# Rohan

**Test ID:** 450015228868474 | ☎ 8279360051 | ✉ rohan.232@lpu.in

**Test Date:** November 29, 2025

| Computer Science | Logical Ability | Computer Programming | Quantitative Ability (Advanced) |
|---|---|---|---|
| 29 /100 | 70 /100 | 62 /100 | 58 /100 |

| English Comprehension | Automata Fix | Automata Pro | Personality |
|---|---|---|---|
| 58 /100 | 29 /100 | 89 /100 | ✓ Completed |

## Computer Science

29 / 100

| OS and Computer Architecture | DBMS | Computer Networks |
|---|---|---|
| 21 / 100 | 46 / 100 | 29 / 100 |

## Logical Ability

70 / 100

| Inductive Reasoning | Deductive Reasoning | Abductive Reasoning |
|---|---|---|
| 71 / 100 | 77 / 100 | 63 / 100 |

## Computer Programming

62 / 100

| Basic Programming | Data Structures | OOP and Complexity Theory |
|---|---|---|
| 63 / 100 | 70 / 100 | 54 / 100 |

## Quantitative Ability (Advanced)

58 / 100

| Basic Mathematics | Advanced Mathematics | Applied Mathematics |
|---|---|---|
| 62 / 100 | 55 / 100 | 56 / 100 |

## English Comprehension

58 / 100

CEFR: **B2**

| Grammar | Vocabulary | Comprehension |
|---|---|---|
| 68 / 100 | 52 / 100 | 53 / 100 |

## Automata Fix

29 / 100

| Code Reuse | Logical Error | Syntactical Error |
|---|---|---|
| 0 / 100 | 50 / 100 | 0 / 100 |

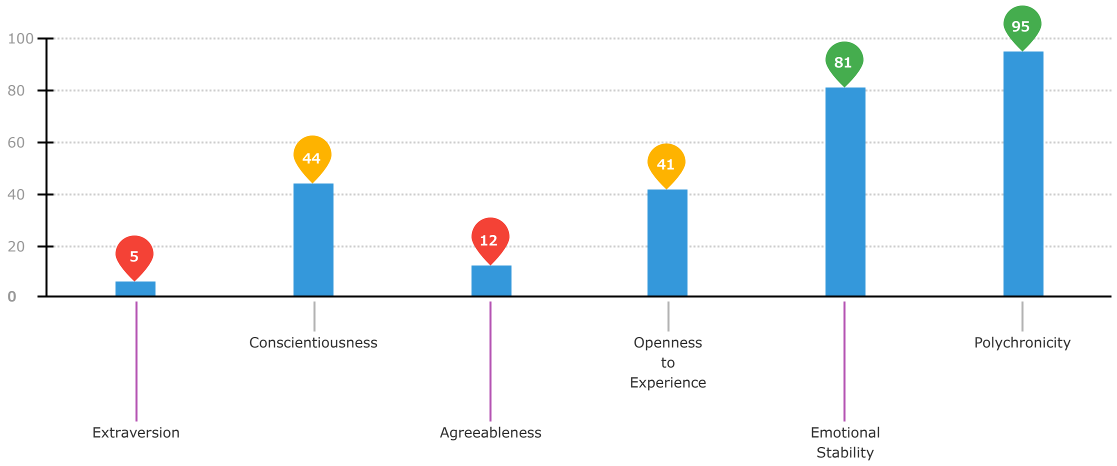
## Automata Pro

89 / 100

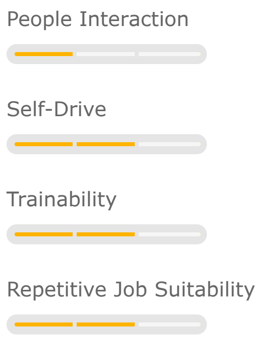
| Programming Practices | Functional Correctness |
|---|---|
| 100 / 100 | 80 / 100 |

## Personality

Completed



Competencies



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

58 / 100 | CEFR: **B2**

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

70 / 100

#### Inductive Reasoning

71 / 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

77 / 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.

#### Abductive Reasoning

63 / 100

### Quantitative Ability (Advanced)

58 / 100

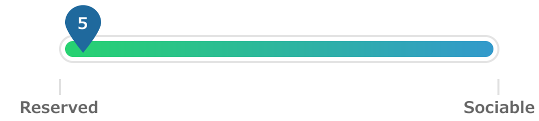
This test aims to measure your ability to solve problems on basic arithmetic operations, probability, permutations and combinations, and other advanced concepts.

