Análisis de Algoritmos 2015/2016

Práctica 1

Alfonso Villar y Víctor García, Grupo 12.

Código	Gráficas	Memoria	Total

1. Introducción.

Esta, como primera práctica de Análisis de Algoritmos, ha servido para adaptarnos de nuevo a programar en C tras varios meses y a implementar algoritmos de forma rigurosa por primera vez. Trabajamos con algoritmos como InsertSort, y mediante la implementación de las funciones descritas más abajo, además de entender su funcionamiento, mediante la elaboración de gráficas y gracias a trabajar con la potencia y capacidad de cálculo de un ordenador, podemos comprobar (a un nivel bastante razonable) la aleatoriedad de nuestras funciones.

2. Objetivos

La práctica básicamente consiste en la generación de varios ficheros .c llamados permutaciones.c ordenacion.c tiempos.c a partir de los .h dados por la práctica. Los 3 primeros ejercicios serán para implementar permutaciones.c, el 4 y 6 para implementar ordenacion.c y el 5 para implementar tiempos.c

2.1 Ejercicio 1

Consiste en usar la función rand que se encuentra en la librería stdlib para construir una rutina int aleat_num (int inf, int sup) que genere un número aleatorio equiprobable entre los enteros inf, sup, ambos inclusive. Dicho ejercicio será comprobado ejecutando ejercicio1.c, además de elaborar un histograma para comprobar la aleatoriedad de la función y la equiprobabilidad de los dígitos.

2.2 Ejercicio 2

En este ejercicio hay que escribir una rutina C int *genera_perm(int n) que implemente la función aleat_num creada en el ejercicio1. Esta nueva función creará una permutación de tamaño n.

2.3 Ejercicio 3

Para este ejercicio hay que construir la rutina int ** genera_permutaciones(int n_perms, int tamanio) que, utilizando la función creada en el ejercicio2, genera “n_perms” permutaciones de longitud “tamanio”.

2.4 Ejercicio 4

Hay que crear la función int InsertSort(int *tabla, int ip, int iu) siendo “tabla” una tabla de enteros, ip es el primer elemento de la tabla e iu es el último elemento de esta. Este algoritmo ordena los elementos de la tabla entre ip e iu.

2.5 Ejercicio 5

Aquí implementaremos en el fichero tiempos.c la función short tiempo_medio_ordenacion (pfunc_ordena metodo, int n_perms, int tamanio, PTIEMPO ptiempo), siendo “metodo” el método que se utilizará, “n_perms” el número de permutaciones, “tamanio” el tamaño de las permutaciones y “ptiempo” una estructura con los datos obtenidos al realizar la ordenación de las permutaciones. Esta función por tanto generará permutaciones aleatorias, las ordenará y finalmente introducirá los datos obtenidos en ptiempo.

Además crearemos la función short genera_tiempos_ordenacion (pfunc_ordena metodo, char * fichero, int num_min,int num_max,int incr, int n_perms) siendo “metodo” el método a utilizar, “fichero” donde guardaremos los resultados, “num_min” y “num_max” será el tamaño mínimo y máximo de las permutaciones respectivamente, “incr” el incremento del tamaño de las permutaciones y “n_perms” el número de permutaciones a realizar por tamaño.

2.6 Ejercicio 6

Implementaremos la rutina int InsertSortInv(int* tabla, int ip, int iu) con los mismos argumentos y retorno que InsertSort pero que ordene la tabla en orden inverso (de mayor a menor) y después compararemos los tiempos de ejecución y los resultados obtenidos con esta nueva función con los de la rutina InsertSort.

3. Herramientas y metodología

Aquí ponéis qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas habéis utilizado (Netbeans, 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.

3.1 Ejercicio 1

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

En lo que respecta a la metodología, a la hora de implementar la función se propusieron dos métodos en clase. A pesar de que ambos eran válidos, el primero propuesto (1/rand() +1)*(sup-inf +1), dado que se trataba de un algoritmo que realiza una división, en lo referente a la velocidad de procesamiento, la división hace que este algoritmo no sea tan rápido como el que nosotros implementamos (inf + rand() % (sup - inf + 1)). El nivel de aleatoriedad de este algoritmo es alto, como se puede apreciar en el histograma que se encuentra más adelante, pero al hacer la operación de módulo, varios números aleatorios distintos pueden generar, al realizar esa operación, el mismo número.

3.2 Ejercicio 2

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

Para que la permutación generada fuera aleatoria, decidimos hacer un bucle donde, por cada dígito de la permutación, por cada elemento del array (desde el primero al último), se hacía un swap entre éste y otro en una posición aleatoria del array, llamando a la función implementada anteriormente.

3.3 Ejercicio 3

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

Para la realización de esta función utilizamos la del ejercicio 2 para generar las permutaciones en un ciclo for que las realiza “n_perms” veces de tamaño “tamanio”.

3.4 Ejercicio 4

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

Para la realización del algoritmo InsertSort nos hemos apoyado en el pseudocódigo dado en teoría, lo que nos ha facilitado la realización de este algoritmo a la hora de implementarlo en código C.

3.5 Ejercicio 5

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

En este ejercicio realizaremos 3 funciones; tiempo_medio_ordenacion el cual generará la estructura ptiempo y le dará valores para su posterior utilización, genera_tiempos_ordenacion escribe en el fichero los tiempos medios, y los números promedio, mínimo y máximo de veces que se ejecuta la OB y guarda_tabla_tiempos la cual imprime una tabla con tamaño tiempo medio_ob max_ob y min_ob. También cabe destacar que tuvimos un error en ptiempo->tiempo, ya que no calculaba bien el tiempo promedio y nos dimos cuenta al realizar las gráficas y así lo cambiamos al correcto.

