# Ordinamenti e crescita della complessità

Informatica@SEFA 2017/2018 - Lezione 11

Massimo Lauria <massimo.lauria@uniroma1.it>*

Venerdì, 27 Ottobre 2017

**Nota bibliografica:** Il contenuto di questa e di alcune delle prossime lezioni può essere trovato nei primi capitoli del libro *Introduzione agli Algoritmi e strutture dati* (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein). Tuttavia il libro è grosso e costoso, quindi cerco di mettere tutto il materiale negli appunti del corso.

## La misura della complessità computazionale

Quando consideriamo le prestazioni di un algoritmo non vale la pena considerare l'effettivo tempo di esecuzione o la quantità di RAM occupate. Queste cose dipendono da caratteristiche tecnologiche che niente hanno a che vedere con la qualità dell'algoritmo. Al contrario è utile considerare il numero di **operazioni elementari** che l'algoritmo esegue.

- accessi in memoria
- operazioni aritmetiche
- ecc. . .

Che cos'è un'operazione elementare? Per essere definiti, questi concetti vanno espressi su modelli computazionali **formali e astratti**, sui quali definire gli algoritmi che vogliamo misurare. Per noi tutto questo non è necessario. Ci basta capire a livello intuitivo il numero di operazioni elementari che i nostri algoritmi fanno per ogni istruzione.

*http://massimolauria.net/courses/infosefa2017/

### Esempi

- una somma di due numeri `float` : 1 op.
- verificare se due numeri `float` sono uguali: 1 op.
- confronto lessicografico tra due stringhe di lunghezza $n$: $n$ op.
- ricerca lineare in una lista Python di $n$ elementi: $n$ op.
- ricerca binaria in una lista ordinata di $n$ elementi: $\log n$ op.
- inserimento nella 4a posizione in una lista di $n$ elementi: $n$ op.
- `seq[a:b]` : $b - a$ operazioni
- `seq[:]` : `len(seq)` operazioni
- `seq1 + seq2` : `len(seq1) + len(seq2)` operazioni

## Un algoritmo di ordinamento: "Insertion sort"

Nella lezione precedente abbiamo visto due tipi di ricerca. Quella *sequenziale* o *lineare* e quella *binaria*. Per utilizzare la ricerca binaria è necessario avere la sequenza di dati ordinata. Oggi vedremo un **semplice** algoritmo di ordinamento.

- non è molto efficiente
- semplice da realizzare

L'idea dell'insertion sort è simile a quella di una persona che gioca a carte e che vuole ordinare la propria mano. Prende la carta più piccola e la sposta all'inizio. Poi prende la carta più piccola tra le rimanenti e la mette alla seconda posizione, poi la più piccola tra le rimanenti va in terza posizione e così via...

In sostanza se una sequenza ha $n$ elementi, l'insertion sort fa $n$ iterazioni per $i$ da 0 a $n - 1$. In ognuna:

- si calcola il minimo tra gli elementi della sequenza nelle posizioni $i \dots n - 1$;
- si sposta il minimo nella posizione $i$, traslando gli elementi nella sequenza per fagli spazio.

Per prima cosa vediamo come trovare il minimo di una sequenza. La funzione `min` di Python non è sufficiente, perché non restituisce la posizione dell'elemento.[1]

```
def argmin(seq,start,end):
    minimo = seq[start]
    indice = start
    for i in range(start+1,end+1):
        if seq[i] < minimo:
            minimo = seq[i]
            indice = i
    return minimo,indice
```

Poi scriviamo una funzione che sposta tutti di elementi in un intervallo della lista di una posizione verso destra.

```
def moveright(seq,start,end):
    for i in range(end,start-1,-1):
        seq[i+1] = seq[i]
```

```
def insertion_sort(seq):
    print("Step 0: {}".format(seq))
    for i in range(0,len(seq)-1):
        val,pos = argmin(seq,i,len(seq)-1)
        moveright(seq,i,pos-1)
        seq[i] = val
        print("Step {}: {}".format(i+1,seq))
```

