

Dynamic programming

Dynamic programming is a technique for solving certain type of complex problems **efficiently** by breaking them down into simpler subproblems and solving each subproblem **exactly once**. Dynamic programming stores the results of subproblems in a table and reuse them when needed to avoid solving the same subproblems again and again.

What types of problem can be solved using DP?

The problem should have following properties

1. Optimal substructure
2. Overlapping subproblems

Optimal Substructure

- A given problem has Optimal Substructure Property if optimal solution of the given problem can be obtained using optimal solutions of its subproblems.
- If we can define the solution to the problem by using a recurrence relationship based on its subproblems.
- Dynamic programming is very useful in solving optimization problems.

Overlapping subproblems

A given problem has overlapping subproblems property if to solve the problem we have to solve its subproblems multiple times.

Fibonacci number

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = 0 + 1 = 1$$

$$\text{Fib}(3) = 1 + 1 = 2$$

$$\text{Fib}(4) = 2 + 1 = 3$$

$$\text{Fib}(5) = 3 + 2 = 5$$

$$\text{Fib}(6) = 5 + 3 = 8$$

$$\text{Fib}(7) = 8 + 5 = 13$$

$$\text{Fib}(8) = 13 + 8 = 21$$

$$\text{Fib}(9) = 21 + 13 = 34$$

$$\text{Fib}(10) = 34 + 21 = 55$$

$$\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$$

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

Java

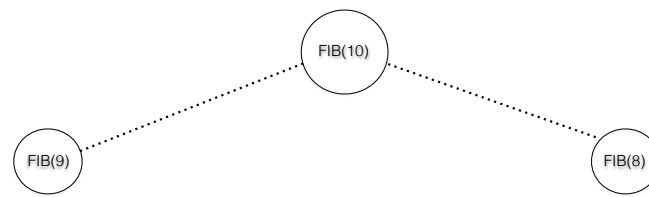
```
public static int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n - 1) + fib(n - 2);  
}
```

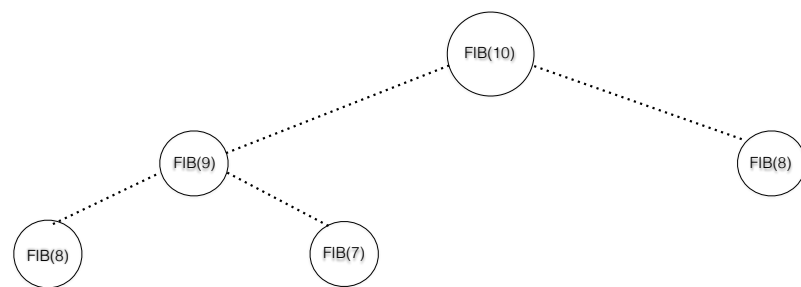
Python

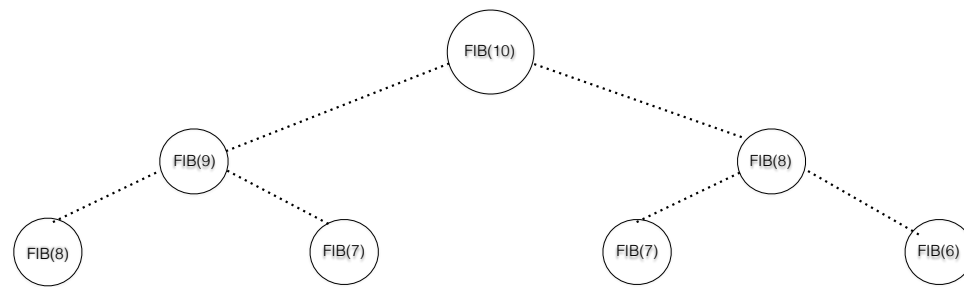
```
def fib(N):  
    if N == 0:  
        return 0  
    if N == 1:  
        return 1  
    return fib(N - 1) + fib(N - 2)
```

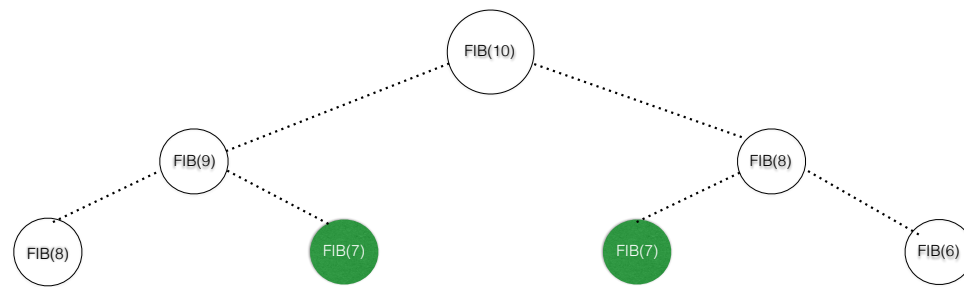
$$\text{FIB}(N) = \text{FIB}(N-1) + \text{FIB}(N-2)$$

