

Análisis de Algoritmos 2015/2016

Práctica 1

Litzy Tatiana Rochas Ávila y Lucía Colmenarejo Pérez, 1201.

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

1. Introducción.

Aquí ponéis una introducción y discusión previa a la práctica.

En esta primera práctica del curso, lo que vamos a realizar va a ser la implementación de varios ficheros con el fin de conseguir analizar un algoritmo. Para ello crearemos tres ficheros distintos cada uno de los cuales se dedicará a realizar una parte del análisis del algoritmo. Con ello se pretende afianzar los conceptos adquiridos en clase y ponerlos en práctica. Esto nos servirá de gran utilidad para medir el rendimiento(o tiempo de ejecución)de un algoritmo y las distintas formas de ordenación y sus respectivos tiempos de ejecución.

2. Objetivos

En esta sección explicaremos el trabajo realizado en cada un de los apartados o ejercicios.

En el primer apartado vamos a realizar una función que genere un número equiprobable entre dos límites(superior e inferior) que se pasan como argumentos de la función.

En el segundo apartado vamos a realizar una rutina que genere permutaciones aleatorias. Par ello utilizaremos la función implementada en el apartado 1 (aleat_num) para conseguir de forma aleatoria los números que conforman la permutación.

En el tercer apartado lo que llevaremos a cabo será un añadido de la función implementada en el apartado anterior, ya que lo que haremos será una función que genere un cierto numero de permutaciones de un cierto tamaño(ambos valores pasados como argumentos de la rutina).

En el cuarto apartado vamos a crear un nuevo fichero en el cual implementaremos en una función el algoritmo de ordenación Insert Sort visto en clase al cual le pasaremos como argumentos la tabla a ordenar, el primer elemento de la tabla y el último de la tabla.

En el quinto apartado haremos la implementación de tres funciones, todas relacionadas con el tiempo de ordenación dl algoritmo sobre el conjunto de permutaciones.

2.1 Apartado 1:

El objetivo de esta función será su posterior utilización en los siguientes apartados para generar una aleatoriedad de números y así poder recurrir a esta función siempre que queramos obtener números aleatorios.

2.2 Apartado 2:

El objetivo de este ejercicio es la implementación de una función que consiga obtener transposiciones aleatorias de unos ciertos números que se encuentren entre unos límites (los establecidos en la función de `aleat_num`) y de cierta longitud (que se pasa como argumento a la función implementada en este apartado). Para lograr este cometido necesitaremos la implementación de una función auxiliar, a la cual llamaremos `intercambiar`, con el fin de intercambiar números para conseguir todas las transposiciones posibles.

2.3 Apartado 3:

El objetivo de este tercer apartado es poder obtener el número de permutaciones que queramos con el tamaño que nosotros deseemos y que le pasemos como argumento de la nueva función.

2.4 Apartado 4:

El objetivo del cuarto apartado es que entendamos el algoritmo de ordenación de Insert Sort y seamos capaces de implementarlo. La finalidad de esta función es ordenar de forma creciente los números de la tabla dada a medida que vamos insertando sus números.

2.5 Apartado 5:

En este apartado el objetivo a conseguir es la implementación de una función que escriba en un fichero los tiempos medios y los números promedio, máximo y mínimo de veces que se ejecuta la operación básica (OB) en la ejecución del algoritmo de ordenación con un número de permutaciones definidos en un dominio dado (entre un máximo y un mínimo) y con un cierto incremento que nosotros definimos. Para ello Además haremos otra función con el fin de obtener el tiempo que tarda el algoritmo en ordenar la tabla. Además programaremos una función con la que se imprime una tabla con cinco columnas correspondientes al tamaño de la permutación, al tiempo de ejecución y al número promedio, máximo y mínimo de veces que se ejecuta la OB.

3. Herramientas y metodología

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

3.1 Apartado 1:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos). Además hemos empleado la herramienta de Gnuplot para la realización del histograma.

La solución que hemos propuesto para este apartado es la implementación de la función `aleat_num` utilizando la función `rand` para generar números aleatorios.

