Análisis de Algoritmos

Práctica 1

Grupo 1261

Pareja 09

Ari Handler Gamboa

Adrián Lorenzo Mateo

1. Bloque 1

1.1 Introducción

Este primer bloque se centra en el desarrollo de rutinas en C que generen permutaciones aleatorias de un determinado número de elementos de un vector. Para ello es necesario utilizar la función **rand()** de la librería **stdlib** y evaluar el nivel de aleatoriedad del algoritmo utilizado.

1.2 Código

```
/*************************************************/
/* Funcion: swap Fecha:     19/09/2012           */
/* Autores: Adrián Lorenzo Mateo                 */
/*          Ari Handler Gamboa                   */
/*                                               */
/* Rutina que intercambia dos elementos de un array*/
/*         de enteros                            */
/*                                               */
/* Entrada:                                      */
/* int pos_1: posicion del primer elemento       */
/* int pos_2: posicion del segundo elemento      */
/* int * vector: array sobre el que se operara   */
/* Salida:                                       */
/* OK en caso de realizar el intercambio, ERR en  */
/* caso contrario                                */
/*************************************************/

int swap(int pos_1, int pos_2, int *vector)
{
      int temp;

      if((vector == NULL) || ((pos_1 < 0) || (pos_2 < 0)))
            return ERR;

      temp = *(vector+pos_1);
      *(vector+pos_1) = *(vector+pos_2);
      *(vector+pos_2) = temp;

      return OK;

}
```

Esta función sirve para intercambiar dos elementos de un vector dados sus índices. Es utilizada internamente por la librería **ordena.c** por lo que no aparece en la cabecera **.h** de la misma.

```
/*************************************************/
/* Funcion: aleat_num Fecha: 19/09/2012          */
/* Rutina que genera un numero aleatorio         */
/* entre dos numeros dados                       */
/*                                               */
/* Entrada:                                      */
/* int inf: limite inferior                      */
/* int sup: limite superior                      */
/* Salida:                                       */
/* int: numero aleatorio                         */
/*************************************************/
int aleat_num(int inf, int sup)
{
     if ((sup < inf) ||
         (sup > RAND_MAX) ||
         ((inf < 0 ) ||
         (sup < 0)))

           return ERR;
     else
           return (int)((rand()/(RAND_MAX+1.))*(sup-inf+1))+inf;
}
```

Esta rutina genera un número aleatorio comprendido entre los límites inferior y superior pasados como parámetros.

```
/*************************************************/
/* Funcion: genera_perm Fecha: 19/09/2012        */
/* Rutina que genera una permutacion             */
/* aleatoria                                     */
/*                                               */
/* Entrada:                                      */
/* int n: Numero de elementos de la              */
/* permutacion                                   */
/* Salida:                                       */
/* int *: puntero a un array de enteros          */
/* que contiene a la permutacion                 */
/* o NULL en caso de error                       */
/*************************************************/
```

```
int* genera_perm(int n)
{
      int *array_perm = NULL;
      int i;

      if (n <= 0)
            return NULL;

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

      if (array_perm == NULL)
                  return NULL;

      for (i=0; i<n ;i++)
            array_perm[i] = i;

      for (i=0; i<n ;i++)
            /* Puesto que se opera con un array de n elementos
               el limite superior del indice debe ser n-1 */
            swap(i,aleat_num(i,n-1),array_perm);

      return array_perm;
}
```

Esta rutina genera permutaciones aleatorias entre elementos de un array valiéndose de las funciones swap y aleat_num.

1.3 Resultados

La Figura 1 demuestra como la distribución de números al generar 100000 dígitos entre el 1 y el 18, es bastante equiprobable, con una desviación de resultados de no más de 100-150 ocurrencias de diferencia, siendo ésta del orden del 0,1%-0,15% del total.

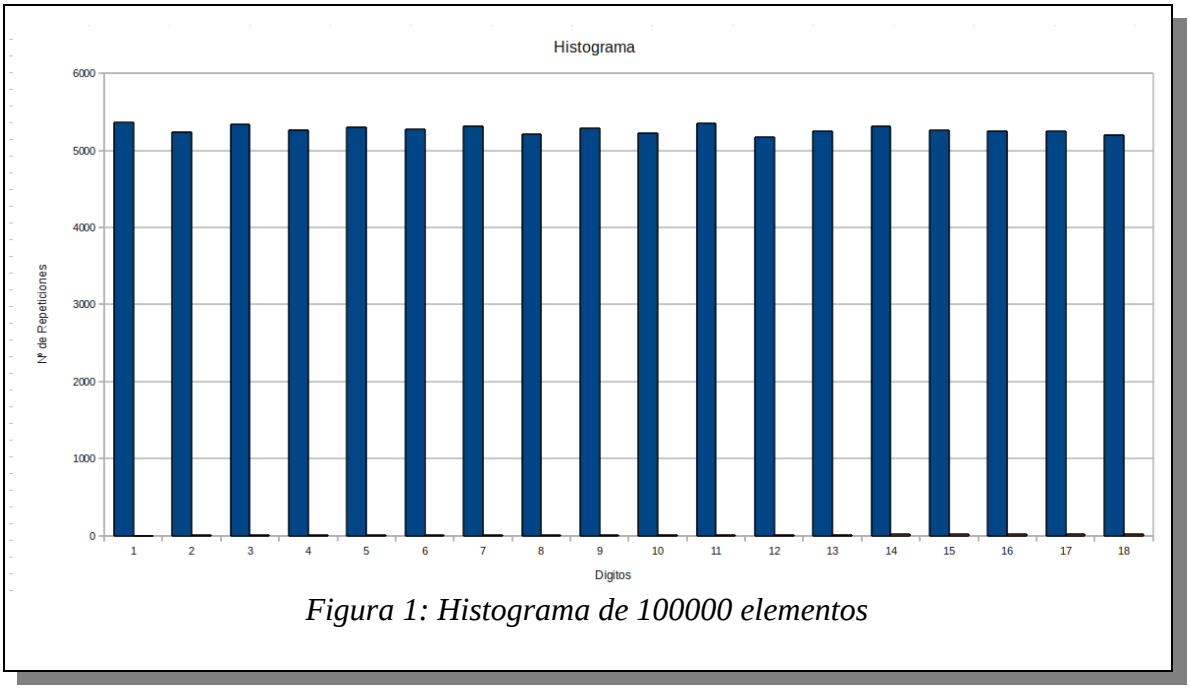

Figura 1: Histograma de 100000 elementos

Si se hace la misma prueba con un número de elementos 10 veces menor (Figura 2), se puede comprobar como la distribución sigue siendo bastante buena, con una variación de resultados del orden de las 50 repeticiones, siendo aproximadamente del 5%.

