Dynamic Programming

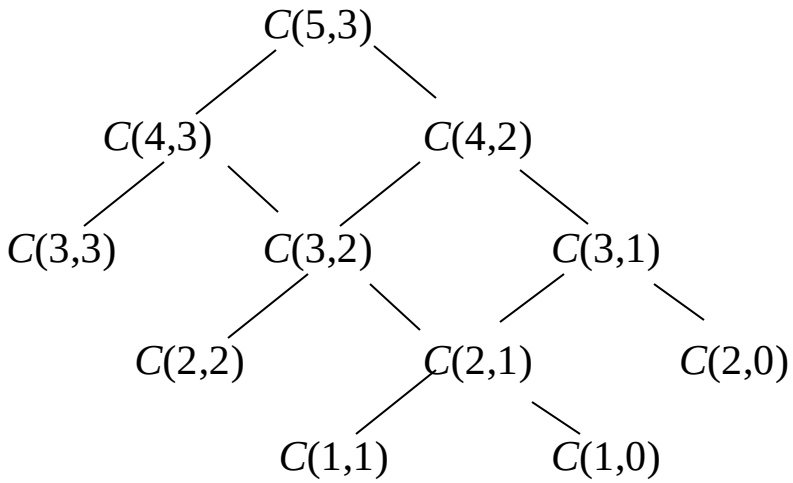
One technique that attempts to solve problems by dividing them into subproblems is called dynamic programming. It uses a "bottom-up" approach in that the subproblems are arranged and solved in a systematic fashion, which leads to a solution to the original problem. This bottom-up approach implementation is more efficient than a "top-down" counterpart mainly because duplicated computation of the same problems is eliminated. This technique is typically applied to solving optimization problems, although it is not limited to only optimization problems.

Dynamic programming typically involves two steps: (1) develop a recursive strategy for solving the problem; and (2) develop a "bottom-up" implementation without recursion.

Example: Compute the binomial coefficients $C(n, k)$ defined by the following recursive formula:

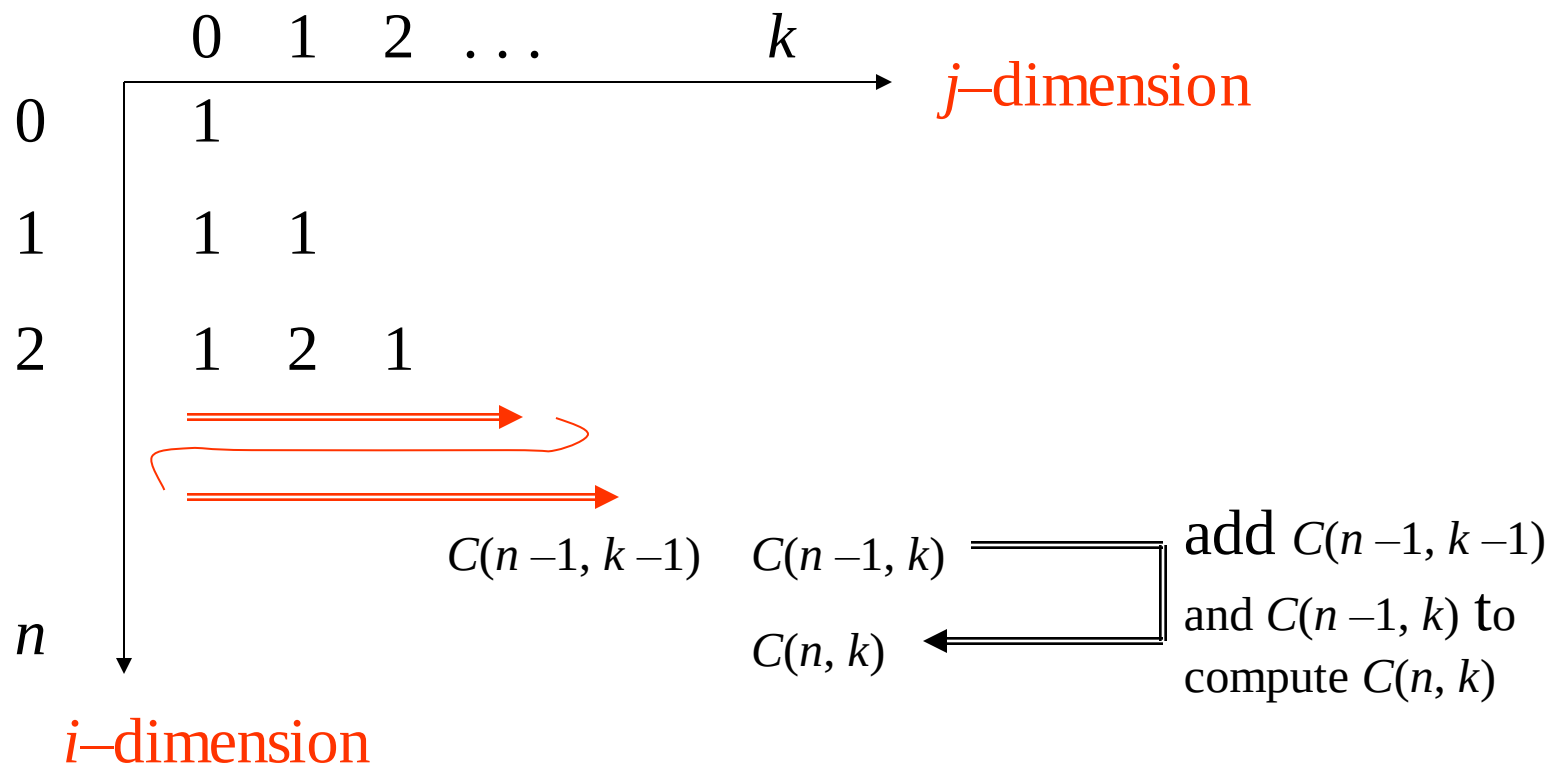
$$C(n,k) = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n; \\ C(n-1,k) + C(n-1,k-1), & \text{if } 0 < k < n; \\ 0, & \text{otherwise.} \end{cases}$$

