CS106X
Winter 2017

Chris Gregg

Section Handout #2

This week has more practice with data structures, in particular Stacks, Queues, Sets, and Maps, as well as some practice recursion problems. When writing a recursive solution, try to make it as elegant as possible. Avoid redundant cases and if statements. In addition, think about what kinds of inputs are invalid for each problem, and the proper way to handle those invalid inputs.

1. Reorder
Write a function named reorder that takes a queue of integers that are already sorted by absolute value, and modifies it so that the integers are sorted normally. The only auxiliary data structure you can use is a single `Stack<int>`. For example, passing the queue `{1, -2, 3, 4, -5, -6, 7}` changes it to `{-6, -5, -2, 1, 3, 4, 7}`.

2. Stacks or Queues?
Write the following methods:

```
void push(Queue<int> &q, int entry)
int pop(Queue<int>& q)
```

These methods should allow someone to use a `Queue<int>` as though it were a `Stack<int>`, pushing and popping as though it were a LIFO structure. For an added challenge, write a solution that doesn't create any additional data structures.

3. Twice
Write a function named `twice` that takes a vector of integers and returns a set containing all of the numbers in the vector that appear exactly twice. You can only use Sets as auxilary storage. For example, passing `{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}` returns `{3, 7}`.

4. Friend List
Write a function named `friendList` that takes in a file name, reads friend relationships from a file, and writes them to a map. You should return the populated map. Friendships are bi-directional; if Chris is friends with Aaron, Aaron is friends with Chris. The file contains one friend relationship per line. The names are separated by a single space. You don't have to worry about malformed entries (assume all entries are formatted correctly).

If an input file named `buddies.txt` looked like this:

```
Anupama Chris
Chris Jason
```

Then the call of `friendList("buddies.txt")` should return a resulting map that looks like this:

```
{"Anupama": {"Chris"}, "Jason": {"Chris"}, "Chris": {"Anupama", "Jason"}}
```

5. Tracing a Mystery
For each call to the following method, indicate what value is returned.

Thanks to Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106B and X instructors for contributing content on this handout. Thanks to Wesley Rodriguez and Anupama Rajan for proofreading.

```
void mystery1(int x, int y) {
  if (y == 1) {
    cout << x;
  } else {
    cout << (x * y) << ", ";
    mystery1(x, y - 1);
    cout << ", " << (x * y);
  }
}
```

Call	Output
`mystery1(4, 1)`	
`mystery1(8, 2)`	
`mystery1(3, 4)`	

6. Sum of Squares
Write a recursive function named `sumOfSquares` that takes in an integer `n` returns the sum of squares from 1 to `n` inclusive. For example, `sumOfSquares(3)` should return 14 (because $1^2 + 2^2 + 3^2 = 14$). You can assume `n` ≥ 1.

7. Reverse
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"`. You shouldn't modify the original string.

8. Star String
Write a recursive function named `starString` that takes in an integer `n` and returns a string of 2^n asterisks. For example,

```
starString(1)                    "**"
starString(2)                    "****"
starString(4)                    "**********////" // 16 stars
```

What should your function do if `n` is negative? How many recursive calls does your function end up making (as a function of `n`)?

9. Stutter Stack
Write a recursive function named `stutterStack` that takes in a reference to a stack of integers and replaces each integer with two copies of that integer. For example, if a stack `s` stores `{1, 2, 3}`, then `stutterStack(s)` changes it to `{1, 1, 2, 2, 3, 3}`.

10. Subsequence
Write a recursive function named `isSubsequence` that takes two strings and returns true if the second string is a subsequence of the first string. 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 all lowercase characters. For example,

```
isSubsequence("computer", "core")            false

isSubsequence("computer", "cope")            true

isSubsequence("computer", "computer")        true
```

11. Tower of Hanoi

The Tower of Hanoi is a game where you have three pegs arranged side-by-side (#1, #2, and #3) and *n* circular discs of different sizes that slide onto the pegs. All of the discs start on one peg in increasing size order (the largest on the bottom).

The goal is to move all the discs from one peg to another by following these rules:

1. You may only move one disk at a time from peg to peg.
2. No disk may be placed on top of a smaller disk

Write a function named `hanoi` that takes in the number of disks, a source peg, and a destination peg, all as integers. The function should print the solution for a game where those disks start on the source peg and end on the destination peg. For example, `hanoi(3, 1, 3)` should print:

```
move disk 1 from peg 1 to peg 3
move disk 2 from peg 1 to peg 2
move disk 1 from peg 3 to peg 2
move disk 3 from peg 1 to peg 3
move disk 1 from peg 2 to peg 1
move disk 2 from peg 2 to peg 3
move disk 1 from peg 1 to peg 3
```

You might find it helpful to play through an example or two to figure out how to win the game and what stategies you might use. Note that in this problem, there can be any number of disks, but there will always be 3 pegs.

12. Edit Distance

Write a recursive function named `editDistance` that accepts two string parameters and returns the edit distance between those two strings as an integer. The edit distance (also called the *Levenshtein distance*) is defined as the minimum number of changes that are required to change one string to another. A change can be defined as one of three operations:

1. inserting a character (e.g. turning "CS106" into "CS106X")
2. deleting a character (e.g. turning "Colin" into "Coin")
3. changing a character (e.g. turning "Leslie" into "Weslie")

For example,

```
editDistance("driving", "diving")            1 // delete r

editDistance("debate", "irate")              3 // delete d, change e to i, change b to r

editDistance("football", "cookies")          6 // work through this one on your own!
```

Figuring out which sequence of changes gives you the minimum number of changes will require exploring multiple possible sequences. This is a sneak peek at exhaustive recursion, a technique we'll be talking more about next week.