

# Algoritmi e Strutture Dati

## Programmazione dinamica – Parte 1

Alberto Montresor

Università di Trento

2021/03/02

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



# Sommario

- 1 Introduzione
- 2 Domino
- 3 Hateville
- 4 Zaino

# Tecniche di soluzione problemi

- Divide-et-impera
- Programmazione dinamica / memoization
- Tecnica greedy
- Ricerca locale
- Backtrack
- Algoritmi probabilistici
- Tecniche di soluzione per problemi intrattabili

# Programmazione dinamica in pillole

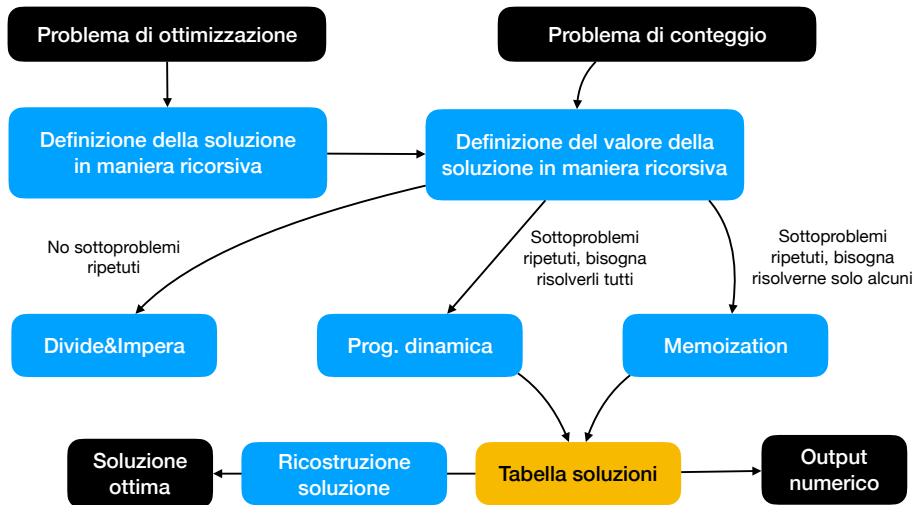
- Un metodo per spezzare un problema ricorsivamente in sottoproblemi
- Ogni sottoproblema viene risolto una volta sola
- La sua soluzione viene memorizzata in una tabella
- Nel caso un sottoproblema debba essere risolto nuovamente, si ottiene la sua soluzione dalla tabella
- La tabella è facilmente indirizzabile (lookup in  $O(1)$ )

Those who cannot remember the past  
are condemned to repeat it

---

*George Santayana, 1905*

# Approccio generale



## Un po' di storia

- Il termine **Dynamic Programming** è stato coniato da Richard Bellman agli inizi degli anni '50, nell'ambito dell'ottimizzazione matematica
- Inizialmente, si riferiva al processo di risolvere un problema compiendo le migliori decisioni una dopo l'altra.
- "Dynamic" doveva dare un senso "temporale"
- "Programming" si riferiva all'idea di creare "programmazioni ottime", per esempio nel campo della logistica

[https://en.wikipedia.org/wiki/Dynamic\\_programming#History](https://en.wikipedia.org/wiki/Dynamic_programming#History)

# Un po' di storia

## DYNAMIC

"IT'S IMPOSSIBLE TO USE THE WORD 'DYNAMIC' IN THE PEJORATIVE SENSE...THUS, I THOUGHT 'DYNAMIC PROGRAMMING' WAS A GOOD NAME."

- RICHARD BELLMAN, EXPLAINING HOW HE PICKED A NAME FOR HIS MATH RESEARCH TO TRY TO PROTECT IT FROM CRITICISM (EYE OF THE HURRICANE, 1984)

## ENTROPY

"YOU SHOULD CALL IT 'ENTROPY'... NO ONE KNOWS WHAT ENTROPY REALLY IS, SO IN A DEBATE YOU WILL ALWAYS HAVE THE ADVANTAGE."

- JOHN VON NEUMANN, TO CLAUDE SHANNON, ON WHY HE SHOULD BORROW THE PHYSICS TERM IN INFORMATION THEORY (AS TOLD TO MYRON TRIBUS)



**DYNAMIC ENTROPY**

SCIENCE TIP: IF YOU HAVE A COOL CONCEPT YOU NEED A NAME FOR, TRY "DYNAMIC ENTROPY."

<https://xkcd.com/2318/>

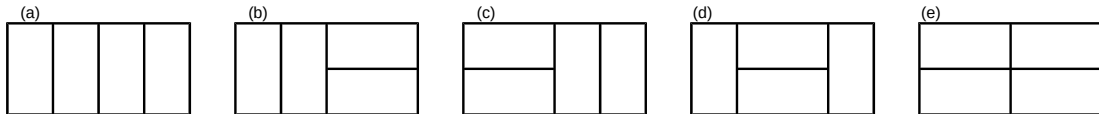
# Problema 1 – Domino lineare

## Definizione

Il gioco del domino è basato su tessere di dimensione  $2 \times 1$ . Scrivere un algoritmo efficiente che prenda in input un intero  $n$  e restituisca il numero di possibili disposizioni di  $n$  tessere in un rettangolo  $2 \times n$ .

