

Mohammad Dayyan

Test ID: 432008779000030 | 9529719884 | 24dayyanm@rbunagpur.in

Test Date: January 9, 2025

Logical Ability 48 /100	Computer Programming 49 /100	Quantitative Ability (Advanced) 36 /100	English Comprehension 55 /100
Automata 4 /100	Automata Fix 29 /100	Personality Completed	

Logical Ability			48 / 100
Inductive Reasoning	Deductive Reasoning	Abductive Reasoning	
51 / 100	49 / 100	44 / 100	

Computer Programming			49 / 100
Basic Programming	Data Structures	OOP and Complexity Theory	
47 / 100	43 / 100	57 / 100	

Quantitative Ability (Advanced)			36 / 100
Basic Mathematics	Advanced Mathematics	Applied Mathematics	
35 / 100	37 / 100	37 / 100	

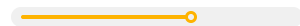
English Comprehension



55 / 100

CEFR: **B2**

Grammar



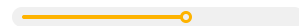
61 / 100

Vocabulary



45 / 100

Comprehension



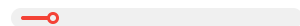
59 / 100

Automata



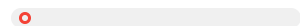
4 / 100

Programming Ability



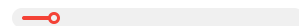
10 / 100

Programming Practices



0 / 100

Functional Correctness



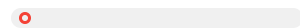
10 / 100

Automata Fix



29 / 100

Code Reuse



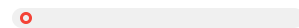
0 / 100

Logical Error



50 / 100

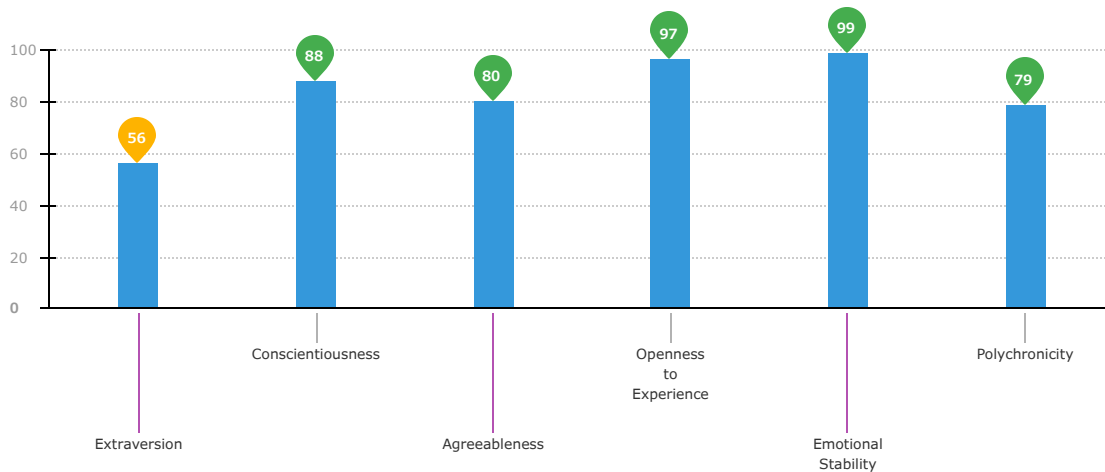
Syntactical Error



0 / 100

Personality

Completed



Competencies

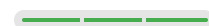
People Interaction



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:

- Scores between 67 and 100
- Scores between 33 and 67
- Scores between 0 and 33

2 | Insights

English Comprehension



55 / 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



48 / 100



Inductive Reasoning



51 / 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.



Deductive Reasoning



49 / 100

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

You are able to work out simple rules based on specific evidence or information. This skill is required in high end analytics jobs where one is required to infer patterns based on predefined rules from different sets of data.



