Algoritmi e Strutture Dati

Programmazione dinamica – Parte 1

Alberto Montresor

Università di Trento

2019/02/20

Sommario

Tecniche di soluzione problemi

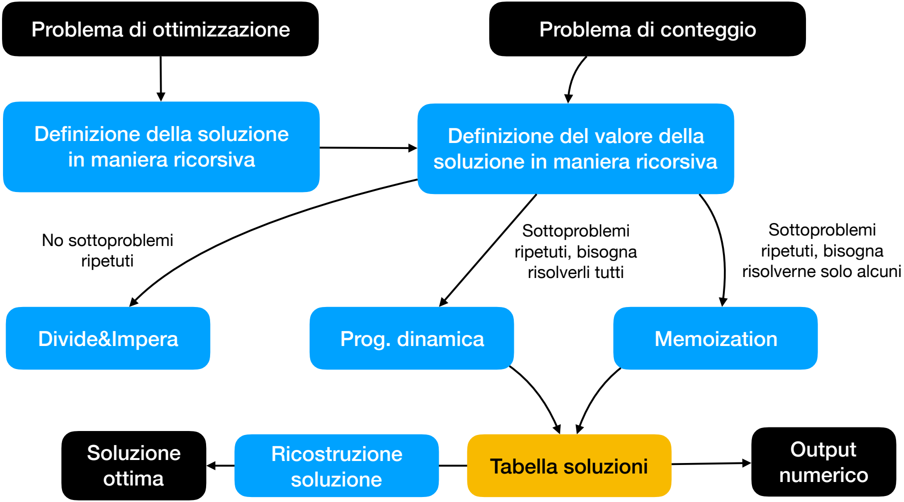
- Divide-et-impera
- Programmazione dinamica / memoization
- Tecnica greedy
- Ricerca locale
- Backtrack
- Algoritmi probabilistici
- Tecniche di approssimazione

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`

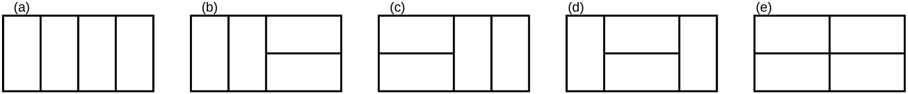
Problema 1 – Domino lineare

Definizione

Il gioco del domino è basato su tessere di dimensione 2×1. 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 2×4.



Domino

Definizione ricorrenza

Definiamo una formula ricorsiva $DP[n]$ che ci 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

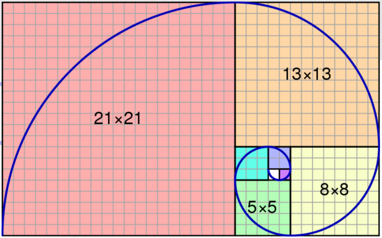
Definizione ricorrenza

Definiamo una formula ricorsiva $DP[n]$ che ci 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, ...`

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()

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

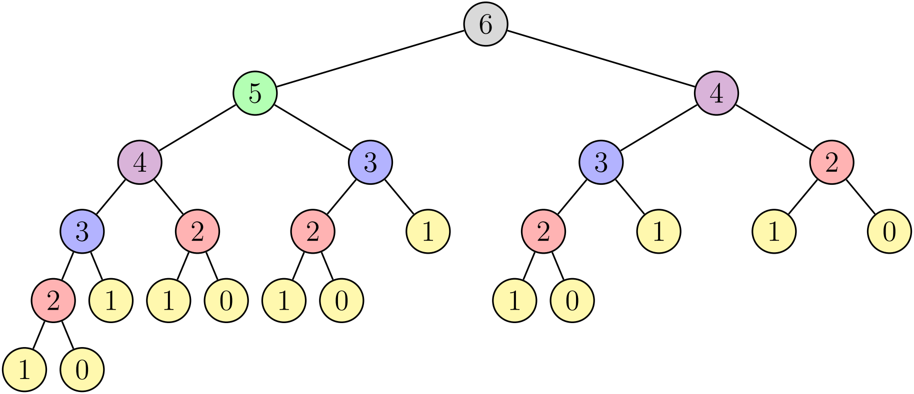
Qual è la complessità di domino1()?

