Dynamic programming

Dynamic programming

Dynamic → changing problems

programming → Making Decisions

$5 + 1 = 6$

$5 + 1 + 2 = 8$

$5 + 1 + 2 + 2 = 10$

Already computed values

Dynamic programming is a technique using which we can solve some of the problems in polynomial time.

Dynamic programming is mainly an optimization over plain recursion.

Difference between DP algorithm vs recursion.

→ DP – Breaking a bigger problem into smaller subproblems and storing the results of these problems so that we don't have to re-solve | re-compute these problems.

→ Recursion – A function calling itself and executing itself.

Techniques to solve DP problems –

Dynamic programming

↓ solve using

Types of approaches

Top-Down Approach → uses → Memoization

Bottom-up approach → uses → Tabulation

1. Top - Down approach

Break the given problem in order to solve it. If you see a subproblem that has been computed already then return the saved solution. If you see a subproblem that has not been computed already then solve it and store the solution which can be used by other subproblems.

2. Bottom - up approach

Analyze the problem and see in which order we can solve the sub-problems. This process ensures that we are solving all subproblems before the main problem.

	Tabulation	Memoization
Code	Code gets complicated because of lots of conditions.	Code is easy and less complicated.
Speed	Fast	Slow
Sub problem solving	All subproblems will be solved atleast once.	Not required to solve all subproblems.
Approach	Iterative	Recursive

How to solve DP problems ?

In DP, we need to check for two conditions —

(i) Overlapping subproblem

(ii) optimal substructure property

Steps to solve a DP problems —

1. Identify if given problem is a DP problem or not.
2. Decide a state expression with least parameters.
3. Formulate the state and transition relationship.
4. Do tabulation or memoization.

Fibonacci Series

n	0	1	2	3	4	5	6	. .
fib	0	1	1	2	3	5	8	. . .

Base

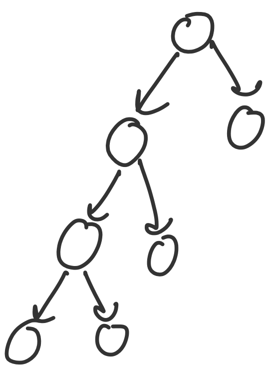
Any term in fib series = Sum of previous two values

$$fib(n) = fib(n-1) + fib(n-2)$$

```
int fib(n)
{
    if (n == 0 || n == 1)
    {
        return n
    }

    return fib(n-1) + fib(n-2);
}
```

