Pre-requisite Strings Online Assessment 1 ... > abhishekyadav88846@gmail.com

You can view this report online at : https://www.hackerrank.com/x/tests/746399/candidates/13646206/report

Full Name:	Abhishek Yadav
Email:	abhishekyadav88846@gmail.com
Test Name:	**Pre-requisite Strings Online Assessment 1 (Coding & Problem solving)**
Taken On:	20 Mar 2020 14:57:36 IST
Time Taken:	48 min 7 sec/ 150 min
Work Experience:	< 1 years
City:	Gharuan, Punjab, India
Student Roll Number:	17BCS1612
Personal Email Address:	abhishekyadav88846@gmail.com
CGPA:	7.98
Contact Number:	08732837125
Stream/Branch:	CSE
Section:	K2
Resume:	https://cdn.hackerrank.com/files/uploads/recruit-resumes/a2ef0270-f606-4737-bf61-8dec1941d4f9/Resume Updated.pdf
Invited by:	Jagandeep
Tags Score:	Algorithms 175/175 Arrays 50/50 Core CS 75/75 Data Structures 100/100 Easy 150/150 Hashing 50/50 Implementation 50/50 Medium 75/75 Problem Solving 125/125 Sets 50/50 Strings 225/225

100%

225/225

scored in **Pre-requisite Strings Online Assessment 1 (Coding & Problem solving)** in 48 min 7 sec on 20 Mar 2020 14:57:36 IST

Recruiter/Team Comments:

No Comments.

Plagiarism flagged

We have marked questions with suspected plagiarism below. Please review.

	Question Description	Time Taken	Score	Status
Q1	Fewest Coins › Coding	9 min 15 sec	50/ 50	✓
Q2	String Reduction › Coding	21 min 12 sec	50/ 50	✓
Q3	Balancing Parentheses › Coding	5 min 57 sec	50/ 50	✓
Q4	Good Binary Strings › Coding	11 min 34 sec	75/ 75	!

QUESTION 1

✓

Correct Answer

Score 50

Fewest Coins › Coding

Algorithms | Strings | Data Structures | Problem Solving | Easy | Sets | Hashing

QUESTION DESCRIPTION

An online coin dealer offers bags of coins that are guaranteed to contain at least one full set. Given a string comprised of lowercase letters in the range *ascii[a-z]*, where each letter represents a coin type, determine the length of the shortest substring that contains at least one of each type of coin.

Example:
coins = dabbcabcd

The list of all characters in the string is *[a, b, c, d].*
Two of the substrings that contain all letters are *dabbc* and *abcd.*
The shortest substring that contains all of the letters is *4* characters long.

Function Description

Complete the function *fewestCoins* in the editor below.

fewestCoins has the following parameter:
string coins: a string

Return

int: the length of the shortest substring that contains at least one of each characters in *coins*

Constraints

- $1 \leq \text{size of coins} \leq 10^5$
- each *coins[i]* is in the set *ascii[a-z]*

▼ Input Format For Custom Testing

The first line contains a string, *coins*.

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN       Function
-----       --------
bab    →    coins = 'bab'
```

Sample Output

```
2
```

Explanation

"ba" is a substring that contains all the characters in *coins*.

▼ Sample Case 1

Sample Input For Custom Testing

```
STDIN                       Function
-----                       --------
asdfkjeghfalawefhaef →      coins = 'asdfkjeghfalawefhaef'
```

Sample Output

```
13
```

Explanation

The *11* distinct characters in *coins* are *[a, d, e, f, g, h, j, k, l, s, w]*. The shortest substring with all of the characters is 13 characters long: *sdfkjeghfalaw*.

CANDIDATE ANSWER

Language used: **C++14**

```cpp
/*
 * Complete the 'fewestCoins' function below.
 *
 * The function is expected to return an INTEGER.
 * The function accepts STRING coins as parameter.
 */

