Análisis de Algoritmos 2019/2020

Práctica 1

Junco de las Heras y Jiaxue Jin

Grupo 1201

Código	Gráficas	Memoria	Total

1. Introducción.

Esta práctica se divide en tres bloques:

1. La generación de números aleatorios y permutaciones.

2. La implementación del algoritmo de ordenación de InsertSort.

3. La implementación de las funciones que calculan el tiempo de ejecución de algoritmo de ordenación.

Cada bloque está compuesto por varios apartados, cuyos objetivos son:

1) Implementación de la función que genera números aleatorios equiprobables: **int aleat _num (int inf, int sup)**.

2) Implementación de la función que genera permutaciones: **int *genera perm (int N)**.

3) Implementación de la función que genera un número determinado de permutaciones equiprobables: **int** genera_permutaciones (int n_perms, int N)**.

4) Implementación del algoritmo de ordenación InsertSort: **int InsertSort (int* tabla, int ip, int iu)**.

5) Partiendo de las permutaciones aleatorias generadas, medir y analizar el tiempo de ordenación del algoritmo de ordenación InsertSort. Las funciones a implementar son:

short tiempo medio ordenacion (pfunc ordena metodo, int n perms, int N, PTIEMPO ptiempo)

short genera tiempos ordenacion (pfunc ordena metodo, char * fichero, int num min, int num max, int incr, int n perms)

short guarda tabla tiempos (char *fichero, PTIEMPO tiempo, int n tiempos)

6) Implementación del algoritmo de ordenación InsertSortInv que ordena en orden de mayor a menor: **int InsertSortInv (int* tabla, int ip, int iu)**.

2. Objetivos

2.1 Apartado 1

En este apartado vamos a implementar la función **int aleat_num (int inf, int sup)** utilizando la función rand() incluida en la biblioteca stdlib.

El objetivo de la función es generar un número aleatorio equiprobable en el rango establecido por los enteros inf, sup.

2.2 Apartado 2

En este apartado implementamos una función que genera permutaciones aleatorias: **int *genera perm (int N)**. La función devuelve un puntero a enteros, que serán las permutaciones generados mediante la función.

2.3 Apartado 3

En este apartado, implementamos la función **int** genera_permutaciones (int n_perms, int N)**, cuyo objetivo es: generar mediante la función implementada en el apartado 2 n_perms permutaciones equiprobables de N elementos cada una.

La función devuelve una tabla de punteros a enteros.

2.4 Apartado 4

En este apartado implementamos el algoritmo de ordenación InsertSort: **int InsertSort (int* tabla, int ip, int iu),** que tiene como argumentos de entrada una tabla de enteros, el primer elemento de la tabla y el último elemento de la tabla.

El algoritmo ordena la tabla dada en orden de menor a mayor y devuelve el número de veces de la ejecución de la operación básica del algoritmo.

2.5 Apartado 5

Este ejercicio consiste en: primero generar las permutaciones aleatorias, y posteriormente calcular y analizar el tiempo de ordenación del algoritmo definida en el apartado anterior.

Tenemos que implementar las siguientes funciones:

short tiempo_medio_ordenacion (pfunc_ordena metodo, int n_perms, int N, PTIEMPO ptiempo)

Esta función mide el tiempo medio que tarda el algoritmo InsertSort en ordenar una lista de elementos, calcula el número promedio de ejecución de la operación básica (OB) y analiza, obteniendo el número mínimo y máximo de ejecución de la OB.

Tiene como variables de entrada: un puntero a la función de ordenación previamente definida en el fichero de extensión .h: pfunc_ordena metodo, el numero de permutaciones a generar: n_perms, el tamaño de cada permutación: N y un puntero a una estructura: PTIEMPO.

Devuelve OK si no ha habido errores, sino ERR.

short genera tiempos ordenacion (pfunc_ordena metodo, char * fichero, int num_min, int num_max, int incr, int n perms)

Está función escribe en un fichero los tiempos medios y los números promedio, mínimo y máximo de veces que ejecuta la OB en la ejecución del algoritmo de ordenación.

Tiene como variables de entrada un puntero a la función de ordenación (previamente definida en el fichero de extensión .h): pfunc_ordena, el fichero en donde se va a escribir los datos: fichero, el número mínimo de veces que ha ejecutado la OB: num_min, y el máximo: num_max, el incremento de tamaño: incr, y el número de permutaciones: n_perms.

