# Esercizi Capitolo 13 - Programmazione dinamica

Alberto Montresor

15 Giugno, 2021

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

# 1 Problemi

## 1.1 $k$-occorrenza (Esercizio 13.1 del libro)

Data la tabella $D$ calcolata da stringMatching(), scrivere una procedura che stampa l'occorrenza $k$-approssimata di $P$ che compare in $T$ e termina in $t_j$.

**Soluzione:** Sezione 2.1

## 1.2 Algoritmo di Bellman, Ford e Moore (Esercizio 13.3 del libro)

Si consideri il problema dei cammini minimi da un singolo nodo $r$ a tutti i rimanenti $n-1$ nodi in un grafo orientato pesato (con eventuali lunghezze negative degli archi). Si definisca $d_v^k$ come la lunghezza di un cammino minimo dal nodo $r$ al nodo $v$ che contiene al più $k$ archi. Si dimostri che l'algoritmo di Bellman-Ford-Moore equivale a risolvere le seguenti relazioni di programmazione dinamica:

$$d_v^{k+1} = \min\{d_v^k, \min_{h \neq v}\{d_h^k + w(h,v)\}\}.$$

Qual è la complessità dell'algoritmo nella versione di programmazione dinamica?

**Soluzione:** Sezione 2.2

## 1.3 Cammini minimi su grafi aciclici (Esercizio 13.4 del libro)

Si consideri il problema dei cammini minimi da un singolo nodo $r$ a tutti i rimanenti nodi in un grafo (con eventuali lunghezze negative degli archi) nel quale esiste l'arco $(u,v)$ solo se $u < v$. Si dimostri che le relazioni di programmazione dinamica dell'Esercizio 2.2 diventano semplicemente

$$d_v = \min_{h<v}\{d_h + w(h,v)\},$$

dove $d_v$ è la lunghezza di un cammino minimo dal nodo $r$ al nodo $v$. Qual è la complessità dell'algoritmo?

**Soluzione:** Sezione 2.3

## 1.4 Cammini minimi dinamici (Esercizio 13.5 del libro)

Un algoritmo dinamico è un algoritmo che è in grado di aggiornare il risultato di una funzione a fronte di modifiche dei dati di ingresso senza dover ricalcolare tutto da capo. Sia $G = (V, E)$ un grafo orientato con funzione costo $w : V \times V \to \mathbf{R}$. Progettare un algoritmo dinamico che è in grado di aggiornare le distanze fra una sorgente $s$ e tutte gli altri vertici del grafo a fronte dell'inserimento di un nuovo arco nel grafo. Se l'inserimento dell'arco introduce un ciclo negativo nel grafo, l'algoritmo deve essere in grado di lanciare un'eccezione. Quali difficoltà sorgerebbero se volessimo cancellare archi invece di inserirli?

**Soluzione:** Sezione 2.4

## 1.5 Massima sottosequenza crescente (Esercizio 13.6 del libro)

Sia data una sequenza $V$ di $n$ numeri interi distinti. Si scriva una procedura efficiente basata sulla programmazione dinamica per trovare la più lunga sottosequenza crescente di $V$ (per esempio, se $V = 9, 15, 3, 6, 4, 2, 5, 10, 3$, allora la più lunga sottosequenza crescente è: $3, 4, 5, 10$).

**Soluzione:** Sezione 2.5

## 1.6 Harry Potter (Esercizio 13.11 del libro)

Al celebre maghetto Harry Potter è stata regalata una scopa volante modello Nimbus3000 e tutti i suoi compagni del Grifondoro gli chiedono di poterla provare. Il buon Harry ha promesso che nei giorni a venire soddisferà le richieste di tutti, ma ogni ragazzo è impaziente e vuole provare la scopa il giorno stesso. Ognuno propone ad Harry un intervallo di tempo della giornata durante il quale, essendo libero da lezioni di magia, può fare un giro sulla scopa, e per convincerlo gli offre una certa quantità di caramelle Tuttigusti+1. Allora Harry decide di soddisfare, tra tutte le richieste dei ragazzi, quelle che gli procureranno la massima quantità di caramelle (che poi spartirà coi suoi amici Ron e Hermione). Aiutalo a trovare la migliore soluzione possibile.

**Soluzione:** Sezione 2.6

## 1.7 Mosse su scacchiera

Supponete di avere una scacchiera composta da $n \times n$ caselle e un pedone che dovete muovere dall'estremità inferiore a quella superiore. Il pedone si può muovere (1) una casella in alto, oppure (2) una casella in diagonale alto-destra, oppure (3) una casella in diagonale alto-sinistra. Non si può tornare indietro. Le caselle sono denotate da una coppia di coordinate $(r, c)$ (riga, colonna).
Alla scacchiera è associata una matrice $P$; quando una cella $(r, c)$ viene visitata, si guadagna un profitto $P[r][c]$.
Potendosi muovere di una riga alla volta, un *percorso dall'estremità inferiore all'estremità superiore* contiene esattamente $n$ celle; il suo *guadagno totale* è dato dalla somma dei profitti delle celle che lo compongono.
Data in input una matrice $P$, scrivere un algoritmo che restituisca il massimo guadagno ottenibile partendo da una qualunque cella dell'estremità inferiore e raggiungendo una qualunque cella dell'estremità superiore, seguendo le regole appena descritte.

Esempio di tabella $P$ per una scacchiera $5 \times 5$; sono evidenziati (uno in grassetto e uno con sottolineatura) due percorsi ottimi dalla riga inferiore a quella superiore, entrambi con valore 33.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 6 | **7** | 4 | 7 | 8 |
| 2 | 7 | **6** | 1 | 1 | 4 |
| 3 | 3 | 5 | **7** | 8 | 2 |
| 4 | 2 | **6** | 7 | 0 | 2 |
| 5 | **7** | 3 | 5 | 6 | 1 |

**Soluzione:** Sezione 2.7

## 1.8 Batterie

Siete a bordo di una modernissima auto elettrica su un'autostrada. Entrate in autostrada al km 0 con la batteria carica e dovete uscire al km $L$.
Prima che la vostra batteria si esaurisca (dopo $r$ km), dovete fermarvi in un'area di servizio e sostituirla con una batteria di ricambio.
Siano $D[1 \ldots n]$ e $C[1 \ldots n]$ due vettori di interi, dove $D[i]$ è la distanza dell'area di servizio $i$-isima dall'inizio dell'autostrada, e $C[i]$ è il costo di una nuova batteria nell'area $i$.
Il costo totale del viaggio è dato dalla somma dei costi delle batterie sostituite per arrivare al km $N$. Scrivete un algoritmo che prenda in input $N$, $r$, $n$, $D$ e $C$ e restituisca il costo totale minimo. Discutere correttezza e complessità.
Nota: Negli esercizi aggiuntivi del Capitolo 14 relativi alla tecnica greedy si trova una formulazione diversa di questo problema.

