Dynamic Programming

DP → enhanced Recursion

Identification

(to check if a problem is of DP or not)

- choice of including the current element
- optimization is asked.

Recursive

- memoization
- Top-down.

1) 0-1 knapsack (6)
2) unbounded knapsack (5)
3) Fibonacci (7)
4) LCS (15)
5) LIS (10)
6) Kandane's Algorithm (6)
7) Matrix chain multiplication (7) ⭐
8) DP on trees (4)
9) DP on grids (14)
10) Others (5)

0-1 knapsack

- subset sum
- equal sum
- count of subset sum
- minimum subset sum
- Target sum
- # of subset with given difference

knapsack

- Fractional knapsack → greedy
- 0/1 → add items as a whole
- unbounded knapsack → can add multiple instances of an element

I/P → weight [ ] →
value [ ] →
W → 7

O/P → max profit

choice → item → ✓ / ✗

choice diagram

for item 1 → item 1

- w1.weight < W → choice (✓ / ✗)
- w1.weight > W → ✗

```
int knapsack (int wt[], int val[], int cap)
// Base diagram
// think of smallest valid input
if (n == 0 || W == 0) return 0;

if (wt[n-1] <= W){
    return max( knapsack(wt, val, cap, n-1),
                val[n-1] + knapsack(wt, val, cap - wt[n-1], n-1));
}
```

max profit

int n

```
else return knapsack (wt, val, cap, n-1);
}
```

Memoization ⇒ Recursive code + 2 lines

this matrix will be have params which are changing in recursive call

dp → (n) × (W)

dp[n+1][W+1]

- size of array
- weight of knapsack

filling the matrix with -1

```
memset (dp, -1, sizeof(dp))
```

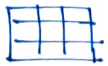
Before calling the Recursive function we check if the mat is already filled or not

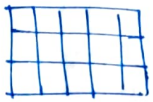
```
int knapsack (int wt[], int val[], int W, int n){
    if (n == 0 || W == 0) return 0;
    if (t[n][W] != -1) return t[n][W];
    if (wt[n-1] <= W){
        t[n][W] = max( val[n-1] + knp(wt, val, W - wt[n-1], n-1),
                       knp(wt, val, W, n-1));
        return t[n][W];
    }
    else{
        t[n][W] = knp(wt, val, W, n-1);
        return t[n][W];
    }
}
```

Top-down approach

Recursion
↓
memoization
↓
top-down

O(same)

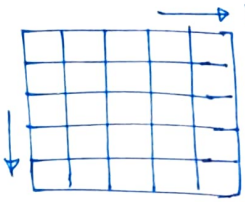
memoization
↓
Recur + 

Top down → 

no recursion

ans → O/P

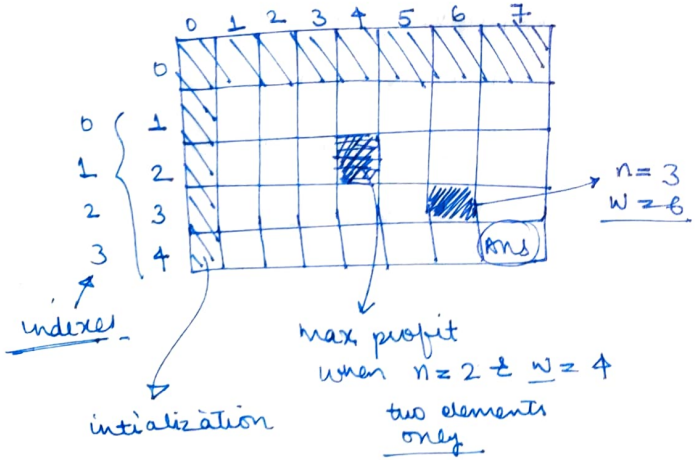
Stack overflow avoided → as not recursion calls are there.

matrix → 

step-1 → Initialize

step-2 → Recursive calls → Iterative version

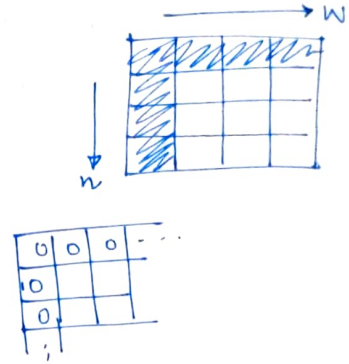
W = 7
N = ~~5~~ 4



Recursion → Iterative

Base condition → initialization
(Recur) (Top-down)

return 0 → n == 0 || w == 0 → 0



Recursive	Top-down
if (n == 0 || w == 0) return 0;	for (int i → n+1) for (int j = 0 → w+1) if (i == 0 || j == 0) dp[i][j] = 0;
if (~~val~~ wt[n-1] ≤ w) { return max(val[n-1] + kp(wt, val, w - wt[n-1], n-1), kp(wt, val, w, n-1))	if (wt[n-1] ≤ w) { t[n][w] = max(val[n-1] + t[w - wt(n-1)][n-1], ~~t[w][n-1]~~ t[n-1][w]) else t[n][w] = t[n-1][w].

if (wt[n-1] ≤ w) {
t[n][w] = max(val[n-1] + t[n-1][w - wt(n-1)] , t[n-1][w])

else
t[n][w] = t[n-1][w]

now → (n, w) → (i, j)
and run a loop

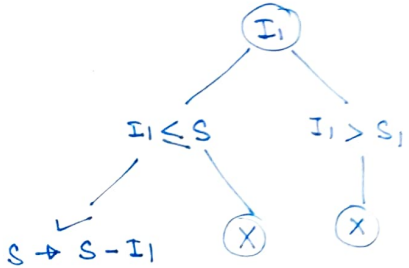
```
for (int i = 1; i < n+1; i++){
    for (int j = 1; j < n+1; j++){
        if (wt[i-1] <= j){
            t[i][j] = max( val[i-1] + t[i-1][j-wt[i-1]],
                           t[i-1][j] )
        else{
            t[i][j] = t[i-1][j];
    }.
Return t[n][w];
```