Devuelve OK si no ha habido errores sino ERR.

short guarda tabla tiempos (char *fichero, PTIEMPO tiempo, int n tiempos)

Esta función imprime en un fichero una tabla con cinco columnas de un tamaño determinado, al tiempo de ejecución y al número promedio, mínimo y máximo de ejecución de OB.
Tiene como argumentos de entrada: el fichero donde se imprime la tabla: fichero, la tabla que contiene los datos a guardar en el fichero: tiempo y el número de elementos de la tabla tiempo: n_tiempos.

Devuelve OK si no ha habido errores sino ERR.

2.6 Apartado 6

En este apartado tenemos que implementar el algoritmo de ordenación similar al de InsertSort: **int InsertSortInv (int* tabla, int ip, int iu)**, pero en este caso se ordena de mayor a menor los elementos dados.

Es decir, la lista de elementos que resulta después de la ordenación es el inverso al resultado que se obtiene tras la ejecución del algoritmo de InsertSort.

3. Herramientas y metodología

Aquí ponéis qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas habéis utilizado (Netbeans, Visual Studio, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc.) y qué metodologías de desarrollo y soluciones al problema planteado habéis empleado en cada apartado. Así como las pruebas que habéis realizado a los programas desarrollados.

En esta práctica hemos utilizado Linux como el entorno de desarrollo, y las herramientas utilizadas son gcc, Valgrind, Gnuplot y Atom. Las pruebas eran ejecutarlas con valores mínimos y extremos y luego graficarlas, además de pasarles Valgrind.

3.1 Apartado 1

Basando en que la función rand() nos devuelve un número entre 0 y RAND_MAX, implementamos esta función.

La división de rand()/RAND_MAX es 0 o 1, por esta razón, hacemos el casting de RAND_MAX a float, en nuestro caso le sumamos una unidad (1.0). Esta suma hace posible que el resultado obtenido en la división referida sea un número entre 0 y 1 sin incluir el 1.

Nuestro objetivo en este apartado es de generar un número aleatorio equiprobable en un rango dado: (inf, sup).

Para ello, realizamos la operación (rand() / (RAND_MAX + 1.0))*(sup - inf + 1) + inf. Con rand() / (RAND_MAX + 1.0)) nos genera un número entre 0 y 1(no incluido), y a este número le multiplicamos para que el numero obtenido sea dentro del rango dado.

3.2 Apartado 2

La función genera_perm utiliza la función que genera números aleatorios para construir permutaciones. El resultado de la función genera_perm es una lista con N números aleatorios.

3.3 Apartado 3

Para generar una lista de permutaciones equiprobables, hacemos uso de la función genera_perm ya que en esta función se apoya de otra función que asegura la generación números aleatorios equiprobables.

3.4 Apartado 4

Basando en el funcionamiento del algoritmo de InsertSort implementamos esta función.
El algoritmo de inserción itera sobre los índices (posiciones) del arreglo.
Para insertar el elemento que está en una posición en el su arreglo a su izquierda, lo comparamos repetidamente con los elementos a su izquierda, moviéndonos de derecha a izquierda.
Esta comparación corresponde a la instrucción **tabla[j] > var_swap** del bucle, donde tabla[j] es el elemento a compararse y var_swap el elemento que deseamos insertar.
En el caso de que cumpla la condición, desplazamos el elemento comparado una posición a la derecha: **tabla[j+1] = tabla[j];**

Repetimos esta operación hasta que el elemento a insertar encuentre la posición correspondiente. Y lo insertamos: **tabla[j+1] = var_swap.**

3.5 Apartado 5

En este apartado había que crear tres funciones, una que rellenase los datos de un struct tiempo, en la que se guardan el tamaño de los elementos, el número de ellos, el tiempo promedio que se tarda en ejecutar la función, el número de operaciones medio y el máximo y el mínimo de operaciones básicas. Otra función es para administrar la llamada de la función previa y la de imprimir, asegurándose de reservar y liberar memoria correctamente. La función de imprimir cogía el array de structs y los imprimía en un fichero.

3.6 Apartado 6

La implementación del algoritmo de ordenación InsertSortInv es muy similar al de InsertSort. La clave está en que este caso tenemos que ordenar de mayor a menor los números.
Para conseguirlo, cambiamos la condición de inserción. Es decir, en tanto que en el algoritmo de ordenación de InsertSort desplazamos el elemento comparado a la derecha cuando cumple la condición de que es mayor que el elemento a insertar: tabla[j] > var_swap, el InsertSortInv lo hace cuando es menor: tabla[j] < var_swap.

