Erik Boswell

CS 506

5/16/2021

Course Project

For this project I’ve chosen to evaluate a calculator program posted on github.com by user NikeshCS, linked at the end of this document. The rationale to this choice is that a calculator needs to receive a number of inputs and use logical expressions to determine which stored variables are being used, making it ideal for ISP testing, Graph testing, and Logic testing. Additionally, a calculator is essentially interpreting a mathematical language that can be developed as a BNF Grammar for syntactical evaluation. I would note that this calculator is incomplete and while including buttons for operations like square roots and exponents, these functions were not incorporated. Each section will detail the process and results behind developing each test type to apply to this program. This project will focus on the CalculatorMainFrame.java specifically where all the functions of the calculator are contained. The other files designate the button layouts, graphical design, and visual readouts of the calculator program rather than the actual functional performance. Additionally, the CalculatorMainFrameTest.java file contains all tests mentioned in this paper.

ISP

Input Space Partition testing focuses on testing inputs of methods. For this case we will focus on the equateOperations method. These inputs form a domain that is partitioned in testing in order to test all values separately. To start the process of this test design we will identify the input and state variables of this program. These variables are

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Table A: Identify input space and determine characteristics** | | | | | | | |
| **Method** | **Parameters** | **Returns** | **Values** | **Exception** | **Char ID** | **Characteristic** | **Covered by** |
| equateOperations() | val1 | long | long |  | C1 | Has a value |  |
|  | val2 | long | long |  | C2 | Has a value |  |
|  | var | long | string |  | C3 | Is an operator for string |  |
|  |  |  |  |  |  |  |  |

Now that we’ve established our input variables and there are no state variables, we define characteristics based on these inputs. We assign the following possible characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| **Table B: Design Partitioning** | | | |
| **Char ID** | **Characteristic** | **f1:equateOperations()** | **Partition** |
| C1 | Has a value | C1 | true/false |
| C2 | Has a value | C2 | true/false |
| C3 | String is an operator | C3 | true(is equal to “+”, “-”, “\*”, “/”, “%”)/false(else) |
|  |  |  |  |

With these characteristics for each input defined, we can now create a table detailing each possible input and create tests according to outcomes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table C: Define test requirements for Base Choice Criteria (BCC) (make sure to indicate your base case)** | | | | | |
| **Method** | **Characteristics** | **Test Requirements** | **Infeasible TRs** | **Revised TRs** | **# TRs** |
| equateOperations() | C1C2C3 | **TTT(base), FTT, TFT, TTF** |  |  | 4 |
|  |  |  |  |  |  |

Following this process, we can use our chosen test requirements as tests. In this case, we let one characteristic assume each possible characteristic while all other characteristics are locked to the designated base value. While these criteria only satisfy Base Choice Coverage, we could test any possible combination depending on what test type we use. In the case of any possibly infeasible tests, we would revise the TR to be a valid TR while retaining that our chosen characteristic is still on the value we had designated previously. The following table shows the collection of tests for Base Choice Coverage.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case design** | | | |
| **Method** | **Case** | **Set up data** | **Steps** |
| equateOperations() | **TTT** | CalculatorMainFrame calc = new CalculatorMainFrame | calc.equateOperations(1,1,”+”) |
|  |  |  |  |
|  | FTT | CalculatorMainFrame calc = new CalculatorMainFrame | calc.equateOperations(null,1,”+”) |
|  |  |  |  |
|  | **TFT** | CalculatorMainFrame calc = new CalculatorMainFrame | calc.equateOperations(1,null,”+”) |
|  |  |  |  |
|  | TTF | CalculatorMainFrame calc = new CalculatorMainFrame | calc.equateOperations(1,1,””) |
|  |  |  |  |

