Analisi della Complessità e Tecniche Algoritmiche

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

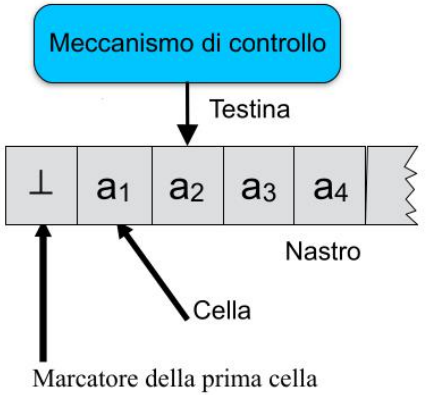
Modelli di calcolo

- I *modelli di calcolo* permettono di rappresentare in maniera astratta un calcolatore
- Questi devono avere alcune proprietà (a volte in contrasto l'una con l'altra) per essere appetibili per l'analisi della complessità di un algoritmo
  - *Astrazione*: il modello deve permettere di nascondere i dettagli
  - *Realismo*: il modello deve riflettere la situazione reale
  - *Potenza matematica*: il modello deve permettere di trarre conclusioni *formali* sul costo

- Nella storia sono stati proposti molteplici modelli di calcolo

La Macchina di Turing

- Una macchina ideale che manipola i dati contenuti su un nastro di lunghezza infinita, secondo un insieme prefissato di regole
- Ad ogni passo d'esecuzione la macchina:
  - Legge il simbolo sotto la testina
  - Modifica il proprio stato interno
  - Scrive un nuovo simbolo nella cella
  - Muove la testina a destra o a sinistra



- 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
- Esegue le istruzioni di un algoritmo in sequenza
- Caratterizzata da una memoria ad accesso casuale di dimensione infinita, accessibile in tempo costante, organizzata in celle
- Ciascuna cella può contenere un intero qualsiasi
- Singolo processore
- Supporta le seguenti operazioni primitive a ***costo unitario***:
  - Assegnazione
  - Operazione aritmetica
  - Confronto
  - Lettura/Scrittura

Modello di costo RAM

- Costo di operazioni complesse:
  - **Ciclo**: somma tra il costo del test della condizione di ripetizione ed il costo del corpo del ciclo
  - **if/then/else**: costo del test della/delle condizioni più il costo del blocco (a seconda del caso)
  - **Chiamata a funzione**: somma del costo di tutte le istruzioni della funzione
- Costo totale:
  - Somma di tutti i costi

Un esempio

7	1	4

```
arrayMax(A, n):
    currentMax ← A[o]                          Costo: 1
    for i ← 0 to n:                            Costo: 2 · 3 + 2
        if A[i] > currentMax then              Costo = 1 · 3
            currentMax ← A[i]                  Costo = 0
    return currentMax                          Costo = 1
```

- Costo totale = 13

Un esempio

1	8	6	3	4

```
arrayMax(A, n):
    currentMax ← A[0]                          Costo: 1
    for i ← 0 to n:                            Costo: 2 · 5 + 2
        if A[i] > currentMax then              Costo = 1 · 5
            currentMax ← A[i]                  Costo = 1
    return currentMax                          Costo = 1
```

- Costo totale = 20

Modello di costo RAM

- Perché si tratta di un modello **accettabile**?
- Il costo di un'operazione è valutato a meno di un fattore costante (possibilmente *arbitrariamente grande*) perché:
  - Il numero di operazioni elementari per ciascuna operazione è finito
  - Ogni variabile occupa una quantità finita di memoria
  - I tempi di accesso a due variabili diverse sono comunque legati da una costante
- **Vantaggi**: prescinde dall'architettura hardware/software e dal linguaggio di programmazione
- **Svantaggi**: l'indicazione che si ottiene è *qualitativa*, non esatta

Modello di costo RAM

- Nell'esempio precedente, **i risultati dipendono dal particolare input**, anche per la stessa dimensione di vettore
- È auspicabile una misura *dipendente* dalla dimensione dell'input, *indipendente* dallo specifico input considerato
- **Analisi del caso pessimo**: si considera il costo di esecuzione considerando, data la dimensione di un input, la sua configurazione più sfavorevole per l'algoritmo
- **Analisi del caso medio**: si considera il costo medio rispetto ad una distribuzione delle configurazioni dell'input
  - È necessario conoscere tale distribuzione
- In entrambi i casi occorre definire la **dimensione dell'input**.

Teoria e Pratica possono essere differenti

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	Scelte implementative non banali
La memoria non ha limiti (modello RAM) ed ha costo costante	Grandi quantità di dati ed effetti dovuti alla gerarchia 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, allocatori, cache, ...)
Un singolo processore	Architetture multicore

Dimensione dell'input

- Per ciascun problema occorre indicare la dimensione dell'input: è in base ad essa 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

Dimensione dell'input

- In molti casi possiamo assumere che gli elementi siano rappresentati da un numero costante di bit
  - Le due misure coincidono a meno di una costante moltiplicativa
  - Nei casi dubbi, una misura ragionevole è il numero di bit necessari a rappresentare l'input

- Esempio: si consideri il problema di determinare la scomposizione in fattori primi di un numero intero
  - La dimensione dell'input può essere il numero di bit necessari a rappresentare un intero.

Un esempio di analisi del caso pessimo

1	4	6	7	8

```
arrayMax(A, n):
    currentMax ← A[o]                          Costo: 1
    for i ← 0 to n:                            Costo: 2 · 5 + 2
            if A[i] > currentMax then          Costo = 1 · 5
                    currentMax ← A[i]          Costo = 4
    return currentMax                          Costo = 1
```

- Costo totale = 23

Un esempio di analisi del caso pessimo

- Generalizzando si ottiene quanto segue:

```
ARRAYMAX(A, n):
    currentMax ← A[o]                          COSTO: 1
    for i ← 0 to n:                            COSTO: 2 · n + 2
        if A[i] > currentMax then              COSTO = 1 · n
            currentMax ← A[i]                  COSTO = n - 1
    return currentMax                          COSTO = 1
```

- Costo totale = $1 + 2n + 2 + n + (n - 1) + 1 = \mathbf{4n + 3}$

Considerazioni sull'analisi del caso pessimo

- Il costo che abbiamo calcolato nel caso pessimo dell'algoritmo ARRAYMAX è $4n + 3$

```
ARRAYMAX(A, n):
    currentMax ← A[o]
    for i ← 0 to n:
            if A[i] > currentMax then
                    currentMax ← A[i]
    return currentMax
```

- Si tratta comunque di un algoritmo molto semplice
  - Se si considerano tutte le costanti, l'analisi può diventare *eccessivamente complessa*

La Ricorsione

La ricorsione

- Una funzione è detta *ricorsiva* se chiama se stessa
- Se due funzioni si chiamano l'un l'altra, sono dette *mutuamente ricorsive*
- Una funzione ricorsiva è in grado di individuare direttamente la soluzione di alcuni casi particolari del problema: i **passi base**
  - In questi casi, una funzione ricorsiva restituisce dei valori
- Se una funzione ricorsiva non individua uno dei passi base, allora chiama se stessa (**passo ricorsivo**) su un *sottoproblema* o su dei *dati semplificati*
- Le soluzioni ricorsive sono spesso eleganti, ma non sempre le più efficienti
- In generale, è intercambiabile con una funzione iterativa

La ricorsione

- La ricorsione trae le sue origini dal *metodo induttivo*
  - Letteralmente “portar dentro”, “chiamare a sé”, nel senso di *ricondurre*
- È un procedimento che, partendo da singoli casi particolari cerca di stabilire una legge universale
- Un esempio (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 vedrò sarà probabilmente nero
  - *Conclusione 2*: tutti i corvi sono probabilmente neri

Esempio: somma dei primi *n* interi positivi

- Principio di induzione:
  - un enunciato sui numeri naturali che consente di provare che una certa proprietà è valida per tutti i numeri interi

- Se P è una proprietà che vale per il numero 0, e se $P(n) \Longrightarrow P(n+1)$ per ogni n, allora $P(n)$ vale per ogni n
- Definizione induttiva dell'insieme $\mathbb{N}$:
  1. $0 \in \mathbb{N}$;
  2. $n \in \mathbb{N} \Longrightarrow (n+1) \in \mathbb{N}$;
  3. null'altro è in $\mathbb{N}$.
- La seconda proposizione è il passo induttivo, la prima è il passo base, mentre la terza afferma che $\mathbb{N}$ è il più piccolo insieme che soddisfa 1 e 2

Esempio: somma dei primi *n* interi positivi

- Dimostriamo che $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Nel caso in cui $n = 0$ (caso base) la prova è diretta.
- Supponendo che $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ sia vera (ipotesi induttiva), allora: $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$
- Quindi, la tesi vale per $n + 1$. Per il principio di induzione, se ne conclude che vale per ogni n.

- Come trasformare questo concetto in un algoritmo?

Esempio: somma dei primi *n* interi positivi

- La somma dei primi 0 interi positivi vale 0
- La somma dei primi n interi positivi è uguale alla somma dei primi $n-1$ interi positivi più n.

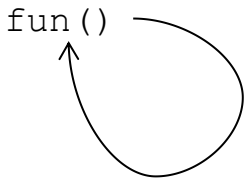
```
def sum(n):
    if (n == 0):
        return 0
    else:
        return (n + sum(n-1))
```

- Questo processo viene definito anche "*per accumulazione*"

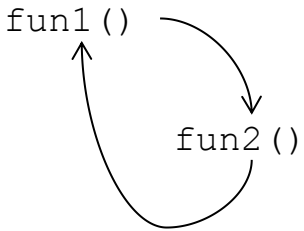
Ricorsione diretta e indiretta

- Se una funzione chiama sé stessa direttamente si parla di *ricorsione diretta*
- Se intercorrono più chiamate a funzione prima di tornare alla funzione ricorsiva, si parla di *ricorsione indiretta*

```
def fun ():
    …
    fun ()
    …
```




```
def fun1():
    fun2()

def fun2():
    fun1()

fun1()
```



Ricorsione terminale/non terminale

- *Tail recursion*: prima si effettuano le operazioni della procedura ricorsiva e solo in ultimo si effettua la chiamata
  - Beneficia della *Tail Call Optimization* (non in Python)

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

```
recSum(5)
5 + recSum(4)
5 + (4 + recSum(3))
5 + (4 + (3 + recSum(2)))
5 + (4 + (3 + (2 + recSum(1))))
5 + (4 + (3 + (2 + 1)))
15
```

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

```
tailRecSum(5, 0)
tailRecSum(4, 5)
tailRecSum(3, 9)
tailRecSum(2, 12)
tailRecSum(1, 14)
tailRecSum(0, 15)
15
```

Il calcolo del fattoriale

- Si può applicare la ricorsione in maniera semplice per calcolare il fattoriale di un numero:
  - $n! := \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$

- Questo operatore è facilmente definibile in maniera ricorsiva:
  - $n! := \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \geq 1 \end{cases}$

- Una possibile implementazione è immediata:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Il calcolo del fattoriale

- L'esecuzione di questa procedura determina l'attivazione dei seguenti record sullo stack:

  - fact(3)
  - 3 × fact(2)
  - 3 × 2 × fact(1)
  - 3 × 2 × 1 × fact(0)
  - 3 × 2 × 1 × 1
  - 6

Il calcolo del fattoriale: soluzione iterativa

- È sempre possibile trasformare un algoritmo ricorsivo in un algoritmo iterativo
- In alcuni casi, è possibile "accumulare" esplicitamente i valori calcolati dalle funzioni ricorsive:

```
def fact(n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f
```

- Quale delle due implementazioni è "migliore"?

Generazione degli anagrammi

- Un anagramma si ottiene scambiando tra loro le lettere di una parola data in input
  - È un caso particolare della generazione di tutte le *permutazioni* di una sequenza
- Un approccio:
  1. Si spezza la parola in input, rimuovendo il primo carattere
  2. Si generano nuove parole inserendo il primo carattere in tutte le altre possibili posizioni degli anagrammi generati dalla parte restante della parola in input
- Un esempio: “abc”
  - Si rimuove “a”, la restante parte è “bc”
  - Gli anagrammi di “bc” sono “bc” e “cb”
  - Inserendo “a” in tutte le posizioni otteniamo:
    - [“abc”, “bac”, “bca”, “acb”, “cab”, “cba”]

Generazione degli anagrammi

```
def anagrams(s):
    ans = []
    for w in anagrams(s[1:]):
        for pos in range(len(w)+1):
            ans.append(w[:pos]+s[0]+w[pos:])
    return ans
```

- È corretto questo algoritmo?

```
RecursionError: maximum recursion depth exceeded
```

Generazione degli anagrammi

```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

La funzione di Ackermann

- Si tratta di una funzione ricorsiva inventata dal matematico tedesco Wilhelm Friedrich Ackermann (1896–1962), definita come:

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

- Si tratta di una funzione *totalmente ricorsiva*, ovvero è definita per tutte le coppie di argomenti non negativi ed è calcolabile.
- Allo stesso tempo, è una *funzione ricorsiva non primitiva*, ovvero una funzione che non può essere implementata da un programma che utilizzi soltanto cicli
  - il numero di iterazioni non è determinabile prima di iniziare i cicli
  - Un effetto è che questa funzione cresce in maniera incredibilmente veloce

La funzione di Ackermann

- L'implementazione è estremamente semplice:

```
def ackermann(m,n):
    if m == 0:
        return n+1
    elif m > 0 and n == 0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1,ackermann(m,n-1))
```

- Anche input semplici, come `ackermann(4,1)` generano un errore di tipo "`maximum recursion depth exceeded in comparison`"

La funzione di Ackermann

- È possibile aumentare il limite di profondità della ricorsione dell'interprete Python con il seguente comando:
  - `sys.setrecursionlimit(100000)`
- Con questa configurazione, `ackermann(4,1)` termina:
  - Il risultato è 65533
  - Il tempo d'esecuzione è pari ad alcuni minuti
- `ackermann(4,2)` non termina (crash dell'applicazione)
- A volte, nell'analisi degli algoritmi, si incontra la funzione inversa di Ackermann
  - cresce molto molto lentamente
  - ha un valore <5 praticamente per tutti gli input trattabili
- Quanto “in fretta” cresce questa funzione?

La funzione di Ackermann

- La crescita di questa funzione è molto rapida
- Per `ackermann(4,3)`, la sequenza di chiamate viene mostrata nella figura a fianco
- `ackermann(6,0)` ha come risultato $2 \uparrow\uparrow\uparrow\uparrow 3 - 3$
- $\uparrow$ è la notazione a frecce di Knuth

$$
\begin{aligned}
A(4,3) &= A(3, A(4,2)) \\
&= A(3, A(3, A(4,1))) \\
&= A(3, A(3, A(3, A(4,0)))) \\
&= A(3, A(3, A(3, A(3,1)))) \\
&= A(3, A(3, A(3, A(2, A(3,0))))) \\
&= A(3, A(3, A(3, A(2, A(2,1))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(2,0)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(1,1)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(1,0))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(0,1))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0,2)))))) \\
&= A(3, A(3, A(3, A(2, A(1,3))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(1,2)))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(1,1))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(1,0)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(0,1)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0,2))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0,3)))))) \\
&= A(3, A(3, A(3, A(2, A(0,4))))) \\
&= A(3, A(3, A(3, A(2,5)))) \\
&= \dots \\
&= A(3, A(3, A(3,13))) \\
&= \dots \\
&= A(3, A(3,65533)) \\
&= \dots \\
&= A(3, 2^{65536} - 3) \\
&= \dots \\
&= 2^{2^{65536}} - 3.
\end{aligned}
$$

Notazione a frecce di Knuth

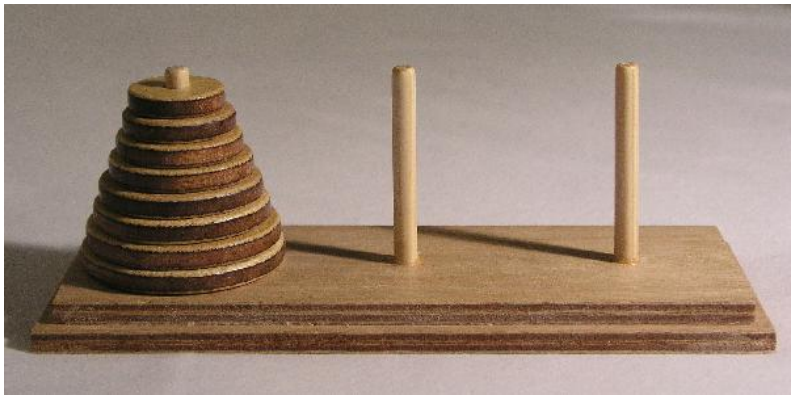
- Introdotta nel 1976 da Donald Knuth, per rappresentare in maniera compatta interi molto molto grandi
- Anch'essa ha una definizione ricorsiva:

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

- Una singola freccia rappresenta il calcolo di un esponenziale:
  - $2 \uparrow 4 = 2 \cdot \big(2 \cdot (2 \cdot 2)\big) = 2^4 = 16$
- Le frecce aggiuntive indicano l'iterazione dell'operazione associata con meno frecce:
  - $2 \uparrow\uparrow 4 = 2 \uparrow \big(2 \uparrow (2 \uparrow 2)\big) = 2^{2^{2^2}} = 65536$

Le torri di Hanoi

- Anche chiamate “Le torri di Brahma”, è un problema definito dal matematico francese Édouard Lucas nel 1883
- Il gioco comincia con un insieme di dischi impilati su un palo in ordine crescente di diametro
- L'obiettivo del gioco è quello di spostare la pila su un altro palo, seguendo tre semplici regole:
  1. Si può muovere un disco per volta;
  2. Si può spostare solo il disco che si trova in cima ad una pila;
  3. Non si può mettere un disco di diametro maggiore su un disco di diametro minore;



Le torri di Hanoi: casi particolari

- Un solo disco:
  - Sposta il disco da A a C
- Due dischi:
  - Sposta il disco da A a B
  - Sposta il disco da A a C
  - Sposta il disco da B a C
- Tre dischi:
  - Per spostare il disco più largo in C, occorre prima togliere di mezzo i due dischi più piccoli. Questi dischi formano una piramide di due dischi, che sappiamo spostare
  - Sposta una torre da due da A a B
  - Sposta un disco da A a C
  - Sposta una torre da due da B a C

Le torri di Hanoi: un algoritmo ricorsivo

- Algoritmo: sposta una torre di n dischi dalla sorgente alla destinazione utilizzando un punto d'appoggio:
  1. Sposta una torre di n-1 dischi dalla sorgente al punto d'appoggio;
  2. Sposta un disco dalla sorgente alla destinazione;
  3. Sposta una torre di n-1 dischi dal punto d'appoggio alla destinazione.

Le torri di Hanoi: un algoritmo ricorsivo

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print("Move disk from", source, "to", dest + ".")
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)


def hanoi(n):
    moveTower(n, "A", "C", "B")
```

Le torri di Hanoi: un algoritmo ricorsivo

- Anche se la definizione è semplice, si tratta di un problema complesso da risolvere

dischi	passi
1	1
2	3
3	7
4	15
5	31

- In generale, per risolvere un problema con n dischi sono necessari $2^n - 1$ passi
- Se consideriamo 64 dischi ed uno spostamento di un disco al secondo, sarebbero necessari 580 milioni di anni per risolvere il gioco

La successione di Fibonacci

- Leonardo da Pisa (noto anche come Fibonacci) studiò molti problemi nella sua vita
- Ad esempio:

Problema: dinamica delle popolazioni

- Quanto velocemente si espande una popolazione di conigli sotto appropriate condizioni?

- Ovverosia: se portiamo una coppia di conigli in un'isola deserta, quante coppie si avranno nell'anno n?

