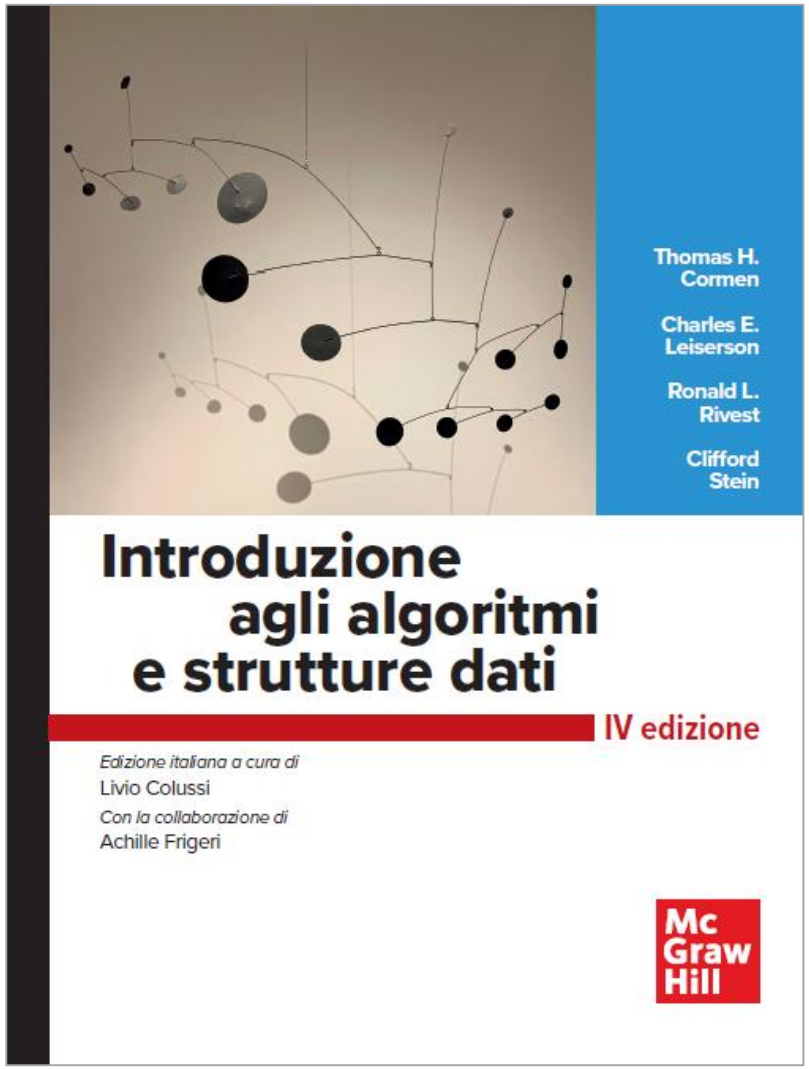


Capitolo 14

Programmazione dinamica

La tecnica della *programmazione dinamica* ci permette di risolvere efficientemente un problema algoritmico quando i sottoproblemi in cui esso si può scomporre non sono indipendenti.

Vediamo dapprima un esempio.

Programmazione Dinamica: taglio delle aste

Problema del taglio delle aste

E’ data un'asta metallica di lunghezza n che deve essere tagliata in pezzi di lunghezza intera (con un costo di taglio trascurabile).

Per ogni lunghezza $l = 1,\ldots,n$ è dato il prezzo p_l a cui si possono vendere i pezzi di quella lunghezza.

Si vuole decidere come tagliare l’asta in modo da rendere massimo il ricavo della vendita dei pezzi ottenuti.

Esempio: tabella dei ricavi

ℓ	1	2	3	4	5	6	7	8	9	10
p_ℓ	1	5	8	9	10	17	17	20	24	30

Lunghezza asta n	Ricavo massimo r_n	Suddivisione ottima
1	1	1
2	5	2
3	8	3
4	10	2+2
5	13	2+3
6	17	6
7	18	1+6 o 2+2+3
8	22	2+6
9	25	3+6
10	30	10

Un'asta di lunghezza n può essere tagliata in 2^{n-1} modi distinti in quanto abbiamo una opzione tra tagliare o non tagliare in ogni posizione intera **1,…,*n*-1.**

Ad esempio per $n = 4$ abbiamo i seguenti 8 modi

Suddivisioni possibili:

4	**1+1+2**
1+3	**1+2+1**
2+2	**2+1+1**
3+1	**1+1+1+1**

In generale il ricavo massimo r_n o è il costo p_n dell'asta intera oppure si ottiene effettuando un primo taglio in posizione i e quindi sommando i ricavi massimi del primo e del secondo pezzo, ossia

$$r_n = r_i + r_{n-i}$$

Quindi

$$r_n = \begin{cases} p_1 & \text{se } n = 1 \\ \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) & \text{se } n > 1 \end{cases}$$

Osserviamo che la soluzione ottima del problema di ottiene da soluzioni ottime di sottoproblemi. Diciamo che il problema ha una ***sottostruttura ottima***.

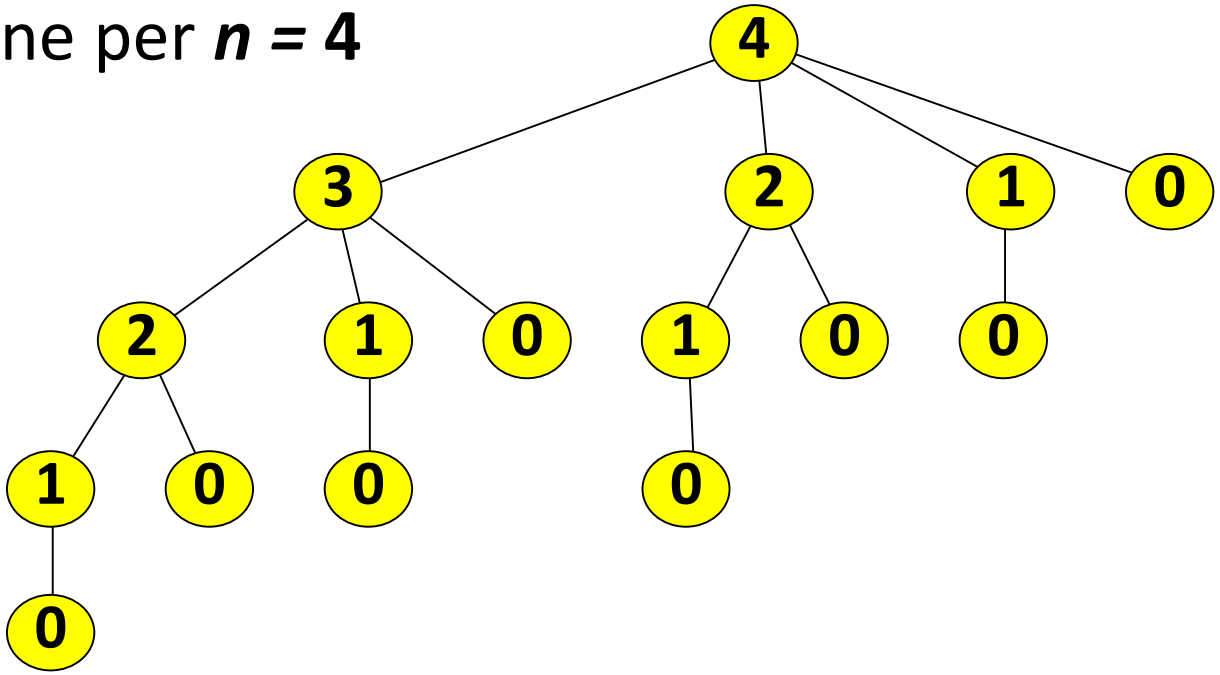
Otteniamo una struttura ricorsiva più semplice se invece di scegliere la posizione *i* di un primo taglio intermedio scegliamo la lunghezza *i* del primo pezzo per cui $r_n = p_i + r_{n-i}$

