Section Solutions 9

A ***huge*** thanks to Ali Malik for putting together most of the code here!

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.

Here's one solution:

```
void reverseInPlace(string& str) {
    /* We need to make sure to only loop up to halfway though the string
     * or we will end up with the original string! Try it out on
     * paper if this isn't clear.
     */
    for (int i = 0; i < str.length() / 2; i++) {
        swap(str[i], str[str.length () - 1 - i]); // Question: Why is the -1 here?
    }
}
```

To see that this only uses O(1) storage space, we can account for where all the memory we're using is coming from. We need space for the integer `i`, plus a little stack space for the `swap` function, but aside from that there's no major allocations or storage required.

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.

You can come up with all sorts of different answers to this problem depending on how you define what a word boundary looks like. We've decided to do this by just finding whitespace, but a more intelligent implementation might opt to do more creative checking.

```
/* Essentially the same function as before, except that we specify our own start
 * and end indices so we can reverse parts of a string rather than the whole
 * string at each point. The end index is presumed to be one past the end of the
 * substring in question.
 */
void reverseInPlace(string& str, int start, int end) {
    for (int i = 0; i < (end - start) / 2; i++) {
        swap(str[i], str[end - 1 - i]);
    }
}

void reverseWordOrderingIn(string& sentence) {
    /* Scan across the sentence looking for words. The start variable holds the
     * index of the start point of the current word in the sentence. The variable
     * i holds the index of the character we're currently scanning.
     */
    int start = 0;
    for (int i = 0; i < sentence.length(); i++) {
        if (sentence[i] == ' ') {
            reverseInPlace(sentence, start, i);
            start = i + 1;
        }
    }

    /* We need to account for the fact that there might be a word flush up at the
     * end of the sentence.
     */
    reverseInPlace(sentence, start, sentence.length());

    /* Now reverse everything. */
    reverseInPlace(sentence, 0, sentence.length());
}
```

Again, we can see that we're using O(1) space because we're only using a few temporary variables to hold indices.

3. 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.

The key insight here is to look at the two middle elements of the array to see which direction they slope downhill. Imagine that the water would flow to the left. Then, if we look in the first half of the array, we know there has to be a good collection point somewhere to the left, since if the water flows downhill it has to collect somewhere over there. Using this insight, we can modify our binary search to look like this:

```
/* Given an index into the ridge, returns whether the given index is a good
 * collection point. That happens if both of the position's neighbors are higher
 * than the position itself.
 */
bool isGoodPoint(const Vector<double>& heights, int index) {
    /* Handle boundary cases by pretending the boundaries are infinitely high. */
    double left  = (index == 0? INFINITY : heights[index - 1]);
    double right = (index == heights.size() - 1? INFINITY: heights[index + 1]);

    return heights[index] < left && heights[index] < right;
}

/* Return the index of a good collection point in the interval [start, end). Note
 * that start is inclusive and that end is exclusive.
 */
int findCollectionHelper(const Vector<double>& heights, int start, int end) {
    /* Base case: If the midpoint is a collection point, we're done. */
    int mid = start + (end - start) / 2;
    if (isGoodPoint(heights, mid)) return mid;

    /* If we are on a downward facing slope (left is greater than curr)
     * then search to the right of here
     */
    if (mid > 0 && heights[mid - 1] > heights[mid]) {
        return findCollectionHelper(heights, mid + 1, end);
    }
    /* Otherwise we are on upward facing slope; collection point is to the left
     * so we can limit our search to the left range.
     */
    else {
        return findCollectionHelper(heights, start, mid);
    }
}

int findCollectionPoint(const Vector<double>& heights) {
    return findCollectionHelper(heights, 0, heights.size());
}
```

Week Two: Container Classes

1. Write a function that, given a `HashMap<string, int>` associating string values with integers, produces a `HashMap<int, HashSet<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.)

Here's one possible implementation. Note the use of the map autoinsertion feature.

```
HashMap<int, HashSet<string>> reverseMap(const HashMap<string, int>& map) {
    HashMap<int, HashSet<string>> result;

    for (string oldKey : map) {
        result[map[oldKey]] += oldKey;
    }

    return revMap;
}
```

2. How are `Map` and `HashMap` implemented internally? What's one advantage of `Map` over `HashMap`? One advantage of `HashMap` over `Map`?

The `Map` is internally layered on top of a balanced BST. The `HashMap` is layered on top of a hash table. Because `Map` is backed by a BST, it stores its elements in sorted order, so there's predictable behavior when we iterate over it (we always get things back in sorted order). This contrasts with `HashMap`, where iteration order will visit things in the order in which the elements appear in the underlying hash table, something that's much harder to predict.

The `HashMap` uses a hash table internally, which is generally faster than a binary search tree (expected runtime O(1) for all major operations, compared with O(log n) runtime).

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.

```
bool isCompoundWord(const string& word, const Lexicon& dict) {
    /* Try splitting the word into two possible words for every possible position
     * where you can make the split.
     */
    for (int i = 1; i < word.length(); i++) {
        /* The two words are formed by taking the substring up to the splitting
         * point and the substring starting at the splitting point.
         */
        if (dict.contains(word.substr(0, i)) && dict.contains(word.substr(i))) {
            return true;
        }
    }
    return false;
}

void printCompoundWords(const Lexicon &dict) {
    for (string word: dict) {
        if (isCompoundWord(word, dict)) cout << word << endl;
    }
}
```

Week Three: Graphical Recursion and Recursive Enumeration

1. Pull up the code we wrote in lecture to generate the Sierpinski carpet. To avoid drawing the center of the carpet, we used the code

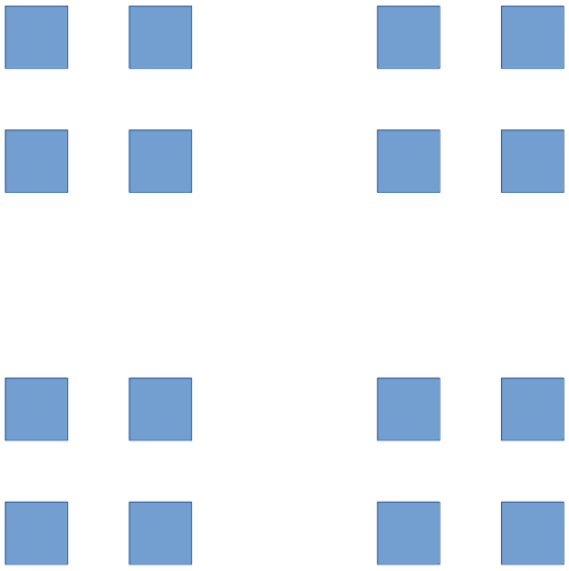
   ```
   if (row != 1 || col != 1) { … }
   ```

   to avoid drawing the center square. What would happen if we changed the `||` to an `&&`? Draw the order-2 image that results.

Of the nine subsquares we could draw, this will only draw in the four corner squares, since those are the squares that are not in the same row *and* not in the same column as the middle square. In other words, the self-similar image we're drawing is given as follows:

- An order-0 image is a filled square.
- An order-*n* image, where $n > 0$, is four order-(n-1) images, each one-third as wide and one-third as tall as the original image, placed in the corners of the bounding square.

That gives rise to this image:



2. Imagine you have a $2 \times n$ 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 $2 \times n$ grid with 2×1 dominoes.

If you draw out a couple of sample tilings, you might notice that every tiling either starts with a single vertical domino or with two horizontal dominoes. That means that the number of ways to tile a $2 \times n$ (for $n \geq 2$) is given by the number of ways to tile a $2 \times (n - 1)$ grid (because any of them can be extended into a $2 \times n$ grid by adding a vertical domino) plus the number of ways to tile a $2 \times (n - 2)$ grid (because any of them can be extended into a $2 \times n$ grid by adding two horizontal dominoes). From there the question is how to compute this. You could do this with regular recursion, like this:

```
int numWaysToTile(int n) {
    /* There's one way to tile a 2 x 0 grid: put down no dominoes. */
    if (n == 0) return 1;

    /* There's one way to tile a 2 x 1 grid: put down a vertical domino. */
    if (n == 1) return 1;

    /* Recursive case: Use the above insight. */
    return numWaysToTile(n - 1) + numWaysToTile(n - 2);
}
```

With a little bit of thought we can note that this function is begging for memoization: we're going to be making a lot of subcalls on the exact same subproblems. Here's one way to address this, given that we know we're going to make recursive calls on values 0, 1, 2, 3, …, n.

```
int numWaysToTile(int n) {
    Vector<int> memo;
    for (int i = 0; i <= n; i++) {    // 0 through n, inclusive
        memo += -1;
    }
    return numWaysRec(n, memo);
}

int numWaysRec(int n, Vector<int>& memo) {
    /* There's one way to tile a 2 x 0 grid: put down no dominoes. */
    if (n == 0) return 1;

    /* There's one way to tile a 2 x 1 grid: put down a vertical domino. */
    if (n == 1) return 1;

    /* If we haven't yet computed this, compute it and stash it for later. */
    if (memo[n] == -1) {
        memo[n] = numWaysRec(n - 1, memo) + numWaysRec(n - 2, memo);
    }
    return memo[n];
}
```

Week Four: Recursive Enumeration and Backtracking

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:

$$3 \quad 2 + 1 \quad 1 + 2 \quad 1 + 1 + 1$$

The key insight here is that some positive number has to come first in our ordering, so we can just try all possible ways of breaking off some initial bit and see what we find.

```
void printSumsOf(int n) {
    /* Handle edge cases. */
    if (n < 0) error("Can't make less than nothing from more than nothing.");

    printSumsRec(n, {});
}

/* Print all ways to sum up to n, given that we've already broken off the numbers
 * given in soFar.
 */
void printSumsRec(int n, const Vector<int>& soFar) {
    /* Base case: Once n is zero, we need no more numbers. */
    if (n == 0) {
        printAsSum(soFar);
    } else {
        /* The next number can be anything between 1 and n, inclusive. */
        for (int i = 1; i <= n; i++) {
            printSumsRec(n - i, soFar + i);
        }
    }
}

/* Prints a Vector<int> nicely as a sum. */
void printAsSum(const Vector<int>& sum) {
    /* The empty sum prints as zero. */
    if (sum.isEmpty()) {
        cout << 0 << endl;
    } else {
        /* Print out each term, with plus signs interspersed. */
        for (int i = 0; i < sum.size(); i++) {
            cout << sum[i];
            if (i + 1 != sum.size()) cout << " + ";
        }
        cout << endl;
    }
}
```

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.

There are many ways that we can do this, but one nice option would be to generate the terms in the sum in nonincreasing order. That is, given the option of writing either 1 + 2 or 2 + 1, we'd choose to print 2 + 1. To do this, we'll use a strategy similar to the one from before, except that we'll cap the maximum value we can use in future recursive calls to ensure things are in nonincreasing order.

```
void printSumsOf(int n) {
    /* Handle edge cases. */
    if (n < 0) error("Can't make less than nothing from more than nothing.");

    /* Initially, the maximum value we can use is n. */
    printSumsRec(n, n, {});
}

/* Print all ways to sum up to n, given that we've already broken off the numbers
 * given in soFar, without using any numbers greater than maxVal.
 */
void printSumsRec(int n, int maxVal, const Vector<int>& soFar) {
    /* Base case: Once n is zero, we need no more numbers. */
    if (n == 0) {
        printAsSum(soFar);
    } else {
        /* The next number can be anything between 1 and maxVal, inclusive. */
        for (int i = 1; i <= maxVal; i++) {
            /* Cap all future terms at the one we just added in. */
            printSumsRec(n - i, i, soFarCopy + i);
        }
    }
}
```

3. Write a function that, given a set of strings and a number k, lists 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

This one is half combinations, half permutations. We use the permutations strategy of asking "what is the next term in our ordered list?" at each step, and the combinations strategy of cutting off our search as soon as we have enough terms.

```
void listKOrderings(const HashSet<string>& choices, int k) {
    /* Quick edge case check: if we want more items than there are options, there
     * are no orderings we can use.
     */
    if (k > choices.size()) {
        listOrderingHelper(choices, k, {});
    }
}

void listOrderingHelper(const HashSet<string>& choices, int k
                        const Vector<string>& soFar) {
    /* Base case: If no more terms are needed, print what we have. */
    if (k == 0) {
        cout << soFar << endl;
    }
    /* Recursive case: What comes next? Try all options. */
    else {
        for (string choice: choices) {
            listOrderingHelper(remaining - choice, k - 1, soFar + choice);
        }
    }
}
```

4. 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.

The main insight here is that a word can be broken apart into two or more words if it can be split into two pieces such that the first piece is a word, and the second piece is either (1) a word or (2) itself something that can be split apart into two or more words.

```
void printMultCompoundWords(const Lexicon& dict) {
    for (string word : dict) {
        if (isMultCompoundWord(word, dict)) {
            cout << word << endl;
        }
    }
}

bool isMultCompoundWord(const string& word, const Lexicon& dict) {
    /* In an unusual twist, our base case is folded into the recursive step. We
     * will try all possible splits into two pieces, and if one of them happens
     * to be a pair of words, we stop.
     */

    /* Try all ways of splitting things. */
    for (int i = 1; i < word.length(); i++) {
        /* Only split if the first part is a word. */
        if (dict.contains(word.substr(0, i))) {
            string remaining = word.substr(i);

            /* We're done if the remainder is either a word or a compound word. */
            if (dict.contains(remaining) || isMultCompoundWord(remaining, dict)) {
                return true;
            }
        }
    }

    /* Nothing works; give up. */
    return false;
}
```

5. 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.

The core idea here is that if we are already in the lower-right corner, great! We're done. Otherwise, we need to take some step to get closer to that point, so we can try all of them.

With a bit of thought, you might realize that what we're doing here is a graph search problem! So we'll approach this as a depth-first search and keep track of which positions we've visited before to speed things up.

```
bool canMoveToBottom(const Grid<int>& grid) {
    /* Where we've been before. */
    Grid<bool> visited(grid.numRows(), grid.numCols(), false);

    /* (0, 0) is upper left corner */
    return canMoveToBottomHelper(0, 0, grid, visited);
}

bool canMoveToBottomHelper(int row, int col,
                           const Grid<int>& grid,
                           Grid<bool>& visited) {
    /* If we are out of bounds or been here before, stop. We either are in an
     * impossible state, or we're somewhere we're repeating ourselves.
     */
    if (!grid.inBounds(row, col) || visited[row][col]) return false;

    /* If we are at the final spot, we've made it! */
    if (row == grid.numRows() - 1 && col == grid.numCols() - 1) return true;

    /* Mark this spot as visited so we don't end up in an infinite loop. */
    visited[row][col] = true;

    /* See how much we can move. */
    int delta = grid[row][col];

    /* Try moving in each direction by the allowed amount */
    return    canMoveToBottomHelper(r + delta, c,         grid, visited)
           || canMoveToBottomHelper(r - delta, c,         grid, visited)
           || canMoveToBottomHelper(r,         c + delta, grid, visited)
           || canMoveToBottomHelper(r,         c - delta, grid, visited);
}
```

Week Five: Big-O and Sorting

1. Below are eight functions. Determine the big-O runtime of each of those pieces of code.

```
void function1(int n) {
  for (int i = 0; i < n; i++) {
    cout << '*' << endl;
  }
}
```

The runtime of this code is O(n): We print out a single star, which takes time O(1), a total of n times.

```
void function2(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cout << '*' << endl;
    }
  }
}
```

The runtime of this code is O(n^2). The inner loop does O(n) work, and it runs O(n) times for a net total of O(n^2) work.

```
void function3(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      cout << '*' << endl;
    }
  }
}
```

This one also does O(n^2) work. To see this, note that the first iteration of the inner loop runs for $n - 1$ iterations, the next for $n - 2$ iterations, then $n - 3$ iterations, etc. Adding all this work up across all iterations gives $(n - 1) + (n - 2) + \ldots + 3 + 2 + 1 + 0 = O(n^2)$, as we saw in class when doing the analysis of insertion sort and selection sort.

```
void function4(int n) {
  for (int i = 1; i <= n; i *= 2) {
    cout << '*' << endl;
  }
}
```

This one runs in time O(log n). To see why this is, note that after k iterations of the inner loop, the value of `i` is equal to 2^k. The loop stops running when 2^k exceeds n. If we set $2^k = n$, we see that the loop must stop running after $k = \log_2 n$ steps.

Another intuition for this one: the value of `i` doubles on each iteration, and you can only double O(log n) times before you overtake the value n.

```
void function5(int n) {
  if (n == 0) return;
  function5(n - 1);
}
```

Each recursive call does a constant amount of work (specifically, it just checks the value of n and optionally makes a recursive call). The question, then, is how many recursive calls there are. Each recursive call fires off one more recursive call with n decreased by one. After $O(n)$ total recursive calls are made, therefore, the value of n will drop to zero and we'll stop. Since we're doing $O(1)$ work $O(n)$ times, this function takes time $O(n)$ to run.

```
void function6(int n) {
  if (n == 0) return;
  function6(n - 1);
  function6(n - 1);
}
```

As before, each recursive call does $O(1)$ work, and so the question is how many recursive calls are made. This one, however, is more subtle than before. Think about the shape of the recursion tree: each node in the tree will branch and have two nodes beneath it. That means that if we make an initial call of `function6(n)`, there will be two calls to `function6(n - 1)`. Each of those calls fires off two more recursive calls to `function6(n - 2)`, so there are four calls to `function6(n - 2)`. There's then eight calls to `function6(n - 3)`, sixteen calls to `function6(n - 4)`, and more generally 2^k calls to `function6(n - k)`. Overall, the total number of recursive calls made is equal to

$$2^0 + 2^1 + 2^2 + \ldots + 2^n.$$

Since these values grow exponentially quickly, the very last one accounts for almost the complete value of the sum, so the work done here is $O(1)$ work per call times $O(2^n)$ total calls for a total of $O(2^n)$ work.

```
void function7(int n) {
  if (n == 0) return;
  function7(n / 2);
}
```

Each recursive call here does $O(1)$ work, as before, so the question is how many calls get made. Notice that the value of n drops by half each time a recursive call is made, and that can only happen $O(\log n)$ times before the value drops all the way to zero. (Remember that integer division in C++ rounds down, so when we get to $n = 1$ the next call will be with $n = 0$). Therefore, the runtime is $O(\log n)$.

```
void function8(int n) {
  if (n == 0) return;

  for (int i = 0; i < n; i++) {
    cout << '*' << endl;
  }

  function8(n / 2);
  function8(n / 2);
}
```

Each recursive call here does $O(n)$ work for the `for` loop, then makes two recursive calls on two inputs half as big as the original. You might recognize this as the same pattern that mergesort follows! As a result, the runtime here is $O(n \log n)$.

2. What is the big-O runtime of this function in terms of n, the number of elements in v?

```
int squigglebah(const Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        Vector<int> values = v.subList(0, i);
        for (int j = 0; j < values.size(); j++) {
            result += values[j];
        }
    }
    return result;
}
```

Let's follow the useful maxim of "when in doubt, work inside out!" The innermost `for` loop (the one counting with `j`) does work proportional to the size of the `values` list, and the `values` list has size equal to `i` on each iteration. Therefore, we can simplify this code down to something that looks like this:

```
int squigglebah(const Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        Vector<int> values = v.subList(0, i);
        do O(i) work;
    }
    return result;
}
```

Now, how much work does it take to create the `values` vector? We're copying a total of `i` elements from `v`, and so the work done will be proportional to `i`. That gives us this:

```
int squigglebah(const Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        do O(i) work;
        do O(i) work;
    }
    return result;
}
```

Remember that doing O(i) work twice takes time O(i), since big-O ignores constant factors. We're now left with this:

```
int squigglebah(const Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        do O(i) work;
    }
    return result;
}
```

This is the same pattern as function2 in the previous problem, and it works out to O(n^2) total time.

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 together two bignums or produce a string representation of a bignum, and a constructor that lets you initialize the bignum to some integer value. For simplicity, you don't need to worry about negative numbers.

Let's begin with the interface for our class, which will look like this:

```
class BigNum {
public:
    BigNum(int value = 0);   // Default to zero unless specified otherwise.
    ~BigNum();

    std::string toString() const; // Get a string representation
    void add(const BigNum& value);

private:
    int* digits;       // Stored in reverse order
    int allocatedSize; // In # of digits
    int numDigits;     // In # of digits.

    void reserve(int space); // Ensure we have space to hold numDigits digits
    int numDigitsOf(int value) const;  // How many digits are in value?
};
```

Here, the constructor takes in the value we'll store. The `toString` function produces a string representation of the number, and the `add` function takes in another `BigNum` and adds its value to the total.

Interally, we'll represent the `BigNum` as an array of integers, each of which represents a single digit. The `allocatedSize` and `numDigits` variables track how many digits we have space for and how many digits we actually have, respectively. (A note: it's actually not a good use of space to store each digit as an integer because the `int` type can hold much, much larger values than a single digit, but for simplicity we'll opt for that approach. Take CS107 to see some alternatives!)

We'll also write a function named `reserve()`, which takes as input a number of digits and then does whatever needs to be done to ensure that we have space for at least that many digits. Think of it as a more cautious version of the `grow()` function we wrote for our stack type: it'll make the array bigger, but only if it needs to.

To make the implementation simpler, we'll store the digits of our number in reverse order, so the number 137 would be stored as 7, 3, 1. This makes the math easier and makes it easier to add digits in, since it's usually easier to add new items further in an array rather than earlier.

The implementation itself is on the next page.

```
BigNum::BigNum(int value) {
    /* Set up an initial array of elements. */
    digits = new int[kDefaultSize]; // Some reasonable default
    allocatedSize = kDefaultSize;

    /* Count how many digits are in our number. 0 is an edge case, so we'll make
     * this work by reducing the number of digits down to the last one.
     */
    logicalSize = numDigitsOf(value);

    /* Copy the number into the array, one digit at a time. */
    for (int i = 0; i < logicalSize; i++) {
        digits[i] = value % 10;
        value /= 10;
    }
}

int BigNum::numDigitsOf(int value) const {
    /* Pro C++ tip: Since this function doesn't read or write any member
     * variables, this should either be a free function or a static member
     * function. We didn't discuss those sorts of concerns in CS106B, though.
     */
    int result = 1; // All numbers have at least one digit.

    /* We've already counted one digit. Now count the rest. */
    while (value >= 10) {
        result++;
        value /= 10;
    }

    return result;
}

BigNum::~BigNum() {
    delete[] digits;
}

string BigNum::toString() const {
    /* Because characters are stored in reverse order, we have to scan them
     * backwards to make our number.
     */
    string result;
    for (int i = numDigits - 1; i >= 0; i--) {
        result += to_string(digits[i]);
    }
    return result;
}
```

/* * * * Continued on the next page * * * */

```
void BigNum::add(const BigNum& value) {
    /* First, ensure we have space to hold the result. Adding two numbers produces
     * a result whose size is at most one digit bigger than either input number.
     */
    int digitsToVisit = max(numDigits, value.numDigits);
    reserve(1 + digitsToVisit);

    /* Use the grade school algorithm to add the numbers. */
    int carry = 0;

    for (int i = 0; i < digitsToVisit; i++) {
        int sum = digits[i] + value.digits[i] + carry;

        /* Write the one's place. */
        digits[i] = sum % 10;

        /* Store the carry. */
        carry = sum / 10;
    }

    /* We need at least as many digits as before, plus one if there is a final
     * carry.
     */
    numDigits = digitsToVisit + carry;

    if (carry == 1) digits[digitsToVisit] = 1;
}

/* Reserving space uses the regular "double in size" trick. */
void BigNum::reserve(int space) {
    /* If we have the space, then we don't need to do anything. */
    if (space <= allocatedSize) return;

    /* Double and copy. We deliberately don't just grow to the size requested,
     * since that may not be efficient.
     */
    allocatedSize *= 2;
    int* newDigits = new int[allocatedSize];

    for (int i = 0; i < numDigits; i++) {
        newDigits[i] = digits[i];
    }

    delete[] digits;
    digits = newDigits;
}
```

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.

We're going to show off a fun technique in our solution called *lazy evaluation*.

Unlike the `Stack`, `Vector`, `Map`, etc., the `Grid` type has its size specified when it is created and does not grow or shrink. One simple way to represent an $m \times n$ grid of values in C++ is as a multidimensional array of size $m \times n$. When the grid is created, all of the elements in the grid are then initialized to an initial value (for `int`s, this is `0`). This requires O(mn) storage space.

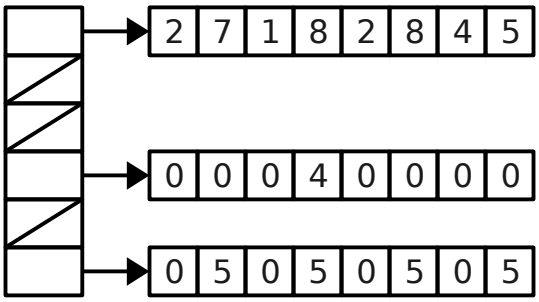
In many cases, though, the client of the `Grid` will not actually fill in all of the grid locations, leaving many of them holding their initial values. For example, suppose that only the top and bottom row of the grid are actually updated to hold new values. In that case, it's wasteful to store the middle rows of the grid, since we already know what values are stored there (namely, each cell holds the initial value). In the case where in advance it's known that not all rows of the `Grid` will be used, it's possible to reduce the amount of space the `Grid` will use by not actually allocating space for those rows until values are written to them.

Here is how this data structure will work. To represent a 2D array of size `numRows` × `numCols`, we will use an "array of arrays," with a top-level array of `numRows` pointers, each of which either points to `nullptr` or to an array of `numCols` values. If none of the elements in row `m` have been written to, then the `m`th pointer in the top-level array will be `nullptr`. Otherwise, if any element in row `m` is written to, then the `m`th pointer in the top-level array will point to an array of `numCols` elements that represents the values stored in that row.

For example, suppose that we want to store this 2D array of `int`s with our data structure:

2	7	1	8	2	8	4	5
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	4	0	0	0	0
0	0	0	0	0	0	0	0
0	5	0	5	0	5	0	5

Since `0` is the initial value for `int`s, we would represent this grid as



Notice how all of the rows that are all `0` are represented as null pointers, while the rows containing nonzero values actually have arrays allocated for them.

When the data structure is initially constructed, the top-level array of size `numRows` is allocated, but none of the arrays for any of the rows are allocated. Whenever a value is read from the grid, if the row containing the value has not been allocated yet, the implementation can just return `0`, since that's the default value for `int`s. There's no need to allocate the row in this case. Whenever a value is written to one of the rows in the grid, if the array for that row is already allocated, the new value is placed directly into that array. If a value is written and the array for that row has not yet been allocated, the array is then allocated, all of its elements are set to `0`, and the new value is then written. (In principle, you could deallocate a row if it ever becomes all `0`s, but for simplicity we won't do this.)

Here's our implementation:

```
class LazyGrid {
public:
   LazyGrid(int nRows, int nCols);
   ~LazyGrid();

   int getAt(int row, int col);
   void setAt(int row, int col, int value);

private:
   /* Pointer to the top-level array-of-arrays used by this data structure.  Since
    * each element in this array is a pointer to an array (an int*), we need the
    * pointer to be an int** (a pointer to an array of int*'s).
    */
   int** elems;
   int numRows, numCols;
};

LazyGrid::LazyGrid(int nRows, int nCols) {
    if (nRows < 0 || nCols < 0) error("Can't make a negative-sized grid.");
    numRows = nRows;
    numCols = nCols;

    /* There are two levels of initialization here. First, we have to make the
     * top-level array. Then, we have to set each entry in the top-level array to
     * null to indicate nothing is in that row yet.
     */
    elems = new int*[numRows];
    for (int i = 0; i < numRows; i++) {
        elems[i] = nullptr;
    }
}

LazyGrid::~LazyGrid() {
    /* Two levels of deallocation are necessary. We need to clean up the memory
     * for each row, then clean up the top-level pointer to the array of rows.
     */
    for (int i = 0; i < numRows; i++) {
        delete[] elems[i];
    }
    delete[] elems;
}

int LazyGrid::getAt(int row, int col) {
    /* Bounds-checking. */
    if (row < 0 || col < 0 || row >= numRows || col >= numRows) {
        error("Out of bounds!");
    }

    /* If the row is empty, the default is zero. */
    return elems[row] != nullptr? elems[row][col] : 0;
}


                /* * * * Continued on the next page * * * */
```

```
void LazyGrid::setAt(int row, int col, int value) {
    /* Bounds-check. */
    if (row < 0 || col < 0 || row >= numRows || col >= numRows) {
        error("Out of bounds!");
    }

    /* If there wasn't already a row there, pretend one was there all along! */
    if (elems[row] == nullptr) {
        elems[row] = new int[numCols];
        for (int col = 0; col < numCols; col++) {
            elems[row][col] = 0;
        }
    }
    elems[row][col] = value;
}
```

Week Seven: Hashing and Hash Tables

1. Is it ever possible, in a linear probing hash table, that a lookup for an item whose hash code is 5 would end up finding that element in slot 3? Justify your answer.

Sure, that's possible. If slots 5, 6, 7, …, *numSlots* – 1, 0, 1, and 2 are all filled when the item was inserted, it would wrap around the table and end up in slot 3.

2. Suppose you insert the numbers 1, 2, 3, 4, 5, …, *n* into one linear probing table, then insert the numbers *n*, *n*-1, *n*-2, …, 3, 2, 1 into another linear probing table. 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?

No, this is not guaranteed to happen. Imagine, for example, that the table has the (terrible) hash function of "drop everything into slot zero." Then in the first case the items will be stored in sorted order, and in the second case they'll be stored in reverse-sorted order.

It is, however, *possible* the two internal structures will be the same. For example, if there are no hash collisions between the elements, then each element ends up in the same slot regardless of what order elements were inserted in.

Week Seven: Eight Lists

1. Write a function that, given a pointer to a singly-linked list and a number k, returns the kth-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.

There are a couple of ways we could do this. One option would be to sweep across the list from the front to the back, counting how many nodes there are, then calculate the index we need. That's shown here:

```
struct Node {
    string value;
    Node* next;
};

int listLength(Node* list) {
    int count = 0;

    /* Cute little for loop trick to visit everything in a linked list. This loop
     * is great if you are not making any changes to the list, but if the list is
     * either being rewired or being deallocated, this loop won't work.
     */
    for (Node* curr = list; curr != nullptr; curr = curr->next) {
        count++;
    }
    return count;
}

Node* kthToLastSimple(Node* list, int k) {
    /* Find length of the list */
    int len = listLength(list);

    /* If the list is too small, just return nullptr. */
    if (len < k) return nullptr;

    /* Move len - k + 1 steps into the list */
    Node* curr = list;
    for (int i = 0; i < len - k; i++) {
        curr = curr->next;
    }
    return curr;
}
```

Another option, which is a bit less obvious but is quite beautiful, is to walk down the list with two concurrent pointers, one of which is k steps ahead of the other. As soon as the lead pointer falls off the list, the pointer behind it is k steps from the end. Do you see why?

```
Node* kthToLast(Node* list, int k) {
    /* Set up two pointers, one leader and one follower. */
    Node* leader   = list;
    Node* follower = list;

    /* March the leader k steps forward. If we fall of the list in this time,
     * there is no kth-to-last node.
     */
    if (leader == nullptr) return nullptr;
    for (int i = 0; i < k; i++) {
        leader = leader->next;

        /* Question to ponder: Why are there two checks, above and here? */
        if (leader == nullptr) return nullptr;
    }

    /* Keep walking the leader and follower forward. As soon as the leader walks
     * off the follower is in the right spot.
     */
    while (true) {
        leader = leader->next;
        if (leader == nullptr) return follower;

        follower = follower->next;
    }
}
```

2. Write an implementation of insertion sort that works on singly-linked lists.

The singly-linked list requirement here suggests that our pattern of repeatedly swapping elements back in the sequence is not going to be easy to implement – we'll keep losing track of where our preceding element is. There are many ways we could deal with this. One would be to build the sorted list in reverse, then fix up the pointers at the end to reverse it. Another, shown below, works by taking the element, moving it to the front of the list, then swapping it forward until it's in the right position.

```
void listInsertionSort(Node*& list) { // Question to ponder: why by reference?
    Node* sortedList = nullptr;

    Node* curr = list;
    while (curr != nullptr) {
        /* Need to store the pointer to the next cell in the list, since after
         * we do the insert operation we'll lose track of where the next cell
         * is.
         */
        Node* next = curr->next;
        sortedInsert(curr, sortedList);
        curr = next;
    }
    list = sortedList;
}

/* Adds the given node into a sorted, singly-linked list. */
void sortedInsert(Node* toIns, Node*& list) {
    /* See if we go at the beginning. */
    if (list == nullptr || toIns->value < list->value) {
        toIns->next = list;
        list = toIns;
    } else {
        /* Find the spot right before where we go, since that's the pointer we
         * need to rewire.
         */
        Node* curr = list;
        Node* prev = nullptr;

        while (curr != nullptr && curr->value < toIns->value) {
            prev = curr;
            curr = curr->next;
        }

        /* Splice us in. */
        toIns->next = curr;
        prev->next = toIns;
    }
}
```

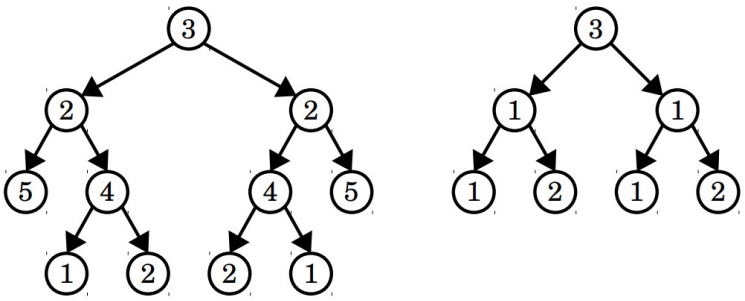
3. Imagine that you have two linked lists that meet at some common point in a Y shape (the head pointer of each linked list would be on the top of the Y, and they merge at a common node). Write a function that finds their intersection point. The "branches" of the Y don't have to have the same lengths, 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.

Suppose the first list appears to have m elements and the second appears to have n, where $m \geq n$. The key observation is that the intersection point can only be in the last n nodes of the two lists – any earlier and both lists would have to be longer than length n. (Do you see why?) This suggests a nice algorithm: make a pass over each list to compute their lengths, skip down so that we're n steps from the end of both lists, then walk forward until they meet.

```
Node* findIntersection(Node* list1, Node* list2) {
    /* Find the difference in length between the two lists */
    int len1 = listLength(list1);
    int len2 = listLength(list2);

    /* Swap the lists so that list1 is longer. */
    if (len1 < len2) {
        swap(list1, list2);
        swap(len1,  len2);
    }

    /* Advance the first pointer forward until the paths to the end of both lists
     * appear equal.
     */
    for (int i = 0; i < len1 - len2; i++) {
        list1 = list1->next;
    }

    /* March both lists forward until they intersect. */
    while (true) {
        if (list1 == list2) return list1;

        list1 = list1->next;
        list2 = list2->next;
    }
}
```

Week Nine: Trees

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.

To solve this problem, we'll solve a slightly more general problem: given two trees, are they mirrors of one another? We can then check if a tree is a palindrome by seeing whether that tree is a mirror of itself.

```
bool isPalindromicTree(Node* root) {
    return areMirrors(root, root);
}

bool areMirrors(Node* root1, TreeNode* root2) {
    /* If either tree is empty, both must be. */
    if (root1 == nullptr || root2 == nullptr) {
        return root1 == root2;
    }

    /* Neither tree is empty. The roots must have equal values. */
    if (root1->value != root2->value) {
        return false;
    }

    /* To see if they're mirrors, we need to check whether the left subtree of
     * the first tree mirrors the right subtree of the second tree and vice-versa.
     */
    return areMirrors(root1->left,  root2->right) &&
           areMirrors(root1->right, root2->left);
}
```

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, with the values in sorted order, 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.

This is a beautiful recursion problem. Essentially, what we want to do is the following:

- The empty tree is already indistinguishable from the empty list.
- Otherwise, flatten the left subtree and right subtree, then concatenate everything together.

To implement that last step efficiently, we'll have our recursive function hand back two pointers: one to the front of the flattened list and one to the back.

```
struct Range {
    Node* first;
    Node* last;
};

Node* treeToList(Node* root) {
    return flatten(root).first;
}

Range flatten(Node* root) {
    /* If the tree is empty, it's already flattened. */
    if (root == nullptr) return { nullptr, nullptr };

    /* Flatten the left and right subtrees. */
    Range left  = flatten(root->left);
    Range right = flatten(root->right);

    /* Glue things together. */
    root->left = left.last;
    if (left.last != nullptr) left.last->right = root;

    root->right = right.first;
    if (right.first != nullptr) right.first->left = root;

    /* Return the full range. */
    return {
        left.first == nullptr? root : left.first,
        right.last == nullptr? root : right.last
    };
}
```

Thanks for reading this far!

Good luck on the final exam!