Progettazione di Algoritmi **Anno Accademico 2023–2024**

Lezione 5

Ugo Vaccaro

In questa lezione parleremo di *Analisi di Algoritmi*. Con il termine Analisi di Algoritmi intendiamo il calcolo delle risorse usate da algoritmi per risolvere un dato problema. Per *risorse* intendiamo (generalmnete) il *tempo impiegato dall'algoritmo* e la *quantità di memoria usata dall'algoritmo*. Per la maggior parte dei casi, analizzeremo gli algoritmi in termini del tempo di esecuzione da loro impiegato per produrre l'output richiesto.

Il tempo di esecuzione dipende da svariati fattori, ad esempio:

- Dall'abilità del programmatore e dal linguaggio di programmazione usato
- Dal linguaggio macchina del computer
- Dal codice che il compilatore genera
- Dalla taglia del problema (=la "dimensione" dell' input)
- Da come é fatto l'input (ad es., ordinato/non-ordinato)
- Dal numero di operazioni elementari eseguite dall'algoritmo

Non tutti questi fattori sono precisamente quantificabili, per cui valuteremo le risorse usate da un algoritmo a meno di una costante moltiplicativa (useremo quindi le notazioni asintotiche O, Θ e Ω). E perchè? Perchè le costanti moltiplicative dipendenti dai primi 3 aspetti prima enunciati non sono a priori esattamente valutabili, in quanto dipendono da aspetti a noi ignoti. Inoltre, per input molto grandi (il che corrisponde a situazioni pratiche) l'influenza che tali costanti hanno sul "reale" tempo di esecuzione di un algoritmo è minima.

Misureremo la complessità di un algoritmo in termini del *numero di operazioni elementari* eseguite. Esempi di operazioni elementari sono: addizione e moltiplicazione di numeri (se piccoli), confronto tra elementi, seguire un puntatore, etc. Si assume che le operazioni elementari richiedano un tempo *costante* per la loro esecuzione (=una unità di tempo).

Il tempo di esecuzione di un algoritmo in generale dipende quanto grande è l'input (taglia o dimensione dell'input). Ma cosa intendiamo esattamente per *taglia dell'input*? Si usano in generale due criteri per valutare la dimensione dell'input:

- criterio di costo logaritmico: la taglia dell'input è il numero di bits necessari per rappresentarlo
- Criterio di costo uniforme: la taglia dell'input è il numero di "elementi" che lo costituiscono
- Nell'ipotesi che ogni istruzione del computer richieda un'unità di tempo e che un numero o "elemento" richieda una parola di memoria, le due misure coincidono a meno di una costante moltiplicativa

Ad esempio:

Problema	taglia dell'input	operazione elementare
Trovare x in una lista	# di elementi della lista	confronto tra elementi
Moltiplicazione di matrici	dimensione delle matrici	moltiplicazione di due numeri
Ordinamento di un array	# di elementi dell'array	confronto tra elementi
Visita di un albero/grafo	#numero di nodi/archi	seguire un puntatore
Moltiplicazione di due numeri	# bits per rappresentarli	addizione/sottrazione

Nell'ultimo caso della moltiplicazione di due numeri abbiamo usato il criterio di costo logaritmico per misurare la taglia dell'input, nei primi quattro casi abbiamo usato il criterio di costo uniforme.

La complessità di un algoritmo può dipendere anche da come è fatto l'input (ad es., ordinare n elementi puó essere fatto piú velocemente se essi sono già (quasi) ordinati) Useremo come strumento di analisi il *caso peggiore*: ovvero il tempo usato dall'algoritmo nella situazione piú "sfavorevole" in input.

La complessità di tempo di un algoritmo $\mathcal{A}$, su input di taglia n, che denoteremo con $T(n)$ sarà quindi

$T(n)$ = il massimo tempo richiesto da $\mathcal{A}$, dove il massimo è calcolato su tutti i *possibili* input di taglia n

Ciò ci darà un limite superiore alla complessità dell'algoritmo $\mathcal{A}$, su tutti i possibili input (nel senso che $\mathcal{A}$ non richiederà mai tempo superiore a $T(n)$ su di un qualsivoglia input di taglia n).