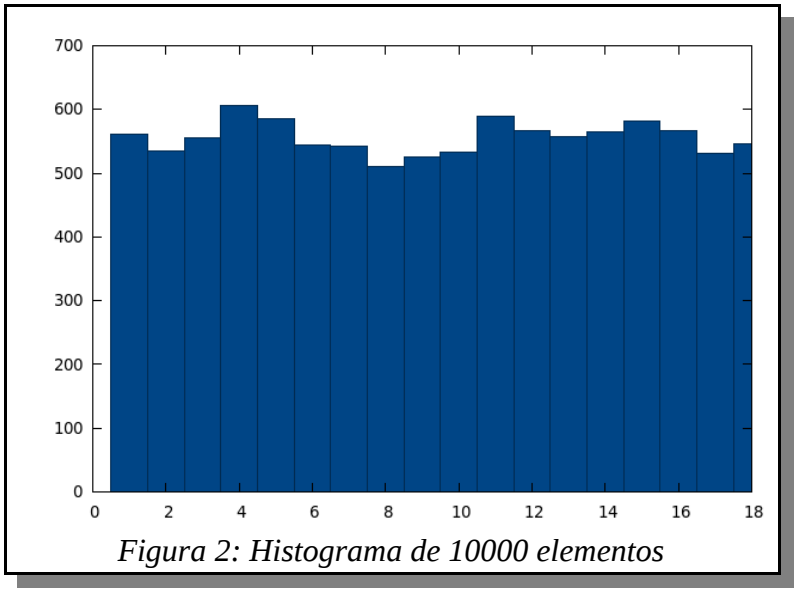

Figura 2: Histograma de 10000 elementos

En conclusión, el método para generar números pseudo-aleatorios pensado es aceptable, siendo mejor cuánto más alto sea el número de dígitos generados.

Al pasar la herramienta **Valgrind** a cada uno de los dos ejercicios se obtienen los siguientes resultados:

```
==4241==
==4241== HEAP SUMMARY:
==4241==     in use at exit: 0 bytes in 0 blocks
==4241==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4241==
==4241== All heap blocks were freed -- no leaks are possible
==4241==
==4241== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4241== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wuola@wuola-P67A-D3-B3:~/Descargas/practica1$
```

Figura 3: Resultado del Valgrind sobre el Ejercicio 1

```
==4241==
==4241== HEAP SUMMARY:
==4241==     in use at exit: 0 bytes in 0 blocks
==4241==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4241==
==4241== All heap blocks were freed -- no leaks are possible
==4241==
==4241== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4241== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wuola@wuola-P67A-D3-B3:~/Descargas/practica1$
```

Figura 4: Resultado del Valgrind sobre el Ejercicio 2

Como se puede observar en ambas capturas de pantalla, toda la memoria reservada es liberada (en este caso es 0 B), además de no existir fugas de memoria.

1.3.1 Resultados de las pruebas realizadas con ejercicio1

- **limSup < limInf:** La rutina aleat_num encuentra el error devolviendo -1.

```
./ejercicio1 -limInf 2 -limSup 1 -numN 2
Practica numero 1, apartado 1
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametros fuera del rango permitido
```

- **limInf < 0:** La rutina aleat_num encuentra el error devolviendo -1.

```
./ejercicio1 -limInf -2 -limSup 5 -numN 2
Practica numero 1, apartado 1
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametros fuera del rango permitido
```

- **numN = 0:** El programa no ejecuta el bucle para generar números aleatorios.

```
./ejercicio1 -limInf 9 -limSup 10 -numN 0
Practica numero 1, apartado 1
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametro numN fuera del rango permitido
```

- **numN < 0:** El programa encuentra el error evitando la ejecución del bucle.

```
./ejercicio1 -limInf 2 -limSup 5 -numN -2
Practica numero 1, apartado 1
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametro numN fuera del rango permitido
```

- **limInf = 0, limSup = 10, numN = 10:** El programa se ejecuta correctamente y devuelve 10 dígitos comprendidos en el rango [0,10].

```
./ejercicio1 -limInf 0 -limSup 10 -numN 10
Practica numero 1, apartado 1
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
0
7
```

```
4
0
5
7
10
10
1
4
```

1.3.2 Resultados de las pruebas realizadas con ejercicio2

- **tamanio = 0:** El programa encuentra el error evitando la generación de las permutaciones.

```
./ejercicio2 -tamanio 0 -numP 1
Practica numero 1, apartado 2
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

Parametros fuera del rango permitido
```

- **tamanio < 0:** El programa encuentra el error evitando la generación de las permutaciones.

```
./ejercicio2 -tamanio -1 -numP 1
Practica numero 1, apartado 2
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

Parametro tamanio fuera del rango permitido
```

- **numP = 0:** El programa encuentra el error evitando la generación de las permutaciones.

```
./ejercicio2 -tamanio 3 -numP 0
Practica numero 1, apartado 2
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

Parametros fuera del rango permitido
```

- **numP < 0:** El programa encuentra el error evitando la generación de las permutaciones.

```
./ejercicio2 -tamanio 3 -numP -1
Practica numero 1, apartado 2
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

Parametro numP fuera del rango permitido
```

- **tamanio = 10, numP = 10:** El programa se ejecuta correctamente y devuelve 10 permutaciones aleatorias de tamaño 10.

```
./ejercicio2 -tamanio 10 -numP 10
Practica numero 1, apartado 2
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

6 2 7 4 9 3 0 1 5 8
7 6 0 5 9 4 3 1 8 2
6 1 3 5 7 0 9 4 2 8
1 4 2 8 7 9 5 6 0 3
6 8 4 5 3 2 7 0 1 9
5 7 9 8 4 6 0 1 2 3
0 8 5 3 7 2 6 1 9 4
1 5 8 6 9 4 2 7 3 0
2 7 5 3 1 8 0 9 4 6
6 7 2 3 8 4 0 9 5 1
```

2. Bloque 2

2.1 Introducción

En este bloque se va a diseñar y codificar una rutina que genera una cantidad determinada de vectores con dígitos aleatorios del tamaño que se requiera. Para ello se hará uso de las funciones desarrolladas en el Bloque 1. Asimismo se diseñará una rutina de ordenación por selección la cual retornará el número de veces que se ejecuta su OB, que en este caso será la comparación de claves (CDC).

