Eric Roberts
CS 106B

Handout #26A
February 6, 2015

Answers to Practice Midterm Exam #1

Review session: Sunday, February 8, 7:00–9:00 P.M., TBA
Midterm #1: Tuesday, February 10, 2:15–4:15 P.M., Annenberg Auditorium
Midterm #2: Tuesday, February 10, 7:00–9:00 P.M., Bishop Auditorium

Problem 1: Function tracing and/or memory diagramming (10 points)

The `mystery` function calculates 2^n (using an iterative version of the `raiseIntToPower` algorithm from problem 4). 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)

There are many strategies for solving this problem. A linear-time solution that uses the remainder operator to achieve the wrap-around semantics of `roll` looks like this:

```
/*
 * Function: roll
 * Usage: roll(stack, n, k);
 * -------------------------
 * Rotates the top n elements of the stack k positions toward the top.
 * This function generates an error if either of the arguments is negative
 * or if n is greater than the size of the stack.
 */

void roll(Stack<char> & stack, int n, int k) {
   if (n < 0 || k < 0 || n > stack.size()) {
      error("roll: argument out of range");
   }
   Vector<char> vec;
   for (int i = 0; i < n; i++) {
      vec.add(stack.pop());
   }
   for (int i = n - 1; i >= 0; i--) {
      stack.push(vec[(i + k) % n]);
   }
}
```

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

```
/*
 * Function: generateCompletions
 * Usage: Vector<string> completions =
 *          generateCompletions(digits, lexicon, cache);
 * ----------------------------------------------------
 * Generates all possible completions from a string of digits
 * and returns the result as a vector.  The third argument (cache)
 * is used to store previously generated values so that they can
 * be returned instantly once they have been calculated.
 */

Vector<string> generateCompletions(string digits, Lexicon & lex,
                           Map< string, Vector<string> > & cache) {
   if (cache.containsKey(digits)) return cache.get(digits);
   Vector<string> result;
   for (string word : lex) {
      if (prefixMatches(digits, word)) result.add(word);
   }
   cache.put(digits, result);
   return result;
}

/*
 * Function: prefixMatches
 * Usage: if (prefixMatches(digits, word)) . . .
 * ---------------------------------------------
 * Checks whether the initial letters in word can be spelled
 * on a cell-phone dial by the digits in the first argument.
 * This code takes advantage of the fact that word comes from
 * a Lexicon and is therefore composed only of lowercase letters.
 * In an exam setting, most of you would use a switch statement
 * to figure out what digit corresponded to the current letter in
 * word; selecting the digit from a string is a bit more compact.
 */

bool prefixMatches(string digits, string word) {
   if (digits.length() > word.length()) return false;
   for (int i = 0; i < digits.length(); i++) {
      char digit = "22233344455566677778889999"[word[i] - 'a'];
      if (digit != digits[i]) return false;
   }
   return true;
}
```

Problem 4: Recursive functions (10 points)

The following implementation computes n^k in $O(\log k)$ time:

```
/*
 * Function: raiseIntToPower
 * Usage: p = raiseIntToPower(n, k);
 * ---------------------------------
 * This function returns n to the kth power.  It depends on
 * the recursive insight that n to the k is the square of
 * n to the k / 2 power, for even values of k.  For odd
 * values of k, the result is the same except for an extra
 * factor of n.
 */

int raiseIntToPower(int n, int k) {
   if (k == 0) {
      return 1;
   } else {
      int halfPower = raiseIntToPower(n, k / 2);
      int result = halfPower * halfPower;
      if (k % 2 == 1) result *= n;
      return result;
   }
}
```

And if the assignments seem inelegant, here is another coding in a more conventional recursive form:

```
int raiseIntToPower(int n, int k) {
   if (k == 0) {
      return 1;
   } else if (k % 2 == 0) {
      return square(raiseIntToPower(n, k / 2));
   } else {
      return n * square(raiseIntToPower(n, k / 2));
   }
}

/*
 * Function: square
 * Usage: int sq = square(n);
 * --------------------------
 * Returns the square of the argument n.  This function exists only
 * to ensure that the solution is entirely functional, in the sense
 * that it uses no assignment statements.
 */

int square(int n) {
   return n * n;
}
```

Problem 5: Recursive procedures (15 points)

Once again, there are several strategies you might use to solve this problem. The one that probably requires the least amount of new work is to adapt the `listPermutations` code from Chapter 8 to generate all permutations of the domino vector and then see if any of them contain only dominos that match end for end. The following implementation is somewhat more efficient.

```
/*
 * Function: formsDominoChain
 * Usage: if (formsDominoChain(dominos)) . . .
 * -------------------------------------------
 * Returns true if the vector forms a domino chain.  This function is
 * implemented as a wrapper to the method extendsDominoChain.
 */

bool formsDominoChain(Vector<Domino> & dominos) {
   return extendsDominoChain(-1, dominos);
}

/*
 * Function: extendsDominoChain
 * Usage: if (lastDots, dominos) . . .
 * -----------------------------------
 * Checks to see if the domino vector forms a chain, starting
 * with the a domino that matches lastDots; if lastDots is -1
 * (which is used to indicate the first value in a chain), then
 * the next domino can begin with any number of dots.  The
 * recursive insight is that the entire set forms a chain if
 * and only if there is some domino whose left side matches the
 * designated starting number and the remaining dominos form a
 * chain starting with the count from that domino's right side
 * or, conversely, there is a domino that fits if you reverse
 * its left and right sides.
 */

bool extendsDominoChain(int lastDots, Vector<Domino> & dominos) {
   if (dominos.isEmpty()) {
      return true;
   } else {
      for (int i = 0; i < dominos.size(); i++) {
         Domino candidate = dominos[i];
         Vector<Domino> rest = dominos;
         rest.remove(i);
         if (lastDots == -1 || lastDots == candidate.leftDots) {
            if (extendsDominoChain(candidate.rightDots, rest)) {
               return true;
            }
         }
         if (lastDots == -1 || lastDots == candidate.rightDots) {
            if (extendsDominoChain(candidate.leftDots, rest)) {
               return true;
            }
         }
      }
      return false;
   }
}
```