---

1) Subset Sum Problem

I/P → 2 3 7 8 10 → T/F ⇒ given an array, check if subset sum is sum

Sum → 11



(i, sum) ⟸

dp[n+1][sum+1]

```
bool dfs(int i, int sum, arr){
    if (i == arr.size())
        return false;
    if (sum == 0) return T;
    if (arr[i] <= sum){
        return dfs(i+1, sum - arr[i], arr)
            || dfs(i+1, sum, arr)
    }
    else
        dfs(i+1, sum, arr);
}.
```

```
bool dp[n+1][sum+1];
memset(dp, -1, sizeof(dp));
bool dfs(int i, arr, sum){
    if (i == arr.size()) return false;
    if (dp[i][sum] != -1) return dp[i][sum];
```

Initialization of matrix

n ↓ \ sum →	T	F	F	F - - - -
	T			
	T			
	T			
	T			

sum = 0, n = 0 → T

sum = S, n = 0 → F

sum = 0, n → n → T

Initialization

```
for (int i=0 → n)
    for (int j=0 → sum){
        if (i==0) dp[i][j] = False
        if (j==0) dp[i][j] = True.
}.
```

```
for (int i → 1 → n)
    for (int j = 1 → sum){
        if (arr[i] <= j){
            t[i][j] = t[i+1][j-arr[i]] ||
                      t[i+1][j];
        }
        else
            t[i][j] = t[i+1][j];
    }
}.
Return t[n][sum]
```

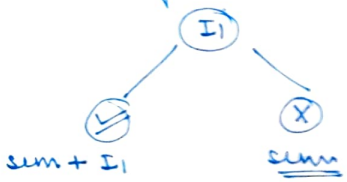
Equal sum Partition

arr[] → {1 5 11 5}.

O/P → (T/F) → divide array in two subsets such that both have equal sum

sum → sum(array) = 22

at any element



S_1 S_2 → initially all items are here

bool subsetSum (int arr[], int sum, int n){

→ $S_1 + S_2 \rightarrow$ sum

as $S_1 = S_2$

$2S$ = sum — hence sum must be even

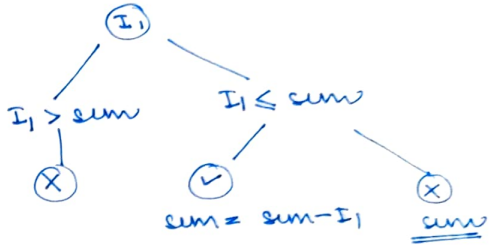
⇒ we just have to find a subset with sum of sum/2

return subsetSum (arr, i, sum/2);

Count of subset of given sum :-

I/P → arr[] → 2 3 5 6 8 10

sum → 10



dp[n+1][sum+1]

n = 0 → F

sum = 0 → T

```
int dfs ( int i, arr, sum) {
  if ( i == arr.size()) return 0;
  if (dp[i][sum] != -1) return dp[i][sum];
  if (sum == 0) return 1;
  if (arr[i] ≤ sum)
    dp[i][sum] = dfs (i+1, arr, sum - arr[i])+
                 dfs (i+1, arr, sum);
  else dp[i][sum] = dfs (i+1, arr, sum);
  return dp[i][sum];
}
```

→ sum

n ↓	0	0	0	0	0	0	0	0
1								
1								
1								

```
for (int i → 1 → n) {
    for (int j = 1 → sum)
      if (arr[i] ≤ sum) {
          dp[i][j] = dp[i+1][j-arr[i]] + dp[i+1][j];
      }
      else
          dp[i][j] = dp[i+1][j];
}
```

Minimum subset sum difference

arr[] → [1 | 6 | 11 | 5]

O/P → 1

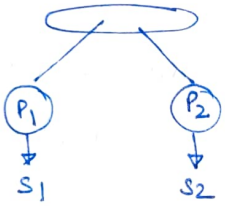
{1, 6} {11, 5}.

abs (16 − 7) → 9.

{1, 6, 5} {11}.

(2) (11)

O/P → abs (12 − 11) = 1



abs (S1 − S2) → min

previous question

abs (S1 − S2) = 0


```
int dfs (int i, vector arr, int S1) {
    S2 = sum - S1;
    if (i == arr.size()) return INT_MAX;
    return
        min ( dfs(i+1, arr, S1 + arr[i]),
              dfs(i+1, arr, S1) )
}.
```

S2 → S2 − S1 → (min)

(Range − S1 − S1) → min

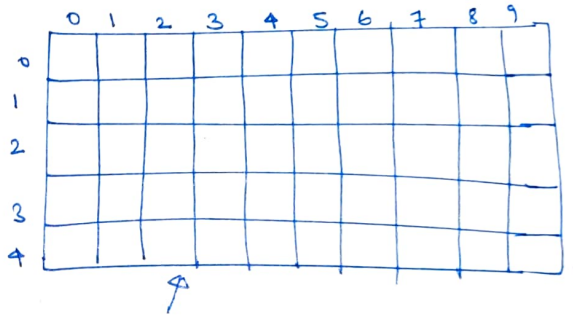
Range − 2S1 → min

As we have to return abs then S2 − S1

S1 always smaller

we have to calculate maximum value

S1 < range/2



for range/2 the maximum val of s for which the cell is T is the value of S1.

we'll see this row as it will give the value of possible subset sums. where num of el = n

⇐ we can solve this by subset sum problem

Count the # of subsets with given difference.

arr[] → [1 | 1 | 2 | 3]

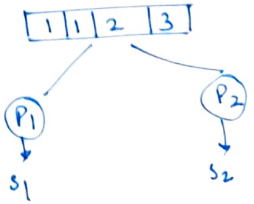
diff → 1

O/P = 3

{1, 2} {1, 3}

{1, 3} {1, 2}

{1, 1, 2} {3}.



