Insegnamento di algoritmi avanzati

Paradigma programmazione dinamica

Roberto Posenato

ver. 0.5, 01/10/2009

Introduzione

In questa lezione si richiamano i concetti fondamentali del paradigma *programmazione dinamica*, già studiato e usato nel corso di Algoritmi e strutture dati.

Idea generale

La tecnica della programmazione dinamica è stata introdotta da Bellman (1957) per progettare algoritmi che risolvono problemi di ottimizzazione.

La tecnica consiste di 4 fasi:

1. Caratterizzazione della struttura di una soluzione ottima.
2. Definizione ricorsiva del valore di una soluzione ottima.
3. Calcolo del valore di una soluzione ottima secondo uno schema bottom-up.
4. Costruzione della soluzione ottima dalle informazioni calcolate.

Idea generale

Confronto programmazione dinamica - *divide et impera*

	programmazione dinamica	***divide et impera***
tipo risoluzione	bottom-up	top-down
tipo di programma	iterativa	ricorsiva
uso memoria	esplicito per i risultati	implicito
applicazione ideale	sottoistanze non disgiunte	sottoistanze disgiunte

Dettagli

Quando si usa

Si usa quando:

- una soluzione ottima contiene al suo interno le soluzioni ottime dei sottoproblemi.
- un algoritmo *divide et impera* per lo stesso problema richiederebbe di risolvere più volte alcuni sottoproblemi elevando troppo il costo computazionale (sottoproblemi ripetuti).

Nota!

Proprietà della sottostruttura ottima usata anche dagli algoritmi greedy.
Greedy: si sceglie quale sottoproblema risolvere e poi si risolve.
Prog. dinamica: si risolvono i sottoproblemi e poi si sceglie.

Esempi di applicazione

Massima Sottosequenza Crescente

Sottosequenza

Data una sequenza di n oggetti $a_1, \ldots, a_n$, una sottosequenza è un sottoinsieme ordinato di questi oggetti tipo $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$.
Una sottosequenza $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ è crescente quando gli elementi sono confrontabili e sono in ordine strettamente crescente: $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

PROBLEMA MASSIMA SOTTOSEQUENZA CRESCENTE

DESCRIZIONE: Una sequenza di interi $a_1, \ldots, a_n$.

QUESITO: Determinare la sottosequenza crescente di lunghezza massima.

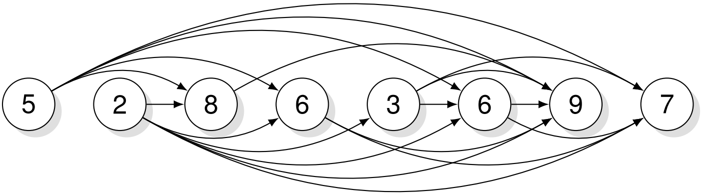
Esempio: 5, 2, 8, 6, 3, 6, 9, 7 $\Longrightarrow$ 2, 3, 6, 9

Esempi di applicazione

Massima Sottosequenza Crescente

Fase 1: Caratterizzazione della struttura di una soluzione ottima.

- Data un'istanza, si rappresentano graficamente le relazioni fra gli elementi rispetto alla soluzione richiesta:



- Il grafo è un DAG in quanto tutti gli archi (i, j) hanno $i < j$.
- C'è una corrispondenza biunivoca tra sottosequenze crescenti e cammini nel DAG.
- Trovare la massima sottosequenza crescente è equivalente a trovare il massimo cammino nel DAG.

Esempi di applicazione

Massima Sottosequenza Crescente

Fase 2: Definizione ricorsiva del valore di una soluzione ottima.

- Sia $L(j)$ la lunghezza del cammino più lungo che termina in j:

$$L(j) = \max\{L(i) \mid (i,j) \in E\} + 1$$

- Se si determinano $L(1), \ldots, L(n)$, il valore della soluzione ottima è dato dal valore $L()$ massimo.
- $L(1), \ldots, L(n)$ sono le sottoistanze da risolvere.

Fase 3: Calcolo del valore di una soluzione ottima secondo uno schema bottom-up.

- Se si risolvono in ordine partendo da $L(1)$, in un'unica "passata" si determinano tutti i valori.
- Per risolvere occorre conoscere i predecessori per ciascun j: come si fa? quanto costa?

Fase 4: Costruzione della soluzione ottima: Cosa si deve aggiungere per risolvere la versione costruttiva?

