Análisis de Algoritmos 2022/2023

Práctica 1

Ignacio Sánchez, Fabio Desio. Grupo 1272.

Código	Gráficas	Memoria	Total

Índice:

1. Introducción.

En esta práctica debemos realizar tres bloques de trabajo en los que implementaremos una serie de funciones que servirán como base a prácticas posteriores. Además, las rutinas se relacionan entre ellas conforme avanza la práctica. Como objetivo principal tenemos el de observar los tiempos de ejecución y complejidad del algoritmo de ordenación *SelectSort* así como de *SelectSortInv*, que actúan sobre tablas de números enteros desordenadas y las ordenan ascendente y descendentemente respectivamente. De esta manera, la práctica se basará en gran parte en la creación de números aleatorios para formar permutaciones.

2. Objetivos

Procedemos a indicar el trabajo que hemos realizado en cada apartado:

2.1 Apartado 1: Random number

En este primer apartado se nos ha pedido desarrollar la función *random_num*, la cual devuelve un número aleatorio entre un límite inferior y superior reduciendo lo máximo posible el sesgo. Debíamos comprobar esta rutina con el archivo *exercise1.c*.

2.2 Apartado 2: Generate Perm

A partir del pseudocódigo proporcionado, se ha desarrollado una rutina que devuelve un puntero a una permutación aleatoria de enteros (empleando *random_num* para la generación de estos) de tamaño dado. Para comprobar su funcionamiento disponemos del archivo *exercise2.c*.

2.3 Apartado 3: Generate Permutations

Con el número de permutaciones a generar (*n_perms*) y el tamaño de estas (*N*) como argumentos, la función *generate_permutations* crea una matriz de permutaciones cuyo correcto funcionamiento hemos podido comprobar con el archivo *exercise3.c*.

2.4 Apartado 4: SelectSort y función min

En este apartado se ha desarrollado el algoritmo de ordenación *SelectSort*. Con el array de enteros y los índices del primer y último elemento de este como argumentos, *SelectSort* devuelve un puntero al array ordenado. Para implementar el algoritmo hemos hecho uso de otra función llamada *min* que devuelve el índice del elemento mínimo entre los índices de los argumentos y cuenta el número de operaciones básicas realizadas. Su funcionamiento ha sido comprobado con el fichero *exercise4.c.*

2.5 Apartado 5: Tiempos de ejecución

En este apartado debemos implementar tres funciones diferentes. La primera es *average_sorting_time* que recibe un puntero a una función de ordenación, el número de permutaciones a generar y ordenar por el método que se use, el tamaño de cada permutación y un puntero a la estructura *time_aa* definida en el archivo *times.h*. El retorno de esta función devuelve *OK* si se han rellenado todos los campos de la estructura con éxito o *ERR* si se ha producido algún error. La segunda función *generate_sorting_times* escribe en el fichero file los tiempos medios, y los números promedio, mínimo y máximo de veces que se ejecuta la *OB* en la ejecución del algoritmo de ordenaci´on method con n perms permutaciones de tama˜nos en el rango desde num min hasta num max, ambos incluidos, usando incrementos de tama˜no incr. La rutina devolverá el valor *ERR* en caso de error y *OK* en caso contrario. La última

función *save_time_table* la encontraremos dentro de la función anterior y en la encargada de imprimir en el fichero todos los campos de la estructura *time_aa* que se han rellenado para cada incremento del tamaño de permutación.

2.6 Apartado 6: SelectSortInv

En este apartado debemos implementar una función *SelectSortInv* con el mismo prototipo que *SelectSort*, y cuyo comportamiento difiere únicamente en que debe devolver el array de enteros pero ordenado en orden descendente.

3. Herramientas y metodología

En todos los apartados hemos utilizado el entorno de desarrollo Ubuntu, el compilador gcc y el editor de código Visual Studio Code, además de la herramienta Valgrind para comprobar la correcta utilización de recursos y memoria. Cabe destacar, que al inicio de cada implementación y tras llamadas anidadas a funciones, realizamos el pertinente control de errores, por lo que se obvia su mención en la explicación de la solución. Además, en cada implementación usamos como base el pseudocódigo proporcionado por la profesora y discutido en grupo en la clase, al cual podemos acceder desde moodle.

3.1 Apartado 1: Random Number

Metodología: hemos utilizado los comandos sort y uniq al ejecutar el código con las banderas -n y -c respectivamente para ordenar numéricamente los resultados únicos. De esta manera, nos genera un archivo donde aparecen por columnas, la frecuencia con la que aparece un número y el número correspondiente. Posteriormente,

utilizamos la herramienta gnuplot para visualizar cuán equiprobable es la distribución. Cabe destacar que en este apartado hemos hecho uso de un libro de referencia[1] para extraer ideas sobre la implementación.

Solución: comenzamos con el pertinente control de errores, con el objetivo de prevenir casos erróneos como la situación en la que los límites sean negativos o el ínfimo supere al supremo. Después, devolvemos el ínfimo si los dos límites coinciden o el número aleatorio entre los límites que resulta de la fórmula hallada apoyándonos en el libro mencionado.

3.2 Apartado 2: Generate Perm

Metodología: nos hemos servido del pseudocódigo proporcionado en el enunciado de la práctica como base para nuestra implementación. Podemos afirmar que este pseudocódigo se basa en el algoritmo de reproducción aleatoria de Fisher-Yates[2].

Solución: reservamos memoria para un array de enteros y lo rellenamos con enteros de manera que estén ordenados. Posteriormente, ejecutamos un bucle for de manera que mediante la rutina random_num encontremos un índice aleatorio entre el actual y el último de la tabla e intercambiamos los elementos con dichos índices. Al finalizar, habremos desordenado todos los elementos de nuestra tabla creando una permutación y la devolveremos con un puntero.

---

[1] Iserles, A. (1989). Numerical recipes in C—the art of scientific computing, by WH Press, BP Flannery, SA Teukolsky and WT Vetterling. Pp 735.£ 27· 50. 1988. ISBN 0-521-35465-X (Cambridge University Press). *The Mathematical Gazette*, *73*(464), 167-170.

[2] Mezclar una matriz determinada con el algoritmo de reproducción aleatoria de Fisher-Yates – Acervo Lima. (s/f). Acervolima.com. Recuperado el 20 de octubre de 2022, de *https://es.acervolima.com/mezclar-una-matriz-determinada-con-el-algoritmo-de-reproduccion-aleatoria-de-fisher-yates/*

3.3 Apartado 3: Generate Permutations

Metodología: no usamos ninguna herramienta que difiera de las ya especificadas al inicio. Sin embargo, hemos declarado una nueva función privada en el fichero permutations.h y la implementamos en el permutations.c.

