

# INGEGNERIA DEGLI ALGORITMI

## 1. INTRODUZIONE

### • Definizioni di base:

- **STRUTTURA DATI**: è un'organizzazione sistematica dei dati e del loro accesso, che facilita le manipolazioni;

- **ALGORITMO**: deriva da Muhammad al-Khwarizmi che ha scritto l'"al-Jabar", in cui era contenuta la formula  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ .

Un ALGORITMO è una procedura suddivisa in passi elementari che, eseguiti in sequenza, consentono di risolvere un problema in tempo finito.

Esso è uno STRUMENTO per risolvere un **PROBLEMA COMPUTAZIONALE**.

E' corretto se, per ogni istanza di input, l'algoritmo termina con l'output corretto.

Non è corretto se esiste un'istanza di input per cui l'output non è corretto oppure l'algoritmo non termina.

- **PROBLEMA COMPUTAZIONALE**: Dati un dominio di input ed un dominio di output, un PROBLEMA COMPUTAZIONALE è rappresentato dalla **RELAZIONE MATEMATICA** che associa un elemento del dominio di output ad ogni elemento del dominio di input.  
(Risoluzione in forme chiuse).

### • PROBLEMA: sottovettore di somme massime.

- **INPUT**: un vettore di interi  $A[1..n]$

- **OUTPUT**: il sottovettore  $A[i..j]$  di somme massime, ovvero il sottovettore le cui somme degli elementi  $\sum_{k=i}^j A[k]$  è più grande o uguale alle somme degli elementi di qualunque altro sottovettore.

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2
---	---	---	----	---	---	----	---	---	----	----	----	---

- Soluzione naïf:  $O(n^3)$

Cicla su tutte le coppie  $(i, j)$  tali che  $i \leq j$ :

Calcola le somme dei valori compresi tra  $A[i]$  e  $A[j]$   
aggiorna maxSoFar con il massimo fra la somma appena trovata ed il massimo fino a quel punto.

```
def maxsum1(A):
    maxSoFar = 0 # Maximum found so far
    for i in range(0, len(A)):
        for j in range(i, len(A)):
```

$$\text{maxSoFar} = \max(\text{maxSoFar}, \text{sum}(A[i:j]))$$

return maxSoFar

## - Prima Otimizzazione : $O(m^2)$

INTUZIONE: se ho calcolato la somma  $s$  dei valori in  $A[i \dots j]$ , la somma dei valori in  $A[i \dots j+1]$  è pari ad  $s + A[j+1]$ .

def maxsum2(A):

    maxSoFar = 0     # Maximum found so far

    for i in range(0, len(A)):

        Sum = 0     # Accumulator

        for j in range(i, len(A)):

            sum = sum + A[j]

            maxSoFar = max(maxSoFar, sum)

    return maxSoFar

## - Seconda Otimizzazione : $O(m \log m)$

DIVIDE ET IMPERA: dividiamo il vettore nelle metà di destra e sinistra, in due parti più o meno uguali;

maxL = somma massimale delle parte sinistra;

maxR = somma massimale delle parte destra;

maxLL + maxRR = valore delle sottoliste massimali al centro;

Restituisce il massimo tra questi 3 valori. È ovviamente una funzione RICORSIVA.

def maxsum\_rec(A, i, j):

    if (i == j):

        return max(0, A[i])

    m = (i+j)/2

        # Maximal subvector on the left ending in m

    maxLL = 0

    sum = 0

    for k in range(m, i-1, -1):     # Going back from m to i

        sum += A[k]

        maxLL = max(maxLL, sum)

    maxRR = 0     # Maximal subvector on the right starting in m+1

    sum = 0

    for k in range(m+1, j+1):

        sum += A[k]

        maxRR = max(maxRR, sum)

    maxL = maxsum\_rec(A, 0, m)     # Maximal subvector on the left

    maxR = maxsum\_rec(A, m+1, j)     # Maximal subvector on the right

    return max(maxL, maxR, maxLL + maxRR)

def maxsum3(A):

    return maxsum\_rec(A, 0, len(A)-1)

### - Terza Ottimizzazione: $O(m)$

Venne TENUTA TRACCIA di quanto calcolato fino ad un certo punto di esecuzione dell'algoritmo.

### PROGRAMMAZIONE DINAMICA:

Sia  $\text{maxHere}[i]$  il valore del sottovettore massimale che termina in posizione  $A[i]$ .

$$\text{maxHere}[i] = \begin{cases} 0 & \text{se } i < 0 \\ \max(\text{maxHere}[i-1] + A[i], 0) & \text{se } i \geq 0 \end{cases}$$

```

def maxsum4(A):
    maxSoFar = 0      # Max found so far
    maxHere = 0        # Maximum slice ending at the current position
    start = end = 0    # Start, end of the maximum slice found so far
    best = 0            # Beginning of the maximal slice ending here

    for i in range(0, len(A)):
        maxHere += A[i]

        if maxHere <= 0:
            maxHere = 0
            best = i+1

        if maxHere > maxSoFar:
            maxSoFar = maxHere
            start, end = best, i

    return (start, end)

```

A	1	3	4	-8	2	3	-1	3	4	-3	10	-3	2
maxHere	1	4	8	0	2	5	4	7	11	8	18	15	17
maxSoFar	1	4	8	8	8	8	8	8	11	11	18	18	18
best	0	0	0	4	4	4	4	4	4	4	4	4	4
start	0	0	0	0	0	0	0	0	4	4	4	4	4
end	0	1	2	2	2	2	2	2	8	8	10	10	10

Le colonne associate ad ogni elemento del vettore contiene le valori delle variabili dopo l'esecuzione del ciclo per quell'elemento.

## • COME DESCRIVERE E STUDIARE UN ALGORITMO:

Bisogna cercare un linguaggio che non sia impreciso e ambiguo come il linguaggio naturale.

Gli algoritmi possono essere rappresentati in maniera INDEPENDENTE dal linguaggio tramite lo PSEUDOCODICE. Questa rappresentazione deve essere formale.

Lo pseudocodice consente anche di effettuare un'analisi degli algoritmi INDEPENDENTE DALL'ARCHITETTURA DEL CALCOLATORE, anche se, in realtà, esse può avere un grande impatto sulle ~~architetture~~ possibili e sull'efficienza.

## • ESEMPI DI PSEUDOCODICE

Calcolo del massimo in un vettore di  $m$  interi:

orreyMax ( $A, m$ ):

    currentMax  $\leftarrow A[1]$

    for  $i \leftarrow 1$  to  $m$ :

        if  $A[i] > currentMax$  then

            currentMax  $\leftarrow A[i]$

    return currentMax

\* Più vicino al linguaggio di implementazione

orreyMax ( $A, m$ ):

    currentMax  $\leftarrow A[0]$

    for  $i \leftarrow 0$  to  $m-1$ :

        if  $A[i] > currentMax$  then

            currentMax  $\leftarrow A[i]$

    return currentMax

## • QUALITÀ DEGLI ALGORITMI

### - EFFICIENZA:

- TEMPO di esecuzione

- SPAZIO di memoria occupato

I due aspetti dipendono spesso l'uno dall'altro.

Uno stesso problema può essere risolto mediante più algoritmi, che possono avere una efficienza molto differente, non solo legate ad hardware o software usati.

## • COME MISURARE L'EFFICIENZA

• COMPLESSITÀ DI UN ALGORITMO: analisi delle risorse impiegate in funzione delle DIMENSIONI e TIPOLOGIA dell'input.

### - RISORSE:

- TEMPO: tempo impiegato per completare l'algoritmo;

- SPAZIO: quantità di memoria utilizzata;

- BANDA: quantità di bit spediti (algoritmi distribuiti).

## COME MISURARE L'EFFICIENZA TEMPORALE:

Ce sono 2 strategie da prendere in considerazione;

### (1) MISURA DIPENDENTE DA HARDWARE \ SOFTWARE:

tipicamente utile in caso di ottimizzazione a basso livello;

utilizza il concetto di wall-clock time: il tempo EFFETTIVAMENTE impiegato.

### (2) MISURA INDIPENDENTE:

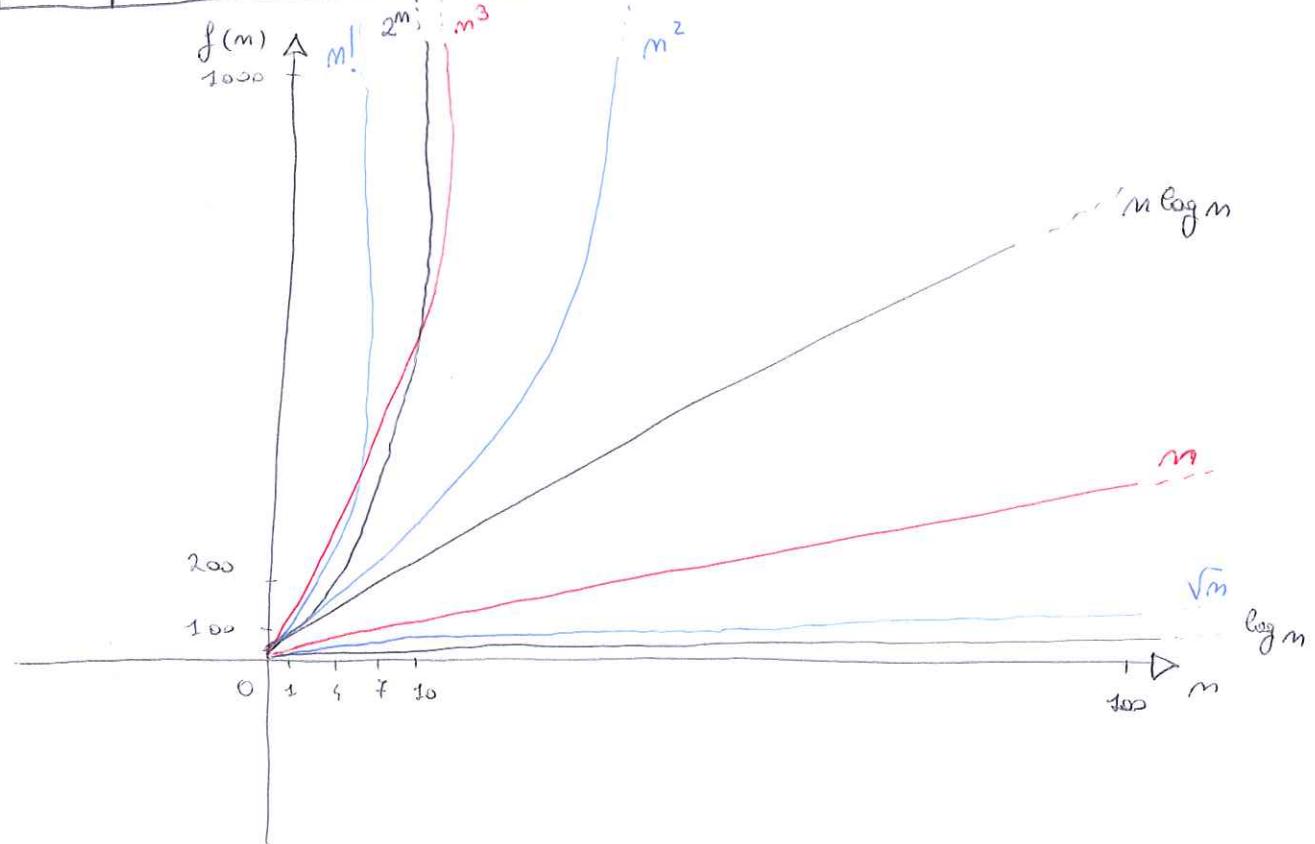
Si concentra di più sull'algoritmo in sé;

Più generale, può catturare le caratteristiche dell'algoritmo su più input.

