Dynamic Programming

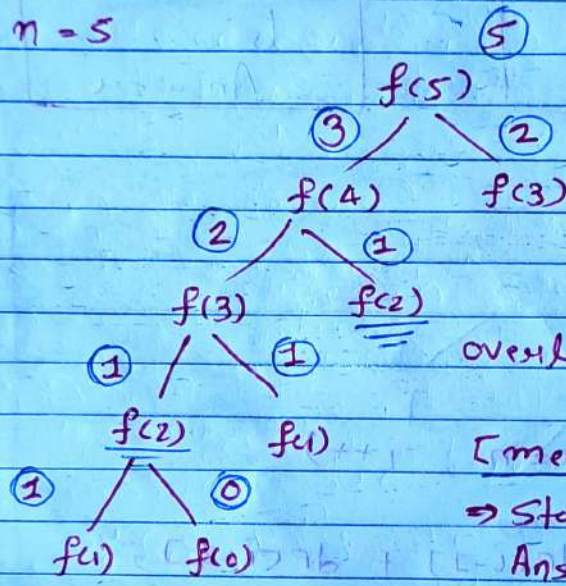
Date
Page

1) Fibonacci Sequence

⇒ 0, 1, 1, 2, 3, 5, 8, 13, 21 ...

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

⇒ n = 5



Overlapping Subproblem

[memoization]

⇒ Store Value of Previous Answer

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

⇒

```
f(n)
    if (n <= 1)
        return n;
    return f(n-1) + f(n-2);

    dp[n] = f(n-1) + f(n-2);

if (f(n) != -1)
    return f(n);        // Direct return
```

→ TC() → O(n)
SC() → O(n) + O(n)

Recursion → Tabulation
(Top down) (Bottom up)

Answer ↳ Base Case	Base Case ↳ Answer

⇒ Tabulation

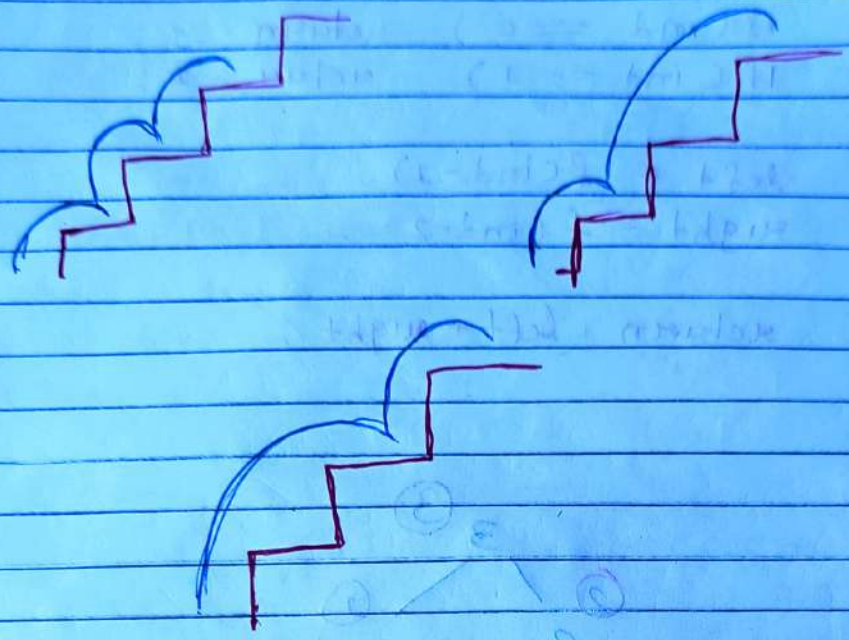
```
dp[0] = 0
dp[1] = 1
for (i=2 ; i<=n ; i++)
  {
    dp[i] = dp[i-1] + dp[i-2];
  }
```

TC → O(n)
SP → O(n) (No array)

→
```
Prev_2 = 0
Prev   = 1
for (i=2 ; i<=n ; i++)
  {
    Curr = Prev_2 + Prev;
    Prev_2 = Prev
    Prev = Curr          //return
  }                      Prev;
```

SP → O(1)

Distinct ways to reach n^{th} Stair



→ You can take either 1 or 2 steps

→ How can Identify

- Count total ways
- Find out best way from All Possible

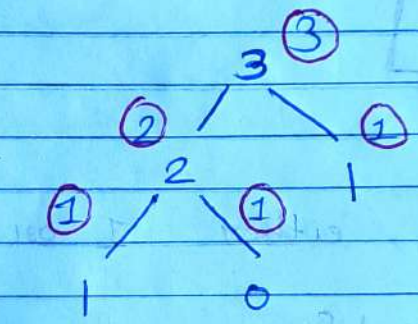
○ Short cut

→ Try to represent Problem in index term

→ Do all Possible stuff According to Problem

→ Count — Sum
min — min of all

```
f(index) {

    if(ind ==0)  return 1;
    if(ind ==1)  return 1;

    left = f(ind-1);
    right = f(ind-2);

    return left+right;
```



Frog Jump

⇒ Frog at → 0
wants to go → n

Can Jump 1 or 2 step

→ energy For Jump is Value at Curr stair − Value at new Stair

→ return min energy

→ Greedy won't work here

⇒ Return

0	1	2	3	4th Stair
10	20	30	10	3rd Index

→ fun-1) ⇒ min energy for reach n-1 From 0

```
f(Ind) {

    if (Ind ==0) return 0;
    left = f(Ind-1) + abs[a[Ind] -
                          a[Ind-1]);
if Ind>1
    right = f(Ind-2) + abs[a[Ind] -
                           a[Ind-2]);

    return min(left, right);

}
```

⇒ There are overlapping Subproblems

Do memoization

Follow up Using K

Jump Can be

i+1
i+2
:
i+K

f(Ind) {

if(Ind == 0) return 0;
min_steps = Int_MAX;
for (j=1 ; j<=k ; j++)
{
Jump = f(Ind-j) + abs(a[ind] - a[Ind-j])
min_steps = min (min_steps, Jump);
}
return min_steps;
}

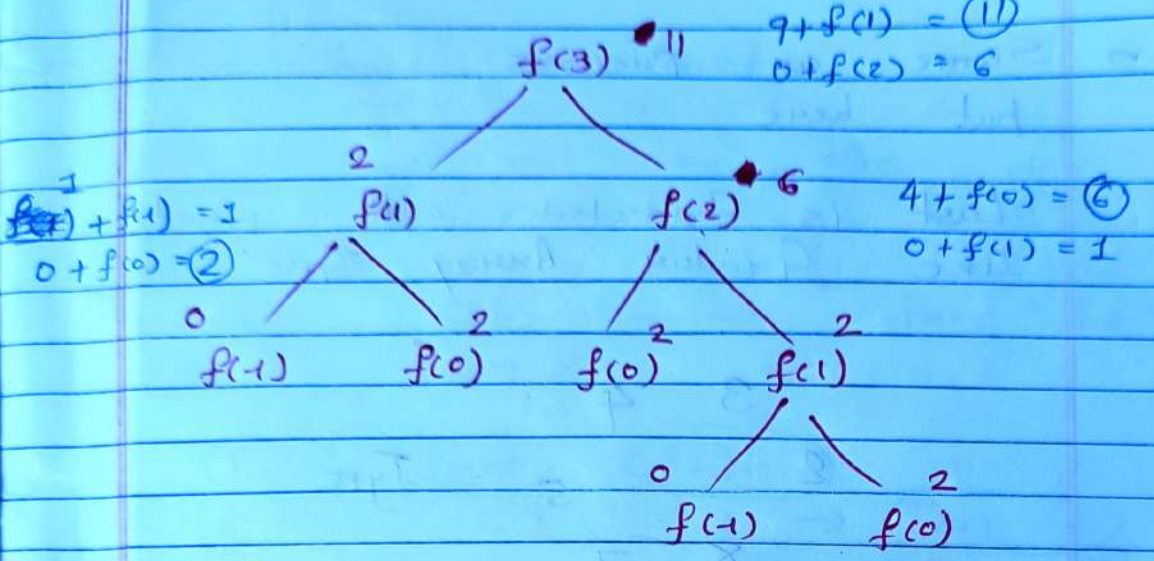
Maximum Sum of adjacent (non) elemen

⇒ Check all Subsequence / recursion

f(Ind)
{
if (Ind == 0)
return a[ind];
if (Ind < 0) return 0;
Pick = a[ind] + f(ind-2);
notPick = 0 + f(ind+1);

return max (Pick, notpick)