Solución: reservamos memoria para un array de punteros que va a almacenar todas las permutaciones. Asignamos después una permutación a cada elemento del array de punteros con la rutina generate_perm dentro de un bucle for, y terminamos devolviendo el doble puntero después de haber considerado todos los casos en los que puede haber un error. Además, implementamos una función free_perms que recibe como argumentos un puntero a un array de permutaciones y el tamaño de las permutaciones y libera la memoria reservada para ellas.

3.4 Apartado 4: SelectSort y función min

Metodología: utilizamos el editor de código para la implementación y el compilador para el correcto funcionamiento del programa, y el gnuplot en última instancia para graficar los resultados. Además de las dos funciones a implementar, hemos declarado otra función privada en un nuevo fichero swap.h, y la implementamos en el correspondiente fichero swap.c. En consecuencia, se debió modificar el archivo makefile para poder compilar estos dos archivos y usarlos en las diferentes implementaciones.

Solución: comenzamos con la función SelectSort. Utilizamos un bucle for para hallar el índice del menor elemento de la tabla mediante la función min e intercambiamos el elemento de la tabla con el índice correspondiente a la iteración del bucle con el menor elemento de la tabla. De esta manera, en cada iteración

incrementamos el índice que marca el inicio de la subtabla, quedando a la izquierda los elementos ya ordenados y a la derecha la subtabla a ordenar. La función min asigna asigna en un primer lugar el índice del menor elemento al índice que recibe como inicio de la tabla, y después itera sobre toda la tabla asignando al mínimo el índice de cualquier elemento que sea menor. Así, al finalizar el bucle, devolverá el índice del menor elemento. La función swap utiliza una variable temporal para almacenar uno de los elementos y poder intercambiar dos elementos de la tabla entre sí.

3.5 Apartado 5: Tiempos de ejecución

Metodología: utilizaremos la herramienta gnuplot una vez comprobado el funcionamiento del programa para realizar las gráficas que se piden en apartados sucesivos dentro de la memoria.

Solución: en este apartado debemos implementar tres funciones diferentes. En la primera, average_sorting_time, gereramos las permutaciones necesarias y después las ordenamos con la rutina método que se pasa como puntero, la cual devolverá la cantida de operaciones básicas realizadas. Necesitamos además medir el tiempo de ejecución del algoritmo de ordenación, para lo que utilizamos la función clock()[3] de la biblioteca times.h. Una vez terminado, queda rellenar los campos de la estructura time_aa. En la segunda función, generate_sorting_times calculamos la cantidad de veces que se realiza el incremento de tamaño de la permutación realizando la diferencia entre el máximo y el mínimo, para después dividirlo entre el incremento y acabar sumando 1, pues el tamaño inicial cuenta como uno válido. Además, al ser el campo n_times de tipo entero, redondeará siempre hacia el número decimal menor. Queda entonces reservar memoria

[3] C library function - clock(). (s/f). Tutorialspoint.com. Recuperado el 20 de octubre de 2022, de https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm

para array de punteros a time_aa de tamaño n_times, y realizar un bucle for ejecutando la función del apartado anterior para cada tamaño de permutación y elemento del array de punteros. Terminamos llamando a la función save_time_table. Esta función abrirá un fichero para escribir cada campo de la estructura times en una misma línea, de manera que cada línea corresponderá a un tamaño de permutación, el cual incrementa según cambiamos de línea.

3.6 Apartado 6: SelectSortInv

Metodología: como adición a las herramientas mencionadas al inicio, usamos el gnuplot para visualizar los resultados de forma gráfica. En este apartado hacemos uso también del fichero swap.c y su correspondiente swap.h.

Solución: el procedimiento que usamos en este apartado es muy parecido al de SelectSort, sin embargo el algoritmo se utiliza en orden inverso. En este caso el bucle decrementa una variable que comienza teniendo el valor del último índice, mientras que este último se mantiene intacto. De esta manera, iremos colocando los elementos ya ordenados al final de la tabla, y las subtablas que se van generando progresivamente van siempre desde el primer elemento de la tabla hasta el elemento con el índice de la variable que se decrementa en el bucle. Al finalizar, la función SelectSortInv devuelve un puntero a un array de números enteros ordenado descendentemente.

4. Código fuente

4.1 Apartado 1: Random number

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

/* Function: random_num Date:                   */
```

```c
/* Authors: Ignacio Sánchez and Fabio Desio        */

/*                                                 */

/* Routine that generates a random number          */

/* between two given numbers                       */

/*                                                 */

/* Input:                                          */

/* int inf: lower limit                            */

/* int sup: upper limit                            */

/* Output:                                         */

/* int: random number                              */

/* ERR in case of error                            */

/*-------------------------------------------------*/

int random_num(int inf, int sup)

{

 if (inf < 0 || sup > RAND_MAX || inf > sup)

 {

   return ERR;

 }

 if (inf == sup)
```

```
   return sup;

 return inf + (int)(((double)rand() * (sup - inf + 1.0) / (RAND_MAX
+ 1.0)));

}
```

4.2 Apartado 2: Generate Perms

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

/* Function: generate_perm Date:                     */

/* Authors: Ignacio Sánchez and Fabio Desio          */

/*                                                   */

/* Routine that generates a random permutation       */

/*                                                   */

/* Input:                                            */

/* int n: number of elements in the permutation      */

/* Output:                                           */

/* int *: pointer to integer array                   */

/* that contains the permutation                     */

/* or NULL in case of error                          */

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

```c
int *generate_perm(int N)

{

 int *perm = NULL;

 int i, tmp, random;



 if (N < 1 || N > RAND_MAX)

   return NULL;

 /* Reservamos memoria para el array de manera dinámica */

 perm = (int *)malloc(N * sizeof(perm[0]));

 if (perm == NULL)

 {

   return NULL;

 }

 for (i = 0; i < N; i++)

 {

   perm[i] = i + 1;

 }



 for (i = 0; i < N; i++)
```

```
 {

   random = random_num(i, N-1);

   tmp = perm[i];

   perm[i] = perm[random];

   perm[random] = tmp;

 }



 return perm;

