Chapter 7

Dynamic Programming

Outline

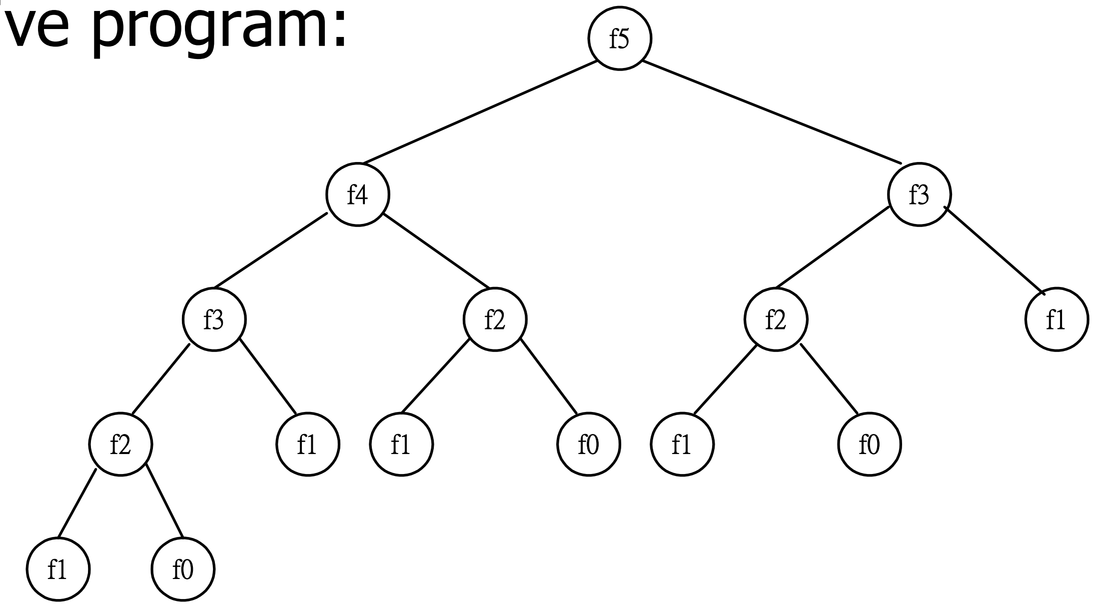
- Introduction
- The resource allocation problem
- The traveling salesperson (TSP) problem
- Longest common subsequence problem
- 0/1 knapsack problem
- The optimal binary tree problem
- Matrix Chain-Products

Fibonacci sequence

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

  $F_i = i$ if $i \leq 1$

  $F_i = F_{i-1} + F_{i-2}$ if $i \geq 2$
- Solved by a recursive program:



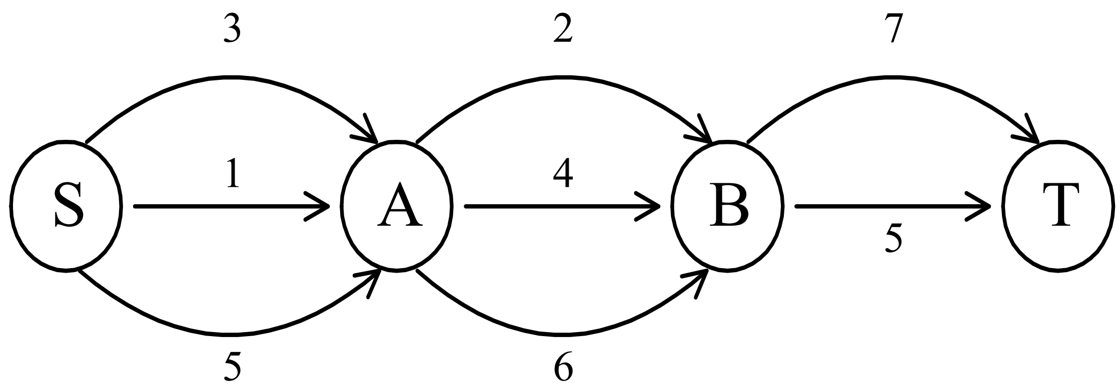
- Much replicated computation is done.
- It should be solved by a simple loop.

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

The shortest path

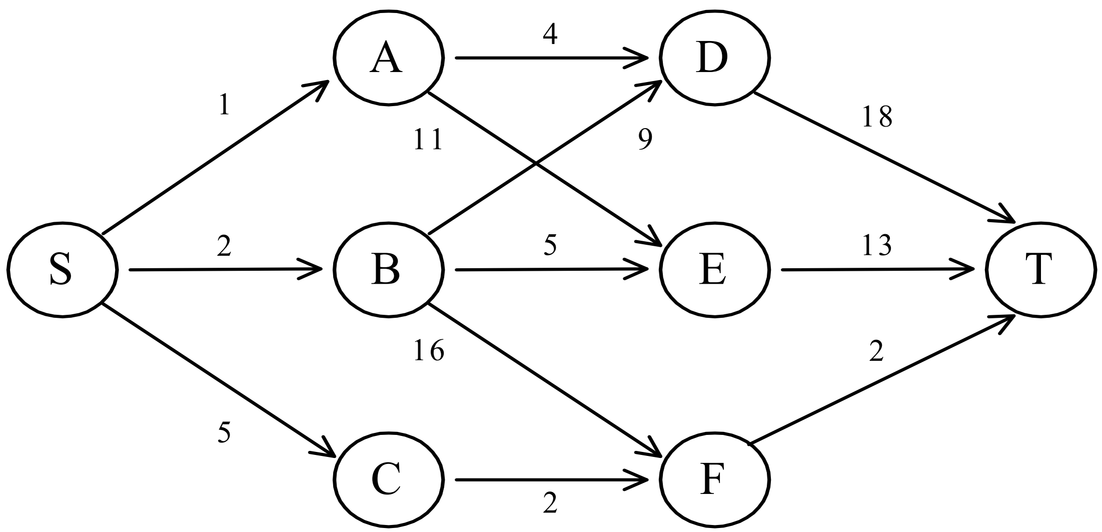
- To find a shortest path in a multi-stage graph



- Apply the greedy method :
  the shortest path from S to T :
  $1 + 2 + 5 = 8$

The shortest path in multistage graphs

- e.g.

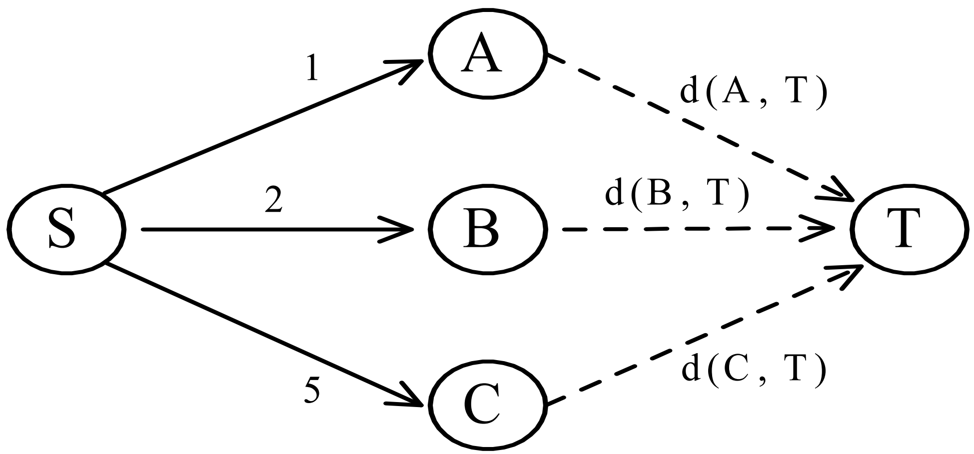


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

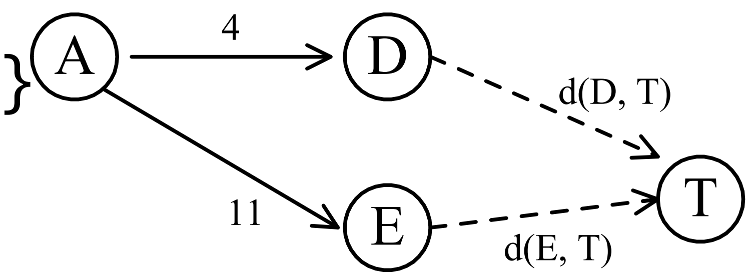
  (S, C, F, T) 5+2+2 = 9.

