Análisis de Algoritmos 2023/2024

Práctica 1

Iker González Sánchez, Grupo 1261.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica hemos abarcado distintos temas, entre ellos: la generación de permutaciones, donde hemos tenido que diseñar diversas funciones para generar números y permutaciones; el algoritmo SelectSort; almacenar y estudiar tiempos de ejecución; además de responder varias cuestiones.

2. Objetivos

2.1 Apartado 1

Implementar una función que genere números aleatorios equiprobables en un rango dado.

2.2 Apartado 2

Implementar una función que genere permutaciones de números aleatorios con la función anteriormente hecha.

2.3 Apartado 3

Implementar una función que genera mediante la función generate perm del ejercicio anterior n perms permutaciones equiprobables de N elementos cada una.

2.4 Apartado 4

Implementar una función que aplique el algoritmo de ordenación SelectSort.

2.5 Apartado 5

Guardar los tiempos de ejecución, en específico guardar el número de permutaciones promediadas, el tamaño de las permutaciones, el tiempo medio de ejecución (en segundos), el número promedio de veces que se ejecutó la OB, el número mínimo de veces que se ejecutó la OB y el número máximo de veces que se ejecutó la OB.

2.6 Apartado 6

Implementar SelectSortInv, que es como SelectSort pero ordenando de mayor a menor, y comparar los resultados con los de SelectSort.

3. Herramientas y metodología

El entorno de desarrollo seleccionado ha sido linux-ubuntu, haciendo uso de VSCode y el terminal.

3.1 Apartado 1

Se han planteado diversas opciones para este primer apartado, como utilizar un bucle. Finalmente estudiando el comportamiento de la función rand(), la propuesta consiste en calcular el intervalo entre el límite inferior y el superior y, después sumar s el límite inferior para que el número aleatorio final, esté dentro del rango deseado

3.2 Apartado 2

Mediante el pseudocódigo dado se ha implementado la función de generar permutaciones dado un N número de atributos, para ello se ha reservado memoria con malloc, que es liberada en el ejercicio correspondiente. Se utiliza un bucle que llama a la función anterior con el fin de colocar en posiciones aleatorias los números de la permutación.

3.3 Apartado 3

Utilizamos un doble puntero y reservamos memoria que será liberada en el ejercicio correspondiente. Se utiliza un bucle que llama a la función anterior enviando N.

3.4 Apartado 4

Para la implementación de este apartado se ha buscado información sobre su pseudocódigo, para su correcta implementación.

3.5 Apartado 5

Para este apartado utilizamos la función del apartado 3 para medir sus tiempos de ejecución, operaciones básicas y otros datos. También nos aseguramos de liberar memoria correctamente tanto de la que reservamos en estas funciones como la de las funciones anteriores. Además para la función que imprime por pantalla hicimos una versión más estética, pero que descartamos(esta comentada) dado que no cumplía con los estándares de gnuplot.

3.6 Apartado 6

Basándonos en la función de SelectSort implementada, simplemente cambiamos el bucle para que empezase en la última posición y fuese bajando.

4. Código fuente

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

```c
int random_num(int inf, int sup)

{

  if(inf > sup) return ERR;



  int num;

  int dif = sup - inf;



  num = rand() % (dif+1) + inf;



  return num;

}
```

4.2 Apartado 2

```c
int* generate_perm(int N)

{

  if(N < 1) return NULL;



  int i = 0, randomNum = 0, aux = 0;



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

  if (perm == NULL) {

    return NULL;

  }



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

```
    perm[i] = i;

  }


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

    randomNum = random_num(i, N-1);

    aux = perm[i];

    perm[i] = perm[randomNum];

    perm[randomNum] = aux;


  }


  return perm;

}
```

4.3 Apartado 3

```
int** generate_permutations(int n_perms, int N)

{

  if(n_perms < 1 || N < 1) return NULL;


  int i = 0;

  int **perm = NULL;


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


  if (perm == NULL) {
```

```
    return NULL;
  }

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

  return perm;
}
```

4.4 Apartado 4

```
int SelectSort(int* array, int ip, int iu)
{
  if (!array || ip < 0 || iu < 0 || ip > iu) return ERR;

  int aux = 0, i = 0, ob = 0, imin = 0;

  for(i = ip; i < iu; i++) {
    imin = min(array, i, iu);

    if(i != imin){
      aux = array[i];
```

```
      array[i] = array[imin];

      array[imin] = aux;

    }

    ob+=iu-i;

  }


  return ob;

}

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

{

  if(!array || ip < 0 || iu < 0 || ip > iu) return ERR;


  int i = 0, min = ip;


  for(i = ip + 1; i <= iu; i++) {

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

  }


  return min;

}
```

4.5 Apartado 5

```
short average_sorting_time(pfunc_sort metodo,

                           int n_perms,

                           int N,
```

```
                            PTIME_AA ptime)

