

# INTRODUZIONE AGLI ALGORITMI

PIETRO DI LENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
UNIVERSITÀ DI BOLOGNA

ALGORITMI E STRUTTURE DI DATI  
ANNO ACCADEMICO 2022/2023



# INTRODUZIONE

- Cos'è un algoritmo?
  - Procedura per risolvere un problema in un numero **finito** di passi
- Qual è l'origine della parola "algoritmo"?
  - Deriva dal nome del matematico Persiano **Muhammad ibn Musa al-Khwarizmi**
  - **al-Khwarizmi** latinizzato in **Algoritmi**
  - Autore di **Algoritmi de numero Indorum** (820), responsabile della diffusione del sistema numerico Hindu-Arabico in Medio Oriente ed Europa



Musa al-Khwarizmi  
(c. 780 – c. 850)

# ALGORITMO VS PROGRAMMA

- Un **algoritmo** è una **descrizione di alto livello** di una procedura
  - Gli algoritmi esistono da prima dell'avvento dei computer
  - **Non può essere eseguito** su un computer
  - Può assumere un quantitativo **illimitato** di memoria
- Un **programma** è l'**implementazione** di un algoritmo
  - Deve essere scritto in qualche linguaggio di programmazione
  - **Può essere eseguito** su un computer
  - Deve tenere conto dei limiti di memoria del computer

# PROSPETTIVA STORICA (INCOMPLETA)

- Euclide ( $\sim 300$  AC), Eratostene ( $\sim 200$  AC)
  - calcolo aritmetico, geometrico
- al-Khwarizmi ( $\sim 800$ )
  - calcolo aritmetico, algebrico, geometrico
- Newton ( $\sim 1700$ )
  - calcolo differenziale, integrale
- Turing ( $\sim 1940$ )
  - crittoanalisi, risolubilità

# INGREDIENTI DI UN ALGORITMO

- Un algoritmo:
  - prende in **input** alcuni valori
  - esegue una sequenza **finita** di **istruzioni**
  - produce (eventualmente) un **output**
- Un algoritmo rappresenta la soluzione ad un **problema** da risolvere
  - La definizione del problema vincola input e output
  - Ci sono infiniti set di istruzioni che risolvono lo stesso problema
    - Esattamente lo stesso output per lo stesso input

# ESEMPIO: ALGORITMO DI EUCLIDE

Calcolare il Massimo Comune Divisore (MCD) tra  $x$  e  $y$  ( $x \geq y$ )

- 1 Dividi  $x$  per  $y$  e chiama  $r$  il resto della divisione
- 2 Se  $r = 0$  allora  $y$  è il MCD tra  $x$  e  $y$
- 3 Se  $r \neq 0$  allora assegna  $y$  ad  $x$ ,  $r$  ad  $y$  e ritorna al punto 1

```
1: function MCD(INT  $x$ , INT  $y$ )  $\rightarrow$  INT  
2:   if  $y == 0$  then  
3:     return  $x$   
4:   else  
5:      $r = x \bmod y$   
6:     return MCD( $y, r$ )
```

- Algoritmo di Euclide in **pseudo-codice**: linguaggio umano sulla sinistra, linguaggio più vicino a quello di un calcolatore sulla destra
- Nota: assumiamo che  $x \geq y$  ma possiamo rilassare tale assunzione modificando di poco il nostro pseudo-codice

# COSA CI SERVE PER POTER SVILUPPARE ALGORITMI?

- **Capire** il problema che vogliamo risolvere
  - Quali sono i possibili input e output?
  - Come sono gli input mappati sugli output?
  - Ci sono proprietà matematiche legate al nostro problema?
- **Apprendere** come stimare l'efficienza di un algoritmo
  - Tempo: quanto è veloce un algoritmo?
  - Memoria: di quanta memoria ha bisogno?
  - Come possiamo individuare l'algoritmo migliore tra due scelte?
- **Studiare** tecniche algoritmiche e strutture dati note
  - Problemi differenti spesso condividono la stessa struttura di base
  - Molti problemi possono essere risolti modificando algoritmi noti
  - Come possiamo essere sicuri che la nostra soluzione sia corretta?

