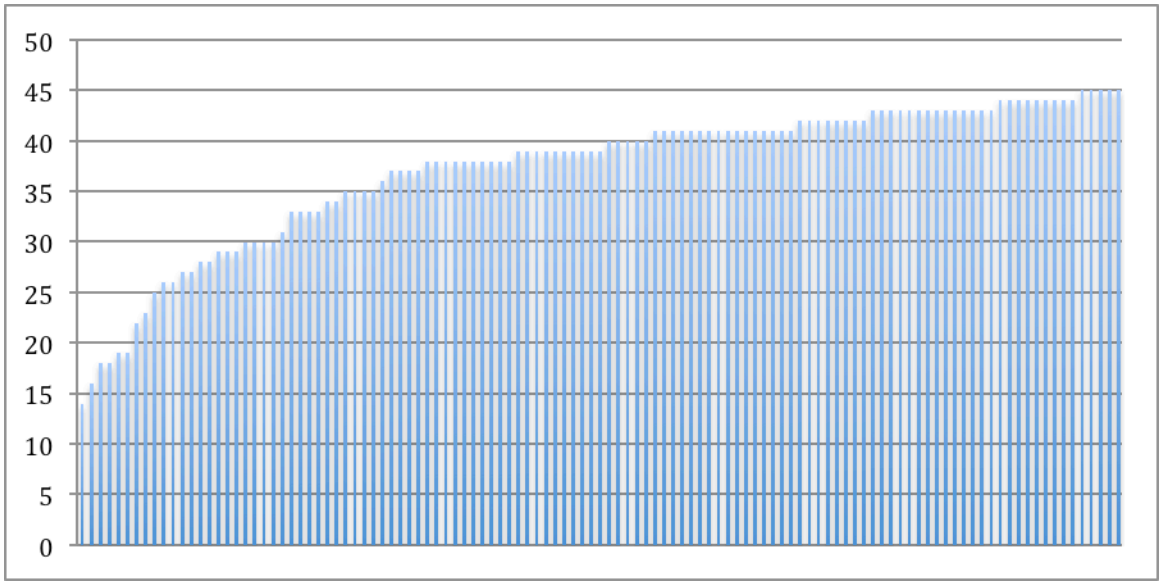
CS106X Midterm Examination Solution

Thanks to TA Ben Holtz and the gaggle of CS106X section leaders, your midterms are graded and are available for pickup. I'll bring the exams to Monday morning lecture, and those who can't make lecture should just come by my office during office hours to retrieve your own (and only your own, unless I get an email from your friend or dorm mate saying it's okay to pick up his or her exam too).



The height of each bar in the above graph is the score that some student got on the exam, so the chart should give you a sense as to how you did relative to everyone else. The median grade was a 39 out of 45 (wow!), and the average grade was 36.9, with a standard deviation of 7.9. The exam is a only a small portion of your grade, but if you did poorly and you want to protect against a similar experience on the second exam, then feel free to come by my office hours to discuss a strategy for doing better.

If you notice a grading error, then you need to submit your exam back to me (Jerry) some time in the next two weeks (before November 9th) so I can take a look at it. All regrade requests must go through Jerry.

Solution 1: Stepping Stones

Because the exam was closed note, closed book, closed everything, I thought it was fair to test your understanding of the very algorithm backing your **word-ladder** submission for Assignment 2. By doing so, I got to exercise your understanding of algorithms and your ability to express that understanding in terms of off-the-shelf data structures. The primary algorithmic complexities (beyond the breadth-first search itself) were:

- producing the correct test to see if a solution has been found
- producing the logic to keep play on the grid and block out U-turns
- encoding the logic needed to decide whether a color change was required or forbidden with any given move

The second bullet item was obtuse enough that I went with a helper function to improve the narrative of the overall algorithm and to isolate the logic to a context where it was easier to get it right. You'll notice that I didn't bother preventing loops, because I knew that allowing them was only an efficiency concern, not a functionality one. I was looking for a clear understanding of the algorithm and fluency with all of the relevant ADTs.

```
static void generatePath(Grid<string>& stones,
                         const stone& start, const stone& finish,
                         Vector<stone>& shortest) {

    Vector<stone> startPath;
    startPath += start;
    Queue<Vector<stone> > partialPaths;
    partialPaths.enqueue(startPath);

    while (true) {
        shortest = partialPaths.dequeue();
        const stone& last = shortest[shortest.size() - 1];
        if (last == finish && shortest.size() % 3 == 2) return;
        for (Direction dir = NORTH; dir <= WEST; dir++) {
            stone neighbor = getNeighboringStone(last, dir);
            if (hopIsAllowed(stones, shortest, neighbor)) {
                Vector<stone> clone = shortest;
                clone += neighbor;
                partialPaths.enqueue(clone);
            }
        }
    }
}
```

