# 20. Valid Parentheses

难度(easy)

Given a string containing just the characters **'(', ')', '{', '}', '[' and ']'**, determine if the input string is valid.

An input string is valid if:

* Open brackets must be closed by the same type of brackets.
* Open brackets must be closed in the correct order.

Note that **an empty string** is also considered valid.

Example 1:

Input: "()"

Output: true

Example 2:

Input: "()[]{}"

Output: true

Example 3:

Input: "(]"

Output: false

Example 4:

Input: "([)]"

Output: false

Example 5:

Input: "{[]}"

Output: true

注意：成对的括号的ASCII不是挨着的。



## 算法1：利用栈实现

思路：首先将字符串转换成字符数组，遍历字符数组。当遇到左括号的时候直接进栈，若遇到右括号，需要判断当前栈是否为空，若为空则返回false；或者前一个左括号是否和当前右括号匹配，若不匹配也是返回false。遍历结束之后，若栈为空则返回true，否则返回false。

//利用栈实现

class Solution {

public boolean isValid(String s) {

if(s == null||"".equals(s)) return true;

Deque<Character> stack = new ArrayDeque<Character>();

**List<Character> parentheses = Arrays.asList('(',')','[',']','{','}');**

char[] chars = s.toCharArray();

for(char c : chars){

int index = parentheses.indexOf(c);

if(index%2 == 0){

stack.push(c);

}else{

if(**stack.isEmpty() || index - parentheses.indexOf(stack.pop()) != 1){**

return false;

}

}

}

return stack.isEmpty();

}

}

## 算法2：利用栈，但是不需要List

思路：算是算法1的改进。这里用了一个小技巧，不向栈中存储左括号，而是存储其对应的右括号。目的就是为了真正来了右括号的时候判断是否相等即可。而算法1是通过获取List中的索引是否相差1来实现的。

//利用栈

class Solution {

public boolean isValid(String s) {

if (s == null || s.length() == 0) return true;

Stack<Character> stack = new Stack<>();

for (char c : s.toCharArray()) {

if (c == '**(**') {

stack.push(')');

} else if (c == '{') {

stack.push('}');

} else if (c == '[') {

stack.push(']');

} else if (stack.isEmpty() || stack.pop() != c) {

return false;

}

}

return stack.isEmpty();

}

}

## 算法3：利用数组和switch语句

思路：很简单，利用数组存储左括号，利用top指针指示位置。如果左括号来了，就向右移动一位，(这里不能超过元字符串长度的一半)，如果右括号来了，就向右移动一位，前提是top需要大于0。

//利用字符数组实现

class Solution {

public boolean isValid(String s) {

if (s == null || s.length() == 0) return true;

char[] chars = new char[s.length()/2+1];

int top = 0;

for(int i = 0;i < s.length();i++){

switch(s.charAt(i)){

case '(':

case '[':

case '{':

**if(top >= s.length()/2+1)**

**return false;**

**chars[top++] = s.charAt(i);**

**break;**

case ')':

**if(top > 0){**

**if(chars[top-1] == '('){**

**top--;**

**break;**

**}**

**}**

**return false;**

case ']':

if(top > 0){

if(chars[top-1] == '['){

top--;

break;

}

}

return false;

case '}':

if(top > 0){

if(chars[top-1] == '{'){

top--;

break;

}

}

return false;

default: return false;

}

}

return top == 0;

}

}

# 22. Generate Parentheses

Given **n pairs of parentheses**, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

## 算法1：递归算法。利用字符串不断相加，不好，算法2采用字符数组。

思路：通过**左右括号计数**控制递归的结束。递归结束条件是openNum和closeNum都变成0；递归调用的条件：当openNum不为0就可以继续添加左括号，只有当closeNum大于openNum的时候才可以添加右括号；这样两种条件就可以把添加左括号和添加右括号涵盖了。（当然，进一步细致分析，如果openNum==closeNum，只能添加左括号，若openNum>closeNum，添加左括号和右括号都可以，不存在closeNum>openNum的情况。）

所以，如果写成这样也可以：

// if(openNum == closeNum){

// helper(openNum -1,closeNum,s+"(",result);

// }else{//这里其实只存在openNum>closeNum的情况

// if(openNum != 0)

// helper(openNum -1,closeNum,s+"(",result);

// if(closeNum != 0)

// helper(openNum,closeNum-1,s+")",result);

// }

class Solution {

public List<String> generateParenthesis(int n) {

List<String> result = new ArrayList<String>();

if(n <= 0) return result;

helper(n,n,"",result);

return result;

}

private void helper(int openNum,int closeNum,String s,List<String> result){

if(openNum == 0&&closeNum == 0){

result.add(s);

return;

}

**//重要是什么时候可以递归调用**

**if(openNum != 0){**

**helper(openNum -1,closeNum,s+"(",result);**

**}**

**if(openNum < closeNum){**

**helper(openNum,closeNum-1,s+")",result);**

**}**

}

}

## 算法2：原理和算法1一致，利用字符数组实现。

思路：同算法1。也是递归算法。利用字符数组替代了字符串的不断增长，效果较好。

**建议：记住这种算法**。

class Solution {

public List<String> generateParenthesis(int n) {

List<String> res = new ArrayList<>();

if(n == 0) return res;

**char[] array = new char[2 \* n];//注意字符数组的长度是2\*n**

helper(array, 0, n, n, res);

return res;

}

private void helper(char[] array, int index, int left, int right, List<String> res) {

if(index == array.length) {

res.add(new String(array));

return;

}

if(left > 0) {

**array[index] = '(';**

helper(array, index + 1, left - 1, right, res);

}

if(right > left) {

**array[index] = ')';**

helper(array, index + 1, left, right - 1, res);

}

}

}