The following "call tree" demonstrates repeated (duplicated) computations in a straightforward recursive implementation:



Notice repeated calls to $C(3,2)$ and to $C(2,1)$. In general, the number of calls for computing $C(n, k)$ is $2C(n, k) - 1$, which can be exponentially large.

A more efficient way to compute $C(n, k)$ is to organize the computation steps of $C(i, j)$, $0 \le i \le n$ and $0 \le j \le k$, in a tabular format and compute the values by rows (the i–dimension) and within the same row by columns (the j–dimension):



It can be seen that the number of steps (add operation) is O(*nk*) in computing *C*(*n*, *k*), using O(*nk*) amount of space (I.e., the table). In fact, since only the previous row of values are needed in computing the next row of the table, space for only two rows is needed reducing the space complexity to O(*k*). The following table demonstrates the computation steps for calculating *C*(5,3):

	0	1	2	3	
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	
5	1	5	10	10	= *C*(5,3)

Note that this table shows Pascal's triangle in computing the binomial coefficients.

Example: Solve the make-change problem using dynamic programming. Suppose there are n types of coin denominations, d_1, d_2, …, and d_n. (We may assume one of them is penny.) There are an infinite supply of coins of each type. To make change for an arbitrary amount j using the minimum number of coins, we first apply the following recursive idea:

If there are only pennies, the problem is simple: simply use j pennies to make change for the total amount j. More generally, if there are coin types 1 through i, let $C[i, j]$ stands for the minimum number of coins for making change of amount j. By considering coin denomination i, there are two cases: either we use at least one coin denomination i, or we don't use coin type i at all.

In the first case, the total number of coins must be $1+ C[i, j–d_i]$ because the total amount is reduced to $j – d_i$ after using one coin of amount d_i, the rest of coin selection from the solution of $C[i, j]$ must be an optimal solution to the reduced problem with reduced amount, still using coin types 1 through i.

In the second case, i.e., suppose no coins of denomination i will be used in an optimal solution. Thus, the best solution is identical to solving the same problem with the total amount j but using coin types 1 through $i – 1$, i.e. $C[i –1 , j]$. Therefore, the overall best solution must be the better of the two alternatives, resulting in the following recurrence:

$$C[i, j] = \min (1 + C[i, j – d_i], C[i –1 , j])$$

The boundary conditions are when $i \leq 0$ or when $j < 0$ (in which case let $C[i, j] = \infty$), and when $j = 0$ (let $C[i, j] = 0$).

