Tema 3 Análisis básico de algoritmos

ESTRATEGIAS DE PROGRAMACION Y ESTRUCTURAS DE DATOS

CA Guadalajara (UNED)

Análisis básico de algoritmos

Un **algoritmo** describe, de forma precisa, los **pasos** a seguir para alcanzar la solución de un problema.

Cualquier tipo de **algoritmo** debe disponer de 3 características:

- **Precisión**: Un algoritmo debe expresarse sin ambigüedad.
- **Determinismo**: Todo algoritmo debe responder del mismo modo antes las mismas condiciones.
- **Finito**: La descripción de un algoritmo debe ser finita.

Además un algoritmo debe ser **eficiente: consumo de los mínimos recursos** en **tiempo empleado**, espacio en memoria y consumo de procesadores – CPU.

Análisis básico de algoritmos

La **complejidad algorítmica** representa la cantidad de recursos que necesita un **algoritmo para resolver un problema** y, por tanto, permite determinar la **eficiencia** de dicho algoritmo.

A la hora de evaluar el **coste/tiempo** del algoritmo, tendremos tres opciones, además el **tiempo** requerido por un algoritmo **en función del tamaño de los datos de entrada:**

– **El coste esperado o promedio**. Nos tendremos que apoyar en una distribución estadística de los datos de entrada y estudiamos aquel que es más probable.

– **El coste mejor**. Se estudia el tiempo empleado en la ejecución cuando los datos de entrada son óptimos.

– **El coste peor.** Aquella ejecución del algoritmo en la que los datos de entrada hacen que el algoritmo vaya a emplear **más tiempo** en su ejecución. -> ***Coste asintótico temporal en el caso peor***

Ejemplo: **Ordenación de un array (desordenado)**

Análisis básico de algoritmos

Pero ¿cómo vamos a analizar el coste?.

- Para ello, se utilizarán métricas de análisis, donde podremos clasificar las funciones en distintas familias.
- Además, de **reglas prácticas** (cuando tengáis experiencia, con ver un algoritmo sabréis casi analizar el coste) para el análisis de dos tipos de algoritmos:

  - **Programas Iterativos**
  - **Programas Recursivos**

Métricas y Órdenes

La **Notación de Landau**, notación de orden, es una forma de expresar el comportamiento **asintótico de una función matemática**. Asumimos que la función matemática en nuestro caso será un algoritmo.

La notación de Landau se representa mediante el **símbolo "O"**, que significa "del mismo orden que" o "en el peor de los casos". Por ejemplo, si tenemos una función f(x) y queremos describir su comportamiento asintótico a medida que x tiende a infinito, podemos decir que f(x) es del mismo orden que g(x), y escribirlo como:

f(x) = O(g(x)) cuando x tiende a infinito.

Esto significa que, en el peor de los casos, f(x) crece tan rápido como g(x) a medida que x tiende a infinito. En otras palabras, g(x) actúa como una cota superior para f(x) a medida que x crece.

Métricas y Órdenes

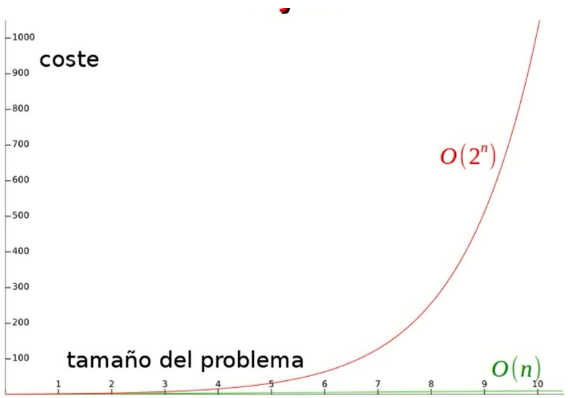
La notación de Landau también se utiliza en otras formas, como "omega" (Ω) y "theta" (Θ), que se utilizan para describir límites inferiores y exactos, respectivamente, en el comportamiento asintótico de una función.

Función	Orden de Crecimiento
1	O(1)
log n	O(log n)
√n	O(√n)
n	O(n)
n log n	O(n log n)
n^2	O(n^2)
n^3	O(n^3)
2^n	O(2^n)
n!	O(n!)

