Dynamic Programming

A sketch of a dynamic programming algorithm :

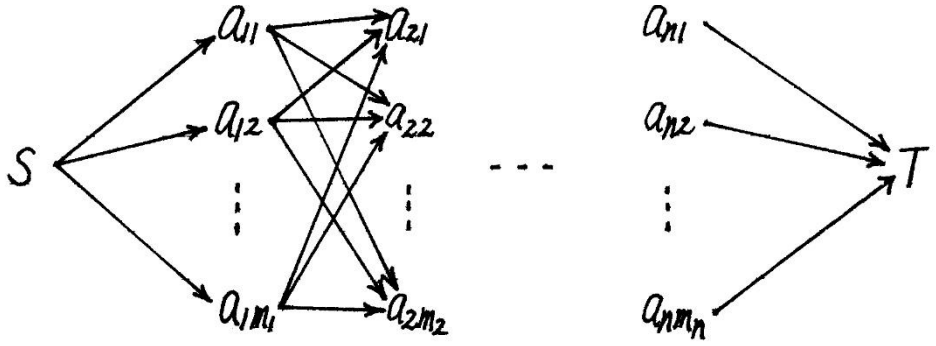
-
-
-

statements using the principle of optimality

-
-
-

• Principle of Optimality

Consider a multistage graph



If $S \to a_{1r_1} \to a_{2r_2} \to \ldots \to a_{nr_n} \to T$ **is the shortest (longest) path between** S **and** T, **then for any** $1 \le i < j \le n$, **the subpath**

$$a_{ir_i} \to a_{(i+1)r_{i+1}} \to \ldots \to a_{jr_j}$$

is the shortest (longest) path between a_{ir_i} **and** a_{jr_j}.

In general, the principle of optimality can be stated as follows.

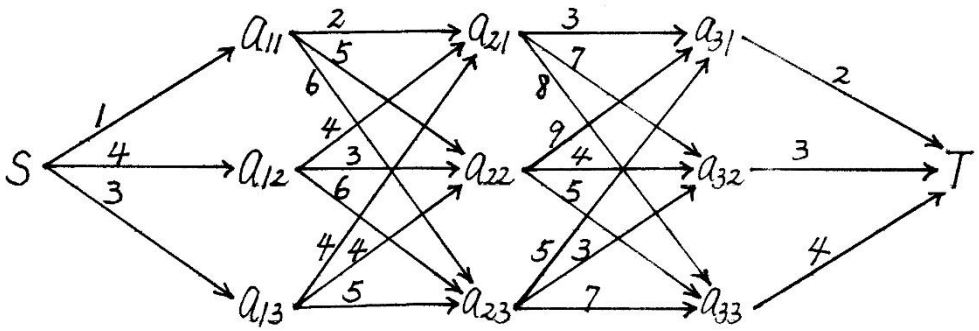
> **“Any subpolicy of an optimum policy must itself be an optimum policy with regard to the initial and terminal states of the subpolicy.”**

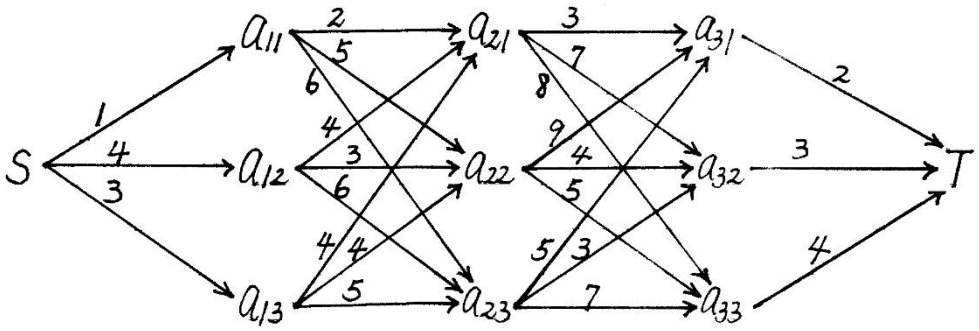
After determining an optimum policy for some initial state S_i and some terminal state S_t, we may ignore all non-optimum policies for (S_i, S_t) whenever we are looking for an optimum policy that includes (S_i, S_t) as a subpolicy.

For example, let l_i be the length of the shortest path from a_{1i} to T and w_i be the length of $S \rightarrow a_{1i}$.
If the shortest path from S to T is via a_{1i}, then it must include the shortest paths from S to a_{1i} and from a_{1i} to T.
Thus, the minimum length from S to T is $w_i + l_i$.

Based on the principle of optimality, we may devise some approaches to reducing the search space, while solving a problem by exhaustive search.
Such approaches are named *dynamic programming*.

For example, consider the following multistage graph. Three illustrative dynamic programming approaches are applied to compute the minimum length from *S* to *T*.





1. Denote by l_{ij} the minimum length from a_{ij} to T.

Step 1. Compute l_{31}, l_{32}, l_{33}.

$$(l_{31}, l_{32}, l_{33}) = (2, 3, 4) \qquad (a_{3i} \to T)$$

Step 2. Compute l_{21}, l_{22}, l_{23}.

$$l_{21} = \min\{3 + l_{31},\ 7 + l_{32},\ 8 + l_{33}\} = 5$$

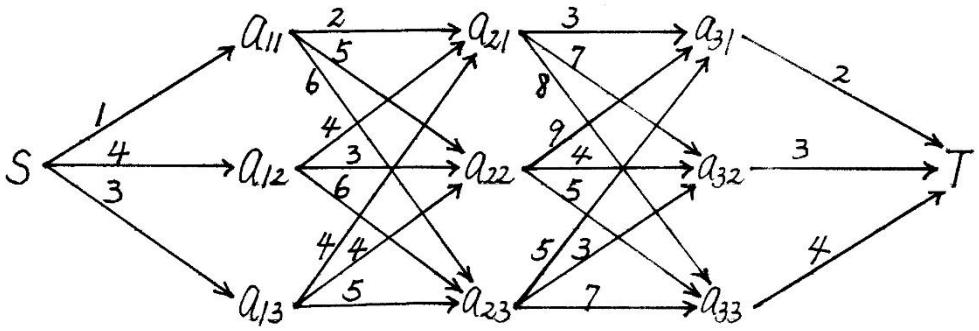
$$(a_{21} \to a_{31} \to T)$$

$$l_{22} = \min\{9 + l_{31},\ 4 + l_{32},\ 5 + l_{33}\} = 7$$

$$(a_{22} \to a_{32} \to T)$$

$$l_{23} = \min\{5 + l_{31},\ 3 + l_{32},\ 7 + l_{33}\} = 6$$

$$(a_{23} \to a_{32} \to T)$$



Step 3. Compute l_{11}, l_{12}, l_{13}.

