CS106B
Winter 2017

Handout #25
March 13, 2017

Section Handout 9

This final section handout consists of a bunch of cumulative review problems from throughout the quarter. Feel free to work on whatever seems interesting or useful!

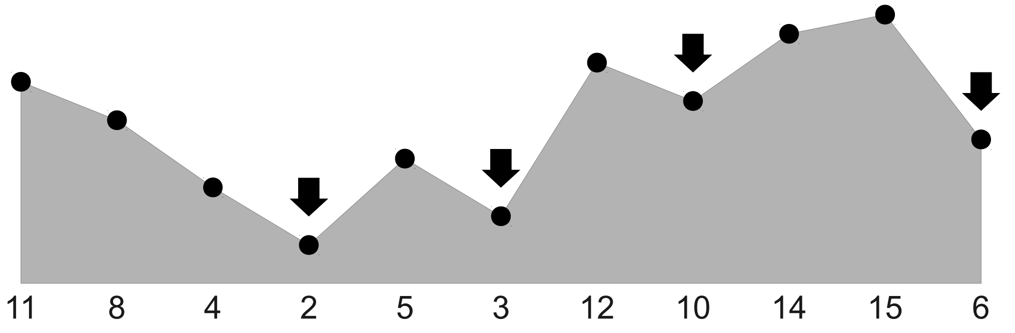
In a break from tradition, we have ***not*** compiled solutions for these problems. If you're unsure how to solve any of these problems, ask your section leader for input or stop by the CLaIR for assistance!

Week One: Basic Recursion and String Processing

1. Write a function that reverses a string "in-place." That is, you should take the string to reverse as a reference parameter and modify it so that it ends up holding its reverse. Your function should use only O(1) auxiliary space.
2. Imagine you have a string containing a bunch of words from a sentence. Here's a nifty little algorithm for reversing the order of the words in the sentence: reverse each individual string in the sentence, then reverse the entire resulting string. (Try it – it works!) Go and code this up in a way that uses only O(1) auxiliary storage space.
3. The ***binary search*** algorithm is a fast algorithm for finding an element in a sorted array. It works like this. First, look in the middle of the array. If that element is the one you're looking for, you're done! Otherwise, if the element in the middle is *bigger* than the element you're looking for, since the array is sorted, if the element is anywhere at all, it's in the first half of the array, so look there. Otherwise, look in the second half of the array. Oh, and if your array is empty, then the element you're looking for isn't there.

   Implement binary search, both recursively and iteratively. What's the runtime of your solution? Can you get it to work in worst-case time O(log *n*)?
4. Suppose that you are interested in setting up a collection point to funnel rainwater into a town's water supply. The town is next to a ridge, which for simplicity we will assume is represented as an array of the elevations of different points along the ridge.

   When rain falls on the ridge, it will roll downhill along the ridge. We'll call a point where water naturally accumulates (that is, a point lower than all neighboring points) a "good collection point." For example, here is a possible ridge with good collection points identified:

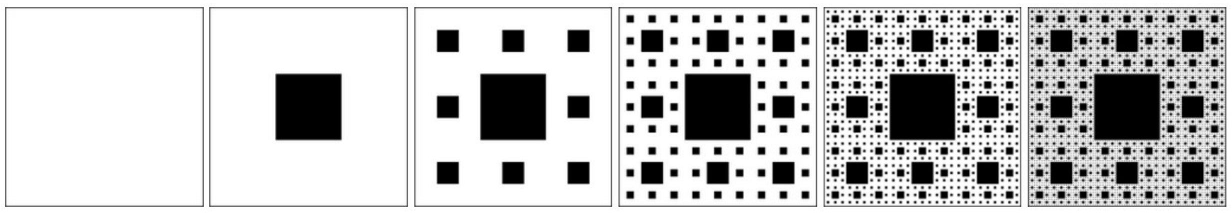
   Write a recursive function to find a good collection point. See if you can solve this with a solution that runs in time O(log *n*). As a hint, think about binary search. You can assume that all elements in the array are distinct.

