Practice all Concepts atleast 5 times.

Page No.
Date 07-07-25

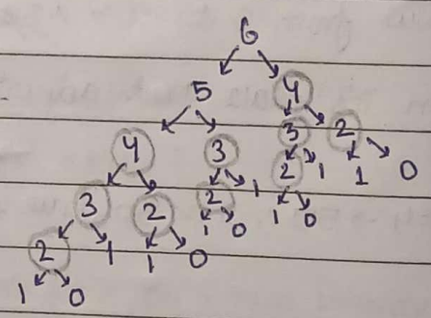
Dynamic Programming

* Optimised Recursion
* Definition → technique that helps to efficiently solve a class of problem that have overlapping subproblems and optimal substructure property.
  - identify → optimal sol^{n} needed + some choices are there.
* Ways :→ 1. Memoization → (Top Down) e.g. Fibonacci
  - → Recursion
  - → storage of subproblems for reuse*
  2. Tabulation → (Bottom UP)
  - → Iteration
  - → uses a table/set for iteration.

* Concept-1 : Fibonacci

I. * Fibonacci No.

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



1. Recursion :- $O(2^n)$

```
int n=6;
print(fib(n));
}
Static int fib(int n){
    if(n<2){
    return n;
    }
    return fib(n-1) + fib(n-2);
}
```

→ all left traverse the right at the end sum up.

Tricks to study 5x → 1. 25 gehri saans after 15 sec hold
2. Ek chiz ko 30 s stare without blink
3. For every 10 min → 10 s stop & do nothing

Ajanta
Page No.
Date

2. Memoization :- $O(n) = T.C$, $S.C = O(n)$

* Start storing all values in array
* if fib(n) already calculated return its value
* Store continuously.

chota packet bada dhamaka

Code →

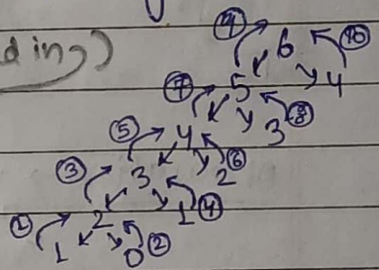
```
psvm {
int n=6;
~~print~~ int[] f = new int[n+1];  → // all initialized with zeros
print(fib(n,f));

static int fib(int n, int[] f){
    if(n<2){
        return n;
    }
    if(f[n]!=0){  → // Already calculated
        return f[n]; → already computed ans.
    }
    f[n] = fib(n-1,f)+fib(n-2,f); → // storing
    return f[n]; (intermediate ans. stored in f)
}
```

vvv imp ← ∵ non-static method can't be referenced from static (main→psvm)

3. Dry run →

0	1	2	3	4	5	6
✓	✓	1	2	3	5	8



f[6] = fib(5,f)+fib(4,f) ← for fib(6,f)

for fib(5,f) called first → f[5] = fib(4,f)+fib(3,f)
" " (4,f) → f[4] = fib(3,f)+fib(2,f)
" " (3,f) → f[3] = fib(2,f)+fib(1,f)
" " (2,f) → f[2] = fib(1,f)+fib(0,f) ~~1+0=1 → f[2]~~
(1) " " (1,f) → 1 — then this called
(2) " " (0,f) → 0

PEAK RECURSION

Now reverse order mei lautega :-

(3) f[2] = 1+0 = 1 (4) fib(1,f) → 1 (5) f[3] = 1+1 = 2 (6) f[2] = 1 "already computed
(7) f[4] = 2+1 = 3 (8) f[3] = 2 (already computed) (9) f[5] = 3+2 = 5
(10) f[4] = 3 (already computed) (11) f[6] = 5+3 = 8 Ans.

3. Tabulation :-

```
static int fibTab(int n){
```

① Create Table → `int[] dp = new int[n+1];`

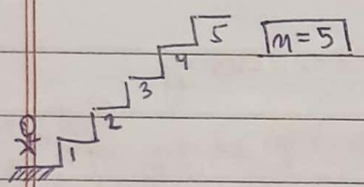
② Meaning + Initialise → `dp[0] = 0;` → //no need to write this ; its already initialized 0.
`dp[1] = 1;`

```
for(i=2 to i<=n){
```

③ Filling →
```
    dp[i] = dp[i-1] + dp[i-2];
}
return dp[n];
```
Qus. v. easy

II. * Climbing Stairs

Count no. of ways to reach n^{th} stair.
When person can climb either 1 or 2 stairs at a time.



n=5

ways for 3
1 1 1
1 2
2 1

ways for 4
1 1 1 1
1 2 1
2 1 1
1 1 2
2 2

Observation → ways of 5 ⇒ (ways for 3)+2 and (ways for 4)+1

1 1 1 2
1 2 1 2
2 1 2
ways to 3 jitna hi hai bs 2 juda hai

1 1 1 1 1
1 2 1 1
2 1 1 1
1 1 2 1
2 2 1
ways to 4 jitna bs 1 add/append hua hai

∴ 3+5 = 8 Qus.