Vediamo un semplice esempio. Siano dati in input una sequenza di interi `a=a[0]a[1].... a[n-1]` ed un numero `x`. Vogliamo scoprire se `x` è presente in `a`. Un semplice algoritmo potrebbe essere il seguente.

```
Scopri(a,x)
i=0
WHILE(i < n && a[i]!=x){
   i=i+1
}
IF (i=n){
   RETURN (‘‘x non c’è’’)
   ELSE RETURN (‘‘x=a[i]’’)
}
```

É chiaro che a seconda del caso che `x` appaia o meno in `a` (ed in quale posizione di `a` esso appaia), l'algoritmo `Scopri(a,x)` può eseguire un *qualunque* numero di operazioni comprese tra 1 ed n. Potremmo quindi dire che la complessità $T(n)$ è (in generale) $O(n)$. Visto però che abbiamo convenuto di valutare la complessità di un algoritmo in base al suo caso peggiore, è più corretto dire che complessità $T(n)$ è $\Theta(n)$, in quanto nel caso peggiore (ad esempio nel caso in cui `x` non appare in `a`) l'algoritmo `Scopri(a,x)` esegue $\Omega(n)$ operazioni.

Quindi, se diciamo che la complessità $T(n)$ di un algoritmo $\mathcal{A}$ è $O(n^2)$, vuol dire che, avendo un qualsivoglia input di dimensione n, l'algoritmo $\mathcal{A}$ non richiede mai tempo superiore a cn^2, per produrre il suo output, per c opportuna e n sufficientemente grande.

Se diciamo invece che la complessità $T(n)$ di un algoritmo $\mathcal{A}$ è $\Omega(n^2)$, vuol dire che $\mathcal{A}$ richiede almeno tempo cn^2, per produrre il suo output nel caso peggiore, per c opportuna e n sufficientemente grane.

Quindi NON si intende dire che $\mathcal{A}$ impiega tempo cn^2, su ogni input di taglia n, ma nel secondo caso si intende che ESISTE almeno un input di taglia n (con n sufficientemente grande) su cui $\mathcal{A}$ richiede tempo cn^2 (e nel primo caso si intende che NON esistono input per cui l'algoritmo richiede tempo $> cn^2$).

Vediamo un altro esempio, ancora più estremo:

```
Alg(n)
1. x=0
2. IF (n è pari){
3.     RETURN 0
4.     ELSE FOR(i=1; i<n+1; i=i+1){
5.       x=x+1
    }
}
RETURN x
```

La complessità dell'algoritmo *nel caso peggiore* è $\Theta(n)$. Possiamo dire che l'algoritmo esegue $\Theta(n)$ passi *per ogni* input? No! Infatti esise un numero infinito di input su cui l'algoritmo esegue *un solo passo*. Ció che possiamo dire è che l'algoritmo non esegue mai piú di $O(n)$ passi *per ogni possibile input*, e che esegue almeno n passi *per certi input*.

Effettueremo quasi esclusivamente analisi nel caso peggiore, usando le seguenti regole:

- il costo di istruzioni semplici, quali assegnazione, letture/scrittura di variabili è $O(1)$ (ovvero, una costante che non dipende dall'input)
- il costo di istruzioni tipo `IF .... ELSE` è pari al tempo per effettuare il test (tipicamente $O(1)$) piú O(costo della alternativa piú costosa)
- il costo di iterazioni (`FOR`, `WHILE`, `REPEAT`) è pari alla somma su tutte le iterazioni del costo di ogni iterazioni

Apriamo una parentesi per ricordare come si valutano le progressioni geometriche.

Somme finite: Sia $S_n = \sum_{i=0}^{n} \alpha^i$. Se $\alpha \neq 1$ allora

$$\begin{aligned} S_n(\alpha - 1) &= \alpha \cdot S_n - S_n = \sum_{i=0}^{n} \alpha^{i+1} - \sum_{i=0}^{n} \alpha^i \\ &= \alpha + \alpha^2 + \ldots + +\alpha^n + \alpha^{n+1} - 1 - \alpha - \alpha^2 - \ldots - \alpha^n \\ &= \alpha^{n+1} - 1 \end{aligned}$$

da cui $S_n = (\alpha^{n+1} - 1)/(\alpha - 1)$.

