ANALISIS DE ALGORITMOS – PRÁCTICA 1

Daniel Brito Sotelo y Manuel Suárez Román– grupo 1202

26 de octubre de 2018

1. Introducción

En esta practica se nos pide, primero, se nos pide crear funciones que generen números aleatorios que mas tarde usaremos en arrays para generar permutaciones. Más tarde ordenaremos estas permutaciones.

2. Objetivos

Apartado 1

Se nos pide que creemos una función cuyo uso será la generación de números aleatorios equiprobables entre dos enteros.

Apartado 2

En este caso y usando la función desarrollada en el apartado 1, se busca la obtención de código para generar permutaciones de N elementos dados.

Apartado 3

Ahora, y de nuevo empleando las funciones ya creadas, haremos una nueva rutina que, esta vez generara un número dado de permutaciones de N elementos.

Apartado 4

En este apartado implementaremos SelectSort, un método de ordenación, su funcionamiento es el siguiente. En cada iteración busca el mínimo de la tabla, o subtabla correspondiente y lo coloca en primera posición, en la primera iteración buscará por tanto el mínimo en toda la tabla, en el resto buscará el mínimo en la subtabla.

Apartado 5

Ahora se nos pide contabilizar el número de OBs (Operaciones Básicas) y tiempos de ejecución del algoritmo de SelectSort, al ordenar unas permutaciones.

Apartado 6

Repetir el apartado anterior para el SelectSort Inverso.

3. Herramientas y metodología

Para realizar esta práctica hemos empleado el editor de texto Atom en Ubuntu al considerarlo el más versátil y cómodo de ente los que disponemos en la EPS. Para compilar haremos uso de gcc y para comprobar que nuestro código no tuviera fugas de código empleamos Valgrind. Como herramienta de para la generación de graficos empleamos GNUPlot, puesto que es el generador de gráficos más cómodo y que nos permite utilizar scripts para generarlos automáticamente.

Apartado 1

Empleamos la función dada por el profesor Carlos Aguirre, que hace uso de la función rand(). Para conseguir una distribución equiprobable entre el 0 y el 1 dividimos rand() por RAND_MAX + 1 y todo ello lo multiplicamos por (sup-inf+1) para conseguir que el numero aleatorio obtenido se encuentre entre 0 y sup-inf. Sumando + inf al final conseguimos que el numero aleatorio se encuentre entre inf y sup.

Apartado 2

En este apartado se nos pide crear una rutina que genere permutaciones de N elementos, creamos la función int* genera_perm que recibe como argumentos únicamente un entero N que dictara el tamaño de la permutación. Primero reserva memoria para un puntero a int que tendrá tamaño N, primero creamos una tabla ordenada del 1 a N, y después haciendo uso de un int temporal, y un int random, va generando números aleatorios y moviendo en cada iteración la posición “i” de la tabla a la posición aleatoria de random, haciendo llamadas a aleat_num (función del apartado 1). Asegurando así que la tabla será una tabla desordenada que contendrá todos los enteros desde el 1 a N.

Apartado 3

Para resolver este apartado creamos la función int** genera_permutaciones que recibe dos argumentos de tipo entero, n_perms (número de permutaciones a crear) y N (el tamaño de estas). Primero reserva memoria para el doble puntero a entero perms, que usaremos para guardar todas las permutaciones. Reservamos memoria n_perms para este y procedemos a crear cada permutación y a guardarlas luego en el doble puntero. Para ello hacemos un bucle for que llame n_perms veces a la función creada para el apartado anterior, genera_perm, pasando como argumento N, el tamaño de las permutaciones. En caso de que se produjera un error en alguna iteración del bucle se liberaran todas las permutaciones creadas antes de la que dio el error, y finalmente se liberaría el doble puntero.

Apartado 4

Diseñamos el método de ordenación int SelectSort, que recibe una tabla y en cada iteración obtiene el mínimo de la tabla y lo coloca en la primera posición de la tabla, después realiza lo mismo para la subtabla obtenida desde el nuevo segundo elemento al final de la tabla. Así hasta que la ultima subtabla sea un solo elemento. Para hacer los movimientos hacemos uso de una variable temporal.