Hence,
ways(n) = ways(n-1) + ways(n-2).

2 imp. base condition → 1. n==0 → return 1; (ground to ground)
2. n<0 → return 0; → to avoid ways(-1) when to find ways(1).

Code v. easy (if you understood fib) → check github.

Page No.
Date

* Concept-2 : 0-1 Knapsack

I. * To select max. valued item for W kg knapsack by selecting items by take it or ignore it method for reaching max. weight (not completely fill necessary)

	0	1	2	3	4	5
eg. value = {	100 ,	50 ,	30 ,	25 ,	5 ,	15 }
wt. = {	10 ,	5 ,	2 ,	1 ,	0.5 ,	1 }

W = 10.

∴ using 0-1 knapsack → case: 1 → A_0 = 10 Kg → 100 = val

case: 2 → $A_1 + A_2 + A_3 + A_4 + A_5$ = 9.5 kg → 125 = val ✓

Scene will be diff. for unbounded → $10 \times A_3$ = 250 val of 10 kg.

How DP? → when we need to find best sol^{n} from feasible sol^{n} possible.

Approach → *

item (val, wt)

(wt <= w) valid ← → invalid

exclude : invalid

include (1) exclude (0) : select max.

* base condition → W capacity = 0 or no items left

pseudocode →

```
Knapsack (val[], wt[], wt, n)
    if (W==0 || n==0){
        return 0;
    }
    if (wt <= w) {→ valid
① include → take val + W-wt, i-1
② exclude →          W, i-1
    }
    else { → invalid
        exclude → W, i-1
    }
```

* make recursion tree for n=5, w=7 yourself.

1. Recursion $O(2^n)$ → v.v.easy

bad

```
static int knapsack(int[] val, int[] wt, w, n){
    if(w==0 || n==0){ → base condition
        return 0;
    }
    if(wt[n-1] <= w){ //valid
include→  int include = val[n-1] + knapsack(val, wt, w-wt[n-1], n-1);
exclude→  int exclude = knapsack(val, wt, w, n-1);
        return Math.max(include, exclude); →choose lena samajhdari hai ya chorna
    }else{
        return knapsack(val, wt, w, n-1);
    }
```

2. Memoization $O(n*w)$

good

- first of all in main function make a 2D array dp & initialize it with -1 (∵not zero because our ans. can come zero also).

```
int[][] dp = new int[val.length+1][w+1];
```

(∵ val[], wt[] → fixed.
w, n → variables in recursion ∴ 2D array of these.)

```
for(i=0 to dp.length){
    for(j=0 to dp[0].length){
        dp[i][j] = -1;
    }
}
```

-

```
static int knapsackMemo(int[]val, int[]wt, w, n, int[][]dp){
    if(w==0 || n==0){
        return 0;
    if(dp[n][w] != -1){
        return dp[n][w];
    }
```

```
if (wt[n-1] <= W) {
int include = val[n-1] + knapsackMemo(val, wt, W-wt[n-1], n-1, dp);
int exclude = knapsackMemo(val, wt, W, n-1, dp);
dp[n][W] = Math.max(include, exclude);
return dp[n][W];
} else {
dp[n][W] = knapsackMemo(val, wt, W, n-1, dp);
return dp[n][W];
}
```

exclude (invalid) →

3. Tabulation

① Create Table → dp[n+1][W+1] → ∵ these are variables only with 0 at row & col → by default.

