C Adikesava

Test ID: 450015228556817 | 9347756802 | adikesava23@lpu.in

Test Date: November 29, 2025

Computer Science	Logical Ability	Computer Programming	Quantitative Ability (Advanced)
34 /100	61 /100	49 /100	51 /100

English Comprehension	Automata Fix	Automata Pro	Personality
48 /100	0 /100	4 /100	Completed

Computer Science — 34 / 100

OS and Computer Architecture	DBMS	Computer Networks
44 / 100	0 / 100	49 / 100

Logical Ability — 61 / 100

Inductive Reasoning	Deductive Reasoning	Abductive Reasoning
64 / 100	60 / 100	59 / 100

Computer Programming — 49 / 100

Basic Programming	Data Structures	OOP and Complexity Theory
51 / 100	46 / 100	51 / 100

Quantitative Ability (Advanced)

51 / 100

Basic Mathematics	Advanced Mathematics	Applied Mathematics
60 / 100	46 / 100	48 / 100

English Comprehension

48 / 100 | CEFR: **B1**

Grammar	Vocabulary	Comprehension
45 / 100	51 / 100	49 / 100

Automata Fix

0 / 100

Code Reuse	Logical Error	Syntactical Error
0 / 100	0 / 100	0 / 100

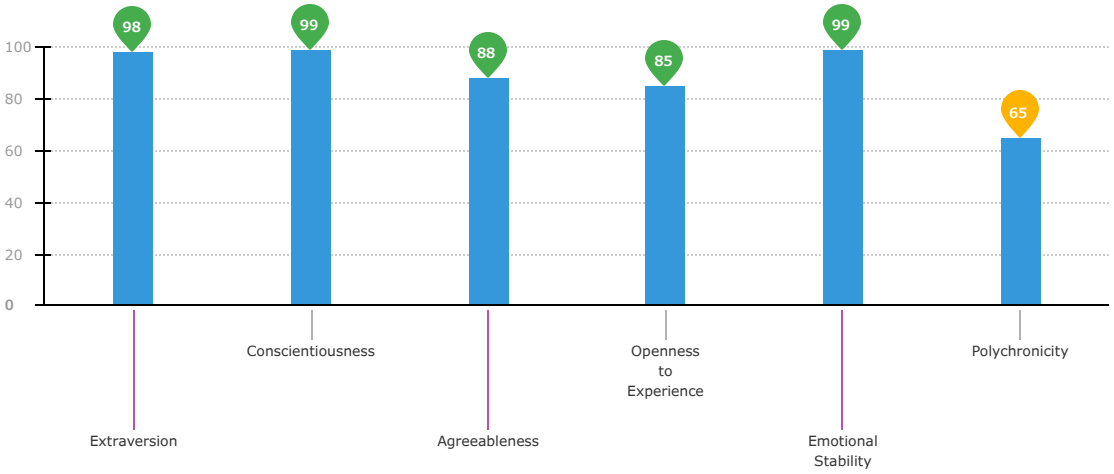
Automata Pro

4 / 100

Programming Ability	Programming Practices	Functional Correctness
10 / 100	0 / 100	10 / 100

Personality

Completed



Competencies

People Interaction

Self-Drive

Trainability

Repetitive Job Suitability

Work attributes

1 | Introduction

About the Report

This report provides a detailed analysis of the candidate's performance on different assessments. The tests for this job role were decided based on job analysis, O*Net taxonomy mapping and/or criterion validity studies. The candidate’s responses to these tests help construct a profile that reflects her/his likely performance level and achievement potential in the job role

This report has the following sections:

The **Summary** section provides an overall snapshot of the candidate’s performance. It includes a graphical representation of the test scores and the subsection scores.

The **Insights** section provides detailed feedback on the candidate’s performance in each of the tests. The descriptive feedback includes the competency definitions, the topics covered in the test, and a note on the level of the candidate’s performance.

The **Response** section captures the response provided by the candidate. This section includes only those tests that require a subjective input from the candidate and are scored based on artificial intelligence and machine learning.

The **Learning Resources** section provides online and offline resources to improve the candidate's knowledge, abilities, and skills in the different areas on which s/he was evaluated.

