Dynamic Programming

Outline

- Introduction
- The resource allocation problem
- The traveling salesperson (TSP) problem
- Longest common subsequence problem
- 0/1 knapsack problem
- The optimal binary tree problem
- Matrix Chain-Products

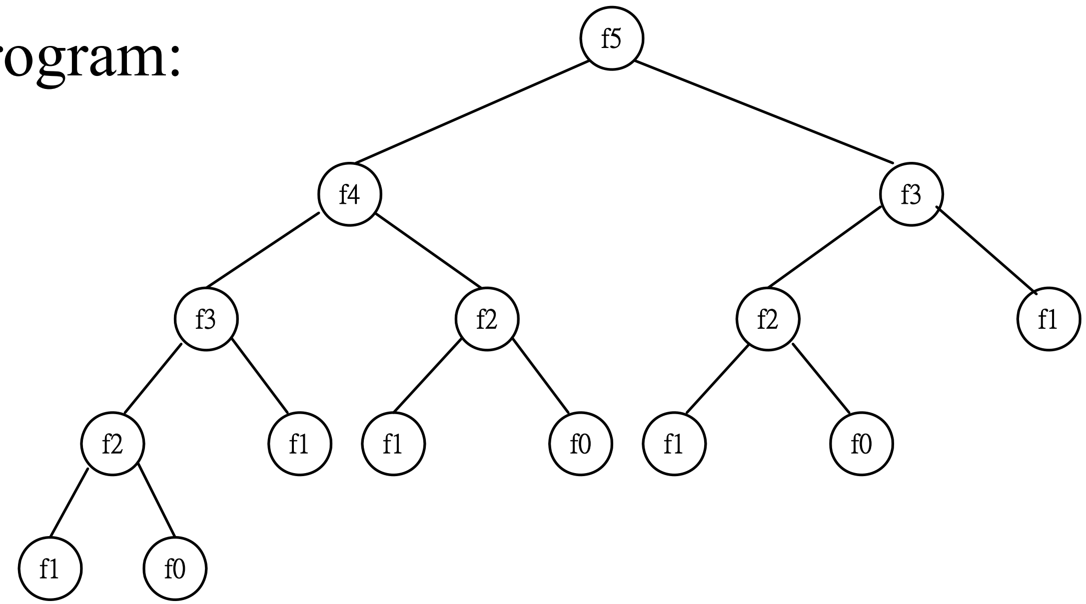
Fibonacci sequence

- **Fibonacci sequence**: 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , …

  $\mathbf{F_i = \textit{i}}$ **if** $\boldsymbol{i \le 1}$

  $\mathbf{F_i = F_{i-1} + F_{i-2}}$ **if** $\boldsymbol{i \ge 2}$

- Solved by a recursive program:



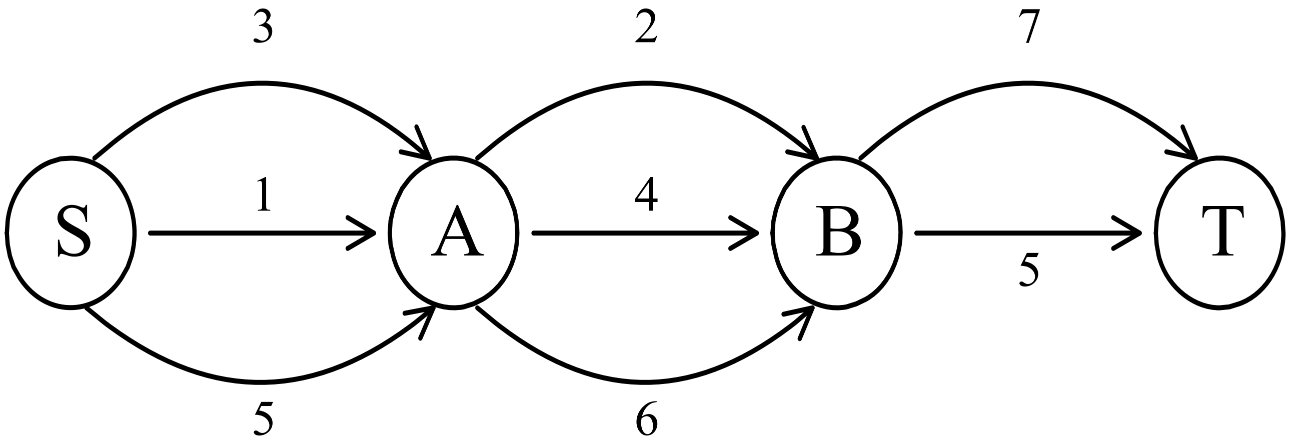
- Much replicated computation is done.
- A simple loop should solve it.

Dynamic Programming

- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions

DP using The shortest path problem

- To find the shortest path in a **multi-stage** graph



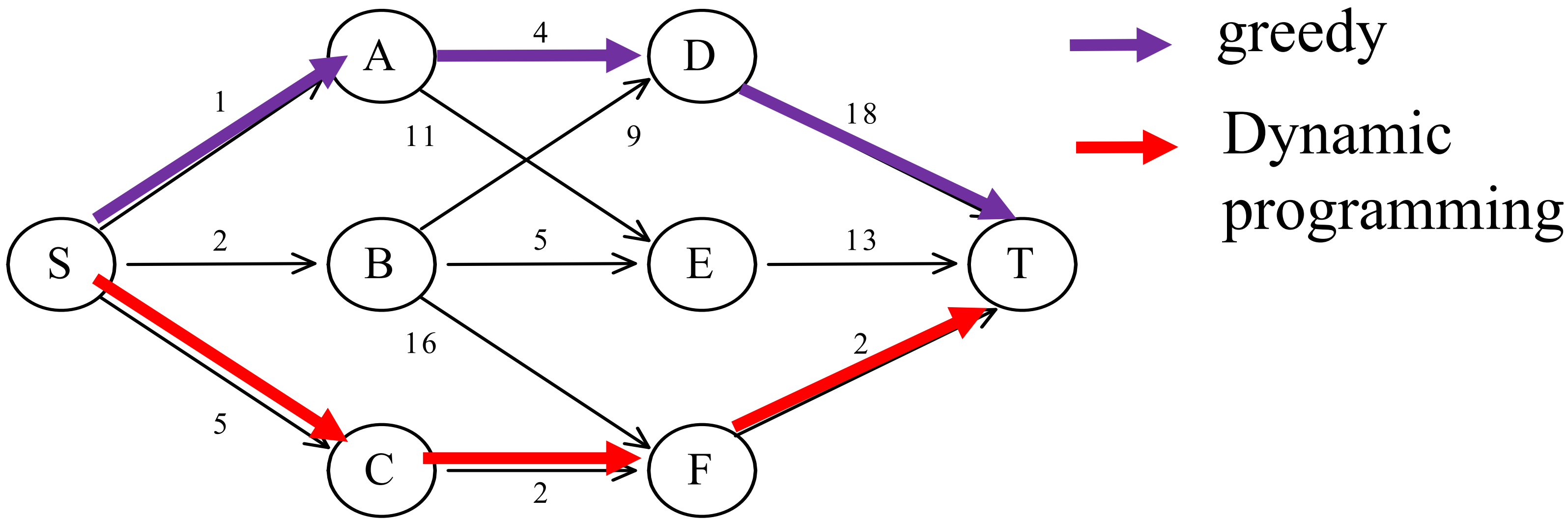
- Apply the greedy method :

The shortest path from S to T :

$1 + 2 + 5 = 8$

The shortest path in multistage graphs

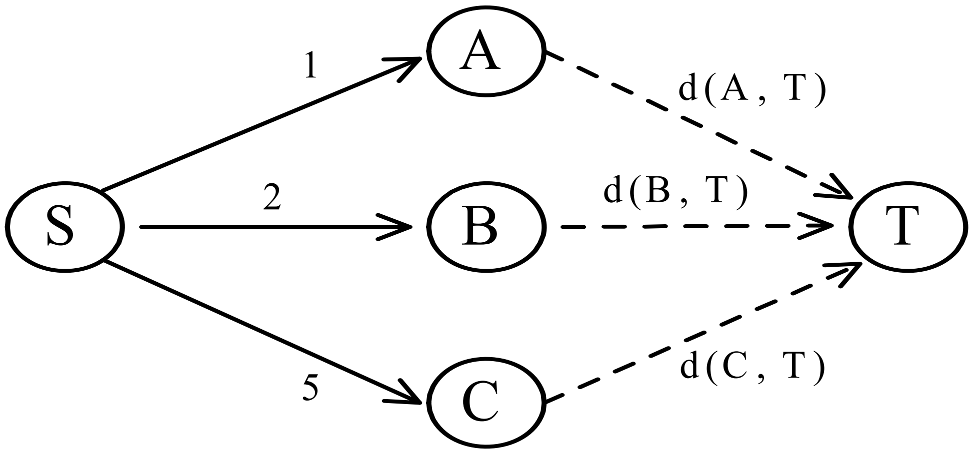
- e.g.

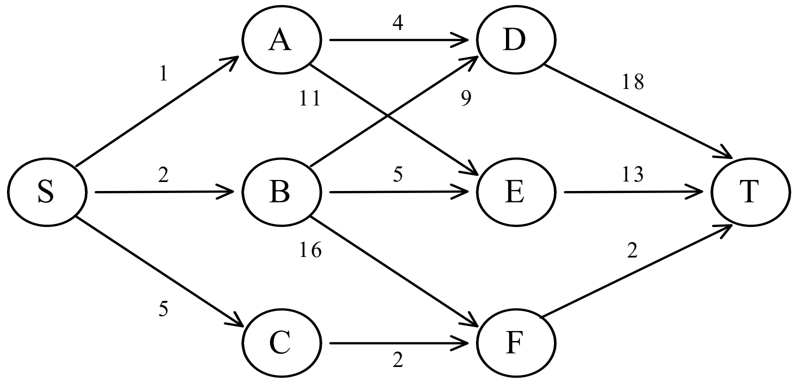


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The actual shortest path is:

  (S, C, F, T) 5+2+2 = 9.

Dynamic programming approach

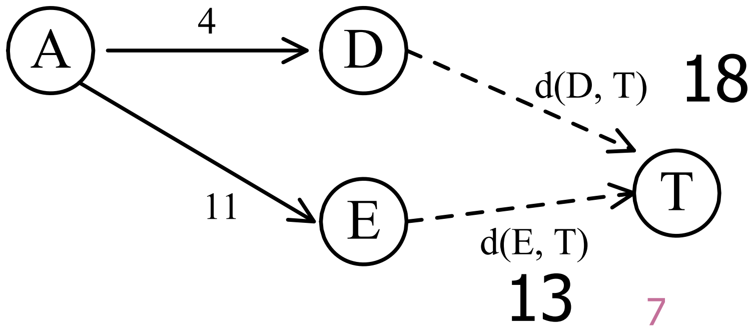




- **d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}**

Next page

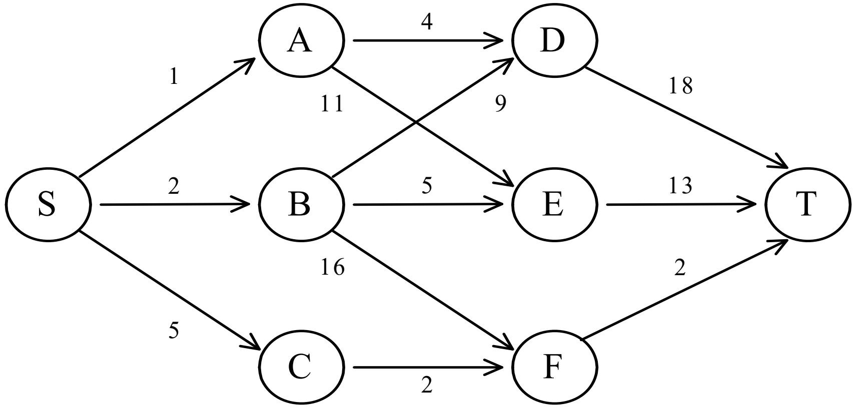
- **d(A,T)** = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.

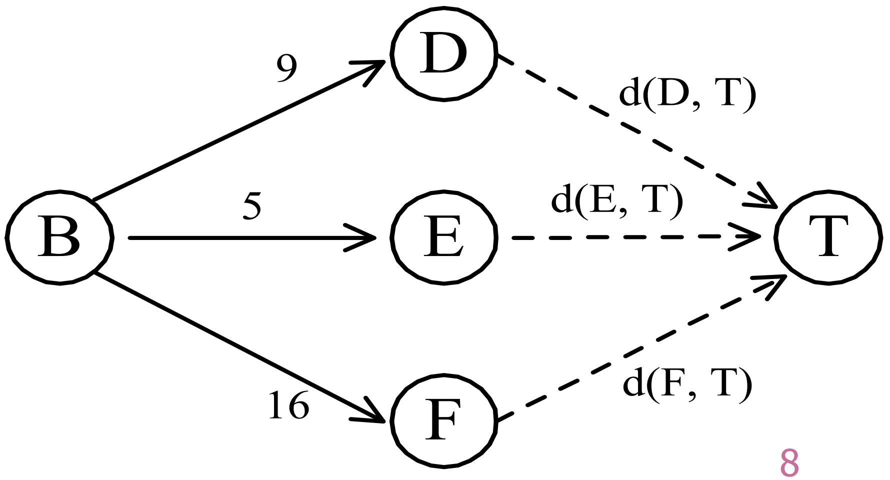


Dynamic programming

- $d(B, T) = \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\}$
  $= \min\{9+18, 5+13, 16+2\} = 18.$
- $d(C, T) = \min\{ 2+d(F, T) \} = 2+2 = 4$
- **$d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}$**
  $= \min\{1+22, 2+18, 5+4\} = 9.$

- The above way of reasoning is called backward reasoning.



