Análisis de Algoritmos 2022/2023

Práctica 1

Trabajo realizado por: Marcos Alonso Pardo y Jorge Paniagua Moreno

Código	Gráficas	Memoria	Total

1. Introducción.

En esta primera práctica hemos ido escalando poco a poco, empezando por generar números aleatorios hasta producir distintas permutaciones. Luego hemos hecho el SelectSort, y hemos terminado utilizando todas estas cosas para crear unas funciones que analizan el tiempo y las operaciones básicas que hacía este algoritmo. A lo largo de esta memoria se expondrán los objetivos, la metodología, el código, los resultados, y las respuestas a las preguntas teóricas.

2. Objetivos

2.1 Apartado 1

El objetivo del apartado 1 es crear una función que genere números aleatorios en un rango dado utilizando la función rand(). Hay que conseguir que sea lo más aleatoria posible.

2.2 Apartado 2

El objetivo del apartado 2 es crear una función que cree una tabla ordenada con los valores de 0 a N, y que posteriormente haga N “swaps”, es decir, intercambiar dos posiciones del array de forma aleatoria para así generar una permutación.

2.3 Apartado 3

El objetivo del apartado 3 es crear una función que usando la del apartado 2, guarde en un array bidimensional ‘n_perms’ permutaciones.

2.4 Apartado 4

El objetivo del apartado 4 es programar el algoritmo local de ordenación SelectSort, al que le pasas una tabla y el índice inicial y final, y lo devuelve ordenado. Hay que devolver el número de OB hechas por el algoritmo. Una función auxiliar llamada min es usada para calcular el número más pequeño en la subtabla correspondiente a cada iteración.

2.5 Apartado 5

El objetivo del apartado 5 es analizar los tiempos de ejecución de SelectSort. Para este apartado creamos 3 funciones y hacemos uso de una estructura donde guardaremos distintos parámetros a medir. La primera función genera datos medios realizando pruebas con distintas permutaciones en un tamaño de tabla dado. La segunda función utiliza esta varias veces para probar con distintos tamaños de entrada, y usando la tercera se escriben en un fichero los resultados de las pruebas.

2.6 Apartado 6

El objetivo del apartado 6 es crear una función SelectSortInv que ordene los números de mayor a menor, al contrario que la función original.

3. Herramientas y metodología

Hemos desarrollado esta práctica en un entorno Linux, y utilizando VSCode, gcc, valgrind, y a veces gdb para debuggear. Hemos usado GNUplot para las gráficas.

3.1 Apartado 1

rand() devuelve un número aleatorio a partir de la semilla generada en srand() presente en todos los exerciseX.c. Como rand() devuelve un número entre 0 y RAND_MAX, le hacemos el módulo entre la cota superior menos la inferior mas uno, y luego le sumamos la cota inferior. De esta manera nos lo genera entre los números deseados.

3.2 Apartado 2

En este apartado, reservamos memoria para una tabla de tamaño N y con un bucle la llenamos de los números correspondientes a sus índices. Luego hacemos un segundo bucle, en el que recorriendo todos los índices les aplicamos un intercambio con otra posición aleatoria de la tabla, y devolvemos la tabla.

3.3 Apartado 3

Reservamos memoria para una tabla bidimensional, ya que cada elemento de esta tabla va a ser una tabla también. Hacemos un bucle en el que en cada posición de la tabla de tablas generamos una permutación de tamaño N, Esto lo hacemos n_perms veces. Devolvemos por último la tabla de tablas.

3.4 Apartado 4

Para implementar el SelectSort, tenemos un bucle que recorre desde el primer índice hasta el último indicado, y lo primero que hace en cada iteración es encontrar el elemento mas pequeño desde i hasta el final. Si este es distinto del elemento en el índice i, lo intercambia por el. De esta manera, en cada iteración va colocando el número más pequeño al principio, luego el siguiente, etc. Para encontrar el mínimo en cada iteración usamos la función min, que va comparando desde i por cada índice, y cuando encuentra uno menor se lo asigna a la variable minim, que es la que al final contendrá el menor y será devuelto.

3.5 Apartado 5

En la primera función, primero generamos n_perms permutaciones y las guardamos. Luego hacemos un bucle de 0 a n_perms en el que aplicamos SelectSort a cada permutación y guardamos datos. Para medir el tiempo, usamos clock justo antes y después. También guardamos el número de OB para hacer la media, y comprobamos a ver si encontramos algún mínimo o máximo de OB ya que también lo piden. Luego, liberamos todas las permutaciones, y asignamos en la estructura todos los valores medidos.