3.6 Ejercicio 6

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

Para la realización del algoritmo InsertSortInv hemos visto que su estructura iba a ser prácticamente igual que la de InsertSort, pero que debería ordenar al revés, por lo que cambiamos el signo de la ordenación.

4. Código fuente

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

4.1 Apartado 1

```
/**************************************************/
/* Funcion: aleat_num Fecha: 30/09/2016          */
/* Autores: Alfonso Villar y Victor Garcia       */
/*                                                */
```

```
/* Rutina que genera un numero aleatorio    */
/* entre dos numeros dados                  */
/*                                          */
/* Entrada:  0 <= inf <= sup                */
/* int inf: limite inferior                 */
/* int sup: limite superior                 */
/*                                          */
/* Salida:                                  */
/* int: numero aleatorio                    */
/********************************************/
int aleat_num(int inf, int sup)
{
        assert(inf <= sup);
        return inf + rand() % (sup - inf + 1);
}
```

4.2 Apartado 2

```
/********************************************/
/* Funcion: genera_perm Fecha: 30/09/2016   */
/* Autores: Alfonso Villar y Victor Garcia  */
/*                                          */
/* Rutina que genera una permutacion        */
/* aleatoria                                */
/*                                          */
/* Entrada:     0 < n                       */
/* int n: Numero de elementos de la         */
/* permutacion                              */
/*                                          */
/* Salida:                                  */
/* int *: puntero a un array de enteros     */
```

```
/* que contiene a la permutacion                */
/* o NULL en caso de error                      */
/************************************************/
int* genera_perm(int n)
{
        int i;
        int* perm = (int *) malloc(n * sizeof(perm[0]));

        if (perm == NULL)
                return NULL;

        for (i = 0; i < n; i++)
                perm[i] = i + 1;

        for (i = 0; i < n; i++){
                int aux = perm[i];
                int ran = aleat_num(i, n - 1);
                perm[i] = perm[ran]; /*Hacemos un swap iterativo entre cada elemento de la
tabla y otro en una posición aleatoria*/
                perm[ran] = aux;
        }

        return perm;
}
```

4.3 Apartado 3

```
/************************************************/
/* Funcion: genera_permutaciones Fecha: 30/09/2016 */
/* Autores: Alfonso Villar y Victor Garcia      */
/*                                              */
/* Funcion que genera n_perms permutaciones     */
```

```
/* aleatorias de tamanio elementos              */
/*                                              */
/* Entrada:   0 < n_perms && 0 < tamanio        */
/* int n_perms: Numero de permutaciones         */
/* int tamanio: Numero de elementos de cada     */
/* permutacion                                  */
/*                                              */
/* Salida:                                      */
/* int**: Array de punteros a enteros           */
/* que apuntan a cada una de las                */
/* permutaciones                                */
/* NULL en caso de error                        */
/************************************************/
int** genera_permutaciones(int n_perms, int tamanio)
{
        int i;
        int** resul = (int **) malloc(n_perms * sizeof (resul[0]));

        if (resul == NULL)
                return NULL;

        for (i = 0; i < n_perms; i++)
                if ((resul[i] = genera_perm(tamanio)) == NULL){
                        free (resul);
                        return NULL;
                        }
        return resul;
}
```

4.4 Apartado 4

```
/************************************************/
/* Funcion: InsertSort Fecha: 7/10/2016         */
/* Autores: Alfonso Villar y Víctor García      */
/*                                              */
/* Entrada:   0 < ip <= iu                      */
/* int *tabla: Tabla con los numeros            */
/* int ip: Direccion del primer elemento a ordenar */
/* int iu: Direccion del ultimo elemento a ordenar */
/*                                              */
/* Salida:                                      */
/* cont: El numero de veces que se ejecuto la OB   */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                   */
/************************************************/
int InsertSort(int* tabla, int ip, int iu)
{
        int cont = 0;
        int i;
        if (tabla == NULL) return ERR;
                for (i = ip + 1; i <= iu; i++){
                        int j = i-1;
                        int aux = tabla[i];
                        while (j >= ip && tabla[j] > aux){
                                tabla[j+1] = tabla[j];
                                j--;
                                cont++;
                                }
                        cont++;
                        tabla[j+1] = aux;
                }
```

```
        return cont;
}
```

