KMP算法

1.判断字符串1是否包含字符串2，不含有返回-1，含有返回字符串2在字符串1中开始的位置

package basic\_class\_02;

publicclass Code\_01\_KMP {

publicstaticint getIndexOf(String s,String m) {

if(s==null||m==null||m.length()<1||s.length()<m.length()) {

return -1;

}

char[]ss=s.toCharArray();

char[]ms=m.toCharArray();

intsi=0;

intmi=0;

int[] next=*getNextArray*(ms);

while(si<ss.length&&mi<ms.length) {

if(ss[si]==ms[mi]) {

si++;

mi++;

}elseif(next[mi]==-1) {

si++;

}else {

mi=next[mi];

}

}

returnmi==ms.length?si-mi:-1;

}

publicstaticint[] getNextArray(char[]ms){

if(ms.length==1) {

returnnewint[] {-1};

}

int[]next=newint[ms.length];

next[0]=-1;

next[1]=0;

intpos=2;

intcn=0;

while(pos<next.length) {

if(ms[pos-1]==ms[cn]) {

next[pos++]=++cn;

}elseif(cn>0) {

cn=next[cn];

}else {

next[pos++]=0;

}

}

returnnext;

}

publicstaticvoid main(String[] args) {

String str = "abcabcababaccc";

String match = "ababa";

System.*out*.println(*getIndexOf*(str, match));

}

}

2.将原序列扩展为含有2个原序列的最短序列

package basic\_class\_02;

publicclass Code\_02\_KMP\_ShortestHaveTwice {

publicstatic String answer(String str) {

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

return"";

}

char[] chas=str.toCharArray();

if(chas.length==1) {

returnstr+str;

}

if(chas.length==2) {

returnchas[0]==chas[1]?(str+String.*valueOf*(chas[0])):(str+str);

}

intendNext=*endNextLength*(chas);

returnstr+str.substring(endNext);

}

publicstaticint endNextLength(char[] chas) {

int[] next=newint[chas.length+1];//注意

next[0]=-1;

next[1]=0;

intpos=2;

intcn=0;

while(pos<next.length) {

if(chas[pos-1]==chas[cn]) {

next[pos++]=++cn;

}elseif (cn>0) {

cn=next[cn];

}else {

next[pos++]=0;

}

}

returnnext[next.length-1];

}

publicstaticvoid main(String[] args) {

String test1 = "a";

System.*out*.println(*answer*(test1));

String test2 = "aa";

System.*out*.println(*answer*(test2));

String test3 = "ab";

System.*out*.println(*answer*(test3));

String test4 = "abcdabcd";

System.*out*.println(*answer*(test4));

String test5 = "abracadabra";

System.*out*.println(*answer*(test5));

}

}

3.字符串是否由子串重复拼接

package basic\_class\_02;

publicclass Code\_02\_KMP\_Subpinjie {

publicstatic String answer(String str) {

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

return"F";

}

char[] chas=str.toCharArray();

if(chas.length==1) {

return"F";

}

intlensub=*SubLength*(chas);

if(lensub==0) {

return"F";

}

returnstr.substring(0,lensub);

}

publicstaticint SubLength(char[] chas) {

int[] next=newint[chas.length+1];//注意

next[0]=-1;

next[1]=0;

intpos=2;

intcn=0;

while(pos<next.length) {

if(chas[pos-1]==chas[cn]) {

next[pos++]=++cn;

}elseif (cn>0) {

cn=next[cn];

}else {

next[pos++]=0;

}

}

intlensub=chas.length-next[next.length-1];//举例可推出

return (lensub!=chas.length&&chas.length%lensub==0)?lensub:0;

}

publicstaticvoid main(String[] args) {

String test1 = "a";

System.*out*.println(*answer*(test1));

String test2 = "aa";

System.*out*.println(*answer*(test2));

String test3 = "abcabc";

System.*out*.println(*answer*(test3));

String test4 = "abcabcab";

System.*out*.println(*answer*(test4));

String test5 = "abcabcabcabb";

System.*out*.println(*answer*(test5));

}

}

4.生成窗口最大值数组

package advancedclass\_02;

import java.util.LinkedList;

publicclass Code\_01\_windowmax {//生成窗口最大值数组

publicstaticint[] getMaxWindow(int[] arr,intw) {

if(arr==null||w<1||arr.length<w) {

returnnull;

}

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

int[] res=newint[arr.length-w+1];

intindex =0;

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

while(!qmax.isEmpty()&&arr[qmax.peekLast()]<=arr[i]) {

qmax.pollLast();

}

qmax.addLast(i);

if (qmax.peekFirst()==i-w) {//头部过期

qmax.pollFirst();

}

if(i>=w-1) {//窗口形成

res[index++]=arr[qmax.peekFirst()];

}

}

returnres;

}

publicstaticvoid printArray(int[] arr) {

for (inti = 0; i != arr.length; i++) {

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

}

System.*out*.println();

}

publicstaticvoid main(String[] args) {

int[] arr= {4,3,5,4,3,3,6,7};

*printArray*(*getMaxWindow*(arr,3));

}

}

1. 小于或等于某数的子数组数量

package advancedclass\_02;

import java.util.LinkedList;

publicclass Code\_03\_AllLessNumSubArray {//小于或等于某数的子数组数量

publicstaticint getNum(int[]arr,intnum) {

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

return 0;

}

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

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

intL=0;

intR=0;

intres=0;

while(L<arr.length) {

while(R<arr.length) {

while (!qmin.isEmpty()&&arr[qmin.peekLast()]>=arr[R]) {

qmin.pollLast();

}

qmin.addLast(R);

while (!qmax.isEmpty()&&arr[qmax.peekLast()]<=arr[R]) {

qmax.pollLast();

}

qmax.addLast(R);

if (arr[qmax.getFirst()]-arr[qmin.getFirst()]>num) {//不达标

break;

}

R++;

}

if(qmin.peekFirst()==L) {

qmin.pollFirst();

}

if(qmax.peekFirst()==L) {

qmax.pollFirst();

}

res += R-L;

L++;

}

return res;

}

publicstaticvoid main(String[] args) {

int[] arr= {4,3,7};

intnum=3;

System.*out*.println(*getNum*(arr,num));

}

}

单调栈结构

6.构造数组的MaxTree

package advancedclass\_02;

import java.util.Stack;

import java.util.HashMap;

public class Code\_02\_maxtree {

public static class Node{

public int value;

public Node left;

public Node right;

public Node(int data) {

this.value=data;

}

}

public static Node getMaxtree(int[] arr) {

Node[] nArr=new Node[arr.length];

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

nArr[i]=new Node(arr[i]);

}

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

HashMap<Node,Node> lBigMap=new HashMap<Node,Node>();