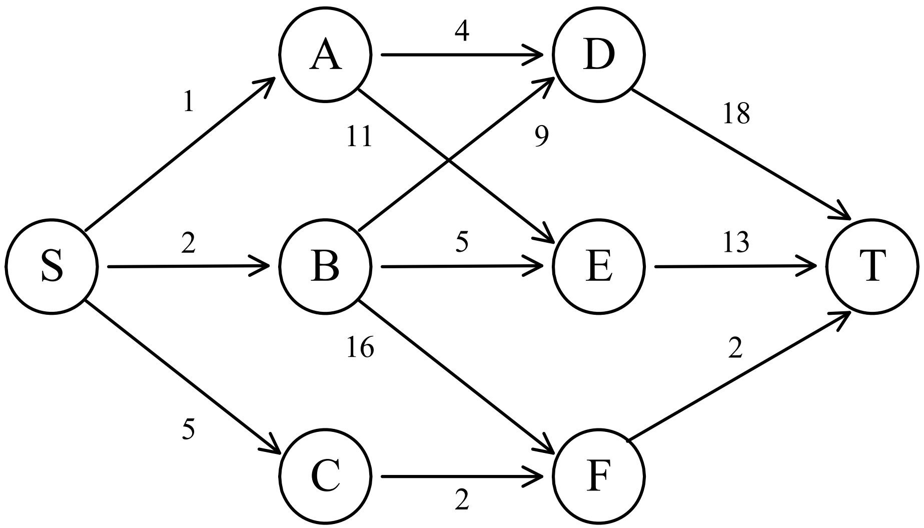


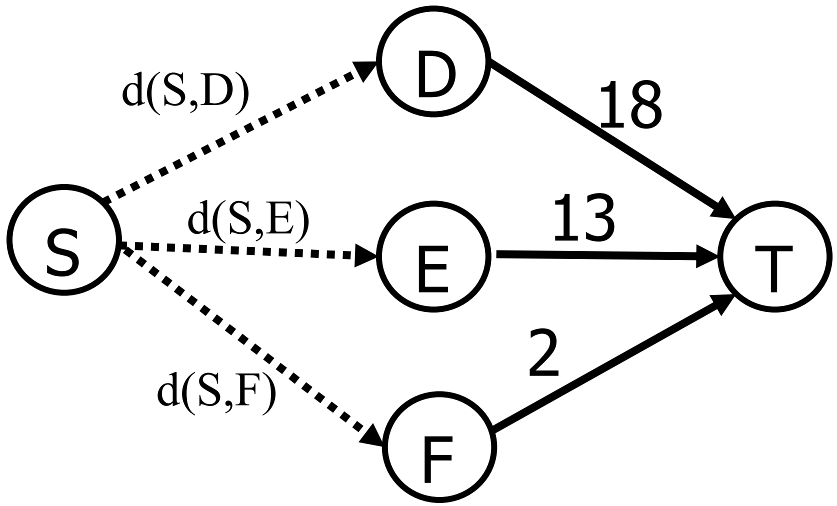
Forward reasoning

- $d(S,T) = \min\{d(S, D)+d(D, T),$
  $d(S,E)+d(E,T),$
  $d(S, F)+d(F, T)\}$

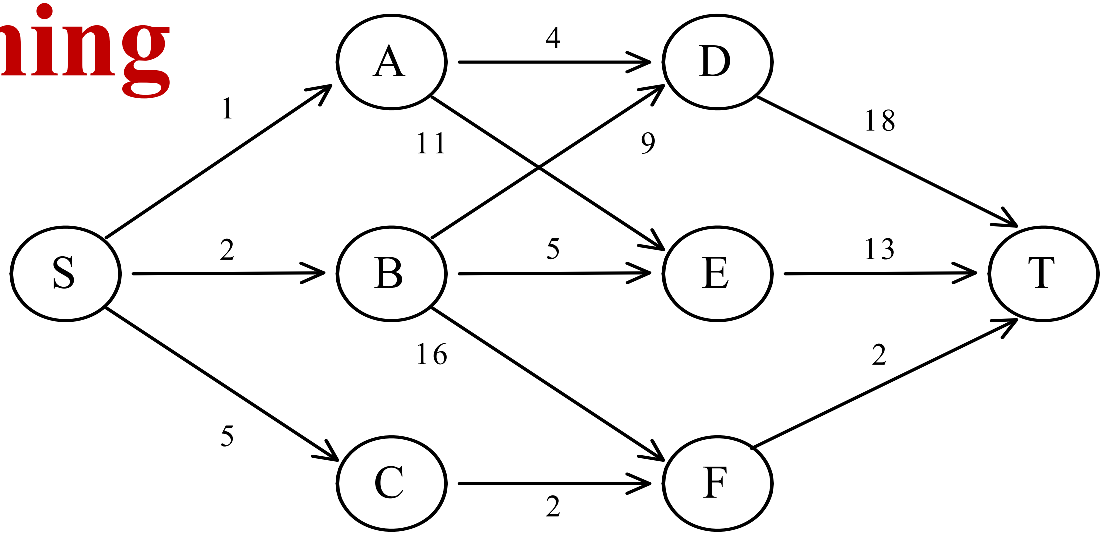
$= \min\{d(S, D)+18, d(S, E)+13, d(S, F)+2\}$

$= \min\{5+18, 7+13, 7+2\} = 9$





Forward reasoning



- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5

- d(S,D)=min{d(S, A)+d(A, D),d(S, B)+d(B, D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S, A)+d(A, E),d(S, B)+d(B, E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S, A)+d(A, F),d(S, B)+d(B, F)}
  = min{ 2+16, 5+2 } = 7

Principle of optimality

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. **If this sequence is optimal, then the last k decisions, $1 < k < n$, must be optimal.**
- e.g. the shortest path problem
  - If $\mathbf{i, i_1, i_2, \ldots, j}$ is a shortest path from i to j, then $\mathbf{i_1, i_2, \ldots, j}$ must be a shortest path from i_1 to j
- In summary, if a problem can be described by a **multistage graph**, it can be solved by dynamic programming.

Dynamic programming

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach, then the relations are solved backward. i.e., beginning with the last decision
  - On the other hand, if the relations are formulated using the backward approach, they are solved forwards.

- To solve a problem by using dynamic programming:
  - Prove the optimality
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

The resource allocation problem

The resource allocation problem

- m resources, n projects

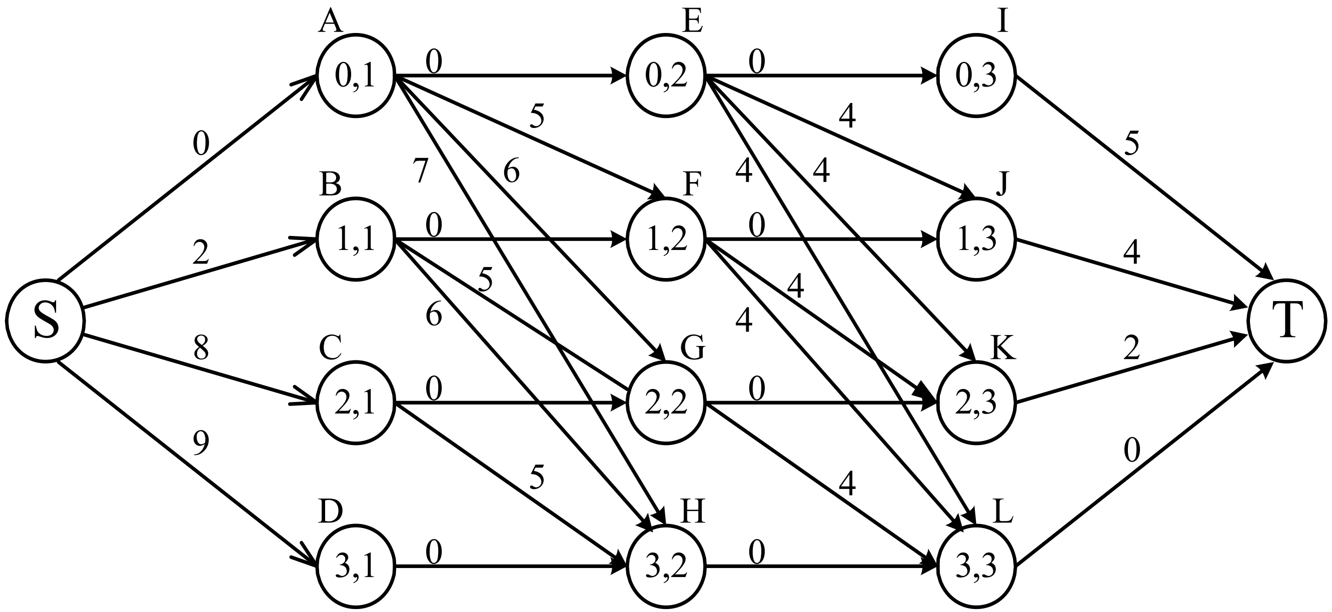
profit p(i, j) : j resources are allocated to project i. P(i, 0)=0 for each i

maximize the total profit.

Project \ Resource	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

- To make a sequence of decision to determine the number
- Resources to be allocated to project i.

The multistage graph solution



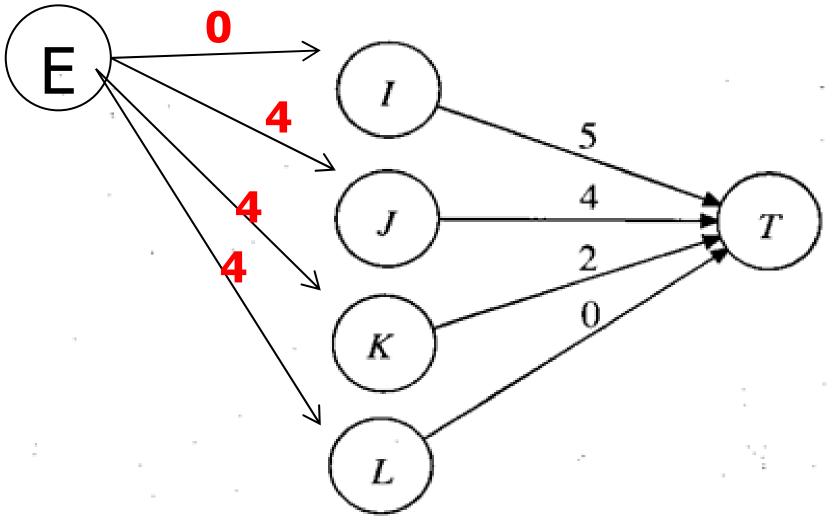
- The resource allocation problem can be described as a multistage graph.
- **(i, j) : i resources allocated to projects 1, 2, …, j**

e.g. node H=(3, 2) : 3 resources allocated to projects 1, 2.

Project \ Resource	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

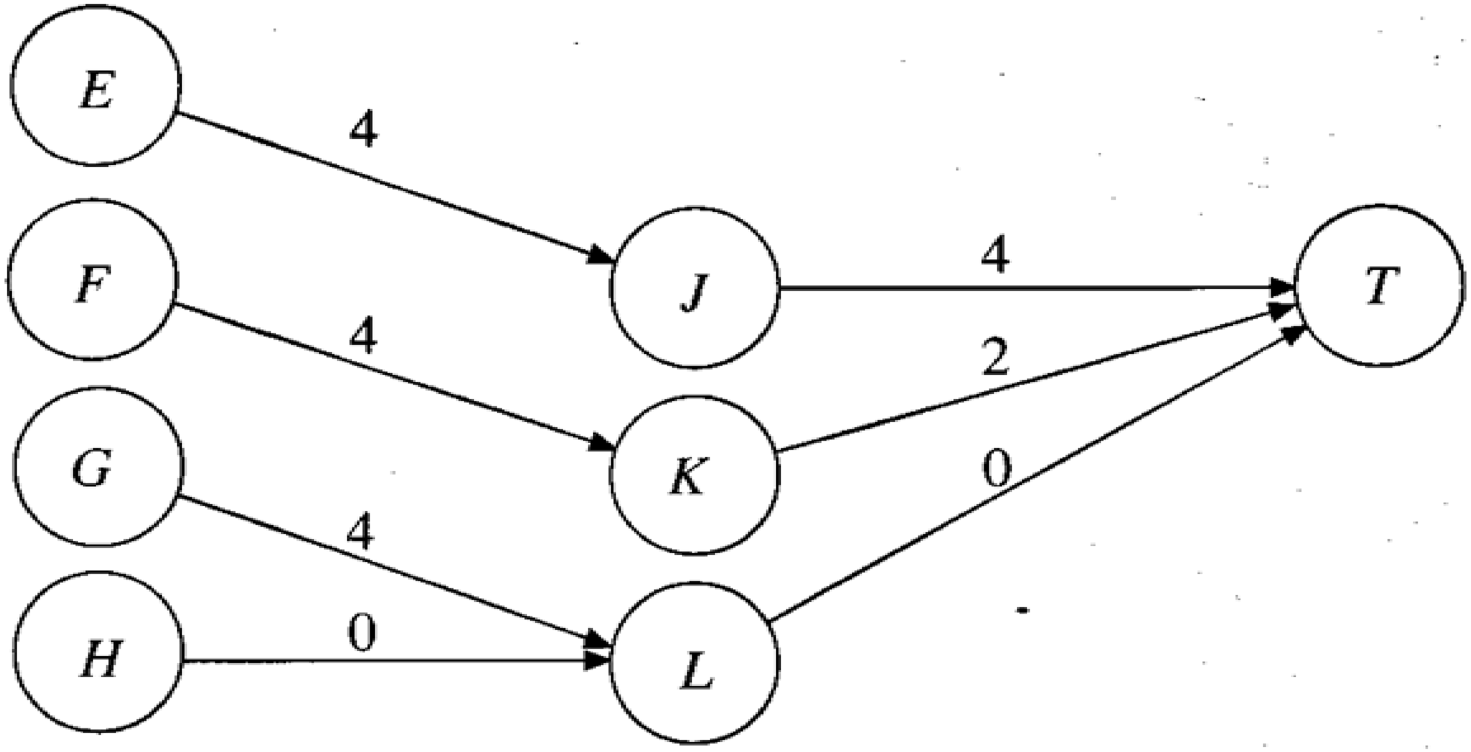
- To get the maximum profit = find the longest path from S to T .

Backward reasoning approach

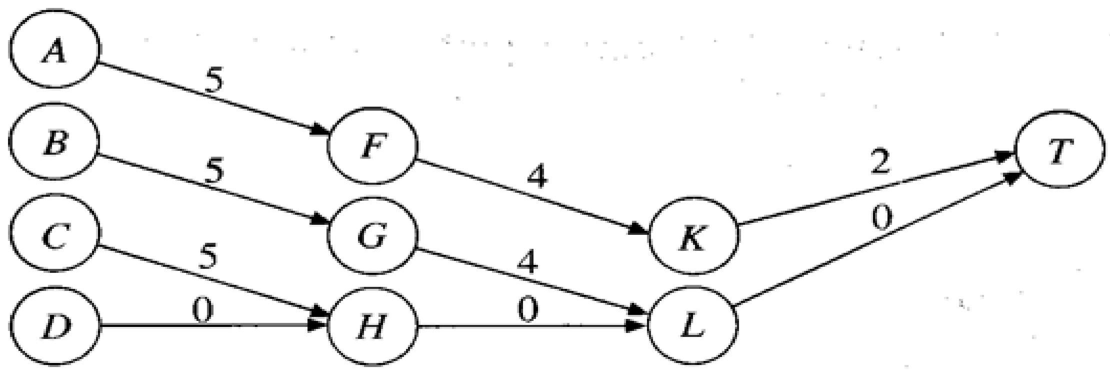


2) Having obtained the longest paths from I, J, K and L to T, we can obtain the longest paths from E, F, G and H to T easily. For instance, the longest path from E to T is determined as follows:

$$\begin{aligned} d(E, T) &= \max\{d(E, I) + d(I, T), d(E, J) + d(J, T), \\ &\qquad d(E, K) + d(K, T), d(E, L) + d(L, T)\} \\ &= \max\{0 + 5, 4 + 4, 4 + 2, 4 + 0\} \\ &= \max\{5, 8, 6, 4\} \\ &= 8. \end{aligned}$$


E
F
G
H
J
K
L
T
4
4
4
0
4
2
0

(3) The longest paths from A, B, C and D to T respectively are found by the same method and shown in Figure !.



(4) Finally, the longest path from S to T is obtained as follows:

$$\begin{aligned} d(S, T) &= \max\{d(S, A) + d(A, T), d(S, B) + d(B, T), \\ &\qquad d(S, C) + d(C, T), d(S, D) + d(D, T)\} \\ &= \max\{0 + 11, 2 + 9, 8 + 5, 9 + 0\} \\ &= \max\{11, 11, 13, 9\} \\ &= 13. \end{aligned}$$

The longest path is

$$S \to C \to H \to L \to T.$$

- Find the longest path from S to T :

(S, C, H, L, T), 8+5+0+0=13

2 resources were allocated to Project 1.

1 resource was allocated to Project 2.

