Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Programmazione Dinamica 2b

Angelo Monti

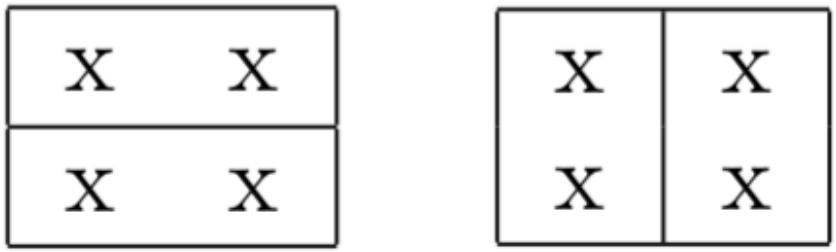
SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO 11

Dato l'intero n vogliamo contare il numero di differenti tassellamenti di una superficie di dimensione $n \times 2$ tramite tessere di domino di dimensione 1×2.

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 1
- per $n = 2$ la risposta deve essere 2 perché sono possibili i soli due seguenti tassellamenti:



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

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

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
  $T[i]$ = numero di tassellamenti possibili per la superficie di dimensione $i \times 2$.
- Una volta riempita la tabella la soluzione al nostro problema la troveremo nella locazione $T[n]$.

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

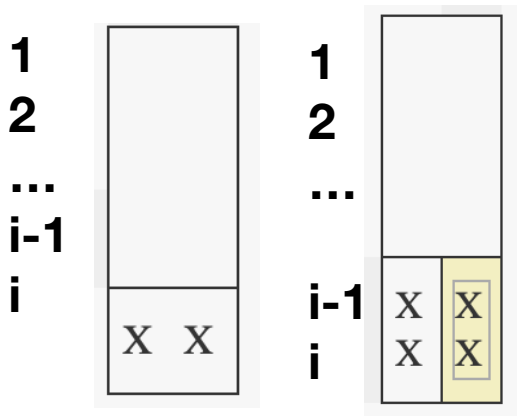
$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ T[i-1] \; + \; \cdot T[i-2] & \text{altrimenti} \end{cases}$$

.

la ricorrenza viene fuori dal seguente ragionamento:

- possiamo vedere i tassellamenti della superficie di dimensione $i \times 2$ come la somma di due diverse tipologie: i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale e i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in verticale:

  1. i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale sono $T[i-1]$, infatti il tassello occupa l'i-esima riga della superficie e resta da tassellare una superficie di dimensione $(i-1) \times 2$ (le prime $i-1$ righe della superficie) in tutti i modi possibili. Questi modi sono appunto $T[i-1]$.
  2. i tassellamenti in cui il tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale sono $T[i-2]$ infatti in questo caso il vertice in basso a destra deve essere anch'esso coperto da un tassello in verticale, questi due tasselli lasciano da coprire una superficie di dimensione $(i-2) \times 2$ (la prime $i-2$ righe della superficie iniziale) in tutti i modi possibili. Questi modi sono appunto $T[i-1]$.



$$T[i] = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ T[i-1] + T[i-2] & \text{altrimenti} \end{cases}$$

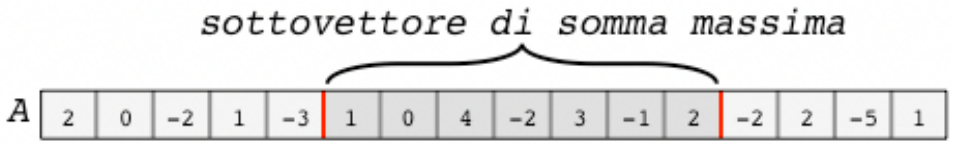
Implementazione:

```
def es(n):
    m= max(3,n)
    T = [0]*(n+1)
    T[1], T[2] = 1, 2
    for i in range(3, n+1):
        T[i] = T[i-1] + T[i-2]
    return T[n]

>>> es(3)
3

>>> es(7)
21
```

Complessità $\Theta(n)$

ESERCIZIO

Il problema del massimo sottovettore: Data una lista A di n interi, vogliamo trovare una sottolista (una sequenza di elementi consecutivi della lista) la somma dei cui elementi è massima.