S1 − S2 = d

S1 + S2 = arr.

& 2S1 = d + arr

S1 = (d + arr)/2

$$sum(s_1) = \frac{d + sum(arr)}{2}$$

problem reduced to count of subset sum where $\boxed{s = \frac{d + sum(arr)}{2}}$

Target sum

arr :- | 1 | 1 | 2 | 3 |

sum :- 1

o/p → 3.

→ assign either +/- in front of every element & then check the result

Ex:- ~~+1-1+2~~ +1-1-2+3

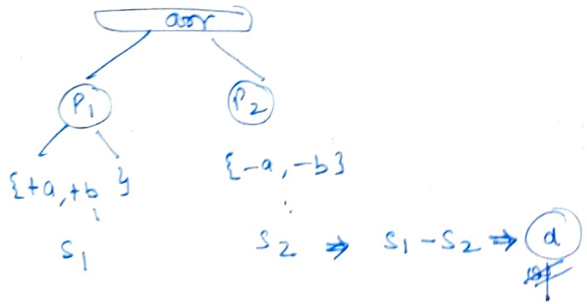
= 1

+ 1 -1-2+3

= 1

I^st approach → dfs(i+1, nums, sum+arr[i]);
dfs(i+1, nums, sum-arr[i]);

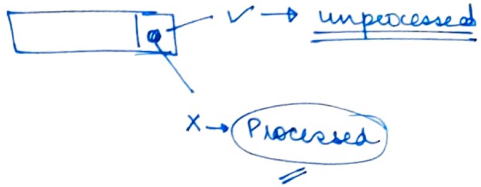
II^nd approach → same as previous one



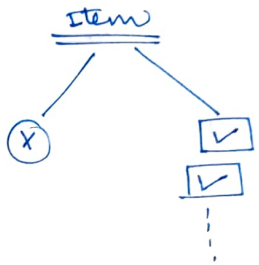
Unbounded Knapsack

↓
multiple occurances of same item is allowed.

variations
- Road cutting
- coin change-1
- coin change-2
- Maximum Ribbon cut.



previously
- either item is taken or not it is taken as considered (processed)
- in unbound if the item is not considered then only it is processed otherwise it is in unprocessed state.



return $\max\left(\begin{matrix} val[i] + dfs(i-1, cap - wt[i]), \\ dfs(i-1, cap) \end{matrix}\right)$

↓ unbound.

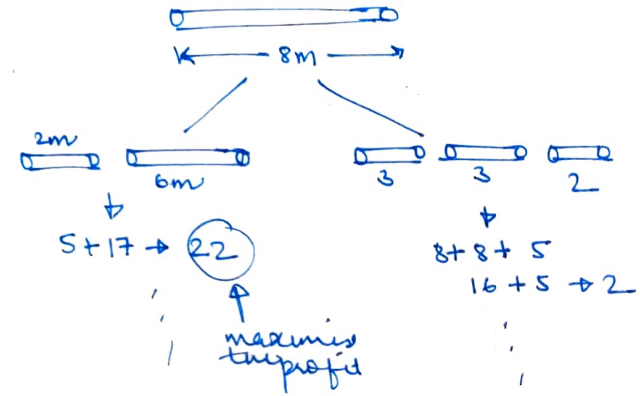
$\max\left(\begin{matrix} val[i] + dfs(i, cap - wt[i]), \\ dfs(i-1, cap) \end{matrix}\right)$ ⇒ $t[i][j] =$

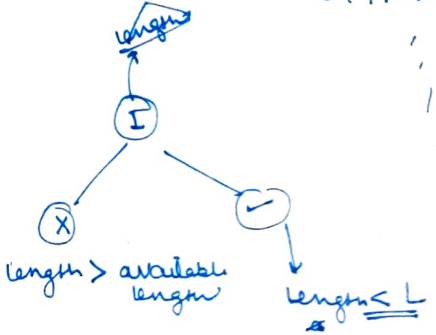
$\max\left(\begin{matrix} t[i][j - wt[i]] + val[i-1], \\ t[i-1][j] \end{matrix}\right)$

Rod-Cutting Problem

length[] → 1 2 3 4 5 6 7 8
price[] → 1 5 8 9 10 17 17 20

L → 8





↳ we can include any piece two or more times

length → 1 . . . . n
price → P1 . . . . Pn

```
dfs (i, L) {
if (length
```

```
global → len[], price[]
int dfs(i, L) {
    if (i ≥ len.size()) return 0;
    if (len[i] ≤ L) {
        dfs. return
    max( dfs(i, L - len[i]) + price[i]
       , dfs(i-1, L)
    else
        return dfs(i-1, L)
```

Coin Change - I # of ways

coin[] → 1 | 2 | 3

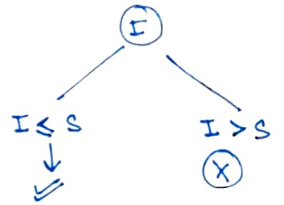
sum → 5

1+2+2
2+1+1+1
3+2
1+1+1+1+1
3+1 1

sum = 5

O/P → 5 → 5 number of ways to produce 5 as output

unbounded
↳ because we can have multiple instance of single coin




```
int dfs ( int i, int [] arr, S) {
    if (arr[i] > S) return 0;
    return max ( dfs (i, arr, S - arr[i]) +
                 dfs (i+1, arr, S)
```

```
int dfs (int i, int [] arr, S) {
    if (i ≥ arr.size()) return 0;  if (S == 0)
                                       return 1
    if ( arr[i] ≤ S) {
        return  dfs (i, arr, S - arr[i]) +
                dfs (i+1, arr, S)
    else
        return dfs (i+1, arr, S);
```

using DP

→ sum.