Posteriormente, en la segunda función, reservamos memoria para un array de estructuras, ya que vamos a realizar la primera función mas de una vez. Para ser

concretos, la haremos desde un numero min hasta un número max, incrementando el valor que sea pedido. Después de guardar todos los datos en la tabla de estructuras, llamamos a la tercera función, que va recorriendo la tabla y mediante fprintf escribe en un fichero indicado como parámetro al programa los resultados de las pruebas. Por último se libera la tabla de estructuras, y en la tercera función se cierra el fichero.

3.6 Apartado 6

Metodología y solución adoptada del apartado 6

Este SelectSortInv funciona de una forma muy parecida al SelectSort. Esta función se encarga de encontrar el menor elemento y en vez de colocarlo el primero, lo coloca en el último lugar utilizando la función min.

A la hora de comparar el tiempo medio el tiempo medio de reloj podemos observar que son semejantes y comparando el tiempo medio de Obs es exactamente el mismo.

4. Código fuente

4.1 Apartado 1

```
#include "permutations.h"

#include <stdio.h>
#include <stdlib.h>



int random_num(int inf, int sup)

{
int num;
if (inf >= sup)
return ERR;
num = (rand() % (sup - inf + 1)) + inf;
return num;
}
```

4.2 Apartado 2

```
#include "permutations.h"

#include <stdio.h>
#include <stdlib.h>
```

```
int* generate_perm(int N)

{
int i;
int aux;
int aux2;
int *perm = NULL;

if (N<=0)
return NULL;
i = 0;
perm = malloc(sizeof(int) * N);
if (!perm)
return NULL;
for (i = 0; i < N; i++)
perm[i] = i;
for (i = 0; i < N; i++)
{
aux2 = random_num(i, N) - 1;
aux = perm[i];
if (aux2 == -1)
aux2++;
perm[i] = perm[aux2];
perm[aux2] = aux;
}
return perm;
}
```

4.3 Apartado 3

```
#include "permutations.h"

#include <stdio.h>
#include <stdlib.h>


int** generate_permutations(int n_perms, int N)

{
int **perm;
int i;

if (n_perms<= 0 || N<= 0)
return NULL;
i = 0;
perm = malloc(sizeof(int*) * n_perms);
```

```
if (!perm)
return NULL;
for(i = 0; i < n_perms; i++)
perm[i] = generate_perm(N);
return (perm);
}
```

4.4 Apartado 4

```
#include "sorting.h"

#include <stdio.h>
#include <stdlib.h>


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

{
int i;
int minim;
int aux;
int count;
int N = iu-ip +1;

count = N*(N-1)/2;
if (!array || ip<0 || iu<0)
return ERR;
i = ip;
while (i < iu)
{
minim = min(array, i, iu);
if (minim != i)
{
aux = array[i];
array[i] = array[minim];
array[minim] = aux;
}
i++;
}
return count;
}

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

{
int j;
```

```
int minim;

if (!array || ip<0 || iu<0)
return ERR;
minim = ip;
for (j = ip + 1; j <= iu; j++)
{
if (array[j] < array[minim])
minim = j;
}
return (minim);
}
```

