanto, no había una aproximación con los puntos experimentales. Así que, entre mayor era el grado de la función mayor también es la aproximación. Con los puntos experimentales y con las gráficas mayores 8 la aproximación entre los puntos y las gráficas de la función polinómica no variaba demasiado, por lo que se optó por tomar la gráfica de grado 8.

Heap sort



Como se puede ver, la gráfica de grado 6 es la que mejor se aproxima por lo cual se tomara esta gráfica.

Aproximación a la función complejidad.
Con las aproximaciones polinomiales y/o logarítmicas realizadas en la sección anterior, se observará que ecuación modela mejor a F(n).

Función polinomial Burbuja.

$$y = 5792n^2 - 83855n + 208620$$

Función polinomial Burbuja optimizada 1

$$y = 23.067n^2 - 321.38n + 811.35$$

Función polinomial Burbuja optimizada 2

$$y = -0.0052n^6 + 0.2952n^5 - 6.1189n^4 + 54.6n^3 - 191.93n^2 + 206.02n - 20.443$$

Función polinomial Inserción

$$y = 2.3344E - 18n^3 + 0.000000000063n^2 - 0.000002685n + 0.5028$$

Función polinomial Selección.

$$y = 3.1972n^6 - 180.04n^5 + 3874.4n^4 - 39808n^3 + 198732n^2 - 432468n + 292999$$

Función polinomial Shell sort.

$$y = -9E\text{-}07x^6 + 7E\text{-}05n^5 - 0.0018n^4 + 0.0212n^3 - 0.1135n^2 + 0.2563n - 0.1772$$

Función polinomial Tree sort.

$$y=4E\text{-}06n^6-0.0003n^5+0.0094n^4-0.1203n^3+0.7076n^2-1.7225n+1.2451$$

Función polinomial Merge sort.

$$y = 5.6314E\text{-}32n^6-1.4909E\text{-}24n^5+1.3711E\text{-}17n^4-4.9839E\text{-}11n^3+4.1894E\text{-}15n^2+371.1221n-1.8490E6$$

Función polinomial Rápido.

$$y = -0.0833n^6 + 2.7114n^5 - 34.079n^4 + 210.3n^3 - 653.78n^2 + 940.59n - 468.9$$

Función polinomial Heap sort.

$$y = 3E\text{-}05n^6 - 0.0018n^5 + 0.0388n^4 - 0.403n^3 + 2.033n^2 - 4.462n + 3.0411$$

Cálculos a priori a tiempo real para cada Algoritmo.

En esta parte, con base en las aproximaciones seleccionadas (sección anterior), determinaremos cuál será el tiempo real de cada algoritmo para ordenar: 15000000, 20000000, 500000000, 1000000000, 5000000000 números.

Burbuja.

Tamaño del problema (n)	Tiempo real
15000000	$1.3031x10^{18}$
20000000	$2.3167x10^{18}$
500000000	$1.4479x10^{21}$
1000000000	$5.7919x10^{21}$
5000000000	$1.4479x10^{23}$

Burbuja optimizada 1

Tamaño del problema (n)	Tiempo real
15,000,000	5.19007E+15
20,000,000	9.22679E+15
500,000,000	5.76675E+18
1000,000,000	2.3067E+19
5000,000,000	5.76675E+20

Selección.

Tamaño del problema (n)	Tiempo real
15000000	$3.6417x10^{43}$
20000000	$2.0462x10^{44}$
500000000	$4.9956x10^{52}$
1000000000	$3.1971x10^{54}$
5000000000	$4.9956x10^{58}$

Shell sort.

Tamaño del problema (n)	Tiempo real
15000000	1.03E+37
20000000	5.76E+37
500000000	1.41E+46
1000000000	9E+47
5000000000	1.41E+52

Rápido.

Tamaño del problema (n)	Tiempo real
15000000	$-9.4883x10^{41}$
20000000	$-5.3311x10^{42}$
500000000	$-1.3015x10^{51}$
1000000000	$-8.3299x10^{52}$
5000000000	$-1.3015x10^{57}$

Heap sort.

Tamaño del problema (n)	Tiempo real
15000000	3.41717E+38
20000000	1.91999E+39
500000000	4.6875E+47
1000000000	3E+49
5000000000	4.6875E+53

Cuestionario.

1. ¿Cuál de los 8 algoritmos es más fácil de implementar?

R= El algoritmo de burbuja simple, al ser el más conocido y simple, como dice su nombre, lo hace más sencillo de codificar.

2. ¿Cuál de los 8 algoritmos es el más difícil de implementar?

R= Heap Sort

3. ¿Cuál algoritmo tiene menor complejidad temporal?

R= El algoritmo Quick Sort.

4. ¿Cuál algoritmo tiene mayor complejidad temporal?

R= Burbuja simple, es el más lento.