HashMap<Node,Node> rBigMap=new HashMap<Node,Node>();

for(int i=0;i!=nArr.length; i++) {//找每个数左边第一个比它大的数

Node curNode=nArr[i];

while((!stack.isEmpty())&&stack.peek().value<curNode.value) {

*popStackSetMap*(stack,lBigMap);

}

stack.push(curNode);

}

while(!stack.isEmpty()) {

*popStackSetMap*(stack,lBigMap);

}

for(int i=nArr.length-1;i!=-1; i--) {//找每个数右边第一个比它大的数

Node curNode=nArr[i];

while((!stack.isEmpty())&&stack.peek().value<curNode.value) {

*popStackSetMap*(stack,rBigMap);

}

stack.push(curNode);

}

while(!stack.isEmpty()) {

*popStackSetMap*(stack,rBigMap);

}

Node head=null;

for(int i=0;i!=nArr.length; i++) {

Node curNode=nArr[i];

Node left=lBigMap.get(curNode);

Node right=rBigMap.get(curNode);

if(left==null&&right==null) {

head=curNode;

}else if(left==null){

if(right.left==null){

right.left=curNode;

}else {

right.right=curNode;

}

}else if(right==null) {

if(left.left==null){

left.left=curNode;

}else {

left.right=curNode;

}

}else {

Node parent=left.value<right.value?left:right;

if(parent.left==null) {

parent.left=curNode;

}else {

parent.right=curNode;

}

}

}

return head;

}

public static void popStackSetMap(Stack<Node> stack,HashMap<Node,Node> map) {

Node popNode=stack.pop();

if(stack.isEmpty()) {

map.put(popNode, null);

}else {

map.put(popNode,stack.peek());

}

}

public static void main(String[] args) {

int[] arr={4,3,7};

Node head=*getMaxtree*(arr);

System.*out*.println(head.value);

}

}

7.求最大矩阵的大小

package advancedclass\_02;

import java.util.Stack;

public class Code\_04\_zijuzhen {//求最大子矩阵的大小

public static int maxRecSize(int[][] map ) {

if(map==null||map.length==0||map[0].length==0) {

return 0;

}

int maxArea=0;

int[] height=new int[map[0].length];

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

for(int j=0;j<map[0].length;j++) {

height[j]=map[i][j]==0?0:height[j]+1;

}

maxArea=Math.*max*(*maxRecFromBottom*(height), maxArea);

}

return maxArea;

}

public static int maxRecFromBottom(int[] height) {//在直方图找到最大矩形

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

return 0;

}

int maxArea=0;

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

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

while(!stack.isEmpty()&&height[i]<=height[stack.peek()]) {

int j=stack.pop();

int k=stack.isEmpty()?-1:stack.peek();//左边界

int curArea=(i-k-1)\*height[j];

maxArea=Math.*max*(maxArea, curArea);

}

stack.push(i);

}

while(!stack.isEmpty()) {//遍历后，栈中剩余的元素

int j=stack.pop();

int k=stack.isEmpty()?-1:stack.peek();

int curArea=(height.length-k-1)\*height[j];//右边界为height.length

maxArea=Math.*max*(maxArea, curArea);

}

return maxArea;

}

public static void main(String[] args) {

int[][]arr= {{1,0,1,1},

{1,1,1,1},

{1,1,1,0},};

System.*out*.println(*maxRecSize*(arr));

}

}

8.烽火台

package advancedclass\_02;

import java.util.Scanner;

import java.util.Stack;

public class Code\_05\_MountainsAndFlame {

public static void main(String[] args) {

Scanner in=new Scanner(System.*in*);

while(in.hasNextInt()) {

int size=in.nextInt();

int[] arr=new int[size];

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

arr[i]=in.nextInt();

}

System.*out*.println(*communications*(arr));

}

in.close();

}

public static int nextIndex(int size,int i) {

return i<(size-1)?(i+1):0;

}

public static long getInternalSum(int n) {

return n==1L?0L:(long)n\*(long)(n-1)/2L;

}

public static class Pair{

public int value;

public int times;

public Pair(int value) {

this.value=value;

this.times=1;

}

}

public static long communications(int[] arr) {

if(arr==null||arr.length<2) {

return 0;

}

int size=arr.length;

int maxIndex=0;

for(int i=0;i<size;i++) {//找到最大值

maxIndex=arr[maxIndex]<arr[i]?i:maxIndex;

}

int value=arr[maxIndex];

int index=*nextIndex*(size,maxIndex);

long res=0L;

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

stack.push(new Pair(value));

while(index!=maxIndex) {//从最大值开始遍历

value=arr[index];

while(!stack.isEmpty()&&stack.peek().value<value) {

int times=stack.pop().times;

res+=*getInternalSum*(times)+2\*times;

}

if(!stack.isEmpty()&&stack.peek().value==value) {

stack.pop().times++;

} else {

stack.push(new Pair(value));

}

index=*nextIndex*(size,index);

}

while(!stack.isEmpty()) {//遍历完成后，栈中仍有元素

int times=stack.pop().times;

res+=*getInternalSum*(times);

if(!stack.isEmpty()) {

res+=times;

if(stack.size()>1) {

res+=times;

}else {

res+=stack.peek().times>1?times:0;

}

}

}

return res;

}

}

树型DP

9.最大搜索二叉子树

package advancedclass\_04;

public class Code\_04\_BiggestSubBSTInTree {

public static class Node{

public int value;

public Node left;

public Node right;

public Node(int data) {

this.value=data;

}

}

public static int biggestSubBST(Node head) {

return *process*(head).size;

}

public static class ReturnType{

public int size;

public Node head;

public int min;

public int max;

public ReturnType(int a,Node b,int c,int d) {

this.size=a;

this.head=b;

this.min=c;

this.max=d;

}

}

public static ReturnType process(Node head) {

if(head==null) {

return new ReturnType(0,null,Integer.*MAX\_VALUE*,Integer.*MIN\_VALUE*);

}

Node left=head.left;

Node right=head.right;

ReturnType leftSubTreeInfo=*process*(left);

ReturnType rightSubTreeInfo=*process*(right);

int includeItSelf=0;

if(leftSubTreeInfo.head==left

&&rightSubTreeInfo.head==right

&&head.value>leftSubTreeInfo.max

&&head.value<rightSubTreeInfo.min

) {

includeItSelf=leftSubTreeInfo.size+1+rightSubTreeInfo.size;

}

int p1=leftSubTreeInfo.size;

int p2=rightSubTreeInfo.size;

int maxSize=Math.*max*(Math.*max*(p1, p2), includeItSelf);

Node maxHead=p1>p2?leftSubTreeInfo.head:rightSubTreeInfo.head;

if(maxSize==includeItSelf) {

maxHead=head;

