Computational Complexity

Table of Contents

Introduction

The Time and Space complexity is very crucial in determining the effectiveness of an algorithm. Declaring functions and solving problems in programming can be achieved in various ways. Efficiency is vital in programming because space and time is considered as money in the real world. These factor plays a major impact in programming. **Space complexity** of an algorithm basically quantifies the memory and amount of space that a programmer requires to successfully implement an algorithm with a function that has an input of specified length. **Time complexity** of an algorithm refers to the amount of time taken to run a function.

Different types of time complexities

- Linear Time O(n)
- Constant time O(1)
- Logarithmic time O(log n)
- Quadratic time O(n^2)
- Cubic time O(n^3)

doBracketsMatch

Without squandering time, we should discuss this idea by using the following function while optimizing it's efficiency.

```
function doBracketsMatch (inputString, openingSymbol, closingSymbol) {
//the function will take parameters and use them locally
let stack = new BracketStack()
let isOpeningSymbol = isSymbol(openingSymbol)
let isClosingSymbol = isSymbol(closingSymbol)
// we have an input string within our for loop, this will split and assign brackets to two functions
that checks if the bracket is an opening or closing bracket
for ( let i = 0 ; i < inputString.length; i++) {

let value = inputString[i]
//let me try to structure this code for readability

if (isOpeningSymbol(value)) {
 stack.push() // Add to stack
}
else if (isClosingSymbol(value)) {
```

```
          if (stack.isEmpty()) {

              return false
}}

else {
 stack.pop() //remove from the stack

   }
}
return stack.isEmpty() // which means the brackets match
}
```

The best solution and justifications

We don't require to call millions of methods to perform this task, the logic is very easy. We can use the following steps:

a) We should declare a character stack that hold only the opening brackets of the given string.
b) Traverse the expression.
c) If the given character is a opening bracket then add (push) it to stack
d) Else remove the bracket given that the stack id not empty and the bracket type matches with the stacked opening bracket.
e) If the stack is empty, return true.

This will reduce the time taken to execute the code and the space taken within the memory of the computer will be less. The best way to avoid problems in this function is to add the following in the loop.

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

    // If character is an opening brace just add it to the declared stack
    if (inputString [i] === '(' || inputString [i] === '{' || inputString [i] === '[') {
            stack.push(str[i]);

         }
             // If that character is a closing brace, just pop from the stack because this is zero and one,
true or false situation

        else {

                if(stack.isEmpty()){

                  return false;

                      } else { stack.pop();}

          // doesn't match the closing brace bracket type in the stack, then return false

         // we can declare a map object to achieve this comparison

        }
```

The time and the complexity of the function depends upon the number of characters in our given input string. Assurance of bracket types is very crucial in this case because we are only dealing with one string that comprises of various brackets. The computation will be using linear time of N time, this could get larger depending on the string input. Hence, Time= O(n) as our algorithm scale in a linear fashion.

Valid and Invalid computations of Brackets

There are many ways to present this section, we can use the following string as our input to the function, inputString = “{} {} {}”. This string does not use much of the space because it keeps removing the brackets after stacking. That will also give the True output; thus, the following will use more space and give False as output. For instance, inputString = “{{{{{{{{{{{{{{}}}” will use more space while staking the bracket before removing them one by one.

Best solution based on Time and Space Complexity

Mimicking the given `doBracketsMatch` algorithm

```
let doBracketsMatch = (inputString,) => {
return !(inputString).split('').reduce((openBracketsCount,character)=>{
// If character is an opening brace just add it to the declared stack

        if (character === '(' || character === '{' || character === '['
) {

            return ++openBracketsCount;
              } //else pop it, remember that we did not specify the
type of bracket, which makes the logic invalid for some reasi=ons
else if (character === ')' || character === '}' || character === ']') {

 return --openBracketsCount;

