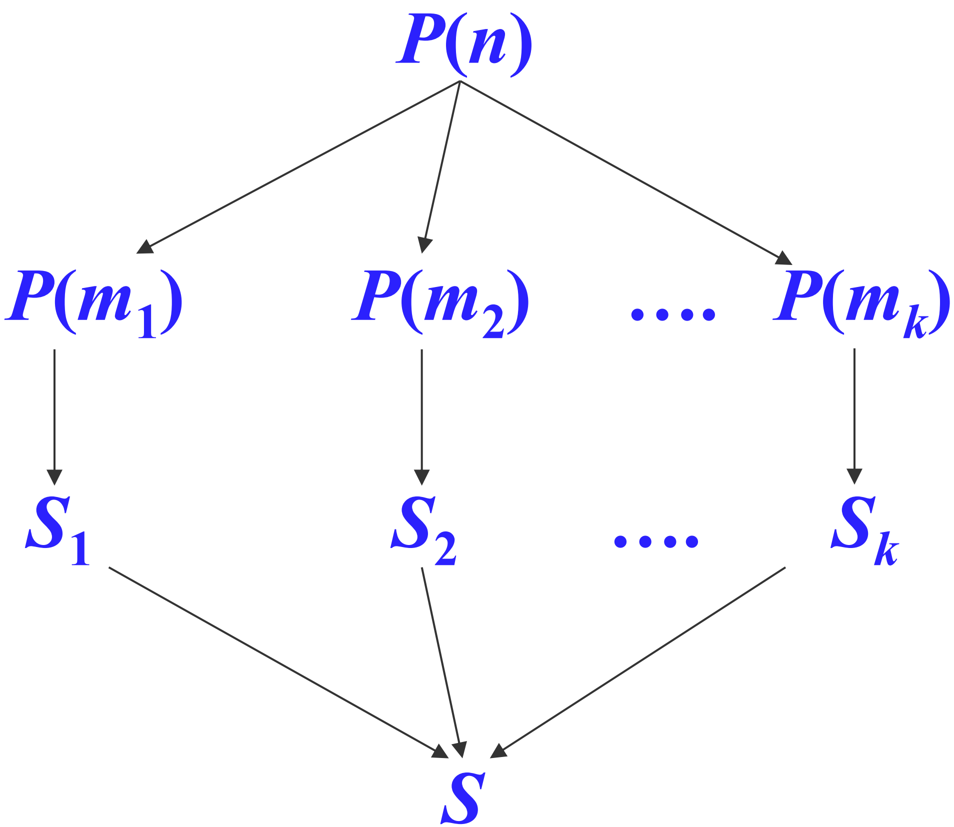
Dynamic Programming

又稱

動態規劃

經常被用來解決最佳化問題

Dynamic Programming



- 與 divide-and-conquer 法類似, 是依遞迴方式設計的演算法.
- 與 divide-and-conquer 的最大差別在於子問題間不是獨立的, 而是重疊的.

鐵條切割問題 (Rod cutting problem)

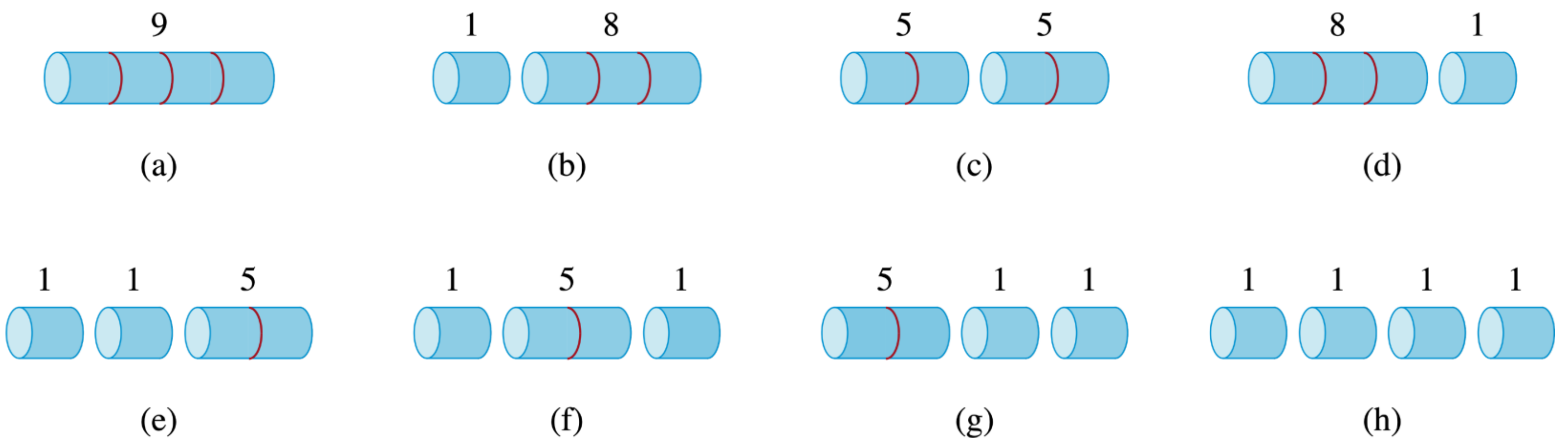
給一段長度為N(整數)單位的鐵條, 令i為任一正整數, 假設p[i]表示長度為i的鐵條可以賣出的價格,試問應如何切割該鐵條使得其總賣價最高?