2.2 Código

```
/*************************************************/
/* Funcion: genera_permutaciones Fecha: 20/09/2012 */
/* Autores: Ari Handler Gamboa                    */
/*          Adrián Lorenzo Mateo                  */
/*                                                */
/* Funcion que genera n_perms permutaciones       */
/* aleatorias de tamanio elementos                */
/*                                                */
/* Entrada:                                       */
/* int n_perms: Numero de permutaciones           */
/* int tamanio: Numero de elementos de cada       */
/* permutacion                                    */
/* Salida:                                        */
/* int**: Array de punteros a enteros             */
/* que apuntan a cada una de las                  */
/* permutaciones                                  */
/* NULL en caso de error                          */
/*************************************************/
int** genera_permutaciones(int n_perms, int tamanio)
{
     int ** matriz_perm;
     int i, j;

     if (n_perms <=0 || tamanio <= 0)
          return NULL;

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

     if (matriz_perm == NULL)
          return NULL;
```

```
     for (i=0; i<n_perms ;i++){
            matriz_perm[i] = genera_perm(tamanio);
            if (matriz_perm[i] == NULL){
                /* Si no se ha podido reservar una de las
                   permutaciones se liberan todas las reservadas de
                   forma descendente */
           for (j=i; j>=i ;j++){
                 free(matriz_perm[j]);
           }
           free(matriz_perm);
           return NULL;
            }
       }

     return matriz_perm;
}
```

Esta función genera un total de **n_perms** permutaciones aleatorias de tamaño **tamanio** en una matriz que es devuelta como valor de retorno de la rutina. Para lograrlo se hacen **n_perms** llamadas a la función **genera_perm**, una por cada posición de la matriz.

```
/*****************************************************/
/* Funcion: SelectSort    Fecha: 26/10/2012          */
/* Autores: Adrián Lorenzo Mateo                     */
/*        Ari Handler Gamboa                         */
/*                                                   */
/* Funcion que ordena un array de enteros haciendo   */
/* uso del método SelectSort, devolviendo el número  */
/* de OBs durante la ejecución                       */
/*                                                   */
/* Entrada:                                          */
/* int tabla: Tabla a ordenar                        */
/* int ip: Primer elemento de la tabla               */
/* int iu: Ultimo elemento de la tabla               */
/* Salida:                                           */
/* int: Número de OBs ejecutadas o                   */
/* ERR en caso de error                              */
/*****************************************************/
int SelectSort(int* tabla, int ip, int iu)
{
     int i, j, ob_count, min;

    if((tabla==NULL)||(ip<0)||(iu<ip))
        return ERR;

    ob_count=0;
    /*recorre la tabla desde
```

```
      ip hasta iu*/
    for(i=ip; i<iu; i++){
        min=i;
        /*una vez seleccionado el elemento con indice ip
        itera entre los elementos restantes hasta iu*/
        for(j=i+1;j<=iu;j++){
            ob_count++;
            if(tabla[j]<tabla[min])/*OB*/
                min=j;
        }
        swap(i,min,tabla);
    }

      return ob_count;
}
```

Esta función recorre la tabla de elementos hasta el índice **iu**, seleccionado en primer lugar el elemento con índice **ip**, después obtiene el mínimo entre la selección y el resto de la tabla y finalmente hace la permutación entre el elemento seleccionado y el mínimo. A continuación realizará esas mismas operación de forma sucesiva hasta llegar al final de la tabla, finalmente ordenada.

2.3 Resultados

Al pasar la herramienta **Valgrind** a la ejecución del **ejercicio3.c** se obtienen los siguientes resultados:

```
==4306==
==4306== HEAP SUMMARY:
==4306==     in use at exit: 0 bytes in 0 blocks
==4306==   total heap usage: 11 allocs, 11 frees, 240 bytes allocated
==4306==
==4306== All heap blocks were freed -- no leaks are possible
==4306==
==4306== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4306== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wuola@wuola-P67A-D3-B3:~/Descargas/practica1$
```

Figura 5: Resultados de Valgrind sobre el Ejercicio 3

Se puede observar como toda la memoria reservada (240 B en 11 bloques) es liberada a la salida del programa, así como la inexistencia de fugas de memoria.

2.3.1 Resultados de las pruebas realizadas con ejercicio3

- **tamanio = 0:** La rutina genera_perm encuentra el error devolviendo -1.

```
./ejercicio3 -tamanio 0 -numP 10
Practica numero 1, apartado 3
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametros fuera del rango permitido
```

- **tamanio < 0:** La rutina genera_perm encuentra el error devolviendo -1.

```
./ejercicio3 -tamanio -3 -numP 10
Practica numero 1, apartado 3
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametros fuera del rango permitido
```

- **tamanio = 'a' y numP = 0:** La rutina genera_perm encuentra el error devolviendo -1.

```
./ejercicio3 -tamanio a -numP 0
Practica numero 1, apartado 3
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Parametros fuera del rango permitido
```

- **tamanio = 10, numP = 10:** El programa se ejecuta correctamente y devuelve 10 permutaciones aleatorias de tamaño 10 cada una.

```
./ejercicio3 -tamanio 10 -numP 10
Practica numero 1, apartado 3
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261

7 6 8 0 2 4 3 5 9 1
4 0 2 1 5 6 9 8 7 3
3 2 7 4 6 1 0 8 9 5
0 6 8 9 5 2 4 1 7 3
3 7 6 5 9 4 8 0 2 1
6 0 3 5 8 7 9 1 4 2
0 3 1 9 4 7 8 2 6 5
0 4 5 7 8 6 3 1 2 9
4 1 0 6 5 2 3 7 9 8
8 9 5 3 0 4 2 6 7 1
```

2.3.2 Resultados de las pruebas realizadas con ejercicio4

- **tamanio <= 0:** La rutina genera_perm encuentra el error devolviendo -1.

```
./ejercicio4 -tamanio 0
Practica numero 1, apartado 4
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
Error: No hay memoria
```

- **tamanio >=0:** El programa se ejecuta correctamente retornando la tabla ordenada mediante SelectSort.

```
./ejercicio4 -tamanio 8
Practica numero 1, apartado 4
Realizada por: Ari Handler Gamboa y Adrián Lorenzo Mateo
Grupo: 261
0     1     2     3     4     5     6     7
```

Al pasar la herramienta Valgrind al **ejercicio4** con un tamaño de tabla de 10 elementos y utilizando las banderas de ejecución correspondientes a la opción *verbose* y de identificación de fugas de memoria obtenemos el resultado mostrado en la Figura 6.

```
==4869==
==4869== HEAP SUMMARY:
==4869==     in use at exit: 0 bytes in 0 blocks
==4869==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==4869==
==4869== All heap blocks were freed -- no leaks are possible
==4869==
==4869== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4869== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wuola@wuola-P67A-D3-B3:~/Descargas/practica1_v2.3$
```

Figura 6: Resultado de Valgrind sobre el ejercicio 4

Lo cual nos lleva a la conclusión de que toda la memoria reservada es liberada al finalizar el programa y que no se han producido ningún tipo de fugas de memoria.

3. Bloque 3

3.1 Introducción

Este ultimo bloque se centra en la medida de los tiempos que tarda el algoritmo de ordenación utilizado, en este caso SelectSort. Para ello se define la estructura TIEMPO para almacenar en sus campos el tiempo medio de las OBs, el tamaño de las permutaciones, etc. Del mismo modo se crearán rutinas para la obtención de la información relativa a los tiempos así como funciones que escribirán los resultados en un fichero de salida.