Example: There are 3 coin denominations $d_1 = 1$, $d_2 = 4$, and $d_3 = 6$, and the total amount to make change for is $K = 8$. The following table shows how to compute $C[3,8]$, using the recurrence as a basis but arranging the computation steps in a tabular form (by rows and within the row by columns):

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

$C[2, 8]$

Boundary condition for amount $j = 0$

$C[3, 8 - 6]$

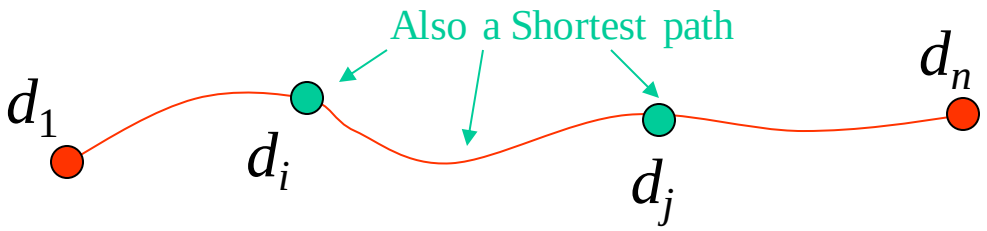
$C[3, 8] = \min(1 + C[3, 8 - 6], C[2, 8])$

Note the time complexity for computing $C[n, K]$ is $O(nK)$, using space $O(K)$ by maintaining last two rows.

The Principle of Optimality:

In solving optimization problems which require making a sequence of decisions, such as the change making problem, we often apply the following principle in setting up a recursive algorithm: Suppose an optimal solution made decisions d_1, d_2, and …, d_n. The subproblem starting after decision point d_i and ending at decision point d_j, also has been solved with an optimal solution made up of the decisions d_i through d_j. That is, any subsequence of an optimal solution constitutes an optimal sequence of decisions for the corresponding subproblem. This is known as the principle of optimality which can be illustrated by the shortest paths in weighted graphs as follows:

A shortest path from d_1 to d_n



The 0–1 Knapsack Problem:

Given n objects 1 through n, each object i has an integer weight w_i and a real number value v_i, for $1 \leq i \leq n$. There is a knapsack with a total integer capacity W. The 0–1 knapsack problem attempts to fill the sack with these objects within the weight capacity W while maximizing the total value of the objects included in the sack, where an object is totally included in the sack or no portion of it is in at all. That is, solve the following optimization problem with $x_i = 0$ or 1, for $1 \leq i \leq n$:

$$\text{Maximize } \sum_{i=1}^{n} x_i v_i \text{ subject to } \sum_{i=1}^{n} x_i w_i \leq W.$$

To solve the problem using dynamic programming, we first define a notation (expression) and derive a recurrence for it.

Let $V[i, j]$ denote the maximum value of the objects that fit in the knapsack, selecting objects from 1 through i with the sack's weight capacity equal to j. To find $V[i, j]$ we have two choices concerning the decisions made on object i (in the optimal solution for $V[i, j]$): We can either ignore object i or we can include object i. In the former case, the optimal solution of $V[i, j]$ is identical to the optimal solution to using objects 1 though $i - 1$ with sack's capacity equal to W (by the definition of the V notation) . In the latter case, the parts of the optimal solution of $V[i, j]$ concerning the choices made to objects 1 through $i - 1$, must be an optimal solution to $V[i - 1, j - w_i]$, an application of the principle of optimality Thus, we have derived the following recurrence for $V[i, j]$:

$$V[i, j] = \max(V[i - 1, j], v_i + V[i - 1, j - w_i])$$

The boundary conditions are $V[0, j] = 0$ if $j \geq 0$, and $V[i, j] = -\infty$ when $j < 0$.

The problem can be solved using dynamic programming (i.e., a bottom-up approach to carrying out the computation steps) based on a tabular form when the weights are integers.

Example: There are $n = 5$ objects with integer weights $w[1..5] = \{1,2,5,6,7\}$, and values $v[1..5] = \{1,6,18,22,28\}$. The following table shows the computations leading to $V[5,11]$ (i.e., assuming a knapsack capacity of 11).

Time: O(nW)
space: O(W)

Sack's capacity		0	1	2	3	4	5	6	7	8	9	10	11
wi	*vi*												
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	6	7	7	18	19	24	25	25	25	25
6	22	0	1	6	7	7	18	22	24	28	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	40

$V[3, 8]$

$V[4, 8]$ =max ($V[3, 8]$, 22 + $V[3, 2]$)

$V[3,8 - w_4] = V[3, 2]$

All-Pairs Shortest Paths Problem:

Given a weighted, directed graph represented in its weight matrix form $W[1..n][1..n]$, where n = the number of nodes, and $W[i][j]$ = the edge weight of edge (i, j). The problem is find a shortest path between every pair of the nodes. We first note that the principle of optimality applies:

If node k is on a shortest path from node i to node j, then the subpath from i to k, and the subpath from k to j, are also shortest paths for the corresponding end nodes.