② Meaning + initialise → ↓ for thinking only
- at index i,j → (2,3) ∴ n=2, W=3
- take two values from start ⇒ (1,2) & (1,4,5).
- initialize → Base case → W=0 or n=0 ∴ profit=0

③ Filling →
- all rows & ~~col~~ col with zero
- code (main crux).

```
→ static int knapsackTab(int[] val, int[] wt, int W) {
    int n = val.length;
    int[][] dp = new int[n+1][W+1];
    for (int i = 0 to n+1) {
        for (int j = 0 to W+1) {
            int v = val[i-1];
            int w = wt[i-1];
            if (w <= j) {
                int include = v + knapsackTab(val, wt, W-w);
                int exclude = knapsa
                int include = v + dp[i-1][j-w];
                int exclude = dp[i-1][j];
                dp[i][j] = Math.max(include, exclude);
            }
```

∴ 0th col & 0th row main zero stored hai already ∴ start from zero.

but val, wt to zero index se lena start karenge val!

start i,j from 1 not zero

Empty → return type → node ∴ return ;
↳ " " → void ∴ return ;

Ajanta
Page No.
Date

```
        else {
            dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][W];
```

* II. * Target Sum Subset

num[] = 4, 2, 7, 1, 3.
Target Sum = 10
Achievable or Not ?
→ Yes ({7,3}, {4,2,1,3}, {7,2,1}).

(Note:- Whenever you're in the choice of element (proces, unproces) then after that if we need to get pairs of elements or greater no. of items like sets, P&C then it's a backtracking ~~corupt~~ Ques. which takes higher complexities but in such of elements when we need to get one solution that's which is optimum one only than → Greedy, DP Ques. with low complexities.)

1. Tabulation → O(n * sum) (items "i", target sum "j")
   a. Table → dp[n+1][sum+1];
   b. meaning + initialise →
      smaller problem → dp(i,j) → i item ⇒ subset sum=j ? T/F
      e.g. dp(3,5) → 3 items ⇒ subset sum = 5 ? False (can't make)
      (4,2,7)
      initialise → base case : 1. sum=0 ∴ always True using φ set
      2. items=0 & sum>0 ∴ False always
      ∴ 0th col → all True ∴ 0th row from 1st col → all False
      ↓ no need for this ∵ by default is false.
   c. Filling → small to large

→ static boolean targetSumSubset(int[] arr, int sum){

int n = arr.length;

create ⓐ. boolean[][] dp = new boolean[n+1][sum+1];

Initialize (meaning is in memory) ⓑ. for(i=0 to n+1){

dp[i][0] = true;

} all rows zero^th col.

Filling ⓒ. for(i=1 to n+1){

for(j=1 to sum+1){

int v = arr[i-1];

(j-v) sum kiya (i-1) items leke aa payenge.

include karne ke baad kya value reh jayegi

check include → if(v<=j && dp[i-1][j-v] == true){ (Chosen)

(checks valid)

dp[i][j] = true; → Now included

}

exclude → if(dp[i-1][j] == true){

dp[i][j] = true;

}}}

return dp[n][sum];

III. * Rod Cutting — Unbounded Knapsack (practice 0-1 knapsack Q. using this too!)

A rod of length 'n' includes prices of all pieces of size <= n. Determine max value obtained by cutting rod & selling pieces

length = {1, 2, 3, 4, 5, 6, 7, 8}; rod length = 8

price = {1, 5, 8, 9, 10, 17, 17, 20}.

case:1 → 4,4 → 9+9=18 ; case:2 → 7,1 } 17+1=18 ; case:3 → 2,6 → 5+17=22 ✗

unbounded within limit

1. Tabulation

```
code → static int rodCuttingTab(int[] price, int[] length, int rod){
    int n = price.length;
    int[][] dp = new int[n+1][rod+1];
    ---
    for(i=1 to n+1){
        for(j=1 to rod+1){
            int v = price[i-1];
            int wt = length[i-1];
            if(wt <= j){ // valid
```

baar baar le skte hai uss length piece ko → `int include = v + dp[i][j-wt];`

`int exclude = ~~dp[i-1][j-wt]~~ dp[i-1][j];`

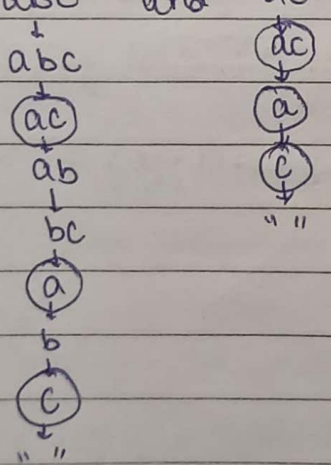
```
            dp[i][j] = Max(include, exclude);
        }else{
            dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][rod];
```

✱ Concept-3: Longest Common Subsequence

I. LCS

Imp: Subsequence of a string is new string generated from original string with some characters ⇒ without changing the relative order.