I recognize that statements like **end == finish && shortest.size() % 3 == 2** don't just roll off the pencil during a timed exam, but you should have recognized that the details of stone hopping could be largely abstracted away into a helper function or two so that the top-level algorithm is unambiguously breadth-first search. The many details managed by my **hopIsAllowed** predicate function are pretty crunchy, which is why I packaged those details into a nicely named routine that reads well within the larger algorithm.

```
static bool hopIsAllowed(Grid<string>& stones, Vector<stone>& shortest,
```

```
                        const stone& neighbor) {
    if (!stones.inBounds(neighbor.row, neighbor.col) ||
        (shortest.size() > 1 && neighbor == shortest.get(shortest.size() - 2))) {
       return false;
    } // out of bound for a U-turn? reject

    const stone& end = shortest[shortest.size() - 1];
    const string& beforeColor = stones[end.row][end.col];
    const string& afterColor = stones[neighbor.row][neighbor.col];
    return (beforeColor == afterColor) == (shortest.size() % 3 != 1);
}
```

Problem 1 Criteria: 15 points

Crucial Design Considerations

- Understands that a **`Queue`** is the best data structure to manage the breadth-first search, and conveys that understanding by declaring one: 1 point
- Models the partial path as either a **`Vector`** or a **`Stack`**, since the algorithm only needs to do surgery on the last/top two elements with any given iteration: 1 point

Algorithmic Issues

- Correctly seeds the **`Queue`** with a singleton **`Vector`** containing just **`start`** (or **`finish`**, if they search backwards): 2 points
- Correctly loops tirelessly until a solution is found (or until the queue is empty, even though I guaranteed that a solution always exists): 1 point
- Extracts the best partial path that should be manipulated at that moment: 1 point
- Properly surfaces the last stone so it can be examined for its coordinate and its color: 1 point
- Correctly detects (either before it's enqueued or the instant it's dequeued) whether a partial path is actually the path of interest: 2 points
- Iterates away from the current stone in all four directions using some reasonable idiom: 1 point
- Detects whether the next possible step is valid: 3 points (1 point for a well-placed inbounds check, 1 point for rejecting U-turns, and 1 point for correctly allowing or forbidding a color change)
- Properly clones the partial path when it can be legally extended, extends it, and then enqueues it: 2 points

Solution 2: Multitions

Most people saw the recursive structure perfectly well, but many—at least many of the twentysomething exams I looked over to see how everyone did—fumbled with the **Stack**, imitating too closely the manner in which a stack accumulates information during recursive backtracking (and recursive backtracking this is **not**).

The recursive variable was indeed **count**, and the recursive structure was such that a multition with **count** multiplications is some nonempty prefix of the number multiplied by the multition of the complementary suffix with **count - 1** multiplications. Of course, there are many ways to pull off a prefix—length 1, length 2, and so forth, so we need to recursive generate multitions for all possible prefix/suffix subdivisions.

```
static int optimalMultition(int number, int count, Stack<int>& factors) {
   if (count == 0) {
      factors.push(number);
      return number;
   }

   int best = 0;
   string str = integerToString(number);
   for (int len = 1; len < str.size(); len++) {
      int first = stringToInteger(str.substr(0, len));
      int rest = stringToInteger(str.substr(len));
      Stack<int> factorsOfRest;
      int product = first * optimalMultition(rest, count – 1, factorsOfRest);
      if (product > best) {
         best = product;
         factors = factorsOfRest;
         factors.push(first);
      }
   }

   return best;
}
```

Note that an empty stack is passed to every single recursive call, allowing it to accumulate the ordered subdivisions of the number being multitioned. Unless the product of those subdivisions is better than anything we've seen before, we discard the stack entirely. When the product is better, we copy the stack it into the space addressed by **factors** and press the leading factor on top of it. In the process, each recursively populated stack competes with previously populated ones and displaces the best previous one if it's even better. After all prefix/suffix subdivisions have been considered, **factors** references the sequence of factors in the multition, and **best** stores their product.

Problem 2 Criteria: 15 points

Recursion Issues [Ignoring Stack]

- Properly identifies the **`count == 0`** situation as a base case, and returns the number unmodified: 1 point
- Employs a valid technique for partitioning the digits on an integer into a nonempty prefix and a nonempty suffix (most likely using **`stringToInteger`** and **`integerToString`**): 4 points
    - Iterates over all valid division points: 2 points
    - General substringing strategy is solid, minus edge cases: 1 point
    - Manages edge cases properly as well: 1 point
- For each prefix, properly recurs on the integer form of the suffix: 3 points
    - Passes in the integer form of the suffix: 1 point
    - Passes in a decremented count as the second argument: 1 point
    - Catches the return value for comparison with a running best: 1 point
- Allocates and initializes **`best`** to be 0 as the default return value when there are no valid multitions: 1 point
- Updates **`best`** as recursively computed results are detected to be even better: 1 point (don't worry about returning it—easy to forget and not algorithmically interesting)