}

return new ReturnType(maxSize,maxHead,

Math.*min*(Math.*min*(leftSubTreeInfo.min,rightSubTreeInfo.min), head.value),

Math.*max*(Math.*max*(leftSubTreeInfo.max,rightSubTreeInfo.max), head.value));

}

public static void main(String[] args) {

Node head = new Node(6);

head.left = new Node(1);

head.left.left = new Node(0);

head.left.right = new Node(3);

head.right = new Node(12);

head.right.left = new Node(10);

head.right.left.left = new Node(4);

head.right.left.left.left = new Node(2);

head.right.left.left.right = new Node(5);

head.right.left.right = new Node(14);

head.right.left.right.left = new Node(11);

head.right.left.right.right = new Node(15);

head.right.right = new Node(13);

head.right.right.left = new Node(20);

head.right.right.right = new Node(16);

System.*out*.println(*biggestSubBST*(head));

}

}

10.二叉树上最远距离

package advancedclass\_05;

public class Code\_03\_MaxDistanceInTree {

public static class Node {

public int value;

public Node left;

public Node right;

public Node(int data) {

this.value = data;

}

}

public static int maxdistance(Node head) {

return *process*(head).maxDistance;

}

public static class ReturnType{

public int maxDistance;

public int h;

public ReturnType(int m,int h) {

this.maxDistance=m;//最大距离

this.h=h;//深度

}

}

public static ReturnType process(Node head) {

if(head==null) {

return new ReturnType(0,0);

}

ReturnType leftReturnType=*process*(head.left);

ReturnType rightReturnType=*process*(head.right);

//左子树上离h.left最远距离+1+右子树上离h.right最远距离

int includeHeadDistance=leftReturnType.h+1+rightReturnType.h;

int p1=leftReturnType.maxDistance;//来自左子树的最大距离

int p2=rightReturnType.maxDistance;//来自右子树的最大距离

int resultDistance=Math.*max*(Math.*max*(p1,p2),includeHeadDistance);

int hitself=Math.*max*(leftReturnType.h,rightReturnType.h)+1;

return new ReturnType(resultDistance,hitself);

}

}

11.最大活跃度

package advancedclass\_05;

import java.util.ArrayList;

public class Code\_04\_MaxHappy {

public static class Node{

public int huo;

public ArrayList<Node> nexts;

public Node(int huo) {

this.huo=huo;

nexts=new ArrayList<>();

}

}

public static int getMaxHuo(Node head) {

ReturnData data=*process*(head);

return Math.*max*(data.bu\_lai\_huo, data.lai\_huo);

}

public static class ReturnData{

public int lai\_huo;

public int bu\_lai\_huo;

public ReturnData(int lai\_huo,int bu\_lai\_huo) {

this.lai\_huo=lai\_huo;

this.bu\_lai\_huo=bu\_lai\_huo;

}

}

public static ReturnData process(Node head) {

int lai\_huo=head.huo;

int bu\_lai\_huo=0;

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

Node next=head.nexts.get(i);

ReturnData nextData=*process*(next);

lai\_huo+=nextData.bu\_lai\_huo;

bu\_lai\_huo+=Math.*max*(nextData.lai\_huo, nextData.bu\_lai\_huo);

}

return new ReturnData(lai\_huo,bu\_lai\_huo);

}

//以上输入不是数组

public static int maxHappy(int[][] matrix) {

int[][] dp = new int[matrix.length][2];

boolean[] visited = new boolean[matrix.length];

int root = 0;

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

if (i == matrix[i][0]) {

root = i;

}

}

*process*(matrix, dp, visited, root);

return Math.*max*(dp[root][0], dp[root][1]);

}

public static void process(int[][] matrix, int[][] dp, boolean[] visited, int root) {

visited[root] = true;

dp[root][1] = matrix[root][1];

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

if (matrix[i][0] == root && !visited[i]) {

*process*(matrix, dp, visited, i);

dp[root][1] += dp[i][0];

dp[root][0] += Math.*max*(dp[i][1], dp[i][0]);

}

}

}

public static void main(String[] args) {

int[][] matrix = { { 1, 8 }, { 1, 9 }, { 1, 10 } };

System.*out*.println(*maxHappy*(matrix));

}

}

12.是否为平衡二叉树

package advancedclass\_05;

public class Code\_02\_IsBalancedTree {

public static class Node{

public int value;

public Node left;

public Node right;

public Node(int data) {

this.value = data;

}

}

public static boolean isB(Node head) {

return *process*(head).isB;

}

public static class ReturnData{

public boolean isB;

public int h;

public ReturnData(boolean isB,int h) {

this.isB=isB;

this.h=h;

}

}

public static ReturnData process(Node head) {

if(head==null) {

return new ReturnData(true,0);

}

ReturnData leftData=*process*(head.left);

if(!leftData.isB) {

return new ReturnData(false,0);

}

ReturnData rightData=*process*(head.right);

if(!rightData.isB) {

return new ReturnData(false,0);

}

if(Math.*abs*(leftData.h-rightData.h)>1) {

return new ReturnData(false,0);

}

return new ReturnData(true,Math.*max*(leftData.h,rightData.h)+1);

}

}

动态规划

（1）用暴力递归尝试

（2）画dp表，可变参数变化范围为横纵坐标

（3）找到目标位置

（4）base case

（5）其他位置（依赖于其他位置，参考暴力递归）

13.换钱的方法数

package advancedclass\_06;

import java.util.HashMap;

public class Code\_01\_CoinsWay {

public static int coins1(int[]arr,int aim) {

if(arr==null||arr.length==0||aim<0) {

return 0;

}

return *process1*(arr,0,aim);

}

//int[] arr:不变的变量，面值数组

//index:可以任意自由使用index及其之后所有的钱

//aim:目标钱数

//返回值：方法数

public static int process1(int[]arr,int index,int aim) {

int res=0;

if(index==arr.length) {

res=aim==0?1:0;

}else {

for(int i=0;arr[index]\*i<=aim;i++) {

res+=*process1*(arr,index+1,aim-arr[index]\*i);

}

}

return res;

}

//key:"index\_aim"

//value:返回值

public static HashMap<String,Integer> *map*=new HashMap<>();

public static int process\_map(int[] arr,int index,int aim) {

int res=0;

if(index==arr.length) {

res=aim==0?1:0;

}else {

for(int zhang=0;arr[index]\*zhang<=aim;zhang++) {

int nextAim=aim-arr[index]\*zhang;

String key=String.*valueOf*(index+1)+"-"+String.*valueOf*(nextAim);

if(*map*.containsKey(key)) {

res+=*map*.get(key);

}else {

res+=*process\_map*(arr,index+1,nextAim);

}

}

}

*map*.put(String.*valueOf*(index)+"-"+String.*valueOf*(aim),res);

return res;

}

//动态规划

public static int coins3(int[] arr,int aim) {

if(arr==null||arr.length==0||aim<0) {

return 0;

