

原始人工作室

FIRST
PEOPLE STUDIO



firstpeoplestudio@gmail.com

目录

第一篇 刀疤鸭之数据结构面试题.....	8
1.1. 简介	8
1.2. 面试题集合（一）	8
1.2.1. 把二元查找树转变成排序的双向链表.....	8
1.2.2. 下排每个数都是先前上排那十个数在下排出现的次数	11
1.2.3. 设计包含 min 函数的栈	14
1.2.4. 求子数组的最大和.....	20
1.2.5. 在二元树中找出和为某一值的所有路径	22
1.2.6. Top K 算法详细解析---百度面试	29
1.2.7. 翻转句子中单词的顺序.....	31
1.2.8. 判断整数序列是不是二元查找树的后序遍历结果	33
1.2.9. 查找最小的 K 个元素-使用最大堆.....	35
1.2.10. 求二叉树中节点的最大距离.....	37
1.3. 面试题集合（二）	40
1.3.1. 求 $1+2+\dots+n$	40
1.3.2. 输入一个单向链表，输出该链表中倒数第 k 个结点.....	44
1.3.3. 输入一个已经按升序排序过的数组和一个数字.....	46
1.3.4. 输入一颗二元查找树，将该树转换为它的镜像.....	48
1.3.5. 输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印.....	49
1.3.6. 在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b	52
1.3.7. n 个数字 (0,1,...,n-1) 形成一个圆圈.....	53
1.3.8. 定义 Fibonacci 数列.....	58
1.3.9. 左移递减数列查找某一个数.....	60
1.3.10. 对于一个整数矩阵，存在一种运算，对矩阵中任意元素加一时，需要其相邻（上下左右）某一个元素也加一	63
1.4. 面试题集合（三）	73
1.4.1. 递归和非递归俩种方法实现二叉树的前序遍历.....	73
1.4.2. 请修改 append 函数，利用这个函数实现.....	78
1.4.3. 有 n 个长为 m+1 的字符串	82
1.4.4. n 支队伍比赛	84

1.4.5.	求一个矩阵中最大的二维矩阵(元素和最大).....	86
1.4.6.	强大的和谐	90
1.4.7.	通过交换 a,b 中的元素, 使[序列 a 元素的和]与[序列 b 元素的和]之间的差 最小	94
1.4.8.	计算 1 到 N 的十进制数中 1 的出现次数	97
1.4.9.	栈的 push、pop 序列[数据结构].....	99
1.4.10.	统计整数二进制表示中 1 的个数.....	102
1.5.	面试题集合 (四)	104
1.5.1.	跳台阶问题	104
1.5.2.	左旋转字符串.....	105
1.5.3.	在字符串中找出连续最长的数字串	109
1.5.4.	链表操作.....	111
1.5.5.	有 4 张红色的牌和 4 张蓝色的牌.....	115
1.5.6.	输入两个整数 n 和 m, 从数列 1, 2, 3.....n 中 随意取几个数	116
1.5.7.	输入一个表示整数的字符串, 把该字符串转换成整数并输出.....	118
1.5.8.	给出一个数列, 找出其中最长的单调递减 (或递增) 子序列.....	121
1.5.9.	四对括号可以有多少种匹配排列方式.....	124
1.5.10.	输入一个正数 n, 输出所有和为 n 连续正数序列.....	125
1.6.	面试题集合 (五)	126
1.6.1.	输入一棵二元树的根结点, 求该树的深度.....	126
1.6.2.	输入一个字符串, 打印出该字符串中字符的所有排列	128
1.6.3.	输入一个整数数组, 调整数组中数字的顺序, 使得所有奇数位于数组的前 半部分, 所有偶数位于数组的后半部分	130
1.6.4.	给定链表的头指针和一个结点指针, 在 O(1)时间删除该结点.....	132
1.6.5.	输入一个链表的头结点, 从尾到头反到来输出每个结点的值.....	134
1.6.6.	用 C++设计一个不能被继承的类	136
1.6.7.	给定链表的头指针和一个结点指针, 在 O(1)时间删除该结点.....	138
1.6.8.	一个数组中除了两个数字之外, 其余数字均出现了两次.....	141
1.6.9.	两个单向链表, 找出它们的第一个公共结点	142
1.6.10.	输入两个字符串, 从第一字符串中删除第二个字符串中所有的字符 ...	147
1.7.	面试题集合 (六)	148
1.7.1.	寻找丑数.....	148
1.7.2.	输入数字 n, 按顺序输出从 1 最大的 n 位 10 进制数.....	152
1.7.3.	用递归颠倒一个栈.....	156

1.7.4.	从扑克牌中随机抽 5 张牌，判断是不是一个顺子.....	158
1.7.5.	把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 S.....	162
1.7.6.	排出的所有数字中最小.....	165
1.7.7.	数组的旋转	170
1.7.8.	给出一个函数来输出一个字符串的所有排列	171
1.7.9.	实现函数 double Power(double base,int exponent)	173
1.7.10.	更优的解法：	175
1.7.11.	单列模式.....	176
1.8.	面试题集合（七）	178
1.8.1.	找出该字符串中对称的子字符串的最大长度	178
1.8.2.	数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字	180
1.8.3.	输入二叉树中的两个结点，输出这两个结点在数中最低的共同父结点	180
1.8.4.	复杂链表.....	187
1.8.5.	链表面试题	190
1.8.6.	链表和数字的区别在哪里	197
1.8.7.	strstr()函数功能	198
1.8.8.	一个 int 数组，里面数据无任何限制，要求求出所有这样的数 a[i]，其左边的数都小于等于它，右边的数都大于等于它.....	199
1.8.9.	一个文件，内含一千万行字符串，每个字符串在 1K 以内，要求找出所有相反的串对，如 abc 和 cba。	200
1.8.10.	给出一个文件，里面包含两个字段{url、size}，即 url 为网址，size 为对应网址访问的次数	205
1.9.	面试题集合（八）	209
1.9.1.	给定一个存放整数的数组，重新排列数组使得数组左边为奇数，右边为偶数	209
1.9.2.	用 C 语言实现函数 void * memmove(void *dest,const void *src,size_t n)	210
1.9.3.	随机发生器	212
1.9.4.	搜索引擎.....	212
1.9.5.	已知一个字符串，比如 asderwsde,寻找其中的一个子字符串比如 sde 的个数，如果没有返回 0，有的话返回子字符串的个数	216
1.9.6.	编写一个程序，把一个有序整数数组放到二叉树中.....	218
1.9.7.	大整数数相乘的问题	220
1.9.8.	求最大连续递增数字串.....	221
1.9.9.	函数将字符串中的字符'*'移到串的前部分	222

1.9.10. 单链表，编程实现其逆转	223
1.10. 面试题集合（九）	225
1.10.1. 删除字符串中的数字并压缩字符串	225
1.10.2. 求两个串中的第一个最长子串（神州数码以前试题）	226
1.10.3. 不开辟用于交换数据的临时空间，如何完成字符串的逆序	228
1.10.4. 求随机数构成的数组中找到长度大于=3 的最长的等差数列.....	228
1.10.5. 外排序	230
1.10.6. 用递归的方法判断整数组 a[N]是不是升序排列	232
1.10.7. N 个鸡蛋放到 M 个篮子中，篮子不能为空	232
1.10.8. Hash	234
1.10.9. 如何迅速匹配兄弟字符串	242
1.10.10. 腾讯数组乘积赋值的问题	243
1.11. 面试题集合（十）	244
1.11.1. 有一个整数数组，请求出两两之差绝对值最小的值.....	244
1.11.2. 给出一个函数来合并两个字符串 A 和 B。字符串 A 的后几个字节和字符串 B 的前几个字节重叠	245
1.11.3. 编程实现两个正整数的除法（不能用除法操作符）	250
1.11.4. 平面上 N 个点，没两个点都确定一条直线，求出斜率最大的那条直线所通过的两个点.....	251
1.11.5. 字符串原地压缩	252
1.11.6. 一排 N（最大 1 M）个正整数+1 递增，乱序排列	253
1.11.7. 找出被重复的数字.....	254
1.11.8. Hashtable 和 HashMap 的区别	264
1.11.9. 用 1、2、2、3、4、5 这六个数字，写一个 main 函数，打印出所有不同的排列	268
1.11.10. 局部变量、全局变量和静态变量的含义	269
1.12. 面试题集合（十一）	272
1.12.1. 有两个双向循环链表 A, B, 知道其头指针为： pHeadA,pHeadB, 请写一函数将两链表中 data 值相同的结点删除.....	272
1.12.2. 找出两个字符串中最大公共子字符串,如"abccade","dgcadde"的最大子串为"cad"	274
1.12.3. 把十进制数(long 型)分别以二进制和十六进制形式输出，不能使用 printf 系列	275
1.12.4. 40 亿个整数	277

1.12.5. bitmap 减少 hash 算法所用空间.....	281
1.12.6. 定义一个类似函数的宏，宏运算的结果来表示大于和小于	285
1.12.7. 给定一个集合 A.....	286
1.12.8. 已知一个函数 f 可以等概率的得到 1-5 间的随机数，问怎么等概率的得到 1-7 的随机数.....	289
1.12.9. 判断一个自然数是否是某个数的平方.....	290
1.12.10. 一棵排序二叉树，令 $f=(\text{最大值}+\text{最小值})/2$ ，设计一个算法，找出距离 f 值最近、大于 f 值的结点。复杂度如果是 $O(n^2)$ 则不得分。	291
1.12.11. strstr 和 strncmp 源码实现.....	294
1.13. 面试题集合（十二）	295
1.13.1. 对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集 合的数字和是相等.....	295
1.13.1. 对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集 合的数字和是相.....	295
1.13.2. Topk	298
1.13.3. Collection	301
1.13.4. 输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字	311
1.13.5. 求集合的所有子集的算法.....	314
1.13.6. 将一个数中的偶数位 bit 和奇数位 bit 交换.....	318
1.13.7. 二分查找实现.....	318
1.13.8. 集合合并.....	322
1.13.9. 把求子集运算转换为组合问题	323
1.13.10. 算法设计.....	324
1.14. 面试题集合（十三）	327
1.14.1. 各种排序算法.....	327
1.15. 面试题集合（十四）	393
1.15.1. 判断图里有环.....	393
1.15.2. 整数的素数和分解问题.....	422
1.15.3. 求两个或 N 个数的最大公约数(gcd)和最小公倍数(lcm)的较优算法....	425
1.16. 面试题集合（十五）	426
1.16.1. ApplicationContext	426
1.16.2. ApplicationContext 事件传播	431
1.16.3. mysql 有多种存储引擎.....	433
1.16.4. 论 MySQL 何时使用索引，何时不使用索引.....	436

1.16.5. SQL 多表连接查询实现语句	439
1.17. 面试题集合（十六）	442
1.17.1. 12 个高矮不同的人,排成两排,每排必须是从矮到高排列,而且第二排比对应的第一排的人高,问排列方式有多少种	442
1.17.2. 毒酒.....	448
1.17.3. 用代码验证阿里巴巴的一道关于男女比例的面试题.....	448
1.17.4. 金币.....	451
1.17.5. 海盗.....	452
1.17.6. 1024.....	454
1.17.7. 最少零钱问题 最少硬币问题.....	455
1.17.8. 石子合并.....	456
1.18. 面试题集合（十七）	460
1.18.1. 生产者-消费者模式	460
1.18.2. 动态规划.....	466
1.18.3. 01 背包	472
1.18.4. 贪心算法.....	477
1.18.5. 装箱问题.....	482
1.19. 教你如何迅速秒杀掉：99%的海量数据处理面试题	484
1.20. 附录.....	498

第一篇 刀疤鸭之数据结构面试题

1.1. 简介

数据的逻辑结构：指反应数据元素之间的逻辑关系的数据结构，其中的逻辑关系是指数据元素之间的前后件关系，而与他们在计算机中的存储位置无关。数据结构是每个程序员面试必须掌握的基础。

1.2. 面试题集合（一）

1.2.1. 把二元查找树转变成排序的双向链表

题目：

输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。

要求不能创建任何新的结点，只调整指针的指向。

```
10
/\ 
6 14
/\ \
4 8 12 16
```

转换成双向链表

4=6=8=10=12=14=16。

首先我们定义的二元查找树节点的数据结构如下：

```
struct BSTreeNode
{
    int m_nValue; // value of node
    BSTreeNode *m_pLeft; // left child of node
    BSTreeNode *m_pRight; // right child of node
};
```

1. // 1: 构造二叉查找树；
2. // 2: 中序遍历二叉查找树，因此结点按从小到大顺序访问，假设之前访问过的结点已经调整为一个双向链表，那么
3. // 只需要将当前结点连接至双向链表的最后一个结点即可，访问完后，双向链表也就调整完了

```
4. #include <iostream>
5. using namespace std;
6. struct BSTreeNode
7. {
8.     int m_nValue;// value of node
9.     BSTreeNode *m_pLeft;// left child of node
10.    BSTreeNode *m_pRight;// right child of node
11. };
12. void addBSTreeNode(BSTreeNode *&pCurrent, int value);
13. void inOrderBSTree(BSTreeNode* pBSTree);
14. void convertToDoubleList(BSTreeNode* pCurrent);
15. BSTreeNode *pHead=NULL;//指向循环队列头结点
16. BSTreeNode *pIndex=NULL;//指向前一个结点
17. int main()
18. {
19.     BSTreeNode *pRoot=NULL;
20.     addBSTreeNode(pRoot, 10);
21.     addBSTreeNode(pRoot, 6);
22.     addBSTreeNode(pRoot, 14);
23.     addBSTreeNode(pRoot, 4);
24.     addBSTreeNode(pRoot, 8);
25.     addBSTreeNode(pRoot, 12);
26.     addBSTreeNode(pRoot, 16);
27.     inOrderBSTree(pRoot);
28.     return 0;
29. }
30. ****
31. /* 建立二叉排序树 */
32. void addBSTreeNode(BSTreeNode *&pCurrent, int value)//在这个函数中会要改变指针值，一定要记得
   使用引用传递
33. {
34.     if(pCurrent==NULL)
35.     {
36.         BSTreeNode* pBSTree=newBSTreeNode();
37.         pBSTree->m_nValue=value;
```

```
38. pBSTree->m_pLeft=NULL;
39. pBSTree->m_pRight=NULL;
40. pCurrent=pBSTree;
41. }
42. elseif(pCurrent->m_nValue<value)
43. {
44. addBSTreeNode(pCurrent->m_pRight,value);
45. }
46. elseif(pCurrent->m_nValue>value)
47. {
48. addBSTreeNode(pCurrent->m_pLeft,value);
49. }
50. else
51. {
52. cout<<"node repeated"<<endl;
53. }
54. }
55. /*************************************************************************/
56. /*************************************************************************/
57. /* 中序遍历二叉树，同时调整结点指针 */
58. voidinOrderBSTree(BSTreeNode* pBSTree)
59. {
60. if(NULL==pBSTree)
61. {
62. return;
63. }
64. if(NULL!=pBSTree->m_pLeft)
65. {
66. inOrderBSTree(pBSTree->m_pLeft);
67. }
68. // if (NULL!=pBSTree)
69. // {
70. // cout<<pBSTree->m_nValue;
71. // }
72. convertToDoubleList(pBSTree);
```

```

73. if(NULL!=pBSTree->m_pRight)
74. {
75.     inOrderBSTree(pBSTree->m_pRight);
76. }
77. }
78. ****
79. ****
80. /* 调整结点指针 */
81. voidconvertToDoubleList(BSTreeNode* pCurrent)
82. {
83.     pCurrent->m_pLeft=pIndex;//使当前结点的左指针指向双向链表中最后一个结点
84.     if(NULL==pIndex)//若最后一个元素不存在, 此时双向链表尚未建立, 因此将当前结点设为双向链
表头结点
85.     {
86.         pHead=pCurrent;
87.     }
88.     else//使双向链表中最后一个结点的右指针指向当前结点
89.     {
90.         pIndex->m_pRight=pCurrent;
91.     }
92.     pIndex=pCurrent;//将当前结点设为双向链表中最后一个结点
93.     cout<<pCurrent->m_nValue<<" ";
94. }
95. ****

```

1.2.2. 下排每个数都是先前上排那十个数在下排出现的次数

给你 10 分钟时间, 根据上排给出十个数, 在其下排填出对应的十个数
要求下排每个数都是先前上排那十个数在下排出现的次数。

上排的十个数如下:

【0, 1, 2, 3, 4, 5, 6, 7, 8, 9】

举一个例子,

数值:0,1,2,3,4,5,6,7,8,9

分配:6,2,1,0,0,0,1,0,0,0

0 在下排出现了 6 次，1 在下排出现了 2 次，
2 在下排出现了 1 次，3 在下排出现了 0 次....
以此类推..

```
10 0 0 0 0 0 0 0 0  
9 0 0 0 0 0 0 0 1  
8 1 0 0 0 0 0 1 0  
7 2 1 0 0 0 1 0 0  
6 2 1 0 0 0 1 0 0  
6 2 1 0 0 0 1 0 0  
0 1 2 3 4 5 6 7 8 9  
6 2 1 0 0 0 1 0 0 0
```

[java] [view plain](#) [copy](#) [print?](#)

```
1.  /**
2.  * ...
3. */
4. class BottomArrGenerate {
5. /**
6. * 按规则生成数组
7. * @param 数组长度
8. * @return 生成的数组
9.*/
10. public static int[] generateArr(int n) {
11. if (n < 4) {
12. System.out.println("请输入不小于 4 的整数");
13. return null;
14. }
15. int[] top = new int[n]; // 0-n 一次排列的数字
16. int[] bottom = new int[n]; // 结果
17. for (int i = 0 ; i < top.length; i++)
18. top[i] = i;
19. for (int m = 0; m < n + 2; m++) {
20. boolean flag = true; // 找到结果标志
21. for (int i = 0; i < n; i++) { // 逐位对应
22. int count = getCount(i, bottom);
```

```
23. if (bottom[i] != count) { // 和上一次算出来的次数不符
24.     bottom[i] = count;
25.     flag = false;
26. }
27. }
28.
29. if (flag) // 连续 2 次得到的次数一样，则为正确结果
30.     break;
31. if (n + 1 == m && !flag) { // 算法有缺陷，n=5 时是有结果 2,1,2,0,0 的，底下结果如果有 2 个数字等于上面的数字就算不出来
32.     System.out.println("找不到结果");
33.     return null;
34. }
35. }
36.
37. return bottom;
38. }
39. /**
40. * 获得某数在数组中出现的次数
41. * @param 数
42. * @param 数组
43. * @return 数字在数组中出现次数
44. */
45. public static int getCount(int num, int[] arr) {
46.     int count = 0;
47.     for (int i = 0; i < arr.length; i++) {
48.         if (num == arr[i]) {
49.             count++;
50.         }
51.     }
52.     return count;
53. }
54. }
55.
56.
```

```
57. public class Six {  
58.     public static void main(String[] args) {  
59.         Scanner scan = new Scanner(System.in);  
60.         int[] a = null;  
61.         try {  
62.             a = BottomArrGenerate.generateArr(scan.nextInt());  
63.         } catch (InputMismatchException e) {  
64.             System.out.println("请输入整数");  
65.         }  
66.  
67.         if (a != null) {  
68.             for (int i = 0; i < a.length; i++) {  
69.                 System.out.print(i + "\t");  
70.             }  
71.             System.out.println();  
72.             for (int i : a) {  
73.                 System.out.print(i + "\t");  
74.             }  
75.         }  
76.     }  
77. }
```

1.2.3. 设计包含 min 函数的栈

[print?](#)

```
1. import java.util.ArrayList;  
2. import java.util.List;  
3.  
4.  
5. public class MinStack {  
6.  
7.     //maybe we can use origin array rather than ArrayList  
8.     private List<Integer> dataStack;  
9.     private List<Integer> auxStack;//store the position of MinElement  
10.
```

```
11.    public static void main(String[] args) {
12.        MinStack minStack=new MinStack();
13.        minStack.push(3);
14.        minStack.printStatus();
15.        minStack.push(4);
16.        minStack.printStatus();
17.        minStack.push(2);
18.        minStack.printStatus();
19.        minStack.push(1);
20.        minStack.printStatus();
21.        minStack.pop();
22.        minStack.printStatus();
23.        minStack.pop();
24.        minStack.printStatus();
25.        minStack.push(0);
26.        minStack.printStatus();
27.    }
28.
29.    public MinStack(){
30.        dataStack=new ArrayList<Integer>();
31.        auxStack=new ArrayList<Integer>();
32.    }
33.    public void push(int item){
34.        if(isEmpty()){
35.            dataStack.add(item);
36.            auxStack.add(0);//position=0
37.        }else{
38.            dataStack.add(item);
39.            int minIndex=auxStack.get(auxStack.size()-1);
40.            int min=dataStack.get(minIndex);
41.            if(min>item){
42.                auxStack.add(dataStack.size()-1);
43.            }else{
44.                auxStack.add(minIndex);
45.            }

```

```
46.    }
47.    }
48.    public int pop(){
49.        int top=-1;
50.        if(isEmpty()){
51.            System.out.println("no element,no pop");
52.        }else{
53.            auxStack.remove(auxStack.size()-1);
54.            top=dataStack.remove(dataStack.size()-1);
55.        }
56.        return top;
57.
58.    }
59.    public int min(){
60.        int min=-1;
61.        if(!isEmpty()){
62.            int minIndex=auxStack.get(auxStack.size()-1);
63.            min=dataStack.get(minIndex);
64.        }
65.        return min;
66.    }
67.    public boolean isEmpty(){
68.        return dataStack.size()==0;
69.    }
70.    public void printStatus(){
71.        System.out.println("数据栈      辅助栈      最小值");
72.        for(int i=0;i<dataStack.size();i++){
73.            System.out.print(dataStack.get(i)+",");
74.        }
75.        System.out.print("      ");
76.        for(int i=0;i<dataStack.size();i++){
77.            System.out.print(auxStack.get(i)+",");
78.        }
79.        System.out.print("      ");
80.        System.out.print(this.min());
```

```

81.     System.out.println();
82. }
83. /*
84. 步骤      数据栈      辅助栈      最小值
85. 1.push 3  3      0      3
86. 2.push 4  3,4    0,0    3
87. 3.push 2  3,4,2  0,0,2  2
88. 4.push 1  3,4,2,1 0,0,2,3  1
89. 5.pop    3,4,2  0,0,2    2
90. 6.pop    3,4    0,0    3
91. 7.push 0  3,4,0  0,0,2    0
92. */
93. }
```

题目：[定义栈的数据结构，要求添加一个 min 函数，能够得到栈的最小元素](#)。要求函数 min、push 以及 pop 的时间复杂度都是 O(1)。

分析：这是去年 google 的一道面试题。

我看到这道题目时，第一反应就是每次 push 一个新元素时，将栈里所有逆序元素排序。这样栈顶元素将是最小元素。但由于不能保证最后 push 进栈的元素最先出栈，这种思路设计的数据结构已经不是一个栈了。

在栈里添加一个成员变量存放最小元素（或最小元素的位置）。每次 push 一个新元素进栈的时候，如果该元素比当前的最小元素还要小，则更新最小元素。

乍一看这样思路挺好的。但仔细一想，该思路存在一个重要的问题：如果当前最小元素被 pop 出去，如何才能得到下一个最小元素？

因此仅仅只添加一个成员变量存放最小元素（或最小元素的位置）是不够的。我们需要一个辅助栈，每次 push 一个新元素的时候，同时将最小元素（或最小元素的位置，考虑到栈元素的类型可能是复杂的数据结构，用最小元素的位置将能减少空间消耗）push 到辅助栈中；每次 pop 一个元素出栈的时候，同时 pop 辅助栈。

参考代码：



```
#include <deque>
#include <assert.h>
```

```
template <typename T>
class CStackWithMin
```

```

{

public:
    CStackWithMin(void) {}

    virtual ~CStackWithMin(void) {}

    T& top(void);
    const T& top(void) const;

    void push(const T& value);

    void pop(void);

    const T& min(void) const;

private:
    stack<T> m_data;           // the elements of stack
    stack<size_t> m_minIndex;   // the indices of minimum elements
};

// get the last element of mutable stack
template <typename T>
T& CStackWithMin<T>::top() {
    return m_data.back();
}

// get the last element of non-mutable stack
template <typename T>
const T& CStackWithMin<T>::top() const {
    return m_data.back();
}

// insert an element at the end of stack
template <typename T>
void CStackWithMin<T>::push(const T& value) {
    // append the data into the end of m_data
    m_data.push_back(value);
}

```

```

// set the index of minimum elment in m_data at the end of m_minIndex

if(m_minIndex.size() == 0)

    m_minIndex.push_back(0);

else {

    if(value < m_data[m_minIndex.back()])

        m_minIndex.push_back(m_data.size() - 1);

    else

        m_minIndex.push_back(m_minIndex.back());

}

}

// erase the element at the end of stack

template <typename T>

void CStackWithMin<T>::pop() {

    // pop m_data

    m_data.pop_back();

    // pop m_minIndex

    m_minIndex.pop_back();

}

// get the minimum element of stack

template <typename T>

const T& CStackWithMin<T>::min() const {

    assert(m_data.size() > 0);

    assert(m_minIndex.size() > 0);

    return m_data[m_minIndex.back()];

}

```

举个例子演示上述代码的运行过程：

步骤 数据栈 辅助栈 最小值

1.push 3 3 0 3

2.push 4 3,4 0,0 3

3.push 2 3,4,2 0,0,2 2

4.push 1 3,4,2,1 0,0,2,3 1

5.pop 3,4,2 0,0,2 2

6.pop 3,4 0,0 3

7.push 0 3,4,0 0,0,2 0

1.2.4. 求子数组的最大和

题目：

输入一个整形数组，数组里有正数也有负数。

数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。

求所有子数组的和的最大值。要求时间复杂度为 $O(n)$ 。

例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5， 和最大的子数组为 3, 10, -4, 7, 2,

因此输出为该子数组的和 18。

[java] view plain copy

```
1. Java 代码
2. /**
3. *
4. */
5. package com.lhp;
6. /**
7. 3.求子数组的最大和
8. 题目：
9. 输入一个整形数组，数组里有正数也有负数。
10. 数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。
11. 求所有子数组的和的最大值。要求时间复杂度为 O(n)。
12. 例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5， 和最大的子数组为 3, 10, -4, 7, 2,
13. 因此输出为该子数组的和 18。
14. */
15. class GIntArrDispose {
16. /**
17. */
```

```
17. * 获得数组最大子数组的和
18. * @param 整形 数组
19. * @return 最大子数组的和
20. */
21. public static int getMaxChild(int[] intArr) {
22.     if (null == intArr || 0 == intArr.length) {
23.         throw new NullPointerException("GIntArrDispose.getMaxChild(int[]):传入没有内容的数
组!");
24.     }
25.     int result = 0; // 最大和
26.     int tempSum = 0; // 累加和
27.     int maxNegative = intArr[0]; // 数组最大值
28.     for (int i = 0; i < intArr.length; i++) {
29.         if (tempSum <= 0) {
30.             tempSum = intArr[i]; // 始终保持为正, 可以理解为排除最大子数组左边所有数的和
31.         } else {
32.             tempSum += intArr[i];
33.         }
34.         if (tempSum > result) {
35.             result = tempSum; // 始终保持最大, 可以理解为排除最大子数组右边所有数的和
36.         }
37.         if (intArr[i] > maxNegative) {
38.             maxNegative = intArr[i];
39.         }
40.     }
41.     if (maxNegative < 0) {
42.         return maxNegative; // 所有数都为负数, 只返回最大的一个数
43.     }
44.     return result;
45. }
```

```

46. public static synchronized void print(int[] intArr) {
47.     if (null == intArr || 0 == intArr.length) {
48.         throw new NullPointerException("GIntArrDispose.print(int[]):传入没有内容的数组!");
49.     }
50.     System.out.print("数组: ");
51.     for (int v : intArr) {
52.         System.out.print(v + " ");
53.     }
54.     System.out.println();
55. }
56. }

57. public class Three {
58.     public static void main(String[] args) {
59.         int[] intArr = {1, -2, 3, 10, -4, 7, 2, -5, 4};
60.         try {
61.             GIntArrDispose.print(intArr);
62.             System.out.print("最大子数组和: " + GIntArrDispose.getMaxChild(intArr));
63.         } catch (Exception e) {
64.             e.printStackTrace();
65.         }
66.     }
67. }

```

1.2.5. 在二元树中找出和为某一值的所有路径

在二元树中找出和为某一值的所有路径

题目：输入一个整数和一棵二元树。

从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。

打印出和与输入整数相等的所有路径。

例如输入整数 22 和如下二元树

```
/\ 
5 12
^
4 7
```

则打印出两条路径：10, 12 和 10, 5, 7。

二元树节点的数据结构定义为：

```
struct BinaryTreeNode // a node in the binary tree
{
    int m_nValue; // value of node
    BinaryTreeNode *m_pLeft; // left child of node
    BinaryTreeNode *m_pRight; // right child of node
};
```

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.  *
3.  */
4. package com.lhp;
5.
6. import java.util.ArrayList;
7. import java.util.List;
8.
9. /**
10. 4.在二元树中找出和为某一值的所有路径
11. 题目：输入一个整数和一棵二元树。
12. 从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。
13. 打印出和与输入整数相等的所有路径。
14. 例如输入整数 22 和如下二元树
15. 10
16. /\
17. 5 12
18. ^
19. 4 7
20. 则打印出两条路径：10, 12 和 10, 5, 7。
21. 二元树节点的数据结构定义为：
```

```
22.     struct BinaryTreeNode // a node in the binary tree
23.     {
24.         int m_nValue; // value of node
25.         BinaryTreeNode *m_pLeft; // left child of node
26.         BinaryTreeNode *m_pRight; // right child of node
27.     };
28. */
29.
30. /**
31. * 二叉树
32. */
33. class BinaryTree {
34.     private BinaryTreeNode root; // 根
35.
36.     public BinaryTreeNode getRoot() {
37.         return root;
38.     }
39.
40.     public void setRoot(BinaryTreeNode root) {
41.         this.root = root;
42.     }
43.
44. /**
45. * 增加子节点
46. * @param 节点
47. */
48.     public synchronized void addNode(BinaryTreeNode node) {
49.         if (null == this.root) {
50.             this.root = node;
51.             return;
52.         }
53.
54.         BinaryTreeNode tempNode = this.root;
55.
56.         while (true) {
```

```
57.         if (node.getM_nValue() > tempNode.getM_nValue()) { // 大于父节点
58.             if (null == tempNode.getM_pRight()) {
59.                 tempNode.setM_pRight(node);
60.                 return;
61.             } else {
62.                 tempNode = tempNode.getM_pRight();
63.                 continue;
64.             }
65.         } else if (node.getM_nValue() < tempNode.getM_nValue()) { // 小于父节点
66.             if (null == tempNode.getM_pLeft()) {
67.                 tempNode.setM_pLeft(node);
68.                 return;
69.             } else {
70.                 tempNode = tempNode.getM_pLeft();
71.                 continue;
72.             }
73.         } else { // 等于父节点
74.             return;
75.         }
76.     }
77. }
78.
79. /**
80. * 输出指定路径和大小的所有路径
81. * @param 路径的和
82. */
83. public synchronized void printSumPath(int sumValue) {
84.     printSumPath(this.root, new ArrayList<Integer>(), 0, sumValue);
85. }
86.
87. /**
88. * @param 节点
89. * @param 路径存储集合
90. * @param 临时路径的和
91. * @param 路径的和
```

```
92.      */
93.  private void printSumPath(BinaryTreeNode node, List<Integer> path, int tempSum, int sumValue) {

94.      if (null == node) {
95.          return;
96.      }
97.
98.      tempSum += node.getM_nValue();
99.      path.add(node.getM_nValue());
100.
101.     boolean isLeaf = (null == node.getM_pLeft() && null == node.getM_pRight()); // 是否为叶子
102.
103.     if (isLeaf && tempSum == sumValue) { // 满足
104.         System.out.print("sumPath(" + sumValue + "): ");
105.         for (int i : path) {
106.             System.out.print(i + " ");
107.         }
108.         System.out.println();
109.     }
110.
111.     // 《向左走，向右走》 :-)
112.     printSumPath(node.getM_pLeft(), path, tempSum, sumValue);
113.     printSumPath(node.getM_pRight(), path, tempSum, sumValue);
114.
115.     // 保证递归完成后返回父节点时路径是根结点到父节点的路径，之后遍历父节点的其他子节
点，没有则返回到爷爷节点...
116.     path.remove(path.size() - 1); // 删除当前节点
117.     // 最后补充一下，如果 path 不是指针而是基本类型的话，这句话就没用了（放在递归调用下
面就没用了），算法也废了，比如在这里加入一句 tempSum+=XXX;对结果没有任何影响，不会影
响递归 return 时其他函数里的参数
118. }
119.
120. /**
121. * 打印前序遍历
122. */
```

```
123. public synchronized void print() {
124.     if (null == this.root) {
125.         System.out.print("HashCode: " + this.hashCode() + "; 空树;");
126.         return;
127.     }
128.     System.out.print("HashCode: " + this.hashCode() + "; 树: ");
129.     print(this.root);
130.     System.out.println();
131. }
132.
133. private void print(BinaryTreeNode node) {
134.     if (null != node) {
135.         System.out.print(node.getM_nValue() + " ");
136.         print(node.getM_pLeft());
137.         print(node.getM_pRight());
138.     }
139. }
140. }
141.
142. /**
143. * 节点
144. */
145. class BinaryTreeNode {
146.     private int m_nValue; // value of node
147.     private BinaryTreeNode m_pLeft; // left child of node
148.     private BinaryTreeNode m_pRight; // right child of node
149.
150.     BinaryTreeNode(int value) {
151.         this.m_nValue = value;
152.     }
153.
154.     public int getM_nValue() {
155.         return m_nValue;
156.     }
157.     public void setM_nValue(int mNValue) {
```

```
158.     m_nValue = mNValue;
159. }
160. public BinaryTreeNode getM_pLeft() {
161.     return m_pLeft;
162. }
163. public void setM_pLeft(BinaryTreeNode mPLeft) {
164.     m_pLeft = mPLeft;
165. }
166. public BinaryTreeNode getM_pRight() {
167.     return m_pRight;
168. }
169. public void setM_pRight(BinaryTreeNode mPRight) {
170.     m_pRight = mPRight;
171. }
172. }
173.
174. public class Four {
175.     public static void main(String[] args) {
176.         BinaryTree tree = new BinaryTree();
177.         tree.addNode(new BinaryTreeNode(10));
178.         tree.addNode(new BinaryTreeNode(5));
179.         tree.addNode(new BinaryTreeNode(12));
180.         tree.addNode(new BinaryTreeNode(4));
181.         tree.addNode(new BinaryTreeNode(7));
182.         tree.addNode(new BinaryTreeNode(9));
183.         tree.addNode(new BinaryTreeNode(3));
184.         tree.print();
185.         tree.printSumPath(22);
186.         tree.printSumPath(31);
187.     }
188. }
```

1.2.6. Top K 算法详细解析---百度面试

问题描述:

这是在网上找到的一道百度的面试题：

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。假设目前有一千万个记录，这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

问题解析:

【分析】：要统计最热门查询，首先就是要统计每个 Query 出现的次数，然后根据统计结果，找出 Top 10。所以我们可以基于这个思路分两步来设计该算法。下面分别给出这两步的算法：

第一步：Query 统计

算法一：直接排序法

首先我们能想到的算法就是排序了，首先对这个日志里面的所有 Query 都进行排序，然后再遍历排好序的 Query，统计每个 Query 出现的次数了。但是题目中有明确要求，那就是内存不能超过 1G，一千万条记录，每条记录是 225Byte，很显然要占据 2.55G 内存，这个条件就不满足要求了。

让我们回忆一下数据结构课程上的内容，当数据量比较大而且内存无法装下的时候，我们可以采用外排序的方法来进行排序，这里笔者采用归并排序，是因为归并排序有一个比较好的时间复杂度 $O(N \lg N)$ 。

排完序之后我们再对已经有序的 Query 文件进行遍历，统计每个 Query 出现的次数，再次写入文件中。

综合分析一下，排序的时间复杂度是 $O(N \lg N)$ ，而遍历的时间复杂度是 $O(N)$ ，因此该算法的总体时间复杂度就是 $O(N \lg N)$ 。

算法二：Hash Table 法

在上个方法中，我们采用了排序的办法来统计每个 Query 出现的次数，时间复杂度是 $N \lg N$ ，那么能不能有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，虽然有一千万个 Query，但是由于重复度比较高，因此事实上只有 300 万的 Query，每个 Query 255Byte，因此我们可以考虑 把他们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，Hash Table 绝对是我们优先的选择，因为 Hash Table 的查询速度非常的快，几乎是 $O(1)$ 的时间复杂度。

那么，我们的算法就有了：维护一个 Key 为 Query 字串，Value 为该 Query 出现次数的

`HashTable`, 每次读取一个 `Query`, 如果该字串 不在 `Table` 中, 那么加入该字串, 并且将 `Value` 值设为 1; 如果该字串在 `Table` 中, 那么将该字串的计数加一即可。最终我们在 $O(N)$ 的时间复杂度 内完成了对该海量数据的处理。

本方法相比算法一: 在时间复杂度上提高了一个数量级, 但不仅仅是时间复杂度上的优化, 该方法只需要 IO 数据文件一次, 而算法一的 IO 次数较多的, 因此该算法比算法一在工程上有更好的可操作性。

第二步: 找出 Top 10

算法一: 排序

我想对于排序算法大家都已经不陌生了, 这里不在赘述, 我们要注意的是排序算法的时间复杂度是 $N \lg N$, 在本题目中, 三百万条记录, 用 1G 内存是可以存下的。

算法二: 部分排序

题目要求是求出 Top 10, 因此我们没有必要对所有的 `Query` 都进行排序, 我们只需要维护一个 10 个大小的数组, 初始化放入 10 `Query`, 按照每个 `Query` 的统计次数由 大到小排序, 然后遍历这 300 万条记录, 每读一条记录就和数组最后一个 `Query` 对比, 如果小于这个 `Query`, 那么继续遍历, 否则, 将数组中最后一条数 据淘汰, 加入当前的 `Query`。最后当所有的数据都遍历完毕之后, 那么这个数组中的 10 个 `Query` 便是我们要找的 Top10 了。

不难分析出, 这样的算法的时间复杂度是 $N * K$, 其中 K 是指 top 多少。

算法三: 堆

在算法二中, 我们已经将时间复杂度由 $N \lg N$ 优化到 NK , 不得不说这是一个比较大的改进了, 可是有没有更好的办法呢?

分析一下, 在算法二中, 每次比较完成之后, 需要的操作复杂度都是 K , 因为要把元素插入到一个线性表之中, 而且采用的是顺序比较。这里我们注意一下, 该数组 是有序的, 一次我们每次查找的时候可以采用二分的方法查找, 这样操作的复杂度就降到了 $\log K$, 可是, 随之而来的问题就是数据移动, 因为移动数据次数增多 了。不过, 这个算法还是比算法二有了改进。

基于以上的分析, 我们想想, 有没有一种既能快速查找, 又能快速移动元素的数据结构呢? 答案是肯定的, 那就是堆。

借助堆结构, 我们可以在 \log 量级的时间内查找和调整/移动。因此到这里, 我们的算法可以改进为这样, 维护一个 K (该题目中是 10)大小的小根堆, 然后遍历 300 万的 `Query`, 分别和根元素进行对比。。。

那么这样, 这个算法发时间复杂度就降到了 $N \log K$, 和算法而相比, 又有了比较大的改进。

1.2.7. 翻转句子中单词的顺序

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。

例如输入“`I am a student.`”，则输出“`student. a am I`”。

分析：由于编写字符串相关代码能够反映程序员的编程能力和编程习惯，与字符串相关的问题一直是程序员笔试、面试题的热门题目。本题也曾多次受到包括微软在内的大量公司的青睐。

由于本题需要翻转句子，我们先颠倒句子中的所有字符。这时，不但翻转了句子中单词的顺序，而且单词内字符也被翻转了。我们再颠倒每个单词内的字符。由于单词内的字符被翻转两次，因此顺序仍然和输入时的顺序保持一致。

还是以上面的输入为例子。翻转“`I am a student.`”中所有字符得到“`.tneduts a ma I`”，再翻转每个单词中字符的顺序得到“`students. a am I`”，正是符合要求的输出。

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

void revert(char* str, int begin, int end)
{
    int len = end - begin + 1;
    int i;
    char tmp;
    for(i=0;i<len/2;i++)
    {
        tmp = str[begin+i];
        str[begin+i] = str[begin+len-i-1];
        str[begin+len-i-1] = tmp;
    }
}
```

```
char* str_process(char* str)

{
    int len = strlen(str);
    revert(str,0,len-1);
    printf("after revert the str is %s\n",str);
    //char* tmp = str;

    int start = 0;
    int end = 0;

    while(str[end]!='\0')
    {
        system("PAUSE");
        printf("start is %c, end is %c\n", str[start],str[end]);
        if(str[start] == ' ')
        {
            start++;
            end++;
            continue;
        }
        else if(str[end+1] == ' ')
        {
            revert(str, start, end);
            end++;
            start = end;
        }
        else if( str[end + 1] == '\0')
        {
            revert(str, start, end);
            end++;
        }
    }
}
```

```

    }

else

end++;

}

return str;

}

int main(int argc, char *argv[])
{
char* str = (char*)malloc(MAX);

sprintf(str, "%s", "We love china");

printf("str is %s\n",str);

printf("after process the str is %s\n",str_process(str));

system("PAUSE");

return 0;
}

```

1.2.8. 判断整数序列是不是二元查找树的后序遍历结果

题目：输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果。如果是返回 true，否则返回 false。例如输入 5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

```

8
/ \
6 10
/ \ \
5 7 9 11

```

因此返回 true。

如果输入 7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回 false。

分析：这是一道 trilogy 的笔试题，主要考查对二元查找树的理解。

在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是树的右子树。根据这样的划分，

把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。

参考代码：

[cpp] [view](#) [plain](#) [copy](#) [print?](#)

```
1.  using namespace std;
2.  ///////////////////////////////////////////////////
3.  // Verify whether a sequence of integers are the post order traversal
4.  // of a binary search tree (BST)
5.  // Input: sequence - the sequence of integers
6.  //         length - the length of sequence
7.  // Return: return true if the sequence is traversal result of a BST,
8.  //         otherwise, return false
9.  ///////////////////////////////////////////////////
10. bool verifySequenceOfBST(int sequence[], int length)
11. {
12.     if(sequence == NULL || length <= 0)
13.         return false;
14.     // root of a BST is at the end of post order traversal sequence
15.     int root = sequence[length - 1];
16.     // the nodes in left sub-tree are less than the root
17.     int i = 0;
18.     for(; i < length - 1; ++ i)
19.     {
20.         if(sequence[i] > root)
21.             break;
22.     }
23.
24.     // the nodes in the right sub-tree are greater than the root
25.     int j = i;
26.     for(; j < length - 1; ++ j)
27.     {
28.         if(sequence[j] < root)
29.             return false;
30.     }
31.
32.     // verify whether the left sub-tree is a BST
```

```

33.     bool left = true;
34.     if(i > 0)
35.         left = verifySquenceOfBST(squence, i);
36.
37.     // verify whether the right sub-tree is a BST
38.     bool right = true;
39.     if(i < length - 1)
40.         right = verifySquenceOfBST(squence + i, length - i - 1);
41.
42.     return (left && right);

```

1.2.9. 查找最小的 K 个元素-使用最大堆

```

1. import java.util.Arrays;
2. import java.util.Random;
3.
4.
5. public class MinKElement {
6.
7.     /**
8.      * 5.最小的 K 个元素
9.      * I would like to use MaxHeap.
10.     * using QuickSort is also OK
11.     */
12.    public static void main(String[] args) {
13.        MinKElement mke=new MinKElement();
14.        int[] a={ 1,2,3,4,5,6,7,8 };
15.        int k=4;
16.        mke.disArrange(a);
17.        System.out.println("after disarranging,the array a["+Arrays.toString(a)+");
18.        mke.findKMin(a,k);
19.
20.    }
21.
22.    //rearrange the array just for test. 随机的调换数组
23.    public void disArrange(int[] a){

```

```

24.     for(int i=0,len=a.length;i<len;i++){
25.         Random random=new Random();
26.         int j=random.nextInt(len);
27.         swap(a,i,j);
28.     }
29. }
30. public void findKMin(int[] a,int k){
31.     int[] heap=a;//you can do this:int[] heap=new int[k].but that maybe space-cost
32.
33.     //create MaxHeap of K elements.from the lastRowIndex to 0. 先建立 K 的最大堆， K 个
   值，那么起始点为 K/2-1
34.     int rootIndex=k/2-1;
35.     while(rootIndex>=0){
36.         reheap(heap,rootIndex,k-1);
37.         rootIndex--;
38.     }
39.     for(int i=k,len=heap.length;i<len;i++){ //建立好 K 的最大堆之后，循环，将 length-k 之
   外的值不断的和最大值进行对比，小于最大值，就纳入堆中，调整堆的最大堆
40.         if(heap[i]<heap[0]){
41.             heap[0]=heap[i];
42.             reheap(heap,0,k-1);
43.         }
44.     }
45.     System.out.print("the K min elements=");
46.     for(int i=0;i<k;i++){
47.         System.out.print(heap[i]+",");
48.     }
49. }
50.
51. //reheap:from root to lastIndex.
52. public void reheap(int[] heap,int rootIndex,int lastIndex){
53.     int orphan=heap[rootIndex];
54.     boolean done=false;
55.     int leftIndex=rootIndex*2+1;
56.     while(!done&&leftIndex<=lastIndex){

```

```

57.     int largerIndex=leftIndex;
58.     if(leftIndex+1<=lastIndex){
59.         int rightIndex=leftIndex+1;
60.         if(heap[rightIndex]>heap[leftIndex]){
61.             largerIndex=rightIndex;
62.         }
63.     }
64.     //Attention! should not use -->heap[root]<heap[largerIndex]<--.
65.     //I spend time to find the problem....
66.     if(orphan<heap[largerIndex]){
67.         heap[rootIndex]=heap[largerIndex];
68.         rootIndex=largerIndex;
69.         leftIndex=rootIndex*2+1;
70.     }else{
71.         done=true;
72.     }
73. }
74. heap[rootIndex]=orphan;
75.
76. }
77. public void swap(int[] a,int i,int j){
78.     int temp=a[i];
79.     a[i]=a[j];
80.     a[j]=temp;
81. }
82. }
```

1.2.10. 求二叉树中节点的最大距离

如果我们把二叉树看成一个图，父子节点之间的连线看成是双向的，我们姑且定义“距离”为两个节点之间边的个数。

写一个程序求一颗二叉树中相距最远的两个节点之间的距离。

分析与解法

我们先画几个不同形状的二叉树，，看看能否得到一些启示。

从例子中可以看出，相距最远的两个节点，一定是两个叶子节点，或者是一个叶子节点到它的根节点。（为什么？）

解法一

根据相距最远的两个节点一定是叶子节点这个规律，我们可以进一步讨论。

对于任意一个节点，以该节点为根，假设这个根有 K 个孩子结点，那么相距最远的两个节点 U 和 V 之间的路径与这个根节点的关系有两种情况：

1. 若路径经过根 $Root$ ，则 U 和 V 是属于不同子树的，且它们都是该子树中道根节点最远的节点，否则跟它们的距离最远相矛盾。
2. 如果路径不经过 $Root$ ，那么它们一定属于根的 K 个子树之一。并且它们也是该子树中相距最远的两个顶点

因此，问题就可以转化为在字数上的解，从而能够利用动态规划来解决。

设第 K 棵子树中相距最远的两个节点： U_k 和 V_k ，其距离定义为 $d(U_k, V_k)$ ，那么节点 U_k 或 V_k 即为子树 K 到根节点 R_k 距离最长的节点。不失一般性，我们设 U_k 为子树 K 中道根节点 R_k 距离最长的节点，其到根节点的距离定义为 $d(U_k, R)$ 。取 $d(U_i, R) (1 \leq i \leq k)$ 中最大的两个值 \max_1 和 \max_2 ，那么经过根节点 R 的最长路径为 $\max_1 + \max_2 + 2$ ，所以树 R 中相距最远的两个点的距离为： $\max\{d(U_1, V_1), \dots, d(U_k, V_k), \max_1 + \max_2 + 2\}$ 。

采用深度优先搜索如图 3-15，只需要遍历所有的节点一次，时间复杂度为 $O(|E|)=O(|V|-1)$ ，其中 V 为点的集合， E 为边的集合。

示例代码如下，我们使用二叉树来实现该算法。

```
//数据结构定义

struct NODE
{
    NODE* pLeft; //左孩子
    NODE* pRight; //右孩子
    int nMaxLeft; //左孩子中的最长距离
    int nMaxRight; //右孩子中的最长距离
    char chValue; //该节点的值
};

int nMaxLen=0;
//寻找树中最长的两段距离

void FindMaxLen(NODE* pRoot)
{
    //遍历到叶子节点，返回
    if(pRoot==NULL)
    {
        return;
    }
```

```
}

//如果左子树为空，那么该节点的左边最长距离为 0

if(pRoot->pLeft==NULL)
{
    pRoot->nMaxLeft=0;
}

//如果右子树为空，那么该节点的右边最长距离为 0

if(pRoot->pRight==NULL)
{
    pRoot->nMaxRight=0;
}

//如果左子树不为空，递归寻找左子树最长距离

if(pRoot->pLeft!=NULL)
{
    FindMaxLen(pRoot->pLeft);
}

//如果右子树不为空，递归寻找右子树最长距离

if(pRoot->pRight!=NULL)
{
    FindMaxLen(pRoot->pRight);
}

if(pRoot->pLeft!=NULL)
{
    int nTempMax=0;

    if(pRoot->pLeft->nMaxLeft > pRoot->pLeft->nMaxRight)
        nTempMax=pRoot->pLeft->nMaxLeft;
    else
        nTempMax=pRoot->pLeft->nMaxRight;

    pRoot->nMaxLeft=nTempMax+1;
}

//计算右子树最长节点距离

if(pRoot->pRight!=NULL)
{
    int nTempMax=0;

    if(pRoot->pRight->nMaxLeft > pRoot->pRight->nMaxRight)
```

```

nTempMax= pRoot->pRight->nMaxLeft;
else
    nTempMax= pRoot->pRight-> nMaxRight;
    pRoot->nMaxRight=nTempMax+1;
}
//更新最长距离
if(pRoot->nMaxLeft+pRoot->nMaxRight > nMaxLen)
    nMaxLen=pRoot->nMaxLeft+pRoot->nMaxRight;
}

```

1.3. 面试题集合（二）

1.3.1. 求 $1+2+\dots+n$

要求不能使用乘除法、`for`、`while`、`if`、`else`、`switch`、`case` 等关键字
以及条件判断语句 (`A?B:C`)。

思路一（转）

循环只是让相同的代码执行 n 遍而已，我们完全可以不用 `for` 和 `while` 达到这个效果。
比如定义一个类，我们 `new` 一含有 n 个这种类型元素的数组，
那么该类的构造函数将确定会被调用 n 次。我们可以将需要执行的代码放到构造函数里。

```

#include <iostream.h>
class Temp
{
public:
Temp()
{
    ++N;
    Sum += N;
}
static void Reset() { N = 0; Sum = 0; }
static int GetSum() { return Sum; }
private:

```

```

static int N;
static int Sum;
};

int Temp::N = 0;
int Temp::Sum = 0;
int solution1_Sum(int n)
{
    Temp::Reset();
    Temp *a = new Temp[n]; //就是这个意思， new 出 n 个数组。
    delete []a;
    a = 0;
    return Temp::GetSum();
}

int main()
{
    cout<<solution1_Sum(100)<<endl;
    return 0;
}

//运行结果:
//5050
//Press any key to continue
//第二种思路: (转)

-----

```

既然不能判断是不是应该终止递归，我们不妨定义两个函数。
一个函数充当递归函数的角色，另一个函数处理终止递归的情况，
我们需要做的就是在两个函数里二选一。
从二选一我们很自然的想到布尔变量，
比如 ture/(1)的时候调用第一个函数， false/(0)的时候调用第二个函数。
那现在的问题是如何把数值变量 n 转换成布尔值。
如果对 n 连续做两次反运算，即!!n，那么非零的 n 转换为 true，0 转换为 false。

```

#include <iostream.h>
class A;
A* Array[2];
class A
{

```

```

public:
virtual int Sum (int n) { return 0; }
};

class B: public A
{
public:
virtual int Sum (int n) { return Array[!n]->Sum(n-1)+n; }
};

int solution2_Sum(int n)
{
A a;
B b;
Array[0] = &a;
Array[1] = &b;
int value = Array[1]->Sum(n);
//利用虚函数的特性，当 Array[1]为 0 时，即 Array[0] = &a; 执行 A::Sum,
//当 Array[1]不为 0 时，即 Array[1] = &b; 执行 B::Sum。
return value;
}

int main()
{
cout<<solution2_Sum(100)<<endl;
return 0;
}

运行结果
//5050
//Press any key to continue
这种方法是用虚函数来实现函数的选择。当 n 不为零时，执行函数 B::Sum；当 n 为 0 时，执行 A::Sum。我们也可以直接用函数指针数组，这样可能还更直接一些：
typedef int (*fun)(int);
int solution3_f1(int i)
{
return 0;
}
int solution3_f2(int i)

```

```
{  
fun f[2]={solution3_f1, solution3_f2};  
return i+f[!i](i-1);  
}
```

第三种，我的思路：

由上面第二种思路的启发，运用 `&&` 短路的特性，用以设置递归出口

```
#include<iostream>  
using namespace std;  
int mySum(int n)  
{  
int temp=0;  
(!n) && (temp=mySum(n-1));//n=0 时，第一个条件为 false， (temp=mySum(n-1)) 不会执行，  
因此不会再递归调用函数，递归结束。  
return temp+n;  
}  
int main()  
{  
int sum=mySum(100);  
cout<<sum<<endl;  
return 0;  
}
```

运行结果

```
//5050  
//Press any key to continue
```

第四种思路（转）

让编译器帮我们来完成类似于递归的运算，比如如下代码：

```
template <int n> struct solution4_Sum  
{  
enum Value { N = solution4_Sum<n - 1>::N + n};  
};  
template <> struct solution4_Sum<1>  
{  
enum Value { N = 1};  
};
```

`solution4_Sum<100>::N` 就是 $1+2+\dots+100$ 的结果。当编译器看到 `solution4_Sum<100>` 时，就

是为模板类 solution4_Sum 以参数 100 生成该类型的代码。但以 100 为参数的类型需要得到以 99 为参数的类型，因为 `solution4_Sum<100>::N=solution4_Sum<99>::N+100`。这个过程会递归一直到参数为 1 的类型，由于该类型已经显式定义，编译器无需生成，递归编译到此结束。由于这个过程是在编译过程中完成的，因此要求输入 `n` 必须是在编译期间就能确定，不能动态输入。这是该方法最大的缺点。而且编译器对递归编译代码的递归深度是有限制的，也就是要求 `n` 不能太大。

1.3.2. 输入一个单向链表，输出该链表中倒数第 `k` 个结点

题目：输入一个单向链表，输出该链表中倒数第 `k` 个结点。链表的倒数第 0 个结点为链表的尾指针。链表结点定义如下：

```
struct ListNode
{
    int m_nKey;
    ListNode* m_pNext;
};
```

分析：为了得到倒数第 `k` 个结点，很自然的想法是先走到链表的尾端，再从尾端回溯 `k` 步。可是输入的是单向链表，只有从前往后的指针而没有从后往前的指针。因此我们需要打开我们的思路。

既然不能从尾结点开始遍历这个链表，我们还是把思路回到头结点上来。假设整个链表有 `n` 个结点，那么倒数第 `k` 个结点是从头结点开始的第 `n-k-1` 个结点（从 0 开始计数）。如果我们能够得到链表中结点的个数 `n`，那我们只要从头结点开始往后走 `n-k-1` 步就可以了。如何得到结点数 `n`？这个不难，只需要从头开始遍历链表，每经过一个结点，计数器加一就行了。这种思路的时间复杂度是 $O(n)$ ，但需要遍历链表两次。第一次得到链表中结点个数 `n`，第二次得到从头结点开始的第 `n-k-1` 个结点即倒数第 `k` 个结点。

如果链表的结点数不多，这是一种很好的方法。但如果输入的链表的结点个数很多，有可能不能一次性把整个链表都从硬盘读入物理内存，那么遍历两遍意味着一个结点需要两次从硬盘读入到物理内存。我们知道把数据从硬盘读入到内存是非常耗时间的操作。我们能不能把链表遍历的次数减少到 1？如果可以，将能有效地提高代码执行的时间效率。

如果我们在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第 `k-1` 步之前，第二个指针保持不动；在第 `k-1` 步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在 `k-1`，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第 `k` 个结点。

这种思路只需要遍历链表一次。对于很长的链表，只需要把每个结点从硬盘导入到内存一次。因此这一方法的时间效率前面的方法要高。

思路一的参考代码：

```
////////////////////////////////////////////////////////////////
// Find the kth node from the tail of a list
// Input: pListHead - the head of list
// k - the distance to the tail
// Output: the kth node from the tail of a list
////////////////////////////////////////////////////////////////
ListNode* FindKthToTail_Solution1(ListNode* pListHead, unsigned int k)
```

```

{
if(pListHead == NULL)
return NULL;

// count the nodes number in the list
ListNode *pCur = pListHead;
unsigned int nNum = 0;
while(pCur->m_pNext != NULL)
{
pCur = pCur->m_pNext;
nNum++;
}

// if the number of nodes in the list is less than k
// do nothing
if(nNum < k)
return NULL;

// the kth node from the tail of a list
// is the (n - k)th node from the head
pCur = pListHead;
for(unsigned int i = 0; i < nNum - k; ++ i)
pCur = pCur->m_pNext;

return pCur;
}

```

思路二的参考代码：

```

///////////////////////////////
// Find the kth node from the tail of a list
// Input: pListHead - the head of list
// k - the distance to the tail
// Output: the kth node from the tail of a list
///////////////////////////////

ListNode* FindKthToTail_Solution2(ListNode* pListHead, unsigned int k)
{
if(pListHead == NULL)
return NULL;

ListNode *pAhead = pListHead;
ListNode *pBehind = NULL;

for(unsigned int i = 0; i < k; ++ i)
{
if(pAhead->m_pNext != NULL)

```

```

pAhead = pAhead->m_pNext;
else
{
// if the number of nodes in the list is less than k,
// do nothing
return NULL;
}
}

pBehind = pListHead;

// the distance between pAhead and pBehind is k
// when pAhead arrives at the tail, p
// Behind is at the kth node from the tail
while(pAhead->m_pNext != NULL)
{
pAhead = pAhead->m_pNext;
pBehind = pBehind->m_pNext;
}

return pBehind;
}

```

讨论：这道题的代码有大量的指针操作。在软件开发中，错误的指针操作是大部分问题的根源。因此每个公司都希望程序员在操作指针时有良好的习惯，比如使用指针之前判断是不是空指针。这些都是编程的细节，但如果这些细节把握得不好，很有可能就会和心仪的公司失之交臂。

另外，这两种思路对应的代码都含有循环。含有循环的代码经常出的问题是在循环结束条件的判断。是该用小于还是小于等于？是该用 k 还是该用 k-1？由于题目要求的是从 0 开始计数，而我们的习惯思维是从 1 开始计数，因此首先要想好这些边界条件再开始编写代码，再者要在编写完代码之后再用边界值、边界值减 1、边界值加 1 都运行一次（在纸上写代码就只能在心里运行了）。

扩展：和这道题类似的题目还有：输入一个单向链表。如果该链表的结点数为奇数，输出中间的结点；如果链表结点数为偶数，输出中间两个结点前面的一个。如果各位感兴趣，请自己分析并编写代码。

1.3.3. 输入一个已经按升序排序过的数组和一个数字

题目：输入一个已经按升序排序过的数组和一个数字，

在数组中查找两个数，使得它们的和正好是输入的那个数字。

要求时间复杂度是 $O(n)$ 。如果有对数字的和等于输入的数字，输出任意一对即可。

例如输入数组 1、2、4、7、11、15 和数字 15。由于 $4+11=15$ ，因此输出 4 和 11。

[java] [view plain](#) [copy print?](#)

```
1.  public class TwoElementEqualSum {  
2.  
3.      /**  
4.      * 第 14 题:  
5.      题目: 输入一个已经按升序排序过的数组和一个数字,  
6.      在数组中查找两个数, 使得它们的和正好是输入的那个数字。  
7.      要求时间复杂度是 O(n)。如果有多对数字的和等于输入的数字, 输出任意一对即可。  
8.      例如输入数组 1、2、4、7、11、15 和数字 15。由于 4+11=15, 因此输出 4 和 11。  
9.      */  
10.     public static void main(String[] args) {  
11.  
12.         int[] a={1,2,4,7,11,15};  
13.         find(a,18);  
14.  
15.     }  
16.  
17.     static void find(int[] a,int sum){  
18.         int i=0;  
19.         int j=a.length-1;  
20.         while(i<j){  
21.             if(a[i]+a[j]>sum){  
22.                 j--;  
23.             }else if(a[i]+a[j]<sum){  
24.                 i++;  
25.             }else{  
26.                 System.out.println(a[i]+"+" +a[j]+ "=" +sum);  
27.             return;  
28.         }  
29.     }  
30.     System.out.println("not found");  
31. }  
32. }
```

1.3.4. 输入一颗二元查找树，将该树转换为它的镜像

题目：输入一颗二元查找树，将该树转换为它的镜像，
即在转换后的二元查找树中，左子树的结点都大于右子树的结点。
用递归和循环两种方法完成树的镜像转换。

例如输入：

```
8
//
6 10
///
5 7 9 11
```

输出：

```
8
//
10 6
///
11 9 7 5
```

定义二元查找树的结点为：

```
struct BSTreeNode // a node in the binary search tree (BST)
{
    int m_nValue; // value of node
    BSTreeNode *m_pLeft; // left child of node
    BSTreeNode *m_pRight; // right child of node
};
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. public static void mirrorHelp1(Node node){
2.     if(node==null) return;
3.     swapChild(node);
4.     mirrorHelp1(node.getLeft());
5.     mirrorHelp1(node.getRight());
6. }
7. //use no recursion but stack
```

```
8.     public static void mirrorHelp2(Node node){  
9.         if(node==null) return;  
10.        Stack<Node> stack=new Stack<Node>();  
11.        stack.add(node);  
12.        while(!stack.isEmpty()){  
13.            node=stack.pop();  
14.            swapChild(node);  
15.            if(node.getLeft() != null){  
16.                stack.push(node.getLeft());  
17.            }  
18.            if(node.getRight() != null){  
19.                stack.push(node.getRight());  
20.            }  
21.  
22.        }  
23.    }  
24.  
25.    public static void swapChild(Node node){  
26.        /*not like c/c++,you cannot do this:  
27.        Node temp=left;  
28.        left=right;  
29.        right=temp;  
30.        */  
31.        Node left=node.getLeft();  
32.        node.setLeft(node.getRight());  
33.        node.setRight(left);  
34.    }
```

1.3.5. 输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印

```
*****  
****/
```

/* 题目：输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。

例如输入

```
8
/\ 
6 10
/\ 
5 7 9 11
```

输出 8 6 10 5 7 9 11。

思路：

广度优先遍历

deque(STL)*/

```
/*********************  
****/  
#include <iostream>  
#include <deque>  
using namespace std;  
struct BTree  
{  
    BTree* pLeft;  
    BTree* pRight;  
    int value;  
};  
void InsertBTree(BTree* &pRoot,int n)  
{  
    if(!pRoot)  
    {  
        pRoot=new BTree;  
        pRoot->pLeft=NULL;  
        pRoot->pRight=NULL;  
        pRoot->value=n;  
    }  
    else  
    {  
        if (n>pRoot->value)
```

```

{
InsertBTree(pRoot->pRight,n);
}
if (n<pRoot->value)
{
InsertBTree(pRoot->pLeft,n);
}
if (n==pRoot->value)
{
cout<<"repeated insertion"<<endl;
}
}
}

void VisitByLevel(BTree* pRoot)
{
if (!pRoot)
{
return;
}
deque<BTree*> BTDeque;
BTDeque.push_back(pRoot);
while(BTDeque.size())
{
BTree* p=BTDeque.front();
cout<<p->value<<"\t";
BTDeque.pop_front();
if (p->pLeft)
{
BTDeque.push_back(p->pLeft);
}
if (p->pRight)
{
BTDeque.push_back(p->pRight);
}
}
}

```

```

}

int main()
{
    BTree* pRoot=NULL;
    InsertBTree(pRoot,8);
    InsertBTree(pRoot,6);
    InsertBTree(pRoot,10);
    InsertBTree(pRoot,5);
    InsertBTree(pRoot,7);
    InsertBTree(pRoot,9);
    InsertBTree(pRoot,11);
    VisitByLevel(pRoot);
    return 0;
}

```

1.3.6. 在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b

```

*****
*/
/* 在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b
如果从头遍历，与后面字符进行比较出现次数是否为 1，算法复杂度为 O(n^2)，
考虑以空间换时间，因为一个字符最多两个字节，占 8 位。因此可用一个 256
位数组保存每个字符出现的次数
*/
*****
#include <iostream>
#include <string>
using namespace std;
void FindChar(string &s)
{
    int str[256];
    for (int i=0;i<256;i++)
    {
        str[i]=0;
    }
}
```

```

}

for (int i=0;i<s.size();i++)
{
    str[s[i]]++;
}

for (int i=0;i<256;i++)
{
    if (str[i]==1)
    {
        printf("%c",i);
    }
}
}

int main()
{
    string s;
    cout<<"请输入字符串 "<<endl;
    cin>>s;
    FindChar(s);
    cout<<endl;
    return 0;
}

```

1.3.7. n 个数字 (0,1,...,n-1) 形成一个圆圈

题目：n 个数字 (0,1,...,n-1) 形成一个圆圈，从数字 0 开始，每次从这个圆圈中删除第 m 个数字（第一个为当前数字本身，第二个为当前数字的下一个数字）。当一个数字删除后，从被删除数字的下一个继续删除第 m 个数字。求出在这个圆圈中剩下的最后一个数字。

分析：既然题目有一个数字圆圈，很自然的想法是我们用一个数据结构来模拟这个圆圈。在常用的数据结构中，我们很容易想到用环形列表。我们可以创建一个总共有 m 个数字的环形列表，然后每次从这个列表中删除第 m 个元素。

在参考代码中，我们用 STL 中 std::list 来模拟这个环形列表。由于 list 并不是一个环形的结构，因此每次迭代器扫描到列表末尾的时候，要记得把迭代器移到列表的头部。这样就是按照一个圆圈的顺序来遍历这个列表了。

这种思路需要一个有 n 个结点的环形列表来模拟这个删除的过程，因此内存开销为 $O(n)$ 。而且这种方法每删除一个数字需要 m 步运算，总共有 n 个数字，因此总的时间复杂度是 $O(mn)$ 。当 m 和 n 都很大的时候，这种方法是很慢的。

接下来我们试着从数学上分析出一些规律。首先定义最初的 n 个数字 $(0, 1, \dots, n-1)$ 中最后剩下的数字是关于 n 和 m 的方程为 $f(n, m)$ 。

在这 n 个数字中，第一个被删除的数字是 $m \% n - 1$ ，为简单起见记为 k 。那么删除 k 之后的剩下 $n-1$ 的数字为 $0, 1, \dots, k-1, k+1, \dots, n-1$ ，并且下一个开始计数的数字是 $k+1$ 。相当于在剩下的序列中， $k+1$ 排到最前面，从而形成序列 $k+1, \dots, n-1, 0, \dots, k-1$ 。该序列最后剩下的数字也应该是关于 n 和 m 的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从 0 开始的连续序列），因此该函数不同于前面函数，记为 $f'(n-1, m)$ 。最初序列最后剩下的数字 $f(n, m)$ 一定是剩下序列的最后剩下数字 $f'(n-1, m)$ ，所以 $f(n, m) = f'(n-1, m)$ 。

接下来我们把剩下的这 $n-1$ 个数字的序列 $k+1, \dots, n-1, 0, \dots, k-1$ 作一个映射，映射的结果是形成一个从 0 到 $n-2$ 的序列：

$k+1 \rightarrow 0$

$k+2 \rightarrow 1$

...

$n-1 \rightarrow n-k-2$

$0 \rightarrow n-k-1$

...

