

# Algoritmi e Strutture Dati<sup>1</sup>

Corso di Laurea in Ingegneria dell'Informazione  
Sapienza Università di Roma – sede di Latina

Fabio Patrizi

Dipartimento di Ingegneria Informatica, Automatica e Gestionale (DIAG)  
SAPIENZA Università di Roma – Italy  
[www.dis.uniroma1.it/~patrizi](http://www.dis.uniroma1.it/~patrizi)  
[patrizi@dis.uniroma1.it](mailto:patrizi@dis.uniroma1.it)



---

<sup>1</sup>Slides prodotte a partire dal materiale didattico fornito con il testo *Demetrescu, Finocchi, Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione.*

- Docente: Fabio Patrizi
- email: `patrizi@dis.uniroma1.it`
- web: `http://www.dis.uniroma1.it/~patrizi`
- Libro di testo adottato: Demetrescu, Finocchi, Italiano, *Algoritmi e strutture dati*, McGraw-Hill, seconda edizione. 2008.
- Pagina web del corso: `http://www.dis.uniroma1.it/~patrizi/sections/teaching/asd-latina-22-23/index.html`
- Ricevimento: consultare pagina web

## Obiettivi principali:

- Illustrare le tecniche algoritmiche e le strutture dati fondamentali per risolvere in modo efficiente problemi computazionali
- Introdurre gli strumenti fondamentali per valutare la qualità degli algoritmi e delle strutture dati

# Introduzione all'Analisi di Algoritmi

Un **algoritmo** è un insieme di passi semplici che, se eseguiti meccanicamente, producono un risultato **determinato**.

### Esempio

#### Algoritmo preparaCaffè

Svita la caffettiera

Riempi d'acqua il serbatoio della caffettiera

Inserisci il filtro

Riempi il filtro con la polvere di caffè

Avvita la parte superiore della caffettiera

Metti la caffettiera sul fornello

Accendi il fornello

Spegni il fornello quando il caffè è pronto

- Algoritmo: **Procedimento** per risolvere un problema
- Programma: **Implementazione** di un algoritmo

Algoritmo  $\neq$  Programma

Questo corso introduce gli **strumenti di analisi degli algoritmi e delle strutture dati**

Useremo lo **pseudocodice** per definire gli algoritmi

Analizziamo il **procedimento** adottato, non il programma che lo implementa

Vantaggi dell'approccio:

- **Generale**: analisi indipendente da
  - ▶ dettagli implementativi
  - ▶ particolari istanze
- **Economico**: Stima delle prestazioni di un programma **prima** dell'implementazione

Obiettivi principali:

- Valutare la **qualità** di un algoritmo
- **Scegliere** tra più alternative, ove presenti

# Esempio: i numeri di Fibonacci

## Numeri di Fibonacci

Successione di numeri interi  $F_1, F_2, \dots$ , definita come segue:

$$F_n = \begin{cases} 1, & \text{se } 1 \leq n \leq 2 \\ F_{n-1} + F_{n-2}, & \text{se } n > 2 \end{cases}$$

## Problema

Dato un intero positivo  $n$ , calcolare l' $n$ -esimo numero di Fibonacci,  $F_n$



È possibile dimostrare che:

## Formula di Binet, 1843

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

Dove:

- $\phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\dots$
- $\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.6180339887\dots$

Osservazione:

- $\phi$  e  $\hat{\phi}$  irrazionali: **non rappresentabili finitamente**

# L' algoritmo fibonacci\_1

Possiamo usare il seguente algoritmo?

Algoritmo fibonacci\_1(*intero*  $n$ )  $\rightarrow$  *intero*

**return**  $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$

- Purtroppo  $\phi$  e  $\hat{\phi}$  sono *irrazionali*
- Il valore restituito **approssima**  $F_n$  ma **non è quello cercato**

Con  $\phi = 1.618$  e  $\hat{\phi} = -0.618$ , abbiamo:

$n$	fibonacci_1	arrotondamento	$F_n$
3	1.99992	2	2
16	986.698	987	987
18	2583.1	<b>2583</b>	<b>2584</b>

L' algoritmo **non è corretto!**

Vediamo altre possibili soluzioni...