[2, 1, 4, 9]
0 1 2 3



Tabulation :

```
dp[0] = a[0];
int neg = 0;

for (i=1; i<n; i++)
{
  Take = a[ind] + dp[i-2];   if(i>1) else + 0;
  nonTake = 0 + dp[i-1];
  dp[i] = max(Take, nonTake);
}
```

Space optimization

```
Prev = a[0]
Prev2 = 0;
Curr =
```

→

```
Prev = Curr
Prev2 = Prev
Curr =
```

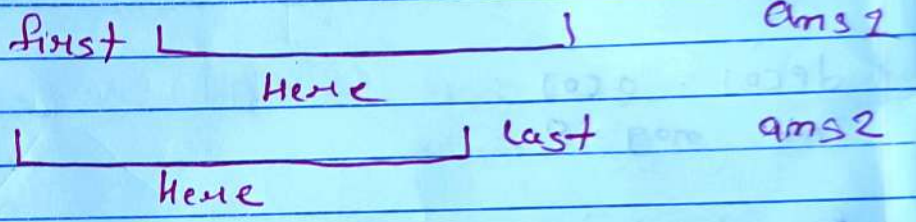
House Robber

→ Same as Previous but here

Last is Connected with first like Circular Array Type

3 4
2 5 = Type
8 7

→ Apply same logic



→ Max (ans1, ans2)

Ninja Training

→ Ninja have 3 activity

He can do one at one day and gain point
He can't do a activity he did yesterday Calc max points he can gain

Run	Fight	Practice	
10	50	1	→ Day 0
5	100	11	→ Day 1

→ Greedy fails
Try all by Recursion

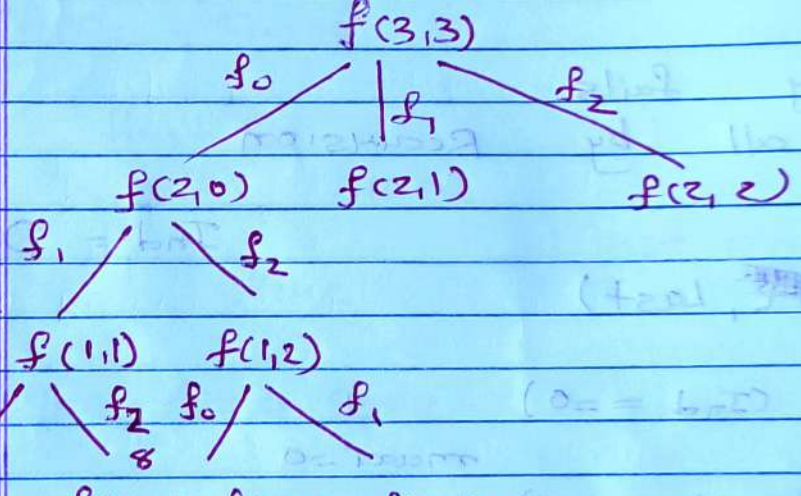
Ind = Day

```
f(Ind, Last)
 {
   if (Ind == 0)
   {                    maxi = 0
    for (i=0 → 2)
        if (i != last)
            maxi = max(maxi, task[0][i])
   return maxi;
 }
 maxi = 0;
 for (i=0 → 2)
  {
    if (i != last)
        Points = task[day][i] +
                 f(day-1, i]
```

```
        maxi = max(maxi, points);
    }
    return maxi;
}
```

→

t_0	t_1	t_2
2	1	3
3	4	6
10	1	6
8	3	7



3+f(0,0)	6+f(0,1)	3+3	4+3
3+3	6+2	6	7
6	8		

⇒ Do memoization

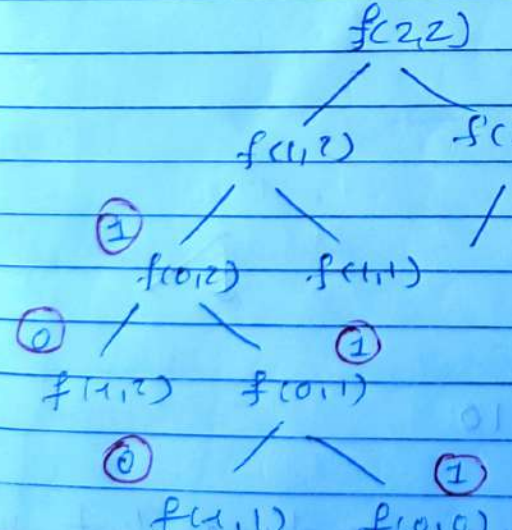
Unique Path

- 

2 ways

⇒
```
f(i,j)
{
if (i==0 & j==0)
    return 1;
if (i<0 || j<0)
    return 0;
up = f(i-1, j);
left = f(i, j-1);
return up+left;
}
```

⇒ Do memoization like dp[][]



(In case 3×3 mat)

Min Path Sum

⇒ Same as unique path but find path with smallest cost.

→ Check all path ⇒ Recursion

```
f(i,j)
{
  if (i==0 && j==0) return a[0][0];
  if (i<0 || j<0) return INT_MAX;

  up = a[i][j] + f(i-1, j);
  left = a[i][j] + f(i, j-1);

  return min(left, up);

}
```

⇒ Recursion + memoization > Tabulation
100 % 6.31 %

Triangle

```
1
2 3
4 5 6
7 8 9 10
```

→ min value for reaching last row

→ Try out all Paths

Recursion

→ In prev we know ending Point here we can reach any Point at last row

→ one sol is try recursion on every Point of last row give the Minimum one

→ but we can start from start

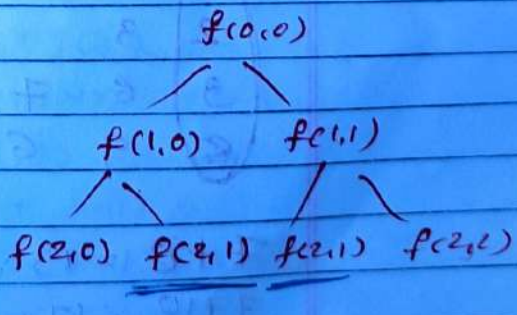
→

```
f(i,j){
  if (i==n-1) return a[n-1][j];
  down = a[i][j] + f(i+1,j);
  down-dia = a[i][j] + f(i+1,j+1);

  return min(down, down-dia);

}
```

→ $O(2)^{n+2+3+..m-1 \text{ row}}$

→ Do memoization

→ $O(n \times m) = O(n^2)$



→ Tabulation

Recursion → 0 to n-1
Tabulation → n-1 to 0
reverse

```
int dp[n][n]

for (j=0; j<n; j++)
    dp[n-1][j] = a[n-1][j];

for (i=n-2; i>=0; i--) {
    for (j=i; j>=0; j--) {

        dp = a[i][j] + dp[i+1][j];
        dg = a[i][j] + dp[i+1][j+1];

        dp[i][j] = min(dp, dg)
}
return dp[0][0];
```

→ O(n×n) = O(n²)

1				14			
2	3			13	15		
3	6	7		11	12	13	
8	9	6	10	8	9	6	10

7+6 = 13 6+9 = 15 3+8 = 11
7+10 = 17 6+6 = 12 3+9 = 12

Minimum / Maximum Falling Path Sum

n x m matrix

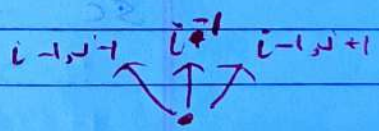
⇒ Starting From any Point of first row Reach at any Point of ending row

⇒ for a[i][j] you can go a[i+1][j], a[i+1][j-1], a[i+1][j+1]

⇒ Such that Sum of Path is min/max

⇒ we don't know where to start or end from.