**Soluzione:** Sezione 2.8

## 1.9 Versione annotata di LCS

Quali sono i vantaggi e gli svantaggi della versione basata su *memoization* del problema Longest Common Subsequence (LCS)?

**Soluzione:** Sezione 2.9

## 1.10 Come passare l'esame!

Si consideri un esame con $n$ problemi, ognuno dei quali vale $v[i]$ punti e richiede $t[i]$ minuti per essere risolto.
Il vostro obiettivo è di passare l'esame con il minimo sforzo possibile. Trovate quindi un algoritmo che dati $v[1 \ldots n]$, $t[1 \ldots n]$ e un valore $V$ che rappresenta la sufficienza, trova il sottoinsieme di problemi che richiede il minor tempo possibile e permette di passare l'esame.

1. Sia $T(i, v)$ il numero minimo di minuti richiesti per ottenere almeno $v$ risolvendo un qualunque sottoinsieme dei primi $i$ problemi. Scrivere un'espressione ricorsiva per $T(i, v)$.
2. Scrivere l'algoritmo basato su programmazione dinamica.
3. Valutare la complessità dell'algoritmo proposto.
4. Supponete ora che risolvere un esercizio in maniera parziale dia la possibilità di ottenere una porzione di voto proporzionale. Descrivete un algoritmo di costo $O(n \log n)$ per risolvere lo stesso problema.

**Soluzione:** Sezione 2.10

## 1.11 Insieme indipendente di intervalli pesati

Nel problema dell'insieme indipendente di intervalli pesati, viene descritto brevemente un metodo per trovare i valori $p_i$, definiti nel modo seguente: $p_i = j$ se e solo se $j < i$ è il massimo indice tale che $[a_j, b_j[$ non interseca $[a_i, b_i[$ (se non esiste $j$, allora $p_i = 0$).
Progettare un metodo alternativo che faccia uso della ricerca binaria.

**Soluzione:** Sezione 2.11

## 1.12 Ma è italiano?

Ricevete una stringa di caratteri $S[1 \ldots n]$; avete il dubbio che sia una stringa di testo italiano, in cui sono stati tolti tutti gli spazi e i segni di interpunzione. Ad esempio, "eraunanottebuiaetempestosa". Avete a disposizione una struttura dati dizionario $d$ che contiene... un dizionario della lingua italiana! $d.\mathsf{contains}(w)$ ritorna **true** se la parola $S[i \ldots j]$ (contenuta fra l'indice $i$ e l'indice $j$) è una parola italiana. Potete anche assumere di avere a disposizione una funzione $\mathsf{contains}(S, i, j)$ che restituisce il testo compreso fra gli indici $i$ e $j$, estremi inclusi.
Scrivere un algoritmo che stabilisca se la frase che ricevete in input è un testo italiano e stampi la frase risultante, con le parole separate da spazi.
Discutere correttezza e complessità dell'algoritmo proposto, quest'ultima misurata come numero di chiamate a $contains()$.

**Soluzione:** Sezione 2.12

## 1.13 Palinsesto

Ogni giorno, il conduttore di una radio deve compilare il palinsesto del suo programma musicale, prevedendo la programmazione di alcune delle canzoni dalla lista di canzoni della stazione, indicizzate con i numeri da 1 a $n$, in modo da massimizzare il gradimento dei suoi ascoltatori. La radio lancia ogni giorno un sondaggio col quale gli ascoltatori possono votare le loro canzoni preferite. Il conduttore dispone quindi della funzione $g[i], i = 1 \ldots n.$ che contiene il risultato del giorno precedente, in forma di valori reali da 0 (nessun gradimento) ad 1 (massimo gradimento). Il programma radiofonico ha la durata di $N$ minuti e ciascuna canzone ha la sua durata $d[i], i = 1 \ldots n.$ Definire un algoritmo che presi i dati descritti sopra, fornisca in uscita il palinsesto per il programma che massimizza il gradimento del pubblico (nota che il palinsesto non può contenere piu occorrenze della stessa canzone).

**Soluzione:** Sezione 2.13

## 1.14 LCS di lunghezza $k$

Scrivere le ricorrenze di Programmazione Dinamica che, date due sequenze $X = x_1, x_2, \ldots, x_n$ e $Y = y_1, y_2, \ldots, y_m$ di caratteri e un numero intero (positivo) $k$, dicano se esiste una sottosequenza comune (LCS) di $X$ e $Y$ di lunghezza maggiore o uguale a $k$.

**Soluzione:** Sezione 2.14

## 1.15 Stringhe palindrome

Una stringa si dice palindroma se è uguale alla sua trasposta, cioè se è identica se letta da sinistra e destra o da destra a sinistra: per esempio, "alla" e "ara" sono palindrome.
Scopo dell'esercizio è scrivere una funzione $\mathsf{minpal}(\textsc{Item}[\,]\ s,$ **int** $n)$ che ritorna il numero minimo di caratteri da *inserire* in $s$ necessari per rendere $s$ palindroma.
Per esempio, quanti caratteri sono necessari per rendere palindroma la parola "casacca"?

- Una soluzione banale potrebbe essere quella di "specchiare" la parola, aggiungendo $n = 7$ caratteri: "casaccaACCASAC" (le lettere inserite sono scritte in maiuscolo).
- Migliorando un poco, è possibile specchiare la parola sull'ultima lettera: "casaccaCCASAC", che richiede $n - 1 = 6$ caratteri.
- Si potrebbe notare che "casacca" contiene "acca" che è già palindroma. Si può sfruttare questo fatto, inserendo 3 caratteri: "casaccaSAC";
- Infine, si potrebbe notare che "casac" è ancora palindroma, e quindi bastano due caratteri: "ACcasacca".

Notate che non necessariamente i caratteri si inseriscono in testa o in fondo; per esempio, "anta" può essere resa palindroma con un carattere: "antNa".
Scrivere un algoritmo

$$\textbf{int}\mathsf{minpal}(\textsc{Item}[\,]\ s, \textbf{int}\ n)$$

. Discutere la complessità dell'algoritmo.