La successione di Fibonacci

- Modello e ipotesi sui conigli:
  - una coppia di conigli genera una coppia di coniglietti ogni anno
  - i conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita
  - i conigli sono immortali
- La terza ipotesi potrebbe sembrare irrealistica: si tratta di una semplificazione per rendere più trattabile il problema
- F_n: coppie presenti nell'anno n:
  - $F_1 = 1$ (*coppia iniziale di conigli*)
  - $F_2 = 1$ (*coppia iniziale ancora troppo giovane per procreare*)
  - $F_3 = 2$ (*prima nuova coppia di coniglietti*)
  - $F_4 = 3$ (*seconda nuova coppia di coniglietti*)
  - $F_5 = 5$ (*prima coppia di nipotini*)

La successione di Fibonacci

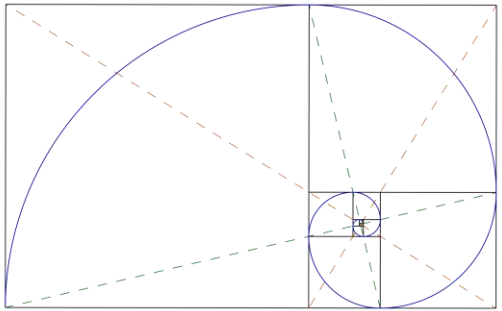
- In generale, in un certo anno si osserveranno tutte le coppie di conigli dell'anno precedente, più una nuova coppia di conigli presente nell'anno precedente
- Si può quindi definire una *relazione di ricorrenza*:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1,2 \end{cases}$$

- Questa relazione di ricorrenza definisce un *problema computazionale*
- Quale può essere un algoritmo che risolve tale problema computazionale?

La successione di Fibonacci

- La relazione di ricorrenza che abbiamo definito può essere risolta utilizzando la *formula di Binet* (1843):
  - $\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \varphi$, dove $\varphi = \frac{1+\sqrt{5}}{2}$ (sezione aurea di un segmento)
  - Il rapporto tra un numero di Fibonacci e il suo successivo tende al reciproco della sezione aurea $\frac{1}{\varphi}$
  - per φ valgono le seguenti relazioni:
    - $\varphi - 1 = \frac{1}{\varphi} = \frac{-1+\sqrt{5}}{2}$
    - $1 - \varphi = -\frac{1}{\varphi} = \frac{1-\sqrt{5}}{2}$
  - Si ha pertanto che l'n-esimo numero di Fibonacci corrisponde a:



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

n-esimo numero di Fibonacci: primo approccio

- Questa soluzione in forma chiusa ci permette di definire il primo algoritmo per il calcolo dell'n-esimo numero di Fibonacci:

FIBONACCI1(n):
return $\frac{1}{\sqrt{5}}(\varphi^n - (-\varphi)^{-n})$

- Quali sono i vantaggi di questa soluzione?

- Cosa non va in questa soluzione?

n-esimo numero di Fibonacci: primo approccio

- Questa soluzione in forma chiusa ci permette di definire il primo algoritmo per il calcolo dell'n-esimo numero di Fibonacci:

FIBONACCI1(n):
return $\frac{1}{\sqrt{5}}(\varphi^n - (-\varphi)^{-n})$

- Quali sono i vantaggi di questa soluzione?
  - Ripensiamo al costo computazionale: si tratta di un algoritmo a costo costante
- Cosa non va in questa soluzione?
  - Ripensiamo alla "pratica": si commettono errori di approssimazione

n-esimo numero di Fibonacci: primo approccio

```
phy = 1.618
def fibonacci1(n):
      return 1/math.sqrt(5) * (math.pow(phy, n) - math.pow(-phy, -n))
```

n	fibonacci1(n)	arrotondamento	F_n
1	0.99999060586254	1	1
3	1.9998872716742242	2	2
16	986.668322168999	987	987
18	2583.023133876576	2583	2584

n-esimo numero di Fibonacci: secondo approccio

- Il primo tentativo si è rivelato computazionalmente efficiente, ma la "pratica" ha mostrato che la soluzione genera un errore di approssimazione troppo elevato
- Possiamo sfruttare direttamente la definizione ricorsiva che ha portato alla realizzazione del primo algoritmo:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1,2 \end{cases}$$

- Otteniamo quindi il seguente algoritmo:

```
def fibonacci2(n):
    if n <= 2:
        return 1
    else:
        return fibonacci2(n-1) + fibonacci2(n-2)
```

- È una soluzione accettabile?

n-esimo numero di Fibonacci: secondo approccio

- Analizziamo il costo secondo il modello RAM:

```
FIBONACCI2(n):
    if n ≤ 2 then                                          COSTO: 1
        return 1                                           COSTO: 1
    else
        return FIBONACCI2(n-1) + FIBONACCI2(n-2)           COSTO: F(n-1)+F(n-2)
```

n-esimo numero di Fibonacci: secondo approccio

- Analizziamo il costo secondo il modello RAM:

```
FIBONACCI2(n):
    if n ≤ 2 then                                          COSTO: 1
        return 1                                           COSTO: 1
    else
        return FIBONACCI2(n-1) + FIBONACCI2(n-2)           COSTO: F(n-1)+F(n-2)
```

- Se $n \leq 2$ il costo totale è 2 (costante)
- Se $n = 3$ il costo totale è 4:
  - 2 per la chiamata FIBONACCI2(3)
  - 1 per la chiamata FIBONACCI2(2)
  - 1 per la chiamata FIBONACCI2(1)

n-esimo numero di Fibonacci: secondo approccio

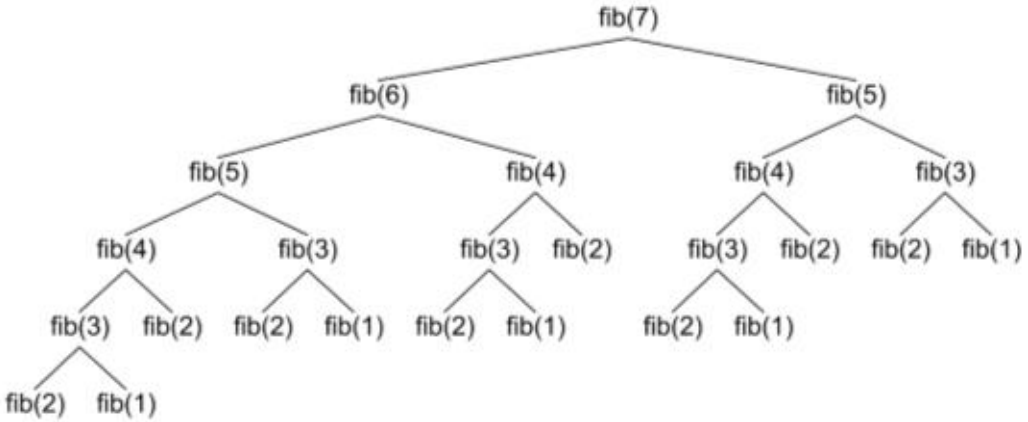
- Analizziamo il costo secondo il modello RAM:

```
FIBONACCI2(n):
    if n ≤ 2 then                                          COSTO: 1
        return 1                                           COSTO: 1
    else
        return FIBONACCI2(n-1) + FIBONACCI2(n-2)           COSTO: F(n-1)+F(n-2)
```

- Se $n \geq 3$ il costo totale aumenta di 2 per ciascuna invocazione:
- Il costo totale dipende quindi dal valore n passato in input:

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

n-esimo numero di Fibonacci: secondo approccio

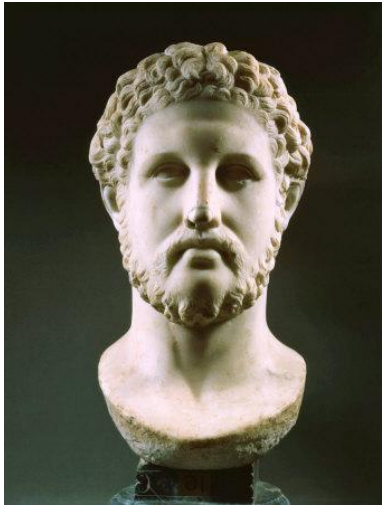


- Lo stesso valore viene ricalcolato più volte!
  - $T(n) \approx F(n) \approx \varphi^n$
- Vedremo in seguito come si può fare di meglio…

Divide et Impera

Divide et Impera

- “Dividi e Comanda”
- Un problema viene suddiviso in un numero di sottoproblemi di taglia inferiore
- Tecnica di progettazione algoritmica organizzata in tre fasi differenti
  - *Divide*: il problema viene suddiviso in un numero di sottoproblemi di taglia inferiore
  - *Impera*: i sottoproblemi vengono risolti ricorsivamente o direttamente se di dimensione piccola a sufficienza
  - *Combina*: le soluzioni dei sottoproblemi sono combinate per ottenere la soluzione al problema originale

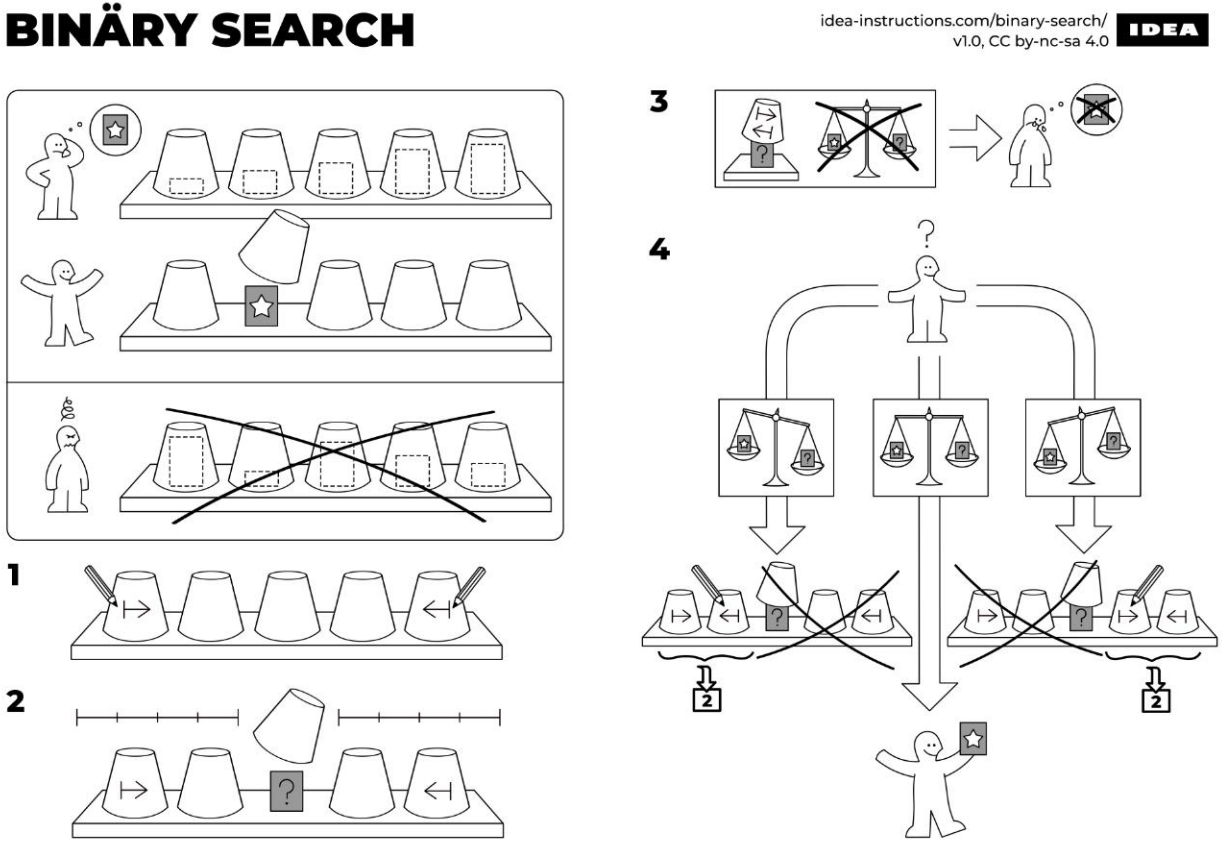


Filippo il Macedone:
«διαίρει καὶ βασίλευε»,
«dividi e regna»

Ricerca binaria

- La ricerca binaria (o dicotomica) è un algoritmo per individuare l'indice di un certo valore (input) in un insieme ordinato di dati
- Tipicamente utilizzato come metodo di ricerca in array ordinati
- Si basa sulla tecnica del *divide et impera*
- Può essere implementato sia in maniera iterativa che ricorsiva

Ricerca binaria



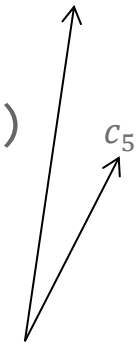
Ricerca binaria ricorsiva

```
BinarySearch(A[0..N-1], low, high, value):
    if high < low then
        return ⊥
    mid = (low + high) / 2
    if A[mid] > value then
        return BinarySearch(A, low, mid-1, value)
    else if A[mid] < value then
        return BinarySearch(A, mid+1, high, value)
    else
        return mid
```

Ricerca binaria ricorsiva

```
BINARYSEARCH(A[0..N-1], low, high, value):
    if high < low then
        return ⊥
    mid = (low + high) / 2
    if A[mid] > value then
        return BINARYSEARCH(A, low, mid-1, value)
    else if A[mid] < value then
        return BINARYSEARCH(A, mid+1, high, value)
    else
        return mid
```

Costo	H < L	L ≤ H
c_1	1	1
c_2	1	0
c_3	0	1
c_4	0	1
$c_5 + T(\lfloor (n-1)/2 \rfloor)$	0	1/0
c_6	0	1
$c_5 + T(\lfloor n/2 \rfloor)$	0	0/1
c_7	0	0



vettore suddiviso in due parti

Ricerca binaria ricorsiva

- Assunzioni per il calcolo del costo nel caso pessimo:
  - Per semplicità assumiamo n potenza di due: $n = 2^k$
  - L'elemento cercato non è presente
  - Ad ogni passo, scegliamo sempre la sottoparte destra di dimensione $n/2$
- Due casi:
  - high < low (n=0): $T(n) = c_1 + c_2 = c$
  - low ≤ high (n>0): $T(n) = T(n/2) + c_1 + c_3 + c_4 + c_5 + c_6 = T(n/2) + d$

- Si ottiene quindi la seguente relazione di ricorrenza:

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

Tempo di calcolo della ricerca binaria

- La relazione di ricorrenza può essere risolta mediante *espansione*:
  - $T(n) = T(n/2) + d =$
    $= T(n/4) + 2d =$
    $= T(n/8) + 3d =$
    $\dots$
    $= T(1) + kd =$
    $= T(0) + (k+1)d =$
    $= kd + c + d$
  - L'assunzione iniziale poneva $n = 2^k$, da cui $k = \log n$
  - $T(n) = d\log n + e$

Ricerca binaria iterativa

```
BinarySearch(A[0..N-1], value):
    low = 0
    high = N - 1
    while low <= 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 ⊥
```

Relazioni di Ricorrenza

Una definizione

- Una relazione di ricorrenza per una sequenza $\{a_n\}$ è un'equazione che esprime a_n in termini di uno o più degli elementi precedenti nella sequenza, ovverosia $a_0, a_1, \ldots, a_{n-1}$ per tutti gli interi $n \geq n_0$, dove n_0 è un intero non negativo
- Una *sequenza* è detta *soluzione di un'equazione di ricorrenza* se i suoi termini soddisfano l'equazione di ricorrenza

- Un esempio: data la relazione di ricorrenza $a_n = 2a_{n-1} - a_{n-2}$:
  - 3, 5, 7, 9, 11, ... è una sequenza che soddisfa la relazione
  - 2, 3, 4, 5, 6, ... è un'altra sequenza che soddisfa la relazione

Relazione di ricorrenza lineari

- Una relazione di ricorrenza lineare di grado k è espressa nella forma:
  - $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} + F(n)$
  - c_i è una costante, per $i = 1, 2, \ldots, k$
  - $c_k \neq 0$

- Le relazioni di ricorrenza lineari si dividono in due sottoclassi:
  - Omogenee: $F(n) = 0$
  - Non omogenee: $F(n) \neq 0$

Alcuni esempi

- Lineari omogenee
  - $a_n = 1.2\ a_{n-1}$ (grado 1)
  - $f_n = f_{n-1} + f_{n-2}$ (grado 2)
  - $a_n = 3a_{n-3}$ (grado 3)
- Non lineari omogenee
  - $a_n = a_{n-1}^2 + a_{n-2}$
  - $a_n = na_{n-1} - 2a_{n-2}$
- Lineari non omogenee
  - $a_n = a_{n-1} + 2^n$
  - $h_n = 2h_{n-1} + 1$
  - $a_n = 3a_{n-1} + n$
- Non lineari non omogenee
  - $a_n = a_{n-1}^2 + 2^n$
  - $a_n = n^2\, a_{n-1} + n$

Come risolvere relazioni lineari omogenee

- La sequenza $\{a_n\}$ è una soluzione della relazione di ricorrenza $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ se e solo se $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ dove:
  - r_1 ed r_2 sono due *radici distinte* di $r^2 - c_1 r - c_2 = 0$
  - $a_0 = \alpha_1 + \alpha_2$
  - $a_1 = \alpha_1 r_1 + \alpha_2 r_2$
  - $\alpha_1 = \frac{a_1 - a_0 r_2}{r_1 - r_2}$ e $\alpha_2 = \frac{a_0 r_1 - a_1}{r_1 - r_2}$

Un esempio

- Qual è la soluzione di $a_n = a_{n-1} + 2a_{n-2}$ con $a_0 = 2$ e $a_1 = 7$?

- Abbiamo $c_1 = 1$ e $c_2 = 2$
- L'equazione caratteristica è $r^2 - r - 2 = 0$ le cui radici sono $r_1 = 2$ ed $r_2 = -1$
- La soluzione finale è quindi:
  - $\alpha_1 = \frac{a_1 - a_0 r_2}{r_1 - r_2} = \frac{7+2}{2+1} = 3$
  - $\alpha_2 = \frac{a_0 r_1 - a_1}{r_1 - r_2} = \frac{4-7}{2+1} = -1$
  - $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n = 3 \cdot 2^n - (-1)^n$

Un esempio: i numeri di Fibonacci

- Qual è la soluzione di $a_n = a_{n-1} + a_{n-2}$ con $a_0 = 0$ e $a_1 = 1$?

- Abbiamo $c_1 = 1$ e $c_2 = 1$
- L'equazione caratteristica è $r^2 - r - 1 = 0$ le cui radici sono $r_1 = \frac{1+\sqrt{5}}{2}$ ed $r_2 = \frac{1-\sqrt{5}}{2}$
- La soluzione finale è quindi:
  - $\alpha_1 = \frac{a_1 - a_0 r_2}{r_1 - r_2} = \frac{1-0}{\sqrt{5}} = \frac{1}{\sqrt{5}}$
  - $\alpha_2 = \frac{a_0 r_1 - a_1}{r_1 - r_2} = \frac{0-1}{\sqrt{5}} = -\frac{1}{\sqrt{5}}$
  - $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} - \frac{\left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\varphi^n}{\sqrt{5}} - \frac{(1-\varphi)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$

Come risolvere relazioni lineari omogenee (2)

- La sequenza $\{a_n\}$ è una soluzione della relazione di ricorrenza $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ se e solo se $a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$ dove:
  - r_0 è *l'unica radice* di $r^2 - c_1 r - c_2 = 0$
  - $a_0 = \alpha_1$
  - $a_1 = \alpha_1 r_0 + \alpha_2 r_0 = r_0(\alpha_1 + \alpha_2)$

  - $\alpha_1 = a_0$ e $\alpha_2 = \frac{a_1}{r_0} - a_0$

Un esempio

- Qual è la soluzione di $a_n = 6a_{n-1} - 9a_{n-2}$ con $a_0 = 1$ e $a_1 = 6$?

- Abbiamo $c_1 = 6$ e $c_2 = -9$
- L'equazione caratteristica è $r^2 - 6r + 9 = 0$ le cui radici $r_0 = 3$
- La soluzione finale è quindi:
  - $\alpha_1 = a_0 = 1$
  - $\alpha_2 = \frac{a_1}{r_0} - a_0 = \frac{6}{3} - 1 = 1$
  - $a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n = 3^n + n3^n = 3^n(1+n)$

Come risolvere relazioni lineari omogenee (3)

- La sequenza $\{a_n\}$ è una soluzione della relazione di ricorrenza $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}$ se e solo se $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \alpha_k r_k^n$ dove:
  - $r_1, r_2, \ldots, r_k$ sono *radici distinte* di $r^k - c_1 r^{k-1} - c_2 r^{k-2} \ldots - c_k = 0$
  - $a_0 = \alpha_1 + \alpha_2 + \ldots + \alpha_k$
  - $a_1 = \alpha_1 r_1 + \alpha_2 r_2 + \ldots + \alpha_k r_k$
  - $a_2 = \alpha_1 r_1^2 + \alpha_2 r_2^2 + \ldots + \alpha_k r_k^2$
  - …

Un esempio

- Qual è la soluzione di $a_n = 6a_{n-1} - 11a_{n-2} + 6a_{n-3}$ con $a_0 = 2$, $a_1 = 5$ e $a_2 = 15$?