eg. abc and ac

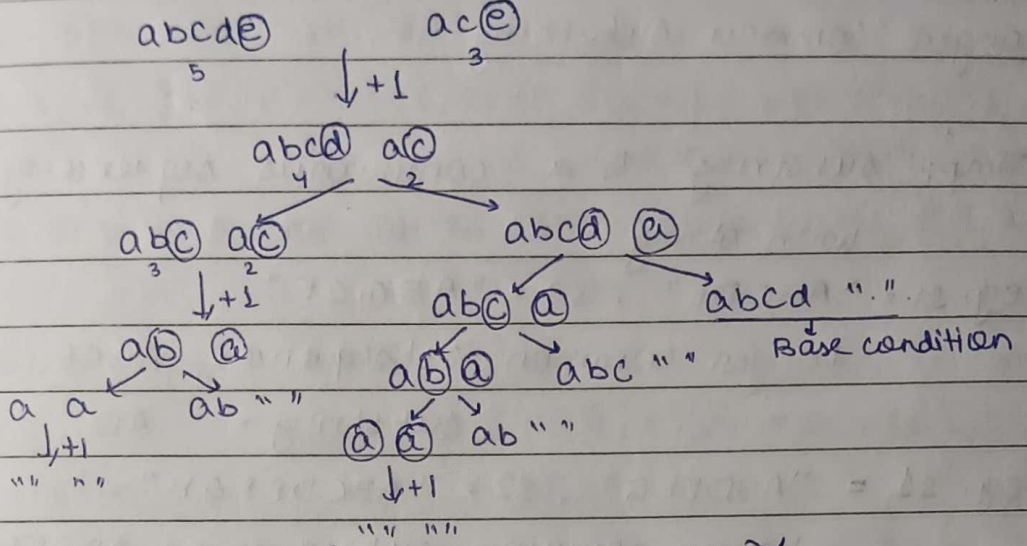


from common subsequences, longest one is ac → size = 2 ans

m=5 n=3

• approach → str1 = "abcde", str2 = "ace"

ans = "ace"



1. recursion → lcs(str1, str2, n, m){

Base condition → if (n==0 || m==0){
return 0;

same → if (str1(n-1) == str2(m-1)){
return lcs(str1, str2, n-1, m-1) + 1
}

different → else {
ans1 = lcs(str1, str2, n-1, m);
ans2 = lcs(str1, str2, n, m-1);
return max(ans1, ans2);
}

O(n×m) :-

2. memoization → initialise dp[n+1][m+1] with -1 in main fn.

static int lcs(String str1, String str2, int n, int m, int[][] dp){
if(n==0||m==0){ return 0; }
if(dp[n][m] != -1){ return dp[n][m]; }

same → if(str1.charAt(n-1) == str2.charAt(m-1)){
return dp[n][m] = lcs(str1, str2, n-1, m-1, dp) + 1;
}

different → else {
int ans1 = lcs(str1, str2, n-1, m, dp);
" ans2 = " " " n, m-1 " ";
return dp[n][m] = max(ans1, ans2);
}
}

3. Tab^{n} code → go and check github NOW.

II. Longest common substring

Imp:- substring is a continuous sequence of characters with string.

e.g. S1 = "ABCDE", S2 = "ABGCE"

Longest common subsequence = ABCE

" " substring = AB

e.g. S1 = "ABDEFG", S2 = "ABCDEFGH"

Longest common subsequence = ABDEFG

" " substring = DEFG

Concept → substring(n, m)

- Same: dp[i][j] = (n-1), (m-1) + 1
- different: dp[i][j] → 0 ∴ counter resets

1. Tabulation code →

```
static int substring(String str1, String str2){
    int n = str1.length();
    int m = ~
    int count = 0;                    n+1
Create Table → int[][] dp = new int[n+1][m+1];
    for( i=1 to n+1){
        for (j= 1 to m+1){
    if(str1.charAt(i-1) == str2.charAt(j-1)){
        dp[i][j] = dp[i-1][j-1] + 1
        count= max(count, dp[i][j]);
    }
    else{
        dp[i][j] = 0;
    }
    }
}
return count;
```

Ajanta
Page No.
Date

III. Longest increasing subsequence

arr[] = {50, 3, 10, 7, 40, 80}

length of LIS = 4 ∵ (3, 7, 40, 80) → one with longest increasing.
∵ duplicate elements consider 1 only.

