Find the maximum depth of nested parentheses in a string

Posted on July 11, 2019 | by Prashant Yadav

Posted in Algorithms, Stack, String | Tagged Easy

An algorithm to find the maximum depth of nested parentheses in a string using javascript.

We will implement an algorithm which will find the maximum depth of nested parentheses in a string and return the count. If the parentheses are not balanced then return `-1`.

Example

```
Input:
'( p((q)) ((s)t) )'
'b) (c) ()'

Output:
3
-1
```

In the first input, As `q` is surrounded by `3` balanced parentheses and it is the highest in the string we return `3`.

In the second input the parentheses are not balanced so we return `-1`.

We can solve this problem with two different methods.

1)Brute Force Approach.

2)With stack.

Practically prepare for your JavaScript interview

JavaScript Revision
JavaScript-Concept Based Problems
Data Structures
Algorithms
Machine Coding
Web Fundamentals

Using Brute Force Approach to find the maximum depth of nested parentheses

Implementation

- We will use two variables to keep track of the current maximum depth and the total maximum depth of the nested parentheses.
- Then we will iterate the string and check if the current character is `'('` then increase the current max count and check if the current max is greater than total max than update it too.
- Else if the current character is `')'` then check if current max is greater than `0` or not and reduce the count if it is greater else the parentheses are unbalanced and return `-1`.
- After the loop again check if current max is `0` or not to make sure that parentheses are properly balanced. If it is not then return `-1` else return total maximum count.

```
let maximumDepth = (str) => {
  //Keep track of the current max and total max
  let max = 0;
  let total_max = 0;

  for(let i = 0; i < str.length; i++){

    if(str[i] == '('){
      max++;

      //If current max is greater than total max then update
      if(max > total_max){
        total_max = max;
      }

    }else if(str[i] == ')'){

      //Check for balanced parentheses
      if(max > 0){
        max--;
      }else{
        return -1;
      }

    }
  }

  //Again check for balanced parentheses
  if(max != 0){
    return -1;
  }

  //Return total
  return total_max;
}
```

```
Input:
console.log(maximumDepth('( a(b) (c) (d(e(f)g)h) I (j(k)l)m)'));
console.log(maximumDepth('( p((q)) ((s)t) )'));
console.log(maximumDepth(' '));
console.log(maximumDepth('b) (c) ()'));

Output:
4
3
0
-1
```

Time complexity: O(n).

Space complexity: O(1).

Time and Space complexity

- We are iterating the whole string, so Time complexity is O(n).
- We are using constant space, so Space complexity is O(1).

This method is more efficient than the below method because we only need O(1) space. Where as with the stack we will need O(n) space.

Using Stack

Implementation using stack to find the maximum depth of balanced parentheses

- We will use the same approach as we do to check the balanced parentheses in a string with two extra variables to keep track of the current max and total maximum parentheses.

```
let maximumDepthWithStack = (str) => {

  //Keep track of the current max and total max
  let max = 0;
  let total_max = 0;
  let stack = [];

  for(let i = 0; i < str.length; i++){

    if(str[i] == '('){
      max++;

      //Push '(' in stack
      stack.push('(');

      //If current max is greater than total max then update
      if(max > total_max){
        total_max = max;
      }

    }else if(str[i] == ')'){

      //Check for balanced parentheses
      let open = stack.pop();

      if(max > 0 && open == '('){
        max--;
      }else{
        return -1;
      }

    }
  }

  //Again check for balanced parentheses
  if(stack.length != 0){
    return -1;
  }

  //Return total
  return total_max;
}
```

```
Input:
console.log(maximumDepthWithStack('( a(b) (c) (d(e(f)g)h) I (j(k)l)m)'));
console.log(maximumDepthWithStack('( p((q)) ((s)t) )'));
console.log(maximumDepthWithStack(' '));
console.log(maximumDepthWithStack('b) (c) ()'));

Output:
4
3
0
-1
```

Time complexity: O(n).

Space complexity: O(n).

Recommended Posts:

Recursive Insertion Sort Algorithm

Find the longest common prefix

6 ways to convert string to a number in javascript

Set matrix zeroes

Top view of a binary tree

Find inorder successor of a given key in a BST.

Implement stack with max and min function

Find missing alphabets to make a string panagram

Find the maximum sum of products of two arrays.

3 sum problem algorithm

Prev Next

Handcrafted with ♥somewhere in **Mumbai**