Therefore, the problem of finding shortest paths for all pairs of nodes becomes developing a strategy to compute these shortest paths in a systematic fashion.

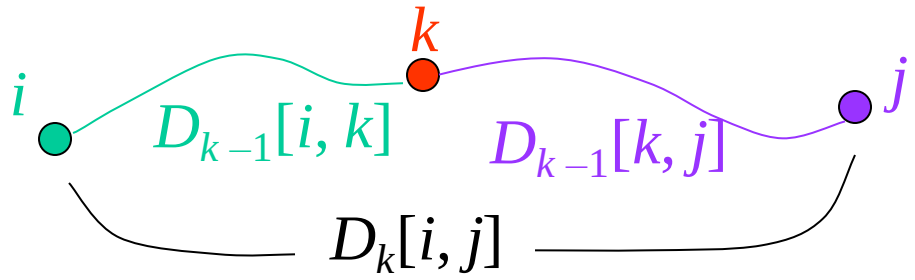
Floyd's Algorithm: Define the notation $D_k[i, j]$, $1 \le i, j \le n$, and $0 \le k \le n$, that stands for the shortest distance (via a shortest path) from node i to node j, passing through nodes whose number (label) is $\le k$. Thus, when $k = 0$, we have

$$D_0[i, j] = W[i][j] = \text{the edge weight from node } i \text{ to node } j$$

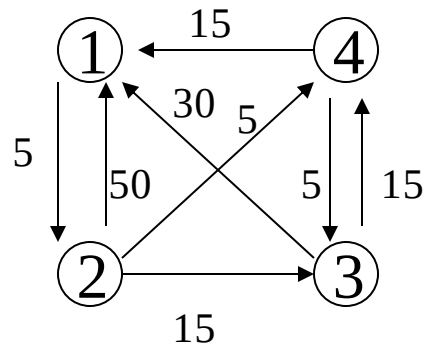
This is because no nodes are numbered ≤ 0 (the nodes are numbered 1 through n). In general, when $k \ge 1$,

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

The reason for this recurrence is that when computing $D_k[i, j]$, this shortest path either doesn't go through node k, or it passes through node k exactly once. The former case yields the value $D_{k-1}[i, j]$; the latter case can be illustrated as follows:



Example: We demonstrate Floyd's algorithm for computing $D_k[i, j]$ for $k = 0$ through $k = 4$, for the following weighted directed graph:



$$D_0 = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

reduced from ∞ because the path (3,1,2) going thru node 1 is possible in D_1

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Implementation of Floyd's Algorithm:

Input: The weight matrix $W[1..n][1..n]$ for a weighted directed graph, nodes are labeled 1 through n.

Output: The shortest distances between all pairs of the nodes, expressed in an $n \times n$ matrix.