$k-1 \rightarrow n-2$

把映射定义为 p ，则 $p(x) = (x - k - 1) \% n$ ，即如果映射前的数字是 x ，则映射后的数字是 $(x - k - 1) \% n$ 。对应的逆映射是 $p^{-1}(x) = (x + k + 1) \% n$ 。

由于映射之后的序列和最初的序列有同样的形式，都是从 0 开始的连续序列，因此仍然可以用函数 f 来表示，记为 $f(n-1, m)$ 。根据我们的映射规则，映射之前的序列最后剩下的数字 $f'(n-1, m) = p^{-1}[f(n-1, m)] = [f(n-1, m) + k + 1] \% n$ 。把 $k = m \% n - 1$ 代入得 到 $f(n, m) = f'(n-1, m) = [f(n-1, m) + m] \% n$ 。

经过上面复杂的分析，我们终于找到一个递归的公式。要得到 n 个数字的序列的最后剩下的数字，只需要得到 $n-1$ 个数字的序列的最后剩下的数字，并可以依此类推。当 $n=1$ 时，也就是序列中开始只有一个数字 0，那么很显然最后剩下的数字就是 0。我们把这种关系表示为：

$0 \ n=1$

$f(n, m) = \{$

$[f(n-1, m) + m] \% n \ n > 1$

尽管得到这个公式的分析过程非常复杂，但它用递归或者循环都很容易实现。最重要的是，这是一种时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 的方法，因此无论在时间上还是空间上都优于前面的思路。

思路一的参考代码：

C++代码   

```
1. /////////////////
2. // n integers (0, 1, ... n - 1) form a circle. Remove the mth from
3. // the circle at every time. Find the last number remaining
4. // Input: n - the number of integers in the circle initially
5. // m - remove the mth number at every time
6. // Output: the last number remaining when the input is valid,
7. // otherwise -1
8. /////////////////
9. int LastRemaining_Solution1(unsigned int n, unsigned int m)
10. {
11.     // invalid input
12.     if(n < 1 || m < 1)
13.         return -1;
14.     unsigned int i = 0;
15.     // initiate a list with n integers (0, 1, ... n - 1)
16.     list<int> integers;
17.     for(i = 0; i < n; ++ i)
18.         integers.push_back(i);
19.     list<int>::iterator curinteger = integers.begin();
20.     while(integers.size() > 1)
21.     {
22.         // find the mth integer. Note that std::list is not a circle
23.         // so we should handle it manually
24.         for(int i = 1; i < m; ++ i)
25.         {
26.             curinteger ++;
```

```
27. if(curinteger == integers.end())
28. curinteger = integers.begin();
29. }
30. // remove the nth integer. Note that std::list is not a circle
31. // so we should handle it manually
32. list<int>::iterator nextinteger = ++ curinteger;
33. if(nextinteger == integers.end())
34. nextinteger = integers.begin();
35. -- curinteger;
36. integers.erase(curinteger);
37. curinteger = nextinteger;
38. }
39. return *(curinteger);
40. }
```

[c++] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /////////////////
2. // n integers (0, 1, ... n - 1) form a circle. Remove the nth from
3. // the circle at every time. Find the last number remaining
4. // Input: n - the number of integers in the circle initially
5. //       m - remove the nth number at every time
6. // Output: the last number remaining when the input is valid,
7. //       otherwise -1
8. ///////////////
9. int LastRemaining_Solution1(unsigned int n, unsigned int m)
10. {
11.     // invalid input
12.     if(n < 1 || m < 1)
13.         return -1;
14.     unsigned int i = 0;
15.     // initiate a list with n integers (0, 1, ... n - 1)
```

```

16.     list<int> integers;
17.     for(i = 0; i < n; ++ i)
18.         integers.push_back(i);
19.     list<int>::iterator curinteger = integers.begin();
20.     while(integers.size() > 1)
21.     {
22.         // find the mth integer. Note that std::list is not a circle
23.         // so we should handle it manually
24.         for(int i = 1; i < m; ++ i)
25.         {
26.             curinteger++;
27.             if(curiinteger == integers.end())
28.                 curinteger = integers.begin();
29.         }
30.
31.         // remove the mth integer. Note that std::list is not a circle
32.         // so we should handle it manually
33.         list<int>::iterator nextinteger = ++ curinteger;
34.         if(nextinteger == integers.end())
35.             nextinteger = integers.begin();
36.         -- curinteger;
37.         integers.erase(curiinteger);
38.         curinteger = nextinteger;
39.     }
40.
41.     return *(curinteger);
42. }
```

思路二的参考代码：

C++代码   

1. [cpp] view plaincopy

1. ////////////////

2. // n integers (0, 1, ... n - 1) form a circle. Remove the mth from

```

3. // the circle at every time. Find the last number remaining
4. // Input: n - the number of integers in the circle initially
5. // m - remove the mth number at every time
6. // Output: the last number remaining when the input is valid,
7. // otherwise -1
8. /////////////////////////////////
9. int LastRemaining_Solution2(int n, unsigned int m)
10. {
11. // invalid input
12. if(n <= 0 || m < 0)
13. return -1;
14. // if there are only one integer in the circle initially,
15. // of course the last remaining one is 0
16. int lastinteger = 0;
17. // find the last remaining one in the circle with n integers
18. for (int i = 2; i <= n; i++)
19. lastinteger = (lastinteger + m) % i;
20. return lastinteger;
21. }

```

1.3.8. 定义 Fibonacci 数列

题目：定义 Fibonacci 数列如下：

/ 0 n=0

f(n)= 1 n=1

/ f(n-1)+f(n-2) n=2

输入 n，用最快的方法求该数列的第 n 项。

分析：在很多 C 语言教科书中讲到递归函数的时候，都会用 Fibonacci 作为例子。

```
*****  
****/
```

/* 定义 Fibonacci 数列如下：

/ 0 n=0

f(n) = 1 n=1

\ f(n-1)+f(n-2) n=2

输入 n, 用最快的方法求该数列的第 n 项。

递归方法重复计算了很多项，通过定义两个额外空间保存前两项的值，可以空间换取时间。

复杂度为 O(N)同时看到有人用数学方法进行分析处理，提出了一个 O(logN)的算法，

<http://blog.csdn.net/jxy859/article/details/6685700> 有兴趣的可以看下 */

```
*****  
****/
```

```
#include <iostream>
```

```
using namespace std;
```

```
int Fibonacci1(int n)//递归
```

```
{
```

```
if (n==0||n==1)
```

```
{
```

```
return n;
```

```
}
```

```
else
```

```
{
```

```
return Fibonacci1(n-1)+Fibonacci1(n-2);
```

```
}
```

```
}
```

```
int Fibonacci2(int n)//额外空间
```

```
{
```

```
int result=0;
```

```
int a[2]={0,1};
```

```
if (n==0||n==1)
```

```
{
```

```
return n;
```

```
}
```

```
else
{
for (int i=1;i<n;i++)
{
result=a[0]+a[1];
a[0]=a[1];
a[1]=result;
}
}

return result;
}

int main()
{
cout<<Fibonacci2(5)<<endl;
cout<<Fibonacci1(5);
return 0;
}
```

1.3.9. 左移递减数列查找某一个数

微软（运算）：

一个数组是由一个递减数列左移若干位形成的，比如{4, 3, 2, 1, 6, 5}
是由{6, 5, 4, 3, 2, 1}左移两位形成的，在这种数组中

[csharp] [view](#) [plain](#) [copy](#) [print](#)?

1.

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include<iostream>
2. #include<cassert>
3. #include<stack>
4. using namespace std ;
5. int FindNumberInLeftShiftSequence(int *A,int nLen,int expectedNum)
6. {
7.     assert(A!=NULL&&nLen>0);
```

```
8.     int start=0;
9.     int end=nLen-1;
10.    while(start<=end)
11.    {
12.        int mid=start+((end-start)>>2);
13.        if(expectedNum==A[mid])
14.            return mid;
15.        if (A[mid]<A[start])
16.        {
17.            if(expectedNum>A[mid])
18.                end=mid-1;
19.            else
20.                start=mid+1;
21.        }
22.        else if(A[mid]>A[start])
23.        {
24.            if(expectedNum<A[mid])
25.                start=mid+1;
26.            else
27.                end=mid-1;
28.        }
29.        else
30.        {
31.            for (int i=start;i<mid;i++)
32.            {
33.                if(A[i]==expectedNum)
34.                    return i;
35.            }
36.            start=mid+1;
37.        }
38.    }
39.    return -1;
40. }
41. int main()
42. {
```

```
43. int A[]={6,5,4,3,2,1};  
44. int nLen=sizeof(A)/sizeof(int);  
45. cout<<FindNumberInLeftShiftSequence(A,nLen,0)<<endl;  
46. int B[]={1,1,1,1,0,1};  
47. int nLen2=sizeof(B)/sizeof(int);  
48. cout<<FindNumberInLeftShiftSequence(B,nLen2,0);  
49. return 1;  
50. }
```

思路：在此序列不断二分的过程中，由于原序列是一个递减序列经过旋转得到的，将它从任何位置分开，都会得到两个序列，其中一个是递减序列，另一个可以通过一个递减序列通过旋转得到。这样在不断地二分查找时，我们处理的序列子片段要么就是一个旋转后递减序列，要么就是一个纯递减序列，而无论是前者还是后者，在继续分成两个片段时，至少有一个纯递减序列（可能两个都是，如果之前的序列片段就是纯递减序列的话）。这样我们可以保证能找到一个片段是纯递减序列（if(data[i]>=data[j])），然后判断我们要找的数是否在这个片段中（这是很直观的，if(data[i]<=num&&num<=data[j]))，如果在则继续在此片段中查找，否则说明在另一个序列中，则递归在其中查找

代码：

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include<iostream>  
2. using namespace std;  
3. int bisearch(int a[],int left,int right,int num)  
4. {  
5.     if(a==NULL||right<0)  
6.         return -1;  
7.  
8.     if(left==right)  
9.     {  
10.         if(a[left]==num)  
11.             return left;  
12.         else  
13.             return -1;  
14.     }
```

```
15.  
16. int mid=(left+right)/2;  
17. if(a[mid]==num)  
18.     return mid;  
19. if(a[mid]<=a[left])  
20. {  
21.     if(num>a[mid]&&num<=a[left])  
22.         return bisearch(a,left,mid-1,num);  
23.     else  
24.         return bisearch(a,mid+1,right,num);  
25. }  
26. else  
27. {  
28.     if(num>=a[right]&&num<a[mid])  
29.         return bisearch(a,mid+1,right,num);  
30.     else  
31.         return bisearch(a,left,mid-1,num);  
32. }  
33. }  
34. int main()  
35. {  
36.     int a[100]={4,3,2,1,6,5};  
37.     cout<<bisearch(a,0,5,3)<<endl;  
38.     return 0;  
39. }
```

1.3.10. 对于一个整数矩阵，存在一种运算，对矩阵中任意元素加一时，需要其相邻（上下左右）某一个元素也加一

1. 对于一个整数矩阵，存在一种运算，对矩阵中任意元素加一时，需要其相邻（上下左右）

某一个元素也加一，

现给出一正数矩阵，判断其是否能够由一个全零矩阵经过上述运算得到。

2. 一个整数数组，长度为 n ，将其分为 m 份，使各份的和相等，求 m 的最大值

比如{3, 2, 4, 3, 6} 可以分成{3, 2, 4, 3, 6} $m=1$;

{3,6}{2,4,3} $m=2$

{3,3}{2,4}{6} $m=3$ 所以 m 的最大值为 3

一个整数数组，长度为 n ，将其分为 m 份，使各份的和相等，求 m 的最大值

比如{3, 2, 4, 3, 6} 可以分成{3, 2, 4, 3, 6} $m=1$;

{3,6}{2,4,3} $m=2$

{3,3}{2,4}{6} $m=3$ 所以 m 的最大值为 3

算法 原理的思想是将大问题转换成小问题。

就{3, 2, 4, 3, 6}的操作步骤:

第一步: 想将数组递减排序得{6, 4, 3, 3, 2}，求出数组中所有数的和 $m=18$,第一个最大的数 $b=6$ ， $m/b=3$ 余数为 0,

当除数为 1，余数为 0 时终止。当余数不为 0 时，转到第三步。当余数为 0 时将数组划分为{6}, {4, 3, 3, 2}两个。把{4, 3, 3, 2}看成一个新的数组。

第二步: 先用{4, 3, 3, 2}中的最大数与 $b=6$ 比较，即 $4 < b$ ，所以再将 4 与最右边的数即 2 相加与 b 比较，

结果相等，则将这两个数从该数组中除去生成新的数组，转到第一步，现在的结果是{6},{4,2},{3,3},把{3,3}看成一个新的数组继续重复第二步。

第三步，将数组中最大的数与最小的数取出构成一个新数组 Z，剩余的构成一个数组，然后，

判断 m/Z 中数字之和看是否余数为 0，若为 0，把 b 替换为 Z 中数字之和转第二步，若不为 0，

继续从剩余的数字中取出最小值加入到 Z 中，再判断 m/Z 中数字之和看是否余数为 0，直到为 0，转第二步为止。

最后得到的结果是{6},{4,2},{3,3} 这时可以计算出 m 为 3，也可以在程序中作记载。

在第二步工程过，若出现两个数相加大于上一次的 b ，则将程序转到第三步。

[cpp] [view plain](#) [copy](#) [print](#)

1. // p44_YahooReviewerQ.cpp : Defines the entry point for the console application.
2. //
3. //-----
4. /*
5. 1.对于一个整数矩阵，存在一种运算，对矩阵中任意元素加一时，需要其相邻（上下左右）
6. 某一个元素也加一，现给出一正数矩阵，判断其是否能够由一个全零矩阵经过上述运算得到。

```
7.  */
8.  /*
9.      2.一个整数数组，长度为 n，将其分为 m 份，使各份的和相等，求 m 的最大值
10.     比如{3, 2, 4, 3, 6} 可以分成{3, 2, 4, 3, 6} m=1;
11.     {3,6}{2,4,3} m=2
12.     {3,3}{2,4}{6} m=3 所以 m 的最大值为 3
13. */
14. //-----
15. #include "stdafx.h"
16. #include <stdlib.h>
17. #include <stdio.h>
18. #include <conio.h>
19. #include <iostream>
20. #include <stack>
21. #include <math.h>
22.
23. using namespace std;
24.
25. #define INIT_SIZE 100
26.
27. #define Q_MODE_ 1 // 若为 1： 表示第一题； 若为 2： 表示为第二题；
28.
29. int nMxtric[INIT_SIZE][INIT_SIZE];
30. int nXLen = 0;
31. int nYLen = 0;
32.
33. int nArray[INIT_SIZE];
34. int nACurLen = 0;
35. // 判断是否可以由零矩阵转换而得
36. //-----
37. // 注： 算法不知，只知道每次增加 5，即矩阵和为 5 的
38. //    倍数。
39. //-----
40. // 判读改点是 模板否?...
```

```

41. bool testPosIsTemplate(int nM axtric[INIT_SIZE][INIT_SIZE],int nXLen,int nYLen,int nXPos,int nYPos)
42. {
43.     bool bIs = false;
44.     int nUp_x = 0,nDown_x = 0,nLeft_x = 0,nRight_x = 0;
45.     int nUp_y = 0,nDown_y = 0,nLeft_y = 0,nRight_y = 0;
46.
47.     nUp_x = nXPos-1;  nUp_y = nYPos;
48.     nDown_x = nXPos+1;nDown_y = nYPos;
49.     nLeft_x = nXPos;nLeft_y = nYPos-1;
50.     nRight_x = nXPos;nRight_y = nYPos+1;
51.
52.     if(nUp_x <= 0)nUp_x = 0;
53.     if(nUp_x >= nXLen)nUp_x = nXLen - 1;
54.     if(nDown_x <= 0)nDown_x = 0;
55.     if(nDown_x >= nXLen)nDown_x = nXLen - 1;
56.     if(nLeft_x <= 0)nLeft_x = 0;
57.     if(nLeft_x >= nXLen)nLeft_x = nXLen - 1;
58.     if(nRight_x <= 0)nRight_x = 0;
59.     if(nRight_x >= nXLen)nRight_x = nXLen - 1;
60.
61.     if(nUp_y <= 0)nUp_y = 0;
62.     if(nUp_y >= nYLen)nUp_y = nYLen - 1;
63.     if(nDown_y <= 0)nDown_y = 0;
64.     if(nDown_y >= nYLen)nDown_y = nYLen - 1;
65.     if(nLeft_y <= 0)nLeft_y = 0;
66.     if(nLeft_y >= nYLen)nLeft_y = nYLen - 1;
67.     if(nRight_y <= 0)nRight_y = 0;
68.     if(nRight_y >= nYLen)nRight_y = nYLen - 1;
69.
70.     if(nM axtric[nXPos][nYPos] && nM axtric[nUp_x][nUp_y] && nM axtric[nDown_x][nDown_y] &&
71.         nM axtric[nLeft_x][nLeft_y] && nM axtric[nRight_x][nRight_y])
72.         bIs = true;
73.
74.     return bIs;

```

```

75. }
76. // 判断是否为零矩阵
77. bool isZeroM axtric(int nM axtric[INIT_SIZE][INIT_SIZE],int nXLen,int nYLen)
78. {
79.     bool bIsZ = true;
80.
81.     for(int i = 0;i < nXLen;i++)
82.         for(int j = 0;j < nYLen;j++)
83.         {
84.             if(nM axtric[i][j])
85.             {
86.                 bIsZ = false;
87.                 goto END;
88.             }
89.         }
90. END:
91.     return bIsZ;
92.
93. }
94. // 减去模板矩阵
95. bool DecTemplateM axtric(int nM axtric[INIT_SIZE][INIT_SIZE],int nXLen,int nYLen,int nXPos,int nY
   Pos)
96. {
97.     int nUp_x = 0,nDown_x = 0,nLeft_x = 0,nRight_x = 0;
98.     int nUp_y = 0,nDown_y = 0,nLeft_y = 0,nRight_y = 0;
99.
100.    nUp_x = nXPos; nUp_y = nYPos-1;
101.    nDown_x = nXPos;nDown_y = nYPos+1;
102.    nLeft_x = nXPos-1;nLeft_y = nYPos;
103.    nRight_x = nXPos+1;nRight_y = nYPos;
104.
105.    nM axtric[nXPos][nYPos] -= 1;
106.    if(nUp_x >= 0 && nUp_x < nXLen && nUp_y >= 0 && nUp_y < nYLen)nM axtric[nUp_x][nUp_y]
      -= 1;

```

```

107.    if(nDown_x >= 0 && nDown_x < nXLen && nDown_y >= 0 && nDown_y < nYLen)nMaxtric[nDo
wn_x][nDown_y] -= 1;
108.    if(nLeft_x >= 0 && nLeft_x < nXLen && nLeft_y >= 0 && nLeft_y < nYLen)nMaxtric[nLeft_x][nL
eft_y] -= 1;
109.    if(nRight_x >= 0 && nRight_x < nXLen && nRight_y >= 0 && nRight_y < nYLen)nMaxtric[nRight
_x][nRight_y] -= 1;
110.
111.    return true;
112. }
113. // 递归判读
114. bool MaxtricTrans(int nMaxtric[INIT_SIZE][INIT_SIZE],int nXLen,int nYLen)
115. {
116.    if(isZeroMaxtric(nMaxtric,nXLen,nYLen))return true;
117.
118.    for(int i = 0;i < nXLen;i++)
119.        for(int j = 0;j < nYLen;j++)
120.        {
121.            if(testPosIsTemplate(nMaxtric,nXLen,nYLen,i,j))
122.            {
123.                DecTepplateMaxtric(nMaxtric,nXLen,nYLen,i,j);
124.                goto END;
125.            }
126.        }
127.
128.    return isZeroMaxtric(nMaxtric,nXLen,nYLen);
129. END:
130.    if(MaxtricTrans(nMaxtric,nXLen,nYLen))return true;
131.    else return false;
132. }
133. bool isMaxtricTrans(int nMaxtric[INIT_SIZE][INIT_SIZE],int nXLen,int nYLen)
134. {
135.    if(nMaxtric == NULL ||
136.        nXLen <= 0 ||
137.        nYLen <= 0)
138.    return false;

```

```
139.  
140.     bool bTrue = false;  
141.  
142.     bTrue = MaxtricTrans(nMaxtric,nXLen,nYLen);  
143.  
144.     if(bTrue)cout << "可以" << endl;  
145.     else    cout << "不可以" << endl;  
146.  
147.     return true;  
148. }  
149.// 判读数组最大可分的数组份数  
150.bool getMaxAveSumOfArray(int nArray[INIT_SIZE],int nACurLen)  
151. {  
152.     if(nArray == NULL ||  
153.         nACurLen <= 0)  
154.         return false;  
155.  
156.  
157.     return true;  
158. }  
159.//-----  
160.// 答案:  
161.// 此处也是经典案例啊!....  
162.int testShares(int a[], int n, int m, int sum, int groupsum, int aux[], int goal, int& groupId) // 注，此处  
必须加引用，不然出错，因为值传递错误  
163. {  
164.     if (goal == 0)  
165.     {  
166.         groupId++;  
167.         if (groupId == m+1) return 1;  
168.  
169.         return 0; // 增加返回， 提供程序效率  
170.     }  
171.     for (int i=0; i <n; i++)  
172.     {
```

```

173. if (aux[i] != 0) continue;
174. aux[i] = groupId;
175.
176. if(testShares(a, n, m, sum, groupSum, aux, goal-a[i], groupId))
177. {
178.     return 1;
179. }
180. else
181.     aux[i] = 0;
182. }
183. return 0;
184. }

185. int maxShares(int a[], int n)
186. {
187.     int sum = 0;
188.     int i, m;
189.     for (i=0; i < n; i++) sum += a[i];
190.     for (m=n; m >= 2; m--)
191.     {
192.         if (sum % m != 0) continue;
193.         int aux[INIT_SIZE];
194.         int groupId = 1;
195.         for (i=0; i < INIT_SIZE; i++) aux[i] = 0;
196.         if (testShares(a, n, m, sum, sum/m, aux, sum/m, groupId)) return m;
197.     }
198.     return 1;
199. }
200.
201. int main(int argc, char* argv[])
202. {
203.     memset(nMatrix, 0, sizeof(int)*INIT_SIZE*INIT_SIZE);
204.     memset(nArray, 0, sizeof(int)*INIT_SIZE);
205.
206.     int nNum = 0;
207. #if(Q_MODE_ == 1)

```

```
208. cout<<"请输入 nMaxtric 相应的数据<输入符号时结束相应的 行/列 输入>:\n"<<endl;
209. int nX = 1,nY = 1;
210. while(1)
211. {
212.     cout<<"请输入第"<<nX<<"行数据"<<endl;
213.     cin>>nNum;
214.     if (cin.fail())
215.     {
216.         break;
217.     }
218.     else
219.     {
220.         nMaxtric[nX-1][nY-1] = nNum;
221.         nY++;
222.     }
223. // 继续输入改行的数据
224.     while(1)
225.     {
226.         cin>>nNum;
227.         if (cin.fail())
228.         {
229.             // 更新 X 轴的最大值
230.             if(nY > nXLen)nXLen = nY-1;
231.             //清理缓冲区
232.             cin.clear();
233.             cin.ignore(1000, '\n');
234.             break;
235.         }
236.     else
237.     {
238.         nMaxtric[nX-1][nY-1] = nNum;
239.         nY++;
240.     }
241. }
242.
```

```
243.     nX++;
244.     nY = 1;
245. }
246. nYLen = nX-1;
247.
248. if(!isMaxtricTrans(nMaxtric,nYLen,nXLen))
249. {
250.     cout << "Param Error!..." << endl;
251. }
252. #else if(Q_MODE_ == 2)
253.     cout << "请输入数组相应的数据<输入符号时结束输入>:\n" << endl;
254.     while(nACurLen <= INIT_SIZE)
255.     {
256.         cin >> nNum;
257.         if (cin.fail())
258.         {
259.             break;
260.         }
261.         else
262.         {
263.             nArray[nACurLen ++] = nNum;
264.         }
265.     }
266.
267.     cout << "最大的可分份数为 " << maxShares(nArray,nACurLen) << endl;
268. #endif
269.
270. cin.ignore(1000, '\n'); //清理缓冲区
271. getch();
272. return 0;
273. }
```

1.4. 面试题集合（三）

1.4.1. 递归和非递归俩种方法实现二叉树的前序遍历

咱们先来复习下，基础知识。

二叉树结点存储的数据结构：

```
typedef char datatype;  
typedef struct node  
{  
datatype data;  
struct node* lchild,*rchild;  
} bintnode;  
typedef bintnode* bintree;  
bintree root;
```

1.树的前序遍历即：

按根 左 右 的顺序，依次

前序遍历根结点->前序遍历左子树->前序遍历右子树

前序遍历，递归算法

```
void preorder(bintree t)  
//注， bintree 为一指向二叉树根结点的指针  
{  
if(t)  
{  
printf("%c",t->data);  
preorder(t->lchild);  
preorder(t->rchild);  
}  
}
```

然后，依葫芦画瓢，得到....

2.中序遍历，递归算法

```
void preorder(bintree t)  
{  
if(t)  
{
```

```
inorder(t->lchild);
printf("%c",t->data);
inorder(t->rchild);
}
}
```

3.后序遍历，递归算法

```
void preorder(bintree t)
{
if(t)
{
postorder(t->lchild);
postorder(t->rchild);
printf("%c",t->data);
}
}
```

二叉树的创建方法，

```
void createbintree(bintree* t)
{
char ch;
if( (ch=getchar())==' ')
*t=NULL;
else
{
*t=(bintnode*) malloc(sizeof(bintnode));
(*t)->data=ch;
createbintree(&(*t)->lchild);
createbintree(&(*t)->rchild);
}
}
```

接下来，

咱们在讨论二叉树遍历算法的非递归实现之前，
先看一个顺序栈的定义及其部分操作的实现

```
typedef struct stack
{
```

```

bintree data[100];
int tag[100];
int top;
}seqstack;
void push(seqstack* s,bintree t)
{
s->data[s->top]=t;
s->top++;
}
bintree pop(seqstack* s) //出栈
{
if(s->top!=0)
{
s->top--;
return (s->data[s->top]);
}
else
return NULL;
}

```

好了，现在，我们可以看二叉树前序遍历的非递归实现了。

按照二叉树前序遍历的定义，无论是访问整棵树还是其子树，均应该遵循先访问根结点，然后访问根结点的左子树，最后访问根结点的右子树的。

因为对于一棵树（子树） t ,如果 t 非空，访问完 t 的根结点值后，就应该进入 t 的左子树，但此时必须将 t 保存起来，以便访问完其左子树后，进入其右子树的访问。

yeah，就是这个意思。:) ...

即在 t 处设置一个回溯点，并将该回溯点进栈保存。

在整个二叉树前序遍历的过程中，程序始终要做的工作分成俩个部分：

- 1.当前正在处理的树（子树）
- 2.保存在栈中等待处理的部分。

//注：当栈中元素位于栈顶即将出栈时，意味着其根结点和左子树已访问完成，

//出栈后，进入其右子树进行访问，

//前序遍历，非递归实现

```

void preorderT(bintree t)
{

```

```

seqstack s;
s.top=0;
while( (t)|(s.top!=0) )//当前处理的子树不为空或栈不为空
{
    while(t) //子树不为空
    {
        printf("%c",t->data); //1.先访问根结点
        push(&s,t); //2.访问左子树之前, 记得先把根结点进栈保存
        t=t->lchild; //3.然后才访问左子树,
    }
    if(s.top>0) //栈不为空
    {
        t.pop(&s); //4.pop 根结点
        t=t->rchild; //5.访问右子树
    }
}

//中序遍历, 非递归实现,
void inorderT(bintree t)
{
    seqstack s;
    s.top=0;
    while( (t)|(s.top!=0) )//当前处理的子树不为空或栈不为空
    {
        while(t) //子树不为空
        {
            push(&s,t); //1.访问左子树之前, 记得先把根结点 push 进栈
            t=t->lchild; //2.访问左子树
        }
        if(s.top!=0) //栈不为空
        {
            t.pop(&s); //3.pop 根结点(访问完左子树后)
            printf("%c",t->data); //4.访问根结点 (把先前保存的 t 给拿出来, 要用了..)
            t=t->rchild; //5.访问右子树
        }
    }
}

```

```

}
}

}

//后序遍历，非递归实现

后序遍历的非递归算法，稍微复杂点。请看，

按照二叉树后序遍历的定义，无论是访问整棵树还是起子树，

均应该遵循先访问根结点左子树，然后访问根结点的右子树，最后访问根结点。

值得注意的是，当一个元素位于栈顶即将处理的是，其左子树的访问一定完成，

如果其右子树不为空，接下来应该进入其右子树尽情访问。

//注意了，

但此时该栈顶元素时不能出栈的，因为它作为根结点，其本身的值还未被访问。

只有等到其右子树也访问完成后，该栈顶元素才能出栈，并输出它的值。

因此，在二叉树后序遍历的算法中，必须使用 seqstack 类型的数组 tag，

其每个元素取值为 0 或 1，用于标识栈中每个元素的状态。

```

- 1.当一个元素刚进栈时，其对应的 tag 值置为 0；
- 2.当它位于栈顶即将被处理时，其 tag 值为 0.意味着应该访问其右子树。
于是将右子树作为当前处理的对象，此时该栈顶元素仍应该保留在栈中。
并将其对应的 tag 值改为 1.
- 3.当其右子树访问完成后，该元素又一次位于栈顶，而此时其 tag 值为 1，
意味着其右子树已访问完成，接下来，应该直接访问的就是它，将其出栈。

```

void postorderT(bintree t)
{
    seqstack s;
    s.top=0;
    while( (t)||(s.top!=0) )
    {
        while(t)
        {
            s.data[s.top]=t;
            s.tag[s.top]=0; //tag 置为 0
            s.top++;
            t=t->lchild; //访问左子树
        }
        while( (s.top>0)&&(s.tag[s.top-1]==1) )

```

```
{  
    s.top--;  
    t=s.data[s.top];  
    printf("%c",t->data);  
}  
if(s.top>0)  
{  
    t=s.data[s.top-1];  
    s.tag[s.top-1]=1;  
    t=t->rchild;  
}  
else  
    t=NULL;  
}  
}
```

1.4.2. 请修改 append 函数，利用这个函数实现

两个非降序链表的并集，1->2->3 和 2->3->5 并为 1->2->3->5

另外只能输出结果，不能修改两个链表的数据。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <stdio.h>  
2. #include <stdlib.h>  
3. #include <malloc.h>  
4.  
5. struct Node  
6. {  
7.     int num;  
8.     Node * next;  
9. };  
10.  
11. Node * createTail()  
12. {
```

```
13. int x;
14. Node *head = NULL, *p = NULL, *tail = NULL;
15. puts("\nplease enter some digits(end of '.'):");
16. while( scanf("%d",&x) )
17. {
18.     p = (Node *)malloc(sizeof(Node));
19.     p->num = x;
20.     p->next = NULL;
21.     if( NULL == head )
22.     {
23.         tail = p;
24.         head = tail;
25.     }
26.     else
27.     {
28.         tail->next = p;
29.         tail = p;
30.     }
31. }
32. getchar();
33. return head;
34. }
35.
36. Node * CombinationNode(Node* head1, Node* head2)
37. {
38.     Node *head,*tail,*p = head1,*q = head2,*s;
39.
40.     if( NULL == p )
41.         return q;
42.     if( NULL == q )
43.         return p;
44.
45.     tail = p;
46.     if( p->num > q->num)
47.         tail = q;
```

```
48.     head = tail;
49.
50.     while( NULL != p && NULL != q )
51.     {
52.         if(p->num <= q->num )
53.         {
54.             s = p;
55.             p = p->next;
56.         }
57.         else
58.         {
59.             s = q;
60.             q = q->next;
61.         }
62.         tail->next = s;
63.         tail = s;
64.     }
65.
66.     if( NULL == p ) p = q;
67.
68.     s = p;
69.     tail->next = s;
70.
71.     return head;
72. }
73.
74. void printHead(Node *head)
75. {
76.     if( NULL == head )
77.         return;
78.     printf("List: ");
79.     while(head)
80.     {
81.         printf("%d->",head->num);
82.         head = head->next;
```

```
83.    }
84.    puts("NUL");
85. }
86.
87. void main( void )
88. {
89.    Node* head1,*head2,*head;
90.    head1 = createTail();
91.    printHead(head1);
92.
93.    head2 = createTail();
94.    printHead(head2);
95.
96.    head = CombinationNode(head1,head2);
97.    printHead(head);
98. }
99. ///////////////////////////////////////////////////
100.
101. please enter some digits(end of '.'):
102. 1
103. 3
104. 5
105. 7
106. 9
107. .
108. List: 1->3->5->7->9->NUL
109.
110. please enter some digits(end of '.'):
111. 2
112. 4
113. 5
114. 6
115. 7
116. 8
117. 9
```

```
118. .
119. List: 2->4->5->6->7->8->9->NUL
120. List: 1->2->3->4->5->5->6->7->7->8->9->9->NUL
121. Press any key to continue
122. /////////////////////////////////
```

1.4.3. 有 n 个长为 m+1 的字符串

有 n 个长为 m+1 的字符串，如果某个字符串的最后 m 个字符与某个字符串的前 m 个字符匹配，则两个字符串可以联接，问这 n 个字符串最多可以连成一个多长的字符串，如果出现循环，则返回错误。

分析一下，将各个字符串作为一个节点，首尾链接就好比是一条边，将两个节点连接起来，于是问题就变成一个有关图的路径长度的问题。链接所得的字符串最长长度即为从图的某个节点出发所能得到的最长路径问题，与最短路径类似，可以应用弗洛伊德算法求解。对于循环，即可认为各个节点通过其他节点又回到自己，反应在路径长度上，就表示某个节点到自己节点的路径大于零（注：初始化个节点到自己的长度为零）。

[java] [view plain](#) [copy](#) [print](#)?

```
1. public class MaxCatenate {
2.
3.     public static void main(String[] args)
4.     {
5.         String[] text = new String[]{
6.             "abcd",
7.             "bcde",
8.             "cdea",
9.             "deab",
10.            "eaba",
11.            "abab",
12.            "deac",
13.            "cdei",
14.            "bcd",
```

```

15.         "cdfi",
16.         "dfic",
17.         "cdfk",
18.         "bcdg",
19.     );
20.     maxCatenate(text);
21. }
22.
23. public static void maxCatenate(String[] text)
24. {
25.     int[][] G = new int[text.length][text.length];
26.     for(int i=0; i<G.length; i++)
27.     {
28.         String suffix = text[i].substring(1);
29.         for(int j=0; j<G.length; j++)
30.             if(text[j].indexOf(suffix)==0)
31.                 G[i][j] = 1;
32.     }
33.     for(int k=0; k<G.length; k++)
34.         for(int i=0; i<G.length; i++)
35.             for(int j=0; j<G.length; j++)
36.                 if(G[i][k]!=0&&G[k][j]!=0)
37.                 {
38.                     int dist = G[i][k] + G[k][j];
39.                     if(dist>G[i][j])
40.                     {
41.                         G[i][j] = dist;
42.                     }
43.                 }
44.     for(int i=0; i<G.length; i++)
45.         if(G[i][i]>1)
46.         {
47.             System.out.println("circle is detected!");
48.             return;
49.         }

```

```
50.     int max = 0;
51.     for(int i=0; i<G.length; i++)
52.         for(int j=0; j<G.length; j++)
53.             max = Math.max(max, G[i][j]);
54.     System.out.println("Max length is " + (max+text[0].length()));
55. }
56.
57. }
```

1.4.4. n 支队伍比赛

n 支队伍比赛，分别编号为 0, 1, 2。。。。n-1，已知它们之间的实力对比关系，存储在一个二维数组 w[n][n] 中，w[i][j] 的值代表编号为 i, j 的队伍中更强的一支。所以 w[i][j]=i 或者 j，现在给出它们的出场顺序，并存储在数组 order[n] 中，比如 order[n] = {4,3,5,8,1.....}，那么第一轮比赛就是 4 对 3， 5 对 8。.....胜者晋级，败者淘汰，同一轮淘汰的所有队伍排名不再细分，即可以随便排，下一轮由上一轮的胜者按照顺序，再依次两两比，比如可能是 4 对 5，直至出现第一名编程实现，给出二维数组 w，一维数组 order 和 用于输出比赛名次的数组 result[n]，求出 result。

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. //coder:Lee,20120320
2.
3. #include<iostream>
4. #include<cassert>
5. using namespace std;
6. void GameResult(int w[][4],int *order,int n)
7. {
8.     assert(w!=NULL&&*w!=NULL&&order!=NULL&&n>0);
9.     if(n==1)
10.    return;
11.    int *result=new int[n];
12.    int eliminationIndex=n/2;
```

```
13. if(n%2==1)
14. eliminationIndex+=1;
15. for(int i=0;i<n/2;i++)
16. {
17. if (w[order[2*i]][order[2*i+1]]==order[2*i])
18. {
19. result[i]=order[2*i];
20. result[i+eliminationIndex]=order[2*i+1];
21. }
22. else
23. {
24. result[i]=order[2*i+1];
25. result[i+eliminationIndex]=order[2*i];
26. }
27. }
28. if(n%2==1)
29. result[i]=order[2*i];
30. for(i=0;i<n;i++)
31. order[i]=result[i];
32. GameResult(w,order,eliminationIndex);
33. }
34.
35.
36.
37.
38. int main()
39. {
40.
41.
42. int w[4][4]={ {0,0,2,3},{0,1,2,3},{2,2,2,2},{3,3,2,3} };
43. int order[4]={2,3,0,1};
44. GameResult(w,order,4);
45. for(int i=0;i<4;i++)
46. cout<<order[i]<<" ";
47. return 0;
```

```
48. }
```

1.4.5. 求一个矩阵中最大的二维矩阵(元素和最大)

求一个矩阵中最大的二维矩阵(元素和最大).如:

1 2 0 3 4

2 3 4 5 1

1 1 5 3 0

中最大的是:

4 5

5 3

要求:(1)写出算法;(2)分析时间复杂度;(3)用 C 写出关键代码

[java] [view plain](#) [copy](#) [print](#)?

```
1. public class MaxSubMatrix {  
2.  
3.     /**see http://blog.csdn.net/zhanxinhang/article/details/6731134  
4.     * Q35  
5.     求一个矩阵中最大的二维矩阵 ( 元素和最大 ). 如 :  
6.     1 2 0 3 4  
7.     2 3 4 5 1  
8.     1 1 5 3 0  
9.     中最大的是 :  
10.    4 5  
11.    5 3  
12.    three solutions:  
13.    1.brutalFind:calculate all the possible sum and find the largest  
14.    2.verticalFind:when matrix's rowIndex is larger than columnIndex,like "int[][] b" in the following code  
15.    3.horizonalFind:when matrix's rowIndex is smaller than columnIndex,like "int[][] a" in the following cod  
e  
16.    both 2 and 3 avoid duplicate calculation  
17.    */  
18.    public static void main(String[] args) {
```

```

19.     MaxSubMatrix msm=new MaxSubMatrix();
20.
21.     int[][] a={
22.         {2,3,4},
23.         {1,3,3},
24.         {1,4,6},
25.         {1,4,8},
26.         {2,3,2},
27.     };
28.
29.     int[][] b={
30.         {1, 2, 0, 3, 4},
31.         {2, 3, 4, 5, 1},
32.         {1, 1, 5, 3, 0},
33.     };
34.
35.     int[][] result=msm.findSumMaxSubMatrix(a);
36.     msm.printArray(result);
37.     System.out.println("-----");
38.     result=msm.brutalFind(a);
39.     msm.printArray(result);
40.     System.out.println("-----");
41.     result=msm.findSumMaxSubMatrix(b);
42.     msm.printArray(result);
43.     System.out.println("-----");
44.     result=msm.brutalFind(b);
45.     msm.printArray(result);
46.
47.     public int[][] findSumMaxSubMatrix(int[][] a){
48.         int[][] result=null;
49.         int row=a.length;
50.         int col=a[0].length;
51.         if(row>=col){
52.             result=herizontalFind(a,row,col);
53.         }else{

```

```
54.         result=verticalFind(a,row,col);
55.     }
56.     return result;
57. }
58.
59. public int[][] brutalFind(int[][] a){
60.     int[][] result=new int[2][2];
61.     int row=a.length;
62.     int col=a[0].length;
63.     int sum=0;
64.     int p=0;
65.     int q=0;
66.     for(int i=0;i<row-1;i++){
67.         for(int j=0;j<col-1;j++){
68.             int x=a[i][j];
69.             int y=a[i][j+1];
70.             int z=a[i+1][j];
71.             int k=a[i+1][j+1];
72.             int temp=x+y+z+k;
73.             if(temp>sum){
74.                 sum=temp;
75.                 p=i;
76.                 q=j;
77.             }
78.         }
79.     }
80.     result[0][0]=a[p][q];
81.     result[0][1]=a[p][q+1];
82.     result[1][0]=a[p+1][q];
83.     result[1][1]=a[p+1][q+1];
84.     return result;
85. }
86.
87. public int[][] horizontalFind(int[][] a,int row,int col){
88.     int[][] result=new int[2][2];
```

```

89.     int lastHorizontalSum=a[0][0]+a[0][1];
90.     int sum=0;
91.     int p=0;
92.     int q=0;
93.     for(int i=1;i<row;i++){
94.         for(int j=0;j<col-1;j++){
95.             int temp=lastHorizontalSum+a[i][j]+a[i][j+1];
96.             lastHorizontalSum=a[i][j]+a[i][j+1];
97.             if(temp>sum){
98.                 sum=temp;
99.                 p=i;
100.                q=j;
101.            }
102.        }
103.    }
104.    result[0][0]=a[p-1][q];
105.    result[0][1]=a[p-1][q+1];
106.    result[1][0]=a[p][q];
107.    result[1][1]=a[p][q+1];
108.    return result;
109. }
110.
111. public int[][] verticalFind(int[][] a,int row,int col){
112.     int[][] result=new int[2][2];
113.     int lastVerticalSum=a[0][0]+a[1][0];
114.     int sum=0;
115.     int p=0;
116.     int q=0;
117.     for(int i=0;i<row-1;i++){
118.         for(int j=1;j<col;j++){
119.             int temp=lastVerticalSum+a[i][j]+a[i+1][j];
120.             lastVerticalSum=a[i][j]+a[i+1][j];
121.             if(temp>sum){
122.                 sum=temp;
123.                 p=i;

```

```

124.         q=j;
125.     }
126. }
127. }
128. result[0][0]=a[p][q-1];
129. result[0][1]=a[p][q];
130. result[1][0]=a[p+1][q-1];
131. result[1][1]=a[p+1][q];
132. return result;
133. }
134.
135. public void printArray(int[][] a){
136.     int row=a.length;
137.     int col=a[0].length;
138.     for(int i=0;i<row;i++){
139.         for(int j=0;j<col;j++){
140.             System.out.print(a[i][j]+" ");
141.         }
142.         System.out.println();
143.     }
144. }
145. }

```

1.4.6. 强大的和谐

实现一个挺高级的字符匹配算法：

给一串很长字符串，要求找到符合要求的字符串，例如目的串：123

1*****3***2 ， 12*****3 这些都要找出来，其实就是要找一些和谐系统。。。

这题的真正意思就是，给你一个目标串，如“123”，只要一个字符串里面同时包含1、2和3，那么这个字符串就匹配了。系统越和谐，说明错杀的可能行也就越大。加入目标串的长度为m，模式串的长度为n，我们很容易想到O(mn)的算法，就是两遍for循环搞定。那么有没有更快的方法呢？

我们考虑问题的时候，如果想时间变得快，有一种方法就叫做“空间换时间”。哈希表是一种比较复杂的数据结构。由于比较复杂，STL中没有实现哈希表，因

此需要我们自己实现一个。但由于本题的特殊性，我们只需要一个非常简单的哈希表就能满足要求。由于字符（char）是一个长度为 8 的数据类型，因此总共有可能 256 种可能。于是我们创建一个长度为 256 的数组，每个字母根据其 ASCII 码值作为数组的下标对应数组的对应项，而数组中存储的 0、1 对应每个字符是否出现。这样我们就创建了一个大小为 256，以字符 ASCII 码为键值的哈希表。（并不仅限于英文字符，所以这里要考虑 256 种可能）。

知道了这点，我们可以构建一个数组来统计模式串中某个字符是否出现，然后在对目标串进行扫描，看看对应的所有位上是否出现，从而判断是否匹配。分析一下复杂度，大概是 $O(m+n)$ 。

实现代码如下：

```
[cpp] view plaincopy
1. //强大的和谐系统
2. int isContain(char *src, char *des)
3. {
4.     //创建一个哈希表，并初始化
5.     const int tableSize = 256;
6.     int hashTable[tableSize];
7.     int len,i;
8.     for(i = 0; i < tableSize; i++)
9.         hashTable[i] = 0;
10.    len = strlen(src);
11.    for(i = 0; i < len; i++)
12.        hashTable[src[i]] = 1;
13.
14.    len = strlen(des);
15.    for(i = 0; i < len; i++)
16.    {
17.        if(hashTable[des[i]] == 0)
18.            return 0;      //匹配失败
19.    }
20.    return 1; //匹配成功
21. }
```

```
// p33_SearchDesStr.cpp : Defines the entry point for the console application.
// -----
//
```

```
/*
实现一个挺高级的字符匹配算法：
给一串很长字符串，要求找到符合要求的字符串，例如目的串：123
1*****3***2,12*****3 这些都要找出来
其实就是类似一些和谐系统。。。。

*/
//-----
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <string>

using namespace std;

#define SELFCODE
#define INIT_SIZE100

char szEWordsStr[INIT_SIZE];
int nEWLen = 0;

char szDesStr[INIT_SIZE];
int nDesLen = 0;

bool IsDesStr(char szCH,char *szDesStr,int nDesLen)
{
    for(int i = 0; i < nDesLen; i++)
    {
        if(*(szDesStr+i) == szCH)
            return true;
    }
}
```

```
}

return false;
}

bool GetDesStr(char *szEWordsStr,int nEWLen,char *szDesStr,int nDesLen)
{
if(szEWordsStr == NULL ||
szDesStr == NULL)
return false;
cout << "含有目的字符串的位置为: \n";
for(int i = 0; i< nEWLen ;i++)
{
if(IsDesStr(*(szEWordsStr+i),szDesStr,nDesLen))
{
cout << i+1 << ' ';
}
}
cout << endl;
return true;
}

int main(int argc, char* argv[])
{
memset(szEWordsStr,0,sizeof(char)*INIT_SIZE);
memset(szDesStr,0,sizeof(char)*INIT_SIZE);

cout<<"请输入相应的数据<最大数组为 100 元素>:\n"<<endl;
cin.getline(szEWordsStr,INIT_SIZE);
nEWLen = strlen(szEWordsStr);

cout<<"请输入相应的目的字符串<最大长度为 100 元素>:\n"<<endl;
cin.getline(szDesStr,INIT_SIZE);
nDesLen = strlen(szDesStr);
```

```
if(!GetDesStr(szEWordsStr,nEWLen,szDesStr,nDesLen))  
{  
    cout<< "Error!..."<<endl;  
}  
  
  
cin.ignore(1000,'\n'); //清理缓冲区  
getchar();  
return 0;  
}
```

1.4.7. 通过交换 a,b 中的元素，使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小

[java-32.通过交换 a,b 中的元素，使\[序列 a 元素的和\]与\[序列 b 元素的和\]之间的差最小。](#)

[java] view plaincopy

```
1. Java 代码  
2. import java.util.Arrays;  
3. public class MinSumASumB {  
4.     /**  
5.      * Q32.有两个序列 a,b，大小都为 n,序列元素的值任意整数，无序.  
6.      *  
7.      * 要求：通过交换 a,b 中的元素，使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小。  
8.      * 例如：  
9.      * int[] a = { 100,99,98,1,2,3};  
10.     * int[] b = { 1, 2, 3, 4,5,40};  
11.     *  
12.     * 求解思路：  
13.     * 当前数组 a 和数组 b 的和之差为 A = sum(a) - sum(b) a 的第 i 个元素和 b 的第 j 个元素交换后，  
14.     * a 和 b 的和之差为 A'  
15.     * =sum(a) - a[i] + b[j] - (sum(b) - b[j] + a[i])
```

```
16. * = sum(a) - sum(b) - 2 (a[i] - b[j])
17. * = A - 2 (a[i] - b[j])
18. * 设 x = a[i] - b[j], 则交换后差值变为 A' = A - 2x
19. *
20. * 假设 A > 0, 当 x 在 (0,A)之间时, 做这样的交换才能使得交换后的 a 和 b 的和之差变小, x 越接近
    A/2 效果越好,
21. * 如果找不到在(0,A)之间的 x, 则当前的 a 和 b 就是答案。所以算法大概如下:
22. * 在 a 和 b 中寻找使得 x 在(0,A)之间并且最接近 A/2 的 i 和 j, 交换相应的 i 和 j 元素, 重新计算 A
    后,
23. * 重复前面的步骤直至找不到(0,A)之间的 x 为止。
24. */
25. public static void main(String[] args) {
26.     MinSumASumB minSumASumB=new MinSumASumB();
27.     //int[] a = { 100,99,98,1,2,3};
28.     //int[] b = { 1, 2, 3, 4,5,40};
29.     //int[] a={3,5,10};
30.     //int[] b={20,25,50};
31.     int[] a={3,5,-10};
32.     int[] b={20,25,50};
33.     minSumASumB.swapToMinusDiff(a, b);
34.     System.out.println(Arrays.toString(a));
35.     System.out.println(Arrays.toString(b));
36. }
37. public void swapToMinusDiff(int[] a,int[] b){
38.     int sumA=sum(a);
39.     int sumB=sum(b);
40.     if(sumA==sumB) return;
41.     if(sumA<sumB){
42.         int[] temp=a;
43.         a=b;
```

```
44. b=temp;
45. }
46. int curDiff=1;
47. int oldDiff=Integer.MAX_VALUE;
48. int pA=-1;
49. int pB=-1;
50. boolean shift=true;
51. int len=a.length;//the length of a and b should be the same
52. while(shift&&curDiff>0){
53.     shift=false;
54.     curDiff=sum(a)-sum(b);
55.     for(int i=0;i<len;i++){
56.         for(int j=0;j<len;j++){
57.             int temp=a[i]-b[j];
58.             int newDiff=Math.abs(curDiff-2*temp);
59.             if(newDiff<curDiff&&newDiff<oldDiff){
60.                 shift=true;
61.                 oldDiff=newDiff;
62.                 pA=i;
63.                 pB=j;
64.             }
65.         }
66.     }
67.     if(shift){
68.         int temp=a[pA];
69.         a[pA]=b[pB];
70.         b[pB]=temp;
71.     }
72. }
73. System.out.println("the min diff is "+oldDiff);
```

```
74. }
75. public int sum(int[] a){
76.     int sum=0;
77.     for(int each:a){
78.         sum+=each;
79.     }
80.     return sum;
81. }
82. }
```

1.4.8. 计算 1 到 N 的十进制数中 1 的出现次数

问题描述：给定一个十进制正整数 N，写下从 1 开始，到 N 的所有整数，然后数一下其中出现的所有"1"的个数。例如：

N = 2，写下 1, 2。这样只出现了 1 个"1"。

N = 12，写下 1, 2, ..., 12，这样有 5 个"1"。

写一个函数 f(N)，返回 1 到 N 之间出现的"1"的个数，比如 $f(12) = 5$ 。

假设 $N = abcde$ ，这里 a,b,c,d,e 分别是十进制数 N 的各个数位上的数字。如果要计算百位上出现 1

的次数，将受 3 方面因素影响：百位上的数字，百位以下(低位)的数字，百位(更高位)以上的数字。

如果百位上的数字为 0，则可以知道百位上可能出现 1 的次数由更高位决定，比如 12 013，则可以知

道百位出现 1 的情况可能是 100—199, 1 100—1 199, ..., 11 100—11 199，一共有 1 200 个。也就是

由更高位数字(12) 决定，并且等于更高位数字(12)×当前位数(100)。

如果百位上的数字为 1，则可以知道，百位上可能出现 1 的次数不仅受更高位影响，还受低位影响，

也就是由更高位和低位共同决定。例如 12 113，受更高位影响，百位出现 1 的情况是 100—199, 1 100

—1 199, ..., 11 100—11 199，一共有 1 200 个，和上面第一种情况一样，等于更高位数字(12)×当

前位数(100)。但它还受低位影响，百位出现 1 的情况是 12 100—12 113，一共 114 个，等于低位数字

(113)+1。

如果百位上数字大于 1(即为 2—9)，则百位上可能出现 1 的次数也仅由更高位决定，比如 12 213，则

百位出现 1 的情况是：100—199，1 100—1 199，……，11 100—11 199，12 100—12 199，共 1300 个

， 并且等于更高位数字+1(12+1)×当前位数(100)。

```
#include <iostream>
#include <windows.h>
using namespace std;
ULL Sum1s( ULL n )
{
    ULL iCount = 0;
    ULL iFactor = 1;
    ULL iLowerNum = 0;
    ULL iCurrNum = 0;
    ULL iHigherNum = 0;
    while( n / iFactor != 0 )
    {
        iLowerNum = n - ( n / iFactor ) * iFactor;
        iCurrNum = (n / iFactor) % 10;
        iHigherNum = n / ( iFactor * 10 );
        switch( iCurrNum )
        {
            case 0:
                iCount += iHigherNum * iFactor;
                break;
            case 1:
                iCount += iHigherNum * iFactor + iLowerNum + 1;
                break;
            default:
                iCount += ( iHigherNum + 1 ) * iFactor;
                break;
        }
    }
}
```

```
iFactor *= 10;  
}  
return iCount;  
}  
  
int main()  
{  
    cout << Sum1s(100000000);  
    cin.get();  
    return 0;  
}
```

1.4.9. 栈的 push、pop 序列[数据结构]

题目：输入两个整数序列。其中一个序列表示栈的 push 顺序，判断另一个序列有没有可能是对应的 pop 顺序。为了简单起见，我们假设 push 序列的任意两个整数都是不相等的。

比如输入的 push 序列是 1、2、3、4、5，那么 4、5、3、2、1 就有可能是一个 pop 系列。

因为可以有如下的 push 和 pop 序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，这样得到的 pop 序列就是 4、5、3、2、1。但序列 4、3、5、1、2 就不可能是 push 序列 1、2、3、4、5 的 pop 序列。

分析：这道题除了考查对栈这一基本数据结构的理解，还能考查我们的分析能力。

这道题的一个很直观的想法就是建立一个辅助栈，每次 push 的时候就把一个整数 push 进入这个辅助栈，同样需要 pop 的时候就把该栈的栈顶整数 pop 出来。

我们以前面的序列 4、5、3、2、1 为例。第一个希望被 pop 出来的数字是 4，因此 4 需要先 push 到栈里面。由于 push 的顺序已经由 push 序列确定了，也就是在把 4 push 进栈之前，数字 1，2，3 都需要 push 到栈里面。此时栈里的包含 4 个数字，分别是 1，2，3，4，其中 4 位于栈顶。把 4 pop 出栈后，剩下三个数字 1，2，3。接下来希望被 pop 的是 5，由于仍然不是栈顶数字，我们接着在 push 序列中 4 以后的数字中寻找。找到数字 5 后再一次 push 进栈，这个时候 5 就是位于栈顶，可以被 pop 出来。接下来希望被 pop 的三个数字是 3，2，1。每次操作前都位于栈顶，直接 pop 即可。

再来看序列 4、3、5、1、2。pop 数字 4 的情况和前面一样。把 4 pop 出来之后，3 位于栈顶，直接 pop。接下来希望 pop 的数字是 5，由于 5 不是栈顶数字，我们到 push 序列中没有被 push 进栈的数字中去搜索该数字，幸运的时候能够找到 5，于是把 5 push 进入栈。此时 pop 5 之后，栈内包含两个数字 1、2，其中 2 位于栈顶。这个时候希望 pop 的数字是 1，由于不是栈顶数字，我们需要到 push 序列中还没有被 push 进栈的数字中去搜索该数字。但此时 push 序列中所有数字都已被 push 进入栈，因此该序列不可能是一个 pop 序列。

也就是说，如果我们希望 pop 的数字正好是栈顶数字，直接 pop 出栈即可；如果希望 pop 的数字目前不在栈顶，我们就到 push 序列中还没有被 push 到栈里的数字中去搜索这个数字，并把在它之前的所有数字都 push 进栈。如果所有的数字都被 push 进栈仍然没有找到这个数字，表明该序列不可能是一个 pop 序列。

基于前面的分析，我们可以写出如下的参考代码：

```
#include <stack>

///////////
// Given a push order of a stack, determine whether an array is possible to
// be its corresponding pop order
// Input: pPush - an array of integers, the push order
// pPop - an array of integers, the pop order
// nLength - the length of pPush and pPop
// Output: If pPop is possible to be the pop order of pPush, return true.
// Otherwise return false
///////////

bool IsPossiblePopOrder(const int* pPush, const int* pPop, int nLength)
{
    bool bPossible = false;

    if(pPush && pPop && nLength > 0)
    {
        const int *pNextPush = pPush;
        const int *pNextPop = pPop;

        // ancillary stack
        std::stack<int> stackData;

        // check every integers in pPop
        while(pNextPop - pPop < nLength)
        {
            // while the top of the ancillary stack is not the integer
            // to be popped, try to push some integers into the stack
            while(stackData.empty() || stackData.top() != *pNextPop)
            {

```

```

// pNextPush == NULL means all integers have been
// pushed into the stack, can't push any longer
if(!pNextPush)
break;

stackData.push(*pNextPush);

// if there are integers left in pPush, move
// pNextPush forward, otherwise set it to be NULL
if(pNextPush - pPush < nLength - 1)
pNextPush++;
else
pNextPush = NULL;
}

// After pushing, the top of stack is still not same as
// pNextPop, pNextPop is not in a pop sequence
// corresponding to pPush
if(stackData.top() != *pNextPop)
break;

// Check the next integer in pPop
stackData.pop();
pNextPop++;
}

// if all integers in pPop have been check successfully,
// pPop is a pop sequence corresponding to pPush
if(stackData.empty() && pNextPop - pPop == nLength)
bPossible = true;
}

return bPossible;
}

```

1.4.10. 统计整数二进制表示中 1 的个数

这是一个很有意思的问题，也是在面试中最容易被问到的问题之一。这个问题有个正式的名字叫 [Hamming_weight](#)，而且 wikipedia 上也提供了很好的位运算解决的方法，这个下面也会提到。

解决这个问题的第一想法是一位一位的观察，判断是否为 1，是则计数器加一，否则跳到下一位，于是很容易有这样的程序。

```
?  
1 int test(int n)  
2 {  
3     int count=0;  
4     while(n != 0){  
5         if(n%2 ==1)  
6             count++;  
7         n /= 2;  
8     }  
9     return count;  
10}
```

或者和其等价的位运算版本：

```
?  
1 int test(int n)  
2 {  
3     int count=0;  
4     while(n != 0){  
5         count += n&1;  
6         n >>= 1;  
7     }  
8     return count;  
9 }  
10
```

这样的方法复杂度为二进制的位数，即 $\Theta(\log n)$ ，于是可是想一下，有没有只与二进制中 1 的位数相关的算法呢。

可以考虑每次找到从最低位开始遇到的第一个 1，计数，再把它清零，清零的位运算操作是与一个零，但是在有 1 的这一位与零的操作要同时不影响未统计过的位数和已经统计过的位数，于是可以有这样一个操作 $n \& (n-1)$ ，这个操作对比当前操作位高的位没有影响，对低位则完全清零。拿 6 (110) 来做例子，第一次 $110 \& 101 = 100$ ，这次操作成功的把从低位起第一个 1 消掉了，同时计数器加 1，第二次 $100 \& 011 = 000$ ，同理又统计了高位的一个 1，此时 n 已变为 0，不需要再继续了，于是 110 中有 2 个 1。

代码如下：

```
?  
1 int test(int n)  
2 {  
3     int count=0;  
4     while(n != 0){  
5         n &= n-1;  
6         count++;  
7     }  
8     return count;  
9 }
```

这几个方法虽然也用到了位运算，但是并没有体现其神奇之处，下面这个版本则彰显位运算的强大能力，若不告诉这个函数的功能，一般一眼看上去是想不到这是做什么的，这也是 wikipedia 上给出的计算 hamming_weight 方法。

```
?  
1 int test(int n)  
2 {  
3     n = (n&0x55555555) + ((n>>1)&0x55555555);  
4     n = (n&0x33333333) + ((n>>2)&0x33333333);  
5     n = (n&0x0f0f0f0f) + ((n>>4)&0x0f0f0f0f);  
6     n = (n&0x00ff00ff) + ((n>>8)&0x00ff00ff);  
7     n = (n&0x0000ffff) + ((n>>16)&0x0000ffff);  
8     return n;  
9 }  
10 }
```

没有循环，5 个位运算语句，一次搞定。

比如这个例子，143 的二进制表示是 10001111，这里只有 8 位，高位的 0 怎么进行与的位运算也是 0，所以只考虑低位的运算，按照这个算法走一次

```
+---+---+---+---+---+---+  
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | <-- 143  
+---+---+---+---+---+---+  
| 0 1 | 0 0 | 1 0 | 1 0 | <-- 第一次运算后  
+-----+-----+  
| 0 0 0 1 | 0 1 0 0 | <-- 第二次运算后  
+-----+  
| 0 0 0 0 0 1 0 1 | <-- 第三次运算后，得数为 5  
+-----+
```

这里运用了分治的思想，先计算每对相邻的 2 位中有几个 1，再计算每相邻的 4 位中有几个 1，下来 8 位，16 位，32 位，因为 $2^5=32$ ，所以对于 32 位的机器，5 条位运算语句就够了。

像这里第二行第一个格子中，01 就表示前两位有 1 个 1，00 表示下来的两位中没有 1，其实同理。再下来 $01+00=0001$ 表示前四位中有 1 个 1，同样的 $10+10=0100$ 表示低四位中有 4 个 1，最后一步 $0001+0100=00000101$ 表示整个 8 位中有 5 个 1。

1.5. 面试题集合（四）

1.5.1. 跳台阶问题

题目：一个台阶总共有 n 级，如果一次可以跳 1 级，也可以跳 2 级。求总共有多少总跳法。

分析：

这道题最近经常出现，包括 MicroStrategy 等比较重视算法的公司都曾先后选用过这个这道题作为面试题或者笔试题。

首先我们考虑最简单的情况。如果只有 1 级台阶，那显然只有一种跳法。

如果有 2 级台阶，那就有两种跳的方法了：一种是分两次跳，每次跳 1 级；另外一种就是一次跳 2 级。

现在我们再来讨论一般情况。我们把 n 级台阶时的跳法看成是 n 的函数，记为 $f(n)$ 。

当 $n=1$ 时有 1 种跳法

一次跳一阶

当 $n=2$ 时有 2 种跳法

一次跳一阶， 1 1

一次跳二阶 2

当 $n>2$ 时，第一次跳的时候就有两种不同的选择：

一是第一次只跳 1 级，此时跳法数目等于后面剩下的 $n-1$ 级台阶的跳法数目，即为 $f(n-1)$ ；

另外一种选择是第一次跳 2 级，此时跳法数目等于后面剩下的 $n-2$ 级台阶的跳法数目，即为 $f(n-2)$ 。

因此 n 级台阶时的不同跳法的总数 $f(n)=f(n-1)+ f(n-2)$ 。

```
#coding=gbk
# 题目：一个台阶总共有 n 级，如果一次可以跳 1 级，也可以跳 2 级。
# 求总共有多少总跳法？
```

```
def f( n ):
    a = 1
    b = 2
    if n <= 1:
        return a
    if n<=2:
        return b
    for i in range( 3,n+1):
        t= a + b
        a = b
        b = t
    return b
print( f( 5))
结果---
```

1.5.2. 左旋转字符串

题目描述：

定义字符串的左旋转操作：把字符串前面的若干个字符移动到字符串的尾部。
如把字符串 abcdef 左旋转 2 位得到字符串 cdefab。

请实现字符串左旋转的函数，要求对长度为 n 的字符串操作的时间复杂度为 $O(n)$ ，
空间复杂度为 $O(1)$ 。

编程之美上有这样一个类似的问题，咱们先来看一下：

设计一个算法，把一个含有 N 个元素的数组循环右移 K 位，要求时间复杂度为 O (N) ，且只允许使用两个附加变量。

分析：

我们先试验简单的办法，可以每次将数组中的元素右移一位，循环 K 次。

abcd1234→4abcd123→34abcd12→234abcd1→1234abcd。

```
RightShift(int* arr, int N, int K)
```

```
{  
    while(K--){  
        int t = arr[N - 1];  
        for(int i = N - 1; i > 0; i --)  
            arr[i] = arr[i - 1];  
        arr[0] = t;  
    }  
}
```

虽然这个算法可以实现数组的循环右移，但是算法复杂度为 $O (K * N)$ ，不符合题目的要求，要继续探索。

假如数组为 abcd1234，循环右移 4 位的话，我们希望到达的状态是 1234abcd。

不妨设 K 是一个非负的整数，当 K 为负整数的时候，右移 K 位，相当于左移 (-K) 位。

左移和右移在本质上是一样的。

解法一：

大家开始可能会有这样的潜在假设， $K < N$ 。事实上，很多时候也的确是这样的。但严格来说，我们不能用这样的“惯性思维”来思考问题。

尤其在编程的时候，全面地考虑问题是很重要的，K 可能是一个远大于 N 的整数，在这个时候，上面的解法是需要改进的。

仔细观察循环右移的特点，不难发现：每个元素右移 N 位后都会回到自己的位置上。因此，如果 $K > N$ ，右移 $K-N$ 之后的数组序列跟右移 K 位的结果是一样的。

进而可得出一条通用的规律：

右移 K 位之后的情形，跟右移 $K = K \% N$ 位之后的情形一样，如代码清单 2-34 所示。

//代码清单 2-34

```
RightShift(int* arr, int N, int K)
```

```
{  
    K % = N;  
    while(K--){  
    }
```

```

int t = arr[N - 1];
for(int i = N - 1; i > 0; i --)
    arr[i] = arr[i - 1];
arr[0] = t;
}
}

```

可见，增加考虑循环右移的特点之后，算法复杂度降为 $O(N^2)$ ，这跟 K 无关，与题目要求又接近了一步。但时间复杂度还不够低，接下来让我们继续挖掘循环右移前后，数组之间的关联。

解法二：

假设原数组序列为 abcd1234，要求变换成的数组序列为 1234abcd，即循环右移了 4 位。比较之后，不难看出，其中有两段的顺序是不变的：1234 和 abcd，可把这两段看成两个整体。右移 K 位的过程就是把数组的两部分交换一下。

变换的过程通过以下步骤完成：

逆序排列 abcd: abcd1234 → dcba1234；

逆序排列 1234: dcba1234 → dcba4321；

全部逆序: dcba4321 → 1234abcd。

伪代码可以参考清单 2-35。

//代码清单 2-35

Reverse(int* arr, int b, int e)

{

for(; b < e; b++, e--)

{

int temp = arr[e];

arr[e] = arr[b];

arr[b] = temp;

}

}

RightShift(int* arr, int N, int k)

{

K % = N;

Reverse(arr, 0, N - K - 1);

Reverse(arr, N - K, N - 1);

```
        Reverse(arr, 0, N - 1);  
    }  
}
```

这样，我们就可以在线性时间内实现右移操作了。

稍微总结下：

编程之美上，

（限制书中思路的根本原因是，题目要求：“且只允许使用两个附加变量”，去掉这个限制，思路便可如泉喷涌）

- 1、第一个想法，是一个字符一个字符的右移，所以，复杂度为 $O(N*K)$
- 2、后来，它改进了，通过这条规律：右移 K 位之后的情形，跟右移 $K = K \% N$ 位之后的情形一样

复杂度为 $O(N^2)$

- 3、直到最后，它才提出三次翻转的算法，得到线性复杂度。

下面，你将看到，本章里我们的做法是：

- 1、三次翻转，直接线性
- 2、两个指针逐步翻转，线性
- 3、stl 的 rotate 算法，线性

好的，现在，回到咱们的左旋转字符串的问题中来，对于这个左旋转字符串的问题，咱们可以如下这样考虑：

1.1、思路一：

对于这个问题，咱们换一个角度，可以这么做：

将一个字符串分成两部分，X 和 Y 两个部分，在字符串上定义反转的操作 X^T ，即把 X 的所有字符反转（如， $X="abc"$ ，那么 $X^T="cba"$ ），那么我们可以得到下面的结论：

$(X^T Y^T)^T = YX$ 。显然我们这就可以转化为字符串的反转的问题了。

不是么？就拿 abcdef 这个例子来说（[非常简短的三句，请细看，一看就懂](#)）：

- 1、首先分为俩部分，X:abc，Y:def；
- 2、 $X \rightarrow X^T$, abc->cba, $Y \rightarrow Y^T$, def->fed。
- 3、 $(X^T Y^T)^T = YX$, cbafed->defabc，即整个翻转。

我想，这下，你应该了然了。

然后，代码可以这么写（已测试正确）：

```
1. //Copyright@ 小桥流水 && July  
2. //c 代码实现，已测试正确。  
3. //http://www.smallbridge.co.cc/2011/03/13/100% E9%A2%98  
4. //_21-%E5%B7%A6%E6%97%8B%E8%BD%AC%E5%AD%97%E7%AC%A6%E4%B8%B2.html  
5. //July、updated, 2011.04.17。  
6. #include <stdio.h>  
7. #include <string.h>  
8.
```

```

9.  char * invert(char *start, char *end)
10. {
11.     char tmp, *ptmp = start;
12.     while (start != NULL && end != NULL && start < end)
13.     {
14.         tmp = *start;
15.         *start = *end;
16.         *end = tmp;
17.         start++;
18.         end--;
19.     }
20.     return ptmp;
21. }
22.
23. char *left(char *s, int pos) //pos 为要旋转的字符个数, 或长度, 下面主函数测试中, pos=3。
24. {
25.     int len = strlen(s);
26.     invert(s, s + (pos - 1)); //如上, X->X^T, 即 abc->cba
27.     invert(s + pos, s + (len - 1)); //如上, Y->Y^T, 即 def->fed
28.     invert(s, s + (len - 1)); //如上, 整个翻转, (X^TY^T)^T=YX, 即 cbafed->defabc。
29.     return s;
30. }
31.
32. int main()
33. {
34.     char s[] = "abcdefghij";
35.     puts(left(s, 3));
36.     return 0;
37. }

```

1.5.3. 在字符串中找出连续最长的数字串

功能：在字符串中找出连续最长的数字串，并把这个串的长度返回，并把这个最长数字串付给其中一个函数参数 outputstr 所指内存。例如："abcd12345ed125ss123456789"的首地址传给 inputstr 后，函数将返回 9， outputstr 所指的值为 123456789

```

#include <stdio.h>
#include <stdlib.h>
int Find maxlen(char *input,char *output)
{
    char *in = input ;
    char *out = output ;

```

```

char *temp ;
char *final ;
int count = 0 , maxlen = 0 ,i;
while(*in != '\0')
{
    if(*in>='0'&&*in<='9')
    {
        count = 0;
        for(temp = in ;*in >= '0' && *in <= '9' ; in++)
        {
            count++;
        }
        if(maxlen<count)
        {
            maxlen=count;
            final=temp;
        }
        in++;
    }
    for(i=0;i<maxlen;i++)
        *out++=*final++;
    *out='\0';
    return maxlen;
}
int main()
{
    char input[]="abc123def123456ee123456789dd";
    char output[50]={0};
    int maxlen;
    maxlen=Findmax len(input,output);
    printf("the str %s\n",output);
    printf("the maxlen is %d \n",maxlen);
    return 0;
}

```

1.5.4. 链表操作

题目：

链表操作，

(1) 单链表就地逆置，

(2) 合并链表

```
1. #include <cstdio>
2. #include <stdlib.h>
3.
4. typedef struct node
5. {
6.     struct node *next;
7.     int value;
8. }LinkList;
9.
10. void CreateList(LinkList *L);
11. void Reverse(LinkList *L);
12. void Output(LinkList *L);
13.
14. int main()
15. {
16.     LinkList *L=(LinkList*)malloc(sizeof(LinkList)); //L 为空头结点
17.     L->next=NULL;
18.
19.     CreateList(L);
20.     printf("Before reverse,the items are: \n");
21.     Output(L);
22.
23.     Reverse(L);
24.     printf("After reverse,the items are: \n");
25.     Output(L);
26.
27.     return 0;
28. }
```

```
30. void CreateList(LinkList *L) //头插法建立单链表
31. {
32.     int value;
33.     LinkList *tail=L,*newNode=NULL; //tail 为链表尾结点
34.     printf("Please input values:\n");
35.     while (scanf("%d",&value)!=EOF)
36.     {
37.         newNode=(LinkList*)malloc(sizeof(LinkList));
38.         newNode->value=value;
39.         newNode->next=NULL;
40.
41.         tail->next=newNode;
42.         tail=newNode;
43.     }
44. }
45.
46. void Reverse(LinkList *L) //把链表结点拆开，重新按尾插法建立单链表
47. {
48.     LinkList*tail=L->next,*head=NULL,*s=NULL; //tail 为链表尾部， head 为断开的链表头结点， s
        为 head 后一个结点
49.
50.     if (tail==NULL)
51.     {
52.         return;
53.     }
54.     head=tail->next;
55.     tail->next=NULL;
56.
57.     while (head!=NULL)
58.     {
59.         s=head->next;
60.         head->next=tail;
61.         L->next=head;
62.         tail=head;
63.
64.         head=s;
65.     }
66. }
67.
68. void Output(LinkList *L)
69. {
70.     LinkList *r=L->next;
71.     while (r!=NULL)
72.     {
```

```
73.     printf("%d ",r->value);
74.     r=r->next;
75.   }
76.   printf("\n");
77. }
```

解释一下逆置算法：



还有一种方法：

从一本书上截了一张图



关键代码如下：

```
[cpp] view plaincopy
1. LinkList *p=head,*q;
2. head=NULL;
3. while (p!=NULL)
4. {
5.     q=p->next;
6.     p->next=head;
7.     head=p;
8.     p=q;
9. }
```

给出完整代码：

```
[cpp] view plaincopy
1. #include <cstdio>
2. #include <stdlib.h>
3.
4. typedef struct node
5. {
6.     struct node *next;
7.     int value;
8. }LinkList;
9.
10. void CreateList(LinkList *L);
11. void Reverse(LinkList *L);
12. void Output(LinkList *L);
13.
14. int main()
```

```
15. {
16.     LinkList *L=(LinkList*)malloc(sizeof(LinkList)); //L 为空头结点
17.     L->next=NULL;
18.
19.     CreateList(L);
20.     printf("Before reverse,the items are: \n");
21.     Output(L);
22.
23.     Reverse(L);
24.     printf("After reverse,the items are: \n");
25.     Output(L);
26.
27.     return 0;
28. }
29.
30. void CreateList(LinkList *L) //头插法建立单链表
31. {
32.     int value;
33.     LinkList *tail=L,*newNode=NULL; //tail 为链表尾结点
34.     printf("Please input values:\n");
35.     while (scanf("%d",&value)!=EOF)
36.     {
37.         newNode=(LinkList*)malloc(sizeof(LinkList));
38.         newNode->value=value;
39.         newNode->next=NULL;
40.
41.         tail->next=newNode;
42.         tail=newNode;
43.     }
44. }
45.
46. void Reverse(LinkList *L) //把链表结点拆开, 重新按尾插法建立单链表
47. {
48.     LinkList*head=L->next,*p=NULL,*q=NULL; //tail 为链表尾部, head 为断开的链表头结点, s 为
49.     head 后一个结点
50.     if (head==NULL)
51.     {
52.         return;
53.     }
54.
55.     p=head;
56.     head=NULL;
57.     while (p!=NULL)
```

