Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 6: this Friday @ 11:59 pm
  - Homework 7: next Friday @ 11:59 pm
  - Lab 6: Monday 10/31 @ 11:59 pm
- Midterm Exam: Thursday 10/20
  - closed book, paper, in-person
- Live QA Session on Piazza every Friday 4:30-5:30 pm

Previous Lecture …

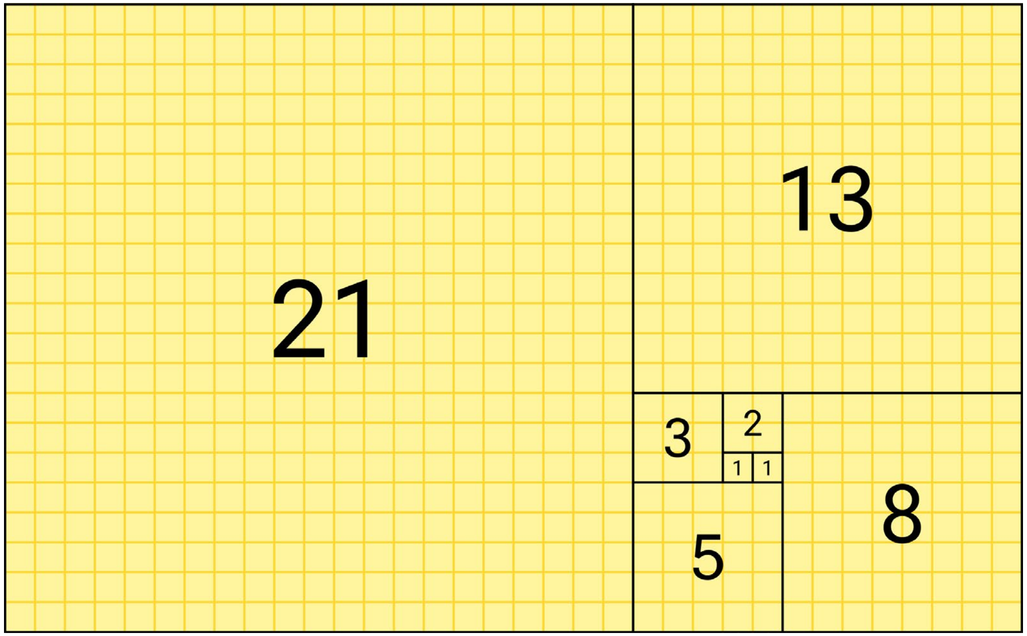
- Recursion Applications
  - Divide and Conquer
  - Backtracking
- Limitations of Recursion

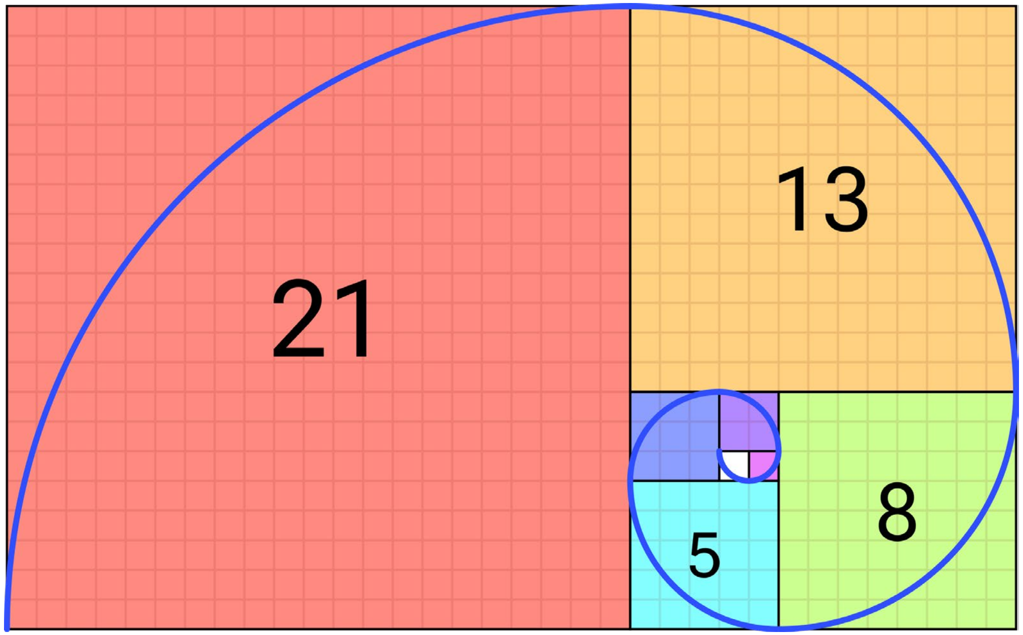
Today …

- More recursion examples
  - Fibonacci numbers
  - linear and binary search
  - finding words in a grid of letters
- Recursion tree analysis
- Recursion may lead to poor solutions
- Average and amortized running time analysis

Generating Fibonacci Numbers

- Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ….