**Soluzione:** Sezione 2.15

## 1.16 Abbasso Trenitalia!

Siamo nel 2020. Le ferrovie tedesche-austriache (DB,OBB), dopo aver conquistato la tratta del Brennero, hanno fatto fallire Trenitalia e dominano il mercato italiano. La loro politica dei prezzi è la seguente: è possibile acquistare un abbonamento per $X$ euro con validità un mese a partire dal giorno di emissione, valido su tutte le tratte. Oppure pagare un biglietto a tariffa intera per un particolare viaggio. Siete un rappresentantante di commercio e avete programmato una serie di viaggi per il prossimo anno, nei giorni $d_1, \ldots, d_n$. La tariffa intera per ognuno di questi viaggi è $f_1, \ldots, f_n$. Scrivete un algoritmo che minimizzi il costo totale, stampando i giorni in cui comprare gli abbonamenti.
Valutare la complessità e discutere la correttezza del vostro algoritmo.

**Soluzione:** Sezione 2.16

## 1.17 Data center

Siete i sistemisti di un data center in grado di "macinare" diversi terabyte di dati al giorno. In un periodo di $n$ giorni, vi viene comunicato in anticipo la quantità $x_i$ di dati da gestire nel giorno $i$. I dati non macinati nel giorno $i$ non possono essere gestiti il giorno successivo.
Purtroppo, la scelta del sistema operativo non è stata fra le più felici. Infatti le performance del sistema peggiorano di giorno in giorno (forse per memory leaks?), fino a quando non decidete di fare reboot. Dopo $j$ giorni dall'ultimo reboot, la quantità massima di terabyte "macinabili" è pari a $s_j$, con $s_1 > s_2 > \ldots > s_n$. Peggio ancora, per fare reboot ci vuole un giorno intero e durante quel giorno non è possibile gestire alcun dato.
Dato un vettore $x[1 \ldots n]$ che misura la quantità di dati in input e un vettore $s[1 \ldots n]$ che misura le performance di sistema dall'ultimo reboot, scrivere un programma che calcola il numero massimo di terabyte che possono essere processati dal sistema, e valutarne la sua complessità.
Esempio:

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| x | 10 | 1 | 7 | 7 |
| s | 8 | 4 | 2 | 1 |

La soluzione ottimale consiste nel fare un singolo reboot nel giorno 2. Si processano quindi 8TB nel giorno 1, 7TB nel giorno 3, 4TB nel giorno 4 per un totale di 19TB.

**Soluzione:** Sezione 2.17

## 1.18 Stringhe primitive

Dato un insieme di $m$ stringhe dette *primitive* ed una stringa $X = x_1 \dots x_n$, si vuole determinare se $X$ è ottenibile dalla concatenazione di stringhe primitive. Ad esempio: dato l'insieme di primitive $\{01, 10, 011, 101\}$, per la stringa $X = 0111010101$ la risposta è sì (due possibili soluzioni sono 011-10-10-101 e 011-101-01-01) mentre per la stringa $X = 0110001$ la risposta è no.

- Descrivere in pesudo–codice un algoritmo che risolve il problema. Spiegare perché l'algoritmo proposto è corretto. (Suggerimento: programmazione dinamica).
- Modificare l'algoritmo proposto in modo che, nel caso $X$ sia ottenibile dalla concatenazione di primitive, vengano prodotta in output la stringa originale in cui la separazione fra stringhe primitive sia evidenziata da "-". Ad esempio: data la stringa $X = 0111010101$ e le primitive $Y_1 = 01$, $Y_2 = 10$, $Y_3 = 011$ e $Y_4 = 101$, l'algoritmo deve produrre in output la sequenza 011-10-10-101 o la sequenza 011-101-01-01.

**Soluzione:** Sezione 2.18

# 2 Soluzioni

## 2.1 $k$-occorrenza (Esercizio 13.1 del libro)

Come suggerito nel libro, si cerca l'indice $j$ tale per cui $DP[m][j]$ è minimo. A partire da quella posizione, si analizzano la tabella $D$ per capire se (1) i caratteri coincidono, (2) bisogna cancellare un carattere dal testo, (3) bisogna inserire un carattere del pattern, (4) bisogna sostituire un carattere del testo con uno del pattern.

```
printApprox(ITEM[] P, ITEM[] T, int[][] D, int m, int n)
int i = m
int j = 2
for k = 2 to n do
    if DP[m][k] < DP[m][j] then
        j = k
while i ≥ 0 and j ≥ 0 do
    if P[i] == T[j] then
        print P[i]
        i = i − 1; j = j − 1
    else if DP[i][j] == DP[i][j − 1] + 1 then
        print Cancellazione T[j]
        j = j − 1
    else if DP[i][j] == DP[i − 1][j] + 1 then
        print Inserimento P[i]
        i = i − 1
    else
        print Sostituzione T[j], P[i]
        i = i − 1; j = j − 1
```

## 2.2 Algoritmo di Bellman, Ford e Moore (Esercizio 13.3 del libro)

Rifacendosi alla nozione di passata presente nel libro, è facile vedere che alla zeresima passata, si ottiene l'unico cammino di contenente 0 archi; alla passata $k$-esima, si ottengono tutti i cammini contenenti al più $k$-archi.
La versione di camminiMinimi() basata sulla programmazione dinamica è organizzata come segue: un ciclo che ripete $n - 1$ volte la verifica, su tutti gli archi, se è possibile migliorare l'attuale distanza minima dalla sorgente. Il costo è quindi $O(mn)$, identico a quello ottenuto adattando il nostro algoritmo prototipo.

```
camminiMinimi(GRAPH G, NODE r, int[] T)
int[] d = new int[1 ... G.n]                    % d[u] è la distanza da r a u
foreach u ∈ G.V() − {r} do
    T[u] = nil
    d[u] = +∞
T[r] = nil
d[r] = 0
for i = 1 to G.n − 1 do
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            if d[u] + w(u, v) < d[v] then
                T[v] = u
                d[v] = d[u] + w(u, v)
```

## 2.3 Cammini minimi su grafi aciclici (Esercizio 13.4 del libro)

In un grafo aciclico, non esistono cicli per definizione; in altre parole, è impossibile ritornare ad un nodo, dopo che questo è stato visitato. Visitiamo quindi i nodi a partire dal nodo 1, il primo ad apparire in ordine topologico, seguendo la direzione degli archi.
Al momento della visita del nodo $u$, tutti i cammini che possono portare ad $u$ sono già stati visitati; il valore $d[u]$ è quindi definitivo, e può essere utilizzato per migliorare la stima $d[v]$ per tutti gli archi $(u, v) \in E$. Il costo dell'algoritmo risultante è $O(m + n)$.

```
camminiMinimi-DAG(GRAPH G, int[] T)
int[] d = new int[1 ... G.n]                    % d[u] è la distanza da 1 a u
for u = 2 to G.n do
    T[u] = nil
    d[u] = +∞
T[1] = nil
d[1] = 0
for u = 1 to G.n − 1 do
    foreach v ∈ G.adj(u) do
        if d[u] + w(u, v) < d[v] then
            T[v] = u
            d[v] = d[u] + w(u, v)
```

## 2.4 Cammini minimi dinamici (Esercizio 13.5 del libro)

Si lavora a partire dall'algoritmo di Bellman-Ford-Moore visto nella sezione 2.2; tutte le volte, che viene aggiunto un arco, si prova a verificare se il nuovo arco $(u, v)$ permette di raggiungere $v$ a partire da un nodo $x$ passando per $u$, con un costo inferiore all'attuale valore $d[x, v]$. Se questo avviene, si aggiorna il valore.
Cosa succede nel caso si voglia invece togliere un arco? In questo caso, quest'arco può essere utilizzato in tanti cammini minimi, il problema è identificare quali...

