

Advanced Computer Contest Preparation  
Lecture 13

# DYNAMIC PROGRAMMING

# Sample Problem:

## Mouse Journey (CCC '12 S5)

- ➊ You are a mouse in a  $R \times C$  grid of cages
  - ➋ For this version:  $1 \leq R, C \leq 1,000$
- ➋ You start at the top-left corner  $(1,1)$
- ➌ You may move to the cage directly right or below
- ➍ There are cages that contain cats; you cannot enter these cages
- ➎ How many paths are there from  $(1,1)$  to  $(R,C)$ ?

# Mouse Journey (CCC '12 S5)



# Solutions?

- Brute force?
  - Generate all possible paths?
  - What is the runtime of this algorithm?
  - Between  $O(2^N)$  and  $O(4^N)$
- Greedy?
  - Not an optimization problem
  - How can we easily compare two squares?
  - We cannot use greedy

# Dynamic Programming

- ➊ *Dynamic programming (DP)* is a problem solving technique
- ➋ DP can solve optimization problems
  - The “best” answer
  - For example, maximum/minimum answer
- ➌ DP can solve counting problems
  - “What are the number of ways...”
- ➍ Correct DP *should always* compute the correct answer

# Subproblems

- Suppose we are given a problem
- A *subproblem* is another instance of the problem, but with smaller constraints
- Example: what is the  $N^{th}$  Fibonacci number?
- The problem is: find  $fib(N)$
- The subproblems are: find  $fib(i)$  for  $i = 1$  to  $N$

# Optimal Substructure

- *Optimal substructure* is the property where a problem's solution depends on its subproblems' solutions
- The relationship between a problem's solution and its subproblems' solutions must be found
- Eventually, subproblems can be easily solved
  - These subproblems are the *base cases*

# Optimal Substructure

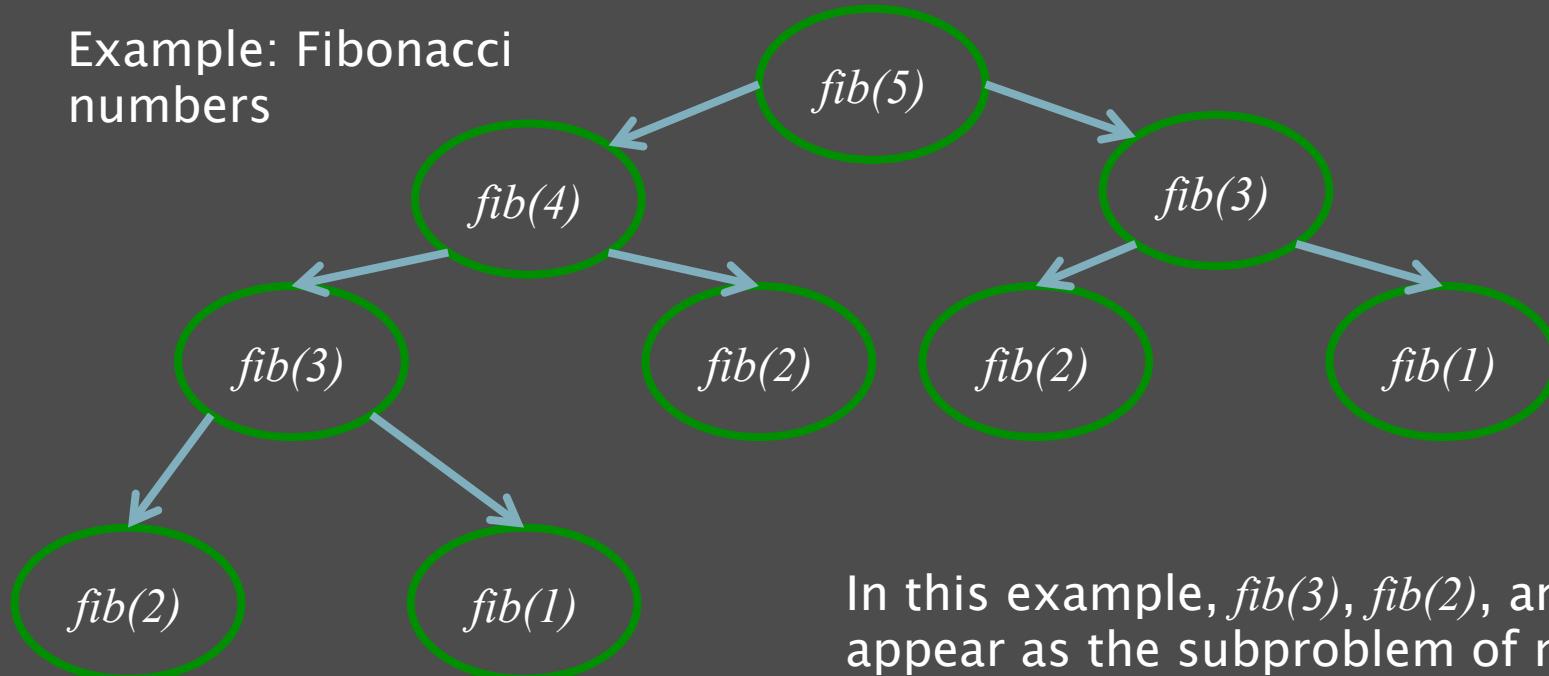
- Example: Fibonacci numbers
- The dependent subproblems of  $\text{fib}(i)$  are  $\text{fib}(i-1)$  and  $\text{fib}(i-2)$
- The exact relationship between  $\text{fib}(i)$  and its subproblems is  $\text{fib}(i) = \text{fib}(i-1) + \text{fib}(i-2)$
- The base cases are  $\text{fib}(1) = 1, \text{fib}(2) = 1$