}
```

4.3 Apartado 3: Generate Permutations

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

/* Function: generate_permutations Date:         */

/* Authors: Ignacio Sánchez and Fabio Desio      */

/*                                               */

/* Function that generates n_perms random        */

/* permutations with N elements                  */

/*                                               */

/* Input:                                        */
```

```c
/* int n_perms: Number of permutations              */

/* int N: Number of elements in each permutation    */

/* Output:                                          */

/* int**: Array of pointers to integer that point   */

/* to each of the permutations                      */

/* NULL en case of error                            */

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

int **generate_permutations(int n_perms, int N)

{

 int i, j;

 int **perms = NULL;



 if (n_perms < 1 || n_perms > RAND_MAX || N < 1 || N > RAND_MAX)

   return NULL;



 perms = (int **)malloc(n_perms * sizeof(perms[0]));



 if (perms == NULL)

   return NULL;
```

```
 for (i = 0; i < n_perms; i++)

 {

   perms[i] = generate_perm(N);



   if (perms[i] == NULL) {

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

       free(perms[j]);

     }

     free(perms);

     return NULL;

   }

 }



 return perms;

}
```

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

/* Function: free_permutations Date:               */
```

```
/* Authors: Ignacio Sánchez and Fabio Desio         */

/*                                                  */

/* Function that frees the allocated memory for     */

/* a number of permutations                         */

/*                                                  */

/* Input:                                           */

/* int **perms: Matrix of permutations              */

/* int n_perms: Number of permutations              */

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

void free_permutations(int  **perms, int n_perms) {

 int i;



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

   free(perms[i]);

 }



 free(perms);

}
```

4.4 Apartado 4: SelectSort y función min

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

/* Function: SelectSortDate:                             */

/* Authors: Ignacio Sánchez and Fabio Desio              */

/*                                                       */

/* Function that sorts a disordered integer table        */

/* ascendantly                                           */

/*                                                       */

/* Input:                                                */

/* int *array: integer table                             */

/* int ip: first index of the array                      */

/* int iu: last index of the array                       */

/* Output:                                               */

/* int ob: number of ob performed during the sorting     */

/* ERR in case of error                                  */

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

int SelectSort(int *array, int ip, int iu)

{

 int i, ob = 0, minimum;
```

```
 if (array == NULL || ip < 0 || iu < ip)

 {

   return ERR;

 }



 for (i = ip; i < iu; i++)

 {

   minimum = min(array, i, iu, &ob);

   swap(array + i, array + minimum);

 }



 return ob;

}
```

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

/* Function: min Date:                          */

/* Authors: Ignacio Sánchez and Fabio Desio     */

/*                                              */
```

```c
/* Function that saves the index of the smallest  */

/* element between the elements corresponding     */

/* to the indexes ip and iu                       */

/*                                                */

/* Input:                                         */

/* int *array: table of integers                  */

/* int ip: first index of the array               */

/* int iu: last index of the array                */

/* int *ob: pointer that saves the number of      */

/* comparisons made                               */

/* Output:                                        */

/* int min: index of the smallest element         */

/* ERR in case of error                           */

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

int min(int *array, int ip, int iu, int *ob)

{

 int i, min;



 if (array == NULL || ip < 0 || iu < ip)
```

```
 {

   return ERR;

 }



min = ip;

 for (i = ip; i <= iu; i++)

 {

   (*ob)++;

   if (array[i] < array[min])

   {

     min = i;

   }

 }



 return min;

}
```

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

/* Function: swap Date:                                 */
```

```c
/* Authors: Ignacio Sánchez and Fabio Desio                */

/*                                                         */

/* Function that swaps two elements of an integer table */

/*                                                         */

/* Input:                                                  */

/* int *p1: element of the integer table                   */

/* int *p2: element of the integer table                   */

/* int iu: last index of the array                         */

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

void swap (int *p1, int *p2)

{

   int tmp=0;



   tmp=(*p1);

   (*p1)=(*p2);

   (*p2)=tmp;



}
```

4.5 Apartado 5: Tiempos de ejecución

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

/* Function: average_sorting_time Date:              */

/*                                                   */

/* Function that sorts a number of permutations      */

/* with a method and stores time spent and number    */

/* of ob in a structure                              */

/*                                                   */

/* Input:                                            */

/* pfunc_sort metodo: pointer to the method that sorts */

/* the permutation                                   */

/* int n_perms: Number of permutations to sort       */

/* int N: Number of elements of each permutation     */

/* PTIME_AA ptime: pointer to the struct time_aa     */

/* Output:                                           */

/* OK                                                */

/* ERR in case of error                              */

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

```c
short average_sorting_time(pfunc_sort metodo, int n_perms, int N,
PTIME_AA ptime)

{

 int i;

 int ob, min_ob = INT_MAX, max_ob = 0;

 long suma_obs = 0;

 int **perms = NULL;

 double tiempo;

 long double suma_tiempo = 0;

 clock_t begin, end;



 /* Control de errores */

 if (metodo == NULL || n_perms < 1 || N < 1 || ptime == NULL)

   return ERR;



 /* Rellenamos los dos primeros campos de la estructura time */

 ptime->N = N;

 ptime->n_elems = n_perms;
```

```
/* Generamos las permutaciones */

perms = generate_permutations(n_perms, N);



if (perms == NULL)

  return ERR;



 /* Ordenamos cada permutación con un bucle desde la pos 0 a
n_perms */

for (i = 0; i < n_perms; i++)

{

  begin = clock();

  if (begin == -1)

  {

    free_permutations(perms, n_perms);

    return ERR;

  }



  ob = metodo(perms[i], 0, N - 1);

  if (ob == ERR)
```

```
{
  free_permutations(perms, n_perms);
  return ERR;
}


end = clock();
if (end == -1)
{
  free_permutations(perms, n_perms);
  return ERR;
}


/* Almacenamos las obs */
suma_obs += ob;


/* Almacenamos valor mínimo y máximo */
if (min_ob > ob)
  min_ob = ob;
```

```
    if (max_ob < ob)

      max_ob = ob;



    /* Almacenamos los tiempos */

      tiempo = (double)(end - begin) / CLOCKS_PER_SEC * 1e9; /*
Multiplicamos por 1e9 para extraer el tiempo en nanosegundos */

    suma_tiempo += tiempo;

  }



  /* Almacenamos los campos que faltan */

  ptime->time = suma_tiempo / (n_perms * 1.0);

  ptime->average_ob = suma_obs / (n_perms * 1.0);

  ptime->min_ob = min_ob;

  ptime->max_ob = max_ob;



  free_permutations(perms, n_perms);

  return OK;

}
```