## 2.5 Massima sottosequenza crescente (Esercizio 13.6 del libro)

Denotiamo con $DP[i]$ la dimensione della massima sottosequenza crescente che termina nella posizione $i$-esima. È possibile calcolare $DP[i]$ in maniera ricorsiva. Si consideri l'indice $i$ e si considerino gli elementi minori di $V[i]$ nel sottovettore $V[1 \ldots i-1]$; $V[i]$ può essere utilizzato per estendere la più lunga sottosequenza che termina in uno di questi elementi. Se non esistono elementi minori, allora dobbiamo "ricominciare da capo", ovvero considerare la sequenza composta dal singolo valore $V[i]$.
Un modo per esprimerlo è il seguente:

$$DP[i] = \begin{cases} 1 & i = 1 \\ 1 & i > 1 \textbf{ and } \forall j, 1 \leq j < i : V[i] < V[j] \\ \max\{DP[j] : 1 \leq j \leq i-1 \wedge V[j] < V[i]\} + 1 & \text{altrimenti} \end{cases}$$

Il codice per risolvere questo problema è quindi il seguente, dove il vettore $P[i]$ memorizza l'indice precedente nella sequenza che termina in $V[i]$. La funzione calcola tutti i valori $DP[i]$ e $P[i]$ con $i = 1 \ldots n$, in ordine. Tutte le volte che viene aggiornato $DP[i]$, viene aggiornato anche $P[i]$. La variabile *max* mantiene l'indice del valore in cui termina la più lunga sottosequenza trovata finora. Al termine del calcolo, viene chiamata la procedura ricorsiva printLongest(), che stampa prima la sottosequenza che termina nel valore di indice $P[i]$ precedente a $i$, e poi stampa $V[i]$ stesso. Il costo della procedura è $O(n^2)$.

```
longestIncreasingSequence(int[] V, int n)
    int[] DP = new int[1...n]
    int[] P = new int[1...n]
    int max = 1                          % Indice del valore più alto in D
    for i = 1 to n do
        DP[i] = 1                % Se non troviamo valori minori, consideriamo solo questo elemento
        P[i] = 0                                % la sottosequenza composta dal solo valore v[i]
        for j = 1 to i - 1 do
            if V[j] < V[i] and DP[j] + 1 > DP[i] then
                DP[i] = DP[j] + 1
                P[i] = j
                if DP[i] > DP[max] then
                    max = i
    printLongest(v, P, max)
```

```
printLongest(int[] V, int[] P, int i))
    if i > 0 then
        printLongest(V, P, P[i])
        print V[i]
```

## 2.6 Harry Potter (Esercizio 13.11 del libro)

Il problema è analogo a quello dell'insieme indipendente di intervalli pesati.

## 2.7 Mosse su scacchiera

Sia $c_n \in [1 \ldots n]$ la colonna di una cella che si trovi sulla riga in basso della scacchiera ($r = n$); è possibile partire da una qualunque di tale celle. Dobbiamo poi scegliere $n-1$ coordinate $c_{n-1}, \ldots, c_2, c_1$ che rappresentano mosse valide che ci portano alla prima riga della scacchiera ($r = 1$).

Definiamo con il termine *percorso* una sequenza $C = (c_n, \ldots, c_1)$ di mosse valide sulla scacchiera che ci portano da $(n, c_n)$ a $(1, c_1)$. Il guadagno totale per il percorso $C$ è pari a:

$$G(C) = \sum_{r=1}^{n} P[r][c_r]$$

A questo punto, dobbiamo verificare se il nostro problema ha sottostruttura ottima. Dato un percorso $C = c_n, \ldots, c_1$, definiamo $C[r] = (c_r, c_{r-1}, \ldots, c_1)$ il sotto-percorso che porta da $(r, c_r)$ a $(1, c_1)$. Diciamo che un percorso $C[r]$ da $(r, c_r)$ è ottimo se non esiste un percorso $C'[r] = (c_r, c'_{r+1}, \ldots, c'_n)$ che parta dalla cella $(r, c_r)$ e arrivi ad una cella della riga 1 con guadagno superiore.

**Teorema 1.** Sia $C[r] = (c_r, c_{r-1}, \ldots, c_1)$ un percorso ottimo dalla cella $(r, c_r)$ ad una cella della riga 1. Allora $C[r-1]$ è un percorso ottimo dalla cella $(r-1, c_{r-1})$ alla riga $n$.

*Dimostrazione.* Supponiamo che esista un percorso $C'[r-1] = c_{r-1}, c'_{r-2}, \ldots, c'_1$ fra $(r-1, c_{r-1})$ e la riga 1 che abbia un guadagno superiore a $C[r-1]$:

$$G(C'[r-1]) > G(C[r-1])$$

Allora $C''[r] = (c_r, c_{r-1}, c'_{r-2}, \ldots, c'_1)$ è un percorso fra $(r, c_r)$ e la riga 1 che ha guadagno superiore a $C[r]$:

$$G(C''[r]) = G(C'[r-1]) + P[r][c_r] > G(C[r-1]) + P[r][c_r] = G(C[r])$$

Ma questo è assurdo per definizione di $C$. □

Costruiamo un vettore $DP[1 \ldots n][1 \ldots n]$, in cui l'elemento $DP[r][c]$ rappresenta il guadagno massimo che si ottiene partendo da $(r, c)$ e arrivando ad una qualsiasi cella della riga destinazione ($r = 1$). È possibile definire $g$ ricorsivamente in questo modo:

$$DP[r][c] = \begin{cases} -\infty & c < 1 \textbf{ or } c > n \\ P[1][c] & r = 1 \textbf{ and } 1 \le c \le n \\ \max \begin{pmatrix} DP[r-1][c-1], \\ DP[r-1][c], \\ DP[r-1][c+1] \end{pmatrix} & 2 \le r < n \textbf{ and } 1 \le c \le n \end{cases}$$

- Se la colonna non è valida, restituiamo il valore $-\infty$ che non verrà mai considerato nella massimizzazione;
- Sulla prima riga, il massimo guadagno ottenibile a partire dalla cella $(r, c)$ è il profitto $P[r][c]$, perché abbiamo già raggiunto la riga 1.
- In generale, sulla riga $r$ si prende il massimo profitto ottenibile da una delle tre celle della riga precedente $r-1$ che sono raggiungibili dalla posizione $(r, c)$.

