# Análisis de Algoritmos 2019/2020

# Práctica 1

Rubén García

Elena Cano

Grupo 120

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

## 1. Introducción.

Esta práctica sirve como una primera toma de contacto para las futuros ejercicios a realizar. La podemos dividir en tres bloques, el primero consistirá en generar permutaciones de números aleatorios, en el segundo implementaremos el algoritmo de ordenación Insert Sort y una variación de este. Finalmente en el tercer bloque usando lo que hemos creado en los dos anteriores, ordenaremos tablas de permutaciones y observaremos algunos detalles como cuanto tarda en ejecutarse el programa o cuántas veces se repite la OB, entre otras cosas.

## 2. Objetivos

### 2.1 Apartado 1

En este apartado nuestro objetivo consiste en generar números aleatorios equiprobables mediante una función entre un máximo y un mínimo, que le pasamos como argumentos a dicha función.

### 2.2 Apartado 2

Nuestro objetivo es crear una función que devuelva permutaciones de un cierto número de elementos el cual le pasaremos como argumento.

### 2.3 Apartado 3

En este caso lo que queremos es generar varias  permutaciones. Le pasaremos como argumentos a la función cuantas permutaciones queremos y hasta que número tienen que ser.

### 2.4 Apartado 4

Queremos crear la función Insert Sort, que en este caso es una función de ordenación, a la cual le pasaremos una tabla que ordenará de menor a mayor y nos devolverá el número de veces que se ha ejecutado la operación básica. Como argumentos recibirá aparte de la tabla a ordenar,  el primer y el último elemento.

### 2.5 Apartado 5

Primeramente queremos crear una función a la cual le pasamos un algoritmo de ordenación y nos calcula y amacena cual es el mínimo, máximo y media  de veces que se ha ejecutado la operación básica. Además también guarda cual ha sido el tiempo medio de ejecución del algoritmo.

Una vez creada esta función nuestro objetivo es implementar otra que lo que hará es llamar varias veces a la anterior y almacenar todos esos datos en un array para luego imprimirlos en un fichero. Para ello tendremos que crear una función que se capaz de llevar a cabo esta labor.

### 2.6 Apartado 6

Queremos crear la función Insert Sort Inv a la cual le pasaremos una tabla, y que a diferencia de Insert Sort, la ordenará de mayor a menor y nos devolverá el número de veces que se ha ejecutado la operación básica. Como argumentos recibirá aparte de la tabla a ordenar,  el primer y el último elemento.

## 3. Herramientas y metodología

Para llevar acabo esta práctica hemos utilizado el entorno de Linux, usando atom para escribir el código y editarlo. Sin embargo debido a los fallos que hemos ido cometiendo hemos necesitado utilizar un depurador, en este caso hemos elegido el de Net Beans. Una vez teníamos el código depurado y funcionando correctamente pasamos Valgrind para detectar así las perdidas de memoria y corregirlas.

### 3.1 Apartado 1

Para llevar a acabo la implementación de la función de este apartado hemos tenido que llamar a la rutina C rand, de la librería stdlib, que genera números aleatorios

equiprobables entre 0 y el valor de RAND MAX definido por defecto. Obteniendo así un número aleatorio cada vez que llamemos a la función.

Para comprobar su funcionamiento ejecutamos el ejercicio1_test.c generando 100000 números entre el 1 y el 10 y viendo cuantas veces se repite cada uno, a partir de este fichero de frecuencia generamos un histograma para ver que la solución es equiprobable.

### 3.2 Apartado 2

Diseñaremos una función que reserve espacio para un array de N elementos y lo vaya rellenando con números del 1 al N. Para generar la permutación dentro de un bucle iremos cambiando el numero del primer elemento del array por otro de los que tenga detrás de forma aleatoria.

Para ver que efectivamente actuá como queremos llamamos a ejercicio2_test.c y nos devolverá tantas permutaciones como le hallamos puesto en el makefile. Podemos ver que los resultados son correctos.