Call all ending Point Return max

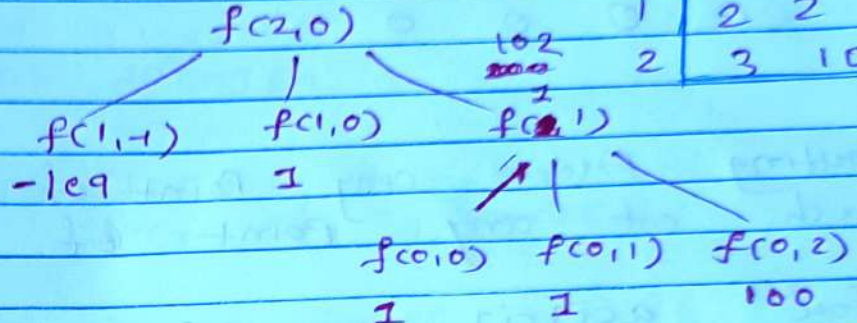



```
f(i,j)
{
if(i==0) return a[0][j];
if(j<0 || j>=m) return -1e9;
int s = a[i][j] + f(i-1,j);
int ld = a[i][j] + f(i+1, j-1);
int rd = a[i][j] + f(i+1, j+1);
return max(s, ld, rd);
}
```

TC → $O(3^n)$ ← 1 → | n length
SC → $O(n)$

→ do memoization

	0	1	2
0	1	1	100
1	2	2	3
2	3	10	2



→ 3 + -1e^9
3 + 1
3 + 102 = 105

→ overlapping Subproblem will be in different points.

TC → O(N x m)
SC → O(N x m) + O(N)

- Tabulation
- base case
- observe i, j Convert to loops
- recursion n-1 → 0
  Tabulation 0 → n-1

```
int dp[n][m];
for (j=0 → m-1)
    dp[0][j] = a[0][j];
for (i=1 → n-1) {
    for (j=0 → m-1) {
        u = a[i][j] + dp[i-1][j];
if(j-1>=0)  ld = a[i][j] + dp[i-1, j-1];
if(j+1<m)   rd = a[i][j] + dp[i-1, j+1];
        dp[i][j] = max (u, ld, rd);
    }
}
return dp[n-1][m-1];

maxi = dp[n-1][0];
for (j=1 → m-1)
    maxi = maxi (maxi, dp[n-1][j]);

return maxi;
```

Cherry Pick up II (3D DP)

⇒ Grid every point = chocolate

(0,0) = Human 1 (0,c-1) = Human 2

⇒ Take max chocolate

(2)	3	1	(2)
3	(4)	2	(2)
5	(6)	3	(5)

2+4+6 = 12
2+2+5 = 9
21
max

Date
Page

⇒ Fixed starting Point, Variable ending Point

⇒ All Path by Alice + All Path by bob

Recurse Recursion

⇒ express every thing in terms of (i_2, j_1) & (i_2, j_2)

⇒ explore all paths ↙↓↘

⇒ Max Sum return

ans = f(0,0,0,m-1)
Alice bob

```
f(i1,j1,i2,j2)  or  f(i,j1,j2)
{
if (j1<0 || j1>=m || j2<0 || j2>=m) {
    return -1e8;
}
if (i == n-1)
 {
  if (j1 == j2)
     return a[i][j1,j2];
  else
     return a[i][j1] + a[i][j2];
}
```

[3×3 = 9 Combos of Path]

// explore all paths alice & bob can go together

```
for (dj1 → -1 to +1) {
  for (dj2 → -1 to +1) {

  if (j1 == j2)
    res = max(max, a[i][j1] + f(i+1, j1+dj1, j2+dj2))

  else
    res = max(max, a[i][j1] + a[i][j2] + f(i+1, j1+dj1, j2+dj2))

  }
}

return res;
```

TC → $(3^n \times 3^n)$ ≃ exponential

SC → O(N)

⇒ do memoization

```
d[i][j1][j2]
d[N][M][M]
```

TC → O(N × M × M) ≈ $O(n^3)$

Subset Sum equal to target

⇒ Return true if there exist a subarray (subsequence) where Sum = target.

(1) express (ind, target)

(2) explore all possibility at point

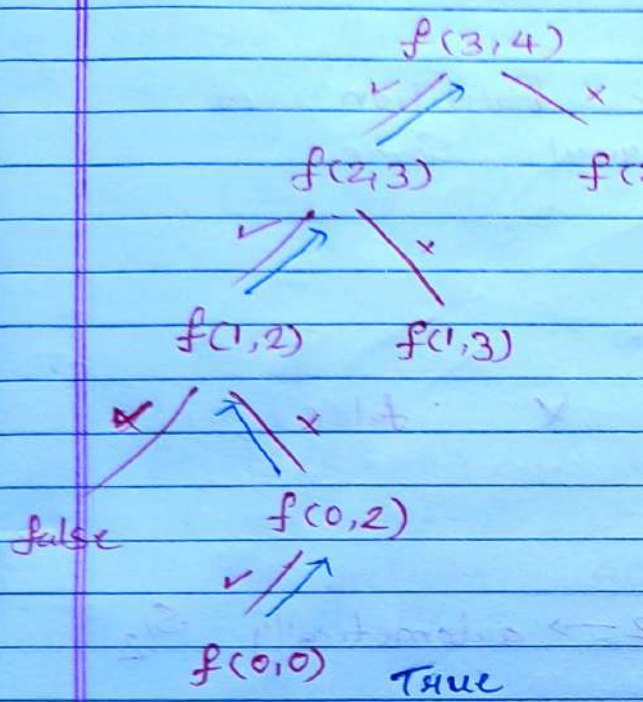
Part of Subsequence / Not Part of Subsequence

(3) Return True/ false

⇒ Ans = f(n-1, target)

```
f(ind, target)
 {
  if (target == 0) return true;
  if (ind == 0) {
      if (a[0] = target)
            return True;
      else
            return false;
  }
  bool nottake
        = f(ind-1, target);
  bool take
        = false
    if (target >= a[ind])
        take = f(ind-1, target-a[ind])

  return take || nottake;
}
```

arr = [2, 3, 1, 1] target = 4



T(c)	→	O(2^n)
S(c)	→	O(N)

→ Do memoization [ind] [target] [10^3+1] [10^3+1]

T(c)	→	O(N x target)
S(c)	→	O(N x target) + O(N)

Do Tabulation for Reducing Auxiliary Array Space

Partition equal Subset

⇒ Two or more Partition of Array have equal Some.

$S_1 = S_2 = S/2$

S → odd ✗ false

S → even

Subset → $\frac{S}{2}$

S1 → $S/2$ S2 → automatically $S/2$

⇒ Now find Subset target = $S/2$

Count Subsets with Sum K

⇒ How many different ways to choose elements out of array such that Sum = K.

