Amit Test

Test ID: 11261015363082 | 09898898998 | t@gmail.com

Test Date: Mar 26, 2020

Automata Pro

87 /100

Automata Pro

87 / 100

Programming Ability	Programming Practices	Functional Correctness
100 / 100	50 / 100	100 / 100

Runtime Complexity Score

83 / 100

High similarity detected with another source. Please check the candidate's response

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 **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 **Proctoring** section captures the output of the different proctoring features used during the test.

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 | Response

Automata Pro Code Replay 87 / 100

Question 1 (Language: Java) High similarity detected : **100%**

An increment matrix is one whose elements are the incremented values of an initial value s.
For example, if the initial value is “s = 1” and the dimensions are “m = 3” and “n = 3,” then the increment matrix will be:
1 2 3
4 5 6
7 8 9

Write an algorithm to multiply the original increment matrix with its transpose.

Input
The input to the function/method consists of three arguments:
firstValue, a positive integer representing the initial value (s);
rows, a positive integer representing the number of rows in the increment matrix (m);
columns, a positive integer representing the number of columns in the increment matrix (n).

Output
Return a two-dimensional matrix of integers obtained from the multiplication of the increment matrix and its transpose.

Example
Input:
firstValue = 1
rows = 3
columns = 3

Output:
14 32 50
32 77 122
50 122 194

Explanation:
For *firstValue* = 1, *rows* = 3 and *columns* = 3, the increment matrix will be:
1 2 3
4 5 6
7 8 9
And its transpose matrix will be:
1 4 7
2 5 8
3 6 9
Thus, the resultant multiplication matrix will be:
14 32 50

32 77 122
50 122 194

Scores

Programming Ability

100 / 100

Completely correct. A correct implementation of the problem using the right control-structures and data dependencies.

Programming Practices

75 / 100

Low readability, high on program structure. The source code does not follow best practices in its formatting and may contain a few 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.

Runtime Complexity Score

100 / 100

Efficient complexity. The predicted complexity matches the ideal complexity with which the problem could have been solved.

Final Code Submitted

Compilation Status: Pass

```
1  public class Solution
2  {
3     public int[][] transposeMultMatrix(int s,int m, int n)
4     {
5        System.out.println("hello");
6        if(m>=0 && n>=0)
7        {
8           int i,j,k;
9           int [][] omatrix = new int[m][n];
10          int [][] tmatrix = new int[n][m];
11          int [][] mmatrix = new int[m][m];
12
13
14          for(i=0;i<m;i++)
15          {
16             for(j=0;j<n;j++)
17             {
18                omatrix[i][j]=s++;
19             }
20          }
21
22          for(i=0;i<n;i++)
23             for(j=0;j<m;j++)
24                tmatrix[i][j]=omatrix[j][i];
25             for(i=0;i<m;i++)
26             {
27                for(j=0;j<m;j++)
28                {
```

Code Analysis

Average-case Time Complexity

Candidate code: $O(N^3)$

Best case code: $O(N^3)$

*N represents dimensions of a square matrix

Test Case Execution

Total Score

100%

100%	100%	100%
Basic	Advance	Edge

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

Readability & Language Best Practices

Line 1,3: The class 'Solution' has a Cyclomatic Complexity of 11 (Highest = 10).
Line 3,8: Variables are given very short names.

```
29                  mmatrix[i][j]=0;
30                  for(k=0;k<n;k++)
31                     mmatrix[i][j]+=omatrix[i][k]*tmatrix[k][j];
32
33               }
34            }
35            return mmatrix;
36         }
37         else
38            return null;
39   }
40 }
```

Compilation Statistics

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

Response time:	**00:01:23**
Average time taken between two compile attempts:	**00:01:23**
Average test case pass percentage per compile:	**0%**

Similarity Detected

Every response is checked against:

- The responses of peers who took this assessment in the same event
- All candidate responses submitted in the last week
- Content currently available on the web

This response has a high degree of similarity to one of these sources, which means it is possibly not the candidate's original work:

The candidate's code exactly matches with the peer's code

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) High similarity detected : **100%**

John misses his bus and has to walk all the way to school. The distance between his home and the school is D units. He starts his journey with an initial energy of K units. His energy decreases by 1 unit for each unit of distance he walks. On the way to school, there are N juice stalls. Each stall possesses a specific amount of juice in liters. His energy increases by 1 unit for every liter of juice he consumes. Note that in order for him to keep walking he must have non-zero energy.