$$l_{11} = \min\{2 + l_{21},\ 5 + l_{22},\ 6 + l_{23}\} = 7$$

$$(a_{11} \to a_{21} \to a_{31} \to T)$$

$$l_{12} = \min\{4 + l_{21},\ 3 + l_{22},\ 6 + l_{23}\} = 9$$

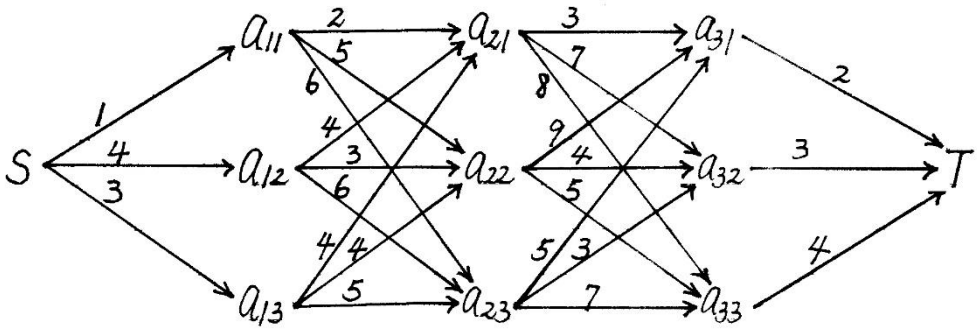
$$(a_{12} \to a_{21} \to a_{31} \to T)$$

$$l_{13} = \min\{4 + l_{21},\ 4 + l_{22},\ 5 + l_{33}\} = 9$$

$$(a_{13} \to a_{21} \to a_{31} \to T)$$

Step 4. The minimum length from S to T is equal to $\min\{1 + l_{11},\ 4 + l_{12},\ 3 + l_{13}\} = 8$

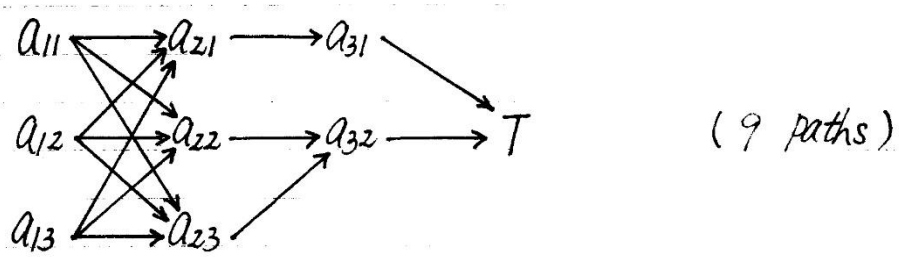
$$(S \to a_{11} \to a_{21} \to a_{31} \to T)$$

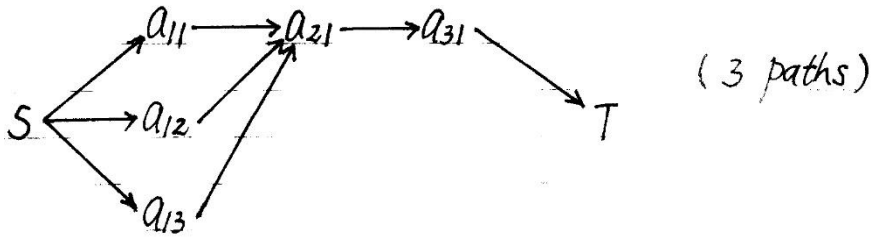


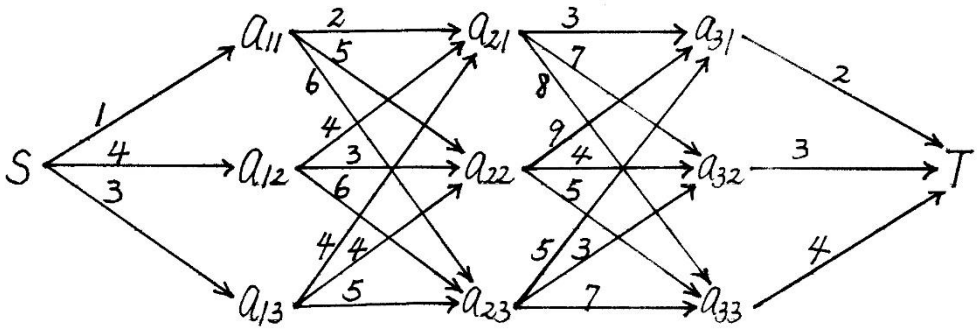
There are 24 paths computed above.

$a_{3i} \rightarrow T$ $i = 1, 2, 3$ (3 paths)

$a_{2j} \rightarrow a_{3i} \rightarrow T$ $i = 1, 2, 3$ and $j = 1, 2, 3$ (9 paths)







2. Denote by p_{ij} the minimum length from S to a_{ij}.

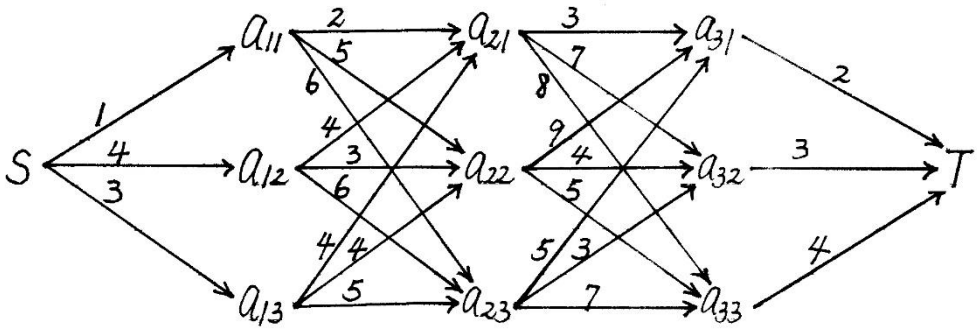
Step 1. Compute p_{1i} $i = 1, 2, 3$

Step 2. Compute p_{2i} $i = 1, 2, 3$

Step 3. Compute p_{3i} $i = 1, 2, 3$

Step 4. Compute the minimum length from S to T.