Apartado 5

Creamos la función tiempo_medio_ordenacion que recibirá el método de ordenación, n_perms, el tamaño de estas permutaciones, N, y una variable PTIEMPO tiempo donde guardaremos el tiempo medio, el número máximo, mínimo y medio de OBs y el número de elementos. Iniciamos un clock justo antes de ordenar la tabla con el método deseado y lo asignamos a una variable temporal t1 y al terminar de nuevo haciendo uso de otra variable temporal t2 que volvemos a igualar a clock() podemos obtener el tiempo medio haciendo la diferencia de t2 y t1 y partiéndolo por el número de permutaciones, n_perms. En cada iteración contamos el número de iteraciones y las añadimos a un contador con el que luego daremos el número medio de OBs.

Apartado 6

Consiste en aplicar las mismas funciones que utilizamos en el apartado 5, solo que esta vez utilizaremos el SelectSort Inverso, la única diferencia entre estos dos métodos de ordenación es que en cada iteración el SelectSort Inverso buscara el máximo de la tabla o subtabla y lo coloca al final

4. Código fuente

A continuación incluimos el código generado en el desarrollo de toda la práctica dividiendolo por apartados.

Apartado 1

```
int aleat_num(int inf, int sup)
{
  return rand()/(RAND_MAX+1.)*(sup-inf+1)+inf;
}
```

Apartado 2

```c
int* genera_perm(int N)
{

  int *perm;
  int i, random, dum;
  perm = calloc(N,sizeof(int));
  if (!perm) return NULL;

  for(i=0;i<N;i++)  perm[i]=i+1;
  for(i=0;i<N;i++){
    random = aleat_num(i,N-1);
    dum = perm[random];
    perm[random] = perm[i];
    perm[i]=dum;
  }

  return perm;
}
```

Apartado 3

```c
int** genera_permutaciones(int n_perms, int N)
{

  int **perms;
  int i;
  perms = calloc(n_perms,sizeof(int*));
  if (!perms) return NULL;
  for(i=0;i<n_perms;i++){
    perms[i]=genera_perm(N);
    if(!perms[i]){
      for(i=i-1;i>=0;i--){
        free(perms[i]);
      }
      free(perms)
      return NULL;
    }
  }
  return perms;
}
```

Apartado 4

```
int SelectSort(int* tabla, int ip, int iu)
 /*****************************************************/
 short tiempo_medio_ordenacion(pfunc_ordena metodo,
                               int n_perms,
                               int N,
                               PTIEMPO ptiempo)
 {
   int i, res, minimo, max, ac = 0;
   int **perms;
   clock_t t1,t2;
   perms = genera_permutaciones(n_perms,N);
   if(perms == NULL){return ERR;}
     t1 = clock();
     for(i=0;i<n_perms;i++){
       res = metodo(perms[i],0,N-1);
       if(res == ERR){
         ptiempo->N = 0;
         ptiempo->tiempo = 0.0;
         ptiempo->min_ob = INT_MAX;
         ptiempo->max_ob = 0;
         ptiempo->medio_ob =0.0;
         ptiempo->n_elems = 0;
       }
       if(res<minimo || i == 0) minimo = res;
       if(res>max || i == 0) max = res;
       ac +=res;
       free(perms[i]);
   }
   t2 = clock();

   ptiempo->N = n_perms;
   ptiempo->tiempo = (t2-t1)/n_perms;
   ptiempo->min_ob = minimo;
   ptiempo->max_ob = max;
   ptiempo->medio_ob = ac/n_perms;
   ptiempo->n_elems = N;

   free(perms);
   return OK;
 }
```