5. ¿Cuál algoritmo tiene menor complejidad espacial? ¿por qué?

R= El de burbuja simple, porque el número de variables que maneja es menor. Tiene un arreglo de tamaño n, y 3 variables auxiliares de tamaño 1. Dicho arreglo se va modificando sin hacer más uso de memoria.

6. ¿Cuál algoritmo tiene mayor complejidad espacial? ¿por qué?

R= El de árbol binario porque al ir creciendo el árbol, se van aumentando nodos, lo que hace que se use más memoria. Además, tiene más variables, apuntadores, etc., para que se ejecute correctamente.

7. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

R= Si, porque fue como el que adelantamos en clase, con la explicación respecto a la velocidad de cada uno de los algoritmos al ordenar cierta cantidad de números. Aquellos que tienen ciclos anidados, tienden a tener tiempos más elevados.

8. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿cuál fue?

R= Si, fue en una computadora con las siguientes características:

- OS: Manjaro Linux x86_64
- CPU: Intel Celeron J1800 (2) @2.582GHz
- GPU: Intel Atom Processor Z36xxx/Z37xxx Series Graphics and Display

9. ¿Facilito las pruebas mediante scripts u otras automatizaciones? ¿cómo lo hizo?

R= Si, primero implementé el código del algoritmo de ordenación en C, luego un script.sh donde dirigí la salida del programa a un archivo llamado sal.txt. Fue más fácil porque no tuve que correr 1 por 1.

10. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

R= Serían 2 principales: no usar la computadora para otras tareas, cuando realicen las mediciones que la máquina sólo haga eso, sino afectará la medida del tiempo y entre antes empiecen a medir los tiempos mejor, porque luego se tarda mucho más de lo que uno podría esperar.

Anexos.

Anexo A. Códigos en ANSI C.

Burbuja Simple.

```
/*
*************************************************************************
    Curso: Análisis de algoritmos
    ESCOM-IPN
    Algoritmo de ordenación Bubble Sort
    Compilación: "gcc main.c tiempo.x  -o main (tiempo.c si se
tiene la implementación de la libreria o tiempo.o si solo se tiene
el codigo objeto)"
    Ejecución: "./main n" (Linux y MAC OS)
    NOTA: Si se hace desde un script.sh, solo se tienen que cambiar
los permisos para que pueda ejecutarse como programa dentro de la
computadora y en la terminal poner "./myscript.sh"

*************************************************************************
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

//PROGRAMA PRINCIPAL
int main(int argc, char **argv){
      //Variables del programa
    double utime0, stime0, wtime0,utime1, stime1, wtime1; //para
medir el tiempo
    int *arr, n, aux, i, j; //arreglo a ordenar, tamaño del
algoritmo, variables para algoritmo

      //Recepción y decodificación de argumentos

      //Si no se introducen exactamente 2 argumentos (Cadena de
ejecución y cadena=n)
      if (argc!=2){
            printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
            exit(1);
      }
      //Tomar el segundo argumento como tamaño del algoritmo
      else{
            n=atoi(argv[1]);
      }

      //Creacion del arreglo
      arr = malloc(sizeof(int)*n);

      //Guardar números en el arreglo
      for(i=0; i<n; i++){
         scanf("%d",&arr[i]);
    }

    //Iniciar conteo para evaluaciones de rendimiento
    uswtime(&utime0, &stime0, &wtime0);

    //Algoritmo
    for(i=0; i<=n-2; i++){ //recorre el arreglo
        for(j=0; j<=n-2; j++){
            if(arr[j]>arr[j+1]){ //por cada posición se compara si
hay uno menor a su derecha
                aux=arr[j]; //si es así, se guarda la posición
actual en una variable auxiliar
                arr[j]=arr[j+1]; //se intercambian
                arr[j+1]=aux; //el auxiliar se le asigna a la
siguiente posición
            }
        }
    }
```

```
    //Evaluar tiempos de ejecución
    uswtime(&utime1, &stime1, &wtime1);

     //Cálculo del tiempo de ejecución del programa
     printf("\n");
     printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10f s\n",
utime1 - utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1
- stime0);
     printf("CPU/Wall    %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
     printf("\n");

     //Mostrar los tiempos en formato exponecial
     printf("\n");
     printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10e s\n",
utime1 - utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",  stime1
- stime0);
     printf("CPU/Wall    %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
     printf("\n");

    //Terminar el programa
    exit(0);
}
```