# L'algoritmo fibonacci\_2

Sfruttiamo la definizione ricorsiva di  $F_n$

Algoritmo fibonacci\_2(*intero*  $n$ )  $\rightarrow$  *intero*

```
1: if  $n \leq 2$  then  
2:   return 1  
3: else  
4:   return (fibonacci_2( $n - 1$ ) + fibonacci_2( $n - 2$ ))
```

fibonacci\_2 è corretto

# L'algorithmo fibonacci\_3

Generiamo  $F_1, F_2, \dots, F_n$  iterativamente

Algorithmo fibonacci\_3(*intero*  $n$ )  $\rightarrow$  *intero*

- 1: *Fib*: array di  $n$  interi
- 2:  $Fib[1] \leftarrow 1$
- 3:  $Fib[2] \leftarrow 1$
- 4: **for**  $i \leftarrow 3, n$  **do**
- 5:      $Fib[i] \leftarrow Fib[i - 1] + Fib[i - 2]$
- 6: **return**  $Fib[n]$

Anche fibonacci\_3 è corretto

# Confronto di Algoritmi

Sappiamo che `fibonacci_2` e `fibonacci_3` sono entrambi corretti

- Sono equivalenti o preferiamo uno all'altro?
- Se anche `fibonacci_1` fosse corretto, quale preferiremmo? Perché?

# Costo di un Algoritmo

In presenza di più soluzioni, è conveniente scegliere quella più **economica**, ovvero che usa una quantità inferiore di **risorse**

Le risorse usate da un algoritmo sono:

- Tempo
- Spazio (di memoria)

Come misurare tali risorse?

- Vogliamo una misura che sia *funzione della dimensione dell'input*

# Costo di un Algoritmo

Tempo:

- Secondi?
  - 1 Stiamo valutando l'**algoritmo**, non il programma: non possiamo eseguirlo
  - 2 Dipenderebbe dal compilatore, dalla macchina, dalle istanze di input

Spazio:

- Memoria utilizzata?
  - ▶ Considerazioni simili a quanto detto sopra

# Modello di Costo

Adottiamo le seguenti misure:

- Tempo: numero di **operazioni elementari** eseguite
- Spazio: dimensione complessiva delle **strutture dati necessarie** a memorizzare i valori usati dall'algoritmo

Consideriamo come **operazioni elementari** (dette anche *passi base*):

- allocazioni di variabile e assegnazioni a variabile
- valutazioni di espressione e test di condizioni booleane
- restituzione del risultato

Consideriamo come **strutture dati**:

- variabili: dimensione unitaria
- insiemi, liste, pile, code, vettori, etc.: dimensione pari alla cardinalità
- record di attivazione (per funzioni ricorsive): dimensione pari ad 1 più dimensione delle strutture dati nel record



Le misure adottate trascurano vari dettagli:

- Il costo della valutazione delle espressioni dipende dal numero di valori e di operatori coinvolti
- I valori hanno un impatto sul costo della valutazione di espressioni
- Analoghe considerazioni valgono per le condizioni booleane
- Lo spazio necessario a memorizzare un valore dipende dal valore stesso
- Bisognerebbe distinguere il caso di valori interi da valori reali
- ...

Tuttavia, il **modello di costo** scelto consente un'accuratezza sufficiente a valutare l'andamento di un algoritmo al crescere della dimensione dell'input ed a confrontare algoritmi diversi

# Funzione di Costo

Valutiamo il costo di un algoritmo rispetto ad una risorsa (tempo o spazio) in funzione della **dimensione  $n$  dell'input**

Ovvero, cerchiamo una funzione che, data la dimensione  $n$  dell'istanza del problema, fornisca la quantità di risorsa utilizzata dall'algoritmo

Indichiamo con  $T(n)$  il tempo (numero di passi base) richiesto dall'algoritmo

Indichiamo con  $S(n)$  lo spazio (unità di memoria) usato dall'algoritmo

NOTA: Le funzioni di costo sono *sempre positive*

Confronteremo gli algoritmi sulla base del loro **costo asintotico**, ovvero al crescere della dimensione dell'input (dettagli in seguito)

