

Autogenerated Questions Paper Draft

Contents

1	Problem Description (needs work)	1
2	Input/output Flow	2
3	Common Functions	2
3.1	Difficulty	3
3.2	Unique Answer Guarantee	5
4	Testing (is this needed?)	5
5	Categories	5
5.1	Introductory Questions	6
5.1.1	Identifiers	6
5.1.2	Conditional Expressions	7
5.1.3	Expressions	8
5.2	Loops	9
5.2.1	Loop Counting	9
5.2.2	Loop Printing	12
5.3	Advanced	12
5.3.1	Switch Cases	13
5.3.2	Arrays	15
5.3.3	Functions	15
6	Future Improvements	15
7	Conclusion	15

1 Problem Description (needs work)

For any large introductory undergraduate course, testing becomes virtually impossible on a (clerical?) level, especially for a small examination such as an in-class quiz. However, technology allows instructors to develop and distribute exams in an online environment, easing the arduous task of passing out and collecting paper exams. However, extra problems are presented with online tests (especially those without a proctor), such as authentication, blah, and most

importantly cheating (reference). Some solutions have been discussed before (reference), but problems still remain. In an effort to prevent cheating, instructors may use different questions for different students, but the task of creating exam questions (let alone multiple ones) has been noted to be very intensive and time-consuming (reference). A possible solution to this problem has been presented by (reference), but this implementation is in the somewhat narrow field of pharmacology, and it doesn't discuss important details such as the methods for generating distractors or the affect of randomness on each question – which may be of interest to some as it can affect difficulty and the objective of some questions. Furthermore, there is no discussion of the reusability or additions of the system, making it seem completely isolated and specific [VERFIY THESE POINTS].

2 Input/output Flow

Given the executable file and a templated file for input, it becomes extremely easy to generate any number of questions repeatedly. In most cases, we wish to generate a single category of questions at a time, and this is the basis for the overall structure of the main program. It simply checks the user's input from the predefined file to read basic inputs such as the Main Category and Subcategory names (for identification in Moodle), the filename for output, the number of total questions to generate, the question set – or category – to generate from (such as Arrays, Functions, etc.), the base difficulty, and the max difficulty of questions. By having this input file, it becomes easy to slightly modify any parameters and repeatedly run the program.

However, this method becomes tedious if one wishes to generate questions from every category. For this reason, the executable program also accepts 7 command-line arguments (that, in any other case, are cumbersome and virtually unusable) which allows a small shell script to easily generate questions from every category in one command.

Once the program has finished, all output files generated in .txt format containing the GIFT-formatted questions, and they can be uploaded directly to the Moodle course with no extra effort. Each category is contained in its own file, and any difficulties selected within the category are automatically separated into their own question banks, allowing for exam questions to be selected by difficulty.

3 Common Functions

Each question, regardless of category, follows the same format: the title (including unique question number), a prompt to explain the question, the right answer, and a number of wrong answers (that can vary between categories). A standard question in GIFT looks something like this:

::Dynamic Expressions Level 1 #005::

Solve the following problem:

```
<pre>int item = 4;
int bar = item / 3 - 3;</pre>
```

What is the variable "bar" equal to?

```
{
=-2
~-0.5
~-1.667
~1.5
~0
}
```

NOTE: why am I describing how GIFT or Moodle works? This seems unnecessary and boring.

The title at the beginning (enclosed by two colons on each side) contains the subcategory name, "level" (i.e. difficulty) and a unique number (simply the current number of questions generated in this batch). This information is necessary in Moodle to differentiate between 1) different question banks (e.g. "Dynamic Expressions Level 1") and 2) different questions within the question bank (e.g. #005 vs. #006). For this reason, it is recommended that only one output file per category be used to avoid any confusion in the naming of question banks on Moodle.