Vediamo un esempio con una sequenza arbitraria

```
dati = [21,-4,3,6,1,-5,19]
insertion_sort(dati)
```

```
Step 0: [21, -4, 3, 6, 1, -5, 19]
Step 1: [-5, 21, -4, 3, 6, 1, 19]
Step 2: [-5, -4, 21, 3, 6, 1, 19]
Step 3: [-5, -4, 1, 21, 3, 6, 19]
Step 4: [-5, -4, 1, 3, 21, 6, 19]
Step 5: [-5, -4, 1, 3, 6, 21, 19]
Step 6: [-5, -4, 1, 3, 6, 19, 21]
```

Vediamo un esempio con una sequenza invertita

```
dati = [10,9,8,7,6,5,4,3,2,1]
insertion_sort(dati)
```

```
Step 0: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Step 1: [1, 10, 9, 8, 7, 6, 5, 4, 3, 2]
Step 2: [1, 2, 10, 9, 8, 7, 6, 5, 4, 3]
```

[1] In realtà potremmo usare un trucco. Magari lo vediamo dopo se avanza tempo.

```
Step 3: [1, 2, 3, 10, 9, 8, 7, 6, 5, 4]
Step 4: [1, 2, 3, 4, 10, 9, 8, 7, 6, 5]
Step 5: [1, 2, 3, 4, 5, 10, 9, 8, 7, 6]
Step 6: [1, 2, 3, 4, 5, 6, 10, 9, 8, 7]
Step 7: [1, 2, 3, 4, 5, 6, 7, 10, 9, 8]
Step 8: [1, 2, 3, 4, 5, 6, 7, 8, 10, 9]
Step 9: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Quante operazioni esegue l'insertion sort?

Per ogni ciclo $i$, con $i$ da 0 a $n-1$, l'algoritmo trova il minimo su una sequenza di $n-i$ elementi. Poi sposta tutti gli elementi tra quell $i$-esimo e la posizione del minimo a quel ciclo. Il numero esatto di spostamenti dipende da dove era collocato il minimo. In ogni caso alla peggio sono $n-i$ spostamenti. Dunque in totale sono al massimo

$$\sum_{i=0}^{n-1} 2(n-i) = 2\sum_{i=1}^{n} i = n(n+1) \approx n^2 \tag{1}$$

**Esercizio:** per quali input il numero di operazioni è maggiore?

## Vale la pena ordinare prima di fare ricerche?

Abbiamo visto che $t$ ricerche costano $nt$ se si utilizza la ricerca sequenziale. E se vogliamo utilizzare **insertion sort** seguito da ricerche binarie?

Il costo è $n^2 + t \log n$, che è comparabile con $nt$ solo se si effettuano almeno $t \approx n$ ricerche.

## Una piccola variazione

Eseguiamo una piccola variazione dell'insertion sort, che differisce per due caratteristiche:

- al passo $i$ mettiamo il **massimo** elemento alla posizione $n-i$;
- la ricerca del massimo ed il suo spostamento alla fine nella lista sono fatte simultaneamente.

Vediamo ad esempio come funziona il primo passo: cominciando dall'inizio si confrontano i primi due elementi adiacenti. Se il primo è più grande del secondo, i due elementi si invertono. Poi si passa al second ed al terzo. Poi al terzo con il quarto, e così via.

```
def bubbleup(seq,end,log=False):
    if log:
        print("Step 0: {}".format(seq))
    for i in range(0,end): # si ferma a end-1
        seq[i], seq[i+1] = min(seq[i],seq[i+1]),max(seq[i],seq[i+1])
        if log:
            print("Step {}: {}".format(i+1,seq))
```

```
bubbleup([5,-4,3,6,19,1,-5],6,log=True)
```

```
Step 0: [5, -4, 3, 6, 19, 1, -5]
Step 1: [-4, 5, 3, 6, 19, 1, -5]
Step 2: [-4, 3, 5, 6, 19, 1, -5]
Step 3: [-4, 3, 5, 6, 19, 1, -5]
Step 4: [-4, 3, 5, 6, 19, 1, -5]
Step 5: [-4, 3, 5, 6, 1, 19, -5]
Step 6: [-4, 3, 5, 6, 1, -5, 19]
```

Notate come il massimo sia messo sul fondo della lista, ma al contrario di insertion sort, gli altri elementi potrebbero aver cambiato il loro ordine.

L'ordinamento finale si ottiene ripetendo la procedura, ma al passo $i$ si opera solo sulla sottolista degli elementi dalla posizione 0 alla posizione $n - i$.