### 3.3 Apartado 3

Implementamos la función reservando memoria para un array tan grande como permutaciones queremos generar. Por cada elemento del array llamamos a la función que nos genera permutaciones y le pasamos como argumento el número de elementos que queremos que haya en cada una de estas que siempre será el mismo.

Una vez mas llamamos a ejercicio3_test.c y en el makefile le indicamos cuantas permutaciones y de que tamaño queremos que genere. Si ejecutamos el comando vemos que que el resultado es el esperado.

### 3.4 Apartado 4

Inser Sort es un algoritmo que lo que hace es ir comparando uno por uno los elementos de la tabla con los anteriores. Coge un número y lo compara con el de su izquierda, si mi número es menor pongo el valor del que es mayor en mi elemento correspondiente del array y sigo comparando. Cuando encuentre alguno que sea mayor, o llegue al final de la tabla paro y meto su valor en dicha posición. De esta forma conforme se va ejecutando el algoritmo los primeros elementos de la tabla siempre están ordenandos.

En el makefile ponemos cuantos elementos queremos que tenga nuestra tabla para probar con el ejerccio4_test. Este llama a genera_perm y le pasa como argumento N creando una permutación desordenada de N elementos. Luego llama a Insert Sort y ha

un print para que nos devuelva la tabla ordenada de N elementos observando así su correcto funcionamiento.

### 3.5 Apartado 5

Creamos la función tiempo_medio_ordenacion a la cual pasamos como uno de los argumentos una nueva estructura de datos que es ptiempo donde almacenaremos una serie de valores. Para calcular min_ob y max_ob lo que hacemos es llamar a genera_permutaciones y para cada una de ellas llamar al método de ordenación y ver cuantas veces se ejecuta la OB. Al principio del bucle estableceremos que el primer OB que nos devuelva será el min_ob y el max_ob y en cada nueva iteración compararemos el valor que nos devuelve con estos. En caso de ser menor que el min_ob este pasará a tener ese nuevo valor y lo mismo ocurre si es mayor que el max_ob. Al final contaremos cuantas veces se ha ejecutado la OB y las sumaremos y dividiremos entre el numero de permutaciones para obtener medio_ob. El tiempo medio de ordenación lo obtenemos ya que pusimos un reloj mientras se ejecutaba la ordenación y lo paramos cuando finalizó. Este tiempo lo dividimos entre el número de permutaciones y así obtenemos el tiempo medio. Todos estos datos los guardamos en la estructura ptiempo.

Después hemos creado la función genera tiempos ordenación que escribe en el fichero los tiempos medios, y los números promedio, mínimo y máximo de veces que se ejecuta la OB del algoritmo de ordenación método con (n_perms) permutaciones de tamaños en el rango desde num_min hasta num_max, ambos incluidos, usando incrementos de tamaño incr. Para ello reserva memoria para un array de elementos de tipo ptiempo. Para cada índice del array llama a tiempo_medio_ordenacion y almacena los datos que le devuelve esta función en el elemento correspondiente del array. Finalmente llama a otra función que también hemos creado, guarda_tabla_tiempos, para que imprima todos los elementos almacenado en el array en un fichero que le pasaremos como argumento.

Finalmente comprobamos con ejercicio5_test.c que todas esta funciones se ejecutan correctamente. Metemos en el makefile los valores que deseamos se ejecuten ya que ejercicio5_test llamará a genera_tiempos_ordenacion y todos estos datos se imprimirán en un fichero que nosotros le pasemos. Comprobamos que se crea el fichero y se guardan correctamente todos los datos.

### 3.6 Apartado 6

Inser Sort Inv es un algoritmo que lo que hace es ir comparando uno por uno los elementos de la tabla con los anteriores. Coge un número y lo compara con el de su izquierda, si mi número es mayor pongo el valor del que es menor en mi elemento del array y sigo comparando. Cuando encuentre alguno que sea menor, o llegue al final de la tabla paro y meto su valor en dicha posición.  De esta forma conforme se va ejecutando el algoritmo los primeros elementos de la tabla siempre están ordenandos de mayor a menor.