Graph Coverage

With this implementation of the Base Choice Coverage, we have displayed a method of Input Space Partition testing. The next example we can continue onto is Graph based testing. With Graph based testing, we establish a control flow graph of the program. Using this graph, we designate nodes where certain portions of code are being executed and then branch out to different nodes based on certain conditions in the code. Typically if statements can split the branches of the graph or instances like for or while loops can create loops in the graph. The graph for this section is attached in the portfolio as Graph.jpg since it is too big to fit in the document. It depicts a graph of the actionPerformed method of this program, since this method is where a lot of the calculator actually functions, receiving inputs and calling other methods. The file for this graph is attached in the portfolio and can be opened at <https://diagrams.freebusinessapps.net/diagrams>. This graph has been developed with a lot of exit nodes. Each if statement provides for a specific input, so if one of the if statements is passed, then the result is executed, and then no other if statement will be executed given the exclusivity of each statement. The graph gets most interesting at the cases of ‘=’ input, backspace, the ‘±’ input and when changing number bases, and has a more complex result in each case. Following this, we can list nodes as the respective points along the graph in the a table at the end of the document, detailing the node, it’s type, and what code portions the node runs when reached. Another table at the end of the document can be referenced to list all prime paths of this graph. A simple path is any path in the graph, but a prime path is a path that is not subset to any other path. The intuition in this case for prime paths is simple since we need only take every possible path from our only entry node to each exit node. This could get a little more complex if there was a loop anywhere in this graph. With all these listings we can now develop a Prime Path Coverage test set. This takes all the Prime Paths as test requirements and tests all start to finish possibilities in the program. It is important to note that this graph tests everything all of our button presses can accomplish, a lot of which is two line tests. As such, the basic buttons presses grouped by buttons 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, +, -, \*, /, and % are functionally the same test, checking that the button updates a value when pressed. What’s far more interesting among these tests is examples like the equals button, since the equals button follows several checks to determine all the possible outputs it could generate given the inputs that are already stored on the calculator. Using the methods of logic coverage, our RACC for the = button will actually cover the prime path coverage examples for the = button which is the strongest demonstration of path coverage within our graph. As such, our RACC tests from the next section will be labelled with the corresponding path it covers in our code as well to show this. This will be the only display of the Graph Coverage given the sheer volume of tests necessary to cover our whole graph, but each prime path is included in the prime path table at the end, along with pseudocode for each prime path’s test.

Logic Coverage

The next testing area to cover is logic testing. Throughout programs, one can find the use of various logical statements to determine what the program does. These logical statements can be identified as predicates with variables as their clauses which evaluate to boolean values. We can use these constructions of logical statements in the program to then design a table detailing the possible logical outcomes and test based on these possible outcomes and also when certain clauses determine the outcome of the predicate. Looking at the following tables we can see a display of all the predicates with their clauses and outcomes for the respective methods.

|  |  |  |
| --- | --- | --- |
|  | else if(e.getSource() == ButtonsPanel.equalButton) from actionPerformed method |  |
| Predicate | Code Fragment | Clause Representation |
| p1 | if(ButtonsPanel.resultField.getText().equals("Can't divide by zero") || ButtonsPanel.resultField.getText().equals("Error")) | c1 || c2 |
| p2 | else if(!operator.isEmpty() && !varPushed) | c3 && c4 |
| p3 | if(!operator.isEmpty()) | c3 |
| p4 | if(!operator.equals("/") && rightValue == 0) | c5 && c6 |
| p5 | else if(operator.equals("%") && rightValue == 0) | c7 && c6 |
| p6 | if(BaseGroupPanel.decimalButton.isSelected()) | c8 |
| p7 | else if(BaseGroupPanel.binaryButton.isSelected()) | c9 |
| p8 | else if(BaseGroupPanel.octalButton.isSelected()) | c10 |
| p9 | else if(BaseGroupPanel.hexadecimalButton.isSelected()) | c11 |

|  |  |
| --- | --- |
| Clause List |  |
| c1 | ButtonsPanel.resultField.getText().equals("Can't divide by zero") |
| c2 | ButtonsPanel.resultField.getText().equals("Error") |
| c3 | !operator.isEmpty() |
| c4 | !varPushed |
| c5 | !operator.equals("/") |
| c6 | rightValue == 0 |
| c7 | operator.equals("%") |
| c8 | BaseGroupPanel.decimalButton.isSelected() |
| c9 | BaseGroupPanel.binaryButton.isSelected() |
| c10 | BaseGroupPanel.octalButton.isSelected() |
| c11 | BaseGroupPanel.hexadecimalButton.isSelected() |

