目录

[二分查找算法(Binary Search)的实现 2](#_Toc369035886)

[二叉查找树的常用方法（Binary Search Tree） 3](#_Toc369035887)

[把一个排好序的数组数组转化成一颗平衡二叉查询树 7](#_Toc369035888)

[二叉树的宽度The diameter of a binary tree 8](#_Toc369035889)

[在O(1)平均时间删除链表结点 10](#_Toc369035890)

[找出数组中两个只出现一次的数字 11](#_Toc369035891)

[二叉查询树三种遍历的非递归写法 12](#_Toc369035892)

[判断两个字符串是否是anagram 14](#_Toc369035893)

[分层打印二叉树 15](#_Toc369035894)

[二叉查询树的保存和读取 16](#_Toc369035895)

[implement a queue by using two stacks 17](#_Toc369035896)

[在O(1)时间内返回栈的最小值 18](#_Toc369035897)

[判断字符串可否通过另一个字符串的循环移位得到 20](#_Toc369035898)

[找出丢失的数字（思维经典） 20](#_Toc369035899)

[两个巨大正整数相加 20](#_Toc369035900)

[rand5()构造rand7() 22](#_Toc369035901)

[最长递增子序列 [No. 39] 22](#_Toc369035902)

[不用＋、－、×、÷数字运算符做加法 23](#_Toc369035903)

[判断一个单链表是否有环，如果有，找出环的起始位置 23](#_Toc369035904)

[pow(double a,int b) 25](#_Toc369035905)

[找出两个已经排好序的数组的中位数 25](#_Toc369035906)

[一个数组先从小到大递增后从大到小递减，找出最大的值 26](#_Toc369035907)

[一个已经排序的数组，去除里面重复的元素 27](#_Toc369035908)

[把一个字符串转成一个整数 28](#_Toc369035909)

[“I love china”反转后变成 “china love I” 29](#_Toc369035910)

[反转单向链表 (单个反转) 29](#_Toc369035911)

[两两反转单向链表 30](#_Toc369035912)

[格雷码的实现 31](#_Toc369035913)

[字符串排列和组合的JAVA实现 33](#_Toc369035914)

[顺时针打印矩阵 （JAVA实现） 35](#_Toc369035915)

[最大子矩阵和 36](#_Toc369035916)

[最近公共祖先 37](#_Toc369035917)

[判断一个树是不是二叉查询树 38](#_Toc369035918)

[打印二叉树所有的路径 38](#_Toc369035919)

[检查一个数组是否可能是一棵二叉查询树后序遍历的结果 39](#_Toc369035920)

[由A转成B所需的最少编辑操作次数 41](#_Toc369035921)

[左旋转字符串 41](#_Toc369035922)

[找出字符串中对称的子字符串的最大长度(最长回文) 42](#_Toc369035923)

[数组中出现次数超过一半的数字 43](#_Toc369035924)

[输入一个正数 n，输出所有和为 n 的连续正数序列 45](#_Toc369035925)

[在数组里查找这样的数，它大于等于左侧所有数，小于等于右侧所有数 45](#_Toc369035926)

[只使用加法实现减法，乘法和除法 [# 68] 46](#_Toc369035927)

[aaabbbbc， 压缩后成为：a3b4c1 48](#_Toc369035928)

[迷宫问题 49](#_Toc369035929)

[把手机键盘输入转化成短消息 51](#_Toc369035930)

[把一个字符串通过相邻字符移位变成另一个字符串 53](#_Toc369035931)

[求两个数组的交集 54](#_Toc369035932)

[间隔问题 [No. 56] 55](#_Toc369035933)

[间隔问题2 [No. 57] 58](#_Toc369035934)

[间隔问题3 [No. 58] 60](#_Toc369035935)

[判断一个字符串里是否包含另一个字符串所有的字符 61](#_Toc369035936)

[在一个字符串中找到第一个只出现一次的字符 62](#_Toc369035937)

[把一个字符串转成double类型的数[# 61] 63](#_Toc369035938)

[判断一个数字是否是回文 64](#_Toc369035939)

[找出二叉查找树中第n大的值 65](#_Toc369035940)

[给定一个二叉树，从左到右，找出第 k 个叶子节点 66](#_Toc369035941)

[在两个已经排好序的数组里找出第K小的数 67](#_Toc369035942)

[概率面试题骰子和收益 68](#_Toc369035943)

[判断两个链表是否相交 68](#_Toc369035944)

[求一个数字数组里的最大连续数字的个数 69](#_Toc369035945)

[第二个字符串是第一个的子串，打印第二个字符串缺了的字符位置 70](#_Toc369035946)

[括号匹配 70](#_Toc369035947)

[打印二叉树的边缘节点 71](#_Toc369035948)

[名人问题 (Celebrity problem) 72](#_Toc369035949)

[生男生女的面试题 73](#_Toc369035950)

[check whether two binary trees are identical 73](#_Toc369035951)

[不用循环反转字符串 73](#_Toc369035952)

[版本号比较 74](#_Toc369035953)

[约瑟夫 75](#_Toc369035954)

[不通过比较，找出两个数的最大值 76](#_Toc369035955)

## 二分查找算法(Binary Search)的实现

http://blog.csdn.net/beiyeqingteng/article/details/5736004

用二分查找在已排序的数组中查看该数组是否含有一个特定的值是非常快速的，时间复杂度为O(lgn). 二分查找思想很简单，但是实现的时候会在边界条件上出现一些意想不到的问题。

public int BinarySearchIteration(int[] array, int key)

{

int begin = 0;

int end = array.Length - 1;

while (begin <= end)

{

int mid = begin + (end - begin) / 2;

if (array[mid] > key)

{

end = mid - 1;

}

else if (array[mid] < key)

{

begin = mid + 1;

}

else

{

return mid;

}

}

return -1;

}

**基于递归方式实现：**

public int BinaraySearchRecursive(int[] array, int begin, int end, int key)

{

if (begin <= end)

{

int mid = begin + (end - begin) / 2;

if (array[mid] > key)

{

return BinaraySearchRecursive(array, begin, mid - 1, key);

}

else if (array[mid] < key)

{

return BinaraySearchRecursive(array, mid + 1, end, key);

}

else

{

return mid;

}

}

else

{

return -1;

}

}

## 二叉查找树的常用方法（Binary Search Tree）

http://blog.csdn.net/beiyeqingteng/article/details/6661683

二叉查找树（Binary Search Tree），或者是一棵空树，或者是具有下列性质的二叉树：

若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

它的左、右子树也分别为二叉排序树。

**二叉查找树****的常用方法：**

**1. 查找(lookup)**

static boolean lookup(Node rootNode, int target) {

// 1. Base case == empty tree

// in that case, the target is not found so return false

if (rootNode == null) return false;

else if (target == rootNode.data) return true;

else if (target < rootNode.data) return(lookup(rootNode.leftChild, target));

else return(lookup(rootNode.rightChild, target));

}

**2. 最大（max）**

public Node maxNode(Node rootNode) {

while (rootNode.righttChild != null) {

rootNode = rootNode.rightChild;

}

return rootNode;

}

**3. 最小 （min）**

public Node minNode(Node rootNode) {

while (rootNode.leftChild != null) {

rootNode = rootNode.leftChild;

}

return rootNode;

}

**4. 插入（insert）**

public static Node insert(Node root, int data) {

// 1. If the tree is empty, the new node is the root

if (root == null) {

return new Node(data);

}

else {

// 2. Otherwise, recur down the tree

if (data <= root.data) root.leftChild = insert(root.leftChild, data);

else root.rightChild = insert(root.rightChild, data);

}

return root;

}

**5. in-order tree walk**

public void inorder(Node rootNode) {

if (rootNode != null) {

inorder(rootNode.leftChild);

print(nodeNode.data);

inorder(rootNode.rightChild);

}

}

**6. successor and predecessor**

**successor**

public Node Successor(Node node) {

if (node.rightChild != null) {

return minNode(node.rightChild);

}

Node parentNode = node.parent;

while (parentNode != null && node == parentNode.rightChild) {

node = parentNode;

parentNode = parentNode.parent;

}

return parentNode;

}

**predecessor**

public Node predecessor (Node node) {

if (node.leftChild != null) {

return maxNode(node.rightChild);

}

Node parentNode = node.parent;

while (parentNode != null && node == parentNode.leftChild) {

node = parentNode;

parentNode = parentNode.parent;

}

return parentNode;

}

**8. 求节点的个数（size）**

/\*

Compute the number of nodes in a tree.

\*/

int size(Node rootNode) {

if (node == NULL) {

return(0);

} else {

return(size(rootNode.leftChild) + 1 + size(rootNode.rightChild));

}

}

**9. 把一个二叉树转成它的“镜像”（mirror）**

private void mirror(Node rootNode) {

if (rootNode != null) {

// do the sub-trees

mirror(rootNode.leftChild);

mirror(rootNode.rightChild);

// swap the left/right pointers

Node temp = rootNode.leftChild;

rootNode.leftChild = rootNode.rightChild;

rootNode.rightChild = temp;

}

}

**10. check balanced (isBalanced)**

int maxDepth(Node rootNode) {

if (rootNode == null) {

return 0;

}

return 1 + max(maxDepth(rootNode.leftChild), maxDepth(rootNode.rightChild));

}

int minDepth(Node rootNode) {

if (rootNode == null) {

return 0;

}

return 1 + min(minDepth(rootNode.leftChild), minDepth(rootNode.rightChild));

}

bool isBalanced(Node rootNode) {

return maxDepth(rootNode) - minDepth(rootNode) <= 1;

}

**11. First common ancestor**

/\*

\* get the first common ancestor of node p and node q

\*/

public static Node commonAncestor(Node rootNode, Node p, Node q) {

if (covers(rootNode.leftChild, p) && covers(rootNode.leftChild, q))

return commonAncestor(rootNode.leftChild, p, q);

if (covers(rootNode.rightChild, p) && covers(rootNode.rightChild, q))

return commonAncestor(rootNode.rightChild, p, q);

return rootNode;

}

/\*

\* check whether the node n is in the tree

\*/

private static boolean covers(Node rootNode, Node n) {

if(rootNode == null) return false;

if(rootNode == n) return true;

return covers(rootNode.leftChild, n) || covers(rootNode.rightChild, n);

}

如果这个Binary Tree 是 BST的话，可以利用左子树的值小于根节点的值，右子树的值大于根节点的值进行判断两个节点（p, q）的位置，这样更简单。

public Node LCA(Node root, Node p, Node q) {

if (root == null || p == null || q == null ) return NULL;

if (max(p.data, q.data) < root.data)

return LCA(root.left, p, q);

else if (min(p.data, q.data) > root.data)

return LCA(root.right, p, q);

else

return root;

}

这个算法的复杂度是O(h)。

**12. ISBST()**

Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all of the nodes in its left tree must be <= the node, and all of the nodes in its right subtree must be > the node.

private boolean isBST(Node node) {

if (node==null) return(true);

if (node.left!=null && maxValue(node.left) > node.data) return(false);

if (node.right!=null && minValue(node.right) <= node.data) return(false);

// check that the subtrees themselves are ok

return( isBST(node.left) && isBST(node.right) );

}

However, the approach given above is very expensive (O(n\*h)), because the maxValue(Node node) function will be called n times, and the complexity of the maxValue(Node node) is h (the height of the tree).

Because of the properity of BST, we can simply pass down the minvalue and maxvalue after narrowing the range of the minvalue and maxvalue, and the complexity of this algorithm is O(n). The code below shows the algorithm.

bool isBSTHelper(Node p, int low, int high) {

if (p == null) return true;

if (low < p.data && p.data < high)

return isBSTHelper(p.leftChild, low, p.data) &&

isBSTHelper(p.rightChild, p.data, high);

else

return false;

}

bool isBST(Node root) {

return isBSTHelper(root, INT\_MIN, INT\_MAX);

}

## 把一个排好序的数组数组转化成一颗平衡二叉查询树

http://blog.csdn.net/beiyeqingteng/article/details/6672018

BinaryTree\* sortedArrayToBST(int arr[], int start, int end) {

if (start > end) return NULL;

// same as (start+end)/2, avoids overflow.

int mid = start + (end - start) / 2;

BinaryTree \*node = new BinaryTree(arr[mid]);

node->left = sortedArrayToBST(arr, start, mid-1);

node->right = sortedArrayToBST(arr, mid+1, end);

return node;

}

BinaryTree\* sortedArrayToBST(int arr[], int n) {

return sortedArrayToBST(arr, 0, n-1);

}

## 二叉树的宽度The diameter of a binary tree

Definition: The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree.

The diameter of a tree T is the largest of the following quantities:

the diameter of T's left subtree

the diameter of T's right subtree

the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

/\* Function to get diameter of a binary tree \*/

static int diameter(Node root) {

if (root == null) {

return 0;

}

int lHeight = height(root.leftChild);

int rHeight = height(root.rightChild);

int lDiameter = diameter(root.leftChild);

int rDiameter = diameter(root.rightChild);

return max(lHeight + rHeight + 1, max(lDiameter, rDiameter));

}

/\* The function Compute the "height" of a tree. Height is the

number f nodes along the longest path from the root node

down to the farthest leaf node.\*/

static int height(Node root) {

if (root == null) {

return 0;

}

return 1 + max(height(root.leftChild), height(root.rightChild));

}

The complexity of the program is O(n^2), because the height function of called everytime when we go through the tree.