1	0	0	0 - - - -
1			
1			
⋮			

↓ size

if (wt[i-1] ≤ j)

t[i][j] = t[i][j - arr[i-1]] + t[i-1][j]

else

t[i][j] = t[i-1][j];

Coin Change - II

→ min # of coins

coin[] → |1|2|3|/|/|

sum → 5

o/p → min # → 2

(2+3)
1+2+2
1+1+1+2.
1+1+3
1+1+1+1+1

min = 2

(w)

→ sum

0	∞	∞ - - - - - -
0		
0		
0		

n ↓

Longest Common Subsequence

I/p → x: a b c d e f h

y: a b e d f h r

lcs → abdh

→ o/p → 4

subsequence
we can skip element

substring
we have to choose continuous

```
int dfs (string x, string y, i, j) {

    if (i >= x.length() || j >= y.length())
        return 0;

    if (x[i] == y[j]) {
        return 1 + dfs (x, y, i+1, j+1);
    }
    else {
        return max ( dfs (x, y, i+1, j),
                     dfs (x, y, i, j+1) );
    }
}
```

added 1 and call next

---

memoized

→ build an array / vector of longest inp + 1

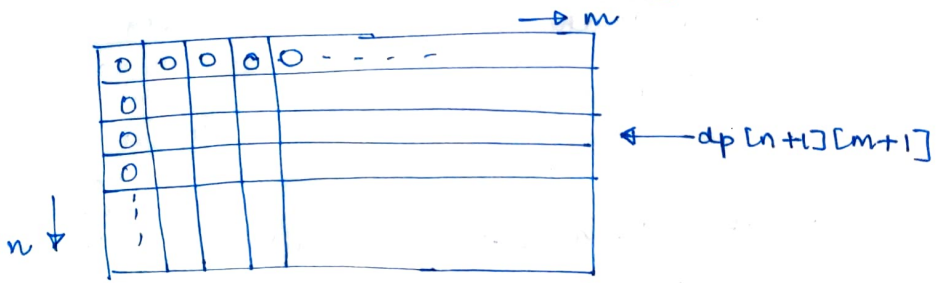
→ vector<vector<int>> memo (n+1, vector<int> (m+1, -1));

n → x.length()

m → y.length()

Top down DP

X : a b c d f → abcf
Y : a c b c f → abcf




```
if (x[i-1] == y[j-1]) {
    t[i][j] = 1 + t[i-1][j-1];
}
else {
    t[i][j] = max(t[i-1][j], t[i][j-1]);
```

Longest common substring

X → abcde
Y → abfde

must be contagious

O/P → 2

```
int dfs(string x, string y, int i, int j) {
    if (i >= x.length() || j >= y.length()) return 0;
    if (x[i] == y[j]) {
        int temp = 1 + dfs(i+1, j+1);
        res = max(res, temp);
    }
    else
        ret dfs(i+1, j+1)
}
```

```
int dfs(string x, string y, int i, int j, int cnt) {
    if (i >= x.length() || j >= y.length()) return cnt;
    if (x[i] == y[j]) {
        cnt = max(dfs(i+1, j+1, cnt+1),
    }
    else
        count = max({count,
                     dfs(i+1, j, 0),
                     dfs(i, j+1, 0)});
    return count;
}
```

Print longest common subsequence

X : a c b c f
Y : a b c d a f

X : a c b c f
Y : a b c d a f

O/P → abcf

		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1
c	2	0	1	1	2	2	2
b	3	0	1	2	2	2	2
c	4	0	1	2	3	3	3
f	5	0	1	2	3	3	4

we'll do the reverse engineering

start with i = n & j = m.

```
if X(i-1) == Y(j-1) {
    i--;
    j--;
} else {
    if(
```

if equal i, j → i--, j--

else i, j → max(i-1, j ; i, j-1)

```
int i = m; j = n;
while (i > 0 && j > 0) {
    if (a[i-1] == b[j-1]) {
        s.push_back(a[i-1]);
        i--; j--;
    }
    else {
        if (t[i][j-1] > t[i-1][j])
            j--;
        else i--;
    }
}
reverse(s.begin(), s.end());
}
```

Shortest common supersequence

a: "geek"
b: "eke"

merge → geek, eke → geeke

geeke

a:- AGGTAB
b: GXTXAYB

→ AGGTGXABTXAYB

not necessarily ~~contagious~~ continous.

both sequences are present in this string

AGGXTXAYB

→ length → 9

Shortest supersequence.

AGGXTXAYB

GTAB

we have used this only once as it is present in both.

In other terms, it is called as LCS

to calculate the length we can just

n+m - ~~xx~~ (LCS)

Shortest length → m+n- LCS

Minimum # of Insertion and Deletion

a → b

perform insert and delete on string a to convert it to 'b'

a → heap
b → pea

p heap → pea

↓

pea

Insert p
Remove h
Remove p

Ins : 1
del : 2

⊙ If we note the special point

⊙ ea remains intact as this is the

LCS

a → b

LCS (ea)

heap → pea

2D

1I

LCS
ea

~~D~~ # of deletions = a.length() - LCS.length

# of insertion = b.length() - LCS.length

Longest Palindromic Subsequence

I/P → S: agbcba

we can have discontinous/ continous sequence → that is a palindrome

abcba
→ bcb
→ b.

Longest

O/P → 5

⊙ we can find the longest common subsequence of s and reverse (s).

