UNIDAD I
Fundamentos de Análisis de Algoritmos

Programación III.

Fundamentos de Análisis de algoritmos

- Una vez se disponga de un algoritmo que funcione correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

Teoría de la Complejidad

- La complejidad de un algoritmo hace referencia a la cantidad de tiempo y espacio necesarios para ejecutar el algoritmo.
- Con la tecnología actual se puede decir que la memoria de las computadoras es abundante y barata, es por eso la complejidad del algoritmo se puede limitar al tiempo de ejecución del algoritmo.

- El uso eficiente de los recursos, suele medirse en función de dos parámetros:
  - el *espacio*, es decir, memoria que utiliza,
  - y el *tiempo*, lo que tarda en ejecutarse.

- El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: *los datos de entrada que se le suministren*, *la calidad del código generado por el compilador para crear el programa objeto*, *la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa,* y *la complejidad intrínseca del algoritmo.*

- Ambas medidas son importantes puesto que, si bien la primera ofrece estimaciones del comportamiento de los algoritmos de forma independiente de la computadora en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo.
- Estas medidas son *funciones temporales* de los datos de entrada.

TIEMPO DE EJECUCIÓN

- El tiempo de ejecución de un algoritmo es una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

- A la hora de medir el tiempo, siempre se hará en función del *número de operaciones elementales* que realiza dicho algoritmo, entendiendo por operaciones elementales (OE) aquellas que la computadora realiza en tiempo acotado por una constante.

Operaciones Elementales - OE

- Las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Ejemplo

```
int buscar(int& array, int c,int n)
{
  int j;
  j = 1;                          //1 = OE = 1
  while(a[j]<c) and (j<n))        //2 = OE = 4
  {
    j +=1;                        //3 =OE = 2
  }
  if(a[j]==c)                     //4 = OE = 2
     return j;                    //5 = OE = 1
  else
    return 0;                     //6=OE=1
}
```

Calcule el OE para cada línea de código

```
void burbujam(int v[],int n){
  int i,j=0,movimientos=0,aux;
  for(i=0;i<(n-1);i++)
  {
    for(j=1;j<n;j++)
    {
        if(v[j]>v[i-1])
        {
            aux=v[j];
            v[j]=v[j-1];
            v[i-1]=aux;
            movimientos++;
        }
    }
  }
  return movimientos;
}
```

Análisis: Mejor Escenario

- Se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan con "cortocircuito", es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (4) a (6). En consecuencia,

$$T(n)=1+2+3=6.$$

Análisis: Peor Escenario

- Se efectúa la línea (1), el bucle se repite $n-1$ veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle.

Análisis: Escenario Medio

- El bucle se ejecutará un número de veces entre 0 y $n-1$, y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder.
- Como existen n posibilidades (puede que el número buscado no esté) por tanto cada una tendrá una probabilidad asociada de $1/n$.

Asíntotas

- El análisis de la eficiencia algorítmica nos lleva a estudiar el comportamiento de los algoritmos frente a condiciones extremas. Matemáticamente hablando, cuando *N* tiende al infinito, es un comportamiento asintótico.
- Entiéndase N como el tamaño de entradas

Notación O (Omicrón, cota superior)

- Dada una función *f*, se quiere estudiar aquellas funciones *g* que a lo sumo crecen tan deprisa como *f*. Al conjunto de tales funciones se le llama cota superior de *f* y lo denominamos O(*f*). Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota

Orden de complejidad

O(1)	Orden constante
O(log n)	Orden logarítmico
O(n)	Orden lineal
O(n log n)	Orden cuasi-lineal
O(n2)	Orden cuadrático
O(n3)	Orden cúbico
O(na)	Orden polinómico
O(2n)	Orden exponencial
O(n!)	Orden factorial

Descripción de las OC

- O(1): Complejidad constante. Cuando las instrucciones se ejecutan una vez.
- O(*log n*): Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.
- O(*n*): Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la evaluación de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.
- O(*n log n*): Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si *n* se duplica, el tiempo de ejecución es ligeramente mayor del doble.

- O(n^2): Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces.
- O(n^3): Complejidad cúbica. Suele darse en bucles con triple anidación. Si *n* se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de n empieza a crecer dramáticamente.