★ [Take longest common sorted unique subsequence. = LIS]

Approach ⇒ e.g. arr[] = {✓}; → A

sorted ⇒ {3, 7, 10, 40, 50, 80}. → B.

then LCS of A and B will give LIS.

V.V. easy

Code →

```
Static int lis(int[] arr1){
Hashset<Integer> set = new Hashset<>();
for( i=0 to arr1.length) {
    set.add(arr1[i]); → got unique elements
}
```

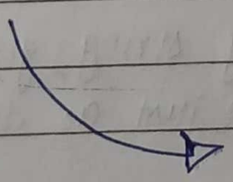
Store in another array →

```
int[] arr2 = new int[set.size()];
                    ∵ can have duplicates
int i = 0;
for(int nums : set){
    arr2[i] = nums;
    i++;
}
Arrays.sort(arr2);
return lcs(arr1, arr2); → Ans
```

→ very easy

★ IV. Edit Distance

Convert word1 to word2 using 1. Insert 2. Delete
3. Replace a character.



word 1 : abcdef (n-size)

word 2 : bdeg (m-size) .

Approach* → * check last character

1. Same :. [str1(n-1), str2(m-1);] ~~→ check in abc~~

2. Different : a). Add → we added g in word 1 .

∴ [str1(n), str2(m-1) + 1]

('n' since after adding iska size n+1 ho chuka hai) (now check in abcdef(g) bde)

But Q. says to convert 1 to 2. ← if we would've added in word 2 → str1(n-1) str2(m) + 1 ✓

b). Delete

word 1 : abcde* → ∴ str1(n-1), str2(m) + 1 .

word 2 : bdeg

(now compare in:-)

c). Replace

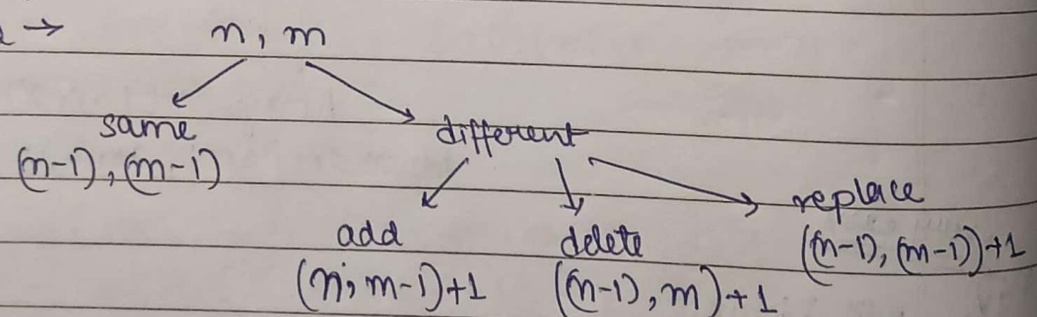
1 : abcde(f) → g

2 : bdeg

→ ∴ str1(n-1), str2(m-1) + 1 .

~~Note:- Better for~~

Decision Tree →



Imp.: Always think for Base condition .

↓

* i.e. v.v.v.easy Initialisation for Tabulation = if any string gets empty then no. of operations further will be just as the length remaining string.

Ajanta
Page No.
Date

ie. if str2 gets empty then delete all items of str1.
if str1 " " " add " " " str2.
~~dp(i,j) → i=n, j=m~~

code → static int editDist(String str1, String str2){
int n=✓, int m=✓.
create table → int[][] dp=✓
initialisation* → for(i=0 to n+1){
~~dpt~~ for(j=0 to m+1){
* [if(i==0){
dp[i][j]=j; } → no. of operations will be length of remaining string.
if(j==0){
dp[i][j]=i; }]
}}

filling → for(i=1 to n+1){ for(j=1 to m+1){
if(str1.charAt(i-1)==str2.charAt(j-1)){
~~return editDis~~ dp[i][j]= dp[i-1][j-1];
}
else{
if you're memorizing this then clapping for you { int add = dp[i][j-1] + 1; *
int delete = dp[i-1][] + 1;
int replace = dp[i-1][j-1] + 1;
dp[i][j] = min(add, delete, replace)
}}}
return dp[n][m]; not possible in this way

So, min(add, min(delete, replace))
∵ takes only 2 args.

V. Wildcard Matching

Matching wildcard ~~with~~ '?' and '*' with text.

