Análisis de Algoritmos

Práctica 01: "Pruebas a posteriori"

Nombres:

Martínez Partida Jair Fabian

Martínez Rodríguez Alejandro

Monteros Cervantes Miguel Angel

Luis Fernando Ramírez Cotonieto

Fecha de entrega: 1 de Abril del 2021

Grupo: 3CM13

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice

Práctica 01: Pruebas a posteriori

Análisis de Algoritmos

1. Planteamiento del problema

El propósito del análisis de un problema es ayudar al programador a llegar a una cierta comprensión de la naturaleza del problema. Una buena definición del problema, junto con una descripción detallada de las especificaciones de entrada/salida, son los requisitos más importantes para llegar a una solución eficaz.
Para medir la eficiencia de los algoritmos y poder decidir cual es más conveniente según que caso, existen técnicas o enfoques como el empírico o a posteriori (tratado en esta práctica), consiste en programar técnicas competidoras e ir probándolas en distintos casos con ayuda de una computadora. Este análisis nos dará como resultado una función que será llamada función de complejidad, la cual nos permitirá calcular el costo de ejecutar el algoritmo según la entrada que se le proporciona, este valor llega a ser subjetivo debido a que depende de otro parámetros como lo son el hardware y otros aspectos sobre todo físicos.
La ordenación es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cómputos que tiene que realizar un programa son más eficientes si los datos sobre los que operan están ordenados. Una de los procedimientos más usados en la computación es el ordenamiento de arreglos. Desde los inicios de la computación, las computadoras no eran capaces de ordenar arreglos grandes debido a su complejidad y poco poder de procesamiento, es por eso que fue necesaria la invención de métodos de ordenamiento más eficaces. Hoy en día, incluso con los grandes avances en tecnología de hardware, resulta necesario mantener esa eficacia, pues en la actualidad, las computadoras deben de realizar dicho proceso repetidas veces con cantidades aún más grandes de información.
Para este problema, se proporcionan 10,000,000 de números diferentes, y se establecen los siguiente métodos de ordenamiento que nos ayudaran al análisis de su complejidad.

- Burbuja (Bubble Sort)
  - Burbuja Simple
  - Burbuja Optimizada
- Inserción (Insertion Sort)
- Selección (Selection Sort)
- Shell (Shell Sort)
- Ordenamiento con árbol binario de búsqueda (Tree Sort)

2. Actividades y Pruebas

2.1. Ordenamiento de burbuja

2.1.1. Burbuja simple

Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiandolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

- El método de la burbuja es uno de los mas simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.
- Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo.

```
Procedimiento 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 Procedimiento
```

Figura 1: Burbuja simple

2.1.2. Burbuja optimizada

Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración, además pude existir la posibilidad que realizar iteraciones de más si el arreglo ya fue ordenado totalmente.

```
Procedimiento 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]) hacer
                            aux = A[j]
                            A[j] = A[j+1]
                            A[j+1] = aux
                            cambios = "Si"
                     FinSi
              FinPara
               i= i+1
       FinMientras
fin Procedimiento
```

Figura 2: Burbuja optimizada

2.2. Ordenamiento por Inserción

Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.
Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y éste es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

Ordenamiento por Selección

Se basa en buscar el mínimo elemento de la lista e intercambiarlo con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo y así sucesivamente.

```
Procedimiento 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 Procedimiento
```

Figura 3: Ordenamiento por inserción

```
Procedimiento 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 Procedimiento
```

Figura 4: Ordenamiento por selección

Ordenamiento con un Árbol Binario de Búsqueda (ABB)

El ordenamiento con la ayuda de un árbol binario de búsqueda es muy simple debido a que solo requiere de dos pasos simples.
1. Insertar cada uno de los números del vector a ordenar en el árbol binario de búsqueda.
2. Reemplazar el vector en desorden por el vector resultante de un recorrido InOrden del Árbol Binario, el cual entregará los números ordenados.
La eficiencia de este algoritmo está dada según la eficiencia en la implementación del árbol binario de búsqueda, lo que puede resultar mejor que otros algoritmos de ordenamiento.

```
Procedimiento OrdenaConArbolBinarioBusqueda(A,n)

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

      GuardarRecorridoInOrden(ArbolBinBusqueda,A);

fin Procedimiento
```

Figura 5: Ordenamiento con un Árbol Binario de Búsqueda (ABB)

2.3. Ordenamiento Shell

El Shell es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:
1. El ordenamiento por inserción es eficiente si la entrada está çasi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.
El algoritmo Shell mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes"hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del ordenamiento Shell es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.
Shell propone que se haga sobre el arreglo una serie de ordenaciones basadas en la inserción directa, pero dividiendo el arreglo original en varios sub-arreglos tales que cada elemento esté separado k elementos del anterior (a esta separación a menudo se le llama salto o gap)
Se debe empezar con k=n/2, siendo n el número de elementos del arreglo, y utilizando siempre la división entera

(TRUNC)
Después iremos variando k haciéndolo más pequeño mediante sucesivas divisiones por 2, hasta llegar a k=1.