Somme infinite: Supponiamo $0 < \alpha < 1$ e sia $S = \sum_{i=0}^{\infty} \alpha^i$.

Allora $\alpha S = \sum_{i=1}^{\infty} \alpha^i$, da cui

$$S(1 - \alpha) = S - \alpha S = \sum_{i=0}^{\infty} \alpha^i - \sum_{i=1}^{\infty} \alpha^i = \alpha^0 = 1, \text{ovvero } S = 1/(1 - \alpha)$$

Vediamo un esempio. Calcoliamo $\sum_{i=0}^{\lfloor \log_3 n \rfloor}(1/3)^i$. Ovviamente vale che $\sum_{i=0}^{\lfloor \log_3 n \rfloor}\left(\frac{1}{3}\right)^i < \sum_{i=0}^{\infty}\left(\frac{1}{3}\right)^i$

Applicando quindi la formula

$$S_n = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

alla espressione

$$\sum_{i=0}^{\infty}\left(\frac{1}{3}\right)^i$$

otteniamo

$$\sum_{i=0}^{\lfloor \log_3 n \rfloor}\left(\frac{1}{3}\right)^i < \sum_{i=0}^{\infty}\left(\frac{1}{3}\right)^i = \frac{1}{1-(1/3)} = (3/2).$$

Parliamo ora di Analisi di Algoritmi Ricorsivi, ovvero di: Come esprimere la complessità di algoritmi ricorsivi mediante relazioni di ricorrenza, come derivare le relazioni di ricorrenza, e come risolvere le relazioni di ricorrenza.

Gli algoritmi ricosivi hanno, generalmente, una struttura di questo tipo:

```
Algoritmo A(I)
IF ( I è ‘‘di taglia piccola’’){
    RETURN output
   }ELSE
⋮
   A(I_1)
⋮
   A(I_a)
⋮
```

dove

- I è l'input iniziale all'algoritmo
- i puntini " $\vdots$ " sono istruzioni di qualche tipo
- $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ sono le a chiamate ricorsive all'algoritmo $\mathcal{A}$, rispettivamente sugli input $I_1, \ldots, I_a$.

Per derivare le relazioni di ricorrenza che descrivono il tempo di esecuzione $T(n)$ di un algoritmo occorre:

1. Determinare la taglia dell'input n
2. Determinare quale valore n_0 di n è usato per la base della ricorsione (generalmente, ma non sempre, $n_0 = 1$).
3. Determinare il valore $T(n_0)$ del tempo di esecuzione sulla base della ricorsione (spesso basterà essere certi che $T(n_0) = c$, per qualche costante c)

Il valore $T(n)$ sarà generalmente uguale ad una somma del tipo $T(m_1)+\ldots+T(m_a)$ (per le chiamate ricorsive), piú la somma di eventuale altro lavoro fatto. Spesso le a chiamate ricorsive saranno effettuate tutte su sottoproblemi di taglia uguale $f(n)$, dando un termine $aT(f(n))$ nella relazione di ricorrenza.

Una tipica relazione di ricorrenza sarà:

$$T(n) = \begin{cases} c & \text{se } n = n_0 \\ \\ aT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- n_0 =base ricorsione, c =tempo di esecuzione per la base
- a =numero di volte che le chiamate ricorsive sono effettuate
- $f(n)$ =taglia dei problemi risolti nelle chiamate ricorsive
- $g(n)$ =tutto il tempo di calcolo non incluso nelle chiamate ricorsive

Vediamo qualche esempio.

```
Algoritmo1(n)
IF (n==1) {
   fà qualcosa
   ELSE {
     Algoritmo1(n-1)
     Algoritmo1(n-2)
    }
  }
FOR(i=1; i<n+1;i=i+1){
   fà qualcos'altro
}
```

L'equazione di ricorrenza che descrive la complessità $T(n)$ dell'`Algoritmo1(n)` sarà