LPS(a) = LCS(a, reverse(a))

Minimum # of deletions in a string to make it Palindrome

agbcba

~~agbcba~~ → bcb

~~agbcba~~ → c

agbcba → abcba

Minimum # of deletion = s.length - LPS(s)

Print SCS → shortest ~~#~~ common supersequence

aebcf

abc daf

ac bcdaf

worst case → m+n

better → m+n - LCS

↓

length

	ϕ	a	b	c	d	a	f
ϕ	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

abcdaf
acbcf

LCS → abcf

```
while (i>0 && j>0){
    if (a[i-1] == b[j-1]){
        i--; j--;
        res.push_back(a[i-1]);
    }
    else {
        if (t[i-1][j] > t[i][j-1]){
            res.push_back(a[i-1]);
            i--;
        }
        else {
            res.push_back(b[j-1])
            j--;
        }
    }
```

```
while (i-->0){
    s.push_back(a[i-1])
}
while (j-->0){
    s.push_back(b[j-1])
}
reverse(res)
```

Longest Repeating subsequence

ctr = "AABECDD"

we have to find a subsequence in this string which is repeating

A A B E B C D D → ABD

A B E C D → repeating ABD

O/P → ABD ⇒ length = 3

we will find LCS among A and dup(str)

A A B E B C D D
A A B E B C D D

while including in result

if a[i-1] == b[j-1] && at i == j
→ we will not include it

a[i-1] == b[j-1] && i != j ⇒ included

```
if (a[i-1] == b[j-1] && i != j) {
    t[i][j] = t[i-1][j-1] + 1;
}
else
```

$$t[i][j] = \max\begin{pmatrix} t[i][j-1], \\ t[i-1][j] \end{pmatrix}$$

Sequence Pattern Matching

I/P → a : "AXY"
b : "ADXCPY"

if A is a subsequence of B

we can run two loops and check if it is equal or we can just try LCS

LCS("AXY", "ADXCPY") == "AXY"
↳ true
else false

Minimum Number of insertion to make it palindrome

S : a e b e b d a

S → aebcbda

a d e b c b d a

we'll find the LCS and the remaining elements (which are not included in LCS) are those which do not have a pair

hence we'll add

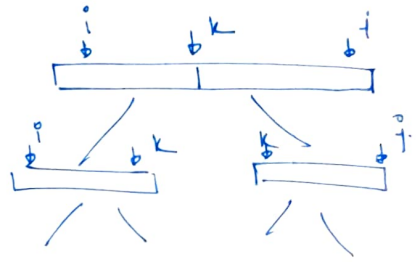
s.length − (LCS.length) characters

Matrix Chain Multiplication

1) MCM
2) Printing MCM
3) Evaluate Exp to True / Boolean Parenthesis
4) Max/min value.
5) Palindrome partitioning
6) Scramble string
7) Egg dropping.

Identification + formal

breaking the string at any point



fun(i, j)
fun(i, k) fun(k, j)

```
int solve (int arr[], int i, int j){
    if (i > j) return 0;
    for (int k = i ; k < j ; k++){
        temp ans = solve (arr, i, k)
                   ⊕ ← (any operator)
                   solve (arr, k+1, j)
    }   ans = fun (temp ans);
}
```

Matrix chain multiplication

Cost of multiplying two matrices

A_1 → $a \times b$, A_2 → $b \times c$

$\downarrow$

$a \times b \times c$

arr[] → | | | | |

$a_1\ a_2\ a_3\ a_4$

$((a_1 a_2)\ (a_3 a_4))$

$((a_1 (a_2 a_3)) a_4)$

A → 5 * 30

B → 30 * 7

C → 7 * 10

ABC → | 5 | 30 | 7 | 10 |

AB → (5 × 30 × 7)

= ~~5×~~ 1050

5 × 7

Return minimum cost after multiplications

arr[] → 40 (0) 20 (1) 30 (2) 10 (3) 30 (4)

no of matrices = a.size() - 1

a_1 → 40 × 20

a_2 → 20 × 30

a_3 → 30 × 10

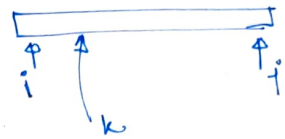
a_4 → 10 × 30

A_i → arr[i-1] * arr[i]

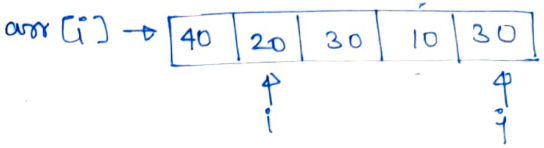
(A1)(A2 A3 A4)

k

min cost , min cost



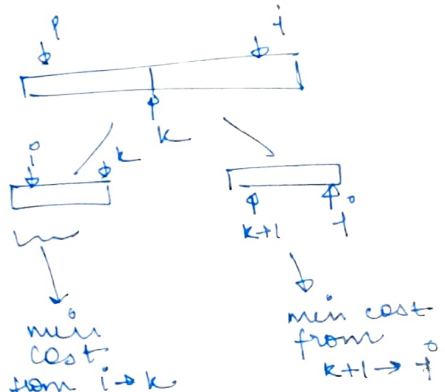
$i+1 \le k < j$



A_i → arr[i-1] × arr[i]

```
int solve(int i, int j, int arr[]) {
    if (i >= j) return 0;
    // move k i -> j
    for (int k = i; k < j; k++) {
        solve(arr, i, k);
        solve(arr, k+1, j);
    }
```

we have ensured that a group contains atleast 1 matrix ...

