Análisis de Algoritmos 2022/2023

Práctica 1

Miguel Lozano Alonso y Eduardo Junoy Ortega, Grupo 1263.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta primera práctica se presenta el código que se ha programado para distintas funciones en C que generan número aleatorios, permutaciones aleatorias, ejecutan algoritmos de ordenación como SelectSort y miden los tiempos de ejecución de estas.

2. Objetivos

2.1 Apartado 1

Obtener los conocimientos necesarios para programar una función que genere números aleatorios equiprobables y aprender a realizar consultas en documentos de programación.

2.2 Apartado 2

Conseguir extraer la información del pseudocódigo, desarrollar los conocimientos sobre la función que genera números aleatorios obtenidos en el apartado 1 y programar una función que genera permutaciones.

2.3 Apartado 3

Continuar trabajando la generación de permutaciones para conseguir programar una función que genere un número determinado de permutaciones de la función programada anteriormente.

2.4 Apartado 4

Programar una función SelectSort que ordene los elementos de un array probarla con un fichero dado.

2.5 Apartado 5

Calcular y guardar en un fichero los tiempo de reloj y ejecución de OBs de un algoritmo de ordenación.

2.6 Apartado 6

Invertir la rutina SelectSort con éxito y comparar SelectSort y SelectSortInv en el caso medio el número de ejecuciones de OBs y el tiempo medio de reloj.

3. Herramientas y metodología

Hemos trabajado en Manjaro (Linux) y Windows, utilizando como entorno de desarrollo Visual Studio Code, como gestor de versiones Github y como medio de comunicación en el equipo Whatsapp.Los comandos que hemos utilizado son gcc, gnuplot, sort y uniq.

3.1 Apartado 1

Como la función rand() devuelve un número aleatorio entre 0 y MAX_RAND, pensamos en que si creamos el número aleatorio cogiendo el resto entre rand() y el límite superior. Siempre obtendremos un número menor al límite superior. A esto le sumamos 1 para incluir también al límite superior y le restamos el límite inferior que más tarde sumaremos para no obtener un número menor al límite inferior. Y obtuvimos este código:

$$aleatorio = rand() \% (sup - inf + 1) + inf$$

3.2 Apartado 2

Creamos una permutación ordenada de N elementos y mezaclamos los elementos aleatoriamente utilizando la función del apartado anterior.

3.3 Apartado 3

Llamamos n_perms veces a la función del apartado anterior pasandole el argumento N. Así creamos n_perms permutaciones de N elementos.

3.4 Apartado 4

Creamos la función min que devuelve el menor elemento de un array. Después creamos la función SelectSort que recorre el array llamando a la funcion min en cada iteración y pasándole como parámetro el subarray desde i hasta N-1.

3.5 Apartado 5

Creamos la función average_sorting_times que calcula el tiempo de ejecución en segundos y las OBs ejecutadas en el caso mejor, peor y medio. También creamos la función save_time_table que guarda los parametros anteriores en un fichero. Por último, creamos generate_shorting_times que llama a las dos funciones anteriores para guardar en un fichero el rendimiento de un algoritmo de ordenación según el tamaño de la entrada.

3.6 Apartado 6

Tomamos el mismo código que SelectSort, pero haciendo que el elemento menor se intercambie por el último elemento.

4. Código fuente

4.1 Apartado 1

```
/**
* @brief genera un número aleatorio entre inf y sup
* @param inf el límite inferior
* @param sup el límite superior
* @return el número aleatorio generado
*/
int random_num(int inf, int sup)
{
 int aleatorio;

 aleatorio = rand() % (sup-inf+1) + inf;
  return aleatorio;
}
```

4.2 Apartado 2

```c
/**
* @brief Intercambia dos elementos
* @param a puntero al primer elemento
* @param b puntero al segundo elemento
*/
void swap(int *a, int *b) {
 int aux = *a;
 *a = *b;
 *b = aux;
}


/**
* @brief Genera una permutación
* @param N tamaño de la permutación
* @returns puntero al array que contiene la permutación o NULL en caso de error
*/
int* generate_perm(int N)
{
 int i;
 int *perm;

 if(N <= 0) return NULL;
```

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

 if(perm == NULL) return NULL;



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

   perm[i] = (i+1);

 }

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

   swap(&perm[i], &perm[random_num(i, N-1)]);

 }

 return perm;