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

/* Function: generate_sorting_times Date:         */

/*                                                */

/* Function that calls average_sorting_time to sort a   */

/* number of permutations that increments its size      */

/* according to a parameter and prints the sorting      */

/* times in a file                                */

/*                                                */

/* Input:                                         */

/* pfunc_sort metodo: pointer to the method that sorts  */

/* the permutation                                */

/* char *file: pointer to the name of the file          */

/* int num_min: minimum size of the permutations        */

/* int num_max: maximum size of the permutations        */

/* int incr: increment to the size of the permutation   */

/* int n_perms: Number of permutations to sort          */

/* Output:                                        */

/* OK                                             */

/* ERR in case of error                           */
```

```c
/***************************************************/

short generate_sorting_times(pfunc_sort method, char *file, int
num_min, int num_max, int incr, int n_perms)

{

 TIME_AA *time = NULL;

 int i, j, n_times;

 short status = ERR;



 /* Control de errores inicial */

 if (method == NULL || file == NULL || num_min < 1 || num_min >
num_max || n_perms < 1)

   return ERR;



 /* TODO: Comentar esto en la memoria */

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

 /* Guardamos memoria para los tiempos de ejecución */

 time = (TIME_AA *)malloc(n_times * sizeof(time[0]));



 if (time == NULL)
```

```
   return ERR;



  /*  Generamos  en  este  array  todos  los  tiempos  medios  y  los
imprimimos */

 for  (i  =  0,  j  =  num_min;  i  <  n_perms  &&  j  <=  num_max;  i++,  j  +=
incr)

 {

   status = average_sorting_time(method, n_perms, j, &time[i]);



   if (status == ERR)

   {

     free(time);

     return ERR;

   }

 }



 status = save_time_table(file, time, n_times);



 if (status == ERR) {
```

```
   free(time);

   return ERR;

 }



 free(time);

 return OK;

}



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

/* Function: save_time_table Date:                       */

/*                                                       */

/* Function that prints the elements of the struct for   */

/* each size of permutation in a file                    */

/*                                                       */

/* Input:                                                */

/* char *file: pointer to the name of the file           */

/* PTIME_AA ptime: pointer to the struct time_aa         */

/* int n_times: number of times that the increment is    */

/* realized                                              */
```

```
/* Output:                                                  */

/* OK                                                       */

/* ERR in case of error                                     */

/*----------------------------------------------------------*/

short save_time_table(char *file, PTIME_AA ptime, int n_times)

{

 FILE *pf;

 int i;



 if (ptime == NULL)

   return ERR;



 pf = fopen(file, "w");



 if (pf == NULL)

   return ERR;



 for (i = 0; i < n_times; i++)

 {
```

```
        if   (fprintf(pf,   "%d   %.2f   %.2f   %d   %d\n",   ptime[i].N,
ptime[i].time,ptime[i].average_ob,ptime[i].min_ob,ptime[i].max_ob)<
0)

    {

      return ERR;

    }

  }



 fclose(pf);



 return OK;

}
```

4.6 Apartado 6: SelectSortInv

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

/* Function: SelectSortInv Date:                  */

/* Authors: Ignacio Sánchez and Fabio Desio       */

/*                                                */

/* Function that sorts a disordered integer table */
```

```c
/* in descending order                                    */

/*                                                        */

/* Input:                                                 */

/* int *array: integer table                              */

/* int ip: first index of the array                       */

/* int iu: last index of the array                        */

/* Output:                                                */

/* int ob: number of ob performed during the sorting */

/* ERR in case of error                                   */

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

int SelectSortInv(int *array, int ip, int iu)

{

 int i, ob = 0, minimum;



 if (array == NULL || ip < 0 || iu < ip)

 {

   return ERR;

 }
```

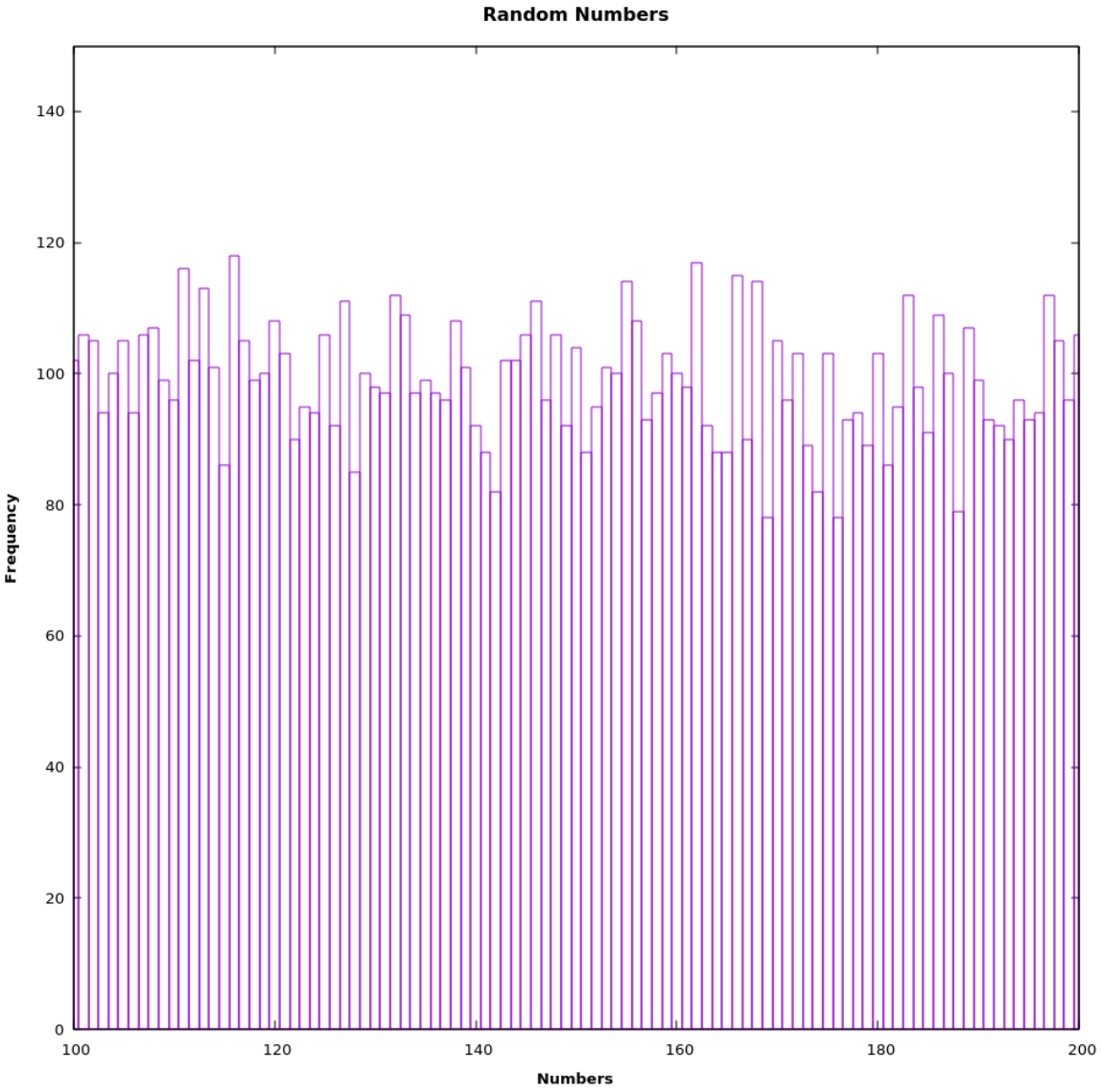
```
 for (i = iu; i > 0; i--)

 {

   minimum = min(array, ip, i, &ob);

   swap(array + i, array + minimum);

 }

 return ob;

}
```