Base case

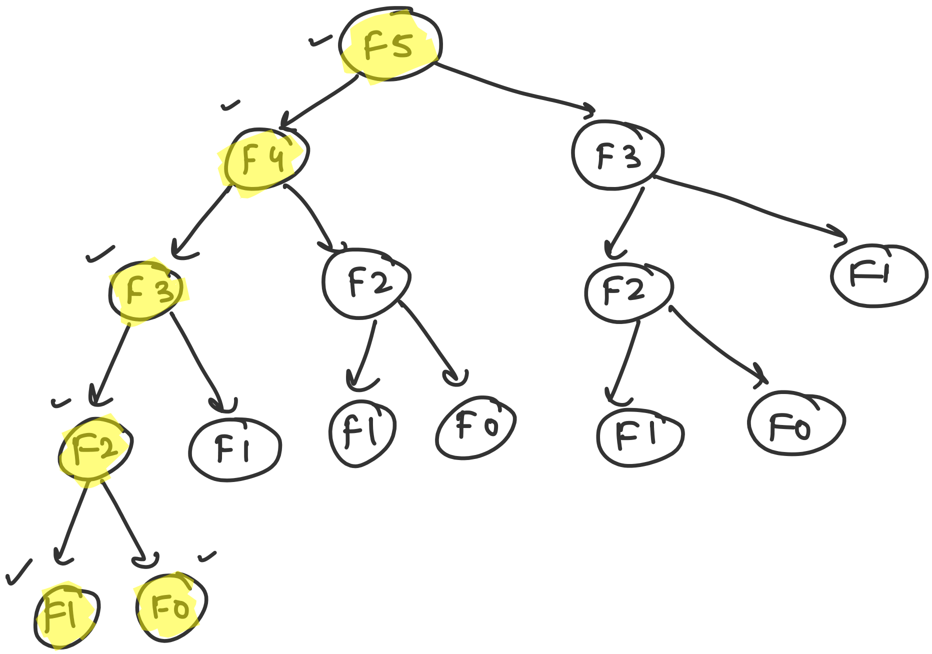


T.C. = $O(2^n)$

S.C. = $O(n)$

Recursion tree

$n = 5$



Observation - Already solved | overlapping subproblems

$n = 5$

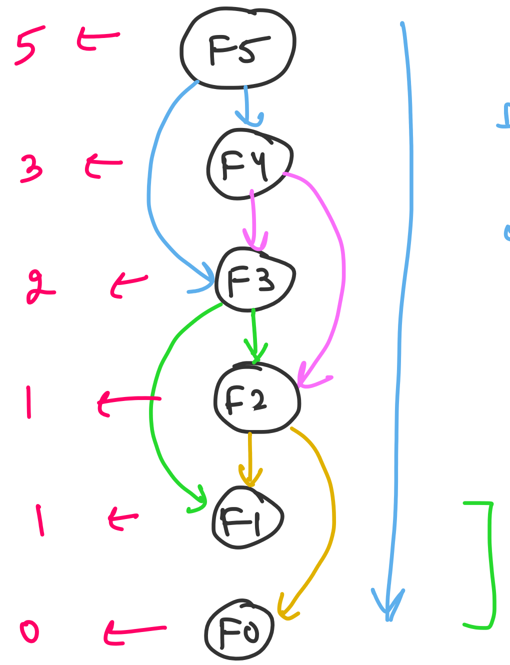
↓

for this we need 6 values

Goal - optimize T.C. by avoiding repetitive calculations.

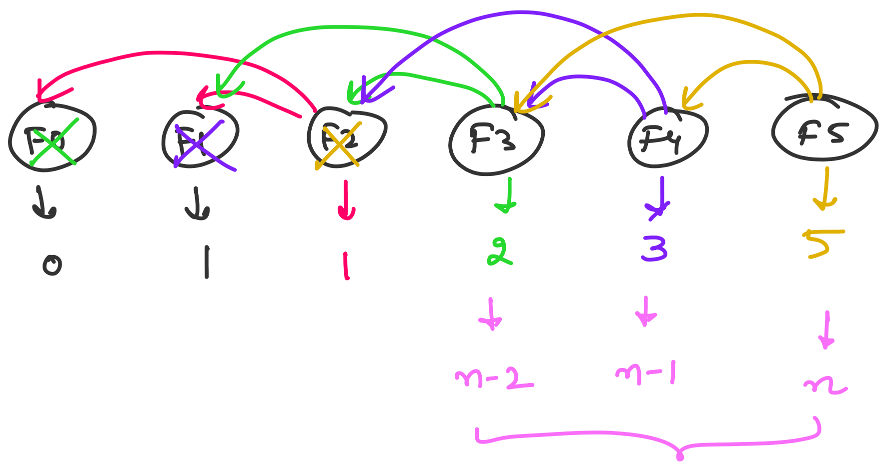
Approach 1 -

Top- Down approach



Defined order of solving subproblems

Base case




```
int fib (int n)
{

    a= 0, b=1 , c

    if (n = = 0)
        ↓
```

```
        return a

    for (i = 2 to n)
    {
        c = a + b

        a = b

        b = c
    }

    return b

}
```

T.C. = O(n)

S.C. = O(1)