You are able to solve word problems on basic concepts of percentages, ratio, proportion, interest, time and work. Having a strong hold on these concepts can help you understand the concept of work efficiency and how interest is accrued on bank savings. It can also guide you in time management, work planning, and resource allocation in complex projects.

### Personality

## Competencies

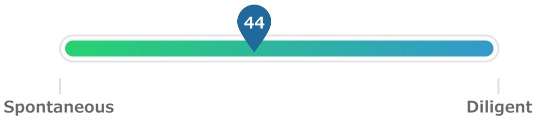
### Extraversion



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 feel comfortable spending time by yourself.
- You prefer spending time alone rather than in social gatherings.
- You may not enjoy activities that involve thrill and excitement.
- You are thoughtful, introspective and refrains from impulsive remarks/actions. You often keep your opinions and ideas to yourself
- You prefer to work on individual projects rather than group projects.
- You are more likely to prefer jobs that require minimal interaction with people.

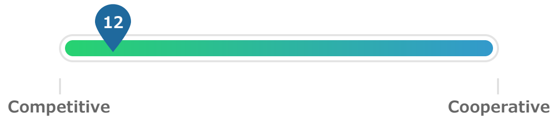
### Conscientiousness



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 are flexible and able to adapt your work pace to the job at hand.
- You are usually spontaneous but you are likely to stick to a plan whenever necessary.
- You tend to be cautious when you deem it necessary.
- You may prefer to act according to the rules.
- You are confident in your ability to achieve goals but may need support to overcome occasional setbacks.
- You are an efficient worker and try to perform better than your peers. You are well suited for jobs allowing flexibility regarding operating procedures.

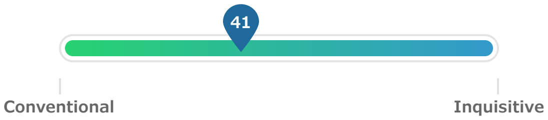
## Agreeableness



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 outspoken. You often play the role of a devil's advocate in discussions and question others' opinions and views.
- You are not gullible and are likely to carefully examine the situation before trusting something/someone.
- You may not be strongly affected by human suffering and may be perceived as indifferent.
- You are confident of your achievements and do not shy away from talking about them.
- You sometimes place self-interest above the needs of those around you. You are not willing to compromise your own views in order to accommodate the views of others.
- You are suitable for jobs that require tough objective decisions and hard negotiation.

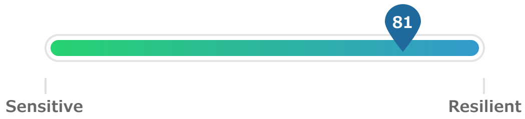
## Openness to Experience



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 may try new things but would prefer not to venture too far beyond your comfort zone.
- You tend to be open to accepting abstract ideas after weighing them against existing solutions.
- You appreciate the arts to a certain extent but may lack the curiosity to explore them in depth.
- You may express your feelings only to people you are comfortable with.
- Your personality is more suited for jobs involving a mix of logical and creative thinking.

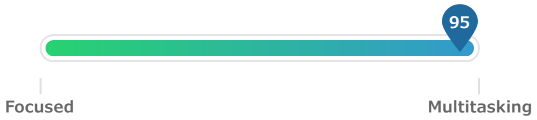
## Emotional Stability



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 pursue multiple tasks simultaneously, switching between them when needed.
- You prefer working to achieve some progress on multiple tasks simultaneously than completing one task before moving on to the next task.
- You tend to believe that multitasking is an efficient way of doing things and prefers an action packed work life with multiple projects.

## 3 | Response

### Automata Pro

● 89 / 100 | Code Replay

### Question 1 (Language: C++20)

You are given a list of N unique positive numbers ranging from 0 to (N -1). Write an algorithm to replace the value of each number with its corresponding index value in the list.

### Scores

**Programming Practices**

100 / 100

High readability, high on program structure. The source code is readable and does not consist of any significant redundant/improper coding constructs.

**Functional Correctness**

100 / 100

Functionally correct source code. Passes all the test cases in the test suite for a given problem.