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

- Quando risolviamo un problema, memorizziamo il risultato che otteniamo in una tabella *DP* (vettore, matrice, dizionario, etc)
- La tabella deve contenere un elemento per ogni sottoproblema che dobbiamo risolvere

Casi base

- Memorizziamo i casi base direttamente nelle posizioni relative

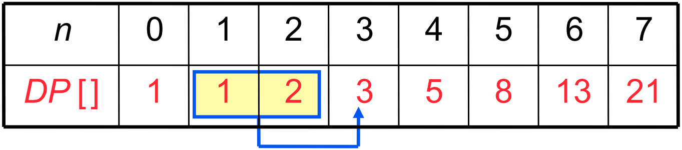
Iterazione bottom-up

- Si parte da quelli che possono essere risolti a partire dai casi base
- Si sale verso problemi via via più grandi ...

Domino: algoritmo iterativo

Algoritmo iterativo che risolve il problema Domino

```
int domino2(int $n$)
$DP$ = new int$[0 \ldots n]$
$DP[0] = DP[1] = 1$
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 $n$)
$DP$ = new int$[0 \ldots n]$
$DP[0] = DP[1] = 1$
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 $DP_0 = 1$
int $DP_1 = 1$
int $DP_2 = 1$
for $i = 2$ to $n$ do
    $DP_0 = DP_1$
    $DP_1 = DP_2$
    $DP_2 = DP_0 + DP_1$
return $DP_2$
```

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?

Osservate di nuovo la serie generata

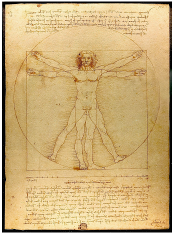
`1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...`

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 ϕ è la sezione aurea:

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

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

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 $S \subseteq \{1, \ldots, n\}$ 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 $S \subseteq \{1, \ldots, n\}$ 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

E' possibile definire una formula ricorsiva che ci 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 $1 \ldots n$
- $HV(n)$ è la soluzione del problema originale

Passo ricorsivo

Considerate il vicino i-esimo

- Cosa succede se non accetto la sua donazione?

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

- Cosa succede se accetto la sua donazione?

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

- Come faccio a decidere se accettare o meno?

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

Sottostruttura ottima

Vi ho convinti?

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

Sottostruttura ottima

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

Sottostruttura ottima – Dimostrazione

- Sia $HV_p(i)$ il problema dato dalle prime i case
- Sia $HV_s(i)$ una soluzione ottima per il problema $HV_p(i)$
- Sia $|HV_s(i)|$ il totale di donazioni di $HV_s(i)$

Caso 1: $i \notin HV_s(i)$

- $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$
- Se così non fosse, esisterebbe una soluzione $HV_s'(i-1)$ per il problema $HV_p(i-1)$ tale che $|HV_s'(i-1)| > |HV_s(i)|$
- Ma allora $HV_s'(i-1)$ sarebbe una soluzione per $HV_p(i)$ tale che $|HV_s'(i-1)| > |HV_s(i)|$, assurdo

Sottostruttura ottima – Dimostrazione

- Sia $HV_p(i)$ il problema dato dalle prime i case
- Sia $HV_s(i)$ una soluzione ottima per il problema $HV_p(i)$
- Sia $|HV_s(i)|$ il totale di donazioni di $HV_s(i)$

Caso 2: $i \in HV_s(i)$

- $i - 1 \notin HV_s(i)$, altrimenti non sarebbe una soluzione ammissibile
- Quindi, $HV_s(i) - \{i\}$ è una soluzione ottima per $HV_p(i-2)$
- Se così non fosse, esisterebbe una soluzione $HV'_s(i-2)$ per il problema $HV_p(i-2)$ tale che $|HV'_s(i-2)| > |HV_s(i) - \{i\}|$
- Ma allora $HV'_s(i-2) \cup \{i\}$ sarebbe una soluzione per $HV_p(i)$ tale che $|HV'_s(i-2) \cup \{i\}| > |HV_s(i)|$, assurdo

Completare la ricorsione

Quali sono i casi base?