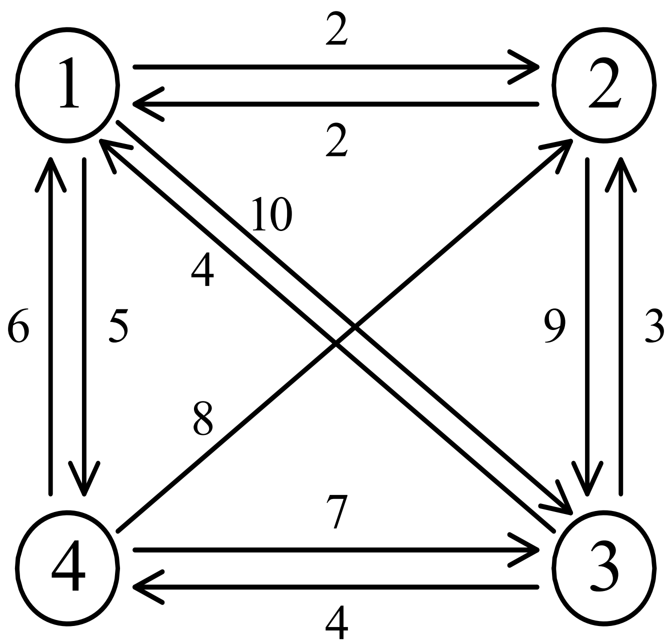
0 resources were allocated to Projects 3 and 4.

Resource Project	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

The Traveling Salesperson Problem (TSP)

The traveling salesperson problem (TSP)

- e.g. a directed graph :

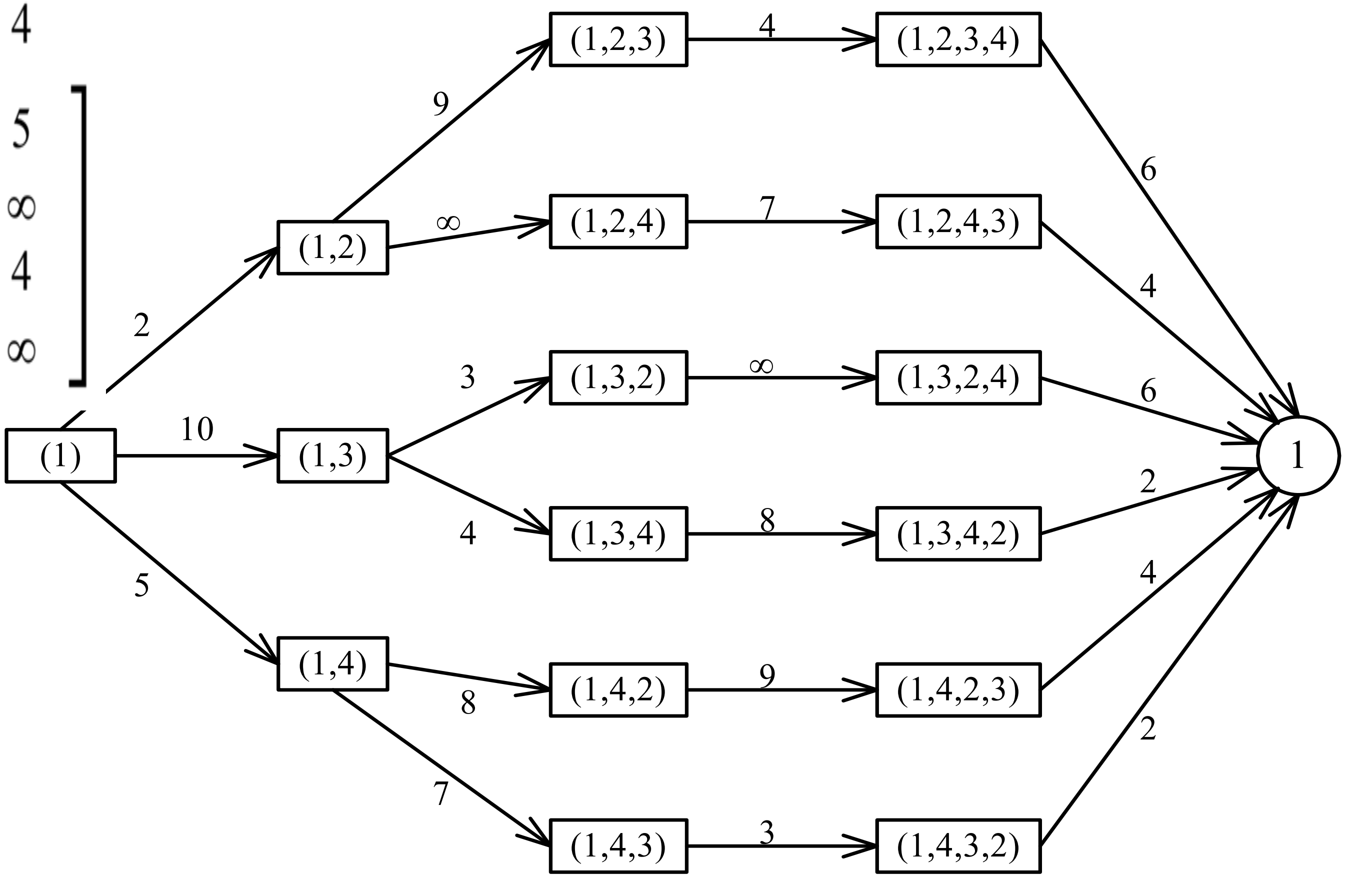


Cost matrix:

$$
\begin{array}{c}
\\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\left[\begin{array}{cccc}
\infty & 2 & 10 & 5 \\
2 & \infty & 9 & \infty \\
4 & 3 & \infty & 4 \\
6 & 8 & 7 & \infty
\end{array}\right]
\end{array}
$$

The multistage graph solution

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} \infty & 2 & 10 & 5 \\ 2 & \infty & 9 & \infty \\ 4 & 3 & \infty & 4 \\ 6 & 8 & 7 & \infty \end{bmatrix} \end{array}$$

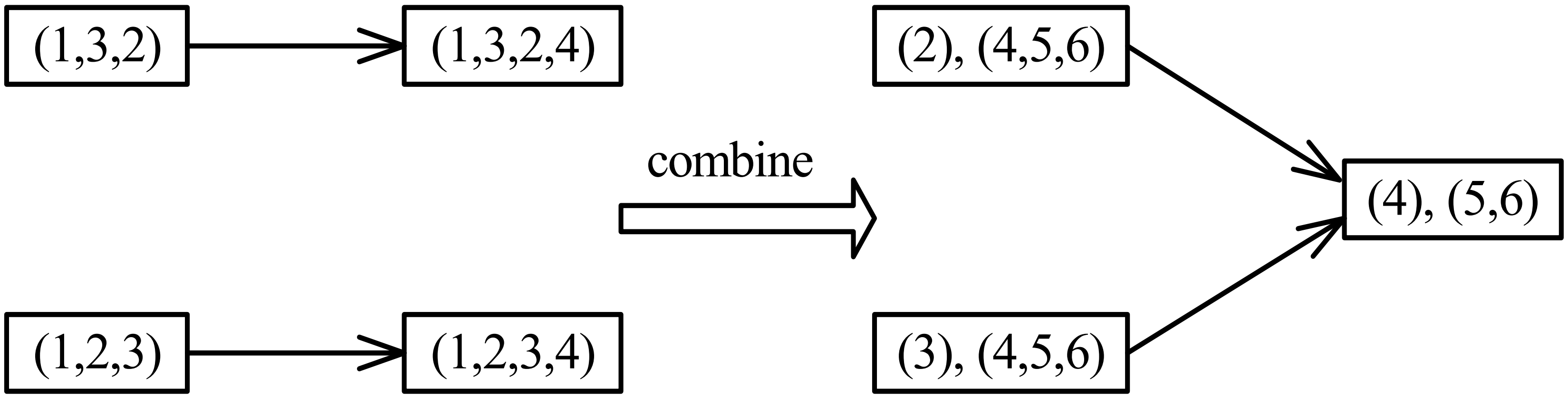


- A multistage graph can describe all possible tours of a directed graph.
- Find the shortest path:

  (1, 4, 3, 2, 1)   5+7+3+2=17

The representation of a node

- Suppose that we have six vertices in the graph.
- We can combine {1, 2, 3, 4} and {1, 3, 2, 4} into one node.



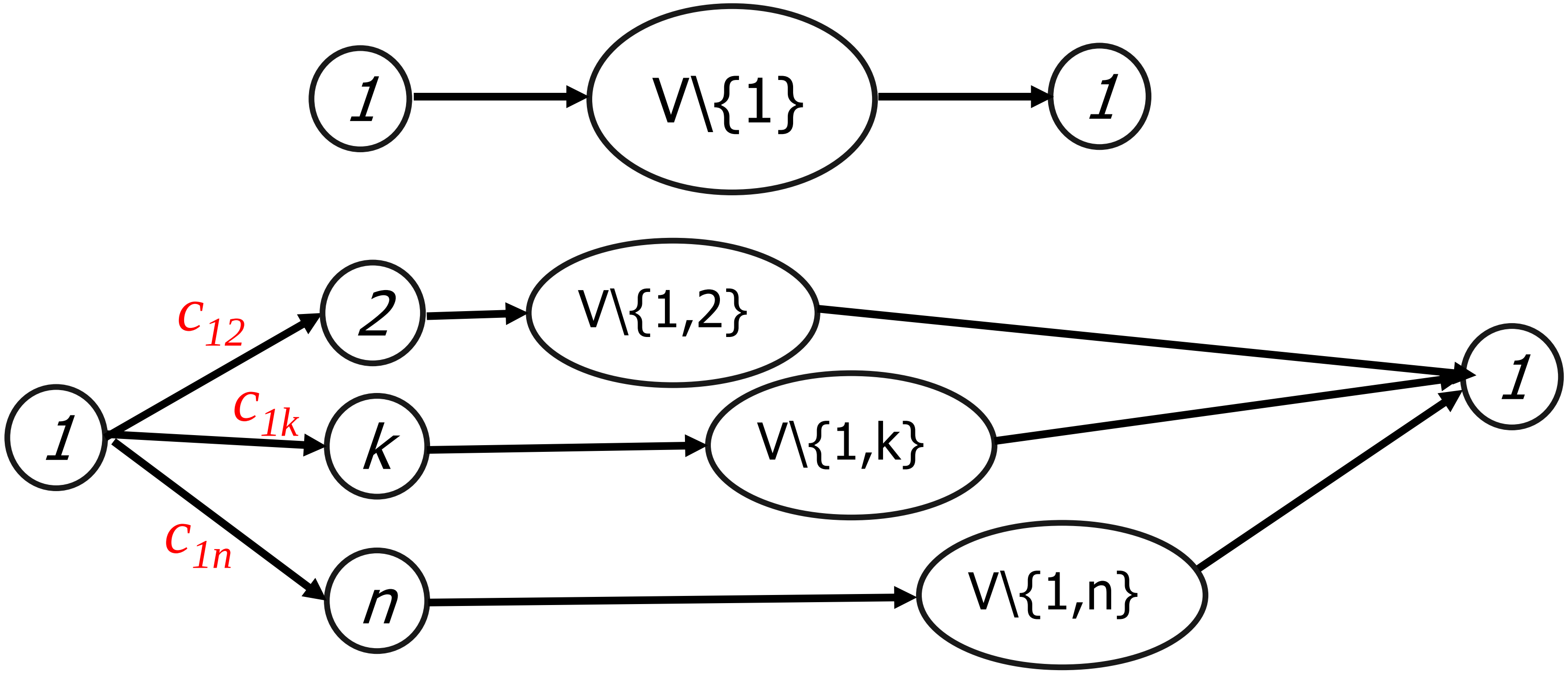
(a) (b)

- (3),(4, 5, 6) means that the last vertex visited is 3 and the remaining vertices **to be visited are (4, 5, 6).**

The dynamic programming approach

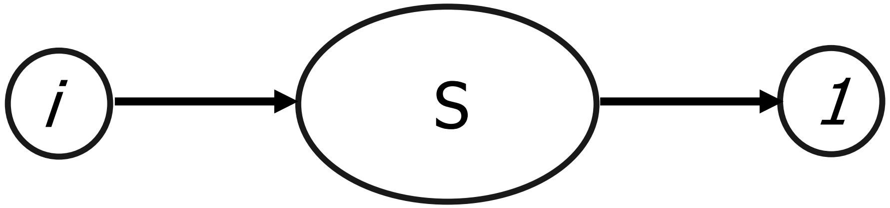
- Let **g(i, S)** be the length of the shortest path starting at vertex 1, going through all vertices in S, and terminating at vertex 1.
- The length of an optimal tour :

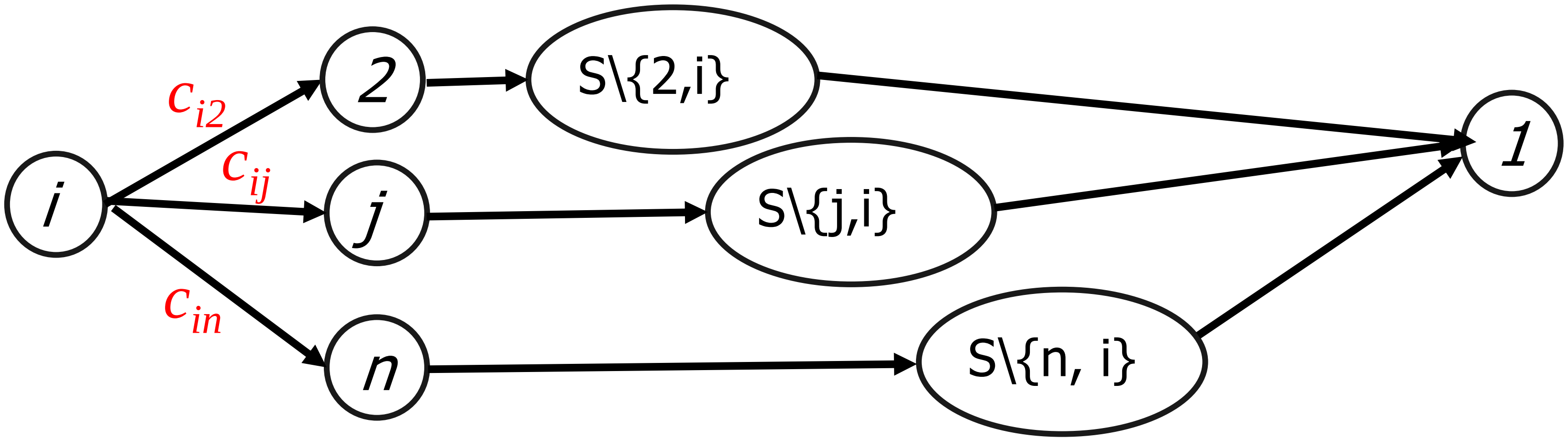
$$g(1, V\backslash\{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V\backslash\{1, k\})\}$$



- The general form:

$$g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S \backslash \{j, i\})\}$$





- Time complexity:

$$n + \sum_{k=2}^{n} (n-1)\binom{n-2}{n-k}(n-k)$$

$$= O(n^2 2^n)$$

( ), ( )
↑ ↑
(n-1) (n-k)

The longest common subsequence (LCS) problem

The longest common subsequence (LCS) problem

- A string : A = b a c a d
- A subsequence of A: deleting 0 or more symbols from A (not necessarily consecutive).

  e.g. ad, ac, bac, acad, bacad, bcd.
- Common subsequences of A = b a c a d and

  B = a c c b a d c b : ad, ac, bac, acad.
- The longest common subsequence (LCS) of A and B: a c a d.