There is another solution and the complexity is O(N). The main idea of this approach is that the node stores its left child's and right child's maximum diameter if the node's child is the "root", therefore, there is no need to recursively call the height method. The drawback is we need to add two extra variables in the node class.

int findMaxLen(Node root) {

int nMaxLen = 0;

if (root == null) return 0;

if (root.leftChild == null) {

root.nMaxLeft = 0;

}

if (root.rightChild == null) {

root.nMaxRight = 0;

}

if (root.leftChild != null) {

findMaxLen(root.leftChild);

}

if (root.rightChild != null) {

findMaxLen(root.rightChild);

}

if (root.leftChild != null) {

int nTempMaxLen = 0;

nTempMaxLen = (root.leftChild.nMaxLeft > root.leftChild.nMaxRight) ?

root.leftChild.nMaxLeft : root.leftChild.nMaxRight;

root.nMaxLeft = nTempMaxLen + 1;

}

if (root.rightChild != null) {

int nTempMaxLen = 0;

nTempMaxLen = (root.rightChild.nMaxLeft > root.rightChild.nMaxRight) ?

root.rightChild.nMaxLeft : root.rightChild.nMaxRight;

root.nMaxRight = nTempMaxLen + 1;

}

if (root.nMaxLeft + root.nMaxRight > nMaxLen) {

nMaxLen = root.nMaxLeft + root.nMaxRight;

}

return nMaxLen;

}

## 在O(1)平均时间删除链表结点

///////////////////////////////////////////////////////////////////////

// Delete a node in a list

// Input: pListHead - the head of list

// pToBeDeleted - the node to be deleted

///////////////////////////////////////////////////////////////////////

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted)

{

if(!pListHead || !pToBeDeleted)

return;

// if pToBeDeleted is not the last node in the list

if(pToBeDeleted->m\_pNext != NULL)

{

// copy data from the node next to pToBeDeleted

ListNode\* pNext = pToBeDeleted->m\_pNext;

pToBeDeleted->m\_nKey = pNext->m\_nKey;

pToBeDeleted->m\_pNext = pNext->m\_pNext;

// delete the node next to the pToBeDeleted

delete pNext;

pNext = NULL;

}

// if pToBeDeleted is the last node in the list

else

{

// get the node prior to pToBeDeleted

ListNode\* pNode = pListHead;

while(pNode->m\_pNext != pToBeDeleted)

{

pNode = pNode->m\_pNext;

}

// deleted pToBeDeleted

pNode->m\_pNext = NULL;

delete pToBeDeleted;

pToBeDeleted = NULL;

}

}

值得注意的是，为了让代码看起来简洁一些，上面的代码基于两个假设：（1）给定的结点的确在链表中；（2）给定的要删除的结点不是链表的头结点。不考虑第一个假设对代码的鲁棒性是有影响的。至于第二个假设，当整个列表只有一个结点时，代码会有问题。但这个假设不算很过分，因为在有些链表的实现中，会创建一个虚拟的链表头，并不是一个实际的链表结点。这样要删除的结点就不可能是链表的头结点了。当然，在面试中，我们可以把这些假设和面试官交流。这样，面试官还是会觉得我们考虑问题很周到的。

## 找出数组中两个只出现一次的数字

///////////////////////////////////////////////////////////////////////

// Find two numbers which only appear once in an array

// Input: data - an array contains two number appearing exactly once,

// while others appearing exactly twice

// length - the length of data

// Output: num1 - the first number appearing once in data

// num2 - the second number appearing once in data

///////////////////////////////////////////////////////////////////////

void FindNumsAppearOnce(int data[], int length, int &num1, int &num2)

{

if (length < 2)

return;

// get num1 ^ num2

int resultExclusiveOR = 0;

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

resultExclusiveOR ^= data[i];

// get index of the first bit, which is 1 in resultExclusiveOR

unsigned int indexOf1 = FindFirstBitIs1(resultExclusiveOR);

num1 = num2 = 0;

for (int j = 0; j < length; ++ j)

{

// divide the numbers in data into two groups,

// the indexOf1 bit of numbers in the first group is 1,

// while in the second group is 0

if(IsBit1(data[j], indexOf1))

num1 ^= data[j];

else

num2 ^= data[j];

}

}

///////////////////////////////////////////////////////////////////////

// Find the index of first bit which is 1 in num (assuming not 0)

///////////////////////////////////////////////////////////////////////

unsigned int FindFirstBitIs1(int num)

{

int indexBit = 0;

while (((num & 1) == 0) && (indexBit < 32))

{

num = num >> 1;

++ indexBit;

}

return indexBit;

}

///////////////////////////////////////////////////////////////////////

// Is the indexBit bit of num 1?

///////////////////////////////////////////////////////////////////////

bool IsBit1(int num, unsigned int indexBit)

{

num = num >> indexBit;

return (num & 1);

}

## 二叉查询树三种遍历的非递归写法

http://blog.csdn.net/beiyeqingteng/article/details/6725796

public static void BSTPreorderTraverse(Node node) {

if (node == null) {

return;

}

Stack<Node> s = new Stack<Node>();

s.push(node);

while (!s.empty()) {

node = s.pop();

System.out.println(node.toString());

if (node.rightChild != null) {s.push(node.rightChild);}

if (node.leftChild != null) {s.push(node.leftChild);}

}

}

public static void BSTInorderTraverse(Node node) {

Stack<Node> s = new Stack<Node>();

while (!s.empty() || node != null) {

if (node != null) {

s.push(node);

node = node.leftChild;

} else {

node = s.pop();

System.out.println(node.toString());

node = node.rightChild;

}

}

}

public static void BSTPostorderTraverse(Node node) {

if (node == null) {

return;

}

Stack<Node> s = new Stack<Node>();

Node cur = node;

while (true) {

if (cur != null) {

if (cur.rightChild != null) {

s.push(cur.rightChild);

}

s.push(cur);

cur = cur.leftChild;

continue;

}

if (s.empty()) {

return;

}

cur = s.pop();

if (cur.rightChild != null && !s.empty() && cur.rightChild == s.peek()) {

s.pop();

s.push(cur);

cur = cur.rightChild;

} else {

System.out.println(cur.toString());

cur = null;

}

}

}

## 判断两个字符串是否是anagram

http://blog.csdn.net/beiyeqingteng/article/details/6746494

所谓 anagram, 就是两个词所用的字母及其个数都是一样的，但是，字母的位置不一样。比如 abcc 和 cbca 就是 anagram. 判断的方法比较简单，先把单个字母（字符）转成整数，然后利用了hashtable+计数器的原理进行判断。

public static boolean anagrams(String a, String b) {

if (a.length() != b.length()) return false;

int letters[] = new int[256];

//initialization

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

letters[i] = 0;

}

//process the string a

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

++letters[c];

}

//if the char appears in string b, decrease the corresponding number of counts.

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

if (letters[c] == 0) {

return false;

}

--letters[c];

}

//if string a and b are anagrams, all the values in array letters should be 0

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

if (letters[i] != 0) {

return false;

}

}

return true;

}

## 分层打印二叉树

http://blog.csdn.net/beiyeqingteng/article/details/6746497

问题：

把一个二叉树，安装从root到leaf的顺序把每一层上的node从左到右打印出来。

分析：

利用两个arraylist，一个arraylist装上一层的node, 另一个arraylist装上一层的child。如果上一层arraylist空了，两个arraylist互换。

代码：

//print the binary tree by level

public static void printByLevel(Node node) {

ArrayList<Node> list1 = new ArrayList<Node>();

ArrayList<Node> list2 = new ArrayList<Node>();

list1.add(node);

while(list1.size() != 0) {

for (int i = 0; i < list1.size(); i++) {

System.out.print(list1.get(i).value + " ");

if (list1.get(i).leftChild != null) list2.add(list1.get(i).leftChild);

if (list1.get(i).rightChild != null) list2.add(list1.get(i).rightChild);

}

System.out.println();

list1.clear();

ArrayList<Node> temp = list1;

list1 = list2;

list2 = temp;

}

}

class Node {

Node leftChild = null;

Node rightChild = null;

String name;

Node(String name) {

this.name = name;

}

}

## 二叉查询树的保存和读取

http://blog.csdn.net/beiyeqingteng/article/details/6770006

问题：把一个二叉查询树保存到一个文件，然后通过这个文件把这个二叉查询树还原出来。

我们使用pre-order遍历把一个二叉查询树保存，原因是只有pre-order遍历是从root开始保存，这样，当我们读取的时候，才能够把那个值放在root。

这里我用print来表示保存。

public void preOrderWrite(Node root) {

if (root!= null) {

System.out.println(root.value);

preOrderWrite(root.leftChild);

preOrderWrite(root.rightChild);

}

}

我们把保存的二叉树复原的时候，只需要使用二叉树的插入方法即可。

public static Node insert(Node root, int data) {

// 1. If the tree is empty, the new node is the root

if (root == null) {

return new Node(data);

}

else {

// 2. Otherwise, recur down the tree

if (data <= root.data) root.leftChild = insert(root.leftChild, data);

else root.rightChild = insert(root.rightChild, data);

}

return root;

}

插入操作的复杂度是：O(nlgn).

如果不是二叉查询树，而是二叉树，上面这个方法是不能用的。这里有两种方法，一种是先把二叉树保存为一颗满二叉树，然后保存在数组里。第二种方法是保存这颗二叉树的先序和中序数组，利用这两个数组来恢复二叉树。

struct Node

{

char data;

Node \* lchild;

Node \* rchild;

};

Node\* CreatTree(string pre, string in)

{

Node \* root = NULL; //树的初始化

if(pre.length() > 0)

{

root = new Node; //为根结点申请结构体所需要的内存

root->data = pre[0]; //先序序列的第一个元素为根结点

int index = in.find(root->data); //查找中序序列中的根结点位置

root->lchild = CreatTree(pre.substr(1, index), in.substr(0, index)); //递归创建左子树

root->rchild = CreatTree(pre.substr(index + 1), in.substr(index + 1)); //递归创建右子树

}

return root;

}

## implement a queue by using two stacks

http://blog.csdn.net/beiyeqingteng/article/details/6905814

For a queue, we need to implement enqueue and dequeue methods.

The idea is if we perform enqueue operation, we simply put that input into the stack 1 by performing the push operation; when we perform dequeue, as the first item is at the bottom of the stack, we need to use the second stack to reverse the order. Therefore, after the reverse, we can pop the first item in the stack 2. Next time, when we perform enqueue, we keep pushing the input into the stack 1, but do dequeue operation, we need to check whether the stack 2 is empty, if it is not empty, we need to pop it out as the first item in the queue.

public class Test {

Stack<Integer> stack1;

Stack<Integer> stack2;

public Test() {

stack1 = new Stack<Integer>();

stack2 = new Stack<Integer>();

}

public void enqueue (int value) {

stack1.push(value);

}

public Object dequeue() {

/\*

\* If the stack2 is not empty, then the top one must be the first one in the queue

\*/

if (stack2.empty() == false) {

return stack2.pop();

}

/\*

\* If the stack2 is empty, we need to get the enqueued data from stack 1

\*/

if (stack1.empty() == false) {

stack2.push(stack1.pop());

}

/\*

\* after finishing getting the data from stack1, we shoud pop the fist item in stack2

\*/

return stack2.pop();

}

}

## 在O(1)时间内返回栈的最小值

http://blog.csdn.net/beiyeqingteng/article/details/6915988

public class NewStack extends Stack<Node> {

public void push(Node node) {

if (node.getValue() < this.min()) {

node.setMin(node.getValue());

}

super.push(node);

}

public Node pop() {

return super.pop();

}

public int min() {

if (peek() == null) {

return MAX\_INT;

} else {

peek().getMin();

}

}

}

class Node {

int value;

int min;

public int getValue() {

return value;

}

public int getMin() {

return min;

}

public void setMin(int min) {

this.min = min;

}

}

n the previous approach, the main drawback is that in each node, they have to store the min value, it costs too many spaces. As the minimum value may stay the same if no larger element appears, so we can improve the algorithm above by removing duplicated min values.

public class StackWithMin {

Stack<Integer> stack1 = new Stack<Integer>();

Stack<Integer> stack2 = new Stack<Integer>();

public void push(int value) {

if (value <= min()) {

stack2.push(value);

}

stack1.push(value);

}

public int pop() {

if (stack1.peek() == stack2.peek() ) {

stack2.pop();

}

return stack1.pop();

}

public int min() {

if (stack2.peek() == null) {

return MAX\_INT;

} else {

return stack2.peek();

}

}

}

## 判断字符串可否通过另一个字符串的循环移位得到

http://blog.csdn.net/beiyeqingteng/article/details/6919788

这里，我们可以发现，实际对s1的循环移位得到的字符串实际为s1s1。

那么我们判断s2是否可以通过s1循环移位得到包含，则只需要判断s1s1中是否含有s2即可以。

用提高空间复杂度来换取时间复杂度的减低的目的。

public class StrigMove{

public static void main (String[] args) {

String s1="AABCD";

String s2="CDAA";

if(s1.concat(s1).indexOf(s2)!=-1){

System.out.println ("true");

}else{

System.out.println ("false");

}

}

}

## 找出丢失的数字（思维经典）

http://blog.csdn.net/beiyeqingteng/article/details/6934234

## 两个巨大正整数相加

http://blog.csdn.net/beiyeqingteng/article/details/6950679

两个巨大整数相加，可能会造成溢出，或者它的大小已经超出基本数据类型的范围，所以，我们对巨大整数进行相加时，可以把它们转换成字符串，然后通过字符串的处理进行整数相加。

这里有两种做法：第一种，把整数存在一个字符数组里进行处理。代码如下：

public static String addThroughArray(String a, String b) {

char[] adder = a.toCharArray();

char[] addee = b.toCharArray();

StringBuffer result = new StringBuffer();

int N = Math.max(adder.length, addee.length);

int r = 0;

for (int i = 0; i < N; i++) {

int bit1 = adder.length > i ? adder[adder.length - i - 1] - '0' : 0;

int bit2 = addee.length > i ? addee[addee.length - i - 1] - '0' : 0;

int sum = r + bit1 + bit2;

result.append(sum%10);

r = sum/10;

}

//important. we need to consider the last addition.

if (r > 0) {

result.append(r);

}

return result.reverse().toString();

}

第二种方法使用StringBuffer. 代码如下：

public static String add(String a, String b) {

StringBuffer c = new StringBuffer();

StringBuffer sa = new StringBuffer(a).reverse();

StringBuffer sb = new StringBuffer(b).reverse();

int N = Math.max(sa.length(), sb.length());

int r = 0;

for (int i = 0; i < N; i++) {

int bit1 = sa.length() > i ? sa.charAt(i)-'0' : 0;

int bit2 = sb.length() > i ? sb.charAt(i)-'0' : 0;

int sum = r + bit1 + bit2;

c.append(sum%10);

r = sum/10;

}

if (r > 0) c.append(r);

return c.reverse().toString();

}

## rand5()构造rand7()

http://blog.csdn.net/beiyeqingteng/article/details/6951622

给一个方法，比如 rand5(), 它能够等概率生成 1-5 之间的整数。 所谓等概率就是1,2,3,4,5 生产的概率均为 0.2 。现在利用rand5(), 构造一个能够等概率生成 1- 7 的方法。

这里有两个特别重要的点，一是 如果 rand5() + rand5(), 我们能够产生一个均匀分布的 1 - 10 吗？ 答案是否定的。比如对于 6来讲（4+2， 2+4， 3+3），它被生成的生成的概率比1 （1+0，0+1）要大.