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

Figura 6: Ordenamiento Shell

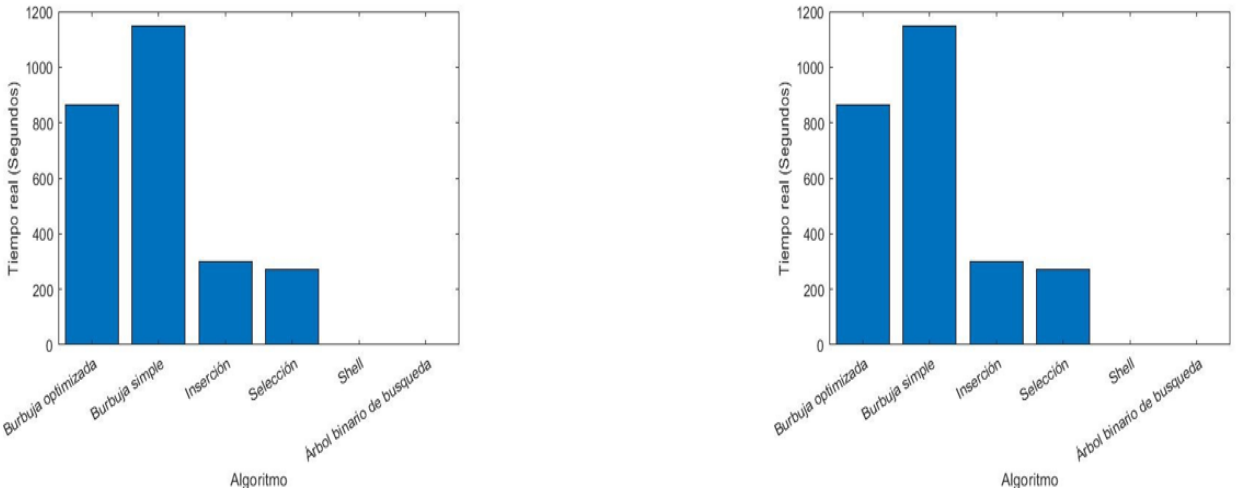
3. Medición con n = 500,000

Para cada algoritmo se hizo una prueba con n=500,000 y los resultados obtenidos se presentan en la siguiente tabla.

Algoritmo	Tiempo real	Tiempo CPU	Tiempo E/S	%CPU/Wall
Árbol binario de busqueda	0.4162189960 s	0.4117910000 s	0.0040060000 s	99.8986120163
Burbuja	1150.1160268784 s	1149.8105470000 s	0.0119940000 s	99.9744820634
Burbuja optimizada	864.4807510376 s	864.1368760000 s	0.0679810000 s	99.9680855777
Inserción	300.5317440033 s	300.3154710000 s	0.1076180000 s	99.9638457483
Selección	270.0255920887 s	270.0667530000 s	0.0040000000 s	100.0167246782
Shell	0.6290349960 s	0.6264240000 s	0.0001150000 s	99.6032023578

Cuadro 1: Medición con n=500,000

Para representarlo se una gráfica, se probó poniendo el eje “y” en segundos, milisegundos, microsegundos y nanosegundos y el resultado es el mismo, los que menos tardan no se aprecian.



(a) Variación 1

(b) Variación 2

Figura 7: Comparación de algoritmos.