Abductive Reasoning



44 / 100

Quantitative Ability (Advanced)



36 / 100

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

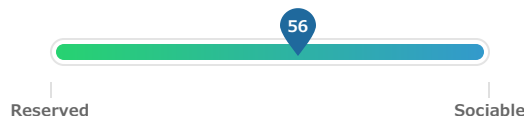
You are able to perform simple arithmetic operations. Apart from their relevance in monetary transactions, these operations are used in other situations, such as dividing up tasks with one's colleagues, managing one's time at work, and planning the resources required to complete a task.

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 comfortable socializing to a certain extent. You prefer small gatherings in familiar environments.
- You feel at ease interacting with your close friends but may be reserved among strangers.
- You indulge in activities involving thrill and excitement that are not too risky.
- You contemplate the consequences before expressing any opinion or taking an action.
- You take charge when the situation calls for it and you are comfortable following instructions as well.
- Your personality may be suitable for jobs demanding flexibility in terms of working well with a team as well as individually.



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 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



4 / 100

[Code Replay](#)

Question 1 (Language: Java 11)

A company is transmitting its data to another server. To secure the data against malicious activity, they plan to reverse the data before transmitting. They want to know the number of data characters that do not change position even after the data stream is reversed. The network administrator has been tasked with ensuring the smooth transmission of the data.

Write an algorithm for the network administrator to help in finding the number of data characters that do not change position even after the data stream is reversed.

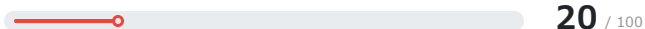
Scores

Programming Ability



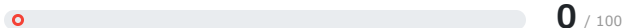
Code seems to be unrelated to the given problem.

Functional Correctness



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.

Programming Practices



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.

Final Code Submitted

Compilation Status: Pass

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 /*
6 *
7 */
8 public class Solution
9 {
10     public static int unaffectedChar(String dataStream)
11     {
12         int answer = 0;
13         // Write your code here
```

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: $O(\log N)$

*N represents size of the string.

Errors/Warnings

There are no errors in the candidate's code.


```

14  int i= 0;
15  while(i<=dataStream.length()/2)
16  {
17      char temp;
18      char first = dataStream.charAt(i);
19
20      char last = dataStream.charAt(dataStream.length()-i-1);
21      temp = first;
22      first = last;
23      last = temp;
24
25
26
27  }
28
29
30  return answer;
31  }
32
33  public static void main(String[] args)
34  {
35      Scanner in = new Scanner(System.in);
36
37      // input for dataStream
38      String dataStream = in.nextLine();
39
40      int result = unaffectedChar(dataStream);
41      System.out.print(result);
42
43  }
44  }
45

```

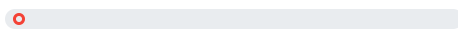
Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: **0%**

Total score

 **0/10**

0%

Basic(0/4)

0%

Advance(0/4)

0%

Edge(0/2)

Compilation Statistics

7

Total attempts

3

Successful

4

Compilation errors

0

Sample failed

0

Timed out

3

Runtime errors

Response time:

00:32:30

Average time taken between two compile attempts:

00:04:39

Average test case pass percentage per compile:

0%

i 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.

i 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 11)

A prime number is divisible only by 1 and itself. The teacher writes a positive integer on the board. Write an algorithm to find all the prime numbers from 2 to the given positive number.

Scores

Programming Ability

0 / 100

Programming ability score cannot be generated. This is because source code has syntax/ runtime errors and is unparseable.

Functional Correctness

0 / 100

Syntactically incorrect code. The source code has syntax errors in it.

Final Code Submitted

Compilation Status: Fail

Code Analysis