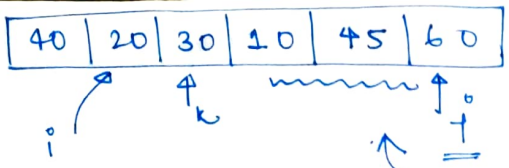


(AB)(CD)

min cost of multiply AB

min cost of multiply (CD)

and then multiply both resultant matrix



min cost of multiplication of i → k

(40 x 20)
(20 x 30)

min cost of multiplication of

30 x 10
10 x 45
45 x 60

temp ans 1 = solve (i, k)
temp ans 2 = solve (k+1, j)

sum → temp ans 1 + temp ans 2 +

dimensions of matrix

40 x 30
i x k

arr[i-1] x arr[k]

dimensions of res mat.

arr[k] x arr[j]

arr[i-1] x arr[k] x arr[j]

temp_res = temp 1 + temp 2 + arr[i-1] x arr[k] x arr[j].

```
int solve (int arr[], int i, int j) {
    if (i >= j) return 0;   int mn = INT_MAX;
    for (int k = i; k <= j-1; k++) {
        int temp ans = solve (arr, i, k) +
                       solve (arr, k+1, j) +
                       arr[i-1] * arr[k] * arr[j];
        if (temp < mn) mn = temp;
    }
    return mn;
}
```

memoization

matrix → int memo[i+1][j+1]

size of array | size of array

= int memo[n+1][n+1];

→ if (dp[i][j] != -1) return dp[i][j]
→ dp[i][j] = temp_res
→ return dp[i][j];

Palindrome Partitioning

s: nitin

n | iti | n

palindromes

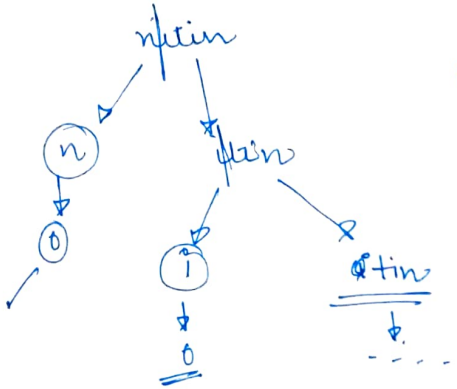
no of partitions → 2

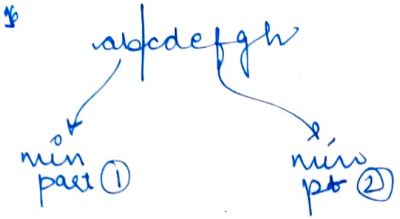
min

worst case

partition after every character as every letter is a palindrome

n i t i n → 4





1 + mp1 + mp2

whenever we find a palindrome string we need 0 partitions

```
int solve (int arr[], int i, int j){
    if (i >= j) return 0;   int res = INT_MAX;
    if (isPalindrome(arr,i,j)) return 0;
    for (int k = i; k <= j-1; k++){
        int temp1 = solve(i,k);
        int temp2 = solve(k+1,j);
        int temp_res = 1 + temp1 + temp2;
        res = min(res, temp_res);
    }
    return res;
}
```

Optimization → after memoization

There is a probability that one recursive problems i.e solve (i, k) or solve (k+1, j) might get solved previously.

```
int temp = 1 + solve(s,i,k) + solve(s,k+1,j);
```

~~if (solve(s,i,k) == -1) left = 0~~

```
if (dp[i][k] != -1) left = dp[i][k];
if (dp[k+1][j] != -1) right = dp[k+1][j];
```

Storing values if already present

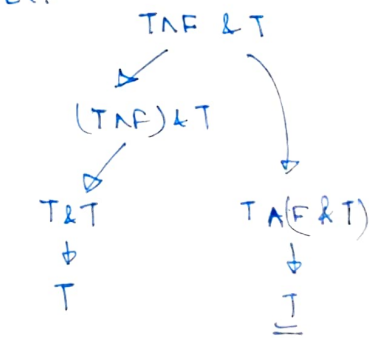
Evaluate Expressions to True | Boolean Parenthesis

string → T | F & F

symbols can be of T, F, |, &, ^

put brackets in a way so that the evaluated string becomes true.

Ex:-



T | F & T ^ F

↓

(T | F) & (T ^ F)

T & T

↓

T

T | (F & (T ^ F))

T | (F & T)

↓

T

* We'll put the position of k in such a way that k-1 we & k+1 we have brackets

T | (F & T ^ F

↑ k

T or F and T xor F

↑ k

exp^r (i to k-1) XOR exp^r (k+1 to j)

We'll always put a bracket on operators

To jump on operators always k = k+2

1) Find i & j

i = 1 , j = n-2

n → s.length()

2) Find base condition

~~if (i >=~~ if (i > j) return False

if (i == j) {

if (isTrue == true)

return s[i] == 'T';

else

return s[i] == 'F';

T

if we wanted true only return T else F

we have to find both

1) when the expression is evaluated true
2) when the expression is evaluated false

exp1 ^ exp2

T ^ F → T

F ^ T → T

exp 1: T → 2, F → 4

exp 2: T → 3, F → 5

2 x 5 + 4 x 3

10 + 12 = 22

so for doing function call, we'll pass parameter (T/F)