}
```

4.3 Apartado 3

```
/**
* @brief Genera n_perms permutaciones aleatorias de N elementos
* @param n_perms número de permutaciones
* @param N tamaño de las permutaciones
* @returns array de punteros a las permutaciones o NULL en caso de error
*/
int** generate_permutations(int n_perms, int N)
{
 int i;
 int **permutations;
 permutations = (int**)malloc(sizeof(int*) * n_perms);
 for(i=0; i<n_perms; i++) {
   permutations[i] = generate_perm(N);
 }
 return permutations;
}
```

4.4 Apartado 4

```c
/**
* @brief Ordena de menor a mayor un array
* @param array el array a ordenar
* @param ip primer índice del array
* @param iu último índice del array
* @return el número de operaciones básicas (comparaciones de clave) que ha realizado el algortimo
*/
int SelectSort(int* array, int ip, int iu)
{
 int i, count, minimum;
 if(array == NULL || ip > iu) return ERR;

 count = 0;
 i = ip;
 while(i < iu) {
   minimum = min(array, i, iu);
   count += iu-i;
   swap(&array[i], &array[minimum]);
   i++;
 }
 return count;
}
```

```
/**
* @brief Localiza el menor elemento de un array
* @param array el array en el que busca
* @param ip primer índice del array
* @param iu último índice del array
* @return el índice del menor elemento del array
*/
int min(int* array, int ip, int iu)
{
 int i;
 int min;
  min = ip;
 for(i = ip+1; i<=iu; i++) {
   if(array[i] < array[min]) {
     min = i;
   }
 }

 return min;
}
```

4.5 Apartado 5

```c
/**
* @brief Calcula los tiempo de ejecución de un algoritmo de ordenación
* @param metodo la función de ordenación
* @param n_perms el número de permutaciones
* @param N el tamaño de las permutaciones
* @param ptime un puntero a la estructura time_aa
* @return ERR en caso de error y OK en caso contrario
*/
short average_sorting_time(pfunc_sort metodo,
                           int n_perms,
                           int N,
                           PTIME_AA ptime)
{
 int **permutaciones;
 int ini, fin, i, ob;
 ptime->N = N;
 ptime->n_elems = n_perms;
 ptime->min_ob = ptime->max_ob = ptime->average_ob = 0;

 permutaciones = generate_permutations(n_perms, N);
 if(permutaciones == NULL){
   return ERR;
```

```
}


ini = clock();
for (i = 0; i < n_perms; i++)
{
  ob = metodo(permutaciones[i], 0, N-1);
  if(ob == ERR) {
    for(i=0; i<n_perms; i++) {
      free(permutaciones[i]);
    }
    free(permutaciones);
    return ERR;
  }


  ptime->average_ob += ob;


  if(ptime->min_ob > ob || ptime->min_ob == 0){
    ptime->min_ob = ob;
  }
  else if(ptime->max_ob < ob || ptime->max_ob == 0){
    ptime->max_ob = ob;
  }
}
```

```
 fin = clock();



 ptime->time = (double)(fin-ini)/CLOCKS_PER_SEC/n_perms;

 ptime->average_ob /= n_perms;



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

 {

   free(permutaciones[i]);

 }

 free(permutaciones);

 return OK;

}



/**

* @brief utiliza average_sorting_time y save_time_table para calcular los tiempo de ejecución de un algoritmo y guardarlos en un archivo

* @param method un algoritmo de ordenación

* @param file el nombre del archivo donde se guardarán los tiempos de ejecución

* @param num_min el ínfimo del conjunto de tamaños que tomarán las permutaciones

* @param num_max el supremo del conjunto de tamaños que tomarán las permutaciones

*  @param incr incremento del tamaño de la permutación en cada iteración
```

```
* @param n_perms número de permutaciones a ordenar en cada iteración

* @return ERR en caso de error y OK en caso contrario

*/

short generate_sorting_times(pfunc_sort method, char* file,

                             int num_min, int num_max,

                             int incr, int n_perms)

{

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

 int tamanyo, i;

 PTIME_AA p_time, times;



 p_time = (PTIME_AA)malloc(sizeof(p_time[0]));

 if(p_time == NULL)

 {

   return ERR;

 }

 times = (PTIME_AA)malloc(sizeof(times[0]) * n_times);

 if(times == NULL)

 {

   free(p_time);

   return ERR;

 }
```

```
 for (i=0, tamanyo = num_min; tamanyo <= num_max; tamanyo += incr,
i++)