Accumulation of Factors in `Stack<int>`

- In the **`count == 0`** scenario, presses the number unmodified onto the referenced stack: 1 point
- Allocates an empty stack (or clears an existing one) and passes it to the recursive calls: 2 points
- Every update of best is accompanied for an **`operator=`** overwrite of the stack referenced by **`factors`**: 1 point
- Presses the integer form of a prefix onto the top of the **`factors`** stack (or on top of **`factorsOfRest`** before it's replicated into the space referenced by **`factors`**): 1 point

Solution 3: Valency

There was ambiguity as to whether or not **getCandidates** could tell whether a particular 0 in the **valencies** grid was always a zero or whether it was previously positive but was demoted 0 during the search for a solution. I didn't catch the ambiguity until after about 5 students took the exam, so to be fair to those who took the exam early I need to operate like **getCandidates** could tell the difference and that it always (magically) returns the correct **Set** of circle coordinates needed to discover a solution when and only when it actually exists.

```
static bool findCircle(Grid<int>& valencies, int& row, int& col) {
   for (row = 0; row < valencies.numRows(); row++) {
      for (col = 0; col < valencies.numCols(); col++) {
         if (valencies[row][col] > 0) {
            return true;
         }
      }
   }

   return false;
}

static bool solve(Grid<int>& valencies, Map<connection, int>& connections) {
    int row, col;
    if (!findCircle(valencies, row, col)) return true;

    coord location = {row, col};
    Set<coord> candidates = getCandidates(location, valencies);
    valencies[row][col]--;
    foreach (coord candidate in candidates) {
        valencies[candidate.row][candidate.col]--;
        if (solve(valencies, connections)) {
            connection conn = { location, candidate };
            connections[conn]++;
            return true;
        }
        valencies[candidate.row][candidate.col]++;
    }
    valencies[row][col]++;
    return false;
}
```

My solution above doesn't use the **computeValencySum** function, but another solution I wrote did, and you may have used it if you structured your recursion differently than I did.

Now, some of you know that **getCandidates** can't *magically* tell whether a 0 at any particular location was original or was positive at some point and just decremented down to 0 through a stack of recursive calls. The A+ fix (and we didn't require this of you, since the problem statement wasn't clear, in my opinion) was to pass through not only the working state of the **valencies** grid, but to also pass through the original one, using the original to find candidate circles and then confirming each candidates has some remaining valency before involving it in a recursive call. If you recognized this, then that's really, really impressive! Here's the solution that correctly manages this new information:

```
static bool solve(Grid<int>& originals, Grid<int>& valencies,
                  Map<connection, int>& connections) {
   int row, col;
   if (!findCircle(valencies, row, col)) return true;

   coord location = {row, col};
   Set<coord> candidates = getCandidates(location, originals);
   valencies[row][col]--;
   foreach (coord candidate in candidates) {
      if (valencies[candidate.row][candidate.col] == 0) continue;
      valencies[candidate.row][candidate.col]--;
      if (solve(originals, valencies, connections)) {
         connection conn = { location, candidate };
         connections[conn]++;
         return true;
      }
      valencies[candidate.row][candidate.col]++;
   }
   valencies[row][col]++;
   return false;
}

static bool solve(Grid<int>& valencies,
                  Map<connection, int>& connections) {
    Grid<int> originals = valencies;
    return solve(originals, valencies, connections);
}
```

Problem 3 Criteria: 15 points

- Understands to search for the first coordinate with a positive valency: 1 point
- Correctly detects a zero valency sum as the base case: 1 point
- For that base case, returns **`true`** to reflect the fact that the valency puzzle is trivially solvable: 1 point
- Constructs a **`coord`** for that **`row`**-**`col`** coordinate: 1 point
- Calls **`getCandidates`** to (magically) get the collection of nonzero-valency circles that the circle at the supplied **`coord`** can potentially connect to: 1 point
- Correctly demotes the valency of the central coordinate being manipulated, and eventually promotes it if none of the recursive calls work out: 1 point
- Properly uses **`foreach`** to consider each of the possible candidates in the **`Set<coord>`**: 1 point
- For each of the **`coord`**s in the **`Set`**, levies the **`--`** before and the **`++`** after the recursive call: 1 point
- Properly makes the recursive call and acts on its return value: 2 points
- If the recursive call returns **`true`**, the implementation ensures the relevant **`connection`** is added to the **`Map`** referenced by **`connections`**, and that the value attached to the **`connection`** is either a newborn 1, or one more than it was previously: 1 point
- If the recursive call returns **`true`**, the implementation returns **`true`** without further search: 1 point

- If the recursive call returns **`false`**, then the **`Map`** referenced by **`connections`** is unchanged (and if it was changed before the recursive call, then that change is erased **completely**): 1 point
- If the recursive calls returns **`false`**, then the implementation just advances to the next option: 1 point
- Returns **`false`** only after all options have been recursively explored and nothing worked out: 1 point