## CONFRONTO DI TEMPI D'ESECUZIONE

Per  $m \rightarrow +\infty$ , in ordine decrescente di efficienza si ha:

$\log m$	$\sqrt{m}$	$m$	$m \log m$	$m^2$	$m^3$	$2^m$	$m!$
----------	------------	-----	------------	-------	-------	-------	------



## 2. ANALISI DELLA COMPLESSITÀ E DELLE TECNICHE ALGORITMICHE

### • MODELLI DI CALCOLO:

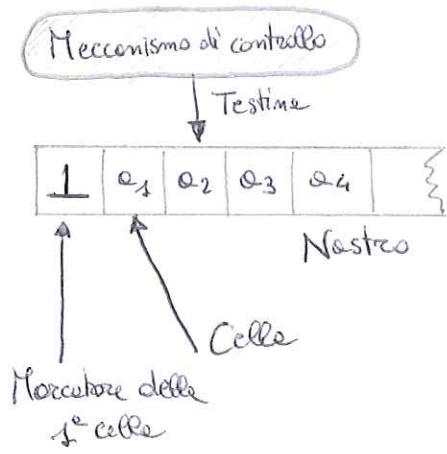
Permettono di rappresentare in maniera estrema un calcolatore, ma devono avere alcune proprietà:

(1) ASTRAZIONE: il modello deve permettere di trascurare i dettagli

(2) REALISMO: il modello deve riflettere la situazione reale

(3) POTENZA MATEMATICA: il modello deve permettere di trovare conclusioni FORMALI sul costo di un algoritmo.

### • ESEMPIO: LA MACCHINA DI TURING



- MACCHINA IDEALE, manipola dati contenuti su un nastro di LUNGHEZZA INFINITA, secondo delle regole prefissate;

Ad ogni passo d'esecuzione, la macchina:

- (1) Legge il simbolo sotto la testina
- (2) Modifica il proprio stato interno
- (3) Scrive un nuovo simbolo nella cella
- (4) Muove la testina a destra o sinistra

- È fondamentale per lo studio della Calcolabilità;

- ! Offre un livello di ASTRAZIONE NON ADATTO per i nostri scopi.

## • RANDOM Access MACHINE (RAM):

- È un'ASTRAZIONE di un elaboratore sequenziale reale, cioè esegue le istruzioni di un algoritmo in sequenze;
- È caratterizzata da una MEMORIA ad accesso casuale (tempo di accesso costante) di Dimensione Infinita ed organizzata in celle. Ciascuna cella può contenere un intero.
- Ha un SINGOLO PROCESSORE
- Operazioni primitive o COSTO 1:
  - Assegnazione
  - Operazione aritmetica
  - Confronto
  - Lettura / Scrittura
- COSTO di operazioni complesse:
  - CICLO: somme tra il costo del test delle condizioni per entrare nel ciclo ed il costo del corpo del ciclo tante volte quanta viene eseguito.
  - if / then / else: costo delle condizioni + costo del blocco (a seconda del caso).
  - CHIAMATA A FUNZIONE: somma del costo di tutte le istruzioni della funzione.

### ESEMPIO:

$$A = [1, 8, 6, 3, 4]$$

ARRAY MAX(A, m):

```
currentMax ← A[0]
for i ← 0 to m:
    if A[i] > currentMax then
        currentMax ← A[i]
return currentMax
```

COSTI

1	2 · 5 + 2
1	1 · 5
1	1
1	1

→ COSTO TOTALE = 20

- Il costo di un'operazione è valutato a meno di un fattore costante (possibilmente arbitrariamente grande) perché:

- 1) Il numero di operazioni elementari per ciascuna istruzione è finito
- 2) Ogni variabile occupa una quantità finita di memoria
- 3) I tempi d'accesso a due variabili diverse sono comunque legati da una costante.

- VANTAGGI: prescinde dall'architettura HARDWARE / SOFTWARE e del linguaggio di programmazione, quindi è più universale.

- Svantaggi: ci dà un'indicazione QUALITATIVA, NON ESATTA.

## • ANALISI CON IL Modello di Costo RAM :

In generale, i risultati dipendono dal particolare INPUT, principalmente per 2 aspetti:  
DIMENSIONE dell'input e TIPOLOGIA dell'input.

E' auspicabile una DIPENDENZA SOLO dalla DIMENSIONE, e quindi un'indipendenza dal particolare tipo di input considerato.

- **ANALISI DEL CASO PESSIMO:** fissate la dimensione dell'input, si considera la sua configurazione più sfavorevole per l'algoritmo, cioè quelle per cui ha costo maggiore e lo si calcola.

- **ANALISI DEL CASO MEDIO:** fissate la dimensione dell'input, si considera il costo medio rispetto ad una distribuzione delle configurazioni dell'input.

E' NECESSARIO conoscere tale distribuzione.

## • DIFFERENZE TRA TEORIA E PRATICA;

TEORIA	PRATICA
Tipi di numeri: $\mathbb{N}, \mathbb{R}$	int, float, double: problemi di precisione
Quello che conta è l'analisi ASINTOTICA	Quello che conta sono i secondi
Descrizione ASTRATTA degli algoritmi	Scegli implementative non banali
la memoria non ha limiti (modello RAM) ed ha costo costante	Gran parte di dati ed effetti dovuti alle gerarchie di memoria (cache, RAM, disco,...)
la memoria è affidabile	la memoria è soggetta a errori
le operazioni elementari hanno costo costante	Le CPU possono richiedere tempi differenti per istruzioni differenti (pipeline, elaboratori, cache,...)
Un singolo processore	Architetture multicore

## • DIMENSIONE DELL'INPUT:

Per ciascun problema occorre FISSARE la DIMENSIONE dell'input: è in base ad esse che si calcola il costo degli algoritmi. La scelta deve essere ragionevole:

- Criterio di COSTO LOGARITMICO: la taglia dell'input è il numero di bit necessari per rappresentarlo;

- Criterio di COSTO UNIFORME: la taglia dell'input è il numero di elementi di cui è costituito (array).

In molti casi si può assumere che gli elementi siano rappresentati da un numero costante di bit:

- le 2 misure coincidono e sono di una costante multiplicative;

- Nei casi dubbi, è ragionevole scegliere il Criterio di Costo logaritmico.

## • ESEMPIO di Analisi del Caso PESSIMO

ARRAYMAX (A, m):		<u>COSTO</u>
currentMax $\leftarrow A[0]$		1
for $i \leftarrow 0$ to $m$ :		$2 \cdot m + 2$
if $A[i] > currentMax$ then		$1 \cdot m$
currentMax $\leftarrow A[i]$		$m - 1$
return currentMax		1

$$\Rightarrow TOT = 1 + 2m + 2 + m + m - 1 + 1 = \boxed{4m + 3}$$

- Per algoritmi più complessi, se si considerassero tutte le costanti, l'analisi può diventare estremamente complessa e complicate.

## • LA RICORSIONE:

Una funzione è detta ricorsiva se chiama se stessa; se due funzioni si chiamano l'un l'altro sono dette mutuamente ricorsive.

Una funzione ricorsiva se individua direttamente la soluzione di alcuni casi particolari, i PASSI BASE, e in questi casi restituisce dei valori.

Se non individua uno dei passi base, allora chiama se stessa (PASSO RICORSIVO) su un sottoproblema o su dati semplificati.

\* Le soluzioni ricorsive sono eleganti, ma Poco EFFICIENTI; in genere sono intercambiabili con una soluzione iterativa.

La RICORSIONE trae origine dal METODO INDUTTIVO, il quale, partendo dai casi particolari, cerca di stabilire una legge universale.

PARADOSSO DEI CORVI di Carl Gustav Hempel, 1940:

1. Ho visto un corvo ed era nero;
2. Ho visto un secondo corvo ed era nero;
3. Ho visto un terzo corvo ed era nero;
4. ---

- Conclusione 1: Il prossimo corvo che vedrai sarà probabilmente nero
- Conclusione 2: Tutti i corvi sono probabilmente neri.

## • PRINCIPIO DI INDUZIONE:

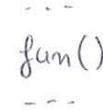
Se  $P$  è una proprietà che vale per il numero 0, e se  $P(m)$  è vera  $\Rightarrow P(m+1)$  è vera, allora  $P(n)$  vale per ogni  $n$ .

- Definizione inductive dell'insieme  $\mathbb{N}$ :

- 1)  $0 \in \mathbb{N}$ ;
- 2)  $m \in \mathbb{N} \Rightarrow (m+1) \in \mathbb{N}$
- 3) nulla' altro è in  $\mathbb{N}$ .

## • RICORSIONE DIRETTA E INDIRETTA

RICORSIONE DIRETTA: se una funzione chiama sé stessa direttamente; def fun():



RICORSIONE INDIRETTA: Se intercorrono più chiamate a funzione prima della chiamata ricorsiva.

def fun1():

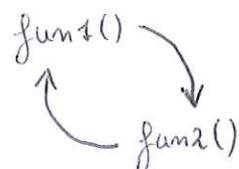
    fun2()

def fun2():

    fun1()

fun1()

→



## • RICORSIONE TERMINALE \ NON TERMINALE

TAIL RECURSION: prima si effettuano le operazioni delle ~~chiamate~~ ricorsive e solo alla fine si effettua la chiamata.

Beneficio della TAIL CALL OPTIMIZATION (non in Python)

```
def recSum(x):  
    if x == 1:  
        return x  
    else:  
        return x + recSum(x-1)
```

```
recSum(4)  
4 + recSum(3)  
4 + (3 + recSum(2))  
4 + (3 + (2 + recSum(1)))  
4 + (3 + (2 + 1))  
10
```

```
def tailRecSum(x, runningTotal=0):  
    if x == 0:  
        return runningTotal  
    else:  
        return tailRecSum(x-1, runningTotal+x)
```

```
tailRecSum(4, 0)  
tailRecSum(3, 4)  
tailRecSum(2, 7)  
tailRecSum(1, 9)  
tailRecSum(0, 10)  
.10
```

## • RECORD DI STACK:

Le chiamate di funzione generano un record di attivazione sullo STACK di memoria; quindi, le funzioni RICORSIVE, che chiamano sé stesse più volte, occupano molta memoria.

SOLUZIONE: quasi sempre, un algoritmo RICORSIVO può essere sostituito da un algoritmo di tipo ITERATIVO, molto più efficiente sia per tempo che per quantità di memoria occupata.

## LA FUNZIONE DI ACKERMAN

E' una funzione RICORSIVA, definita proprio ricorsivamente:

$$A(m, n) = \begin{cases} m+1 & \text{se } m=0 \\ A(m-1, 1) & \text{se } m>0 \text{ ed } n=0 \\ A(m-1, A(m, n-1)) & \text{se } m>0 \text{ ed } n>0 \end{cases}$$

E' TOTALMENTE RICORSIVA ed è NON PRIMITIVA, cioè non puo' essere implementata da un programma che utilizzi soli cicli. Cresce velocissimamente.

