Computer Science I

Recursion

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

1. Introduction
2. Designing Recursive Functions
3. Avoiding Recursion

Part I: Introduction

Challenge

Challenge: write code to count down from 10 to 1 *without* using a loop.

Challenge


```
1  void countDown(int n) {
2
3    if(n < 0) {
4      printf("Error: cannot count down from negatives!");
5    } else if(n == 0) {
6      printf("Blast Off\n");
7    } else {
8      printf("%d\n", n);
9      countDown(n-1);
10   }
11   return;
12 }
```

Recursion

- *Recursion* is when something is defined in terms of itself

Recursion

- *Recursion* is when something is defined in terms of itself
- Mathematics: recurrence relations, Fibonacci Sequence

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

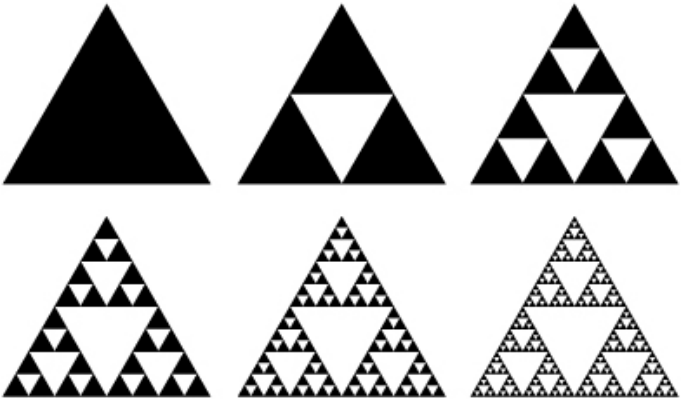
Recursion

- *Recursion* is when something is defined in terms of itself
- Mathematics: recurrence relations, Fibonacci Sequence

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

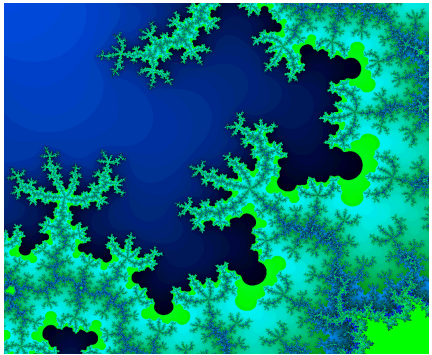
$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55 \tag{1}$$

Fractals



Serpinski Triangles

Fractals



Mandelbrot Set

Recursive Functions

- A *recursive function* is a function that makes one or more "recursive calls" to itself

Recursive Functions

- A *recursive function* is a function that makes one or more "recursive calls" to itself
- Generally recursive calls pass in different parameter values

Recursive Functions

- A *recursive function* is a function that makes one or more "recursive calls" to itself
- Generally recursive calls pass in different parameter values
- More generally: divide-and-conquer style problem solving

Part II: Writing Recursive Functions

Recursion

In general, every recursive function must have:

1. A *base case* – a condition after which the recursion stops

Recursion

In general, every recursive function must have:

1. A *base case* – a condition after which the recursion stops
2. Each recursive call must make progress toward the base case

Recursion

In general, every recursive function must have:

1. A *base case* – a condition after which the recursion stops
2. Each recursive call must make progress toward the base case
3. Corner cases may need to be handled separately

Thinking Recursively

- A recursive solution requires you to think about a general "case"

Thinking Recursively

- A recursive solution requires you to think about a general "case"
- Similar to loops: at the i-th iteration what do you do?

Thinking Recursively

- A recursive solution requires you to think about a general "case"
- Similar to loops: at the i-th iteration what do you do?
- Given the input, how do you *divide-and-conquer* it?

Examples

- Write a recursive function to compute the fibonacci sequence

Examples

- Write a recursive function to compute the fibonacci sequence
- Write a recursive function to find the largest element in an array of integers (simulate a traditional loop)

Examples

- Write a recursive function to compute the fibonacci sequence
- Write a recursive function to find the largest element in an array of integers (simulate a traditional loop)
- Write a recursive, *divide-and-conquer*-style function to sum the elements in an array of integers

Fibonacci Solution


```
1  int fibonacci(int n) {
2
3    if(n < 1) {
4      return -1;
5    } else if(n == 1 || n == 2) {
6      return 1;
7    } else {
8      return fibonacci(n-1) + fibonacci(n-2);
9    }
10
11 }
```

Largest Element Solution


```
1  int largestElement(const int *arr, int largest, int index) {
2    if(index < 0) {
3      return largest;
4    } else {
5      if(arr[index] > largest) {
6        return largestElement(arr, arr[index], index-1);
7      } else {
8        return largestElement(arr, largest, index-1);
9      }
10   }
11 }
```

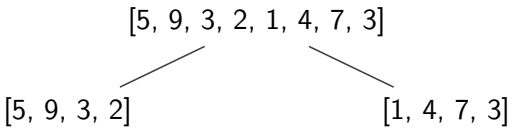
Summation Solution


```
1  int recursiveSum(const int *arr, int l, int r) {
2    if(l > r) {
3      return 0;
4    } else if(l == r) {
5      return arr[l];
6    } else {
7      int m = (l + r) / 2;
8      return recursiveSum(arr, l, m) + recursiveSum(arr, m+1, r);
9    }
10 }
```

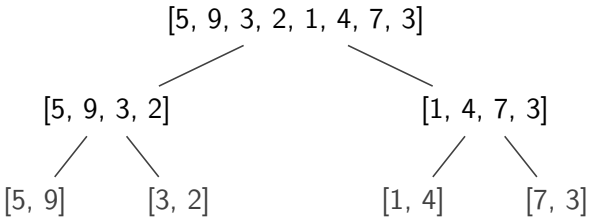
Divide & Conquer

[5, 9, 3, 2, 1, 4, 7, 3]

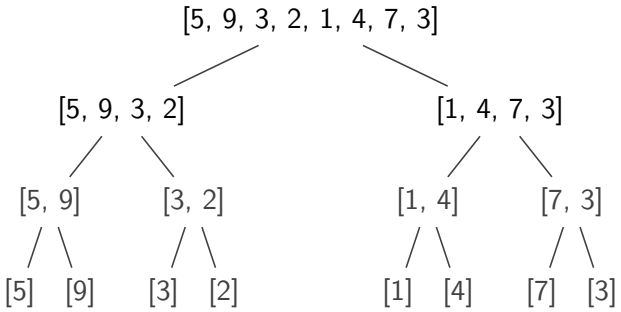
Divide & Conquer



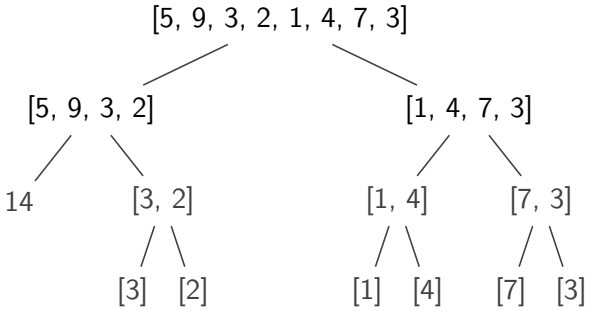
Divide & Conquer



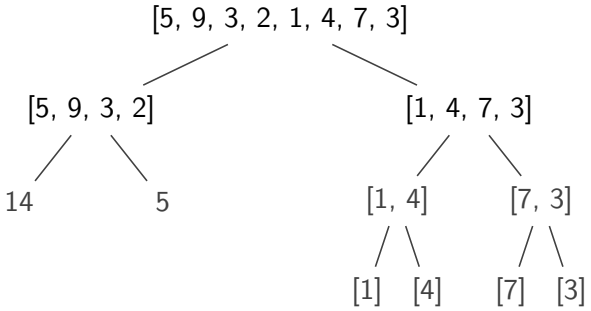
Divide & Conquer



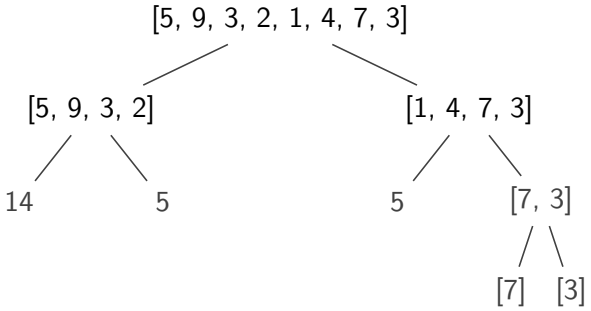
Divide & Conquer



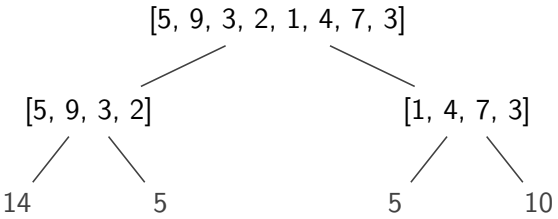
Divide & Conquer



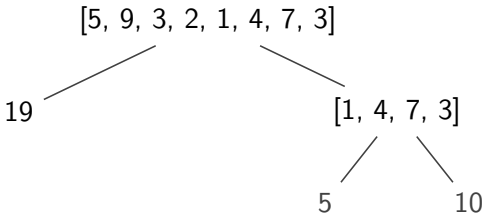
Divide & Conquer



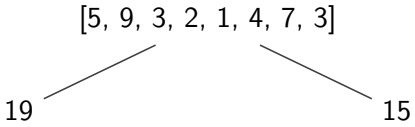
Divide & Conquer



Divide & Conquer



Divide & Conquer



Divide & Conquer

Part III: Avoiding Recursion

Recursion: Advantages

- Recursion can be a useful problem solving technique

Recursion: Advantages

- Recursion can be a useful problem solving technique
- Divide & Conquer strategies are generally presented as recursive