⇒ [1, 2, 2, 3] tar = 3

Count no. of Subsets

[1, 2], [1, 2], [3] = 3

⇒ Recursion

⇒ Express In term of Index (Ind, target)

⇒ explore all Posibilities

⇒ Sum of

1	2	2	3

S = 3

f(n-1, S)

f(3, 3)

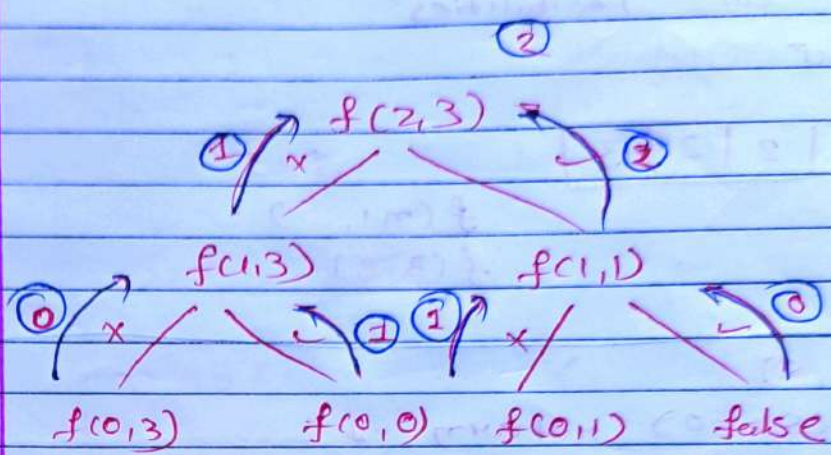
```
f(Ind, S)
{  if(S==0) return 1;
 if(ind==0)
    return   a[ind] == S;

notPick = f(n-1, Sum)
 Pick = 0;
   if(a[ind] <= S) {
     Pick = f(n-1, S-a[ind]);
  }
  return  Pick + notPick;
}
```

T(C)	→	$O(2^n)$
S(C)	→	O(N)

Do memoization

arr → {1, 3, 2}

S = 3



Ind	Sum	DP
[N]	[Sum+1]	

TC	O(N x Sum)
SC	O(N x Sum) + O(N) ASP

→ Convert to tabulation

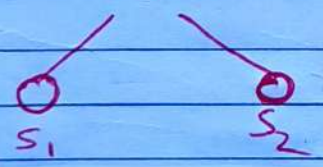
1. Base case
2. look at changing Parameters
   Create nested loops
3. Copy recursion

Recursion loop reverse
= Tabulation

```
Int dp[n][sum+1];

for(i=0 to n-1)    dp[i][0] = 1;
if(a[0]<=s)        dp[0][a[0]] = 1;        } Base

for(i=1 → n-1) {
  for(s=0 → sum) {
  nottake = dp[ind-1][sum];
    take = 0
    if(num[ind] <= sum)
    take = dp[ind-1][sum-num[ind]];

    dp[ind][sum] = nottake + take;
  }
}
return dp[n-1][tar];
```

Partition with given difference

⇒ array $S_1 > S_2$



$S_1 - S_2 = D$

⇒ (5,2,6,4) D = 3

(6,4) (5,2)
10 - 7 = 3

$S_1 + S_2 = \text{Totalsum}$

$S_1 - S_2 = D$

$\text{Totsum} - S_2 - S_2 = D$

$\text{Totsum} - D = 2S_2$

$$S_2 = \frac{\text{Totsum} - D}{2}$$

⇒ Count of subset target

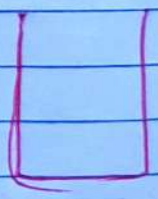
$$= \frac{\text{Totsum} - D}{2}$$

⇒ $\text{Totsum} - D >= 0$

01 Knapsack

Item				
weight	→	3	4	5
Value	→	30	50	60

Bag wt = 8



⇒ Try out all combination
recursion

⇒ express (Ind, Val)

⇒ Pick, noPick

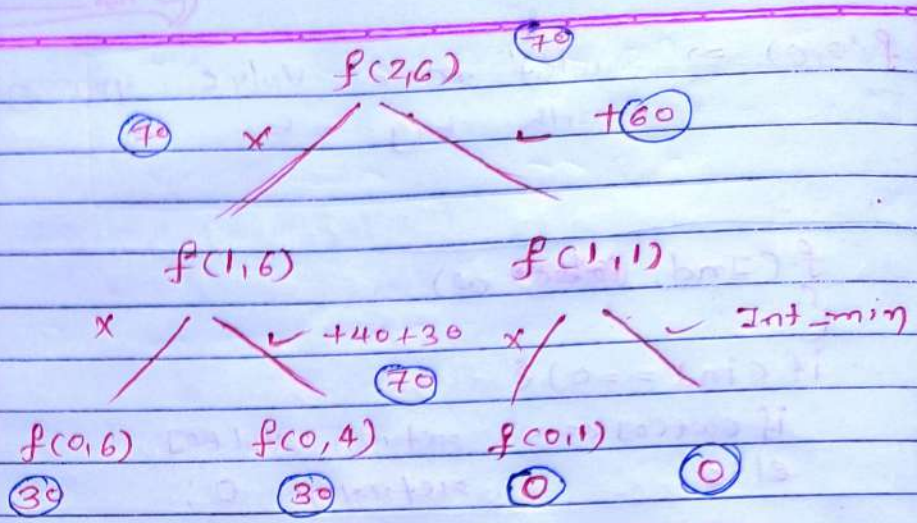
max

f(2,c) ⇒ what max Value you get? with bag = c

```
f(Ind, w)
{
if(ind==0){
  if(wt[0]<=w) return Val[0];
  else         return 0;
  nottake = 0 + f(Ind-1, w)
  take = Int_min;
  if(wt[Ind] <= w)
      take = Val[Ind] + f(ind-1, w-wt[Ind])

  return max(take, nottake);
}
```

T(c)	⟶ O(2^n)
S(c)	⟶ O(N)

	0	1	2
→ wt	→ 3	2	5
Val	→ 30	40	60



⇒ Do memoization

DP [Ind] [w+1]

⇒

T(C) ⟶ O(N×w)

S(C) ⟶ O(N×w) + O(N)

Ass

⇒ Tabulation

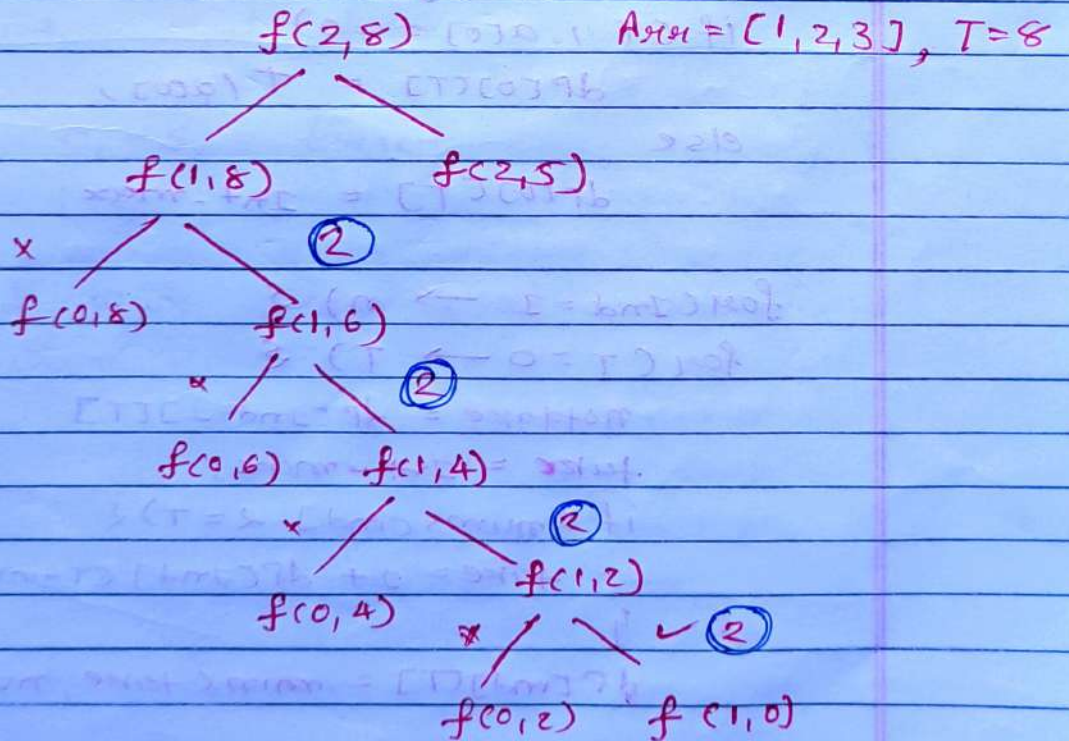
Ind dp [n] [w]

```
for (int i = wt[0] → w)
    dp[0][i] = Val[0];
for (Ind = 1 → n) {
    for (w = 0 → w) {
    nottake = dp[Ind-1, w]
    Take = Int_min;
    if (wt[Ind] <= w){
        take = Value[Ind] + dp[Ind-1] [w - wt[Ind]
    dp[Ind][w] = max (take, nottake)
}
return dp[n-1][w];
```