```
def stupid_bubblesort(seq):
      print("Step 0: {}".format(seq))
      for i in range(1,len(seq)):
          bubbleup(seq,len(seq)-i)
          print("Step {}: {}".format(i,seq))
```

Vediamo un esempio casuale

```
stupid_bubblesort([5,-4,3,6,19,1,-5])
```

```
Step 0: [5, -4, 3, 6, 19, 1, -5]
Step 1: [-4, 3, 5, 6, 1, -5, 19]
Step 2: [-4, 3, 5, 1, -5, 6, 19]
Step 3: [-4, 3, 1, -5, 5, 6, 19]
Step 4: [-4, 1, -5, 3, 5, 6, 19]
Step 5: [-4, -5, 1, 3, 5, 6, 19]
Step 6: [-5, -4, 1, 3, 5, 6, 19]
```

```
stupid_bubblesort([10,9,8,7,6,5,4,3,2,1])
```

```
Step 0: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Step 1: [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
Step 2: [8, 7, 6, 5, 4, 3, 2, 1, 9, 10]
Step 3: [7, 6, 5, 4, 3, 2, 1, 8, 9, 10]
Step 4: [6, 5, 4, 3, 2, 1, 7, 8, 9, 10]
Step 5: [5, 4, 3, 2, 1, 6, 7, 8, 9, 10]
Step 6: [4, 3, 2, 1, 5, 6, 7, 8, 9, 10]
Step 7: [3, 2, 1, 4, 5, 6, 7, 8, 9, 10]
Step 8: [2, 1, 3, 4, 5, 6, 7, 8, 9, 10]
Step 9: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Vediamo un esempio ordinato in maniera opposta.

```
Step 0: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Step 1: [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
Step 2: [8, 7, 6, 5, 4, 3, 2, 1, 9, 10]
Step 3: [7, 6, 5, 4, 3, 2, 1, 8, 9, 10]
Step 4: [6, 5, 4, 3, 2, 1, 7, 8, 9, 10]
Step 5: [5, 4, 3, 2, 1, 6, 7, 8, 9, 10]
Step 6: [4, 3, 2, 1, 5, 6, 7, 8, 9, 10]
Step 7: [3, 2, 1, 4, 5, 6, 7, 8, 9, 10]
Step 8: [2, 1, 3, 4, 5, 6, 7, 8, 9, 10]
Step 9: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Notazione asintotica

Quando si contano il numero di operazioni in un algoritmo, non è necessario essere estremamente precisi. A che serve distinguere tra $10n$ operazioni e $2n$ operazioni se in un linguaggio di programmazione le $10n$ operazioni sono più veloci delle $2n$ operazioni implementate in un altro linguaggio? Al contrario é molto importante distinguere, ad esempio, $\frac{1}{100}n^2$ operazioni da $20\sqrt{n}$ operazioni. Anche una macchina più veloce paga un prezzo alto se esegue un algoritmo da $\frac{1}{100}n^2$ operazioni invece che uno da $\frac{1}{100}\sqrt{n}$.

Oltretutto un algoritmo asintoticamente più veloce spesso paga un prezzo più alto in fase di inizializzazione (per esempio perché deve sistemare i dati in una certa maniera). Dunque ha senso fare un confronto solo per lunghezze di input grandi abbastanza.

Per discutere in questi termini utilizziamo la notazione asintotica che è utile per indicare quanto cresce il numero di operazioni di un algoritmo, al crescere della lunghezza dell'input.

**Definizione**: date $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ diciamo che

$$f \in O(g)$$

se esistono $c > 0$ e $n_0$ tali che $f(n) \leq cg(n)$ per ogni $n > n_0$.

- la costante $c$ serve a ignorare la dimensione atomica delle operazioni.
- $n_0$ serve a ignorare i valori piccoli di $n$.

Ad esempio $\frac{n^2}{100} + 3n - 10$ è in $O(n^2)$, non è in $O(n^\ell)$ per nessun $\ell < 2$ ed è in $O(n^\ell)$ per $\ell > 2$.

**Complessità di un algoritmo**: dato un algoritmo $A$ consideriamo il numero di operazioni $t_A(x)$ eseguite da $A$ sull'input $x \in \{0,1\}^*$. Per ogni taglia di input $n$ possiamo definire il **costo nel caso peggiore** come

$$c_A(n) := \max_{x \in \{0,1\}^n} t_A(x) \ .$$

