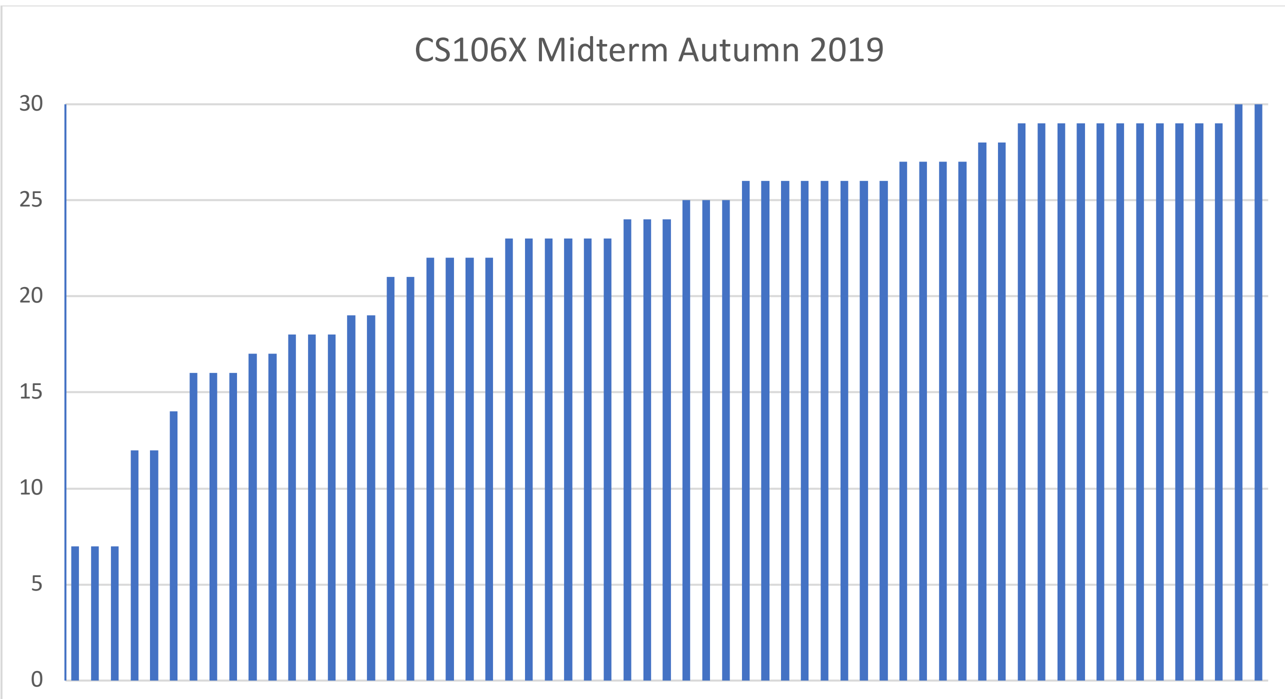
CS106X Midterm Examination Solution

The CS106X midterms are graded, and we'll release grades on Gradescope sometime today if we haven't already by the time you read this handout.

I thought the exam was challenging, but the majority of you pretty much nailed it. The median grade was a 24 out of 30, and the standard deviation was just a smidge below 6.0. Here's a graph of how everyone did:



Each vertical bar represents a single exam, and scores ranged from 7 to a perfect 30.

If you didn't do as well as you hoped to, you're more than welcome to come chat during Jerry office hours. And even though we expect the second midterm to be equally (if not more) challenging, understand that the two midterms test very different topics, so it's very possible to recover from a poor first midterm score and do well on the second.

The rest of this handout includes my own solutions and the criteria used to grade your work. We understand the exams count for a large portion of your grade, so we try to be as transparent as possible about the criteria. If you have a legitimate concern about how your midterm was graded, come talk to Jerry during his office hours.

All regrade requests must come in by November 15th. After that, Midterm 1 grades are frozen.

Solution 1: Prefix to Postfix

```
/*
 * Function: prefixToPostfix
 * -------------------------
 * Converts single digit prefix expressions (e.g. "*-2+46-+259") to postfix
 * expressions (e.g. "246+-25+9-*"), as outlined in the midterm itself.
 *
 * My own solution retains some of the error checking I did while
 * constructing my own solution.  You, however, didn't need to include
 * error checking in your own solution.
 */
static string prefixToPostfix(const string& prefix) {
   Stack<string> partials;
   for (int i = prefix.size() - 1; i >= 0; i--) {
      string ch = prefix.substr(i, 1);
      assert(ch.size() == 1);
      if (isdigit(ch[0])) {
         partials.push(ch);
      } else {
         if (partials.size() < 2) error("Invalid prefix expression.");
         string first = partials.pop(); // first operand read after second
         string second = partials.pop();
         partials.push(first + second + ch);
      }
   }
   if (partials.size() != 1) error("Invalid prefix expression.");
   return partials.pop();
}
```

Criteria for Problem 1: 10 points

- Correctly traverses from back to front as instructed: **2 points**
- Properly extract each character, either as a **`char`** or a **`string`**, and properly dispatches on whether it's a digit string or an operator: **2 points**
- Properly pushes strings on to the stack, even if digits are temporarily manipulated as **`char`**s or **`int`**s: **1 point**
- Properly materializes the correct postfix subexpression out of the most recently read operator and the two most recently scanned prefix subexpressions: 3 points
    - Pops the two items off the stack: **1 point**
    - Concatenates the two popped items in the correct order: **1 point**
    - Pushes proper concatenation back onto the stack: **1 point**
- Recognizes that, post for loop, that the top item of the stack contains the postfix realization of the original prefix expression, and returns it: **2 points**

