Análisis de Algoritmos 2022/2023

# Práctica 1

Alejandro García Hernando y Diego Rodríguez Ortiz,  Grupo 1202

| Código | Gráficas | Memoria | Total |
|---|---|---|---|
| | | | |

# 1. Introducción.

En esta práctica hemos comprobado de manera experimental lo visto en clase. Para ello, hemos diseñado un programa que implementa distintos algoritmos de ordenación aprendidos y mide el tiempo empleado por el mismo en función del tamaño de la entrada. Para ello recurrimos a los conocimientos obtenidos en la parte de teoría e hicimos una serie de aproximaciones teóricas para estimar los resultados que nuestro código iba a proporcionar.

# 2. Objetivos

## 2.1 Apartado 1

En este primer apartado desarrollamos una función que genere números enteros aleatorios entre 2 parámetros de forma equiprobable. Además, también hemos comparado dos funciones de generación y visto porque tan solo una de ellas es equiprobable.

Lo comprobamos en el fichero exercise1.c y generamos un histograma para comprobar la equiprobabilidad.

## 2.2 Apartado 2

Crear una función, que dada una lista de números, genera una permutación aleatoria de manera equiprobable. Para esta parte es esencial asegurar la equiprobabilidad de las permutaciones obtenidas, porque si no se hace adecuadamente, los datos obtenidos a lo largo de la práctica serán erróneos.

## 2.3 Apartado 3

Hacer una función, que dado el número de elementos y el número de permutaciones deseadas, genera un conjunto de permutaciones aleatorias equiprobables.

## 2.4 Apartado 4

Medir experimentalmente mediante la ordenación de un gran número de permutaciones de distintos tamaños las características del algoritmo SelectSort. En este apartado se obtendrá el tiempo medio de ejecución, el máximo, medio y menor número de

veces que se realiza operación básica del algoritmo. Esto nos permite analizar como varían estas propiedades en función del tamaño de la permutación proporcionada, y compararlo con el resultado teórico.

### 2.5 Apartado 5

Ser capaces de medir el tiempo medio de ejecucion de una función de ordenación y el número máximo, medio y mínimo de veces que se ejecuta la OB, dependiendo de los distintos tamaños de listas a ordenar. Y ver que tanto el tiempo medio como el número medio de operaciones básicas ejecutadas asciende con el cuadrado de la longitud de las permutaciones proporcionadas.

### 2.6 Apartado 6

Usando todas las herramientas desarrolladas anteriormente compararemos la eficiencia de los algoritmos SelectSort y SelectSortInv. Dado que el algoritmo es muy similar, debemos comprobar si las pequeñas diferencias afectan significativamente al coste de ejecución.

## 3. Herramientas y metodología

Cómo ambos tenemos ordenadores Windows, hemos decidido programar en Visual Code y ejecutar nuestros programas en Ubuntu. Además, para realizar las gráficas hemos empleado gnuplot y hemos comprobado que no haya fugas de memoria con Valgrind.

### 3.1 Apartado 1

Para obtener la función generadora de números aleatorios nos hemos basado en el procedimiento descrito en el capítulo 7 del libro "Numerical recipes in C". Creemos que es una gran elección pues genera números enteros entre dos valores de forma equiprobable; a diferencia de otros programas, que pueden tener pequeñas variaciones de probabilidad que comprometen la efectividad del código.

### 3.2 Apartado 2

Para este apartado nos fue muy útil el pseudocódigo proporcionado en la documentación de la práctica. Para generar la permutación añadimos la función definida en el apartado anterior, de este modo nos asegurábamos la aleatoriedad del proceso, lo cual es esencial para el correcto funcionamiento del mismo.

Con todo esto, comprobamos que todo funciona adecuadamente con el fichero exercise2.c

## 3.3 Apartado 3

Nos limitamos a definir una tabla que fuese capaz de almacenar el conjunto de permutaciones deseado y las generamos gracias al código desarrollado para el apartado 2.

Comprobamos con el fichero exercise3.c que el conjunto de las permutaciones resultantes eran generadas equiprobablemente, demostrando de este modo que la función del apartado anterior también lo hacía.

## 3.4 Apartado 4

Para hacer la función de ordenación nos basamos en las diapositivas facilitadas por la rama teórica, dónde aparece el pseudocódigo de la misma. Además, creamos la función auxiliar min.

Este código nos permitió obtener el número de operaciones básicas realizadas para ordenar cada array.

## 3.5 Apartado 5

En primer lugar, incluimos la estructura proporcionada en la documentación a los archivos e implementamos las 3 funciones indicadas. Todo ello sumado al código generado en los ejercicios anteriores que nos facilitó enormemente la tarea.

En último lugar, añadimos la función clock no sin antes consultar variedad de documentación acerca de la misma, y comprobamos que el programa funcionaba correctamente con el fichero exercise5.c

## 3.6 Apartado 6

Para realizar la función SelectSortInv empleamos un algoritmo muy similar al proporcionado en clase sobre el SelectSort. El cambio principal de esta nueva implementación es empezar por el límite superior en lugar del inferior.