There are 24 paths computed.



3. Compute l_{ij} and p_{ij} in opposite directions.

Step 1. Compute p_{1i} $i = 1, 2, 3$

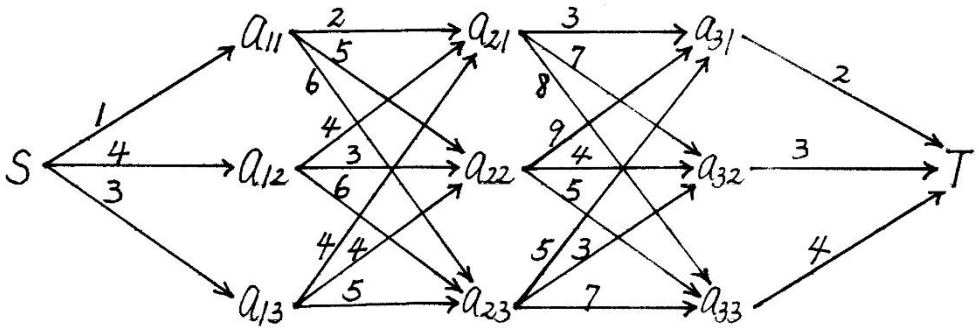
Step 2. Compute p_{2i} $i = 1, 2, 3$

Step 3. Compute l_{3i} $i = 1, 2, 3$

Step 4. Compute l_{2i} $i = 1, 2, 3$

Step 5. Compute the minimum length from *S* to *T*.

There are 27 paths computed.



If we compute the minimum length from *S* to *T* by exhaustive search, then there are 27 paths (each of length 4) computed.

In general, the numbers of paths computed are

(1) $m_1 + m_1m_2 + m_2m_3 + \ldots + m_{n-1}m_n + m_n$ **(approach 1)**

(2) $m_1 + m_1m_2 + m_2m_3 + \ldots + m_{n-1}m_n + m_n$ **(approach 2)**

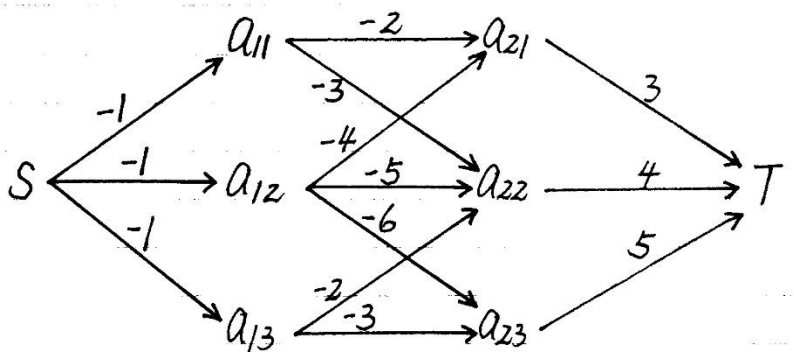
(3) $m_1 + m_1m_2 + m_2m_3 + \ldots + m_{n-1}m_n + m_n + m_{\lceil n/2 \rceil}$
(approach 3)

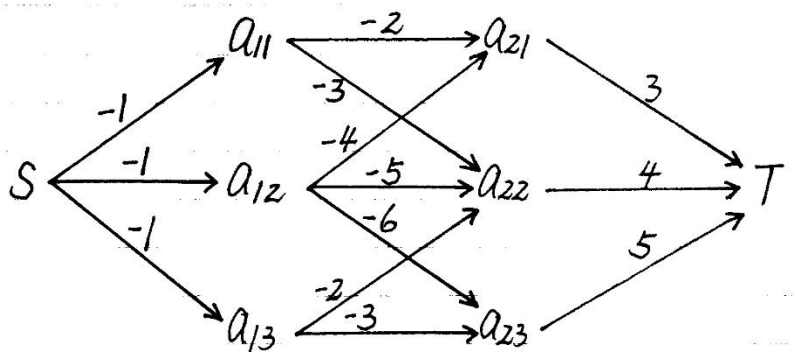
(4) $m_1m_2\ldots m_n$ **(exhaustive search)**

$\Rightarrow$ **Dynamic programming can reduce the search space considerably.**

If the principle of optimality does not hold, then dynamic programming may fail.

For example, if the lengths of the arcs are allowed to be negative and the length of a path is computed as the product of the arc lengths, instead of the sum of the arc lengths, then the minimum length cannot be computed by dynamic programming.





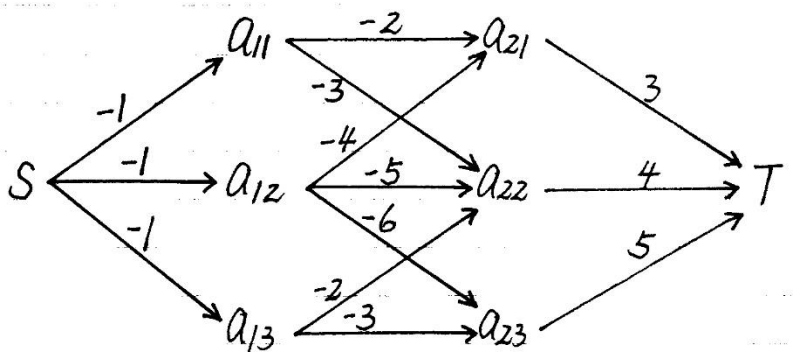
The shortest path between *S* and *T* is

$$S \to a_{11} \to a_{21} \to T,$$

but the shortest path between a_{11} and *T* is

$$a_{11} \to a_{22} \to T.$$

$\Rightarrow$ The principle of optimality does not hold.