## Esempio

I casi (a)-(e) della figura rappresentano le cinque disposizioni possibili con cui è possibile riempire un rettangolo  $2 \times 4$ .





# Domino

Come risolvereste il problema?

# Domino

## Definizione ricorrenza

Definiamo una formula ricorsiva  $DP[n]$  che permetta di calcolare il numero di disposizioni possibili quando si hanno  $n$  tessere.

- Quante disposizioni possibili se non ho tessere ( $n = 0$ )?
- Quante disposizioni possibili se ho solo una tessera ( $n = 1$ )?

$$DP[n] = \begin{cases} ? & n \leq 1 \\ ? & n > 1 \end{cases}$$

# Domino

## Definizione ricorrenza

Definiamo una formula ricorsiva  $DP[n]$  che permetta di calcolare il numero di disposizioni possibili quando si hanno  $n$  tessere.

- Con  $n = 0$ , esiste una sola disposizione possibile (nessuna tessera)
- Con  $n = 1$ , esiste una sola disposizione possibile (tessera verticale)

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ ? & n > 1 \end{cases}$$

# Domino

## Definizione ricorrenza

Definiamo una formula ricorsiva  $DP[n]$  che permetta di calcolare il numero di disposizioni possibili quando si hanno  $n$  tessere.

- Cosa succede se decido di mettere l'ultima tessera in verticale?
- Cosa succede se decido di mettere l'ultima tessera in orizzontale?

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ ? & n > 1 \end{cases}$$

# Domino

## Definizione ricorrenza

Definiamo una formula ricorsiva  $DP[n]$  che permetta di calcolare il numero di disposizioni possibili quando si hanno  $n$  tessere.

- Se metto una tessera in verticale, risolverò il problema di dimensione  $n - 1$
- Se metto una tessera in orizzontale, ne devo mettere due; risolverò il problema di dimensione  $n - 2$
- Queste due possibilità si sommano insieme (conteggio)

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ DP[n - 2] + DP[n - 1] & n > 1 \end{cases}$$

## Serie matematica

La serie generata è la seguente

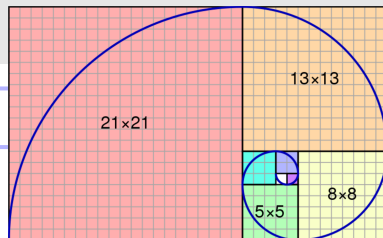
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Ricorda niente?

# Serie matematica

La serie generata è la seguente

1, 1, 2, 3, 5, 8, 13,  
21, 34, 55, 89, ...



## Successione di Fibonacci

$DP[n]$  è pari al  $(n+1)$ -esimo numero della serie di Fibonacci, introdotta da Leonardo Pisano detto il Fi'Bonacci (1175–1235).

- Definiti per descrivere la crescita di una popolazione di conigli (!)
- In natura: Pigne, conchiglie, parte centrale dei girasoli, etc.
- In informatica: Alberi AVL minimi, Heap di Fibonacci, etc.

# Domino - Algoritmo ricorsivo

Algoritmo ricorsivo che risolve il problema Domino

```
int domino1(int n)
{
    if  $n \leq 1$  then
        return 1
    else
        return domino1( $n - 1$ ) + domino1( $n - 2$ )
}
```

Qual è l'equazione di ricorrenza associata a `domino1()`?



# Complessità computazionale

Equazione di ricorrenza associata a `domino1()`

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

Qual è la complessità di `domino1()`?

# Complessità computazionale

Equazione di ricorrenza associata a `domino1()`

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

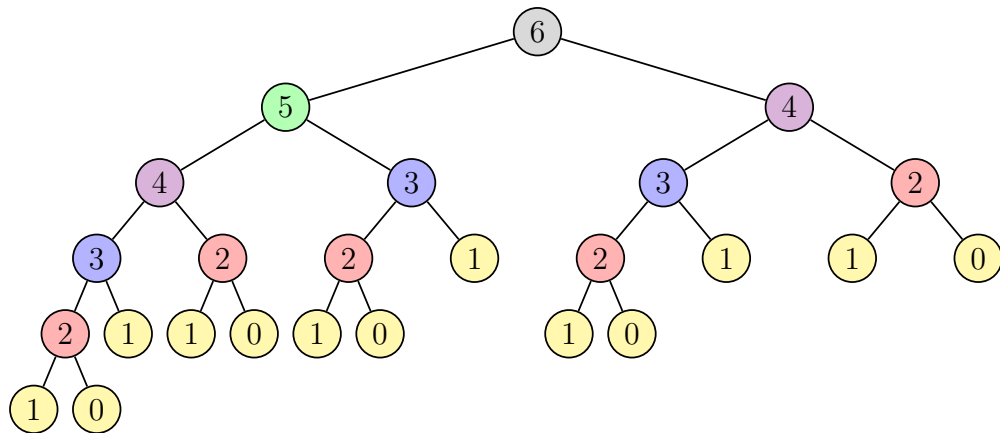
Qual è la complessità di `domino1()`?