Determine the length of the LCS

- Instead of finding the longest common subsequence, let us try to determine the length of the LCS.
- Then, tracking back to find the LCS.
- Consider $a_1a_2\ldots a_m$ and $b_1b_2\ldots b_n$.
- **Case 1: $a_m = b_n$.** The LCS must contain a_m; we must find the LCS of $a_1a_2\ldots a_{m-1}$ and $b_1b_2\ldots b_{n-1}$.
- **Case 2: $a_m \neq b_n$.** We have to find the LCS of $a_1a_2\ldots a_{m-1}$ and $b_1b_2\ldots b_n$, and $a_1a_2\ldots a_m$ and $b_1b_2\ldots b_{n-1}$

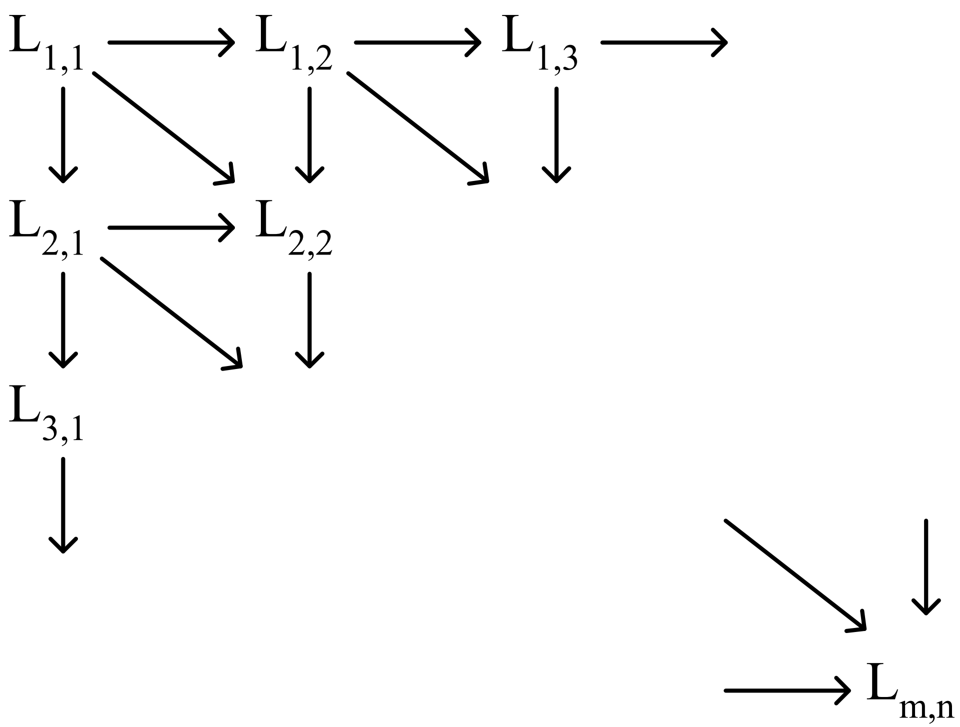
The LCS algorithm

- Let $A = a_1\ a_2 \ldots a_m$ and $B = b_1\ b_2 \ldots b_n$
- Let $L_{i,j}$ denote the length of the longest common subsequence of $a_1\ a_2 \ldots a_i$ and $b_1\ b_2 \ldots b_{j.}$
- $$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\{ L_{i-1,j}, L_{i,j-1} \} & \text{if } a_i \neq b_j \end{cases}$$

  $L_{0,0} = L_{0,j} = L_{i,0} = 0$ for $1 \leq i \leq m$, $1 \leq j \leq n$.

Solving approach: Find $L_{1,1}$

- The dynamic programming approach for solving the LCS problem:



- Time complexity: O(mn)

Tracing back in the LCS algorithm

- e.g. A = b a c a d, B = a c c b a d c b

B

			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	1	1	1	2	2	2	2
A	c	0	1	2	2	2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	4	4

- After all $L_{i,j}$'s have been found, we can trace back to find the longest common subsequence of A and B.

0/1 knapsack problem

0/1 knapsack problem

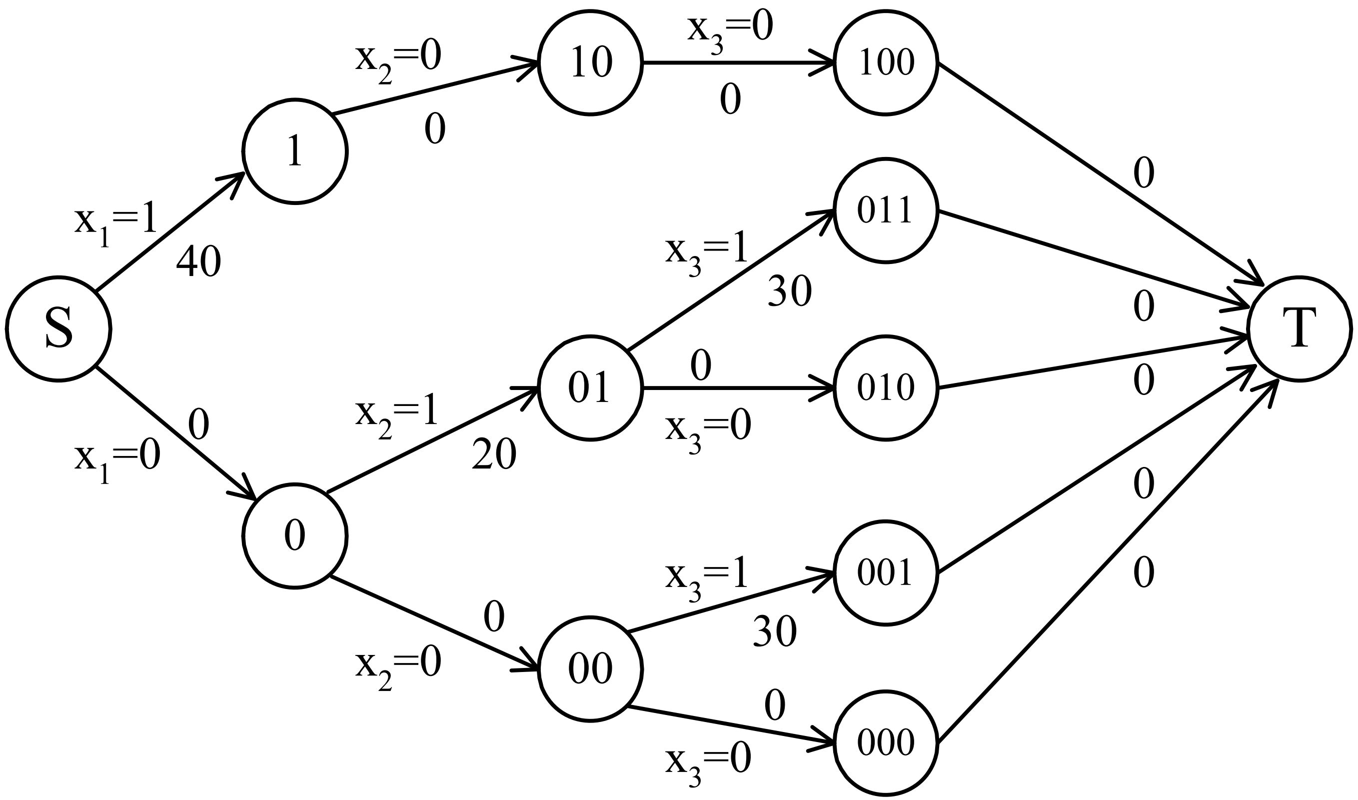
- n objects, weight $W_1, W_2, \ldots, W_n$
  profit $P_1, P_2, \ldots, P_n$
  capacity M
  maximize $\sum_{1 \le i \le n} P_i x_i$
  subject to $\sum_{1 \le i \le n} W_i x_i \le M$
  $x_i = 0$ or 1, $1 \le i \le n$
- e. g.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

M=10

The multistage graph solution

- The 0/1 knapsack problem can be described by a multistage graph.



The dynamic programming approach

- The longest path represents the optimal solution:

  $x_1=0,\ x_2=1,\ x_3=1$

$$\sum P_i x_i = 20+30 = 50$$

- Let $\mathbf{f_i(Q)}$ be the value of an optimal solution to objects 1, 2, 3,…, i with capacity Q.
- $f_i(Q) = \max\{\ f_{i-1}(Q),\ f_{i-1}(Q-W_i)+P_i\ \}$

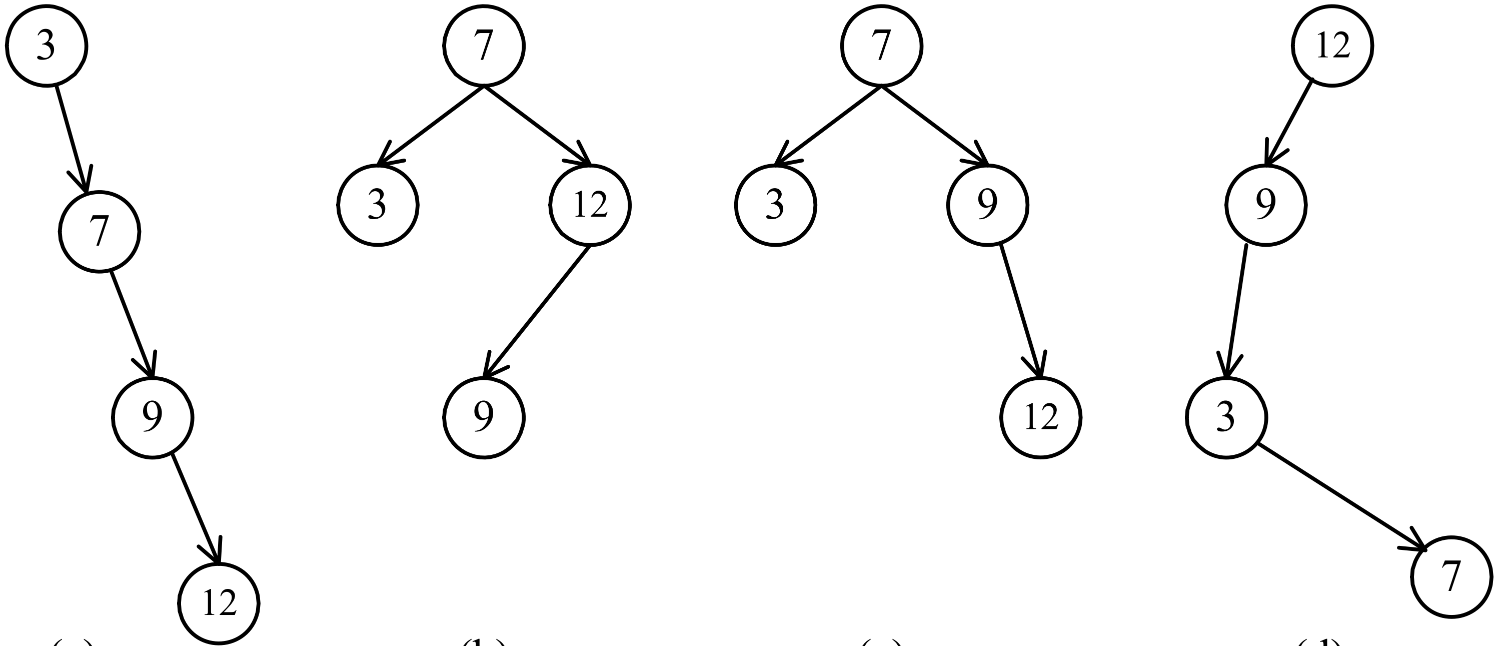
  $= \max\{$ 第i個不選獲利，第i個必選獲利$\}$
- The optimal solution is $f_n(M)$.

  $f_n(M) = \max\{f_{i-1}(M),\ f_{i-1}(M-W_i) + P_i\ \}$

Optimal binary search trees

Optimal binary search trees

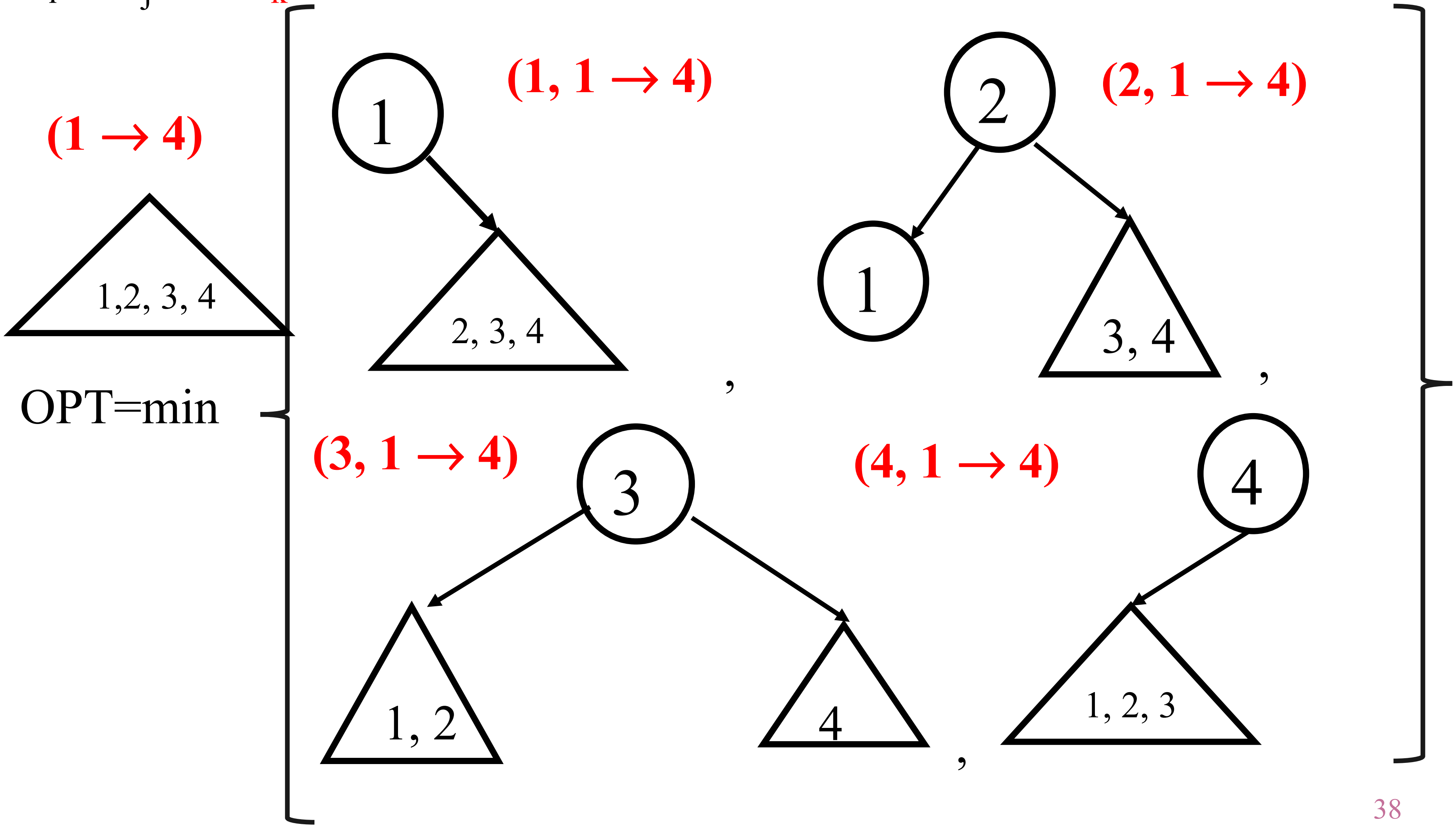
- e.g. binary search trees for 3, 7, 9, 12;