```

1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 /*
6  * num, representing the number written on the board.
7  */
8 public class Solution
9 {
10     public static int[] calculatePrimeNumbers(int num)
11     {
12         int[] answer = new int[100];
13         // Write your code here
14         for(int i= 2;i<num;i++)
15         {
16             if(%i != 0)
17             {
18                 answer[i] = num;
19             }
20         }
21
22
23         return answer;
24     }
25
26     public static void main(String[] args)
27     {
28         Scanner in = new Scanner(System.in);
29         // input for num
30         int num = in.nextInt();
31
32
33         int[] result = calculatePrimeNumbers(num);
34         for(int idx = 0; idx < result.length - 1; idx++)
35         {
36             System.out.print(result[idx] + " ");
37         }
38         System.out.print(result[result.length - 1]);
39     }
40 }
41

```

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 upto which primes are to be printed

Errors/Warnings

Compiling failed with exitcode 1, compiler output:
source_777.java:17: error: illegal start of expression
if(%i != 0)
^
1 error

Compilation Statistics

3

Total attempts

3

Successful

0

Compilation errors

0

Sample failed

0

Timed out

2

Runtime errors

Response time:

00:12:05

Average time taken between two compile attempts:

00:04:02

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

Automata Fix



29 / 100

[Code Replay](#)

Question 1 (Language: Java)

The function/method ***allExponent*** returns a real number representing the result of exponentiation of base raised to power exponent for all input values. The function/method ***allExponent*** accepts two arguments - *baseValue*, an integer representing the base and *exponentValue*, an integer representing the exponent.

The incomplete code in the function/method ***allExponent*** works only for positive values of the exponent. You must complete the code and make it work for negative values of exponent as well.

Another function/method ***positiveExponent*** uses an efficient way for exponentiation but accepts only positive *exponent* values. You are supposed to use this function/method to complete the code in ***allExponent*** function/method.

Helper Description

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

```
public class Exponent
{
    public int base;

    public int exponent;

    int positiveExponent()
    {
        /*It calculate the Exponent for positive value of exponentValue

        This can be called as -

        Exponent exp = new Exponent(baseValue, exponentValue);

        float res = exp.positiveExponent();*/
    }
}
```

Scores

Final Code Submitted

Compilation Status: Pass

```
1 // You can print the values to stdout for debugging
2 class Solution
3 {
4     float allExponent(int baseValue, int exponentValue)
5     {
6         float res = 1;
7         if(exponentValue >=0)
8         {
9             Exponent exp = new Exponent(baseValue, exponentValue);
10            res = (float)exp.positiveExponent();
11        }
12        else
13        {
14
15            // write your code here for negative exponentInput
16        }
17        return res;
18    }
```

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.

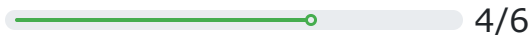
Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: **66.67%**

Total score


100%

Basic(2/2)

33%

Advance(1/3)

100%

Edge(1/1)

Compilation Statistics

0

Total attempts

0

Successful

0

Compilation errors

1

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:02:31

Average time taken between two compile attempts:

00:00:00

Average test case pass percentage per compile:

66.7%

i 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.

i 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)

You are given predefined structure **Time** containing *hour*, *minute*, and *second* as members. A collection of functions/methods for performing some common operations on times is also available. You must make use of these functions/methods to calculate and return the difference.

The function/method **difference_in_times** accepts two arguments - *time1*, and *time2*, representing two times and is supposed to return an integer representing the difference in the number of seconds.

You must complete the code so that it passes all the test cases.

.

Helper Description

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

```
public class Time
{
    public int hour;
    public int minute;
    public int second;
    public Time(int h, int m, int s)
    {
        hour = h;
        minute = m;
        second = s;
    }
    public int compareTo(Object anotherTime)
    {
        /*Return 1, if time1 is greater than time2.
        Return -1 if time1 is less than time2
        or, Return 0, if time1 is equal to time2
        This can be called as -
        * If time1 and time2 are two Time then -
        * time1.compareTo(time2) */
    }
    public void addSecond()
    {
        /* Add one second in the time;
        This can be called as -
```