Ricorrenza lineare di ordine costante:

- $a_1 = 1, a_2 = 1, a = a_1 + a_2 = 2, \beta = 0$
- Complessità:  $\Theta(a^n \cdot n^\beta)$

$$T(n) = \Theta(2^n)$$

# Albero di ricorsione di domino1()



Molti sotto-problemi ripetuti!

# Come evitare di risolvere un problema più di una volta

## Tabella DP

- Memorizziamo il risultato ottenuto risolvendo un particolare problema in una **tabella DP** (vettore, matrice, dizionario)
- La tabella deve contenere un elemento per ogni sottoproblema che dobbiamo risolvere

## Casi base

- Memorizziamo i casi base direttamente nelle posizioni relative

## Iterazione bottom-up

- Si parte dai problemi risolubili come basi base
- Si sale verso problemi via via più grandi ...
- ... fino a raggiungere il problema originale

# Domino: algoritmo iterativo

Algoritmo iterativo che risolve il problema Domino

---

```
int domino2(int n)
```

---

```
    DP = new int[0...n]
```

```
    DP[0] = DP[1] = 1
```

```
    for i = 2 to n do
```

```
        DP[i] = DP[i - 1] + DP[i - 2]
```

```
    return DP[n]
```

---

$n$	0	1	2	3	4	5	6	7
$DP[]$	1	1	2	3	5	8	13	21

# Domino

---

```
int domino2(int  $n$ )
```

---

```
 $DP = \text{new int}[0 \dots n]$ 
```

```
 $DP[0] = DP[1] = 1$ 
```

```
for  $i = 2$  to  $n$  do
```

```
     $DP[i] = DP[i - 1] + DP[i - 2]$ 
```

```
return  $DP[n]$ 
```

---

Qual è la complessità in **tempo** di  $\text{domino2}(n)$ ?

# Domino

---

```
int domino2(int n)
```

---

```
DP = new int[0...n]
```

```
DP[0] = DP[1] = 1
```

```
for i = 2 to n do
```

```
    DP[i] = DP[i - 1] + DP[i - 2]
```

```
return DP[n]
```

---

Qual è la complessità in **tempo** di `domino2(n)`?

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

Qual è la complessità in **spazio** di `domino2(n)`?

# Domino

---

```
int domino2(int n)
```

---

```
DP = new int[0...n]
```

```
DP[0] = DP[1] = 1
```

```
for i = 2 to n do
```

```
    DP[i] = DP[i - 1] + DP[i - 2]
```

```
return DP[n]
```

---

Qual è la complessità in **tempo** di `domino2(n)`?

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

Qual è la complessità in **spazio** di `domino2(n)`?

$$S(n) = \Theta(n)$$

Possiamo fare "**meglio di così**"?



# Domino

---

```
int domino2(int n)
```

---

```
DP = new int[0...n]
```

```
DP[0] = DP[1] = 1
```

```
for i = 2 to n do
```

```
    DP[i] = DP[i - 1] + DP[i - 2]
```

```
return DP[n]
```

---

Qual è la complessità in **tempo** di `domino2(n)`?

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

Qual è la complessità in **spazio** di `domino2(n)`?

$$S(n) = \Theta(n)$$

Possiamo fare "**meglio di così**"?

Possiamo ridurre lo spazio utilizzato

# Domino

---

```
int domino3(int n)
```

---

```
int DP0 = 1
```

```
int DP1 = 1
```

```
int DP2 = 1
```

```
for i = 2 to n do
```

```
    DP0 = DP1
```

```
    DP1 = DP2
```

```
    DP2 = DP0 + DP1
```

```
return DP2
```

---

$n$	0	1	2	3	4	5	6	7
$DP_0$	-	-	1	1	2	3	5	8
$DP_1$	1	1	1	2	3	5	8	13
$DP_2$	1	1	2	3	5	8	13	21

Qual è la complessità in **spazio** di domino3( $n$ )?

## Domino

---

```
int domino3(int n)
```

---

```
int DP0 = 1
```

```
int DP1 = 1
```

```
int DP2 = 1
```

```
for i = 2 to n do
```

```
    DP0 = DP1
```

```
    DP1 = DP2
```

```
    DP2 = DP0 + DP1
```

```
return DP2
```

---

$n$	0	1	2	3	4	5	6	7
$DP_0$	-	-	1	1	2	3	5	8
$DP_1$	1	1	1	2	3	5	8	13
$DP_2$	1	1	2	3	5	8	13	21

Qual è la complessità in **spazio** di domino3( $n$ )?

$$S(n) = \Theta(1)$$

## Ripasso sulla complessità computazionale

Siete sicuri che i calcoli sulla complessità siano corretti?