3.2 Código

```
/*****************************************************/
/* Funcion: tiempo_medio_ordenacion Fecha: 26/09/2012   */
/* Autores: Adrián Lorenzo Mateo                        */
/*        Ari Handler Gamboa                            */
/*                                                      */
/* Funcion que calcula el tiempo medio de ordenacion    */
/* de una matriz de permutaciones aplicando el metodo   */
/* de ordenacion que se pasa como argumento,            */
/* rellenando una estructura tiempo con datos como      */
/* tiempo medio de ordenacion, promedio de Obs,         */
/* tamaño de las permutaciones, etc.                    */
/*                                                      */
/* Entrada:                                             */
/* pfunc_ordena metodo: Metodo de ordenacion que se     */
/* que se medirá.                                       */
/* int n_perms: Numero de permutaciones a utilizar      */
/* int tamanio: Numero de elementos de cada             */
/* permutacion                                          */
/* PTIEMPO ptiempo: Puntero a una estructura de tipo    */
/* TIEMPO que se rellenará una vez finalizada la        */
/* ejecucion de la rutina                               */
/* int n_veces: Numero de veces que se repetirá cada    */
/* ordenacion de la misma permutacion con el fin        */
/* de conseguir un retraso en el tiempo de ejecucion.   */
/*                                                      */
/* Salida:                                              */
/* ERR en caso de error y OK en el caso de que las      */
/* tablas se ordenen correctamente.                     */
```

```
/***************************************************/
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int tamanio,
                              PTIEMPO ptiempo,
                              int n_veces)
{
     int max_ob, min_ob, actual_ob;
     clock_t t_inicio, t_final;
     double t_medio, medio_ob;
     int ** matriz_perm;
     int i, j;
     int * perm_aux=NULL;

     /* Comprobacion de los parametros de entrada */
     if (metodo == NULL || n_perms <= 0 || tamanio <= 0 || ptiempo
== NULL)
          return ERR;

     /* Reserva y comprobacion de memoria de la matriz de
permutaciones */
     if ((matriz_perm = genera_permutaciones(n_perms, tamanio)) ==
          NULL)
          return ERR;

     /* Inicializacion de variables */
     medio_ob = max_ob = actual_ob = t_medio = 0;
     min_ob = INT_MAX;

     /* Reserva de memoria de la permutacion auxiliar que servira
     * para crear el retardo necesario reordenandola n_veces */
     if ((perm_aux = (int *)malloc(tamanio*sizeof(int))) == NULL){
               /* Se libera la matriz de permutaciones */
               for (i=0; i< n_perms ;i++){
                    free(matriz_perm[i]);
               }
               free(matriz_perm);
               return ERR;
     }

     /* Iteracion de cada una de las permutaciones de la matriz */
     for (i=0; i<n_perms ;i++){

               /* Se copia la permutacion original */
               memcpy(perm_aux, matriz_perm[i], tamanio);

          t_inicio = clock();

          /* Se ordena la permutación original n_veces veces */
          for (j=0; j<n_veces; j++){
              if((actual_ob = metodo(matriz_perm[i], 0, tamanio-1))
                    == ERR){
                /* Se liberan la matriz de permutaciones */
```

```
                        for (i=0; i< n_perms ;i++){
                              free(matriz_perm[i]);
                        }
                        free(matriz_perm);
                  return ERR;
                }
                /* La permutacion es restaurada */
                memcpy(matriz_perm[i], perm_aux, tamanio);
          }

          t_final = clock();

          /* Se evalua la cuenta de OBs actual */
          if (actual_ob < min_ob)
                min_ob = actual_ob;
          if (actual_ob > max_ob)
                max_ob = actual_ob;

          /* Se promedian las OB */
          medio_ob += (double)actual_ob/n_perms;

          /* Se promedian los tiempos */
          t_medio += (double) (t_final - t_inicio)/(double)
                                    CLOCKS_PER_SEC/n_perms;
     }

     /* Se completa la estructura tiempo */
     ptiempo->n_perms = n_perms;
     ptiempo->tamanio = tamanio;
     ptiempo->medio_ob = medio_ob;
     ptiempo->min_ob = min_ob;
     ptiempo->max_ob = max_ob;
     ptiempo->tiempo = t_medio;

     /* Se liberan la matriz de permutaciones y la permutacion
auxiliar*/
     for (i=0; i< n_perms ;i++){
           free(matriz_perm[i]);
     }
     free(matriz_perm);
     free(perm_aux);

     return OK;
}
```

Esta función evalúa el tiempo medio de ordenación, número de OBs máxima, mínima y promedio ejecutadas aplicando el método de ordenación **metodo** a un número **n_perms** permutaciones de tamaño **tamanio** y rellenando una estructura TIEMPO con los resultados obtenidos. Además se utilizará un retardo para conseguir que los tiempos

de ejecución resulten significativo; para ello se utiliza el parámetro **n_veces** para ordenar un número determinado de veces la misma permutación antes de pasar a la siguiente.

```
/*****************************************************/
/* Funcion: genera_tiempos_ordenacion Fecha: 27/09/2012 */
/* Autores: Adrián Lorenzo Mateo                     */
/*       Ari Handler Gamboa                          */
/*                                                   */
/* Funcion que escribe en un fichero los tiempos y OBs  */
/* evaluadas de un metodo de ordenacion con entradas    */
/* de tamaño incremental.                            */
/*                                                   */
/* Entrada:                                          */
/* pfunc_ordena metodo: Metodo de ordenacion que se     */
/* que se medirá.                                    */
/* char* fichero: Fichero de salida de las mediciones   */
/* int num_min: Tamaño inicial de las permutaciones     */
/* int num_max: Tamaño maximo de las permutaciones      */
/* int incr: Ratio de crecimiento del tamaño de las     */
/* permutaciones                                     */
/* int n_perms: Numero de permutaciones a utilizar      */
/* int n_veces: Numero de veces que se repetirá cada    */
/* ordenacion de la misma permutacion con el fin        */
/* de conseguir un retraso en el tiempo de ejecucion.   */
/*                                                   */
/* Salida:                                           */
/* ERR en caso de error y OK en el caso de que las      */
/* tablas se ordenen correctamente y se escriba en el   */
/* fichero.                                          */
/*****************************************************/
short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                                int num_min, int num_max,
                                int incr, int n_perms, int n_veces)
{
    int i,j;
    PTIEMPO tabla_tiempo=NULL;
    int tamanio_tabla_tiempo;

    /* Control de errores de los parametros de entrada */
     if (metodo == NULL || fichero == NULL || num_max<num_min ||
          num_min <0 || incr<=0 || n_perms <=0)
      return ERR;

    /* Se añade 1 puesto que el rango esta incluido en el limite
superior */
    tamanio_tabla_tiempo = ((num_max-num_min)/incr)+1;

    if( (tabla_tiempo=(TIEMPO
*)malloc(tamanio_tabla_tiempo*sizeof(TIEMPO)))== NULL)
         return ERR;
```

```
    /* Se miden los tiempos de cada tamaño y se rellena la tabla de
tiempos */
    for (i=num_min, j=0; i<=num_max; i+=incr, j++){
         if (tiempo_medio_ordenacion(metodo, n_perms, i,
&tabla_tiempo[j], n_veces) == ERR){
             free(tabla_tiempo);
             return ERR;
         }
        fprintf(stdout, "\nPermutaciones de tamaño %d finalizadas\n",
i);
    }
    /* Escritura de la tabla de tiempos en el fichero mediante la
funcion guarda_tabla_tiempos */
    if (guarda_tabla_tiempos(fichero, tabla_tiempo,
tamanio_tabla_tiempo) == ERR){
           free(tabla_tiempo);
           return ERR;
    }
    free(tabla_tiempo);
    return OK;
}
```