## Algoritmo fibonacci\_2: Tempo di Esecuzione

Algoritmo fibonacci\_2(*intero*  $n$ )  $\rightarrow$  *intero*

```
1: if  $n \leq 2$  then  
2:   return 1  
3: else  
4:   return (fibonacci_2( $n - 1$ ) + fibonacci_2( $n - 2$ ))
```

Tempo di esecuzione:

- se  $n \leq 2$ : test (linea 1) e restituzione del risultato (l.2)
- se  $n > 2$ : test (l.1), due chiamate ricorsive, somma e restituzione (l.4)

Le chiamate ricorsive non sono operazioni elementari:

- costo:  $T(n - 1)$  e  $T(n - 2)$

Il tempo d'esecuzione soddisfa la seguente relazione:

$$T(n) = \begin{cases} 2, & \text{se } n \leq 2 \\ 3 + T(n - 1) + T(n - 2), & \text{se } n > 2 \end{cases}$$

# Relazioni di ricorrenza

L'equazione

$$T(n) = 3 + T(n - 1) + T(n - 2)$$

è detta **relazione (o equazione) di ricorrenza**

Per conoscere  $T(n)$  occorre risolvere l'equazione, ovvero trovare un'espressione che, sostituita a  $T(n)$ , soddisfi la relazione (metodi di risoluzione in seguito)

È possibile dimostrare che:  $T(n) = 5F_n - 3$

Quindi, essendo  $F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$

$$T(n) = \frac{5}{\sqrt{5}}(\phi^n - \hat{\phi}^n) - 3$$

(Andamento esponenziale in  $n$ )

## Algoritmo fibonacci\_3: Tempo di Esecuzione

Algoritmo fibonacci\_3(*intero*  $n$ )  $\rightarrow$  *intero*

- 1:  $Fib$ : array di  $n$  interi
- 2:  $Fib[1] \leftarrow 1$
- 3:  $Fib[2] \leftarrow 1$
- 4: **for**  $i \leftarrow 3, n$  **do**
- 5:      $Fib[i] \leftarrow Fib[i - 1] + Fib[i - 2]$
- 6: **return**  $Fib[n]$

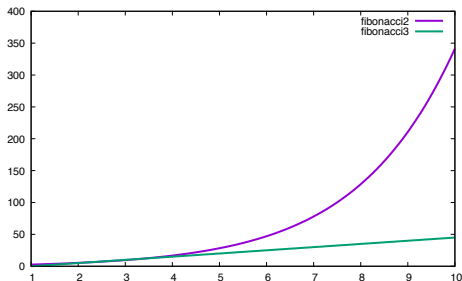
$$T(n) = \begin{cases} n + 5, & \text{se } n < 3 \\ n + 2 + 4(n - 2) + 1 = 5n - 5, & \text{altrimenti} \end{cases}$$

(Andamento lineare)

# fibonacci\_2, fibonacci\_3: confronto

Consideriamo solo il tempo di esecuzione

Algoritmo	$T(n)$
fibonacci_2	$\frac{5}{\sqrt{5}}(\phi^n - \hat{\phi}^n) - 3$
fibonacci_3	$5n - 5$



Quale algoritmo preferiamo?

L'algoritmo che esegue “meno operazioni” è chiaramente preferibile

Tuttavia, un algoritmo potrebbe andare meglio di un altro solo per alcuni valori di input. Quale scegliere?

Osservazioni:

- Per valori piccoli dell'input, le differenze di costo sono poco apprezzabili
- Normalmente, il tempo d'esecuzione cresce con la dimensione dell'input, quindi è preferibile l'algoritmo con funzione di costo a crescita meno rapida

Siamo cioè interessati all'**andamento asintotico** delle funzioni di costo



## Algoritmo fibonacci\_2: Occupazione di Memoria

Valutiamo ora il costo in termini di spazio di `fibonacci_2` e `fibonacci_3`

Si considera solo il **costo aggiuntivo** rispetto alla dimensione dell'input (in quanto è su questa grandezza che vengono confrontati due algoritmi che prendono lo stesso input)