{

  if(!metodo || n_perms < 1 || N < 1 || !ptime) return ERR;



  ptime->N = N;

  ptime->n_elems = n_perms;

  ptime->time = 0;

  ptime->average_ob = 0;

  ptime->min_ob = -1;

  ptime->max_ob = 0;



  int i=0;

  short status = OK;




  int **perm = generate_permutations(n_perms, N);

  if(!perm) return ERR;



  for(i=0; i<n_perms && status==OK; i++) {



    clock_t t = clock();

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

    t = clock() - t;



    if(ob == ERR){
```

```
      status = ERR;
    }

    ptime->time += ((double)t);
    ptime->average_ob += ob;

    if(ptime->min_ob == -1 || ob < ptime->min_ob) ptime->min_ob = ob;
    if(ob > ptime->max_ob) ptime->max_ob = ob;
  }

  ptime->time /= n_perms;
  ptime->average_ob /= n_perms;

  for(i=0; i<n_perms; i++) free(perm[i]);
  free(perm);
  return status;
}
short generate_sorting_times(pfunc_sort method, char *file,
                             int num_min, int num_max,
                             int incr, int n_perms)
{
  if(!method || !file || num_min < 1 || num_max < num_min || incr < 1
|| n_perms < 1) return ERR;

  int veces = (num_max - num_min)/incr + 1, i;
```

```
  short status = OK;


  PTIME_AA ptime = (PTIME_AA)malloc(veces*sizeof(TIME_AA));
  if(!ptime) return ERR;


  for(i=0; i<veces && status == OK; i++){
    status = average_sorting_time(method, n_perms, num_min + i*incr,
&ptime[i]);
  }


  if(status == OK) status = save_time_table(file, ptime, veces);


  free(ptime);


  return status;
}
short save_time_table(char *file, PTIME_AA ptime, int n_times)
{
  if(!file || !ptime || n_times < 1) return ERR;


  FILE *f = fopen(file, "w");
  int status = OK, i;
  if(!f) return ERR;
  for(i = 0; i<n_times && status == OK; i++){
    if(fprintf(f, "%d,%f,%f,%d,%d\n", ptime[i].N, ptime[i].time,
ptime[i].average_ob, ptime[i].min_ob, ptime[i].max_ob)<0)
```

```
      status = ERR;

  }



  fclose(f);

  return status;

}
```

4.6 Apartado 6

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

{

  if (!array || ip < 0 || iu < 0 || ip > iu) return ERR;



  int aux = 0, i = 0, ob = 0, imin = 0;



  for(i = iu; i > ip; i--) {

    imin = min(array, ip, i);



    if(i != imin){

      aux = array[i];

      array[i] = array[imin];

      array[imin] = aux;

    }

    ob+=iu-i;

  }
```

```
  return ob;

}
```

5. Resultados, Gráficas

NOTA: Algunos tests han sido modificados para ver correctamente el funcionamiento

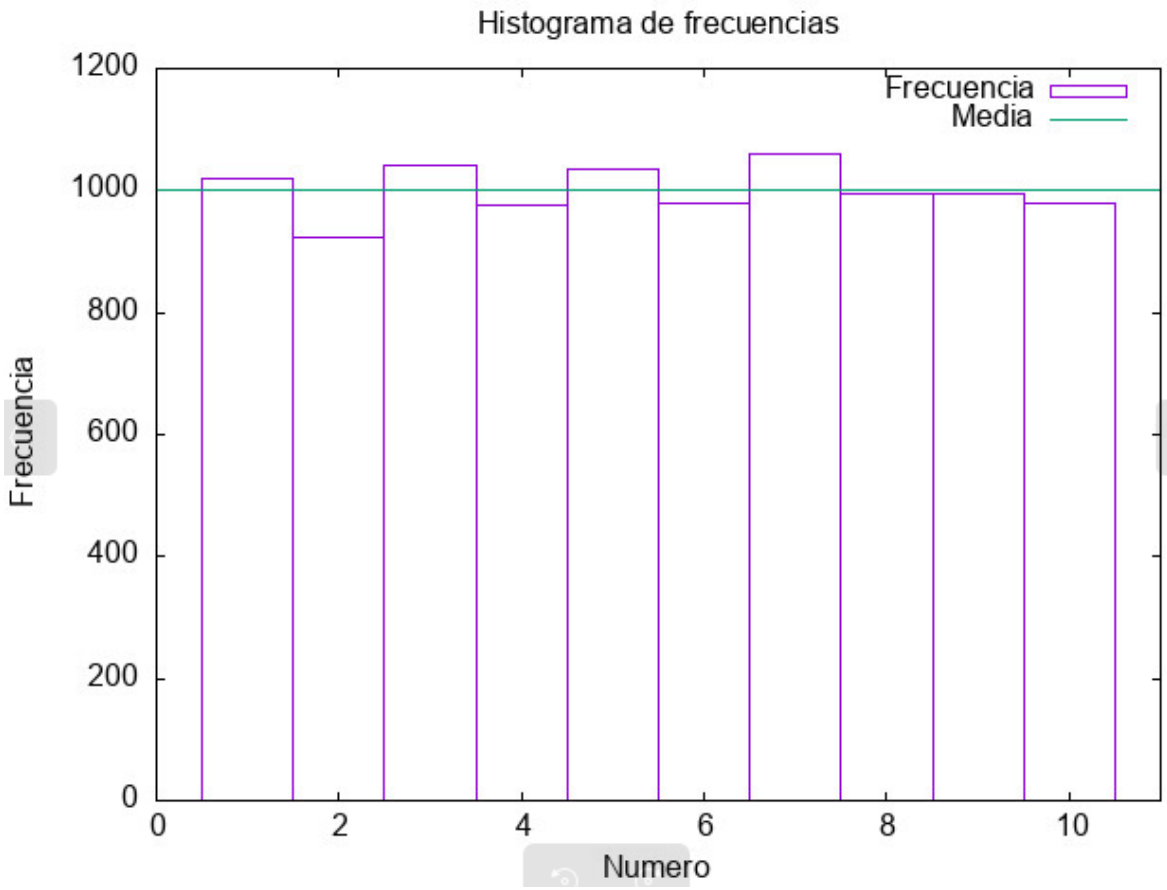
5.1 Apartado 1

```
@./exercise1 -limInf 1 -limSup 5 -numN 10
```