? → matches with any single character except null.
* → " " " sequence of " including null.

e.g. Text = " baaabab"
Pattern = " * * * * * ba * * * * * ab"
(* * * * * → ϕ ; * * * → aab ; * * → ϕ)

Output = True

e.g. Text = " baaabab" → 'n'
Pattern = " a*ab" → 'm'
O/P = false.

1. Tabulation

① Create Table → n, m are variable ∴ 2D array.

② meaning + initialise →

check at smaller problem dp(i,j) → i=1, j=4
s="a", p="**?b".
O/P = false.

1. s=0, p=0 → length of strings.
ie. i=0, p=0 ∴ True always.

2. s≠0, p=0 → false e.g. s="aa", p=""

3. s=0, p≠0

s=""	p="*"	p="?"	p= anycharacter
	True	False	False

Don't use double quotes for character values.

Ajanta
Page No.
Date

leetcode hard (easy).

code → static boolean isMatch(String s, String p) {
int n = ✓ , int m = ✓ .

create table → boolean[][] dp = new boolean[n+1][m+1];

s=0, p=0 ∴ true → dp[0][0] = true;

→ s≠0, p=0 ∴ false → for (i = 1 to n+1) { dp[i][0] = false; } } No need to write.

check do we need to write for it or not ∴ false default

s=0, p≠0 → for (j = 1 to m+1) { p.charAt(j) if (p == "*") { dp[0][j] = true; ← check rest already false

∵ pichla ke basis se daling hai

for (int j=1; j<m+1; j++) { if (p.charAt(j-1) == '*') { dp[0][j] = dp[0][j-1]; } }

* filling → * Note:- Don't use double quotes for character values.

for (i = 1 to n+1) {
for (j = 1 to m+1) {

base conditions already covered

if (s.charAt(i-1) == p.charAt(j-1) || p.charAt(j-1) == '?') { dp[i][j] = dp[i-1][j-1]; } → proceed as true
↳ i.e. pichla wala true tha to ye bhi

* else if (p.charAt(j-1) == '*') {

reduce/proceed in either of them is true then true. → dp[i][j] = dp[i-1][j] || dp[i][j-1]; }

socho easy hai

else { dp[i][j] = false; → Not matched. } } }

return dp[n][m];

* Concept-4 : Catalan's Number

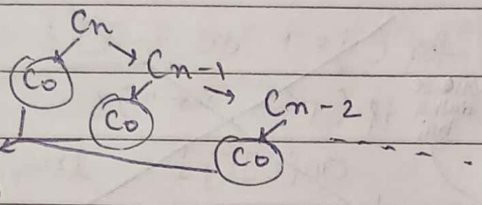
I. Binomial Catalan No.

Fixed → $C_0 = 1, C_1 = 1$; $C_2 = C_1C_0 + C_0C_1 = 2$

$C_3 = C_0 \cdot C_2 + C_1 \cdot C_1 + C_2 \cdot C_0 = 2 + 1 + 2 = 5$

$$\left[C_n = C_{n-1}C_0 + C_{n-2}C_1 + \dots + C_0C_{n-1}\right]$$

Why DP?



Overlapping subproblem

1. Recursion Approach ($O(2^n)$)

for i we have $n-i-1$. $\{i \to n-i-1\}$

for 0 " " $n-1$

1 " " $n-2$

2 " " $n-3$

~~Heuristic~~ →

```
for (i=0 to n-1){
    Cn += Cn-i-1 * Ci ;
}
```

Pseudocode →

```
catalan(n){
  if(n==0 || n==1){
    return 1;
  }
  int ans=0;
  for(i=0 to n-1){
    ans += catalan(n-i-1) * catalan(i);
  }
  return ans;
}
```

2. memoization O(n)

n → variable ∴ 1D array.

in main fn. → int[] dp = new int[n+1];
Arrays.fill(dp, -1);

Code → understand concept same as fibonacci

```
static int catalanMem(int n, int[] dp){
    if(n==0 || n==1){
        return 1;
    }
    if(dp[n] != -1){
        return dp[n];
    }
    int ans=0;
    for(i=0 to n){
        ans += catalanMem(i, dp) * catalanMem(n-i-1, dp);
    }
    return dp[n] = ans;
```

3. Tabulation

1. table → dp[n+1]
2. initialize → dp[0] = dp[1] = 1;
3. filling →

```
for(i=2 to n){
    for(j=0 to i){
        dp[i] += dp[j] * dp[i-j-1];
    }
}
return dp[n];
```

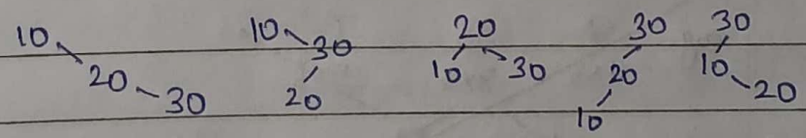
dp(2) = dp(0) * dp(1) + dp(1) * dp(0)

II. Counting Trees

Find all the BST with 'n' nodes.

n=3 (10,20,30)

Ans = 5



↓

works for any n=3 ans. 5.