# Overlapping Subproblems

- *Overlapping subproblems* occur when the same subproblem might occur more than once
- In other words, more than one problem might depend on the same subproblem to compute their solutions
- If we form a graph of the subproblem dependencies, we will form a DAG

# Overlapping Subproblems

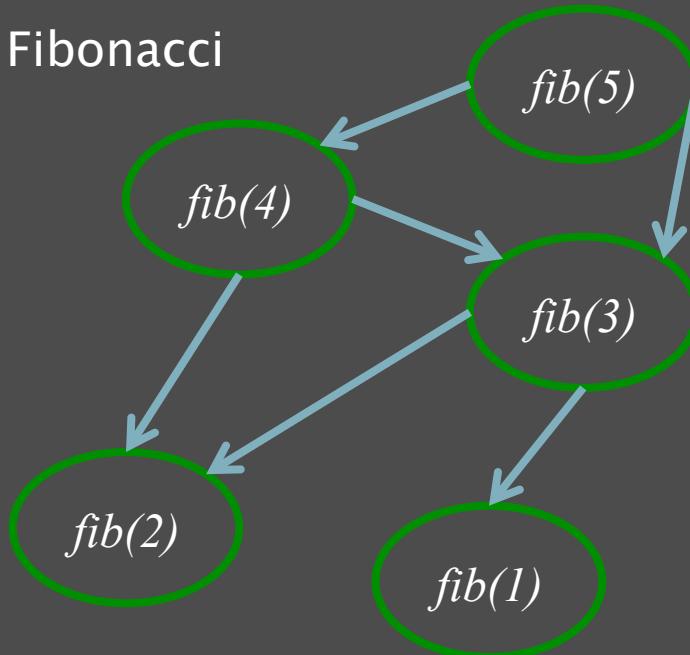
Example: Fibonacci numbers



In this example,  $fib(3)$ ,  $fib(2)$ , and  $fib(1)$  appear as the subproblem of more than one problem. Therefore, the subproblems overlap.