4.5 Apartado 5

```
/**************************************************/
/* Funcion: tiempo_medio_ordenacion Fecha:12/10/16 */
/* Autores: Alfonso Villar y Víctor García        */
/*                                                */
/* Entradas:  0 <= n_perms  &&   0<= tamanio      */
/* pfunc_ordena metodo: es un puntero a la funcion */
/* de ordenacion                                  */
/* int n_perms: numero de permutaciones a generar y*/
/* ordenar                                        */
/* int tamanio: tamanio de cada permutacion       */
/* ptiempo: puntero a la estructura de tipo PTIEMPO*/
/*                                                */
/* Salida:                                        */
/* Devuelve ERR si hay error u OK si ordena       */
/* las tablas correctamente                       */
/*                                                */
/**************************************************/
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                    int n_perms,
                    int tamanio,
                    PTIEMPO ptiempo)
{
        int i;
        int medio_ob = 0;
        int max_ob = 0;
        int min_ob = INT_MAX;
```

```
	int ob = 0;
	clock_t ini, fin;
	int **perms = genera_permutaciones(n_perms, tamanio);
	if (perms == NULL){
		return ERR;
	}
	ini = clock(); /*ini = clock_gettime(); */
	for (i = 0; i < n_perms; i++){
		int ob = metodo(perms[i], 0, tamanio - 1);
		if (ob == ERR)
			return ERR;
		if (max_ob < ob) /*Comparamos OB iterativamente con max_ob para
actualizar
							el numero maximo de OB*/
			max_ob = ob;
		if (min_ob > ob)/*Comparamos OB iterativamente con min_ob para
actualizar
							el numero minimo de OB*/
			min_ob = ob;
		medio_ob += ob;
	}

	fin = clock(); /*fin = clock_gettime(); */
	ptiempo->n_perms = n_perms;
	ptiempo->tamanio = tamanio;
	ptiempo->medio_ob = (double) medio_ob / n_perms;
	ptiempo->max_ob = max_ob;
	ptiempo->min_ob = min_ob;
	ptiempo->tiempo = (double) (fin - ini) / n_perms / CLOCKS_PER_SEC;
	ptiempo->n_veces = ob;
	for (i = 0; i < n_perms; i++)
		free(perms[i]);
```

```
        free(perms);
        return OK;
}

/***************************************************/
/* Funcion: genera_tiempos_ordenacion Fecha:12/10/16*/
/* Autores: Alfonso Villar y Víctor García        */
/*                                                 */
/* Entradas:  0 <= num_min, num_max, incr, n_perms */
/*                 && num_min < num_max       */
/* pfunc_ordena metodo: es un puntero a la funcion */
/* de ordenacion                                   */
/* char* fichero: fichero donde se escriben los    */
/* tiempos medios, los numeros promedio, minimo y  */
/* maximo de veces que se ejecuta la OB           */
/* int num_min: numero minimo (e incluido) del     */
/* rango del tamanio de las permutaciones         */
/* int num_max: numero maximo (e incluido) del     */
/* rango del tamanio de las permutaciones         */
/* int incr: valor del incremento del rango del    */
/* tamanio de las permutaciones                   */
/* int n_perms: numero de permutaciones           */
/*                                                 */
/* Salida:                                         */
/* Devuelve ERR si hay error u OK si ordena       */
/* las tablas correctamente                       */
/*                                                 */
/***************************************************/
short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                        int num_min, int num_max,
                        int incr, int n_perms)
```

```
{
	int i;
	int iteraciones;
	iteraciones = (int)((num_max-num_min)/(incr))+1;
	PTIEMPO ptiempo;
	ptiempo = (PTIEMPO) malloc (iteraciones * (sizeof (TIEMPO)));
	if (ptiempo == NULL)
		return ERR;

	for (i = 0; i < iteraciones; i++){
		if (tiempo_medio_ordenacion(metodo, n_perms, (num_min+i*incr) ,
&ptiempo[i]) == ERR){
			free(ptiempo);
			return ERR;
		}
		if (guarda_tabla_tiempos (fichero, &ptiempo[i], 1) == ERR){
			free(ptiempo);
			return ERR;
			}
		}
	free(ptiempo);
	return OK;
}

/************************************************/
/* Funcion: guarda_tabla_tiempos Fecha:  13/10/16  */
/*                                  */
/* Entradas:  0 <= N  && fichero /= NULL         */
/*                                  */
/* char* fichero: fichero donde se escriben los    */
/* tamanios de la permutacion, el tiempo de       */
```

```
/* ejecucion y el numero promedio, maximo y minimo */
/* de veces que se ejecuta la OB                */
/* maximo de veces que se ejecuta la OB         */
/* PTIEMPO tiempo: guarda los tiempos de ejecucion */
/* int N: tamanio del array tiempo              */
/*                                              */
/* Salida:                                      */
/* Devuelve ERR si hay error u OK si ordena     */
/* las tablas correctamente                     */
/*                                              */
/************************************************/
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int N)
{
        FILE *fp = fopen(fichero, "a");
        if (fp == NULL)
                return ERR;

        fprintf (fp, "%-20d %-20.9f %-20.2f %-20d %-20d\n", tiempo->tamanio,
                                tiempo->tiempo,
                                tiempo->medio_ob,
                                tiempo->max_ob,
                                tiempo->min_ob);
        fclose(fp);
        return OK;
}
```