Diciamo che un algoritmo $A$ ha complessità $O(g)$ se il suo costo nel caso peggiore $c_A \in O(g)$. Per esempio abbiamo visto due algoritmi di ricerca (ricerca lineare e binaria) che hanno entrambi complessità $O(n)$, ed il secondo ha anche complessità $O(\log n)$.

La notazione $O(g)$ quindi serve a dare un approssimativo limite superiore al numero di operazioni che l'algoritmo esegue.

**Esercizio:** osservare che per ogni $1 < a < b$,

- $\log_a(n) \in O(\log_b(n))$; e
- $\log_b(n) \in O(\log_a(n))$.

Quindi specificare la base non serve.

**Definizione ($\Omega$ e $\Theta$)**: date $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ diciamo che

$$f \in \Omega(g)$$

se esistono $c > 0$ e $n_0$ tali che $f(n) \geq cg(n)$ per ogni $n > n_0$. Diciamo anche che

$$f \in \Theta(g)$$

se $f \in \Omega(g)$ e $f \in O(g)$ simultaneamente.

**Complessità di un algoritmo (II)**: dato un algoritmo $A$ consideriamo ancora il **costo nel caso peggiore** di $A$ su input di taglia $n$ come

$$c_A(n) := \max_{x \in \{0,1\}^n} t_A(x) \ .$$

Diciamo che un algoritmo $A$ ha complessità $\Omega(g)$ se $c_A \in \Omega(g)$, e che ha complessità $\Theta(g)$ se $c_A \in \Theta(g)$.

Per esempio abbiamo visto due algoritmi di ricerca (ricerca lineare e binaria) che hanno entrambi complessità $\Omega(\log n)$, ed il primo anche complessità $\Omega(n)$. Volendo essere più specifici possiamo dire che la ricerca lineare ha complessità $\Theta(n)$ e quella binaria ha complessità $\Theta(\log n)$.

Fate attenzione a questa differenza:

- se $A$ ha complessità $O(g)$, allora $A$ gestisce tutti gli input con al massimo $g$ operazioni circa, asintoticamente.
- se $A$ ha complessità $\Omega(g)$, allora esiste una famiglia infinita di input tali che $A$ richiede almeno $g$ operazioni circa.

### Note su come misuriamo la lunghezza dell'input

Gli ordini di crescita sono il linguaggio che utilizziamo quando vogliamo parlare dell'uso di risorse computazionali, indicando come le risorse crescono in base alla dimensione dell'input.

La lunghezza dell'input $n$ consiste nel numero di bit necessari a codificarlo. Tuttavia per problemi più specifici è utile, e a volte più comodo, indicare con $n$ la dimensione "logica" dell'input. Ad esempio:

- trovare il massimo in una sequenza. $n$ sarà la quantità di numeri nella sequenza.
- trovare una comunità in una rete sociale con $x$ nodi, e $y$ connessioni. La lunghezza dell'input può essere sia $x$ che $y$ a seconda dei casi.
- Moltiplicare due matrici quadrate $n \times n$. La dimensione dell'input è il numero di righe (o colonne) delle due matrici.

Naturalmente misurare così la complessità ha senso solo se consideriamo, ad esempio, dati numerici di grandezza limitata. Ovvero numeri di grandezza tale che le operazioni su di essi possono essere considerate atomiche senza l'analisi dell'algoritmo ne sia influenzata.

## Esercitarsi sulla notazione asintotica

Ci sono alcuni fatti ovvi riguardo la notazione asintotica, che seguono immediatamente le definizioni. Dimostrate che

1. per ogni $0 < a < b$, $n^a \in O(n^b)$, $n^b \notin O(n^a)$;
2. per ogni $0 < a < b$, $n^b \in \Omega(n^a)$, $n^a \notin \Omega(n^b)$;
3. se $f \in \Omega(h)$ allora $f \cdot g \in \Omega(h \cdot g)$;
4. se $f \in O(h)$ allora $f \cdot g \in O(h \cdot g)$;
5. che $(n + a)^b \in \Theta(n^b)$.
6. preso un qualunque polinomio $p = \sum_{i=0}^{d} \alpha_i x^i$ si ha che $p(n) \in \Theta(n^d)$;
7. che $n! \in \Omega(2^n)$ ma che $n! \notin O(2^n)$;
8. che $\binom{n}{d} \in n^d$.