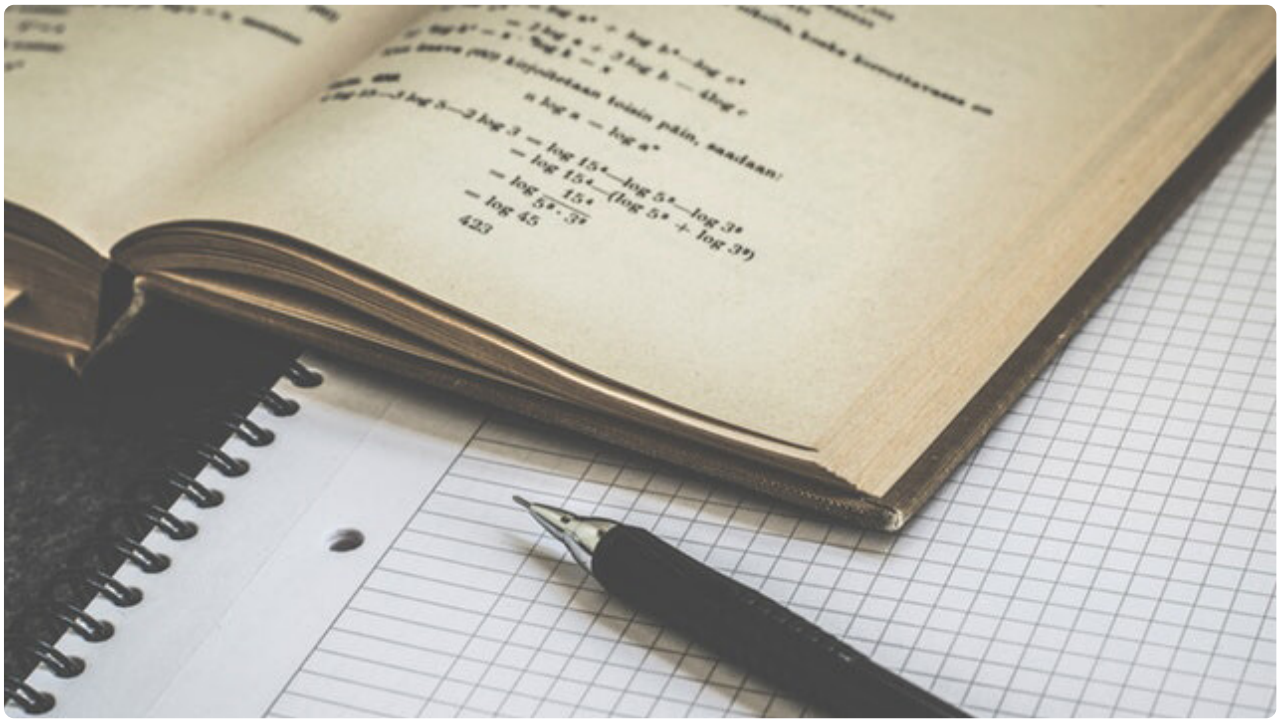
09 有效的括号

更新时间：2019-08-15 09:43:24



> 知识犹如人体的血液一样宝贵。
>
> ——高士其

刷题内容

> 难度: **Easy**
>
> 原题链接：https://leetcode-cn.com/problems/valid-parentheses/。

内容描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：
1.左括号必须用相同类型的右括号闭合。
2.左括号必须以正确的顺序闭合。

注意：空字符串可被认为是有效字符串。

示例 :

输入: "()"
输出: true

输入: "()[]{}"
输出: true

输入: "(]"
输出: false

输入: "{[]}"
输出: true

解题方案

思路 1：时间复杂度: O(N^2) 空间复杂度: O(N)

这道题其实挺简单的，我们用 `replace()` 方法就好。

> replace() 方法把字符串中的 old（旧字符串） 替换成 new(新字符串)，如果指定第三个参数max，则替换不超过 max 次。

要知道在“有效的括号”中，不论是 `()` 还是 `[]` 还是 `{}`，肯定都是成对出现的，不会有半边括号出现，所以我们只需要用 `replace()` 方法将字符串中的 `()` ， `[]` ， `{}` 全部替换成空字符串 `''` 即可。如果将全部成对出现的括号全部替换为空字符串的话，字符串的长度应该为 0 。如果不为 0 说明字符串不是一个 “有效的括号” 字符串。

Python beats 15.81%

```python
class Solution:
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        while '[]' in s or '()' in s or '{}' in s:
            s = s.replace('[]','').replace('()','').replace('{}','')
        return len(s) == 0
```

Java beats 6.12%

```java
class Solution {
    /**
     * 无限循环，每次将成对的括号替换为空字符串
     * 如果替换之前和替换之后的长度一样，说明没有成对的字符串，或者字符串为空字符串，跳出循环体
     * 如果替换之前和替换之后的长度不一样，说明有成对的字符串被替换掉，继续循环替换成对的字符串
     * 直到替换之前和替换之后的长度一样，跳出循环体为止
     * 最后判断字符串的长度是否为 0 来判断字符串是否是有效的括号
     * @param s
     * @return
     */
    public boolean isValid(String s) {
        int length;
        while (true) {
            length = s.length();
            s = s.replace("()","");
            s = s.replace("[]","");
            s = s.replace("{}","");
            if (length == s.length()) {
                break;
            }
        }
        return length == 0;
    }
}
```