Dynamic programming approach

- Dynamic programming approach

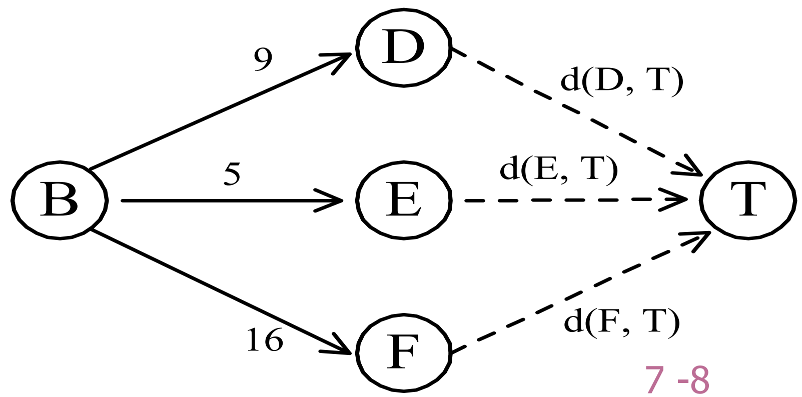


- $d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}$
- $d(A,T) = \min\{4+d(D,T), 11+d(E,T)\}$
  $= \min\{4+18, 11+13\} = 22.$

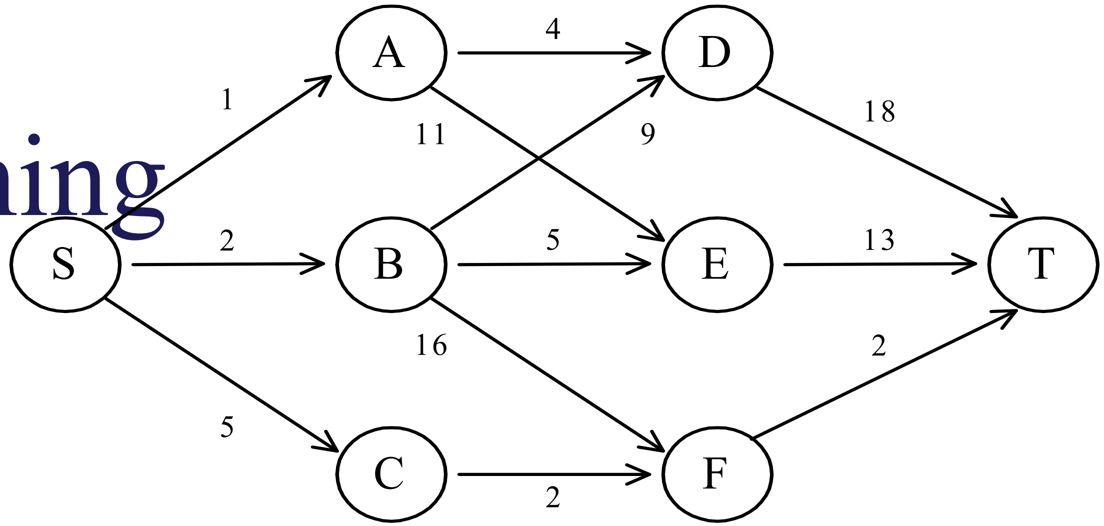


Dynamic programming

- $d(B, T) = \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\}$
  $= \min\{9+18, 5+13, 16+2\} = 18.$
- $d(C, T) = \min\{ 2+d(F, T) \} = 2+2 = 4$
- $d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}$
  $= \min\{1+22, 2+18, 5+4\} = 9.$
- The above way of reasoning is called backward reasoning.



Forward reasoning



- d(S, 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

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

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

  $= 9$

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 $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j
- In summary, if a problem can be described by a **multistage graph**, then 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 backwards . 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.

7-1 The resource allocation problem

- m resources, n projects

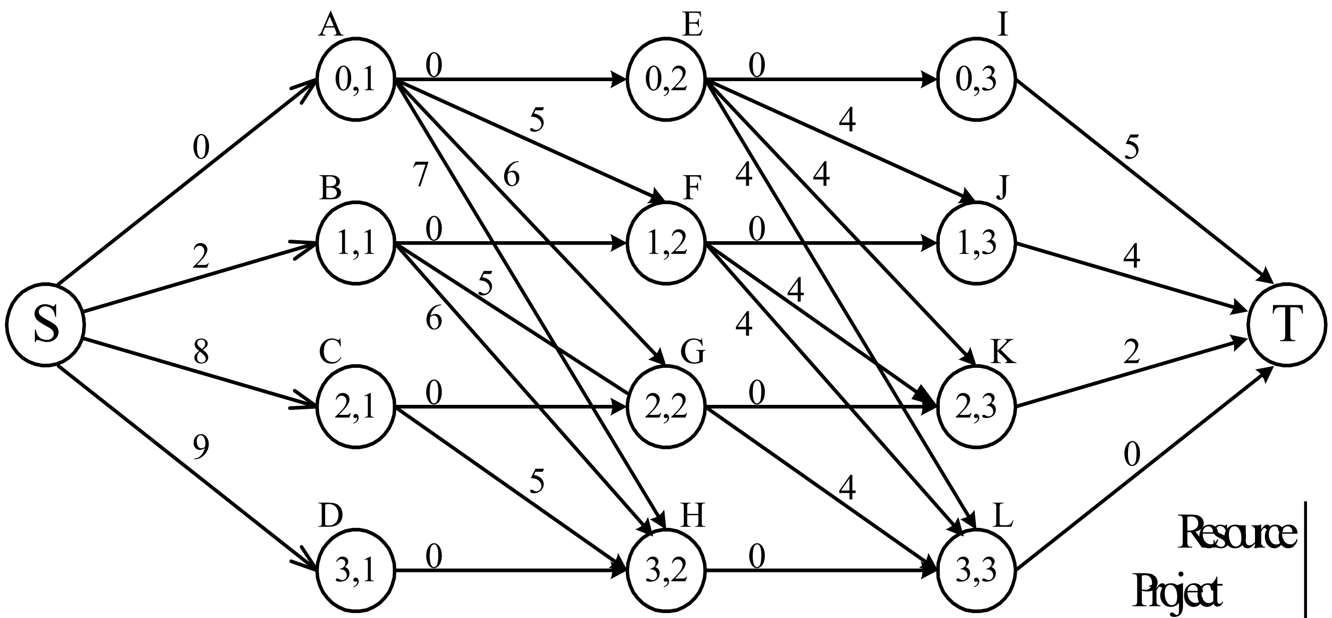
  profit p(i, j) : j resources are allocated to project i. P(*i*, 0)=0 for each *i*

  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



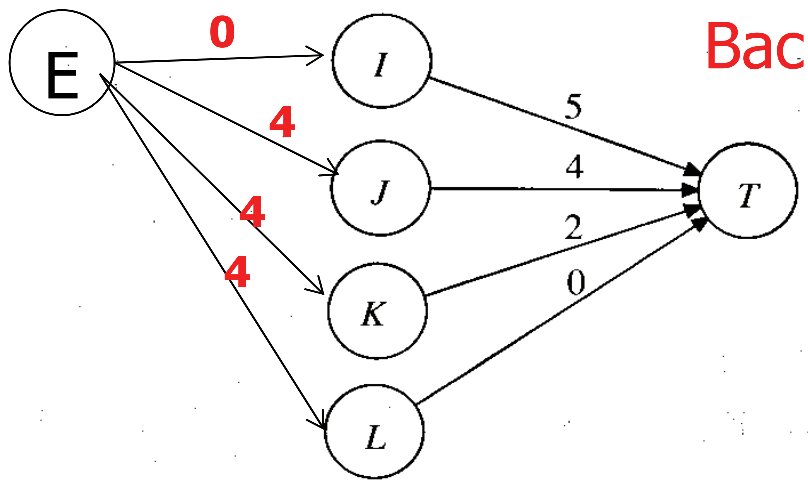
Project \ Resource	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

- 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.

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

FIGURE 7–10 The longest paths from I, J, K and L to T.