## Ripasso sulla complessità computazionale

Siete sicuri che i calcoli sulla complessità siano corretti?

Osservate di nuovo la serie generata

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Quanti bit sono necessari per memorizzare  $F(n)$ ?

# Modello costo uniforme vs modello costo logaritmico

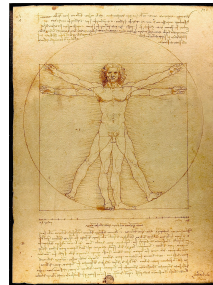
## Formula di Binet per i numeri di Fibonacci

$$DP[n-1] = F(n) = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}}$$

dove  $\phi$  è la **sezione aurea**:

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,6180339887 \dots$$

$$\frac{1}{\phi} = \phi - 1 = \frac{2}{1 + \sqrt{5}} = 0,6180339887 \dots$$



Quanti bit sono richiesti per memorizzare  $F(n)$ ?

Quanto costa sommare due numeri di Fibonacci consecutivi?

# Modello costo uniforme vs modello costo logaritmico

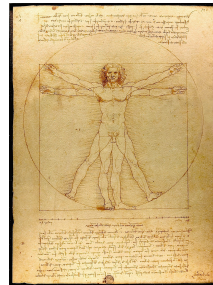
## Formula di Binet per i numeri di Fibonacci

$$DP[n-1] = F(n) = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}}$$

dove  $\phi$  è la **sezione aurea**:

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,6180339887\dots$$

$$\frac{1}{\phi} = \phi - 1 = \frac{2}{1 + \sqrt{5}} = 0,6180339887\dots$$



Quanti bit sono richiesti per memorizzare  $F(n)$ ?

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

Quanto costa sommare due numeri di Fibonacci consecutivi?

## Modello costo uniforme vs modello costo logaritmico

Sotto il modello di costo logaritmico, le tre versioni hanno la seguente complessità:

Funzione	Complessità (Tempo)	Complessità (Spazio)
domino1()	$O(n2^n)$	$O(n^2)$
domino2()	$O(n^2)$	$O(n^2)$
domino3()	$O(n^2)$	$O(n)$

Si può fare meglio di così utilizzando l'esponenziazione di matrici basata su quadrati:

<https://brilliant.org/wiki/fast-fibonacci-transform/>



# Hateville

- Hateville è un villaggio particolare, composto da  $n$  case, numerate da 1 a  $n$  lungo una singola strada.
- Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati.
- Quindi, il vicino  $i$  odia i vicini  $i - 1$  e  $i + 1$  (se esistenti).
- Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi.
- Ogni abitante  $i$  ha intenzione di donare una quantità  $D[i]$ , ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

# Hateville

## Problemi

- Scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta
- Scrivere un algoritmo che restituisca il sottoinsieme di indici  $S \subseteq \{1, \dots, n\}$  tale per cui la donazione totale  $T = \sum_{i \in S} D[i]$  è massimale

## Esempio

- Vettore donazioni:  $D = [4, 3, 6, 5]$
- Raccolta fondi massima: 10
- Insieme indici:  $\{1, 3\}$

# Hateville

Come risolvereste il problema?

# Hateville

## Problemi

- Scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta
- Scrivere un algoritmo che restituisca il sottoinsieme di indici  $S \subseteq \{1, \dots, n\}$  tale per cui la donazione totale  $T = \sum_{i \in S} D[i]$  è massimale

## La vostra soluzione funziona con questo esempio?

- Vettore donazioni:  $D = [10, 5, 5, 10]$
- Raccolta fondi massima: 20
- Insieme indici:  $\{1, 4\}$

# Definizione ricorsiva

## Definizione ricorrenza

È possibile definire una formula ricorsiva che permetta di calcolare il sottoinsieme di case che, se selezionate, dà origine alla maggior quantità di donazioni?

Ri-definiamo il problema

- Sia  $HV(i)$  uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale dalle prime  $i$  case di Hateville, numerate  $1 \dots n$
- $HV(n)$  è la soluzione del problema originale

## Passo ricorsivo

Considerate il vicino  $i$ -esimo

- Cosa succede se non accetto la sua donazione?

$$HV(i) =$$

## Passo ricorsivo

Considerate il vicino  $i$ -esimo

- Cosa succede se non accetto la sua donazione?

$$HV(i) = HV(i - 1)$$

- Cosa succede se accetto la sua donazione?

$$HV(i) =$$

## Passo ricorsivo

Considerate il vicino  $i$ -esimo

- Cosa succede se non accetto la sua donazione?

$$HV(i) = HV(i - 1)$$

- Cosa succede se accetto la sua donazione?

$$HV(i) = \{i\} \cup HV(i - 2)$$

- Come faccio a decidere se accettare o meno?

$$HV(i) =$$



## Passo ricorsivo

Considerate il vicino  $i$ -esimo

- Cosa succede se non accetto la sua donazione?