Computing the minimum length between *S* and *T* by dynamic programming :

$(l_{21}, l_{22}, l_{23}) = (3, 4, 5)$;

$l_{11} = \min\{-2 * l_{21}, -3 * l_{22}\} = -12$;

$l_{12} = \min\{-4 * l_{21}, -5 * l_{22}, -6 * l_{23}\} = -30$;

$l_{13} = \min\{-2 * l_{22}, -3 * l_{23}\} = -15$;

$\text{minimum length} = \min\{-1 * l_{11}, -1 * l_{12}, -1 * l_{13}\}$

$= 12.$

The shortest path obtained is $S \rightarrow a_{11} \rightarrow a_{22} \rightarrow T$, which is incorrect.

Ex. The optimal parenthesization problem.

Given a string of n items, find an optimal parenthesization of the string.

A parenthesization of n items has $n-1$ pairs of parentheses.

Without losing generality, let n items be 1, 2, …, n.

The cost of a parenthesis pair is the sum of the enclosed numbers, and the cost of a parenthesization is the sum of the individual costs.

(((1 2) 3) 4) → 19
((1 2) (3 4)) → 20
(1 (2 (3 4))) → 26
((1 (2 3)) 4) → 21
(1 ((2 3) 4)) → 24

The optimal parenthesization is the one with the minimum cost.

If this problem is solved by exhaustive search, then the time required is exponential to *n*, for there is an exponential number of distinct parenthesizations.

A dynamic programming approach for this problem is proposed below.

The principle of optimality holds:

> **an optimal parenthesization for the whole string assures an optimal parenthesization for each of its substrings.**

⇒ **calculate optimal parenthesizatios of successively larger substrings of the original string, starting from the singleton items.**

Let $c(i, j)$ denote the minimum cost of parenthesizing the substring from the ith item to the jth item, $1 \leq i \leq j \leq n$.

When $i = j$, we have $c(i, j) = 0$.

Then, $c(1, n)$, which is the optimal value, can be computed by the following recurrence equation,

$$c(i, j) = \min_{i \leq k < j} \{c(i, k) + c(k+1, j) + \sum_{l=i}^{j} v_l\},$$

where v_l denotes the value of the lth item.

$j - i = 0$: $c(1, 1) = c(2, 2) = c(3, 3) = c(4, 4) = 0.$

$j - i = 1$: $c(1, 2) = 3, \quad c(2, 3) = 5, \quad c(3, 4) = 7.$

$j - i = 2$:

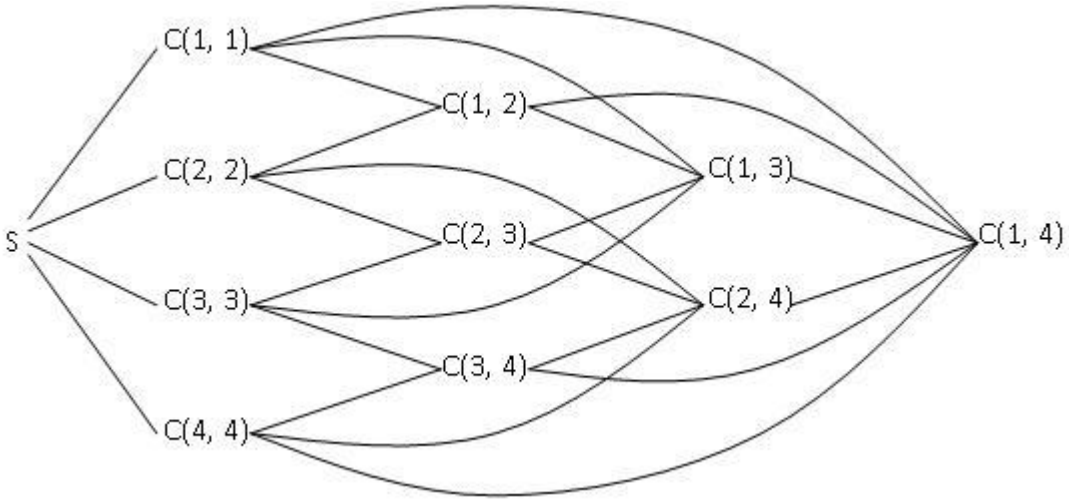
(i, j)	$c(i, k) + c(k+1, j) + \sum v_l$	$c(i, j)$	k^*
(1, 3)	$0 + 5 + 6 = 11$		
	$3 + 0 + 6 = 9$	9	2
(2, 4)	$0 + 7 + 9 = 16$		
	$5 + 0 + 9 = 14$	14	3

$j - i = 3$:

(i, j)	$c(i, k) + c(k+1, j) + \sum v_l$	$c(i, j)$	k^*
(1, 4)	$0 + 14 + 10 = 24$		
	$3 + 7 + 10 = 20$		
	$9 + 0 + 10 = 19$	19	3

The optimal value is 19, and the optimal parenthesization, which is (((1 2) 3) 4), can be obtained by the aid of k^*.

Corresponding multistage graph :



Time complexity :

Let $m = j - i$.

It takes $O(m(n-m))$ time to compute all $c(i, j)$'s with $j - i = m$.

Hence, the total time required is

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

Ex. The longest common subsequence (LCS) problem

Given two strings $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$, find their longest common subsequence.

$A = abadc$, $B = adcb$

$\Rightarrow$ **common subsequences of A and B :**

a, b, c, d, ab, ac, ad, dc, adc

$\Rightarrow$ **the longest common subsequence of A and B :**

adc

Let $L(i, j)$ denote the length of an LCS of $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$, where $1 \le i \le n$ and $1 \le j \le m$.

$L(n, m)$ is the length of an LCS of A and B.

The principle of optimality holds.

(1) a_i and b_j are included in an LCS ($\Rightarrow a_i = b_j$) :

$$L(i, j) = L(i-1, j-1) + 1.$$

(2) a_i is not included in any LCS, but b_j is included in an LCS ($\Rightarrow a_i \neq b_j$) :

$$L(i, j) = L(i-1, j).$$

(3) b_j is not included in any LCS, but a_i is included in an LCS ($\Rightarrow a_i \neq b_j$) :

$$L(i, j) = L(i, j-1).$$

(4) Neither a_i nor b_j is included in any LCS ($\Rightarrow a_i \neq b_j$) :

$$L(i, j) = L(i-1, j-1).$$

$$\Rightarrow \quad L(i, j) = \max\{L(i-1, j), L(i, j-1), l_{ij} + L(i-1, j-1)\},$$

where $l_{ij} = 1$ if $a_i = b_j$, and $l_{ij} = 0$ if $a_i \neq b_j$.