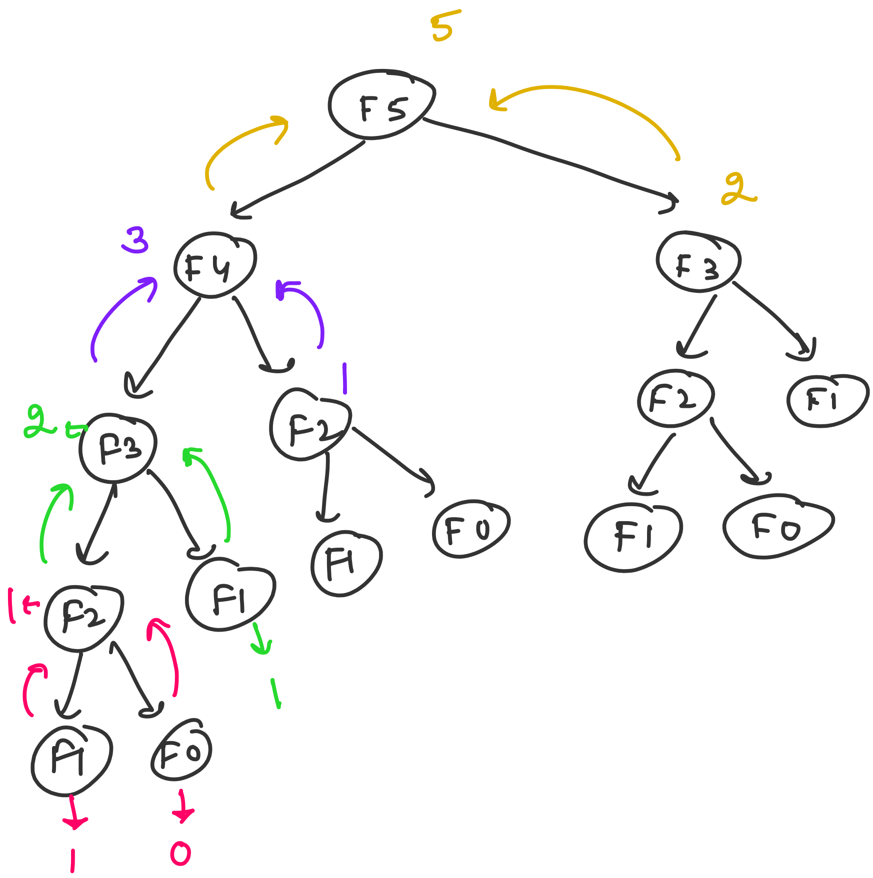
Approach - 2

Bottom up approach

fib(n) $\longrightarrow$ Is "n" in table ?

Yes $\downarrow$ Return value from table

No $\downarrow$ Compute value and store in the table.



n = 5

n+1 → Table | Array

0	1	2	3	4	5
0	1	1	2	3	5

Psuedo code

fib → fibHelper

int fib (int n)

{

create an array of size n+1

↓

fibSeries [ ] = new int [n+1]

Initialize all array values to -1

↓

(i) use Arrays. fill

OR

(ii) DO using iteration

```
for (int i=0 ; i<=n ; i++)
{
    fibSeries [i] = -1
}
```

call Helper f^n

↓

fibHelper (n, fibSeries);

}

int fibHelper (int n, int[] fibSeries)

{

check for the base case

if (n == 0 || n == 1)

↓

return n

check if the value is already

there in the table or not

↓

if (fibSeries [n] != -1)

↓

(means it is there in table)

↓

return the value

↓

return fibSeries [n]

otherwise calculate the value

and store in the array

(i) Find last term → $n-1$

x = fibHelper ($n-1$, fibSeries)

(ii) Find second last term → $n-2$

y = fibHelper ($n-2$, fibSeries)

Store the value for future use

fibSeries [n] = $x + y$

Return the value

return fibSeries [n]

}

Matrix chain multiplication

↓

Given the dimensions of sequence of matrices in an array arr [] , the task is to find out the product of these matrices in the most efficient manner such that the total number of matrix multiplications is minimum.

$c1 = r2$

arr = [3, 2, 3]

$A\ (3 \times 2)$ $\qquad$ $B\ (2 \times 3)$

$r1 \quad c1 \qquad\qquad r2 \quad c2$

$C\ (3 \times 3)$ ✓

$A\ (3 \times 2)$ $\qquad\qquad$ $B\ (2 \times 3)$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \qquad \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

$$C \begin{bmatrix} a_{11} b_{11} + a_{11} b_{21} & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

Total no of scalar multiplications to

calculate $\underbrace{a_{11} b_{11}} + \underbrace{a_{11} b_{21}} = 2$

order of matrix $C = 3 \times 3$

Total no of scaler multiplication $= 3 \times 3 \times 2$

$= 18$

Another eg -

$$\underbrace{\begin{bmatrix} \quad \end{bmatrix}_{P \times R} \qquad \begin{bmatrix} \quad \end{bmatrix}}_{R \times Q}$$