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ \\ T(n-1) + T(n-2) + dn & \text{altrimenti} \end{cases}$$

```
Algoritmo2(n)
IF ((n==1)||(n==2) {
   fà qualcosa
   ELSE Algoritmo2(n/2)
        Algoritmo2(n/2) }
FOR(i=1; i<n+1;i=i+1){
   fà qualcos'altro
}
```

L'equazione di ricorrenza che descrive la complessità $T(n)$ dell'`Algoritmo2(n)` sarà

$$T(n) = \begin{cases} c & \text{se } n \leq 2 \\ \\ 2T(n/2) + dn & \text{altrimenti} \end{cases}$$

```
Algoritmo4(n)
IF (n==1) {
   fà qualcosa
   } ELSE {
   FOR(i=1; i<n+1;i=i+1){
   Algoritmo4(i)
   }
}
   fà qualcos'altro
```

L'equazione di ricorrenza che descrive la complessità $T(n)$ dell'`Algoritmo4(n)` sarà

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ \\ \sum_{i=1}^{n} T(i) + d & \text{altrimenti} \end{cases}$$

Vale il seguente utile risultato:

Teorema: Per arbitrarie costanti c, a, e d la soluzione alla ricorrenza

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ aT(n/c) + bn & \text{altrimenti} \end{cases}$$

è data da

$$T(n) = \begin{cases} O(n) & \text{se } a < c \\ O(n \log n) & \text{se } a = c \\ O(n^{\log_c a}) & \text{se } a > c \end{cases}$$

Esempi di applicazione del Teorema:
• Se $T(n) = 2T(n/3) + dn$, allora $T(n) = O(n)$
• Se $T(n) = 2T(n/2) + dn$, allora $T(n) = O(n \log n)$
• Se $T(n) = 4T(n/2) + dn$, allora $T(n) = O(n^2)$

Diamo una bozza di dimostrazione del Teorema quando $n = c^i$ per qualche intero i. Si ha

$$\begin{aligned}
T(n) &= a \cdot T(n/c) + bn \text{ (per ipotesi)} \\
&= a(a \cdot T(n/c^2) + bn/c) + bn \text{ (in quanto } T(n/c) = aT(n/c^2) + bn/c) \\
&= a^2 \cdot T(n/c^2) + abn/c + bn \\
&= a^2(a \cdot T(n/c^3) + bn/c^2) + abn/c + bn \\
&= a^3 \cdot T(n/c^3) + a^2 bn/c^2 + abn/c + bn \\
&\cdots \\
&= a^i T(n/c^i) + bn \sum_{j=0}^{i-1} (a/c)^j \\
&= a^{\log_c n} T(1) + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j \text{ (poichè } i = \log_c n) \\
&= d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j
\end{aligned}$$

avendo usato il fatto che $T(1) = d$ e $a^{\log_c n} = n^{\log_c a}$.

Per calcolare

$$\sum_{j=0}^{\log_c n - 1} (a/c)^j$$

occorre ricordare ció che abbiamo già visto sulle Progressioni Geometriche, ovvero:

Somme finite: Se $\alpha \neq 1$ allora

$$\sum_{i=0}^{n} \alpha^i = \frac{\alpha^{n+1} - 1}{\alpha - 1} \qquad (1)$$

Somme infinite: Se $0 < \alpha < 1$. Allora

$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha} \qquad (2)$$

Caso 1: $a < c$ (cioè $(a/c) < 1$)

$$\sum_{j=0}^{\log_c n - 1} (a/c)^j < \sum_{j=0}^{\infty} (a/c)^j = \frac{1}{1 - a/c} = \frac{c}{c - a}$$

dalla formula (2) precedente).

Pertanto

$$T(n) < d \cdot n^{\log_c a} + bcn/(c - a) = O(n)$$

(Infatti, $a < c \Rightarrow \log_c a < 1$ e quindi il termine $d \cdot n^{\log_c a}$ è $O(n)$).