3.2 Apartado 2:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución adoptada para este apartado es la realización de dos bucles `for`; uno para crear el array de la permutación, y otro para dar la aleatoriedad y así poder crear todas la permutaciones posibles gracias a las llamadas realizadas a la función `aleat_num` implementada en el apartado anterior y a la función de intercambiar los números.

3.3 Apartado 3:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución por la que hemos optado en este problema es la realización de un bucle `for` con el objetivo de llamar a `genera_perm`(implementada en el apartado anterior) que nos generará las permutaciones de tamaño `n`(tamaño que le pasamos como argumento de la función).

3.4 Apartado 4:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución a la que hemos llegado en este problema es la realización de un bucle `for` en el cual recorreremos la tabla que nos dan y vamos insertando en una posición u otra teniendo en cuenta el bucle `while` que hay dentro de este(bucles anidados). En el bucle `for` cogemos la variable `'i'` inicializándola en la primera posición más uno(debido a que en el array se comienza en la posición 0) y siempre que sea menor o igual que el último elemento de la tabla, guardamos el valor en una variable `'a'` e incrementamos en uno el valor de `'i'`. Después con el bucle `while` lo que hacemos será comparar los elementos que hay insertados con el que se va a insertar para poder decidir en qué posición se inserta.

3.5 Apartado 5:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución que hemos implementado en este apartado es una función (genera_tiempos_ordenacion) que calcula el tiempo que tarda un algoritmo en ordenar permutaciones cuyo tamaño está definido entre dos números y se incrementa según un valor establecido y escribe el resultado en un archivo. Para ello nos vamos a valer de una función (tiempo_medio_ordenacion) que calcula el tiempo que tarda un algoritmo en ordenar una tabla. Y, por último, una función que escribe en un fichero los datos de los tiempos de ordenación(guarda_tabla_tiempos).

Cabe destacar que en todos los apartados hemos utilizado las herramientas de valgrind y gcc para la detección de errores y el compilado de los programas. Además en ciertos apartados hemos tenido que emplear el debbuger para encontrar el punto en el cual se encuentra el error y así poder arreglarlo, y, además nos ha ayudado bastante a la comprensión del funcionamiento del algoritmo.

4. Código fuente

Aquí ponemos el código fuente exclusivamente de las rutinas que hemos desarrollado nosotros en cada apartado.

4.1 Apartado 1:

```
int aleat_num(int inf, int sup) {  
  
    int r = 0;  
  
    if (inf < 0 || sup < 0) return -1; /*comprobamos que nos pasan correctamente los    argumentos*/  
  
    r = (rand() / ((double) RAND_MAX + 1.)) * (sup - inf + 1) + inf; /*utilizamos la    función rand para generar  
numeros aleatorios*/  
  
    return r;  
  
}
```

4.2 Apartado 2:

```

void intercambiar(int *a, int *b) { /*función privada que intercambia numeros*/

    int aux = 0;

    aux = *a;

    *a = *b;

    *b = aux;

}

int* genera_perm(int n) {

    int* permutacion = NULL;

    int i = 0;

    if (n < 1) return NULL; /*Comprobamos que nos pasan bien el tamaño de la permutación a generar*/

    permutacion = (int*) malloc((n + 1) * sizeof (int)); /*Reservamos memoria para la permutación*/

    if (!permutacion) return NULL; /*Comprobamos que la reserva se realizó correctamente*/

    for (i = 0; i < n; i++) {

        permutacion[i] = i + 1;

    }

    for (i = 0; i < n - 1; i++) {

        intercambiar(&(permutacion[i]), &(permutacion[aleat_num(i + 1, n - 1)])); /*Intercambiamos los números de el
array*/

    }

    return permutacion;

}

```

4.3 Apartado 3:

```

int** genera_permutaciones(int n_perms, int tamaño) {

    int** permu = NULL;

    int i = 0;

    if (n_perms < 1 || tamaño < 1) return NULL; /*Comprobamos que nos pasan correctamente los argumentos de
entrada*/