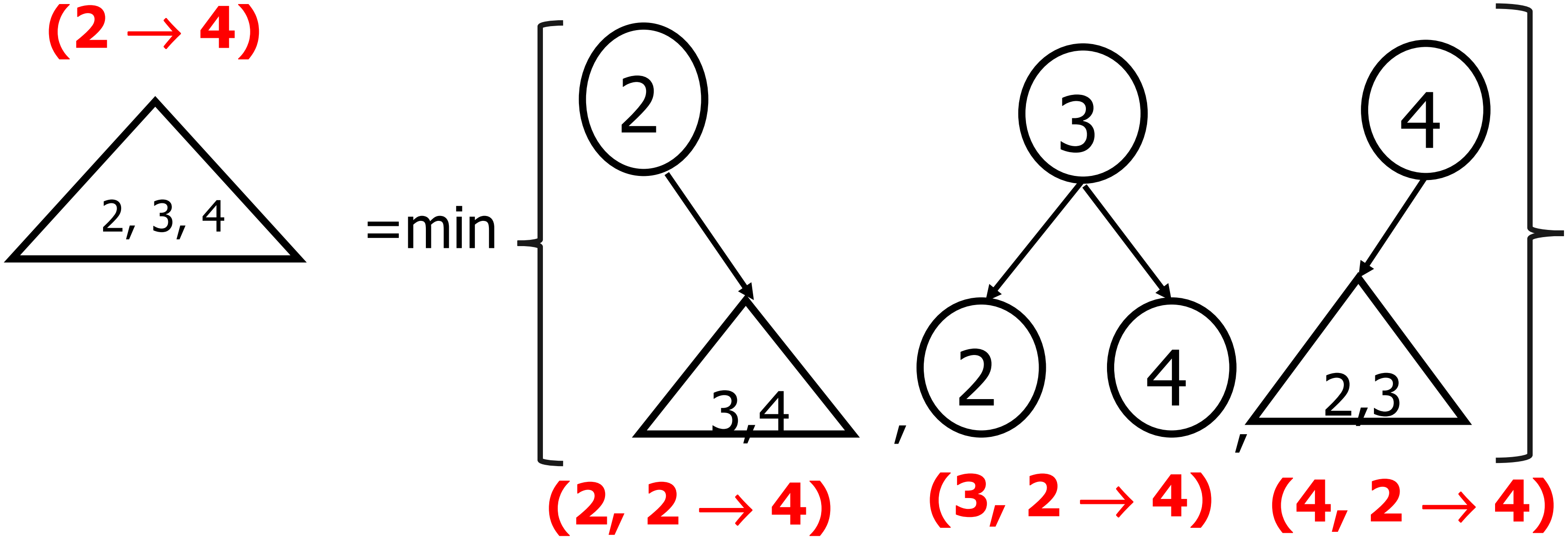


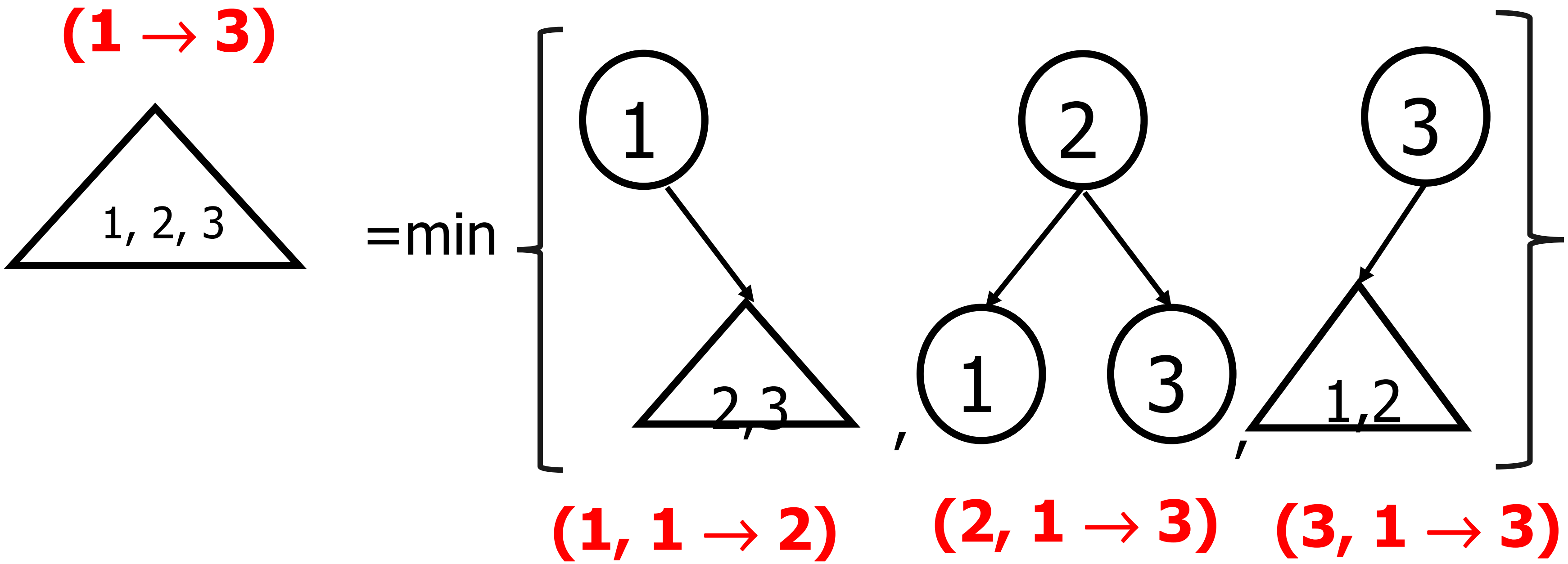
(a) (b) (c) (d)

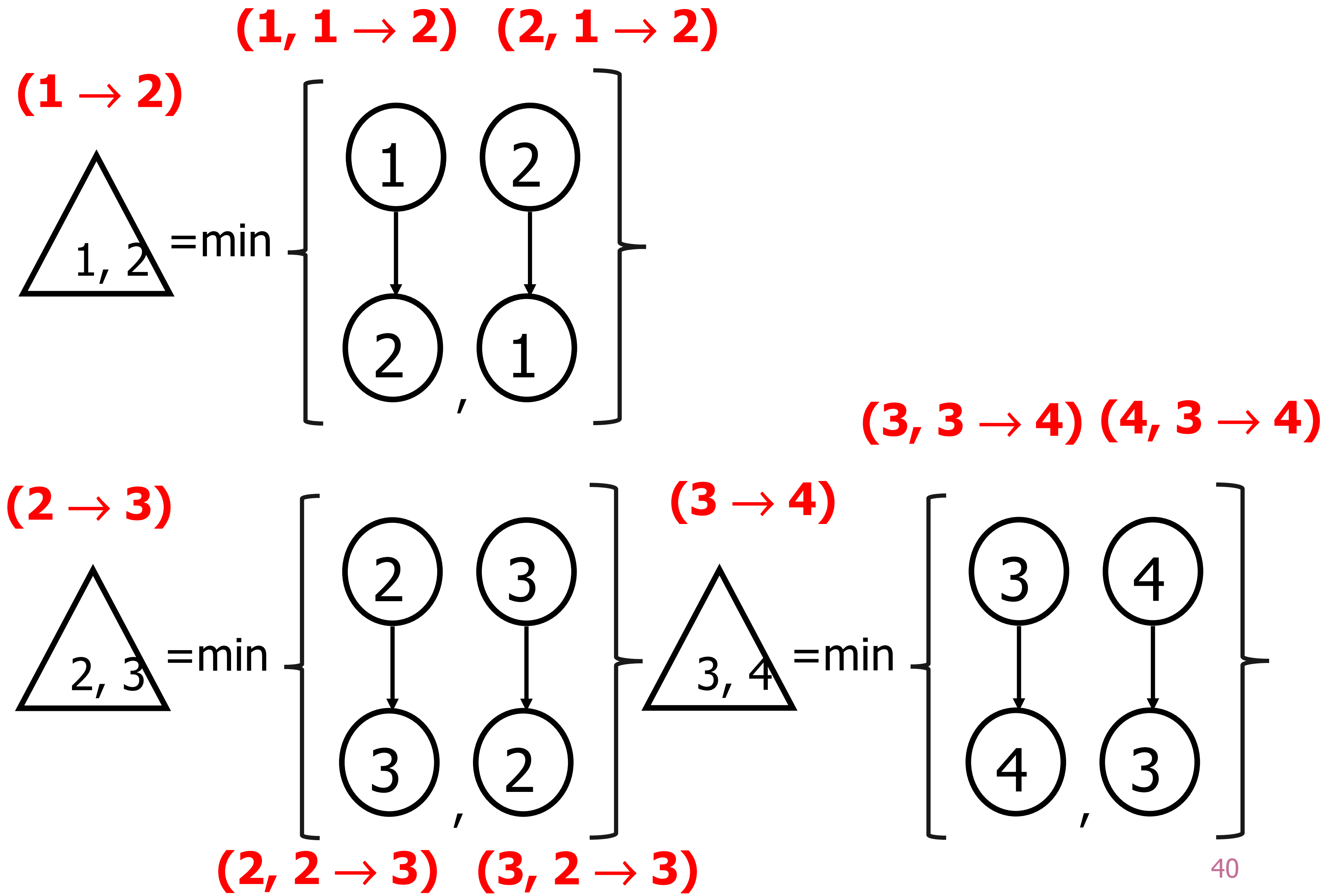
$(a_i \rightarrow a_j)$ denote the optimal binary tree containing identifiers a_i to a_j.

$(a_k, a_i \rightarrow a_j)$ denote an optimal binary tree containing the identifier a_i to a_j and a_k as its root.




(2 → 4)
2, 3, 4
=min
2
3,4
3
2
4
4
2,3
(2, 2 → 4)
(3, 2 → 4)
(4, 2 → 4)


(1 → 3)
1, 2, 3
=min
1
2,3
2
1
3
3
1,2
(1, 1 → 2)
(2, 1 → 3)
(3, 1 → 3)


(1 → 2)
(1, 1 → 2)
(2, 1 → 2)
1, 2
=min
1
2
2
1
(2 → 3)
2, 3
=min
2
3
3
2
(2, 2 → 3)
(3, 2 → 3)
(3 → 4)
3, 4
=min
(3, 3 → 4)
(4, 3 → 4)
3
4
4
3

Optimal binary tree

- Identifiers stored close to the root of the tree can be searched rather quickly.
- For each identifier a_i, associated with probability P_i.
- For each identifier not stored in tree also given probability Q_i.

Optimal binary search trees

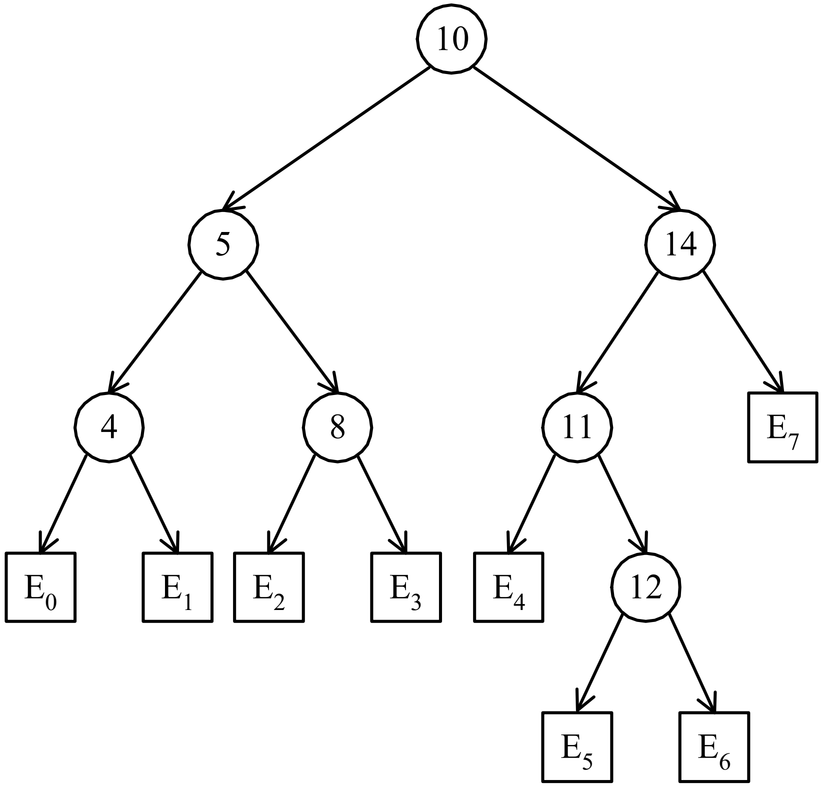
- n identifiers : $a_1 < a_2 < a_3 < \ldots < a_n$

P_i, $1 \leq i \leq n$: the probability that a_i is searched.

Q_i, $0 \leq i \leq n$: the probability that x is searched

where $a_i < x < a_{i+1}$ ($a_0 = -\infty$, $a_{n+1} = \infty$).

$$\sum_{i=1}^{n} P_i + \sum_{i=1}^{n} Q_i = 1$$

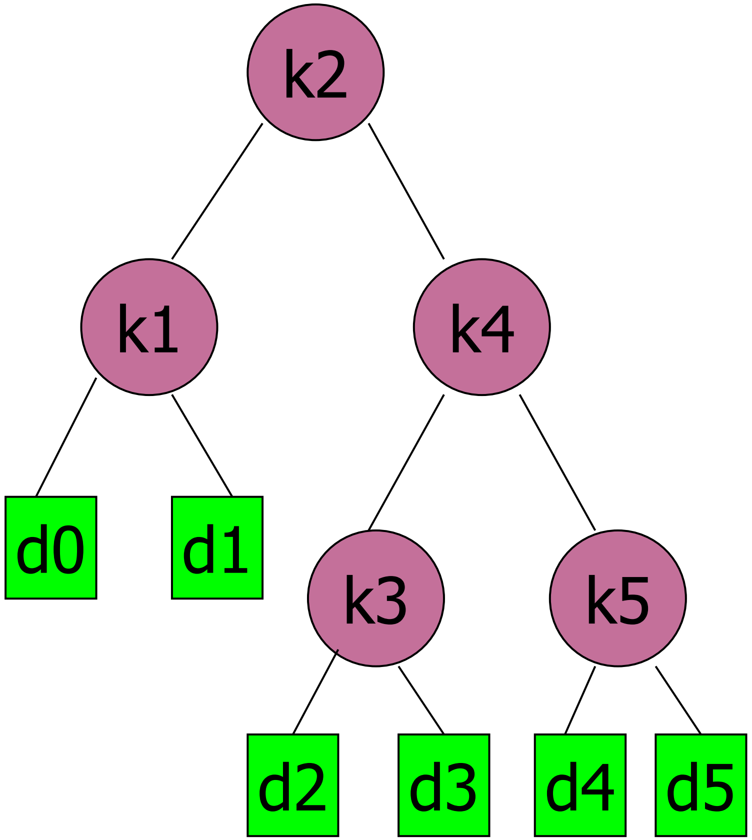


- Identifiers : 4, 5, 8, 10, 11, 12, 14
- Internal node : successful search, P_i
- External node : unsuccessful search, Q_i

- The expected cost of a binary tree:

$$\sum_{n=1}^{n} P_i * \text{level}(a_i) + \sum_{n=0}^{n} Q_i * (\text{level}(E_i) - 1)$$

- The level of the root : 1
- The optimal binary tree is a tree with minimal cost.