- Abbiamo $c_1 = 6, c_2 = -11$ e $c_2 = 6$
- L'equazione caratteristica è $r^3 - 6r^2 + 11r - 6 = 0$ le cui radici sono $r_1 = 1, r_2 = 2$ ed $r_3 = 3$
- La soluzione finale è quindi:
  - $a_0 = \alpha_1 + \alpha_2 + \alpha_3$
  - $a_1 = \alpha_1 r_1 + \alpha_2 r_2 + \alpha_3 r_3 = \alpha_1 + 2\alpha_2 + 3\alpha_3$
  - $a_2 = \alpha_1 r_0^2 + \alpha_2 r_2^2 + \alpha_3 r_3^2 = \alpha_1 + 4\alpha_2 + 9\alpha_3$
  - $\alpha_1 = 1, \alpha_2 = -1, \alpha_3 = 2$
  - $a_n = 1 - 2^n + 2 \cdot 3^n$

Come risolvere relazioni lineari non omogenee

- Se la sequenza $\{a_n^{(p)}\}$ è una soluzione particolare della relazione di ricorrenza $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} + F(n)$, allora tutte le soluzioni sono nella forma $\{a_n^{(p)} + a_n^{(h)}\}$, dove $\{a_n^{(h)}\}$ è una soluzione della relazione di ricorrenza omogenea associata $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}$

Un esempio

- Qual è la soluzione di $a_n = 3a_{n-1} + 2n$ con $a_1 = 3$?

- Abbiamo $c_1 = 3$
- L'equazione caratteristica associata è $r - 3 = 0$, la cui radice è $r = 3$.
- Allora, $a_n^{(h)} = \alpha 3^n$
- Sia $p_n = cn + d$
- Allora, da $a_n = 3a_{n-1} + 2n$, $cn + d = 3(c(n-1) + d) + 2n$
- Pertanto, $c = -1$, $d = -\frac{3}{2}$ e $a_n^{(p)} = -n - \frac{3}{2}$
- $a_n = a_n^{(p)} + a_n^{(h)} = \alpha \cdot 3^n - n - \frac{3}{2}$
- Da $a_1 = 3 = 3\alpha - 3 - \frac{3}{2}$, $\alpha = \frac{11}{6}$
- La soluzione è $a_n = \frac{11}{6} \cdot 3^n - n - \frac{3}{2}$

Un esempio: la somma dei primi n numeri

- Qual è la soluzione di $a_n = a_{n-1} + n$ con $a_1 = 1$?

- Abbiamo $c_1 = 1$
- L'equazione caratteristica associata è $r - 1 = 0$, la cui radice è $r = 1$.
- Allora, $a_n^{(h)} = \alpha 1^n = \alpha$
- Sia $p_n = n(cn + d) = cn^2 + dn$
- Allora, da $a_n = a_{n-1} + n$, $cn^2 + dn = c(n-1)^2 + d(n-1) + n$
- Pertanto $cn^2 + dn = cn^2 - 2cn + c + dn - d + n$, e $c - d + n(1 - 2c) = 0$
- $c - d = 0$ e $1 - 2c = 0$. Quindi, $c = d = \frac{1}{2}$
- $a_n = a_n^{(p)} + a_n^{(h)} = \alpha + \frac{n(n+1)}{2}$
- Da $a_1 = 0 = \alpha + 1, \alpha = 0$
- La soluzione è $a_n = \frac{n(n+1)}{2}$

Analisi Asintotica

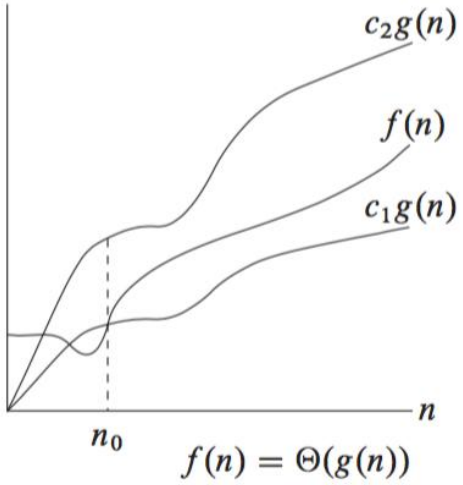
Motivazioni

- È interessante studiare gli algoritmi **al variare della dimensione dell'input**, *a meno di costanti*
  - Di fatto, il modello RAM “nasconde” già alcune costanti
  - Occorre comunque tenere a mente che tali costanti possono essere arbitrariamente grandi e possono avere un impatto di efficienza su determinate architetture hardware/software
- L'**analisi asintotica** trascura i fattori costanti
- Questa viene influenzata dall'**operazione dominante**
  - Intuitivamente, si tratta dell'operazione eseguita “più frequentemente”

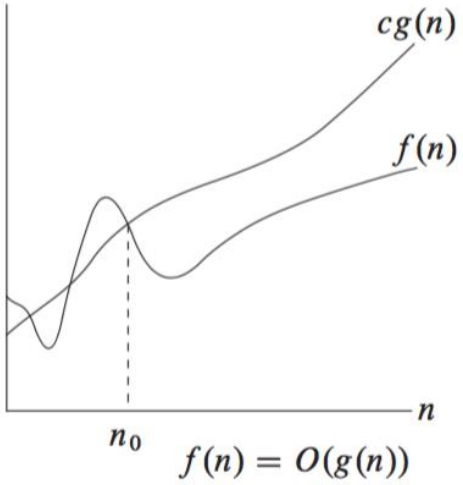
Definizioni fondamentali

- Siano $f(n)$: $\mathbb{N} \mapsto \mathbb{R}$ e $g(n)$: $\mathbb{N} \mapsto \mathbb{R}$ due funzioni *non negative* e *non decrescenti*
- $f(n) = O(g(n))$ se esistono $c > 0$ reale e $n_0 \geq 1$ intero tali che:
  - $f(n) \leq c \cdot g(n)$, per $n > n_0$
  - f cresce **al più** come g
- $f(n) = \Omega(g(n))$ se esistono $c > 0$ reale e $n_0 \geq 1$ intero tali che:
  - $f(n) \geq c \cdot g(n)$, per $n > n_0$
  - f cresce **almeno** come g
- $f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$:
  - f cresce **come** g

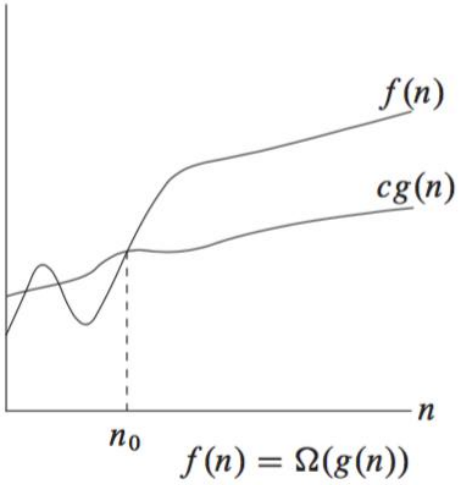
Una visualizzazione grafica



$f(n) = \Theta(g(n))$



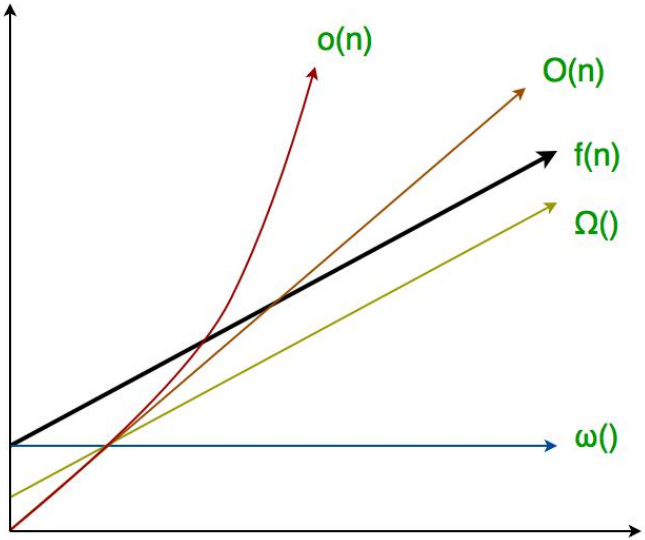
$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Definizioni fondamentali

- $f(n) = o(g(n))$ se $\lim_{n\to\infty} f(n)/g(n) = 0$
  - si tratta di un limite superiore *non stretto*
- $f(n) = \omega(g(n))$ se $g(n) = o(f(n))$
  - Significa che $\lim_{n\to\infty} f(n)/g(n) = \infty$
  - si tratta di un limite inferiore *non stretto*



Confronto tra notazioni grandi e piccole

- $f(n) = O(g(n))$:
  - i limiti $0 \leq f(n) \leq c \cdot g(n)$ sono verificati per *qualche* costante $c > 0$
- $f(n) = o(g(n))$:
  - i limiti $0 \leq f(n) \leq c \cdot g(n)$ sono verificati per *tutte* le costanti $c > 0$
- Un'analogia:
  - $f(n) = O(g(n))$ *ricorda* $a \leq b$
  - $f(n) = \Omega(g(n))$ *ricorda* $a \geq b$
  - $f(n) = \Theta(g(n))$ *ricorda* $a = b$
  - $f(n) = o(g(n))$ *ricorda* $a < b$
  - $f(n) = \omega(g(n))$ *ricorda* $a > b$

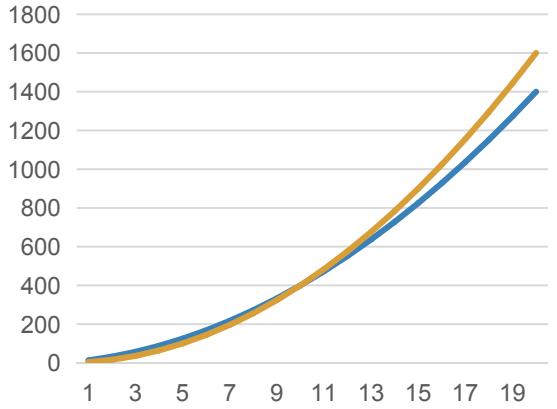
Alcuni esempi

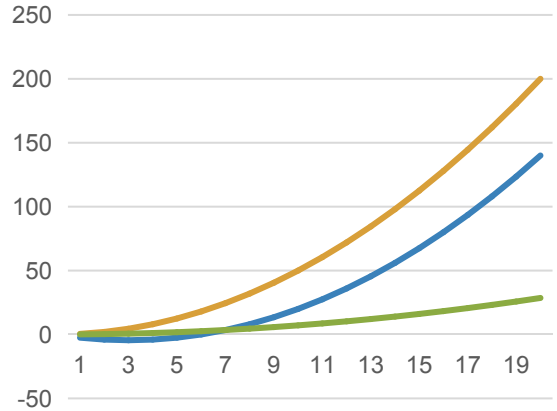
- L'algoritmo ARRAYMAX ha costo $\Theta(n)$

- $f(n) = 3n^2 + 10n = O(n^2)$
  - per $c = 4$ e $n_0 \geq 10 \Rightarrow 3n^2 + 10n \leq 4n^2$

- $f(n) = n^2/2 - 3n = \Theta(n^2)$
  - per $c_1 = 1/14$, $c_2 = 1/2$, $n_0 \geq 7 \Rightarrow$
    $\Rightarrow c_1 n^2 \leq n^2/2 - 3n \leq c_2 n^2$

- Domanda: che cosa significa **O(1)**?





Operazione dominante

- Un'operazione di un algoritmo di costo $f(n)$ si dice **dominante** se, per ogni n, essa viene eseguita, *nel caso pessimo di input di dimensione n*, un numero di volte $d(n)$ che soddisfa:

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

per opportune costanti reali a e b.

- Un esempio: l'istruzione **if** $A[i] > currentMax$ **then** nell'algoritmo ARRAYMAX

Limiti dell'analisi asintotica del caso pessimo

- Le *costanti nascoste* possono essere **molto grandi**
  - Un algoritmo con costo $10^{50}n$ è $O(n)$ (*lineare*), ma potrebbe essere poco pratico
- Cosa succede quando n è molto piccolo?
  - Ad esempio, $3n$ contro n^2
- Il caso pessimo potrebbe essere **molto raro**
  - Un'analisi nel caso medio potrebbe fornire risultati sulla complessità molto differente

Insertion Sort

- Ordina in modo non decrescente
- Inserisce l'elemento A[i] nella posizione corretta nel vettore ordinato A[0, ..., i-1]

```
INSERTIONSORT(A, n):
    for i ← 1 to n:
            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
```

Insertion Sort

- Ordina in modo non decrescente
- Inserisce l'elemento A[i] nella posizione corretta nel vettore ordinato A[0, ..., i-1]

```
INSERTIONSORT(A, n):
    for i ← 1 to n:
            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
```

6 5 3 1 8 7 2 4

Analisi dell'Insertion Sort

INSERTIONSORT(*A*, *n*):	*COSTO*	*RIPETIZIONI*
for $i \leftarrow 1$ **to** n:	c_1	n
$key \leftarrow A[i]$	c_2	$n - 1$
$j \leftarrow i - 1$	c_3	$n - 1$
while $j \geq 1$ and $A[j] > key$:	c_4	$\sum_{i=2}^{n} t_i$
$A[j + 1] \leftarrow A[j]$	c_5	$\sum_{i=2}^{n} (t_i - 1)$
$j \leftarrow j - 1$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$A[j + 1] \leftarrow key$	c_7	$n - 1$

- Costo:
  - $c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$
- L'operazione dominante è una qualunque di quelle eseguite nel ciclo più interno

Analisi dell'Insertion Sort

- Il costo di esecuzione non dipende solo dalla dimensione, ma anche dalla distribuzione dei dati in ingresso
- Qual è il costo nel caso in cui il vettore sia già ordinato?
- Qual è il costo nel caso in cui il vettore sia ordinato al contrario?
- Qual è il caso medio?

Analisi dell'Insertion Sort

- Il costo di esecuzione non dipende solo dalla dimensione, ma anche dalla distribuzione dei dati in ingresso

- Qual è il costo nel caso in cui il vettore sia già ordinato?
  - In ogni iterazione il primo elemento della sottosequenza non ordinata viene confrontato solo con l'ultimo della sottosequenza ordinata: $\Theta(n)$
- Qual è il costo nel caso in cui il vettore sia ordinato al contrario?
  - ogni iterazione dovrà scorrere e spostare ogni elemento della sottosequenza ordinata prima di poter inserire il primo elemento della sottosequenza non ordinata: $O(n^2)$
- Qual è il caso medio?
  - Anche in questo caso è quadratico, il che lo rende impraticabile per input grandi

Insertion sort nel caso medio

- L'insertion sort è basato sul confronto: i valori passati in input non contano molto per l'analisi, solo il loro ordinamento relativo
- Assunzione: tutti gli elementi nel vettore sono differenti
  - Ma non cambierebbe poi molto nell'analisi
- Sia X_{ij} una variabile aleatoria che vale 1 se A[i] deve essere scambiato con A[j]
  - Abbiamo bisogno di $\frac{n(n-1)}{2}$ variabili di questo tipo per caratterizzare il generico vettore
- Sia $I = \sum X_{ij}$ una variabile aleatoria che indica il numero di scambi necessari a calcolare l'output
- $E[I] = \mathrm{E}[\sum X_{ij}] = \sum E[X_{ij}]$

Insertion sort nel caso medio

- $E[I] = \mathrm{E}[\sum X_{ij}] = \sum E[X_{ij}]$
  - Se siamo in grado di calcolare il numero atteso di scambi e quindi il costo d'esecuzione
- $E[X_{ij}] = Pr[X_{ij} = 1] = Pr[A[i] \text{ ed } A[j] \text{ sono scambiati}]$
- Statisticamente, A[i] ed A[j] sono invertiti la metà delle volte
  - Stiamo assumendo che tutti gli elementi siano differenti
- $E[I] = \sum E[X_{ij}] = \sum \frac{1}{2}$
- Poiché ci sono $\frac{n(n-1)}{2}$ elementi nella somma, $E[I] = \frac{n(n-1)}{4} = \Theta(n^2)$

Classi di complessità

f(n)	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Classe
lg*n*	3	6	9	13	logaritmica
√*n*	3	10	31	100	sublineare
n	10	100	1000	10000	lineare
n lg*n*	30	664	9965	132877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratica
n^3	10^3	10^6	10^9	10^{12}	cubica
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale
n!	3628800	$\sim 9 \cdot 10^{157}$	$\sim 4 \cdot 10^{2567}$	$\sim 3 \cdot 10^{35659}$	fattoriale

Analisi delle Ricorrenze

Relazioni di ricorrenza e classi computazionali

- Molto spesso gli algoritmi ricorsivi danno luogo a relazioni di ricorrenza
- Il tempo necessario all'esecuzione di una chiamata ricorsiva è pari al tempo speso all’interno della chiamata più il tempo speso nelle successive chiamate ricorsive
- Nell'ambito dell'analisi asintotica, vogliamo capire a quale classe computazionale appartiene un algoritmo di questo tipo

Alcuni esempi

- Le torri di Hanoi:
  - Quante mosse sono necessarie per risolvere il gioco delle torri di Hanoi con *n* dischi?
  - $H(1) = 1$
  - $H(n) = 2H(n-1) + 1$
  - $H(n) = 2(2H(n-2) +1) +1$

    $= 4H(n-2) + 2 + 1$

    $= 4(2(H(n-3) + 1) + 2 + 1$

    $= 8H(n-3) + 4 + 2 + 1$

    $= 8(2H(n-4) +1) + 4 + 2 + 1$

    $= 16H(n-4) + 8 + 4 + 2 + 1$

    $= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 1$

    $= 2^n - 1$

Alcuni esempi

- Ricerca binaria

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

- Mediante espansione eravamo già giunti alla complessità:

$$T(n) = d\log n + e$$

- Volendo rappresentare in forma chiusa la classe di complessità della funzione otteniamo:

$$T(n) = \Theta(\log n)$$

Tecniche per derivare le classi di complessità

- Analisi dei livelli
  - “srotoliamo” la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione

- Analisi per tentativi o per sostituzione
  - cerchiamo di “indovinare” una soluzione e dimostriamo induttivamente che questa soluzione è vera

- Ricorrenze comuni
  - vi è una classe di ricorrenze che possono essere risolte facendo riferimento ad alcuni teoremi specifici

Analisi dei livelli: primo esempio

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

- Possiamo "espandere" la relazione di ricorrenza, assumendo per semplicità $n = 2^k$ (ovvero $k = \log n$):
  - $T(n) = d + T(n/2)$
  - $= d + d + T(n/4)$
  - $= d + d + d + T(n/8)$
  - …
  - $= d + d + \ldots + d + T(1) =$
  - $= d\log n + e$
- Quindi, $T(n) = \Theta(\log n)$

Analisi dei livelli: secondo esempio

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

- Possiamo "espandere" la relazione di ricorrenza:
  - $T(n) = n + 4T(n/2)$
  - $= n + 4n/2 + 16T(n/2^2)$
  - $= n + 2n + 16n/4 + 64T(n/8)$
  - …
  - $= n + 2n + 4n + 8n + \ldots + 2^{\log n - 1} n + 4^{\log n} T(1) =$
  - $= n \sum_{j=0}^{\log n - 1} 2^j + 4^{\log n}$

Analisi dei livelli: secondo esempio

- Concentriamoci sulla prima parte del costo:

$$n \sum_{j=0}^{\log n - 1} 2^j$$

- Notiamo la presenza di una serie geometrica finita:

$$\forall x \neq 1: \sum_{j=0}^{k} x^k = \frac{x^{k+1} - 1}{x - 1}$$

- Quindi:

$$n \sum_{j=0}^{\log n - 1} 2^j = n \frac{2^{\log n} - 1}{2 - 1} = n(n-1)$$

Analisi dei livelli: secondo esempio

- Introduciamo nuovamente l'ultima parte del costo:

$$T(n) = n(n-1) + 4^{\log n}$$

- Utilizziamo la regola del cambiamento di base dei logaritmi:

$$\log_b n = (\log_b a) \cdot (\log_a n) \Longrightarrow a^{\log_b n} = n^{\log_b a}$$