4.5 Apartado 5

```
#include "times.h"

#include "permutations.h"
#include "sorting.h"
#include <stdlib.h>


short average_sorting_time(pfunc_sort metodo,

int n_perms,
int N,
PTIME_AA ptime)
{
int **perms;
int i = 0;
clock_t start, end;
double media = 0;
int OBaux = 0;
double mediaOB = 0;
int OBlow = 0;
int OBhigh = 0;

if (!metodo || !ptime || N<=0 || n_perms<=0)
return ERR;
perms = generate_permutations(n_perms, N);
if (!perms)
return ERR;
while (i < n_perms)
{
start = clock();
OBaux = metodo(perms[i], 0, N - 1);
```

```
end = clock();
if (OBlow < OBaux || OBlow == 0)
OBlow = OBaux;
if (OBhigh < OBaux || OBhigh == 0)
OBhigh = OBaux;
media += (double)(end - start);
mediaOB += OBaux;
i++;
}
for(i=0;i<n_perms;i++)
free(perms[i]);
free(perms);
media /= (double)n_perms;
mediaOB /= n_perms;
ptime->time = media;
ptime->N = N;
ptime->n_elems = n_perms;
ptime->average_ob = mediaOB;
ptime->min_ob = OBlow;
ptime->max_ob = OBhigh;
return OK;
}

short generate_sorting_times(pfunc_sort method, char* file,

int num_min, int num_max,
int incr, int n_perms)
{
int i = num_min;
int j = 0;
PTIME_AA ptime;
if (!method || !file || num_min<=0 || num_max< num_min || incr <= 0 ||
n_perms<=0)
return ERR;
ptime = malloc(sizeof(TIME_AA) * ((num_max - num_min)/incr + 1));
if (!ptime)
return ERR;
j = 0;
while(j < (num_max - num_min)/incr + 1)
{
if (average_sorting_time(method, n_perms, i, &ptime[j]) == ERR)
return ERR;
i+=incr;
j++;
}
if (save_time_table(file, ptime, j - 1) == ERR)
return ERR;
free(ptime);
return OK;
```

```
}

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

{
FILE *fd;
int i = 0;

if (!file || !ptime || n_times<=0)
return ERR;
fd = fopen(file, "w");
if (fd<=0)
return ERR;
while (i <= n_times)
{
fprintf(fd, "| size: %d | average time: %f | average OB: %f | max OB: %d | min
OB: %d |\n", ptime[i].N, ptime[i].time, ptime[i].average_ob, ptime[i].max_ob,
ptime[i].min_ob);
i++;
}
fclose(fd);
return OK;
}
```

4.6 Apartado 6

```
#include "sorting.h"

#include <stdio.h>
#include <stdlib.h>


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

{
int i;
int minim;
int aux;
int count;
int N = iu-ip + 1;

i = iu;
count = N*(N-1)/2;
while (i > ip)
{
minim = min(array, ip, i);
```

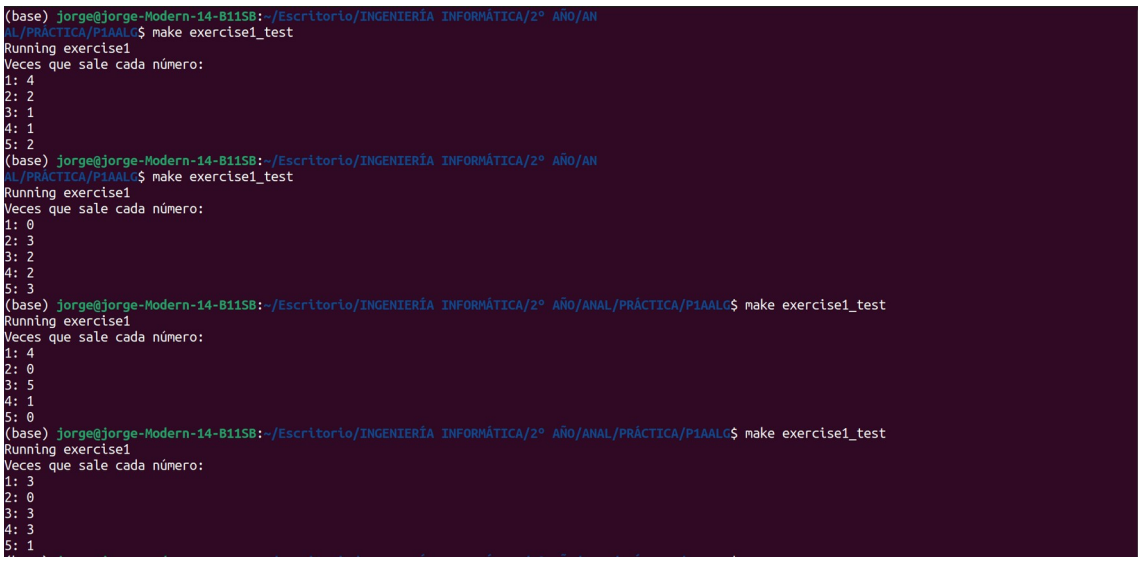
```
if (minim != i)
{
aux = array[i];
array[i] = array[minim];
array[minim] = aux;
}
i++;
}
return count;
}
```