```
* If time1 is Time then -

* time1.addSecond() */

}
```

Scores

Final Code Submitted

Compilation Status: Fail

```
1 // You can print the values to stdout for debugging
2 class Solution
3 {
4     int difference_in_times(Time time1, Time time2)
5     {
6         // write your code here
7     }
8 }
9
```

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

Solution.java:7: error: missing return statement
 }
 ^

Time.java:5: warning: [rawtypes] found raw type: Comparable
 public class Time implements Comparable
 ^

missing type arguments for generic class Comparable
 where T is a type-variable:
 T extends Object declared in interface Comparable
 1 error
 1 warning

Structural Vulnerabilities and Errors

There are no errors in the candidate's code.

Compilation Statistics

0

Total attempts

0

Successful

0

Compilation errors

0

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:01:02

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 3 (Language: Java)

The function/method ***replaceValues*** is modifying the input list in such a way - if the length of input list is odd, then all the elements of the input list are supposed to be replaced by 1s and in case it is even, the elements should be replaced by 0s.

For example: given the input list [0 1 2], the function will modify the input list like [1 1 1]

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

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

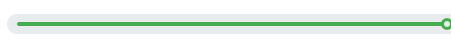
Scores

Final Code Submitted	Compilation Status: Pass	Code Analysis
<pre> 1 // You can print the values to stdout for debugging 2 class Solution 3 { 4 void replaceValues(int size, int[] inputList) 5 { 6 int i, j; 7 if(size % 2 == 0) 8 { 9 i=0; 10 while(i<size) 11 { 12 inputList[i] = 0; 13 i+=1; 14 } 15 } 16 else 17 { 18 j=0; 19 while(j<size) 20 { 21 inputList[j] = 1; 22 j+=1; 23 } 24 } 25 } 26 public static void main(String args[]) 27 { 28 int list[] = {1,2,3}; 29 int size = list.length; 30 Solution s = new Solution(); 31 s.replaceValues(size,list); 32 } 33 } 34 </pre>		<p>Average-case Time Complexity</p> <p>Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.</p> <p>Best case code:</p> <p>*N represents</p>
		<p>Errors/Warnings</p> <p>There are no errors in the candidate's code.</p>
		<p>Structural Vulnerabilites and Errors</p> <p>There are no errors in the candidate's code.</p>

Test Case Execution

Passed TC: 100%

Total score

 8/8

100%

Basic(6/6)

0%

Advance(0/0)

100%

Edge(2/2)

Compilation Statistics

2

Total attempts

2

Successful

0

Compilation errors

1

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:04:27

Average time taken between two compile attempts:

00:02:14

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 4 (Language: Java)

The function/method **maxReplace** prints space separated integers representing the input list after replacing all elements of the input list with the maximum element of the input list.

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

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

Scores

Final Code Submitted

Compilation Status: Fail

Code Analysis

```

1 // You can print the values to stdout for debugging
2 class Solution
3 {
4     void maxReplace(int size, int inputList[])
5     {
6         if(size>0)
7         {
8             int max =inputList[0];
9             for(int i=0;i<size;i++)
10            {
11                if(max<inputList[i])
12                {
13                    max = inputList[i];
14                }
15            }
16        }
17        for(int i=0;i<size;i++)
18        {
19            inputList[i]=max;
20            System.out.print(inputList[i]+" ");
21        }
22    }
23 }

```

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

Solution.java:19: error: cannot find symbol
inputList[i]=max;
^
symbol: variable max
location: class Solution
1 error

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Compilation Statistics

5

Total attempts

0

Successful

5

Compilation errors

0

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:04:48

Average time taken between two compile attempts:

00:00:58

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 5 (Language: Java)

The function/method **productMatrix** accepts three arguments - *rows*, an integer representing the rows of the matrix; *columns*, an integer representing the columns of the matrix and *matrix*, a two-dimensional array of integers, respectively.

The function/method **productMatrix** return an integer representing the product of the odd elements whose i^{th} and j^{th} index are the same. Otherwise, it returns 0.

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

Scores

Final Code Submitted

Compilation Status: Pass

```

1 public class Solution
2 {
3     public int productMatrix(int rows, int columns, int matrix[][])
4     {
5         int result=0;
6         for(int i=0;i<rows;i++)
7             for(int j=0;j<columns;j++)
8                 if((i==j) || (matrix[i][j]%2!=0))
9                     result *=matrix[i][j];
10        if(result<=1)
11            return 0;
12        else

```

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

