

## Section Handout #1

This week's section handout has practice with Grids, Vectors, file reading, as well as review of Big-Oh Notation.

### 1. Mirror (CodeStepByStep)

Write a function `mirror` that accepts a reference to a grid of integers as a parameter and flips the grid along its diagonal, so that each index `[i][j]` contains what was previously at index `[j][i]` in the grid. You may assume the grid is square, that is, it has the same number of rows as columns. For example, the grid below at left would be altered to give it the new grid state at right:

```
{ { 6,  1,  9,  4},      { {6, -2, 14, 21},
  {-2,  5,  8, 12},      -->  {1,  5, 39, 55},
  {14, 39, -6, 18},      {9,  8, -6, 73},
  {21, 55, 73, -3}}      {4, 12, 18, -3}}
```

### 2. Rotate Clockwise (CodeStepByStep)

Write a function `rotateClockwise90Degrees` that accepts a reference to a grid of integers as a parameter and rotates the Grid 90 degrees clockwise. You may assume the grid is square, that is, it has the same number of rows as columns. For example, the grid below at the left would be altered to give it the new grid state at right.

```
{ { 6,  1,  9,  4},      { {21, 14, -2,  6},
  {-2,  5,  8, 12},      -->  {55, 39,  5,  1},
  {14, 39, -6, 18},      {73, -6,  8,  9},
  {21, 55, 73, -3}}      {-3, 18, 12,  4}}
```

### 3. Stretch (CodeStepByStep)

Write a function named `stretch` that accepts a reference to a vector of integers as a parameter and modifies it to be twice as large, replacing every integer with a pair of integers, each half the original. If a number in the original vector is odd, then the first number in the new pair should be one higher than the second so that the sum equals the original number. For example, passing the vector `{18, 7, 4, 24, 11}` should modify the vector to contain `{9, 9, 4, 3, 2, 2, 12, 12, 6, 5}`.

### 4. Big-Oh Analysis

Give a tight bound on the nearest runtime complexity for each of the following code fragments in Big-Oh, in terms of the variable `N`. In other words, find the growth rate of the code's runtime as `N` grows. Do not worry if you are unfamiliar with some of the data structures presented in this problem. We'll learn about them soon enough!

```
// a)
int sum = 0;
for (int i = 1; i <= N + 2; i++) {
    sum++;
}
for (int j = 1; j <= N * 2; j++) {
    sum += 5;
}
cout << sum << endl;
```

```
// b)
int sum = 0;
for (int i = 1; i <= N - 5; i++) {
    for (int j = 1; j <= N - 5; j += 2) {
        sum++;
    }
}
cout << sum << endl;
```

```

// c)
int sum = N;
for (int i = 0; i < 1000000; i++) {
    for (int j = 1; j <= 1; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
}
cout << sum << endl;

```

```

// d)
HashSet<int> set1;
for (int i = 1; i <= N; i++) {
    set1.add(i);
}

Set<int> set2;
for (int i = 1; i <= N; i++) {
    set1.remove(i);
    set2.add(i + N);
}
cout << "done!" << endl;

```

### 5. Oh? More Big-Oh?

Give a tight bound on the nearest runtime complexity for each of the following code fragments in Big-Oh, in terms of the variable  $N$ . In other words, find the growth rate of the code's runtime as  $N$  grows.

```

// a)
int sum = 0;
for (int i = 1; i <= N - 2; i++) {
    for (int j = 1; j <= i + 4; j++) {
        sum++;
    }
    sum++;
}
cout << sum << endl;

```

```

// b)
int sum = 0;
for (int i = 1; i <= N * 2; i++) {
    for (int j = 1; j <= i / 2; j += 2) {
        for (int k = 0; k < N * N; k++) {
            sum++;
        }
    }
}
cout << sum << endl;

```

```

// c)
Vector<int> list;
for (int i = 0; i < N; i++) {
    list.insert(0, i * i);
}
Set<int> set;
for (int k : list) {
    set.add(k);
}
cout << "done!" << endl;

```

```

// d)
int sum = 0;
for (int i = 1; i <= 100000; i++) {
    for (int j = 1; j <= i; j++) {
        for (int k = 1; k <= N; k++) {
            sum++;
        }
    }
}
cout << sum << endl;

```

### 6. Keith Numbers (CodeStepByStep)

A Keith Number is defined as any  $n$ -digit integer that appears in the sequence that starts off with the number's  $n$  digits and then continues such that each subsequent number is the sum of the preceding  $n$ . All one-digit numbers are trivially Keith numbers, but there are more interesting ones as well. For example, the number 7385 is a Keith number because of the following sequence:

7, 3, 8, 5, 23, 39, 75, 142, 279, 535, 1031, 1987, 3832, 7385

Keith numbers are computationally hard to calculate; there are only about 100 known right now. Write a function `findKeithNumbers` that takes a minimum and maximum value and finds all Keith numbers between those values (inclusive). For each number, it should print the sequence that proves it is a Keith number. For example, if you call `findKeithNumbers(1, 1000)`, it should print:

```
1: {1}
2: {2}
3: {3}
4: {4}
5: {5}
6: {6}
7: {7}
8: {8}
9: {9}
14: {1, 4, 5, 9, 14}
19: {1, 9, 10, 19}
28: {2, 8, 10, 18, 28}
47: {4, 7, 11, 18, 29, 47}
61: {6, 1, 7, 8, 15, 23, 38, 61}
75: {7, 5, 12, 17, 29, 46, 75}
197: {1, 9, 7, 17, 33, 57, 107, 197}
742: {7, 4, 2, 13, 19, 34, 66, 119, 219, 404, 742}
```

### 7. Average in File. *(file input/output)*

Write a function named **averageValueInFile** that reads a file and returns the average (mean) of the numbers in that file. The parameter, `filename`, gives the name of a file that contains a list of real numbers, one per line. You may assume that the file exists and follows the proper format.

```
double averageValueInFile(string filename) { . . . }
```