Análisis de Algoritmos 2017/2018

Práctica 1

Victoria Pelayo e Ignacio Rabuñal, grupo

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

# Introducción.

En esta práctica hemos implementado varias rutinas diferentes. En primer lugar, construímos las rutinas de generación de numeros aleatorios y permutaciones y comprobamos que tienen la aleatoreidad correcta. Después implementamos el algoritmo de ordenación BubbleSort y finalmente las funciones para generar y manejar los tiempos de ejecución pedidos.

# Objetivos

* 1. Apartado 1

Construir una rutina que genere un numero aleatorio equiprobable entre dos enteros definidos.

* 1. Apartado 2

Implementar la rutina de generación de permutaciones aleatorias a partir del pseudocódigo dado.

* 1. Apartado 3

Implementar la rutina que genera un numero de permutaciones equiprobables de un numero determinado de elementos (ambos pasados como argumento)

* 1. Apartado 4

Implementar una rutina para el algoritmo de ordenación BubbleSort.

* 1. Apartado 5

Implementar la estructura y rutinas para poder determinar el tiempo medio de ejecución y el numero medio, máximo y mínimo de veces que se ejecuta la operación básica del algoritmo anteriormente implementado.

Estos datos se guardarán en un fichero.

# Herramientas y metodología

En todos los apartados hemos utilizado el entorno Linux y las herramientas Gedit, gcc, Valgrind, Gnuplot, Sort, uniq y Make..

* 1. Apartado 1

Adaptamos la rutina rand para que generara los numeros aleatorios entre los enteros que deseemos. Comprobamos que funcionaba correctamente con el fichero ejercicio1.c y posteriormente cambiando los parametros de entrada por números más grandes para comprobar su correcta aleatoreidad. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

* 1. Apartado 2

Adaptamos el pseudocódigo dado en el ejercicio para crear la rutina de generación de permutaciones. Comprobamos que funcionaba correctamente con el fichero ejercicio2.c . Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

* 1. Apartado 3

Partiendo de la función creada en el apartado 2 creamos la rutina que genera las permutaciones que queramos del numero de elementos que queramos. Comprobamos que funcionaba correctamente con el fichero ejercicio3.c y posteriormente cambiando los parametros de entrada por números más grandes para comprobar su correcta aleatoreidad. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

* 1. Apartado 4

Partiendo del pseudocódigo que aparece en las transparencias de la parte teórica de la asignatura creamos la rutina que implementa el algoritmo de ordenación BubbleSort. Comprobamos que funcionaba correctamente ejecutando el fichero ejercicio4.c varias veces para así ver que la ordenación fuera siempre correcta. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

* 1. Apartado 5

Metodología y solución adoptada del apartado 5

# Código fuente

Código Fuente de las rutinas que hemos desarrollado en cada apartado.

* 1. Apartado 1

int aleat\_num(int inf, int sup){

return inf + (rand() % (sup - inf + 1));

}

* 1. Apartado 2

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++)

swap (&perm[i], &perm[aleat\_num(i, N - 1)]);

return perm;

}

* 1. Apartado 3

int\*\* genera\_permutaciones(int n\_perms, int N){ int i;

int \*\*perms=(int\*\*)malloc(n\_perms\*sizeof(perms[0])); if(perms == NULL) return NULL;

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

perms[i]=genera\_perm(N); if (perms[i]==NULL){

int j;

for (j=0;j<i;j++) free(perms[j]); free(perms);

}

}

return perms;

}

* 1. Apartado 4

int BubbleSort(int\* tabla, int ip, int iu){ int cont = 0;

int i;

int flag = 1;

i = iu;

if (tabla == NULL) return ERR;

while (flag == 1 && i >= ip + 1) { int j;

flag = 0;

for (j = ip; j <= i - 1; j++){ cont++;

if (tabla[j] > tabla[j+1]){

swap(&tabla[j], &tabla[j + 1]); flag = 1;

}

}

i--;

}

return cont;

}

* 1. Apartado 5

short tiempo\_medio\_ordenacion(pfunc\_ordena metodo,

int n\_perms, int N,

PTIEMPO ptiempo)

{

clock\_t ini,fn;

int\*\* perms= (int\*\*)genera\_permutaciones(n\_perms,N); int i;

if (perms == NULL) return ERR;

ptiempo->N = N;

ptiempo->min\_ob = INT\_MAX; ptiempo->min\_ob = INT\_MAX; ptiempo->max\_ob = 0;

ptiempo->medio\_ob = 0; ini = clock( );

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

int ob = metodo(perms[i], 0, N - 1); if(ob == ERR){

for(i=0; i < n\_perms; i++) free(perms[i]); free(perms);

return ERR;

}

if (ob < ptiempo->min\_ob) ptiempo->min\_ob = ob; if (ob > ptiempo->max\_ob) ptiempo->max\_ob = ob; ptiempo->medio\_ob += ob;