```

58.  {
59.      q=p->next;
60.      p->next=head;
61.      head=p;
62.
63.      if (q==NULL) //如果 p 为最后一个结点, 则把头结点指向它
64.      {
65.          L->next=p;
66.      }
67.
68.      p=q;
69.  }
70. }
71.
72. void Output(LinkList *L)
73. {
74.     LinkList *r=L->next;
75.     while (r!=NULL)
76.     {
77.         printf("%d ",r->value);
78.         r=r->next;
79.     }
80.     printf("\n");
81. }
```

1.5.5. 有 4 张红色的牌和 4 张蓝色的牌

有 4 张红色的牌和 4 张蓝色的牌，主持人先拿任意两张，再分别在 A、B、C 三人额头上贴任意两张牌，

A、B、C 三人都可以看见其余两人额头上的牌，看完后让他们猜自己额头上是什么颜色的牌，

A 说不知道，B 说不知道，C 说不知道，然后 A 说知道了。

请教如何推理，A 是怎么知道的。

如果用程序，又怎么实现呢？

思路：目的是推导出 A 的颜色，由于 A 先看 B、C，则应先假定 B、C 的颜色然后推导 A

分析：头上可能出现的牌为 bb、rr、rb（blue 、 red）

A：不知道 说明 B 、 C 中颜色相加没有等于 4 的牌

B：不知道 说明 A、C 中颜色相加没有等于 4 的牌

C：不知道 说明 A、B 中颜色相加没有等于 4 的牌

过程: 1> B:rr(bb) C:rr(bb) A 肯定知道, 所以不符合要求

2>

B:rr(bb) C:bb(rr) A 不知道, 由于 B 不知道, C 不知道 所以 A 只能取 rb

3> B:rr C: rb C 不知道->A 不是 rr A 若是 bb ->C 根

据 A、B 判断可以知道自己为 rb, 但 C 不知道, 所以排除 bb A=rb

B:bb C: rb 同理可证 A=rb

B:rb C: rr 同理可证 A=rb

B:rb C: bb 同理可证 A=rb

4> B:rb C: rb 如果 A=rr B=rb C 猜如果自

己=bb 则 B 不可能知道自己为 rb 所以 C 可以猜到 C=rb

如果 A=bb B=rb 同理 C

可以猜到自己=rb

由于 A 排除了 rr bb 所以只能

取 A=rb

结论: 不能同时存在 rr 和 bb, A 不能是 rr 和 bb

1.5.6. 输入两个整数 n 和 m, 从数列 1, 2, 3.....n 中 随意取几个数

输入两个整数 n 和 m, 从数列 1, 2, 3.....n 中 随意取几个数,
使其和等于 m, 要求将其中所有的可能组合列出来.

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.   * 输入两个整数 n 和 m, 从数列 1, 2, 3.....n 中随意取几个数,
3.   * 使其和等于 m, 要求将其中所有的可能组合列出来.
4.
5.   e.g
6.   n=6,m=6  1,2,3  2,4  1,5
7.   n=
8.   * @author wangxm
9.   */
10. public class Comp {
11.     static void getAllComp(int n,int m){
12.         String pre = m+"=";
13.         int theMax = (1+n)*n/2;
14.         if(theMax<m){
15.             System.out.println("不存在该数! ");
16.         }else{
17.             for(int i=1;i<=m/2;i++){
18.                 //从 1 开始计数, 打印出两个数的组合, 并且两数不相等
19.                 if(i != m-i && n>=m-i)
20.                     System.out.println(pre+i+"+"+(m-i));
21.                 //调用递归, 继续求得大于 2 个数的组合
22.                 getResult(m-i,pre+i,i,n);
23.             }
24.         }
25.     }
26.     //调用递归, 继续求得大于 2 个数的组合,j 为组合中已用过的数, 所以取大于该数的。
27.     static void getResult(int m,String pre,int j,int n){
28.         for(int i=j+1;i<=m/2;i++){
29.             if(i != m-i && n>=m-i)
30.                 System.out.println(pre+"+"+i+"+"+(m-i));
31.             getResult(m-i,pre+"+"+i,i,n);
32.         }
33.     }
34.
35.     public static void main(String[] args) {
```

```
36.     getAllComp(6,10);  
37. }  
38. }
```

1.5.7. 输入一个表示整数的字符串，把该字符串转换成整数并输出

题目：输入一个表示整数的字符串，把该字符串转换成整数并输出。例如输入字符串"345"，则输出整数 345。分析：这道题尽管不是很难，学过 C/C++语言一般都能实现基本功能，但不同程序员就这道题写出的代码有很大区别，可以说这道题能够很好地反应出程序员的思维和编程习惯，因此已经被包括微软在内的多家公司用作面试题。建议读者在往下看之前自己先编写代码，再比较自己写的代码和下面的参考代码有哪些不同。

首先我们分析如何完成基本功能，即如何把表示整数的字符串正确地转换成整数。还是以"345"作为例子。当我们扫描到字符串的第一个字符'3'时，我们不知道后面还有多少位，仅仅知道这是第一位，因此此时得到的数字是 3。当扫描到第二个数字'4'时，此时我们已经知道前面已经一个 3 了，再在后面加上一个数字 4，那前面的 3 相当于 30，因此得到的数字是 $3*10+4=34$ 。接着我们又扫描到字符'5'，我们已经知道了'5'的前面已经有了 34，由于后面要加上一个 5，前面的 34 就相当于 340 了，因此得到的数字就是 $34*10+5=345$ 。

分析到这里，我们不能得出一个转换的思路：每扫描到一个字符，我们把在之前得到的数字乘以 10 再加上当前字符表示的数字。这个思路用循环不难实现。

由于整数可能不仅仅之含有数字，还有可能以'+'或者'-'开头，表示整数的正负。因此我们需要把这个字符串的第一个字符做特殊处理。如果第一个字符是'+'号，则不需要做任何操作；如果第一个字符是'-'号，则表明这个整数是个负数，在最后的时候我们要把得到的数值变成负数。

接着我们试着处理非法输入。由于输入的是指针，在使用指针之前，我们要做的第一件是判断这个指针是不是为空。如果试着去访问空指针，将不可避免地导致程序崩溃。另外，输入的字符串中可能含有不是数字的字符。每当碰到这些非法的字符，我们就没有必要再继续转换。最后一个需要考虑的问题是溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出。

现在已经分析的差不多了，开始考虑编写代码。首先我们考虑如何声明这个函数。由于是把字符串转换成整数，很自然我们想到：

```
int StrToInt(const char* str);
```

这样声明看起来没有问题。但当输入的字符串是一个空指针或者含有非法的字符时，应该返回什么值呢？0 怎么样？那怎么区分非法输入和字符串本身就是"0"这两种情况呢？

接下来我们考虑另外一种思路。我们可以返回一个布尔值来指示输入是否有效，而把转换后

的整数放到参数列表中以引用或者指针的形式传入。于是我们就可以声明如下：

```
bool StrToInt(const char *str, int& num);
```

这种思路解决了前面的问题。但是这个函数的用户使用这个函数的时候会觉得不是很方便，因为他不能直接把得到的整数赋值给其他整形变量，显得不够直观。

前面的第一种声明就很直观。如何在保证直观的前提下当碰到非法输入的时候通知用户呢？一种解决方案就是定义一个全局变量，每当碰到非法输入的时候，就标记该全局变量。用户在调用这个函数之后，就可以检验该全局变量来判断转换是不是成功。

下面我们写出完整的实现代码。参考代码：

```
enum Status {kValid = 0, kInvalid};
```

```
int g_nStatus = kValid;
```

```
////////////////////////////////////////////////////////////////////////
```

```
// Convert a string into an integer
```

```
////////////////////////////////////////////////////////////////////////
```

```
int StrToInt(const char* str)
```

```
{
```

```
    g_nStatus = kInvalid;
```

```
    longlong num = 0;
```

```
    if(str != NULL)
```

```
{
```

```
        const char* digit = str;
```

```
        // the first char in the string maybe '+' or '-'
```

```
        bool minus = false;
```

```
        if(*digit == '+')
```

```
            digit ++;
```

```
        elseif(*digit == '-')
```

```
{
```

```
            digit ++;
```

```
            minus = true;
```

```
}
```

```
        // the remaining chars in the string
```

```
        while(*digit != '\0')
```

```

{
    if(*digit >= '0' && *digit <= '9')
    {
        num = num * 10 + (*digit - '0');

        // overflow
        if(num > std::numeric_limits<int>::max())
        {
            num = 0;
            break;
        }

        digit++;
    }

    // if the char is not a digit, invalid input
    else
    {
        num = 0;
        break;
    }
}

if(*digit == '/0')
{
    g_nStatus = kValid;
    if(minus)
        num = 0 - num;
}
}

return static_cast<int>(num);
}

```

讨论：在参考代码中，我选用的是第一种声明方式。不过在面试时，我们可以选用任意一种声明方式进行实现。但当面试官问我们选择的理由时，我们要对两者的优缺点进行评价。第一种声明

方式对用户而言非常直观，但使用了全局变量，不够优雅；而第二种思路是用返回值来表明输入是否合法，在很多 API 中都用这种方法，但该方法声明的函数使用起来不够直观。

最后值得一提的是，在 C 语言提供的库函数中，函数 atoi 能够把字符串转换整数。它的声明是 int atoi(**constchar** *str)。该函数就是用一个全局变量来标志输入是否合法的。

1.5.8. 给出一个数列，找出其中最长的单调递减（或递增）子序列

问题描述：给出一个数列，找出其中最长的单调递减（或递增）子序列。

解题思路：动态规划。假设 0 到 $i-1$ 这段数列的最长递减序列的长度为 s ，且这些序列们的末尾值中的最大值是 t 。对于 $a[i]$ 有以下情况：

- (1) 如果 $a[i]$ 比 t 小，那么将 $a[i]$ 加入任何一个子序列都会使 0 到 i 的最长单调序列长度变成 $s+1$ ，这样的话，在 0 到 i 的数列中，长度为 $s+1$ 的递减子序列的末尾值最大值就是 $a[i]$ ；
- (2) 如果 $a[i]$ 和 t 相等，那么说明数列从 0 项到 i 项的最长单调子序列长度就是 s ；
- (3) 如果 $a[i]$ 比 t 大，那么 $a[i]$ 就不一定能够成为长度为 s 的递减子序列的末项，这取决于长度为 $s-1$ 的各个递减子序列的末尾值的最大值 t' 。

如果 t' 比 $a[i]$ 要大，那么就可以形成长度为 s 的递减子序列，如果 t' 比 $a[i]$ 小，那么问题就在往前递推，把 $a[i]$ 和长度为 $s-2$ 的各个递减子序列的末尾值的最大值比较，直到：(1) $a[i]$ 比长度为 s' 的递减子序列的末尾值的最大值要小，那么 $a[i]$ 就是数列 0 到 i 部分长度为 $s'+1$ 的递减子序列的末尾值中的最大值；(2) $a[i]$ 比任何长度的递减子序列的末尾值的最大值都要大，那么 $a[i]$ 就是长度为 1 的递减子序列的最大值。

所以，引入数组 $c[i]$ 表示长度为 i 的递减子序列的末尾值的最大值。显然 c 数组必然是单调递减的。 $b[i]$ 数组用于子序列的输出， $b[i]$ 表示从 $a[0]$ 到 $a[i]$ 且终止于 $a[i]$ 的最长递减序列的长度。

算法复杂度： $O(n \log n)$ ，对于数组 c 的查找使用二分查找，降低了整体的算法复杂度。

算法步骤：

- 1) 读入 n 和 $a[i]$ 。
- 2) 将数组 c 全部赋值为 -1。
- 3) 定义变量 s ，初始化为 1， s 表示目前为止最长单调序列的长度，同时也是数组 C 的有效容量。 $c[1] = a[0]$ 。
- 4) 对于 0 到 $n-1$ 的每个 i ：

查找 $c[1]$ 到 $c[s]$ ，找到一个值 k 满足下列几种情况：

- (1) $c[k] \leq a[i]$ 而 $c[k-1] > a[i]$ （如果 $k > 1$ ）
- (2) 找不到 (1) 中 k 的话， k 等于 $s+1$ ，并且 s 自加一。

$c[k] = a[i]$ ；

$b[i] = k$ ；

5) 最后所得 s 即为所求值。

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. import java.util.Scanner;
2.
3. public class LongestSubSeq {
4.
5.     public static int bsearch(int[] a, int s, int m) {
6.         int low = 1;
7.         int high = s;
8.         int mid;
9.
10.        while (low < high) {
11.            mid = (low + high) / 2;
12.            if (a[mid] == m )
13.                return mid;
14.            if (a[mid] > m)
15.                low = mid + 1;
16.            else
17.                high = mid;
18.        }
19.        if (a[low] <= m)
20.            return low;
21.        else
22.            return low+1;
23.    }
24.
25.    public static void print(int[] a, int[] b, int level, int start) {
26.        if (level == 0)
27.            return;
28.        int i = start;
29.        while (b[i] != level)
30.            i--;
31.        print(a, b, level-1, i-1);
32.        System.out.print(a[i] + " ");
```

```
33.    }
34.
35.    public static void main(String[] args) {
36.        Scanner in = new Scanner(System.in);
37.        int n = in.nextInt();
38.
39.        int[] array = new int[n];
40.        int[] b = new int[n];
41.        int[] c = new int[n+1];
42.
43.        for (int i = 0; i < n; i++) {
44.            array[i] = in.nextInt();
45.            c[i] = -1;
46.        }
47.
48.        int s = 1;
49.        int k;
50.        c[1] = array[0];
51.
52.        for (int i = 0; i < n; i++) {
53.            k = bsearch(c, s, array[i]);
54.            if(k > s)
55.                s++;
56.            c[k] = array[i];
57.            b[i] = k;
58.        }
59.
60.        System.out.println("The length of longest sequence is: " + s);
61.        System.out.print("The sequence is: ");
62.        print(array, b, s, n-1);
63.    }
64. }
```

1.5.9. 四对括号可以有多少种匹配排列方式

四对括号可以有多少种匹配排列方式？比如两对括号可以有两种：() () 和 (())

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. #include<iostream>
2. #include<cassert>
3. #include <vector>
4. using namespace std ;
5. void Print(vector<char> v)
6. {
7.     for (vector<char>::iterator beg=v.begin();beg!=v.end();++beg)
8.         cout<<*beg<<" ";
9.     cout<<endl;
10. }
11. void MatchNums(int nSize,int nLen,vector<char> &v)
12. {
13.     int nLeftBrackets=0;
14.     int nRightBrackets=0;
15.     for (vector<char>::iterator beg=v.begin();beg!=v.end();++beg)
16.     {
17.         if(*beg=='(')
18.             nLeftBrackets++;
19.         else
20.             nRightBrackets++;
21.         if(nRightBrackets>nLeftBrackets)
22.             return;
23.         if(nLeftBrackets+nRightBrackets==nSize&&nLeftBrackets==nRightBrackets)
24.             Print(v);
25.     }
26.
27.     if (nLen>0)
28.     {
29.         v.push_back(')');
30.         MatchNums(nSize,nLen-1,v);
31.         v.pop_back();
32.     }
33. }
```

```
32.     v.push_back(')');
33.     MatchNums(nSize,nLen-1,v);
34.     v.pop_back();
35. }
36. }
37. int main()
38. {
39.     vector<char> v;
40.     int n=6;
41.     MatchNums(n,n,v);
42.     return 1;
43. }
```

1.5.10. 输入一个正数 n，输出所有和为 n 连续正数序列

题目：输入一个正数 n，输出所有和为 n 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：这是网易的一道面试题。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. //和为 n 连续正数序列
2. #include<iostream>
3.
4.
5. using namespace std;
6. void print_sq(int start,int end)
7. {
    //打印序列
8.     for(int i=start;i<=end;i++)
9.         cout<<i<<" ";
10.    cout<<endl; }
11.
12. void find_the_sq(int num){
13.     int start=1,end=2,mid=num/2;
```

```

14. int sum=(start+end);
15. while(start<=mid){
16. //检索，若是 sum 大了，则 start 右移，若是小了则 end 右移
17. if(sum<num)
18. sum+=++end;
19. else if(sum>num)
20. sum-=start++;
21. else{
22. print_sq(start,end);
23. sum-=start++;
24. }
25. }
26. }
27. int main(void){ find_the_sq(25); return 0; }

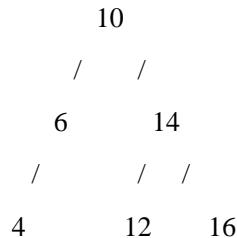
```

1.6. 面试题集合（五）

1.6.1. 输入一棵二元树的根结点，求该树的深度

题目：输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树：



输出该树的深度 3。

二元树的结点定义如下：

```

struct SBinaryTreeNode // a node of the binary tree
{
    int             m_nValue; // value of node
    SBinaryTreeNode *m_pLeft; // left child of node
    SBinaryTreeNode *m_pRight; // right child of node
}
  
```

```
};
```

分析：这道题本质上还是考查二元树的遍历。

题目给出了一种树的深度的定义。当然，我们可以按照这种定义去得到树的所有路径，也就能得到最长路径以及它的长度。只是这种思路用来写程序有点麻烦。

我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个结点，它的深度为 1。如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加 1；同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加 1。如果既有右子树又有左子树呢？那该树的深度就是其左、右子树深度的较大值再加 1。

上面的这个思路用递归的方法很容易实现，只需要对遍历的代码稍作修改即可。

参考代码如下：

```
//////////  
//  
// Get depth of a binary tree  
// Input: pTreeNode - the head of a binary tree  
// Output: the depth of a binary tree  
/////////  
//  
int TreeDepth(SBinaryTreeNode *pTreeNode)  
{  
    // the depth of a empty tree is 0  
    if(!pTreeNode)  
        return 0;  
  
    // the depth of left sub-tree  
    int nLeft = TreeDepth(pTreeNode->m_pLeft);  
    // the depth of right sub-tree  
    int nRight = TreeDepth(pTreeNode->m_pRight);  
  
    // depth is the binary tree  
    return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);  
}
```

1.6.2. 输入一个字符串，打印出该字符串中字符的所有排列

题目：输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串 abc，则输出由字符 a、b、c 所能排列出来的所有字符串 abc、acb、bac、bca、cab 和 cba。

分析：这是一道很好的考查对递归理解的编程题，因此在过去一年中频繁出现在各大公司的面试、笔试题中。

我们以三个字符 abc 为例来分析一下求字符串排列的过程。首先我们固定第一个字符 a，求后面两个字符 bc 的排列。当两个字符 bc 的排列求好之后，我们把第一个字符 a 和后面的 b 交换，得到 bac，接着我们固定第一个字符 b，求后面两个字符 ac 的排列。现在是把 c 放到第一位的时候了。记住前面我们已经把原先的第一个字符 a 和后面的 b 做了交换，为了保证这次 c 仍然是和原先处在第一位的 a 交换，我们在拿 c 和第一个字符交换之前，先要把 b 和 a 交换回来。在交换 b 和 a 之后，再拿 c 和处在第一位的 a 进行交换，得到 cba。我们再次固定第一个字符 c，求后面两个字符 b、a 的排列。

既然我们已经知道怎么求三个字符的排列，那么固定第一个字符之后求后面两个字符的排列，就是典型的递归思路了。

基于前面的分析，我们可以得到如下的参考代码：

```
void Permutation(char* pStr, char* pBegin);
///////////////////////////////
///
// Get the permutation of a string,
// for example, input string abc, its permutation is
// abc acb bac bca cba cab
///////////////////////////////
///
void Permutation(char* pStr)
{
    Permutation(pStr, pStr);
}

///////////////////////////////
///
// Print the permutation of a string,
// Input: pStr - input string
```

```

//      pBegin - points to the begin char of string
//              which we want to permute in this recursion
///////////////////////////////
///

void Permutation(char* pStr, char* pBegin)
{
    if(!pStr || !pBegin)
        return;

    // if pBegin points to the end of string,
    // this round of permutation is finished,
    // print the permuted string
    if(*pBegin == '\0')
    {
        printf("%s\n", pStr);
    }
    // otherwise, permute string
    else
    {
        for(char* pCh = pBegin; *pCh != '\0'; ++ pCh)
        {
            // swap pCh and pBegin
            char temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
            Permutation(pStr, pBegin + 1);
            // restore pCh and pBegin
            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
        }
    }
}

```

扩展 1：如果不是求字符的所有排列，而是求字符的所有组合，应该怎么办呢？

当输入的字符串中含有相同的字符串时，相同的字符交换位移是不同的排列，但

是同一个组合。举个例子，如果输入 aaa，那么它的排列是 6 个 aaa，但对应的组合只有一个。

扩展 2：输入一个含有 8 个数字的数组，判断有没有可能把这 8 个数字分别放到正方体的 8 个顶点上，使得正方体上三组相对的面上的 4 个顶点的和相等。

1.6.3. 输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分

题目：输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。要求时间复杂度为 $O(n)$ 。

分析：如果不考虑时间复杂度，最简单的思路应该是从头扫描这个数组，每碰到一个偶数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于碰到一个偶数，需要移动 $O(n)$ 个数字，因此总的时间复杂度是 $O(n^2)$ 。

要求的是把奇数放在数组的前半部分，偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候，如果发现有偶数出现在奇数的前面，我们可以交换他们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化为数组的第一个数字，它只向 后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶 数而第二个指针指向的数字是奇数，我们就交换这两个数字。

基于这个思路，我们可以写出如下的代码：

```
void Reorder(int *pData, unsigned int length, bool (*func)(int));
bool isEven(int n);

///////////
// Devide an array of integers into two parts, odd in the first part,
// and even in the second part
// Input: pData - an array of integers
// length - the length of array
///////////

void ReorderOddEven(int *pData, unsigned int length)
{
    if(pData == NULL || length == 0)
        return;
```

```

Reorder(pData, length, isEven);
}

///////////
// Devide an array of integers into two parts, the intergers which
// satisfy func in the first part, otherwise in the second part
// Input: pData - an array of integers
// length - the length of array
// func - a function
///////////

void Reorder(int *pData, unsigned int length, bool (*func)(int))
{
    if(pData == NULL || length == 0)
        return;

    int *pBegin = pData;
    int *pEnd = pData + length - 1;

    while(pBegin < pEnd)
    {
        // if *pBegin does not satisfy func, move forward
        if(!func(*pBegin))
        {
            pBegin++;
            continue;
        }

        // if *pEnd does not satisfy func, move backward
        if(func(*pEnd))
        {
            pEnd--;
            continue;
        }
    }
}

```

```
// if *pBegin satisfy func while *pEnd does not,  
// swap these integers  
  
int temp = *pBegin;  
*pBegin = *pEnd;  
*pEnd = temp;  
}  
}
```

```
//////////
```

```
// Determine whether an integer is even or not
```

```
// Input: an integer  
// otherwise return false
```

```
//////////
```

```
bool isEven(int n)
```

```
{  
    return (n & 1) == 0;  
}
```

讨论：

上面的代码有三点值得提出来和大家讨论：

- 1 . 函数 isEven 判断一个数字是不是偶数并没有用%运算符而是用&。理由是通常情况下位运算符比%要快一些；
- 2 . 这道题有很多变种。这里要求是把奇数放在偶数的前面，如果把要求改成：把负数放在非负数的前面等，思路都是都一样的。
- 3 . 在函数 Reorder 中，用函数指针 func 指向的函数来判断一个数字是不是符合给定的条件，而不是用在代码直接判断（hard code）。这样的好处是把调整顺序的算法和调整的标准分开了（即解耦，decouple）。当调整的标准改变时，Reorder 的代码不需要修改，只需要提供一个新的确定调整标准的函数即可，提高了代码的可维护性。例如要求把负数放在非负数的前面，我们不需要修改 Reorder 的代码，只需添加一个函数来判断整数是不是非负数。这样的思路在很多库中都有广泛的应用，比如在 S T L 的很多算法函数中都有一个仿函数（functor）的参数（当然仿函数不是函数指针，但其思想是一样的）。如果在面试中能够想到这一层，无疑能给面试官留下很好的印象。

1.6.4. 给定链表的头指针和一个结点指针，在 O(1)时间删除该结点

题目：给定链表的头指针和一个结点指针，在 O(1)时间删除该结点。

链表结点的定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

函数的声明如下：

```
void DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted);
```

分析：这是一道广为流传的 Google 面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。由于需要顺序查找，时间复杂度自然就是 $O(n)$ 了。

我们之所以需要从头结点开始查找要删除的结点，是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个结点。这个时候我们实际删除的是它的下一个结点，由于我们已经得到实际删除的结点的前面一个结点，因此完全是可以实现的。当然，在删除之前，我们需要把给定的结点的下一个结点的数据拷贝到给定的结点中。此时，时间复杂度为 $O(1)$ 。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是 $O(n)$ 。

那题目要求我们需要在 $O(1)$ 时间完成删除操作，我们的算法是不是不符合要求？实际上，假设链表总共有 n 个结点，我们的算法在 $n-1$ 总情况下时间复杂度是 $O(1)$ ，只有当给定的结点处于链表末尾的时候，时间复杂度为 $O(n)$ 。那么平均时间复杂度 $[(n-1)*O(1)+O(n)]/n$ ，仍然为 $O(1)$ 。

基于前面的分析，我们不难写出下面的代码。

参考代码：

```
///////////
// 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。

1.6.5. 输入一个链表的头结点，从尾到头反到来输出每个结点的值

题目：输入一个链表的头结点，从尾到头反到来输出每个结点的值。链表结点定义如下：

```

struct ListNode
{
    int m_nKey;
    ListNode* m_pNext;
};

```

这道题难度该当不大，看能想出几种做法，另外即便要当心这个题目的变体，原文中有讲到。

我能想到的思路有三种，和原文差不多。

措施一、先把链表反向，然后再过去到后遍历一遍。

措施二、设一个栈，过去到后遍历顺次，把结点值压入栈中，再出栈打印。

措施三、递归。

题目的变体在原文中有讲到两种：

1. 从尾到头输出一个字符串；
2. 定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。

===== 以下内容引自原文 =====

分析：这是一道很含蓄的面试题。该题以及它的变体经常出现在各大公司的面试、笔试题中。

看到这道题后，第一反应是过去到后输出比拟容易。于是很慷慨地想到把链表中链接结点的指针反转过来，改换链表的方向。然后就能够过去到后输出了。反转链表的算法详见本人面试题精选系列的第 19 题，在此不再细述。但该措施必需额外的垄断，该当还有更好的措施。接下来的设想是过去到后遍历链表，每穿越一个结点的时候，把该结点放到一个栈中。当遍历彻底端链表后，再从栈顶开始输出结点的值，此刻输出的结点的次序曾经反转过来了。该措施必需维护一个额外的栈，告终起来比拟繁琐。

既然想到了栈来告终这个函数，而递归性质上即便一个栈构造。于是很慷慨的又想到了用递归来告终。要告终反来到输出链表，我们每拜会到一个结点的时候，先递归输出它后面的结点，再输出该结点本身，这么链表的输出收获就反过来了。

基于这么的思路，不难写出如下代码：

```
//////////  
// Print a list from end to beginning  
// Input: pListHead - the head of list  
/////////  
void PrintListReversely(ListNode* pListHead)  
{  
    if(pListHead != NULL)  
    {  
        // Print the next node first  
        if (pListHead->m_pNext != NULL)  
        {  
            PrintListReversely(pListHead->m_pNext);  
        }  
        // Print this node  
        printf("%d", easygle.com pListHead->m_nKey);  
    }  
}
```

```
    }
}
```

伸展：该题还有两个常见的变体：

1. 从尾到头输出一个字符串；
2. 定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。

1.6.6. 用 C++设计一个不能被继承的类

分析：这是 Adobe 公司 2007 年校园招聘的最新笔试题。这道题除了考察应聘者的 C++ 基本功底外，还能考察反应能力，是一道很好的题目。

在 Java 中定义了关键字 final，被 final 修饰的类不能被继承。但在 C++ 中没有 final 这个关键字，要实现这个要求还是需要花费一些精力。

首先想到的是在 C++ 中，子类的构造函数会自动调用父类的构造函数。同样，子类的析构函数也会自动调用父类的析构函数。要想一个类不能被继承，我们只要把它的构造函数和析构函数都定义为私有函数。那么当一个类试图从它那继承的时候，必然会由于试图调用构造函数、析构函数而导致编译错误。

可是这个类的构造函数和析构函数都是私有函数了，我们怎样才能得到该类的实例呢？这难不倒我们，我们可以通过定义静态来创建和释放类的实例。基于这个思路，我们可以写出如下的代码：

```
////////////////////////////////////////////////////////////////
// Define a class which can't be derived from
////////////////////////////////////////////////////////////////

class FinalClass1
{
public :
    static FinalClass1* GetInstance()
    {
        return new FinalClass1;
    }

    static void DeleteInstance( FinalClass1* pInstance)
    {
        delete pInstance;
        pInstance = 0;
    }

private :
    FinalClass1() {}
    ~FinalClass1() {}
```

```
};
```

这个类是不能被继承，但在总觉得它和一般的类有些不一样，使用起来也有点不方便。比如，我们只能得到位于堆上的实例，而得不到位于栈上实例。

能不能实现一个和一般类除了不能被继承之外其他用法都一样的类呢？办法总是有的，不过需要一些技巧。请看如下代码：

```
////////////////////////////////////////////////////////////////
```

```
// Define a class which can't be derived from
```

```
////////////////////////////////////////////////////////////////
```

```
template <typename T> class MakeFinal
```

```
{
```

```
    friend T;
```

```
private :
```

```
    MakeFinal() {}
```

```
    ~MakeFinal() {}
```

```
};
```

```
class FinalClass2 : virtual public MakeFinal<FinalClass2>
```

```
{
```

```
public :
```

```
    FinalClass2() {}
```

```
    ~FinalClass2() {}
```

```
};
```

这个类使用起来和一般的类没有区别，可以在栈上、也可以在堆上创建实例。尽管类 MakeFinal <FinalClass2> 的构造函数和析构函数都是私有的，但由于类 FinalClass2 是它的友元函数，因此在 FinalClass2 中调用 MakeFinal <FinalClass2> 的构造函数和析构函数都不会造成编译错误。

但当我们试图从 FinalClass2 继承一个类并创建它的实例时，却不同通过编译。

```
class Try : public FinalClass2
```

```
{
```

```
public :
```

```
    Try() {}
```

```
    ~Try() {}
```

```
};
```

```
Try temp;
```

由于类 FinalClass2 是从类 MakeFinal <FinalClass2> 虚继承过来的，在调用 Try 的构造函数的时候，会直接跳过 FinalClass2 而直接调用 MakeFinal <FinalClass2> 的构造函数。非常遗憾的是，Try 不是 MakeFinal <FinalClass2> 的友元，因此不能调用其私有的构造函数。

基于上面的分析，试图从 FinalClass2 继承的类，一旦实例化，都会导致编译错误，因此是 FinalClass2 不能被继承。这就满足了我们设计要求。

1.6.7. 给定链表的头指针和一个结点指针，在 O(1)时间删除该结点

题目：给定链表的头指针和一个结点指针，在 O(1)时间删除该结点。链表结点的定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

函数的声明如下：

```
void DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted);
```

这个题我的想法是这样的：

传统的做法是，从头结点开始，一直找到要删除结点的前趋结点，然后前趋结点的 next 指针直接指向要删除结点的后继，再 free 掉要删除的结点就 OK 了。当然，这个方法的时间复杂度是 O(n)。

要在 O(1)时间内删除某个结点，所以不能用传统的做法从头结点开始遍历链表找到要删除结点的前趋，而从要删除的结点出发也找不到其自身的前趋，这样要改变一种思路，就是不找前趋！

具体做法是这样的，把头结点的数据直接 copy 到要删除的结点处，然后头指针向后移动一个结点，再 free 掉原来的头指针指向的结点，这样等于把要删除的结点删除了。当链表只有一个结点或者要删除的结点是头结点或尾结点时，这种方法也是成立的，所以不需要做特殊的处理。大概的代码如下，没有测试过。

1. **void** DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted){
2. **if**(NULL==pListHead||NULL==pToBeDeleted)
3. **return**;
- 4.
5. pToBeDeleted->m_nKey = pListHead->m_nKey;
- 6.
7. pToBeDeleted = pListHead;
8. pListHead = pListHead->m_pNext;
- 9.
10. free(pToBeDeleted);
11. }

另外，原文中思路有一点不一样，原文作者还是想找到前趋，然后再删除结点。他的做法是，把要删除结点的后继结点的数据 copy 到要删除结点中，则后继结点就成了要删除的结点了，那么要删除的结点就变成真正要删除结点的前趋了，然后完成删除。不过这种思路要考虑到待删除结点是尾结点的特殊情况，因为尾结点是没有后继的。

两种方法比较起来，我觉得第一种是比较方便的，但是第二种方法可以保证原来链表的数据排序（如果需要的话）。各有好处吧。

下面就引用原文作者的做法。

===== 以下内容引自原文=====

分析：这是一道广为流传的 Google 面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。

由于需要顺序查找，时间复杂度自然就是 $O(n)$ 了。

我们之所以需要从头结点开始查找要删除的结点，是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个结点。这个时候我们实际删除的是它的下一个结点，由于我们已经得到实际删除的结点的前面一个结点，因此完全是可以实现的。当然，在删除之前，我们需要需要把给定的结点的下一个结点的数据拷贝到给定的结点中。此时，时间复杂度为 $O(1)$ 。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是 $O(n)$ 。

那题目要求我们需要在 $O(1)$ 时间完成删除操作，我们的算法是不是不符合要求？实际上，假设链表总共有 n 个结点，我们的算法在 $n-1$ 总情况下时间复杂度是 $O(1)$ ，只有当给定的结点处于链表末尾的时候，时间复杂度为 $O(n)$ 。那么平均时间复杂度 $[(n-1)*O(1)+O(n)]/n$ ，仍然为 $O(1)$ 。

基于前面的分析，我们不难写出下面的代码。

参考代码：

```
//////////  
// Delete a node in a list  
// Input: pListHead - the head of list  
//         pToDelete - the node to be deleted  
/////////  
  
void DeleteNode(ListNode* pListHead, ListNode* pToDelete)  
{  
    if(!pListHead || !pToDelete)  
        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。

1.6.8. 一个数组中除了两个数字之外，其余数字均出现了两次

问题描述：一个数组中除了两个数字之外，其余数字均出现了两次(或偶数次)。请写出程序查找出这两个只出现一次的数字，要求时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

问题分析：这是一道很新颖的关于位运算的面试题。首先考虑这个问题的一个简单版本：一个数组中除了一个数字之外，其余的数字均出现两次，请写程序找出这个出现一次的数字。这个问题的突破口在哪里？题目为什么强调有一个数字只出现一次，其他的出现两次？我们想到异或运算的性质，任何一个数字异或自身都为 0。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现一次的数字。因为出现两次的数字均在异或运算中抵消了。

有了上面简单问题的解决方案后，我们回到原始问题。如果能把一个数组分为两个子数组，在每个子数组中，包含一个只出现一次的数字，其余数字均出现两次。如果能够这样拆分数组，则按照前面的方法，很容易求出原始数组中两个只出现一次的数字。

我们还是从头到尾依次异或数组中每个数字，那么最终得到的结果就是两个只出现一次的数字的异或值，其他出现两次的数字在异或运算中抵消了。由于这两个数字肯定不一样，所以最终异或结果必然不为 0。也就是说异或结果的二进制数值中肯定有一位为 1。我们在异或结果数字中找到第一个不为 1 的位的位置，记为 N，我们以第 N 位是否为 1 为标准把原始数组划分为两个子数组。第一个子数组每个数字第 N 位都为 1，第二个子数组每个数字第 N 位都为 0。现在，我们已经把原始数组划分为两个子数组。两个子数组都包含一个只出现 1 次的数字，而其余数字均出现两次。到此为止，原始问题就得以解决了。

```
#include<iostream>
using namespace std;
int findFirstOne(int value);
bool testBit(int value,int pos);
int findNums(int date[],int length,int &num1,int &num2){
    if(length<2){return -1;}
    int ansXor=0;
    for(int i=0;i<length;i++){
        ansXor^=date[i];           //异或
    }
    int pos=findFirstOne(ansXor);
    num1=num2=0;
    for(int i=0;i<length;i++){
        if(testBit(date[i],pos))
```

```

        num1^=date[i];
    else
        num2^=date[i];
    }
    return 0;
}

int findFirstOne(int value){ //取二进制中首个为 1 的位置
    int pos=1;
    while((value&1)!=1){
        value=value>>1;
        pos++;
    }
    return pos;
}

bool testBit(int value,int pos){ //测试某位置是否为 1
    return ((value>>pos)&1);
}

int main(void){
    int date[10]={1,2,3,4,5,6,4,3,2,1};
    int ans1,ans2;
    if(findNums(date,10,ans1,ans2)==0)
        cout<<ans1<<" "<<ans2<<endl;
    else
        cout<<"error"<<endl;
    return 0;
}

```

1.6.9. 两个单向链表，找出它们的第一个公共结点

题目：两个单向链表，找出它们的第一个公共结点。

链表的结点定义为：

```

struct ListNode
{
    int      m_nKey ;
    ListNode *  m_pNext ;

```

};

分析：这是一道微软的面试题。微软非常喜欢与链表相关的题目，因此在微软的面试题中，链表出现的概率相当高。

如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的 `m_pNext` 都指向同一个结点。但由于是单向链表的结点，每个结点只有一个 `m_pNext`，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个 Y，而不可能像 X。

看到这个题目，第一反应就是蛮力法：在第一链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为 m ，第二个链表的长度为 n ，显然，该方法的时间复杂度为 $O(mn)$ 。

接下来我们试着去寻找一个线性时间复杂度的算法。我们先把问题简化：如何判断两个单向链表有没有公共结点？前面已经提到，如果两个链表有一个公共结点，那么该公共结点之后的所有结点都是重合的。那么，它们的最后一个结点必然是重合的。因此，我们判断两个链表是不是有重合的部分，只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的，说明它们用重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长 1 个结点，我们先在长的链表上遍历 1 个结点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点考试到链表的尾结点，这一部分是重合的。因此，它们肯定也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为 m ，第二个链表的长度为 n ，该方法的时间复杂度为 $O(m+n)$ 。

[java] [view plain](#) [copy](#) [print](#)?

```
1. package structure.list;
2.
3. import structure.list.node.LNode_01;
4.
5. /**
6.  * 题目：两个单向链表，找出它们的第一个公共结点
7. *
```

```
8.     * @author Toy
9.     *
10.    */
11.   public class First_CommonNode {
12.
13.     LNode_01 head1 = null;
14.     LNode_01 head2 = null;
15.     LNode_01 head = null;
16.
17.     public LNode_01 method_01() {
18.
19.         init();
20.
21.         return getFirstCommonNode_01(head1, head2);
22.     }
23.
24.     /**
25.      * 逐个考察 O(m*n)
26.      *
27.      * @param head1
28.      * @param head2
29.      * @return
30.      */
31.     private LNode_01 getFirstCommonNode_01(LNode_01 head1, LNode_01 head2) {
32.
33.     LNode_01 p = head1;
34.     LNode_01 q = head2;
35.
36.     while (p != null) {
37.         q = head2;
38.         while (q != null) {
39.             if (q == p) {
40.                 return p;
41.             }
42.             q = q.next;

```

```
43.         }
44.         p = p.next;
45.     }
46.     return null;
47. }
48.
49. public LNode_01 method_02() {
50.     return getFirstCommonNode_02(head1, head2);
51. }
52.
53. /**
54. * 先找到长度差 K，然后长的先遍历 K 步,二者再同步遍历
55. *
56. * @param head1
57. * @param head2
58. * @return
59. */
60. private LNode_01 getFirstCommonNode_02(LNode_01 head1, LNode_01 head2) {
61.     int len1 = LinkList_01.getLength(head1);
62.     int len2 = LinkList_01.getLength(head2);
63.     int delta = Math.abs(len1 - len2);
64.
65.     LNode_01 first = head1;
66.     LNode_01 follow = head2;
67.     if (len2 > len1) {
68.         first = head2;
69.         follow = head1;
70.     }
71.     first = LinkList_01.toNodeK(first, delta + 1);
72.
73.     while (follow != null && first != null) {
74.         if (follow == first) {
75.             return follow;
76.         }
77.         follow = follow.next;
```

```
78.         first = first.next;
79.     }
80.
81.     return null;
82. }
83.
84. public void init() {
85.     head = creatList();
86.     head1 = creatList();
87.     head2 = creatList();
88.     head1 = concatList(head1, head);
89.     head2 = concatList(head2, head);
90. }
91.
92. private LNode_01 creatList() {
93.     System.out.println("Creat new list");
94.     LinkList_01 list = new LinkList_01();
95.     list.header = null;
96.     list.creat();
97.     list.show();
98.     return list.header;
99. }
100.
101. public LNode_01 concatList(LNode_01 head1, LNode_01 head) {
102.     LNode_01 p = head1;
103.     while (p.next != null) {
104.         p = p.next;
105.     }
106.     p.next = head;
107.     return head1;
108. }
109.
110. /**
111. * @param args
112. */
```

```
113. public static void main(String[] args) {  
114.     First_CommonNode f = new First_CommonNode();  
115.     LNode_01 n = f.method_01();  
116.     if (n == null) {  
117.         System.out.println("not find");  
118.     } else {  
119.         System.out.println("first common elem: " + n.data);  
120.     }  
121.  
122.     n = f.method_02();  
123.     if (n == null) {  
124.         System.out.println("not find");  
125.     } else {  
126.         System.out.println("first common elem: " + n.data);  
127.     }  
128. }  
129.  
130. }
```

1.6.10. 输入两个字符串，从第一字符串中删除第二个字符串中所有的字符

题目：输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。例如，输入”They are students.”和”aeiou”，则删除之后的第一个字符串变成”Thy r stdnts.”。

其实这类题有个特点，字符串中的字符分为两类，就可以联想快速排序里的将当前的数组分为左右两组，其中左边的数字小于某值，右边的数字大于某值。这种大于和小于就是将分为两类。当然，等于也是，但是可以忽略。

还有一道面试题，将 int 数组转换为奇数偶数各一边。

这道题就是将字符分为在删除字符串中 和不在删除字符串中两种。

[java] [view plain](#) [copy](#) [print](#)?

```
1. public static void delete(String source, String dest){  
2.     boolean[] del = new boolean[128];  
3.     Arrays.fill(del, false);  
4.     char[] src = new char[source.length()];  
5.     src = source.toCharArray();  
6.     int j = 0;  
7.     for(int i=0; i<dest.length(); i++){  
8.         del[dest.charAt(i)] = true;  
9.     }  
10.    for(int i=0; i<source.length(); i++){  
11.        char c = source.charAt(i);  
12.        if(!del[c]){  
13.            src[j] = c;  
14.            j++;  
15.        }  
16.    }  
17.}  
18.    System.out.println(new String(src,0,j));  
19.  
20.}
```

1.7. 面试题集合（六）

1.7.1. 寻找丑数

题目：我们把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 1500 个丑数。

分析：寻找一个数是不是满足某种数（质数，水仙数）等，最简单的方法就是遍历，对于任意一个丑数必定可以写成 $2^m \cdot 3^n \cdot 5^p$ ，因而对于一个丑数，只含有 2, 3, 5 因子，也就意味着该数 $number \% 2 == 0$; $number \% 3 == 0$; $number \% 5 == 0$ ，如果一个数能被 2 整除，我们就连续除以 2；能被 3 整除，我们就连续除以 3；能被 5 整除，我们就连续除以 5；如果最后得到 1，则该数是素数，否则是丑数。

代码如下：

```
1 #include
2 #include<string>
3 using namespace std;
4
5 //判断一个给定的数 number 是否是丑数
6 bool IsUgly(int number)
7 {
8     while(number % 2 == 0)
9     {
10         number /= 2;
11     }
12     while(number % 3 ==0)
13     {
14         number /= 3;
15     }
16     while(number % 5 ==0)
17     {
18         number /= 5;
19     }
20     return(number == 1)?true:false;
21 }
22
23 //返回从 1 开始第 index 个丑数
24 int GetUglyNNumber(int index)
25 {
26     if(index <= 0)
27     {
28         return 0;
29     }
30
31     int number=0;
32     int count=0;
33     while(count < index)
34     {
35         ++number;
```

```

36         if(IsUgly(number))
37     {
38         ++count;
39     }
40
41 }
42
43     return number;
44 }
45
46 int main()
47 {
48     cout<<"Enter A Number:"<<endl;
49     int idx=0;
50     cin>>idx;
51     cout<<GETUGLYNUMBER(IDX)<<endl;
52     return 0;
53 }
```

上面计算中主要的不足在于，逐一遍历，这样对于不是丑数的数的判断会造成大量的时间浪费，如果能够根据已经计算好的丑数，计算出下一个丑数就可以避免这种情况，实现从丑数到丑数的高效算法，根据定义可知，后面的丑数肯定是前面已知丑数乘以 2, 3, 5 得到的。

我们假设一个数组中已经有若干丑数，并且这些丑数是按顺序排列的，我们把现有的最大丑数记为 max，则下一个丑数肯定是前面丑数乘以 2, 3, 5 得到的。不妨考虑乘以 2 得到的情况，我们把数组中的每一个数都乘以 2，由于原数组是有序的，因为乘以 2 后也是有序递增的，这样必然存在一个数 M2，它前面的每一个数都是小于等于 max，而包括 M2 在内的后面的数都是大于 max 的，因为我们还是要保持递增顺序，所以我们取第一个大于 max 的数 M2。同理对于乘以 3 的情况，可以取第一个大于 max 的数 M3，对于乘以 5 的情况，可以取第一个大于 max 的数 M5。

最终下一个丑数取: $\min\{M2, M3, M5\}$ 即可

代码如下：

```

1 #include
2 #include<string>
3 using namespace std;
4
```

```
5 //返回三个数中的最小者
6 int Min(int number1,int number2,int number3)
7 {
8     int min = (number1 < number2) ? number1 : number2;
9     min = (min < number3) ? min : number3;
10    return min;
11 }
12
13 //返回第 index 个丑数
14 int GetUglyNumber(int index)
15 {
16     if(index <= 0)
17     {
18         return 0;
19     }
20
21     int *pUglyNumbers = new int[index];
22     pUglyNumbers[0] = 1;
23     int nextUglyIndex = 1;
24
25     int *pMultiply2 = pUglyNumbers;
26     int *pMultiply3 = pUglyNumbers;
27     int *pMultiply5 = pUglyNumbers;
28
29     while(nextUglyIndex < index)
30     {
31         int min = Min(*pMultiply2 * 2,*pMultiply3 * 3,*pMultiply5 * 5);
32         pUglyNumbers[nextUglyIndex] = min;
33
34         while(*pMultiply2 * 2 <= pUglyNumbers[nextUglyIndex])
35         {
36             ++pMultiply2;
37         }
38         while(*pMultiply3 * 3 <= pUglyNumbers[nextUglyIndex])
39         {
```

```

40         ++pMultiply3;
41     }
42     while(*pMultiply5 * 5 <= pUglyNumbers[nextUglyIndex])
43     {
44         ++pMultiply5;
45     }
46
47     ++nextUglyIndex;
48 }
49
50     int ugly = pUglyNumbers[nextUglyIndex-1];
51     delete[] pUglyNumbers;
52     return ugly;
53
54 }
55
56 int main()
57 {
58     cout<<"Enter A number:"<<ENDL;
59     int number=0;
60     cin>>number;
61     cout<<GETUGLYNUMBER(NUMBER)<<ENDL;
62     return 0;
63 }
```

1.7.2. 输入数字 n，按顺序输出从 1 最大的 n 位 10 进制数

题目：输入数字 n，按顺序输出从 1 最大的 n 位 10 进制数。比如输入 3，则输出 1、2、3 一直到最大的 3 位数即 999。

分析：这是一道很有意思的题目。看起来很简单，其实里面却有不少的玄机。应聘者在解决这个问题的时候，最容易想到的方法是先求出最大的 n 位数是什么，然后用一个循环从 1 开始逐个输出。很快，我们就能写出如下代码：



```
// Print numbers from 1 to the maximum number with n digits, in order
void Print1ToMaxOfNDigits_1(int n)
```

```
{  
    // calculate 10^n  
  
    int number = 1;  
  
    int i = 0;  
  
    while(i++ < n)  
        number *= 10;  
  
  
    // print from 1 to (10^n - 1)  
  
    for(i = 1; i < number; ++i)  
        printf("%d\t", i);  
}
```



初看之下，好像没有问题。但如果我们仔细分析这个问题，就能注意到这里没有规定 n 的范围，当我们求最大的 n 位数的时候，是不是有可能用整型甚至长整型都会溢出？分析到这里，我们很自然的就想到我们需要表达一个大数，最常用的也是最容易实现的表达大数的方法是用字符串或者整型数组（当然不一定是最有效的）。

采用类似“ n 位 k 进制枚举”的算法（也类似于《编程之美》“电话号码对应英文单词”），代码如下：



```
const unsigned MaxBit = 10;  
static unsigned outArr[MaxBit];  
const char *HighBit = "123456789";  
const unsigned HighNum = 10;  
const char *OtherBit = "0123456789";  
const unsigned OtherNum = 11;  
  
void PrintAllNumberWithNBits(unsigned int N)  
{  //输出位数为 N 位的数，即：10^(N-1)...10^N-1  
    if (N < 1) return;  
  
    for (unsigned i = MaxBit-N; i < MaxBit; i++)  
        if (i == MaxBit-1)  
            outArr[i] = 0;  
        else  
            outArr[i] = 1;
```

```
while (1)
{
    int k = N;
    outArr[MaxBit-1-(N-k)]++;

    if(k==1 && outArr[MaxBit-1-(N-k)]>=HighNum)
        return; //前一个数字所有位均为 9, 循环结束
    else if (outArr[MaxBit-1-(N-k)] >= OtherNum) {
        int j = k;
        while (j>1 && outArr[MaxBit-1-(N-j)]>=OtherNum) {
            outArr[MaxBit-1-(N-j)] = 1;
            j--;
            outArr[MaxBit-1-(N-j)]++;
        }
        if (j==1 && outArr[MaxBit-1-(N-j)]>=HighNum)
            return;
    }

    for (unsigned int i = MaxBit-N; i< MaxBit; i++) {
        if (i == MaxBit-N)
            cout << HighBit[outArr[i]-1];
        else
            cout << OtherBit[outArr[i]-1];
    }
    cout << endl;
}

int main()
{
    unsigned number = 0;
    cin >> number;
```

```

for (unsigned i= 0; i< MaxBit; i++)
    outArr[i] = 0;

// 依次输出位数为 1,2,3...n-1 位的数
for (unsigned i= 1; i<= number; i++)
    PrintAllNumberWithNBits(i);

return 0;
}

```

1. **public class** Print_1_To_NDigit {
- 2.
3. /**
- 4. * Q65.输入数字 n, 按顺序输出从 1 最大的 n 位 10 进制数。比如
 输入 3, 则输出 1、2、3 一直到最大的 3 位数即 999
- 5. * 1.使用字符串存放数字。int a=123 --> char[] a={'1','2','3'};
- 6. * 2.递归。设置好第 n 位(最高位, 对应 char 数组的第 0 个元素)
 后, 接下来设置第 n-1,n-2.....位
- 7. * 3.打印时候, 前面的 0 不输出, 见 printNumber(char[] number)
- 8. */
- 9. **public static void** main(String[] args) {
- 10. **int** n=3;
- 11. Print_1_To_NDigit p=**new** Print_1_To_NDigit ();
- 12. p.print(n);
- 13. }
- 14.
- 15. **public void** print(**int** n){
- 16. **char**[] result=**new** **char**[n];
- 17. printHelpRecursive(result,n,0);
- 18. }
- 19.
- 20. //from result[0] to result[n-1],set 0-9 into it
- 21. **public void** printHelpRecursive(**char**[] result,**int** length,**int** index){

- 22. **if**(index==length){

```

23.         printNumber(result);
24.     }else{
25.         for(int i=0;i<=9;i++){
26.             result[index]=(char)('0'+i);
27.             printHelpRecursive(result,length,index+1);
28.         }
29.     }
30. }
31.
32. //don't print the prefix '0'.e.g,when "0012",print "12"
33. public void printNumber(char[] re){
34.     int len=re.length;
35.     boolean canPrint=false;
36.     for(int i=0;i<len-1;i++){
37.         if(!canPrint&&re[i]!='0'){
38.             canPrint=true;
39.         }
40.         if(canPrint){
41.             System.out.print(re[i]);
42.         }
43.     }
44.     System.out.println(re[len-1]);//the last bit is always printed.
45. }
46. }

```

1.7.3. 用递归颠倒一个栈

题目：用递归颠倒一个栈。例如输入栈{1, 2, 3, 4, 5}，1在栈顶。颠倒之后的栈为{5, 4, 3, 2, 1}，5处在栈顶。

分析：乍一看到这道题目，第一反应是把栈里的所有元素逐一 pop 出来，放到一个数组里，然后在数组里颠倒所有元素，最后把数组中的所有元素逐一 push 进入栈。这时栈也就颠倒过来了。颠倒一个数组是一件很容易的事情。不过这种思路需要显示分配一个长度为 O(n) 的数组，而且也没有充分利用递归的特性。

我们再来考虑怎么递归。我们把栈{1, 2, 3, 4, 5}看成由两部分组成：栈顶元素 1 和剩下的部分{2, 3, 4, 5}。如果我们能把{2, 3, 4, 5}颠倒过来，变成{5, 4, 3, 2}，然后在把原来的栈顶元素 1 放到底部，那么就整个栈就颠倒过来了，变成{5, 4, 3, 2, 1}。

接下来我们需要考虑两件事情：一是如何把{2, 3, 4, 5}颠倒过来变成{5, 4, 3, 2}。我们只要把{2, 3, 4, 5}看成由两部分组成：栈顶元素 2 和剩下的部分{3, 4, 5}。我们只要把{3, 4, 5}先颠倒过来变成{5, 4, 3}，然后再把之前的栈顶元素 2 放到最底部，也就变成了{5, 4, 3, 2}。

至于怎么把{3, 4, 5}颠倒过来……很多读者可能都想到这就是递归。也就是每一次试图颠倒一个栈的时候，现在栈顶元素 pop 出来，再颠倒剩下的元素组成的栈，最后把之前的栈顶元素放到剩下元素组成的栈的底部。递归结束的条件是剩下的栈已经空了。这种思路的代码如下：

```
// Reverse a stack recursively in three steps:  
// 1. Pop the top element  
// 2. Reverse the remaining stack  
// 3. Add the top element to the bottom of the remaining stack  
template<typename T> void ReverseStack(std::stack<T>& stack)  
{  
    if(!stack.empty())  
    {  
        T top = stack.top();  
        stack.pop();  
        ReverseStack(stack);  
        AddToStackBottom(stack, top);  
    }  
}
```

我们需要考虑的另外一件事情是如何把一个元素 e 放到一个栈的底部，也就是如何实现 AddToStackBottom。这件事情不难，只需要把栈里原有的元素逐一 pop 出来。当栈为空的时候，push 元素 e 进栈，此时它就位于栈的底部了。然后再把栈里原有的元素按照 pop 相反的顺序逐一 push 进栈。

注意到我们在 push 元素 e 之前，我们已经把栈里原有的所有元素都 pop 出来了，我们需要把它们保存起来，以便之后能把他们再 push 回去。我们当然可以开辟一个数组来做，但这没有必要。由于我们可以用递归来做这件事情，而递归本身就是一个栈结构。我们可以用递归的栈来保存这些元素。

基于如上分析，我们可以写出 AddToStackBottom 的代码：

```
// Add an element to the bottom of a stack:  
template<typename T> void AddToStackBottom(std::stack<T>& stack, T t)
```

```
{  
    if(stack.empty())  
    {  
        stack.push(t);  
    }  
    else  
    {  
        T top = stack.top();  
        stack.pop();  
        AddToStackBottom(stack, t);  
        stack.push(top);  
    }  
}
```

1.7.4. 从扑克牌中随机抽 5 张牌，判断是不是一个顺子

题目：从扑克牌中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2-10 为数字本身，A 为 1，J 为 11，Q 为 12，K 为 13，而大小王可以看成任意数字。

思路一：

我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把 5 张牌看成由 5 个数字组成的数组。大小王是特殊的数字，我们不妨把它们都当成 0，这样和其他扑克牌代表的数字就不重复了。

接下来我们来分析怎样判断 5 个数字是不是连续的。最直观的是，我们把数组排序。但值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数组中的空缺。也就是排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但如果我们将足够的 0 可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0, 1, 3, 4, 5}。在 1 和 3 之间空缺了一个 2，刚好我们有一个 0，也就是我们可以将它当成 2 去填补这个空缺。

于是我们需要做三件事情：把数组排序，统计数组中 0 的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意的是，如果数组中的非 0 数字重复出现，则该数组不是连续的。换成扑克牌的描述方式，就是如果一副牌里含有对子，则不可能是顺子。

更好的思路二：

1) 确认 5 张牌中除了 0，其余数字没有重复的（可以用表统计的方法）；

2) 满足这样的逻辑：（max, min 分别代表 5 张牌中的除 0 以外的最大值最小值）

如果没有 0，则 max-min=4，则为顺子，否则不是

如果有一个 0，则 max-min=4 或者 3，则为顺子，否则不是

如果有两个 0，则 max-min=4 或者 3 或者 2，则为顺子，否则不是

最大值和最小值在 1) 中就可以获得，这样就不用排序了

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. /*
2. 题目：从扑克牌中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2-10 为数
3. 字本身，A 为 1，J 为 11，Q 为 12，K 为 13，而大小王可以看成任意数字。
4. */
5.
6. /*
7. Author: Unimen
8. Date: 26/09/2011 07:23
9. Description: 数组问题
10. */
11.
12. /*
13. 解题报告：
14. 1、把王看成数字 0
15. 2、对所有数字排序，看其中连续数字的空缺数，与 0 的个数进行比较，当小于等于 0 的个数时
16. 就是连续的，即顺子
17. 3、要注意除了 0 以后，数字当中是否有重复的
18. 注意：
19. 按着上面的思路来，但不一定非得排序、查找重复数字，用些其他的办法，降低代码的编写难度
20. */
21.
22. #include <iostream>
23. #include <set>
24. #include <vector>
25. using namespace std;
26.
27. void GetMaxMin(const set<int> &setNum, int &nMax, int &nMin)
```

```
28. {
29.     nMin = 13;
30.     nMax = 1;
31.     set<int>::const_iterator iter = setNum.begin();
32.     for (; iter!=setNum.end(); ++iter)
33.     {
34.         if (*iter <nMin)
35.         {
36.             nMin = *iter;
37.         }
38.         if (*iter > nMax)
39.         {
40.             nMax = *iter;
41.         }
42.     }
43. }
44.
45. int Del0Num(set<int> &setNum, const vector<int> &data)
46. {
47.     int Num0 = 0;
48.     vector<int>::const_iterator iter = data.begin();
49.     for (; iter!=data.end(); ++iter)
50.     {
51.         if (*iter != 0)
52.         {
53.             setNum.insert(*iter);
54.         }
55.         else
56.         {
57.             Num0++;
58.         }
59.     }
60.     return Num0;
61. }
62.
```

```
63. bool IsContinuous(vector<int> data)
64. {
65.     int nMax = 0, nMin = 0;
66.     set<int> setNum;
67.     int num0 = Del0Num(setNum, data);
68.
69.     //如果其中存在对子，即除了 0 外有两个数字是重复的
70.     if (num0 + setNum.size() < data.size())
71.     {
72.         return false;
73.     }
74.
75.     GetMaxMin(setNum, nMax, nMin);
76.
77.     return nMax - nMin <= data.size() - 1; //数学方法
78.
79. }
80.
81. int main()
82. {
83.     vector<int> vec;
84.     int n;
85.     while (cin>>n)
86.     {
87.         for (int i=0; i<n; ++i)
88.         {
89.             int nTemp;
90.             cin>>nTemp;
91.             vec.push_back(nTemp);
92.         }
93.         cout<<IsContinuous(vec)<<endl;
94.     }
95.     return 0;
96. }
```

1.7.5. 把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 S

题目：把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 S。输入 n，打印出 S 的所有可能的值出现的概率。

分析：玩过麻将的都知道，骰子一共 6 个面，每个面上都有一个点数，对应的数字是 1 到 6 之间的一个数字。所以，n 个骰子的点数和的最小值为 n，最大值为 $6n$ 。因此，一个直观的思路就是定义一个长度为 $6n-n$ 的数组，和为 S 的点数出现的次数保存到数组第 $S-n$ 个元素里。另外，我们还知道 n 个骰子的所有点数的排列数 6^n 。一旦我们统计出每一点数出现的次数之后，因此只要把每一点数出现的次数除以 n^6 ，就得到了对应的概率。

该思路的关键就是统计每一点数出现的次数。要求出 n 个骰子的点数和，我们可以先把 n 个骰子分为两堆：第一堆只有一个，另一个有 $n-1$ 个。单独的那个有可能出现从 1 到 6 的点数。我们需要计算从 1 到 6 的每一种点数和剩下的 $n-1$ 个骰子来计算点数和。接下来把剩下的 $n-1$ 个骰子还是分成两堆，第一堆只有一个，第二堆有 $n-2$ 个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加，再和剩下的 $n-2$ 个骰子来计算点数和。分析到这里，我们不难发现，这是一种递归的思路。递归结束的条件就是最后只剩下一个骰子了。

基于这种思路，我们可以写出如下代码：

```
int g_maxValue = 6;

void PrintSumProbabilityOfDices_1(int number)
{
    if(number < 1)
        return;

    int maxSum = number * g_maxValue;
    int* pProbabilities = new int[maxSum - number + 1];
    for(int i = number; i <= maxSum; ++i)
        pProbabilities[i - number] = 0;

    SumProbabilityOfDices(number, pProbabilities);

    int total = pow((float)g_maxValue, number);
    for(int i = number; i <= maxSum; ++i)
    {
        float ratio = (float)pProbabilities[i - number] / total;
        printf("%d: %f\n", i, ratio);
```

```

    }

    delete[] pProbabilities;
}

void SumProbabilityOfDices(int number, int* pProbabilities)
{
    for(int i = 1; i <= g_maxValue; ++i)
        SumProbabilityOfDices(number, number, i, 0, pProbabilities);
}

void SumProbabilityOfDices(int original, int current, int value, int tempSum, int* pProbabilities)
{
    if(current == 1)
    {
        int sum = value + tempSum;
        pProbabilities[sum - original]++;
    }
    else
    {
        for(int i = 1; i <= g_maxValue; ++i)
        {
            int sum = value + tempSum;
            SumProbabilityOfDices(original, current - 1, i, sum, pProbabilities);
        }
    }
}

```

上述算法当 `number` 比较小的时候表现很优异。但由于该算法基于递归，它有很多计算是重复的，从而导致当 `number` 变大时性能让人不能接受。关于递归算法的性能讨论，详见[本博客系列的第 16 题](#)。

我们可以考虑换一种思路来解决这个问题。我们可以考虑用两个数组来存储骰子点数每一总数出现的次数。在一次循环中，第一个数组中的第 `n` 个数字表示骰子和为 `n` 出现的次数。那么在下一循环中，我们加上一个新的骰子。那么此时和为 `n` 的骰子出现的次数，应该等于上一次循环中骰子点数和为 `n-1`、`n-2`、`n-3`、`n-4`、`n-5` 与 `n-6` 的总和。所以我们把另一个数组的

第 n 个数字设为前一个数组对应的第 n-1、n-2、n-3、n-4、n-5 与 n-6 之和。基于这个思路，我们可以写出如下代码：

```
void PrintSumProbabilityOfDices_2(int number)
{
    double* pProbabilities[2];
    pProbabilities[0] = new double[g_maxValue * number + 1];
    pProbabilities[1] = new double[g_maxValue * number + 1];
    for(int i = 0; i < g_maxValue * number + 1; ++i)
    {
        pProbabilities[0][i] = 0;
        pProbabilities[1][i] = 0;
    }

    int flag = 0;
    for (int i = 1; i <= g_maxValue; ++i)
        pProbabilities[flag][i] = 1;

    for (int k = 2; k <= number; ++k)
    {
        for (int i = k; i <= g_maxValue * k; ++i)
        {
            pProbabilities[1 - flag][i] = 0;
            for(int j = 1; j <= i && j <= g_maxValue; ++j)
                pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];
        }
        flag = 1 - flag;
    }

    double total = pow((double)g_maxValue, number);
    for(int i = number; i <= g_maxValue * number; ++i)
    {
        double ratio = pProbabilities[flag][i] / total;
        printf("%d: %f\n", i, ratio);
    }
}
```

```
delete[] pProbabilities[0];
delete[] pProbabilities[1];
}
```

值得提出来的是，上述代码没有在函数里把一个骰子的最大点数硬编码(hard code)为 6，而是用一个变量 `g_maxValue` 来表示。这样做的好处时，如果某个厂家生产了最大点数为 4 或者 8 的骰子，我们只需要在代码中修改一个地方，扩展起来很方便。如果在面试的时候我们能对面试官提起对程序扩展性的考虑，一定能给面试官留下一个很好的印象。

1.7.6. 排出的所有数字中最小

题目：输入一个正整数数组，将它们连接起来排成一个数，输出能排出的所有数字中最小的一个。例如输入数组{32, 321}，则输出这两个能排成的最小数字 32132。请给出解决问题的算法，并证明该算法。

===== 以下内容引自原文 =====

分析：这是 09 年 6 月份百度新鲜出炉的一道面试题，从这道题我们可以看出百度对应聘者在算法方面有很高的要求。`m` 和 `n`，我们需要确定一个规则 `m` 和 `n` 哪个更大，而不是仅仅只是比较这两个数字的数值哪个更大。`m` 和 `n` 排成的数字 `mn` 和 `nm`，如果 `mn < nm`，那么我们应该输出 `mn`，也就是 `m` 应该排在 `n` 的前面，也就是 `m` 小于 `n`；反之，如果 `nm < mn`，`n` 小于 `m`。如果 `mn == mn`，`m` 等于 `n`。

这道题其实是希望我们能找到一个排序规则，根据这个规则排出来的数组能排成一个最小的数字。要确定排序规则，就得比较两个数字，也就是给出两个数字

根据题目的要求，两个数字

接下来我们考虑怎么去拼接数字，即给出数字 `m` 和 `n`，怎么得到数字 `mn` 和 `nm` 并比较它们的大小。直接用数值去计算不难办到，但需要考虑到的一个潜在问题是 `m` 和 `n` 都在 `int` 能表达的范围内，但把它们拼起来的数字 `mn` 和 `nm` 就不一定能用 `int` 表示了。所以我们需要解决大数问题。一个非常直观的方法就是把数字转换成字符串。

另外，由于把数字 `m` 和 `n` 拼接起来得到的 `mn` 和 `nm`，它们所含有的数字的个数肯定是相同的。因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

基于这个思路，我们可以写出下面的代码：

```
// Maximum int number has 10 digits in decimal system
const int g_MaxNumberLength = 10;
```

```

// String buffers to combine two numbers

char* g_StrCombine1 = new char[g_MaxNumberLength * 2 + 1];
char* g_StrCombine2 = new char[g_MaxNumberLength * 2 + 1];

// Given an array, print the minimum number
// by combining all numbers in the array

void PrintMinNumber(int* numbers,int length)

{
    if(numbers == NULL || length <= 0)
        return;

    // Convert all numbers as strings

    char** strNumbers = (char**)(new int[length]);
    for(int i = 0; i < length; ++i)
    {
        strNumbers[i] = new char[g_MaxNumberLength + 1];
        sprintf(strNumbers[i], "%d", numbers[i]);
    }

    // Sort all strings according to algorithm in function compare
    qsort(strNumbers, length, sizeof(char*), compare);

    for(int i = 0; i < length; ++i)
        printf("%s", strNumbers[i]);
    printf("\n");

    for(int i = 0; i < length; ++i)
        delete[] strNumbers[i];
    delete[] strNumbers;
}

// Compare two numbers in strNumber1 and strNumber2
// if [strNumber1][strNumber2] > [strNumber2][strNumber1],
// return value > 0
// if [strNumber1][strNumber2] = [strNumber2][strNumber1],

```

```

// return value = 0
// if [strNumber1][strNumber2] < [strNumber2][strNumber1],
// return value < 0
int compare(const void* strNumber1, const void* strNumber2)
{
    // [strNumber1][strNumber2]
    strcpy(g_StrCombine1, *(const char**)(strNumber1));
    strcat(g_StrCombine1, *(const char**)(strNumber2));

    // [strNumber2][strNumber1]
    strcpy(g_StrCombine2, *(const char**)(strNumber2));
    strcat(g_StrCombine2, *(const char**)(strNumber1));

    return strcmp(g_StrCombine1, g_StrCombine2);
}

```

上述代码中，我们在函数 `compare` 中定义比较规则，并根据该规则用库函数 `qsort` 排序。最后把排好序的数组输出，就得到了根据数组排成的最小的数字。

找到一个算法解决这个问题，不是一件容易的事情。但更困难的是我们需要证明这个算法是正确的。接下来我们来试着证明。

首先我们需要证明之前定义的比较两个数字大小的规则是有效的。一个有效的比较需要三个条件：1.自反性，即 a 等于 a ; 2.对称性，即如果 a 大于 b ，则 b 小于 a ; 3.传递性，即如果 a 小于 b ， b 小于 c ，则 a 小于 c 。现在分别予以证明。

1.