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

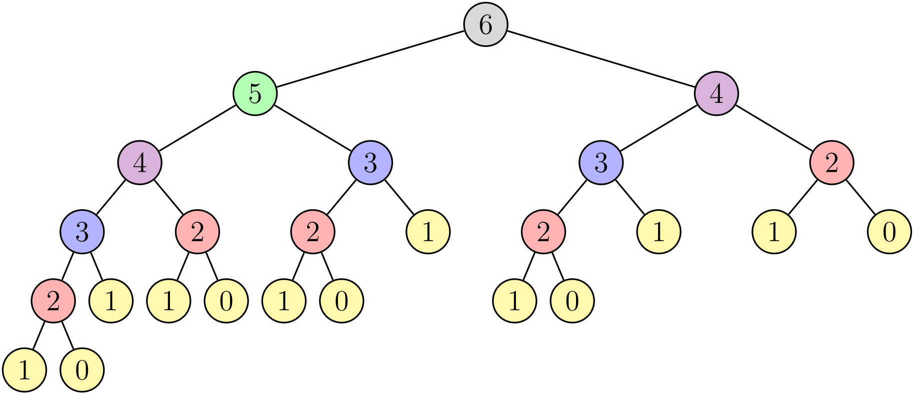
Tutto insieme!

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

Algoritmo ricorsivo

Domanda

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



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

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

Hateville: Algoritmo iterativo

Algoritmo iterativo che risolve il problema Hateville

```
hateville(int[] $D$, int $n$)
int[] $DP$ = new int[$0 \dots n$]
$DP[0] = 0$
$DP[1] = D[1]$
for $i = 2$ to $n$ do
    $DP[i] = \mathsf{max}(DP[i-1], DP[i-2] + D[i])$
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

Approccio

- 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!
- Utilizziamo questa informazione per ricostruire la soluzione in modo ricorsivo:
  - Per ricostruire la soluzione fino ad i, ricostriamo la soluzione fino ad $i-2$ e aggiungiamo i
  - Oppure, ricostriuamo la soluzione fino ad $i-1$ senza aggiungere nulla

Ricostruire la soluzione originale

```
SET solution(int[] $DP$, int[] $D$, int $i$)
if $i == 0$ then
    return $\emptyset$
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 $sol$
```

```
hateville(int[] $D$, int $n$)
[...]
return solution($DP, D, n$)
```

Complessità computazionale

Qual è la complessità computazionale di solution()?

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

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

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

E' possibile migliorare la complessità spaziale di hateville()?

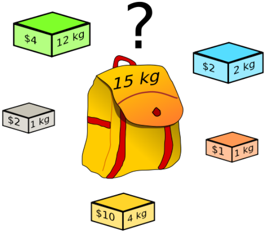
No, se vogliamo ricostruire la soluzione.

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



Zaino

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 $S \subseteq \{1, \ldots, n\}$ 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 $p(S) = \sum_{i \in S} p[i]$

Esempi

Quali sono gli oggetti migliori per questo esempio?

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

$C = 12$

$S = \{1\}$

Progettare un algoritmo che risolve il problema dello zaino

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 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?

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

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?

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

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 $C$)
$DP$ = 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
            $DP[i][c] = \mathsf{max}(\overbrace{DP[i-1][c-w[i]] + p[i]}^{\text{Taken}}, \overbrace{DP[i-1][c]}^{\text{Not taken}})$
        else
            $DP[i][c] = \overbrace{DP[i-1][c]}^{\text{Not taken}}$
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()`?

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

E' un algoritmo polinomiale?

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

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

Zaino ricorsivo

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

```
int knapsackRec(int[] $w$, int[] $p$, int $i$, int $c$)
if $c < 0$ then
    return $-\infty$
else if $i == 0$ or $c == 0$ then
    return $0$
else
    int $nottaken$ = knapsackRec($w, p, i-1, c$)
    int $taken$ = knapsackRec($w, p, i-1, c-w[i]$) + $p[i]$
    return max($nottaken, taken$)
```

Qual è la complessità della funzione knapsack() ricorsiva?

Complessità computazionale

Qual è la complessità della funzione knapsack() ricorsiva?

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

$$T(n) = O(2^n)$$

E' un algoritmo polinomiale?

Ovviamente no!

Possiamo fare meglio di così?

No, secondo l'opinione di quasi tutti gli informatici del mondo.

Memoization

Osservazione

Non tutti gli elementi della matrice sono necessari alla risoluzione del nostro problema.