int fewestCoins(string s) {
  vector<int> ar(26);
  for(char c: s) ar[c - 'a'] = 1;
  int n = (int) s.length();
  vector<vector<int>> p(n, vector<int>(26, 0));
  int l = 0, r = n;
  int res = -1;
  auto can = [&](int x) {
    vector<int> here(26);
    for(int i = 0; i < n; ++i) {
      here[s[i] - 'a']++;
      if(i >= x) here[s[i - x] - 'a']--;
      bool cn = true;
      for(int j = 0; j < 26; ++j) {
        cn &= here[j] >= ar[j];
      }
      if(cn) return true;
    }
    return false;
  };
  while(l <= r) {
    int m = (l + r) >> 1;
    if(can(m)) {
      res = m;
      r = m - 1;
    } else {
      l = m + 1;
    }
  }
  return res;
}



```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED

TestCase 0	Easy	Sample case	✓ Success	1	0.1257 sec	8.88 KB
TestCase 1	Easy	Sample case	✓ Success	1	0.145 sec	8.95 KB
TestCase 2	Easy	Sample case	✓ Success	1	0.1369 sec	8.86 KB
TestCase 3	Easy	Sample case	✓ Success	3	0.1398 sec	9.71 KB
TestCase 4	Easy	Sample case	✓ Success	3	0.1325 sec	9.73 KB
TestCase 5	Easy	Hidden case	✓ Success	3	0.11 sec	10.5 KB
TestCase 6	Medium	Hidden case	✓ Success	4	0.11 sec	10.5 KB
TestCase 7	Medium	Hidden case	✓ Success	4	0.1789 sec	11.1 KB
TestCase 8	Medium	Hidden case	✓ Success	5	0.1502 sec	11.8 KB
TestCase 9	Hard	Hidden case	✓ Success	5	0.2186 sec	11.9 KB
TestCase 10	Hard	Hidden case	✓ Success	5	0.1068 sec	12.1 KB
TestCase 11	Hard	Hidden case	✓ Success	5	0.1136 sec	12.6 KB
TestCase 12	Hard	Hidden case	✓ Success	10	0.162 sec	21.2 KB

No Comments

QUESTION 2

✓
Correct Answer

Score 50

String Reduction › Coding

Easy Strings

QUESTION DESCRIPTION

Given a string, reduce it in such a way that all of its substrings are distinct. To do so, you may delete any character of the string at any index. What is the minimum number of deletions needed in order to complete this task?

Note: A substring is a contiguous sequence of characters within a string. It can be formed by deleting some (0 or more) characters from the left of the string and some (0 or more) characters from the right of the string.

For example, let's say the given string is *s = "abab"*. Currently, the substrings are not distinct—the substring "ab" is found starting at both index 0 and index 2. By deleting s[2] and s[3], the string becomes "ab", where all substrings are distinct. Therefore, the answer is 2 because this required 2 deletions. (Note that "aba" is not acceptable because the character 'a' counts as a substring. In "aba", there are two instances of the substring "a".)

Function Description
Complete the function *getMinDeletions* in the editor below.

getMinDeletions has the following parameter(s):
string *s:* the given string
Returns:
int: the minimum number of deletions needed to make *s* have only distinct substrings in it

Constraints

- $1 \le n \le 10^5$

▼ Input Format For Custom Testing

The first line contains a string, *s*.

▼ Sample Case 0

Sample Input For Custom Testing

Sample Input For Custom Testing

```
STDIN     Function
-----     --------
abcab  => s = "abcab"
```

Sample Output

```
2
```

Explanation

By deleting the first 2 characters, the string becomes "cab", which has only distinct substrings in it. Therefore, the answer is 2.

▼ Sample Case 1

Sample Input For Custom Testing

```
abcabc
```

Sample Output

```
3
```

Explanation

By deleting the characters at indices 0, 4, and 5, the string becomes "bca", which has only distinct substrings in it. Because this required 3 deletions, the answer is 3.

INTERNAL NOTES

For distinct sub-strings, all the characters in the string must be distinct. So we calculate the number of distinct characters and hence our answer equals *n - distinct_characters*.

```
int getMin(string s) {
    int ans = 0;

    int freq[26] = {0};
    for (int i = 0; s[i]; i++) {
        if (freq[ s[i] - 'a' ] == 0) {
            freq[ s[i] - 'a' ] = 1;
            ans++;
        }
    }
    ans = s.length() - ans;

    return ans;
}
```

Tester's solution:

```
def getMin(s):

    diff_chr = set()

    for i in range(len(s)):
        diff_chr.add(s[i])

    return (len(s) - len(diff_chr))
```

CANDIDATE ANSWER

Language used: **C++14**

```
/*
 * Complete the 'getMinDeletions' function below.
 *
 * The function is expected to return an INTEGER.
 * The function accepts STRING s as parameter.
 */

int getMinDeletions(string s) {
  int n = (int) s.length();
  sort(s.begin(), s.end());
  s.erase(unique(s.begin(), s.end()), s.end());
  return n - (int) s.size();