Minimum Coins

arr[] = {1,2,3} target = 7
Coins [3,3,1]

```
f(Ind, T) {
if (Ind == 0) {
    if (target % arr[ind] == 0) return T / arr[Ind];
    else return 1e7

notTake = 0 + f(Ind-1, T);
take = Int_max;
if(T <= Arr[Ind]) {
    take = 1 + f(Ind, T - Coins[Ind]);

return min(take, notTake);
```



T(c)	→	>> 2^n [exponential]
S(c)	→	>> O(n) O(T)

→ Do memoization

dp[ind][T]

T(c)	→	O(N x T)
S(c)	→	O(N x T) + O(N) ASS

Tabulation

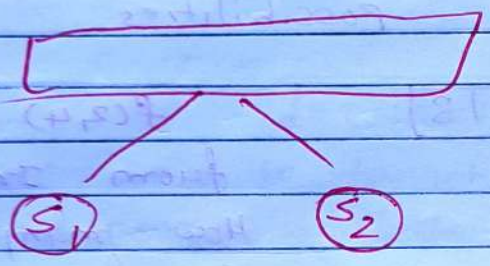
```
dp[N][target]
for (T=0 → target)
  if (T % a[0] == 0)
      dp[0][T] = T / a[0];
  else
      dp[0][T] = Int.max;

for (ind = 1 → n) {
  for (T = 0 → T) {
      notake = dp[ind-1][T]
      take = Int.max;
      if (nums[ind] <= T) {
          take = 1 + dp[ind][T-nums[ind]];
      }
      dp[ind][T] = min(take, notake);

return dp[n-1][target];
}
```

Target Sum

⇒ arr[] → {1, 2, 3, 1} | assign sign
target = 3

- −1 +2 +3 −1 = 3
  {−, +, +, −}
  +1 −2 +3 +1 = 3
  {+, −, +, −}

⇒ How many ways



$S_1 - S_2 = D$

Devide Seek that

$S_1 - S_2 = target$

Find way expansion;
Same

Coins change 2

→ | 1 | 2 | 3 | ← target = 4

→ total coins to make = 4
{1,1,1,1}
{2,1,1}
{2,2}
{3,1}

1. Express in term of Ind
2. Explore all possibilyes
3. Sum all possibilities

| 1 | 2 | 3 |

f(2,4)
from Index 2
How many times you
Can form 4.

```
f(Ind, T)
{
 if (Ind == 0)  (T-1. arr[Ind] == 0)
    if arr[Ind] == T  return 1;
    return 0;

  nottake = f(Ind-1, T)
  take = 0
  if (arr[Ind] <= T) {
     take = f(Ind, T - arr[Ind])

  return nottake + take;

}
```

T(C)	⟶	exponential
S(C)	⟶	>> O(n)

⇒ Do memoization

• DP [Ind] [target + 1]

T(C)	⟶	O(N x T)
S(C)	⟶	O(N x T) + O(n) Ass

Tabulation

```
                dp[N][T]
for (T -> 0 to target)
    dp[0][T] = T % arr[0] == 0

for (Ind = 1 to N) {
    for (T = 0 to target) {

        notTake = dp[Ind-1][T];
        take = 0;
        if (a[Ind] <= T)
            take = dp[ind][T - a[Ind]);

        dp[Ind][T] = take + notTake


return    dp[n-1][Value];
```

do space optimizing using prev[], curr[]

Unbounded Knapsack [Infinite Supply]

wt = {2, 4, 6}
Val = {5, 11, 13}

w = 10

wt → 6 = 13
wt → 4 = 11
24

wt → 6 = 13
wt → 2 = 5
wt → 2 = 5
23

wt → 4 (1)
wt → 4 11
wt → 2 5
27

```
f(Ind, w)
{
  if(Ind == 0) {
    if(w/wt[Ind] == 0) return w/wt[Ind] * Value[Ind]
    return 0;
  }

  nottake = 0 + f(Ind-1, w);
  take = 0;
  if(wt[Ind] <= w) {
      take = Val[Ind] + f(Ind, w - wt[Index]);
  }

  return max(nottake, take);

}
```

T(C)	→	exponential
S(C)	→	>> O(n)

Do memoization
dp[N][W]

T(C)	→	O(N x W)
S(C)	→	O(N x W) + O(N)
		Ass

Tabulation

```
                dp[N][W+1]
for (w -> 0 to bag w)
    dp[0][w] = (w / wt[0]) x Val[0]

for (Ind = 1 to n) {
    for (w = 0 to weigh) {
    nottake = 0 + dp[Ind-1][w];
    take = 0
    if (wt[Ind] <= w) {
        take = Val[Ind] + dp[Ind]
                          [w - wt[Ind]]

    return  dp[Ind][w]
            = max(take, nottake)
    }
}
return  dp[n-1][w]
```

Rod Cutting Problem

→ N = 5 (Rod length)

Price [] →

2	5	7	8	10
1	2	3	4	5

(2) (2) (1)

5 + 5 + 2 = 12

→ make Price maximum.

Try to pick length & Sum them up to Value N.

1. Express → (Ind, N)
2. Explore all posibilities → nottake, take
3. max return

2	5	7	8	10
0	1	2	3	4

f(4, N)
max Price
It can gain

```
f(Ind, N)
{
  if(Ind == 0)
     return N * Price[0];

  nottake = 0 + f(Ind-1, N);
  take = Int_min;
  Rod_length = ind + 1;
  if (Rod_length <= N)
     take = Price[Ind] + f(Ind, N-Rod_len);

  return max(nottake, take);
}
```

T(n) ⟶ O(exponential)
S(n) ⟶ > O(n)

→ Do memoization

T(n) ⟶ O(N×N)
S(n) ⟶ O(N×N) × O(N)
Ass

DP on Strings

longest Common Subsequence

⇒ Maintance order on strings

n word ⇒ 2^n sequence

S_1 = "adebc" S_2 = "dcadb"

[32 sub sequence]

adb adb

⇒ Length = 3

⊙ Brute force

32 - 32 O(exponential)

linear comparison

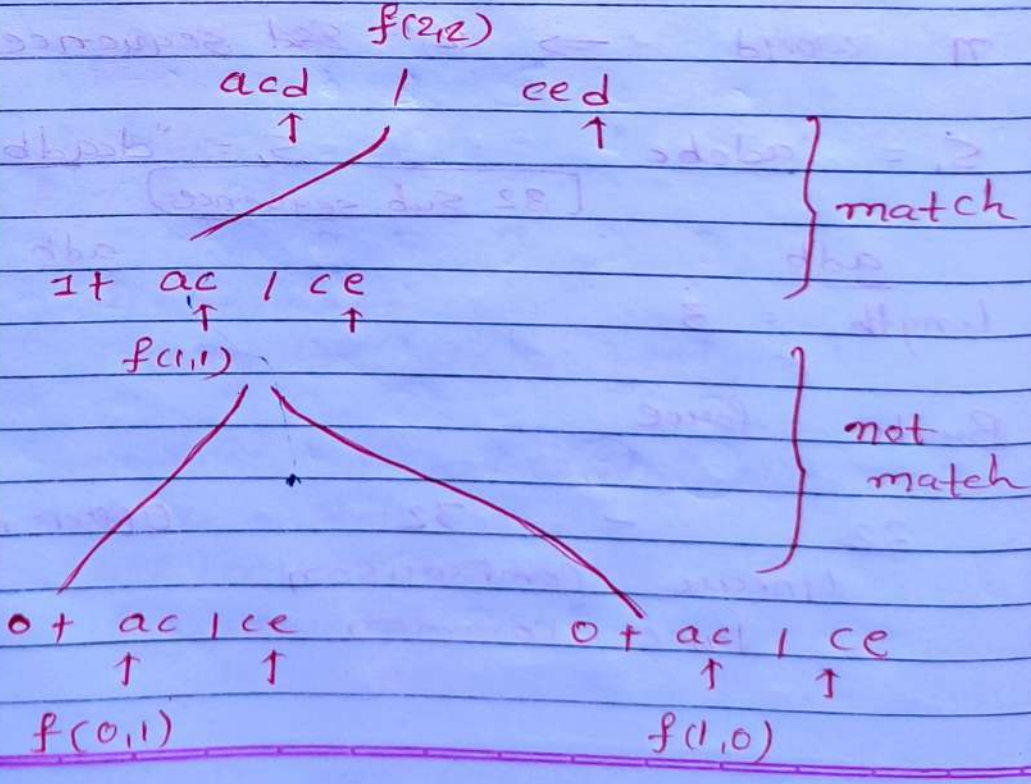
take common

Generate all subsequence → Compare on way

acd		ced
a		c
c		e
d		d
ac		ce
ad		ed
cd	=	cd (2)
acd		ced

Rules
⇒ express Index f(Ind 1, Ind 2)
⇒ explore Possibility
⇒ Take best among them