$$HV(i) = HV(i - 1)$$

- Cosa succede se accetto la sua donazione?

$$HV(i) = \{i\} \cup HV(i - 2)$$

- Come faccio a decidere se accettare o meno?

$$HV(i) = \text{highest}(HV(i - 1), \{i\} \cup HV(i - 2))$$

## Sottostruttura ottima

Vi ho convinti?

$$HV(i) = \textit{highest}(HV(i-1), \{i\} \cup HV(i-2))$$

## Sottostruttura ottima

Vi ho convinti?

$$HV(i) = \text{highest}(HV(i-1), \{i\} \cup HV(i-2))$$

### Teorema: Sottostruttura ottima

- Sia  $P_i$  il problema dato dalle prime  $i$  case
- Sia  $S_i$  una soluzione ottima per il problema  $P_i$
- Ne consegue:
  - Se  $i \notin S_i$ , allora  $S_i = S_{i-1}$
  - Se  $i \in S_i$ , allora  $S_i = S_{i-2} \cup \{i\}$

## Sottostruttura ottima – Dimostrazione

- Sia  $P_i$  il problema dato dalle prime  $i$  case
- Sia  $S_i$  una soluzione ottima per il problema  $P_i$
- Sia  $|S_i|$  il totale di donazioni di  $S_i$

Caso 1:  $i \notin S_i$

- $S_i$  è una soluzione ottima anche per  $P_{i-1}$
- Se così non fosse, esisterebbe una soluzione  $S'_{i-1}$  per il problema  $P_{i-1}$  tale che  $|S'_{i-1}| > |S_i|$
- Ma allora  $S'_{i-1}$  sarebbe una soluzione anche per  $P_i$  tale che  $|S'_{i-1}| > |S_i|$ , assurdo

## Sottostruttura ottima – Dimostrazione

- Sia  $P_i$  il problema dato dalle prime  $i$  case
- Sia  $S_i$  una soluzione ottima per il problema  $P_i$
- Sia  $|S_i|$  il totale di donazioni di  $S_i$

Caso 2:  $i \in S_i$

- $i - 1 \notin S_i$ , altrimenti non sarebbe una soluzione ammissibile
- Quindi,  $S_i - \{i\}$  è una soluzione ottima per  $P_{i-2}$
- Se così non fosse, esisterebbe una soluzione  $S'_{i-2}$  per il problema  $P_{i-2}$  tale che  $|S'_{i-2}| > |S_i - \{i\}|$
- Ma allora  $S'_{i-2} \cup \{i\}$  sarebbe una soluzione per  $P_i$  tale che  $|S'_{i-2} \cup \{i\}| > |S_i|$ , assurdo

## Completare la ricorsione

Quali sono i casi base?

## Completare la ricorsione

Quali sono i casi base?

- $HV(0) = \emptyset$
- $HV(1) = \{1\}$

Tutto insieme!

$$HV(i) = \begin{cases} \emptyset & i = 0 \\ \{1\} & i = 1 \\ \text{highest}(HV(i-1), HV(i-2) \cup \{i\}) & i \geq 2 \end{cases}$$

# Algoritmo ricorsivo

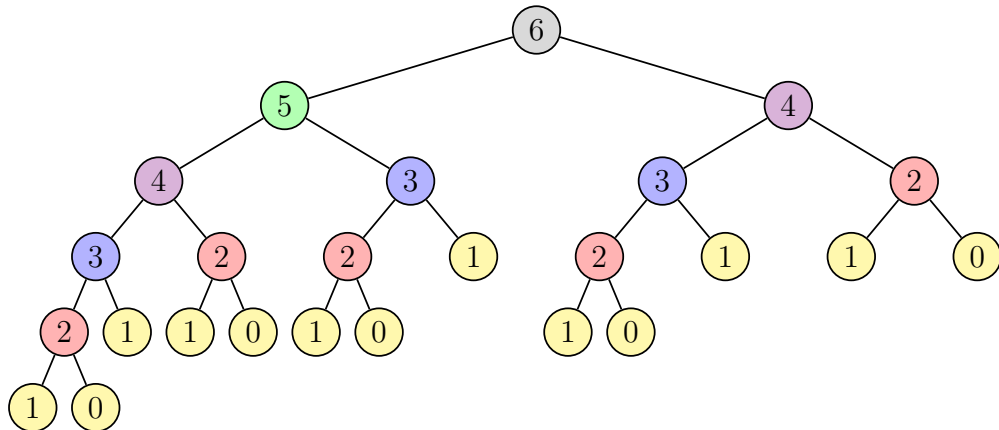
## Domanda

Vale la pena scrivere un algoritmo ricorsivo, basato su divide-et-impera, per risolvere il problema di Hateville?

$$HV(i) = \begin{cases} \emptyset & i = 0 \\ \{1\} & i = 1 \\ \text{highest}(HV(i-1), HV(i-2) \cup \{i\}) & i \geq 2 \end{cases}$$



Vale la pena scrivere un algoritmo ricorsivo, basato su divide-et-impera, per risolvere il problema di Hateville?