Burbuja optimizada 1

```
//*****************************************************************
//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
//Curso: Análisis de algoritmos
//(C) Enero 2013
//ESCOM-IPN
//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo
UNIX
//Compilación: "gcc main.c tiempo.x  -o main(teimpo.c si se tiene la
implementación de la libreria o tiempo.o si solo se tiene el codigo
objeto)"
//Ejecución: "./main n" (Linux y MAC OS)
//*****************************************************************



//*****************************************************************
//LIBRERIAS INCLUIDAS
//*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <math.h>
#include "tiempo.h"
//*****************************************************************
//DEFINICION DE CONSTANTES DEL PROGRAMA
//*****************************************************************


//*****************************************************************************
********
//DECLARACION DE ESTRUCTURAS
//*****************************************************************************
********


//*****************************************************************
//DECLARACIÓN DE FUNCIONES
//*****************************************************************


//*****************************************************************
//VARIABLES GLOBALES
//*****************************************************************


//*****************************************************************
//PROGRAMA PRINCIPAL
//*****************************************************************
int main (int argc, char **argv)
{
    //*****************************************************************
    //Variables del main
    //*****************************************************************
    double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
medición de tiempos
    int n;  //n determina el tamaño del algorito dado por argumento al
ejecutar
    int i; //Variables para loops
    int *array,aux,j,k=0,temp=0,b=0;


    //*****************************************************************
    //Recepción y decodificación de argumentos
    //*****************************************************************
```

```
    //Si no se introducen exactamente 2 argumentos (Cadena de ejecución y
cadena=n)
    if (argc!=2)
    {
        printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }
    //Tomar el segundo argumento como tamaño del algoritmo
    else
    {
        n=atoi(argv[1]);
    }

    array =  malloc(sizeof(int)*n);
    //guardamos numeros en el arreglo
    for ( i = 0; i < n; i++)
    {
        scanf("%d",&array[i]);
    }
    //*********************************************************************
    //Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*********************************************************************
    uswtime(&utime0, &stime0, &wtime0);
    //*********************************************************************

    //*********************************************************************
    //Algoritmo
    //*********************************************************************
     for(i=0;i<n;i++){
        for(j=0;j<(n)-i;j++){
            if(array[j]>array[j+1]){
                aux = array[j];
                array[j] = array[j+1];
                array[j+1] = aux;
            }
        }
    }

    // for(i=1;i<=n;i++){
    // printf("el numero de la posicion %d es %d\n",i,array[i]);
    // }


    //*********************************************************************
```

```
    //*****************************************************************
    //Evaluar los tiempos de ejecución
    //*****************************************************************
    uswtime(&utime1, &stime1, &wtime1);

    //Cálculo del tiempo de ejecución del programa
    printf("\n");
    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
    printf("\n");

    //Mostrar los tiempos en formato exponecial
    printf("\n");
    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 -
utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
    printf("\n");
    //*****************************************************************


    //Terminar programa normalmente
    exit (0);
}
```

Burbuja Optimizada 2

```
//Burbuja Optimizada 2
/*
Compilacion del programa: "gcc BurbujaOptimizada2.c tiempo.c -o
"nombre ejecutable"

Ejecucion del programa: "nombre ejecutable " n(numeros a ordenar)
>"nombre del archivo en donde se desee ver el resultado" <
"nombre del archivo que se obtendran los datos a ordenar"
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"


int main (int argc, char* argv[]){


int n,j, aux=0;
double utime0, stime0, wtime0,utime1, stime1, wtime1;
n = atoi(argv[1]);
int *A = (int*) malloc(sizeof(int)*n);
//Ciclo para leer las entradas
for(int k = 0; k< n; k++){
scanf("%d",&A[k]);
}

if (argc!=2){
exit(1);
}


else{
n=atoi(argv[1]);
}
int i=0;
int cambios = 1;
uswtime(&utime0, &stime0, &wtime0);

while(i<(n-1) && cambios==1){
cambios = 0;
for (j=0;j<(n-1)-i; j++){
if (A[j] < A[j+1]){
aux = A[j];
A[j] = A[j+1];
A[j+1] = aux;
cambios = 1;
}
}
i++;
}
uswtime(&utime1, &stime1, &wtime1);


printf("%d----------------------------------------------------------
-------------------------", n);
printf("\n");
printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1
- utime0);
```

```
printf("sys (Tiempo en acciones de E/S)  %.10f s\n",  stime1 -
stime0);
printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1
- stime0) / (wtime1 - wtime0));
printf("\n");

printf("\n");
printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1
- utime0);
printf("sys (Tiempo en acciones de E/S)  %.10e s\n",  stime1 -
stime0);
printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1
- stime0) / (wtime1 - wtime0));
printf("\n");

exit (0);
}
```