}

fn = clock();

ptiempo->medio\_ob /= n\_perms;

ptiempo->tiempo = (double)(fn - ini)/ n\_perms / CLOCKS\_PER\_SEC;

for(i=0; i < n\_perms; i++) free(perms[i]); free(perms);

return OK;

}

short genera\_tiempos\_ordenacion(pfunc\_ordena metodo, char\* fichero, int num\_min, int num\_max,

int incr, int n\_perms){

int n = ((num\_max - num\_min) / incr) + 1; int j, tamanio;

PTIEMPO tiempos = (PTIEMPO)malloc(n \* sizeof(tiempos[0]));

if(tiempos == NULL) free(tiempos);

for (j = 0,tamanio = num\_min; tamanio <= num\_max; j++, tamanio+=incr){

short codigo = tiempo\_medio\_ordenacion(metodo, n\_perms, tamanio, &tiempos[j]); fprintf(stderr, "%d\n", tamanio);

if (codigo == ERR){

free(tiempos); return ERR;

}

}

if (guarda\_tabla\_tiempos(fichero, tiempos, n) == ERR) return ERR; free(tiempos);

return OK;

}

short guarda\_tabla\_tiempos(char\* fichero, PTIEMPO tiempo, int n\_tiempos){ int i;

FILE \*f = fopen(fichero,"w"); if (f == NULL) return ERR;

for (i = 0; i < n\_tiempos; i++){ fprintf(f, "%d\t", tiempo[i].N);

fprintf(f, "%f\t", tiempo[i].tiempo); fprintf(f, "%f\t", tiempo[i].medio\_ob); fprintf(f, "%d\n", tiempo[i].max\_ob); fprintf(f, "%d\t", tiempo[i].min\_ob);

}

fclose(f); return OK;

}

# Resultados, Gráficas

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

* 1. Apartado 1

Resultados del apartado 1.

Gráfica del histograma de números aleatorios, comentarios a la gráfica

* 1. Apartado 2

Resultados del apartado 2.

* 1. Apartado 3

Resultados del apartado 3.

* 1. Apartado 4

Resultados del apartado 4.

* 1. Apartado 5

Resultados del apartado 5.

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

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

# Respuesta a las preguntas teóricas.

* 1. Pregunta 1

Utilizamos la función rand() que devuelve un número aleatorio entre 0 y RAND\_MAX. Dado que queremos un número entre el ínfimo y el supremo, este número será un número aleatorio en un intervalo de (sup-inf +1) número.

Primero hacemos rand() % (sup-inf+1), así obtenemos un número entre 0 y sup-inf.

Para obtener un número entre el ínfimo y el supremo a ese número obtenido hay que sumarle el ínfimo. Y este número finalmente será el que devuelva la función.

Un método alternativo podría ser:

int aleat\_num(int inf, int sup){

int num;

num = rand % sup;

while (num < inf)

num = rand() %sup;

return num;

}

Las desventajas de este otro método principalmente serían el tiempo de ejecución ya que si el intervalo entre el ínfimo y el supremo fuera muy pequeño podría tardarse bastante en encontrar un número aleatorio entre ellos.

La nuestra simplemente toma un número aleatorio y realiza una serie de operaciones para encontrar un número aleatorio en el intervalo que se nos pide.

* 1. Pregunta 2

El algoritmo Bubble Sort empieza ordenando desde el final. Dentro del bucle principal hay otro dentro que se encarga de ir comparando desde el primer element hasta en el que nos encontremos y ir realizando los swaps correspondientes.

Hay dos implementaciones una con dos bucles for, esta no la hemos realizado ya que se t arda lo mismo tanto con la table ordenada como desordenada.

La otra version, la que hemos implementado, tiene un flag, el bucle principal se realizará siempre que el flag esté a 1. Al principio se incializa a 1 para que entre en el bucle, una vez dentró s epodrá a 0, excepto si se produce algún swap, de esta manera si el resto de la tabla está ordenada el algortimo parará.

* 1. Pregunta 3

Porque no es necesario hacer una última interación ya que los dos últimos elementos en el peor de los casos se ordenan a la vez

* 1. Pregunta 4

La operación básica de BubbleSort es la comparación entre un element de la table y su consecutivo.

5.5 Pregunta 5

Caso peor: W(n) = O(n^2) Caso mejor: B(n) = O(n)

# Conclusiones finales.

Esta práctica, a parte de introducción para las siguientes, nos ha servido para familiarizarnos con el manejo de numeros aleatorios y la metodología para medir los tiempos de ejecución. Los resultados finales de los primeros apartados generaban grandes cantidades de numeros aleatorios correctamente y pudimos comprobar que nuestra implementación de BubbleSort cumplía con los tiempos de ejecución teóricos.

También nos ha servido para familiarizarnos con algunas herramientas como gnuplot, que no habíamos usado anteriormente.