Si puo' aumentare il LIMITE DELLO STACK in Python con `sys.setrecursionlimit(100000)`: con questa configurazione,  $\text{ackermann}(4, 1) = 65533$ , ma richiede alcuni minuti.  $\text{ackermann}(4, 2)$  vaeshe comunque.

## NOTAZIONE A FRECCE DI KNOTH

E' stata introdotta nel 1976 da DONALD KNUTH per rappresentare interi molto grandi; è definita RICORSIVAMENTE:

$$a \uparrow^m b = \begin{cases} a^b & \text{se } m=1 \\ 1 & \text{se } m \geq 1 \text{ e } b=0 \\ a \uparrow^{m-1} (a \uparrow^m (b-1)) & \text{altrimenti} \end{cases}$$

Una singola freccia rappresenta il calcolo di un esponentiale:

$$2 \uparrow 4 = 2 \cdot (2 \cdot (2 \cdot 2)) = 2^4 = 16$$

Le frecce aggiuntive indicano l'iterazione dell'operazione associata con meno frecce:

$$2 \uparrow\uparrow 4 = 2 \uparrow (2 \uparrow (2 \uparrow 2)) = 2^{2^2} = 2^{16} = 64 \cdot 1024 = 65536$$

## TORRE DI HANOI:

Algoritmo: (1) Sposte una torre di  $m-1$  dischi dalla sorgente al punto d'appoggio;

(2) Sposte 1 disco dalla sorgente alla destinazione;

(3) Sposte una torre di  $m-1$  dischi dal punto d'appoggio alla destinazione.

def moveTower(m, source, dest, temp):

if  $m == 1$ :  
print("Move disk from", source, "to", dest + ".")

else:  
moveTower(m-1, source, temp, dest)

moveTower(1, source, dest, temp)

moveTower(m-1, temp, dest, source)

def hanoi(m):

moveTower(m, "A", "C", "B")

E' un problema complesso da risolvere:

DISCHI	PASSI
1	1
2	3
3	7
4	15
5	31

Per risolvere un problema con  $m$  dischi, sono necessari  $2^m - 1$  PASSI.

## La Successione di Fibonacci:

Problema della Dinamica delle Popolazioni  $\rightarrow$  Espansione dei Conigli

- Modello sui conigli:
- 1 coppia di conigli genera 1 coppia di coniglietti all'anno;
  - i conigli cominciano a riprodursi a partire dal 2° anno di vita;
  - i conigli sono immortali

Sia  $F_m$  il numero di coppie presenti all'anno  $m$ :

$$F_1 = 1$$

$$F_2 = 1 \quad (\text{i conigli iniziano a riprodursi dal 2° anno})$$

$$F_3 = 2$$

$$F_4 = 3$$

$$F_5 = 5$$

$$F_6 = 8$$

In generale  $F_m = \begin{cases} F_{m-1} + F_{m-2}, & \text{se } m \geq 3 \\ 1 & \text{se } m=1,2 \end{cases}$   $\rightarrow$  RELAZIONE DI RICORRENZA

Questa RELAZIONE DI RICORRENZA individua un PROBLEMA COMPUTAZIONALE; esiste un algoritmo che lo risolve?

FORMULA DI BINET:  $\lim_{m \rightarrow \infty} \frac{F_m}{F_{m-1}} = \varphi$ , dove  $\varphi = \frac{1+\sqrt{5}}{2}$  (SEZIONE AURCA)

$$\text{Per } \varphi \text{ vengono le seguenti proprietà: } \varphi - 1 = \frac{-1 + \sqrt{5}}{2} = \frac{1}{\varphi}$$

$$1 - \varphi = \frac{1 - \sqrt{5}}{2} = -\frac{1}{\varphi}$$

$$\Rightarrow F_m = \frac{\varphi^m}{\sqrt{5}} - \frac{(1-\varphi)^m}{\sqrt{5}} = \underbrace{\frac{\varphi^m - (-\varphi)^{-m}}{\sqrt{5}}}_{\text{In forma chiusa}}$$

PROBLEMA: La soluzione è molto efficace (costo COSTANTE), MA è INESATTA:

SI COMMETTONO ERRORI DI APPROXIMAZIONE !

## "DIVIDE ET IMPERA"

"Divide et Impera" ~ Filippo il Mecenate (posto di Alessandro Magni)

- **DIVIDE**: il problema viene suddiviso in sottoproblemi di dimensioni minori;
- **IMPERA**: i sottoproblemi vengono risolti ricorsivamente o direttamente se di dimensione sufficientemente piccole;
- **COMBINA**: combinando le soluzioni dei sottoproblemi, si ottiene quella del problema originale.

### Ricerca Binaria:

Algoritmo per individuare l'indice di un certo valore (dato in input) in un insieme di elementi ORDINATO.

#### • Implementazione Ricorsiva:

binarySearch ( $A[0, \dots, N-1]$ , low, high, value);

    if high < low then  
        return  $\perp$

    mid =  $(\text{low} + \text{high}) / 2$

    if  $A[\text{mid}] > \text{value}$  then

        return (binarySearch ( $A$ , low, mid-1, value))

    else if  $A[\text{mid}] < \text{value}$  then

        return (binarySearch ( $A$ , mid+1, high, value))

    else

        return mid

COSTI

|  
 $c_1$

|  
 $c_2$

|  
 $c_3$

|  
 $c_4$

|  
 $c_5 + T(L^{(m-1)/2})$

|  
 $c_6$

|  
 $c_5 + T(L^{m/2})$

|  
 $c_7$

Caso Pessimo: non viene restituito niente e si prende sempre la sottoparte di destra di dimensioni  $m/2$ . Per semplicità, sia  $m = 2^k$ .

Due casi:

-  $\text{high} < \text{low}$  ( $m=0$ ):  $T(m) = c_1 + c_2 = c$

-  $\text{low} < \text{high}$  ( $m>0$ ):  $T(m) = T(m/2) + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 = T(m/2) + d$

$$\Rightarrow T(m) = \begin{cases} c & \text{se } m=0 \\ T\left(\frac{m}{2}\right) + d & \text{se } m>0 \end{cases}$$

Espandendo la relazione di ricorrenza:  $T(m) = T(m/2) + d = T(m/4) + 2d = T(m/8) + 3d = \dots = T(1) + kd = T(0) + (k+1)d = c + kd + d$

$$\text{Poiché } m = 2^k \Rightarrow k = \log_2 m \Rightarrow T(m) = d \log m + c + d = \boxed{d \log m + c}$$

## FIBONACCI : implementazione ricorsiva

FIBONACCI 2 (n) :

```
if m ≤ 2 then
    return 1
```

```
else
    return (FIBONACCI (m-1) + FIBONACCI (m-2))
```

COSTI

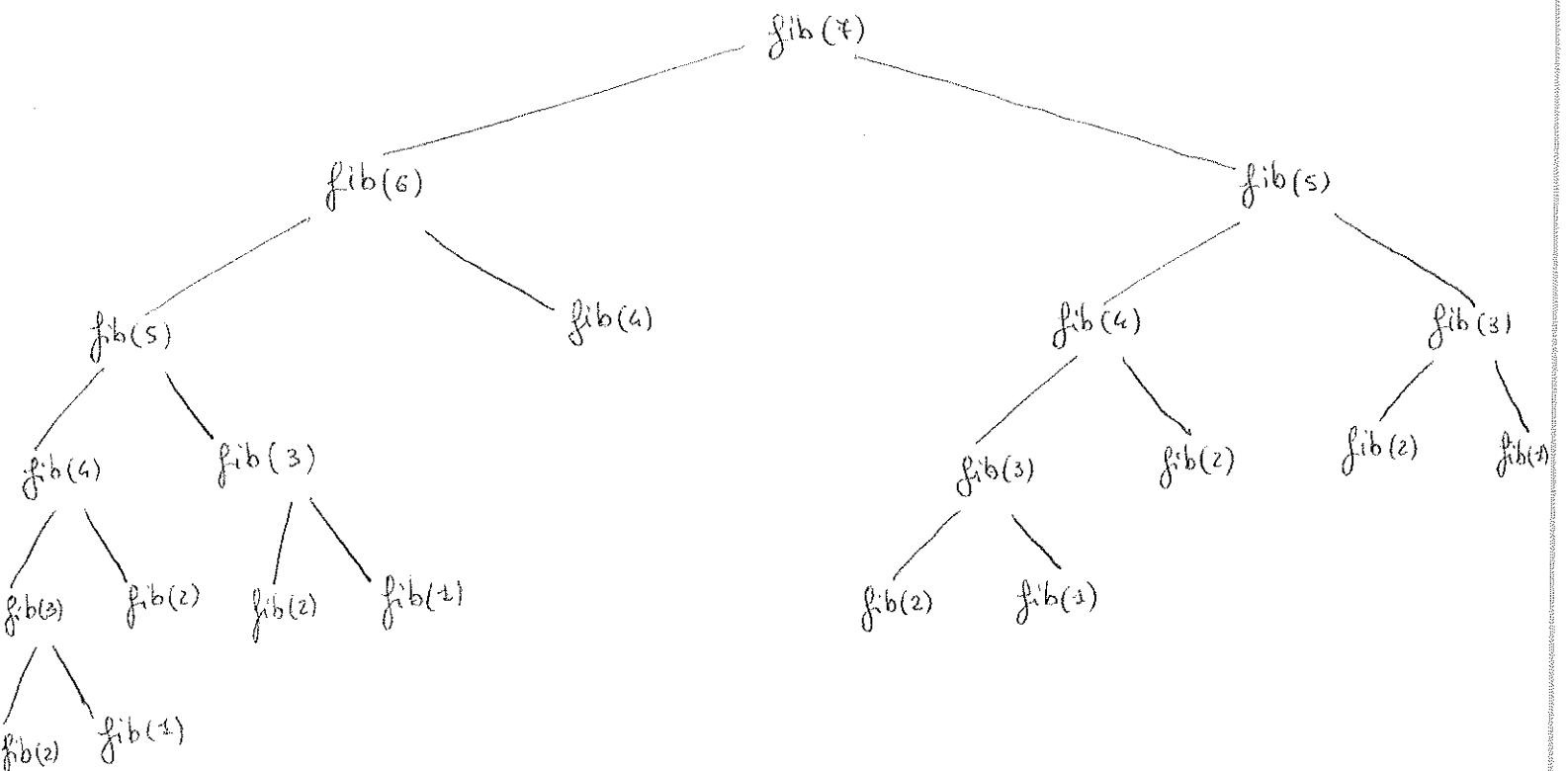
1

1

$F(m-1) + F(m-2)$

- Se  $m \geq 3$ , il costo totale aumenta di 2 per ciascuna invocazione.

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



\* LO STESSO VALORE VIENE CALCOLATO PIÙ VOLTE!

$$T(m) \approx F(m) \approx \varphi^m$$

Vedremo in seguito come fare di meglio ...

## • Implementazione Iterativa:

binary Search ( $A[0 \dots N-1]$ , value):

low = 0

high =  $N-1$

while low  $\leq$  high do

    mid = (low + high) / 2

    if  $A[mid] > value$  then

        high = mid - 1

    else if  $A[mid] < value$  then

        low = mid + 1

    else

        return mid

return -1

## RELAZIONI DI RICORRENZA

### • Definizione:

Una RELAZIONE DI RICORRENZA per una sequenza  $\{e_m\}$  è un' EQUAZIONE MATEMATICA che esprime  $e_m$  negli elementi precedenti della successione  $e_0, e_1, \dots, e_{m-1}$  per tutti gli interi  $m \geq m_0$ , dove  $m_0$  è un intero non negativo.