 {

  if(average_sorting_time(method, n_perms, tamanyo, p_time) == ERR)

   {

     free(p_time);

     free(times);

     return ERR;

   }

   times[i].N = p_time->N;

   times[i].n_elems = p_time->n_elems;

   times[i].time = p_time->time;

   times[i].average_ob = p_time->average_ob;

   times[i].max_ob = p_time->max_ob;

   times[i].min_ob = p_time->min_ob;

 }

 if(save_time_table(file, times, n_times) == ERR) {

   free(p_time);

   free(times);

   return ERR;

 }



 free(p_time);
```

```c
 free(times);

 return OK;

}



/**

* @brief Escribe los tiempos de ejecución y otros parámetros del rendimiento del algoritmo en un archivo

* @param file el nombre del archivo

* @param ptime puntero a la información del rendimiento del algoritmo

* @param n_times número de tiempos a los que apunta ptime

* @return ERR en caso de error y OK en caso contrario

*/

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

{

 int i;



 FILE *pf = fopen(file, "w");

 if(pf == NULL) return ERR;



 for(i=0; n_times > 0; i++, n_times--)

 {

  fprintf(pf,"%d  %.10f  %.2f  %d  %d\n", ptime[i].N, ptime[i].time, ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob);

 }
```

```
  fclose(pf);

 return OK;

}
```

4.6 Apartado 6

```
/**
* @brief Ordena de mayor a menor un array
* @param array el array a ordenar
* @param ip primer índice del array
* @param iu último índice del array
* @return el número de operaciones básicas (comparaciones de clave) que ha realizado el algortimo
*/
int SelectSortInv(int* array, int ip, int iu)
{
 int i, count, minimum;
 if(array == NULL || ip > iu) return ERR;
 for(i=0; i<iu; i++);

 count = 0;
 while(i >= 0) {
   minimum = min(array, ip, i);
   count += i-ip;
   swap(&array[i], &array[minimum]);
   i--;
 }
 return count;
}
```

5. Resultados, Gráficas

En este punto se presentan los resultados y gráficas obtenidas en cada apartado.

5.1 Apartado 1

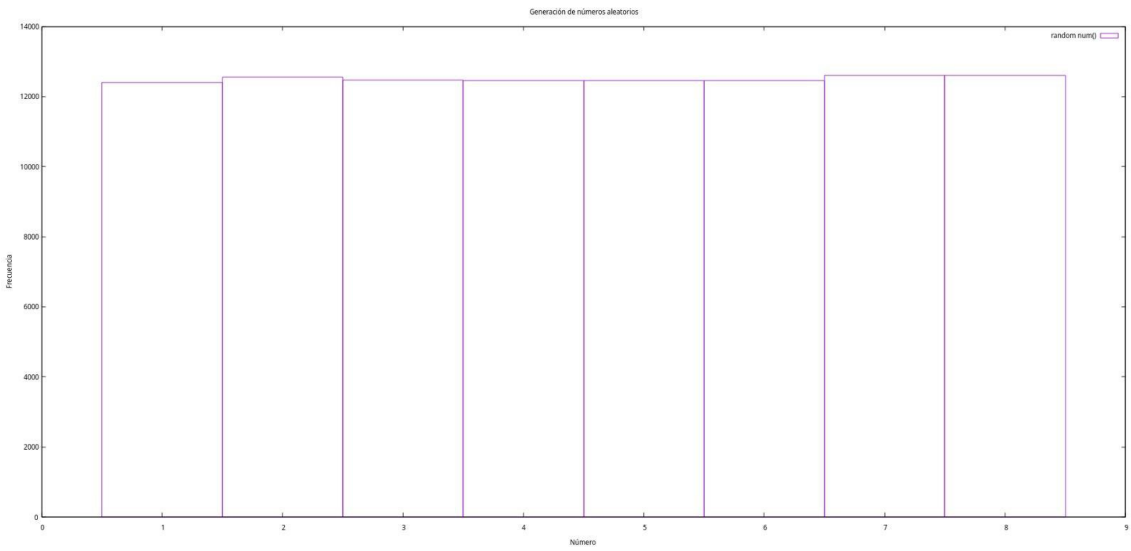
Aplicamos el comando de ejecución:

./exercise1 -limInf 1 -limSup 8 -numN 100000 | sort | uniq -c

Nos da el siguiente resultado:

```
12515 1
12365 2
12654 3
12755 4
12387 5
12543 6
12308 7
12473 8
```

Y la siguiente gráfica:



En este diagrama de barras se observa que la función de creación de númerosaleatorios es equiprobable.

5.2 Apartado 2

Aplicamos el siguiente comando de ejecución:

```
./exercise2 -size 3 -numP 100000 | sort | uniq -c
```

Y nos da el siguiente resultado:

```
16593 1 2 3

16727 1 3 2

16772 2 1 3

16685 2 3 1

16661 3 1 2

16562 3 2 1
```

5.3 Apartado 3

Aplicamos el siguiente comando de ejecución:

```
./exercise3 -size 3 -numP 100000 | sort | uniq -c
```

Y nos da el siguiente resultado:

```
16586 1 2 3

17088 1 3 2

16548 2 1 3

16654 2 3 1

16459 3 1 2

16665 3 2 1
```

5.4 Apartado 4

Aplicamos el siguiente comando de ejecución:

./exercise4 -size 10

Y nos da el siguiente resultado:

1 2 3 4 5 6 7 8 9 10

5.5 Apartado 5

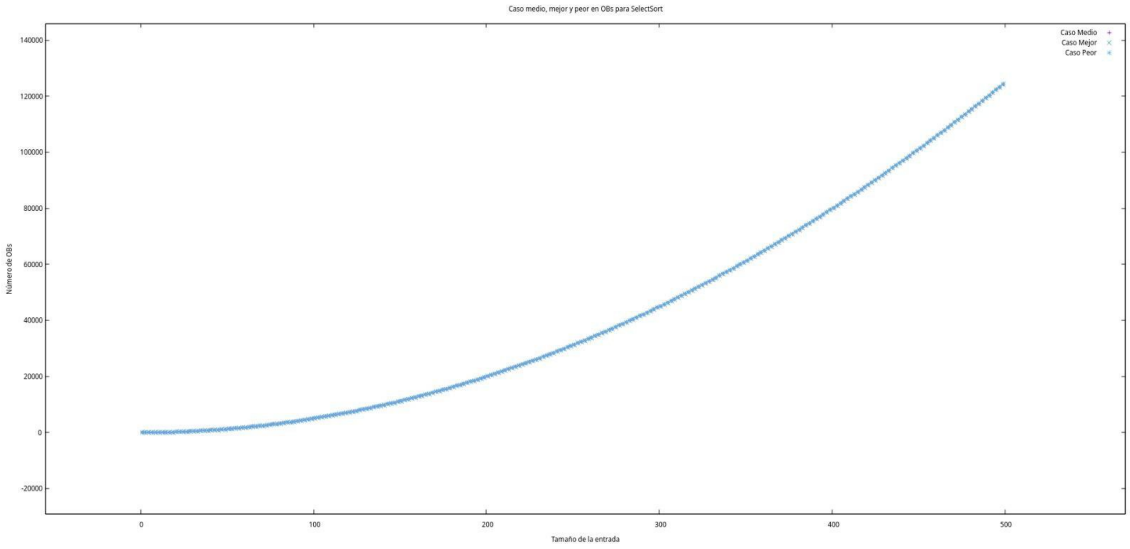
Aplicamos el siguiente comando de ejecución:

./exercise5 -num_min 1 -num_max 500 -incr 2 -numP 100 -outputFile exercise5.log

Y nos da el siguiente resultado:

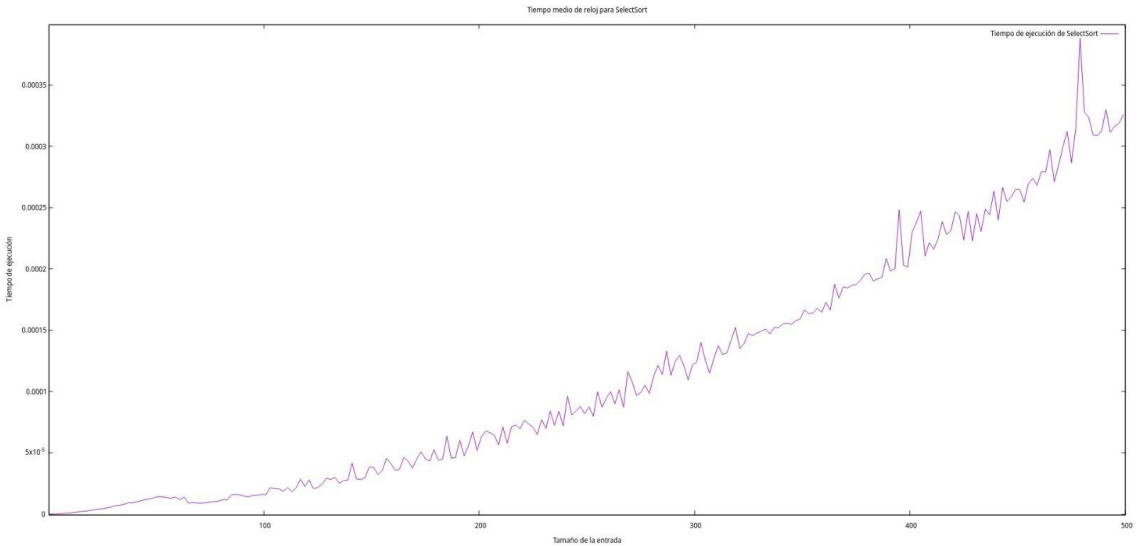
exercise5.log

Gráfica comparando los tiempos medio, mejor y peor en OBs para SelectSort:



Como se puede observar en la gráfica, de los tiempos mejor, peor y medio en OBs, la función que siguen es exactamente la misma para los tres casos: $\frac{N(N-1)}{2}$

Gráfica con el tiempo medio de reloj para SelectSort:



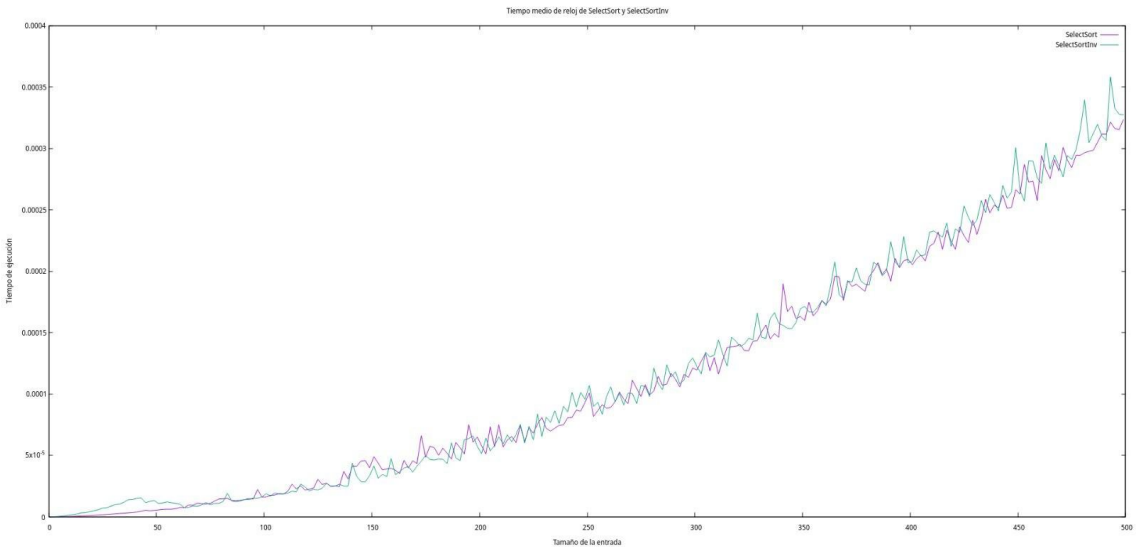