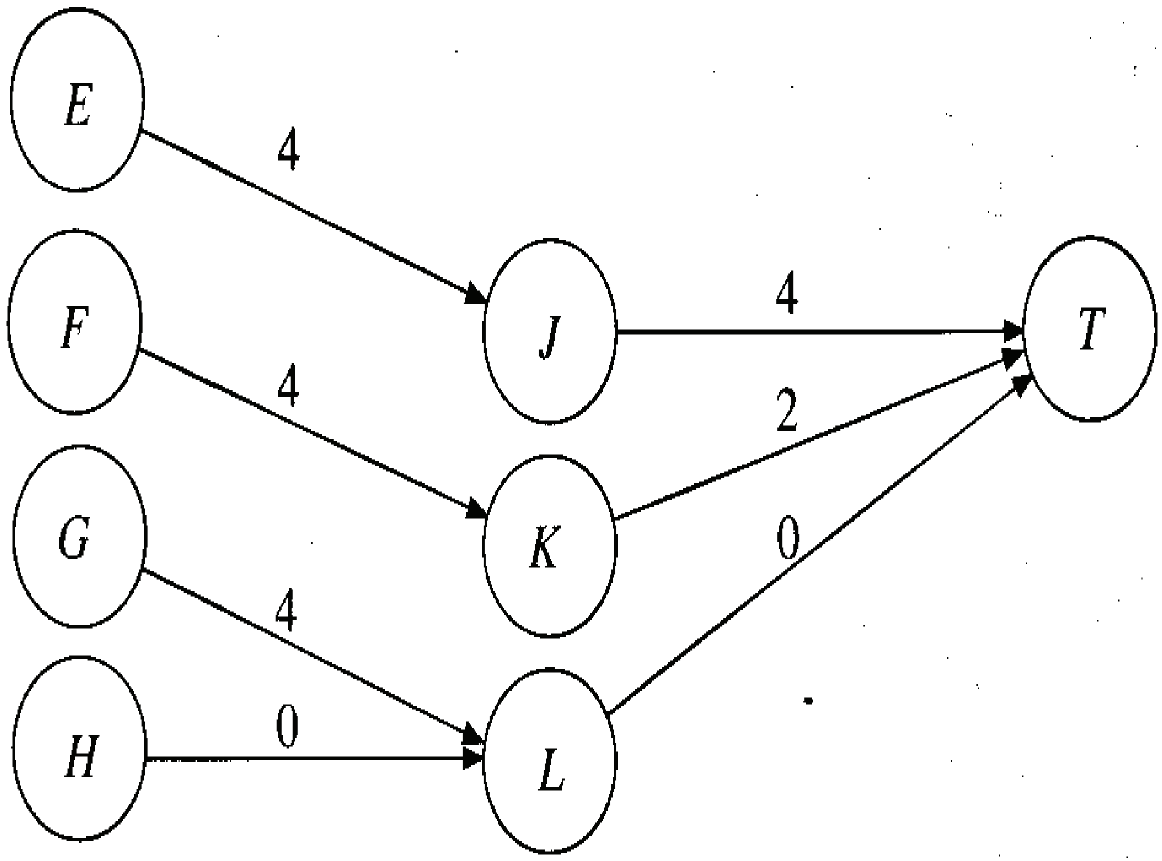


Backward reasoning approach

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:

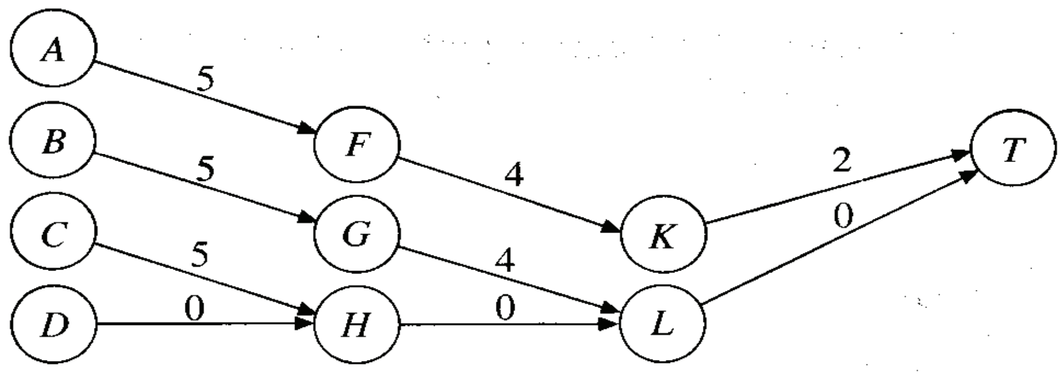
$$\begin{aligned} d(E, T) &= \max\{d(E, I) + d(I, T), d(E, J) + d(J, T), \\ &\qquad d(E, K) + d(K, T), d(E, L) + d(L, T)\} \\ &= \max\{0 + 5, 4 + 4, 4 + 2, 4 + 0\} \\ &= \max\{5, 8, 6, 4\} \\ &= 8. \end{aligned}$$

FIGURE 7–11 The longest paths from E, F, G and H to T.



(3) The longest paths from A, B, C and D to T respectively are found by the same method and shown in Figure 7–12.

FIGURE 7–12 The longest paths from A, B, C and D to T.



(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

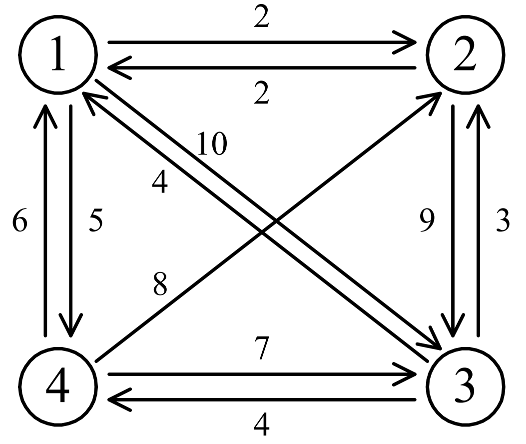
  2 resources allocated to project 1.

  1 resource allocated to project 2.

  0 resource allocated to projects 3, 4.

The traveling salesperson (TSP) problem

- e.g. a directed graph :

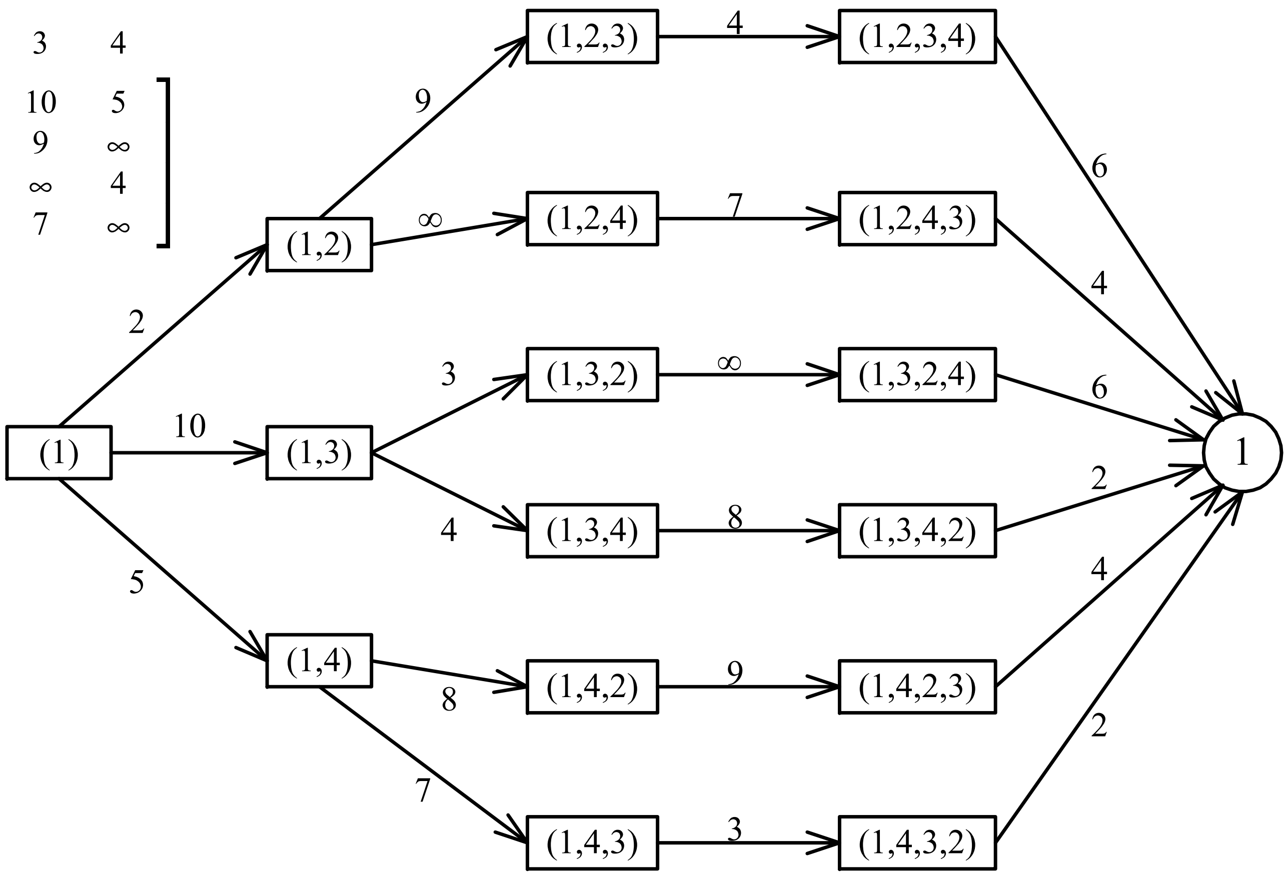


- Cost matrix:

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

The multistage graph solution

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

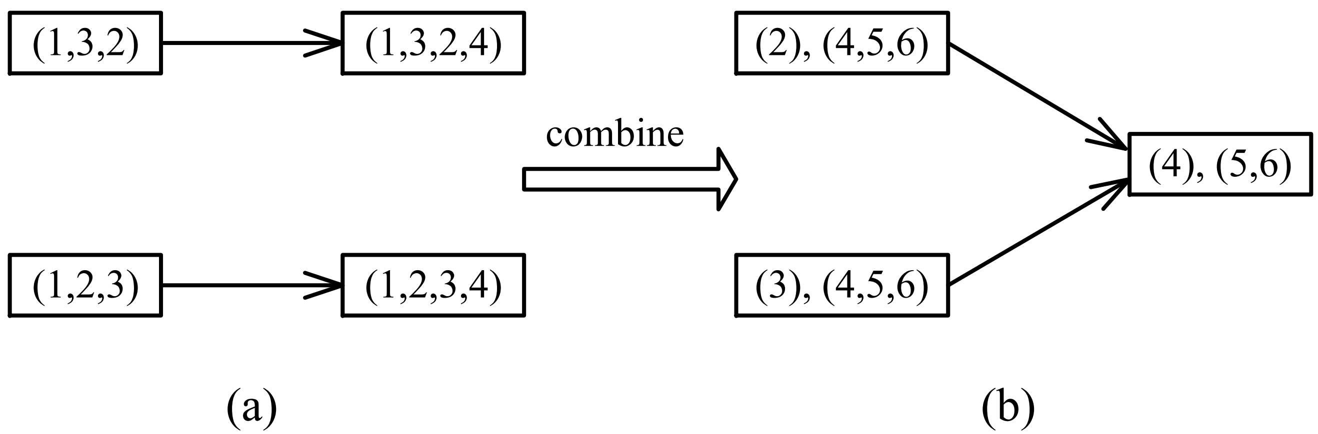


- A multistage graph can describe all possible tours of a directed graph.
- Find the shortest path:

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

The representation of a node

- Suppose that we have 6 vertices in the graph.
- We can combine {1, 2, 3, 4} and {1, 3, 2, 4} into one node.



- (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 a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1.
- The length of an optimal tour :

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

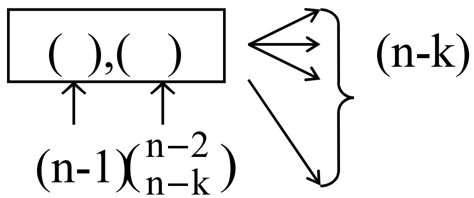
- The general form:

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

- Time complexity:

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

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



7-2 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 have to 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 $b_1b_2\ldots b_{n-1}$

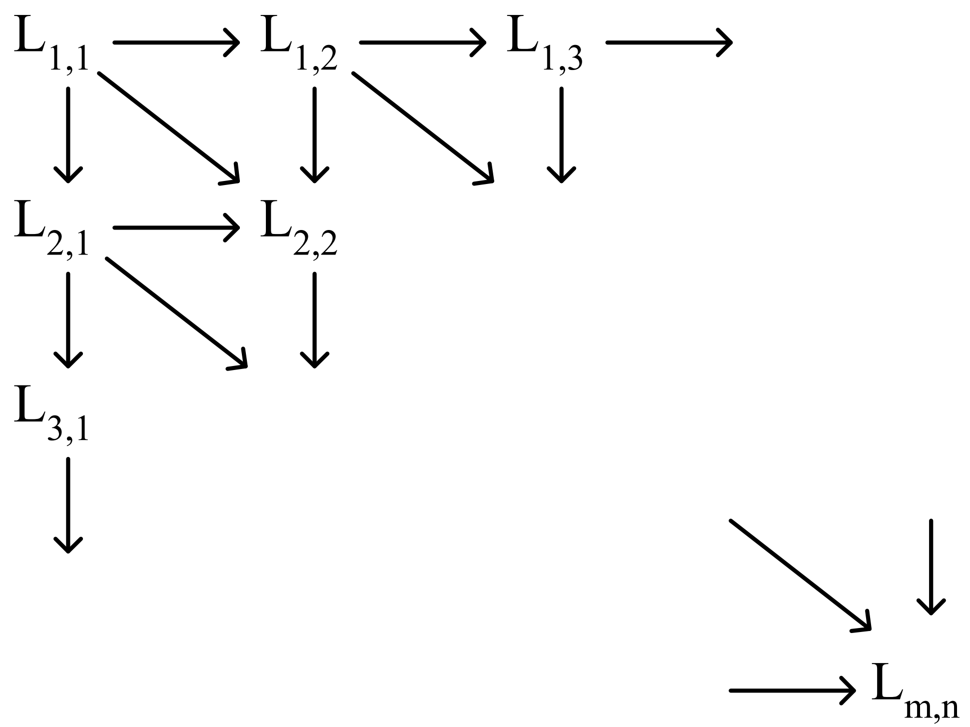
The LCS algorithm

- 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 $1 \le i \le m$, $1 \le j \le n$.

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

- 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

0/1 knapsack problem

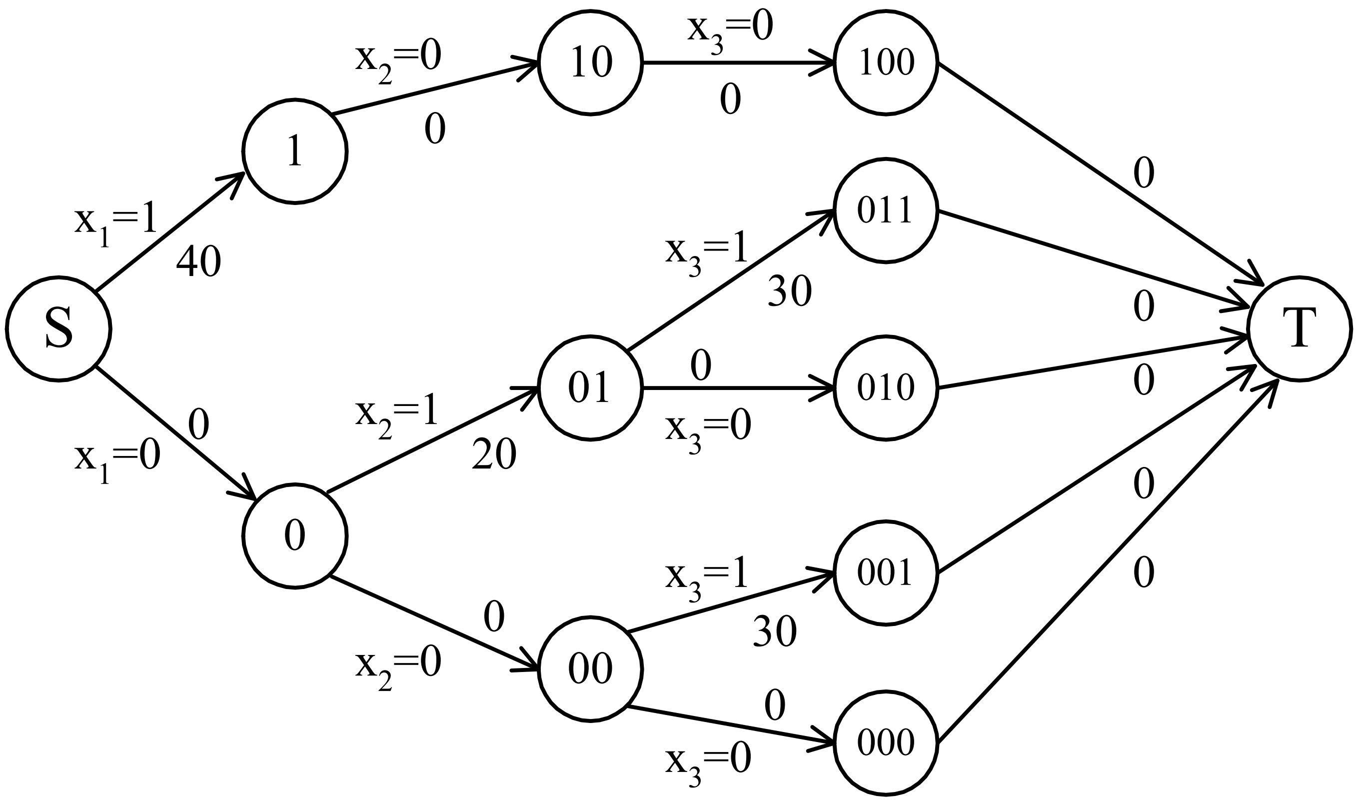
There are a sequence of actions to be taken. Let X_i be the variable denoting whether object i is chosen or not. That is, we let $X_i = 1$ if object i is chosen and 0 if it is not. If X_1 is assigned 1 (object 1 is chosen), then the remaining problem becomes a modified 0/1 knapsack problem where M becomes $M - W_1$. In general, after a sequence of decisions represented by $X_1, X_2, \ldots, X_i$ are made, the problem will be reduced to a problem involving decisions $X_{i+1}, X_{i+2}, \ldots, X_n$ and