$$r_n = \begin{cases} 0 & \text{se } n = 0 \\ \max\limits_{1 \le i \le n}(p_i + r_{n-i}) & \text{se } n > 0 \end{cases}$$

```
Cut-Road(p, n)
    if n == 0
        return 0
    q = -1
    for i =1 to n
        q = max(q, p[i]+Cut-Road(p, n-i))
    return q
```

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n$$

Albero di ricorsione per $n = 4$



Lo stesso problema di dimensione **2** viene risolto due volte, quello di dimensione **1** quattro volte e quello di dimensione **0** otto volte.

Questo spiega la complessità $\mathbf{2}^{n}$.

Sottoproblemi ripetuti

Possiamo ridurre la complessità evitando di risolvere più volte gli stessi problemi.

Un primo modo per farlo è dotare l'algoritmo di un blocco note in cui ricordare le soluzioni dei problemi già risolti: metodo top-down con memoizzazione.

Un secondo modo per farlo è calcolare prima i problemi più piccoli memorizzandone le soluzioni e poi usare tali soluzioni per risolvere i problemi più grandi: metodo bottom-up.

Taglio delle aste: versione top-down con memoizzazione

```
Memoized-Cut-Road(p, n)
  for i = 0 to n
    // inizializza il blocco note
    r[i] = -1
  return Cut-Road-Aux(p, n, r)
```

$$T(n) = \Theta(n^2)$$

```
Cut-Road-Aux(p, j, r)
  if r[j] ≥ 0      // il problema è già stato risolto
    return r[j]
  if j == 0
    q = 0
  else q = -1
      for i = 1 to j
        q = max(q, p[i] + Cut-Road-Aux(p, j-i, r))
  r[j] = q
  return q
```

Taglio delle aste: versione bottom-up

```
Bottom-Up-Cut-Road(p, n)
    r[0] = 0       // il problema più semplice
    for j = 1 to n
        q = -1
        for i = 1 to j
            q = max(q, p[i] + r[j-i])
        r[ j] = q
    return r[n]
```

$$T(n) = \Theta(n^2)$$

Taglio delle aste: versione bottom-up estesa per calcolare anche la soluzione ottima

```
Extended-Bottom-Up-Cut-Road( p, n)
  r[0] = 0
  for j = 1 to n
    q = -1
    for i = 1 to j
      if q < p[i]+r[ j - i]
        q = p[i]+r[ j - i]
        s[ j] = i      // memorizzo il taglio ottimo
    r[ j] = q
  return r ed s
```

Taglio delle aste: stampa della soluzione

```
Print-Cut-Road-Solution( p, n)
  (r, s) = Extended-Bottom-Up-Cut-Road( p, n)
  j = n
  while j > 0
    print s[j]
    j = j - s[j]
```

Moltiplicazione di matrici

L'algoritmo per moltiplicare due matrici A e B di dimensioni $p \times q$ e $q \times r$ è:

```
Matrix-Multiply(A, B)
  for i = 1 to A.rows
    for j = 1 to B.columns
      C[i, j] = 0
      for k = 1 to A.columns
        C[i, j] = C[i, j] + A[i, k] B[k, j]
  return C
```

e richiede $p \times q \times r$ prodotti tra scalari

Problema della moltiplicazione di matrici

Si vuole calcolare il prodotto

$$A_1 A_2 \ldots A_n$$

di *n* matrici di dimensioni

$$p_0 \times p_1 , p_1 \times p_2 , \ldots , p_{n-1} \times p_n$$

Poiché il prodotto di matrici è associativo possiamo calcolarlo in molti modi.

Esempio:

Per calcolare il prodotto $A_1 A_2 A_3$ di 3 matrici di dimensioni **200×5**, **5×100**, **100×5** possiamo:

a) moltiplicare A_1 per A_2 (**100000** prodotti scalari) e poi moltiplicare per A_3 la matrice **200×100** ottenuta (**100000** prodotti tra scalari).

In totale **200000** prodotti tra scalari.

b) moltiplicare A_2 per A_3 (**2500** prodotti tra scalari) e poi moltiplicare A_1 per la matrice **5×5** ottenuta (**5000** prodotti tra scalari).

In totale **7500** prodotti tra scalari.

Vogliamo trovare il modo per minimizzare il numero totale di prodotti tra scalari.

In quanti modi possiamo calcolare il prodotto?

Tanti quanti sono le parentesizzazioni possibili del prodotto $A_1 A_2 \ldots A_n$. Ad esempio per $n = 4$:

$$(A_1 (A_2 (A_3 A_4)))$$
$$(A_1 ((A_2 A_3) A_4))$$
$$((A_1 A_2) (A_3 A_4))$$
$$((A_1 (A_2 A_3)) A_4)$$
$$(((A_1 A_2) A_3) A_4)$$

Il numero $P(n)$ di parentesizzazioni possibili del prodotto $A_1 A_2 \ldots A_n$ di n matrici si esprime ricorsivamente come segue:

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

Si può dimostrare che $P(n)$ cresce in modo esponenziale.