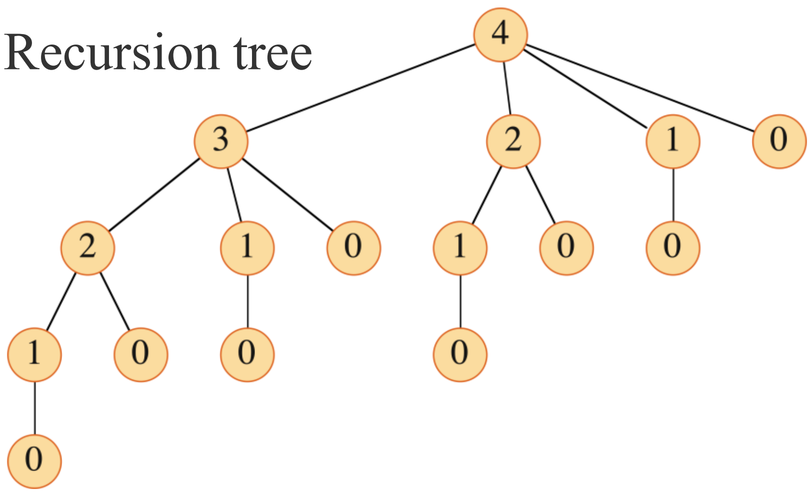
例：

長度 i	1	2	3	4	5	6	7	8	9	10
價格p[i]	1	5	8	9	10	17	17	20	24	30

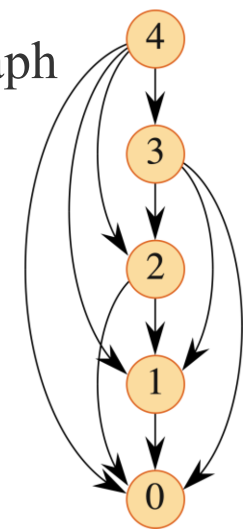
N=7,
7=3+4可賣得17
7=1+6=2+2+3 可賣得18.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30





Subproblem graph



鐵條切割問題 (Rod cutting problem)

假設最佳的答案是將鐵條切割成k段,即

$N=i_1+i_2+\ldots+i_k$

$r[N]=p[i_1]+\ldots+p[i_k]$ ---- 總價格

$r[N] = \max_{i=1..n} \{ p[i] + r[N-i] \}$,

$r[0]=0$.

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

鐵條切割問題 (Rod cutting problem)

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Time =O(n^2), why?

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

長度 i	1	2	3	4	5	6	7	8	9	10
價格p[i]	1	5	8	9	10	17	17	20	24	30
r[i]	1	5	8	10	13	17	18	22	25	30
s[i]	1	2	3	2	2	6	1	2	3	10

Crazy eights puzzle

- Given a sequence of cards c[0], c[1],…,c[n-1], e.g. 7H, 6H, 7D, 3D, 8C, JS,..
- Find the longest subsequence $c[i_1], c[i_2],\ldots, c[i_k]$, $(i_1 < i_2 < \ldots < i_k)$, where $c[i_j]$ and $c[i_{j+1}]$ have the same suit or rank or one has rank 8.-- match

•Let T[i] be the length of the longest subsequence starting at c[i].

•T[i]= 1 + max {T[j]: c[i] and c[j] have a match and j < n }

•Optimal solution: max {T[i]}.

串列矩陣相乘 (定義)

給一串列的矩陣 $\langle A_1, A_2, \ldots, A_n \rangle$, 其中矩陣 A_i 的大小為 $p_{i-1} \times p_i$, 找一計算 $A_1A_2\ldots A_n$ 乘積的方式, 使得所用 scalar 乘法的計算量為最少.

例： $A_1 \times A_2 \times A_3 \times A_4$

p_i :13　5　89　3　34

總共有 5 種方式來計算這 4 個矩陣的乘積：

$(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$,

$((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

串列矩陣相乘(例)

$(A_1(A_2(A_3A_4))) \rightarrow A_1 \times (A_2A_3A_4) \rightarrow A_2 \times (A_3A_4) \rightarrow A_3 \times A_4$

$$\begin{aligned} \text{cost} &= 13*5*34 + 5*89*34 + 89*3*34 \\ &= 2210 + 15130 + 9078 \\ &= 26418 \end{aligned}$$

$A_1 \times A_2 \times A_3 \times A_4$

13 5 89 3 34

$(A_1(A_2(A_3A_4)))$, costs = 26418

$(A_1((A_2A_3)A_4))$, costs = 4055

$((A_1A_2)(A_3A_4))$, costs = 54201

$((A_1(A_2A_3))A_4)$, costs = 2856

$(((A_1A_2)A_3)A_4)$, costs = 10582

Catalan Number

For any n, # ways to fully parenthesize the product of a chain of n+1 matrices

= # binary trees with n nodes.

= # permutations generated from 1 2 … n through a stack.

= # n pairs of fully matched parentheses.

