CS 435 - Homework 8

Deadline

Due by 11:59 pm on **TUESDAY**, **May 6**, 2014 **<-- NOTE THE UNUSUAL DEADLINE!!**

How to submit

Submit your files for this homework using `~st10/435submit` on nrs-labs, with a homework number of `8`

Purpose

To read and reflect on an ACM Queue column regarding a 2004 survey of then-current software engineering practices, to consider the distinction between verification and validation, and to consider and try out examples of some of the metrics from Jalote Chapter 7.

Important notes

- Note that some of your submissions for this assignment may be posted to the course Moodle site.
- Create a file named `435hw8.txt` or `435hw8.pdf` (your choice) that starts with your name. Then give the problem number and your answer(s) for each of the following problems.

Problem 1

Go the HSU Library's link to the ACM Digital Library, and find the article:

LaPlante and Neill, "The Demise of the Waterfall Model is Imminent" and Other Urban Myths", ACM Queue, February 2004, pp. 10-15

(When I searched using the string "Demise of the Waterfall Model is Imminent", the desired article was the first result.)

- BEFORE you follow that link, take a screen shot of your ACM Digital Library window at this point.
  - (This can be a screen-shot of that window, or a cell-phone photo of that window, etc. Make sure it can be saved as a .jpg, .gif, or .png, whichever screen-shot means you use -- and name it `acm-demise-img` followed by the appropriate suffix.)
- Read this article, and answer the following questions in your file `435hw8.txt` or `435hw8.pdf`

1 part a

In the cited survey, what was the dominant software development process model the practitioners claimed to have used?

1 part b

Which myth that they discuss did you find either the most interesting or the most surprising (your choice)? Why did you find that myth to be most interesting/most surprising?

1 part c

Give at least one of their recommendations from this article, and also say why you happened to choose it.

Problem 2

Consider the following definitions from Wikipedia, citing IEEE-STD-610:

Verification - "The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

Validation - "The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements."

These are not quite the same thing, and I think it is important to understand the distinction -- one way I have heard this described is:

- verification ensures 'you built it right', it meets the specifications;
- validation ensures 'you built the right thing', it actually meets the user's needs, meets the requirements (and that the specifications were correct in the first place)

2 part a

Is unit testing of a module more accurately characterized as verification or validation?

2 part b

Is acceptance testing of a module more accurately characterized as verification or validation?

2 part c

Is a requirements review (as described in Jalote Chapter 3, p. 65) more accurately characterized as verification or validation?

2 part d

Is a code inspection (as describe in Jalote Chapter 7, section 7.5) more accurately characterized as verification or validation?