f(2) ⇒ string till index 2
f(2,2) ⇒ LCS of string (0...2)



match

$1 + f(Ind1-1, Ind2-1)$

Not match

$0 + f(Ind1-1, Ind2)$
$0 + f(Ind1, Ind2-1)$
} max out of them

```
f(Ind1, Ind2)
{
if (Ind1 < 0 || Ind2 < 0) {
    return 0;
}

if (s1[Ind1] == s2[Ind2])
    return 1 + f(Ind1-1, Ind2-1);

return 0 + max(f(Ind1-1, Ind2),
               f(Ind1, Ind2-1));
}
```

T(c) ⟶ $O(2^n \times 2^m)$ ≈ expone.
S(c) ⟶ $O(n)$

⇒ Do memoization
DP [n][m]

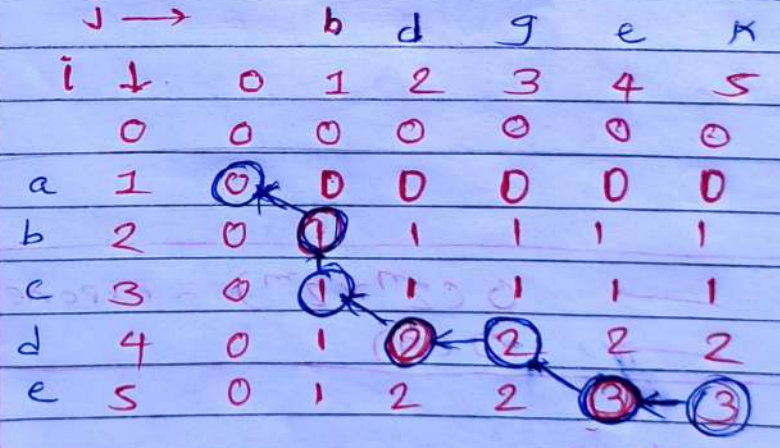
T(c) ⟶ $O(m \times n)$
S(c) ⟶ $O(m \times n) + O(n)$ +m

Tabulation..

→ for Tabulation shifting of Index

```
for(j=0 → m+1)      dp[0][j] = 0
for(i=0 → n+1)      dp[i][0] = 0

for (i = 1 → n) {
    for (j=1 → m) {

    if (s[i-1] = t[j-1])
  else  dp[i][j] = 1 + f(i-1, j-1);
      dp[i][j] = max ( f(i-1, j) , f(j-1, i) );
}
}
Return dp[n][m];
```

Print LCS



bde

```
Ind = len-1
i=n , j=m
while ( i>0  ff  j>0)
{
 if(s1[i-1] == s2[j-1])
 {
  s[Ind] = s1[i-1];
  Ind--;
  i--; j--;
 }

-else if (dp[i-1][j] > dp[i][j-1])
 {
    i--;
 }

 else ##
 {
    j--;
 }
```

Longest Palindrome Subsequence

S = "bbbab" → ab
→ bb ✓
→ bbbb ✓
→ bab ✓

⇒ longest of all Palindrome

Brute force ✗

Recursion (LCS) ✓

S_1 = "bbabcbcab" S_2 = "bacbcbabb"

A = babcbab

⇒ LCS In S and $1/S$ is Longest Palindrome

⇒ LCS (S, reverse(S)) ;

Minimum Insertion Required to make Palindrome

S = "abeaa"

⇒ Joint reverse
abcaa aacba

operation = length of String

"a@bc@baa" 2 Insertion

→ Keep Longest Palindrome Intake

abc aa Here aaa

abc a cba Insert reverse bc between a a

codingninjas

codi s a jn ingni njas idoc

⇒ n - longest Palindrome Subsequence

Ans

min number of operation to make str1 to str2

⇒ str1 str2
n m

n delete + m Insert
n + m

⇒ There is no need to change longest Common Subsequence (LCS)

abcd ⟶ anc

LCS ⟶ ac

⇒ abcd - ac = 2 delete
anc - ac = 1 Insertion

deletion = n - Len(LCS)
Insertion = m - Len(LCS)

Shortest Common Supersequence

S_1 = "brute" S_2 = "groot"

"brutegroot" = 10

"bgruoote" = 8

S_1 = "bleed" S_2 = "blue"

"bleued" = 6

Common guys taken once (LSC)

$m+n - len(LCS)$

What about what is String?

		j		g	r	o	o	t
i		0	1	2	3	4	5	
	0	0	0	0	0	0	0	
b	1	0	0	0	0	0	0	
r	2	0	0	1	1	1	1	
u	3	0	0	1	1	1	1	
t	4	0	0	1	1	1	2	
e	5	0	0	1	1	1	2	

Common add once

etouorbg

←

gbruoute

```
i=n, j=m                                ans = ""
while ( i>0 ff j>0)
{
    if (s1[i-1] == s2[j-1])
    {
        ans+ = s1[i-1];
        i-- ; j--;
    }
    else if (dp[i-1][j] > dp[i][j-1] ) // up
    {
        ans+ = s1[i-1];
        i--;
    }
    else                                // down
    {
        ans+ = s2[j-1];
        j--;
    }
}
while (i>0)
        ans += s1[i-1];
        i--;
    while (j>0)
        ans += s2[j-1];
        j--;

return ans.reverse();
```

Distint Subsequence

S = "babgbag" S_2 = "bag"

babgbag
babgbag
babgbag
babgbag
babgbag

} 5 occurance

⇒ Different methods of comparing
Trying all ways

Recursion

Count no. of ways

1. Represent In terms of Index
2. Explore all possibilities
3. Return Sum of all Posibbilities
4. Base Case (either 0 or 1)

⇒ $f(n-1, m-1)$

→ Number of distinct subsequences between of S_2 from [0...j] In string S_1 [0...i]

babgbag bag

i j

```
f(i,j)
{
  if (j<0)  return 1;
  if (i<0)  return 0;

  if (s1[i] == s2[j]){

     return  f(i-1,j-1) + f(i-1,j);
  }
  else {
     return  f(i-1,j);
  }
}
```

T(c) ⟶ O(exponential) $(2^n \times 2^m)$

S(c) ⟶ O(n)

Ass

Do memoization dp[n][m]

T(c) ⟶ O(N×m)

S(c) ⟶ O(n×m) + O(n)

Ass

Tabulation

```
dp[m][n]
dp[n][m]          Increase all
dp[n+1][m+1]      Index by one
```

```
for (i=0 → n)
    dp[i][0] = 1;
for (j=1 → m)
    dp[0][j] = 0;
```

Base

loop

```
for (i=1 → n) {
   for (j=1 → m) {

    if (s[i-1] == t[j-1]) {
      dp[i][j] = d[i-1][j-1] +
                 dp[i-1][j];
    } else {
      dp[i][j] = dp[i-1][j]; }
   }
}
return dp[n][m];
```

copy recursion

Edit Distance

S_1 = "horse"
S_2 = "ros"