## 4. Código fuente

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

### 4.1 Apartado 1

```
int aleat_num(int inf, int sup)
{
  return (rand()/(RAND_MAX+1.))*(sup-inf+1)+inf;
}
```

### 4.2 Apartado 2

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

int* genera_perm(int N)
{
  int* p=NULL;
  int i;
  p=(int*)malloc(N*sizeof(int));
  if (p==NULL)
    return NULL;
  for (i=0; i<N; i++) {
    p[i]=i+1;
  }
  for (i=0; i<N; i++) {
    swap(&(p[i]), &(p[aleat_num(i, N-1)]));
  }
  return p;
}
```

### 4.3 Apartado 3

```
int** genera_permutaciones(int n_perms, int N)
{
  int** perms=NULL;
  int i;
```

```
  int j;
  perms=malloc(n_perms*sizeof(int*));
  if (perms==NULL)
    return NULL;
  for (i=0; i<n_perms; i++) {
    perms[i]=genera_perm(N);
    if (perms[i]==NULL) {
      for (j=0; j<i; j++) {
        free(perms[j]);
        perms[j]=NULL;
        break;
      }
    return NULL;
    }
  }
  return perms;
}
```

### 4.4 Apartado 4

```
int InsertSort(int* tabla, int ip, int iu)
{
  int i, j, pos, OB=0;
  if (tabla==NULL || ip<0 || iu<ip)
    return ERR;
  for (i=ip; i<=iu; i++) {
    pos=tabla[i];
    j=i-1;
    while (j>=ip && ++OB && tabla[j]>=pos) {
      tabla[j+1]=tabla[j];
      j--;
    }
    tabla[j+1]=pos;
  }
  return OB;
}
```

### 4.5 Apartado 5

```
short tiempo_medio_ordenacion(pfunc_ordena metodo, int
n_perms, int N, PTIEMPO ptiempo) {
  int** tablas=NULL;
  int OB=0, OBTOTAL=0, j=0;
```

```
      double t1, t2;
      if (metodo==NULL || n_perms<1 || N<1 || ptiempo==NULL)
        return ERR;
      ptiempo->N=N;
      ptiempo->n_elems=n_perms;
      tablas=genera_permutaciones(n_perms, N);
      t1=clock();
      for (j=0; j<n_perms; j++) {
        OB=metodo(tablas[j], 0, N-1);
        if (j==0) {
          ptiempo->min_ob=OB;
          ptiempo->max_ob=OB;
        }
        if (OB<ptiempo->min_ob)
          ptiempo->min_ob=OB;
        else if (OB>ptiempo->max_ob)
          ptiempo->max_ob=OB;
        OBTOTAL+=OB;
      }
      t2=clock();
      ptiempo->tiempo=((t2-t1)/n_perms);
      ptiempo->medio_ob=((double)OBTOTAL)/n_perms;
      for (j=0; j<n_perms; j++) {
        free(tablas[j]);
      }
      free(tablas);
      return OK;
    }
    short genera_tiempos_ordenacion(pfunc_ordena metodo, char*
fichero, int num_min, int num_max, int incr, int n_perms) {
      PTIEMPO tiempos=NULL;
      int j, counter=0;
      if (metodo==NULL || fichero==NULL || num_min<0 ||
num_max<num_min || incr<1)
        return ERR;
      tiempos=malloc((((num_max-num_min)/incr)
+1)*sizeof(TIEMPO));
      for(j=num_min; j<=num_max; j+=incr, counter++) {
        if (tiempo_medio_ordenacion(metodo, n_perms, j,
&tiempos[counter])==ERR) {
          free(tiempos);
          return ERR;
        }
```