Resultados del apartado 1.

```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise1_test
Running exercise1
Practice no 1, Section 1
Done by: Iker González Sánchez
Group: 1261
4
5
4
1
2
4
2
5
1
5
```

Gráfica del histograma de números aleatorios, comentarios a la gráfica



Se puede observar que los datos son bastante equiprobables para cada número, como también se puede apreciar con la media.

5.2 Apartado 2

```
@./exercise2 -size 5 -numP 10
```

Resultados del apartado 2.

```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise2_test
Running exercise2
Practice number 1, section 2
Done by: Iker González Sánchez
Group: 1261
0 4 1 3 2
2 3 4 1 0
2 4 0 3 1
4 0 1 2 3
0 4 3 2 1
2 3 1 4 0
0 3 2 4 1
1 0 3 2 4
3 1 2 0 4
0 4 3 1 2
```

5.3 Apartado 3

```
@./exercise3 -size 5 -numP 10
```

Resultados del apartado 3.

```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise3_test
Running exercise3
Practice number 1, section 3
Done by: Iker González Sánchez
Group: 1261
3 2 0 4 1
3 4 0 2 1
4 3 1 2 0
3 0 4 2 1
1 3 4 2 0
4 3 1 0 2
3 2 0 1 4
4 1 0 3 2
1 3 4 0 2
1 2 0 3 4
```

5.4 Apartado 4

```
@./exercise4 -size 10
```

Resultados del apartado 4.

```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise4_test
Running exercise4
Practice number 1, section 4
Done by: Iker González Sánchez
Group: 1261
0       1       2       3       4       5       6       7       8       9
```