Esta función mide los tiempos de ejecución y OBs al aplicar el método de ordenación pasado como argumento sobre tablas de **n_perms** permutaciones de tamaño incremental empezando por **num_min** hasta **num_max**, aumentando dicho tamaño en cada iteración a razón del parámetro **incr**. Una vez recogidos los tiempos, almacenados en una tabla **PTIEMPO,** son escritos en **fichero** haciendo uso de la función **guarda_tabla_tiempos**, descrita y comentada a continuación.

```
/*****************************************************/
/* Funcion: guarda_tabla_tiempos Fecha: 28/09/2012      */
/* Autores: Adrián Lorenzo Mateo                       */
/*       Ari Handler Gamboa                             */
/*                                                     */
/* Funcion que imprime en un fichero la informacion que */
/* contiene una tabla de tiempos.                     */
/*                                                     */
/* Entrada:                                            */
/* char* fichero: Fichero de salida de las mediciones   */
/* PTIEMPO tiempo: Tabla de tiempos a imprimir          */
/* int N: Numero de elementos de la tabla de tiempos    */
/*                                                     */
/* Salida:                                             */
/* ERR en caso de error y OK en el caso de que se la    */
/* tabla de tiempos se escriba correctamente.          */
/*****************************************************/
```

```
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int N)
{
    FILE * fichero_out = NULL;
    int i;

    /* Control de errores de los parametros de entrada */
    if (fichero == NULL || tiempo == NULL || N <= 0)
           return ERR;

    if ((fichero_out = fopen(fichero, "w")) == NULL)
           return ERR;
    /* Impresion de la tabla de tiempos */
    for (i=0; i<N ;i++){
     fprintf(fichero_out, "%d\t%.7f\t%.7f\t%d\t%d\n",
           tiempo[i].tamanio, tiempo[i].tiempo, tiempo[i].medio_ob,
           tiempo[i].max_ob, tiempo[i].min_ob);
    }

    fclose(fichero_out);

    return OK;
}
```

Esta función guarda la información de tamaño de la permutaciones, tiempo medio de ordenación y promedio, mínimo y máximo de OBs ejecutadas, contenido en la tabla de tiempos tiempo en un fichero de salida fichero.

3.3 Resultados

3.3.1 Resultados de las pruebas realizadas con tiempo_medio_ordenacion

Con el fin de poder realizar pruebas sobre esta función, se ha desarrollado el siguiente código:

```
int main(int argc, char** argv)
{
  int i, tamanio, n_veces, n_perms;
  short ret;
  PTIEMPO tiempo;

  srand(time(NULL));
```

```
  if (argc != 7) {
    fprintf(stderr, "Error en los parametros de
                     entrada:\n\n");
  }

  printf("Practica numero 1, apartado 5\n");
  printf("Realizada por: Vuestros nombres\n");
  printf("Grupo: Vuestro grupo\n");

  /* comprueba la linea de comandos */
  for(i = 1; i < argc ; i++) {
    if (strcmp(argv[i], "-n_perms") == 0) {
      n_perms = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-tamanio") == 0) {
      tamanio = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-n_veces") == 0) {
      n_veces = atoi(argv[++i]);
    } else {
      fprintf(stderr, "Parametro %s es incorrecto\n",
                     argv[i]);
    }
  }

  tiempo = (PTIEMPO)malloc(sizeof(TIEMPO));

  ret = tiempo_medio_ordenacion(SelectSort, n_perms,
tamanio, tiempo, n_veces);

  if (ret == ERR) {
     printf("Error en la funcion
     genera_tiempos_ordenacion\n");
    exit(-1);
  }



printf("\n# de permutaciones: %d\n", tiempo->n_perms);
printf("Tamaño de cada permutacion: %d\n",
          tiempo->tamanio);
printf("Tiempo promedio: %f\n", tiempo->tiempo);
printf("Maximo, minimo y promedio de OBS: %d %d %f\n\n",
       tiempo->max_ob, tiempo->min_ob, tiempo->medio_ob);

free(tiempo);

  return 0;
}
```

Así, se han realizado las siguientes pruebas:

- **n_veces <= 0:** La rutina encuentra el error devolviendo -1.

```
./prueba -n_perms 10 -tamanio 10 -n_veces -10
Practica numero 1, funcion tiempo_medio_ordenacion
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion genera_tiempos_ordenacion
```

- **n_perms <= 0:** La rutina encuentra el error devolviendo -1.

```
./prueba -n_perms -10 -tamanio 10 -n_veces 10
Practica numero 1, funcion tiempo_medio_ordenacion
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion genera_tiempos_ordenacion
```

- **tamanio <= 0:** La rutina encuentra el error devolviendo -1.

```
./prueba -n_perms 10 -tamanio -10 -n_veces 10
Practica numero 1, funcion tiempo_medio_ordenacion
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion genera_tiempos_ordenacion
```

- **tamanio=100, n_perms=100, n_veces=100:** La rutina realiza correctamente la medición e imprime los resultados por pantalla.

```
./prueba -n_perms 100 -tamanio 100 -n_veces 100
Practica numero 1, funcion tiempo_medio_ordenacion
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261


# de permutaciones: 100
Tamaño de cada permutacion: 100
Tiempo promedio: 0.001200
Maximo, minimo y promedio de OBS: 4950 4950 4950.000000
```

Al pasar la herramienta Valgrind a dicha rutina con un tamaño de tabla de 100 elementos, 100 permutaciones y un retardo de 100 ordenaciones adicionales obtenemos los resultados de la Figura 7.

```
==6494==
==6494== HEAP SUMMARY:
==6494==     in use at exit: 0 bytes in 0 blocks
==6494==   total heap usage: 103 allocs, 103 frees, 40,832 bytes allocated
==6494==
==6494== All heap blocks were freed -- no leaks are possible
==6494==
==6494== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==6494== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wuola@wuola-P67A-D3-B3:~/Descargas/practica1_v2.3$
```

Figura 7: Resultado de Valgrind sobre la rutina tiempo_medio_ordenacion

3.3.2 Resultados de las pruebas realizadas con ejercicio5

- **num_max < num_min:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5 -num_min 100 -num_max 10 -incr 10 -numP 10
-fichSalida salida.dat -retardo 10
Practica numero 1, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion Time_Ordena
```

- **incr = 0:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5 -num_min 100 -num_max 1000 -incr 0 -numP 10
-fichSalida salida.dat -retardo 10
Practica numero 1, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion Time_Ordena
```

- **numP < 0:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5 -num_min 100 -num_max 1000 -incr 10 -numP -3
-fichSalida salida.dat -retardo 10
Practica numero 1, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Error en la funcion Time_Ordena
```

- **num_min=100, num_max=1000, incr=100, numP=100, retardo = 100:** El programa genera el fichero **salida.dat** con las OBs y tiempos medios de cada permutación.