Initially, $L(0, j) = 0$, $L(i, 0) = 0$, and $L(0, 0) = 0$.

For example, assume $A = abadc$ and $B = adcb$.

(i, j)	$L(i-1, j)$	$L(i, j-1)$	$l_{ij} + L(i-1, j-1)$	$L(i, j)$
(1, 1)	0	0	**1**	**1**
(1, 2)	0	1	0	1
(2, 1)	**1**	0	0	**1**
(1, 3)	0	1	0	1
(2, 2)	1	1	1	1
(3, 1)	**1**	0	**1**	**1**
(1, 4)	0	1	0	1
(2, 3)	1	1	1	1
(3, 2)	1	1	1	1
(4, 1)	1	0	0	1
(2, 4)	1	1	2	2
(3, 3)	1	1	1	1
(4, 2)	1	1	**2**	**2**
(5, 1)	1	0	0	1
(3, 4)	2	1	1	2
(4, 3)	1	2	1	2
(5, 2)	2	1	1	2
(4, 4)	2	2	1	2
(5, 3)	2	2	**3**	**3**
(5, 4)	2	**3**	2	**3**

$$L(5, 4) \rightarrow L(5, 3) \rightarrow L(4, 2) \rightarrow L(3, 1) \rightarrow L(2, 0)$$
$$(a_5 = b_3) \quad (a_4 = b_2) \quad (a_3 = b_1)$$

or

$$L(5, 4) \rightarrow L(5, 3) \rightarrow L(4, 2) \rightarrow L(3, 1) \rightarrow L(2, 1) \rightarrow$$
$$(a_5 = b_3) \quad (a_4 = b_2)$$

$$L(1, 1) \rightarrow L(0, 0)$$
$$(a_1 = b_1)$$

LCS : $(a_1, a_4, a_5) = (b_1, b_2, b_3)$ **or**

$(a_3, a_4, a_5) = (b_1, b_2, b_3)$

The time complexity is $O(nm)$**.**

Ex. The knapsack problem

$$\max \quad \sum_{i=1}^{n} p_i x_i \qquad (1)$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le M$$

$$x_i \ge 0 \quad \text{integer} \quad i = 1, 2, \ldots, n.$$

Let $f(k, g)$ be the maximal value of the objective function using only the first k $(1 \le k \le n)$ items with capacity limitation g $(0 \le g \le M)$, i.e.,

$$f(k, g) = \max \quad \sum_{i=1}^{k} p_i x_i \qquad (2)$$

$$\text{subject to} \quad \sum_{i=1}^{k} w_i x_i \le g$$

$$x_i \ge 0 \quad \text{integer} \quad \text{for } 1 \le i \le k.$$

$f(n, M)$ is the optimal value of (1).

The principle of optimality holds:

If $(x_1^*, x_2^*, \ldots, x_k^*)$ is an optimal solution to $f(k, g)$, then $(x_1^*, x_2^*, \ldots, x_{k-1}^*)$ is an optimal solution to $f(k-1, g-w_k x_k^*)$.

$$
\begin{aligned}
f(k, g) &= \max \sum_{i=1}^{k} p_i x_i \\
&\quad \text{subject to } \sum_{i=1}^{k} w_i x_i \le g \\
&\quad\quad x_i \ge 0 \text{ integer for } 1 \le i \le k \\
&\quad (\text{set } x_k = x_k^*) \\
&= p_k x_k^* + \max \sum_{i=1}^{k-1} p_i x_i \\
&\quad\quad \text{subject to } \sum_{i=1}^{k-1} w_i x_i \le g - w_k x_k^* \\
&\quad\quad\quad x_i \ge 0 \text{ integer for } 1 \le i \le k-1 \\
&= p_k x_k^* + f(k-1, g - w_k x_k^*) \\
&= p_k x_k^* + \sum_{i=1}^{k-1} p_i x_i^* .
\end{aligned}
$$

Dynamic programming 1 :

$$f(k, g) = \max\{p_k x_k + f(k-1, g - w_k x_k) \mid x_k = 0, 1, \ldots, \lfloor g/w_k \rfloor\}.$$

Time complexity :

$$T(n, M) = O\left(\sum_{k=1}^{n} \sum_{g=1}^{M} (1 + \lfloor g/w_k \rfloor)\right)$$

$$= O\left(nM + \frac{M(M+1)}{2} \sum_{k=1}^{n} \frac{1}{w_k}\right).$$

Dynamic Programming 2 :

Let $(x_1^*, x_2^*, \ldots, x_k^*)$ be an optimal solution to $f(k, g)$.

Case 1. $x_k^* = 0$.

$(x_1^*, x_2^*, \ldots, x_{k-1}^*)$ is an optimal solution to $f(k-1, g)$.

Case 2. $x_k^* \geq 1$.

$(x_1^*, x_2^*, \ldots, x_{k-1}^*, x_k^* - 1)$ is an optimal solution to $f(k, g - w_k)$.

$\Rightarrow \; f(k, g) = \max\{f(k-1, g), \; p_k + f(k, g - w_k)\}$.

Time complexity : $T(n, M) = O(nM)$.