# PROBLEMA ALGORITMICO: I NUMERI DI FIBONACCI

- I numeri di Fibonacci sono una sequenza

$$F_1 = 1, F_2, \dots, F_n, \dots$$

in cui ogni numero è dato dalla somma dei due numeri precedenti

$$F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \dots$$

- I numeri di Fibonacci possono essere definiti da una relazione ricorsiva

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ o } n = 2 \\ F_{n-1} + F_{n-2} & \text{altrimenti} \end{cases}$$

- Tipicamente si assume che  $F_0 = 0$



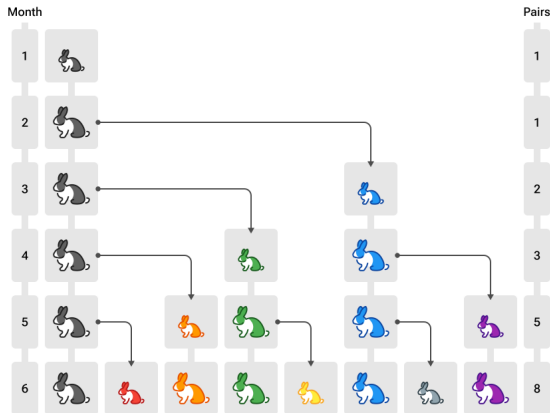
Leonardo Fibonacci  
(c. 1170 – c. 1250)



# COSA RAPPRESENTA LA SEQUENZA DI FIBONACCI?

Crescita (idealizzata) di una popolazione di conigli:

- Ogni coppia diventa fertile dopo un mese di vita
- A partire dal secondo mese la coppia dà alla luce una nuova coppia
- I conigli sono immortali



# ALGORITMO 1: FORMULA CHIUSA

- **Buona notizia:** esiste una formula chiusa per calcolare il valore  $F_n$

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

dove

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \text{ e } \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

```
1: function FIB1(INT  $n$ )  $\rightarrow$  INT
2:   return  $\frac{1}{\sqrt{5}} \left( \phi^n - \hat{\phi}^n \right)$   $\triangleright$  arrotondato ad int
```

- **Cattiva notizia:** approssimazioni numeriche per l'irrazionalità di  $\phi$  e  $\hat{\phi}$ 
  - Non è possibile calcolare esattamente  $F_n$  su un computer
  - Es.  $FIB1(3) = 1.999863 \approx 2 = F_3$
  - Es.  $FIB1(18) = 2583.023 \approx 2583 \neq F_{18} = 2584$
  - Aumentare la precisione di  $\phi$  e  $\hat{\phi}$  non aiuta
- Dobbiamo pensare ad un'altra soluzione

## ALGORITMO 2: SOLUZIONE RICORSIVA “INGENUA”

- Possiamo convertire direttamente la formula ricorsiva

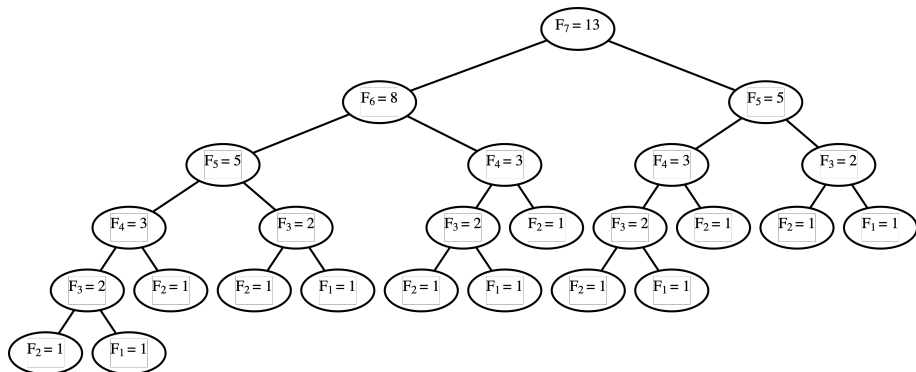
$$F_n = \begin{cases} 1 & n \leq 2 \\ F_{n-1} + F_{n-2} & n > 2 \end{cases}$$

in un algoritmo ricorsivo

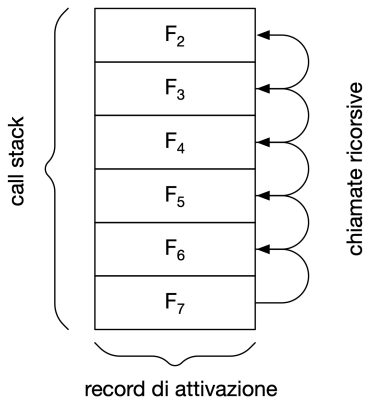
```
1: function FIB2(INT  $n$ )  $\rightarrow$  INT  
2:   if  $n \leq 2$  then  
3:     return 1  
4:   else  
5:     return FIB2( $n - 1$ ) + FIB2( $n - 2$ )
```

- Quanta memoria richiede FIB2?
- Quanto è veloce FIB2?

