Today's Agenda :-

Recursion
- → Intro
- → Sum of N numbers
- → Power
- → fibonacci

Dynamic Programming
- → Intro
- → Types
- → Fibonacci
- → 0-1 knapsack
- → Unbounded knapsack
- → Coin Change
- → Edit Distance

Recursion :- function calling itself

↳ solving problems using smaller instance of same problem.

$sum(N) = 1 + 2 + 3 + \cdots N.$

$sum(4) = sum(3) + 4.$

$$\underbrace{sum(N)}_{\text{↳ Bigger Problem}} = \underbrace{sum(N-1)}_{\text{↳ Subproblem}} + N.$$

How do we write a recursive code?

1) Assumption :- you will assume your function works for subproblem.
↳ faith

2) Main logic :- solve bigger problem with subproblem.

3) Base Condn :- Just write the answer for smallest input you know.

```
int sum(N) {
    if (N==1) { return 1 }
    return sum(N-1) + N
}
```

fact(3) = 1*2*3 = 6. fact(N) = fact(N-1) * N

fact(4) = 1*2*3*4 ⇒ fact(4) = fact(3)*4.

```
int fact(N) {
    if ( N==0) { return 1 }
    return fact(N-1) * N
}
```

Tracing, N=4

10

```
int sum(N=4) {
    if (N==1) {
        return 1;
    }
           6
    return sum(N-1) + n
}
```

```
int sum(N=3) {
    if (N==1) {
        return 1;
    }
           3
    return sum(N-1) + n
}
```

T.C → Recurrence Relation.

$T(n) = T(n-1) + 1$

T.C → O(N).

S.C → O(N).

```
int  sum(N=2) {
    if (N==1) {
        return 1;
    }
    return sum(N-1) + n
}
```

```
int  sum(N=1) {
    if (N==1) {
        return 1;
    }
    return sum(N-1) + n
}
```

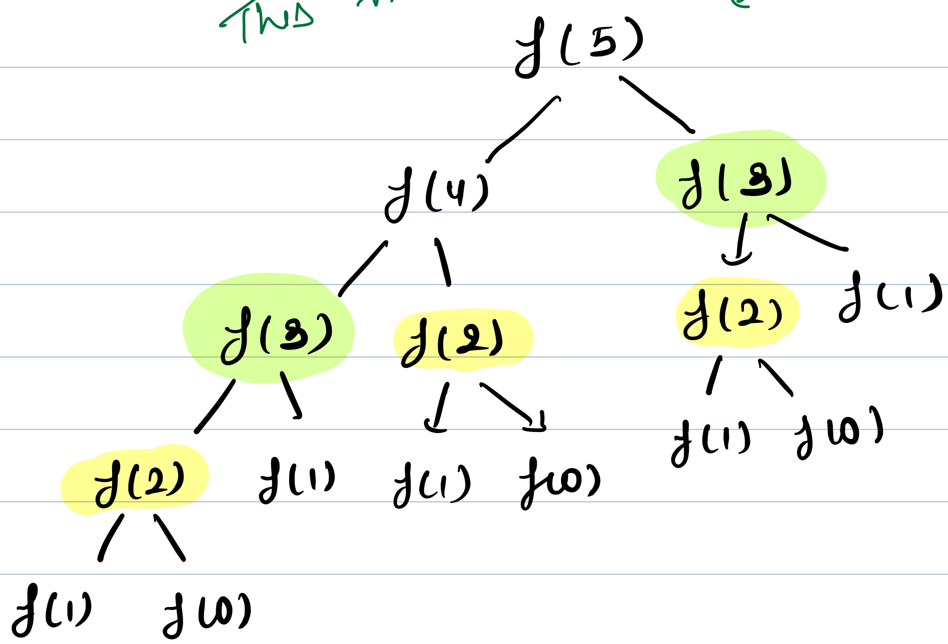
fib() :	0	1	1	2	3	5	8	13	21	34	55
Input →	0	1	2	3	4	5	6	7	8	9	10

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

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

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

This simple code is very poisonous.



← Dynamic Programming →

1. Overlapping Subproblems
2. Optimum Substructure.

↓

The answer to a larger problem can be optimally computed as a combination of answer of smaller problems of same type.