}

int[][]dp=new int[arr.length+1][aim+1];

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

if(j==0) {

dp[arr.length][j]=1;

}else {

dp[arr.length][j]=0;

}

}

int num=0;

for(int i=arr.length-1;i>=0;i--) {

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

num=0;

for(int k=0;j-arr[i]\*k>=0;k++) {

num+=dp[i+1][j-arr[i]\*k];

}

dp[i][j]=num;

}

}

return dp[0][aim];

}

//省去了枚举过程

public static int coins4(int[] arr,int aim) {

if(arr==null||arr.length==0||aim<0) {

return 0;

}

int[][]dp=new int[arr.length+1][aim+1];

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

if(j==0) {

dp[arr.length][j]=1;

}else {

dp[arr.length][j]=0;

}

}

for(int i=arr.length-1;i>=0;i--) {

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

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

dp[i][j]+=j-arr[i]>=0?dp[i][j-arr[i]]:0;

}

}

return dp[0][aim];

}

public static void main(String[] args) {

int[] coins = { 5, 10,25,1 };

int aim =20 ;

System.*out*.println(*coins1*(coins,aim));

System.*out*.println(*process\_map*(coins,0,aim));

System.*out*.println(*coins3*(coins,aim));

System.*out*.println(*coins4*(coins,aim));

}

}

14.纸牌博弈

package advancedclass\_06;

public class Code\_02\_CardsInLine {

//暴力递归

public static int win1(int[] arr) {

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

return 0;

}

return Math.*max*(*f*(arr,0,arr.length-1),*s*(arr,0,arr.length-1) );

}

//f(i,j)表示如果arr[i...j]上的纸牌被绝顶聪明的人先拿，最终获得的分数

public static int f(int[]arr,int i,int j) {

if(i==j) {

return arr[i];

}

return Math.*max*(arr[i]+*s*(arr,i+1,j),arr[j]+*s*(arr,i,j-1));

}

//s(i,j)表示如果arr[i...j]上的纸牌被绝顶聪明的人后拿，最终获得的分数

public static int s(int[]arr,int i,int j) {

if(i==j) {

return arr[i];

}

return Math.*max*(*f*(arr,i+1,j),*f*(arr,i,j-1));

}

//动态规划

public static int win2(int[] arr) {

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

return 0;

}

//f[i][j]表示函数f的返回值

//s[i][j]表示函数s的返回值

//矩阵大小为N\*N，i<=j，对角线方向为base case

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

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

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

f[j][j]=arr[j];

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

f[i][j]=Math.*max*(arr[i]+s[i+1][j], arr[j]+s[i][j-1]);

s[i][j]=Math.*max*(f[i+1][j], f[i][j-1]);

}

}

return Math.*max*(f[0][arr.length-1],s[0][arr.length-1]);

}

public static void main(String[] args) {

int[] arr = { 1, 9, 1 };

System.*out*.println(*win1*(arr));

System.*out*.println(*win2*(arr));

}

}

15.机器人的路径数

package advancedclass\_06;

public class RobotPath {

//动态规划

//N 一共有1~N个位置

//curPosition 来到的位置

//restSteps 可以走的步数

//K 最终停在的位置

//返回值：一共有多少种走法

public static int ways(int N,int curPosition,int restSteps,int K) {

if(N<2||curPosition<1||curPosition>N||restSteps<0||K<1||K>N) {

return 0;

}

if(restSteps==0) {

return curPosition==K?1:0;

}

int res=0;

if(curPosition==1) {

res=*ways*(N,curPosition+1,restSteps-1, K);

}else if(curPosition==N) {

res=*ways*(N,curPosition-1,restSteps-1, K);

}else {

res=*ways*(N,curPosition+1,restSteps-1, K)+*ways*(N,curPosition-1,restSteps-1, K);;

}

return res;

}

}

16.最小路径和

package class\_08;

public class Code\_07\_MinPath {

public static int minPath1(int[][] matrix) {

return *process1*(matrix,matrix.length-1,matrix[0].length-1);

}

public static int process1(int[][]matrix,int i,int j) {

int res=matrix[i][j];

if(i==0&&j==0) {

return res;

}

if(i==0&&j!=0) {

return res+*process1*(matrix,i,j-1);

}

if(i!=0&&j==0) {

return res+*process1*(matrix,i-1,j);

}

return res+Math.*min*(*process1*(matrix,i,j-1),*process1*(matrix,i-1,j));

}

public static int minPath2(int[][] m) {

if(m==null||m.length==0||m[0]==null||m[0].length==0) {

return 0;

}

int row=m.length;

int col=m[0].length;

int [][] dp=new int[row][col];

dp[0][0]=m[0][0];

for(int i=1;i<row;i++) {

dp[i][0]=dp[i-1][0]+m[i][0];

}

for(int j=1;j<col;j++) {

dp[0][j]=dp[0][j-1]+m[0][j];

}

for(int i=1;i<row;i++) {

for(int j=1;j<col;j++) {

dp[i][j]=Math.*min*(dp[i-1][j],dp[i][j-1])+m[i][j];

}

}

return dp[row-1][col-1];

}

public static void main(String[] args) {

int[][] m = { { 1, 3, 5, 9 }, { 8, 1, 3, 4 }, { 5, 0, 6, 1 }, { 8, 8, 4, 0 } };

System.*out*.println(*minPath1*(m));

System.*out*.println(*minPath2*(m));

}

}

17.是否能累加到aim

package class\_08;

public class Code\_08\_Money\_Problem {

public static boolean money1(int[] arr,int aim) {

return *isSum*(arr,0,0,aim);

}

public static boolean isSum(int[]arr,int i,int sum,int aim) {

if(i==arr.length) {

return sum==aim;

}

return *isSum*(arr,i+1,sum,aim)||*isSum*(arr,i+1,sum+arr[i],aim);

}

public static boolean money2(int[] arr,int aim) {

boolean[][]dp=new boolean[arr.length+1][aim+1];//dp[i][j]表示累加了数组中的i个数

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

dp[i][aim]=true;

}

//dp[arr.length][aim]=true;

for(int i=arr.length-1;i>=0;i--) {

for(int j=aim-1;j>=0;j--) {

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

if(j+arr[i]<=aim) {

dp[i][j]=dp[i][j]||dp[i+1][j+arr[i]];

}

}

}

return dp[0][0];

}

public static void main(String[]args) {

int []arr= {3,2,5,9};

int aim=20;

System.*out*.println(*money1*(arr, aim));

System.*out*.println(*money2*(arr, aim));

}

}

18.字符串匹配

package advancedclass\_06;

public class Code\_03\_RegularExpressionMatch {

//有效性

public static boolean isValid(char[] s,char[] e) {

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

if(s[i]=='\*'||s[i]=='.') {

return false;

}

}

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

if(e[i]=='\*'&&(i==0||e[i-1]=='\*')) {

return false;

}

}

return true;

}

public static boolean isMatch(String str,String exp) {

if(str==null||exp==null) {

return false;

}

char[] s=str.toCharArray();

char[] e=exp.toCharArray();

return *isValid*(s,e)? *process*(s,e,0,0):false;

}

//函数的意义，从str的i位置开始一直到结束位置的子串，是否能被

//exp的j位置开始一直到结束位置的子串

public static boolean process(char[] s,char[] e,int i,int j) {

if(j==e.length) {

return i==s.length;

}

if(j+1==e.length||e[j+1]!='\*') {

return i!=s.length&&(e[j]==s[i]||e[j]=='.')