= n-th Catalan Number = $C(2n, n)/(n+1) = \Omega(4^n/n^{3/2})$

乘法樹

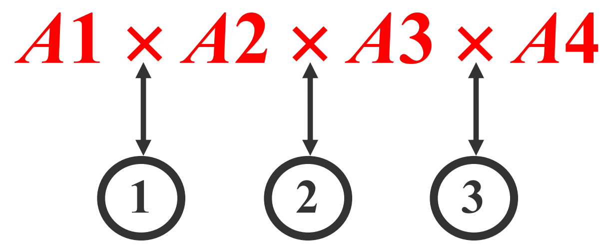
$(A_1(A_2(A_3A_4)))$

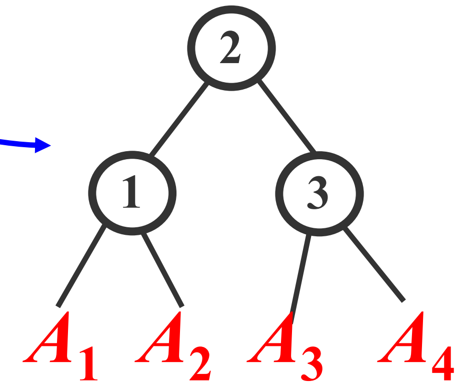
$(A_1((A_2A_3)A_4))$

$\boxed{((A_1A_2)(A_3A_4))}$

$((A_1(A_2A_3))A_4)$

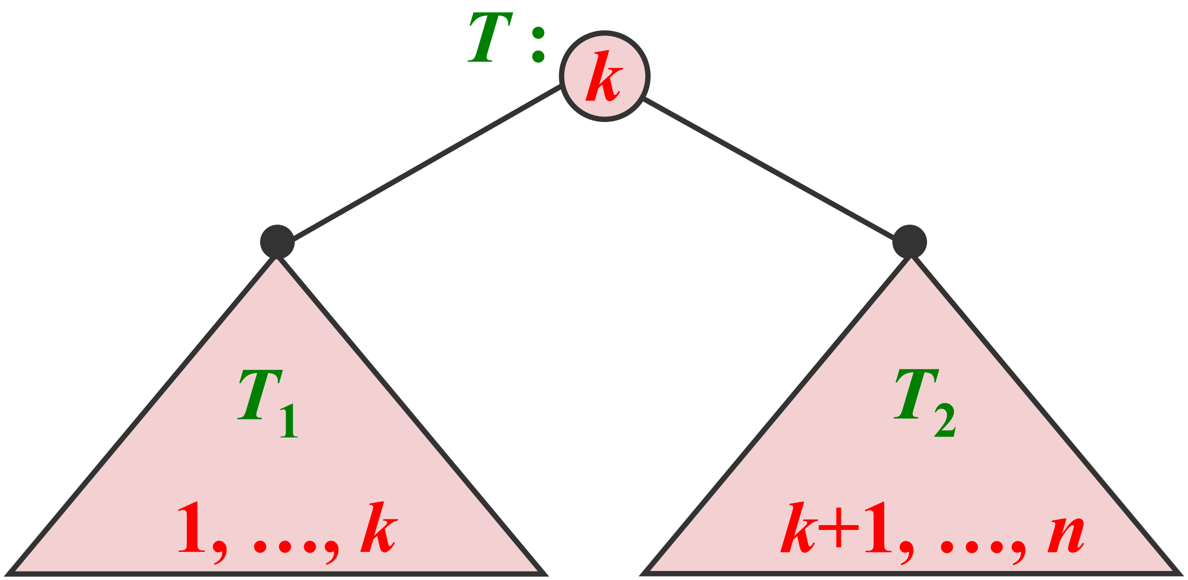
$(((A_1A_2)A_3)A_4)$





串列矩陣相乘(設計1)

- If T is an optimal solution for $A_1, A_2, \ldots, A_n$



- then, T_1 (resp. T_2) is an optimal solution for $A_1, A_2, \ldots, A_k$ (resp. $A_{k+1}, A_{k+2}, \ldots, A_n$).

串列矩陣相乘 (設計2)

- Let $m[i, j]$ be the minmum number of scalar multiplications needed to compute the product $A_i \ldots A_j$, for $1 \le i \le j \le n$.
- If the optimal solution splits the product $A_i \ldots A_j = (A_i \ldots A_k) \times (A_{k+1} \ldots A_j)$, for some k, $i \le k < j$, then $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$. Hence, we have :

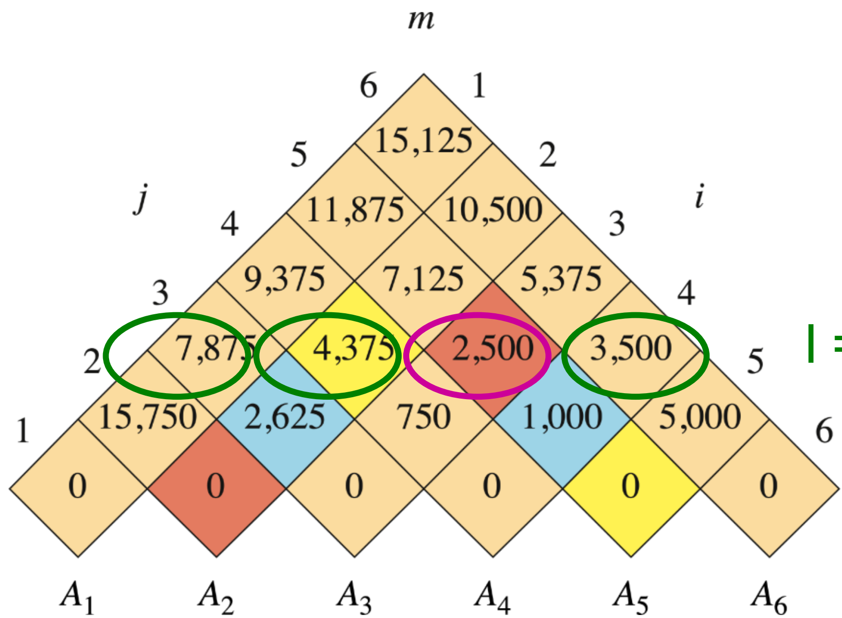
$$
\begin{aligned}
m[i, j] &= \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} \\
&= 0 \text{ if } i = j
\end{aligned}
$$