Una versione ricorsiva basata su questa definizione ovviamente è inefficiente, perché i sottoproblemi sono ripetuti.
Calcoliamo il vettore $DP$ in modo iterativo.

```
int searchPath(int[][] P, int n)
    int DP = new int[1...n][1...n]
    for c = 1 to n do                                  % Copia la prima riga nel vettore DP
        DP[1][c] = P[1][c]
    for r = 2 to n do
        for c = 1 to n do
            DP[r][c] = -∞
            foreach d ∈ {-1, 0, +1} do
                int newc = c + d
                if 1 ≤ newc ≤ n then
                    DP[r][c] = max(DP[r][c], DP[r - 1][newc] + P[r][c])
    return max(DP[n])                                  % Restituisce il massimo dell'ultima riga
```

E' anche possibile, sebbene non richiesto, stampare uno dei percorsi ottimali utilizzando semplicemente la tabella $DP$, in modo ricorsivo. Nella chiamata wrapper, si individua la colonna della cella con il valore più alto nella riga $n$. La funzione ricorsiva, nel caso $r > 1$, confronta il valore $DP[r][c]$ con quelli ottenuti dalle caselle $DP[r-1][c-1]$, $DP[r-1][c]$, $DP[r-1][c+1]$ a cui viene sommato $P[r][c]$. Se coincidono, viene stampato il percorso ottimale da $DP[r-1][c']$ e poi $(r, c)$. Nel caso base ($r = 1$), viene stampato solo la casella $(r, c)$.

```
printPath(int[][] P, int[][] DP, int n)
    int c = argmax(DP[n])                              % Returns the column with maximum value
    printPathRec(P, DP, n, c)
```

```
printPathRec(int[][] P, int[][] DP, int r, int c)
    if r > 1 then
        if DP[r][c] == DP[r - 1][c - 1] + P[r][c] then
            printPathRec(P, DP, r - 1, c - 1)
        else if DP[r][c] == DP[r - 1][c] + P[r][c] then
            printPathRec(P, DP, r - 1, c)
        else
            printPathRec(P, DP, r - 1, c + 1)
    print (r, c)
```

Alcune considerazioni sulla complessità:

- Il costo della funzione searchPath() è $\Theta(n^2)$.
- Il costo della funzione printpath() è $\Theta(n)$.
- Se non è necessario stampare il percorso, ma solo trovare il valore massimo, allora due righe di dimensione $n$ sono sufficienti per risolvere il problema.
- La versione memoized del codice non è efficiente, perché comunque tutti i sottoproblemi dovranno essere risolti.

## 2.8 Batterie

La sottostruttura ottima è facile da dimostrare, facendo notare che se viene effettuata una ricarica alla stazione $i$, ci si riduce al problema $DP[i+1 \dots n]$ con una lunghezza della strada pari a $N - D[i]$; qualunque

soluzione ottima per questo problema fa parte della soluzione ottima del problema originale, per la proprietà taglia e cuci.
È possibile definire ricorsivamente il problema come segue. Assumiamo che esistano due stazione fittizie $0, n+1$, con $D[0] = 0$ e $D[n+1] = L$ e $C[0] = C[n+1] = 0$. Non è quindi necessario passare $L$.
Definiamo una tabella $DP[0 \dots n+1]$, tale che $DP[i]$ rappresenti il costo minimo da pagare nel caso si acquisti la batteria alla stazione $i$-esima. La nostra soluzione si trova in $DP[0]$. $DP[i]$ può essere definito in maniera ricorsiva nel modo seguente:

$$DP[i] = \begin{cases} 0 & \text{se } i = n+1 \\ \min_{j:j>i \wedge D[j] \leq D[i]+r}\{DP[j]\} + C[i]\} & \text{altrimenti} \end{cases}$$

Inviduare l'indice $j$ per cui $D[j] \leq D[i] + r$ può richiedere $O(n)$; quindi il costo pessimo di tale algoritmo è $O(n^2)$.
Notate che sarebbe possibile realizzare un algoritmo di costo $O(nr)$ creando una tabella bidimensionale $n \times r$. L'algoritmo risultante sarebbe pseudopolinomiale.
Il codice è il seguente:

```
int minCostStops(int[] D, int n, int r)
    int[] DP = new int[0 ... n+1]
    DP[n+1] = 0
    for i = n downto 0 do
        int j = i + 1
        int DP[i] = +∞
        while j ≤ n + 1 and D[j] ≤ D[i] + r do
            DP[i] = min(DP[i], DP[j])
            j = j + 1
        DP[i] = DP[i] + C[i]
    return DP[0]
```

## 2.9 Versione annotata di LCS

Se le stringhe sono ragionevolmente simili, il numero di sottoproblemi generati tenderà a scendere a $O(\max\{m, n\})$. La versione annotata risolve solo quelli. La versione normale invece costa $O(mn)$, che è lo stesso costo che si ottiene nel caso si utilizzi la versione annotata su due stringhe con alfabeti differenti.

## 2.10 Come passare l'esame!

1. 
   - Per ottenere almeno 0 o un voto negativo, posso uscire subito dall'aula:
   $$DP[i][v] = 0 : \forall i, \forall v \leq 0$$
   - Se non ho esercizi da risolvere, ma voglio ottenere $v > 0$, resterò chiuso dentro l'aula in eterno
   $$DP[0][v] = +\infty : \forall v > 0$$
   - Se considero anche l'esercizio $i$, ci possono essere due casi: se risolvo l'esercizio $i$, mi basta cercare di ottenere $v - v[i]$ con i primi $i-1$ problemi; altrimenti, devo utilizzare i primi $i-1$ problemi per ottenere $v$:
   $$DP[i][v] = min\{t[i] + DP[i-1][v - v[i]], DP[i-1][v]\}$$

2.

```
int studente-pigro(int[] v, int[] t, int n, int V)
    int[][] DP = new int[0...n][1...V]
    for v = 1 to V do
        DP[0][v] = +∞
    for i = 1 to n do
        DP[i][v] = min(T[i-1][v], t[i] + iif(v - v[i] > 0, DP[i-1][v - v[i]], 0))
    return DP[n][V]
```

3. La complessità è $O(nV)$.

4. Un approccio greedy può funzionare qui (cosÈ come funzionava con lo zaino frazionario). È sufficiente ordinare i problemi per $m[i]/v[i]$ crescente (o per $v[i]/m[i]$ decrescente). A questo punto, iteriamo sulla lista ordinata, fino a quando $V$ non é stato raggiunto. Solo l'ultimo problema può essere risolto parzialmente.

