函数参数传递：

基本数据类型：值传递

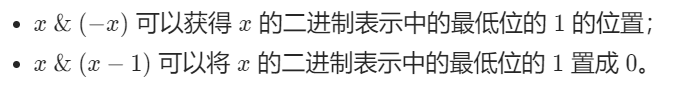
数组：引用传递

对象类型：引用传递

String对象: **值传递**，因为其内部的字符保存数组为final

位操作

((1<<n)-1)&(~column) 这个操作可以选取出n bit范围内，column位为0的位置。如n=4，1111，column=2，0010，结果就是13，1101



Integer.bitCount(num) 获得num二进制中1的数量，num只有一个1时，可以用这个函数来获取这个1的位置。如0100，Integer.bitCount(num-1)=2.

字符串：

str.charAt(i) //访问下标为i的字符

str.equals(str1) //判断两个字符串是否相等

str.toLowerCase() //把str字符串转换成小写形式

str.trim() //把str字符串首尾的空格字符消去

char[] sb = str.toCharArray(); //把String转换成char数组，对String的修改操作之前可以用这个函数来转化，转化后可以修改了。

String.join("/",stack),stack是存储字符串的栈，这个可以把stack里的字符串用"/"连接起来并返回一个字符串。是一个静态方法。

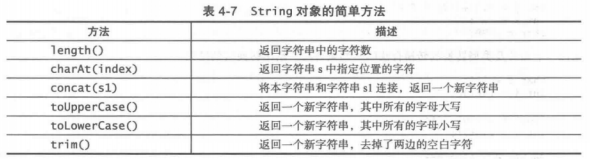
str.split("/")，把str字符串根据“/”分成一个字符串数组。

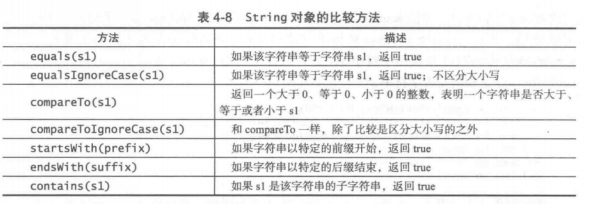
int index=str1.indexOf(str2); //在str1中查找str2并返回下标，没有找到返回-1，str2是空串返回0

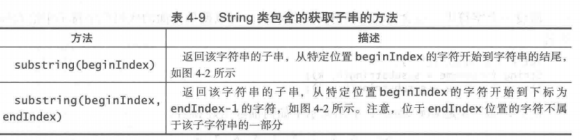
String subs=str1.substring(start,end); //返回下标为start到end之间的字串（包括start，不包括end）

int len=subs.length(); //返回字符串长度

subs.equals(str1)； //比较两个字符串是否一样，'=='和'equals'都是比较对象的地址，只是equals可以重写



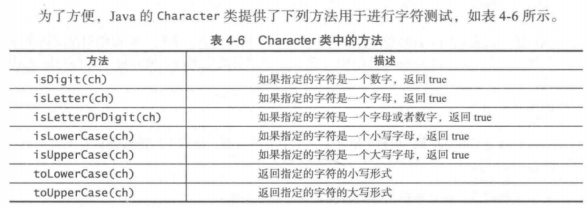




字符：

Character.isLetterOrDigit(ch) //判断ch字符是不是字母或者数字字符

Character.toLowerCase(ch) //把ch字符转换成小写



不可以直接修改String的某一位字符，可以用StringBuilder和StringBuffer来处理

StringBuilder：**非线程安全**(单线程中效率比StringBuffer高)

1. StringBuilder s = new StringBuilder();

2. s.insert(0,‘a’);

3. s.toString()

4.s.append(’a‘) / s.append(“haha”)

5.s.setCharAt(0,’b’)

7.s.delete(0,s.length())

StringBuffer: **线程安全**

1. StringBuffer sb = new StringBuffer();
2. sb.append(‘a’)
3. sb.reverse().toString() //sb内容反转之后转换成字符串

数组：

num.length //得到数组的长度

int[] price = new int[4]; //初始化数组

Arrays.fill(price, 1)