- Quindi:

$$4^{\log n} = n^{\log 4} = n^2$$

- Da cui:

$$T(n) = n(n-1) + n^2 = 2n^2 - n = \Theta(n^2)$$

Metodo di sostituzione

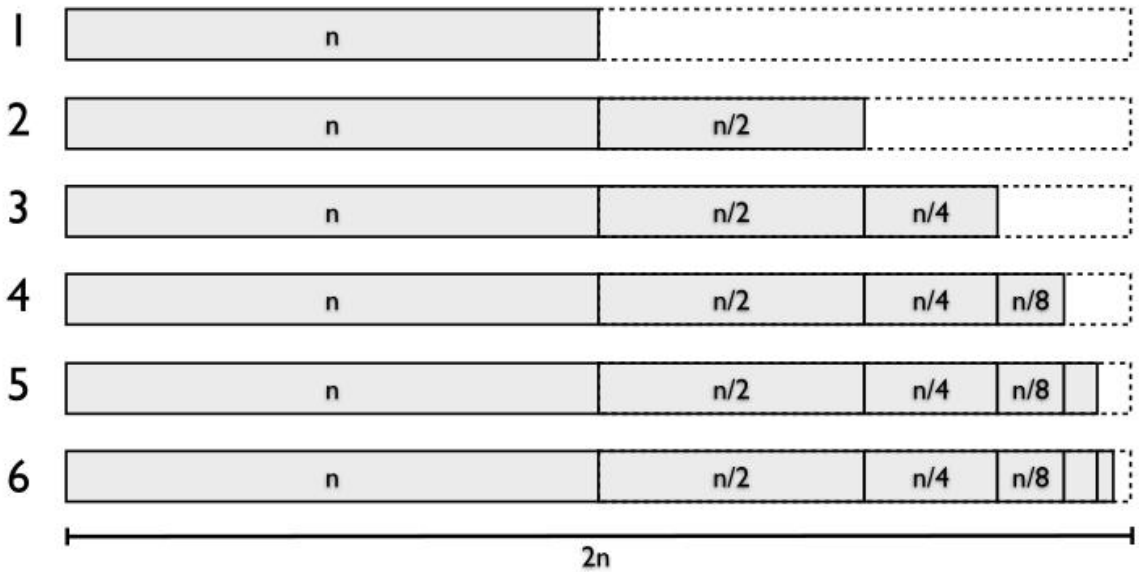
- Cerchiamo di “indovinare” la classe di complessità per:

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

Metodo di sostituzione

- Cerchiamo di "indovinare" la classe di complessità per:

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



Metodo di sostituzione

- Cerchiamo di indovinare la classe di complessità per:

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

$$T(n) = n \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i \leq n \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i \leq n \frac{1}{1 - \frac{1}{2}} = 2n$$

- Sfruttiamo la proprietà della serie geometrica descrescente infinita:

$$\forall x, |x| < 1: \sum_{i=0}^{+\infty} x^i = \frac{1}{1 - x}$$

Metodo di sostituzione: limite superiore

- Facciamo l'ipotesi che la classe sia $T(n) = O(n)$
- Dalla definizione:
  - $\exists c > 0, \exists n_0 \geq 1 : T(n) \leq cn, \forall n \geq n_0$
- Proviamo a dimostrare il caso base:
  - $T(1) = 1 \leq 1 \cdot c \Leftrightarrow \forall c \geq 1$

Metodo di sostituzione: limite superiore

- Dimostriamo induttivamente il caso generale:
  - Ipotesi: $\forall k < n: T(k) \leq ck$
- $T(n) = T(\lfloor n/2 \rfloor) + n$

  $\leq c\lfloor n/2 \rfloor + n$

  $\leq cn/2 + n$

  $= (c/2 + 1)n$

  $\leq cn \Leftrightarrow c/2 + 1 \leq c \Leftrightarrow c \geq 2$
- Quindi, $T(n) \leq cn$ per $c \geq 1$ (caso base) e $c \geq 2$ (passo induttivo)
- Possiamo prendere $c = 2$
- Questo vale per $n \geq 1$, quindi possiamo prendere $n_0 = 1$
- Abbiamo dimostrato che $T(n) = O(n)$

Metodo di sostituzione: limite inferiore

- Facciamo l'ipotesi che la classe sia $T(n) = \Omega(n)$
- Dalla definizione:
  - $\exists c > 0, \exists n_0 \geq 1 : T(n) \geq cn, \forall n \geq n_0$
- Proviamo a dimostrare il caso base:
  - $T(1) = 1 \geq 1 \cdot c \Leftrightarrow \forall c \leq 1$

Metodo di sostituzione: limite inferiore

- Dimostriamo induttivamente il caso generale:
  - Ipotesi: $\forall k < n: T(k) \geq ck$
- $T(n) = T(\lfloor n/2 \rfloor) + n$

  $\geq c\lfloor n/2 \rfloor + n$

  $\geq cn/2 - 1 + n$

  $= \left(\frac{c}{2} - \frac{1}{n} + 1\right) n$

  $\geq cn \Leftrightarrow \frac{c}{2} - \frac{1}{n} + 1 \geq c \Leftrightarrow c \leq 2 - 2/n$
- Quindi, $T(n) \geq cn$ per $c \leq 1$ (caso base) e $c \leq 2 - 2/n$ (passo induttivo)
- Possiamo prendere $c = 1$
- Questo vale per $n \geq 1$, quindi possiamo prendere $n_0 = 1$
- Abbiamo dimostrato che $T(n) = \Omega(n)$

Metodo di sostituzione: analisi finale

- Abbiamo dimostrato che $T(n) = O(n)$
- Abbiamo dimostrato che $T(n) = \Omega(n)$

- Per definizione, se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$, allora $f(n) = \Theta(n)$

- In questo caso $T(n) = O(n)$ e $T(n) = \Omega(n)$, allora:

$$T(n) = \Theta(n)$$

Riccorrenze comuni: il Teorema Principale

- Il *master theorem* permette di determinare la classe di complessità di alcune famiglie di relazioni di ricorrenza nella forma:

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$$

- dove $a \geq 1$ e $b \geq 2$ sono costanti intere, $c > 0$ e $\beta \geq 0$ sono costanti reali.
- Posto: $\alpha = \log_b a$, abbiamo:

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

- Intuizione: è una "gara" tra cn^{β} e n^{α}

Alcuni esempi

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$$

- $T(n) = 9T\left(\frac{n}{3}\right) + n$
- $a = 9, b = 3, c = 1, \beta = 1$
- $\alpha = \log_3 9 = 2$
- $\alpha > \beta \Longrightarrow T(n) = \Theta(n^{\alpha}) = \Theta(n^2)$

Alcuni esempi

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$$

- $T(n) = T\left(\frac{n}{2}\right) + 1$ [ricerca binaria]
- $a = 1, b = 2, c = 1, \beta = 0$
- $\alpha = \log_2 1 = 0$
- $\alpha = \beta \Longrightarrow T(n) = \Theta(n^{\alpha} \log n) = \Theta(\log n)$

Ricorrenze lineari di ordine costante

- Data una relazione di ricorrenza nella forma:

$$T(n) = \begin{cases} \sum_{1 \le i \le h} a_i T(n-i) + cn^{\beta} & n > m \\ \Theta(1) & n \le m \le h \end{cases}$$

- dove $a_1, a_2, \ldots, a_h$ sono costanti intere non negative, con h costante positiva, $c > 0$ e $\beta \ge 0$ sono costanti reali.
- Posto $a = \sum_{1 \le i \le h} a_i$, abbiamo:

$$T(n) = \begin{cases} \Theta(n^{\beta+1}) & \text{se } a = 1 \\ \Theta(a^n n^{\beta}) & \text{se } a \ge 2 \end{cases}$$

Alcuni esempi

$$T(n) = \begin{cases} \sum_{1 \le i \le h} a_i T(n-i) + cn^{\beta} & n > m \\ \Theta(1) & n \le m \le h \end{cases}$$

- $T(n) = T(n-10) + n^2$
- $a = 1, c = 1, \beta = 2$
- $a = 1 \Longrightarrow T(n) = \Theta(n^{\beta+1}) = \Theta(n^3)$

Alcuni esempi

$$T(n) = \begin{cases} \sum_{1 \le i \le h} a_i T(n-i) + cn^{\beta} & n > m \\ \Theta(1) & n \le m \le h \end{cases}$$

- $T(n) = T(n-2) + T(n-1) + 1$ [Fibonacci]
- $a = 2, c = 1, \beta = 0$
- $a \geq 2 \Longrightarrow T(n) = \Theta(a^n n^{\beta}) = \Theta(2^n)$

Esempi Riassuntivi

Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

- Abbiamo visto quattro soluzioni algoritmiche differenti a questo problema, proviamo a studiarne di nuovo la complessità

Soluzione naïf al problema

```
def maxsum1(A):
    maxSoFar = 0; # Maximum found so far
    for i in range(0, len(A)):
        for j in range(i, len(A)):
            maxSoFar = max(maxSoFar, sum(A[i:j+1]))
    return maxSoFar;
```

- Utilizzando il concetto di operazione dominante (la riga più interna che calcola l'accumulatore) possiamo scrivere:

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

Soluzione naïf al problema

- Facciamo l'ipotesi che la classe sia $T(n) = O(n^3)$
- Dalla definizione:
  - $\exists c_1 > 0, \exists n_0 \geq 1 : T(n) \leq c_1 n^3, \forall n \geq n_0$
- $T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$

  $\leq \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} n \leq \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$

  $= \sum_{i=0}^{n-1} n^2 = n^3 \leq c_1 n^3$
- Questa dimostrazione è vera per $n \geq n_0 = 0$ e per $c_1 \geq 1$

Soluzione naïf al problema

- Facciamo l'ipotesi che la classe sia $T(n) = \Omega(n^3)$
- Dalla definizione:
  - $\exists c_2 > 0, \exists n_0 \geq 1 : T(n) \geq c_2 n^3, \forall n \geq n_0$
- $T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$

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

  $= \sum_{i=0}^{n/2} n^2/4 = n^3/8 \geq c_2 n^3$
- Questa dimostrazione è vera per $n \geq n_0 = 0$ e per $c_2 \leq \frac{1}{8}$

Prima ottimizzazione

```
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
```

- Utilizzando il concetto di operazione dominante (la riga più interna che calcola l'accumulatore) possiamo scrivere:

$$T(n) = \sum_{i=0}^{n-1} n - i$$

Prima ottimizzazione

- Vogliamo dimostrare che $T(n) = \Theta(n^2)$
- $T(n) = \sum_{i=0}^{n-1} n - i$
- $= \sum_{i=1}^{n} i$
- $= \frac{n(n+1)}{2} = \Theta(n^2)$

Seconda ottimizzazione

- Divide et impera:
  - dividiamo il vettore nelle metà di destra e sinistra, in due parti più o meno uguali
  - `maxL` è la somma massimale nella parte sinistra
  - `maxR` è la somma massimale nella parte destra
  - `maxLL+maxRR` è il valore della sottolista massimale "al centro"
  - Restituisce il massimo dei tre valori

Seconda ottimizzazione

- Trattandosi di un algoritmo ricorsivo, possiamo scrivere la relazione di ricorrenza:
- $T(n) = 2T\left(\frac{n}{2}\right) + n$

- Applichiamo il Teorema Principale
- $a = 2, b = 2, c = 1, \beta = 1$
- $\alpha = \log_2 2 = 1$
- $\alpha = \beta \Longrightarrow T(n) = \Theta(n^{\alpha} \log n) = \Theta(n \log n)$

Terza ottimizzazione

```
def maxsum4(A):
  maxSoFar = 0 # Maximum found so far
  maxHere = 0 # Maximum slice ending at the current pos
  start = end = 0 # Start, end of the maximal slice found so far
  last = 0 # Beginning of the maximal slice ending here
  for i in range(0, len(A)):
    maxHere = maxHere + A[i]
    if maxHere <= 0:
      maxHere = 0
      last = i+1
    if maxHere > maxSoFar:
      maxSoFar = maxHere
      start, end = last, i
  return (start, end)
```

È facile vedere che in questo caso la complessità è $\Theta(n)$

Analisi Ammortizzata

Analisi ammortizzata

- Si tratta di una tecnica di analisi della complessità che valuta il tempo richiesto per eseguire, *nel caso pessimo*, una sequenza di operazioni su una struttura dati
  - Esistono operazioni più o meno costose
  - Se le operazioni più costose sono poco frequenti, allora il loro costo può essere ammortizzato dalle operazioni meno costose

- **Analisi del caso medio**: probabilistica, su singola operazioni
- **Analisi ammortizzata**: deterministica, su operazioni multiple, nel caso pessimo

Analisi ammortizzata: metodologie

- **Metodo dell'aggregazione** (tecnica derivata dalla matematica):
  - Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel *caso pessimo*

- **Metodo degli accantonamenti** (tecnica derivata dall'economia):
  - Alle operazioni vengono assegnati *costi ammortizzati* che possono essere maggiori/minori del loro costo effettivo

- **Metodo del potenziale** (tecnica derivata dalla fisica):
  - Lo stato del sistema viene descritto con una *funzione di potenziale*

Un esempio: il contatore binario

- Si tratta di 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 bit più significativo è in A[k-1]
  - Il valore del contatore è $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$
  - È prevista un'operazione INCREMENT() che incrementa il contatore di 1

```
INCREMENT(A, k):
    i ← 0
    while i < k and A[i] == 1:
            A[i] ← 0
            i ← i + 1
    if i < k then
            A[i] = 1
```

Metodo dell'aggregazione

- Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel *caso pessimo*

- *Sequenza*: si considera l'evoluzione della struttura dati data una sequenza di operazioni
- *Caso pessimo*: si considera la peggior sequenza di operazioni
- *Aggregazione*: si sommano tutte le complessità individuali

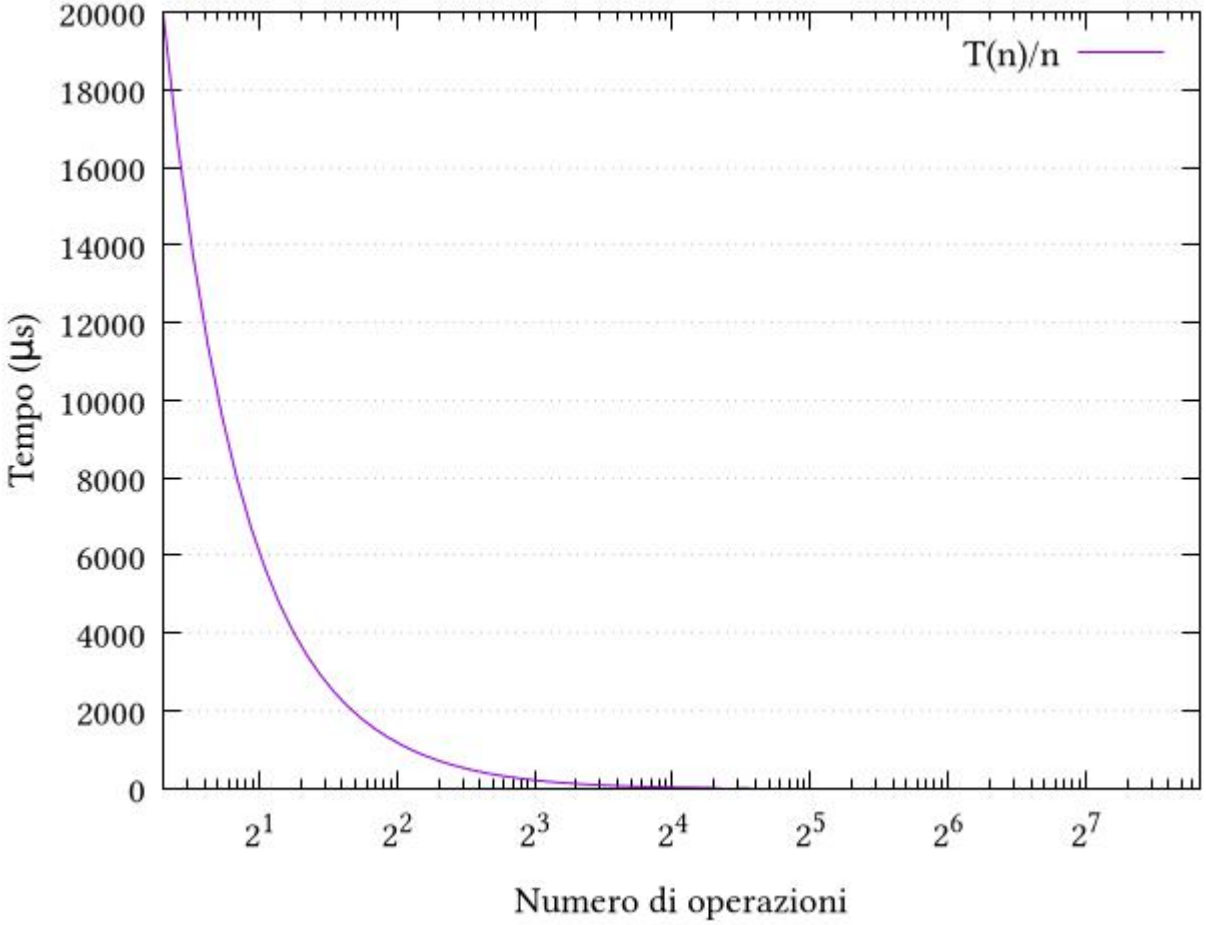
Una prima analisi

- Una chiamata ad INCREMENT() richiede tempo $O(k)$ nel caso pessimo
- È prevista una sola operazione: c'è quindi un'unica sequenza possibile
- Il limite superiore è $T(n) = O(nk)$ per una sequenza di n incrementi

Quanto ci siamo andati vicini?

- Sono necessari $k = \lceil \log n \rceil$ bit per rappresentare n
- Costo di una operazione: $T(n)/n = O(k)$
- Costo di n operazioni: $T(n) = O(nk)$

Quanto ci siamo andati vicini?



Una seconda analisi

- Il tempo necessario ad eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati
- Quanti bit vengono modificati?

Una seconda analisi

i	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	#bit
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5

Una seconda analisi

i	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	#bit
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5
cambi	0	0	0	1	2	4	8	16	

Una seconda analisi

- Il bit in A[0] viene modificato ad ogni incremento
- Il bit in A[1] viene modificato ogni 2 incrementi
- Il bit in A[2] viene modificato ogni 4 incrementi
- …
- Il bit in A[i] viene modificato ogni 2^i incrementi
- Costo totale:

$$T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = 2n$$

- Costo ammortizzato:

$$\frac{T(n)}{n} = \frac{2n}{n} = 2 = O(1)$$

Metodo degli accantonamenti

- Si considerano tutte le operazioni possibili e, ad ognuna, si associa un *costo ammortizzato potenzialmente distinto*
- Il costo ammortizzato può essere diverso dal costo effettivo
  - Le operazioni meno costose vengono caricate di un costo aggiuntivo detto *credito*
    - costo ammortizzato = costo effettivo + credito prodotto
  - I crediti accumulati sono usati per pagare le operazioni più costose
    - costo ammortizzato = costo effettivo – credito consumato

Nel caso del contatore binario

- Costo effettivo dell'operazione INCREMENT(): d
  - d è il numero di bit che cambiano valore
- Costo ammortizzato dell'operazione INCREMENT(): 2
  - 1 per il cambio di un bit da 0 ad 1 (costo effettivo)
  - 1 per il futuro cambio dello stesso bit da 1 a 0
- Quindi:
  - In ogni istante, il credito è pari al numero di bit impostati ad 1 presenti
  - Costo totale: $O(n)$
  - Costo ammortizzato: $O(1)$

Metodo del potenziale

- Si associa ad una certa 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

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

- dove D_i è il potenziale associato alla i-esima operazione

Metodo del potenziale

- Data una sequenza di n operazioni:
  - $A = \sum_{i=1}^{n} a_i$
  - $= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$
  - $= C + \Phi(D_1) - \Phi(D_0) + \Phi(D_2) - \Phi(D_1) + \ldots + \Phi(D_n) - \Phi(D_{n-1})$
  - $= C + \Phi(D_n) - \Phi(D_0)$

- Se la variazione $\Phi(D_n) - \Phi(D_0) > 0$, il costo ammortizzato è un limite superiore al costo reale

Nel 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: $1 + t$
  - Varazione di potenziale: $1 - t$
  - Costo ammortizzato: $1 + t + 1 - t = 2$
  - t è il numero di bit impostati ad 1 incontrati a partire dal meno significativo, prima di incontrare un bit impostato a zero.
- All'inizio, $\Phi(D_0) = 0$ (nessun bit impostato a 1)
- Alla fine, $\Phi(D_n) \geq 0$
- La differenza di potenziale è non negativa

Programmazione Dinamica

Programmazione dinamica

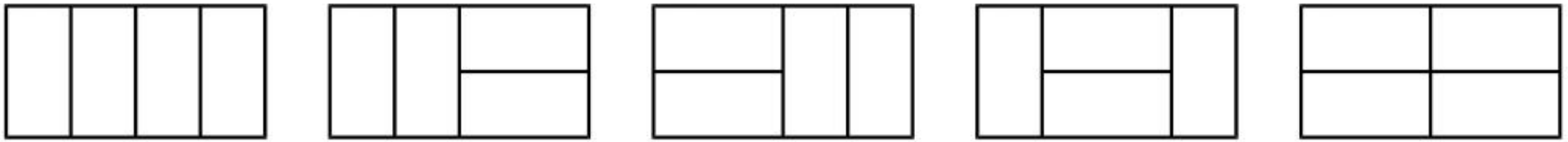
- Un metodo per spezzare un problema in sottoproblemi in maniera ricorsiva
- Ogni sottoproblema viene risolto una sola volta
- La soluzione ad un sottoproblema viene memorizzata in una tabella
- Se si incontra un sottoproblema già risolto si ricava la soluzione dalla tabella
- La tabella è facilmente indirizzabile (ricerca in $O(1)$)

Those who cannot remember the past are condemned to repeat it

—George Santayana, The Life of Reason, Introduction and Reason in Common Sense (1905)

Un esempio: domino lineare

- Il gioco del domino è basato su tessere di dimensione 2x1. Scrivere un algoritmo che prenda in input un intero *n* e restituisca il numero di possibili disposizioni in un rettangolo 2x*n*
- Fissando *n=4* si ottengono le seguenti soluzioni



Un approccio ricorsivo

- Proviamo a definire una formula ricorsiva per calcolare il numero di disposizioni possibili dato *n*:
  - Se non ho tessere, quante sono le disposizioni possibili?
  - Se ho una tessera, quante sono le disposizioni possibili?