Quindi, tranne per valori di n molto piccoli, non è possibile enumerare tutte le parentesizzazioni.

Passo 1: *struttura di una parentesizzazione ottima*

Supponiamo che una parentesizzazione ottima di $A_1 A_2 \ldots A_n$ preveda come ultima operazione il prodotto tra la matrice $A_{1..k}$ (prodotto delle prime k matrici $A_1 \ldots A_k$) e la matrice $A_{k+1..n}$ (prodotto delle ultime n-k matrici $A_{k+1} \ldots A_n$).

Le parentesizzazioni di $A_1 \ldots A_k$ e di $A_{k+1} \ldots A_n$ sono parentesizzazioni ottime per il calcolo di $A_{1..k}$ e di $A_{k+1..n}$.

Perché?

Passo 2: soluzione ricorsiva

Prendiamo come sottoproblemi il calcolo dei prodotti parziali $A_{i..j}$ delle matrici $A_i \ldots A_j$.

Ricordiamo che la generica matrice A_i ha dimensioni $p_{i-1} \times p_i$.

Di conseguenza la matrice prodotto parziale $A_{i..j}$ è una matrice $p_{i-1} \times p_j$ con lo stesso numero p_{i-1} di righe della prima matrice A_i e lo stesso numero p_j di colonne dell'ultima matrice A_j .

Se $i = j$ allora $A_{i..j} = A_i$ ed $m[i,i] = 0$.

Se $i < j$ allora $A_{i..j} = A_i \ldots A_j$ si può calcolare come prodotto delle due matrici $A_{i..k}$ e $A_{k+1..j}$ con k compreso tra i e j-1.

Il costo di questo prodotto è $p_{i-1} p_k p_j$.

Quindi

$$m[i,j] = \begin{cases} 0 & \text{se } i = j \\ \min\limits_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & \text{se } i < j \end{cases}$$

Passo 3

Esempio

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

p	i
30	1
35	2
15	3
5	4
10	5

35	15	5	10	20	25	p
1	2	3	4	5	6	j

$A_{1..1}$ m 0 k	$A_{1..2}$ 15750 1	$A_{1..3}$ 7900 1	$A_{1..4}$ 9400 3	$A_{1..5}$ 11900 3	$A_{1..6}$ 15125 3
	$A_{2..2}$ m 0 k	$A_{2..3}$ 2625 2	$A_{2..4}$ 4375 3	$A_{2..5}$ 7125 3	$A_{2..6}$ 10500 3
		$A_{3..3}$ m 0 k	$A_{3..4}$ 750 3	$A_{3..5}$ 1500 4	$A_{3..6}$ 5375 3
			$A_{4..4}$ m 0 k	$A_{4..5}$ 1000 4	$A_{4..6}$ 3500 5
				$A_{5..5}$ m 0 k	$A_{5..6}$ 5000 5
					$A_{6..6}$ m 0 k

$A_{1..1}A_{2..6}$: *0+10500+30×35×25 = 36750*
$A_{1..2}A_{3..6}$: *15750+5375+30×15×25 = 32375*
$A_{1..3}A_{4..6}$: *7900+3500+30×5×25 = 15150*
$A_{1..4}A_{5..6}$: *9400+5000+30×10×25 = 21900*
$A_{1..5}A_{6..6}$: *11900+0+30×20×25 = 26900*

Passo 3: *calcolo del costo minimo*

```
Matrix-Chain-Order(p, n)
    for i = 1 to n
        m[i, i] = 0
    for j = 2 to n
        for i = j-1 downto 1
            m[i, j] = ∞
            for k = i to j-1
                q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                if q < m[i, j]
                    m[i, j] = q
                    s[i, j] = k
    return m, s
```

Complessità: $O(n^3)$

Passo 4

Esempio

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

i \ j	1	2	3	4	5	6
1	$A_{1..1}$ m s	$A_{1..2}$ 1	$A_{1..3}$ 1	$A_{1..4}$ 3	$A_{1..5}$ 3	$A_{1..6}$ 3
2		$A_{2..2}$ m s	$A_{2..3}$ 2	$A_{2..4}$ 3	$A_{2..5}$ 3	$A_{2..6}$ 3
3			$A_{3..3}$ m s	$A_{3..4}$ 3	$A_{3..5}$ 3	$A_{3..6}$ 3
4				$A_{4..4}$ m s	$A_{4..5}$ 4	$A_{4..6}$ 5
5					$A_{5..5}$ m s	$A_{5..6}$ 5
6						$A_{6..6}$ m s

$A_{1..6}$

$(A_{1..3}\ A_{4..6})$

$((A_1\ A_{2..3})\ (A_{4..5}\ A_6))$

$((A_1\ (A_2\ A_3))\ ((A_4\ A_5)\ A_6))$

Passo 4: *stampa della soluzione ottima*

```
Print-Optimal-Parens(s, i, j)
    if i == j
        print "A_i"
    else
        k = s[i, j]
        print "("
        Print-Optimal-Parens(s, i, k)
        print "×"
        Print-Optimal-Parens(s, k+1, j)
        print ")"
```

Complessità: $O(n)$

Calcolo del prodotto di una sequenza di matrici

```
Matrix-Chain-Multiply(A_1...A_n , i, j, s)
  if i == j
     return A_i
  else
     k = s[i, j]
     A = Matrix-Chain-Multiply(A_1...A_n, i, k, s)
     B = Matrix-Chain-Multiply(A_1...A_n, k+1, j, s)
  return Matrix-Multiply(A, B)
```

Si potrebbe anche usare direttamente la definizione ricorsiva del costo minimo per il prodotto di matrici

$$m[i,j] = \begin{cases} 0 & \text{se } i = j \\ \min\limits_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & \text{se } i < j \end{cases}$$

per calcolarlo ricorsivamente senza usare le matrici m ed s.


```
Rec-Matrix-Chain-Cost(p, i, j)
  if i = j
    return 0
  else
    cmin = ∞
    for k = i to j-1
      q = Rec-Matrix-Chain-Cost(p, i, k) +
            Rec-Matrix-Chain-Cost(p, k+1, j) + p_{i-1} p_k p_j
      if q < cmin
        cmin = q
    return cmin
```

Complessità $T(n)$ con $n = j-i+1$

$$T(n) = \begin{cases} a & \text{se } n = 1 \\ a + \sum_{h=1}^{n-1} \left(T(h) + T(n-h) + b \right) & \text{se } n > 1 \end{cases}$$

$$T(n) = \begin{cases} a & \text{se } n = 1 \\ a + \sum_{h=1}^{n-1} \left(T(h) + T(n-h) + b \right) & \text{se } n > 1 \end{cases}$$

Per sostituzione si può dimostrare che

$$T(n) \geq c\,2^{n-1}$$

dove $c = \min(a,b)$.