```
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**

Memoization

Memoization (annotazione)

Tecnica che fonde l'approccio di memorizzazione della programmazione dinamica con l'approccio top-down di divide-et-impera

- Si crea una tabella DP, inizializzata con un valore speciale ad indicare che un certo sottoproblema non è ancora stato risolto
- Ogni volta che si deve risolvere un sottoproblema, si controlla nella tabella se è già stato risolto precedentemente
  - SI: si usa il risultato della tabella
  - NO: si calcola il risultato e lo si memorizza
- In tal modo, ogni sottoproblema viene calcolato una sola volta e memorizzato come nella versione bottom-up

Zaino con memoization

```
knapsack(int[] $w$, int[] $p$, int $n$, int $C$)
$DP$ = new int$[1 \ldots n][1 \ldots C]$
for $i = 1$ to $n$ do
    for $c = 1$ to $C$ do
        $DP[i][c] = -1$
return knapsackRec$(w, p, n, C, DP)$
```

- La tabella viene inizializzata esternamente, nella funzione wrapper
- Il valore -1 è scelto per indicare una cella non ancora calcolata

Zaino con memoization

```
int knapsackRec(int[] $w$, int[] $p$, int $i$, int $c$, int[][] $DP$)
if $c < 0$ then
    return $-\infty$
else if $i == 0$ or $c == 0$ then
    return 0
else
    if $DP[i][c] < 0$ then
        int $nottaken$ = knapsackRec($w, p, i-1, c, DP$)
        int $taken$ = knapsackRec($w, p, i-1, c-w[i], DP$) $+ p[i]$
        $DP[i][c]$ = max($nottaken, taken$)
    return $DP[i][c]$
```

Zaino con memoization

```
def knapsackRec(w, p, i, c, DP):
  if c < 0:
    return -math.inf
  elif i == 0 or c == 0:
    return 0
  else:
    if DP[i][c] < 0:
      nottaken = knapsackRec(w, p, i-1, c, DP)
      taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
      DP[i][c] = max(nottaken, taken)
    return DP[i][c]

def knapsack(w,p,C):
  n = len(w)
  DP = [[-1]*(C+1) for i in range(n+1)]
  return knapsackRec(w,p,n,C,DP)
```

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	-1	**0**	**0**	**10**	**10**	**10**	**10**	-1	**10**
2	0	-1	**7**	-1	-1	**10**	**17**	-1	-1	**17**
3	0	-1	-1	-1	-1	**15**	-1	-1	-1	**25**
4	0	-1	-1	-1	-1	-1	-1	-1	-1	**25**

Dizionario vs tabella

Inizializzazione tabella

- Il costo di inizializzazione è pari a $O(nC)$
- Applicata in questo modo, non c'è alcun vantaggio nell'utilizzare la tecnica di memoization
- Permette tuttavia di tradurre in fretta le espressioni ricorsive

Utilizzo di un dizionario (hash table)

- Invece di utilizzare una tabella, si utilizza un dizionario
- Non è necessario fare inizializzazione
- Il costo di esecuzione è pari a $O(\min(2^n, nC))$

Zaino con dizionario (Python)

```
def knapsackRec(w, p, i, c, DP):
  if c < 0:
    return -math.inf
  elif i == 0 or c == 0:
    return 0
  else:
    if not (i,c) in DP:
      nottaken = knapsackRec(w, p, i-1, c, DP)
      taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
      DP[i,c] = max(nottaken, taken)
    return DP[i,c]

def knapsack(w,p,C):
  n = len(w)
  DP = {}
  return knapsackRec(w,p,n,C,DP)
```

Memoization automatica in Python

```
from functools import wraps

def memo(func):
  cache = {}
  @wraps(func)
  def wrap(*args):
    if args not in cache:
      cache[args] = func(*args)
    return cache[args]
  return wrap
```

Memoization automatica in Python

```
@memo
def knapsackRec(w, p, i, c):
  if c < 0:
    return -math.inf
  elif i == 0 or c == 0:
    return 0
  else:
    nottaken = knapsackRec(w, p, i-1, c)
    taken = knapsackRec(w, p, i-1, c-w[i-1]) + p[i-1]
    return max(nottaken, taken)

def knapsack(w, p, C):
  return knapsackRec(w, p, len(w), C)
```

Ricostruzione della soluzione

Per esercizio

Variante dello zaino: senza limiti

Problema dello Zaino, senza limiti di scelta

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 $c \leq C$, senza porre limiti al numero di volte che un oggetto può essere selezionato.

Come modificare la formula ricorsiva?

