1. **程序设计实践**

本部分内容的5道编程题综合考察学员的程序设计能力，算法的设计能力，递归的使用等，答案不唯一，学生需要综合考虑程序的时间复杂度和空间复杂度。

1. **买卖股票的最佳时机**

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

**示例 1:**

输入: [7,1,5,3,6,4]输出: 5解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

**示例 2:**

输入: [7,6,4,3,1]输出: 0解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

请补全以下代码：

class Solution {

public int maxProfit(int[] prices) {//找差值最大的就可以

**int** current = 0;

**int** max = 0;

**int** i=0;

**int** j=i+1;

**while**(i<j&&j<prices.length&&i<prices.length)

{

**if**(prices[i]<prices[j])

{

current = prices[j] - prices[i];

**if**(current > max)

{

max = current;

}

j++;

}

**else**

{

i=j;

j++;

}

}

**return** max;

}

}

1. **正则表达式匹配**

给定一个输入字符串s, 和一个模式p, 实现正则匹配规则，支持. 和 \*， 要求模式匹配整个输入字符串，而不是部分匹配。

'.' 匹配任意一个字符.

'\*' 匹配零个或多个前序字符串.

**注意:**

s 可以为空，并且只包含小写的 a-z.

p 可以为孔，并且只包含小写的 a-z, 以及 . 和 \*.

**示例 1:**

**输入:**

s = "aa"

p = "a"

**输出:** false

**解释:** "a" 不能匹配整个输入字符串 "aa".

**示例2:**

**输入:**

s = "aa"

p = "a\*"

**输出:** true

**解释:** '\*' 代表零个或者多个前序字符串 'a'.因此，只要重复a一次，就可以得到输入字符串 "aa".

**示例 3:**

**输入:**

s = "ab"

p = ".\*"

**输出:** true

**解释:** ".\*" 代表零个或者多个任意字符 (.)".

**示例 4:**

**输入:**

s = "aab"

p = "c\*a\*b"

**输出:** true

**解释:** c 重复0次, a 重复2次. 因此它匹配输入字符串 "aab".

**示例 5:**

**输入:**

s = "mississippi"

p = "mis\*is\*p\*."

**输出:** false

解释：p无法匹配输入字符串中的mississ**i**ppi

请补全以下代码：

class Solution {

public boolean isMatch(String s, String p) {

**if** (p.isEmpty()) {

**return** s.isEmpty(); //空的话判断另一个是否为空

} **else** {

**if** (s.isEmpty()) {

**if** (p.equals("\*")) {

**return** **true**; //如果为空,但p为\*,则为true

} **else** **if** (p.charAt(0) != '\*') {

**return** **false**; //否则为false

} **else** **if** (p.charAt(0) == '\*' && p.length() > 1) {

**return** isMatch(s, p.substring(1)); //再比较下一个字母

}

}

**else** { //s不为空

**if** ((s.charAt(0) == p.charAt(0) || p.charAt(0) == '.') && (p.length() == 1 || p.charAt(1) != '\*')) {

**return** isMatch(s.substring(1), p.substring(1)); //上述情况是第一个匹配成功,再匹配下一个

} **else** **if** (p.length() > 1 && p.charAt(1) == '\*') {

**if** (s.charAt(0) != p.charAt(0)) {

**if** (p.charAt(0) != '.') {

**return** isMatch(s, p.substring(2)); //再往后走一个

} **else** {

**return** **true**;

}

} **else** {

**int** index = 0;

**do** {

index++;

} **while** (index < s.length() && (s.charAt(index) == s.charAt(0)));

**return** isMatch(s.substring(index), p.substring(2));

}

}

}

}

**return** **false**;

}

}

1. **朋友圈**

班上有 **N**名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 **N \* N**的矩阵 **M**，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生**互为**朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

**示例 1:**

**输入:**

**[[1,1,0],**

**[1,1,0],**

**[0,0,1]]**

**输出: 2**

**说明：已知学生0和学生1互为朋友，他们在一个朋友圈。**

**第2个学生自己在一个朋友圈。所以返回2。**

**示例 2:**

**输入:**

**[[1,1,0],**

**[1,1,1],**

**[0,1,1]]**

**输出: 1**

**说明：已知学生0和学生1互为朋友，学生1和学生2互为朋友，所以学生0和学生2也是朋友，所以他们三个在一个朋友圈，返回1。**

**注意：**

1. N 在[1,200]的范围内。
2. 对于所有学生，有M[i][i] = 1。
3. 如果有M[i][j] = 1，则有M[j][i] = 1。

请补全以下代码：

class Solution {

**public** **int** findCircleNum(**int**[][] M) {

**int** res = 0; //朋友圈数

**int** length = M.length;

**boolean**[] record = **new** **boolean**[length];

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

**if** (!record[i])

{

deepfinds(M, record, i); //对每行每列判断,现在是i和剩下的人判断

res++;

}

}

**return** res;

}

**public** **void** deepfinds(**int**[][] M, **boolean**[] record, **int** m) {

**if** (record[m]) {

**return**;

} **else** {

record[m] = **true**;