Ex. The 0/1 knapsack problem

$$\max \quad \sum_{i=1}^{n} p_i x_i$$

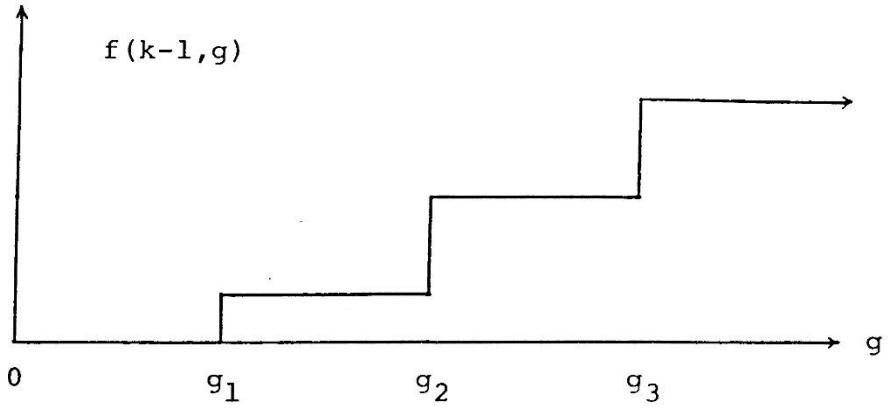
$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le M$$

$$x_i \in \{0, 1\} \quad i = 1, 2, \ldots, n.$$

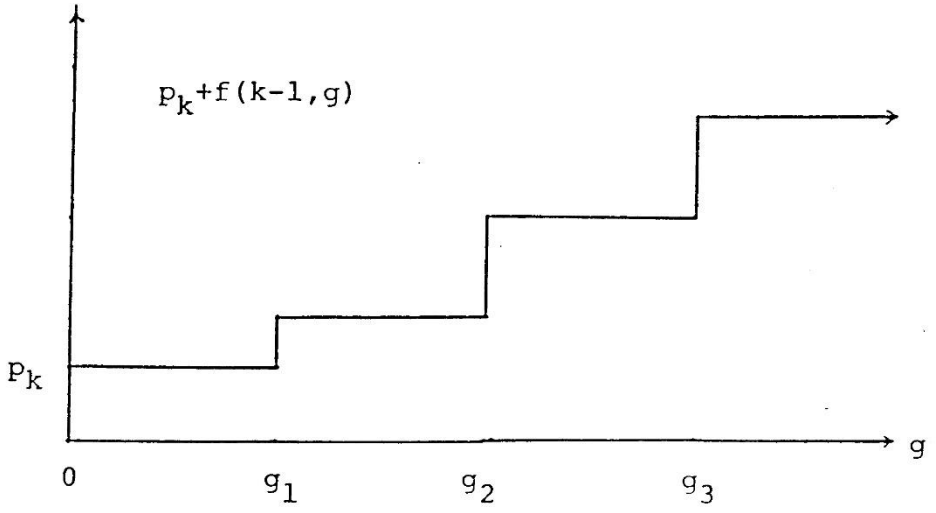
$$f(k, g) = \max\{f(k-1, g),\ p_k + f(k-1, g-w_k)\} \qquad (3)$$

Since $f(k, g)$ is a monotone nondecreasing step function for any fixed value of k, the function f can be evaluated for intervals $[a, b)$, instead of integers g, where f has the same value in $[a, b)$.

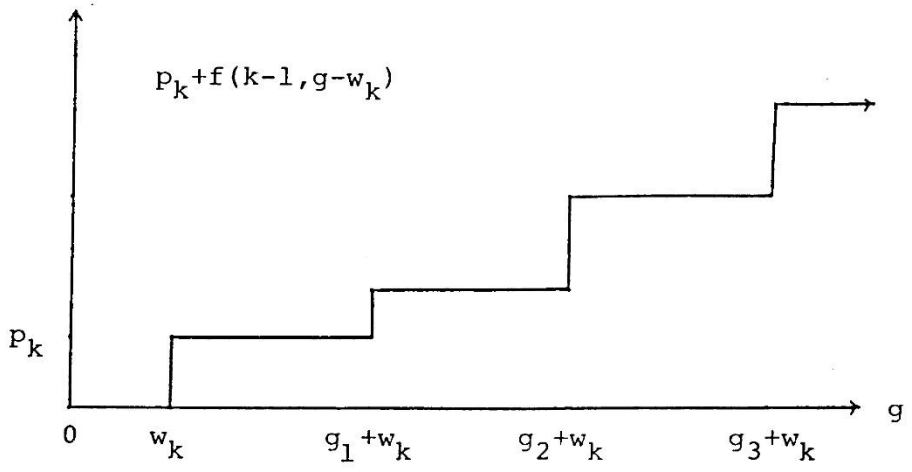
A geometric representation of computing $f(k, g)$ according to (3) is below.


f(k-1,g)
g
0
g_1
g_2
g_3

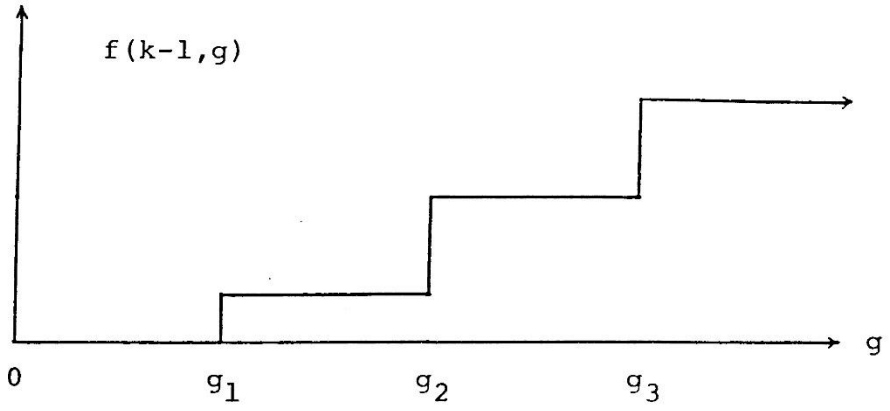
(a)


p_k+f(k-1,g)
p_k
g
0
g_1
g_2
g_3

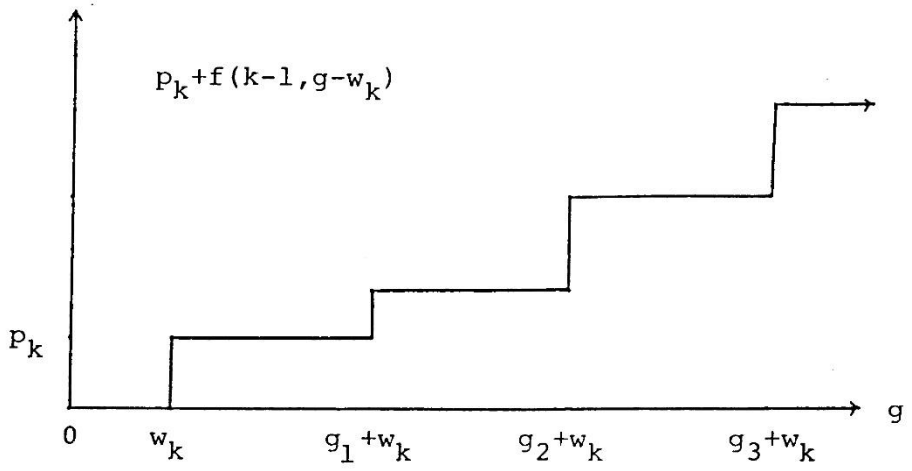
(b)