# ALGORITMO 2: ALBERO DI RICORSIONE DI FIBONACCI

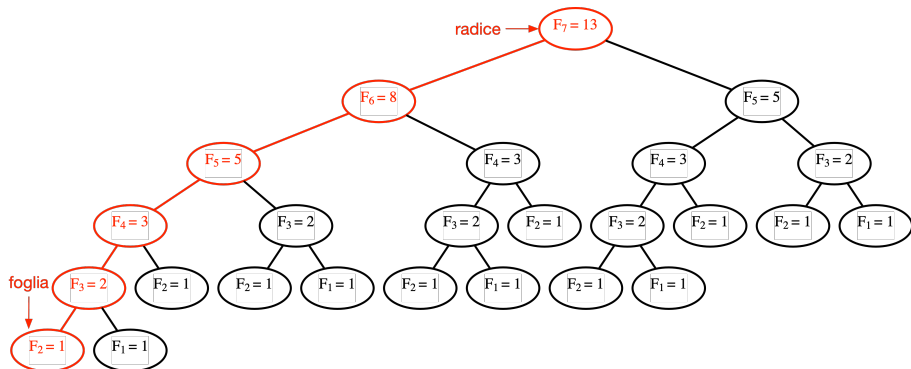


## ALGORITMO 2: ATTIVAZIONE CHIAMATE RICORSIVE



Ogni chiamata ricorsiva causa l'allocazione di un record di attivazione sul call stack

# ALGORITMO 2: STIMA DELLA MEMORIA

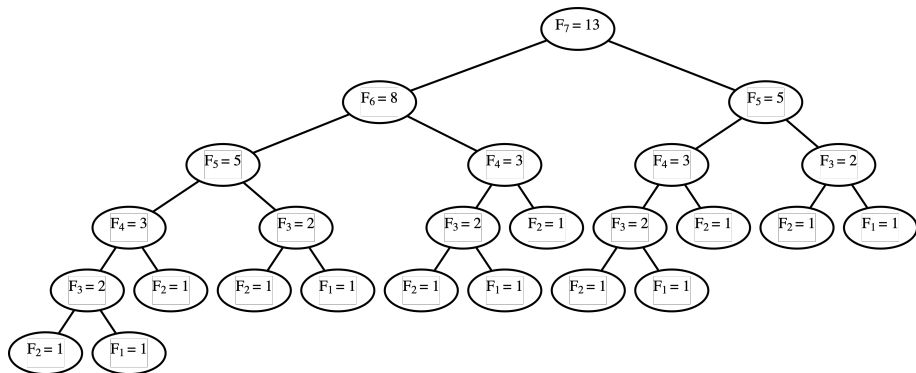


- Apparentemente, FIB2 richiede memoria solo per memorizzare  $n$
- In realtà, le chiamate ricorsive sono gestite dal *call stack*
  - la memoria al tempo  $t$  dipende dalle **chiamate attive** al tempo  $t$
- La memoria usata è **proporzionale** ad  $n$  (input) nel caso peggiore
  - Il numero **massimo** possibile di **chiamate attive** in FIB2 è limitato dal percorso *radice-foglia* più lungo nell'albero di ricorsione

## ALGORITMO 2: STIMA DEL TEMPO DI CALCOLO

- Dovremmo misurare il tempo di calcolo in **secondi**?
  - Dipende dalla macchina su cui eseguiamo il programma
- Dovremmo contare il numero di **istruzioni-macchina** nell'algoritmo?
  - Stesso problema: dipende dal linguaggio di programmazione
  - Difficile stimare il numero di istruzioni-macchina da pseudocodice
- Approccio indipendente dalla macchina e dal linguaggio:
  - contiamo il numero di **operazioni elementari** nello pseudocodice

## ALGORITMO 2: STIMA DEL TEMPO DI CALCOLO



- Idea: stimiamo il numero di chiamate ricorsive

$T(n)$  = numero di nodi nell'albero di ricorsione di  $F_n$

- Ogni chiamata ricorsiva richiede alla peggio un numero **costante** di **operazioni elementari**: una somma, due chiamate ricorsive, istruzione di return (stesso costo per ogni chiamata)