$< P_0, P_1, \ldots, P_n >$ $\quad A_{P_0 \times P_1} A_{P_1 \times P_2} \ldots A_{P_{n-1} \times P_n}$

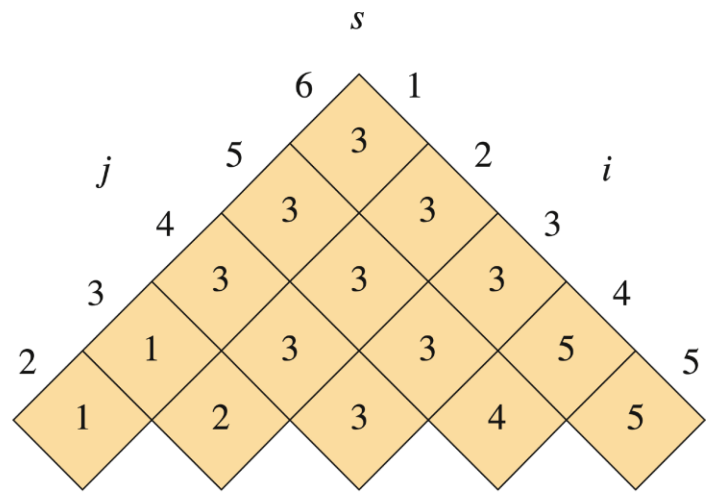
MATRIX-CHAIN-ORDER(P)

```
n = p.length -1;
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables;
for i= 1 to n:     m[i, i]=0;
for l= 2 to n:
      { for i = 1 to n – l + 1
              { j = i + l- 1;
                m[i, j] = ∞;
                for k = i to j-1
                      { q = m[i, k] + m[k+1, j]+ P_{i-1}P_kP_j
                        if q<m[i, j]
                              { m[i, j] = q ;  s[i, j] = k ;}
                      }  }  }
return m and s
```

Time $= O(n^3)$



l =3



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

$$m[3,5] = \min \begin{cases} m[3,4]+m[5,5] + 15*10*20 \\ =750 + 0 + 3000 = 3750 \\ m[3,3]+m[4,5] + 15*5*20 \\ =0 + 1000 + 1500 = 2500 \end{cases}$$

串列矩陣相乘 (實例)

- **Consider an example with sequence of dimensions <5,2,3,4,6,7,8>**

$$m[i,j] = \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}\,p_k p_j\}$$

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Constructing an optimal solution

Each entry s[i, j]=k records that the optimal parenthesization of $A_iA_{i+1}\ldots A_j$ splits the product between A_k and A_{k+1}

$A_{i..j} \rightarrow (A_{i..s[i..j]})(A_{s[i..j]+1..j})$

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

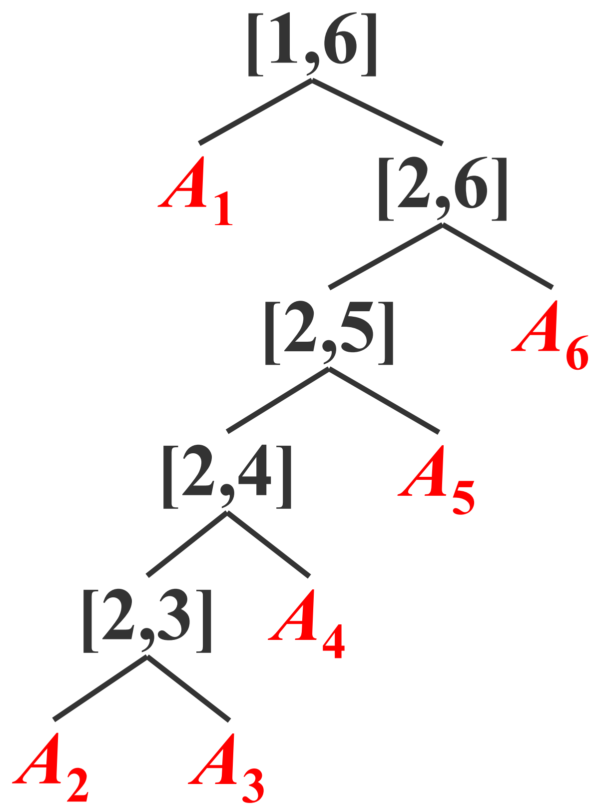
串列矩陣相乘 (找解)