```
./ejercicio5 -num_min 100 -num_max 1000 -incr 100 -numP
100 -fichSalida salida.dat -retardo 100
Practica numero 1, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Permutaciones de tamaño 100 finalizadas

Permutaciones de tamaño 200 finalizadas

Permutaciones de tamaño 300 finalizadas

Permutaciones de tamaño 400 finalizadas

Permutaciones de tamaño 500 finalizadas

Permutaciones de tamaño 600 finalizadas

Permutaciones de tamaño 700 finalizadas

Permutaciones de tamaño 800 finalizadas

Permutaciones de tamaño 900 finalizadas

Permutaciones de tamaño 1000 finalizadas
Salida correcta
```

Con el fichero **salida.dat** y la herramienta **gnuplot** representamos el número medio de OBs con respecto al tamaño de permutación y lo comparamos con el resultado teórico, es decir $A_{SS}(N) = N^2/2$:

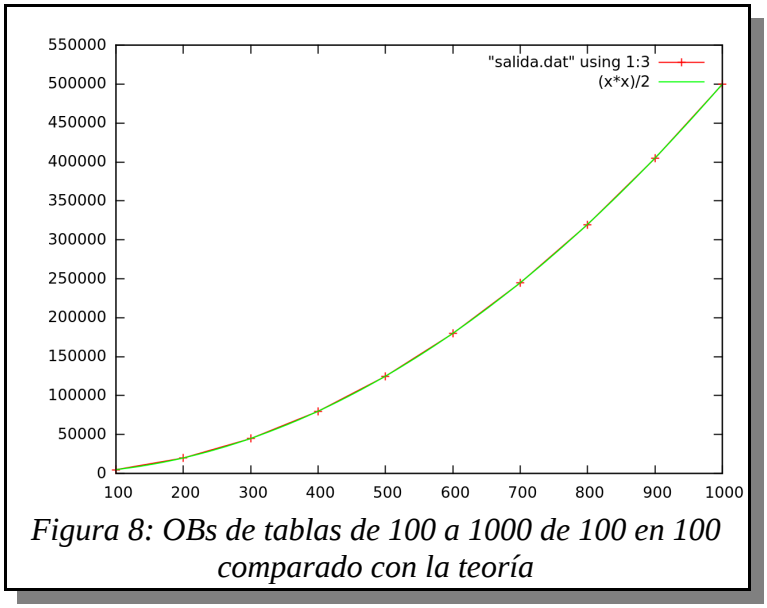

Figura 8: OBs de tablas de 100 a 1000 de 100 en 100 comparado con la teoría

La Figura 8 nos demuestra que efectivamente el número de OBs para el caso medio del algoritmo SelectSort es igual al teórico, es decir, $A_{SS}(N) = N^2/2$.

Si ahora utilizamos como datos los tiempos medios de ordenación con los mismos tamaños de tablas, obtenemos la gráfica de la Figura 9, que es comparada con la función N^2/X, donde X es un factor de normalización los suficientemente grande como para que ambas curvas se representen a la vez. Éste cambio de denominador en el resultado teórico no es lo suficientemente significativo con respecto al crecimiento de la función cuadrática, por lo que la gráfica generada es correcta desde el punto de vista de comparación con los resultados prácticos.

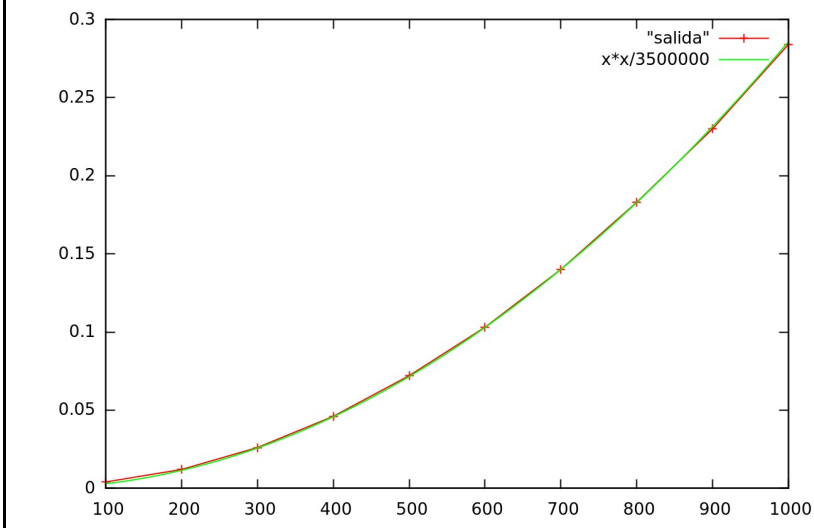

Figura 9: Tiempos de ordenación para tablas de tamaño de 100 a 1000

- num_min=1000, num_max=10000, incr=1000, numP=100, retardo = 10: El programa genera el fichero salida.dat con las OBs y tiempos medios de cada permutación.

```
./ejercicio5 -num_min 1000 -num_max 10000 -incr 1000 -numP
100 -fichSalida salida.dat -retardo 10
Practica numero 1, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261