Inserción.

```
//************************
//
//Curso: An□lisis de algoritmos
//Marzo 2022
//ESCOM-IPN
//Medicion de los tiempos en los que tarda el algoritmo "Insercion"
para ordenar n numeros
//Compilacion: "gcc Inserción.c tiempo.x -o Inserci□n.out
//(De cada uno de las librerias se pone el ".c" si se tiene la
implementaci□n de la libreria □ ".o" si solo se tiene el codigo
objeto)
//Ejecucion: "./Inserción n" (oLinux y MAC OS)
//************************

//*****************************************************************
**
//Librerias incluidas
//*****************************************************************
**
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"
//*****************************************************************
**
//DEFINICION DE CONSTANTES DEL PROGRAMA
//*****************************************************************
**
```

```
//*****************************************************************
*****************
//DECLARACION DE ESTRUCTURAS
//*****************************************************************
*****************

//*****************************************************************
**
//DECLARACIÓN DE FUNCIONES
//*****************************************************************
**

//*****************************************************************
**
//VARIABLES GLOBALES
//*****************************************************************
**

//*****************************************************************
**
//PROGRAMA PRINCIPAL
//*****************************************************************
**
int main(int argc, char* argv[]){
          //*********************************************************
*************
          //Variables del main
          //*********************************************************
*************
          double utime0, stime0, wtime0,utime1, stime1, wtime1;
//Variables para medición de tiempos
          int n, i, j, temp; // Tama?o de arreglo, auxiliar para
cambio y contadores para los ciclos
          int* A; // Apuntador para el arreglo de elementos

          //*********************************************************
*************
          //Recepción y decodificación de argumentos
          //*********************************************************
*************

          //Si no se introducen exactamente 2 argumentos (Cadena
de ejecución y cadena=n)

          if (argc!=2)
          {
           exit(1);
          }
          //Tomar el segundo argumento como tamaño del algoritmo
          else
```

```
            {
             n=atoi(argv[1]);
            }
            A  =  (int*)malloc(n*sizeof(int));  //  Asignaci?n  de
memoria para el arreglo  de elementos
            for(i=0;i<n;i++){ // Ciclo de lectura de datos
             scanf("%i", &A[i]);
            }
            //************************************************************
*************
            //Iniciar el conteo del tiempo para las evaluaciones de
rendimiento
            //************************************************************
*************
            uswtime(&utime0, &stime0, &wtime0);
            //************************************************************
*************

            //************************************************************
*************
            // Algoritmo de ordenamiento por incersi?n
            //************************************************************
*************
            for(i=0; i<=n-1; i++){
             j=i;
             temp = A[i];
             while((j>0)&&(temp<A[j-1])){
                   A[j]=A[j-1];
                   j--;
             }
             A[j]=temp;
            }
            //************************************************************
*************



            //************************************************************
*************
            //Evaluar los tiempos de ejecución
            //************************************************************
*************
            uswtime(&utime1, &stime1, &wtime1);

            //************************************************************
*************
            // Impresi?n del arreglo ordenado
            //************************************************************
*************
            for(i=0;i<n;i++){
```

```
        printf("%d\n",A[i]);
    }
        //*************************************************************
*************

        free(A);
        //Cálculo del tiempo de ejecución del programa
        printf("\n\n");
        printf("real  (Tiempo  total)   %.10f  s\n",   wtime1  -
wtime0);
        printf("user  (Tiempo  de  procesamiento  en  CPU)  %.10f
s\n",  utime1 - utime0);
        printf("sys (Tiempo en acci??nes de E/S)  %.10f s\n",
stime1 - stime0);
        printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0
+ stime1 - stime0) / (wtime1 - wtime0));
        printf("\n");

        //Mostrar los tiempos en formato exponecial
        printf("\n");
        printf("real  (Tiempo  total)   %.10e  s\n",   wtime1  -
wtime0);
        printf("user  (Tiempo  de  procesamiento  en  CPU)  %.10e
s\n",  utime1 - utime0);
        printf("sys  (Tiempo  en  acciónes  de  E/S)   %.10e  s\n",
stime1 - stime0);
        printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0
+ stime1 - stime0) / (wtime1 - wtime0));
        printf("\n");
        //*************************************************************
*************

        //Terminar programa normalmente
        exit (0);
}

//*****************************************************************
*********
//DEFINICIÓN DE FUNCIONES
//*****************************************************************
*********
```