第二个点就是我们不可能用rand5()直接产生 1- 7 的数，不管你用加减乘除都不行。

所以，我们要构造一个更大的范围，使得范围里每一个值被生成的概率是一样的，而且这个范围是7的倍数，代码如下：

int rand7() {

while (1) {

int rand = 5 \* (rand5() -1) ;

rand = rand + rand5() -1;

if (rand < 22 && ran > 0) {

return rand%7 + 1 ;

}

}

}

第三行代码产生一个均匀分布的 0， 5， 10， 15， 20的数。

第四行代码产生一个均匀分布的 0， 1， 2， 3， 4 的数。

相加以后，会产生一个 0到24的数，而且每个数（除0外）生成的概率是一样的。我们只取 1 - 21 这一段，和7 取余以后+1就能得到完全均匀分布的1-7的随机数了。

## 最长递增子序列 [No. 39]

http://blog.csdn.net/beiyeqingteng/article/details/6954571

给定一个长度为n的数组，找出一个最长的单调递增子序列（不一定连续，当时先后顺序不能乱）。 更正式的定义是：

设L=<a1,a2,…,an>是n个不同的实数的序列，L的递增子序列是这样一个子序列Lin=<ak1,ak2,…,akm>，其中k1<k2<…<km且ak1<ak2<…<akm。求最大的m值。

比如数组A 为{10， 11， 12， 13， 1， 2， 3， 15}， 那么最长递增子序列为{10,11,12,13,15}。

这道题用DP解，假设从第1位到第 j 位的最长的子序列长度为L(j)，那么它一定是 L(j) = max\_{i < j and A[i] <= A[j]} L(i) + 1. 然后我们遍历整个 L(j), 找出最大的长度的位置。整个操作的复杂度为 O(n^2).

代码如下：

public int LIS(int[] arr) {

// array a is used to store the maximum length

int[] a = new int[arr.length];

// initialization, set all the elements in array a to 1;

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

a[i] = 1;

}

for (int i = 1; i < arr.length; i++) {

for (int j = 0; j < i; j++ ) {

if (arr[i] > arr[j] && a[i] <= a[j] ) {

a[i] = a[j] + 1;

}

}

}

int max = -1;

//find the longest subsequence

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

if (a[i] > max ) max = a[i];

}

return max;

}

## 不用＋、－、×、÷数字运算符做加法

http://blog.csdn.net/beiyeqingteng/article/details/6958373

第一步不考虑进位，对每一位相加。0加0与1加1的结果都0，0加1与1加0的结果都是1。我们可以注意到，这和异或的结果是一样的。对异或而言，0和0、1和1异或的结果是0，而0和1、1和0的异或结果是1。接着考虑第二步进位，对0加0、0加1、1加0而言，都不会产生进位，只有1加1时，会向前产生一个进位。此时我们可以想象成是两个数先做位与运算，然后再向左移动一位。只有两个数都是1的时候，位与得到的结果是1，其余都是0。第三步把前两个步骤的结果相加。如果我们定义一个函数AddWithoutArithmetic，第三步就相当于输入前两步骤的结果来递归调用自己。

int AddWithoutArithmetic(int num1, int num2)

{

if(num2 == 0)

return num1;

int sum = num1 ^ num2;

int carry = (num1 & num2) << 1;

return AddWithoutArithmetic(sum, carry);

}

## 判断一个单链表是否有环，如果有，找出环的起始位置

http://blog.csdn.net/beiyeqingteng/article/details/6959617

public boolean hasLoop(Node head) {

if (head == null) return false;

//slow pointer

Node sPointer = head.next();

if (sPointer == null) return false;

//fast pointer

Node fPointer = sPointer.next();

while (sPointer != null && fPointer != null) {

if (sPointer == fPointer) return true;

sPointer = sPointer.next();

fPointer = fPointer.next();

if (fPointer != null) {

fPointer = fPointer.next();

}

}

return false;

}

当我们知道这个链表有环了，那么找出起始位置就比较简单。

/\* (Step 1) Find the meeting point. This algorithm moves two pointers at

\* different speeds: one moves forward by 1 node, the other by 2. They

\* must meet (why? Think about two cars driving on a track—the faster car

\* will always pass the slower one!). If the loop starts k nodes after the

\* start of the list, they will meet at k nodes from the start of the

\* loop. \*/

n1 = n2 = head;

while (TRUE) {

n1 = n1->next;

n2 = n2->next->next;

if (n1 == n2) {

break;

}

}

// Find the start of the loop.

n1 = head;

while (n1 != n2) {

n1 = n1->next;

n2 = n2->next;

}

Now n2 points to the start of the loop.

分析：上面的代码为何能够找到环的起始位置？

假设环的长度是 m, 进入环前经历的node的个数是 k , 那么，假设经过了时间 t，那么速度为2 的指针距离起始点的位置是: k + (2t - k) % m = k + (2t - k) - xm . 同理，速度为1的指针距离起始点的位置是 k + (t - k) % m = k + (t - k) - ym。

如果 k + (2t - k) - xm = k + (t - k) - ym ，可以得到 t = m (x - y)。 那么当t 最小为m的时候，也就是说，两个指针相聚在 距离 起始点 m - k的环内。换句话说，如果把一个指针移到链表的头部，然后两个指针都以 1 的速度前进，那么它们经过 k 时间后，就可以在环的起始点相遇。

## pow(double a,int b)

http://blog.csdn.net/beiyeqingteng/article/details/6973072

double pow(double a,int b)

{

if(b == 0) return 1;

if(b == 1) return a;

double temp = pow(a,b/2);

temp = temp\*temp;

return ((b % 2 == 0) ? temp : temp\*a);

}

上述代码不完整

备注：如果 b 可以是负数，则要考虑 a 是情况。

## 找出两个已经排好序的数组的中位数

http://blog.csdn.net/beiyeqingteng/article/details/6992685

问题：

给两个已经排好序的数组，一个长度为 m (m >= 1), 一个长度为 n (n >= 1)，找出这两个数组的中位数。时间复杂度要求为 O(m+n), 空间复杂度为 O(1)。

其实这个问题本身来讲是不难的，关键的关键，是对一些边界条件的处理。

思路：

我们首先不管两个数组里含有多少个数字，而是先得到第 (m+n+1)/2个数字的值。 如果 m + n为奇数，我们返回第 (m+n+1)/2，如果m + n为偶数，那么我们找出第 (m+n+2)/2个数字, 然后求平均。

public class Med {

public static float getMedian(int[] a, int[] b) {

if (a == null || b == null || (a.length + b.length) == 0) return Float.MIN\_VALUE;

int pa = 0;

int pb = 0;

float median = 0;

while (pa + pb != (a.length + b.length + 1) / 2) {

int Ai = (pa == a.length) ? Integer.MAX\_VALUE : a[pa];

int Bj = (pb == b.length) ? Integer.MAX\_VALUE : b[pb];

if (Ai < Bj) {

median = a[pa];

pa++;

} else {

median = b[pb];

pb++;

}

}

if ((a.length + b.length) % 2 == 1) {

return median;

} else {

int Ai = (pa == a.length) ? Integer.MAX\_VALUE : a[pa];

int Bj = (pb == b.length) ? Integer.MAX\_VALUE : b[pb];

int median2 = (Ai < Bj) ? Ai : Bj;

return (float)(median + median2) / 2;

}

}

public static void main(String[] args) {

int[] b = {1, 2};

int[] a = {5, 6};

System.out.println(getMedian(a,b));

}

}

## 一个数组先从小到大递增后从大到小递减，找出最大的值

http://blog.csdn.net/beiyeqingteng/article/details/6995092

思路：最简单的办法就是从第二个值开始，判断是否满足 A[i] > A[i-1] && A[i] > A[i+1]. 如果满足，i 就是那个最大值的下标。该算法复杂度为O(n).

我们可以改进这种算法，因为这个数组是排好序的，所以我们可以利用二分查找的思想，更快速的找到最大值，时间复杂度为O(lg n)。

二分查找的算法可以在后面的link处得到，二分查找的退出条件和这道题的退出条件是不一样的（详见while loop），这是很值得注意的地方。

public int turningPoint(int[] A) {

int m = A.length;

int begin = 0;

int end = m - 1;

int tp = begin + (end - begin)/2;

// the condition "tp > 0 && tp < m -1" makes sure that tp is not at the beginning or the end

while (tp > 0 && tp < m -1) {

if (A[tp] > A[tp + 1] && A[tp] > A[tp -1]) {

return tp;

} else if (A[tp] < A[tp+1]) {

begin = tp + 1;

tp = begin + (end - begin)/2;

} else {

end = tp - 1;

tp = begin + (end - begin)/2;

}

}

return -1;

}

## 一个已经排序的数组，去除里面重复的元素

http://blog.csdn.net/beiyeqingteng/article/details/6995988

问题：

对于一个已经排好序的数组，去除里面重复的元素，比如A = {1,2,2,2,3,3,4,4}， 去掉重复以后，就变成A = {1,2,3,4}.

原理非常简单，关键是写代码的时候注意“指针”位置，和如何比较重复，参见代码里的第一个for循环。

public int[] uniqueArray(int[] array) {

if (array.length == 0) return null;

if (array.length == 1) return array;

int pointer = 0;

for (int i = 1; i < array.length; i++) {

if (array[i] != array[pointer]) {

pointer++;

array[pointer] = array[i];

}

}

//copy the data to another array

int[] uniArray = new int[pointer+1];

for (int i = 0; i <= pointer; i++) {

uniArray[i] = array[i];

}

return uniArray;

}

## 把一个字符串转成一个整数

http://blog.csdn.net/beiyeqingteng/article/details/7000034

其实，这道题考的不是怎么去把一个数转成一个整数，而是考你是否能够把所有的情况都考虑完全，我们应该考虑的因素如下：

1. 这个字符串是否是空的。

2. 这个字符串是否有非法字符（非0-9之间的字符）。

3.这个数是正数或者是负数的情况（第一个字符是否为+，-）。

4. 是否存在溢出的情况（这个比较难考虑到）。

public static long atoi(String str) throws Exception {

boolean negative = false;

long value = 0;

if (str == null || str.equals("")) {

throw new Exception("null string or the string has no character!");

}

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

if (i == 0 && (str.charAt(0) == '-' || str.charAt(0) == '+')) {

if (str.charAt(0) == '-') {

negative = true;

}

} else {

if (str.charAt(i) >= '0' && '9' >= str.charAt(i)) {

value = value \* 10 + (str.charAt(i) - '0');

if (value > Integer.MAX\_VALUE) {

throw new Exception("OUT OF INTEGER RANGE");

}

} else {

throw new NumberFormatException("not an integer");

}

}

}

return negative == true ? value \* -1 : value;

}

## “I love china”反转后变成 “china love I”

http://blog.csdn.net/beiyeqingteng/article/details/7006441

public class StringReverse {

public void swap(char[] arr, int begin, int end) {

while(begin < end) {

char temp = arr[begin];

arr[begin] = arr[end];

arr[end] = temp;

begin++;

end--;

}

}

public String swapWords(String str) {

char[] arr = str.toCharArray();

swap(arr, 0, arr.length - 1);

int begin = 0;

for (int i = 1; i < arr.length; i++) {

if (arr[i] == ' ') {

swap(arr, begin, i - 1);

begin = i + 1;

}

}

swap(arr, begin, arr.length - 1);

return new String(arr);

}

public static void main(String[] args) {

String str = "I";

System.out.println(new StringReverse().swapWords(str));

}

}

## 反转单向链表 (单个反转)

http://blog.csdn.net/beiyeqingteng/article/details/7030020

给一个单向链表，把它从头到尾反转过来。比如： a -> b -> c ->d 反过来就是 d -> c -> b -> a 。

这里讲解两种方法：

第一种方法就是把每个Node按照顺序存入到一个stack里面，这样，最后面一个就在最上面了。然后，把每一个再取出来，这样顺序就换过来了。

public static Node reverse(Node head) {

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

// put all the nodes into the stack

while (head != null) {

stack.add(head);

head = head.next();

}

//reverse the linked list

Node current = stack.pop();

head = current;

while (stack.empty() != true) {

Node next = stack.pop();

//set the pointer to null, so the last node will not point to the first node.

next.setNext(null);

current.setNext(next);

current = next;

}

return head;

}

第二种方法就是利用两个指针，分别指向前一个节点和当前节点，每次做完当前节点和下一个节点的反转后，把两个节点往下移，直到到达最后节点。

public static Node reverse(Node head) {

Node previous = null;

while (head != null&&head!=null) {

Node nextNode = head.next();

head.setNext(previous);

previous = head;

head = nextNode;

}

head.setNext(previous);

return previous;

}

## 两两反转单向链表

http://blog.csdn.net/beiyeqingteng/article/details/7049242

两两反转单项链表就是把每两个数反转一次。如下：

A -> B -> C ->D -> E -> F ->G -> H -> I 两两反转后变为 B -> A -> D ->C -> F -> E ->H -> G -> I

public static Node reverseInPair(Node current) {

if (current == null || current.next == null) return current;

Node head = current.next;//save the head of the list

Node previousNode = null;

while(current != null && current.next != null) {

//get the current node's next and "nextnext" node

Node nextNode = current.next;

Node nextNextNode = nextNode.next;

//exchange the "next" node

nextNode.next = current;

current.next = nextNextNode;

//update the "next" value of the previous node

if (previousNode != null) {

previousNode.next = nextNode;

}

previousNode = current;

current = nextNextNode;

}

return head;

}

## 格雷码的实现

http://blog.csdn.net/beiyeqingteng/article/details/7044471

问题：产生n位元的所有格雷码。

格雷码(Gray Code)是一个数列集合，每个数使用二进位来表示，假设使用n位元来表示每个数字，任两个数之间只有一个位元值不同。

例如以下为3位元的格雷码： 000 001 011 010 110 111 101 100 。

如果要产生n位元的格雷码，那么格雷码的个数为2^n.