Métricas y Órdenes

```
F1 (n) {
  Si n < 2 → 1
  Si n >= 2 → F1(n-1)+F1(n-2)
}
```

$O(2^n)$




```
F2 (n) {
  → F2aux (n,1,1)
}
```

```
F2aux (x,y,z) {
  Si x == 0 → z
  Si x == 1 → y
  Si x >= 2 → F2aux(x-1,y+z,y)
}
```

$O(n)$

Reglas algoritmos iterativos

```
void ordenar (T[] v) {
  for (int i=2 ; i <= v.length ; i++) {
    int p=i; int x=v[i]; boolean seguir=True;
    while (p>1 and seguir) {
      if (x<v[p-1]) {v[p]=v[p-1]}
      else {seguir=False}
      p=p-1;
    }
    v[p]=x;
  }
}
```

Primera Regla: Determinar el tamaño del problema: longitud de V (n).

Segunda Regla: Calcular el coste de cada instrucción.

Reglas algoritmos iterativos

Operaciones básicas: Son aquellas que no dependen del tamaño del problema, típicamente las instrucciones de tipo E/S, asignaciones, expresiones, etc. y el coste asociado será constante, es decir O(1)

```
void ordenar (T[] v) {
  for (int i=2 ; i <= v.length ; i++) {
    int p=i; int x=v[i]; boolean seguir=True;
    while (p>1 and seguir) {
      if (x<v[p-1]) {v[p]=v[p-1]}
      else {seguir=False}
      p=p-1;
    }
    v[p]=x;
  }
```

Composición secuencial de sentencia: Varias sentencias que se ejecutan secuencialmente una tras otra.
Para **calcular el coste total**, se calcula el coste de cada instrucción y se suman:

$$\max_i \{O(c_{Si})\}$$

Reglas algoritmos iterativos

Operaciones condicionales:

```
if (e) {S1} else {S2}
```

- Equivale a:
  - Calcular e y ejecutar S_1 (si e es cierto)
  - Calcular e y ejecutar S_2 (si e es falso)
- Composición secuencial `e;S1;` ó `e;S2;`

max{O(ce),O(cS1),O(cS2)}

```
switch (e) {
  case v1 : S1;      e;S1;
  case v2 : S2;      e;S2;
  ...
  case vn : Sn;      e;Sn;
}
```

max{O(ce),O(cS1),O(cS2),…,O(cSn)}

Reglas algoritmos iterativos

Estructuras repetitivas, BUCLES:

```
for (ini;e;inc) {S}
```

- Ejecutar `ini`
- Para cada vuelta del bucle (v(n) vueltas):
  - Calcular e
  - Si no es falsa → ejecutar `S;inc;`
  - Si es falsa → salir

$\max\{O(c_{ini}), O(v(n)) \cdot \max\{O(c_e), O(c_S), O(c_{inc})\}\}$

Producto de órdenes.

$O(f) \cdot O(g) = O(f \cdot g)$

$O(n) \cdot O(n) = O(n \cdot n) = O(n2)$

Reglas algoritmos iterativos

Estructuras repetitivas, BUCLES:

```
while (e) {S}
```

- Para cada vuelta del bucle (v(n) vueltas):
  - Calcular e
  - Si no es falsa → ejecutar `S;`
  - Si es falsa → salir

$$O(v(n)) \cdot \max\{O(c_e), O(c_S)\}$$

Reglas algoritmos iterativos

Estructuras repetitivas, BUCLES:

```
do {S} while (e)
```

- Ejecutar S;
- Para cada vuelta del bucle (v(n) vueltas):
  - Calcular e
  - Si no es falsa → ejecutar S;
  - Si es falsa → salir

$$O(v(n)) \cdot \max\{O(c_e), O(c_S)\}$$

Reglas algoritmos iterativos

```
while (p>1 and seguir) {
  if (x<v[p-1]) {v[p]=v[p-1]}
  else {seguir=False}
  p=p-1;
}
```

$O(n)$

```
for (int i=2 ; i <= v.length ; i++) {
  int p=i; int x=v[i]; boolean seguir=True;
  while (p>1 and seguir) {
    if (x<v[p-1]) {v[p]=v[p-1]}
    else {seguir=False}
    p=p-1;
  }
  v[p]=x;
}
```

$O(n^2)$

Reglas algoritmos iterativos

```
void ordenar (T[] v) {
  for (int i=2 ; i <= v.length ; i++) {
    int p=i; int x=v[i]; boolean seguir=True;
    while (p>1 and seguir) {
      if (x<v[p-1]) {v[p]=v[p-1]}
      else {seguir=False}
      p=p-1;
    }
    v[p]=x;
  }
}
```