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

Variante dello zaino: senza limiti

Semplificazione formula

In un caso come questo, è possibile semplificare la formula riducendo lo spazio occupato

Valore della soluzione

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

$$DP[c] = \begin{cases} 0 & c = 0 \\ \max_{w[i] \leq c} \{DP[c - w[i]] + p[i]\} & c > 0 \end{cases}$$

Implementazione tramite memoization

```
knapsack(int[] $w$, int[] $p$, int $n$, int $C$)
int[] $DP$ = new int$[0 \ldots C]$
for $i = 0$ to $C$ do
    $DP[i] = -1$
knapsackRec$(w, p, n, C, DP)$
return $DP[C]$
```

Implementazione tramite memoization

```
knapsackRec(int[] $w$, int[] $p$, int $n$, int $c$, int[] $DP$)
if $c == 0$ then
    return 0
if $DP[c] < 0$ then
    $DP[c] = 0$
    for $i = 1$ to $n$ do
        if $w[i] \leq c$ then
            int $val$ = knapsackRec($w, p, n, c - w[i], DP$) $+ p[i]$)
            $DP[c]$ = max($DP[c], val$)
return $DP[c]$
```

Complessità computazionale?

Qual è la complessità della funzione knapsack()?

Ricostruire la soluzione

Vantaggi

- Non è detto che tutti gli elementi debbano essere riempiti
- La complessità in spazio è pari a $\Theta(C)$

Svantaggi

Questo approccio rende più difficile ricostruire la soluzione.

- Possiamo ispezionare tutti gli elementi per capire da dove deriva il massimo
- Conviene tuttavia memorizzare l'indice da cui deriva il massimo

Implementazione tramite memoization

```
knapsack(int[] $w$, int[] $p$, int $n$, int $C$)
int[] $DP$ = new int[$0 \ldots C$]
int[] $pos$ = new int[$0 \ldots C$]
for $i = 0$ to $C$ do
    $DP[i] = -1$
    $pos[i] = -1$
knapsackRec($w, p, n, C, DP, pos$)
return solution($w, C, pos$)
```

Implementazione tramite memoization

```
knapsackRec(int[] $w$, int[] $p$, int $n$, int $c$, int[] $DP$, int[] $pos$)
if $c == 0$ then
    return 0
if $DP[c] < 0$ then
    $DP[c] = 0$
    for $i = 1$ to $n$ do
        if $w[i] \leq c$ then
            int $val = \mathsf{knapsackRec}(w, p, n, c - w[i], DP, pos) + p[i])$
            if $val \geq DP[c]$ then
                $DP[c] = val$
                $pos[c] = i$
return $DP[c]$
```

Implementazione tramite memoization

```
solution(int[] $w$, int $c$, int[] $pos$)
if $c == 0$ or $pos[c] < 0$ then
    return List()
else
    List $L$ = solution($w, c - w[pos[c]], pos$)
    $L$.insert($L$.head(), $pos[c]$)
    return $L$
```

- Restituisce una lista di indici selezionati (multinsieme, gli indici possono comparire più volte)
- Se $c = 0$, lo zaino è stato riempito perfettamente
- Se $pos[c] < 0$, lo zaino non può essere riempito interamente (e.g., pesi pari con capacità dispari).

Problema generale

DNA

Una stringa di molecole chiamate basi (Adenina, Citosina, Guanina, Timina)

Problema

Date due sequenze di DNA, trovare quanto siano "simili"

Esempi

- Una sottostringa dell'altra?
  CCTT $\subseteq$ AGACCCTTAA
- Distanza di edit:
  AGACCCTTAA può essere trasformata in AGACTCTTAA sostituendo una T con una C

Sottosequenza comune massimale

Definizione: sottosequenza

- Una sequenza P è una sottosequenza di T se P è ottenuto da T rimuovendo uno o più dei suoi elementi
- Alternativamente: P è definito come il sottoinsieme degli indici $\{1, \ldots, n\}$ degli elementi di T che compaiono anche in P
- I rimanenti elementi sono elencati nello stesso ordine, senza essere necessariamente contigui

Esempi

- P = `"AAATA"`
- T = `"AAAATTGA"`

Note

La sequenza vuota ∅ è sottosequenza di ogni altra sequenza

Sottosequenza comune massimale

Definizione: sottosequenza comune

Una sequenza X è una sottosequenza comune (common subsequence) di due sequenze T, U, se è sottosequenza sia di T che di U

- Scriviamo $X \in \mathcal{CS}(T, U)$

Definizione: sottosequenza comune massimale

Una sequenza $X \in \mathcal{CS}(T, U)$ è una sottosequenza comune massimale (longest common subsequence) di due sequenze T, U, se non esiste altra sottosequenza comune $Y \in \mathcal{CS}(T, U)$ tale che Y sia più lunga di X ($|Y| > |X|$).