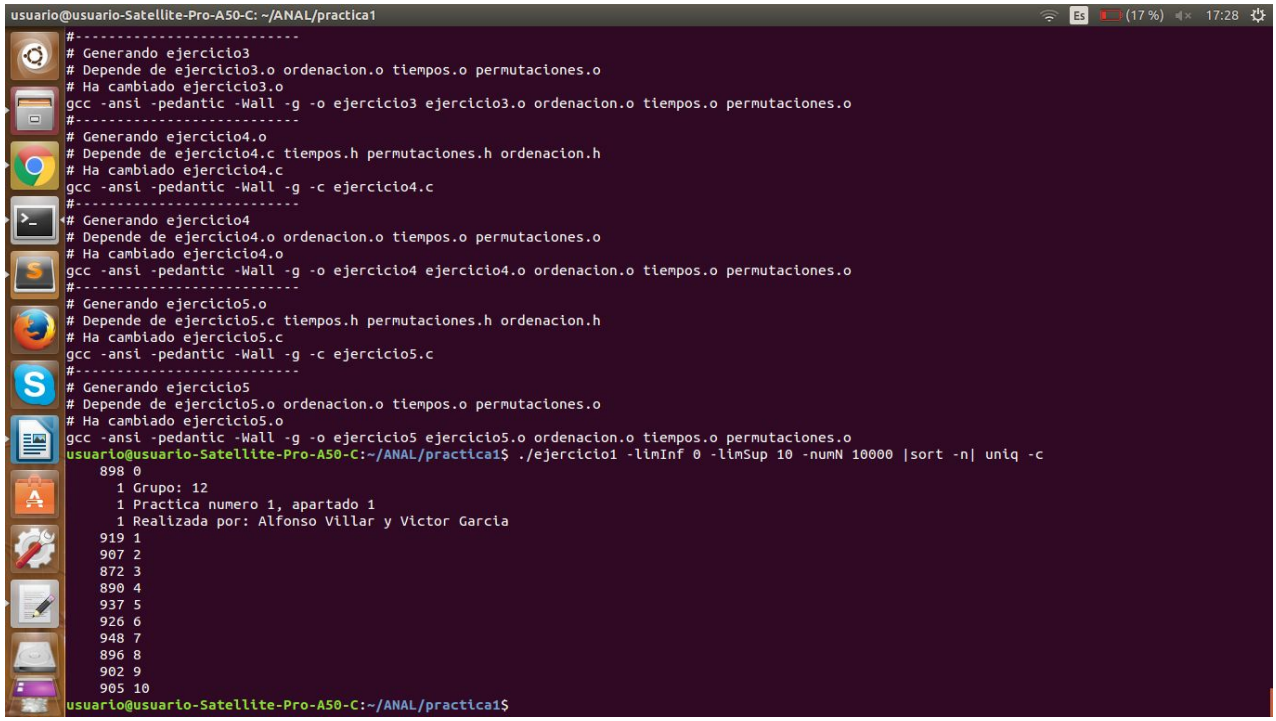
4.6 Apartado 6

```
/************************************************/
/* Funcion: InsertSortInv Fecha: 7/10/2016      */
/* Autores: Alfonso Villar y Víctor García      */
```