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) {

if (inf > sup)

return ERR;

return (rand() / (RAND_MAX + 1.0))*(sup - inf + 1) + inf;

}
```

4.2 Apartado 2

```
int* genera_perm(int N) {

        int *perm = NULL, i, ale, swap_var;
```

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


        perm = (int*) malloc(N * sizeof (int));
        if (!perm)
                return NULL;
        for (i = 0; i < N; i++)
                perm[i] = i + 1;
        for (i = 0; i < N; i++) {
                ale = aleat_num(i, N - 1);
                swap_var = perm[ale];
                perm[ale] = perm[i];
                perm[i] = swap_var;
        }


        return perm;
}
```

4.3 Apartado 3

```
int** genera_permutaciones(int n_perms, int N) {
        int **perms = NULL, i;
        if (n_perms < 0 || N < 0)
                return NULL;
```

```
	perms = (int**) malloc(n_perms * sizeof (int*));
	if (!perms)
		return NULL;

	for (i = 0; i < n_perms; i++) {
		perms[i] = genera_perm(N);
		if (!perms[i]) {
			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) {
	/* vuestro codigo */
	/*var_swap is the temporaly variable to swap elements*/
	int i, j, var_swap, cont = 0;
```

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

        for (i = ip + 1; i <= iu; i++) {
                var_swap = tabla[i];
                j = i - 1;
                while (j >= ip && ++cont && tabla[j] > var_swap) {
                        tabla[j + 1] = tabla[j];
                        j--;
                }
                tabla[j + 1] = var_swap;
        }

        return cont;
}
```

4.5 Apartado 5

```
int InsertSortInv(int* tabla, int ip, int iu) {
        /* vuestro codigo */
        /*var_swap is the temporaly variable to swap elements*/
        int i, j, var_swap, cont = 0;

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

```
        for (i = ip + 1; i <= iu; i++) {
                var_swap = tabla[i];
                j = i - 1;
                while (j >= ip && ++cont && tabla[j] < var_swap) {
                        tabla[j + 1] = tabla[j];
                        j--;
                }
                tabla[j + 1] = var_swap;
        }

        return cont;
}
```

4.6 Apartado 6

```
short tiempo_medio_ordenacion(pfunc_ordena metodo,
int n_perms,
int N,
PTIEMPO ptiempo) {
        /* vuestro codigo */
        int **tablas = NULL, OBS, max_obs = -1, min_obs = -1, j;
        clock_t t1, t2;

        if (!metodo || n_perms <= 0 || N <= 0 || !ptiempo)
```

```
        return ERR;


tablas = genera_permutaciones(n_perms, N);
if (!tablas)
        return ERR;


t1 = clock();


ptiempo->medio_ob = 0;


for (j = 0; j < n_perms; j++) {
        OBS = metodo(tablas[j], 0, N - 1);


        if (OBS == ERR) {
                for (j = 0; j < n_perms; j++) {
                        free(tablas[j]);
                }
                free(tablas);
                return ERR;
        }


        ptiempo->medio_ob += OBS / (double) n_perms;
```

```
        if (j == 0)
                max_obs = min_obs = OBS;
        else {
                if (OBS > max_obs)
                max_obs = OBS;
                if (OBS < min_obs)
                min_obs = OBS;
        }
}

t2 = clock();

ptiempo->N = N;
ptiempo->n_elems = n_perms;

ptiempo->tiempo = (t2 - t1) / (double) n_perms;

ptiempo->min_ob = min_obs;
ptiempo->max_ob = max_obs;

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) {
        /* vuestro codigo */
        int j, cont = 0, tam_tabla;
        PTIEMPO tiempos;

        if (!metodo || !fichero || num_min > num_max || incr <= 0 || n_perms <= 0)
                return ERR;

        /*tam_tabla = ceil((num_max - num_min) / (double) incr));*/
        tam_tabla = (num_max - num_min + 1 + (incr - 1)) / incr;

        tiempos = (PTIEMPO) malloc(tam_tabla * sizeof (TIEMPO));
        if (!tiempos)
                return ERR;

        for (j = num_min; j <= num_max; j += incr) {
                if (tiempo_medio_ordenacion(metodo, n_perms, j, &tiempos[cont++])
        == ERR) {
                        free(tiempos);
```

```
                return ERR;
            }
        }

        if (guarda_tabla_tiempos(fichero, tiempos, n_perms) == ERR) {
            free(tiempos);
            return ERR;
        }

        free(tiempos);
        return OK;
}
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos) {
        /* vuestro codigo */
        int i;
        FILE *f = NULL;
        if (!fichero || !tiempo || n_tiempos < 0)
            return ERR;
        f = fopen(fichero, "w");
        if (!f)
            return ERR;

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