Fibonacci number. (2022, October 13). In Wikipedia.
https://en.wikipedia.org/wiki/Fibonacci_number

Generating Fibonacci Numbers

- Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ….

Algorithm Fibonacci(n)

```
if (n <= 1)
    return 1
```

Generating Fibonacci Numbers

- Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ….

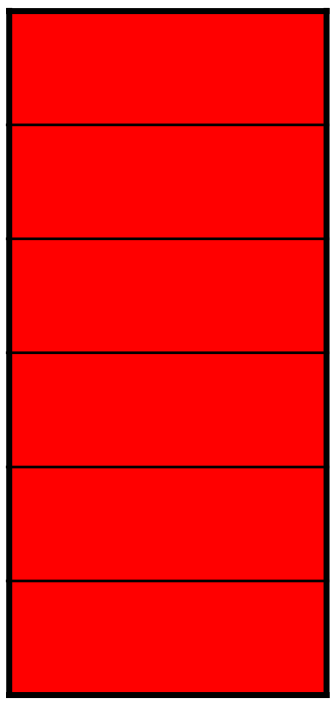
Algorithm Fibonacci(n)

```
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

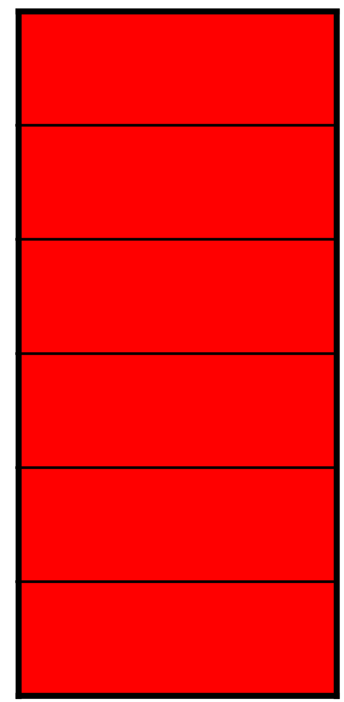
Double Recursion!

Single recursion

- A recursive algorithm with a single recursive call provides a linear chain of calls



Calls build run-time stack

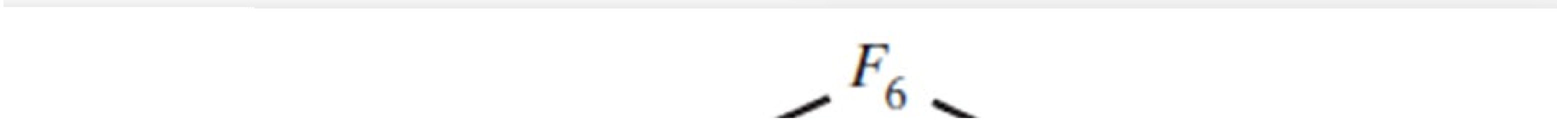


Stack shrinks as calls finish

Double Recursion

