Nick Troccoli
CS 106X

Section #2
Week 3

Section Handout #2: Recursion

Based on handouts by various current and past CS106B/X instructors and TAs.

Extra practice problems: CodeStepByStep – Sequence, Hanoi; Textbook – chapter 7 exercises

We'll be getting some practice with approaching problems recursively during this week's section. To start off, recall that this is our checklist for writing recursive functions from lecture. Please put it to use when writing recursive code on this handout (problems #3-7)!

Recursion Checklist

- ❒ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❒ **Find your base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❒ **Find your recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❒ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

---

1. Recursion code trace *(recursion, tracing)*

For each call to the following function, indicate what value is returned.

```
void mysteryTrace(int x, int y) {
    if (y == 1) {
        cout << x;
    } else {
        cout << (x * y) << ", ";
        mysteryTrace(x, y - 1);
        cout << ", " << (x * y);
    }
}
```

```
// Call                Output

mysteryTrace(4, 1)     ____________________

mysteryTrace(8, 2)     ____________________

mysteryTrace(3, 4)     ____________________
```

2. Debugging recursion *(recursion, debugging)*

The following function recursively finds the maximum integer in a `Vector` between two indices (inclusive) by taking the maximum from the left half, the maximum from the right half, and then returning the max of those two. For example, if a `Vector` variable named *vec* contained the values **`{1, 2, 4, 2, 3, 5}`**, the call of **`recursiveMax(vec, 0, 2)`** looks at the elements in indices 0, 1, and 2, and returns 4 because it is the largest number in those indices. What is the bug in this code?

```
int recursiveMax(const Vector<int>& v, int left, int right) {
    if (left == right) {
        return v[left];
    } else if (left < right) {
        int middle = (left + right) / 2;
        int leftMax = recursiveMax(v, left, middle);
        int rightMax = recursiveMax(v, middle, right);
        if (leftMax > rightMax) {
            return leftMax;
        } else {
            return rightMax;
        }
    } else {
        throw "Invalid range.";
    }
}
```

3. cannonballs *(recursion)*

You are a pirate sailing the seven seas, and you would like to know how many cannonballs you have stored on deck. You also happen to be learning about recursion. Write a recursive function named **`cannonballs`** that returns the number of cannonballs in a square pyramid of height *n*, just like how you store them on deck. For example, in a square pyramid of height 3, the bottom layer has 9 cannonballs, the middle layer has 4, and there is one cannonball on top, so `cannonballs(3)` returns 14. If passed a negative number, your function should throw the invalid parameter as an int exception.

Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

4. reverseString *(recursion, strings)*

Write a recursive function **`reverse`** that takes in a string *s* and returns a string with the same characters in reverse order. For example, `reverse("Hi, you!")` returns `"!uoy ,iH"`. Don't modify the original string.

Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

Bonus: What data structure might be helpful if you were to solve this problem iteratively?

5. doubleStack *(recursion, Stack)*

Write a recursive function named **`doubleStack`** that takes a reference to a `Stack<int>` called *s* and replaces each integer with two consecutive copies of that integer. For example, if *s* stores `{1, 2, 3}`, then `doubleStack(s)` changes it to `{1, 1, 2, 2, 3, 3}`.

Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

6. combinations *(recursion)*

Write a recursive function named **combinations** that accepts integers *n* and *k* and returns "*n* choose *k*," which is the number of unique combinations of *k* values taken from *n* values. This can be expressed using one of these formulas:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ or } \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

Bonus: memoization. You can speed up a recursive function by caching previously calculated or returned values in a structure such as a map, vector, etc. This is called *memoization*. Fortunately, it often does not require much additional code beyond adding an extra base case and managing the cache when recursing. Modify your solution to this problem to memoize it.

Bonus Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

7. isSubsequence *(recursion)*

Write a recursive function named `isSubsequence` that takes two strings and returns `true` if the second string is a subsequence of the first string and `false` otherwise. A string is a subsequence of another if it contains the same letters in the same order, but not necessarily consecutively. You can assume both strings are already entirely lowercase. For example:

```
isSubsequence(“computer”, “core”)        false
isSubsequence(“computer, “cope”)         true
isSubsequence(“computer”, “computer”)    true
```

Checklist: ❑ Inputs/outputs/wrapper? ❑ Base case(s) ❑ Recursive step(s) ❑ Handles all input

8. reverseMap *(Map, extra practice)*

Write a function named reverse that takes a `Map` from ints to strings and returns a `Map` with the associations reversed. For example, if a `Map` variable named *map* stores `{1:"a", 2:"b", 3:"c"}`, the call of `reverse(map)` should return `{"a":1, "b":2, "c":3}`. If there are duplicate values (k1, v) and (k2, v) in the original map, your returned map may contain either (v, k1) or (v, k2).