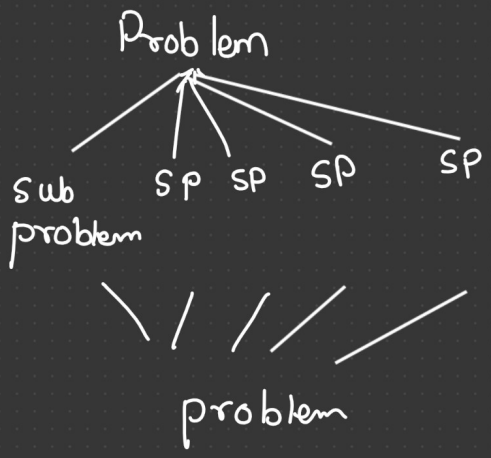
# Dynamic Programming




Recursion

Fibonacci Series :-

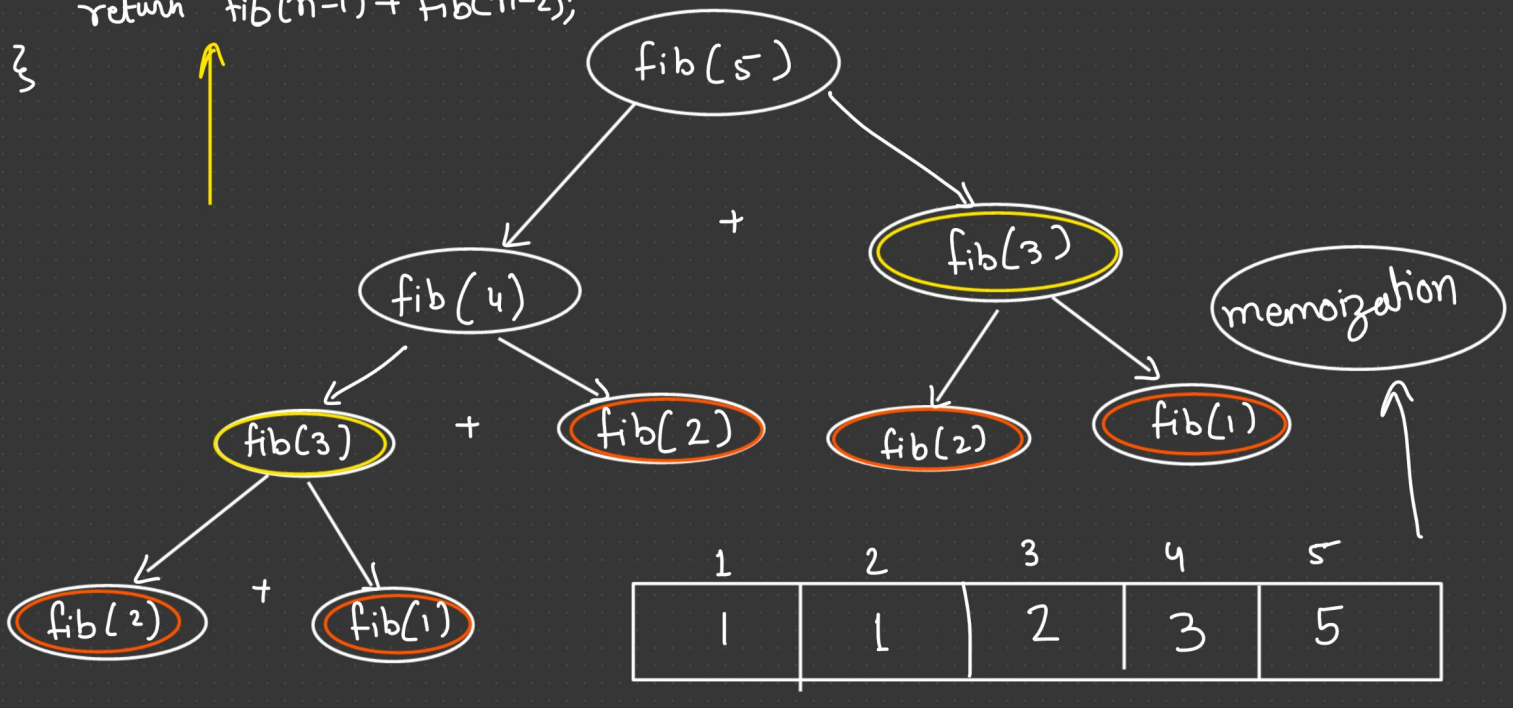
1, 1, 2, 3, 5, 8, 13, ....