Total no of multiplications $= \underline{P \times Q \times R}$

$= \underline{\underline{P \times R \times Q}}$

Another eg-

$$ABC$$

$(A \times B) \times C \qquad\qquad A * (B \times C)$

$A - 2 \times 1$

$B - 1 \times 2$

$C - 2 \times 4$

$D = A \times B$ — O — 2×2

— M — $2 \times 1 \times 2 = 4$

$\underbrace{(A \times B)} \times C$

$\downarrow$

$D \times C$ — O — 2×4

— M — $2 \times 2 \times 4 = 16$

Total no of multiplications $= 4 + 16$

$= 20$

$A \times (B \times C)$

A — 2×1

B — 1×2

C — 2×4

$E = B \times C$ — O — 1×4

— M — $1 \times 2 \times 4 = 8$

$A \times (\underline{\underline{B \times C}})$

$\downarrow$

$A \times E$ — O — 2×4

— M — $2 \times 1 \times 4 = 8$

Total no of multiplications = 8 + 8

= 16

ABC

(A × B) × C   A × (B × C)

↓   ↓

20   16 → Min value

Another eg –

ABCD

A × (BCD)   (A × B) × (C × D)   (ABC) × D

A × ((B × C) × D)   A × (B × (C × D))   ((A × B) × C) × D

(A × (B × C)) × D

A – 1 × 2

B – 2 × 1

C – 1 × 4

D – 4 × 1

$A \times ((B \times C) \times D)$

$B \times C$ — O — 2×4
└ M — $2 \times 1 \times 4 = 8$

$(B \times C) \times D$ — O — 2×1
└ M — $2 \times 4 \times 1 = 8$

$A \times ((B \times C) \times D)$ — O — 1×1
└ M — $1 \times 2 \times 1 = 2$

Total $= 8 + 8 + 2 = 18$

$A \times (B \times (C \times D))$

A — 1×2
B — 2×1
C — 1×4
D — 4×1

$C \times D$ — O — 1×1
└ M — $1 \times 4 \times 1 = 4$

$B \times (C \times D)$ — O — 2×1
└ M — $2 \times 1 \times 1 = 2$

$A \times (B \times (C \times D))$ — O — 1×1

$M - 1 \times 2 \times 1 = 2$

$\text{Total} = 4 + 2 + 2 = 8$

$(A \times B) \times (C \times D)$

$A - 1 \times 2$
$B - 2 \times 1$
$C - 1 \times 4$
$D - 4 \times 1$

$A \times B$ — $O - 1 \times 1$
$M - 1 \times 2 \times 1 = 2$

$C \times D$ — $O - 1 \times 1$
$M - 1 \times 4 \times 1 = 4$

$(A \times B) \times (C \times D)$ — $O - 1 \times 1$
$M - 1 \times 1 \times 1 = 1$

$\text{Total} = 2 + 4 + 1 = 7$

$((A \times B) \times C) \times D$

$A - 1 \times 2$
$B - 2 \times 1$

$C - 1 \times 4$
$D - 4 \times 1$

$A \times B$ — $O - 1 \times 1$
└ $M - 1 \times 2 \times 1 = 2$

$(A \times B) \times C$ — $O - 1 \times 4$
└ $M - 1 \times 1 \times 4 = 4$

$((A \times B) \times C) \times D$ — $O - 1 \times 1$
└ $M - 1 \times 4 \times 1 = 4$

Total $= 2 + 4 + 4 = 10$

$(A \times (B \times C)) \times D$

$A - 1 \times 2$
$B - 2 \times 1$
$C - 1 \times 4$
$D - 4 \times 1$

$B \times C$ — $O - 2 \times 4$
└ $M - 2 \times 1 \times 4 = 8$

$A \times (B \times C)$ — $O - 1 \times 4$
└ $M - 1 \times 2 \times 4 = 8$

$(A \times (B \times C)) \times D$ — O — 1×1

M — $1 \times 4 \times 1 = 4$

Total = $8 + 8 + 4 = 20$

ABCD

$A \times (BCD)$ — $A \times ((B \times C) \times D) \rightarrow 18$; $A \times (B \times (C \times D)) \rightarrow 8$

$(A \times B) \times (C \times D) \rightarrow 7$ Min

$(ABC) \times D$ — $((A \times B) \times C) \times D \rightarrow 16$; $(A \times (B \times C)) \times D \rightarrow 20$

optimal substructure

$A_1 \quad A_2 \quad A_3 \quad \ldots\ldots \quad A_n$

$p_0 \times p_1 \quad p_1 \times p_2 \quad p_2 \times p_3 \quad \ldots \quad p_{n-1} \times p_n$

array = $\{p_0, p_1, p_2, \ldots\ldots p_n\}$

$A_i = p_{i-1} \times p_i$

for partitioning at $\underline{\underline{k}}$

$i \le k \le j$

$A_i \, A_{i+1} \, A_{i+2} \, \ldots \, A_j =$

$(\underline{A_i \, A_{i+1} \ldots A_k}) \times (\underline{A_{k+1} \, A_{k+2} \ldots A_j})$