Selección

```
/*

*****************************************************************
    Curso: Análisis de algoritmos
    ESCOM-IPN
    Algoritmo de ordenación Selection Sort
```

```
    Compilación: "gcc main.c tiempo.x  -o main (tiempo.c si se
tiene la implementación de la libreria o tiempo.o si solo se
tiene el codigo objeto)"
    Ejecución: "./main n" (Linux y MAC OS)
    NOTA: Si se hace desde un script.sh, solo se tienen que
cambiar los permisos para que pueda ejecutarse como programa
dentro de la computadora y en la terminal poner "./myscript.sh"

*************************************************************
*/

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include "tiempo.h"

int main(int argc, char **argv){
    //Variables del programa
    double utime0, stime0, wtime0,utime1, stime1, wtime1; //para
medir el tiempo
    int *A, n, p, k, temp, i=0;  //variables del algoritmo

    //Recepción y decodificación de argumentos

     //Si no se introducen exactamente 2 argumentos (Cadena de
ejecución y cadena=n)
     if (argc!=2){
          printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
          exit(1);
     }
     //Tomar el segundo argumento como tamaño del algoritmo
     else{
          n=atoi(argv[1]);
     }

    //Creacion del arreglo
    A = malloc(sizeof(int)*n);

    //Iniciar conteo para evaluaciones de rendimiento
    uswtime(&utime0, &stime0, &wtime0);

    //Asigna los valores recogidos a una posición del arreglo
    do{
        scanf("%d", &A[i++]);
    }while(i<n);

    //Algoritmo
    for(k=0; k<=n-2;k++){ //recorre el arreglo
        p=k;  //se le asigna una variable p a la posición actual
        for(i=k+1; i<=n-1; i++){ //recorre el arreblo desde un
lugar posterior a la posición actual
            if(A[i]<A[p]) //si la posición posterior a la actual
es menor a la actual
```

```
                p=i; //se le asigna a p el valor más pequeño
        }
        temp=A[p]; //se le asigna a una variable temporal el
valor de p
        A[p]=A[k]; //se intercambia el valor de la posición k a
la posición p
        A[k]=temp; //a la posición k se le asigna el valor de la
variable temporal
    }

    //Evaluar tiempos de ejecución
    uswtime(&utime1, &stime1, &wtime1);

     //Cálculo del tiempo de ejecución del programa
     printf("\n");
     printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10f s\n",
utime1 - utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",
stime1 - stime0);
     printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
     printf("\n");

     //Mostrar los tiempos en formato exponecial
     printf("\n");
     printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10e s\n",
utime1 - utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",
stime1 - stime0);
     printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
     printf("\n");

    //Terminar el programa
    exit(0);
}
```

Shell sort

```
//*************************************************************
//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
//Curso: Análisis de algoritmos
//(C) Enero 2013
//ESCOM-IPN
//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo
UNIX
//Compilación: "gcc main.c tiempo.x  -o main(teimpo.c si se tiene la
implementación de la libreria o tiempo.o si solo se tiene el codigo
objeto)"
//Ejecución: "./main n" (Linux y MAC OS)
//*************************************************************
```

```
//*****************************************************************
//LIBRERIAS INCLUIDAS
//*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "tiempo.h"
//*****************************************************************
//DEFINICION DE CONSTANTES DEL PROGRAMA
//*****************************************************************


//*****************************************************************
********
//DECLARACION DE ESTRUCTURAS
//*****************************************************************
********


//*****************************************************************
//DECLARACIÓN DE FUNCIONES
//*****************************************************************


//*****************************************************************
//VARIABLES GLOBALES
//*****************************************************************


//*****************************************************************
//PROGRAMA PRINCIPAL
//*****************************************************************
int main (int argc, char **argv)
{
    //*****************************************************************
    //Variables del main
    //*****************************************************************
    double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
medición de tiempos
    int n;  //n determina el tamaño del algorito dado por argumento al
ejecutar
    int i; //Variables para loops
```

```
    int *array,aux,j,k=0,temp=0,b=0;


    //*****************************************************************
    //Recepción y decodificación de argumentos
    //*****************************************************************


    //Si no se introducen exactamente 2 argumentos (Cadena de ejecución y
cadena=n)
    if (argc!=2)
    {
        printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }
    //Tomar el segundo argumento como tamaño del algoritmo
    else
    {
        n=atoi(argv[1]);
    }

    array =  malloc(sizeof(int)*n);
    //guardamos numeros en el arreglo
    for ( i = 0; i < n; i++)
    {
        scanf("%d",&array[i]);
    }
    //*****************************************************************
    //Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*****************************************************************
    uswtime(&utime0, &stime0, &wtime0);
    //*****************************************************************

    //*****************************************************************
    //Algoritmo
    //*****************************************************************
     k=trunc(n/2);
    while(k>=1){
        b=1;
        while(b!=0){
            b=0;
            for(i=k;i<=n;i++){
                if(array[i-k]>array[i]){
                    temp=array[i];
```

```
                    array[i]=array[i-k];
                    array[i-k]=temp;
                    b=b+1;
                }
            }
        }
        k=trunc(k/2);
    }
     /*for(i=1;i<=n;i++){
    //printf("el numero de la posicion %d es %d\n",i,array[i]);
    }*/


    //*****************************************************************


    //*****************************************************************
    //Evaluar los tiempos de ejecución
    //*****************************************************************
    uswtime(&utime1, &stime1, &wtime1);

    //Cálculo del tiempo de ejecución del programa
    printf("\n");
    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
    printf("\n");

    //Mostrar los tiempos en formato exponecial
    printf("\n");
    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 -
utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
    printf("\n");
    //*****************************************************************
```

```
    //Terminar programa normalmente
    exit (0);
}
```