# Memorizzare una tabella

## Esempi

$i$	0	1	2	3	4	5	6	7
$D$		10	5	5	8	4	7	12
$HV$	$\emptyset$	$\{1\}$	$\{1\}$	$\{1, 3\}$	$\{1, 4\}$	$\{1, 3, 5\}$	$\{1, 4, 6\}$	$\{1, 3, 5, 7\}$

$i$	0	1	2	3	4	5	6	7
$D$		10	1	1	10	1	1	10
$HV$	$\emptyset$	$\{1\}$	$\{1\}$	$\{1, 3\}$	$\{1, 4\}$	$\{1, 4\}$	$\{1, 4, 6\}$	$\{1, 4, 7\}$

## Problemi

- Dobbiamo definire la funzione *highest()*
- Memorizzare gli insiemi nella tabella è costoso

# Tabella DP

## Valore della soluzione ottima

- Sia  $DP[i]$  il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime  $i$  case di Hateville.
- $DP[n]$  è il valore della soluzione ottima

$$DP[i] = \begin{cases} ? & i = 0 \\ ? & i = 1 \\ ? & i \geq 2 \end{cases}$$

# Tabella DP

## Valore della soluzione ottima

- Sia  $DP[i]$  il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime  $i$  case di Hateville.
- $DP[n]$  è il valore della soluzione ottima

$$DP[i] = \begin{cases} 0 & i = 0 \\ ? & i = 1 \\ ? & i \geq 2 \end{cases}$$

# Tabella DP

## Valore della soluzione ottima

- Sia  $DP[i]$  il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime  $i$  case di Hateville.
- $DP[n]$  è il valore della soluzione ottima

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ ? & i \geq 2 \end{cases}$$

# Tabella DP

## Valore della soluzione ottima

- Sia  $DP[i]$  il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime  $i$  case di Hateville.
- $DP[n]$  è il valore della soluzione ottima

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ \max(DP[i-1], DP[i-2] + D[i]) & i \geq 2 \end{cases}$$

# Hateville: Algoritmo iterativo

Algoritmo iterativo che risolve il problema Hateville

# Hateville: Algoritmo iterativo

Algoritmo iterativo che risolve il problema Hateville

---

```
int hateville(int[] D, int n)  


---

int[] DP = new int[0...n]  
DP[0] = 0  
DP[1] = D[1]  
for i = 2 to n do  
     $DP[i] = \max(DP[i - 1], DP[i - 2] + D[i])$   
return DP[n]
```

---



# Sulla risoluzione con "veri" linguaggi di programmazione

## Java

```
public int hateville(int[] D, int n) {  
    int[] DP = new int[n+1];  
    DP[0] = 0;  
    DP[1] = D[0];  
    for (int i=2; i <= n; i++) {  
        DP[i] = max(DP[i-1], DP[i-2]+D[i-1]);  
    }  
    return DP[n];  
}
```

## Python

```
def hateville(D):  
    DP = [ 0, D[0] ]  
    for i in range(1, len(D)):  
        DP.append( max(DP[-1], DP[-2] + D[i]) )  
    return DP[-1]
```

# Memorizzare una tabella

## Esempi

$i$	0	1	2	3	4	5	6	7
$D$		10	5	5	8	4	7	12
$DP$	0	10	10	15	18	19	25	31

$i$	0	1	2	3	4	5	6	7
$D$		10	1	1	10	1	1	10
$DP$	0	10	10	11	20	20	21	30

## Problema

Abbiamo il valore della soluzione massimale, ma non abbiamo la soluzione!

## Ricostruire la soluzione originale

Si guarda l'elemento  $DP[i]$ . Da cosa deriva il suo valore?

- Se  $DP[i] = DP[i - 1]$ , la casa  $i$  **non è stata selezionata**
- Se  $DP[i] = DP[i - 2] + D[i]$ , la casa  $i$  **è stata selezionata**
- Se entrambe le equazioni sono vere, una vale l'altra!

Per ricostruire la soluzione fino ad  $i$ , lavoriamo in modo ricorsivo:

- Se  $DP[i] = DP[i - 1]$ , si prende la soluzione fino a  $i - 1$  **senza aggiungere nulla**
- **Altrimenti**, si prende la soluzione fino a  $i - 2$  e **si aggiunge  $i$**

## Ricostruire la soluzione originale

---

```
SET solution(int[] DP, int[] D, int i)
```

---

```
if i == 0 then
```

```
    return ∅
```

```
else if i == 1 then
```

```
    return {1}
```

```
else if DP[i] == DP[i - 1] then
```

```
    return solution(DP, D, i - 1)
```

```
else
```

```
    SET sol = solution(DP, D, i - 2)
```

```
    sol.insert(i)
```

```
    return sol
```

---



---

```
SET hateville(int[] D, int n)
```

---

```
[...]
```

```
return solution(DP, D, n)
```

---

Note: per come è stato costruito il codice,  $D$  non è necessario

## Complessità computazionale

Qual è la complessità computazionale di `solution()`?

