Tema 6: Análisis de coste y trazas

1 Introducción.

Hasta ahora hemos aprendido a diseñar algoritmos para resolver determinados problemas, y como hemos podido ver, existen numerosas formas de realizar una misma tarea. En este tema veremos cómo medir la calidad de un programa desde el punto de vista del coste temporal del mismo. De esta forma, si tenemos dos algoritmos que son igualmente eficientes siempre nos quedaremos con el que mejor coste tenga (más rápido).

Por otro lado, veremos cómo analizar la ejecución de un programa para comprobar su eficacia sin necesidad de ejecutarlo, realizando lo que se denomina una **traza** del mismo.

2 Análisis de coste

El coste temporal de un programa es una medida del tiempo de ejecución del mismo independientemente de la máquina en la que se ejecute. El coste, por tanto, no se mide en segundos sino en lo que llamaremos **pasos**. Un paso es la unidad mínima de tiempo a nivel de programación. Para analizar el coste de un programa analizaremos el número de pasos de los que consta cada instrucción y los sumaremos.

Utilizaremos las siguientes reglas:

1) **Estructuras secuenciales**.

   A cada asignación u operación simple (aritmética, booleana, etc…) se le asigna **1 paso**. El número de pasos asociado a una secuencia de instrucciones se calcula sumando los pasos asociados a cada instrucción por separado.

2) **Estructuras de decisión**.

   En el caso de un condicional, el coste vendrá dado por el coste de la evaluación de la condición mas el coste del bloque asociado al peor caso.

condición + max(bloque_si, bloque_no)

3) **Estructuras de repetición**.

En el caso de un bucle vendrá dado por el número de veces que se ejecuta multiplicado por el coste del cuerpo del bucle mas el coste asignado a la evaluación de la condición del mismo:

Nºejecuciones (cuerpo + condicion) pasos.

En el caso de tener un bucle con condición inicial tendremos que sumar el coste de la evaluación de la condición una vez más (la última):

Nºejecuciones (cuerpo + condicion) + condicion pasos.

En el caso de bucles con contador, tendremos que tener en cuenta el incremento del contador en el cuerpo del bucle y la inicialización del mismo.

Siguiendo estas normas podemos etiquetar cada instrucción y estructura del programa de forma sencilla. El coste final del programa será la suma de los costes de todas las instrucciones y estructuras.

Veamos un primer ejemplo en el que vamos a evaluar el coste de una función sencilla:

```
void Funcion1(void) {
    int i,salida;  //La definición de variables no cuenta
    printf("Funcion sencilla\n");   // 1p
    acum=0;                         // 1p
    for(i=-10;i<10;i++){            // 1+20*(1+8+2)+1=222 p
        if( i>=0  && i<100){                    // 3+Max(5,4)=8p
            printf("Positivo\n");               // 1p
            salida=i*2+1;                       // 3 p
            printf("Numero: %d\n",salida);      // 1p
        }else{
            printf("Negativo\n");               // 1p
            salida=i*3;                         // 2p
            printf("Numero: %d\n",salida);      // 1p
        }
    }

    printf("Cuenta atrás:\n");      // 1p
    i=10;                           // 1p
    while(i>=0){                    // 1+11*(3+1)=45p
        printf("Numero:%d\n",i);    //1p
        i=i-1;                      //2p
    }
```

```
        printf("Cuenta alante:\n");      // 1p
        i=0;                             // 1p
        do{                              // 10*(3+1)=40p
                printf("Numero:%d\n",i);        //1p
                i=i+1;                   //2p
        }while(i<10);
        return;
}
```

El cose total de la función sería 1+1+222+1+1+45+1+1+40=313 pasos.

Como vemos, el resultado es un valor numérico que será el mismo para todas las ejecuciones posibles del algoritmo. En este caso diremos que el coste de la función es **constante**.

En la mayoría de las ocasiones no es posible calcular de forma absoluta el valor del coste de un algoritmo, puesto que el número de pasos depende del algún factor externo. En estos casos se suele calcular el coste en función de las variables implicadas.

Veamos un ejemplo del cálculo del coste para una función que calcula el factorial de un número **n** que se le pasa como parámetro:

```
int Factorial(int n) {
        int salida,int i;
        salida=1;       // 1p
        for(i=n;i>0;i--){          // 1+n*(1+2+2)+1= 2+5n p
                salida=salida*i;         // 2p
        }
        return(salida);
}
```

El coste de la función sería $3+5n$ pasos, que como vemos, depende de la variable de entrada n.

Lo que más importa en el cálculo del coste de un algoritmo es el orden de magnitud del mismo, esto es, su **complejidad**. En el ejemplo anterior, el polinomio que define el coste es de orden 1 (**lineal**). En este sentido, el peor coste vendrá definido por la peor complejidad, sea cual sea la magnitud de la expresión. Por ejemplo, el coste $3200+n^2$, frente al coste $100+(2n+3)^2$, poseen la misma complejidad (**cuadrática**). La expresión de coste n^3 tendría una complejidad mucho mayor (**cúbica**) que las dos anteriores.