5.5 Apartado 5

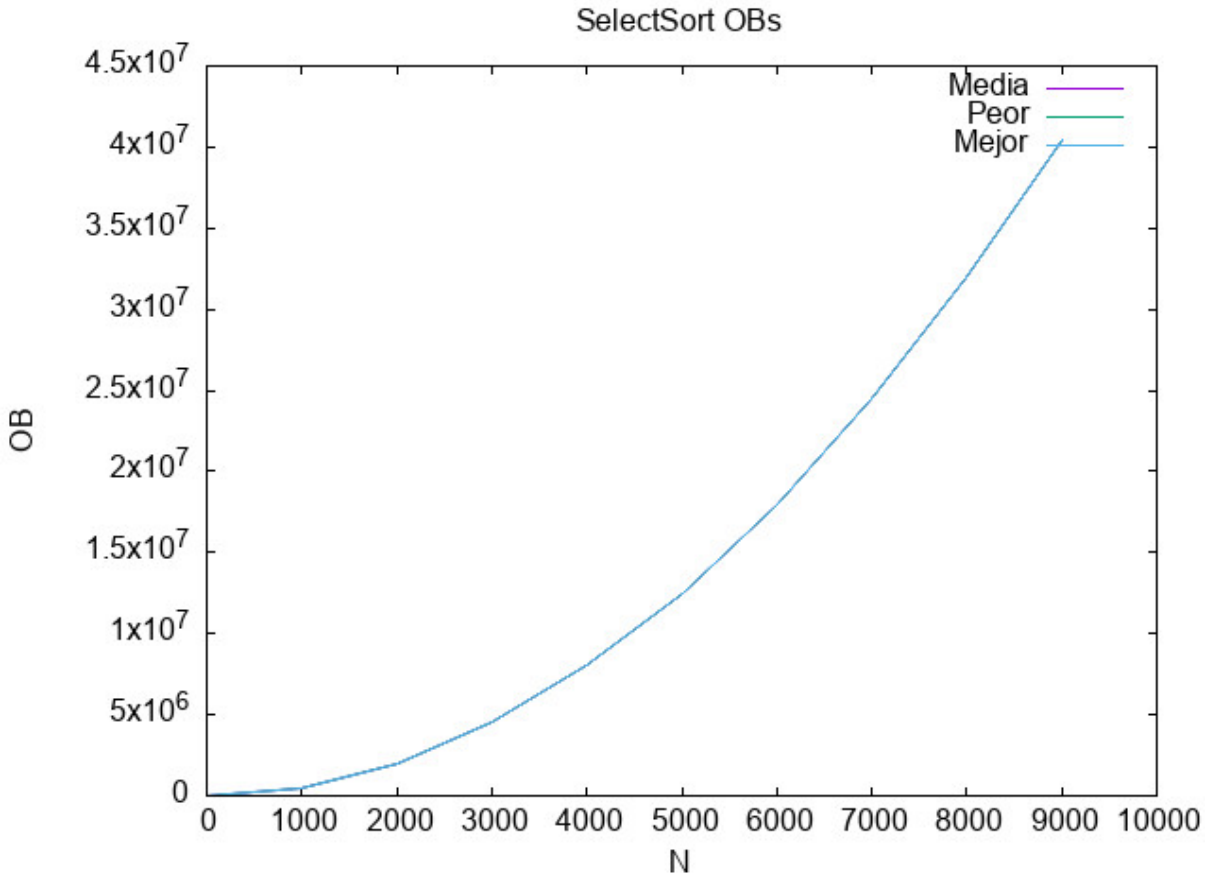
```
@./exercise5 -num_min 1 -num_max 10000 -incr 1000 -numP 5 -outputFile
exercise5.log

Usado para SelectSortInv:

@./exercise5 -num_min 1 -num_max 10000 -incr 1000 -numP 5 -outputFile
exercise5_2.log
```

Resultados del apartado 5.

```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise5_test
Running exercise5
Practice number 1, section 5
Done by: Iker González Sánchez
Group: 1261
Correct output
```

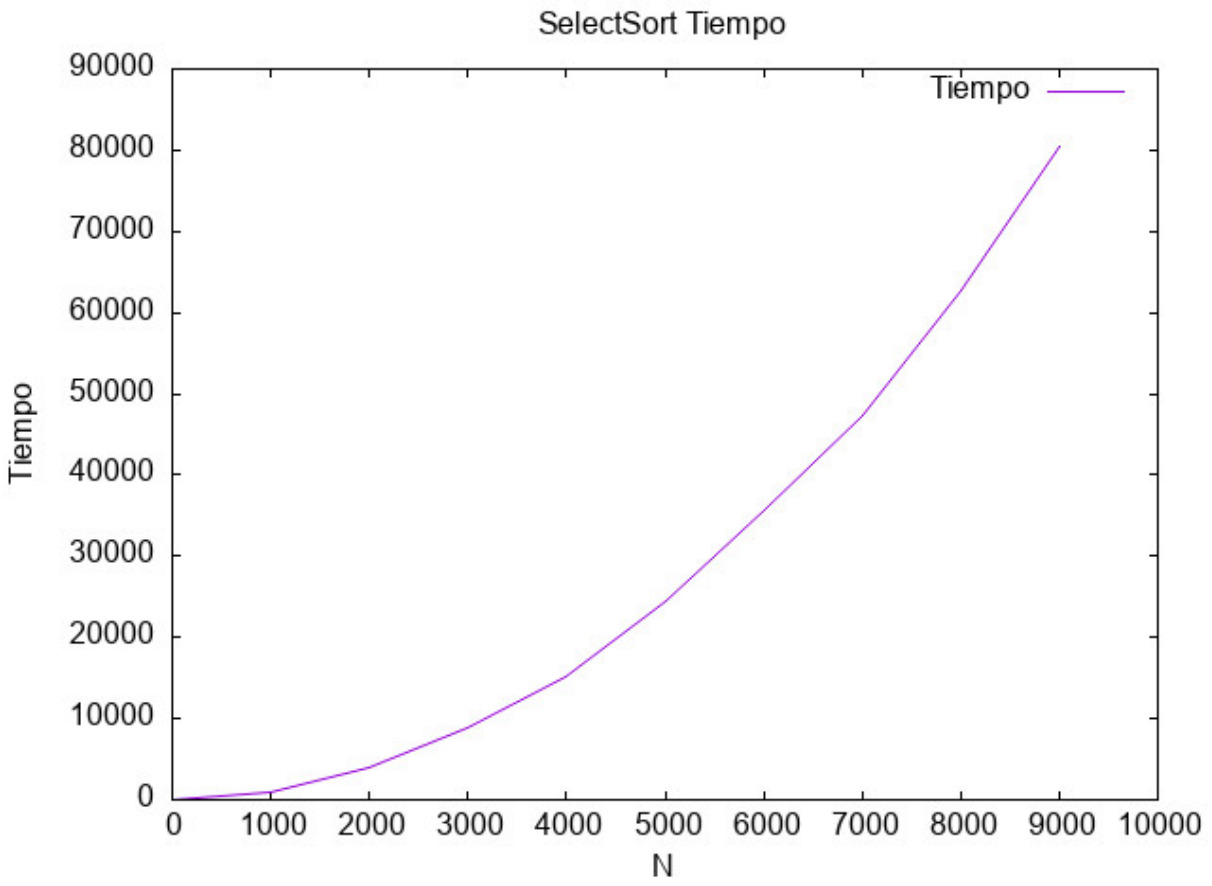
```
1,0.800000,0.000000,0,0
1001,924.000000,499500.000000,499500,499500
2001,3856.800000,1999000.000000,1999000,1999000
3001,8927.800000,4498500.000000,4498500,4498500
4001,15110.600000,7998000.000000,7998000,7998000
5001,24412.200000,12497500.000000,12497500,12497500
6001,35577.200000,17997000.000000,17997000,17997000
7001,47310.600000,24496500.000000,24496500,24496500
8001,62801.000000,31996000.000000,31996000,31996000
9001,80433.800000,40495500.000000,40495500,40495500
```

