# Analisi degli algoritmi

Progettazione di Algoritmi a.a. 2023-24

Matricole congrue a 1

Docente: Annalisa De Bonis

# Analisi degli algoritmi

- E' possibile progettare diversi algoritmi per risolvere uno stesso problema
  - Si pensi ad esempio agli algoritmi di ordinamento di n numeri: Merge Sort, Quick Sort, Insertion Sort, Bubble Sort, Selection Sort, Heap Sort, ...
- Un algoritmo può impiegare molto meno tempo di un altro
  - MergeSort: tempo proporzionale a nlog n (tempo pari circa a cnlogn dove c è una costante che non dipende da n)
  - QuickSort: tempo nel caso peggiore proporzionale a $n^2$ (tempo pari circa a $c'n^2$ dove c' è una costante che non dipende da n)

  o usare molto meno spazio di un altro
  - Alcuni algoritmi di ordinamento non utilizzano strutture dati ausiliarie in quanto ordinano "sul posto" andando a modificare la posizione degli elementi all'interno della sequenza input. Questi algoritmi richiedono solo una piccola quantità di memoria aggiuntiva che è molto inferiore rispetto alla dimensione n dell'input e in ogni momento mantengono al piu` un numero costante di elementi in memoria aggiuntiva.
    - Esempi: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort,....

# Analisi degli algoritmi

- È utile avere un modo per confrontare tra loro diverse soluzioni per capire quale sia la migliore. Migliore in base ad un certo criterio di efficienza, come ad esempio uso della memoria o velocità .
- Abbiamo bisogno di tecniche di analisi che consentano di valutare un algoritmo **solo in base alle sue caratteristiche e non a quelle del codice che lo implementa o della macchina su cui è eseguito.**
- Come informatici, oltre a dover essere in grado di trovare soluzioni ai problemi, dobbiamo essere in grado di valutare la nostra soluzione e capire se c'è margine di miglioramento.
  - Limiti inferiori

## Efficienza degli algoritmi

Proviamo a definire la nozione di efficienza (rispetto al tempo di esecuzione):

- ***Un algoritmo è efficiente se, quando è implementato, viene eseguito velocemente su istanze input reali.***

  - Concetto molto vago.

    - Non chiarisce **dove** viene eseguito l'algoritmo e **quanto veloce** deve essere la sua esecuzione

      - Anche un algoritmo molto cattivo può essere eseguito molto velocemente se è applicato a un input molto piccolo o se è eseguito con un processore molto veloce

      - Anche un algoritmo molto buono può richiedere molto tempo per essere eseguito se implementato male

    - …

## Efficienza degli algoritmi

- ...
- Non chiarisce cosa è un'istanza input reale
  - Noi non conosciamo a priori tutte le possibili istanze input reali
  - Alcune istanze potrebbero essere più "cattive" di altre
- Inoltre non fa capire come la velocità di esecuzione dell'algoritmo deve variare al crescere della dimensione dell'input
  - Due algoritmi possono avere tempi di esecuzione simili per input piccoli ma tempi di esecuzione molto diversi per input grandi

# Efficienza degli algoritmi

- **Vogliamo una definizione concreta di efficienza**
  - **indipendente dal processore**
  - **indipendente dal tipo di istanza**
  - **che dia una misura di come aumenta il tempo di esecuzione al crescere della dimensione dell'input.**

# Efficienza

Forza bruta. Per molti problemi non triviali, esiste un naturale algoritmo di forza bruta che controlla ogni possibile soluzione.

- Tipicamente impiega tempo $2^N$ (o peggio) per input di dimensione N.
- Non accettabile in pratica.

- Esempio:
  - Voglio ordinare in modo crescente un array di N numeri distinti
  - Soluzione (ingenua) esponenziale: permuto i numeri ogni volta in modo diverso fino a che ottengo la permutazione ordinata (posso verificare se una permutazione è ordinata con al più N-1 confronti, confrontando ciascun elemento con il successivo)
    - Nel caso pessimo genero N! permutazioni
    - NB: $N! = 1 \times 2 \times 3 \times 4 \times \ldots \times n > 2^N$ per $N>3$

# Efficienza

- **Problemi con l'approccio basato sulla ricerca esaustiva nello spazio di tutte le possibili soluzioni (forza bruta)**
  - **Ovviamente richiede molto tempo**
  - **Non fornisce alcuna informazione sulla struttura del problema che vogliamo risolvere.**

- **Proviamo a ridefinire la nozione di efficienza**: Un algoritmo è efficiente se ha una performance migliore, da un punto di vista analitico, dell'algoritmo di forza bruta.