Score Interpretation

All the test scores are on a scale of 0-100. All the tests except personality and behavioural evaluation provide absolute scores. The personality and behavioural tests provide a norm-referenced score and hence, are percentile scores. Throughout the report, the colour codes used are as follows:

- 70 ≤ Score < 100
- 30 ≤ Score < 70
- 0 ≤ Score < 30

2 | Insights

English Comprehension

48 / 100 | CEFR: **B1**

This test aims to measure your vocabulary, grammar and reading comprehension skills.

You are able to construct short sentences and understand simple text. The ability to read and comprehend is important for most jobs. However, it is of utmost importance for jobs that involve research, content development, editing, teaching, etc.

Logical Ability

61 / 100

Inductive Reasoning

64 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

It is commendable that you have excellent inductive reasoning skills. You are able to make specific observations to generalize situations and also formulate new generic rules from variable data.

Deductive Reasoning

60 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

You are able to work out rules based on specific information and solve general work problems using these rules. This skill is required in data-driven research jobs where one needs to formulate new rules based on variable trends.

Abductive Reasoning

59 / 100

Quantitative Ability (Advanced)

51 / 100

This test aims to measure your ability to solve problems on basic arithmetic operations, probability, permutations and combinations, and other advanced concepts.

You are good at basic arithmetic. You are able to solve real-world problems that involve simple addition, subtraction, multiplication and division.

Personality

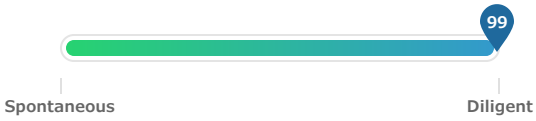
Competencies

Extraversion

Extraversion refers to a person's inclination to prefer social interaction over spending time alone. Individuals with high levels of extraversion are perceived to be outgoing, warm and socially confident.

- You are outgoing and seek out opportunities to meet new people.
- You tend to enjoy social gatherings and feels comfortable amongst strangers and friends equally.
- You display high energy levels and like to indulge in thrilling and exciting activities.
- You may tend to be assertive about your opinions and prefer action over contemplation.
- You take initiative and are more inclined to take charge than to wait for others to lead the way.
- Your personality is well suited for jobs demanding frequent interaction with people.

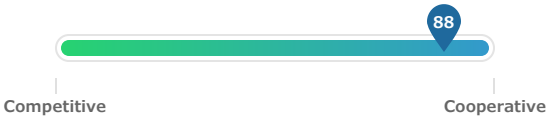
Conscientiousness



Conscientiousness is the tendency to be organized, hard working and responsible in one's approach to your work. Individuals with high levels of this personality trait are more likely to be ambitious and tend to be goal-oriented and focused.

- You value order and self discipline and tends to pursue ambitious endeavours.
- You believe in the importance of structure and is very well-organized.
- You carefully review facts before arriving at conclusions or making decisions based on them.
- You strictly adhere to rules and carefully consider the situation before making decisions.
- You tend to have a high level of self confidence and do not doubt your abilities.
- You generally set and work toward goals, try to exceed expectations and are likely to excel in most jobs, especially those which require careful or meticulous approach.

Agreeableness



Agreeableness refers to an individual's tendency to be cooperative with others and it defines your approach to interpersonal relationships. People with high levels of this personality trait tend to be more considerate of people around them and are more likely to work effectively in a team.

- You are considerate and sensitive to the needs of others.
- You tend to put the needs of others ahead of your own.
- You are likely to trust others easily without doubting their intentions.
- You are compassionate and may be strongly affected by the plight of both friends and strangers.
- You are humble and modest and prefer not to talk about personal accomplishments.
- Your personality is more suitable for jobs demanding cooperation among employees.

Openness to Experience

Openness to experience refers to a person's inclination to explore beyond conventional boundaries in different aspects of life. Individuals with high levels of this personality trait tend to be more curious, creative and innovative in nature.

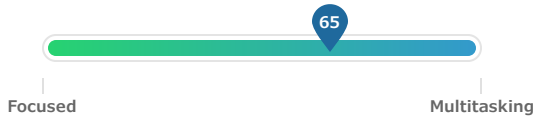
- You tend to be curious in nature and is generally open to trying new things outside your comfort zone.
- You may have a different approach to solving conventional problems and tend to experiment with those solutions.
- You are creative and tends to appreciate different forms of art.
- You are likely to be in touch with your emotions and is quite expressive.
- Your personality is more suited for jobs requiring creativity and an innovative approach to problem solving.

