Programación II

Bloque temático 1. Lenguajes de programación

Bloque temático 2. Metodología de programación

Bloque temático 3. Esquemas algorítmicos

Tema 4. Introducción a los Algoritmos

4.1 ¿Qué es un algoritmo?

Un algoritmo (de al-Jwarizmi, matemático árabe del siglo IX) es un *conjunto finito de instrucciones* o pasos que sirven para ejecutar una tarea o resolver un problema

Un algoritmo debe ser:

- ***Preciso*: cada paso a seguir tiene un orden**
- ***Finito*: tiene un determinado número de pasos, o sea, que tiene un fin**
- ***Definido*: si se sigue el mismo proceso más de una vez llegaremos al mismo resultado**

El primer algoritmo escrito para computador fue el creado en el siglo XIX por Ada Byron

- **cálculo de los números de Bernouilli para la máquina analítica de Charles Babbage**

Ejemplo de Algoritmos

Un problema se puede resolver utilizando diferentes algoritmos

Algoritmos para la multiplicación con lápiz y papel:

```
  357          357          27    357    357
 ×27          ×27           13    714    714
2499          714            6   1428
 714          2499            3   2856   2856
9639          9639            1   5712   5712
                                         9639
```

Clásico | Inglés | à la russe

Qué un algoritmo sea mejor que otro podría depender de:

- **el número de operaciones que haya que realizar**
- **de lo compleja que sea cada operación (multiplicación de números de una cifra, suma, división y multiplicación por 2, ...)**

Problemas y ejemplares

Los algoritmos propuestos para el *problema* de la multiplicación no sólo son aplicables al caso `357×27`

- **sino que constituyen una solución general**

El producto `357×27` es un *ejemplar* (o caso) del problema de la multiplicación de enteros positivos

- **otros ejemplares serían: `12430×841` o `3×8`**
- **cada ejemplar será más o menos difícil de resolver**

Los productos `46×7.2` o `-7×2` *no son* ejemplares del problema

- **cuando especificamos un problema hay que definir su *dominio***

Un algoritmo que pretende resolver un problema es *incorrecto* si

- **es posible encontrar un ejemplar dentro dominio para el que el algoritmo no produce una solución correcta**

4.2 Eficiencia de los algoritmos

Un algoritmo es más eficiente cuantos menos recursos consuma

- **Eficiencia espacial: cantidad de memoria extra requerida**
- **Eficiencia temporal: tiempo consumido**

La eficiencia de un algoritmo resolviendo un ejemplar depende de:

- **las características del propio algoritmo**
- **las características ejemplar a resolver**
  - **especialmente su tamaño**
- **la velocidad del computador**

Normalmente nos importaran los tiempos de peor caso:

- **`T(n)`: tiempo de peor caso para un ejemplar de tamaño n**
- **$T_{avg}(n)$: tiempo promedio para un ejemplar de tamaño n**

Tiempo de ejecución de un algoritmo

```
método busca(entero[1..n] a,entero e) retorna entero
  desde i=1 hasta n hacer
    si a[i] == e entonces
      retorna i
    fsi
  fdesde
  retorna -1
fmétodo
```

Suponemos que, en un ordenador dado, cada iteración del lazo tarda t unidades de tiempo

El tiempo de cómputo total T será:

- **mejor caso (e==a[1])** $\longrightarrow T=t$
- **caso promedio (e==a[n/2])** $\longrightarrow T_{avg}(n)=t\cdot n/2$
- **peor caso (e==a[n] $\vee$ e$\neq$a[i], i$\in$[1,n])** $\rightarrow T(n)=t\cdot n$

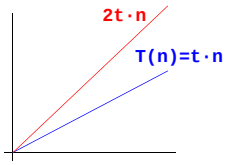
Notación asintótica

En el análisis y diseño de algoritmos nos importa el ritmo de crecimiento del tiempo de cómputo (no su valor exacto)

Para expresar los ritmos de crecimiento se usa la notación $O(n)$:

- **decimos que $T(n)$ es $O(f(n))$ si existen constantes c y n_0 tales que $T(n)\leq c\cdot f(n)$ para todo $n\geq n_0$**

La notación $O(n)$ muestra una cota superior al ritmo de crecimiento de un algoritmo

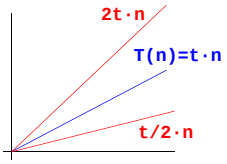


En el ejemplo anterior:

- **$T(n)=t\cdot n$, luego diremos que $T(n)$ es $O(n)$**
  - **ya que existen $c=2t$ y $n_0=0$ tales que $T(n)=t\cdot n \leq 2t\cdot n$ para todo $n\geq 0$**
- **esto se cumple independientemente del valor de t (es decir, independientemente de la velocidad del ordenador)**

La notación $\Omega(n)$ nos da una cota inferior al ritmo de crecimiento

- **decimos que $T(n)$ es $\Omega(f(n))$ si existen constantes c y n_0 tales que $T(n)\geq c\cdot f(n)$ para todo $n\geq n_0$**



Cuando el ritmo de crecimiento es a la vez $O(f(n))$ y $\Omega(f(n))$ se dice que es $\Theta(f(n))$

En el ejemplo anterior:

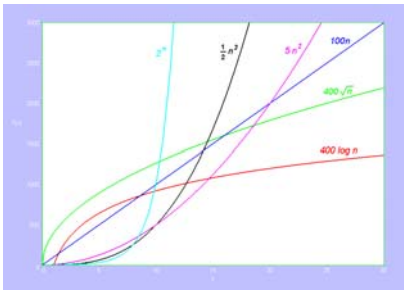
- **$T(n)=t\cdot n$, luego podremos decir que $T(n)$ es $\Omega(n)$**
  - **ya que existen $c=t/2$ y $n_0=0$ tales que $T(n)=t\cdot n \geq t/2\cdot n$ para todo $n\geq 0$**
- **por lo tanto en este ejemplo podemos decir que $T(n)$ es $\Theta(n)$**

La notación asintótica sólo proporciona límites a la velocidad de crecimiento, "ocultando" las posibles constantes multiplicativas:

- **$T_1(n)=n$ y $T_2(n)=1000000\cdot n$ son ambas $\Theta(n)$**

Ritmos de crecimiento más habituales

- O(1), o constante
- O(log(n)), o logarítmico
- O(n), o lineal
- O(n·log(n))
- O(n^2), o cuadrático
- O(n^x), o polinómico
- O(2^n), o exponencial



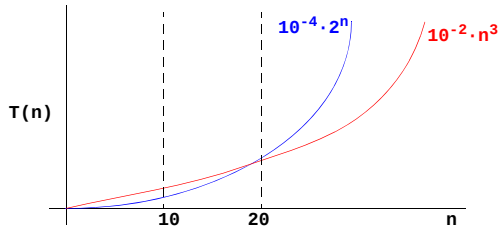
constante	log n	n	n log n	n^2	n^3	2^n
1	0	1	0	1	1	2
1	1	2	2	4	8	4
1	2	4	8	16	64	16
1	3	8	24	64	512	256
1	4	16	64	256	4096	65536
1	5	32	160	1024	32768	4294967296

Importancia de encontrar un algoritmo eficiente

El ritmo de crecimiento se acaba imponiendo para valores de n suficientemente grandes

T(n)	n=10	n=20	n=30	n=200
$10^{-4}\cdot 2^n$	0.1 seg	105 seg	un día	más que la edad del universo
$10^{-2}\cdot n^3$	10 seg	80 seg	4.5 minutos	un día

- En un día, un computador 100 veces más rápido utilizando el algoritmo $T(n)=10^{-4}\cdot 2^n$
  - ¡sólo podría resolver un ejemplar de tamaño n=37!



4.3 Un vistazo a la NP-completitud

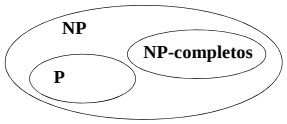
Los problemas se pueden agrupar en conjuntos:

- **NP** ⇨ problemas para los que la comprobación de que una solución es correcta puede realizarse en tiempo polinómico
  - p.e.: encontrar un subconjunto de suma 0 en un array
    (5,-3,4,-2,6) → (5,-3,-2)
  - comprobar que la solución es correcta es muy sencillo (O(n))
- **P** ⇨ problemas que pueden resolverse en tiempo polinómico
  - p.e.: buscar el máximo en un array (O(n))
  - $P \subseteq NP$ ya que para comprobar que una solución es correcta basta con volver a ejecutar el algoritmo (tiempo polinómico)
- **NP-completos** ⇨ problemas para los que "parece" que nunca se encontrará un algoritmo que los resuelva en tiempo polinómico
  - p.e.: encontrar un subconjunto de suma 0 en un array