Algoritmo `fibonacci_2(intero n) → intero`

```
1: if  $n \leq 2$  then  
2:   return 1  
3: else  
4:   return (fibonacci_2( $n - 1$ ) + fibonacci_2( $n - 2$ ))
```

Sembrerebbe che non vengano usate variabili aggiuntive rispetto ad  $n$

Tuttavia, per funzioni ricorsive, occorre considerare lo stack delle chiamate

Per un'invocazione di `fibonacci_2` su input  $n$ , si può dimostrare (dettagli in seguito) che la dimensione massima dello stack delle chiamate è:

$$S(n) = \begin{cases} 1, & \text{se } n \leq 2 \\ n - 1, & \text{se } n > 2 \end{cases}$$

# Algoritmo fibonacci\_3: Occupazione di Memoria

Algoritmo fibonacci\_3(*intero*  $n$ )  $\rightarrow$  *intero*

```
1: Fib: array di  $n$  interi
2: Fib[1]  $\leftarrow$  1
3: Fib[2]  $\leftarrow$  1
4: for  $i \leftarrow 3, n$  do
5:   Fib[ $i$ ]  $\leftarrow$  Fib[ $i - 1$ ] + Fib[ $i - 2$ ]
6: return Fib[ $n$ ]
```

Usiamo:

- un vettore di dimensione  $n$
- una variabile contatore (di dimensione unitaria)

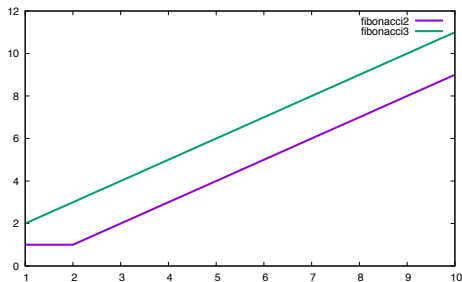
Abbiamo quindi:

$$S(n) = n + 1$$

# fibonacci\_2, fibonacci\_3: confronto

Consideriamo ora lo spazio di memoria

Algoritmo	$S(n)$
fibonacci_2	$n - 1$
fibonacci_3	$n + 1$



Quale algoritmo preferiamo?

## fibonacci\_2, fibonacci\_3: confronto

`fibonacci_2` sembra preferibile

Tuttavia:

- l'andamento delle funzioni di costo (spaziale) è essenzialmente identico a meno di due unità
- dobbiamo considerare le semplificazioni del modello di costo, che non rendono conto di piccole differenze (formalizzeremo questa idea in seguito)

È lecito quindi considerare i due costi spaziali sostanzialmente equivalenti

Concludiamo pertanto che `fibonacci_3` è preferibile, in quanto:

- ha migliori prestazioni in termini di tempo
- è comparabile a `fibonacci_2` in termini di occupazione di memoria

# L'algoritmo fibonacci\_4

Possiamo ridurre l'occupazione di memoria di fibonacci\_3?

Algoritmo fibonacci\_4(*intero*  $n$ )  $\rightarrow$  *intero*

$a, b, c$ : variabili intere

$b \leftarrow 1$

$c \leftarrow 1$

**for**  $i \leftarrow 3, n$  **do**

$a \leftarrow b$

$b \leftarrow c$

$c \leftarrow a + b$

**return**  $c$

## fibonacci\_3, fibonacci\_4: confronto

`fibonacci_4` usa un numero costante di variabili (3), non un numero crescente con l'input (`fibonacci_3`:  $n + 1$ )

È immediato vedere che `fibonacci_4` ha costo temporale  $T(n) = 6n - 6$ , essenzialmente analogo a quello di `fibonacci_3` (le costanti moltiplicative possono essere trascurate per le approssimazioni del modello di costo –dettagli in seguito)

`fibonacci_4` è pertanto preferibile

Possiamo migliorare il tempo d'esecuzione di fibonacci\_4?