Apartado 5

```c
/*************************************************/
short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                                int num_min, int num_max,
                                int incr, int n_perms)
{
  int i, cont = 0;
  PTIEMPO tiempo;
  tiempo = malloc(sizeof(PTIEMPO)*((num_max-num_min)/(incr)));
  if(!tiempo){
    return ERR;
  }
  for (i = num_min; i<=num_max; i+=incr){
    if(tiempo_medio_ordenacion(metodo,n_perms,i,&tiempo[cont])==ERR){
      free(tiempo);
      return ERR;
    }
    cont++;
  }
  if(guarda_tabla_tiempos(fichero, tiempo,(num_max-num_min)/(incr))==ERR){
    free(tiempo);
    return ERR;
  }
free(tiempo);
return OK;
}
```

```c
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
{
  int i, flag;
  FILE *fp;
  if(fichero == NULL || tiempo == NULL){
    return ERR;
  }
  fp = fopen(fichero, "a");
  if(fp == NULL){
    return ERR;
  }
  for(i=0;i<n_tiempos;i++){
   flag = fprintf(fp, "%d %f %d %f %d \n", tiempo[i].n_elems, tiempo[i].tiempo, tiempo[i].min_ob,
    tiempo[i].medio_ob, tiempo[i].max_ob);
   if(flag == ERR){
    fclose(fp);
    return ERR;
   }
  }
 fclose(fp);
 return OK;
}
```

Apartado 6

En este apartado volvemos a usar las funciones del apartado 5 y empleamos el SelectSort inverso expuesto a continuación.

```
int SelectSortInv(int* tabla, int ip, int iu)
{
  int i,max,j,temp,count=0;
  i = ip;
  while(i<iu){
    max = i;
    for(j=i+1;j<=iu;j++){
      if(tabla[j]>tabla[max]){
        max = j;
      }
      count++;
    }
    temp = tabla[i];
    tabla[i] = tabla[max];
    tabla[max] = temp;
    i++;
  }
  return count;
}
```

5. Resultados, Gráficas

A continuación mostramos las gráficas generadas y los resultados obtenidos en la ejecución de los distintos ejercicios

Apartado 1

Adaptamos el makefile para que devuelva 20 números entre el 1 y el 20

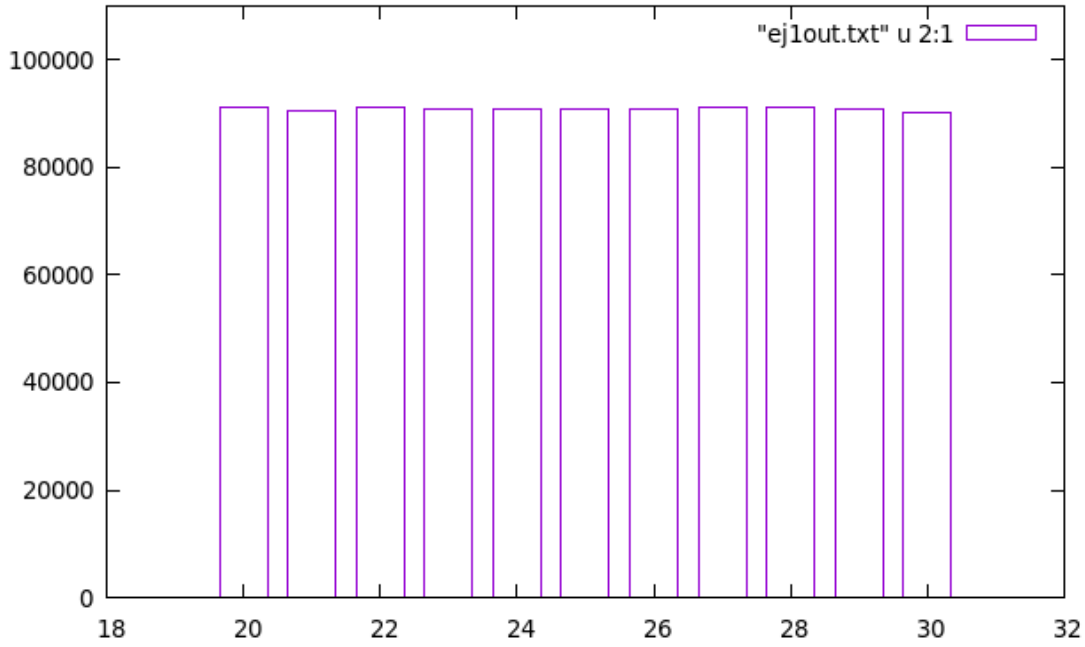
```
Ejecutando ejercicio1
Practica numero 1, apartado 1
Realizada por: Daniel Brito, Manuel Suárez
Grupo: 1201
3
8
3
7
3
9
6
9
19
18
19
14
11
2
7
17
12
16
20
10
```

