CS5800 – ALGORITHMS

MODULE 8. DYNAMIC PROGRAMMING - I

Lesson 1: Fibonacci Series

Ravi Sundaram

Topics

- What is Dynamic Programming
- What is the Fibonacci sequence
- Top-down approach
  - How is it inefficient?
- Bottom-up approach
  - Memoization or table building
- Summary

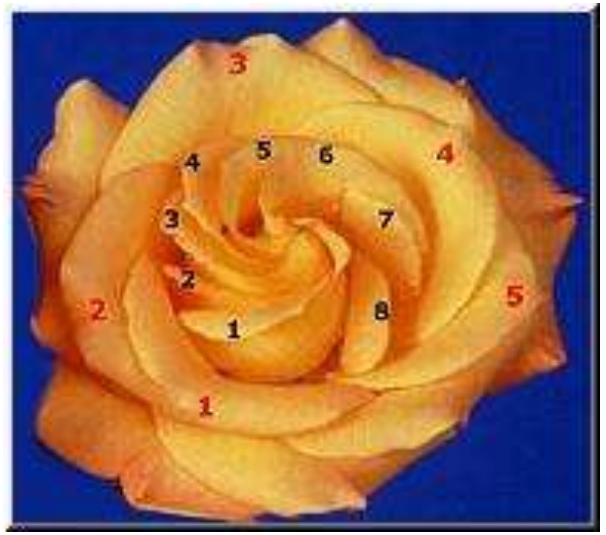
Dynamic Programming

- Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to sub-problems.
- Unlike divide and conquer, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems,

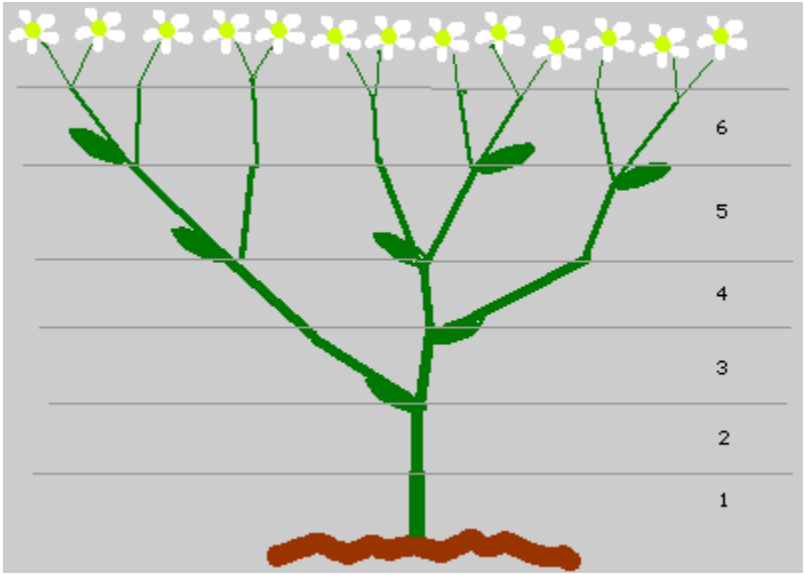
What is the Fibonacci sequence?

- The sequence begins with one. Each subsequent number is the sum of the two preceding numbers.
  Fib(n) = Fib(n-1) + Fib(n-2)
- Thus the sequence begins as follows:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144….
- Introduced to Western Europe in the 1200s by Fibonacci
- Appeared earlier in 200BC in a treatise on Sanskrit poetry by Pingala

Fibonacci numbers in nature



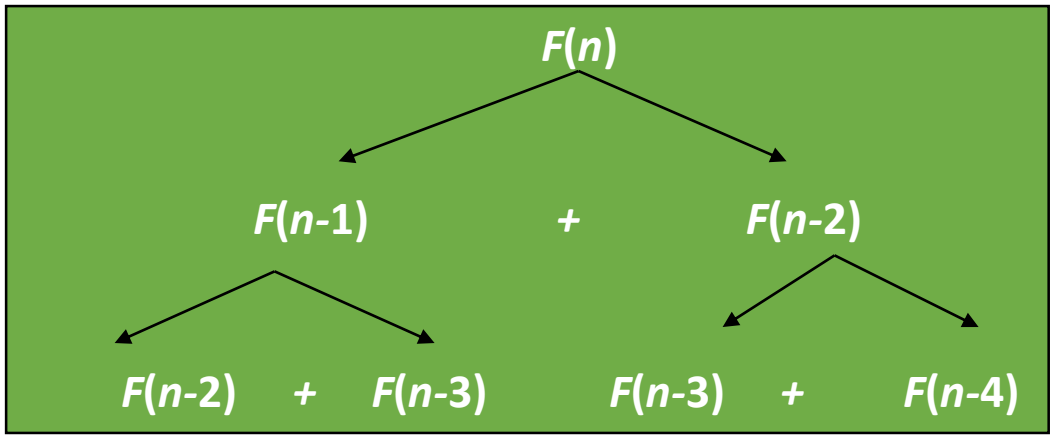
- Many aspects of nature are grouped in bunches equaling Fibonacci numbers.
- For example, the number of petals on a flower tend to be a Fibonacci number.
- Leaves and branches are also found in groups of Fibonacci numbers.