Un approccio ricorsivo

- Proviamo a definire una formula ricorsiva per calcolare il numero di disposizioni possibili dato *n:*
  - Se non ho tessere, quante sono le disposizioni possibili?
    - una sola disposizione: *nessuna tessera*
  - Se ho una tessera, quante sono le disposizioni possibili?
    - una sola disposizione: *una tessera, in verticale*

Un approccio ricorsivo

- Proviamo a definire una formula ricorsiva per calcolare il numero di disposizioni possibili dato *n:*
  - Se non ho tessere, quante sono le disposizioni possibili?
    - una sola disposizione: *nessuna tessera*
  - Se ho una tessera, quante sono le disposizioni possibili?
    - una sola disposizione: *una tessera, in verticale*
  - Cosa succede se metto l'ultima tessera in verticale?
  - Cosa succede se metto l'ultima tessera in orizzontale?

Un approccio ricorsivo

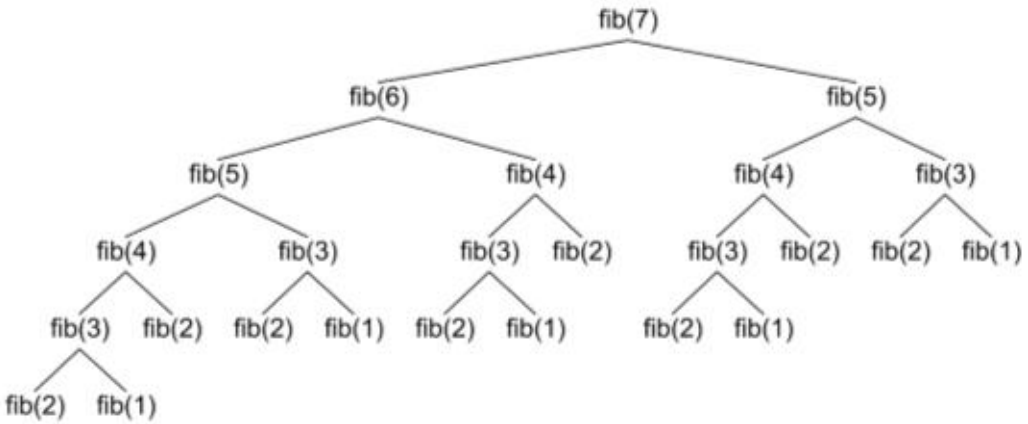
- Proviamo a definire una formula ricorsiva per calcolare il numero di disposizioni possibili dato *n:*
  - Se non ho tessere, quante sono le disposizioni possibili?
    - una sola disposizione: *nessuna tessera*
  - Se ho una tessera, quante sono le disposizioni possibili?
    - una sola disposizione: *una tessera, in verticale*
  - Cosa succede se metto l'ultima tessera in verticale?
    - Risolvo il problema di dimensione $n - 1$
  - Cosa succede se metto l'ultima tessera in orizzontale?
    - Ho bisogno di un'altra tessera: risolvo il problema di dimensione $n - 2$

Un approccio ricorsivo

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

- Il numero di disposizioni è quindi $a_n = a_{n-1} + a_{n-2}$ con $a_0 = 1$ e $a_1 = 1$
- Si tratta di una relazione di ricorrenza omogenea con $c_1 = 1$ e $c_2 = 1$ che abbiamo già incontrato:
  - $a_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$
- La sequenza generata è: 1, 1, 2, 3, 5, 8, 13, 21, 34, …

n-esimo numero di Fibonacci: secondo approccio



- Lo stesso valore viene ricalcolato più volte!
  - $T(n) \approx F(n) \approx \varphi^n$
- Adesso vediamo come si può fare di meglio!

n-esimo numero di Fibonacci: terzo approccio

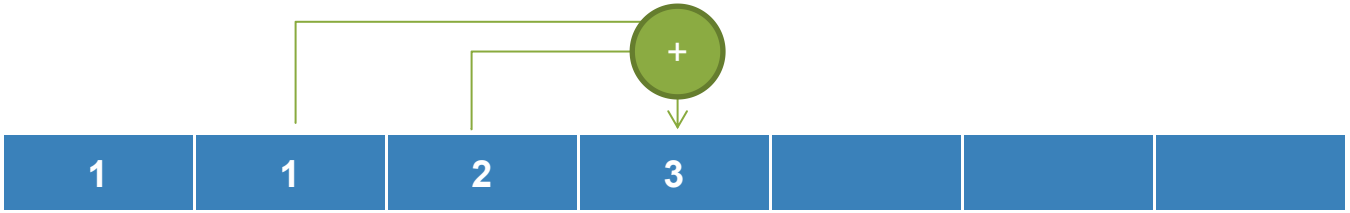
- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`

n-esimo numero di Fibonacci: terzo approccio

- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`

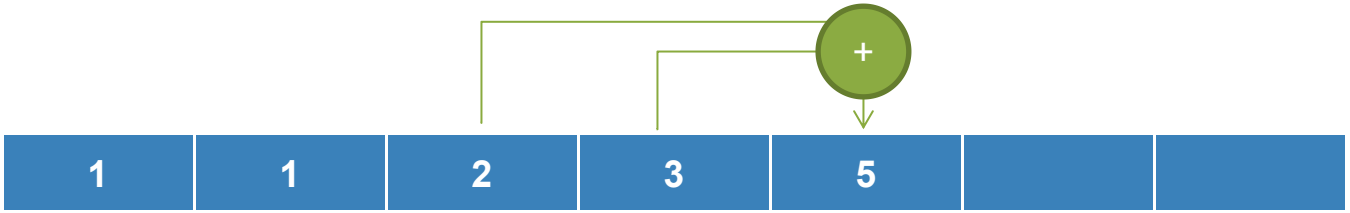
n-esimo numero di Fibonacci: terzo approccio

- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`



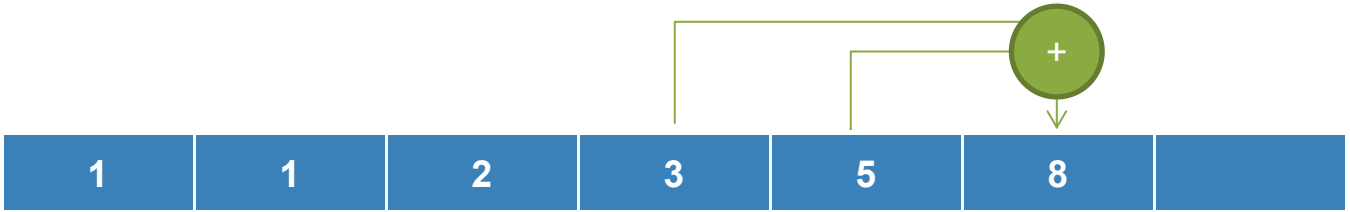
n-esimo numero di Fibonacci: terzo approccio

- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`



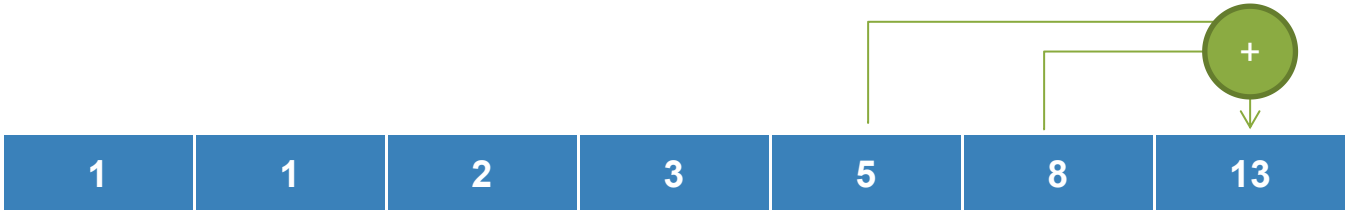
n-esimo numero di Fibonacci: terzo approccio

- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`



n-esimo numero di Fibonacci: terzo approccio

- Applichiamo la tecnica della programmazione dinamica e, anziché ricalcolare più volte gli stessi valori, li salviamo in una tabella
  - Utilizziamo un vettore: un elemento per ogni sottoproblema da risolvere
  - `Fib[i]` memorizza F_i
  - Per calcolare `Fib[i]` utilizziamo `Fib[i-1]` e `Fib[i-2]`



n-esimo numero di Fibonacci: terzo approccio

```
def fibonacci3(n):
    Fib = [0] * n
    Fib[0] = 1
    Fib[1] = 1

    for i in range(2, n):
        Fib[i] = Fib[i-1] + Fib[i-2]
    return Fib[n-1]
```

- Qual è la complessità in *tempo*?
- Qual è la complessità in *spazio*?

n-esimo numero di Fibonacci: terzo approccio

```
def fibonacci3(n):
    Fib = [0] * n
    Fib[0] = 1
    Fib[1] = 1

    for i in range(2, n):
        Fib[i] = Fib[i-1] + Fib[i-2]
    return Fib[n-1]
```

- Qual è la complessità in *tempo*?
  - $T(n) = \Theta(n)$
- Qual è la complessità in *spazio*?
  - $S(n) = \Theta(n)$

n-esimo numero di Fibonacci: terzo approccio

- Per quale motivo dobbiamo “ricordare” tutta la sequenza di Fibonacci?
- Ci basta ricordare soltanto gli ultimi due elementi

I've got news for Mr. Santayana: we're doomed to repeat the past no matter what.

—Kurt Vonnegut

n-esimo numero di Fibonacci: quarto approccio

```
def fibonacci4(n):
    F0 = 1
    F1 = 1
    F2 = 1

    for i in range(2, n):
        F0 = F1
        F1 = F2
        F2 = F0 + F1
    return F2
```

- Qual è la complessità in *spazio*?

n-esimo numero di Fibonacci: quarto approccio

```
def fibonacci4(n):
    F0 = 1
    F1 = 1
    F2 = 1

    for i in range(2, n):
        F0 = F1
        F1 = F2
        F2 = F0 + F1
    return F2
```

- Qual è la complessità in *spazio*?
  - $S(n) = \Theta(1)$

n-esimo numero di Fibonacci: quarto approccio

- È proprio vero che la complessità spaziale è $S(n) = \Theta(1)$?

n-esimo numero di Fibonacci: quarto approccio

- È proprio vero che la complessità spaziale è $S(n) = \Theta(1)$?
- La sequenza generata è: 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Quanti bit servono per memorizzare $F(n)$?
  - $\log F(n) = \Theta(n)$
- Quanto costa sommare due numeri di Fibonacci consecutivi?

n-esimo numero di Fibonacci: quinto approccio

- Possiamo utilizzare l'esponenziale di matrici per calcolare i numeri di Fibonacci:

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

- Questa relazione è ovvia per $n = 1$
- Possiamo provare la sua correttezza in maniera induttiva

n-esimo numero di Fibonacci: quinto approccio

- Assumiamo che la relazione sia vera per un qualche 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 lati per $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k+1} = \begin{bmatrix} F_k + F_{k+1} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

- Pertanto la relazione vale per ogni n

n-esimo numero di Fibonacci: quinto approccio

FIBONACCI5(n):

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

for i ← 1 **to** n:

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

return M[0][0]

- Cosa ci abbiamo guadagnato?

n-esimo numero di Fibonacci: sesto approccio

- Possiamo calcolare la potenza n-esima elevando al quadrato la potenza $(n/2)$-esima
- Se n è dispari, eseguiamo una ulteriore moltiplicazione
- Si tratta di un'applicazione anche della tecnica Divide et Impera

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

n-esimo numero di Fibonacci: sesto approccio

```
PotenzaDiMatrice(A, k):
        if k ≤ 1 then
                M ← [1 0; 0 1]
        else
                M ←PotenzaDiMatrice(A, ⌊k/2⌋)
                M ←M • M
         if k is odd then
                M ←M • A
         return M

Fibonacci6(n):
        A ← [1 1; 1 0]
        M ←PotenzaDiMatrice(A, n − 1)
        return M[0][0]
```

n-esimo numero di Fibonacci: sesto approccio

```
PotenzaDiMatrice(A, k):
    if k ≤ 1 then
        M ← [1 0; 0 1]
    else
        M ←PotenzaDiMatrice(A, ⌊k/2⌋)
        M ←M • M
     if k is odd then
        M ←M • A
     return M
```

Il tempo d'esecuzione è $O(\log n)$

```
Fibonacci6(n):
    A ← [1 1; 1 0]
    M ←PotenzaDiMatrice(A, n − 1)
    return M[0][0]
```

Knapsack

- Dato un insieme di oggetti, ciascuno caratterizzato da un *peso* ed un *profitto*, ed uno zaino con un limite di capacità, individuare un sottoinsieme di oggetti che:
  - totalizzano un peso inferiore alla capacità dello zaino
  - massimizzano il valore degli oggetti inseriti nello zaino

- Input:
  - un vettore w dove $w[i]$ è il peso dell'oggetto i-esimo
  - un vettore p dove $p[i]$ è il profitto dell'oggetto i-esimo
  - la capacità C dello zaino
- Output:
  - Un insieme $S \subseteq \{1, \ldots, n\}$ tale che:
    - $w(S) = \sum_{i \in S} w[i] \leq C$
    - $p(S) = \sum_{i \in S} p[i]$

Knapsack

- Dato uno zaino di capacità C ed n oggetti di peso w e profitto p, DP[i][c] è il massimo profitto che si può ottenere dai primi $i \leq n$ oggetti contenuti in uno zaino di capacità $c \leq C$.

- Il massimo profitto ottenibile è DP[n][C].

Knapsack: parte ricorsiva

- Partiamo dall'ultimo oggetto
- Cosa succede se non lo prendiamo?
  - $DP[i][c] = DP[i-1][c]$
    - la capacità non cambia e non c'è profitto
- Cosa succede se lo prendiamo?
  - $DP[i][c] = DP[i-1][c-w[i]] + p[i]$
    - diminuisce la capacità ma aumenta il profitto

- Come scegliere?
  - $DP[i][c] = \max\left(DP[i-1][c]\,, DP[i-1][c-w[i]] + p[i]\right)$

Knapsack: casi base

- Qual è il profitto se non si hanno più oggetti?
- Qual è il profitto se non si ha più capacità?
- Cosa succede se la capacità è negativa?

$$DP[i][c] = \begin{cases} 0 & i = 0 \text{ o } c = 0 \\ -\infty & c < 0 \end{cases}$$

Knapsack

- Mettendo tutto insieme:

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

Knapsack

```
KNAPSACK(w, p, n, C):
    DP = [0...n][0...C]
    for i ← 0 to n:
        DP[i][0] = 0
    for c ← 0 to C:
        DP[0][c] = 0
    for i ← 0 to n:
        for c ← 0 to C:
            if w[i] ≤ 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[n][C]
```

Un esempio

- w = [4, 2, 3, 4]
- p = [10, 7, 8, 6]
- C = 9

	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

Knapsack: complessità computazionale

- Qual è la complessità computazionale?

Knapsack: complessità computazionale

- Qual è la complessità computazionale?
  - $T(n) = O(nC)$

- È un algoritmo polinomiale?

Knapsack: complessità computazionale

- Qual è la complessità computazionale?
  - $T(n) = O(nC)$

- È un algoritmo polinomiale?
  - No!
  - Sono necessari $k = \log C$ bit per rappresentare C:
  - $T(n) = O(n2^k)$
  - Si tratta di un algoritmo *pseudo-polinomiale*

Algoritmi Greedy

Algoritmi Greedy

- Gli algoritmi greedy (“golosi”, o meglio “ingordi”) sono algoritmi che si basano su una “scelta”
  - Non è molto diverso dalla situazione che abbiamo visto nel problema di Knapsack
- Tuttavia, a differenza di altre tecniche algoritmiche, gli algoritmi greedy non analizzano tutte le possibilità per fare la loro scelta
  - Identificano quella che sembra, in quella fase dell'esecuzione, la scelta ottima
  - Non è detto che si tratti di un ottimo globale!
- In alcuni casi, è possibile dimostrare che l'ottimo locale corrisponde all'ottimo globale
  - Altrimenti, si rischia di individuare una soluzione subottima

Problema del resto

- Problema molto comune, legato ai distributori automatici
- Input:
  - un vettore di interi positivi $t[1 \ldots n]$ che descrive i tagli di monete disponibili
  - un intero R pari al resto che deve essere restituito
- Problema:
  - Individuare un vettore x di interi non negativi tale che:
    $R = \sum_{i=1}^{n} x[i] \cdot t[i]$ e $m = \sum_{i=1}^{n} x[i]$ ha valore minimo
  - Significa trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli memorizzati in t, assumendo di avere un numero illimitato di monete per ogni taglio.

Un primo approccio: programmazione dinamica

- Sia $S(i)$ il problema di dare un resto pari ad i
- Assumiamo di avere $A(i)$, soluzione ottima di $S(i)$, e sia $j \in A(i)$
- Allora, $S(i - t[j])$ è un sottoproblema di $S(i)$, la cui soluzione ottima è $A(i) - \{j\}$.

- Possiamo quindi dare la seguente definizione ricorsiva:
  - $DP[0 \ldots R]$, tabella di programmazione dinamica
  - $DP[i]$: numero minimo di monete necessario a risolvere $S(i)$
  - $$DP[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{1 \le j \le n} \{DP[i - t[j]] \mid t[j] \le i\} + 1 & i > 0 \end{cases}$$

Un primo approccio: programmazione dinamica

```
RESTO(t, n, R):
    DP = [0…R]
    S = [0…R]
    DP[0] ← 0
    for i ← 1 to R:
        DP[i] ← +∞
        for j ← 1 to n:
            if i > t[j] and DP[i - t[j]] + 1 < DP[i] then
                DP[i] ← DP[i - t[j]] + 1
                S[i] ← j
    …
    …
        x = [1…n]
        for i ← 1 to n:
            x[i] ← 0
        while R > 0:
            x[S[r]] = x[S[r]] + 1
            R = R - t[S[R]]
        return x
```

Un primo approccio: programmazione dinamica

```
RESTO(t, n, R):
    DP = [0…R]
    S = [0…R]
    DP[0] ← 0
    for i ← 1 to R:
        DP[i] ← +∞
        for j ← 1 to n:
            if i > t[j] and DP[i - t[j]] + 1 < DP[i] then
                DP[i] ← DP[i - t[j]] + 1
                S[i] ← j
    …
    …
        x = [1…n]
            for i ← 1 to n:
                x[i] ← 0
            while R > 0:
                x[S[r]] = x[S[r]] + 1
                R = R - t[S[R]]
            return x
```

- Complessità?