The answer bank at the end (enclosed by the braces) contains all answer options for the question. The correct answer is preceded by the "=" and all wrong answers are denoted by the "~". When either of these identifiers is used, an answer is taken to be any text until a line break.

Anything between the title and the answer bank is considered to be the prompt, or the actual question being asked. GIFT format also allows HTML tags to be inserted, so usually `<pre>` tags are used to separate code excerpts and for readability.

Since we follow this basic structure for each question, we can abstract the implementation so each category only needs to generate a prompt and the corresponding answers. For this reason, the act of creating a new category of questions takes much less time to implement, and question-writers can be focused on the content of the question, rather than its final format. This also allows questions to have some certain specialized attributes (e.g. a different type of answer options, a true/false question, etc.), but most attributes are fixed or required (such as each question must have a title, a prompt, etc.).

3.1 Difficulty

The most important of these fixed attributes are the static number of difficulties offered. Each question category is limited by four possible difficulties, however it is not necessary that all four difficulties are implemented by each category as long as it is made explicit to the user. Although this fixed number of difficulties

appears to be a shortfall of this implementation (and it is a focus for improvement in the future), these difficulties offer a great deal of flexibility within a category of a question.

The most simple case of difficulty is found in the Expressions category (see Expressions below) that simply serves to test a student's understanding of integer operations and variable assignment in programming. Each difficulty is the same essential type of question, but the number of operands increases with increasing difficulty. Some may consider this a naive approach to difficulty, but it is effective and showcases the fundamental relation between question categories and difficulties.

Other question categories (such as Switch Cases) don't exhibit such a straightforward notion of difficulty, but still an interesting example. Still, each difficulty is essentially the same where students are tested on their ability to follow the control flow of a switch case statement by determining the final value of a variable (see Switch Cases for more detail). The first difficulty randomly generates break statements (for each case) and it may or may not include a default case, where the second difficulty always includes a break statement for each case and automatically includes a default case. Each difficulty of the Switch Case category follows the same pattern by simply modifying random elements of the same core question. While one may seem like more common – but not necessarily always correct – example of a switch case block, it seems difficult to argue that one is inherently more "difficult" than another. However, it is clear that each difficulty of this category would be beneficial to test in its own right, while still being able to test students fairly.

In contrast to these categories that implement the same essential question with slight changes, some categories (such as Functions or Arrays), hold completely different questions in each difficulty. Here, a descriptor like "subcategory" seems more appropriate (since it becomes harder to evaluate question on difficulty in relation to a different question), but to avoid confusion, we will continue to reference these subcategories as difficulties. At first, it also seems like these questions shouldn't be included in the same category, as each may be warranted their own category. However, because each relates to the same category (by testing the same fundamental concept) and any less or extra randomization in the question would not provide anything beneficial to the question, but rather would shift focus from the objective of the question. We won't discuss in detail how a level of randomness affects a question, but there is a certain amount of randomness in a question that no longer serves the purpose of making a question unique, but instead it makes a question unnecessarily difficult to comprehend. In short, these different difficulties are not substantial enough to create their own category and are similar enough to each other to be included in the same category.

Although it now may seem that this concept of difficulty is unclear and unfocused, it is important to note that each of these in Moodle are formed as their own question bank (meaning students are each given questions from the same bank – meaning the same difficulty – and not only the same category – with different difficulties. The instructor then has the option of using whichever

question bank (i.e. difficulty) is most appropriate, and it offers a fair test to each student. Overall, it gives the instructor more choice on how to exactly test his or her students.

