Midterm practice solutions

Midterm Exam: Tuesday, Feb 19th 7-9pm
Terman Auditorium

We'd love to see everyone Tuesday evening, but if that's impossible for you, e-mail head TA Jason by noon Thursday to arrange an alternate Tuesday time. Remote SCPD students should email Jason by Thursday to initiate arrangements for taking the exam onsite.

1) Version A assembles the result char-by-char using string concatenation in a loop.

```
string ConvertMacLineEndingsToPC(string str)
{
   string result = "";
   for (int i = 0; i < str.length(); i++) {
      if (s[i] == '\n')
         result += "\r\n"
      else
         result += s[i];
   }
   return result;
}
```

Version B does find-replace operations on the original string to produce the result.

```
string ConvertMacLineEndingsToPC(string str)
{
   int pos = 0;
   while ((pos = str.find("\n", pos)) != string::npos) {
      str.replace(pos, 1, "\r\n"); // or str.insert(pos, "\r");
      pos += 2;
   }
   return str;
}
```

2)
```
double Similarity(Vector<char> &v1, Vector<char> &v2)
{
   int pos1 = 0, pos2 = 0, nMatches = 0;
   while (pos1 < v1.size() && pos2 < v2.size()) {
      if (v1[pos1] < v2[pos2]) {
         pos1++;
      } else if (v1[pos1] > v2[pos2]) {
         pos2++;
      } else {
         nMatches++;
         pos1++;
         pos2++;
      }
   }
   double avgSize = double(v1.size()+ v2.size())/2;
   return nMatches/avgSize;
}
```

```
bool MarkovMatch(Map<Vector<char> > &m1, Map<Vector<char> > &m2)
{
   Map<Vector<char> >::Iterator itr = m1.iterator();
   while (itr.hasNext()) {
      string key = itr.next();
      if (m2.containsKey(key)) {
         if (Similarity(m1[key], m2[key]) < .7)
             return false;
      }
   }
   return true;
}
```

3) Changes shown in **bold**:

```
int Permute(string soFar, string rest, Lexicon &lex)
{
   if (!lex.containsPrefix(soFar)) return 0;
   if (rest == "") {
      return (lex.containsWord(soFar) ? 1 : 0);
   } else {
      int count = 0;
      for (int i = 0; i < rest.length(); i++) {
         if (rest.find(rest[i], i+1) == string::npos) { // skip dup char
            string rem = rest.substr(0, i) + rest.substr(i+1);
            count += Permute(soFar + rest[i], rem, lex);
         }
      }
      return count;
   }
```

4)

```
bool CanSpell(string word, Vector<string> & cubes)
{
   if (word == "") return true;

   for (int i = 0; i < cubes.size(); i++) {
      string curCube = cubes[i];
      if (curCube.find(word[0]) != string::npos) {
         cubes.removeAt(i); // remove cube so not used again
         if (CanSpell(word.substr(1), cubes)) {
            cubes.insertAt(i, curCube);
            return true;
         }
         cubes.insertAt(i, curCube);// backtrack, replace cube
      }
   }
   return false;  // trigger backtracking
}
```

5)
```
Queue<int> ExtractStrand(Queue<int> &q)
{
   Queue<int> strand;

   int last = q.dequeue();
   strand.enqueue(last);

   int qsize = q.size();// need to cache since changes inside loop
   for (int i = 0; i < qsize; i++) {
      int cur = q.dequeue();
      if (cur >= last) {
         strand.enqueue(cur);
         last = cur;
      } else
         q.enqueue(cur);
   }
   return strand;
}
```

6) There are many correct variations, here is one recursive and one iterative solution.

```
bool Contains(cellT *list, cellT *sub)
{
    if (sub == NULL) return true;
    if (list == NULL) return false;
    if (list->val == sub->val)
      return Contains(list->next, sub->next);
    else
      return Contains(list->next, sub);
}

// iterate over main list, whenever matches sub, advance sub
bool Contains(cellT *list, cellT *sub)
{
   for (; list != NULL; list = list->next) {
      if (sub == NULL) break;
      if (list->val == sub->val)
         sub = sub->next;
   }
   return (sub == NULL);
}
```

7)
```
void Roll(Stack<char> &s, int nElems, int nPos)
{
    Queue<char> q;
    Stack<char> tmp;
    for (int i = 0; i < nElems; i++)    // remove top n elems
        q.enqueue(s.pop());
    for (int i = 0; i < nPos; i++)      // cycle nPos
        q.enqueue(q.dequeue());
    while (!q.isEmpty())                // flip around
        tmp.push(q.dequeue());
    while (!tmp.isEmpty())              // put back on stack
        s.push(tmp.pop());
}
```

8a) `Winky` is $O(2^N)$. The recurrence is T(n) = T(n-1) + T(n-2) + T(n-3) + … + T(2) + T(1) + T(0). Repeatedly substituting to expand and grouping terms will allow you to see the doubling pattern that arises. Another strategy is to notice that T(n-2) + T(n-3) + …+ T(1) + T(0) is T(n –1) and reverse-substitute to get T(n) = 2T(n-1) which is the Towers of Hanoi recurrence that solves to $O(2^N)$.

8b) As is, the code prints "cucumber" because the salad stack is unaffected by the call. If the stack is instead passed by reference, the code will print "lettuce" because the Toss function reversed the stack contents.

8c) Selection Sort. It does many comparisons to determine the remaining max, but does at most N-1 swaps total, in contrast to insertion sort, which shuffles each student down the hall into the correct spot, on average requiring N/2 moves per student or $N^2/2$ total moves.