5. Resultados, Gráficas

Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

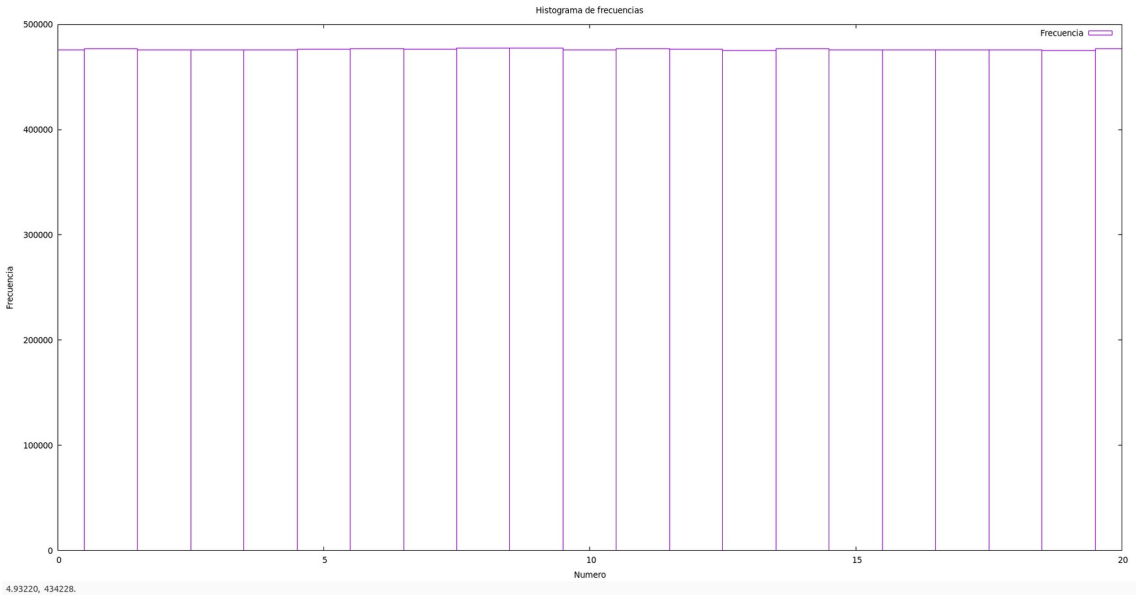
5.1 Apartado 1

Resultados del apartado 1.



PARÁMETROS: -limInf 1 -limSup 5 -numN 10

Gráfica del histograma de números aleatorios, comentarios a la gráfica



La aleatoriedad de nuestro algoritmo es correcta. Los parámetros usados para sacar la gráfica han sido: -limInf 0 -limSup 20 -numN 10000000

5.2 Apartado 2

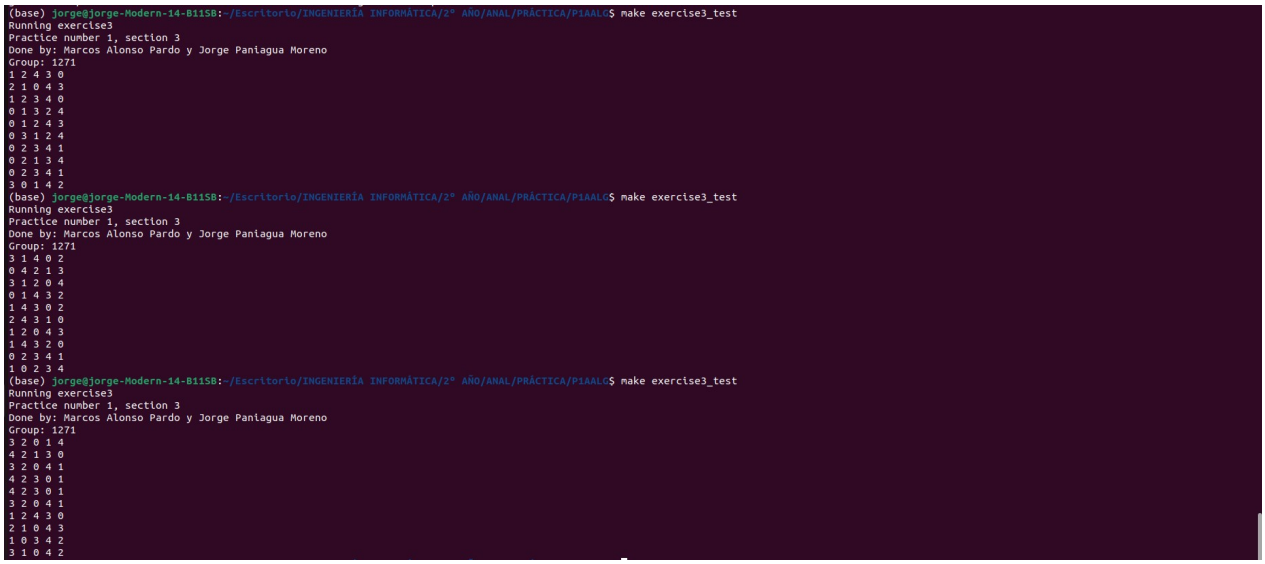
Resultados del apartado 2.

```
(base) jorge@jorge-Modern-14-B11SB:~/Escritorio/INGENIERÍA INFORMÁTICA/2º AÑO/ANAL/PRÁCTICA/P1AALG$ make exercise2_test
Running exercise2
Practice number 1, section 2
Done by: Marcos Alonso Pardo y Jorge Paniagua Moreno
Group: 1271
3 0 1 4 2
4 1 0 3 2
0 2 3 4 1
1 4 2 3 0
0 4 2 1 3
2 4 1 3 0
2 0 4 1 3
3 4 0 1 2
2 1 3 4 0
2 0 3 1 4
(base) jorge@jorge-Modern-14-B11SB:~/Escritorio/INGENIERÍA INFORMÁTICA/2º AÑO/ANAL/PRÁCTICA/P1AALG$ make exercise2_test
Running exercise2
Practice number 1, section 2
Done by: Marcos Alonso Pardo y Jorge Paniagua Moreno
Group: 1271
3 2 0 1 4
2 0 1 3 4
1 3 0 4 2
2 0 1 4 3
1 0 3 2 4
0 3 1 4 2
1 2 3 4 0
1 3 4 2 0
1 2 3 0 4
1 3 4 2 0
(base) jorge@jorge-Modern-14-B11SB:~/Escritorio/INGENIERÍA INFORMÁTICA/2º AÑO/ANAL/PRÁCTICA/P1AALG$ make exercise2_test
Running exercise2
Practice number 1, section 2
Done by: Marcos Alonso Pardo y Jorge Paniagua Moreno
Group: 1271
3 1 2 4 0
4 3 1 0 2
1 0 3 4 2
1 3 0 4 2
3 1 0 4 2
4 3 2 0 1
1 2 3 0 4
1 3 4 0 2
1 3 0 2 4
3 4 0 1 2
```