Computing the Fibonacci Numbers

- Computing the n^{th} Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - Top-down approach

```
int Fib(int n)
{
   if (n <= 1)
      return 1;
   else
      return Fib(n - 1) + Fib(n - 2);
}
```

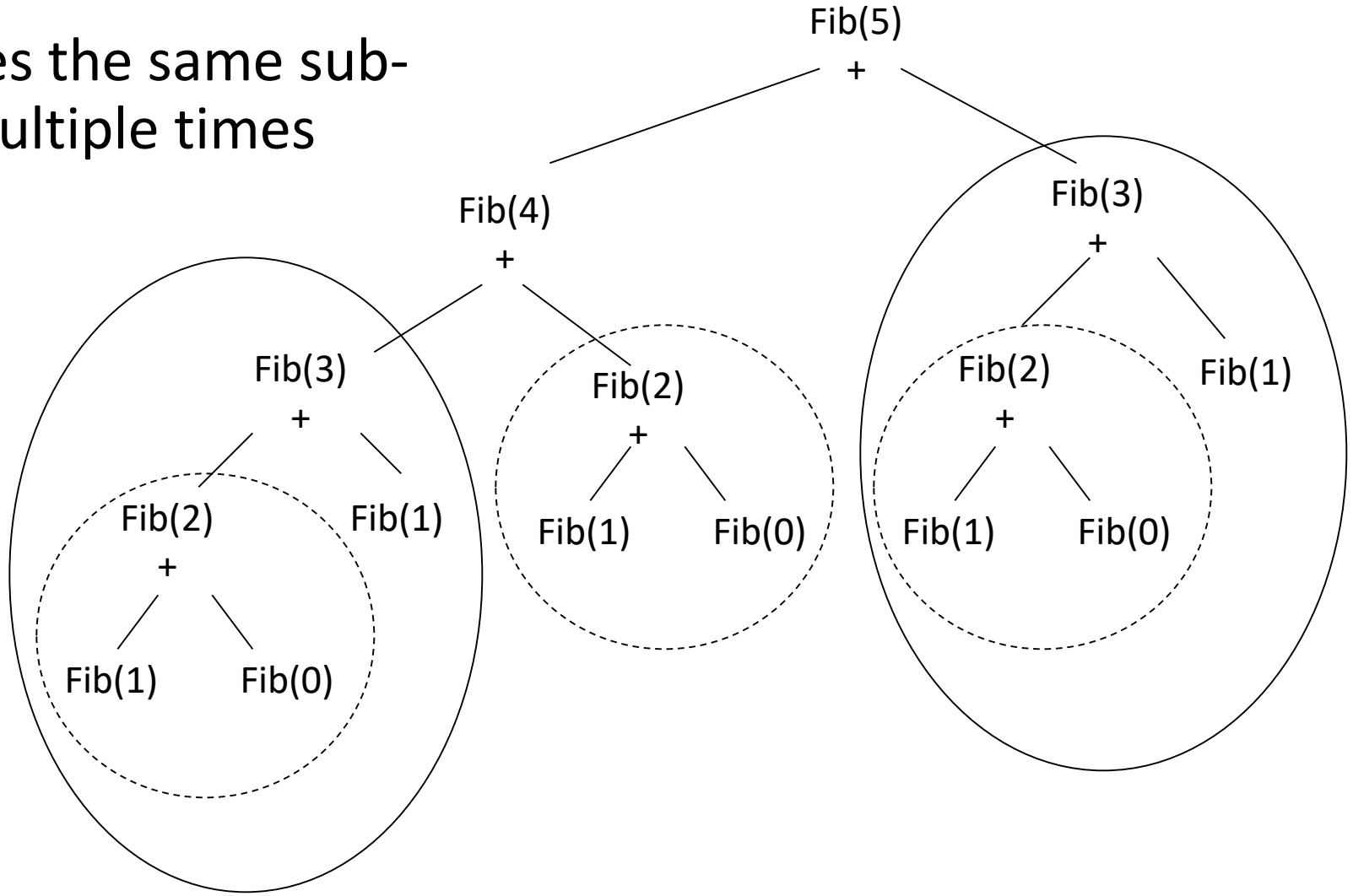


Time complexity of top-down approach

- What is the Recurrence relationship?
  - $T(n) = T(n-1) + T(n-2) + 1$
- What is the solution to this?
  - Since $T(n) < 2T(n-1)$ an upper bound is $O(2^n)$.
  - Since $T(n) > 2T(n-2)$ A lower bound is $\Omega(2^{n/2})$.
  - In fact $T(n) = \Theta(F(n))$, since T(n) grows very similarly to F(n),
- This is exponential in n; can we do better?

Why is the top-down approach so inefficient

- Recomputes the same sub-problem multiple times



Computing Fibonacci Numbers – bottom up

- Computing the n^{th} Fibonacci number using a bottom-up approach:
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

0	**1**	**1**	**. . .**	$F(n-2)$	$F(n-1)$	$F(n)$

- Efficiency:
  - Time – O(n) word operations
  - Space – O(n), actually O(1) since only last terms need remembering
    [will consider space again when studying edit distance and sequence alignment

Some technical issues

- Taking a precise bit-complexity viewpoint
  - Input is specified using log n bits
  - Output is $\Omega(n)$ bits or exponential
- Time complexity of bottom-up is $O(n^2)$
- Using a matrix powering approach can do O(nlogn)

Dynamic Programming

- The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.
- In dynamic programming we usually reduce time by increasing the amount of space
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

Summary

- *Dynamic programming* refers to a strategy of solving (and remembering) solutions to subproblems that can be used to compute larger solutions
  - Typically this involves filling in a table of problem solutions
  - Should be polynomial number of subproblems and a recurrence that describes how to combine the answers
  - Good fit when Divide and Conquer can't neatly divide up the work
- When solving problems recursively, *memoization* can ensure values aren't recomputed by remembering function call results directly
  - Without this, we may end up redoing a lot of work even while using the same basic recurrence as iterative approach
- Main Takeaway: Remember solutions to smaller sub-problems and reuse to solve the larger problem. Often beats divide & conquer and greedy