Esempi di applicazione

Massima Sottosequenza Crescente

Versione valutativa:

Procedura `MSC-V`$(a_1, a_2, \ldots, a_n)$

Si costruisca il DAG $G = (V, E)$ da $(a_1, a_2, \ldots, a_n)$;
$L =$new int$[n]$; // Soluzioni parziali
$ottimo = -\infty$; // Valore soluzione ottima
foreach $(j = 1, \ldots, n)$ **do**
    $max = 0$;
    foreach $(i|(i, j) \in E)$ **do** // $(i, j) \in E$ è da risolvere!
        if $(L[i] > max)$ **then** $max = L[i]$;
    endfch
    $L[j] = 1 + max$;
    if $(L[j] > ottimo)$ **then** $ottimo = L[j]$;
endfch
return *ottimo;*

Esempi di applicazione

Massima Sottosequenza Crescente

Versione costruttiva:

Procedura `MSC`$(a[1], a[2], \ldots, a[n])$

Si costruisca il DAG $G^T = (V, E^T)$ da $(a[1], a[2], \ldots, a[n])$;
L =new int[n]; ottimo $= -\infty$;
int[] prec; // Mantiene i precedenti
iFine $= 0$; // Indice nodo finale sequenza ottima
foreach $(j = 1, \ldots, n)$ **do**
    $max = 0$;
    foreach $(i | (j, i) \in E^T)$ **do** // $(i, j) \in E$ risolto!
        if $(L[i] > max)$ **then** $max = L[i]; prec[j] = i$;
    endfch
    $L[j] = 1 + max$;
    if $(L[j] > ottimo)$ **then** $ottimo = L[j]; iFine = j$;
endfch
return `reversePrint`$(G, prec, iFine)$;

Esempi di applicazione

Massima Sottosequenza Crescente: complessità

Correttezza

Si veda la costruzione.

Complessità

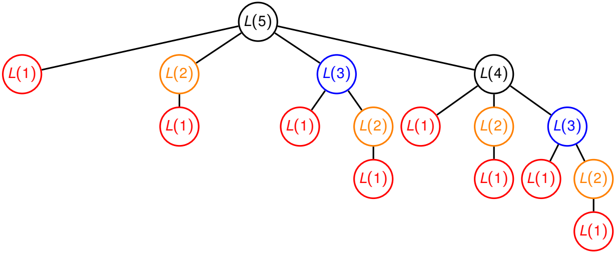
- Costruzione grafo trasposto richiede tempo $O(n^2)$ in quanto per ciascun elemento si deve inserire un arco orientato verso il nodo che rappresenta l'elemento in ciascuna lista di adiacenza dei nodi che rappresentano elementi inferiori del corrente.
- Il calcolo di $L(j)$ richiede tempo proporzionale al grado del nodo j. Sommando su tutti i nodi: $O(|E|)$. $|E| \leq (n-1)n/2$
- Totale: $O(n^2)$.

Esempi di applicazione

Massima Sottosequenza Crescente: approccio *divide et impera*

Se si risolvesse con il *divide et impera*, non si ottiene un algoritmo più semplice?

- La soluzione ricorsiva è sempre la medesima $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$.
- Se il DAG contiene tutti gli archi possibili, la soluzione diventa $L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$. Per esempio, se $j = 5$, l'albero delle chiamate ricorsive diventa:



Numero delle chiamate ricorsive: $2^{j-1} - 1$!

Esempi di applicazione

String matching approssimato

Definizione 1 (Stringhe k-approssimate)

Data una stringa T, testo, di dimensione n e una stringa P, pattern, di $m < n$ caratteri, un'occorrenza k-approssimata, $0 \leq k \leq m$, di P in T è una copia di P in T dove ci sono k errori del tipo:

1. i corrispondenti caratteri sono diversi;
2. come in 1) e il carattere in P non è presente in T (o T ha un '-' (gap));
3. come in 1) e il carattere in T non è presente in P (o P ha un '-' (gap)).

Esempio 1

Se $T =$ "`It's snowy`" e $P =$ "`sunny`"

3-approssimata:

```
It's snowy     It's s-nowy
     sunny          sunn-y
```

Esempi di applicazione

String matching approssimato

PROBLEMA STRING MATCHING APPROSSIMATO

DESCRIZIONE: Una testo T di dimensione n e un pattern P di dimensione $m < n$.

QUESITO: Determinare un'occorrenza k-approssimata minima (rispetto a k) di P in T.

Nota!