$$fib(n) = fib(n-1) + f(n-2), \quad n > 2$$
$$f(1) = 1$$
$$f(2) = 1$$

$f(3) = fib(2) + fib(1) = 1 + 1 = 2$

$f(4)$ . . .

```
int fib ( int n)        (1)
{
  if (n == 1 || n == 2)
      return 1;
  else
  return fib(n-1) + fib(n-2);
}
```

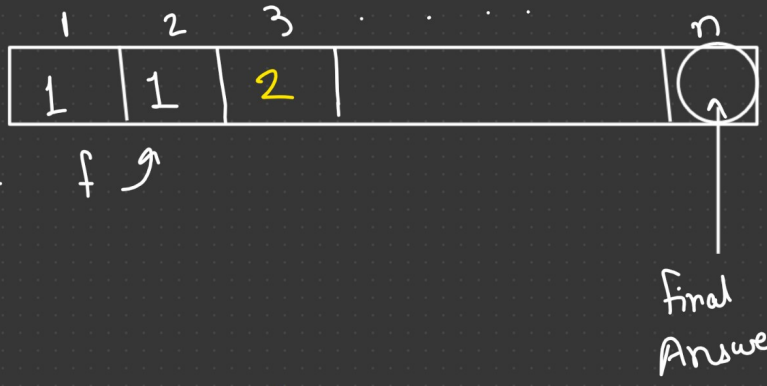


```
int f[n+1];   f[1]=f[2]=1;
int fib ( int n)
{
   if (n==1 || n==2)
      return 1;

   for(int i=3; i<=n ;i++)
    {
       f[i]= f[i-1] + f[i-2];
    }
    return f[n];
}
```

→ Top-down

→ O(n)




---

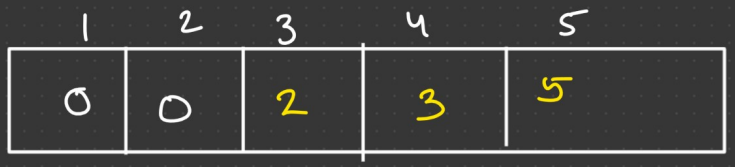
```
int fib ( int n)
{
   if (n==1 || n==2)
      return 1;

   if( f[n] != 0)
      return f[n];

   return f[n] = fib(n-1) + fib(n-2);
}
```

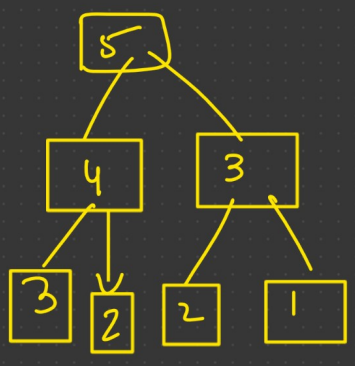
(2)