PARÁMETROS: -size 5 -numP 10

5.3 Apartado 3

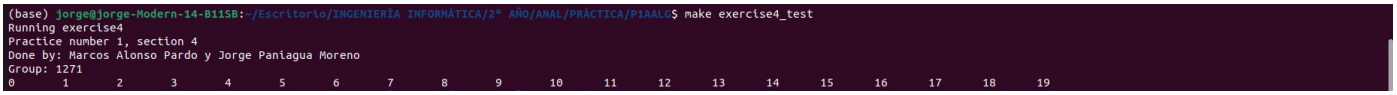
Resultados del apartado 3.



PARÁMETROS: -size 5 -numP 10

5.4 Apartado 4

Resultados del apartado 4.



PARÁMETROS: -size 20

5.5 Apartado 5

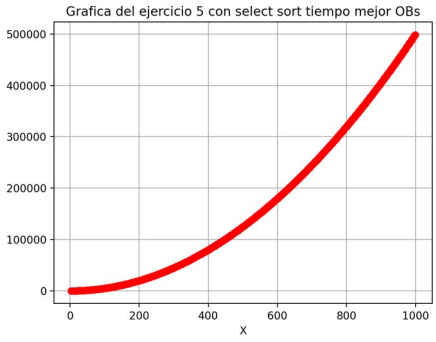
Resultados del apartado 5.

