Análisis de Algoritmos 2018/2019

Práctica 1

Román García, 1271.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se pide codificar los módulos necesarios para la creación de tablas de elementos, la permutación, y ordenado de las mismas según el algoritmo “SelectSort”. Adicionalmente, otro módulo que permita medir el tiempo de ejecución de la ordenación.

2. Objetivos

2.1 Apartado 1

Codificar la función “aleat_num” de forma que devuelva un número aleatorio atendiendo a sus parámetros.

2.2 Apartado 2

Codificar la funcion “genera_perm” para que cree una tabla de números aleatorios del tamaño especificado por parámetro.

2.3 Apartado 3

Codificar la función “genera_permutaciones” que genere varias tablas con números aleatorios.

2.4 Apartado 4

Implementar el algoritmo “SelectSort” de forma que devuelva el número de operaciones básicas que ha realizado.

2.5 Apartado 5

Implementar en el módulo de tiempos las funciones necesarias para la medición de tiempos de ejecución de las funciones de ordenación que se le pase por parámetro.

2.6 Apartado 6

Implementar una versión del algoritmo “SelectSort” que invierte la forma de ordenación de este.

3. Herramientas y metodología

Para todos los apartados he utilizado la misma metodología y las mismas herramientas:

Para saber qué piden exactamente he usado un boceto en papel de lo que debería hacer cada una de las funciones a codificar.

Una vez analizado el problema y diseñada la solución, he usado la combinación del IDE "Visual Studio Code" para codificar, "GitHub" para mantener un repositorio de versiones y una "bash Ubuntu 18.04" de "Windows" que he utilizado para compilar y ejecutar el código.

Para la creación de graficas he usado "Gnuplot" y bash script para el procesamiento de datos.

4. Código fuente

4.1 Apartado 1

```
int aleat_num(int inf, int sup)
{
  int num;

  if(inf>sup){
    return -1;
  }

  num = (rand()%(sup-inf+1)) + inf;

  return num;
}
```

4.2 Apartado 2

```
int* genera_perm(int n)
{
  int i;
  int* perm;
  int aux;
  int ale_num;

  if(n<1){
    return NULL;
  }

  perm = (int*) malloc (n*sizeof(int));

  if(!perm){
    return NULL;
  }

  for(i=0; i<n; i++){
    perm[i] = i+1; /*sumas 1 paras que la perm empiece en el 1*/
  }

  for(i=0; i<n; i++){

    ale_num = aleat_num(0, n-1); /*posicion donde voy a meter aux*/
    aux = perm[ale_num];    /*guardo el numero que esta en la posicion a
al que lo voy a mover*/
    perm[ale_num] = perm[i];   /*permuto el numero*/
    perm[i] = aux;               /*meto el numero de antes en la posicion
del que he querido mover*/
  }

  return perm;
}
```

4.3 Apartado 3

```
int** genera_permutaciones(int n_perms, int tamanio)
{
  int i;
    int** array;


    array = (int**) malloc(n_perms*sizeof(int*));

    if(!array)
      return NULL;

    for(i=0; i<n_perms; i++){
      array[i] = genera_perm(tamanio);
      if(!array[i]){
        while(i != 0){
          free(array[i]);
          i--;
        }
        free (array);
        return NULL;
      }
    }

    return array;
}
```

Adicionalmente he desarrollado una función que libera la memoria reservada por las funciones anteriores

```
int libera_permutaciones(int** perm, int n_perms){
  int i;

  if(!perm){
    return ERR;
  }

  for(i=0; i<n_perms; i++){
    free(perm[i]);
  }
  free(perm);

  return OK;
}
```

4.4 Apartado 4

```
int SelectSort(int* tabla, int ip, int iu)
{
  int index = ip;
    int min = ip;
    int obs = 0;

    for (; ip < iu; ip++ ){
        for (min = ip, index = ip; index <= iu; index++ ){

            obs++;
            if ( tabla[index] < tabla[min] ){
                min = index;
            }

        }

        if ( min != ip ){
            swap ( &tabla[min], &tabla[ip] );
        }
    }

    return obs;
}
```