## ALGORITMO 2: STIMA DEL TEMPO DI CALCOLO

```
1: function FIB2(INT  $n$ )  $\rightarrow$  INT
2:   if  $n \leq 2$  then
3:     return 1
4:   else
5:     return FIB2( $n - 1$ ) + FIB2( $n - 2$ )
```

- Domanda: come trovare un limite superiore/inferiore per  $T(n)$ ?
- Risposta: estraiamo una relazione di ricorrenza dallo pseudocodice

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

L'albero  $T(n)$  contiene 1 nodo + i nodi dei sotto-alberi  $T(n-1), T(n-2)$

## ALGORITMO 2: LIMITE INFERIORE PER $T(n)$

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

$$\geq 2T(n-2) + 1$$

Poiché  $T(n-1) \geq T(n-2)$

$$\geq 4T(n-4) + 2 + 1$$

Poiché  $T(n-2) \geq 2T(n-4) + 1$

$$\geq 8T(n-6) + 4 + 2 + 1$$

Poiché  $T(n-4) \geq 2T(n-6) + 1$

$$\geq \dots$$

$$\geq 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i$$

Generalizzazione al k-esimo passo

$$\geq \dots$$

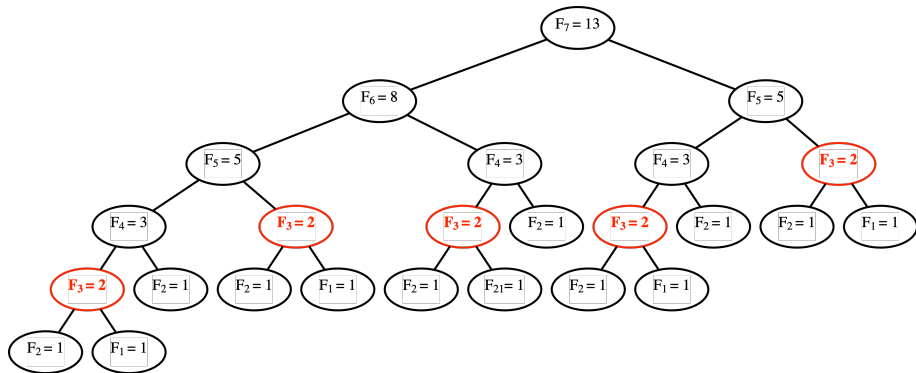
Stop quando  $n-2k \leq 2 \Rightarrow k = \lfloor n/2 \rfloor \geq n/2 - 1$

$$\geq 2^{\lfloor n/2 \rfloor} + \frac{2^{\lfloor n/2 \rfloor} - 1}{2 - 1}$$

$$\geq 2^{\lfloor n/2 \rfloor}$$

Soluzione:  $T(n)$  ha un numero esponenziale di nodi maggiore di  $2^{n/2}$

## ALGORITMO 2: QUAL È IL PROBLEMA?



- La maggior parte dei numeri sono ri-calcolati più di una volta
  - La ricorsione non ricorda i numeri precedentemente calcolati
- Possiamo fare di meglio?

## ALGORITMO 3: SOLUZIONE ITERATIVA

- La ricorsione non è una buona idea, cerchiamo una soluzione **iterativa**
- Idea: usiamo un array di lunghezza  $n$  per memorizzare  $F_1, F_2, \dots, F_n$

```
1: function FIB3(INT  $n$ )  $\rightarrow$  INT  
2:   LET  $F[1, \dots, n]$  BE A NEW ARRAY OF INT  
3:    $F[1] = 1$   
4:    $F[2] = 1$   
5:   for  $i = 3, \dots, n$  do  
6:      $F[i] = F[i - 1] + F[i - 2]$   
7:   return  $F[n]$ 
```

- Quanto è veloce FIB3?
- Quanta memoria richiede FIB3?

# ALGORITMO 3: STIMA DEL TEMPO E MEMORIA

## ■ Cerchiamo di stimare

- **tempo di calcolo**: contiamo il numero di operazioni elementari
- **memoria usata**: sommiamo la dimensione delle variabili usate

```
1: function FIB3(INT  $n$ )  $\rightarrow$  INT
2:   LET  $F[1, \dots, n]$  BE A NEW ARRAY OF INT       $\triangleright 1$  volta (creazione)
3:    $F[1] = 1$                                         $\triangleright 1$  volta (assegnamento)
4:    $F[2] = 1$                                         $\triangleright 1$  volta (assegnamento)
5:   for  $i = 3, \dots, n$  do                         $\triangleright n-2$  volte (assegnamento)
6:      $F[i] = F[i-1] + F[i-2]$      $\triangleright n-2$  volte (assegnamento+somma)
7:   return  $F[n]$                                    $\triangleright 1$  volta (return)
```

## ■ Tempo di calcolo *proporzionale ad $n$*

- Totale:  $4 + 3(n - 2) = 3n - 2$  operazioni elementari (per  $n > 2$ )

## ■ Memoria *proporzionale ad $n$* (array $F$ e variabili $i, n$ )

- Abbiamo davvero bisogno di utilizzare un array di lunghezza  $n$ ?