Ahora para realizar el histograma generamos un millón de números entre el 20 y el 30 por ejemplo usando esta etiqueta

```
./ejercicio1 -limInf 20 -limSup 30 -numN 1000000 | sort | uniq -c > ej1out.txt
```

después usando GNUPlot creamos el histograma correspondiente, cambiando ligeramente conseguimos que la grafica de gnuplot sea un diagrama de barras. El atributo boxwidth se cambia para una visualización mas sencilla del resultado.

```
gnuplot> set boxwidth 0.7
gnuplot> plot "ej1out.txt" u 2:1 with boxes
```



Con estos resultados podemos observar gráficamente que nuestra función genera números aleatorios equiprobables.

Apartado 2

```
Ejecutando ejercicio2
Practica numero 1, apartado 2
Realizada por: Daniel Brito, Manuel Suárez
Grupo: 1201
5 10 4 6 2 7 3 9 8 1
3 2 5 6 1 8 9 10 7 4
1 7 3 10 9 8 6 5 2 4
8 6 4 10 3 7 2 1 5 9
2 3 1 7 4 6 10 8 9 5
4 3 10 6 5 9 2 1 8 7
8 9 6 4 2 7 3 10 5 1
9 6 4 10 3 1 7 5 8 2
2 1 5 4 7 3 9 8 10 6
2 8 7 3 9 4 10 6 5 1
```