5. Resultados, Gráficas

5.1 Apartado 1: Random Numbers

Ejecutamos el archivo exercise1.c con el siguiente comando desde la terminal: *./exercise1 -limInf 100 -limSup 200 -numN 10000 | sort -n | uniq -c > 'datos.txt'*. Elegimos los límites 100 y 200 para ver que la distribución es correcta aun no empezando en el cero, y 1000 para la cantidad de números a generar pues se trata de una muestra lo suficientemente significativa para formar conclusiones. El resultado esperado es de una frecuencia de 100 por cada número, lo cual concuerda en gran medida con lo que se muestra en la *Gráfica 1*, variando todos los resultados entre el 80 y 120.

Gráfica 1: Random Numbers



5.2 Apartado 2: Generate Perm

En este apartado se ha ejecutado el fichero exercise2.c con el comando:

./exercise2 -size 10 -numP 10 > 'datos.txt' para generar diez permutaciones con valores entre el 1 y el 10 en el archivo datos.txt. Aun habiendo probado con otros límites,

vemos a continuación un ejemplo de salida que confirma el buen funcionamiento de la rutina:

9 3 7 1 10 5 2 8 6 4

1 8 6 5 7 9 4 10 2 3

1 6 9 7 2 5 8 4 10 3

8 1 9 5 2 10 3 7 6 4

7 6 10 1 8 9 3 4 2 5

7 9 10 5 3 8 4 2 1 6

1 7 10 9 6 4 5 2 8 3

7 3 8 1 10 4 2 9 5 6

2 5 7 1 10 3 6 8 9 4

7 3 1 6 4 2 5 8 10 9

5.3 Apartado 3: Generate Permutations

De manera análoga al apartado 2, hemos utilizado el fichero exercise3.c para generar las pertinentes permutaciones. Como se trata de un archivo muy similar al exercise2.c con la única diferencia de utilizar la función *generate_permutations* en vez de ejecutar iterativamente *generate_perm*, debemos de esperar una salida similar para una misma entrada.

./exercise3 -size 10 -numP 10 > 'datos.txt' devuelve:

6 7 8 9 1 4 5 10 3 2

9 1 6 2 3 10 7 4 8 5

7 1 6 8 10 4 9 5 3 2

2 8 1 7 6 4 5 3 10 9

9 8 5 1 2 6 3 7 4 10

3 5 10 7 2 8 4 1 6 9

7 3 6 2 4 5 1 10 9 8

5 2 6 4 10 3 8 1 7 9

7 10 2 4 6 1 5 3 9 8

7 9 3 5 1 4 2 10 8 6

5.4 Apartado 4: Select Sort y función min

Usando el archivo exercise4.c como referencia, hemos comprobado el correcto funcionamiento de nuestras rutinas:

./exercise4 -size 10 > 'datos.txt' nos devuelve:

1	2	3	4	5	6	7	8	9	10	(Select Sort)
10	9	8	7	6	5	4	3	2	1	(SS Inv),

tal y como esperábamos.

5.5 Apartado 5: Tiempo de ejecución

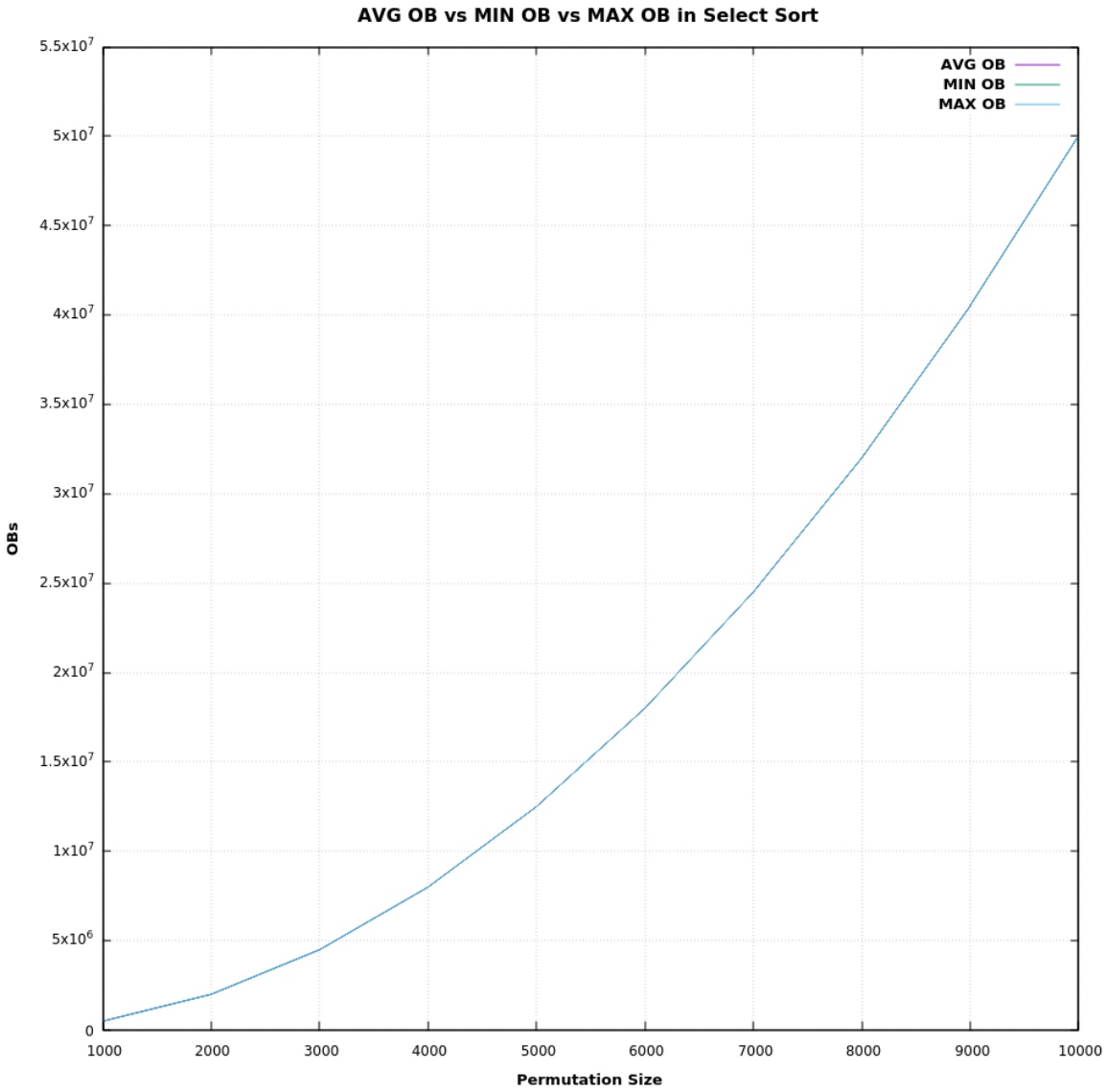
En estos dos últimos apartados hemos utilizado el main exercise5.c como fichero de pruebas. A partir de una misma salida, computada a partir del comando:

./exercise5 -num_min 1000 -num_max 10000 -incr 1000 -numP 100 -outputFile 'ej5.txt',

hemos graficado empleando la herramienta gnuplot los resultados.

En la *Gráfica 2* observamos que el número de OBs máximo y mínimo (y por consiguiente el medio también) coinciden para cualquier tamaño de permutación. Esto es lo esperado si tenemos en cuenta la naturaleza del algoritmo, que como hemos visto en la clase de teoría, tiene complejidad cuadrática sin importar el tamaño de la entrada.

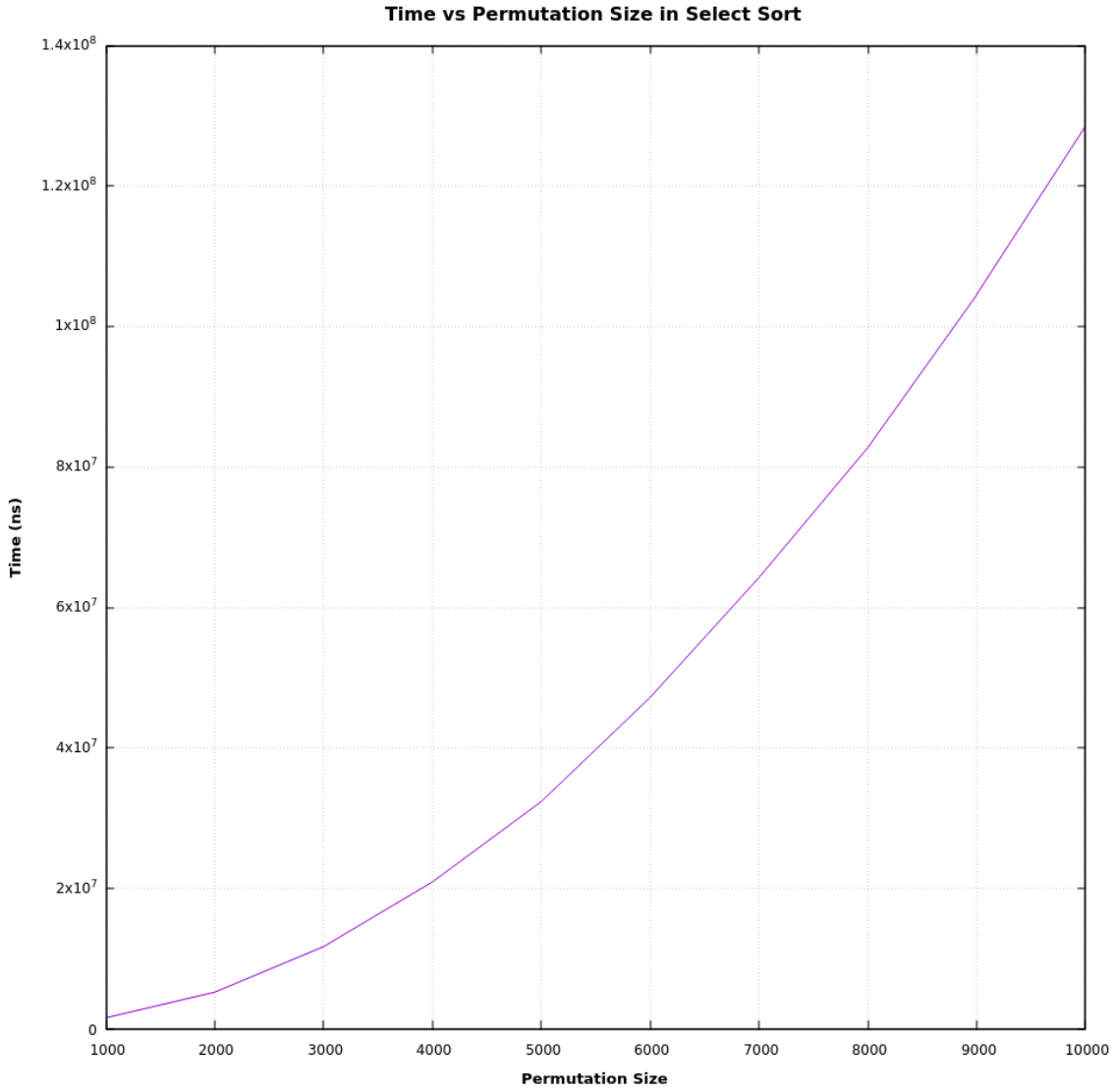
Gráfica 2: AVG OB vs MIN OB vs MAX OB in Select Sort



En la *Gráfica 3* se han recogido los tiempos de ejecución en nanosegundos (cuestión de escala) para cada tamaño de permutación en función del número medio de operaciones básicas realizadas. De la misma manera que con la *Gráfica 2*, no existen

picos ni valles anómalos en la gráfica lo cual indica un crecimiento uniforme del tiempo de ejecución en función del tamaño de la permutación. A modo de observación, vemos que, aun con una escala distinta, el crecimiento es exponencial en ambos casos.