$M' = M - \sum_{j=1}^{i} X_j W_j$. Thus, whatever the decisions $X_1, X_2, \ldots, X_i$ are, the rest of

decisions $X_{i+1}, X_{i+2}, \ldots, X_n$ must be optimal with respect to the new knapsack

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

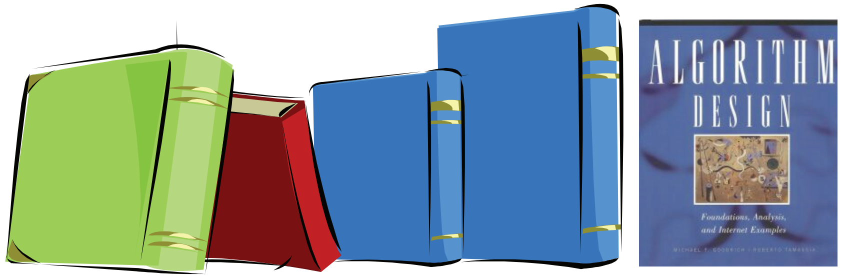
The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint: $\sum_{i \in T} w_i \leq W$

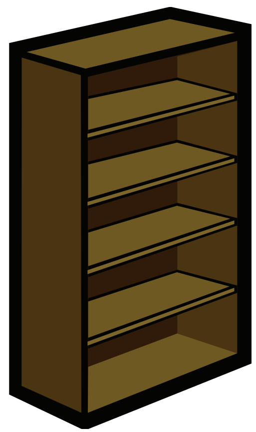
Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80



"knapsack"

9 in

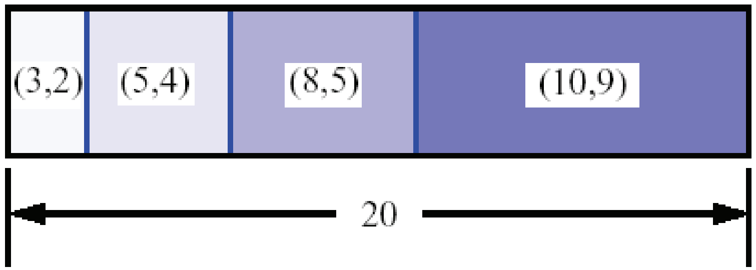
Solution:
- 5 (2 in)
- 3 (2 in)
- 1 (4 in)

A 0/1 Knapsack Algorithm, First Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider S={(3,2),(5,4),(8,5),(4,3),10,9)} weight-benefit pairs

Best for S_4:

Best for S_5:



A 0/1 Knapsack Algorithm, Second Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k,w] = best selection from S_k with weight exactly equal to w
- Good news: this does have subproblem optimality:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., best subset of S_k with weight exactly w is either the best subset of S_{k-1} w/ weight w or the best subset of S_{k-1} w/ weight $w-w_k$ plus item k.

The 0/1 Knapsack Algorithm

- Recall definition of B[k,w]:

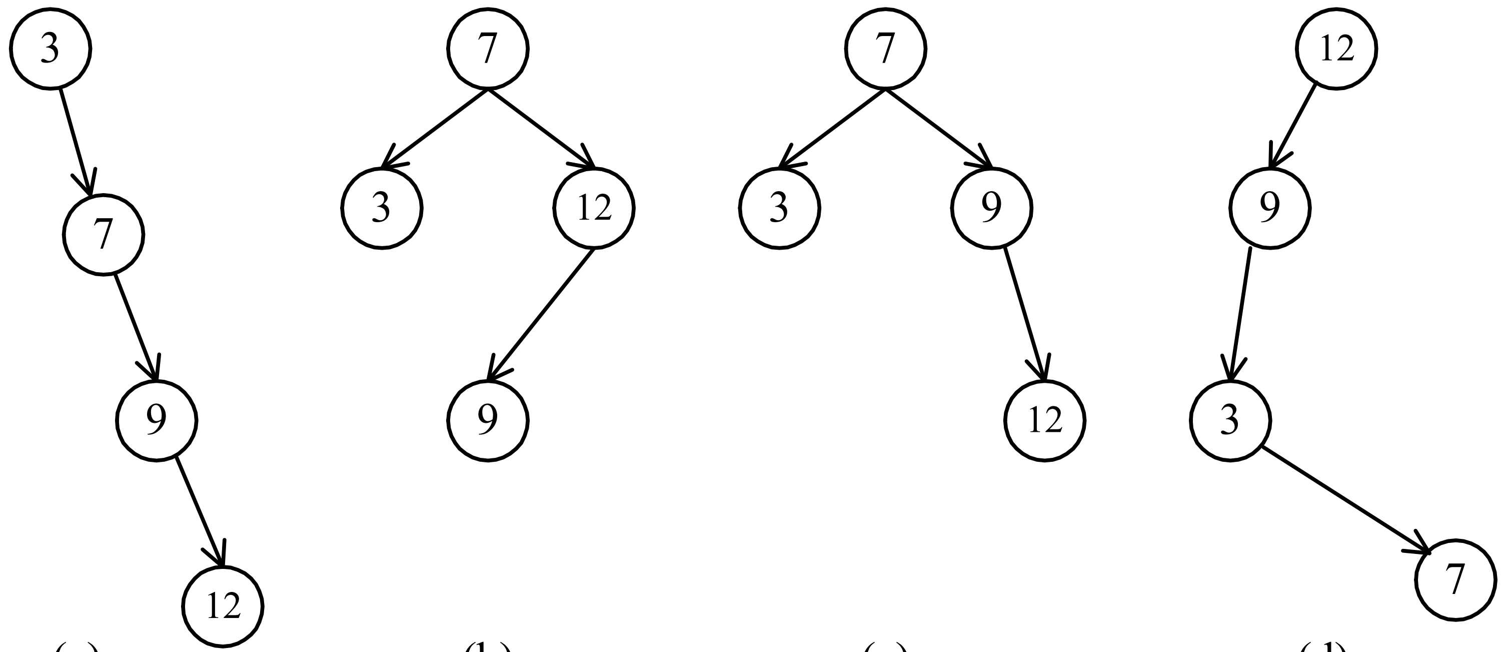
$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Since B[k,w] is defined in terms of B[k-1,*], we can reuse the same array
- Running time: O(nW).
- Not a polynomial-time algorithm if W is large
- This is a pseudo-polynomial time algorithm

```
Algorithm 01Knapsack(S, W):
    Input: set S of items w/ benefit b_i
        and weight w_i; max. weight W
    Output: benefit of best subset with
        weight at most W
    for w ← 0 to W do
        B[w] ← 0
    for k ← 1 to n do
        for w ← W downto w_k do
            if B[w-w_k]+b_k > B[w] then
                B[w] ← B[w-w_k]+b_k
```

Optimal binary search trees

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



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

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.