- Una SEQUENZA è detta SOLUZIONE di un'EQUAZIONE DI RICORRENZA se i suoi termini soddisfano l'equazione.

### RELAZIONI DI RICORRENZA LINEARI:

Una relazione di ricorrenza LINEARE di GRADO K è dello stesso forma:

$$Q_m = c_1 Q_{m-1} + c_2 Q_{m-2} + \dots + c_k Q_{m-k} + F(m)$$

dove  $c_i$  è una costante  $\forall i=1, \dots, k$  e  $c_k \neq 0$ .

Le relazioni LINEARI si dividono in 2 SOTTOCLASSI:

- OMOGENEE:  $F(m) = 0$
- Non OMOGENEE:  $F(m) \neq 0$

### • ESEMPI:

#### Lineari O'mogenee

$$e_m = 1 \cdot e_{m-1} \quad (\text{grado } 1)$$

$$f_m = f_{m-1} + f_{m-2} \quad (\text{grado } 2)$$

$$g_m = 3g_{m-3} \quad (\text{grado } 3)$$

#### Non Lineari O'mogenee

$$h_m = h_{m-1}^2 + 2^m$$

$$k_m = m k_{m-1} - 2k_{m-2}$$

#### Lineari Non O'mogenee

$$l_m = l_{m-1} + 2^m$$

$$m_m = 2m_{m-1} + 1 \quad (\text{Hanoi})$$

$$n_m = 3n_{m-1} + m$$

#### Non Lineari Non O'mogenee

$$p_m = p_{m-1}^2 + 2^m$$

$$q_m = m^2 q_{m-1} + m$$

## COME RISOLVERE RELAZIONI LINEARI OMOGENEE:

$\{a_m\}$  è soluzione delle RELAZIONI LINEARI  $a_m = c_1 a_{m-1} + c_2 a_{m-2} \iff a_m = d_1 r_1^m + d_2 r_2^m$

dove:

-  $r_1$  ed  $r_2$  sono 2 radici DIVERSE dell'equazione:  $r^2 - c_1 r - c_2 = 0$

$$- a_0 = d_1 + d_2$$

$$- a_1 = d_1 r_1 + d_2 r_2$$

$$- d_1 = \frac{a_1 - a_0 r_2}{r_1 - r_2} \quad e \quad d_2 = \frac{a_0 r_1 - a_1}{r_1 - r_2}$$

Per esempio:  $a_m = a_{m-1} + 2a_{m-2}$ , con  $a_0 = 2$  e  $a_1 = 7$

$$\rightarrow c_1 = 1, c_2 = 2 \Rightarrow r^2 - r - 2 = 0 \Rightarrow r_{1,2} = \frac{1 \pm \sqrt{1+8}}{2} = \begin{cases} -1 \\ 2 \end{cases}$$

$$\Rightarrow \cancel{r_1 = -1}, \cancel{r_2 = 2}, r_1 = 2, r_2 = -1$$

$$\left. \begin{aligned} \Rightarrow d_1 &= \frac{7 - 2 \cdot (-1)}{1+2} = \frac{9}{3} = 3 \\ d_2 &= \frac{2 \cdot 2 - 7}{2-(-1)} = \frac{-3}{3} = -1 \end{aligned} \right\} \Rightarrow a_m = 3 \cdot 2^m - 1 \cdot (-1)^m$$

## COME RISOLVERE RELAZIONI LINEARI OMOGENEE (2):

$\{a_m\}$  è soluzione di  $a_m = c_1 a_{m-1} + c_2 a_{m-2} \iff a_m = d_1 r_0^m + d_2 r_0^m \cdot m$

dove:

-  $r_0$  è l'UNICA SOLUZIONE di:  $r^2 - c_1 r - c_2 = 0$

$$- a_0 = d_1$$

$$- a_1 = d_1 r_0 + d_2 r_0 = r_0 (d_1 + d_2)$$

$$- d_1 = a_0 \quad e \quad d_2 = \frac{a_1 - r_0 d_1}{r_0} = \frac{a_1 - a_0}{r_0} = -1$$

## COME RISOLVERE RELAZIONI LINEARI OMOGENEE (3):

CASO GENERALE:  $a_m = c_1 a_{m-1} + c_2 a_{m-2} + \dots + c_k a_{m-k} \iff a_m = d_1 r_1^m + d_2 r_2^m + \dots + d_k r_k^m$

dove:

-  $r_1, r_2, \dots, r_k$  sono RADICI DISTINTE di  $r^k - c_1 r^{k-1} - \dots - c_k = 0$

$$- a_0 = d_1 + d_2 + \dots + d_k$$

$$- a_1 = d_1 r_1 + d_2 r_2 + \dots + d_k r_k$$

$$- a_2 = d_1 r_1^2 + d_2 r_2^2 + \dots + d_k r_k^2$$

... ...

## COME RISOLVERE RELAZIONI LINEARI NON OMOGENEE

Se la sequenza  $\{\alpha_m^{(P)}\}$  è una soluzione particolare della relazione di ricorrenza

$\alpha_m = c_1 \alpha_{m-1} + \dots + c_k \alpha_{m-k} + F(m) \Rightarrow$  Tutte le Soluzioni sono delle forme

$$\left\{ \alpha_m^{(P)} + \alpha_m^{(h)} \right\},$$

dove  $\{\alpha_m^{(h)}\}$  è SOLUZIONE della RELAZIONE OMOGENEA ASSOCIASTA:

$$\alpha_m = c_1 \alpha_{m-1} + \dots + c_k \alpha_{m-k}$$

### ESEMPIO:

Quel è la soluzione di  $\alpha_m = 3\alpha_{m-1} + 2m$ , con  $\alpha_1 = 3$ ?

$$\text{Abbiamo } c_1 = 3 \Rightarrow r - 3 = 0 \Rightarrow r = 3$$

- $\alpha_m^{(h)} = \alpha 3^m$

- Sia  $p_m = cm + d \Rightarrow$  de  $\alpha_m = 3\alpha_{m-1} + 2m \Rightarrow cm + d = 3(c(m-1) + d) + 2m$

$$\Rightarrow cm + d = 3cm - 3c + 3d + 2m \Rightarrow 2cm + 2m + 2d - 3c = 0$$

$$\Rightarrow 2m(c+1) + 2d - 3c = 0 \Rightarrow c = -1, d = -\frac{3}{2}, \alpha_m^{(P)} = -m - \frac{3}{2}$$

- $\boxed{\alpha_m = \alpha_m^{(P)} + \alpha_m^{(h)} = \alpha 3^m - m - \frac{3}{2}}$

Da cui:  $\alpha_1 = 3\alpha - 1 - \frac{3}{2} = 3 \Rightarrow 3\alpha = \frac{11}{2} \Rightarrow \alpha = \frac{11}{6}$

$\Rightarrow$  La soluzione è:  $\boxed{\alpha_m = \frac{11}{6} \cdot 3^m - m - \frac{3}{2}}$

### ESEMPIO: SOMMA DEI PRIMI M NUMERI

Quel è la soluzione di  $\alpha_m = \alpha_{m-1} + m$  con  $\alpha_1 = 1$ ?

- $c_1 = 1 \Rightarrow r - 1 = 0 \Rightarrow r = 1$

- $\alpha_m^{(h)} = \alpha 1^m = \alpha$

- Sia  $p_m = m(cm + d) = cm^2 + dm \Rightarrow$  de  $\alpha_m = \alpha_{m-1} + m \Rightarrow$

$$\Rightarrow cm^2 + dm = c(m-1)^2 + d(m-1) + m \Rightarrow cm^2 + dm = cm^2 - 2cm + c + dm - d + m$$

$$\Rightarrow m(1-2c) + c - d = 0 \Rightarrow c = \frac{1}{2}, d = \frac{1}{2}, \alpha_m^{(P)} = \frac{1}{2}(m^2 + m)$$

- $\alpha_m = \alpha_m^{(P)} + \alpha_m^{(h)} = \alpha + \frac{m(m+1)}{2};$  Da cui, poiché  $\alpha_1 = 1: 1 = \alpha + \frac{1 \cdot 2}{2} \Rightarrow \alpha = 0$

$\Rightarrow$  La SOLUZIONE è:  $\boxed{\alpha_m = \frac{m(m+1)}{2}}$

OK!!

## ANALISI ASINTOTICA

Motivazioni:

- E' interessante studiare gli algoritmi al varire delle dimensione dell'input, o meno di costanti:  
già il modello RAM "nascondeva" alcune costanti.  
occorre comunque tenere a mente che tali costanti possono essere arbitrariamente piccole ed avere un impatto sull'efficienza su determinate strutture hardware/software.
- L'ANALISI ASINTOTICA TRASCURA LE COSTANTI;
- Viene influenzata dal concetto di **OPERAZIONE DOMINANTE**, cioè quella che intuitivamente viene eseguita più volte,

### DEFINIZIONI FONDAMENTALI:

Siano  $f(m): \mathbb{N} \rightarrow \mathbb{R}$  e  $g(m): \mathbb{N} \rightarrow \mathbb{R}$  due funzioni tali che  $f, g \geq 0$  e non decrescenti:

O grande:  $f(m) = O(g(m)) \Leftrightarrow \exists c \in \mathbb{R}, c > 0$  e  $m_0 \in \mathbb{N}, m_0 > 1$  t. che  $f(m) \leq c \cdot g(m)$  per  $m > m_0$

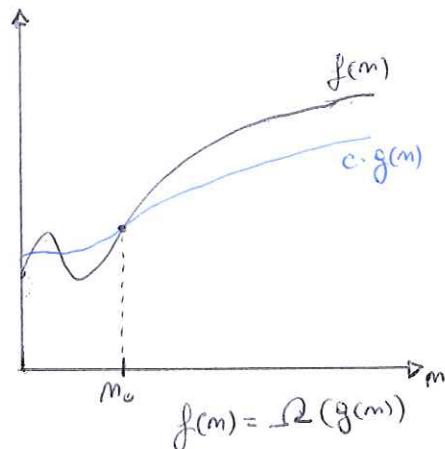
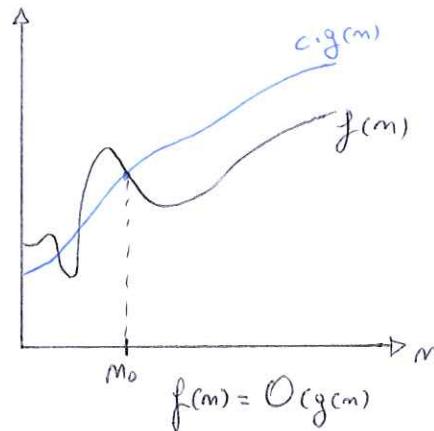
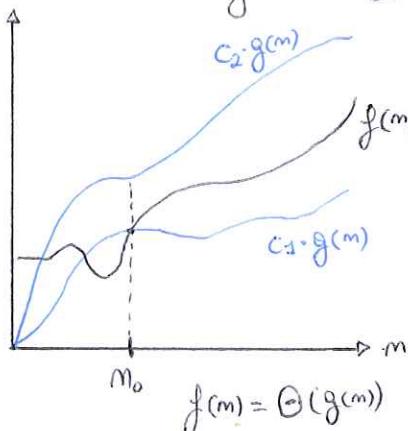
$f$  cresce AL PIÙ come  $g$