&&*process*(s,e,i+1,j+1);

}

while(i!=s.length&&(e[j]==s[i]||e[j]=='.')) {

if(*process*(s,e,i,j+2)) {

return true;

}

i++;

}

//若s[i]和e[j]不能匹配，则使e[j+1]==\*字符的前一个字符的数量为0，

return *process*(s,e,i,j+2);

}

public static boolean isMatchDP(String str,String exp) {

if(str==null||exp==null) {

return false;

}

char[] s=str.toCharArray();

char[] e=exp.toCharArray();

if(!*isValid*(s,e)) {

return false;

}

boolean[][] dp=*initDPMap*(s,e);

for(int i=s.length-1;i>-1;i--) {

for(int j=e.length-2;j>-1;j--) {

if(e[j+1]!='\*') {

dp[i][j]=(e[j]==s[i]||e[j]=='.')

&&dp[i+1][j+1];

}else {

int si=i;

while(si!=s.length&&(e[j]==s[si]||e[j]=='.')) {

if(dp[si][j+2]) {

dp[i][j]=true;

break;

}

si++;

}

if(dp[i][j]!=true) {

dp[i][j]=dp[si][j+2];

}

}

}

}

return dp[0][0];

}

//倒数两列和最后一行

public static boolean[][] initDPMap(char[] s,char[] e){

int slen=s.length;

int elen=e.length;

boolean[][] dp=new boolean[slen+1][elen+1];

dp[slen][elen]=true;

for(int j=elen-2;j>-1;j=j-2) {

if(e[j]!='\*'&&e[j+1]=='\*') {

dp[slen][j]=true;

}else {

break;

}

}

if(slen>0&&elen>0) {

if((e[elen-1]=='.'||s[slen-1]==e[elen-1])) {

dp[slen-1][elen-1]=true;

}

}

return dp;

}

public static void main(String[] args) {

String str = "abcccdefg";

String exp = "ab.\*d.\*e.\*";

System.*out*.println(*isMatch*(str, exp));

System.*out*.println(*isMatchDP*(str, exp));

}

}

19.最长递增子序列

package advancedclass\_07;

public class Code\_02\_LIS {

//时间复杂度O(N\*N)

//dp[i]表示在以arr[i]为结尾的情况下，arr[0..i]中的最大递增子序列长度

public static int[] getdp1(int[] arr) {

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

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

dp[i]=1;

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

if(arr[i]>arr[j]) {

dp[i]=Math.*max*(dp[i], dp[j]+1);

}

}

}

return dp;

}

public static int[] generateLIS(int[] arr,int[] dp) {

int len=0;

int index=0;

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

if(dp[i]>len) {

len=dp[i];//找到最大值及位置

index=i;

}

}

int[] lis=new int[len];

lis[--len]=arr[index];

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

if(arr[i]<arr[index]&&dp[i]==dp[index]-1) {

lis[--len]=arr[i];

index=i;

}

}

return lis;

}

public static int[] lis1(int[] arr) {

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

return null;

}

int[] dp = *getdp1*(arr);

return *generateLIS*(arr, dp);

}

//时间复杂度O(N\*logN)

//利用二分查找来优化生成dp数组

public static int[] getdp2(int[] arr) {

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

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

ends[0]=arr[0];

dp[0]=1;

int right=0;

int l=0;

int r=0;

int m=0;

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

l=0;

r=right;

while(l<=r) {

m=(l+r)/2;

if(arr[i]>ends[m]) {

l=m+1;

}else {

r=m-1;

}

}

right=Math.*max*(right, l);

ends[l]=arr[i];

dp[i]=l+1;

}

return dp;

}

public static int[] lis2(int[] arr) {

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

return null;

}

int[] dp = *getdp2*(arr);

return *generateLIS*(arr, dp);

}

public static void printArray(int[] arr) {

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

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

}

System.*out*.println();

}

public static void main(String[] args) {

int[] arr = { 2, 1, 5, 3, 6, 4, 8, 9, 7 };

*printArray*(arr);

*printArray*(*lis1*(arr));

*printArray*(*lis2*(arr));

}

}

20.最长公共子序列

package advancedclass\_07;

public class Code\_03\_LCSubsequence {

//动态规划

//dp[i][j]代表str1[0..i]与str2[0..j]的最长公共子序列长度

public static int[][] getdp(char[] str1,char[] str2){

int[][] dp=new int[str1.length][str2.length];

dp[0][0]=str1[0]==str2[0]?1:0;

//第一列

for(int i=1;i<str1.length;i++) {

dp[i][0]=Math.*max*(dp[i-1][0], str1[i]==str2[0]?1:0);

}

//第一行

for(int j=1;j<str2.length;j++) {

dp[0][j]=Math.*max*(dp[0][j-1], str1[0]==str2[j]?1:0);

}

for(int i=1;i<str1.length;i++) {

for(int j=1;j<str2.length;j++) {

dp[i][j]=Math.*max*(dp[i-1][j], dp[i][j-1]);

if(str1[i]==str2[j]) {

dp[i][j]=Math.*max*(dp[i][j], dp[i-1][j-1]+1);

}

}

}

return dp;

}

public static String lcse(String str1,String str2) {

if(str1==null||str2==null||str1.equals("")||str2.equals("")) {

return "";

}

char[] chs1=str1.toCharArray();

char[] chs2=str2.toCharArray();

int[][] dp=*getdp*(chs1,chs2);

int m=chs1.length-1;

int n=chs2.length-1;

char[] res=new char[dp[m][n]];

int index=res.length-1;

//通过dp求解最长公共子序列的过程就是还原出当时如何求解dp的过程

while(index>=0) {

if(n>0&&dp[m][n]==dp[m][n-1]) {

n--;//向左移动

}else if(m>0&&dp[m][n]==dp[m-1][n]) {

m--;//向上移动

}else {//选择了决策dp[i-1][j-1]+1

res[index--]=chs1[m];

m--;

n--;

}

}

return String.*valueOf*(res);

}

public static void main(String[] args) {

String str1 = "A1BC2D3EFGH45I6JK7LMN";