- The computation of the Fibonacci number F_6 using recursion

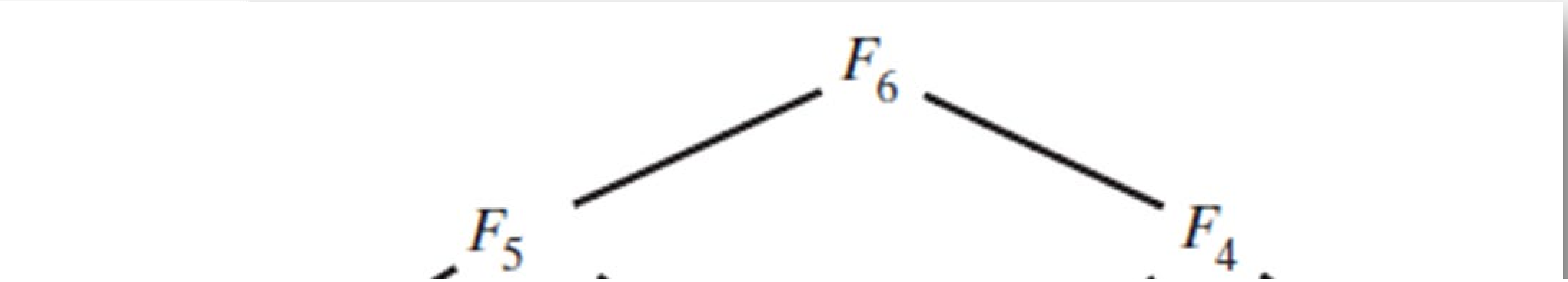
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

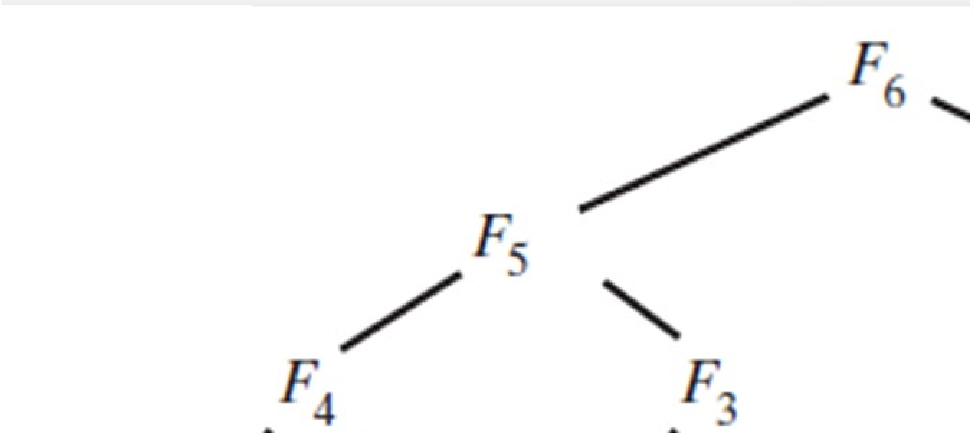
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

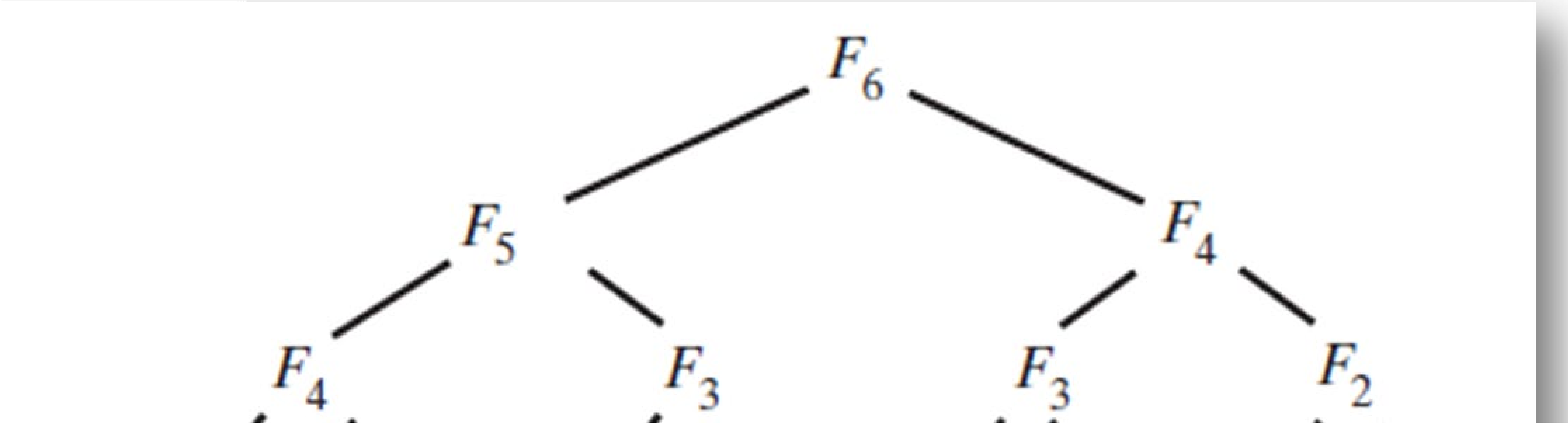
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