自反性。显然有 $aa=aa$ ，所以 $a=a$ 。

2. 对称性。如果 a 小于 b ，则 $ab < ba$ ，所以 $ba > ab$ 。因此 b 大于 a 。
3. 传递性。如果 a 小于 b ，则 $ab < ba$ 。当 a 和 b 用十进制表示的时候分别为 1 位和 m 位时， $ab = a \times 10^m + b$ ， $ba = b \times 10^1 + a$ 。所以 $a \times 10^m + b < b \times 10^1 + a$ 。于是有 $a \times 10^m - a < b \times 10^1 - b$ ，即 $a(10^m - 1) < b(10^1 - 1)$ 。所以 $a/(10^1 - 1) < b/(10^m - 1)$ 。

如果 b 小于 c , 则 $bc < cb$ 。当 c 表示成十进制时为 m 位。和前面证明过程一样, 可以得到 $b/(10^m - 1) < c/(10^n - 1)$ 。

所以 $a/(10^l - 1) < c/(10^n - 1)$ 。于是 $a(10^n - 1) < c(10^l - 1)$, 所以 $a \times 10^n + c < c \times 10^l + a$, 即 $ac < ca$ 。

所以 a 小于 c 。

在证明了我们排序规则的有效性之后, 我们接着证明算法的正确性。我们用反证法来证明。我们把 n 个数按照前面的排序规则排好顺序之后, 表示为 $A_1 A_2 A_3 \dots A_n$ 。我们假设这样排出来的两个数并不是最小的。即至少存在两个 x 和 y ($0 < x < y < n$), 交换第 x 个数和第 y 个数后, $A_1 A_2 \dots A_y \dots A_x \dots A_n < A_1 A_2 \dots A_x \dots A_y \dots A_n$ 。

由于 $A_1 A_2 \dots A_x \dots A_y \dots A_n$ 是按照前面的规则排好的序列, 所以有

$A_x < A_{x+1} < A_{x+2} < \dots < A_{y-2} < A_{y-1} < A_y$ 。

由于 A_{y-1} 小于 A_y , 所以 $A_{y-1} A_y < A_y A_{y-1}$ 。我们在序列 $A_1 A_2 \dots A_x \dots A_{y-1} A_y \dots A_n$ 交换 A_{y-1} 和 A_y , 有 $A_1 A_2 \dots A_x \dots A_{y-1} A_y \dots A_n < A_1 A_2 \dots A_x \dots A_y A_{y-1} \dots A_n$ (这个实际上也需要证明。感兴趣的读者可以自己试着证明)。我们就这样一直把 A_y 和前面的数字交换, 直到和 A_x 交换为止。于是就有 $A_1 A_2 \dots A_x \dots A_{y-1} A_y \dots A_n < A_1 A_2 \dots A_x \dots A_y A_{y-1} \dots A_n < A_1 A_2 \dots A_x \dots A_y A_{y-2} A_{y-1} \dots A_n < \dots < A_1 A_2 \dots A_y A_{y-2} A_{y-1} \dots A_n$ 。

同理由于 A_x 小于 A_{x+1} , 所以 $A_x A_{x+1} < A_{x+1} A_x$ 。我们在序列 $A_1 A_2 \dots A_y A_x A_{x+1} \dots A_{y-2} A_{y-1} \dots A_n$ 仅仅只交换 A_x 和 A_{x+1} , 有 $A_1 A_2 \dots A_y A_x A_{x+1} \dots A_{y-2} A_{y-1} \dots A_n < A_1 A_2 \dots A_y A_{x+1} A_x \dots A_{y-2} A_{y-1} \dots A_n$ 。我们接下来一直拿 A_x 和它后面的数字交换, 直到和 A_{y-1} 交换为止。于是就有

$A_1 A_2 \dots A_y A_x A_{x+1} \dots A_{y-2} A_{y-1} \dots A_n < A_1 A_2 \dots A_y A_{x+1} A_x \dots A_{y-2} A_{y-1} \dots A_n < \dots <$

$A_1 A_2 \dots A_y A_{x+1} A_{x+2} \dots A_{y-2} A_{y-1} A_x \dots A_n$ 。

所以 $A_1 A_2 \dots A_x \dots A_y \dots A_n < A_1 A_2 \dots A_y \dots A_x \dots A_n$ 。这和我们的假设的 $A_1 A_2 \dots A_y \dots A_x \dots A_n < A_1 A_2 \dots A_x \dots A_y \dots A_n$ 相矛盾。

所以假设不成立, 我们的算法是正确的。

===== 以上内容引自原文 =====

下面写我的思路。

刚拿到这题, 我没想到用字符串来做, 而是想把整数拆开成一位一位的数字来进行比较。这种方法在原文的评论中, 有其他人也是这么想的。

原文的分析已经说得比较明白了, 这个题其实就是要明确一种两个数之间的比较策略, 也就是一组数的排序规则, 具体点说, 就是要重写 compare 方法, 如果是 java 语言, 只要重载 compareTo 方法, 然后用 sort 方法就行了。

假设有两个数：A 和 B，其中，A 由 m 个数字组成，表示成 $a_1a_2\dots a_m$,B 由 n 个数字组成，表示成 $b_1b_2\dots b_n$. 比较的规则是这样的,从左到右比较，即从最高位开始，到最低位（个位）

- 1、如果 $a_i = b_i$,则比较下一位数;
- 2、如果 $a_i < b_i$,则 A 应该排到 B 前面;
- 3、如果 A 的所有位和 B 的前 m 位相同，即 $a_1=b_1,a_2=b_2,\dots,a_m=b_m$,另外， $n>m$ 。则继续比较 a_1 和 b_{m+1} 。

利用上面那个规则进行比较，直到确定 A 和 B 之间的关系。

伪代码:

[cpp] [view plain](#) [copy](#) [print?](#)

```
1. int compare(int A,int B){  
2.     m=A 的位数;  
3.     n=B 的位数;  
4.  
5.     k = min(m,n);  
6.  
7.     for i=[1,k]{  
8.         if a[i]<b[i]  
9.             return -1;  
10.        else if a[i]>b[i]  
11.            return 1;  
12.    }  
13.  
14. //上一个 for 循环如果没有 return，则说明某个数和另一个数的前部分完  
15. //全相同，则进行下面的比较  
16.  
17.     k = m-n;  
18.  
19.     if(k<0){  
20.         for i=[1,-k]{  
21.             if B[i]<B[m+i]  
22.                 return -1;  
23.             else if B[i]>B[m+i]  
24.                 return 1;
```

```
25.     }
26. }else{
27.     for i=[1,k]
28.         //对 A 做与上面同样的比较,
29.     }
30.
31.     return 0;
32. }
```

1.7.7. 数组的旋转

题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个排好序的数组的一个旋转，输出旋转数组的最小元素。例如数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为 1。

分析：这道题最直观的解法并不难。从头到尾遍历数组一次，就能找出最小的元素，时间复杂度显然是 $O(N)$ 。但这个思路没有利用输入数组的特性，我们应该能找到更好的解法。

我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面的子数组的元素都大于或者等于后面子数组的元素。我们还可以注意到最小的元素刚好是这两个子数组的分界线。我们试着用二元查找法的思路在寻找这个最小的元素。

首先我们用两个指针，分别指向数组的第一个元素和最后一个元素。按照题目旋转的规则，第一个元素应该是大于或者等于最后一个元素的（当然这里有特列，就是旋转个数为 0 即没有旋转的时候，我们要单独处理）。

接着我们得到处在数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一指针指向该中间元素，这样可以缩小寻找的范围。同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样同样可以缩小寻找的范围。我们接着再用更新之后的两个指针，去得到和比较新的中间元素，循环下去。按照上述的思路，我们的第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最后第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面对子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```

1. #include <iostream>
2. using namespace std;
3.
4. int FindMin(int *a, int length)
5. {
6.     int low=0;
7.     int high=length-1;
8.     int mid;
9.
10.    if(a[low]<a[high])//如果旋转个数为0，即没有旋转，单独处理，直接返回数组头元素
11.        return a[low];
12.
13.    while(low<=high)
14.    {
15.        mid=(low+high)/2;
16.        if(a[low]<a[mid])
17.            low=mid;
18.        else if(a[low]>a[mid])
19.            high=mid;
20.        else if(a[low]==a[mid])
21.            return a[low+1];
22.    }
23. }
24.
25. int main()
26. {
27.     int a[]={3,4,5,6,7,1,2};
28.     cout<<FindMin(a, 7)<<endl;
29.
30. }
```

1.7.8. 给出一个函数来输出一个字符串的所有排列

```

1. #include <stdio.h>
2.
3. void permutation(char * p_str, char * p_begin)
```

```

4.  {
5.      if(!p_str || !p_begin)
6.      {
7.          return;
8.      }
9.
10.     /*
11.     * If p_begin points to the end of string,
12.     * this round of permutation is finished,
13.     * print the permuted string.
14.    */
15.    if('\0' == *p_begin)
16.    {
17.        printf("%s\n", p_str);
18.    }
19.    /* Otherwise, permute string. */
20.    else
21.    {
22.        char * p_ch;
23.
24.        for(p_ch = p_begin; *p_ch != '\0'; ++p_ch)
25.        {
26.            char temp;
27.
28.            /* Swap p_ch and p_begin. */
29.            temp = *p_ch;
30.            *p_ch = *p_begin;
31.            *p_begin = temp;
32.
33.            permutation(p_str, p_begin + 1);
34.
35.            /* Restore p_ch and p_begin. */
36.            temp = *p_ch;
37.            *p_ch = *p_begin;
38.            *p_begin = temp;

```

```
39.     }
40.   }
41. }
42.
43. int main(int argc, char * argv[])
44. {
45.
46.
47.   char strr[4]="123";
48.   char strd[4];
49.   permutation(strr, strr);
50.   return 0;
51. }
```

1.7.9. 实现函数 double Power(double base,int exponent)

题目：实现函数 double Power(double base,int exponent)，求 base 的 exponent 次方。

不得使用库函数，同时不需要考虑大树问题。

这道题目有以下几点需要注意：

1. 0 的 0 次方是无意义的，非法输入
2. 0 的负数次方相当于 0 作为除数，也是无意义的，非法输入
3. base 如果非 0，如果指数 exponent 小于 0，可以先求 base 的 $|exponent|$ 次方，然后再求倒数
4. 判断 double 类型的 base 是否等于 0 不能使用==号。因为计算机表述小数(包括 float 和 double 型小数)都有误差，不能直接使用等号(==)判断两个小数是否相等。如果两个数的差的绝对值很小，那么可以认为两个 double 类型的数相等。

根据以上 4 个注意点，我们可以写出求指数的程序，代码如下：

```
#include<iostream>
#include<stdlib.h>
using namespace std;

bool isValidInput=false;
```

```
double PowerWithUnsingedExponent(double base,unsigned int absExp)
{
    double result=1.0;
    for(int i=0;i<absExp;i++)
        result*=base;
    return result;
}
```

//由于精度原因，double类型的变量不能用等号判断两个数是否相等，因此需要写 equal 函数

```
bool equal(double a,double b)
{
    if((a-b>-0.000001)&&(a-b<0.000001))
        return true;
    else
        return false;
}
```

```
double Power(double base,int exponent)
{
    //如果底数为 0 且指数小于 0，则表明是非法输入。
    if(equal(base,0.0) && exponent<=0)
    {
        isInvalidInput=true;
        return 0;
    }
```

```
unsigned int absExp;
//判断指数正负，去指数的绝对值
if(exponent<0)
    absExp=(unsigned int)(-exponent);
else
    absExp=(unsigned int)exponent;
```

```
double result=PowerWithUnsingedExponent(base,absExp);
```

```

//如果指数小于 0 则取倒数
if(exponent<0)
    result=1/result;

return result;
}

void main()
{
    double a=Power(2.0,13);
    cout<<a<<endl;

    system("pause");
}

```

1.7.10. 更优的解法:

假设我们求 2^{32} , 指数是 32, 那么我们需要进行 32 次循环的乘法。但是我们在求出 2^{16} 以后, 只需要在它的基础上再平方一次就可以求出结果。同理可以继续分解 2^{16} 。也就是 $a^n=a^{(n/2)} \cdot a^{(n/2)}$, (n 为偶数); 或者 $a^n=a^{((n-1)/2)} \cdot a^{((n-1)/2)} \cdot a$, (n 为奇数)。这样就将问题的规模大大缩小, 从原来的时间复杂度 $O(n)$ 降到现在的 $O(\log n)$ 。可以用递归实现这个思路, 代码如下:

```

double PowerWithUnsingedExponent(double base,unsigned int absExp)
{
    if(absExp==0)
        return 1;
    else if(absExp==1)
        return base;

    double result=PowerWithUnsingedExponent(base,absExp/2);
    result*=result;//指数减少一半以后用底数来乘
    if(absExp%2==1)//如果指数为奇数, 还得再乘一次底数
        result*=base;
}

```

```
    return result;  
}
```

上述程序使用了递归的方法，这样会增加程序的空间复杂度，下面我们使用循环实现递归的思路，代码如下：

```
double PowerWithUnsignedExponent(double base,unsigned int absExp)  
{  
    if(absExp==0)  
        return 1;  
    else if(absExp==1)  
        return base;  
  
    double result=1.0*base;  
    for(int i=2;i<=absExp;i=i*2)  
        result*=result;  
    if(absExp%2==1)//如果指数为奇数，还得再乘一次底数  
        result*=base;  
  
    return result;  
}
```

1.7.11. 单列模式

出来这样的效果，叫做单列模式。

你可以参考一下的下面的笔记：单例模式(Singleton)：

保证程序永远能获得同一个 Java 对象。

示例一： ---- 称饿汉式：

开始就急着创建对象，所以称为饿汉式

```
class A{  
    private static A a = new A();  
    private A(){  
        //使用 private 访问控制符来让外界不能 new 该对象；只能调用静态方法来获取。  
        //这样方可实现外界永远获得的是同一个 Java 对象。  
    }  
    public static A getInstance()
```

```
{  
    return a;  
}  
}  
}
```

示例二： --- 称懒汉式：

开始不创建对象，等到用的时候才创建，行为有点懒；所以称为俄汉式。

```
class A{  
    private static A a=null;  
    private A(){ }  
    public synchronized static A getInstance()  
    { if(a == null){ a = new A(); }  
        return a;  
    }  
}
```

设计一个类，该类只能生成 3 个实例,java 实现

```
public class ClassicSingleton {  
    private static List instanceList = new ArrayList();  
    protected ClassicSingleton() {  
        // Exists only to defeat instantiation.  
    }  
    public static ClassicSingleton getInstance() {  
        if(instanceList.size() < 3) {  
            ClassicSingleton c = new ClassicSingleton();  
            instanceList.add(c); return c;  
        }  
        return instanceList.get(0);  
    }  
}
```

1.8. 面试题集合 (七)

1.8.1. 找出该字符串中对称的子字符串的最大长度

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`.

[java] [view plain](#) [copy](#) [print](#)?

```
1. public static int maxLengthMiddle(char[] array, int index) {  
2.     int length = 1; //最长的子字符串长度  
3.     int j = 1; //前后移动的指针  
4.     while ((array[index - j] == array[index + j]) && (index - j) >= 0 && array.length > (index + j)) {  
5.         length += 2;  
6.         j++;  
7.     }  
8.  
9.     return length;  
10. }
```

2. 第 `i` 个字符串是对称字符串的其中一个中心。比如“abba”。

[java] [view plain](#) [copy](#) [print](#)?

```
1. public static int maxLengthMirror(char[] array, int index) {  
2.     int length = 0; //最长的子字符串长度  
3.     int j = 0; //前后移动的指针
```

```
4.     while ((array[index - j] == array[index + j + 1]) && (index - j) >= 0 && array.length > (index + j + 1))
5.     ) {
6.         length += 2;
7.         j++;
8.
9.     return length;
10. }
```

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。

[java] [view plain](#) [copy](#) [print](#)?

```
1. public static int palindrain(char[] array) {
2.     if (array.length == 0) return 0;
3.     int maxLength = 0;
4.     for (int i = 0; i < array.length; i++) {
5.         int tempMaxLength = -1;
6.         int length1 = maxLengthMiddle(array, i);
7.         int length2 = maxLengthMirror(array, i);
8.         tempMaxLength = (length1 > length2) ? length1 : length2;
9.         if (tempMaxLength > maxLength) {
10.             maxLength = tempMaxLength;
11.         }
12.     }
13.     return maxLength;
14. }
```

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 $O(N^2)$ 。

另一种方法是把原始字符串反转，这样最长对称子字符串问题可以转化成求两个字符串的最长公共子字符串问题（备注：不是最长公共子序列）。

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，有兴趣的可以搜索一下。

1.8.2. 数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字

题目：数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字。

分析：

1.首先我们想到如果是一个排序好的数组，那么我们只需要遍历一次数组，统计好每个数字出现的次数，如果大于数组长度的一半就输出这个数字。或者只需要直接输出 $\text{array}[\lfloor N/2 \rfloor]$ 的值即可。

2.如果是杂乱无章的数据我们可能回想先排序，然后按 1 操作即可。但是排序的最小时间复杂度（快速排序） $O(N \log N)$ ，加上遍历，时间复杂度为： $O(N \log N + N)$ ，如果选择直接输出 $\text{array}[\lfloor N/2 \rfloor]$ 的值的话，时间复杂度缩小为 $O(N \log N)$ 。

3.如果说数字只有 0-9 的话可以考虑设计一个 Hash table，遍历一次就能知道每个数字出现的次数。但是数字范围不知，所以 Hash 表不好创建。

4.出现的次数超过数组长度的一半，表明这个数字出现的次数比其他数字出现的次数的总和还多。所以我们可以考虑每次删除两个不同的数，那么在剩下的数中，出现的次数仍然超过总数的一半。通过不断重复这个过程，不断排除掉其它的数，最终找到那个出现次数超过一半的数字。这个方法，免去了上述思路一、二的排序，也避免了思路三空间 $O(N)$ 的开销，总得说来，时间复杂度只有 $O(N)$ ，空间复杂度为 $O(1)$ ，不失为最佳方法。

例：数组 $a[5]=\{0,1,2,1,1\}$ ；

我们要查找的数字为 1，操作步骤为：遍历整个数组，然后每次删除不同的两个数字，过程如下：

0 1 2 1 1 => 2 1 1 => 1

1.8.3. 输入二叉树中的两个结点，输出这两个结点在数中最低的共同父结点

题目：二叉树的结点定义如下：

```
struct TreeNode
{
    int m_nvalue;
    TreeNode* m_pLeft;
    TreeNode* m_pRight;
};
```

输入二叉树中的两个结点，输出这两个结点在数中最低的共同父结点。

分析：求数中两个结点的最低共同结点是面试中经常出现的一个问题。这个问题至少有两个变种。

第一变种是二叉树是一种特殊的二叉树：查找二叉树。也就是树是排序过的，位于左子树上的结点都比父结点小，而位于右子树的结点都比父结点大。我们只需要从根结点开始和两个

结点进行比较。如果当前结点的值比两个结点都大，则最低的共同父结点一定在当前结点的左子树中。如果当前结点的值比两个结点都小，则最低的共同父结点一定在当前结点的右子树中。

第二个变种是树不一定是二叉树，每个结点都有一个指针指向它的父结点。于是我们可以从任何一个结点出发，得到一个到达树根结点的单向链表。因此这个问题转换为两个单向链表的第一个公共结点。我们在[本面试题系列的第 35 题](#)讨论了这个问题。

现在我们回到这个问题本身。所谓共同的父结点，就是两个结点都出现在这个结点的子树中。因此我们可以定义一函数，来判断一个结点的子树中是不是包含了另外一个结点。这不是件很难的事，我们可以用递归的方法来实现：

```
//////////  
// If the tree with head pHead has a node pNode, return true.  
// Otherwise return false.  
/////////  
bool HasNode(TreeNode* pHead, TreeNode* pNode)  
{  
    if(pHead == pNode)  
        return true;  
  
    bool has = false;  
  
    if(pHead->m_pLeft != NULL)  
        has = HasNode(pHead->m_pLeft, pNode);  
  
    if(!has && pHead->m_pRight != NULL)  
        has = HasNode(pHead->m_pRight, pNode);  
  
    return has;  
}
```

我们可以从根结点开始，判断以当前结点为根的树中左右子树是不是包含我们要找的两个结点。如果两个结点都出现在它的左子树中，那最低的共同父结点也出现在它的左子树中。如果两个结点都出现在它的右子树中，那最低的共同父结点也出现在它的右子树中。如果两个结点一个出现在左子树中，一个出现在右子树中，那当前的结点就是最低的共同父结点。基于这个思路，我们可以写出如下代码：

```
/////////  
// Find the last parent of pNode1 and pNode2 in a tree with head pHead
```

```

||||||||||||||||||||||||||||||||||||||||||||||||

TreeNode* LastCommonParent_1(TreeNode* pHead, TreeNode* pNode1, TreeNode* pNode2)
{
    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)
        return NULL;

    // check whether left child has pNode1 and pNode2
    bool leftHasNode1 = false;
    bool leftHasNode2 = false;
    if(pHead->m_pLeft != NULL)
    {
        leftHasNode1 = HasNode(pHead->m_pLeft, pNode1);
        leftHasNode2 = HasNode(pHead->m_pLeft, pNode2);
    }

    if(leftHasNode1 && leftHasNode2)
    {
        if(pHead->m_pLeft == pNode1 || pHead->m_pLeft == pNode2)
            return pHead;

        return LastCommonParent_1(pHead->m_pLeft, pNode1, pNode2);
    }

    // check whether right child has pNode1 and pNode2
    bool rightHasNode1 = false;
    bool rightHasNode2 = false;
    if(pHead->m_pRight != NULL)
    {
        if(!leftHasNode1)
            rightHasNode1 = HasNode(pHead->m_pRight, pNode1);
        if(!leftHasNode2)
            rightHasNode2 = HasNode(pHead->m_pRight, pNode2);
    }

    if(rightHasNode1 && rightHasNode2)

```

```

{
    if(pHead->m_pRight == pNode1 || pHead->m_pRight == pNode2)
        return pHead;

    return LastCommonParent_1(pHead->m_pRight, pNode1, pNode2);
}

if((leftHasNode1 && rightHasNode2)
   || (leftHasNode2 && rightHasNode1))
    return pHead;

return NULL;
}

```

接着我们来分析一下这个方法的效率。函数 HasNode 的本质就是遍历一棵树，其时间复杂度是 $O(n)$ (n 是树中结点的数目)。由于我们根结点开始，要对每个结点调用函数 HasNode。因此总的时间复杂度是 $O(n^2)$ 。

我们仔细分析上述代码，不难发现我们判断以一个结点为根的树是否含有某个结点时，需要遍历树的每个结点。接下来我们判断左子结点或者右子结点为根的树中是否含有要找结点，仍然需要遍历。第二次遍历的操作其实在前面的第一次遍历都做过了。由于存在重复的遍历，本方法在时间效率上肯定不是最好的。

前面我们提过如果结点中有一个指向父结点的指针，我们可以把问题转化为求两个链表的共同结点。现在我们可以想办法得到这个链表。我们在[本面试题系列的第 4 题](#)中分析过如何得到一条中根结点开始的路径。我们在这里稍作变化即可：

```

// Get the path from pHead and pNode in a tree with head pHead
// ...
bool GetNodePath(TreeNode* pHead, TreeNode* pNode, std::list<TreeNode*>& path)
{
    if(pHead == pNode)
        return true;

    path.push_back(pHead);

    bool found = false;
    if(pHead->m_pLeft != NULL)

```

```

    found = GetNodePath(pHead->m_pLeft, pNode, path);
    if(!found && pHad->m_pRight)
        found = GetNodePath(pHead->m_pRight, pNode, path);

    if(!found)
        path.pop_back();

    return found;
}

```

由于这个路径是从跟结点开始的。最低的共同父结点就是路径中的最后一个共同结点：

```

// Get the last common Node in two lists: path1 and path2
// ...
TreeNode* LastCommonNode
(
    const std::list<TreeNode*>& path1,
    const std::list<TreeNode*>& path2
)
{
    std::list<TreeNode*>::const_iterator iterator1 = path1.begin();
    std::list<TreeNode*>::const_iterator iterator2 = path2.begin();

    TreeNode* pLast = NULL;

    while(iterator1 != path1.end() && iterator2 != path2.end())
    {
        if(*iterator1 == *iterator2)
            pLast = *iterator1;

        iterator1++;
        iterator2++;
    }

    return pLast;
}

```

有了前面两个子函数之后，求两个结点的最低共同父结点就很容易了。我们先求出从根结点出发到两个结点的两条路径，再求出两条路径的最后一个共同结点。代码如下：

```
////////////////////////////////////////////////////////////////  
// Find the last parent of pNode1 and pNode2 in a tree with head pHead  
////////////////////////////////////////////////////////////////  
  
TreeNode* LastCommonParent_2(TreeNode* pHead, TreeNode* pNode1, TreeNode* pNode2)  
{  
    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)  
        return NULL;  
  
    std::list<TreeNode*> path1;  
    GetNodePath(pHead, pNode1, path1);  
  
    std::list<TreeNode*> path2;  
    GetNodePath(pHead, pNode2, path2);  
  
    return LastCommonNode(path1, path2);  
}
```

这种思路的时间复杂度是 $O(n)$ ，时间效率要比第一种方法好很多。但同时我们也要注意到，这种思路需要两个链表来保存路径，空间效率比不上第一个方法。

[java] [view plain](#) [copy print](#)

```
1. import java.util.LinkedList;  
2. import java.util.List;  
3.  
4. import ljn.help.*;  
5. public class BTreelowestParentOfTwoNodes {  

```

```

14.          / \
15.          8  7
16.          / \
17.          9  2
18.          / \
19.          4  7
20.        */
21.    int[] data={ 1,8,7,9,2,0,0,0,0,4,7 };
22.    Node head=Helper.createTree(data);
23.    Node node1=new Node(4);
24.    Node node2=new Node(9);//their lowest parent should be 8
25.    LinkedList<Node> path1=new LinkedList<Node>();//should be 1,8,2,4
26.    LinkedList<Node> path2=new LinkedList<Node>();//should be 1,8,9
27.    createPath(head,node1,path1);
28.    createPath(head,node2,path2);
29.    Node lowestParent=lastCommonNode(path1,path2);
30.    System.out.println(lowestParent.getData());
31.  }
32.
33. //create a path from BTrees root to the specific node
34. public static boolean createPath(Node head,Node node,LinkedList<Node> path){
35.   if(head.getData()==node.getData()){
36.     return true;
37.   }
38.   boolean found=false;
39.   path.addLast(head);
40.   if(head.getLeft()!=null){
41.     found=createPath(head.getLeft(),node,path);
42.   }
43.   if(!found&&head.getRight()!=null){
44.     found=createPath(head.getRight(),node,path);
45.   }
46.   if(!found){
47.     path.removeLast();
48.   }

```

```

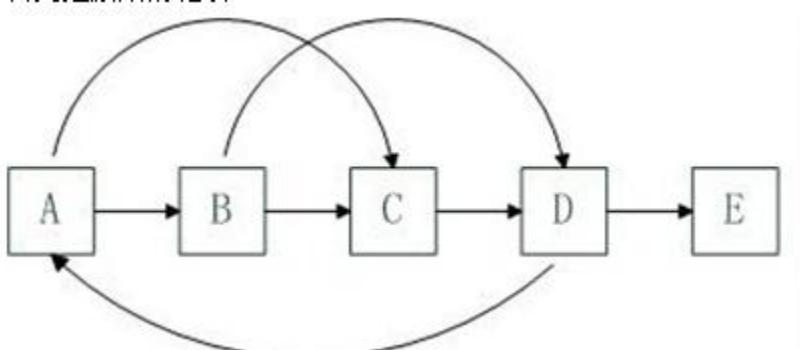
49.     return found;
50. }
51. /*
52. * find 'lastCommonNode' of two list and return.
53. * e.g
54. * list1=1,2,3,5
55. * list2=1,2,3,4
56. * we return 3
57. */
58. public static Node lastCommonNode(List<Node> list1,List<Node> list2){
59.     Node result=null;
60.     int len1=list1.size();
61.     int len2=list2.size();
62.     if(len1==0||len2==0){
63.         return null;
64.     }
65.     for(int i=0,j=0;i<len1&&j<len2;i++,j++){
66.         if(list1.get(i)==list2.get(j)){
67.             result=list1.get(i);
68.         }
69.     }
70.     return result;
71. }
72. }
```

1.8.4. 复杂链表

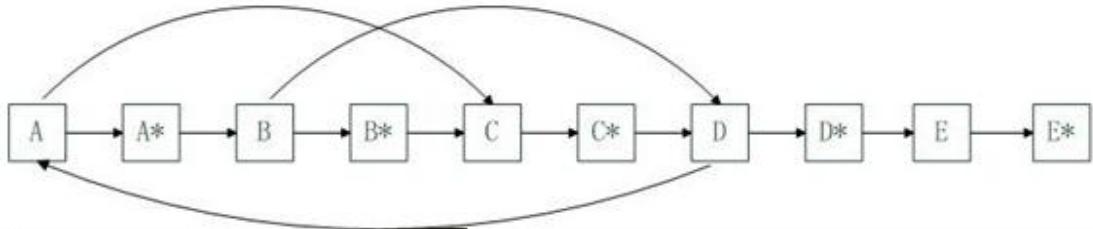
Q:有一个复杂链表，其结点除了有一个 `m_pNext` 指针指向下一个结点外，还有一个 `m_pSibling` 指向链表中的任一结点或者 `NULL`。请完成函数 `ComplexNode* Clone(ComplexNode* pHead)`，以复制一个复杂链表。

A:一开始想这道题毫无思路，如果蛮来，首先创建好正常的链表，然后考虑 `sibling` 这个分量，则需要 $O(n^2)$ 的时间复杂度，然后一个技巧便可以巧妙的解答此题。看图便知。

首先是原始的链表



然后我们还是首先复制每一个结点 N 为 N^* , 不同的是我们将 N^* 放在对应的 N 后面, 即为



然后我们要确定每一个 N^* 的 $sibling$ 分量, 非常明显, N 的 $sibling$ 分量的 $next$ 就是 N^* 的 $sibling$ 分量。

最后, 将整个链表拆分成原始链表和拷贝出的链表。

这样, 我们就解决了一个看似非常混乱和复杂的问题。

[cpp] [view plain](#) [copy](#) [print?](#)

```
1. struct Node
2. {
3.     int val;
4.     Node* next;
5.     Node* sibling;
6. };
7. void Clone(Node* head)
8. {
9.     Node* current=head;
10.    while(current)
11.    {
12.        Node* temp=new Node;
13.        temp->val=current->val;
14.        temp->next=current->next;
15.        temp->sibling=NULL;
```

```
16.     current->next=temp;
17.     current=temp->next;
18. }
19. }
20.
21.
22. void ConstructSibling(Node* head)
23. {
24.     Node* origin=head;
25.     Node* clone;
26.     while(origin)
27.     {
28.         clone=origin->next;
29.         if(origin->sibling)
30.             clone->sibling=origin->sibling->next;
31.         origin=clone->next;
32.     }
33. }
34.
35. Node* Split(Node* head)
36. {
37.     Node *CloneHead,*clone,*origin;
38.     origin=head;
39.     if(origin)
40.     {
41.         CloneHead=origin->next;
42.         origin->next=CloneHead->next;
43.         origin=CloneHead->next;
44.         clone=CloneHead;
45.     }
46.     while(origin)
47.     {
48.         Node* temp=origin->next;
49.         origin->next=temp->next;
50.         origin=origin->next;
```

```
51.     clone->next=temp;
52.     clone=temp;
53. }
54. return CloneHead;
55. }
56.
57.
58. //the whole thing
59. Clone(head);
60. ConstructSibling(head);
61. return Split(head);
```

1.8.5. 链表面试题

题一、给定单链表，检测是否有环。

使用两个指针 p1,p2 从链表头开始遍历，p1 每次前进一步，p2 每次前进两步。如果 p2 到达链表尾部，说明无环，否则 p1、p2 必然会在某个时刻相遇($p1==p2$)，从而检测到链表中有环。

http://ostermiller.org/find_loop_singly_linked_list.html

这篇文章讲了很多好的坏得相关算法。

题二、给定两个单链表(**head1, head2**)，检测两个链表是否有交点，如果有返回第一个交点。

如果 $head1==head2$ ，那么显然相交，直接返回 $head1$ 。

否则，分别从 $head1, head2$ 开始遍历两个链表获得其长度 $len1$ 与 $len2$ 。假设 $len1>=len2$ ，那么指针 $p1$ 由 $head1$ 开始向后移动 $len1-len2$ 步。指针 $p2=head2$ ，下面 $p1, p2$ 每次向后前进一步并比较 $p1p2$ 是否相等，如果相等即返回该结点，否则说明两个链表没有交点。

题三、给定单链表(**head**)，如果有环的话请返回从头结点进入环的第一个节点。

运用题一，我们可以检查链表中是否有环。

如果有环，那么 $p1p2$ 重合点 p 必然在环中。从 p 点断开环，方法为： $p1=p, p2=p->next, p->next=NULL$ 。此时，原单链表可以看作两条单链表，一条从 $head$ 开始，另一条从 $p2$ 开始，于是运用题二的方法，我们找到它们的第一个交点即为所求。

也可以不断开环。设重合点为 $p3$ ，从 $p3$ 开始遍历这个环，同时从表头开始走，检查每步是否在那个环中。这个方法大概有 $nlogn$ 。

使用快慢指针，第一次相遇，表明存在循环。继续快慢指针，第二次相遇，得到的 iteration 步长为环的长度。分别从相遇点和第一个节点出发，都是步长为 1 的指针，当相遇时，得到的 iteration 步长为环首的位置。

题四、只给定单链表中某个结点 p(并非最后一个结点，即 $p->next \neq NULL$)指针，删除该结点。

办法很简单，首先是放 p 中数据，然后将 p->next 的数据 copy 入 p 中，接下来删除 p->next 即可。

题五、只给定单链表中某个结点 p(非空结点)，在 p 前面插入一个结点。

办法与前者类似，首先分配一个结点 q，将 q 插入在 p 后，接下来将 p 中的数据 copy 入 q 中，然后再将要插入的数据记录在 p 中。

题六、给定单链表头结点，删除链表中倒数第 k 个结点。

使用两个节点 p1,p2，p1 初始化指向头结点，p2 一直指向 p1 后第 k 个节点，两个结点平行向后移动直到 p2 到达链表尾部(NULL)，然后根据 p1 删除对应结点。

题七、链表排序

链表排序最好使用归并排序算法。堆排序、快速排序这些在数组排序时性能非常好的算法，在链表只能“顺序访问”的魔咒下无法施展能力；但是归并排序却如鱼得水，非但保持了它 $O(n \log n)$ 的时间复杂度，而且它在数组排序中广受诟病的空间复杂度在链表排序中也从 $O(n)$ 降到了 $O(1)$ 。真是好得不得了啊，哈哈。以上程序是递推法的程序，另外值得一说的是看看那个时间复杂度，是不是有点眼熟？对！这就是分治法的时间复杂度，归并排序又是 divide and conquer。

```
double cmp(ListNode *p ,ListNode *q)
{return (p->keyVal - q->keyVal);}

ListNode* mergeSortList(ListNode *head)
{
    ListNode *p, *q, *tail, *e;
    int nstep = 1;
    int nmerges = 0;
    int i;
    int psize, qsize;

    if (head == NULL || head->next == NULL)
        {return head;}
    while (1)
    { p = head;
    tail = NULL;
```

```

nmerges = 0;
while (p)
{ nmerges++; q = p; psize = 0;
for (i = 0; i < nstep; i++) {
psize++;
q = q->next;
if (q == NULL)break;
}
qsize = nstep;
while (psize >0 || (qsize >0 && q))
{
if (psize == 0 ){e = q; q = q->next; qsize--;}
elseif (q == NULL || qsize == 0){e = p; p = p->next; psize--;}
elseif (cmp(p,q) <= 0){e = p; p = p->next; psize--;}
else{e = q; q = q->next; qsize--;}
if (tail != NULL){tail->next = e;}
else{head = e;}
tail = e;
}
p = q;
}
tail->next = NULL;
if (nmerges <= 1){return head;}
else{nstep <<= 1;}
}
}


```

题八、倒转单链表

给出非递归和递归解法：

```
#include <iostream>

using namespace std;

struct Node
{
    int data;
    Node* next;
}*head;

// 非递归写法
Node* InverseLinkedList(Node* head)
{
    if(head == NULL)
        return NULL;
    Node* newHead = NULL;
    while(head != NULL)
    {
        Node* nextNode = head->next;
        head->next = newHead;
        newHead = head;
        head = nextNode;
    }
    return newHead;
}

// 递归写法
Node* InverseLinkedListRecur(Node* head)
{
    if(head == NULL)
        return NULL;
    if(head->next == NULL)
        return head;
    Node* newHead = InverseLinkedListRecur(head->next);
    Node *tmp = newHead;
```

```
while(tmp->next != NULL)
{
    tmp = tmp->next;
}
tmp->next = head;
head->next = NULL;
return newHead;
}

int main()
{
// 构建链表 9->8->7->6->5->4->3->2->1->0->NULL
for(int i = 0; i < 10; i++)
{
    Node* node = new Node();
    node->data = i;
    node->next = head;
    head = node;
}
head = InverseLinkedList(head);
Node *tmp = head;
while(head != NULL)
{
    cout << head->data << " ";
    head = head->next;
}
cout << endl;
head = InverseLinkedListRecur(tmp);
tmp = head;
while(head != NULL)
{
    cout << head->data << " ";
    head = head->next;
}
cout << endl;
```

```
while(tmp != NULL)
{
    Node* next = tmp->next;
    delete tmp;
    tmp = next;
}
```

题九、两个有序链表的合并

有两个有序链表，各自内部是有序的，但是两个链表之间是无序的

```
typedef struct node{
    int data;
    struct node * next;
}* List;
```

```
List mergeSortedLinkList(List list1, List list2)
```

```
{
    List pList1,pList2,mergedList,pCurNode;

    if (list1 == NULL)
    {
        return list2;
    }
    if (list2 == NULL)
    {
        return list1;
    }
```

```
pList1 = list1;
pList2 = list2;
mergedList = NULL;
if (pList1==pList2)
{
    mergedList = pList1;
    pList1 = pList1->next;
    pList2 = pList2->next;
```

```

    }
else
{
    if (pList1->data <= pList2->data)
    {
        mergedList = pList1;
        pList1 = pList1->next;
    }
    else
    {
        mergedList = pList2;
        pList2 = pList2->next;
    }
}
pCurNode = mergedList;
while(pList1 && pList2)
{
    if (pList1==pList2)
    {
        pCurNode->next = pList1;
        pCurNode = pList1;
        pList1 = pList1->next;
        pList2 = pList2->next;
    }
    else
    {
        if (pList1->data <= pList2->data)
        {
            pCurNode->next = pList1;
            pCurNode = pList1;
            pList1 = pList1->next;
        }
        else
        {
            pCurNode->next = pList2;

```

```

    pCurNode = pList2;
    pList2 = pList2->next;
}
}

pCurNode->next = pList1 ? pList1 : pList2;

return mergedList;
}

```

题十、找出链表的中间元素

单链表的一个比较大的特点用一句广告语来说就是“不走回头路”，不能实现随机存取（random access）。如果我们想要找一个数组 a 的中间元素，直接 $a[\text{len}/2]$ 就可以了，但是链表不行，因为只有 $a[\text{len}/2 - 1]$ 知道 $a[\text{len}/2]$ 在哪儿，其他人不知道。因此，如果按照数组的做法依样画葫芦，要找到链表的中点，我们需要做两步(1)知道链表有多长(2)从头结点开始顺序遍历到链表长度的一半的位置。这就需要 $1.5n$ (n 为链表的长度)的时间复杂度了。有没有更好的办法呢？有的。想法很简单：两个人赛跑，如果 A 的速度是 B 的两倍的话，当 A 到终点的时候，B 应该刚到中点。这只需要遍历一遍链表就行了，还不用计算链表的长度。

1.8.6. 链表和数字的区别在哪里

链表和数字的区别在哪里？数组是线性结构，可以直接索引，即要去第 i 个元素， $a[i]$ 即可。链表也是线性结构，要取第 i 个元素，只需用指针往后遍历 i 次就可。貌似链表比数组还要麻烦些，而且效率低些。

想到这些相同处中的一些细微的不同处，于是他们的真正不同处渐渐显现了：链表的效率为何比数组低些？先从两者的初始化开始。数组无需初始化，因为数组的元素在内存的栈区，系统自动申请空间。而链表的结点元素在内存的堆区，每个元素须手动申请空间，如 `malloc`。也就是说数组是静态分配内存，而链表是动态分配内存。链表如此麻烦为何还要用链表呢？数组不能完全代替链表吗？回到这个问题只需想想我们当初是怎么完成学生信息管理系统的。为何那时候要用链表？因为学生管理系统中的插入，删除等操作都很灵活，而数组则大小固定，也无法灵活高效的插入，删除。因为堆操作灵活性更强。数组每次插入一个元素就需要移动已有元素，而链表元素在堆上，无需这么麻烦。

说了这么多，数组和链表的区别整理如下：

数组静态分配内存，链表动态分配内存；

数组在内存中连续，链表不连续；
数组元素在栈区，链表元素在堆区；
数组利用下标定位，时间复杂度为 $O(1)$ ，链表定位元素时间复杂度 $O(n)$ ；
数组插入或删除元素的时间复杂度 $O(n)$ ，链表的时间复杂度 $O(1)$ 。

1.8.7. strstr()函数功能

```
1. <PRE class=java name="code"></PRE><PRE class=java name="code">char *strstr(const char *s1, const char *s2)
2. {
3.     int n;
4.     if (*s2)
5.     {
6.         while (*s1)
7.         {
8.             for (n=0; *(s1 + n) == *(s2 + n); n++)
9.             {
10.                 if (!*(s2 + n + 1))
11.                     return (char *)s1;
12.             }
13.             s1++;
14.         }
15.         return NULL;
16.     }
17.     else
18.         return (char *)s1;
19. }
20. </PRE>
21. <PRE></PRE>
```

1.8.8. 一个 int 数组，里面数据无任何限制，要求求出所有这样的数 a[i]，其左边的数都小于等于它，右边的数都大于等于它

题目：一个 int 数组，里面数据无任何限制，要求求出所有这样的数 a[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。

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. <PRE class=java name="code">public static void smallLarge(int[] array) throws Exception{
2.     //the array's size must be larger than 2
3.     if (array == null || array.length < 1) {
4.         throw new Exception("the array is null or the array has no element!");
5.     }
6.     int[] rightMin = new int[array.length];</PRE><PRE class=java name="code">    //最
    右边的一个数的右边最小就是自己
7.     rightMin[array.length - 1] = array[array.length - 1];
8.     //get the minimum value of the array[] from i to array.length - 1 从右向左扫描
9.     for (int i = array.length - 2; i >= 0; i--) {</PRE><PRE class=java name="code"><SPAN style="W
    HITE-SPACE: pre"></SPAN> //当前数组值比最小数组值小，填充到最下数组中，否则当前数值
    位置的值的右边最小值就是上一个最小值
10.    if (array[i] < rightMin[i + 1]) {
11.        rightMin[i] = array[i];
12.    }
13. }
```

```

12.         } else {
13.             rightMin[i] = rightMin[i + 1];
14.         }
15.     }
16.     int leftMax = Integer.MIN_VALUE;
17.     for (int i = 0; i < array.length; i++) {
18.         if (leftMax <= array[i]) {
19.             leftMax = array[i];
20.             if (leftMax == rightMin[i]) {
21.                 System.out.println(leftMax);
22.             }
23.         }
24.     }
25. }</PRE><BR>
26. <BR>
27. <P></P>
28. <PRE></PRE>
29. <BR>
30. <BR>
31. <P></P>
```

1.8.9. 一个文件，内含一千万行字符串，每个字符串在 1K 以内，要求找出所有相反的串对，如 abc 和 cba。

文件的大小上限是 10G，不可能在内存操作了。考虑设计一种 hash 使得如果两个字符串维相反串能得出相同的 hash 值，然后用该 hash 将文件中的字符串散列到不同的文件中，再在各文件中进行匹配。比如这样的 hash 函数对字符串上所有字符的 ascii 求和，因为长度在 1K 以内，因此范围在 int 之内。更进一步，可以在上面那个 hash 后面再加一个字符串长度，可以得到更好的散列效果。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```

1.  ****
2.  /* 一个文件，内含一千万行字符串，每个字符串在 1K 以内，
```

```
3.    要求找出所有相反的串对，如 abc 和 cba。
4.    思路：使用 hash 表
5.                */
6.    /***** *****/
7.
8.
9.
10.   #include<iostream>
11.   #include<string>
12.   #include<malloc.h>
13.   #include<stdlib.h>
14.   #include<fstream>
15.   using namespace std;
16.
17.   #define ERROR 0
18.
19.   struct _LinkList//哈希表拉链结构
20.   {
21.       char *strValue;
22.       _LinkList *next;
23.   };
24.   void Reverse(char str[],int len)//字符串反转函数，对字符串进行反转时，要使用数据表示字符串
25.   {
26.       int start,end;
27.       start=0;
28.       end=len-1;
29.       while(start<end)
30.       {
31.           char temp;
32.           temp=str[start];
33.           str[start]=str[end];
34.           str[end]=temp;
35.
36.           start++;
37.           end--;
```

```
38.     }
39.
40. }
41. class CHash
42. {
43. private:
44.     _LinkList *List[350];//根据哈希地址构造拉链式哈希表
45. public:
46.     CHash();
47.     ~CHash();
48.     int HashFunc(char *str);
49.     void InitHash();
50. };
51. CHash::CHash()
52. {
53.     int i;
54.     for(i=0;i<320;i++)
55.     {
56.         List[i]=(_LinkList*)malloc(sizeof(_LinkList));//初始化
57.         if(!List[i])
58.             exit(ERROR);
59.         List[i]->next=NULL;
60.         List[i]->strValue=new char[10];
61.     }
62. }
63. CHash::~CHash()
64. {
65.
66. }
67. int CHash::HashFunc(char *str)//哈希函数，将字符串各字符对 90 取模，取模值为偶数乘权重 2，  
    将各值相加作为哈希地址
68. {
69.     char *p=str;
70.     int sum=0;
71.     while(*p!='\0')
```

```
72.    {
73.        int mod=(int)(*p)%90;
74.        if(mod%2==0)
75.            mod=mod*2;
76.        sum=sum+mod;
77.        p++;
78.    }
79.    return sum;
80. }
81. void CHash::InitHash()
82. {
83.     ifstream fin;
84.     ofstream fout;
85.     char str[10];
86.     char tempstr[10];
87.     fin.open("data.txt");
88.     if(fin.is_open())//从文件中读入字符串
89.     {
90.         while(fin>>str)
91.         {
92.
93.             int addr=HashFunc(str);
94.
95.             bool isRight=false;
96.             _LinkList *newNode,*head;
97.             newNode=(_LinkList*)malloc(sizeof(_LinkList));//新建一个结点
98.             if(!newNode)
99.                 exit(ERROR);
100.            newNode->next=NULL;
101.            newNode->strValue=new char[10];
102.            strcpy(newNode->strValue,str);//结点赋值
103.
104.            head=List[addr];
105.            strcpy(tempstr,str);//保存下原始字符串
106.            Reverse(str,strlen(str));//将字符串反转
```

```
107.  
108.     while(head->next)//遍历哈希表拉链  
109.     {  
110.         head=head->next;  
111.  
112.         if(strcmp(str,head->strValue)==0)//如果读入的字符串反转后与链表中字符串相等，则输出  
113.         {  
114.             cout<<head->strValue<<" "<<tempstr<<endl;  
115.             isRight=true;//标志位，字符串已处理  
116.             break;  
117.         }  
118.  
119.  
120.     }  
121.     if(!isRight)//如果读入的字符串反转后与链表中字符串不相等，则将新结点插入到链表中  
122.     {  
123.         head->next=newNode;  
124.         head=newNode;  
125.         head->next=NULL;  
126.     }  
127.  
128. }
```

129. }

```
130. fin.clear();  
131. fin.close();  
132. }  
133. int main()  
134. {  
135.     CHash CH=CHash();  
136.     CH.InitHash();  
137.     return 1;  
138. }  
139.  
140. 程序中为了验证方便，在文件字符串设定长度为 5;
```

STL 的 set 用什么实现的？为什么不用 hash？

是用红黑树实现的，红黑树是一种平衡性很好的二分查找树。要使用 hash 的话，就需要为不同的存储类型编写哈希函数，这样就照顾不到容器的模板性了，而是用红黑树只需要为不同类型重载 operator<就可以了。

1.8.10. 给出一个文件，里面包含两个字段{url、size}，即 url 为网址，size 为对应网址访问的次数

地址：<http://blog.csdn.net/zhangfei2018/article/details/7842192>

给出一个文件，里面包含两个字段{url、size}，

即 url 为网址，size 为对应网址访问的次数，

要求：

问题 1、利用 Linux Shell 命令或自己设计算法，

查询出 url 字符串中包含“baidu”子字符串对应的 size 字段值；

问题 2、根据问题 1 的查询结果，对其按照 size 由大到小的排列。

（说明：url 数据量很大，100 亿级以上）

题目大意：给出一个文件，里面包含两个字段{url、size}，url 即为网址，size 为网址对应访问的次数，要求：问题 1、利用 Linux Shell 命令或自己设计算法，查询出 url 字符串中包含“baidu”子字符串对应的 size 字段值；问题 2、根据问题 1 的查询结果，对其按照 size 由大到小的排列

面试时我采用的是自己设计算法，面试回来后我 man sort 查了排序命令的参数使用手册，下面我就详细讲一下用 Shell 命令的做法

分析题意： baidu.txt{url, size}，baidu.txt 是文件，{url, size}是文件中的两个字段，并且 url 和 size 都是字符串型，字段之间用 tab (/t) 隔开

第一步，查询匹配 url 字符串中的字串“baidu”，直接用 grep 命令，具体格式 grep "baidu" baidu.txt（每行仅 url 可能含有 baidu 子字符串）

第二步，显示含有字串“baidu”的 url 及其对应的 size，可以直接用 ls 命令，具体格式 ls -l | grep "baidu" baidu.txt（管道传值）

第三步，将步骤 2 的结果，通过重定向命令>>保存在 baidu2.txt 文件中，即 grep "baidu" baidu.txt >> baidu2.txt 保存匹配结果

第四步，排序，直接利用 sort 命令，即格式 ls -l | sort -rnk 2 baidu.txt (sort 反向 r、以第二个字段 k 2、数值型 n 进行排序)

第五步，将步骤 4 的结果，通过重定向命令>>保存在 baidu3.txt 文件中，即 sort -rnk 2 baidu2.txt >> baidu3.txt 保存排序结果

按照上面五步，我们先看结果，用数据说话，然后我将在下面依次详细介绍上面五步中用到的 Shell 命令及其参数的确切含义：

首先，新建 baidu.txt 文件，即用 Vim 编辑器输入创建的测试用例（我用过的主流搜索引擎及百度的部分产品），然后利用 cat 命令查看文件

其次，我们查询 url 字符串中包含子串"baidu"的项，并打印出 url 及其 size

然后，我们新建 baidu2.txt 文件，用于保存问题 1 的结果(即匹配字符串 url 中包含子串 "baidu" 的结果项 url 和 size)

接着，我们对上述问题 1 的结果，利用 sort 命令按照 size 由大到小进行排序

最后，我们新建 baidu3.txt 文件，用于保存 sort 排序结果

附图说明：

以上截图，均截自我电脑 Linux RedHat 5.2（安装在 VMWare 7.0 虚拟机上）

其中的 Shell 命令都在 Redhat Linux OS 环境已测试通过

=====

=====

好啦，今晚在 man sort 查看了 sort 详细参数使用方法后，似乎可以不用 awk 命令就可以搞定此题，看来并没有我面试时想得那么复杂。现在就让我们具体看看 sort 的参数以及 grep、重定向等 Shell 命令的详细使用方法吧

sort 命令

格式：sort 【参数】 【文件】

举例：sort -rnk 2 baidu.txt

参数：r 逆序； n 字符串按数值处理； k 2 表示第二个字段（列）

说明：在文件 baidu.txt 中，按照第二个字段的数值型由大到小进行排序

首先，在 Linux Shell 命令行界面输入 man sort 查看 sort 的帮助手册

然后，查看本题中，我们需要用到的三个参数 r n k 的详细使用方法

由 sort 帮助文档显示：

- 1、参数 n 是把 string 字符串转换成 numerical 的值 value 进行比较（即把字符串转换成数值型，再进行比较）
- 2、参数 r 是反向，即逆序。由于 sort 默认排序是由小到大，而题意需要从大到小排序，因此此处需要逆序（注意：v 在某些命令中也可表示反向，如正则表达式中）
- 3、参数 k 是字段分隔，即从哪个字段开始直到哪个字段结束，按其进行排序（注意：Linux Redhat 第一列从 1 开始，而不是 0，本题格式中 size 为第二列，因此我们 k 定为 2，而不是 1。区别于通常用的 Array 数组下限和 Python 语言中分组的下限，即从 0 开始）

grep 命令

格式： grep 【参数】 【查询字符串】 【文件】

举例： grep "baidu" baidu.txt

参数： 此处无参数（省略了参数）

说明： 查询匹配文件 baidu.txt 中，判断是否包含"baidu"子字符串

在 Linux Shell 命令行输入： man grep

管道| 命令

格式： 【命令 1】（目标） | 【命令 2】（源数据）

参数： 为进程通信，无参数

举例： ls -l | grep "baidu" baidu.txt

说明： 把文件 baidu.txt 中，包含"baidu"子字符串的结果，通过管道|命令，传给 ls -l 作为目标内容，进行显示

重定向>或>> 命令

格式： 【命令 1】（源数据）> 【命令 2】（目标） 或者 【命令 1】（源数据）>> 【命令 2】（目标）

参数： 为信道值，无参数

举例： sort -rnk 2 baidu.txt > baidu2.txt

说明： 把文件 baidu.txt 中，对第二个字段按照数值型进行由大到小的排序，并将结果保存到文件 baidu2.txt（清空后重写）

附图举例说明：先清空文件 baidu2.txt 中原有内容，然后再将 sort 结果重定向保存到 baidu2.txt 文件中

附图举例说明：先并未清空文件 baidu2.txt 中原有内容，而是直接追加 sort 结果，重定向保存到 baidu2.txt 文件中（保留 baidu2.txt 原有内容）

ls touch cat 等其它基本命令

第二步，查询匹配 url 字符串中是否含有字串"baidu"的另一种做法（在网友 [showmsg](#) 的提示下，改用 awk 命令代替 grep 命令进行正则匹配）

当然，我们还是先看结果，然后我再介绍 awk 命令的使用方法

首先，我们查询并打印出含有"baidu"字串的 url 及其 size

其次，我们对查询含有"baidu"的结果，对 size 进行有大到小的排序

最后，保存查询匹配"baidu"并对 size 由大到小排序后的结果到 baidu2.txt 文件中（重定向）

附加 1，如果只想按照 size 由大到小打印出 url（即不打印 size，也就是分离字段），则如下

附加 2，如果只想按照 size 由大到小打印出 size（即不打印 url，也就是分离字段），则如下



这便是我面试时想用 awk 命令的解法，不过当时我只知道此命令功能但没具体用过，今在网友 [showmsg](#) 指点下，总算略通一二，对此表示谢意^ ^

awk 命令

格式：awk commands file 或者 awk script-file file

参数：print 打印； \$1 第一个字段； \$2 第二个字段； '' 单引号需加上，并可写入正则表达式，如查询 baidu 字串

举例：awk '/baidu/ && \$2<800' baidu.txt

说明：查询文件 baidu.txt 中满足第一字段含有"baidu"字串并且第二个字段数字 size<800 的所有记录，并显示出结果

附图 1：查询文件 baidu.txt 中满足第一字段含有"baidu"字串并且第二个字段数字 size<800 的所有记录，并显示出结果

附图 2：查询文件 baidu.txt 中满足第一字段含有"baidu"字串并且第二个字段数字 size<800 的所有记录，并打印出结果的第一个字段（url）

1.9. 面试题集合（八）

1.9.1. 给定一个存放整数的数组，重新排列数组使得数组左边为奇数，右边为偶数

```
1. [cpp] view plaincopyprint?
2. /*
3. 给定一个存放整数的数组，重新排列数组使得数组左边为奇数，右边为偶数。
4. 要求：空间复杂度 O(1)，时间复杂度为 O(n)。
5. /**
6.
7. #include <iostream>
8. #include <iomanip>
9. #include <limits>
10.
11. using namespace std;
12.
13. void swap_int(int& a, int& b)
14. {
15.     int t = a;
16.     a = b;
17.     b = t;
18. }
19.
20. int main()
21. {
22.     int numel[] = {1, 23, 2, 34, 21, 45, 26, 22, 41, 66, 74, 91, 17, 64};
23.     int sz = sizeof(numel)/sizeof(numel[0]);
24.
25.     for(int i=0; i<sz; ++i){
26.         cout << numel[i] << " ";
27.     }
28.     cout << endl;
29.
30.     int begin = 0;
31.     int end = sz -1;
32.     while(begin < end){
```

```
33.     while(numel[begin]%2 == 1 && end > begin){
34.         ++begin;
35.     }
36.     while(numel[end]%2 == 0 && end > begin){
37.         --end;
38.     }
39.     swap_int(numel[begin], numel[end]);
40. }
41.
42. for(int i =0; i<sz; ++i){
43.     cout << numel[i] << " ";
44. }
45. cout << endl;
46.
47. return 0;
48. }
```

1.9.2. 用 C 语言实现函数 void * memmove(void *dest,const void *src,size_t n)

由于可以把任何类型的指针赋给 void 类型的指针

这个函数主要是实现各种数据类型的拷贝。

用 C 语言实现函数 void * memmove(void *dest,const void *src,size_t n)。

memmove 函数的功能是拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址上。

考虑到内存可能重叠的情况，要在函数中避免有以下方式

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. void *memmove(void * dest, void *src, size_t n)
2. {
3.     void *tmp;
4.     int p;
5.     if ((tmp=malloc(n))==NULL)
6.         return NULL;
7.     for(p=0;p<n;p++)
```

```
8.     *((char *)tmp+p)=*((char *)src+p);
9.     for(p=0;p<n;p++)
10.        *((char *)dest+p)=*((char *)tmp+p);
11.     return dest;
12. }
```

[cpp] [view plain](#) [copy](#) [print?](#)

```
1. #include "stdio.h"
2.
3.
4. void* memmove(void *dest, const void* src, size_t n)
5. {
6.     if(dest == NULL || src == NULL)
7.         return NULL;
8.     char* p =(char*) dest;
9.     char* q =(char*) src;
10.    while(n--)
11.    {
12.        *p++ = *q++;
13.    }
14.    return dest;
15. }
16. int main()
17. {
18.     char* p = "hello,world";
19.     char dest[6] = {0};
20.     char *q = (char*)memmove(dest,p,5);
21.     printf("%s\n",dest);
22.     printf("%s\n",q);
23.     return 0;
24. }
```

1.9.3. 随机发生器

已知一随机发生器，产生 0 的概率是 p ，产生 1 的概率是 $1-p$ ，现在要你构造一个发生器，使得它构造 0 和 1 的概率均为 $1/2$ ；构造一个发生器，使得它构造 1、2、3 的概率均为 $1/3$ ；…，构造一个发生器，使得它构造 1、2、3、… n 的概率均为 $1/n$ ，要求复杂度最低。

首先是 $1/2$ 的情况，我们一次性生成两个数值，如果是 00 或者 11 丢弃，否则留下，01 为 1，10 为 0，他们的概率都是 $p*(1-p)$ 是相等的，所以等概率了。

然后是 $1/n$ 的情况了，我们以 5 为例，此时我们取 $x=2$ ，因为 $C(2x,x)=C(4,2)=6$ 是比 5 大的最小的 x ，此时我们就是一次性生成 4 位二进制，把 1 出现个数不是 2 的都丢弃，这时候剩下六个：0011,0101,0110,1001,1010,1100，取最小的 5 个，即丢弃 1100，那么我们对于前 5 个分别编号 1 到 5，这时候他们的概率都是 $p*p*(1-p)*(1-p)$ 相等了。

关键是找那个最小的 x ，使得 $C(2x,x) \geq n$ 这样能提升查找效率。

因为 $C(n,i)$ 最大是在 i 接近 $n/2$ 的地方取得，此时我有更大比率的序列用于生成，换句话说被抛掉的更少了，这样做是为了避免大量生成了丢弃序列而使得生成速率减慢，实际上我之所以将 x 取定是为了让我取得的序列生成的概率互相相等，比如 $C(2x,x)$ 的概率就是 $[p(1-p)]^x$ ，互等的样例空间内保证了对应的每个值取得的样例等概率。

使用已知的随机发生器构造 n 个数，则其中有 k 个 1 的概率是 $n!/(k!(n-k)!)*p^k * (1-p)^{n-k}$

1.9.4. 搜索引擎

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。假设目前一个日志文件中有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

1000 万条记录，每条记录最大为 255Byte，那么日志文件最大有 2.5G 左右，大于 1G 内存。但是题目中又提到这样的 1000 万条记录中有许多是重复的，出去重复的话只有 300 万条记录，存储这样的 300 万条记录需要 0.75G 左右的内存，小于 1G 内存。那么我们可以考虑将

这些无重复的记录装入内存，这是我们需要一种数据结构，这种数据结构即能够存储查询串，又能存储查询串的出现次数，我们可以通过 hashmap<query,count>来保存。读取文件，创建一个 hashmap，如果 hashmap 中存储了遍历到的 query，则修改该 query 所对应的 count 值，使其+1；如果 hashmap 中没有这个 query，那么往 hashmap 中插入<query,1>。这样我们就创建好了一个包含所有 query 和次数的 hashmap。

然后我们创建一个长度为 10 最大堆 MaxHeap，遍历 hashmap，如果 MaxHeap 未满，那么往 MaxHeap 中插入这个键值对，如果 MinHeap 满了，则比较遍历到的元素的 count 值堆顶的 count，如果遍历到元素的 count 大于堆顶 count 值，删除堆顶元素，插入当前遍历到的元素。遍历完整个 hashmap 以后，在 MaxHeap 中存储的就是最热门 10 个查询串。

百度面试题：将 query 按照出现的频度排序（10 个 1G 大小的文件）。有 10 个文件，每个文件 1G，每个文件的每一行都存放的是用户的 query，每个文件的 query 都可能重复。如何按照 query 的频度排序？

网上给出的答案：

1) 读取 10 个文件，按照 hash(query)%10 的结果将 query 写到对应的 10 个文件

(file0,file1,...,file9) 中，这样的 10 个文件不同于原先的 10 个文件。这样我们就有 10 个大小约为 1G 的文件。任意一个 query 只会出现在某个文件中。

2) 对于 1) 中获得的 10 个文件，分别进行如下操作

- 利用 hash_map (query, query_count) 来统计每个 query 出现的次数。
- 利用堆排序算法对 query 按照出现次数进行排序。
- 将排序好的 query 输出的文件中。

这样我们就获得了 10 个文件，每个文件中都是按频率排序好的 query。

3) 对 2) 中获得的 10 个文件进行归并排序，并将最终结果输出到文件中。

注：如果内存比较小，在第 1) 步中可以增加文件数。

统计外站的搜索关键词的词频

通过外站的链接主要是百度，谷歌，soso 等，每天都有通过记录在日志文件中，每天会运行程序进行统计。

每天产生有 10 多个文件，每个文件 1G 左右，每个文件的每一行都存放的是用户的 query，每个文件的 query 都可能重复。要按照解析 query 中的关键词，并对统计其频度，取出搜索次数最多的前 1000 个关键词。

第一次直接遍历所有文件并按照 Map<String, Integer>方式来统计，统计差不多共有四千万条

记录，词也有一百万多个，最后排序实现。方法简单，但也有很大的缺点，占用的内存太大，可能会将服务器弄垮掉。

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 初始写文件流
3. */
4. public void init() {
5.     if (numFiles < 1 && numFiles > 1000) {
6.         throw new RuntimeException("中间保存搜索关键词的文件数目不能小于 1 或大于 1000");
7.     }
8.     outs = new BufferedWriter[numFiles];
9.     outFiles = new File[numFiles];
10.    File tempDir = new File("temp");
11.    if (!tempDir.exists()) {
12.        tempDir.mkdir();
13.    }
14.    for (int i = 0; i < numFiles; i++) {
15.        try {
16.            outFiles[i] = new File("temp/" + String.valueOf(i));
17.            outFiles[i].createNewFile();
18.            outs[i] = new BufferedWriter(new FileWriter(outFiles[i]));
19.        } catch (IOException e) {
20.            throw new RuntimeException(e);
21.        }
22.    }
23. }
```

先不统计，利用外存处理，将分析出来的每一个关键词，取出第一个字符的哈希值%500，分别放进 500 个临时文件中，然后才对每个文件进行统计。

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 记录关键词到相应的文件
3. */
```

```
4.     * @param keyword
5.     */
6.     public void addKeyword(String keyword) {
7.         // 写入不同的外存 (文件)
8.         if (keyword.length() < 1)
9.             return;
10.        String firstStr = keyword.substring(0, 1);
11.        int index = firstStr.hashCode() % numFiles;
12.        index = Math.abs(index);
13.        try {
14.            outs[index].write(key word + lineSep);
15.        } catch (IOException e) {
16.            logger.error("", e);
17.        }
18.    }
```

由于只取前 1000 个关键词，所以每统计完一个文件后，采用堆排序，将前 1000 名的关键词保存在一个堆里，这样内存中只维护一份堆，可以减少许多内存的消耗，只占用到之前的五分之一以下。速度较之前也提高不少。

[java] [view plain](#) [copy](#) [print](#)?

```
1.     public List<Keyword> sortKeywordMap(int size) {
2.         final KeywordQueue queue = new KeywordQueue(size);
3.         try {
4.             TObjectIntHashMap<String> key2numMap = new TObjectIntHashMap<String>();
5.             // 统计所有文件
6.             for (int i = 0; i < numFiles; i++) {
7.                 // 对单个文件统计
8.                 LineIterator lines = FileUtils.lineIterator(outFiles[i]);
9.                 while (lines.hasNext()) {
10.                     String line = lines.nextLine();
11.                     int num = key2numMap.get(line);
12.                     num++;
13.                     key2numMap.put(line, num);
14.                 }
15.             }
16.             // 堆排序
17.             Queue<Keyword> queue = new PriorityQueue<Keyword>(size, new Comparator<Keyword>() {
18.                 public int compare(Keyword o1, Keyword o2) {
19.                     return o1.getNum() - o2.getNum();
20.                 }
21.             });
22.             for (int i = 0; i < size; i++) {
23.                 queue.add(key2numMap.getEntry(i));
24.             }
25.             List<Keyword> result = new ArrayList<Keyword>(size);
26.             for (int i = 0; i < size; i++) {
27.                 result.add(queue.remove());
28.             }
29.             return result;
30.         } catch (Exception e) {
31.             logger.error("Error in sorting keywords", e);
32.         }
33.     }
```

```

15.         lines.close();
16.
17.         // 将统计完的 map 中的所有关键词放入最小堆中
18.         key2numMap.forEachEntry(new TObjectIntProcedure<String>() {
19.
20.             @Override
21.
22.             public boolean execute(String key, int num) {
23.
24.                 Keyword k = new Keyword(key, num);
25.
26.                 queue.insertWithOverflow(k);
27.
28.                 return true;
29.             }
30.
31.         });
32.         // 进行清除
33.         key2numMap.clear();
34.     }
35. }
36. } catch (Exception e) {
37.     logger.error("", e);
38. }
39. List<Keyword> list = new ArrayList<Keyword>();
40. for (int i = queue.size() - 1; i >= 0; i--) {
41.     list.add(queue.pop());
42. }
43. Collections.reverse(list);
44. return list;
45. }

```

1.9.5. 已知一个字符串，比如 asderwsde,寻找其中的一个子字符串比如 sde 的个数，如果没有返回 0，有的话返回子字符串的个数

```

1. /*
2.      已知一个字符串，比如 asderwsde,寻找其中的一个子字符串比如 sde 的个数，如果没有
3.      返回 0，有的话返回子字符串的个数。
4.
5. #include <iostream>

```

```
6. #include <iomanip>
7. #include <limits>
8.
9. using namespace std;
10. bool matchsub(char* pchar, char* schar, int pos);
11. int main()
12. {
13.     char pchar[] = "asderwsde";
14.     char schar[] = "sde";
15.
16.     int cnt = 0;
17.     int pos = 0;
18.     int psz = sizeof(pchar);
19.     int ssz = sizeof(schar);
20.
21.     while(pos < psz - ssz){
22.         while(pchar[pos] != schar[0]){
23.             ++pos;
24.         }
25.         if(pos > psz - ssz){
26.             break;
27.         }else{
28.             if(matchsub(pchar, schar, pos)){
29.                 ++cnt;
30.                 ++pos;
31.             }else{
32.                 ++pos;
33.             }
34.         }
35.     }
36.
37.     if(cnt > 0){
38.         cout << cnt << " substring found!" << endl;
39.     }
40.
```

```
41.  
42.    return 0;  
43. }  
44.  
45. bool matchsub(char* pchar, char* schar, int pos)  
46. {  
47.    bool flag = true;  
48.    int i = 0;  
49.    while(schar[i] != '\0' && pchar[pos+i] != '\0') {  
50.        if(pchar[pos+i] != schar[i]) {  
51.            flag = false;  
52.            break;  
53.        }  
54.        ++i;  
55.    }  
56.    return flag;  
57. }
```

1.9.6. 编写一个程序，把一个有序整数数组放到二叉树中

分析:本题考察二叉搜索树的建树方法，简单的递归结构。

关于树的算法设计一定要联想到递归，因为树本身就是递归的定义。

而，学会把递归改称非递归也是一种必要的技术。

毕竟，递归会造成栈溢出，关于系统底层的程序中不到非不得以最好不要用。

但是对某些数学问题，就一定要学会用递归去解决。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <stdio.h>  
2. #include <stdlib.h>  
3.  
4. struct student {  
5.     int value;  
6.     struct student *lchild;
```

```
7.     struct student *rchild;
8. };
9.
10. void arraytotree(int *a, int len, struct student **p) {
11.     if(len) {
12.         *p = (struct student*)malloc(sizeof(struct student));
13.         (*p)->value = a[len/2];
14.         arraytotree(a, len/2, &(*p)->lchild);
15.         arraytotree(a+len/2+1, len-len/2-1, &(*p)->rchild));
16.     } else {
17.         *p = NULL;
18.     }
19. }
20.
21. void display_tree(struct student *head) {
22.     if(head->lchild)display_tree(head->lchild);
23.     printf("%d\t", head->value);
24.     if(head->rchild)display_tree(head->rchild);
25. }
26.
27. int main() {
28.
29.     int a[] = { 1,2,3,4,9,10,33,56,78,90};
30.     struct student *tree;
31.     arraytotree(a, sizeof(a)/sizeof(a[0]), &tree);
32.     printf("After convert:\n");
33.     display_tree(tree);
34.     printf("\n");
35.     return 0;
36.
37. }
```

1.9.7. 大整数数相乘的问题

```
1. void Multiple(char A[], char B[], char C[]) {  
2.  
3.     int TMP, In=0, LenA=-1, LenB=-1;  
4.  
5.     while(A[++LenA] != '\0');  
6.  
7.     while(B[++LenB] != '\0');  
8.  
9.     int Index, Start = LenA + LenB - 1;  
10.  
11.    for(int i=LenB-1; i>=0; i--) {  
12.  
13.        Index = Start-;  
14.  
15.        if(B[i] != '0') {  
16.  
17.            for(int In=0, j=LenA-1; j>=0; j--) {  
18.  
19.                TMP = (C[Index]-'0') + (A[j]-'0') * (B[i] - '0') + In;  
20.  
21.                C[Index--] = TMP % 10 + '0';  
22.  
23.                In = TMP / 10;  
24.  
25.            }  
26.  
27.            C[Index] = In + '0';  
28.  
29.        }  
30.  
31.    }  
32.  
33. }
```

```
34.  
35.  
36.  
37. int main(int argc, char* argv[]) {  
38.  
39.     char A[] = "218392444444444888008888889";  
40.  
41.     char B[] = "3888888888899999999999988";  
42.  
43.     char C[sizeof(A) + sizeof(B) - 1];  
44.  
45.  
46.  
47.     for(int k=0; k<sizeof(C); k++)  
48.  
49.         C[k] = '0';  
50.  
51.     C[sizeof(C)-1] = '\0';  
52.  
53.  
54.  
55.     Multiple(A, B, C);  
56.  
57.     for(int i=0; C[i] != '\0'; i++)  
58.  
59.         printf("%c", C[i]);  
60.  
61. }
```

1.9.8. 求最大连续递增数字串

1. 求最大连续递增数字串（如“ads3sl456789DF3456ld345AA”中的“456789”）

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```

1. int GetSubString(char *strSource, char *strResult) {
2.     int iTmp=0, iHead=0, iMax=0;
3.     for(int Index=0, iLen=0; strSource[Index]; Index++) {
4.         if(strSource[Index] >= '0' && strSource[Index] <= '9' &&
5.             strSource[Index-1] > '0' && strSource[Index] == strSource[Index-1]+1) {
6.             iLen++;           // 连续数字的长度增 1
7.         } else {           // 出现字符或不连续数字
8.             if(iLen > iMax) {
9.                 iMax = iLen; iHead = iTmp;
10.            }
11.            // 该字符是数字, 但数字不连续
12.            if(strSource[Index] >= '0' && strSource[Index] <= '9') {
13.                iTmp = Index;
14.                iLen = 1;
15.            }
16.        }
17.    }
18.    for(iTmp=0 ; iTmp < iMax; iTmp++) // 将原字符串中最长的连续数字串赋值给结果串
19.        strResult[iTmp] = strSource[iHead++];
20.        strResult[iTmp]='\0';
21.    return iMax; // 返回连续数字的最大长度
22. }
23. int main(int argc, char* argv[]) {
24.     char strSource[]="ads3sl456789DF3456ld345AA", char strResult[sizeof(strSource)];
25.     printf("Len=%d, strResult=%s \nstrSource=%s\n",
26.     GetSubString(strSource, strResult), strResult, strSource);
27. }
```

1.9.9. 函数将字符串中的字符'*'移到串的前部分

2005 年 11 月金山笔试题。编码完成下面的处理函数。

函数将字符串中的字符'*'移到串的前部分，

前面的非'*'字符后移，但不能改变非'*'字符的先后顺序，函数返回串中字符'*'的数量。

如原始串为： ab**cd**e*12，

处理后为*****abcde12， 函数并返回值为 5。 (要求使用尽量少的时间和辅助空间)

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. int movStar(char * p, int n)
2. {
3.     char * q1 = p+(n-1), *q2 = p+(n-1);
4.     while(q1>=p)
5.     {
6.         if(*q1 != '*')
7.             *q2-- = *q1;
8.         q1--;
9.     }
10.    int nRet = q2 - q1;
11.    while(q2>=p)
12.    {
13.        *q2-- = '*';
14.    }
15.    return nRet;
16. }
```

1.9.10. 单链表，编程实现其逆转

给定一个单链表，编程实现其逆转。单链表的逆转过程的关键是 3 个指针：一个记录当前逆转节点的前一个节点，一个记录当前逆转节点的下一个节点，另外一个则记录当前的逆转节点。为什么要 3 个指针呢？其原因就在于当修改了当前逆转节点的 next 域后（逆转），链表就暂时断了（即无法再定位下一个节点），因此需要一个指针记录其下一个节点位置（要能够依次遍历）。记录前一个节点位置则很明显：因为要将当前逆转节点的 next 域指向其上一个节点（这样才叫逆转）。

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. //逆转单链表
2. struct node *reverse_LList(node *head)
```

```
3.  {
4.    //以 p 作为当前结点处理,头结点为逆转前的头结点
5.    struct node *p,*q,*temp;
6.    if(head == NULL)
7.      cout << "空表" << endl;//头结点为 0,空表
8.    q = head;      //q 指向头结点
9.    p = head->next; //p 指向头结点后一个结点
10.   head = p;     //头结点进一位
11.   temp = p->next; //保存当前结点指针域
12.   q->next = NULL; //逆转头结点为尾结点的情况,头指针为空
13.   p->next = q;     //后一个结点指向头结点
14.
15.   while(temp->next != NULL) //判断当前结点的后一结点指针域是否为空,即判断是否为尾结
点
16.   {
17.     q = head;      //q 指向头结点
18.     p = temp;      //p 指向头结点后一个结点
19.     temp = p->next; //保存当前结点指针域
20.     head = p;     //头结点进一位
21.     p->next = q; //后一个结点指向头结点
22.   }
23. //此时 temp 表示以前的尾结点,p 表示 temp 结点的前一结点
24. head = temp; //逆转尾结点为头结点
25. head->next = p; //头结点指针域指向 p
26. return head;
27.
28. }
```