第一步：产生 0, 1 两个字符串。

第二步：在第一步的基础上，每一个字符串都加上0和1，但是每次只能加一个，所以得做两次。这样就变成了 00,01,11,10 （注意对称）。

第三步：在第二步的基础上，再给每个字符串都加上0和1，同样，每次只能加一个，这样就变成了 000,001,011,010,110,111,101,100。

好了，这样就把3位元格雷码生成好了。

如果要生成4位元格雷码，我们只需要在3位元格雷码上再加一层0,1就可以了： 0000,0001,0011,0010,0110,0111,0101,0100,1100,1101,1110,1010,0111,1001,1000.

也就是说，n位元格雷码是基于n-1位元格雷码产生的。

如果能够理解上面的部分，下面部分的代码实现就很容易理解了。

public String[] GrayCode(int n) {

// produce 2^n grade codes

String[] graycode = new String[(int) Math.pow(2, n)];

if (n == 1) {

graycode[0] = "0";

graycode[1] = "1";

return graycode;

}

String[] last = GrayCode(n - 1);

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

graycode[i] = "0" + last[i];

graycode[graycode.length - 1 - i] = "1" + last[i];

}

return graycode;

}

格雷码还有一种实现方式是根据这个公式来的 G(n) = B(n) XOR B(n+1), 这也是格雷码和二进制码的转换公式。代码如下：

public void getGrayCode(int bitNum){

for(int i = 0; i < (int)Math.pow(2, bitNum); i++){

int grayCode = (i >> 1) ^ i;

System.out.println(num2Binary(grayCode, bitNum));

}

}

public String num2Binary(int num, int bitNum){

String ret = "";

for(int i = bitNum-1; i >= 0; i--){

ret += (num >> i) & 1;

}

return ret;

}

## 字符串排列和组合的JAVA实现

http://blog.csdn.net/beiyeqingteng/article/details/7051942

字符串的组合：

给一个字符串，比如ABC， 把所有的组合，即：A, B, C, AB, AC, BC, ABC, 都找出来。

解题思路：

假设我们想在长度为n的字符串中求m个字符的组合。我们先从头扫描字符串的第一个字符。针对第一个字符，我们有两种选择：一是把这个字符放到组合中去，接下来我们需要在剩下的n-1个字符中选取m-1个字符；二是不把这个字符放到组合中去，接下来我们需要在剩下的n-1个字符中选择m个字符。这两种选择都很容易用递归实现。

public static void combiantion(char chs[]){

if(chs.length == 0) return ;

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

for(int i = 1; i <= chs.length; i++){

combine(chs, 0, i, stack);

}

}

//从字符数组中第begin个字符开始挑选number个字符加入list中

public static void combine(char []chs, int begin, int number, Stack<Character> stack){

if(number == 0){

System.out.println(stack.toString());

return ;

}

if(begin == chs.length){

return;

}

stack.push(chs[begin]);

combine(chs, begin + 1, number - 1, stack);

stack.pop();

combine(chs, begin + 1, number, stack);

}

给一个字符串，比如ABC， 把所有的排列，即：ABC, ACB, BAC, BCA, CAB, CBC 都找出来。

解题思路：

对于一个n 位的字符串来讲，它是n-1位字符串的排列 加上 没有在 n -1 位字符串里 那个字符 的排列。 有点难理解，用例子说明：

对于字符串ABC来讲，它所有的排列就是 A + BC 的排列 加上 B + AC 的排列，再加上 C + AB的排列。

而BC的排列是 B + C 的排列 加上 C + B 的排列。

所以，对一个字符串，我们从中去一个值，然后求剩余部分的排列，然后把它们再组合在一起。所有，代码如下：

// str has been sorted.

public static LinkedList<String> permutation(String str) {

LinkedList<String> linkedString = new LinkedList<String>();

if (str.length() <= 1) {

linkedString.add(str);

return linkedString;

}

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

char ch = str.charAt(i);

//consider the case in which the characters may be duplicated.

if (i > 0 && ch == str.charAt(i - 1)) {

continue;

}

String newStr = remove(str, i);

LinkedList<String> newStrList= permutation(newStr);

for (int j = 0; j < newStrList.size(); j++) {

linkedString.add(ch + newStrList.get(j));

}

}

return linkedString;

}

//remove the ith character from the string

public static String remove(String str, int i) {

if (i == 0) return str.substring(1, str.length());

if (i == str.length() - 1) return str.substring(0, i );

return str.substring(0, i) + str.substring(i + 1, str.length());

}

**这里贴出另一个permutation 的版本：**

static void swap(char[] arr, int idx1, int idx2) {

char temp = arr[idx1];

arr[idx1] = arr[idx2];

arr[idx2] = temp;

}

static void permutation(char[] arr, int index, int size) {

if (index == size) {

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

System.out.print(arr[i] + " ");

}

System.out.println();

} else {

for (int i = index; i < size; i++) {

if(i != index && arr[i] == arr[index])

continue;

swap(arr, i, index);

permutation(arr, index+1, size);

swap(arr, i, index);

}

}

}

}

## 顺时针打印矩阵 （JAVA实现）

http://blog.csdn.net/beiyeqingteng/article/details/7055499

一个矩阵，给定起点（startX，startY）和终点（endX，endY）（即位于对角线上的两个点）就可以打印一周，然后向里进一周（即++startX,++startY,--endX,--endY）即可，如果起始点坐标<终止点坐标（即startX<endX或者startY<endY）循环结束。

给定起点和终点，打印一周的结束条件是什么？我画了三种情况的矩阵4×4矩阵，3×5矩阵，5×3矩阵（所有矩阵无非就这三种类型，正方形的，偏“胖”的，偏“瘦”的），很快发现，只有三种情况：一直循环到结束，只剩下一行，只剩下一列。所以我们的函数首先判定：只有一行？打印该行；只有一列，打印该列。都不是，打印四条边上的数字。

好了到这里为止：我们可以写出完整的代码了，如下：

public void printCircle(int[][] matrix, int startX, int startY, int endX, int endY) {

// only one row left

if (startX == endX) {

for (int i = startY; i <= endY; i++ ) {

System.out.println(matrix[startX][i]);

}

return;

}

// only one column left

if (startY == endY) {

for (int i = startX; i <= endX; i++ ) {

System.out.println(matrix[i][endY]);

}

return;

}

for (int i = startY; i < endY; i++ ) {

System.out.println(matrix[startX][i]);

}

for (int i = startX; i < endX; i++ ) {

System.out.println(matrix[i][endY]);

}

for (int i = endY; i > startY; i-- ) {

System.out.println(matrix[endX][i]);

}

for (int i = endX; i > startX; i-- ) {

System.out.println(matrix[i][startY]);

}

}

public void printMatrix(int[][] matrix) {

if (matrix == null) {

return;

}

int startX = 0;

int startY = 0;

int endY = matrix[0].length - 1;

int endX = matrix.length - 1;

while ((startX <= endX) && (startY <= endY)) {

printCircle(matrix, startX, startY, endX, endY);

startX++;

startY++;

endX--;

endY--;

}

}

## 最大子矩阵和

http://blog.csdn.net/beiyeqingteng/article/details/7056687

## 最近公共祖先

http://blog.csdn.net/beiyeqingteng/article/details/7058849

我们这里先考虑一般的二叉树(BT)，然后再考虑这个二叉树是二叉搜索树(BST)的情况。

查找两个node的最早的公共祖先，分三种情况：

1. 如果两个node在root的两边，那么最早的公共祖先就是root。

2. 如果两个node在root的左边，那么把root.leftChild作为root，再递归。

3. 如果两个node在root的右边，那么把root.rightChild作为root，再递归。

那么我们如何知道能否通过原始节点到达某一个节点呢？这里我们需要定义一个递归函数covers (Node root, Node node)，让root的左右子节点不断的调用这个函数，如果某一个子节点就是要找到的节点，那么返回true，否则返回false. 具体代码如下：

/\*

\* check whether the node n is in the tree

\*/

private static boolean covers(Node rootNode, Node n) {

if(rootNode == null) return false;

if(rootNode == n) return true;

return covers(rootNode.leftChild, n) || covers(rootNode.rightChild, n);

}

有个covers这个函数，我们要实现前面三种不同的情况，那就简单了。代码如下：

/\*

\* get the first common ancestor of node p and node q

\*/

public static Node commonAncestor(Node rootNode, Node p, Node q) {

// case 2

if (covers(rootNode.leftChild, p) && covers(rootNode.leftChild, q))

return commonAncestor(rootNode.leftChild, p, q);

//case 3

if (covers(rootNode.rightChild, p) && covers(rootNode.rightChild, q))

return commonAncestor(rootNode.rightChild, p, q);

//case 1

return rootNode;

}

如果这个二叉树是BST，那么我们可以利用BST的特点，把根节点的值与两个节点的值进行比较，如果两个节点的值都比根节点的值小，那么一定在根节点的左边，所以我们把根节点的左子节点作为起始点，然后递归。如果两个节点的值都比祖先节点的值大，那么一定在根节点的右边，所以我们把根节点的右子节点作为起始点，然后递归，如果上面两张情况都不是，那么很明显，这个根节点就是LCA. 代码如下：

public Node LCA(Node root, Node p, Node q) {

if (root == null || p == null || q == null ) return NULL;

if (max(p.data, q.data) < root.data)

return LCA(root.left, p, q);

else if (min(p.data, q.data) > root.data)

return LCA(root.right, p, q);

else

return root;

}

备注：这里并没有考虑 p 和 q 不在 BST里的情况。

## 判断一个树是不是二叉查询树

http://blog.csdn.net/beiyeqingteng/article/details/7059611

private boolean isBST(Node node) {

if (node==null) return(true);

if (node.left!=null && maxValue(node.left) > node.data) return(false);

if (node.right!=null && minValue(node.right) <= node.data) return(false);

// check that the subtrees themselves are ok

return( isBST(node.left) && isBST(node.right) );

}

这个算法的复杂度是 O(n \* h)。原因是每做一次递归（向下走），我们都要去查找最大值和最小值。

其实，我们可以不用每次都去查找最大值和最小值，每次向下递归时，我们只要把该节点的值， 作为一个最大值， 传给它的左节点，也就是左边所有节点的值都要比它小；并且把它的值，作为最小值，传给它的右节点，也就是右边所有节点的值都要比它大。每次向下走，分别更新最大值和最小值即可。

bool isBSTHelper(Node p, int low, int high) {

if (p == null) return true;

if (low < p.data && p.data < high)

return isBSTHelper(p.leftChild, low, p.data) &&

isBSTHelper(p.rightChild, p.data, high);

else

return false;

}

bool isBST(Node root) {

return isBSTHelper(root, INT\_MIN, INT\_MAX);

}

这样，算法的复杂度就降为 O(n)。

## 打印二叉树所有的路径

http://blog.csdn.net/beiyeqingteng/article/details/7060642

思路：

从根节点开始，把自己的值放在一个数组里，然后把这个数组传给它的子节点，子节点同样把自己的值放在这个数组里，又传给自己的子节点，直到这个节点是叶节点，然后把这个数组打印出来。所以，我们这里要用到递归。

/\*\*

Given a binary tree, prints out all of its root-to-leaf

paths, one per line. Uses a recursive helper to do the work.

\*/

public void printPaths(Node root, int n) {

String[] path = new String[n];

printPaths(root, path, 0);

}

/\*\*

Recursive printPaths helper -- given a node, and an array containing

the path from the root node up to but not including this node,

prints out all the root-leaf paths.

\*/

private void printPaths(Node node, String[] path, int pathLen) {

if (node == null) return;

// append this node to the path array

path[pathLen++] = node.value;

// it's a leaf, so print the path that led to here

if (node.leftChild == null && node.rightChild == null) {

printArray(path, pathLen);

}

else {

// otherwise try both subtrees

printPaths(node.leftChild, path, pathLen);

printPaths(node.rightChild, path, pathLen);

}

}

/\*\*

Utility that prints strings from an array on one line.

\*/

private void printArray(String[] ints, int len) {

for (int i = 0; i < len; i++) {

System.out.print(ints[i] + " ");

}

System.out.println();

}

## 检查一个数组是否可能是一棵二叉查询树后序遍历的结果

http://blog.csdn.net/beiyeqingteng/article/details/7070024

思路：

因为二叉树后序遍历的时候，先左后右然后中间。所以，数组的最后一个一定是整个二叉查询树的根（比如 8），而且，数组前一部分比数组最后一个值（也就是二叉查询树的根）小，因为它们是根的左子树部分。数组剩余部分，是根的右子树部分。而且，在剩余部分里，所有的值都必须比根要大。这样，我们可以把原来的数组分成两个部分，然后每一个部分可以进行递归判断，直到数组的长度小于等于2。（因为当长度小于等于2时，什么样的情况都是可以的）。

我们先把给定的数组分成两个部分，即我们需要先找到数组的分界点。代码如下：

/\*\*

\*

\* @param array the input array

\* @param begin the starting index of the array

\* @param end the ending index of the array

\* @return the turning point of the array, where the left part of turning point is smaller than array[end], the right

\* part of the turning point is larger than array[end].