O mega grande:  $f(m) = \Omega(g(m)) \Leftrightarrow \exists c > 0, \exists m_0 \geq 1$  t. che:  $f(m) \geq c \cdot g(m)$  per  $m > m_0$

$f$  cresce ALMENO come  $g$

Teta grande:  $f(m) = \Theta(g(m)) \Leftrightarrow f(m) = O(g(m))$  ed  $f(m) = \Omega(g(m))$

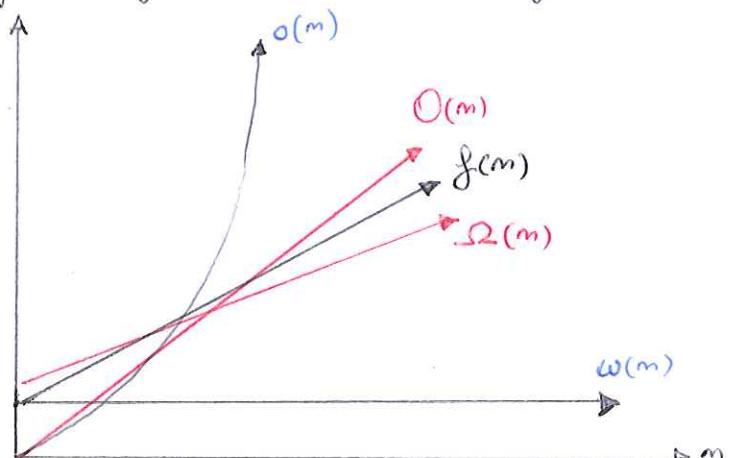
$f$  cresce COME  $g$



o piccolo:  $f(m) = O(g(m))$  se  $\lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = 0$   
è un limite superiore NON STRETTO

w piccolo:  $f(m) = w(g(m))$  se  $g(m) = O(f(m))$   
cioè  $\lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = \infty$

è un limite inferiore NON STRETTO



## CONFRONTO TRA NOTAZIONI GRANDI E PICCOLE:

- $f(m) = O(g(m)) \rightarrow$  i limiti  $0 \leq f(m) \leq c \cdot g(m)$  sono verificati per QUALCHE costante  $c > 0$ .
- $f(m) = o(g(m)) \rightarrow$  i limiti  $0 \leq f(m) \leq c \cdot g(m)$  sono verificati per TUTTE le costanti  $c > 0$ .

ANALOGIE:  $f(m) = O(g(m))$  ricorda  $a \leq b$   
 $f(m) = \Omega(g(m)) \rightarrow a \geq b$   
 $f(m) = \Theta(g(m)) \rightarrow a = b$   
 $f(m) = o(g(m)) \rightarrow a < b$   
 $f(m) = w(g(m)) \rightarrow a > b$

\*  $O(1)$  vuol dire che l'algoritmo non dipende dalla dimensione dell'input, MA  
esegue un NUERO DI ISTRUZIONI COSTANTE.

## OPERAZIONE DOMINANTE:

Un'operazione di un algoritmo di costo  $f(m)$  si dice DOMINANTE se, per ogni  $m$ , essa viene eseguita, nel caso pessimo, un numero di volte  $d(m)$  che soddisfa:

$$f(m) < a \cdot d(m) + b$$

per opportuni  $a, b \in \mathbb{R}$ .

Per esempio: l'operazione "if  $A[i] > \text{currentMax}$  then" nell'algoritmo everyMax è DOMINANTE.

## LIMITI DELL'ANALISI ASINTOTICA DEL CASO PESSIMO:

- Le COSTANTI TRASCURATE possono essere molto grandi: un algoritmo con costo  $10^{50}m$  è  $O(m)$ , ma potrebbe essere poco pratico
- Quando  $m$  è molto piccolo, per esempio, un algoritmo quadratico potrebbe essere più efficiente di uno lineare ( $3m$  contro  $m^2$ )
- IL CASO PESSIMO potrebbe essere molto raro  $\rightarrow$  ANALISI DEL CASO MEDIO

## Problema: INSERTION Sort

Ordinare un array in modo non decrescente: ciclo che inserisce  $A[i]$  nella posizione corretta nel vettore ordinato  $A[0, \dots, i-1]$ , per  $i=1, \dots, m$ .

def insertionSort(A, m):  
 for i ← 1 to m:  
 Key ← A[i]  
 j ← i-1  
 while j ≥ 1 and A[j] > key:  
 A[j+1] ← A[j]  
 j ← j-1  
 A[j+1] ← key

## ANALISI INSERTION SORT:

insertionSort ( $A, m$ ):	COSTO	RIPETIZIONI
for $i \leftarrow 1$ to $m$ :	$c_1$	$m$
Key $\leftarrow A[i]$	$c_2$	$m-1$
$J \leftarrow i-1$	$c_3$	$m-1$
while $J \geq 0$ and $A[J] > \text{key}$ :	$c_4$	$\sum_{i=2}^m t_i$
$A[J+1] \leftarrow A[J]$	$c_5$	$\sum_{i=2}^m (t_i - 1)$
$J \leftarrow J-1$	$c_6$	$\sum_{i=2}^m (t_i - 1)$
$A[J+1] \leftarrow \text{key}$	$c_7$	$m-1$

$$\text{Costo Tot} = c_1 m + c_2 (m-1) + c_3 (m-1) + c_4 \sum_{i=2}^m t_i + c_5 \sum_{i=2}^m (t_i - 1) + c_6 \sum_{i=2}^m (t_i - 1) + c_7 (m-1)$$

L'OPERAZIONE DOMINANTE è una qualunque delle 3 eseguite nel ciclo più interno

- \* Il costo non dipende solo dalla dimensione dell'array, ma anche dalla TIPICITÀ, cioè della DISTRIBUZIONE DEI DATI D'INGRESSO.

### • CASO MIGLIORE: vettore già ordinato

In ogni iterazione il 1° elemento delle sottosequenze non ordinate viene confrontato solo con l'ultimo di quelle ordinate  $\rightarrow$  COSTO:  $\Theta(m)$

### • CASO PESSIMO: vettore ordinato al contrario

ogni iterazione dovrà scorrere e spostare ogni elemento delle sottosequenze ordinate prima di poter inserire il 1° elemento di quelle non ordinate  $\rightarrow$  COSTO:  $O(m^2)$

### • CASO MEDIO:

Anche in questo caso il costo è quadratico  $O(m^2)$ , il che lo rende impraticabile per input grandi.

## ANALISI CASO MEDIO:

Assumiamo che tutti gli elementi siano diversi fra loro.

Sia  $X_{ij}$  una variabile aleatoria che vale 1 se  $A[i]$  deve essere scambiato con  $A[j]$

$\Rightarrow$  Abbiamo bisogno di  $\frac{m(m-1)}{2}$  variabili di questo tipo per caratterizzare un vettore di dimensione  $m$ .

Sia  $I = \sum X_{ij}$  un'altra VARIABILE ALEATORIA che indica il numero di scambi necessari a calcolare l'output.

$E[I] = E\left[\sum X_{ij}\right] = \sum E[X_{ij}]$  ;  $E[X_{ij}] = \text{Prob. } [X_{ij}=1] = \begin{matrix} \text{Probabilità che } A[i] \\ \text{sia scambiato con } A[j] \end{matrix}$

STATISTICAMENTE,  $A[i]$  e  $A[j]$  sono scambiati la metà delle volte.

$E[X_{ij}] = \frac{1}{2} \Rightarrow E[I] = \sum \frac{1}{2}$  ; Poiché ci sono  $\frac{m(m-1)}{2}$  variabili  $\Rightarrow E[I] = \frac{m(m-1)}{4} =$

$$= \frac{m^2 - m}{4} = \underline{\underline{\Theta(m^2)}}$$

ok!!!

## CLASSI DI COMPLESSITÀ

La seguente tabella riporta il numero di microsecondi impiegati per risolvere programmi al crescere della dimensione  $m$  dell'input, e secondo delle efficienze delle algoritmi:

$f(m)$	$m=10$	$m=10^2$	$m=10^3$	$m=10^4$	CLASSE
$\log m$	3	6	9	13	logaritmica
$\sqrt{m}$	3	10	31	100	sublineare
$m$	10	100	1000	10000	lineare
$m \log m$	30	664	9965	132877	loglineare
$m^2$	$10^2$	$10^4$	$10^6$	$10^8$	quadratiche
$m^3$	$10^3$	$10^6$	$10^9$	$10^{12}$	cubica
$2^m$	1024	$10^{30}$	$10^{300}$	$10^{3000}$	esponentiale
$m!$	3628800	$\approx 10^{157}$	$\approx 10^{2567}$	$\approx 10^{38659}$	fattoriale

## ANALISI DELLE RICORRENZE

$$\text{FIBONACCI : } F(m) = \begin{cases} F_{m-1} + F_{m-2} & \text{se } m \geq 3 \\ 1 & \text{se } m \leq 2 \end{cases}$$

$$\text{HANDI : } T(m) = 2^m - 1$$

$$\text{RICERCA BINARIA : } T(m) = \begin{cases} c & \text{se } m = 0 \\ T(m/2) + d & \text{se } m \geq 1 \end{cases}$$

Vogliamo rappresentare in forme chiuse le classi di complessità delle funzioni:  $\Theta(\log m)$

## TECNICHE PER DERIVARE LE CLASSI DI COMPLESSITÀ

- ANALISI DEI LIVELLI: si "struttura" la relazione di ricorrenze in un albero i cui nodi rappresentano i costi ai vari livelli delle ricorsione;
- ANALISI PER TENTATIVI o PER SOSTITUZIONE: si cerca di "individuare" una soluzione e si dimostra INDUTTIVAMENTE che è giusta
- RICORRENZE COMUNI: vi è una classe di ricorrenze che possono essere risolte facendo riferimento ad alcuni teoremi specifici.

