University of Pittsburgh

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 …

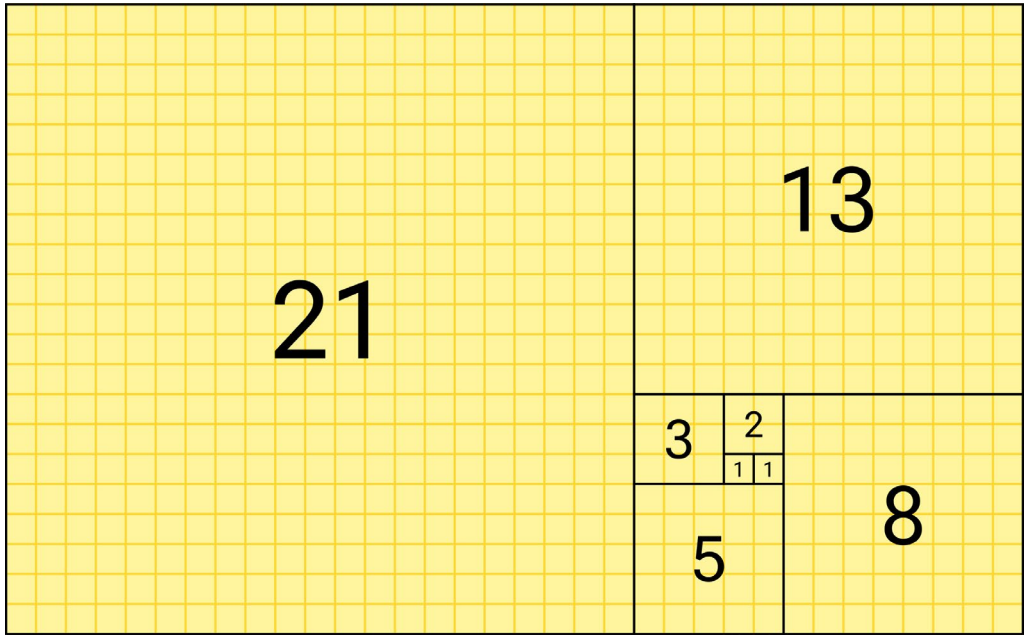
- Recursion Applications
  - Divide and Conquer
  - Backtracking
- Limitations of Recursion

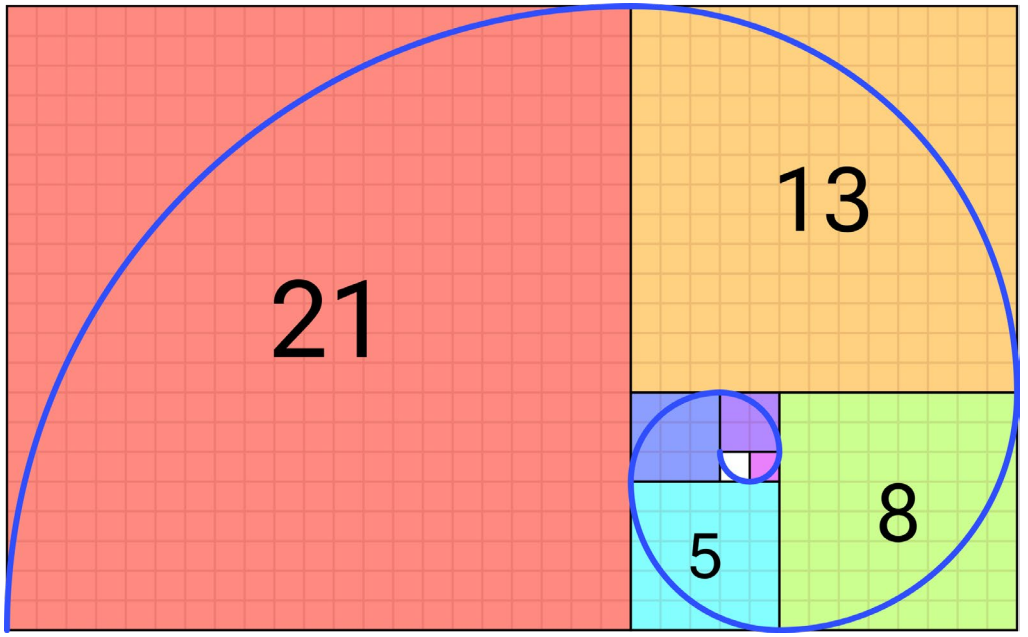
Today …

- 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, 24, 35, 59, ….





Fibonacci number. (2022, October 13). In Wikipedia.
https://en.wikipedia.org/wiki/Fibonacci_number
CC BY-SA 4.0

Generating Fibonacci Numbers

- Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 24, 35, 59, ….

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

Generating Fibonacci Numbers

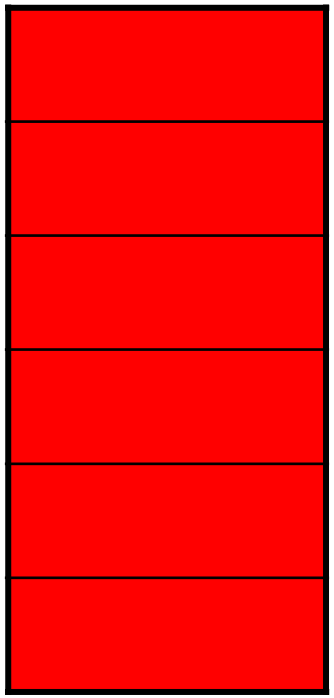
- Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 24, 35, 59, ….