Si verifica che $k \leq m$.
Se si ottiene $k = 0 \implies$ istanza positiva di RICERCA DI STRINGA.
Il problema EDIT DISTANCE è una generalizzazione in cui i gap sono visti come operazioni di inserimento/cancellazione da fare su T per trasformalo in P.
Per questo problema non ci sono vincoli per n, m.

Esempi di applicazione

String matching approssimato

Fase 1: Caratterizzazione della struttura di una soluzione ottima.

- Si supponga di avere un'occorrenza k-approssimata K ottima di P in T di lunghezza l.
  Esempio 1: $P =$ "$\texttt{ssnowy}$" $\implies K =$ "$\texttt{-snowy}$", $P =$ "$\texttt{snoy}$" $\implies K =$ "$\texttt{sno-y}$".
- Verifichiamo se è possibile scrivere K come composizione di una soluzione ottima di un sottoproblema.
- Si consideri il carattere K_l, in corrispondenza con T_j e P_i. Notare: $(K_l == T_j)$ o è un gap.
- Supponiamo che $K = K' \cdot K_l$ con K' ottimo per un qualche sottoproblema.
- Ci sono 3 casi possibili:
  1. K_l coincide o non coincide con P_i o
  2. K_l non coincide perché P_i non è presente in T (gap in T) o
  3. K_l non coincide perché non è presente in P (gap in P).

Esempi di applicazione

String matching approssimato

- Caso 1: K' è un'occorrenza k'-approssimata ottima di $P_{1,\ldots,i-1}$ in $T_{1,\ldots,j-1}$ con $k' = k$ se $T_j == P_i$, $k-1$ altrimenti.
- Caso 2 (gap in T): K' è un'occorrenza $k-1$-approssimata ottima di $P_{1,\ldots,i-1}$ in $T_{1,\ldots,j}$.
  Infatti avanzando di un carattere nel pattern, si commette un errore e si ottiene una stringa (K) che risulta essere k-approssimata per $P_{1,\ldots,i}$ in $T_{1,\ldots,j}$.
- Caso 3 (gap in P): K' è un'occorrenza $k-1$-approssimata ottima di $P_{1,\ldots,i}$ in $T_{1,\ldots,j-1}$.
  Infatti avanzando di un carattere nel testo, si commette un errore e si ottiene una stringa (K) che risulta essere k-approssimata per $P_{1,\ldots,i}$ in $T_{1,\ldots,j}$.

Esempi di applicazione

String matching approssimato

Fase 2: Definizione ricorsiva del valore di una soluzione ottima.

- Il valore della soluzione ottima (k) può essere espresso come in funzione di $K[i,j] \stackrel{\text{def}}{=}$ il minimo numero di errori nell'approssimare $P[1,\ldots,i]$ in $T[1,\ldots,j]$.
- Vale che $k = \min_j K[m,j]$.
- Viste le relazioni della sottostruttura ottima:
  - Caso 1: se $T[j] \neq P[i]$ c'è un errore
    $\Rightarrow K[i,j] = 1 + K[i-1, j-1]$
    altrimenti, $K[i,j] = K[i-1,j-1]$.
  - Caso 2: c'è un errore dato dal gap in T. Il minimo k è 1 + il minimo del sottoproblema in cui si cerca $P[1,\ldots,i-1]$ in $T[1,\ldots,j] \Rightarrow K[i,j] = 1 + K[i-1,j]$.
  - Caso 3: c'è un errore dato dal gap in P. Il minimo k è 1 + il minimo del sottoproblema in cui si cerca $P[1,\ldots,i]$ in $T[1,\ldots,j-1] \Rightarrow K[i,j] = 1 + K[i,j-1]$.

Esempi di applicazione

String matching approssimato

- Se si risolvono i 3 sottoproblemi e si prende il minimo, si ha la soluzione dell'istanza $K[i,j]$:
  $K[i,j] = \min\{\mathrm{diff}(i,j) + K[i-1,j-1], 1 + K[i-1,j], 1 + K[i,j-1]\}$
  dove $\mathrm{diff}(i,j) \stackrel{\text{def}}{=} 0$ se $P[i] = T[j]$, 1 altrimenti.

Esempio 2

$T = \texttt{It's snow}\boxed{\texttt{y}}$ e $P = \texttt{sunn}\boxed{\texttt{y}}$

$K[5,10] = \min\{0 + K[4,9], 1 + K[4,10], 1 + K[5,9]\}$.

$K[4,10] = \min\{1 + K[3,9], 1 + K[3,10], 1 + K[4,9]\}$.