BST

∴ check individually → $n=0$ 1 (null node)

$n=1$ 1 (root node)

$n=2$ 2 (10–20, 20–16)

$n=3$ 5

check yourself ly, $n=4$ 14

Hence,

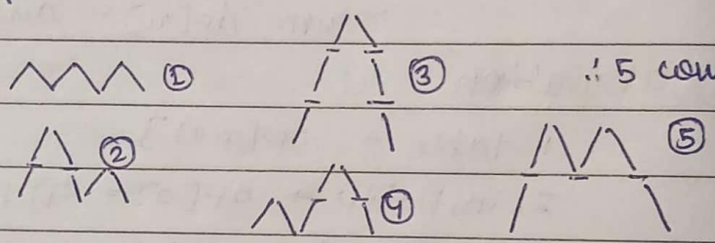
follows catalan no. ∴ [$n=5 \Rightarrow$ Catalan of 5 = No. of BST]

II. Mountain Ranges

Imp!- at any moment no. of down strokes can't be greater than no. of upstrokes.

/ = upstroke + \ = down stroke = 1 pair

for, pairs = 3 (should be mountains)



∴ 5 combination

Hence,

Pairs = $n \Rightarrow$ catalan of $n \Rightarrow$ combinations. Ans.

✱ Concept-5 : DP on grid → Matrix Chain Multiplication

I. mcm

To check multiplication of matrices in chain(ABCD) are feasible or not based on their orders and obtain the min. cost.

cost → $A_{2\times3}$ $B_{3\times4}$ → $2\times3\times4 = 24$.

e.g. A (1×3), B (3×5), C (5×6)

(AB)C

$(1\times5)(5\times6)$

$1\times5\times6$

$= 30$

A(BC)

$1\times3\times6$

$= 18$

e.g. A B C
1×3 3×5 5×6

((AB)C) (A(BC)) → Based on feasibility: combinations formed

A1 = AB: 1×3×5 = 15 (1×5) | B1 = BC: 3×5×6 = 90 (3×6)
A1·C: 1×5×6 = 30 | A·B1: 1×3×6 = 18
result = 15+30 | 108 = result
= 45

min. ∴ Ans.

∴ New, Q. arr[] = { 1, 2, 3, 4, 3 }

Matrices = A B C D → ★ Yes, Possible for MCM (Observation)
1×2 2×3 3×4 4×3

feasible combinations → (((AB)C)D) ((AB)(CD)) ((A(BC))D) .....

where, 1. A = arr[0] × arr[1]
2. B = arr[1] × arr[2]
⋮
i. Ai = arr[i-1] × arr[i]

i → starting from one, or we'll get negative index problem.

** Approach:-

If chain of matrix multiplication is possible, then make combination like this :-

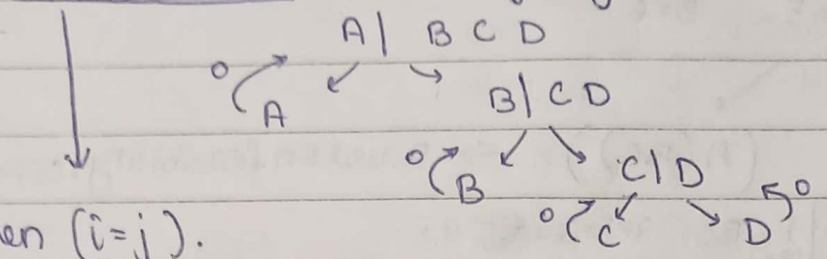
A1 | A2 A3 A4
i=1 | j=n-1
K=1

(i, K) (K+1, j)
set (2,4) set
A1 A2A3A4

∴ 1 combination will be → ((A1)(A2A3A4)) ↗ Inside recursive calls
Base condition → further subproblem in this too.

∴ For more combination proceed K → (i, K) , (K+1, j) , K++
set 1 set 2

1. Base condition → cost of single mcm is zero



when (i=j).

