DYNAMIC PROGRAMMING

Analysis of Algorithm

Punjab University College of Information Technology (PUCIT)
University of the Punjab, Lahore, Pakistan.

Algorithm Design Paradigms

- Greedy Algorithms
  - Build up a solution incrementally
  - Myopically and locally optimizing some local criterion
- Divide and Conquer
  - Break up a problem into (independent) sub-problems
  - Solve each sub-problem independently
  - Combine solution to sub-problems to form solution to original problem
- Dynamic programming = planning over time
  - More general and powerful than divide and conquer
  - Break up a problem into (in)(dependent) sub-problems
  - Generally, there is a sequence of problems
  - Identify the optimal substructure: when optimal solution to a problem
  - is made up of optimal solution to smaller subproblems
  - Build up solution to larger and larger subproblems
  - Identify redundancy and repetitions
  - Use memoization or build up memo on the run

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method
- Divide and conquer is used for disjoint subproblems however dynamic programming is for overlap subproblems
- Here “Programming” refers to a tabular method, not to writing computer code.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Dynamic Programming (Cont.)

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Fibbonacci Series

- Fibonacci was born in Pisa (Italy), the city with the famous Leaning Tower
- Full name was Leonardo Pisano
- He introduced the decimal number system into Europe
- The original problem that Fibonacci investigated (in the year 1202) was about how fast rabbits could breed in ideal circumstances. Read more from: http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html

Fibbonacci Series

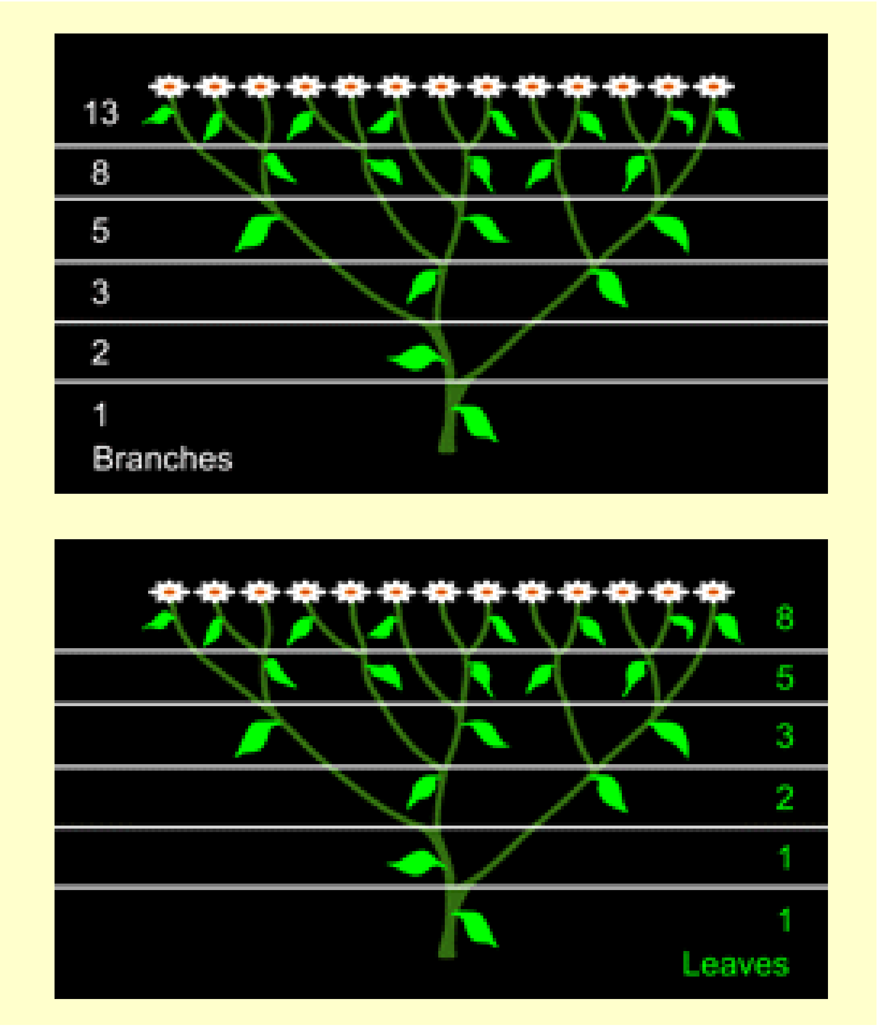
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

For $n \geq 8$ $\quad F_n > 2^{n/2}$

$\triangleright$ Prove it by induction

Fibbonacci Series (Cont.)



Source: http://britton.disted.camosun.bc.ca/fibslide/jbfibslide.htm

Fibbonacci Series

Implementing the recursive definition of F_n

function $\text{FIB1}(n)$

    if $n = 0$ **then**

        return 0

    else if $n = 1$ **then**

        return 1

    else

        return $\text{FIB1}(n-1) + \text{FIB1}(n-2)$

A call tree:

```
            5
          /   \
        4       3
       / \     / \
      3   2   2   1
     / \ / \ / \
    2  1 1  0 1  0
   / \
  1   0
```