$O(n^2)$

Reglas algoritmos iterativos

Llamadas a subprogramas

$$f(e_1, e_2, \ldots, e_n)$$

- Calcular $e_1, e_2, \ldots, e_n$
- Ejecutar el cuerpo del subprograma f

$$\max\{O(c_{e1}), O(c_{e2}), \ldots, O(c_{en}), O(c_f)\}$$

Reglas algoritmos recursivos

```
F1 (n) {
  Si n < 2 → 1
  Si n >= 2 → F1(n-1)+F1(n-2)
}
```

Se va a determinar el **orden de complejidad** de dos familias recursivas:

1.- Reducción mediante sustración.

2.- Reducción mediante división.

```
F2 (n) {                F2aux (x,y,z) {
  → F2aux (n,1,1)         Si x == 0 → z
}                         Si x == 1 → y
                          Si x >= 2 → F2aux(x-1,y+z,y)
                        }
```

Reglas algoritmos Recursivos

Reducción mediante sustracción

$$T(n) \in \begin{cases} O(n \cdot c_{nr}(n) + c_b(n)) & si\ a = 1 \\ O(a^{(n \,\mathbf{div}\, b)} \cdot (c_{nr}(n) + c_b(n))) & si\ a > 1 \end{cases}$$

Cuando sólo hay una llamada recursiva, el coste de nuestra función estará en el orden de sumar n por el coste de las operaciones no recursivas más el coste de los casos base.
En el caso de varias llamadas recursivas, el orden será a elevado a n div b y para cada llamada se ejecutarán las operaciones no recursivas más el coste del caso base.

Reglas algoritmos Recursivos

Reducción mediante sustracción

```
F1 (n) {
  Si n < 2 → 1
  Si n >= 2 → F1(n-1)+F1(n-2)
}
```

$$T(n) \in \begin{cases} O(n \cdot c_{nr}(n) + c_b(n)) & si\, a=1 \\ O(a^{(n \,\mathbf{div}\, b)} \cdot (c_{nr}(n) + c_b(n))) & si\, a>1 \end{cases}$$

$c_b(n) \in ?$

$c_{nr}(n) \in ?$

$a = ?$

$b = ?$

```
F1 (n) {
  Si n < 2 → 1
  Si n >= 2 → F1(n-1)+F1(n-2)
}
```

$c_b(n) \in O(1)$

$c_{nr}(n) \in O(1)$

$a = 2$

$b = 1$

$O(2^n)$

Reglas algoritmos Recursivos

Reducción mediante sustracción

```
F2 (n) { → F2aux(n,1,1) }
F2aux (x,y,z) {
  Si x == 0 → z
  Si x == 1 → y
  Si x >= 2 → F2aux(x-1,y+z,y)
}
```

$$T(n) \in \begin{cases} O(n \cdot c_{nr}(n) + c_b(n)) & si\, a=1 \\ O(a^{(n \operatorname{div} b)} \cdot (c_{nr}(n) + c_b(n))) & si\, a>1 \end{cases}$$

$$c_b(n) \in ?$$
$$c_{nr}(n) \in ?$$
$$a = ?$$
$$b = ?$$
$$n = ?$$

$$c_b(n) \in O(1)$$
$$c_{nr}(n) \in O(1)$$
$$a = 1$$
$$b = 1$$
$$n = x$$

$$O(n)$$

Reglas algoritmos Recursivos

Reducción mediante división

$$T(n) \in \begin{cases} O\left(\log_b(n) \cdot c_{nr}(n) + c_b(n)\right) & si\ a = 1 \\ O\left(n^{\log_b(a)} \cdot \left(c_{nr}(n) + c_b(n)\right)\right) & si\ a > 1 \end{cases}$$

```
adivina (min,max) {
  med = (min+max) div 2;
  Si Correcto(med) → med
  Si MayorQue(med) → adivina(med+1,max)
  Si MenorQue(med) → adivina(min,med-1)
}
```

$O(\log_2(n))$

$c_b(n) \in ?$	$c_b(n) \in O(1)$
$c_{nr}(n) \in ?$	$c_{nr}(n) \in O(1)$
$a = ?$	$a = 1$
$b = ?$	$b = 2$
$n = ?$	$n = max - min + 1$

Tema 3 Análisis básico de algoritmos

ESTRATEGIAS DE PROGRAMACION Y ESTRUCTURAS DE DATOS

CA Guadalajara (UNED)