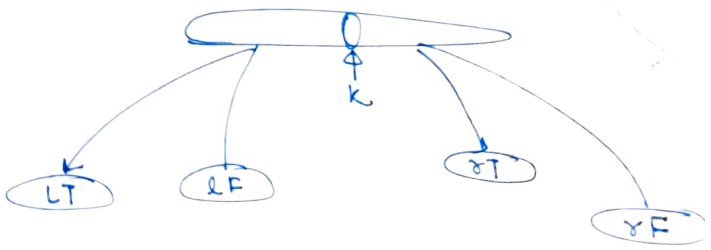
solve (i, j, T)

solve (i, j, F)

```
for (int k = i+1 ; k <= j-1 ; k = k+2){

    int lT = solve (i, k-1, T);
    int rT = solve (k+1, j, T);
    int lF = solve (i, k-1, F);
    int rF = solve (k+1, j, F);

    // answer logic
```



```
if (s[k] == '&') {
    if (isTrue == true){
        ans = ans + (LT * RT);
    else
        ans += (LF * RT + LF * RF + LT * RF);
}
else if (s[k] == '|') {
    if (isTrue) {
        ans += ( LT * RT +
                 LT * RF +
                 LF * RT );
    }
    else
        ans += LF * RF;
}
else {if (s[k] == '^') {
    if (isTrue) {
        ans += ( LF * RT +
                 LT * RF );
    }
    else
        ans += ( LF * RF +
                 LT * RT );
}
}
Return ans;
```

memoization — 1D | 2D | 3D

↳ dimension depends on number of variables changing in function calls

Here variables (changing) → i, j, isTrue
n, n, 2

dimensions → [n+1][n+1][2]

Better way

we can use map

map
i+j+isTrue

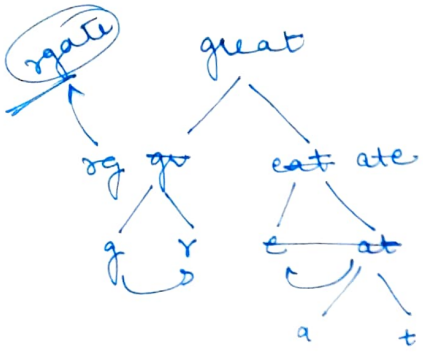
key = to_string(i) + to_string(j) + to_string(isTrue);
= "5+9+F"
↳ key of the map

Scrambled string

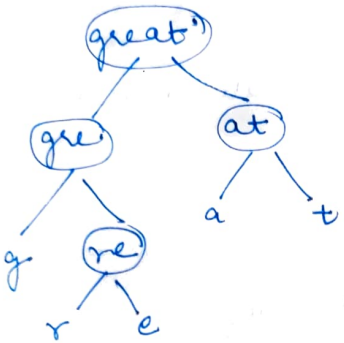
I/P → a: "great" b: "rgeat" O/P → True

We represent the string as a binary tree by partioning it into two non-empty string recursively.

Scrambled string → to generate a scrambled string we can choose any two non-leaf node and swap them

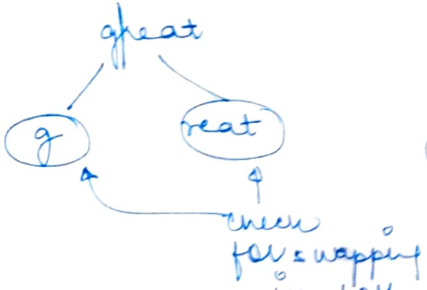


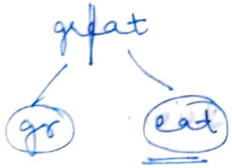
1) Binary tree only
2) No empty string child



Non-leaf node can be swapped by children

great
rgate
> Scrambled strings (T)





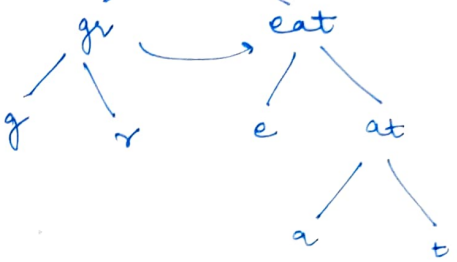
K = i+1 → n-1

gfkfaft

swapping → zero or more

i = 2

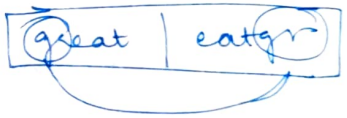
great ⇒ eatgr



we can have 2 case at any i
- either swap
- not swap

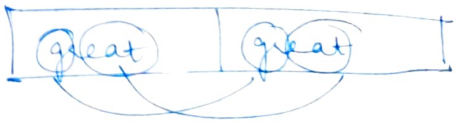
great → great swap X
great → eatgr swap ✓
> return True in both cases

if swapped →

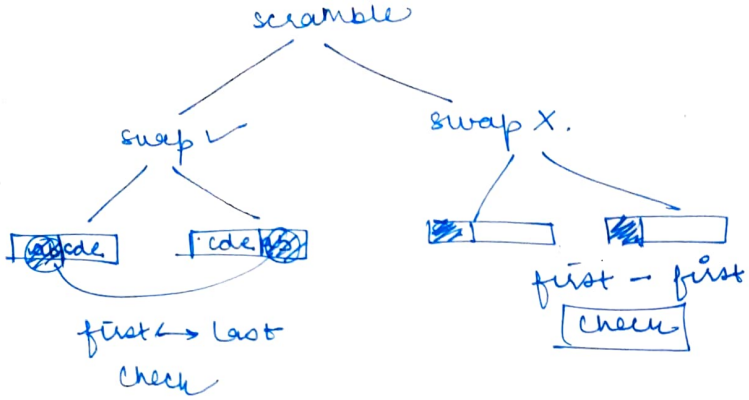


we'll check the first k letters of string with last k if it's scrambled.