Sfruttiamo la seguente proprietà, facilmente dimostrabile per induzione

Proprietà

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Cioè, moltiplicando la matrice  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  per se stessa  $n$  volte, troviamo l' $(n + 1)$ -esimo numero di Fibonacci in posizione  $(0, 0)$



Consideriamo il seguente algoritmo

Algoritmo `fibonacci_5`(*intero*  $n$ )  $\rightarrow$  *intero*

$M$ : matrice  $2 \times 2$  // 4 passi base

$M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$  // 4 passi base

**for**  $i \leftarrow 1, n - 1$  **do**

$M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  // 8 passi base

**return**  $M[0][0]$  // 1 passo base

Tempo di esecuzione:  $T(n) = 9 + 8(n - 1) = 9n + 1$

Abbiamo un tempo di esecuzione essenzialmente analogo a quello di `fibonacci_4` ( $6n - 6$ )

Possiamo ridurre il numero di iterazioni per la potenza di matrice

## Quadrati ripetuti

$$M^n = \begin{cases} P \cdot P, & \text{se } n \text{ è pari} \\ P \cdot P \cdot M, & \text{altrimenti} \end{cases}, \text{ con } P = M^{\lfloor \frac{n}{2} \rfloor}$$

Nota:  $P$  viene calcolata una volta sola

# Calcolo di potenze

Possiamo calcolare la potenza di una matrice con la seguente procedura (effettua side-effect su  $M$ )

Procedura potenza(*matrice*  $2 \times 2$   $M$ , *intero*  $n$ )

**if** ( $n > 1$ ) **then**

    potenza( $M$ ,  $\lfloor \frac{n}{2} \rfloor$ )

$M \leftarrow M \cdot M$

**if** ( $n$  dispari) **then**  $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Con tempo d'esecuzione (assumendo 4 operazioni per il prodotto):  
 $T(n) = 18 + T(\frac{n}{2}) \approx 18 \log_2 n$  (metodo di soluzione in seguito)

E occupazione di memoria:  $S(n) = \log_2 n$

Algoritmo fibonacci\_6(*intero*  $n$ )  $\rightarrow$  *intero*

$M$ : matrice  $2 \times 2$

$M \leftarrow I$  // matrice identità

potenza( $M, n - 1$ )

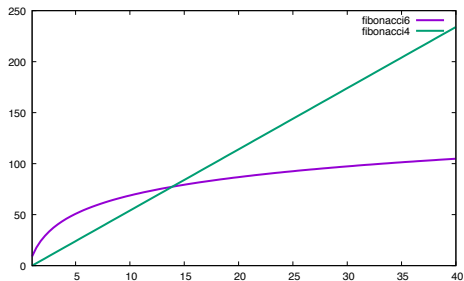
**return**  $M[0][0]$

Tempo di esecuzione:  $T(n) = 9 + 18 \log_2(n - 1)$

Occupazione di memoria:  $S(n) = 4 + \log_2 n$

# fibonacci\_4, fibonacci\_6: confronto

Algoritmo	$T(n)$
fibonacci_4	$6n - 6$
fibonacci_6	$9 + 18 \log_2(n - 1)$



# Algoritmi a Confronto

Algoritmo	Tempo di esecuzione	Occupazione di memoria
fibonacci_2	$\frac{5}{\sqrt{5}}(\phi^n - \hat{\phi}^n) - 3$	$n - 1$
fibonacci_3	$5n - 5$	$n + 1$
fibonacci_4	$6n - 6$	4
fibonacci_5	$9n + 1$	4
fibonacci_6	$9 + 18 \log_2(n - 1)$	$4 + \log_2 n$

Concludiamo che fibonacci\_6 è (asintoticamente) l'algoritmo più veloce tra quelli considerati

fibonacci\_4 e fibonacci\_5 sono invece gli algoritmi con minor occupazione di memoria

- Le funzioni di costo ottenuto non descrivono fedelmente i costi degli algoritmi ma l'andamento rispetto alla dimensione dell'input
- Causa: i dettagli trascurati dal modello di costo rendono irrilevanti differenze di costo per valori costanti (anche all'interno di cicli)
- Pertanto, è ragionevole trascurare costanti additive e moltiplicative
- I tempi d'esecuzione di fibonacci\_3, fibonacci\_4 e fibonacci\_5 sono sostanzialmente analoghi: andamento lineare
- fibonacci\_2 è invece più costoso: andamento esponenziale
- fibonacci\_6 è il più veloce: andamento logaritmico