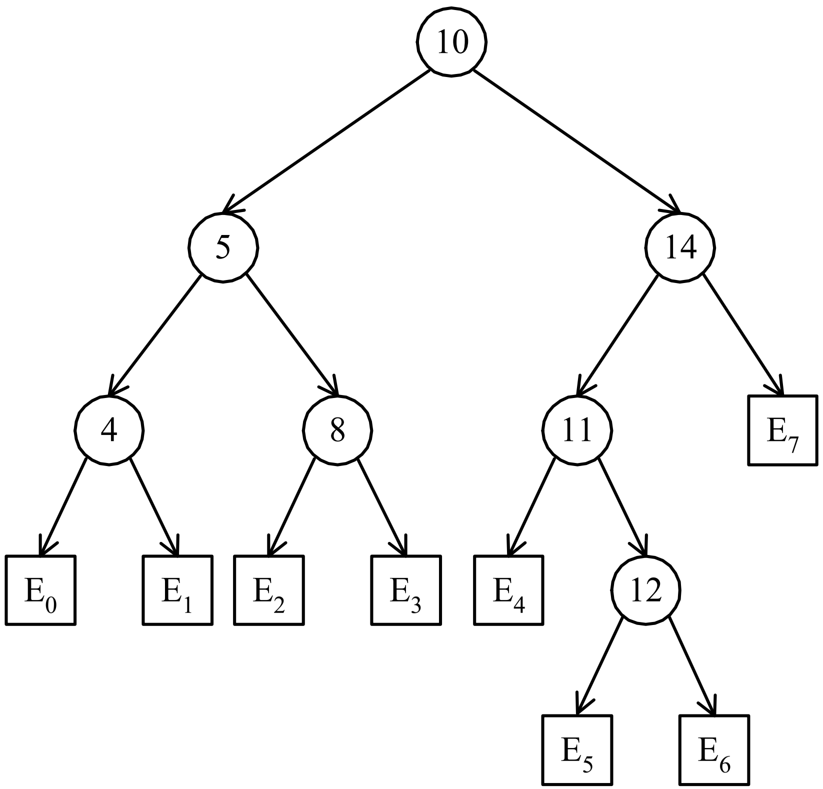
Optimal binary search trees

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

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

Q_i, $0 \le i \le n$: 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:

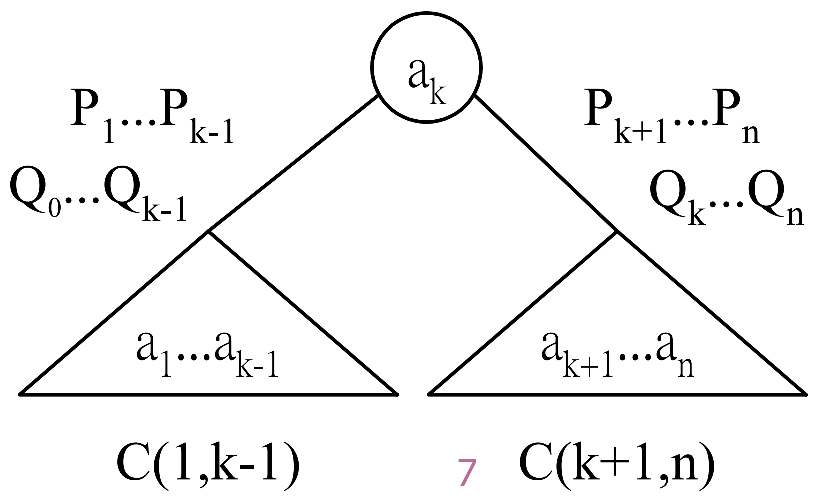
$$\sum_{n=1}^{n} P_i * \text{level}(a_i) + \sum_{n=0}^{n} Q_i * (\text{level}(E_i) - 1)$$

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

The dynamic programming approach

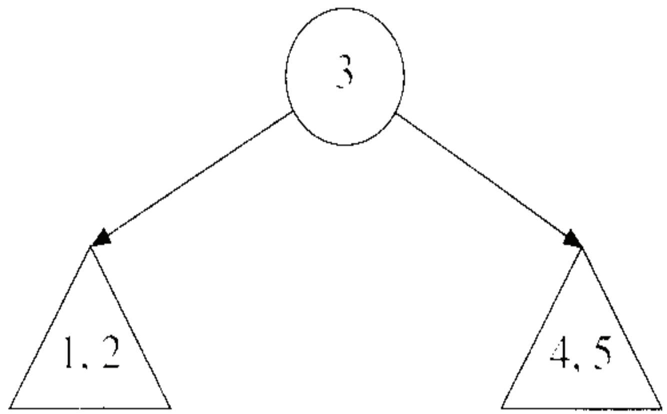
- Select an identifier, a_k, to be the root of the tree, all identifier < a_k ($>a_k$) will constitute the left (right) descendant.
- Let C(i, j) denote the cost of an optimal binary search tree containing $a_i,\ldots,a_j$.
- The cost of the optimal binary search tree with a_k as its root :

$$C(1,n)=\min_{1\le k\le n}\left\{P_k+\left[Q_0+\sum_{i=1}^{k-1}(P_i+Q_i)+C(1,k-1)\right]+\left[Q_k+\sum_{i=k+1}^{n}(P_i+Q_i)+C(k+1,n)\right]\right\}$$



First step for construct a binary tree

FIGURE 7–23 A binary tree with a certain identifier selected as the root.



(a) A subtree containing 1 and 2 with 1 as its root.
(b) A subtree containing 1 and 2 with 2 as its root.
(c) A subtree containing 4 and 5 with 4 as its root.
(d) A subtree containing 4 and 5 with 5 as its root.

Consider 1, 2, 3, 4

(a_k, a_i->a_j) denote an optimal binary tree containing identifier a_i to a_j and with a_k as its root.

(a_i->a_j) denote the optimal binary tree containing identifiers a_i to a_j.

(1) We start by finding

- $(1, 1 \to 2)$
- $(2, 1 \to 2)$
- $(2, 2 \to 3)$
- $(3, 2 \to 3)$
- $(3, 3 \to 4)$
- $(4, 3 \to 4)$.

(2) Using the above results, we can determine

- $(1 \to 2)$ (Determined by $(1, 1 \to 2)$ and $(2, 1 \to 2)$)
- $(2 \to 3)$
- $(3 \to 4)$.

(3) We then find

- $(1, 1 \to 3)$ (Determined by $(2 \to 3)$)
- $(2, 1 \to 3)$
- $(3, 1 \to 3)$
- $(2, 2 \to 4)$
- $(3, 2 \to 4)$
- $(4, 2 \to 4)$.

(4) Using the above results, we can determine

- $(1 \to 3)$ (Determined by $(1, 1 \to 3)$, $(2, 1 \to 3)$ and $(3, 1 \to 3)$)
- $(2 \to 4)$.

(5) We then find

$(1, 1 \rightarrow 4)$ (Determined by $(2 \rightarrow 4)$)

$(2, 1 \rightarrow 4)$

$(3, 1 \rightarrow 4)$

$(4, 1 \rightarrow 4)$.

(6) Finally, we can determine

$(1 \rightarrow 4)$

because it is determined by

$(1, 1 \rightarrow 4)$

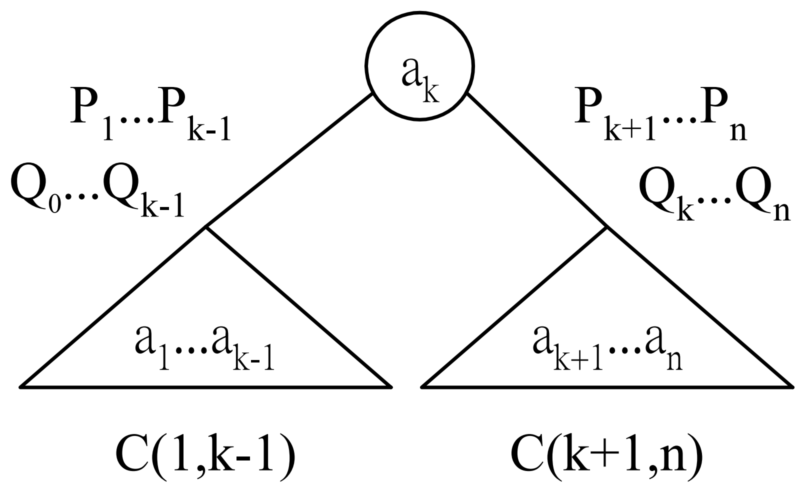
$(2, 1 \rightarrow 4)$

$(3, 1 \rightarrow 4)$

$(4, 1 \rightarrow 4)$.