## ANALISI DEI LIVELLI : 1° esempio

$$T(m) = \begin{cases} c & \text{se } m=0 \\ T(m/2) + d & \text{se } m>0 \end{cases}$$

Espandiamo, assumendo  $m=2^k$  ( $k=\log_2 m$ ):

$$\begin{aligned} T(m) &= T(m/2) + d = T(m/4) + d + d = T(m/8) + 3d = \dots = T(1) + d \log m = \\ &= d \log m + c \quad \Rightarrow \boxed{T(m) = \Theta(\log m)} \end{aligned}$$

## ANALISI DEI LIVELLI : 2° esempio

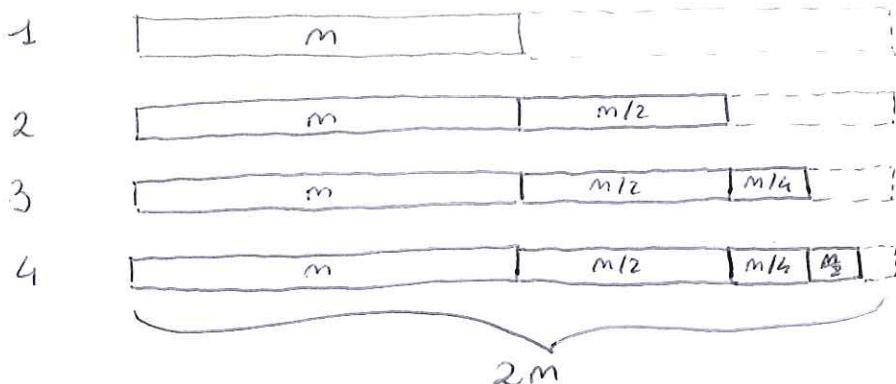
$$T(m) = \begin{cases} 1 & \text{se } m=0 \\ 4T(m/2) + m & \text{se } m>0 \end{cases}$$

Espandiamo:

$$\begin{aligned} T(m) &= 4T(m/2) + m = m + 4\frac{m}{2} + 16T(m/4) = m + 2m + \frac{16m}{4} + 64T(m/8) = \dots = \\ &= m + 2m + 4m + 8m + \dots + 2^{\log_2 m - 1} m + 4^{\log_2 m} T(1) = m \sum_{j=0}^{\log_2 m - 1} 2^j + 4^{\log_2 m} = \\ &= m \cdot \frac{2^{\log_2 m} - 1}{2 - 1} + (2^{\log_2 m})^2 = m \cdot (m-1) + m^2 = 2m^2 - m \quad \Rightarrow \boxed{T(m) = \Theta(m^2)} \end{aligned}$$

## METODO DI SOSTITUZIONE:

$$T(m) = \begin{cases} T(m/2) + m & \text{se } m > 1 \\ 1 & \text{se } m \leq 1 \end{cases}$$



\* Intuisco che il costo potrebbe essere  $2m$ :

$$T(m) = m \cdot \sum_{i=0}^{\log_2 m} \left(\frac{1}{2}\right)^i \leq m \cdot \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = m \cdot \frac{1}{1-\frac{1}{2}} = (2m) \quad \underline{\underline{\text{oh!!!}}}$$

## METODO DI SOSTITUZIONE: Limite Superiore

Ipotizziamo che  $T(m) = \mathcal{O}(m)$ .

Dalla definizione:  $\exists c > 0, \exists m_0 \geq 1 : T(m) \leq cm, \forall m \geq m_0$

Dimostriamo il CASO BASE:  $T(1) = 1 \leq 1 \cdot c \Leftrightarrow \forall c \geq 1$

PASSO INDUTTIVO: ipotesi:  $\forall k \geq m : T(k) \leq ck$

$$T(m) = T(\lfloor m/2 \rfloor) + m \leq c\lfloor m/2 \rfloor + m \leq c(m/2) + m = m\left(\frac{c}{2} + 1\right) \leq cm \Leftrightarrow \\ \Leftrightarrow \frac{c}{2} + 1 \leq c \Leftrightarrow c \geq 2$$

$\Rightarrow$  Scegliamo  $c=2$  e  $m_0=1 \Rightarrow$  Abbiamo dimostrato che  $\boxed{T(m) = \mathcal{O}(m)}$

## METODO DI SOSTITUZIONE: Limite Inferiore

Ipotizziamo che  $T(m) = \Omega(m)$

Dalla definizione:  $\exists c > 0, \exists m_0 \geq 1 : T(m) \geq cm, \forall m \geq m_0$

Dimostriamo il CASO BASE:  $T(1) = 1 \geq 1 \cdot c \Leftrightarrow \forall c \leq 1$

PASSO INDUTTIVO: Ipotesi:  $\forall k \geq m : T(k) \geq ck$

$$T(m) = T(\lfloor m/2 \rfloor) + m \geq c\lfloor m/2 \rfloor + m \geq \frac{cm}{2} - \frac{c}{2} + m = m\left(\frac{c}{2} - \frac{c}{m} + 1\right) \geq cm \Leftrightarrow \\ \Leftrightarrow \frac{c}{2} - \frac{c}{m} + 1 \geq c \Leftrightarrow c \leq 2 - \frac{2}{m}$$

$\Rightarrow T(m) \geq cm$  per  $c \leq 1$  (caso base) e  $T(m) \geq cm$  per  $c \leq 2 - \frac{2}{m}$  (passo induttivo).

Scegliamo  $c=1$  e  $m_0=1 \Rightarrow$  Abbiamo dimostrato che  $\boxed{T(m) = \Omega(m)}$

IN CONCLUSIONE:  $T(m) = \mathcal{O}(m)$  e  $T(m) = \Omega(m) \Rightarrow \boxed{T(m) = \Theta(m)}$

## MASTER THEOREM

Il Master Theorem permette di determinare la classe di complessità di alcune famiglie di RELAZIONI di RICORRENZA del tipo:

$$T(m) = \begin{cases} \alpha \cdot T(m/b) + cm^\beta & \text{se } m > 1 \\ d & \text{se } m \leq 1 \end{cases}$$

dove:  $\alpha > 1$  e  $b > 1$  ( $a, b \in \mathbb{N}$ ) e  $c > 0, \beta \geq 0$  ( $c, \beta \in \mathbb{R}$ ).

Posto  $\alpha = \log_b a$ , il Teorema ci dice:

$$T(m) = \begin{cases} \Theta(m^\alpha) & \text{se } \alpha > \beta \\ \Theta(m^\alpha \log m) & \text{se } \alpha = \beta \\ \Theta(m^\beta) & \text{se } \alpha < \beta \end{cases}$$

INTUIZIONE: è una "gara" fra  $m^\beta$  e  $m^\alpha$ .

• ESEMPIO:

$$T(m) = \begin{cases} \alpha T(m/b) + cm^\beta & m > 1 \\ d & M \leq 1 \end{cases}$$

Sia  $T(m) = gT\left(\frac{m}{3}\right) + m \Rightarrow \alpha = 9, b = 3, c = 1, \beta = 1$

$$\Rightarrow \alpha = \log_b \alpha = \log_3 9 = 2 \Rightarrow \alpha > \beta \Rightarrow T(m) = \Theta(m^\alpha) = \Theta(m^2)$$

COSTO QUADRATICO

• ESEMPIO: Ricette binarie

$$T(m) = T\left(\frac{m}{2}\right) + 1$$

Dunque:  $\alpha = 1, b = 2, c = 1, \beta = 0 \Rightarrow \alpha = \log_2 1 = 0$

$$\Rightarrow \alpha = \beta = 0 \Rightarrow T(m) = \Theta(m^\alpha \log m) = \Theta(\log m) \rightarrow \text{COSTO LOGARITMICO}$$

RELAZIONI LINEARI DI ORDINE COSTANTE:

Date una RELAZIONE di RICORRENZA della forma:  $T(m) = \begin{cases} \sum_{1 \leq i \leq h} \alpha_i T(m-i) + cm^\beta & \text{se } m > m \\ \Theta(1) & \text{se } m \leq m \end{cases}$

dove:  $\alpha_1, \dots, \alpha_h \in \mathbb{N}^+, \text{ con } h > 0$   
 $c > 0, \beta \geq 0, \text{ con } c, \beta \in \mathbb{R}$ .

• Posto,  $\alpha = \sum_{1 \leq i \leq h} \alpha_i$ , abbiamo:

$$T(m) = \begin{cases} \Theta(m^{\beta+1}) & \text{se } \alpha = 1 \\ \Theta(m^\beta) & \text{se } \alpha \geq 2 \end{cases}$$

• ESEMPIO:

$$T(m) = T(m-10) + m^2$$

$$\Rightarrow \alpha_1 = \alpha_2 = \dots = \alpha_9 = 0; \alpha_{10} = 1 \Rightarrow \alpha = \sum_{i=1}^{10} \alpha_i = 1; \beta = 2, c = 1$$

$$\Rightarrow T(m) = \Theta(m^{2+1}) = \Theta(m^3) \rightarrow \text{COSTO CUBICO}$$

• ESEMPIO: Fibonacci

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

$$\Rightarrow \alpha = 2, c = 1, \beta = 0 \Rightarrow T(m) = \Theta(c^m m^\beta) = \Theta(2^m) \rightarrow \text{COSTO ESPONENZIALE}$$

## ESEMPI SUL PROBLEMA DEL SOTTOVETTORE DI SOMMA MASSIMALE:

- SOLUZIONE NAÏF (doppio ciclo for):

def maxSum2(A):

$$\text{maxSoFar} = 0$$

for i in range(0, len(A)):

    for j in range(i, len(A)):

$$\quad \text{maxSoFar} = \max(\text{maxSoFar}, \sum(A[i:j+1])) \leftarrow$$

return maxSoFar

OPERAZIONE DOMINANTE

- Utilizzando il concetto di operazione dominante, possiamo scrivere:

$$T(m) = \sum_{i=0}^{m-1} \sum_{j=i}^{m-1} (j-i+1) \leq \sum_{i=0}^{m-1} \sum_{j=i}^{m-1} m \leq \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} m = \sum_{i=0}^{m-1} m^2 = m^3 \leq c_1 m^3 \Leftrightarrow c_1 \geq 1$$

$\Rightarrow$  Scegliendo  $c_1 \geq 1$  ed  $m_0 \geq 1$ , si ha che  $T(m) = \Theta(m^3)$

- Verifichiamo se è anche  $\Omega(m^3)$ : per def.  $\exists c > 0, \exists m_0 \geq 1 : T(m) \geq c_2 m^3, \forall m \geq m_0$

$$T(m) = \sum_{i=0}^{m-1} \sum_{j=i}^{m-1} (j-i+1) \geq \sum_{i=0}^{m/2} \sum_{j=i}^{i+m/2-1} (j-i+1) \geq \sum_{i=0}^{m/2} \sum_{j=0}^{m/2} m/2 = \sum_{i=0}^{m/2} m^2/4 = \frac{m^3}{8} \geq c_2 m^3$$

$$\Leftrightarrow c_2 \leq \frac{1}{8} \Rightarrow T(m) = \Omega(m^3) \text{ per } c_2 \leq \frac{1}{8} \text{ ed } m_0 \geq 1$$

$\Rightarrow T(m) = \Theta(m^3)$ , ~~per c2 <= 1/8 ad m0 >= 1?~~

- PRIMA OTTIMIZZAZIONE (Contatore Somme):

def maxSum2(A):

$$\text{maxSoFar} = 0$$

for i in range(0, len(A)):

$$\text{sum} = 0$$

for j in range(i, len(A)):

$$\quad \text{sum} += A[j]$$

$$\quad \text{maxSoFar} = \max(\text{maxSoFar}, \text{sum}) \leftarrow$$

return maxSoFar

OPERAZIONE

DOMINANTE

$$T(m) = \sum_{i=0}^{m-1} m - i = \sum_{i=1}^m i = \frac{m(m-1)}{2} \Rightarrow T(m) = \Theta(m^2)$$

- SECONDA OTTIMIZZAZIONE (Divide et Impera):

Essendo ricorsivo, possiamo scrivere la RELAZIONE DI RICORRENZA:

$$T(m) = 2T\left(\frac{m}{2}\right) + m \Rightarrow \alpha = 2, b = 2, c_1 = 1, \beta = 1 \Rightarrow \alpha = \log_2 2 = 1$$

$$\Rightarrow \alpha = \beta = 1 \Rightarrow \text{Per il MASTER THEOREM: } T(m) = \Theta(m^{\log_2 2}) = \Theta(m \log m)$$

- TERZA OTTIMIZZAZIONE (Programmazione Dinamica):

OPERAZIONE DOMINANTE  $\rightarrow$  unico ciclo for  $\Rightarrow T(m) = \Theta(m)$ ,  $\rightarrow$  COSTO LINEARE

## ANALISI AMMORTIZZATA

E' una tecnica di analisi delle complessità che valuta il tempo richiesto (nel CASO PESSIMO) per eseguire una sequenza di operazioni su una STRUTTURA DATI:

- esistono operazioni più o meno costose;
- se le operazioni più costose sono meno frequenti, allora il loro costo può essere AMMORTIZZATO dalle operazioni meno costose.