Recursive F_n computation

Let $T(n)$ be the number of operations on input n

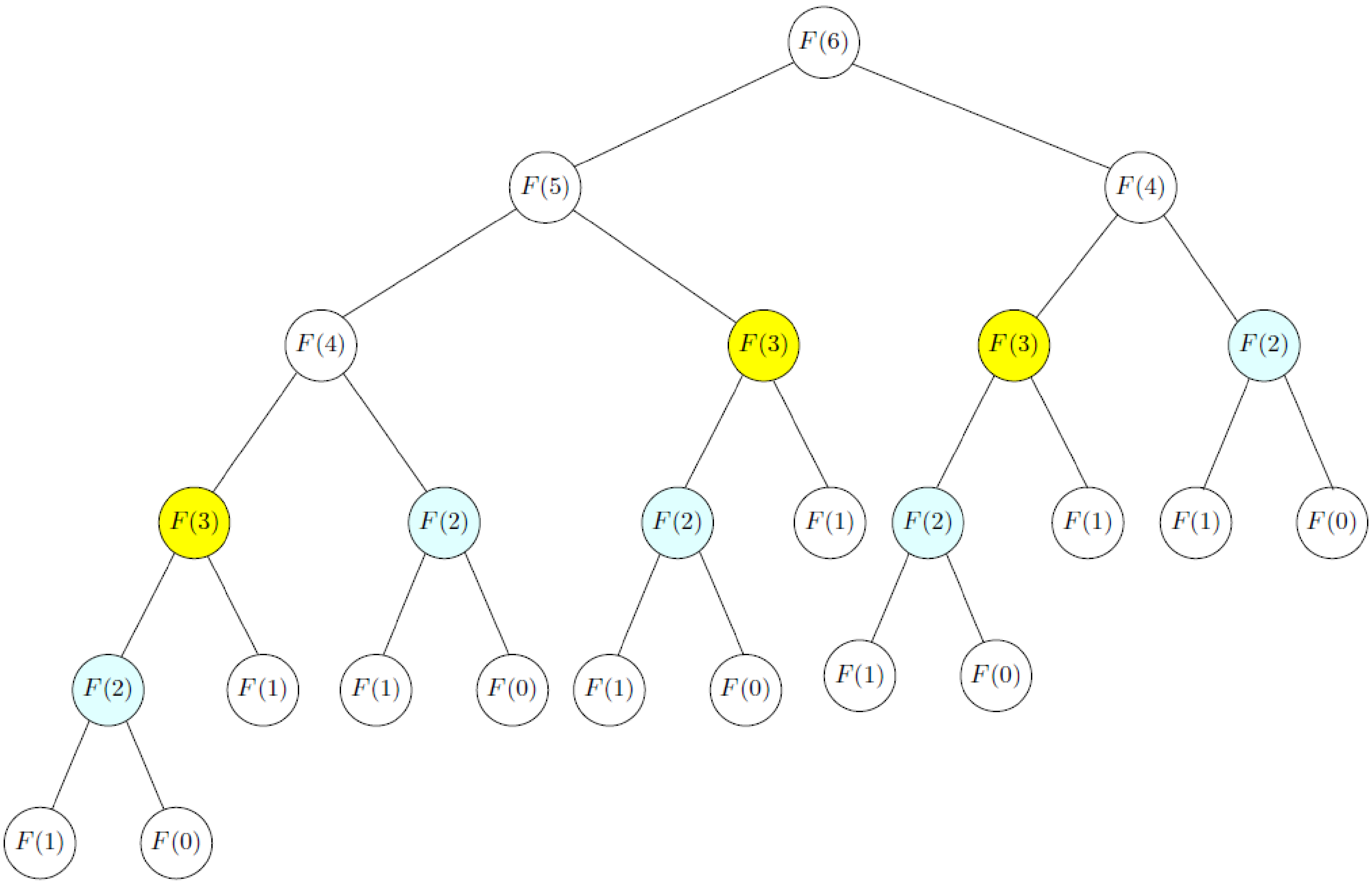
$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 3 & \text{if } n \geq 2 \end{cases}$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

For $n \geq 8$, $\quad T(n) > F_n \geq 2^{n/2}$ $\qquad \triangleright$ **exponential** in n

Problem is unnecessarily repeated recursive calls

Recursive Fn computation



Memoization

- Save results of subproblems in a memo
- Use the memo when needed instead of recomputing

Algorithm F_n computation with memoization

$F[0 \ldots n] \leftarrow \textsc{negOnes}(n+1)$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
function $\textsc{fib2}(n)$
  if $F[n-1] = -1$ **then**
    $F[n-1] \leftarrow \textsc{fib2}(n-1)$ ▷ Call $\textsc{fib2}$ function only if $F[n-1] = -1$
  if $F[n-2] = -1$ **then**
    $F[n-2] \leftarrow \textsc{fib2}(n-2)$
  return $F[n-1] + F[n-2]$

F_n computation with Memoization

Algorithm Compute F_n with memo

$F[0 \ldots n] \leftarrow \textsc{negOnes}(n+1)$

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

function $\textsc{Fib2}(n)$

  if $F[n-1] = -1$ **then**

    $F[n-1] \leftarrow \textsc{Fib2}(n-1)$

  if $F[n-2] = -1$ **then**

    $F[n-2] \leftarrow \textsc{Fib2}(n-2)$

  return $F[n-1] + F[n-2]$

- Let T(n) be runtime of fib2(n)
- Count number of calls
- Only calls if F[·] = −1
- Total calls n + 1
- O(1) operations per call
- T(n) = O(n)

▷ Compare with $T(n) = O(2^n)$

F_n computation Bottom Up Approach

Algorithm Bottom-Up F_n Computation

$F[0 \dots n] \leftarrow \text{NEGONES}(n+1)$

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

for $i = 2$ to n **do**

  $F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

- No recursion overhead
- Analyze time needed to fill up memo
- Total number of updates to memo is n + 1
- Total runtime T(n) = O(n)

▷ Compare with $T(n) = O(2^n)$