}

return openBracketsCount
},0);
}
// this can be tested by calling a function with some string brackets i
n it
console.log(doBracketsMatch("{[]}"));
```

// this can be tested by calling a function with some string brackets in it

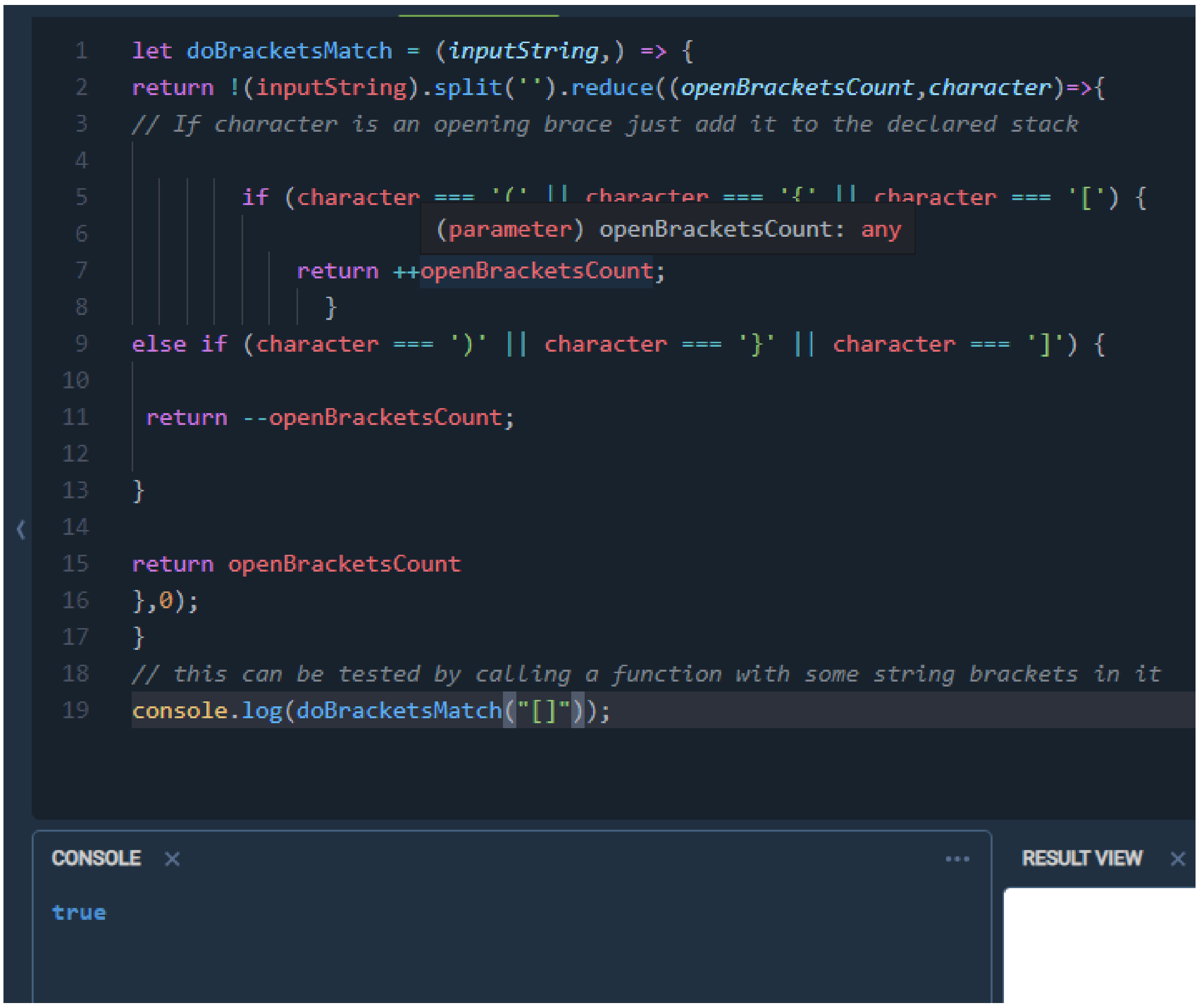
Best solution for `doBracketsMatch` algorithm