# Overlapping Subproblems

Example: Fibonacci numbers



2 problems directly depend on  $fib(3)$ :  
 $fib(5)$  and  $fib(4)$

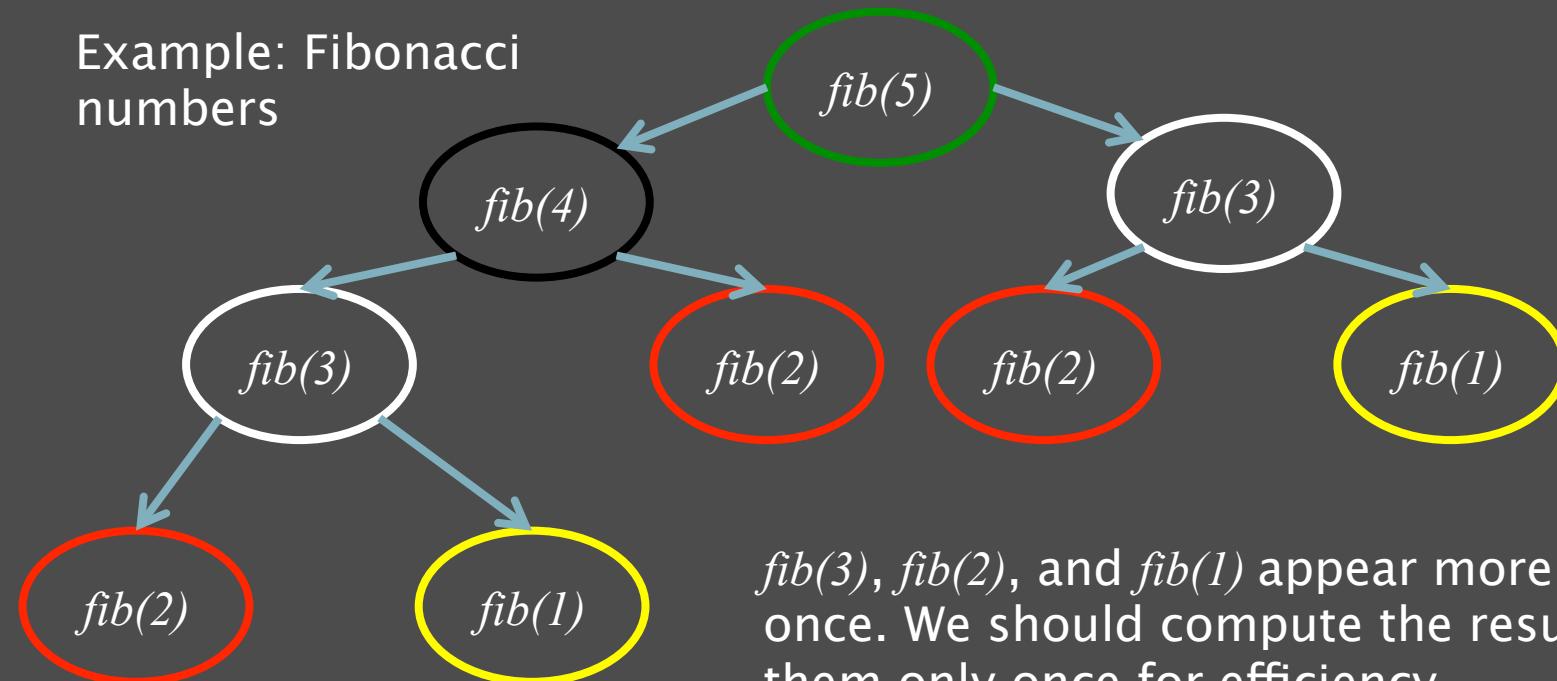
Note that the graph of the problem dependencies is a DAG

# Memoization

- ◎ *Memoization* is the storing of computed solutions of subproblems
- ◎ The solution of each subproblem is computed exactly once and stored in some data structure
- ◎ When the solution for the subproblem is needed, we look it up in the data structure instead of re-computing it
- ◎ Also known as *tabling*, data structure is known as *lookup table*

# Memoization

Example: Fibonacci numbers



$\text{fib}(3)$ ,  $\text{fib}(2)$ , and  $\text{fib}(1)$  appear more than once. We should compute the result of them only once for efficiency.

# DP State

- The most common data structure used to store solutions to subproblems is the array
- The constraints of a subproblem are collectively called the *DP state*
- The array must be able to quickly locate an answer given the state
- Normally, we will use a  $N$ -dimensional array, one dimension for each variable
- The size of the  $i^{th}$  dimension is the number of possible values of the  $i^{th}$  variable

# DP State

- ◎ Example 1: Fibonacci numbers
  - We only have 1 variable,  $N$ , which corresponds to the  $N^{th}$  Fibonacci number
  - $1 \leq N \leq 1,000$
  - The DP array is `dp[1005]`
- ◎ Example 2:
  - We have 3 variables:  $x$ ,  $y$ , and  $z$
  - $1 \leq x \leq 1,000$ ,  $1 \leq y \leq 100$ ,  $z$  is a `bool`
  - The DP array is `dp[1005][105][2]`

# Dynamic Programming - Summary

- In order to use DP, the problem must have overlapping subproblems and the optimal substructure property
  - Subproblems occur more than once
  - Problem's solution depends on its subproblems' solutions
- Relationship between problem's solution and its subproblems' solutions must be found
  - Most challenging step!
- Solution to each subproblem is stored in a data structure for future reference so that it doesn't need to be computed again

# Two DP Directions

- DP can be done iteratively or recursively
- Iterative DP starts from the base cases and works upward to the final problem
  - Typically, there is a for loop for every variable
  - Order of for loops matters; ensure that all dependent subproblems for any problem are already computed
- Recursive DP starts from the final problem, works towards the base cases, and works back to the final problem