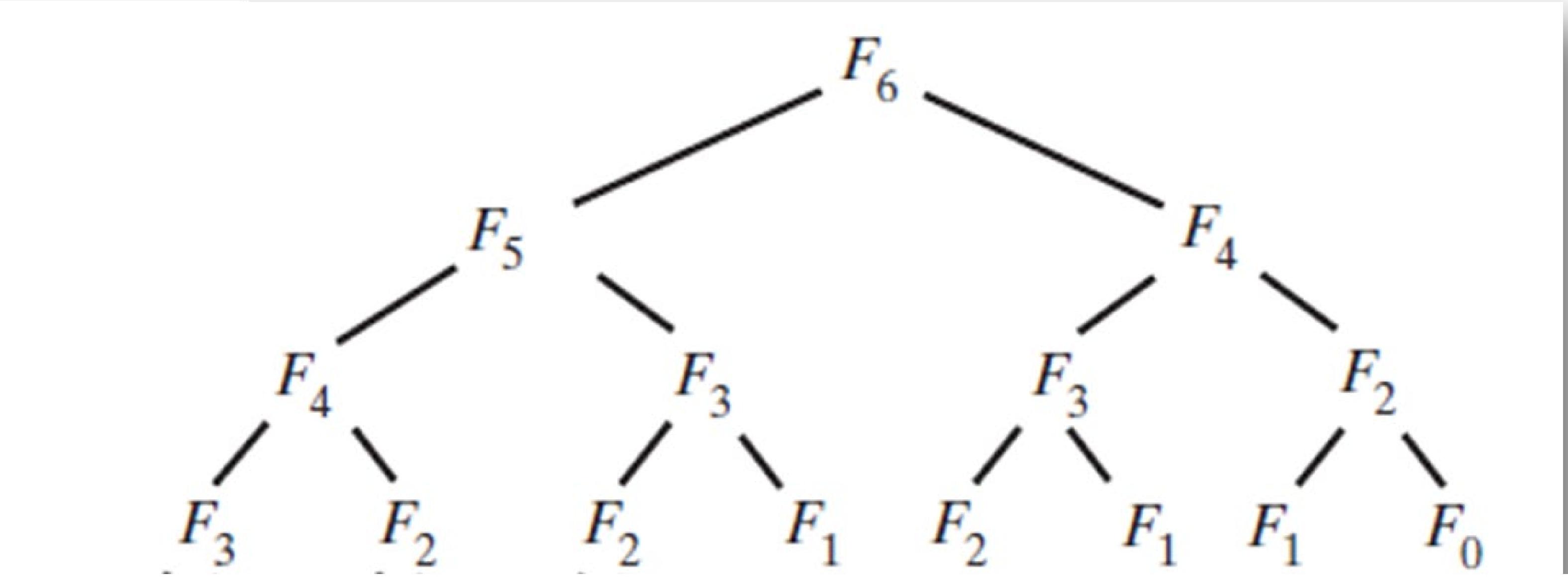
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

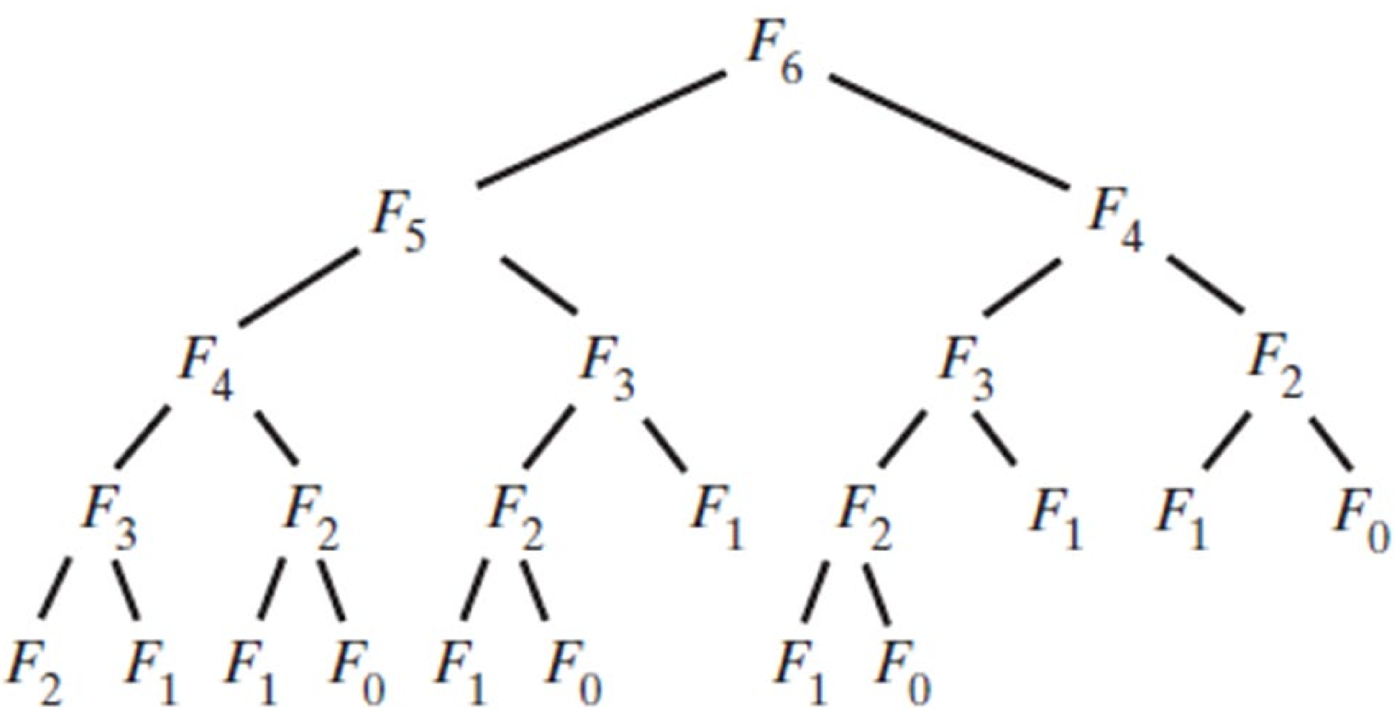
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

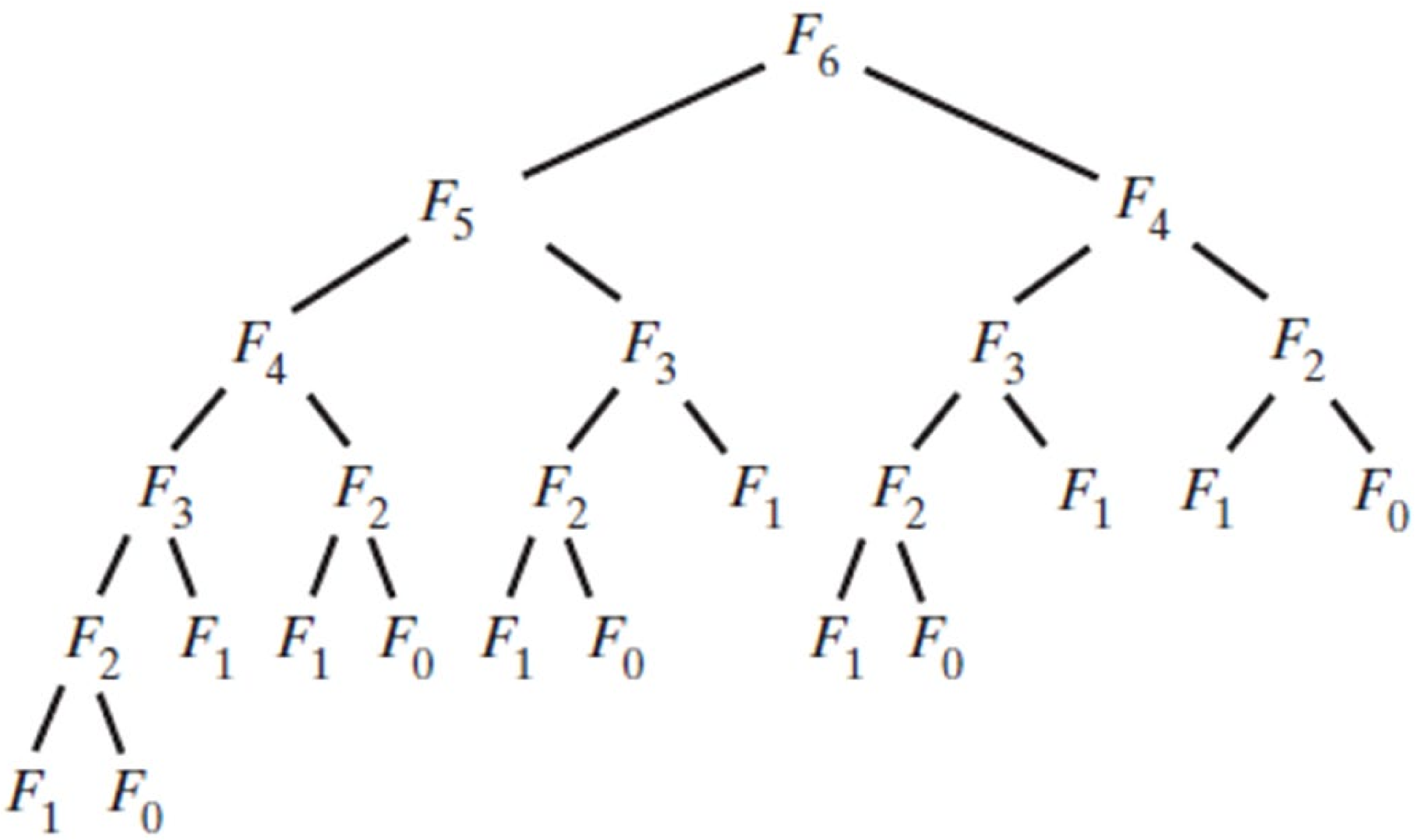
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

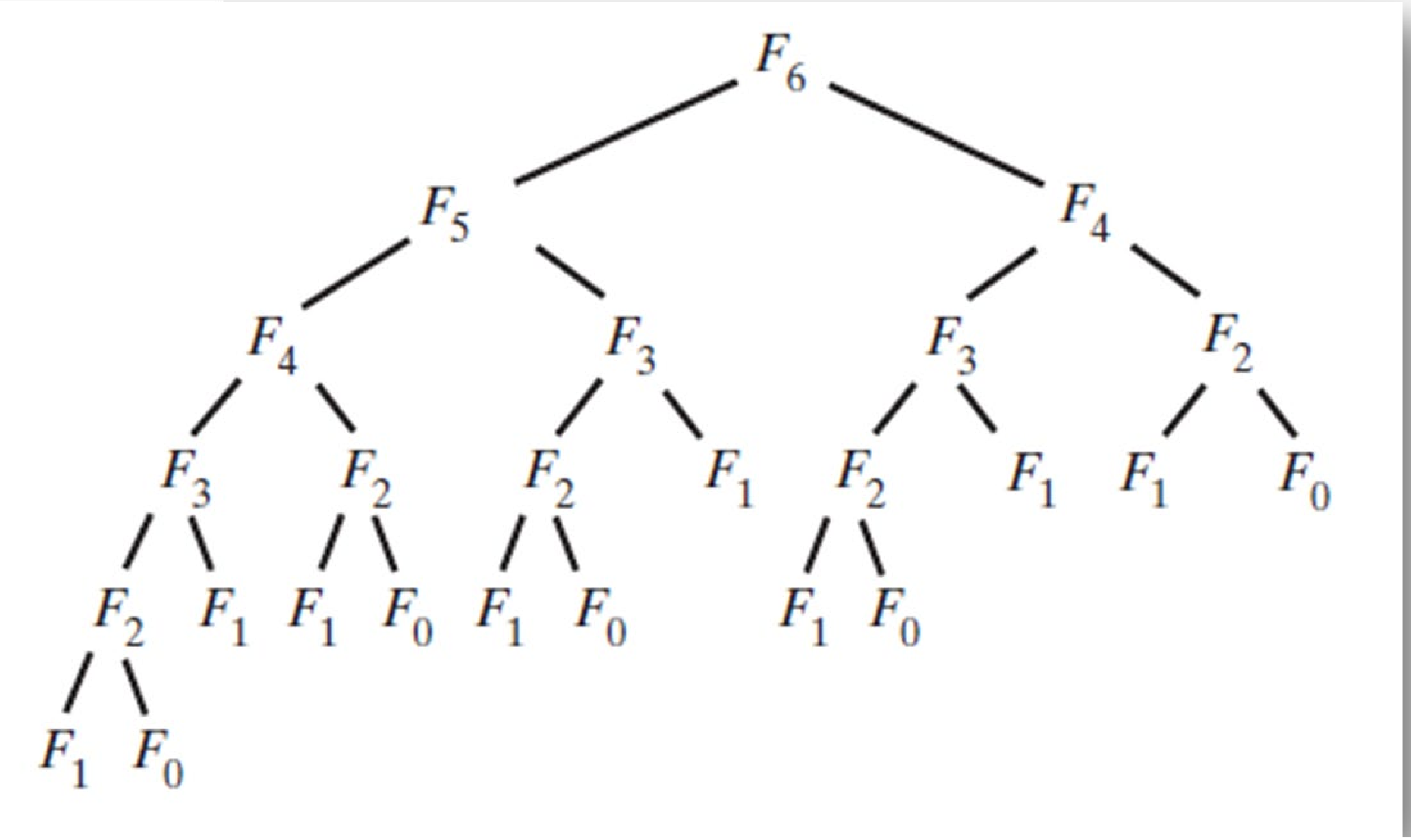
- The computation of the Fibonacci number F_6 using recursion

