

Corso di laurea in Informatica

Progettazione d'algoritmi

Tecnica Programmazione Dinamica 3a

Angelo Monti



SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO

Dato un intero n vogliamo sapere quante sono le sequenze di cifre decimali non decrescenti lunghe n .

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 10
- per $n = 2$ la risposta deve essere 55.
Infatti alla cifra x al primo posto possono seguire $10 - x$ cifre diverse. Quindi si ha

$$\sum_{x=0}^9 (10 - x) = \sum_{i=1}^{10} i = \frac{10 \cdot 11}{2} = 55$$

Progettare un algoritmo che trova la risposta in tempo $O(n)$.

- Utilizzeremo una tabella bidimensionale di dimensioni $n \times 10$ e definiamo il contenuto delle celle come segue:
 $T[i][j]$ = numero di sequenze decimali non decrescenti lunghe i che terminano con la cifra j .
- Una volta riempita la tabella la soluzione al nostro problema sarà data dalla somma degli elementi nell'ultima riga: $\sum_{j=0}^9 T[n][j]$.

- Resta da definire la regola ricorsiva con cui calcolare i valori $T[i][j]$ della tabella

$$T[i][j] = \begin{cases} 1 & \text{se } i = 1 \\ \sum_{k=0}^j T[i-1][k] & \text{altrimenti} \end{cases}$$

.

la ricorrenza viene fuori dal seguente ragionamento:

- esiste un'unica sequenza lunga 1 che termina con la cifra j , la cifra j ed è ovviamente non decrescente.
- la sequenza lunga i non decrescente che termina con j è composta da una sequenza non decrescente lunga $i-1$ cui si accoda la cifra j . Perché la sequenza risultante sia non decrescente la sequenza nondecrescente lunga $i-1$ deve terminare con una qualunque cifra k non superiore a j (vale a dire: deve essere una della sequenza contata con $T[i-1, k]$ e $k \leq j$)

$$T[i][j] = \begin{cases} 1 & \text{se } i = 1 \\ \sum_{k=0}^j T[i-1][k] & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es(n):
    T = [ [0]*10 for _ in range(n+1)]
    for j in range(10):
        T[1][j] = 1
    for i in range(2, n+1):
        for j in range(10):
            for k in range(j+1):
                T[i][j] += T[i-1][k]
    return sum(T[n])
```

```
>>> es(1)
10
>>> es(2)
55
>>> es(3)
220
```

Complessità $\Theta(n)$

- **inizializzare la tabella** costa $\Theta(n)$ (nota che la tabella ha $n + 1$ righe ma un numero costante di colonne: 10)
- dei 3 *for* annidati il primo viene iterato **esattamente n** volte, il secondo viene ogni volta iterato **esattamente 10 volte** ed il terzo viene iterato ogni volta **al più 10 volte**. Questo significa che il **costo totale dei 3 *for*** è $\Theta(n)$.

Esercizio: data una matrice M binaria $n \times n$ vogliamo verificare se nella matrice è possibile raggiungere la cella in basso a destra partendo da quella in alto a sinistra senza mai toccare celle che contengono il numero 1. Si può assumere che $M[0][0] = 0$.

Si tenga conto che ad ogni passo ci si può spostare solo di un passo verso destra o un passo verso il basso.

Ad esempio per la matrice A la risposta è SI mentre per la matrice B la risposta è NO

0	0	0	0	0	1
0	1	0	1	1	1
0	0	0	1	0	1
0	1	0	0	0	0
0	0	0	0	1	0
1	1	0	1	0	0

0	0	0	0	0	0
0	1	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	0	0	0	0
0	0	0	0	1	0

Progettare un algoritmo che in tempo $\Theta(n^2)$ risolve il problema rispondendo *True* o *False*.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Utilizziamo una tabella T bidimensionale $n \times n$ dove:

$T[i][j] = \text{True}$ se esiste un cammino che in M va dalla cella $M[0][0]$ in alto a sinistra e arriva alla cella $M[i][j]$ senza toccare celle con valore 1 (*False* altrimenti).

La soluzione al problema sarà nella cella $T[n-1][n-1]$

Ecco di seguito la regola ricorsiva che permette di calcolare i valori della tabella:

$$T[i][j] = \begin{cases} False & \text{se } M[i][j] = 1 \\ True & \text{se } i = j = 0 \\ T[i][j - 1] & \text{se } i = 0 \\ T[i - 1][j] & \text{se } j = 0 \\ T[i][j - 1] \text{ or } T[i - 1][j] & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- la cella $M[0][0]$ è ovviamente raggiungibile.
 - se la cella $M[i][j]$ contiene 1 allora non è possibile raggiungerla.
- Assumiamo quindi nel seguito che $M[i][j]=0$:
- una cella della prima riga può essere raggiunta solo dalla cella che la precede sulla riga.
 - Una cella della prima colonna può essere raggiunta solo dalla cella che la precede sulla colonna.
 - una cella "interna" è raggiungibile se posso raggiungerla dalla cella che la precede sulla riga e/o dalla cella che la precede sulla colonna.

Implementazione:

```
def es(M):
    n=len(M)
    T=[[True]*n for _ in range(n)]
    for j in range(1,n):
        T[0][j]=T[0][j-1]
        if M[0][j]==1:
            T[0][j]= False
    for i in range(1,n):
        T[i][0]=T[i-1][0]
        if M[i][0]==1:
            T[i][0]= False
    for i in range(1,n):
        for j in range(1,n):
            T[i][j]=T[i-1][j] or T[i][j-1]
            if M[i][j]== 1:
                T[i][j]=False
    return T[n-1][n-1]
```

Complessità $\Theta(n^2)$

```
>>> M = [[0, 1, 1], [0, 0, 0], [0, 1, 0]]
>>> es(M)
>>> True # inoltre vale T = [[True, True, True], [True, True, True], [True, False, True]]

>>> M1 = [[0, 0, 0], [0, 1, 1], [1, 0, 0]]
>>> es(M)
>>> False # inoltre vale T = [[True, True, True], [True, False, False], [False, False, False]]
```

ESERCIZIO.

Abbiamo una matrice quadrata binaria M di dimensione $n \times n$ e vogliamo sapere qual è la dimensione massima per le sottomatrici quadrate di soli uni contenute in M .

Ad esempio, per la matrice M in figura la risposta è 3 (in blu è evidenziata la sottomatrice di dimensione 3×3)

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & 0 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

Progettare un algoritmo che data la matrice M restituisce il massimo intero l per cui la matrice $l \times l$ è una sottomatrice di soli uni di M .

L'algoritmo deve avere complessità $O(n^2)$.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Utilizziamo una tabella T bidimensionale $n \times n$ dove:

$T[i][j]$ = il lato della matrice quadrata più grande contenente tutti uni e con cella in basso a destra $M[i][j]$ (se $M[i][j]=0$ allora $T[i][j]=0$).

La soluzione al nostro problema sarà il massimo tra i valori della tabella.
Ecco di seguito la regola ricorsiva che permette di calcolare i vari valori della tabella:

$$T[i][j] = \begin{cases} 0 & \text{se } M[i][j] = 0 \\ \min \{ T[i][j-1], T[i-1][j-1], T[i-1][j] \} + 1 & \text{altrimenti} \end{cases}$$

La ricorrenza viene fuori dal seguente ragionamento:

- Se $M[i][j] = 0$ allora il rettangolo di tutti uni che ha come cella in basso a destra quella cella ha lato zero.
- Se $M[i][j] = 1$ allora il lato del rettangolo di soli uni che ha come cella in basso a destra quella cella non può superare $T[i-1][j] + 1$ né $T[i][j-1] + 1$ né $T[i-1][j-1] + 1$ ne segue che il lato di quel rettangolo è al più $\min \{ T[i][j-1], T[i-1][j-1], T[i-1][j] \} + 1$ d'altra parte non è difficile vedere che preso $l = \min \{ T[i][j-1], T[i-1][j-1], T[i-1][j] \}$ allora il rettangolo di lato $l+1$ con cella in basso a destra $M[i][j]$ è un rettangolo contenente solo uni.

Implementazione:

Complessità $\Theta(n^2)$