## ALGORITMO 4: SOLUZIONE EFFICIENTE IN MEMORIA

- Idea: per calcolare  $F_n$  possiamo memorizzare unicamente  $F_{n-1}$  e  $F_{n-2}$

```
1: function FIB4(INT  $n$ )  $\rightarrow$  INT  
2:    $a = 1$   
3:    $b = 1$   
4:   for  $i = 3, \dots, n$  do  
5:      $c = a + b$   
6:      $a = b$   
7:      $b = c$   
8:   return  $b$ 
```

- Quanto è veloce FIB4?
- Quanta memoria richiede FIB4?

# ALGORITMO 4: STIMA DEL TEMPO E DELLA MEMORIA

- Come prima, contiamo il numero di operazioni elementari e variabili

1: <b>function</b> FIB4(INT $n$ ) $\rightarrow$ INT	
2: $a = 1$	▷ 1 volta (assegnamento)
3: $b = 1$	▷ 1 volta (assegnamento)
4: <b>for</b> $i = 3, \dots, n$ <b>do</b>	▷ $n-2$ volte (assegnamento)
5: $c = a + b$	▷ $n-2$ volte (assegnamento+somma)
6: $a = b$	▷ $n-2$ volte (assegnamento)
7: $b = c$	▷ $n-2$ volte (assegnamento)
8: <b>return</b> $b$	▷ 1 volta (return)

- Tempo di calcolo proporzionale ad  $n$

- Totale:  $3 + 5(n - 2) = 5n - 7$  operazioni (più lento di FIB3?)

- Memoria costante

- Usiamo cinque variabili ( $n, i, a, b, c$ ), indipendentemente da  $n$

# POSSIAMO RITENERCI SODDISFATTI?

## Teorema

Consideriamo  $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ . Per ogni  $n \geq 2$  abbiamo che

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

**Dimostrazione** (per induzione su  $n$ )

- 1 Caso base. Per  $n = 2$ ,  $A^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$  è vera
- 2 Caso induttivo. Assumiamo che per qualche  $k \geq 2$  sia vero che

$$A^{k-1} = \begin{pmatrix} F_k & F_{k-1} \\ F_{k-1} & F_{k-2} \end{pmatrix}$$

allora dobbiamo mostrare che la proprietà è vera per  $A^k = A^{k-1} \times A$

$$A^k = \begin{pmatrix} F_k & F_{k-1} \\ F_{k-1} & F_{k-2} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_k + F_{k-1} & F_k \\ F_{k-1} + F_{k-2} & F_{k-1} \end{pmatrix} = \begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix}$$



# INDUZIONE MATEMATICA

- L'induzione matematica è una tecnica di **dimostrazione matematica**
- Consideriamo una **proprietà**  $P$  sui numeri naturali  $\mathbb{N}$
- L'induzione matematica dice che  $P$  vale per tutti i numeri naturali se
  - 1 **Caso base**:  $P$  è vera per il valore iniziale (es. 0)
  - 2 **Caso induttivo**: assumendo che  $P$  sia vera per valori  $\leq n$  possiamo dimostrare che  $P$  è vera per  $n + 1$
- Non dobbiamo necessariamente considerare tutti i numeri naturali
  - Esempio: la proprietà  $P$  è vera per tutti i valori  $n \geq 2$
- Possiamo avere più di una assunzione nel caso induttivo
  - Esempio: assumendo che  $P$  sia vera per  $n$  ed  $n + 1$  dimostriamo che  $P$  è vera per  $n + 2$
  - In tal caso, bisogna mostrare anche che  $P$  è vera per più casi base

## ALGORITMO 5: SOLUZIONE CON MATRICI

- Idea: usiamo il Teorema precedente

```
1: function FIB5(INT  $n$ )  $\rightarrow$  INT  
2:    $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$   
3:   for  $i = 2, \dots, n$  do  
4:      $A = A \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$   $\triangleright \times$  indica il prodotto di matrici  
5:   return  $A[1][1]$ 
```

- Tempo di calcolo proporzionale ad  $n$

- $n - 1$  moltiplicazioni di matrici +  $2n - 1$  assegnamenti + 1 return

- Memoria costante

- Usiamo 3 variabili ( $A, i, n$ ), indipendentemente da  $n$

- Nessun miglioramento rispetto a FIB4 ma ...