$m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1}\, p_k p_j\}$

$s[i, j]$ = a value of k that gives the minimum

s	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

$A_1((((A_2A_3)A_4)A_5)\,A_6)$



串列矩陣相乘 (分析)

$$m[i,j] = \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}\,p_k p_j\}$$

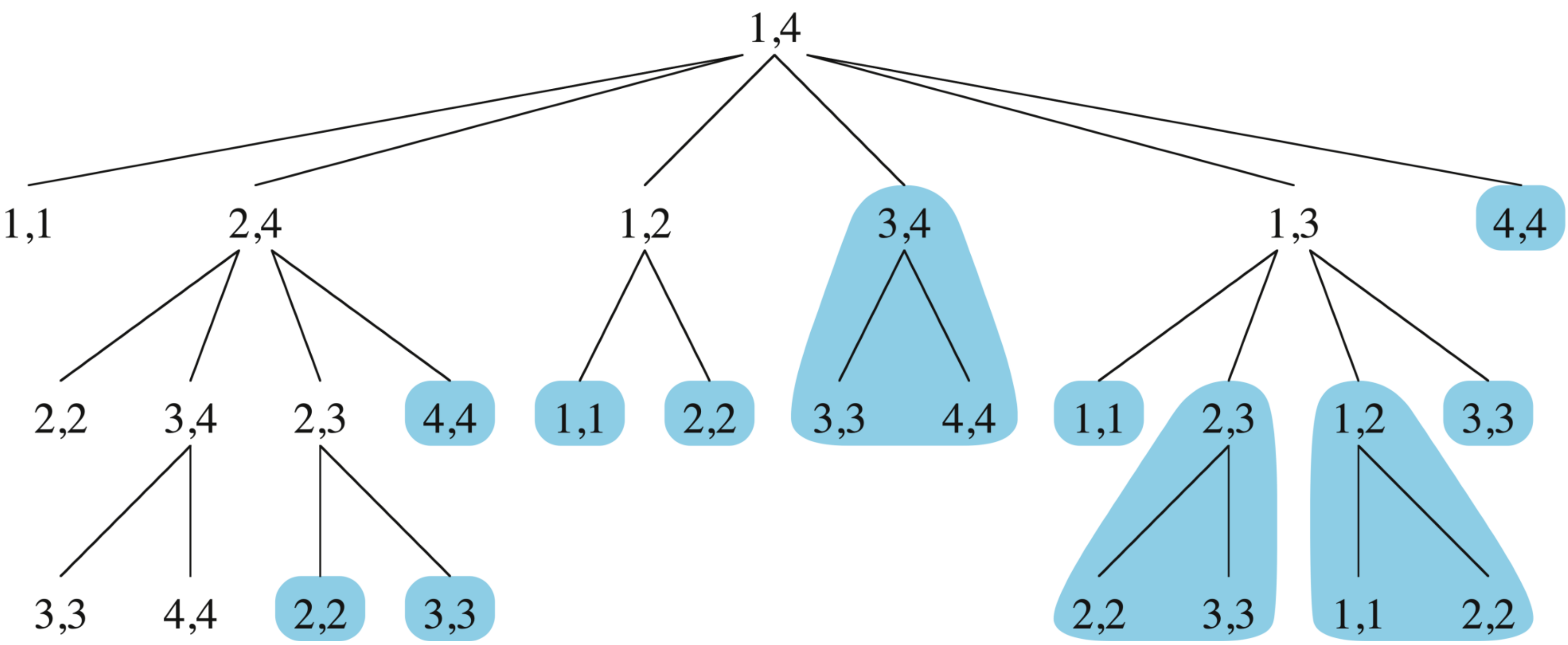
- To fill the entry $m[i, j]$, it needs $\Theta(j-i)$ operations. Hence the execution time of the algorithm is

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{j=1}^{n}\sum_{i=1}^{j}(j-i) = \sum_{j=1}^{n}[j^2 - \frac{j(j+1)}{2}]$$
$$= \sum_{j=1}^{n}\Theta(j^2) = \Theta(n^3)$$

Time: $\Theta(n^3)$

Space: $\Theta(n^2)$

Recursive-Matrix-Chain(p,1,4)



```
MEMOIZED-MATRIX-CHAIN(P)
    n = p.length -1;
    let m[1..n, 1..n] be a new table;
    for i= 1 to n
        for j= i to n
                m[i, j]= ∞;
    return Lookup-Chain(m, p, 1, n)
```

$<P_0, P_1, \dots, P_n>$

```
Lookup-Chain(m, P, i, j)
    if m[i, j] < ∞  return m[i, j];
    if  i==j   m[i, j] = 0
    else for k=i  to  j-1
        { q= Lookup-Chain(m, P, i, k)
                + Lookup-Chain(m, P, k+1, j) + P_{i-1}P_kP_j ;
          if  q < m[i, j]  m[i, j] = q;}
    return m[i, j] ; }
```

time: $O(n^3)$ space: $\Theta(n^2)$

設計 DP 演算法的步驟

1. **Characterize** the structure of an optimal solution.
2. **Derive** a **recursive formula** for computing the values of optimal solutions.
3. **Compute the value** of an optimal solution **in a bottom-up fashion** (top-down is also applicable).
4. **Construct** an optimal solution in a top-down fashion.