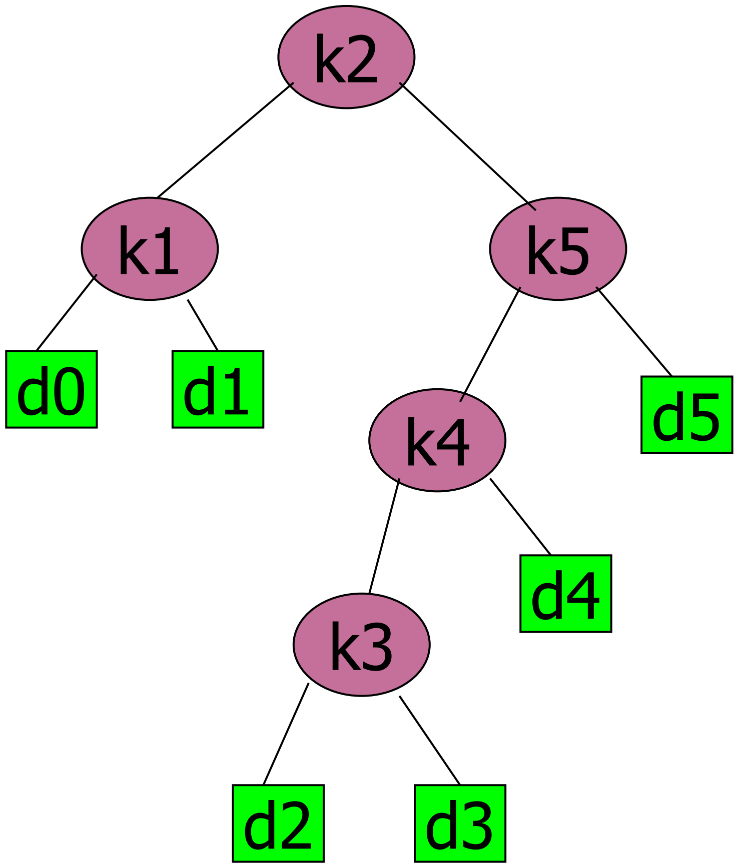


Node	level	probability	cost
k1	2	0.15	0.30
k2	1	0.10	0.10
k3	3	0.05	0.15
k4	2	0.10	0.20
K5	3	0.20	0.60
d0	3	0.05	0.10
d1	3	0.10	0.20
d2	4	0.05	0.15
d3	4	0.05	0.15
d4	4	0.05	0.15
d5	4	0.10	0.30

$$\sum_{n=1}^{n} P_i * \text{level}(a_i) + \sum_{n=0}^{n} Q_i * (\text{level}(E_i) - 1)$$

i	0	1	2	3	4	5
P_i		0.15	0.10	0.05	0.10	0.20
Q_i	0.05	0.10	0.05	0.05	0.05	0.10

Total cost=2.4



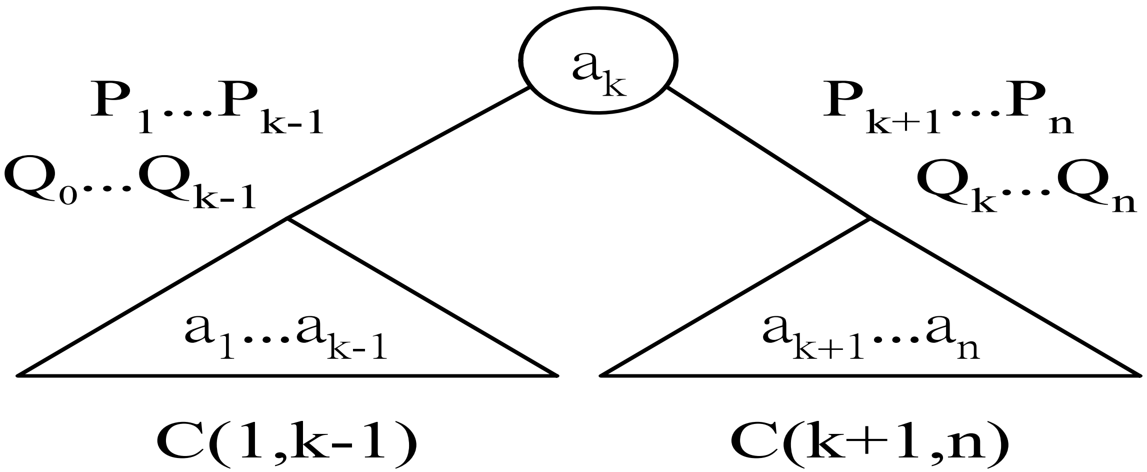
Node	level	probability	cost
k1	2	0.15	0.30
k2	1	0.10	0.10
k3	4	0.05	0.20
k4	3	0.10	0.30
K5	2	0.20	0.40
d0	3	0.05	0.10
d1	3	0.10	0.20
d2	5	0.05	0.20
d3	5	0.05	0.20
d4	4	0.05	0.15
d5	3	0.10	0.20

i	0	1	2	3	4	5
P_i		0.15	0.10	0.05	0.10	0.20
Q_i	0.05	0.10	0.05	0.05	0.05	0.10

Total cost=2.35

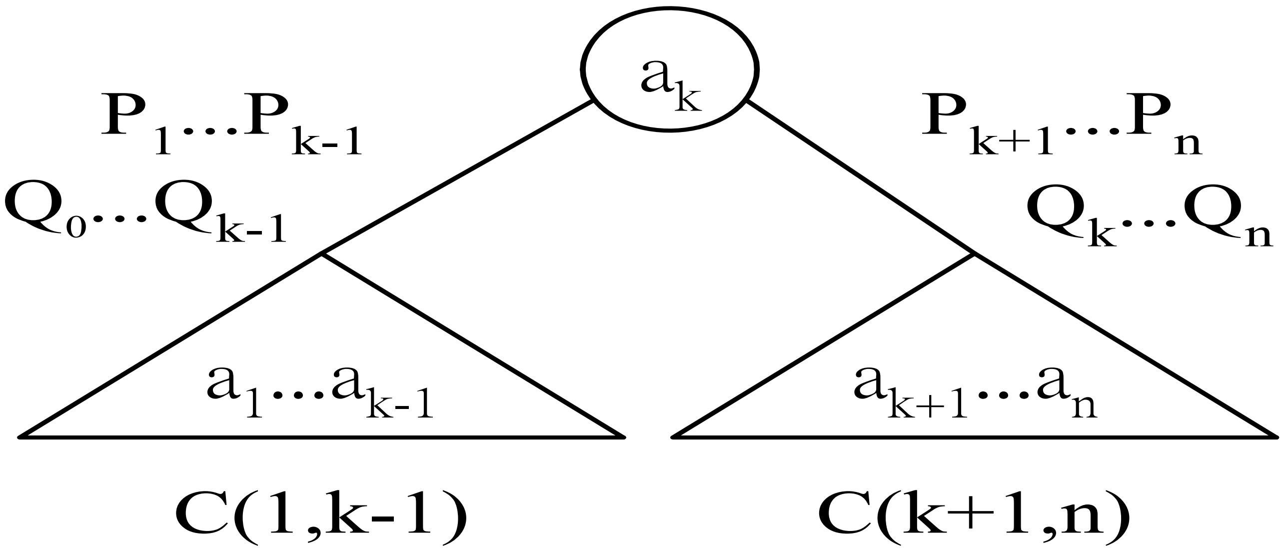
The dynamic programming approach

- Select an identifier, a_k, to be the root of the tree, all identifier < a_k (>a_k) will constitute the left (right) descendant.
- Let C(i, j) denote the cost of an optimal binary search tree containing $a_i,\ldots,a_j$.
- The cost of the optimal binary search tree with a_k as its root :
- $C(1,n) = \min_{1\le k\le n}\left\{\begin{array}{l} P_k + \left[Q_0 + \sum_{i=1}^{k-1}(P_i + Q_i) + C(1,k-1)\right] \\ +\left[Q_k + \sum_{i=k+1}^{n}(P_i + Q_i) + C(k+1,n)\right] \end{array}\right\}$



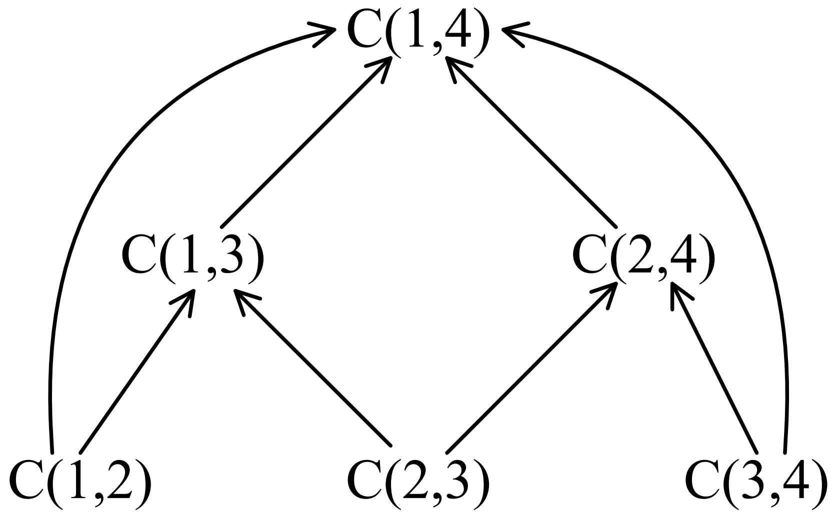
General formula

$$
\begin{aligned}
C(i, j) &= \min_{i \le k \le j} \left\{ P_k + \left[Q_{i\text{-}1} + \sum_{m=i}^{k-1} (P_m + Q_m) + C(i, k-1) \right] \right. \\
&\qquad \left. + \left[Q_k + \sum_{m=k+1}^{j} (P_m + Q_m) + C(k+1, j) \right] \right\} \\
&= \min_{i \le k \le j} \left\{ C(i, k-1) + C(k+1, j) + Q_{i\text{-}1} + \sum_{m=i}^{j} (P_m + Q_m) \right\}
\end{aligned}
$$



Computation relationships of subtrees

- e.g. n=4



- Time complexity : $O(n^3)$

  when j-i=m, there are (n-m) C(i, j)'s to compute.

  Each C(i, j) with j-i=m can be computed in O(m) time.

$$O\left(\sum_{1 \le m \le n} m(n-m)\right) = O(n^3)$$

Find the optimal binary search tree for N = 6, having keys $k_1 \ldots k_6$ and weights $p_1 = 10$, $p_2 = 3$, $p_3 = 9$, $p_4 = 2$, $p_5 = 0$, $p_6 = 10$; $q_0 = 5$, $q_1 = 6$, $q_2 = 4$, $q_3 = 4$, $q_4 = 3$, $q_5 = 8$, $q_6 = 0$. The following figure shows the arrays as they would appear after the initialization and their final disposition.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
k		3	7	10	15	20	25
p	-	10	3	9	2	0	10
q	5	6	4	4	3	8	0

R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							

W	0	1	2	3	4	5	6
0	5	21	28	41	46	54	64
1		6	13	26	31	39	49
2			4	17	22	30	40
3				4	9	17	27
4					3	11	21
5						8	18
6							0

C	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							

- To help our discussion, we define
- C_{ij} =expected time searching keys in (k_i, k_{i+1}, …, k_j; d_{i-1}, d_i, …, d_j)
- w_{ij}= sum of the probabilities of keys in in (k_i, k_{i+1}, …, k_j; d_{i-1}, d_i, …, d_j)

$$w_{i,j} = \sum_{s=i+1}^{j} P_s + \sum_{t=i}^{j} Q_t$$

$$C_{i,j} = \min_{i<k\leq j}\{C_{1,k-1} + C_{k,j}\} + w_{i,j}$$

$$C_{i,i} = w_{i,i}$$

Notations for example

- OBST(i, j) denotes the optimal binary search tree containing the keys ki, ki+1, …, kj;
- $W_{i,j}$ denotes the weight matrix for OBST(i, j)
- $W_{i,j}$ can be defined using the following formula:

$$W_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

- $C_{i,j}$, $0 \le i \le j \le n$ denotes the cost matrix for OBST(i, j)
- $C_{i,j}$ can be defined recursively, in the following manner:

  $C_{i,i} = W_{i,j}$

  $C_{i,j} = W_{i,j} + \min_{i<k\le j}(C_{i,k-1} + C_{k,j})$
- $R_{i,j}$, $0 \le i \le j \le n$ denotes the root matrix for OBST(i, j)

The values of the weight matrix have been computed according to the formulas previously stated, as follows:

W (0, 0) = q0 = 5
W (1, 1) = q1 = 6
W (2, 2) = q2 = 4
W (3, 3) = q3 = 4
W (4, 4) = q4 = 3
W (5, 5) = q5 = 8
W (6, 6) = q6 = 0

W (0, 1) = q0 + q1 + p1 = 5 + 6 + 10 = 21
W (0, 2) = W (0, 1) + q2 + p2 = 21 + 4 + 3 = 28
W (0, 3) = W (0, 2) + q3 + p3 = 28 + 4 + 9 = 41
W (0, 4) = W (0, 3) + q4 + p4 = 41 + 3 + 2 = 46
W (0, 5) = W (0, 4) + q5 + p5 = 46 + 8 + 0 = 54
W (0, 6) = W (0, 5) + q6 + p6 = 54 + 0 + 10 = 64
W (1, 2) = W (1, 1) + q2 + p2 = 6 + 4 + 3 = 13

--- and so on ---
until we reach:

W (5, 6) = q5 + q6 + p6 = 18

C (0, 0) = W (0, 0) = 5
C (1, 1) = W (1, 1) = 6
C (2, 2) = W (2, 2) = 4
C (3, 3) = W (3, 3) = 4
C (4, 4) = W (4, 4) = 3
C (5, 5) = W (5, 5) = 8
C (6, 6) = W (6, 6) = 0

C	0	1	2	3	4	5	6
0	5						
1		6					
2			4				
3				4			
4					3		
5						8	
6							0

Figure 5: Cost Matrix after first step

C (0, 1) = W (0, 1) + (C (0, 0) + C (**1**, 1)) = 21 + 5 + 6 = 32

C (1, 2) = W (0, 1) + (C (1, 1) + C (**2**, 2)) = 13 + 6 + 4 = 23

C (2, 3) = W (0, 1) + (C (2, 2) + C (**3**, 3)) = 17 + 4 + 4 = 25

C (3, 4) = W (0, 1) + (C (3, 3) + C (**4**, 4)) = 9 + 4 + 3 = 16

C (4, 5) = W (0, 1) + (C (4, 4) + C (**5**, 5)) = 11 + 3 + 8 = 22

C (5, 6) = W (0, 1) + (C (5, 5) + C (**6**, 6)) = 18 + 8 + 0 = 26

**The bolded numbers represent the elements added in the root matrix.*

C	0	1	2	3	4	5	6
0	5	32					
1		6	23				
2			4	25			
3				4	16		
4					3	22	
5						8	26
6							0

R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							

C (0, 2) = W (0, 2) + min (C (0, 0) + C (**1**, 2), C (0, 1) + C (2, 2)) = 28 + min (**28**, 36) = 56

C (1, 3) = W (1, 3) + min (C (1, 1) + C (2, 3), C (1, 2) + C (**3**, 3)) = 26 + min (31, **27**) = 53

C (2, 4) = W (2, 4) + min (C (2, 2) + C (**3**, 4), C (2, 3) + C (4, 4)) = 22 + min (**20**, 28) = 42

C (3, 5) = W (3, 5) + min (C (3, 3) + C (4, 5), C (3, 4) + C (**5**, 5)) = 17 + min (26, **24**) = 41

C (4, 6) = W (4, 6) + min (C (4, 4) + C (5, 6), C (4, 5) + C (**6**, 6)) = 21 + min (29, **22**) = 43

C	0	1	2	3	4	5	6
0	5	32	56				
1		6	23	53			
2			4	25	42		
3				4	16	41	
4					3	22	43
5						8	26
6							0

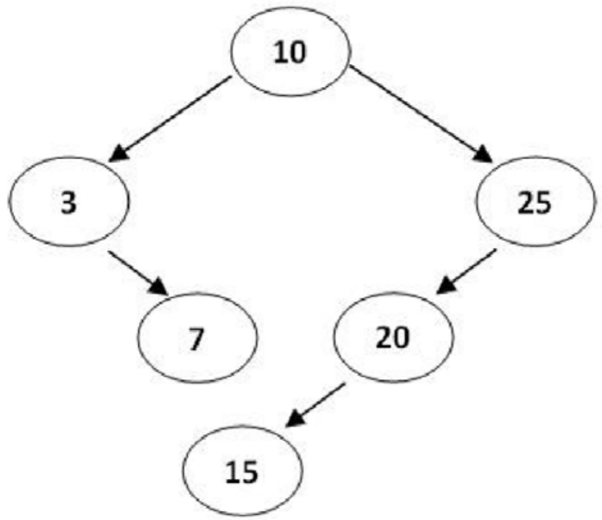
R	0	1	2	3	4	5	6
0		1	1				
1			2	3			
2				3	3		
3					4	5	
4						5	6
5							6
6							

Final array values:

C	0	1	2	3	4	5	6
0	5	32	56	98	118	151	188
1		6	23	53	70	103	140
2			4	25	42	75	108
3				4	16	41	68
4					3	22	43
5						8	26
6							0

R	0	1	2	3	4	5	6
0	0	1	1	2	3	3	3
1		0	2	3	3	3	3
2			0	3	3	3	4
3				0	4	5	6
4					0	5	6
5						0	6
6							0

The resulting optimal tree is shown in the bellow figure and has a weighted path length of 188.