String str2 = "12OPQ3RST4U5V6W7XYZ";

System.*out*.println(*lcse*(str1, str2));

}

}

21.最长公共子串

package advancedclass\_07;

public class Code\_04\_LCSubstring {

//dp[i][j]代表在必须把str[i]和str[j]当做公共子串最后一个字符时，

//公共子串最长能有多长

public static int[][] getdp(char[] str1,char[] str2){

int[][] dp=new int [str1.length][str2.length];

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

if(str1[i]==str2[0]) {

dp[i][0]=1;

}

}

for(int j=1;j<str2.length;j++) {

if(str1[0]==str2[j]) {

dp[0][j]=1;

}

}

for(int i=1;i<str1.length;i++) {

for(int j=1;j<str2.length;j++) {

if(str1[i]==str2[j]) {//向左扩多大长度

dp[i][j]=dp[i-1][j-1]+1;

}

}

}

return dp;

}

//额外空间复杂度O(M\*N)

public static String lcst1(String str1, String str2) {

if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {

return "";

}

char[] chs1 = str1.toCharArray();

char[] chs2 = str2.toCharArray();

int[][] dp = *getdp*(chs1, chs2);

int end=0;

int max=0;

for(int i=0;i<chs1.length;i++) {

for(int j=0;j<chs2.length;j++) {

if(dp[i][j]>max) {

end=i;

max=dp[i][j];

}

}

}

return str1.substring(end-max+1,end+1);

}

public static String lcst2(String str1, String str2) {

if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {

return "";

}

char[] chs1 = str1.toCharArray();

char[] chs2 = str2.toCharArray();

int row=0;//斜线开始的行

int col=chs2.length-1;//斜线开始的列

int max=0;//记录最大长度

int end=0;//最大长度更新时，记录子串的结尾位置

while(row<chs1.length) {

int i=row;

int j=col;

int len=0;

//从(i,j)开始向右下方遍历

while(i<chs1.length&&j<chs2.length) {

if(chs1[i]!=chs2[j]) {

len=0;

}else {

len++;

}

//记录最大值，以及结束字符的位置

if(len>max) {

end=i;

max=len;

}

i++;

j++;

}

if(col>0) {//斜线开始位置的列先向左移动

col--;

}else {//列移动到最左之后，行向下移动

row++;

}

}

return str1.substring(end-max+1,end+1);

}

public static void main(String[] args) {

String str1 = "ABC1234567DEFG";

String str2 = "HIJKL1234567MNOP";

System.*out*.println(*lcst1*(str1, str2));

System.*out*.println(*lcst2*(str1, str2));

}

}

22.最小编辑代价

package advancedclass\_07;

public class Code\_05\_EditCost {

//时间复杂度O(M\*N) 额外空间复杂度为O(M\*N)

//dp的大小为(M+1)\*(N+1)

//dp[i][j]表示str1[0..i-1]编辑成str2[0..j-1]的最小代价

public static int minCost1(String str1,String str2,int ic,int dc,int rc) {

if(str1==null||str2==null) {

return 0;

}

char[] chs1=str1.toCharArray();

char[] chs2=str2.toCharArray();

int row=chs1.length+1;

int col=chs2.length+1;

int[][] dp=new int[row][col];

for(int i=1;i<row;i++) {

dp[i][0]=dc\*i;

}

for(int j=1;j<col;j++) {

dp[0][j]=ic\*j;

}

for(int i=1;i<row;i++) {

for(int j=1;j<col;j++) {

if(chs1[i-1]==chs2[j-1]) {

dp[i][j]=dp[i-1][j-1];

}else {

dp[i][j]=dp[i-1][j-1]+rc;

}

dp[i][j]=Math.*min*(dp[i][j], dp[i][j-1]+ic);

dp[i][j]=Math.*min*(dp[i][j], dp[i-1][j]+dc);

}

}

return dp[row-1][col-1];

}

public static void main(String[] args) {

String str1 = "ab12cd3";

String str2 = "abcdf";

System.*out*.println(*minCost1*(str1, str2, 5, 3, 2));

}

}

23.回文子串的最小分割数

package advancedclass\_07;

public class Code\_06\_PalindromeMinCut {

public static int minCut(String str) {

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

return 0;

}

char[] chas=str.toCharArray();

int len=chas.length;

//dp[i]代表子串str[i..len+1]至少需要切割几次，

//才能把str[i..len+1]全部切成回文子串

int[] dp=new int[len+1];

//p[i][j]为true,字符串str[i..j]是回文串

//p[i][j]为true,一定是以下三种情况

//str[i..j]有一个字符组成

//str[i..j]有2个字符组成且两个字符相等

//str[i+1..j-1]为回文串，且str[i]==str[j]

boolean[][]p=new boolean[len][len];

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

dp[i]=Integer.*MAX\_VALUE*;

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

if(chas[i]==chas[j]&&(j-i<2||p[i+1][j-1])) {

p[i][j]=true;

dp[i]=Math.*min*(dp[i], dp[j+1]+1);

}

}

}

return dp[0];

}

// for test

public static String getRandomStringOnlyAToD(int len) {

int range = 'D' - 'A' + 1;

char[] charArr = new char[(int) (Math.*random*() \* (len + 1))];

for (int i = 0; i != charArr.length; i++) {

charArr[i] = (char) ((int) (Math.*random*() \* range) + 'A');

}

return String.*valueOf*(charArr);

}

public static void main(String[] args) {

int maxLen = 10;

int testTimes = 5;

String str = null;

for (int i = 0; i != testTimes; i++) {

str = *getRandomStringOnlyAToD*(maxLen);

System.*out*.print("\"" + str + "\"" + " : ");

System.*out*.println(*minCut*(str));

}

}

}

24.字符串的交错组成

package advancedclass\_07;

public class Code\_07\_StringCross {

public static boolean isCross1(String str1,String str2,String aim) {

if(str1==null||str2==null||aim==null) {

return false;

}

char[] ch1=str1.toCharArray();

char[] ch2=str2.toCharArray();

char[] chaim=aim.toCharArray();

if(chaim.length!=ch1.length+ch2.length) {

return false;

}

boolean[][] dp=new boolean[ch1.length+1][ch2.length+1];

dp[0][0]=true;

for(int i=1;i<=ch1.length;i++) {

if(ch1[i-1]!=chaim[i-1]) {

break;

}

dp[i][0]=true;

}

for(int j=1;j<=ch2.length;j++) {

if(ch2[j-1]!=chaim[j-1]) {

break;

}

dp[0][j]=true;

}

for(int i=1;i<=ch1.length;i++) {

for(int j=1;j<=ch2.length;j++) {

if((ch1[i-1]==chaim[i+j-1]&&dp[i-1][j])

||(ch2[j-1]==chaim[i+j-1]&&dp[i][j-1])) {

dp[i][j]=true;