Un primo approccio: programmazione dinamica

```
RESTO(t, n, R):
    DP = [0…R]
    S = [0…R]
    DP[0] ← 0
    for i ← 1 to R:
        DP[i] ← +∞
        for j ← 1 to n:
            if i > t[j] and DP[i - t[j]] + 1 < DP[i] then
                DP[i] ← DP[i - t[j]] + 1
                S[i] ← j
    …
    …
        x = [1…n]
        for i ← 1 to n:
            x[i] ← 0
        while R > 0:
            x[S[r]] = x[S[r]] + 1
            R = R - t[S[R]]
        return x
```

- Complessità? $O(nR)$

Applichiamo l'approccio greedy

- Si seleziona la moneta j più grande tale che $t[j] \leq R$
- Si risolve poi il problema $S(R - t[j])$

```
RESTO(t, n, R):
  for i = 1 to n:
    x[i] = ⌊R/t[i]⌋
    R = R – x[i] · t[i]
  return x
```

- Qual è il costo?

Applichiamo l'approccio greedy

- Si seleziona la moneta j più grande tale che $t[j] \leq R$
- Si risolve poi il problema $S(R - t[j])$

RESTO(t, n, R):

```
for i = 1 to n:
    x[i] = ⌊R/t[i]⌋
    R = R – x[i] · t[i]
return x
```

- Qual è il costo?
  - Assumendo che t[i] sia già ordinato, $O(n)$

Problema dello scheduling

- Supponiamo di avere un processore ed *n* job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ noto a priori
- Trovare una sequenza (permutazione) di esecuzione che minimizzi il *tempo di completamento medio*
- Dato un vettore $A[1 \ldots n]$ contenete una sequenza $\{1, \ldots, n\}$, il tempo di completamento del k-esimo job è dato da:

$$T_A(k) = \sum_{i=1}^{k} t[A[i]]$$

Problema dello scheduling

- Un possibile scheduling con 4 job di durata 4, 1, 6, 3:

- Tempo di completamento medio: $(4 + 5 + 11 + 14)/4 = 8.5$

Problema dello scheduling

- Un possibile scheduling con 4 job di durata 4, 1, 6, 3:

- Tempo di completamento medio: $(4 + 5 + 11 + 14)/4 = 8.5$

- Cerchiamo di applicare una scelta ingorda:
  - scegliamo per primo il job con durata minore
- L'idea è intuitiva: inserendo il job più corto all'inizio, questo peserà di meno nella sommatoria

Shortest Job First

- Lo scheduling con gli stessi 4 job di durata 4, 1, 6, 3 diventa:

- Tempo di completamento medio: $(1 + 4 + 8 + 14)/4 = 6.75$

Backtracking

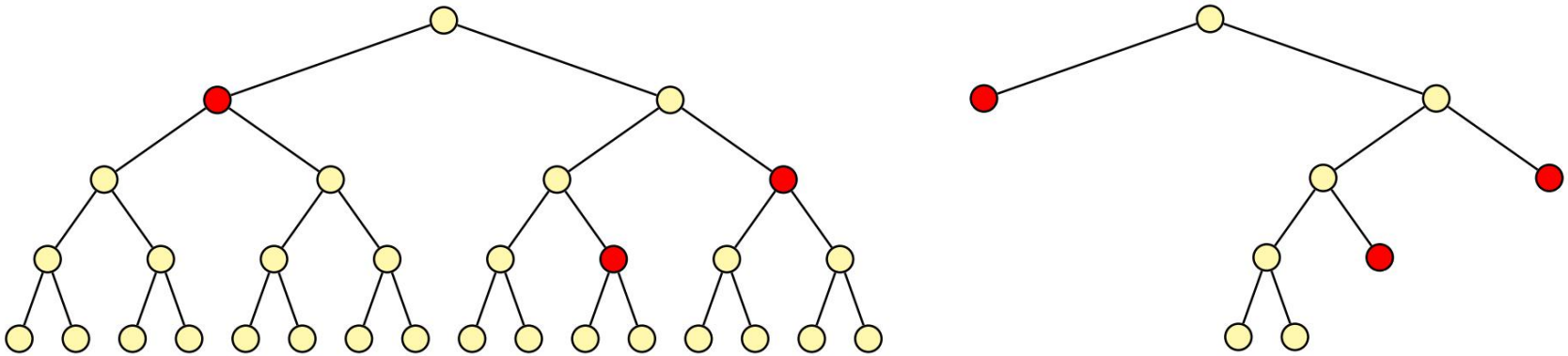
Bracktracking

- È una tecnica algoritmica che si adatta a classi di problemi decisionali, di ricerca, di ottimizzazione
  - Tipicamente, questi problemi hanno un sottoinsieme di soluzioni ammissibili
- Si basa su un'idea semplice:
  - Prova a fare qualcosa
  - Se non va bene, butta via un pezzo del lavoro fatto
  - Ritenta e sarai più fortunato

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

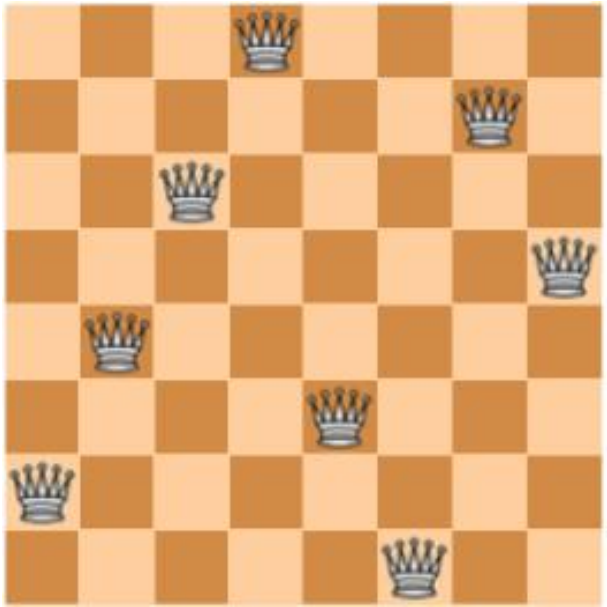
Pruning

- La “potatura” consente di escludere una parte dello spazio di ricerca che non fornisce soluzioni ammissibili



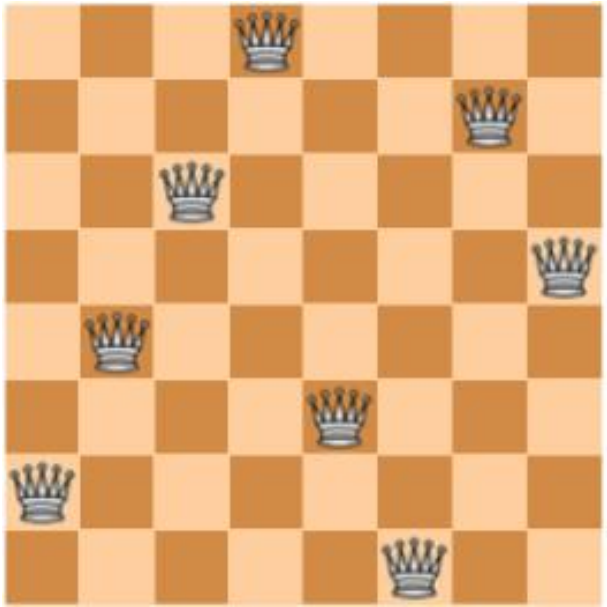
Un esempio: il problema delle otto regine

- Posizionare *n* regine su una scacchiera *n* x *n* in maniera tale che nessuna regina possa mangiare nessun'altra regina
- Presentato nel 1848 da Max Bezzel
- Gauss trovò 72 delle 92 possibili soluzioni



Analisi del problema

- Utilizziamo un vettore $S[1 \ldots n]$ per rappresentare le colonne
  - In ogni colonna ci deve essere soltanto una regina!
- $S[i]$ identifica la riga della regina nella colonna i-esima
- L'algoritmo termina quando $i = n$
- Il pruning elimina le diagonali



Implementazione

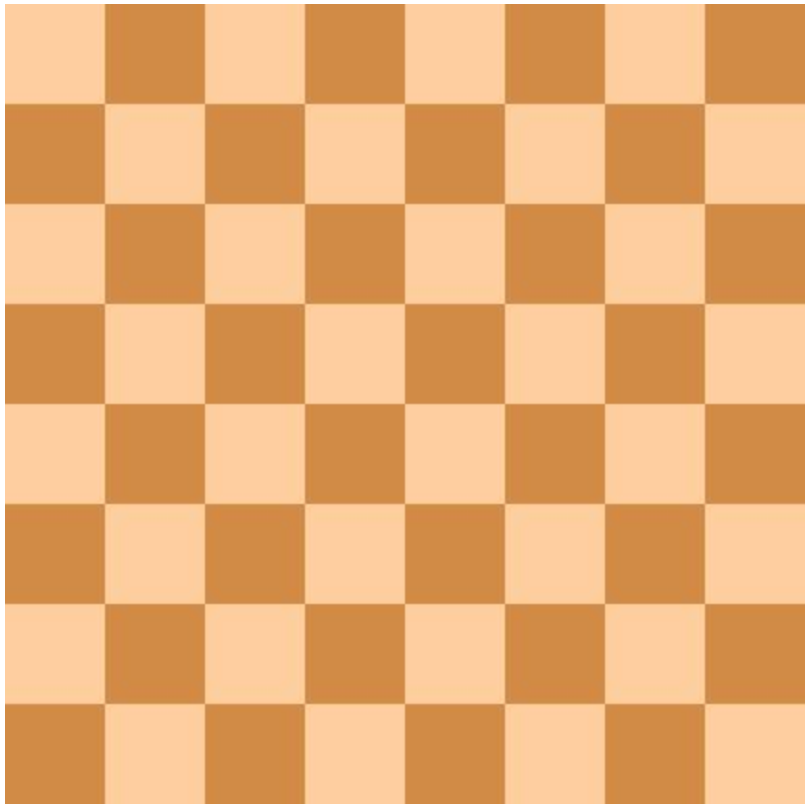
```
def solve(n, i, a, b, c):
  if i < n:
    for j in range(n):
      if j not in a and i+j not in b and i-j not in c:
        for solution in solve(n, i+1, a+[j], b+[i+j], c+[i-j]):
          yield solution
  else:
    yield a

for solution in solve(8, 0, [], [], []):
    print(solution)
```

diagonali “coperte”

Esecuzione

- `a=[]`
- `b=[]`
- `c=[]`

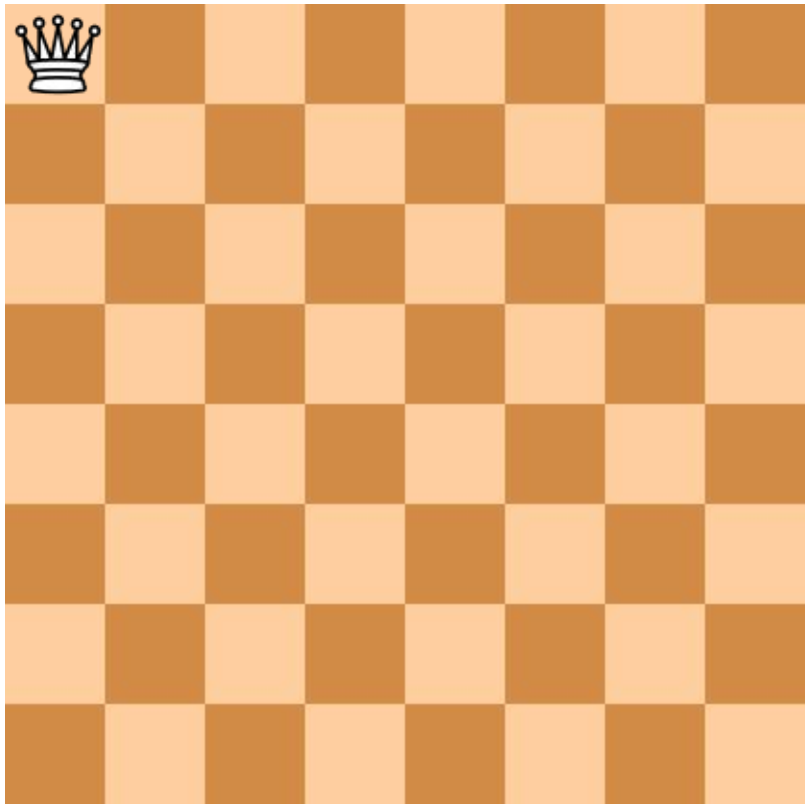
- `i=0`



Esecuzione

- `a=[0]`
- `b=[0]`
- `c=[0]`

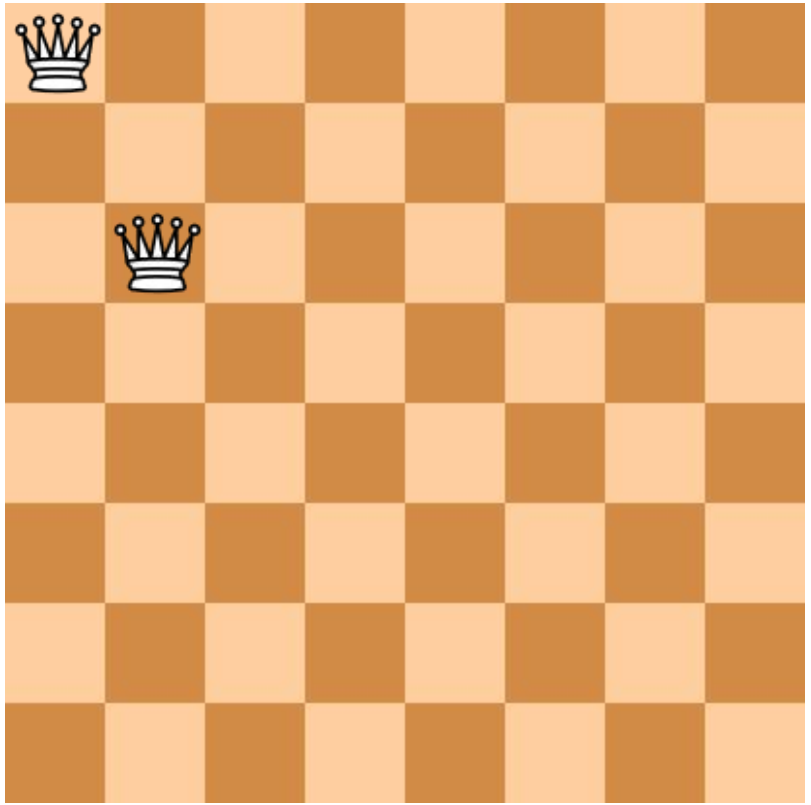
- `i=1`



Esecuzione

- `a=[0, 2]`
- `b=[0, 3]`
- `c=[0, -1]`

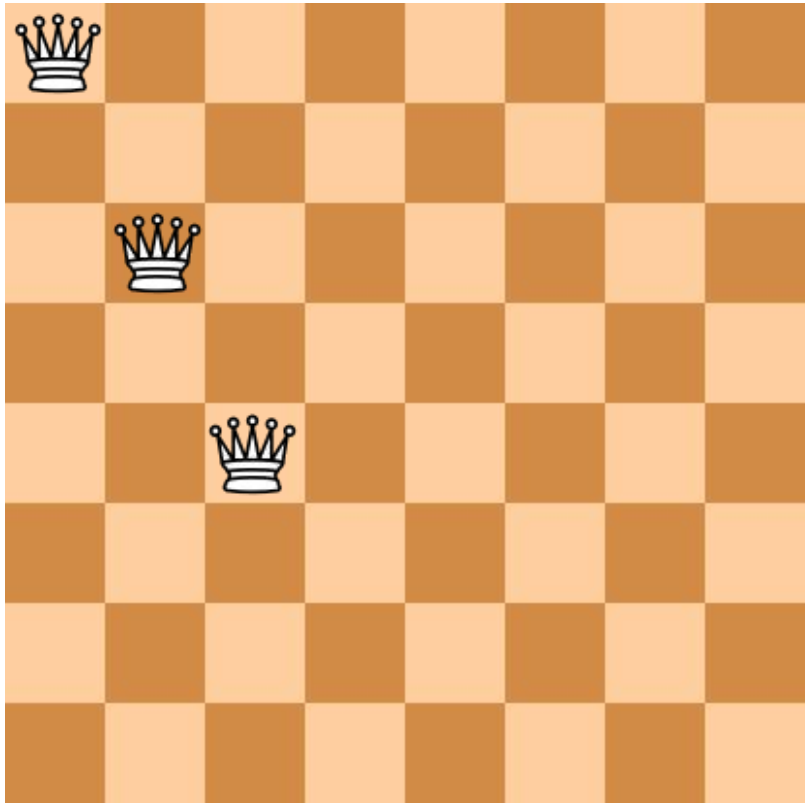
- `i=2`



Esecuzione

- `a=[0, 2, 4]`
- `b=[0, 3, 6]`
- `c=[0, -1, -2]`

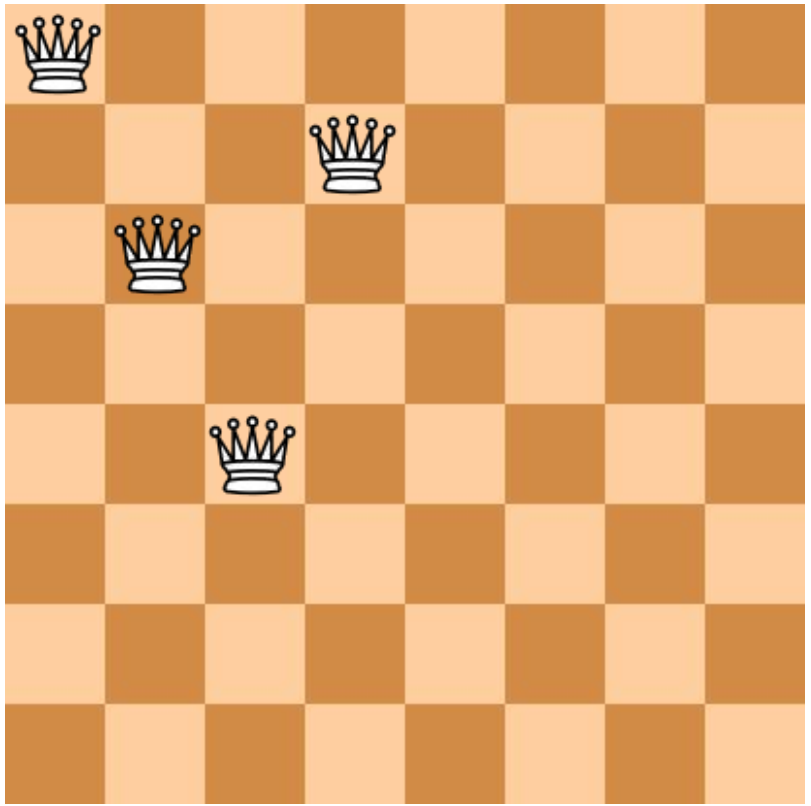
- `i=3`



Esecuzione

- `a=[0, 2, 4, 1]`
- `b=[0, 3, 6, 4]`
- `c=[0, -1, -2, 2]`

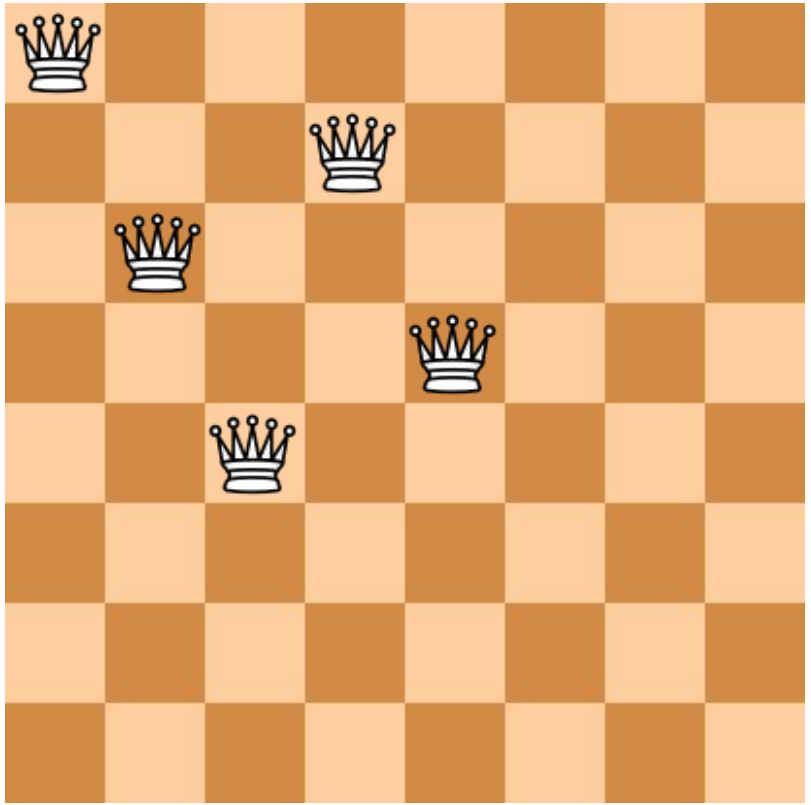
- `i=4`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

