Algorithm Design-2

(Dynamic Programming-Introduction)

Presented
By

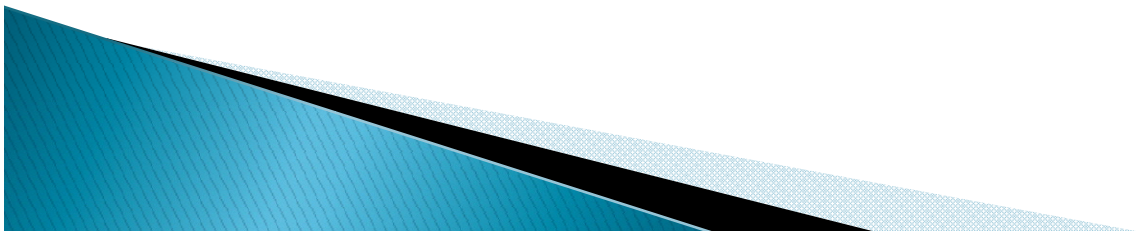
Dr. Rajesh Purkait
Asst. Professor
Dept. of CSE ITER
(S`O'A Deemed To Be University)
E-mail: rajeshpurkait@soa.ac.in

Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Computing Fibonacci Numbers

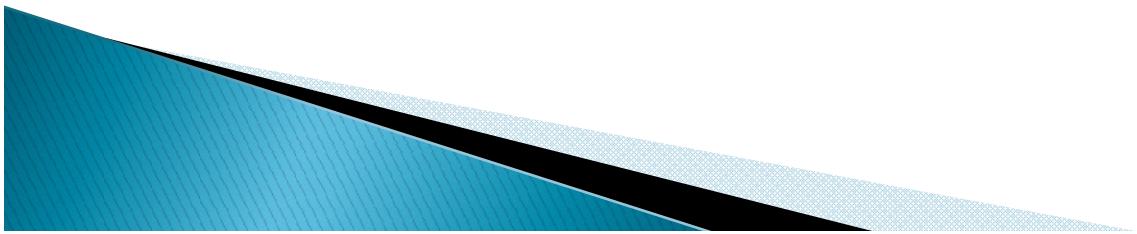
- **Fibonacci numbers:**
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- Sequence is 0, 1, 1, 2, 3, 5, 8, 13, …