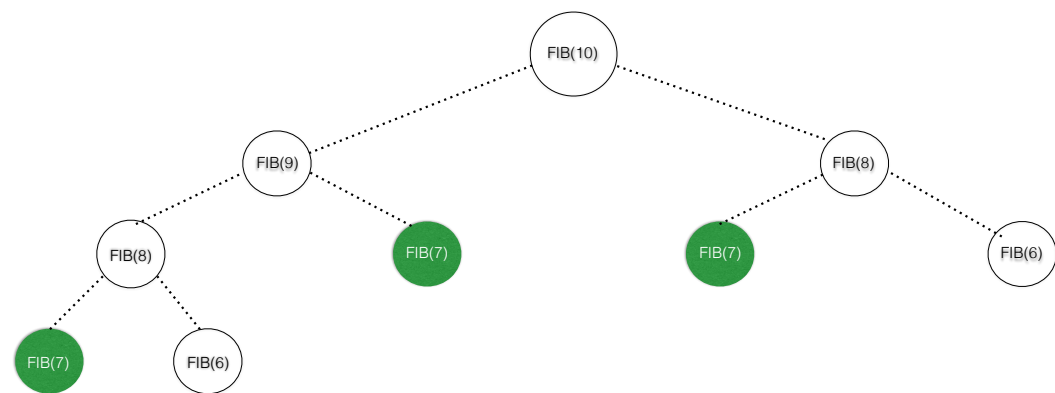
$\text{FIB}(10)$

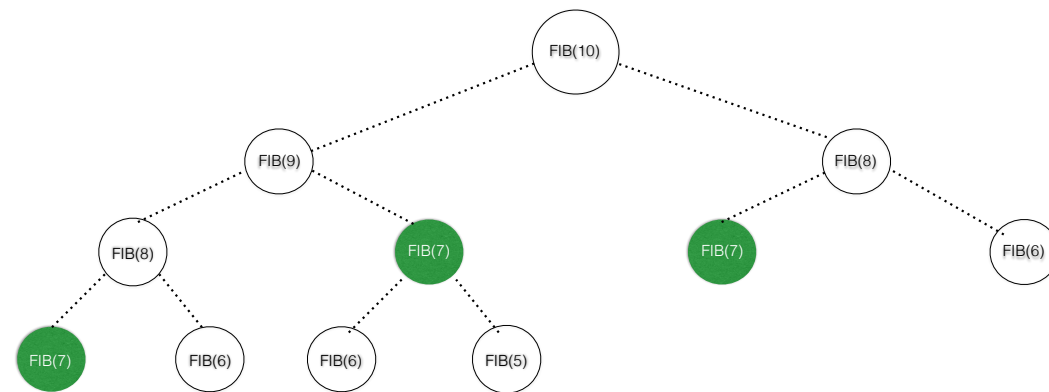


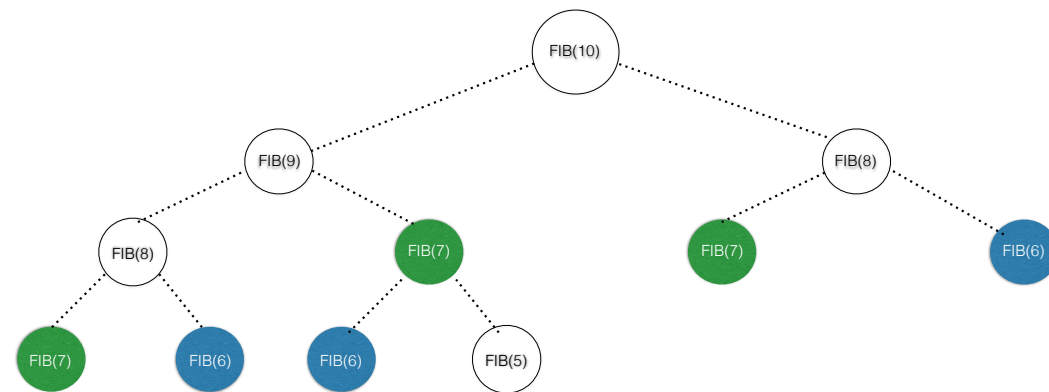


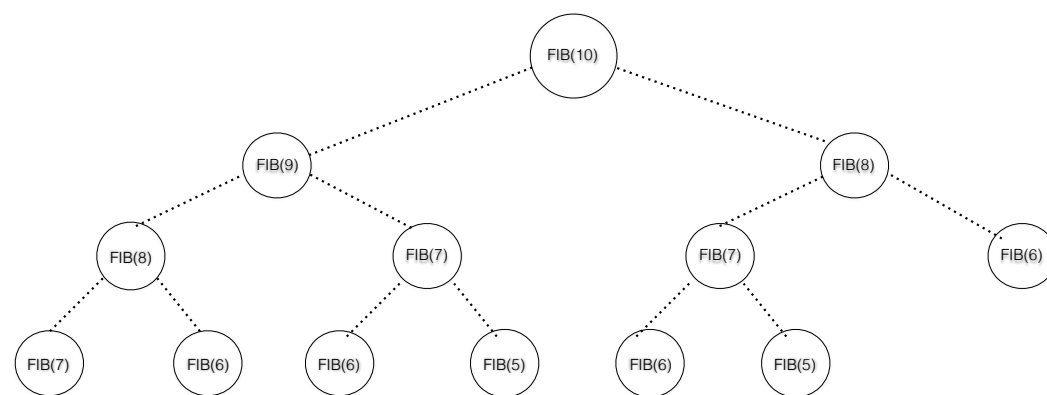


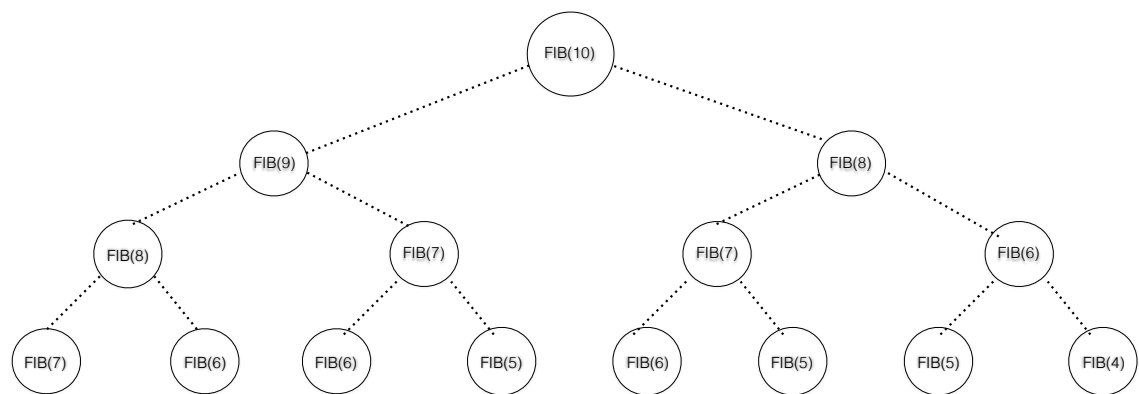


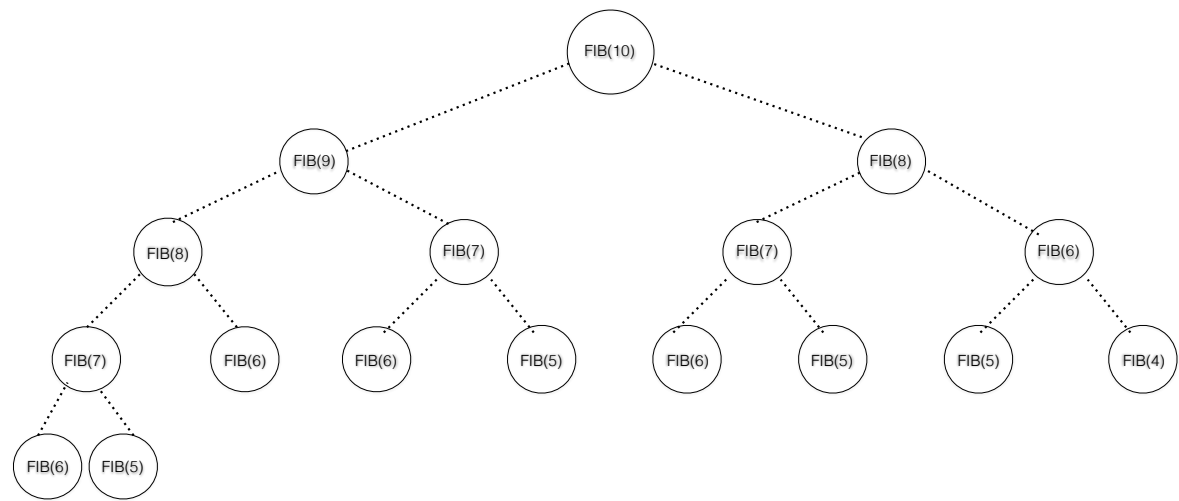


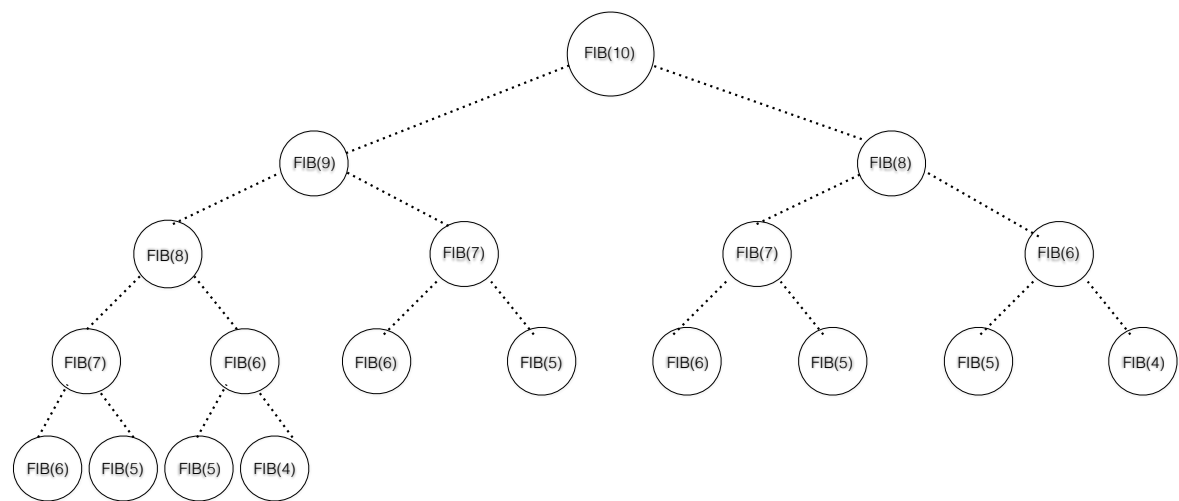


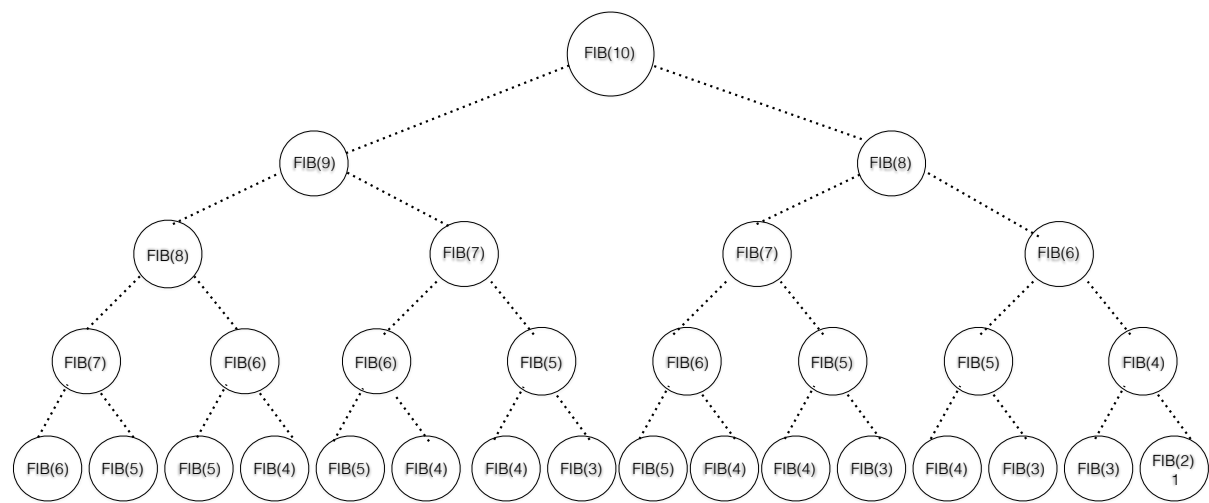


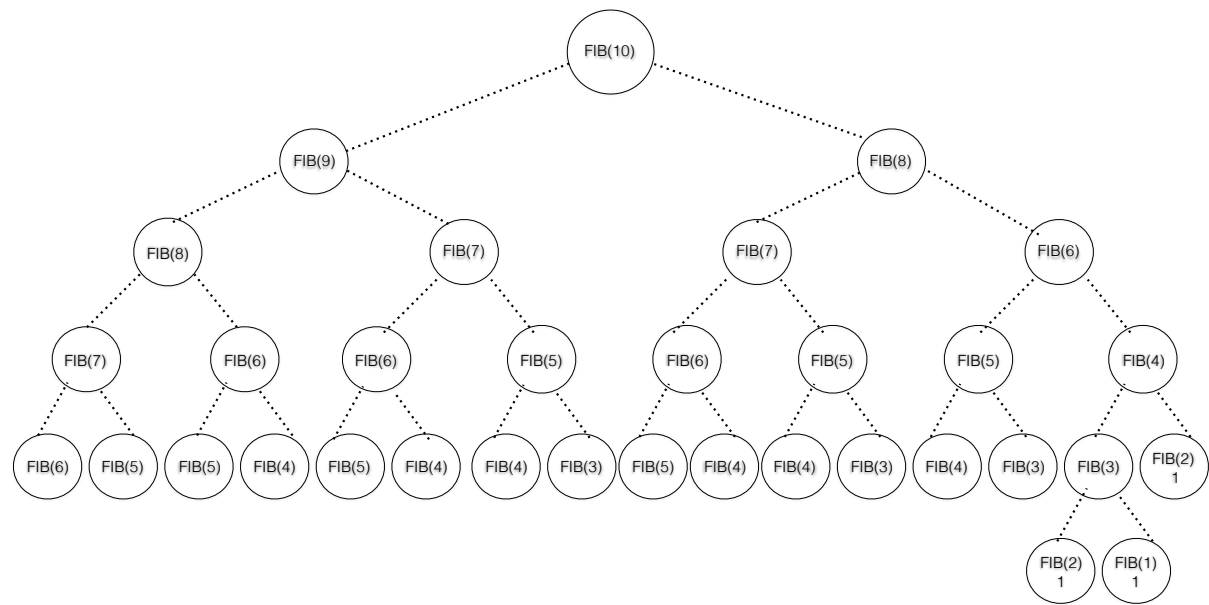


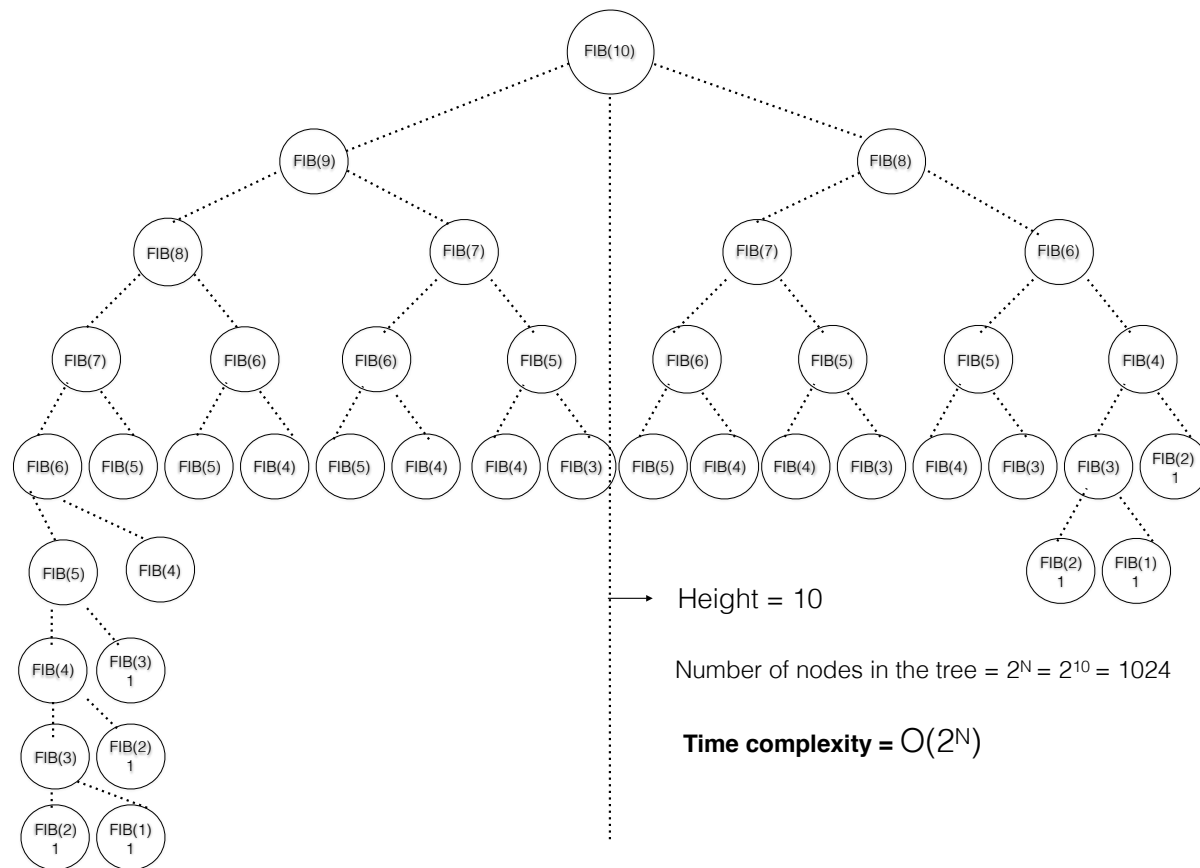












T
i
m
e

c
o
m
p
l
e
x
i
t
y

Exponential $O(2^n)$

Quadratic $O(n^2)$

Linear $O(n)$



Dynamic Programming to the rescue !!