Qual è la complessità computazionale e spaziale di `hateville()`?

È possibile migliorare la complessità spaziale di `hateville()`?

## Complessità computazionale

Qual è la complessità computazionale di `solution()`?

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

Qual è la complessità computazionale e spaziale di `hateville()`?

È possibile migliorare la complessità spaziale di `hateville()`?

# Complessità computazionale

Qual è la complessità computazionale di `solution()`?

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

Qual è la complessità computazionale e spaziale di `hateville()`?

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

È possibile migliorare la complessità spaziale di `hateville()`?

## Complessità computazionale

Qual è la complessità computazionale di `solution()`?

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

Qual è la complessità computazionale e spaziale di `hateville()`?

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

È possibile migliorare la complessità spaziale di `hateville()`?

No, se vogliamo ricostruire la soluzione.

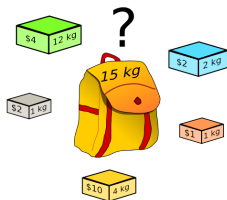


# Zaino (Knapsack)

## Descrizione del problema

Dato un insieme di oggetti, ognuno caratterizzato da un **peso** e un **profitto**, e uno "zaino" con un limite di capacità, individuare un sottoinsieme di oggetti

- il cui peso sia inferiore alla capacità dello zaino;
- il valore totale degli oggetti sia massimale, i.e. più alto o uguale al valore di qualunque altro sottoinsieme di oggetti



[https://en.wikipedia.org/wiki/Knapsack\\_problem#/media/File:Knapsack.svg](https://en.wikipedia.org/wiki/Knapsack_problem#/media/File:Knapsack.svg)

# Un po' di storia

- Problemi simili allo zaino erano trattati già intorno al 1896  
(G. Mathews. *On the partition of numbers*. Proceedings of the London Mathematical Society, 1(1):486–490, 1896)
- Il nome Knapsack è stato introdotto da Dantzig nel 1957.  
(George B. Dantzig. *Discrete-variable extremum problems*. Operations research, 5(2):266–288, 1957)

less than 128. By continuing in this manner, we can eventually arrive at a situation where it is not possible to find any directed link which leads to an improvement of the distance shown in any circle. If so, we have arrived at an optimal solution. For the example at hand, the optimal tree is the same as that shown in Fig. 2 except that the arrow from Washington to Boston is dropped and one from Chicago to Boston is inserted. The 191 at Boston is changed to 186. The values of the  $x_{ij}$  are unity along the path in the final tree from Los Angeles to Boston and are zero elsewhere. Hence the optimal path is from Los Angeles to Salt Lake City, then to Chicago, and finally to Boston.

## 5. THE KNAPSACK PROBLEM

In certain types of problems, we can get extreme-point solutions for which not all the values of the  $x_{ij}$  are either zero or one. When any of the  $x_{ij}$  have fractional values, the corresponding extreme points are referred to as fractional extreme points. Now an example of this occurs in the knapsack problem. In this problem a person is planning a hike and has decided not to carry more than 70 lb of different items, such as bed roll, geiger counters (these days), cans of food, etc.

We try to formulate this in mathematical terms. Let  $a_j$  be the weight of the  $j^{\text{th}}$  object and let  $b_j$  be its relative value determined by the hiker in comparison with the values of the other objects he would like to have on his trip.

# Zaino (Knapsack)

## Input

- Vettore  $w$ , dove  $w[i]$  è il **peso** (**weight**) dell'oggetto  $i$ -esimo
- Vettore  $p$ , dove  $p[i]$  è il **profitto** (**profit**) dell'oggetto  $i$ -esimo
- La **capacità**  $C$  dello zaino

## Output

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

Il **volume totale** deve essere  
minore o uguale alla capacità

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

Il **profitto totale** deve essere  
massimizzato

$$\operatorname{argmax}_S p(S) = \sum_{i \in S} p[i]$$

# Esempi

Quali sono gli oggetti migliori per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	6	12

$$C = 12$$

## Esempi

Quali sono gli oggetti migliori per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	6	12

$$C = 12$$

$$S = \{1\}$$

Come risolvereste il problema?

# Esempi

Quali sono gli oggetti migliori per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	6	12

$$C = 12$$

$$S = \{1\}$$

Come risolvereste il problema?

Il vostro algoritmo funziona per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	7	15

$$C = 12$$

# Esempi

Quali sono gli oggetti migliori per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	6	12

$$C = 12$$

$$S = \{1\}$$

Come risolvereste il problema?

Il vostro algoritmo funziona per questo esempio?