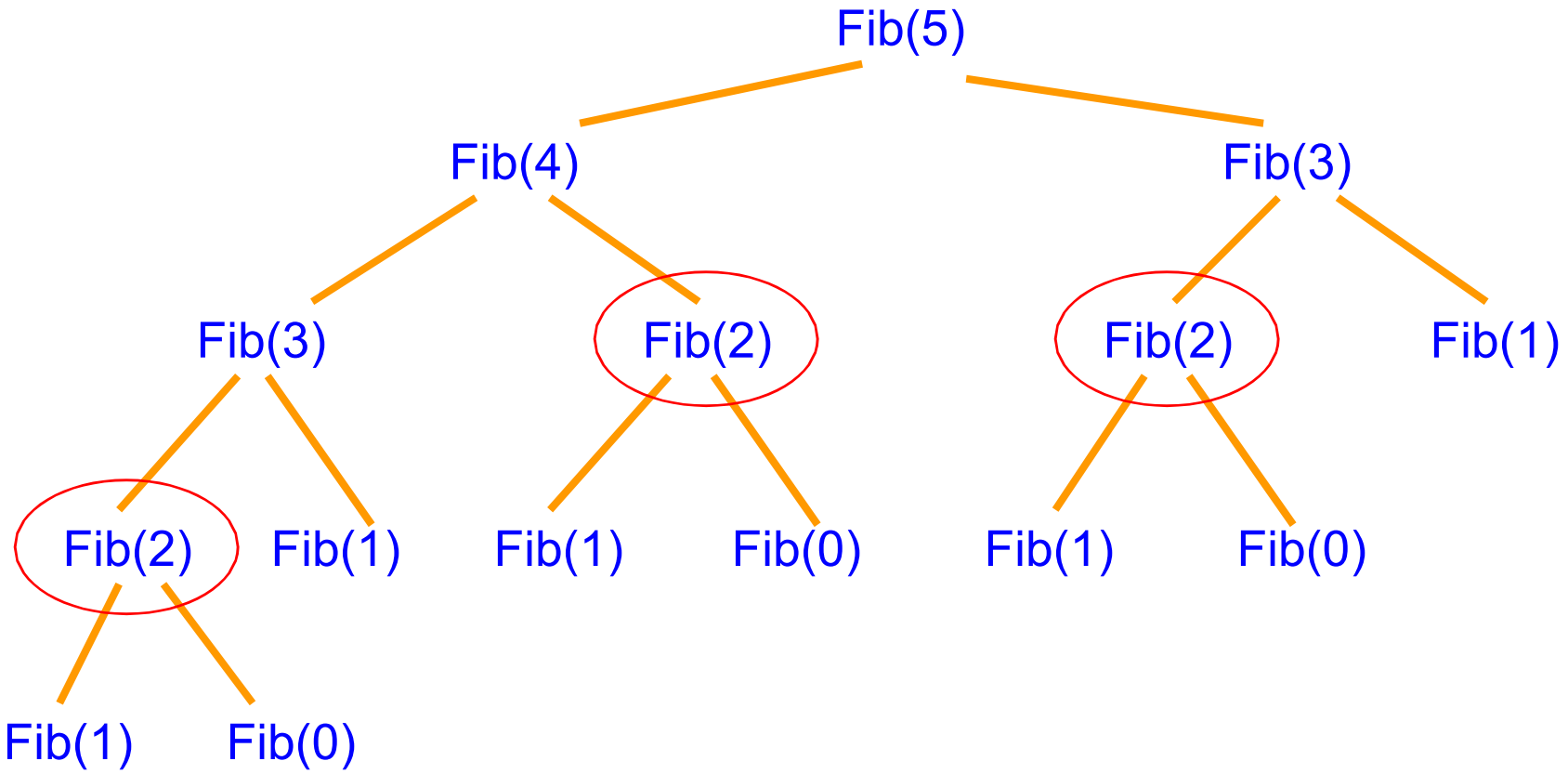
Computing Fibonacci Numbers

- Obvious recursive algorithm:

- Fib(n):
  - if $n = 0$ or 1 then return n
  - else return (F(n–1) + Fib(n–2))

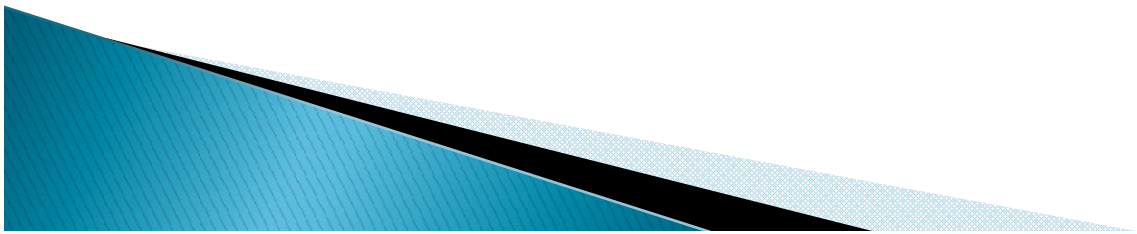


Recursion Tree for Fib(5)



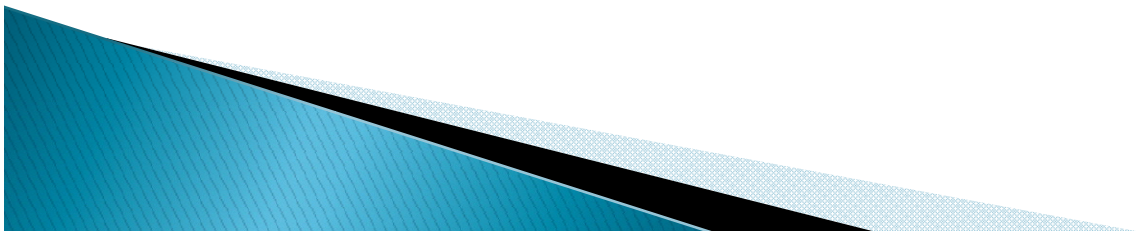
How Many Recursive Calls?

- If all leaves had the same depth, then there would be about 2^n recursive calls.
- But this is over-counting.
- However with more careful counting it can be shown that it is $\Omega((1.6)^n)$
- Exponential!



Save Work

- Wasteful approach - repeat work unnecessarily
  - Fib(2) is computed three times
- Instead, compute Fib(2) once, store result in a table, and access it when needed