Using Dynamic Programming we avoid solving overlapping subproblems and we solve each subproblem only once and save the result in a cache. When its needed again we will get the result from the cache instead of solving then again.

Time and space tradeoff

Dynamic programming is essentially a tradeoff of space for time.

Dyanmic programming = recursion + caching

Dynamic programming approaches

1. Top down approach (Memoization)
2. Bottom up approach (Tabulation)

Top down approach

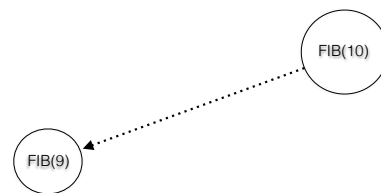
1. Implemented using recursion and caching.
2. Whenever the recursive function is called we check the cache to see if the problem has already been solved. If its already solved then we return the result from the cache else we will solve the subproblem, save the result in a cache and return the result.

$$\text{FIB}(N) = \text{FIB}(N-1) + \text{FIB}(N-2)$$

FIB(10)

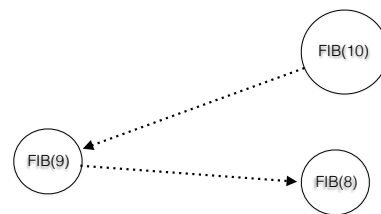
Cache

Key	Value
0	0
1	1



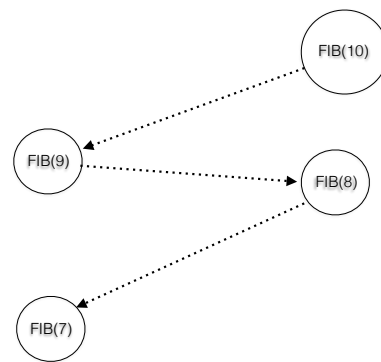
Cache

Key	Value
0	0
1	1



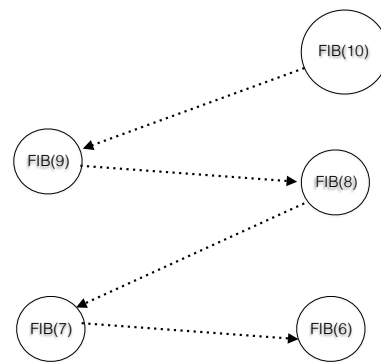
Cache

Key	Value
0	0
1	1



Cache

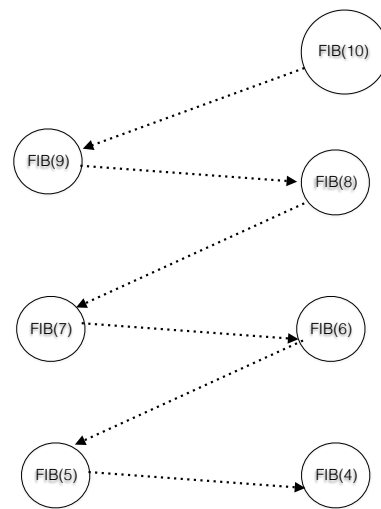
Key	Value
0	0
1	1



Cache

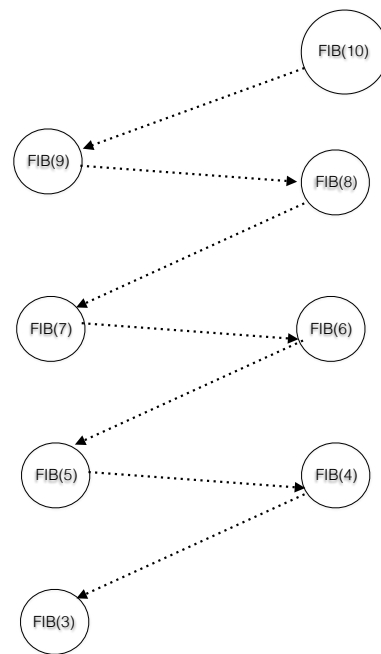
Key	Value
0	0
1	1

Key	Value
0	0
1	1



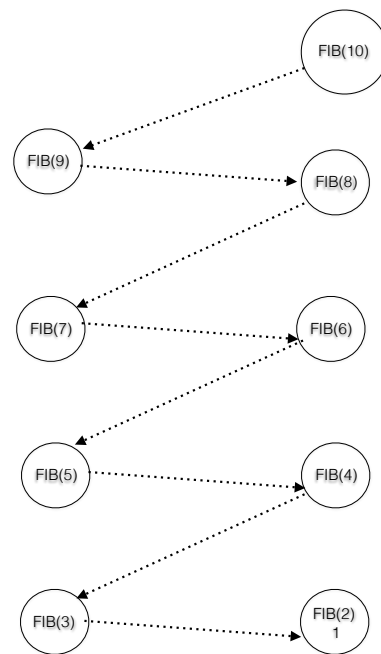
Cache

Key	Value
0	0
1	1



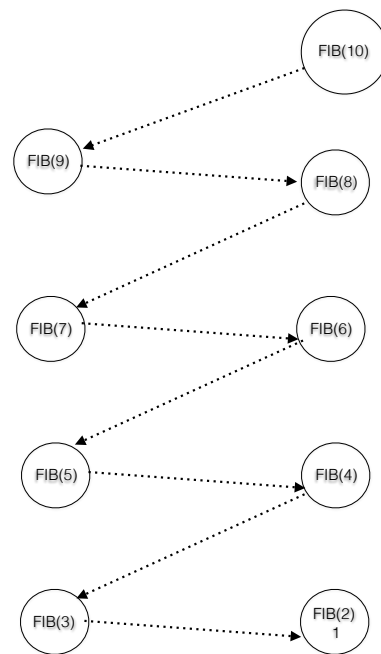
Cache

Key	Value
0	0
1	1



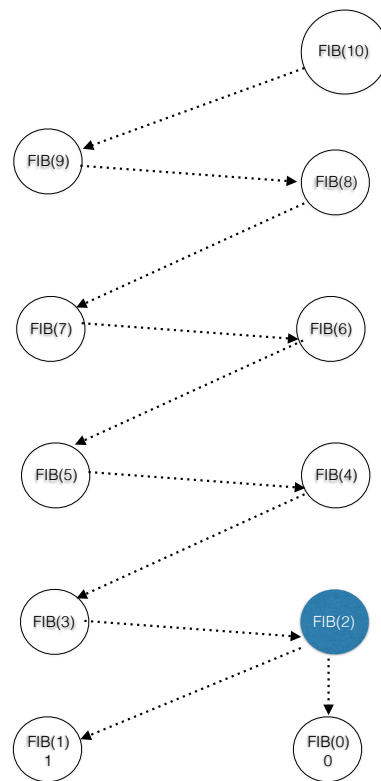
Cache

Key	Value
0	0
1	1



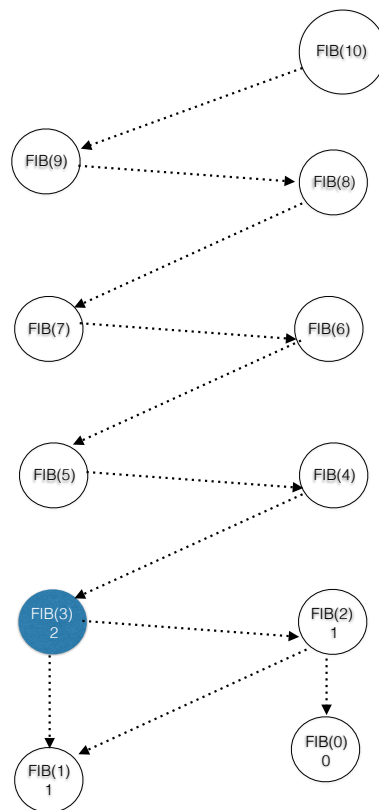
Cache

Key	Value
0	0
1	1



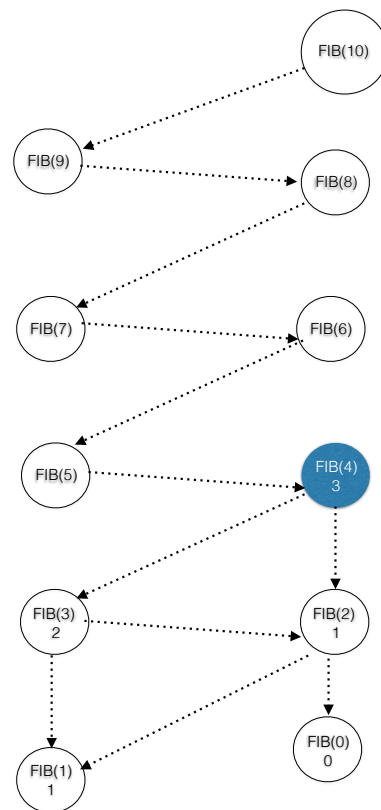
Cache

Key	Value
0	0
1	1
2	1



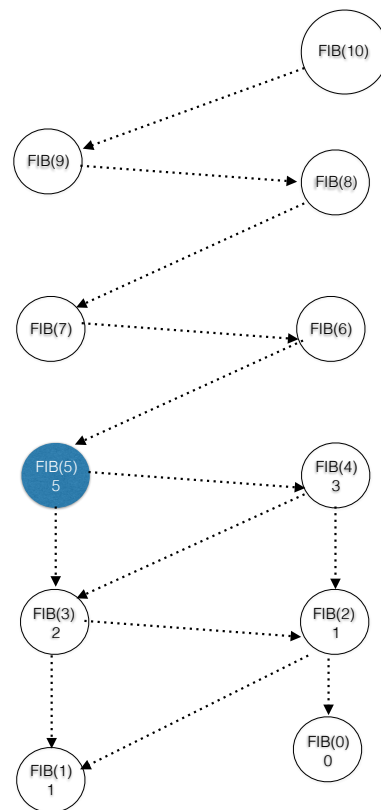
Cache

Key	Value
0	0
1	1
2	1
3	2



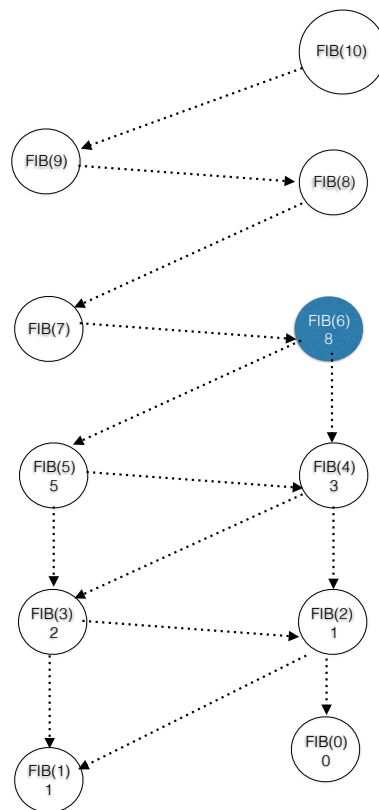
Cache

Key	Value
0	0
1	1
2	1
3	2
4	3



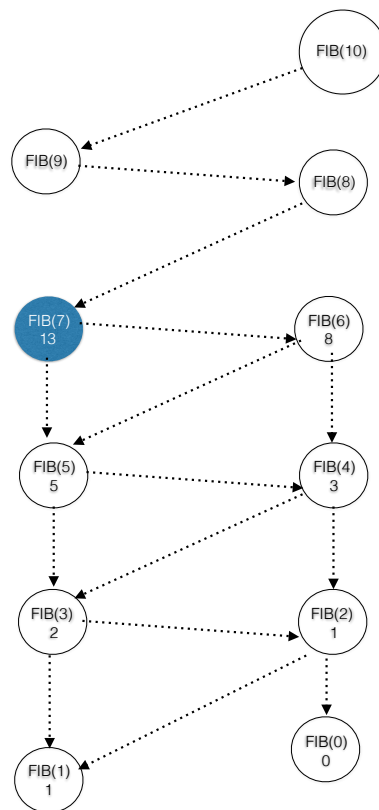
Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5



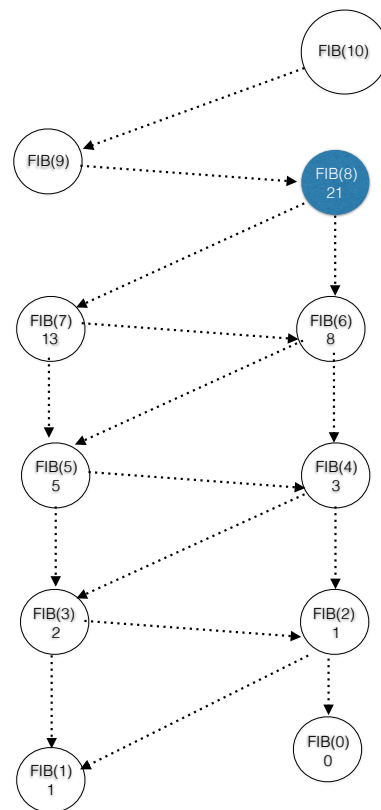
Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8



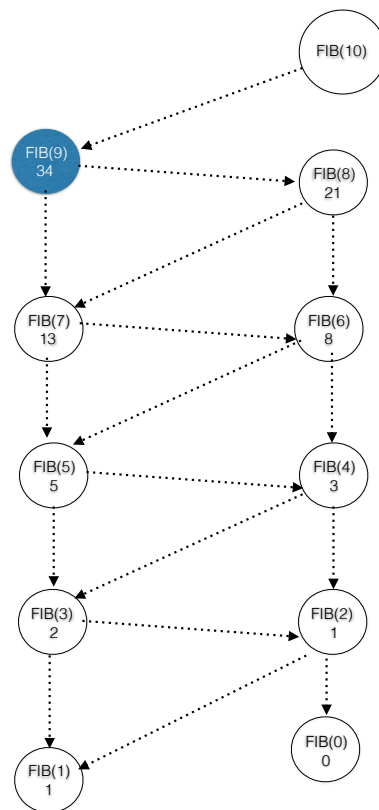
Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13



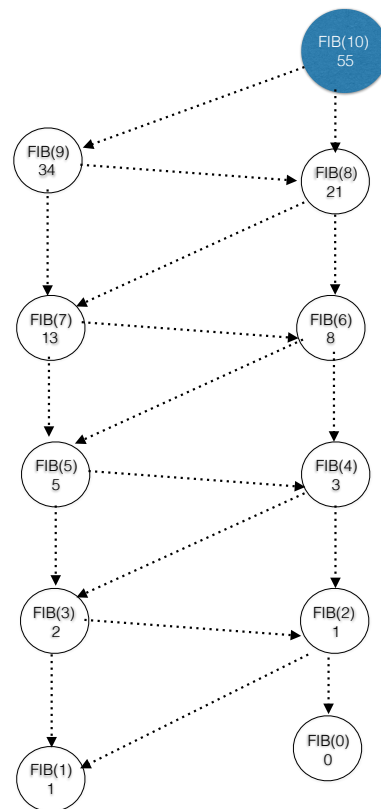
Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21



Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34



Cache

Key	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

Time complexity = $O(N)$

Java

```
public static int fibMemo(int n, int[] cache) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    if (cache[n] != 0) {  
        return cache[n];  
    }  
    int res = fibMemo(n - 1, cache) + fibMemo(n - 2, cache);  
    cache[n] = res;  
    return res;  
}
```

Python

```
def fib_mem(N, cache):  
    if N == 0:  
        return 0  
    if N == 1:  
        return 1  
    if cache[N] != 0:  
        return cache[N]  
    res = fib_mem(N - 1, cache) + fib_mem(N - 2, cache)  
    cache[N] = res  
    return res
```

Top down approach

Pros:

1. Very easy to understand and implement
2. Solves the sub problems only when its needed. **On demand**
3. Easy to debug.

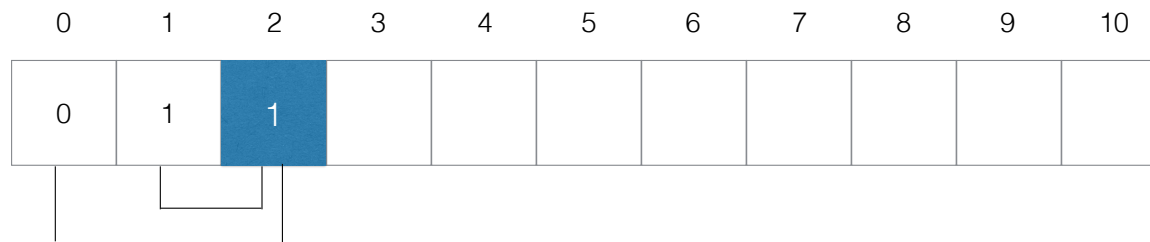
Cons:

1. Uses recursion which uses more memory in the call stack. If the recursion is too deep then it may cause StackOverflow.
2. Performance overhead due to recursion.

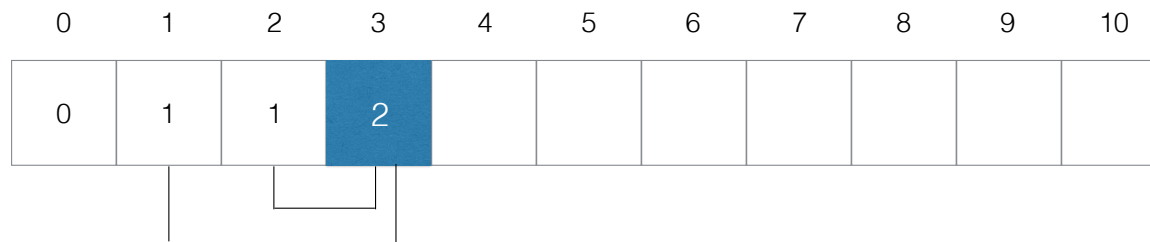
Bottom up approach / Tabulation

1. We solve all the smaller sub problems first which will be needed to solve the larger problems then move on to solving larger problems using the results of smaller subproblems.
2. We use for-loop to iterate over all the subproblems and solve them.
3. Also called tabulation method or table filling method.

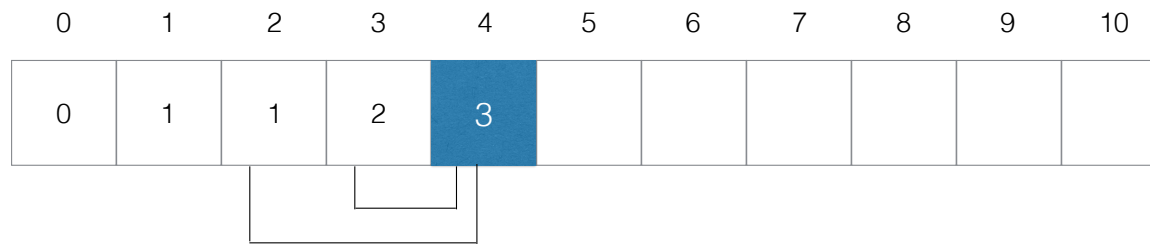
Fibonacci Bottom up



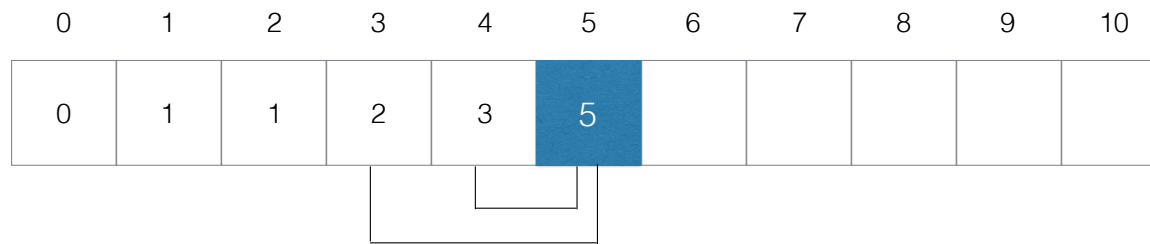
Fibonacci Bottom up



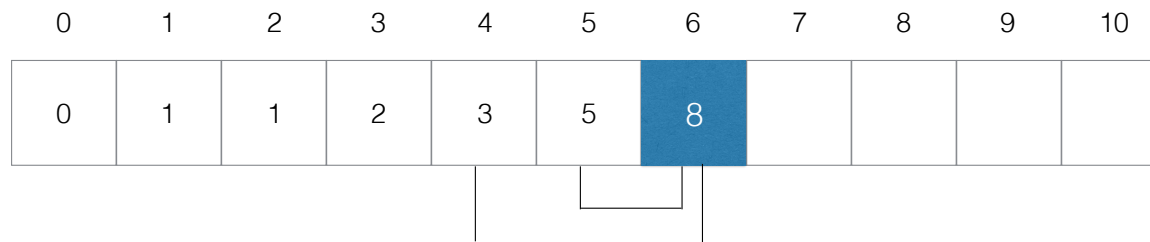
Fibonacci Bottom up



Fibonacci Bottom up

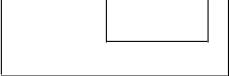


Fibonacci Bottom up



Fibonacci Bottom up


0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13			



The diagram illustrates the bottom-up calculation of the 7th Fibonacci number (13). The table shows the sequence of Fibonacci numbers from index 0 to 10. The value 13 is highlighted in the cell for index 7. Below the table, a bracket indicates the addition of the 5th and 6th Fibonacci numbers (5 + 8) to produce the 7th Fibonacci number (13).