Computing the node positions in the tree:

- The root of the optimal tree is R(0, 6) = k3;
- The root of the left subtree is R(0, 2) = k1;
- The root of the right subtree is R(3, 6) = k6;
- The root of the right subtree of k1 is R(1, 2) = k2
- The root of the left subtree of k6 is R(3, 5) = k5
- The root of the left subtree of k5 is R(3, 4) = k4

http://software.ucv.ro/~cmihaescu/ro/laboratoare/SDA/docs/arboriOptimali_en.pdf

Matrix Chain-Products

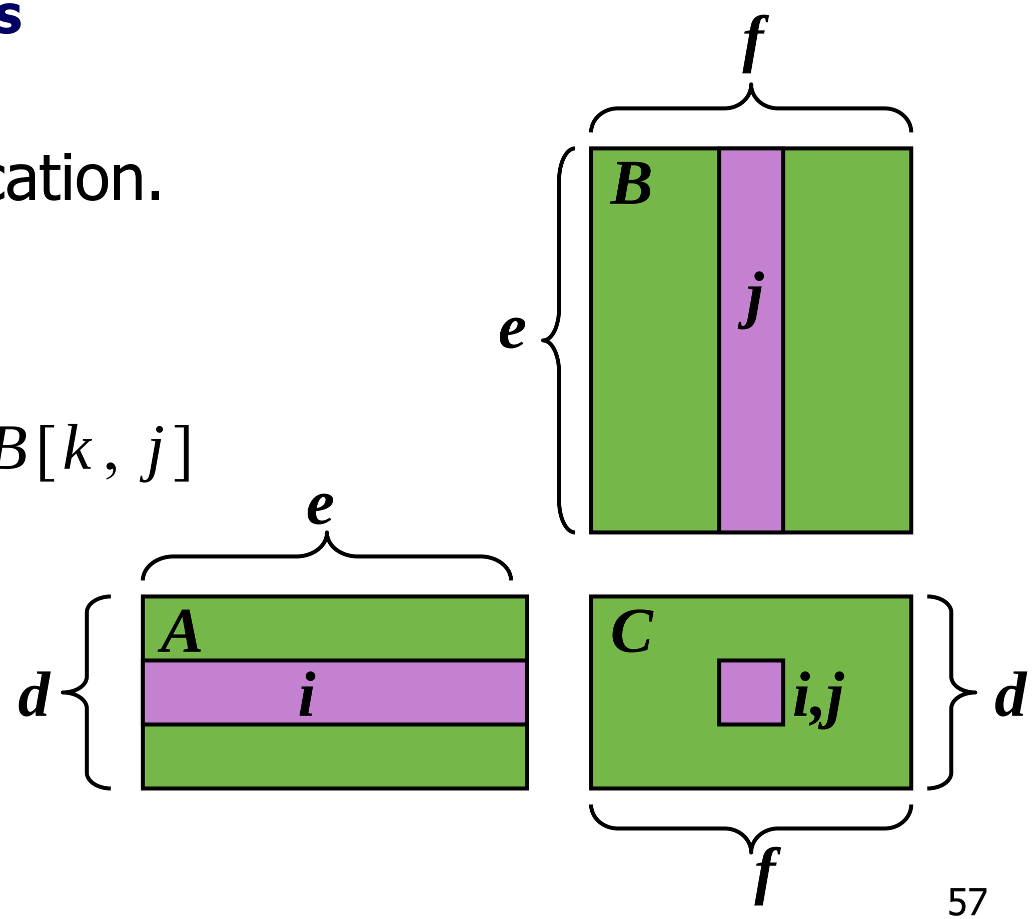
Matrix Chain-Products

- Dynamic Programming is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] * B[k, j]$$

  - $O(def)$ time



Matrix-chain multiplication

- n matrices $A_1, A_2, \ldots, A_n$ with size
  $p_1 \times p_2,\ p_2 \times p_3,\ \ldots,\ p_{n-1} \times p_n, p_n \times p_{n+1}$
- **$\mathbf{A_i}$ with size $p_i \times p_{i+1}$**
- To determine the multiplication order such that # of scalar multiplications is minimized.
- To compute $\mathbf{A_i} \times \mathbf{A_{i+1}}$, we need $p_i p_{i+1} p_{i+2}$ scalar multiplications.

e.g. n=4, A_1: 3×5, A_2: 5×4, A_3: 4×2, A_4: 2×5

$((A_1 \times A_2) \times A_3) \times A_4$, # of scalar multiplications:

$3 * 5 * 4 + 3 * 4 * 2 + 3 * 2 * 5 = 114$

$(A_1 \times (A_2 \times A_3)) \times A_4$, # of scalar multiplications:

$3 * 5 * 2 + 5 * 4 * 2 + 3 * 2 * 5 = 100$ (optimal)

$(A_1 \times A_2) \times (A_3 \times A_4)$, # of scalar multiplications:

$3 * 5 * 4 + 3 * 4 * 5 + 4 * 2 * 5 = 160$

- Note: n個matrix相乘有 $C_{n-1} = \binom{2(n-1)}{n-1} \Big/ n$ 種可能的配對組合(括號方式)
    - Ex: 以下有四個矩陣相乘:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

由Note得知共有五種不同的相乘順序，不同的順序需要不同的乘法次數：

$$\begin{array}{ll} A(B(CD)) & 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680 \\ (AB)(CD) & 20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880 \\ A((BC)D) & 2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232 \\ ((AB)C)D & 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320 \\ (A(BC))D & 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120 \end{array}$$

其中，以第三組是最佳的矩陣相乘順序。

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_1*A_2*\ldots*A_n$
  - Calculate number of operations for each one
  - Pick the one that is best
- Running time:
  - The number of paranethesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

The *n*th Catalan number can be expressed directly in terms of the central binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \geq 0.$$

Catalan number

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

Recursive formula

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i\, C_{n-i} \quad \text{for } n \ge 0.$$

它也滿足

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n,$$

這提供了一個更快速的方法來計算卡塔蘭數。

卡塔蘭數的漸近增長為

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

https://en.wikipedia.org/wiki/Catalan_number

Examples of Catalan number

- C_n is the number of Dyck words[4] of length $2n$. A Dyck word is a string consisting of n X's and n Y's such that no initial segment of the string has more Y's than X's. For example, the following are the Dyck words up to length 6:

XY

XXYY XYXY

XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY

- Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, C_n counts the number of expressions containing n pairs of parentheses which are correctly matched:

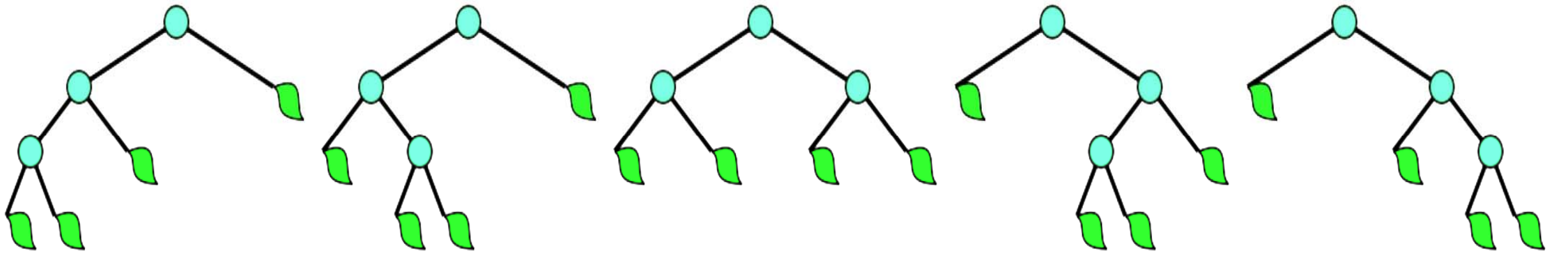
((())) (()()) (())() ()(()) ()()()

- C_n is the number of different ways $n + 1$ factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator, as in the matrix chain multiplication problem). For $n = 3$, for example, we have the following five different parenthesizations of four factors:

((ab)c)d (a(bc))d (ab)(cd) a((bc)d) a(b(cd))

Examples of Catalan number

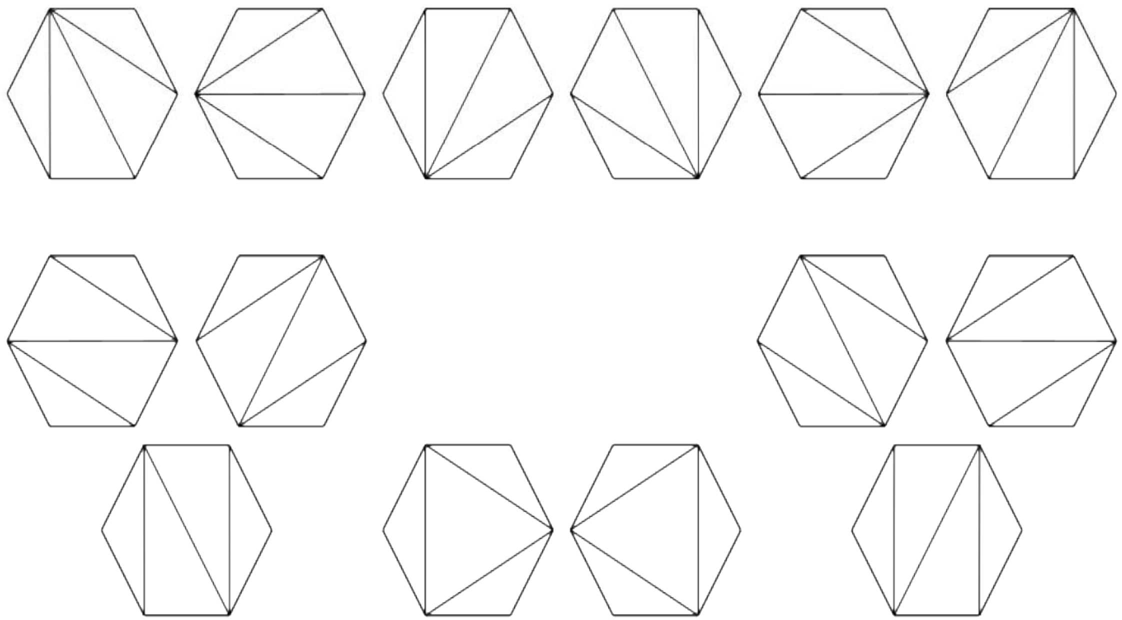
- Successive applications of a binary operator can be represented in terms of a full binary tree, by labeling each leaf *a*,*b*,*c*,*d*. It follows that C_n is the number of full binary trees with $n + 1$ leaves, or, equivalently, with a total of n internal nodes:



- C_n is the number of non-isomorphic ordered (or plane) trees with $n + 1$ vertices.[5] See encoding general trees as binary trees. For example, C_n is the number of possible parse trees for a sentence (assuming binary branching), in natural language processing.

Examples of Catalan number

- A convex polygon with $n + 2$ sides can be cut into triangles by connecting vertices with non-crossing line segments (a form of polygon triangulation). The number of triangles formed is n and the number of different ways that this can be achieved is C_n. The following hexagons illustrate the case $n = 4$:



- 六個矩陣相乘的最佳乘法順序可以分解成以下的其中一種型式：

$$A=A_1*(A_2*A_3*A_4*A_5*A_6)$$

$$A=(A_1*A_2)*(A_3*A_4*A_5*A_6)$$

$$A=(A_1*A_2*A_3)*(A_4*A_5*A_6)$$

$$A=(A_1*A_2*A_3*A_4)*(A_5*A_6)$$

$$A=A_1*(A_2*A_3*A_4*A_5*A_6)$$

- 第**k**個分解型式所需的乘法總數，爲前後兩部份 **(**一爲$\mathbf{A_1, A_2, \ldots, A_k}$和$\mathbf{A_{k+1}, \ldots, A_6}$**)** 各自所需乘法數目的最小值相加，再加上相乘這前後兩部份矩陣所需的乘法數目。

$$M_{1,6} = \min_{1 \le k < 6} \{M_{1,k} + M_{k+1,6} + p_1 p_k p_7\}$$

A “Recursive” Approach

- Define **subproblems**:
  - Find the best parenthesization of $A_i * A_{i+1} * \ldots * A_j$.
  - Let $M_{i,j}$ (or M[i][j]) denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $\mathbf{M_{1,n}}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say the final multiply is at index i: $(A_1 * \ldots * A_i) * (A_{i+1} * \ldots * A_n)$.
  - Then the optimal solution $M_{1,n}$ is the **sum of two optimal subproblems, $\mathbf{M_{1,i}}$ and $\mathbf{M_{i+1,n}}$ plus the time for the last multiply.**
  - If the global optimum did not have these subproblems, we could define an even better “optimal” solution.

A Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that **A_i is a $p_i \times p_{i+1}$** dimensional matrix.
  - So, a characterizing equation for $M_{i,j}$ is the following:

$$M_{i,j} = \min_{i \le k < j} \{ M_{i,k} + M_{k+1,j} + p_i p_{k+1} p_{j+1} \}$$

- Note that subproblems are not independent--the **subproblems overlap**.

A Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $M_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

Algorithm ***matrixChain***(***S***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal paranethization of ***S***

for $i \leftarrow 1$ **to** n **do**

  $M_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** n **do**

  for $i \leftarrow 1$ **to** $n-b$ **do**

    $j \leftarrow i+b$

    $M_{i,j} \leftarrow$ **+infinity**

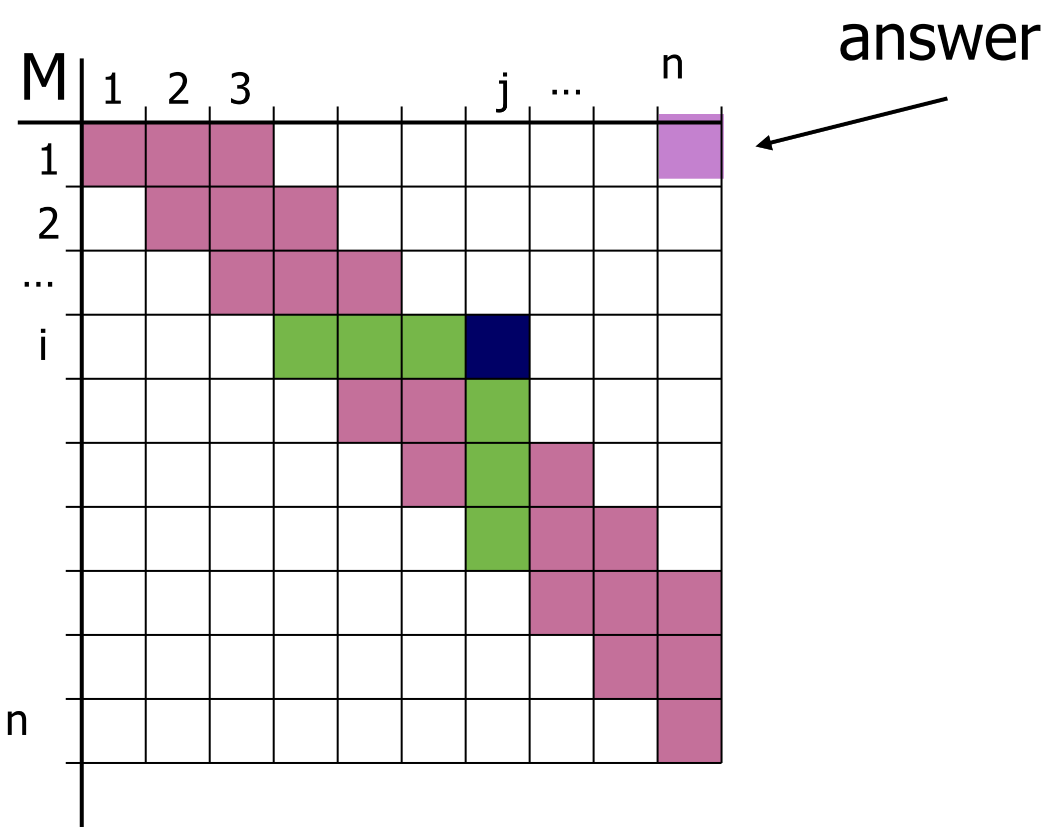
    for $k \leftarrow i$ **to** $j-1$ **do**

      $M_{i,j} \leftarrow \min\{M_{i,j}, M_{i,k} + M_{k+1,j} + p_i\, p_{k+1}\, p_{j+1}\}$

A Dynamic Programming Algorithm Visualization

$$M_{i,j} = \min_{1 \le k < j} \{M_{i,k} + M_{k+1,j} + p_i p_{k+1} p_{j+1}\}$$

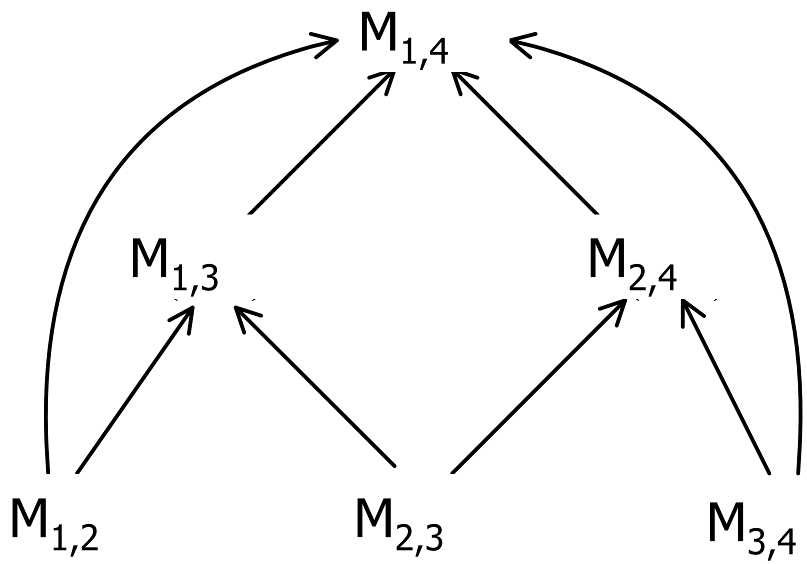
- The bottom-up construction fills in the M array by diagonals
- $M_{i,j}$ gets values from pervious entries in i-th row and j-th column
- Filling in each entry in the M table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each M entry



- Let $M_{i,j}$ denote the minimum cost for computing $A_i \times A_{i+1} \times \dots \times A_j$

$$M_{i,j} = \begin{cases} 0 & \text{if } i = j \\ M_{i,j} = \min\limits_{1 \le k < j} \{M_{i,k} + M_{k+1,j} + p_i p_{k+1} p_{j+1}\} & \text{if } i < j \end{cases}$$

- Computation sequence :



- Time complexity : $O(n^3)$

- Matrix Chain的遞迴式

$$M_{ij} = \begin{cases} 0, \ if\ i = j \\ \min\limits_{1 \le k < j} \{M_{i,k} + M_{k+1,j} + p_i p_{k+1} p_{j+1}\}, \ if\ i < j \end{cases}$$

- Example: $A^1_{3\times3}$, $A^2_{3\times7}$, $A^3_{7\times2}$, $A^4_{2\times9}$, $A^5_{9\times4}$, 求此五矩陣的最小乘法次數。

Sol:

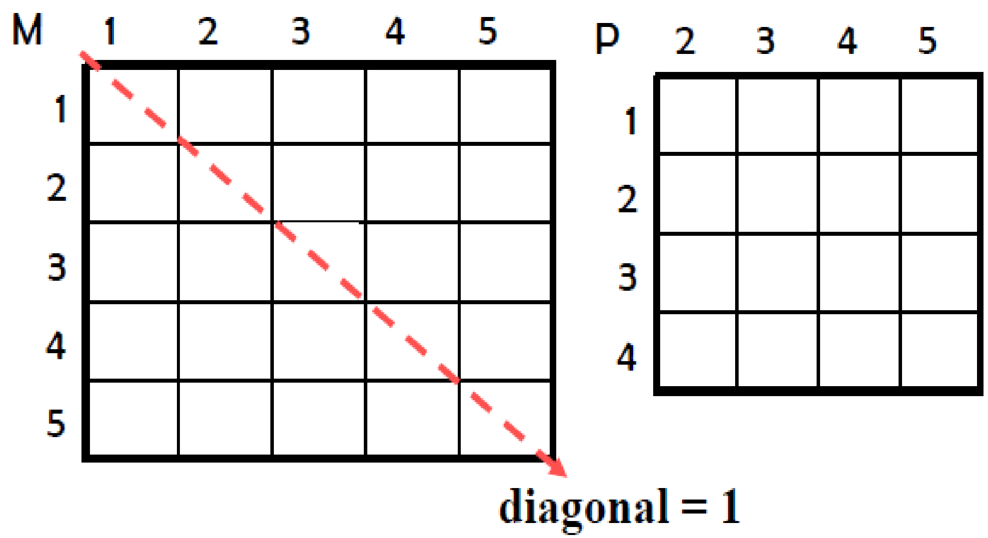
建立兩陣列 M[1...5, 1...5]及P[1...4, 2...5]

M	1	2	3	4	5
1					
2					
3					
4					
5					

P	2	3	4	5
1				
2				
3				
4				

Case ① (When diagonal = 1)

- diagonal = 1，∵只有1個矩陣，∴不會執行乘法動作
- 陣列M的中間對角線爲0，陣列P則不填任何數值



Case ② (When diagonal > 1)

- diagonal = 2，有2個矩陣相乘
- 當 i = 1及 j = 2，爲A^1及A^2矩陣相乘，此時:

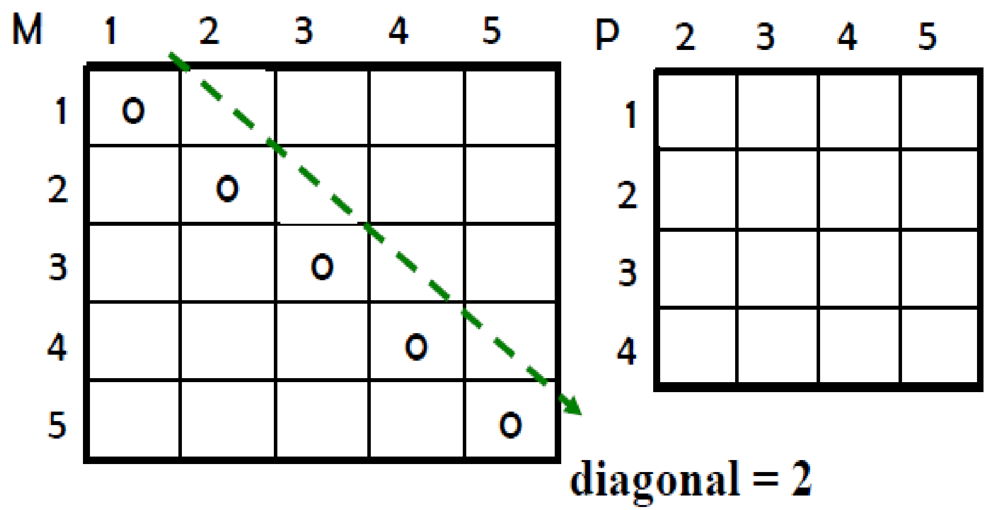
$M[1, 2] = M[1,1]+M[2,2]+3\times3\times7 = 63$，

其中 A^1 及A^2的分割點 k 如下:

$A^1\times A^2$

↑

分割點k = 1



: $A^1_{3\times3}$, $A^2_{3\times7}$, $A^3_{7\times2}$, $A^4_{2\times9}$, $A^5_{9\times4}$,

Case ② (When diagonal > 1)

: $A^1_{3\times3}, A^2_{3\times7}, A^3_{7\times2}, A^4_{2\times9}, A^5_{9\times4}$

- diagonal = 3，有3個矩陣相乘
- 當 i = 2及 j = i+diagnal-1 = 2+3-1=4，爲A^2至A^4間的所有矩陣相乘，此時:

$$M[2,4] = \min \begin{cases} M[2,2]+M[3,4]+3\times7\times9=315, & \text{分割點}k=2 \\ M[2,3]+M[4,4]+3\times2\times9=96, & \text{分割點}k=3 \end{cases}$$

M	1	2	3	4	5
1	0	63	60		
2		0	42	96	
3			0	126	128
4				0	72
5					0

diagonal = 3

P	2	3	4	5
1	1	1		
2		2	3	
3			3	3
4				4

Case ② (When diagonal > 1)

: $A^1_{3\times3}, A^2_{3\times7}, A^3_{7\times2}, A^4_{2\times9}, A^5_{9\times4}$,

- diagonal = 4，有4個矩陣
- 當 i = 1及 j = 4，為A^1至A^4間的所有矩陣相乘，此時:

M	1	2	3	4	5
1	0	63	60	114	
2		0	42	96	138
3			0	126	128
4				0	72
5					0

diagonal = 4

P	2	3	4	5
1	1	1	3	
2		2	3	3
3			3	3
4				4

$$M[1,4] = \min\begin{cases} M[1,1] + M[2,4] + 3\times3\times9 = 177, & \text{分割點} k = 1 \\ M[1,2] + M[3,4] + 3\times7\times9 = 378, & \text{分割點} k = 2 \\ M[1,3] + M[4,4] + 3\times2\times9 = 114, & \text{分割點} k = 3 \end{cases}$$

Case ② (When diagonal > 1)

: $A^1_{3\times3}, A^2_{3\times7}, A^3_{7\times2}, A^4_{2\times9}, A^5_{9\times4},$

- diagonal = 5，有5個矩陣
- 當 i = 1及 j = 5，為A^1至A^5間所有矩陣相乘，此時:

M	1	2	3	4	5
1	0	63	60	114	156
2		0	42	96	138
3			0	126	128
4				0	72
5					0

diagonal = 5

P	2	3	4	5
1	1	1	3	3
2		2	3	3
3			3	3
4				4

$$M[1,5] = \min\begin{cases} M[1,1]+M[2,5]+3\times3\times4=174, & \text{分割點}k=1 \\ M[1,2]+M[3,5]+3\times7\times4=275, & \text{分割點}k=2 \\ M[1,3]+M[4,5]+3\times2\times4=156, & \text{分割點}k=3 \\ M[1,4]+M[5,5]+3\times9\times4=222, & \text{分割點}k=4 \end{cases}$$

◆[Note]此演算法的概念如下: $\boxed{A^1_{3\times3}, A^2_{3\times7}, A^3_{7\times2}, A^4_{2\times9}, A^5_{9\times4}}$

$A^1, \quad A^2, \quad A^3, \quad A^4, \quad A^5$ diagonal = 1

$A^1A^2, \quad A^2A^3, \quad A^3A^4, \quad A^4A^5$ diagonal = 2

$A^1A^2A^3, \quad A^2A^3A^4, \quad A^3A^4A^5$ diagonal = 3

$A^1A^2A^3A^4, \quad A^2A^3A^4A^5$ diagonal = 4

$A^1A^2A^3A^4A^5$ diagonal = 5

Related Problems

- **Easy:**
  - **Fibonacci numbers**
  - **nth Catalan Number**
  - **Bell Numbers (Number of ways to Partition a Set)**
  - **Binomial Coefficient**
  - **Coin change problem**
  - **Subset Sum Problem**
  - **Compute nCr % p**
  - **Cutting a Rod**

https://www.geeksforgeeks.org/dynamic-programming/

Related Problems

- **Easy:**
  - Painting Fence Algorithm
  - Longest Common Subsequence
  - Longest Increasing Subsequence
  - Longest subsequence such that difference between adjacents is one
  - Maximum size square sub-matrix with all 1s
  - Min Cost Path
  - Minimum number of jumps to reach end
  - Longest Common Substring (Space optimized DP solution)
  - Count ways to reach the nth stair using step 1, 2 or 3
  - Count all possible paths from top left to bottom right of a mXn matrix
  - Unique paths in a Grid with Obstacles

Related Problems

1. **Medium:**
    1. **Floyd Warshall Algorithm**
    2. **Bellman–Ford Algorithm**
    3. **0-1 Knapsack Problem**
    4. **Printing Items in 0/1 Knapsack**
    5. **Unbounded Knapsack (Repetition of items allowed)**
    6. **Egg Dropping Puzzle**
    7. **Word Break Problem**
    8. **Vertex Cover Problem**
    9. **Tile Stacking Problem**
    10. **Box-Stacking Problem**

Related Problems

1. Partition Problem
2. Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)
3. Longest Palindromic Subsequence
4. Longest Common Increasing Subsequence (LCS + LIS)
5. Find all distinct subset (or subsequence) sums of an array
6. Weighted job scheduling
7. Count Derangements (Permutation such that no element appears in its original position)
8. Minimum insertions to form a palindrome
9. Wildcard Pattern Matching
10. Ways to arrange Balls such that adjacent balls are of different types

Related Problems

- **Hard:**
  - **Palindrome Partitioning**
  - **Word Wrap Problem**
  - **The painter's partition problem**
  - **Program for Bridge and Torch problem**
  - **Matrix Chain Multiplication**
  - **Printing brackets in Matrix Chain Multiplication Problem**
  - **Maximum sum rectangle in a 2D matrix**
  - **Maximum profit by buying and selling a share at most k times**
  - **Minimum cost to sort strings using reversal operations of different costs**
  - **Count of AP (Arithmetic Progression) Subsequences in an array**
  - **Introduction to Dynamic Programming on Trees**
  - **Maximum height of Tree when any Node can be considered as Root**
  - **Longest repeating and non-overlapping substring**