$K[5,9] = \min\{1 + K[4,8], 1 + K[4,9], 1 + K[5,8]\}$.

...

Esempi di applicazione

String matching approssimato

Fase 3: Calcolo del valore di una soluzione ottima secondo uno schema bottom-up.

- Le soluzioni $K[i, j]$ formano una tabella bidimensionale.
- Tale tabella può essere completata riga per riga (da 1 a m) o colonna per colonna (da 1 a n).
- Rimane solo da definire la base della ricorsione: $K[\cdot, 0]$ e $K[0, \cdot]$:
  - $K[i, 0] = i$ in quanto P ha i caratteri in più di ε;
  - $K[0, j] = 0$ perché ε non ha errori di approssimazione.

Nota!

Per il problema EDIT DISTANCE,
$K[0, j] = j$ perché servono j cancellazioni per allineare T a ε.

Esempi di applicazione

String matching approssimato

Esempio 3

T = `It's snowy` e P = `sunny`

	ε	I	t	'	s		s	n	o	w	y
ε	0	0	0	0	0	0	0	0	0	0	0
s	1	1	1	1	0	1	0	1	1	1	1
u	2	2	2	2	1	1	1	1	2	2	2
n	3	3	3	3	2	2	2	1	2	3	3
n	4	4	4	4	3	3	3	2	2	3	4
y	5	5	5	5	4	4	4	3	3	3	3

Istanza per String Matching

	ε	I	t	'	s		s	n	o	w	y
ε	0	1	2	3	4	5	6	7	8	9	10
s	1	1	2	3	3	4	5	6	7	8	9
u	2	2	2	3	4	4	5	6	7	8	9
n	3	3	3	3	4	5	5	5	6	7	8
n	4	4	4	4	4	5	6	5	6	7	8
y	5	5	5	5	5	5	6	6	6	7	7

Istanza per Edit Distance

Esempi di applicazione

String matching approssimato: pseudocodifica

Procedura `SMA(`T, P`)`

$n = |T|;\ m = |P|;$

for $i = 0; i <= n; i++)$ **do** $K[0, i] = 0;$

for $i = 0; i <= m; i++)$ **do** $K[i, 0] = i;$

for $(i = 1; i <= m; i++)$ **do**

  for $(j = 1; j <= n; j++)$ **do**

    $Min = \min(K[i-1, j] + 1, K[i, j-1] + 1);$

    $\text{diff} = (T[j] == P[i])?0 : 1;$

    $K[i, j] = \min(Min, \text{diff} + K[i-1, j-1]);$

  endfor

endfor

$Min = K[m, 0];$

for $(j=1; j<=n; j++)$ **do if** $(K[m, j] < Min)$ **then** $Min = K[m, j];\ i = j;$

return *Errori = Min; Fine Matching =* $i - 1$*;*

Esempi di applicazione

String matching approssimato: complessità

Correttezza

Per costruzione.

Complessità

- La complessità è data dal costo di determinazione della tabella: $O(nm)$.
- Si può completare `SMA()` in modo che restituisca anche la stringa approssimata sempre al medesimo costo.
- Se il problema esplicita anche k, si può modificare `SMA()` in modo che la complessità divenga $O(kn)$ (Algoritmo di Landau e Vishkin).

Fase 4: Costruzione della soluzione ottima: lasciato per esercizio.

Esempi di applicazione

Problema dello ZAINO

PROBLEMA ZAINO

DESCRIZIONE: Un insieme di n di oggetti caratterizzati ciascuno da un valore v_i e da un peso p_i, entrambi interi positivi; un intero positivo P, portata massima dello zaino.

QUESITO: Determinare un sottoinsieme $X = \{x_1, x_2, \ldots, x_n\}$, con $x_i = 0$ o 1, degli oggetti tale che il valore totale $\sum_{i=1}^{n} x_i v_i$ sia massimo e il peso totale $\sum_{i=1}^{n} x_i p_i \leq P$.

Nota!

Questo problema è già stato risolto sia usando il backtracking (versione con ripetizione) sia usando il branch & bound (versione classica).
Qui si presenta la soluzione per la versione classica lasciando quella per la versione con ripetizione come esercizio.

Esempi di applicazione

Problema dello ZAINO

Fase 1: Caratterizzazione della struttura di una soluzione ottima.

