# Section Handout 4

## Problem 1: Saving Memory with the Stack

In class, we implemented our own version of the stack. One of the biggest efficiency tweaks we made was to dynamically resize our array by multiplying our allocated length by two when the logical length reached the allocated length.

In this problem, you will reimplement `pop();` so that it saves us memory. If the logical length is ¼ the size of the allocated length, we will cut the allocated length in two and shrink our array. Also, discuss why we don't want to shrink our array by two if our logical length is ½ the size of the allocated length.

## Problem 2: The Chat and Cut

We've all been in long lines and have often contemplated cutting them. One of the best techniques is called the "chat and cut." People who use this technique find someone further ahead in line to chat with. After chatting with them, they end up right behind them in line, effectively cutting everyone else behind that person.

For example, let's say Larry wants to cut in line. There are 10 people in line, and Larry's friend Jeff is 3rd in line. If Larry successfully performs the "chat and cut" with Jeff, Larry will be 4th in line, Jeff will remain in 3rd, and the people previously behind Jeff will now also be behind Larry.

Your job is to write a function:

```
bool chatAndCut(Person *firstInLine, string cutter, Set<string> & friends);
```

The line is represented by a linked list using the struct Person. You are given a pointer to the first person in line. Each person in line points to the next person in line. The Person struct has a field called "name" that contains a string with the name of the person in line. You also get a set of friends the cutter has.

Your job is to see if the cutter has any friends in the line. If so, find the friend that is closest to the front of the line and insert the cutter into the line right behind that friend. You should return true if the cutter successfully cuts in line and false if the cutter has no friends in line.

## Problem 3: Reversing a Linked List

In this problem, you will reverse a singly linked list. You should write a function that takes the root of a linked list, reverses it, and returns a pointer to the new root.

```
Node *reverse(Node *root);
```