3.2 Unique Answer Guarantee

Because of the universal format for questions, extra error checks can be done. While there is no general way to ensure that each question provides the correct answer (see Testing), we can confirm that each answer is unique (and especially that no distractor is the same as the right answer) – a seemingly trivial, but essential, restraint. This can become complicated for two reasons: i) in an effort to create more convincing distractors, each difficulty in each category generates its own distractors using "clever" methods (e.g. evaluating an expression without order of operations, or evaluating an expression based on wrong datatypes) and ii) complex answers (in particular, answers containing vectors; see Arrays) become more difficult to compare and even more difficult to generate an arbitrary new distractor (for the sake of uniqueness) that is still a somewhat plausible answer. In the case of (i), it is fairly simple to compare all answers, and any distractors that are not unique are reassigned a random value close to the answer. This is usually done with computational questions where the answer is simply a numerical value. However, in the case of (ii), if we were to simply alter a value in the vector, the distractor would become completely implausible because the answer only contains values that are originally in the array (see Arrays Difficulty 1). It would make no sense at all to change a value in the array. In order to generate distractors that are still plausible (but are guaranteed to be unique), we can simply shuffle the order of the vector. Yet, this has another precondition that the number of permutations for the correct answer must be greater than or equal to the number of available answers. Without this restraint, there would not be enough unique answers. If this is not met, then it suffices to revert to the original strategy and simply assign a random value to the vector. Although it may not be very convincing, unique distractors take precedence over plausible ones.

4 Testing (is this needed?)

5 Categories

There are basically three types/waves of categories: introductory, loops (intermediate), and advanced. The names of these categories reference not only how they are implemented, but how they are presented to students. As with any course, it is imperative to test student knowledge on material as it is presented to ensure that learning is occurring and prerequisite concepts are understood well enough to proceed. In this section, we'll give a more in-depth explanation of each category, their subcategories, and how some categories are related, based on their order of relative difficulty of content and implementation.

There are four essential components we wish to answer with each of these categories: a) what is the goal of the question; what is it asking? b) an example question (removed from GIFT format) for easy reference, c) what parts of the question are random? which of those parts are essential to the question (i.e. what parts affect the answer or structure of the question)? which of those parts are non-essential (i.e. what parts have no affect over the question whatsoever)? and are there any essential parts of the question that are not randomized (i.e. are there any constant factors of the question)? and lastly d) what methods are used to generate the distractors? It is important to ask (a) so each question can have a definitive objective, and it should be easy to note if certain randomness adds/hinders that objective or if the randomness does not preserve a comparable difficulty for different students. This basically tries to prevent elements making a question "too random". It is also important to ask (c) for a deep understanding of how the question is randomized – and how some parts are kept constant – to produce independent yet similarly difficult questions. This part answers the main methodology of all question generation. And as a consequence, (d) becomes relevant to ask to offer convincing distractors for a cohesive question.

The first wave we will examine is introductory questions.

5.1 Introductory Questions

The categories included in this wave are identifiers, conditional expressions, and expressions. Each of these are topics are presented fairly early, and their main focus includes – explicitly and implicitly – basic ideas such as syntax, variable access, integer operations, logical operations, and operator precedence. Two of these categories (identifiers and conditional expressions) are simply true/false questions, so there are no sophisticated methods of generating distractors, and this is part of the reason these categories are considered simple (from the point of view of implementation). However, these categories are still worth discussing for their testing methods and more of a concrete introduction to the implementation. Let's take a detailed look at each category.

5.1.1 Identifiers

This first category is the simplest of all the questions here. The main objective of the question is to test students' knowledge on identifier syntax in C++. All difficulties are also the same, just with a differing length of string, which only contributes a small amount to the essence of the question, so we'll consider them all at once. Here is an example of the true/false question:

Is the following a valid C++ identifier?
Tb306wN

The reason this question is so simple is because there are not a lot of different elements that attribute to the question; there is only the identifier. Furthermore, as noted before, the true/false nature of the question obviously limits the number

of available answers and doesn't allow for convincing distractors. The entire string of the identifier is randomized. First, the length is selected (between 5 and 9 characters over all difficulties), and then each character is randomly selected from a group (either a numeric character, alpha character, or an invalid character). There is a 1 in 5 chance that the first character is a number (and therefore invalid) and a 1 in 10 chance for every character that it is an invalid character. Consequently, a string of 5 characters has a 45.9% chance of being valid, and a string of length 9 has a 30.1% chance of being valid.

