A Multi-National Study of Reading and Tracing Skills in Novice Programmers

Raymond Lister
Faculty of Information Technology
University of Technology, Sydney
Broadway, NSW 2007, Australia
+61 2 9514 1850
raymond@it.uts.edu.au

Elizabeth S. Adams
Department of Computer Science
James Madison University
Harrisonburg, VA 22807, USA
+1 (540) 568 1667
adamses@jmu.edu

Sue Fitzgerald
Information and Computer Sciences
Metropolitan State University
St. Paul, MN 55106 USA
+1 (651) 793-1473
sue.fitzgerald@metrostate.edu

William Fone
Staffordshire University
Stafford, ST18 0DG
United Kingdom
+44-1785-353304
W.Fone@Staffs.ac.uk

John Hamer
Department of Computer Science
University of Auckland
Auckland, New Zealand
+64 9 3737 599
j.hamer@cs.auckland.ac.nz

Morten Lindholm
Department of Computer Science
Aarhus University
Aarhus, DK-8200, Denmark
+45 8942 5621
lindholm@daimi.au.dk

Robert McCartney
Dept Comp. Sci. and Engineering
University of Connecticut
Storrs, CT 06268 USA
+1 (860) 486-5232
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
905 86 Umeå, Sweden
+46 90 147 289
jem@cs.umu.se

Kate Sanders
Math/CS Department
Rhode Island College
Providence, RI 02908 USA
+1 (401) 456-9634
ksanders@ric.edu

Otto Seppälä
Laboratory of Info. Processing Science
Helsinki University of Technology
PL 5400, 02015 TKK, Finland
+358 (9)451 5193
oseppala@cs.hut.fi

Beth Simon
Dept of Maths and Computer Science
University of San Diego
San Diego, CA 92110
+1 (619) 260-7929
bsimon@sandiego.edu

Lynda Thomas
Department of Computer Science
University of Wales
Aberystwyth
+44 (1970) 622452
ltt@aber.ac.uk

ABSTRACT

A study by a ITiCSE 2001 working group ("the McCracken Group") established that many students do not know how to program at the conclusion of their introductory courses. A popular explanation for this incapacity is that the students lack the ability to problem-solve. That is, they lack the ability to take a problem description, decompose it into sub-problems and implement them, then assemble the pieces into a complete solution. An alternative explanation is that many students have a fragile grasp of both basic programming principles and the ability to systematically carry out routine programming tasks, such as tracing (or "desk checking") through code. This ITiCSE 2004 working group studied the alternative explanation, by testing students from seven countries, in two ways. First, students were tested on their ability to predict the outcome of executing a short piece of code. Second, students were tested on their ability, when given the desired function of short piece of near-complete code, to select the correct completion of the code from a small set of possibilities. Many students were weak at these tasks, especially the latter task, suggesting that such students have a fragile grasp of skills that are a pre-requisite for problem-solving.

1. INTRODUCTION

Despite the best efforts of teachers in our discipline, many students are still challenged by programming. A 2001 ITiCSE working group (the "McCracken group") assessed the programming ability of a large population of students from several universities, in the United States and other countries [McCracken, 2001]. The authors tested students on a common set of programming problems. The majority of students performed much more poorly than expected. In fact, most students did not even get close to finishing the set task. The results are compelling, given the multi-national nature of the collaboration. Whereas a similar report from an author at a single institution might be dismissed as a consequence of poor teaching at that institution, it is difficult to dismiss a multinational study.

While the work of the McCracken group has highlighted the extent of the problem, the nature of the McCracken study does not isolate the causes of the problem. A popular explanation for the poor performance of students is that they lack the ability to problem-solve. The McCracken group defined problem-solving as a five step process: (1) Abstract the problem from its description, (2) Generate sub-problems, (3) Transform sub-problems into sub-solutions, (4) Re-compose, and (5) Evaluate and iterate. However, there are other potential explanations for why students struggle to program. For example, students may simply not understand the programming constructs they need to produce a program (e.g. arrays or recursion). Another and more subtle explanation is that the student's knowledge is "fragile". That is, while a student may be able to articulate particular items of knowledge when explicitly prompted for any of them, when that student is asked to apply that knowledge in a program writing context, the student "sort of knows, has some fragments, can make some moves, has a notion, without being able to marshal enough knowledge with sufficient precision to carry a problem through to a clean solution" [Perkins and Martin, 1986, p. 214]. In this paper, we use the term "fragile knowledge" to also include basic skills, such as the ability to systematically, manually execute ("trace") a piece of code.

As Perkins and Martin point out, general problem solving and knowledge are not "two independent dimensions of programming" [p. 226]. Nor is the comprehensive acquisition of programming knowledge and skills an absolute precursor to manifesting the ability to problem-solve. However, some minimal grasp of programming concepts and associated skills is required before a student can manifest problem-solving skills in the strong five-step sense as defined by the McCracken group. These considerations led us to ask the following question: to what degree did students perform poorly in the McCracken study because of poor problem solving skills, or because of fragile knowledge and skills that are a precursor to problem-solving?

This paper is the report of an ITiCSE 2004 working group, which explored the above question by asking students to demonstrate their comprehension of existing code. If a student can consistently demonstrate an understanding of existing code, but struggles to write similar programs, then it may be reasonable to conclude that the student lacks the skills for problem-solving. However, if a student cannot consistently demonstrate understanding of existing code, then such a student's difficulty is a lack of knowledge and skills that are a prerequisite for non-trivial, five-step, problem-solving. The data for this working group was collected by asking students to answer twelve Multiple Choice Questions (MCQs). The complete set of MCQs is given in appendix A. In that appendix, the programming code is given in Java. However, the questions are intended to test student knowledge and skills for generic, iterative processes on arrays, and therefore can be easily translated into many programming languages. Within the working group, one participating institution translated the MCQs into another programming language, which was C++.

In this study, we used MCQs as the vehicle for studying the students for two reasons. First, since this is a multi-institutional study, we wanted a way of scoring student performance on the MCQs that did not require subjective judgment on the part of each working group member. Second, we were concerned that if students were instead required to explain the function of a piece of code, poor performance might be due to a lack of eloquence, not a lack of understanding of the code (especially where the student was being interviewed in a language other than their first language).

2. DATA COLLECTION

While answering the 12 MCQs, students provided three types of data, described in the next three subsections.

2.1 Performance Data

The purpose of collecting this data was to gauge the difficulty of the MCQs across all participating institutions.

Each working group member tested students on the twelve MCQs, under exam conditions. The primary data collected was the students' answer for each MCQ, from which a score out of 12 could be calculated. A total of 941 students contributed data to this part of the study, but of those students only 556 students were given all twelve questions.

Most students who undertook this performance test had either recently completed, or had nearly completed, their first semester of studying programming. However, at several institutions, many students were not in the first semester of their studies, or even their first year.

In some institutions, the MCQs were used as part of the procedure for assigning a final grade to the students. In other institutions, students volunteered to be part of the study.

2.2 Interview Transcripts

The purpose of collecting this data was to investigate how students went about answering the MCQs.

Each of the twelve working group members interviewed at least three students. A total of 37 students were interviewed. In the interview, students were asked to "think out loud" as they answered the core set of MCQs. The interviews were recorded and then transcribed. Some students were interviewed in a language other than English. In such cases, the final version of the transcription was a translation of the interview into English.

In institutions where the twelve MCQs were used as part of the grading process, the interviews were conducted after the students sat for the test. In such cases, the interview was a debrief, as students recollected how they answered each question. In other interviews, where the MCQs were not part of the grading process, students saw the questions for the first time at the interview. Their choice of answer for each MCQ was also included in the performance data discussed in section 2.1.

Of the interviewees, the median age was 20 years, ranging from 18 to 54 years. Ten percent were female. The average number of years at their respective institutions was 1.4 with a standard deviation of 0.9. Most students were in their first programming course. The Java students had been studying programming for an average of 0.45 years, C++ students an average of 0.3 years.

2.3 Doodle Data

The purpose of collecting this data was to investigate how students went about answering the MCQs.

Students were given "scratch" paper upon which they were allowed to draw pictures or perform calculations as part of answering the MCQs. Within the working group, such drawings and calculations are referred to as "doodles". Usually, a student was allowed to doodle on the same page upon which the MCQ was given to the student.

Doodles were collected from all 37 students who were interviewed. At some institutions, doodles were also collected from students who supplied performance data.

3. PERFORMANCE DATA ANALYSIS

This section contains a statistical analysis of the performance data, without any detailed reference to the transcript or doodle data.

Figure 3.1 shows a histogram of scores (out of 12) of the 556 students who were given all twelve questions. The scores cover the complete range, from 0 to 12, with a mode of 8. Table 3.1 shows the quartile boundaries for these students. These quartile boundaries are used extensively in this section's analysis of the performance data.

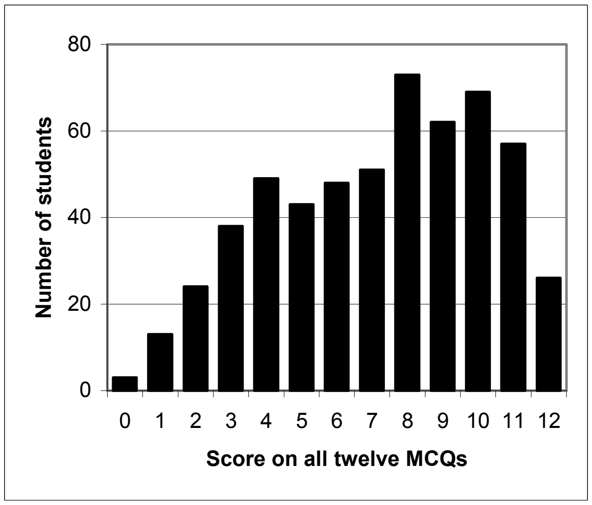


Figure 3.1. Distribution of scores for students who were given all 12 MCQs (N=556).

Quartile	Score Range	No. of Students	Percent of Students
1st "top"	10 - 12	152	27%
2nd	8 - 9	135	24%
3rd	5 - 7	142	25%
4th "bottom"	0 - 4	127	23%

Table 3.1. The quartile boundaries for students who were given all 12 MCQs (N=556).

Quartile	Score Range	No. of Students	Percent of Students
1st "top"	10 - 12	46	20%
2nd	8 - 9	56	25%
3rd	5 - 7	60	26%
4th "bottom"	0 - 4	66	29%

Table 3.2. The quartile boundaries for students who were given all 12 MCQs, from all but one institution (N=228).

A single institution contributed over half the performance data summarized in Table 3.1. (The next largest contribution by a single institution was 11%. Six institutions contributed data for all 12 MCQs from at least 20 students.) There is therefore the possibility that the quartile boundaries in Table 3.1 are influenced unduly by the students of the institution that contributed over half the performance data. Table 3.2 shows a breakdown of student numbers, by the quartile boundaries established in Table 3.1, but excluding students from the institution that contributed the majority of the performance data. Table 3.2 shows that the percentage of students from all other institutions in the top quartile is lower (20% compared with 27% in Table 3.1) and the percentage of students in the bottom quartile is higher (29% compared with 23% in Table 3.1). The percentage of students in the middle two quartiles is nearly the same. On the basis of this comparison, and for the purposes of this study, we conclude that the student performance statistics are influenced but not dominated by that single institution. To the extent that the performance statistics are influenced by that single institution, the effect is to increase the scores of students in the total population.

The above performance statistics are not as strikingly bad as the performance figures in the McCracken study, where over half the students did very poorly. We expected the distribution of performance in our study to be better than the distribution in the McCracken study, given our assumption that the ability to read and understand small pieces of code is a prerequisite skill for five-step problem solving. However, it is difficult to ascertain what is a "good" score on the 12 MCQs without first studying those MCQs. For that reason, the next two subsections each examine a specific MCQ. A similar analysis of the remaining ten MCQs is given in appendix B. Subsequent subsections examine statistical data across all twelve MCQs.

3.1 Question 2: A Mid-Range MCQ

Figure 3.2 contains an MCQ, Question 2 from the complete set of 12 MCQs. It was answered correctly by 65% of all students who attempted it, and is a question of mid-range difficulty in the set of 12 MCQs.

At first glance, Question 2 might appear to count the number of common elements in both arrays, which is 3 (choice A). However, on closer inspection of the code, it can be seen that the elements at position 0 in the arrays are not counted, so the correct answer is 2 (choice B). Ignoring the first element of an array is not idiomatic – it's not what many programmers might expect – but it may be correct. For example, the first element of an array in C or C++ is sometimes used to store the length of the array, rather than an element of the array. Also, many bugs are caused by the fact that the code we write does not always do what we intended it to do, so the ability to read what the code actually does, rather than what we think it should do, is an important programming skill. Irrespective of whether Question 2 is a piece of code that a teacher should show their class, it is certainly a piece of code that a student might write and need to debug.

MCQs are a common way of testing students in many disciplines, and there is considerable body of literature devoted to the construction and analysis of such tests [Ebel & Frisbie, 1986; Linn & Gronlund, 1995; Haladyna, 1999]. A common way of analyzing the effectiveness of a MCQ is based upon the notion that MCQs should be answered correctly by most strong students, and incorrectly by most weak students. For Question 2, approximately 90% of students in the first quartile (i.e. students who scored 10-12 on all 12 MCQs) answered this question correctly, whereas approximately 30% of students in the bottom quartile (i.e. scored 0-4 on all 12 MCQs) answered this question correctly. (Note that a student who understands nothing about the question, and simply guesses, stands a 25% chance of guessing correctly, while a student who can eliminate one option stands a 33% chance of guessing correctly.) On the basis of these two percentages for the top and bottom quartiles, this MCQ does distinguish between stronger and weaker students.

A similar but more comprehensive quartile analysis of Question 2 is given in Figure 3.3. This type of figure is an established way of analyzing MCQs [Haladyna, 1999]. It shows the performance of all four student quartiles, and also summarizes the actual choices made by students in each quartile. The horizontal axis represents the four student quartiles. The uppermost trend line in that figure represents choice B, the correct choice for Question 2. As stated earlier, approximately 90% of students in the first quartile chose option B. The percentage of students who chose option B drops in the second quartile, but it remains by far the most popular choice among second quartile students. For third quartile students, just over half chose option B, but approximately 30% of third quartile students chose option A. On those figures, it would appear that most third quartile students grasped the basic function of the code in Question 2, although 30% missed the non-idiomatic detail that the elements in position zero of the arrays were not counted. Among fourth quartile students, the correct choice was less popular than option A, and many students chose options C or D.