p_k+f(k-1,g-w_k)
p_k
g
0
w_k
g_1+w_k
g_2+w_k
g_3+w_k

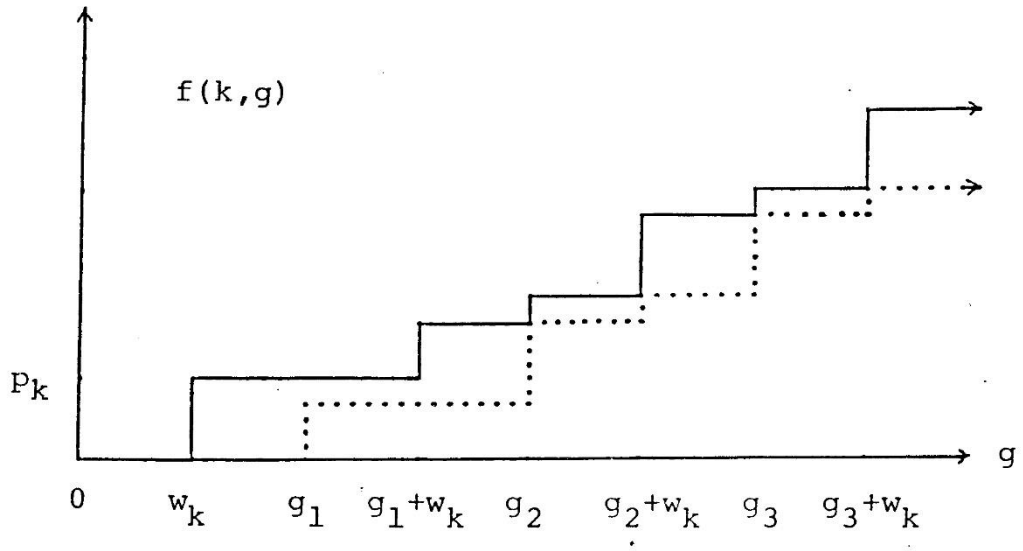
(c)


f(k-1,g)
g
0
g_1
g_2
g_3

(a)


$p_k+f(k-1,g-w_k)$
p_k
g
0
w_k
g_1+w_k
g_2+w_k
g_3+w_k

(c)


f(k,g)
p_k
g
0
w_k
g_1
g_1+w_k
g_2
g_2+w_k
g_3
g_3+w_k

(d)

Let g_i's be those values of g at which $f(k-1, g)$ changes value. The possible values of g at which $f(k, g)$ changes value are w_k, g_i, and $g_i + w_k$.

If x is one of those values and $f(k, g)$ does not change value at x, then x is called a *dominated value*. For example, g_1 is a dominated value.

Denote by S_k the set of values of g at which $f(k, g)$ changes value. Initially, $S_1 = \{0, w_1\}$.

For $1 < k \le n$, S_k can be determined as follows.

1. **$S_k \leftarrow S_{k-1} \cup \{w_k + d \mid d \in S_{k-1}$ and $w_k + d \le M\}$.**
2. **Remove dominated values from S_k.**

   (i) $x \in S_{k-1}$ is a dominated value if

   $$f(k-1, x) \le p_k + f(k-1, x - w_k).$$

   (ii) $x = w_k + d$ is a dominate value if

   $$f(k-1, x) \ge p_k + f(k-1, x - w_k).$$

   (If $x \in S_{k-1}$ and $x = w_k + d$, then x is not a dominated value.)

An illustrative example :

$$\begin{aligned} \text{Max} \quad & 10x_1 + 8x_2 + 5x_3 \\ \text{s.t.} \quad & 12x_1 + 9x_2 + 4x_3 \le 16 \\ & x_1, x_2, x_3 \in \{0, 1\} \end{aligned}$$

$S_1 = \{0, 12\}$

$[a, b)$	$f(0, [a, b))$	$p_1 + f(0, [a - w_1, b - w_1))$	$f(1, [a, b))$
$[0, 12)$	0	$10 + (-\infty)$	0
		$(f(k, g) = -\infty$, if $g < 0)$	
$[12, 17)$	0	$10 + 0$	10

$S_2 = S_1 \cup \{9\} = \{0, 9, 12\}$ (no dominated value)

$[a, b)$	$f(1, [a, b))$	$p_2 + f(1, [a - w_2, b - w_2))$	$f(2, [a, b))$
$[0, 9)$	0	$8 + (-\infty)$	0
$[9, 12)$	0	$8 + 0$	8
$[12, 17)$	10	$8 + 0$	10

($9\ (= 0 + 9)$ is not dominated, because $f(1, 9) = 0 < 8 = p_2 + f(1, 9 - w_2) = p_2 + f(1, 0)$;

$12 \in S_1$ is not dominated, because $f(1, 12) = 10 > 8 = p_2 + f(1, 12 - w_2) = p_2 + f(1, 3)$.)

$S_2 = \{0, 9, 12\}$

$[a, b)$	$f(1, [a, b))$	$p_2 + f(1, [a - w_2, b - w_2))$	$f(2, [a, b))$
$[0, 9)$	0	$8 + (-\infty)$	0
$[9, 12)$	0	$8 + 0$	8
$[12, 17)$	**10**	$8 + 0$	**10**

$S_3 = S_2 \cup \{4, 13, 16\} = \{0, 4, 9, 12, 13, 16\}$

(no dominated value)

$[a, b)$	$f(2, [a, b))$	$p_3 + f(2, [a - w_3, b - w_3))$	$f(3, [a, b))$
$[0, 4)$	0	$5 + (-\infty)$	0
$[4, 9)$	0	$5 + 0$	5
$[9, 12)$	8	$5 + 0$	8
$[12, 13)$	10	$5 + 0$	10
$[13, 16)$	10	$5 + 8$	13
$[16, 17)$	10	$5 + 10$	**15**

Optimal value : $f(3, 16) = 15$

Optimal solution : $(x_1^*, x_2^*, x_3^*) = (1, 0, 1)$

Ex. The traveling salesman problem

A salesman wishes to visit a number of cities and then return to his starting city. Given the distances between cities, in what order should he travel so as to visit every city precisely once (except the starting city), with the minimum mileage traveled ?