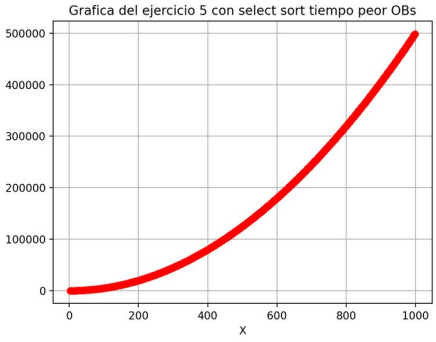
```
| size: 1 | average time: 0.300000 | average OB: 0.000000 | max OB: 0 | min OB: 0 |
| size: 3 | average time: 0.700000 | average OB: 3.000000 | max OB: 3 | min OB: 3 |
| size: 5 | average time: 0.700000 | average OB: 10.000000 | max OB: 10 | min OB: 10 |
| size: 7 | average time: 1.000000 | average OB: 21.000000 | max OB: 21 | min OB: 21 |
| size: 9 | average time: 1.500000 | average OB: 36.000000 | max OB: 36 | min OB: 36 |
| size: 11 | average time: 1.600000 | average OB: 55.000000 | max OB: 55 | min OB: 55 |
| size: 13 | average time: 2.000000 | average OB: 78.000000 | max OB: 78 | min OB: 78 |
| size: 15 | average time: 2.600000 | average OB: 105.000000 | max OB: 105 | min OB: 105 |
| size: 17 | average time: 2.900000 | average OB: 136.000000 | max OB: 136 | min OB: 136 |
| size: 19 | average time: 3.300000 | average OB: 171.000000 | max OB: 171 | min OB: 171 |
| size: 21 | average time: 3.800000 | average OB: 210.000000 | max OB: 210 | min OB: 210 |
| size: 23 | average time: 4.300000 | average OB: 253.000000 | max OB: 253 | min OB: 253 |
| size: 25 | average time: 4.700000 | average OB: 300.000000 | max OB: 300 | min OB: 300 |
| size: 27 | average time: 5.300000 | average OB: 351.000000 | max OB: 351 | min OB: 351 |
| size: 29 | average time: 6.000000 | average OB: 406.000000 | max OB: 406 | min OB: 406 |
| size: 31 | average time: 6.400000 | average OB: 465.000000 | max OB: 465 | min OB: 465 |
| size: 33 | average time: 7.500000 | average OB: 528.000000 | max OB: 528 | min OB: 528 |
| size: 35 | average time: 9.800000 | average OB: 595.000000 | max OB: 595 | min OB: 595 |
| size: 37 | average time: 8.200000 | average OB: 666.000000 | max OB: 666 | min OB: 666 |
| size: 39 | average time: 8.700000 | average OB: 741.000000 | max OB: 741 | min OB: 741 |
| size: 41 | average time: 9.700000 | average OB: 820.000000 | max OB: 820 | min OB: 820 |
| size: 43 | average time: 10.100000 | average OB: 903.000000 | max OB: 903 | min OB: 903 |
| size: 45 | average time: 10.500000 | average OB: 990.000000 | max OB: 990 | min OB: 990 |
| size: 47 | average time: 11.700000 | average OB: 1081.000000 | max OB: 1081 | min OB: 1081 |
| size: 49 | average time: 12.300000 | average OB: 1176.000000 | max OB: 1176 | min OB: 1176 |
| size: 51 | average time: 13.400000 | average OB: 1275.000000 | max OB: 1275 | min OB: 1275 |
| size: 53 | average time: 13.500000 | average OB: 1378.000000 | max OB: 1378 | min OB: 1378 |
| size: 55 | average time: 14.200000 | average OB: 1485.000000 | max OB: 1485 | min OB: 1485 |
| size: 57 | average time: 14.800000 | average OB: 1596.000000 | max OB: 1596 | min OB: 1596 |
| size: 59 | average time: 15.600000 | average OB: 1711.000000 | max OB: 1711 | min OB: 1711 |
| size: 61 | average time: 16.700000 | average OB: 1830.000000 | max OB: 1830 | min OB: 1830 |
| size: 63 | average time: 18.000000 | average OB: 1953.000000 | max OB: 1953 | min OB: 1953 |
| size: 65 | average time: 18.400000 | average OB: 2080.000000 | max OB: 2080 | min OB: 2080 |
| size: 67 | average time: 20.200000 | average OB: 2211.000000 | max OB: 2211 | min OB: 2211 |
| size: 69 | average time: 21.100000 | average OB: 2346.000000 | max OB: 2346 | min OB: 2346 |
| size: 71 | average time: 22.200000 | average OB: 2485.000000 | max OB: 2485 | min OB: 2485 |
| size: 73 | average time: 23.000000 | average OB: 2628.000000 | max OB: 2628 | min OB: 2628 |
| size: 75 | average time: 23.900000 | average OB: 2775.000000 | max OB: 2775 | min OB: 2775 |
| size: 77 | average time: 25.800000 | average OB: 2926.000000 | max OB: 2926 | min OB: 2926 |
| size: 79 | average time: 25.700000 | average OB: 3081.000000 | max OB: 3081 | min OB: 3081 |
| size: 81 | average time: 8.400000 | average OB: 3240.000000 | max OB: 3240 | min OB: 3240 |
| size: 83 | average time: 8.600000 | average OB: 3403.000000 | max OB: 3403 | min OB: 3403 |
| size: 85 | average time: 9.100000 | average OB: 3570.000000 | max OB: 3570 | min OB: 3570 |
| size: 87 | average time: 9.300000 | average OB: 3741.000000 | max OB: 3741 | min OB: 3741 |
| size: 89 | average time: 9.700000 | average OB: 3916.000000 | max OB: 3916 | min OB: 3916 |
| size: 91 | average time: 10.100000 | average OB: 4095.000000 | max OB: 4095 | min OB: 4095 |
| size: 93 | average time: 10.500000 | average OB: 4278.000000 | max OB: 4278 | min OB: 4278 |
| size: 95 | average time: 10.700000 | average OB: 4465.000000 | max OB: 4465 | min OB: 4465 |
| size: 97 | average time: 11.200000 | average OB: 4656.000000 | max OB: 4656 | min OB: 4656 |
| size: 99 | average time: 11.700000 | average OB: 4851.000000 | max OB: 4851 | min OB: 4851 |
```