Emotional Stability

Emotional stability refers to the ability to withstand stress, handle adversity, and remain calm and composed when working through challenging situations. People with high levels of this personality trait tend to be more in control of their emotions and are likely to perform consistently despite difficult or unfavourable conditions.

- You are calm and composed in nature.
- You tend to maintain composure during high pressure situations.
- You are very confident and comfortable being yourself.
- You find it easy to resist temptations and practice moderation.
- You are likely to remain emotionally stable in jobs with high stress levels.

Polychronicity



Polychronicity refers to a person's inclination to multitask. It is the extent to which the person prefers to engage in more than one task at a time and believes that such an approach is highly productive. While this trait describes the personality disposition of a person to multitask, it does not gauge their ability to do so successfully.

- You neither have a strong preference nor dislike to perform multiple tasks simultaneously.
- You are open to both options - pursuing multiple tasks at the same time or working on a single project at a time.
- Whether or not you will succeed in a polychronous environment depends largely on your ability to do so.

3 | Response

Automata Pro

4 / 100 Code Replay

Question 1 (Language: Java)

In a city there are N houses. A buyer is looking for a plot of land in the city on which to build their house. They want to buy the largest plot of land that will allow them to build the largest possible house. All the houses in the city lie in a straight line and all of them have a house number and a second number indicating the position of the house relative to the entry point in the city. The buyer wants to find the houses between which they can build the largest possible house.

Write an algorithm to help the buyer find the house numbers between which they can build their largest possible house.

Scores

Programming Ability

20 / 100

Code seems to be unrelated to the given problem.

Programming Practices

0 / 100

Programming practices score cannot be generated. This is because source code has syntax/runtime errors and is unparseable or the source code does not meet the minimum code-length specifications.

Functional Correctness

20 / 100

The source code does not pass any basic test cases. It is either due to incorrect logic or runtime errors. Some advanced or edge cases may randomly pass.

Final Code Submitted Compilation Status: Pass

```
1  import java.util.*;
2  import java.lang.*;
3  import java.io.*;
4
5  /*
6   * In the code, houses grid represents N rows of houses and two co
   lumns representing the house number and positions of N houses.
7   */
8  public class Solution
9  {
10    public static int[]  largestLand(int[][] houses)
11    {
12       int[]  answer = new int[2];
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code: O(N logN)

*N represents number of houses

Errors/Warnings

There are no errors in the candidate's code.

```
13      // Write your code here
14
15      System.out.println("4 5");
16
17      return answer;
18    }
19
20    public static void main(String[] args)
21    {
22      Scanner in = new Scanner(System.in);
23      // input for houses
24      int houses_row = in.nextInt();
25      int houses_col = in.nextInt();
26      int houses[][] = new int[houses_row][houses_col];
27      for(int idx = 0; idx < houses_row; idx++)
28      {
29        for(int jdx = 0; jdx < houses_col; jdx++)
30        {
31          houses[idx][jdx] = in.nextInt();
32        }
33      }
34
35      int[] result = largestLand(houses);
36      for(int idx = 0; idx < result.length - 1; idx++)
37      {
38        System.out.print(result[idx] + " ");
39      }
40      System.out.print(result[result.length - 1]);
41    }
42  }
43
```

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution Passed TC: **0%**

Total score 0/11

0%	0%	0%
Basic(**0**/4)	Advance(**0**/6)	Edge(**0**/1)

Compilation Statistics

3	3	0	0	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:11:03**
Average time taken between two compile attempts:	**00:03:41**
Average test case pass percentage per compile:	**0%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 2 (Language: Java)

The city authorities conduct a study of the houses in a residential area for a city planning project. The area is depicted in an aerial view and divided into an N x M grid. If a grid cell contains some part of a house roof, then it is assigned the value 1; otherwise, the cell represents a vacant lot and is assigned the value 0. Clusters of adjacent grid cells with value 1 represent a single house. Diagonally placed grids with value 1 do not represent a single house. The area of a house is the number of 1s that it spans.

Write an algorithm to find the area of the largest house.

Scores

Programming Ability

0 / 100

NA

Programming Practices

0 / 100

Programming practices score cannot be generated. This is because source code has syntax/runtime errors and is unparseable or the source code does not meet the minimum code-length specifications.

Functional Correctness

0 / 100

NA

Final Code Submitted

Compilation Status: Pass

