Dynamic Programming

The glorified Recursion with caching

Recursion

$$\left[\, 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \,\right]$$

```c
// Function to calculate factorial using recursion
int factorial(int n) {
    // Base case: factorial of 0 or 1 is 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case: n * factorial of n-1
    else {
        return n * factorial(n - 1);
    }
}
```

$n \cdot (n-1)!$ — $n!$

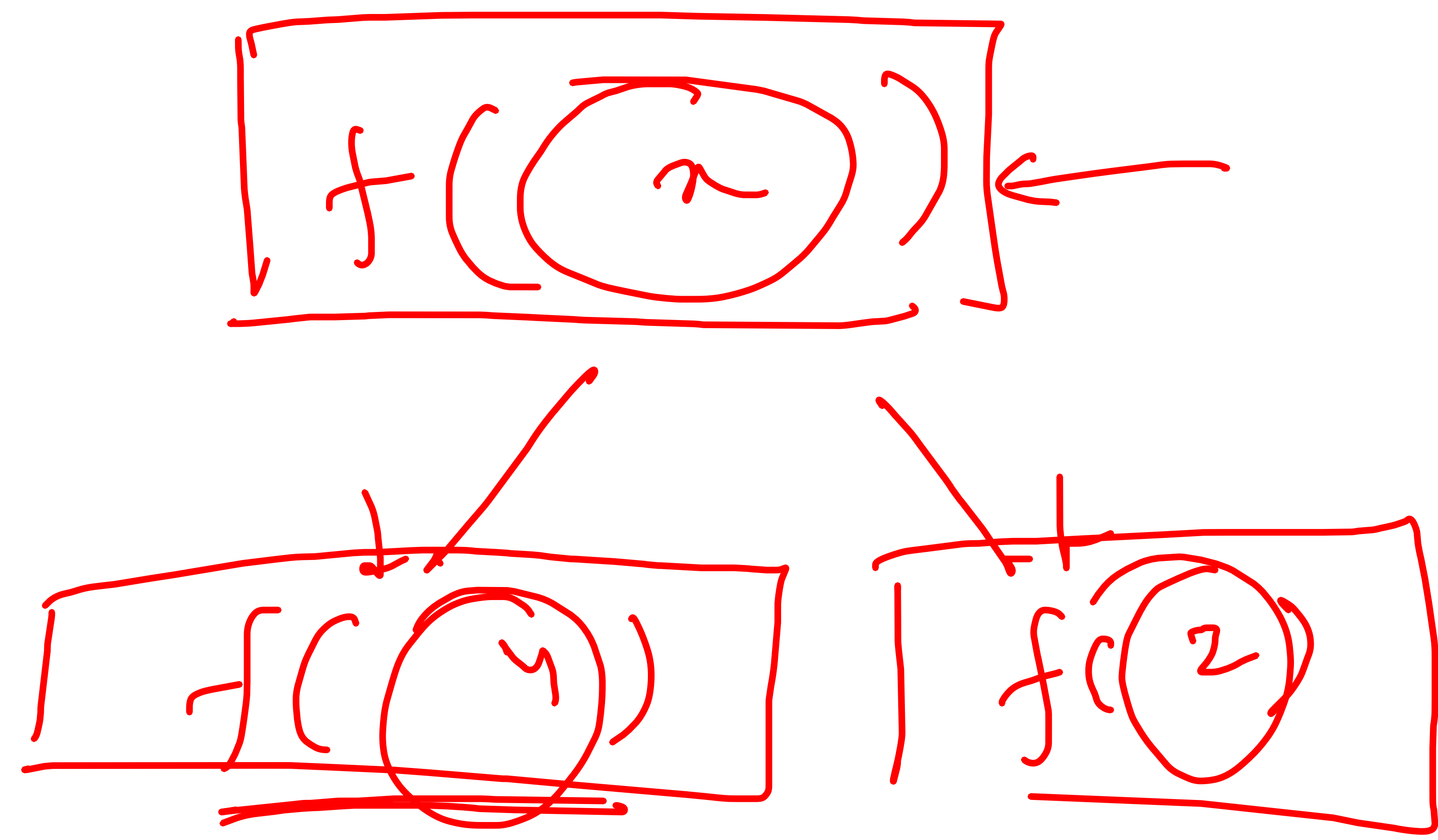
$$C(n,r) = \boxed{C(n-1,r-1)} + \boxed{C(n-1,r)}$$