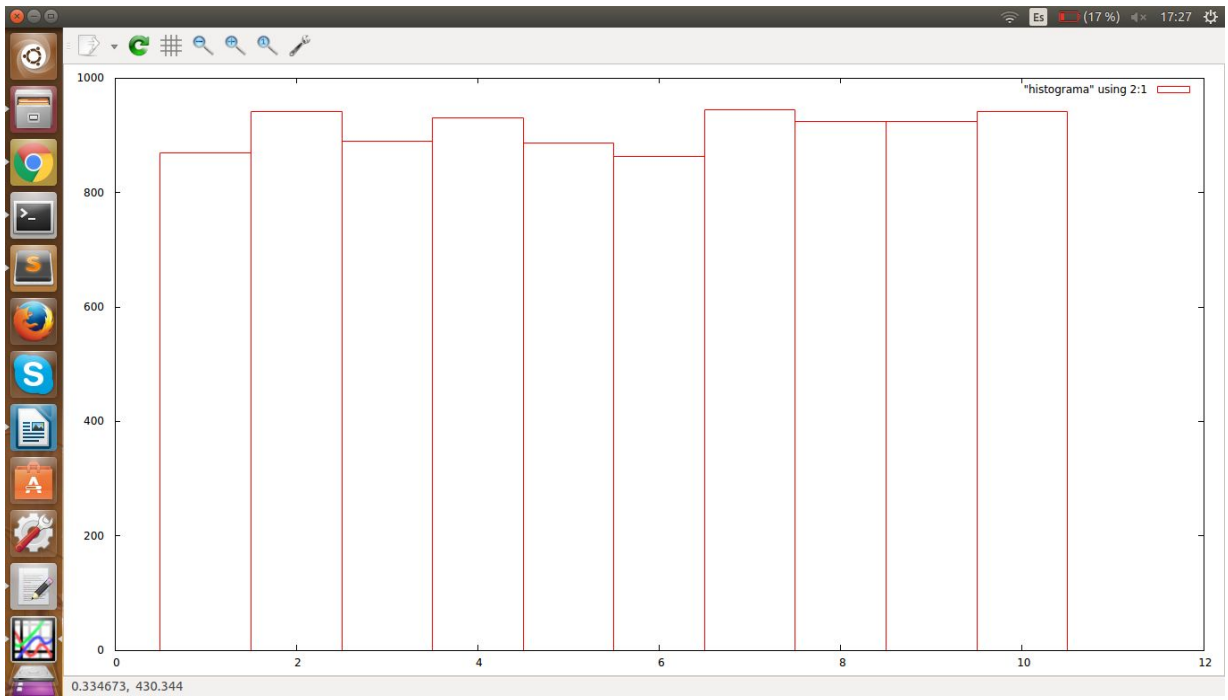
```
/*                                    */
/* Entrada:   0 < ip <= iu                    */
/* int *tabla: Tabla con los numeros             */
/* int ip: Direccion del primer elemento a ordenar */
/* int iu: Direccion del ultimo elemento a ordenar */
/*                                    */
/* Salida:                                */
/* cont: El numero de veces que se ejecuto la OB   */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                 */
/************************************************/
int InsertSortInv(int* tabla, int ip, int iu)
{
        int cont = 0;
        int i;
        if (tabla == NULL) return ERR;
        for (i = ip + 1; i <= iu; i++){
                int j = i-1;
                int aux = tabla[i];
                while (j >= ip && tabla[j] < aux){
                   tabla[j+1] = tabla[j];
                  cont++;
                   j--;
                   }
                cont++;
                tabla[j+1] = aux;
        }
        return cont;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

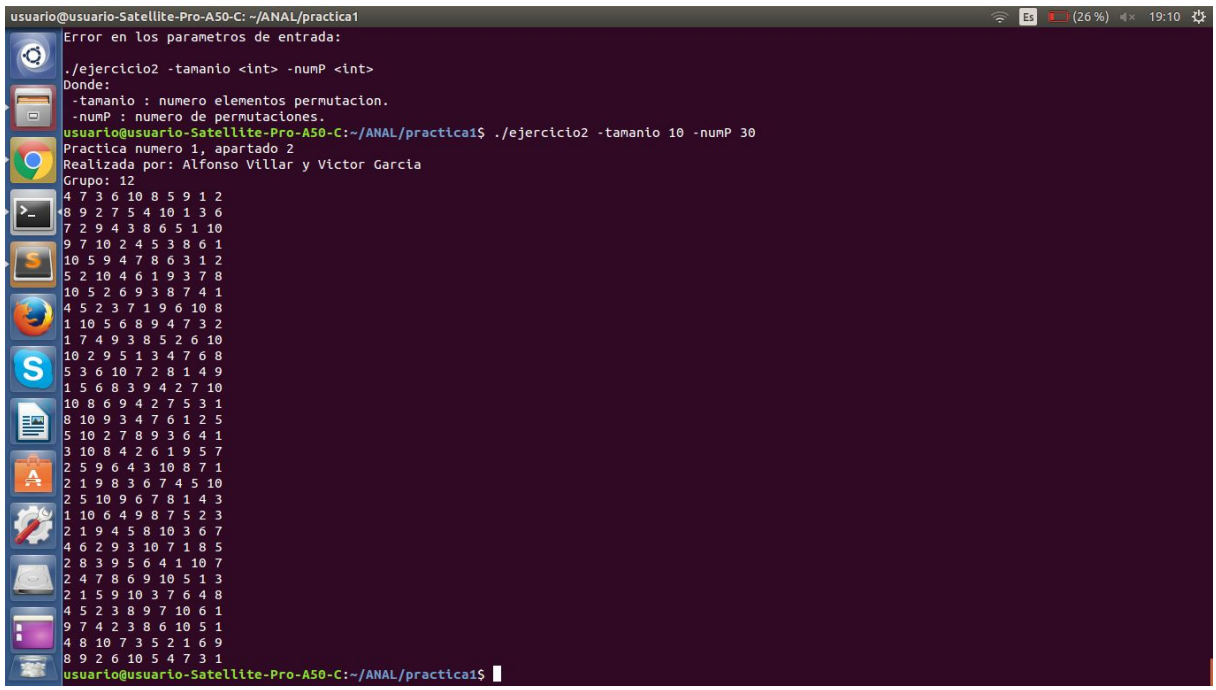


Esta es la salida en la terminal de la ejecución del primer ejercicio. Aparecen en la primera columna la frecuencia con la que aparece cada número, indicados en la segunda columna. Se han realizado 10000 generaciones de números aleatorios en el rango de números del 0 al 10.



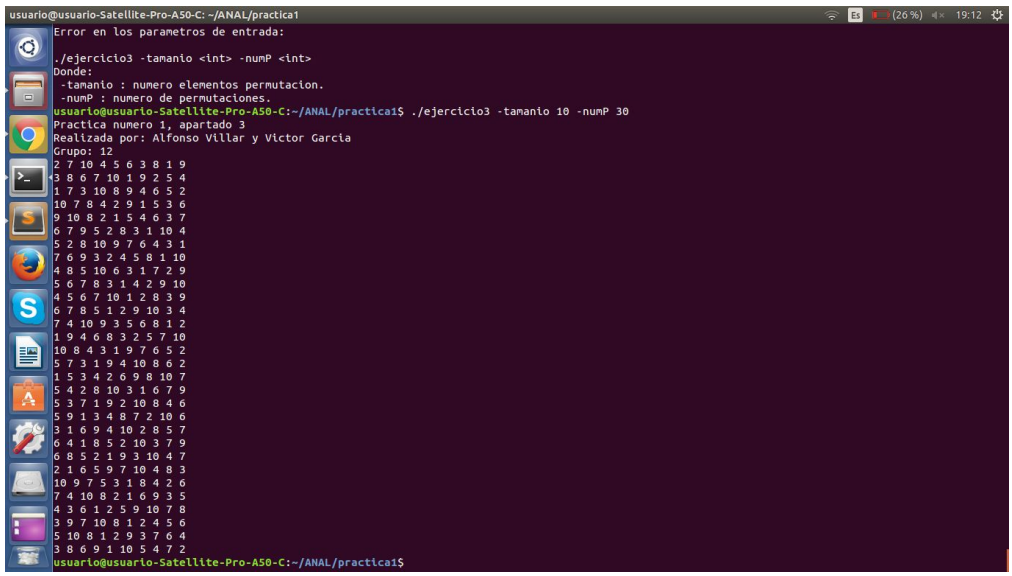