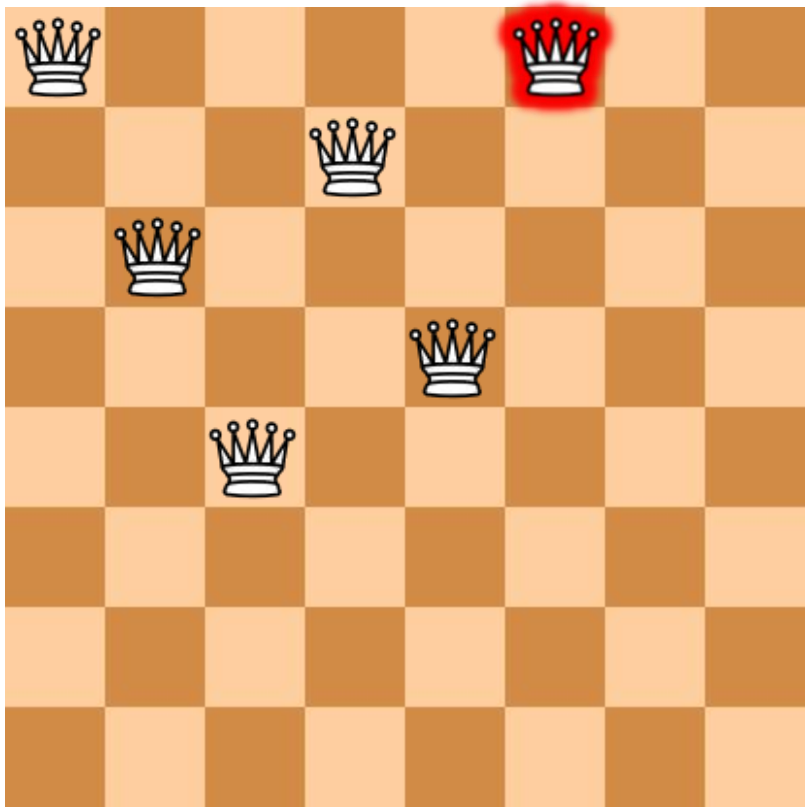
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

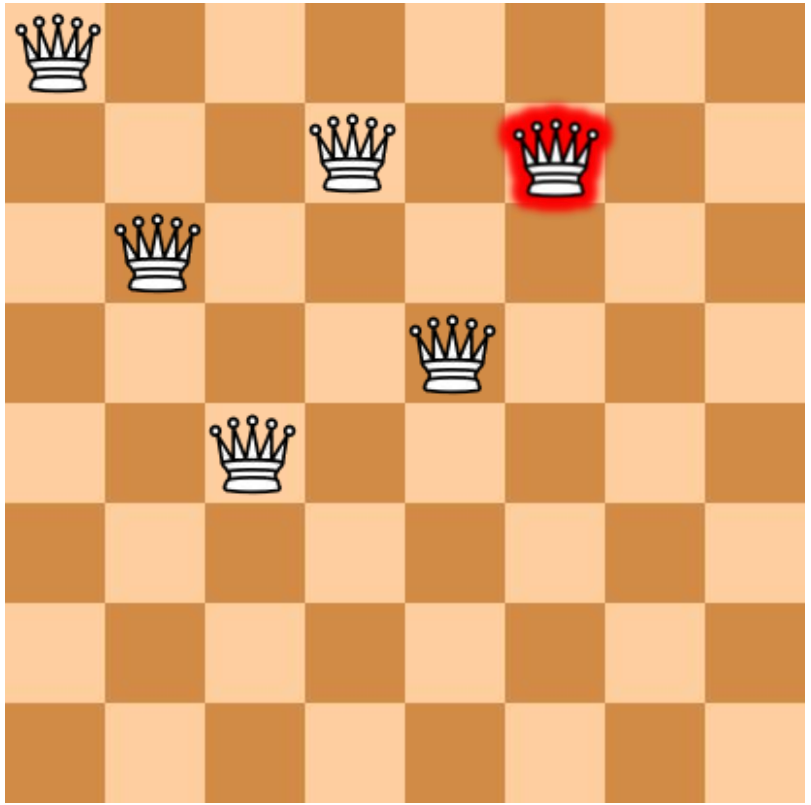
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

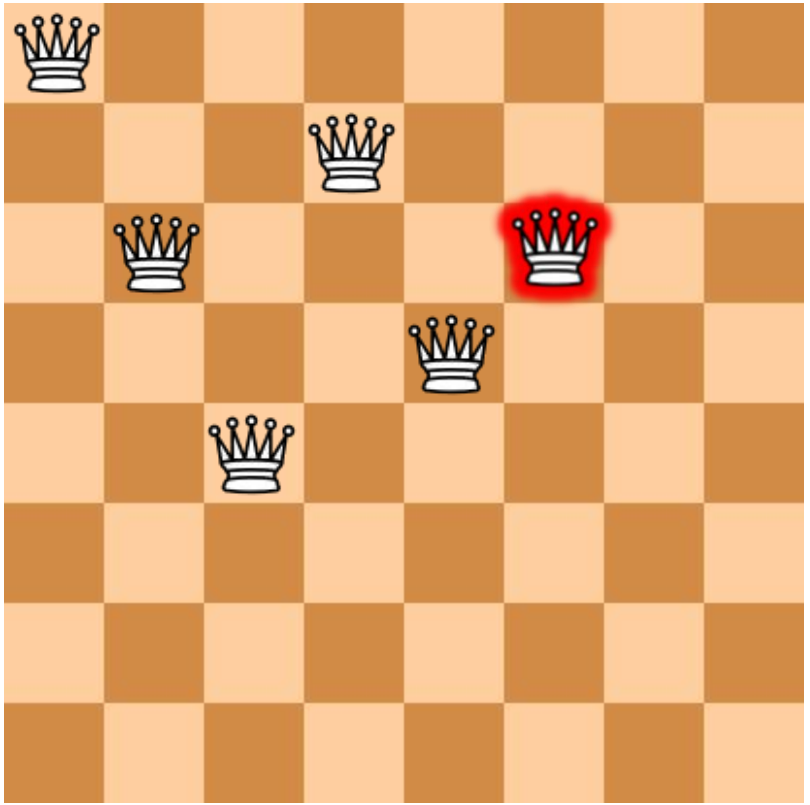
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

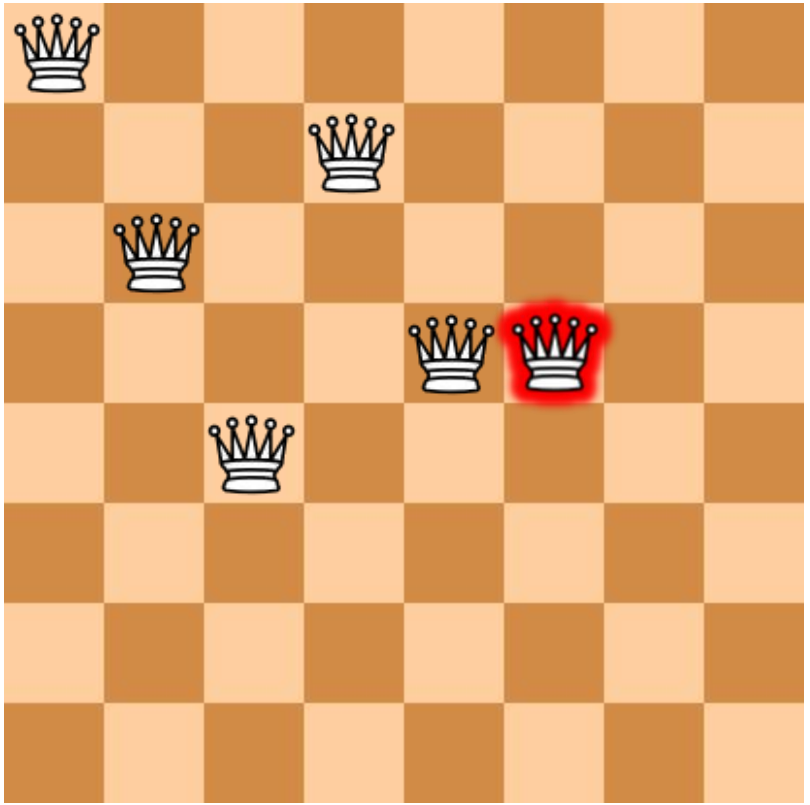
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

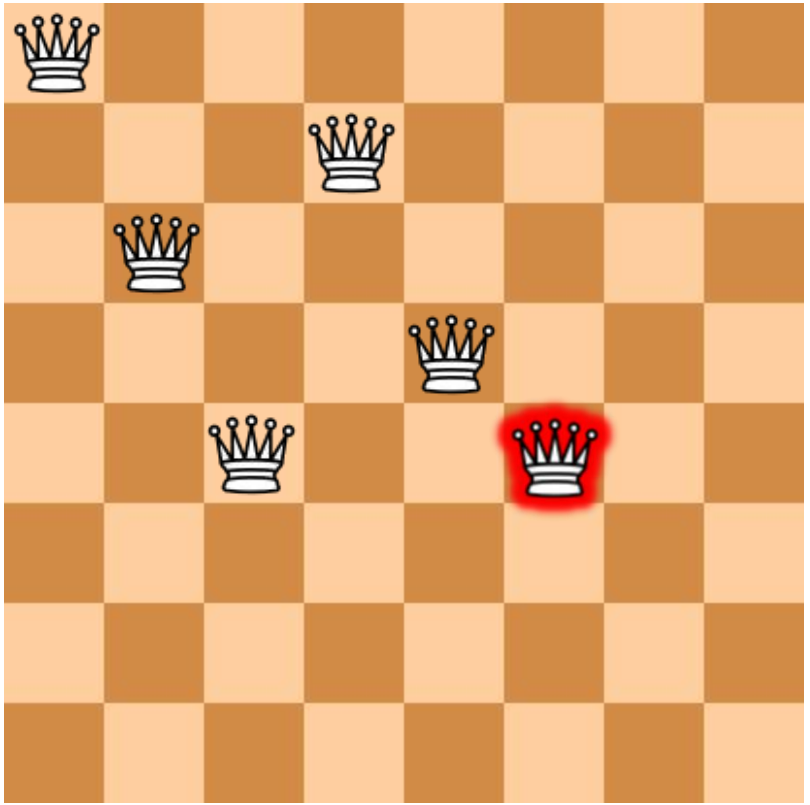
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

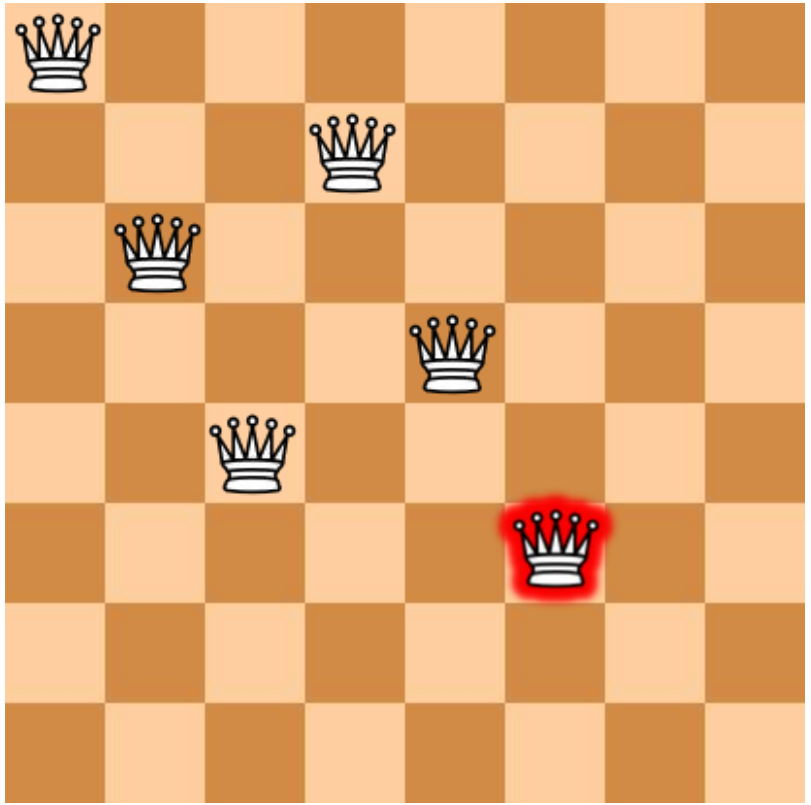
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

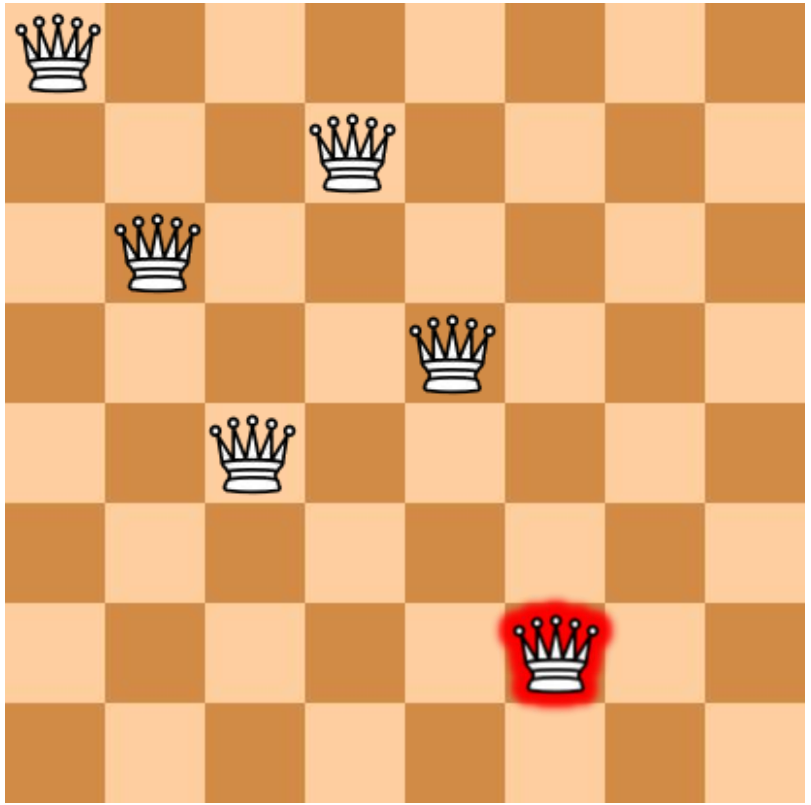
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

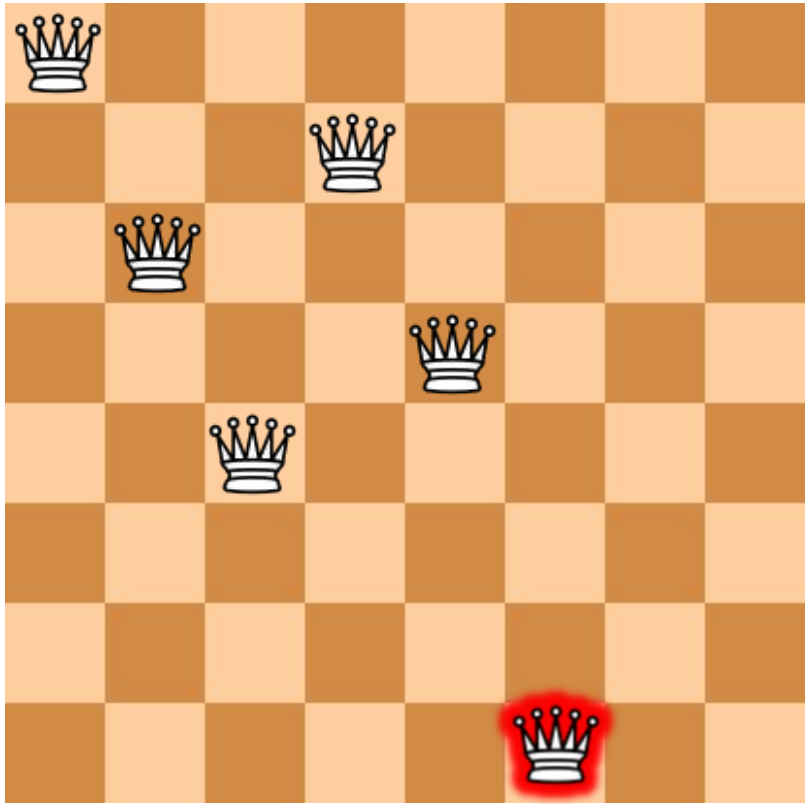
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

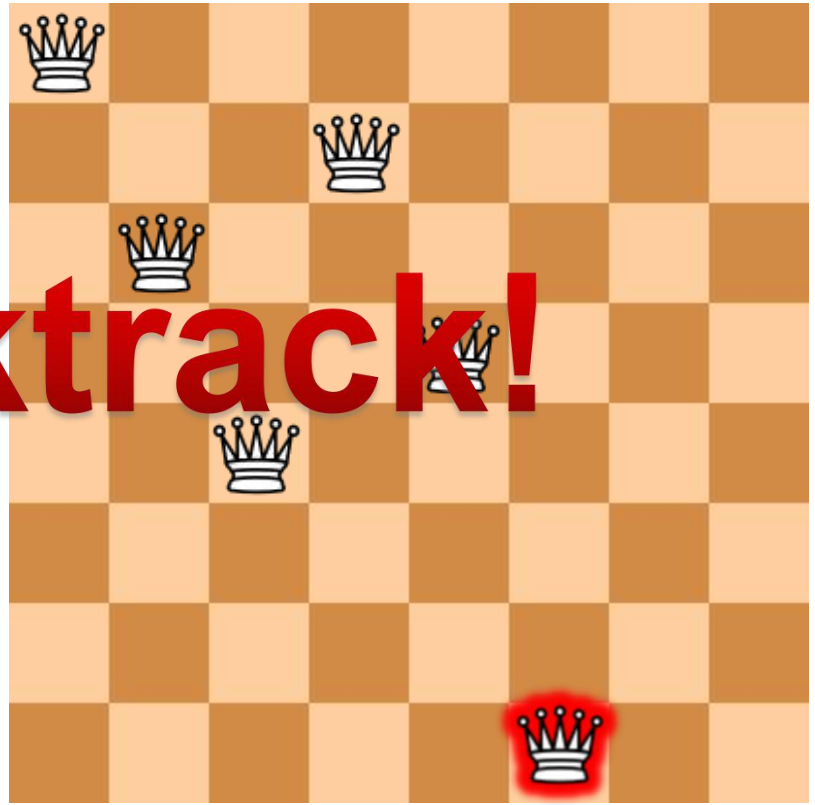
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 3]`
- `b=[0, 3, 6, 4, 7]`
- `c=[0, -1, -2, 2, 1]`