- Scriviamo $X \in \mathcal{LCS}(T, U)$

Definizione del problema

Problema: LCS

Date due sequenze T e U, trovare la più lunga sottosequenza comune di T e U.

Esempio

- $T =$ “`AAAATTGA`”
- $U =$ “`TAACGATA`”
- Output?

Come risolvereste questo problema?

Una soluzione di "forza bruta"

```
int LCS(ITEM[] $T$, ITEM[] $U$)
ITEM[] $maxsofar$ = nil
foreach subsequence $X$ of $T$ do
    if $X$ is subsequence of $U$ then
        if len($X$) > len($maxsofar$) then
            $maxsofar = X$
return $\ell$
```

Complessità computazionale

Domande

- Data una sequenza T lunga n, quante sono le sottosequenze di T?
  2^n
- Quanto costa verificare se una sequenza è sottosequenza di un'altra?
  $O(m+n)$
- Qual è la complessità computazionale di LCS()?
  $T(n) = \Theta(2^n(m+n))$
- Possiamo fare meglio di così?

Descrizione matematica della soluzione ottima

Prefisso (Prefix)

Data una sequenza T composta dai caratteri $t_1 t_2 \ldots t_n$, $T(i)$ denota il prefisso di T dato dai primi i caratteri, i.e.:

$$T(i) = t_1 t_2 \ldots t_i$$

Esempi

- $T =$ "`ABDCCAABD`"
- $T(0) =$ ""
- $T(3) =$ "`ABD`"
- $T(6) =$ "`ABDCCA`"

Descrizione matematica della soluzione ottima

Goal

Date due sequenze T e U di lunghezza n e m, scriviamo una formula ricorsiva $LCS(T(i), U(j))$ che restituisca la LCS dei prefissi $T(i)$ e $U(j)$.

$$LCS(T(i), U(j)) = \begin{cases} ? & \text{Caso base} \\ ? & \text{Casi ricorsivi} \end{cases}$$

Casi ricorsivi

Caso 1

Considerate due prefissi $T(i)$ e $U(j)$ tali per cui l'ultimo loro carattere coincide: $t_i = u_j$. Come calcolereste la LCS di $T(i)$ e $U(j)$?

- Esempio: $T(i) =$ "`ALBERTO`", $U(j) =$ "`PIERO`"

Soluzione

$$LCS(T(i), U(j)) = LCS(T(i-1), U(j-1)) \oplus t_i$$

dove $\oplus$ è l'operatore di concatenazione.

- $LCS($"`ALBERTO`", "`PIERO`"$) = LCS($"`ALBERT`", "`PIER`"$) \oplus$ "`O`"

Casi ricorsivi

Caso 2

Considerate due prefissi $T(i)$ e $U(j)$ tali per cui l'ultimo loro carattere è differente: $t_i \neq u_j$. Come calcolereste la LCS di i e j?

- Esempio: $T(i) = $ "`ALBERT`", $U(j) = $ "`PIER`"

Soluzione

$$LCS(T(i), U(j)) = longest(LCS(T(i-1), U(j)), LCS(T(i), U(j-1))$$

- $LCS($"`ALBERT`", "`PIER`"$) =$ $longest(LCS($"`ALBER`", "`PIER`"$), LCS($"`ALBERT`", "`PIE`"$))$

Casi base

Casi base

Qual è la più lunga sottosequenza di $T(i)$ e $U(j)$, quando uno dei prefissi è vuoto, i.e. se $i = 0$ **or** $j = 0$?

- Esempio: $T(i) =$ "`ALBERTO`", $U(0) = \emptyset$

Soluzione

$$LCS(T(i), U(0)) = \emptyset$$

- $LCS($"`ALBERTO`"$, \emptyset) = \emptyset$

La formula completa

$$LCS(T(i), U(j)) = \begin{cases} \emptyset & i = 0 \textbf{ or } j = 0 \\ LCS(T(i-1), U(j-1)) \oplus t_i & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i = u_j \\ longest(LCS(T(i-1), U(j)), & \\ \qquad LCS(T(i), U(j-1)) & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i \neq u_j \end{cases}$$

Dimostrazione

Il fatto che la formula sia corretta dovrebbe essere provato. La dimostrazione è per assurdo.