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



No se aprecian diferencias.

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



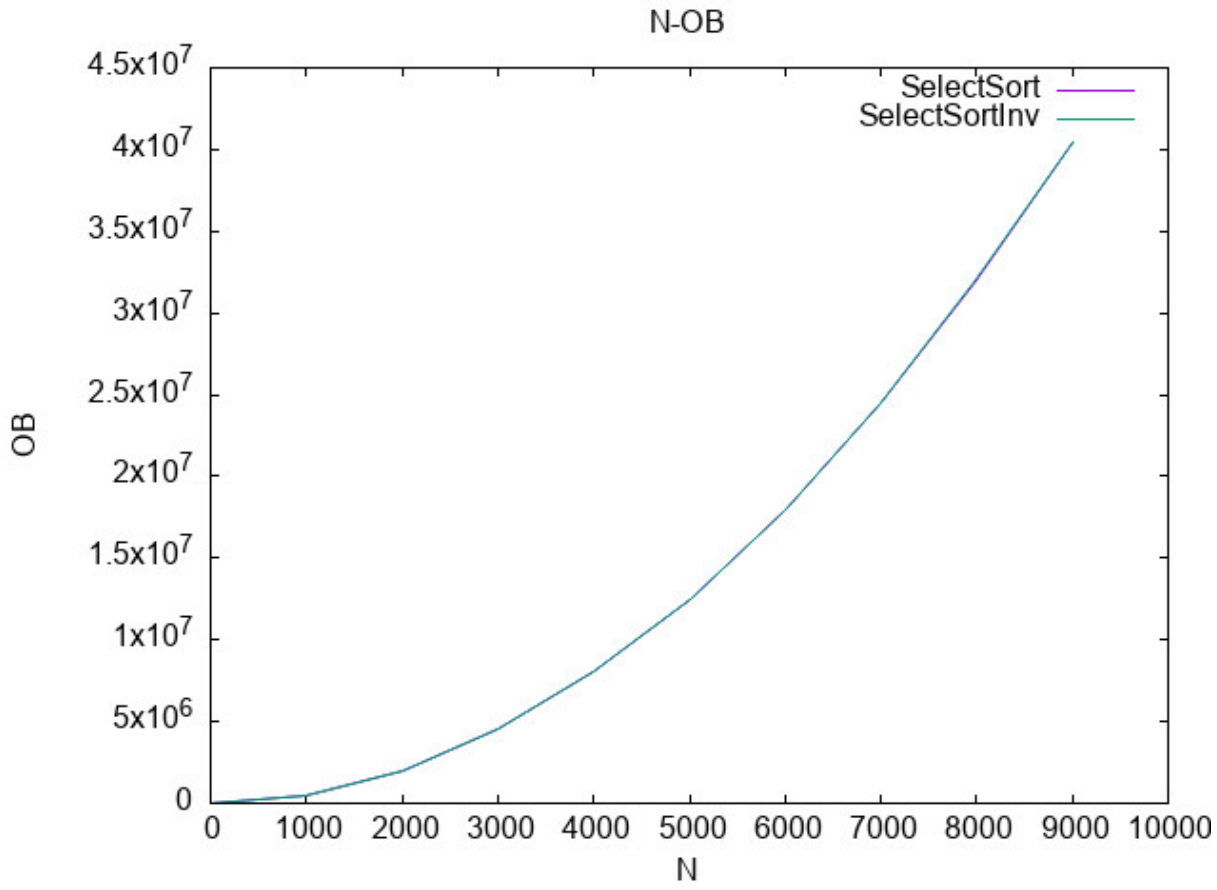
Se puede apreciar la curva del tiempo según aumenta N.