}


```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
TestCase 0	Easy	Sample case	✓ Success	1	0.1001 sec	9.09 KB
TestCase 1	Easy	Sample case	✓ Success	1	0.1305 sec	8.98 KB
TestCase 2	Easy	Sample case	✓ Success	1	0.2176 sec	8.92 KB
TestCase 3	Easy	Hidden case	✓ Success	2	0.1086 sec	8.83 KB
TestCase 4	Easy	Hidden case	✓ Success	2	0.1207 sec	8.89 KB
TestCase 5	Easy	Hidden case	✓ Success	3	0.109 sec	9.06 KB
TestCase 6	Easy	Hidden case	✓ Success	3	0.1174 sec	9 KB
TestCase 7	Easy	Hidden case	✓ Success	3	0.1256 sec	9.1 KB
TestCase 8	Easy	Hidden case	✓ Success	3	0.0987 sec	8.96 KB
TestCase 9	Easy	Hidden case	✓ Success	3	0.1189 sec	8.96 KB
TestCase 10	Easy	Hidden case	✓ Success	5	0.1125 sec	9.08 KB
TestCase 11	Easy	Hidden case	✓ Success	5	0.1639 sec	9.11 KB
TestCase 12	Easy	Hidden case	✓ Success	6	0.1953 sec	9 KB
TestCase 13	Easy	Hidden case	✓ Success	6	0.1436 sec	9.23 KB
TestCase 14	Easy	Hidden case	✓ Success	6	0.1389 sec	9.18 KB

No Comments

QUESTION 3

✓ Correct Answer

Score 50

Balancing Parentheses › Coding

Easy | Implementation | Data Structures | Strings | Arrays | Algorithms

QUESTION DESCRIPTION

Given a string that consists of only two types of characters: '(' and ')', balance the parentheses by inserting either a '(' or a ')' as many times as necessary. Determine the minimum number of characters that must be inserted.

Example

s = '(()))'

To make it a valid sequence, insert a '('at the beginning of the string, resulting in "((()))". The string is balanced after *1* insertion.

Function Description

Complete the function *getMinOperations* in the editor below. The function must return the minimum number of operations needed to make the parentheses sequence valid.

getMinOperations has the following parameter(s):

string s: a string of parentheses

Return

int: the minimum number of insertions required to balance the parentheses

Constraints

- 1 ≤ length of *s* ≤ 10^5

▼ Input Format For Custom Testing

The first line contains a string, *s*, the initial parentheses sequence.

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN    Function
-----    --------
()))     →    s = '()))'
```

Sample Output

```
2
```

Explanation

Insert a '(' *2* times at the beginning of the string to make it valid: "((()))".

▼ Sample Case 1

Sample Input For Custom Testing

```
STDIN     Function
-----     --------
()()   →  s = '()()'
```

Sample Output

```
0
```

Explanation

The sequence is already valid, so no insertions are needed.

INTERNAL NOTES

We iterate through the string and maintain a balance of the parenthesis and store the minimum amount it touches throughout the string. That minimum value is the number of '**(**' we add in the beginning of the string and then the balance of the string at the end is added.

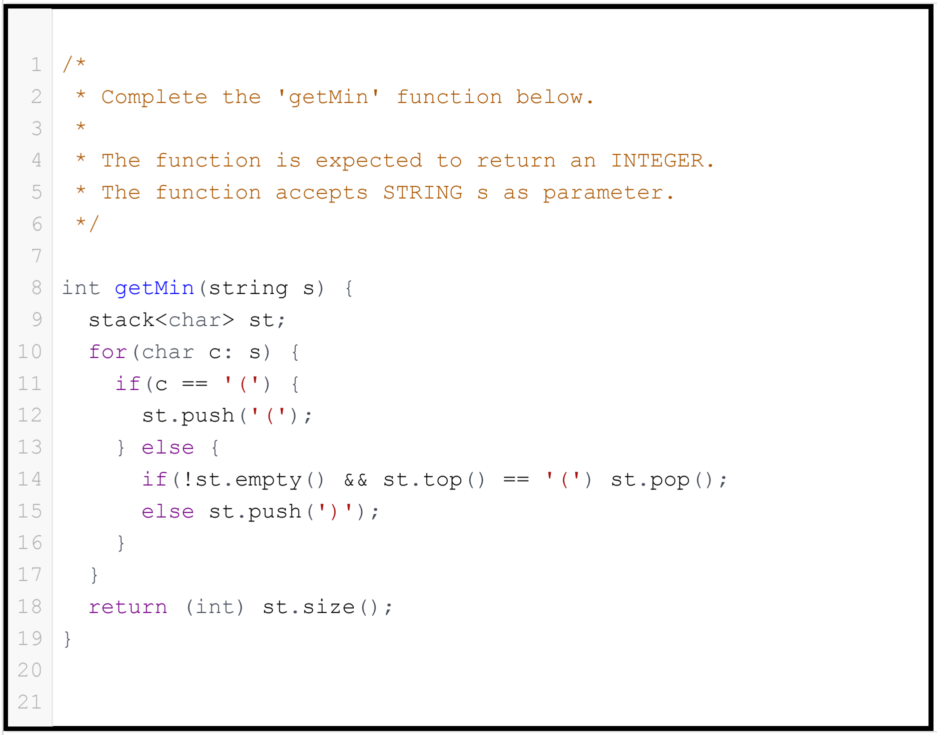
```
int getMin(string s) {
    int ans = 0;

    int bal = 0, mi = 0;
    for (int i = 0; s[i]; i++) {
        if (s[i] == '(') {
            bal++;
        }
        else {
            bal--;
        }
        mi = min(mi, bal);
        mi = min(mi, bal);
    }
    ans = -mi + (bal - mi);

    return ans;
}
```