```
        if (fprintf(f, "%d %f %f %d %d\n", tiempo[i].N, tiempo[i].tiempo,
tiempo[i].medio_ob, tiempo[i].max_ob, tiempo[i].min_ob) <= 0) {
                fclose(f);
                return ERR;
        }
    }

    if (fclose(f) != 0)
        return ERR;
    return OK;
}
```

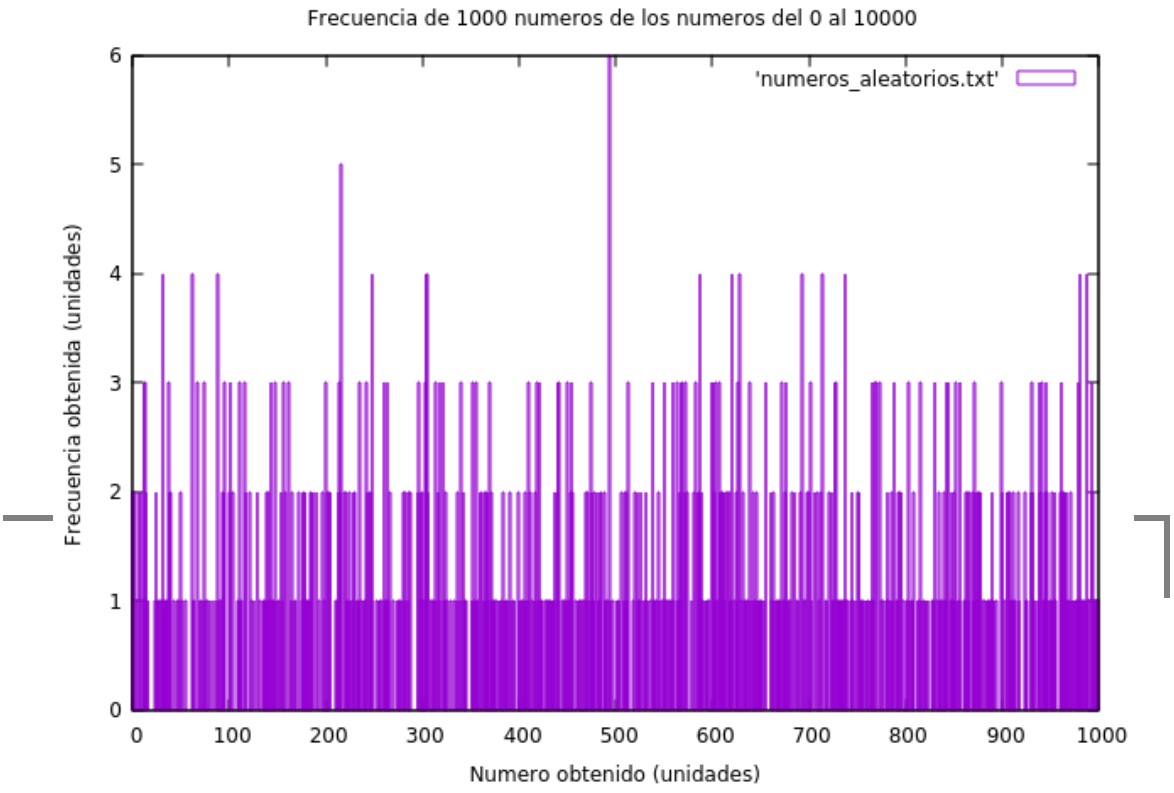
5. Resultados, Gráficas

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

5.1 Apartado 1

Resultados del apartado 1.

Gráfica del histograma de números aleatorios, comentarios a la gráfica



Es una gráfica de frecuencia donde en el eje x están los posibles números candidatos a ser elegidos (desde el 0 al 1000) y en el eje y está la frecuencia en la que ha salido (el número de veces que han salido). Podemos observar que la gran mayoría ha salido 1 vez, lo cual dice que hay bastante equiprobabilidad, además, ningún número ha salido más de 4 veces, con 2 excepciones. Se han sacado 1000 números.

5.2 Apartado 2

Resultados del apartado 2.

Utilizando la función aleatoria se creaba una permutación. Para más información del apartado 2 mirar el correspondiente apartado 3.

5.3 Apartado 3

Resultados del apartado 3.

Utilizando la función de crear permutaciones creaba n_perm permutaciones de N elementos. Para más información del apartado 2 mirar el correspondiente apartado 3.

5.4 Apartado 4

Resultados del apartado 4.

Utilizando InsertSort, primero se generaban permutaciones y luego se ordenaban, en tiempos cuadráticos. Para más información del apartado 2 mirar el correspondiente apartado 3.

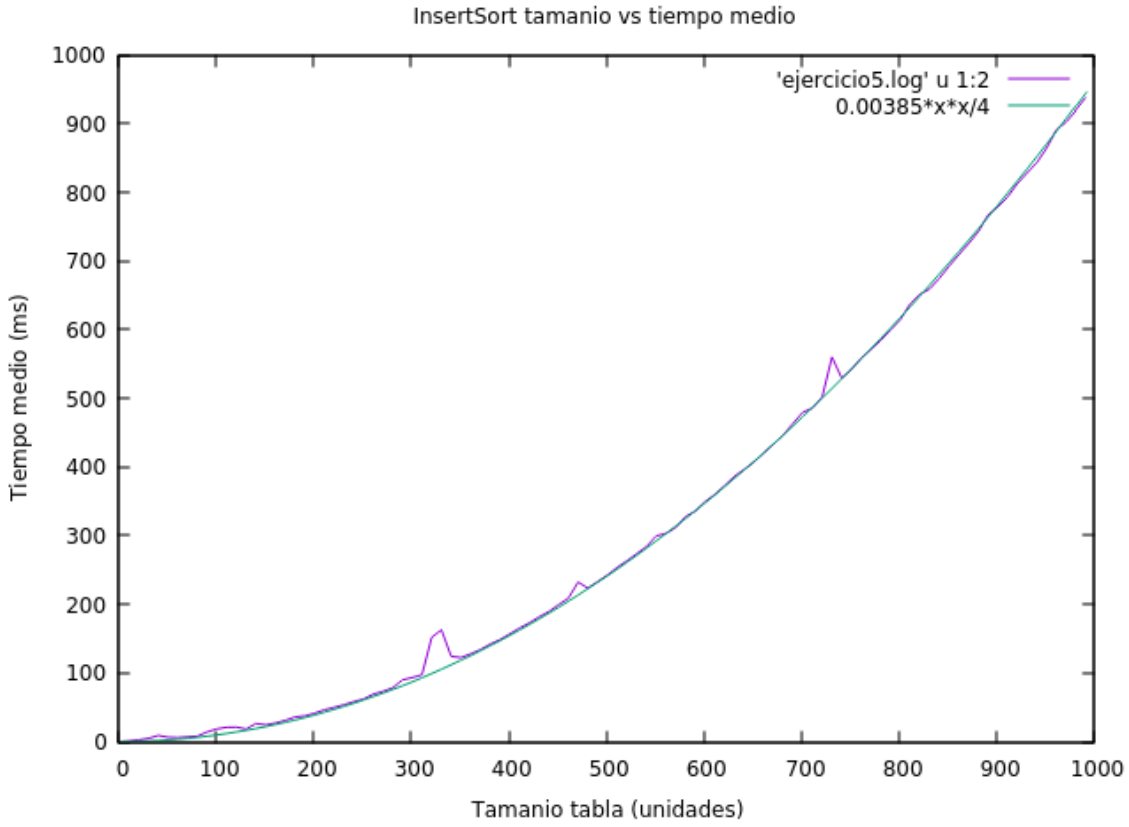
5.5 Apartado 5

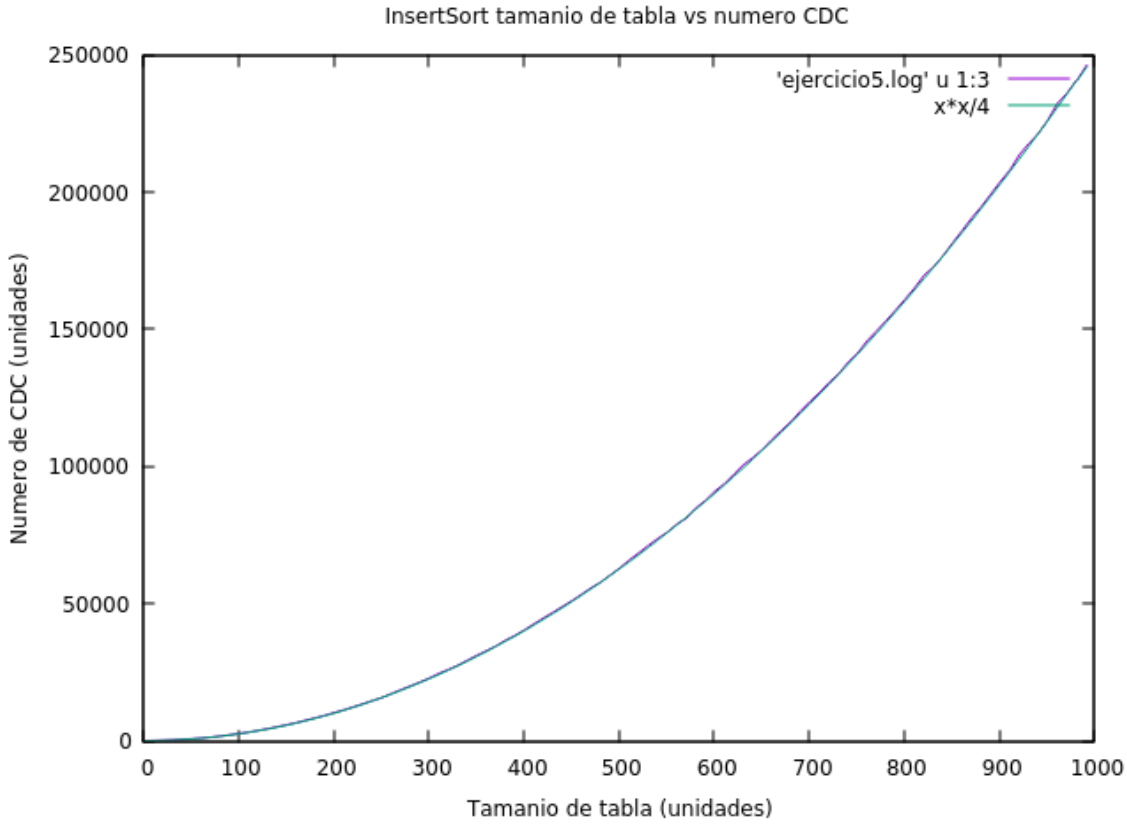
Resultados del apartado 5.

