

# Análisis de Algoritmos 2019/2020

## Práctica 1

César Ramírez - Martín Sánchez, Grupo 1201

Código	Gráficas	Memoria	Total

## **1. Introducción.**

La práctica a realizar constará de tres partes principales divididas en la realización de archivos escritos en C, usados para: generar permutaciones y números aleatorios, ordenar listas de enteros, y medir y guardar tiempos reales y abstractos de ejecución. Así pues, con lo obtenido al finalizar la práctica, podremos estudiar cómo se comporta un algoritmo de ordenación en la realidad.

Lo que se pretenderá será poner a prueba InsertSort y analizar su comportamiento.

Lo primero que se abordará será la creación de un mecanismo para poder generar las listas que se utilizarán como entrada de InsertSort, así pues, necesitaremos un mecanismo de generación de permutaciones, y, por lo tanto, tendremos que conseguir números aleatorios para evitar estudiar todas las posibles permutaciones, y en vez solo unas pocas.

Lo segundo será escribir el algoritmo en cuestión, que en este caso será InsertSort.

Por último, analizaremos cómo se relaciona la teoría con la práctica, poniendo a prueba nuestro algoritmo. Tendremos que comprobar que, en efecto lo estudiado previamente corresponde con la realidad.

## **2. Objetivos**

### **2.1 Apartado 1**

El objetivo de este apartado es, simplemente, crear una función en el fichero `permutaciones.c` que consiga generar números aleatorios.

Para ello desarrollamos la función que hacía justo eso, generar números aleatorios entre unos valores mínimo y máximo dados.

## 2.2 Apartado 2

El objetivo de este apartado es generar una función que genere una permutación totalmente aleatoria de N elementos. Para ello hará uso de la función creada en el apartado anterior. Esta función también ha sido implementada en el fichero `permutaciones.c`.

## 2.3 Apartado 3

El objetivo de este apartado es generar un número de permutaciones aleatorias equiprobables. Es decir, haciendo uso de las funciones ya implementadas en los anteriores apartados, se pide realizar una función que, dados dos enteros (`n_perms` y `N`), genere el número de permutaciones dado (`n_perms`), con un número concreto de elementos (`N`) por permutación. Es muy importante haber realizado correctamente las funciones de los apartados anteriores, ya que es vital que los números generados sean totalmente aleatorios. Es decir, que todos (entre el máximo y mínimo dados) tengan exactamente las mismas probabilidades de aparecer. La tarea de este apartado será también realizada en el fichero `permutaciones.c`.

## 2.4 Apartado 4

El objetivo de este apartado es diseñar la función `InsertSort` para dos cosas:

1. La primera y más evidente, para ordenar los elementos de las tablas/permutaciones aleatorias previamente creadas.
2. Y la segunda, para contar el número de OBs realizadas por el algoritmo para cada tabla.

Se ha escogido el algoritmo de ordenación `InsertSort` (y no otro) ya que, hasta ahora, de los vistos en clase, es el que mejor rendimiento tiene. La función `InsertSort` se realizará en el fichero `ordenación.c`.

## 2.5 Apartado 5

El obtivo del apartado 5 es diseñar una serie de estructuras y funciones que nos ayuden a calcular los tiempos de ejecución del algoritmo `InsertSort` para cada tabla aleatoriamente generada. Para ello se desarrolla una estructura (`tiempo`) que nos ayuda con el almacenaje de los tiempos de ejecución del algoritmo.

A continuación se crea una función cuya tarea es calcular el tiempo medio que ha tardado el algoritmo `InsertSort` en ordenar un número determinado de permutaciones (`n_perms`) de N elementos cada una.

Después necesitaremos una nueva función que recopile la información obtenida al pasar dichas tablas a la función anterior. Para ello creamos la función `genera_tiempos_ordenacion`, que llamando a la función `guarda_tabla_tiempos` (creada

también en este apartado), será capaz de imprimir toda la información obtenida anteriormente (número máximo, medio y mínimo de OB que realiza el algoritmo por tabla, y tiempos medios que tarda en ordenarlas).

Finalmente se imprimirá en el fichero solicitado una tabla con 5 columnas con toda la información requerida. Para obtener los tiempos haremos uso de funciones como clock, y algún mecanismo más para medir los tiempos de manera precisa. Todo lo realizado en este apartado ha sido desarrollado en el fichero tiempo.c.

## 2.6 Apartado 6

El objetivo de este apartado, era el de implementar el algoritmo InsertSort pero de manera inversa. Es decir, que ordenara los elementos de una tabla en orden inverso. Para ello modificamos ligeramente el algoritmo InsertSort previamente desarrollado.

Finalmente comparamos los resultados obtenidos con ambos algoritmos gracias a las gráficas. Este algoritmo también fue implementado en el fichero ordenacion.c.

## 3. Herramientas y metodología

Toda la práctica fue realizada en el sistema operativo Ubuntu de los ordenadores del laboratorio.

Como entorno de desarrollo del código hemos empleado el editor de texto ‘Atom’. Hemos compilado y enlazado mediante el fichero makefile facilitado usando la terminal, y Valgrind fue usado para chequear el correcto manejo de memoria.

### 3.1 Apartado 1

Para la generación de números aleatorios se optó por un algoritmo visto en clase, que es el siguiente:

```
(rand() / (RAND_MAX + 1.) * ((sup - inf) + 1) + inf)
```

En la sección de problemas, apartado 1, se entra en detalle sobre las opciones barajadas y por qué esta fue nuestra decisión final.