# 4. Código fuente

## 4.1 Apartado

```c
int random_num(int inf, int sup) /*genera un numero entero aleatorio entre el inf y el sup*/
{
    if (inf > sup){ /*Control de errores*/

        return -1;
    }

    return inf + (int)((sup - inf + 1.0) * rand() / (RAND_MAX + 1.0)); /*Formula del libro*/
}
```

## 4.2 Apartado 2

```c
/*Cambia el valor almacenado en cada variable por el de la otra*/

void swap(int *x, int *y) {
    int aux;

    if(!x||!y) return;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

```c
/*Genera una permutacion de tamanyo N ordenado aleatoriamente de forma equiprobable*/
int *generate_perm(int N) {

    int i;
    int *perm=NULL;

    if (N < 1) /*Control de errores*/
        return NULL;

    perm = (int *)malloc(N * sizeof(int)); /*Reserva de memoria*/
    if (!perm) /*Control de errores*/
        return NULL;

    for (i = 0; i < N; i++) /*Almacena en cada posicion de la permutacion el valor de su posicion mas 1*/
    {
        perm[i] = i + 1;
    }
    for (i = 0; i < N; i++) /*Desordena la permutacion generada anteriormente*/
    {
        swap(&perm[i], &perm[random_num(i, N - 1)]);
    }

    return perm; /*Devuelve el puntero de la permutacion*/
}
```

## 4.3 Apartado 3

```c
int **generate_permutations(int n_perms, int N)
{
    int **tabla = NULL;
    int i;

    if (n_perms < 1 || N < 1) /*Control de errores*/
        return NULL;
    /*Reservamos la memoria*/
    tabla = (int **)malloc(n_perms * sizeof(int *));
    if (!tabla) /*Controlamos que la memoria se haya reservado adecuadamente*/
        return NULL;

    for (i = 0; i < n_perms; i++) /*Generacion de n_perms permutaciones aleatorias*/
    {
        tabla[i] = generate_perm(N);
        if (!tabla[i]) /*Control de errores*/
        {
            for (i = i - 1; i >= 0; i--)
            {
                free(tabla[i]);
            }
            free(tabla);
            return NULL;
        }
    }

    return tabla;
}
```

## 4.4 Apartado 4

```c
int min(int *array, int ip, int iu)
{
  int i, min;
  if (!array || ip > iu || ip < 0) /*Control de errores*/
    return ERR;

  min = ip;
  for (i = ip; i <= iu; i++) /*Compara todos los valores de la lista proporcionada para encontrar el minimo*/
  {
    if (array[min] > array[i])
    {
      min = i;
    }
  }
  return min;
}
```

```c
/*Cambia los valores almacenados de una variable en la de la otra*/
void swap1(int *x, int *y)
{
  int aux = *x;
  *x = *y;
  *y = aux;
}
```

```c
int SelectSort(int *array, int ip, int iu)
{
  int i, m, cont = 0;

  if (!array || ip < 0 || iu < 0 || iu < ip) /*Control de errores*/
    return ERR;

  for (i = ip; i < iu; i++) /*Ordenacion*/
  {
    m = min(array, i+1 , iu); /*Almacenamiento del minimo*/
    cont += iu - i;
    if (m == -1) /*Control de errores*/
      return ERR;

    swap1(&array[m], &array[i]);
  }
  return cont;
}
```

## 4.5 Apartado 5

```
short average_sorting_time(pfunc_sort metodo, int n_perms, int N, PTIME_AA ptime)
{
  int i, **p = NULL, suma_ob = 0, n_ob = 0, min_ob, max_ob = 0;
  clock_t t1, t2;

  if (n_perms < 1 || N < 1 || !ptime || !metodo)
    return ERR; /*Control de errores*/

  p = generate_permutations(n_perms, N); /*Generacion de permutaciones y control de errores*/
  if (!p)
    return ERR;

  t1 = clock(); /*Almacenamiento del tiempo al comenzar*/
  /*LLamamos a la función una primera vez para inicializar los valores*/
  n_ob = metodo(p[0], 0, N - 1);
  if ((n_ob < 0)) /*Control de errores*/
  {
    for (i = N - 1; i >= 0; i--)
    {
      free(p[i]);
    }
    free(p);
    return ERR;
  }
  /*Inicializamos los valores*/
  min_ob = n_ob;
  max_ob = n_ob;
  suma_ob += n_ob;
  /*Seguimos ejecutando metodo n_perms veces*/
  for (i = 1; i < n_perms; i++)
  {

    n_ob = metodo(p[i], 0, N - 1);
    if ((n_ob) < 0) /*Control de errores*/
    {
      for (i = n_perms - 1; i >= 0; i--)
      {
        free(p[i]);
      }
      free(p);
      return ERR;
    }

    if (n_ob < min_ob) /*Actualizacion del min_ob si es pertinente*/
    {
      min_ob = n_ob;
    }
    else if (n_ob > max_ob) /*Actualizacion del max_ob si es pertinente*/
    {
      max_ob = n_ob;
    }
    suma_ob += n_ob;
  }
  t2 = clock(); /*Almacenamiento del tiempo al finalizar*/