Recursion: Advantages

- Recursion can be a useful problem solving technique
- Divide & Conquer strategies are generally presented as recursive
- Functional programming languages (Lisp, Haskell) use recursion as a fundamental control flow mechanism

Recursion: Disadvantages

- In practice, recursion is not necessary: any (computable) recursive function can be rewritten using a loop and/or smart data structure

Recursion: Disadvantages

- In practice, recursion is not necessary: any (computable) recursive function can be rewritten using a loop and/or smart data structure
- Many style guides discourage or forbid the use of recursion

Recursion: Disadvantages

- In practice, recursion is not necessary: any (computable) recursive function can be rewritten using a loop and/or smart data structure
- Many style guides discourage or forbid the use of recursion
- Recursive functions can "abuse" the program stack by creating and destroying many stack frames

Recursion: Disadvantages

- In practice, recursion is not necessary: any (computable) recursive function can be rewritten using a loop and/or smart data structure
- Many style guides discourage or forbid the use of recursion
- Recursive functions can "abuse" the program stack by creating and destroying many stack frames
- Deep recursion risks *stack overflow*

Recursion: Disadvantages

- In practice, recursion is not necessary: any (computable) recursive function can be rewritten using a loop and/or smart data structure
- Many style guides discourage or forbid the use of recursion
- Recursive functions can "abuse" the program stack by creating and destroying many stack frames
- Deep recursion risks *stack overflow*
- Demonstration

Recursion: Disadvantages

- Recursion can be extremely inefficient when not done properly

Recursion: Disadvantages

- Recursion can be extremely inefficient when not done properly
- Perfect example: Fibonacci code

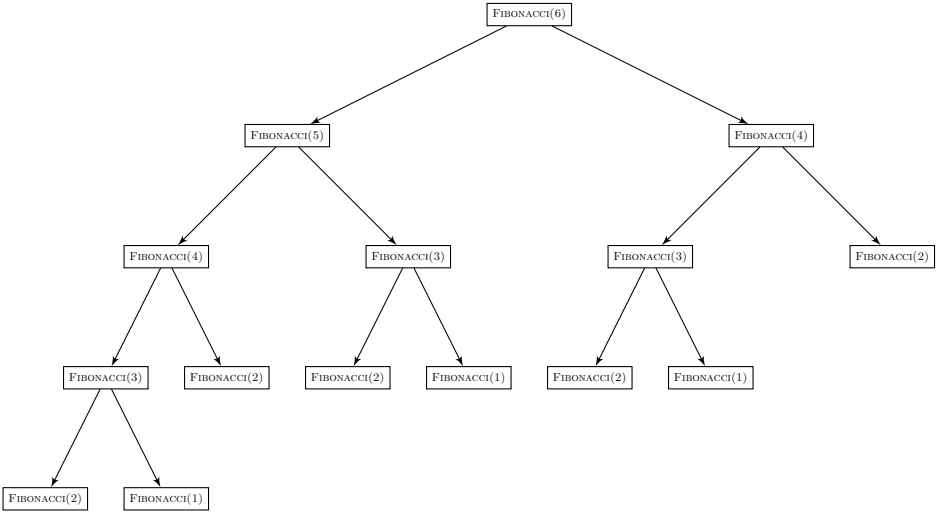
Recursion: Disadvantages

- Recursion can be extremely inefficient when not done properly
- Perfect example: Fibonacci code
- The same computations are performed over and over multiple times

Recursion: Disadvantages

- Recursion can be extremely inefficient when not done properly
- Perfect example: Fibonacci code
- The same computations are performed over and over multiple times
- Computation Tree

Computation Tree



Mitigating Problems with Recursion

- Alternative solution: *memoization*

Mitigating Problems with Recursion

- Alternative solution: *memoization*
- Cache values so they can be reused (and not recomputed)

Mitigating Problems with Recursion

- Alternative solution: *memoization*
- Cache values so they can be reused (and not recomputed)
- Each recursive call checks to see if the value has been computed

Mitigating Problems with Recursion

- Alternative solution: *memoization*
- Cache values so they can be reused (and not recomputed)
- Each recursive call checks to see if the value has been computed
- If yes: use it (avoid further recursion)

Mitigating Problems with Recursion

- Alternative solution: *memoization*
- Cache values so they can be reused (and not recomputed)
- Each recursive call checks to see if the value has been computed
- If yes: use it (avoid further recursion)
- If no: pay for the recursion, but store the answer

Mitigating Problems with Recursion

- Alternative solution: *memoization*
- Cache values so they can be reused (and not recomputed)
- Each recursive call checks to see if the value has been computed
- If yes: use it (avoid further recursion)
- If no: pay for the recursion, but store the answer
- Demonstration