```
let doBracketsMatch  = (inputString) => {
    let stackBacket = [];
    let openBracket = {
        '{': '}',
        '[': ']',
        '(': ')'
    };
    let closedBracket = {
        '}': true,
        ']': true,
        ')': true
    };
    for (let i = 0; i < inputString.length; i++) {
        let character = inputString[i];
          if (openBracket[character]) {
            stackBacket.push(character);//add if this is an open
bracket
        } else if (closedBracket[character]) {
//else pop
            if (openBracket[stackBacket.pop()] !== character) return fa
lse;
        }
    }
    return stackBacket.length === 0;
}//output the called function
console.log(doBracketsMatch("[]{[]}[}"));
```

Conclusion

Time and space complexity is very crucial in programming because we should be able to trace the time taken to compute an algorithm. The size of input has strong impact in this computation because a smaller size reduces the number of comparison and memory taken by the algorithm or function. Hence, since we are using a single string in this example, we don't have to worry about the space of that variable but the computations during run-time.

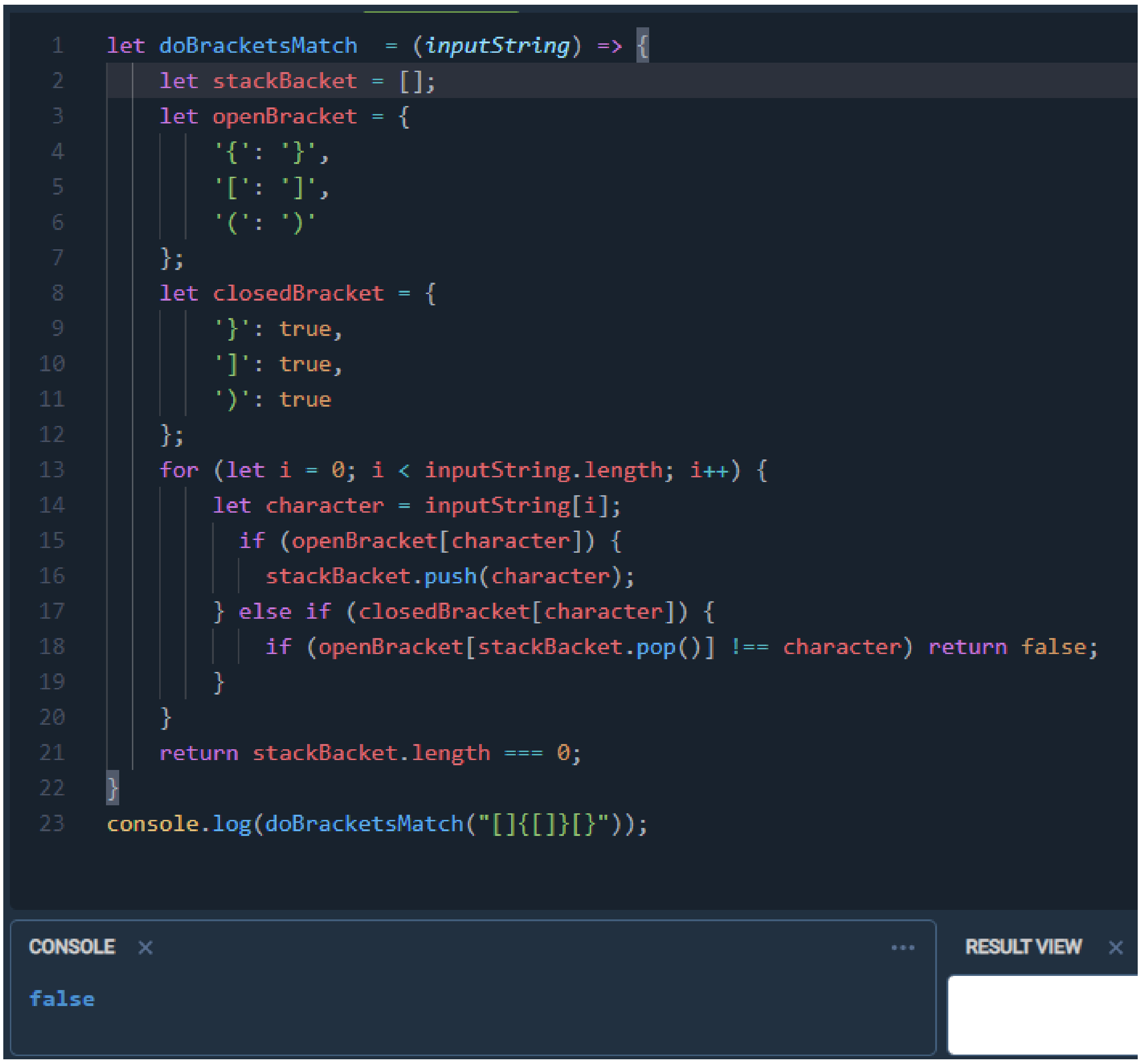
Appendix A



```
let doBracketsMatch = (inputString,) => {
return !(inputString).split('').reduce((openBracketsCount,character)=>{
// If character is an opening brace just add it to the declared stack

        if (character === '(' || character === '{' || character === '[') {

            return ++openBracketsCount;
              }
else if (character === ')' || character === '}' || character === ']') {

 return --openBracketsCount;

}

return openBracketsCount
},0);
}
// this can be tested by calling a function with some string brackets in it
console.log(doBracketsMatch("[]"));
```

```
true
```

Appendix B



```
let doBracketsMatch  = (inputString) => {
    let stackBacket = [];
    let openBracket = {
        '{': '}',
        '[': ']',
        '(': ')'
    };
    let closedBracket = {
        '}': true,
        ']': true,
        ')': true
    };
    for (let i = 0; i < inputString.length; i++) {
        let character = inputString[i];
          if (openBracket[character]) {
            stackBacket.push(character);
        } else if (closedBracket[character]) {
            if (openBracket[stackBacket.pop()] !== character) return false;
        }
    }
    return stackBacket.length === 0;
}
console.log(doBracketsMatch("[]{[]}[}"));
```

```
false
```