Para los problemas `NP-completos` no se ha encontrado una solución mejor que recorrer (más o menos inteligentemente) todas las posibles soluciones

- **veremos muchos problemas NP-completos durante el curso:**
  - **mochila, coloreado de grafos, problema del viajante, ...**

Todo indica que la relación entre `P`, `NP` y `NP-completos` es:



- **La pregunta del millón de dólares [4] es: `¿P = NP`?**
- **está demostrado que si se consigue resolver un problema `NP-completo` en tiempo polinómico se podrían resolver todos**

4.4 Cálculo de tiempos de ejecución

- **Regla de las sumas:**
  - **si $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces**
  - **$T_1(n)+T_2(n)$ es $O(max(f(n),g(n)))$**

  Es decir: la ejecución de dos algoritmos que se realizan *uno después del otro* tiene un ritmo de crecimiento igual al del *máximo de los dos*

- **Regla de los productos:**
  - **si $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces**
  - **$T_1(n) \cdot T_2(n)$ es $O((f(n) \cdot g(n)))$**

  Es decir: la ejecución de dos algoritmos *anidados* uno dentro del otro tiene un ritmo de crecimiento igual al *producto de los ritmos de crecimiento de ambos*

- **Instrucciones simples (asignación y op. aritméticas): `O(1)`**
- **Secuencia de instrucciones simples: `O(max(1,1,1)) = O(1)`**
- **Instrucción condicional:**
  - **si es un "`if`" simple y no se conoce el valor de la condición, se supone que la parte condicional se ejecuta siempre**
  - **si es un "`if`" con parte "`else`" y no se conoce el valor de la condición, se elige la que más tarde de las dos partes**
- **Lazo: número de veces, por lo que tarden sus instrucciones**
- **Procedimientos recursivos: número de veces que se llama a sí mismo, por lo que tarda cada vez**

Ejemplos de calculo del ritmo de crecimiento

```
método transponeMatriz(real[1..n][1..n] m)
  desde i=1 hasta n hacer
    desde j=1 hasta n hacer
      int tmp=m[i][j]
      m[i][j]=m[j][i]
      m[j][i]=tmp
    fdesde
  fdesde
fmétodo
```

- El bucle externo se realiza n veces: $O(n)$
- El bucle interno se realiza n veces: $O(n)$
- las operaciones realizadas en el bucle interno son todas $O(1)$

$O(n)\cdot O(n)\cdot O(max(1,1,1))$

$O(n^2)$


```
int binarySearch(int key, int[] a){
  int mid, bottom= 0;
  int top=a.length-1;
  while (bottom<=top) {
    mid=(top+bottom)/2;
    if (a[mid]==key) {
      return mid;
    } else {
      if (a[mid]<key)
        bottom =mid+1;
      else
        top=mid-1;
    }
  }
  return -1;
}
```

- El peor de los casos es cuando el elemento buscado es el último o el primero
- Cada pasada del lazo se divide entre dos el tamaño del array
  - en el peor caso son necesarias $\log_2 n$ divisiones

$O(\log n)$

da igual decir $O(\log n)$ que $O(\log_2 n)$ ya que corresponden al mismo ritmo de crecimiento puesto que sólo se diferencian en una constante

$$\log_2 n = \frac{\log n}{\log 2}$$

4.5 Diseño de algoritmos

El diseño de un algoritmo que resuelva un problema es, en general, una tarea difícil

- **puede facilitarse recurriendo a esquemas muy generales que pueden adaptarse a cada problema particular**

Algunos esquemas algorítmicos:

- **Fuerza bruta (*Brute force*)**
- **Divide y vencerás (*Divide and Conquer*)**
- **Algoritmos voraces (*Greedy*)**
- **Programación dinámica (*Dynamic Programming*)**
- **Vuelta atrás (*Backtracking*)**
- **Ramificación y poda (*Branch & Bound*)**
- **Algoritmos Genéticos**

4.6 Bibliografía

[1] **Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.**

[2] **Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.**

[3] **Entrada en la *Wikipedia* para "algoritmo"**

[4] **The Millennium Prize Problems http://www.claymath.org/millennium/**