DP = Recursion + Memoization

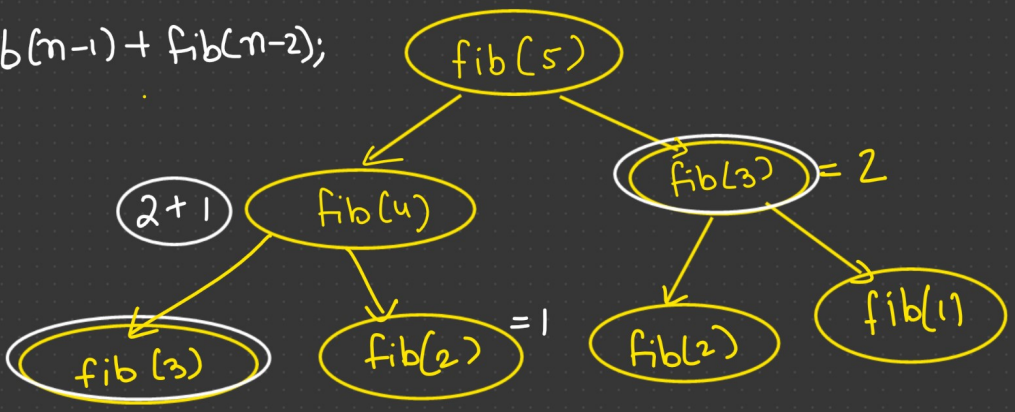



→ O(n)

→ O(n)

