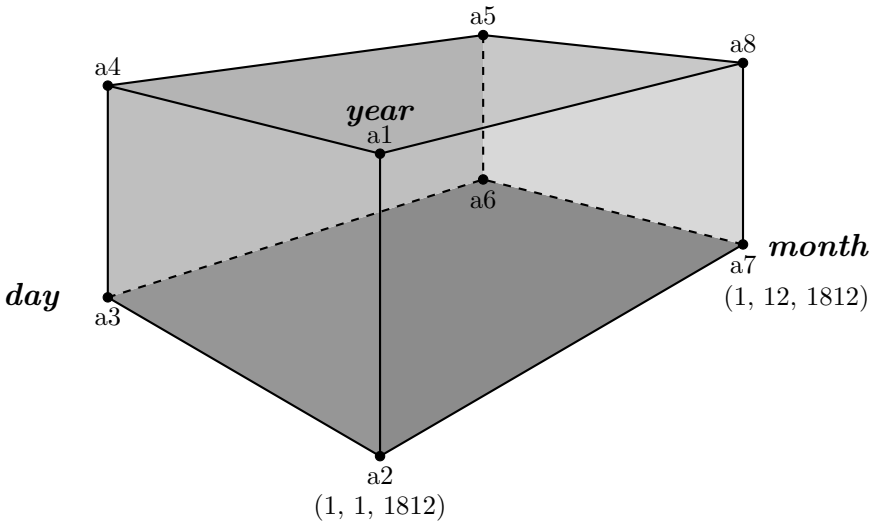
Answer all questions in the space provided here.

Question	Points	Score
Question 1	16	
Question 2	15	
Question 3	13	
Question 4	6	
Total	50	

Question 1 ..... **16 marks**

In the lecture we discussed the testing of a **NextDate** method, which, given a day, a month, and a year, returns the date of the following day. Assume the year variable ranges over [1812, 2016].

Since there are three variables (day, month, and year), we can visualise the boundaries using a 3D plot, as follows.



Each of the 3 axes represents day, month, and year, respectively, and the internal of the 3D shape represents the valid values. Each of the end points in the rectangular cube, labelled a1, a2 to a8, represents a specific tuple of boundary values for the three variables day, month, and year. Point a2 represents the allowed min values for day, month, as well as year, i.e., (1, 1, 1812). Point a7 represents the allowed min values for day and year, but the allowed max value for month, i.e., (1, 12, 1812).

Given the above visualisation, we can reason about test cases for **strong**, **normal** boundary value testing (BVT) on points, lines, planes, and the cube itself. An example, for points, is given below.

Points. For each point (denoted a1 to a8), there are 8 test cases for the 3 variables. For example, for points a2 and a7, we have the following test cases at and around the min values of the 3 variables:

(a) Test cases for point a2.

day	month	year
1	1	1812
1	1	1813
1	2	1812
1	2	1813
2	1	1812
2	1	1813
2	2	1812
2	2	1813

(b) Test casse for point a7.

day	month	year
1	11	1812
1	11	1813
1	12	1812
1	12	1813
2	11	1812
2	11	1813
2	12	1812
2	12	1813

Hence, for all 8 points, we need 8*8=64 test cases.

Now we'd like to extend the method to **NextHour**, with an additional variable, *hour*, that represents the 24 hours of a day (ranging between 0 and 23). Given an hour, a day, a month, and a year, **NextHour** returns the hour as well as the date of the following hour. The 3D cube now becomes a 4D *tesseract*. A tesseract is a four-dimensional analog of a cube, with 16 points, 32 lines, 24 planes, 8 cubes, and (of course) the tesseract itself.

For **NextHour**, please complete the following tasks.

(a) (14 marks) Identify details of test cases for **strong, normal** BVT testing. Specifically,

1. Give details of test cases for (1) points, (2) (the mid point of) lines, (3) (the centre of) planes, (4) (the centre of) cubes, and (5) the (4D) tesseract.
2. Calculate the total number of test cases.

— blank page for answers if required. Will be marked. —
— Indicate clearly question number. —

(b) (2 marks) Generalising from the previous part, what is the number of total test cases for strong, normal BVT for n variables?

Question 2 ..... **15 marks**

The A* algorithm is a widely-used, heuristics-based graph search algorithm. It finds a path from a given *start* node to a given *goal* node. A* uses a distance-plus-cost heuristic function $f(x) = g(x) + h(x)$ to determine the order of traversal. For the current node x, $g(x)$ is the cost from the *start* node to x, and $h(x)$ is an estimate of the distance to the *goal* from x.

Algorithm 0: The A* algorithm.

Input: *start* node
Input: *goal* node
Output: a *path* from *start* to *goal*

```
1  closedset ← ∅
2  openset ← {start}
3  came_from ← ∅                                              // the path to goal
4  foreach node in the graph do                               // set initial values
5  |  g(node) ← ∞ f(node), h(node) ← ∞
6  end
7  g(start) ← 0                                  // Cost from start along best known path
8  f(start), h(start) ← estimate_cost(start, goal)            // Estimate cost to goal
9  while openset ≠ ∅ do
10 |  current ← node, where node has the smallest f value among all nodes in openset
11 |  if current = goal then
12 |  |  return reconstruct_path(came_from, goal)             // Found the path
13 |  end
14 |  openset ← openset \ {current}
15 |  closedset ← closedset ∪ {current}
16 |  for neighbour ∈ neighbour_nodes(current) do
17 |  |  if neighbour ∈ closedset then
18 |  |  |  continue                                         // Already evaluated, ignore
19 |  |  end
20 |  |  if neighbour ∉ openset then
21 |  |  |  openset ← openset ∪ {neighbour}
22 |  |  end
23 |  |  temp_g_score ← g(current) + dist(current, neighbour)   // Distance from current to
   |  |    neighbour
24 |  |  if temp_g_score ≥ g(neighbour) then
25 |  |  |  continue          // No good, skipping to evaluate the next neighbour
26 |  |  end
27 |  |  came_from ← came_from ∪ (neighbour, current)
28 |  |  g(neighbour) ← temp_g_score
29 |  |  h(neighbour) ← estimate_cost(neighbour, goal)
30 |  |  f(neighbour) ← g(neighbour) + h(neighbour)
31 |  end
32 end
33 return failure                                             // Didn't find a path
```

Note that *reconstruct_path*, *neighbour_nodes*, *dist* and *estimate_cost* are functions defined elsewhere.