### Final Code Submitted — Compilation Status: Pass

```
 1 // Sample code to read input and write output:
 2
 3 /*
 4 #include <iostream>
 5
 6 using namespace std;
 7
 8 int main()
 9 {
10     char name[20];
11     cin >> name;              // Read input from STDIN
12     cout << "Hello " << name;     // Write output to STDOUT
13     return 0;
14 }
15 */
16
17 // Warning: Printing unwanted or ill-formatted data to output will cause the test cases to fail
18
19 #include <iostream>
20
21 using namespace std;
22
23 int main()
24 {
⚠25     int n;
```

### 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)

*N represents number of elements in the input list

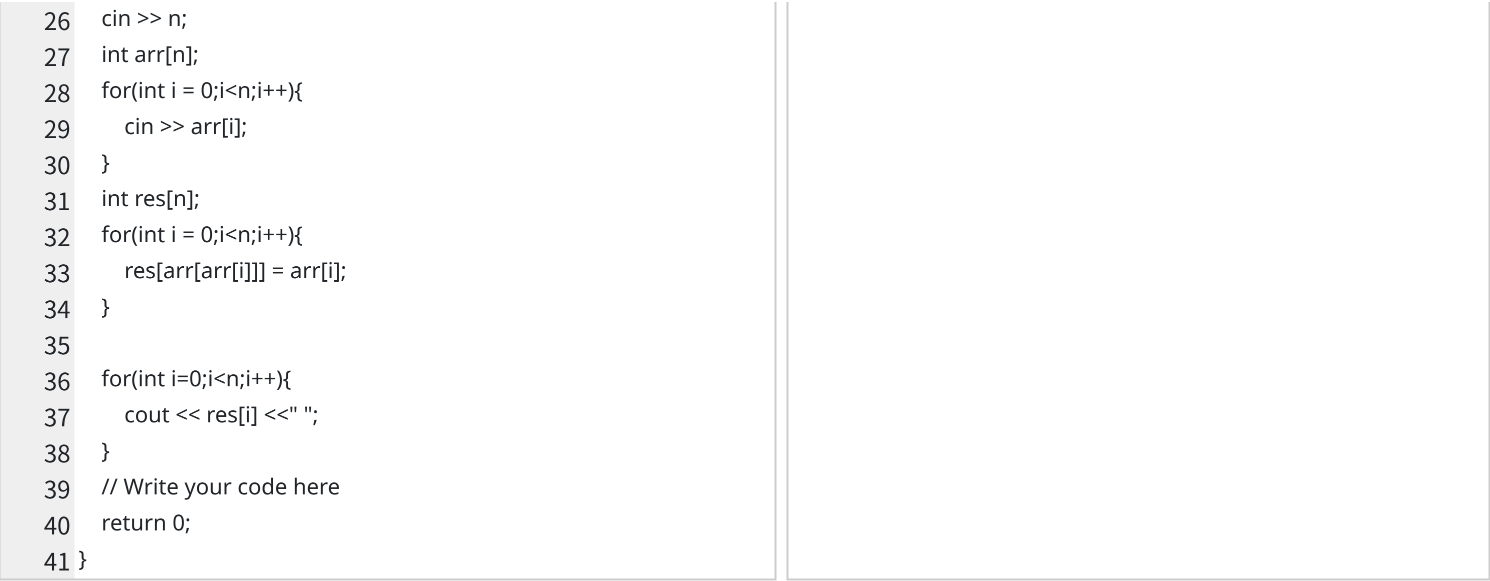
**Errors/Warnings**

There are no errors in the candidate's code.

**Structural Vulnerabilites and Errors**

**Readability & Language Best Practices**

Line 25: Variables are given very short name.



```
    cin >> n;
    int arr[n];
    for(int i = 0;i<n;i++){
        cin >> arr[i];
    }
    int res[n];
    for(int i = 0;i<n;i++){
        res[arr[arr[i]]] = arr[i];
    }

    for(int i=0;i<n;i++){
        cout << res[i] <<" ";
    }
    // Write your code here
    return 0;
}
```

## Test Case Execution

Passed TC: **100%**

| Total score | | | |
|---|---|---|---|
| 10/10 | **100%**<br>Basic(**7**/7) | **100%**<br>Advance(**1**/1) | **100%**<br>Edge(**2**/2) |

## Compilation Statistics

| 3 | 2 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:08:49** |
| Average time taken between two compile attempts: | **00:02:56** |
| 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 2 (Language: C++20)

In a connected graph, a path runs between every node. This path does not need to be an edge directly connecting the nodes. An adjacency matrix for a graph with *n* vertices is an *n x n* two-dimensional matrix with *i j* entry as 1 if there is an edge from the ith vertex to the jth vertex; otherwise it is 0.