Arrays.sort（price）//数组排序

int[] Array=new int[]{1,2,3} //初始化

函数中可以返回局部数组。

Integer是对象

int是基本数据类型

(char)(26+'A') //保存为char型，否则是int型

Integer.toString(5) //把数字5转换成字符串”5”

Integer.parseInt(“123”) //把字符串“123”转换成数字123

空对象：null

Integer类型最大值：Integer.MAX\_VALUE

O(logn!)=O(nlogn)

List是一个接口，LinkedList和ArrayList是实现类：

List<Integer> res = new LinkedList<Integer>();

List<Integer> res1 = new ArrayList<Integer>();

res.add(0,1); //再列表表头添加数字1。

res.get(0); //读取list中位置0的值

res.set(0,2); //设置list中位置0的值为2

Set<Integer> seen = new HashSet<Integer>()

seen.contains(n) //是否包含数字n

seen.add(n) //添加数字n

map

Map<Character,Character> s2t = new HashMap<Character,Character>();//其中的元素要引用类型

s2t.containsKey(x) //查询是否有x的key

s2t.get(x) //获取key为x的value值

s2t.put(x,y) //放入数据，key为x，value为y

s2t. getOrDefault(x,’0’) //获取key为x对应的value值，如果不存在就返回‘0’

Queue

Queue<Integer> q1 = new LinkedList<Integer>()

q1.offer(x) //队尾加入元素

q1.poll() //返回队首元素并删除

q1.peek() //返回队首元素

q1.isEmpty() //是否为空

队列：

Queue<TreeNode> que = new LinkedList<TreeNode>() //LinkedList实现了Queue接口；

que.offer(root) //向队列添加元素

TreeNode front = que.poll(); //弹出并返回队首元素。

Stack（与Queue不同，Stack是类，不是接口）

Stack<Integer> sta= new Stack<Integer>()

sta.push(x)

sta.pop()

sta.peek()

sta.empty()

可以使用Deque替代Stack：

Deque<TreeNode> stack = new LinkedList<>(); //基本操作和函数和Stack的差不多。

入栈：stack.offerLast(index);或者 stack.push(index);

出栈：stack.pollLast(); 或者 stack.poll()

出队列：stack.pollFirst();

入队列：stack.offerFirst();

PriorityQueue：

是基于优先堆的一个无界队列，**默认是最小堆**。动态增加数组，容量无限。

这个队列中的元素可以默认自然排序或者通过提供的Comparator在队列实例化时的排序。

maxHeap = new PriorityQueue<>();

minHeap = new PriorityQueue<>((a,b)->(a-b)); //最小堆

//下面也是最小堆，和上面的顺序不太一样

minHeap = new PriorityQueue<>(new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o1-o2;

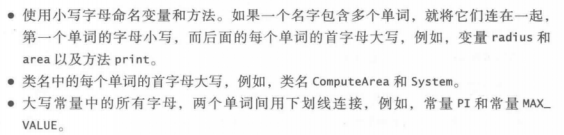
}

});

获取时间：



变量命名习惯：



算法：

动态规划：  
 1.无后效性：f(i, j)f(i,j) 和任何的 f(i', j')无关。(其中i' > i, j' > j)  
 2.最优子结构：即当前问题的最优解取决于子问题的最优解  
动态规划的题目分为两大类：  
 一种是求最优解类，典型问题是背包问题（最优子结构），  
 另一种就是计数类，比如这里的统计方案数的问题，它们都存在一定的递推性质。（当前问题的方案数取决于子问题的方案数）  
动态规划空间优化：滚动数组思想  
 当我们定义的状态在动态规划的转移方程中只和某几个状态相关的时候，就可以考虑这种优化方法  
 比如这里f(i,j) 只与f(i−1,j) 和f(i,j−1) 相关，并且循环是按照行来的。