```
    }
    guarda_tabla_tiempos(fichero, tiempos, counter);
    free(tiempos);
    return OK;
  }
  short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo,
int n_tiempos)
  {
    FILE *f=NULL;
    int i;
    if (fichero==NULL || tiempo==NULL || n_tiempos<1)
      return ERR;

    f=fopen(fichero, "w");
    for (i=0; i<n_tiempos; i++) {
      fprintf(f, "%d %f %f %d %d", tiempo[i].N,
tiempo[i].tiempo, tiempo[i].medio_ob, tiempo[i].min_ob,
tiempo[i].max_ob);
      fprintf(f, "\n");
    }
    fclose(f);
    return OK;
  }
```

## 4.6 Apartado 6

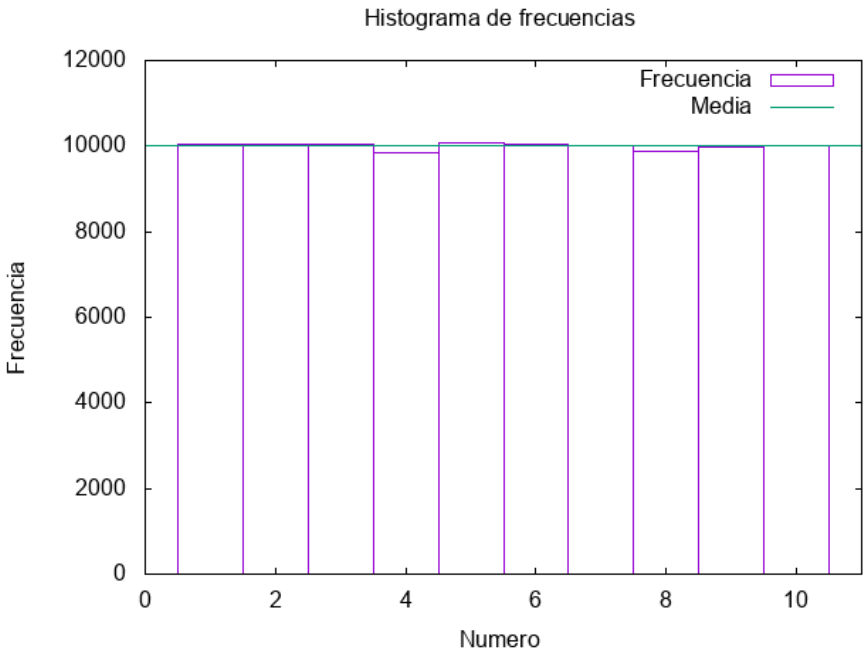
```
  int InsertSortInv(int* tabla, int ip, int iu)
  {
    int i, j, pos, OB=0;
    if (tabla==NULL || ip<0 || iu<ip)
      return ERR;
    for (i=ip; i<=iu; i++) {
      pos=tabla[i];
      j=i-1;
      while (j>=ip && ++OB && tabla[j]<=pos) {
        tabla[j+1]=tabla[j];
        j--;
      }
      tabla[j+1]=pos;
    }
    return OB;
  }
```

## 5. Resultados, Gráficas

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

5.1 Apartado 1

Generamos 100000 números aleatorios entre el 1 y el 10, de manera que si son equiprobables cada uno de ellos debería aparecer una media de 10000 veces. Realizamos el histograma con los números y su frecuencia, que queda de la siguiente manera:




En el histograma, la frecuencia de los números aparecen representados con cajas, y una línea con la media establecida en 1000. Se puede apreciar que cada uno de los números tiene una frecuencia alrededor de esta media, por lo que el código implementado genera números aleatorios con una frecuencia equiprobable.