An undirected connected graph is given in the adjacency matrix form. Write an algorithm to determine whether it is a tree.

For example, the result for the adjacency matrix given below should be 1 as it represents a tree.
0 1 0 1
1 0 1 0
0 1 0 0
1 0 0 0

### Scores

**Programming Practices**

100 / 100

High readability, high on program structure. The source code is readable and does not consist of any significant redundant/improper coding constructs.

**Functional Correctness**

60 / 100

Correct basic functionality with partially correct advanced functionality. Passes all the basic test cases in the test suite and a percentage of the advanced test cases.

**Final Code Submitted** **Compilation Status: Pass**

```
// Sample code to read input and write output:

```

**Code Analysis**

**Average-case Time Complexity**

```
 3 /*
 4 #include <iostream>
 5 
 6 using namespace std;
 7 
 8 int main()
 9 {
10     char name[20];
11     cin >> name;                    // Read input from STDIN
12     cout << "Hello " << name;       // Write output to STDOUT
13     return 0;
14 }
15 */
16 
17 // Warning: Printing unwanted or ill-formatted data to output will cause the test cases to fail
18 
19 #include <iostream>
20 
21 using namespace std;
22 
23 int main()
24 {
25     int n,m;
26     cin >>n;
27     cin >> m;
28     int arr[n][m];
29     int ans;
30     for(int i = 0;i<n;i++){
31         for(int j = 0;j<m;j++){
32             cin >> arr[i][j];
33         }
34     }
35     // for(int i = 0;i<n-1;i++){
36     //     if(arr[i][i] != arr[i+1][i+1] ) ans = 0;
37     //     else ans = 1;
38     // }
39     for(int i = 0;i<n;i++){
40         for(int j = 0;j<m;j++){
41             if(arr[i][j] != arr[j][i]){
42                 ans = 0;
43             }
44             else ans = 1;
45         }
46     }
47     if(ans == 1){
48             for(int i = 0;i<n-1;i++){
49         if(arr[i][i] != arr[i+1][i+1] ) ans = 0;
50         else ans = 1;
51     }
```

**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 vertices in a fully-connected graph

### Errors/Warnings

There are no errors in the candidate's code.

### Structural Vulnerabilites and Errors

**Readability & Language Best Practices**

Line 25: Variables are given very short name.

```

    }



    // if(n%2==0){
    //     ans = 0;
    // }
    // else ans = 1;

    // if(arr[0][0] == arr[n-1][m-1]) ans = 1;
    // else ans = 0;

    cout << ans;

    // Write your code here
    return 0;
}
```

## Test Case Execution

Passed TC: **70.59%**

Total score 12/17

| 100% | 50% | 67% |
|---|---|---|
| Basic(**6**/6) | Advance(**4**/8) | Edge(**2**/3) |

## Compilation Statistics

| 42 | 39 | 3 | 0 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:30:54** |
| Average time taken between two compile attempts: | **00:00:44** |
| Average test case pass percentage per compile: | **52.52%** |

### 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

**29** / 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: Pass**

```
// You can print the values to stdout for debugging
using namespace std;
int isRightTriangle(Point *P1, Point *P2, Point *P3)
{
    int a = P1->Point_calculateDistance(P2);
    int b = P2->Point_calculateDistance(P3);
    int c = P3->Point_calculateDistance(P1);

    if(a*a == b*b + c*c || b*b == a*a+c*c || c*c == a*a+b*b) return 1;

    return 0;
}

```

### 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: **87.5%**

Total score

7/8

| 100% | 75% | 0% |
|---|---|---|
| Basic(**4**/4) | Advance(**3**/4) | Edge(**0**/0) |

## Compilation Statistics

| | | | | | |
|---|---|---|---|---|---|
| 2 | 1 | 1 | 0 | 0 | 0 |
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:04:06** |
| Average time taken between two compile attempts: | **00:02:03** |
| Average test case pass percentage per compile: | **87.5%** |