Apartado 3

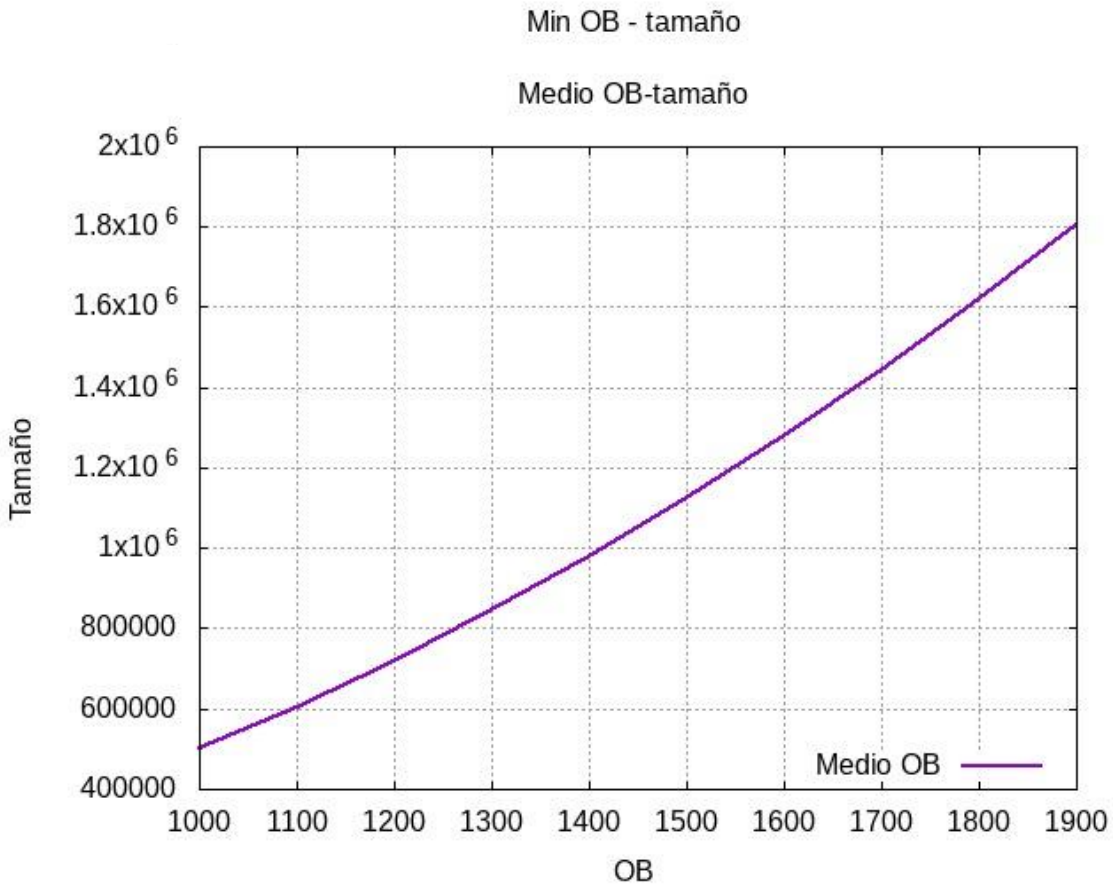
```
Ejecutando ejercicio3
Practica numero 1, apartado 3
Realizada por: Daniel Brito, Manuel Suárez
Grupo: 1201
7 1 10 3 5 8 2 9 4 6
7 3 10 4 9 1 5 6 2 8
3 10 6 5 2 1 9 7 4 8
4 1 2 5 6 7 9 8 10 3
8 7 9 1 4 5 10 6 3 2
2 4 8 3 7 1 9 6 5 10
8 5 7 10 9 4 3 2 1 6
1 9 8 2 3 7 6 4 10 5
1 8 7 2 6 10 3 4 9 5
4 10 8 2 1 7 5 3 6 9
2 8 9 10 7 5 6 4 1 3
10 9 6 7 4 2 8 1 5 3
10 3 7 8 5 2 1 6 9 4
9 7 5 1 2 10 3 4 6 8
4 5 7 9 10 6 1 8 2 3
```