Adicionalmente he implementado una funcion “swap” que intercambia los valores de los punteros pasados por parametro

```
void swap ( int * A, int * B ){
    int mem = *A;

    *A = *B;
    *B = mem;
}
```

4.5 Apartado 5

```
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int N,
                              PTIEMPO ptiempo)
{
  int i, ob, mob;
    int** permutaciones;
    clock_t t_ini, t_fin;
    double secs;


    if(n_perms<1 || N<1 || !ptiempo)
      return -1;

    permutaciones = genera_permutaciones(n_perms,N);

    t_ini = clock();

    for(i = 0, mob = 0 ; i < n_perms ; i++){

      ob = metodo(permutaciones[i],0, N-1);
      mob += ob;


      if(i == 0){

        ptiempo->min_ob = ob;
        ptiempo->max_ob = ob;

      }
      else{

        if(ob < ptiempo->min_ob){
          ptiempo->min_ob = ob;
        }

        if(ob > ptiempo->max_ob){
          ptiempo->max_ob = ob;
        }
      }

    }

    t_fin = clock();

    if(libera_permutaciones(permutaciones, n_perms)==ERR){
      return ERR;
```

```
    }

    secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;

    /* ptiempo->tiempo  = secs*1000000000.0; */
    ptiempo->tiempo  = (secs*1000000000.0)/(double)n_perms;

    ptiempo->medio_ob = (double)mob/(double)n_perms;
    /*printf("%d / %d = %lf\n",mob, n_perms, (double)mob/(double)n_perms
); */

    ptiempo->n_elems = n_perms;
    ptiempo->N = N;

    return OK;
}
```

```
short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                                int num_min, int num_max,
                                int incr, int n_perms)
{
  int i, p;
  PTIEMPO ptiempo;


  if(!fichero || incr < 1 || num_min < 1 || num_min > num_max || n_perms
< 1){
    return ERR;
  }


  p = ((num_max-num_min)/incr) + 1;


  ptiempo = (PTIEMPO) malloc(p*sizeof(TIEMPO));

  if(!ptiempo)
    return ERR;


  for(i=num_min, p=0; i<=num_max; i= i+incr, p++){
    tiempo_medio_ordenacion(metodo, n_perms, i, &ptiempo[p]);

  }

  guarda_tabla_tiempos(fichero, ptiempo, (p-1));
```

```
  free(ptiempo);

  return OK;
}
```

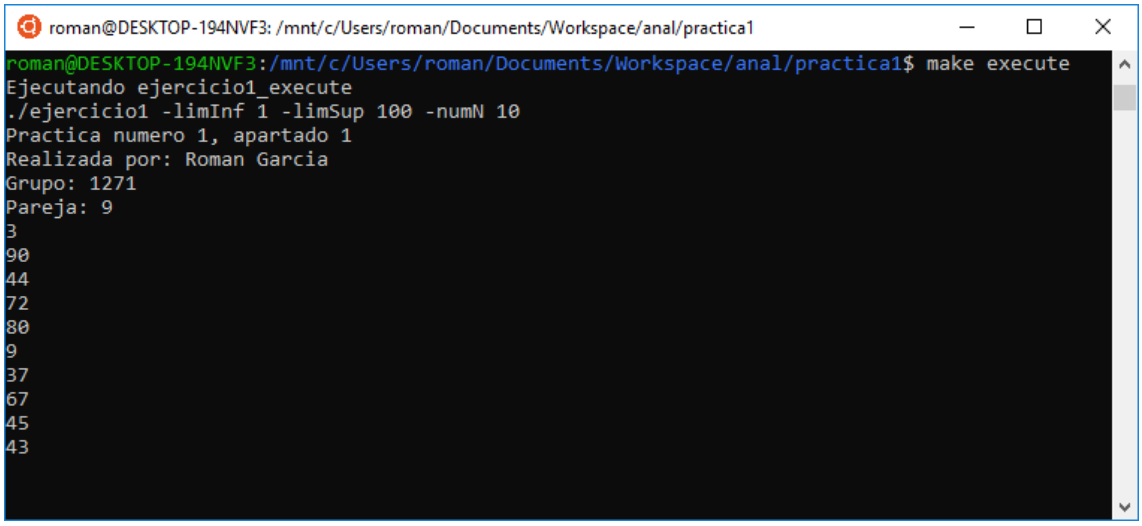
```
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
{
  PTIEMPO taux;
  int i;
  FILE* pf;

  if(!fichero || !tiempo || n_tiempos < 0){
    return ERR;
  }

  pf = fopen(fichero, "w");

  fprintf(pf, "Tamano  Tiempo(ns)  media_OB  max_OB  min_OB\n");

  for(taux = tiempo, i=0 ; i<=n_tiempos ; taux++, i++ ){
  fprintf(pf, "   %d    %.2f     %.2f         %d      %d\n",taux->N, taux-
>tiempo, taux->medio_ob, taux->max_ob, taux->min_ob);
  }

  fclose(pf);
  return OK;
}
```