NOTE: should i add something on how this adds to difficulty? I'm not sure how to close out this section

5.1.2 Conditional Expressions

This category is also fairly simple, mostly because it is another true/false question, but it does introduce some of the structure that we will reuse in other categories (mostly the expression structure). The aim for this category is to have students evaluate a conditional expression, which tests logical operations and their precedence. It also implicitly tests variable access, but this concept should be understood by this point, and it does not affect the question greatly. Here is an example:

What is the result of the following logical expression?

```
int numbers = 3;
int marks = 8;
(numbers <= 4) == (marks <= 10)
```

The only important random elements are the operand values (each is between 1 and 10) and all operators. It should first be noted that the structure of the expression is always in the form of (`<sub-expression> <logical operator> (<sub-expression>)`), where each sub-expression is of the form `<variable> <comparison op> <literal>`. Each comparison operator is chosen randomly from (`>`, `>=`, `<`, `<=`) and each logical operator is chosen randomly from (`==`, `!=`, `&&`, `||`). Of course, the names of these operators do not seem apt, but they are sufficient for our conversation.

Each difficulty does not differ greatly from the other. Difficulties 1 and 3 always have 2 sub-expressions, and difficulties 2 and 4 always have 3. And the other difference between difficulties 1 and 2 from 3 and 4 is that the latter two difficulties have a random order of operands in the expression. For example,

```
int documents = 8;
int bar = 10;
(bar <= documents) != (10 < 9)
```

This does not offer any significant difficulty change, but it does alter a format that students may be familiar with, and does a better job of testing the variable accesses.

5.1.3 Expressions

This category is arguably the most important of all, as it serves as a basis for all other, more complex (sophisticated?) categories, and it is even utilized in other other categories (namely Functions). It still follows the basic structure as Identifiers and Conditional expressions (with simple prompts), but it introduces methods used to generate convincing distractors. The main aim is to test students' ability to evaluate simple arithmetic expressions in C++ using integer operands and operations. Similar to Conditional Expressions, it also tests minor variable accessing. Here is an example:

What is the variable "bar" equal to?

```
int foo = 2;  
int bar = foo * 4 % 3 + 7;
```

Again, each difficulty does not change significantly (only the number of operands are increased), but these simple questions are sufficient for one last category for introduction. The main sources of randomness obviously come from the value (between 2 and 10) and number of operands (between 3 and 7 depending on difficulty). It is also important to note that operator precedence is implicitly tested (no parentheses are generated), but there is room to grow and generate interesting parenthetical expressions in the future that may make these types of questions more challenging for students.

The most interesting change in this category from the last is the greater possibility of answers (that is, it is not a simple true/false question) that allow for new distractor methods to be implemented. Compared to later categories, these methods are not very complex, but they focus on common mistakes made by students, which makes for a more difficult question. The main purpose of these distractors (and really of all distractors) is to try and generalize common mistakes for convincing answers, just as an instructor would try to create with a single question.

In this instance, the four distractor methods alter how the expression is evaluated by 1) disregarding order of operations, 2) using float values and operations, 3) evaluating the expression from right to left, and 4) disregarding the order of operations and using the float values (a combination of 1 and 2). These methods, while not particularly novel or sophisticated, aid the main objective of the question in their deliberate misunderstanding of the main concepts. For example, any student that is unfamiliar with integer operations is certain to select the distractor provided by method 2. Of course, in some cases, certain methods would not provide a different answer than the correct one (as integer addition and float addition are similar, etc.), however different answers are still guaranteed by the unique answer checking (see Unique Answer Guarantee above).