1. Recursion $O(2^n)$

main → arr[] = {1, 2, 3, 4, 3};

int n = arr.length;

print(mcm(arr, 1, n-1));

start from 1.

code → static int mcm(int[] arr, int i, int j){

if (i == j){

return 0;

}

i → k → j

int ans = Integer.MAX_VALUE; → ans = ∞

for (K = i; K <= j-1; K++){

check approach

Ai = arr[i-1] × arr[i]

Ak = arr[k-1] × arr[k]

result 1 :: Ai ..... Ak

Ak+1 = arr[k] × arr[k+1]

Aj = arr[j-1] × arr[j]

result 2 :: Ak+1 .... Aj

int cost1 = mcm(arr, i, k);

int cost2 = mcm(arr, k+1, j);

int cost3 = arr[i-1] * arr[k] * arr[j];

int finalCost = cost1 + cost2 + cost3;

ans = Math.min(ans, finalcost);

}

return ans;

∴ result 1 = arr[i-1] × arr[k] → socho starting se end tak kaisa ho jayega.

result 2 = arr[k] × arr[j] → " " " " " " " " "

cost3 = arr[i-1] × arr[k] × arr[j]

2. memoization (O(n×n)).

2D array since → i, j → ∵ 2 partitions.
(1 to k) (k+1, n-1)

arr[] = { 1, 2, 3, 4, 3 }

meaning of each (i,j) → e.g. (1,2)

1×2 2×3 ∴ 1×2×3 = 6
A1 A2 will be stored.

(Jis bhi cell (i,j) pe ho woha agar 'i' se 'j' tak matrix ko multiply karna aur us cell pe uska un saare mcm me se min. cost rakhna.

code → main : int[] arr = { };
int n = arr.length;
int[][] dp = new int[n][n];

filling 2D array all cells with -1 → for(i=0 to n){
Arrays.fill(dp[i], -1);
saare rows pe -1 store ho jayega.
∴ completely full.
print(mcmMem(arr, 1, n-1, dp));

mcmMem : static int mcmMem(int[] arr, int i, int j, int[][] dp){
base condition → if(i==j){ return 0; }.
values store karte jao if found → if(dp[i][j] != -1){
return dp[i][j];
int ans = Integer.MAX_VALUE;
same code for recursion ← for( ){ }
return dp[i][j] = ans;
}

(Tabulation is unintuitive for this)

0/1 knapsack variation

Ajanta
Page No.
Date

Minimum Partitioning / min. subset sum Diff. / Partitioning subsets.

numbers[] = {1, 6, 11, 5}

minDiff = 1 ans.

⇒ Aisa set choose karo jin do ke partition ke sum ka diff. min. aaye.

e.g. set 1 {11, 5} , set 2 {1, 6}

sum ↓ 16 , sum ↓ 7

Difference = 16 - 7 = 9

Ulty. others → {1, 11} and {6, 5} = |12 - 11| = 1 ✓ ans.

{1, 5} and {6, 11} = |6 - 17| = 11 .

minDiff = |sum1 - sum2|

We've to select/ignore element in such a way to get minDiff.

∴ 0/1 knapsack ⇒

valid → take, ignore

invalid

☆ (check greedy ques. (different hai wo))

Approach → To get min. make sure to put sum/2 elements or close to it in each set.

∵ Sum = 1 + 6 + 11 + 5 = 23 ∴ $\frac{sum}{2} \simeq 11.5$

∴ set 1 → (11, 1)
set 2 → (6, 5)
→ close to $\frac{sum}{2}$ gives min value.

Hence ∵ using conventional 0-1 knapsack process → val = elements of array
W = sum/2 .

∴ 2D dp array.

understand properly

Code →

```
static int minPartition( int[] arr){
    int sum=0;
    for ( i = 0 to arr.length){
        sum += arr[i]
    }
    int W= sum/2;
    int[][] dp = new int[n+1][ W+1];
    for ( i=1 to n+1) {
        for (j=1 to W+1) {
        if ( arr[i-1] <=j) {        // valid
        int include =  arr[i-1] + dp[i-1][j - arr[i-1]];
        int exclude =    dp[i-1][j];
        dp[i][j] =  max(include, exclude);
        else {                      // invalid
            dp[i][j] = dp[i-1][j];
    }}}
    int sum1 =  dp[n][W];
    int sum2 =  sum - sum1;
    return   Math.abs(sum1 - sum2);
}
```