Question 2 is not especially difficult. The first quartile students did very well on this question and even the second and third quartiles had a strong preference for the correct answer to the question. However it is an effective question for distinguishing between most students and the particularly weak students (i.e. fourth quartile). On the basis of this question, it would seem that most students in the top three quartiles have a reasonable grasp of the concepts tested by this question (primarily arrays and iteration).

One of the most challenging aspects of writing a multiple-choice test is writing the distracters (i.e. the incorrect options). A good distracter should be plausible enough to appeal to some students. It should be clearly incorrect, however, so that it does not mislead those students who really know the material. In practice, it is difficult to write three good distracters. Figure 3.3 shows that distracter A was the most effective, whereas distracter D was not effective for the top three quartiles.

To summarize the above discussion, Figure 3.3 illustrates two visual properties that are usually regarded as desirable in most such graphs of an MCQ [Haladyna, 1999], particularly where students are being norm-referenced (i.e. graded according to a desired distribution of grades). Those two properties are:

1) The trend line for the correct answer is monotonically decreasing from left to right (i.e. from the strongest to weakest students).
2) The trend line for each distracter is monotonically increasing from left to right.

It is less clear, however, that MCQs should consistently exhibit these properties when students are being criterion-referenced (i.e. tested for their mastery of certain knowledge and skills), which is the object of this research project. The next subsection discusses a harder MCQ from the set of 12, which does not exhibit all the above "desirable" norm-referencing properties.

<u>Question 2.</u>
Consider the following code fragment.

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;

while ((i1 > 0 ) && (i2 > 0 ))
{
     if ( x1[i1] == x2[i2] )
     {
          ++count;
          --i1;
          --i2;
     }
     else if (x1[i1] < x2[i2])
     {
          --i2;
     }
     else
     {    // x1[i1] > x2[i2]
          --i1;
     }
}
```

After the above while loop finishes, "count" contains what value?

a) 3
b) 2
c) 1
d) 0

Figure 3.2. Question 2 from the set of 12 MCQs

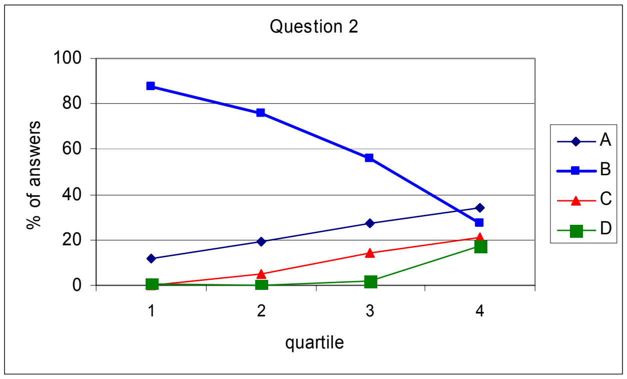


Figure 3.3. Student responses to Question 2, by quartiles.

Question 8.

If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array "x" that contains the following six numbers:

```
4  5  6  2  1  3
```

There are 10 inversions in that array, as:

```
x[0]=4   >   x[3]=2
x[0]=4   >   x[4]=1
x[0]=4   >   x[5]=3
x[1]=5   >   x[3]=2
x[1]=5   >   x[4]=1
x[1]=5   >   x[5]=3
x[2]=6   >   x[3]=2
x[2]=6   >   x[4]=1
x[2]=6   >   x[5]=3
x[3]=2   >   x[4]=1
```

The skeleton code below is intended to count the number of inversions in an array "x":

```
int inversionCount = 0;

for ( int i=0 ; i<x.length-1 ; i++ )
{
      for  xxxxxx
      {
            if (  x[i] > x[j] )
                  ++inversionCount;
      }
}
```

When the above code finishes, the variable "inversionCount" is intended to contain the number of inversions in array "x". Therefore, the "xxxxxx" in the above code should be replaced by:

```
a) ( int j=0  ; j<x.length  ; j++ )
b) ( int j=0  ; j<x.length-1; j++ )
c) ( int j=i+1; j<x.length  ; j++ )
d) ( int j=i+1; j<x.length-1; j++ )
```

Figure 3.4. Question 8 from the set of 12 MCQs

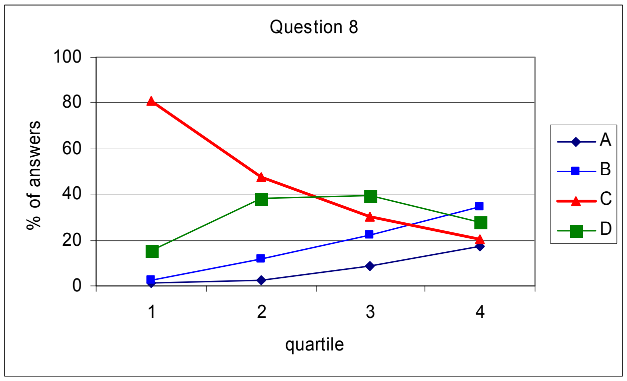


Figure 3.5. Student responses to Question 8, by quartiles.

3.2 Question 8: A Harder MCQ

Figure 3.4 contains Question 8 from the complete set of 12 MCQs. It was answered correctly by 51% of all students who attempted it, the third lowest percentage for all 12 MCQs. The four choices given to the students focus on two points of the inner loop:

1) Should the inner loop start processing at position 0 or at position i+1?

2) Should the inner loop continue processing while j<x.length or while j<x.length-1?

All four possible combinations of the above two options are given as choices, respectively:

a) This option is wrong on the first of the above two points, but it is the most idiomatic of the four choices. That is, it is a loop where the control variable "j" would sweep across the entire array, which is likely to be the first and most common iterative process on an array seen by these novice programmers.

b) This option is wrong on both of the above two points, but it is the option most similar to the outer loop.

c) The correct answer.

d) This option is wrong on the second of the above two points, but it does have a test condition similar to the test condition of the outer loop.

A quartile analysis of Question 8 is given in Figure 3.5. Approximately 80% of students in the first quartile chose the correct answer, option C. That option was the most popular with second quartile students, but distracter D was almost as popular, with the incorrect loop termination condition `j<x.length-1`. Distracter D was in fact the most popular choice for third quartile students. Distracter D was the second most popular option with fourth quartile students; both it and the more popular distracter contain the incorrect loop termination condition `j<x.length-1`. In summary, the most popular

distracter choice across all quartiles contained the incorrect loop termination condition.

An examination of transcripts indicates that students who thought that `j<x.length-1` was correct understood that the "j" subscript needed to go all the way to the end of the array, and they recognized that the last position of the array was `x.length-1`, but they neglected the effect of the "<" symbol, effectively selecting their answer as if that symbol was "<=" instead. A large number of students made this mistake, which is surprising since the very next question in the set of 12 MCQs requires students to select another loop test condition, and students performed much better on that question, with 73% of students answering it correctly. It may be that the students were primed to select the incorrect test condition in Question 8 because of the presence, in this same question, of the test condition "`i<x.length-1`" for the outer loop. In one interview transcript, a second quartile student explicitly acknowledged that as the reason for the choice: "Yeah, I chose this one because it would be the same ….". If this conjecture is correct – that students were distracted by the presence on the same page of another loop condition - then it highlights the fragility of the knowledge of many novice programmers.

3.3 Aggregate Performance on MCQs

Table 3.3 shows the percentage of students who answered each of the 12 MCQs correctly. The "Rank" column shows relative difficulty of each question, as defined by the percentage of students who answered the question correctly. Questions 2 and 8, which were studied in detail above, are ranked 6th most difficult and 3rd most difficult respectively.

(At some institutions, some students were not given all 12 MCQs. In general, those students cannot be included the data analysis of this paper, as much of the data analysis depends upon classifying each student into a quartile, based upon their performance on all 12 MCQs. However, in Table 3.3, such students were included.)

Question 6 was ranked as the second most difficult question. A detailed analysis of that question is in Appendix B. This question involves the use of a "return" from within a "for" loop. From the transcripts, it is apparent that many students do not understand that the "return" will terminate the loop immediately. This misconception is consistent across institutions and countries.

Ignoring question 6, because it involves a conceptual misunderstanding by students, the questions ranked hardest in Table 3.3 are Questions 8, 11, and 12. A common characteristic to these three questions is that they are all "skeleton-code questions." That is, these questions require the student to select the correct code to complete given "skeleton" code. As a general rule, the easier MCQs are "fixed-code questions". That is, the easier questions require the student to hand execute some code and select the outcome. Question 2 is such a fixed-code question. The exception to this general rule is Question 9, which is a skeleton code question, but which is ranked second easiest in Table 3.3.

MCQ	%correc t	Rank	No. Students
1	68	8/9	644
2	65	6	644
3	67	7	611
4	62	5	611
5	74	(easiest) 12	611
6	42	2	611
7	72	10	798
8	51	3	798
9	73	11	798
10	68	8/9	644
11	59	4	611
12	38	(hardest) 1	611

Table 3.3. Percentage of students who answered each of the 12 MCQs correctly.

Finally, we note that it is possible that Question 12 proved difficult simply because it was the final question. Students may have been tiring by that stage. Students for whom this test was a significant determinant of their final grade may have had the motivation to overcome such tiredness. However, students for whom the test counted for little or nothing would have had low motivation to overcome tiredness.

3.4 Relative Performance Across Institutions

At the heart of any multi-institutional study is the assumption that data collected at different institutions can be compared in some meaningful way. Inevitably, however, there will be differences between the institutions, both in the nature of the institutions and details of how the data is collected. In this study, differences included:

1) Student ability. Clearly, some institutions attract students with a greater innate talent for programming.
2) Student experience. Most students who provided performance data were at or near the end of their first semester of programming, but the total population of

students ranged from the middle of the first semester to the end of the third semester.

3) Student motivation: At some institutions this test had an impact on course grades, to varying degrees; at others, participating students were volunteers.

4) Programming language: In one institution, the Java code in the MCQs was translated into C++.

5) Modality of test: At one site the test was taken on a computer, at the others it was taken on paper.

6) Formatting of exam: Some researchers reformatted the questions to match the indenting style used in their classes. Also, one researcher prepared multiple versions of the questions, with the options of each MCQ reordered, to deter copying.

Figure 3.6 displays the variation in student performance across institutions, but also shows some general trends common to institutions. This figure effectively breaks down the information supplied in Table 3.3 according to institution, showing the percentage of students who answered each question correctly. However, Figure 3.6 only shows a subset of the data from Table 3.3, the subset comprising the 6 institutions that provided performance data for at least 20 students, where all those students were given the full 12 MCQs. Figure 3.6 highlights that, on any given question, the percentage performance varied considerably between institutions. However, some clear trends across questions are common to most of the institutions. For example, questions 6, 8, and 12 stand out as being difficult questions at most of the institutions.

One of the institutions in Figure 3.6 runs contrary to the general trend of most other institutions shown on that figure. This is the institution with a particularly low percentage on question 4. Students from this same institution performed relatively well on harder questions, 8, 11 and 12. There are three possible contributing factors to the unusual performance of this institution. First, this institution contributed data from 20 students, the minimum for inclusion in that figure. Therefore data from that institution is more prone to variation due to small sample size. Second, those students were in their first semester of Java programming, and were taught Java "objects early", which may have not prepared them well for questions about iterative processes on arrays. Third, inspection of transcripts for this institution reveals that these students had studied, or were studying in parallel, algorithms in a language independent context, and they recognized some of the latter MCQs as being very similar to algorithms they had studied. If we ignore that institution in Figure 3.6, trends are even more apparent across institutions.

As explained in an earlier section, one institution contributed more than half of the performance data for students given all 12 MCQs. In Figure 3.6, that institution is represented by diamonds, and follows the same trends as most of the other institutions.

On the basis of the analysis in this sub-section, as illustrated in Figure 3.6, we conclude that there are trends in the data across institutions. In general, a question that is substantially more difficult for students at one institution is also more difficult for students at other institutions.

3.5 Performance Across Quartiles

Table 3.4 shows the percentage of students with the correct answer for each question, broken down by performance quartile. The rightmost column of that table shows the average percentage performance across all questions for each quartile. In other columns, those numbers in bold and underlined are the percentages below the average for that quartile. As with earlier data analysis, questions 6, 8, and 12 stand out as being particularly difficult questions; here we see that they are relatively difficult for students in all performance quartiles.

Question 11 is interesting in that the middle two quartiles performed below their respective averages on this question. However, the difference in performance on this specific question and the overall average performance is no more than 4% for all four quartiles, so the relatively poor performance of the middle two quartiles needs to be treated with caution.

3.6 Familiarity with MCQ Exams

Most institutions do not grade programming students by MCQs. The most common grading practices are to ask students to write code, or explain in words what a piece of code does. The question therefore arises as to whether the students who participated in this study were not well prepared for the task required of them.

The MCQs used in this study were provided by one participating institution, where students at the end of their first semester of programming pass or fail that programming course on the basis of a MCQ exam. In the remainder of this subsection, we shall refer to this institution as the "base" institution, and all other institutions as the "satellite" institutions. Students at the base institution are routinely given a pool of practice MCQs several weeks prior to the grading exam. These students are therefore well prepared (and highly motivated) to do well on the type of MCQs used in this study. The performance of students on the 12 MCQs at the base institution are in Table 3.5. The remainder of this section compares the data in that table with the data from satellite institutions.

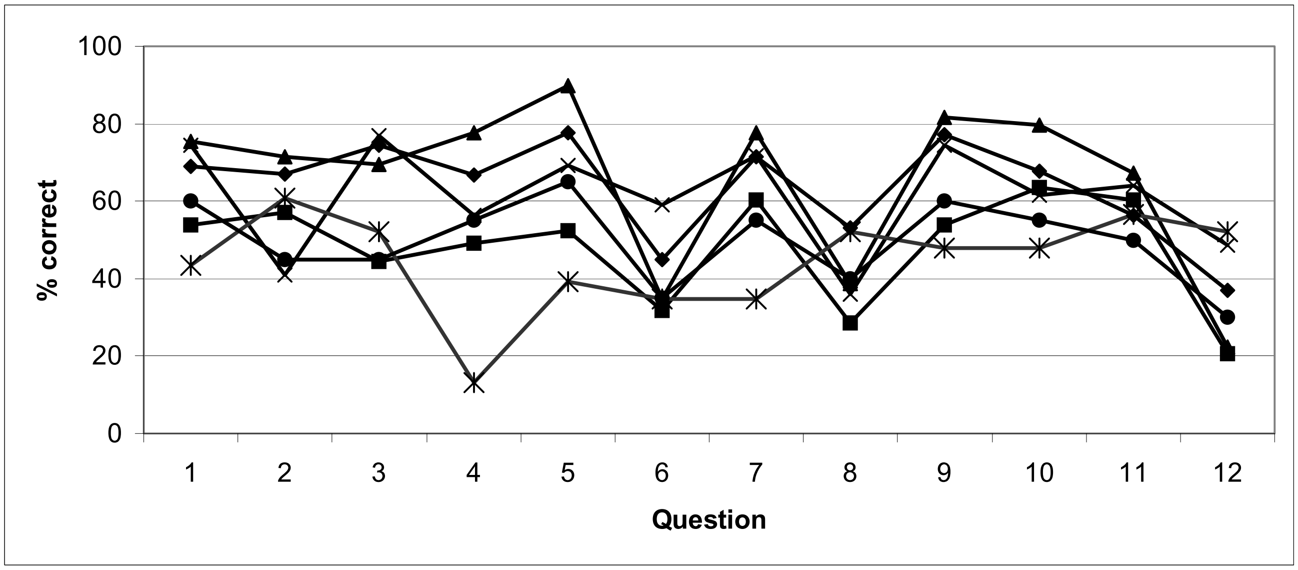


Figure 3.6. Percentage of students with the correct answer for each question, for the 6 institutions that provided performance data for at least 20 students, where those students were given all 12 MCQs. Each trend line corresponds to one institution.

question / quartile	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Average, all questions
Top (10-12)	93	**<u>87</u>**	99	90	99	**<u>74</u>**	95	**<u>81</u>**	98	93	89	**<u>73</u>**	89
Second (8-9)	79	76	81	73	93	**<u>50</u>**	82	**<u>47</u>**	86	81	**<u>67</u>**	**<u>31</u>**	70
Third (5-7)	50	56	62	52	72	**<u>28</u>**	61	**<u>30</u>**	67	65	**<u>46</u>**	**<u>18</u>**	50
Bottom (0-4)	35	28	25	25	**<u>21</u>**	**<u>14</u>**	31	**<u>20</u>**	32	**<u>21</u>**	27	**<u>13</u>**	24
All quartiles	65	63	68	61	73	**<u>43</u>**	69	**<u>46</u>**	72	67	**<u>58</u>**	**<u>35</u>**	60

Table 3.4. Percentage of students with correct answer for each question, by performance quartile**.**

Question No.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Percentage	**<u>84</u>**	**<u>79</u>**	55	69	80	42	78	**<u>63</u>**	76	80	69	**<u>67</u>**

Table 3.5. Percentage of students with correct answer for each question, at the "base" institution where the MCQs were written. Percentages in bold and underlined are outside the range of percentages across the other institutions in Figure 3.6.

In Table 3.5, the four percentages in bold and underlined are outside the range of percentage figures across the satellite institutions in Figure 3.6. In all four cases, the percentage for base students in the table is higher than the percentages for satellite students in the figure.

Questions 1 and 2 were answered more poorly by satellite students than by the base students, which may indicate that satellite students took a couple of questions to become familiar with what was required of them in this test.

A striking difference between Tables 3.3 and 3.5 is the different percentage of students who correctly answered Question 12. At the base institution, 67% of students answered that question correctly, compared with only 38% at the satellite institutions. At the base institution, a minor variation on Questions 12 was part of a pool of practice questions given to the students. Therefore, students at the base institution could be expected to do better on this question.

We conclude that unfamiliarity with MCQ exams was not a major impediment for students at satellite institutions. We suspect that, had the students at satellite institutions also had access to an extensive pool of practice questions prior to attempting these 12 MCQs, then their performance may have been a little higher, but not substantially higher.

(We end this subsection with the following passing note of clarification. The base institution provided data for all 12 MCQs from only three students. With the exception of Table 3.3, data analysis in this paper concerns students who completed all 12 MCQs. Therefore, the data from the base institution cannot inject a large bias into the data analysis.)

3.7 Reliability

There are established methods in the educational literature for evaluating the "reliability" of a test. Reliability has a fairly narrow meaning here: a test is reliable if it would discriminate consistently between students, based on partitioning the questions, grading the partitions, and seeing if the results agree. Therefore reliability tests are based upon the assumption that all questions in a MCQ exam test the same knowledge and skills. We computed Cronbach's coefficient alpha (Ebel and Frisbie, 1986), which provides the average correlation between scores of all possible two-group partitions of the questions. Using the data for students who were given all 12 MCQs (N=556), we obtained a value of 0.75. By convention, a value of 0.80 or higher is considered "reliable". In the set of 12 MCQs used in this study, Questions 6, 8, and 12 (one quarter of the complete set of questions) stand out as being relatively difficult questions. Therefore it is not surprising that the reliability of the complete set of 12 MCQs falls a little below the conventional threshold of 0.80.

3.8 Time Taken To Do The Twelve MCQs

It is possible that some students performed poorly on MCQs late in the complete set of 12 because they were running out of time. The time to be given to students to complete the exam was not specified as part of the experiment design, beyond the advice to participating institutions that students should be given at least an hour to do all 12 MCQs.

Time-duration data was collected for 339 of the students who were tested on all 12 MCQs. Most of this time data came from the institution that contributed the majority of performance data, where no upper time limit was given to students. Half the students took between 30 and 60 minutes to answer all 12 questions. That is, 75% of all students completed the 12 MCQs in under an hour. The longest time taken by a student in the first performance quartile (i.e. score of 10-12) was 105 minutes. The longest time by any student was 115 minutes, by a student in the third performance quartile (i.e. score 5-7). The distribution of times to complete the 12 MCQs was similar across the top three performance quartiles. Students in the bottom quartile tended to be quicker. Half of the bottom quartile students took between 20 and 50 minutes to answer all 12 questions, with the longest taking 65 minutes.

3.9 Performance Data Discussion

Students who fall within the bottom performance quartile, scoring 0-4 out of 12, probably suffer from problems more fundamental than an inability to problem-solve. Students who fall within the top quartile probably have a strong grasp of the basics of programming. If top quartile students manifest a subsequent weakness in writing code, then they probably do suffer from a weakness in problem solving.

Having categorized the top and bottom quartiles, there remains the harder task of making observations about the middle 50% of students, who scored between 5 and 9. It is sobering to consider the following hypothetical argument. Suppose the students who participated in this study were all studying their first semester of programming at a single institution. Suppose further they were given these 12 MCQs as their exam, and the institution regarded a 25% failure rate as the upper limit of what was acceptable. Then students who scored 5 out of 12 on these MCQs would be progressing to the second semester programming course. In the view of the authors, students scoring 5 out of 12 on these MCQ's probably do not have the level of skill and knowledge needed for a follow on course. The authors leave the readers to decide their own acceptable passing score on these 12 MCQs, and to calculate the resultant failure rate for this hypothetical class.

One observation we make about the middle 50% of students is that, by virtue of the fact that they answered some questions correctly, they have demonstrated a conceptual grasp of loops and arrays. Therefore, the weakness of these students is not that they do not understand the language constructs. Their weakness is the inability to reliably work their way through the long chain of reasoning required to hand execute code, and/or an inability to reason reliably at a more abstract level to select the missing line of code. Based on the performance data, it is not possible to draw any firm conclusions as to the exact cause of this weakness. To explore that issue further, the remainder of the paper examines the other data gathered in this project, the doodle data and the interview transcripts.

4. DOODLES

When faced with a piece of code to read and understand, experienced programmers frequently "doodle". That is, they draw diagrams and make other annotations as part of determining the function of the code.

Figure 4.1 shows the actual annotations made by a student while correctly choosing option "c" (as indicated by the circle around that option) for Question 1. (Recall that, as shown in Table 3.3, this question was answered correctly by 68% of all students, thus ranking it as one of the easier MCQs used in this study.) Above the first line of Java code in Figure 4.1, where array "x" is initialized, the student has annotated the various positions of the array. In this study, we refer to that as a "position" doodle (or just "P"). Above the test condition of the "while" loop, the student has written the values of the variables. In that test condition, as the variables "sum" and "i" changed, the updated values were written alongside the old value. We refer to this as a "number" doodle ("N"). Above and to the right of the test condition, the student has written the changing values in several Boolean expressions, such as "0 < 3" and "2 < 3". This is an example of a "trace" doodle ("T'). Under the statement "sum += x[i];", the student has written a "computation" doodle ("C"), "sum = 1 + 4 = 5". Near the four options, the student has written "sum = 0" and "i = 0" to record the value in these two variables; those are further examples of "N" doodles. At the lower right is a table, with a header row containing "sum", "lim", "i", and "len". As the student hand executed the code, and the variables listed in the header row of that table changed value, the new values were entered into the table. This is an example of the most elaborate type of doodle identified in this study, the "synchronized trace" ("S").

4.1 Categorizing Doodles

In this study, we analyzed the doodles of 56 students, consisting of all 37 students who were interviewed, plus others chosen at random from all participating institutions. Two authors of this study jointly went through the pages collected from these students, and developed a draft set of doodle categories.

After the two authors had devised the draft categories, three other authors then separately went through a subset of the same pages examined by the first two authors, and counted the frequency of occurrence of each type of doodle, as defined in the draft categorization. The frequencies of each of these three authors were compared to the frequencies obtained by the first two authors. In all three cases, the discrepancy was approximately 10%. Some of the discrepancies were caused by an author simply missing a doodle, while others occurred because the explanations of the categories were not clear enough. The explanations were re-written to clarify the categories, and produce the final draft of doodle categories.

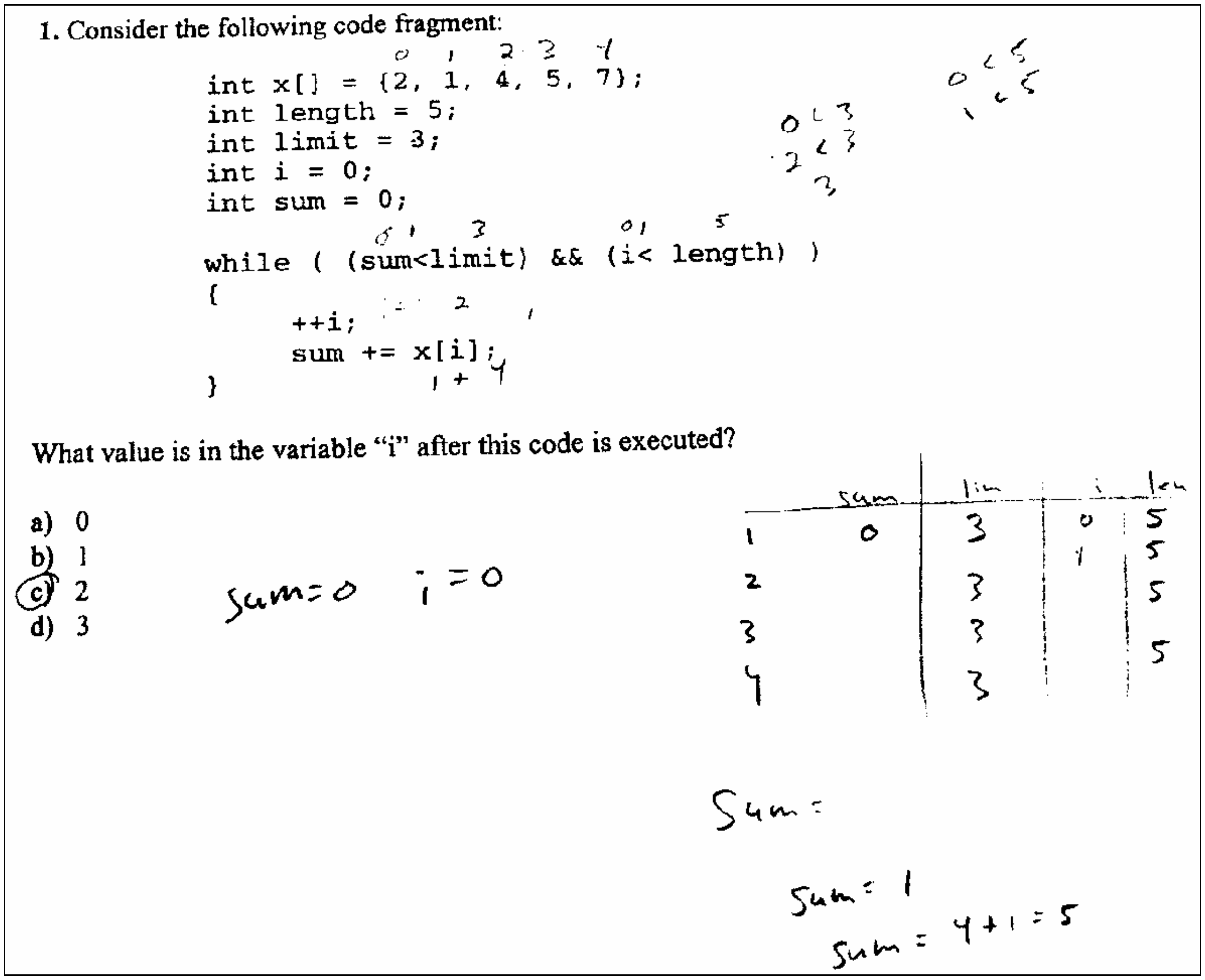
Table 4.1 contains the final draft of doodle categories, giving all types of doodles identified in this study. Note that any mark on a question paper is considered a doodle, other than the indication of the multiple choice option chosen. Also, note that a single MCQ answered by a student may have several doodle categories assigned to it, as would be the case for the question shown in Figure 4.1.

4.2 Doodle Category and Answer Accuracy

An obvious issue to consider is the effectiveness of each these doodle categories for answering the MCQs. To investigate that question, we analyzed the doodles of the 56 students on all 12 MCQs, as follows. For each doodle category, we calculated the percentage of questions answered correctly using that doodle (as a percentage of all questions where that doodle category appeared). Those percentages are given in Table 4.2. The "Data Points" column in that table indicates the total number of occurrences of each doodle type. The maximum number of possible data points for each doodle type is 56 students × 12 MCQs = 672.

Not surprisingly, Table 4.2 indicates that, if a student carefully traces through the code (S, T, or O), thus documenting changes in variables, the likelihood of getting the correct answer is high. In contrast, not doodling (B) only leads to the correct answer 50% of the time. While these statistics will surprise few teachers, these are very useful statistics for teachers to quote to their students.

Note that a student may use more than one doodle category to answer a question. Therefore some categories, which have little to do with actually determining the correct answer, may appear in conjunction with another doodle category that is primarily responsible for answering the question (e.g. The position doodle by itself is unlikely to lead to a correct answer). This may also explain the 100% success of the Keeping Tally doodle (K). In any event, the number of data points for the keeping tally doodle is too low to be considered significant.



1. Consider the following code fragment:

```
int x[] = {2, 1, 4, 5, 7};
int length = 5;
int limit = 3;
int i = 0;
int sum = 0;

while ( (sum<limit) && (i< length) )
{
     ++i;
     sum += x[i];
}
```

What value is in the variable "i" after this code is executed?

a) 0
b) 1
[✓ c)] 2
d) 3

sum=0 i=0

	sum	lim	i	len
1	0	3	0	5
			1	5
2		3		5
3		3		5
4		3		

Sum =

sum = 1
sum = 4 + 1 = 5

Figure 4.1 A student's doodles for Question 1

Code	Name	Description	Examples
A	Alternate Answer	Student changed their answer to this question	~~e.~~ d.
B	Blank Page	No doodles for this question	

C	Computation	An arithmetic or Boolean computation. Rewriting a comparison was not counted as a computation (see N).	`3 + 5`
			`3==5 false`

E	Extraneous Marks	Markings that appear meaningless or are ambiguous (i.e. could not be definitively characterized by researcher). Includes such things as: circled array elements; meaningless arrows; dots on the page, miscellaneous crossed out numbers, etc.) Does not include a crossed out copy of a trace (an obvious re-do of the trace)	. - |
K	Keeping Tally	Some value being counted multiple times (specific variable not indicated)	||||| ~~||||~~

N	Number	Shows value of a variable. Most frequently in a comparison. Distinct from S or T below. Values characterized as numbers were associated with variables whose values didn't change in the code fragment.	`0   5`
			`sum<limit`
			- or -
			`x.length 6`

O	Odd Trace	Odd kind of trace (i.e. used arrows, couldn't be characterized as either S or T below) but appeared to be a trace. Consistent with the example given here, these doodles may be pictorial representations of arrays, "before" and "after" operations on the array.	`0 1 2`
			`/ /`
			`1 2`

P	Position	Picture of correspondence between position (index) of array element and value of element
		Note: in example, top row is indices, bottom row is printed array as shown as question paper
			`0 1 2 3`
			`x={2 4 6 8}`

S	Synchronized Trace	Shows the values of multiple values every time one of them changes. Essentially a table. In some cases, students also devote table columns to "variables" that do not change during the execution of the given code.	`i | sum`
			`0 |  0`
			`0 |  3`
			`1 |  3`

T	Trace	Shows the values of a variable (not necessarily in table form) as it changes (i.e. shows more than 1 value for at least 1 variable)
		- or -
		a variable's value has been overwritten with a new value
			i1 = ~~1 2 3~~ 4
			- or -
			`i1 = 1`
			`i1 = 2`
			- or -
			`i1 0 1`

U	Underlined	Some part of the question was either underlined or shaded for emphasis by student	
X	Ruled out	One or more alternative answers were crossed out so that answer appeared to be selected by elimination.	

Table 4.1. Categorization of Doodles

Doodle Category	%Correct	Data Points
Keeping Tally (K)	100	6
Odd Trace (O)	78	23
Synchronized Trace (S)	77	73
Trace (T)	75	215
Alternate answer (A)	69	26
Position (P)	64	75
Number (N)	70	189
Computation (C)	60	30
X-ruled out (X)	60	60
Extraneous marks (E)	57	89
Underlined (U)	52	44
Blank Page (B)	50	256

Table 4.2: Percentage of correct answers when students (N=56) use a particular doodle type.

4.3 Blank Page Doodles

One category in Table 4.1 is the "blank page" doodle ("B") which, in fact, indicates that there were no doodles for that question. Given the propensity of experienced programmers to doodle, it would be reasonable to expect that novice programmers do likewise, but this is not always the case. For example, Thomas, Ratcliffe, and Thomasson (2004) have described their experiences with encouraging students to doodle. They report that, when a student approaches an instructor with a problem, the student is "often impatient when the instructor resorts to drawing a diagram, then amazed that the approach works" [p. 250]. They also report that, on one occasion when they tested students, and they provided each student with a piece of paper upon which the student was free to doodle, almost two thirds of the returned pages were blank.

In this study, of the 56 students for whom doodle data was examined, one fifth of them did not make any doodles (B) while answering Question 2, and over half (55%) did not doodle for Question 8. These relative doodling rates for Question 2 and 8 appear counter-intuitive. Unlike the novices in this study, anecdote suggests that experienced programmers habitually doodle when trying to understand a difficult piece of code. Question 2 is easier than Question 8, but the students we studied are less inclined to doodle for the harder question. (This issue is addressed further in the discussion of the Question 8 transcripts.)

Davies (1996) reports that, when writing programs, experts make extensive use of external records to manage information, whereas novices rely heavily upon their short-term working memory. From this working group study, it would seem that novices also rely heavily on their short-term working memory when attempting to trace code.

The actual frequency of blank doodles for Question 2 and 8 is likely to be an underestimate of the general lack of effective doodling. It would seem unlikely that any student who merely used the position doodle, and perhaps some of the other doodle categories, achieved any advantage in answering Questions 2 and 8. Therefore, the percentage of students who made no effective doodle is probably even higher for Questions 2 and 8.

There are three possible explanations why a student could answer a MCQ with a "B" doodle:

1) The MCQ is relatively simple.
2) The student has internalized a sophisticated reasoning strategy for answering that type of MCQ.
3) The student is either guessing, or has heuristics for selecting a plausible answer without genuinely understanding the MCQ, which is essentially an educated guess.

Given the high difficulty of Question 8, it seems unlikely that the first of the above explanations is plausible. An assessment of the other two explanations requires a closer look at transcript data, which is done in the next section.

5. TRANSCRIPTS

Having seen the student performance data, and having seen their doodling patterns, this section throws light upon that earlier analysis by an examination of the interview transcripts. Whereas the other sections were primarily quantitative, this section is primarily qualitative.

5.1 Walkthroughs

In the following examination of the transcripts, there is frequent reference to "walkthrough". We use this term to describe any transcript for a question where the student hand executes the code in meticulous detail. Figure 5.1 contains a walkthrough for Question 2.

Perkins et. al (1989) refer to walkthroughs as "close tracking". They note that, while close tracking is an important skills for diagnosing bugs, it is mentally demanding, and therefore many students do not track their code carefully. Perkins et. al. also note that bugs are frequently overlooked during tracking because the student projects their own intentions onto the code.

The transcript analysis in this section was done entirely separate from the doodle analysis. It seems plausible that a walkthrough in a transcript is evidence for a detailed style of doodling, but we have not investigated that connection.

5.2 Higher Level Reasoning

There is an extensive literature on mental models for programmers, where programs are represented at a level higher than the code itself. Much of that literature focuses upon the concept of a schema [Soloway and Ehrlich, 1984; Rist, 1986 & 2004; Detienne, 1990]. Schemas are an abstract solution to a programming problem, that can be applied in many situations. Closely related concepts including "plans", "templates", "idioms", and "patterns" [Clancy and Linn, 1999].

Much of the literature on schema focuses upon the writing of programs, but schema are also used in models of program comprehension. In some models of reading, comprehension is a "bottom-up" process, where elements of the text suggest the schema that may have been used by the author of the program. Some other models are "top-down", where schema suggest elements to look for in the program text. Other models allow a mix of "bottom-up" and "top-down" processes. Schema models correctly predict that an expert programmer will understand/memorize more quickly code that conforms to a schema than code that does not conform [Soloway, Adelson, and Ehrlich, 1988].

Some have advocated that students should be taught schema explicitly [Soloway, 1986; Clancy and Linn, 1999]. Recently, it has been advocated that students be explicitly taught to recognize and use ten common roles of variables [Kuittinen and Sajaniemi, 2004].

The study by this working group was designed principally to benchmark student performance across institutions and countries. The design of the study does not lend itself to a detailed analysis of any higher-level comprehension strategies used by students. The students were merely asked to find the correct answer to each multiple choice question, not articulate any higher-level reasoning process. However, some cautious inferences can be made from the transcripts. For example, if a student chooses an answer, without a complete walkthrough, then either the student has made a guess, or the student has employed some sort of higher-level reasoning strategy. The converse, however, is not necessarily true. If a student employs a thorough walkthrough to find the answer, it cannot be inferred that the student did not make any higher-level inference about the code. For example, after one student had chosen their (incorrect) answer for Question 2, via a walkthrough, the following exchange occurred between the interviewer and the student:

Interviewer: "*When you are doing that one you're stepping through the code line-by-line, did you form an intuitive idea as to what the program is doing*?"

Student: "*Um, yeah, I knew that it would count the number of equal elements. I wasn't entirely sure at a glance what the less-thans and greater-thans would do*."

Even though a detailed analysis of higher-level comprehension strategies is not possible in this study design, the transcripts do lend themselves to an investigation of one important issue: whether higher-performing students use a qualitatively different approach to lower-performing students. It could be that all students are using the same qualitative approach, but the lower performing students make more errors.

5.3 Guessing

Multiple-choice exams are frequently criticized as being answerable by guesswork. We looked at the transcripts to see what evidence there was for guessing. Two forms of guessing were observed in the transcripts. Students sometimes made an educated guess. For example, a student might perform an abortive walkthrough, or some other process, followed by a statement like "so it must be A or C – I'm going to guess A". A guess was deemed to have occurred when a student made a statement like "I'm just guessing" or "I'm completely lost, I just picked one".

In the transcripts, we found evidence for guessing in 10% of all student answers to questions, with approximately half of those being educated guesses and half pure guesses. However, guessing rates per question vary widely. We detected guessing at levels higher than 10% in five questions: Question 10 (30%), Question 12 (16%), Question 8 (15%), Question 6 (13%), and Question 9 (12%). We note that most of these were identified as difficult questions in the performance analysis. We did not detect any evidence of guessing for Question 7. As expected, it was observed that those students who guessed significantly more than their peers did perform noticeably more poorly.

This time I'm going to look to see what the question wants first. So trying to find the value of count. So now I'm looking to see what each variable is set to and what's happening inside the program. I'm going to go ahead and write down that x1's length is 4 and x2's length is 4. And then e loop, 4 is greater than 0 and 4 is greater than 0. So inside the loop, the, checking to see if the ... [indecipherable] .. *checking, noticing that I didn't subtract 1 from each of i1 and i2. So changing i1 to 3 and i2 to 3. So when the 3rd spot, last spot of each array, they're equal, so it enters the first if loop so count is now equal to 1; i1 equals 2 and i2 equals 2. Now we're going through the loop again. 2 is greater than 0 and 2 is greater than 0 so we're inside the loop. Checking the first "if" are the second indexes equal to each other? no. So going to the else if and 'is 4 less than 5?', which is true so i2 is subtracted 1. I2 now equals 1 and we're going back to the while loop again. 2 is greater than 0 and 1 is greater than 0. So checking the first if 'is 4 equal to 2?', which is false so checking the second if. "Is 4 less than 2?', which is false so doing the else statement which is subtract 1 from i1. so i1 now equals 1. So going through the while loop again. 1 is greater than 0 and 1 is greater than 0. So checking the first if "is 2 equal to 2?" which is true so increment count by 1. Count is now 2. Subtract 1 from i1 which becomes 0 and i2 which becomes 0. So now the while loop fails because 0 is not greater than 0. So it asks for the value of count which is 2.*

Figure 5.1 A complete transcript for one student as they correctly answer Question 2 by a walkthrough.

5.4 Question 2 by Quartile

Recall that Question 2 is a fixed-code question of medium difficulty. In this subsection, we examine student transcripts on that question by performance quartile.

Of the thirty seven transcripts, twenty were set aside from the analysis of Question 2, for the following reasons. In four cases, there was either no recording or a partial recording for Question 2, due to errors in the use of equipment. In two cases, the student responses to all 12 questions had not been recorded in the performance database, so the students could not be allocated to a quartile. In seven cases, there was either no clear articulation in the transcript of the answer chosen by the student, or there was a mismatch between the apparent choice of answer in the transcript and the answer recorded in the database (in at least one case, it is likely that the student verbalized one choice of answer, but marked down another choice on their answer sheet). In seven cases, the student had done the test prior to the interview, as part of the performance data gathering, and the interview was a "debrief". That is, the interview was the students' recollection of how they went about answering the question.

5.4.1 Upper Quartile Students

There were eight transcripts from upper quartile students. All students answered correctly, and all walked through the code meticulously. All these students articulated their walkthrough with sufficient detail so that a reader of the transcript can follow the reasoning (as in Figure 5.1).

While working out their answer, none of these students volunteered any realization of the intent of the code, to count the number of identical elements in the two arrays (in positions higher than position zero).

5.4.2 Second Quartile Students

There were five transcripts from second quartile students. Three students answered the question correctly. Two of those students answered via a meticulous walkthrough with the same clarity as the upper quartile students. The third student gave a brief and unclear walkthrough:

> "*x1 is an array with four integers, x2 is an array with four integers. i1 equals three and i2 equals three, i1 and i2 are greater than nought so equal three and x1 equals four in this statement and x2 equals five in this statement. i1 equals 2 and i2 equals 2 still greater than zero, so x1 will equal two and x2 will equal 2… therefore the answer will be B.*"

Of the two students who answered incorrectly, one chose option A. This is the option a student would choose if they grasped the general intent of the code, but did not realize that the comparison of elements in the arrays stops without examining the first elements of the arrays. Unlike the upper quartile students, this student did make explicit and unprovoked comments about the intent of the code. Early in his deliberations, the student commented that the code used "*meaningless variable names*". Midway through deliberations, he comments, "*I can see that's what it's doing but I do it slowly so I don't stuff it up*". Having reached the incorrect decision that the answer is option A, he volunteers, "*I can see that that was going to happen*", adding that the code "*pretty much counts similar digits in the arrays*". This student demonstrated a style of reading that we would like to encourage in students, where the student abstracts from the lower-level code to a higher-level schema. However, because he did not check carefully, he missed that the counting stops without examining the first positions in the arrays.

The remaining student chose distracter C. While the student appeared to walk through the code, the transcript is brief and unclear:

"*There are 2 variables i1 and i2 that start at 3. Two arrays x1 and x2. Conditional loop which loops while they are both greater than zero , compares x1 and x2 the elements at position i1 and i2 which are 3, sees if they are equal, which they are. It increments count and decrements i1 and i2, goes through loop again. And compares the second elements of x1 and x2, but this time they are different and x1 is less than x2 so this time it decrements i2 and goes through loop again. This time compares the ..... second in x1 and the third in x2 and they aren't equal and again decrement i2 – this time it is 0 so false so count was only incremented once*."

5.4.3 Third Quartile Students

There were two transcripts from third quartile students. Both students walked through the code, but incorrectly chose option A.

One of these students failed to manage the walkthrough well, and twice backtracked after realizing that an error had been made. At the third attempt, the student failed to track correctly the value of the variable "i2":

"*So first they are three, i1 and i2. We take the sevens, they are equal, count becomes one. And now they become 2, and then comes... yes they become twos, x1 and x2 We take 4 and 5, go to the else-if. i2 is decreased by one. It becomes one. And i2 is still 2. And then we take the second, so 4 and 2, and go to the last section. And there i1 is decreased. i1 becomes 1 and i2 2*."

The above transcript continues, but the error has been made.

The other student demonstrated a fragile grasp of the difference between a position in an array, and the contents of that position:

"*... So I have incremented count which would be from 0 to 1, subtract i1 and subtract i2, so they've moved the pointers.* By "pointers", the student means variables i1 and i2. *I've moved the pointers so now x1 pointer would be on 4 and in x2 the pointer would be on 5"*. The student appears to mean that x1[i1] contains 4 and x2[i2] contains 5. However, he then goes on to say "*... they are both equal still so, probably do the first one again. So count is 2*". By "both equal still", the student is asserting that the Boolean condition x1[i1] == x2[i2] is true, which is not only incorrect, but contradicts what he said immediately before, "x1 pointer would be on 4 and in x2 the pointer would be on 5". He incorrectly interpreted the Boolean condition as "i1 == i2".

Many teachers would recognize this student's problem as a common one among novices, and it is also recognized in the literature [du Boulay, 1989]. For this student, the confusion over array position versus array contents is more a case of fragility rather than a complete misconception, since the student did answer five other questions correctly. In fact, the student incorrectly answered the first four questions because of this confusion, before realizing his error, and went on to correctly answer 5 of the remaining 8 MCQs, which included the harder MCQs.

5.4.4 Bottom Quartile Students

There were two transcripts from bottom quartile students. Both students admitted to guessing.

5.4.5 Accounts of Question 2 by Debrief Students

As mentioned earlier, some transcripts were not included in the above quartile analysis of Question 2, because they were the students' recollection of how they had gone about answering the question earlier. However, it is interesting to examine those debriefs, for insight into how the students view the process. The following are taken from the debriefs of some third quartile students, who all answered Question 2 correctly:

1) "*... and basically what I did is that I tried to take all the initial values and write them down so I wouldn't lose track of what I was doing if I thought about too much at once. And again I went through loops, and I tried to keep track of where the variables were changing, and I'd make a little notation on the side*".
2) "*Its the exact same type of problem in question 2, ... its the same way I try to solve it ... writing down the variables, and for every iteration ... try to figure out what the variables are ... its just, just a way of working that's similar to using System.out.println to figure out the values of the variables. That's the method that is used manually, I imagine. That's recurring in many if these questions - its all about putting some numbers into your head ... follow the structure of the program and see what happens. And try to remember the numbers ... all the questions are more or less like question 1*".
3) "*The challenge is to keep a mental picture, keep on watching ... the state of the variables, depending on how the comparisons work out ... over all, I find them quite difficult ... I spend some time thinking about what happens in the first run through the loop what happens in the second and keep an eye on the individual ... counters ...*".

5.4.6 Discussion of Question 2 Transcripts

Due to the small number of transcripts that could be analyzed, no firm conclusions can be drawn. However, some interesting tentative observations can be made, which could be confirmed by collecting more transcripts.

Most students in the top three quartiles answered the question by a walkthrough. The principal difference between top quartile students and the other two quartiles is that the top quartile students were more meticulous in the walkthrough process, and therefore made fewer errors.

Few students in any quartile reasoned explicitly at a higher level than the walkthrough. That is, few students articulated the intent of the code, to count the number of common elements in portions of the two arrays.

5.5 Question 8 by Quartile

Recall that Question 8 is a skeleton-code MCQ, and one of the harder questions in the set of 12. In this subsection, we examine student transcripts on that question by performance quartile.