Tester's code:

```
def getMin(s):
    n = len(s)
    assert 1 <= n <= 10 ** 5
    for i in s:
        assert i == '(' or i == ')'
    st = []
    for i in s:
        if i == '(':
            st.append(0)
        else:
            if len(st) > 0 and st[-1] == 0:
                del st[-1]
            else:
                st.append(1)
    # return len(s)
    return len(st)
```

CANDIDATE ANSWER

Language used: **C++14**



```
/*
 * Complete the 'getMin' function below.
 *
 * The function is expected to return an INTEGER.
 * The function accepts STRING s as parameter.
 */

int getMin(string s) {
  stack<char> st;
  for(char c: s) {
    if(c == '(') {
      st.push('(');
    } else {
      if(!st.empty() && st.top() == '(') st.pop();
      else st.push(')');
    }
  }
  return (int) st.size();
}


```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
TestCase 0	Easy	Sample case	✓ Success	1	0.1511 sec	9.1 KB
TestCase 1	Easy	Sample case	✓ Success	1	0.1039 sec	8.86 KB
TestCase 2	Easy	Sample case	✓ Success	1	0.1205 sec	8.77 KB
TestCase 3	Easy	Hidden case	✓ Success	2	0.1046 sec	8.83 KB

TestCase 4	Easy	Sample case	✓ Success	2	0.1308 sec	9.02 KB
TestCase 5	Easy	Sample case	✓ Success	3	0.1014 sec	9.04 KB
TestCase 6	Easy	Hidden case	✓ Success	3	0.0998 sec	9.07 KB
TestCase 7	Easy	Hidden case	✓ Success	3	0.1169 sec	8.88 KB
TestCase 8	Easy	Hidden case	✓ Success	3	0.1244 sec	9.05 KB
TestCase 9	Easy	Hidden case	✓ Success	3	0.1213 sec	9.14 KB
TestCase 10	Easy	Hidden case	✓ Success	5	0.1302 sec	9.13 KB
TestCase 11	Easy	Hidden case	✓ Success	5	0.1382 sec	8.92 KB
TestCase 12	Easy	Hidden case	✓ Success	6	0.143 sec	9.05 KB
TestCase 13	Easy	Hidden case	✓ Success	6	0.1072 sec	9.24 KB
TestCase 14	Easy	Hidden case	✓ Success	6	0.1178 sec	9.27 KB

No Comments

QUESTION 4

Needs Review

Score 75

Good Binary Strings › Coding

Strings | Medium | Algorithms | Problem Solving | Core CS

QUESTION DESCRIPTION

We define the following:

- A *binary string* is a string consisting only of *0*'s and/or *1*'s. For example, *01011*, *1111*, and *00* are all binary strings.
- The *prefix* of a string is any substring of the string that includes the beginning of the string. For example, the prefixes of *11010* are *1*, *11*, *110*, *1101*, and *11010*.

We consider a non-empty binary string to be *good* if the following two conditions are true:

1. The number of *0*'s is equal to the number of *1*'s.
2. For every prefix of the binary string, the number of *1*'s should not be less than the number of *0*'s.