1. Insert
2. Remove
3. Replace

→ Min operation to Convert S_1 to S_2 (we have one more operation Replace)

→ Replace h with r "horse"
Remove r "hose"
Remove e "ros"
(3) steps

→ S_1 = "Intention" S_2 = "execution"

Remove t "Inention"
Replace i→e "enention"
Replace n→x "exention"
Replace n→c "exection"
Insert u "execution"
(5) Operation

⇒ Insert of same character
⇒ Delete, Find Somewhere else
⇒ Replace and match

```
f(i,j)
{
if (i<0)
    return j+1;
if (j<0)
    return i+1;

if (s1[i] == s2[j]
    return f(i-1, j-1);

Insert = 1 + f(i, j-1);
Delete = 1 + f(i-1, j);
Replace = 1 + f(i-1, j-1);

return min(Insert, delete, Replace);

}
```

T(C) ⟶ $O(3^m \times 3^n)$ exponential

S(C) ⟶ O(n+m)

Ass

Do memoization

dp[n][m]

T(C) ⟶ O(N x m)

S(C) ⟶ O(N x m) + O(n+m)

Ass

Wildcard matching

? → matches with single character

* → matches with sequence of length 0 or more

S_1 = "?ay"
S_2 = "ray"

True

S_1 = "ab*cd"
S_2 = "abdefcd"

True

S_1 = "**abcd"
S_2 = "abcd"

True

S_1 = "ab?d"
S_2 = "abcc"

False

1. express (i, j)
2. Explore all comparison
3. out of all comparison, if anyone can return true.

f(n-1, m-1)

$S_1[0, 1, 2 \ldots n]$
$S_2[0, 1, 2 \ldots m]$

True / false

```
f(i, j)
{
 if (i<0 ff j<0) return True;
 if (i<0 ff j>=0) return false;
 if (j<0 ff i>=0) {
    for (x=0 → i) if (s1[x] != "*") return false
    return True
 }
 if (s1[i] == s2[j] || s1[i] == '?')
        return f(i-1, j-1);
 if (s1[i] == '*')
        return f(i-1, j) || f(i, j-1)
 return false;
}
```

T(c) ⟶ $O(3^n \times 3^m)$ exponential

S(c) ⟶ O(N+M)

Ass

Do memoization

dp[N][M]

T(c) ⟶ O(N x M)

S(c) ⟶ O(N x M) + O(N+M)

Ass

Dynamic Programming 2.0

CELEBRO™ DATE PAGE

Best time to Buy and sell stocks

arr[] = {7, 1, 5, 3, 6, 4} → Price at i^{th} day

Buy sell

6 - 1 = 5 Max

⇒ if selling at i^{th} day
Buy on minimum of [0 ... i-1]th day

```
mini = a[0]  , Profit = 0 ;
for (i=1 ; i<n ;i++) {
{
    Cost = a[i] - mini ;
    Profit = max (Profit , Cost) ;
    mini = min (mini , a[i]) ;
}
return Profit ;
```

⇒ Taking mini while Continuing into loop :- Dynamic Programming

⇒

T(c)	⟶	O(n)
S(c)	⟶	O(n)
		Ass

Best time to Buy and sell stock II

→ Can Buy as many time as want
Can sell as many time as want

have to sell for buying second time

Price[] = [7, 1, 5, 3, 6, 4]

4 + 3 = 7

```
f(Ind, Buy)
{
if(Ind == n) return 0;
    return 0;

if(Buy)
   Profit = -Prices[Ind] + f(ind+1, 0)  // Buy
            0 + f(ind+1, 1)             // Not Buy
          max

else
   Profit = Prices[Ind] + f(ind+1, 1)   // sell
            0 + f(Ind+1, 0)             // not sell
          max

return Profit;

}
```

T(C)	⟶	O(2^n)
S(C)	⟶	O(n)

Do memoization

DP[N][2] ;

T(C)	⟶	O(N×2)
S(C)	⟶	O(N×2) + O(n)
		Ass

Best Time to Buy & Sell Stocks III

Prices = [3,3,5,0,0,3,1,4] 2+3=5, 3+3=6

⇒ At max 2 Transactions

```
f(Ind, Buy, CaP)
{
 if (Ind==n) return 0;
 if (CaP==0) return 0;

 if (Buy) {
 max (-Price[Ind] + f(ind+1, 0, CaP),
          0       + f(Ind+1, 1, CaP));
 }
 else {
   max ( Price(Ind) + f(Ind+1, 1, CaP-1),
          0         + f(Ind+1, 0, CaP));
 }
```

T(c)	⟶	O (exponential)
S(c)	⟶	O (N)
		Ass

Do memoization

Ind	Buy	CaP
0...n-1	0/1	0/1/2
N	2	3

dP[N][2][3]

T(c)	⟶	O (N x 2 x 3)
S(c)	⟶	O(N x 2 x 3) + O(N)
		Ass

Best_time_to_Buy_&_sell_stock_IV

Prices[] = [3,2,6,5,0,3] k = 2

⇒ Atmost K transaction
Same code for K Transaction

Longest Increasing Subsequence

arr[] = [10, 9, 2, 5, 3, 7, 101, 18]

{2, 3, 7, 101}
{2, 3, 7, 18}
{2, 5, 7, 101}.

⇒ Brute force
↳ check all store longest
↳ $O(2^n)$

⇒ Trying all way
↳ recursion } f(0, -1) ans

⇒

```
f(Ind, Prev_Ind)
{
  if (Ind == n)
      return 0;

  len = 0 + f(Ind+1, Prev_Ind)   // not take

  if (Prev_Ind == -1 || arr[Ind] > arr[Prev_Ind]){

  len = max(len, 1 + f(Ind+1, Ind))

  }
  return len;
}
```

T(c)	⟶	O(2^n)
S(c)	⟶	O(N)

Do memoization

DP [N][N]

T(c)	⟶	O(N x N)
S(c)	⟶	O(N x N) + O(N) Ass

Coordinate change (shift)

-1	0	1	2	3	....	n-1
0	1	2	3	4	....	n

Tabulation

```
dp[n][n+1]
for (ind = n-1 ⟶ 0) {
    for (prev_ind = ind-1 ⟶ -1) {
    len = 0 + dp[ind+1][prev_ind+1];
    if (prev_ind == -1 || a[ind] >
                          a[prev_ind])
        len = max(len, 1+dp[ind+1]
                          [ind+1]);
    }
    dp[ind][prev_ind+1] = len;
  }
}
return dp[0][-1+1];
```

1	1	2	1	3	2
5	4	11	1	16	8

→ dp[n]

dp[i] ⟶ Signify the longest Increasing Subsequence that ends at i.

```
dp[n]
for ( ind = 0 → n-1) {
  for (prev = 0 → ind-1) {
      if (arr[prev] < arr[ind])
      dp[ind] = max (1+dp[prev], dp[ind]);
  }
}
max of dp
```

Binary Search

⇒ [1, 7, 8, 4, 5, 6, -1, 9]

[1, 5, 6, 9]

Ans

[1, 7, 8, 9] Len = 4

[1, 4, 5, 6, 9] = 5

[-1, 9] = 2

Longest Divisible Subset

arr[] = {1, 16, 7, 8, 4}

arr[i] % arr[j] == 0 or
arr[j] % arr[i] == 0

{16, 8, 4} ⇒ (16, 8) (8, 4)
(16, 4)

{1, 16, 8, 4} ⇒ Ans

Longest Divisible Subsequence

```
for(i = 1 ⟶ n) {
    for (j = 0 ⟶ i) {
    if (arr[i] % arr[j] == 0 && dp[i] < dp[j]+1)
        dp[i] = dp[j] + 1;
    }
}
```

⇒ Do the hush thing By yourself.