Este es el histograma elaborado a partir de los datos obtenidos en la ejecución anterior. Como se puede apreciar, se han generado 10000 números, y la frecuencia con la que cada número aleatorio en el rango de 0 a 10 aparece es bastante similar. Es decir, la función que genera los números aleatorios es equiprobable.

5.2 Apartado 2



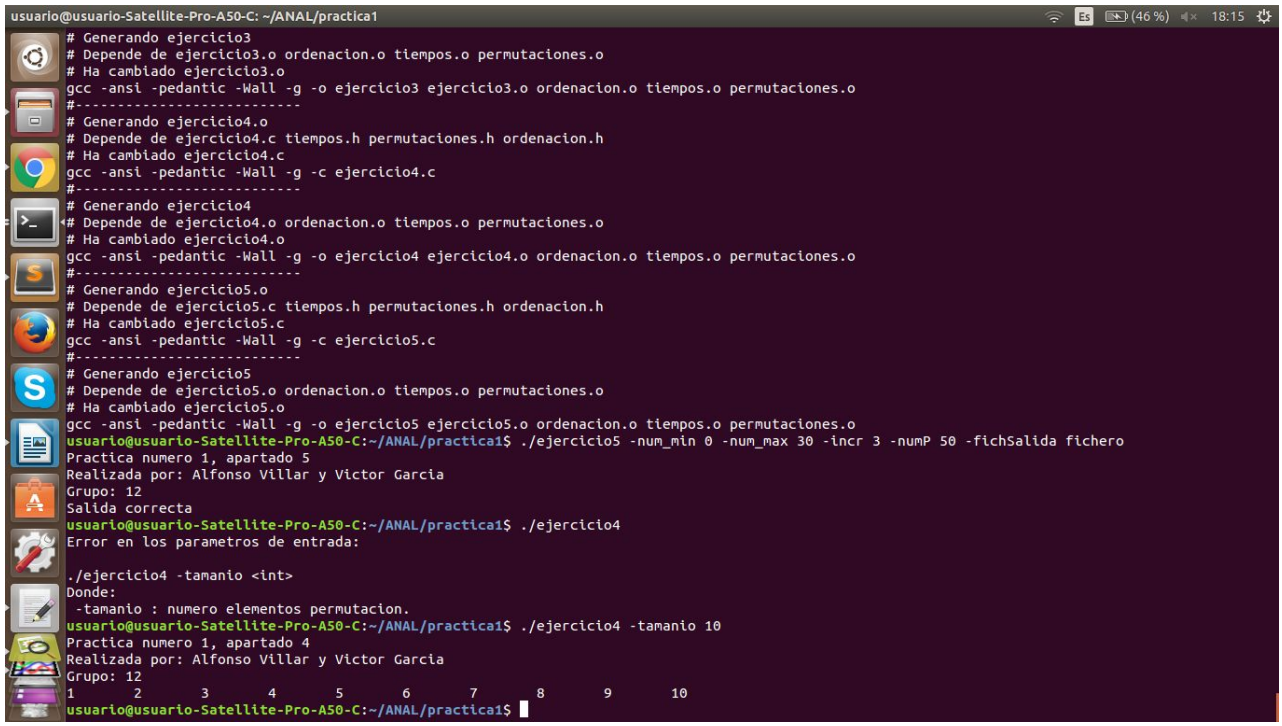
Esta es la salida en la terminal de la ejecución del segundo ejercicio. Aparecen 30 permutaciones de tamaño 10 aleatorias. En la imagen se puede apreciar que las permutaciones son distintas entre sí.

5.3 Apartado 3



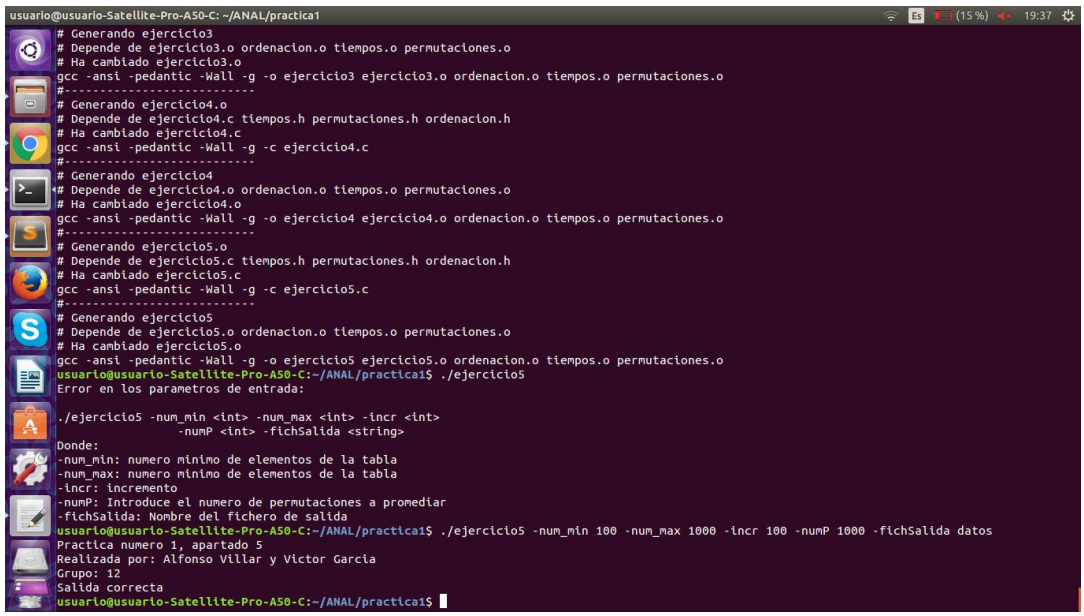
Esta es la salida en la terminal de la ejecución del tercer ejercicio. Aparecen 30 permutaciones de tamaño 10 aleatorias. En la imagen se puede apreciar que las permutaciones son distintas entre sí.

5.4 Apartado 4



Esta es la salida en la terminal de la ejecución del cuarto ejercicio, donde se genera una permutación aleatoria y mostramos cómo InsertSort devuelve la permutación ordenada.

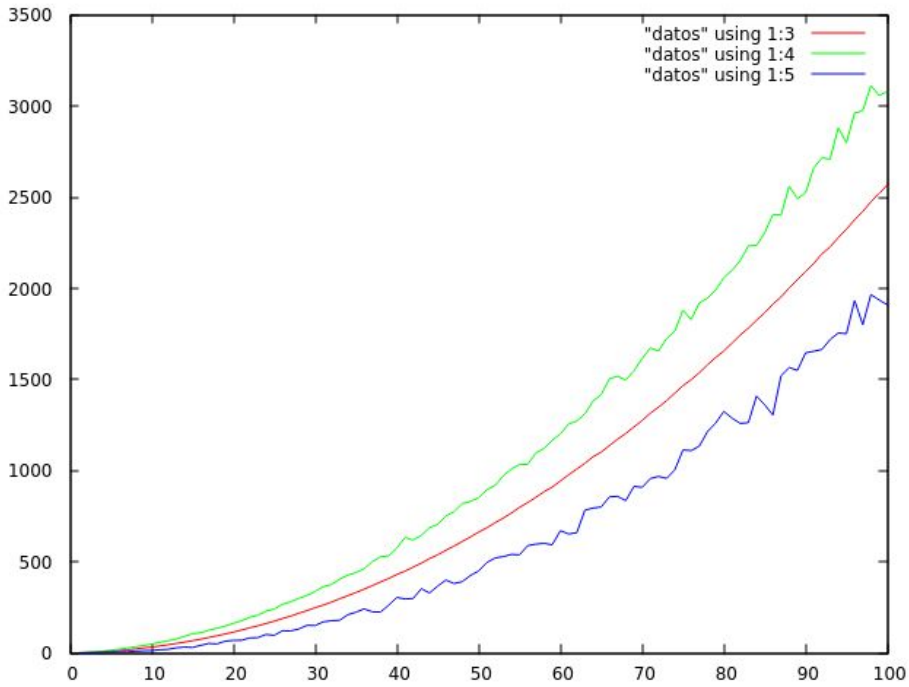
5.5 Apartado 5



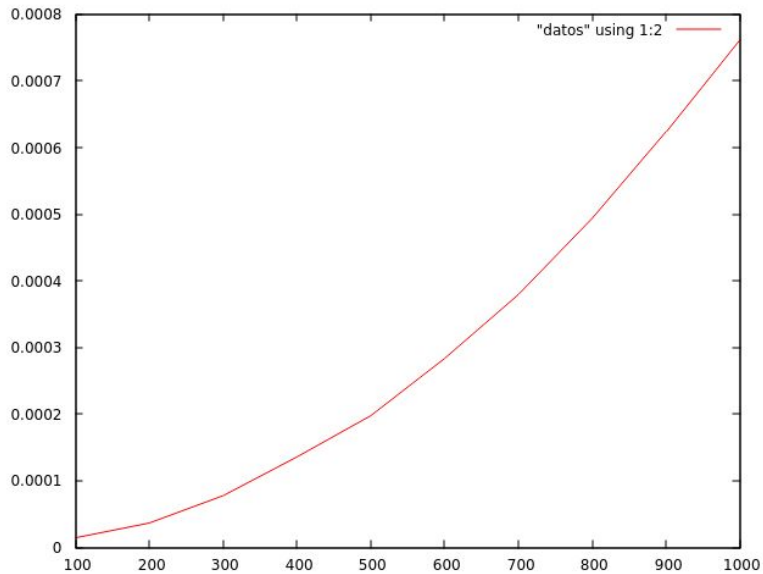
Esta es la salida en la terminal de la ejecución del quinto ejercicio. Empezando con permutaciones de tamaño 100 hasta 1000 con num_min 100, num_max 1000, incr 100 y numP 1000

Gráfica comparando los tiempos mejor, peor y medio en OBs para InsertSort

Aquí apreciamos con facilidad que la línea verde indica el tiempo peor, la roja el tiempo medio y la azul el tiempo mejor de las OBs.Se puede apreciar el que tiempo medio está en la mitad de los otros 2 como es lógico.

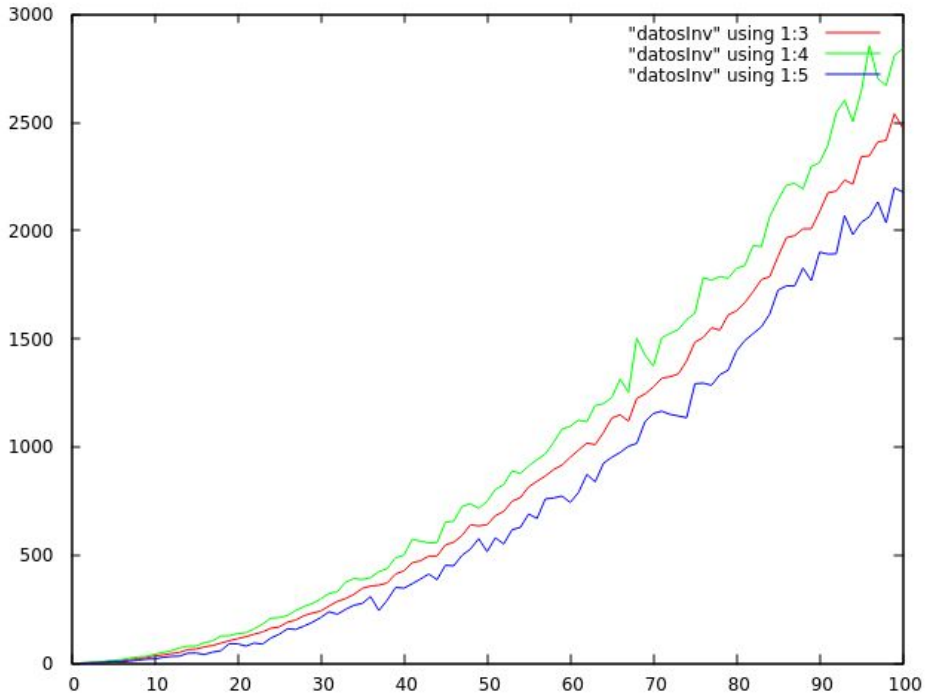


Gráfica con el tiempo medio de reloj para InsertSort

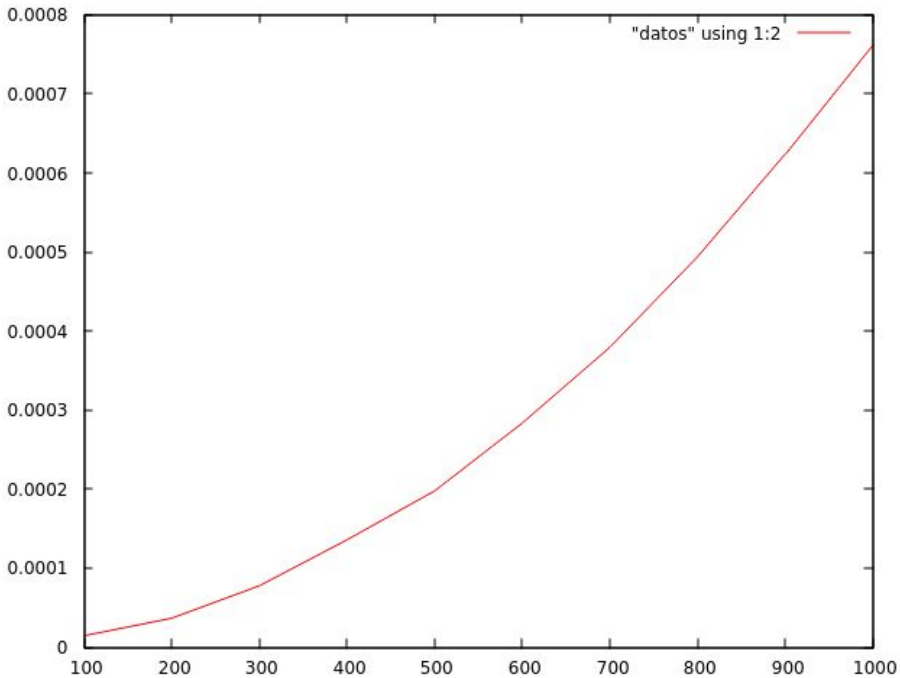