### 3.2 Apartado 2

Para generar una permutación lo primero fue crear una lista con los números del 0 al N-1, y luego, por cada elemento i, intercambiarlo por un elemento entre su índice y el más alto; obteniéndose mediante la función de generar un número aleatorio.

### 3.3 Apartado 3

Para el tercer ejercicio realizamos una función que reservaba memoria para N permutaciones, cada una de las cuales se creaba en la función 'genera\_perm'.

### 3.4 Apartado 4

Para el apartado 4 implementamos el algoritmo de ordenación InsertSort en C, el cual consiste en ir insertando elementos dentro de la parte ya ordenada de la tabla. A partir de su pseudocódigo visto en clase, pudimos traducirlo a C teniendo en cuenta que la función debía devolver el número de operaciones básicas realizadas (CDC).

### 3.5 Apartado 5

El apartado 5 fue el más extenso y se realizó secuencialmente, primero escribiendo la función 'tiempo\_medio\_ordenación', la cual, mediante el uso de la función clock() pudimos medir el tiempo medio al dividir entre el número de permutaciones. Además nos servimos de la biblioteca limits.h para poner el valor mínimo al entero máximo posible, aunque esto puede también resolverse realizando la primera iteración del bucle fuera; lo cual nos pareció antiestético.

Para las siguientes funciones realizamos una tabla de tiempos que se iba rellenando mediante llamadas a 'tiempo\_medio\_ordenacion'. Finalmente, para guardar los resultados llamamos a la función 'guarda\_tabla\_tiempos', que abre el fichero e imprime los datos de cada tiempo. Cabe destacar que para conocer el número de iteraciones (el tamaño total de la tabla creada en 'genera\_tiempos\_ordenacion') la obtuvimos dividiendo la diferencia entre los tamaños de las permutaciones por el incremento y sumando uno, basándonos en la división entera.

### 3.6 Apartado 6

Para que la tabla quedase ordenada de mayor a menor en vez de al revés, lo único que tuvimos que hacer fue redefinir la condición de la comparación de cable. En vez de intercambiar elementos si el siguiente es menor que el anterior, lo hicimos si el siguiente es mayor, por lo cambio fue mínimo.

## 4. Código fuente

### 4.1 Apartado 1

```
int aleat_num(int inf, int sup)
{
    if(sup < inf) return sup;

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

### 4.2 Apartado 2

```
int* genera_perm(int N)
{
    if(N <= 0) return NULL;

    int *perm = NULL;

    int i, a;

    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++) {
        a = aleat_num(i, N-1);

        SWAP(perm[i], perm[a]);
    }
}
```

```
}
```

```
return perm;
```

```
}
```

#### 4.3 Apartado 3

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

```
{
```

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

```
int** perms = NULL;
```

```
int i;
```

```
perms = (int**)malloc(n_perms*sizeof(int*));
```

```
if(perms == NULL) return NULL;
```

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

```
    perms[i] = genera_perm(N);
```

```
    /* Gestion de Errores*/
```

```
    if(perms[i] == NULL) {
```

```
        for(i--; i >= 0; i--) {
```

```
            free(perms[i]);
```

```
        }
```

```
        free(perms);
```

```
        return NULL;
```

```
    }
```

```
}
```

```
return perms;
```

```
}
```

#### 4.4 Apartado 4

```
int InsertSort(int* tabla, int ip, int iu)
```

```
{
```

```
    if(tabla == NULL || ip < 0 || iu < ip) return ERR;
```

```
    int obs = 0;
```

```
    /*Numero de elementos ordenados/ proximo indice a ordenar */
```

```
    int j = ip+1;
```

```
    int i, swap;
```

```
    while(j <= iu) {
```

```
        /*i: ultimo indice ordenado*/
```

```
        i = j-1;
```

```
        while((i >= ip) && (++obs) && (tabla[i] > tabla[i+1])) {
```

```
            swap = tabla[i+1];
```

```
            tabla[i+1] = tabla[i];
```

```
            tabla[i] = swap;
```

```
            i--;
```

```
        }
```

```
        j++;
```

```
}
```



```

return obs;

}

```

#### 4.5 Apartado 5

```

short tiempo_medio_ordenacion(pfunc_ordena metodo,

                                int n_perms,

                                int N,

                                PTIEMPO ptiempo)

{

    if(metodo == NULL || ptiempo == NULL || n_perms <= 0 || N <= 0) return ERR;


    int **tablas = NULL;

    clock_t t1;

    clock_t t2;

    int j = 0;

    int obs_total = 0;

    int obs_min = INT_MAX;

    int obs_max = -1;

    int obs = 0;


    ptiempo->N = N;

    ptiempo->n_elems = n_perms;


    /* Generamos las n_perms tablas de tamaño N a ordenar*/

    tablas = genera_permutaciones(n_perms, N);

```

```

if(tablas == NULL) return ERR;

/* Calculamos el tiempo necesario para ordenar las tablas como la diferencia */
/* entre t1 y t2 */

t1 = clock();

for(j=0; j<n_perms; j++) {
    obs = metodo(tablas[j], 0, N-1);
    if(obs < obs_min) {
        obs_min = obs;
    }
    else if(obs > obs_max) {
        obs_max = obs;
    }
    obs_total += obs;
}

t2 = clock();

ptiempo->tiempo = (double)(t2 - t1)/n_perms;
ptiempo->medio_ob = ((double)obs_total)/n_perms;
ptiempo->min_ob = obs_min;
ptiempo->max_ob = obs_max;

```

```

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)

{

    if(metodo == NULL || fichero == NULL || num_min < 1 || num_max < num_min ||

        incr < 1 || n_perms < 1) return ERR;


