Coding Manual for Programming Problems

(for SIGCSE 2016 paper "Modernizing Plan Composition Studies, by Fisler, Krishnamurthi, and Siegmund)

Most problems in this study have

- a "parsing" component (in which the student could reasonably change the data representation of the input to one better suited to the problem),
- a "cleaning" component (in which noisy or irrelevant data gets skipped), and
- a "main" component that performs the core computation of the problem (assuming clean, parsed data).

To give us useful measures across the problems, we record how the student composed and handled each of these components; for each problem, we add extra coding bits for details/tasks specific to that problem.

Per-problem definitions of parsing/cleaning/main components

Palindrome:

- mark "parsing" as occurring if the student converts the string into a list
- "cleaning" has three parts: removing punctuation, ignoring whitespace, and downcasing.
- "main" is the comparison of the forward/reversed strings

Sum Over Table:

- neither "parsing" nor "cleaning" arise much here. Pretty much everything is in main.

Adding Machine:

- "parsing" would occur if the student reformatted the data as a list of lists (taking out the intermediate 0s)
- "cleaning" occurs when the student truncates at the 00
- "main" includes the summing of sublists and the construction of the list of sublist sums

Rainfall:

- "cleaning" occurs if the student removes negative and/or the sentinel before computing the average
- "main" includes computing the average

The Scoring Spreadsheet

A blank template for work is in the same source location as this coding manual. The template has one sheet for each problem.

Many columns are common to all problems. These column descriptions are as follows:

A-E : record the school and course in which the data was gathered, the id tag for an individual student, which programming language was used and which problem this row is marking codes for. These columns allow data from multiple problems or courses to be combined into a single document for analysis in a stats package.

F: the structure of the solution. This is a regexp over the characters P, C, and M (for parsing, cleaning, and main, respectively). The regexps use semicolon to indicate separate loops or functions. For example, a code of "CP;M" indicates that the student had one loop/function that handed cleaning and parsing together, followed by a loop that did the main computation for that problem. "P;C;M" had three loops, one for each phase.

Parsing and Cleaning can appear twice if the student did parts of these tasks in multiple loops/functions. For example, a student could parse both before and after cleaning, resulting in "P;C;P;M". Or a student could handle case and punctuation (both cleaning) separately in Palindrome, resulting in "P;C;C;M".

Use alphabetical order when two components appear in the same loop (so CMP rather than MCP for a single loop that interleaves all parts).

G: A correctness rating, from 0 (no code), 1 (something relevant but not close), 2 (mostly right), 3 (appears correct). This is a rough guide, not an exact correctness score. A solution that looks right but actually had a small error can be coded as 3 without hurting the level of analysis planned for this project. If a quick eyeball spots an error in largely correct code, use 2.

H-J: how did the student implement each component? If they used a built-in iterator or function, name it. If they wrote their own loop, write "own". Write "none" if the student did not implement the component at all (blank, in contrast, will be for students who skipped the entire problem).

K: how many test cases they wrote for the full problem. Do not include test cases for helper functions.

L: did they test the base case (empty data)

Starting at **M**: columns for test cases that are interesting for each problem (such as testing for no data remaining after cleaning). These column headings will differ per problem

After that: Columns for other interesting traits of each problem (such as how they implemented the reversal task for palindrome). Columns can be added here as needed.

Last column: unstructured notes to self. Use to flag anything interesting or solutions we may want to return to later.

Coding Palindrome

In the Structure column (F), use one of the following:

- Mequal if both case/normalization and filtering happen prior to the equality check.
- Mboth if both case/normalization and filtering happen as part of the loop that does the equality check
- Mcase if filtering happens before the equality check but normalization happens with the equality check
- Mskip if normalization happens before the equality check but filtering happens within the equality check

In the MainIndexing column (S), record:

- y(es) if the main equality check uses indices into string/lists
- char-level if the main equality check does char-level comparisons, but is not using indices (such as a Racket loop over the string represented as a list of chars)
- n if the main equality check directly compares or iterates over the data without using indices to access individual chars

The MissCleaning column (P) records which kinds of cleaning (if any) the solution omits. Use "caps", "punc", "white" for each of capitalization, punctuation, and whitespace.

The NumParses column (Q) summarizes how many times the code changes the representation of the data (between strings and lists, for example)

Coding Sum Over Table

In the Structure column (F), use one of the following:

- NL if they wrote nested loops. If the inner loop is pulled out into a separate function (that is called within the outer loop, but has its own name), put "y" in the "SepRowFunction" column.

- accum if they iterate over the table and maintain a data structure with the running collection of maxes, then have a separate (non-nested) loop to sum the maxes.

- SL if they wrote only one loop (and never handle the second loop in any fashion)

For the MaxImpl column (H) and SumImpl column (I), record any built-in functions that they used to implement these tasks (such as using sort to get to the max element).