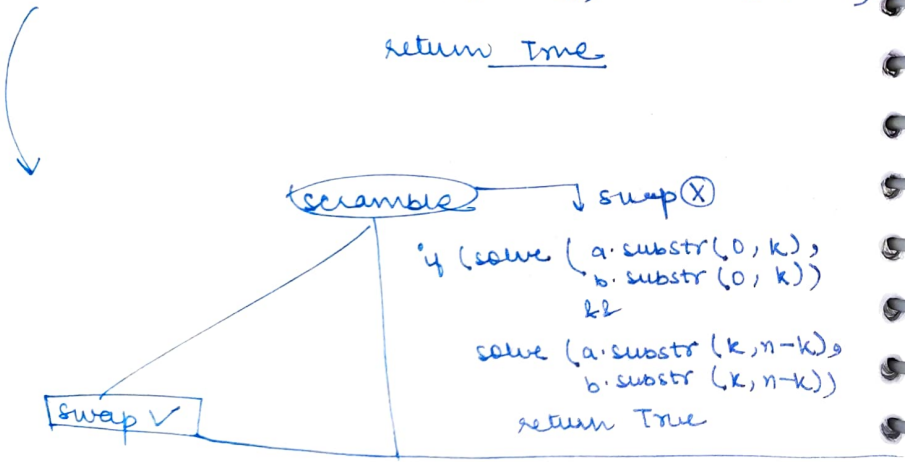
not swapped →



check in order only




```
if (solve (a.substring (0, k), b.substr (n-k, k)) {
    &&
solve (a.substr (k, n-k), b.substr (0, n-k))
return True
```



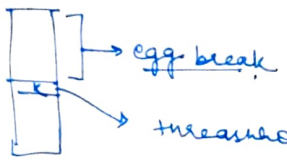

```
if (solve (a.substr (0, k), b.substr (n-k, k))
    &&
solve (a.substr (k, n-k), b.substr (0, n-k))
return True;
```

```
bool solve (string a, string b) {
    if (a.compare (b) == 0)
        return True;
    if (a.length() <= 1)
        return false;
    int n = a.length();
    bool flag = false;
    for (int i = 1; i < n; i++) {
        if (cond1 || cond2) {
            flag = true
            break;
        }
    }
    return flag;
}
```

memoization → use map → key can be
a + '' + b
a + "$" + b
a + "#" + b

Egg Dropping Problem

e = 3
f = 5



minimum number of attempts to find critical floor.

- we drop egg from every floor and check if it breaks.

we have to use 'e' number of eggs wisely so that in minimum attemps we can find the critical floor.

OR

suppose no of floor >>> number of eggs.

in this case, if we don't use the eggs wisely we won't be able to find the critical floor after using all the eggs also

Also

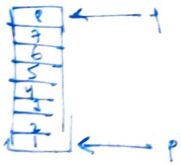
suppose we'll start from bottom we can find the critical floor in 1 egg only but the attempts → huge

-) use egg wisely & minimize # of attemps.

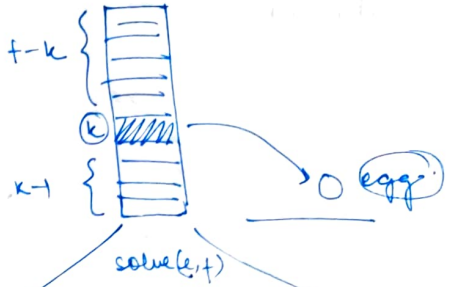
arr

string

f → number of floors



Basically we don't know from which floor I should make my first attempt.



min number of attemps
↓
worst case

Break ✓ (k)

it means that the threshold floor must be somewhere between (f-(k-1))

↓

solve(e-1, k-1)

Break X

solve(e, f-k)

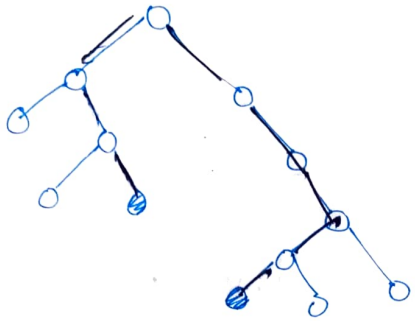
egg will only drop after k

```
int solve (int e, int f) {
    if (f == 0 || f == 1) return f;
    if (e == 1) return f;
    int mn = INT_MAX;
    for (int k = 1; k <= f; k++) {
        int temp = 1 + max (solve(e-1,k-1)
                            solve(e, f-k))
        mn = min (mn, temp);
    }
    return mn;
}
```

DP ON TREES

diameter of tree

maximum path between two leaf nodes



identification

↳ we're traversing a tree and on each node we again need to call recursive functions.

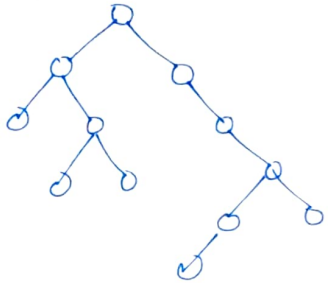
General Syntax

```
int solve ( Node *root, int *res ){
    if (root == NULL) return 0.
    int l = solve (root → left, res);
    int r = solve (root → right, res);
    int temp = calculate temp ans
    res = max( res, temp )
}
```

hypothesis

question dependent

Diameter of Binary Tree



```
int res = INT_MIN;
int solve (Node *root) {
    if (root == NULL)
        return 0;
    int l = solve (root→left);
    int r = solve (root→right);
    int temp_ans = 1 + l + r ;
    res = max (res, temp_ans);
    return 1 + max (l, r) ;
}
```

Maximum Path sum

Maximum path sum from any node to any node

↓

question becomes different when value of node becomes negative

if the value of 'l' or 'r' comes out to be negative then we won't include that

max (max (l, r) + root → val , root → val .)

```
int solve (Node *root, int &res){
    if (root == NULL) return 0;
    // getting l & r
    int temp = max ( root→val + max(l, r) , root→val )
    res = max (temp, res)
    return temp;
}
```

int ans = max(temp, l+r+root->val)

int ans = ??

Maximum Path sum from leaf to leaf

```
int solve (Node *root) {
    if (root == NULL) return 0;
    int l = solve (root->left);
    int r = solve (root->right);
    int temp_ans = max (l, r) + root->val;
    or  res = max (res, root->val + l + r);
    int ans = max (temp_ans, l+r+root->val);
    res = max (ans, res);
    return temp;
}
```