En esta ocasión, la gráfica se aproxima menos a la función $\frac{N(N-1)}{2}$, ya que hay casos en los que se ejecuta más línea de código que en otros, no se ejecuta simplemente la OB como hacíamos en el apartado anterior.

5.6 Apartado 6

Resultados del apartado 6:

SelectSortInv.log

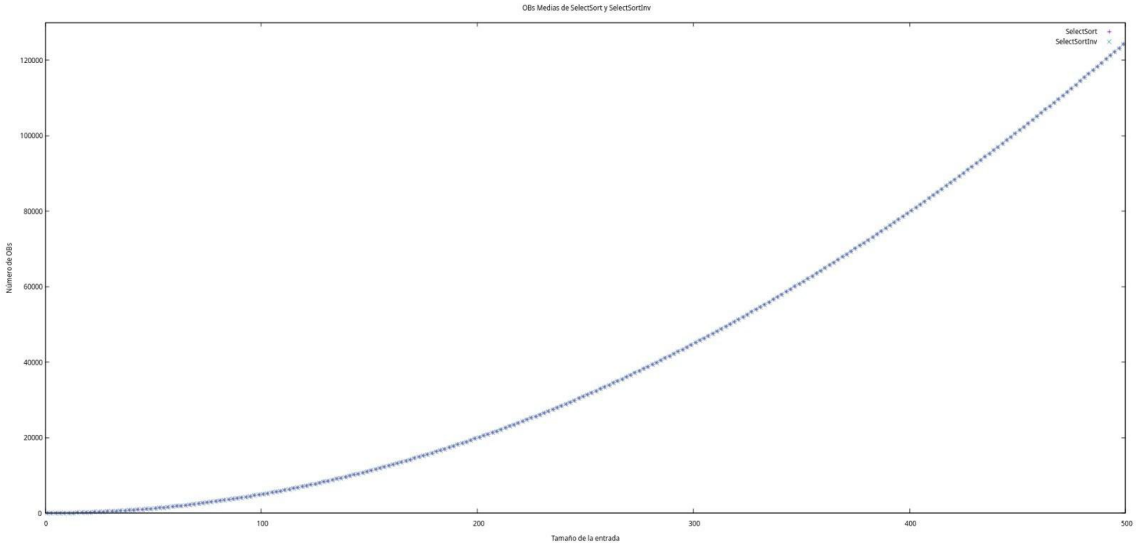
Gráfica comparando el tiempo medio de OBs para SelectSort y SelectSortInv:



Se puede observar que el tiempo de ejecución de las dos funciones sigue la misma función que ya comentábamos antes: $\frac{N(N-1)}{2}$

Únicamente se distinguen pequeñas variaciones que tienen relación con la cantidad de condiciones que se utilizan para cada ejecución.

Gráfica comparando el tiempo medio de reloj para SelectSort y SelectSortInv:



Por último, en este resultado gráfico se observa que el tiempo de ejecución medio para SelectSort y SelectSortInv es el mismo, ya que la OB es la misma en ambos y no está sujeta a posibles cambios de longitud de código. La función correspondiente a las funciones es la misma que comentábamos antes: $\frac{N(N-1)}{2}$

6. Respuesta a las preguntas teóricas.

En el presente punto se resuelven las preguntas teóricas planteadas.

6.1 Pregunta 1

Al principio probamos el algoritmo:`aleatorio = inf+(int)(sup*rand()/(RAND_MAX+1.0));` extraído de Numerical recipes in C: the art of scientific computing que nos trajo problemas en la aplicación. Para posteriormente utilizar este: `aleatorio = rand() % (sup-inf+1) + inf;`, que se basa en obtener el módulo de un número aleatorio generado por la función rand() con la diferencia del número superior y el inferior, para después sumarle el número inferior.