|  |  |  |
| --- | --- | --- |
|  | public void numberChecking() |  |
| p1 | if(varSolved) | c1 |
| p2 | else if (!operator.isEmpty() && !varSaved.isEmpty() && !varSaved.equals(" ")) | c2 && c3 && c4 |
| p3 | if(ButtonsPanel.resultField.getText().equals("0")) | c5 |
| p4 | if(ButtonsPanel.resultField.getText().equals("Error") || ButtonsPanel.resultField.getText().equals("Can't divide by zero")) | c6 || c7 |

|  |  |
| --- | --- |
| Clause List |  |
| c1 | varSolved |
| c2 | !operator.isEmpty() |
| c3 | !varSaved.isEmpty() |
| c4 | !varSaved.equals(" ") |
| c5 | ButtonsPanel.resultField.getText().equals("0") |
| c6 | ButtonsPanel.resultField.getText().equals("Error") |
| c7 | ButtonsPanel.resultField.getText().equals("Can't divide by zero") |

From this, we develop a Restricted Active Clause Coverage test set. This test is developed by taking a pair of cases where a major clause is true in one case and false in the other and determines the predicate in each case. Additionally, all other clauses must remain the same in each test case in the pair. While this test only needs one pair to be performed, there can be multiple pairs that satisfy the test requirements. In the following truth tables, we can see how some of the tests are determined and each RACC that needs to be developed. Do take note that test feasibility always needs to be in mind, since in numberChecking method p2, test rows 3 and 7 are infeasible due to a contradiction between c3 and c4.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| else if(e.getSource() == ButtonsPanel.equalButton) | | | | | |  | RACC Tests | c1 Major | c2 Major |
|  | c1 | c2 | p1 | p\_c1 | p\_c2 |  |  | (2,4) | (3,4) |
| 1 | T | T | T | F | F |  |  |  |  |
| 2 | T | F | T | T | F |  |  |  |  |
| 3 | F | T | T | F | T |  |  |  |  |
| 4 | F | F | F | T | T |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| else if(e.getSource() == ButtonsPanel.equalButton), same table for p4 and p5 | | | | | |  | RACC Tests | c3 Major | c4 Major |
|  | c3 | c4 | p2 | p\_c3 | p\_c4 |  |  | (1,3) | (1,2) |
| 1 | T | T | T | T | T |  |  |  |  |
| 2 | T | F | F | F | T |  |  |  |  |
| 3 | F | T | F | T | F |  |  |  |  |
| 4 | F | F | F | F | F |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| else if(e.getSource() == ButtonsPanel.equalButton), same table for p6, p7, p8, and p9 | | | |  | RACC Tests | C3 Major |
|  | c3 | p3 | p\_c3 |  |  | (1,2) |
| 1 | T | T | T |  |  |  |
| 2 | F | F | T |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| public void numberChecking() | | | |  | RACC Tests | c1 Major |
|  | c1 | p1 | p\_c1 |  |  | (1,2) |
| 1 | T | T | T |  |  |  |
| 2 | F | F | T |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public void numberChecking() | | | | | | | |  | RACC Tests | c2 Major | c3 Major | c4 Major |
|  | c2 | c3 | c4 | p2 | p\_c2 | p\_c3 | p\_c4 |  |  | (1,5) | (1,3) | (1,2) |
| 1 | T | T | T | T | T | T | T |  |  |  |  |  |
| 2 | T | T | F | F | F | F | T |  |  |  |  |  |
| 3 | T | F | T | F | F | T | F |  |  |  |  |  |
| 4 | T | F | F | F | F | F | F |  |  |  |  |  |
| 5 | F | T | T | F | T | F | F |  |  |  |  |  |
| 6 | F | T | F | F | F | F | F |  |  |  |  |  |
| 7 | F | F | T | F | F | F | F |  |  |  |  |  |
| 8 | F | F | F | F | F | F | F |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| public void numberChecking() | | | | | |  | RACC Tests | c6 Major | c7 Major |
|  | c6 | c7 | p4 | p\_c6 | p\_c7 |  |  | (2,4) | (3,4) |
| 1 | T | T | T | F | F |  |  |  |  |
| 2 | T | F | T | T | F |  |  |  |  |
| 3 | F | T | T | F | T |  |  |  |  |
| 4 | F | F | F | T | T |  |  |  |  |

Finally we write our tests based on these RACC test requirements in the program to observe the outcomes. All of these tables are from the attached spreadsheet on sheets 3 and 4, and sheet 4 also includes the initial drafts of all RACC testing. I would say it’s also noteworthy the overlap that can be achieved in test requirements in Logic Coverage and Graph Coverage, especially in the case of RACC and Prime Path Coverage. This is especially the case in this program due toe the nature of these if statements working like a switchboard.