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

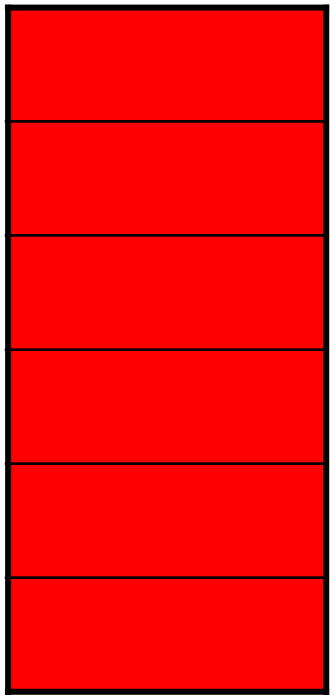
Double Recursion!

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

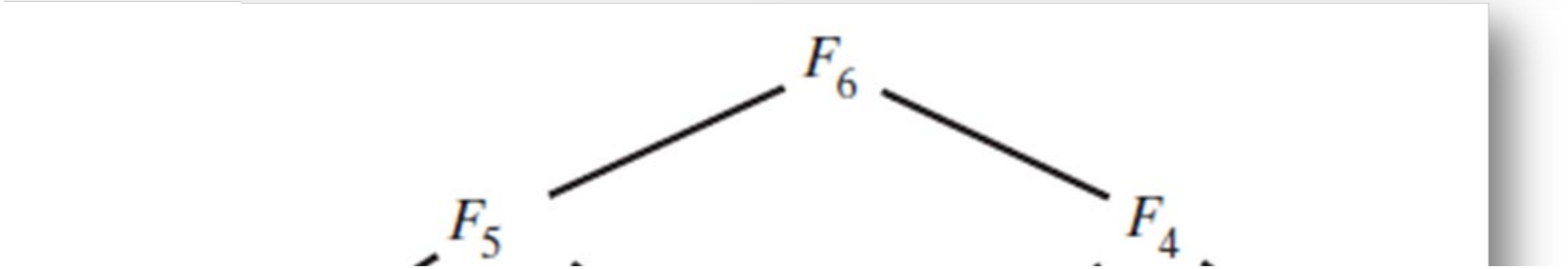
- The computation of the Fibonacci number F_6 using recursion

```
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

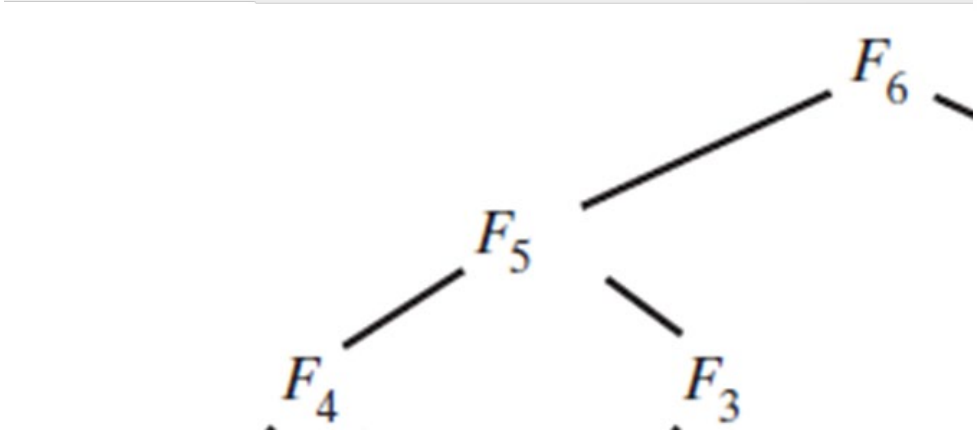
```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