Fibonacci Bottom up

0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13	21		



Fibonacci Bottom up

0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13	21	34	



Fibonacci Bottom up

0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13	21	34	55

Java

```
public static int fib(int n){
    int[] dp = new int[n+1];
    dp[0]=0;
    dp[1]=1;
    for(int i=2;i<=n;i++){
        dp[i]=dp[i-1]+dp[i-2];
    }
    return dp[n];
}
```

Python

```
def fib_dp(N):
    if N == 0: return 0
    if N == 1: return 1
    dp = [0 for _ in range(0, N + 1)]
    dp[1] = 1
    for i in range(2, N + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[N]
```

Bottom up approach

Pros:

1. Its efficient. Recursion adds lot of over head which make recursive approach slow.
2. It also space efficient. Recursion uses space in call stack.

Cons:

1. Its difficult to implement as compared to top down approach. Its not very intuitive. But with practice it becomes easier as most algorithms use few common techniques.
2. Its solves all the subproblems once whether its used or not as opposed to top down approach where the subproblems are solved on demand.

Exercise

Binomial coefficient / Combinations

$\binom{n}{k}$ n choose k , denote number of ways to choose k items from n items where sequence does not matter.

Formula

$$\binom{n}{k} = \frac{n!}{(n-k)! k!}$$

Recursive formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \binom{n}{n} = 1, \binom{n}{0} = 1$$

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

Base case

$$C(n,n) = 1, C(n,0) = 1$$

Java

```
public static int binomialCoefficient(int n, int k){  
    if(n == k || k == 0){  
        return 1;  
    }  
    return binomialCoefficient(n-1,k-1)+  
binomialCoefficient(n-1,k);  
}
```

Python

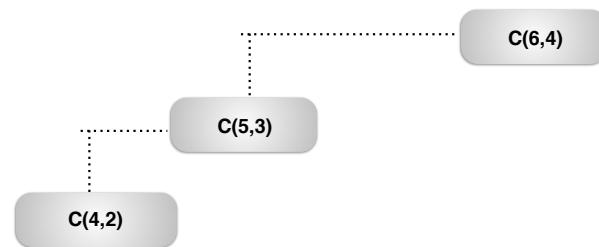
```
def binomial_coefficient(n, k):  
    if n == k or k == 0: return 1  
    return binomial_coefficient(n - 1, k - 1) +  
binomial_coefficient(n - 1, k)
```

C(6,4)

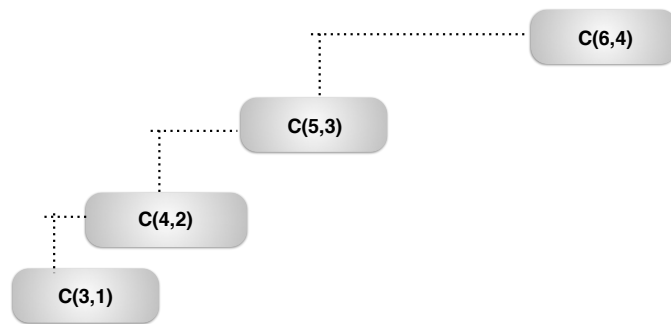
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



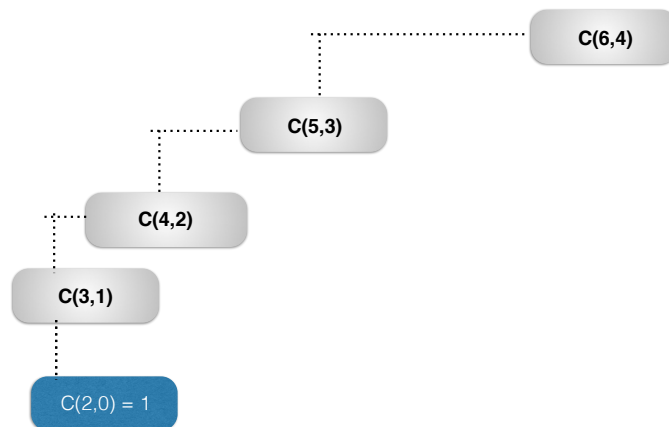
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



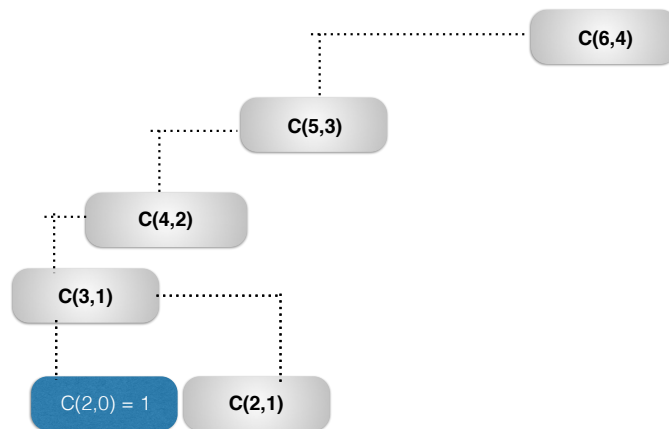
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



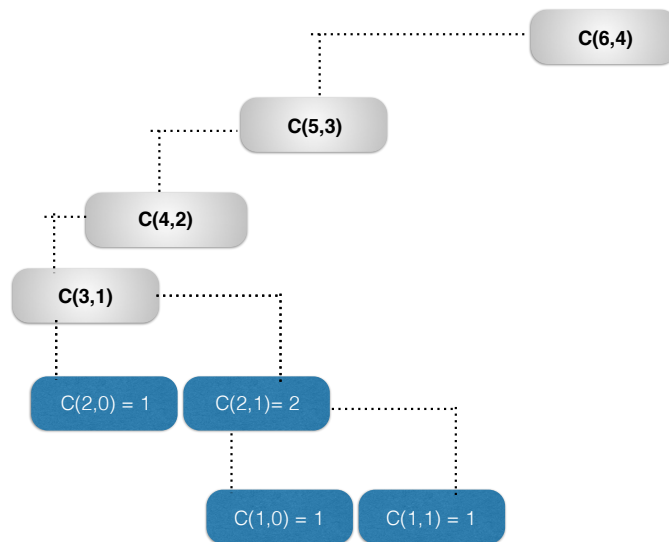
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



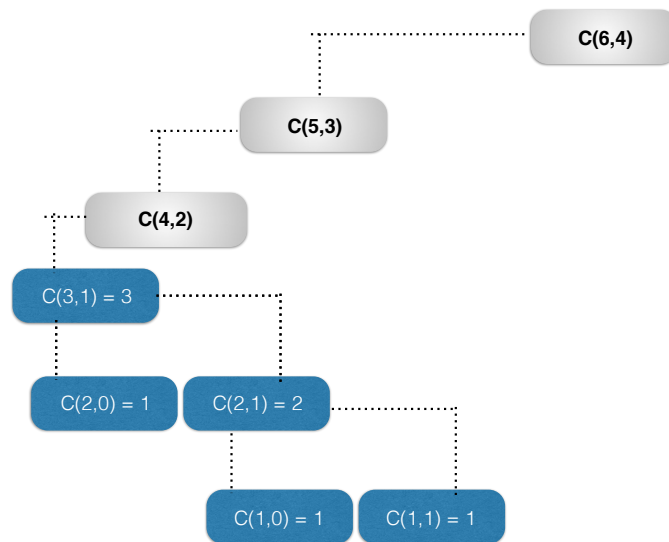
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



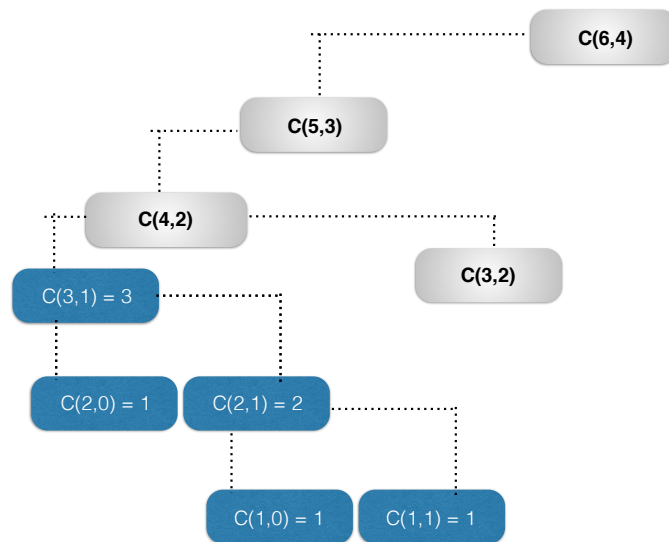
	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					



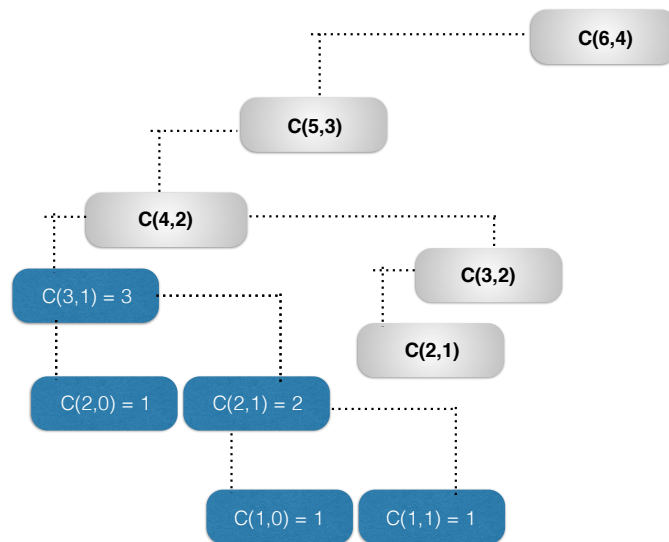
	0	1	2	3	4
0					
1					
2		2			
3					
4					
5					
6					



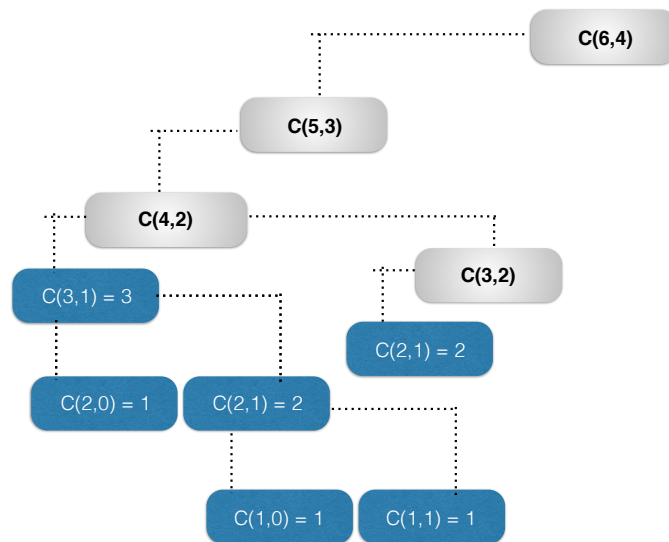
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