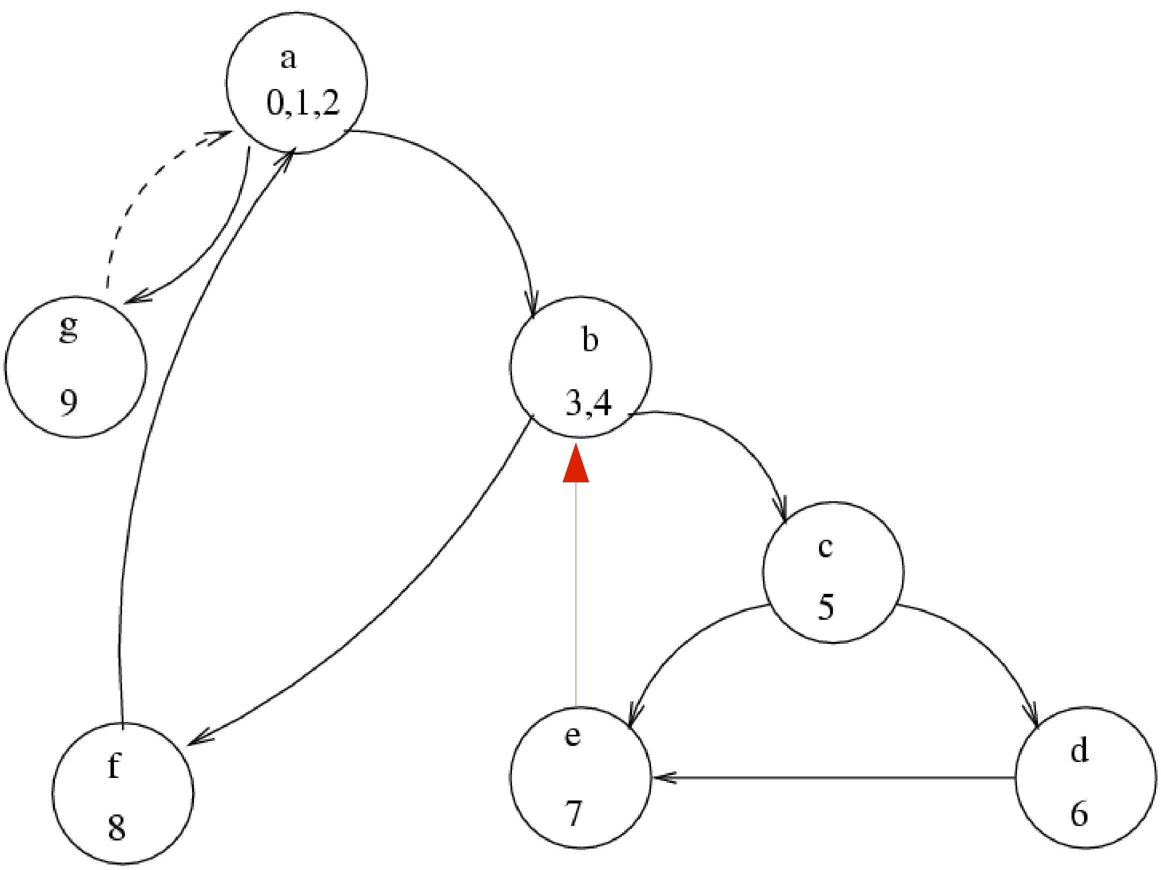
Clarification: cyclomatic complexity

First: consider the following example of cyclomatic complexity, from Jalote Ch. 7 -- I think there's a piece missing in the textbook example, which I believe is corrected here.

I'm keeping Jalote's example bubble-sort code fragment as given in the text, in spite of its icky formatting, since he uses line numbers based on that icky formatting:

```
0. {
1.      i=1;
2.      while (i<=n) {
3.         j=i;
4.         while(j <= i) {
5.             If (A[i]<A[j])
6.                 Swap(A[i], A[j]);
7.             j=j+1; }
8.      i = i+1; }
9. }
```

And, here is the control flow graph, with the missing edge shown in red:



Discussing/explaining this control flow graph a bit:

- the lines of code from the beginning of the fragment to the first decision, the outer-while loop condition, numbered 0, 1, and 2, correspond to the node marked with **a** and the line numbers 0, 1, 2
- the lines of code from the beginning of the body of the outer-while to the next decision, the inner-while loop condition, numbered 3 and 4, correspond to the node marked with **b** and the line numbers 3, 4
- the line of code with the next decision, the if-statement condition, numbered 5, corresponds to the node marked with **c** and the line number 5.
- the line of code with the if-statement action (and implied "end of if-statement body"), numbered 6, corresponds to the node marked with **d** and the line number 6.
- the line of code that increments j (and implied "end of inner-while-loop body"), numbered 7, corresponds to the node marked with **e** and the line number 7.
- the line of code that increments i (and implied "end of outer-while loop body"), numbered 8, corresponds to the node marked with **f** and the line number 8.
- the line of code that ends this fragment -- just the closing brace! -- numbered 9, corresponds to the node marked with **g** and the line number 9. This is where control is transferred when the outer while loop is completed.

Hopefully, that's so far, so good.

Now, consider the edges in this graph:

- at the first decision, the outer-while loop condition, control can pass to either the body of the outer while loop (if the condition is true) or after the loop (if the condition is false). So, there are two edges from node a/lines 0, 1, 2:
  - one from node a/0,1,2 to node b/3,4 (the beginning of the outer while-loop body)
  - and one from node a/0,1,2 to node g/9 (after the outer-while body)
- at the second decision, the inner-while loop condition, control can pass to either the body of the inner while loop (if the condition is true) or after the inner loop body (if the condition is false). So, there are two edges from node b/lines 3, 4:
  - one from node b/3,4 to node c/5 (the beginning of the inner while-loop body)
  - and one from node b/3,4 to node f/8 (after the inner-while body)
- at the third decision, the if-statement within the inner while loop, control can pass to either the if-action (if the condition is true) or after the if-action (if the condition is false) -- note that there is no else part. So, there are two edges from node c/line 5:
  - one from node c/5 to node d/6 (the if-action)
  - and one from node c/5 to node e/7 (the statement after the if-action)
- node d/6 only has one edge from it, because it is the end of the if-action, and control passes to node e/7 (the statement after the if-action)
- node e/7 has no edge leading out of it! That's the omission I was talking about -- here, there SHOULD be an edge from node e/7 to node b/3,4, since control passes from the end of the inner loop to the beginning of the inner loop. I've included this edge in red.

- node f/8 only has one edge from it, to node a/0,1,2, because it is the end of the outer loop, and control passes to the beginning of the outer loop.
- and, node g/8 has an implied/dotted edge back to node a/0,1,2, I'm guessing because once the fragment is done, you perhaps could start it over if you'd like...? p. 218 implies this is to result in a strongly-connected graph (adding an edge from the exit node to the entry node).

I hope the above makes it easier to understand the discussion on pp. 218-219 of Jalote. You should read over that again, and note:

- the cyclomatic complexity of a module is defined to be the cyclomatic number of such a control-flow graph.
- the cyclomatic number $V(G)$ of a control-flow graph G with n nodes, e edges, and p connected components is:

  $V(G) = e - n + p$

  - in the given graph, there are 10 edges, 7 nodes, and 1 connected component (ah -- because starting at a, you can reach every other node and come back to node a?) [notice that you only get 7 edges if you add the edge I've shown in red -- that's one reason I think this is an omission.]