Syntax Coverage

The final test type we move to is Syntax based testing. For production testing, we will define the language for the calculator so it’s validity can be tested on the equateOperations method. To define the expressions in BNF we set the following grammar

expr ::= num | expr op expr  
num ::= digit | digit num  
op ::= “+” | “-” | “\*” | “/” | “%”

digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”  
  
If we wanted to define these for binary, octal or decimal, we could adjust digits accordingly to have more or less terminals, and remove dec since there won’t be decimal results in those number bases. To establish a production test for this grammar, we need to test every terminal and production rule. According to the grammar, this will take the five following tests since we need individual tests for each operator.  
  
 1. equateOperations(32, 6, “+”), 32 + 6

2. equateOperations(107, 94, “-”) 107 - 94

3. equateOperations(125, 804, “\*”) 125 \* 804

4. equateOperations(21, 7, “/”) 21 / 7

5. equateOperations(15, 6, “%”) 15 % 6

Plugging in these tests on the program, we can see the results in our output. Then there’s mutant testing.

For Mutant testing, we want to make mutants for this example of grammar and show that they’re killed. A mutant is strongly killed when a test requirement shows that the test results in a failure. There are plenty of ways to create valid mutants that still fit our definitions of being a collection numbers, followed by an operator and then more numbers. If we want to violate this can create a killable mutant on this grammar, we can put a number in our operator. Let our statement for testing be

equateOperations(18, 6, “6”)

Where we have mutated the statement 18 + 6. A string has been provided, but as an invalid operator, we get 0 as an output and the statement fails since we should still expect a result of 24.. So this mutant would be strongly killed by this test requirement.