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. Node* LinkListReserve(LinkNode head)
2.
3. {
4.
```

```
5.     LinkNode p = NULL; //记录逆转节点的前一个节点;
6.
7.     LinkNode r = head; //记录当前节点;
8.
9.     LinkNode q = NULL; //记录逆转节点的下一个节点;
10.
11.
12.    while (r != NULL)
13.
14.    {
15.
16.        q = r->_next; //保存下一个节点
17.
18.        r->_next = p; //逆转
19.
20.        p = r;      //下一次遍历
21.
22.        r = q;
23.
24.    }
25.    return p;
26.
27. }
```

1.10.面试题集合（九）

1.10.1. 删除字符串中的数字并压缩字符串

删除字符串中的数字并压缩字符串（神州数码以前笔试题），如字符串”abc123de4fg56”处理后变为”abcdefg”。注意空间和效率。（下面的算法只需要一次遍历，不需要开辟新空间，时间复杂度为 O(N)）

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include "stdafx.h"
```

```

2. void delNum(char *str) {
3.     int i, j=0;
4.     // 找到串中第一个数字的位子
5.     for(i=j=0; str[i] && (str[i]<'0' || str[i]>'9'); j=++i);
6.
7.     // 从串中第一个数字的位置开始，逐个放入后面的非数字字符
8.     for(; str[i]; i++)
9.         if(str[i]<'0' || str[i]>'9')
10.            str[j++] = str[i];
11.            str[j] = '\0';
12.    }
13.
14. int main(int argc, char* argv[]) {
15.     char str[] = "abc123ef4g4h5";
16.     printf("%s\n", str);
17.     delNum(str);
18.     printf("%s\n", str);
19. }
```

1.10.2. 求两个串中的第一个最长子串（神州数码以前试题）

```

1. #include <iostream>
2. using namespace std;
3. const int N = 1000;
4.
5. char* FirstMaxSubString(const char *str1,char *str2)
6. {
7.     int pos; //存放第一个最长子串的起始位置
8.     int max = 0; //第一个最长子串的长度
9.     int i,j;
10.    for(i=0;str1[i];i++)
11.    {
12.        for(j=0;str2[j];j++)
13.        {
```

```
14.     for(int k=0;str1[i+k]==str2[j+k] && (str1[i+k] || str2[i+k]);k++)
15.         if(k>max)
16.             {
17.                 pos = j;
18.                 max = k+1;
19.             }
20.         }
21.     }
22.
23.     char *result = new char[max+1];
24.     for(i=0;i<max;i++)
25.         result[i] = str2[pos++];
26.     result[i] = '\0';
27.     return result;
28. //或者直接用下面的语句返回，好处是不用申请空间
29. /*str2[pos+max] = '\0';
30. return (char *) (str2+pos);*/
31. }
32.
33. int main()
34. {
35.     char *str1 = new char[N];
36.     char *str2 = new char[N];
37.     cout<<"string_1:";
38.     cin.getline(str1,N);
39.     cout<<"string_2:";
40.     cin.getline(str2,N);
41. //固定测试例
42. /*
43.     char *str1 = "abracadabra";
44.     char *str2 = "dabdab";
45. */
46.     cout<<"FirstMaxSubString:"<<FirstMaxSubString(str1,str2)<<endl;
47.     return 0;
48. }
```

1.10.3. 不开辟用于交换数据的临时空间，如何完成字符串的逆序

不开辟用于交换数据的临时空间，如何完成字符串的逆序(在技术一轮面试中，有些面试官会这样问)

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. #include "stdafx.h"
2. void change(char *str) {
3.     for(int i=0,j=strlen(str)-1; i<j; i++, j-){
4.         str[i] ^= str[j] ^= str[i] ^= str[j];
5.         /*      s[i] = s[i]^s[j];
6.
7.         s[j] = s[i]^s[j];
8.
9.         s[i] = s[i]^s[j];*/
10.
11.
12.    }
13. }
14. int main(int argc, char* argv[]) {
15.     char str[] = "abcdefg";
16.     printf("strSource=%s\n", str);
17.     change(str);
18.     printf("strResult=%s\n", str);
19.     return getchar();
20. }
```

1.10.4. 求随机数构成的数组中找到长度大于=3 的最长的等差数列

```
1. @SuppressWarnings("unchecked")
2. private static void getArithmeticProgression(int[] array) {
3.     /* 排序 */
4.     Arrays.sort(array);
```

```
5.     /* 存放最大等差数列的容器 */
6.     ArrayList<Integer> maxList = new ArrayList<Integer>();
7.     /* 存放等差数列的临时容器 */
8.     ArrayList<Integer> tmpList = new ArrayList<Integer>();
9.     int len = array.length;
10.    for (int i = 0; i < len; i++) {
11.        for (int j = i + 1; j < len; j++) {
12.            /* 清空 list */
13.            tmpList.clear();
14.            /* 添加数列的第一个元素 */
15.            tmpList.add(i);
16.            /* 添加数列的第二个元素 */
17.            tmpList.add(j);
18.            /* 当前数列的元素个数 */
19.            int num = 2;
20.            /* 等差 */
21.            int progression = array[j] - array[i];
22.            /* 当前元素位置 */
23.            int current = j;
24.            /* 下一个元素位置 */
25.            int next = j + 1;
26.            /* 从 current 开始遍历 */
27.            while (next < len) {
28.                /*下一个元素和当前元素是等差数列*/
29.                if (array[next] - array[current] == progression) {
30.                    tmpList.add(next);
31.                    current = next;
32.                    num++;
33.                }
34.                next++;
35.            }
36.            /* 设定最大等差数列 */
37.            if (num >= maxList.size()) {
38.                maxList = (ArrayList<Integer>)tmpList.clone();
39.            }

```

```
40.    }
41.    }
42.    /* 输出结果 */
43.    for (Integer o : maxList) {
44.        System.out.println(array[o]);
45.    }
46. }
```

1.10.5. 外排序

外排序（External sorting）是指能够处理极大量数据的排序算法。通常来说，外排序处理的数据不能一次装入内存，只能放在读写较慢的外存储器（通常是硬盘）上。外排序通常采用的是一种“排序-归并”的策略。在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件。尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

外排序的一个例子是外归并排序（External merge sort），它读入一些能放在内存内的数据量，在内存中排序后输出为一个顺串（即是内部数据有序的临时文件），处理完所有的数据后再进行归并。[1][2]比如，要对 900 MB 的数据进行排序，但机器上只有 100 MB 的可用内存时，外归并排序按如下方法操作：

外归并排序

读入 100 MB 的数据至内存中，用某种常规方式（如快速排序、堆排序、归并排序等方法）在内存中完成排序。

将排序完成的数据写入磁盘。

重复步骤 1 和 2 直到所有的数据都存入了不同的 100 MB 的块（临时文件）中。在这个例子中，有 900 MB 数据，单个临时文件大小为 100 MB，所以会产生 9 个临时文件。

读入每个临时文件（顺串）的前 10 MB ($= 100 \text{ MB} / (9 \text{ 块} + 1)$) 的数据放入内存中的输入缓冲区，最后的 10 MB 作为输出缓冲区。（实践中，将输入缓冲适当调小，而适当增大输出缓冲区能获得更好的效果。）

执行九路归并算法，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。直至所有数据归并完成。

为了增加每一个有序的临时文件的长度，可以采用置换选择排序（Replacement selection sorting）。它可以产生大于内存大小的顺串。具体方法是在内存中使用一个最小堆进行排序，设该最小堆的大小为 。算法描述如下：

初始时将输入文件读入内存，建立最小堆。

将堆顶元素输出至输出缓冲区。然后读入下一个记录：

若该元素的关键码值不小于刚输出的关键码值，将其作为堆顶元素并调整堆，使之满足堆的性质；

否则将新元素放入堆底位置，将堆的大小减 1。

重复第 2 步，直至堆大小变为 0。

此时一个顺串已经产生。将堆中的所有元素建堆，开始生成下一个顺串。[\[3\]](#)

此方法能生成平均长度为 的顺串，可以进一步减少访问外部存储器的次数，节约时间，提高算法效率。

附加的步骤

上述例子的外排序有两个步骤：排序和归并。我们用一次多路归并就完成了所有临时文件的归并，而并非按内存中的二路归并那样，一次归并两个子串，耗费 次归并。外排序中不适用上述方法的原因在于每次读写都需要对硬盘进行读写，而这时非常缓慢的。所以应该尽可能减小磁盘的读写次数。

不过，在上述方法中也存在权衡。当临时文件（顺串）的数量继续增大时，归并时每次可从顺串中读入的数据减少了。比如说，50 GB 的数据量，100 MB 的可用内存，这种情况下用一趟多路归并就显得不划算。读入很多的顺串花费的时间占据了排序时间的大部分。

这时，我们可以用多次（比如两次）归并来解决这个问题。

这时排序算法变为下述这样：

第一步不变。

将小的顺串合并为大一些的顺串，适当减小顺串的数目。

将剩余的大一些的顺串归并为最终结果。

和内排序一样，高效的外排序所耗的时间依然是 。若利用好现在计算机上 GB 的内存，可使时间复杂度中的对数项增长比较缓慢。

优化性能

计算机科学家吉姆 格雷的 **Sort Benchmark** 网站用不同的硬件、软件环境测试了实现方法不同的多种外排序算法的效率。效率较高的算法具有以下的特征：

并行计算

用多个磁盘驱动器并行处理数据，可以加速顺序磁盘读写。[\[4\]](#)

在计算机上使用多线程，可在多核心的计算机上得到优化。

使用异步输入输出，可以同时排序和归并，同时读写。

使用多台计算机用高速网络连接，分担计算任务。[\[5\]](#)

提高硬件速度

增大内存，减小磁盘读写次数，减小归并次数。

使用快速的外存设备，比如 15000 RPM 的硬盘或固态硬盘。

使用性能更优良个各种设备，比如使用多核心 CPU 和延迟时间更短的内存。

提高软件速度

对于某些特殊数据，在第一阶段的排序中使用基数排序。

压缩输入输出文件和临时文件。

1.10.6. 用递归的方法判断整数组 a[N]是不是升序排列

```
/*用递归的方法判断整数组 a[N]是不是升序排列
*/
int isAscending(int a[], int length){
    if(length==1) return 1;
    if (a[length-1]>a[length])
    {
        return 0;
    }else
    {
        return isAscending(a,length-1);
    }
}
```

1.10.7. N 个鸡蛋放到 M 个篮子中， 篮子不能为空

```
1. 分析：此题和前面有一题
2. N 个整数(1,2,3...N)中任取 K 个，使得和为 M，求出所有情况
3. 比较类似，但是也有所不同，首先每个篮子放得鸡蛋理论上最多可以到 N，其实每个篮
子都必须有鸡蛋
4.
5.
6.
7. package com.java.ly2011.Semptember;
8.
9. /**
10. * n 个鸡蛋 m 个篮子 每个篮子必须有鸡蛋
11. * @author liuyang
12. *
13. */
14. public class N_eggs_M_baskets {
15.     public static void main(String[] args) {
16.         //getResult(6, 3);
17.     }
```

```
18.    easyWay(6, 3, 1, new StringBuffer());
19.
20. }
21.
22. public static void easyWay(int N , int M ,int start ,StringBuffer path){
23.     if(N==0)
24.         System.out.println( path );
25.
26.     if(start<=N&&M==1){
27.         path.append(N);
28.         System.out.println( path );
29.         path.setLength(path.length()-1);
30.         return;
31.     }
32.
33.     for(int i = start; i<=N-1; i++){//此处 i<=N-1 的意思就是最多取到 N-1 因为如果你取满了,
   后面的就必然有空的,此题一定要 N 个篮子都用上
34.             //取 N 的话代表的意思就是最多 N 个篮子 , 可以有的不用(不放鸡蛋)
35.
36.             easyWay(N-i, M-1, i, path.append(i));
37.             path.setLength(path.length()-1);
38.
39.     }
40. }
41.
42.
43. public static void getResult(int N, int M){
44.
45.     getSolution(N-M, M, 1, new StringBuffer());
46.
47. }
48.
49. public static void getSolution(int N , int M ,int start ,StringBuffer path){
50.
51.     if(N==0){
```

```
52.    StringBuffer newpath = new StringBuffer(path);
53.    for(int j = 1;j<=M;j++)
54.        newpath.append(0);
55.    for(int i = 0; i<newpath.length();i++)
56.        System.out.print( ((newpath.charAt(i)-'0')+1) + " ");
57.    System.out.println();
58.
59.    return;
60. }
61.
62.
63.    for(int i= start;i<=N;i++){
64.        if(M==1){
65.            path.append(N);
66.            for(int j = 0; j<path.length();j++)
67.                System.out.print( ((path.charAt(j)-'0')+1)+ " ");
68.            System.out.println();
69.            path.setLength(path.length()-1);
70.            break;
71.
72.        }
73.        else{
74.            getSolution(N-i, M-1, i, path.append(i));
75.            path.setLength(path.length()-1);
76.        }
77.    }
78. }
79. }
```

1.10.8. Hash

虽然很想很早就想写一个 hash 表，但一直都未去实现。通过这次机会，算是对 hash 表有了一个比较直观的了解，主要有以下几点（都是个人见解）：

1. 哈希表的目的在于加快查找速度，用一个形象的比喻就是 hash 是将一个排好序的数据存

入 数组中，所以在查找时能通过这个索引迅速找到所需要的元素，在 hash 表中，数组才是主体，链表只是辅助，甚至可以不存在。

2.产生这个索引（在 hash 中是 key）的函数和方法各种各样，而判别这个方法优劣就是让其尽可能少得产生冲突，因为产生冲突后，就会调用处理冲突的方法，无论哪一种方法都不是很合适，以链表为例，显然链表是不适合查找的，所以这个方法对表性能的影响是很大的。这里我才用了和系统差不多的方法，采用了&运算，其实类似于 mod，但速度更快。

3.处理冲突的方法也有不少。其中我比较明白的是再散列法和拉链法。在散列法的思想很简单，就是产生冲突后重新计算 hash 函数地址（索引值）直到找到空的空间放数据。而拉链法则是运用的链表的逻辑连续特性，使不同的数据存在同一个哈希地址下。我采用了第二种，所以对其优缺点比较了解，缺点很明显就是增加查找的复杂度，而相对与再散列法的优点还是很大的，可以避免重复的计算 hash 地址。

4.如何 resize，当装载数与数组长度的比例大于等于比例因子，这是说明数组容量快要饱和，为了避免产生大量的冲突，必须采用 resize。

而在写代码的过程中也考虑了很久。比如在数组中存什么，数组长度定义多少之类。这里我说说我的方法。数组中我存的是链表的首节点，因为链表的特性，只要知道头就可以获得整个链表，于此匹配的，我也采用了头插法，当产生冲突，我把它放在链表的头位置。这样代码可以减少不少。至于数组长度我定义为 2^n ，原因与&运算有关，因为 2^{n-1} 的二进制所有位都为 1，这样的与运算可以使冲突尽量减少。当产生冲突时，数组长度扩大一倍，感觉也是比较合理的。

以下是部分代码

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.   * 结点类
3.   * @author zrq
4.   *所有数据类都必须继承这个结点类
5.   */
6. public class Node {
7.     private String account;//账号是唯一标示
8.     private String password;
9.     private Node Next;
10.
11.    public int hashCode()
12.    {
13.        int hash=1;
14.        hash=Integer.parseInt(account);
```

```
15.     return hash;
16. }
17.
18. public String getAccount() {
19.     return account;
20. }
21.
22. public void setAccount(String account) {
23.     this.account = account;
24. }
25.
26. public String getPassword() {
27.     return password;
28. }
29.
30. public void setPassword(String password) {
31.     this.password = password;
32. }
33.
34. public Node getNext() {
35.     return Next;
36. }
37.
38. public void setNext(Node next) {
39.     Next = next;
40. }
41.
42. }
```

由于我是采用了数字作为测试数据，所以节点中的属性是直接定义的（account 是唯一的）。
重写了 hashCode ()，产生的方法就是采用了 account 这个唯一的属性（不通用）。

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class MyHashMap {
2.     private int initial_size = 16;// 数组初始大小
```

```
3.     private int finally_size = 16;// 用来记录现在数组大小
4.     private double balance = 0.75;// 装载因子
5.     private Node[] mapNodes;//组成数组
6.     private int count = 0;// 用于计数
7.     int hash;
8.     int index;
9.
10.    // 初始化时开辟数组空间
11.    public MyHashMap() {
12.        mapNodes = new Node[initial_size];
13.    }
14.
15.    public void put(Node node) {
16.        hash = node.hashCode();// 可修改
17.        index = hash & (finally_size - 1);// 获得索引位置
18.        Node fi = mapNodes[index];
19.        if (fi == null) {
20.            mapNodes[index] = node;
21.            count++;
22.            if (count > finally_size * balance)
23.                reSize();
24.
25.        } else {
26.            // System.out.println("test " + mapNodes[index].head.getAccount());
27.            node.setNext(fi);
28.            mapNodes[index] = node;
29.        }
30.    }
31.    /**
32.     * 通过 account 查找的方法
33.     * @param x: account
34.     * @return: password
35.     */
36.    public String GetValue(String x) {
37.        Node s = new Node();
```

```
38.     s.setAccount(x);
39.     boolean f = false;
40.     int hash = s.hashCode();
41.     index = hash & (finally_size - 1);
42.     Node n = mapNodes[index];
43.     while (n != null) {
44.         if (n.getAccount().equals(x)) {
45.             f = true;
46.             break;
47.         }
48.         n = n.getNext();
49.     }
50.     if (f)
51.         return n.getPassword();
52.     else {
53.         System.out.println(finally_size);
54.         return "null";
55.     }
56. }
57.
58. /**
59. * 当达到装载因子时扩容
60. */
61. public void reSize() {
62.     count=0;
63.     Node[] copy = new Node[finally_size * 2];
64.     for (int i = 0; i < finally_size; i++) {
65.         Node node = mapNodes[i];
66.         Node dnext;
67.         while (node != null) {
68.             //System.out.println("de"+node.getAccount());
69.             dnext = node.getNext();
70.             int index = node.hashCode() & (finally_size * 2 - 1);
71.             if (copy[index] == null) {
72.                 node.setNext(null);
```

```
73.         copy[index] = node;
74.         count++;
75.     } else {
76.         Node t = copy[index];
77.         node.setNext(t);
78.         copy[index] = node;
79.
80.     }
81.     node = dnext;
82. }
83.
84. }
85. finally_size = finally_size * 2;
86. mapNodes = copy;
87. }
88. }
```

写的比较急，没怎么优化。

以下是测试类。我用 1000000 个连续的号码进行存储和查找的测试。系统的时间大概在 2600ms，我的大概在 3000ms。查找 100000 个数据，我的约为 90ms，系统在 120ms 左右，当然这和我的 node 设置的针对性也有关系。

以下是 myhashmap 的测试主函数（将 da 放的位置改变就可以测试不同的时间）

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public static void main(String[] args) {
2.     // TODO Auto-generated method stub
3.     File fi = new File("F:\\test.txt");
4.
5.     try {
6.         MyHashMap map = new MyHashMap();
7.         // HashMap< String, String> map=new HashMap<String, String>();
8.         Scanner scan = new Scanner(fi);
9.
10.        while (scan.hasNext()) {
```

```
11. String s = scan.next();
12. Node node = new Node();
13. node.setAccount(s);
14. node.setPassword(s);
15. map.put(node);
16. }
17.
18. int x=100000;
19. Date da = new Date();
20. long n = da.getTime();
21. while(x-->0)
22. {
23. map.GetValue(""+x);
24. }
25. Date de = new Date();
26. long t = de.getTime();
27. System.out.println(t - n + "ms");
28. } catch (FileNotFoundException e) {
29. // TODO Auto-generated catch block
30. e.printStackTrace();
31. }
32.
33. }
```

以下是系统的测试主函数

[java] [view plain](#) [copy](#) [print?](#)

```
1. public static void main(String[] args) {
2.     // TODO Auto-generated method stub
3.     File fi = new File("F:\\test.txt");
4.
5.     try {
6.         HashMap<String, String> map = new HashMap<String, String>();
7.         Scanner scan = new Scanner(fi);
```

```
8.     int x = 100000;
9.     while (scan.hasNext()) {
10.         String s = scan.next();
11.         map.put(s, s);
12.     }
13.     Date da = new Date();
14.     long n = da.getTime();
15.     while (x-- > 0) {
16.         map.get("" + x);
17.     }
18.     Date de = new Date();
19.     long t = de.getTime();
20.     System.out.println(t - n + "ms");
21. } catch (FileNotFoundException e) {
22.     // TODO Auto-generated catch block
23.     e.printStackTrace();
24. }
25.
26. }
27.
28. public static void main(String[] args) {
29.     // TODO Auto-generated method stub
30.     File fi = new File("F:\\test.txt");
31.
32.     try {
33.         HashMap<String, String> map = new HashMap<String, String>();
34.         Scanner scan = new Scanner(fi);
35.         int x = 100000;
36.         while (scan.hasNext()) {
37.             String s = scan.next();
38.             map.put(s, s);
39.         }
40.         Date da = new Date();
41.         long n = da.getTime();
42.         while (x-- > 0) {
```

```
43.         map.get("") + x);
44.     }
45.     Date de = new Date();
46.     long t = de.getTime();
47.     System.out.println(t - n + "ms");
48. } catch (FileNotFoundException e) {
49.     // TODO Auto-generated catch block
50.     e.printStackTrace();
51. }
52.
53. }
```

1.10.9. 如何迅速匹配兄弟字符串

如果两个字符串的字符一样，但是顺序不一样，被认为是兄弟字符串，问如何迅速匹配兄弟字符串？

首先：接到题目，匹配字符串，这不简单了，遍历嘛。。

方法一：

步骤如下：

- 1.判断两个字符串的长度是否一样。
- 2.循环提取第一个字符串的字符去第二个字符串中寻找是否存在？
- 3.全部都有则是兄弟字符串，其他则不是兄弟字符串。

时间复杂度 N^2 , 平方级。

额，这算法真的就正确么？？？？？？

来看看这种情况：字符串 A 为 aab；字符串 B 为 abc，一看就知道它们是 false，那按照上面我写的算法得出的结论却是 true。

上面的算法错误的，考虑不周，那以遍历这种思路到底是否能判断兄弟字符串？

能，只要把上面的算法第 2 步稍微改动一下，改为“2.循环提取第一个字符串的字符去第二个字符串中寻找是否存在？存在则移除第二个字符串中的那个字符。”

好了，遍历思路算是可以解决了这个问题了。

不过嘛，遍历思路的时间复杂度是指数级，太耗时间，性能不好。

方法二：

赋予字符额外的意义。什么意思了，给 26 个字符依次赋予质数。质数是比较特殊的一堆数字，它们只能被 1 和本身整除。

给 a 赋值 2、给 b 赋值 3、给 c 赋值 5、给 d 赋值 7、给 e 赋值 11、给 f 赋值 13 等等.....

好了，给两个字符串中的所有字符都赋值了，，接着让它们各自相加，如果两个字符串得出的结果是一样的，那它们是兄弟字符串。。

嘎嘎，时间复杂度是常数。性能好了是不？？？？

别太高兴了，这个算法到目前为止也是有问题的，来看看这种情况：bf 和 ce 不是兄弟字符串，按照上面的赋值规律 $b+f=3+13=16$; $c+e=5+11=16$ ，看吧，明明他们就不是兄弟字符串，但是按照上面的算法就错了。

怎么解决这个问题了？用乘积：每个字符串内部求乘积，相等就是兄弟字符串。

好了，这算法是正确的，但是呢，又有个算法外的问题：字符串相乘及其容易出现结果溢出，说得简单点就是乘积太大了，大于程序语言的内置的整数类型（int、long）所能表示的最大值。这怎么解决？有个比较偏的方法就是用数组来存储乘积。具体方法我不说了，跟本偏篇文章无关。。。。

那怎么解决兄弟字符串的问题？？？用平方和或者立方和。。。。。既然直接用加法不行，用乘法还会溢出。那换个思路用平方和。。。

$b*b+f*f=3*3+13*13=178$; $c*c+e*e=5*5+11*11=146$

看吧它们不是兄弟字符串吧。。。。。

这只是我的算法思路，可能有误，仅供参考。。。

1.10.10.腾讯数组乘积赋值的问题

2012 年 4 月 67 日的腾讯暑期实习生招聘笔试中，出了一道与上述 21 题类似的题，原题大致如下：

两个数组 $a[N]$, $b[N]$, 其中 $A[N]$ 的各个元素值已知, 现给 $b[i]$ 赋值, $b[i] = a[0]*a[1]*a[2]...*a[N-1]/a[i]$;

要求:

- 1.不准用除法运算
- 2.除了循环计数值, $a[N], b[N]$ 外, 不准再用其他任何变量 (包括局部变量, 全局变量等)
- 3.满足时间复杂度 $O(n)$, 空间复杂度 $O(1)$ 。

说白了，你要我求 $b=a[0]*a*...a[i-1]*a*a[i+1]..*a[N-1]/a$ ，就是求：
 $a[0]*a[1]*...a[i-1]*a[i+1]..*a[N-1]$ 。只是我把 $a[i]$ 左边部分标示为 $left[i]$, $b[i]$ 右边部分标示为 $right[i]$ ，而实际上完全不申请 $left[i]$ ，与 $right[i]$ 变量，之所以那样标示，无非就是为了说明：除掉当前元素 $a[i]$ ，其他所有元素($a[i]$ 左边部分, 和 $a[i]$ 右边部分)的积。

[java] [view plain](#) [copy](#) [print](#)?

```
1. static void arrayMultiplication(int A[], int output[], int n) {
```

```
2.     int left = 1;
3.     int right = 1;
4.     for (int i = 0; i < n; i++) {
5.         output[i] = 1;
6.     }
7.     for (int i = 0; i < n; i++) {
8.         output[i] *= left;
9.         output[n - 1 - i] *= right;
10.        left *= A[i];
11.        right *= A[n - 1 - i];
12.        System.out.println("i:" + output[i]);
13.        System.out.println("n - 1 - i:" + output[n - 1 - i]);
14.        System.out.println("left:" + left);
15.        System.out.println("right:" + right);
16.
17.    }
18.    for (int i = 0; i < n; i++) {
19.        System.out.println(output[i]);
20.    }
21. }
```

1.11.面试题集合（十）

1.11.1. 有一个整数数组，请求出两两之差绝对值最小的值

有一个整数数组，请求出两两之差绝对值最小的值，记住，只要得出最小值即可，不需要求出是哪两个数

如果没有空间复杂度的限制啊，可以借助于桶排序的思想。

数组为 $a[]$

遍历得到最大 \max ，遍历得到最小 \min 。

位图长度为 $\text{abs}(\max) + \text{abs}(\min)$ ，即为 $\text{byte } b[]$

遍历 a ，遍历到 $a[i]$ ，则将 $b[a[i]-\min]$ 置为 1；

然后遍历 b，比较相邻两个为 1 的下标差值。

复杂度为 $O(\text{abs}(\max) + \text{abs}(\min))$ ，汗

ps:

直接用桶排序就可以了~~

考虑到可能有重复数字的情况，可以用两个 bit 位表示

1.11.2. 给出一个函数来合并两个字符串 A 和 B。字符串 A 的后几个字节和字符串 B 的前几个字节重叠

```
1. [cpp] view plaincopyprint?
2. /*
3. 给出一个函数来合并两个字符串 A 和 B。字符串 A 的后几个字节和字符串 B 的前几个字
   节重叠。
4. /**
5.
6. #include <iostream>
7. #include <iomanip>
8. #include <limits>
9.
10. using namespace std;
11. void copystr(char pachar[], char pbchar[], int sza, int szb, char* &result);
12.
13. int main()
14. {
15.     char pachar[] = "asdfvxcbnbvcxzghgh";
16.     char pbchar[] = "ghjklqwe";
17.
18.     int sza = sizeof(pachar);
19.     int szb = sizeof(pbchar);
20.     int sz = sza + szb - 1;
21.     char* result = new char[sz];
22.
23.     copystr(pachar, pbchar, sza, szb, result);
24.
```

```
25.     cout << result << endl;
26.
27.     return 0;
28. }
29.
30. void copystr(char pachar[], char pbchar[], int sza, int szb, char* &result)
31. {
32.     int posa = 0;
33.     int posb = 0;
34.     int pos = 0;
35.     int sa = 0;
36.     int sz = sza + szb - 1;
37.     while(pos < sz && posa < sza - 1 && posb < szb - 1){ //字符串索引为 [0 ~ size -1], 最后
   ——一个字符为结束字符‘\0’，因此有效字符索引 index < size -1
38.
39.     //如果 a 串中没找到与 b 串中首字符相匹配的，则说明不存在重叠，
40.     //在找到重叠之前，将 a 串中相应字符拷贝到目标串
41.     while((pachar[posa] != pbchar[0]) && (posa < sza - 1)){
42.         result[pos] = pachar[posa];
43.         ++pos;
44.         ++posa;
45.     }
46.
47.     //拷贝 a 串停止，两种可能情况分别处理：一是拷贝到 a 串末尾，二是遇到与 b 串首
   ——字符相匹配的情况
48.     if(posa == sza - 1){ //拷贝到末尾，继续拷贝 b 串直至结束
49.         while(posb <= szb - 1){
50.             result[pos] = pbchar[posb];
51.             ++pos;
52.             ++posb;
53.         }
54.         break;
55.     }else{ //遇到相同字符，分两种情况，一是遇到重叠字串，即 a 串此处开始的部分与
   ——b 串的起始部分完全相同，二是偶遇相同字符，不是头尾重叠
56.         sa = posa; //为不是重叠的结果保存备用记录
```

```
57.         while((pachar[posa] == pbchar[posb]) && (posa < sza - 2) && (posb < szb - 2)){ // 循环
    判断是否完全重叠
58.             ++posa;
59.             ++posb;
60.         }
61.         if(posa == sza - 2){ // 重叠
62.             posb = 0; // 恢复 posb 初始位置
63.             while(posb <= szb - 1){
64.                 result[pos] = pbchar[posb];
65.                 ++pos;
66.                 ++posb;
67.             }
68.             break;
69.         }else{ // 偶遇相同字符
70.             posa = sa; // 恢复 posa 偶遇 b[0] 位置
71.             posb = 0; // 恢复 posb 初始位置
72.             result[pos] = pachar[posa];
73.             ++pos;
74.             ++posa;
75.         }
76.     }
77. }
78. }
79.
80. /*
81. 给出一个函数来合并两个字符串 A 和 B。字符串 A 的后几个字节和字符串 B 的前几个字
节重叠。
82. /**
83.
84. #include <iostream>
85. #include <iomanip>
86. #include <limits>
87.
88. using namespace std;
89. void copystr(char pachar[], char pbchar[], int sza, int szb, char* &result);
```

```
90.  
91. int main()  
92. {  
93.     char pachar[] = "asdfvxcbnbvcxzghgh";  
94.     char pbchar[] = "ghjklqwe";  
95.  
96.     int sza = sizeof(pachar);  
97.     int szb = sizeof(pbchar);  
98.     int sz = sza + szb - 1;  
99.     char* result = new char[sz];  
100.  
101.    copystr(pachar, pbchar, sza, szb, result);  
102.  
103.    cout << result << endl;  
104.  
105.    return 0;  
106. }  
107.  
108. void copystr(char pachar[], char pbchar[], int sza, int szb, char* &result)  
109. {  
110.     int posa = 0;  
111.     int posb = 0;  
112.     int pos = 0;  
113.     int sa = 0;  
114.     int sz = sza + szb - 1;  
115.     while(pos < sz && posa < sza - 1 && posb < szb - 1){ //字符串索引为 [0 ~ size -1], 最后  
           //一个字符为结束字符‘\0’，因此有效字符索引 index < size -1  
116.  
117.         //如果 a 串中没找到与 b 串中首字符相匹配的，则说明不存在重叠，  
118.         //在找到重叠之前，将 a 串中相应字符拷贝到目标串  
119.         while((pachar[posa] != pbchar[0]) && (posa < sza - 1)){  
120.             result[pos] = pachar[posa];  
121.             ++pos;  
122.             ++posa;  
123.         }
```

```
124.  
125.      //拷贝 a 串停止，两种可能情况分别处理：一是拷贝到 a 串末尾，二是遇到与 b 串首  
          字符相匹配的情况  
126.      if(posa == sza - 1){ //拷贝到末尾,继续拷贝 b 串直至结束  
127.          while(posb <= szb - 1){  
128.              result[pos] = pbchar[posb];  
129.              ++pos;  
130.              ++posb;  
131.          }  
132.          break;  
133.      }else{ //遇到相同字符，分两种情况，一是遇到重叠字串，即 a 串此处开始的部分与  
          b 串的起始部分完全相同，二是偶遇相同字符，不是头尾重叠  
134.          sa = posa; //为不是重叠的结果保存备用记录  
135.          while((pachar[posa] == pbchar[posb]) && (posa < sza - 2) && (posb < szb - 2)){ //循环  
          判断是否完全重叠  
136.              ++posa;  
137.              ++posb;  
138.          }  
139.          if(posa == sza - 2){ //重叠  
140.              posb = 0; //恢复 posb 初始位置  
141.              while(posb <= szb - 1){  
142.                  result[pos] = pbchar[posb];  
143.                  ++pos;  
144.                  ++posb;  
145.              }  
146.              break;  
147.          }else{ //偶遇相同字符  
148.              posa = sa; //恢复 posa 偶遇 b[0] 位置  
149.              posb = 0; //恢复 posb 初始位置  
150.              result[pos] = pachar[posa];  
151.              ++pos;  
152.              ++posa;  
153.          }  
154.      }  
155.  }
```

156. }

1.11.3. 编程实现两个正整数的除法（不能用除法操作符）

```
1.  /*
2.   编程实现两个正整数的除法（不能用除法操作符）。
3.  */
4.
5. // 只考虑大数除小数，因小数除大数结果为 0，余数为小数，无需计算
6.
7. #include <iostream>
8. #include <iomanip>
9. #include <limits>
10.
11. using namespace std;
12.
13.
14. void devide(int val1, int val2, int& res, int &rev)
15. {
16.     int maxv = max(val1, val2);
17.     int minv = min(val1, val2);
18.     res = 0;
19.     rev = 0;
20.     if(maxv == minv){
21.         res = 1;
22.         rev = 0;
23.         return;
24.     }else{
25.         while(maxv > minv){
26.             maxv = maxv - minv;
27.             res += 1;
28.         }
29.         rev = maxv;
30.     }
31. }
```

```
31.    }
32.
33. }
34.
35. int main()
36. {
37.     int val1 = 100;
38.     int val2 = 9;
39.     int res,rev; // res 为结果， rev 为余数
40.     devide(val1,val2, res,rev);
41.
42.     cout << "results of " << val1 << " and " << val2 << " devide is " << res << " and " << rev << endl;
43.
44.     return 0;
45. }
```

1.11.4. 平面上 N 个点，没两个点都确定一条直线，求出斜率最大的那条直线所通过的两个点

平面上 N 个点，没两个点都确定一条直线，求出斜率最大的那条直线所通过的两个点（斜率不存在的情况不考虑）。时间效率越高越好。

先把 N 个点按 x 排序。

斜率 k 最大值为 $\max(\text{斜率}(\text{point}[i], \text{point}[i+1])) \quad 0 \leq i < n-2$ 。

复杂度 $N \log(N)$ 。

以 3 个点为例,按照 x 排序后为 ABC,假如 3 点共线,则斜率一样,假如不共线,则可以证明 AB 或 BC 中,

一定有一个点的斜率大于 AC,一个点的斜率小于 AC。

1.11.5. 字符串原地压缩

```
1.  /*
2.  * Copyright (c) 2011 alexingcool. All Rights Reserved.
3.  */
4. #include <iostream>
5. #include <iterator>
6. #include <algorithm>
7.
8. using namespace std;
9.
10. char array[] = "eeeeeaaff";
11. char array2[] = "geeeeeaaaffg";
12. const int size = sizeof array / sizeof *array;
13. const int size2 = sizeof array2 / sizeof *array2;
14.
15. void compression(char *array, int size)
16. {
17.     int i = 0, j = 0;
18.     int count = 0;
19.
20.     while(j < size) {
21.         count = 0;
22.         array[i] = array[j];
23.
24.         while(array[j] == array[i]) {
25.             count++;
26.             j++;
27.         }
28.         if(count == 1) {
29.             i++;
30.         }
31.         else {
32.             array[+i] = '0' + count;
33.             ++i;
```

```

34.     }
35.   }
36.
37.   array[i] = 0;
38. }
39.
40. void main()
41. {
42.   compression(array, size);
43.   cout << array << endl;
44.   compression(array2, size2);
45.   cout << array2 << endl;
46. }
```

1.11.6. 一排 N (最大 1 M) 个正整数+1 递增，乱序排列

题目大意如下：

一排 N (最大 1 M) 个正整数+1 递增，乱序排列，第一个不是最小的，把它换成-1，最小

数为 a 且未知求第一个被

-1 替换掉的数原来的值，并分析算法复杂度。

解题思路：

一般稍微有点算法知识的人想想就会很容易给出以下解法：

设 $S_n = a + (a+1) + (a+2) + \dots + (a+n-1) = na + n(n-1)/2$

扫一次数组即可找到最小值 a，时间复杂度 $O(n)$

设 $S =$ 修改第一项后所有数组项之和，求和复杂度为 $O(n)$

则被替换掉的第一项为 $a_1 = S_n - S - 1$

总的时间复杂度为 $O(1) + O(n) + O(n) = O(n)$

根据该算法写出程序很简单，就不写了

主要是解题过程中没有太考虑题目中给的 1 M 这个数字，一面的时候被问到**求和溢出怎么办？**

当时我一想，如果要考虑溢出，必然是要处理大数问题，以前没有看到大数就头疼……所以立马想了个绕过大数加法的方法，如下：

设定另外一个数组 $b[N]$

用 $a, a+1, a+2, \dots, a+n-1$ 依次分别减去原数组，得到的差放在该数组里，此求差过程复杂度为 $O(n)$

对该数组各项求和即可得到 S_{n-S}

面试官让证明一下我的设想，当时还没有给我纸和笔，用手在桌子上比划了一下没想出来，回来躺在床上想了一会就想出来了，也没什么难度：

相减求和后的数组，最差情况下应该是连续 $n/2$ 个负数或者正数相加，如果不溢出，后面正负混合相加的话肯定不会溢出；这种情况下最差特殊情况下是，原数列按照降序排列（除了第一项被替换掉了），而我们减时所用数列是增序排列。所得结果将是 1 个正数， $n/2-1$ 个负数， $n/2$ 个正数；而且我们相当于用最大的 $n/2$ 个数减去最小的 $n/2$ 个数，差值之和最大，取到了最差情况，我们只考虑后面一半求和的情况即可（前面有个-1 不方便处理）：

$S(n/2) = (n-1) + (n-3) + (n-5) + \dots + 1$ (n 为奇数时最后一项是 0，不影响我们讨论数量级计算溢出)

$$= [(n-1)+1] * n/4 = n^2/4$$

题目中给定 n 最大为 $1M = 1024*1024$

那么 $S(n/2)$ 的最大量级为 $1024^4 = 2^{40}$

而 `long long` 类型为 64 位，可以存放下该和，成功避免大数问题。

直接求和办法，一是和可能溢出，二是面试官要求把原始数组改称 `long long` 的话(即 a 可以也可能很大，求和时稍微加一下就会溢出)就得考虑大数求解了；而这种差值办法可以直接消掉 a ，求和只和 n 相关，和 a 无关。

1.11.7. 找出被重复的数字

题目：

数组 $a[N]$ ，1 至 $N-1$ 这 $N-1$ 个数存放在 $a[N]$ 中，其中某个数重复一次。写一个函数，找出被重复的数字。

方法一：异或法。

数组 $a[N]$ 中的 N 个数异或结果与 1 至 $N-1$ 异或的结果再做异或，得到的值即为所求。

- 设重复数为 A ，其余 $N-2$ 个数异或结果为 B 。
- N 个数异或结果为 A^A^B
- 1 至 $N-1$ 异或结果为 A^B
- 由于异或满足交换律和结合律，且 $X^X = 0$ $0^X = X$;
- 则有
- $(A^B)^(A^A^B) = A^B^B = A$

代码：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include<time.h>
5. void xor_findDup(int * a,int N)
6. {
7.     int i;
8.     int result=0;
9.     for(i=0;i<N;i++)
10.    {
11.        result ^= a[i];
12.    }
13.
14.    for (i=1;i<N;i++)
15.    {
16.        result ^= i;
17.    }
18.
19.    printf("%d\n",result);
20.
21. }
22.
23.
24.
25. int main(int argc, char* argv[])
26. {
27.     int a[] = {1,2,1,3,4};
28.     xor_findDup(a,5);
29.     return 0;
30. }
```

方法二：数学法。

对数组的所有项求和，减去 1 至 N-1 的和，即为所求数。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include<time.h>
5. void xor_findDup(int * a,int N)
6. {
7.     int tmp1 = 0;
8.
9.     int tmp2 = 0;
10.
11.    for (int i=0; i<N-1; ++i)
12.
13.    {
14.
15.        tmp1+=(i+1);
16.
17.        tmp2+=a[i];
18.
19.    }
20.    tmp2+=a[N-1];
21.    int result=tmp2-tmp1;
22.    printf("%d\n",result);
23.
24. }
25.
26.
27.
28. int main(int argc, char* argv[])
29. {
30.     int a[] = {1,2,4,3,4};
31.     xor_findDup(a,5);
32.     return 0;
33. }
```

对于求和，可以直接根据公式定义一个宏。#define sum(x) (x*(x+1)/2)

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include<time.h>
5. #define sum(x) (x*(x+1)/2)
6. void xor_findDup(int * a,int N)
7. {
8.     int tmp1 = sum((N-1));//注意 N-1 要加括号
9.     int tmp2 = 0;
10.
11.    for (int i=0; i<N; ++i)
12.    {
13.        tmp2+=a[i];
14.    }
15.    int result=tmp2-tmp1;
16.    printf("%d\n",result);
17. }
18.
19. int main(int argc, char* argv[])
20. {
21.     int a[] = {1,2,4,2,3};
22.     xor_findDup(a,5);
23.     return 0;
24. }
```

方法三：标志数组法

申请一个长度为 n-1 且均为'0'组成的字符串。然后从头遍历 a[n]数组，取每个数组元素 a[i] 的值，将其对应的字符串中的相应位置置 1，如果已经置过 1 的话，那么该数就是重复的数。就是用位图来实现的。 如果考虑空间复杂度的话，其空间 O (N)

```
1. #include <stdio.h>
2. #include <stdlib.h>
```

```
3. #include <math.h>
4. #include<time.h>
5. #define sum(x) (x*(x+1)/2)
6. void xor_findDup(int * arr,int NUM)
7. {
8.     int *arrayflag = (int *)malloc(NUM*sizeof(int));
9.     int i=1;
10.
11.    while(i<NUM)
12.    {
13.        arrayflag[i] = false;
14.        i++;
15.    }
16.
17.    for( i=0; i<NUM; i++)
18.    {
19.        if(arrayflag[arr[i]] == false)
20.            arrayflag[arr[i]] = true;      // 置出现标志
21.
22.        else
23.        {
24.            printf("%d\n",arr[i]);
25.            return ;//返回已经出现的值
26.        }
27.
28.    }
29. }
30.
31. int main(int argc, char* argv[])
32. {
33.     int a[] = {1,3,2,4,3};
34.     xor_findDup(a,5);
35.     return 0;
36. }
```

方法四：固定偏移量法

$a[N]$, 里面是 1 至 $N-1$ 。原数组 $a[i]$ 最大是 $N-1$, 若 $a[i]=K$ 在某处出现后, 将 $a[K]$ 加一次 N , 做标记, 当某处 $a[i]=K$ 再次成立时, 查看 $a[K]$ 即可知道 K 已经出现过。该方法不用另外开辟 $O(N)$ 的内存空间, 但是在查重之后要将数组进行恢复。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include<time.h>
5. void xor_findDup(int * arr,int NUM)
6. {
7.     int temp=0;
8.     for(int i=0; i<NUM; i++)
9.     {
10.
11.         if(arr[i]>=NUM)
12.             temp=arr[i]-NUM;      // 该值重复了, 因为曾经加过一次
13.             了
14.         else
15.             temp=arr[i];
16.         if(arr[temp]<NUM)
17.         {
18.             arr[temp]+=NUM; //做上标记
19.         }
20.
21.     else
22.     {
23.         printf("有重复 %d\n",temp);
24.     return;
25.     }
26. }
27.
28. printf("无重复");
29. return ;
30. }
```

```

31. void clear(int *data,int num)//清理数据
32. {
33.     for(int i=0;i<num;i++)
34.     {
35.         if(data[i]>num)
36.             data[i]-=num;
37.     }
38.
39. }
40. int main(int argc, char* argv[])
41. {
42.     int a[] = {2,4,3,4,1};
43.     xor_findDup(a,5);
44.     clear(a,5);
45.     return 0;
46. }
```

方法五：符号标志法

上个方法出现后是加 N，也可以出现后加个负号，就是符号标志法。

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <math.h>
5. #include<time.h>
6.
7. void xor_findDup(int * arr,int NUM)
8. {
9.     int temp=0;
10.    for(int i=0; i<NUM; i++)
11.    {
12.        if(arr[i]<0)
13.            temp=0-arr[i]; // 该值重复了，因为曾经加过一次了
14.        else
15.            temp=arr[i];
```

```

16.     if(arr[temp]>0)
17.     {
18.         arr[temp]=0-arr[temp]; //做上标记
19.     }
20.     else
21.     {
22.         printf("有重复 %d\n",temp);
23.         return;
24.     }
25. }
26. printf("无重复 ");
27. return ;
28. }

29. void clear(int *data,int num)//清理数据
30. {
31.     for(int i=0;i<num;i++)
32.     {
33.         if(data[i]<0)
34.             data[i]=0-data[i];
35.     }
36. }

37. int main(int argc, char* argv[])
38. {
39.     int a[] = {3,2,1,4,1};
40.     xor_findDup(a,5);
41.     clear(a,5);
42.     return 0;
43. }

```

以上的方法对数组元素的值的范围是有限制的,如果数组元素的值不是在 1 至 N-1 范围时,可以先求出数组元素的最大值。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>

```
4. #include <math.h>
5. #include<time.h>
6.
7. int do_dup_mal(int arr[], int n, int *pre, int *back)
8. {
9.     int MAX = -1;
10.    int i = 0;
11.    int sameVal = -1;
12.    *pre = *back = -1;
13.
14.    for (int j=0; j<n; j++)
15.    {
16.        if (arr[j] > MAX) MAX = arr[j];//找出数组中的最大数
17.    }
18.
19.    char *arrayflag = new char[MAX+1];
20.    if (NULL == arrayflag)
21.        return -1;
22.    memset(arrayflag, 0, MAX+1 ); // '\0' == 0
23.    for(i=0; i<n; i++)
24.    {
25.        if(arrayflag[arr[i]] == '\0')
26.            arrayflag[arr[i]] = '\1'; // 置出现标志
27.        else
28.        {
29.            sameVal = arr[i]; //返回已经出现的值
30.            *back = i;
31.            break;
32.        }
33.    }
34.    delete[] arrayflag;
35.    if (i < n)
36.    {
37.        for (int j=0; j<n; j++)
38.        {
```

```
39.         if (sameVal == arr[j])
40.         {
41.             *pre = j;
42.             return true;
43.         }
44.     }
45. }
46. return false;
47. }
48.
49.
50.
51.
52.
53. void main(int argc, char *argv[])
54. {
55.     int prePos = -1, backPos = -1;
56.     int myArry[11];
57.     myArry[0] = 1;
58.     myArry[1] = 3;
59.     myArry[2] = 3;
60.     myArry[3] = 4;
61.     myArry[4] = 5;
62.     myArry[5] = 22;
63.     myArry[6] = 7;
64.     myArry[7] = 13;
65.     myArry[8] = 9;
66.     myArry[9] = 2;
67.     myArry[10] = 12;
68.
69.
70.     if (do_dup_mal(myArry, 11, &prePos, &backPos) )
71.         printf("%d\n",myArry[prePos]);
72. }
73.
```

1.11.8. Hashtable 和 HashMap 的区别

Hashtable 和 HashMap 的区别:

1. Hashtable 是 Dictionary 的子类, HashMap 是 Map 接口的一个实现类;
2. Hashtable 中的方法是同步的, 而 HashMap 中的方法在缺省情况下是非同步的。即是说, 在多线程应用程序中, 不用专门的操作就安全地可以使用 Hashtable 了; 而对于 HashMap, 则需要额外的同步机制。但 HashMap 的同步问题可通过 Collections 的一个静态方法得到解决:

Map Collections.synchronizedMap(Map m)

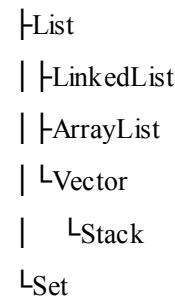
这个方法返回一个同步的 Map, 这个 Map 封装了底层的 HashMap 的所有方法, 使得底层的 HashMap 即使是在多线程的环境中也是安全的。

3. 在 HashMap 中, null 可以作为键, 这样的键只有一个; 可以有一个或多个键所对应的值为 null。当 get()方法返回 null 值时, 即可以表示 HashMap 中没有该键, 也可以表示该键所对应的值为 null。因此, 在 HashMap 中不能由 get()方法来判断 HashMap 中是否存在某个键, 而应该用 containsKey()方法来判断。

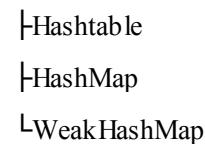
Vector、ArrayList 和 List 的异同

线性表, 链表, 哈希表是常用的数据结构, 在进行 Java 开发时, JDK 已经为我们提供了一系列相应的类来实现基本的数据结构。这些类均在 java.util 包中。本文试图通过简单的描述, 向读者阐述各个类的作用以及如何正确使用这些类。

Collection



Map



Collection 接口

Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素（Elements）。一些 Collection 允许相同的元素而另一些不行。一些能排序而另一些不行。Java SDK 不提供直接继承自 Collection 的类，Java SDK 提供的类都是继承自 Collection 的“子接口”如 List 和 Set。

所有实现 Collection 接口的类都必须提供两个标准的构造函数：无参数的构造函数用于创建一个空的 Collection，有一个 Collection 参数的构造函数用于创建一个新的 Collection，这个新的 Collection 与传入的 Collection 有相同的元素。后一个构造函数允许用户复制一个 Collection。

如何遍历 Collection 中的每一个元素？不论 Collection 的实际类型如何，它都支持一个 iterator() 的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问 Collection 中每一个元素。典型的用法如下：

```
Iterator it = collection.iterator(); // 获得一个迭代子
while(it.hasNext()) {
    Object obj = it.next(); // 得到下一个元素
}
```

由 Collection 接口派生的两个接口是 List 和 Set。

List 接口

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

和下面要提到的 Set 不同，List 允许有相同的元素。

除了具有 Collection 接口必备的 iterator() 方法外，List 还提供一个 listIterator() 方法，返回一个 ListIterator 接口，和标准的 Iterator 接口相比，ListIterator 多了一些 add() 之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。

实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和 Stack。

LinkedList 类

LinkedList 实现了 List 接口，允许 null 元素。此外 LinkedList 提供额外的 get，remove，insert 方法在 LinkedList 的首部或尾部。这些操作使 LinkedList 可被用作堆栈（stack），队列（queue）或双向队列（dequeue）。

注意 LinkedList 没有同步方法。如果多个线程同时访问一个 List，则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List：

```
List list = Collections.synchronizedList(new LinkedList(...));
```

ArrayList 类

ArrayList 实现了可变大小的数组。它允许所有元素，包括 null。ArrayList 没有同步。

`size`, `isEmpty`, `get`, `set` 方法运行时间为常数。但是 `add` 方法开销为分摊的常数，添加 n 个元素需要 $O(n)$ 的时间。其他的方法运行时间为线性。

每个 `ArrayList` 实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并没有定义。当需要插入大量元素时，在插入前可以调用 `ensureCapacity` 方法来增加 `ArrayList` 的容量以提高插入效率。

和 `LinkedList` 一样，`ArrayList` 也是非同步的（unsynchronized）。

Vector 类

`Vector` 非常类似 `ArrayList`，但是 `Vector` 是同步的。由 `Vector` 创建的 `Iterator`，虽然和 `ArrayList` 创建的 `Iterator` 是同一接口，但是，因为 `Vector` 是同步的，当一个 `Iterator` 被创建而且正在被使用，另一个线程改变了 `Vector` 的状态（例如，添加或删除了一些元素），这时调用 `Iterator` 的方法时将抛出 `ConcurrentModificationException`，因此必须捕获该异常。

Stack 类

`Stack` 继承自 `Vector`，实现一个后进先出的堆栈。`Stack` 提供 5 个额外的方法使得 `Vector` 得以被当作堆栈使用。基本的 `push` 和 `pop` 方法，还有 `peek` 方法得到栈顶的元素，`empty` 方法测试堆栈是否为空，`search` 方法检测一个元素在堆栈中的位置。`Stack` 刚创建后是空栈。

Set 接口

`Set` 是一种不包含重复的元素的 `Collection`，即任意的两个元素 `e1` 和 `e2` 都有 `e1.equals(e2)=false`，`Set` 最多有一个 `null` 元素。

很明显，`Set` 的构造函数有一个约束条件，传入的 `Collection` 参数不能包含重复的元素。

请注意：必须小心操作可变对象（Mutable Object）。如果一个 `Set` 中的可变元素改变了自身状态导致 `Object.equals(Object)=true` 将导致一些问题。

Map 接口

请注意，`Map` 没有继承 `Collection` 接口，`Map` 提供 `key` 到 `value` 的映射。一个 `Map` 中不能包含相同的 `key`，每个 `key` 只能映射一个 `value`。`Map` 接口提供 3 种集合的视图，`Map` 的内容可以被当作一组 `key` 集合，一组 `value` 集合，或者一组 `key-value` 映射。

Hashtable 类

`Hashtable` 继承 `Map` 接口，实现一个 `key-value` 映射的哈希表。任何非空（non-null）的对象都可作为 `key` 或者 `value`。

添加数据使用 `put(key, value)`，取出数据使用 `get(key)`，这两个基本操作的时间开销为常数。

`Hashtable` 通过 `initial capacity` 和 `load factor` 两个参数调整性能。通常缺省的 `load factor` 0.75 较好地实现了时间和空间的均衡。增大 `load factor` 可以节省空间但相应的查找时间将增大，这会影响像 `get` 和 `put` 这样的操作。

使用 `Hashtable` 的简单示例如下，将 1, 2, 3 放到 `Hashtable` 中，他们的 `key` 分别是”one”，”two”，”three”：

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

要取出一个数，比如 2，用相应的 key:

```
Integer n = (Integer)numbers.get("two");
System.out.println("two = " + n);
```

由于作为 key 的对象将通过计算其散列函数来确定与之对应的 value 的位置，因此任何作为 key 的对象都必须实现 hashCode 和 equals 方法。hashCode 和 equals 方法继承自根类 Object，如果你用自定义的类当作 key 的话，要相当小心，按照散列函数的定义，如果两个对象相同，即 obj1.equals(obj2)=true，则它们的 hashCode 必须相同，但如果两个对象不同，则它们的 hashCode 不一定不同，如果两个不同对象的 hashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 hashCode()方法，能加快哈希表的操作。

如果相同的对象有不同的 hashCode，对哈希表的操作会出现意想不到的结果（期待的 get 方法返回 null），要避免这种问题，只需要牢记一条：要同时复写 equals 方法和 hashCode 方法，而不要只写其中一个。

Hashtable 是同步的。

HashMap 类

HashMap 和 Hashtable 类似，不同之处在于 HashMap 是非同步的，并且允许 null，即 null value 和 null key。但是将 HashMap 视为 Collection 时(values()方法可返回 Collection)，其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将 HashMap 的初始化容量设得过高，或者 load factor 过低。

WeakHashMap 类

WeakHashMap 是一种改进的 HashMap，它对 key 实行“弱引用”，如果一个 key 不再被外部所引用，那么该 key 可以被 GC 回收。

总结

如果涉及到堆栈，队列等操作，应该考虑用 List，对于需要快速插入，删除元素，应该使用 LinkedList，如果需要快速随机访问元素，应该使用 ArrayList。

如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个类，应该使用同步的类。

要特别注意对哈希表的操作，作为 key 的对象要正确复写 equals 和 hashCode 方法。

尽量返回接口而非实际的类型，如返回 List 而非 ArrayList，这样如果以后需要将 ArrayList 换成 LinkedList 时，客户端代码不用改变。这就是针对抽象编程。

1.11.9. 用 1、2、2、3、4、5 这六个数字，写一个 main 函数，打印出所有不同的排列

```
1. import java.util.ArrayList;
2.
3. public class Test {
4.     public static void f(String in, ArrayList al) {
5.         if (al.size() == 1) {
6.             String tmp = in;
7.             if (!(tmp.contains("35") || tmp.contains("53") || tmp.indexOf('4') == 2)) {
8.                 System.out.println(tmp);
9.             }
10.        } else {
11.            ArrayList hsc = (ArrayList) al.clone();
12.            if (in.length() != 0) {
13.                hsc.remove(in.substring(in.length() - 1));
14.            }
15.            for (Object i : hsc) {
16.                f(in + i, hsc);
17.            }
18.        }
19.    }
20.
21.    public static void main(String[] args) {
22.        ArrayList al = new ArrayList();
23.        al.add("1");
24.        al.add("2");
25.        al.add("3");
26.        al.add("4");
27.        al.add("5");
28.        al.add("6");
29.        f("", al);
30.    }
31. }
```

1.11.10. 局部变量、全局变量和静态变量的含义

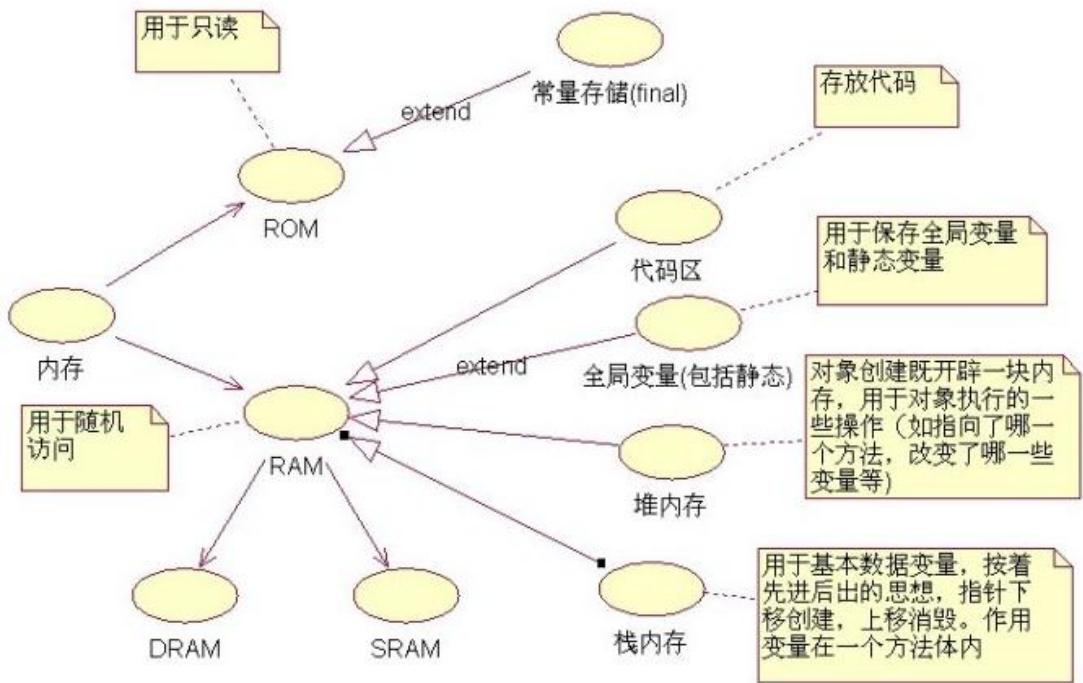
一：数据分配位置

- (1) 寄存器。这是最快的保存区域，因为它位于和其他所有保存方式不同的地方：处理器内部。然而，寄存器的数量十分有限，所以寄存器是根据需要由编译器分配。我们对此没有直接的控制权，也不可能在自己的程序里找到寄存器存在的任何踪迹。
- (2) 栈。驻留于常规 RAM（随机访问存储器）区域，但可通过它的“堆栈指针”获得处理的直接支持。堆栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java 编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些 Java 数据要保存在堆栈里——特别是对象句柄，但 Java 对象并不放到其中。
- (3) 堆。一种常规用途的内存池（也在 RAM 区域），其中保存了 Java 对象。和堆栈不同，“内存堆”或“堆”（Heap）最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据在堆里停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然付出一定的代价：在堆里分配存储空间时会花掉更长的时间！
- (4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管也在 RAM 里）。程序运行期间，静态存储的数据将随时等候调用。可用 static 关键字指出一个对象的特定元素是静态的。但 Java 对象本身永远都不会置入静态存储空间。
- (5) 常数存储。常数值通常直接置于程序代码内部。这样做是安全的，因为它们永远都不会改变。有的常数需要严格地保护，所以可考虑将它们置入只读存储器（ROM）。
- (6) 非 RAM 存储。若数据完全独立于一个程序之外，则程序不运行时仍可存在，并在程序的控制范围之外。其中两个最主要的例子便是“流式对象”和“固定对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，对象存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普通的、基于 RAM 的对象。Java 1.1 提供了对 Lightweight persistence 的支持。未来的版本甚至可能提供更完整的方案。

按照编译原理的观点，程序运行时的内存分配有三种策略，分别是静态的、栈式的、和堆式的。

静态存储分配是指在编译时就能确定每个数据目标在运行时刻的存储空间需求，因而在编译时就可以给他们分配固定的内存空间。这种分配策略要求程序代码中不允许有可变数据结构（比如可变数组）的存在，也不允许有嵌套或者递归的结构出现，因为它们都会导致编译程序无法计算准确的存储空间需求。

控制器—处理器—缓存—内存



为了增加系统的速度，把缓存扩大不就行了吗，扩大的越大，缓存的数据越多，系统不就越快了吗？缓存通常都是静态 RAM，速度是非常的快，但是静态 RAM 集成度低（存储相同的数据，静态 RAM 的体积是动态 RAM 的 6 倍），价格高（同容量的静态 RAM 是动态 RAM 的四倍），由此可见，扩大静态 RAM 作为缓存是一个非常愚蠢的行为，但是为了提高系统的性能和速度，我们必须要扩大缓存，这样就有一个折中的方法，不扩大原来的静态 RAM 缓存，而是增加一些高速动态 RAM 做为缓存，这些高速动态 RAM 速度要比常规动态 RAM 快，但比原来的静态 RAM 缓存慢，我们把原来的静态 ram 缓存叫一级缓存，而把后来增加的动态 RAM 叫二级缓存。

(1)局部变量

在一个函数内部定义的变量是内部变量，它只在本函数范围内有效，也就是说只有在本函数内才能使用它们，在此函数以外时不能使用这些变量的，它们称为局部变量。

1. 主函数 `main` 中定义的变量也只在主函数中有效，而不因为在主函数中定义而在整个文件或程序中有效。
2. 不同函数中可以使用名字相同的变量，它们代表不同的对象，互不干扰。
3. 形式参数也使局部变量。
4. 在一个函数内部，可以在复合语句中定义变量，这些变量只在本复合语句中有效。

(2)全局变量

在函数外定义的变量是外部变量，外部变量是全局变量，全局变量可以为本文件中其它函数所共用，它的有效范围从定义变量的位置开始到本源文件结束。

1. 设全局变量的作用：增加了函数间数据联系的渠道。
2. 建议不再必要的时候不要使用全局变量，因为
 - a. 全局变量在程序的全部执行过程中都占用存储单元。
 - b. 它使函数的通用性降低了。
 - c. 使用全局变量过多，会降低程序的清晰性。
3. 如果外部变量在文件开头定义，则在整个文件范围内都可以使用该外部变量，如果不在文件开头定义，按上面规定作用范围只限于定义点到文件终了。如果在定义点之前的函数想引用该外部变量，则应该在该函数中用关键字 `extern` 作外部变量说明。
4. 如果在同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量不起作用。

(3) 静态变量

静态变量的作用范围要看静态变量的位置，如果在函数里，则作用范围就是这个函数。

静态全局变量，只在本文件可以用，虽然整个程序包含多个文件，但静态全局变量只能用在定义它的那个文件里，却不能用在程序中的其他文件里。它是定义存储因型为静态型的外部变量，其作用域是从定义点到程序结束，所不同的是存储类型决定了存储地点，静态型变量是存放在内存的数据区中的，它们在程序开始运行前就分配了固定的字节，在程序运行过程中被分配的字节大小是不改变的，只有程序运行结束后，才释放所占用的内存。

【attention】

操作系统和编译器如何判断全局变量和局部变量？

操作系统和编译器是根据程序运行的内存区域来获取该变量的类型。程序的全局数据放在所分配内存的全局数据区，程序的局部数据放在栈区。

【attention】

<1>`static` 全局变量与普通的全局变量有什么区别？

全局变量(外部变量)的说明之前再冠以 `static` 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其它源文件中引起错误。从以上分析可以看出，把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。

<2>`static` 函数与普通函数有什么区别？

`static` 函数与普通函数作用域不同。仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(`static`)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

<3>static 局部变量和普通局部变量有什么区别？

static 局部变量只被初始化一次，下一次依据上一次结果值；

1.12.面试题集合（十一）

1.12.1. 有两个双向循环链表 A, B, 知道其头指针为： pHeadA,pHeadB, 请写一函数将两链表中 data 值相同的结点删除

有双向循环链表结点定义为：

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. struct node
2. { int data;
3. struct node *front,*next;
4. };
5. 有两个双向循环链表 A, B, 知道其头指针为： pHeadA,pHeadB, 请写一函数将两链表中 data 值
    相同的结点删除
6. BOOL DeleteNode(Node *pHeader, DataType Value)
7. {
8.     if (pHeader == NULL) return;
9.
10.    BOOL bRet = FALSE;
11.    Node *pNode = pHead;
12.    while (pNode != NULL)
13.    {
14.        if (pNode->data == Value)
15.        {
16.            if (pNode->front == NULL)
17.            {
18.                pHeader = pNode->next;
19.                pHeader->front = NULL;
20.            }
21.            else
22.            {
```

```
23. if (pNode->next != NULL)
24. {
25.     pNode->next->front = pNode->front;
26. }
27. pNode->front->next = pNode->next;
28. }
29.
30. Node *pNextNode = pNode->next;
31. delete pNode;
32. pNode = pNextNode;
33.
34. bRet = TRUE;
35. //不要 break 或 return, 删除所有
36. }
37. else
38. {
39.     pNode = pNode->next;
40. }
41. }
42.
43. return bRet;
44. }
45.
46. void DE(Node *pHeadA, Node *pHeadB)
47. {
48.     if (pHeadA == NULL || pHeadB == NULL)
49.     {
50.         return;
51.     }
52.
53.     Node *pNode = pHeadA;
54.     while (pNode != NULL)
55.     {
56.         if (DeleteNode(pHeadB, pNode->data))
57.     }
```

```
58. if (pNode->front == NULL)
59. {
60.     pHeadA = pNode->next;
61.     pHeadA->front = NULL;
62. }
63. else
64. {
65.     pNode->front->next = pNode->next;
66.     if (pNode->next != NULL)
67.     {
68.         pNode->next->front = pNode->front;
69.     }
70. }
71. Node *pNextNode = pNode->next;
72. delete pNode;
73. pNode = pNextNode;
74. }
75. else
76. {
77.     pNode = pNode->next;
78. }
79. }
80. }
```

1.12.2. 找出两个字符串中最大公共子字符串，如

"abccade","dgcadde"的最大子串为"cad"

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. int GetCommon(char *s1, char *s2, char **r1, char **r2)
2. {
3.     int len1 = strlen(s1);
4.     int len2 = strlen(s2);
```

```
5. int maxlen = 0;
6.
7. for(int i = 0; i < len1; i++)
8. {
9.     for(int j = 0; j < len2; j++)
10.    {
11.        if(s1[i] == s2[j])
12.        {
13.            int as = i, bs = j, count = 1;
14.            while(as + 1 < len1 && bs + 1 < len2 && s1[++as] == s2[++bs])
15.                count++;
16.
17.            if(count > maxlen)
18.            {
19.                maxlen = count;
20.                *r1 = s1 + i;
21.                *r2 = s2 + j;
22.            }
23.        }
24.    }
25. }
```

1.12.3. 把十进制数(long型)分别以二进制和十六进制形式输出，不能使用 printf 系列

编程实现：把十进制数(long型)分别以二进制和十六进制形式输出，不能使用 printf 系列。

实现了 unsigned long 型的转换。

[\[cpp\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. // 十进制转换为二进制，十进制数的每 1bit 转换为二进制的 1 位数字
2. char *int_to_bin(unsigned long data)
3. {
```

```

4.     int bit_num = sizeof(unsigned long) * 8;
5.     char *p_bin = new char[bit_num+1];
6.     p_bin[bit_num] = '\0';
7.     for (unsigned int i = 0; i < bit_num; ++i)
8.     {
9.         p_bin[i] = data << i >> (bit_num-1);
10.        if (p_bin[i] == 0)
11.            p_bin[i] = '0';
12.        else if (p_bin[i] == 1)
13.            p_bin[i] = '1';
14.        else
15.            p_bin[i] = 'a';
16.    }
17.    return p_bin;
18. }
19.
20. // 十进制转换为十六进制，十进制数的每 4bit 转换为十六进制的 1 位数字
21. char *int_to_hex(unsigned long data)
22. {
23.     int bit_num = sizeof(unsigned long) * 8;
24.     char *p_hex = new char[sizeof(unsigned long)*8/4+3];
25.     p_hex[0] = '0';
26.     p_hex[1] = 'x';
27.     p_hex[bit_num/4+2] = '\0';
28.     char *p_tmp = p_hex + 2;
29.     for (unsigned int i = 0; i < bit_num/4; ++i)
30.     {
31.         p_tmp[i] = data << (4*i) >> (bit_num-4);
32.         if (p_tmp[i] >= 0 && p_tmp[i] <= 9)
33.             p_tmp[i] += '0';
34.         else if (p_tmp[i] >= 10 && p_tmp[i] <= 15)
35.             p_tmp[i] = p_tmp[i] - 10 + 'A';
36.     }
37.     return p_hex;
38. }
```

printf("%x",d)直接把 d 用 16 进制输出了 题目有着要求是叫你自己写 不要用 printf 投机取巧

实际上你这题目的算法用到位运算 整数是 32 个 bit

16 进制实际上 是 4 个 bit 4 个比特一组 一共 8 组

每次把 4 个 bit 取出来做成一个整数 输出就行了

追问

能否详细介绍以下三行代码

```
temp[i] = (char)(num<<4*i>>28);
temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
```

回答

temp[i] = (char)(num<<4*i>>28); 左移 4i 位 去掉高位的 4i 右移 28 位 去掉右边的 这就获得 4 个 bit

第一次是最高位 32-29 4 个 bit 第二次 28-25 4 个 bit

```
temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
```

然后因为它可能是个有符号的数 所以加上 16 吧这个数变成 0~15 之间的正数

```
temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
```

然后 <10 就是变成字符‘0’~‘9’ 所以加上 4~8， 如果 >= 10 那就要用 'A'~'F' 来表示

所以加上 55， 因为'A'变成整数正好是 65

1.12.4. 40 亿个整数

这个 CSDN 上看到的腾讯面试题，总算是搞清楚了！

以下转：

一个文件中有 40 亿个整数，每个整数为四个字节，内存为 1GB，写出一个算法：求出这个文件里的整数里不包含的一个整数

4 个字节表示的整数，总共只有 2^{32} 约等于 4G 个可能。

为了简单起见，可以假设都是无符号整数。

分配 500MB 内存，每一 bit 代表一个整数，刚好可以表示完 4 个字节的整数，初始值为 0。

基本思想每读入一个数，就把它对应的 bit 位置为 1，处理完 40G 个数后，对 500M 的内存遍历，找出一个 bit 为 0 的位，输出对应的整数就是未出现的。

算法流程：

1) 分配 5 0 0 M B 内存 buf, 初始化为 0

2) unsigned int x=0x1;

for each int j in file

buf=buf|x < <j;

end

(3) for(unsigned int i=0; i <= 0xffffffff; i++)

if (!(buf & x < <i))

{

output(i);

break;

}

以上只是针对无符号的，有符号的整数可以依此类推。

该题有两种比较优良的算法。。其中一种就是楼上所述的位图操作。。另一种就是使用很少的内存，预料不到的二分法，因为不存在的数必然在最大数和最小数及中间数的两个范围里，必然较小的范围里，通过不断缩短范围，就可以求出来。。。

这里只给出第一种算法。。

详细见《编程珠玑》

位图运算

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */
/* bitsort.c -- bitmap sort from Column 1 * Sort distinct integers in the range [0..N-1] */
#include <stdio.h>
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 10000000
int a[1 + N/BITSPERWORD];
void set(int i) { a[i>>SHIFT] |=(1 <<(i & MASK)); }
void clr(int i) { a[i>>SHIFT] &= ~ (1 <<(i & MASK)); }
int test(int i){ return a[i>>SHIFT] & (1 <<(i & MASK)); }
int main(){
int i;
for (i = 0; i < N; i++)
clr(i);
```

```
while (scanf("%d", &i) != EOF)
set(i);
for (i = 0; i < N; i++)
if (test(i))
printf("%d\n", i);
return 0;
}

=====
```

本文是在

<http://blog.csdn.net/king821221/archive/2008/01/20/2054353.aspx>

结合自己总结完成的

感谢 csdn 感谢...

案例 1 :假设一个文件中有 9 亿条不重复的 9 位整数，现在要求对这个文件进行排序。

方法:bit 位操作

首先

32 位机的寻址能力是 2 的 32 次方，即 4G. 寻址能力最大是这样了.

一个最大的 9 位整数为 999999999

这 9 亿条数据是不重复的

声明一个 bit 数组,长度为 10 亿

一共需要 $10 \text{ 亿} / 8 / 1024 / 1024 = 120\text{M}$

把内存中的数据全部初始化为 0

读取文件中的数据，并将数据放入内存。比如读到一个数据为 341245909 这个数据，那就先在内存中找到 341245909 这个 bit，并将 bit 值置为 1

遍历整个 bit 数组，将 bit 为 1 的数组下标存入文件

然后再遍历一次,找出不为零的就可以了.

注意:

如果是有重复的数字,

需要考虑重复的情况

如果普遍多于 2 个,那么就要设置包含 2 个 bit 的数组

或者使用 bitmap 类型.

案例 2

最快的方法找出一篇文章中各个字符的出现次数

声明数组 int[256]

以该字符的 aci 码作为下标即可

案例 3 全排列

a,b,c ,d, e 的所有排列

令 a =1 , b = 2, c = 3, ...

那么从 12345 到 54321 的所有数打印出来就可以了

如果有别的限制,比如 ac 不相邻

那么就再转换成字符串判断好了.

案例 4 组合

a b c d e 5 个字母中选 3 个的组合

可以使用

a b c d e ----- 数组 R(i)

a b c d e ----- 数组 S(j)

a b c d e ----- 数组 T(k)

模拟 5 进制的加法,R,S,T 下标从 0 开始,然后 T 下标加一,加到 5 的话就让 s 下标加一

还有,既然是组合,就要保证 s 的下标从 R 的开始,T 的从 S 的开始

```
for ( i = 0 ; i < 5; i ++)
```

```
    for ( j = i ; j < 5; j ++)
```

```
        for ( k = j; k < 5; k ++)
```

```
        ...{
```

```
            // 打印 R(i)S(j)T(k)
```

```
        }
```

案例 5 把 m 个球分 n 组

0 1 2 ... m

...

0 1 2 ... m

总共 n 行

模拟 M 进制的加法即可

(其实和回溯法一个原理,只是这个更好理解)

案例 6 换零钱问题

有足够的 1 元,5 元,10 元,20 元硬币

组合成 170 元有多少方法?

其实就是解方程

$$1 * x_1 + 5 * x_2 + 10 * x_3 + 20 * x_4 = 170$$

然后试解循环法解 x1,x2,x3,x4

案例 6 一个文件中有 40 亿个整数，每个整数为四个字节，内存为 1GB，写出一个算法：求出这个文件里的整数里不包含的一个整数

如果采用 bit 数组, 40 亿个 bit 也就 $120 \times 4 = 480M$ 的空间, 所以 1G 绰绰有余.

而 40 亿的整数占用的是 $480M \times 32 = 15360M$ 即大约 15G, 问题是怎样把这数字读进来

可以一个一个的读.

另外 csdn 上有个方法, 不知道在这里有什么帮助, 不过原理还是不错:

使用分段的方法:

因为 4 字节的整数可以用 16 进制表示为

00000000H 到 FFFFFFFFH

我们可以分成 1000H 为一段, 把整数分割,

统计每个区间的占有情况

如果是 1000H, 那么统计不包含的数字就略过这个区间好了

但是这个前提还是要读进来吧?

1.12.5. bitmap 减少 hash 算法所用空间

主要讨论了一个对一个包含 1000 万条的整数文件排序问题, 很有 baidu 面试题的风格。

主要思想是 hash 来解决排序问题, 但是 hash 的空间复杂度又相对比较大, 所以用 [bitmap](#) 来减少 hash 算法所需的空间。

一般的 hash, 例如对数组[2, 3, 5, 10] 运用桶排序算法, 需要声明 10 个整数的 bucket, 如下图所示:

0	1	1	0	1	0	0	0	0	1
1	2	3	4	5	6	7	8	9	10

但是如果用 bitmap 的话, 只需一个整数, 因为一个整数(32 位机)的话有 32bit, 每个 bit 都能 map 一个整数, 如下图所示:

									1				1		1	1	
31	30	...	3	2	1	0	31	...	9	...	5	4	3	2	1	0	
数组元素 1										数组元素 0							

所以关键就是位操作 set, clear, test:

01#define BITSPERWORD 32

02#define SHIFT 5

03#define MASK 0x1f

```

04
05typedef long long int64;
06
07int array[100000];
08
09/**
10*
11* i / 32 对应具体数组元素
12
13* i % 32 对应具体数组元素的 bit 位
14* i >> SHIFT == i / 32
15* i & MASK == i % 32
16* (32) = (100000)
17*
18*/
19
20void set(int64 i)
21{
22    array[ i >> SHIFT] |= ( 1 << ( i & MASK));
23}
24
25void clear(int64 i)
26{
27    array[ i >> SHIFT] &= ~( 1 << ( i & MASK));
28}
29
30int test(int64 i)
31{
32    return array[ i >> SHIFT] & ( 1 << ( i & MASK));
33}

```

在一个文件中有 10G 个整数，乱序排列，要求找出中位数。内存限制为 2G。只写出思路即可。

10G 整数做 bitmap 需要 $10G/32 = 0.3G$ 个整数(< 32 位机器总共可以的 $2^{31}-1 = 2G$ 个整数)，
 0.3G 个整数只需要 $0.3G \times 4 = 1.2G$ 的存储空间，故可以只扫描一遍，就可以求出中位数。

[java] [view plain](#) [copy](#) [print](#)?