Gráfica comparando los tiempos mejor peor y medio en OBs para InsertSortInv

Se aprecia que la línea verde indica el peor tiempo, la roja el tiempo medio y la azul el mejor tiempo. Se puede apreciar el que tiempo medio está en la mitad de los otros 2 como es lógico.



Gráfica con el tiempo medio de reloj para InsertSortInv



6. Respuesta a las preguntas teóricas.

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

Hemos utilizado la función rand de stdlib como motor principal del generador de números aleatorios. A la hora de elegir qué algoritmo utilizar, nos hemos planteado 2, el que hemos implementado; más rápido pero con una ligera mayor probabilidad de que ciertos números salgan más veces pero despreciable a la hora de realizar esta práctica y otro en el que todos tienen la misma probabilidad de salir, pero más complejo y por tanto más lento, por lo cual elegimos el primero por el ahorro significativo de tiempo con respecto al anterior, sobretodo sabiendo la cantidad de veces que íbamos a ejecutar la función aleat_num.

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

Este algoritmo lo que hace es ordenar una tabla desde el comienzo, por lo que desde el segundo elemento a ordenar hasta el último, los va seleccionando y recolocando (si fuera necesario) a la posición en la que estaría ordenado con los elementos anteriores a su posición inicial, estando estos anteriores ya ordenados. Obteniendo así tras realizar la recolocación de todos los elementos de la tabla seleccionados menos el primero, una tabla perfectamente ordenada.

