Eric Roberts
CS106B

Handout #54A
June 1, 2009

Answers to Practice Final Examination #2

1. Simple algorithmic tracing (5 points)

STAN, SRI, UTAH, BBN, CMU, NRL, HARV, MIT, RAND, UCLA

2. Recursion (15 points)

```
/*
 * Implementation notes: PuzzleIsSolvable
 * --------------------------------------
 * This level tries every possible first piece (which must have a
 * flat side) and then calls RecIsSolvable to do the real work.
 */

bool PuzzleIsSolvable(Set<PuzzlePiece> & puzzle) {
    if (puzzle.isEmpty()) return true;
    foreach (PuzzlePiece p in puzzle) {
        Set<PuzzlePiece> copy = puzzle;
        copy.remove(p);
        if (p.hasFlatLeftSide()) {
            if (RecIsSolvable(p, copy)) return true;
        }
        p = p.flip();
        if (p.hasFlatLeftSide()) {
            if (RecIsSolvable(p, copy)) return true;
        }
    }
    return false;
}

/*
 * Implementation notes: RecIsSolvable
 * -----------------------------------
 * This function checks whether it is possible to solve the rest
 * of the puzzle starting with a piece that attaches to currPiece
 * which is the final piece in the chain.
 */

bool RecIsSolvable(PuzzlePiece & currPiece, Set<PuzzlePiece> & puzzle) {
    if (puzzle.isEmpty()) return currPiece.flip().hasFlatLeftSide();
    foreach (PuzzlePiece p in puzzle) {
        Set<PuzzlePiece> copy = puzzle;
        copy.remove(p);
        if (currPiece.attachesTo(p)) {
            if (RecIsSolvable(p, copy)) return true;
        }
        p = p.flip();
        if (currPiece.attachesTo(p)) {
            if (RecIsSolvable(p, copy)) return true;
        }
    }
    return false;
}
```

3. Linear structures and hash tables (15 points)

```
/*
 * Implementation notes: rehash
 * ----------------------------
 * This method begins by copying the old bucket array into a
 * temporary variable so that it can cycle through the old lists.
 * For each of the old variables, it simply calls put to update
 * the new table.
 */

template <typename ValueType>
void Map<ValueType>::rehash(int nBuckets) {
    int oldBucketCount = this->nBuckets;
    cellT **oldBuckets = buckets;
    this->nBuckets = nBuckets;
    nEntries = 0;
    buckets = new cellT *[nBuckets];
    for (int i = 0; i < nBuckets; i++) {
        buckets[i] = NULL;
    }
    for (int i = 0; i < oldBucketCount; i++) {
        for (cellT *cp = oldBuckets[i]; cp != NULL; cp = cp->link) {
            put(cp->key, cp->value);
        }
    }
    delete[] oldBuckets;
}
```

4. Function pointers (10 points)

The first version of the practice exam handout omitted the `&` in the parameter declaration. Leaving it out doesn't change the solution but does make it less efficient.

```
/*
 * Function: LongestKey
 * Usage: string key = LongestKey(map);
 * ------------------------------------
 * This function returns the longest key in the map.
 */

string LongestKey(Map<string> & map) {
    string longest = "";
    map.mapAll(UpdateLongestKey, longest);
    return longest;
}

void UpdateLongestKey(string key, string value, string & longest) {
    if (key.length() > longest.length()) {
        longest = key;
    }
}
```

5. Trees (15 points)

The sneaky (but perfectly correct) way to implement this method looks like this:

```
bool ExpMatch(expressionT e1, expressionT e2) {
    return e1->toString() == e2->toString();
}
```

The code in the box below represents the more typical solution:

```
bool ExpMatch(expressionT e1, expressionT e2) {
    if (e1->getType() == e2->getType()) {
        switch (e1->getType()) {
          case ConstantType:
            ConstantNode *c1 = (ConstantNode *) e1;
            ConstantNode *c2 = (ConstantNode *) e2;
            return c1->getValue() == c2->getValue();
          case IdentifierType:
            IdentifierNode *id1 = (IdentifierNode *) e1;
            IdentifierNode *id2 = (IdentifierNode *) e2;
            return id1->getName() == id2->getName();
          case CompoundType:
            CompoundNode *x1 = (CompoundNode *) e1;
            CompoundNode *x2 = (CompoundNode *) e2;
            return x1->getOp() == x2->getOp()
                   && ExpMatch(x1->getLHS(), x2->getLHS())
                   && ExpMatch(x1->getRHS(), x2->getRHS());
        }
    }
    return false;
}
```

6. Graphs (15 points)

```
/*
 * Implementation notes: IsPartOfCycle
 * Usage: if (IsPartOfCycle(np)) . . .
 * -----------------------------------
 * Checks whether the node is part of a cycle.  The easiest way
 * to write this program is to use the PathExists method from
 * Section #8.  A node is part of a cycle if a path exists from
 * any of its successors back to the node.
 */

bool IsPartOfCycle(nodeT *np) {
    foreach (arcT *ap in np->arcs) {
        if (PathExists(ap->to, np)) return true;
    }
    return false;
}

/*
 * Include the definitions of PathExists and RecPathExists from
 * Handout #52A.
 */
```