Week Two: Container Classes

1. Write a function that, given a `Map<string, int>` associating string values with integers, produces a `Map<int, Set<string>>` that's essentially the reverse mapping, associating each integer value with the set of strings that map to it. (This is an old job interview question from 2010.)
2. How are `Map` and `HashMap` implemented internally? What's one advantage of `Map` over `HashMap`? One advantage of `HashMap` over `Map`?
3. A ***compound word*** is a word that can be cut into two smaller strings, each of which is itself a word. The words "keyhole" and "headhunter" are examples of compound words, and less obviously so is the word "question" ("quest" and "ion"). Write a function that takes in a Lexicon of all the words in English and then prints out all the compound words in the English language.

Week Three: Graphical Recursion and Recursive Problem-Solving

1. The ***Sierpinski carpet*** is a fractal image in the shape of a square. An order-0 Sierpinski carpet is just an empty square. An order-(n+1) Sierpinski carpet can be formed by subdividing a square into nine smaller squares in a 3 × 3 grid, filling the central square black, then recursively drawing order-n Sierpinski carpets in each of the remaining grid cells. Here are Sierpinski carpets of orders 0, 1, 2, 3, 4, and 5:



   Write a function to draw an order-n Sierpinski carpet in a given window.

2. Imagine you have a 2 × n grid that you'd like to cover using 2 × 1 dominoes. The dominoes need to be completely contained within the grid (so they can't hang over the sides), can't overlap, and have to be at 90° angles (so you can't have diagonal or tilted tiles). There's exactly one way to tile a 2 × 1 grid this way, exactly two ways to tile a 2 × 2 grid this way, and exactly three ways to tile a 2 × 3 grid this way (can you see what they are?) Write a recursive function that, given a number n, returns the number of ways you can tile a 2 × n grid with 2 × 1 dominoes.

Week Four: Recursive Enumeration

1. Given a positive integer n, write a function that finds all ways of writing n as a sum of nonzero natural numbers. For example, given n = 3, you'd list off these options:

   $$3 \qquad 2+1 \qquad 1+2 \qquad 1+1+1$$

2. Solve the previous problem assuming that order doesn't matter, so 1 + 2 and 2 + 1 would be treated identically. See if you can find a way to do this that doesn't generate the same option more than once.
3. Write a function that, given a list of distinct strings and a number k, lists off all ways of choosing k elements from that list, given that order *does* matter. For example, given the objects A, B, and C and k = 2, you'd list

   A, B   A, C   B, A   B, C   C, A   C, B

Week Five: Recursive Backtracking, Big-O and Sorting

1. One of the problems from the "Container Classes" section of this handout discussed compound words, which are words that can be cut into two smaller pieces, each of which is a word. You can generalize this idea further if you allow the word to be chopped into even more pieces. For example, the word "longshoreman" can be split into "long," "shore," and "man," and "whatsoever" can be split into "what," "so," and "ever." Write a function that takes in a word and returns whether it can be split apart into two *or more* smaller pieces, each of which is itself an English word.
2. You are standing on the upper-left corner of a grid of nonnegative integers. You're interested in moving to the lower-right corner of the grid. The catch is that at each point, you can only move up, down, left, or right a number of steps exactly equal to the number you're standing on. For example, if you were standing on the number three, you could move exactly three steps up, exactly three steps down, exactly three steps left, or exactly three steps right. (You can't move off the board). Write a function that determines whether it's possible to get from the upper-left corner (where you're starting) to the lower-right corner while obeying these rules.
3. The pancake sorting problem from the midterm asked you to see whether it was possible to sort a stack of pancakes within *k* flips, for some number *k*. This problem is a lot easier to solve if you don't have the upper limit. Write a function that takes in a stack of pancakes and sorts it, provided that the only legal move you can make is to put a spatula under one of the pancakes, then flip it and all the pancakes above it upside down.