5.6 Apartado 6

Resultados del apartado 6.

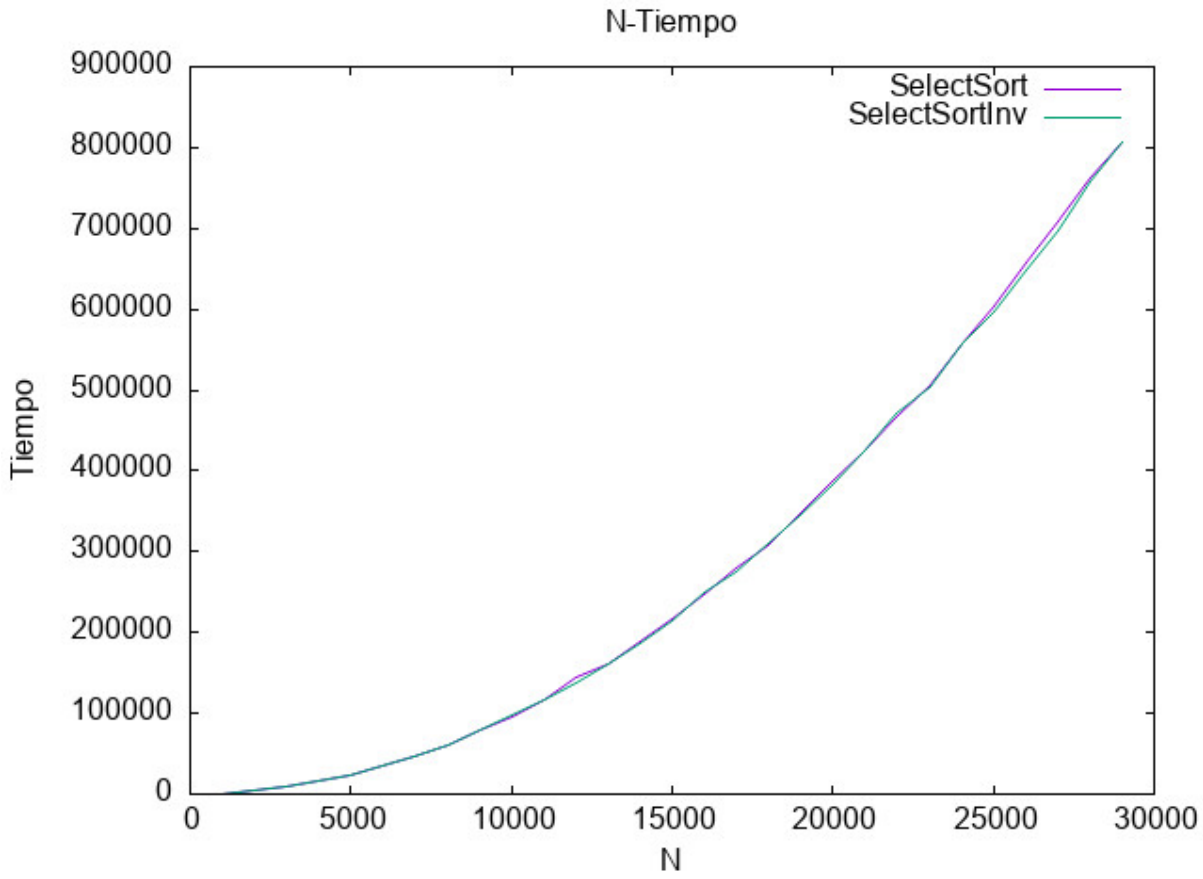
```
eps@vmlabsdocentes:~/ANAL-main/P1/P1_ANAL$ make exercise4_test
Running exercise4
Practice number 1, section 4
Done by: Iker González Sánchez
Group: 1261
9       8       7       6       5       4       3       2       1       0
```

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



Se puede apreciar que las OBs son las mismas tanto para SelectSort como para SelectSortInv

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



En esta gráfica se aprecia diferencia, que puede ser debida al ordenador y no al algoritmo en sí.

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.

Dado que la función rand() genera números entre 0 y un número dado, primero calculamos el intervalo entre el límite inferior y el superior y, después le sumamos el límite inferior para que el número aleatorio final, esté dentro del rango deseado.
Otra opción sería hacer un bucle que genere números aleatorios, descartando aquellos que no entren en el intervalo deseado. Esta sería una opción más ineficiente ya que tardaría mucho, aunque su ventaja es que todos los números del intervalo son equiprobables, cosa que no sucede con la primera opción en la mayoría de los casos(solo equiprobable si RAND_MAX es múltiplo del tamaño del intervalo).