For example, *11010* is not good because it doesn't have an equal number of *0*'s and *1*'s, but *110100* is good because it satisfies both of the above conditions.

A good string can contain multiple good substrings. If two *consecutive substrings* are good, then we can *swap* the substrings as long as the resulting string is still a good string. Given a good binary string, *binString*, perform zero or more swap operations on its consecutive good substrings such that the resulting string is as lexicographically large as possible. Two substrings are considered to be consecutive if the last character of the first substring occurs exactly one index before the first character of the second substring.

For example, if we look at the good binary string *binString* = *1010111000*, we see two good binary substrings, *1010* and *111000* among others. If we swap these two substrings we get a larger value: *1110001010*. This is the largest possible good substring that can be formed.

Function Description

Complete the function *largestGood* in the editor below. The function must return a string denoting the lexicographically largest possible good string that can be formed by performing zero or more swap operations on consecutive good substrings of *binString*.

largestGood has the following parameter(s):

binString: a string

Constraints

- Each character of *binString* ∈ {01}.

- Each character of *binString* ∈ {0,1}.
- *1 ≤ |binString| ≤ 50*
- *binString* is a good string.

▼ Input Format For Custom Testing

The only line of input contains the string *binString*.

▼ Sample Case 0

Sample Input 0

```
STDIN           Function Parameters
-----           -------------------
11011000  →     binString = "11011000"
```

Sample Output 0

```
11100100
```

Explanation 0

Given the good string *binString* = *11011000*, we can choose two consecutive good substrings, *10* and *1100*, to swap such that the resultant string, *str* = *11100100*, is the lexicographically largest good string possible.

▼ Sample Case 1

Sample Input 1

```
STDIN       Function Parameters
-----       -------------------
1100  →     binString = "1100"
```

Sample Output 1

```
1100
```

Explanation 1

The only good substring of *binString* is *1100*. So none of the operations can be applied on the string.

▼ Sample Case 2

Sample Input For Custom Testing

```
STDIN           Function Parameters
-----           -------------------
1101001100 →    binString = "1101001100"
```

Sample Output

```
1101001100
```

Explanation

The only consecutive good substrings of *binString* are *110100* and *1100.* Note that *100* is not a good substring because it contains more zeroes than ones. If we were to swap them, it would result in a lexicographically smaller string. Thus, *binString* is already the lexicographically largest good string that can be formed.

CANDIDATE ANSWER

Language used: **C++14**

```
/*
 * Complete the 'largestMagical' function below.
 *
 * The function is expected to return a STRING.
 * The function accepts STRING binString as parameter.
 */
```

```
string largestMagical(string S) {
  if (S.size()==2) return S;

        vector<string>strs;

        for (int i=0; i<S.size(); i++)
        {
            int i0=i;
            int count=0;
            while (i<S.size())
            {
                if (S[i]=='1')
                    count++;
                else
                    count--;
                if (count==0)
                    break;
                i++;
            }
            strs.push_back("1"+largestMagical(S.substr(i0+1,i-i0-1))+"0");
        }

        sort(strs.begin(),strs.end(),greater<string>());
        string result;
        for (auto a:strs) result+=a;
        return result;
}
```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
TestCase 0	Easy	Sample case	✓ Success	1	0.1015 sec	8.95 KB
TestCase 1	Easy	Sample case	✓ Success	1	0.1089 sec	9 KB
TestCase 2	Easy	Sample case	✓ Success	1	0.1032 sec	9.12 KB
TestCase 3	Easy	Hidden case	✓ Success	3	0.1112 sec	8.93 KB
TestCase 4	Easy	Hidden case	✓ Success	3	0.1281 sec	8.77 KB
TestCase 5	Easy	Hidden case	✓ Success	3	0.1093 sec	8.95 KB
TestCase 6	Medium	Hidden case	✓ Success	6	0.133 sec	8.97 KB
TestCase 7	Medium	Hidden case	✓ Success	6	0.1069 sec	9 KB
TestCase 8	Medium	Hidden case	✓ Success	6	0.1083 sec	8.94 KB
TestCase 9	Hard	Hidden case	✓ Success	10	0.1351 sec	8.84 KB
TestCase 10	Hard	Hidden case	✓ Success	10	0.1145 sec	9.02 KB
Testcase 11	Hard	Hidden case	✓ Success	10	0.1187 sec	8.96 KB
Testcase 12	Hard	Hidden case	✓ Success	15	0.1343 sec	8.86 KB

No Comments

PDF generated at: 20 Mar 2020 10:16:55 UTC