4. Comportamiento temporal de cada algoritmo

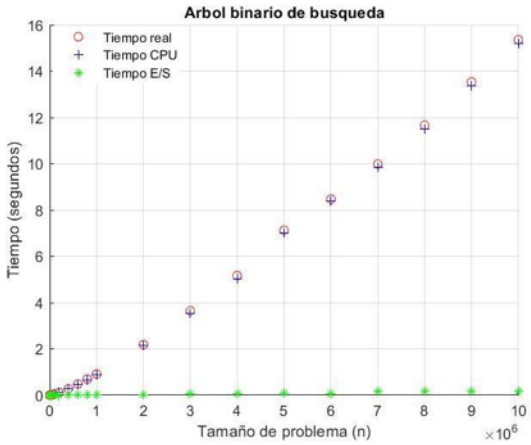


Figura 8: Árbol binario de busqueda

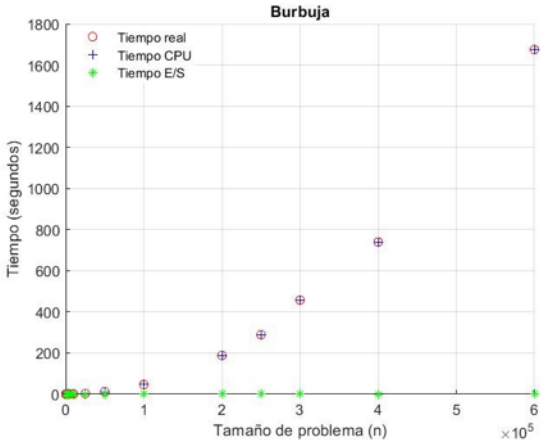


Figura 9: Burbuja simple

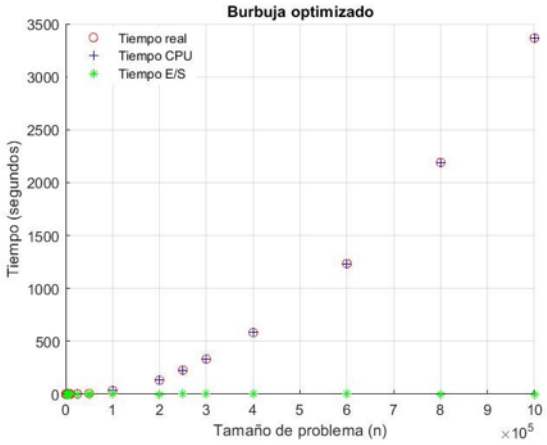


Figura 10: Burbuja optimizada

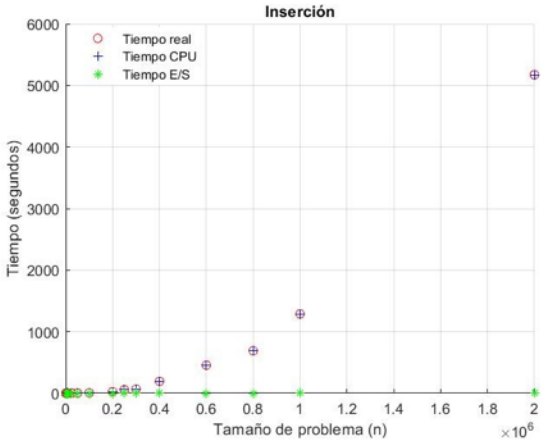


Figura 11: Inserción

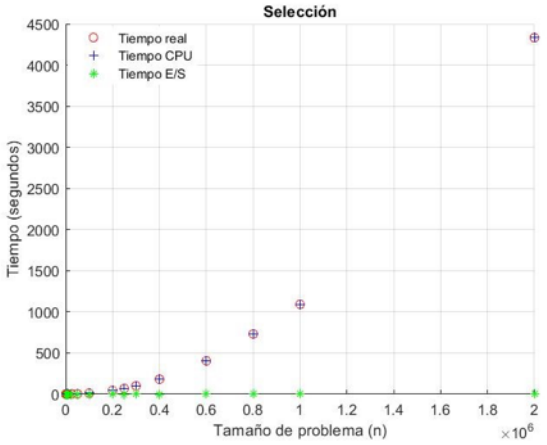


Figura 12: Selección

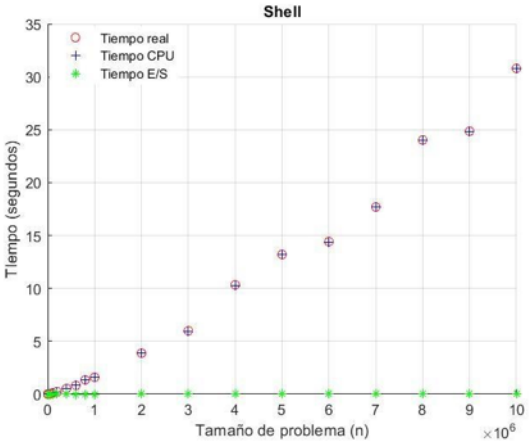


Figura 13: Shell

5. Comparativa de algoritmos en tiempo real

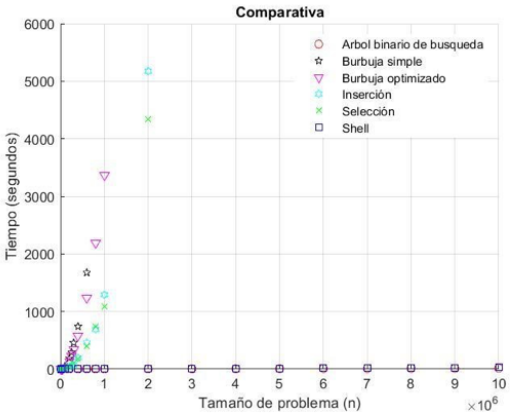


Figura 14: Comparativa de tiempos entre algoritmos

6. Aproximación polinomial

6.1. Árbol binario de búsqueda

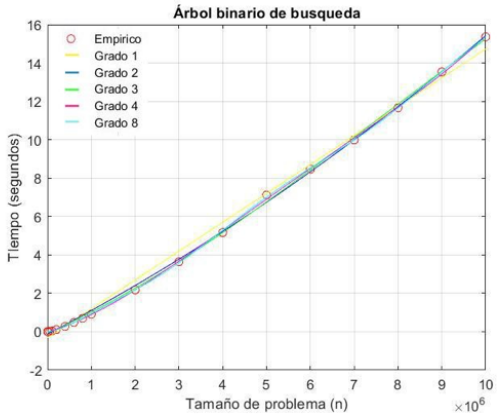


Figura 15: Aproximación polinomial de árbol binario de búsqueda

Para este algoritmo, se toma el grado 2 como la opción más viable ya que, aunque no pase exactamente por los mismos puntos obtenidos en la prueba empírica, se aproxima bastante. También coincide con que sea el algoritmo más rápido ya que su función no se dispara tanto de un momento a otro.

6.2. Burbuja simple

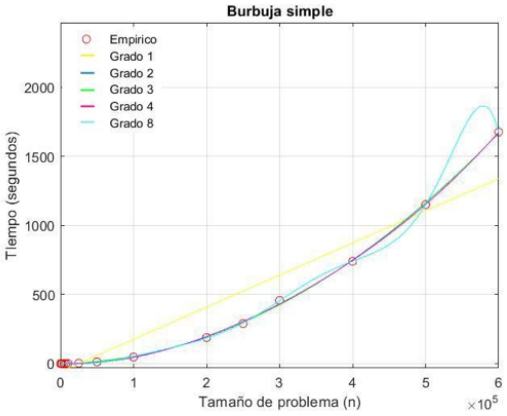


Figura 16: Aproximación polinomial de burbuja simple

Para este algoritmo,el grado 2 lo define mejor, ya que pasa por todos los puntos y al probar la función en una graficadora, se tiene una aproximación muy cercana.