    V(G) = 10 - 7 + 1 = 4
- It turns out that the cyclomatic complexity of a module (or cyclomatic number of its control flow graph) is equal to the maximum number of linearly-independent circuits in the graph, where a circuit is linearly independent if no circuit is totally contained in another circuit or is a combination of other circuits.
  - There are indeed 4 such circuits in this graph: (and again notice how several of these include that red edge):

    linearly-independent circuit 1: b c e b

    linearly-independent circuit 2: b c d e b

    linearly-independent circuit 3: a b f a

    linearly-independent circuit 4: a g a
- Here's the money-shot: it can also be shown that the cyclomatic complexity of a module is the number of decisions in the module plus one, where a decision is effectively any conditional statement in the module!
  - so -- after all that! -- you can ALSO compute the cyclomatic complexity simply by counting the number of decisions in the module and adding 1 --
  - here, we have 3 such decisions, for the outer while, inner while, and if, plus 1 again gives us 4!
- Can you see how this cyclomatic number can be one quantitative measure of a module's **complexity**? (Notice I said a quantitative measure of complexity, not of size!) Two modules might have the same number of lines, but quite different cyclomatic numbers, and it seems reasonable to argue that the module with more decisions is more complex than one that is simply a sequence of statements.
  - McCabe, who proposed this measure, also proposed that the cyclomatic complexity of modules should, in general, be kept below 10.

- Experiments indicate that the cyclomatic complexity is highly correlated to the size of the module in LOC; it has also been found to be correlated to the number of faults found in modules.

That said...

...NOW consider the following two algorithms (which look to me like they're written in a version of Algol), one for linear search and one for binary search, adapted from p. 223 of the course text. You should use **these** versions in your answers to the following problems.

(You should already be aware that binary search is generally more efficient in terms of execution time than linear search as the number of elements being searched increases.)

```
function lin_search(A, E): boolean
var
    i: integer;
    found: boolean;
begin
    found := false;
    i := 1;
    while (not found) and (i <= n) do begin
        if (A[i] = E) then
            found := true;
        i := i + 1;
    end;
    lin_search := found;
end;


function bin_search(A, E): boolean
var
    low, high, mid, i, j: integer;
    found: boolean;
begin
    low := 1;
    high := n;
    found := false;
    while (low <= high) and (not found) do begin
        mid := (low + high) / 2;
        if E < A[mid] then
```

```
            high := mid - 1
        else if E > A[mid] then
            low := mid + 1
        else
            found := true;
    end;
    bin_search := found;
end;
```

Problem 3

Determine and give the **cyclomatic complexity** for each of these two functions. (A control-flow graph is **NOT** required.)

Problem 4

See Jalote pp. 219-220 for a description of live variable complexity. Note in particular:

- A variable is considered to be **live** from its first to its last reference within a module, including all statements **between** the first and last statement where the variable is referenced.
  - note, then -- if a variable is first referenced in statement 5, and last referenced in statement 10, then it is included in the live variable count for EACH of the statements 5, 6, 7, 8, 9, and 10.
- Using this definition, the set of live variables for each statement can be computed by analysis of the module's code.
- For a statement, the number of live variables represents the degree of difficulty of the statement.
- This notion can be expanded to the entire module by defining the average number of live variables per executable statement --
  - that is, get the number of live variables per executable statement,
  - sum the number of live variables for all of the module's executable statements,
  - divide by the number of executable statements

  ...and the result is the live variable complexity for that module.

Determine and give the **live variable complexity** for each of these two functions.

Problem 5

Give the ratio of `lin_search`'s cyclomatic complexity over `bin_search`'s cyclomatic complexity, and give the ratio of `lin_search`'s live variable complexity over `bin_search`'s live variable complexity. Also answer this: are these two ratios similar?

Problem 6

Determine and give the number of non-comment, non-blank lines in each of these functions.

Problem 7

Consider Halstead's size metrics on Jalote pp. 215-216.

Determine and give **Halstead's length measure**:

$N = N_1 + N_2$

...where N_1 is the total number of occurrences of all of the operators in a module, and

...where N_2 is the total number of occurrences of all of the operands in a module,

for each of these functions.

Problem 8

Still considering Halstead's size metrics on Jalote pp. 215-216, determine and give **Halstead's volume measure**:

$V = N \log_2 (n)$

...where N is Halstead's length measure that you computed in Problem 7, and

$n = n_1 + n_2$

...where n_1 is the number of distinct/unique operators in the module, and

...where n_2 is the number of distinct/unique operands in the module,

for each of these functions.

Problem 9

Make and give a chart showing and comparing the size measured in LOC, measured as Halstead's length measure, and measured as Halstead's volume measure, for these two functions.

Which of these measures seems more pertinent to you? Why? (notice that you are being asked TWO questions here, in addition to giving the chart.)

Submit your resulting file `435hw8.txt` or `435hw8.pdf` along with Problem 1's `acm-demise-img.*` screenshot image file.