Quindi $T(n) = \Omega(2^n)$.

$$T(n) \geq c2^{n-1} \qquad \text{per } c = \min(a,b)$$

$$T(1) = a \geq c = c2^{1-1}$$

$$T(n) = a + \sum_{k=1}^{n-1}(T(k) + T(n-k) + b)$$

$$\geq a + \sum_{k=1}^{n-1}(c2^{k-1} + c2^{n-k-1} + b)$$

$$= a + c\sum_{k=1}^{n-1}2^{k-1} + c\sum_{k=1}^{n-1}2^{n-k-1} + bn$$

$$= a + c(2^{n-1} - 1) + c(2^{n-1} - 1) + bn$$

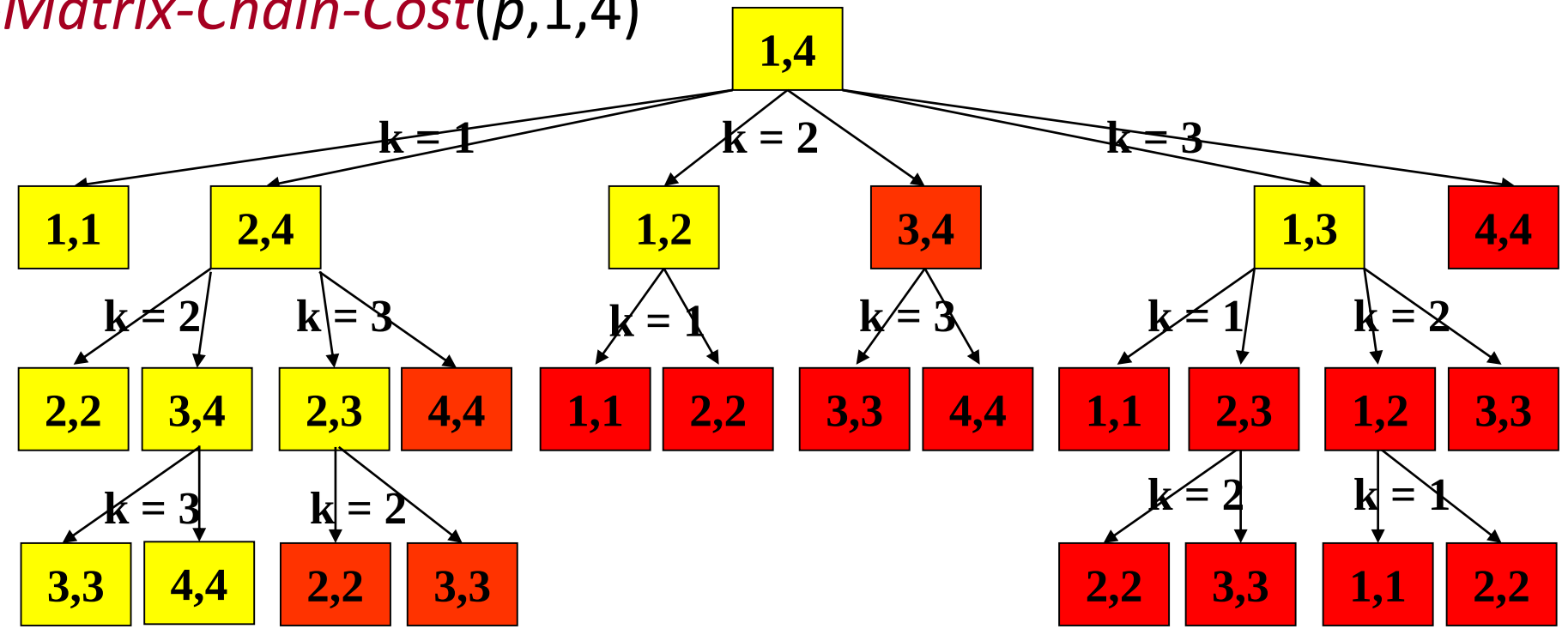
$$= a + c2^{n} - 2c + bn$$

$$\geq a + c2^{n} + (b-c)n$$

$$\geq c2^{n-1}$$

Causa della complessità esponenziale:

Rec-Matrix-Chain-Cost(*p*,1,4)



La complessità diventa esponenziale perché vengono risolti più volte gli stessi sottoproblemi.

Possiamo evitare il ricalcolo dei costi minimi dei sottoproblemi dotando la procedura ricorsiva di un blocco notes (una matrice m di dimensione $n \times n$) in cui annotare i costi minimi dei sottoproblemi già risolti.

```
Memoized-Matrix-Chain-Order(p, n)
    for i = 1 to n
        for j = i to n do
            m[i, j] = ∞
    return Memoized-Chain-Cost(p,1, n, m)
```


```
Memoized-Chain-Cost(p, i, j, m)
  if m[i, j] = ∞
    if i == j
      m[i, j] = 0
    else
      for k = i to j-1
        q = Memoized-Chain-Cost(p, i, k, M)
            + Memoized-Chain-Cost(p, k+1, j, M)
            + p_{i-1} p_k p_j
        if q < m[i, j]
          m[i, j] = q
  return m[i, j]
```

Complessità: $O(n^3)$

Problemi risolvibili con la programmazione dinamica

Abbiamo usato la programmazione dinamica per risolvere due problemi.
Cerchiamo ora di capire quali problemi si possono risolvere con questa tecnica.

Sono dei ***problemi di ottimizzazione*** in cui da un insieme (generalmente molto grande) di soluzioni possibili vogliamo estrarre una soluzione ottima rispetto ad una determinata misura.

Per poter applicare vantaggiosamente la programmazione dinamica bisogna che:

a) una soluzione ottima si possa costruire a partire da soluzioni ottime di sottoproblemi:

Proprietà di ***sottostruttura ottima*** **.**

b) che il numero di sottoproblemi distinti sia molto minore del numero di soluzioni possibili tra cui cercare quella ottima.

Altrimenti una enumerazione di tutte le soluzioni può risultare più conveniente.

Se ciò è vero significa che uno stesso problema deve comparire molte volte come sottoproblema di altri sottoproblemi.

Proprietà della ***ripetizione dei sottoproblemi*** .

Supposto che le condizioni a) e b) siano verificate, occorre scegliere l'ordine in cui calcolare le soluzioni dei sottoproblemi.