4.6 Apartado 6

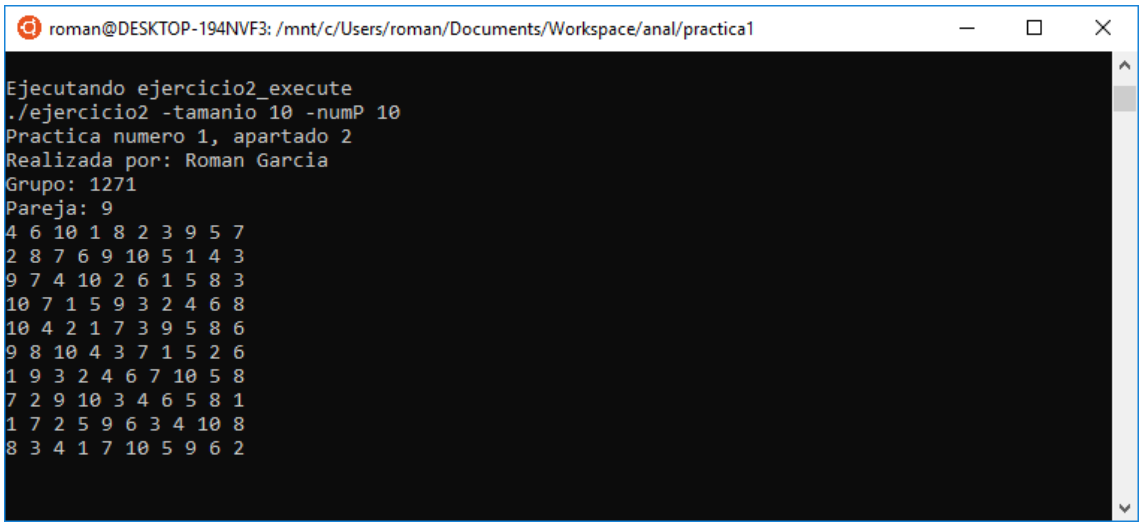
```
int SelectSortInv(int* tabla, int ip, int iu)
{
    int index = ip;
    int min = ip;
    int obs = 0;

    for (; ip < iu; ip++ ){

        for (min = ip, index = ip ; index <= iu; index++ ){

            obs++;
            if ( tabla[index] > tabla[min] ){
                min = index;
            }
        }

        if ( min != ip ){
            swap ( &tabla[min], &tabla[ip] );
        }

    }

    return obs;
}
```