```
1  import java.util.*;
2  import java.lang.*;
3  import java.io.*;
4
5  /*
6   * grid represents the two-dimensional grid with N rows and M colu
   mns.
7   */
8  public class Solution
9  {
10   public static int  areaOfLargestHouse(int[][] grid)
11   {
12      int  answer = 0;
13      // Write your code here
14
15
16      return answer;
17   }
18
19   public static void main(String[] args)
20   {
21      Scanner in = new Scanner(System.in);
22      // input for grid
23      int grid_row = in.nextInt();
24      int grid_col = in.nextInt();
25      int grid[][] = new int[grid_row][grid_col];
26      for(int idx = 0; idx < grid_row; idx++)
27      {
28         for(int jdx = 0; jdx < grid_col; jdx++)
29         {
30            grid[idx][jdx] = in.nextInt();
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code: $O(N^2)$

*N represents number of rows/columns of the input grid

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

```
31          }
32      }
33
34      int result = areaOfLargestHouse(grid);
35      System.out.print(result);
36
37  }
38 }
39
```

Test Case Execution

Passed TC: **0%**

Total score

0/19

0%	0%	0%
Basic(**0**/6)	Advance(**0**/10)	Edge(**0**/3)

Compilation Statistics

1	1	0	0	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:02:45**
Average time taken between two compile attempts:	**00:02:45**
Average test case pass percentage per compile:	**5.26%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Automata Fix

0 / 100 Code Replay

Question 1 (Language: C++)

You are given a predefined structure/class ***Point*** and also a collection of related functions/methods that can be used to perform some basic operations on the structure.

The function/method ***isRightTriangle*** returns an integer '1', if the points make a right-angled triangle otherwise return '0'.
The function/method ***isRightTriangle*** accepts three points - *P1, P2, P3* representing the input points.

You are supposed to use the given function to complete the code of the function/method ***isRightTriangle*** so that it passes all test cases.

Helper Description
The following class is used to represent point and is already implemented in the default code (Do not write these definitions again in your code):

```
class Point
{
   private:
   int X;
   int Y;
   double Point_calculateDistance(Point *point1, Point *point2)
   {
      /*Return the euclidean distance between two input points.
      This can be called as -
      *  If P1 and P2 are two points then -
      *  P1->Point_calculateDistance(P2);*/
    }
}
```

Scores

Final Code Submitted **Compilation Status: Fail**

```
1 // You can print the values to stdout for debugging
2 using namespace std;
```

Code Analysis

Average-case Time Complexity

```
3 int isRightTriangle(Point *P1, Point *P2, Point *P3)
4 {
5    // write your code here
6 }
7
```

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

```
In file included from main_23.cpp:8:
source_23.cpp: In function 'int isRightTriangle(Point*,
Point*, Point*)':
source_23.cpp:6:1: error: no return statement in
function returning non-void [-Werror=return-type]
}
^
cc1plus: some warnings being treated as errors
```

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Compilation Statistics

0	0	0	0	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:02:15**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**0%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 2 (Language: C++)

The function/method ***manchester*** print space-separated integers with the following property: for each element in the input list *arr*, if the bit arr[i] is the same as *arr*[i-1], then the element of the output list is 0. If they are different, then its 1. For the first bit in the input list, assume its previous bit to be 0. This encoding is stored in a new list.

The function/method ***manchester*** accepts two arguments - *len,* an integer representing the length of the list and *arr* and *arr*, a list of integers, respectively. Each element of *arr* represents a bit - 0 or 1

For example - if *arr* is {0 1 0 0 1 1 1 0}, the function/method should print an list {0 1 1 0 1 0 0 1}.

The function/method compiles successfully but fails to print the desired result for some test cases due to logical errors. Your task is to fix the code so that it passes all the test cases.