Coding Adding Machine

In the Structure column (F), use one of the following:

- AccumSum if they traverse the data once, accumulating the sum of each sublist in a parameter/variable as they go, but do not accumulate the result list. When single-zero encountered, accumulated sum is inserted into output list and running sum is reset to 0. Expect recursive calls to be in tail position other than those made when a single-zero has been encountered.

- AccumSublist if they traverse the data once, accumulating the current sublist in a parameter/variable as they go, but do not accumulate the result list. When single-zero encountered, sum of accumulated sublist is computed and inserted into output list; running sublist is reset to empty. Expect recursive calls to be in tail position other than those made when a single-zero has been encountered.

- AccumResult if they traverse the data once, accumulating the list of sums of sublists that have already been processed in a parameter/variable; as each new element is encounted, solution updates the contents of the current sublist sum within the accumulated output list. Expect recursive calls to be in tail position.

- AccumBoth if they traverse the data once, accumulating both the sum of the current sublist and the list of sums of sublists that have already been processed, each in its own parameter or variable. When single-zero encountered, push running sum into sums of sublists and start afresh. Expect recursive calls to be in tail position.

- OnePassModOutput if they traverse the data once, have no accumulator parameters/variables, and integrate the current element by modifying the result obtained on the rest of the elements (i.e., adding the current element to the first sum in a recursively-computed output list). When single-zero encountered, return the result of the function on the rest of the input. Recursive calls will (almost) never be in tail position.

- OnePassModInput if they traverse the data once, have no accumulator parameters/variables, and integrate the current element by combining it with the next element in the list (i.e., recur on an input list that has the sum of the first and second elements as the first element, followed by all-but-the-first-two elements of the given input list). This solution will check for the second element being 0 in order to decide whether a sublist has ended. Such solutions may produce the wrong answer if a sublist sums to 0. Expect recursive calls to be in tail position other than those made when a single-zero has been encountered.

- NL if they have nested loops/functions, one that sums a sublist inside one that builds the list of the sums. There may be multiple traversals over the data for each sublist (i.e., one to sum and one to remove the current sublist from the remaining data). The outer loop will not have recursive calls in tail position.

- Parse if they parse/split the input list into sublists in one loop, then separately process the sublists to create list of the sums

- Trunc; if they first look for and pull out the entries up to 0 0; follow ";" with whatever the rest of the processing looks like. If the truncation is done after, write ";trunc". [NB: It's hard to do it after _correctly_, because a segment with sum 0 does not imply it was from a 0 0: it could be a non-trivial segment that summed to 0.]

If none of these apply, mark in the notes column

In the SumImpl column (H) and MainListImpl column (I), record any built-in operators they use for these tasks. Write "none" if that part is missing entirely. If this is blank, will assume that they wrote their own loop or function to handle this.

The TestBase column (K) covers testing an input list that starts with 00.

The TestZeroSum column (L) covers testing an input list in which a sublist sums to 0 (through adding pos/neg nums)

Coding Rainfall

In the Structure column (F), use one of the following:

- SingleLoop: one traversal over data that includes cleaning, summing, and counting. Average done at end of loop.

- CleanFirst: at least one of removing negatives or truncating at the sentinel is done in a separate traversal prior to summing/counting

- CleanMult: one traversal to sum, one to count, each include cleaning
- NoCleaning: just includes sum/count/average. Assumes no negatives or sentinel
- None: nothing (or close to nothing) relevant
- Other: for stuff that doesn't fit -- describe in Notes column

Correctness column (G):

- 3: seems correct
- 2: straightforward errors that testing would have caught (such as checking positive instead of non-negative, missing base case)
- 1: noticeable logical errors. Got somewhere, but lots of problems
- 0: nothing relevant to problem or very far off

In the LoopingImpl column (H), list the constructs used to traverse the list, such as while, for-index, for-elt, for-size, while-size, length, size. If multiple traversals, just create a semi-colon separated list of all constructs used.

The Testing columns (I-M) each take a number denoting how many tests were written for the corresponding kind of test. The specific columns/tests are as follows:

- TestEmptyAfterClean = Test empty
- TestSentinelFirstElt = Test if sentinel is first elt
- TestNoPosBeforeSentinel = Test no pos before sentinel, includes cases when no pos and no sentinel (all negative elts)
- TestDataPostSentinel = Test when there's data after sentinel
- TestNoSentinel = Test data has no sentinel

The "Handle/Consider Empty Valid Data" column (N) records how the solution handles the possibility that there are no positive numbers before the -999. Enter one of the following:

- Flag, if maintained a variable to record whether a pos number has been seen
- Checked, if checked count = 0 everywhere needed (before and after cleaning out negatives)

- Partial, if only catch some cases (shows us if they check the original list for being empty but not the list after clearing negatives)

- Comment, if they mention this in a comment but don't handle it in their code