- **Definizione molto utile.** Algoritmi che hanno performance migliori rispetto agli algoritmi di forza bruta di solito **usano euristiche interessanti** e **forniscono informazioni rilevanti sulla struttura intrinseca del problema e sulla sua trattabilità computazionale.**
- **Problema con questa definizione.** Anche questa definizione è vaga. Cosa vuol dire "perfomance migliore"?

# Tempo polinomiale

**Proprietà desiderata.** Quando la dimensione dell'input raddoppia, l'algoritmo dovrebbe risultare più lento solo di un fattore costante

**Mergesort:**

taglia input= N --> tempo $c \times N \times \log_2 N$;

taglia input= 2N --> tempo $c \times 2N \times \log_2(2N) = 2 \times c \times N \times (\log_2 N + 1)$

$= 2 \times c \times N \times \log_2 N + 2 \times c \times N$

$\leq 2 \times c \times N \times \log_2 N + 2 \times c \times N \times \log_2 N$

$= \mathbf{4} \times c \times N \times \log_2 N$ per ogni $N \geq 2$;

aumenta di al più **4** volte

**Algoritmo di forza bruta:** taglia input= N --> tempo $c \times N!$

taglia input= 2N --> tempo $c \times (2N)! = c \times (2N \times (2N-1) \times \cdots \times (N+1)) \times N!$

$> N! \times c \times N!$

N! non è una costante

## Tempo polinomiale

Def. Si dice che un algoritmo impiega tempo polinomiale (poly-time) se quando la dimensione dell'input aumenta di un fattore costante, ad esempio raddoppia, l'algoritmo risulta più lento solo di un fattore costante c

Esistono due costanti $c > 0$ e $d > 0$ tali che su ciascun input di dimensione N, il numero di passi è $\leq c\,N^d$ .

Se si passa da un input di dimensione N ad uno di dimensione 2N allora il tempo di esecuzione passa da $c\,N^d$ a $c\,(2N)^d = c2^dN^d = 2^d\,c\,N^d$

NB: $2^d$ è una costante

# Analisi del caso pessimo

Tempo di esecuzione nel caso pessimo. Ottenere un bound sul **più grande tempo di esecuzione possibile** per tutti gli input di una certa dimensione N.

- In genere è una buona misura di come si comportano gli algoritmi nella pratica
- Approccio "pessimistico" (in molti casi l'algoritmo potrebbe comportarsi molto meglio)
  - ma è difficile trovare un'alternativa efficace a questo approccio

Tempo di esecuzione nel caso medio. Ottenere un bound al tempo di esecuzione su un **input random** in funzione di una certa dimensione N dell'input.

- Difficile se non impossibile modellare in modo accurato istanze reali del problema mediante distribuzioni input.
- Un algoritmo disegnato per una certa distribuzione di probabilità sull'input potrebbe comportarsi molto male in presenza di altre distribuzioni.

## Tempo polinomiale nel caso pessimo

Def. Un algoritmo è **efficiente** se il suo tempo di esecuzione nel caso pessimo è polinomiale.

Motivazione: Funziona veramente in pratica!

- Sebbene $6.02 \times 10^{23} \times N^{20}$ sia, da un punto di vista tecnico, polinomiale, un algoritmo che impiega questo tempo potrebbe essere inutile in pratica.
- Per fortuna, i problemi per cui esistono algoritmi che li risolvono in tempo polinomiale, quasi sempre ammettono algoritmi polinomiali il cui tempo di esecuzione è proporzionale a polinomi che crescono in modo moderato ($c \times N^d$ con c e d piccoli).
- Progettare un algoritmo polinomiale porta a scoprire importanti informazioni sulla struttura del problema

# Eccezioni

- Alcuni algoritmi polinomiali hanno costanti moltiplicative e/o all'esponente grandi e sono inutili nella pratica

- Alcuni algoritmi esponenziali sono largamente usati perchè il caso pessimo si presenta molto raramente.
  - Esempio: algoritmo del simplesso per risolvere problemi di programmazione lineare

# Perchè l'analisi della complessità è importante

La tabella riporta i tempi di esecuzione su input di dimensione crescente, per un processore che esegue un milione di istruzioni per secondo.
Nei casi in cui il tempo di esecuzione è maggiore di $10^{25}$ anni, la tabella indica che il tempo richiesto è molto lungo (very long). N.B.: la formazione del pianeta Terra risale a circa $4{,}54 \times 10^9$ di anni fa.

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 4 sec |
| $n = 30$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 18 min | $10^{25}$ years |
| $n = 50$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 11 min | 36 years | very long |
| $n = 100$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | $< 1$ sec | $< 1$ sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | $< 1$ sec | $< 1$ sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | $< 1$ sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

cosa accadrebbe se usassimo un processore 1000 volte piu` veloce che esegue $10^9$ istruzioni al sec?