Of the thirty seven transcripts, twenty four were set aside from the analysis of Question 8, for the same reasons that transcripts were set aside for Question 2.

In the earlier performance data analysis, it was shown that distracter D was very popular with students. This distracter has the incorrect loop termination condition `j<x.length-1`. We therefore decided to concentrate our analysis on those students who first eliminated options A and B before deciding between either D or the correct option, C.

5.5.1 Upper Quartile Students

There were four transcripts from upper quartile students. One of these students chose option B, and another narrowed his choice to B and D before choosing D. Those two transcripts will be ignored in the rest of this analysis. Of the remaining two students, one correctly chose option C, and the other chose distracter D

The student who answered correctly used a long walkthrough. Less than 10% of the way into the two-page transcript of the walkthrough, that student says:

> "*... So I'm just going to go ahead and trial and error. I'll set it equal, I'll put in the A, the A answer.* "

The student does that, and proceeds in a manner similar to that of the transcript in Figure 5.1 About 40% of the way through the two-page transcript, the student starts the walkthrough again:

> "*I'm just going to start over because I didn't write down enough.*"

The student proceeds as before. At the turning from page 1 to page 2 of the transcript, which is early in this second walkthrough, the student makes an inference about the code in option A:

> "*Actually the problem with putting in that code would be that it starts at the beginning of the array every time for the second value, the value of j. ... So it can't be A. So now I'm trying ... but I'm not going to bother with B because it's starting out at the beginning of the array for j too. So now I'm looking at C.*"

Early in the walkthrough of option C, the student becomes more confident:

> "*I don't see any reason why this one shouldn't work so I'm going to go ahead and try the last value when i equals 4. j equals 5, so 5 is less than 6. We go inside the for loop. Is 1 greater than 3? No. So j equals 6. 6 is not less than 6 and i would get incremented and 5 is not less than 5. So it seems that one would work. I'm going to go ahead and try D real fast. I see the only difference is that in j, in the for loop with j, the length is equal to, is compared to length-1. So I'm just going to try it real fast but I think it's going to end up skipping a spot. So i is 0, 0 is less than 5, true. j equals 0 plus 1, so 1. 1 is less than 5. I'm just going to go ahead and skip to where j is 4. So is 4 less than 1? It's true. And j would get incremented but it would be 5 but that doesn't pass the for loop test so it would skip the last value in the array so D would not work ...*"

In the above transcript, we see signs of an emerging understanding of the concept and importance of boundary conditions.

The remaining student, who narrowed the choice to C and D before incorrectly choosing D, reasoned much more quickly about his choice:

> "*If you try to go all way to position 6, we'll have an array index out of bounds error, so want it to go to 6-1 because that'll be the actual array index*".

It is difficult to see how students could incorrectly reason like this in Question 8, but go on to answer the very next question correctly, when it also contains a loop termination condition. However, as Table 3.3 shows, only 51% of students answered Question 8 correctly, but 73% answered Question 9 correctly. As we speculated earlier, perhaps when students are answering Question 8 they are distracted by the loop condition on the same page, in the outer loop: `i<x.length-1.`

5.5.2 Second Quartile Students

In the earlier performance data analysis, it was shown that options C and D were almost equally popular with second quartile students. There were six examinable transcripts from these students. All six students narrowed their choice to option C or D, and two correctly chose C:

1) "*It has to be either C or D. Does it go until the end? Or doesn't it? But it really should go until the end, shouldn't it? Yes, it should, so that's C.*"
2) "*...so the problem is, is it x.length or x.length-1? So, then it goes to x.length-1. So you look at the number up to the second last number. The next one should look at every number, should look at every number plus the last number which is ...* [indecipherable] *... If you say x.length-1 you're not going over two numbers so this is the correct answer. C is the correct answe*r".

Of the four students who chose option D, one admitted to guessing. The explanations offered by two of the remaining three students for choosing option D seem little different from the explanations given by the students who chose option C:

1) "*And you need to compare until you get to the end of the array and the last element of the array is length-1 ...*".
2) "*... you want to go to the end of the array so that's x.length-1*".

The remaining student was quoted in the paper earlier. This is the student who admitted to choosing option D because that loop condition was most like the outer loop condition:

> "*Yeah I chose this one because I thought it would be the same, if your looking at the same number of values in here*."

5.5.3 Third Quartile Students

There were two examinable transcripts from these students. Both narrowed their choice to option C or D, but then incorrectly chose D:

1) "*The last element is like length-1 ... so then it is D.*"
2) The student reasoned that if option C was used *"... it would sort of causes segmentation error*".

5.5.4 Bottom Quartile Students

There was one transcript from a bottom quartile student, and the student admitted to guessing.

5.5.5 Discussion of Transcript data for Question 8

As with Question 2, few firm conclusions can be drawn about Question 8, due to the small number of transcripts that could be analyzed.

In choosing between options C and D, students manifest a nascent ability to reason at a higher level. At this stage of their development, many reason incorrectly, and choose option D. Given that they were reasoning at a higher level, it is perhaps not surprising that many students did not doodle when they attempted Question 8.

While students in all quartiles struggled to choose between options C or D, the performance data shows that students in the top three quartiles were consistently able to eliminate options A and B. Therefore, students are capable of some degree of analytical thinking on these skeleton-code MCQs. However, that many students struggle to correctly choose option C over D is evidence that these analytical skills are, at this stage of their development, fragile.

6. GENERAL DISCUSSION

Having looked at all the evidence available to this study, we now make some general observations.

6.1 Reading versus Writing

The question arises as to what relevance there is between the comprehension tasks studied by this working group and the ability of students to write code.

Even when our principal aim is to teach students to write code, we require students to learn by reading code. In our classrooms we typically place example code before students, to illustrate general principles. In so doing, we assume our students can read and understand those examples. When we exhort students to read the textbook, we assume that students will be able to understand the examples in that book.

Perkins et. al (1989) claim that the ability to perform a walkthrough is an important skill for diagnosing bugs, and therefore the ability to review code is an important skill in writing code. Soloway (1986) claims that, among many other abilities, skilled programmers carry out frequent "mental simulations", of both abstract designs-in-progress and code being enhanced, as a check against unwanted dynamic interactions between components of the system. He argues that such simulation strategies should be taught explicitly to students. Many of our teaching traditions date back to the era of punch cards. In the days of overnight batch runs, there was little need to explicitly encourage students to carefully check their code before submitting it for a batch run, as a careless error could waste a whole day. In an era where the next test-run is only a mouse-click away, we need to place greater explicit emphasis on mental simulation as part of the process of writing code.

Wiedenbeck [1985] found that expert programmers carry out low-level programming-related activities faster than novices. Such activities include identifying syntax errors in a single line of code, and assessing the correctness of loops containing less than 10 lines of code. Wiedenbeck concluded that the expert programmers had "automated" these processes, so that little mental attention was required. This automation allowed the experts to concentrate on higher-level problem-solving tasks. As a consequence of these findings, Wiedenbeck suggested that the teaching of novice programmers should "stress continuous practice with basic materials" until the novices have automated the practice [p. 389]. Therefore, perhaps an early emphasis on program comprehension and tracing, with the aim of automating basic skills, might then free the minds of students to concentrate on problem-solving.

And of course, at some time of their career, many programmers will maintain programs written by others, where program comprehension skills are vital [Deimel and Naveda, 1990].

6.2 Misconceptions

Spohrer and Soloway [1986, 1989] collected data about bugs in programs written by novices, where the bugs could be attributed to misconceptions about programming constructs. They concluded that mistakes due to misconceptions were not as widespread as was generally believed.

In this study, we also see few comprehension errors due to misconceptions. In only one of the twelve multiple choice questions did there appear to be a frequent misconception about a programming construct. That was in Question 6, where students did not completely understand the semantics of "return". Also, in the transcript of one student, who scored 5 out of 12, there was evidence of confusion between the position in an array and the contents of that position. It is therefore possible, nor would it be surprising, that students who scored a particularly low mark out of twelve on these multiple choice questions may have misconceptions. Among the top 75% of students, in all but one question, there is little evidence of construct misconceptions. However, the multiple choice questions in this study used only code fragments performing iterative processes on arrays. Students may have misconceptions associated with concepts not examined by these multiple choice questions.

6.3 Non-Idiomatic Questions

Recall that Question 2 is, in the terminology of this study, non-idiomatic. That is, it does not process the entire contents of the two arrays, as some programmers might first expect. The performance data shows that strong students tend to see that non-idiomatic detail, but weaker students do not. To what extent then, are MCQs like Question 2 "trick" questions?

As Perkins et. al (1989) argued, the ability to read what a piece of code actually does, rather than what we might think on a first quick inspection, is an important debugging skill. If students are forewarned that the pieces of code in the MCQs may be non-idiomatic (or buggy, depending upon one's point of view), then we regard such questions as legitimate. A weakness of this study is that only students at the "base" institution where the questions were written were guaranteed to be so forewarned. However, since the data for the base institution is broadly consistent with data from the satellite institutions, this weakness did not have a major impact on the study.

The authors of this working group acknowledge, however, that a weakness of non-idiomatic code is that it encourages students to mechanically hand execute code, when we would also like them to read with the aim of abstracting to schema.

6.4 Meaning and Context

When teaching novices to write programs, we emphasize the importance of using meaningful variable names. They are one type of beacon [Brooks, 1983; Wiedenbeck, 1986] which helps a programmer understand a piece of code. There are also rules of programming discourse [Soloway and Ehrlich, 1984] that set up expectations in the minds of programmers, as they read code. Beyond the program itself, observations by Pennington [1987] indicate that programmers also use a "cross-referencing" reading strategy, where they relate parts of a program to the problem domain. For example, when reading a program that tracks engineering wiring specifications, a programmer uses their real-world knowledge of wiring.

The MCQs in this study tend not to contain meaningful variable names, other beacons or conventions of discourse. Nor do these MCQs relate to a real-world domain. Therefore, to some extent, these MCQs are artificial problems, removed to some degree from the task of reading real programs. However, most of the code in these MCQs involve some sort of plausible operation on arrays. For example, Question 2 uses code for counting the number of common elements in two arrays (albeit non-idiomatically).

While the comprehension of real programs may involve higher-level strategies using beacons, rules of discourse, and cross-referencing, those skills do not replace the ability to systematically trace through code. Detienne and

Soloway [1990] found that when higher-level skills fail to elucidate program behavior, expert programmers resort to other skills, including simulating the program. Furthermore, the authors of this paper believe that it may be appropriate to first teach systematic tracing as a base skill, then allow students to build these higher-level comprehension skills upon that base.

6.5 Other Languages

Of the twelve participating institutions in this working group, eleven teach Java as a first language and one teaches C++. This restriction to only two languages may be an indication of their popularity as first programming languages. However, it may also be an indication that Java and C++ are particularly difficult to teach as first languages. It would be interesting to see this study replicated using other programming languages. We suspect, however, that the iterative process on arrays studied in this paper are generic to such a degree that students will perform similarly irrespective of what language they are taught.

6.6 Comparative Data

Much of the literature that studies novice programmers contrasts their performance on a given task with the performance of expert programmers. As a follow-up to this study, it would be interesting to collect data from expert programmers to see how their approach to answering these twelve multiple choice questions compares with that of the novices studied in this paper.

6.7 Availability of Data

The working group intends to eventually release portions of its data so that others may do their own analysis. Information abut data availability can be found at a web site [Lister, 2004].

7. CONCLUSION

This paper is a report from an ITiCSE 2004 working group. It builds on the work of the earlier McCracken working group. The McCracken group established that many first-year programming students cannot program at the conclusion of their introductory courses. While a popular explanation for that inability is that students cannot problem-solve, in the strong five-step sense defined by the McCracken group, this working group has established that many students lack knowledge and skills that are a precursor to problem-solving. These missing elements relate more to the ability of students to read code than to write it. Many of the students manifested a fragile ability to systematically analyze a short piece of code.

This working group does not argue that all students who manifest a weakness in problem-solving do so because of reading-related factors. We accept that a student who scores well on the type of tests used in this study, but who cannot write novel code of similar complexity, is most likely suffering from a weakness in problem solving. This working group merely makes the observation that any research project that aims to study problem-solving skills in novice programmers must include a mechanism to screen for subjects weak in precursor, reading-related skills.

This study assumed that the knowledge and skills that are the focus of this study are precursors to problem solving. The next logical research step is to examine that assumption, by combining the designs of the McCracken study and this study, to ask students to both solve tasks like those in this study, and also write code of similar complexity.

Acknowledgements

Eight of the twelve authors began discussing this study during the "Bootstrapping" and "Scaffolding" Projects, run by Sally Fincher, Marian Petre, and Josh Tenenberg. These projects were funded by the National Science Foundation (No. DUE-0122560). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. At some institutions, the authors of this paper did not teach the classes in which students were given the MCQ tests. We are therefore grateful for the cooperation of the teachers of those classes, David Bernstein (James Madison), Kerttu Pollari-Malmi (Helsinki), Mohamed Kerasha (Connecticut), Roger Simons and Ying Zhou (Rhode Island). The authors thank Mike McCracken for his advice and support, and also thank the organizers of the Leeds conference for providing the working group with excellent facilities. Finally, the author's thank George Burdell, for his thoughful comments on a draft of this paper.

REFERENCES

Brooks, R. (1983) Towards a theory of the comprehension of computer programs. International Journal of man-Machine Studies, **18**, pp. 543-554.

Clancy, M. and Linn, M. (1999), Patterns and Pedagogy. 30th Technical Symposium on Computer Science Education (SIGCSE 1999), New Orleans, LA USA. pp. 37-42.

Davies, S. (1996) Display-based problem solving strategies in computer programming. In Gray, W, and Boehm-Davis, D. (Eds) Empirical Studies of Programmers: 6th Workshop. Ablex Publishing Corporation, NJ. pp. 59-76.

Detienne, F. (1990) Expert Programming Knowledge: A Schema-based Approach. In Hoc, J, Green, T, Samurcay, and Gilmore, D. (Eds) Psychology of Programming. Academic Press, London. pp 206-222.

Detienne, F, and Soloway, E. (1990) An empirically-derived control structure for the process of program understanding. Int. J. of Man-Machine Studies, **33**, pp. 323-342.

Deimel, L.E. & Naveda, J. F. (1990) Reading Computer Programs: Instructor's Guide and Exercises. Software Engineering Institute, Carnegie-Mellon University. http://www.deimel.org/comp_sci/reading_computer_programs.htm (August 2004) Updated bibliography at http://www2.umassd.edu/SWPI/ProcessBibliography/bib-codereading2.html (August 2004).

du Boulay, B. (1989) Some Difficulties of Learning to Program. In Soloway, E. and Spohrer, J., Eds. pp 283–299.

Ebel, R. and Frisbie, D. (1986) Essentials of Educational Measurement. Prentice Hall, Englewood Cliffs, NJ.

Haladyna, T. (1999) Developing and Validating Multiple-Choice Questions (2nd Edition), Lawrence Erlbaum Associates, Mahwah, NJ.

Kuittinen, M, and Sajaniemi, J. (2004) Teaching Roles of Variables in Elementary Programming Courses. 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'04), Leeds, UK. pp 57–61.

Linn, R. and Gronlund, N. (1995) Measurement and Assessment in Teaching, . Prentice Hall, Upper Saddle River, NJ.

Lister, R. (2004) Availability of working group data. http://www-staff.it.uts.edu.au/~raymond/leeds2004.

McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz, (2001) A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students, SIGCSE Bulletin, **33**(4). pp 125-140.

Pennington, N. (1987) Comprehension Strategies in Programming. In Olson, G., Sheppard, S., and Soloway, E. (Eds) Empirical Studies of Programmers: Second Workshop. Ablex, NJ, USA. pp 100-113.

Perkins, D. and Martin, F. (1986) Fragile Knowledge and Neglected Strategies in Novice Programmers. In Soloway, E. and Iyengar, S. (Eds) pp. 213-229.

Perkins, D, Hancock, C, Hobbs, R, Martin, F, and Simmons, R. (1989). Conditions of Learning in Novice Programmers. In Soloway, E. and Spohrer, J., Eds. pp 261–279.

Rist, R. S. (1986). Plans in Programming: Definition, Demonstration and Development. In Soloway, E. and Iyengar, S., Eds. pp 28-47.

Rist, R. (2004) Learning to Program: Schema Creation, Application, and Evaluation. In Fincher, S and Petre, M., Eds (2004) Computer Science Education Research. Swets & Zeitlinger.

Soloway, E. and Ehrlich, K (1984) Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, SE-10(5):595-609.

Soloway, E. (1986). Learning to program = Learning to construct mechanisms and explanations. Communications of the ACM, 29(9). pp. 850-858.

Soloway, E. and Iyengar, S., Eds (1986) Empirical Studies of Programmers. Ablex, NJ, USA.

Soloway, E, Adelson, B, and Ehrlich, K. (1988) Knowledge and Processes in the Comprehension of Computer Programs. In Glaser, M, Chi, R, Farr, M, Glaser, R (Eds) The Nature of Expertise. Lawrence Erlbaum Associates, Hillsdale, NJ, USA. pp 129-152.

Soloway, E. and and Spohrer, J, Eds (1989), Studying the Novice Programmer. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

Spohrer, J. and Soloway, E. (1986) Analyzing the High Frequency Bugs in Novice Programs. In Soloway, E. and Iyengar, S. (Eds) pp. 230-251.

Spohrer, J. and Soloway, E. (1989) Novice Mistakes: Are the Folk Wisdoms Correct? In Soloway, E. and Spohrer, J., Eds. pp 401–416.

Thomas, L. Ratcliffe, M., and Thomasson, B. (2004) Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results. 35th Technical Symposium on Computer Science Education (SIGCSE 2004), Norfolk, VA USA. pp. 250-254.

Wiedenbeck, S. (1985) Novice/expert differences in programming skills. Int. J. of Man-Machine Studies, **23**, pp. 383-390.

Wiedenbeck, S. (1986) Processes in Computer Program Comprehension. In Soloway, E. and Iyengar, S., Eds. pp 48-57.

Appendix A: The 12 Multiple Choice Questions

Questions were given to students in single column format, usually with one complete question per page. Here, the indenting of some questions has been altered to fit the journal format. Answers for the questions are given after the final question.

Question 1

Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 3;
int i = 0;
int sum = 0;
while ( (sum<limit) && (i<x.length)){
      ++i;
      sum += x[i];
}
```