Tale ordine ci deve assicurare che nel momento in cui si risolve un sottoproblema le soluzioni dei sottoproblemi da cui esso dipende siano già state calcolate.

Ordine ***bottom-up***.

Alternativamente si può usare una procedura ricorsiva ***top-down*** che esprima direttamente la soluzione di un sottoproblema in termini delle soluzioni dei sottoproblemi da cui essa dipende.

In questo caso occorre però memorizzare le soluzioni trovate in modo che esse non vengano ricalcolate più volte.

Confronto tra algoritmo iterativo ***bottom-up*** ed algoritmo ricorsivo ***top-down*** con memoizzazione

Se per il calcolo della soluzione globale servono le soluzioni di tutti i sottoproblemi l'algoritmo ***bottom-up*** è migliore rispetto a quello ***top-down***.

Entrambi gli algoritmi calcolano una sola volta le soluzioni dei sottoproblemi, ma il secondo è ricorsivo ed inoltre effettua un controllo in più.

Se per il calcolo della soluzione globale servono soltanto alcune delle soluzioni dei sottoproblemi, l'algoritmo ***bottom-up*** le calcola comunque tutte mentre quello ***top-down*** calcola soltanto quelle che servono effettivamente.

In questo caso l'algoritmo ***top-down*** può risultare migliore di quello ***bottom-up***.

Il prossimo problema è un esempio di questa situazione.

Massima sottosequenza comune

In questo problema sono date due sequenze

$$X = x_1 x_2 \ldots x_m \quad \text{e} \quad Y = y_1 y_2 \ldots y_n$$

e si chiede di trovare la più lunga sequenza

$$Z = z_1 z_2 \ldots z_k$$

che è sottosequenza sia di X che di Y

Una sottosequenza di una sequenza X è una qualsiasi sequenza ottenuta da X cancellando alcuni elementi.

Il problema della massima sottosequenza ha molte applicazioni.

Per citarne solo alcune:

- individuare le parti comuni di due versioni dello stesso file (sequenze di caratteri ASCII).

- valutare la similitudine tra due segmenti di DNA (sequenze di simboli A,C,G,T).

Passo 1: *Struttura di una massima sottosequenza comune* (*LCS*)

Sia $Z = z_1 \ldots z_k$ una ***LCS*** di

$X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$

La sottostruttura ottima di Z discende dalle seguenti ***proprietà***:

1. se $x_m = y_n$ allora $z_k = x_m = y_n$ e Z_{k-1} è una ***LCS*** di X_{m-1} e Y_{n-1}
2. altrimenti se $z_k \neq x_m$ allora Z è ***LCS*** di X_{m-1} e Y
3. altrimenti $z_k \neq y_n$ e Z è una ***LCS*** di X e Y_{n-1}

Dimostrazione:

1. Supponiamo $x_m = y_n$

   Se $z_k \neq x_m = y_n$ potremmo aggiungere il simbolo $x_m = y_n$ in coda a Z ottenendo una sottosequenza comune più lunga contro l'ipotesi che Z sia una ***LCS***.

   Quindi $z_k = x_m = y_n$ e Z_{k-1} è sottosequenza comune di X_{m-1} e Y_{n-1} .

2. Se $z_k \neq x_m$ allora Z è sottosequenza di X_{m-1} e Y

   Essendo Z una ***LCS*** di X e Y essa è anche una ***LCS*** di X_{m-1} e Y.

3. il caso $z_k \neq y_n$ è simmetrico.

Data una sequenza

$$X = x_1 x_2 \ldots x_m$$

indicheremo con

$$X_i = x_1 x_2 \ldots x_i$$

il prefisso di X di lunghezza i.

L'insieme dei sottoproblemi è costituito quindi dalla ricerca delle ***LCS*** di tutte le coppie di prefissi (X_i, Y_j), per $i = 0,\ldots,m$ e $j = 0,\ldots,n$.

Totale $(m+1)(n+1) = \Theta(mn)$ sottoproblemi.

Passo 2: *soluzione ricorsiva*

Siano $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$ le due sequenze di cui vogliamo calcolare una ***LCS*** e per $i = 0,1,\ldots,m$ e $j = 0,1,\ldots,n$ sia $c_{i,j}$ la lunghezza di una ***LCS*** dei due prefissi X_i e Y_j .

Usando le proprietà che abbiamo appena dimostrato possiamo scrivere:

$$c_{i,j} = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c_{i-1,j-1} + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(c_{i,j-1}, c_{i-1,j}) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Passo 3

Esempio

X=ABCBDAB
Y=BDCABA

		Y		B	D	C	A	B	A
		j	0	1	2	3	4	5	6
X	i								
	0	c s	0	0	0	0	0	0	0
A	1	c s	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
B	2	c s	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
C	3	c s	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
B	4	c s	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
D	5	c s	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
A	6	c s	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
B	7	c s	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Terzo passo: *lunghezza di una LCS*

```
LCS-Length(X, Y, m, n)
    for i = 0 to m
        c[i, 0] = 0
    for j = 1 to n
        c[0, j] = 0
    for j = 1 to n
        for i = 1 to m
            if x_i == y_j
                    c[i, j] = c[i-1, j-1]+1, s[i, j] = "↖"
            elseif c[i-1, j] ≥ c[i, j-1]
                    c[i, j] = c[i-1, j], s[i, j] = "↑"
            else c[i, j] = c[i, j-1], s[i, j] = "←"
    return c,s
```

Quarto passo

Esempio

X=ABCBDAB
Y=BDCABA

LCS=BCBA

		Y		B	D	C	A	B	A
		0	1	2	3	4	5	6	j
X	0	c 0	0	0	0	0	0	0	
		s							
A	1	c 0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖	
		s							
B	2	c 0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←	
		s							
C	3	c 0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	
		s							
B	4	c 0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←	
		s							
D	5	c 0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑	
		s							
A	6	c 0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖	
		s							
B	7	c 0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑	
		s							