Write an algorithm to help John determine the minimum number of juice stalls at which he must stop in order to reach his school. In the case that he cannot reach the school, the output will be -1.

Input
The input to the function/method consists of five arguments:
numOfStalls, an integer representing the number of juice stalls (N);
distOfStalls, a list of integers representing the distance of stalls from John's home;
juiceQuantity, a list of integers representing the quantity of juice available at each juice stall;
distance, an integer representing the distance between John's home and the school (D);
initialEnergy, an integer representing John's initial energy (K).

Output
Return an integer representing the minimum number of juice stalls at which John must stop in order to reach his school.

Constraints
$1 \leq numOfStalls \leq 10^4$
$1 \leq distOfStalls[i] < distance \leq 10^5$
$1 \leq juiceQuantity[i] \leq 1000$
$0 \leq initialEnergy \leq 10^5$
$0 \leq i < numOfStalls$

Example
Input:
numOfStalls = 4
distOfStalls = [5, 7, 8, 10]
juiceQuantity = [2, 3, 1, 5]
distance = 15
initialEnergy = 5

Output:
3

Explanation:
John's initial energy is 5 units, with which he can reach the first stall. At this point, his energy is zero units. He can get 2 units of energy from this stall.
With 2 units of energy, he can move a further 2 units of distance to reach the second stall.
Here his energy is zero again. He can get 3 units of energy from this stall.
He will skip the third stall as he has enough energy to reach the fourth stall. The energy available at the fourth stall is enough to reach the school.
So, the number of stalls visited by John is 3.

Scores

Programming Ability

100 / 100

Completely correct. A correct implementation of the problem using the right control-structures and data dependencies.

Programming Practices

25 / 100

Low readability, low on program structure. The source code is poorly formatted and contains 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.

Runtime Complexity Score

67 / 100

One-away from efficient complexity. The source code implements an algorithm that is not as efficient as the ideal complexity but is one complexity measure away from it.

Final Code Submitted **Compilation Status: Pass**

```
1  import java.util.AbstractMap;
2  import java.util.Comparator;
3  import java.util.LinkedList;
4  import java.util.Map;
5  import java.util.Map.Entry;
6  import java.util.PriorityQueue;
7  import java.util.Queue;
8  import java.util.Scanner;
9  import java.util.Set;
10 import java.util.TreeSet;
```

Code Analysis

Average-case Time Complexity

Candidate code: O(N logN)

Best case code: O(N)

*N represents number of juice stalls

Test Case Execution

```
11
12 class Solution
13 {
14 class Ss{
15    int[] queue_array = new int[500];
16    int rear;
17    int front;
18 };
19    void SsPush(Ss q, int data)
20 {
21    //printf("Enter push for %d",data);
22    if ((q.front == 0) && (q.rear == 0))
23    {
24       q.queue_array[0] = 99999;
25       q.front++;
26       q.rear++;
27       q.queue_array[q.rear] = data;
28       return;
29    }
30    else
31       check(q, data);
32
33    //printf("Rear is %d and front is %d\n",q.rear,q.front);
34    // for(int i=1;i<q.rear;i++)
35    // printf("last queue element at %d = %d\n",i,q.queue_array[i]);
36 }
37
38 /* Function to check priority and place element */
39 void check(Ss q, int data)
40 {
41    int i,j;
42
43    for (i = q.front; i <= q.rear; i++)
44    {
45
46       if (data >= q.queue_array[i])
47       {
48          for (j = q.rear + 1; j > i; j--)
49          {
50             q.queue_array[j] = q.queue_array[j - 1];
51          }
52          q.queue_array[i] = data;
53          q.rear++;
54          return;
55       }
56    }
57    q.queue_array[i] = data;
58    q.rear++;
59 }
60 boolean SsEmpty(Ss q){
```