```
def es(M):
    n = len(M)
    T = [ [0]*n for _ in range(n) ]
    for i in range(n):
        for j in range(n):
            if M[i][j] == 0:
                T[i][j] = 0
            else:
                T[i][j] = min(T[i][j-1], T[i-1][j-1], T[i-1][j] ) + 1
    m = 0
    for i in range(n):
        m = max(m, max(T[i]))
    return m
```

```
M=[
    [1,0,1,0],
    [0,1,1,1],
    [1,1,1,0],
    [0,0,0,1]
```

```
]
>>> es(M)
2
```

ESERCIZIO.

Il problema dello zaino: Abbiamo uno zaino di capacità c ed n oggetti, ognuno con un peso p_i e un valore v_i .
Vogliamo sapere il valore massimo che si può inserire nello zaino.

Ad Esempio, si consideri l'istanza con $c=11$ e $n=5$ oggetti con peso e valore riportati nella seguente tabella

oggetto	valore	peso
1	1	1
2	6	4
3	18	5
4	22	5
5	28	7

- il sottoinsieme di oggetti $\{3, 5\}$ non è una soluzione ammissibile (perché di peso $12 > C$)
- il sottoinsieme di oggetti $\{1, 2, 4\}$ è una soluzione ammissibile (di peso 10 e valore 29) ma non ottima (perché ne esistono di valore maggiore).
- il sottoinsieme di oggetti $\{1, 3, 4\}$ è una soluzione ammissibile (di peso 11 e valore 41) si può dimostrare che è anche una soluzione ottima.

Progettare un algoritmo che, data la capacità c dello zaino e i vettori P (dei pesi) e V (dei valori) degli oggetti, risolve il problema in tempo $\Theta(nc)$.

Utilizziamo una tabella bidimensionale T di dimensione $(n+1) \times (C+1)$ dove:

$T[i][j]$ = massimo valore ottenibile dai primi i oggetti per uno zaino di capacità j .

La soluzione al nostro problema sarà $T[n][c]$.

Ecco di seguito la regola ricorsiva che permette di calcolare i vari valori della tabella:

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j] & \text{se } p_i > j \\ \max\{ T[i-1][j], v_i + T[i-1][j-p_i] \} & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se non si hanno oggetti (vale a dire $i = 0$) o la capacità dello zaino è nulla allora il valore della soluzione sarà 0.
- se l' i -esimo oggetto ha un peso superiore alla capacità j dello zaino (vale a dire $j < P[i-1]$, allora non può esservi inserito e quindi il valore della soluzione dipenderà da quello che si può ottenere dagli altri $i-1$ oggetti (vale a dire $T[i-1, c]$)
- in generale, se nello zaino c'è possibilità di inserire l'oggetto i (vale a dire $P[i-1] \leq j$) allora ci sono due possibilità:
 - *l'oggetto i non viene inserito nello zaino.* In questo caso il massimo valore della soluzione sarà $T[i-1, c]$.
 - *l'oggetto i viene inserito nello zaino.* In questo caso c'è un guadagno $V[i]$ e per i rimanenti $i-1$ oggetti resta disponibile una capacità residua di $j - P[i]$. Di conseguenza il valore massimo della soluzione sarà $V[i-1] + T[i-1, j - P[i-1]]$.

Pertanto il valore della soluzione sarà $\max(T[i-1, j], V[i-1] + T[i-1, c - P[i-1]])$

Implementazione:

Complessità $\Theta(nc)$

```
def es(P, V, c):
    n=len(P)
    T=[[0]*(c + 1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, c+1):
            if j < P[i-1]:
                T[i][j]=T[i-1][j]
            else:
                T[i][j]= max( T[i-1][j], V[i-1] + T[i-1][j - P[i-1]] )
    return T, T[n][c]
```

```
>>> V = [1, 6, 18, 22, 28]
```

```
>>> P = [1, 4, 5, 5, 7]
```

```
>>> es(P, V, 10)
```

```
>>> 40
```

e per la tabella **T** di dimensioni **6 x 11** | si avrà:

```
T= [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 6, 7, 7, 7, 7, 7, 7],
    [0, 1, 1, 1, 6, 18, 19, 19, 19, 24, 25],
    [0, 1, 1, 1, 6, 22, 23, 23, 23, 28, 40],
    [0, 1, 1, 1, 6, 22, 23, 28, 29, 29, 40]
]
```

ESERCIZIO:

Il problema del resto: Ho n diversi tagli di banconote ed un resto c da dare. Voglio sapere quanti modi ci sono di produrre il resto c se ho quantità illimitata di banconote dei vari tagli.

Ad esempio, con i tre tagli 1,2 e 3 e $c = 5$ la risposta è 5 (infatti si hanno i seguenti possibili resti: 11111 1112 113 23 122.

Dato l'intero c ed il vettore M con gli n tagli delle banconote ($M[i]$ è l'iesimo taglio) progettare un algoritmo che risolve il problema.

L'algoritmo proposto deve avere complessità $\Theta(n \cdot c)$.

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

● Utilizzeremo una tabella bidimensionale di dimensioni $(n + 1) * (c + 1)$ e definiamo il contenuto delle celle come segue:

$T[i][j] =$ differenti modi di dare il resto j quando si dispone dei soli primi i tagli $0 \leq i \leq n, 0 \leq j \leq c$

● una volta riempita la tabella la soluzione al problema sarà data da $T[n][c]$

Ecco di seguito la ricorrenza che permette di riempire la tabella

$$T[i][j] = \begin{cases} 1 & \text{se } j = 0 \\ 0 & \text{se } i = 0 \\ T[i-1][j] & \text{se } j < A[i-1] \\ T[i-1][j] + T[i][j - A[i-1]] & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- C'è un solo modo di dare resto zero. $T[i][0] = 1$
- Con zero tagli non c'è modo di dare un resto $j > 0$. $T[0][j] = 0$ (se $j > 0$)
- se l' i -esimo taglio è maggiore del resto da dare allora banconote di quel taglio non possono contribuire alla produzione del resto. $T[i][j] = T[i-1][j]$
- in generale, se la banconota può contribuire al resto posso avere un resto in cui la uso ed uno in cui non la uso.

- *La banconota i non viene usata per produrre il resto j .* In questo caso il resto posso ottenerlo solo con le altre banconote $T[i][j] = T[i-1][j]$.
- *La banconota i viene usata per produrre il resto i .* In questo caso dovrò produrre ancora il resto $j - M[i]$ e quindi $T[i][j] = T[i][j - M[i]]$

Pertanto i modi di dare il resto sono $T[i][j] = T[i-1][j] + T[i][j - M[i]]$

```
def es(A, c):
    n = len(A)
    T = [[0]*(c+1) for _ in range(n+1)]
    for i in range(n+1):
        T[i][0] = 1
    for j in range(1, c+1):
        T[0][j]=0
    for i in range(1, n+1):
        for j in range(1, c+1):
            if j < A[i-1]:
                T[i][j] = T[i-1][j]
            else:
                T[i][j] = T[i-1][j] + T[i][j-A[i-1]]
    return T, T[n][c]
```

