

Algoritmi e Strutture Dati

Programmazione dinamica – Parte 2

Alberto Montresor

Università di Trento

2021/03/02

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



Sommario

- 1 Zaino con memoization
- 2 Variante dello zaino, senza limiti
- 3 Sottosequenza comune massimale

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:

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

Zaino ricorsivo

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

```

int 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)
  
```

Complessità computazionale

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

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

È un algoritmo polinomiale?

Ovviamente no!

Possiamo fare meglio di così?

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

Osservazione

Osservazione

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

$$w = [4, 2, 3, 4]$$

$$p = [10, 7, 8, 6] \quad C = 9$$

	<i>c</i>									
<i>i</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
- Quando si deve risolvere un sottoproblema, si controlla nella tabella se è già stato risolto:
 - 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

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

```
    DP = new int[1...n][1...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

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?

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

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?

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

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 $c \leq C$.

$$DP[c] = \begin{cases} ? & c = 0 \\ ? & c > 0 \end{cases}$$

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 $c \leq C$.

$$DP[c] = \begin{cases} 0 & c = 0 \\ ? & c > 0 \end{cases}$$

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

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

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

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

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

```
knapsackRec( $w, p, n, C, DP$ )
```

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

Implementazione tramite memoization

```

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

```

Complessità computazionale?

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

```

int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)


---


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

Complessità computazionale?

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

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

- Nel caso pessimo, è necessario riempire ognuno dei C elementi del vettore DP
- Riempire un elemento costa $O(n)$

Riduzione dello spazio occupato

Vantaggi

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

Svantaggi

Questo approccio rende più difficile ricostruire la soluzione.

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

Implementazione tramite memoization

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

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

```
int[] pos = new int[0...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

```

int 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] ≤ c then
            int val = knapsackRec(w, p, n, c - w[i], DP, pos) + p[i]
            if val ≥ DP[c] then
                DP[c] = val
                pos[c] = i
    return DP[c]

```

Implementazione tramite memoization

```
LIST 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 (A denina, C itosina, G uanina, T imina)

Problema

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

Esempi

- Una **sottostringa** dell'altra?

CCTT \subseteq AGAC**CCTT**AA

- **Distanza di edit:**

AGAC**CCTT**AA può essere trasformata in AGAC**TCTT**AA 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 $\{1, \dots, n\}$ degli elementi di T che compaiono anche in P
- I rimanenti elementi sono elencati nello stesso ordine, senza essere necessariamente contigui

Esempi

- $P = \text{"AAATA"}$
- $T = \text{"AAAATTGA"}$

Note

La sequenza vuota \emptyset 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