• ANALISI DEL CASO MEDIO: probabilistica, su singole operazioni.

• ANALISI AMMORTIZZATA: deterministica, su operazioni multiple, nel caso pessimo.

### METODOLOGIE:

• METODO DELL'AGGREGAZIONE: (della Matematica)

Si calcola la complessità  $T(m)$  per eseguire  $m$  operazioni in sequenze nel CASO PESSIMO

• METODO DEGLI ACCANTONAMENTI: (dell'Economia)

Alle operazioni vengono assegnati COSTI AMMORTIZZATI che possono essere maggiori/minori del loro costo effettivo

• METODO DEL POTENZIALE: (della Fisica)

Lo stato del sistema viene descritto con una FUNZIONE DI POTENZIALE

### ESEMPIO: CONTATORE BINARIO

E' una STRUTTURA DATI definita in questo modo:

- contatore di  $K$  bit rappresentato da un vettore  $A[]$  di booleani (0/1)
- il bit meno significativo è in  $A[0]$ , il più significativo è in  $A[K-1]$
- il valore del contatore è  $x = \sum_{i=0}^{K-1} A[i] \cdot 2^i$
- E' prevista un'operazione INCREMENT() che incrementa il contatore di 1

INCREMENT( $A, K$ ):

$i \leftarrow 0$

while  $i < K$  and  $A[i] == 1$ :

$A[i] \leftarrow 0$

$i \leftarrow i + 1$

if  $i < K$  then

$A[i] \leftarrow 1$

## METODO DELL'AGGREGAZIONE:

Una chiamata ad `INCREMENT()` richiede tempo  $O(k)$  nel caso peggiore;

E' prevista una sola operazione: c'è quindi un'unica sequenza possibile (chiamate consecutive di `INCREMENT()`);

IL LIMITE SUPERIORE È  $T(m) = O(mk)$  per una sequenza di  $m$  incrementi.

Quanto ci sono ostacoli vicini?

Sono necessari  $K = \lceil \log m \rceil$  bit per rappresentare  $m$ .

Costo di 1 operazione:  $T(m)/m = O(k) \rightarrow$  ANALISI AMMORTIZZATA

Costo di  $m$  operazioni:  $T(m) = O(mk)$



→ Ci SIAMO ANDATI MOLTO LONTANO!

## SECONDA ANALISI:

Quanti bit vengono modificati?

- Il bit in  $A[0]$  viene modificato ad ogni incremento
- Il bit in  $A[1]$  ogni 2 incrementi
- Il bit in  $A[2]$  ogni 4 incrementi

- Il bit in  $A[i]$  ogni  $2^i$  incrementi

$$\Rightarrow \text{Costo Totale} = T(m) = \sum_{i=0}^{K-1} \left\lfloor \frac{m}{2^i} \right\rfloor \leq m \sum_{i=0}^{K-1} \frac{1}{2^i} \leq m \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = m \cdot \frac{1}{1-\frac{1}{2}} = 2m$$

COSTO AMMORTIZZATO:  $\frac{T(m)}{m} = \frac{2m}{m} = 2 = O(1)$

## METODO DEGLI AMMORTAMENTI

Si considerano tutte le operazioni possibili e, ad ognuna, si associa un COSTO AMMORTIZZATO POTENZIALMENTE DISTINTO:

può essere diverso dal costo effettivo, in quanto le operazioni meno costose vengono caricate di un costo aggiuntivo detto CREDITO;

$$\text{Costo Ammortizzato} = \text{costo effettivo} + \text{credito prodotto}$$

I crediti accumulati sono usati per pagare le operazioni più costose;

$$\text{costo Ammortizzato} = \text{costo effettivo} - \text{credito consumato}$$

### CASO DEL CONTATORE BINARIO

Costo effettivo di `INCREMENT()` =  $d$ , dove  $d = \#$  bit che cambiano valore

Costo Ammortizzato :  $\text{INCREMENT}() = 2 : 1$  per il cambio di un bit da 0 a 1 (c. effettivo)  
 $1$  per il futuro cambio dello stesso bit da 1 a 0

Quindi, in ogni istante, il CREDITO è uguale al numero di bit al momento impostati ad 1.

Costo Totale:  $O(m)$ ; Costo Ammortizzato:  $\frac{T(m)}{m} = O(1)$

## METODO DEL POTENZIALE

Si associa ad una struttura dati D una FUNZIONE DI POTENZIALE  $\Phi(D)$ :

- Le operazioni meno costose incrementano  $\Phi(D)$ ;
- Le operazioni più costose decrementano  $\Phi(D)$ ;

Il COSTO AMMORTIZZATO è dato dalla somma tra il costo effettivo e la variazione di potenziale:

$$e_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

dove  $D_i$  è il potenziale associato all' $i$ -esima operazione.

- Data una sequenza di  $m$  operazioni:

$$\begin{aligned} A &= \sum_{i=1}^m e_i = \sum_{i=1}^m c_i + \Phi(D_i) - \Phi(D_{i-1}) = \sum_{i=1}^m c_i + \sum_{i=1}^m (\Phi(D_i) - \Phi(D_{i-1})) = \\ &= C + \cancel{\Phi(D_1)} - \cancel{\Phi(D_0)} + \cancel{\Phi(D_2)} - \cancel{\Phi(D_1)} + \dots + \cancel{\Phi(D_m)} - \cancel{\Phi(D_{m-1})} = \\ &= C + \Phi(D_m) - \Phi(D_0) \end{aligned}$$

- Se la variazione  $\Phi(D_m) - \Phi(D_0) > 0$ , il costo ammortizzato è un LIMITE SUPERIORE del costo reale.

## CASO DEL CONTATORE BINARIO:

Scegliamo come funzione di potenziale  $\Phi(D)$  il numero di bit impostati ad 1 presenti nel contatore.

### • Operazione INCREMENT():

costo effettivo:  $1+t$

variazione di potenziale:  $1-t$

$$\text{COSTO AMMORTIZZATO: } 1+\frac{t}{2} + 1-\frac{t}{2} = 2$$

-  $t$  è il numero di bit impostati ad 1 incontrati e portare del meno significativo, prima di incontrare un bit impostato a 0.

Alle' inizio,  $\Phi(D_0) = 0$  (nessun bit a 1)

Alle fine,  $\Phi(D_m) \geq 0$

da DIFFERENZA DI POTENZIALE E' NON NEGATIVA  $\rightarrow$  il costo ammortizzato (2) è un limite superiore al costo effettivo.

## PROGRAMMAZIONE DINAMICA

E' un metodo che spezza il problema in sottoproblemi im menire RICORSIVA; perciò, ogni sottoproblema viene risolto una volta sola e la soluzione viene salvata in una tabella, cui si accede se c'è bisogno di risolvere un sottoproblema già risolto in precedenza.

### • ESEMPIO DI DOMINO LINEARE.

Tessere di dimensione  $2 \times 1$ . Scrivere un algoritmo che prende in input un intero  $m$  e restituisce il numero di possibili combinazioni di tessere in un rettangolo  $2 \times m$ .

Se NON ho tessere:  $\rightarrow$  1 sola disposizione

Se ho 1 tessere:  $\rightarrow$  1 disposizione (verticale).

Se ho  $m \geq 2$  tessere:  $\begin{cases} \text{metto l'ultima tessere verticale e risolvo per } m-1 \\ \text{metto 2 tessere orizzontali e risolvo per } m-2 \end{cases}$

$$\Rightarrow \text{RELAZIONE DI RICORRENZA: } D(m) = \begin{cases} 1 & \text{se } m \leq 1 \\ D(m-2) + D(m-1) & \text{se } m \geq 2 \end{cases}$$

E' fibonacci! Il numero di disposizioni è:  $a_m = a_{m-1} + a_{m-2}$ , con  $a_0 = a_1 = 1$

E' una Relazione di ricorrenza omogenea con  $c_1 = 1$  e  $c_2 = 1$ , già trovata in Fibonacci:

$$a_m = \frac{\varphi^m - (-\varphi)^{-m}}{\sqrt{5}}$$

Avevamo visto che lo stesso valore, nell'albero delle chiamate ricorsive, veniva calcolato più volte! Ora vediamo come fare di meglio con la programmazione dinamica.

### • FIBONACCI : 3° APPROCCIO (Programmazione Diminutiva) :

Solviamo i numeri della sequenza in un vettore  $\text{Fib}[]$  e calcoliamo  $F[m] = F[m-1] + F[m-2]$ .

```
def fibonacc3(m):
    Fib = [0]*m
    Fib[0] = 1
    Fib[1] = 1
    for i in range(2, m):
        Fib[i] = Fib[i-1] + Fib[i-2]
    return Fib[m-1]
```

• Complessità in TEMPO :  $\Theta(m) = T(m)$

• Complessità in SPAZIO :  $\Theta(m) = S(m)$

Ma, si può fare meglio in SPAZIO!

### • FIBONACCI : 4° APPROCCIO

Perché ricordare tutti i numeri di Fibonacci quando ci servono solo gli ultimi 2?

```
def fibonacc4(m):
    F0 = 1
    F1 = 1
    F2 = 1
    for i in range(2, m):
        F0 = F1
        F1 = F2
        F2 = F0 + F1
    return F2
```

Ora le complessità in SPAZIO è  $S(m) = \Theta(1)$ .

Ma me siamo proprio sicuri?

Quanti bit servono per memorizzare  $F(m)$ ? Di certo  $\log F(m) = \Theta(m)$

Inoltre, quanto costa sommare 2 numeri di Fibonacci consecutivi?

Vediamo ora un approccio che sfrutta proprietà delle matrici

## FIBONACCI: 5° APPROCCIO

Utilizziamo l'esponentiale di matrici:  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^m = \begin{bmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{bmatrix}$

• Ovvio per  $m=1$

• Sia vero per  $K$ :  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^K = \begin{bmatrix} F_{K+1} & F_K \\ F_K & F_{K-1} \end{bmatrix}$ ; moltiplichiamo entrambi i membri per  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ :

$$\Rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{K+1} = \begin{bmatrix} F_{K+2} + F_K & F_{K+1} + 0 \\ F_K + F_{K-1} & F_K + 0 \end{bmatrix} = \begin{bmatrix} F_{K+2} & F_{K+1} \\ F_{K+1} & F_K \end{bmatrix} \rightarrow \text{OK!} \Rightarrow \underline{\text{Vale } F_{m+2}}$$

$\Rightarrow$  def FIBONACCI 5(m):

$$M \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

for  $i \leftarrow 1$  to  $m$ :

$$M \leftarrow M \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

return  $M[0][0]$

Siamo finiti però in un ciclo for del costo lineare.

Vediamo se riusciamo a migliorarlo.

