

Problem solving with dynamic programming

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

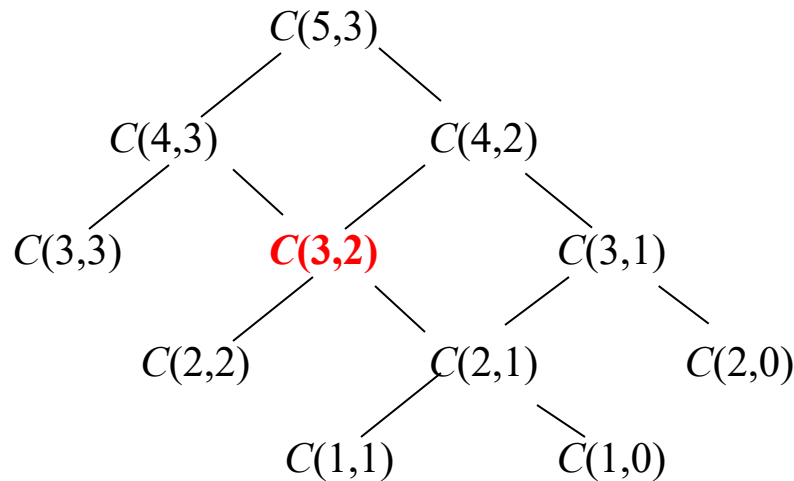
Email: bd_sahu@nitrkl.ac.in, 9937324437, 2462358

Binomial coefficients $C(n, k)$

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

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

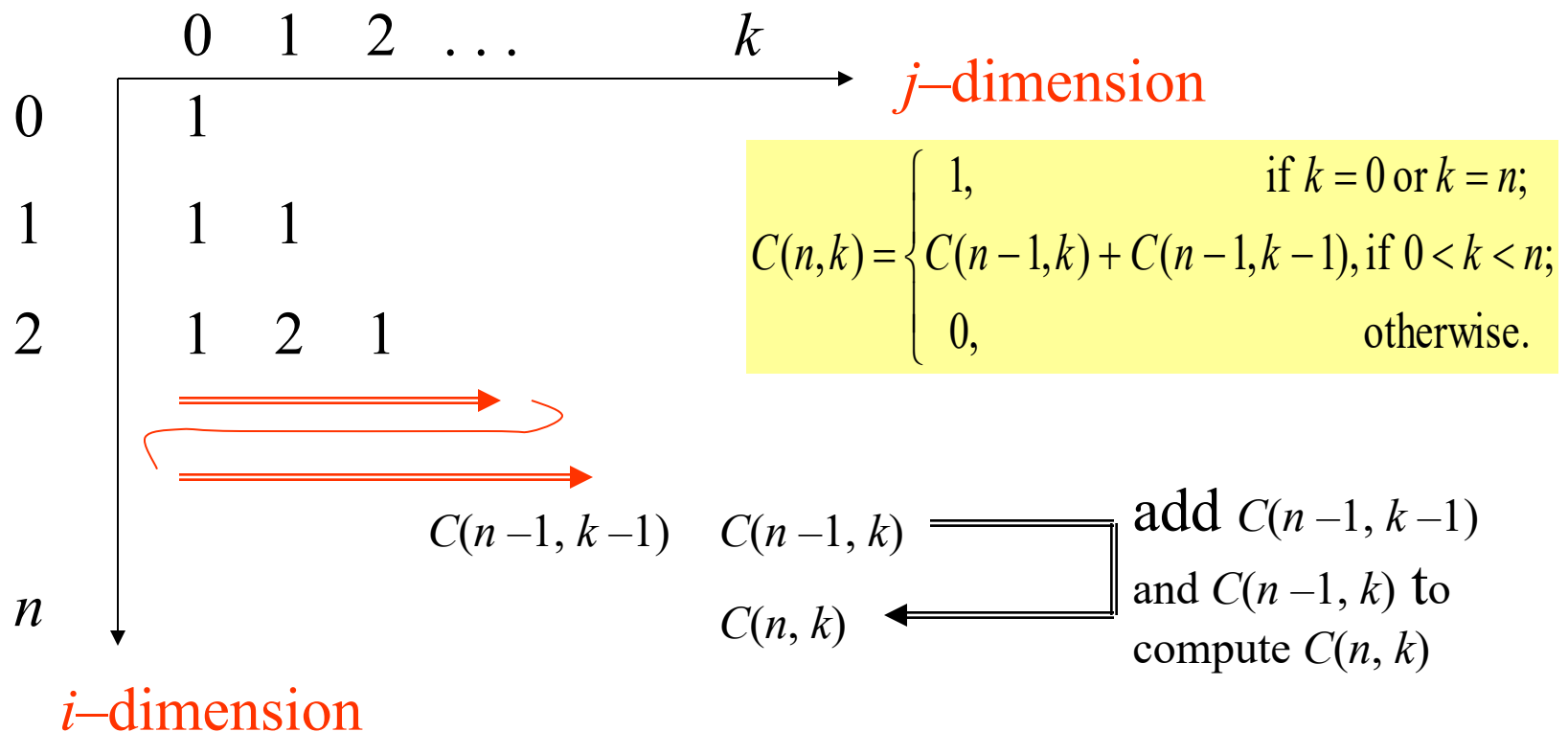
The following “**call tree**” demonstrates repeated (duplicated) computations in a straightforward recursive implementation:



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.