Rápido.

```
/*
*********************************************************************
    Curso: Análisis de algoritmos
    ESCOM-IPN
    Algoritmo de ordenación Quick Sort
    Compilación: "gcc main.c tiempo.x  -o main (tiempo.c si se
tiene la implementación de la libreria o tiempo.o si solo se
tiene el codigo objeto)"
    Ejecución: "./main n" (Linux y MAC OS)
    NOTA: Si se hace desde un script.sh, solo se tienen que
cambiar los permisos para que pueda ejecutarse como programa
dentro de la computadora y en la terminal poner "./myscript.sh"

*********************************************************************
*/

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
#include "tiempo.h"

void intercambiar(int elem1, int elem2);
int particionar(int *A, int inicio, int final);
void quickSort(int *A, int inicio, int final);
void mostrar(int *A, int n);

int main(int argc, char **argv){
    //Variables del programa
    double utime0, stime0, wtime0,utime1, stime1, wtime1; //para
medir el tiempo
    int *mi_A, p=0, r=0, n, i=0; //para el algoritmo

    //Recepción y decodificación de argumentos

     //Si no se introducen exactamente 2 argumentos (Cadena de
ejecución y cadena=n)
     if (argc!=2){
          printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
          exit(1);
     }
     //Tomar el segundo argumento como tamaño del algoritmo
     else{
          n=atoi(argv[1]);
```

```
      }

    //Iniciar conteo para evaluaciones de rendimiento
    uswtime(&utime0, &stime0, &wtime0);

    //Arreglo en la memoria
    mi_A = malloc(sizeof(int)*n);
    //Asigna los valores recogidos a una posición del arreglo
    do{
        scanf("%d", &mi_A[i++]);
    }while(i<n);

    //Algoritmo
    quickSort(mi_A,0,n-1);
    //mostrar(mi_A, n);

    //Evaluar tiempos de ejecución
    uswtime(&utime1, &stime1, &wtime1);

      //Cálculo del tiempo de ejecución del programa
      printf("\n");
      printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
      printf("user (Tiempo de procesamiento en CPU) %.10f s\n",
utime1 - utime0);
      printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",
stime1 - stime0);
      printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
      printf("\n");

      //Mostrar los tiempos en formato exponecial
      printf("\n");
      printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
      printf("user (Tiempo de procesamiento en CPU) %.10e s\n",
utime1 - utime0);
      printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",
stime1 - stime0);
      printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
      printf("\n");

    //Terminar el programa
    exit(0);
}

//hace todo el ordenamiento
void quickSort(int *A, int inicio, int final){
    if(inicio<final){
        int pi;
        pi= particionar(A, inicio, final); //se selecciona el
pivote
        quickSort(A,inicio,pi-1); //antes del pivote
        quickSort(A,pi+1, final);  //después del pivote
    }
```

```
}

/*
    Toma el último elemento como el pivote y pone los menores a
él a su izq y los más grandes a su derecha
*/
int particionar(int *A, int inicio, int final){
    int pivote;
    pivote=A[final];

    int i = inicio -1; //índice del elemento más pequeño e indica
la posición correcta del pivote encontrado

    for(int j=inicio ; j<= final-1; j++){
        //si el elemento actual es más chico que el pivote
        if(A[j]<pivote){
            i++; //se incrementa el index del elemento más
peuqeño
            intercambiar(A[i], A[j]); //se intercambian
        }
    }
    intercambiar(A[i+1], A[final]);
    return (i+1);
}

//cambia 2 elementos
void intercambiar(int elem1, int elem2){
    int temp;
    temp=elem1;
    elem2=elem1;
    elem1=temp;
}

/*
void mostrar(int *A, int n){
    for(int i=0; i<n; i++){
        printf("%d \n", &A[i]);
    }
}*/
```