go beats 9.52%

```go
import (
 "strings"
)

func isValid(s string) bool {
  for {
  l := len(s)
  s = strings.Replace(s, "()", "", -1)
  s = strings.Replace(s, "[]", "", -1)
  s = strings.Replace(s, "{}", "", -1)
     //判断s是否没变过，相当于s不存在(),[],{}
  if len(s) == l {
   break
  }
 }
 return len(s) == 0
}
```

c++ beats 14.08%

```
class Solution {
public:
    //由于c++没有replace函数，自己写一个，o(n)复杂度
    string replace(string s, string r) {
        int index = s.find(r);
        if (index == -1) {
            return s;
        } else {
            return s.substr(0, index) + s.substr(index + r.size());
        }
    }
    bool isValid(string s) {
        while (1) {
            int l = s.size();
            s = replace(s, "()");
            s = replace(s, "[]");
            s = replace(s, "{}");
            //判断s是否没变过，相当于s不存在(),[],{}
            if (l == s.size()) {
                break;
            }
        }
        return s.size() == 0;
    }
};
```

但是很明显这种方法的时间和空间复杂度都挺高的，因为需要多次遍历整个字符串，所以会占用很多的资源。那我们该如何优化一下呢？

思路 2：时间复杂度: O(N) 空间复杂度: O(N)

其实关于这道题我们可以用 栈 这个数据结构来做。我们知道栈是后进先出的数据结构，这样的特性方便我们进行左右括号的对比，先来看下大体的思路：

1. 遍历整个字符串，如果是左括号就入栈；
2. 如果是右括号则查看当前栈顶元素是否与之相匹配，如果不匹配直接返回 false；
3. 遍历完成之后，如果栈内没有元素则说明全部匹配成功，返回 true；如果栈内还有元素则说明不匹配，返回 false，其实直接 return stack == 0 就行。

下面来看具体的代码实现：

Python 91.92%

```python
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        leftP = '([{'
        rightP = ')]}'
        stack = []
        for char in s:
            if char in leftP:
                stack.append(char)
            if char in rightP:
                if not stack:
                    return False
                tmp = stack.pop()
                if char == ')' and tmp != '(':
                    return False
                if char == ']' and tmp != '[':
                    return False
                if char == '}' and tmp != '{':
                    return False
        return stack == []
```

Java beats 65.35%

```java
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
            // 如果是左括号就入栈
            if (s.charAt(i) == '{'|| s.charAt(i) == '[' || s.charAt(i) == '(') {
                stack.push(s.charAt(i));
            }
            // 如果是右括号就判断栈是否为空，以及栈顶是否和右括号匹配，如果不匹配直接可以返回 false
            if (s.charAt(i) == ']') {
                if (stack.isEmpty() || stack.peek() != '[') {
                    return false;
                }
                stack.pop();
            }
            if (s.charAt(i) == '}') {
                if (stack.isEmpty() || stack.peek() != '{') {
                    return false;
                }
                stack.pop();
            }
            if (s.charAt(i) == ')') {
                if (stack.isEmpty() || stack.peek() != '(') {
                    return false;
                }
                stack.pop();
            }
        }
        // 如果全部匹配，则栈为空，否则不为空
        return stack.isEmpty();
    }
}
```

c++ beats 100%

```
class Solution {
public:
    //用数组模拟栈，放在这里避免重复申请内存，提升速度，不然也不会beats 100%
    char a[100000];
    bool isValid(string s) {
        int res = 0;
        for (int i = 0;i < s.size();i++) {
            //所有的左括号都直接进栈
            if (s[i] == '(' || s[i] == '{' || s[i] == '[') {
                a[res++] = s[i];
            } else if (s[i] == ')')
                //正确的表达式右括号一定在栈顶有对应的左括号，下同
                if (res == 0 || a[res - 1] != '(') {
                    return false;
                }
                res--;
            } else if (s[i] == ']') {
                if (res == 0 || a[res - 1] != '[') {
                    return false;
                }
                res--;
            } else {
                if (res == 0 || a[res - 1] != '{') {
                    return false;
                }
                res--;
            }
        }
        return res == 0;
    }
};
```

go beats 93.06%

```
import (
 "strings"
)

func isValid(s string) bool {
 stack := make([]rune, len(s))
 n := 0
 for _, c := range(s) {
  if c == '(' || c == '[' || c == '{' {
   stack[n] = c
   n++
  } else if c == ')' {
   if (n == 0 || stack[n - 1] != '(') {
    return false
   }
   n--
  } else if c == ']' {
   if (n == 0 || stack[n - 1] != '[') {
    return false
   }
   n--
  } else if c == '}' {
   if (n == 0 || stack[n - 1] != '{') {
    return false
   }
   n--
  }
 }
 if n == 0 {
  return true
 } else {
  return false
 }
}
```

可以看到思路 2 的时间复杂度明显降低了。

小结

其实我感觉这道题挺经典的，第一种方法写起来简单，但是耗时又占空间；第二种方法明显降低了时间复杂度，但是代码写起来不如第一种方法快。这是一个取舍问题，如果不考虑时间和空间的话直接选第一种方法就好，如果要考虑时间和空间的话最好还是第二种方法。在最合适的时候选用最合适的算法才是最完美的。

如果你有更好的方法的话，欢迎你在评论区给我留言。

}

08 最长公共前缀

10 合并两个有序链表