6.3. Burbuja optimizada

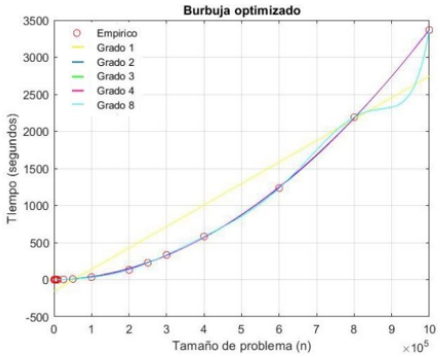


Figura 17: Aproximación polinomial de burbuja optimizada

Para este algoritmo,el grado 2 lo define mejor, ya que se tiene una buena precisión, se le considera la mejor opción.

6.4. Inserción

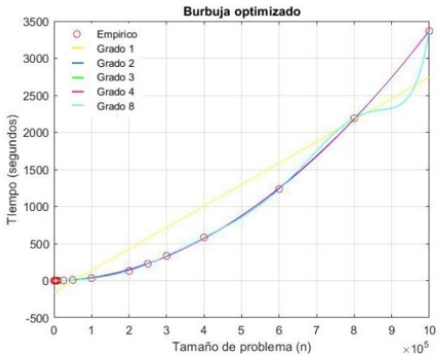


Figura 18: Aproximación polinomial de inserción

Para este algoritmo,el grado 2 tiene una precisión cercana al 100 por ciento en los puntos evaluados y la forma en la que crece el tiempo en función del tamaño de problema.

6.5. Selección

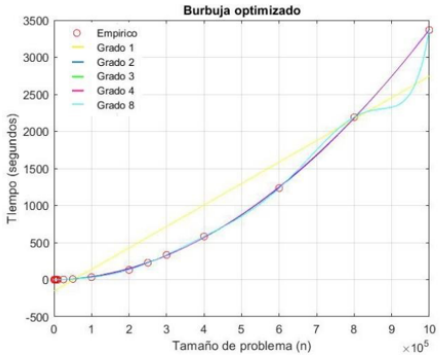


Figura 19: Aproximación polinomial de selección

Para este algoritmo,el grado 2 coincide el dato calculado con el dato empírico en todos los casos tomados antes y después de hacer la gráfica.

6.6. Shell

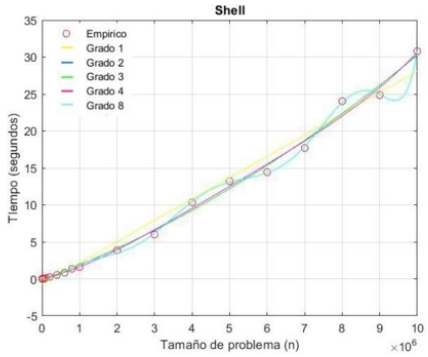


Figura 20: Aproximación polinomial de selección

Para este algoritmo,el grado 2 desde el punto empírico da más coincidencia.

7. Comparativa: Mejores funciones

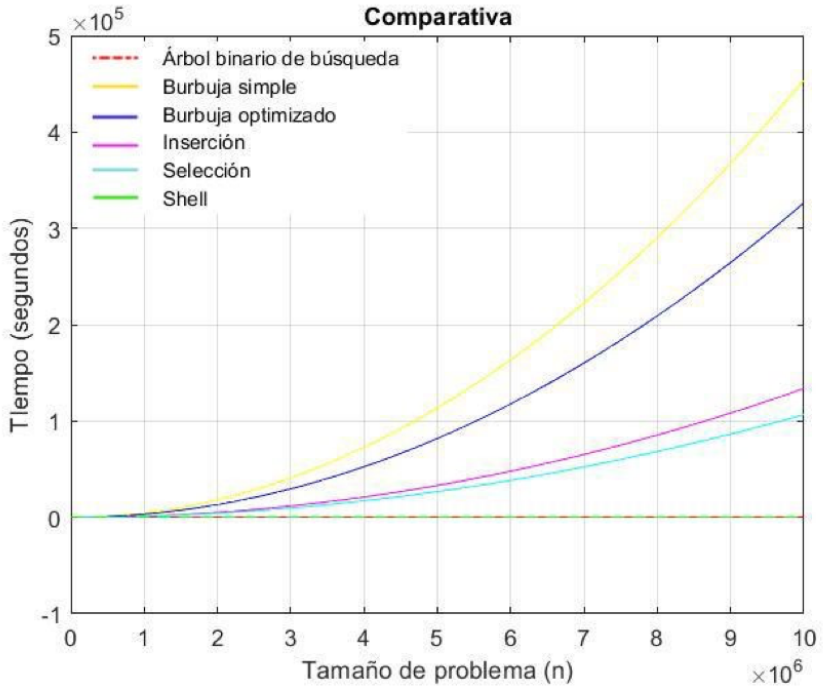


Figura 21: Comparativa para mejores funciones

Gracias a las heramientas de MATLAB se pudo estimar los tiempos que ocuparía cada algoritmo para ordenar las cantidades de números dados, los resultados son los siguientes.

