14-0: Recursion Review

- Factorial from visualization
14-1: Recursion – Power

• Write a method power
  int power(int x, int n)
  • Return $x^n$

• What is the base case?
• How can we make the problem smaller?
• How can we use the solution to the smaller problem to solve the original problem?
int power(int x, int n)

• What is the base case?
  • What is a version of the problem that is easy, that we can solve immediately?
int power(int x, int n)

- What is the base case?
  - Raising a number to the 1st power is easy
  - \( x^1 = x \)

```c
int power(int x, int n)
{
    if (n == 1)
    {
        return x;
    }
}
```
int power(int x, int n)

• What is the recursive case?
  • How do we make the problem smaller?
  • How do we use the solution to the smaller problem to solve the original problem?
Recursion – Power

```c
int power(int x, int n)
{
    • What is the recursive case?
    • How do we make the problem smaller?
      • $x^{n-1}$ is a smaller problem than $x^n$
    • How do we use the solution to the smaller problem to solve the original problem?
      • $x^n = x^{n-1} \cdot x$
}
```
int power(int x, int n)
{
    if (n == 1)
    {
        return x;
    }
    else
    {
        return x * power(x, n - 1);
    }
}

• What about $x^0$?
int power(int x, int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return x * power(x, n - 1);
    }
}
What happens if we forget the base case?

```c
int power(int x, int n)
{
    return x * power(x, n - 1);
}
```
What happens if we don’t make progress towards the base case?

```c
int power(int x, int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return power(x, n);
    }
}
```
14-10: Recursion – Reversing Digits

- Function that takes as input an integer
- Writes out the digits in reverse order

```c
void printReversed(int n)
{
    ...
}
```
14-11: Recursion – Reversing Digits

- What’s a easy number to print reversed?

```c
void printReversed(int n)
{
    ...
}
```
What’s a easy number to print reversed?

```java
void printReversed(int n)
{
    if (n < 10)
    {
        System.out.println(n);
    }
    ...
}
```
• How can we make the problem smaller
• We have to make the problem smaller such that a solution to the smaller problem helps us solve the original problem
How can we make the problem smaller
  • Remove the last digit (dividing by 10)
  • How can this help?
void printReversed(int n)
{
    if (n < 10)
    {
        System.out.println(n);
    }
    else
    {
        System.out.print(n % 10);
        printReversed(n / 10);
    }
}
A method is *tail recursive* if no more work needs to be done after the recursive call.

- We return the value of the recursive call unchanged.
- None of the functions that we have seen so far have been tail-recursive.
Tail Recursion

- Typically, when a function is tail recursive, we have an extra parameter
  - Extra parameter builds up the solution to the problem
  - Each recursive call adds to the solution
  - Base case returns this solution
  - Solution is returned all the way to the end
int factorialTR(int n, int result)
{
    if (n == 0)
        return result;
    return factorialTR(n - 1, result * n);
}

int factorial(int n)
{
    return factorialTR(n, 1);
}
Tail Recursion

- Tail recursion is a little easier to see with reversing a string:
  - Start with an empty result
  - Remove first character from input, push it on to result
  - repeat until the input is empty
public static String reverseTR(String s, String reversed)
{
    if (s.length() == 0)
        return reversed;
    return reverseTR(s.substring(1), s.charAt(0) + reversed);
}

public static String reverse(String s)
{
    return reverseTR(s, "");
}
Why is tail recursion useful?

“Standard” recursive functions require an activation record on the stack for each recursive call

- We need to do some work after the recursive call is done
- We need the information stored on the stack
- Examples: factorial / reversing
Why is tail recursion useful?

Tail recursive functions don’t need to maintain the activation record after the function is called:
- Just return the value returned by the recursive call
- We could reuse the same activation record
- Could even change the recursive call to a loop (scheme)
Searching a String

- Anywhere you use a loop, you could use recursion instead (and vice-versa)
  - Though there are some problems that are easier to solve recursively, and some that are easier to solve iteratively
- Create a recursive function countLetters, which takes as input a String s and a character c, and returns the number of times that c occurs in s.

```java
int occurs(String s, char c)
```
public static int occurances(String s, char c)
{
    if (s.equals(""))
    {
        return 0;
    }
    else if (s.charAt(0) == c)
    {
        return 1 + occurances(s.substring(1), c);
    }
    else
    {
        return occurances(s.substring(1), c);
    }
}
Searching an array

- Previous string searching code called substring over and over
  - A little inefficient, creating lots of new strings
- For searching a list, we *really* don’t want to make extra copies
- Instead, we will write a function that searches a *range* of indices in a list, instead of an entire list
boolean search(int A[], int elem, int lowIndex,

• return true if elem is in the list, between lowIndex and highIndex (inclusive)

• First up: Iterative solution
public static boolean find(int A[], int elem, int lowIndex, int highIndex) {
    for (int i = lowIndex; i <= highIndex; i++) {
        if (A[i] == elem)
            return true;
    }
    return false;
}
14-28: Searching an array

boolean search(int A[], int elem, int lowIndex, int highIndex)

• return true if elem is in the list, between lowIndex and highIndex (inclusive)
• Next up: Recursive solution.
public static boolean findR(int A[], int elem, int lowIndex, int highIndex) {
    if (lowIndex > highIndex) {
        return false;
    }
    else if (A[lowIndex] == elem) {
        return true;
    }
    else {
        return findR(A, elem, lowIndex+1, highIndex);
    }
}
14-30: Binary Search

- We have a sorted list of integers
- Want to determine if a given integer is in the list
- We could do a linear search (start from beginning, search to the end)
- Is there a better way?
Binary Search

- If we are looking for an element in an empty list, return false.
- If the element in the center of the list is what we are looking for, return true.
- If the element in the center of the list is less than what we are looking for, discard the left half of the list, continue looking.
- If the element in the center of the list is greater than what we are looking for, discard the right half of the list, continue looking.
• As before, actually throwing away half the list (creating a new list half as large) is not efficient

• We can have our binary search take as input parameters the range in which we are searching

• `boolean search(int A[], int lowIndex, int highIndex)`
public static boolean search(int A[], int elem, int lowIndex, int highIndex) {
    if (lowIndex > highIndex)
    {
        return false;
    }
    int midIndex = (lowIndex + highIndex) / 2;
    if (A[midIndex] == elem)
        return true;
    else if (A[midIndex] < elem)
        return search(A, elem, lowIndex, midIndex - 1);
    else
        return search(A, elem, midIndex + 1, highIndex);
}
Our implementation of Binary Search is tail recursive.

How could we modify binary search to return the *index* of the element in the array, if it exists, or -1 if it does not?
public static int search(int A[], int elem, int lowIndex, int highIndex) {
    if (lowIndex > highIndex) {
        return -1;
    }
    int midIndex = (lowIndex + highIndex) / 2;
    if (A[midIndex] == elem) {
        return midIndex;
    } else if (A[midIndex] < elem) {
        return search(A, elem, lowIndex, midIndex - 1);
    } else {
        return search(A, elem, midIndex + 1, highIndex);
    }
}
Hands On

- Write a recursive version of toUpperCase
  - String toUpperCase(String input)
  - Easier: Use Character.toUpperCase
  - Harder: Use casting of char to int (and back again)

- write a version of occurences that does not create any extra strings (you may add parameters)