  /*Almacenamiento de las caracteristicsd de la funcion de ordenacion indicada con los parametros dados*/
  ptime->time = (double)(t2 - t1) / (double)n_perms;
  ptime->N = N;
  ptime->n_elems = n_perms;
  ptime->average_ob = (double)suma_ob / n_perms;
  ptime->min_ob = min_ob;
  ptime->max_ob = max_ob;

  for (i = 0; i <n_perms; i++) /*Liberacion de la memoria*/
      {
        free(p[i]);
      }
  free(p);
  return OK;
}
```

```c
short generate_sorting_times(pfunc_sort method, char *file, int num_min, int num_max, int incr, int n_perms)
{
  TIME_AA *ptime = NULL;
  int i, j, flag, tam;

  if (!file || num_min < 0 || num_max < 0) /*Control de errores*/
    return ERR;



  tam = (num_max - num_min) / incr + 1; /*Reserva dinamica de la tabla de datos*/

  ptime = (TIME_AA *)calloc(tam, sizeof(TIME_AA)); /*Reserva de memoria*/
  if (!ptime)
    return ERR;

  for (i = num_min, j = 0, flag = 0; i <= num_max; j++, i += incr)
  { /*Ordenacion y almacenamiento de datos*/
    flag = average_sorting_time(method, 100, i, &ptime[j]);
    if (flag == -1)
    {
      free(ptime);
      return ERR;
    }
  }

  if (save_time_table(file, ptime, tam) < 0) /*Guardar en fichero controlando errores*/
  {

    free(ptime);
    return ERR;
  }

  free(ptime); /*Liberacion de memoria*/
  return OK;
}


short save_time_table(char *file, PTIME_AA ptime, int n_times)
{
  FILE *f = NULL;
  int i;
  if (!file || !ptime || n_times < 1) /*Control de errores*/
    return ERR;

  f = fopen(file, "w"); /*Apertura de archivo*/
  if (!f)
    return ERR;

  for (i = 0; i < n_times; i++) /*Escritura de la tabla en el fichero proporcionado*/
  {
    if (5 > fprintf(f, "%d    %f    %f    %d   %d\n", ptime[i].N, ptime[i].time, ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob))
    {
      fclose(f);
      return ERR;
    }
  }
  fclose(f);

  return OK;
}
```

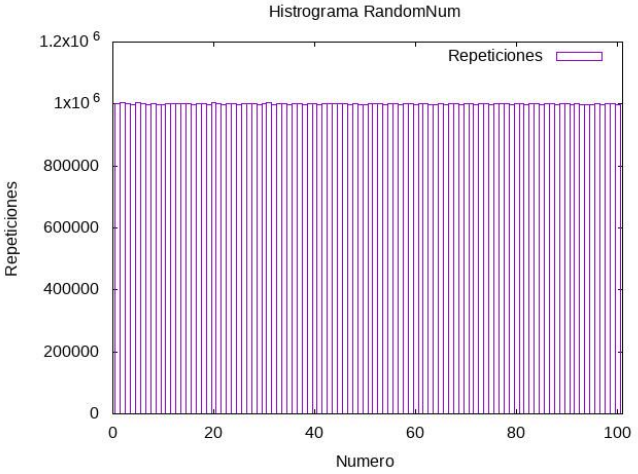
## 4.6 Apartado 6

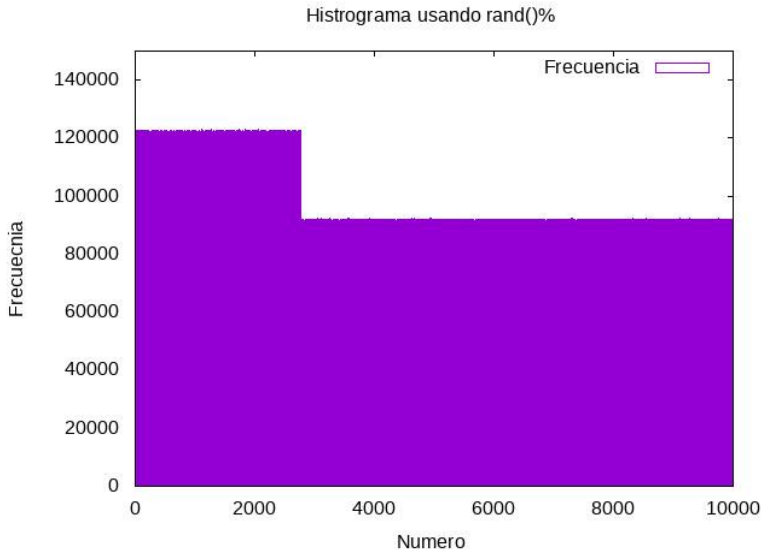
```c
int SelectSortInv(int *array, int ip, int iu)
{
  int i, m, cont = 0;

   if (!array || ip < 0 || iu < 0 || iu < ip) /*Control de errores*/
    return ERR;