Scores

Final Code Submitted **Compilation Status: Pass**

```
1 // You can print the values to stdout for debugging
2 void manchester(int len, int* arr)
3 {
4    int *res = new int[len];
5    res[0] = arr[0];
6    for(int i = 1; i < len; i++){
7      res[i] = (arr[i]==arr[i-1]);
8    }
9    for(int i=0; i<len; i++)
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

```
10      printf("%d ", res[i]);
11  }
```

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: **20%**

Total score 1/5

50%	0%	0%
Basic(**1**/2)	Advance(**0**/3)	Edge(**0**/0)

Compilation Statistics

0	0	0	1	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:01:16**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**20%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 3 (Language: C++)

The function/method ***drawPrintPattern*** accepts *num*, an integer.
The function/method ***drawPrintPattern*** prints the first *num* lines of the pattern shown below.

For example, if *num* = 3, the pattern should be:
1 1
1 1 1 1
1 1 1 1 1 1

The function/method ***drawPrintPattern*** compiles successfully but fails to get the desired result for some test cases due to incorrect implementation of the function/method. Your task is to fix the code so that it passes all the test cases.

Scores

Final Code Submitted **Compilation Status: Fail**

```
1  using namespace std;
2  void drawPrintPattern(int num)
3  {
4     int i,j,print = 1;
5     for(i=1;i<=num;i++)
6     {
7        for(j=1;j<=2*i;j++);
8        {
9           cout << print <<" ";
10       }
11       cout <<"\n";
12    }
13    return drawPrintPattern();
14 }
15
16 int main() {
17    int i,j,print = 1;
18    int num;
19    cin >> num;
20    return 0;
21 }
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

```
In file included from main_7.cpp:7:
source_7.cpp: In function 'void drawPrintPattern(int)':
source_7.cpp:13:29: error: too few arguments to
function 'void drawPrintPattern(int)'
return drawPrintPattern();
^
source_7.cpp:2:6: note: declared here
void drawPrintPattern(int num)
^~~~~~~~~~~~~~~~~
source_7.cpp:13:29: error: return-statement with a
value, in function returning 'void' [-fpermissive]
return drawPrintPattern();
^
main_7.cpp: At global scope:
main_7.cpp:8:5: error: conflicting declaration of C
function 'int main(int, const char**)'
int main (int argc, const char compile compile.meta
compile.out compile.tmp testcases validate
validate.meta validate.out validate.tmp worstcases
argv[])
^~~~
In file included from main_7.cpp:7:
source_7.cpp:16:5: note: previous declaration 'int
```

main()'
int main() {
^~~~

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Compilation Statistics

6	2	4	2	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:06:31**
Average time taken between two compile attempts:	**00:01:05**
Average test case pass percentage per compile:	**0%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 4 (Language: C++)

The function/method ***median*** accepts two arguments - *size* and *inputList*, an integer representing the length of a list and a list of integers, respectively.

The function/method ***median*** is supposed to calculate and return an integer representing the median of elements in the input list. However, the function/method **median** works only for odd-length lists because of incomplete code.

You must complete the code to make it work for even-length lists as well. A couple of other functions/methods are available, which you are supposed to use inside the function/method **median** to complete the code.

Helper Description

The following function is used to represent a quick_select and is already implemented in the default code (Do not write this definition again in your code):

```
int quick_select(int* inputList, int start_index, int end_index, int median_order)
{
  /*It calculate the median value
  This can be called as -
  quick_select(inputList, start_index, end_index, median_order)
  where median_order is the half length of the inputList
}
```

Scores

Final Code Submitted **Compilation Status: Pass**

```
// You can print the values to stdout for debugging
using namespace std;
float median(int size, int* inputList)
{
   int start_index = 0;
   int end_index = size-1;
   float res = -1;
   if(size%2!=0) // odd length arrays
   {
      int median_order = ((size+1)/2);
      res = (float)quick_select(inputList, start_index, end_index, median_order);
   }
   else // even length arrays
   {
      // write your code here
   }
   return res;
}
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: **42.86%**

Total score

3/7

50%	0%	100%
Basic(**2**/4)	Advance(**0**/2)	Edge(**1**/1)

Compilation Statistics

0	0	0	1	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:00:59**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**42.9%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 5 (Language: C++)

The function/method ***multiplyNumber*** returns an integer representing the multiplicative product of the maximum two of three input numbers. The function/method ***multiplyNumber*** accepts three integers- *numA*, *numB* and *numC*, representing the input numbers.

The function/method ***multiplyNumber*** compiles unsuccessfully due to syntactical error. Your task is to debug the code so that it passes all the test cases.

Scores

Final Code Submitted — Compilation Status: Fail