\*/

public int turningPoint(int[] array, int begin, int end) {

int tp = -1;

for (int i = begin; i < end; i++) {

if (array[i] > array[end]) {

tp = i;

break;

}

}

return tp;

}

找到分界点后，我们还要判断从分界点开始，到数组末，是否所有的值都比数组的最后一个值（也就是根）大，否则，这样的数组不是一个二叉查询树的后序遍历。代码如下：

//check whether all the elements from strat are larger than array[end]

public boolean checkValid(int[] array, int start, int end) {

for (int i = start; i < end; i++) {

if (array[i] < array[end]) return false;

}

return true;

}

有了上面两个方法，我们就可以把递归写出来了，代码如下：

// the main method to check whether the array is identical to a BST's post order array

public boolean isPostOrder(int[] array, int begin, int end) {

// the exit

if ((end - begin + 1) <= 2) return true;

//get the turning point

int turningPoint = turningPoint(array, begin, end);

boolean result = checkValid(array, turningPoint, end); //check whether the right part of the array is valid

if (result == false) {

return false;

}

// recursion

return isPostOrder(array, begin, turningPoint - 1) && isPostOrder(array, turningPoint, end - 1);

}

## 由A转成B所需的最少编辑操作次数

http://blog.csdn.net/beiyeqingteng/article/details/7075962

## 左旋转字符串

http://blog.csdn.net/beiyeqingteng/article/details/7090051

public class Rotate {

/\*\*

\*

\* @param array: the array to be rotated

\* @param begin: the rotation starting point

\* @param end: the rotation ending point

\*/

public static void rotate(char[] array, int begin, int end) {

while (begin < end) {

char temp = array[begin];

array[begin] = array[end];

array[end] = temp;

begin++;

end--;

}

}

public static void leftRotate(char[] test, int number) {

// do x^T

rotate(test, 0, number - 1);

// do y^T

rotate(test, number, test.length - 1);

//do (x^T y^T)^T

rotate(test, 0, test.length - 1);

}

public static void main(String[] args) {

char[] test = {'a','b','c','d','e','f','g'};

int number = 4;

leftRotate(test, number);

}

}

## 找出字符串中对称的子字符串的最大长度(最长回文)

http://blog.csdn.net/beiyeqingteng/article/details/7097146

背景：

所谓对称子字符串，就是这个子字符串要么是以其中一个词对称：比如 “aba”， “abcba”；要么就完全对称：比如"abba", "abccba"。

问题：

给你一个字符串，找出该字符串中对称的子字符串的最大长度。

思路：

首先，我们用字符数组 char[] array 来保持这个字符串，假设现在已经遍历到第 i 个字符，要找出以该字符为“中心”的最长对称字符串，我们需要用另两个指针分别向前和向后移动，直到指针到达字符串两端或者两个指针所指的字符不相等。因为对称子字符串有两种情况，所以需要写出两种情况下的代码：

1. 第 i 个字符是该对称字符串的真正的中心，也就是说该对称字符串以第 i 个字符对称， 比如： “aba”。代码里用 index 来代表 i.

public static int maxLengthMiddle(char[] array, int index) {

int length = 1; //最长的子字符串长度

int j = 1; //前后移动的指针

while ((array[index - j] == array[index + j]) && (index - j) >= 0 && array.length > (index + j)) {

length += 2;

j++;

}

return length;

}

2. 第 i 个字符串是对称字符串的其中一个中心。比如“abba”。

public static int maxLengthMirror(char[] array, int index) {

int length = 0; //最长的子字符串长度

int j = 0; //前后移动的指针

while ((array[index - j] == array[index + j + 1]) && (index - j) >= 0 && array.length > (index + j + 1)){

length += 2;

j++;

}

return length;

}

有了这样两个函数，我们只需要遍历字符串里所有的字符，就可以找出最大长度的对称子字符串了。

public static int palindrain(char[] array) {

if (array.length == 0) return 0;

int maxLength = 0;

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

int tempMaxLength = - 1;

int length1 = maxLengthMiddle(array, i);

int length2 = maxLengthMirror(array, i);

tempMaxLength = (length1 > length2) ? length1 : length2;

if (tempMaxLength > maxLength) {

maxLength = tempMaxLength;

}

}

return maxLength;

}

因为找出以第 i 个字符为“中心”对称字符串复杂度为 O（N），所以整个算法的复杂度为O（N^2）。

有一种可以把复杂度降到O（N）的算法，但是这个算法要利用 suffix tree

## 数组中出现次数超过一半的数字

http://blog.csdn.net/beiyeqingteng/article/details/7110749

问题：

给定一个 int 数组，找出里面出现次数超过一半的数字。

分析：

首先，我们假定该数组中存在超过一半的数字，要把那个数字找出来，我们有以下方法：

1. 排序。出现次数最大的那个数字一定是排序后第 n/2 + 1 那个值(假定是从1 开始数)。复杂度为O(nlgn).

2. 遍历一遍数组，找出数组里的最大值max和最小值min，然后创建一个大小为 max - min + 1的数组，把每一个出现的值所对应的位置加1。比如数组里最大值是1000，最小值是-100，那么我们创建一个1101大小的数组 A，如果原数组的值是50，我们就把 A[50 - (-100)] 的值 加 1。 这样，我们可以得到所有值出现的次数。但是，这样做的话，空间消耗有可能会特别大。复杂度为O(n)。

3. 本文算法：我们保存两个值，一个是value --- 当前出现次数最大的值， 一个是 count ---value出现的次数。我们遍历整个数组，如果遇到与value值相同的值，就把count 加1，否则减1。如果在做减法以前，count的值已经为0，就把那个不同的值设成value， count的值设成 1. 这样，我们保证能够找出出现次数超过一半的数字 value。

代码如下：

public int overHalf(int[] array) throws Exception {

//alert! array == null is different from array.length == 0

if (array == null || array.length == 0) {

throw new Exception("null array or the array has no element");

}

int value = array[0];

int count = 1;

for(int i = 1; i < array.length; i++) {

// the same

if (array[i] == value) {

count++;

}

//not the same

else {

//count is equal to 0

if (count == 0) {

value = array[i];

count = 1;

} else {

count--;

}

}

}

// check whether the array is valid

int total = 0;

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

if (array[i] == value) total++;

}

if (total \* 2 <= array.length) {

throw new Exception(" the array is not valid!");

}

return value;

}

代码说明：这里，我们假设的是数组里面存在超过一半的数字，但是，我们需要对数组检查是否真的存在这样的值，代码的后部分就是做相应的检查。

## 输入一个正数 n，输出所有和为 n 的连续正数序列

http://blog.csdn.net/beiyeqingteng/article/details/7167268

问题：

输入一个正数n，输出所有和为 n 的连续正数序列。例如：输入15，由于1+2+3+4+5 = 4+5+6 = 7+8=15，所以输出3个连续序列1-5、4-6和7-8。

分析：

首先，假设我们是从6开始判断的，因为 6 比 15 小， 而且由于要求序列是连续的，所以，下一次只能选择5，剩余的值是 15 - 6 = 9，因为5 比9 小，我们只能再继续下去，这次选择是 4， 而剩余的值是 9 - 5 = 4。因为剩余的值刚好等于当前值4，所以，我们在这里结束。

什么时候退出呢？只能在当前值比剩余值 大 或者 当前值已经比 1 小的情况下。

## 在数组里查找这样的数，它大于等于左侧所有数，小于等于右侧所有数

http://blog.csdn.net/beiyeqingteng/article/details/7167823

问题：

一个int数组, 比如 array[]，里面数据无任何限制，要求求出所有这样的数array[i]，其左边的数都小于等于它，右边的数都大于等于它。能否只用一个额外数组和少量其它空间实现。

分析：

最原始的方法是检查每一个数 array[i] ，看是否左边的数都小于等于它，右边的数都大于等于它。这样做的话，要找出所有这样的数，时间复杂度为O(N^2)。

其实可以有更简单的方法，我们使用额外数组，比如rightMin[]，来帮我们记录原始数组array[i]右边（包括自己）的最小值。假如原始数组为： array[] = {7, 10, 2, 6, 19, 22, 32}， 那么rightMin[] = {2, 2, 2, 6, 19, 22, 32}. 也就是说，7右边的最小值为2, 2右边的最小值也是2。

有了这样一个额外数组，当我们从头开始遍历原始数组时，我们保存一个当前最大值 max， 如果当前最大值刚好等于rightMin[i]， 那么这个最大值一定满足条件。还是刚才的例子。

第一个值是7，最大值也是7，因为7 不等于 2， 继续，

第二个值是10，最大值变成了10，但是10也不等于2，继续，

第三个值是2，最大值是10，但是10也不等于2，继续，

第四个值是6，最大值是10，但是10不等于6，继续，

第五个值是19，最大值变成了19，而且19也等于当前rightMin[4] = 19, 所以，满足条件。

如此继续下去，后面的几个都满足。

代码：

/\*\*

\* @author beiyeqingteng

\* @link http://blog.csdn.net/beiyeqingteng

\*/

public static void smallLarge(int[] array) throws Exception{

//the array's size must be larger than 2

if (array == null || array.length < 1) {

throw new Exception("the array is null or the array has no element!");

}

int[] rightMin = new int[array.length];

rightMin[array.length - 1] = array[array.length - 1];

//get the minimum value of the array[] from i to array.length - 1

for (int i = array.length - 2; i >= 0; i--) {

if (array[i] < rightMin[i + 1]) {

rightMin[i] = array[i];

} else {

rightMin[i] = rightMin[i + 1];

}

}

int leftMax = Integer.MIN\_VALUE;

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

if (leftMax <= array[i]) {

leftMax = array[i];

if (leftMax == rightMin[i]) {

System.out.println(leftMax);

}

}

}

}

## 只使用加法实现减法，乘法和除法 [# 68]

http://blog.csdn.net/beiyeqingteng/article/details/7170911

问题：

给你两个整数 a 和 b，仅仅使用加法，实现 a - b, a \* b, a / b。

分析：

对于减法，因为 a - b = a + (-1) \* b， 换句话说，我们只要得到 b 的相反数，就可以用加法实现 a - b。

对于乘法，a \* b 相当于是 对 |a| 做 |b| 次连加， 或者对 |b| 做 |a| 次连加，但是，这里我们必须要注意，a 和 b 都有可能是负数。

对于除法，a / b 相当于是让|b| 自己加自己，然后我们设置一个计数器，当|b| 加自己大于 |a| 的时候，计数器停。但是，这里有可能会带余数（除不尽），这里，假设我们考虑的是四舍五入的方法。

减法代码：

因为我们要对b 取反，所以首先实现取反的代码：

public static int negate(int b) {

int value = (b < 0 ? 1 : -1);

int negate = 0;

while (b!= 0) {

negate += value;

b += value;

}

return negate;

}

当对 b 取反以后， a + negate(b) 就实现 a - b 了。

public static int substract(int a, int b) {

return a + negate(b);

}

乘法代码：

因为我们需要对 |a| 进行 |b| 次连加，但是，我们不知道 a, b 的正负，所以，我们首先需要取得 a, b 的绝对值，然后再决定整个乘法最后值的正负。

取绝对值代码如下：

public int abs(int value) {

if (value >= 0 ) return value;

else return negate(value);

}

决定整个乘法的正负的代码：

public static boolean positive(int a, int b) {

if ((a > 0 && b > 0) || (a < 0 && b < 0)) return true;

else return false;

}

实现乘法的代码：

public static int times(int a, int b) {

if (abs(a) > abs(b)) return times(b, a); //faster

int result = 0;

for (int i = 1; i <= abs(a); i++) {

result += abs(b);

}

if (positive(a, b) == true) return result;

else return negate(result);

}

除法代码：

public static int divide(int a, int b) {

int count = 0;

for(int i = abs(a); i >= abs(b); i -= abs(b)) {

count++;

}

int remaining = substract(abs(a), times(abs(b), count));

if (times(remaining, 2) >= abs(b)) {

count++;

}

if (positive(a, b) == true) return count;

else return negate(count);

}

## aaabbbbc， 压缩后成为：a3b4c1

http://blog.csdn.net/beiyeqingteng/article/details/7172221

问题：

把字符串压缩，比如aaabbbbc， 压缩后成为：a3b4c1。

分析：

这题很简单，我们只需要从头到尾遍历一遍字符串即可。首先设置一个计数器count, 每次“指针移位”的时候，判断当前字符是否与前一个字符相等，如果相等，count++, 指针继续下移，否则，我们需要对前面已经遍历的字符串进行处理，然后重新初始化count，直到字符串遍历结束。这题的关键是对最后一个字符的处理。

public static String compress(char[] array) {

if (array == null || array.length == 0 ) return null;

int count = 1;

StringBuilder sb = new StringBuilder(); //save the compressed string

for (int index = 1; index < array.length; index++) {

if (array[index] == array[index - 1]) {

count++;

} else {

sb.append(array[index - 1]);

sb.append(count);

count = 1;

}

}

//important! add the last character to the stringbuilder.

sb.append(array[array.length - 1]);

sb.append(count);

return sb.toString();

}

## 迷宫问题

http://blog.csdn.net/beiyeqingteng/article/details/7214643

问题：

在一个n\*m的迷宫里,每一个坐标点有两种可能: 0 或 1。0表示该位置允许通过,1表示该位置不允许通过。从坐标(0,0)点出发，找出所有通往出口(n-1, m-1) 的路径。

如果我们用二维矩阵来表示，那么地图可以表示成：

0 0 0 0 0

1 0 1 0 1

0 0 0 01

0 1 0 0 0

0 0 0 1 0

在上面一个例子里，其中一条路径为0,0 -> 0,1 -> 0,2 -> 0,3 -> 1, 3 -> 2, 3 ->3, 3 -> 3, 4 -> 4,4

分析：

假设我们到了点 （i, j）， 如果这个点不是 1， 那么，我们可以继续向右边走，即到点（i, j+1），也可以往下走，即到点 （i+1, j）, 向左（i, j-1）,向上（i -1, j）。对于这样每一个点，什么时候不再“前进”了呢？

1、这个点已经超出了矩阵的范围；

2、这个点值为 1；

3、这个点是终点。

4、这个点已经被遍历了。

如果这个点是终点，我们把路径上经历过的点都打印出来，所以，我们需要一个存储所有点的一个数据结构，这里，我们使用ArrayList.