Tabulation method

Step 1 —

1. Take variables —

$i, \; j, \; k, \; l, \; q$

k – partition

chain length

cost

2. Iterate from $l = 2$ to $n-1$ which denotes length of the range.

(x) Iterate from $i = 0$ to $n-1$

(i) find the right end of the range (j) having l matrices.

(ii) Iterate $K = i+1$ to j which denotes point of partition.

(a) Multiply the matrices in the range (i, j) and (j, k)

(b) This will create two matrices with dimensions — $arr[i-1] * arr[k]$ and $arr[k] * arr[j]$

(c) The total no of multiplications to be performed to multiply 2 matrices —

$$arr[i-1] * arr[k] * arr[j]$$

(d) Total no of multiplication

$$m[i][k] + m[k+1][j] + q$$

2. Return output — $\underline{\underline{m[1][n-1]}}$

Program to find nth catalan number

used to find the number of possibilites of various combinations

$$\left[C_n = \frac{(2n)!}{(n+1)!\, n!} \right]$$

$$\left\{ \begin{array}{l} 1! = 1 \\ 0! = 1 \\ n! = n * (n-1)! \end{array} \right\}$$

$\underline{n = 0}$

$$C_0 = \frac{0!}{1!\, 0!} = 1$$

$\underline{n = 1}$

$$C_1 = \frac{\cancel{2}!}{\cancel{2}!\, 1!} = 1$$

$\underline{n=2}$

$$C_2 = \frac{4!}{3!\ 2!} = \frac{\overset{2}{\cancel{4}} \times \cancel{3!}}{\cancel{3!} \times \cancel{2} \times 1}$$

$$= 2$$

$\underline{n=3}$

$$C_3 = \frac{6!}{4!\ 3!}$$

$$= \frac{\overset{2}{\cancel{6}} \times 5 \times \cancel{4!}}{\cancel{4!} \times \cancel{3} \times \cancel{2} \times 1} = 5$$

n	0	1	2	3	4	5	6	...
c	1	1	2	5	14	42	132	...

Recursive sol^n -

Base case - $C_0 = 1$
$C_1 = 1$

$$C_{n+1} = \sum_{i=0}^{n} C_i * C_{n-i} \ , \ n \geq 0$$

```
int catalan (int n)
{
    result = 0

    // Base case

      if ( n ≤ 1 )
           ↳ return 1

     for ( i = 0 to n)
     {
         result = result +
              catalan (i) * catalan (n-i-1)
     }
   return result
}
```

use DP to optimize -

Steps -

1. Create an array catalan[ ] to store

ith catalan number.

2. Initialize $\begin{cases} \text{Catalan}[0] = 1 \\ \text{Catalan}[1] = 1 \end{cases}$

3. Iterate from $i = 2$ to n

↓

loop through $j = 0$ to i

and keep adding values of

catalan [i] * catalan [i-j-1] into

catalan [i]

4. Finally return result — catalan [n]

Difference between greedy approach and dynamic programming

<u>Greedy</u> –

① In this we make locally optimal

choices at each step hoping to get global optimal without considering the future consequences.

② This approach is faster and simpler but it does not guarantee optimal solution always.

DP-

① In this we break our problem into subproblems and then we solve our subproblems recursively and store the output of these subproblems which can be reused later in order to avoid recomputation.

② This is slow and more complex but it guarantees optimal solution always.