Graph Node Table

|  |  |  |
| --- | --- | --- |
| Node | Node Type | Lines and Fragments Executed |
| 1 | Entry | 292,293 |
| 2 | Standard | 295,296 |
| 3 | Standard | 298,299 |
| 4 | Standard | 301,303 |
| 5 | Exit | 305 |
| 6 | Standard | 307 |
| 7 | Exit | 309-311 |
| 8 | Standard | 313 |
| 9 | Exit | 315-317 |
| 10 | Standard | 319 |
| 11 | Exit | 321-323 |
| 12 | Standard | 325 |
| 13 | Exit | 327-329 |
| 14 | Standard | 331 |
| 15 | Exit | 333-335 |
| 16 | Standard | 337 |
| 17 | Exit | 339-341 |
| 18 | Standard | 343 |
| 19 | Exit | 345-347 |
| 20 | Standard | 349 |
| 21 | Exit | 351-353 |
| 22 | Standard | 355 |
| 23 | Exit | 357-359 |
| 24 | Standard | 361 |
| 25 | Exit | 363-365 |
| 26 | Standard | 367 |
| 27 | Exit | 369-371 |
| 28 | Standard | 373 |
| 29 | Exit | 375-377 |
| 30 | Standard | 379 |
| 31 | Exit | 381-383 |
| 32 | Standard | 385 |
| 33 | Exit | 387-389 |
| 34 | Standard | 391 |
| 35 | Exit | 393-395 |
| 36 | Standard | 397 |
| 37 | Exit | 399-401 |
| 38 | Standard | 403 |
| 39 | Exit | 405-407 |
| 40 | Standard | 409 |
| 41 | Exit | 411-413 |
| 42 | Standard | 415 |
| 43 | Exit | 417-419 |
| 44 | Standard | 421 |
| 45 | Standard | 423 |
| 46 | Exit | 425 |
| 47 | Standard | 427 |
| 48 | Exit | 429 |
| 49 | Standard | 431,433 |
| 50 | Standard | 435 |
| 51 | Standard | 437,438 |
| 52 | Standard | 441 |
| 53 | Standard | 443,444 |
| 54 | Standard | 446 |
| 55 | Standard | 448 |
| 56 | Standard | 450 |
| 57 | Standard | 452 |
| 58 | Standard | 454 |
| 59 | Standard | 456 |
| 60 | Standard | 458 |
| 61 | Standard | 460 |
| 62 | Standard | 462 |
| 63 | Standard | 463,464 |
| 64 | Exit | 469-471 |
| 65 | Standard | 474 |
| 66 | Exit | 476-483 |
| 67 | Standard | 485 |
| 68 | Exit | 487,488 |
| 69 | Standard | 490 |
| 70 | Standard | 492,493 |
| 71 | Exit | 495,496 |
| 72 | Standard | 498 |
| 73 | Exit | 500,501 |
| 74 | Standard | 503 |
| 75 | Standard | 505,506 |
| 76 | Standard | 508 |
| 77 | Standard | 510,511 |
| 78 | Exit | 513 |
| 79 | Standard | 516 |
| 80 | Standard | 518 |
| 81 | Exit | 520 |
| 82 | Standard | 522,524,525 |
| 83 | Standard | 527,528 |
| 84 | Standard | 530 |
| 85 | Standard | 532,533 |
| 86 | Standard | 535 |
| 87 | Standard | 537,538 |
| 88 | Standard | 540 |
| 89 | Standard | 542,543 |
| 90 | Exit | 545 |
| 91 | Standard | 547 |
| 92 | Exit | 549-551 |
| 93 | Standard | 553 |
| 94 | Standard | 555 |
| 95 | Standard | 557 |
| 96 | Standard | 561 |
| 97 | Standard | 563,564 |
| 98 | Standard | 566 |
| 99 | Standard | 568,569 |
| 100 | Standard | 571 |
| 101 | Standard | 573,574 |
| 102 | Exit | 576-593 |
| 103 | Standard | 596 |
| 104 | Standard | 598 |
| 105 | Standard | 600,601 |
| 106 | Standard | 603 |
| 107 | Standard | 607 |
| 108 | Standard | 609,610 |
| 109 | Standard | 612 |
| 110 | Standard | 614,615 |
| 111 | Exit | 617-634 |
| 112 | Standard | 636 |
| 113 | Standard | 638 |
| 114 | Standard | 640,641 |
| 115 | Standard | 643 |
| 116 | Standard | 645,646 |
| 117 | Standard | 648 |
| 118 | Standard | 652 |
| 119 | Standard | 654,655 |
| 120 | Exit | 657-675 |
| 121 | Standard | 677 |
| 122 | Standard | 679 |
| 123 | Standard | 681,682 |
| 124 | Standard | 684 |
| 125 | Standard | 686,687 |
| 126 | Standard | 689 |
| 127 | Standard | 691,692 |
| 128 | Standard | 694 |
| 129 | Exit | 698-716 |

List of Prime Paths

|  |
| --- |
| 1,2,4,5 |
| 1,2,3,5 |
| 1,6,7 |
| 1,6,8,9 |
| 1,6,8,10,11 |
| 1,6,8,10,12,13 |
| 1,6,8,10,12,14,15 |
| 1,6,8,10,12,14,16,17 |
| 1,6,8,10,12,14,16,18,19 |
| 1,6,8,10,12,14,16,18,20,21 |
| 1,6,8,10,12,14,16,18,20,22,23 |
| 1,6,8,10,12,14,16,18,20,22,24,25 |
| 1,6,8,10,12,14,16,18,20,22,24,26,27 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,29 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,31 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,33 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,35 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,37 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,39 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,41 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,43 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,46 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,48 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,51,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,52,53,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,52,54,55,56,63,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,52,54,57,58,63,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,52,54,59,60,63,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,45,47,49,50,52,54,61,62,63,64 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,66 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,68 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,70,71 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,70,72,73 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,70,72,74,75,78 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,70,72,74,76,77,78 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,80,81 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,80,82,83,90 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,80,82,84,85,90 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,80,82,84,86,87,90 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,80,82,84,86,88,89,90 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,92 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,95,102 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,95,96,97,102 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,95,96,98,99,102 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,95,96,98,100,101,  102 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,104,105,111 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,104,106,111 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,104,106,107,  108,111 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,104,106,107,  109,110,111 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,113,114,  120 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,113,115,  116,120 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,113,115,  117,120 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,113,115,  117,118,119,120 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,121,122,  123,129 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,121,122,  124,125,129 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,121,122,  124,126,127,129 |
| 1,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,65,67,69,79,91,93,94,103,112,121,122,  124,126,128,129 |

Github Link for Program: https://github.com/NikeshCS/Calculator