PS. 其实这个问题与打印一个二叉树所有路径的问题本质上是一模一样的

public class Maze {

public void findPath(int[][] array, ArrayList<String> visited, int length2, ArrayList<Point> list, int length, int i, int j) {

//exit

if (i >= array.length || j >= array[0].length || i < 0 || j < 0 || array[i][j] == 1 || visited(visited, length2, i, j)== true) {

return;

}

//add the point to list

list.add(length, new Point(i, j));

++length;

visited.add(length2, i+","+j);

++length2;

//we find the exit, and print all the points on the path

if (i == array.length - 1 && j == array[0].length - 1) {

print(list, length);

}

// the right direction

findPath(array, visited, length2, list, length, i, j+1);

// the down direction

findPath(array, visited, length2, list, length, i+1, j);

// the left direction

findPath(array, visited, length2, list, length, i, j-1);

// the up direction

findPath(array, visited, length2, list, length, i-1, j);

}

// check whether the point has been visited or not

public boolean visited(ArrayList<String> visited, int length2, int i, int j) {

String point = i+","+j;

for (int k = 0; k < length2; k++) {

if (visited.get(k).equals(point)) return true;

}

return false;

}

public void print(ArrayList<Point> list, int length) {

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

System.out.println(list.get(i).getX()+","+list.get(i).getY());

}

System.out.println("---------------------------");

}

public static void main(String[] args) {

int[][] array = {{0,0, 0}, {0,0, 0}};

Maze maze = new Maze();

ArrayList<Point> list = new ArrayList<Point>();

ArrayList<String> visited = new ArrayList<String>();

maze.findPath(array, visited, 0, list, 0, 0, 0);

}

}

class Point {

int x = 0;

int y = 0;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

}

扩展：

只找出所有路径里最短的一条？

答案：我们可以设置一个全局变量保存当前最短路径，只需要把所有可行的路径进行比较，我们就可以找到最短的路径

## 把手机键盘输入转化成短消息

http://blog.csdn.net/beiyeqingteng/article/details/7304383

手机键盘每个数字对应的字符如下：

0 ---> "0"；

1 ---> "1"；

2 --- > "ABC2"；

3 ---> "DEF3"；

4 ---> "GHI4"；

5 ---> "JKL5"；

6 ---> "MNO6"；

7 ---> "PQRS7"；

8 ---> "TUV8"；

9 ---> "WXYZ9"；

\* ---> 空格；

# ---> 断开；

比如，输入为

222 ---> C,

222222 ---> B

22233 ---> CE

222#33 ---> CE

222\*33 ---> C E

2#3\*4\*4 ---> AD G G

问题：

现在给你一串字符输入，把它转化成为短消息内容。

分析：

分析：

其实，这个问题只需要对三种情况进行处理：当字符为 #， 当字符为 \* 或者前后字符不一样的时候。只是处理的时候，需要小心，要注意一些特殊情况，比如特殊字符在前面：###33，最后一个是特殊字符：3\*\*\* 等。

public class PhonePad {

public static final String[] keypad = {"0", "1", "ABC2", "DEF3", "GHI4", "JKL5", "MNO6", "PQRS7", "TUV8", "WXYZ9"};

public static String translate(String input) {

if (input == null || input.length() == 0 || input.equals("\*") || input.equals("#")) return "";

int start = 0;

char preChar = ' '; //previous character

char curChar = ' '; //curent character

int times = - 1; // the number of times a specific character appears continuously

int value = -1;

int length = -1; // the length of a digit. i.e., the length of digit 5: "JKL5" is 4

String msg = "";

while (start < input.length()) {

curChar = input.charAt(start);

if (curChar == '#' || curChar == '\*' || (curChar != preChar && times != -1)) {

// deal with case that '#' and '\*' appear continuously

if (curChar == '#' && times == -1) {

start++;

continue;

}

if (curChar == '\*' && times == -1) {

msg = msg + " ";

start++;

continue;

}

value = preChar - '0';

length = keypad[value].length();

times = times % length; // the position of character

msg = msg + String.valueOf(keypad[value].charAt(times));

if (curChar == '\*') {

msg = msg + " ";

}

if (curChar >= '0' && curChar <= '9') {

times = 0;

preChar = curChar;

} else {

times = -1;

}

} else {

preChar = curChar;

times++;

}

start++;

}

//deal with the last character

if (times != -1) {

value = curChar - '0';

length = keypad[value].length();

times = times % length;

msg = msg + String.valueOf(keypad[value].charAt(times));

}

return msg;

}

public static void main(String[] args) {

System.out.println(PhonePad.translate("##\*2#211\*\*2\*"));

}

}

## 把一个字符串通过相邻字符移位变成另一个字符串

http://blog.csdn.net/beiyeqingteng/article/details/7304388

问题：

给你两个字符串，比如 ABCD 和 DBCA，通过交换相邻字符的位置，把第一个字符串，变成第二个字符串，打印所有的步骤。比如从ABCD到DBCA，步骤是：

ABCD ---> BACD ---> BCAD ---> BCDA ---> BDCA ---> DBCA

分析：

如果两个字符串里所含的字符是一样的，只是位置不一样，我们是可以证明通过交换相邻的字符，能够从一个字符串变成另一个字符串，但是，这里忽略证明，大家只需要知道这个事实就可以了。

要从一个字符到另一个字符，最方便也是最有效的方法是从目标字符串的最后一位开始（比如刚刚的例子里，目标字符串最后一个字符是A），找出改字符在原始字符串的位置，然后通过两两移位使得该字符从原始字符串移动到目标字符串的位置。我们不断遍历目标字符串，直到顺数第二个字符（为何不是顺数第一个字符？自己思考。）。

这样做有一个好处，可以非常方便处理字符串里有重复字符的情况。如果不相信，可以用其它方法试试，看是否比刚刚给出的方法更好。

public class Transpose {

public void transpose(String strFrom, String strTo) {

char[] charFrom = strFrom.toCharArray();

char[] charTo = strTo.toCharArray();

int length = strFrom.length();

int index = length - 1;

int position = - 1;

//print the original string

swap(0,0, charFrom);

//start from the end

while (index > 0) {

position = findPosition(charTo[index], charFrom, index);

//swap the characters

for (int j = position; j < index ; j++) {

swap(j, j + 1, charFrom);

}

index--;

}

}

// swap and print

public void swap (int p1, int p2, char[] strTrans) {

char temp = strTrans[p1];

strTrans[p1] = strTrans[p2];

strTrans[p2] = temp;

for (char ch : strTrans) {

System.out.print(ch);

}

System.out.println();

}

// find the position of charater "ch" in the ori

public int findPosition(char ch, char[] charFrom, int index) {

int position = -1;

for (int i = index; i >= 0; i--) {

if (charFrom[i] == ch) {

position = i;

break;

}

}

return position;

}

public static void main(String[] args) {

new Transpose().transpose("ABCD", "DBCA");

}

}

## 求两个数组的交集

http://blog.csdn.net/beiyeqingteng/article/details/7391422

问题： 给你两个排序的数组，求两个数组的交集。

比如： A = 1 3 4 5 7， B = 2 3 5 8 9， 那么交集就是 3 5.

思路：

1. 每一次从B数组中取一值，然后在A数组里逐个比较，如果有相等的，则保存。该算法复杂度为 O(MN). M, N 分别为数组 A B 的长度。

2. 因为A B 都排过序，所以，每一次从B数组取值后，可以利用二分查找看是否在数组A里有B所对应的值，这样复杂度变成了O(N lg M)。 这里，如果N 比 M 大，可以从A中取值，然后在B中判断是否有A的值，这样，复杂度为 O(M lg N)。

3. 利用hashtable. 先把A中的值存在hashtable里，然后对于每一个B中的值，判断是否在A中存在，因为hashtable查找的平均复杂度为 O(1), 所以，整个算法复杂度为O(N), 但是，这里的空间复杂度为 O(M) 。但是，这种方法适合数组比较小的情况。因为如果A数组比较大，hashtable会出现collision的情况，这样，查找的平均复杂度就不再是 O(1)。

本文方法：

因为数组A B均排过序，所以，我们可以用两个“指针”分别指向两个数组的头部，如果其中一个比另一个小，移动小的那个数组的指针；如果相等，那么那个值是在交集里，保存该值，这时，同时移动两个数组的指针。一直这样操作下去，直到有一个指针已经超过数组范围。

public LinkedList<Integer> intersection(int[] A, int[] B) {

if (A == null || B == null || A.length == 0 || B.length == 0) return null;

LinkedList<Integer> list = new LinkedList<Integer>();

int pointerA = 0;

int pointerB = 0;

while (pointerA < A.length && pointerB < B.length) {

if (A[pointerA] < B[pointerB]) pointerA++;

else if (A[pointerA] > B[pointerB]) pointerB++;

else {

list.add(A[pointerA]);

pointerA++;

pointerB++;

}

}

return list;

}

通过上面的算法可以得知，该算法复杂度为O(N + M).

扩展：

给两个排序的数组，求两个数组的并集。

## 间隔问题 [No. 56]

http://blog.csdn.net/beiyeqingteng/article/details/7401328

问题：

给定多个不重合的间隔（interval），比如 [3, 8] [ 12, 17] [22, 29], 然后插入一个新的间隔， 比如 [ 5, 15], 求最后不重合的间隔。比如，起初的不重合间隔变成 [3, 17] [22, 29].

分析：

首先，我们要知道怎么确定两个间隔是重合的。比如[1, 5] [3, 7] 是重合的，[1, 5] [6, 7] 是不重合的。那么，观察后发现，如果两个间隔的起始值的最大值小于两个间隔的结束值的最小值，那么，这两个间隔一定重合。代码如下：

/\*

\* s1: start point of interval 1

\* e1: end point of interval 1

\* s2: start point of interval 2

\* e2: end point of interval 2

\*/

public boolean overlap(int s1, int e1, int s2, int e2) {

return Math.max(s1, s2) < Math.min(e1, e2);

}

假定给定的间隔按照起始值已经排序了，就是说第一个间隔的起始值是所有间隔起始值中最小的，最后一个间隔的起始值是所有间隔起始值中最大的。当插入一个新的间隔时，我们只需要判断它是否和每一个间隔相交，如果不相交，保持不变，如果相交，则把新的间隔求出来，再继续和后面的间隔进行判断。代码如下：

public class Test {

//check whether the two intervals intersect

public boolean overlap(Interval i1, Interval i2) {

return Math.max(i1.start, i2.start) < Math.min(i1.end, i2.end);

}

//add the new interval to the interval list and return a new interval list without overlapping

public ArrayList<Interval> add(ArrayList<Interval> list, Interval interval) {

ArrayList<Interval> newList = new ArrayList<Interval>();

int i = 0;

while (i < list.size()) {

//no intersection

if (overlap(list.get(i), interval) == false) {

newList.add(list.get(i));

i++;

} else {

//get the new interval

while (i < list.size() && overlap(list.get(i), interval) == true) {

int start = Math.min(list.get(i).start, interval.start);

int end = Math.max(list.get(i).end, interval.end);

interval = new Interval(start, end);

i++;

}

newList.add(interval);

}

}

return newList;

}

public static void main(String[] args) {

ArrayList<Interval> list = new ArrayList<Interval>();

list.add(new Interval(3, 8));

list.add(new Interval(22, 29));

list.add(new Interval(12, 17));

list.add(new Interval(40, 47));

Collections.sort(list);

Interval interval = new Interval(40, 48);

Test test = new Test();

ArrayList<Interval> newList = test.add(list, interval);

for(Interval inter : newList) {

System.out.println("[" + inter.start + ", " + inter.end + "]");

}

}

}

class Interval implements Comparable<Interval>{

int start;

int end;

Interval(int start, int end) {

this.start = start;

this.end = end;

}

public int compareTo(Interval o) {

if (start > ((Interval)o).start ||

(start == ((Interval)o).start && end > ((Interval)o).end)) {

return 1;

} else if (start == ((Interval)o).start && end == ((Interval)o).end) {

return 0;

}

return -1;

}

}

## 间隔问题2 [No. 57]

http://blog.csdn.net/beiyeqingteng/article/details/7402369

问题：

给一组间隔，把这些间隔合并，使得间隔之间不会有重合。比如：[1, 5], [3, 8], [10, 20], [7, 9] 合并后变成 [1, 9], [10, 20]。

分析：

首先，因为我们需要判断是否两个间隔是否重合，这样才能合并，检查两个间隔是否重合的代码是：

/\*

\* s1: start point of interval 1

\* e1: end point of interval 1

\* s2: start point of interval 2

\* e2: end point of interval 2

\*/

public boolean overlap(int s1, int e1, int s2, int e2) {

return Math.max(s1, s2) < Math.min(e1, e2);

}

然后，我们把间隔按照开始值排序，这样，从头开始检查，如果当前间隔和前一个间隔重合，就合并。否则，直接把它加入到list里面。

代码如下：

public class Test {

//check whether the two intervals intersect

public boolean overlap(Interval i1, Interval i2) {

return Math.max(i1.start, i2.start) < Math.min(i1.end, i2.end);

}

//merge the intervals and return a new interval list without overlapping

public void merge(ArrayList<Interval> list) {

int currentPosition = 0;

for (int i = 1; i < list.size(); i++) {

//no intersection

if (overlap(list.get(i), list.get(currentPosition)) == false) {

currentPosition++;