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

Es obvio, ya que como sería el único elemento de la tabla en la primera iteración no se puede ordenar, ya que siempre está ordenado en la única posición posible existente.

6.4. ¿Cuál es la operación básica de InsertSort?

Comprobar si el elemento de la tabla actualmente seleccionado es mayor que el de su izquierda.

6.5. Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WSS(n) y el caso mejor BSS(n) de InsertSort. Utilizar la notación asintótica (O, Θ, o, Ω,etc) siempre que se pueda. ¿Son estos valores los mismos que los del algoritmo InsertSortInv ?

En InsertSort, el caso peor WSS(N) ~ N(N-1)/2 y el caso mejor BSS(N) ~ N-1, que teóricamente coincide con el de InsertSortInv.

6.6. 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).

Teóricamente deberían tardar el mismo tiempo y así lo comprobamos al realizar las gráficas en la práctica, ya que lo único que cambia es el sentido de la ordenación en el algoritmo lo que no varía el tiempo de ejecución entre uno y otro.

7. Conclusiones finales.

Esta práctica nos ha servido para iniciarnos en esta nueva asignatura y nos ha ayudado a profundizar en los algoritmos trabajados en esta práctica y vistos en las clases de teoría, básicos en esta asignatura para así afianzarlos y no tener problemas con estos algoritmos a la hora de trabajar con ellos, así como enfocar un mismo problema desde distintos puntos como en el primer ejercicio que nos planteamos 2 algoritmos distintos para una misma finalidad, cada uno con sus ventajas e inconvenientes y saber cual implementar en cada ocasión y su porqué.