Week Six: Dynamic Arrays

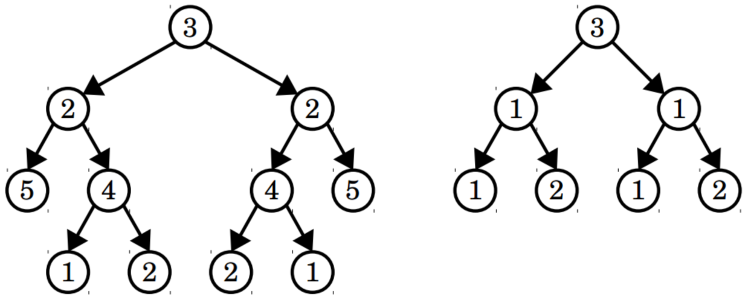
1. The `int` type in C++ can only support integers in a limited range (typically, -2^{31} to $2^{31} - 1$). If you want to work with integers that are larger than that, you'll need to use a type often called a ***big number*** type (or "bignum" for short). Those types usually work internally by storing a dynamic array that holds the digits of that number. For example, the number 78979871 might be stored as the array 7, 8, 9, 7, 9, 8, 7, 1 (or, sometimes, in reverse as 1, 7, 8, 9, 7, 9, 8, 7). Implement a bignum type layered on top of a dynamic array. Your implementation should provide member functions that let you add or multiply together two bignums. *(Hint: start with addition, then use that to implement multiplication)*.
2. Implement a version of the `Grid` type that supports creating a grid of a certain size, reading from grid locations, and writing to grid locations. Do all your own memory management.

Week Seven: Linked Lists

1. Write a function that, given a pointer to a singly-linked list and a number *k*, returns the *k*th-to-last element of the linked list (or a null pointer if no such element exists). How efficient is your solution, from a big-O perspective? As a challenge, see if you can solve this in O(*n*) time with only O(1) auxiliary storage space.
2. Write an implementation of insertion sort that works on singly-linked lists.
3. Imagine that you have two linked lists that meet at some common point in a Y shape (the head pointer of each linked list would be on the top of the Y, and they merge at a common node). Write a function that finds their intersection point. The "branches" of the Y don't have to have the same lengths, and the elements stored within the linked lists might coincidentally match even before their intersection point. As a challenge, see if you can do this in O(1) auxiliary space.

Week Eight: Trees and Hashing

1. A binary tree (not necessarily a binary *search tree*) is called a ***palindromic tree*** if it's its own mirror image. For example, the tree on the left is a palindromic tree, but the tree on the right is not:



Write a function that takes in a pointer to the root of a binary tree and returns whether it's a palindrome tree.

2. (*The Great Tree List Recursion Problem, by Nick Parlante*) A node a binary tree has the same fields as a node in a doubly-linked list: one field for some data and two pointers. The difference is what those pointers mean: in a binary tree, those fields point to a left and right subtree, and in a doubly-linked list they point to the next and previous elements of the list. Write a function that, given a pointer to the root of a binary *search* tree, flattens the tree into a doubly-linked list without allocating any new cells. You'll end up with a list where the pointer `left` functions like the `prev` pointer in a doubly-linked list and where the pointer `right` functions like the `next` pointer in a doubly-linked list.

3. Suppose you insert the numbers 1, 2, 3, 4, 5, …, *n* into one hash table, then insert the numbers *n*, *n*-1, *n*-2, …, 3, 2, 1 into another hash table. Assuming the hash tables are implemented the same way, is it *guaranteed* that the internal structure of the two hash tables will be the same? Is it *possible* that their internal structure will be the same? Is it *never* going to be the case that the internal structure will be the same?

Week Nine: Graphs and Graph Algorithms

1. Explain how question (2) from the section on Recursive Backtracking in this handout is essentially a graph search problem, then explain how to solve it using breadth-first search.

2. Imagine you have a graph representing a social network. Your friends are the people one hop away from you. Someone would be considered a "friend of a friend" if they were two hops away from you (and also not zero or one hops away from you), and someone would be considered a "friend of a friend of a friend" if they were three hops away from you (and also not zero, one, or two hops away from you). Write a function that, given the graph and a number *k*, returns everyone who is a *k*th-order friend of yours.