Tree Sort.

```
// Arbol Binario de Busqueda y recorrido InOrden

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "arbol.h"
#include "tiempo.h"

int posicion = 0;

struct arbol *crearArbol(struct arbol *a)
```

```
{
    a = NULL;
    return a;
}



struct arbol *insertarNumeros(struct arbol *a, int num)
{
    if (a == NULL)
    {
        struct arbol *aux = NULL;
        aux = (struct arbol *)malloc(sizeof(struct arbol));
        aux->nodo_izq = NULL;
        aux->nodo_der = NULL;
        aux->numero = num;

        return aux;
    }

    if (num < (a->numero))
        a->nodo_izq = insertarNumeros(a->nodo_izq, num);
    else
        a->nodo_der = insertarNumeros(a->nodo_der, num);

    return a;
}


void guardarRecorridoInOrden(struct arbol *a, int *arreglo)
{
    if (a != NULL)
    {
        guardarRecorridoInOrden(a->nodo_izq, arreglo);
        *(arreglo + posicion) = a->numero;
        posicion++;
        guardarRecorridoInOrden(a->nodo_der, arreglo);
    }
}



int main(int narg, char **varg)
{
    int n, *numeros;
    double utime0,stime0,wtime0,utime1,stime1,wtime1;



    struct arbol *a;
    a = crearArbol(a);
```

```
    if (narg != 2)
    {
        exit(1);
    }

    n = atoi(varg[1]);

    numeros = malloc(sizeof(int) * (n));
    if (numeros == NULL)
    {
        exit(1);
    }


    for (int i = 0; i < n; i++)
        scanf("%d", &numeros[i]);


    uswtime(&utime0,&stime0,&wtime0);


    for (int j = 0; j < n; j++)
        a = insertarNumeros(a, numeros[j]);

    guardarRecorridoInOrden(a, numeros);
    uswtime(&utime1, &stime1, &wtime1);


printf("%d>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>", n);

    printf("\n");
    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",
utime1 - utime0);
    printf("sys (Tiempo en acciÃƒÂ³nes de E/S)  %.10f s\n",
stime1 - stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
    printf("\n");


    printf("\n");
    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",
utime1 - utime0);
    printf("sys (Tiempo en acciÃƒÂ³nes de E/S)  %.10e s\n",
stime1 - stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
    printf("\n\n");
```

```
exit (0);
}
```

Árbol.h :

```
struct arbol
{
    int numero; //Dato
    struct arbol *nodo_izq; //Hijo izquierdo
    struct arbol *nodo_der; //Hijo Derecho
};

//Prototipos
struct arbol *crearArbol(struct arbol *a);
struct arbol *insertarNumeros(struct arbol *a, int num);
void guardarRecorridoInOrden(struct arbol *a, int *arreglo);
```

Merge Sort.

```
#include <stdio.h>
#include <stdlib.h>

void mergeSort(int *arreglo, int elementos){
    int i;
    int mitad, restante;
    int indiceIzquierda, indiceDerecha;
    int *copiaIzquierda, *copiaDerecha;

    /*Si ya sólo es 1 elemento, ya está ordenado el arreglo*/
    if(elementos == 1){
        return;
    }

    /*
    Si hay más de 1 elemento, hay que ordenarlo :
    Hay que partir el arreglo en 2 partes.
    Uno que contenga la primer mitad de elementos.
    Y otra que contenga la segunda mitad de elementos.
    Por eso calculamos <<mitad>> para saber cuántos habrá en la
mitad izquierda
    y <<restante>> es lo que habrá en la derecha. Esto porque si
n es impar
    pueden ser diferentes estos números.
    */
    mitad = elementos / 2;
    restante = elementos - mitad;
    copiaIzquierda = malloc(mitad * sizeof(int));
    copiaDerecha = malloc(restante * sizeof(int));

    /* Copiamos los elementos del original a sus respectivas
mitades */
```

```
    for(i = 0; i < mitad; i ++){
        copiaIzquierda[i] = arreglo[i];
    }

    for(i = 0; i < restante; i ++){
        copiaDerecha[i] = arreglo[mitad + i];
    }

    /*
    Ordenamos recursivamente. Es decir, después de llamar
merge(), cada mitad
    estará ordenada
    */
    mergeSort(copiaIzquierda, mitad);
    mergeSort(copiaDerecha, restante);

    /*
    Ahora hay que unir las 2 mitades ya ordenadas para que el
completo esté
    ordenado también
    */

    indiceIzquierda = 0;
    indiceDerecha = 0;

    i = 0;
    /*
    En cada momento hay que preguntar, ¿Cuál va primero, el de la
izquierda o el de la derecha?
    Y hay que tomar el más pequeño. Así sabemos que al final,
<<arreglo>> esta ordenado.
    ¿ Por qué? Imaginemos que la mitad izquierda se ve así :
    a < b < c < d < ... porque está ordenado
    Y luego la derecha:
    A < B < C < D < ...
    Así, supongamos si estamos comparando un elemento s de la
izquierda y un S de la derecha, sabemos que
    si el arreglo tiene elementos
    w1 < w2 < w3 < w4 < ... < wk, entonces w1, w2, w3, w4, ...,
wk < s y que w1, w2, w3, w4, ..., wk < S
    porque hemos puesto valores menores a s y S ya que las
mitades estaban ordenadas anteriormente
    entonces si ponemos a s y S en orden entonces el arreglo
estará ordenado.
    */
    while(indiceIzquierda < mitad && indiceDerecha < restante){
        if(copiaIzquierda[indiceIzquierda] <
copiaDerecha[indiceDerecha]){
            arreglo[i] = copiaIzquierda[indiceIzquierda];
            indiceIzquierda ++;
        } else {
            arreglo[i] = copiaDerecha[indiceDerecha];
            indiceDerecha ++;
        }
```

```
        i ++;
    }

    /* Ahora, si ya se acabó la mitad derecha, hay que terminar
de pasar la mitad izquierda */
    while(indiceIzquierda < mitad){
        arreglo[i] = copiaIzquierda[indiceIzquierda];
        indiceIzquierda ++;
        i ++;
    }

    /* Y si se acabó la mitad izquierda, hay que vaciar la mitad
derecha */
    while(indiceDerecha < restante){
        arreglo[i] = copiaDerecha[indiceDerecha];
        indiceDerecha ++;
        i ++;
    }
}

int main(void){
    int n ;
    int i;
    int *arreglo;

    /* leer n */
    scanf("%d", &n);

    /* crear un arreglo de tamaño n */
    arreglo = malloc(n * sizeof(int));

    /* leer los n elementos */
    for(i = 0; i < n; i++){
        scanf("%d", &arreglo[i]);
    }

    /* ordenarlos */
    mergeSort(arreglo, n);

    /* imprimirlos */
    for(i = 0; i < n; i ++){
        printf("%d ", arreglo[i]);
    }
    printf("\n");
}
```