# POTENZA DI MATRICI VELOCIZZATA

- Metodo rapido per esponenziazione: **exponentiation by squaring**

$$A^n = \begin{cases} (A^{n/2})^2 & \text{se } n \text{ pari} \\ A \cdot (A^{(n-1)/2})^2 & \text{se } n \text{ dispari} \end{cases}$$

```
1: function FIBMATPOW(INT  $n$ )  $\rightarrow$  FIBMAT
2:    $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 
3:   if  $n > 1$  then
4:      $M = \text{FIBMATPOW}(n/2)$             $\triangleright n/2$  divisione intera
5:      $A = M \times M$ 
6:     if  $n \bmod 2 \neq 0$  then            $\triangleright n$  dispari
7:        $A = A \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 
8:   return  $A$ 
```

- Tempo di calcolo **proporzionale a  $\log_2 n$**  (lo vedremo più avanti)
  - Per calcolare  $A^n$  eseguiamo  $\approx \log_2 n$  chiamate ricorsive ( $2^i = n$ )
- Memoria **proporzionale a  $\log_2 n$** 
  - Memoria costante  $(A, M, n)$  per ognuna delle  $\log n$  chiamate

## ALGORITMO 6: SOLUZIONE CON POTENZA VELOCE

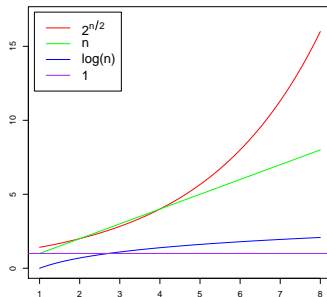
- Idea: usiamo l'algoritmo velocizzato per la potenza di matrici

```
1: function FIB6(INT  $n$ )  $\rightarrow$  INT  
2:    $M = \text{FIBMATPOW}(n - 1)$   
3:   return  $M[1][1]$ 
```

- Tempo di calcolo proporzionale a  $\log_2 n$ 
  - Equivalente al tempo di calcolo di FIBMATPOW
- Memoria proporzionale a  $\log_2 n$ 
  - Occupazione di memoria di FIBMATPOW

- Tempo di calcolo e occupazione di memoria in **notazione asintotica**

Algoritmo	Tempo	Memoria
Fib2	$\Omega(2^{n/2})$	$O(n)$
Fib3	$O(n)$	$O(n)$
Fib4	$O(n)$	$O(1)$
Fib5	$O(n)$	$O(1)$
Fib6	$O(\log n)$	$O(\log n)$



- Siamo partiti da un algoritmo con tempo esponenziale di calcolo (inutilizzabile) e siamo arrivati ad un algoritmo logaritmico (efficiente)
- Benché tutti gli algoritmi sono equivalenti in termini di risultato, la loro efficienza computazionale li rende utilizzabili o meno
- Vedremo meglio la notazione asintotica, per il momento diciamo che ..

# VALORE IN INPUT VS DIMENSIONE DELL'INPUT

- Abbiamo stimato l'efficienza rispetto al **valore in input**  $n$
- Normalmente valutiamo l'efficienza in termini di **dimensione dell'input**
- La dimensione dell'input è il **numero di bit** sufficienti per rappresentarlo

$$|n| = \lceil \log_2 n \rceil \approx \log_2 n \Rightarrow n \approx 2^{|n|}$$

- Tempi di calcolo e memoria in termini della dimensione dell'input

Algorithm	Time	Space
Fib2	$\Omega(2^{2^{ n }})$	$O(2^{ n })$
Fib3	$O(2^{ n })$	$O(2^{ n })$
Fib4	$O(2^{ n })$	$O(1)$
Fib5	$O(2^{ n })$	$O(1)$
Fib6	$O( n )$	$O( n )$

- **Nota.** Il tempo di calcolo (in sec) non cambia, cambia solo come valutiamo in tempo di calcolo di un algoritmo: in termini della **dimensione** dell'input e non del **valore** in input