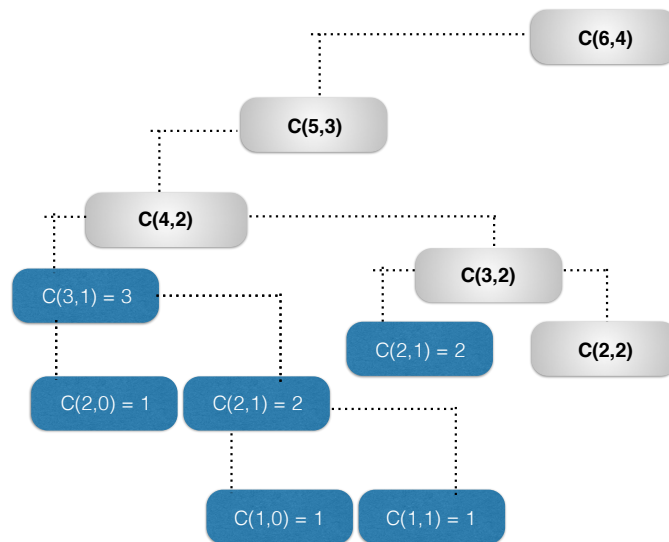
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



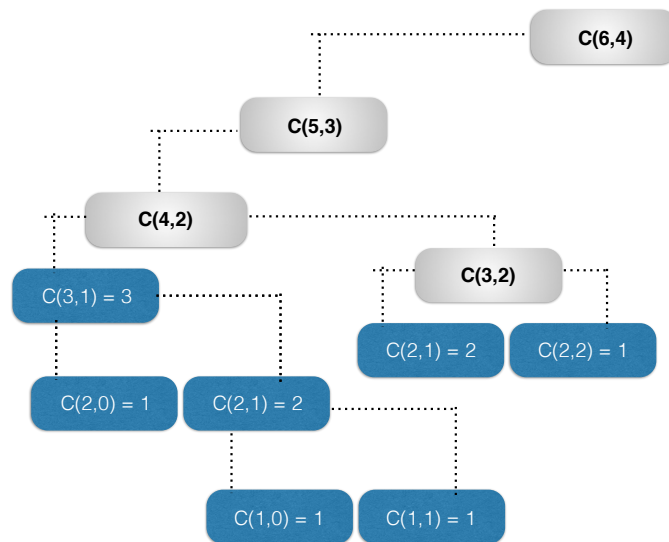
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



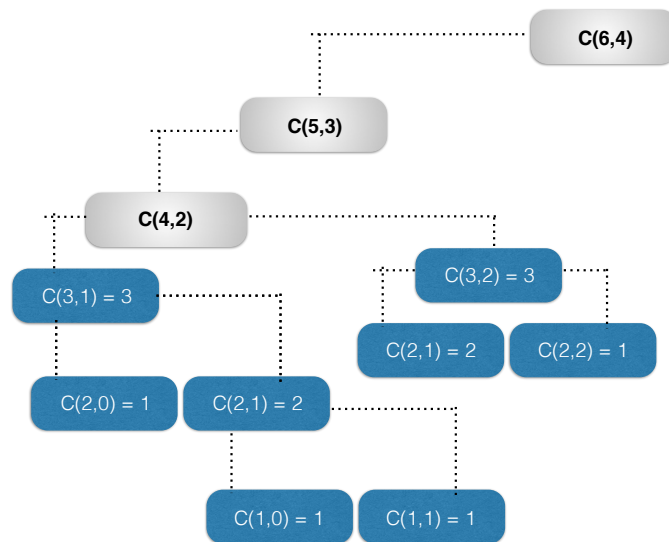
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



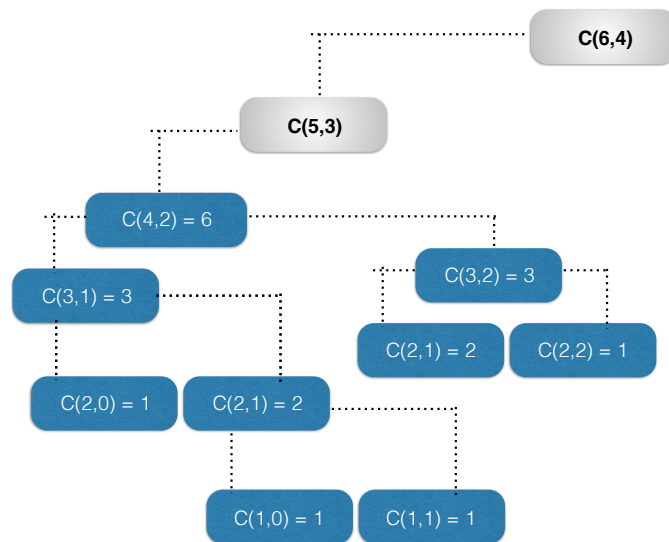
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



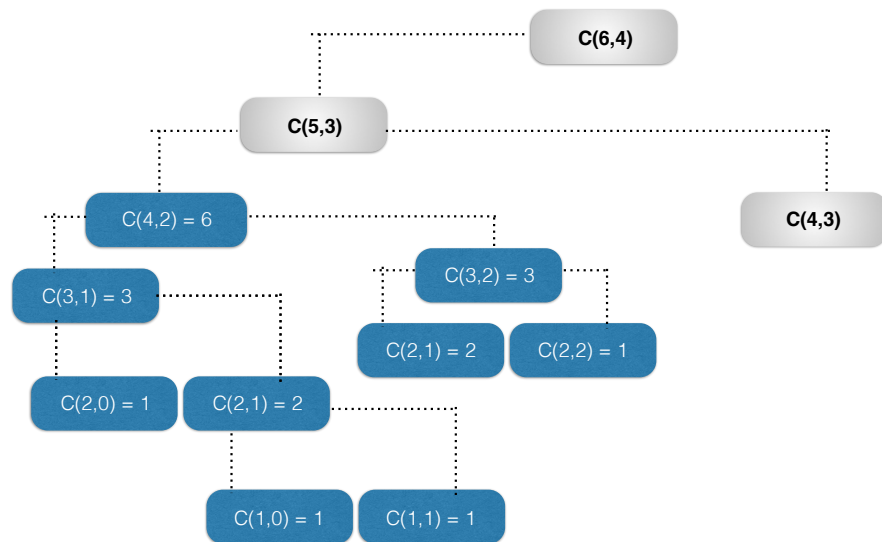
	0	1	2	3	4
0					
1					
2		2			
3		3			
4					
5					
6					



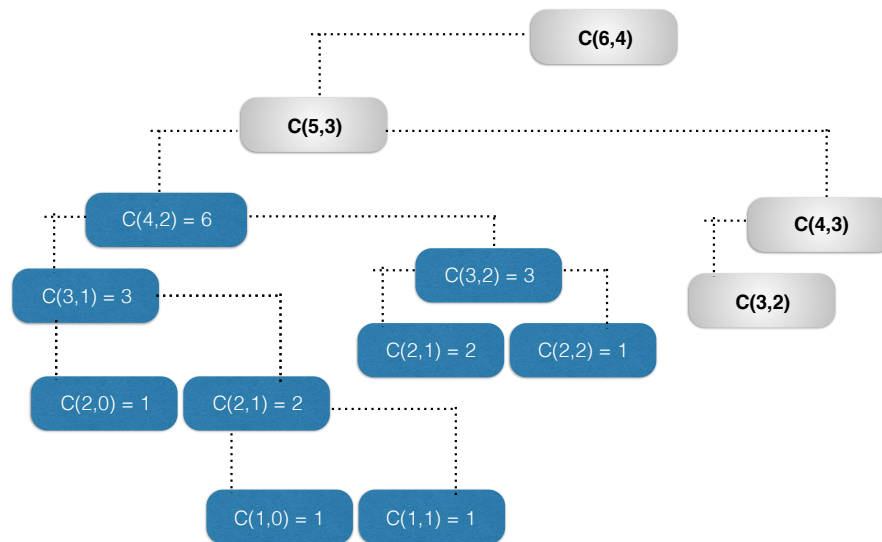
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4					
5					
6					



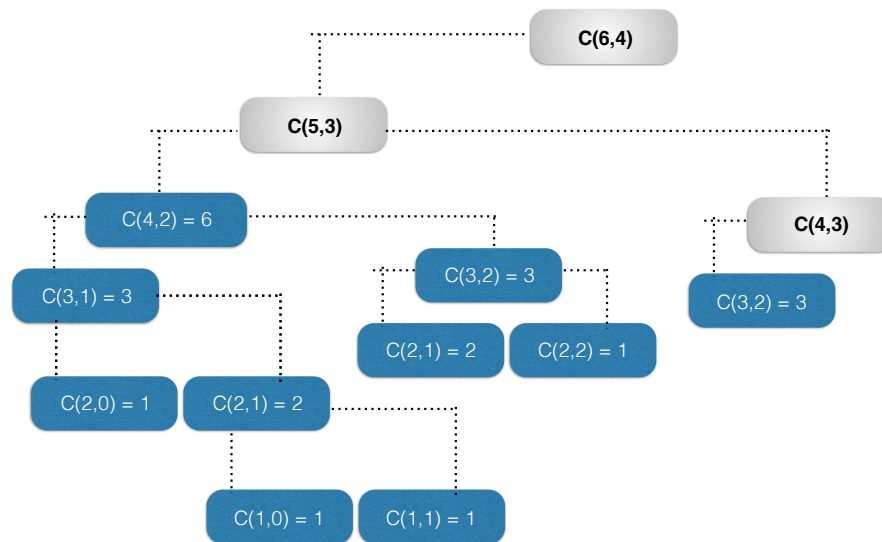
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



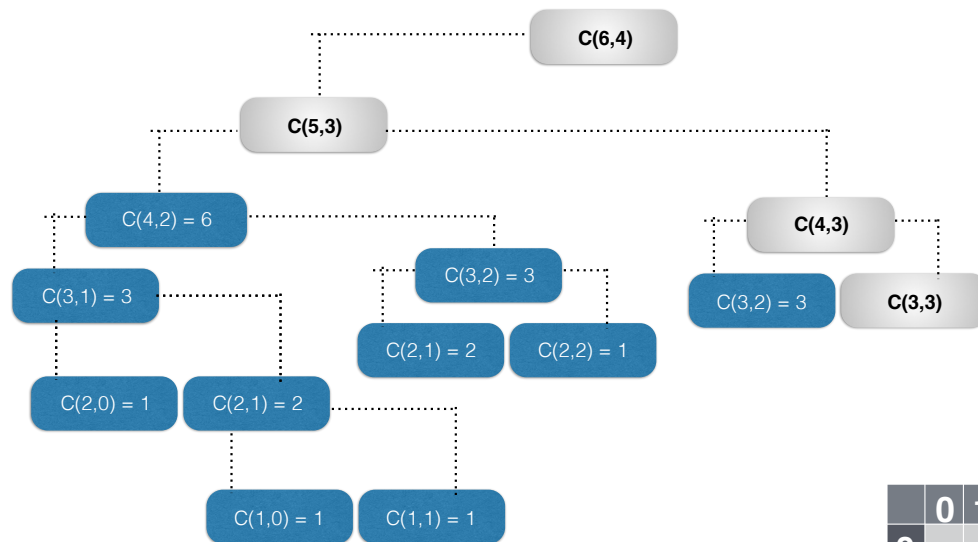
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