```

13     return result;
14 }
15 }

```

There are no errors in the candidate's code.

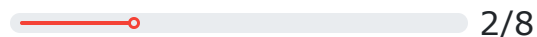
Structural Vulnerabilities and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: 25%

Total score



0%

Basic(0/5)

100%

Advance(2/2)

0%

Edge(0/1)

Compilation Statistics

0

Total attempts

0

Successful

0

Compilation errors

1

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:00:16

Average time taken between two compile attempts:

00:00:00

Average test case pass percentage per compile:

25%

i 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.

i 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: Java)

The function/method **sortArray** modify the input list by sorting its elements in descending order.

The function/method **sortArray** accepts two arguments - *len*, representing the length of the list and *arr*, a list of

integers representing the input list, respectively.

The function/method **sortArray** compiles successfully but fails to get 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 class Solution
3 {
4     public void sortArray(int len, int[] arr)
5     {
6         int i , max , location , j , temp;
7         for( i = 0 ; i < len ; i ++ )
8         {
9             max = arr[i];
10            location = i;
11            for( j = i ; j < len ; j ++ )
12            {
13                if( max < arr[j] )
14                {
15                    max = arr[j];
16                    location = j;
17                }
18            }
19            temp = arr[i];
20            arr[i] = arr[location];
21            arr[location] = temp;
22        }
23    }
24 }
```

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.

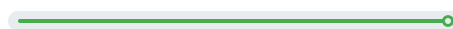
Structural Vulnerabilities and Errors

There are no errors in the candidate's code.

Test Case Execution

Passed TC: 100%

Total score

 8/8

100%

Basic(3/3)

100%

Advance(4/4)

100%

Edge(1/1)

Compilation Statistics

1

Total attempts

1

Successful

0

Compilation errors

0

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:00:29

Average time taken between two compile attempts:

00:00:29

Average test case pass percentage per compile:

100%

i 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.

i 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 ***calculateMatrixSum*** returns an integer representing the sum of odd elements of the given matrix whose i^{th} and j^{th} index are the same.

The function/method ***calculateMatrixSum*** accepts three arguments - *rows*, an integer representing the number of rows of the given matrix, *columns*, an integer representing the number of columns of the given matrix and *matrix*, representing a two-dimensional array of integers.

The function/method ***calculateMatrixSum*** 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
  t for debugging
2 class Solution
3 {
4     int calculateMatrixSum(int rows, int columns, int matrix[][])
5     {
6         int i, j, sum=0;
7         if((rows>0) && (columns>0))
8         {
9             for(i=0;i<rows;i++)
10            {
11                sum =0;
12                for(j=0;j<columns;j++)
13                {
14                    if(i==j)
15                    {
16                        if(matrix[i][j]/2!=0)
17                            sum += matrix[i][j];
18                    }
19                }
20            }
21            return sum;
22        }
23        else
24            return sum;
25    }
26 }

```

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: 12.5%

Total score

1/8

0%

Basic(0/4)

0%

Advance(0/3)

100%

Edge(1/1)

Compilation Statistics

1

Total attempts

1

Successful

0

Compilation errors

1

Sample failed

0

Timed out

0

Runtime errors

Response time:

00:02:14

Average time taken between two compile attempts:

00:02:14

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 written english comprehension](#)



[Learn about spoken english comprehension](#)



[Test your comprehension skills](#)



Logical Ability

[Test your inductive logic!](#)



[Play Tic-Tac-Toe to develop your inductive reasoning skills](#)



[Learn about finding the next number in the series!](#)



Quantitative Ability (Advanced)

[Learn about real world mathematics](#)



[Learn about multiplication and division in the real world contexts](#)



[Learn about multiplication and division](#)



Icon Index



Free Tutorial



Paid Tutorial



Youtube Video



Web Source



Wikipedia



Text Tutorial



Video Tutorial



Google Playstore