Sottostruttura ottima

Teorema – Sottostruttura ottima

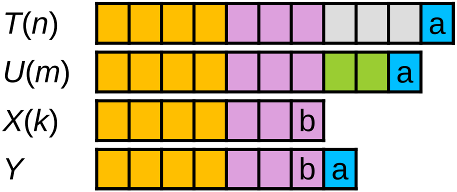
Date le due sequenze $T = (t_1, \ldots, t_n)$ e $U = (u_1, \ldots, u_m)$, sia $Z = (x_1, \ldots, x_k)$ una LCS di T e U. Sono dati tre casi:

1. $t_n = u_m \quad \Rightarrow \quad x_k = t_n = u_m$ **and** $X(k-1) \in \mathcal{LCS}(T(n-1), U(m-1))$
2. $t_n \neq u_m \wedge x_k \neq t_n \quad \Rightarrow \quad Z \in \mathcal{LCS}(T(n-1), U)$
3. $t_n \neq u_m \wedge x_k \neq u_m \quad \Rightarrow \quad Z \in \mathcal{LCS}(T, U(m-1))$

Dimostrazione – Punto 1 – Parte A

$t_n = u_m \Rightarrow x_k = t_n = u_m$

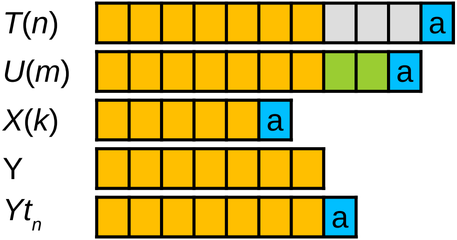
- Per assurdo: $x_k \neq t_n = u_m$.
- Si consideri $Y = Xt_n$.
- Allora $Y \in \mathcal{CS}(T, U)$ e $|Y| > |X|$, contraddizione.



Dimostrazione – Punto 1 – Parte B

$$t_n = u_m \Rightarrow X(k-1) \in LCS(T(n-1), U(m-1))$$

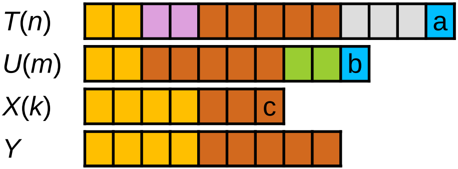
- Per assurdo: $X(k-1) \notin \mathcal{LCS}(T(n-1), U(m-1))$.
- Allora $\exists Y \in \mathcal{LCS}(T(n-1), U(m-1))$ tale che $|Y| > |X(k-1)|$.
- Quindi $Yt_n \in \mathcal{CS}(T, U)$ e $|Yt_n| > |X(k-1)t_n| = X$, contraddizione.



Dimostrazione – Punto 2 (Punto 3 simmetrico)

$$t_n \neq u_m \wedge x_k \neq t_n \Rightarrow X \in \mathcal{LCS}(T(u-1), U)$$

- Per assurdo: $X \notin \mathcal{LCS}(T(n-1), U)$.
- Allora $\exists Y \in \mathcal{LCS}(T(n-1), U)$ tale che $|Y| > |X|$.
- Quindi è anche vero che $Y \in \mathcal{LCS}(T, U)$.
- Quindi X non è una LCS di T e U, assurdo.



Ricorrenza per il valore della soluzione ottimale

Lunghezza della LCS

Due due sequenze T e U di lunghezza n e m, scrivere una formula ricorsiva $DP[i][j]$ che restituisca la lunghezza della LCS dei prefissi $U(i)$ e $U(j)$.

$$DP[i][j] = \begin{cases} 0 & i = 0 \textbf{ or } i = 0 \\ DP[i-1][j-1] + 1 & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i = u_j \\ \max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i \neq u_j \end{cases}$$

Dove si trova l'informazione relativa al problema originale?

$DP[n][m]$ contiene la lunghezza della LCS del problema originale.

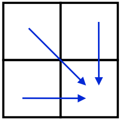
Calcolare la lunghezza della LCS

```
int lcs(ITEM[] $T$, ITEM[] $U$, int $n$, int $m$)
int[][] $DP$ = new int$[0 \ldots n][0 \ldots m]$
for $i = 0$ to $n$ do
    $DP[i][0] = DP[0][i] = 0$
for $i = 1$ to $n$ do
    for $j = 1$ to $m$ do
        if $T[i] == U[j]$ then
            $DP[i][j] = DP[i-1][j-1] + 1$
        else
            $DP[i][j] = \mathsf{max}(DP[i-1][j], DP[i][j-1])$
return $DP[n][m]$
```