Abbiamo già parlato di questo problema come esempio di applicazione della tecnica del *divide et impera*, lo riprendiamo ora per tentare un approccio basato sulla programmazione dinamica. Come è tipico di questa tecnica ci concentriamo prima sul calcolare il valore della soluzione:

Cominciamo con l'individuare i sottoproblemi dalla composizione delle cui soluzioni sarà poi possibile risolvere il problema originario.

- una scelta che ci porta a definire n sottoproblemi è la seguente:

  $T[i]$ = massima somma possibile per le sottoliste di A che terminano nella posizione i

- Poiché la sottolista a valore massimo deve terminare in una qualche posizione il valore della soluzione che cerchiamo sarà poi dato da

$$\max_{0 \le i < n} T[i]$$

- $T[i]$ = massima somma possibile per le sottoliste di A che terminano nella posizione i

  vediamo ora come risolvere in modo efficiente gli n sottoproblemi:

  - Il caso base è facile: $T[0] = A[0]$.
  - dobbiamo ora poter calcolare $T[i]$, con $i > 0$, sfruttando la conoscenza di $T[i-1]$:

    $$T[i] = \max(A[i], A[i] + T[i-1])$$

    Infatti la sottolista di valore massimo che termina con $A[i]$ può essere di due soli tipi:

    - consiste del solo elemento A[i]
    - ha lunghezza superiore ad 1. In questo caso vale $A[i] + S$ dove S è la massima somma di un sottovettore che termina in $i-1$ (il che significa che $S = T[i-1]$)

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

>>> A = [2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, 2, -5, 1]
>>> es(A)
7
```

Complessità $O(n)$

Ottimizzazioni:

- possiamo evitare di dover calcolare al termine il massimo in T, basta una variabile m che tiene traccia del massimo per il valore delle celle di T che via via calcoliamo.
- possiamo evitare di tenere in memoria l'intera tabella T visto che per calcolare $T[i]$ basta disporre della sola cella $T[i-1]$ (e del valore $A[i]$) (in questo modo la complessità di spazio passa da $O(n)$ ad $O(1)$).

Implementazione 2:

```
def es(A):
    n = len(A)
    m = t = 0
    for i in range(1,n):
        t = max(A[i], A[i] + t)
        m= max(m, t)
    return m
```

Complessità $O(n)$

ESERCIZIO

Data una sequenza S di elementi una *sottosequenza* di S si ottiene eliminando zero o più elementi da S (Nota: non necessariamente le eliminazioni devono avvenire in testa o in coda alla sequenza)

Ad esempio: data la sequenza $9, 3, 2, 4, 1, 5, 8, 6, 7, 2$ una sua sottosequenza è $3, 1, 5, 6$ (si ottiene eliminando dalla sequenza gli elementi di seguito indicati in rosso **9**,3,**2**,**4**,1,5,**8**,6,**7**,**2.**

NOTA: le sottosequenze possibili per una sequenza di n elementi sono $\Theta(2^n)$.
Una sottosequenza è detta crescente se i suoi elementi risultano ordinati in modo crescente.

Il problema della sottosequenza crescente più lunga: Data una sequenza S di n interi, vogliamo trovare la lunghezza massima per le sottosequenze crescenti presenti in S

Ad esempio: per la sequenza $S = 9, 3, 2, 4, 1, 5, 8, 6, 7, 2$ la risposta è 5 infatti la sottosequenza più lunga in S è $3, 4, 5, 6, 7$ (non è l'unica, un'altra possibile soluzione è $2, 4, 5, 6, 7$).

Progettare un algoritmo che data una sequenza S di n elementi, in tempo $O(n^2)$ risolve il problema.
Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

- Utilizzeremo una tabella monodimensionale di dimensioni n e definiamo il contenuto delle celle come segue:
  $T[i]$ = la lunghezza massima per una sottosequenza crescente che termina con l'elemento di S in posizione i.
  - Poiché la sottosequenza di lunghezza massima da qualche parte dovrà pur terminare la soluzione al nostro problema sarà data da:

$$\max_{0 \leq i < n} (T[i])$$

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

$$T[i] = \begin{cases} 1 & \text{se } S_j > S_i \text{ per } 0 \leq j < i \\ \max_{0 \leq j < i \text{ con } S_j < S_i}(T[j]) + 1 & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se tutti gli elementi che precedono S_i sono maggiori di S_i allora l'unica sottosequenza crescente che termina in posizione i è data dal solo elemento S_i.

- in caso contrario la sottosequenza che termina con S_i avrà come prefisso una sottosequenza crescente che termina in una posizione j con $j < i$ e $S_j \leq S_i$ e tra tutti i possibili "agganci" prendo quello che produce la sottosequenza più lunga.

- $T[i] = \begin{cases} 1 & \text{se } S_j > S_i \text{ per } 0 \leq j < i \\ \max_{0 \leq j < i \text{ con } S_j < S_i}(T[j]) + 1 & \text{altrimenti} \end{cases}$.

Implementazione:

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


>>> A = [10, 9, 2, 5, 3, 7, 101, 18]
>>> es(A)
4       #infatti esiste la sottosequenza crescente 2,5,7,18
>>> B=[5,4,3,2]
>>> es(B)
1.  # infatti la stringa è decrescente
```