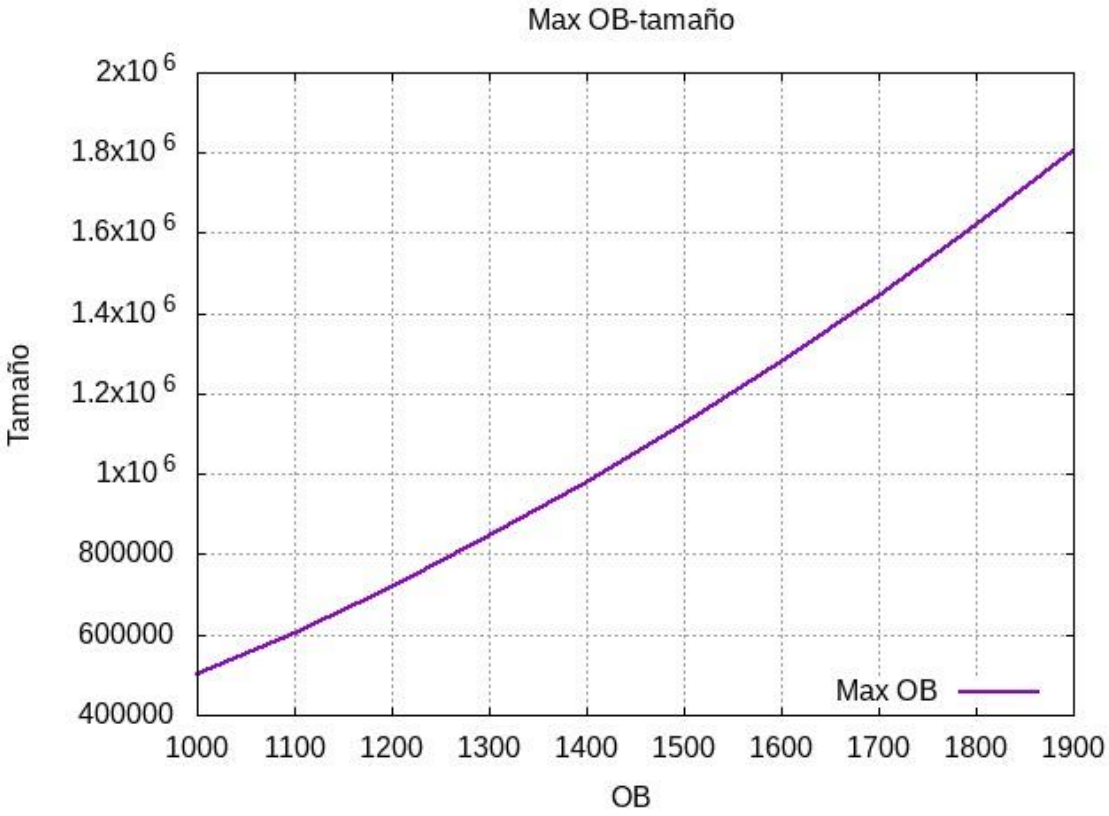
Apartado 4

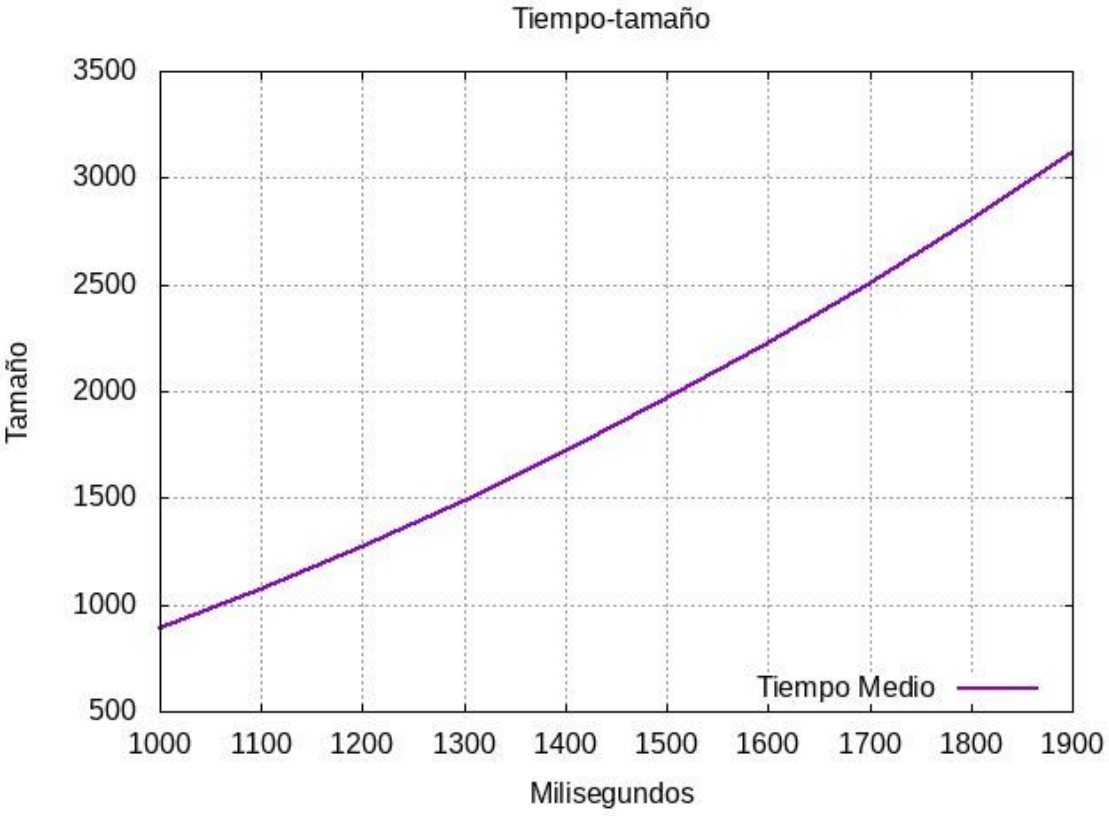
En este apartado el resultado serán los números del 1 al 100 en orden ascendente. Este sería el caso para los números del 1 al 10.

```
Ejecutando ejercicio4
Practica numero 1, apartado 4
Realizada por: Daniel Brito, Manuel Suárez
Grupo: 1201
1       2       3       4       5       6       7       8       9       10
```