    /* Vamos a ir calculando los tiempos que se tarda en ordenar n_perms */

    /* permutaciones de tablas de N elementos, yendo N desde num_min hasta */

    /* num_max de incr en incr. Luego necesitamos un array cuyo tamaño */

    /* sera (num_max - num_min)/incr + 1 */

    int size = (num_max - num_min)/incr + 1; /* Numero de tiempos a medir*/

    int j = 0;

    int N = num_min;

    PTIEMPO tiempos = NULL;

    tiempos = (PTIEMPO)malloc(size*sizeof(TIEMPO));

```

```
if (tiempos == NULL) return ERR;
```

```
for(j=0; j<size; j++) {
```

```
    /* Guardamos en tiempos[j] el tiempo medio para ordenar n_perms de tam N */
```

```
    if(tiempo_medio_ordenacion(metodo, n_perms, N, &tiempos[j]) == ERR) {
```

```
        free(tiempos);
```

```
        return ERR;
```

```
    }
```

```
    N+=incr;
```

```
}
```

```
if(guarda_tabla_tiempos(fichero, tiempos, size)==ERR) {
```

```
    free(tiempos);
```

```
    return ERR;
```

```
}
```

```
free(tiempos);
```

```
return OK;
```

```
}
```

```
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
```

```
{
```

```
    if(fichero==NULL || tiempo == NULL || n_tiempos < 1) return ERR;
```

```
    int i = 0;
```

```
    FILE* pf = NULL;
```

```
TIEMPO t;
```

```
pf = fopen(fichero, "w");
```

```
if(pf == NULL) return ERR;
```

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

```
    t = tiempo[i];
```

```
    if(fprintf(pf, "%d %lf %lf %d %d\n", t.N, t.tiempo, t.medio_ob, t.min_ob, t.max_ob)<0) {
```