What value is in the variable "i" after this code is executed?

a) 0
b) 1
c) 2
d) 3

Question 2.

Consider the following code fragment:

```
     int[] x1 = {1, 2, 4, 7};
     int[] x2 = {1, 2, 5, 7};
     int i1 = x1.length-1;
     int i2 = x2.length-1;
     int count = 0;
     while ((i1 > 0 ) && (i2 > 0 ))
     {
          if ( x1[i1] == x2[i2] )
          {
               ++count;
               --i1;
               --i2;
          }
          else if (x1[i1] < x2[i2])
          {
               --i2;
          }
          else
          {    // x1[i1] > x2[i2]
               --i1;
          }
     }
```

After the above while loop finishes, "count" contains what value?

a) 3
b) 2
c) 1
d) 0

Question 3.

Consider the following code fragment:

```
int [] x = {1, 2, 3, 3, 3};
boolean b[] = new boolean[x.length];

for ( int i = 0; i < b.length; ++i )
     b[i] = false;

for ( int i = 0; i < x.length; ++i )
     b[ x[i] ] = true;

int count = 0;

for (int i = 0; i < b.length; ++i )
{
     if ( b[i] == true ) ++count;
}
```

After this code is executed , "count" contains:

a) 1
b) 2
c) 3
d) 4
e) 5

Question 4.

Consider the following code fragment:

```
int[ ] x1 = {0, 1, 2, 3};
int[ ] x2 = {1, 2, 2, 3};
int i1 = 0;
int i2 = 0;
int count = 0;
while ( (i1 < x1.length) &&
        (i2 < x2.length))
{
     if ( x1[i1] == x2[i2] )
     {
          ++count;
          ++i2;
     }
     else if (x1[i1] < x2[i2])
     {
          ++i1;
     }
     else
     {    // x1[i1] > x2[i2]
          ++i2;
     }
}
```

After this code is executed, "count" contains:

a) 0
b) 1
c) 2
d) 3
e) 4

Question 5.
Consider the following code fragment:

```
int[ ] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;

while (i < j)
{
     temp = x[i];
     x[i] = x[j];
     x[j] = 2*temp;
     i++;
     j--;
}
```

After this code is executed, array "x" contains the values:

a) {3, 2, 2, 0}
b) {0, 1, 2, 3}
c) {3, 2, 1, 0}
d) {0, 2, 4, 6}
e) {6, 4, 2, 0}

Question 6.
The following method "`isSorted`" should return true if the array is sorted in ascending order. Otherwise, the method should return false:

```
public static boolean isSorted(int []x)
{
      //missing code goes here
}
```

Which of the following is the missing code from the method "isSorted" ?

```
(a)   boolean b = true;

      for (int i=0 ; i<x.length-1; i++)
      {
            if ( x[i ] > x[i+1] )
                  b = false;
            else
                  b = true;
      }
      return b;


(b)   for (int i=0; i<x.length-1; i++)
      {
            if (x[i ] > x[i+1] )
                  return false;
      }
      return true;


(c)   boolean b = false;

      for (int i=0; i<x.length-1; i++)
      {
            if (x[i] > x[i+1] )
                  b = false;
      }
      return b;


(d)   boolean b = false;

      for (int i=0;i<x.length-1;i++)
      {
            if (x[i ] > x[i+1] )
                  b = true;
      }
      return b;


(e)   for (int i=0;i<x.length-1;i++)
      {
            if (x[i ] > x[i+1] )
                  return true;
      }
      return false;
```

Question 7.
Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 7;
int i = 0;
int sum = 0;

while ( (sum<limit) && (i<x.length) )
{
      sum += x[i];
      ++i;
}
```

What value is in the variable "i" after this code is executed?

a) 0
b) 1
c) 2
d) 3
e) 4

Question 8.

If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array "x" that contains the following six numbers:

4 5 6 2 1 3

There are 10 inversions in that array, as:

```
x[0]=4   >   x[3]=2
x[0]=4   >   x[4]=1
x[0]=4   >   x[5]=3
x[1]=5   >   x[3]=2
x[1]=5   >   x[4]=1
x[1]=5   >   x[5]=3
x[2]=6   >   x[3]=2
x[2]=6   >   x[4]=1
x[2]=6   >   x[5]=3
x[3]=2   >   x[4]=1
```

The skeleton code below is intended to count the number of inversions in an array "x":

```
int inversionCount = 0;

for ( int i=0 ; i<x.length-1 ; i++ )
{
      for  xxxxxx
      {
            if (  x[i] > x[j] )
                  ++inversionCount;
      }
}
```

When the above code finishes, the variable "inversionCount" is intended to contain the number of inversions in array "x". Therefore, the "xxxxxx" in the above code should be replaced by:

a) `( int j=0  ; j<x.length  ; j++ )`
b) `( int j=0  ; j<x.length-1; j++ )`
c) `( int j=i+1; j<x.length  ; j++ )`
d) `( int j=i+1; j<x.length-1; j++ )`

Question 9.

The skeleton code below is intended to copy into an array of integers called "array2" any numbers in another integer array "array1" that are even numbers. For example, if "array1" contained the numbers:

array1: 4 5 6 2 1 3

then after the copying process, "array2" should contain in its first three places:

array2: 4 6 2

The following code assumes that "array2" is big enough to hold all the even numbers from "array1":