```
>>> A = [1, 2, 3]
```

```
>>> es(A, 5)
```

```
5
```

```
#si ha infatti
```

```
T=[
    [1, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1],
    [1, 1, 2, 2, 3, 3],
    [1, 1, 2, 3, 4, 5]
]
```

ESERCIZIO

Una **transazione** è l'acquisto di un oggetto seguito dalla sua vendita (che non può ovviamente avvenire prima del giorno dell'acquisto).

Disponiamo di un vettore A di interi dove $A[i]$ è la quotazione dell'oggetto nel giorno i .

Dato il vettore A con le quotazioni dei prossimi n giorni e dovendo eseguire una singola transazione vogliamo sapere qual è il guadagno massimo cui possiamo aspirare.

nota che se il vettore è decrescente il guadagno massimo è 0.

Ad esempio:

- per $A = [7, 9, 5, 6, 3, 2]$ il guadagno massimo è 2 (basta acquistare a 7 e rivendere a 9).
- Per $B = [3, 2, 6, 10, 4, 8, 1]$ il guadagno massimo è 8 (basta acquistare a 2 e rivendere a 10).

Progettare un algoritmo che risolve il problema in tempo $\Theta(n)$.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Utilizziamo una tabella T monodimensionale di lunghezza n dove:

$T[i]$ = il guadagno massimo che ottengo se vendo il giorno i .

La soluzione al problema sarà $\max(T)$.

La ricorrenza che permette di riempire la tabella è la seguente:

$$T[i] = \begin{cases} 0 & \text{se } i = 0 \\ A[i] - \min(A[:i+1]) & \text{altrimenti} \end{cases}$$

La ricorrenza viene fuori dal seguente ragionamento:

- Se vendo il giorno iniziale non posso che comprare il giorno stesso e quindi il guadagno è 0.
- Se vendo il giorno i ottengo $A[i]$ per la vendita e per avere il guadagno massimo devo aver acquistato nel giorno tra 0 ed i in cui l'oggetto costava meno.

Per ottenere tempo $\Theta(n)$ devo essere in grado di calcolare ciascuna delle n celle in tempo $O(1)$ e per far ciò basta precalcolare per ciascun giorno il costo minimo fino a quel giorno.

```
def es(A):
    #in B[i] precalcolo il valore min(A[:i+1])
    B = [0]*n
    B[0] = A[0]
    for i in range(1, n):
        B[i] = min(A[i], B[i-1])
    #
    T = [0]*n
    for i in range(1, n):
        T[i] = A[i] - B[i]
    return max(T)
```

Complessità $\Theta(n)$

ESERCIZIO

Una **transazione** è l'acquisto di un oggetto seguito dalla sua vendita (che non può ovviamente avvenire prima del giorno dell'acquisto).

Disponiamo di un vettore A di interi dove $A[i]$ è la quotazione dell'oggetto nel giorno i .

Dato il vettore A con le quotazioni dei prossimi n giorni ed un intero k e dovendo eseguire al più k transazioni vogliamo sapere qual'è il guadagno massimo cui possiamo aspirare. ATTENZIONE: non posso cominciare una nuova transazione se non dopo aver completato la transazione precedente.

nota che se il vettore è decrescente il guadagno massimo è 0.

Ad esempio:

- per $A = [10, 22, 5, 75, 65, 80]$ il guadagno massimo è 87 (basta acquistare a 10 e rivendere a 22 e poi a 5 per rivendere a 8).
- Per $B = [100, 30, 15, 10, 8, 25, 80]$ il guadagno massimo è 72 (basta acquistare a 8 rivendere a 8).

Progettare un algoritmo che risolve il problema in tempo $\Theta(nk)$.

Utilizziamo una tabella T bidimensionale di lunghezza $n \times (k + 1)$ dove:

$T[i][j]$ = il guadagno massimo che ottengo se faccio j transazioni entro il giorno i .

La soluzione al problema sarà $T[n - 1][k]$.

La ricorrenza che permette di riempire la tabella è la seguente:

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ or } j = 0 \\ \max \left(T[i-1][j], \max_{0 \leq s \leq i} \left((A[i] - A[s]) + T[s][j-1] \right) \right) & \text{altrimenti} \end{cases}$$

La ricorrenza viene fuori dal seguente ragionamento:

- Un numero arbitrario di transazioni eseguite nel giorno 0 fruttano 0.
- Conterò transazioni eseguite, indipendentemente dal giorno il guadagno è 0.
- Se devo calcolare $T[i][j]$ con j e i maggiori di zero posso effettuare o meno una delle j transazioni nel giorno i .
 - Se non effettuo transazioni nel giorno i allora il mio guadagno sarà $T[i-1][j]$
 - Se effettuo almeno una transazione nel giorno i allora devo aver acquistato in un giorno $s \leq i$ e quindi il mio guadagno sarà $(A[i] - A[s]) + T[s][j-1]$ e per il guadagno massimo devo prendere $\max_{0 \leq s \leq i} \left((A[i] - A[s]) + T[s][j-1] \right)$

Per ottenere il meglio delle due situazioni devo quindi prendere:

$$\max \left(T[i-1][j], \max_{0 \leq s \leq i} \left((A[i] - A[s]) + T[s][j-1] \right) \right)$$

Implementazione:

Complessità $\Theta(nk)$