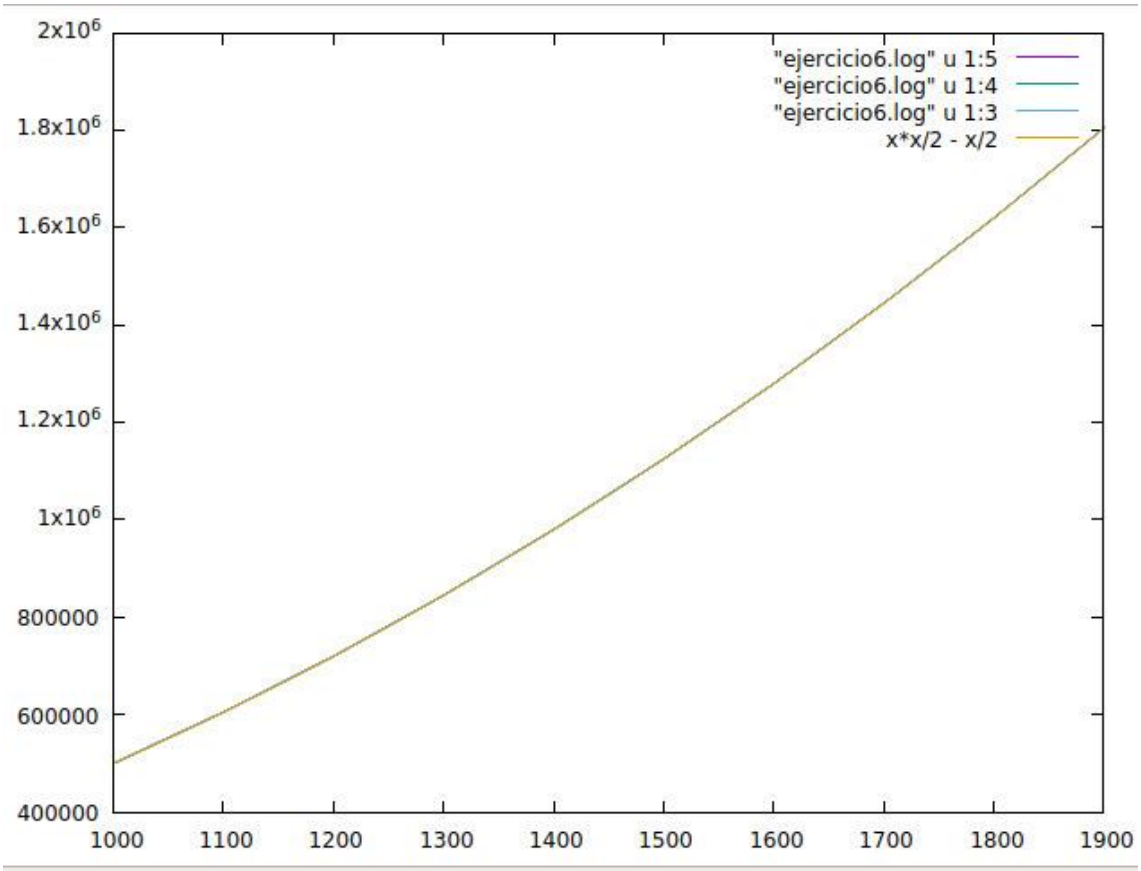
Apartado 5


Min OB - tamaño
Medio OB-tamaño
2x10^6
1.8x10^6
1.6x10^6
1.4x10^6
1.2x10^6
1x10^6
800000
600000
400000
Tamaño
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
OB
Medio OB