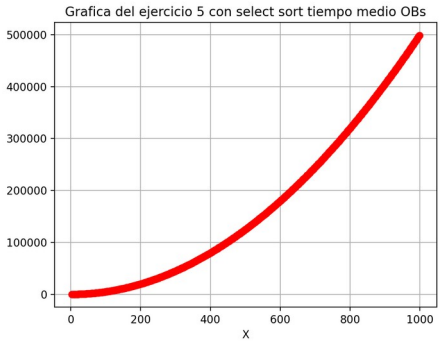
PARÁMETROS: -num_min 1 -num_max 100 -incr 2 -numP 10 -outputFile exercise5.log

Gráfica comparando los tiempos mejor, peor y medio en OBs para SelectSort, comentarios a la gráfica.

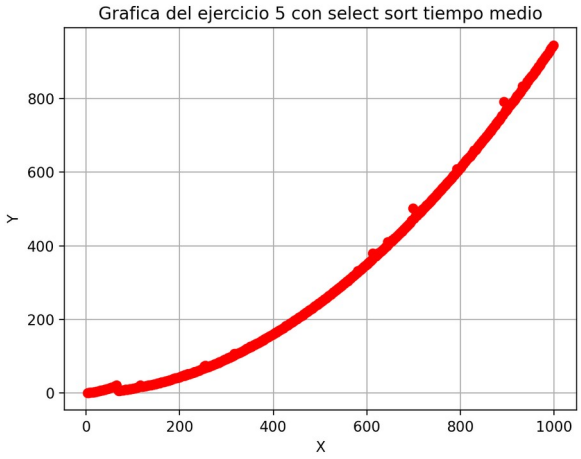
Todas siguen la gráfica x^2 porque en el SelectSort el número de Obs son el mismo.







Gráfica con el tiempo medio de reloj para SelectSort, comentarios a la gráfica.



Esta gráfica tiene que seguir y sigue a la gráfica x^2.

5.6 Apartado 6

Resultados del apartado 6.

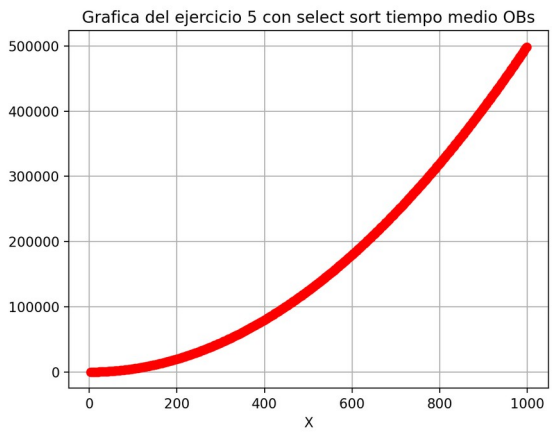
```
 86  int main()
 87  {
 88    int *tabla;
 89
 90    tabla = malloc(sizeof(int) * 5);
 91    tabla[0] = 5;
 92    tabla[1] = 3;
 93    tabla[2] = 4;
 94    tabla[3] = 1;
 95    tabla[4] = 2;
 96    SelectSortInv(tabla, 0, 4);
 97    printf("%d\n", tabla[0]);
 98    printf("%d\n", tabla[1]);
 99    printf("%d\n", tabla[2]);
100    printf("%d\n", tabla[3]);
101    printf("%d\n", tabla[4]);
102  }
103
```