} else {

//get the new interval

int start = Math.min(list.get(i).start, list.get(currentPosition).start);

int end = Math.max(list.get(i).end, list.get(currentPosition).end);

Interval interval = new Interval(start, end);

list.remove(currentPosition);

list.add(currentPosition, interval);

}

}

int size = list.size();

for (int i = currentPosition + 1; i < size; i++) {

list.remove(currentPosition + 1);

}

}

public static void main(String[] args) {

ArrayList<Interval> list = new ArrayList<Interval>();

list.add(new Interval(3, 8));

list.add(new Interval(2, 29));

list.add(new Interval(5, 17));

list.add(new Interval(22, 47));

list.add(new Interval(32, 48));

Collections.sort(list);

new Test().merge(list);

for(Interval inter : list) {

System.out.println("[" + inter.start + ", " + inter.end + "]");

}

}

}

class Interval implements Comparable<Interval>{

int start;

int end;

Interval(int start, int end) {

this.start = start;

this.end = end;

}

public int compareTo(Interval o) {

if (start > ((Interval)o).start ||

(start == ((Interval)o).start && end > ((Interval)o).end)) {

return 1;

} else if (start == ((Interval)o).start && end == ((Interval)o).end) {

return 0;

}

return -1;

}

}

## 间隔问题3 [No. 58]

http://blog.csdn.net/beiyeqingteng/article/details/7405921

问题：

给一组间隔，找出一个点，使得穿过改点的间隔数量最大, 时间复杂度不能超过 O(N lg N)。比如有一组间隔为[1, 5] [ 3, 7] [2, 4] [8, 9], 那么对于点 3.5， 有3个间隔都包含这个点，但是对于点 8.5，只有一个间隔包含该点。所以，最后答案是 3.5. (当然，3.6， 或者 3.7 都可以)。

分析：

对于一组间隔，我们把它所有间隔的起始值和结束值放在一起排序，每遇到一个间隔的起始值，我们把个数加1，遇到一个结束值，我们把个数减1，这样，我们可以得到最大个数。对于 “点” 的值，它一定是在一个结束值和一个起始值之间。

代码如下：

public class Test {

/\*

\* start: stores the start value of the intervals;

\* end: stores the end value of the intervals

\*/

public float findPoint(float[] start, float[] end) {

int currentInterval = 0;

int maximumInterval = 0;

float pointBeginValue = 0.0f;

float pointValue = 0.0f;

//store the interval start values into the list

ArrayList<Point> list = new ArrayList<Point>();

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

list.add(new Point(start[i], true));

}

//store the interval end values into the list

for (int j = 0; j < end.length; j++) {

list.add(new Point(end[j], false));

}

//sort

Collections.sort(list);

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

if (list.get(i).bStart) {

currentInterval++;

if (currentInterval >= maximumInterval) {

maximumInterval = currentInterval;

pointBeginValue = list.get(i).value;

}

} else {

if (currentInterval == maximumInterval) {

pointValue = (pointBeginValue + list.get(i).value) / 2.0f;

}

currentInterval--;

}

}

return pointValue;

}

public static void main(String[] args) {

//start value of the interval

float[] start = {1.2f, 3, 5, 1, 2.4f};

//end value of the interval

float[] end = {2.3f, 4.5f, 7.9f, 2, 4};

System.out.println(new Test().findPoint(start, end));

}

}

class Point implements Comparable<Point>{

float value = 0.0f;

boolean bStart = false;

Point(float value, boolean bStart) {

this.value = value;

this.bStart = bStart;

}

public int compareTo(Point o) {

if (value > ((Point)o).value ) {

return 1;

} else if (value == ((Point)o).value) {

return 0;

}

return -1;

}

}

## 判断一个字符串里是否包含另一个字符串所有的字符

http://blog.csdn.net/beiyeqingteng/article/details/7447908

问题：

有两个字符串，每个字符串的字符从A-Z中选取，比如： B = “ABBC”， A = “ACBBD”。那么A包含所有B中出现的字符。如果A = “ACBD” 或者 A = “ABBD”，则我们认为A不包含B中所有的字符。

分析：

对于“是否包含”的问题，基本上处理的方法都会与hashtable有关。这题也不例外。思路是把所有A中出现的字符放在一个长度为26的字符数组array[26]里，比如，"ABBC" 就是array[26] = {1, 2, 1, 0, ... , 0}， 然后对于每个在B字符串中出现的字符，在array[26]里做相应的减法。只要遇到array[i] < 0, 返回 false。

代码如下：

public boolean contains(String A, String B) {

int[] array = new int[26];

//store the characters in A to array

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

array[A.charAt(i) - 'A'] += 1;

}

//check whether all the characters in B exist in A

for (int j = 0; j < B.length(); j++) {

array[B.charAt(j) - 'A'] -= 1;

if (array[B.charAt(j) - 'A'] < 0) return false;

}

return true;

}

## 在一个字符串中找到第一个只出现一次的字符

http://blog.csdn.net/beiyeqingteng/article/details/7447920

问题：

给定一个字符串，比如 A = “ABCDACD”， 找出第一个只出现一次的字符，在A中，第一个只出现一次的字符是‘B’。

分析：

为了判定某字符是否出现一次，我们可以从从头开始往下遍历，如果没有重复，则选取，否则抛弃。这样做的话复杂度为 O(n^2)。其实，对于判定是否存在或者存在的次数等问题，基本上都会与hastable有关，我们可以构建一个数组 array[256] (ASCII), 然后对字符串先进行处理，对于每个出现的字符，我们可以在相对应的位置+1。这样，我们再次从头开始对字符串进行遍历，如果发现某一字符个数为1，则返回该字符。

代码如下：

//if there is no such character in A, return 0 (null)

public char firstOnlyCharacter(String A) {

int[] array = new int[256];

//store the characters in A to array

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

array[A.charAt(i)] += 1;

}

//get the first charater with only one appearance in A

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

if (array[A.charAt(i)] == 1) return A.charAt(i);

}

return 0;

}

## 把一个字符串转成double类型的数[# 61]

http://blog.csdn.net/beiyeqingteng/article/details/7459664

问题：

给一个字符串，比如“-12.05”，把它转成相应的double类型的数。

分析：

在进行转换的时候，要注意以下问题：

1. 该字符串是否为空

2. 是否该字符串含有符号；

3. 该字符串内是否有非法字符；

4. 小数点的位置；

5. 该数是否越界；

代码如下：

public static double atod(String str) throws Exception {

boolean negative = false;

//get the value before the "."

double valueBeforeDot = 0.0d;

//get the value after the ".";

double valueAfterDot = 0.0d;

boolean pointAppear = false;

int count = 0;

//null or empty string

if (str == null || str.equals("")) {

throw new Exception("null string or the string has no character!");

}

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

//check whether the first character is "+" or "-"

if (i == 0 && (str.charAt(0) == '-' || str.charAt(0) == '+')) {

if (str.charAt(0) == '-') {

negative = true;

}

} else {

//check whether the character is "." and appears for

//the first time and appears at the correct position.

if (pointAppear == false && i != 0 && str.charAt(i) == '.'

&& (str.charAt(0) != '-' || str.charAt(0) != '+')) {

pointAppear = true;

} else {

if (str.charAt(i) >= '0' && '9' >= str.charAt(i)) {

if (pointAppear == false) {

valueBeforeDot = valueBeforeDot \* 10 + (str.charAt(i) - '0');

if (valueBeforeDot > Double.MAX\_VALUE) {

throw new Exception("out of Double range");

}

} else {

valueAfterDot = valueAfterDot \* 10 + (str.charAt(i) - '0');

count++;

}

} else {

throw new NumberFormatException("not a double");

}

}

}

}

valueBeforeDot = valueBeforeDot + valueAfterDot /Math.pow(10, count);

return negative == true ? valueBeforeDot \* -1 : valueBeforeDot;

}

## 判断一个数字是否是回文

http://blog.csdn.net/beiyeqingteng/article/details/7475894

问题：

给一个数字，判断该数字是否是回文。比如 1221是回文，而123不是。

分析：

判断一个数字是否是回文，我们可以先把它转成字符串，然后根据回文“对称”的特性进行判断：查看第 x 个字符是否与第 n - x + 1 字符是否相等 （x 从1开始，n/2 结束, n 是指数字长度）。但是，这道题因为是数字，有一种更好的解法。即根据原始数字，通过数学方法构建出它的回文数字，如果原始数字大小等于回文数字大小，那么该数字是回文，否则不是。

代码如下：

bool isPalindrome(int originalNumber) {

int palindrome = 0;

int origin = originalNumber;

// get the palindrome

while(originalNumber != 0) {

palindrome = palindrome \* 10 + originalNumber % 10;

originalNumber /= 10;

}

return palindrome == origin ;

}

## 找出二叉查找树中第n大的值

http://blog.csdn.net/beiyeqingteng/article/details/7485761

我们可以通过类似中序遍历的方法把BST从大到小排序，然后，就可以得到第 k 大的值了。代码如下：

public class NthNode {

// k refers to the counter. it is a global variable.

static int k = 0;

//get the nth largest value in BST

public void getNthnode(Node root, int n) {

if (root == null) return;

getNthnode(root.rightChild, n);

k++;

if (k == n) {

System.out.print(root.toString());

}

getNthnode(root.leftChild, n);

}

public static void main(String[] args) {

Node a = new Node(8);

Node b = new Node(3);

Node c = new Node(10);

Node d = new Node(1);

Node e = new Node(6);

Node f = new Node(14);

Node g = new Node(4);

Node h = new Node(7);

Node i = new Node(13);

a.leftChild = b;

a.rightChild = c;

b.leftChild = d;

b.rightChild = e;

c.rightChild = f;

e.rightChild = g;

e.rightChild = h;

f.leftChild = i;

//the third largest value in BST

new NthNode().getNthnode(a, 3);

}

}

class Node {

Node leftChild = null;

Node rightChild = null;

int value;

Node(int value) {

this.value = value;

}

public String toString() {

return value + "";

}

}

## 给定一个二叉树，从左到右，找出第 k 个叶子节点

http://blog.csdn.net/beiyeqingteng/article/details/7485763

分析：

因为顺序是从左往右数，所以，对于一个节点下的两个叶子节点来讲（比如 6 下面有两个叶子节点 5 和 11），我们要确保先遍历最左边一个，然后再遍历右边一个。这样，其实，在中序遍历，前序遍历和后序遍历中，都能保证左边叶子节点比右边叶子节点先被遍历。我们只需要对每一个遍历的节点进行检查，看是否是叶子节点，是，则把个数+1. 代码如下：

public class NthLeaf {

static int k = 0;

// get the nth leaf by using preorder traversal

public void getNthleve(Node root, int n) {

if (root == null) return;

if (root.rightChild == null && root.leftChild == null) {

k++;

if (k == n) {

System.out.print(root.toString());

}

}

getNthleve(root.leftChild, n);

getNthleve(root.rightChild, n);

}

## 在两个已经排好序的数组里找出第K小的数

http://blog.csdn.net/beiyeqingteng/article/details/7533304

public static int kthTwoSortedArray(int[] A, int[] B, int k) throws Exception {

if (k > A.length + B.length || k < 1) throw new Exception("out of range!");

// pointer of array A

int pa = 0;

// pointer of array B

int pb = 0;

//store the kth value

int kthValue = 0;

while (pa + pb != k) {

int Ai = (pa == A.length) ? Integer.MAX\_VALUE : A[pa];

int Bj = (pb == B.length) ? Integer.MAX\_VALUE : B[pb];

if (Ai < Bj) {

pa++;

kthValue = Ai;

} else {

pb++;

kthValue = Bj;

}

}

return kthValue;

}

上面这个程序的复杂度是O(k)，分析很简单，就不再讲了。

现在再来看如何得到 O(lg K) 的算法。……

## 概率面试题骰子和收益

http://blog.csdn.net/beiyeqingteng/article/details/7539366

## 判断两个链表是否相交

http://blog.csdn.net/beiyeqingteng/article/details/7603627

问题：

给出两个单向链表的头指针，比如h1、h2，判断链表是否相交，如果不相交返回NULL；如果相交，返回指向第一个相交节点的指针。时间复杂度控制在O(n)。

分析：

如果两单向链表相交的话，一定是Y型相交，不可能出现X型，弄清楚这点后接下来的工作就是：

（1）先找到h1，h2的最后一个节点L1和L2，同时记录节点数量a，b；（这里假设 a > b）

（2）判断最后一个节点是否相同；

如果不相同则没相交；

如果相同，则从第一个节点 (短链表) 和 第 |a-b|+1个节点 （长链表）开始比较 （这样，两个链表的指针到相交点的距离是一样的），看是否相等，不相等就寻找下一个节点直到找到交叉点。

代码如下：

Node intersect(Node h1, Node h2) {

// l1 and l2 refer to the last nodes in the two lists.

// a and b refer to the length of the lists.