}

}

}

return dp[ch1.length][ch2.length];

}

public static void main(String[] args) {

String str1 = "1234";

String str2 = "abcd";

String aim = "1a23bcd4";

System.*out*.println(*isCross1*(str1, str2, aim));

}

}

25.有效括号长度

package advancedclass\_07;

public class Code\_08\_ParenthesesProblem {

public static boolean isValid(String str) {

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

return false;

}

char[] chas=str.toCharArray();

int status=0;

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

if(chas[i]!=')'&&chas[i]!='(') {

return false;

}

//')'更多，直接返回false

if(chas[i]==')'&&--status<0) {

return false;

}

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

status++;

}

}

return status==0;

}

//补充问题

// dp[]长度和字符串的长度一样

//dp[i]代表str[0..i]中必须以字符str[i]结尾的最长的有效括号长度

public static int maxLength(String str) {

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

return 0;

}

char[] chas=str.toCharArray();

int[] dp=new int[chas.length];

int pre=0;

int res=0;

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

//chas[i]=='(' ,有效字符串必然以')'结尾，故dp[i]=0

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

pre=i-dp[i-1]-1;

//str[i]与chas[pre]配成一对，还应该吧dp[pre-1]加上

if(pre>=0&&chas[pre]=='(') {

dp[i]=dp[i-1]+2+(pre>0?dp[pre-1]:0);

}

}

res=Math.*max*(res, dp[i]);

}

return res;

}

public static void main(String[] args) {

String str1 = "((())())";

System.*out*.println(*isValid*(str1));

System.*out*.println(*maxLength*(str1));

}

}

26.子数组的最大累加和

package advancedclass\_08;

public class Code\_02\_MaxSubArraySum {

public static int maxSum(int[] arr) {

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

return 0;

}

int max=Integer.*MIN\_VALUE*;

int cur=0;

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

cur+=arr[i];

max=Math.*max*(max, cur);

//当cur<0时，说明累加到当前数出现小于0的结果，

//那么累加的这一部分肯定不能作为产生最大累加和的子数组的左边部分

cur=cur<0?0:cur;

}

return max;

}

}

27.子矩阵的最大累加和

package advancedclass\_08;

public class Code\_02\_MaxSubMatrixSum {

public static int maxSum(int[][] m) {

if(m==null||m.length==0||m[0].length==0) {

return 0;

}

int max=Integer.*MIN\_VALUE*;

int cur=0;

int[]s=null;//累加数组

for(int i=0;i!=m.length;i++) {

s=new int[m[0].length];

//以每一行为开头的往下所有的子矩阵

for(int j=i;j!=m.length;j++) {

cur=0;

//用求子数组的最大累加和的方式得到每一步的最大子矩阵的累加和

for(int k=0;k!=s.length;k++) {

//利用前一步求出的累加数组

s[k]+=m[j][k];

cur+=s[k];

max=Math.*max*(max, cur);

cur=cur<0?0:cur;

}

}

}

return max;

}

public static void main(String[] args) {

int[][] matrix = { { -90, 48, 78 }, { 64, -40, 64 }, { -81, -7, 66 } };

System.*out*.println(*maxSum*(matrix));

}

}

28.边界都是1的正方形大小

package advancedclass\_08;

public class Code\_03\_MaxOneBorderSize {

//时间复杂度O(N^3)

//预处理矩阵

public static void setBorderMap(int[][] m,int[][] right,int[][] down) {

int r=m.length;

int c=m[0].length;

//right[i][j]的值表示从位置(i,j)出发向右，有多少个连续的1

//down[i][j]的值表示从位置(i,j)出发向下，有多少个连续的1

if(m[r-1][c-1]==1) {

right[r-1][c-1]=1;

down[r-1][c-1]=1;

}

//从矩阵的右下角开始向上计算

for(int i=r-2;i!=-1;i--) {

if(m[i][c-1]==1) {

right[i][c-1]=1;

down[i][c-1]=down[i+1][c-1]+1;

}

}

//从矩阵的右下角开始向左计算

for(int i=c-2;i!=-1;i--) {

if(m[r-1][i]==1) {

right[r-1][i]=right[r-1][i+1]+1;

down[r-1][i]=1;

}

}

//剩下的位置

for(int i=r-2;i!=-1;i--) {

for(int j=c-2;j!=-1;j--) {

if(m[i][i]==1) {

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

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

}

}

}

}

public static int getMaxSize(int[][] m) {

int[][] right=new int[m.length][m[0].length];

int[][] down=new int[m.length][m.length];

*setBorderMap*(m,right,down);

//边长从大到小

for(int size=Math.*min*(m.length, m.length);size!=0;size--) {

if(*hasSizeOfBorder*(size,right,down)) {

return size;

}

}

return 0;

}

public static boolean hasSizeOfBorder(int size,int[][] right,int[][] down) {

for(int i=0;i!=right.length-size+1;i++) {

for(int j=0;i!=right[0].length-size+1;j++) {

if(right[i][j]>=size&&down[i][j]>=size//正方形的左上角

&& right[i+size-1][j]>=size//左下角

&& down[i][j+size-1]>=size){//右上角

return true;

}

}

}

return false;

}

}

29.找到最长无重复字符子串

package advancedclass\_08;

public class Code\_05\_LongestNoRepeatSubstring {

public static int maxUnique(String str) {

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

return 0;

}

char[] chas=str.toCharArray();

int[] map=new int[256];

//map(str[i])的值表示在之前的遍历中最近一次出现str[i]字符的位置

for(int i=0;i<256;i++) {

map[i]=-1;

}

int len=0;

//遍历到str[i],pre表示以str[i-1]结尾的情况下，

//最长无重复字符子串开始位置的前一个位置

int pre=-1;

int cur=0;

for(int i=0;i!=chas.length;i++) {

pre=Math.*max*(pre, map[chas[i]]);

cur=i-pre;

len=Math.*max*(len,cur);

map[chas[i]]=i;

}

return len;

}

public static void main(String[] args) {

String str = "abcvda";

System.*out*.println(*maxUnique*(str));

}

}

30.两个有序数组间相加和的TOPK问题

package advancedclass\_08;

import java.util.Arrays;

import java.util.HashSet;

public class Code\_01\_TopKSumCrossTwoArrays {

public static class HeapNode{

public int row;

public int col;

public int value;

public HeapNode(int row,int col,int value) {

this.row=row;

this.col=col;

this.value=value;

}

}

public static int[] topKSum(int[] a1,int[] a2,int topK) {

if(a1==null||a2==null||topK<1) {

return null;

}

topK=Math.*min*(topK, a1.length\*a2.length);