## 2.11 Insieme indipendente di intervalli pesati

L'idea è la seguente. Innanzitutto $p_1$ è uguale a 0 per definizione. $p_i$ è uguale al più grande valore $j < i$ tale per cui $b_j \leq a_i$. Quindi è sufficiente cercare $a_i$ fra i valori $b_1$ e $b_{i-1}$; se non si trova, è sufficiente ritornare il valore $b_j$ immediatamente precedente ad $a_i$. Per poter utilizzare la ricerca binaria, si assume che gli intervalli siano ordinati per estremi di fine (ordinamento che può essere ottenuto con costo $O(n \log n)$).

```
int binarySearch(ITEM[] A, ITEM v, int i, int j)
    if i > j then
        return j
    else
        int m = ⌊(i + j)/2⌋
        if A[m] == v then
            return m
        else if A[m] < v then
            return binarySearch(A, v, m + 1, j)
        else
            return binarySearch(A, v, i, m - 1)
```

```
computePredecessor(int[] a, int[] b, int[] p, int n)
    p[1] = 0
    for i = 2 to n do
        p[i] = binarySearch(b, a_i, 1, i - 1)
```

Notate la funzione di binary search: invece di ritornare 0 in caso di non successo, ritorna il valore $j$. Questo corrisponde al primo valore precedente ad $a_i$, che è esattamente il valore cercato per $p[i]$.

## 2.12 Ma è italiano?

È possibile risolvere il problema con la programmazione dinamica. Infatti è possibile individuare una *sottostruttura ottima*: dato il prefisso $S(i)$ della stringa $S$, corrispondente ai caratteri $S[1 \ldots i]$, esso è un testo italiano se esiste un indice $k < i$ tale per cui $S(k)$ è un testo italiano e $S[k+1 \ldots i]$ è una parola italiana.

Ovviamente, noi non conosciamo la posizione $k$ in cui spezzare il testo; il modo più semplice è provare tutte le posizioni. Utilizziamo un vettore $DP[1 \dots n]$, dove $DP[i]$ contiene l'indice $k$ in cui spezzare il testo $S(i)$ in due parti, tali che $S(k)$ è un testo italiano e $S[k+1 \dots i]$ è una parola italiana. Come valore speciale, memorizziamo $-1$ in $DP[i]$ se non è possibile interpretare il prefisso $S(i)$ come un testo italiano.

```
isItalian(ITEM[] S, int n)
int[][] DP = new int[0...n]
DP[0] = 0
for i = 1 to n do
    int k = i − 1
    DP[i] = −1
    while k ≥ 0 and DP[i] < 0 do
        if DP[k] ≥ 0 and d.contains(extract(S, k + 1, i)) then
            DP[i] = k
        k = k − 1
if DP[n] < 0 then
    print "Non è italiano"
else
    printItalian(S, n, DP);
```

```
printItalian(ITEM[] S, int i, int[] DP)
if DP[i] == 0 then
    print extract(S, 1, i)
else
    printItalian(S, DP[i], DP)
    print " "
    print extract(S, DP[i] + 1, i)
```

La complessità, misurata in numero di chiamate contains, è pari a $\Theta(n^2)$.
Volendo scrivere una versione più efficiente, è possibile applicare le seguenti ottimizzazioni:

- Innanzitutto, è possibile notare che tutte le parole italiane hanno una lunghezza limitata, che possiamo assumere costante. E' quindi inutile cercare sottostringhe più lunghe di tale lunghezza; limitando le ripetizioni del ciclo while, l'algoritmo proposto ha complessità lineare misurata in numero di chiamate contains (credits: Riccardo Bordon)
- Invece di una hash table, potrebbe essere utile utilizzare una struttura dati di tipo Trie, che non vediamo nel corso.

## 2.13 Palinsesto

Questo è un problema di zaino 0-1. Basta utilizzare l'algoritmo basato su programmazione dinamica visto a lezione (non lo ripeto qui). Nota: nonostante il gradimento sia un valore nell'intervallo $[0, 1]$, questo non è uno zaino frazionario: infatti le canzoni non possono essere suonate parzialmente.

## 2.14 LCS di lunghezza $k$

$$DP[i][j,k] = \begin{cases} \textbf{false} & k > 0 \wedge (i = 0 \vee j = 0) \\ \textbf{true} & k = 0 \\ DP[i-1][j-1,k-1] & i > 0 \wedge j > 0 \wedge k > 0 \wedge x_i = y_j \\ DP[i-1][j,k] \vee DP[i][j-1,k] & i > 0 \wedge j > 0 \wedge k > 0 \wedge x_i \neq y_j \end{cases}$$

## 2.15 Stringhe palindrome

È possibile notare che se il primo e l'ultimo carattere di una stringa sono uguali, è possibile "eliminarli" e concentrarsi sulla parte rimanente. Se invece sono diversi, si possono considerare due possibilità: rendere palindromo il primo carattere oppure rendere palindromo l'ultimo.
Sia $minpal(s)$ il numero di caratteri necessari per rendere palindroma la stringa $s$. $minpal(s)$ può essere calcolato in modo ricorsivo come segue:

- Se la stringa $s = \text{a}s'\text{a}$ è composta da due identici caratteri "a" iniziale e finale, allora:

$$minpal(s) = minpal(s')$$

- Se la stringa $s = \text{a}s'\text{b}$ ha due caratteri iniziale e finale diversi, aggiungiamo o un carattere "b" in testa (e consideriamo il problema $\text{a}s'$, eliminando virtualmente il carattere $b$), oppure aggiungiamo un carattere "a" in coda (e consideriamo il problema $s'\text{b}$, eliminando virtualmente il carattere $a$). Scegliamo fra le due possibilità quella con costo minore. In entrambi i casi, dobbiamo sommare 1 per il carattere aggiunto.

$$minpal(s) = \min\{minpal(\text{a}s'), minpal(s'\text{b})\} + 1$$

- Il caso base corrisponde al caso in cui $s$ contenga un carattere solo o nessuno carattere, nel qual caso la stringa è palindroma e bisogna rispondere 0.