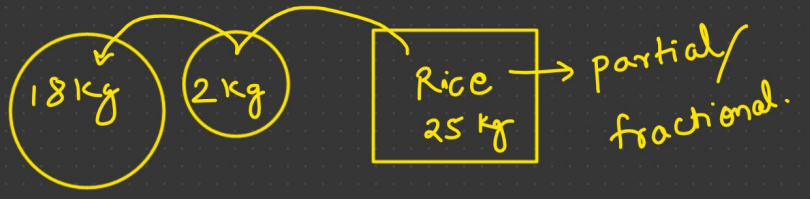


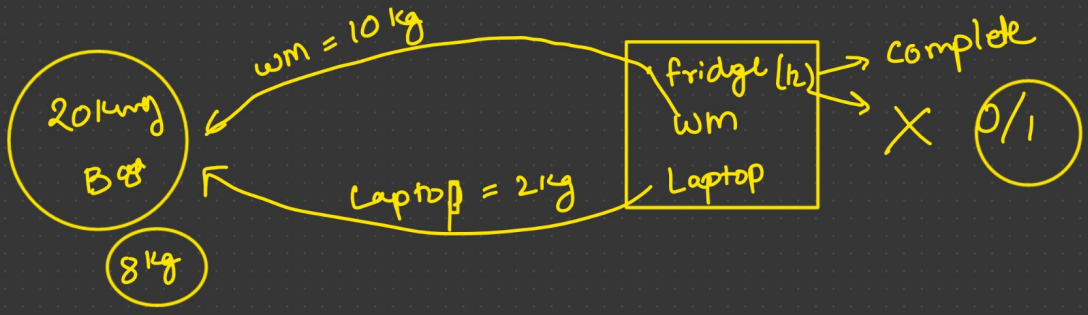

(2) =



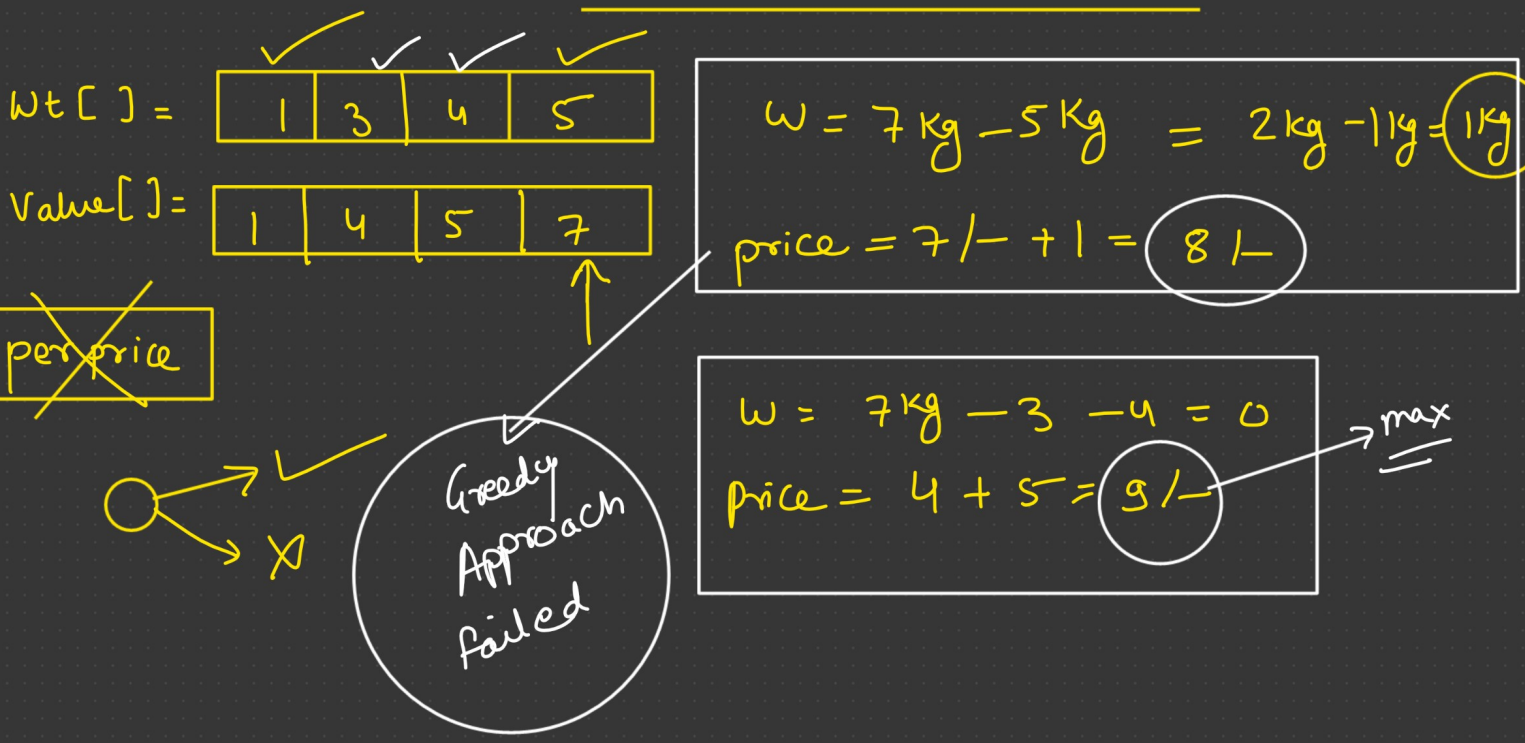
# 0/1 Knapsack problem

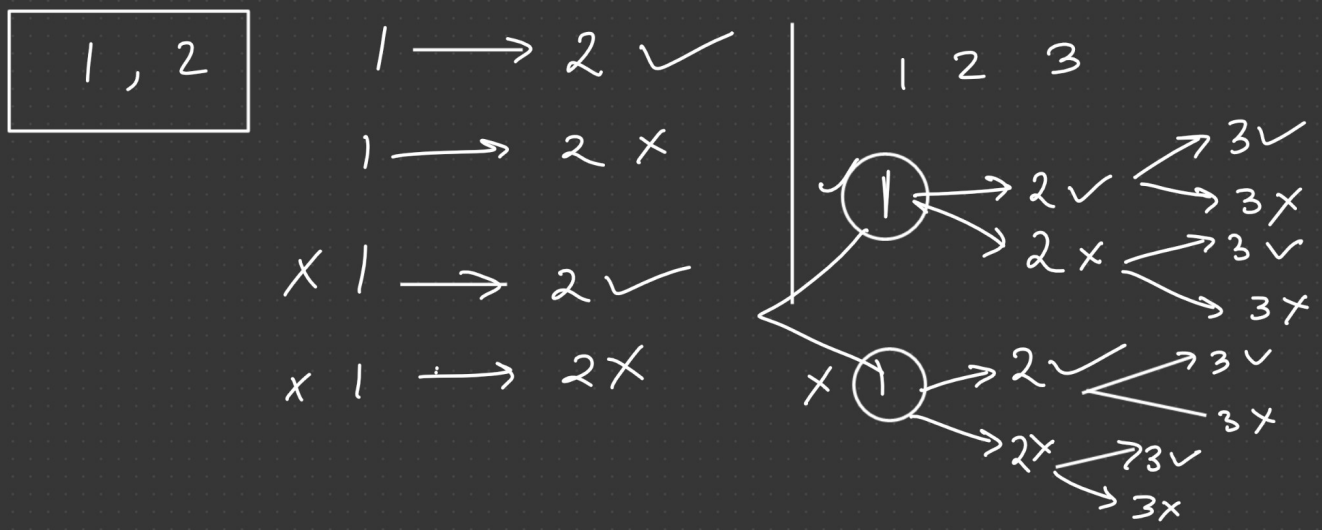
Knapsack problem
- 1) Fractional Knapsack Problem (Greedy Algorithm)
- 2) 0/1 Knapsack Problem → DP
- 3) Unbounded Knapsack problem.