```
        fclose(pf);
```

```
        return ERR;
```

```
    }
```

```
}
```

```
fclose(pf);
```

```
return OK;
```

```
}
```

#### 4.6 Apartado 6

```
int InsertSortInv(int* tabla, int ip, int iu)
```

```
{
```

```
    if(tabla == NULL || ip < 0 || iu < ip) return ERR;
```

```
    int obs = 0;
```

```
    /*Numero de elementos ordenados/ proximo indice a ordenar */
```

```
    int j = ip+1;
```

```
int i, swap;
```

```
while(j <= iu) {
```

```
    /*i: ultimo indice ordenado*/
```

```
    i = j-1;
```

```
    while((i >= ip) && (++obs) && (tabla[i] < tabla[i+1])) {
```

```
        swap = tabla[i+1];
```

```
        tabla[i+1] = tabla[i];
```

```
        tabla[i] = swap;
```

```
        i--;
```

```
    }
```

```
    j++;
```

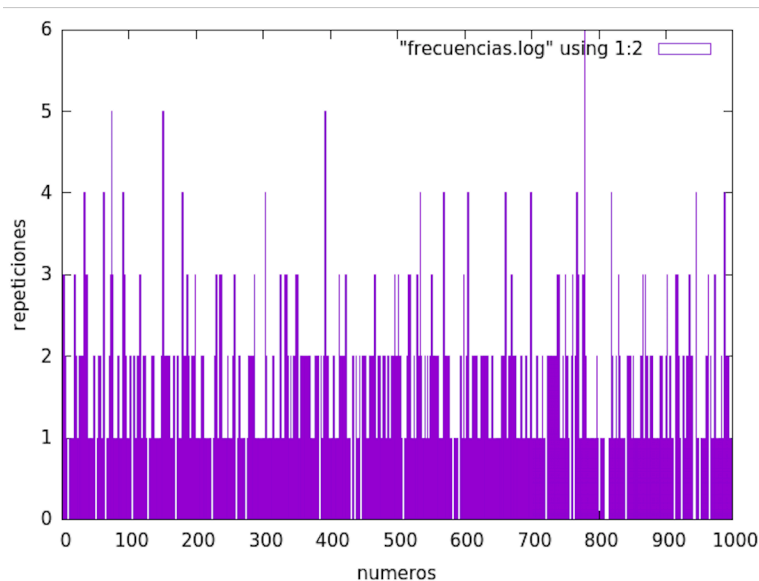
```
}
```

```
return obs;
```

```
}
```

## 5. Resultados, Gráficas

### 5.1 Apartado 1



En esta gráfica de histogramas, se puede apreciar la frecuencia con la que sale cada número entre 0 y 1000 para 1000 números generados.

Como se puede ver el número de veces que se repite cada número es aleatorio pero dentro de un orden lógico, y ningún elemento se repite muchas más veces que otro. Así que con esto podemos confirmar el correcto funcionamiento de la función `aleat_num`.

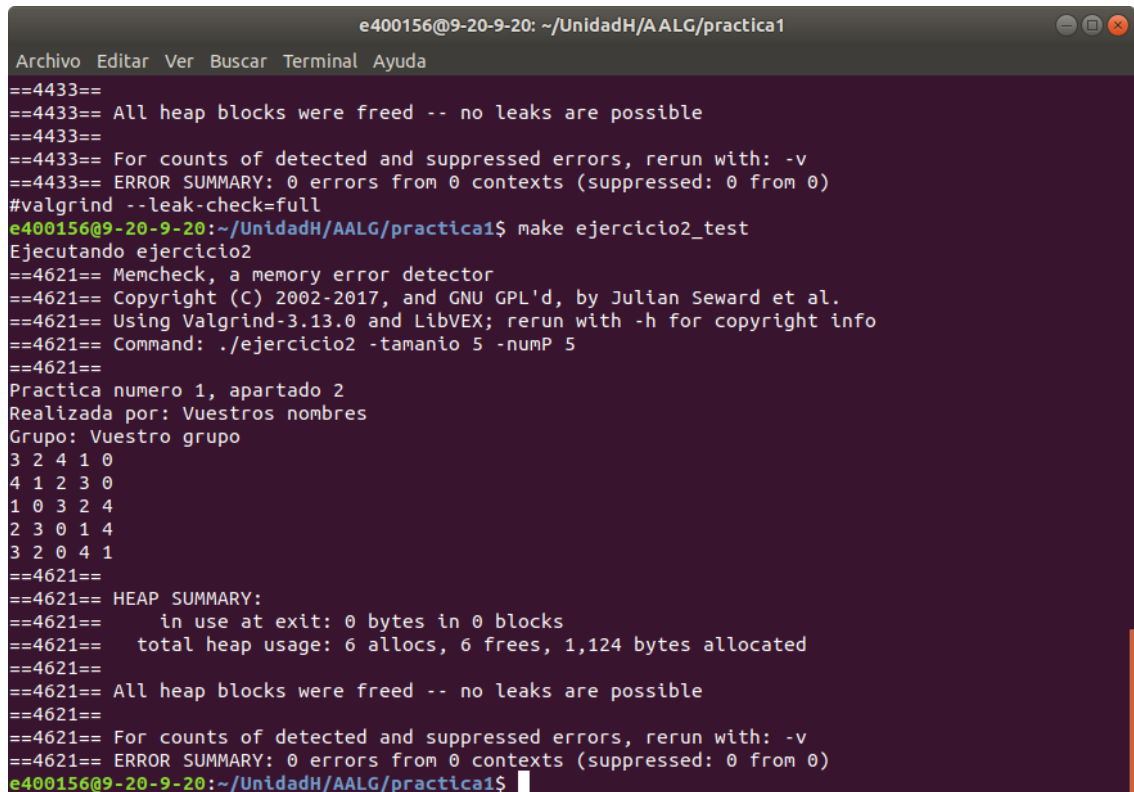
Y la siguiente imagen también es una evidencia de ello.

```
e400156@9-20-9-20: ~/UnidadH/AALG/practica1
Archivo Editar Ver Buscar Terminal Ayuda
e400156@9-20-9-20:~/UnidadH/AALG/practica1$ make ejercicio1_test
Ejecutando ejercicio1
==4433== Memcheck, a memory error detector
==4433== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4433== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4433== Command: ./ejercicio1 -limInf 1 -limSup 5 -numN 10
==4433==
Practica numero 1, apartado 1
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
5
1
5
3
1
1
5
2
2
1
==4433==
==4433== HEAP SUMMARY:
==4433==   in use at exit: 0 bytes in 0 blocks
==4433== total heap usage: 4 allocs, 4 frees, 9,788 bytes allocated
==4433==
==4433== All heap blocks were freed -- no leaks are possible
==4433==
==4433== For counts of detected and suppressed errors, rerun with: -v
==4433== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
#valgrind --leak-check=full
e400156@9-20-9-20:~/UnidadH/AALG/practica1$
```

Se puede apreciar como se han generado 10 números aleatorios entre el 1 el 5.

## 5.2 Apartado 2

Como se puede ver en la imagen adjunta, se han generado 5 tablas de 5 elementos ordenadas aleatoriamente. Para ello hemos tenido que llamar a la función `genera_perm` 5 veces.



```
e400156@9-20-9-20: ~/UnidadH/AALG/practica1
Archivo Editar Ver Buscar Terminal Ayuda
==4433==
==4433== All heap blocks were freed -- no leaks are possible
==4433==
==4433== For counts of detected and suppressed errors, rerun with: -v
==4433== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
#valgrind --leak-check=full
e400156@9-20-9-20:~/UnidadH/AALG/practica1$ make ejercicio2_test
Ejecutando ejercicio2
==4621== Memcheck, a memory error detector
==4621== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4621== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4621== Command: ./ejercicio2 -tamaño 5 -numP 5
==4621==
Practica numero 1, apartado 2
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
3 2 4 1 0
4 1 2 3 0
1 0 3 2 4
2 3 0 1 4
3 2 0 4 1
==4621==
==4621== HEAP SUMMARY:
==4621==    in use at exit: 0 bytes in 0 blocks
==4621== total heap usage: 6 allocs, 6 frees, 1,124 bytes allocated
==4621==
==4621== All heap blocks were freed -- no leaks are possible
==4621==
==4621== For counts of detected and suppressed errors, rerun with: -v
==4621== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
e400156@9-20-9-20:~/UnidadH/AALG/practica1$
```

## 5.3 Apartado 3

En esta imagen se puede observar como la función `genera_permutaciones` (con una sola llamada en la que le hemos señalado el número de permutaciones que queremos) ha generado 5 permutaciones equiprobables de 5 elementos cada una, haciendo uso de la función `genera_perm` del apartado anterior.