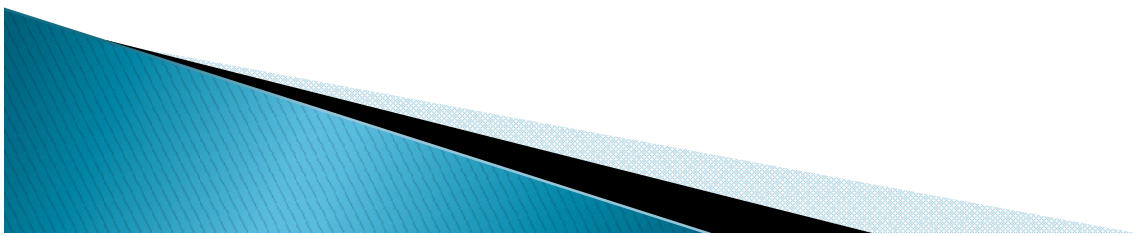


More Efficient Recursive Alg

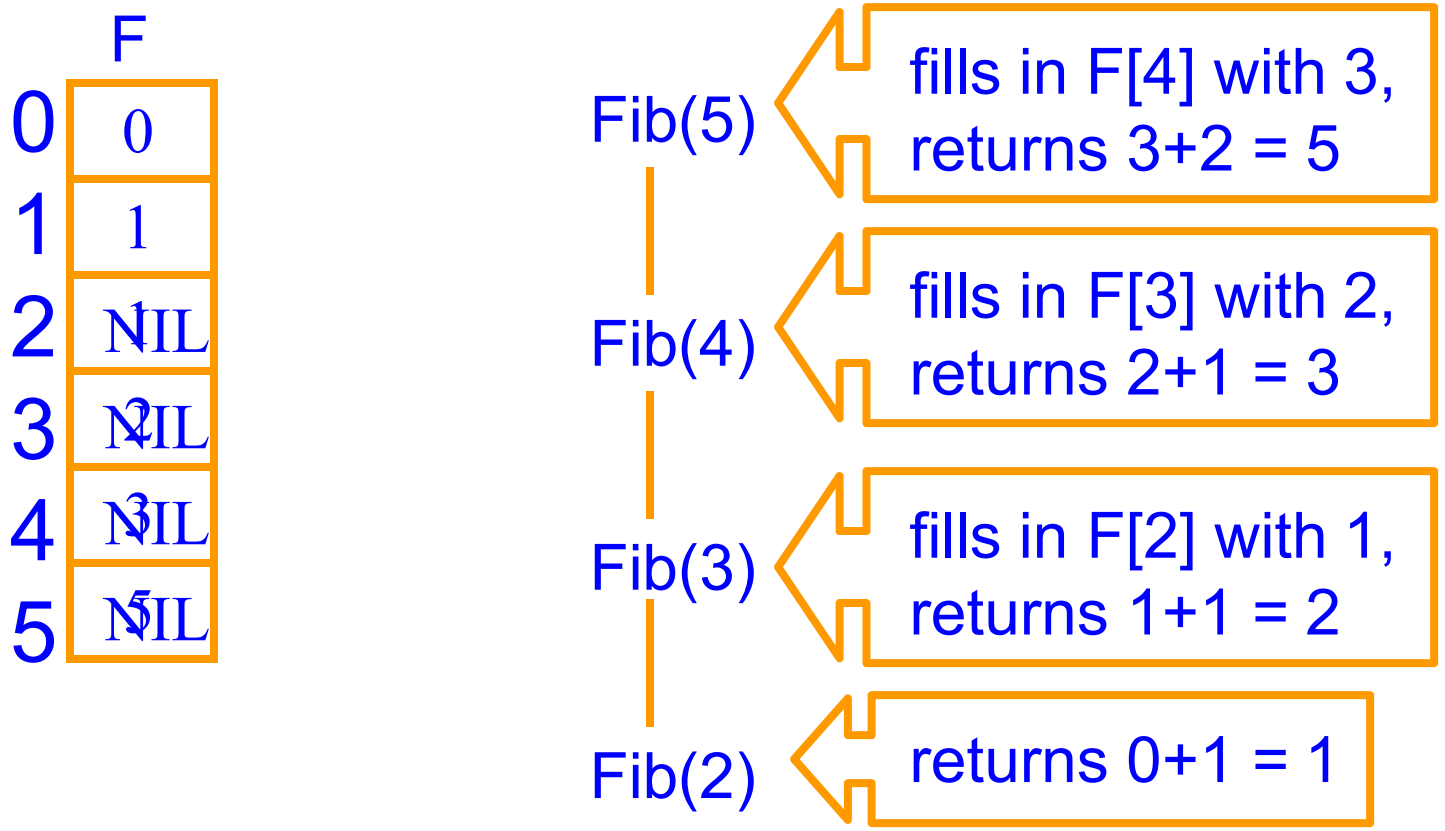
- F[0] := 0; F[1] := 1; F[n] := Fib(n);
- Fib(n):
  - if n = 0 or 1 then return F[n]
  - if F[n-1] = NIL then F[n-1] := Fib(n-1)
  - if F[n-2] = NIL then F[n-2] := Fib(n-2)
  - return (F[n-1] + F[n-2])