# DP Analysis

- The time complexity of a DP algorithm is generally:
  - The number of required states multiplied by
    - The time complexity to compute the solution for a state
      - This time complexity is at least the number of subproblems that the current state depends on

# DP Analysis Example

- Example 1: Fibonacci numbers
  - Find the  $N^{th}$  Fibonacci number
  - Number of states:  $O(N)$
  - Time complexity to solve one problem:  $O(1)$
  - Final time complexity:  $O(N)$
- Example 2:
  - Number of states:  $O(XYZ)$
  - Time complexity to solve one problem:  $O(X^2Y)$
  - Final time complexity:  $O(X^3Y^2Z)$

# Back to Mouse Journey (CCC '12 S5)

- ➊ You are a mouse in a  $R \times C$  grid of cages
  - ➋ For this version:  $1 \leq R, C \leq 1,000$
- ➋ You start at the top-left corner  $(1,1)$
- ➌ You may move to the cage directly right or below
- ➍ There are cages that contain cats; you cannot enter these cages
- ➎ How many paths are there from  $(1,1)$  to  $(R,C)$ ?

# Solution – Dynamic Programming

- Can DP be used?
  - This problem is a *counting problem*
  - Does the problem have the optimal substructure property?
  - Does the problem have overlapping subproblems?

# Solution – Dynamic Programming

- What is the problem we are trying to solve?
  - What is the number of paths to cell  $(R, C)$ ?
- What are the subproblems?
  - What is the number of paths to cell  $(i, j)$  for  $i = 1$  to  $R$  and for  $j = 1$  to  $C$ ?

# Solution – Dynamic Programming

- For a problem  $p(i,j)$ , what are the dependent subproblems?
  - Note that we can only move right or down
  - Thus, we must have came from the cell above or to the left
  - The solution to  $p(i,j)$  depends on subproblems  $p(i-1,j)$  and  $p(i,j-1)$
- What are the base cases?
  - $p(i,j) = 0$  if  $i = 0$  or  $j = 0$  or if cell  $(i,j)$  contains a cat
  - $p(1,1) = 1$

# Solution – Dynamic Programming

- For a problem  $p(i,j)$ , how is it related to its subproblems?
  - $p(i,j) = p(i-1,j) + p(i,j-1)$
  - The number of paths to cell  $(i,j)$  is equal to the sum of the number of paths to cell  $(i-1,j)$  and cell  $(i,j-1)$

# Solution – Dynamic Programming

- What is the DP state format?
  - There are two variables,  $R$  and  $C$
  - Each of them can be in the range  $[1,1000]$
  - Therefore, the DP array is  $\text{dp}[1005][1005]$

# DP Table - Visualization

|                                                                                        |                                                                                        |   |   |                                                                                          |                                                                                             |   |
|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|---|---|------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|---|
| 1<br> | 1                                                                                      | 1 | 1 | 1                                                                                        | 1                                                                                           | 1 |
| 1                                                                                      | 0<br> | 1 | 2 | 3                                                                                        | 4                                                                                           |   |
| 1                                                                                      | 1                                                                                      | 2 | 4 | 0<br> | 4                                                                                           |   |
| 1                                                                                      | 0<br> | 2 | 6 | 6                                                                                        | 10                                                                                          |   |
| 1                                                                                      | 1                                                                                      | 3 | 9 | 15                                                                                       | 25<br> |   |

# Pseudocode - Iterative

```
bool cat[R+5][C+5]; //true if cat is in cell
int dp[R+5][C+5]; //number of paths, all initially 0

//process input here

dp[1][1] = 1;
for (row 1 to R) {
    for (col 1 to C) {
        if (cat[row][col]) continue;
        dp[row][col] = dp[row-1][col] + dp[row][col-1];
    }
}
print dp[R][C];
```

# Pseudocode – Recursive

```
bool cat[R+5][C+5]; //true if cat is in cell
int dp[R+5][C+5]; //number of paths, all initially 0

//process input here

int solve(int r, int c){
    if (dp[r][c] != 0) return dp[r][c];
    if (r == 0 || c == 0 || cat[r][c]) return 0;
    if (r == 1 && c == 1) dp[r][c] = 1;
    else dp[r][c] = solve(r-1,c) + solve(r,c-1);
    return dp[r][c];
}

print solve(R,C);
```

# Mouse Journey Analysis

- ➊ How many states are there?
  - $O(RC)$
- ➋ How many subproblems does each state depend on?
  - $O(1)$
- ➌ What is the time complexity needed to compute the solution to a problem?
  - $O(1)$
- ➍ Therefore, the final time complexity is  $O(RC)$

# THANK YOU!