Total Score

100%

100%	100%	100%
Basic	Advance	Edge

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

Readability & Language Best Practices

Line 19,39,41...: Variables are given very short names.
Line 109: variable starting with uppercase.

Performance & Correctness

Line 15,16,17...: Variable 'queue_array' must be private and have accessor methods.
Line 1,2,3...: Avoid unused imports such as 'java.util.AbstractMap'

```
61    if (q.front == 0 || q.front > q.rear)
62       return true;
63    else
64       return false;
65 }
66 void SsPop(Ss q)
67 {
68    if (q.front == 0 || q.front > q.rear)
69    {
70       //printf("Queue Underflow \n");
71       return ;
72    }
73    else
74    {
75       //printf("Element deleted from queue is : %d\n", queue_array[q.
   front]);
76       q.front++;
77    }
78 } /*End of delete() */
79 int SsTop(Ss q)
80 {
81    //printf("Front is %d \n",q.front);
82    int top = q.queue_array[q.front];
83    //q.front++;
84    return top;
85 }
86 void SsInitialize(Ss q){
87    q.front = 0;
88    q.rear = 0;
89
90 }
91 void pairSort(int[][] number,int n){
92    int a,b,i,j;
93    for (i = 0; i < n; ++i)
94    {
95       for (j = i + 1; j < n; ++j)
96       {
97          if (number[i][0] > number[j][0])
98          {
99             a =  number[i][0];
100            b = number[i][1];
101            number[i][0] = number[j][0];
102            number[i][1] = number[j][1];
103            number[j][1] = b;
104            number[j][0] = a;
105         }
106      }
107   }
108 }
109 int findMinNumberOfJuiceStalls(int n, int dist[], int juice[], int D, int
```

```
K)
{
   int i;
   Ss pq = new Ss();
SsInitialize(pq);
int[][] v = new int[n][2];
   for(i=0 ; i<n ; i++){
      v[i][0] = dist[i];
      v[i][1] = juice[i];
   }
   pairSort(v, n);

   int dis,pdis;
   dis = D;
   pdis = K;

   int ans=0;

      for(i=0;i<n;i++)
      {
         //printf("answer is %d",ans);
         if(pdis>=dis)return ans;
         if(pdis>=v[i][0]){
         //printf("v[%d][1] is %d \n",i,v[i][1]);
            SsPush(pq,v[i][1]);
         }
         else
         {
            while(pdis < v[i][0])
            {

               if(SsEmpty(pq))return -1;
               int val=SsTop(pq);
               //printf("Top at %d is %d\n",i,val);
               SsPop(pq);
               pdis+=val;
               ans++;
            }
            SsPush(pq, v[i][1]);
         }
      }
      while(pdis < dis)
      {
         if(SsEmpty(pq))return -1;
         int val=SsTop(pq);
         SsPop(pq);
         pdis+=val;
         ans++;
      }
      return ans;
```

```
159 }
160
161 }
```

Compilation Statistics

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

Response time:	**00:00:56**
Average time taken between two compile attempts:	**00:00:56**
Average test case pass percentage per compile:	**100%**

Similarity Detected

Every response is checked against:

- The responses of peers who took this assessment in the same event
- All candidate responses submitted in the last week
- Content currently available on the web

This response has a high degree of similarity to one of these sources, which means it is possibly not the candidate's original work:

The candidate's code exactly matches with the peer's code

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

3 | Proctoring