0/1 Knapsack

Knapsack

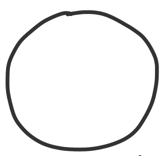
fractional → Greedy Approach

0/1 → DP Approach

S1	S2	S3
100gm	400gm	500gm

$\underbrace{100gm \quad 400gm \quad 500gm}_{1kg}$

Sweets → Fractional



watermelon

[1.5 kg, 2.3 kg] → 0/1

↓

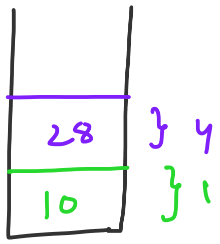
will take whole, not in parts

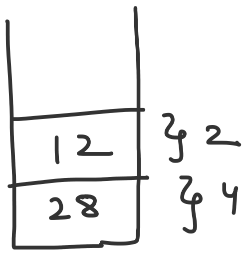
Eg-

$c = 6$

objects	1	2	3
profit	10	12	28
weight	1	2	4

p/w	10 ✓	6	7 ✓





fractional ↓

Total profit = 28 + 10
= 38

0/1 ↓

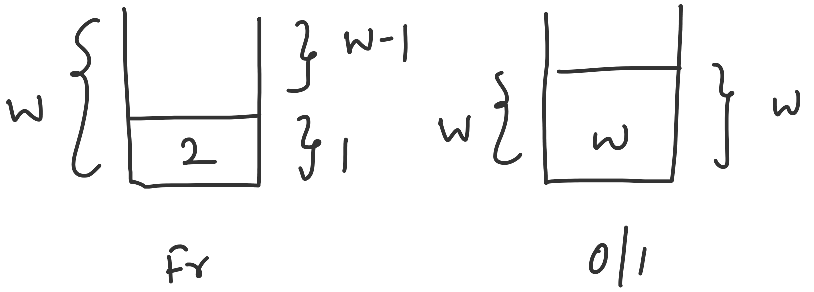
total profit = 12 + 28
= 40

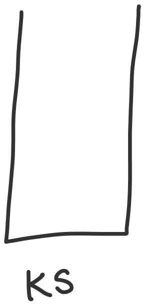
Another eg -

capacity = w

object	1	2
profit	2	W
weight	1	W

p/w	2 ✓	1





$$\underbrace{\begin{matrix} 1, & 2, & 3, & \dots\dots, & n \\ \downarrow & \downarrow & \downarrow & \dots\dots & \downarrow \\ 2 & 2 & 2 & & 2 \end{matrix}}_{2^n}$$

$$KS(i, w) = \begin{cases} 0 & i = 0 \text{ or } w = 0 \\ KS(i-1, w) & w_i > W \\ \max \begin{cases} p_i + KS(i-1, W - w_i) \\ KS(i-1, W) \end{cases} \end{cases}$$

Assume we have 10 objects of weight 1 unit each and the capacity is 10 units.

- KS(10, 10)
  - KS(9, 9)
    - KS(8, 8)
      - KS(7, 7)
        - ⋮
        - KS(0, 0)
      - KS(7, 8)
    - KS(8, 9)
      - KS(7, 8)
      - KS(7, 9)
  - KS(9, 10)
    - KS(8, 9)
    - KS(8, 10)

$2^n \rightarrow n \times w$

n → no of objects

w → capacity

$O(2^n) \longrightarrow O(n \times m)$

Eg -

$$KS(i, w) = \begin{cases} 0 & i = 0 \text{ or } w = 0 \\ KS(i-1, w) & w_i > w \\ \max \begin{cases} p_i + KS(i-1, w - w_i) \\ KS(i-1, w) \end{cases} & \end{cases}$$

$c = 6$

objects	1	2	3
profit	10	12	28
weight	1	2	4

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

→ Max profit

$$KS(3,6) = \max \begin{cases} \overset{28}{P_3} + \overset{12}{KS(2,2)} \\ KS(2,6) \quad 22 \end{cases}$$

$TC = O(n \times w)$

$SC = O(n \times w)$

Subset sum problem

Given a set of non-negative integers and the task is to find out if we have a subset of the given set having sum equal to the given sum.

$\{6, 2, 3, 1\}$ $\qquad\qquad$ sum $= 5$

$$\{ a_1, a_2, a_3, \ldots\ldots a_n \}$$

$$\underbrace{2 \times 2 \times 2 \times \cdots\cdots}_{2^n} \; 2$$

$$SS(i, s) = \begin{cases} \text{True} & s = 0 \\ \text{False} & i = 0 \quad s \neq 0 \\ SS(i-1, s) & s < a_i \\ SS(i-1, s-a_i) \vee SS(i-1, s) & \end{cases}$$