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

Dado un conjunto de elementos a = a=[a1, a2, …, an] de tamaño n, el algoritmo de selección ordena este conjunto en orden ascendente. Se define un índice i tal que $0 \leq i < n$. Para todo j en el rango $i + 1 \leq j < n$, se cumple que $ai \leq aj$.
Es decir, en cada iteración del bucle exterior, el elemento en la posición i (donde i varía de O a n - 1) se considera el mínimo provisional. Luego, se busca en la sublista no ordenada (elementos desde la posición i + 1 hasta n) para encontrar el mínimo real. Si se encuentra un elemento aj que es menor que ai, se intercambian los valores en las posiciones i y j. Al final de cada iteración del bucle exterior, el conjunto de elementos desde la posición O hasta i está ordenado de manera ascendente.
Esta definición formal establece las condiciones y restricciones que el algoritmo de selección cumple en cada iteración, garantizando que el array esté ordenado al final del proceso.

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

Porque al haber ordenado todos los elementos de la tabla, al último elemento no le queda otra que estar ordenado también.

6.4. **¿Cuál es la operación básica de SelectSort?**

La operación básica de SelectSort es la comparación de claves.

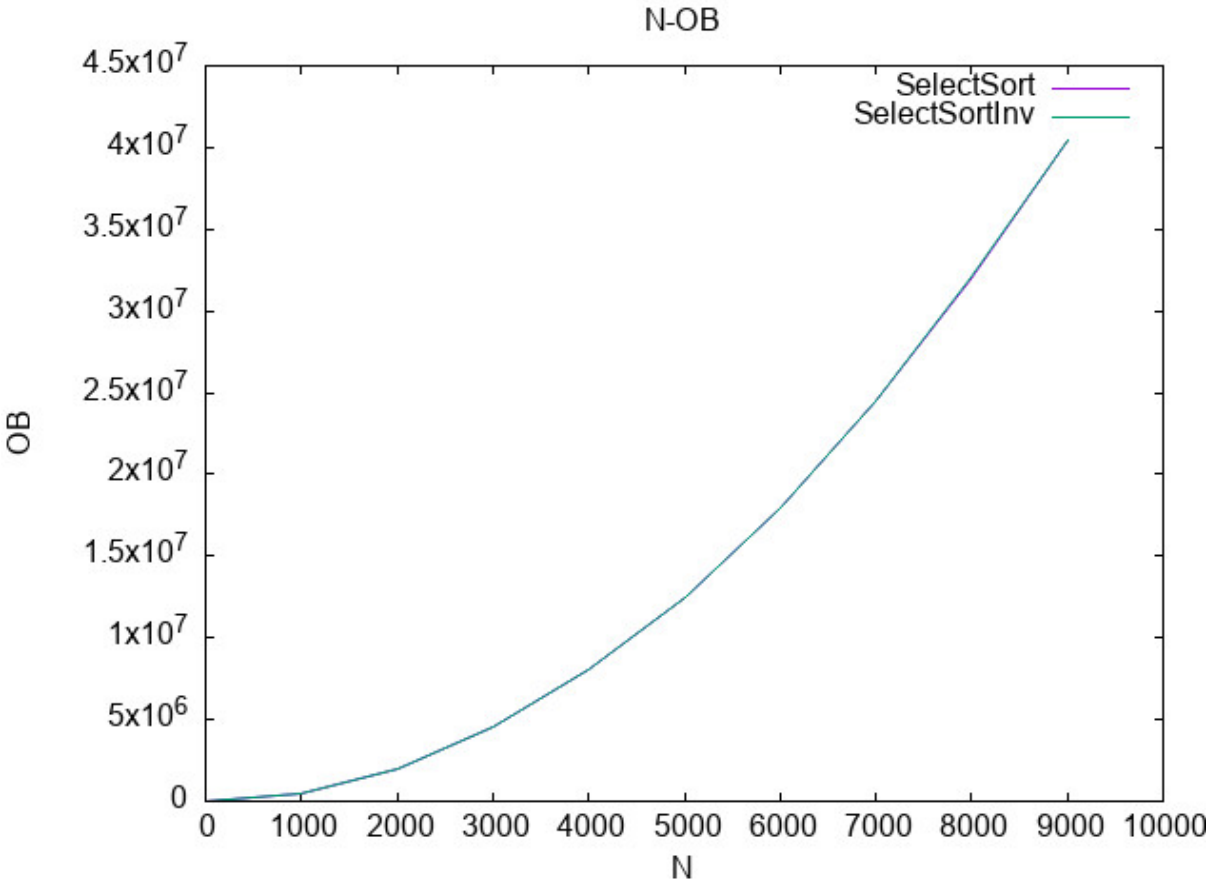
6.5. **Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS(n) y el caso mejor BBS(n) de SelectSort. Utilizad la notación asintótica (O, Θ, o, Ω,etc) siempre que se pueda.**