### 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**

```
// You can print the values to stdout for debugging
void manchester(int len, int* arr)
{
  int *res = new int[len];
  res[0] = arr[0];
  for(int i = 1; i < len; i++){
   if(arr[i] == arr[i-1] ){
    res[i] = arr[i] == 0 ? 1 : 0;
   }
   else{
    res [i] = arr[i];
   }
   // res[i] = (arr[i]==arr[i-1]);
  }
  for(int i=0; i<len; i++)
    printf("%d ", res[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

#### 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

| 3 | 2 | 1 | 2 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:04:22** |
| Average time taken between two compile attempts: | **00:01:27** |
| 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 3 (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
  {
   int median_order = ((size+1)/2);
    res = (float)quick_select(inputList, start_index, end_index, median_order-1);
    res = res/2;
  }
  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: **57.14%**

Total score 4/7

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

### Compilation Statistics

| 10 | 10 | 0 | 9 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:05:22** |
| Average time taken between two compile attempts: | **00:00:32** |
| Average test case pass percentage per compile: | **18.6%** |

### 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 ***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**

```
// You can print the values to stdout for debugging
int countOccurrence(int len, int value, int *arr)
{
    int count = 0;
    for(int i = 0;i<len;i++){
        if(arr[i] == value){
            count++;
        }
    }
    return count;
}
```

### 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: **100%**

Total score

9/9

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

### Compilation Statistics

| 2 | 2 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:01:32** |
| Average time taken between two compile attempts: | **00:00:46** |
| 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 5 (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: Pass

```
using namespace std;
void drawPrintPattern(int num)
{
  int i,j,print = 1;
  for(i=1;i<=num;i++)
  {
    for(j=1;j<=2*i;j++)
    {
      cout<<print<<" ";
    }
    cout<<"\n";
  }
}
```

### 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: **100%**

Total score 8/8

| **100%** | **0%** | **100%** |
|---|---|---|
| Basic(**7**/7) | Advance(**0**/0) | Edge(**1**/1) |

### Compilation Statistics

| 7 | 7 | 0 | 6 | 0 | 0 |
|---|---|---|---|---|---|
| Total attempts | Successful | Compilation errors | Sample failed | Timed out | Runtime errors |

| | |
|---|---|
| Response time: | **00:02:24** |
| Average time taken between two compile attempts: | **00:00:21** |
| Average test case pass percentage per compile: | **14.3%** |

### 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 ***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**

```
// You can print the values to stdout for debugging
using namespace std;
int multiplyNumber(int numA, int numB, int numC)
{
  int result,min,max,mid;
  max=(numA>numB)?numA>numC)?numA:numC):(numB>numC)?numB:numC);
  min=(numA<numB)?((numA<numC)?numA:numC):((numB<numC)?numB:numC);
  mid=(numA+numB+numC)-(min+max);
  result=(max*int(mid));
  return result;
}
```

### 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:6:29: error: expected ':' before ')'
token
max=(numA>numB)?numA>numC)?numA:numC):
(numB>numC)?numB:numC);
^
:
source_31.cpp:6:29: error: expected primary-
expression before ')' token
```

**Structural Vulnerabilites and Errors**

There are no errors in the candidate's code.

### Compilation Statistics

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

| | |
|---|---|
| Response time: | **00:01:50** |
| Average time taken between two compile attempts: | **00:01:50** |
| 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 7 (Language: Java)

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

The candidate did not make any changes in the code.

## Compilation Statistics

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

| | |
|---|---|
| Response time: | **00:00:00** |
| 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

## 4 | Learning Resources

### English Comprehension

| | | | |
|---|---|---|---|
| Learn about business e-mail etiquettes | FREE | 🌐 | 📖 |
| Learn about written english comprehension | FREE | 🌐 | 🎬 |
| Learn about spoken english comprehension | FREE | 🌐 | 🎬 |

### Logical Ability

| | | | |
|---|---|---|---|
| Take a course on advanced logic | FREE | 🌐 | 🎬 |
| Test your deduction skills! | FREE | 🌐 | 📖 |
| Learn about advanced deductive logic | FREE | 🌐 | 📖 |

### Quantitative Ability (Advanced)

| | | | |
|---|---|---|---|
| Watch a video on the history of algebra and its applications | FREE | 🌐 | 🎬 |
| Learn about proportions and its practical usage | FREE | 🌐 | 📖 |
| Learn about calculating percentages manually | FREE | ▶ | 🎬 |

### Icon Index

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FREE | Free Tutorial | $ | Paid Tutorial | ▶ | Youtube Video | 🌐 | Web Source |
| ▶ | Wikipedia | 📖 | Text Tutorial | 🎬 | Video Tutorial | ▷ | Google Playstore |