

Estructura de datos

Tema 1 - Función de eficiencia.

El proceso que vamos a realizar es extraer la función que marca la eficiencia del código de nuestro programa. Existe una función “timer” que me permite ver el tiempo que ha transcurrido en una sección del código que desee. ### Notación “O grande”. Se dice que una $f(n) = O(g(n))$ si a partir de un m , $f(n) \leq k \cdot g(n)$, donde k es una constante. Para usar este tipo de comparación podemos utilizar simplificación de funciones a su estructura básica. ### Hallar la función de eficiencia.

Lo primero es dividir el código en trozos, y tenemos en cuenta que si: 1. Los trozos son independientes, la función de la unión es O del máximo de las funciones de cada uno. 2. Los trozos son anidados o dependientes, la función de la unión es O del producto de las funciones de cada uno.

Teniendo en cuenta que cada acción cuenta como la unidad, es decir:

```
for(int i = 0; i < n; i++){
    cout << "Lmao";
}
```

Serían n iteraciones de valor 1. Sabiendo que la iteración $i++$ también es una operación pero no tiene sentido tenerlo en cuenta ya que $2n = O(n)$, de igual forma que la asignación del principio del bucle, y la comparación que se realiza, es decir, contaremos las operaciones simples como una única.

Análisis de sentencias

Bucles

Sentencias IF-ELSE

Es una sentencia en la que usamos la regla del máximo, ocurrirá el if o el else y el máximo de los 2 marca la eficiencia del bloque.

Puede ser que la condición no sea una sentencia elemental, entonces habría que tenerla en cuenta, pero si es una sentencia trivial, la sentencia será $O(1)$

```
if(A[0][0] == 0)
{
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
            A[i][j] = 0;
    }
}
```

```

    }
}
else
{
    for(k = 0; k < n; k++)
    {
        A[k][k] = 1;
    }
}

```

En este caso, tenemos que tomar máximo entre $O(n^2)$ (lo que cuesta el if) y $O(n)$ (lo que cuesta el else). Por ello, la eficiencia del código es $O(n^2)$.

Bloques de Sentencias

En este caso, si tenemos bloques independientes se va tomando la regla del máximo para todos los bloques

Llamadas a funciones

Hay que mirar cuánto cuesta la llamada a la función, es muy importante para la ejecución del código.

Analizamos un ejemplo de un código de 30 líneas de las transparencias del profesor.

Ejemplos

- Cuando en los bucles haya $i=2, i=4, i=n \dots$ entonces la eficiencia será logaritmo en base n de lo que haya dentro del bucle.

Tema 2 - Abstracción.

Uso de Template.

Nos permite seleccionar el tipo de dato que vamos a utilizar en tiempo de ejecución. Declarando:

```

template <class T, int n>

class array_n {
private:
    T items[n];
};

```

Creando un objeto de esta clase de la forma:

```
array_n<int,1000> w
```

Creando un metodo de la forma:

```
template <class T>
T VectorDinamico::componente(int i) const
{
    return datos[i];
}
```

La compilación a la hora de usar templates es distinta a la que estamos acostumbrados. En lugar de hacer un *#include "clase.h"* en el archivo *.cpp*, se incluirá el archivo *#include "clase.cpp"* al final del archivo *.h*.

Clase vector dinámico

Vamos a estudiar una clase que tenga un vector de datos y el número de elementos. Un primer ejemplo con tipo de dato float sería:

```
class VectorDinamico{

    float* datos;
    int ns;
}
```

Iteradores

Pretendemos hacer recorridos mucho más rápido. No volveremos a recorrer vectores haciendo *v[i]*. Usaremos los punteros para iterar, de la forma:

```
double *v = &a.
double *p;
double * fin;
fin = v+n;
for(p = v; p!= fin; ++p)
    cout << *p << endl;
```

Nota: en un compilador moderno, simplemente activando la optimización de código se consigue el mismo aumento en el rendimiento.

Definiremos incluso un nuevo tipo de dato llamado iterator, haciéndolo de la forma:

```

typedef double* iterator;
iterator begin(double* v, int n){
    return v;
}
iterator end(double* v, int n){
    return v+n;
}

/**-----*/
iterator p;
for(p=begin(v,n); p!=end(v,n);++p)
    cout << *p << endl;

```