i

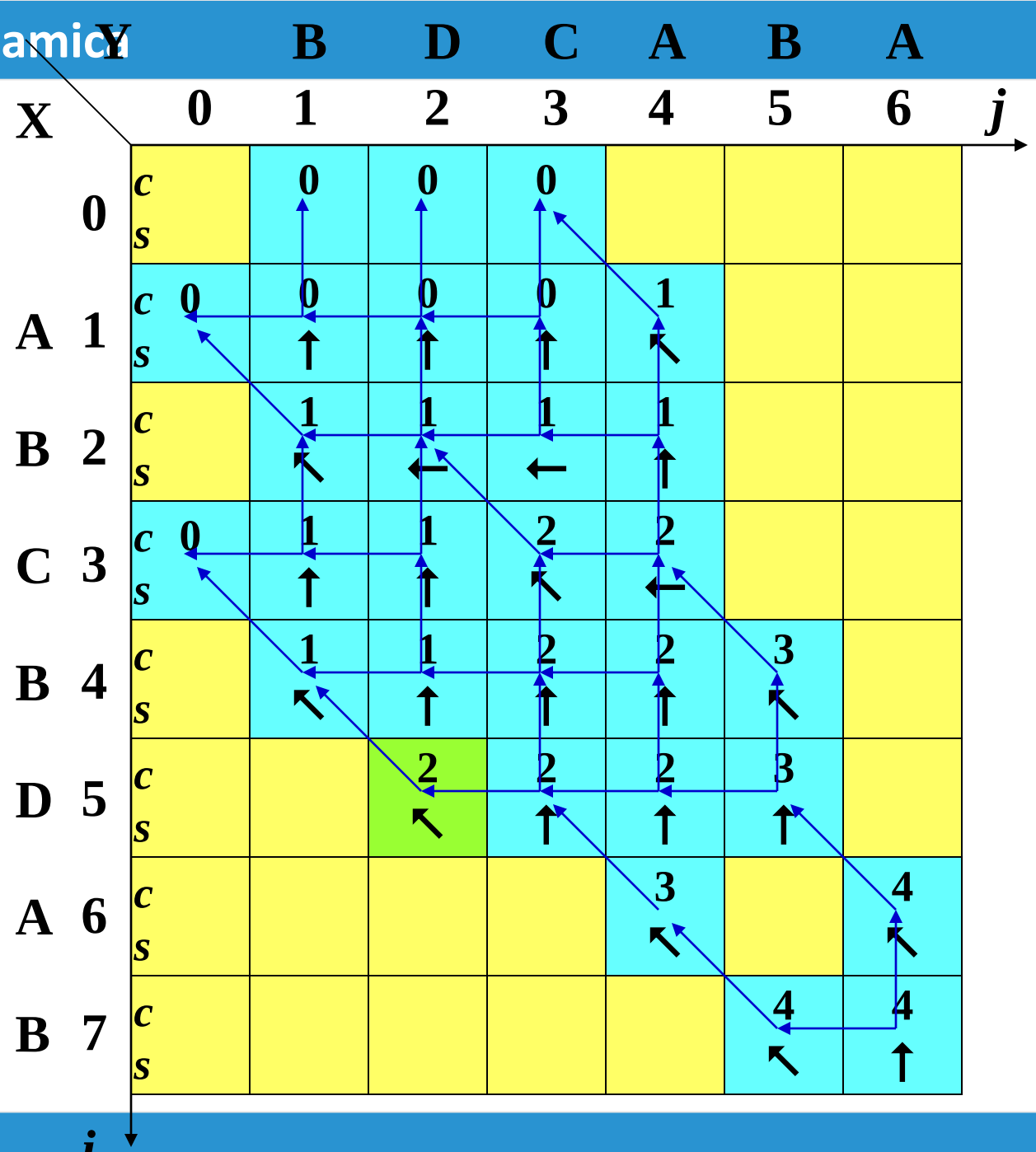
Quarto passo: *Stampa della LCS*

```
Print-LCS(X, s, i, j)
    if i > 0 and j > 0
        if s[i, j] == "↖"
                Print-LCS(X, s, i-1, j-1)
                print X[i]
        elseif s[i, j] == "↑"
                Print-LCS(X, s, i-1, j)
        else Print-LCS(X, s, i, j-1)
```

Metodo top-down

Esempio

X=ABCBDAB
Y=BDCABA

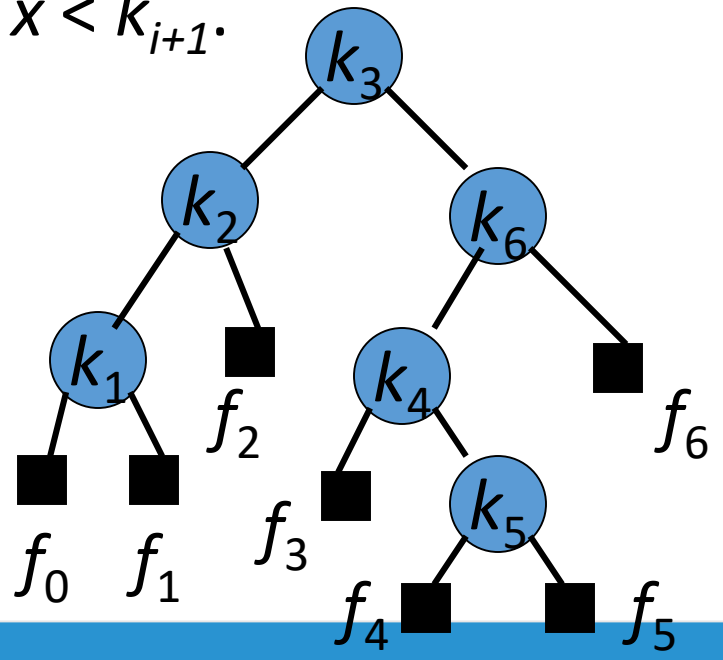


Albero di ricerca ottimo

Siano $k_1, k_2, \ldots, k_n$ le parole chiave prese in ordine lessicografico.

L'albero di ricerca ha n+1 foglie $f_0, f_1, \ldots, f_n$.

La ricerca di una parola x che non è parola chiave termina nella foglia f_i tale che $k_i < x < k_{i+1}$.



Albero di ricerca ottimo

Supponiamo di conoscere la probabilità p_i che una parola x in un programma C++ sia la parola chiave k_i e la probabilità q_i che $k_i < x < k_{i+1}$.

Vorremmo costruire un *albero di ricerca ottimo* che minimizzi la lunghezza aspettata (media) di una ricerca.

Primo passo: *struttura di un albero di ricerca ottimo*.

Sia T un albero di ricerca ottimo e sia T' un suo sottoalbero. I nodi interni di T' contengono un insieme di chiavi consecutive $k_s,\ldots, k_t$ e le foglie $f_{s-1},\ldots, f_t$. Scriveremo $T' = T_{s,t}$.

Siccome T è ottimo per $k_1,\ldots, k_n$ ed $f_0,\ldots, f_n$ anche T' deve esserlo per $k_s,\ldots, k_t$ ed $f_{s-1},\ldots, f_t$ altrimenti potremmo sostituirlo in T con uno migliore ottenendo un albero di ricerca migliore di T.