Gráfica 3: Time vs Permutation Size in Select Sort

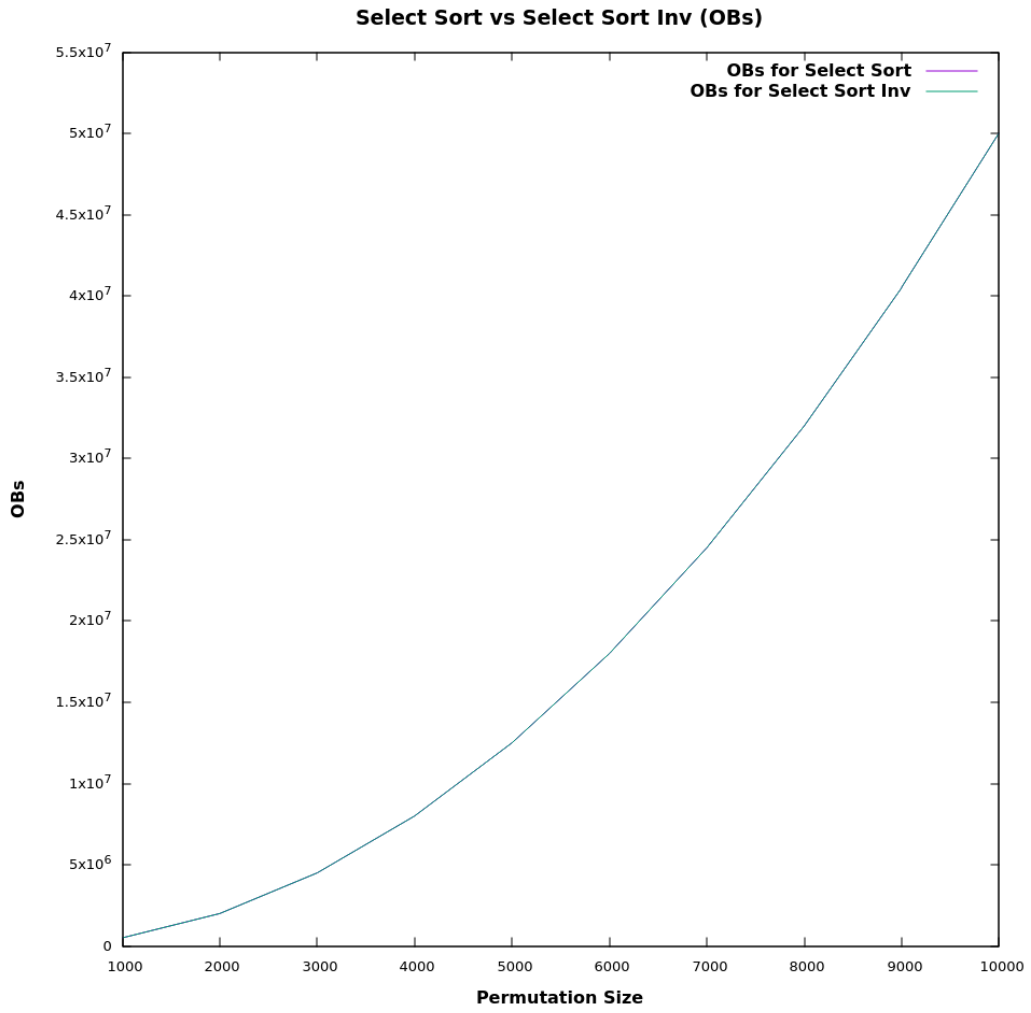


5.6 Apartado 6: Select Sort Inv

En este apartado se han generado las salidas de prueba con el mismo comando descrito en el Apartado 5, pero empleando el algoritmo de ordenación, Select Sort Inverso.

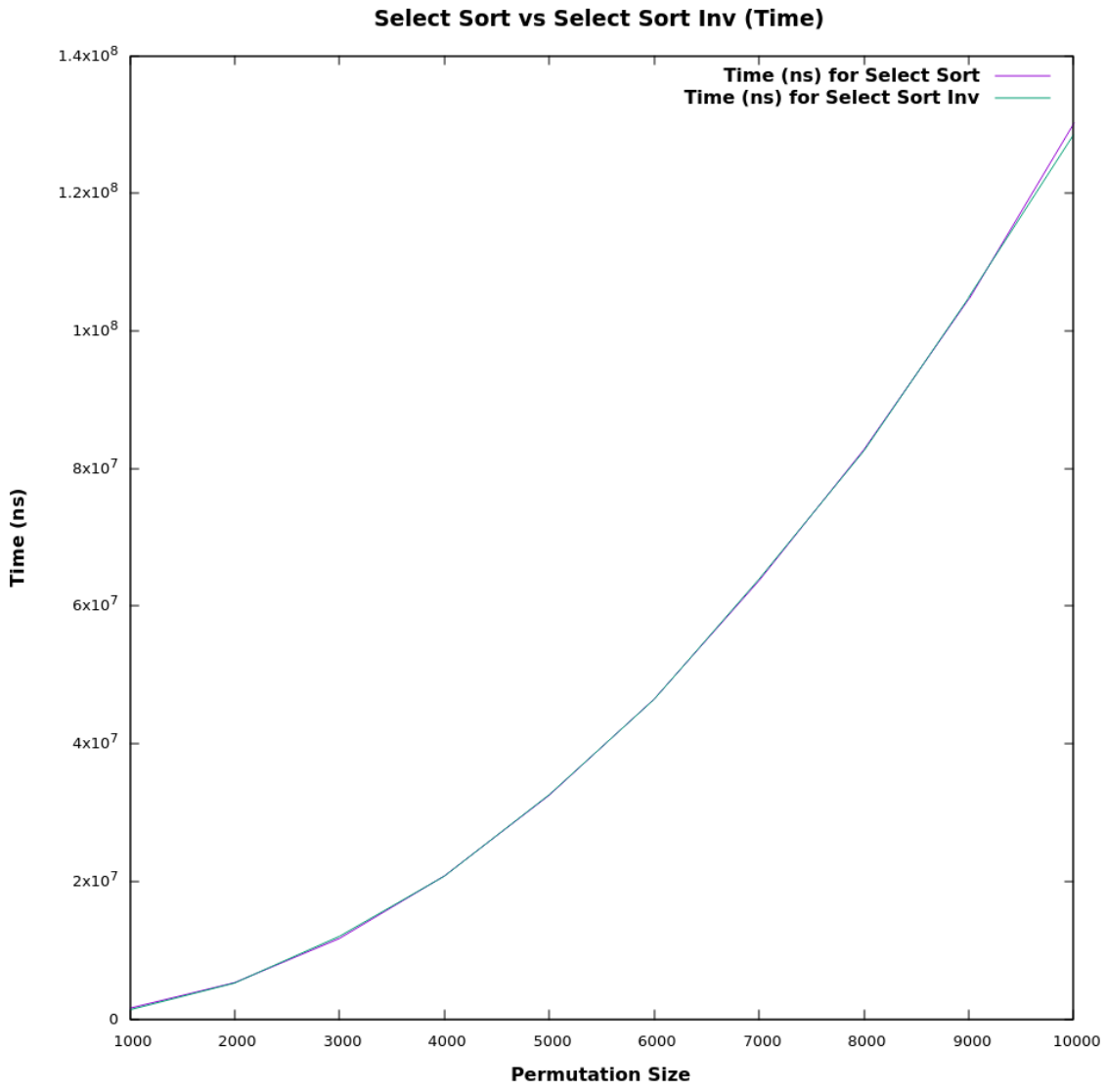
En la *Gráfica 4* podemos apreciar que el tiempo de ejecución medido en OBs para ambos algoritmos coinciden. Si nos centramos en el componente teórico, esta sería la salida esperada pues se trata del mismo algoritmo que el Select Sort, difiriendo únicamente en que el menor elemento se introduce al final de la tabla.