Sia $DP[i][j]$ il numero di caratteri che è necessario inserire per rendere palindroma la stringa $s[i \ldots j]$; può essere calcolato in modo ricorsivo nel modo seguente:

$$DP[i][j] = \begin{cases} 0 & i \geq j \\ DP[i+1][j-1] & i < j \textbf{ and } s[i] = s[j] \\ \max(DP[i+1][j], DP[i][j-1]) & i < j \textbf{ and } s[j] \neq s[j] \end{cases}$$

A questo punto, cerchiamo di risolvere il problema tramite memoization. Data una stringa $s[1 \ldots n]$, utilizziamo una tabella di appoggio $DP[1 \ldots n][1 \ldots n]$.
La funzione minpalRec$(s, i, j)$ calcola il costo per rendere palindroma la sottostringa $s[i \ldots j]$. Il problema originale è ovviamente minpal$(s, 1, n)$.

```
int minpal(ITEM[] s, int n)
    int[][] DP = new int[1...n][1...n] = {-1}        % Initialized to -1
    return minpalRec(s, DP, 1, n)
```

```
int minpalRec(ITEM[] s, int[][] DP, int i, int j)
if j ≤ i then
    return 0
else if DP[i][j] < 0 then
    if s[i] == s[j] then
        DP[i][j] = minpalRec(s, i + 1, j − 1)
    else
        DP[i][j] = min(minpalRec(s, i, j − 1), minpalRec(s, i + 1, j)) + 1
    return DP[i][j]
```

Sulla complessità in tempo: nel caso pessimo, corrispondente a caratteri tutti diversi, bisogna riempire tutta la metà sopra la diagonale della matrice, con costo $O(n^2)$; nel caso ottimo, corrispondente ad una stringa palindroma, il costo sarà pari a $O(n)$.
Una procedura per stampare la risultante stringa palindroma opera ricorsivamente. Nel caso di stringa vuota, non stampare nulla. Nel caso di un carattere solo, è palindromo e va stampato. Se il primo e l'ultimo carattere sono uguali, richiama se stessa sulla parte centrale, stampando prima e dopo quel carattere. Se sono diversi, individua il quale delle due possibilità è stata scelta e chiama se stessa nella parte opportuna.

```
printPal(ITEM[] s, int[][] DP, int i, int j)
if i > j then
    % Print nothing                    % Empty string; it's palindrome, print nothing
if i == j then
    print s[i]                         % One char; it's palindrome, print it
if s[i] == s[j] then
    print s[i]
    print printPal(s, DP, i + 1, j − 1)
    print s[j]
else if DP[i][j] == DP[i + 1, j] + 1 then
    print s[i]
    print printPal(s, DP, i + 1, j)
    print s[i]
else
    print s[j]
    print printPal(s, DP, i, j − 1)
    print s[j]
```

## 2.16 Abbasso Trenitalia!

### 2.16.1 Soluzione base

Innanzitutto, assumiamo che i giorni siano rappresentati da un intero fra 1 e 365 e un abbonamento duri 30 giorni (dal giorno di inizio a giorno di inizio+29 inclusi). Assumiamo inoltre che i viaggi siano ordinati per giorno; se così non fosse, li possiamo ordinare noi al costo di $O(n \log n)$ (o meglio ancora, vedi sotto).
Una scelta greedy è la seguente: per ogni giorno $x$, valuta se il costo di un abbonamento a partire da $x$ è inferiore alla somma delle tariffe nel periodo $[x, x + 29]$; se questo accade, acquista l'abbonamento. Questo non funziona, ad esempio con 31 viaggi in giorni consecutivi, di cui il primo costa meno dell'ultimo.
Si utilizza invece la programmazione dinamica. Sia $C[i]$ il costo minimo per effettuare i viaggi $[i \dots n]$, senza che sia attivo un abbonamento precedente che copra anche il giorno $d[i]$. Il problema globale è calcolare $C[1]$.

Al viaggio $i$-esimo, è possibile (i) acquistare un abbonamento o (ii) pagare la tariffa del giorno. Nel primo caso, tutti i viaggi coperti dall'abbonamento non devono essere più considerati; si deve cercare quindi il primo viaggio $j$ dopo la scadenza dell'abbonamento e il problema si riduce ad $C[j]$. Nel secondo caso, il viaggio $i$ è pagato e il problema si riduce a $C[i+1]$. Si prende quindi il minimo fra $f[i] + C[i+1]$ e $X + C[j]$; il caso base è $C[n+1]$, ovvero il costo minimo per fare un insieme vuoto di viaggi.

$$C[i] = \begin{cases} 0 & i > n \\ \min\{f[i] + C[i+1], X + C[\min\{j : d[j] \geq d[i] + 30\}] & \text{altrimenti} \end{cases}$$

assumendo che $d[n+1] = +\infty$ che agisce come sentinella.

```
ferrovia(int[] d, int[] f, int n)
    int[] C = new int[1 ... n + 1]
    boolean[] B = new boolean[1 ... n]
    C[n + 1] = 0
    for i = n downto 1 do
        j = i
        while j ≤ n and d[j] < d[i] + 30 do
            j = j + 1
        B[i] = (f[i] + C[i + 1] > X + C[j])
        C[i] = iif(B[i], X + C[j], f[i] + C[i + 1])
    { Stampa abbonamenti } last = +∞
    for i = 1 to n do
        if B[i] and d[i] ≤ last + 29 then
            last = d[i]
            print Abbonamento d[i]
    print Costo totale: C[1]
```

Il vettore $C$ mantiene il costo per il sottoproblema $[i \ldots n]$, mentre $B[i]$ contiene **true** se e solo bisogna comprare l'abbonamento nel giorno $i$-esimo. La procedura di stampa deve tener conto che una volta preso un abbonamento, per 30 giorni non è più necessario comprarne uno. Il costo totale della procedura è $O(n^2)$, per via del doppio ciclo.

### 2.16.2 Soluzione alternativa

Si può notare che il fatto di avere 365 giorni possibili gioca a nostro vantaggio. Il nostro problema ha dimensione $n$; ma molti viaggi potrebbero cadere nello stesso giorno. Dal punto di vista di scegliere se comprare un abbonamento oppure no al giorno $i$, non importa se faccio un viaggio dal costo $x$ o un milione di viaggi il cui costo totale sia $x$; dal punto di vista degli abbonamento, non c'è distinzione fra questi due casi.
E' possibile quindi lavorare con 365 giorni, e non con $n$ viaggi; le tecniche di cui sopra vanno specificate leggermente diversamente, ma il principio è lo stesso. In particolare, i vettori $C$ e $B$ cambiano di significato: $C[i]$ è il costo minimo che si deve pagare per tutti i viaggi dal *giorno* al giorno 365; il problema iniziale si riduce a calcolare $C[1]$.

$$C[i] = \begin{cases} 0 & i > 365 \\ \min\{costi[i] + C[i+1], X + C[i+30]\} & 1 \leq i \leq 365 \end{cases}$$

```
ferrovia(int[] d, int[] f, int n)
  int[] C = new int[1...366]
  boolean[] B = new boolean[1...366]
  int[] costi = new int[1...366]
  for i = 1 to n do
      costi[d[i]] = costi[d[i]] + f[i]
  C[366] = 0
  for i = 365 downto 1 do
      B[i] = (X + C[min(i + 30, 366)] < costi[i] + C[i + 1])
      C[i] = min(X + C[min(i + 30, 366)], costi[i] + C[i + 1])
  { Stampa abbonamenti }
  i = 1
  while i ≤ 365 do
      if B[i] then
          print Abbonamento d[i]
          i = i + 30
      else
          i = i + 1
  print Costo totale: C[1]
```

In questo caso il costo è $O(n)$.