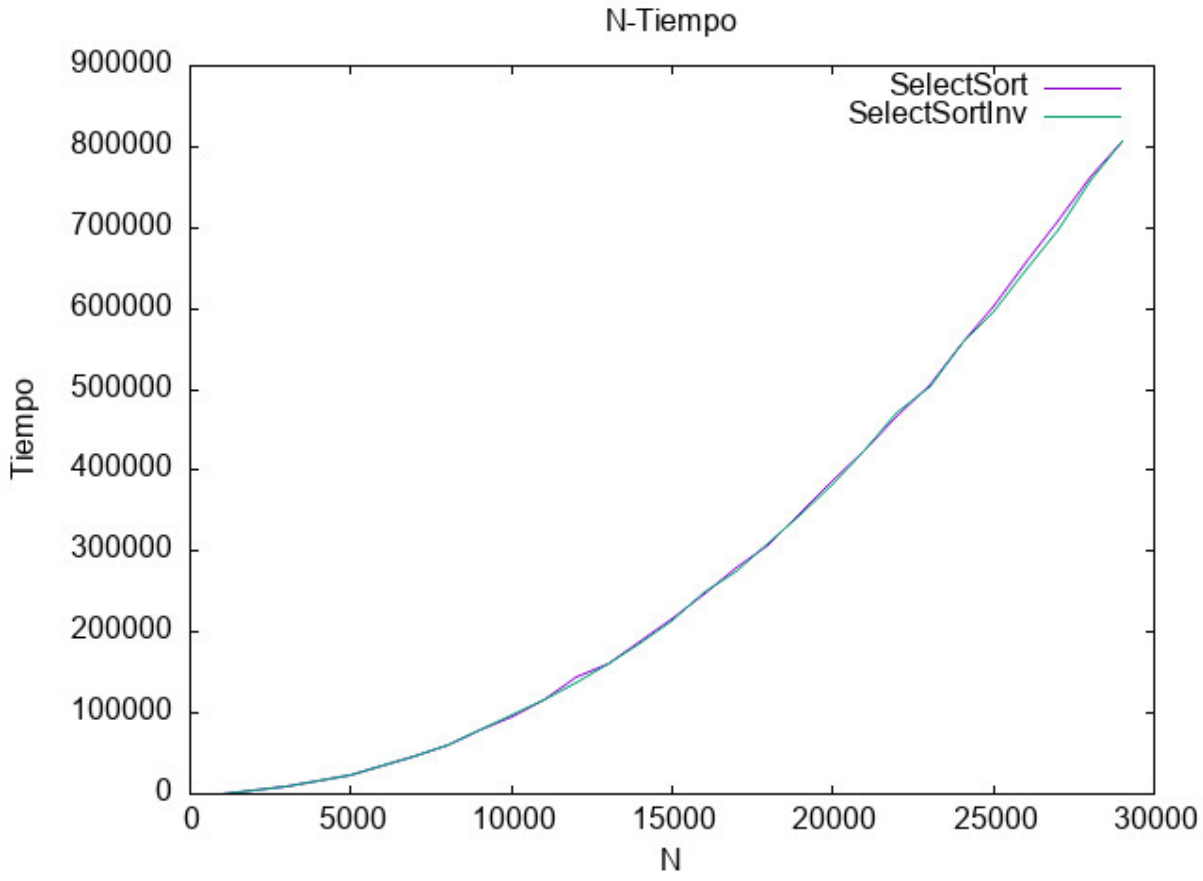
Para el SelectSort el número de comparaciones de clave(CdC) no depende del orden de los términos del vector sino del número de elementos(n), es decir, el mejor y el peor caso son equivalentes.

El algoritmo realiza $\sum_{i=1}^{n-1} i$ que se puede calcular como $\frac{n^2-n}{2}=\Theta(n^2)$

Es decir Wss(n)=Bss(n)= $\frac{n^2-n}{2}=\Theta(n^2)$

6.6. **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).**


N-OB
SelectSort
SelectSortInv
OB
N
0
5x10^6
1x10^7
1.5x10^7
2x10^7
2.5x10^7
3x10^7
3.5x10^7
4x10^7
4.5x10^7
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


N-Tiempo
SelectSort
SelectSortInv
Tiempo
N
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
0
5000
10000
15000
20000
25000
30000

Se puede observar que apenas hay diferencia en operaciones básicas, algo lógico dada nuestra explicación de la lógica del algoritmo de SelectSort. Los tiempos sí que cambian, esto es debido a que puede darse el caso de que los números generados automáticamente estén en una posición más favorable. Aunque también hay que tener en cuenta que el tiempo de ejecución es variable dependiendo del momento de ejecución.

7. Conclusiones finales.

En esta práctica hemos aprendido sobre la generación de diferentes números y permutaciones y además hemos comprobado cómo actúa el algoritmo SelectSort ordenando de manera ascendente y descendente, y podemos concluir que no hay diferencias significativas. Las diferencias se han dado en los tiempos pero siguen siendo similares por lo que sus diferencias se pueden deber a otras razones como lo que esté haciendo el ordenador en ese momento.