	Árbol binario de búsqueda	Burbuja simple	Burbuja optimizado	Inserción	Selección	Shell
			(s)			
n=15000000	2.591258e+01	1.020569e+06	7.336978e+05	3.019450e+05	2.398539e+05	5.408704e+01
n=20000000	3.821784e+01	1.813935e+06	1.303522e+06	5.375288e+05	4.261376e+05	8.363096e+01
n=50000000	9.663860e+03	1.132970e+09	8.132056e+08	3.372843e+08	2.658501e+08	2.962331e+04
n=1000000000	3.746431e+04	4.531820e+09	3.252698e+09	1.349248e+09	1.063360e+09	1.165996e+05
n=5000000000	9.127805e+05	1.132943e+11	8.131495e+10	3.373341e+10	2.658319e+10	2.877110e+06

Cuadro 2: Comparativa: Mejores funciones

8. Cuestionario

1-¿Cuál de los 5 algoritmos es más fácil de implementar?
Burbuja simple.

2-¿Cuál de los 5 algoritmos es el más difícil de implementar?
Árbol binario de búsqueda.

3- ¿Cuál algoritmo tiene menor complejidad temporal?
Árbol binario de búsqueda.

4-¿Cuál algoritmo tiene mayor complejidad temporal?
Burbuja simple.

5-¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?
Podrían ser los burbuja porque usa un arreglo de tamaño n, y 3 variables aparte que sirven para ciclos y auxiliar

6-¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?
El árbol binario de búsqueda porque crea n nodos que son más grandes que los otros tipos de datos simples.

7-¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?
Si, porque desde estructuras de datos vimos parte de este tema, y desde entonces sabíamos que algo exponencial crecía muchísimo y con ciertas técnicas se puede bajar esa complejidad.

8-¿Existió un entorno controlado para realizar las pruebas? ¿Cuál fue?
Si, se realizó en una maquina virtual de una computadora con 16 Ram, Ryzen 2600 y Gpu Radeon 5500xt.

9-¿Facilito las pruebas mediante scripts u otras automatizaciones? ¿Cómo lo hizo?
Sí, mediante el script compilamos cada algoritmo, lo corríamos dando como entrada el archivo de números y la salida iba a archivos para tener las mediciones.

10- ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?
Tener pacieicnia y la computadora cerca de un contacto de la luz, porque todos los procesos serán MUY largos.

9. Anexos

Las pruebas se llevaron en una máquina virtual con ubuntu 20.04.1 en un equipo con 16 Ram, Ryzen 2600 y Gpu Radeon 5500xt. Podemos compilarlo y hacer que se lleve el proceso mediante el comando: ./script.sh

9.1. Ejemplo de script

```
#!/bin/bash
gcc 'shell.c' -o 'shell'

./shell 100 <numeros10millones.txt >shell.txt
./shell 1000 <numeros10millones.txt >>shell.txt
./shell 5000 <numeros10millones.txt >>shell.txt
./shell 10000 <numeros10millones.txt >>shell.txt
./shell 50000 <numeros10millones.txt >>shell.txt
./shell 100000 <numeros10millones.txt >>shell.txt
./shell 200000 <numeros10millones.txt >>shell.txt
./shell 400000 <numeros10millones.txt >>shell.txt
./shell 600000 <numeros10millones.txt >>shell.txt
./shell 800000 <numeros10millones.txt >>shell.txt
./shell 1000000 <numeros10millones.txt >>shell.txt
./shell 2000000 <numeros10millones.txt >>shell.txt
./shell 3000000 <numeros10millones.txt >>shell.txt
./shell 4000000 <numeros10millones.txt >>shell.txt
./shell 5000000 <numeros10millones.txt >>shell.txt
./shell 6000000 <numeros10millones.txt >>shell.txt
./shell 7000000 <numeros10millones.txt >>shell.txt
./shell 8000000 <numeros10millones.txt >>shell.txt
./shell 9000000 <numeros10millones.txt >>shell.txt
./shell 10000000 <numeros10millones.txt >>shell.txt
```