## 2.17 Data center

Il problema può essere risolto con la programmazione dinamica. Sia $DP[i][j]$ la massima quantità di terabyte che si possono gestire nei giorni $[i, \ldots, n]$, nel caso in cui l'ultimo reboot sia stato effettuato $j$ giorni prima del giorno $i$. Il problema originale corrisponde a calcolare $DP[1][1]$.
Si può facilmente vedere che $P(i, j)$ può essere calcolato ricorsivamente come segue:

$$DP[i][j] = \begin{cases} \min\{x[i], s[j]\} & i = n \\ \max\{DP[i+1][1], \min\{x[i], s[j]\} + DP[i+1][j+1]\} & i < n \end{cases}$$

Il codice che ne risulta è il seguente:

```
int datacenter(int[] x, int[] s, int n)
  int[][] DP = new int[1...n][1...n]
  for j = 1 to n do
      DP[n][j] = min(x[i], s[j])
  for i = n − 1 downto 1 do
      for j = 1 to n − 1 do
          DP[i][j] = max(DP[i + 1][1], min(x[i], s[j]) + DP[i + 1][j + 1])
  return P[1][1]
```

Tale programma lavora in tempo $O(n^2)$.

## 2.18 Stringhe primitive

Sia $DP[i]$ pari a vero o falso a seconda che la stringa $X(i)$ (prefisso $i$-esimo di $X$) sia generabile dalle primitive o meno. È facile notare che $DP[0] =$ **true** (la stringa vuota è generabile da qualunque insieme di

primitive), mentre la $DP[i] =$ **true** se e solo se esiste un indice $j < i$ tale che $DP[j] =$ **true** e la sottostringa $x_{j+1}x_{j+2}\ldots x_i = p[k]$, dove $p[k]$ è una delle stringhe primitive la cui lunghezza è $|p[k]|$. Formalmente:

$$DP[i] = \begin{cases} \textbf{true} & i = 0 \\ \exists k \in \{1, 2, \ldots, m\} : DP[i - |p_k|] \wedge x_{i-|p[k]|+1}x_{i-|p[k]|+2}\ldots x_i = p[k] & \text{altrimenti} \end{cases}$$

In base a questa rappresentazione formale, l'algoritmo può essere così descritto:

```
componibile(ITEM[] x, ITEM[][] p, int n, int m)
boolean[] DP = new int[0...n]    % Per ogni lunghezza i, riporta se è componibile o meno
int[] V = new int[0...n]         % Per ogni lunghezza i, riporta la primitiva che fa da prefisso
DP[0] = true
for i = 1 to n do
    int k = 1
    DP[i] = false
    while k < m and DP[i] = false do
        int j = i − |p[k]|
        if x_{j+1}x_{j+2}...x_i = p[k] then
            DP[i] = true
            V[i] = k
if DP[n] then
    stampa(p, V, n)
```

```
boolean stampa(ITEM[][] p, int[] V, int i)
if i == 0 then
    return false
if printPrimitive(p, V, i − |p[V[i]]|) then
    print −
print P[V[i]]
return true
```

Il vettore $V[i]$ contiene l'indice della stringa primitiva che forma il suffisso di $X(i)$. Questi valori vengono utilizzati al momento di stampare la composizione della stringa in stringhe primitive.
La complessità è $O(nl)$, dove $n$ è la lunghezza di $X$ ed $l$ è la lunghezza totale delle stringhe primitive (nel caso pessimo, tutte le stringhe primitive dovranno essere esaminate fino all'ultimo carattere); si noti che $l = \Omega(m)$, in quanto assumiamo che le stringhe abbiano almeno un carattere.

# 3 Problemi aperti

## 3.1 Calcolare il numero di parentesizzazioni

Sia $A_1 \times A_2 \times \ldots \times A_n$ un prodotto di matrici compatibili a due a due per il prodotto. Il numero di parentesizzazioni possibili è espresso da questa equazione di ricorrenza:

$$P(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 2 \end{cases}$$

Utilizzando la programmazione dinamica, scrivere un algoritmo in grado di calcolare $P(n)$ in tempo non esponenziale.

## 3.2 Longest common subsequence per tre stringhe

Scrivere un algoritmo in tempo polinomiale che prende tre stringhe $A, B, C$ come input e ritorna la più lunga sequenza $S$ che è una sottosequenza di $A, B, C$.

## 3.3 Knapsack, con poca memoria

Spiegate come risolve il problema dello zaino O/1 per uno zaino di dimensione $W$ utilizzando solo spazio $O(W)$ e tempo $O(nW)$. Non è necessario trovare la collezione di oggetti, è sufficiente trovare il valore e il peso della soluzione ottimale.

## 3.4 Esami

In un Corso di Laurea sono attivi gli esami denominati $E_1, E_2, \ldots, E_n$; ogni esame $E_i$ ha associato un certo numero (intero positivo) $c_i$ di crediti, ed è inoltre classificato come esame di tipo $A$ oppure esame di tipo $B$. Scrivere un algoritmo che utilizzi la programmazione dinamica per risolvere il seguente problema. Dato in ingresso l'elenco degli esami di un Corso di Laurea, e un intero positivo $k$, determinare se esiste un sottoinsieme di esami la cui somma dei crediti fa esattamente $k$, e inoltre contiene esattamente un esame di tipo $B$.

## 3.5 Sequenze alternanti

Una sequenza $Z = z_1, z_2, \ldots, z_k$ di interi è detta alternante se per ogni coppia di elementi consecutivi vale la proprietà che uno è positivo e l'altro è negativo:

$$\forall i \in \{1 \ldots k-1\} : z_i \cdot z_{i+1} < 0$$

Scrivere un algoritmo che, date in ingresso due sequenze di interi $X = x_1, x_2, ..., x_n$ e $Y = y_1, y_2, \ldots, y_m$, calcoli la lunghezza di una LCS alternante.