Item id	1	2	3
Weight	10	4	8
Profit	20	7	15

$$C = 12$$

$$S = \{2, 3\}$$

## Definizione matematica del valore della soluzione

### Valore della soluzione

Dato uno zaino di capacità  $C$  e  $n$  oggetti caratterizzati da peso  $w$  e profitto  $p$ , definiamo  $DP[i][c]$  come il massimo profitto che può essere ottenuto dai primi  $i \leq n$  oggetti contenuti in uno zaino di capacità  $c \leq C$ .

### Problema originale

Il massimo profitto ottenibile dal problema originale è rappresentato da  $DP[n][C]$ .



## Parte ricorsiva

Considerate l'ultimo oggetto

Cosa succede se  $DP[i][c] =$   
non lo prendete?

## Parte ricorsiva

Considerate l'ultimo oggetto

Cosa succede se  
non lo prendete?  $DP[i][c] =$   
 $DP[i - 1][c]$

La capacità non  
cambia, non c'è profitto

Cosa succede se  
lo prendete?  $DP[i][c] =$

## Parte ricorsiva

Considerate l'ultimo oggetto

Cosa succede se  
non lo prendete?  $DP[i][c] =$   
 $DP[i - 1][c]$

La capacità non  
cambia, non c'è profitto

Cosa succede se  
lo prendete?  $DP[i][c] =$   
 $DP[i - 1][c - w[i]] + p[i]$

Sottraete il peso dalla  
capacità e aggiungete il  
profitto relativo

Come scegliere la soluzione migliore?

$$DP[i][c] =$$

## Parte ricorsiva

Considerate l'ultimo oggetto

Cosa succede se  
non lo prendete?  $DP[i][c] =$   
 $DP[i-1][c]$

La capacità non  
cambia, non c'è profitto

Cosa succede se  
lo prendete?  $DP[i][c] =$   
 $DP[i-1][c-w[i]] + p[i]$

Sottraete il peso dalla  
capacità e aggiungete il  
profitto relativo

Come scegliere la soluzione migliore?

$$DP[i][c] = \max(\overbrace{DP[i-1][c-w[i]] + p[i]}^{\text{Preso}}, \overbrace{DP[i-1][c]}^{\text{Non preso}})$$

## Casi base

Quali sono i casi base?

## Casi base

Quali sono i casi base?

- Qual è il profitto massimo se non avete più oggetti?
- Qual è il profitto massimo se non avete più capacità?
- Cosa succede se la capacità è negativa?

# Casi base

Quali sono i casi base?

- Qual è il profitto massimo se non avete più oggetti?
- Qual è il profitto massimo se non avete più capacità?
- Cosa succede se la capacità è negativa?

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

## Formula completa

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

Come trasformare questa formula in un algoritmo?



## Zaino

---

```
int knapsack(int[]  $w$ , int[]  $p$ , int  $n$ , int  $C$ )
```

---

```
 $DP = \text{new int}[0 \dots n][0 \dots C]$ 
```

```
for  $i = 0$  to  $n$  do
```

```
└  $DP[i][0] = 0$ 
```

```
for  $c = 0$  to  $C$  do
```

```
└  $DP[0][c] = 0$ 
```

```
for  $i = 1$  to  $n$  do
```

```
└ for  $c = 1$  to  $C$  do
```

```
└└ if  $w[i] \leq c$  then
```

```
└└└  $DP[i][c] = \max(\overbrace{DP[i-1][c-w[i]] + p[i]}^{\text{Preso}}, \overbrace{DP[i-1][c]}^{\text{Non preso}})$ 
```

```
└└ else
```

```
└└└  $DP[i][c] = \overbrace{DP[i-1][c]}^{\text{Non preso}}$ 
```

```
return  $DP[n][C]$ 
```

# 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

# Complessità computazionale

Qual è la complessità della funzione `knapsack()`?

```
int knapsack(int[] w, int[] p, int n, int C)
DP = new int[0...n][0...C]
for i = 0 to n do
    DP[i][0] = 0
for c = 0 to C do
    DP[0][c] = 0
for i = 1 to n do
    for c = 1 to C do
        if w[i] ≤ c then
            DP[i][c] = max(DP[i-1][c-w[i]] + p[i], DP[i-1][c])
        else
            DP[i][c] = DP[i-1][c]
return DP[n][C]
```

# Complessità computazionale

Qual è la complessità della funzione `knapsack()`?

$$T(n) = O(nC)$$

È un algoritmo polinomiale?

# Complessità computazionale

Qual è la complessità della funzione `knapsack()`?

$$T(n) = O(nC)$$

È un algoritmo polinomiale?

No, è un algoritmo **pseudo-polinomiale**, perchè sono necessari  $k = \log C$  bit per rappresentare  $C$  e quindi la complessità è:

$$T(n) = O(n2^k)$$

## Reality check

Sebbene il problema di base sia molto semplice, assieme alle sue varianti può essere utilizzato in una miriade di applicazioni.

- Taglio dei materiali (con estensioni nel campo 2D-3D)
- Logistica
- Selezione di portfolio di investimenti
- Selezione di reti cellulari per nodi mobili
- Campionamento adattivo a densità variabile
- ....