```
int a2 = 0;

for ( int a1=0 ; xxx1xxx ; ++a1 )
{
      // if array1[a1] is even
      if ( array1[a1] % 2 == 0 )
      {
            // array1[a1] is even,
            // so copy it
            xxx2xxx;
            xxx3xxx;
      }
}
```

The missing pieces of code "xxx1xxx", "xxx2xxx" and "xxx3xxx" in the above code should be replaced respectively by:

a)
```
a1<array1.length
++a2
array2[a2] = array1[a1]
```

b)
```
a1<array1.length
array2[a2] = array1[a1]
++a2
```

c)
```
a1<=array1.length
array2[a2] = array1[a1]
++a2
```

d)
```
a1<=array1.length
++a2
array2[a2] = array1[a1]
```

Hint: in *all four options above, the second and third parts are the same, just reversed.*

Question 10.
Consider the following code fragment:

```
int[] array1 = {2, 4, 1, 3};
int[] array2 = {0, 0, 0, 0};
int a2 = 0;

for (int a1=1; a1<array1.length; ++a1)
{
      if ( array1[a1] >= 2 )
      {
            array2[a2] = array1[a1];
            ++a2;
      }
}
```

After this code is executed, the array "array2" contains what values?

a) {4, 3, 0, 0}
b) {4, 1, 3, 0}
c) {2, 4, 3, 0}
d) {2, 4, 1, 3}

Question 11.
Suppose an array of integers "s" contains zero or more different positive integers, in ascending order, followed by a zero. For example:

```
      int[] s = {2, 4, 6, 8, 0};
   or int[] s = {0};
```

Consider the following "skeleton" code, where the sequences of "xxxxxx" are substitutes for the correct Java code:

```
      int pos = 0;
      while ( (xxxxxx) && (xxxxxx) )
            ++pos;
```

Suppose an integer variable "e" contains a positive integer. The purpose of the above code is to find the place in "s" occupied by the value stored in "e". Formally, when the above "while" loop terminates, the variable "pos" is determined as follows:

1. If the value stored in "e" is also stored in the array, then "pos" contains the index of that position. For example, if e=6 and s = {2, 4, 6, 8, 0}, then pos should equal 2.

2. If the value stored in "e" is NOT stored in the array, but the value in "e" is less than some of the values in the array then "pos" contains the index of the lowest position in the array where the value is larger than in "e". For example, if e=7 and s = {2, 4, 6, 8, 0}, then pos should equal 3.

3. If the value stored in "e" is larger than any value in "s", then "pos" contains the index of the position containing the zero. For example, if e=9 and s = {2, 4, 6, 8, 0}, then pos should equal 4.

The correct Boolean condition for the above "while" loop is:

```
(a)  (pos < e)      &&  (s[pos] != 0)
(b)  (pos != e)     &&  (s[pos] != 0)
(c)  (s[pos] < e)   &&  (  pos  != 0)
(d)  (s[pos] < e)   &&  (s[pos] != 0)
(e)  (s[pos] != e)  &&  (s[pos] != 0)
```

Question 12.
This question continues on from the previous question. Assuming we have found the position in the array "s" containing the same value stored in the variable "e", we now wish to write code that deletes that number from the array, but retains the ascending order of all remaining integers in the array. For example, given:

```
      s = {2, 4, 6, 8, 0};
      e = 6;
      pos = 2;
```

The desired outcome is to remove the 6 from "s" to give:

```
      s = {2, 4, 8, 0, 0};
```

Consider the following "skeleton" code, where "xxxxxx" is a substitute for the correct Java code:

```
      do {
            ++pos;
            xxxxxx;
      } while (s[pos] != 0 );
```

The correct replacement for "xxxxxx" is:

```
(a)  s[pos+1] = s[pos];
(b)  s[pos]   = s[pos+1];
(c)  s[pos]   = s[pos-1];
(d)  s[pos-1] = s[pos];
```
(e) None of the above

Correct options for the 12 questions

1	2	3	4	5	6	7	8	9	10	11	12
c	b	c	e	a	b	d	c	b	a	d	d

Appendix B. Analysis of individual questions

This appendix contains statistical results and analysis for each of the MCQs. The analyses for questions 2 and 8, found in the body of the text, are not repeated. Questions with similar structure are grouped in sections B.1 to B.3.

B.1 Fixed-code questions with `int` answers

Questions 1-4 and 7 all give an array and ask for the value of an int variable, which represents either the number of times something occurs in the array or the index of some position in the array. Theoretically, there is a finite set of possible answers – the set of indices for the given array, or the range from 0 to the total number of elements in the array.

Consider Question 1, for example. This question involves searching through an array, accumulating the sum of the array elements in a variable, and recording the index at which the sum reaches a certain limit. There are five elements in the array, so the possible index values include 0, 1, 2, 3, and 4. If the index value is incremented at the end of the loop, it might also be 5. Not all of these values are provided as choices: the options only include 0, 1, 2, and 3.

Student responses to this question are summarized in the graph in Figure B1, below. This graph shows that the top students (those in the first quartile) did well; over 90% of them chose C (the correct answer). Although the second and third quartiles don't do as well, they still have a strong preference for the correct answer. The question distinguishes between quartiles 1-3 and the weak students, who choose the correct answer and choice D with approximately equal frequency.

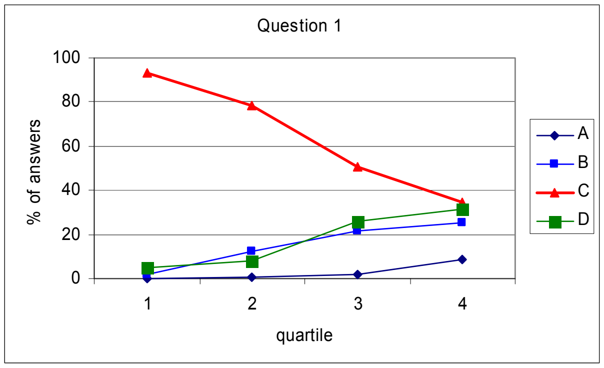


Figure B1: *Student responses to Question 1 by quartiles.*

Even the students in the fourth quartile know something, however: not all choices are equally popular. Choice A is much less popular than the rest, even among the students in the fourth quartile. The value of choice A is 0, and we hypothesize that even weak students can usually see that i++ changes the value of i at least once.

The discussion of Question 2 is in Section 4.1.

Question 3, another fixed-code question, involves manipulating two arrays. In this case, the second array, an array of booleans, is used to keep track of the distinct values contained in the first (int) array. Finally, the number of distinct values in the int array (each corresponding to true in the boolean array) is counted.

The number of elements in the array is 5, so the possible values for the number of distinct elements are 1, 2, 3, 4, and 5. These values are all given as choices.

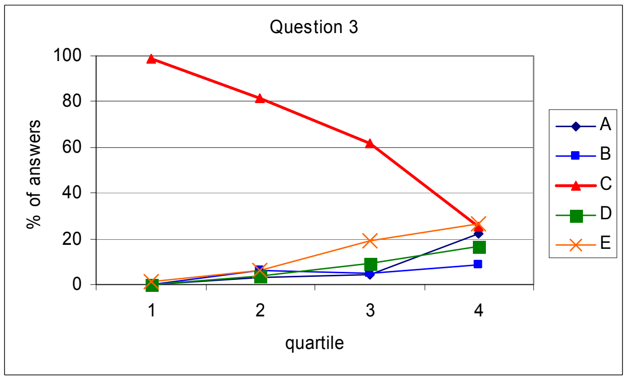


Figure B2: *Student responses to Question 3 by quartiles.*

Judging by the results, the students had little trouble with this one. Once again, the top students do very well, and quartiles 2-3 have a strong preference for the correct answer. Choice E becomes increasingly popular as we go from the top students down to the weak students. Choice E, "5," is the one a student would pick if he or she failed to understand the statement