Esempio 1

- **TACCBT**
- **ATBCBD**

↘ deriva da *i*-1,*j*-1
↓ deriva da *i*-1,*j*
→ deriva da *i*,*j*-1



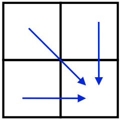
j		0	1	2	3	4	5	6
i			**A**	**T**	**B**	**C**	**B**	**D**
0		0	0	0	0	0	0	0
1	**T**	0	↓0	↘ 1	→1	→1	→1	→1
2	**A**	0	↘ 1	↓ 1	↓ 1	↓ 1	↓ 1	↓ 1
3	**C**	0	↓ 1	↓ 1	↓ 1	↘ 2	→2	→2
4	**C**	0	↓ 1	↓ 1	↓ 1	↘ 2	↓ 2	→2
5	**B**	0	↓ 1	↓ 1	↘ 2	↓ 2	↘ 3	→3
6	**T**	0	↓ 1	↘ 2	↓ 2	↓ 2	↓ 3	↓ 3

$$DP[i][j] = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP[i-1][j-1] + 1 & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i = u_j \\ \max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i \neq u_j \end{cases}$$

Ricostruire la soluzione

- **ACGGCT**
- **CTCTGT**

↘ deriva da *i*-1,*j*-1
↓ deriva da *i*-1,*j*
→ deriva da *i*,*j*-1



j		0	1	2	3	4	5	6
i			**C**	**T**	**C**	**T**	**G**	**T**
0		0	0	0	0	0	0	0
1	**A**	0	↓ 0	↓ 0	↓ 0	↓ 0	↓ 0	↓ 0
2	**C**	0	↘1	→1	↘ 1	→1	→1	→1
3	**G**	0	↓ 1	↓ 1	↓ 1	↓ 1	↘2	→2
4	**G**	0	↓ 1	↓ 1	↓ 1	↓ 1	↘ 2	↓ 2
5	**C**	0	↘ 1	↓ 1	↘2	→2	↓ 2	↓ 2
6	**T**	0	↓ 1	↘2	↓ 2	↘ 3	→3	↘ 3

Utilizzando la tabella, come possiamo ottenere la soluzione?

Ricostruire la sottosequenza comune

```
int lcs(ITEM[] $T$, ITEM[] $U$, int $n$, int $m$)
...
return subsequence($DP, T, U, n, m$)
```

```
subsequence(int[][] $DP$, ITEM[] $T$, ITEM[] $U$, int $i$, int $j$)
if $i == 0$ or $j == 0$ then
    return List()
if $T[i] == U[j]$ then
    $S$ = subsequence($DP, T, U, i-1, j-1$)
    $S$.insert($S$.head(), $T[i]$)
    return $S$
else
    if $DP[i-1][j] > DP[i][j-1]$ then
        return subsequence($DP, T, U, i-1, j$)
    else
        return subsequence($DP, T, U, i, j-1$)
```

Complessità computazionale

Qual è la complessità computazionale di subsequence()?

$$T(n) = O(m + n)$$

Qual è la complessità computazionale di LCS()?

$$T(n) = O(mn)$$

Commenti finali

Take-home message (prendi e porta a casa)

Non sempre è necessario memorizzare informazioni aggiuntive per ricostruire la soluzione.

Reality check – LCS e `diff`

`diff`

- Esamina due file di testo, evidenziondone le differenze a livello di riga.
- Lavorare a livello di riga significa che i confronti fra simboli sono in realtà confronti fra righe, e che n ed m sono il numero di righe dei due file

Ottimizzazioni

- `diff` è utilizzato soprattutto per codice sorgente; è possibile applicare euristiche sulle righe iniziali e finali
- Per distinguere le righe - utilizzo di funzioni hash

Reality check – LCS e `diff`

```
Questo è il testo originale
alcune linee non dovrebbero
cambiare mai
altre invece vengono
rimosse
altre vengono aggiunte
```

```
Questo è il testo nuovo
alcune linee non dovrebbero
cambiare mai
altre invece vengono
cancellate
altre vengono aggiunte
come questa
```

```
- Questo è il testo originale
+ Questo è il testo nuovo
  alcune linee non dovrebbero
  cambiare mai
  altre invece vengono
- rimosse
+ cancellate
  altre vengono aggiunte
+ come questa
```

Figura 13.4: Il file `original.txt` (a sinistra); il file `new.txt` (al centro); l'output di `diff original.txt new.txt` (a destra).