9.2. ABB

```
//Practica 1
//Arbol binario de busqueda, recorrido inorder

#include<stdio.h>
#include<stdlib.h>
#include <sys/resource.h>
#include <sys/time.h>

struct Arbol
{
  int dato;
  struct Arbol *izq;
  struct Arbol *der;
};

void uswtime(double *usertime, double *systime, double *walltime);
struct Arbol *AgregarElemento(struct Arbol *,int);
void InOrden(struct Arbol *,int *);
void ImprimirTiempos(double,double,double,double,double,double);
int count = 0;

int main(int argc, char const *argv[])
{
  struct Arbol *arbol = NULL;
  int num, n, i;
  int *numeros = (int*) malloc(sizeof(int)*10000000);
  double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
      medici n de tiempos
```

```
28
29    //Numero de numeros que se ordenar n
30    n=atoi(argv[1]);
31
32    for(i = 0; i < n + 1; i++){
33      scanf("%d",&num);
34      arbol = AgregarElemento(arbol,num);
35    }
36
37    printf("\tArbol Binario de Busqueda con %d numeros\n",n);
38
39    uswtime(&utime0, &stime0, &wtime0);
40
41    InOrden(arbol,numeros);
42
43    //for (i = 0; i < 10000000; ++i) printf("%d. %d\n",i+1,*(numeros + i));
44
45    uswtime(&utime1, &stime1, &wtime1);
46    ImprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
47    printf("
          ##########################################################################################
          n");
48
49    return 0;
50  }
51
52  struct Arbol *AgregarElemento(struct Arbol *raiz, int dato)
53  {
54    if(raiz == NULL) // Caso base
55    {
56      struct Arbol *nuevo = NULL;
57      nuevo = (struct Arbol *) malloc(sizeof(struct Arbol));
58      nuevo -> dato = dato;
59      nuevo -> izq = NULL;
60      nuevo -> der = NULL;
61      return nuevo;
62    }
63
64    if(dato < raiz -> dato){
65      raiz -> izq = AgregarElemento(raiz -> izq, dato);
66    }
67    else
68    {
69      raiz -> der = AgregarElemento(raiz -> der, dato);
70    }
71
72    return raiz;
73  }
74
75  void InOrden(struct Arbol *arbol, int *numeros)
76  {
77    if(arbol == NULL)
78      return;
79
80    InOrden(arbol -> izq,numeros);
81    *(numeros + count) = arbol -> dato;
82    count++;
83    //printf("%d\n",arbol -> dato);
84    InOrden(arbol -> der,numeros);
85  }
86
```

```c
void uswtime(double *usertime, double *systime, double *walltime)
{
  double mega = 1.0e-6;
  struct rusage buffer;
  struct timeval tp;
  struct timezone tzp;
  getrusage(RUSAGE_SELF, &buffer);
  gettimeofday(&tp, &tzp);
  *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.
     tv_usec;
  *systime  = (double) buffer.ru_stime.tv_sec +1.0e-6 * buffer.ru_stime.
     tv_usec;
  *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
}

void ImprimirTiempos(double utime0,double stime0,double wtime0,double utime1
   ,double stime1,double wtime1){
  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 acci nes de E/S)  %.10f s\n",  stime1 - stime0);
  printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
     ) / (wtime1 - wtime0));

  //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 acci nes de E/S)  %.10e s\n",  stime1 - stime0);
  printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
     ) / (wtime1 - wtime0));
  printf("\n");
}
```

Burbuja Simple

```c
#include<stdio.h>
#include<stdlib.h>
#include <sys/resource.h>
#include <sys/time.h>

void uswtime(double *usertime, double *systime, double *walltime)
{
  double mega = 1.0e-6;
  struct rusage buffer;
  struct timeval tp;
  struct timezone tzp;
  getrusage(RUSAGE_SELF, &buffer);
  gettimeofday(&tp, &tzp);
  *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.
     tv_usec;
  *systime  = (double) buffer.ru_stime.tv_sec +1.0e-6 * buffer.ru_stime.
     tv_usec;
  *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
}

//Funcion para Imprimir los tiempos.
void ImprimirTiempos(double utime0,double stime0,double wtime0,double utime1
   ,double stime1,double wtime1){
```

```
21    printf("\n");
22    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
23    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 - utime0
          );
24    printf("sys (Tiempo en  a c c i nes  de E/S)  %.10f s\n",  stime1 - stime0
          );
25    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
26    printf("\n");
27
28    //Mostrar los tiempos en formato exponecial
29    printf("\n");
30    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
31    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 - utime0
          );
32    printf("sys (Tiempo en  a c c i nes  de E/S)  %.10e s\n",  stime1 - stime0
          );
33    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
34    printf("\n");
35  }
36
37  int main(int argc, char const *argv[])
38  {
39    int *numeros = (int*) malloc(sizeof(int)*10000000); //Reservar Memoria.
40    int n, i, j; //Variables para los for y argumento
41    int temp = 0; //variable auxiliar para el Algoritmo.
42    double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
          m e d i c i n  de tiempos.
43
44    n = atoi(argv[1]); //Se convierte en Int el Argumento y se asigna a n.
45
46    //For para poder leer las entradas.
47    for(i = 0; i < n; i++){
48
49      scanf("%d",&numeros[i]);
50
51    }
52
53    uswtime(&utime0, &stime0, &wtime0); //Se llama la funcion para los tiempos
          .
54
55    //COMIENZA ALGORTIMO DE BURBUJA
56    for (i = 0; i <= n-2; i++)
57    {
58      for (j = 0; j <= (n-2)-i; j++)
59      {
60        if (numeros[j] > numeros[j+1])
61        {
62          temp = numeros[j];
63          numeros[j] = numeros[j+1];
64          numeros[j+1] = temp;
65        }
66      }
67
68    }
69    //ACABA ALGORITMO DE BURBUJA
70
71    uswtime(&utime1, &stime1, &wtime1); //Se llama la funcion para los tiempos
          .
72
```

```
73    //Se llama a la funcion para Imprimir los tiempos.
74    printf("Tamanio de N = %d", n);
75    ImprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
76    printf("############################## \n");
77    return 0;
78  }
```