```
b[x[i]] = true;
```

and simply assumed it was the same as

```
b[i] = true;
```

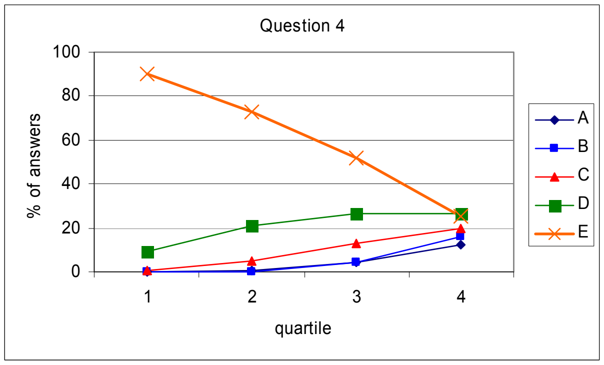


Figure B3: Student responses to Question 4 by quartiles.

Question 4, also a fixed-code question, is very similar to Question 2. It also involves comparing two arrays, looking for matching elements. Again, each array contains four elements.

There are three differences between Questions 2 and 4:

1. Question 4 starts at position 0 and increments the array index; Question 2 starts at the final position of the array and decrements the index.
2. Question 4 examines all positions in the array; Question 2 examines all but one.
3. When Question 4 finds a matching pair, it changes only one of the indices; Question 2 changes both. Thus, for example, suppose we have two arrays: {6, 0} and {6, 6}. The code in Question 4 finds two matches; the Question-2 code only finds one.

Because of these differences, the possible answers here range from 0 (e.g. {1, 2, 3, 4} and {5, 6, 7, 8}) to 4 (e.g. {2, 3, 4, 5} and {2, 2, 2, 2}.)

The choices given for this question are related to the most likely student misconceptions. The correct answer is 4 (choice E). If you think this code is counting duplicates in the same way as Question 2, you would get the answer 3 (which is choice D). If you see that the code is counting slightly differently, but think that it's still ignoring the first element of each array, you would also get 3. If you make both mistakes, you would get 2 (choice C).

In the graph for Question 4, we see the familiar pattern: the top students do very well, quartiles 2-3 do less well but still have a strong preference for the correct answer, and quartile 4 does even less well and does not have a preference for the correct answer.

We also see, however, that all four quartiles distinguish among the distracters, with choice D as the most popular, followed by choice C, consistent with the fact that the likely misconceptions lead to those answers. Choices A and B, which do not correspond to any of the likely misconceptions, are the least popular. And once again, 0 (here, choice A) is a very unpopular choice.

Here is the graph for question 7:

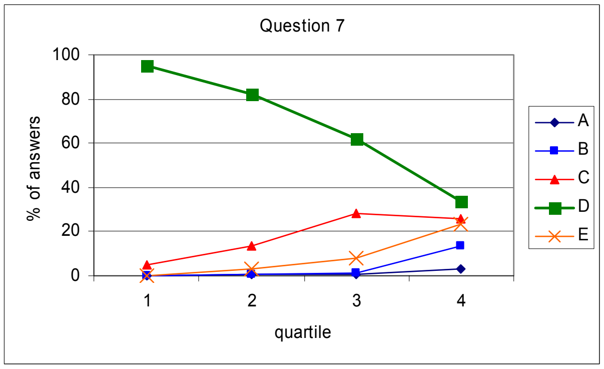


Figure B4: Student responses to Question 7 by quartiles.

Question 7 is very similar to Question 1. Both questions involve a single array. In both cases, the code searches through the array, accumulating the sum of the array elements and recording the index at which the sum reaches a certain limit. The array given in Question 1 is identical to the array given in Question 7, and in both cases, the code starts at position 0 of the array and works forward. In both cases, since the array has five elements, the possible values of the array index are 0, 1, 2, 3, and 4 (and 5, if the array index is incremented after being used).

There are three differences between the questions:

1. The value of the limit (3 for question 1, and 7 for question 7).
2. The order of the steps inside the while loop. In Question 1, the index is incremented first, then the element is added to the sum; in Question 7, the element is added first, and then the index is incremented.
3. the choices for question 7 include 4 (choice E), as well as 0, 1, 2, and 3.

Once again, the top students do very well, and the students in quartiles 2-3 do less well but have a strong preference for the correct answer. Here the students in quartile 4 actually have a slight preference for the correct choice as well. There are relatively few errors, and transcripts indicate that some of those were due to a tracing error, rather than a lack of understanding.

In summary, all these questions are similar: they all help us discriminate between the top three quartiles, on the one hand, and the weakest students on the other. Their trace-line graphs look similar, they are all easy questions for the top students, and in each case the fourth quartile students are the only ones who do not have a strong preference for the correct answer. Sometimes, as in Question 2, we can see evidence that students are assuming the code fits idioms they know (such as processing arrays starting from 0). And 0 always seems to be a bad distracter.

B.2 Fixed-code questions with array answers

Questions 5 and 10 are also fixed-code questions, and they also give an array and some code that processes the array. Instead of an int value, however, they ask for the resulting array.

Here is the graph for Question 5:

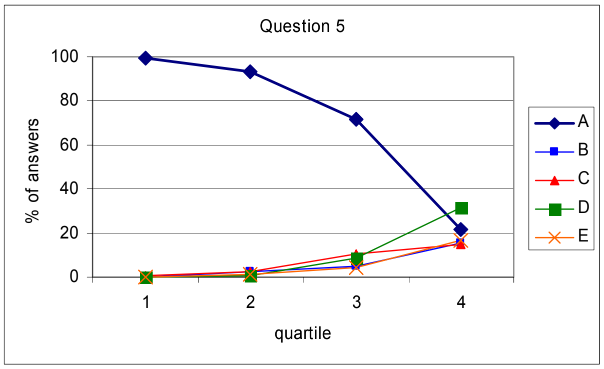


Figure B5: *Student responses to Question 5 by quartiles.*

The code for Question 5 takes one array and reverses the order of the elements in the array, except for one thing: the elements that were in the first half of the array are doubled before being moved to their new positions in the second half of the array. Thus, array {1, 3, 5, 7} would become {7, 5, 6, 2}.

The choices are related to possible student misconceptions. They include:

- The original array
- The array you would get if you reversed the original array, but didn't double any of its elements
- The array you would get if you reversed the original array and doubled *all* of its elements
- The array you would get if you doubled all of the array elements, but didn't reverse it. The correct answer.

This question discriminates even more clearly than the earlier questions between the top three quartiles and the weakest students. It was one of the easiest questions for the top three quartiles, as the graph shows: the trace line for the correct answer has a similar shape, but it is very high, particularly for quartiles 2 and 3. Over 95% of the students in the second quartile chose the correct answer. Even in the third quartile, more than 70% of the students chose the correct answer.

The first and second quartiles show no clear preference for any of the distracters; the third quartile very slightly prefers choice D (doubling the whole array without reversing it), and the fourth quartile prefers choice D even to the correct answer. Student notes on the exams indicate that even weak students can see that something is being multiplied by two, but they find the logic of the swapped elements harder to follow.

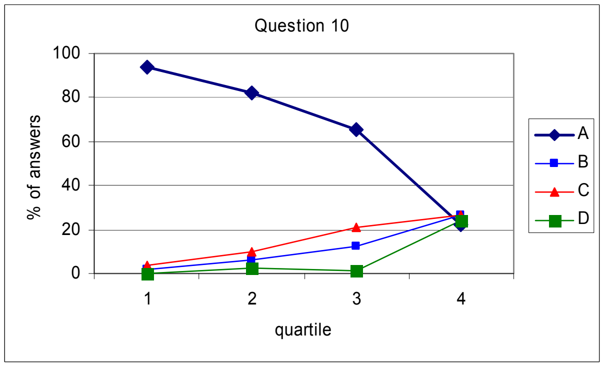


Figure B6: *Student responses to Question 10, by quartiles.*

Question 10 is the last of the fixed-code questions. It involves filtering an array: taking all numbers greater than or equal to two and copying them into a second array. The first element in the array is ignored, however.

The choices, again, are related to possible student misconceptions. They include:

- the array with all the elements from the original array that are greater than or equal to two (what you would get if you filtered, but didn't notice that the copying started with position 1)
- the array you would get if you started with position 1, but didn't filter
- the array you would get if you didn't filter *and* didn't notice that the copying started with position 1 (i.e., the original array)
- the correct answer.

Overall, 70% of the students solved this question correctly, and the students in Quartile 1 overwhelmingly preferred the correct answer (A). Students in Quartiles 2 and 3 also strongly preferred the correct answer. The weak students, on the other hand, evidently guessed: they chose all answers with equal frequency.

All but the bottom quartile noticed something amiss: choice D is the one that contains both of the two possible errors, and, as shown in Figure 10 it is very unpopular with all of Quartiles 1-3. In Quartiles 2-3, Choice C (the answer you get if you assume array processing includes the whole array) is somewhat more popular than choice B (the one you get if you notice that the first element of the array is being ignored, but you don't filter the array).

The students in the fourth quartile appear to be choosing almost at random, perhaps because they have no idioms to rely on.

In summary, the fixed-code questions all have a lot in common. Their trace-line graphs are very similar: the top students do very well, the students in the second and third quartiles do less well, but still strongly prefer the correct answer, and the students in the fourth quartile do not have a preference for the correct answer. We can sometimes draw conclusions based on which distracters the students choose. This evidence tends to support the idea that many students are influenced by standard idioms, such as looping through an array from position 0 to position length-1, but are not yet comfortable enough with those idioms to know when not to use them.

B.3 Skeleton-code questions

By this point, the reader might wonder if trace-line graphs always look the same. Two of the skeleton-code questions were relatively easy for the students, and their graphs are similar to the graphs for the fixed-code questions. The other three questions were consistently the most difficult, across quartiles and across institutions, and their graphs look quite different. Question 8, a difficult skeleton-code question, was presented in section 4, so is not presented here.

Let's begin with an easier question. Here's the graph for Question 9:

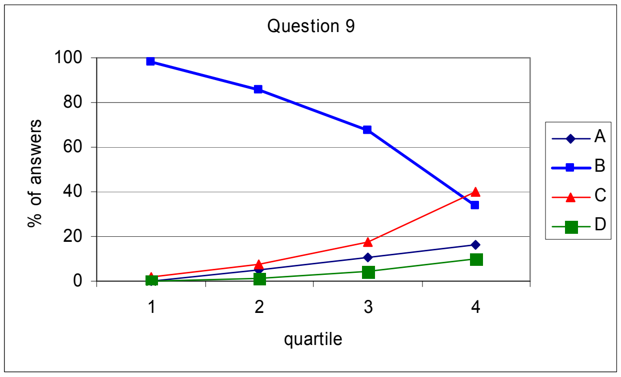


Figure B7: *Student responses to Question 9 by quartiles.*

In this question, students are told that a given code fragment is intended to copy the even numbers contained in one array into a second array; they are asked to fill in three blanks in the code.

The choices are clearly parallel, and if the layout is not enough to make this clear to the students, a hint is given. The students' attention is focused on two points:

1. Should the loop stop processing the array when the index is equal to the length of the array, or when it's equal to the length minus 1?
2. Should the array index of the second array be incremented at the beginning of the loop body or at the end?

The correct answer to both of these questions was the idiomatic one: increment at the end of the loop body, and stop processing at position length-1.

The choices included:

- o Stop processing at the right place, but increment at the beginning of the loop body
- o Stop processing in the right place and increment at the end of the loop body (the correct answer)
- o Stop processing in the wrong place and increment at the beginning of the loop body (two errors combined)
- o Stop processing in the wrong place but increment at the end of the loop body

This question was the easiest of the skeleton-code questions, and one of the easiest questions overall. Of the top quartile, 98% got the question right. What is more striking is that even the weak students could differentiate between answers where the loop counter was incremented in the beginning (B and C) and those where it was done at the end (A and D), favoring the correct order of statements. Given the number of students who guessed in some of the questions, this is even more noticeable.

The students' good performance on this question may be because the idiomatic, predictable answer is in fact the correct one. It may be because the answers are clearly distinguished from each other, and the students' attention is thus drawn to the key points. It is also possible that Question 10 had an effect. An alert student might find the answer to Question 9 in Question 10. Question 10 also involves filtering one array into another, but it is a fixed-code question, so the code is provided. On the other hand, it's arguable that a student who was alert enough to spot the similarity between these two questions would have got them right in any event.

For a very different question, here's the graph for Question 6:

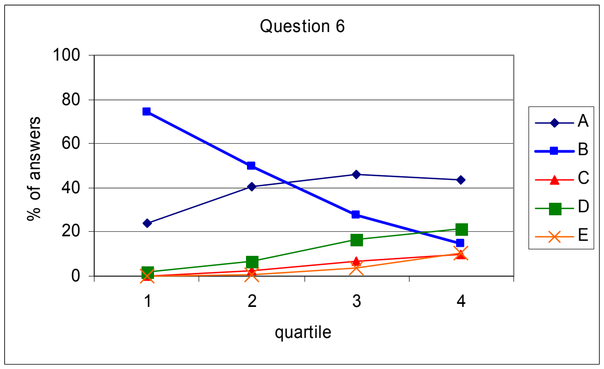


Figure B8: *Student responses to Question 6, by quartiles.*

This question asks the student to select the method body for a boolean method that takes an array of ints and returns true if the array is sorted in ascending order.

The most obvious difference about this graph is where the lines cross: the upper line for the correct answer (choice

B) crosses the trace line for one of the distracters between the second and third quartiles. The line also starts quite low: less than 80% of the students even in the first quartile chose the correct answer.

This question discriminates between the top and bottom students: quartiles 1 and 2 on the one hand, and quartiles 3 and 4 on the other. In addition, it reveals misconceptions that are shared even by some of the top students.

Students at all levels easily ruled out alternative C, which always returns false. They also ruled out alternatives D and E, which return true when the array is not sorted. Top students, however, chose the correct answer (B) almost 75% of the time, students in the second quartile chose distracter A almost as often as the correct answer, and students in the third and fourth quartiles preferred distracter A to the correct answer.

What explains the popularity of distracter A in this question? Even the top students chose distracter A more than 20% of the time. In fact, this was the most effective distracter on any question for the top students.

The transcripts suggest that students were misled due to preconceptions about the meaning of a return statement inside a loop. Choice B, the correct answer, has a return statement inside a loop; choice A does not. Students who chose distracter A were not always satisfied with it, but felt that it was "less bad" than the correct answer, B.

We found three interpretations:

1. When the `return` statement is encountered, it returns a value immediately and exits (the correct interpretation);
2. A `return` statement inside a loop returns multiple values;
3. A `return` statement inside a loop sets a return value each time through the loop and returns its final value.

The second interpretation was indicated by comments such as "It has to return only one true or false, this whole thing, in the end. Not many." The third interpretation was indicated by comments such as "you return false and later you return true, which is not right," and "always returns true," and "If that returns false it's still going to return true every time." Some students commented on the lack of a flag: "no identifiers," "maybe B is wrong because you are not initializing any variables."

The problems with return statements inside a loop also explain the difference between distracters D and E. As shown on the graph, D is preferred to E, even though they both have the same logical flaw (they both return true when the array is not sorted). D sets a flag, however, and E has a return statement inside a loop.

Finally, consider Questions 11 and 12. These questions are both related to the same topic, and Question 12 continues on from Question 11. Here's the graph for Question 11:

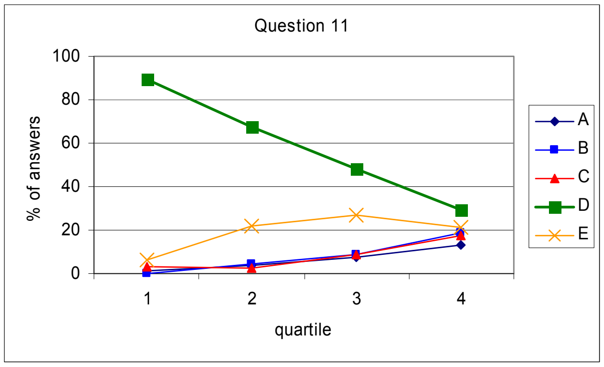


Figure B9: *Student responses to Question 11, by quartiles.*

As can be seen from the graph, this was an easier question for the students. It involves an array of integers, sorted in ascending order, with a "0" as the final element as a flag to indicate the end of the array. The code is designed to locate the position in this array where a particular new element should be inserted. Students were asked to choose the correct loop condition to fill in a blank in the code.

The choices focused on two things:

1. the difference between an index (`pos`) into an array and the element at that index (`s[pos]`).
2. the logic of the search. We can't stop searching until we find an element that's greater than the desired value. The correct condition is `while (s[pos]<e)`, not `while (s[pos] != e)`.

Like the fixed-code questions, this one discriminates between the top three quartiles, on the one hand, and the weakest students, on the other. The top students did very well. They rarely chose any of the distracters, although when they did, they had a slight preference for distracter E. Quartiles 2 and 3 did less well than quartile 1, but they still had a strong preference for the correct answer (D). When they did choose a wrong answer, they preferred distracter E to any of the others. In other words, they had no problem distinguishing between pos and s[pos] (indicated by their rejection of distracters A-C), but they were sometimes wrong about the search logic (indicated by the choice of distracter E).

The fourth-quartile students are much more likely to choose distracters A-C than students in quartiles 1-3, and less likely to choose the correct answer. In other words, they are more likely to be confused about the difference between pos and s[pos] than students in the other quartiles.

Finally, here's the trace-line graph for Question 12:

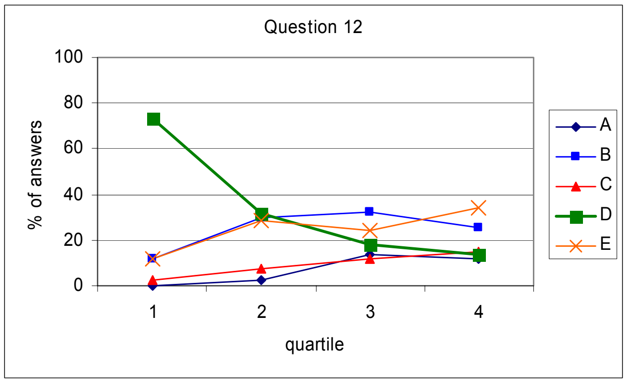


Figure B10: *Student responses to Question 12, by quartiles.*

This was the hardest question, and its graph is the most complex. Question 11 (like the fixed-code questions) separates the weak students from the rest; Question 12 separates the top from the mediocre students.

This question involves a single array of ints sorted in ascending order with a 0 following the last element of the array, as in Question 11. The code to be written here is supposed to delete an element in the given position from the array and shift the rest of the elements to the left to fill in the gap. Thus, if we delete the 7 from array {1,3,5,7,9,0}, the result should be {1,3,5,9,0,0}.

The choices focus on two points:

1. When the assignment statement x=y is executed, it is the value of x that is changed.
2. Inside the loop, the array index is incremented first.

Students who are confused about the direction of the assignment operator but notice that the index is incremented first would choose distracter C. Students who have a solid grasp on the assignment operator but fail to notice that the array index is incremented first would choose distracter B. This is the expected, idiomatic code. Distracter A contains both errors. D is the correct answer, and E is "none of the above." This is the only question that contained a "none of the above" option.

Top students chose the correct answer over 70% of the time, and when they did not, were equally likely to choose distracter B (the idiomatic answer) and distracter E (none of the above). They almost never selected A or C.

In quartiles 2-4, the correct answer is no longer the most popular. In quartile 2, it is approximately tied with distracters B and E; in quartile 3, distracters B and E are more popular than the correct answer, but B (the idiomatic answer) is still the most popular); and in the fourth quartile, distracter E (none of the above) is the most popular choice.

It appears that most students understand the assignment operator, but it's not unusual for them to assume that the index is incremented in the usual place, at the end of the loop body.

This question does not in itself seem much more difficult than the others. Some students may have given up because they didn't get Question 11 (although this question could probably be answered without reading Question 11). Because of the continuation, Question 12 appears to require more reading than the rest. A number of students were misled by the fact that the array index is incremented at the beginning of the loop body. Notably, this is the only question that contains a "none-of-the-above" choice, and that may have misled students: the very fact that they hadn't seen that choice before might make it more significant. And it might just be that students were tired by this point in the test. At least one of the transcripts indicated that by this point, the student had lost concentration and was simply guessing.