  /*Ordenacion al contrario que el SelectShort porque el indice comienza por el ultimo y acaba en el primero*/
  for (i = iu; i > ip; i--)
  {

    m = min(array, ip , i-1); /*Almacenamiento del minimo*/
    cont += i - ip;
    if (m == -1) /*Control de errores*/
      return ERR;

    swap1(&array[m], &array[i]);
  }
  return cont;
}

int min(int *array, int ip, int iu)
{
  int i, min;
  if (!array || ip > iu || ip < 0) /*Control de errores*/
    return ERR;

  min = ip;
  for (i = ip; i <= iu; i++) /*Compara todos los valores de la lista proporcionada para encontrar el minimo*/
  {
    if (array[min] > array[i])
    {
      min = i;
    }
  }
  return min;
}
```

# 5. Resultados, Gráficas

## 5.1 Apartado 1

En este primer histograma se puede comprobar que la función que hemos empleado para generar números aleatorios, nos devuelve valores de forma equiprobable. Todos los números entre el 0 y el 100 aparecen prácticamente el mismo número de veces, a diferencia de otros algoritmos de generación aleatoria que no lo hacen de forma equiprobable (como veremos en la siguiente gráfica).




Como puede verse en el histograma inferior, la función de generación de números aleatorios señalada en el libro como errónea, no genera valores de forma equiprobable.

Pues es más probable que la función nos devuelva un número bajo que uno alto.

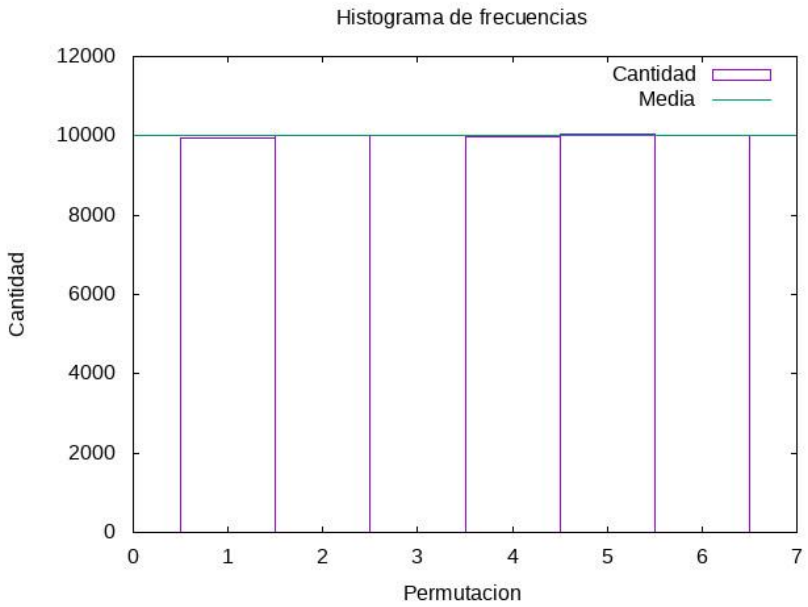
## 5.2/3 Apartado 2 y 3

Para comprobar el correcto funcionamiento de los algoritmos utilizados hemos visto que las permutaciones de tamaño N se generaban de forma equiprobable, pues puede ser que el algoritmo que las desordena tenga tendencia a devolver unas permutaciones sobre otras.

```
int fact( int n){
    if(n==0) return 1;

    return n*fact(n-1);
}
int elev(int b, int e){
    int i,k=1;
    for(i=1; i<=e;i++)
        k*=b;

    return k;
}
int desc(int n, int b){
    int i, suma=0, pot;
    for(i=b-1; i>=0; i--){
        pot=elev(b, i);
        while((n-pot)>=0){
            n-=pot;
            suma+=elev(10, i);
        }
    }

    return suma;
}
```

```
for(i=0;i<N_perms;i++){//Convertimos las permutaciones a base N_elem y almacenamos el numero d
    for(j=N_elem-1,suma=0;j>=0; j--){
        suma+=(tabla[i][j]-1)*elev(N_elem,N_elem-j-1);
    }
    t[suma]++;
}

for(i=0;i<M;i++){
    if(t[i]){//Imprimimos y contamos todas las permutaciones que aparezcan al menos una vez
        cont++;
        fprintf(f,"%d %d\n", cont,t[i]); //para plotear este. Para ver la permutacion el infer
        //fprintf(f,"%d %d\n", desc(i,N_elem),t[i]);
    }