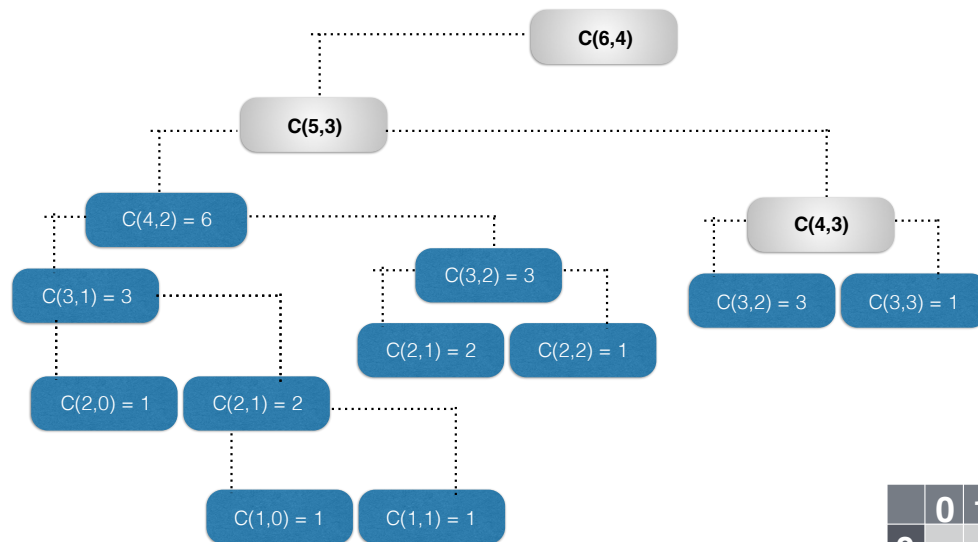
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



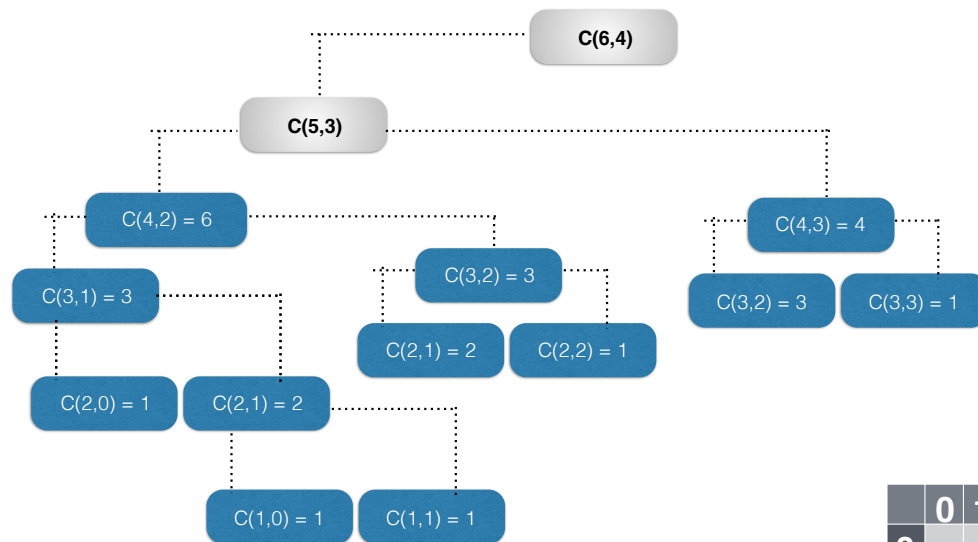
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



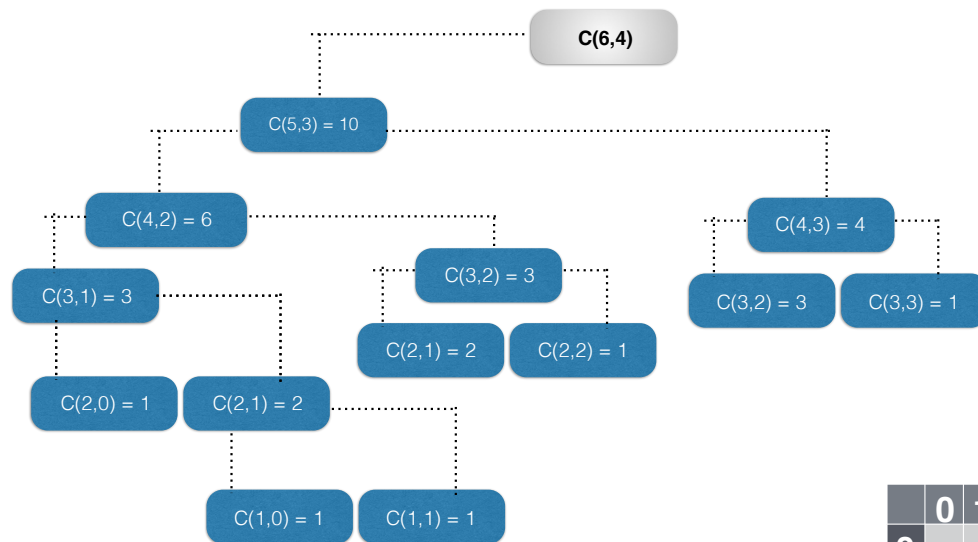
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



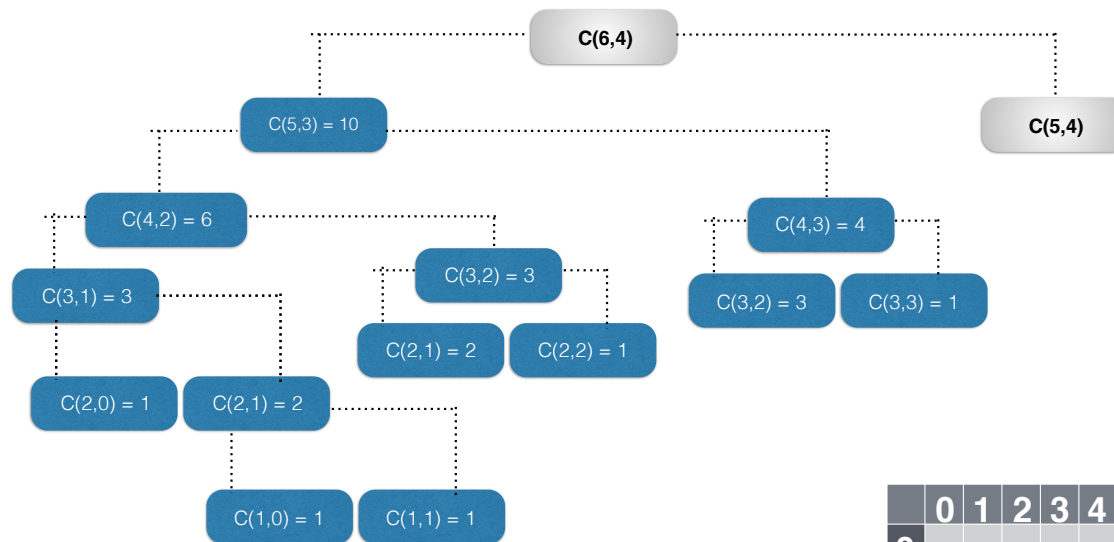
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6		
5					
6					



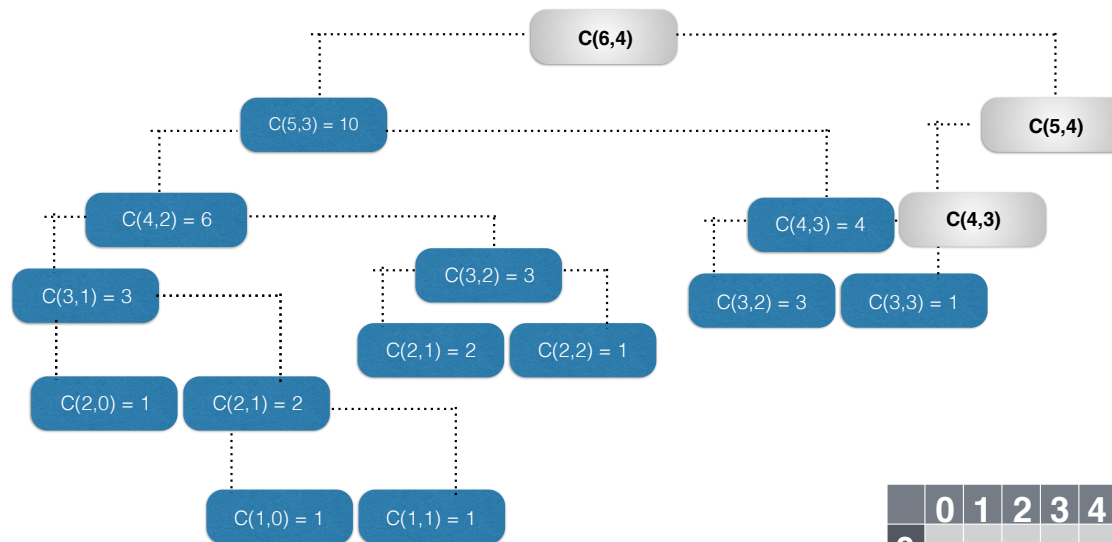
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5					
6					



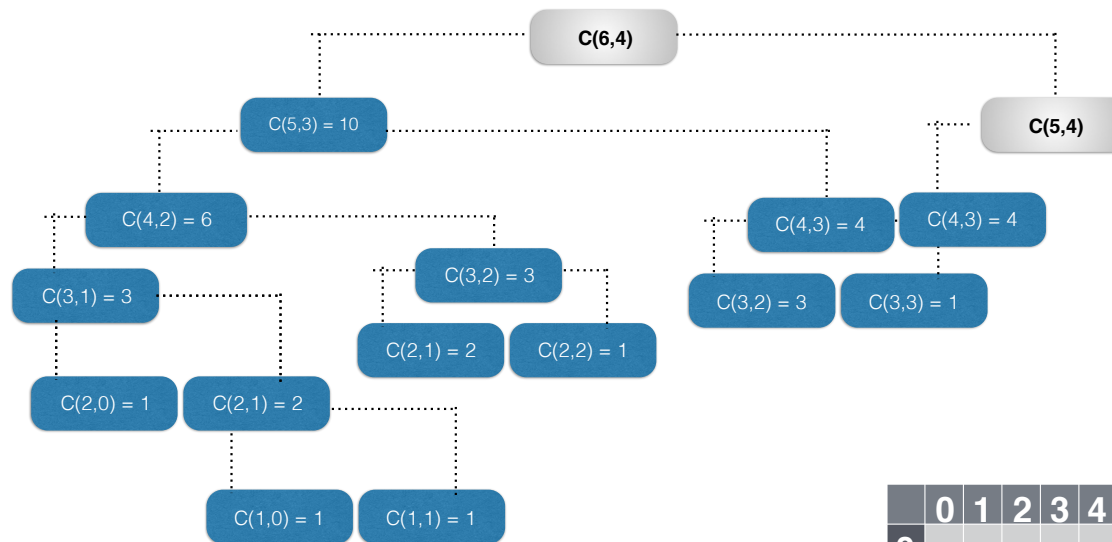
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