- O(n^a): Complejidad polinómica ($a > 3$). Si a crece, la complejidad del programa es bastante mala.
- O($2n$): Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas. N

ESTIMACIÓN DE LA COMPLEJIDAD EN ALGORITMOS NO RECURSIVOS

- **Asignaciones y expresiones simples (=)**
- El tiempo de ejecución de toda instrucción de asignación simple, de la evaluación de una expresión formada por términos simples o de toda constante es ***O(1)***.

Secuencias de instrucciones (;)

- El tiempo de ejecución de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución individuales. Para una secuencia de dos instrucciones S1 y S2 el tiempo de ejecución esta dado por la suma de los tiempos de ejecución de S*1* y S*2:*
- T(S*1* ; S*2*) = *T(S1)* + *T(S2)*

Instrucciones condicionales (IF, SWITCH-CASE)

- El tiempo de ejecución para una instrucción condicional *IF-THEN* es el tiempo necesario para evaluar la condición, más el requerido para el conjunto de instrucciones que se ejecutan cuando se cumple la condición lógica.
- *T(IF-THEN) = T(condición) + T(rama THEN)*

- Aplicando la regla de la suma:
- O(*T(IF-THEN)*) = *max(O(T(condición), T(rama THEN)))*
- El tiempo de ejecución para una instrucción condicional de tipo *IF-THEN-ELSE* resulta de evaluar la condición, más el máximo valor del conjunto de instrucciones de las ramas *THEN* y *ELSE*.

- *T(IF-THEN-ELSE) = T(condición) + max(T(rama THEN), T(rama ELSE))*
- Aplicando la regla de la suma, su orden esta dada por la siguiente expresión:
- *O(T(IF-THEN-ELSE)) = O(T(condición)) + max(O(T(rama THEN)), O(T(rama ELSE)))*

- Aplicando la regla de la suma:
- *O(T(IF-THEN-ELSE)) = max(O(T(condición)), max(O(T(rama THEN)), O(T(rama ELSE))))*

El tiempo de ejecución de un condicional múltiple (*SWITCH-CASE*) es el tiempo necesario para evaluar la condición, más el mayor de los tiempos de las secuencias a ejecutar en cada valor condicional.

Instrucciones de iteración

- El tiempo de ejecución de un bucle *FOR* es el producto del número de iteraciones por la complejidad de las instrucciones del cuerpo del mismo bucle.
- Para los ciclos del tipo *WHILE-DO* y *DO-WHILE* se sigue la regla anterior, pero se considera la evaluación del número de iteraciones para el peor caso posible.

- Si existen ciclos anidados, se realiza el análisis de adentro hacia fuera, considerando el tiempo de ejecución de un ciclo interior y la suma del resto de proposiciones como el tiempo de ejecución de una iteración del ciclo exterior.

Llamadas a procedimientos

- El tiempo de ejecución esta dado por, el tiempo requerido para ejecutar el cuerpo del procedimiento llamado. Si un procedimiento hace llamadas a otros procedimientos "*no recursivos*", es posible calcular el tiempo de ejecución de cada procedimiento llamado, uno a la vez, partiendo de aquellos que no llaman a ninguno.

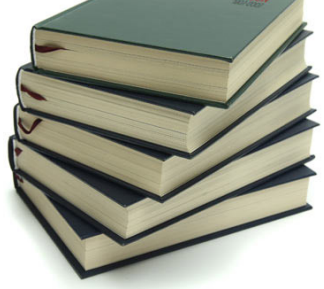
Ejemplo

```
int factorial(int n)//   O(1)
{
int fact = 1;    // O(1)
for(int i = n; i > 0; i--)    //O(n)
fact = fact * i;      //O(1)
return fact;      //O(1)
}
```

Su complejidad es lineal O(n), debido a que se tiene un bucle *FOR* cuyo número de iteraciones es *n*.

Ejercicio

```
int c = 1;
while(c < n)
{
 if(vector[c] < vector[n])
      {
      aux = vector[n];
      vector[n] = vector[c];
      vector[c] = aux;
      }
      c = c * 2;
}
```



Bibliografía

- http://www.lab.dit.upm.es/~lprg/material/apuntes/o/index.html
- http://www.monografias.com/trabajos27/complejidad-algoritmica/complejidad-algoritmica.shtml#orden
- http://www.algoritmia.net/articles.php?id=30
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/
- Fundamentos de Algoritmos – G. Brassard