Types of DP

1. Memoization
   ↓
   recursion
2. Tabulation.
   ↓
   Iteration

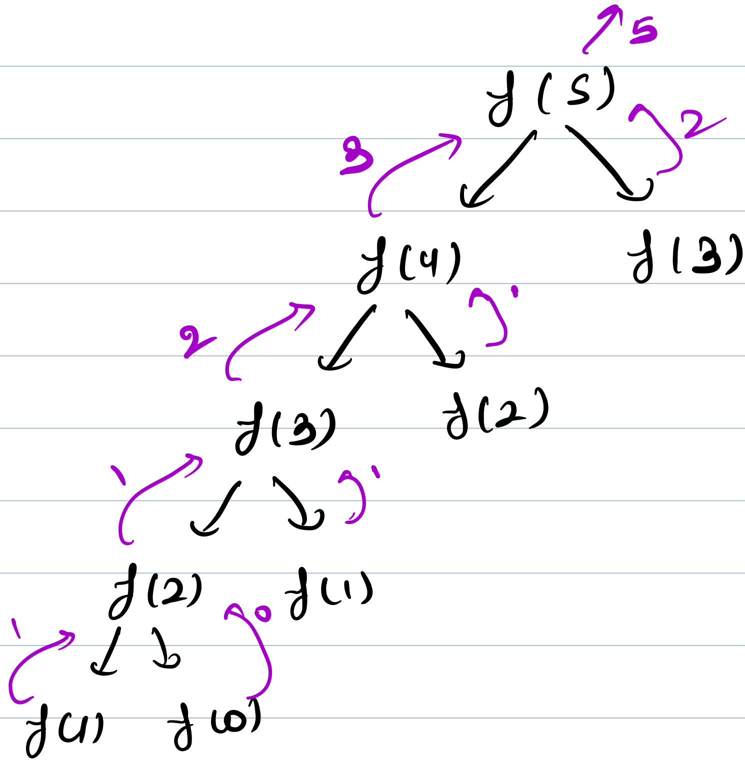
int dp[] =

0	1	2	3	

```
int fib(N, dp ) {
    if (n==0||n==1) { return n }
        if (dp[N]!=0) { return dp[N] }

    return dp[N] = fib(N-1,dp ) + fib(N-2,dp)
}
```

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


5
f(5)
3
2
f(4)
f(3)
2
1
f(3)
f(2)
1
1
f(2)
f(1)
1
0
f(1)
f(0)

KNAPSACK

0-1 knapsack

Given N items each with a weight and value, find max value which can be obtained by picking items such that total weights of all items <= cap

Note 1: Every item can be picked at max 1 time.
Note 2: we can't take part of item.

Ex:- N = 4 items, K = 50.

N =	1	2	3	4
wts =	20	10	30	40
Val =	100	60	120	150
v/w =	5	6	4	3.75

ans = pick 1 and 3 element,
ans = 220.

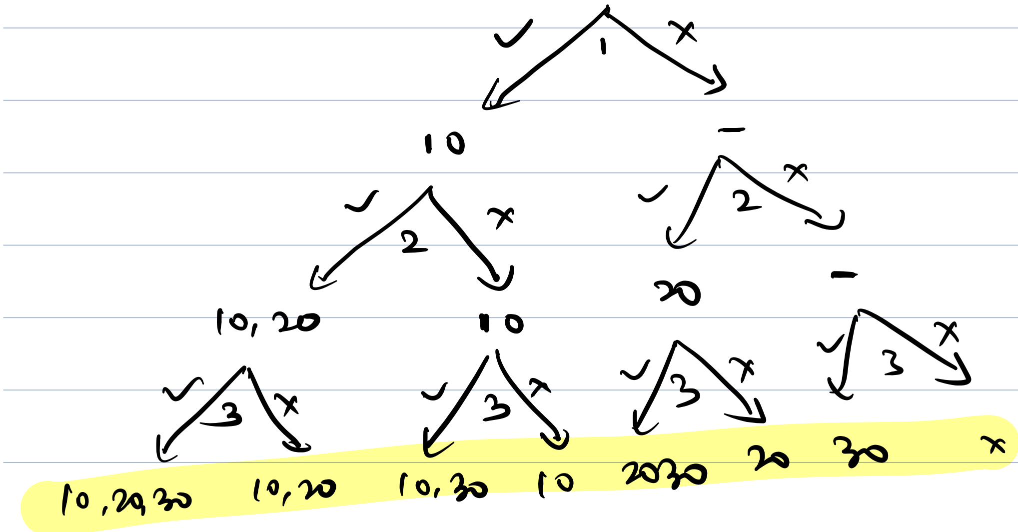
idea 1 :- Take elements in max value :
val : 150 + 60 ⇒ 210
Cap: 50 - 40 ⇒ 10 - 10 ⇒ 0