$\boxed{{}^{n}C_{r}}$

${}^{4}C_{2}$

Visualization of the Magic of DP

2 Key properties for DP !



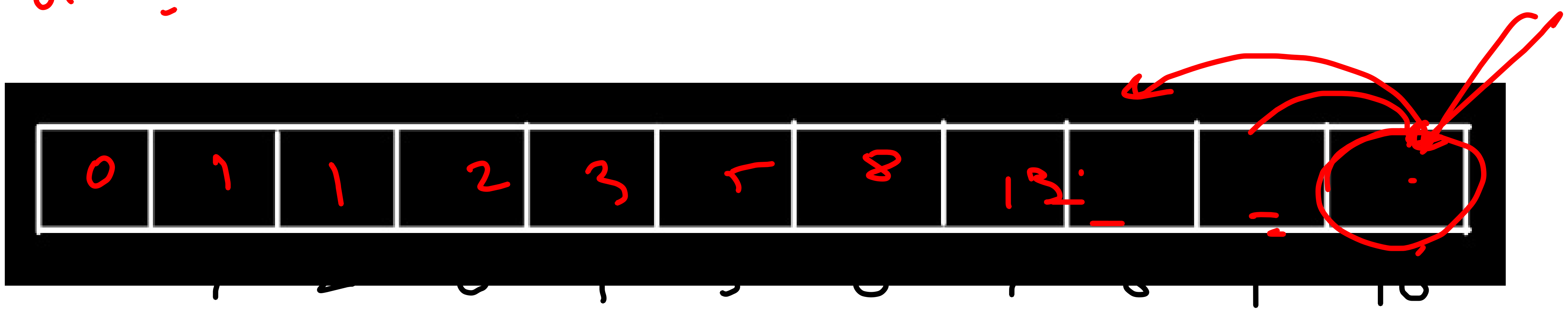
→ **Optimal Substrcuture**

1 → **Overlapping Subproblems**

2 Ways of Calculating DP - Top Down

Fib(10)

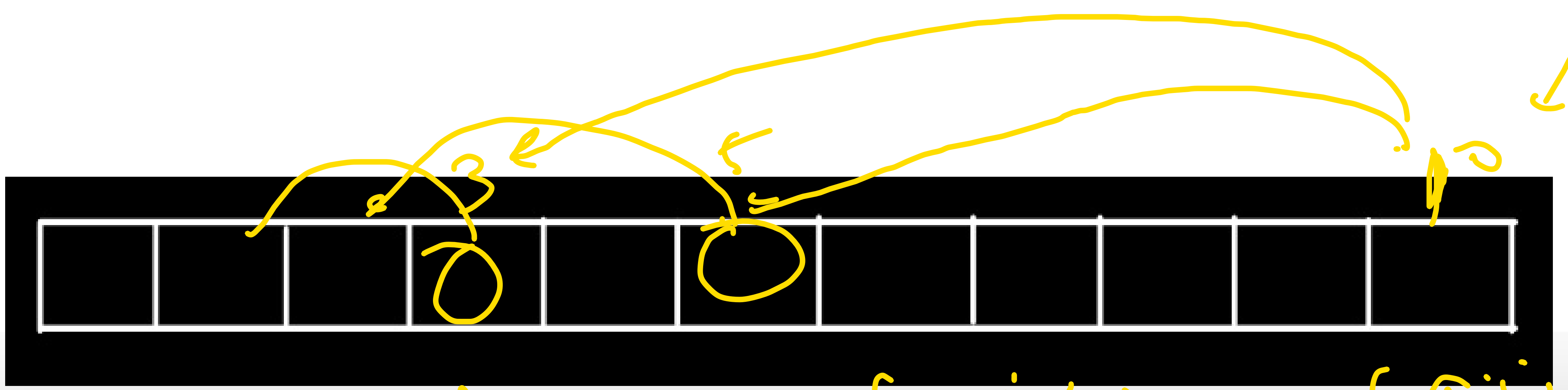
Memo :




```cpp
// Function to calculate Fibonacci using top-down recursion with
memoization
long long fibonacci(int n, std::vector<long long>& memo) {
    // Check if the value has already been computed
    if (memo[n] != -1) {
        return memo[n];
    }
    // Base cases
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Recursive calculation with memoization
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
    return memo[n];
}
```