5.2 Loops

This wave of questions is usually offered to students next, although – implementation wise – it doesn’t have many similarities to the previous wave other than general format. Instead, loops provides a general template for all of its own categories. Since we may wish to test all types of loops (we will only consider for loops, while loops, and do while loops), we can create a general base for all types of loops and test the same questions in each (as the main difference between each is mostly syntax). We separate each of these loops by category so instructors have the option of choosing exactly what kind of loop they wish to test. Instead of outlining each category as in the previous wave, we will outline two different types of questions (called Loop Counting and Loop Printing) that then create a category for each type of loops (6 categories total). Implementation between categories of the same question type (e.g. For Loops Counting vs. While Loops Counting) have virtually the same implementation (except for an edge case in do while loops as readers can imagine), so we will not distinguish between them and instead only talk of the different question types.

5.2.1 Loop Counting

This first question type of loops is probably similar to questions used in most introductory programming courses, as it is the fundamental concept behind loops (especially for loops). Simply, these questions ask a student how many times a statement within a for loop is executed, based on the common structure of a "counter" variable that is updated each loop, and then compared to a terminating condition: the same structure as a for loop. This is the essence of the question at each difficulty, however, due to the many dependent parts of the question, many different forms of randomization (mostly pertaining to the domain) occur between each difficulty, hence the distinction.

Difficulty 1

This difficulty is the most common form of for loops, and, even to some new students, can be trivial to evaluate. However, it still ensures that the main concepts of loop syntax, terminating conditions, and updating statements. Here is an example from the For Loops category:

```
How many times does the '*' print?  
for (int b = 4; b < 7; b = b + 1)  
{  
    cout << '*';  
}
```

To ensure each question is relatively easy, the comparison operator used is always "<" and the increment operator is always addition. This still leaves many essential details to be randomized. First, the increment is chosen (it has the option of being either 1, 5, or 10 for simplicity), and then based on the increment,

the starting value of the variable is chosen (between 2 and 10 and scaled based on the increment). Finally, the value used for the terminating condition is selected, which is between 3 and 6 greater than the starting value (again, also scaled based on the increment). This last value also has a random value added to it (from 0 to the incremental value minus 1) to make the terminating value seem “less nice”, That is, it is not guaranteed that the variable will be equal to the terminating value when it exits the loop. This guarantees that the loop will execute between 3 and 6 times, without making the actual aspects of the loop seem too formulaic.

Due to the nature of this question, there are not many elaborate distractors, yet they are still effective. Obviously, the first two distractors are ± 1 of the correct answer (the common off-by-one error), and the last two distractors are a random amount away from the original question (± 2 to 4). To eliminate any obvious negative answers, the absolute value is used for each of these distractors.

Difficulty 2

This difficulty is very close to the first, but to increase the effort, more comparison operators ($>$, $>=$, $<$, $<=$) are used for the terminating condition, and the subtraction operator is used in the update statement. Here is an example from the Do While Loops category.

```
How many times does the '*' print?
int j = 53;
do
{
    cout << '*';
    j = j - 5;
} while (j >= 30);
```

It uses the same domain of randomized values, except it swaps the initial value and the terminating value when the operator used to update the variable is “-”. There is also a guarantee to match the correct comparison operators (e.g. $>$, $>=$) with the proper operator in the update assignment (e.g. $+$) unless otherwise specified by the user. If they do allow such operators to be paired with each other, “improper” loops can occur. These are loops that run infinitely (or more precisely, a couple billion times due to integer overflow), and this has a possibility of being the correct answer. When this situation does occur, the correct answer becomes “Around a couple billion”, and in all other cases this is used as a distractor. This option aids in the essence of the question by making it more difficulty, and it also introduces the option for non-homogeneous answer types (as all other available options are integer values). Other than this new option, all distractor methods are the same as Difficulty 1.

Difficulty 3

Using the same exact structure as the previous difficulty, the most significant difference is that the update statements use the multiplication and division

operators as opposed to the addition and subtraction operators. Here is an example from the While Loops category.