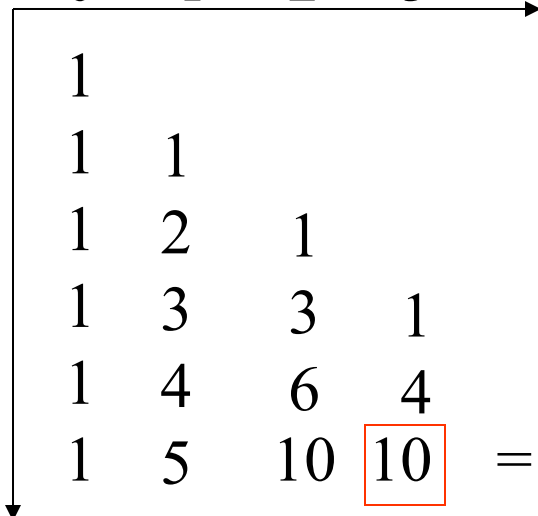
A more efficient way to compute $C(n, k)$ is to organize the computation steps of $C(i, j)$, $0 \leq i \leq n$ and $0 \leq j \leq k$, in a tabular format and compute the values by rows (the i -dimension) and within the same row by columns (the j -dimension):



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.

Computing $C(n, k)$: pseudo code

ALGORITHM *Binomial*(n, k)

//Computes $C(n, k)$ by the dynamic programming algorithm

//Input: A pair of nonnegative integers $n \geq k \geq 0$

//Output: The value of $C(n, k)$

for $i \leftarrow 0$ **to** n **do**

for $j \leftarrow 0$ **to** $\min(i, k)$ **do**

if $j = 0$ **or** $j = i$

$C[i, j] \leftarrow 1$

else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

return $C[n, k]$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Make-change problem

Example: Solve the **make-change problem** using dynamic programming. Suppose there are n types of coin denominations, d_1, d_2, \dots , 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):

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

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

Boundary condition for amount $j = 0$

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

$C[2, 8]$

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

The Partition Problem

The Partition Problem:

Given a set of positive integers, $A = \{a_1, a_2, \dots, a_n\}$.

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 $1 \leq i \leq n$ and $0 \leq j \leq K$,

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 $(i = 1 \text{ and } j = a_1)$.

The Partition Problem:

- $P[i, j] = \text{True}$ if there exists a subset of the first i numbers a_1 through a_i whose sum equals j ; False otherwise.
- Thus, $P[i, j] = \text{True}$ if either $j = 0$ or if $(i = 1 \text{ and } j = a_1)$.

When $i > 1$, we have the following recurrence:

$$P[i, j] = P[i - 1, j] \text{ or } (P[i - 1, j - a_i] \text{ if } j - a_i \geq 0)$$

- 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:

$i \backslash j$	0	1	2	3	4	5	6
$a_1 = 2$	T	F	T	F	F	F	F
$a_2 = 1$	T	T	T	T	F	F	F
$a_3 = 1$	T	T	T	T	T	F	F
$a_4 = 3$	T	T	T	T	T	T	T
$a_5 = 5$	T	T	T	T	T	T	T

Because $j \neq a_1$

Always true

$P[4,5] = (P[3,5]$
or $P[3, 5 - a_4]$)

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

The 0–1 Knapsack Problem

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.