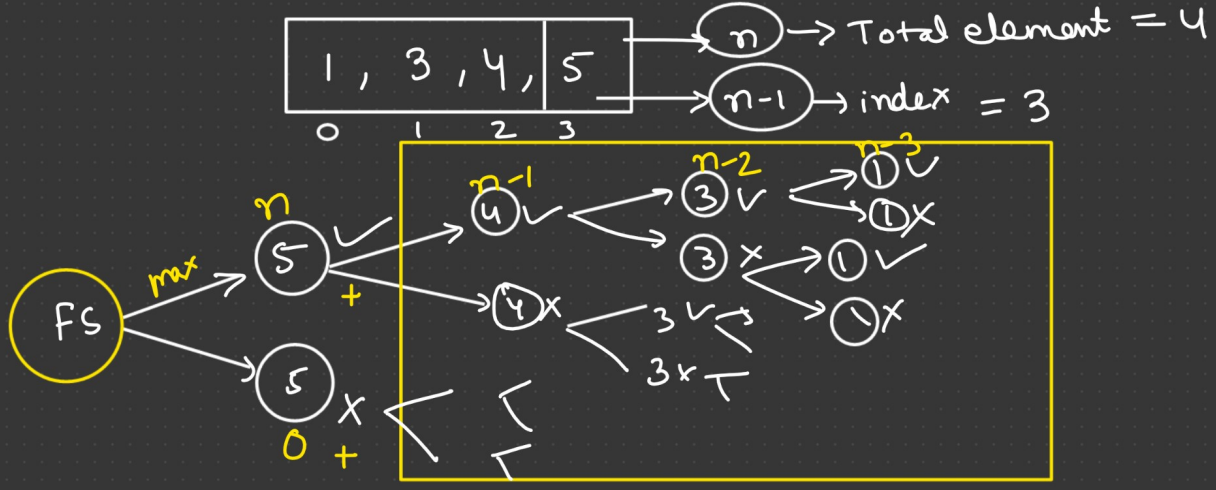




# 0/1 Knapsack Problem




It is growing exponentially



Given:-

```
int wt[] = { 1, 3, 4, 5 };
int value[] = { 1, 4, 5, 7 };
int W = 7;
int n = 4;
```

Change

```
int Knapsack( int wt[], int value[], int W, int n )
{
  // Base condition.
  if ( n == 0 || W = 0)
     return 0;

  if ( wt[n-1] <= W)
  {
    return max [ value[n-1] + Knapsack(wt, value, W - wt[n-1], n-1),
                 0 + Knapsack(wt, value, W, n-1) ];
  }
  else
  {
     return Knapsack ( wt, value, W, n-1);
  }
}
```

W = 0

n = 4 → product

3

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| wt | 1 | 3 | 4 | 5 |
| value | 1 | 4 | 5 | 7 |

