Eric Roberts
CS106B

Handout #33
February 11, 2013

Answers to Midterm Exam

Congratulations! As a class, you knocked this one out of the park. We believed the exam to be fair but not unduly easy, so most of you can take real pride in doing so well. Given the independent corroborative evidence from the section leaders who tell me that things are going well, I've decided to assign a high curve of the sort I think you deserve. The complete histogram appears on page 2, but here are a few noteworthy statistics:

N = 388
Median = 53.0
Mean = 50.3

The scale to letter grades looks like this:

Grade	Range	N
A+	59–60	64
A	55–58	109
A–	53–54	35
B+	51–52	33
B	47–50	44
B–	44–46	27
C+	41–43	19
C	37–40	16
C–	34–36	10
D	25–33	23
NP	00–24	8

Histogram

12 11

18 17 15

22 20 20

29 29 28 28 28 28 28 27 27 26 25 25 25

34 34 34 34 34 32 32 31 31 31 30 30 30 30 30

39 39 39 38 38 38 38 38 37 37 37 37 36 36 36 35 35

44 44 44 44 44 44 44 44 44 43 43 43 43 43 43 43 43 43 42 42 42 42 41 41 41 41 41 41 40 40 40 40

49 49 49 49 49 49 49 49 49 49 49 48 48 48 48 48 48 48 48 48 48 48 47 47 47 47 47 46 46 46 46 46 46 46 46 46 46 45 45 45 45 45 45 45 45

54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 53 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 51 51 51 51 51 51 51 51 51 51 51 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50

59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60

Problem 1: Tracing C++ programs and big-O (10 points)

Just like the `mystery` function on the first practice exam (Handout #23), this version of `mystery` calculates 2^n, this time using an iterative version of the `raiseIntToPower` algorithm from problem 4 on that same practice exam. The result is therefore

`mystery(10)` → 1024

Determining the complexity order requires counting how many times the operations inside the loop are executed as a function of *N*. As in the outer loop of the merge sort algorithm, this loop repeatedly divides the value of `n` by two until it reaches 0. The number of times the loop executes is therefore proportional to $\log_2 N$, which means that the computational complexity is $O(\log N)$.

Problem 2: Vectors, grids, stacks, and queues (10 points)

```
/*
 * Function: smoothImage
 * Usage: smoothImage(image);
 * --------------------------
 * Updates the values in the image grid by replacing each element with
 * the average of its neighbors.
 */

void smoothImage(Grid<double> & image) {
   int rows = image.numRows();
   int cols = image.numCols();
   Grid<double> copy = image;
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         image[i][j] = averageNeighbors(copy, i, j);
      }
   }
}

/*
 * Function: averageNeighbors
 * Usage: double ave = averageNeighbors(image, row, col);
 * ------------------------------------------------------
 * Returns the average intensity of the 3x3 square centered at the
 * specified row and column of the image grid.
 */

double averageNeighbors(Grid<double> & image, int row, int col) {
   double total = 0;
   int count = 0;
   for (int dy = -1; dy <= 1; dy++) {
      for (int dx = -1; dx <= 1; dx++) {
         if (image.inBounds(row + dy, col + dx)) {
            total += image[row + dy][col + dx];
            count++;
         }
      }
   }
   return total / count;
}
```

Problem 3: Lexicons, maps, and iterators (15 points)

```
/*
 * Function: createRhymingDictionary
 * Usage: dict = createRhymingDictionary(lex, endings);
 * ----------------------------------------------------
 * Creates a rhyming dictionary from a specified set of endings.
 * The map that is returned associates endings with a vector of
 * words that "rhyme" with that ending, in the sense that the
 * spelling is the same.
 */

Map< string, Vector<string> >
        createRhymingDictionary(Lexicon & lex, Vector<string> & endings) {
   Map< string, Vector<string> > dict;
   for (string word : lex) {
      for (string ending : endings) {
         if (endsWith(word, ending)) {
            dict[ending].add(word);
         }
      }
   }
   return dict;
}
```

Problem 4: Recursive functions (10 points)

```
/*
 * Function: catalanNumber
 * Usage: int cn = catalanNumber(n);
 * ---------------------------------
 * Returns the nth Catalan number, which is calculated by counting the
 * number of paths from the upper right corner of an n by n square grid
 * that don't venture into the upper-left triangle in which y > x.
 */

int catalanNumber(int n) {
   return catalan2(n, n);
}

/*
 * Function: catalan2
 * Usage: int cn = catalan2(x, y);
 * -------------------------------
 * Returns the number of paths from (x, y) back to the origin that don't
 * venture into the upper-left triangle in which y > x.
 */

int catalan2(int x, int y) {
   if (y > x) return 0;
   if (y == 0) return 1;
   return catalan2(x - 1, y) + catalan2(x, y - 1);
}
```

For more details about why Catalan numbers are interesting, check out the Wikipedia page.

Problem 5: Recursive procedures (15 points)

```
/*
 * Function: solveStepMaze
 * Usage: bool solvable = solveStepMaze(maze);
 *        bool solvable = solveStepMaze(maze, row, col, visited);
 * ---------------------------------------------------------------
 * Attempts to generate a solution to the specified step maze.  The
 * second version of this function takes the row and column index of
 * the current location and a Boolean grid visited, which indicates
 * whether that square has already appeared in the solution path.
 */

bool solveStepMaze(Grid<int> & maze) {
   Grid<bool> visited(maze.numRows(), maze.numCols());
   return solveStepMaze(maze, 0, 0, visited);
}

bool solveStepMaze(Grid<int> & maze, int row, int col, Grid<bool> & visited) {
   if (!maze.inBounds(row, col)) return false;
   if (visited[row][col]) return false;
   if (maze[row][col] < 0) return false;
   if (maze[row][col] == 0) return true;
   int nSteps = maze[row][col];
   visited[row][col] = true;
   return solveStepMaze(maze, row + nSteps, col, visited)
       || solveStepMaze(maze, row - nSteps, col, visited)
       || solveStepMaze(maze, row, col + nSteps, visited)
       || solveStepMaze(maze, row, col - nSteps, visited);
}
```