5.2 Apartado 2

En el apartado 2 generamos una tabla de N elementos con n permutaciones, para una tabla del 1 al 5 y realizando 10 permutaciones, el resultado es el siguiente:

3 4 1 2 5
5 4 1 2 3
1 3 4 5 2
4 3 2 5 1
2 3 4 1 5
3 2 1 4 5
2 1 4 5 3
4 1 5 2 3

5 4 3 2 1
4 5 2 3 1

Como se puede comprobar al ejecutar el apartado 2, nos devuelve una 10 permutaciones de una tabla de 5 elementos, estos elementos están ordenados de matera aleatoria entre sí y no se repite ningún número por lo que el código implementado es correcto.

5.3 Apartado 3

En el apartado 3 se generan n permutaciones equiprobables de N elemetos, al ejecutar el apartado 3 con 10 permutaciones de una tabla de 5 elementos, el resultado es el siguiente:

1 3 4 2 5
4 3 5 1 2
4 5 1 3 2
4 1 5 3 2
2 1 4 3 5
1 5 3 2 4
2 4 3 5 1
3 1 4 2 5
1 2 5 3 4
1 2 5 4 3

El resultado ha generado 10 permutaciones que son equiprobables para una tabla de 5 elementos, a pesar de que podría haber dos repetidas no hay pues el número de permutaciones para una tabla de 5 elementos es n=5!, que es mucho mayor que las 10 permutaciones que generamos, por lo que la probabilidad de que aparezcan repetidas 2 permutaciones es muy baja.

5.4 Apartado 4