La complessità è $O(n^2)$

ESERCIZIO

Un numero intero può sempre essere rappresentato come somma di quadrati di altri numeri interi. Infatti, usando il quadrato 1^2, il generico numero x possiamo sempre scomporlo come somma di n addendi tutti uguali a 1^2 (vale a dire $x = 1^2 + \ldots + 1^2$).
Dato un intero n vogliamo sapere qual'è il numero minimo di quadrati necessari rappresentare n.

Ad esempio:

- per $n = 0$ la risposta è 0.
- per $n = 41$ la risposta è 2 infatti $41 = 5^2 + 4^2$. Nota che vale anche $41 = 6^2 + 2^2 + 1^2$ ma questa scomposizione usa 3 quadrati e non è minimale.
- per $n = 6$ la risposta è 3 infatti $6 = 2^2 + 1^2 + 1^2$ e non è difficile vedere che 6 non può esprimersi come somma di due soli quadrati.

Progettare un algoritmo che dato n in $\Theta\left(n^{\frac{3}{2}}\right)$ calcola il numero minimo di quadrati che servono per rappresentarlo.

- Utilizzeremo una tabella monodimensionale di dimensioni $n+1$ e definiamo il contenuto delle celle come segue:
  $T[i]$ = numero minimo di quadrati per rappresentare l'intero i.

- Una volta riempita la tabella troveremo la soluzione in $T[n]$.

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

$$T[i] = \begin{cases} 0 & \text{se } i = 0 \\ \min_{0 \leq j \leq \lfloor \sqrt{i} \rfloor} (T[i - j^2]) + 1 & \text{altrimenti} \end{cases}$$

.

la ricorrenza viene fuori dal seguente ragionamento:

- per $i = 0$ ovviamente vale $T[0] = 0$

- Per il caso generale dove $i > 0$ servirà utilizzare almeno un quadrato. Tra tutti i quadrati potenzialmente utili ad ottenere i servirà dunque trovare quello che poi permette di ottenere i con il minor numero di quadrati. I quadrati potenzialmente utili sono del tipo j con $1 \leq j \leq \lfloor \sqrt{i} \rfloor$ e, notando che se si usa il quadrato j resterà poi ancora da rappresentare il numero $i - j^2$ si ottiene la seguente formula: $T[i] = \min_{1 \leq j \leq \lfloor \sqrt{i} \rfloor} T[i - j^2] + 1$.

$$T[i] = \begin{cases} 0 & \text{se } i = 0 \\ \min_{0 \leq j \leq \lfloor \sqrt{i} \rfloor} (T[i - j^2]) + 1 & \text{altrimenti} \end{cases}$$

Implementazione:

```
def es(n):
    T=[0]*(n+1)
    for i in range(1, n+1):
        T[i] = n
        j = 1
        while j**2 <= i:
            if T[ i - j**2] + 1 < T[i]:
                T[i] = T[i-j**2] + 1
            j += 1
    return T[n]

>>> es(100)
1
>>> es(12)
3
>>> es(41)
2
```

Complessità $\Theta\left(n^{\frac{3}{2}}\right)$

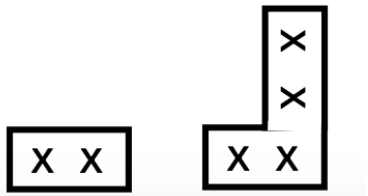
Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO

Dato l'intero n vogliamo contare il numero di differenti tassellamenti di una superficie di dimensione $n \times 2$ tramite tessere di domino dei due formati indicati di seguito:.



Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve ovviamente essere 1 in quanto il solo tassellamento possibile è quello che usa il tassello rettangolare.
- per $n = 3$ la risposta deve essere 7 perché sono possibili 3 differenti tassellamenti con il solo tassello rettangolare e 4 differenti tassellamenti che utilizzano due tasselli di tipo differente.

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

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

```
per n =  1 la risposta è  1
per n =  2 la risposta è  2
per n =  3 la risposta è  7
per n =  4 la risposta è  15
per n =  5 la risposta è  32
per n =  6 la risposta è  79
per n =  7 la risposta è  185
per n =  8 la risposta è  422
per n =  9 la risposta è  987
```

ESERCIZIO.

Data una lista A con una serie di n tagli di monete ed un importo r da raggiungere trovare il numero minimo di monete necessarie per ottenere tale importo. Se non è possibile ottenere quell'importo restituire -1.

Ad esempio
per $A = [5, 7, 10, 25]$ e c=8 la risposta deve essere -1
Per $A = [5, 7, 10, 25]$ e c=62 la risposta è 4 (utilizzando i 4 tagli $7,\ 5,\ 25,\ 25$)

L'algoritmo proposto deve avere complessità $O(r \cdot n)$

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

ESERCIZIO

Una **progressione aritmetica** è una sequenza di numeri in cui la differenza tra ogni coppia successiva di numeri è costante. Questa differenza costante è chiamata "ragione".
Ad esempio, la sequenza 2, 5, 8, 11, 14, è una progressione aritmetica di ragione 3. Una sequenza di un solo elemento è una progressione aritmetica di qualunque ragione.

Dato un array di n interi positivi ed un intero c vogliamo contare le progressioni aritmetiche di ragione c che compaiono come sottosequenze dell'array.
Ad esempio per la sequenza 1,3,4,6 e c=2 la risposta è 6 infatti sono presenti le seguenti 6 sequenze di ragione 2: 1, 3, 4, 6, 1,3 e 4,6

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

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

ESERCIZIO.

Progettare un algoritmo che, dato un intero n, calcola il numero di stringhe binarie lunghe $2n$ per le quali la somma dei primi n bit è uguale alla somma degli ultimi n bit.

Ad esempio:

- per $n = 2$ per la risposta dell'algoritmo deve essere 6 infatti le stringhe di lunghezza 2*2=4 che soddisfano il vincolo sono : $0101,\ 0110,\ 1010,\ 1001,\ 0000\ \ 1111$

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

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

```
per n =  1 la soluzione è  2
per n =  2 la soluzione è  6
per n =  3 la soluzione è  20
per n =  4 la soluzione è  70
per n =  5 la soluzione è  252
per n =  6 la soluzione è  924
per n =  7 la soluzione è  3432
per n =  8 la soluzione è  12870
per n =  9 la soluzione è  48620
```

ESERCIZIO

Data una sequenza S di interi positivi una *sottosequenza* di S si ottiene eliminando zero o più elementi da S (Nota: non necessariamente le eliminazioni devono avvenire in testa o in coda alla sequenza)

Ad esempio: data la sequenza $9, 3, 2, 4, 1, 5, 8, 6, 7, 2$ una sua sottosequenza è $3, 1, 5, 6$ (si ottiene eliminando dalla sequenza gli elementi di seguito indicati in rosso **9**,3,**2**,**4**,1,5,**8**,6,**7**,**2.**

NOTA: le sottosequenze possibili per una sequenza di n elementi sono $\Theta(2^n)$.

Il problema della sottosequenza crescente di valore massimo: Data una sequenza S di n interi, vogliamo trovare il valore massimo per sottosequenze crescenti (vale a dire i cui elementi risultano ordinati in modo crescente). (Il valore di una sottosequenza è dato dalla somma dei suoi elementi massima.

Ad esempio: data la sequenza $S = 22, 3, 9, 4, 1, 5, 20, 6, 7, 2$ una sottosequenza crescente di valore 25 per S è $3, 4, 5, 6, 7$.
Per S la soluzione al problema è 29 grazie alla sottosequenza 9, 20.

Progettare un algoritmo che data una sequenza S di n elementi, in tempo $O(n^2)$ risolve il problema .
Motivare BENE la correttezza e la complessità dell'algoritmo proposto.