```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```



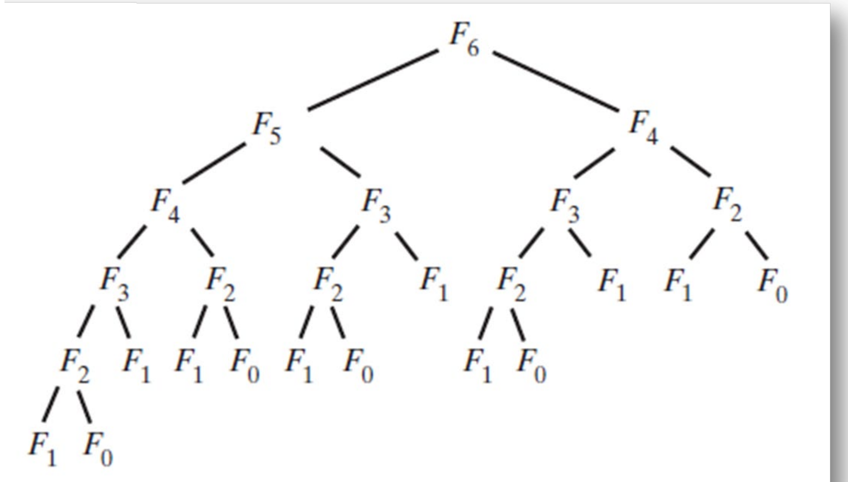
Recursion Tree

- This branching structure is called a tree
  - recursion tree
- Each **node** represents one recursive call
- Terminal nodes are called **leaves**



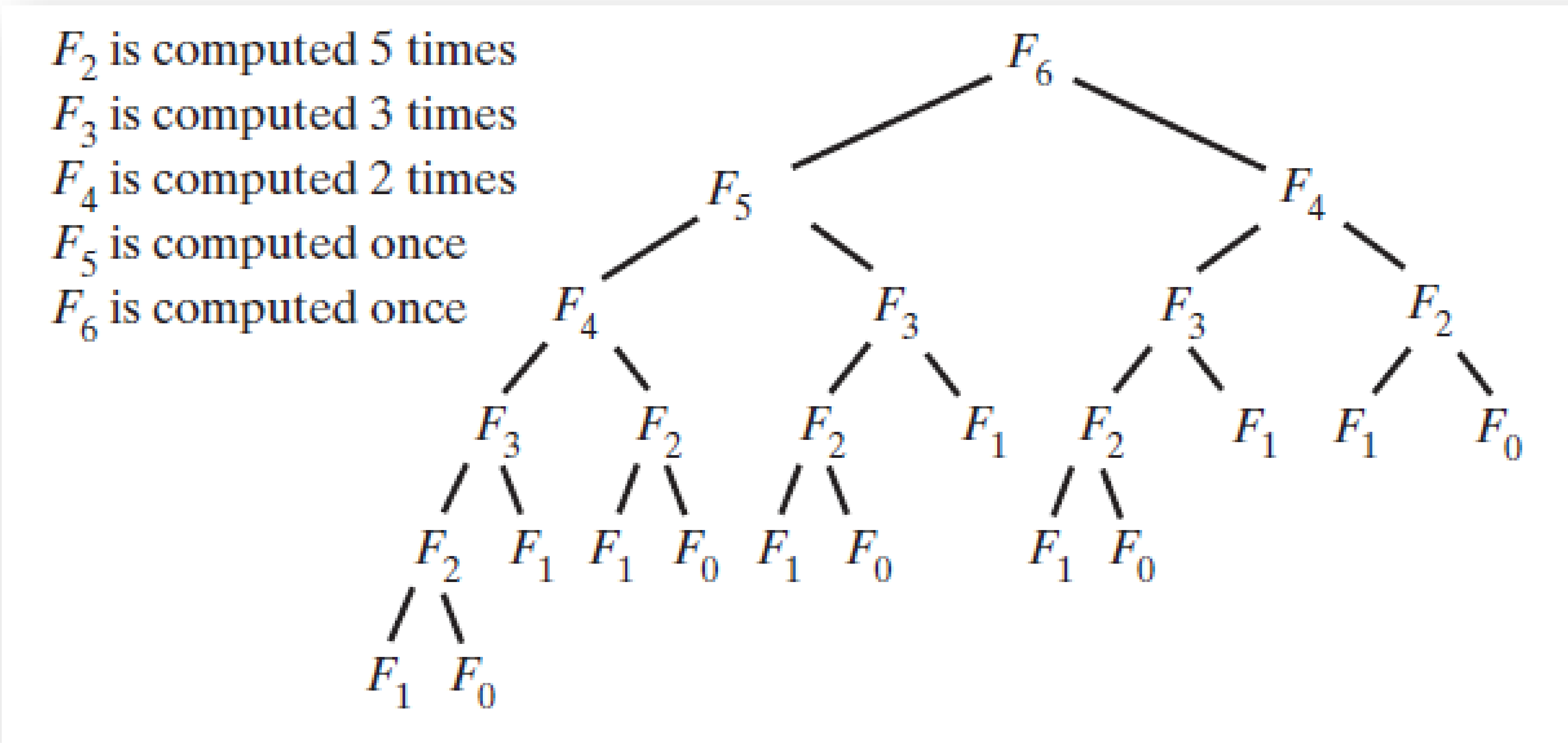
Recursion Tree Analysis

- We can use the recursion tree to estimate the running time of a recursive algorithm
  - running time = number of nodes * time per node
- For the tree below,
  - the number of nodes for each level almost **doubles** as the tree grows down
    - first level: 1 node
    - second level: 2 nodes
    - third level: 4 nodes
    - …
  - number of nodes <= 1 + 2 + 4 + 8 + 16 + …
  - the number of levels = the value of *n*
    - e.g., the recursion tree for F_6 has 6 levels
  - number of nodes <= 1 + 2 + 4 + 8 + … + 2^{n-1}
  - <= $2*2^{n-1} = 2^n$
  - Time per node is O(1)
  - So, running time = $O(2^n)$
  - Exponential!



Double Recursion

- Recursion may lead a poor solution that an iterative approach



Searching

Recursive Sequential Search of an Unsorted Array

- Method that implements this algorithm will need parameters **`first`** and **`last`**.

```
private static <T> boolean search(T[] anArray, int first, int last,
T desiredItem)
{
   boolean found;
   if (first > last)
      found = false; // No elements to search
   else if (desiredItem.equals(anArray[first]))
      found = true;
   else
      found = search(anArray, first + 1, last, desiredItem);
   return found;
} // end search
```

Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that finds its target

(a) A search for 8

Look at the first entry, 9:

9	5	8	4	7

$8 \neq 9$, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

$8 \neq 5$, so search the next subarray.

Look at the first entry, 8:

8	4	7

$8 = 8$, so the search has found 8.

Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that does not find its target

(b) A search for 6

Look at the first entry, 9:

9	5	8	4	7

$6 \neq 9$, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

$6 \neq 5$, so search the next subarray.

Look at the first entry, 8:

Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that does not find its target

6 ≠ 5, so search the next subarray.

Look at the first entry, 8:

8	4	7

6 ≠ 8, so search the next subarray.

Look at the first entry, 4:

4	7

6 ≠ 4, so search the next subarray.

Look at the first entry, 7:

7

Recursive Sequential Search of an Unsorted Chain

Implementation of the method **search**

```
// Recursively searches a chain of nodes for desiredItem,
// beginning with the node that currentNode references.
private boolean search(Node currentNode, T desiredItem)
{
   boolean found;

   if (currentNode == null)
      found = false;
   else if (desiredItem.equals(currentNode.getData()))
      found = true;