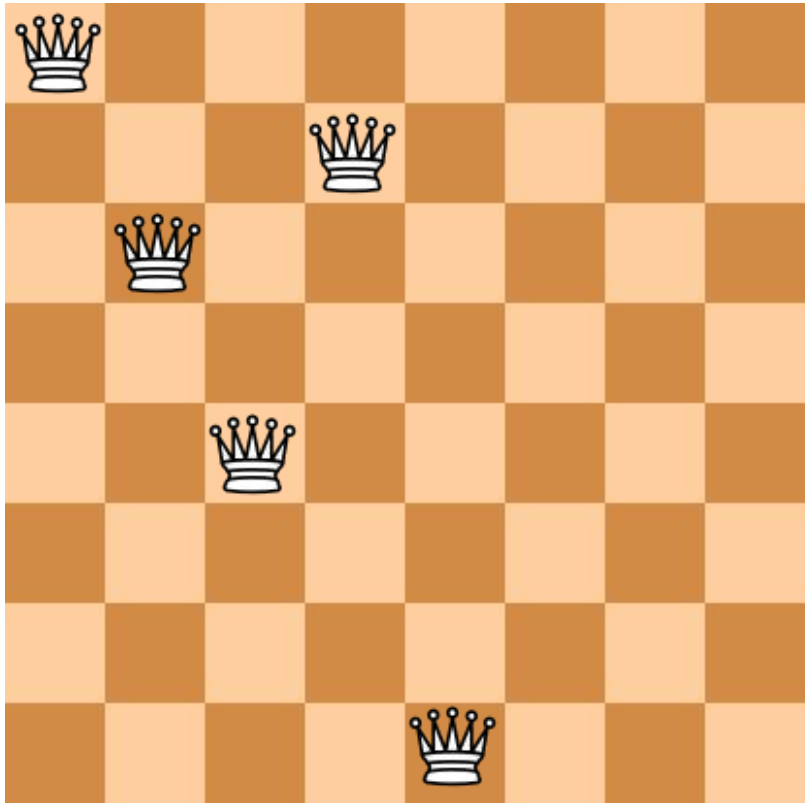
- `i=5`



Esecuzione

- `a=[0, 2, 4, 1, 7]`
- `b=[0, 3, 6, 4, 11]`
- `c=[0, -1, -2, 2, -3]`

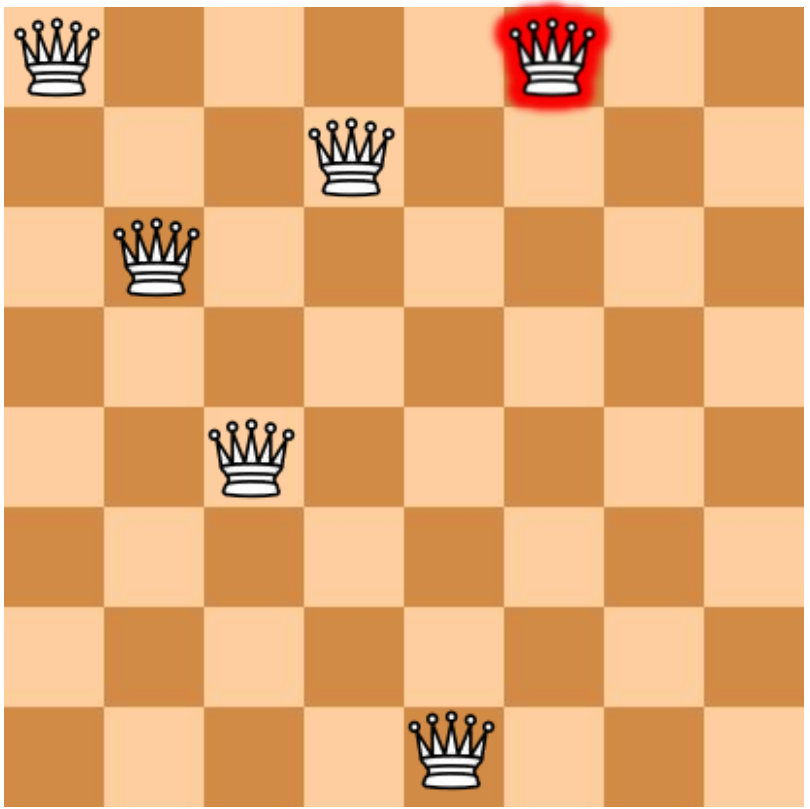
- `i=5`



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

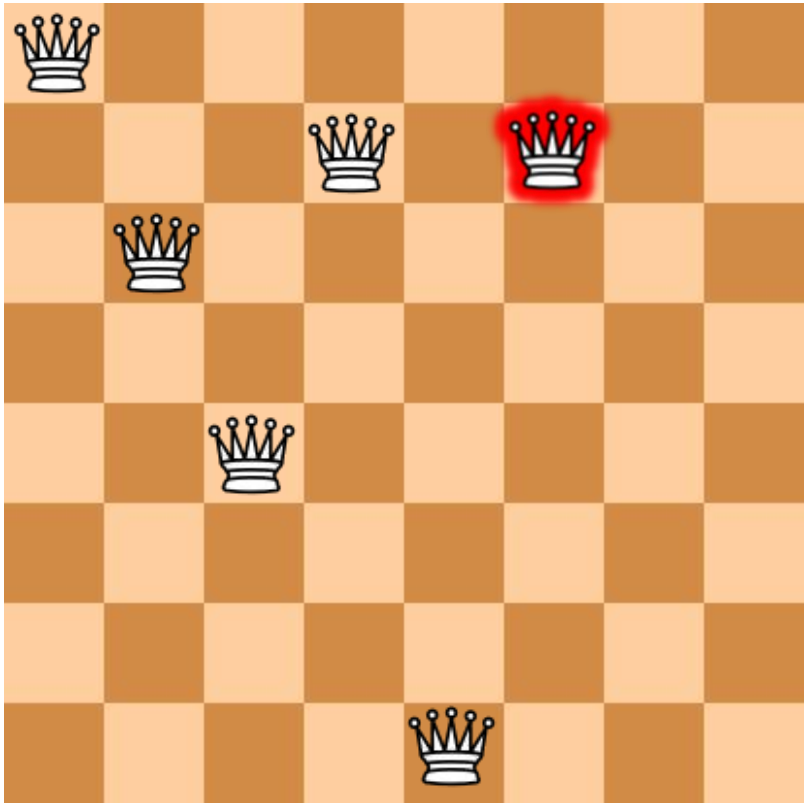
- i=5



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

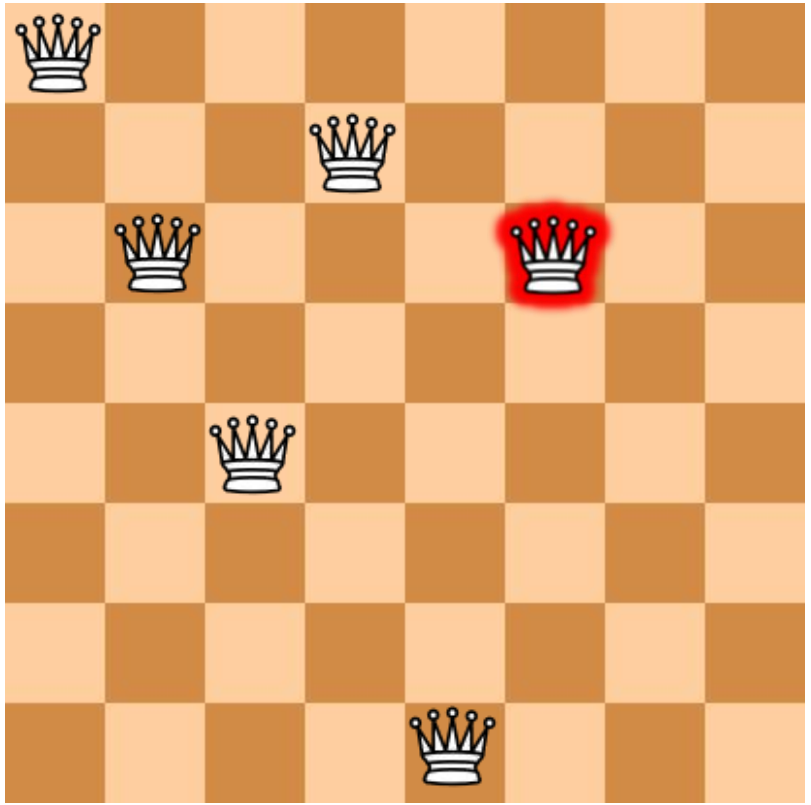
- i=5



Esecuzione

- `a=[0, 2, 4, 1, 7]`
- `b=[0, 3, 6, 4, 11]`
- `c=[0, -1, -2, 2, -3]`

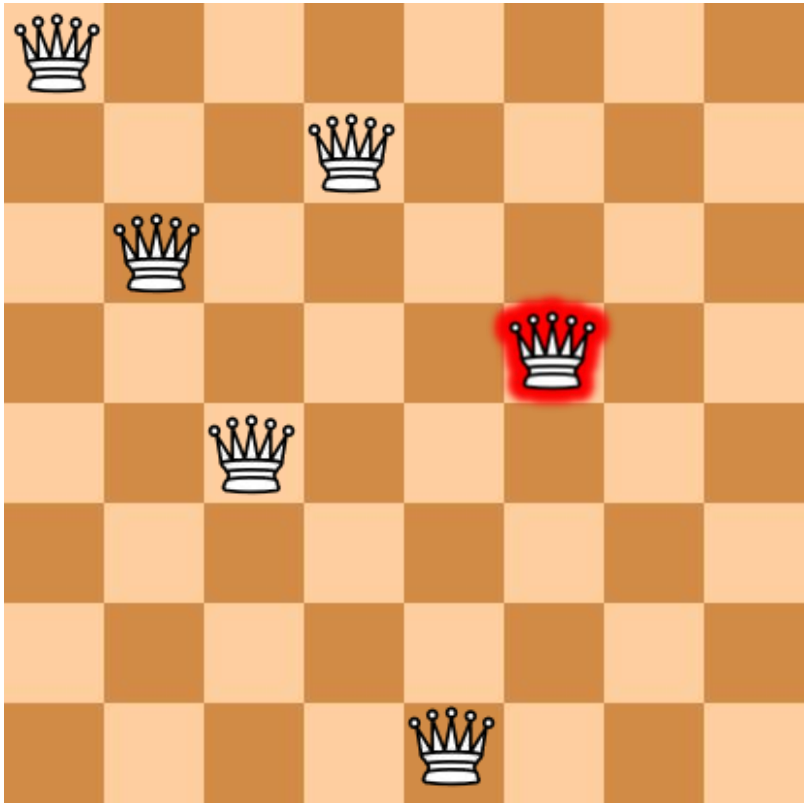
- `i=5`



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

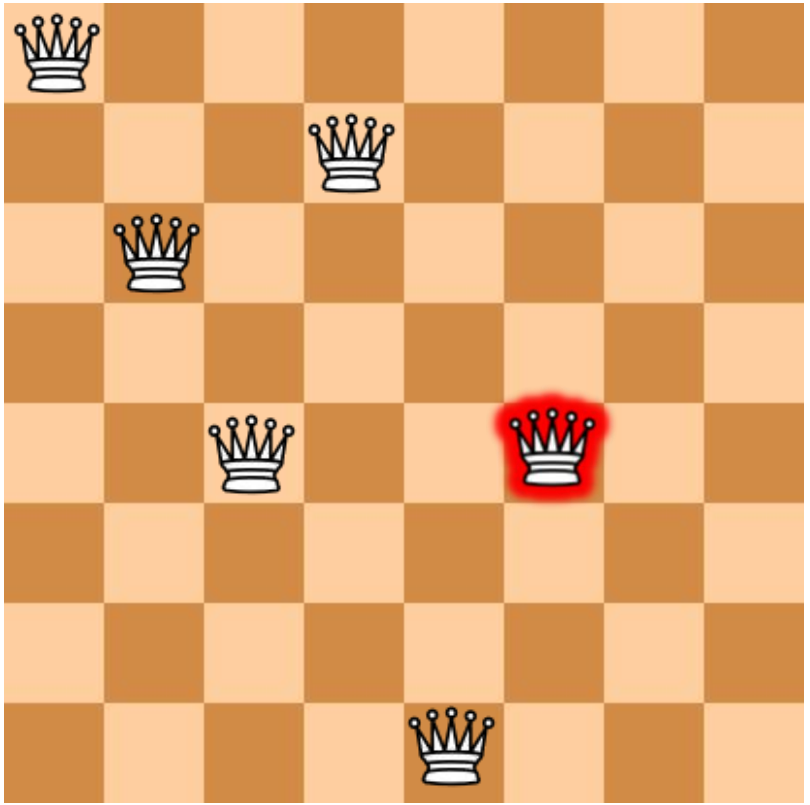
- i=5



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

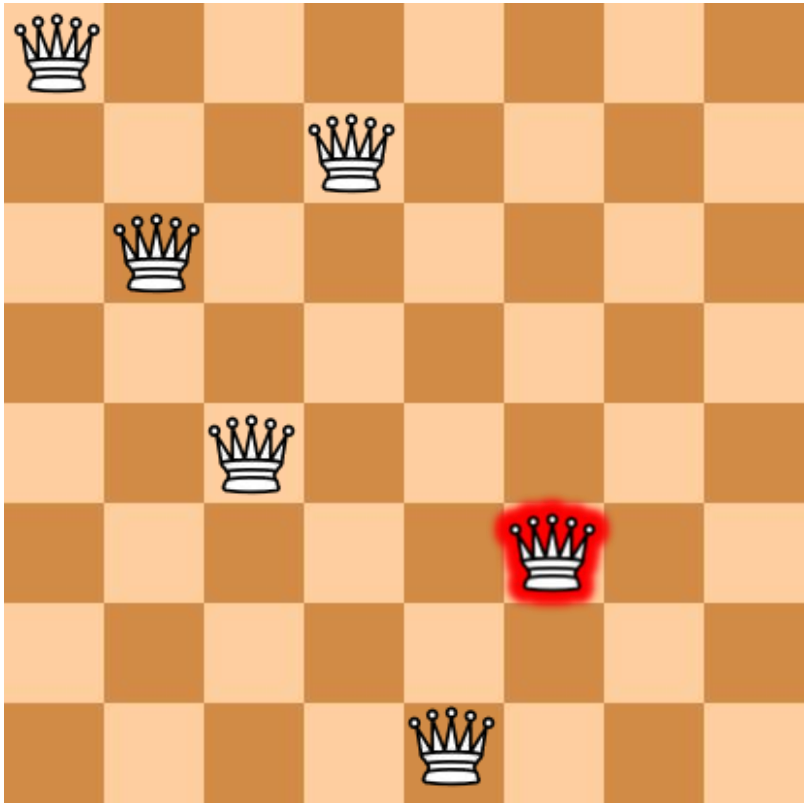
- i=5



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

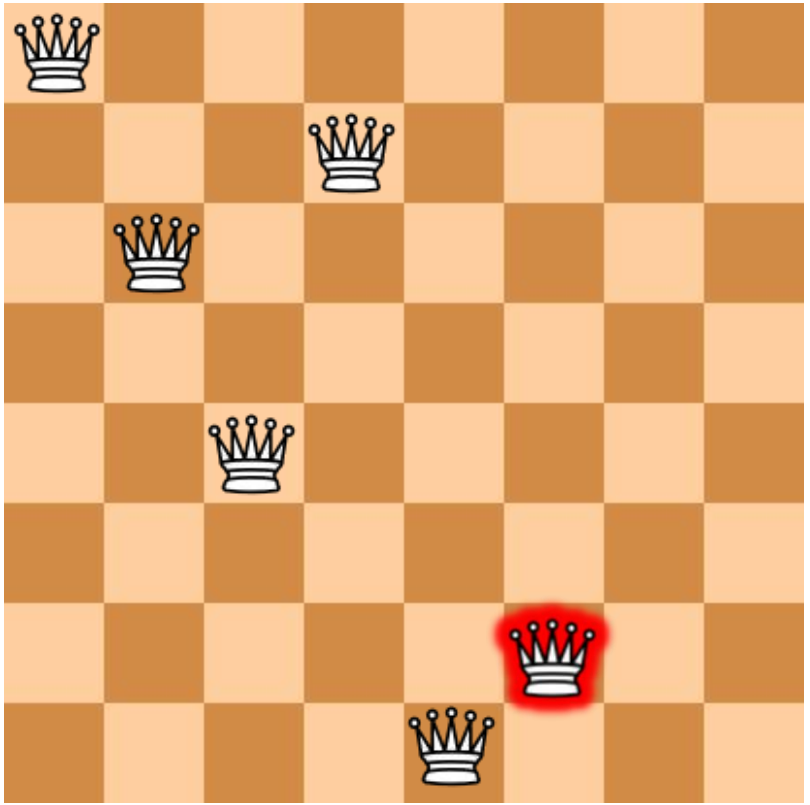
- i=5



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

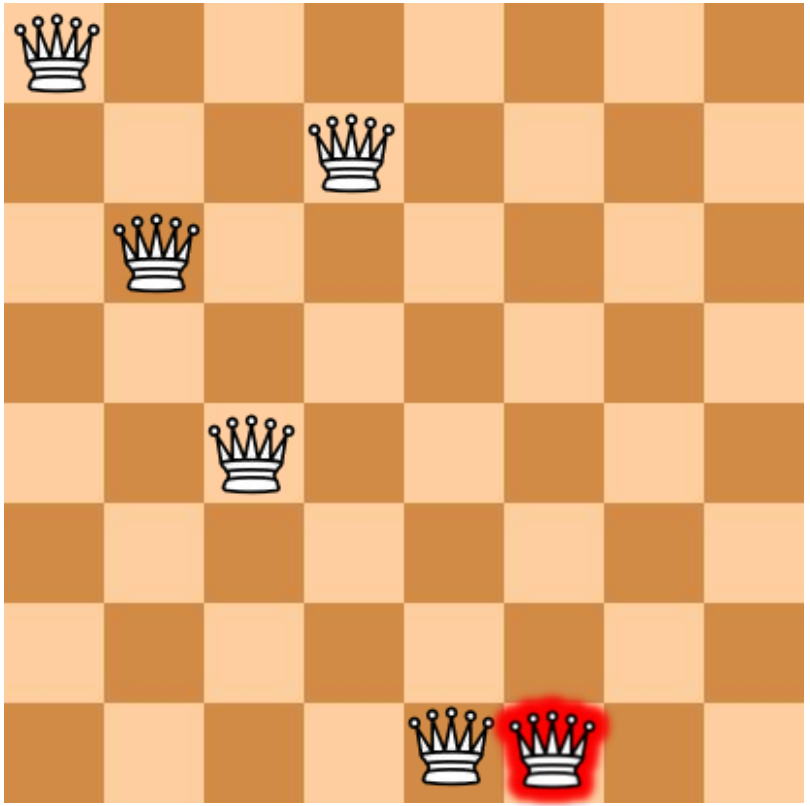
- i=5



Esecuzione

- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

- i=5



Esecuzione

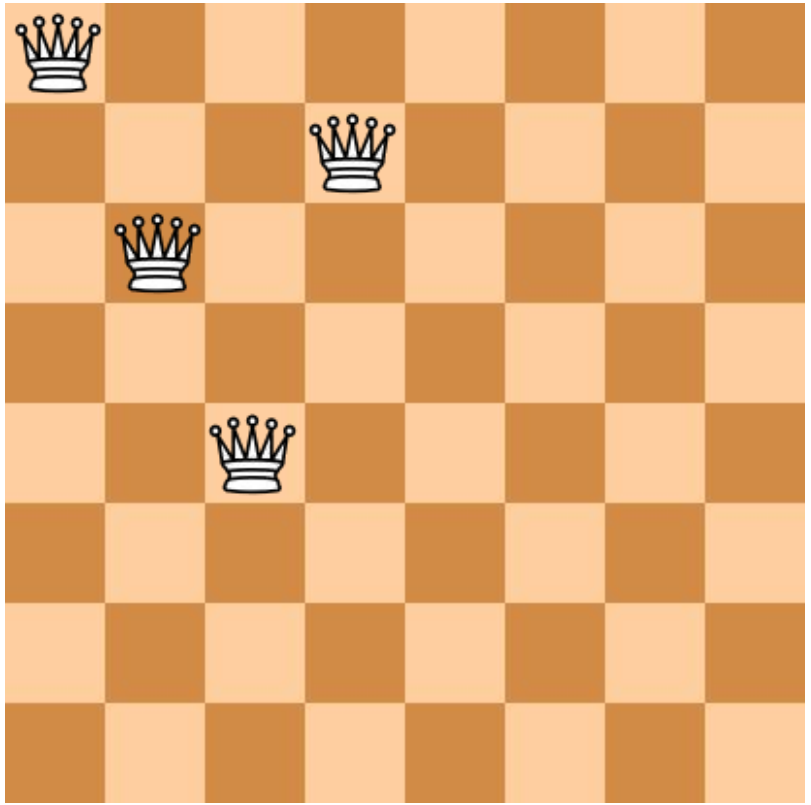
- a=[0, 2, 4, 1, 7]
- b=[0, 3, 6, 4, 11]
- c=[0, -1, -2, 2, -3]

- i=5

Esecuzione

- `a=[0, 2, 4, 1]`
- `b=[0, 3, 6, 4]`
- `c=[0, -1, -2, 2]`

- `i=4`



Esecuzione

- a=[0, 2, 4, 6]
- b=[0, 3, 6, 9]
- c=[0, -1, -2, -3]

- i=4