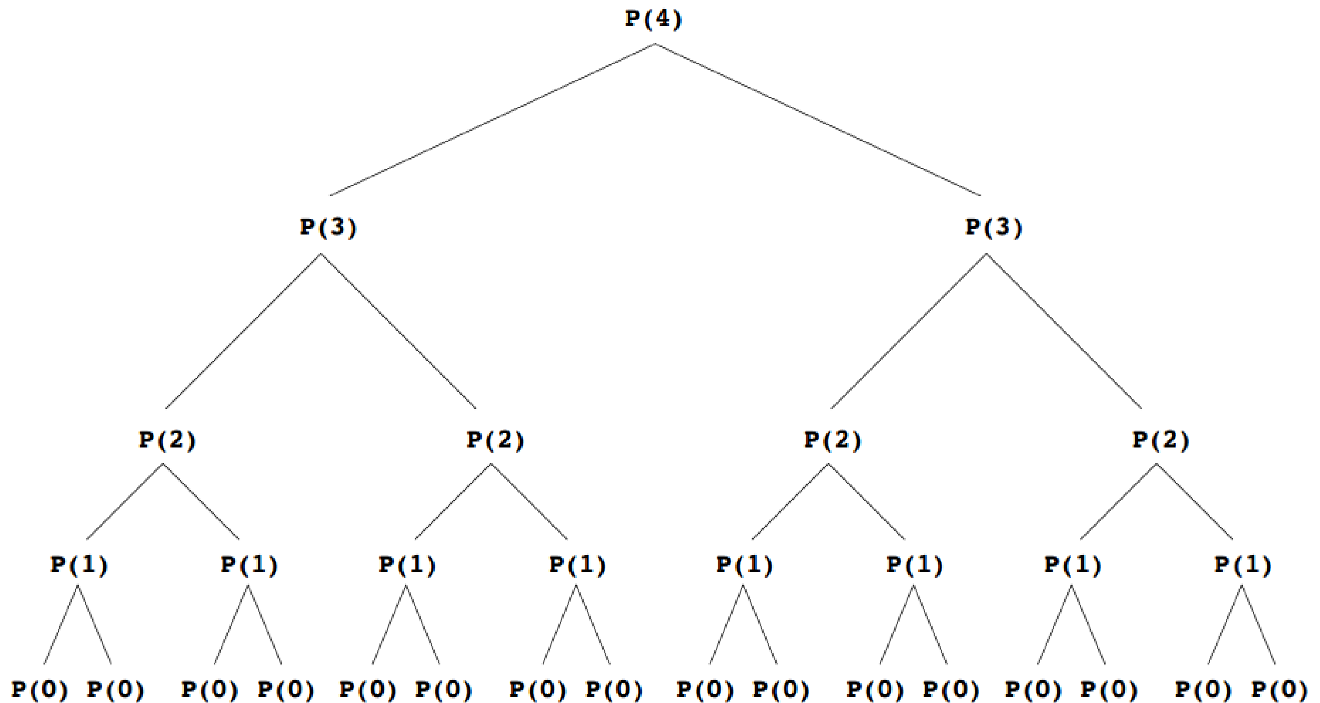
Practice Midterm 1 Solutions

Based on a handout by Eric Roberts

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

If you think about what's happening in the `puzzle` function, it should be clear that the function computes the number of moves required to solve the Tower of Hanoi problem. Thus, the value of `puzzle(4)` is 15. To understand the complexity order of the computation, it helps to draw a tree of the computations involved, which (after abbreviating `puzzle` to `P` to save space) looks like this for `puzzle(4)`:



Each new level doubles the amount of work, so the total amount of work must be $O(2^N)$. Another way to obtain this same result is that the calculation of `puzzle(`*N*`)` requires twice as many additions as the original Tower of Hanoi puzzle requires moves to solve the problem for *N* disks. If Tower of Hanoi is exponential, this function must be as well.