Burbuja optimizada

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include <sys/resource.h>
4   #include <sys/time.h>
5
6   void uswtime(double *usertime, double *systime, double *walltime)
7   {
8     double mega = 1.0e-6;
9     struct rusage buffer;
10    struct timeval tp;
11    struct timezone tzp;
12    getrusage(RUSAGE_SELF, &buffer);
13    gettimeofday(&tp, &tzp);
14    *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.
          tv_usec;
15    *systime  = (double) buffer.ru_stime.tv_sec +1.0e-6 * buffer.ru_stime.
          tv_usec;
16    *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
17  }
18
19  //Funcion para Imprimir los tiempos.
20  void ImprimirTiempos(double utime0,double stime0,double wtime0,double utime1
        ,double stime1,double wtime1){
21    printf("\n");
22    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
23    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 - utime0
          );
24    printf("sys (Tiempo en  a c c i nes  de E/S)  %.10f s\n",  stime1 - stime0
          );
25    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
26    printf("\n");
27
28    //Mostrar los tiempos en formato exponecial
29    printf("\n");
30    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
31    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 - utime0
          );
32    printf("sys (Tiempo en  a c c i nes  de E/S)  %.10e s\n",  stime1 - stime0
          );
33    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
34    printf("\n");
35  }
36
37  int main(int argc, char const *argv[])
38  {
39    int *numeros = (int*) malloc(sizeof(int)*10000000); //Reservar Memoria.
40    int n, i, j;  //Variables para los for y argumento
41    int aux = 0;  //variable auxiliar para el Algoritmo.
42    double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
          m e d i c i n  de tiempos
```

```
43    int cambios = 1;  //Variable para la Bandera del Algoritmo
44
45    n = atoi(argv[1]); //Se convierte en Int el Argumento y se asigna a n.
46
47    //For para poder leer las entradas.
48    for(i = 0; i < n; i++){
49
50      scanf("%d",&numeros[i]);
51
52    }
53
54    uswtime(&utime0, &stime0, &wtime0);//Se llama la funcion para los tiempos.
55
56    //COMIENZA ALGORTIMO DE BURBUJA
57
58    for (i = 0; i <= n-2 && cambios != 0; i++)
59    {
60      cambios = 0;
61      for (j = 0; j <= (n-2)-i; j++)
62      {
63        if (numeros[j] < numeros[j+1])
64        {
65          aux = numeros[j];
66          numeros[j] = numeros[j+1];
67          numeros[j+1] = aux;
68          cambios = 1;
69        }
70      }
71    }
72
73    //ACABA ALGORITMO DE BURBUJA
74
75    uswtime(&utime1, &stime1, &wtime1); //Se llama la funcion para los tiempos
          .
76
77    //Se llama a la funcion para Imprimir los tiempos.
78    printf("Tamanio de N = %d", n);
79    ImprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
80    printf("############################## \n");
81    return 0;
82  }
```

9.3. Inserción

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/resource.h>
4   #include <sys/time.h>
5   //prototipos de la funcion
6   double utime0, stime0, wtime0,utime1, stime1, wtime1;
7
8
9   //Funcion de inserccion
10  void Inserccion(int *A, int n){
11      int j, i, temp=0;
12      for(i=0; i<=n-1;i++){
13          j=i;
14          temp=(int)A[i];
15          while((j>0) && (temp<(int)A[j-1])){
16              A[j]=A[j-1];
```

```
17              j--;
18          }
19          A[j]=temp;
20      }
21  }
22
23  //Librerias para medir el tiempo
24
25  void uswtime(double *usertime, double *systime, double *walltime)
26  {
27    double mega = 1.0e-6;
28    struct rusage buffer;
29    struct timeval tp;
30    struct timezone tzp;
31    getrusage(RUSAGE_SELF, &buffer);
32    gettimeofday(&tp, &tzp);
33    *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.
        tv_usec;
34    *systime  = (double) buffer.ru_stime.tv_sec +1.0e-6 * buffer.ru_stime.
        tv_usec;
35    *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
36  }
37
38
39  //funcion para imprimir tiempos en linux
40  void ImprimirTiempos(double utime0,double stime0,double wtime0,double utime1
      ,double stime1,double wtime1){
41    printf("\n");
42    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
43    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 - utime0
        );
44    printf("sys (Tiempo en acci  nes de E/S)  %.10f s\n",  stime1 - stime0);
45    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
        ) / (wtime1 - wtime0));
46    printf("\n");
47
48    //Mostrar los tiempos en formato exponecial
49    printf("\n");
50    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
51    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 - utime0
        );
52    printf("sys (Tiempo en acci  nes de E/S)  %.10e s\n",  stime1 - stime0);
53    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
        ) / (wtime1 - wtime0));
54    printf("\n");
55  }
56
57
58  int main(int argc, char *argv[]){
59
60
61      int *numeros=(int *)malloc(sizeof(int)*1000000);//Arreglo dinamico de
          aountadores a enteros
62    int var; //variable que guarda los numeros leidos del archivo
63    int cantidad= atoi(argv[1]);//cantidad de numeros a leer
64      int n;
65
66    n=atoi(argv[1]);
67
68    for(int i = 0; i < n + 1; i++){
69      scanf("%d",&var);
```

```
70      numeros[i]=var;
71    }
72
73
74    uswtime(&utime0, &stime0, &wtime0);
75      Inserccion(numeros, cantidad);//llamada a inserccion
76    uswtime(&utime1, &stime1, &wtime1);
77
78    ImprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
79
80
81      return 0;
82  }
```