Node l1 = h1;

int a = 1;

Node l2 = h2;

int b = 1;

while (l1.next != null) {

l1 = l1.next;

a++;

}

while (l2.next != null) {

l2 = l2.next;

b++;

}

// no intersection

if (l1 != l2) {

return null;

}

//move the pointer of the longer list.

for (int i = 0; i < Math.abs(a - b); i++) {

if (a > b) {

h1 = h1.next;

} else {

h2 = h2.next;

}

}

while (h1 != h2) {

h1 = h1.next;

h2 = h2.next;

}

return h1;

}

## 求一个数字数组里的最大连续数字的个数

http://blog.csdn.net/beiyeqingteng/article/details/7682644

问题：

求一个数字数组里的最大连续数字的个数。 比如 3, 4, 4, 4, 2, 2, 3, 4 => return 3。此题为google的面试题。

分析：

设置两个变量：全局最大连续数字个数：maxSucc； 以及局部连续数字个数： temp。从第二个数字开始，如果当前数字比前一个数字大1，则 temp++，遇到不满足条件的，则比较maxSucc 和 temp， 如果temp 比maxSucc 大，则更新maxSucc。

代码：

public static int maxSuccessive(int[] array) {

if (array == null || array.length == 0) return -1;

int maxSucc = 1;

int temp = 1;

for(int i = 1; i < array.length; i++) {

if (array[i] -1 == array[i - 1]) {

temp++;

} else {

if (temp > maxSucc) {

maxSucc = temp;

}

temp = 1;

}

}

//very important, do not return maxSucc

return maxSucc > temp ? maxSucc : temp;

}

## 第二个字符串是第一个的子串，打印第二个字符串缺了的字符位置

http://blog.csdn.net/beiyeqingteng/article/details/7682674

问题：

google 面试题：

两个字符串，第二个字符串是第一个的子串，打印第二个字符串缺了的字符位置。比如：

“abc”, “ab” => print “2”

“abc”, “b” => print “0 2”

“abc”, “ac” => print “1”

“aab”, “ab” => print “0” OR print “1”。

分析：

我们只需要对第一个字符串遍历，如果其值在第二个字符串中不存在，则打印其位置即可。

代码：

public static void missingPosition(String str1, String str2) {

int index = 0;

for (int i = 0; i < str1.length(); i++) {

if (str1.charAt(i) != str2.charAt(index)) {

System.out.print(i + " ");

} else {

index++;

}

}

}

## 括号匹配

http://blog.csdn.net/beiyeqingteng/article/details/7694845

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

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

public static boolean isValid(String s) {

if (s == null || s.length() <= 1) return false;

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

bracket.push(s.charAt(0));

for (int i = 1; i < s.length(); i++) {

if (s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{') {

bracket.push(s.charAt(i));

} else if (s.charAt(i) == ')' && bracket.size() !=0 && bracket.peek() == '(') {

bracket.pop();

} else if (s.charAt(i) == ']' && bracket.size() !=0 && bracket.peek() == '[') {

bracket.pop();

} else if (s.charAt(i) == '}' && bracket.size() !=0 && bracket.peek() == '{') {

bracket.pop();

} else {

return false;

}

}

if (bracket.size() != 0) return false;

return true;

}

## 打印二叉树的边缘节点

http://blog.csdn.net/beiyeqingteng/article/details/7705931

逆时针打印边缘节点后，输出：1 , 3， 5， 9， 8, 6 .

分析：

如果想写一个方法实现这个要求是很难的，但是，我们可以考虑分步实现。

第一步：打印左边的边缘节点；

第二步：打印底部所有节点；

第三步：打印右边所有边缘节点。

public static void printLeft(Node root) {

if (root.leftChild != null || root.rightChild != null) {

print(root);

printLeft(root.leftChild);

}

}

public static void printButtom(Node root) {

if (root != null) {

if (root.leftChild == null && root.rightChild == null) {

print(root);

}

printButtom(root.leftChild);

printButtom(root.rightChild);

}

}

public static void printRight(Node root, Node node) {

if (root.leftChild != null || root.rightChild != null) {

printRight(root.rightChild, node);

// do not print the root again

if (root != node) {

print(root);

}

}

}

## 名人问题 (Celebrity problem)

http://blog.csdn.net/beiyeqingteng/article/details/7707485

问题：

在一个房间里有 N 个人，其中一个是名人，所谓名人就是大家都认识他，但是他不认识任何人。其它人可能认识房间里面另外的一部分人。你可以问任何人问题，但是问题只能是：你认识 X 吗，对方回答 Yes or No. 请问最少要问多少个问题才能把名人找出来？

分析：

我们把人编号，比如从1 到 N。

我们考虑最坏情况：你问每一个人是否认识 X ，如果大家都认识，那么 X 一定是名人，如果其中一个人说不认识，那么那个人一定不是名人。因为 X 的范围是 1 到 N， 所以，在最坏情况下，我们 要问 （N - 1）\* N 个问题。

但是，有一种更好的方法。我们从1开始，依次问他是否认识他的下一个，比如 2， 如果 1 说认识，那么 1 一定不是名人，2 有可能是名人； 如果1 说不认识，2 一定不是名人，1 却有可能是名人。这是这个方法巧的地方。所以，不管1 回答是还是不是，我们都可以确定其中一个不是名人，然后，我们继续问 可能是名人那一位 是否认识 3， 然后依次下去，直到第 N 个人。这样我们就可以只要问 （N - 1） 个问题就可以把名人找出来。

## 生男生女的面试题

http://blog.csdn.net/beiyeqingteng/article/details/7998203

## check whether two binary trees are identical

http://blog.csdn.net/beiyeqingteng/article/details/8034566

boolean sameTree(Node a, Node b) {

// 1. both empty -> true

if (a == null && b == null) return(true);

// 2. both non-empty -> compare them

else if (a!=null && b!=null) {

return (a.data == b.data &&

sameTree(a.leftChild, b.leftChild) &&

sameTree(a.rightChild, b.rightChild)

);

}

// 3. one empty, one not -> false

else return false;

}

## 不用循环反转字符串

http://blog.csdn.net/beiyeqingteng/article/details/8063296

问题：

这是我今天面试时遇到的一道题。要求不使用循环反转字符串，当时被卡住了，一直没有想到要用递归，后面面试官提示用递归，才想到，哎。

分析：

递归的思路是：构建一个方法，该方法把返回从0到 i 的反转字符串，如果i = 0, 返回当前字符。

public class Reverse {

public static void main(String[] args) {

String str = "abcd";

String str2 = reverse(str, str.length() - 1);

System.out.println(str2);

}

public static String reverse(String str, int index) {

if (index == 0)

return str.substring(0,1);

return str.substring(index, index + 1) + "" + reverse(str, index - 1);

}

}

## 版本号比较

http://blog.csdn.net/beiyeqingteng/article/details/8101966

问题：

给定两个版本号，比较哪一个版本号更新。比如 1.2.9 比 1.2.4 新。 1.0.1 比 1.0 新。

思路：

当比较版本号时，我们需要从头到尾比较每一组数据。所以，我们需要首先得到版本号的一个数字，第二个数字，第三个数字，然后和另一个版本号码所对应的数字进行比较。

代码：

public class VersionComparison {

public static void main(String[] args) {

String version1 = "2.20.3";

String version2 = "2.6.9";

int result = compare(version1, version2);

if (result == 1) {

System.out.println(version1 + " is newer.");

} else if (result == 0) {

System.out.println("The two versions are the same.");

} else {

System.out.println(version2 + " is newer.");

}

}

/\*\*

\*

\* @param version1

\* @param version2

\* @return if version1 > version2, return 1, if equal, return 0, else return -1

\*/

public static int compare(String version1, String version2) {

if (version1 == null || version1.length() == 0 || version2 == null || version2.length() == 0 )

throw new IllegalArgumentException("Invalid parameter!");

int index1 = 0;

int index2 = 0;

while(index1 < version1.length() && index2 < version2.length()) {

int[] number1 = getValue(version1, index1);

int[] number2 = getValue(version2, index2);

if (number1[0] < number2[0]) return -1;

else if (number1[0] > number2[0]) return 1;

else {

index1 = number1[1] + 1;

index2 = number2[1] + 1;

}

}

if(index1 == version1.length() && index2 == version2.length()) return 0;

if(index1 < version1.length())

return 1;

else

return -1;

}

/\*\*

\*

\* @param version

\* @param index the starting point

\* @return the number between two dots, and the index of the dot

\*/

public static int[] getValue(String version, int index) {

int[] value\_index = new int[2];

StringBuilder sb = new StringBuilder();

while(index < version.length() && version.charAt(index) != '.') {

sb.append(version.charAt(index));

index++;

}

value\_index[0] = Integer.parseInt(sb.toString());

value\_index[1] = index;

return value\_index;

}

}

## 约瑟夫

http://blog.csdn.net/beiyeqingteng/article/details/8197760

问题：

n 个人围成一圈（编号1-n），数到3的出列，然后又从头开始数，一直循环到最后一个人，请问最后剩下的人的编号？

public int numberToExit(int total, int interval) {

boolean[] arr = new boolean[total];

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

arr[i] = true;

}

int left = arr.length;

int count = 0;

int index = 0;

while (left > 1) {

if(arr[index] == true) {

count++;

if(count == 3) {

count = 0;

arr[index] = false;

left--;

}

}

// we need to change the index if out of range.

index++;

if(index == arr.length) {

index = 0;

}

}

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

if(arr[i] == true) {

return i+1;

}

}

return -1;

}

## 不通过比较，找出两个数的最大值

http://blog.csdn.net/beiyeqingteng/article/details/8197801

问题：

比如：给定两个值 5和10，不通过比较，直接找出最大值。

分析：

一旦涉及到不用比较找最大值，想都不用想，一般只能通过位运算来实现。

max = a - ((a-b)&((a-b)>>31))

或者

max = ((a+b)+|a-b|)/2

如果找最小值，我们只需把两个值相加，减去max即可。

## 从数列1, 2, 3, ... , n 中 随意取几个数,使其和等于sum

http://blog.csdn.net/beiyeqingteng/article/details/7107636

问题：

输入两个整数 n 和 sum，从数列1, 2, 3, ... , n 中 随意取几个数,使其和等于 sum,要求将其中所有的可能组合列出来.

比如n = 5, sum = 8, 那么所有的组合为：

3 5 或者 1 5 2 或者 1 4 3

思路：

对于一个给定的总和sum, 如果我们从数列的尾部开始，即从n开始，sum 要么等于 n 加上 剩余的1到 n - 1之间的某一个组合，或者 等于 1 到 n-1 之间的某一个组合。

递归的方程为：

findSum(sum, n) = n + findSum(sum - n, n -1) 或者 findSum (sum, n) = findSum (sum, n -1)

public class FindSum {

private static Stack<Integer> stack = new Stack<Integer>();

public static void findSum(int sum, int n) {

//找不到满足条件的情况

if ( sum <= 0 || n <= 0) return;

//sum 大于 n， 需要再次递归

if (sum > n) {

stack.push(n); //取 n

findSum(sum - n, n - 1);// n加入，取 n = n-1, sum = sum-n

stack.pop(); // 不取 n

findSum(sum, n - 1); // n没有加入，取n = n-1,sum = sum

} else {

System.out.print(sum); // sum <= n ,直接输出n就可以满足了

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

System.out.print(" "+ stack.get(i));

System.out.println();

//上面部分得到的是一种情况，但是还存在另一种情况，我们需要找出从 1 到 sum - 1 里面是否存在值，它们的和等于sum

if (sum != 1) {

findSum(sum, sum - 1);

}

}

}

public static void main(String[] args) {

int sum = 8;

int n = 5;

findSum(sum,n);

}

}

扩展1：

如果在上面这道题里再加上一个条件：所有组合里数的个数是一定的。下面代码中用number 来表示所有组合数的个数。

public class FindSum {

private static Stack<Integer> stack = new Stack<Integer>();

public static void findSum(int sum, int n, int number) {

//找不到满足条件的情况

if ( sum <= 0 || n <= 0 || number <= 0 || n < number) return;

//sum 大于 n， 需要再次递归

if (sum > n ) {

stack.push(n); //取 n

findSum(sum - n, n - 1, number - 1);// n加入，取 n = n-1, sum = sum-n

stack.pop(); // 不取 n

findSum(sum, n - 1, number); // n没有加入，取n = n-1,sum = sum

} else if (number == 1) {

System.out.print(sum); // sum <= n ,直接输出n就可以满足了

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

System.out.print(" "+ stack.get(i));

System.out.println();

} else {

findSum(sum, sum -1, number);

}

}

public static void main(String[] args) {

int sum = 10;

int n = 12;

int number = 4;

findSum(sum,n, number);

}

}

扩展2：任意给定一个不含有重复值的数组，找出和为 sum 的所有的组合，但是，组合中个数为 number.

比如：数组为 {5， 2， 1， 7， 3}， sum = 8, number = 3， 那么，就只能输出 5， 2，1。

原理是一样的，代码如下：

public class FindSum {

public static int[] array = {5, 2, 1, 7, 3, 4, 6, 9};

private static Stack<Integer> stack = new Stack<Integer>();

public static void findSum(int sum, int n, int number) {

if (number == 0 && sum == 0) {

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

System.out.print(" "+ stack.get(i));

}

System.out.println();

return;

}

if (sum <= 0 || n < 0 || number <= 0 || n < number - 1) return;

stack.push(array[n]); //取 n

findSum(sum - array[n], n - 1, number - 1);// n加入，取 n = n-1, sum = sum-n

stack.pop(); // 不取 n

findSum(sum, n - 1, number); // n没有加入，取n = n-1,sum = sum

}

public static void main(String[] args) {

int sum = 10;

int number = 4;

findSum(sum, array.length - 1, number);

}

}

## Split string to palindrome

http://blog.csdn.net/beiyeqingteng/article/details/8606736

Question:

Give a string, use minimum number of splits to divide the string into multiple parts in which each part a palindrome. For example, if the string is "abbad", we can split it into "abba" and "d", then the total splits is 1.

public class NumberOfSplits {

public static void main(String[] args) {

String str = "abbadd";

System.out.println(new NumberOfSplits().numberOfSplits(str));

}

public int numberOfSplits(String str) {

assert (str != null);

int[] cuts = new int[str.length()];

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

cuts[i] = i;

}

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

if (isPalindrome(str, 0, i)) {

cuts[i] = 0;

continue;

}

for (int j = 1; j <= i; j++) {

if (isPalindrome(str, j, i) && cuts[i] > cuts[j - 1] + 1) {

cuts[i] = cuts[j - 1] + 1;

}

}

}

return cuts[str.length() - 1];

}

public boolean isPalindrome(String str, int i, int j) {

while (i <= j) {

if (str.charAt(i) != str.charAt(j)) {

return false;

}

i++;

j--;

}

return true;

}

}