Se crearon unas funciones para medir el rendimiento de funciones abstractas. Las siguientes gráficas muestran comparaciones entre el tamaño de la tabla y la correspondiente magnitud a medir.

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.


InsertSort tamanio vs tiempo medio
'ejercicio5.log' u 1:2
0.00385*x*x/4
Tiempo medio (ms)
Tamanio tabla (unidades)
0
100
200
300
400
500
600
700
800
900
1000

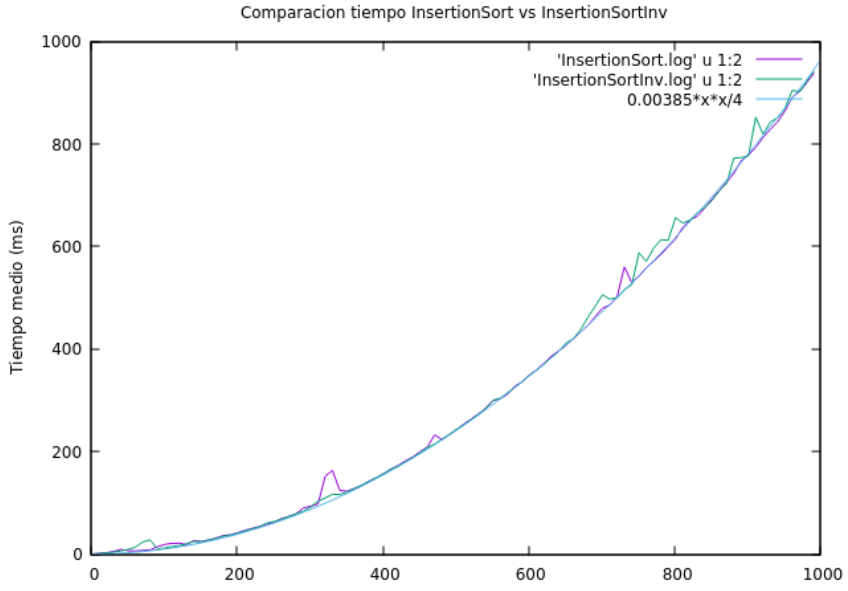

InsertSort tamanio de tabla vs numero CDC
'ejercicio5.log' u 1:3
x*x/4
Numero de CDC (unidades)
Tamanio de tabla (unidades)
0
50000
100000
150000
200000
250000
100
200
300
400
500
600
700
800
900
1000

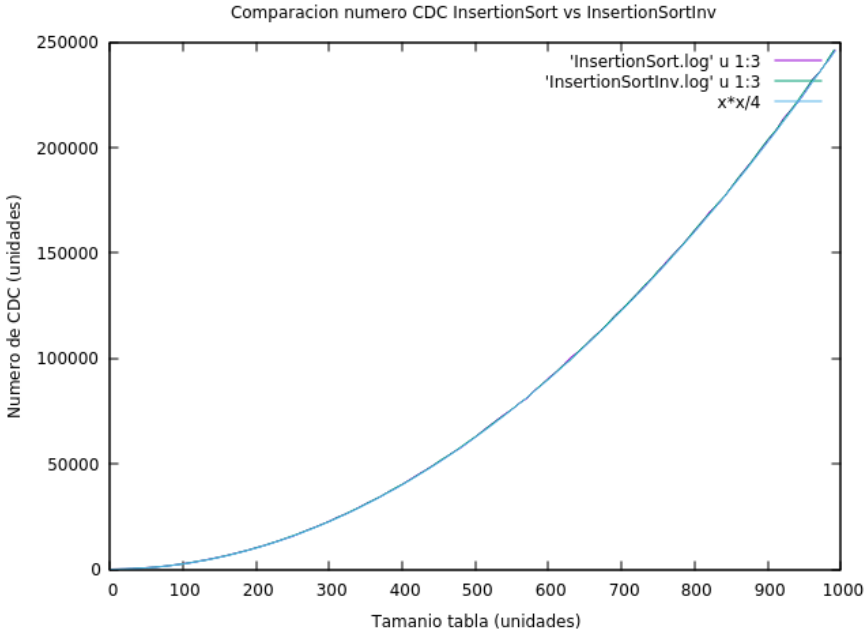
5.6 Apartado 6

Resultados del apartado 6.

Se compara el rendimiento de InsertSort y de InsertSortInv. Como se puede imaginar, ambos rendimientos y gráficas son muy similares (Aunque no tendría por qué si se da el caso de tener una lista ordenada, ya que sería el mejor caso para InsertSort y el peor para InsertSortInv)

Gráfica comparando el tiempo medio de OBs y el tiempo medio de reloj para InsertSort y InsertSortInv, comentarios a la gráfica..





6. Respuesta a las preguntas teóricas.

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

6.1 Pregunta 1

Justifica tu implementación de aleat num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.

Basando en que la función rand() nos devuelve un número entre 0 y RAND_MAX, implementamos esta función.

La división de rand()/RAND_MAX es 0 o 1, por esta razón, hacemos el casting de RAND_MAX a float, en nuestro caso le sumamos una unidad (1.0). Esta suma hace posible que el resultado obtenido en la división referida sea un número entre 0 y 1 sin incluir el 1.

Nuestro objetivo en este apartado es de generar un número aleatorio equiprobable en un rango dado: (inf, sup).