}
```

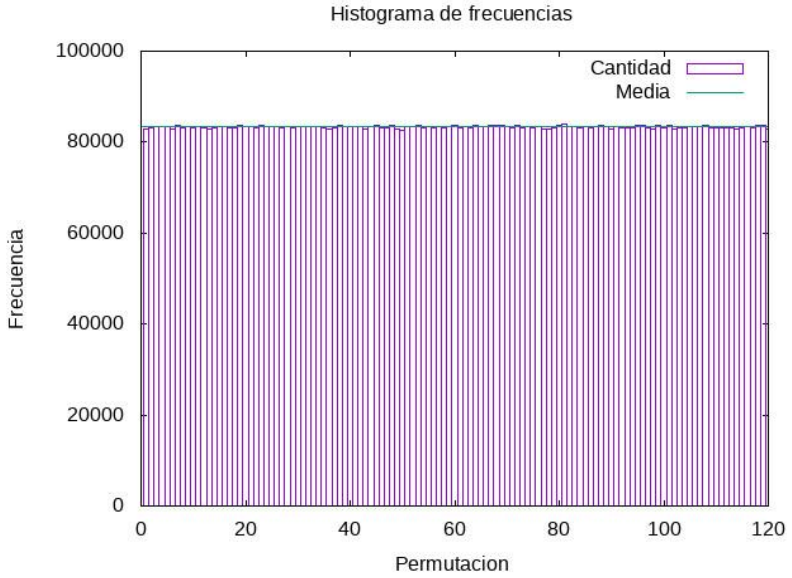
Para hacerlo, hemos transformado (utilizando el código superior) cada permutación en un número de base N siendo éste único para cada orden de números. De este modo hemos contabilizado y generado un fichero en el que se mostraba cuántas veces había aparecido cada permutación.



La gráfica superior nos muestra cuántas veces se ha generado cada una de las permutaciones de 3 elementos. El orden a seguir ha sido 123<132<213<231<312<321 (se puede desglosar el número en base N en la permutación a la que está asociado con la función inferior).

```
int desc(int n, int b){
    int i, suma=0, pot;
    for(i=b-1; i>=0; i--){
        pot=elev(b, i);
        while((n-pot)>=0){
            n-=pot;
            suma+=elev(10, i);
        }
    }

