Amirkabir University of Technology
(Tehran Polytechnic)

Department of
Computer Engineering

Data Structure & Algorithms

Dynamic Programming

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

Dynamic Programming

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

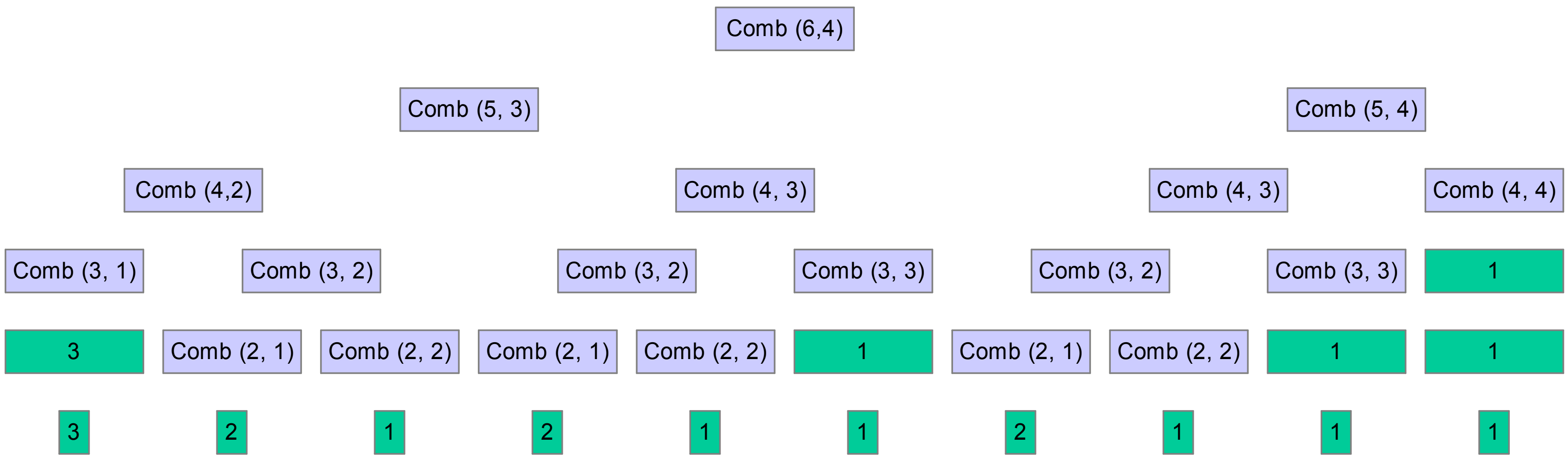
E.g.: Combinations:

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

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

- 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



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

Dynamic Programming

- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

1) Characterize the optimal solution of the problem

- Really understand the problem
- Verify if an algorithm that verifies all solutions (brute force) is not enough
- Try to generalize the problem (it takes practice to understand how to correctly generalize)
- Try to divide the problem in subproblems of the same type Verify if the problem obeys the optimality principle
- Verify if there are overlapping subproblems

2) Recursively define the optimal solution, by using optimal solutions of subproblems

- Recursively define the optimal solution value, exactly and with rigour, from the solutions of subproblems of the same type

- Imagine that the values of optimal solutions are already available when we need them

- No need to code. You can just mathematically define the recursion

3) Compute the solutions of all subproblems: bottom-up

- Find the order in which the subproblems are needed, from the smaller subproblem until we reach the global problem and implement, using a table

- Usually this order is the inverse to the normal order of the recursive function that solves the problem

```
             Normal solving order
+---------+   -------------->   +-----+
|Beginning|                     | End |
+---------+   <--------------   +-----+
             Order when using DP
```

3) Compute the solutions of all subproblems: bottom-up example

- Start computing result for the subproblem. Using the subproblem result solve another subproblem and finally solve the whole problem.
- Let's find the nth member of a Fibonacci series.

  Fibonacci(0) = 0

  Fibonacci(1) = 1

  Fibonacci(2) = 1 (Fibonacci(0) + Fibonacci(1))

  Fibonacci(3) = 2 (Fibonacci(1) + Fibonacci(2))

- We can solve the problem step by step.

  1. Find Oth member
  2. Find 1st member
  3. Calculate the 2nd member using 0th and 1st member
  4. Calculate the 3rd member using 1st and 2nd member
  5. By doing this we can easily find the nth member.

3) Compute the solutions of all subproblems: bottom-up example

Algorithm:

1. set Fib[0] = 0

2. set Fib[1] = 1

3. From index 2 to n compute result using the below formula

    Fib[index] = Fib[index - 1] + Fib[index - 2]

4. The final result will be stored in Fib[n].

3) Compute the solutions of all subproblems: bottom-up example

Code:

```c
int Fibonacci(int N)
{
    //if N = 2, we need to store 3 fibonacci members(0,1,1)
    //if N = 3, we need to store 4 fibonacci members(0,1,1,2)
    //In general to compute Fib(N), we need N+1 size array.
    int Fib[N+1],i;

    //we know Fib[0] = 0, Fib[1]=1
    Fib[0] = 0;
    Fib[1] = 1;

    for(i = 2; i <= N; i++)
        Fib[i] = Fib[i-1]+Fib[i-2];

    //last index will have the result
    return Fib[N];
}

int main()
{
    int n;
    scanf("%d",&n);

    //if n == 0 or n == 1 the result is n
    if(n <= 1)
        printf("Fib(%d) = %d\n",n,n);
    else
        printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;
}
```

3) Compute the solutions of all subproblems: top-down

- There is a technique, known as ”memoization”, that allows us to solve the problem by the normal order.

- Just use the recursive function directly obtained from the definition of the solution and keep a table with the results already computed.

- When we need to access a value for the first time we need to compute it, and from then on we just need to see the already computed result.

3) Compute the solutions of all subproblems: top-down example

- Let's solve the same Fibonacci problem using the top-down approach.

Top-Down starts breaking the problem unlike bottom-up.

Like, If we want to compute Fibonacci(4), the top-down approach will do the following

Fibonacci(4) -> Go and compute Fibonacci(3) and Fibonacci(2) and return the results.

Fibonacci(3) -> Go and compute Fibonacci(2) and Fibonacci(1) and return the results.

Fibonacci(2) -> Go and compute Fibonacci(1) and Fibonacci(0) and return the results.

Finally, Fibonacci(1) will return 1 and Fibonacci(0) will return 0.

```
                 Fib(4)
                /    \
           Fib(3)    Fib(2)
           /    \     /   \
      Fib(2)  Fib(1) Fib(1)  Fib(0)
      /    \
 Fib(1) Fib(0)
```

3) Compute the solutions of all subproblems: top-down example

Algorithm:

Fib(n)

If n == 0 || n == 1 return n;

Otherwise, compute subproblem results recursively.

return Fib(n-1) + Fib(n-2);

3) Compute the solutions of all subproblems: top-down example

Code:

```
#include<stdio.h>

int Fibonacci(int N)
{
    if(N <= 1)
        return N;
    return Fibonacci(N-1) + Fibonacci(N-2);
}

int main()
{
    int n;
    scanf("%d",&n);
    printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;
}
```

4) Reconstruct the optimal solution, based on the computed values

- It may (or may not) be needed, given what the problem asks for

- Two alternatives:

  Directly from the subproblems table

  New table that stores the decisions in each step

- If we do not need to know the solution in itself, we can eventually save some space