6.2 Pregunta 2

SelectSort recorre el array en busca del menor elemento, una vez ha recorrido el array, coloca el menor elemento al principio del array. Repite este proceso ordenando en cada iteración el menor elemento de la subtabla que todavía no ha ordenado.

6.3 Pregunta 3

El bucle exterior de SelectSort no actúa sobre el último elemento porque ese elemento ya está ordenado. Si de un array con k elementos ordenas k-1 elementos, el elemento k está también ordenado.

6.4 Pregunta 4

La operación básica de SelectSort es la comparación de claves (CDCS), que está contenida en la función `int min(int* array, int ip, int iu);`.

6.5 Pregunta 5

Como en SelectSort hay que recorrer un array de tamaño N un total de N veces esté ordenado o desordenado el array. Entonces, el caso mejor y peor es el mismo:

$$W_{SS}(N) = B_{SS}(N) = \Theta(N^2)$$

6.5 Pregunta 6

Las gráficas de SelectSort y SelectSortInv son iguales porque el array se recorre de la misma manera, la única diferencia es que en SelectSort el menor elemento se intercambia con el elemento del principio y en SelectSortInv el menor elemento se intercambia con el elemento del final.

7. Conclusiones finales.

La práctica nos ha ayudado a aplicar los conocimientos que tenemos sobre permutaciones, a conocer nuevos comandos como uniq y sort. A aprender sobre el funcionamiento interno de los algoritmos de ordenación. Y también a entender el rendimiento de estos algoritmos y poder compararlos