Secondo passo: *soluzione ricorsiva*.

Sia $T_{s,t}$ un sottoalbero ottimo per le chiavi $k_s,\ldots, k_t$ e le foglie $f_{s-1},\ldots, f_t$.

Sia x un valore da cercare. La probabilità che la ricerca termini in $T_{s,t}$ è $w_{s,t} = \sum_{i=s}^{t} p_i + \sum_{i=s-1}^{t} q_i$

Le probabilità condizionate che un valore x che sta in $T_{s,t}$ termini in $k_s,\ldots, k_t$ o in $f_{s-1},\ldots, f_t$ sono rispettivamente

$$\frac{p_i}{w_{s,t}} \text{ e } \frac{q_i}{w_{s,t}}$$

Se $t = s\text{-}1$ l'albero ha soltanto la foglia f_t e quindi $w_{s,t} = q_t$ ed $e_{s,t} = 1$.

Se $t \geq s$ l'albero ha una radice k_r , un sottoalbero sinistro con chiavi $k_s,\ldots, k_{r-1}$ e foglie $f_{s-1},\ldots, f_{r-1}$ ed un sottoalbero destro con chiavi $k_{r+1},\ldots, k_t$ e foglie $f_r,\ldots, f_t$. Quindi

$$e_{s,t} = \frac{p_r}{w_{s,t}} + \frac{w_{s,r-1}}{w_{s,t}}(e_{s,r-1} + 1) + \frac{w_{r+1,t}}{w_{s,t}}(e_{r+1,t} + 1)$$

$$= 1 + \frac{w_{s,r-1}e_{s,r-1} + w_{r+1,t}e_{r+1,t}}{w_{s,t}}$$

e dunque

$$e_{s,t} = \begin{cases} 1 & \text{se } t = s-1 \\ \min\limits_{s \le r \le t} \{1 + \dfrac{w_{s,r-1} e_{s,r-1} + w_{r+1,t} e_{r+1,t}}{w_{s,t}}\} & \text{se } t \ge s \end{cases}$$

Terzo passo

Esempio

	p	q
0		0.01
1	0.15	0.13
2	0.07	0.10
3	0.05	0.02
4	0.20	0.09
5	0.01	0.17
	0.48	0.52

$$w_{s,t} = w_{s,t-1} + p_t + q_t$$

$$e_{s,t} = \begin{cases} 1 & \text{se } t = s-1 \\ \min\limits_{s \le r \le t}\{1 + \dfrac{w_{s,r-1}e_{s,r-1} + w_{r+1,t}e_{r+1,t}}{w_{s,t}}\} & \text{se } t \ge s \end{cases}$$

s \ t	0	1	2	3	4	5
1	w .01 e 1 r	0.29 1.48 1	0.46 1.49 1	0.53 1.41 2	0.82 2.02 4	1.00 2.32 4
2		w .13 e 1 r	0.30 1.53 2	0.37 1.94 2	0.66 1.88 2	0.84 2.29 4
3			w .10 e 1 r	0.17 1.71 3	0.46 1.49 4	0.64 2.28 4
4				w .02 e 1 r	0.31 1.35 4	0.49 2.12 4
5					w .09 e 1 r	0.27 1.96 5
6						w .17 e 1 r

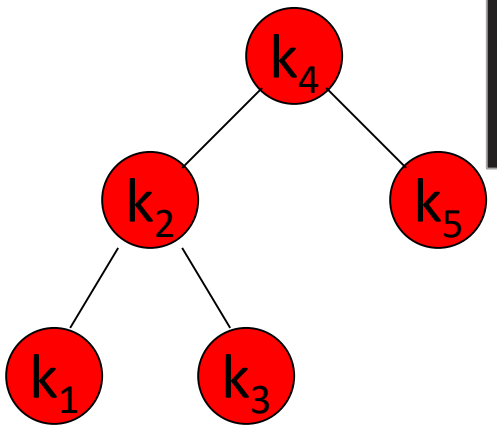
Terzo passo: *calcolo del numero medio di passi*.

```
Optimal-BST(p,q,n)
    e[1:n+1,0:n], w[1:n+1,0:n] e root[1:n,0:n] sono nuove tabelle
    for i = 1 to n+1
        e[i,i-1] = q_{i-1}
        w[i,i-1] = q_{i-1}
    for l = 1 to n
        for i = 1 to n-l+1
            j = i+l-1
            e[i,j] = ∞
            w[i,j] = w[i,j-1] +p_j+q_j
            for r = i to j
                t = e[i,r-1]+e[r+1,j]+w[i,j]
                if t < e[i,j]
                    e[i,j] = t
                    root[i,j] = r
    return e ed r
```

Quarto passo

Esempio

s \ t	0	1	2	3	4	5
1	*w* .01 *e* 1 *r*	0.29 1.48 1	0.46 1.49 1	0.53 1.41 2	0.82 2.02 4	1.00 2.32 4
2		*w* .13 *e* 1 *r*	0.30 1.53 2	0.37 1.94 2	0.66 1.88 2	0.84 2.29 4
3			*w* .10 *e* 1 *r*	0.17 1.71 3	0.46 1.49 4	0.64 2.28 4
4				*w* .02 *e* 1 *r*	0.31 1.35 4	0.49 2.12 4
5					*w* .09 *e* 1 *r*	0.27 1.96 5
6						*w* .17 *e* 1 *r*