```
def es(A, k):
    n = len(A)
    T=[[0]*(k+1) for _ in range(n)]
    for i in range(1, n):
        for j in range(1, k+1):
            T[i][j] = T[i-1][j]
            for s in range(i+1):
                T[i][j]= max( T[i][j], (A[i] - A[s]) + T[s][j-1])
    return T[n-1][k]
```

```
>>> A=[100, 30, 15, 10, 8, 25, 80]
```

```
>>> es3(A, 3)
```

```
>>> 72
```

```
# la sola transazione che frutta è 8-80 e i valori della tabella sono:
```

```
T= [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0],
     [0, 0, 0, 0], [0, 17, 17, 17], [0, 72, 72, 72]
    ]
```

```
>>> B = [12, 14, 17, 10, 14, 13, 12, 15]
```

```
>>> es3(B, 3)
```

```
>>> 12 #le 3 transazioni sono 12-17, 10-14, 12-15 e i valori della tabella sono:
```

```
T=[ [0, 0, 0, 0], [0, 2, 2, 2], [0, 5, 5, 5], [0, 5, 5, 5],
     [0, 5, 9, 9], [0, 5, 9, 9], [0, 5, 9, 9], [0, 5, 10, 12]], 12)
```

Corso di laurea in Informatica

Introduzione agli Algoritmi

Esercizi per casa



SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO.

Progettare un algoritmo che, dato l'intero positivo n , in tempo $O(n)$ conta il numero di stringhe ternarie lunghe n che non contengono né la sottostringa 02 né la sottostringa 20 (vale a dire: le cifre adiacenti presenti nella stringa differiscono al più di 1).

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 3
- per $n = 3$ la risposta deve essere 17 perché delle $3^3 = 27$ stringhe ternarie lunghe 3 le sole stringhe 'vietate' sono le seguenti 10:

020, 021, 022, 002, 102, 202, 200, 201, 120, 220.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

per $n = 1$	la risposta è	3
per $n = 2$	la risposta è	7
per $n = 3$	la risposta è	17
per $n = 4$	la risposta è	41
per $n = 5$	la risposta è	99
per $n = 6$	la risposta è	239
per $n = 7$	la risposta è	577
per $n = 8$	la risposta è	1393
per $n = 9$	la risposta è	3363

ESERCIZIO.

Progettare un algoritmo che, dato un intero n , restituisce il numero di stringhe lunghe n che è possibile ottenere con i 4 simboli 0, 1, 2 e 3 facendo in modo che nelle stringhe non compaiano mai due cifre dispari adiacenti.

Ad esempio

- per $n = 2$ la risposta è 12, infatti le stringhe lecite sono:
00, 01, 02, 03, 10, 12, 20, 21, 22, 23, 30, 32.

L'algoritmo proposto deve avere complessità $O(n)$.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

per $n = 1$ la soluzione è 4
per $n = 2$ la soluzione è 12
per $n = 3$ la soluzione è 40
per $n = 4$ la soluzione è 128
per $n = 5$ la soluzione è 416
per $n = 6$ la soluzione è 1344
per $n = 7$ la soluzione è 4352
per $n = 8$ la soluzione è 14080
per $n = 9$ la soluzione è 45568

Esercizio: dato un intero positivo k ed una matrice M con interi positivi di dimensione $n \times n$ contare i cammini di costo k che in M partono dalle cella in alto a sinistra e raggiungono la cella in basso a destra.

Si tenga conto che ad ogni passo ci si può spostare solo di un passo verso destra o un passo verso il basso e che il costo di un cammino è dato dalla somma dei valori delle celle toccate.

Ad esempio per la matrice A di seguito a sinistra con $k = 12$ la risposta è 2

$A =$

1	2	3
4	6	5
3	2	1

I due cammini sono:

$1 \rightarrow 2 \rightarrow 6 \rightarrow 2 \rightarrow 1$

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 1$

Progettare un algoritmo che in tempo $\Theta(n^2k)$ risolve il problema.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.