$SS(10, 10)$

$SS(9, 9)$ $\qquad\qquad$ $SS(9, 10)$

$SS(8, 8)$ $\quad$ $SS(8, 9)$ $\quad$ $SS(8, 9)$ $\quad$ $SS(8, 10)$

$$\boxed{n \times W}$$

$$ss(i, s) = \begin{cases} \text{True} & s = 0 \\ \text{False} & i = 0 \quad s \neq 0 \\ ss(i-1, s) & s < a_i \\ ss(i-1, s-a_i) \vee ss(i-1, s) & \end{cases}$$

$\{6, 3, 2, 1\}$ sum = 5

$s \rightarrow$

$i \downarrow$

i \ s	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T

→ Yes

$$ss(4,5) = \underset{F}{ss(3,4)} \vee \underset{T}{ss(3,5)}$$

T.C. = $O(n \times w)$

S.C. = $O(n \times w)$

Longest common subsequence

Give two strings - X and Y, the task is to find out length of the longest common subsequence.

Eg-

COMPUTER

↓

OM
PUT
TER
MPU
COMP

Substrings

C O M P U T E R

↓

O P T
M E R
C O M
C E R
C T E R
{ }

Subsequences

C O M P U T E R

$$\underbrace{2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2}_{2^8}$$

Eg.-

X — R A V I N D R A

Y — A J A Y

common subsequence — { {A A}, {A}, { } }

Longest common Subsequence

length of LCS = 2

Optimal substructure

X — $x_1\, x_2\, x_3 \ldots\ldots x_i$

Y — $y_1\, y_2\, y_3 \ldots\ldots y_j$

$$LCS[i,j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ 1 + LCS[i-1, j-1] & i,j>0 \ \& \ x_i = y_j \\ \max \begin{cases} LCS[i, j-1] \\ LCS[i-1, j] \end{cases} & i,j>0 \ \& \ x_i \neq y_j \end{cases}$$

Eg -

$X = A\ A\ B \qquad Y = A\ C\ A$

			0	A 1	C 2	A 3
x_i		0	0	0	0	0
	A	1	0	1	1	1
	A	2	0	1	1	2
	B	3	0	1	1	2

length of LCS

$$LCS[3,3] = \max \begin{cases} LCS[2,3] & 2 \\ LCS[3,2] & 1 \end{cases}$$

LCS = A A

length = 2

Another eg -

X - A B C B D A B

Y - B D C A B A

$y_j \rightarrow$

			B	D	C	A	B	A
		0	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	0	0	0	0	0	0
A	**1**	0	0	0	0	1	1	1
B	**2**	0	1	1	1	1	2	2
C	**3**	0	1	1	2	2	2	2
B	**4**	0	1	1	2	2	3	3
D	**5**	0	1	2	2	2	3	3
A	**6**	0	1	2	2	3	3	4
B	**7**	0	1	2	2	3	4	4

$x_i \downarrow$

$\downarrow$ Length of LCS

Subsequences —

$$\left\{ \begin{matrix} B & D & A & B \\ B & C & A & B \\ B & C & B & A \end{matrix} \right\}$$

Staircase problem

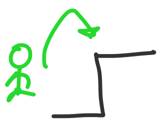
$\downarrow$

count the number of ways to reach the nth stair.

$\downarrow$

There are n stairs and a person standing at bottom who wants to reach the top. The person can take either one stair at a time or two stairs at a time. Count the number of ways in which he can reach the top.