2 Ways of Calculating DP - Bottom Up

Memo :



$f(i) = f(i/2) + f(i/3)$

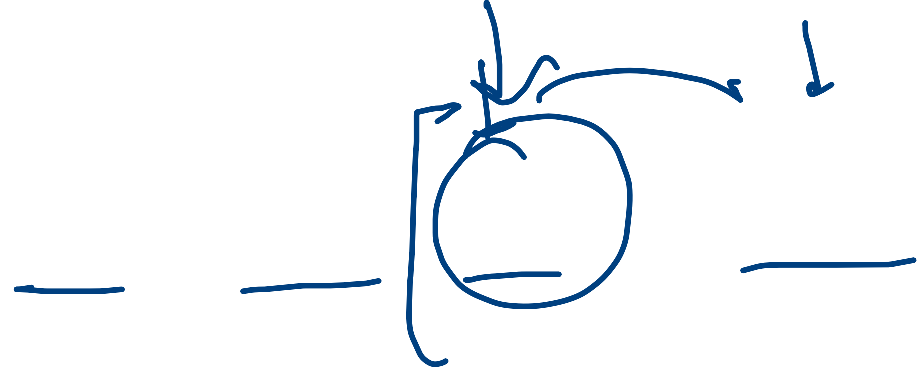
```
// Function to calculate Fibonacci using bottom-up approach with a memo array as argument
void fibonacci(int n, std::vector<long long>& memo) {
    memo.resize(n + 1); // Resize the vector to hold all necessary Fibonacci numbers
    memo[0] = 0; // Base case F(0)
    memo[1] = 1; // Base case F(1)

    // Fill in the Fibonacci numbers in the memo array from 2 to n
    for (int i = 2; i <= n; i++) {
        memo[i] = memo[i - 1] + memo[i - 2];
    }
}

```

Which one is better?

Vivek, this is all too simple for me... show me how to solve !!



Q. N items, each item can be taken **only once!**

weights

$$\left[\begin{pmatrix} w_1 \\ p_1 \end{pmatrix} \quad \begin{pmatrix} w_2 \\ p_2 \end{pmatrix} \cdots \begin{pmatrix} w_n \\ p_n \end{pmatrix}\right]$$

profit

You can take,

1. Atmost W weight
2. Atmost K items.

constraints

You have to Maximize sum of Profits of items taken.

How will you decide to take the items??

Tok /n7

Test m ⊢

Framework

Step 1 : Recognise Form1 → Form

j T/N i

[0 . . . N]

[i . . . N]

T/N

subse

Step 2 : Design Recurrence / State   G5 %

Dp(i , w_taken , k_taken) → [Maximum sum of profit from [i ... N-1]] → what is asked in the problem

form, Restrictions

N-1   N

W K

i

Step 3 : Design Transition

w taken
k taken.

i

i+1

Take

Not take

dp(i , wtaken , ktaken)

P[i]

[Take] → [dp(i+1 , wtaken + w[i] , ktaken +1)]

Not take → 0 → [dp(i+1 , wtaken , ktaken)]

max

Step 4 : TLE check

TL = [#S (1 + Avg # of Transition)]

O(N.W.K (1 + 2))

N, K, W ≤ 100

O(N.K W)

100. W.. K. - 10^6

1 sec → 10^8

Dp(i , wtaken , k taken)

0 ... N   0 ... W   0 ... K

[N.W.K]

$$\begin{array}{c} w \\ p \end{array} \begin{pmatrix}1\\3\end{pmatrix} \begin{pmatrix}5\\5\end{pmatrix} \begin{pmatrix}2\\4\end{pmatrix} \begin{pmatrix}6\\8\end{pmatrix}$$

$k = 2$

$3 \le 6$

$w = 5$

$k = 2$

How do you think ??

1 Subset / subseq / subarray

2 Counting / Optimization

Formil

T/NT   T/NT   T/NT

$$\left\{ \begin{pmatrix} w_1 \\ p_1 \end{pmatrix} \begin{pmatrix} w_2 \\ p_2 \end{pmatrix} \cdots \begin{pmatrix} w_n \\ p_n \end{pmatrix} \right\}$$

2^N