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

**if** (M[m][i] == 1) { //m和i是朋友

deepfinds(M, record, i);//再来一次

}

}

}

}

}

1. **复原IP地址**

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

有效的 IP 地址正好由四个整数（每个整数位于 0 到 255 之间组成），整数之间用 '.' 分隔。

**示例:**

**输入: "25525511135"**

**输出: ["255.255.11.135", "255.255.111.35"]**

请补全以下代码：

class Solution {

public List<String> restoreIpAddresses(String s) {

**int** length = s.length(); //ip地址的总长度

List<String> record = **new** ArrayList<String>();

**for**(**int** i=1;i<=3;i++) //第一个点的位置

{

**for**(**int** j=i+1;(j-i<=3);j++) //第二个点的位置，中间最多3个数

{

**for**(**int** k=j+1;(k-j<=3)&&k<length;k++) //第三个点的位置

{

**int** flag = 0; //相当于true 判断key是否复合要求，不符合就continue

**int** cas = -1; //cas可以看做一个定位器 定位到i/j/k，就放一个点，没定位到是-1

**int** m=0; //m是当前字符串中的字符位置，现在是第一个

**char**[] str = **new** **char**[length+3]; //包含数字与.的一个字符集合

**int** l=0; //l记录当前的总长度

**while**(l<length+3)//一次

{

**if**((cas==i||cas==j||cas==k))

{

str[l] = '.'; //加一个点上去

String key="";

**if**(m==i)

{

key = s.substring(0,l); //取出字符

}

**if**(m==j)

{

key = s.substring(i,m);

}

**if**(m==k)

{

key = s.substring(j,m);

}

cas = -1; //cas 再计为-1

//判断取出的字符是否符合条件

**if**(key.length()>1&&key.startsWith("0"))

{

flag = 1 ;

**break** ;

}

**if**(Integer.*parseInt*(key)>255||(length-k>3))

{

flag = 1 ;

**break** ;

}

}

**else** //刚开始，不满足if条件，先取一个字符

{

str[l] = s.charAt(m);

m++; //往后走一个

cas = m;

}

l++;//每次处理完都要加

}

**if**(flag == 1)

{

**continue**;

}

String ipstr = **new** String(str);

record.add(ipstr);

}

}

}

//现在已经获取了一些字符的集合，但他没有排除最后一串数字不符合的情况，其实这一段代码也可以写到上面

//用来遍历 record

**for**(**int** i=0;i<record.size();i++)

{

String str=record.get(i);

**int** key = str.lastIndexOf(".");

String ss=str.substring(key+1,str.length());

**char**[] temp = str.toCharArray();

**if**((temp[key+1]=='0'&&(key!=temp.length-2))||Integer.*parseInt*(ss)>255) //排除最后一个点后面的数字不符合的情况

{

record.remove(i);

i--;

}

}

**return** record;

}

}

1. **简化路径**

以 Unix 风格给出一个文件的**绝对路径**，你需要简化它。或者换句话说，将其转换为规范路径。

在 Unix 风格的文件系统中，一个点（.）表示当前目录本身；此外，两个点 （..） 表示将目录切换到上一级（指向父目录）；两者都可以是复杂相对路径的组成部分。更多信息请参阅：[Linux / Unix中的绝对路径 vs 相对路径](https://blog.csdn.net/u011327334/article/details/50355600)

请注意，返回的规范路径必须始终以斜杠 / 开头，并且两个目录名之间必须只有一个斜杠 /。最后一个目录名（如果存在）**不能**以 / 结尾。此外，规范路径必须是表示绝对路径的**最短**字符串。

**示例 1：**

**输入："/home/"**

**输出："/home"**

**解释：注意，最后一个目录名后面没有斜杠。**

**示例 2：**

**输入："/../"**

**输出："/"**

**解释：从根目录向上一级是不可行的，因为根是你可以到达的最高级。**

**示例 3：**

**输入："/home//foo/"**

**输出："/home/foo"**

**解释：在规范路径中，多个连续斜杠需要用一个斜杠替换。**

**示例 4：**

**输入："/a/./b/../../c/"**

**输出："/c"**

**示例 5：**

**输入："/a/../../b/../c//.//"**

**输出："/c"**

**示例 6：**

**输入："/a//b////c/d//././/.."**

**输出："/a/b/c"**

请补全以下代码：

class Solution {

public String simplifyPath(String path) {

/".."和之前的都删掉 "//"变成一个/

String[] s = path.split("/");

Stack <String> stack = **new** Stack<String>();

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

**if**(!s[i].equals("") && !s[i].equals(".") && !s[i].equals("..")) {

stack.push("/"+s[i]);

}**else** **if**(s[i].equals("..") && !stack.isEmpty()) {

stack.pop();

}

}

**if**(stack.isEmpty()) {

**return** "/";

}

StringBuffer buffer = **new** StringBuffer();

**for**(**int** i = 0;i<stack.size();i++) {

buffer.append(stack.get(i));

}

**return** buffer.toString();

}

}