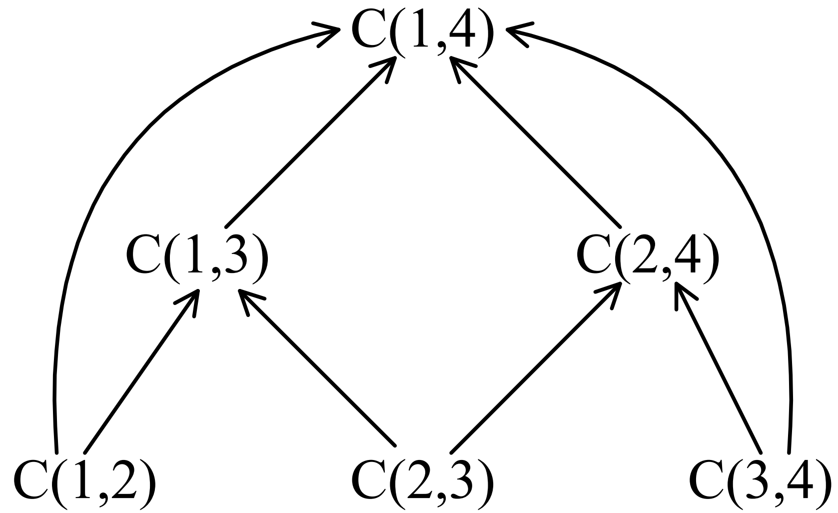
General formula

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



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.

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

Exercise

EXAMPLE OF RUNNING THE ALGORITHM

- Find the optimal binary search tree for $N = 6$, having keys $k_1 \dots k_6$ 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.

Initial array values:

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							

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

The elements of the cost matrix are afterwards computed following a pattern of lines that are parallel with the main diagonal.

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

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							

$C(0, 2) = W(0, 2) + \min(C(0, 0) + C(\mathbf{1}, 2), C(0, 1) + C(2, 2)) = 28 + \min(\mathbf{28}, 36) = 56$

$C(1, 3) = W(1, 3) + \min(C(1, 1) + C(2, 3), C(1, 2) + C(\mathbf{3}, 3)) = 26 + \min(31, \mathbf{27}) = 53$

$C(2, 4) = W(2, 4) + \min(C(2, 2) + C(\mathbf{3}, 4), C(2, 3) + C(4, 4)) = 22 + \min(\mathbf{20}, 28) = 42$

$C(3, 5) = W(3, 5) + \min(C(3, 3) + C(4, 5), C(3, 4) + C(\mathbf{5}, 5)) = 17 + \min(26, \mathbf{24}) = 41$

$C(4, 6) = W(4, 6) + \min(C(4, 4) + C(5, 6), C(4, 5) + C(\mathbf{6}, 6)) = 21 + \min(29, \mathbf{22}) = 43$

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:

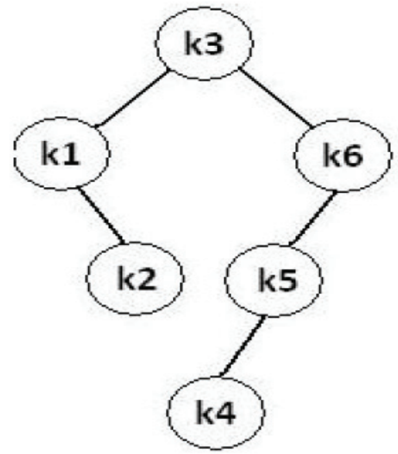
C	0	1	2	3	4	5	6
0	5	32	56	98	118	151	188
1		6	23	53	70	103	140
2			4	25	42	75	108
3				4	16	41	68
4					3	22	43
5						8	26
6							0

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

Code example

15

```
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
      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 and root
```

e紀錄expected cost, root紀錄選擇結果

邊界起始值

填表: 兩層迴圈, 對角線順序

$\boxed{\Theta(n^3)}$

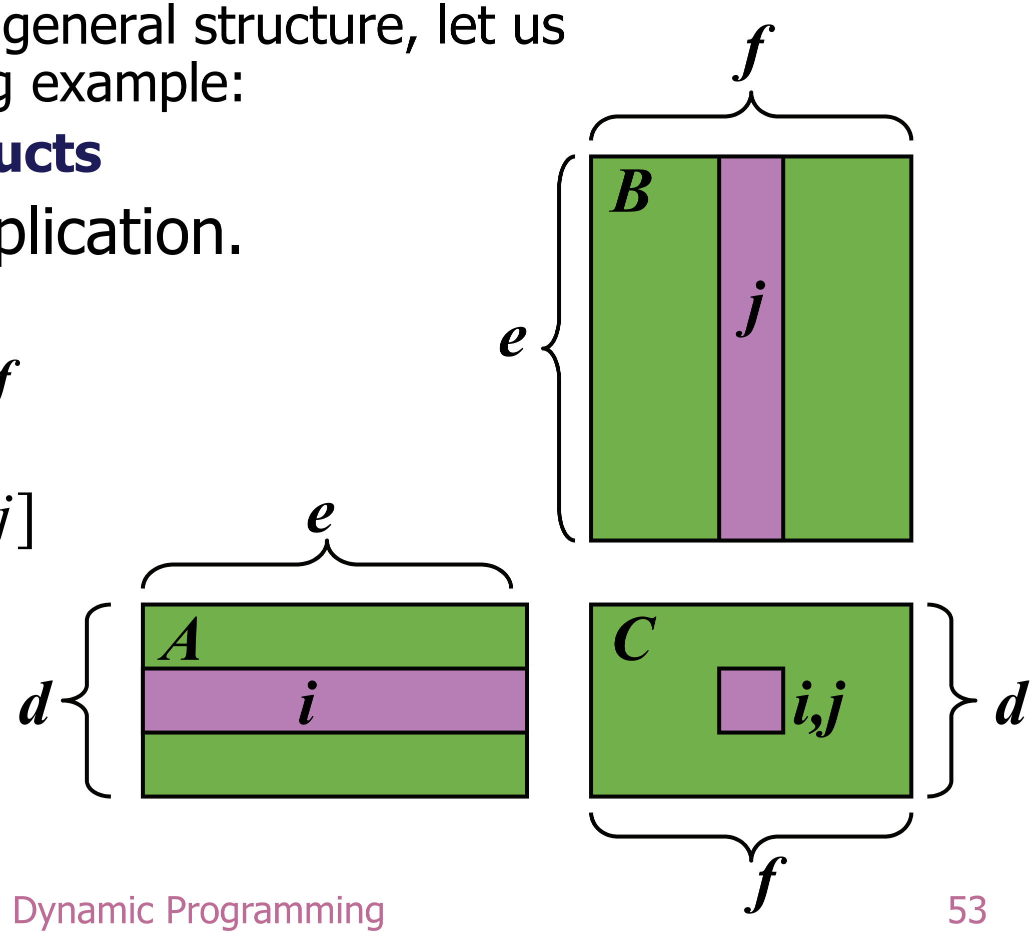
https://www.youtube.com/watch?v=8d0pazeCpgE

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



補充 Matrix-chain multiplication

- n matrices $A_1, A_2, \ldots, A_n$ with size
  $p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n$
  To determine the multiplication order such that # of scalar multiplications is minimized.
- To compute $A_i \times A_{i+1}$, we need $p_{i-1}p_ip_{i+1}$ 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:

$3 * 5 * 2 + 5 * 4 * 2 + 3 * 2 * 5 = 100$

$(A_1 \times A_2) \times (A_3 \times A_4)$, # 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。

Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops 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!

A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

- 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：

  1. $A_1 (A_2A_3A_4A_5A_6)$
  2. $(A_1A_2) (A_3A_4A_5A_6)$
  3. $(A_1A_2A_3) (A_4A_5A_6)$
  4. $(A_1A_2A_3A_4) (A_5A_6)$
  5. $(A_1A_2A_3A_4A_5) (A_6)$

- 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。

$$M[1][6] = \underset{1 \le k \le 5}{minimum}(M[1][k] + M[k+1][6] + d_0 d_k d_6).$$

A "Recursive" Approach

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **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: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal 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 $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