How many times does the '*' print?

```
int i = 13;
while (i >= 3)
{
    cout << '*';
    i /= 2;
}
```

As a consequence of using different operators in the update statement, a slightly different domain of random values is used for all parts. The increment is either 2 or 10, and the starting value is chosen from 1 to 4 (and of course scaled by the increment as before). The value used in the terminating condition is calculated by taking the incremental value and raising it to a power between 1 and 3. That value is then multiplied by the starting value to guarantee that the loop executes at least between 1 and 3 times. However, similar to in Difficulty 1, we wish to make values seem less formulaic, so a random value from 0 to the increment less 1 is added to the stopping value to make it appear "less nice". This may also add another iteration in the loop, so there is a final check to ensure the number of iterations is within the a defined amount, and the members are altered if not.

As in Difficulty 2, there is an option for "improper" loops, and the distractor methods are the same. The most interesting new aspect that this difficulty brings is the division operator, which tests students' ability on integer operations when updating variables, as well as for comparison. Overall, this difficulty is comparable to Difficulty 2, except it adds extra difficulty using new operations.

Difficulty 4

While still offering the main objective of all of these difficulties, this difficulty can appear to be fundamentally different. It uses the same generation methods as Difficulty 2, except it uses a nested loop for execution instead of a single loop. Here is a concrete example from the For Loops category.

How many times does the '*' print?

```
for (int i = 43; i >= 20; i = i - 10)
{
    for (int j = 56; j >45; j = j - 5)
    {
        cout << '*';
    }
}
```

Each of these loops is created independently (yet always within the same category), and then evaluated together to create the correct answer. The nested loop structure allows for many more interesting distractors, and there may be some debate on choosing the optimal distractors (as there is a lot of room for

student errors/misunderstandings). For this section we have used four basic distractors where 1) is the number of iterations of the outer loop, 2) is the number of iterations of the outer loop + 1, 3) is the number of iterations of the outer loop + 1 and then multiplied by the number of iterations for the inner loop, and 4) is the number of iterations of the outer loop + 1 multiplied by the number of iterations in the inner loop + 1. It is important to note that we only used the number of iterations of the first two distractors and we did not introduce the inner loop, as some students may recognize that these two values can be multiplied together to easily find the answer, without actually testing any skill.

5.2.2 Loop Printing

This next type of question in the Loops categories contains similarities to Loop Counting. It aims at the same essence, but it tries to test a deeper understanding of loops by printing out the “counter” variable at each iteration. Implicitly, it still tests the number of iterations the loop executes, but the main objective is to understand, explicitly, when a variable is incremented, how integer operations affect the update, and the value that makes the loop terminate. For the sake of questions not being too difficult, each loop only executes between 3 and 5 iterations, and there is no fourth difficulty including nested loops. Here is an example from the Do While Loops category.

What is the following output of the code?

```
int b = 61;
do
{
    cout << b << ' ';
    b = b - 10;
} while (b >20);
```

Each difficulty is generated using the exact same methods as Loop Counting but obviously has different distractor methods, yet the same distractors are used for each difficulty. The first distractor uses all of the same values as the correct answer, except it omits the first value. The second distractor uses the same values as the correct answer, but it appends one more iteration from the loop. The third distractor is the same values as the correct one, except it omits the last value. Finally, the fourth distractor combines the first two methods by omitting the first value and appends an extra iteration to the end. While there are many options for generating distractors, these all seem to provide convincing answers that some students may create on their own.

5.3 Advanced

While this wave is not particularly more difficult than the others, we will refer to them as advanced because they are usually introduced near the end of a course,

and they some categories also utilize other earlier categories. Some of these difficulties also use very involved methods for question generation, evaluation, and distractor generation – as well as the unique answer guarantee for Arrays as discussed earlier.