```

1.  public class BitMap {
2.      private static byte[] bitMap = null;

```

```
3.  
4.     public BitMap(int size) {  
5.         if (size % 8 == 0) {  
6.             bitMap = new byte[size / 8];  
7.         } else {  
8.             bitMap = new byte[size / 8 + 1];  
9.         }  
10.    }  
11.  
12.    public static void main(String[] args) {  
13.        BitMap map = new BitMap(10);  
14.        map.setTag(11);  
15.        map.printBitMap();  
16.        String string = "";  
17.        for (int i = 0; i < bitMap.length; i++) {  
18.            System.out.println(string + bitMap[i]);  
19.        }  
20.    }  
21.  
22.    public void setTag(int number) {  
23.        int index = 0;  
24.        int bit_index = 0;  
25.        if (number % 8 == 0) {  
26.            index = number / 8 - 1;  
27.            bit_index = 8;  
28.        } else {  
29.            index = number / 8;  
30.            bit_index = number % 8;  
31.        }  
32.        switch (bit_index) {  
33.            case 1:  
34.                bitMap[index] = (byte) (bitMap[index] | 0x01);  
35.                break;  
36.            case 2:  
37.                bitMap[index] = (byte) (bitMap[index] | 0x02);
```

```
38.         break;
39.     case 3:
40.         bitMap[index] = (byte) (bitMap[index] | 0x04);
41.         break;
42.     case 4:
43.         bitMap[index] = (byte) (bitMap[index] | 0x08);
44.         break;
45.     case 5:
46.         bitMap[index] = (byte) (bitMap[index] | 0x10);
47.         break;
48.     case 6:
49.         bitMap[index] = (byte) (bitMap[index] | 0x20);
50.         break;
51.     case 7:
52.         bitMap[index] = (byte) (bitMap[index] | 0x40);
53.         break;
54.     case 8:
55.         bitMap[index] = (byte) (bitMap[index] | 0x80);
56.         break;
57.     }
58.
59. }
60.
61. public void printBitMap() {
62.     int size = bitMap.length;
63.     for (int i = 0; i < size; i++) {
64.
65.         if ((bitMap[i] & 0x01) == 1) {
66.             System.out.print(i * 8 + 1 + " ");
67.         }
68.         if ((bitMap[i] >> 1 & 0x01) == 1) {
69.             System.out.print(i * 8 + 2 + " ");
70.         }
71.         if ((bitMap[i] >> 2 & 0x01) == 1) {
72.             System.out.print(i * 8 + 3 + " ");
```

```

73.     }
74.     if ((bitMap[i] >> 3 & 0x01) == 1) {
75.         System.out.print(i * 8 + 4 + " ");
76.     }
77.
78.     if ((bitMap[i] >> 4 & 0x01) == 1) {
79.         System.out.print(i * 8 + 5 + " ");
80.     }
81.     if ((bitMap[i] >> 5 & 0x01) == 1) {
82.         System.out.print(i * 8 + 6 + " ");
83.     }
84.     if ((bitMap[i] >> 6 & 0x01) == 1) {
85.         System.out.print(i * 8 + 7 + " ");
86.     }
87.     if ((bitMap[i] >> 7 & 0x01) == 1) {
88.         System.out.print(i * 8 + 8 + " ");
89.     }
90. }
91. System.out.println();
92. }
93.
94. }

```

1.12.6. 定义一个类似函数的宏，宏运算的结果来表示大于和小于

就是定义一个类似函数的宏，宏运算的结果来表示大于和小于。为了简单起见我们假设 a,b 为整型。

我是这样想的，如果可以用小于号的话，可以这么写：

```
#define compare(a,b) ((a-b)<0 ? -1 : ((a-b) == 0 ? 0 : 1))
```

这样一来的话：

compare(a,b) == -1 表示 a<b

compare(a,b) == 0 表示 a==b

compare(a,b) == 1 表示 a>b

但是如今不能用小于号,那么我们怎样不用小于号来判断一个数字是否小于 0 呢? 我们可以用: `abs(t) != t ? 1 : -1` 来表示。

也即如果 `abs(t) != t` 那么 $t < 0$, 否则 $t \geq 0$ 。也即 $t < 0$ 等价于 $(abs(t) != t ? 1 : -1) == 1$ 那么前面的宏写成如下:

```
#define compare(a,b) ((abs(a-b) != (a-b) ? 1 : -1) == 1 ? -1 : ((a-b) == 0 ? 0 : 1))
```

代码:

```
//利用宏比较大小
```

```
#include <iostream>
```

```
#include <cmath>
```

```
#define compare(a,b) ((abs(a-b) != (a-b) ? 1 : -1) == 1 ? -1 : ((a-b) == 0 ? 0 : 1))
```

```
int main()
```

```
{
```

```
    int a1 = -1, b1 = 2, a2 = 3, b2 = 3, a3 = 4, b3 = 2;
```

```
    cout << compare(a1, b1) << endl << compare(a2, b2) << endl << compare(a3, b3) << endl;
```

```
    return 0;
```

```
}
```

```
#define max(a,b) ((a-b)&(1<<31))?b:a
```

判断相减后的符号位

1.12.7. 给定一个集合 A

给定一个集合 $A=[0,1,3,8]$ (该集合中的元素都是在 0, 9 之间的数字, 但未必全部包含), 指定任意一个正整数 K , 请用 A 中的元素组成一个大于 K 的最小正整数。比如, $A=[1,0]$ $K=21$ 那么输出结构应该为 100。

1. 一个从小到大排序的整数数组, 元素都是在[0,9]之间的数字, 但未必全部包含
2. // 用数组中的数字(可以重复)组成一个最小的给定位数的正整数
3. `int generate_min_int_containing_duplicate_digit(int *array, int n, int bit_num)`
4. {
5. // 找到最小的非零整数

```
6.     int data;
7.     for (int i = 0; i < n; ++i)
8.     {
9.         if (array[i] > 0)
10.        {
11.            data = array[i];
12.            break;
13.        }
14.    }
15.    for (int i = 1; i < bit_num; ++i)
16.    {
17.        data = data * 10 + array[0];
18.    }
19.    return data;
20. }
21.
22.
23. // Google2009 华南地区笔试题
24. // 给定一个集合 A=[0,1,3,8](该集合中的元素都是在 0, 9 之间的数字, 但未必全部包含),
25. // 指定任意一个正整数 K, 请用 A 中的元素组成一个大于 K 的最小正整数。
26. // 比如, A=[1,0] K=21 那么输出结构应该为 100。
27. int generate_min_int_greater_than_k(int *array, int n, int k)
28. {
29.     std::sort(array, array+n);
30.     // high_digit: k 的最高位数字
31.     // bit_num: k 的位数
32.     int high_digit = k, bit_num = 1;
33.     while (high_digit/10 > 0)
34.     {
35.         ++bit_num;
```

```
36.     high_digit /= 10;
37. }
38. // 查找数组中比 k 的最高位大的最小的数字
39. int i;
40. for (i = 0; i < n; ++i)
41. {
42.     if (array[i] >= high_digit)
43.         break;
44. }
45. if (i == n) // 数组中的数字都比 K 的最高位小
46. {
47.     return generate_min_int_containing_duplicate_digit(array, n, bit_num+1);
48. }
49. else if (array[i] == high_digit)// 数组中有一位数字跟 K 的最高位相等
50. {
51.     int low_data = k - high_digit * pow(10, bit_num-1);
52.     return array[i]*pow(10, bit_num-1)+generate_min_int_greater_than_k(array, n, low_data);
53. }
54. else // 数组中有一位数字比 k 的最高位高
55. {
56.     int data = array[i];
57.     for (int j = 1; j < bit_num; ++j)
58.     {
59.         data = data * 10 + array[0];
60.     }
61.     return data;
62. }
63. return 0;
64. }
```

1.12.8. 已知一个函数 f 可以等概率的得到 1-5 间的随机数，问怎么等概率的得到 1-7 的随机数

已知一个函数 f 可以等概率的得到 1-5 间的随机数，问怎么等概率的得到 1-7 的随机数

```
int rand7()
{
    int a;
    while( (a=rand5()*5+rand5()) > 26 );
    return (a-3)/3;
}
```

算法思路是：

1. 通过 `rand5()*5+rand5()` 产生 6 7 8 9 10 11 26, 27 28 29 30 这 25 个数，每个数的出现机率相等
2. 只需要前面 3*7 个数，所以舍弃后面的 4 个数
3. 将 6 7 8 转化为 1, 9 10 11 转化为 2, , 24 25 26 转化为 7。公式是 $(a-3)/3$

只要我们可以从 n 个数中随机选出 1 到 n 个数，反复进行这种运算，直到剩下最后一个数即可。

我们可以调用 n 次给定函数，生成 n 个 1 到 5 之间的随机数，选取最大数所在位置即可满足以上要求。

例如

初始的 7 个数 [1,2,3,4,5,6,7].

7 个 1 到 5 的随机数 [5, 3, 1, 4, 2, 5, 5]

那么我们保留下[1,6,7],

3 个 1 到 5 的随机数[2,4,1]

那么我们保留下[6]

6 就是我们这次生成的随机数。

编写一个生成 0 和 1 的随机函数：

step1. 调用给定的随机函数 `original_rand()`生成一个数

如果==3 goto step1

如果<3 return 0

如果 >3 return 1

编写一个生成 1 到 7 的随机函数

调用生成 0 和 1 的随机函数 3 次，构成 000 或 001 或 010.....

如果？？？不等于 0 返回，否则重新生成。

代码如下：

```
int rand_01()
{
    int r = original_rand();
    if(r == 3) return rand_01();
    if(r < 3) return 0;
    if(r > 3) return 1;
}

int rand_17()
{
    int i = 0;
    i += rand_01();
    i += rand_01() << 1;
    i += rand_01() << 2;
    if(i == 0) return rand_17();
    return i;
```

1.12.9. 判断一个自然数是否是某个数的平方

$1+3+5+7+\dots+(2n-1)=n^2$ 完全平方数是这样一种数：

它可以写成一个正整数的平方

//vc6.0 c++下实现

```
#include<iostream.h>
void main(){
    long m;
    cin>>m;
    for (long i=1,temp=0; i<m; i++)
    {
        temp=temp + 2*i-1;
        if (temp==m){
            cout<<m<<"="<<i<<"*"<<i<<endl; break;
        }
    }
}
```

```

    }
    if (temp>m){
        cout<<"no"<<endl; break;
    }
}
}

```

输入 10000 结果 10000=100*100

1.12.10.一棵排序二叉树，令 $f=(\text{最大值}+\text{最小值})/2$ ，设计一个算法，找出距离 f 值最近、大于 f 值的结点。复杂度如果是 $O(n^2)$ 则不得分。

```

1. #include "stdafx.h"
2. #include <iostream>
3. #include <cassert>
4. #include <iterator>
5. using namespace std;
6. struct Node
7. {
8.     int elememt;
9.     Node *pLeft;
10.    Node *pRight;
11.    Node(int ele, Node *left = NULL, Node *right = NULL)
12.        :elememt(ele), pLeft(left), pRight(right){ }
13. };
14. //插入元素
15. void InsertNode(Node *&root, int ele)
16. {
17.     if(NULL == root)
18.         root = new Node(ele);
19.     else if (root->elememt > ele)
20.         InsertNode(root->pLeft, ele);
21.     else if (root->elememt < ele)
22.         InsertNode(root->pRight, ele);
23. }

```

```
24. //创建二叉查找树
25. void CreateTree(Node *&root, int arr[], int len)
26. {
27.     for (int i = 0; i < len; i++)
28.         InsertNode(root, arr[i]);
29. }
30. //中序输出树
31. void MiddlePrint(Node *root)
32. {
33.     if (NULL == root)
34.         return;
35.     MiddlePrint(root->pLeft);
36.     cout<<root->element<<" ";
37.     MiddlePrint(root->pRight);
38. }
39. //获得最大值
40. int GetMaxValue(Node *root)
41. {
42.     assert(root != NULL);
43.     while (root->pRight != NULL)
44.         root = root->pRight;
45.     return root->element;
46. }
47. //获得最小值
48. int GetMinValue(Node *root)
49. {
50.     assert(NULL != root);
51.     while (root->pLeft != NULL)
52.         root = root->pLeft;
53.     return root->element;
54. }
55. //查找指定的结点
56. Node *FindNode(Node *root, int target)
57. {
58.     assert(NULL != root);
```

```

59.     int diff = INT_MAX;
60.     Node *p = NULL;
61.     while (root != NULL)
62.     {
63.         if (root->elementt > target)
64.         {
65.             if (root->elementt - target < diff)
66.             {
67.                 diff = root->elementt - target;
68.                 p = root;
69.             }
70.             root = root->pLeft;
71.         }
72.         else if (root->elementt < target)
73.         {
74.             root = root->pRight;
75.         }
76.     }
77.     return p;
78. }
79. int main()
80. {
81.     int arr[] = {20, 8, 22, 4, 12, 10, 14};
82.     int len = sizeof(arr) / sizeof(int);
83.     Node *root = NULL;
84.     CreateTree(root, arr, len);
85.     MiddlePrint(root);
86.     printf("\n");
87.     int minValue = GetMinValue(root);
88.     int maxValue = GetMaxValue(root);
89.     Node *node = FindNode(root, (minValue + maxValue) / 2);
90.     assert(NULL != node);
91.     printf("%d\n", node->elementt);
92. }
```

1.12.11 strstr 和 strcmp 源码实现

extern char *strstr(char *s1, char *s2)

用法: #include <string.h>

功能: 找出 s2 字符串在 s1 字符串中第一次出现的位置 (不包括 s2 的串结束符)

返回结果: 返回该位置的指针, 如找不到, 返回空指针。

源码实现:

```
char *strstr( const char *s1, const char *s2 )
{
    int len2;
    if ( !(len2 = strlen(s2)) )
        return (char *)s1;
    for ( ; *s1; ++s1 )
    {
        if ( *s1 == *s2 && strncmp( s1, s2, len2 )==0 )
            return (char *)s1;
    }
    return NULL;
}
```

int strcmp (char * s1, char * s2, size_t n)

用法: #include <string.h>

功能: 比较字符串 s1 和 s2 的前 n 个字符.

返回结果: 如果前 n 字节完全相等, 返回值就=0; 在前 n 字节比较过程中, 如果出现 s1[n] 与 s2[n] 不等, 则返回(s1[n]-s2[n])

源码实现:

```
int strcmp ( char * s1, char * s2, size_t n)
{
    if ( !n ) //n 为无符号整形变量;如果 n 为 0,则返回 0
        return(0);
    //在接下来的 while 函数中
    //第一个循环条件: --n,如果比较到前 n 个字符则退出循环
    //第二个循环条件: *s1,如果 s1 指向的字符串末尾退出循环
    //第三个循环条件: *s1 == *s2,如果两字符比较不等则退出循环
```

```

while (--n && *s1 && *s1 == *s2)
{
    s1++;//S1 指针自加 1,指向下一个字符
    s2++;//S2 指针自加 1,指向下一个字符
}
return( *s1 - *s2 );//返回比较结果
}

```

1.13.面试题集合（十二）

1.13.1. 对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集合的数字和是相等

1.13.1. 对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集合的数字和是相

2011-05-31 21:45:28| 分类： [算法设计](#) | 标签： |字号大中小 [订阅](#)

对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集合的数字和是相等的。

举个例子，如果 N=3，对于{1, 2, 3}能划分成两个子集合，他们每个的所有数字和是相等的：

{3} and {1,2}

这是唯一一种分发（交换集合位置被认为是同一种划分方案，因此不会增加划分方案总数）

如果 N=7，有四种方法能划分集合{1, 2, 3, 4, 5, 6, 7}，每一种分发的子集合各数字和是相等的：

{1,6,7} and {2,3,4,5} {注 1+6+7=2+3+4+5}

{2,5,7} and {1,3,4,6}

{3,4,7} and {1,2,5,6}

{1,2,4,7} and {3,5,6}

给出 N，你的程序应该输出划分方案总数，如果不存在这样的划分方案，则输出 0。程序不能预存结果直接输出。

PROGRAM NAME: subset

INPUT FORMAT

输入文件只有一行，且只有一个整数 N

SAMPLE INPUT (file subset.in)

7

OUTPUT FORMAT

输出划分方案总数，如果不存在则输出 0。

SAMPLE OUTPUT (file subset.out)

4

/* 1 到 n 的自然数之和为： 1+2+3.....+n=n*(n+1)/2

题目要求：对于从 1 到 N 的连续整集合，划分为两个子集合，且保证每个集合的数字和是相等的。因而，划分之后每个子集全的数字应该为 $n*(n+1)/2$ 的一半，即 $n*(n+1)/4$

由于两个子集中都是整数，所以 $n*(n+1)$ 必为偶数，则可以设 $s=n*(n+1)$ ，并判断 $s \% 4$ 。

则， $s/4$ 是划分之后子集合的数字和； $dyn[i]$ 数组表示任意个数加起来等于 i 的组数*/

动态规划

```
#include<iostream>
#include<malloc.h>
using namespace std;
int main()
{
    int dyn[100];
    memset(dyn,0,100);
    int n,s;
    cin>>n;
    s=n*(n+1);
    if(s%4)
    {
        cout<<0<<endl;
    }
    s/=4;
    int i,j;
    dyn[0]=1;
    for(i=1;i<=n;i++)//表示这个次选取的是数为 i
    {
        for(j=s;j>=i;j--)
            dyn[j]=dyn[j]+dyn[j-i];
    }
}
```

```

dyn[j]+=dyn[j-i];//dyn[j-i]表示加起来等于 j-i 的组数,
}
//dyn[j]表示加起来等于 j 的组数
cout<<dyn[s]/2<<endl;//由于交换两个子集合的位置被认为是同一种划分方案，所以最终结
果为 dyn[s]/2

return 1;
}

//****************************************************************************
/* 背包方法
可以将问题转变为 01 背包恰好装满的情况；将原集合分为两部分，每部分中数的和均
为 n(n+1)/4，则问题转变为从原集合中取物品，放入背包容量为 n(n+1)/4 的背包中，且
恰好装满，有多少种取法。最后将所求得的方法数除以 2 即为可划分的子集合数*/
//****************************************************************************

#include<iostream>
#include<malloc.h>
using namespace std;

int main()
{
    int Pos[10];
    int top=0;
    int Array[]={1,2,3,4,5,6,7};
    int m=14;//背包容量 m=n(n+1)/4;
    int n=7;//物品个数
    int i,j;
    int count=0;//满足条件的方法数
    i=0;
    memset(Pos,0,sizeof(Pos));
    while(i<=n&&top>=0)
    {
        while(m>0&&i<n)
        {
            if(Array[i]<=m)
            {
                Pos[top++]=i;
                m-=Array[i];
            }
        }
        count+=Pos[top];
        m+=Array[i];
        i++;
    }
    cout<<count/2<<endl;
}

```

```

    }
    i++;
}
if(m==0)
{
    for(j=0;j<top;j++)
    {
        cout<<" "<<Array[Pos[j]];
    }
    cout<<endl;
    count++;
}
i=Pos[--top];
if(top<0)break;
Pos[top]=0;
m+=Array[i];
i++;
}
cout<<"可分为组数: "<<count/2;
}

```

1.13.2. Topk

面试到了一个 topk，这个原理很简单，但是以前很少写过。面试时写的有点小慢，没有达到行云流水的地步。于是回来再写一遍练练。其中，堆排序部分采用[简明排序代码](#)。用完整的 TopK 代码：

```

#include <iostream>
#include <algorithm>

using namespace std;

template<typename T>
void unguarded_heapify(T *data, size_t size, size_t top)
{
    while (true)
    {

```

```
size_t min = top;

    if (top * 2 < size && data[top * 2] < data[min])
    {
        min = top * 2;
    }

    if (top * 2 + 1 < size && data[top * 2 + 1] < data[min])
    {
        min = top * 2 + 1;
    }

    if (top == min) return;

    swap(data[top], data[min]);
    top = min;
}

template<typename T>
void make_min_heap(T *begin, T *end)
{
    if (begin == NULL || end == NULL)
    {
        return;
    }

    if (begin == end || begin + 1 == end)
    {
        return;
    }

    size_t len = end - begin;

    for (size_t top = len / 2; top >= 1; --top)
    {
        // Special offset.
        unguarded_heapify(begin - 1, len + 1, top);
    }
}

void topk(const int *begin, const int *end, int *buffer, size_t *k)
{
    if (begin == NULL || end == NULL || buffer == NULL || k == NULL)
```

```
{  
    return;  
}  
  
if (begin == end || *k == 0)  
{  
    return;  
}  
  
memset(buffer, 0, *k * sizeof(int));  
  
const int *p = begin;  
int *dest = buffer;  
  
while (p != begin + *k && p != end)  
{  
    *dest++ = *p++;  
}  
  
if (p == end)  
{  
    *k = end - begin;  
}  
else  
{  
    make_min_heap(buffer, dest);  
  
    while (p != end)  
    {  
        if (*p > *buffer)  
        {  
            *buffer = *p;  
            unguarded_heapify(buffer - 1, *k + 1, 1);  
        }  
  
        ++p;  
    }  
}  
}  
  
int main(int argc, char **argv)  
{  
    int data[] = {4, 5, 1, 3, 5, 6, 7, 2};  
    int *result = new int[10];
```

```

size_t k = 3;
topk(data, data + sizeof(data) / sizeof(data[0]), result, &k);
copy(result, result + k, ostream_iterator<int>(cout, " "));
cout << endl;

k = 10;
topk(data, data + sizeof(data) / sizeof(data[0]), result, &k);
copy(result, result + k, ostream_iterator<int>(cout, " "));
cout << endl;

k = 1;
topk(data, data + sizeof(data) / sizeof(data[0]), result, &k);
copy(result, result + k, ostream_iterator<int>(cout, " "));
cout << endl;

k = 8;
topk(data, data + sizeof(data) / sizeof(data[0]), result, &k);
copy(result, result + k, ostream_iterator<int>(cout, " "));
cout << endl;

k = 0;
topk(data, data + sizeof(data) / sizeof(data[0]), result, &k);
copy(result, result + k, ostream_iterator<int>(cout, " "));
cout << endl;

delete[] result;
return 0;
}

```

1.13.3. Collection

在 Java2 中，有一套设计优良的接口和类组成了 Java 集合框架 Collection，使程序员操作成批的数据或对象元素极为方便。这些接口和类有很多对抽象数据类型操作的 API，而这是我们常用的且在数据结构中熟知的。例如 Map，Set，List 等。并且 Java 用面向对象的设计对这些数据结构和算法进行了封装，这就极大的减化了程序员编程时的负担。程序员也可以以这个集合框架为基础，定义更高级别的数据抽象，比如栈、队列和线程安全的集合等，从而满足自己的需要。

Java2 的集合框架，抽其核心，主要有三种：List、Set 和 Map。如下图所示：

需要注意的是，这里的 Collection、List、Set 和 Map 都是接口（Interface），不是具体的类实现。 List lst = new ArrayList(); 这是我们平常经常使用的创建一个新的 List 的语句，在这里， List 是接口， ArrayList 才是具体的类。

常用集合类的继承结构如下：

```
Collection<--List<--Vector  
Collection<--List<--ArrayList  
Collection<--List<--LinkedList  
Collection<--Set<--HashSet  
Collection<--Set<--HashSet<--LinkedHashSet  
Collection<--Set<--SortedSet<--TreeSet  
Map<--SortedMap<--TreeMap  
Map<--HashMap
```

-----SB 分割线-----

List:

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下 >标）来访问 List 中的元素，这类似于 Java 的数组。

Vector:

基于数组（Array）的 List，其实就是封装了数组所不具备的一些功能方便我们使用，所以它难以避免数组的限制，同时性能也不可能超越数组。所以，在可能的情况下，我们要多运用数组。另外很重要的一点就是 Vector 是线程同步的(synchronized)的，这也是 Vector 和 ArrayList 的一个的重要区别。

ArrayList:

同 Vector 一样是一个基于数组上的链表，但是不同的是 ArrayList 不是同步的。所以在性能上要比 Vector 好一些，但是当运行到多线程环境中时，可需要自己在管理线程的同步问题。

LinkedList:

LinkedList 不同于前面两种 List，它不是基于数组的，所以不受数组性能的限制。

它每一个节点（Node）都包含两方面的内容：

1. 节点本身的数据（data）；

2.下一个节点的信息（nextNode）。

所以当对 LinkedList 做添加，删除动作的时候就不用像基于数组的 ArrayList 一样，必须进行大量的数据移动。只要更改 nextNode 的相关信息就可以实现了，这是 LinkedList 的优势。

List 总结：

- 所有的 List 中只能容纳单个不同类型的对象组成的表，而不是 Key—Value 键值对。
例如：[tom,1,c]
- 所有的 List 中可以有相同的元素，例如 Vector 中可以有 [tom,koo,too,koo]
- 所有的 List 中可以有 null 元素，例如[tom,null,1]
- 基于 Array 的 List (Vector, ArrayList) 适合查询，而 LinkedList 适合添加，删除操作

-----NB 分割线-----

Set:

Set 是一种不包含重复的元素的无序 Collection。

HashSet:

虽然 Set 同 List 都实现了 Collection 接口，但是他们的实现方式却大不一样。List 基本上都是以 Array 为基础。但是 Set 则是在 HashMap 的基础上来实现的，这个就是 Set 和 List 的根本区别。HashSet 的存储方式是把 HashMap 中的 Key 作为 Set 的对应存储项。看看 HashSet 的 add (Object obj) 方法的实现就可以一目了然了。

Java 代码

```
1. public boolean add(Object obj) {  
2.     return map.put(obj, PRESENT) == null;
```

```
3. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. public boolean add(Object obj) {  
2.     return map.put(obj, PRESENT) == null;  
3. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. public boolean add(Object obj) {  
2.     return map.put(obj, PRESENT) == null;  
3. }
```

这个也是为什么在 Set 中不能像在 List 中一样有重复的项的根本原因，因为 HashMap 的 key 是不能有重复的。

LinkedHashSet:

HashSet 的一个子类，一个链表。

TreeSet:

SortedSet 的子类，它不同于 HashSet 的根本就是 TreeSet 是有序的。它是通过 SortedMap 来实现的。

Set 总结:

- Set 实现的基础是 Map (HashMap)
- Set 中的元素是不能重复的，如果使用 add(Object obj)方法添加已经存在的对象，则会覆盖前面的对象

-----2B 分割线-----

Map:

Map 是一种把键对象和值对象进行关联的容器，而一个值对象又可以是一个 Map，依次类推，这样就可形成一个多级映射。对于键对象来说，像 Set 一样，一个 Map 容器中的键对象不允许重复，这是为了保持查找结果的一致性；如果有两个键对象一样，那你想得到那个键对象所对应的值对象时就有问题了，可能你得到的并不是你想的那个值对象，结果会造成混乱，所以键的唯一性很重要，也是符合集合的性质的。当然在使用过程中，某个键所对应的值对象可能会发生变化，这时会按照最后一次修改的值对象与键对应。对于值对象则没有唯一性的要求，你可以将任意多个键都映射到一个值对象上，这不会发生任何问题（不过对你的使用却可能会造成不便，你不知道你得到的到底是哪一个键所对应的值对象）。

Map 有两种比较常用的实现：HashMap 和 TreeMap。

HashMap 也用到了哈希码的算法，以便快速查找一个键，

TreeMap 则是对键按序存放，因此它便有一些扩展的方法，比如 firstKey(),lastKey() 等，你还可以从 TreeMap 中指定一个范围以取得其子 Map。

键和值的关联很简单，用 put(Object key, Object value) 方法即可将一个键与一个值对象相关联。用 get(Object key) 可得到与此 key 对象所对应的值对象。

-----JB 分割线-----

其它：

一、几个常用类的区别

1. ArrayList: 元素单个，效率高，多用于查询
2. Vector: 元素单个，线程安全，多用于查询
3. LinkedList: 元素单个，多用于插入和删除
4. HashMap: 元素成对，元素可为空
5. HashTable: 元素成对，线程安全，元素不可为空

二、Vector、ArrayList 和 LinkedList

大多数情况下，从性能上来说 ArrayList 最好，但是当集合内的元素需要频繁插入、删除时 LinkedList 会有比较好的表现，但是它们三个性能都比不上数组，另外 Vector 是线程同步的。所以：

如果能用数组的时候(元素类型固定，数组长度固定)，请尽量使用数组来代替 List；

如果没有频繁的删除插入操作，又不用考虑多线程问题，优先选择 ArrayList；

如果在多线程条件下使用，可以考虑 Vector；
如果需要频繁地删除插入， LinkedList 就有了用武之地；
如果你什么都不懂，用 ArrayList 没错。

三、Collections 和 Arrays

在 Java 集合类框架里有两个类叫做 Collections（注意，不是 Collection！）和 Arrays，这是 JCF 里面功能强大的工具，但初学者往往会忽视。按 JCF 文档的说法，这两个类提供了封装器实现（Wrapper Implementations）、数据结构算法和数组相关的应用。

想必大家不会忘记上面谈到的“折半查找”、“排序”等经典算法吧，Collections 类提供了丰富的静态方法帮助我们轻松完成这些在数据结构课上烦人的工作：

binarySearch: 折半查找。

sort: 排序，这里是一种类似于快速排序的方法，效率仍然是 $O(n * \log n)$ ，但却是一种稳定的排序方法。

reverse: 将线性表进行逆序操作，这个可是从前数据结构的经典考题哦！

rotate: 以某个元素为轴心将线性表“旋转”。

swap: 交换一个线性表中两个元素的位置。

.....

Collections 还有一个重要功能就是“封装器”（Wrapper），它提供了一些方法可以把一个集合转换成一个特殊的集合，如下：

unmodifiableXXX: 转换成只读集合，这里 XXX 代表六种基本集合接口：Collection、List、Map、Set、SortedMap 和 SortedSet。如果你对只读集合进行插入删除操作，将会抛出 UnsupportedOperationException 异常。

synchronizedXXX: 转换成同步集合。

singleton: 创建一个仅有一个元素的集合，这里 singleton 生成的是单元素 Set，
singletonList 和 singletonMap 分别生成单元素的 List 和 Map。

空集：由 Collections 的静态属性 EMPTY_SET、EMPTY_LIST 和 EMPTY_MAP 表示。

这次关于 Java 集合类概述就到这里，下一次我们来讲解 Java 集合类的具体应用，如 List 排序、删除重复元素。

java.util 之常用数据类型特性盘点

java.util 就相当于 c++ 的 STL，是 Java 的一个非常重要的包，有很多常用的数据类型，不同数据类型有不同的用途，而有些数据类似乎很相似，怎样选择应用，就需要对它们进行辨析。

下面列出了这些数据类型的特点，根据这些特点，就可以有针对性的选用

- * 蓝色为接口，绿色为具体实现类
- * 缩进的层次结构，就是 implement 或 extend 的层次关系
- * 每个接口或类都具备其所有上层接口、类的特性

Collection

```
.....|-----List  
.....|.....|-----ArrayList  
.....|.....|-----Vector  
.....|.....|.....|-----Stack  
.....|.....|-----LinkedList  
.....|-----Set  
.....|-----HashSet .  
.....|.....|-----LinkedHashSet  
.....|-----SortedSet  
.....|-----TreeSet
```

Iterator

```
....|-----ListIterator
```

Map

```
....|-----Hashtable  
....|.....|-----Properties  
....|-----HashMap  
....|.....|-----LinkedHashMap
```

.....|----WeakHashMap

.....|----SortedMap

.....|.....|----TreeMap

Collection .

- ..实现该接口及其子接口的所有类都可应用 clone()方法，并是序列化类.

.....List.

-●..可随机访问包含的元素
-●..元素是有序的
-●..可在任意位置增、删元素
-●..不管访问多少次，元素位置不变
-●..允许重复元素
-●..用 Iterator 实现单向遍历，也可用 ListIterator 实现双向遍历

.....ArrayList

-●..用数组作为根本的数据结构来实现 List
-●..元素顺序存储
-●..新增元素改变 List 大小时，内部会新建一个数组，在将添加元素前将所有数据拷贝到新数组中
-●..随机访问很快，删除非头尾元素慢，新增元素慢而且费资源
-●..较适用于无频繁增删的情况
-●..比数组效率低，如果不是需要可变数组，可考虑使用数组
-●..非线程安全

.....Vector .

-●..另一种 ArrayList，具备 ArrayList 的特性
-●..所有方法都是线程安全的（双刃剑，和 ArrayList 的主要区别）
-●..比 ArrayList 效率低

.....Stack

-●..LIFO 的数据结构

.....LinkedList.

-●..链接对象数据结构（类似链表）
-●..随机访问很慢，增删操作很快，不耗费多余资源
-●..非线程安全

.....**Set** .

-●..不允许重复元素，可以有一个空元素
-●..不可随机访问包含的元素
-●..只能用 Iterator 实现单向遍历

.....**HashSet**

-●..用 HashMap 作为根本数据结构来实现 Set
-●..元素是无序的
-●..迭代访问元素的顺序和加入的顺序不同
-●..多次迭代访问，元素的顺序可能不同
-●..非线程安全

.....**LinkedHashSet**

-●..基于 HashMap 和链表的 Set 实现
-●..迭代访问元素的顺序和加入的顺序相同
-●..多次迭代访问，元素的顺序不便
-●..因此可说这是一种有序的数据结构
-●..性能比 HashSet 差
-●..非线程安全

.....**SortedSet**

-●..加入 SortedSet 的所有元素必须实现 Comparable 接口
-●..元素是有序的

.....**TreeSet** .

-●..基于 TreeMap 实现的 SortedSet
 -●..排序后按升序排列元素
 -●..非线程安全
-

Iterator ..

- ..对 Set、List 进行单向遍历的迭代器

.....ListIterator.

-●..对 List 进行双向遍历的迭代器
-

Map

- ..键值对，键和值一一对应
- ..不允许重复的键.

....Hashtable.

-●..用作键的对象必须实现了 hashCode()、equals()方法，也就是说只有 Object 及其子类可用作键

-●..键、值都不能是空对象
-●..多次访问，映射元素的顺序相同
-●..线程安全的

.....Properties

-●..键和值都是字符串

....HashMap

-●..键和值都可以是空对象
-●..不保证映射的顺序
-●..多次访问，映射元素的顺序可能不同
-●..非线程安全

.....LinkedHashMap

-●..多次访问，映射元素的顺序是相同的
-●..性能比 HashMap 差

....WeakHashMap ..

-●..当某个键不再正常使用时，垃圾收集器会移除它，即便有映射关系存在
-●..非线程安全

.....SortedMap.

.....●..键按升序排列

.....●..所有键都必须实现Comparable接口.

.....TreeMap .

.....●..基于红黑树的SortedMap实现

.....●..非线程安全

1.13.4. 输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字

```
1. public class PrintMatrixClockwisely {  
2.  
3.     /**  
4.      * Q51.输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数  
字。  
5.      例如：如果输入如下矩阵：  
6.  
7.      1       2       3       4  
8.      5       6       7       8  
9.      9       10      11      12  
10.     13      14      15      16  
11.  
12.    则依次打印出数字 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10。  
13.    */  
14.    public enum Direction{  
15.        left,right,up,down,  
16.    }  
17.    public static void main(String[] args) {  
18.        int[][] matrix= {  
19.            { 1, 2, 3, 4, 5},  
20.            { 6, 7, 8, 9,10},  
21.            {11,12,13,14,15},  
22.            {16,17,18,19,20},  
23.            {21,22,23,24,25},
```

```
24.         };
25.         printMatrixClockwisely(matrix);
26.     }
27.
28.     /*
29.      * 思路比较直观,
30.      * 从左往右打印, 到了右边界则向下, 到了下边界就往左, 到了左边界就
31.      * 往上。
32.      */
33.     public static void printMatrixClockwisely(int[][] matrix){
34.
35.         int rowLen=matrix.length;
36.         int columnLen=matrix[0].length;
37.         Direction direction=Direction.right;
38.         int upBound=0;
39.         int downBound=rowLen-1;
40.         int leftBound=0;
41.         int rightBound=columnLen-1;
42.         int row=0;
43.         int column=0;
44.         while(true){
45.             System.out.print(matrix[row][column]+" ");
46.             //2 3 4 5 10 15 20 25 24 23 22 21 16 11 6 7 8 9 14 19 18 17 12 13
47.             if(upBound==downBound&&leftBound==rightBound){
48.                 break;
49.             }
50.             switch (direction){
51.                 case right:
52.                     if(column<rightBound){
53.                         ++column;
54.                     }else{
55.                         ++row;
56.                         direction=Direction.down;
57.                         ++upBound;
```

```
58.         }
59.         break;
60.     case down:
61.         if(row<downBound){
62.             ++row;
63.         }else{
64.             --column;
65.             direction=Direction.left;
66.             --rightBound;
67.         }
68.         break;
69.     case up:
70.         if(row>upBound){
71.             --row;
72.         }else{
73.             ++column;
74.             direction=Direction.right;
75.             ++leftBound;
76.         }
77.         break;
78.     case left:
79.         if(column>leftBound){
80.             --column;
81.         }else{
82.             --row;
83.             direction=Direction.up;
84.             --downBound;
85.         }
86.         break;
87.     default:break;
88. }
89. }
90.
91. }
92. }
```

1.13.5. 求集合的所有子集的算法

对于任意集合 A, 元素个数为 n (空集 n=0), 其所有子集的个数为 2^n 个

如集合 A={a,b,c}, 其子集个数为 8; 对于任意一个元素, 在每个子集中,

要么存在, 要么不存在, 对应关系是:

a->1 或 a->0

b->1 或 b->0

c->1 或 c->0

映射为子集:

(a,b,c)

(1,1,1)->(a,b,c)

(1,1,0)->(a,b)

(1,0,1)->(a, c)

(1,0,0)->(a)

(0,1,1)->(b,c)

(0,1,0)->(b)

(0,0,1)->(c)

(0,0,0)->@ (@表示空集)

算法 (1) :

观察以上规律, 与计算机中数据存储方式相似, 故可以通过一个整型数 (int) 与

集合映射 000...000 ~ 111...111 (0 表示有, 1 表示无, 反之亦可), 通过该整型数

逐次增 1 可遍历获取所有的数, 即获取集合的相应子集。

在这里提一下, 使用这种方式映射集合, 在进行集合运算时, 相当简便, 如

交运算对应按位与&, {a,b,c} 交 {a,b} 得 {a,b} <--> 111&110 == 110

并运算对应按位或|,

差运算对应&~。

算法（2）：

设函数 $f(n)=2^n$ ($n>=0$)，有如下递推关系 $f(n)=2*f(n-1)=2*(2*f(n-2))$

由此可知，求集合子集的算法可以用递归的方式实现，对于每个元素用一个映射列表 marks，标记其

在子集中的有无

很显然，在集合元素个数少的情况下，算法（1）优于算法（2），因为只需通过加法运算，便能映射

出子集，而算法（2）要递归调用函数，速度稍慢。但算法（1）有一个严重缺陷，集合的个数不能大于在

计算机中一个整型数的位数，一般计算机中整型数的为 32 位。对于算法（2）就没这样限制。

算法（1），取低位到高位码段作为映射列表

```
template<class T>

void print(T a[],int mark,int length)

{

bool allZero=true;

int limit=1<<length;

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

{

if(((1<<i)&mark)!=0) //mark 第 i+1 位为 1， 表示取该元素

{

allZero=false;

cout<<a[i]<<" ";

}

}

}
```

```

if(allZero==true)

{
    cout<<"@";

}

cout<<endl;

}

template<class T>

void subset(T a[],int length)

{
    if(length>31) return;

    int lowFlag=0;//对应 000...000
    int highFlag=(1<<length)-1;//对应 111...111

    for(int i=lowFlag;i<=highFlag;++i)

    {
        print(a,i,length);
    }
}

```

算法（2）

```

template<class T>

void print(T a[],bool marks[],int length)

{
    bool allFalse=true;

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

    {

```

```
if(marks[i]==true)

{
allfalse=false;

cout<<a[i]<<" ";

}

}

if(allFalse==true)

{
cout<<"@";

}

cout<<endl;

}

template<class T>

void subset(T a[],bool marks[],int m,int n,int length)

{

if(m>n)

{

print(a,marks,length);

}

else

{

marks[m]=true;

subset(a,marks,m+1,n,length);

marks[m]=false;

subset(a,marks,m+1,n,length);

}
```

```
}
```

```
}
```

1.13.6. 将一个数中的偶数位 bit 和奇数位 bit 交换

将一个数中的偶数位 bit 和奇数位 bit 交换

如：21(010101) 变为 42(101010)

方法：

用 0xaaaaaaaa 提取出偶数位,右移一位

用 0x55555555 提取出奇数位,左移一位

将上诉两个操作的结果进行位或操作

简单代码：

```
int SwapOddEvenBits(int n)
{
    return ((n&0xaaaaaaaa)>>1) | ((n&0x55555555)<<1);
```

测试用例：

```
int N = 21;
cout<<"Swap Odd and Even Bits :"<<SwapOddEvenBits(N)<<endl;
```

1.13.7. 二分查找实现

引言

Jon Bentley：90% 以上的程序员无法正确无误的写出二分查找代码。也许很多人都早已听说过这句话，但我还是想引用《编程珠玑》上的如下几段文字：

“二分查找可以解决（**预排序数组的查找**）问题：只要数组中包含 T（即要查找的值），那么通过不断缩小包含 T 的范围，最终就可以找到它。一开始，范围覆盖整个数组。将数组的中间项与 T 进行比较，可以排除一半元素，范围缩小一半。就这样反复比较，反复缩小范围，最终就会在数组中找到 T，或者确定原以为 T 所在的范围实际为空。对于包含 N 个元素的表，整个查找过程大约要经过 $\log_2 N$ 次比较。

多数程序员都觉得只要理解了上面的描述，写出代码就不难了；但事实并非如此。如果你不认同这一点，最好的办法就是放下书本，自己动手写一写。试试吧。

我在贝尔实验室和 IBM 的时候都出过这道考题。那些专业的程序员有几个小时的时间，可以用他们选择的语言把上面的描述写出来；写出高级伪代码也可以。考试结束后，差不多所

有程序员都认为自己写出了正确的程序。于是，我们花了半个钟头来看他们编写的代码经过测试用例验证的结果。几次课，一百多人的结果相差无几：90%的程序员写的程序中有 bug（我并不认为没有 bug 的代码就正确）。

我很惊讶：在足够的时间内，只有大约 10%的专业程序员可以把这个小程序写对。但写不对这个小程序的还不止这些人：高德纳在《计算机程序设计的艺术 第 3 卷 排序和查找》第 6.2.1 节的“历史与参考文献”部分指出，虽然早在 1946 年就有人将二分查找的方法公诸于世，但直到 1962 年才有人写出没有 bug 的二分查找程序。”——乔恩·本特利，《编程珠玑（第 1 版）》第 35-36 页。

你能正确无误的写出二分查找代码么？不妨一试。

二分查找代码

二分查找的原理想必不用多解释了，不过有一点必须提醒读者的是，二分查找是针对的排好序的数组。OK，纸上读来终觉浅，觉知此事要躬行。我先来写一份，下面是我写的一份二分查找的实现（之前去某一家公司面试也曾被叫当场实现二分查找，不过结果可能跟你一样，当时就未能完整无误写出），有任何问题或错误，恳请不吝指正：

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. //二分查找 V0.1 实现版
2. //copyright@2011 July
3. //随时欢迎读者找 bug, email: zhoulei0907@yahoo.cn。
4.
5. //首先要把握下面几个要点：
6. //right=n-1 => while(left <= right) => right=middle-1;
7. //right=n => while(left < right) => right=middle;
8. //middle 的计算不能写在 while 循环外，否则无法得到更新。
9.
10. int binary_search(int array[],int n,int value)
11. {
12.     int left=0;
13.     int right=n-1;
14.     //如果这里是 int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
15.     //1、下面循环的条件则是 while(left < right)
16.     //2、循环内当 array[middle]>value 的时候，right = mid
17.
18.     while (left<=right)      //循环条件，适时而变
19.     {
```

```
20.     int middle=left + ((right-left)>>1); //防止溢出，移位也更高效。同时，每次循环都需要更新。
21.
22.     if (array[middle]>value)
23.     {
24.         right =middle-1; //right 赋值，适时而变
25.     }
26.     else if(array[middle]<value)
27.     {
28.         left=middle+1;
29.     }
30.     else
31.     {
32.         return middle;
33.     }
34. }
35. return -1;
36. }
```

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. //二分查找 V0.1 实现版
2. //copyright@2011 July
3. //随时欢迎读者找 bug, email: zhoulei0907@yahoo.cn。
4.
5. //首先要把握下面几个要点：
6. //right=n-1 => while(left <= right) => right=middle-1;
7. //right=n => while(left < right) => right=middle;
8. //middle 的计算不能写在 while 循环外，否则无法得到更新。
9.
10. int binary_search(int array[],int n,int value)
11. {
12.     int left=0;
13.     int right=n-1;
14.     //如果这里是 int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
15.     //1、下面循环的条件则是 while(left < right)
16.     //2、循环内当 array[middle]>value 的时候，right = mid
```

```

17.
18.     while (left<=right)      //循环条件，适时而变
19.     {
20.         int middle=left + ((right-left)>>1); //防止溢出，移位也更高效。同时，每次循环都需要更新。
21.
22.         if (array[middle]>value)
23.         {
24.             right =middle-1; //right 赋值，适时而变
25.         }
26.         else if(array[middle]<value)
27.         {
28.             left=middle+1;
29.         }
30.     else
31.         return middle;
32.     //可能不会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的情况更多
33.     //如果每次循环都判断一下是否相等，将耗费时间
34. }
35.     return -1;
36. }
```

也许你之前已经把二分查找实现过很多次了，但现在不妨再次测试一下。关闭所有网页，窗口，打开记事本，或者编辑器，或者直接在本文评论下，不参考上面我写的或其他任何人的程序，给自己十分钟到 N 个小时不等的时间，立即编写一个二分查找程序。独立一次性正确写出来后，可以留下代码和邮箱地址，我给你传一份本 blog 的博文集锦 CHM 文件 && 十三个经典算法研究带标签+目录的 PDF 文档（你也可以去我的资源下载处下载：

http://download.csdn.net/user/v_july_v）。

当然，能正确写出来不代表任何什么，不能正确写出来亦不代表什么，仅仅针对 Jon Bentley 的言论做一个简单的测试而已。本博客算法交流群第 17 群：[Algorithms_17, 192036066](#)（12 月份内有效）。下一章，请见第二十六章：[基于给定的文档生成倒排索引的编码与实践](#)。谢谢。

总结

本文发表后，马上就有很多朋友自己尝试了。根据从朋友们在本文评论下留下的代码，发现出错率最高的在以下这么几个地方：

1. 注释里已经说得很明白了，可还是会有很多朋友犯此类的错误：

1. //首先要把握下面几个要点:
2. //right=n-1 => while(left <= right) => right=middle-1;
3. //right=n => while(left < right) => right=middle;
4. //middle 的计算不能写在 while 循环外, 否则无法得到更新。

2. 还有一个最常犯的错误是@土豆:

`middle = (left+right)>>1;` 这样的话 left 与 right 的值比较大的时候, 其和可能溢出。

各位继续努力。

1.13.8. 集合合并

给定一个字符串的集合, 格式如:

`{aaa bbb ccc}, {bbb ddd}, {eee fff}, {ggg}, {ddd hhh}`

要求将其中交集不为空的集合合并, 要求合并完成后的集合之间无交集, 例如上例应

输出

`{aaa bbb ccc ddd hhh}, {eee fff}, {ggg}`

- (1) 请描述你解决这个问题的思路;
- (2) 请给出主要的处理流程, 算法, 以及算法的复杂度
- (3) 请描述可能的改进(改进的方向如效果, 性能等等, 这是一个开放问题)。

我查了查网上的解法, 只看到一个(参见

<http://blog.csdn.net/lillllll/article/details/4162605>), 我有一点新想法, 说明如下:

1. 首先得到所有元素的集合(aaa, bbb, ccc, ddd, eee, fff, ggg, hhh), 复杂度 O(N), N 是所有元素的个数;
2. 为所有集合标记一个二进制数, 来表示包含哪些集合, 得到数组 int bina[5]复杂度 O(N^2);

如题, `{aaa, bbb, ccc} = 1 1 1 0 0 0 0,`

<code>{bbb, ddd}</code>	<code>= 0 1 0 1 0 0 0,</code>
<code>{eee, fff}</code>	<code>= 0 0 0 0 1 1 0,</code>
<code>{ggg}</code>	<code>= 0 0 0 0 0 1 0,</code>
<code>{ddd, hhh}</code>	<code>= 0 0 0 1 0 0 1,</code>

3. 然后写一个函数, 判断 int 数组中的第一个元素是否与其他元素&操作为空, 复杂度为 O(M), M 是原集合的个数

- a. 为空则输出该元素对应的集合, 并删除第一个元素, 递归调用本方法;
- b. 不为空则与那个元素做或操作合并, 删除那个元素, 递归调用本方法;

- c. 如果只剩下一个元素则输出其对应的字符串；
 - 4. 所有的结合已经得到了。
- 从上面的复杂度看，应该是比原来网上的方法好一些，欢迎大家踊跃拍砖！

1.13.9. 把求子集运算转换为组合问题

算法描述：把求子集运算转换为组合问题。

假设集合中包含 N 个元素， 子集合数 = $C(N, 0) + C(N, 1) + \dots + C(N, N-1) + C(N, N)$ ，对于任一个子集合，可以用一个 N 元组表示，即 $\langle S_1, S_2, \dots, S_{n-1}, S_n \rangle$ ，其中 S_i 取值范围为(0, 1)，0 表示不该子集合不包含该元素，1 表示该子集合包含该元素。因此，求子集合就转换成了罗列所示可能组合的算法。子集合数 = 2^n 。

```
void sub_sets(int i, int n, char *a, char *b)
{
    int j;

    if (i >= n)
    {
        for (j = 0; j < n; j++)
        {
            if (b[j] == '1')
                printf("%c", a[j]);
        }
        printf("\n");
    }
    else {
        b[i] = '1';
        sub_sets(i+1, n, a, b);
        b[i] = '0';
        sub_sets(i+1, n, a, b);
    }
}
```

1.13.10. 算法设计

一、算法设计

1、设 `rand (s, t)` 返回 $[s,t]$ 之间的随机小数，利用该函数在一个半径为 R 的圆内找随机 n 个点，并给出时间复杂度分析。

```
void GetNPointsInCircle(int R, int n)
{
    for (int i=0; i<n; i++)
    {
        float x = rand(-R, R);
        float y = rand(-sqrt(R*R - x*x), sqrt(R*R - x*x));
        // 也可以用极坐标表示
        printf("%f,%f\n", x,y);
    }
}
```

极坐标思路：

在半径为 R 的圆内找随机的 n 个点，既然是找点，那么就需要为其建立坐标系，如果建立平面直角坐标系，以圆心为原点，建立半径为 R 的圆的直角坐标系。要使随机的点在圆内，则必须使其所找的点 `point` 的 x , y 坐标的绝对值小于 R ，问题转化为：随机的找 n 个圆内的点，使其 x , y 坐标的绝对值均小于 R 。这里以 x 为例，首先 x 应满足 $-R \leq x \leq R$; y 同理。这样产生的点均在以圆心为中心，边长为 $2R$ 的正放形内，需要另外判断其距离 R 的值。本文介绍采用极坐标的形式，建立圆的极坐标系，则圆内的任意一点满足 $P(\rho, \theta)$ （用 ρ 表示线段 OP 的长度， θ 表示从 Ox 到 OP 的角度 `angle`），此时问题转化为：找一点，使其到圆心的距离 OP 大于等于 0 小于 R ，极角大于等于 0 小于 360 ，则 $OP = \text{rand}(0, R)$ ，当 $OP == R$ 时重新寻找， $angle = \text{rand}(0, 360)$ ，当 $angle == 360$ 时，重新寻找。每找到一个点 `p`，将其与之前所找到的所有点进行比较，若重合，则继续寻找，否则将其加入到已找到的点集合中，直到找到 n 个点。

[\[cpp\]](#) [view](#) [plain](#) [copy](#) [print](#)?

1. 存放点的数据结构
2. `typedef struct Point{`
3. `double r;`
4. `double angle;`
5. `}Point;`

```
6. 算法流程:  
7. for( i=n ; i>0; i--)  
8. 产生 p.r = rand(0,R), 且 p.r != R  
9. 产生 p.rangle = rand(0,360) ,且 p.rangle != 360  
10. 遍历所有产生的点, 若 p 已经存在, 则重新生成该点  
11. 否则将其加入到产生的点集合中
```

[cpp] [view plain](#) [copy](#) [print?](#)

```
1. 存放点的数据结构  
2. typedef struct Point{  
3.     double r;  
4.     double angle;  
5. }Point;  
6. 算法流程:  
7. for( i=n ; i>0; i--)  
8. 产生 p.r = rand(0,R), 且 p.r != R  
9. 产生 p.rangle = rand(0,360) ,且 p.rangle != 360  
10. 遍历所有产生的点, 若 p 已经存在, 则重新生成该点  
11. 否则将其加入到产生的点集合中
```

生成第 n 个点, 需要先遍历前 n-1 个点, 时间复杂度为 O(n), 故生成 n 个点的时间复杂度为 O(n^2)

2、为分析用户行为, 系统常需存储用户的一些 query, 但因 query 非常多, 故系统不能全存, 设系统每天只存 m 个 query, 现设计一个算法, 对用户请求的 query 进行随机选择 m 个, 请给一个方案, 使得每个 query 被抽中的概率相等, 并分析之, 注意: 不到最后一刻, 并不知用户的总请求量。

[蓄水池抽样问题](#)

[随机抽样问题表示如下:](#)

[要求从 N 个元素中随机的抽取 k 个元素, 其中 N 无法确定。](#)

这种应用的场景一般是数据流的情况下, 由于数据只能被读取一次, 而且数据量很大, 并不能全部保存, 因此数据量 N 是无法在抽样开始时确定的; 但又要保持随机性, 于是有了这个问题。所以搜索网站有时候会问这样的问题。

【解决】

解决方案就是蓄水库抽样（reservoir sampling）。主要思想就是保持一个集合（这个集合中的每个数字出现），作为蓄水池，依次遍历所有数据的时候以一定概率替换这个蓄水池中的数字。

//伪码

Init: a reservoir with the size:K

```
for i= k+1 to N
{
    M = random(1, i);
    if (M < k)
    {
        exchange(M, i);
    }
}
```

解释一下：程序的开始就是把前 k 个元素都放到水库中，然后对之后的第 i 个元素，以 k/i 的概率替换掉这个水库中的某一个元素。

【证明】

(1) 初始情况。出现在水库中的 k 个元素的出现概率都是一致的，都是 1。这个很显然。

(2) 第一步。第一步就是指，处理第 $k+1$ 个元素的情况。分两种情况：元素全部都没有被替换；其中某个元素与第 $k+1$ 个元素交换。

我们先看情况 2：第 $k+1$ 个元素被选中的概率是 $k/(k+1)$ （根据公式 k/i ），所以这个新元素在水库中出现的概率就一定是 $k/(k+1)$ （不管它替换掉哪个元素，反正肯定它是以这个概率出现在水库中）。下面来看水库中剩余的元素出现的概率，也就是 $1-P(\text{这个元素被替换掉})$ 。水库中任意一个元素被替换掉的概率是： $(k/k+1)*(1/k)=1/(k+1)$ ，意即首先要第 $k+1$ 个元素被选中，然后自己在集合的 k 个元素中被选中。那它出现的概率就是 $1-1/(k+1)=k/(k+1)$ 。可以看出来，旧元素和新元素出现的概率是相等的。

情况 1：当元素全部都没有替换掉的时候，每个元素的出现概率肯定是一样的，这很显然。但具体是多少呢？就是 $1-P(\text{第 } k+1 \text{ 个元素被选中})=1-k/(k+1)=1/(k+1)$ 。

(3) 归纳法：重复上面的过程，只要证明第 i 步到第 $i+1$ 步，所有元素出现的概率是相等的即可。

1. 1) C++中 vector 中 pushback 内部是如何实现的?

Push_back 函数在容器尾部创建一个新元素，并使容器长度加 1。新元素为括号中元素的副本。

3、C++ STL 中 vector 的相关问题：

(1)、调用 push_back 时，其内部的内存分配是如何进行的？

(2)、调用 clear 时，内部是如何具体实现的？若想将其内存释放，该如何操作？

Push_back 函数在容器尾部创建一个新元素，并使容器长度加 1。新元素为括号中元素的副本。

1.14.面试题集合（十三）

1.14.1. 各种排序算法

各种排序算法：冒泡路(入)兮(稀)快归堆，桶式排序，基数排序

冒泡排序，选择排序，插入排序，希尔排序，快速排序，归并排序，堆排序，桶式排序，基数排序

一、冒泡排序(BubbleSort)

1. 基本思想：

两两比较待排序数据元素的大小，发现两个数据元素的次序相反时即进行交换，直到没有反序的数据元素为止。

2. 排序过程：

设想被排序的数组 R [1..N] 垂直竖立，将每个数据元素看作有重量的气泡，根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R，凡扫描到违反本原则的轻气泡，就使其向上“漂浮”，如此反复进行，直至最后任何两个气泡都是轻者在上，重者在下为止。

【示例】：

49 13 13 13 13 13 13 13

38 49 27 27 27 27 27 27

65 38 49 38 38 38 38 38

97 65 38 49 49 49 49 49

76 97 65 49 49 49 49 49

13 76 97 65 65 65 65

27 27 76 97 76 76 76 76

49 49 49 76 97 97 97 97

java 代码实现:

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.   * 冒泡排序: 执行完一次内 for 循环后, 最小的一个数放到了数组的最前面(跟那一个排序算法* 不
3.   * 一样)。相邻位置之间交换
4.
5.  public class BubbleSort {
6.
7.  /**
8.   * 排序算法的实现, 对数组中指定的元素进行排序
9.   * @param array 待排序的数组
10.  * @param from 从哪里开始排序
11.  * @param end 排到哪里
12.  * @param c 比较器
13.  */
14. public void bubble(Integer[] array, int from, int end) {
15.     //需 array.length - 1 轮比较
16.     for (int k = 1; k < end - from + 1; k++) {
17.         //每轮循环中从最后一个元素开始向前起泡, 直到 i=k 止, 即 i 等于轮次止
18.         for (int i = end - from; i >= k; i--) {
19.             //按照一种规则 (后面元素不能小于前面元素) 排序
20.             if ((array[i].compareTo(array[i - 1])) < 0) {
21.                 //如果后面元素小了 (当然是大于还是小于要看比较器实现了) 前面的元素, 则前后
22.                 swap(array, i, i - 1);
23.             }
24.         }
25.     }
26. }
27.
28. /**
29. * 交换数组中的两个元素的位置
30. * @param array 待交换的数组
31. * @param i 第一个元素
```

```
32.     * @param j 第二个元素
33.     */
34.     public void swap(Integer[] array, int i, int j) {
35.         if (i != j) {//只有不是同一位置时才需交换
36.             Integer tmp = array[i];
37.             array[i] = array[j];
38.             array[j] = tmp;
39.         }
40.     }
41.
42.
43.     /**
44.      * 测试
45.      * @param args
46.      */
47.     public static void main(String[] args) {
48.         Integer[] intgArr = { 7, 2, 4, 3, 12, 1, 9, 6, 8, 5, 11, 10 };
49.         BubbleSort bubblesort = new BubbleSort();
50.         bubblesort.bubble(intgArr,0,intgArr.length-1);
51.         for(Integer intObj:intgArr){
52.             System.out.print(intObj + " ");
53.         }
54.     }
55. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 冒泡排序: 执行完一次内 for 循环后, 最小的一个数放到了数组的最前面(跟那一个排序算法* 不
3.  * 一样)。相邻位置之间交换
4.
5.  public class BubbleSort {
6.
7.      /**
8.       * 排序算法的实现, 对数组中指定的元素进行排序
```

```
9.      * @param array 待排序的数组
10.     * @param from 从哪里开始排序
11.     * @param end 排到哪里
12.     * @param c 比较器
13.     */
14.    public void bubble(Integer[] array, int from, int end) {
15.        //需 array.length - 1 轮比较
16.        for (int k = 1; k < end - from + 1; k++) {
17.            //每轮循环中从最后一个元素开始向前起泡，直到 i=k 止，即 i 等于轮次止
18.            for (int i = end - from; i >= k; i--) {
19.                //按照一种规则（后面元素不能小于前面元素）排序
20.                if ((array[i].compareTo(array[i - 1])) < 0) {
21.                    //如果后面元素小于了（当然是大于还是小于要看比较器实现了）前面的元素，则前后
22.                    //交换
23.                    swap(array, i, i - 1);
24.                }
25.            }
26.        }
27.
28.        /**
29.         * 交换数组中的两个元素的位置
30.         * @param array 待交换的数组
31.         * @param i 第一个元素
32.         * @param j 第二个元素
33.         */
34.        public void swap(Integer[] array, int i, int j) {
35.            if (i != j) {//只有不是同一位置时才需交换
36.                Integer tmp = array[i];
37.                array[i] = array[j];
38.                array[j] = tmp;
39.            }
40.        }
41.
42.
```

```
43.    /**
44.     * 测试
45.     * @param args
46.     */
47.    public static void main(String[] args) {
48.        Integer[] intgArr = { 7, 2, 4, 3, 12, 1, 9, 6, 8, 5, 11, 10 };
49.        BubbleSort bubblesort = new BubbleSort();
50.        bubblesort.bubble(intgArr,0,intgArr.length-1);
51.        for(Integer intObj:intgArr){
52.            System.out.print(intObj + " ");
53.        }
54.    }
55. }
```

另外一种实现方式：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1.    /**
2.     * 冒泡排序：执行完一次内 for 循环后，最大的一个数放到了数组的最后面。相邻位置之间交换
3.     */
4.    public class BubbleSort2{
5.        public static void main(String[] args){
6.            int[] a = { 3,5,9,4,7,8,6,1,2 };
7.            BubbleSort2 bubble = new BubbleSort2();
8.            bubble.bubble(a);
9.            for(int num:a){
10.                System.out.print(num + " ");
11.            }
12.        }
13.
14.        public void bubble(int[] a){
15.            for(int i=a.length-1;i>0;i--){
16.                for(int j=0;j<i;j++){
17.                    if(new Integer(a[j]).compareTo(new Integer(a[j+1]))>0){
18.                        swap(a,j,j+1);
19.                    }
20.                }
21.            }
22.        }
23.    }
24.
```

```
20.         }
21.     }
22.   }
23.
24.   public void swap(int[] a,int x,int y){
25.       int temp;
26.       temp=a[x];
27.       a[x]=a[y];
28.       a[y]=temp;
29.   }
30. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 冒泡排序：执行完一次内 for 循环后，最大的一个数放到了数组的最后面。相邻位置之间交换
3. */
4. public class BubbleSort2{
5.     public static void main(String[] args){
6.         int[] a = {3,5,9,4,7,8,6,1,2};
7.         BubbleSort2 bubble = new BubbleSort2();
8.         bubble.bubble(a);
9.         for(int num:a){
10.             System.out.print(num + " ");
11.         }
12.     }
13.
14.     public void bubble(int[] a){
15.         for(int i=a.length-1;i>0;i--){
16.             for(int j=0;j<i;j++){
17.                 if(new Integer(a[j]).compareTo(new Integer(a[j+1]))>0){
18.                     swap(a,j,j+1);
19.                 }
20.             }
21.         }
22.     }
}
```

```
23.  
24.     public void swap(int[] a,int x,int y){  
25.         int temp;  
26.         temp=a[x];  
27.         a[x]=a[y];  
28.         a[y]=temp;  
29.     }  
30. }
```

二、选择排序

1. 基本思想：

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。

2. 排序过程：

【示例】：

初始关键字 [49 38 65 97 76 13 27 49]

第一趟排序后 13 [38 65 97 76 49 27 49]

第二趟排序后 13 27 [65 97 76 49 38 49]

第三趟排序后 13 27 38 [97 76 49 65 49]

第四趟排序后 13 27 38 49 [49 97 65 76]

第五趟排序后 13 27 38 49 49 [97 97 76]

第六趟排序后 13 27 38 49 49 76 [76 97]

第七趟排序后 13 27 38 49 49 76 76 [97]

最后排序结果 13 27 38 49 49 76 76 97

java 代码实现：

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**  
2.  * 简单选择排序：执行完一次内 for 循环后最小的一个数放在了数组的最前面。  
3.  */  
4.  public class SelectSort {  
5.
```

```
6.    /**
7.     * 排序算法的实现，对数组中指定的元素进行排序
8.     * @param array 待排序的数组
9.     * @param from 从哪里开始排序
10.    * @param end 排到哪里
11.    * @param c 比较器
12.    */
13.   public void select(Integer[] array) {
14.       int minIndex;//最小索引
15.       /*
16.        * 循环整个数组（其实这里的上界为 array.length - 1 即可，因为当 i= array.length-1
17.        * 时，最后一个元素就已是最大的了，如果为 array.length 时，内层循环将不再循环），每轮
18.        * 假设
19.        * 第一个元素为最小元素，如果从第一元素后能选出比第一个元素更小元素，则让最小元
20.        * 素与第一
21.        * 个元素交换
22.        */
23.       for (int i=0; i<array.length; i++) {
24.           minIndex = i;//假设每轮第一个元素为最小元素
25.           //从假设的最小元素的下一元素开始循环
26.           for (int j=i+1;j<array.length; j++) {
27.               //如果发现有比当前 array[smallIndex]更小元素，则记下该元素的索引于 smallIndex 中
28.               if ((array[j].compareTo(array[minIndex])) < 0) {
29.                   minIndex = j;
30.               }
31.           }
32.           //先前只是记录最小元素索引，当最小元素索引确定后，再与每轮的第一个元素交换
33.           swap(array, i, minIndex);
34.       }
35.
36.       public static void swap(Integer[] int gArr,int x,int y){
37.           //Integer temp; //这个也行
38.           int temp;
```

```
39.     temp=intgArr[x];
40.     intgArr[x]=intgArr[y];
41.     intgArr[y]=temp;
42. }
43.
44. /**
45. * 测试
46. * @param args
47. */
48. public static void main(String[] args) {
49.     Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
50.     SelectSort insertSort = new SelectSort();
51.     insertSort.select(intgArr);
52.     for(Integer intObj:intgArr){
53.         System.out.print(intObj + " ");
54.     }
55.
56. }
57. }
```