(Continued overleaf)

(a) (5 marks) Draw the program graph for the above algorithm, and calculate its cyclomatic complexity.

(b) (3 marks) Recall the concept of *structured programming constructs*. For the algorithm above, identify **all** *violations* of structured programming constructs.

(c) (7 marks) Propose changes to the algorithm to make it free of *violations* of structured programming constructs, and draw the resulting *condensed* program graph.

Listing 1 below shows, on two pages, the class `Foo` in Java. Answer **Question 3** and **Question 4** about the class `Foo` and its tests.

Listing 1: The Java class `Foo`.

```
1  public class Foo {
2
3      private int min;
4      private int max;
5
6      public String fizzBuzz(String input) {
7          int x = Integer.parseInt(input);
8
9          String result = input;
10
11         if (x % 3 == 0 && x % 5 == 0)
12             result = "FizzBuzz";
13         if (x % 3 == 0)
14             result = "Fizz";
15         else if (x % 5 == 0)
16             result = "Buzz";
17
18         return result;
19     }
20
21     public String[] fizzBuzzRange(int low, int high) {
22         if (low <= 1)
23             throw new IllegalArgumentException("low should be >= 1");
24         else if (high > 100)
25             throw new IllegalArgumentException("high should be <= 100");
26
27         String[] result = new String[high - low + 1];
28         for (int i = low; i <= high; i++)
29             result[i - low] = fizzBuzz(Integer.toString(i));
30
31         return result;
32     }
33 }
```

(Continued overleaf)

Question 3 ..... **13 marks**

"Fizz Buzz" has been used as a simple interview question for software developers. In its simplest form, the program takes as input an integer value between 1 and 100 (both inclusive), and prints the number itself when it is not divisible by either three or five. For numbers which are multiples of both three and five the program should print "`FizzBuzz`" instead. Otherwise, for multiples of three the program should print "`Fizz`" instead of the number, for multiples of five the program should print "`Buzz`". Methods `fizzBuzz` and `fizzBuzzRange` in code listing 1 above are a simple implementation in Java.

The following test suite in Listing 2 has been developed for the `fizzBuzz` and `fizzBuzzRange` methods. Answer the following questions about the test suite.

Listing 2: A test suite for the `fizzBuzz()` and `fizzBuzzRange` methods.

```
@Test
public void testFizzBuzzRange() {
    System.out.println(Arrays.toString(new Foo().fizzBuzzRange(1, 100)));
}

@Test
public void wrongNumbersAreNotProcessed() {
    try {
        new Foo().fizzBuzz("0");
    } catch (Exception e) {
        String message = e.getMessage();
        assertTrue("Contains correct message", message.contains(">= 1"));
    }
}

@Test(expected = Exception.class)
public void illegalInputThrowsException() {
    new Foo().fizzBuzz(" ");
    new Foo().fizzBuzz(" a");
    new Foo().fizzBuzz("   ");
}

@Test
public void threeGetsFizzAndFiveGetsBuzz() {
    try {
        assertEquals("Should return Fizz", "Fizz", new Foo().fizzBuzz("3"));
        assertEquals("Should return Fizz", "Fizz", new Foo().fizzBuzz("6 "));
        assertEquals("Should return Buzz", "buzz", new Foo().fizzBuzz("5"));
        assertEquals("Should return Buzz", "buzz", new Foo().fizzBuzz("15"));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

(a) (1 mark) What is the statement coverage of this test suite for these two methods (`fizzBuzz` and `fizzBuzzRange`)?

Note that the lines you need to consider include those lines in the body of the two methods, excluding empty lines. In other words, the total number of lines to cover is 17.

(b) (8 marks) There are some problems (errors or deficiencies) with some of these test cases. (1) List these problems, and (2) discuss how they can be fixed.

(c) (4 marks) Devise a test suite with the smallest possible number of test cases that achieves 100% *branch coverage* for methods `fizzBuzz`, and argue why it is the smallest test suite.

Question 4 ..... **6 marks**

Mutation testing is a technique to assess the efficacy and quality of a test suite. It works by making *mutants*, syntactic variations of the program under test, and measuring how many of the mutants are *killed* by the test suite. The presence of non-equivalent *live* mutants represents inadequacy of the test suite.

Come up with three *non-equivalent*, *first-order* mutants of the method `fizzBuzz`. Each mutant should use one of the following mutation operators. Determine the *kill rate* of the test suite from the last question on the three mutants.

The mutation operators you can use are:

`aor:` Arithmetic operator replacement.

`ror:` Relational operator replacement.

`sdl:` Statement deletion.

`uoi:` Unary operator insertion.

`svr:` Scalar variable replacement.

`vie:` Scalar variable initialisation elimination.

— Additional page for answers if required. Will be marked. —
— Indicate clearly question number. —

— Additional page for answers if required. Will be marked. —
— Indicate clearly question number. —

—— End of the paper ——