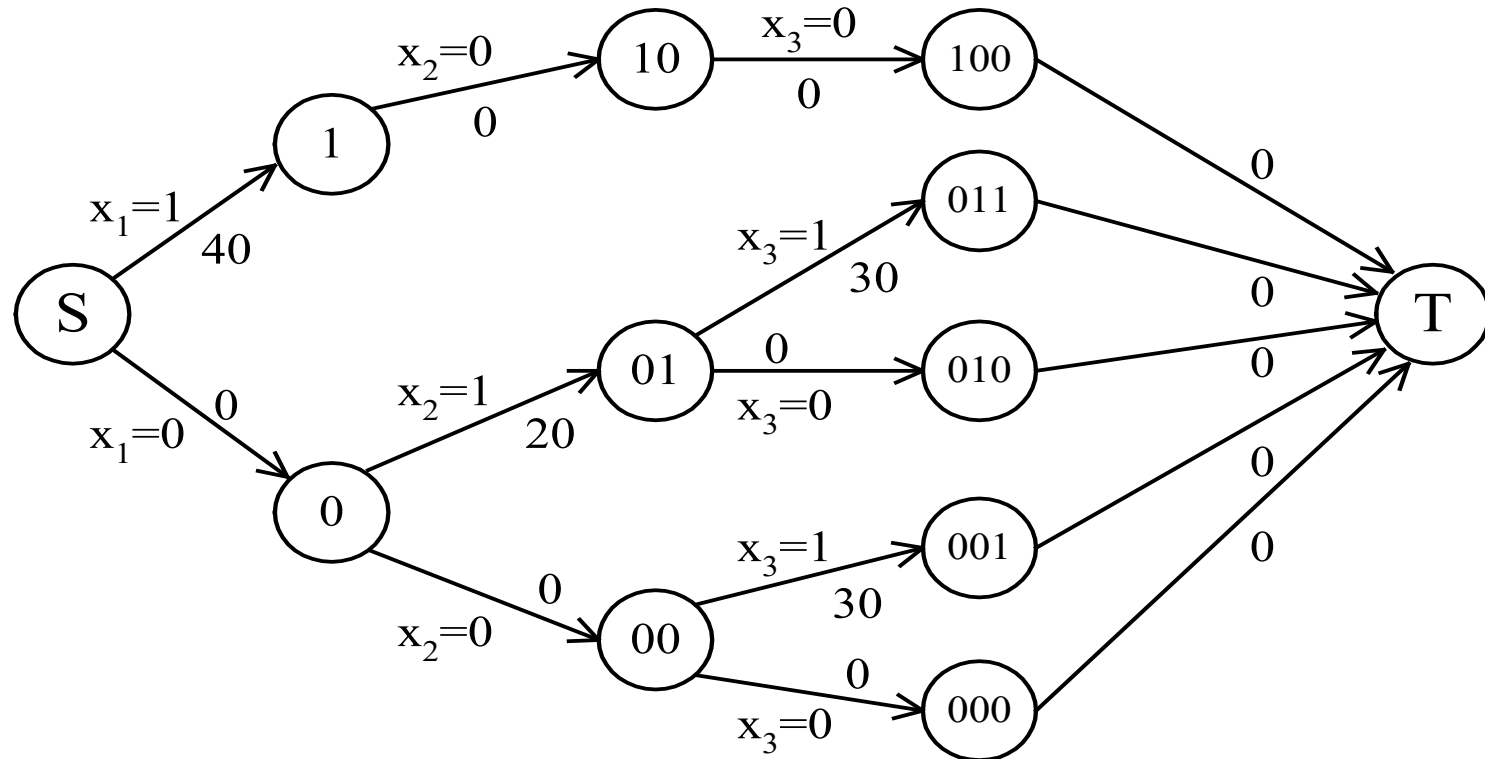
Knapsack problem

❑ *Knapsack problem* : Suppose we have n integers a_1, a_2, \dots, a_n and a constant W . We want to find a subset of integers so that their sum is less than or equal to but is as close to W as possible. There are 2^n subsets. ($n = 100$, $2^n = 1.26765 \times 10^{30}$, it takes 4.01969×10^{12} years if our computer can examine 10^{10} subsets per second.)

❑ A problem is considered *tractable* (computationally easy, $O(n^k)$) if it can be solved by an efficient algorithm and *intractable* (computationally difficult, the lower bound grows fast than n^k) if there is no efficient algorithm for solving it.

❑ The class of *NP-complete* problems: There is a class of problems, including TSP and Knapsack, for which no efficient algorithm is currently known.

Multistage Graph representation of 0/1 Knapsack problem



i	W_i	P_i	$M=10$
1	10	40	
2	3	20	
3	5	30	

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 through $i - 1$ 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).

Sack's capacity		0	1	2	3	4	5	6	7	8	9	10	11
w_i	v_i												
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

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

$V[3, 8]$ (points to 25 in row 5, column 8)

$V[4, 8] = \max(V[3, 8], 22 + V[3, 2])$ (points to 28 in row 6, column 8)

$V[3, 8 - w_4] = V[3, 2]$ (points to 6 in row 5, column 2)

Knapsack Problem by DP (pseudocode)

1. Algorithm DPKnapsack($w[1..n]$, $v[1..n]$, W)
2. var $V[0..n, 0..W]$, $P[1..n, 1..W]$: int
3. for $j := 0$ to W do
4. $V[0, j] := 0$
5. for $i := 0$ to n do
6. $V[i, 0] := 0$
7. for $i := 1$ to n do
8. for $j := 1$ to W do
9. if $w[i] \leq j$ and $v[i] + V[i-1, j-w[i]] > V[i-1, j]$ then
10. $V[i, j] := v[i] + V[i-1, j-w[i]]$; $P[i, j] := j-w[i]$
11. else
12. $V[i, j] := V[i-1, j]$; $P[i, j] := j$
13. return $V[n, W]$ and the optimal subset by backtracing



Thanks for Your Attention!

Exercises

Home assignment

- A binomial coefficient $C(n, k)$ also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k -element subsets (or k -combinations) of a n -element set. Write two dynamic programming algorithm that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$ with space complexity $O(nk)$ and $O(k)$ in computing $C(n, k)$.
-
- Suppose there are n types of coin denominations, d_1, d_2, \dots , and d_n . (We may assume one of them is penny.) There are infinite supplies of coins of each type. Write two dynamic programming algorithm to make change for an arbitrary **amount** k using the minimum number of coins with space complexity $O(nk)$ and $O(k)$.