Para ello, realizamos la operación (rand() / (RAND_MAX + 1.0))*(sup - inf + 1) + inf. Con rand() / (RAND_MAX + 1.0)) nos genera un número entre 0 y 1(no incluido), y a este número le multiplicamos para que el numero obtenido sea dentro del rango dado.

Otra opción alternativa sería rand() % (sup – inf + 1) + inf, pero tiene sus desventajas, como que el 0 será un poco más probable que el sup – inf;

6.2 Pregunta 2

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

Un array con un elemento está ordenado. Estando en el paso i del algoritmo, los i-1 primeros elementos están ordenados entre sí. Basta con probar que si al acabar el paso i los i primeros están ordenados entre sí entonces al acabar los n pasos del algoritmo los n elementos estarán ordenados (inducción). El elemento i se compara con el i-1. Si es mayor entonces se acaba el paso y los i primeros elementos estarán ordenados. Sino el elemento i-1 irá a la posición i y se buscará en el elemento, o hueco, i-2. Si hay un elemento menor en la posición k que el elemento en la posición original i entonces el algoritmo habrá movido todos los elementos desde k + 1 hasta i-1 una posición a la

derecha, creando un hueco el k + 1, para el elemento original. Si el bucle acaba sin encontrar a un elemento menor entonces habrá movido desde la posición 0 hasta la i-1 una posición a la derecha, dejando el hueco 0 libre. Todos están ordenados.

6.3 Pregunta 3.

¿Por qué el bucle exterior de InsertSort no actúa sobre el primer elemento de la tabla?

El algoritmo no actúa sobre el primer elemento de la tabla porque las listas de un elemento están ordenadas, es el caso base.

6.4 Pregunta 4