9.4. Selección

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <sys/resource.h>
5   #include <sys/time.h>
6
7   void uswtime(double *usertime, double *systime, double *walltime)
8   {
9     double mega = 1.0e-6;
10    struct rusage buffer;
11    struct timeval tp;
12    struct timezone tzp;
13    getrusage(RUSAGE_SELF, &buffer);
14    gettimeofday(&tp, &tzp);
15    *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.
          tv_usec;
16    *systime  = (double) buffer.ru_stime.tv_sec +1.0e-6 * buffer.ru_stime.
          tv_usec;
17    *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
18  }
19
20  void ImprimirTiempos(double utime0,double stime0,double wtime0,double utime1
        ,double stime1,double wtime1){
21    printf("\n");
22    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
23    printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 - utime0
          );
24    printf("sys (Tiempo en acci nes de E/S)  %.10f s\n",  stime1 - stime0);
25    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
26
27    //Mostrar los tiempos en formato exponecial
28    printf("\n");
29    printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
30    printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 - utime0
          );
31    printf("sys (Tiempo en acci nes de E/S)  %.10e s\n",  stime1 - stime0);
32    printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 - stime0
          ) / (wtime1 - wtime0));
33    printf("\n");
34  }
35  //Programa principal
36  int main (int argc, char* argv[])
37  {
```

```
38
39    double utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para
          medici n de tiempos
40    int n, i, j, k, aux, cambios=0,temp,p;  //Variables del algoritmo
41    int *A; //Apuntador para hacer el arreglo
42    char true,false; //Variables booleanos
43
44    //Si no recibe el argumento se cierra
45    if (argc!=2)
46      exit(1);
47
48    //Numero de numeros que se ordenar n
49    n=atoi(argv[1]);
50    //Creaci n del arreglo
51    A=malloc(n*sizeof(int));
52
53    //Se guardan los numeros en el arreglo
54    for(i=0; i<n;i++);
55      scanf("%i",&A[i]);
56
57    printf("\tselecion para %i numeros\n", n);
58
59
60    uswtime(&utime0, &stime0, &wtime0);
61
62    //Algoritmo
63      for(k=0;k<n-2;k++)
64      {
65        //Se recorre el arreglo
66          p=k;
67          for(i=k+1;i<n-1;i++)
68          {
69            //Por cada vez que se recorre, comparamos el actual con el minimo
70            if(A[i]<A[p])
71            {
72              p=i;
73            }
74          }
75          //Aqui ya tiene el minimo y lo guarda
76          temp = A[p];
77          A[p]=A[k];
78          A[k]= temp;
79      }
80
81    uswtime(&utime1, &stime1, &wtime1);
82
83    //C lculo del tiempo de ejecuci n del programa
84    ImprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
85    printf("\n");
86    //*****************************************************************
87
88    printf("
          ##########################################################################
          n");
89
90    return 0;
91  }
```

9.5. Shell

```c
//Algoritmo de Shell
//Equipo:
//Martinez Partida Jair Fabian
//Martinez Rodriguez Alejandro
//Monteros Cervantes Miguel Angel
//Ramirez Cotonieto Luis Fernando

//Inicio de librerias//

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

//Fin de librerias//

//Programa Principal//
void main (int args, char *argv[]){
  //Variables para medir tiempo
  double utime0, stime0, wtime0, utime1, wtime1;

  //Numeros que se ordenasn
  if(argc!=2)
    exit(1);

    //Variables
    int n=atoi(argv[1]);
    int b,i,temp;
    int k=trunc(n/2); // Dividimos entre 2 para conocer los saltos
                      // Se trunca el entero para la exactitud
    printf("Proceso de shell en %i digitos\n",n);

    //Creacion de arreglo de forma dinamica
    int *A;
    A= (int*)malloc(n*sizeof(int));
    //Para obtener datos
    for(i=0; i<n; i++)
        scanf("%d",&A[i]);
    //Inicio de conteo de tiempo
    uswtime(&utime0, &stime0, &wtime0);
    //Fin de conteo de tiempo

  //Algoritmo de shell propuesto
  while(k>=1){ //Mientras k sea igual o mayor a 1, el proceso se llevara a
     cabo
    b=1;
    while (b!=0) {
      b=0; //Cuando ya no haya nada que ordenar, continua con k/2
      for (i=k; i<n-1;i++)
      {
        if(A[k-1]>A[i]){//Se compara dependiendo el salto, cmabia si el del
          //la derecha es menor
          temp=A[i];
          A[i]=A[i-k];
          A[i-k]=temp;
          b++;
        }
      }
    }
    k=trunc(k/2);
    //Si k=1 se realiza un ordenamiento de insercion
```

```
61    }
62    //Tiempos de ejecucion
63    uswtime(&utime1, &stime1, &wtime1);
64    //Calculo de tiempo de ejecucion
65    printf("-----------:)----------\n");
66    printf("Tiempo real %.10f s\n", wtime1 - wtime0);
67    printf("Tiempo de procesamiento %.10fs\n", utime1 - utime0);
68    printf("Tiempo en acci nes de sistema (E/S) %.10fs\n", stime1 - stime0);
69    printf("CPU/Wall %.10f%% \n",100.0 * (utime1 - utime0 + stime1 - stime0) /
          (wtime1 - wtim e0));
70    printf("-----------:)----------\n");
71    //tiempos de manera exponencial
72    printf("-----------:)----------\n");
73    printf("Tiempo real %.10e s\n", wtime1 - wtime0);
74    printf("Tiempo de procesamiento %.10es\n", utime1 - utime0);
75    printf("Tiempo en acci nes de sistema (E/S) %.10es\n", stime1 - stime0);
76    printf("CPU/Wall %.10f%% \n",100.0 * (utime1 - utime0 + stime1 - stime0) /
          (wtime1 - wtim e0));
77    printf("-----------:)----------\n");
78    //
79    for(i=0;i<27;i++)
80      printf("\n");
81
82      exit(0);
83  }
```