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

A Dynamic Programming Algorithm

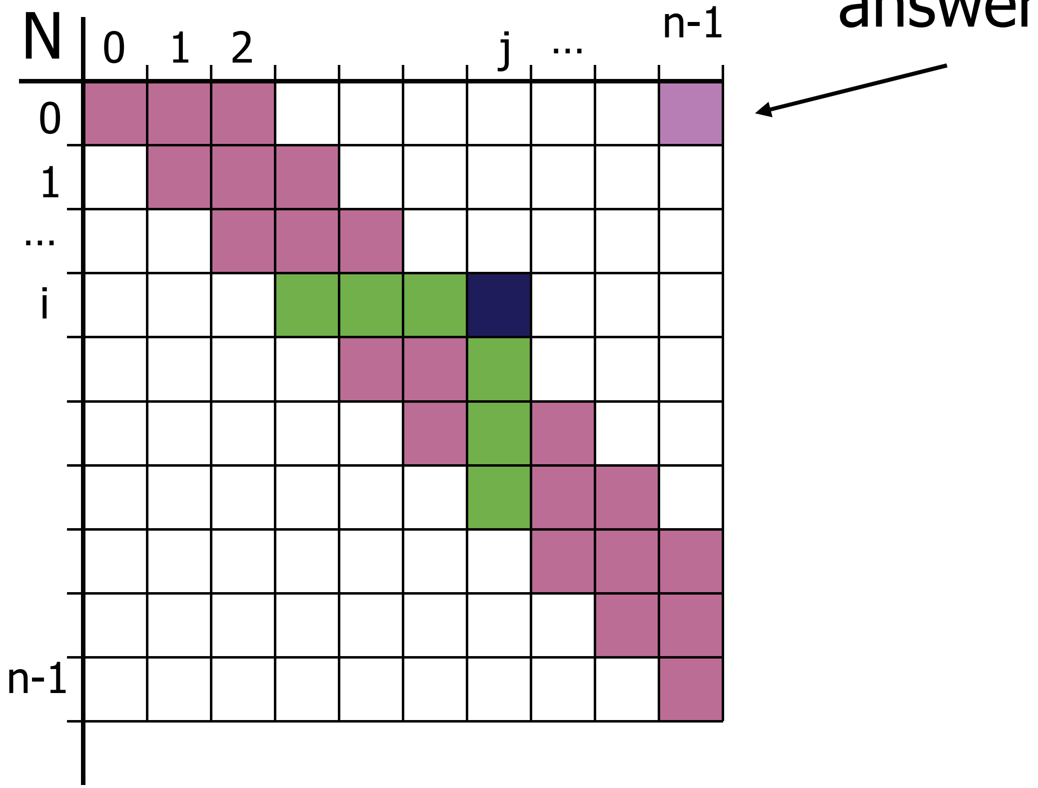
- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

```
Algorithm matrixChain(S):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal
        paranethization of S
    for i ← 1 to n-1 do
        N_{i,i} ← 0
    for b ← 1 to n-1 do
        for i ← 0 to n-b-1 do
            j ← i+b
            N_{i,j} ← +infinity
            for k ← i to j-1 do
                N_{i,j} ← min{N_{i,j} , N_{i,k} +N_{k+1,j} +d_i d_{k+1} d_{j+1}}
```

A Dynamic Programming Algorithm Visualization

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

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



- Let m(i, j) denote the minimum cost for computing $A_i \times A_{i+1} \times \ldots \times A_j$

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

- Computation sequence :

$$\begin{tikzcd}
 & & m(1,4) & & \\
 & m(1,3) \arrow[ur] & & m(2,4) \arrow[ul] & \\
m(1,2) \arrow[ur] \arrow[uurr, bend left] & & m(2,3) \arrow[ul] \arrow[ur] & & m(3,4) \arrow[ul] \arrow[uull, bend right]
\end{tikzcd}$$

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

- Matrix Chain的遞迴式

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{M[i,k] + M[k+1,j] + d_{i-1} d_k d_j\} & \text{if } i < j \end{cases}$$

- 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:

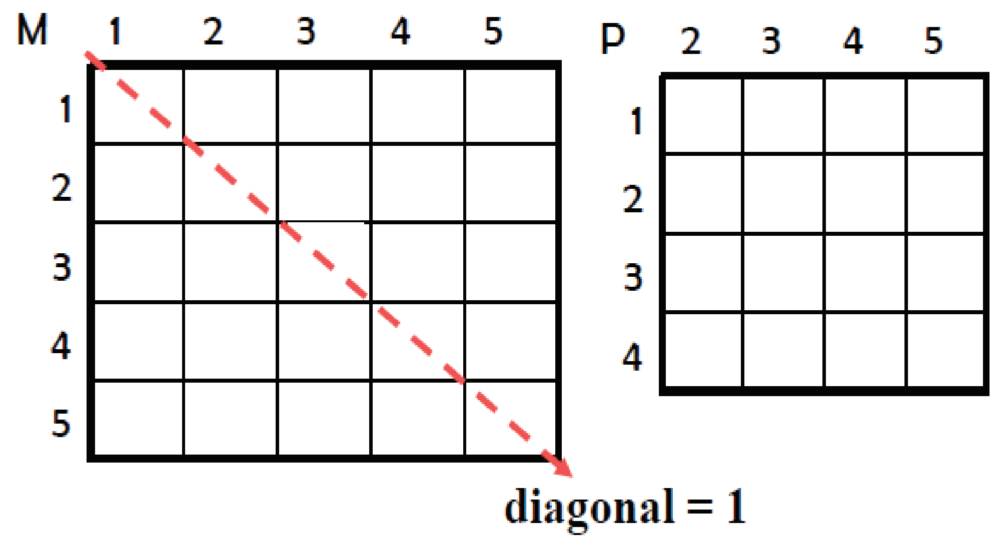
建立兩陣列 M[1…5, 1…5]及P[1…4, 2…5]

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o，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:

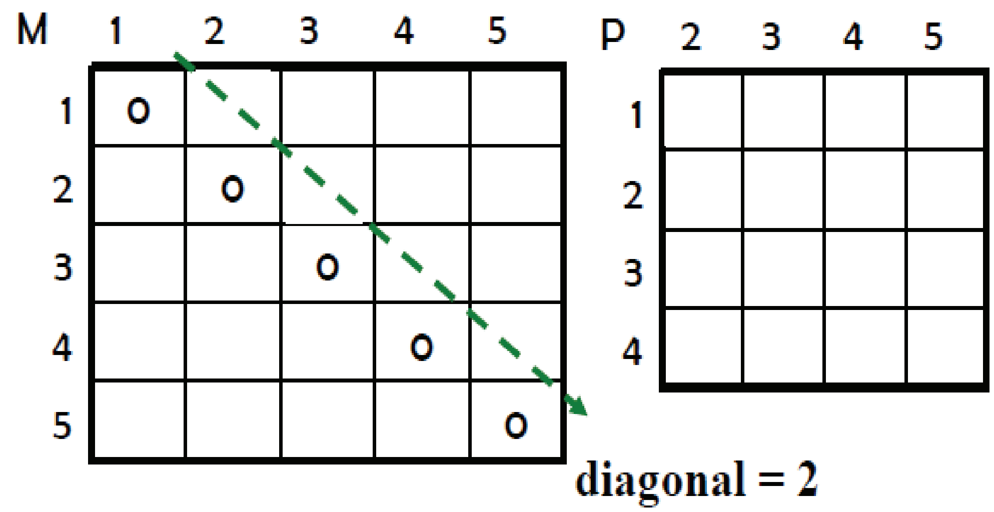
M[1, 2] = M[1,1]+M[2,2]+3×3×7 = 63，

其中 A^1 及A^2的分割點 k 如下:

$$A^1 \times A^2$$

$$\uparrow$$

分割點k = 1



Case ② (When diagonal > 1)

- 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:

$$M[2,4] = \min\begin{cases} M[2,2]+M[3,4]+3\times 7\times 9 = 315, & \text{分割點}k = 2 \\ M[2,3]+M[4,4]+3\times 2\times 9 = 96, & \text{分割點}k = 3 \end{cases}$$

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)

- diagonal = 4，有4個矩陣
- 當 i = 1及 j = 4，爲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)

- diagonal = 5，有5個矩陣
- 當 i = 1及 j = 5，爲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

$$M[1,5] = \min\begin{cases} M[1,1]+M[2,5]+3\times 3\times 4 = 174, & \text{分割點}k=1 \\ M[1,2]+M[3,5]+3\times 7\times 4 = 275, & \text{分割點}k=2 \\ M[1,3]+M[4,5]+3\times 2\times 4 = 156, & \text{分割點}k=3 \\ M[1,4]+M[5,5]+3\times 9\times 4 = 222, & \text{分割點}k=4 \end{cases}$$

◆ [Note]此演算法的概念如下:

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

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

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

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

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

All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.
- We can achieve $O(n^3)$ time using dynamic programming (similar to the Floyd-Warshall algorithm).

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all vertex pairs (i,j)
   if i = j
     D_0[i,i] ← 0
   else if (i,j) is an edge in G
     D_0[i,j] ← weight of edge (i,j)
   else
     D_0[i,j] ← + ∞
 for k ← 1 to n do
   for i ← 1 to n do
     for j ← 1 to n do
       D_k[i,j] ← min{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]}
 return D_n
```