Gráfica 4: Select Sort vs Select Sort Inv (OBs)



En cuanto al tiempo de ejecución en nanosegundos, vemos en la *Gráfica 5* que las líneas de tendencia para ambos algoritmos prácticamente se solapan. Solamente se aprecia una pequeña diferencia en cortos tramos debido a que se obtienen de diferentes muestras de datos.

Gráfica 5: Select Sort vs Select Sort Inv (Time)



6. Respuesta a las preguntas teóricas.

6.1 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.

Nuestra idea de esta función se basa en que la distribución de los números aleatorios sea lo más equiprobable posible. Para ello, utilizaremos operaciones como la división y la suma o resta de los límites para que el número se encuentre en el intervalo establecido. Hemos tomado la idea del libro "Numerical recipes in C: the art of scientific computing". Otro método alternativo de generación de números sería el uso de la operación módulo, de manera que el número resultante se halle siempre dentro de un intervalo determinado. Sin embargo, con este método existe el problema de que si el número aleatorio máximo es muy grande, el sesgo en la distribución es muy grande.

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

El algoritmo de SelectSort funciona bien porque en cada paso va encontrando el mínimo de subtablas que van reduciendo su tamaño, para situar este en la primera posición de la subtabla. De esta manera, tras hallar el mínimo de una tabla y situarlo en la primera posición, el mínimo de la siguiente subtabla resultará ser el número inmediatamente mayor que se encuentre.

6.3 ¿Por qué el bucle exterior de SelectSort no actúa sobre el último elemento de la tabla?

Porque en el momento que el algoritmo llega al último elemento de la tabla, el algoritmo ya ha realizado la ordenación de todos los elementos anteriores,por lo que el último elemento se encuentra ya en la posición en la que debería estar para que la tabla esté ordenada.

6.4 ¿Cuál es la operación básica de SelectSort?

Sabemos que la operación básica de un algoritmo es aquella operación que se repite más veces, lo que nos permite dar una aproximación de las iteraciones que realiza el algoritmo correspondiente. En este caso, la operación básica es la comparación de elementos, pues se realiza entre todos los elementos presentes entre los índices pertinentes al paso en que nos hallemos.

6.5 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS (n) y el caso mejor BBS (n) de SelectSort. Utilizad la notación asintótica (O, Θ, o, Ω,etc) siempre que se pueda.

Los tiempos de ejecución para un tamaño de entrada n son fijos para el algoritmo SelectSort. El algoritmo no es capaz de distinguir los casos en los que la tabla ya está ordenada, por lo que realiza siempre todas las iteraciones. De esta manera, los tiempos de ejecución del caso peor WBS (n) y el caso mejor BBS (n) se pueden dar como f(n)=n^2+O(n).

6.5 Compara los tiempos obtenidos para SelectSort y SelectSortInv, justifica las similitudes o diferencias entre ambos (es decir, indicar si las gráficas son iguales o distintas y por qué).

Los tiempos para SelectSort y SelectSortInv son prácticamente idénticos tal y como se ve en la sección de Resultados. Esto se debe a que SelectSortInv realiza exactamente la misma operación que SelectSort pero a la inversa. Es incapaz, de igual manera, de detectar si la tabla está ya ordenada, por lo que realiza siempre la cantidad máxima de operaciones básicas.

7. Conclusiones finales.

En conclusión, durante esta práctica hemos podido observar de manera práctica que el algoritmo de ordenación SelectSort no distingue entre casos mejor y peor según el tamaño de entrada, ya que no es capaz de identificar cuanda una tabla está ya ordenada. De esta manera, las operaciones básicas realizadas son siempre las mismas para un mismo tamaño de tabla, y en consecuencia, los tiempos de ejecución son también muy similares. También podemos observar que el SelectSortInv actúa de la misma manera que el Select Sort, modificando únicamente el orden de la tabla, pues aparece en orden descendente.

En adición, hemos aprendido a emplear herramientas nuevas como el GnuPlot para poder realizar comparaciones de manera visual, junto a comandos como sort y uniq para poder agrupar elementos en un archivo de texto. Por último, hemos descubierto la función clock() de c, pudiendo medir tiempos dentro de un programa, funcionalidad desconocida para nosotros anteriormente.

Bibliografía:

- Iserles, A. (1989). Numerical recipes in C—the art of scientific computing, by WH Press, BP Flannery, SA Teukolsky and WT Vetterling. Pp 735.£ 27· 50. 1988. ISBN 0-521-35465-X (Cambridge University Press). The Mathematical Gazette, 73(464), 167-170.
- Mezclar una matriz determinada con el algoritmo de reproducción aleatoria de Fisher-Yates–Acervo Lima. (s/f). Acervolima.com. Rec. 20 de octubre de 2022 *https://es.acervolima.com/mezclar-una-matriz-determinada-con-el-algoritmo-de-reproduccion-aleatoria-de-fisher-yates/*
- C library function - clock(). (s/f). Tutorialspoint.com. Rec. 20 de octubre de 2022. https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm