CS106B Spring 2012 | Handout #12S May 2, 2012

Practice Midterm 1 Solutions

Based on a handout by Eric Roberts

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

This problem can be solved either by following through the computation or by figuring out what each function does. In this case, it is easier to undertake the latter approach. The **enigma** function computes the product of **n1** and **n2** by recursively summing **n1** copies of **n2**. The **mystery** function then uses **enigma** to multiply **n** copies of the integer 2. The value of **mystery(n)** is therefore 2^n, which means that **mystery(3)** returns 8.

Computing the complexity order requires noticing that the computation of **mystery(n)** makes **n** calls to **enigma**. Given the coding of **mystery**, each call to **enigma** requires a constant amount of work because the first argument is always 2. The complexity is therefore proportional to **n** times some constant, which is $O(N)$.

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);
 * ----------------------------------------------------------
 * Generates all possible completions from a string of digits
 * and returns the result as a Vector.
 */
Vector<string> generateCompletions(string digits, Lexicon& lex) {
    Vector<string> result;
    foreach (string word in lex) {
        if (matchesDigits(word, digits)) {
            result += word;
        }
    }
    return result;
}

/* Function: prefixMatches
 * Usage: bool prefixMatches(string word, string digits);
 * ----------------------------------------------------------
 * Given a word and a string of digits, returns whether that
 * word could be generated from the given cell phone digits.
 */
bool prefixMatches(string word, string digits) {
    /* If the word is too long, we can immediately report failure. */
    if (word.length() > digits.length()) return false;

    /* Build a map from digits to matching letters. */
    Map<char, string> digitMap;
    digitMap['2'] = "abc";
    digitMap['3'] = "def";
    digitMap['4'] = "ghi";
    digitMap['5'] = "jkl";
    digitMap['6'] = "mno";
    digitMap['7'] = "pqrs";
    digitMap['8'] = "tuv";
    digitMap['9'] = "wxyz";

    /* Scan across the characters of the digits string and confirm each
     * matches the appropriate letter of the word.
     */
    for (int i = 0; i < digits.length(); i++) {
        if (digitMap[digits[i]].find(word[i]) == string::npos) {
            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.removeAt(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;
   }
}
```