```
e400156@9-20-9-20: ~/UnidadH/AALG/practica1
Archivo Editar Ver Buscar Terminal Ayuda
==4621== total heap usage: 6 allocs, 6 frees, 1,124 bytes allocated
==4621==
==4621== All heap blocks were freed -- no leaks are possible
==4621==
==4621== For counts of detected and suppressed errors, rerun with: -v
==4621== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
e400156@9-20-9-20:~/UnidadH/AALG/practica1$ make ejercicio3_test
Ejecutando ejercicio3
==4665== Memcheck, a memory error detector
==4665== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4665== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4665== Command: ./ejercicio3 -tamanio 5 -numP 5
==4665==
Practica numero 1, apartado 3
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
1 4 3 0 2
1 4 3 0 2
2 3 4 1 0
2 4 0 1 3
2 1 3 4 0
==4665==
==4665== HEAP SUMMARY:
==4665== in use at exit: 0 bytes in 0 blocks
==4665== total heap usage: 7 allocs, 7 frees, 1,164 bytes allocated
==4665==
==4665== All heap blocks were freed -- no leaks are possible
==4665==
==4665== For counts of detected and suppressed errors, rerun with: -v
==4665== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
e400156@9-20-9-20:~/UnidadH/AALG/practica1$
```

## 5.4 Apartado 4

Aquí simplemente se puede observar el correcto funcionamiento de InsertSort para una tabla de números del 0 al 9.