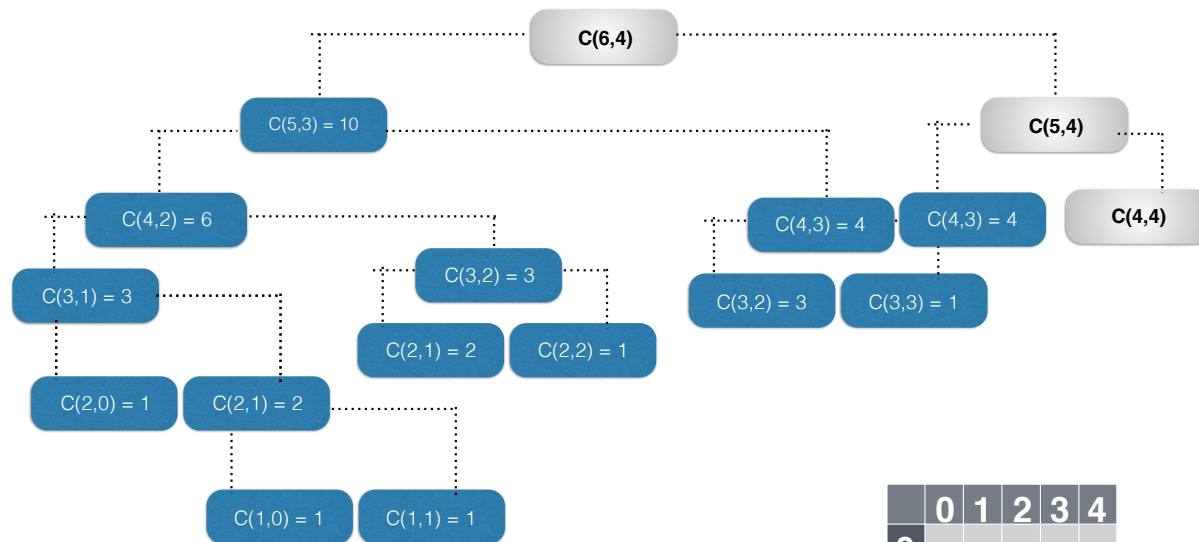
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



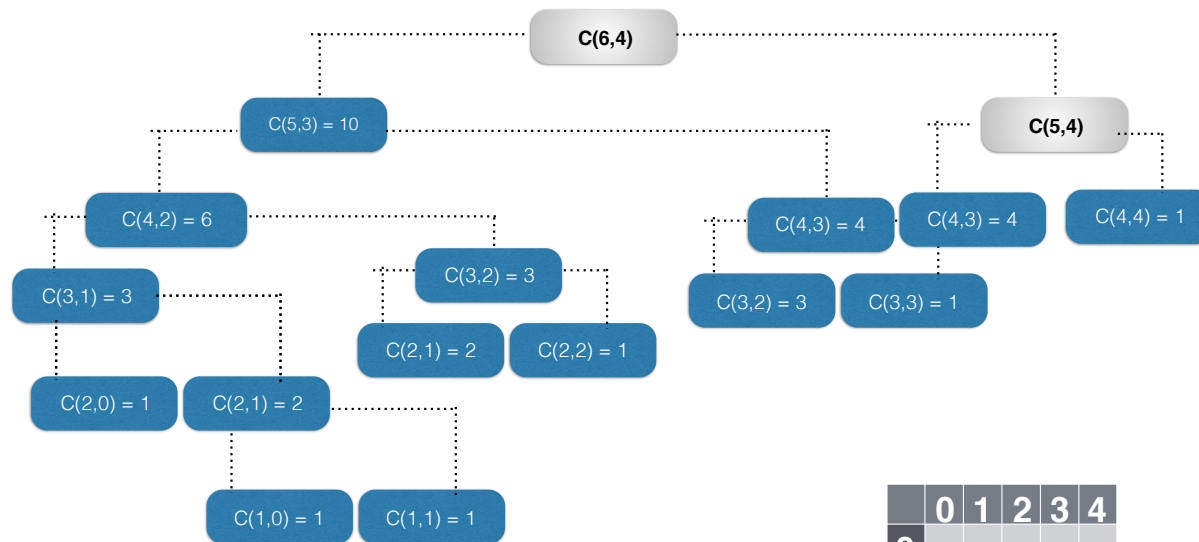
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



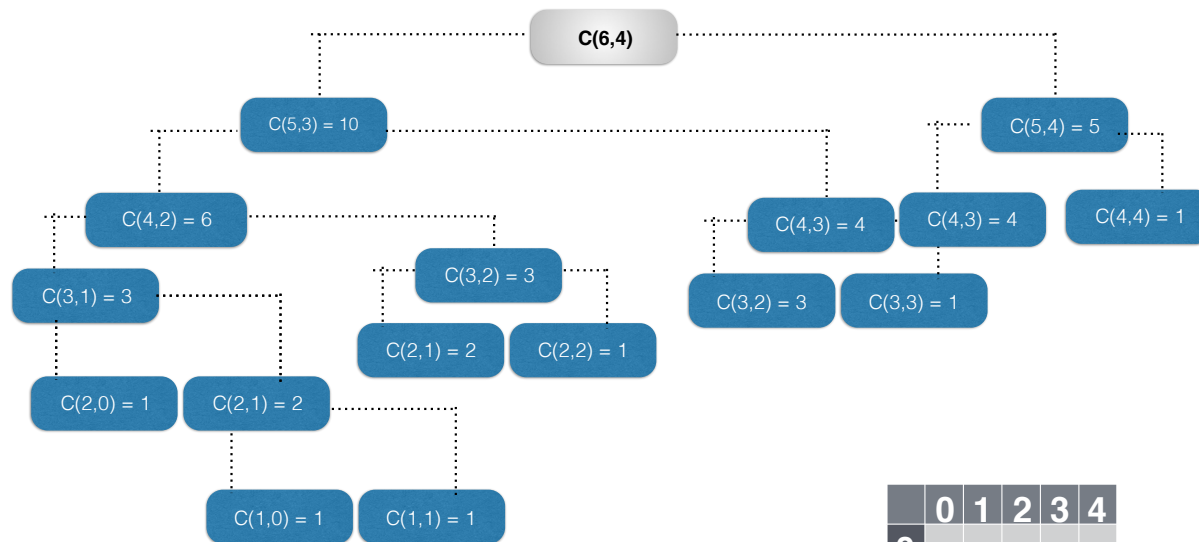
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



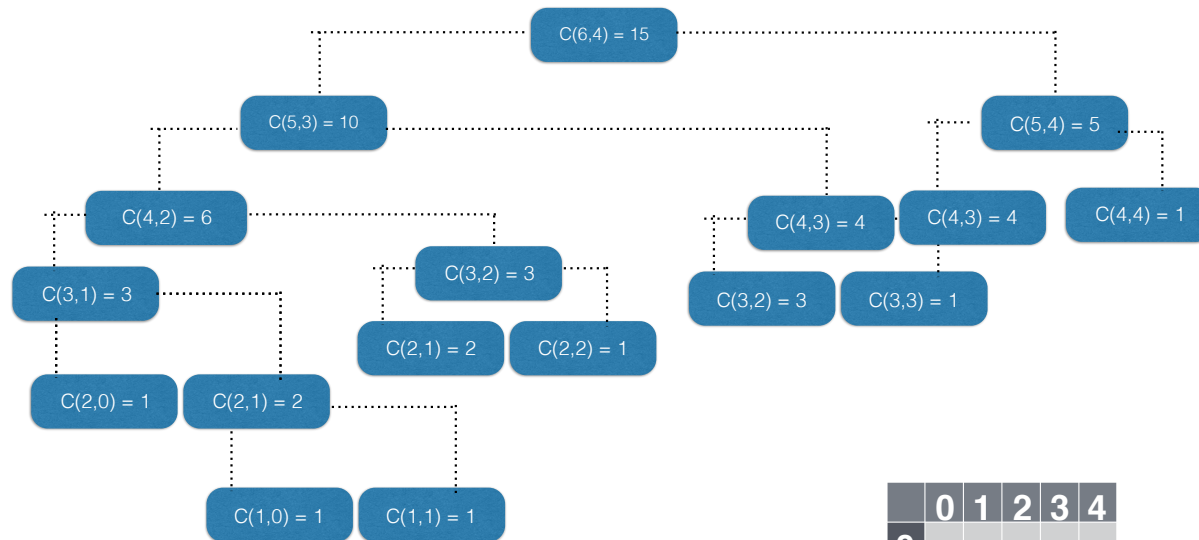
	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	
6					



	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	5
6					



	0	1	2	3	4
0					
1					
2		2			
3		3	3		
4			6	4	
5				10	5
6					15

Java

```
public static int binomialCoefficient(int n, int k, int[][]  
mem){  
    if(k == 0 || n == k) return 1;  
    if(mem[n][k] != 0){  
        return mem[n][k];  
    }  
    int res = binomialCoefficient(n-1,k-1,mem)+  
binomialCoefficient(n-1,k,mem);  
    mem[n][k]=res;  
    return res;  
}
```

Python

```
def binomial_coefficient_memoization(n, k, mem):  
    if n == k or k == 0: return 1  
    if mem[n][k] != 0: return mem[n][k]  
    res = binomial_coefficient_memoization(n - 1, k - 1, mem) +  
binomial_coefficient_memoization(n - 1, k, mem)  
    mem[n][k] = res  
    return res
```

DP Table

		→ K				
		0	1	2	3	4
↓ N	0	1	0	0	0	0
	1	1	1	0	0	0
	2	1	0	1	0	0
	3	1	0	0	1	0
	4	1	0	0	0	1
	5	1	0	0	0	0
	6	1	0	0	0	0

$N = 6$

$K = 4$

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

Initialize

$dp[i][i] = 1, i = 0 \dots k$

$dp[i][0] = 1$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 0 = 1$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	0	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 1 = 2$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 0 = 1$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	0	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	0	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 2 = 3$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$2 + 1 = 3$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 0 = 1$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 0 = 0$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 3 = 4$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	0	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$3 + 3 = 6$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$3 + 1 = 4$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	0	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$0 + 1 = 1$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	0	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 4 = 5$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	0	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$4 + 6 = 10$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	0
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$6 + 4 = 10$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	0	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$4 + 1 = 5$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	0	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$1 + 5 = 6$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	0	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$5 + 10 = 15$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	0

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$10 + 10 = 20$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$10 + 5 = 15$$

DP Table

	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$$

$$10 + 5 = 15$$

$$dp[6][4] = 15$$

Java

```
public static int binomialCoefficientDP(int n, int k){
    int[][] dp = new int[n+1][k+1];
    for(int i=0;i<=n;i++){
        dp[i][0]=1;
        if(i<=k){
            dp[i][i]=1;
        }
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=k;j++){
            dp[i][j] = dp[i-1][j-1]+dp[i-1][j];
        }
    }
    return dp[n][k];
}
```

Python

```
def binomial_coefficient_dp(n, k):  
    dp = [[0 for i in range(0, k + 1)] for i in range(0, n + 1)]  
    for i in range(0, n + 1):  
        dp[i][0] = 1  
        if i <= k:  
            dp[i][i] = 1  
    for i in range(1, n + 1):  
        for j in range(1, k + 1):  
            dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]  
    return dp[n][k]
```