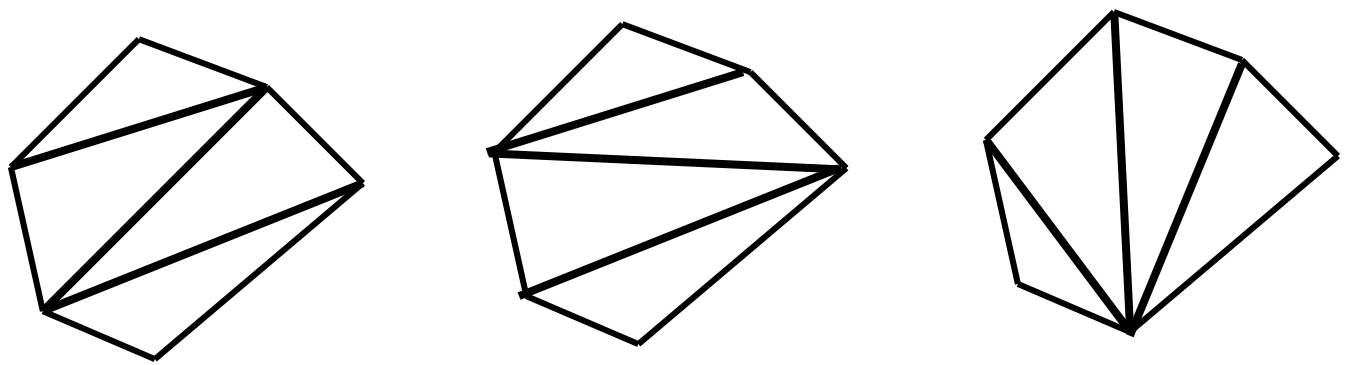
Quarto passo: *Costruzione dell'albero ottimo.*

```
Construct-Optimal-BST(root,i,j,last)
    if i == j
        return
    if last == 0
        print root[i,j] " è la radice"
    elseif  j<last
        print root[i,j] " è il figlio sinistro di " last
    else
        print root[i,j] " è il figlio destro di " last
    Construct-Optimal-BST(root,i,root[i,j]-1,root[i,j])
    Construct-Optimal-BST(root,i,root[i,j]+1,root[i,j])
```

Triangolazione ottima

Una triangolazione di un poligono convesso è una suddivisione del poligono in triangoli ottenuta tracciando delle diagonali che non si intersecano .

Vi sono più triangolazioni possibili dello stesso poligono



Triangolazione ottima

In questo problema sono dati i vertici $q_1, q_2, \ldots, q_n$ di un poligono convesso P presi in ordine antiorario.

Ad ogni triangolo T è attribuito un costo $c(T)$.

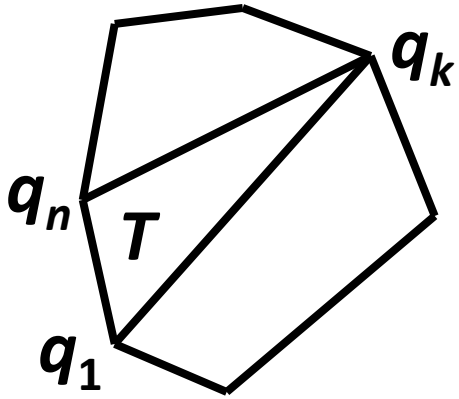
Ad esempio $c(T)$ potrebbe essere la lunghezza del perimetro, la somma delle altezze, il prodotto delle lunghezze dei lati, (l'area ?), ecc.

Si vuole trovare una triangolazione del poligono P tale che la somma dei costi dei triangoli sia minima.

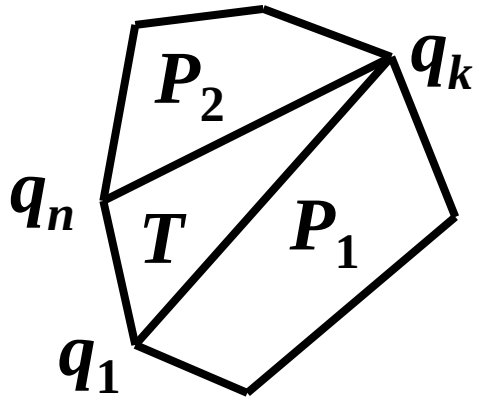
*In quanti modi possiamo suddividere in triangoli un poligono convesso di **n** vertici?*

Ogni lato del poligono ***P*** appartiene ad un solo triangolo della triangolazione .

Siano $q_1 q_k q_n$ i vertici del triangolo ***T*** a cui appartiene il lato $q_1 q_n$



Il triangolo T suddivide il poligono P nel triangolo T stesso e nei due poligoni P_1 e P_2 di vertici $q_1,\ldots,q_k$ e $q_k,\ldots,q_n$



Il vertice q_k può essere scelto in n-2 modi diversi e i due poligoni hanno rispettivamente $n_1 = k$ ed $n_2 = n-k+1$ vertici.

P_1 quando $k = 2$ e P_2 quando $k = n-1$ sono poligoni degeneri, ossia sono un segmento.

Il numero $T(n)$ di triangolazioni possibili di un poligono di n vertici si esprime ricorsivamente come segue

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

E' facile verificare che $T(n) = P(n-1)$ dove $P(n)$ sono le parentesizzazioni del prodotto di n matrici.

Quindi $T(n)$ cresce esponenzialmente.

Primo passo: *struttura di una triangolazione ottima*.

Supponiamo che una triangolazione ottima suddivida il poligono convesso P di vertici $q_1 q_2 \ldots q_n$ nel triangolo T di vertici $q_1 q_k q_n$ e nei due poligoni P_1 e P_2 di vertici $q_1 \ldots q_k$ e $q_k \ldots q_n$ rispettivamente.

Le triangolazioni subordinate di P_1 e di P_2 sono triangolazioni ottime. Perché?

Secondo passo: *soluzione ricorsiva*

I sottoproblemi sono le triangolazioni dei poligoni $P_{i..j}$ di vertici $q_i \ldots q_j$. Sia $c_{i,j}$ la somma dei costi dei triangoli di una triangolazione ottima di $P_{i..j}$.

Se $j = i+1$ allora $P_{i..j}$ è degenere e $c_{i,j} = 0$.

Se $j > i+1$ allora $P_{i..j}$ si può scomporre in un triangolo T di vertici $q_i q_k q_j$ e nei due poligoni P_1 e P_2 di vertici $q_i \ldots q_k$ e $q_k \ldots q_j$ con $i < k < j$

$$c_{i,j} = \begin{cases} 0 & \text{se } j \le i+1 \\ \min\limits_{i<k<j}\left(c_{i,k} + c_{k,j} + c(q_i q_k q_j)\right) & \text{se } j > i+1 \end{cases}$$

Terzo passo: *calcolo del costo minimo*

```
Triangulation-Cost(q, n)
  for i = 1 to n-1
    c[i, i+1] = 0
  for j = 3 to n
    for i = j-2 downto 1
      c[i, j] = ∞
      for k = i+1 to j-1
        q = c[i, k]+c[k, j]+c(q_i q_k q_j)
        if q < c[i, j]
          c[i, j] = q
          s[i, j] = k
  return c e s
```

Complessità: $O(n^3)$

Quarto passo: *Stampa triangolazione*

```
Print-Triangulation(s, i, j)
    if j > i+1
        k = s[i, j]
        Print-Triangulation(s, i, k)
        print "triangolo: ", i, j, k
        Print-Triangulation(s, k, j)
```

Complessità: $O(n)$