idea 2 :- Take elements in (v/w) ratio :-

val = 60 + 100 ⇒ 160
Cap = 50 - 10 = 40 - 20 ⇒ 20

1 2 3

10, 20, 30

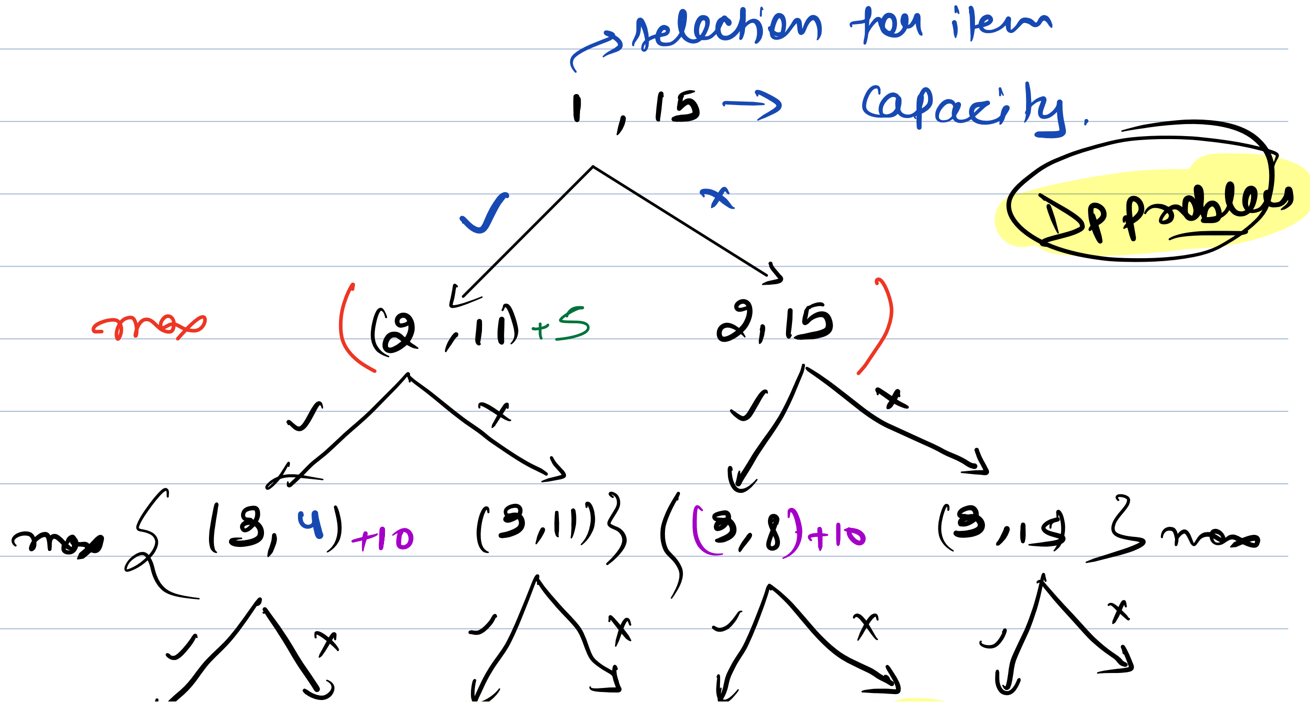


The above two were greedy approaches and they failed.

idea 3:- Get all subsets weight <= k & get max value.