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. * 简单选择排序：执行完一次内 for 循环后最小的一个数放在了数组的最前面。
3. */
4. public class SelectSort {
5.
6. /**
7. * 排序算法的实现，对数组中指定的元素进行排序
8. * @param array 待排序的数组
9. * @param from 从哪里开始排序
10. * @param end 排到哪里
11. * @param c 比较器
12. */
13. public void select(Integer[] array) {
14.     int minIndex;//最小索引
```

```
15.      /*
16.          * 循环整个数组（其实这里的上界为 array.length - 1 即可，因为当 i= array.length-1
17.          * 时，最后一个元素就已是最大的了，如果为 array.length 时，内层循环将不再循环），每轮
18.          * 假设
19.          * 第一个元素为最小元素，如果从第一元素后能选出比第一个元素更小元素，则让最小元
20.          * 素与第一
21.          * 个元素交换
22.          */
23.         for (int i=0; i<array.length; i++) {
24.             minIndex = i;//假设每轮第一个元素为最小元素
25.             //从假设的最小元素的下一元素开始循环
26.             for (int j=i+1;j<array.length; j++) {
27.                 //如果发现有比当前 array[smallIndex]更小元素，则记下该元素的索引于 smallIndex 中
28.                 if ((array [j]).compareTo(array [minIndex]) < 0) {
29.                     minIndex = j;
30.                 }
31.             }
32.             //先前只是记录最小元素索引，当最小元素索引确定后，再与每轮的第一个元素交换
33.             swap(array, i, minIndex);
34.         }
35.
36.         public static void swap(Integer[] intgArr,int x,int y){
37.             //Integer temp;//这个也行
38.             int temp;
39.             temp=intgArr[x];
40.             intgArr[x]=intgArr[y];
41.             intgArr[y]=temp;
42.         }
43.
44.         /**
45.          * 测试
46.          * @param args
47.          */
```

```
48.     public static void main(String[] args) {  
49.         Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };  
50.         SelectSort insertSort = new SelectSort();  
51.         insertSort.select(intgArr);  
52.         for(Integer intObj:intgArr){  
53.             System.out.print(intObj + " ");  
54.         }  
55.  
56.     }  
57. }
```

三、插入排序(Insertion Sort)

1. 基本思想:

每次将一个待排序的数据元素，插入到前面已经排好序的数列中的适当位置，使数列依然有序；直到待排序数据元素全部插入完为止。

2. 排序过程:

【示例】：

[初始关键字] [49] 38 65 97 76 13 27 49

J=2(38) [38 49] 65 97 76 13 27 49
J=3(65) [38 49 65] 97 76 13 27 49
J=4(97) [38 49 65 97] 76 13 27 49
J=5(76) [38 49 65 76 97] 13 27 49
J=6(13) [13 38 49 65 76 97] 27 49
J=7(27) [13 27 38 49 65 76 97] 49
J=8(49) [13 27 38 49 49 65 76 97]

java 代码实现：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1.     /**  
2.      * 直接插入排序:  
3.      * 注意所有排序都是从小到大排。  
4.      */  
5.
```

```
6.  public class InsertSort {  
7.  
8.      /**  
9.       * 排序算法的实现，对数组中指定的元素进行排序  
10.      * @param array 待排序的数组  
11.      * @param from 从哪里开始排序  
12.      * @param end 排到哪里  
13.      * @param c 比较器  
14.      */  
15.     public void insert(Integer[] array, int from, int end) {  
16.  
17.     /*  
18.      * 第一层循环：对待插入（排序）的元素进行循环  
19.      * 从待排序数组断的第二个元素开始循环，到最后一个元素（包括）止  
20.      */  
21.     for (int i=from+1; i<=end; i++) {  
22.         /*  
23.          * 第二层循环：对有序数组进行循环，且从有序数组最第一个元素开始向后循环  
24.          * 找到第一个大于待插入的元素  
25.          * 有序数组初始元素只有一个，且为源数组的第一个元素，一个元素数组总是有序的  
26.          */  
27.         for (int j = 0; j < i; j++) {  
28.             Integer insertedElem = array[i];//待插入到有序数组的元素  
29.             //从有序数组中最一个元素开始查找第一个大于待插入的元素  
30.             if ((array[j].compareTo(insertedElem)) > 0) {  
31.                 //找到插入点后，从插入点开始向后所有元素后移一位  
32.                 move(array, j, i - 1);  
33.                 //将待排序元素插入到有序数组中  
34.                 array[j] = insertedElem;  
35.                 break;  
36.             }  
37.         }  
38.     }  
39.  
40.     //=====以下是 java.util.Arrays 的插入排序算法的实现
```

```
41.      /*
42.      * 该算法看起来比较简洁一 j 点，有点像冒泡算法。
43.      * 将数组逻辑上分成前后两个集合，前面的集合是已经排序好序的元素，而后面集合为待排
44.      * 序的
45.      * 集合，每次内层循从后面集合中拿出一个元素，通过冒泡的形式，从前面集合最后一个元
46.      * 素开
47.      * 始往前比较，如果发现前面元素大于后面元素，则交换，否则循环退出
48.      *
49.      * 总感觉这种算术有点怪怪，既然是插入排序，应该是先找到插入点，而后再将待排序的元
50.      * 素插
51.      * 入到的插入点上，那么其他元素就必然向后移，感觉算法与排序名称不匹，但反过来与上
52.      * 面实
53.      * 现比，其实是一样的，只是上面先找插入点，待找到后一次性将大的元素向后移，而该算
54.      * 法却
55.      * 是走一步看一步，一步一步将待排序元素往前移
56.      */
57.      /*
58.      for (int i = from; i <= end; i++) {
59.          for (int j = i; j > from && c.compare(array[j - 1], array[j]) > 0; j--) {
60.              swap(array, j, j - 1);
61.          }
62.      }
63.      */
64.      /**
65.      * 数组元素后移
66.      * @param array 待移动的数组
67.      * @param startIndex 从哪个开始移
68.      * @param endIndex 到哪个元素止
69.      */
70.      public void move(Integer[] array, int startIndex, int endIndex) {
71.          for (int i = endIndex; i >= startIndex; i--) {
72.              array[i + 1] = array[i];
73.          }
74.      }
75.  }
```

```
71.      }
72.    }
73.
74.
75.    /**
76.     * 测试
77.     * @param args
78.    */
79.  public static void main(String[] args) {
80.    Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
81.    InsertSort insertSort = new InsertSort();
82.    insertSort.insert(intgArr,0,intgArr.length-1);
83.    for(Integer intObj:intgArr){
84.      System.out.print(intObj + " ");
85.    }
86.  }
87. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 直接插入排序:
3.  * 注意所有排序都是从小到大排。
4. */
5.
6. public class InsertSort {
7.
8. /**
9.  * 排序算法的实现，对数组中指定的元素进行排序
10. * @param array 待排序的数组
11. * @param from 从哪里开始排序
12. * @param end 排到哪里
13. * @param c 比较器
14. */
15. public void insert(Integer[] array, int from, int end) {
16.
```

```
17.    /*
18.     * 第一层循环：对待插入（排序）的元素进行循环
19.     * 从待排序数组断的第二个元素开始循环，到最后一个元素（包括）止
20.    */
21.    for (int i=from+1; i<=end; i++) {
22.        /*
23.         * 第二层循环：对有序数组进行循环，且从有序数组最第一个元素开始向后循环
24.         * 找到第一个大于待插入的元素
25.         * 有序数组初始元素只有一个，且为源数组的第一个元素，一个元素数组总是有序的
26.        */
27.        for (int j = 0; j < i; j++) {
28.            Integer insertedElem = array[i];//待插入到有序数组的元素
29.            //从有序数组中最一个元素开始查找第一个大于待插入的元素
30.            if ((array[j].compareTo(insertedElem)) > 0) {
31.                //找到插入点后，从插入点开始向后所有元素后移一位
32.                move(array, j, i - 1);
33.                //将待排序元素插入到有序数组中
34.                array[j] = insertedElem;
35.                break;
36.            }
37.        }
38.    }
39.
40.    //=====以下是 java.util.Arrays 的插入排序算法的实现
41.    /*
42.     * 该算法看起来比较简洁一 j 点，有点像冒泡算法。
43.     * 将数组逻辑上分成前后两个集合，前面的集合是已经排序好序的元素，而后面集合为待排
44.     * 序的
45.     * 集合，每次内层循从后面集合中拿出一个元素，通过冒泡的形式，从前面集合最后一个元
46.     * 素开
47.     * 始往前比较，如果发现前面元素大于后面元素，则交换，否则循环退出
48.     *
49.     * 总感觉这种算术有点怪怪，既然是插入排序，应该是先找到插入点，而后再将待排序的元
50.     * 素插
```

```
48.      * 入到的插入点上，那么其他元素就必然向后移，感觉算法与排序名称不匹，但反过来与上
面实
49.      * 现比，其实是一样的，只是上面先找插入点，待找到后一次性将大的元素向后移，而该算
法却
50.      * 是走一步看一步，一步一步将待排序元素往前移
51.      */
52.      /*
53.      for (int i = from; i <= end; i++) {
54.          for (int j = i; j > from && c.compare(array[j - 1], array[j]) > 0; j--) {
55.              swap(array, j, j - 1);
56.          }
57.      }
58.      */
59.  }
60.
61.
62.  /**
63.  * 数组元素后移
64. * @param array 待移动的数组
65. * @param startIndex 从哪个开始移
66. * @param endIndex 到哪个元素止
67. */
68. public void move(Integer[] array, int startIndex, int endIndex) {
69.     for (int i = endIndex; i >= startIndex; i--) {
70.         array[i+1] = array[i];
71.     }
72. }
73.
74.
75. /**
76. * 测试
77. * @param args
78. */
79. public static void main(String[] args) {
80.     Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
```

```
81.     InsertSort insertSort = new InsertSort();
82.     insertSort.insert(intgArr,0,intgArr.length-1);
83.     for(Integer intObj:int gArr){
84.         System.out.print(intObj + " ");
85.     }
86. }
87. }
```

四、希尔排序

java 代码实现：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. /**
2. * 插入排序----希尔排序：我们选择步长为：15,7,3,1
3. * 我们选择步长公式为：2^k-1,2^(k-1)-1,.....,15,7,3,1 (2^4-1,2^3-1,2^2-1,2^1-1)
4. * 注意所有排序都是从小到大排。
5. */
6. public class ShellSort {
7.
8. /**
9. * 排序算法的实现，对数组中指定的元素进行排序
10. * @param array 待排序的数组
11. * @param from 从哪里开始排序
12. * @param end 排到哪里
13. * @param c 比较器
14. */
15. public void sort(Integer[] array, int from, int end) {
16.     //初始步长，实质为每轮的分组数
17.     int step = initialStep(end - from + 1);
18.
19.     //第一层循环是对排序轮次进行循环。(step + 1) / 2 - 1 为下一轮步长值
20.     for (; step >= 1; step = (step + 1) / 2 - 1) {
21.         //对每轮里的每个分组进行循环
22.         for (int groupIndex = 0; groupIndex < step; groupIndex++) {
23.             }
```

```
24.         //对每组进行直接插入排序
25.         insertSort(array, groupIndex, step, end);
26.     }
27. }
28. }
29.
30. /**
31. * 直接插入排序实现
32. * @param array 待排序数组
33. * @param groupIndex 对每轮的那一组进行排序
34. * @param step 步长
35. * @param end 整个数组要排哪个元素止
36. * @param c 比较器
37. */
38. public void insertSort(Integer[] array, int groupIndex, int step, int end) {
39.     int startIndex = groupIndex;//从哪里开始排序
40.     int endIndex = startIndex;//排到哪里
41.     /*
42.      * 排到哪里需要计算得到，从开始排序元素开始，以 step 步长，可求得下元素是否在数组范
43.      * 围内，
44.      * 如果在数组范围内，则继续循环，直到索引超现数组范围
45.      */
46.     while ((endIndex + step) <= end) {
47.         endIndex += step;
48.     }
49.     // i 为每小组里的第二个元素开始
50.     for (int i = groupIndex + step; i <= end; i += step) {
51.         for (int j = groupIndex; j < i; j += step) {
52.             Integer insertedElem = array[i];
53.             //从有序数组中最一个元素开始查找第一个大于待插入的元素
54.             if ((array[j].compareTo(insertedElem)) >= 0) {
55.                 //找到插入点后，从插入点开始向后所有元素后移一位
56.                 move(array, j, i - step, step);
57.                 array[j] = insertedElem;
```

```
58.         break;
59.     }
60. }
61. }
62. }
63.
64. /**
65. * 根据数组长度求初始步长
66. *
67. * 我们选择步长的公式为:  $2^{k-1}, 2^{(k-1)-1}, \dots, 15, 7, 3, 1$ , 其中  $2^k$  减一即为该步长序列, k
68. * 为排序轮次
69. *
70. * 初始步长: step =  $2^{k-1}$ 
71. * 初始步长约束条件: step < len - 1 初始步长的值要小于数组长度还要减一的值 (因
72. * 为第一轮分组时尽量不要分为一组, 除非数组本身的长度就小于等于 4)
73. *
74. * 由上面两个关系式可以得知:  $2^{k-1} < len - 1$  关系式, 其中 k 为轮次, 如果把  $2^k$  表达式
75. * 转换成 step 表达式, 则  $2^{k-1}$  可使用 (step + 1)*2-1 替换 (因为 step+1 相当于第 k-1
76. * 轮的步长, 所以在 step+1 基础上乘以 2 就相当于  $2^k$  了), 即步长与数组长度的关系不等式
    为
77. * (step + 1)*2 - 1 < len - 1
78. *
79. * @param len 数组长度
80. * @return
81. */
82. public static int initialStep(int len) {
83. /*
84. * 初始值设置为步长公式中的最小步长, 从最小步长推导出最长初始步长值, 即按照以下公
    式来推:
85. * 1, 3, 7, 15, ...,  $2^{(k-1)-1}, 2^{k-1}$ 
86. * 如果数组长度小于等于 4 时, 步长为 1, 即长度小于等于 4 的数组不用分组, 此时直接退
    化为直接插入排序
87. */
88.     int step = 1;
89.
```

```
90.     //试探下一个步长是否满足条件，如果满足条件，则步长置为下一步长
91.     while ((step + 1) * 2 - 1 < len - 1) {
92.         step = (step + 1) * 2 - 1;
93.     }
94.
95.     System.out.println("初始步长：" + step);
96.     return step;
97. }
98.
99. /**
100. * 以指定的步长将数组元素后移，步长指定每个元素间的间隔
101. * @param array 待排序数组
102. * @param startIndex 从哪里开始移
103. * @param endIndex 到哪个元素止
104. * @param step 步长
105. */
106. protected final void move(Integer[] array, int startIndex, int endIndex, int step) {
107.     for (int i = endIndex; i >= startIndex; i -= step) {
108.         array[i + step] = array[i];
109.     }
110. }
111.
112.
113. /**
114. * 测试
115. * @param args
116. */
117. public static void main(String[] args) {
118.     Integer[] intgArr = { 5, 9, 1, 4, 8, 2, 6, 3, 7, 0 };
119.     ShellSort shellSort = new ShellSort();
120.     shellSort.sort(intgArr, 0, intgArr.length - 1);
121.     for (Integer intObj : intgArr) {
122.         System.out.print(intObj + " ");
123.     }
124. }
```

```
125. }
```

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1.  /**
2.   * 插入排序----希尔排序：我们选择步长为：15,7,3,1
3.   * 我们选择步长公式为： $2^{k-1}, 2^{(k-1)-1}, \dots, 15, 7, 3, 1$  ( $2^4-1, 2^3-1, 2^2-1, 2^1-1$ )
4.   * 注意所有排序都是从小到大排。
5.   */
6. public class ShellSort {
7.
8.     /**
9.      * 排序算法的实现，对数组中指定的元素进行排序
10.     * @param array 待排序的数组
11.     * @param from 从哪里开始排序
12.     * @param end 排到哪里
13.     * @param c 比较器
14.     */
15.    public void sort(Integer[] array, int from, int end) {
16.        //初始步长，实质为每轮的分组数
17.        int step = initialStep(end - from + 1);
18.
19.        //第一层循环是对排序轮次进行循环。 $(step + 1) / 2 - 1$  为下一轮步长值
20.        for (; step >= 1; step = (step + 1) / 2 - 1) {
21.            //对每轮里的每个分组进行循环
22.            for (int groupIndex = 0; groupIndex < step; groupIndex++) {
23.
24.                //对每组进行直接插入排序
25.                insertSort(array, groupIndex, step, end);
26.            }
27.        }
28.    }
29.
30.    /**
31.     * 直接插入排序实现
32.     * @param array 待排序数组
33.     */
34. }
```

```
33.     * @param groupIndex 对每轮的哪一组进行排序
34.     * @param step 步长
35.     * @param end 整个数组要排哪个元素止
36.     * @param c 比较器
37.     */
38.    public void insertSort(Integer[] array, int groupIndex, int step, int end) {
39.        int startIndex = groupIndex;//从哪里开始排序
40.        int endIndex = startIndex;//排到哪里
41.        /*
42.         * 排到哪里需要计算得到，从开始排序元素开始，以 step 步长，可求得下元素是否在数组范
43.         围内，
44.         * 如果在数组范围内，则继续循环，直到索引超现数组范围
45.         */
46.        while ((endIndex + step) <= end) {
47.            endIndex += step;
48.
49.            // i 为每小组里的第二个元素开始
50.            for (int i = groupIndex + step; i <= end; i += step) {
51.                for (int j = groupIndex; j < i; j += step) {
52.                    Integer insertedElem = array[i];
53.                    //从有序数组中最一个元素开始查找第一个大于待插入的元素
54.                    if ((array[j].compareTo(insertedElem)) >= 0) {
55.                        //找到插入点后，从插入点开始向后所有元素后移一位
56.                        move(array, j, i - step, step);
57.                        array[j] = insertedElem;
58.                        break;
59.                    }
60.                }
61.            }
62.        }
63.
64.        /**
65.         * 根据数组长度求初始步长
66.         *
```

```

67.    * 我们选择步长的公式为:  $2^{k-1}, 2^{(k-1)-1}, \dots, 15, 7, 3, 1$  , 其中  $2^k$  减一即为该步长序列, k
68.    * 为排序轮次
69.    *
70.    * 初始步长: step =  $2^{k-1}$ 
71.    * 初始步长约束条件: step < len - 1 初始步长的值要小于数组长度还要减一的值 (因
72.    * 为第一轮分组时尽量不要分为一组, 除非数组本身的长度就小于等于 4)
73.    *
74.    * 由上面两个关系式可以得知:  $2^k - 1 < len - 1$  关系式, 其中 k 为轮次, 如果把  $2^k$  表达式
75.    * 转换成 step 表达式, 则  $2^{k-1}$  可使用  $(step + 1) * 2 - 1$  替换 (因为 step+1 相当于第 k-1
76.    * 轮的步长, 所以在 step+1 基础上乘以 2 就相当于  $2^k$  了), 即步长与数组长度的关系不等式
    为
77.    *  $(step + 1) * 2 - 1 < len - 1$ 
78.    *
79.    * @param len 数组长度
80.    * @return
81.    */
82.    public static int initialStep(int len) {
83.        /*
84.         * 初始值设置为步长公式中的最小步长, 从最小步长推导出最长初始步长值, 即按照以下公
            式来推:
85.         * 1,3,7,15,..., $2^{(k-1)-1}, 2^{k-1}$ 
86.         * 如果数组长度小于等于 4 时, 步长为 1, 即长度小于等于 4 的数组不用分组, 此时直接退
            化为直接插入排序
87.        */
88.        int step = 1;
89.
90.        //试探下一个步长是否满足条件, 如果满足条件, 则步长置为下一步长
91.        while ((step + 1) * 2 - 1 < len - 1) {
92.            step = (step + 1) * 2 - 1;
93.        }
94.
95.        System.out.println("初始步长 :" + step);
96.        return step;
97.    }
98.

```

```

99.  /**
100. * 以指定的步长将数组元素后移，步长指定每个元素间的间隔
101. * @param array 待排序数组
102. * @param startIndex 从哪里开始移
103. * @param endIndex 到哪个元素止
104. * @param step 步长
105. */
106. protected final void move(Integer[] array, int startIndex, int endIndex, int step) {
107.     for (int i = endIndex; i >= startIndex; i -= step) {
108.         array[i + step] = array[i];
109.     }
110. }
111.
112.
113. /**
114. * 测试
115. * @param args
116. */
117. public static void main(String[] args) {
118.     Integer[] intgArr = { 5, 9, 1, 4, 8, 2, 6, 3, 7, 0 };
119.     ShellSort shellSort = new ShellSort();
120.     shellSort.sort(intgArr,0,intgArr.length-1);
121.     for(Integer intObj:intgArr){
122.         System.out.print(intObj + " ");
123.     }
124. }
125. }

```

五、快速排序 (Quick Sort)

1. 基本思想:

在当前无序区 $R[1..H]$ 中任取一个数据元素作为比较的"基准"(不妨记为 X)，用此基准将当前无序区划分为左右两个较小的无序区： $R[1..I-1]$ 和 $R[I+1..H]$ ，且左边的无序子区中数据元素均小于等于基准元素，右边的无序子区中数据元素均大于等于基准元素，而基准 X 则

位于最终排序的位置上，即 $R[1..I-1] \leq X.Key \leq R[I+1..H]$ ($1 \leq I \leq H$)，当 $R[1..I-1]$ 和 $R[I+1..H]$ 均非空时，分别对它们进行上述的划分过程，直至所有无序子区中的数据元素均已排序为止。

2. 排序过程：

【示例】：

初始关键字 [49 38 65 97 76 13 27 49]

一趟排序之后 [27 38 13] 49 [76 97 65 49]

二趟排序之后 [13] 27 [38] 49 [49 65] 76 [97]

三趟排序之后 13 27 38 49 49 [65] 76 97

最后的排序结果 13 27 38 49 49 65 76 97

各趟排序之后的状态

java 代码实现：

[java] [view plain](#) [copy](#) [print](#)?

```
1. /**
2. * 快速排序:
3. */
4.
5. public class QuickSort {
6.
7. /**
8. * 排序算法的实现, 对数组中指定的元素进行排序
9. * @param array 待排序的数组
10. * @param from 从哪里开始排序
11. * @param end 排到哪里
12. * @param c 比较器
13. */
14. //定义了一个这样的公有方法 sort, 在这个方法体里面执行私有方法 quickSort (这种方式值得借鉴)。
15. public void sort(Integer[] array, int from, int end) {
16.     quickSort(array, from, end);
17. }
18.
19. /**
20. * 递归快速排序实现
21. * @param array 待排序数组
```

```
22.     * @param low 低指针
23.     * @param high 高指针
24.     * @param c 比较器
25.     */
26.    private void quickSort(Integer[] array, int low, int high) {
27.        /*
28.         * 如果分区中的低指针小于高指针时循环; 如果 low=hight 说明数组只有一个元素, 无需再处
29.         理;
30.         * 如果 low > hight, 则说明上次枢纽元素的位置 pivot 就是 low 或者是 hight, 此种情况
31.         * 下分区不存, 也不需处理
32.         */
33.        if (low < high) {
34.            //对分区进行排序整理
35.
36.            //int pivot = partition1(array, low, high);
37.            int pivot = partition2(array, low, high);
38.            //int pivot = partition3(array, low, high);
39.
40.            /*
41.             * 以 pivot 为边界, 把数组分成三部分[low, pivot - 1]、[pivot]、[pivot + 1, high]
42.             * 其中[pivot]为枢纽元素, 不需处理, 再对[low, pivot - 1]与[pivot + 1, high]
43.             * 各自进行分区排序整理与进一步分区
44.             */
45.            quickSort(array, low, pivot - 1);
46.            quickSort(array, pivot + 1, high);
47.
48.        }
49.
50.        /**
51.         * 实现—
52.         *
53.         * @param array 待排序数组
54.         * @param low 低指针
55.         * @param high 高指针
```

```
56.     * @param c 比较器
57.     * @return int 调整后中枢位置
58.     */
59.     private int partition1(Integer[] array, int low, int high) {
60.         Integer pivotElem = array[low];//以第一个元素为中枢元素
61.         //从前向后依次指向比中枢元素小的元素，刚开始时指向中枢，也是小于与大小中枢的元素的
62.         //分界点
63.
64.         /*
65.             * 在中枢元素后面的元素中查找小于中枢元素的所有元素，并依次从第二个位置从前往后存
66.             放
67.             * 注，这里最好使用 i 来移动，如果直接移动 low 的话，最后不知道数组的边界了，但后面
68.             需要
69.             * 知道数组的边界
70.             */
71.             for (int i = low + 1; i <= high; i++) {
72.                 //如果找到一个比中枢元素小的元素
73.                 if ((array[i].compareTo(pivotElem)) < 0) {
74.                     swap(array, ++border, i);//border 前移，表示有小于中枢元素的元素
75.                 }
76.             }
77.             /*
78.                 * 如果 border 没有移动时说明说明后面的元素都比中枢元素要大，border 与 low 相等，此时
79.                 是
80.                 * 同一位置交换，是否交换都没关系；当 border 移到了 high 时说明所有元素都小于中枢元素，
81.                 此
82.                 * 时将中枢元素与最后一个元素交换即可，即 low 与 high 进行交换，大的中枢元素移到了序
83.                 列最
84.                 * 后；如果 low < minIndex < high，表明中枢后面的元素前部分小于中枢元素，而后部分大于
85.                 中枢元素，此时中枢元素与前部分数组中最后一个小于它的元素交换位置，使得中枢元素
86.                 放置在
87.                 * 正确的位置
88.                 */
89.             swap(array, border, low);
```

```
84.     return border;
85. }
86.
87. /**
88. * 实现二
89. *
90. * @param array 待排序数组
91. * @param low 待排序区低指针
92. * @param high 待排序区高指针
93. * @param c 比较器
94. * @return int 调整后中枢位置
95. */
96. private int partition2(Integer[] array, int low, int high) {
97.     int pivot = low; //中枢元素位置，我们以第一个元素为中枢元素
98.     //退出条件这里只可能是 low = high
99.     while (true) {
100.         if (pivot != high) { //如果中枢元素在低指针位置时，我们移动高指针
101.             //如果高指针元素小于中枢元素时，则与中枢元素交换
102.             if ((array[high].compareTo(array[pivot])) < 0) {
103.                 swap(array, high, pivot);
104.                 //交换后中枢元素在高指针位置了
105.                 pivot = high;
106.             } else { //如果未找到小于中枢元素，则高指针前移继续找
107.                 high--;
108.             }
109.         } else { //否则中枢元素在高指针位置
110.             //如果低指针元素大于中枢元素时，则与中枢元素交换
111.             if ((array[low].compareTo(array[pivot])) > 0) {
112.                 swap(array, low, pivot);
113.                 //交换后中枢元素在低指针位置了
114.                 pivot = low;
115.             } else { //如果未找到大于中枢元素，则低指针后移继续找
116.                 low++;
117.             }
118.         }
}
```

```
119.     if (low == high) {
120.         break;
121.     }
122. }
123. //返回中枢元素所在位置，以便下次分区
124. return pivot;
125. }
126.
127. /**
128. * 实现三
129. *
130. * @param array 待排序数组
131. * @param low 待排序区低指针
132. * @param high 待排序区高指针
133. * @param c 比较器
134. * @return int 调整后中枢位置
135. */
136. private int partition3(Integer[] array, int low, int high) {
137.     int pivot = low; //中枢元素位置，我们以第一个元素为中枢元素
138.     low++;
139.     //----调整高低指针所指向的元素顺序，把小于中枢元素的移到前部分，大于中枢元素的移到
140.     //后面部分
141.     //退出条件这里只可能是 low = high
142.     while (true) {
143.         //如果高指针未超出低指针
144.         while (low < high) {
145.             //如果低指针指向的元素大于或等于中枢元素时表示找到了，退出，注：等于时也要后
146.             //移
147.             if ((array[low].compareTo(array[pivot])) >= 0) {
148.                 break;
149.             } else { //如果低指针指向的元素小于中枢元素时继续找
150.                 low++;
151.             }
152.         }
153.     }
154. }
```

```
152.  
153.     while (high > low) {  
154.         //如果高指针指向的元素小于中枢元素时表示找到，退出  
155.         if ((array[high].compareTo(array[pivot])) < 0) {  
156.             break;  
157.         } else { //如果高指针指向的元素大于中枢元素时继续找  
158.             high--;  
159.         }  
160.     }  
161.     //退出上面循环时 low = high  
162.     if (low == high) {  
163.         break;  
164.     }  
165.  
166.     swap(array, low, high);  
167. }  
168.  
169. //----高低指针所指向的元素排序完成后，还得要把中枢元素放到适当的位置  
170. if ((array[pivot].compareTo(array[low])) > 0) {  
171.     //如果退出循环时中枢元素大于了低指针或高指针元素时，中枢元素需与 low 元素交换  
172.     swap(array, low, pivot);  
173.     pivot = low;  
174. } else if ((array[pivot].compareTo(array[low])) <= 0) {  
175.     swap(array, low - 1, pivot);  
176.     pivot = low - 1;  
177. }  
178.  
179. //返回中枢元素所在位置，以便下次分区  
180. return pivot;  
181. }  
182.  
183. /**  
184. * 交換数组中的两个元素的位置  
185. * @param array 待交換的数组  
186. * @param i 第一个元素
```

```
187.     * @param j 第二个元素
188.     */
189.     public void swap(Integer[] array, int i, int j) {
190.         if (i != j) {//只有不是同一位置时才需交换
191.             Integer tmp = array[i];
192.             array[i] = array[j];
193.             array[j] = tmp;
194.         }
195.     }
196.
197.     /**
198.      * 测试
199.      * @param args
200.     */
201.     public static void main(String[] args) {
202.         Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
203.         QuickSort quicksort = new QuickSort();
204.         quicksort.sort(intgArr,0,intgArr.length-1);
205.         for(Integer intObj:intgArr){
206.             System.out.print(intObj + " ");
207.         }
208.     }
209. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1.     /**
2.      * 快速排序:
3.      */
4.
5.     public class QuickSort {
6.
7.         /**
8.          * 排序算法的实现，对数组中指定的元素进行排序
9.          * @param array 待排序的数组
10.         * @param from 从哪里开始排序
```

```
11.     * @param end 排到哪里
12.     * @param c 比较器
13.     */
14. //定义了一个这样的公有方法 sort，在这个方法体里面执行私有方法 quickSort（这种方式值得借鉴）。
15. public void sort(Integer[] array, int from, int end) {
16.     quickSort(array, from, end);
17. }
18.
19. /**
20. * 递归快速排序实现
21. * @param array 待排序数组
22. * @param low 低指针
23. * @param high 高指针
24. * @param c 比较器
25. */
26. private void quickSort(Integer[] array, int low, int high) {
27.     /*
28.      * 如果分区中的低指针小于高指针时循环；如果 low=hight 说明数组只有一个元素，无需再处理；
29.      * 如果 low > high，则说明上次枢纽元素的位置 pivot 就是 low 或者是 high，此种情况
30.      * 下分区不存，也不需处理
31.      */
32.     if (low < high) {
33.         //对分区进行排序整理
34.
35.         //int pivot = partition1(array, low, high);
36.         int pivot = partition2(array, low, high);
37.         //int pivot = partition3(array, low, high);
38.
39.         /*
40.          * 以 pivot 为边界，把数组分成三部分[low, pivot - 1]、[pivot]、[pivot + 1, high]
41.          * 其中[pivot]为枢纽元素，不需处理，再对[low, pivot - 1]与[pivot + 1, high]
42.          * 各自进行分区排序整理与进一步分区
43.          */

```

```
44.         quickSort(array, low, pivot - 1);
45.         quickSort(array, pivot + 1, high);
46.     }
47.
48. }
49.
50. /**
51. * 实现一
52. *
53. * @param array 待排序数组
54. * @param low 低指针
55. * @param high 高指针
56. * @param c 比较器
57. * @return int 调整后中枢位置
58. */
59. private int partition1(Integer[] array, int low, int high) {
60.     Integer pivotElem = array[low];//以第一个元素为中枢元素
61.     //从前向后依次指向比中枢元素小的元素，刚开始时指向中枢，也是小于与大小中枢的元素的
62.     //分界点
63.     int border = low;
64.     /*
65.      * 在中枢元素后面的元素中查找小于中枢元素的所有元素，并依次从第二个位置从前往后存
66.      放
67.      * 注，这里最好使用 i 来移动，如果直接移动 low 的话，最后不知道数组的边界了，但后面
68.      需要
69.      * 知道数组的边界
70.      */
71.      for (int i = low + 1; i <= high; i++) {
72.          //如果找到一个比中枢元素小的元素
73.          if ((array[i].compareTo(pivotElem)) < 0) {
74.              swap(array, ++border, i);//border 前移，表示有小于中枢元素的元素
75.          }
76.      }
77. }
```

```
76.      * 如果 border 没有移动时说明说明后面的元素都比中枢元素要大，border 与 low 相等，此时
    是
77.      * 同一位置交换，是否交换都没关系；当 border 移到了 high 时说明所有元素都小于中枢元素，
    此
78.      * 时将中枢元素与最后一个元素交换即可，即 low 与 high 进行交换，大的中枢元素移到了序
    列最
79.      * 后；如果 low < minIndex < high，表 明中枢后面的元素前部分小于中枢元素，而后部分大于
80.      * 中枢元素，此时中枢元素与前部分数组中最后一个小于它的元素交换位置，使得中枢元素
    放置在
81.      * 正确的位置
82.      */
83.      swap(array, border, low);
84.      return border;
85.  }
86.
87. /**
88. * 实现二
89. *
90. * @param array 待排序数组
91. * @param low 待排序区低指针
92. * @param high 待排序区高指针
93. * @param c 比较器
94. * @return int 调整后中枢位置
95. */
96. private int partition2(Integer[] array, int low, int high) {
97.     int pivot = low;//中枢元素位置，我们以第一个元素为中枢元素
98.     //退出条件这里只可能是 low = high
99.     while (true) {
100.         if (pivot != high) {//如果中枢元素在低指针位置时，我们移动高指针
101.             //如果高指针元素小于中枢元素时，则与中枢元素交换
102.             if ((array[high].compareTo(array[pivot])) < 0) {
103.                 swap(array, high, pivot);
104.                 //交换后中枢元素在高指针位置了
105.                 pivot = high;
106.             } else {//如果未找到小于中枢元素，则高指针前移继续找

```

```
107.         high--;
108.     }
109. } else { //否则中枢元素在高指针位置
110.     //如果低指针元素大于中枢元素时，则与中枢元素交换
111.     if ((array[low].compareTo(array[pivot])) > 0) {
112.         swap(array, low, pivot);
113.         //交换后中枢元素在低指针位置了
114.         pivot = low;
115.     } else { //如果未找到大于中枢元素，则低指针后移继续找
116.         low++;
117.     }
118. }
119. if (low == high) {
120.     break;
121. }
122. }
123. //返回中枢元素所在位置，以便下次分区
124. return pivot;
125. }
126.
127. /**
128. * 实现三
129. *
130. * @param array 待排序数组
131. * @param low 待排序区低指针
132. * @param high 待排序区高指针
133. * @param c 比较器
134. * @return int 调整后中枢位置
135. */
136. private int partition3(Integer[] array, int low, int high) {
137.     int pivot = low;//中枢元素位置，我们以第一个元素为中枢元素
138.     low++;
139.     //----调整高低指针所指向的元素顺序，把小于中枢元素的移到前部分，大于中枢元素的移到
140.     //后面部分
141.     //退出条件这里只可能是 low = high
```

```
141.  
142.     while (true) {  
143.         //如果高指针未超出低指针  
144.         while (low < high) {  
145.             //如果低指针指向的元素大于或等于中枢元素时表示找到了，退出，注：等于时也要后  
移  
146.             if ((array[low].compareTo(array[pivot])) >= 0) {  
147.                 break;  
148.             } else { //如果低指针指向的元素小于中枢元素时继续找  
149.                 low++;  
150.             }  
151.         }  
152.  
153.         while (high > low) {  
154.             //如果高指针指向的元素小于中枢元素时表示找到，退出  
155.             if ((array[high].compareTo(array[pivot])) < 0) {  
156.                 break;  
157.             } else { //如果高指针指向的元素大于中枢元素时继续找  
158.                 high--;  
159.             }  
160.         }  
161.         //退出上面循环时 low = high  
162.         if (low == high) {  
163.             break;  
164.         }  
165.  
166.         swap(array, low, high);  
167.     }  
168.  
169.     //----高低指针所指向的元素排序完成后，还得要把中枢元素放到适当的位置  
170.     if ((array[pivot].compareTo(array[low])) > 0) {  
171.         //如果退出循环时中枢元素大于了低指针或高指针元素时，中枢元素需与 low 元素交换  
172.         swap(array, low, pivot);  
173.         pivot = low;  
174.     } else if ((array[pivot].compareTo(array[low])) <= 0) {
```

```
175.         swap(array, low - 1, pivot);
176.         pivot = low - 1;
177.     }
178.
179.     //返回中枢元素所在位置，以便下次分区
180.     return pivot;
181. }
182.
183. /**
184. * 交换数组中的两个元素的位置
185. * @param array 待交换的数组
186. * @param i 第一个元素
187. * @param j 第二个元素
188. */
189. public void swap(Integer[] array, int i, int j) {
190.     if (i != j) { //只有不是同一位置时才需交换
191.         Integer tmp = array[i];
192.         array[i] = array[j];
193.         array[j] = tmp;
194.     }
195. }
196.
197. /**
198. * 测试
199. * @param args
200. */
201. public static void main(String[] args) {
202.     Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
203.     QuickSort quicksort = new QuickSort();
204.     quicksort.sort(intgArr, 0, intgArr.length - 1);
205.     for (Integer intObj : intgArr) {
206.         System.out.print(intObj + " ");
207.     }
208. }
209. }
```

六、归并排序

java 代码实现：

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.   归并排序：里面是一个递归程序，深刻理解之。
3.  */
4. public class MergeSort{
5.
6. /**
7.  * 递归划分数组
8.  * @param arr
9.  * @param from
10. * @param end
11. * @param c void
12. */
13. public void partition(Integer[] arr, int from, int end) {
14.     //划分到数组只有一个元素时才不进行再划分
15.     if (from < end) {
16.         //从中间划分成两个数组
17.         int mid = (from + end) / 2;
18.         partition(arr, from, mid);
19.         partition(arr, mid + 1, end);
20.         //合并划分后的两个数组
21.         merge(arr, from, end, mid);
22.     }
23. }
24.
25. /**
26. * 数组合并，合并过程中对两部分数组进行排序
27. * 前后两部分数组里是有序的
28. * @param arr
29. * @param from
30. * @param end
31. * @param mid
```

```
32.     * @param c void
33.     */
34.     public void merge(Integer[] arr, int from, int end, int mid) {
35.         Integer[] tmpArr = new Integer[10];
36.         int tmpArrIndex = 0;//指向临时数组
37.         int part1ArrIndex = from;//指向第一部分数组
38.         int part2ArrIndex = mid + 1;//指向第二部分数组
39.
40.         //由于两部分数组里是有序的，所以每部分可以从第一个元素依次取到最后一个元素，再对两
        部分
41.         //取出的元素进行比较。只要某部分数组元素取完后，退出循环
42.         while ((part1ArrIndex <= mid) && (part2ArrIndex <= end)) {
43.             //从两部分数组里各取一个进行比较，取最小一个并放入临时数组中
44.             if (arr[part1ArrIndex] - arr[part2ArrIndex] < 0) {
45.                 //如果第一部分数组元素小，则将第一部分数组元素放入临时数组中，并且临时数组指
                针
46.                 //tmpArrIndex 下移一个以做好下次存储位置准备，前部分数组指针 part1ArrIndex
47.                 //也要下移一个以便下次取出下一个元素与后部分数组元素比较
48.                 tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
49.             } else {
50.                 //如果第二部分数组元素小，则将第二部分数组元素放入临时数组中
51.                 tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
52.             }
53.         }
54.         //由于退出循环后，两部分数组中可能有一个数组元素还未处理完，所以需要额外的处理，当
        然不可
55.         //能两部分数组都有未处理完的元素，所以下面两个循环最多只有一个会执行，并且都是大于
        已放入
56.         //临时数组中的元素
57.         while (part1ArrIndex <= mid) {
58.             tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
59.         }
60.         while (part2ArrIndex <= end) {
61.             tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
62.         }
```

```
63.  
64.    //最后把临时数组拷贝到源数组相同的位置  
65.    System.arraycopy(tmpArr, 0, arr, from, end - from + 1);  
66. }  
67.  
68. /**  
69. * 测试  
70. * @param args  
71. */  
72. public static void main(String[] args) {  
73.     Integer[] intgArr = { 5,9,1,4,2,6,3,8,0,7 };  
74.     MergeSort insertSort = new MergeSort();  
75.     insertSort.partition(intgArr,0,intgArr.length-1);  
76.     for(Integer a:intgArr){  
77.         System.out.print(a + " ");  
78.     }  
79. }  
80. }
```

[java] [view plain](#) [copy](#) [print?](#)

```
1. /**  
2. 归并排序：里面是一个递归程序，深刻理解之。  
3. */  
4. public class MergeSort {  
5.  
6.     /**  
7.      * 递归划分数组  
8.      * @param arr  
9.      * @param from  
10.     * @param end  
11.     * @param c void  
12.     */  
13.    public void partition(Integer[] arr, int from, int end) {  
14.        //划分到数组只有一个元素时才不进行再划分  
15.        if (from < end) {
```

```
16.     //从中间划分成两个数组
17.     int mid = (from + end) / 2;
18.     partition(arr, from, mid);
19.     partition(arr, mid + 1, end);
20.     //合并划分后的两个数组
21.     merge(arr, from, end, mid);
22. }
23. }
24.
25. /**
26. * 数组合并，合并过程中对两部分数组进行排序
27. * 前后两部分数组里是有序的
28. * @param arr
29. * @param from
30. * @param end
31. * @param mid
32. * @param c void
33. */
34. public void merge(Integer[] arr, int from, int end, int mid) {
35.     Integer[] tmpArr = new Integer[10];
36.     int tmpArrIndex = 0;//指向临时数组
37.     int part1ArrIndex = from;//指向第一部分数组
38.     int part2ArrIndex = mid + 1;//指向第二部分数组
39.
40.     //由于两部分数组里是有序的，所以每部分可以从第一个元素依次取到最后一个元素，再对两
        部分
41.     //取出的元素进行比较。只要某部分数组元素取完后，退出循环
42.     while ((part1ArrIndex <= mid) && (part2ArrIndex <= end)) {
43.         //从两部分数组里各取一个进行比较，取最小一个并放入临时数组中
44.         if (arr[part1ArrIndex] - arr[part2ArrIndex] < 0) {
45.             //如果第一部分数组元素小，则将第一部分数组元素放入临时数组中，并且临时数组指
                针
46.             //tmpArrIndex 下移一个以做好下次存储位置准备，前部分数组指针 part1ArrIndex
47.             //也要下移一个以便下次取出下一个元素与后部分数组元素比较
48.             tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
```

```
49.         } else {
50.             //如果第二部分数组元素小，则将第二部分数组元素放入临时数组中
51.             tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
52.         }
53.     }
54.     //由于退出循环后，两部分数组中可能有一个数组元素还未处理完，所以需要额外的处理，当然不可
55.     //能两部分数组都有未处理完的元素，所以下面两个循环最多只有一个会执行，并且都是大于已放入
56.     //临时数组中的元素
57.     while (part1ArrIndex <= mid) {
58.         tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
59.     }
60.     while (part2ArrIndex <= end) {
61.         tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
62.     }
63.
64.     //最后把临时数组拷贝到源数组相同的位置
65.     System.arraycopy(tmpArr, 0, arr, from, end - from + 1);
66. }
67.
68. /**
69. * 测试
70. * @param args
71. */
72. public static void main(String[] args) {
73.     Integer[] intgArr = {5,9,1,4,2,6,3,8,0,7};
74.     MergeSort insertSort = new MergeSort();
75.     insertSort.partition(intgArr,0,intgArr.length-1);
76.     for(Integer a:intgArr){
77.         System.out.print(a + " ");
78.     }
79. }
80. }
```

七、堆排序(Heap Sort)

1. 基本思想:

堆排序是一树形选择排序，在排序过程中，将 $R[1..N]$ 看成是一颗完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。

2. 堆的定义: N 个元素的序列 $K_1, K_2, K_3, \dots, K_n$ 称为堆，当且仅当该序列满足特性：

$$K_i \leq K_{2i} \quad K_i \leq K_{2i+1} \quad (1 \leq i \leq [N/2])$$

堆实质上是满足如下性质的完全二叉树：树中任一非叶子结点的关键字均大于等于其孩子结点的关键字。例如序列 10, 15, 56, 25, 30, 70 就是一个堆，它对应的完全二叉树如上图所示。这种堆中根结点（称为堆顶）的关键字最小，我们把它称为小根堆。反之，若完全二叉树中任一非叶子结点的关键字均大于等于其孩子的关键字，则称之为大根堆。

3. 排序过程:

堆排序正是利用小根堆（或大根堆）来选取当前无序区中关键字小（或最大）的记录实现排序的。我们不妨利用大根堆来排序。每一趟排序的基本操作是：将当前无序区调整为一个大根堆，选取关键字最大的堆顶记录，将它和无序区中的最后一个记录交换。这样，正好和直接选择排序相反，有序区是在原记录区的尾部形成并逐步向前扩大到整个记录区。

【示例】：对关键字序列 42, 13, 91, 23, 24, 16, 05, 88 建堆

java 代码实现：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1.  /**
2.   * 选择排序之堆排序:
3.   */
4.  public class HeapSort {
5.
6.      /**
7.       * 排序算法的实现, 对数组中指定的元素进行排序
8.       * @param array 待排序的数组
9.       * @param from 从哪里开始排序
10.      * @param end 排到哪里
11.      * @param c 比较器
12.      */
13.     public void sort(Integer[] array, int from, int end) {
```

```
14.     //创建初始堆
15.     initialHeap(array, from, end);
16.
17.     /*
18.      * 对初始堆进行循环，且从最后一个节点开始，直接树只有两个节点止
19.      * 每轮循环后丢弃最后一个叶子节点，再看作一个新的树
20.      */
21.     for (int i = end - from + 1; i >= 2; i--) {
22.         //根节点与最后一个叶子节点交换位置，即数组中的第一个元素与最后一个元素互换
23.         swap(array, from, i - 1);
24.         //交换后需要重新调整堆
25.         adjustNote(array, 1, i - 1);
26.     }
27.
28. }
29.
30. /**
31.  * 初始化堆
32.  * 比如原序列为： 7,2,4,3,12,1,9,6,8,5,10,11
33.  * 则初始堆为： 1,2,4,3,5,7,9,6,8,12,10,11
34.  * @param arr 排序数组
35.  * @param from 从哪
36.  * @param end 到哪
37.  * @param c 比较器
38. */
39. private void initialHeap(Integer[] arr, int from, int end) {
40.     int lastBranchIndex = (end - from + 1) / 2;//最后一个非叶子节点
41.     //对所有的非叶子节点进行循环，且从最一个非叶子节点开始
42.     for (int i = lastBranchIndex; i >= 1; i--) {
43.         adjustNote(arr, i, end - from + 1);
44.     }
45. }
46.
47. /**
48.  * 调整节点顺序，从父、左右子节点三个节点中选择一个最大节点与父节点转换
```

```
49.     * @param arr 待排序数组
50.     * @param parentNodeIndex 要调整的节点，与它的子节点一起进行调整
51.     * @param len 树的节点数
52.     * @param c 比较器
53.     */
54.    private void adjustNote(Integer[] arr, int parentNodeIndex, int len) {
55.        int minNodeIndex = parentNodeIndex;
56.        //如果有左子树，i * 2 为左子节点索引
57.        if (parentNodeIndex * 2 <= len) {
58.            //如果父节点小于左子树时
59.            if ((arr[parentNodeIndex - 1].compareTo(arr[parentNodeIndex * 2 - 1])) < 0) {
60.                minNodeIndex = parentNodeIndex * 2;//记录最大索引为左子节点索引
61.            }
62.
63.            //只有在有或子树的前提下才可能有右子树，再进一步判断是否有右子树
64.            if (parentNodeIndex * 2 + 1 <= len) {
65.                //如果右子树比最大节点更大
66.                if ((arr[minNodeIndex - 1].compareTo(arr[(parentNodeIndex * 2 + 1) - 1])) < 0) {
67.                    minNodeIndex = parentNodeIndex * 2 + 1;//记录最大索引为右子节点索引
68.                }
69.            }
70.        }
71.
72.        //如果在父节点、左、右子节点三都中，最大节点不是父节点时需交换，把最大的与父节点交
73.        if (minNodeIndex != parentNodeIndex) {
74.            swap(arr, parentNodeIndex - 1, minNodeIndex - 1);
75.            //交换后可能需要重建堆，原父节点可能需要继续下沉
76.            if (minNodeIndex * 2 <= len) {//是否有子节点，注，只需判断是否有左子树即可知道
77.                adjustNote(arr, minNodeIndex, len);
78.            }
79.        }
80.    }
81.
82.    /**
83.     * 将从索引 start 到 end 的子数组逆序
84.     * @param arr 待操作数组
85.     * @param start 子数组起始索引
86.     * @param end 子数组结束索引
87.     */
88.    private void reverse(Integer[] arr, int start, int end) {
89.        while (start < end) {
90.            swap(arr, start, end);
91.            start++;
92.            end--;
93.        }
94.    }
95.
96.    /**
97.     * 将 arr 中的元素按升序排序
98.     * @param arr 待排序数组
99.     */
100.    public void sort(Integer[] arr) {
101.        int len = arr.length;
102.        if (len > 1) {
103.            for (int i = len / 2 - 1; i >= 0; i--) {
104.                adjustNote(arr, i, len);
105.            }
106.            for (int i = len - 1; i >= 0; i--) {
107.                if (arr[i] != arr[0]) {
108.                    swap(arr, 0, i);
109.                    adjustNote(arr, 0, i);
110.                }
111.            }
112.        }
113.    }
114.
```

```
83.     * 交换数组中的两个元素的位置
84.     * @param array 待交换的数组
85.     * @param i 第一个元素
86.     * @param j 第二个元素
87.     */
88.    public void swap(Integer[] array, int i, int j) {
89.        if (i != j) { //只有不是同一位置时才需交换
90.            Integer tmp = array[i];
91.            array[i] = array[j];
92.            array[j] = tmp;
93.        }
94.    }
95.
96.    /**
97.     * 测试
98.     * @param args
99.     */
100.    public static void main(String[] args) {
101.        Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
102.        HeapSort heapsort = new HeapSort();
103.        heapsort.sort(intgArr,0,intgArr.length-1);
104.        for(Integer intObj:intgArr){
105.            System.out.print(intObj + " ");
106.        }
107.    }
108.
109. }
```

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 选择排序之堆排序:
3.  */
4. public class HeapSort {
5.
6.     /**
```

```
7.     * 排序算法的实现，对数组中指定的元素进行排序
8.     * @param array 待排序的数组
9.     * @param from 从哪里开始排序
10.    * @param end 排到哪里
11.    * @param c 比较器
12.    */
13.   public void sort(Integer[] array, int from, int end) {
14.       //创建初始堆
15.       initialHeap(array, from, end);
16.
17.       /*
18.           * 对初始堆进行循环，且从最后一个节点开始，直接树只有两个节点止
19.           * 每轮循环后丢弃最后一个叶子节点，再看作一个新的树
20.           */
21.       for (int i = end - from + 1; i >= 2; i--) {
22.           //根节点与最后一个叶子节点交换位置，即数组中的第一个元素与最后一个元素互换
23.           swap(array, from, i - 1);
24.           //交换后需要重新调整堆
25.           adjustNote(array, 1, i - 1);
26.       }
27.
28.   }
29.
30.   /**
31.     * 初始化堆
32.     * 比如原序列为： 7,2,4,3,12,1,9,6,8,5,10,11
33.     * 则初始堆为： 1,2,4,3,5,7,9,6,8,12,10,11
34.     * @param arr 排序数组
35.     * @param from 从哪
36.     * @param end 到哪
37.     * @param c 比较器
38.     */
39.   private void initialHeap(Integer[] arr, int from, int end) {
40.       int lastBranchIndex = (end - from + 1) / 2;//最后一个非叶子节点
41.       //对所有的非叶子节点进行循环，且从最一个非叶子节点开始
```

```
42.     for (int i = lastBranchIndex; i >= 1; i--) {
43.         adjustNote(arr, i, end - from + 1);
44.     }
45. }
46.
47. /**
48. * 调整节点顺序，从父、左右子节点三个节点中选择一个最大节点与父节点转换
49. * @param arr 待排序数组
50. * @param parentNodeIndex 要调整的节点，与它的子节点一起进行调整
51. * @param len 树的节点数
52. * @param c 比较器
53. */
54. private void adjustNote(Integer[] arr, int parentNodeIndex, int len) {
55.     int minNodeIndex = parentNodeIndex;
56.     //如果有左子树，i * 2 为左子节点索引
57.     if (parentNodeIndex * 2 <= len) {
58.         //如果父节点小于左子树时
59.         if ((arr[parentNodeIndex - 1].compareTo(arr[parentNodeIndex * 2 - 1])) < 0) {
60.             minNodeIndex = parentNodeIndex * 2;//记录最大索引为左子节点索引
61.         }
62.
63.         //只有在有或子树的前提下才可能有右子树，再进一步判断是否有右子树
64.         if (parentNodeIndex * 2 + 1 <= len) {
65.             //如果右子树比最大节点更大
66.             if ((arr[minNodeIndex - 1].compareTo(arr[(parentNodeIndex * 2 + 1) - 1])) < 0) {
67.                 minNodeIndex = parentNodeIndex * 2 + 1;//记录最大索引为右子节点索引
68.             }
69.         }
70.     }
71.
72.     //如果在父节点、左、右子节点三都中，最大节点不是父节点时需交换，把最大的与父节点交
73.     if (minNodeIndex != parentNodeIndex) {
74.         swap(arr, parentNodeIndex - 1, minNodeIndex - 1);
75.         //交换后可能需要重建堆，原父节点可能需要继续下沉
```

```
76.         if (minNodeIndex * 2 <= len) { //是否有子节点, 注, 只需判断是否有左子树即可知道
77.             adjustNote(arr, minNodeIndex, len);
78.         }
79.     }
80. }
81.
82. /**
83. * 交换数组中的两个元素的位置
84. * @param array 待交换的数组
85. * @param i 第一个元素
86. * @param j 第二个元素
87. */
88. public void swap(Integer[] array, int i, int j) {
89.     if (i != j) { //只有不是同一位置时才需交换
90.         Integer tmp = array[i];
91.         array[i] = array[j];
92.         array[j] = tmp;
93.     }
94. }
95.
96. /**
97. * 测试
98. * @param args
99. */
100. public static void main(String[] args) {
101.     Integer[] intgArr = { 5, 9, 1, 4, 2, 6, 3, 8, 0, 7 };
102.     HeapSort heapsort = new HeapSort();
103.     heapsort.sort(intgArr,0,intgArr.length-1);
104.     for(Integer intObj:intgArr){
105.         System.out.print(intObj + " ");
106.     }
107. }
108.
109. }
```

八、桶式排序

java 代码实现：

[java] [view plain](#) [copy](#) [print](#)?

```
1.  /**
2.   * 桶式排序:
3.   * 桶式排序不再是基于比较的了，它和基数排序同属于分配类的排序,
4.   * 这类排序的特点是事先要知道待排 序列的一些特征。
5.   * 桶式排序事先要知道待排 序列在一个范围内，而且这个范围应该不是很大的。
6.   * 比如知道待排序列在[0,M) 内，那么可以分配M 个桶，第 I 个桶记录 I 的出现情况,
7.   * 最后根据每个桶收到的位置信息把数据输出成有序的形式。
8.   * 这里我们用两个临时性数组，一个用于记录位置信息，一个用于方便输出数据成有序方式,
9.   * 另外我们假设数据落在 0 到 MAX,如果所给数据不是从 0 开始,你可以把每个数减去最小的数。
10.  */
11. public class BucketSort {
12.     public void sort(int[] keys,int from,int len,int max)
13.     {
14.         int[] temp=new int[len];
15.         int[] count=new int[max];
16.
17.
18.         for(int i=0;i<len;i++)
19.         {
20.             count[keys[from+i]]++;
21.         }
22.         //calculate position info
23.         for(int i=1;i<max;i++)
24.         {
25.             count[i]=count[i]+count[i-1];//这意味着有多少数目小于或等于 i, 因此它也是
26.             position+ 1
27.         }
28.         System.arraycopy(keys, from, temp, 0, len);
29.         for(int k=len-1;k>=0;k--)//从最末到开头保持稳定性
30.         {
31.             keys[--count[temp[k]]]=temp[k];// position +1 =count
```

```
32.     }
33.     }
34.     /**
35.      * @param args
36.     */
37.    public static void main(String[] args) {
38.
39.        int[] a={1,4,8,3,2,9,5,0,7,6,9,10,9,13,14,15,11,12,17,16};
40.        BucketSort bucketSort=new BucketSort();
41.        bucketSort.sort(a,0,a.length,20);//actually is 18, but 20 will also work
42.
43.
44.        for(int i=0;i<a.length;i++)
45.        {
46.            System.out.print(a[i]+",");
47.        }
48.
49.    }
50.
51.
52. }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1.    /**
2.     * 桶式排序:
3.     * 桶式排序不再是基于比较的了, 它和基数排序同属于分配类的排序,
4.     * 这类排序的特点是事先要知道待排 序列的一些特征。
5.     * 桶式排序事先要知道待排 序列在一个范围内, 而且这个范围应该不是很大的。
6.     * 比如知道待排序列在[0,M) 内, 那么可以分配M 个桶, 第I 个桶记录 I 的出现情况,
7.     * 最后根据每个桶收到的位置信息把数据输出成有序的形式。
8.     * 这里我们用两个临时性数组, 一个用于记录位置信息, 一个用于方便输出数据成有序方式,
9.     * 另外我们假设数据落在 0 到 MAX,如果所给数据不是从 0 开始, 你可以把每个数减去最小的数。
10.    */
11.   public class BucketSort {
12.       public void sort(int[] keys,int from,int len,int max)
```

```
13.    {
14.        int[] temp=new int[len];
15.        int[] count=new int[max];
16.
17.
18.        for(int i=0;i<len;i++)
19.        {
20.            count[keys[from+i]]++;
21.        }
22.        //calculate position info
23.        for(int i=1;i<max;i++)
24.        {
25.            count[i]=count[i]+count[i-1];//这意味着有多少数目小于或等于 i, 因此它也是
position+ 1
26.        }
27.
28.        System.arraycopy(keys, from, temp, 0, len);
29.        for(int k=len-1;k>=0;k--)//从最末到开头保持稳定性
30.        {
31.            keys[--count[temp[k]]]=temp[k];// position +1 =count
32.        }
33.    }
34.    /**
35.     * @param args
36.     */
37.    public static void main(String[] args) {
38.
39.        int[] a={1,4,8,3,2,9,5,0,7,6,9,10,9,13,14,15,11,12,17,16};
40.        BucketSort bucketSort=new BucketSort();
41.        bucketSort.sort(a,0,a.length,20);//actually is 18, but 20 will also work
42.
43.
44.        for(int i=0;i<a.length;i++)
45.        {
46.            System.out.print(a[i]+",");
```

```
47.         }
48.
49.     }
50.
51.
52. }
```

九、基数排序

java 代码实现:

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. /**
2. * 基数排序:
3. */
4. import java.util.Arrays;
5. public class RadixSort {
6.
7. /**
8. * 取数 x 上的第 d 位数字
9. * @param x 数
10. * @param d 第几位, 从低位到高位
11. * @return
12. */
13. public int digit(long x, long d) {
14.
15.     long pow = 1;
16.     while (--d > 0) {
17.         pow *= 10;
18.     }
19.     return (int) (x / pow % 10);
20. }
21.
22. /**
```

```
23.     * 基数排序实现，以升序排序（下面程序中的位记录器 count 中，从第 0 个元素到第 9 个元素  
依次用来  
24.     * 记录当前比较位是 0 的有多少个..是 9 的有多少个数，而降序时则从第 0 个元素到第 9 个元素  
依次用来  
25.     * 记录当前比较位是 9 的有多少个..是 0 的有多少个数)  
26.     * @param arr 待排序数组  
27.     * @param digit 数组中最大数的位数  
28.     * @return  
29.     */  
30.    public long[] radixSortAsc(long[] arr) {  
31.        //从低位往高位循环  
32.        for (int d = 1; d <= getMax(arr); d++) {  
33.            //临时数组，用来存放排序过程中的数据  
34.            long[] tmpArray = new long[arr.length];  
35.            //位记数器，从第 0 个元素到第 9 个元素依次用来记录当前比较位是 0 的有多少个..是 9 的  
有多少个数  
36.            int[] count = new int[10];  
37.            //开始统计 0 有多少个，并存储在第 0 位，再统计 1 有多少个，并存储在第 1 位..依次统计  
到 9 有多少个  
38.            for (int i = 0; i < arr.length; i++) {  
39.                count[digit(arr[i], d)] += 1;  
40.            }  
41.            /*  
42.             * 比如某次经过上面统计后结果为：[0, 2, 3, 3, 0, 0, 0, 0, 0, 0]则经过下面计算后结果为：  
43.             * [0, 2, 5, 8, 8, 8, 8, 8, 8, 8]但实质上只有如下[0, 2, 5, 8, 0, 0, 0, 0, 0, 0]中  
44.             * 非零数才用到，因为其他位不存在，它们分别表示如下：2 表示比较位为 1 的元素可以  
存放在索引为 1、0 的  
45.             * 位置，5 表示比较位为 2 的元素可以存放在 4、3、2 三个(5-2=3)位置，8 表示比较位为 3  
的元素可以存放在  
46.             * 7、6、5 三个(8-5=3)位置  
47.             */  
48.            for (int i = 1; i < 10; i++) {  
49.                count[i] += count[i - 1];  
50.            }  
51.        }
```

```
52.      /*
53.       * 注，这里只能从数组后往前循环，因为排序时还需保持以前的已排序好的顺序，不应该
54.       * 打
55.       * 乱原来已排好的序，如果从前往后处理，则会把原来在前面会摆到后面去，因为在处理
56.       * 某个
57.       * 元素的位置时，位记数器是从大到到小（count[digit(arr[i], d)]--）的方式来处
58.       * 理的，即先存放索引大的元素，再存放索引小的元素，所以需从最后一个元素开始处理。
59.       * 如有这样的一个序列[212,213,312]，如果按照从第一个元素开始循环的话，经过第一轮
60.       * 后（个位）排序后，得到这样一个序列[312,212,213]，第一次好像没什么问题，但问题
61.       * 会
62.       * 从第二轮开始出现，第二轮排序后，会得到[213,212,312]，这样个位为3的元素本应该
63.       * 放在最后，但经过第二轮后却排在了前面了，所以出现了问题
64.       */
65.      for (int i = arr.length - 1; i >= 0; i--) {//只能从最后一个元素往前处理
66.          //for (int i = 0; i < arr.length; i++) {//不能从第一个元素开始循环
67.              tmpArray[count[digit(arr[i], d)] - 1] = arr[i];
68.              count[digit(arr[i], d)]--;
69.          }
70.      }
71.      System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);
72.  }
73. /**
74.  * 基数排序实现，以降序排序（下面程序中的位记录器 count 中，从第 0 个元素到第 9 个元素
75.  * 依次用来
76.  * 记录当前比较位是 0 的有多少个..是 9 的有多少个数，而降序时则从第 0 个元素到第 9 个元素
77.  * 依次用来
78.  * 记录当前比较位是 9 的有多少个..是 0 的有多少个数）
79.  * @param arr 待排序数组
80.  * @return
81.  */
82. public long[] radixSortDesc(long[] arr) {
83.     for (int d = 1; d <= getMax(arr); d++) {
```

```
82.     long[] tmpArray = new long[arr.length];
83.     //位记数器，从第 0 个元素到第 9 个元素依次用来记录当前比较位是 9 的有多少个..是 0 的
84.     //有多少个数
85.     int[] count = new int[10];
86.     //开始统计 0 有多少个，并存储在第 9 位，再统计 1 有多少个，并存储在第 8 位..依次统计
87.     //到 9 有多少个，并存储在第 0 位
88.     for (int i = 0; i < arr.length; i++) {
89.         count[9 - digit(arr[i], d)] += 1;
90.     }
91.     for (int i = 1; i < 10; i++) {
92.         count[i] += count[i - 1];
93.     }
94.     for (int i = arr.length - 1; i >= 0; i--) {
95.         tmpArray[count[9 - digit(arr[i], d)] - 1] = arr[i];
96.         count[9 - digit(arr[i], d)]--;
97.     }
98. }
99.
100.    System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);
101. }
102.    return arr;
103. }
104.
105. private int getMax(long[] array) {
106.     int maxIndex = 0;
107.     for (int j = 1; j < array.length; j++) {
108.         if (array[j] > array[maxIndex]) {
109.             maxIndex = j;
110.         }
111.     }
112.     return String.valueOf(array[maxIndex]).length();
113. }
114.
115. public static void main(String[] args) {
```

```
116.     long[] ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
117.     RadixSort rs = new RadixSort();
118.     System.out.println("升 - " + Arrays.toString(rs.radixSortAsc(ary)));
119.
120.     ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
121.     System.out.println("降 - " + Arrays.toString(rs.radixSortDesc(ary)));
122. }
123. }
```

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. /**
2. * 基数排序:
3. */
4. import java.util.Arrays;
5. public class RadixSort {
6.
7. /**
8. * 取数 x 上的第 d 位数字
9. * @param x 数
10. * @param d 第几位, 从低位到高位
11. * @return
12. */
13. public int digit(long x, long d) {
14.
15.     long pow = 1;
16.     while (--d > 0) {
17.         pow *= 10;
18.     }
19.     return (int) (x / pow % 10);
20. }
21.
22. /**
23. * 基数排序实现, 以升序排序 (下面程序中的位记录器 count 中, 从第 0 个元素到第 9 个元素
依次用来
```

```
24.     * 记录当前比较位是 0 的有多少个..是 9 的有多少个数, 而降序时则从第 0 个元素到第 9 个元素  
依次用来  
25.     * 记录当前比较位是 9 的有多少个..是 0 的有多少个数)  
26.     * @param arr 待排序数组  
27.     * @param digit 数组中最大数的位数  
28.     * @return  
29.     */  
30.    public long[] radixSortAsc(long[] arr) {  
31.        //从低位往高位循环  
32.        for (int d = 1; d <= getMax(arr); d++) {  
33.            //临时数组, 用来存放排序过程中的数据  
34.            long[] tmpArray = new long[arr.length];  
35.            //位记数器, 从第 0 个元素到第 9 个元素依次用来记录当前比较位是 0 的有多少个..是 9 的  
有多少个数  
36.            int[] count = new int[10];  
37.            //开始统计 0 有多少个, 并存储在第 0 位, 再统计 1 有多少个, 并存储在第 1 位..依次统计  
到 9 有多少个  
38.            for (int i = 0; i < arr.length; i++) {  
39.                count[digit(arr[i], d)] += 1;  
40.            }  
41.            /*  
42.             * 比如某次经过上面统计后结果为: [0, 2, 3, 3, 0, 0, 0, 0, 0, 0]则经过下面计算后 结果为:  
43.             * [0, 2, 5, 8, 8, 8, 8, 8, 8, 8]但实质上只有如下[0, 2, 5, 8, 0, 0, 0, 0, 0, 0]中  
44.             * 非零数才用到, 因为其他位不存在, 它们分别表示如下: 2 表示比较位为 1 的元素可以  
存放在索引为 1、0 的  
45.             * 位置, 5 表示比较位为 2 的元素可以存放在 4、3、2 三个(5-2=3)位置, 8 表示比较位为 3  
的元素可以存放在  
46.             * 7、6、5 三个(8-5=3)位置  
47.             */  
48.            for (int i = 1; i < 10; i++) {  
49.                count[i] += count[i - 1];  
50.            }  
51.            /*
```

```
53.     * 注，这里只能从数组后往前循环，因为排序时还需保持以前的已排序好的顺序，不应该打
54.     * 乱原来已排好的序，如果从前往后处理，则会把原来在前面会摆到后面去，因为在处理某个
55.     * 元素的位置时，位记数器是从大到到小（count[digit(arr[i], d)]--）的方式来处
56.     * 理的，即先存放索引大的元素，再存放索引小的元素，所以需从最后一个元素开始处理。
57.     * 如有这样一个序列[212,213,312]，如果按照从第一个元素开始循环的话，经过第一轮
58.     * 后（个位）排序后，得到这样一个序列[312,212,213]，第一次好像没什么问题，但问题
      会
59.     * 从第二轮开始出现，第二轮排序后，会得到[213,212,312]，这样个位为 3 的元素本应该
60.     * 放在最后，但经过第二轮后却排在了前面了，所以出现了问题
61.     */
62.     for (int i = arr.length - 1; i >= 0; i--) {//只能从最后一个元素往前处理
63.         //for (int i = 0; i < arr.length; i++) {//不能从第一个元素开始循环
64.             tmpArray[count[digit(arr[i], d)] - 1] = arr[i];
65.             count[digit(arr[i], d)]--;
66.         }
67.
68.         System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);
69.     }
70.     return arr;
71. }
72.
73. /**
74.     * 基数排序实现，以降序排序（下面程序中的位记录器 count 中，从第 0 个元素到第 9 个元素
      依次用来
75.     * 记录当前比较位是 0 的有多少个..是 9 的有多少个数，而降序时则从第 0 个元素到第 9 个元素
      依次用来
76.     * 记录当前比较位是 9 的有多少个..是 0 的有多少个数）
77.     * @param arr 待排序数组
78.     * @return
79.     */
80.     public long[] radixSortDesc(long[] arr) {
81.         for (int d = 1; d <= getMax(arr); d++) {
82.             long[] tmpArray = new long[arr.length];
```

```
83.      //位记数器，从第 0 个元素到第 9 个元素依次用来记录当前比较位是 9 的有多少个..是 0 的  
84.     有多少个数  
85.      int[] count = new int[10];  
86.      //开始统计 0 有多少个，并存储在第 9 位，再统计 1 有多少个，并存储在第 8 位..依次统计  
87.      //到 9 有多少个，并存储在第 0 位  
88.      for (int i = 0; i < arr.length; i++) {  
89.          count[9 - digit(arr[i], d)] += 1;  
90.  
91.      for (int i = 1; i < 10; i++) {  
92.          count[i] += count[i - 1];  
93.      }  
94.  
95.      for (int i = arr.length - 1; i >= 0; i--) {  
96.          tmpArray[count[9 - digit(arr[i], d)] - 1] = arr[i];  
97.          count[9 - digit(arr[i], d)]--;  
98.      }  
99.  
100.     System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);  
101.    }  
102.    return arr;  
103.  }  
104.  
105. private int getMax(long[] array) {  
106.     int maxIndex = 0;  
107.     for (int j = 1; j < array.length; j++) {  
108.         if (array[j] > array[maxIndex]) {  
109.             maxIndex = j;  
110.         }  
111.     }  
112.     return String.valueOf(array[maxIndex]).length();  
113.  }  
114.  
115. public static void main(String[] args) {  
116.     long[] ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
```