Longest String chain

words [] = {"a", "b", "ba", "bca", "bda", "bdca"}

[a, ba, bda]
[a, ba, bca]
[b, ba, bca] ⇒ Insertion In every step

⇒ [a, ba, bca, bdca] ⇒ Answer
len = 4

⇒ longest string subsequence

arr[i]
bdca
④

bd@a
↑↑↑↑

arr[j]
bda
③

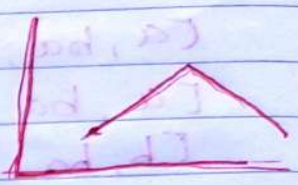
bda
↑↑↑

// Sort According to length first
⇒ Compare (string s, string t) {
if (s1.len() != s2.len()+1) false;
int first = 0, second = 0;
while (first < s1.len) {
if (s1[first] == s2[second]) {
first++;
second++;
} else {
first++;
}
}
if (first == s1.len && second == s2.len) {
return true;
}
}

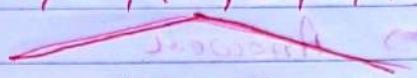
Longest Bitonic Subsequence

arr[] = {1, 11, 2, 10, 4, 5, 2, 1}

→ Increase, Decrease



[1, 2, 10, 4, 2, 1]
[1, 2, 10, 5, 2, 1]



len = 6

→ Can be only Increase or (LIS)
only Decrease (LDS)

[1, 2, 4, 5] ✓

```
for (i=0 → n) {
  dp[i] = 1
  for (j=0 → i) {
    if (arr[j] < arr[i] &&
        1+dp1[j] > dp1[i]) {
      dp1[i] = 1 + dp1[j]
    }
  }
}
for (i=n-1 → 0) {
  for (j=n-1 → i) {
    if (arr[j] < arr[i] &&
        1+dp2[j] > dp2[i]) {
      dp2[i] = 1 + dp1[j]
    }
  }
}
```

arr[]	{1, 11, 2, 10, 4, 5, 2, 1}
dp1[]	{1, 2, 2, 3, 3, 4, 2, 1}
dp2[]	{1, 5, 2, 4, 3, 3, 2, 1}

2 7 4 (7) 6 7 4 2

1 elem is common

⇒ 7 - 1 = (6) ans

⇒

```
maxi = 0
for (i = 0 → n) {
    maxi = max(maxi, dp1[i] + dp2[i] - 1);
}
return maxi;
```

Number of Longest Increasing Subsequence

arr[] → {1, 3, 5, 4, 7}

[1, 3, 4, 7]
[1, 3, 5, 7] } 2

arr[] → [1, 5, 4, 3, 2, 6, 7, 10, 8, 9]
dp[] → [1, 2, 2, 2, 2, 2, 3, →
count[] → [1, 1, 1, 1, 1, 5, 9, →

Matrix chain multiplication

⇒ Solve a problem in a Pattern

$A = 10 \times 30$

$B = 30 \times 5$

$C = 5 \times 60$

⇒ $\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}_{2\times2} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix}_{2\times1}$

$\begin{bmatrix} 1\times2+2\times3 \\ 3\times2+1\times3 \end{bmatrix}_{2\times1}$

⇒ Total multiplication $= 2 \times 2 \times 1$

$= 4$

⇒ $(ABC) = (AB)(C)$

$= (10\times30\times5) \times (5\times60)$

$= 10\times30\times5 + 10\times5\times60$

$= 1500 + 3000$

$= 45000$

$(ABC) = A(BC)$

$= (10\times30) + (30\times5\times60)$

$= 30\times5\times60 + 10\times30\times60$

$= 9000 + 18000$

$= 27000$

⇒ Similliam for four ABCD

arr[] → {10, 20, 30, 40, 50}

N = 5

Dimension of N-1 matrix

A → 10 X 20		1st →	A[0] x A[1]
B → 20 X 30		2nd →	A[1] x A[2]
C → 30 X 40		3rd →	A[2] x A[3]
D → 40 X 50		ith →	A[i-1] x A[i]

1. Start with entire block (i, j)
2. Try all Partition (loop)
3. Return best Possible Two Partition

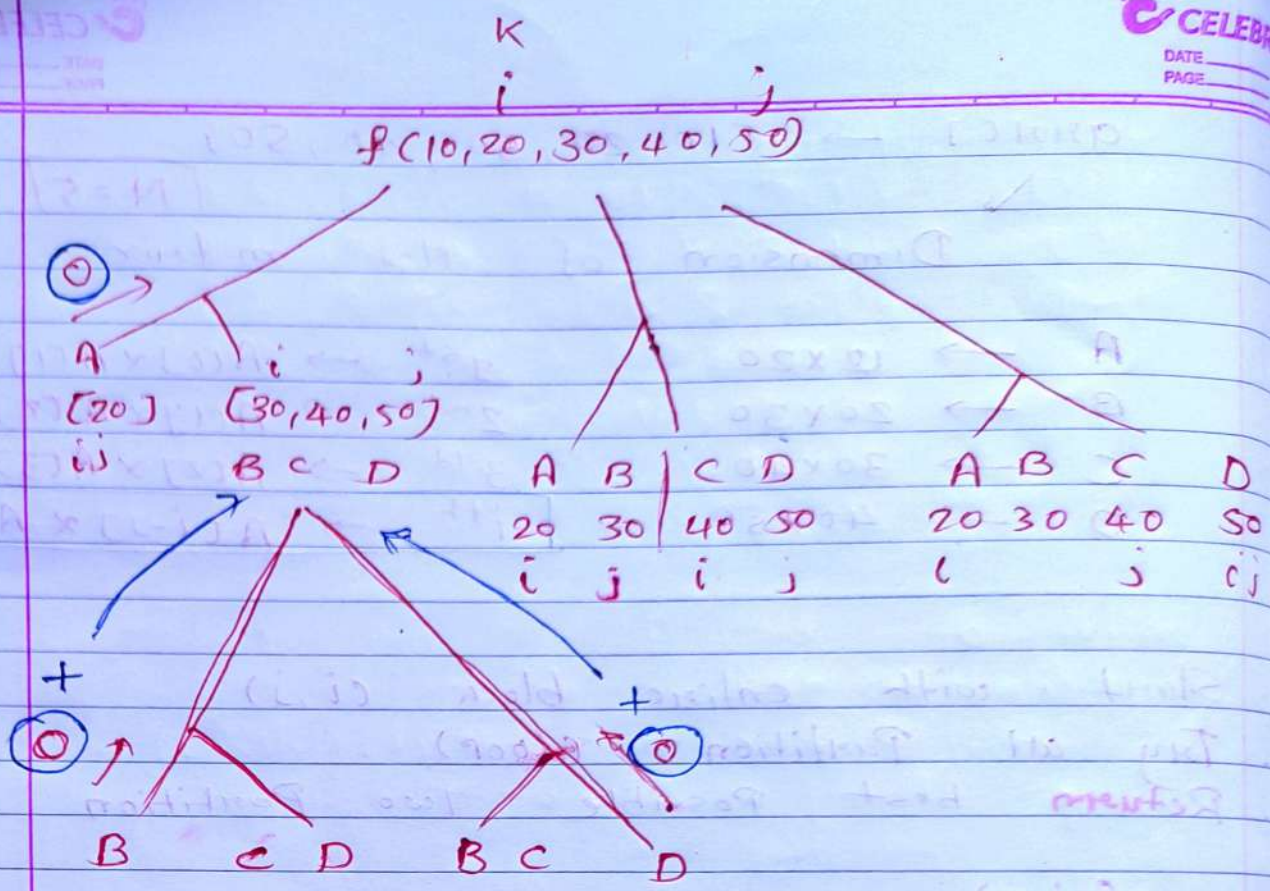
```
f(i,j)
{
if (i == j) return 0;
for (k = i → j-1)
 {
  steps = (A[i] x arr[k] x arr[j]) +
  arr                f(i,k) + f(k+1, j)
  mini = min(mini, steps)
 }

return mini;
}
```

→ [10, 20, 30, 40, 50]

A B C D

i (1) j (n-1)

f(1,4) → return the multiplications


K
i
j
f(10,20,30,40,50)
A
[20]
ij
i
j
[30,40,50]
B C D
A B C D
20 30 40 50
i j i j
A
A B C D
20 30 40 50
i j
+
+
B C D
B C D