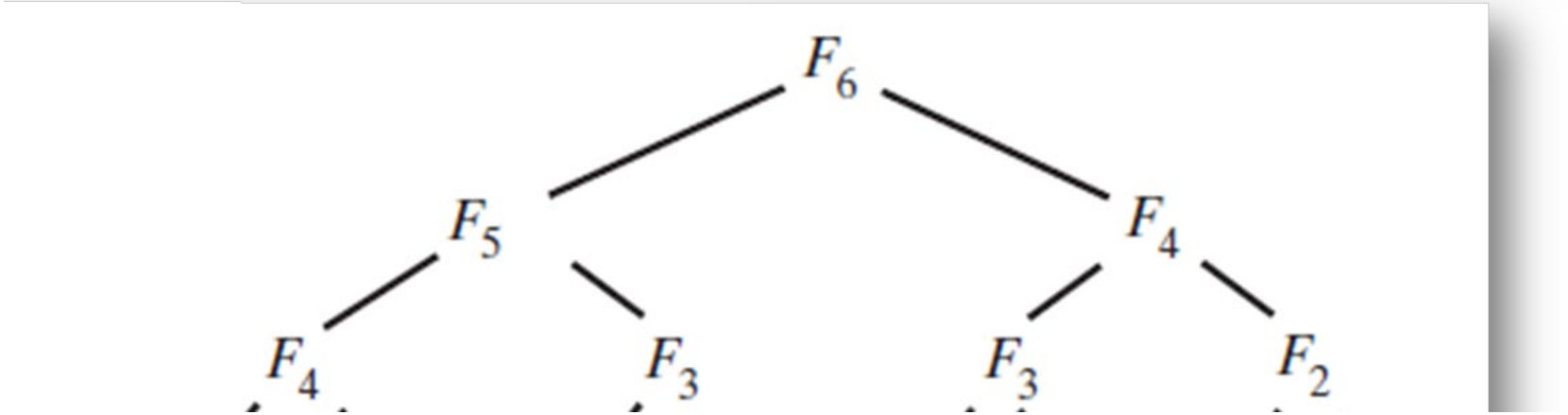
```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

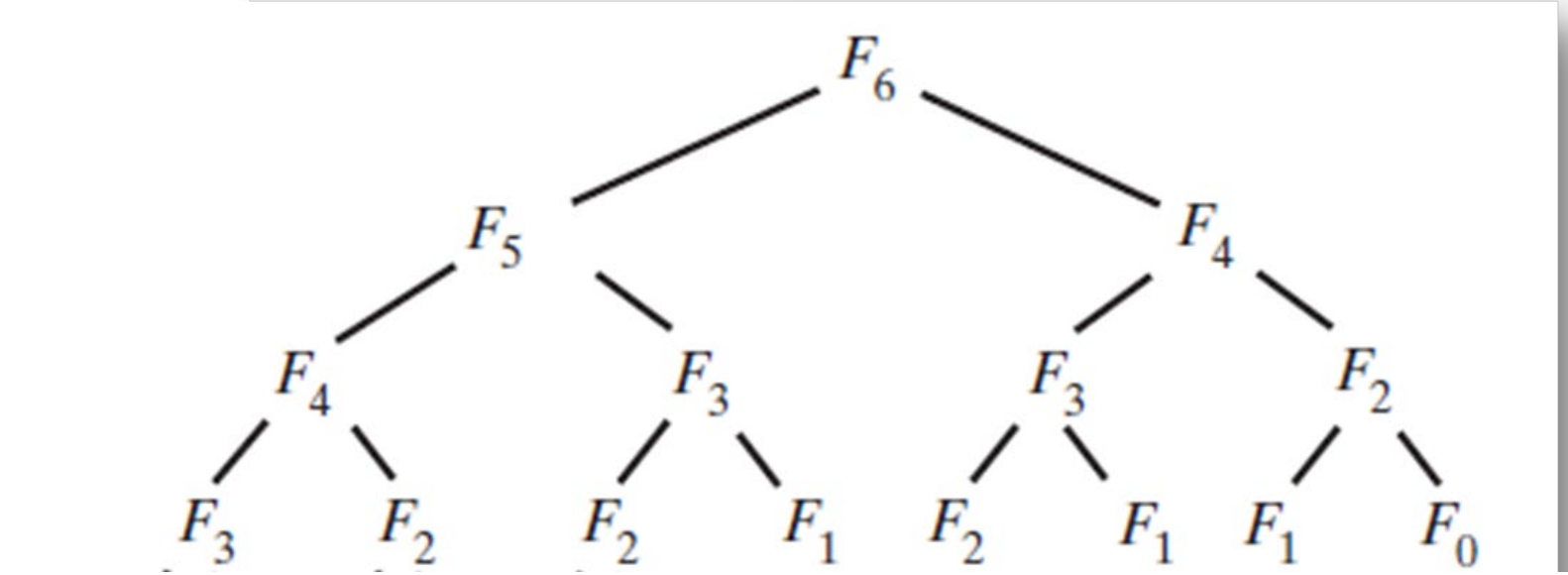
```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

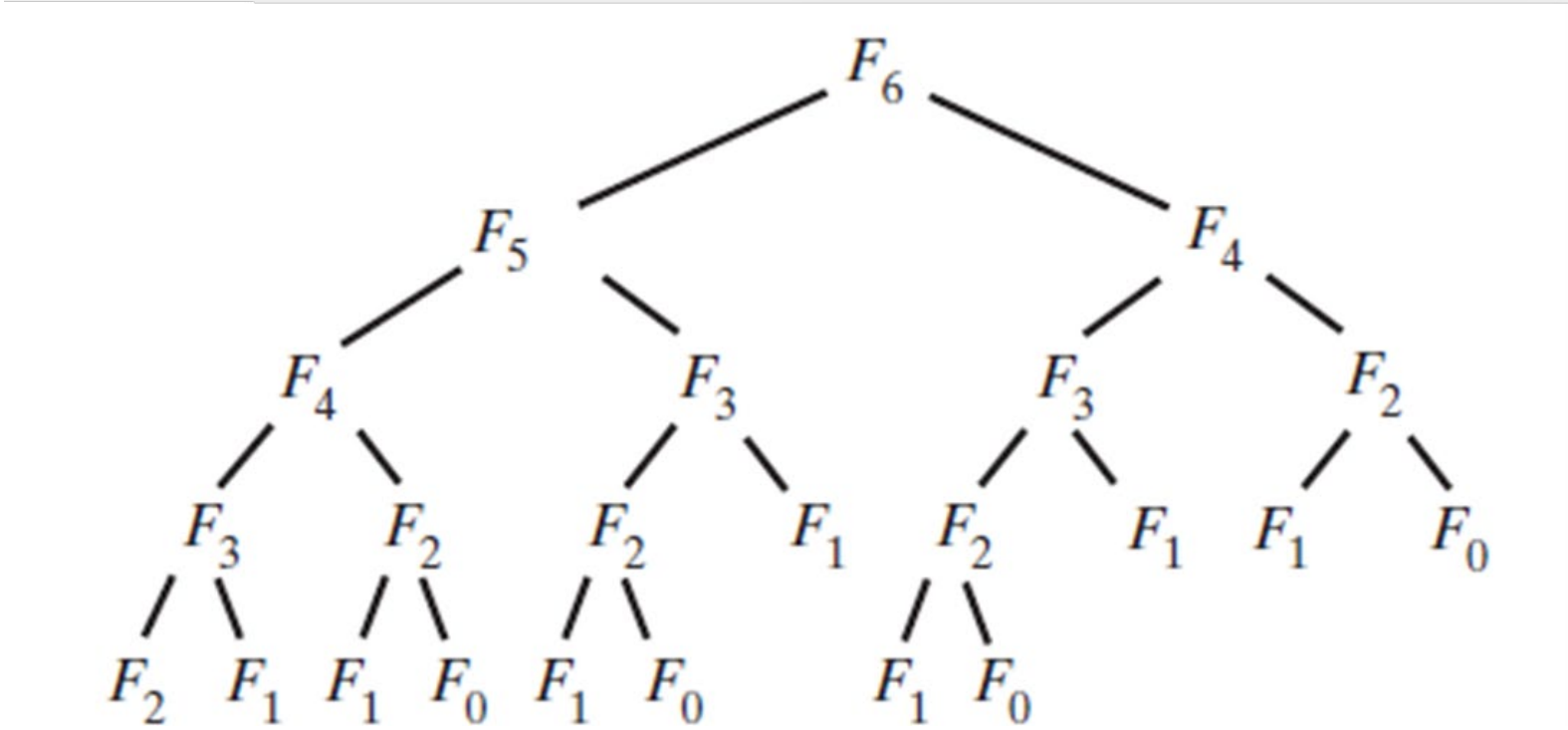
```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

- The computation of the Fibonacci number F_6 using recursion

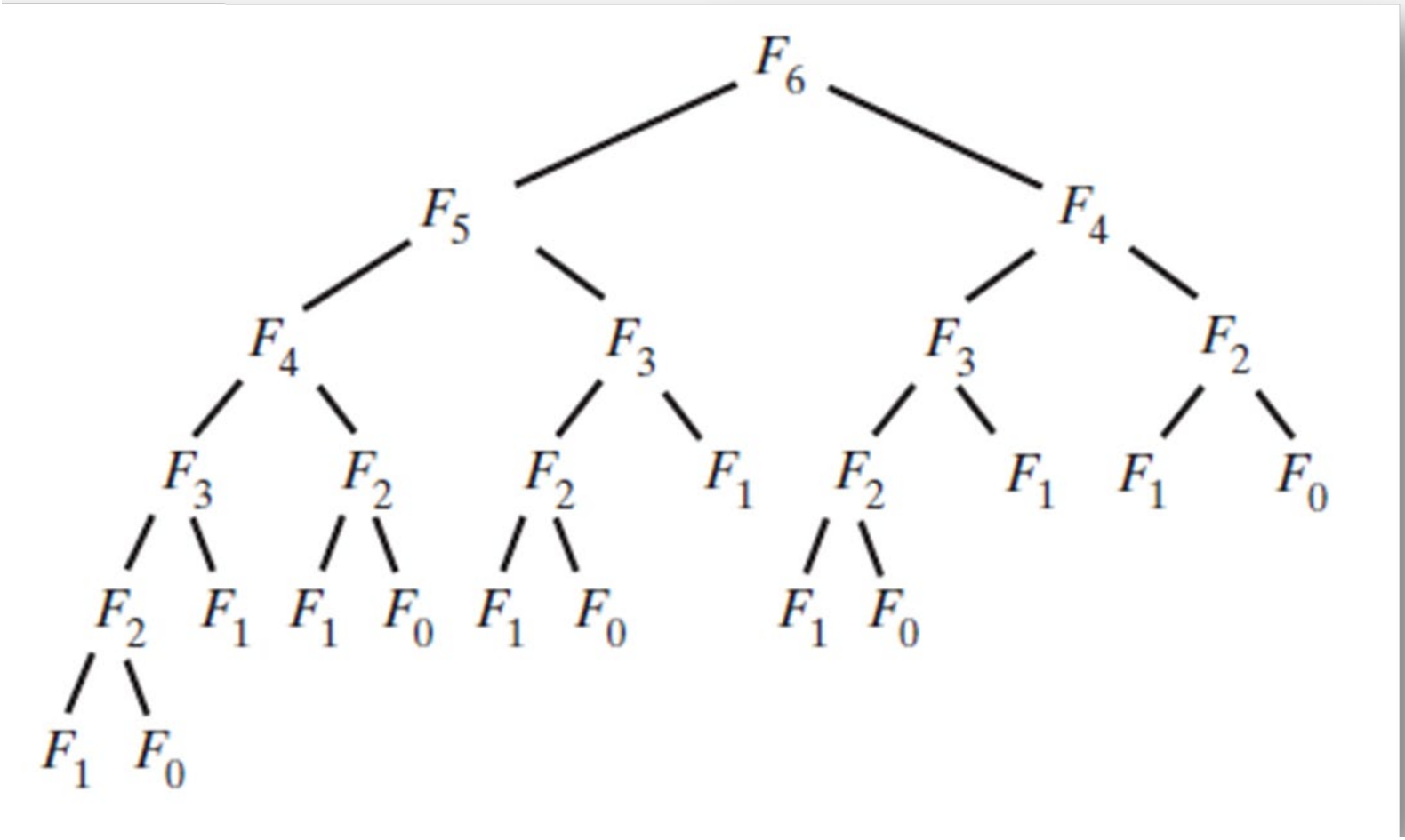
```
Algorithm Fibonacci(n)
if (n <= 1)
   return 1
else
   return Fibonacci(n - 1) + Fibonacci(n - 2)
```



Double Recursion

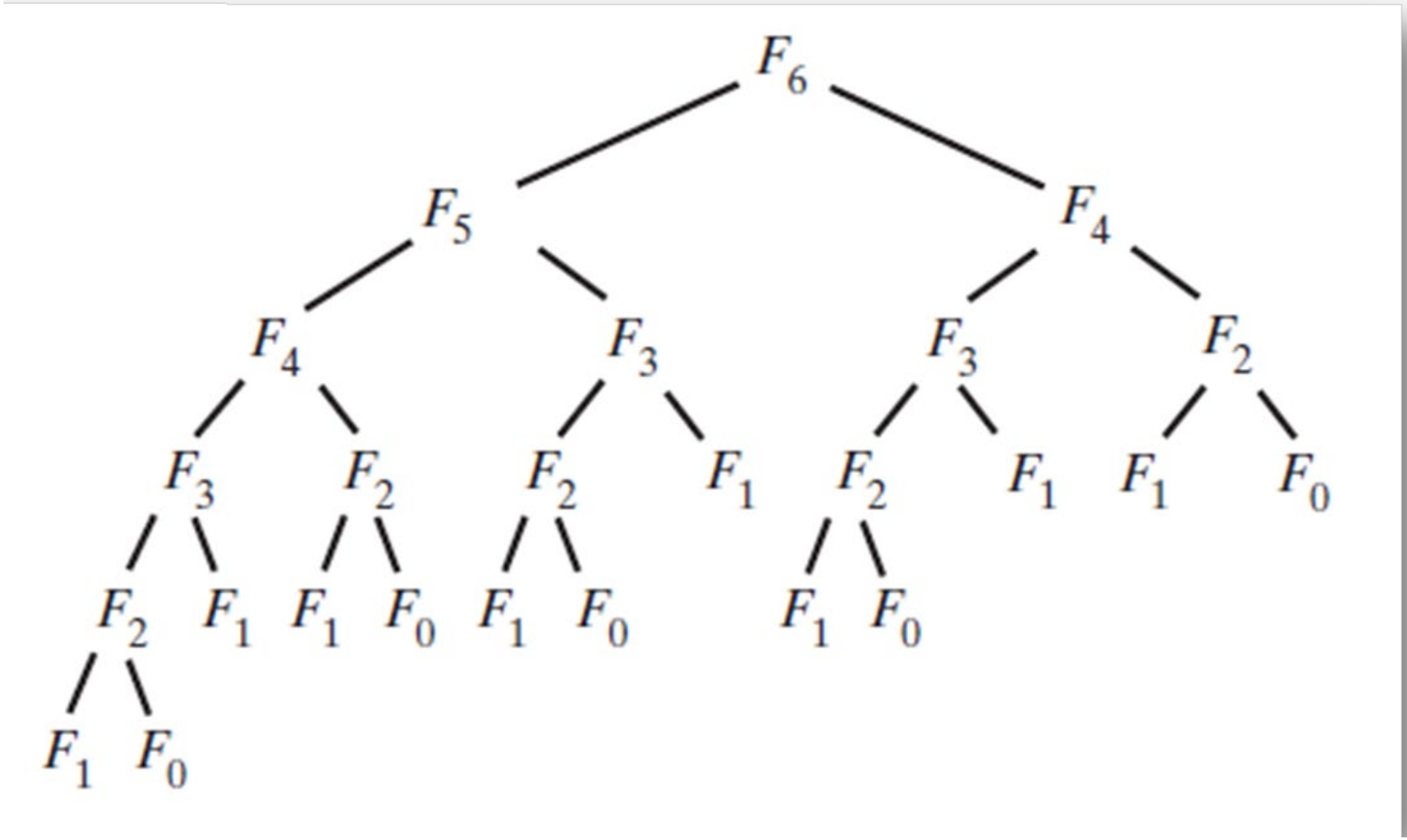
- The computation of the Fibonacci number F_6 using recursion

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



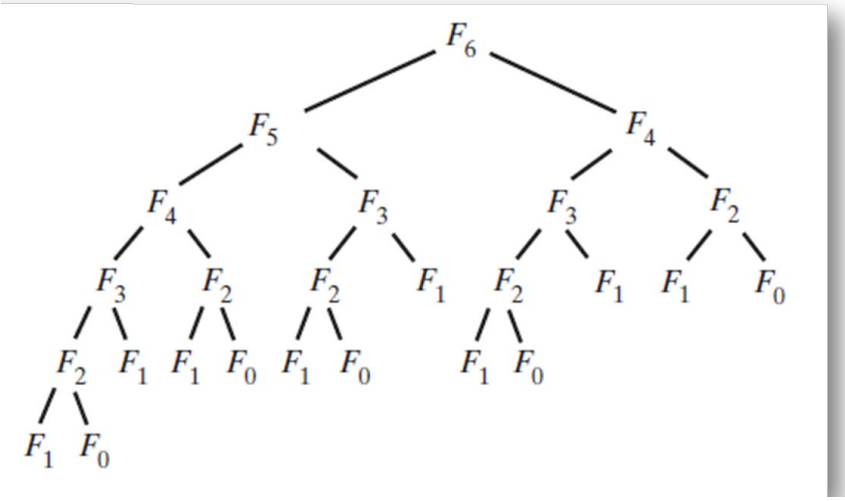
Recursion Tree

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



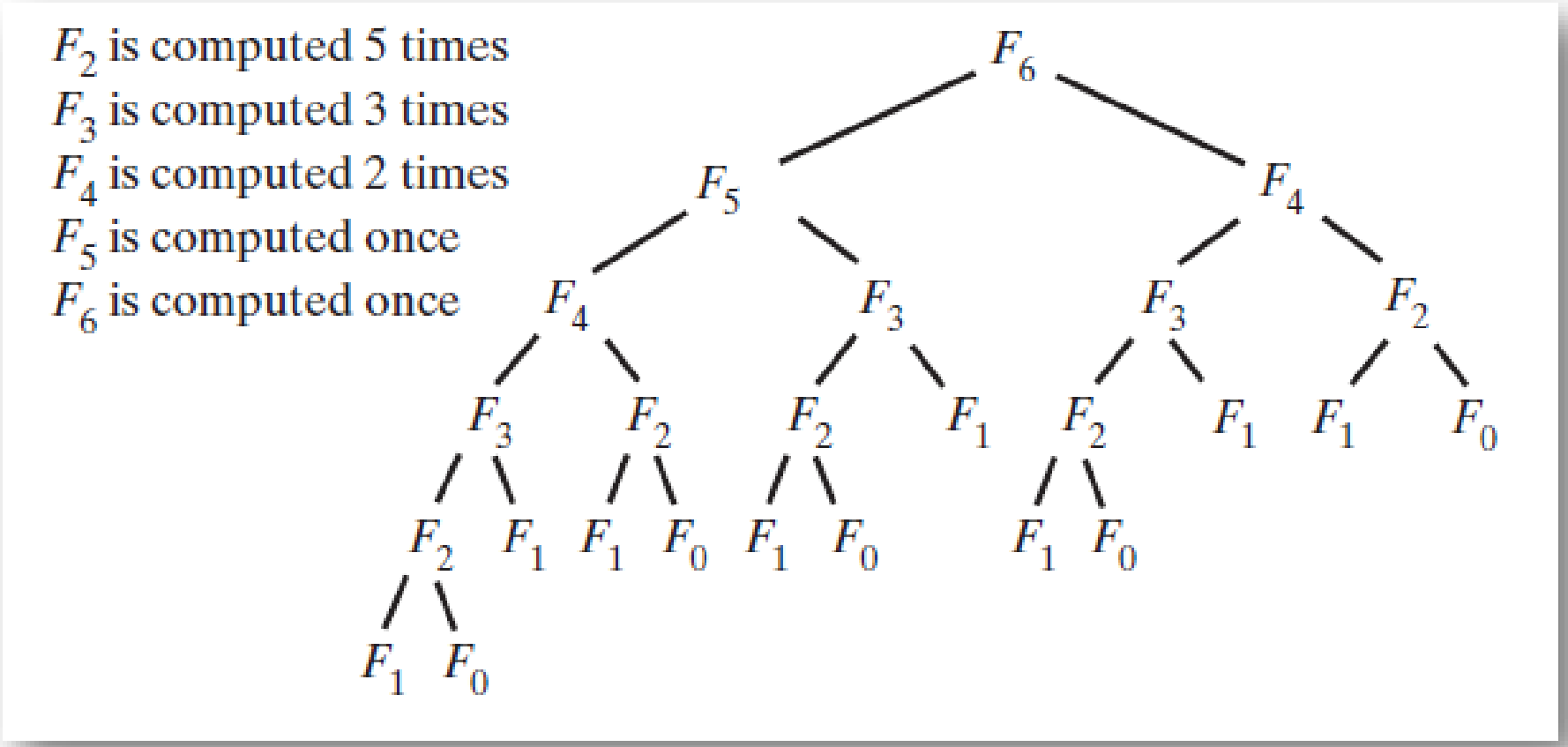
Recursion Tree Analysis

- 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 <= $1 + 2 + 4 + 8 + \ldots + 2^{n-1}$
  - <= $2 * 2^{n-1} = 2^n$
  - 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

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

Look at the first entry, 8:

8	4	7

$6 \neq 8$, so search the next subarray.

Look at the first entry, 4:

4	7

$6 \neq 4$, 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)

Amortized Analysis

1) Increase the array size by 1 each time we resize

- Note that with this approach, once we resize we will have to do it with every add
- Thus, rather than O(1) our add() is now O(n) **all the time**
- Specifically, assume the initial array size is 1
  - On insert 1 we just add the item (1 assignment)
  - On insert 2 we allocate and assign 2 items
  - On insert 3 we allocate and assign 3 items
  - …
  - Overall, **for n add() ops** look at the total number of assignments we have to make:

1 + 2 + 3 + … + n =

$$n(n+1)/2 \rightarrow O(n^2)$$

Amortized Analysis

2) Double the array size each time we resize

Insert #	# of assignments	End array size
1	1	1
2	$2 = 1 + 2^0$	2
3	$3 = 1 + 2^1$	4
4	1	4
5	$5 = 1 + 2^2$	8
…	1	8
9	$9 = 1 + 2^3$	16
…	1	16
17	$17 = 1 + 2^4$	32
…	1	32
32	1	32

Amortized Analysis

- Note that every row has 1 assignment (blue)
- Rows that are $2^K + 1$ for some K have an additional 2^K assignments (red) to copy data
- So for n **adds**, we have a total of
  - n assignments for the actual add
  - $2^0 + 2^1 + \ldots + 2^x$ for the copying
  - We stop when $1+2^x <= n < 1+2^{x+1}$
  - What is x? $\text{floor}(\lg_2(N-1))$
  - This gives us the geometric series
  - $\sum_{i=0}^{lg_2(n-1)} 2^i = O\left(2^{lg_2(n-1)}\right) = O(n)$

Amortized Analysis

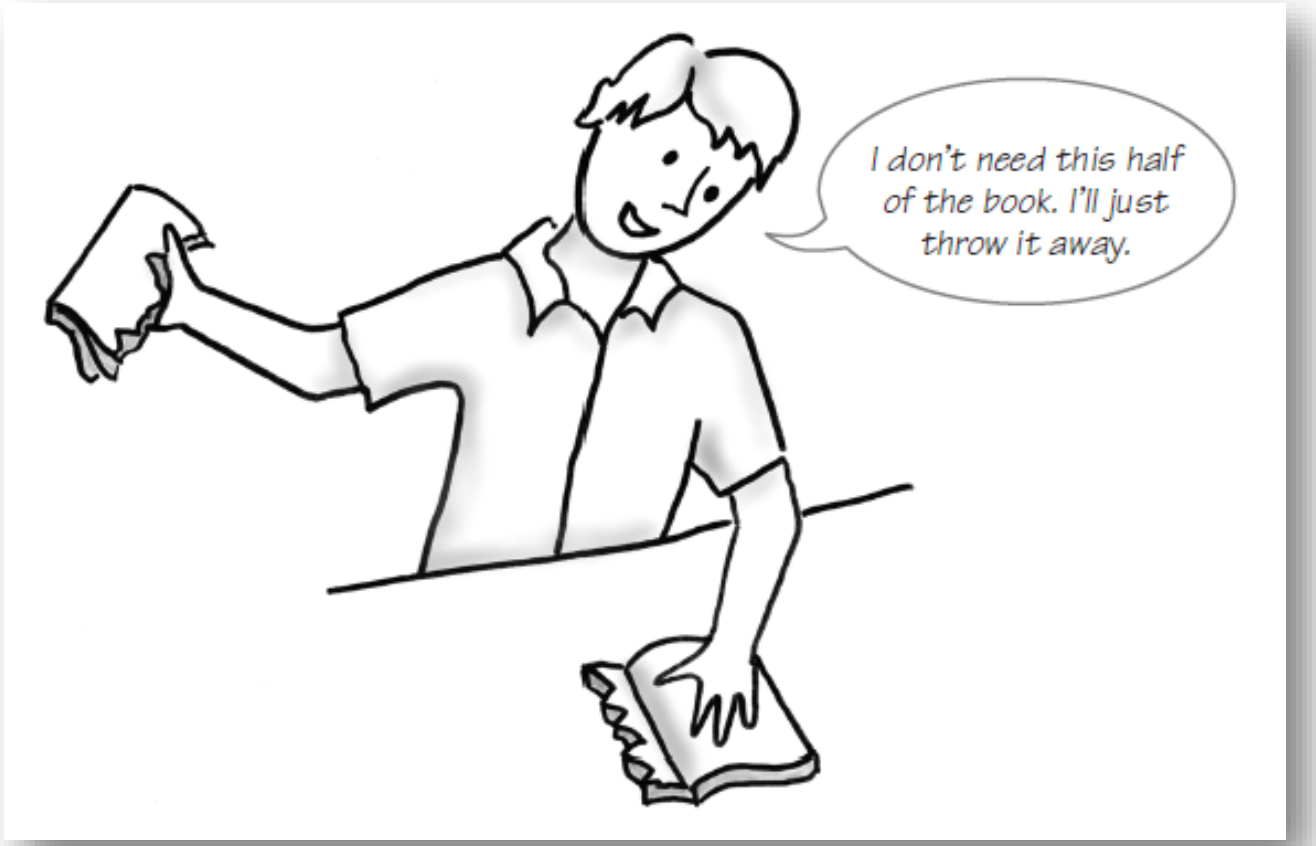
- Total is n + (n-1) = 2n-1 → O(n)
- Since we did n add() operations overall, our amortized time is O(n)/n = O(1), a constant
- Recall that when increasing by 1 we had $O(n^2)$ overall for the sequence, which gives us O(n) in amortized time
  - Note how much better our performance is when we double the array size
- Ok, that one was a bit complicated
  - Had a good deal of math in it
  - But that is what algorithm analysis is all about
  - If you can do some math you can save yourself some programming!

Sequential Search of a Sorted Array

- Coins sorted by their mint dates
- Note: sequential search can be
- more efficient if the data is sorted

Binary Search of a Sorted Array

Ignoring one half of the data
when the data is sorted



Binary Search of a Sorted Array

First draft of an algorithm for a binary search of an array

```
Algorithm to search a[0] through a[n - 1] for desiredItem
mid = approximate midpoint between 0 and n - 1
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return the result of searching a[0] through a[mid - 1]
else if (desiredItem > a[mid])
   return the result of searching a[mid + 1] through a[n - 1]
```

Binary Search of a Sorted Array

Use parameters and make recursive calls look more like Java

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = approximate midpoint between first and last
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else if (desiredItem > a[mid])
   return binarySearch(a, mid + 1, last, desiredItem)
```

Binary Search of a Sorted Array

Refine the logic a bit, get a more complete algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that (b) does not find its target

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

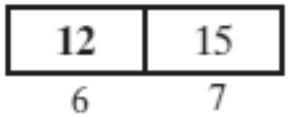
16 < 18, so search the left half of the array.

Look at the middle entry, 12:

Binary Search of a Sorted Array

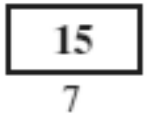
A recursive binary search of a sorted array that does not find its target

Look at the middle entry, 12:



16 > 12, so search the right half of the array.

Look at the middle entry, 15:



16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Binary Search of a Sorted Array

Implementation of the method **`binarySearch`**

```
private static <T extends Comparable<? super T>> boolean binarySearch(T[]
anArray, int first, int last, T desiredItem)
{
   boolean found;
   int mid = first + (last - first) / 2;

   if (first > last)
      found = false;
   else if (desiredItem.equals(anArray[mid]))
      found = true;
   else if (desiredItem.compareTo(anArray[mid]) < 0)
      found = binarySearch(anArray, first, mid - 1, desiredItem);
   else
      found = binarySearch(anArray, mid + 1, last, desiredItem);

   return found;
} // end binarySearch
```

Time complexity of Binary Search

- To simplify calculations we'll cheat a bit:
  1) Assume that the array is cut exactly in half with each iteration
     - In reality it may vary by one element either way
  2) Assume that the initial size of the array, N, is an exact power of 2, or 2^K for some K
     - In reality it can be any value
     - However, it will not affect our results
- Ok, so we have the following:
  - Initially: $N_0 = 2^K$
  - At iteration 1, $N_1 = N_0/2 =$ (in terms of K)
  - ...
  - Last iteration is when N = 1 = (in terms of K)

$$2^{K-1}$$

$$2^0$$

Time complexity of Binary Search

- We do one comparison (test) per iteration
- Thus we have a total of K+1 comparisons maximum
  - But $N = 2^K$
  - So $K = \lg_2 N$
  - Which makes $K + 1 = \lg_2 N + 1$
- This leads to our final answer of $O(\lg_2 N)$

Java Class Library

static method **`binarySearch`** with the specification:

```
/** Searches an entire array for a given item.
    @param array        An array sorted in ascending order.
    @param desiredItem  The item to be found in the array.
    @return  Index of the array entry that equals desiredItem;
             otherwise returns -belongsAt - 1, where belongsAt is
             the index of the array element that should contain
             desiredItem. */
public static int binarySearch(type[] array, type desiredItem);
```

Efficiency of a Binary Search of an Array

The time efficiency of a binary search of an array

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

Searching a Sorted Chain

- Similar to sequentially searching a sorted array.
- Implementation of **`contains`**.

```
public boolean contains(T anEntry)
{
   Node currentNode = firstNode;

   while ( (currentNode != null) &&
           (anEntry.compareTo(currentNode.getData()) > 0) )
   {
      currentNode = currentNode.getNextNode();
   } // end while

   return (currentNode != null) &&
          anEntry.equals(currentNode.getData());
} // end contains
```

Binary Search of a Sorted Chain

- To find the middle of the chain you must traverse the whole chain
- Then must traverse one of the halves to find the middle of that half

- Hard to implement
- Less efficient than sequential search!

Choosing between Sequential Search and Binary Search

The time efficiency of searching, expressed in Big Oh notation

	Best Case	Average Case	Worst Case
Sequential search (unsorted data)	O(1)	O(n)	O(n)
Sequential search (sorted data)	O(1)	O(n)	O(n)
Binary search (sorted array)	O(1)	O(log n)	O(log n)

Choosing between Iterative Search and Recursive Search

- Can save some time and space by using iterative version of a search
- Using recursion will not require much additional space for the recursive calls
- Coding binary search recursively is somewhat easier