    return suma;
}
```

La gráfica que se muestra a continuación ha sido obtenida del mismo modo, únicamente aumentando el tamaño de las permutaciones a 6, creciendo a su vez el número de posibilidades a 120.

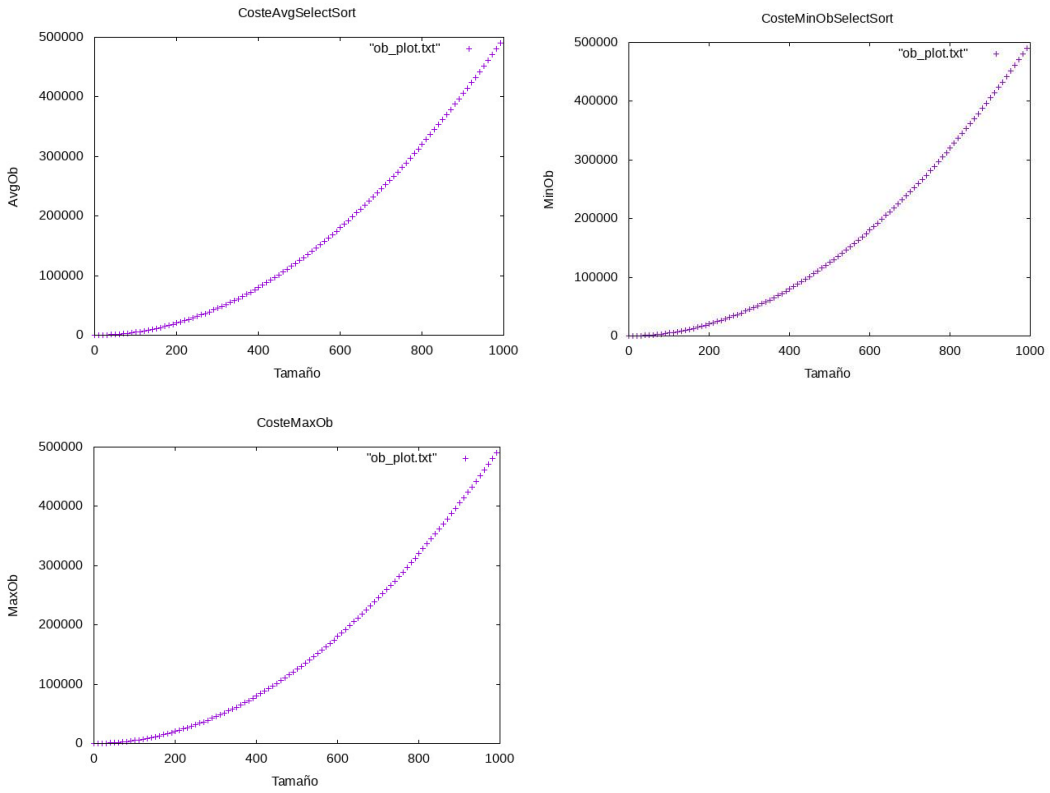



## 5.4 Apartado 4

Medimos las características del algoritmo implementado en este apartado, en las siguientes secciones.

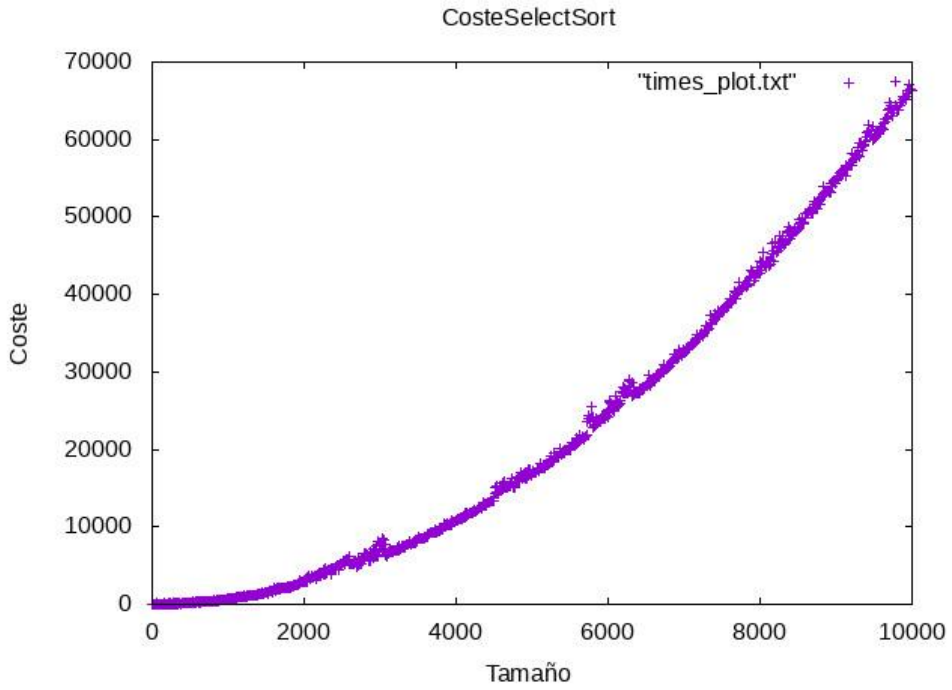
## 5.5 Apartado 5

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




Es curioso observar que las 3 gráficas son exactamente iguales. Esto se debe a que el algoritmo empleado para ordenar las permutaciones es incapaz de reconocer el desorden de una permutación. Provocando que dado un tamaño de array el número de OB ejecutadas sea siempre el mismo, sin importar si se encuentra en el caso mejor, peor o medio, obteniendo como resultado la sorprendente relación entre las gráficas superiores.

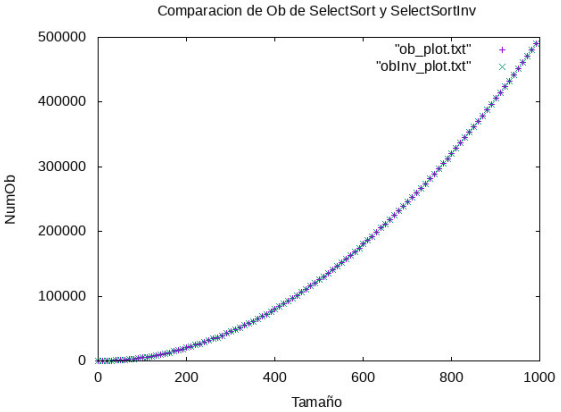
Gráfica con el tiempo medio de reloj para SelectSort, comentarios a la gráfica.




Nos encontramos ante una gráfica muy similar a la generada por la función f(x)=x^2, al igual que la gráfica generada al comparar el tiempo en OBs. Esto se debe a que si se deben ejecutar n^2 operaciones básicas (sabemos que el coste es O(n^2) porque lo hemos aprendido en la teoría y comprobado en la práctica), el tiempo también crecerá de forma cuadrática (además se demuestra en la pregunta teórica 5).

## 5.6 Apartado 6

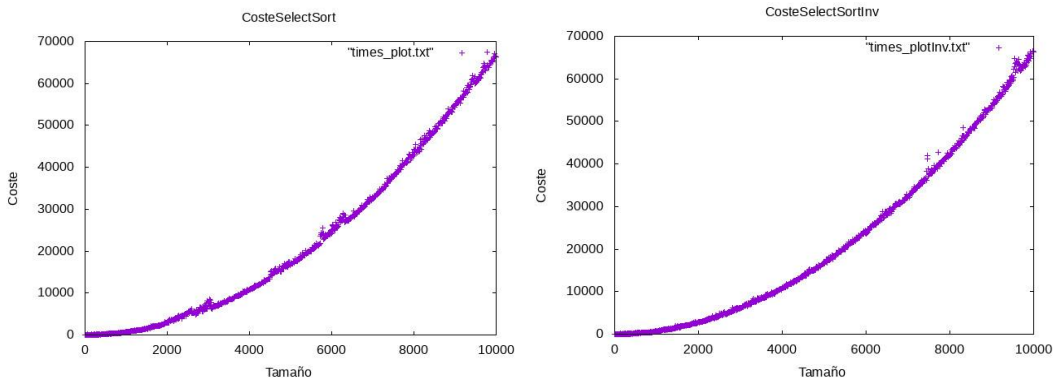
Gráfica comparando el tiempo medio de OBs para SelectSort y SelectSortInv, comentarios a la gráfica.