Heap sort

```
//*****************************************************************
*
//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
//Curso: Análisis de algoritmos
```

```
//(C) Enero 2013
//ESCOM-IPN
//Ejemplo de medición de tiempo en C y recepción de parametros en
C bajo UNIX
//Compilación: "gcc main.c tiempo.x  -o main(teimpo.c si se tiene
la implementación de la libreria o tiempo.o si solo se tiene el
codigo objeto)"
//Ejecución: "./main n" (Linux y MAC OS)
//**************************************************************
*



//**************************************************************
*
//LIBRERIAS INCLUIDAS
//**************************************************************
*
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "tiempo.h"
//**************************************************************
*
//DEFINICION DE CONSTANTES DEL PROGRAMA
//**************************************************************
*



//**************************************************************
****************
//DECLARACION DE ESTRUCTURAS
//**************************************************************
****************



//**************************************************************
*
//DECLARACIÓN DE FUNCIONES
//**************************************************************
*


void swap(int* a, int* b){
    int t = *a;
```

```
    *a = *b;
    *b = t;
}


void heapify(int A[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && A[l] > A[largest])
        largest = l;

    if (r < n && A[r] > A[largest])
        largest = r;

    if (largest != i) {
        swap(&A[i], &A[largest]);
        heapify(A, n, largest);
    }
}


void heapSort(int A[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(A, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(&A[0], &A[i]);
        heapify(A, i, 0);
    }
}
//*****************************************************************
*
//VARIABLES GLOBALES
//*****************************************************************
*



//*****************************************************************
*
//PROGRAMA PRINCIPAL
```

```
//*****************************************************************
*
int main (int argc, char **argv)
{

//*****************************************************************
**
    //Variables del main

//*****************************************************************
**
    double utime0, stime0, wtime0,utime1, stime1, wtime1;
//Variables para medición de tiempos
    int n;  //n determina el tamaño del algorito dado por
argumento al ejecutar
    int i; //Variables para loops
    int *array,aux,j,k=0,b=0,temp=0;



//*****************************************************************
**
    //Recepción y decodificación de argumentos

//*****************************************************************
**


    //Si no se introducen exactamente 2 argumentos (Cadena de
ejecución y cadena=n)
    if (argc!=2)
    {
        printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }
    //Tomar el segundo argumento como tamaño del algoritmo
    else
    {
        n=atoi(argv[1]);
    }

    array =  malloc(sizeof(int)*n);
    //guardamos numeros en el arreglo
    for ( i = 0; i < n; i++)
```

```
    {
        scanf("%d",&array[i]);
    }

//**********************************************************************
**
    //Iniciar el conteo del tiempo para las evaluaciones de
rendimiento

//**********************************************************************
**
    uswtime(&utime0, &stime0, &wtime0);

//**********************************************************************
**



//**********************************************************************
**
    //Algoritmo

//**********************************************************************
**
     //-------------- ALgortimo ---------------
    heapSort(array,n);
    //Evaluar los tiempos de ejecución

//**********************************************************************
**
    uswtime(&utime1, &stime1, &wtime1);

    //Cálculo del tiempo de ejecución del programa
    printf("\n");
    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",
utime1 -    utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
    printf("\n");

    //Mostrar los tiempos en formato exponecial
    printf("\n");
    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
```

```
    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",
utime1 - utime0);
    printf("sys (Tiempo en acciónes de E/S)  %.10e s\n",  stime1 -
stime0);
    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 +
stime1 - stime0) / (wtime1 - wtime0));
    printf("\n");

//*****************************************************************
**


    //Terminar programa normalmente
    exit (0);
}
```