- tempo di esecuzione n: se prima in $t = n/10^6$ secondi eseguivo l'algorimo con input di taglia n ora nello stesso tempo posso eseguirlo con input di taglia m, con $m/10^9 \le t$, cioe` $m \le 10^9\, t = 10^9\, n/10^6 = 10^3\, n$ (posso risolvere problema per input 1000 volte piu` grande: 1000n)
- tempo di esecuzione $2^n$: se prima in $t = 2^n/10^6$ secondi eseguivo l'algorimo con input di taglia n ora nello stesso tempo posso eseguirlo con input di taglia m, con m tale che $2^m/10^9 \le t$, cioe` $2^m \le 10^9\, t = 10^9\, 2^n/10^6 = 10^3\, 2^n$ → $m \le \log_2 (10^3\, 2^n) = 3 \log_2 10 + \log_2 2^n \le 10 + n$ (posso risolvere il problema per un input di grandezza appena piu` grande: n+10)

# Analisi degli algoritmi

Esempio:

```
InsertionSort(a):        //n è la lunghezza di a
 For(i=1;i<n;i=i+1){
   elemDaIns=a[i];
   j=i-1;
   While(j ≥ 0 && a[j] > elemDaIns){ //cerca il posto per a[i]
        a[j+1]=a[j];      //shift a destra degli  elementi più grandi
        j=j-1;
    }
   a[j+1]=elemDaIns;
 }
```

# Analisi degli algoritmi

$t_i$= numero di iterazioni del while all'i-esima iterazione del for

| | costo | numero di volte |
|---|---|---|
| InsertionSort(a): | | |
| For(i=1;i<n;i=i+1){ | $c_1$ | n |
| elemDaIns=a[i]; | $c_2$ | n-1 |
| j=i-1; | $c_3$ | n-1 |
| While(j ≥ 0 && a[j] > elemDaIns){ | $c_4$ | $\sum_{i=1}^{n-1} t_i$ |
| a[j+1]=a[j]; | $c_5$ | $\sum_{i=1}^{n-1} (t_i-1)$ |
| j=j-1; | $c_6$ | $\sum_{i=1}^{n-1} (t_i-1)$ |
| } | | |
| a[j+1]=elemDaIns; | $c_7$ | n-1 |
| } | | |

# Analisi di InsertionSort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n} (t_i - 1) + c_7(n-1)$$

Nel caso pessimo $t_i$ =i+1 per ogni i (elementi in ordine decrescente)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} (i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n} i + c_7(n-1)$$

# Analisi di InsertionSort

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1}(i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4((\sum_{i=1}^{n-1} i) + n - 1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{(n-1)n}{2} + n - 1) + c_5(\frac{(n-1)n}{2}) + c_6(\frac{(n-1)n}{2}) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2}{2} + \frac{n}{2} - 1) + c_5(\frac{n^2}{2} - \frac{n}{2}) + c_6(\frac{n^2}{2} - \frac{n}{2}) + c_7(n-1) \\
&= (c_4 + c_5 + c_6)\frac{n^2}{2} + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 - c_3 - c_4 - c_7) \\
&= an^2 + bn + c
\end{aligned}
$$

## Ordine di grandezza

- Nell'analizzare la complessità di InsertionSort abbiamo operato delle astrazioni
  - Abbiamo ignorato il valore esatto prima delle costanti $c_i$ e poi delle costanti a, b, c.
  - Il calcolo di queste costanti per alcuni algoritmi può essere molto stancante ed è inutile rispetto alla classificazione degli algoritmi che vogliamo ottenere.
  - Queste costanti inoltre dipendono
    - dalla macchina su cui si esegue il programma
    - dal tipo di operazioni che contiamo
      - Operazioni del linguaggio ad alto livello
      - Istruzioni di basso livello in linguaggio macchina

## Ordine di grandezza

- Possiamo aumentare il livello di astrazione considerando solo l'ordine di grandezza
  - Consideriamo solo il termine "dominante"
    - Per InsertionSort: $an^2$
    - Giustificazione: più **grande** è n, minore è il contributo dato dagli altri termini alla stima della complessità
  - Ignoriamo del tutto le costanti
    - Diremo che il tempo di esecuzione di InsertionSort ha ordine di grandezza $n^2$
    - Giustificazione: più grande è n, minore è il contributo dato dalle costanti alla stima della complessità

# Efficienza asintotica degli algoritmi

- Per input piccoli può non essere corretto considerare solo l'ordine di grandezza ma per input "abbastanza" grandi è corretto farlo

- Esempio: $10n^2+100n+10$
  per $n<10$, il secondo termine è maggiore del primo
  man mano che n cresce il contributo dato dai termini meno significativi diminuisce