$\underline{n=1}$

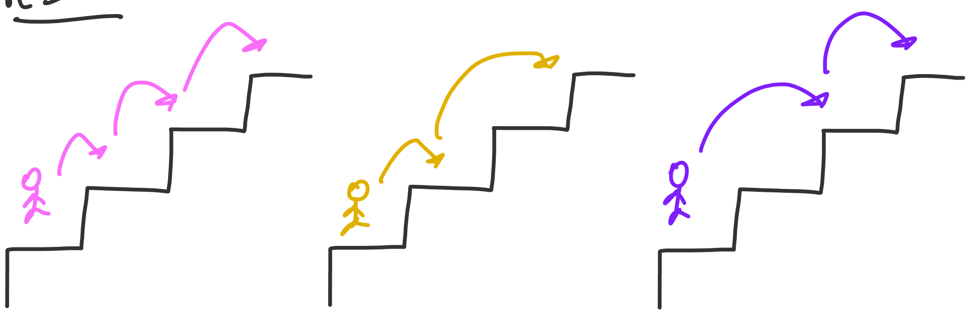


1 way

$\underline{n=2}$

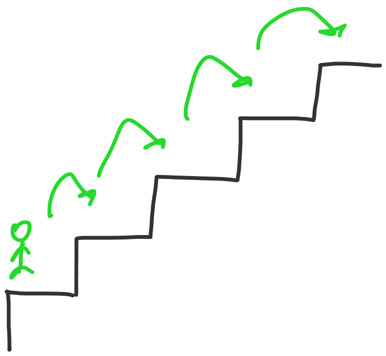
2 ways

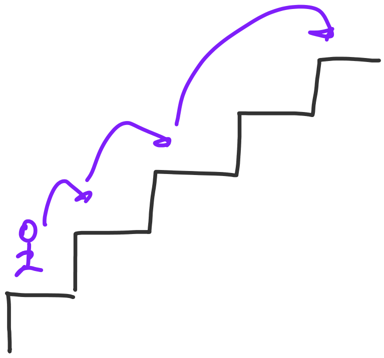
$\underline{n=3}$

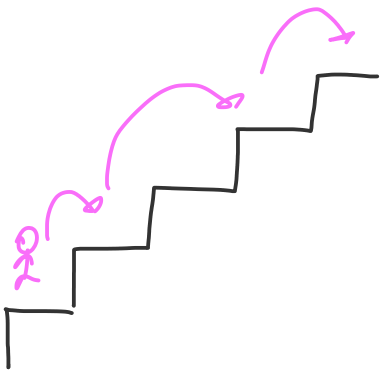


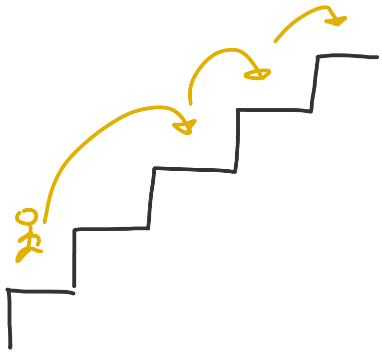
3 ways

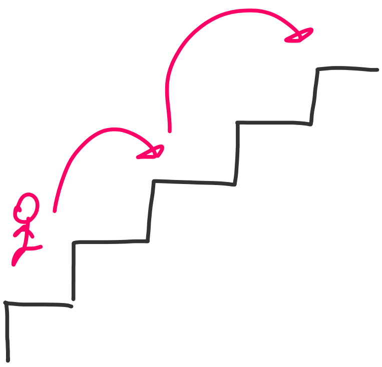
$\underline{n = 4}$











5 ways

	ways
$n = 1$	1
$n = 2$	2
$n = 3$	3
$n = 4$	5

$$ways(n) = ways(n-1) + ways(n-2)$$

DP

↓

create a table & fill in bottom up manner.

$$result[i] = result[i] + result[i-j]$$

for every $(i-j) >= 0$

such that ith index of array which will contain all the ways to reach the ith stair considering all the possibilites (1 to i).

DP approach with space optimization

↓

1. create 2 variables - prev1 & prev2

prev1 - To count number of ways to climb one stair

prev2 - To count number of ways to climb two stairs

2. Run a loop to count the total number of ways to reach the top.

```
count ways (n)
{
    prev1 = 1
    prev2 = 1

    for (i= 2 to n)
    {
        current = prev1 + prev2
```

```
            prev2 = prev1
            prev1 = current
        }
    return prev1
}
```

prev1 → Total Count

Coin change problem

↓

Given an array of coins[] of size N representing the different denominations of coins and an integer sum. The task is to find out the count of the different ways to make the sum using coins[] array.

Eg -

sum = 4 coins = {1, 2, 3}

Ist way - {1, 1, 1, 1}

IInd way - {1, 1, 2}

IIIrd way - {1, 3}

IVth ways - {2, 2}

Total no of ways = 4

Another eg -

sum = 10 coins = {2, 5, 3, 6}

Ist way - {2, 2, 2, 2, 2}

IInd way - {5, 5}

IIIrd way - {2, 2, 6}

IVth way - {2, 2, 3, 3}

Vth way - {2, 3, 5}

Total ways = 5

DP using space optimized technique –

1. Initialize the array – result with values equal to 0.
2. with sum = 0, there is a way.
3. update the level wise number of ways of coin till ith coin.
4. Solve till $j <= sum$.

0	1	2	3	4