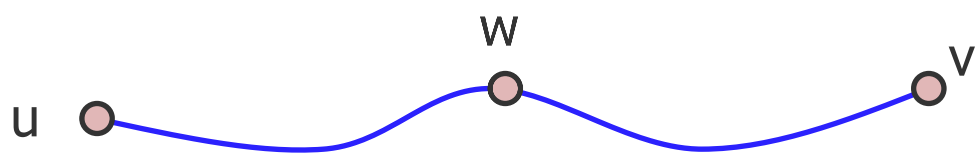
Elements of Dynamic Programming

- **Optimal substructure (a problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems)**
- **Overlapping subproblems**
- **Reconstructing an optimal solution**
- **Memoization**

Given a directed graph $G=(V, E)$ and vertices $u, v \in V$

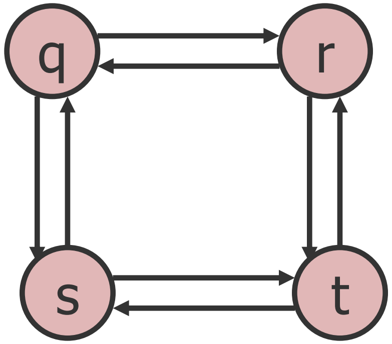
Unweighted shortest path: Find a path from u to v consisting the fewest edges. Such a path must be simple (no cycle).

- Optimal substructure? YES.



Unweighted longest simple path: Find a simple path from u to v consisting the most edges.

- Optimal substructure? NO.
- $q \rightarrow r \rightarrow t$ is longest but $q \rightarrow r$ is not the longest between q and r.



Printing neatly 定義

- Given a sequence of n words of lengths l[1], l[2],…,l[n], measured in characters, want to print it neatly on a number of lines, of which each has at most M characters.
- If a line has words i through j, i<=j, and there is exactly one space between words, the ***cube*** of number of extra space characters at the end of the line is:

  $B[i, j] = (M - j + i - (l[i]+l[i+1]+\ldots+l[j]))^3$.
- Want to *minimize* the **sum** over all lines (except the last line) of **B[i,j]**'s.
- Let c[i] denote the minimum cost for printing words i through n.
- $c[i] = \min_{i<j<=i+p} (c[j+1] + B[i,j])$, where p is the maximum number of words starting from i-th word that can be fitted into a line.

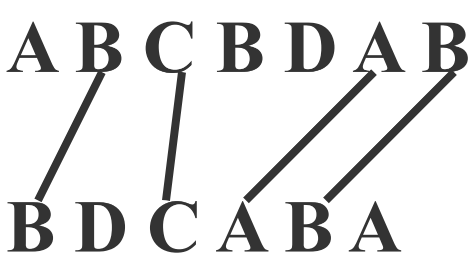
Longest Common Subsequence (定義)

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ find a maximum-length common subsequence of X and Y.

例1：Input: ABCBDAB BDCABA

C.S.'s: AB, ABA, BCB, BCAB, BCBA …

Longest: BCAB, BCBA, … Length = 4



例 2 :

vintner

writers

Step 1: Characterize LCS

Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be a LCS of

$X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $\langle z_1, z_2, \ldots, z_{k-1} \rangle$ is a LCS of $\langle x_1, x_2, \ldots, x_{m-1} \rangle$ and

   $\langle y_1, y_2, \ldots, y_{n-1} \rangle$.

1. If $z_k \neq x_m$, then Z is a LCS of $\langle x_1, x_2, \ldots, x_{m-1} \rangle$ and Y.

2. If $z_k \neq y_n$, then Z is a LCS of X and

   $\langle y_1, y_2, \ldots, y_{n-1} \rangle$.

Step 2: A recursive solution

- Let $C[i, j]$ be the length of an LCS of the prefixes $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$, for $1 \le i \le m$ and $1 \le j \le n$. We have :

$$
\begin{aligned}
C[i, j] &= 0 \quad \text{if } i = 0, \text{ or } j = 0 \\
&= C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j \\
&= \max(C[i, j-1], C[i-1, j]) \text{ if } i, j > 0 \text{ and } x_i \neq y_j
\end{aligned}
$$

Step 3: Computing the length of an LCS

```
LCS-Length(X,Y, m, n)
let b[1..m, 1..n] and c[0..m, 0..n] be new tables;
for I = 1 to m:
    c[i, 0] = 0;
for j = 0 to n:
    c[0, j] = 0;
for i= 1 to m:
     for j= 1 to n:
          { if x_i == y_j:
                { c[i,j] = c[i-1,j-1]+1;   b[i,j] = "↖"; }
            elseif c[i-1, j] ≥ c[i, j-1]:
                { c[i, j] = c[i-1, j];     b[i, j] = "↑"; }
            else: { c[i, j] = c[i, j-1];   b[i, j] = "←"; }
                }
return b, c
```

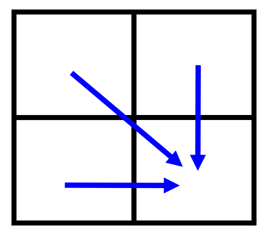
Step 4: Constructing an LCS

```
Print-LCS(b, X, i, j)
    if i==0 or j==0:  return;
    if b[i,j] == "↖ ":
        { Print-LCS(b, X, i-1, j-1);
          print x_i;
          }
    elseif b[i,j] == "↑":
        Print-LCS(b, X, i-1, j);
    else: Print-LCS(b, X, i, j-1);
```