¿Cuál es la operación básica de InsertSort?

La operación básica del InsertSort es la comparación de claves del elemento original con algún elemento anterior a él. Es la condición que se encuentra en el bucle interior **while (j >= ip && ++cont && tabla[j] > var_swap)** de la implementación de la función int InsertSort (int* tabla, int ip, int iu).

6.5 Pregunta 5

Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS(n) y el caso mejor BBS(n) de InsertSort. Utilizad la notación asintótica siempre que se pueda.

Suponemos que en cada ejecución del bucle, el valor que deseamos insertar es menor que cada elemento en el subarreglo a su izquierda.
Cuando ejecutamos la primera vez, k=1. La segunda vez, k=2...Y así sucesivamente hasta la última vez, cuando k=n-1. Por lo tanto, el tiempo total que tarda la inserción en un subarreglo ordenado es:

$$\sum_{i=1}^{n-1} i - 1 = \frac{n(n-1)}{2}$$

En este caso, el tiempo de ejecución es$\Theta(n^2)$
Teniendo en cuenta que este bucle al menos se ejecuta una vez:
Por ejemplo: Dada la lista[1, 2, 3, 4], donde el subarreglo ordenado es la formada por [1, 2, 3] y 4 es el elemento a insertar. Con la primera comparación encontramos que 4 es mayor que 3, por tanto, ningún elemento en el subarreglo necesita recorrerse hacia la derecha.
Entonces esta llamada tarda solo una única vez, como hay n-1 llamadas:

$$\sum_{i=0}^{n-1} 1 = n - 1$$

En este caso, el tiempo de ejecución es$\Theta(n)$.
La ejecución del bucle causa que cada elemento se recorra si el elemento que está siendo insertada es menor que todos los elementos a su izquierda. Entonces, si cada elemento es menor que todos los elementos a su izquierda, el tiempo de ejecución del ordenamiento por inserción es $\Theta(n^2)$. Es decir, el arreglo tendría que empezar en orden inverso, por ejemplo: [4, 3, 2, 1]. Entonces un arreglo ordenado de manera inversa es el peor de los casos para el ordenamiento por inserción.