Permutaciones de tamaño 1000 finalizadas
```

```
Permutaciones de tamaño 2000 finalizadas

Permutaciones de tamaño 3000 finalizadas

Permutaciones de tamaño 4000 finalizadas

Permutaciones de tamaño 5000 finalizadas

Permutaciones de tamaño 6000 finalizadas

Permutaciones de tamaño 7000 finalizadas

Permutaciones de tamaño 8000 finalizadas

Permutaciones de tamaño 9000 finalizadas

Permutaciones de tamaño 10000 finalizadas
Salida correcta
```

Si volvemos a representar los datos obtenidos con esta prueba de la manera que se realizó la anterior, obtenemos las gráficas de las Figuras 10 y 11.

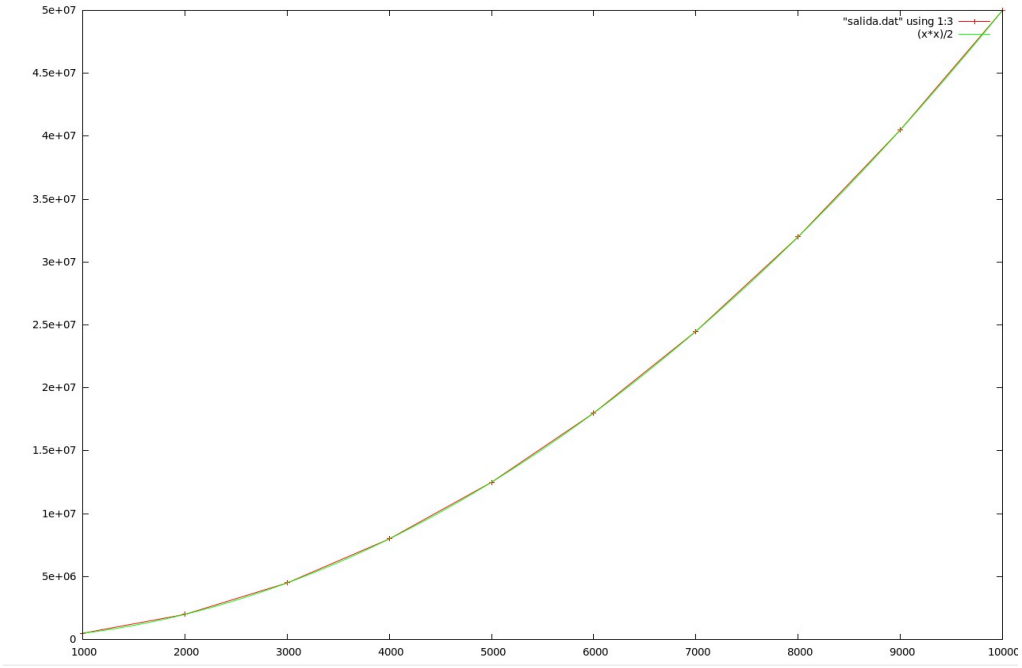

Figura 10: OBs promedio para tablas de tamaño desde 1000 a 10000

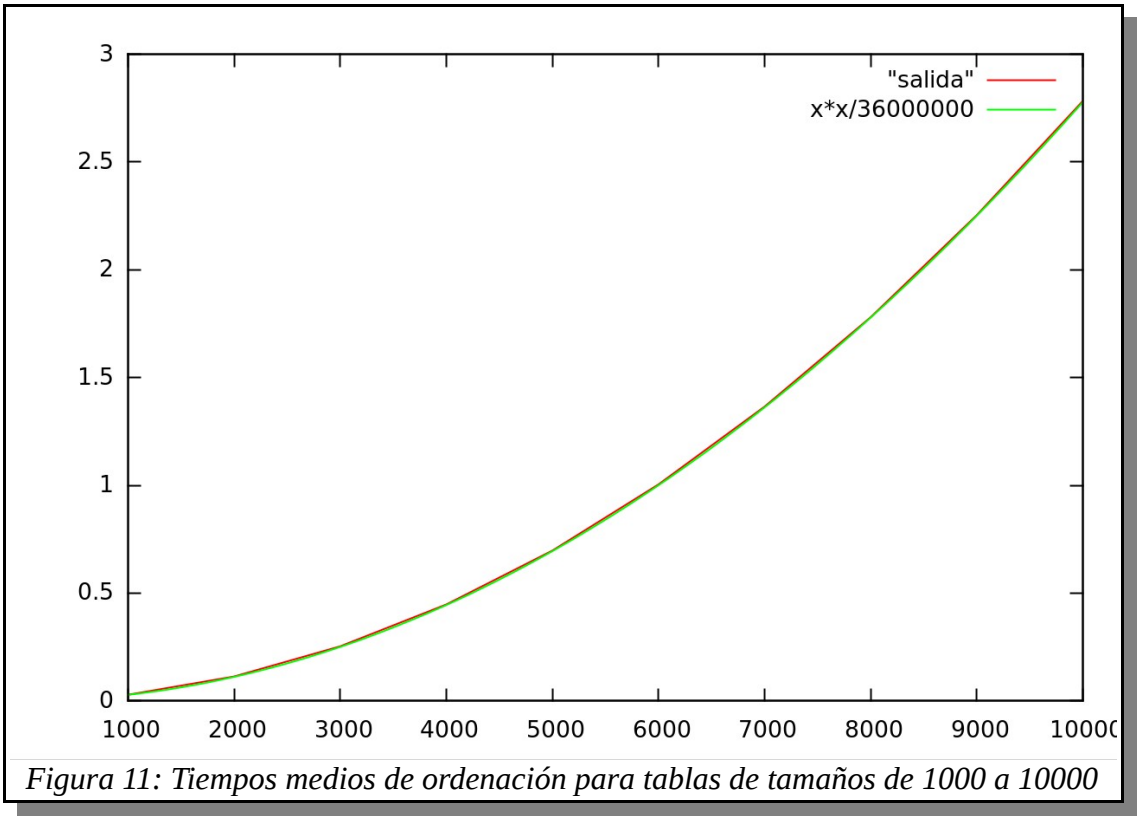

Figura 11: Tiempos medios de ordenación para tablas de tamaños de 1000 a 10000

Los resultados obtenidos con dichas gráficas son equivalentes a las de la prueba anterior, volviendo a confirmar la validez de los resultados prácticos con respecto a los estudiados en la teoría.

4. Cuestiones Optativas

1. Valgrind + memcheck.

Memcheck es un módulo de Valgrind que permite localizar errores en el manejo de memoria. La detección de estos errores se realiza a través de la simulación, por lo que es posible que no se encuentren algunos errores de manejo de memoria. Debido a esto, se recomienda a la hora de desarrollar un determinado módulo, realizar un programa test que haga uso intensivo del módulo para minimizar cualquier posibilidad de error. Los errores detectados por Memcheck son los siguientes:

- Uso de memoria no inicializada.

- Lectura o escritura de memoria liberada con free.
- Lectura o escritura fuera del área pedida con malloc.
- Lectura o escritura inapropiada de la pila.
- Fugas de memoria.
- Correspondencia entre el número de mallocs y frees.

5. Cuestiones

1. Justifica tu implementación de aleat_num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.

Hemos utilizado el siguiente método para generar números pseudo-aleatorios:

```
(int)((rand()/(RAND_MAX+1.))*(sup-inf+1))+inf
```

El método se divide en tres partes, la primera es la generación de un número pseudo-aleatorio mediante la función **rand()** limitado en el rango [0, 1) debido a la división entre RAND_MAX+1. La segunda parte multiplica dicho valor por un número que se encuentre aproximadamente en la mitad del rango (sup-inf +1), de este modo se normaliza el anterior valor acotándolo entre 0 y el número intermedio. Sin embargo, es necesario sumar el límite inferior al resultado con el fin de que se obtenga un valor superior a este.

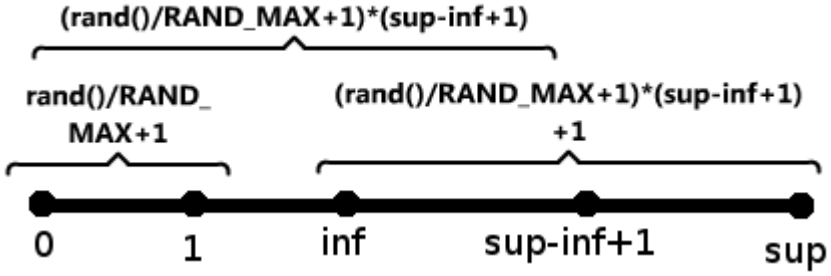

Figura 2. Explicación gráfica del método.

Un método alternativo podría ser el siguiente:

```
inf + (rand()%((sup -inf) +1))
```

Este método es muy parecido, el único cambio significativo consiste en que se hace el módulo del índice intermedio entre el superior y el inferior. Teniendo en cuenta que por ejemplo, rand()%N genera valores entre 0 y N-1. El mínimo valor obtenido es 0, por lo que si se le suma el límite inferior (inf) del intervalo, inf será el mínimo valor obtenido. Si se quiere obtener como máximo valor el límite superior (sup) es necesario reemplazar N por (sup-inf)+1. Si no se sumara 1 el límite superior sería (sup-1).

Al sustituir la operación de división (la más costosa de las operaciones aritméticas desde el punto de vista de ejecución) y la invocación de la constante RAND_MAX por la ligera y rápida ejecución de la función módulo se consigue un incremento en el tiempo de ejecución de la rutina.

2. Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el por qué ordena bien) del algoritmo SelectSort.

SelectSort es un algoritmo de ordenación basado en comparación de claves cuyo funcionamiento es el siguiente:

1. Buscar el mínimo elemento de la lista.
2. Intercambiarlo con el primero.
3. Buscar el mínimo en el resto de la lista.
4. Intercambiarlo con el segundo.
5. …

De esta forma, busca el mínimo elemento entre una posición *i* y el final de la lista, e intercambia el mínimo con el elemento de la posición *i*.

Así, el pseudo-código de SelectSort sería algo parecido a lo siguiente:

```
para i=1 hasta n-1
    minimo = i;
    para j=i+1 hasta n
        si lista[j] < lista[minimo] entonces
            minimo = j
        fin si
    fin para
```

```
    intercambiar(lista[i], lista[minimo])
fin para
```

3. ¿Cuál es la operación básica de SelectSort?

La operación básica es (tabla[j]<tabla[min]) ya que cumple las condiciones para convertirse en una OB, aparece en el bucle más interno del algoritmo, es una de las instrucciones que más veces se ejecuta y es la operación representativa del algoritmo al tratarse de una comparación de clave.

4. Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WSS(n) y el caso mejor BSS(n) de SelectSort. Utilizar la notación asinstótica (O; Θ; o; Ω, etc) siempre que se pueda.

El mejor y el peor caso no dependen del tamaño de la tabla, sino de que varíe el número de instrucciones que se ejecute en cada ocasión. En el caso del algoritmo SelectSort el tiempo de ejecución siempre va a ser el mismo ($N^2/2 + O(N)$) ya que tiene que recorrer la tabla de principio a fin, realizando el mismo número de operaciones básicas, aún encontrándose en el caso peor o mejor.

6. Conclusiones

Los generadores de números pseudo aleatorios son de vital importancia en muchas aplicaciones criptográficas para la generación de claves y códigos de acceso . Por esta razón es necesario tomar ciertas consideraciones a la hora de desarrollar una rutina de estas características, como que los números obtenidos deben aproximarse a las propiedades estadísticas ideales de uniformidad e independencia, la rapidez de ejecución de la rutina, su simpleza, la capacidad de portabilidad, etc.

No se han encontrado excesivas dificultades en el desarrollo del ejercicio3 propuesto del bloque 2, puesto que era bastante parecido al procedimiento que se tomó al codificar la función genera_perm del bloque anterior.

Como mejoras posibles conviene señalar el uso del mismo índice de bucle que el del bucle principal (variable i) a la hora de liberar cada una de las permutaciones reservadas cuando se encuentra un error; sin embargo, hemos utilizado otro diferente (variable j) con el fin de que fuera más sencillo de comprender.

Por otro lado, se ha podido observar que el algoritmo SelectSort realiza siempre el mismo número de comparaciones de clave en función del tamaño n de la tabla a ordenar:

$$c(n)=\frac{n^2-n}{2}$$

Esto significa que el número de comparaciones c(n) no depende del orden de los términos, si no únicamente del número de términos.

Esta práctica como conjunto ofrece los medios necesarios para medir y analizar distintos algoritmos de ordenación, siempre y cuando los parámetros de los mismos se adapten a la definición `typedef int (* pfunc_ordena)(int*, int, int);` localizada en la librería **ordena.h**. Del mismo modo esta práctica también ha servido de introducción en el manejo de **gnuplot**, una herramienta para generar gráficas de funciones y datos. Los resultados obtenidos a través de la

medición del algoritmo de ordenación se han empleado para generar las gráficas de tiempo y trabajo. Una vez realizadas se han comparado con el crecimiento teórico del algoritmo, permitiendo detectar posibles desviaciones en su eficiencia.

Anexo A: Opciones de uso de Valgrind + Memcheck

Memcheck es la herramienta por defecto de Valgrind, pero si se desea se puede especificar la opción **`--tool=memcheck`**.

Para ejecutar memcheck sobre el programa que comúnmente se ejecuta como

```
./test arg1 arg2 ... argn
```

se hace de la siguiente manera

```
valgrind [opciones] ./test arg1 arg2 ... argn
```

Las principales opciones de uso de memcheck son las siguientes:

- **`--leak-check=<no|summary|yes|full> [default: summary]`** Cuando está activada busca memory leaks. Si es summary se indica el número de memory leaks, y si es yes o full se entrega información detallada de cada fuga.
- **`--leak-resolution=<low|med|high> [default: low]`** Muestra información detallada de las fugas (La opción por defecto es la recomendada) .
- **`--num-callers=<number> [default: 4]`** Muestra información detallada de las fugas (La opción por defecto es la recomendada) .
- **`--freelist-vol=<number> [default: 5000000]`** Cuando un programa libera memoria usando free, no se libera automáticamente, sino que se marca como inaccesible y se agrega a una cola de bloques libres. El propósito de esta cola es retardar lo más posible el momento en

que los bloques vuelven a recircular. Esta opción indica el tamaño máximo de la cola en bytes. El incrementar el valor incrementa la memoria usada por memcheck, pero puede detectar manejos de memoria inválidos que de otra manera no se podría

- **`--undef-value-errors=<yes|no> [default: yes]`** Indica si se verifican las variables no inicializadas. No se recomienda deshabilitar esta opción.