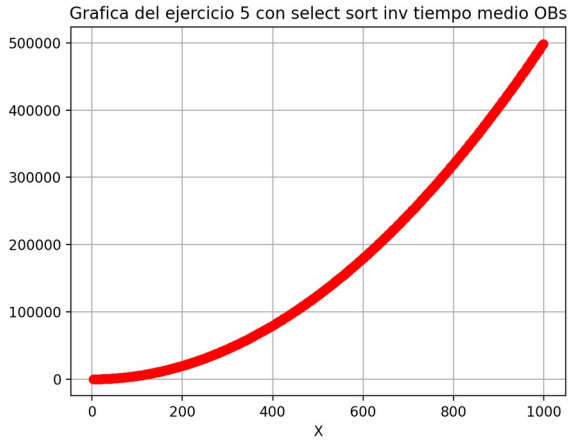
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
[Running] cd "/home/jorge/Escritorio/INGENIERÍ
ANAL/PRÁCTICA/P1AALG/"sorting
5
4
3
2
1

[Done] exited with code=0 in 0.054 seconds
```

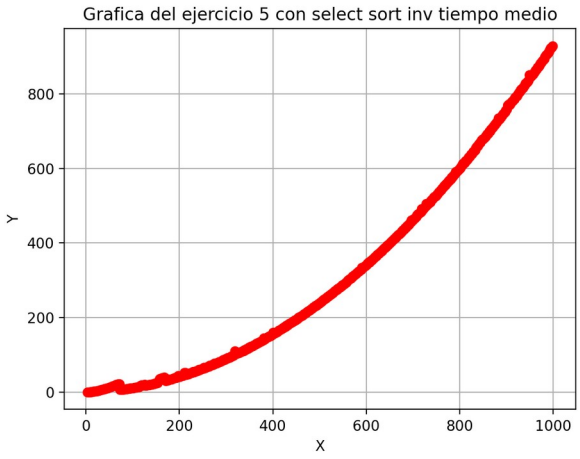
Gráfica comparando el tiempo medio de OBs para SelectSort y SelectSortInv, comentarios a la gráfica.





Como podemos comprobar ambas gráficas son iguales, ya que en ambas funciones se realizan el mismo número de Obs.

Gráfica comparando el tiempo medio de reloj para SelectSort y SelectSortInv, comentarios a la gráfica.



Ambas son muy semejantes y varían muy poco una respecto la otra, siguen ambas la gráfica de x^2.

6. Respuesta a las preguntas teóricas.

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

6.1 Pregunta 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:

La clave esta en que rand() genera siempre un numero aleatorio a partir de la semilla en srand(). Por lo tanto conseguimos la aleatoriedad que deseamos, y luego le aplicamos las operaciones para conseguir el número en el rango dado. Nos apoyamos en stackoverflow.com, en un par de artículos sobre rand() en C. Otra implementación podría ser por ejemplo, que srand() en vez de time(NULL) use otro parámetro para establecer la semilla, pero probablemente sería menos aleatorio que con time(NULL) ya que cambia muy rápido.

6.2 Pregunta 2

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

Ordena bien ya que se asegura de que en cada iteración, el mínimo se encuentra al principio. Esto es así porque para encontrar el mínimo en cada iteración compara con todos los elementos restantes para asegurarse de que lo encuentra. No es el más eficiente, pero es fácil de reconocer porque funciona.

6.3 Pregunta 3

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

Porque al solo quedar uno, el mínimo entre el y los restantes (que es solo el) y el único que puede tomar la posición es el.

6.4 Pregunta 4

¿Cuál es la operación básica de SelectSort?

Es la comparación de claves para encontrar el mínimo en el bucle de la función min.

6.5 Pregunta 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 siempre que se pueda:

El funcionamiento del algoritmo selectSort hace que su número de comparaciones de clave sea siempre el mismo si la entrada es igual. Es decir, da igual que la tabla esté ordenada o desordenada, siempre hace N(N-1)/2 comparaciones de clave, siendo N el tamaño de la entrada. Por lo tanto aumenta con el tamaño de entrada siguiendo esa fórmula. Con notación asintótica se podría decir que $O(n^2)$.

6.6 Pregunta 6

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

Ambas gráficas son iguales, realizan el mismo número de Obs y el tiempo medio es muy similar.

7. Conclusiones finales.

Se puede concluir que el SelectSort no es el algoritmo de ordenación más eficiente que existe, pero ha sido una gran práctica para aprender a poder saber analizar el algoritmo y para poder compararlo con los otros.