- Dimostrazione con tecnica taglia e incolla.
- Si ha uno zaino ottimo Z. Si toglie un elemento, i ad esempio, da tale zaino.
- Lo zaino rimanente Z' è necessariamente lo zaino ottimo del sottoproblema in cui ci sono ancora tutti gli elementi tranne i e $P' = P - p_i$ (lo zaino è senza ripetizione!).
- Se così non fosse perché esiste un sottozaino migliore Z'', allora $Z^* = Z'' \cup x_i$ sarebbe migliore di Z contro l'ipotesi iniziale.

Esempi di applicazione

Problema dello ZAINO

Fase 2: Definizione ricorsiva del valore di una soluzione ottima.

- Sia $K[j, w] \stackrel{\text{def}}{=}$ il massimo valore ottenibile usando uno zaino di capacità w e i primi j oggetti $(1, \ldots, j)$.
- Il problema chiede di determinare il $K[n, P]$.
- Si considera allora un generico $K[j, w]$ e lo si esprime in termini di $K[]$ con argomenti di dimensioni inferiori.
  - Dato l'elemento j si deve decidere se tenerlo o non tenerlo nello zaino.
  - Una strada esplora quale valore finale di zaino si ottiene se si tiene: $K[j-1, w-p_j] + v[j]$;
  - L'altra esplora quale altro valore si ottiene se si scarta: $K[j-1, w]$.
  - Si prende poi il massimo tra i due:
    $K[j, w] = \max\{K[j-1, w-p_j] + v[j], K[j-1, w]\}$.

Esempi di applicazione

Problema dello ZAINO

Fase 3: Calcolo del valore di una soluzione ottima secondo uno schema bottom-up.

- Le soluzioni $K[j, w]$ formano una tabella bidimensionale.
- Tale tabella può essere completata riga per riga (da 1 a n) o colonna per colonna (da 1 a P).
- Rimane solo da definire la base della ricorsione: $K[\cdot, 0]$ e $K[0, \cdot]$:
  - $K[0, w] = 0$ in quanto per qualsiasi capacità w con 0 oggetti non si ha nessun valore;
  - $K[j, 0] = 0$ perché lo zaino non può contenere nulla.

Esempi di applicazione

Problema dello ZAINO

Procedura `ZAINO`(P, p, v)

```
n = |v|;
for i = 0; i <= P; i++) do K[0][i] = 0;
for i = 0; i <= n; i++) do K[i][0] = 0;
for (j = 1; j <= n; j++) do
    for (w = 1; w <= P; w++) do
        if (p[j] > w) then K[j, w] = K[j − 1, w]; // j troppo pesante!
        else K[j, w] = max(K[j − 1, w], K[j − 1, w − p[j]] + v[j]);
    endfor
endfor
return K[n, P];
```

Esempi di applicazione

Problema dello ZAINO

Correttezza

Per costruzione.

Complessità

Si verifica facilmente che è $O(nP)$.

Fase 4: Costruzione della soluzione ottima: lasciato per esercizio.

Esempi di applicazione

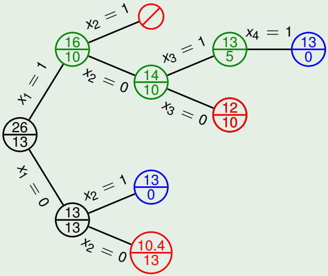
Problema dello ZAINO

Esempio 4

Istanza del Problema dello ZAINO

$v = [6, 13, 4, 3]$, $p = [3, 13, 5, 5]$ e $P = 13$.

Soluzione branch & bound:



Soluzione programmazione dinamica:

v	p	P	1	2	3	4	5	6	7	8	9	10	11	12	13
v	p	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	3	0	0	0	6	6	6	6	6	6	6	6	6	6	6
13	13	0	0	0	6	6	6	6	6	6	6	6	6	6	13
4	5	0	0	0	6	6	6	6	6	10	10	10	10	10	13
3	5	0	0	0	6	6	6	6	6	10	10	10	10	10	13

Casi comuni

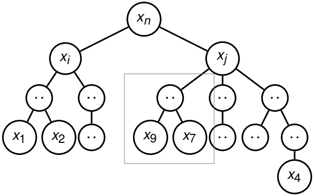
Sottoproblemi comuni

- Aspetto cruciale per applicare la programmazione dinamica è caratterizzare in modo corretto come scomporre il problema in sottoproblemi.
- Servono creatività e esperienza.
- L'esperienza di alcuni autori suggerisce però che esistono 4 modi con cui costruire i sottoproblemi che emergono più di altri:
  1. Input: $x_1, x_2, \ldots, x_n$. Il sottoproblema considera l'input dal primo elemento fino all'i-esimo: $x_1, x_2, \ldots, x_i$.
     Numero dei sottoproblemi: $O(n)$.
  2. Input: $x_1, x_2, \ldots, x_n$ e $y_1, y_2, \ldots, y_m$. Il sottoproblema considera l'input dal primo elemento fino a un certo indice per entrambe le stringhe: $x_1, x_2, \ldots, x_i$ e $y_1, y_2, \ldots, y_j$.
     Numero dei sottoproblemi: $O(nm)$.