Longest Common Subsequence (例+分析)

$$C[i, j] = 0 \quad \text{if } i = 0, \text{ or } j = 0$$
$$= C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j$$
$$= \max(C[i, j-1], C[i-1, j]) \text{ if } i, j > 0 \text{ and } x_i \neq y_j$$

	A	B	C	B	D	A	B
B	0	1	1	1	1	1	1
D	0	1	1	1	2	2	2
C	0	1	2	2	2	2	2
A	1	1	2	2	2	3	3
B	1	2	2	3	3	3	4
A	1	2	2	3	3	4	4

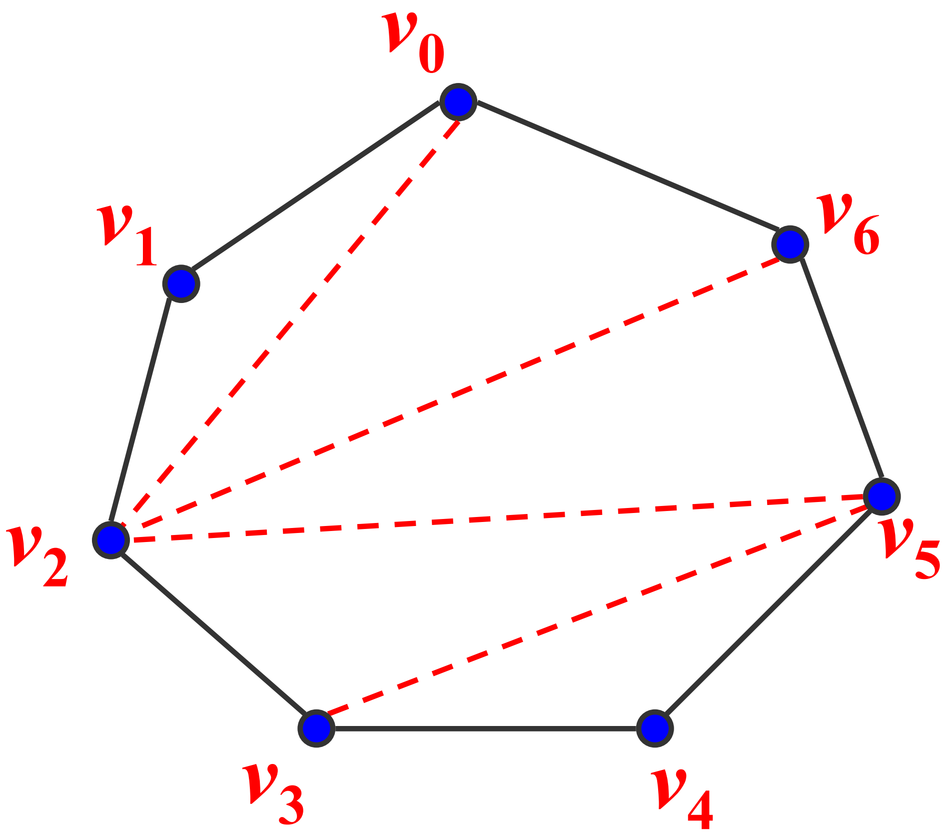


Time: $\Theta(mn)$

Space: $\Theta(mn)$

可得一LCS: BCBA

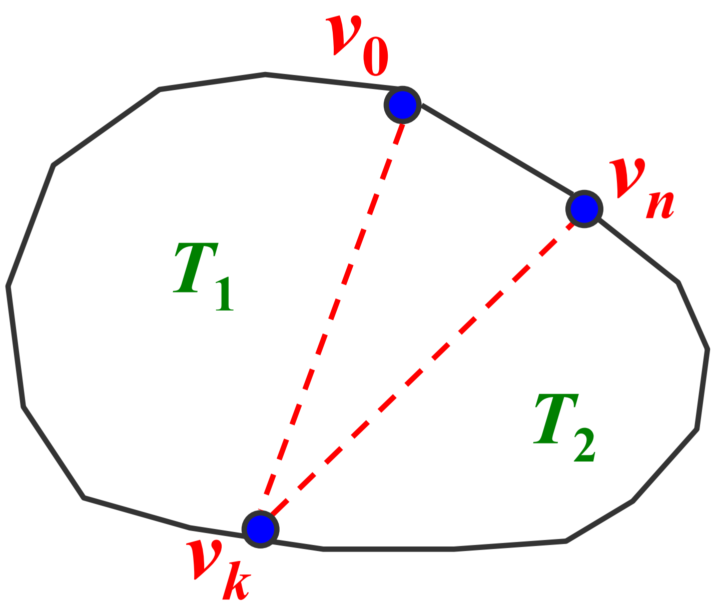
Optimal Polygon Triangulation



Find a *triangulation* s.t. the sum of the weights of the triangles in the triangulation is minimized.

Optimal Polygon Triangulation (設計1)

- If T is an optimal solution for $\langle v_0, v_1, \ldots, v_n\rangle$



$$T = T_1 \cup T_2 \cup \{\overline{v_0 v_k}, \overline{v_k v_n}\}$$

- then, T_1 (resp. T_2) is an optimal solution for $\langle v_0, v_1, \ldots, v_k\rangle$ (resp. $\langle v_k, v_{k+1}, \ldots, v_n\rangle$), $1 \le k < n$.