5.3.1 Switch Cases

The first category in this wave has the same essence for each difficulty; the only significant differences are the random elements, which then determine the difficulty. This is similar to the Loop Counting types described before. For the sake of brevity, here is a description of all shared aspects, and we will later discuss each difficulty’s subtleties. First, a general example:

What is the value of “bar” after the switch case block?

```
int bar = 4;
switch (bar) {
    case 2:
        bar = 5;
        break;
    case 4:
        bar = 6;
    case 1:
        bar = 2;
        break;
    case 3:
        bar = 0;
    case 0:
        bar = 1;
        break;
}
```

As it is easy to see, this category’s main objective is to test students’ knowledge of the control structure through a switch case block. This focuses on the syntax of a switch case statement, how the conditional cases are compared, and how break statements function as opposed to fall through. Occasionally, there may be a default case as well (depending on difficulty).

All distractor methods are also shared between each difficulty, which may be room for improvement, but for now, these methods offer plausible answers for each difficulty. The first distractor is the original value of the variable. The second distractor is always the value that is *compared* to in the first case (e.g. the value 2 in the example above) and is a good distractor to identify students that are not very familiar with the block. The next two distractors (to be used more generally) are based on some conditions.

The third distractor is the value assigned to the variable in the first matching case. In the example above, this would be the value 6 (as the variable starts with value 4). This is a particularly good distractor when there is not a break

statement and fall through occurs. However, if there is no matching case value for the original variable, the third distractor is then the last assigned value *before* the default case, if there is a default case. If there is no default case, the third distractor is simply a random value from those that are assigned within the block (in the case above, this would be one of 5, 6, 2, 0, or 1).

The fourth distractor follows the same conditions as the third, except with different methods at each. First, if there is a matching case to the original variable value, then the fourth distractor is the next compared case value that has a break statement. In the example above, this corresponds to the value 1 (since it is the next compared value with a break statement). Again, this value does not make much sense to one with experience with switch case statements, but it may outline errors for those that are unfamiliar with them. If there is no matching case to the original variable, on the other hand, then the fourth distractor is the compared value in the case statement before the default case (if there is one) or the first case value.

As it can be seen, these methods are somewhat lengthy and complicated, but they offer methods that can be used in each difficulty. However, it can be seen that there are many cases where these distractors are the same value, which we rely upon the unique answer guarantee to solve. It is also important to note that there is a possibility of confusion between the values compared in the case statements, and the values that are assigned after each case statement. This is because, while each assigned value (including the original) is guaranteed to be unique, there is no guarantee that any value compared in a case statement is not the same as one assigned to the variable. This may be improved upon for added clarity in further iterations.

As for the generation of the question, each difficulty shares the domain of nearly all random elements. Namely, the number of case statements (between 3 and 5), all assigned values (between 1 and the number of case statements multiplied by 1.5), and the values compared in the case statement (with the same domain as the assigned values). The only differing randomness occurs in the break statements and default statements.

Difficulty 1

The first difficulty is probably the most common, or most “uniform” switch case statement. There is guaranteed that all cases have a corresponding break statement, and there is always a default case. This is a useful difficulty when just introducing the syntax of switch case statements.

Difficulty 2

Here, there are no break statements, but always a default statement. This is a good test on the concept of fall through in a switch case block.

Difficulty 3

Similar to the last difficulty, this difficulty guarantees that there are no break statements, but that there is no default statement either. This is also useful for

testing fall through, but it is interesting because it does not guarantee that the original value of the variable is changed (in the instance that it matches no case statement), which makes this question slightly more difficulty.

Difficulty 4

This final difficulty is completely random in all aspects. Instead of any guarantees, there is a 50/50 chance for each break statement to be generated, as well as a 50/50 chance for a default case to be present. This is best used when testing all concepts of switch cases including fall through, break statements, and a block with or without a default case. For that reason, this is regarded as the most difficulty question in the category.

5.3.2 Arrays

5.3.3 Functions

6 Future Improvements

7 Conclusion