Caso 2: $a = c$

Allora

$$\begin{aligned} T(n) &= d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j \\ &= d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} (1)^j \\ &= O(n \log n) \end{aligned}$$

Caso 3: $a > c$ (il che implica $(a/c) \neq 1$)

Dalla precedente formula (1) otteniamo

$$\begin{aligned} \sum_{j=0}^{\log_c n - 1} (a/c)^j &= \frac{(a/c)^{\log_c n} - 1}{(a/c) - 1} \\ &= \frac{n^{\log_c a - 1} - 1}{(a/c) - 1} \qquad \text{(poichè } a^{\log_c n} = n^{\log_c a} \text{ e } (1/c)^{\log_c n} = 1/n) \\ &= O(n^{\log_c a - 1}) \end{aligned}$$

Pertanto,

$$T(n) = d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j = O(n^{\log_c a})$$

Cosa accade se n NON è potenza di c?

Osserviamo che per $k = \lfloor \log_c n \rfloor$ vale che

$$c^k \leq n < c^{k+1} \quad \text{e} \quad cn \geq c^{k+1} \qquad (3)$$

Immaginiamo quindi di "aumentare" l'input di taglia n, fino a farlo diventare di taglia c^{k+1}. É ovvio che $T(n) < T(c^{k+1})$.

Applicando ora il teorema a $T(c^{k+1})$, otteniamo

$$T(n) < T(c^{k+1}) = \begin{cases} O(c^{k+1}) = O(cn) = O(n) & \text{se } a < c \\ O(c^{k+1} \log c^{k+1}) = O(cn \log(cn)) = O(n \log n) & \text{se } a = c \\ O((c^{k+1})^{\log_c a}) = O((cn)^{\log_c a}) = O(n^{\log_c a}) & \text{se } a > c \end{cases}$$

Sia $T(1) = 1$. Valutiamo

- $T(n) = 2T(n/2) + 6n \qquad T(n) = O(n \log n)$

- $T(n) = 3T(n/3) + 6n - 9 \qquad T(n) = O(n \log n)$
- $T(n) = 2T(n/3) + 5n \qquad T(n) = O(n)$
- $T(n) = 2T(n/3) + 12n + 16 \qquad T(n) = O(n)$
- $T(n) = 4T(n/2) + n \qquad T(n) = O(n^{\log_2 4}) = O(n^2)$
- $T(n) = 3T(n/2) + 9n \qquad T(n) = O(n^{\log_2 3}) = O(n^{1.584\ldots})$

Con esattamente la stessa tecnica è possibile dimostrare una forma un pò più generale del Teorema, ad esempio la seguente:

Teorema: La soluzione alla ricorrenza

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ aT(n/c) + bn^k & \text{altrimenti} \end{cases}$$

per a, c, b, k costanti, è

$$T(n) = \begin{cases} O(n^k) & \text{se } a < c^k \\ O(n^k \log n) & \text{se } a = c^k \\ O(n^{\log_c a}) & \text{se } a > c^k \end{cases}$$

Esempi. Sia $T(1) = 1$. Valutiamo

- $T(n) = 2T(n/2) + n^3 \qquad T(n) = O(n^3)$
- $T(n) = T(9n/10) + n \qquad T(n) = O(n)$
- $T(n) = 16T(n/4) + n^2 \qquad T(n) = O(n^2 \log n)$
- $T(n) = 7T(n/3) + n^2 \qquad T(n) = O(n^2)$
- $T(n) = 7T(n/2) + n^2 \qquad T(n) = O(n^{\log_2 7})$
- $T(n) = 2T(n/3) + \sqrt{n} \qquad T(n) = O(n^{\log_3 2})$
- $T(n) = T(n-1) + n \qquad T(n) = O(?)$
- $T(n) = T(\sqrt{n}) + 1 \qquad T(n) = O(?)$

Esistono forme ancora piú generali del Teorema, che permettono la risoluzione di equazioni di ricorrenza del tipo generale

$$T(n) = aT(n/b) + f(n)$$

Sussiste il seguente risultato

1.Se $f(n) = O(n^{\log_b a - \epsilon})$, per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$

2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$

3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$, per qualche $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ per qualche costante $c > 1$ ed n sufficientemente grande, allora $T(n) = \Theta(f(n))$