Como puede verse a continuación, ambas gráficas son muy similares a la de la función f(x)=x^2, por ello deducimos que su coste es O(n^2) siendo n el número de elementos del array (además el tiempo de ejecución está directamente relacionado con el número de OBs ejecutadas que sabemos que crece de forma cuadrática). Tiene sentido que ambas gráficas tengan el mismo coste pues su algoritmo es muy similar, únicamente cambia el orden en el que los índices toman los

valores (de menor a mayor en caso de SelectSort y de mayor a menor para el SelectSortInv), pero tomando n-1 valores distintos en ambos.

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




Con todo lo explicado con anterioridad, y sabiendo la relación directa que existe entre el aumento de OBs y el tiempo empleado para ejecutar el algoritmo; es fácil deducir que el tiempo empleado para ejecutar ambos algoritmos crecerá de forma polinomial. Como se puede ver en las gráficas superiores la función del tiempo medio de reloj tiene la misma forma que la del tiempo medio en OB, pues su crecimiento es idéntico, cuadrático.

# 6. Respuesta a las preguntas teóricas.

## 6.1 Pregunta 1

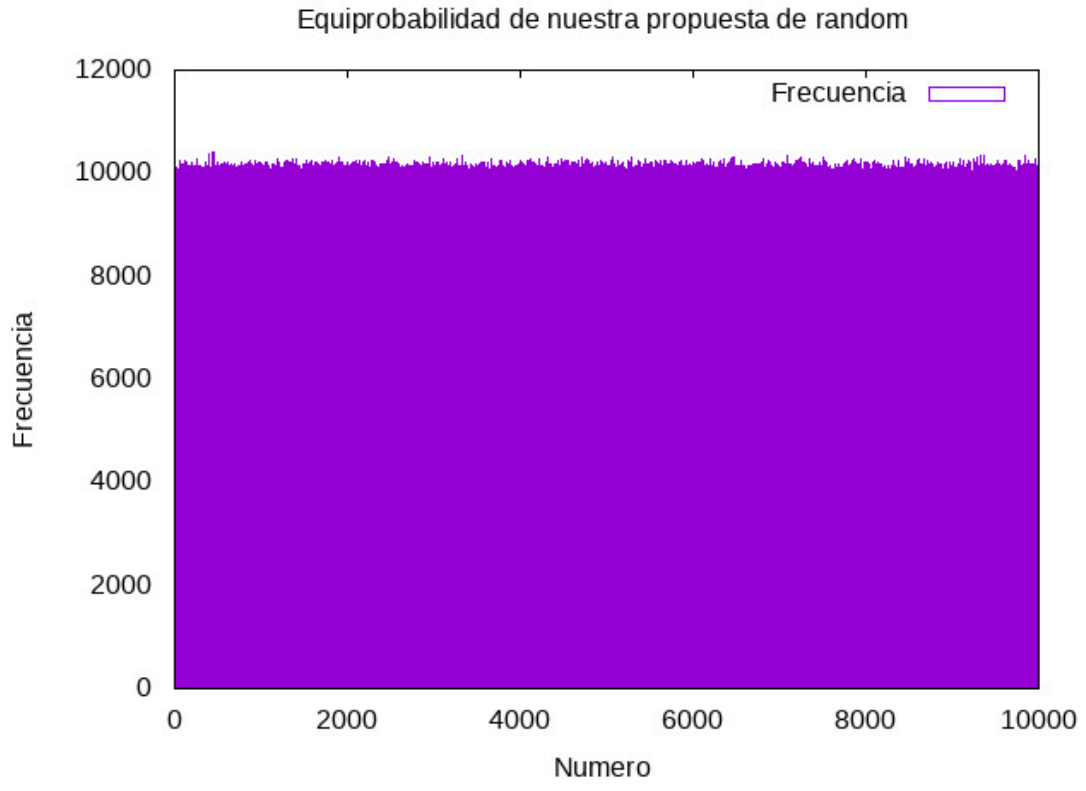
Se basa en la idea de que la función rand() devuelve un valor entre 0 y RAND_MAX, al dividirlo entre RAND_MAX+1 situamos el valor entre 0 y 1. Por último, lo situamos en el rango deseado multiplicándolo por sup - inf + 1.

Esta función la hemos definido basándonos en el procedimiento descrito en el capítulo 7 del libro "Numerical recipes in C".

Otra función para generar números aleatorios es la siguiente:

```c
int aleat_num(int inf, int sup) {
    int a=0;
    if ( inf > sup ) {
        return -1;
    }
    while(!a){
        a=  (int)rand();

        if (a>(RAND_MAX-(RAND_MAX%(sup - inf +1)))) a=0;
    }
    return inf + a%(sup - inf +1);
}
```

En este código la función rand() devuelve un número entre 0 y RAND_MAX. Para que sea equiprobable le tenemos que meter la pequeña correción de que este número no se mayor de RAND_MAX-(RAND_MAX% (sup - inf +1.0))), si lo es generamos otro número hasta que no lo sea. Posteriormente calcula el su módulo en base (sup - inf + 1.0) y para situarlo en el rango deseado le sumamos el ínfimo.



## 6.2 Pregunta 2

Usamos inducción para probar la eficacia de SelectSort sobre el tamaño (N) del array a ordenar. Para N=1 el algoritmo SelectSort no entra en el bucle, ya que i=1 no es menor que N=1 y por tanto el array ya esta ordenado. Ahora veamos que si SelectSort ordena arrays de n elementos también lo hace con arrays de n+1 elementos. En el caso de un array A de n+1 elementos tras la ejecución de la primera iteración del bucle el mínimo elemento del array A está ubicado en la primera posición. En la segunda iteración del bucle tenemos un array de N elementos a ordenar ( i=2 hasta N+1) siendo esto equivalente a un SelectSort de n elementos, luego por hipótesis de inducción sabemos que SelectSort ordena correctamente los n elementos, o sea ( A[2] < . . . <A[N] )

además sabemos que A[1] es el mínimo de todos los elementos, o sea A[1] < A[i] para todo i en particular para i=2, es decir A[1] <A[2] < . . .< A[N] o sea SelectSort ordena correctamente el array. Por tanto queda demostrado que SelectSort es eficaz.

## 6.3 Pregunta 3

No es necesario que el bucle actúe sobre el último elemento de la tabla pues cuando el bucle analiza actúa sobre el penúltimo elemento de la misma tan solo quedan los 2 elementos más grandes del conjunto. En este momento coloca a la derecha de la tabla el máximo y a su izquierda el segundo más grande, quedando de este modo ordenada sin necesidad de que el bucle actúe sobre el último elemento.

## 6.4 Pregunta 4

La operación básica del Select Sort es if (array[min] > array[i]), contenida en la función min. Esto se debe a que está en el bucle más interno del algoritmo y es la que más veces se repite. Además, es muy representativa de los algoritmos locales que resuelven este tipo de problemas, pues todos deben comparar dos elementos.

## 6.5 Pregunta 5

Para resolver esta cuestión utilizaremos los datos empleados para realizar la gráfica del apartado 5 y el conocimiento obtenido en clase. En primer lugar cabe resaltar que el tiempo de ejecución en el peor caso WBS(n) y el caso mejor BBS(n) será igual (pues como hemos comprobado anteriormente el algoritmo es incapaz de diferenciar como de ordenado está un array).

Además, sabiendo que N nos indica el tamaño de la entrada, el número de operaciones básicas que se ejecutarán (coste del SelectSort) será de N(N-1)/2. Y trataremos de demostrar que $N(N-1)/2 = \boldsymbol{\Theta} N^2$.

Demostración:

$N^2 = O(N(N-1)/2) \qquad Definimos\ \boldsymbol{\lambda} = 4\ y\ calculamos$

$4(N(N-1)/2) > N^2 \quad 2N^2 - 2N > N^2 \quad N^2 > 2N \qquad N > 2$

Vemos que es cierto para todo N>2.

$N(N-1)/2 = O(N^2)$ Definimos $\boldsymbol{\lambda} = 1\ y\ calculamos$

$(N^2 - N)/2 < N^2 \qquad -N/2 < N^2/2 \qquad 0 < N\text{^}2 + N$

Vemos que es cierto para todo N>0. Por lo tanto podemos concluir que $N(N-1)/2 = \boldsymbol{\Theta} N^2$ es cierto.

No caigamos en la tentación de pensar $N(N-1)/2N^2 \sim N^2$ pues no es cierto.

Demostración:

$$\lim_{N \to \infty} N(N-1)/(2N^2) = \lim_{N \to \infty} (N^2 - N)/(2N^2) = \lim_{N \to \infty} 1/2 - 1/(2N) = 1/2 \neq 1$$

## 6.6 Pregunta 6

Como se puede ver en las gráficas del 5.6 se aprecia que los tiempos de ejecución para un tamaño del array a ordenar son muy parecidos ambos comportándose de forma cuadrática. Observando el código proporcionado en el apartado 4.3 y 4.6 se puede argumentar que debido a las similitudes de la implementación sus tiempos de ejecución son, de hecho, asintóticamente equivalentes.

# 7. Conclusiones finales.

A lo largo de esta práctica hemos creado un método para medir el coste de los algoritmos SelectSort y SelectSortInv. En primer lugar hemos tenido que definir distintas funciones que nos permitieran generar múltiples permutaciones aleatorias de un tamaño determinado, utilizando para ello una función que nos generara números aleatorios de forma equiprobable. Posteriormente, las hemos ordenado calculando en cada una de ellas el número de operaciones básicas realizadas y el tiempo empleado en el proceso. Con toda esta información hemos realizado diversos histogramas y gráficas que nos han permitido medir las características de cada algoritmo de ordenación, como que el coste de ambos es cuadrático o cuanto tarda en ordenar una lista con un número de elementos determinado, además de todas las conclusiones escritas a lo largo de este documento.