Note that the efficiency is a property of the implementation and not of the underlying mathematical function. If the implementation of `puzzle` were changed to

```
int puzzle(int n) {
   if (n == 0) {
      return 0;
   } else {
      return 2 * puzzle(n - 1) + 1;
   }
}
```

the complexity would be $O(N)$, even though it computes the same value.

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

```
/* Function: reshape
 * Usage: reshape(grid, nRows, nCols);
 * -----------------------------------------------------------------
 * Changes the dimensions of the grid (in the manner of resize) in a
 * way that retains the values of the original elements in the grid.
 * This implementation puts all the old values into a Queue and then
 * fills the new grid with values from the queue.
 */
void reshape(Grid<int>& grid, int nRows, int nCols) {
    Queue<int> values;
    foreach (int value in grid) {
        values.enqueue(value);
    }

    grid.resize(nRows, nCols);

    for (int row = 0; row < nRows; row++) {
        for (int col = 0; col < nCols; col++) {
            if (!values.isEmpty()) {
                grid[row][col] = values.dequeue();
            }
        }
    }
}
```

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

```
/*
 * Function: readSynonymTable
 * Usage: readSynonymTable(infile, table);
 * ---------------------------------------
 * Reads a synonym table from the input stream infile into the map
 * stored in table.  The keys in the map are the words in the data
 * file.  The corresponding value is a lexicon containing all of
 * the synonyms that appear on the same line.
 */

void readSynonymTable(ifstream & infile, Map<string,Lexicon> & table) {
   TokenScanner scanner;
   scanner.ignoreWhitespace();
   while (true) {
      string line;
      getline(infile, line);
      if (infile.fail()) break;
      scanner.setInput(line);
      Vector<string> words;
      while (scanner.hasMoreTokens()) {
         words.add(scanner.nextToken());
      }
      for (int i = 0; i < words.size(); i++) {
         string key = words[i];
         Lexicon lexicon;
         for (int j = 0; j < words.size(); j++) {
            if (i != j) lexicon.add(words[j]);
         }
         table[key] = lexicon;
      }
   }
}
```

Problem 4: Recursive functions (10 points)

```
/*
 * Function: removeDoubledLetters
 * Usage: string shorter = removeDoubledLetters(str);
 * -------------------------------------------------
 * Removes all but the first of a sequence of identical letters from str.
 */

string removeDoubledLetters(string str) {
   if (str.length() <= 1) {
      return str;
   } else if (str[0] == str[1]) {
      return removeDoubledLetters(str.substr(1));
   } else {
      return str[0] + removeDoubledLetters(str.substr(1));
   }
}
```

Problem 5: Recursive procedures (15 points)

```
/*
 * Function: tryAllOperators
 * Usage: tryAllOperators(exp, target);
 *        tryAllOperators(prefix, rest, target);
 * ---------------------------------------------
 * Recursively replaces every ? in the expression by each of the
 * primary arithmetic operators (+, -, *, /).  If the resulting
 * expression evaluates to the target integer, the function
 * prints out the expression string that generated it.  The first
 * version of the function is a simple wrapper for the second,
 * which divides up the string one character at a time, keeping
 * track of the previously considered characters in prefix.
 */

void tryAllOperators(string exp, int target) {
   tryAllOperators("", exp, target);
}

void tryAllOperators(string prefix, string rest, int target) {
   if (rest == "") {
      if (evaluateExpression(prefix) == target) {
         cout << prefix << endl;
      }
   } else if (rest[0] == '?') {
      rest = rest.substr(1);
      tryAllOperators(prefix + "+", rest, target);
      tryAllOperators(prefix + "-", rest, target);
      tryAllOperators(prefix + "*", rest, target);
      tryAllOperators(prefix + "/", rest, target);
   } else {
      tryAllOperators(prefix + rest[0], rest.substr(1), target);
   }
}
```