Max OB-tamaño
2x10 6
1.8x10 6
1.6x10 6
1.4x10 6
1.2x10 6
1x10 6
800000
600000
400000
Tamaño
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
OB
Max OB


Tiempo-tamaño
3500
3000
2500
2000
1500
1000
500
Tamaño
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
Milisegundos
Tiempo Medio

5.6 Apartado 6



Podemos comprobar que se trata de la misma grafica que para todos los casos del SelectSort, es decir, se superponen. Comprobamos así lo aprendido en la clase de teoría, que SelectSort tiene el mismo tiempo y numero de OBs sin importar el estado de la tabla a ordenar.

5. Respuesta a las preguntas teóricas.

A lo largo de la realización de la práctica se nos plantean varias cuestiones teóricas que pretendemos responder lo más preciso a continuación

Pregunta 1

Un método alternativo podría ser el hacer rand ()% (sup-inf +1) + inf. Sin embargo de esta forma, no se conseguiría una equiprobabilidad de resultados ya que, al dividir rand() entre la diferencia entre supremo e ínfimo, depende de esta diferencia, y por ejemplo, los números pequeños tendrían más probabilidad de aparecer que los grandes.

Pregunta 2

SelectSort encuentra el mínimo de cada subtabla en cada iteración, es decir, en la primera iteración encuentra el mínimo de la tabla completa y lo coloca en la primera posición. Después tomando toda la tabla menos la primera posición, halla de nuevo el

mínimo y lo vuelve a colocar en la primera posición de esa subtabla, la segunda posición de la tabla completa, de esta forma colocamos cada elemento en su posición correspondiente.

Pregunta 3

Ya que al llegar al ultimo elemento, sabemos que este es mayor que el resto, ya que no ha sido mínimo de la subtabla en ninguna iteración, por lo que al llegar a la última iteración este elemento no se moverá.

Pregunta 4

La Operación Básica (OB) de SelectSort es la comparación de claves, ya que además de ser la operación por excelencia de los algoritmos de ordenación, cumple además que se encuentra en el bucle mas interno.

Pregunta 5

SelecSort no depende del estado de la tabla que vaya a ordenar, es decir que realizara el mismo número de operaciones básicas si la tabla esta ya ordenada como si esta completamente desordenada, esto es porque sus bucles se ejecutan, el interno de i+1 a n y el externo de 1 a n-1, y no dependen de la entrada.

Por lo que tenemos que Wss(N) = Ass(N) = Bss(N)

$$W_{SS}(N) = \frac{N(N-1)}{2} = N^2 + O(N)$$

Pregunta 6

SelectSort y SelectSortInv hacen el mismo numero de operaciones básicas, comparaciones de clave para una permutación de igual tamaño.

6. Conclusiones finales.

En esta practica hemos podido poner en practica los algoritmos de ordenación SelectSort y su inverso y comprobar que los resultados experimentales que obtenemos se corresponden con los que aprendimos en las clases teóricas. Comenzamos la práctica creando una rutina que generara números aleatorios que luego emplearíamos para la creación de permutaciones, aquí hicimos uso de la función rand y de limites superiores

e inferiores en la creación de los números. Este apartado nos llevo a intentar que hubiera la mayor equiprobabilidad entre los números generados.

Después, podemos comprobar que el algoritmo SelectSort, tal y como aprendimos en clase, es poco eficiente, pero siempre es interesante poder comprobar estos resultados de una forma tan gráfica al emplear GNUplot, algo que nos parece de gran ayuda a la hora de representar datos rápidamente en forma de gráficas, y desde luego será una herramienta a la que recurriremos mucho en el futuro. De nuevo esta práctica nos ha recordado la importancia de que nuestros programas no tengan fugas de memoria y de escribir un código limpio y eficiente.