Veamos un nuevo ejemplo en el que vamos a calcular la complejidad del algoritmo de ordenación por burbuja, estudiado en temas anteriores:

```
void OrdenacionBurbuja(Tipo_base vector[], int tam) {
    Tipo_base aux;
    int i,j;
    for(i=0;i<tam;i++){     //1+tam(1+16tam-14+2)+1 p
        for(j=1;j<tam;j++){   // 1+(tam-1)(1+13+2)+1 = 16tam-14p
            if(vector[j-1]>vector[j]){        // 4+9 = 13p
                aux=vector[j-1];          // 3p
                vector[j-1]=vector[j];    // 4p
                vector[j]=aux;            // 2p
            }
        }
    }
    return;
}
```

El coste del algoritmo sería $16tam^2-11tam+2$. Por tanto, su complejidad sería cuadrática. Dejamos como ejercicio que el alumno calcule el coste y la complejidad de los restantes algoritmos de ordenación, a fin de comparar los mismos desde un punto de vista temporal.

Como vemos únicamente se ha descrito cómo realizar el cálculo de la complejidad sobre algoritmos iterativos. Los algoritmos recursivos requieren de un tratamiento especial que se escapa a los contenidos de este curso.

3 Trazas

Hasta ahora, para comprobar el correcto funcionamiento de un algoritmo, no nos queda más remedio que ejecutarlo. Esta forma de comprobación no siempre es posible y además, no es la mejor, puesto que la información que podemos recoger sobre la evolución del algoritmo resulta muy reducida. Existen numerosos errores de programación que no afectan a la solución final ni se reflejan ejecutando simplemente el programa, y no por ello dejan de ser errores. Para comprobar el correcto funcionamiento de un programa es necesario realizar un conjunto de **trazas** significativas del mismo. Una traza es un seguimiento paso a paso de la evolución de las variables de un algoritmo para un determinado conjunto de datos de entrada.

Para construir una traza para un determinado algoritmo, utilizaremos una tabla en la que cada columna representa una variable del algoritmo y las filas representan la secuencia de instrucciones que se produce al ejecutarlo. En cada casilla de la tabla escribiremos el valor de cada variable después de ejecutar la instrucción.

En la traza introduciremos únicamente las variables que cambian de valor a lo largo del algoritmo (**variables significativas**) y las instrucciones que cambian dichas

variables (**instrucciones significativas**). Antes de realizar la traza leeremos el algoritmo y numeraremos las instrucciones significativas del mismo.

Veamos una primera traza para la función factorial escrita anteriormente:

```
-   int Factorial(int n) {
-           int salida, i;
1           salida=1;
2           for(i=n;i>0;i--){
3                   salida=salida*i;
-           }
-           return(salida);
-   }
```

Como vemos, hemos numerado únicamente las instrucciones significativas, que serán las únicas que aparezcan en la traza. Por otro lado, de todas las variables del programa representaremos únicamente *i* y *salida*, puesto que son las únicas que cambian (variables significativas). Veamos una traza para *n*=4:

	i	salida
1	?	1
2	4	1
3	4	4
2	3	4
3	3	12
2	2	12
3	2	24
2	1	24
3	1	24
2	0	24

Llegados a este punto, terminaría el bucle principal y por tanto, la función. Como vemos, *salida*=24 al terminar el programa, que coincide con *4*!.

Otra prueba significativa para este programa consistiría en realizar una traza para un número *k* igual o inferior a 0 (casos especiales de la función factorial). El resultado con estos datos de entrada sería:

	i	salida
1	?	1
2	k	1

Gracias a estas trazas nos podemos dar cuenta de que el algoritmo funciona correctamente para números mayores o iguales a 0, pero no se realiza ningún tratamiento sobre los números negativos (se toman como si fuesen igual a 0).

Por otro lado, se realiza una iteración innecesaria multiplicando por el valor 1 al final del cálculo, lo cual se refleja en la traza (instrucción significativa que no cambia los datos). Esto se puede arreglar modificando el bucle para que no llegue hasta ese cálculo:

```
int Factorial(int n) {
    int salida, i;
    salida=1;
    for(i=n;i>1;i--){
        salida=salida*i;
    }
    return(salida);
}
```

Veamos un ejemplo un poco más complicado:

```
-   void FuncionPrueba(int a, int b) {
-       int i,j,valor;
1       for(i=0;i<a;i++){
-           printf("Vuelta %d\n",i);
2           for(j=0;j<b;j++){
3               valor=(i+j)*2+1;
-               printf("con i=%d, j=%d vale %d\n",i,j,valor);
-           }
-       }
-       return;
-   }
```

Como vemos, se han numerado aquellas instrucciones que cambian los datos (instrucciones significativas). Por otro lado, las variables significativas (que cambian su valor a lo largo del algoritmo) son únicamente *i, j* y *valor*. El resto de variables (*a* y *b*) no son significativas puesto que no se tocan. Vamos a realizar una traza del algoritmo para los datos de entrada *a*=2 y *b*=3:

	i	**j**	**valor**
1	0	?	?
2	0	0	?
3	0	0	1
2	0	1	1
3	0	1	3
2	0	2	3
3	0	2	5
2	0	3	5
1	1	3	5
2	1	0	5
3	1	0	3
2	1	1	3
3	1	1	5
2	1	2	3
3	1	2	7
2	1	3	7
1	2	3	7

Como se puede comprobar, para poder realizar una traza correctamente, es necesario conocer de forma exacta el comportamiento de cada estructura.

Para realizar una traza de un algoritmo recursivo es necesario recurrir a procedimientos más informales que nos permitan estudiar el orden y la evolución de las llamadas (ver ejemplo del factorial en el tema correspondiente a programación modular y recursividad). El estudio de este tipo de trazas se escapa de los contenidos de este curso.