Eg:- Cap = 15

N = 7	1	2	3	4	5	6	7
wt	4	7	3	4	5	1	4
Val	5	10	7	3	8	2	3



(4, 1) (4, 4) (4, 8) (4, 11) (4, 5) (4, 8) (4, 12) (4, 15)

```
public int helper(int[] val, int[] wts,
                  int idx, int cap, int[][] dp){
    if ( idx == wts.length) {
        return 0;
    }
    int selection = 0;

    if(dp[idx][cap] != 0) {
        return dp[idx][cap]
    }

    if (cap >= wts[idx] {
    selection = helper(val, wts,            -> dp
                       idx + 1, cap - wts[idx])
                + val[idx]

    }
                                            -> dp
    int rejection = helper (val, wts,
                            idx+1, cap);
  return dp[idx][cap]
        = Math.max(selection, rejection);
}
```

$T.C \rightarrow O(2^n)$

dp $\rightarrow$ Trading space for time.

$T.C \rightarrow O(n * cap)$
$S.C \rightarrow O(n) + O(n * Cap)$

Ques) Exactly same as above problem.

Note :- A single item can be picked as many times as we want.

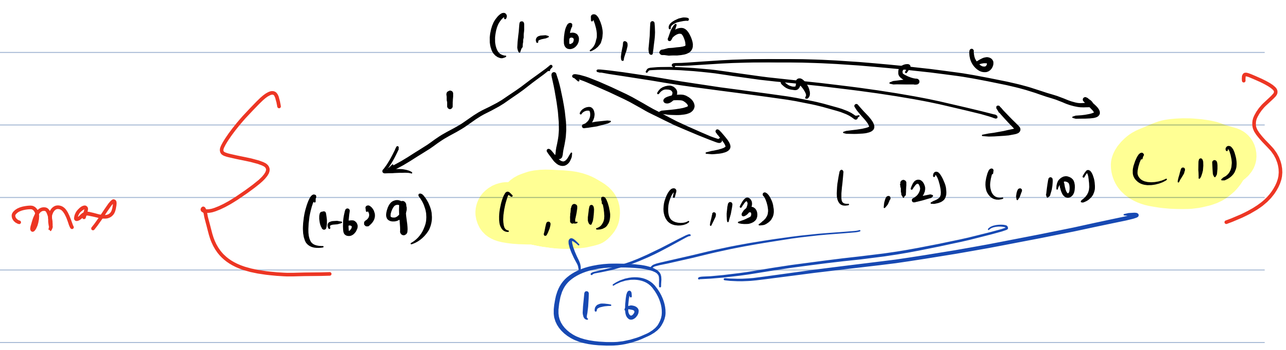
Ex:-

N =	1	2	3	4	Cap = 50
wts =	20	13	10	40	
val =	100	66	40	150	

$\rightarrow$ 200

N = 6	1	2	3	4	5	6
wt	6	4	2	3	5	4
val	10	3	7	5	8	7

Cap = 15.



int[] dp

int n,

```
int  unboundedKnapsack( int n, int cap, int[] wts, int[] val, int[] dp){
    if (cap<=0){
        return 0;
    }
    int max=0;
    for (int i=0 ; i<n; i++){
        if ( cap< wts[i]){
            continue
        } else {
            max = Math.max(max, unboundedKnapsack(
                  n, cap-wts[i], wts, val, dp) +
                  val[i]);
        }
    }
    return dp[cap] = max;
}
```

check if dp array is 0 or not.

Coin change Problem :-

Ques) Given a sum, and some coins with infinite supply of it, you have to find the no. of ways we can make the sum.

e.g.) sum = 4, coins = {1, 2, 3}

Soln:- { {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3} }

Let's try to do it with Tabulation:-

e.g, sum = 7, coins = {2, 3, 5}

0	1	2	3	4	5	6	7
1	0	1	1	1	2	1	2
·		·2	·3	·22	·23 ·5	·222	·223

int [] dp = new int [sum+1]

dp[0] = 1

for (i = 0; i < coins.length; i ++){

for (j = coins[i]; j < dp.length; j++) {

T.C →

O(amt × dp).

dp[j] = dp[j] + dp[j - coins[i]];

dp[6] = dp[6] + dp[6 - 3]

3

3

i

{2, 3, 5, 6}

j

0	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0		0	1	0	1

Notability → gpad.

Ques) Edit Distance :-

Give two strings str1 & str2,
below operations are allowed :-

1) Insert
2) Remove
3) Replace.

Min Operations → Convert str1 to str2.

e.g 1) str1 = 'cat' str2 = 'cut'

ans = 1 operation → a to u.

e.g. 2) 'sunday' to 'saturday'