Casi comuni

Sottoproblemi comuni

3. Input: $x_1, x_2, \dots, x_n$. Il sottoproblema considera una parte dell'input che non inizia dal primo elemento:
   $x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_j$.
   Numero dei sottoproblemi: $O(n^2)$. È sufficiente costruire una tabella $n \times n$ in cui le righe rappresentano i e le colonne j. Che forma assume la matrice?
4. L'input è un albero con radice di ordine n. Il sottoproblema è un sottoalbero con radice.



Numero dei sottoproblemi: $O(n)$.

Memoization

Cos'è

- Memoization, termine senza 'r' coniato da Donald Michie nel 1968, indica una tecnica di ottimizzazione per accelerare l'esecuzione dei programmi:
  Quando ci sono più chiamate di funzioni con medesimi parametri, si memorizzano i risultati delle chiamate per poi riusarli anziché eseguire nuovamente le funzioni.
- Si differenzia dal *caching* perché è esplicito nel contesto di un programma.
- Una funzione scritta usando la memoization deve usare una tabella di supporto globale per memorizzare i risultati delle chiamate in modo da poterli eventualmente riusare nelle chiamate successive.
- La tabella solitamente è una hash table in cui la chiave è data dai parametri presenti nella chiamata della funzione.

Memoization

Esempio

K è una tabella hash globale con chiave (j, w).
P, v e p sono variabili globali.

Procedura `ZAINO`(i, w)

// *i* indica l'elemento in esame. *w* la capacità dello zaino

if K.`containsKey`(i, w) **then return** K.`get(i,w)`;
if $(p[i] > w)$ **then** $k =$ `ZAINO(i-1,w)`; // *i* troppo pesante!
else $k = \max($`ZAINO(i-1,w)`, `ZAINO(i-1,w-p[i])` $+ v[i])$;
K.`put`$((i, w), k)$;
return k;

Complessità: $O(nP)$.
Medesima complessità dell'algoritmo costruito con la programmazione dinamica: le costanti moltiplicative sono però più grandi per l'overhead delle chiamate ricorsive.

Memoization

Quando si usa?

- Programmazione dinamica: risolve tutte le sottoistanze di un problema ricorsivo in modalità bottom-up.
- *divide et impera*: risolve solo le sottoistanze di un problema ricorsive utili per la soluzione finale ma risolve le sottoistanze comuni ogni volta che le “incontra”.
- Memoization esegue lo stesso numero di chiamate della *divide et impera* per le sottoistanze “nuove” riusando i risultati per quelle già incontrare.
- Un algoritmo con memoization potrebbe quindi essere più efficiente di un algoritmo scritto con la programmazione dinamica.
- In generale l’overhead delle chiamate ricorsive è tale che l’algoritmo scritto con la programmazione dinamica è comunque più efficiente se risolve un numero limitato di sottoistanze non necessarie.

Esercizi

Massima sottosequenza crescente

Esercizio 1

Implementare l'algoritmo `MSC()` in linguaggio Java e si verichi quanti cicli interni si eseguono con diversi tipi di sequenze.

Esercizi

Massima sottosequenza crescente

Esercizio 2

La complessità dell'algoritmo `MSC()` è $O(|E|)$. Esprimere tale complessità in funzione della dimensione della sequenza in input e descrivere un esempio di caso peggiore.

Esercizi

String Matching Approssimato

Esercizio 3

Completare l'algoritmo `MSA()` in modo che in output stampi la sottosequenza di T per la quale la k-approssimazione è minima.

Esercizi

ZAINO con ripetizione

Esercizio 4

Completare l'algoritmo `ZAINO()` in modo che in output stampi anche il contenuto dello zaino.

Esercizi

ZAINO con ripetizione

Esercizio 5

Scrivere l'algoritmo risolutore per il problema dello ZAINO con ripetizione usando la programmazione dinamica.