En el ejercicio 4 pasamos una tabla de N elementos que quedan desordenados (a través de los códigos de los ejercicios anteriores) y lo pasamos con InserSort, de tal manera que queden ordenadas una vez ejecutado el código. El resultado para una tabla de 100 elementos es el siguiente:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

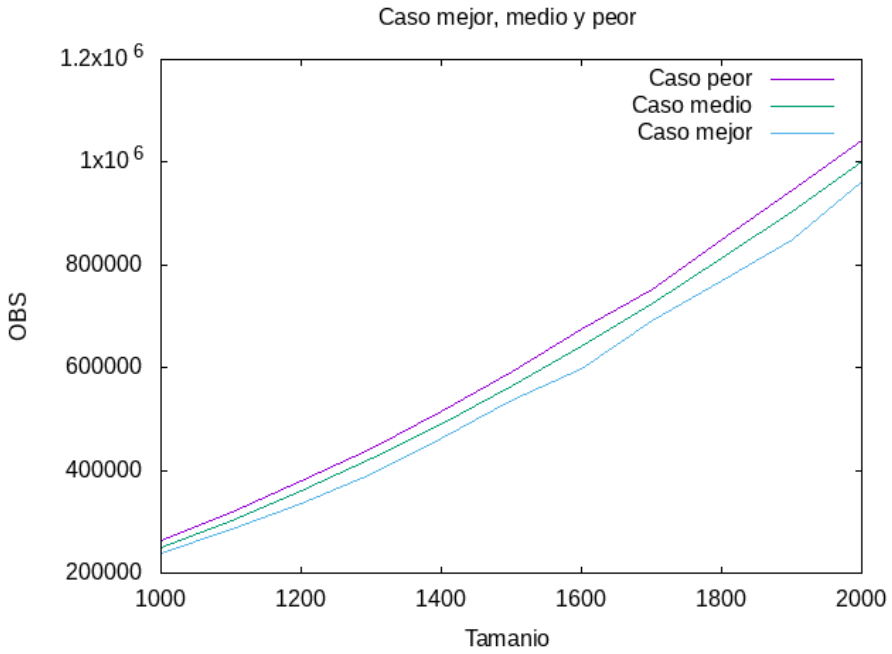
El resultado es una tabla de números ordenados del 1 al 100, lo que era de esperar pues la tabla queda ordenada tras haber ejecutado el código implementado en InsertSort.

## 5.5 Apartado 5

Para el ejercicio 5 realizamos una tabla que contiene los siguientes columnas de datos respectivamente:

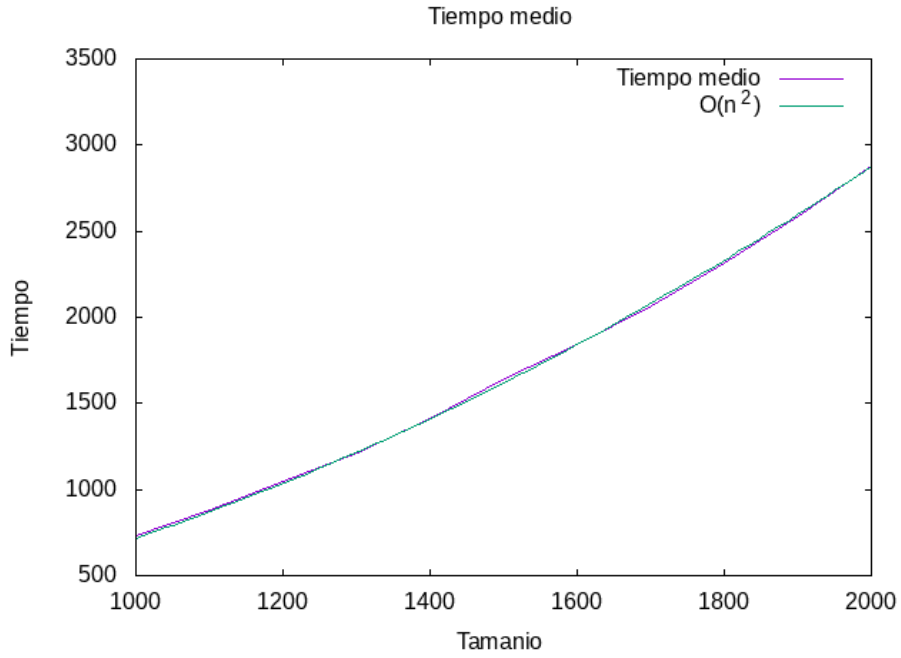
- Columna con tablas que contienen de 1000 a 2000 elementos con incrementos de 100 elementos entre tabla y tabla.
- Columna con los tiempos medios de ejecución de cada una de las tablas.
- Columna con las medias de las operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las mínimas operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las máxomas operaciones básicas (OBs) que se ejecutan para cada tabla.

Usando gnuplot, realizamos una gŕafica comparando los tiempos mejor, peor y medio en OBs para InsertSort, quedando de la siguiente manera:




Se puede apreciar cómo están diferenciados los casos entre sí y como no se llegan a tocar (como es evidente) en ninguno de los casos. También se aprecia que según va aumentando el tamaño de la tabla los casos peor, medio y mejor van aumentando ligeramente entre sí.

Realizamos una gŕafica con el tiempo medio de reloj para InsertSort:

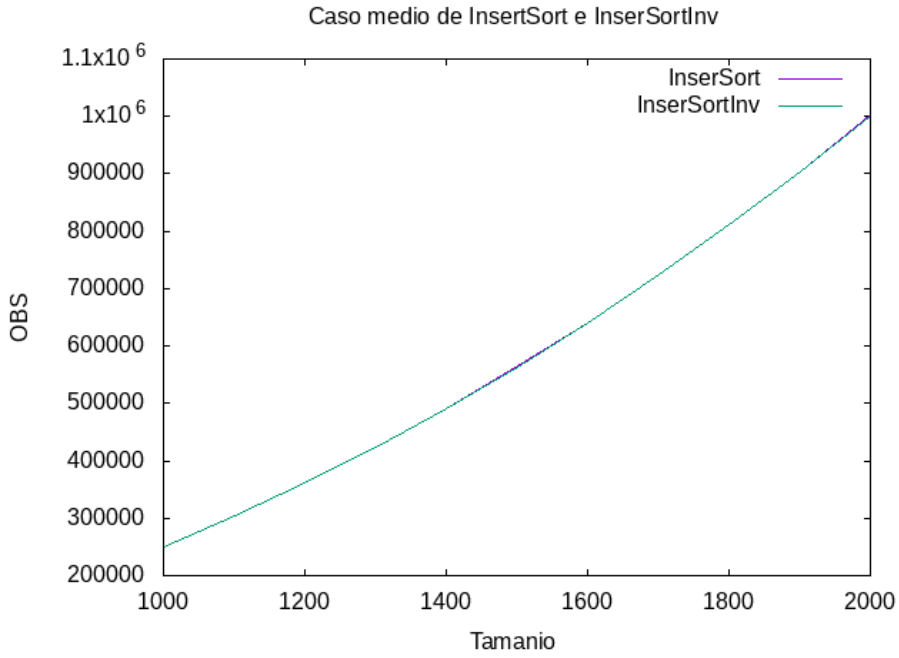



El tiempo medio de InsertSort describe una forma exponencial con $x^2$, por lo que el tiempo medio es $O(n^2)$. Podemos apreciar que ambas gráficas son practicamente equivalentes.

## 5.6 Apartado 6

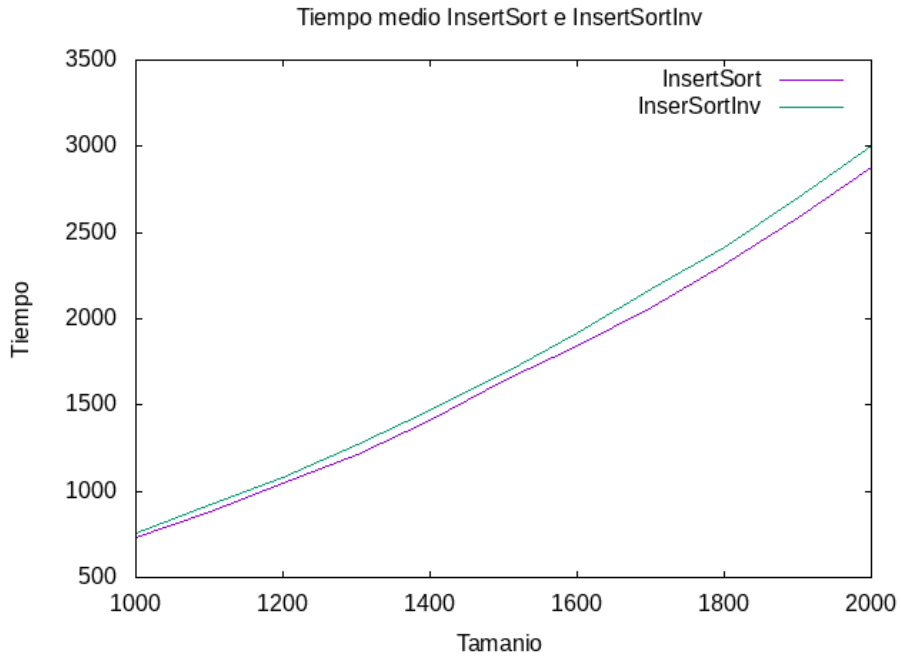
En el ejercicio 6 realizamos lo mismo que en ejercicio 5 pero esta vez con InsertSorInv, que ordena la tabla de mayor a menor, al ejecutar el apartado 6 nos queda una tabla con las columnas antes mencionadas (tamaño, tiempo medio, caso medio, caso mejor, caso peor).

Realizamos una gráfica comparativa del tiempo medio de OBs para InsertSort y InsertSortInv:



Comparando el caso medio en InsertSort e InsertSortInv podemos comprobar que ambas gráficamente son prácticamente la misma, es decir, que el numero de operaciones básicas que ejecuta InsertSort e InsertSortInv es prácticamente el mismo.

Realizamos otra gráfica comparativa del tiempo medio de reloj para InsertSort y InsertSortInv:




Aquí podemos comprobar que los tiempos medios de InsertSort e InsertSortInv son muy parecidos, sin embargo, el tiempo de ejecución de InsertSortInv siempre es algo mayor y se va haciendo mayor a medida que aumenta el número de elementos. Por lo que el algoritmo tarda más en ordenar una tabla de mayor a menor y que al revés.

## 5. Respuesta a las preguntas teóricas.

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

5.1 Pregunta 1

La implementación de aleat_num es la siguiente:

```
int aleat_num(int inf, int sup)
{
  return (rand()/(RAND_MAX+1.))*(sup-inf+1)+inf;
}
```

`rand()` genera un número aleatorio entre 1 y `RAND_MAX`, `RAND_MAX` es un número que depende del compilador, así que dividimos `rand()/(RAND_MAX+1.)`, de tal manera que se genere un número aleatorio entre 0 y 1 sin llegar a 1 de los dos. Al multiplicarlo por `(sup-inf+1)+inf` lo que estamos haciendo es generar un número aleatorio equiprobable entre `inf` y `sup`. Con los recursos que se tienen no hay mejor implementación para obtener números aleatorios equiprobables entre dos dados.

5.2 Pregunta 2

Inser Sort es un algoritmo que lo que hace es ir comparando uno por uno los elementos de la tabla con los anteriores. Primero coge el número que se encuentra en la segunda posición y lo compara con el primero, si mi número es menor pongo el valor del numero con el que lo he comparado en el elemento correspondiente del array. En este caso como es el segundo elemento solo puedo contrastarlo con el primero así que ahora paso a mirar al tercer elemento y así sucesivamente. Comparo cada uno de ellos con los de su izquierda,copiando el valor del número con el que lo he comparado en la posición siguiente del array. Sigo contrastando valores hasta que encuentre alguno que sea mayor, o llegue al final de la tabla. Paro y meto el valor de mi elemento en dicha posición. De esta forma conforme se va ejecutando el algoritmo los primeros elementos de la tabla siempre están ordenandos de menor a mayor.

5.3 Pregunta 3

El bucle exterior no actúa sobre el primer elemento por el funcionamiento interno de InsertSort, una vez en ejecución, dado un elemento i de la tabla, todos los elementos a la izquierda de i están ordenados con respecto a ese número. Como el primer elemento no tiene ningún elemento a su izquierda, ya está “ordenado”.

5.4 Pregunta 4

La operación básica de InsertSort es la comparación de claves, es decir `tabla[j]>=pos`, por ello, en el bucle while sumamos una OBs justo antes de ejecutar esta comparación, quedando de la siguiente manera: `while (j>=ip && ++OB && tabla[j]>=pos)`

5.5 Pregunta 5

Podemos ver que tanto $W_{IS}(n)$ como $B_{IS}(n)$ son $O(N^2)$, esto concuerda con lo estudiado en teoría pues el caso peor de InserSort es $W_{IS}(n)=N^2/2+O(N)$ y el caso medio de insert sort es $A_{IS}(n)=N^2/4+O(N)$, siendo el caso mejor menor que el caso medio y estos dos a su vez menores que el caso peor, pero siendo todos $O(N^2)$.

5.5 Pregunta 6

Como hemos indicado en el apartado 6 del ejercicio anterior, el tiempo medio de ejecución de InsertSortInv siempre es algo mayor que el de InsertSort, aunque el número de OBs sea el mismo para ambos el algoritmo implementado tarda más en ordenar una tabla de mayor a menor que de menor a mayor, y aunque el tiempo medio

sea muy parecido en ambos casos, este se va haciendo mayor a medida que aumenta el número de elementos de la tabla.

## 7. Conclusiones finales.

Al realizar esta práctica hemos aprendido como generar números aleatorios equiprobables descubriendo el comando rand. A partir de estos hemos conseguid generar permutaciones y tablas de estas.

Además hemos implementado Insert Sort lo cual nos ha ayudado para acabar de entender el funcionamiento de esta y ver cuantas veces se ejecuta su OB. Insert Sort Inv una vez crea la anterior fue muy sencillo cambiar determinadas lineas del código para que la ordenación fuera del revés.

Finalmente en el último ejercicio de la práctica vimos como funcionaban las funciones que calculaban los tiempos medios de ejecución. Al principio no sabíamos muy bien como calcular el max_ob y el min_ob pero tras varias pruebas al final conseguimos que nos funcionara.

Puede que la parte más novedosa haya sido generar las gráficas a partir de estas funciones pues nunca habíamos usado GNU plot pero tras varios vídeos explicativos e ir cacharreando al final te sientes un experto.

Cabe concluir que estos ejercicios prácticos siempre son un apoyo más para la parte teórica de la asignatura, pues ahora hemos llegado a entender algunos conceptos teóricos que aplicados a nuestras funciones son más visuales y comprensibles como por ejemplo el calculo del caso mejor y peor de un algoritmo.