5. Resultados, Gráficas

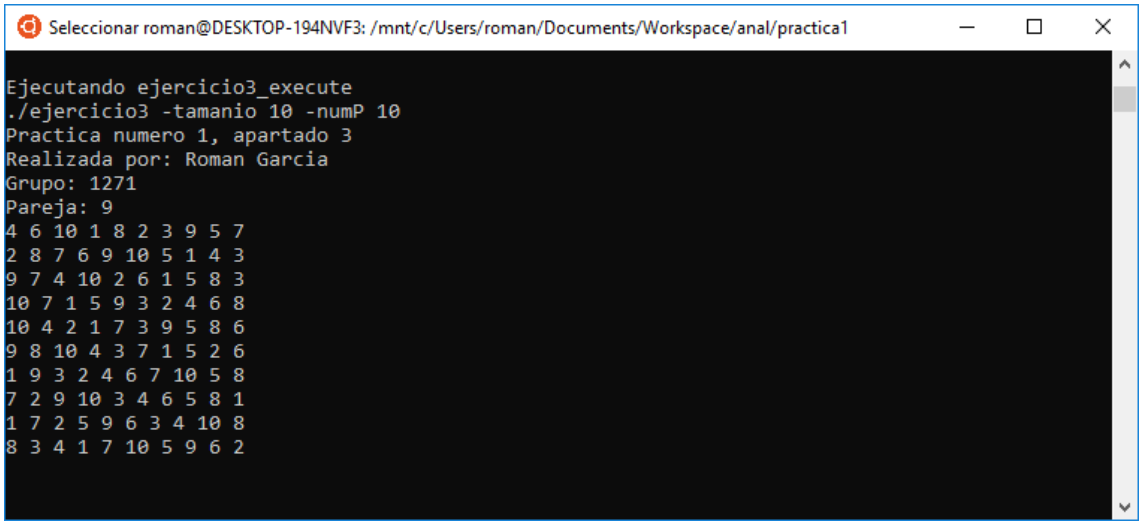
5.1 Apartado 1



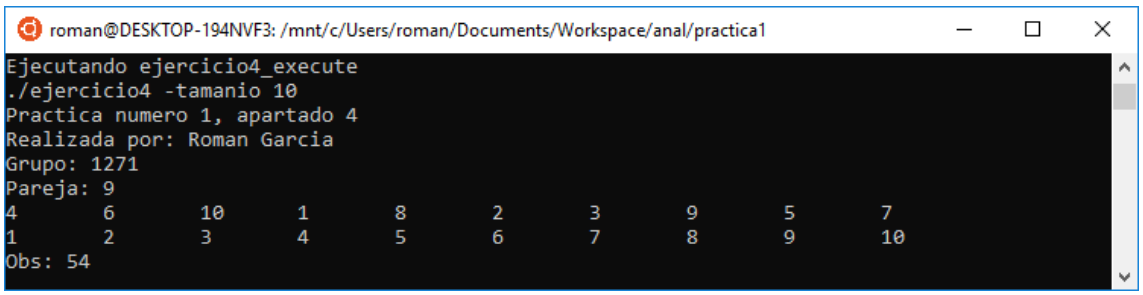
5.2 Apartado 2



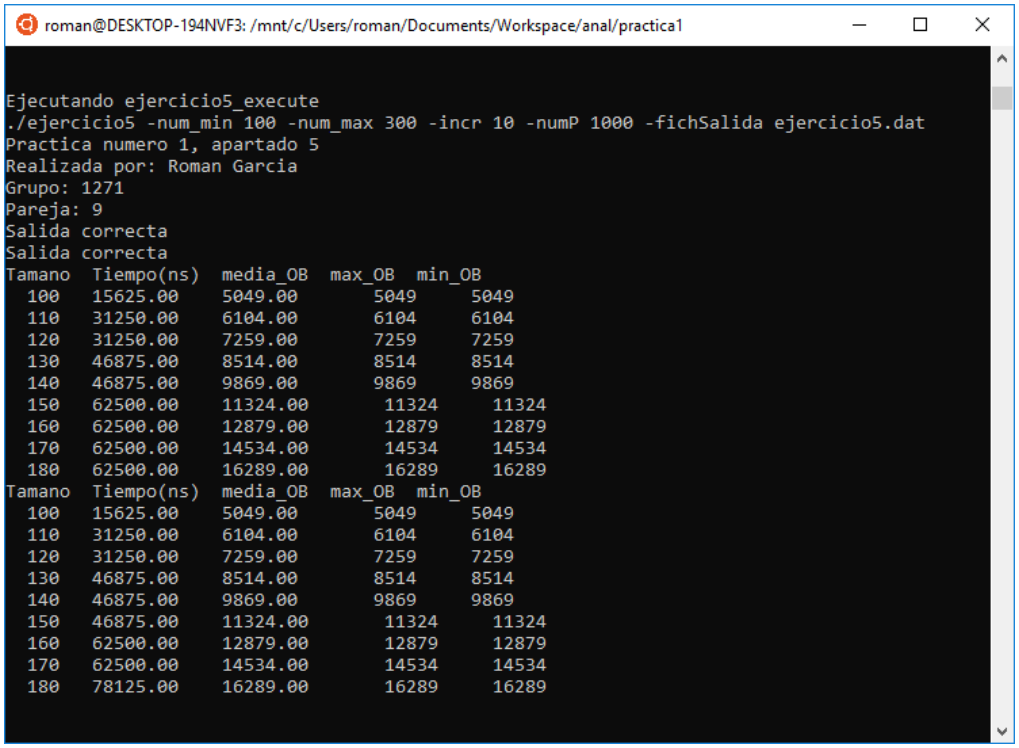
5.3 Apartado 3



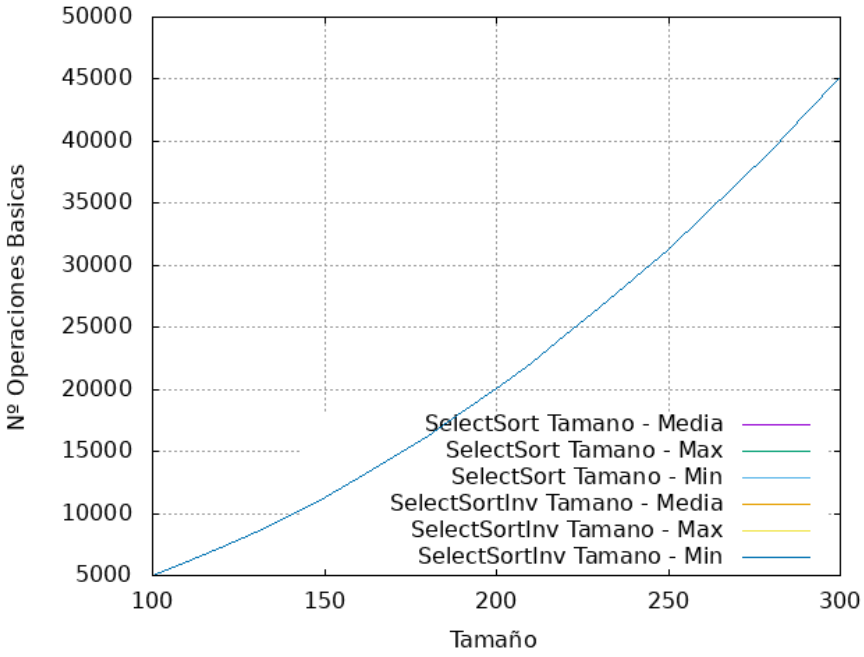
5.4 Apartado 4



5.5 Apartado 5 y Apartado 6

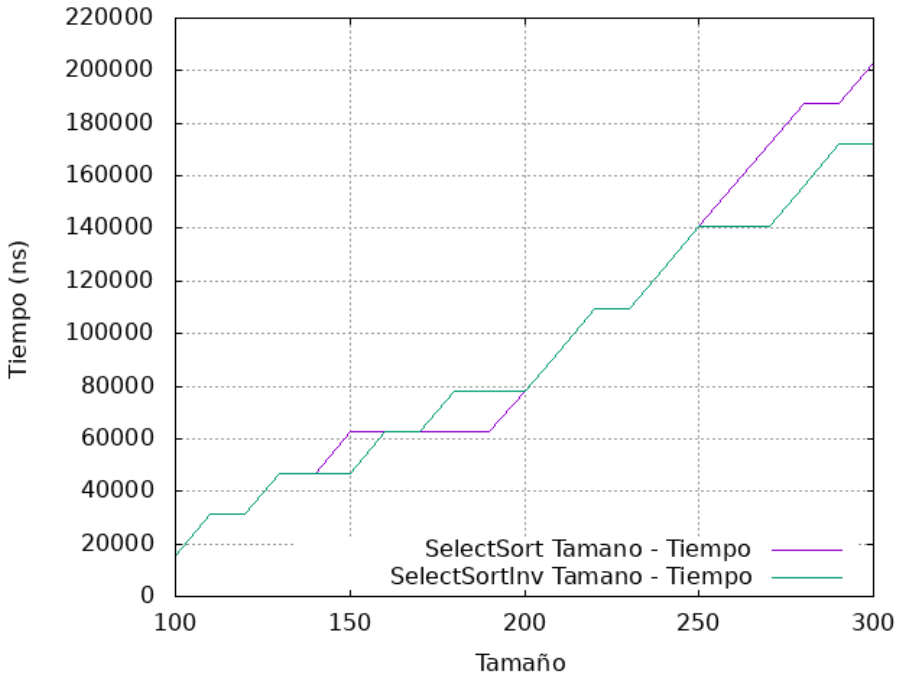


Como podemos observar en la siguiente gráfica, los casos mejor, peor y medio para el algoritmo “SelectSort” tienen los mismos valores:



En la siguiente tabla podemos ver la tabla de tamaño frente a tiempos de “SelectSort” y “SelectSortInv”. Como se puede apreciar, la diferencia es de un máximo de 30000 ns para tablas de mismo tamaño.

Esto se debe al rendimiento del procesador, el cual se ve afectado por procesos externos.



6. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

6.1 Pregunta 1

Para la generación de números aleatorios he usado la funcion rand() básica de C, usando semillas para que el número no dependa del momento de ejecucion, ya que de esa forma sería totalmente predecible.

Una implementación más exacta para conseguir números aleatorios seria la utilización de /dev/rand en Linux. Es mas costoso computacionalmente y requiere más tiempo pero tienen un índice muy bajo de repetición.

6.2 Pregunta 2

SelectSort es un algoritmo que funciona analizando, para cada elemento, lo que queda de tabla sin ordenar y colocando los elementos de menor a mayor. Esta forma, aunque costosa, es eficaz ya que recorre la tabla varias veces sin dejar un elemento fuera de su sitio.

6.3 Pregunta 3

Porque cuando llega a él, el resto de la tabla ya está ordenada.

6.4 Pregunta 4

La comparación de claves contenida en el bucle interno.

6.5 Pregunta 5

$$f(n) = \frac{n^2}{2} + \frac{n}{2} \qquad f(n) = \frac{n^2}{2} + O(n)$$

6.5 Pregunta 6

Los tiempos obtenidos en ambos difieren ligeramente debido al rendimiento del procesador, el cual se encuentra afectado por otros procesos externos a la práctica.

7. Conclusiones finales.

En esta práctica he podido ver la relación que guarda el tiempo de ejecucion de un algoritmo al intentar ordenar una tabla y el número de operaciones básicas que utiliza para ello.

Como conclusión, hace falta tener varios elementos en cuenta al intentar medir el rendimiento de un algoritmo que no están relacionados con el propio algoritmo ni con su implementación.