   else
      found = search(currentNode.getNextNode(), desiredItem);

   return found;
} // end search
```

Efficiency of a Sequential Search

The time efficiency of a sequential search of a chain of linked nodes

- Best case: O(1)
- Worst case: O(*n*)

Average-case Analysis of Sequential Search

- To do this we need to make an assumption about the index where the target exists
- Let's assume that all index values are equally likely
  - If this is not the case, we can still do the analysis, if we know the actual probability distribution for the index
- Our assumption means that, given n choices for an index, the probability of stopping at a given index, i, (which we will call P(i)) is
  - 1/n for any i
- Let's define our key operation to be "looking at" an entry in the list
  - So for a given index i, we will require i operations
  - Let's call this value Ops(i)

Average-case Analysis of Sequential Search

- Now we can define the average number of operations to be:

  Avg Ops = Sum_over_i (Ops(i) * P(i))

  = Sum_over_i (i * 1/n)

  = 1/n * Sum_over_i (i)

  = 1/n * [n * (n+1)]/2

  = (n+1)/2
- This is for success case (target found)
- Running time for the failed search case?
  - n
- **overall average**: successful search probability * (n+1)/2 +

  failed search probability * n
- In an absolute sense, this is better than the worst case, but asymptotically it is the same (why?)
- So in this case the worst and average cases are the same

Amortized Analysis

- Average over a sequence of operations
- **add(newEntry) of ArrayList**
  - Recall that this version of the method adds to the end of the list
  - Runtime for Resizable Array ?

O(1): We can go directly to the last location and insert there

- The answer above is a bit deceptive
- Some adds take significantly more time, since we have to first allocate a new array **and copy all of the data** into it –
  - O(n) time
- So we have O(n) + O(1) → O(n) total
  - when resizing happens!

Amortized Analysis

- So, we have an operation that sometimes takes O(1) and sometimes takes O(N)
- How do we handle this issue?

- Amortized Time (see http://en.wikipedia.org/wiki/Amortized_analysis)
- Average time required over a sequence of operations
- Individual operations may vary in their run-time, but we can get a consistent time for the overall sequence
- Let's stick with the add() method for resizable array list and consider 2 different options for resizing:
  1) Increase the array size by 1 each time we resize
  2) Double the array size each time we resize (which is the way the authors actually did it)