Optimal Polygon Triangulation (設計2)

- Let $t[i, j]$ be the weight of an optimal triangulation of the polygon $\langle v_{i-1}, v_i, \ldots, v_j \rangle$, for $1 \le i < j \le n$.
- If the triangulation splits the polygon into $\langle v_{i-1}, v_i, \ldots, v_k \rangle$ and $\langle v_k, v_{k+1}, \ldots, v_n \rangle$ for some k, then $t[i, j] = t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j)$. Hence, we have :

$$
\begin{aligned}
t[i, j] &= \min_{i \le k < j}\{t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j)\} \\
&= 0 \text{ if } i = j
\end{aligned}
$$

Optimal binary search trees:

$k=(k_1,k_2,\ldots,k_n)$ of n distinct keys in sorted order $(k_1<k_2<\ldots< k_n)$

Each key k_i has a probability p_i that a search will be for k_i

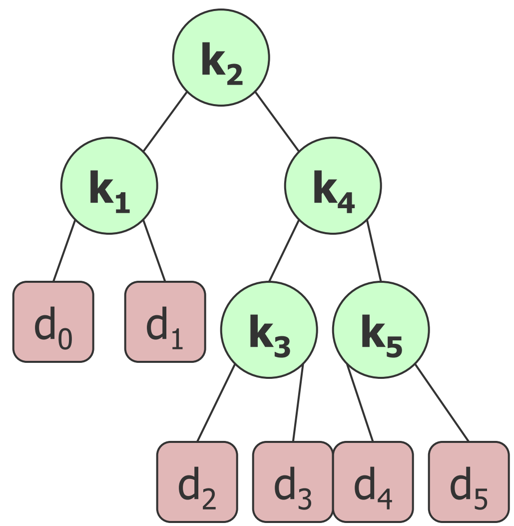
Some searches may fail, so we also have n+1 dummy keys: $d_0,d_1,\ldots,d_n$, where $d_0<k_1$, $k_n<d_n$ and $k_i<d_i<k_{i+1}$ for $i=1,\ldots,n-1$.

For each dummy key d_i, we have a probability q_i that a search will correspond to d_i .

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Optimal binary search trees:範例

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1



Expected search cost: 2.8

E[Search cost in T]

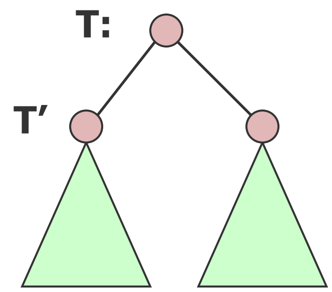
$$= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i$$

Goal: **Given a set of probabilities, want to construct a binary search tree whose expected search cost is smallest .**

Step 1: The structure of an optimal BST.

Consider any subtree of a BST, which must contain contiguous keys $k_i,\ldots,k_j$ for some $1 \le i \le j \le n$ and must also have as its leaves the dummy keys $d_{i-1},\ldots,d_j$.



Optimal-BST has the property of optimal substructure.

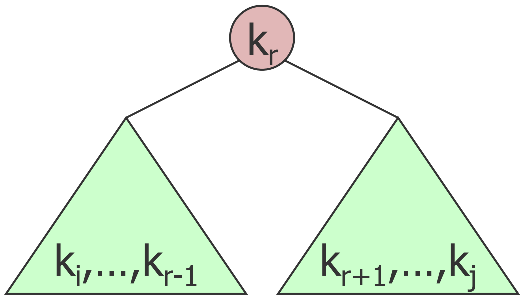
Step 2: A recursive solution

Find an optimal BST for the keys $k_i,\ldots,k_j$, where $j \geq i-1$, $i \geq 1$, $j \leq n$. (When $j=i-1$, there is no actual key, but d_{i-1})

Define $e[i,j]$ as the expected cost of searching an optimal BST containing $k_i,\ldots,k_j$. Want to find $e[1,n]$.

For dummy key d_{i-1}, $e[i,i-1]=q_{i-1}$.

For $j \geq i$, need to select a root k_r among $k_i,\ldots,k_j$.



For a subtree with keys $k_i,\ldots,k_j$, denote

$$w(i,j) = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell = w(i,r-1) + p_r + w(r+1,j)$$

$$e[i,j] = p_r+(e[i,r-1]+w(i,r-1))+(e[r+1,j]+w(r+1,j))$$
$$= e[i,r-1]+e[r+1,j]+w(i,j)$$

Thus

$$e[i,j]= \begin{cases} q_{i-1} & \text{if } j=i-1 \\ \min\limits_{i\le r\le j}\{\, e[i,\, r-1]+e[r+1,\, j]+w(i,\, j)\,\} & \text{if } i\le j \end{cases}$$

Step 3: Computing the expected search cost of an optimal BST

Use a table $w[1..n+1,0..n]$ for $w(i,j)$'s .

$w[i,i-1]=q_{i-1}$

$w[i,j]=w[i,j-1]+p_j+q_j$

```
OPTIMAL-BST(p, q, n)
let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n],
        and root[1 : n, 1 : n] be new tables
for i = 1 to n + 1                // base cases
    e[i, i − 1] = q_{i−1}         // equation (14.14)
    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             // equation (14.15)
        for r = i to j                                // try all possible roots r
            t = e[i, r − 1] + e[r + 1, j] + w[i, j]   // equation (14.14)
            if t < e[i, j]                            // new minimum?
                e[i, j] = t
                root[i, j] = r
return e and root
```