$$W_{IS}(n) = \frac{n(n-1)}{2}$$

En caso contrario, si el elemento que está siendo insertada es mayor o igual que todos los elementos a su izquierda, no causa que ningún elemento se recorra hacia la derecha. Entonces, si todo elemento es mayor o igual a cualquier elemento a su izquierda, el tiempo de ejecución del ordenamiento por inserción es $\Theta(n)$. Esta situación ocurre si el arreglo inicialmente ya se encuentra ordenado.
Por tanto, un arreglo ya ordenado es el mejor caso para un ordenamiento por inserción.

$$B_{IS}(n) = n - 1$$

6.6 Pregunta 6

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

Las funciones a la que tiende los tiempos y el número de comparación de claves entre InsertSort e InsertSortInv son muy similares sino iguales. En la gráfica del punto 5, donde comparábamos los rendimientos de los algoritmos se podían ver que se asemejaban mucho. Esto no ocurre si las tablas no son aleatorias, es decir, casos como que la tabla esté ordenada. Entonces para InsertSort no le costaría mucho (n – 1 operaciones), pero para InsertSortInv estaría todo lo inverso a ordenada (n*(n-1)/2 operaciones).

7. Conclusiones finales.

Discusión final sobre la práctica y los resultados obtenidos.

Hemos aprendido cómo funciona y cómo se programa un algoritmo de ordenación cuadrático (InsertSort), cómo se hace un generador de números aleatorios mejor que rand() % mod y cómo se cronometra correctamente con clock(). También era nuevo la parte de graficar los datos, tarea sencilla con Gnuplot.