# Using memoization

```
int wt[] = {        };
int value[] = {     };
```

```
int dp[n+1][w+1] = {-1};
```

```
int knapsack ( int w, int n)
{
    if( n==0 || w==0)
        return 0;
```

```
    if ( dp[n][w] != -1)
        return dp[n][w];
```

```
    if( wt[n-1] <= w)
    {
    return dp[n][w] = max{ value[n-1] + knapsack(w - wt[n-1], n-1),
                           0 + knapsack(w, n-1) };
    }
    else
        return dp[n][w] = knapsack(w, n-1);

    return dp[n][w];   → Final price.
}
```

$O(n^2)$

3 →
- n (which item I want check)
- wt (weight of $n^{th}$ element)
- price (final price)

| P | wt | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 3 | 2 | 0 | 1 | 1 | 4 | 8 | | |
| 5 | 4 | 3 | 0 | | | | | | |
| 7 | 5 | 4 | 0 | | | | | | |

→ wt of Bag (w)

dp[2][5] (n, w)

→ price of Bag (max)

Ans = dp[n][w]

item no (n)

Top-Down approach

| | 0 | 1 | 2 | 3 | 4 → wt |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 4 | 4 | 4 | 4 |
| 2 | 0 | 4 | 4 | 7 | 7 ← i-1 |
| 3 | 0 | 4 | 4 | 7 | (7) ← i |

↓ n

1st

| | 0 | 1 | 2 |
|---|---|---|---|
| wt | 1 | 2 | 3 |
| value | 4 | 3 | 2 |

W = 4, max = 7/-

KS( 4 , 3 )  (4 ← w, 3 ← n)

dp[i][w] = dp[i-1][w]

w++

```
for(i=0; i<=n ; i++)
{
  for(w=0; w<=W ; w++)
  {
  }
}
```

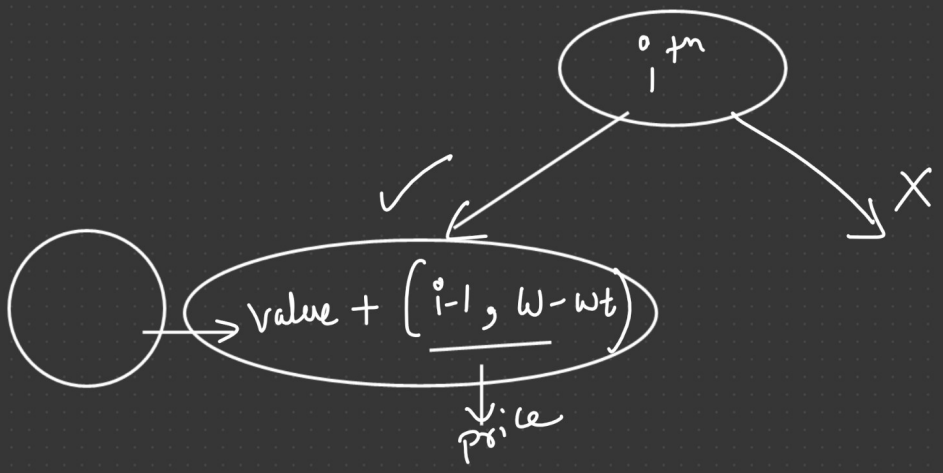
```
int knapsack ( int W, int n)
{
  for (int i=0 ; i<=n ; i++)
  {
    for (int w=0; w<=W ; w++)
    {
      if ( i==0 || w==0)
        dp[i][w] =0;
      else if ( wt[i-1]<= w)
      {
        dp[i][w] = max{ (value[i-1] + dp[i-1][w-wt[i-1]]) , dp[i-1][w] }
      }else
        d[i][w] = dp[i-1][w];
    }
  }
  return[n][w];
}
```

```
int dp[n+1][w+1] = {-1};
```

| | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 0 | | | | | | |
| 0 | | | | | | |
| 0 | | | | | | |

```
for( int i=0; i<=n ; i++)
  dp[i][0] =0;
for ( int j=0; j<=w ; j++)
  dp[0][j] =0;
```

0th i → wt[i-1] value[i-1]




| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 2 | 2 |
| 3 | 2 | 1 | 4 |

7

4 + [i-1, W-wt]

4 + 6 = 10