- $T = \text{“AAAATTGA”}$
- $U = \text{“TAACGATA”}$
- 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 maxsofar

```

Complessità computazionale

Domande

- Data una sequenza T lunga n , quante sono le sottosequenze di T ?

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?

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 $\text{LCS}()$?

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 $\text{LCS}()$?
 $T(n) = \Theta(2^n(m + n))$
- Possiamo fare meglio di così?

Descrizione matematica della soluzione ottima

Prefisso (**P**refix)

Data una sequenza T composta dai caratteri $t_1t_2\dots t_n$, $T(i)$ denota il **prefisso** di T dato dai primi i caratteri, i.e.:

$$T(i) = t_1t_2\dots t_i$$

Esempi

- $T = \text{"ABDCCAABD"}$
- $T(0) = \text{" "}$
- $T(3) = \text{"ABD"}$
- $T(6) = \text{"ABDCCA"}$

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: $T(i) = \text{"ALBERTO"} , U(j) = \text{"PIERO"}$

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: $T(i) = \text{"ALBERTO"} , U(j) = \text{"PIERO"}$

Soluzione

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

dove \oplus è l'operatore di concatenazione.

- $LCS(\text{"ALBERTO"}, \text{"PIERO"}) = LCS(\text{"ALBERT"}, \text{"PIER"}) \oplus \text{"O"}$

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: $T(i) = \text{"ALBERT"} , U(j) = \text{"PIER"}$

Soluzione

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: $T(i) = \text{"ALBERT"} , U(j) = \text{"PIER"}$

Soluzione

$$LCS(T(i), U(j)) = \text{longest}(LCS(T(i-1), U(j)), LCS(T(i), U(j-1)))$$

- $LCS(\text{"ALBERT"}, \text{"PIER"}) = \text{longest}(LCS(\text{"ALBER"}, \text{"PIER"}), LCS(\text{"ALBERT"}, \text{"PIE"}))$

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: $T(i) = \text{"ALBERTO"}$, $U(0) = \emptyset$

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: $T(i) = \text{"ALBERTO"} , U(0) = \emptyset$

Soluzione

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

- $LCS(\text{"ALBERTO"}, \emptyset) = \emptyset$

La formula completa

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

Dimostrazione

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

Sottostruttura ottima

Teorema – Sottostruttura ottima

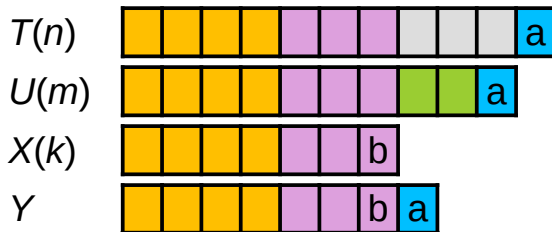
Date le due sequenze $T = (t_1, \dots, t_n)$ e $U = (u_1, \dots, u_m)$, sia $X = (x_1, \dots, x_k)$ una LCS di T e U . Sono dati tre casi:

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

Dimostrazione – Punto 1 – Parte A

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

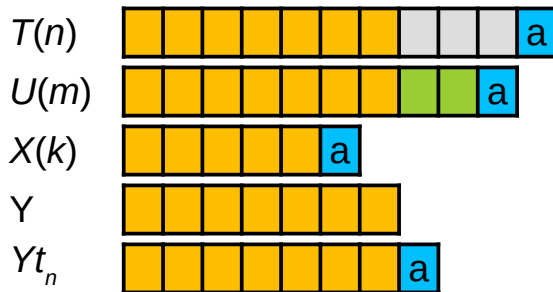
- Per assurdo: $x_k \neq t_n = u_m$.
- Si consideri $Y = Xt_n$.
- Allora $Y \in \mathcal{CS}(T, U)$ e $|Y| > |X|$, contraddizione.



Dimostrazione – Punto 1 – Parte B

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

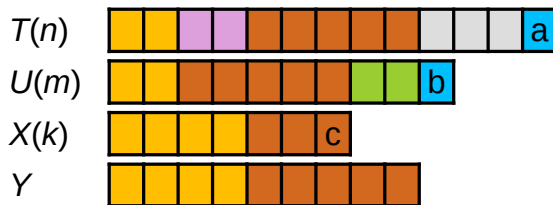
- Per assurdo: $X(k-1) \notin \mathcal{LCS}(T(n-1), U(m-1))$.
- Allora $\exists Y \in \mathcal{LCS}(T(n-1), U(m-1))$ tale che $|Y| > |X(k-1)|$.
- Quindi $Yt_n \in \mathcal{CS}(T, U)$ e $|Yt_n| > |X(k-1)t_n| = X$, contraddizione.



Dimostrazione – Punto 2 (Punto 3 simmetrico)

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

- 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 $T(i)$ e $U(j)$.

$$DP[i][j] = \begin{cases} ? & i = 0 \text{ or } j = 0 \\ ? & i > 0 \text{ and } j > 0 \text{ and } t_i = u_j \\ ? & i > 0 \text{ and } j > 0 \text{ and } t_i \neq u_j \end{cases}$$

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 $T(i)$ e $U(j)$.

$$DP[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ ? & i > 0 \text{ and } j > 0 \text{ and } t_i = u_j \\ ? & i > 0 \text{ and } j > 0 \text{ and } t_i \neq u_j \end{cases}$$

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 $T(i)$ e $U(j)$.

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

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 $T(i)$ e $U(j)$.

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

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 $T(i)$ e $U(j)$.

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

Dove si trova l'informazione relativa al problema originale?

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 $T(i)$ e $U(j)$.

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

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)


---

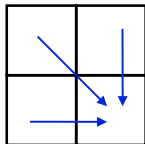

int[][] DP = new int[0...n][0...m]
for i = 0 to n do
    | DP[i][0] = 0
for j = 0 to m do
    | DP[0][j] = 0
for i = 1 to n do
    | for j = 1 to m do
        | if T[i] == U[j] then
            | | DP[i][j] = DP[i - 1][j - 1] + 1
        | else
            | | DP[i][j] = max(DP[i - 1][j], DP[i][j - 1])
    |
return DP[n][m]

```

Esempio 1

- TACCBT
- ATBCBD

↖ deriva da $i-1, j-1$
 ↓ deriva da $i-1, j$
 → deriva da $i, j-1$

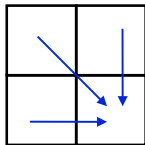


		j						
		0	1	2	3	4	5	6
i			A	T	B	C	B	D
	0		0	0	0	0	0	0
	1	T	0	↓0	↖1	→1	→1	→1
	2	A	0	↖1	↓1	↓1	↓1	↓1
	3	C	0	↓1	↓1	↓1	↖2	→2
	4	C	0	↓1	↓1	↓1	↖2	↓2
	5	B	0	↓1	↓1	↖2	↓2	↖3
	6	T	0	↓1	↖2	↓2	↓2	↓3

Ricostruire la soluzione

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

↖ deriva da $i-1, j-1$
 ↓ deriva da $i-1, j$
 → deriva da $i, j-1$



i \ j		0	1	2	3	4	5	6
			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

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.tail(), 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()`?

Complessità computazionale

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

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

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

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, evidenziandone 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).