Called memorization

- computes each F[i] only once

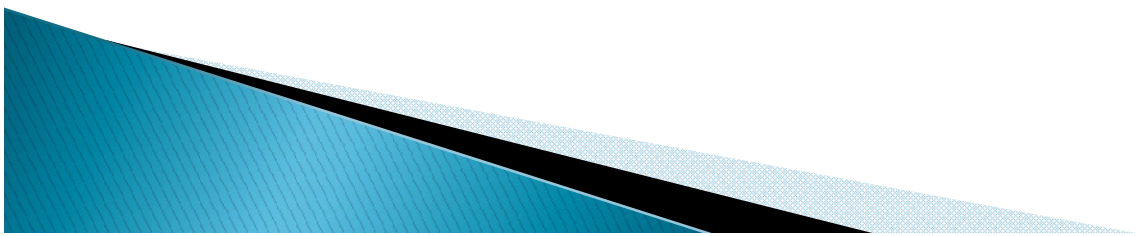


Example of Memoized Fib



Get Rid of the Recursion

- Recursion adds overhead
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- Avoid the recursion overhead by filling in the table entries bottom up, instead of top down.



Subproblem Dependencies

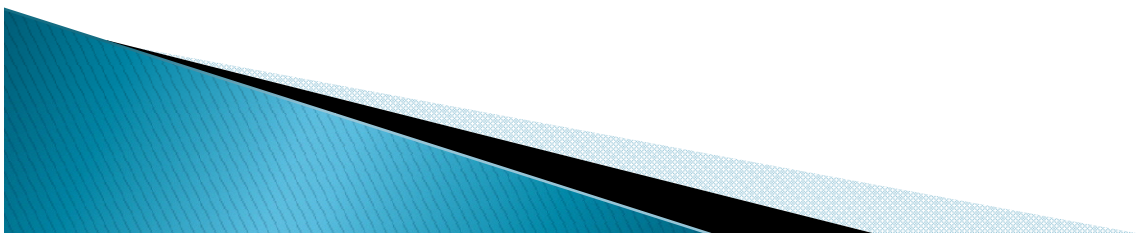
- Figure out which subproblems rely on which other subproblems
- Example:

Order for Computing Subproblems

- Then figure out an order for computing the subproblems that respects the dependencies:
  - when you are solving a subproblem, you have already solved all the subproblems on which it depends
- Example: Just solve them in the order

  $F_0, F_1, F_2, F_3, \ldots$

Called Dynamic Programming



DP Solution for Fibonacci

- Fib(*n*):
  - F[0] := 0; F[1] := 1;
  - for i := 2 to *n* do
    - F[i] := F[i-1] + F[i-2]
  - return F[*n*]
- Can perform application-specific optimizations
  - e.g., save space by only keeping last two numbers computed

Time reduced from exponential to linear

Problem 2: Binomial Coefficients

- $(x + y)^2 = x^2 + 2xy + y^2$, coefficients are 1,2,1
- $(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, coefficients are 1,3,3,1
- $(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, coefficients are 1,4,6,4,1
- $(x + y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$, coefficients are 1,5,10,10,5,1
- The n+1 coefficients can be computed for $(x + y)^n$ according to the formula $c(n, i) = n! / (i! * (n - i)!)$ for each of $i = 0..n$
- The repeated computation of all the factorials gets to be expensive
- We can use dynamic programming to save the factorials as we go

Binomial Coefficients

- Applicable when subproblems are **not** independent
  - Subproblems share subsubproblems

E.g.: Combinations:

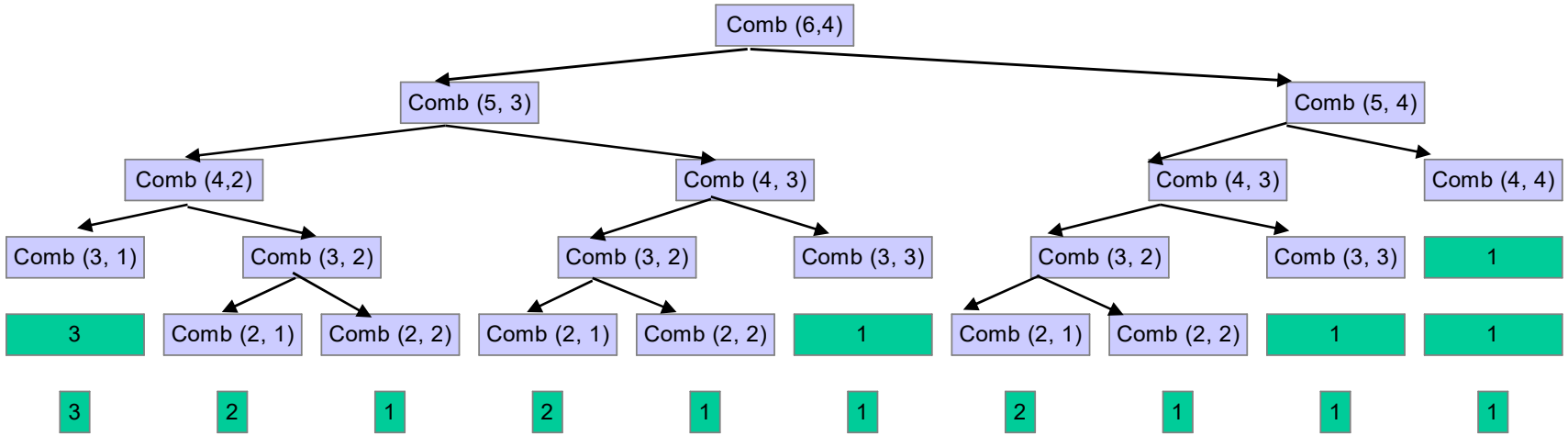
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n}{1} = 1 \qquad \binom{n}{n} = 1$$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Example: Combinations

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n. \end{cases}$$



$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Solution by dynamic programming

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

- Each row depends only on the preceding row
- Only linear space and quadratic time are needed
- This algorithm is known as Pascal's Triangle

The algorithm of Binomial Coefficients

```
long binomial_coefficient(n,m)
int n,m;                            /* computer n choose m */
{
        int i,j;                    /* counters */
        long bc[MAXN][MAXN];        /* table of binomial coefficients */

        for (i=0; i<=n; i++) bc[i][0] = 1;

        for (j=0; j<=n; j++) bc[j][j] = 1;

        for (i=1; i<=n; i++)
                for (j=1; j<i; j++)
                        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

        return( bc[n][m] );
}
```

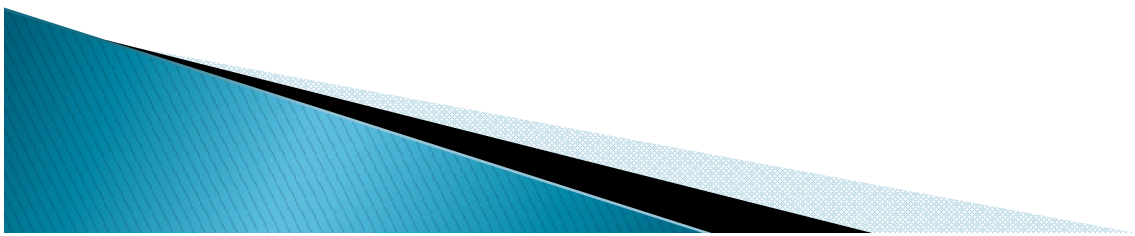
The algorithm in Java

```
public static int binom(int n, int m) {
    int[ ] b = new int[n + 1];
    b[0] = 1;
    for (int i = 1; i <= n; i++) {
        b[i] = 1;
        for (int j = i – 1; j > 0; j--) {
            b[j] += b[j – 1];
        }
    }
    return b[m];
}
```

- **Source:** Data Structures and Algorithms with Object-Oriented Design Patterns in Java *by* Bruno R. Preiss

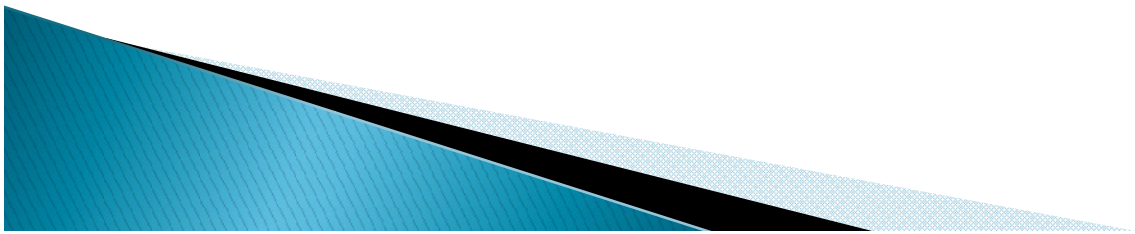
Dynamic Programming (DP) Paradigm

- DP is typically applied to Optimization problems.
- DP can be applied when a problem exhibits:
- Optimal substructure:
  - Is an optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems:
  - If recursive algorithm revisits the same problem over and over again.



Dynamic Programming (DP) Paradigm

- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem



Thank You