## FIBONACCI: 6° APPROCCIO

Possiamo calcolare le potenze  $m$ -esime elevando al quadrato le potenze  $(\frac{m}{2})$ -esime; se  $m$  è dispari con un'ulteriore moltiplicazione. È un'applicazione del DIVIDE ET IMPERA:

$$A^m = \begin{cases} A(A^2)^{\frac{m-1}{2}} & \text{se } m \text{ è dispari} \\ (A^2)^{\frac{m}{2}} & \text{se } m \text{ è pari} \end{cases}$$

POTENZA DI MATRICE (A, k):

if  $k \leq 1$  then:

$$M \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

else:

$$M \leftarrow \text{POTENZA DI MATRICE}(A, \lfloor K/2 \rfloor)$$

$$M \leftarrow M \cdot M$$

if  $K$  is odd then:

$$M \leftarrow M \cdot A$$

return  $M$

FIBONACCI 6(m):

$$A \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$M \leftarrow \text{POTENZA DI MATRICE}(A, m-1)$$

return  $M[0][0]$

• Il tempo di esecuzione è:  $T(m) = O(\log m)$ .

## PROBLEMA DEL KNAPSACK

Dato un insieme di oggetti, ciascuno caratterizzato da un PESO ed un PROFITTO, ed uno ZAINO con un limite di CAPACITA', individuare un sottoinsieme di oggetti che:

- totalizzano un peso contenibile nello zaino;
- massimizzano il valore degli oggetti inseriti nello zaino.

### • Input:

- (1) un vettore  $w$ , dove  $w[i]$  è il PESO dell'oggetto  $i$ -esimo;
- (2) un vettore  $p$ , dove  $p[i]$  è il PROFITTO dell'oggetto  $i$ -esimo;
- (3) la CAPACITA'  $C$  dello zaino.

### • Output:

Un insieme  $S \subseteq \{1, \dots, m\}$  tale che:

- $w(S) = \sum_{i \in S} w[i] \leq C$
- $p(S) = \sum_{i \in S} p[i]$

Dato uno zaino di capacità  $C$  ed  $m$  oggetti di peso  $w$  e profitto  $p$ ,  $DP[i][c]$  è il MASSIMO PROFITTO che si può ottenere dai primi  $i$  oggetti ( $i \leq m$ ) con uno zaino di capacità  $c$  ( $c \leq C$ ).

IL MASSIMO ASSOLUTO sarà  $DP[m][C]$ .

### PARTE RICORSIVA:

Partiamo dall'ultimo elemento:

- se non lo prendiamo:  $DP[i][c] = DP[i-1][c]$   
La capacità non cambia e non c'è profitto;
- se lo prendiamo:  $DP[i][c] = DP[i-1][c-w[i]] + p[i]$   
diminuisce la capacità ed aumenta il profitto;
- COME SCEGLIERE?:  $DP[i][c] = \max(DP[i-1][c], DP[i-1][c-w[i]] + p[i])$ .

Se non si hanno più oggetti o non si ha più capacità, il profitto è nullo.

Se la capacità dovesse essere negativa, il problema non avrebbe senso!

- $DP[i][c] = \begin{cases} 0 & \text{se } i=0 \text{ oppure } c=0 \\ -\infty & \text{se } c < 0 \\ \max(DP[i-1][c], DP[i-1][c-w[i]] + p[i]) & \text{altrimenti} \end{cases}$

KNAPSACK ( $w, p, m, C$ ):

$$DP = [0 \dots m][0 \dots C]$$

for  $i \leftarrow 0$  to  $m$ :

$$DP[i][0] = 0 \quad \# \text{ setting 1st column to 0}$$

for  $c \leftarrow 0$  to  $C$ :

$$DP[0][c] = 0 \quad \# \text{ setting 1st line to 0}$$

for  $i \leftarrow 1$  to  $m$ :

for  $c \leftarrow 1$  to  $C$ :

if  $w[i] \leq c$  then:

$$DP[i][c] = \max(DP[i-1][c], DP[i-1][c-w[i]] + p[i])$$

else:

$$DP[i][c] = DP[i-1][c]$$

return  $DP[m][c]$

• Esempio:

$$w = [4, 2, 3, 4] \quad C = 9 \quad m = 4$$

$$p = [10, 7, 8, 6]$$

	C									
i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

$$\Rightarrow DP[4][9] = 25$$

La complessità computazionale è:  $T(m) = O(mC)$

E' un ALGORITMO POLINOMIALE? NO!!!, poiché, per rappresentare  $C$  sono necessari  $k = \log C$  bit.  $\Rightarrow T(m) = O(m2^k) \rightarrow$  E' un ALGORITMO PSEUDO-POLINOMIALE, poiché non dipende solo da  $m$ .

## ALGORITMI GREEDY

Gli ALGORITMI GREEDY ("ingozzati") si basano su una "SCELTA", tuttavia, a differenza degli algoritmi visti in precedenza, non analizzano tutte le possibili scelte per fare le loro scelte:

- identificano quelle che sembra, in QUELLA FASE DELL'ESECUZIONE, le scelte ottime;
- non è detto che si tratti dell'OTTIMO GLOBALE!

In alcuni casi si puo' dimostrare che l'ottimo locale corrisponde all'ottimo globale; altri esempi si rischia di individuare una soluzione subottima.

### • PROBLEMA DEL RESTO:

INPUT: - un vettore di interi positivi  $t[1 \dots m]$  dei tagli di monete disponibili;  
- intero  $R$  pari al resto da dare.

PROBLEMA: Assumendo di avere monete illimitate per ogni taglio, trovare il più piccolo numero di monete per dare il resto  $R$ :

Trovare un vettore  $x[i]$  di interi non negativi tali che:

$$R = \sum_{i=1}^m x[i] \cdot t[i] \quad \text{e} \quad m = \sum_{i=1}^m x[i] \text{ ha valore minimo.}$$

Risolto con la PROGRAMMAZIONE DINAMICA, la complessità è  $T(m) = O(mR)$ .

### • Appuccio GREEDY:

Sia  $S(i)$  il problema di dare un resto pari a  $i$ .

(1) Si seleziono le monete  $j$  più grande, tali che  $t[j] \leq R = i$

(2) Si risolve poi il problema  $S(i-t[j])$ :

RESTO ( $t, m, R$ ):

for  $i \leftarrow 1$  to  $m$ :

$$x[i] = \lfloor R/t[i] \rfloor$$

$$R = R - x[i] \cdot t[i]$$

return  $x$

Se  $t[i]$  è già ordinato, il costo

$$\Rightarrow \underbrace{i \in T(m) = O(m)}$$

## • PROBLEMA DELLO SCHEDULING:

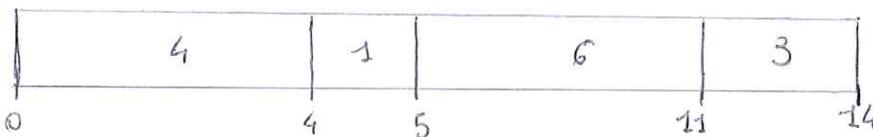
Supponiamo di avere un processore ed  $m$  JOB da eseguire, ciascuno con un tempo di esecuzione  $t[i]$ ,  $i=1, \dots, m$ .

- Trovare una sequenza (permutazione) di esecuzione che minimizzi il tempo di completamento medio.

Dato un vettore  $A[1 \dots m]$  contenente una sequenza  $\{1, \dots, m\}$ , il tempo di completamento del  $k$ -esimo JOB è dato da:

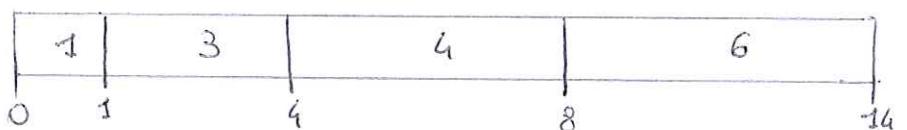
$$T_A(k) = \sum_{i=1}^k t[A[i]] , \text{ cioè il tempo d'esecuzione di tutti i precedenti più il } k\text{-esimo.}$$

Supponiamo di avere 4 job di durata 4, 1, 6, 3:



Tempo di completamento medio:  $(4+5+11+14)/4 = \frac{34}{4} = 8,5$

APPROCCIO GREEDY: Scegliamo per primo il job con durata minore, poiché, intuitivamente, peserà di meno nelle sommatorie.



Tempo di completamento medio:  $(1+4+8+14)/4 = \frac{27}{4} = 6,75 < 8,5$

## BACKTRACKING

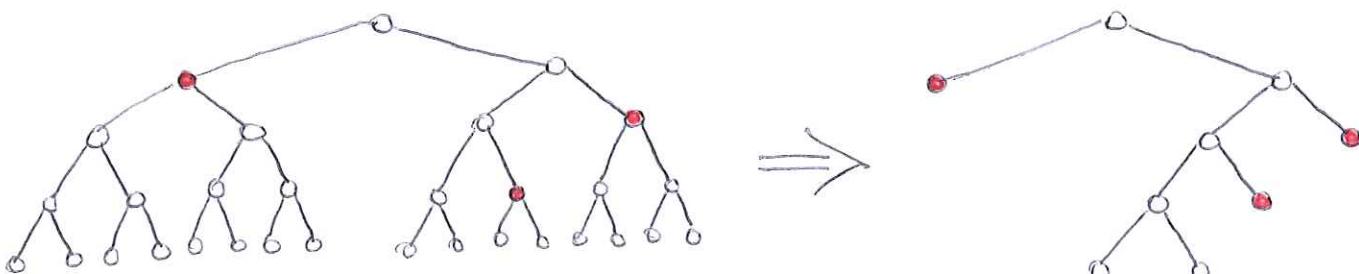
È una TECNICA ALGORITMICA che si adatta a classi di problemi decisionali, di ricerca, di ottimizzazione.

Si basa su un'idea semplice: (1) Prove e fare qualcosa;  
 (2) Se non va bene, butta via un pezzo del lavoro fatto;  
 (3) Ritenta, sarà più fortunato.

Si tratta di una tecnica per esplorare in maniera sistematica uno spazio di ricerca.

## - PRUNING (Potatura):

Consente di escludere una parte dello spazio di ricerca che non fornisce soluzioni ammissibili.



## ESEMPIO: il Problema delle 8 Regine

Problema presentato nel 1848 da Max Bezzel:

Posizionare  $m$  REGINE su una Scacchiera  $m \times m$  in maniera tale che nessuna regina possa mangiare nessun'altra regina.

GAUSS, per  $m=8$ , trovò 92 delle 92 soluzioni possibili.

- Utilizziamo un vettore  $S[1..m]$  per rappresentare le colonne: in ogni colonna ci deve essere una e una sola regina!
- $S[i]$  identifica la riga della regina nella colonna  $i$ -esima (per esempio, se  $S[3]=5$ , vuol dire che alla 5<sup>a</sup> riga della 3<sup>a</sup> colonna c'è una regina);
- d' algoritmo termina quando  $i=M$ ;
- Utilizziamo il PRUNING per eliminare le diagonali non disponibili.

### Implementazione:

```
def solve(m, i, e, b, c): # e = S[1..m], b, c are the diagonals vectors.  
    if i < m:  
        for j in range(m):  
            if (j not in e) and (i+j not in b) and (i-j not in c):  
                for solution in solve(m, i+1, e+[j], b+[i+j], c+[i-j]):  
                    yield solution  
    else:  
        yield e  
for solution in solve(8, 0, [], [], []):  
    print(solution)
```