```

```

permu = (int **) malloc(n_perms * sizeof(int*)); /*Reservamos la memoria para el puntero a permutaciones*/

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

    (permu)[i] = (genera_perm(tamano)); /*llamamos a genera_perm que nos generará permutaciones de tamaño
tamano*/

}

return permu;

}

```

4.4 Apartado 4:

```

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

    int i = 0, j = 0, a = 0, contador = 0;

    if (!tabla || ip < 0 || iu < 0 || iu < ip) return -1;

    for (i = ip + 1; i <= iu; i++) {

        a = tabla[i];

        j = i - 1;

        while (j >= ip && tabla[j] > a) {

            contador++; /*Contador de operación basica*/

            tabla[j + 1] = tabla[j]; /*Intercambiamos los elementos*/

            j--;

        }

        tabla[j + 1] = a; /*Se asigna el número mayor*/

    }

    return contador;

}

```

4.5 Apartado 5:

```

short tiempo_medio_ordenacion(pfunc_ordena metodo, int n_perms, int tamano, PTIEMPO ptiempo) {

```

```

int ob[n_perms]; /*Array donde guardaremos el número de operaciones básicas para cada permutación*/

int i = 0, k = 0;

int **perm = NULL;

int j, medio = 0;

int minimo = INT_MAX;

int maximo = INT_MIN;

double tiempo;

clock_t t1, t2; /*Utilización del reloj para calcular el tiempo de ejecución*/

if (!metodo || n_perms < 1 || tamaño < 1 || !ptiempo) return ERR; /*Comprobamos que nos pasan correctamente los
argumentos de entrada*/

perm = genera_permutaciones(n_perms, tamaño); /*Generamos las permutaciones*/

t1 = clock(); /*Tiempo antes de empezar la ordenación*/

for (j = 0; j < n_perms; j++) {

    i = metodo(perm[j], 0, tamaño - 1); /*llamamos al algoritmo de ordenación*/

    ob[j] = i; /*Introducimos el numero de operaciones básicas en nuestro array de ob*/

    if (ob[j] > maximo) { /*Comprobamos si el numero de ob realizadas para la permutación es mayor que el
máximo actual*/

        maximo = ob[j];

    }

    if (ob[j] < minimo) { /*Comprobamos si el numero de ob realizadas para la permutación es menor que el
mínimo actual*/

        minimo = ob[j];

    }

}

t2 = clock(); /*Tiempo tras terminar la ordenación*/

tiempo = ((double) (t2 - t1)) / (n_perms * CLOCKS_PER_SEC); /*Tiempo promedio de ejecución*/

for (i = 0; i < j; i++) {

    k = k + ob[i];

} /*Sumamos los elementos de nuestro array de ob*/

medio = k / j; /*Calculamos el tiempo medio de ob realizadas*/

```



```

ptiempo->n_perms = n_perms;

ptiempo->tamano = tamano;

ptiempo->min_ob = minimo;

ptiempo->max_ob = maximo;

ptiempo->medio_ob = medio;

ptiempo->tiempo = tiempo;

for (i = 0; i < n_perms; i++) { /*Liberamos la memoria utilizada*/

    free(perm[i]);

}

free(perm);

return OK;

}

```

5. Resultados, Gráficas

Aquí vamos a poner los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

5.1 Apartado 1:

Resultados del apartado 1:

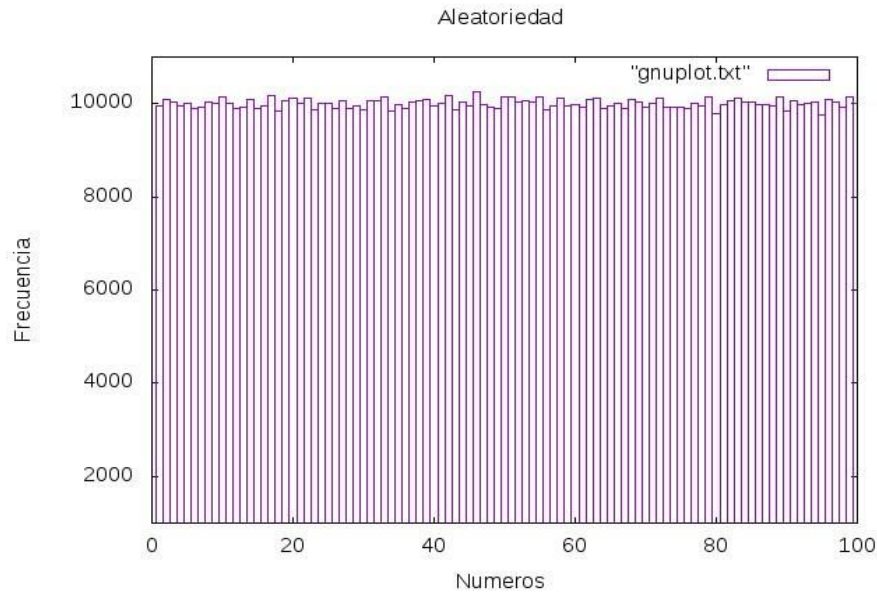
```

ibluue:~/workspace $ ./ejercicio1
Error en los parametros de entrada:

./ejercicio1 -limInf <int> -limSup <int> -numN <int>
Donde:
-limInf : Limite inferior.
-limSup : Limite superior.
-numN : Cantidad de numeros a generar.
ibluue:~/workspace $ ./ejercicio1 -limInf 2 -limSup 5 -numN 6
Practica numero 1, apartado 1
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
4
4
4
3
4
5

```

Gráfica del histograma de números aleatorios, comentarios a la gráfica:



Como podemos ver la generación de números es equiprobable para cada dígito, es decir, prácticamente se generan la misma cantidad de números para cada valor. Esta gráfica se obtiene de introducir como límite inferior 0 y límite superior 1000 con una cantidad de números a generar (numP).

5.2 Apartado 2:

Resultados del apartado 2:

```
ibluue:~/workspace $ ./ejercicio2
Error en los parametros de entrada:

./ejercicio2 -tamanho <int> -numP <int>
Donde:
-tamanho : numero elementos permutacion.
-numP : numero de permutaciones.
ibluue:~/workspace $ ./ejercicio2 -tamanho 6 -numP 7
Practica numero 1, apartado 2
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
5 1 4 2 6 3
3 4 6 1 2 5
6 5 4 1 3 2
6 3 5 2 1 4
4 3 1 5 6 2
5 3 1 6 4 2
6 5 4 2 1 3
```

5.3 Apartado 3:

Resultados del apartado 3:

```

ibluue:~/workspace $ ./ejercicio3
Error en los parametros de entrada:

./ejercicio3 -tamanio <int> -numP <int>
Donde:
-tamanio : numero elementos permutacion.
-numP : numero de permutaciones.
ibluue:~/workspace $ ./ejercicio3 -tamanio 5 -numP 7
Practica numero 1, apartado 3
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
5 4 2 1 3
4 3 5 2 1
5 1 4 2 3
5 4 2 1 3
2 4 1 5 3
3 5 4 2 1
2 5 4 1 3

```

5.4 Apartado 4:

Resultados del apartado 4:

```

ibluue:~/workspace $ ./ejercicio4
Error en los parametros de entrada:

./ejercicio4 -tamanio <int>
Donde:
-tamanio : numero elementos permutacion.
ibluue:~/workspace $ ./ejercicio4 -tamanio 7
Practica numero 1, apartado 4
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
7      6      5      4      3      2      1

```

5.5 Apartado 5:

Resultados del 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.

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

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

5. Respuesta a las preguntas teóricas:

Aquí responderemos a las preguntas teóricas que se nos han planteado en la práctica.

5.1 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.

5.2 Pregunta 2. Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el porqué ordena bien) del algoritmo InsertSort.

5.3 Pregunta 3. ¿Por qué el bucle de InsertSort no actúa sobre el primer elemento de la tabla?

5.4 Pregunta 4. ¿Cuál es la operación básica de InsertSort?

5.5 Pregunta 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 ?

5.6 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).

6. Conclusiones finales.

Discusión final sobre la práctica y los resultados obtenidos.