Solution 2: Magic Numbers

Because the numbers of interest are 10 digits long—more specifically, the numbers are larger than what can be stored in a variable of type **`int`**—the data type should really be a **`long long`**. I didn't want to complicate the problem narrative, so I described the problem in terms of **`int`**s anyway. The below solution uses **`long long`**s, instead of **`int`**s, and **`stoll`** (for **`string`** to **`long long`**) instead of **`stringToInteger`**, so that it compiles and runs as expected. You, of course, were permitted to frame your implementation around **`int`**s and **`stringToInteger`**.

```
/*
 * Recursive function: generateMagicNumbers
 * ----------------------------------------
 * Works to generate all permutations of the first ten digits, except
 * that it prunes those when the relevant prime number fails
 * to perfectly divide into the working permutations last three digits.
 */
static const Vector<int> kPrimes({2, 3, 5, 7, 11, 13, 17});
static void generateMagicNumbers(Set<long long>& numbers,
                                 const string& committed, const string& rest) {
   if (committed.size() > 3 &&
       stoll(committed) % 1000 % kPrimes[committed.size() - 4] > 0) return;

   if (rest.empty()) {
      numbers.add(stoll(committed));
      return;
   }

   for (int i = 0; i < rest.size(); i++) {
      generateMagicNumbers(numbers, committed + rest[i],
                           rest.substr(0, i) + rest.substr(i + 1));
   }
}

static void generateMagicNumbers(Set<long long>& numbers) {
   generateMagicNumbers(numbers, "", "1234567890");
}
```

Criteria for Problem 2: 10 points

- Passes **`numbers`** through at every level of the recursion, by reference, so as to collect all of the magic numbers as they are discovered (or some equivalent): **1 point**
- Understands the need to pass the string **`"1234567890"`** (or some permutation of it) alongside the empty string to make the implicitly understood explicit to the recursion: **1 point**
- Otherwise subscribes to and properly codes to the same recursive formulation as the traditional substructure for permutations, as outlined in lecture, but with more elaborate base cases: **2 points**
- Properly materializes current number modulo 1000 for **`committed`** strings larger than 999: **1 point**
- Properly identifies the entry within **`kPrimes`** relevant to the call's modulo 1000 number: **1 point**

- Correctly uses `%` to test whether relevant prime factor divides into that modulo 1000 number: **1 point**
- Returns if it isn't, and continues if it is: **2 points**
- Detects the situation where **`rest`** is empty, adds the number to the references set provided all divisibility requirements have been met, and then returns: **1 point**

Problem 3: Derivability

```
/*
 * Recursive Function: cbg
 * -----------------------
 * cbg (short for can be generated) searches the provided grammar to see
 * whether or not the supplied string to be matched (called tbm,
 * for to be matched) can be generated from the supplied working
 * production (called wp, for working production).
 *
 * If the working production doesn't contain any nonterminals,
 * then we immediately return true if and only if the working production
 * has evolved to perfectly match tbm, and false otherwise.
 *
 * If wp does contain one or more nonterminals, we confirm that
 * everything up through the opening '<' of the working production
 * matches the beginning of tbm, else we return false without continuing.
 *
 * If it does match, we frame the original problem in terms
 * of new working productions where the first nonterminal is
 * replaced by each of its possible expansions.  If any one
 * of them recursively works out, we return true.  If none of them work
 * out, we ultimately return false.
 */
static bool cbg(Map<string, Vector<string>>& g,
                const string& tbm, const string& wp) {
   int start = wp.find('<');
   if (start == string::npos) return wp == tbm;

   string prefix = wp.substr(0, start);
   if (!startsWith(tbm, prefix)) return false;

   string suffix = tbm.substr(prefix.length());
   int end = wp.find('>', start + 1);
   string nonterminal = wp.substr(start, end - start + 1);
   string rest = wp.substr(end + 1);

   const Vector<string>& productions = g[nonterminal];
   for (const string& production: productions) {
      if (cbg(g, suffix, production + rest)) return true;
   }

   return false;
}

static bool cbg(Map<string, Vector<string>>& g, const string& sentence) {
   return cbg(g, sentence, "<start>");
}
```

Criteria for Problem 3: 10 points

- Introduces **`"<start>"`** as a third parameter into a wrapping function, as per the problem statement: **1 point** (allow for the possibility that they managed this differently)
- Properly identifies the base case scenario where a decision **must** be made by searching for a nonterminal and failing to find one: **1 point**
- Properly separates the prefix of the working production from everything else: **1 point**
- Properly identifies the base case scenario where the working production and the string to be matched don't share (and therefore never share) the same prefix up to the opening **`'<'`**, and returns **`false`**: **1 point** (this isn't technically necessary for a functionally correct solution, but to not include it is to allow the grammar to expand to every single sentence and see if one incidentally matches the one to be matched. That's so intractably time consuming compared to the solution above that it's worth a point. That's the purpose of backtracking—to prune the search as aggressively as possible.)
- Properly tracks portion of original sentence that's been matched versus what still remains to be matched by the working production: **1 point** (I manage this by ignoring the matched prefixes for all recursive calls and pretending they were never part of the original sentence)
- Properly separates the leading nonterminal from everything after it, so that the first nonterminal can be replaced by any of its expansions: **1 point**
- Properly recurses in a for loop, where each iteration considers each expansion in turn: **1 point**
- Properly synthesizes the recursive call's arguments, and makes the call: **1 point**
- Properly responds to a recursive call's return by returning **`true`** if the call returned true, and otherwise allowing for loop to continue: **1 point**
- Properly returns **`false`** only after every single path has been explored without success: **1 point**