Recursion

A New Tool in Writing Algorithms

An **algorithm** is a sequence of steps for solving a problem.

So far, we have a particular toolkit:

- conditionals
- iteration
- other method calls

A New Tool in Writing Algorithms

A **recursive algorithm** is an algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.

Recursive Grading Procedure

How do I grade exams?

- If I have one exam to grade, I grade it.
- Otherwise, I grade 50% of the exams, then the other 50% of the exams.

```
Grading 8 Exams
    Grading 4 Exams                                   Grading 4 Exams
        Grading 2 Exams          Grading 2 Exams          Grading 2 Exams          Grading 2 Exams
            Grading 1 Exam            Grading 1 Exam            Grading 1 Exam            Grading 1 Exam
            Grading 1 Exam            Grading 1 Exam            Grading 1 Exam            Grading 1 Exam
```

Recursion Stops Somewhere

Recursive algorithms eventually have to actually do some computation step instead of just making more recurisve calls.

The **base case** is the case where a recursive algorithm actually does some final work--grading the one exam in the previous case.

Recursive Methods

Methods can call other methods, including the method itself.

```
public static void countDown(int countInt) {
      if (countInt <= 0) {
         System.out.println("GO!");
      }
      else {
         System.out.println(countInt);
         countDown(countInt - 1);
      }
}
```

ZyBook animation 17.2

Practice, Practice, Practice: Largest

Return the largest number in an array of numbers.

```
public int largest(int[] numbers, int index) {
  if <<Missing base case>>
    return numbers[0];
  return Math.max(numbers[index], largest(numbers, index-1));
}
```

`largest({2, 4, 8}, 2) -> 8`, for example

Practice, Practice, Practice: Largest

Return the largest number in an array of numbers.

```
public int largest(int[] numbers, int index) {
  if (index == 0) {
    return numbers[0];
  }
  return Math.max(numbers[index], largest(numbers, index-1));
}
```

`largest({2, 4, 8}, 2) -> 8`, for example

Practice, Practice, Practice: Multiply

Multiply `x * y`

```
public int multiply(int x, int y) {
  if <<Missing base case condition>> {
    <<Missing base case action>>
  } else {
    return multiply(x - 1, y) + y;
  }
}
```

Practice, Practice, Practice: Multiply

```java
public int multiply(int x, int y) {
  if (x == 1) {
    return y;
  } else {
    return multiply(x - 1, y) + y;
  }
}
```

Can you think of another base case?

Practice, Practice, Practice: Multiply

```java
public int multiply(int x, int y) {
  if (x == 0) {
    return 0;
  } else {
    return multiply(x - 1, y) + y;
  }
}
```

Practice, Practice, Practice: GCD

The greatest common divisor (GCD) for a pair of numbers is the largest positive integer that divides both numbers without remainder.

Two helpful facts: `GCD(x, 0) = x` and `GCD(x, y) = GCD(y, x % y)`

```
public int GCD(int x, int y) {
  if <<Missing base case condition>> {
    <<Missing base case action>>
  } else {
    return GCD(y, x % y);
  }
}
```

Example: `GCD(6, 4) --> 2`

Practice, Practice, Practice: GCD

The greatest common divisor (GCD) for a pair of numbers is the largest positive integer that divides both numbers without remainder.

Two helpful facts: `GCD(x, 0) = x` and `GCD(x, y) = GCD(y, x % y)`

```
public int GCD(int x, int y) {
  if (y == 0) {
    return x;
  } else {
    return GCD(y, x % y);
  }
}
```

Example: `GCD(6, 4) --> 2`

Practice, Practice, Practice: SumToK

Sum all values from 1 to k, e.g. sumToK(5) -> 15

```
public int sumToK(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return <<Missing Recursive case action>>
  }
}
```

Practice, Practice, Practice: `SumToK`

Sum all values from `1` to `k`, e.g. `sumToK(5) -> 15`

```
public int sumToK(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return k + sumToK(k - 1);
  }
}
```

Practice, Practice, Practice: `SumToK`

Sum all values from `1` to `k`, e.g. `sumToK(5) -> 15`

```
public int sumToK(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return <<Missing Recursive case action>>
  }
}
```

Practice, Practice, Practice: `SumToK`

Sum all values from `1` to `k`, e.g. `sumToK(5) -> 15`

```
public int sumToK(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return k + sumToK(k - 1);
  }
}
```

Practice, Practice, Practice: `countChr`

Return the number of times `'A'` appears in a given String.

```
public int countChr(String str) {
  if (str.length() == 0) {
    return 0;
  }
  int count = 0;
  if (str.substring(0, 1).equals("A")) {
    count = 1;
  }
  return count + <<Missing a Recursive call>>
}
```

`countChr("ctcowcAt") -> 1`

Practice, Practice, Practice: `countChr`

Return the number of times `'A'` appears in a given String.

```java
public int countChr(String str) {
  if (str.length() == 0) {
    return 0;
  }
  int count = 0;
  if (str.substring(0, 1).equals("A")) {
    count = 1;
  }
  return count + countChr(str.substring(1));
}
```

`countChr("ctcowcAt") -> 1`

Binary Search

- Used to search for a value (the *target*) in a **sorted array**.
- Keep dividing the array in half
- Compare the target with the value at the middle index in the remaining array.
- If the target is less than the mdidle element, then we search the target in the **left half of the array** (the elements less than the middle)
- If the target is greater than the mdidle element, then we search the target in the **right half of the array** (the elements greater than the middle)

Binary Search

- returns the position of the middle element if we find the target there, or
- returns -1 if we can't find the target.

Binary Search: Live Coding

Binary Search

```
public static int binarySearch(String[] A, String target, int leftBound, int rightBound)
{
    if (leftBound > rightBound) {
        return -1
    }
    int middleIdx = (leftBound + rightBound) / 2;
    String middleElem = A[middleIdx];
    if (middleElem.equals(target)) {
        return middleIdx;
    } else if (middleElem.compareTo(target) < 0) {
        return binarySearch(A, target, middleIdx + 1, rightBound);
    } else {
        return binarySearch(A, target, leftBound, middleIdx - 1);
    }
}
```

Writing Your Own Recursive Methods

Step 1: Write the base case.

- (A way to return a value without recursing further.)

Step 2: Write the recursive case.

It really is this simple, but the type of thinking that lets you accomplish this will often take a while to learn.

Writing Your Own Recursive Methods

Tips:

- Make sure that your base case is reachable
    - Your recursive calls should make the problem progressively smaller, typically.
- Consider whether or not the problem is helped with a recursive approach
    - Fibonacci numbers are technically recursive, but the recursive implementation is very bad...