In terms of graph theory, the aim is to find a minimum-weight hamiltonian cycle in a weighted graph.

Without loss of generality, denote the cities by 1, 2, …, n and assume that the starting city is 1.

Let $L(i, S)$ be the length of a shortest tour starting at city i, where $i \notin S$, going through all cities in S exactly once, and ending at city 1.

The principle of optimality holds.

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

where d_{ij} is the distance from city i to city j.

$L(1, \{2, 3, \ldots, n\})$ is the optimal value.

An illustrative example :

Suppose that there are four cities and their distances are given by the following matrix.

from \ to	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$|S| = 0$ $\quad L(i, \varnothing) = d_{i1}$

$|S| = 1$	i	S	j	d_{ij}	$L(j, S-\{j\})$	$L(i, S)$
	2	{3}	3	9	6	15
	2	{4}	4	10	8	18
	3	{2}	2	13	5	18
	3	{4}	4	12	8	20
	4	{2}	2	8	5	13
	4	{3}	3	9	6	15

$\|S\| = 1$	i	S	j	d_{ij}	$L(j, S - \{j\})$	$L(i, S)$
	2	{3}	3	9	6	15
	2	{4}	4	10	8	18
	3	{2}	2	13	5	18
	3	{4}	4	12	8	20
	4	{2}	2	8	5	13
	4	{3}	3	9	6	15

$\|S\| = 2$	i	S	j	d_{ij}	$L(j, S - \{j\})$	$L(i, S)$
	2	{3, 4}	3	9	20	
			4	10	15	25
	3	{2, 4}	2	13	18	
			4	12	13	25
	4	{2, 3}	2	8	15	23
			3	9	18	

$\|S\| = 2$	i	S	j	d_{ij}	$L(j, S - \{j\})$	$L(i, S)$
	2	{3, 4}	3	9	20	
			4	10	15	25
	3	{2, 4}	2	13	18	
			4	12	13	25
	4	{2, 3}	2	8	15	23
			3	9	18	

$\|S\| = 3$	i	S	j	d_{ij}	$L(j, S - \{j\})$	$L(i, S)$
	1	{2, 3, 4}	2	10	25	35
			3	15	25	
			4	20	23	

Optimal value : 35

Optimal solution : $1 \to 2 \to 4 \to 3 \to 1$

Ex. Maximum weight independent set in a tree

Suppose that *T* is a tree whose each vertex *i* is associated with a positive weight *w*(*i*). The problem is to find an independent set of *T* whose weight is maximum.

A vertex subset *I* of *T* is an *independent set* if no two vertices of *I* are adjacent.

The weight of *I* is $\sum_{i \in I} w(i)$.

The problem is NP-hard if it is defined on a general graph.

Let T_i denote the subtree of T rooted at the vertex i.

Define

$$M(i) = \max\{w(I) \mid I \text{ is an independent set of } T_i \text{ and } i \in I\}$$

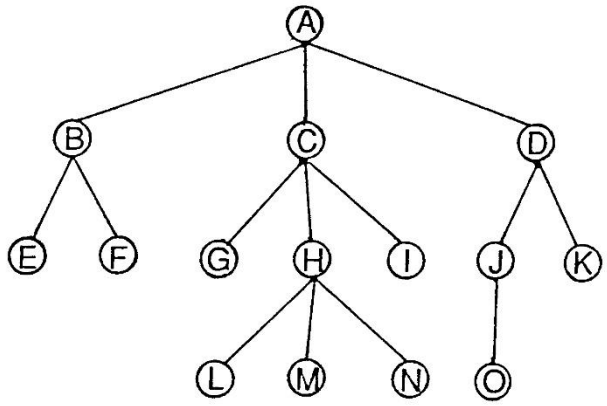
$$M'(i) = \max\{w(I) \mid I \text{ is an independent set of } T_i \text{ and } i \notin I\}$$

$M(i)$ ($M'(i)$) is the maximum weight of the independent sets in T_i that contain i (do not contain) i.

$$M(i) = w(i) + \sum_{j \text{ is a child of } i} M'(j)$$

$$M'(i) = \sum_{j \text{ is a child of } i} \max\{M(j), M'(j)\}$$

$M(i) = w(i)$ and $M'(i) = 0$ for each leaf vertex i.



i	$w(i)$	$M(i)$	$M'(i)$
A	6	53	50
B	4	4	11
C	8	23	20
D	8	10	16
E	5	5	0
F	6	6	0
G	2	2	0
H	8	8	15
I	3	3	0
J	9	9	2
K	7	7	0
L	5	5	0
M	4	4	0
N	6	6	0
O	2	2	0

The maximum weight independent set is {A, E, F, G, L, M, N, I, J, K}.

References for advanced dynamic programming algorithms.

(You can find more skillful dynamic programming algorithms in the following papers.)

(1) Knuth, "Optimal binary search trees," *Acta Informatica*, vol. 1, 1971, 14-25.

(2) Yao, "Efficient dynamic programming using quadrangle inequalities," *Proc. the ACM Symp. on the Theory of Computing*, 429-435. 1980.

(3) Choi and Narahari, "Algorithms for mapping and partitioning chain structured parallel computations," *Proc. the Int'l Conf. on Parallel Processing*, vol. 1, 625-628, 1991.

(4) Hsu and Du, "New algorithms for the LCS problem," *Journal of Computer and System Sciences*, vol. 29, 1984, 133-152.

(5) Wang, Chen, and Park, "On the set LCS and set-set LCS problems," *Journal of Algorithms*, vol. 14, no.3, 1993, 466-477.

Program Assignment 1 :

Write an executable non-recursive program to report all longest common subsequences of two strings (and all LCS locations in the two strings).

Exercise 1 :

Wagner, "The string-to-string correction problem,"

***Journal of the ACM*, vol. 21, no. 1, 1974, pp. 168-173.**