```
e400156@9-20-9-20: ~/UnidadH/AALG/practica1
Archivo Editar Ver Buscar Terminal Ayuda
2 1 3 4 0
==4665==
==4665== HEAP SUMMARY:
==4665== in use at exit: 0 bytes in 0 blocks
==4665== total heap usage: 7 allocs, 7 frees, 1,164 bytes allocated
==4665==
==4665== All heap blocks were freed -- no leaks are possible
==4665==
==4665== For counts of detected and suppressed errors, rerun with: -v
==4665== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
e400156@9-20-9-20:~/UnidadH/AALG/practica1$ make ejercicio4_test
Ejecutando ejercicio4
==4715== Memcheck, a memory error detector
==4715== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4715== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4715== Command: ./ejercicio4 -tamanio 10
==4715==
Practica numero 1, apartado 4
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
0      1      2      3      4      5      6      7      8      9
==4715==
==4715== HEAP SUMMARY:
==4715== in use at exit: 0 bytes in 0 blocks
==4715== total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==4715==
==4715== All heap blocks were freed -- no leaks are possible
==4715==
==4715== For counts of detected and suppressed errors, rerun with: -v
==4715== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
e400156@9-20-9-20:~/UnidadH/AALG/practica1$
```

## 5.5 Apartado 5

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

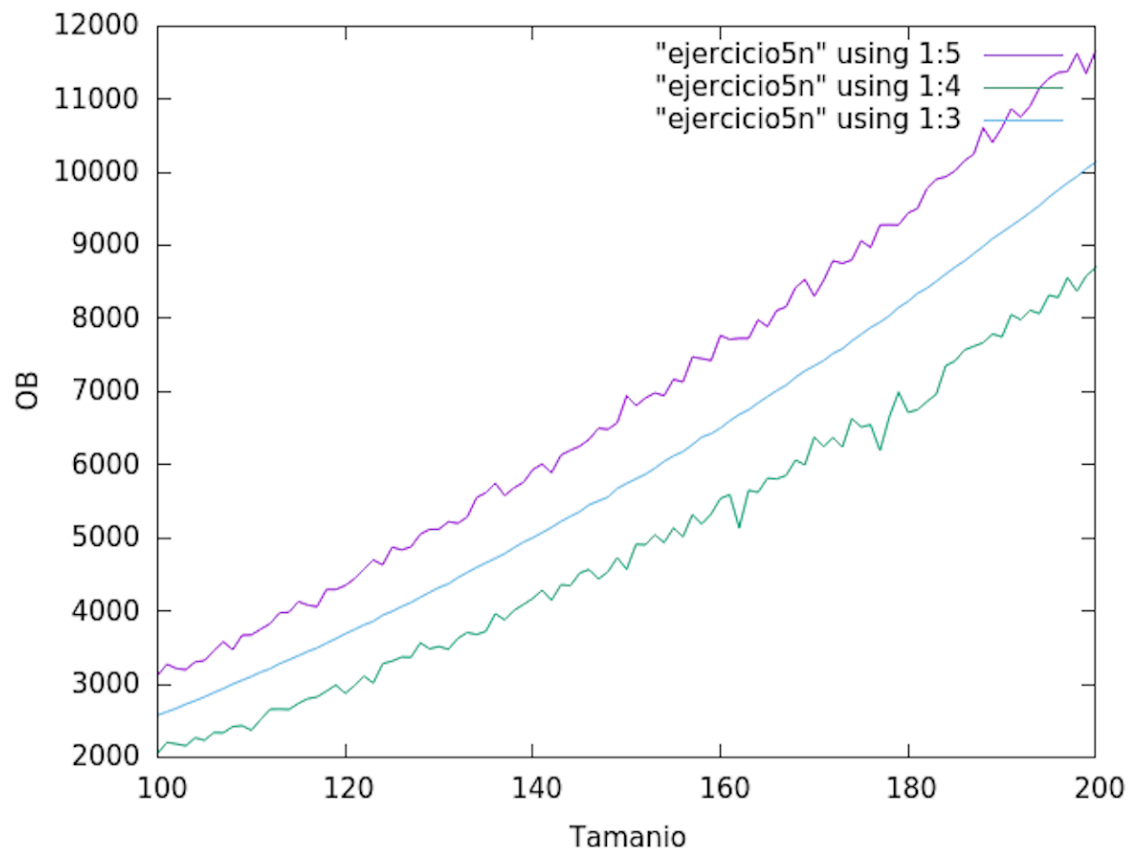
Gráfica con el tiempo medio de reloj para InsertSort, comentarios a la gráfica.

En la primera gráfica apreciamos el número máximo (azul), medio (rosa) y mínimo (verde) de OBs que hace InsertSort para ordenar tablas de cada vez más elementos, hasta llegar a tablas de 100 elementos.

Se aprecia un crecimiento coherente, a mayor número de elementos por tabla, mayor número máximo, mínimo y medio de OBs.

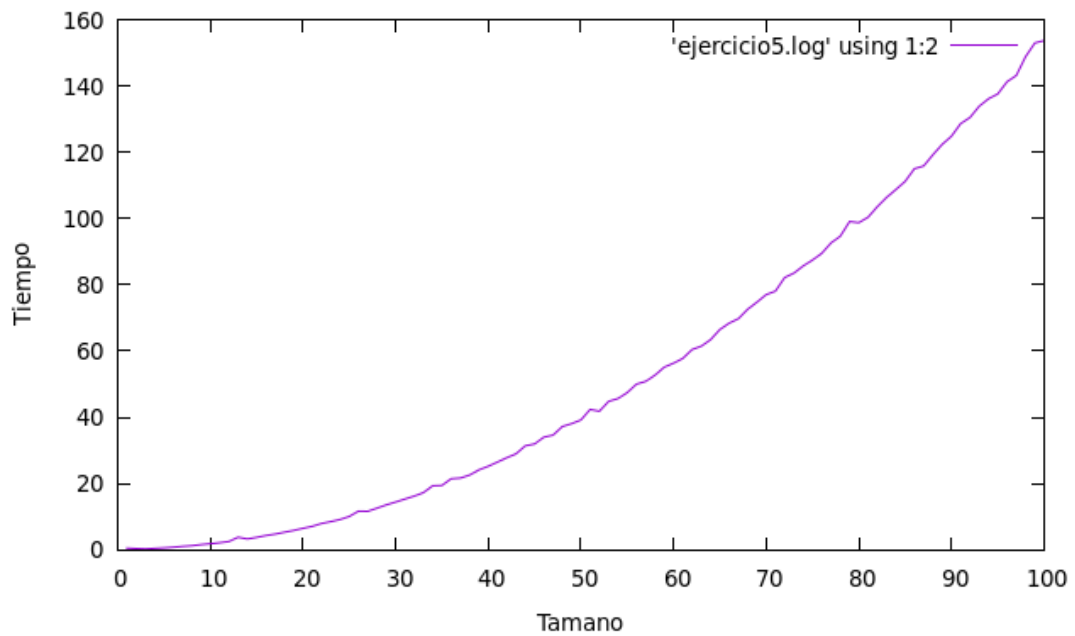
Y es también evidente que a mayor número de elementos por tabla se irá acentuando cada vez más la diferencia del número de OBs mínimo, medio y máximo.

El número medio de OBs siempre se encuentra entre el número máximo y mínimo de OBs, otra evidencia más de la coherencia derivada de la gráfica.



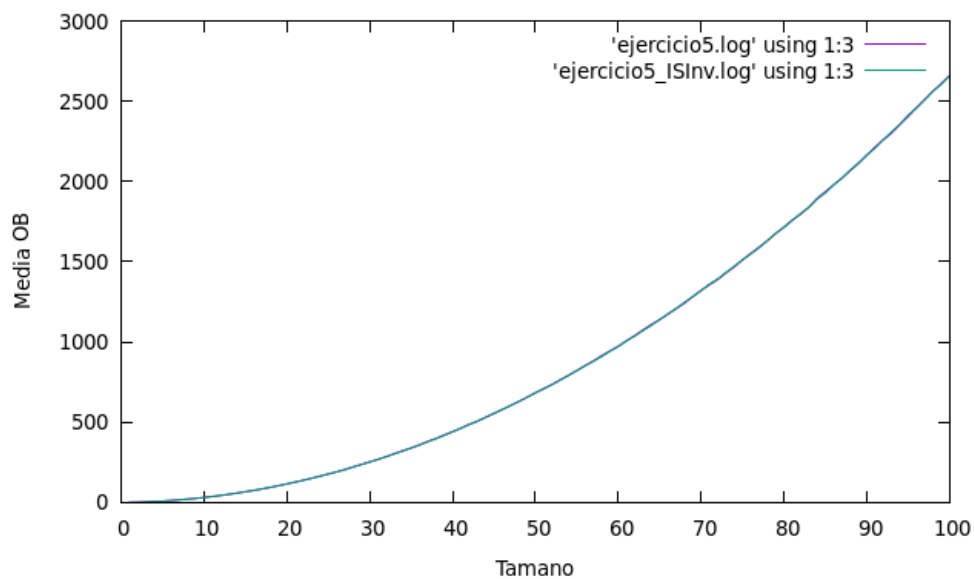
En esta segunda gráfica apreciamos el tiempo medio que tarda InsertSort en ordenar tablas de cada vez más elementos hasta llegar a tablas de 100 elementos.

Se aprecia un crecimiento coherente, a mayor número de elementos por tabla, mayor número de OBs tendrá que realizar InsertSort, luego mayor tiempo medio tardará en ordenarlas.

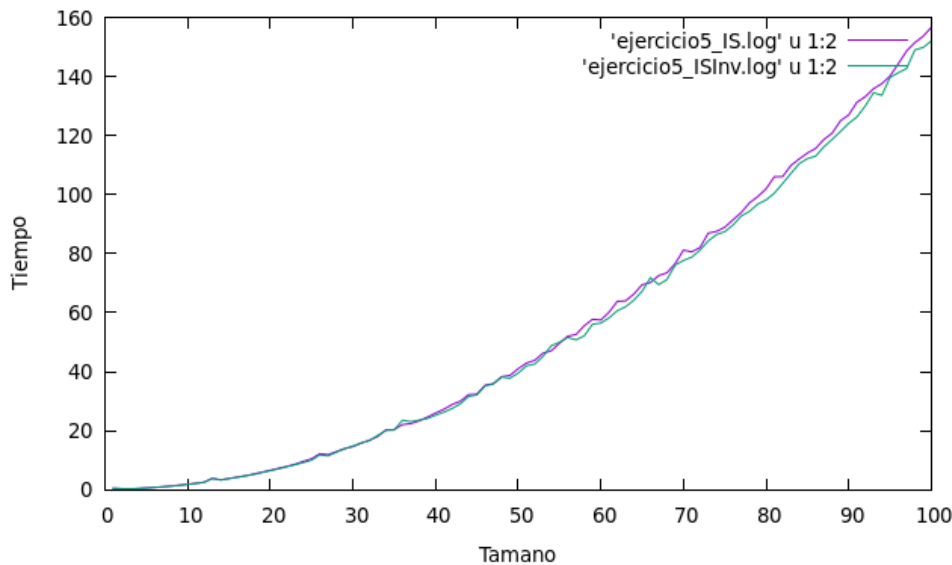


## 5.6 Apartado 6

En la primera gráfica se ve una comparación de las OBs medias que realizan ambos algoritmos para tablas de hasta 100 elementos. Se puede apreciar claramente como prácticamente coincide el número medio de OBs de ambos algoritmos. Esto se debe a que para cada tamaño la mitad de tablas de ese tamaño, son requeridas menos OBs para ordenarlas con un algoritmo, y con la otra mitad de tablas es el otro algoritmo el que requiere de menos OBs para ordenarlas, luego quedan parejas en una gráfica.



En esta última gráfica se ve una comparación del tiempo medio que tardan ambos algoritmos en ordenar tablas de hasta 100 elementos. Se puede apreciar claramente como prácticamente coincide el tiempo medio de ambos algoritmos para todos los tamaños. Esto se debe a que por cada tamaño la mitad de tablas de ese tamaño se ordenarán más rápido con un algoritmo, y la otra mitad con el otro, luego quedan parejas en una gráfica.



## 5. Respuesta a las preguntas teóricas.

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

### 5.1 Pregunta 1

Para la generación de números aleatorios se optó por un algoritmo visto en clase, que es el siguiente:

```
(rand() / (RAND_MAX + 1.) * ((sup - inf) + 1) + inf)
```

El cual garantiza la generación de números pseudoaleatorios equiprobables. Otra alternativa más *naive* sería utilizando el módulo de la siguiente forma:

```
(rand() % (sup-inf+1) + inf)
```

Sin embargo este último método no siempre divide la probabilidad equitativamente, pues si dividimos los posibles resultados de `rand()` en grupos según su congruencia en módulo  $(sup-inf+1)$  la cardinalidad de los conjuntos no tiene por qué ser la misma.

Esto es fácilmente comprobable

### 5.2 Pregunta 2

Para demostrar la corrección de InsertSort, podemos verlo mediante inducción fuerte. Así pues, sea  $N$  el tamaño de la tabla a ordenar, y sea  $K$  la iteración del bucle exterior en el que nos encontramos.

Para  $K = 1$ , en la primera iteración, tenemos a la izquierda un único elemento, por lo que la parte izquierda ya está ordenada. Ahora insertamos el segundo elemento de la lista, tras lo cual ya tenemos ordenados los primeros elementos de la tabla.

Si suponemos que en  $K = M$ , tras  $M$  iteraciones, los  $M$  elementos primeros de la tabla han quedado ordenados, si ahora insertamos el siguiente podemos observar que el orden relativo de los elementos ya ordenados se mantendrá siempre, pues es como si hubiésemos dejado un hueco en la lista y desplazado los elementos.

Así pues, cuando acabamos de insertar el nuevo elemento, no solo se mantiene el orden entre los elementos previos, sino que también hemos introducido uno nuevo que sigue conservando el orden. Pues si la inserción se detiene en un determinado índice, cualquier elemento a la izquierda debería ser menor, pero ya hemos dicho que estos elementos ya están ordenados entre sí, por lo que en efecto, se hallan ordenados a la izquierda.

### 5.3 Pregunta 3

Como cualquier tabla de un solo elemento ya está ordenada, el primer elemento de cualquier tabla ya está en orden relativo consigo mismo. Es decir, como InsertSort va insertando elementos en la parte de la lista ya ordenada relativamente, no necesitamos insertar el primer elemento, pues ya está relativamente ordenado, y podemos saltar al siguiente insertándolo en dicha parte ya ordenada relativamente.

### 5.4 Pregunta 4

InsertSort forma parte del conjunto de algoritmos de ordenación por comparación de claves. Así pues, la operación básica de InsertSort es la comparación de clave, en nuestro caso la sentencia:

```
(tabla[i] > tabla[i+1])
```

### 5.5 Pregunta 5

El caso peor de InsertSort sería la lista invertida, con la cual se obtiene que:

$$W_{IS}(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$$

El caso mejor es la tabla ya ordenada:

$$B_{IS}(n) = \sum_{i=1}^{n-1} 1 = n-1 = O(n)$$

### 5.5 Pregunta 6

A partir de las gráficas observamos que ambas funciones se comportan prácticamente igual, lo cual era de esperar, principalmente por dos motivos.

En primer lugar, para cada algoritmo hemos obtenido permutaciones distintas y aleatorias, con las cuales hemos realizado un estudio del número medio de CDC

realizadas en cada algoritmo. Sin embargo podemos ver que, para una misma permutación, el número de veces que se ejecuta la operación básica es indirectamente proporcional entre ambos algoritmos, pero como para cada algoritmo hemos estudiado numerosas permutaciones, esto no influye.

Por otro lado, lo único que cambiamos fue la condición para la comparación entre claves, es decir, el algoritmo es el mismo, pero lo que se entiende por estar ordenado es distinto, por lo cual resulta lógico pensar que el tiempo de ejecución deba ser el mismo.

## **6. Conclusiones finales.**

Este trabajo nos ha hecho darnos cuenta de una manera práctica de todos los contenidos explicados en las clases teóricas durante este mes y medio.

Hemos sido capaces de observar y entender con mucho mayor detalle cómo funcionan algunos algoritmos de ordenación como son InsertSort e InsertSortInv y, sobre todo, cómo calcular su rendimiento y eficacia.

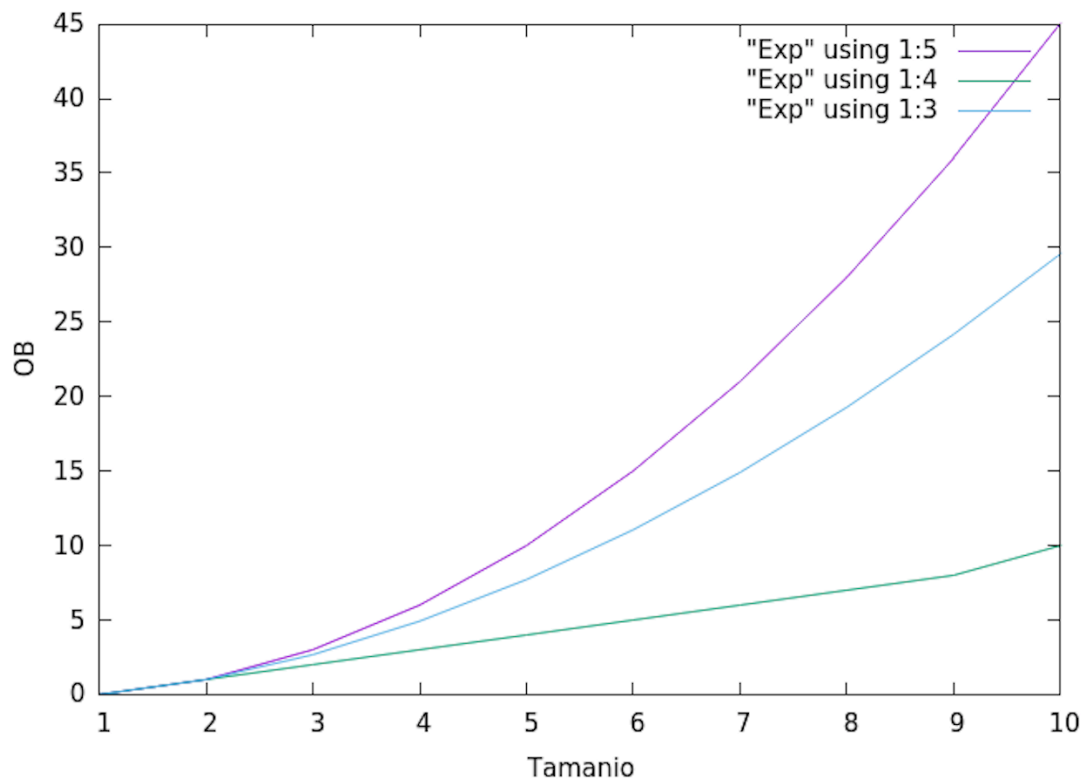
También hemos hecho uso de funciones muy útiles e interesantes como rand, para generar números aleatorios, o clock, para medir tiempos, que nunca antes habíamos usado.

Gracias a las gráficas hemos podido comparar los números de OBs que hacían nuestros algoritmos para permutaciones de diferentes tamaños y entender cómo estos funcionaban en realidad. Otra cosa que llama la atención es cómo, de manera práctica, se puede observar la semejanza esperada existente entre ambos algoritmos (InsertSort e InsertSortInv) para ordenar tablas. Es decir, es evidente que aproximadamente la mitad de permutaciones serán ordenadas más rápidamente con un algoritmo y la otra mitad con el otro, pero ser capaces de observar esto de manera práctica es algo muy interesante.

```
Abrir  [icon] *hola  Guardar  [icon] [icon] [icon]
~/Documentos/Análisis de algoritmos/Practica 1

1 0.006296 0.000000 0 0
2 0.014944 1.000000 1 1
3 0.028782 2.665642 2 3
4 0.044502 4.917514 3 6
5 0.063703 7.717478 4 10
6 0.089199 11.047507 5 15
7 0.113013 14.905144 6 21
8 0.138346 19.280685 7 28
9 0.164167 24.173360 8 36
10 0.193834 29.569611 10 45

Texto plano  Anchura del tabulador: 8  Ln 8, Col 26  INS
root@kali:~/Documentos/Análisis de algoritmos/Practica 1# ./ejercicio5 -num_min
1 -num_max 10 -incr 1 -numP 1000000 -fichSalida hola
Practica numero 1, apartado 5
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
Salida correcta
```





En estas imágenes es posible apreciar como el número de OBs máximo (caso peor), el número mínimo (caso mejor) y el número medio (caso medio) se corresponden con el rendimiento esperado del algoritmo InsertSort. Para comprobar esto lo que hicimos fue generar 1000000 de tablas de 1 a 10 elementos, para intentar obtener todas las posibles permutaciones.

Un ejemplo claro en este caso es el de las tablas de 9 elementos, en las que se aprecia un caso mejor de 8 OBs ( $N-1$ ), uno peor de 36 OBs ( $N(N-1)/2$ ) y uno medio de 24 ( $N^2/4 + O(N)$ ).

Es muy sorprendente poder observar estos resultados teóricos de manera práctica.