HeapNode[] heap=new HeapNode[topK+1];//注意要加1

int heapSize=0;

int headR=a1.length-1;

int headC=a2.length-1;

int uR=-1;

int uC=-1;

int lR=-1;

int lC=-1;

*heapInsert*(heap,heapSize++,headR,headC,a1[headR]+a2[headC]);

HashSet<String> positionSet=new HashSet<String>();

int[] res=new int[topK];

int resIndex=0;

while(resIndex!=topK) {

HeapNode head=*popHead*(heap,heapSize--);

res[resIndex++]=head.value;

headR=head.row;

headC=head.col;

uR=headR-1;

uC=headC;

if(headR!=0&&!*isContains*(uR,uC,positionSet)) {

*heapInsert*(heap,heapSize++,uR,uC,a1[uR]+a2[uC]);

*addPositonToSet*(uR,uC,positionSet);

}

lR=headR;

lC=headC-1;

if(headR!=0&&!*isContains*(lR,lC,positionSet)) {

*heapInsert*(heap,heapSize++,lR,lC,a1[lR]+a2[lC]);

*addPositonToSet*(lR,lC,positionSet);

}

}

return res;

}

public static HeapNode popHead(HeapNode[] heap,int heapSize) {

HeapNode res=heap[0];

*swap*(heap,0,heapSize-1);

heap[--heapSize]=null;

*heapify*(heap,0,heapSize);

return res;

}

public static void heapify(HeapNode[] heap,int index,int heapSize) {

int left=index\*2+1;

int right=index\*2+2;

int largest=index;

while(left<heapSize) {

if(heap[left].value>heap[right].value) {

largest=left;

}

if(right<heapSize&&heap[right].value>heap[largest].value) {

largest=right;

}

if(largest!=index) {

*swap*(heap,largest,index);

}else {

break;

}

index=largest;

left=index\*2+1;

right=index\*2+2;

}

}

public static void heapInsert(HeapNode[] heap,int index,int row,int col,int value) {

heap[index]=new HeapNode(row,col,value);

int parent =(index-1)/2;

while(index!=0) {

if(heap[index].value>heap[parent].value) {

*swap*(heap,parent,index);

index=parent;

parent=(index-1)/2;

}else {

break;

}

}

}

public static void swap(HeapNode[] heap,int index1,int index2) {

HeapNode tmp=heap[index1];

heap[index1]=heap[index2];

heap[index2]=tmp;

}

public static boolean isContains(int row,int col,HashSet<String> set) {

return set.contains(String.*valueOf*(row+"\_"+col));

}

public static void addPositonToSet(int row,int col,HashSet<String> set) {

set.add(String.*valueOf*(row+"\_"+col));

}

}

31.未排序正数数组累加和为给定值的最长子数组长度

package advancedclass\_04;

public class Code\_06\_zhengshuLongestSumSubArrayLength {

public static int getMaxLength(int[] arr,int k) {

if(arr==null||arr.length==0||k<=0) {

return 0;

}

int left=0;

int right=0;

int sum=arr[0];

int len=0;

while(right<arr.length) {

if(sum==k) {

len=Math.*max*(len, right-left+1);

sum-=arr[left++];

} else if(sum<k) {

right++;

if(right==arr.length) {

break;

}

sum+=arr[right];

}else {

sum-=arr[left++];

}

}

return len;

}

}

32.未排序数组累加和为给定值的最长子数组长度

package advancedclass\_04;

import java.util.HashMap;

public class Code\_05\_LongestSumSubArrayLength {//数组元素可正 可负 可0

public static int maxLength(int[] arr, int k) {

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

return 0;

}

HashMap<Integer,Integer> map=new HashMap <Integer,Integer>();

//key:累加和 value:最早出现的位置

map.put(0, -1);

int len=0;

int sum=0;

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

sum+=arr[i];

if (map.containsKey(sum-k)) {

len=Math.*max*(i-map.get(sum-k), len);

}

if(!map.containsKey(sum)) {

map.put(sum, i);

}

}

return len;

}

public static int[] generateArray(int size) {

int[] result = new int[size];

for (int i = 0; i != size; i++) {

result[i] = (int) (Math.*random*() \* 11) - 5;

}

return result;

}

public static void printArray(int[] arr) {

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

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

}

System.*out*.println();

}

public static void main(String[] args) {

int[] arr = *generateArray*(20);

*printArray*(arr);

System.*out*.println(*maxLength*(arr, 10));

}

}

33.未排序数组累加和小于或等于给定值的最长子数组长度

package advancedclass\_04;

public class Code\_07\_LongestSumequallessSubArrayLength {

public static int maxLength(int[] arr,int k) {

int[] h=new int[arr.length+1];

int sum=0;

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

sum+=arr[i];

h[i+1]=Math.*max*(sum, h[i]);

}

sum=0;

int res=0;

int pre=0;

int len=0;

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

sum+=arr[i];

pre=*getLessIndex*(h,sum-k);

len=pre==-1?0:i-pre+1;

res=Math.*max*(res, len);

}

return res;

}

public static int getLessIndex(int[] arr,int num) {

int low=0;

int high=arr.length-1;

int mid=0;

int res=-1;

while(low<=high) {

mid=(low+high)/2;

if(arr[mid]>=num) {

res=mid;

high=mid-1;

}else {

res=mid;

low=mid+1;

}

}

return res;

}

public static int[] generateArray(int size) {

int[] result = new int[size];

for (int i = 0; i != size; i++) {

result[i] = (int) (Math.*random*() \* 11) - 5;

}

return result;

}

public static void printArray(int[] arr) {

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

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

}

System.*out*.println();

}

public static void main(String[] args) {

int[] arr = *generateArray*(5);

*printArray*(arr);

System.*out*.println(*maxLength*(arr, 7));

}

}

package advancedclass\_06;

public class Code\_05\_LongestSubarrayLessSumAwesomeSolution {

public static int maxLengthAwesome(int[] arr,int aim) {

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

return 0;

}

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

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

//从后向前

//sums[],以该位置为开头的子数组的最小累加和

//ends[]获得这个最小累加和的右边界

sums[arr.length-1]=arr[arr.length-1];

ends[arr.length-1]=arr.length-1;

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

if(sums[i+1]<0) {

sums[i]=arr[i]+sums[i+1];

ends[i]=ends[i+1];

}else {

sums[i]=arr[i];

ends[i]=i;

}

}

int R=0;

int sum=0;

int len=0;

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

while(R<arr.length&&sum+sums[R]<=aim) {

sum+=sums[R];

R=ends[R]+1;

}

sum-=R>start?arr[start]:0;

len=Math.*max*(len, R-start);

R=Math.*max*(R, start+1);

}

return len;

}

public static void main(String[] args) {

int[] arr = { 5, -10,-25,10 };

int aim =6 ;

System.*out*.println(*maxLengthAwesome*(arr,aim));

}

}