```
1  // You can print the values to stdout for debugging
2  using namespace std;
3  int multiplyNumber(int numA, int numB, int numC)
4  {
5    int result,min,max,mid;
6    max=(numA>numB)?((numA>numC)?numA:numC):(numB>numC)?numB:numC);
7    min=(numA<numB)?((numA<numC)?numA:numC):((numB<numC)?numB:numC);
8    //mid=(numA+numB+numC)-(min+max);
9    result= max * min;
10   return result;
11 }
12
13 int main() {
14   cout << result;
15 }
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

```
In file included from main_31.cpp:7:
source_31.cpp: In function 'int multiplyNumber(int, int, int)':
source_31.cpp:6:65: error: expected ';' before ')' token
max=(numA>numB)?((numA>numC)?numA:numC):(numB>numC)?numB:numC);
^
;
source_31.cpp: In function 'int main()':
source_31.cpp:14:12: error: 'result' was not declared in this scope
cout << result;
^~~~~~
main_31.cpp: At global scope:
main_31.cpp:9:5: error: conflicting declaration of C function 'int main(int, const char**)'
int main (int argc, const char compile compile.meta compile.out compile.tmp testcases validate validate.meta validate.out validate.tmp worstcases argv[])
^~~~
In file included from main_31.cpp:7:
source_31.cpp:13:5: note: previous declaration 'int main()'
int main() {
^~~~
```

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Compilation Statistics

4	0	4	0	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:06:07**
Average time taken between two compile attempts:	**00:01:32**
Average test case pass percentage per compile:	**0%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 6 (Language: C++)

The function/method ***sameElementCount*** returns an integer representing the number of elements of the input list which are even numbers and equal to the element to its right. For example, if the input list is [4 4 4 1 8 4 1 1 2 2] then the function/method should return the output '3' as it has three similar groups i.e, (4, 4), (4, 4), (2, 2).

The function/method ***sameElementCount*** accepts two arguments - *size*, an integer representing the size of the input list and *inputList*, a list of integers representing the input list.

The function/method compiles successfully but fails to return the desired result for some test cases due to incorrect implementation of the function/method ***sameElementCount***. Your task is to fix the code so that it passes all the test cases.

Note:
In a list, an element at index i is considered to be on the left of index i+1 and to the right of index i-1. The last

element of the input list does not have any element next to it which makes it incapable to satisfy the second condition and hence should not be counted.

Scores

Final Code Submitted **Compilation Status: Pass**

```
1  // You can print the values to stdout for debugging
2  using namespace std;
3  int sameElementCount(int size, int* inputList)
4  {
5     int i,count =0;
6     for(i=0;i<size-1;i++)
7     {
8       if((inputList[i]%2==0)&&(inputList[i]==inputList[i++]))
9         count++;
10    }
11    return count;
12 }
13
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution Passed TC: **50%**

Total score 4/8

43%	0%	100%
Basic(**3**/7)	Advance(**0**/0)	Edge(**1**/1)

Compilation Statistics

0	0	0	1	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:00:03**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**50%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 7 (Language: C++)

The function/method ***countOccurrence*** return an integer representing the count of occurrences of given value in the input list.
The function/method ***countOccurrence*** accepts three arguments - *len*, an integer representing the size of the input list, *value,* an integer representing the given value and *arr*, a list of integers, representing the input list.

The function/method ***countOccurrence*** compiles successfully but fails to return the desired result for some test cases due to logical errors. Your task is to fix the code so that it passes all the test cases.

Scores

Final Code Submitted **Compilation Status: Pass**

```
1  // You can print the values to stdout for debugging
2  int countOccurrence(int len, int value, int *arr)
3  {
4     int i=0, count = 0;
5     while(i<len)
6     {
7        if(arr[i]==value)
8           count += 1;
9     }
10    return count;
11 }
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: **11.11%**

Total score

1/9

0%	0%	100%
Basic(**0**/3)	Advance(**0**/5)	Edge(**1**/1)

Compilation Statistics

0	0	0	1	1	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:00:11**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**11.1%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

4 | Learning Resources

English Comprehension

- Learn about written english comprehension
- Learn about spoken english comprehension
- Test your comprehension skills

Logical Ability

- Watch a video on the art of deduction
- Learn about Sherlock Holmes' puzzles and develop your deductive logic
- Take a course on advanced logic

Quantitative Ability (Advanced)

- Learn about percentages
- Learn about simple and compount interests
- Watch a video on time, speed and distance

Icon Index

Free Tutorial | Paid Tutorial | Youtube Video | Web Source

Wikipedia | Text Tutorial | Video Tutorial | Google Playstore