Algorithm:

```
Create a matrix D and initialize it to W.
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

Note that one single matrix D is used to store D_{k-1} and D_k, i.e., updating from D_{k-1} to D_k is done immediately. This causes no problems because in the kth iteration, the value of $D_k[i, k]$ should be the same as it was in $D_{k-1}[i, k]$; similarly for the value of $D_k[k, j]$. The time complexity of the above algorithm is $O(n^3)$ because of the triple-nested loop; the space complexity is $O(n^2)$ because only one matrix is used.

The Partition Problem:

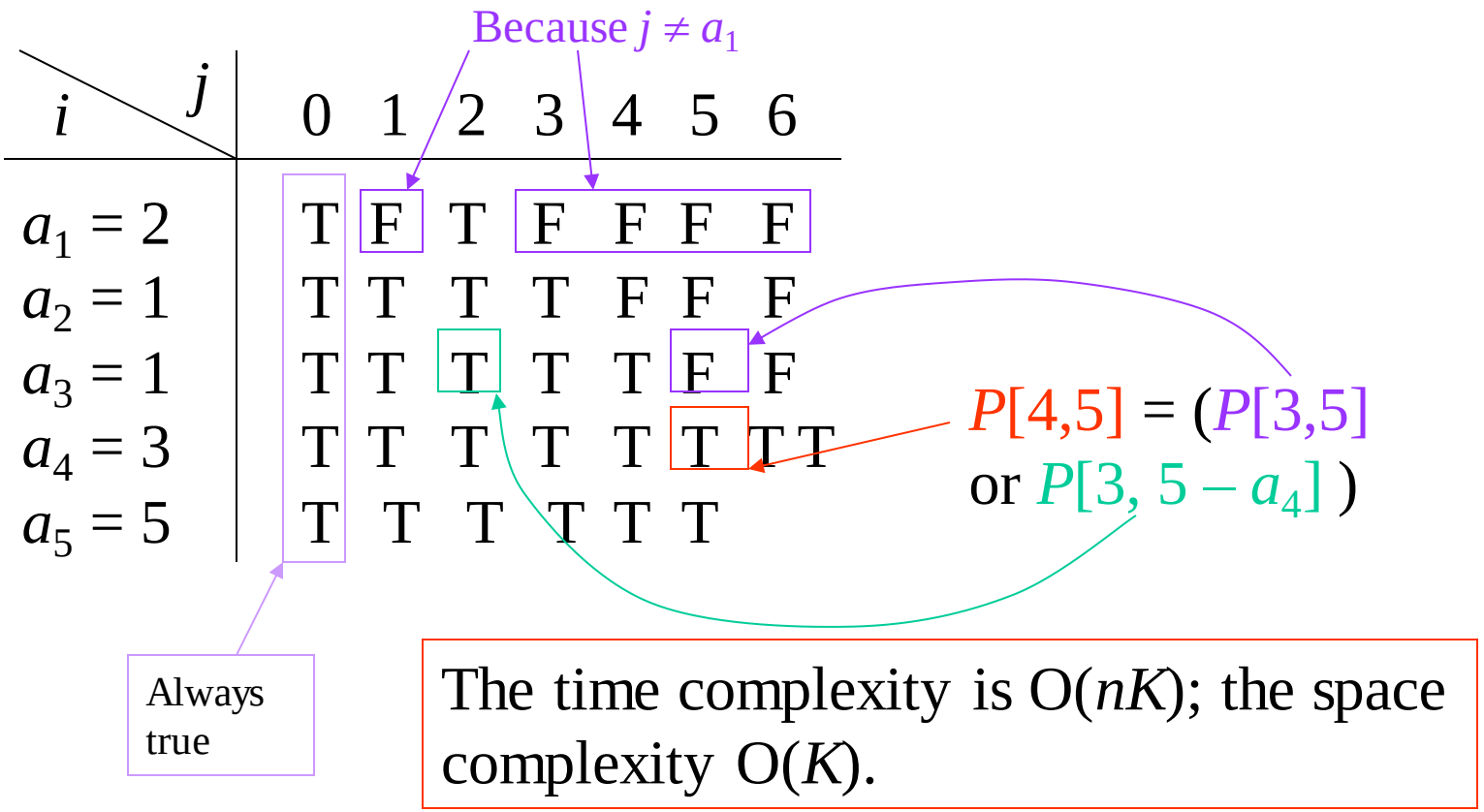
Given a set of positive integers, $A = \{a_1, a_2, \ldots, a_n\}$. The question is to select a subset B of A such that the sum of the numbers in B equals the sum of the numbers not in B, i.e., $\sum_{a_i \in B} a_i = \sum_{a_j \in A-B} a_j$. We may assume that the sum of all numbers in A is $2K$, an even number. We now propose a dynamic programming solution. For $1 \le i \le n$ and $0 \le j \le K$,

define $P[i, j]$ = True if there exists a subset of the first i numbers a_1 through a_i whose sum equals j; False otherwise.

Thus, $P[i, j]$ = True if either $j = 0$ or if ($i = 1$ and $j = a_1$). When $i > 1$, we have the following recurrence:

$P[i, j] = P[i - 1, j]$ or ($P[i - 1, j - a_i]$ if $j - a_i \ge 0$) That is, in order for $P[i, j]$ to be true, either there exists a subset of the first $i - 1$ numbers whose sum equals j, or whose sum equals $j - a_i$ (this latter case would use the solution of $P[i - 1, j - a_i]$ and add the number a_i. The value $P[n, K]$ is the answer.

Example: Suppose $A = \{2, 1, 1, 3, 5\}$ contains 5 positive integers. The sum of these number is 2+1+1+3+5=12, an even number. The partition problem computes the truth value of $P[5, 6]$ using a tabular approach as follows:



The time complexity is O(nK); the space complexity O(K).