```
117.     RadixSort rs = new RadixSort();
118.     System.out.println("升 - " + Arrays.toString(rs.radixSortAsc(ary)));
119.
120.     ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
121.     System.out.println("降 - " + Arrays.toString(rs.radixSortDesc(ary)));
122. }
123. }
```

十、几种排序算法的比较和选择

1. 选取排序方法需要考虑的因素:

- (1) 待排序的元素数目 n ;
- (2) 元素本身信息量的大小;
- (3) 关键字的结构及其分布情况;
- (4) 语言工具的条件, 辅助空间的大小等。

2. 小结:

- (1) 若 n 较小($n \leq 50$), 则可以采用直接插入排序或直接选择排序。由于直接插入排序所需的记录移动操作较直接选择排序多, 因而当记录本身信息量较大时, 用直接选择排序较好。
- (2) 若文件的初始状态已按关键字基本有序, 则选用直接插入或冒泡排序为宜。
- (3) 若 n 较大, 则应采用时间复杂度为 $O(n \log_2 n)$ 的排序方法: 快速排序、堆排序或归并排序。快速排序是目前基于比较的内部排序法中被认为是最好的方法。
- (4) 在基于比较排序方法中, 每次比较两个关键字的大小之后, 仅仅出现两种可能的转移, 因此可以用一棵二叉树来描述比较判定过程, 由此可以证明: 当文件的 n 个关键字随机分布时, 任何借助于"比较"的排序算法, 至少需要 $O(n \log_2 n)$ 的时间。
- (5) 当记录本身信息量较大时, 为避免耗费大量时间移动记录, 可以用链表作为存储结构。

排序简介

排序是数据处理中经常使用的一种重要运算, 在计算机及其应用系统中, 花费在排序上的时间在系统运行时间中占有很大比重; 并且排序本身对推动算法分析的发展也起很大作用。目前已有上百种排序方法, 但尚未有一个最理想的尽如人意的方法, 本章介绍常用的如下排序方法, 并对它们进行分析和比较。

- 1、插入排序（直接插入排序、折半插入排序、希尔排序）；
- 2、交换排序（起泡排序、快速排序）；
- 3、选择排序（直接选择排序、堆排序）；
- 4、归并排序；
- 5、基数排序；

学习重点

- 1、掌握排序的基本概念和各种排序方法的特点，并能加以灵活应用；
- 2、掌握插入排序(直接插入排序、折半插入排序、希尔排序)、交换排序（起泡排序、快速排序）、选择排序（直接选择排序、堆排序）、二路归并排序的方法及其性能分析方法；
- 3、了解基数排序方法及其性能分析方法。

排序（sort）或分类

所谓排序，就是要整理文件中的记录，使之按关键字递增(或递减)次序排列起来。其确切定义如下：

输入：n个记录 R_1, R_2, \dots, R_n ，其相应的关键字分别为 K_1, K_2, \dots, K_n 。

输出： $R_{i1}, R_{i2}, \dots, R_{in}$ ，使得 $K_{i1} \leq K_{i2} \leq \dots \leq K_{in}$ 。（或 $K_{i1} \geq K_{i2} \geq \dots \geq K_{in}$ ）。

1. 被排序对象--文件

被排序的对象--文件由一组记录组成。

记录则由若干个数据项(或域)组成。其中有一项可用来标识一个记录，称为关键字项。该数据项的值称为关键字(Key)。

注意：

在不易产生混淆时，将关键字项简称为关键字。

2. 排序运算的依据--关键字

用来作排序运算依据的关键字，可以是数字类型，也可以是字符类型。

关键字的选取应根据问题的要求而定。

【例】在高考成绩统计中将每个考生作为一个记录。每条记录包含准考证号、姓名、各科的分数和总分数等项内容。若要唯一地标识一个考生的记录，则必须用“准考证号”作为关键字。若要按照考生的总分数排名次，则需用“总分数”作为关键字。

排序的稳定性

当待排序记录的关键字均不相同时，排序结果是惟一的，否则排序结果不唯一。

在待排序的文件中，若存在多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是**稳定的**；若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是**不稳定的**。

注意：

排序算法的稳定性是针对所有输入实例而言的。即在所有可能的输入实例中，只要有一个实例使得算法不满足稳定性要求，则该排序算法就是不稳定的。

排序方法的分类

1. 按是否涉及数据的内、外存交换分

在排序过程中，若整个文件都是放在内存中处理，排序时不涉及数据的内、外存交换，则称之为**内部排序**(简称内排序)；反之，若排序过程中要进行数据的内、外存交换，则称之为**外部排序**。

注意：

- ① 内排序适用于记录个数不很多的小文件
- ② 外排序则适用于记录个数太多，不能一次将其全部记录放入内存的大文件。

2. 按策略划分内部排序方法

可以分为五类：插入排序、选择排序、交换排序、归并排序和分配排序。

排序算法分析

1. 排序算法的基本操作

大多数排序算法都有两个基本的操作：

- (1) 比较两个关键字的大小；
- (2) 改变指向记录的指针或移动记录本身。

注意：

第(2)种基本操作的实现依赖于待排序记录的存储方式。

2. 待排文件的常用存储方式

- (1) 以顺序表(或直接用向量)作为存储结构

排序过程：对记录本身进行物理重排（即通过关键字之间的比较判定，将记录移到合适的位置）

(2) 以链表作为存储结构

排序过程：无须移动记录，仅需修改指针。通常将这类排序称为链表(或链式)排序；

(3) 用顺序的方式存储待排序的记录，但同时建立一个辅助表(如包括关键字和指向记录位置的指针组成的索引表)

排序过程：只需对辅助表的表目进行物理重排（即只移动辅助表的表目，而不移动记录本身）。适用于难于在链表上实现，仍需避免排序过程中移动记录的排序方法。

3. 排序算法性能评价

(1) 评价排序算法好坏的标准

评价排序算法好坏的标准主要有两条：

- ① 执行时间和所需的辅助空间
- ② 算法本身的复杂程度

(2) 排序算法的空间复杂度

若排序算法所需的辅助空间并不依赖于问题的规模 n ，即辅助空间是 $O(1)$ ，则称之为就地排序(In-PlaceSort)。

非就地排序一般要求的辅助空间为 $O(n)$ 。

(3) 排序算法的时间开销

大多数排序算法的时间开销主要是关键字之间的比较和记录的移动。有的排序算法其执行时间不仅依赖于问题的规模，还取决于输入实例中数据的状态。

文件的顺序存储结构表示

```
#define n 100 //假设的文件长度，即待排序的记录数目
typedef int KeyType; //假设的关键字类型
typedef struct{ //记录类型
    KeyType key; //关键字项
    InfoType otherinfo; //其它数据项，类型 InfoType 依赖于具体应用而定义
}RecType;
typedef RecType SeqList[n+1]; //SeqList 为顺序表类型，表中第 0 个单元一般用作哨兵
```

注意：

若关键字类型没有比较算符，则可事先定义宏或函数来表示比较运算。

【例】关键字为字符串时，可定义宏 "#define LT(a, b)(Stromp((a), (b))<0)"。那么算法中 "a<b" 可用 "LT(a, b)" 取代。若使用 C++，则定义重载的算符 "<" 更为方便。

按平均时间将排序分为四类：

(1) 平方阶($O(n^2)$)排序

一般称为简单排序，例如直接插入、直接选择和冒泡排序；

(2) 线性对数阶($O(n \lg n)$)排序

如快速、堆和归并排序；

(3) $O(n^{1+\varepsilon})$ 阶排序

ε 是介于 0 和 1 之间的常数，即 $0 < \varepsilon < 1$ ，如希尔排序；

(4) 线性阶($O(n)$)排序

如桶、箱和基数排序。

各种排序方法比较

简单排序中直接插入最好，快速排序最快，当文件为正序时，直接插入和冒泡均最佳。

影响排序效果的因素

因为不同的排序方法适应不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素：

- ①待排序的记录数目 n ；
- ②记录的大小(规模)；

③关键字的结构及其初始状态；

④对稳定性的要求；

⑤语言工具的条件；

⑥存储结构；

⑦时间和辅助空间复杂度等。

不同条件下，排序方法的选择

(1)若 n 较小(如 $n \leq 50$)，可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插入、冒泡或随机的快速排序为宜；

(3)若 n 较大，则应采用时间复杂度为 $O(n \lg n)$ 的排序方法：快速排序、堆排序或归并排序。

快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

若要求排序稳定，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

4)在基于比较的排序方法中，每次比较两个关键字的大小之后，仅仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程。

当文件的 n 个关键字随机分布时，任何借助于“比较”的排序算法，至少需要 $O(n \lg n)$ 的时间。

箱排序和基数排序只需一步就会引起 m 种可能的转移，即把一个记录装入 m 个箱子之一，因此在一般情况下，箱排序和基数排序可能在 $O(n)$ 时间内完成对 n 个记录的排序。但是，箱排序和基数排序只适用于像字符串和整数这类有明显结构特征的关键字，而当关键字的取值范围属于某个无穷集合(例如实数型关键字)时，无法使用箱排序和基数排序，这时只有借助于“比较”的方法来排序。

若 n 很大，记录的关键字位数较少且可以分解时，采用基数排序较好。虽然桶排序对

关键字的结构无要求，但它也只有在关键字是随机分布时才能使平均时间达到线性阶，否则为平方阶。同时要注意，箱、桶、基数这三种分配排序均假定了关键字若为数字时，则其值均是非负的，否则将其映射到箱(桶)号时，又要增加相应的时间。

(5)有的语言(如 Fortran, Cobol 或 Basic 等)没有提供指针及递归，导致实现归并、快速(它们用递归实现较简单)和基数(使用了指针)等排序算法变得复杂。此时可考虑用其它排序。

(6)本章给出的排序算法，输入数据均是存储在一个向量中。当记录的规模较大时，为避免耗费大量的时间去移动记录，可以用链表作为存储结构。譬如插入排序、归并排序、基数排序都易于在链表上实现，使之减少记录的移动次数。但有的排序方法，如快速排序和堆排序，在链表上却难于实现，在这种情况下，可以提取关键字建立索引表，然后对索引表进行排序。然而更为简单的方法是：引入一个整型向量 t 作为辅助表，排序前令 $t[i]=i(0 \leq i < n)$ ，若排序算法中要求交换 $R[i]$ 和 $R[j]$ ，则只需交换 $t[i]$ 和 $t[j]$ 即可；排序结束后，向量 t 就指示了记录之间的顺序关系：

$$R[t[0]].key \leq R[t[1]].key \leq \dots \leq R[t[n-1]].key$$

若要求最终结果是：

$$R[0].key \leq R[1].key \leq \dots \leq R[n-1].key$$

则可以在排序结束后，再按辅助表所规定的次序重排各记录，完成这种重排的时间是 $O(n)$ 。

1.15.面试题集合（十四）

1.15.1. 判断图里有环

无向图：

法 1：

如果存在回路，则必存在一个子图，是一个环路。环路中所有顶点的度 ≥ 2 。

n 算法：

第一步：删除所有度 ≤ 1 的顶点及相关的边，并将另外与这些边相关的其它顶点的度减一。

第二步：将度数变为 1 的顶点排入队列，并从该队列中取出一个顶点重复步骤一。

如果最后还有未删除顶点，则存在环，否则没有环。

n 算法分析：

由于有 m 条边， n 个顶点。如果 $m \geq n$ ，则根据图论知识可直接判断存在环路。

(证明：如果没有环路，则该图必然是 k 棵树 $k \geq 1$ 。根据树的性质，边的数目 $m = n - k$ 。
 $k \geq 1$ ，所以： $m < n$)

如果 $m < n$ 则按照上面的算法每删除一个度为 0 的顶点操作一次（最多 n 次），或每

删除一个度为 1 的顶点（同时删一条边）操作一次（最多 m 次）。这两种操作的总数不会超过 $m+n$ 。由于 $m < n$ ，所以算法复杂度为 $O(n)$

另：

该方法，算法复杂度不止 $O(V)$ ，首先初始时刻统计所有顶点的度的时候，复杂度为 $(V + E)$ ，即使在后来的循环中 $E \geq V$ ，这样算法的复杂度也只能为 $O(V + E)$ 。其次，在每次循环时，删除度为 1 的顶点，那么就必须将与这个顶点相连的点的度减一，并且执行 `delete node from list[list[node]]`，这里查找的复杂度为 `list[list[node]]` 的长度，只有这样才能保证当 $\text{degree}[i] = 1$ 时，`list[i]` 里面只有一个点。这样最差的复杂度就为 $O(EV)$ 了。

法 2：

DFS 搜索图，图中的边只可能是树边或反向边，一旦发现反向边，则表明存在环。该算法的复杂度为 $O(V)$ 。

有向图：

主要有深度优先和拓扑排序 2 中方法

1、拓扑排序，如果能够用拓扑排序完成对图中所有节点的排序的话，就说明这个图中没有环，而如果不能完成，则说明有环。

2、可以用 Strongly Connected Components 来做，我们可以回忆一下强连通子图的概念，就是说对于一个图的某个子图，该子图中的任意 $u \rightarrow v$ ，必有 $v \rightarrow u$ ，则这是一个强连通子图。这个限定正好是环的概念。所以我想，通过寻找图的强连通子图的方法应该可以找出一个图中到底有没有环、有几个环。

3、就是用一个改进的 DFS

刚看到这个问题的时候，我想单纯用 DFS 就可以解决问题了。但细想一下，是不能够的。如果题目给出的是一个无向图，那么 OK，DFS 是可以解决的。但无向图得不出正确结果的。比如：A->B, A->C->B，我们用 DFS 来处理这个图，我们会得出它有环，但其实没有。

我们可以对 DFS 稍加变化，来解决这个问题。解决的方法如下：

图中的一个节点，根据其 $C[N]$ 的值，有三种状态：

0，此节点没有被访问过

-1，被访问过至少 1 次，其后代节点正在被访问中

1，其后代节点都被访问过。

按照这样的假设，当按照 DFS 进行搜索时，碰到一个节点时有三种可能：

1、如果 $C[V]=0$ ，这是一个新的节点，不做处理

2、如果 $C[V]=-1$ ，说明是在访问该节点的后代的过程中访问到该节点本身，则图中有环。

3、如果 $C[V]=1$, 类似于 2 的推导, 没有环。 在程序中加上一些特殊的处理, 即可以找出图中有几个环, 并记录每个环的路径

判断有向图是否有环有三种方法: 拓扑排序、深度遍历+回溯、深度遍历 + 判断后退边

这里使用 拓扑排序 和 深度遍历 + 回溯判断是不是环。使用 深度遍历 + 判断后退边找出环个数 以及环中元素

1、拓扑排序

思想: 找入度为 0 的顶点, 输出顶点, 删除出边。循环到无顶点输出。

若: 输出所有顶点, 则课拓扑排序, 无环; 反之, 则不能拓扑排序, 有环

使用: 可以使用拓扑排序为有向无环图每一个结点进行编号, 拓扑排序输出的顺序可以为编号顺序

源代码:

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
5. class MGraph
6. {
7. public:
8.     void CreateGraph();//创建图
9.     int LocateVex(VexType v);//返回顶点 v 所在顶点向量中的位置 (下标)
10.    void CheckCircle();
11. private:
12.     VexType vexs[MAX_Vertex_Num];//顶点向量
13.     ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示, 主要是
14.         //为了处理有权值的情况, 比如: 权值可以为小数 (代价), 也可以为整数
15.     int vexnum;//顶点数
16.     int arcnum;//边数
17.     bool TopSort();
18. };
19.
20. template<class VexType,class ArcType>
```

```
21. void MGraph<VexType,ArcType>::CreateGraph()
22. {
23.     VexType first;
24.     VexType Secend;
25.     cout<<"请输入顶点数:";
26.     cin>>vexnum;
27.     cout<<"请输入边数:";
28.     cin>>arcnum;
29.     cout<<"请输入各个顶点值: ";
30.     for (int i=0;i<vexnum;i++)
31.     {
32.         cin>>vexs[i];
33.     }
34. //初始化邻接矩阵
35.     for (int i=0;i<arcnum;i++)
36.     {
37.         for (int j=0;j<arcnum;j++)
38.         {
39.             arcs[i][j]=0;
40.         }
41.     }
42.     cout<<"请输入边的信息:"<<endl;
43.     for (int i=0;i<arcnum;i++)
44.     {
45.         cin>>first>>Secend;
46.         //如果边有权值的话，则还应该输入权值
47.         int x = LocateVex(first);
48.         int y = LocateVex(Secend);
49.         arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
50.     }
51. }
52. /*
53. 参数: v: 表示顶点向量中一个值
54. 函数返回值: 函数返回 v 在顶点向量中的下标
55. */
```

```

56. template<class VexType,class ArcType>
57. int MGraph<VexType,ArcType>::LocateVex(VexType v)
58. {
59.     for (int i=0;i<vexnum;i++)
60.     {
61.         if (vexs[i]==v)
62.         {
63.             return i;
64.         }
65.     }
66.     return -1;
67. }
68. /*
69. 有向图可以拓扑排序的条件是：图中没有环。
70. 具体方法：
71. (1) 从图中选择一个入度为 0 的点加入拓扑序列。
72. (2) 从图中删除该结点以及它的所有出边（即与之相邻点入度减 1）。
73.
74. */
75. template<class VexType,class ArcType>
76. bool MGraph<VexType,ArcType>::TopSort()
77. {
78.     int count = 0;//拓扑排序输出顶点的个数
79.     int top = -1;
80.     int stack[MAX_Vertex_Num];
81.     int indegree[MAX_Vertex_Num]={0};
82.     //求各个顶点的入度--邻接矩阵要查询该元素的列（记录入度情况）--
83.     //如果是邻接表，就是麻烦在这里，查询结点入度很不方便
84.     for (int i=0;i<vexnum;i++)
85.     {
86.         int num=0;
87.         for (int j=0;j<vexnum;j++)
88.         {
89.             if (arcs[j][i]!=0)
90.             {

```

```

91.         num++;
92.     }
93. }
94. indegree[i]=num;
95. }
96. //把入度为 0 的顶点入栈
97. for (int i=0;i<vexnum;i++)
98. {
99.     if (!indegree[i])
100.    {
101.        stack[++top]=i;//顶点的下标
102.    }
103. }
104. //处理入度为 0 的结点：把入度为 0 的结点出栈，删除与之有关的边
105. while (top>-1)
106. {
107.     int x = stack[top--];
108.     cout<<vexs[x];
109.     count++;
110.     //把与下标为 x 的顶点有关的边都去掉（出边）,并改变对应结点的入度
111.     for (int i=0;i<vexnum;i++)
112.     {
113.         if (arcs[x][i]!=0)
114.         {
115.             arcs[x][i]=0;//删除到下标为 i 的顶点的边，这时此顶点的入度减一
116.             indegree[i]--;
117.             if (!indegree[i])//顶点的入度为 0，则入栈
118.             {
119.                 stack[++top]=i;
120.             }
121.         }
122.     }
123. }
124. cout<<endl;
125. if (count == vexnum) //能拓扑排序

```

```
126.  {
127.      return true;
128.  }
129.  return false;
130. }
131. /*
132. 检查图中是不是有环
133. 思想:
134. 能进行拓扑排序, 则无环, 反之有环
135. */
136. template<class VexType,class ArcType>
137. void MGraph<VexType,ArcType>::CheckCircle()
138. {
139.     if (TopSort())
140.     {
141.         cout<<"无环! "<<endl;
142.     }
143.     else
144.     {
145.         cout<<"有环! "<<endl;
146.     }
147. }
148.
149. int main()
150. {
151.     MGraph<char,int> G;
152.     G.CreateGraph();
153.     G.CheckCircle();
154.     system("pause");
155.     return 1;
156. }
```

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
```

```
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
5. class MGraph
6. {
7. public:
8.     void CreateGraph();//创建图
9.     int LocateVex(VexType v);//返回顶点 v 所在顶点向量中的位置（下标）
10.    void CheckCircle();
11. private:
12.     VexType vexs[MAX_Vertex_Num];//顶点向量
13.     ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示，主要是
为了处理有权值的情况，比如：权值可以为小数（代价），也可以为整数
14.     int vexnum;//顶点数
15.     int arcnum;//边数
16. private:
17.     bool TopSort();
18. };
19.
20. template<class VexType,class ArcType>
21. void MGraph<VexType,ArcType>::CreateGraph()
22. {
23.     VexType first;
24.     VexType Secend;
25.     cout<<"请输入顶点数:";
26.     cin>>vexnum;
27.     cout<<"请输入边数:";
28.     cin>>arcnum;
29.     cout<<"请输入各个顶点值： ";
30.     for (int i=0;i<vexnum;i++)
31.     {
32.         cin>>vexs[i];
33.     }
34.     //初始化邻接矩阵
35.     for (int i=0;i<arcnum;i++)
36.     {
```

```

37.     for (int j=0;j<arcnum;j++)
38.     {
39.         arcs[i][j]=0;
40.     }
41. }
42. cout<<"请输入边的信息:"<<endl;
43. for (int i=0;i<arcnum;i++)
44. {
45.     cin>>first>>Secend;
46.     //如果边有权值的话，则还应该输入权值
47.     int x = LocateVex(first);
48.     int y = LocateVex(Secend);
49.     arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
50. }
51. }
52. /*
53. 参数：v：表示顶点向量中一个值
54. 函数返回值：函数返回 v 在顶点向量中的下标
55. */
56. template<class VexType,class ArcType>
57. int MGraph<VexType,ArcType>::LocateVex(VexType v)
58. {
59.     for (int i=0;i<vexnum;i++)
60.     {
61.         if (vexs[i]==v)
62.         {
63.             return i;
64.         }
65.     }
66.     return -1;
67. }
68. /*
69. 有向图可以拓扑排序的条件是：图中没有环。
70. 具体方法：
71. (1) 从图中选择一个入度为 0 的点加入拓扑序列。

```

```
72. (2) 从图中删除该结点以及它的所有出边（即与之相邻点入度减 1）。
73.
74. */
75. template<class VexType,class ArcType>
76. bool MGraph<VexType,ArcType>::TopSort()
77. {
78.     int count = 0;//拓扑排序输出顶点的个数
79.     int top = -1;
80.     int stack[MAX_Vertex_Num];
81.     int indegree[MAX_Vertex_Num]={0};
82.     //求各个顶点的入度--邻接矩阵要查询该元素的列（记录入度情况）--
83.     //如果是邻接表，就是麻烦在这里，查询结点入度很不方便
84.     for (int i=0;i<vexnum;i++)
85.     {
86.         int num=0;
87.         for (int j=0;j<vexnum;j++)
88.         {
89.             if (arcs[j][i]!=0)
90.             {
91.                 num++;
92.             }
93.         }
94.         indegree[i]=num;
95.     }
96.     //把入度为 0 的顶点入栈
97.     for (int i=0;i<vexnum;i++)
98.     {
99.         if (!indegree[i])
100.         {
101.             stack[++top]=i;//顶点的下标
102.         }
103.     }
104.     //处理入度为 0 的结点：把入度为 0 的结点出栈，删除与之有关的边
105.     while (top>-1)
106.     {
```

```

107.     int x = stack[top--];
108.     cout<<vexs[x];
109.     count++;
110.     //把与下标为 x 的顶点有关的边都去掉（出边）,并改变对应结点的入度
111.     for (int i=0;i<vexnum;i++)
112.     {
113.         if (arcs[x][i]!=0)
114.         {
115.             arcs[x][i]=0;//删除到下标为 i 的顶点的边，这时此顶点的入度减一
116.             indegree[i]--;
117.             if (!indegree[i])//顶点的入度为 0，则入栈
118.             {
119.                 stack[++top]=i;
120.             }
121.         }
122.     }
123. }
124. cout<<endl;
125. if (count == vexnum) //能拓扑排序
126. {
127.     return true;
128. }
129. return false;
130. }
131. /*
132. 检查图中是不是有环
133. 思想：
134. 能进行拓扑排序，则无环，反之有环
135. */
136. template<class VexType,class ArcType>
137. void MGraph<VexType,ArcType>::CheckCircle()
138. {
139.     if (TopSort())
140.     {
141.         cout<<"无环！ "<<endl;

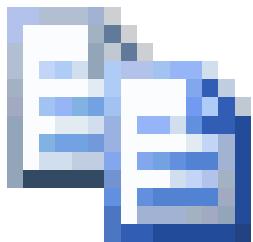
```

```
142. }
143. else
144. {
145.     cout<<"有环！ "<<endl;
146. }
147. }
148.
149. int main()
150. {
151.     MGraph<char,int> G;
152.     G.CreateGraph();
153.     G.CheckCircle();
154.     system("pause");
155.     return 1;
156. }
```

测试：

有向图：

结果：



2、深度遍历 + 回溯

思想：用回溯法，遍历时，如果遇到了之前访问过的结点，则图中存在环。

代码：

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
```

```
5. class MGraph
6. {
7.     public:
8.     void CreateGraph(); //创建图
9.     int LocateVex(VexType v); //返回顶点 v 所在顶点向量中的位置 (下标)
10.    bool CheckCircle(); //检查图中有无环
11.    private:
12.    VexType vexs[MAX_Vertex_Num]; //顶点向量
13.    ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示,主要是
为了处理有权值的情况,比如: 权值可以为小数(代价),也可以为整数
14.    int vexnum; //顶点数
15.    int arcnum; //边数
16.    private:
17.    void CheckCircle(int u, bool& isExist, bool visited[MAX_Vertex_Num], bool Isvisited[MAX_Vertex_
Num]);
18. };
19.
20. template<class VexType, class ArcType>
21. void MGraph<VexType, ArcType>::CreateGraph()
22. {
23.     VexType first;
24.     VexType Secend;
25.     cout << "请输入顶点数:" ;
26.     cin >> vexnum;
27.     cout << "请输入边数:" ;
28.     cin >> arcnum;
29.     cout << "请输入各个顶点值: " ;
30.     for (int i=0; i<vexnum; i++)
31.     {
32.         cin >> vexs[i];
33.     }
34.     //初始化邻接矩阵
35.     for (int i=0; i<arcnum; i++)
36.     {
37.         for (int j=0; j<arcnum; j++)
```

```

38.     {
39.         arcs[i][j]=0;
40.     }
41. }
42. cout<<"请输入边的信息:"<<endl;
43. for (int i=0;i<arcnum;i++)
44. {
45.     cin>>first>>Secend;
46.     //如果边有权值的话，则还应该输入权值
47.     int x = LocateVex(first);
48.     int y = LocateVex(Secend);
49.     arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
50. }
51. }
52. /*
53. 参数：v：表示顶点向量中一个值
54. 函数返回值：函数返回 v 在顶点向量中的下标
55. */
56. template<class VexType,class ArcType>
57. int MGraph<VexType,ArcType>::LocateVex(VexType v)
58. {
59.     for (int i=0;i<vexnum;i++)
60.     {
61.         if (vexs[i]==v)
62.         {
63.             return i;
64.         }
65.     }
66.     return -1;
67. }
68.
69. /*
70. 思想：用回溯法遍历时，如果遇到了之前访问过的结点，则图中存在环。
71. */
72. template<class VexType,class ArcType>

```

```

73. void MGraph<VexType,ArcType>::CheckCircle(int u,bool& isExist,bool visited[MAX_Vertex_Num],bool Isvisited[MAX_Vertex_Num])
74. {
75.     visited[u]=true;
76.     Isvisited[u]=true;
77.     for (int j=0;j<vexnum;j++)
78.     {
79.         if (arcs[u][j]==1)
80.         {
81.             if (visited[j]==false)
82.             {
83.                 CheckCircle(j,isExist,visited,Isvisited);
84.             }
85.             else
86.             {
87.                 isExist = true;
88.             }
89.         }
90.     }
91.     visited[u]=false;//回溯，如果不写就变成一半的深度遍历，不能进行判断是否有边存在
92. }
93.
94. template<class VexType,class ArcType>
95. bool MGraph<VexType,ArcType>::CheckCircle()
96. {
97.     bool isExist = false;
98.     bool Isvisited[MAX_Vertex_Num]={ false };
99.     bool visited[MAX_Vertex_Num]={ false };
100.    for (int i=0;i<vexnum;i++)
101.    {
102.        if (Isvisited[i]==false)
103.        {
104.            CheckCircle(i,isExist,visited,Isvisited);
105.            if (isExist)
106.            {

```

```
107.         return true;
108.     }
109. }
110. }
111. return isExist;
112. }
113.
114. int main()
115. {
116.     MGraph<char,int> G;
117.     G.CreateGraph();
118.     if (G.CheckCircle())
119.     {
120.         cout<<"图存在环！ "<<endl;
121.     }
122.     else
123.     {
124.         cout<<"图不存在环！ "<<endl;
125.     }
126.     system("pause");
127.     return 1;
128. }
```

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
5. class MGraph
6. {
7. public:
8.     void CreateGraph();//创建图
9.     int LocateVex(VexType v);//返回顶点 v 所在顶点向量中的位置（下标）
10.    bool CheckCircle();//检查图中有无环
11. private:
```

```
12.     VexType vexs[MAX_Vertex_Num];//顶点向量
13.     ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示,主要是
为了处理有权值的情况,比如:权值可以为小数(代价),也可以为整数
14.     int vexnum;//顶点数
15.     int arcnum;//边数
16. private:
17.     void CheckCircle(int u,bool& isExist,bool visited[MAX_Vertex_Num],bool Isvisited[MAX_Vertex_
Num]);
18. };
19.
20. template<class VexType,class ArcType>
21. void MGraph<VexType,ArcType>::CreateGraph()
22. {
23.     VexType first;
24.     VexType Secend;
25.     cout<<"请输入顶点数:";
26.     cin>>vexnum;
27.     cout<<"请输入边数:";
28.     cin>>arcnum;
29.     cout<<"请输入各个顶点值: ";
30.     for (int i=0;i<vexnum;i++)
31.     {
32.         cin>>vexs[i];
33.     }
34.     //初始化邻接矩阵
35.     for (int i=0;i<arcnum;i++)
36.     {
37.         for (int j=0;j<arcnum;j++)
38.         {
39.             arcs[i][j]=0;
40.         }
41.     }
42.     cout<<"请输入边的信息:"<<endl;
43.     for (int i=0;i<arcnum;i++)
44.     {
```

```

45.     cin>>first>>Secend;
46.     //如果边有权值的话，则还应该输入权值
47.     int x = LocateVex(first);
48.     int y = LocateVex(Secend);
49.     arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
50.   }
51. }
52. /*
53. 参数：v：表示顶点向量中一个值
54. 函数返回值：函数返回 v 在顶点向量中的下标
55. */
56. template<class VexType,class ArcType>
57. int MGraph<VexType,ArcType>::LocateVex(VexType v)
58. {
59.     for (int i=0;i<vexnum;i++)
60.     {
61.         if (vexs[i]==v)
62.         {
63.             return i;
64.         }
65.     }
66.     return -1;
67. }
68.
69. /*
70. 思想：用回溯法，遍历时，如果遇到了之前访问过的结点，则图中存在环。
71. */
72. template<class VexType,class ArcType>
73. void MGraph<VexType,ArcType>::CheckCircle(int u,bool& isExist,bool visited[MAX_Vertex_Num],bool Isvisited[MAX_Vertex_Num])
74. {
75.     visited[u]=true;
76.     Isvisited[u]=true;
77.     for (int j=0;j<vexnum;j++)
78.     {

```

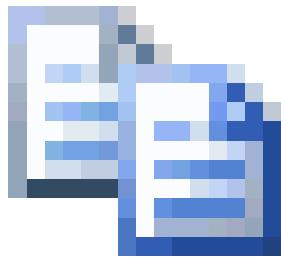
```
79.     if (arcs[u][j]==1)
80.     {
81.         if (visited[j]==false)
82.         {
83.             CheckCircle(j,isExist,visited,Isvisited);
84.         }
85.     else
86.     {
87.         isExist = true;
88.     }
89. }
90. }
91. visited[u]=false;//回溯，如果不写就变成一半的深度遍历，不能进行判断是否有边存在
92. }
93.
94. template<class VexType,class ArcType>
95. bool MGraph<VexType,ArcType>::CheckCircle()
96. {
97.     bool isExist = false;
98.     bool Isvisited[MAX_Vertex_Num]={false};
99.     bool visited[MAX_Vertex_Num]={false};
100.    for (int i=0;i<vexnum;i++)
101.    {
102.        if (Isvisited[i]==false)
103.        {
104.            CheckCircle(i,isExist,visited,Isvisited);
105.            if (isExist)
106.            {
107.                return true;
108.            }
109.        }
110.    }
111.    return isExist;
112. }
113.
```

```
114. int main()
115. {
116.     MGraph<char,int> G;
117.     G.CreateGraph();
118.     if (G.CheckCircle())
119.     {
120.         cout<<"图存在环！ "<<endl;
121.     }
122.     else
123.     {
124.         cout<<"图不存在环！ "<<endl;
125.     }
126.     system("pause");
127.     return 1;
128. }
```

结果测试：

图：

结果：



3、深度遍历 + 判断后退边

思想：用 DFS（深度优先遍历）,判断是否有后退边，若有，则存在环

具体来说，在遍历顶点的每一条边时，判断一下这个边的顶点是不是在栈中，如果在栈中，说明之前已经访问过了，这里再次访问，说明有环存在

判断后退边时，借助一个栈和一个数组

栈：即可以用来输出环

数组：inStack 判断是否在栈中

源代码：

[cpp] [view](#) [plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
5. class MGraph
6. {
7. public:
8.     void CreateGraph();//创建图
9.     int LocateVex(VexType v);//返回顶点 v 所在顶点向量中的位置（下标）
10.    void CheckCircle();
11. private:
12.     VexType vexs[MAX_Vertex_Num];//顶点向量
13.     ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示，主要是为了处理有权值的情况，比如：权值可以为小数（代价），也可以为整数
14.     int vexnum;//顶点数
15.     int arcnum;//边数
16. private:
17.     void DFS(int x,bool visited[MAX_Vertex_Num],int stack[MAX_Vertex_Num],int& top,bool inStack[MAX_Vertex_Num],int& count);
18.
19. };
20.
21. template<class VexType,class ArcType>
22. void MGraph<VexType,ArcType>::CreateGraph()
23. {
24.     VexType first;
25.     VexType Secend;
26.     cout<<"请输入顶点数:";
27.     cin>>vexnum;
28.     cout<<"请输入边数:";
29.     cin>>arcnum;
30.     cout<<"请输入各个顶点值： ";
31.     for (int i=0;i<vexnum;i++)
32.     {
```

```
33.     cin>>vexs[i];
34. }
35. //初始化邻接矩阵
36. for (int i=0;i<arcnum;i++)
37. {
38.     for (int j=0;j<arcnum;j++)
39.     {
40.         arcs[i][j]=0;
41.     }
42. }
43. cout<<"请输入边的信息:"<<endl;
44. for (int i=0;i<arcnum;i++)
45. {
46.     cin>>first>>Secend;
47.     //如果边有权值的话，则还应该输入权值
48.     int x = LocateVex(first);
49.     int y = LocateVex(Secend);
50.     arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
51. }
52. /*
53. 参数： v： 表示顶点向量中一个值
54. 函数返回值： 函数返回 v 在顶点向量中的下标
55. */
56. */
57. template<class VexType,class ArcType>
58. int MGraph<VexType,ArcType>::LocateVex(VexType v)
59. {
60.     for (int i=0;i<vexnum;i++)
61.     {
62.         if (vexs[i]==v)
63.         {
64.             return i;
65.         }
66.     }
67.     return -1;
```

```

68. }
69.
70. /*
71. 检查图中是不是有回向边
72. 思想:
73. 如果有回向边, 则无环, 反之有环
74. */
75. template<class VexType,class ArcType>
76. void MGraph<VexType,ArcType>::CheckCircle()
77. {
78.     int count=0;//环的个数
79.     int top=-1;
80.     int stack[MAX_Vertex_Num];
81.     bool inStack[MAX_Vertex_Num]={false};
82.     bool visited[MAX_Vertex_Num]={false};
83.     for (int i=0;i<vexnum;i++)
84.     {
85.         if (!visited[i])
86.         {
87.             DFS(i,visited,stack,top,inStack,count);
88.         }
89.     }
90. }
91.
92. template<class VexType,class ArcType>
93. void MGraph<VexType,ArcType>::DFS(int x,bool visited[MAX_Vertex_Num],int stack[MAX_Vertex_
Num],int& top,bool inStack[MAX_Vertex_Num],int& count)
94. {
95.     visited[x]=true;
96.     stack[++top]=x;
97.     inStack[x]=true;
98.     for (int i=0;i<vexnum;i++)
99.     {
100.         if (arcs[x][i]!=0)//有边
101.         {

```

```

102.     if (!inStack[i])
103.     {
104.         DFS(i,visited,stack,top,inStack,count);
105.     }
106.     else //条件成立，表示下标为 x 的顶点到 下标为 i 的顶点有环
107.     {
108.         count++;
109.         cout<<"第"<<count<<"环为:";
110.         //从 i 到 x 是一个环， top 的位置是 x， 下标为 i 的顶点在栈中的位置要寻找一下
111.         //寻找起始顶点下标在栈中的位置
112.         int t=0;
113.         for (t=top;stack[t]!=i;t--);
114.         //输出环中顶点
115.         for (int j=t;j<=top;j++)
116.         {
117.             cout<<vexs[stack[j]];
118.         }
119.         cout<<endl;
120.     }
121. }
122. }
123. //处理完结点后，退栈
124. top--;
125. inStack[x]=false;
126. }
127. int main()
128. {
129.     MGraph<char,int> G;
130.     G.CreateGraph();
131.     G.CheckCircle();
132.     system("pause");
133.     return 1;
134. }

```

[cpp] [view plain](#) [copy](#) [print](#)?

```
1. #include <iostream>
2. using namespace std;
3. const int MAX_Vertex_Num = 20;
4. template<class VexType,class ArcType>
5. class MGraph
6. {
7. public:
8.     void CreateGraph();//创建图
9.     int LocateVex(VexType v);//返回顶点 v 所在顶点向量中的位置（下标）
10.    void CheckCircle();
11. private:
12.     VexType vexs[MAX_Vertex_Num];//顶点向量
13.     ArcType arcs[MAX_Vertex_Num][MAX_Vertex_Num]; //这里把邻接矩阵类型用模板表示，主要是
为了处理有权值的情况，比如：权值可以为小数（代价），也可以为整数
14.     int vexnum;//顶点数
15.     int arcnum;//边数
16. private:
17.     void DFS(int x,bool visited[MAX_Vertex_Num],int stack[MAX_Vertex_Num],int& top,bool inStack
[MAX_Vertex_Num],int& count);
18.
19. };
20.
21. template<class VexType,class ArcType>
22. void MGraph<VexType,ArcType>::CreateGraph()
23. {
24.     VexType first;
25.     VexType Secend;
26.     cout<<"请输入顶点数:";
27.     cin>>vexnum;
28.     cout<<"请输入边数:";
29.     cin>>arcnum;
30.     cout<<"请输入各个顶点值： ";
31.     for (int i=0;i<vexnum;i++)
32.     {
33.         cin>>vexs[i];
```

```
34.     }
35.     //初始化邻接矩阵
36.     for (int i=0;i<arcnum;i++)
37.     {
38.         for (int j=0;j<arcnum;j++)
39.         {
40.             arcs[i][j]=0;
41.         }
42.     }
43.     cout<<"请输入边的信息:"<<endl;
44.     for (int i=0;i<arcnum;i++)
45.     {
46.         cin>>first>>Secend;
47.         //如果边有权值的话，则还应该输入权值
48.         int x = LocateVex(first);
49.         int y = LocateVex(Secend);
50.         arcs[x][y]=1;//如果是有权的话，这里应该是 arc[x][y]=权值
51.     }
52. }
53. /*
54. 参数：v： 表示顶点向量中一个值
55. 函数返回值：函数返回 v 在顶点向量中的下标
56. */
57. template<class VexType,class ArcType>
58. int MGraph<VexType,ArcType>::LocateVex(VexType v)
59. {
60.     for (int i=0;i<vexnum;i++)
61.     {
62.         if (vexs[i]==v)
63.         {
64.             return i;
65.         }
66.     }
67.     return -1;
68. }
```

```

69.
70. /*
71. 检查图中是不是有回向边
72. 思想:
73. 如果有回向边, 则无环, 反之有环
74. */
75. template<class VexType,class ArcType>
76. void MGraph<VexType,ArcType>::CheckCircle()
77. {
78.     int count=0;//环的个数
79.     int top=-1;
80.     int stack[MAX_Vertex_Num];
81.     bool inStack[MAX_Vertex_Num]={ false };
82.     bool visited[MAX_Vertex_Num]={ false };
83.     for (int i=0;i<vexnum;i++)
84.     {
85.         if (!visited[i])
86.         {
87.             DFS(i,visited,stack,top,inStack,count);
88.         }
89.     }
90. }
91.
92. template<class VexType,class ArcType>
93. void MGraph<VexType,ArcType>::DFS(int x,bool visited[MAX_Vertex_Num],int stack[MAX_Vertex_
    Num],int& top,bool inStack[MAX_Vertex_Num],int& count)
94. {
95.     visited[x]=true;
96.     stack[++top]=x;
97.     inStack[ x ]=true;
98.     for (int i=0;i<vexnum;i++)
99.     {
100.         if (arcs[ x ][i]!=0)//有边
101.         {
102.             if (!inStack[ i ])

```

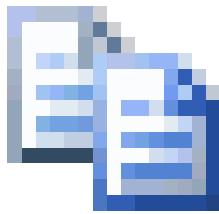
```

103.     {
104.         DFS(i,visited,stack,top,inStack,count);
105.     }
106.     else //条件成立， 表示下标为 x 的顶点到下标为 i 的顶点有环
107.     {
108.         count++;
109.         cout<<"第"<<count<<"环为:";
110.         //从 i 到 x 是一个环， top 的位置是 x， 下标为 i 的顶点在栈中的位置要寻找一下
111.         //寻找起始顶点下标在栈中的位置
112.         int t=0;
113.         for (t=top;stack[t]!=i;t--);
114.         //输出环中顶点
115.         for (int j=t;j<=top;j++)
116.         {
117.             cout<<vexs[stack[j]];
118.         }
119.         cout<<endl;
120.     }
121. }
122. }
123. //处理完结点后，退栈
124. top--;
125. inStack[x]=false;
126. }
127. int main()
128. {
129.     MGraph<char,int> G;
130.     G.CreateGraph();
131.     G.CheckCircle();
132.     system("pause");
133.     return 1;
134. }
```

结果测试：

有向图：

结果：



```
1. //求图中环的个数
2. //由于图中每个点的出度只有 1， 所以不存在一个点处于两个环的交点
3. //因此,求环的个数时每个只需要考虑一次便可得出结果
4. //由于数据规模庞大， 写成递归形式容易暴栈
5. //在读边的过程中先对自环进行预处理，之后对每个点进行不同的染色，对它的下一个点也染同样的颜色
6. //这样染下去如果发现下一个要染的点和正在染的颜色相同，则说明存在一个环
7. //换染色起点的同时也需要更换新的染色，才能保证对环的判断正确
8. #include<iostream>
9. #include<cstring>
10. using namespace std;
11. int next[1000001];//指向下一结点的指针
12. int vis[1000001];//对每个结点进行不同的标记
13. int ans,n,ringID,p;
14. void search()
15. {
16.     for(int i = 1;i <= n;++i)
17.     {
18.         if(vis[i] > 0) continue;
19.         p = i;
20.         ++ringID;//对每一种环进行一种不同的标记， 新的起点必须更换新的染色
21.         while(vis[p] == 0)//当前结点未被染色
22.         {
23.             vis[p] = ringID;//染色
24.             p = next[p];//指向下一个点
25.             if(vis[p] == ringID)//下一个点的颜色和当前染色相同，则说明存在一个环
```

```

26.         ++ans;
27.     }
28. }
29. }

30. int main()
31. {
32.     //freopen("in.txt","r",stdin);
33.     while(scanf("%d",&n) != EOF)
34.     {
35.         ans = 0;
36.         ringID = 1;
37.         memset(vis,0,sizeof(vis));
38.         for(int i = 1;i <= n;++i)
39.         {
40.             scanf("%d",&next[i]);
41.             if(next[i] == i)
42.             {
43.                 vis[i] = ringID++;//先对自环进行预处理
44.                 ans++;
45.             }
46.         }
47.         search();
48.         printf("%d/n",ans);
49.     }
50.     return 0;
51. }

```

1.15.2. 整数的素数和分解问题

【问题描述】

歌德巴赫猜想说任何一个不小于 6 的偶数都可以分解为两个奇素数之和。对此问题扩展，如果一个整数能够表示成两个或多个素数之和，则得到一个素数和分解式。对于一个给定的整数，输出所有这种素数和分解式。注意，对于同构的分解只输出一次（比如 5 只有一个分解 $2 + 3$ ，而 $3 + 2$ 是 $2 + 3$ 的同构分解式）。

例如，对于整数 8，可以作为如下三种分解：

- (1) $8 = 2 + 2 + 2 + 2$
- (2) $8 = 2 + 3 + 3$
- (3) $8 = 3 + 5$

【算法分析】

由于要将指定整数 N 分解为素数之和，则首先需要计算出该整数 N 内的所有素数，然后递归求解所有素数和分解即可。

C++代码实现如下：

```
#include <iostream>
#include <vector>
#include <iterator>
#include <cmath>
using namespace std;

// 计算 num 内的所有素数（不包括 num）
void CalcPrimes(int num, vector<int> &primes)
{
    primes.clear();
    if (num <= 2)
        return;

    primes.push_back(2);
    for (int i = 3; i < num; i += 2) {
        int root = int(sqrt(i));
        int j = 2;
        for (j = 2; j <= root; ++j) {
            if (i % j == 0)
                break;
        }
        if (j > root)
            primes.push_back(i);
    }
}

// 输出所有素数组合(递归实现)
int PrintCombinations(int num, const vector<int> &primes, int from, vector<int> &numbers)
{
    if (num == 0) {
        cout << "Found: ";
        copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));
        cout << '\n';
    }
}
```

```

        return 1;
    }

    int count = 0;

    // 从第 from 个素数搜索，从而避免输出同构的多个组合
    int primesNum = primes.size();
    for (int i = from; i < primesNum; ++i) {
        if (num < primes[i])
            break;
        numbers.push_back(primes[i]);
        count += PrintCombinations(num - primes[i], primes, i, numbers);
        numbers.pop_back();
    }

    return count;
}

// 计算 num 的所有素数和分解
int ExpandedGoldbach(int num)
{
    if (num <= 3)
        return 0;

    vector<int> primes;
    CalcPrimes(num, primes);

    vector<int> numbers;
    return PrintCombinations(num, primes, 0, numbers);
}

int main()
{
    for (int i = 1; i <= 20; ++i) {
        cout << "When i = " << i << ":\n";
        int count = ExpandedGoldbach(i);
        cout << "Total: " << count << "\n\n";
    }
}

```

运行结果：

When i = 1:

Total: 0

When i = 2:

Total: 0

When i = 3:

Total: 0

1.15.3. 求两个或 N 个数的最大公约数(gcd)和最小公倍数(lcm)的较优算法

//求两个或 N 个数的最大公约数(gcd)和最小公倍数(lcm)的较优算法

//两个数的最大公约数--欧几里得算法

```
int gcd(int a, int b)
{
    if (a < b)
        swap(a, b);
    if (b == 0)
        return a;
    else
        return gcd(b, a%b);
}
```

//n 个数的最大公约数算法

//说明:

//把 n 个数保存为一个数组

//参数为数组的指针和数组的大小(需要计算的数的个数)

//然后先求出 gcd(a[0],a[1]), 然后将所求的 gcd 与数组的下一个元素作为 gcd 的参数继续求

gcd

//这样就产生一个递归的求 ngcd 的算法

```
int ngcd(int *a, int n)
{
    if (n == 1)
        return *a;
    return gcd(a[n-1], ngcd(a, n-1));
}
```

//两个数的最小公倍数(lcm)算法

```

//lcm(a, b) = a*b/gcd(a, b)
int lcm(int a, int b)
{
    return a*b/gcd(a, b);
}

//n 个数的最小公倍数算法
//算法过程和 n 个数的最大公约数求法类似
//求出头两个的最小公倍数,再将欺和大三个数求最小公倍数直到数组末尾
//这样产生一个递归的求 nlcm 的算法
int nlcm(int *a, int n)
{
    if (n == 1)
        return *a;
    else
        return lcm(a[n-1], nlcm(a, n-1));
}

```

1.16.面试题集合（十五）

1.16.1. ApplicationContext

ApplicationContext 的 BeanFactory 的子类， 拥有更强大的功能， ApplicationContext 可以在服务器启动的时候自动实例化所有的 bean, 而 BeanFactory 只有在调用 getBean() 的时候才去实例化那个 bean, 这也是我们为什么要得到一个 ApplicationContext 对象， 事实上 Spring2 相关的 web 应用默认使用的是 ApplicationContext 对象去实例化 bean， 换一句话说，在服务器启动的时候， Spring 容器就已经实例化好了一个 ApplicationContext 对象， 所以我们要在老的代码里尝试去获取这个对象。但是如何才能得到一个 ApplicationContext 对象呢？方法很多，最常用的办法就是用 ClassPathXmlApplicationContext, FileSystemClassPathXmlApplicationContext, FileSystemXmlApplicationContext 等对象去加载 Spring 配置文件，这样做也是可以，但是在加载 Spring 配置文件的时候，就会生成一个新的 ApplicaitonContext 对象而不是 Spring 容器帮我们生成的哪一个，这样就产生了冗余，所以我们在里不采用这种加载文件的方式，我们使用 ApplicationContextAware 让 Spring 容器传递自己生成的 ApplicationContext 给我们，然后我

们把这个 ApplicationContext 设置成一个类的静态变量，这样我们就随时都可以在老的代码里得到 Application 的对象了。

ApplicationContextHelper

[java] [view plain](#) [copy](#) [print](#)?

```
1. import org.springframework.beans.BeansException;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.ApplicationContextAware;
4. /**
5. *
6. * @author MinFei
7. */
8. public class ApplicationContextHelper implements ApplicationContextAware {
9.     private static ApplicationContext appCtx;
10.    /**
11.     * 此方法可以把 ApplicationContext 对象 inject 到当前类中作为一个静态成员变量。
12.     * @param applicationContext ApplicationContext 对象。
13.     * @throws BeansException
14.    */
15.    @Override
16.    public void setApplicationContext( ApplicationContext applicationContext ) throws BeansException
17.    {
18.        appCtx = applicationContext;
19.    }
20.    /**
21.     * 这是一个便利的方法，帮助我们快速得到一个 BEAN
22.     * @param beanName bean 的名字
23.     * @return 返回一个 bean 对象
24.    */
25.    public static Object getBean( String beanName ) {
26.        return appCtx.getBean( beanName );
27.    }
```

[java] [view plain](#) [copy](#) [print](#)?

```
1. import org.springframework.beans.BeansException;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.ApplicationContextAware;
4. /**
5. *
6. * @author MinFei
7. */
8. public class ApplicationContextHelper implements ApplicationContextAware {
9.     private static ApplicationContext appCtx;
10.    /**
11.     * 此方法可以把 ApplicationContext 对象 inject 到当前类中作为一个静态成员变量。
12.     * @param applicationContext ApplicationContext 对象.
13.     * @throws BeansException
14.     */
15.    @Override
16.    public void setApplicationContext( ApplicationContext applicationContext ) throws BeansException {
17.
18.        appCtx = applicationContext;
19.    }
20.    /**
21.     * 这是一个便利的方法，帮助我们快速得到一个 BEAN
22.     * @param beanName bean 的名字
23.     * @return 返回一个 bean 对象
24.     */
25.    public static Object getBean( String beanName ) {
26.        return appCtx.getBean( beanName );
27.    }
```

```
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
/**
 *
 * @author MinFei
 */
public class ApplicationContextHelper implements ApplicationContextAware {
    private static ApplicationContext appCtx;
    /**
     * 此方法可以把ApplicationContext对象inject到当前类中作为一个静态成员变量。
     * @param applicationContext ApplicationContext 对象。
     * @throws BeansException
     */
    @Override
    public void setApplicationContext( ApplicationContext applicationContext ) {
        appCtx = applicationContext;
    }
}
```

配置 ApplicationContextHelper

[c-sharp] [view plain](#) [copy](#) [print](#)?

```
1. <bean id="SpringApplicationContext" class="com.company.helper.ApplicationContextHelper"></bean>
```

[c-sharp] [view plain](#) [copy](#) [print](#)?

```
1. <bean id="SpringApplicationContext" class="com.company.helper.ApplicationContextHelper"></bean>
```

```
<bean id="SpringApplicationContext" class="com.company.helper.ApplicationContextHelper"></bean>
```

使用些列方法去得到一个 bean

[c-sharp] [view plain](#) [copy](#) [print](#)?

```
1. BeanExample beanExample= (BeanExample )ApplicationContextHelper.getBean( "beanExample" );
```

[c-sharp] [view plain](#) [copy](#) [print](#)?

```
1. BeanExample beanExample= (BeanExample )ApplicationContextHelper.getBean( "beanExample" );
```

```
BeanExample beanExample= (BeanExample )ApplicationContextHelper.getBean( "beanExample" );
```

这样我们在老代码里取得了一个 Spring 配置的对象， 然后我们就可以自由自在的在老代码里边享受 Spring 提供的功能。

1.编写一个 JavaBean 实现 ApplicationContextAware 方法

Test.java

```
public class Test implements ApplicationContextAware {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void setApplicationContext(ApplicationContext context)  
        throws BeansException {  
        Test test = (Test) context.getBean("test");  
        System.out.println("内部打印 :" + test.getName());  
    }  
}
```

如果实现了 ApplicationContextAware 接口，在 Bean 的实例化时会自动调用 setApplicationContext()方法

2.把这个 JavaBean 加入 Spring 的管理

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans  
    xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
<bean id="test" class="com.wj.spring.lesson3.applicationcontextaware.Test">
<property name="name" value="hello"></property>

</bean>
</beans>
```

3. 测试这个 JavaBean

TestBean.java

```
public class TestBean {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Test hb = (HelloBean)context.getBean("test");
    }
}
```

4. 查看控制台打印的结果

内部打印 : hello

从数据结果可以看出 `setApplicationContext()` 方法确实在 bean 的初始化过程中被调用了

1.16.2. ApplicationContext 事件传播

事件传播

ApplicationContext 基于 Observer 模式（`java.util` 包中有对应实现），提供了针对 Bean 的事件传播功能。通过 `Application.publishEvent` 方法，我们可以将事件通知系统内所有的 `ApplicationListener`。

事件传播的一个典型应用是，当 Bean 中的操作发生异常（如数据库连接失败），则通过事件传播机制通知异常监听器进行处理。在笔者的一个项目中，就曾经借助事件机制，较好的实现了当系统异常时在监视终端上报警，同时发送报警 SMS 至管理员手机的功能。

在目前版本的 Spring 中，事件传播部分的设计还有待改进。同时，如果能进一步支持异步事件处理机制，无疑会更具吸引力。

下面是一个简单的示例，当 LoginAction 执行的时候，激发一个自定义消息“ActionEvent”，此 ActionEvent 将由 ActionListener 捕获，并将事件内容打印到控制台。

LoginActoin.java:

```
public class LoginAction implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public void setApplicationContext(
        ApplicationContext applicationContext
    )
        throws BeansException {
        this.applicationContext = applicationContext;
    }

    public int login(String username, String password) {
        ActionEvent event = new ActionEvent(username);
        this.applicationContext.publishEvent(event);
        return 0;
    }
}
```

ActionEvent.java:

```
public class ActionEvent extends ApplicationEvent {

    public ActionEvent(Object source) {
        super(source);
    }
}
```

ActionListener.java:

```
public class ActionListener implements ApplicationListener {  
    public void onApplicationEvent(ApplicationEvent event) {  
        if (event instanceof ActionEvent) {  
            System.out.println(event.toString());  
        }  
    }  
}
```

配置非常简单：

```
<bean id="loginaction" class="net.xiaxin.beans.LoginAction"/>  
<bean id="listener" class="net.xiaxin.beans.ActionListener"/>
```

运行测试代码：

```
ApplicationContext ctx=new  
FileSystemXmlApplicationContext("bean.xml");  
LoginAction action =(LoginAction)ctx.getBean("action");  
action.login("Erica","mypass");
```

可以看到控制台输出：

```
net.xiaxin.beans.LoginEvent[source=Erica]
```

org.springframework.context.event.ApplicationEventMulticasterImpl 实现了事件传播机制，目前还相对简陋。

在运行期， ApplicationContext 会自动在当前的所有 Bean 中寻找 ApplicationListener 接口的实现，并将其作为事件接收对象。当 Application.publishEvent 方法调用时，所有的 ApplicationListener 接口实现都会被激发，每个 ApplicationListener 可根据事件的类型判断是否是自己需要处理的事件，如上面的 ActionListener 只处理 ActionEvent 事件。

1.16.3. mysql 有多种存储引擎

mysql 有多种存储引擎：MyISAM、InnoDB、MERGE、MEMORY(HEAP)、BDB(BerkeleyDB)、EXAMPLE、FEDERATED、ARCHIVE、CSV、BLACKHOLE。

MySQL 支持数个存储引擎作为对不同表的类型的处理器。MySQL 存储引擎包括处理事务安全表的引擎和处理非事务安全表的引擎：

- MyISAM 管理非事务表。它提供高速存储和检索，以及全文搜索能力。MyISAM 在所有 MySQL 配置里被支持，它是默认的存储引擎，除非你配置 MySQL 默认使用另外一个引擎。
- MEMORY 存储引擎提供“内存中”表。MERGE 存储引擎允许集合将被处理同样的 MyISAM 表作为一个单独的表。就像 MyISAM 一样，MEMORY 和 MERGE 存储引擎处理非事务表，这两个引擎也都被默认包含在 MySQL 中。

注释：MEMORY 存储引擎正式地被确定为 HEAP 引擎。

- InnoDB 和 BDB 存储引擎提供事务安全表。BDB 被包含在为支持它的操作系统发布的 MySQL-Max 二进制分发版里。InnoDB 也默认被包括在所有 MySQL 5.1 二进制分发版里，你可以按照喜好通过配置 MySQL 来允许或禁止任一引擎。
- EXAMPLE 存储引擎是一个“存根”引擎，它不做什么。你可以用这个引擎创建表，但没有数据被存储于其中或从其中检索。这个引擎的目的是服务，在 MySQL 源代码中的一个例子，它演示说明如何开始编写新存储引擎。同样，它的主要兴趣是对开发者。
- NDB Cluster 是被 MySQL Cluster 用来实现分割到多台计算机上的表的存储引擎。它在 MySQL-Max 5.1 二进制分发版里提供。这个存储引擎当前只被 Linux, Solaris, 和 Mac OS X 支持。在未来的 MySQL 分发版中，我们想要添加其它平台对这个引擎的支持，包括 Windows。

- ARCHIVE 存储引擎被用来无索引地，非常小地覆盖存储的大量数据。
- CSV 存储引擎把数据以逗号分隔的格式存储在文本文件中。
- BLACKHOLE 存储引擎接受但不存储数据，并且检索总是返回一个空集。
- FEDERATED 存储引擎把数据存在远程数据库中。在 MySQL 5.1 中，它只和 MySQL 一起工作，使用 MySQL C Client API。在未来的分发版中，我们想要让它使用其它驱动器或客户端连接方法连接到另外的数据源。

当年创建一个新表的时候，你可以通过添加一个 ENGINE 或 TYPE 选项到 CREATE TABLE 语句来告诉 MySQL 你要创建什么类型的表：

```
CREATE TABLE t (i INT) ENGINE = INNODB;
```

```
CREATE TABLE t (i INT) TYPE = MEMORY;
```

虽然 TYPE 仍然在 MySQL 5.1 中被支持，现在 ENGINE 是首选的术语。

如何选择最适合您的存储引擎呢？

下述存储引擎是最常用的：

- **MyISAM:** 默认的 MySQL 插件式存储引擎，它是在 Web、数据仓储和其他应用环境下最常使用的存储引擎之一。注意，通过更改 STORAGE_ENGINE 配置变量，能够方便地更改 MySQL 服务器的默认存储引擎。
- **InnoDB:** 用于事务处理应用程序，具有众多特性，包括 ACID 事务支持。
- **BDB:** 可替代 InnoDB 的事务引擎，支持 COMMIT、ROLLBACK 和其他事务特性。
- **Memory:** 将所有数据保存在 RAM 中，在需要快速查找引用和其他类似数据的环境下，可提供极快的访问。
- **Merge:** 允许 MySQL DBA 或开发人员将一系列等同的 MyISAM 表以逻辑方式组合在一起，并作为 1 个对象引用它们。对于诸如数据仓储等 VLDB 环境十分适合。
- **Archive:** 为大量很少引用的历史、归档、或安全审计信息的存储和检索提供了完美的解决方案。
- **Federated:** 能够将多个分离的 MySQL 服务器链接起来，从多个物理服务器创建一个逻辑数据库。十分适合于分布式环境或数据集市环境。
- **Cluster/NDB:** MySQL 的簇式数据库引擎，尤其适合于具有高性能查找要求的应用程序，这类查找需求还要求具有最高的正常工作时间和可用性。
- **Other:** 其他存储引擎包括 CSV（引用由逗号隔开的用作数据库表的文件），Blackhole（用于临时禁止对数据库的应用程序输入），以及 Example 引擎（可为快速创建定制的插件式存储引擎提供帮助）。

请记住，对于整个服务器或方案，你并不一定要使用相同的存储引擎，你可以为方案中的每个表使用不同的存储引擎，这点很重要。

InnoDB 介绍:

InnoDB 存储引擎，支持事务、行锁、外键。[InnoDB](#) 设计用来处理大数据量时提供最好的性能。

InnoDB 提供自己的缓存(buffer pool) 还缓存数据和索引。innodb 把数据和索引存放到了表空间(tablespace)，表空间是几个磁盘文件或者是原生设备文件(raw disk)。它不像 MyISAM 存储引擎，每个表只是一个文件。(这样在某些系统下最大 2G 限制)。而 Innodb 没有此限制，可以无限扩展。

Mysql4.0 的以后版本都支持 innodb 存储。

InnoDB 配置:

innodb 管理的两个磁盘文件是表空间文件和日志文件。

表空间定义:

innodb_data_file_path=datafile_spec1[;datafile_spec2]...

文件声明格式: file_name:file_size[:autoextend[:max:max_file_size]]
innodb_data_home_dir 声明文件存放目录.

mysql 读取配置文件的顺序:
'/etc/my.cnf' Global options.
'\$DATADIR/my.cnf' Server-specific options.
'--defaults-extra-file' The file specified with the --defaults-extra-file option.
'~/.my.cnf' User-specific options.

内存使用下面的加起来不要超过 2G:

innodb_buffer_pool_size
+ key_buffer_size
+ max_connections*(sort_buffer_size+read_buffer_size+binlog_cache_size)
+ max_connections*2MB

InnoDB 参数说明:

innodb_additional_mem_pool_size
存储数据字典信息和内部结构信息, 如果你的表越多, 这个需要的内存就越多, 如果你预留的空间不够, 就开始向系统申请内存. errlog 会有错误. 缺省设置为 1M.

innodb_autoextend_increment
当表空间满时字段扩展大小.

innodb_buffer_pool_size
数据和索引用的缓存大小. 一般时系统物理内存的 50~80%

1.16.4. 论 MySQL 何时使用索引, 何时不使用索引

其实不管什么数据库, index (索引) 是在经常查询的 column 上建立的, 虽然它可以加快查询, 但是如果表中经常有 DML 语句 (增删改) 的话, 反而会成为数据库的负担, 所以如果笼统的来讲的话, 就是你经常要去找, 要去匹配的字段, 比如姓名, 员工编号啊等等。

索引:

使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构，例如 `employee` 表的姓(name)列。如果要按姓查找特定职员，与必须搜索表中的所有行相比，索引会帮助您更快地获得该信息。

索引是一个单独的、物理的数据库结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。

索引提供指向存储在表的指定列中的数据值的指针，然后根据您指定的排序顺序对这些指针排序。数据库使用索引的方式与您使用书籍中的索引的方式很相似：它搜索索引以找到特定值，然后顺指针找到包含该值的行。

在数据库关系图中，您可以在选定表的“索引/键”属性页中创建、编辑或删除每个索引类型。

当保存索引所附加到的表，或保存该表所在的关系图时，索引将保存在数据库中。

注意：

并非所有的数据库都以相同的方式使用索引。作为通用规则，只有当经常查询索引列中的数据时，才需要在表上创建索引。索引占用磁盘空间，并且降低添加、删除和更新行的速度。在多数情况下，索引用于数据检索的速度优势大大超过它的不足之处。但是，如果应用程序非常频繁地更新数据或磁盘空间有限，则可能需要限制索引的数量。

可以基于数据库表中的单列或多列创建索引。多列索引使您可以区分其中一列可能有相同值的行。

如果经常同时搜索两列或多列或按两列或多列排序时，索引也很有帮助。例如，如果经常在同一查询中为姓和名两列设置判据，那么在这两列上创建多列索引将很有意义。

确定索引的有效性：

- 检查查询的 `WHERE` 和 `JOIN` 子句。在任一子句中包括的每一列都是索引可以选择的对象。
- 对新索引进行试验以检查它对运行查询性能的影响。
- 考虑已在表上创建的索引数量。最好避免在单个表上有许多索引。
- 检查已在表上创建的索引的定义。最好避免包含共享列的重叠索引。
- 检查某列中唯一数据值的数量，并将该数量与表中的行数进行比较。比较的结果就是该列的可选择性，这有助于确定该列是否适合建立索引，如果适合，确定索引的类型。

MySQL 何时使用索引

对一个键码使用`>`, `>=`, `=`, `<`, `<=`, `IF NULL` 和 `BETWEEN`

1. `SELECT * FROM table_name WHERE key_part1=1 and key_part2 > 5 ;`

2. `SELECT * FROM table_name WHERE key_part1 IS NULL;`

当使用不以通配符开始的 LIKE

1. `SELECT * FROM table_name WHERE key_part1 LIKE 'jani%'`

在进行联结时从另一个表中提取行时

1. `SELECT * from t1,t2 where t1.col=t2.key_part`

找出指定索引的 MAX() 或 MIN() 值

1. `SELECT MIN(key_part2),MAX(key_part2) FROM table_name where key_part1=10`

一个键码的前缀使用 ORDER BY 或 GROUP BY

1. `SELECT * FROM foo ORDER BY key_part1,key_part2,key_part3`

在所有用在查询中的列是键码的一部分时间

1. `SELECT key_part3 FROM table_name WHERE key_part1=1`

MySQL 何时不使用索引

如果 MySQL 能估计出它将可能比扫描整张表还要快时，则不使用索引。例如如果 `key_part1` 均匀分布在 1 和 100 之间，下列查询中使用索引就不是很好：

1. `SELECT * FROM table_name where key_part1 > 1 and key_part1 < 90`

如果使用 HEAP 表且不用=搜索所有键码部分。

在 HEAP 表上使用 ORDER BY。

如果不是用键码第一部分

1. `SELECT * FROM table_name WHERE key_part2=1`

如果使用以一个通配符开始的 LIKE

1. `SELECT * FROM table_name WHERE key_part1 LIKE '%jani%'`

搜索一个索引而在另一个索引上做 ORDER BY

1. `SELECT * from table_name WHERE key_part1 = # ORDER BY key2`

1.16.5. SQL 多表连接查询实现语句

1.理论

只要两个表的公共字段有匹配值，就将这两个表中的记录组合起来。

个人理解：以一个共同的字段求两个表中符合要求的交集，并将每个表符合要求的记录以共同的字段为牵引合并起来。

语法

```
select * FROM table1 INNER JOIN table2 ON table1 . field1 compopr table2 . field2
```

`INNER JOIN` 操作包含以下部分：

部分 说明

table1, 要组合其中的记录的表的名称。

table2

要联接的字段的名称。如果它们不是数字，则这些字段的数据类型必须相同，并且包含同类数据，但是，
field1, field2 它们不必具有相同的名称。

compopr 任何关系比较运算符：“=”、“<”、“>”、“<=”、“>=”或者“<>”。

说明

可以在任何 `FROM` 子句中使用 `INNER JOIN` 操作。这是最常用的联接类型。只要两个表的公共字段上存在相匹配的值，Inner 联接就会组合这些表中的记录。

可以将 `INNER JOIN` 用于 `Departments` 及 `Employees` 表，以选择出每个部门的所有雇员。

而要选择所有部分（即使某些部门中并没有被分配雇员）或者所有雇员（即使某些雇员没有分配到任何部门），则可以通过 `LEFT JOIN` 或者 `RIGHT JOIN` 操作来创建外部联接。

如果试图联接包含备注或 OLE 对象数据的字段，将发生错误。

可以联接任何两个相似类型的数字字段。例如，可以联接自动编号和长整型字段，因为它们均是相似类型。然而，不能联接单精度型和双精度型类型字段。

下例展示了如何通过 `CategoryID` 字段联接 `Categories` 和 `Products` 表：

```
SELECT CategoryName, ProductName  
FROM Categories INNER JOIN Products  
ON Categories.CategoryID = Products.CategoryID;
```

在前面的示例中，CategoryID 是被联接字段，但是它不包含在查询输出中，因为它不包含在 SELECT 语句中。若要包含被联接字段，请在 SELECT 语句中包含该字段名，在本例中是指 Categories.CategoryID。

也可以在 JOIN 语句中链接多个 ON 子句，请使用如下语法：

SELECT fields

```
FROM table1 INNER JOIN table2  
ON table1.field1 compopr table2.field1 AND  
ON table1.field2 compopr table2.field2 OR  
ON table1.field3 compopr table2.field3;
```

也可以通过如下语法嵌套 JOIN 语句：

SELECT fields

```
FROM table1 INNER JOIN  
(table2 INNER JOIN [( ]table3  
[INNER JOIN [( ]tablex [INNER JOIN ...])]  
ON table3.field3 compopr tablex.fieldx)]  
ON table2.field2 compopr table3.field3)  
ON table1.field1 compopr table2.field2;
```

LEFT JOIN 或 RIGHT JOIN 可以嵌套在 INNER JOIN 之中，但是 INNER JOIN 不能嵌套于 LEFT JOIN 或 RIGHT JOIN 之中。

2.操作实例

表 A 记录如下：

aID	aNum
1	a20050111
2	a20050112
3	a20050113
4	a20050114
5	a20050115

表 B 记录如下：

bID	bName
1	2006032401
2	2006032402
3	2006032403
4	2006032404
8	2006032408

实验如下:

1.left join

sql 语句如下:

```
select * from A  
left join B  
on A.aID = B.bID
```

结果如下:

```
aID aNum bID bName  
1 a20050111 1 2006032401  
2 a20050112 2 2006032402  
3 a20050113 3 2006032403  
4 a20050114 4 2006032404  
5 a20050115 NULL NULL
```

(所影响的行数为 5 行)

结果说明:

left join 是以 A 表的记录为基础的,A 可以看成左表,B 可以看成右表, left join 是以左表为准的.
换句话说,左表(A)的记录将会全部表示出来,而右表(B)只会显示符合搜索条件的记录(例子中为: A.aID = B.bID).

B 表记录不足的地方均为 NULL.

2.right join

sql 语句如下:

```
select * from A  
right join B  
on A.aID = B.bID
```

结果如下:

```
aID aNum bID bName  
1 a20050111 1 2006032401  
2 a20050112 2 2006032402  
3 a20050113 3 2006032403  
4 a20050114 4 2006032404  
NULL NULL 8 2006032408
```

(所影响的行数为 5 行)

结果说明:

仔细观察一下,就会发现,和 left join 的结果刚好相反,这次是以右表(B)为基础的,A 表不足的地方用 NULL 填充.

3.inner join

sql 语句如下:

```
select * from A
```

```
innerjoin B
```

```
on A.aID = B.bID
```

结果如下:

```
aID aNum bID bName
```

```
1 a20050111 1 2006032401
```

```
2 a20050112 2 2006032402
```

```
3 a20050113 3 2006032403
```

```
4 a20050114 4 2006032404
```

结果说明:

很明显,这里只显示出了 $A.aID = B.bID$ 的记录. 这说明 inner join 并不以谁为基础, 它只显示符合条件的记录. 还有就是 inner join 可以结合 where 语句来使用 如: `select * from A innerjoin B on A.aID = B.bID where b.bname='2006032401'` 这样的话 就只会放回一条数据了

1.17.面试题集合 (十六)

1.17.1. 12 个高矮不同的人,排成两排,每排必须是从矮到高排列,而且第二排比对应的第一排的人高,问排列方式有多少种

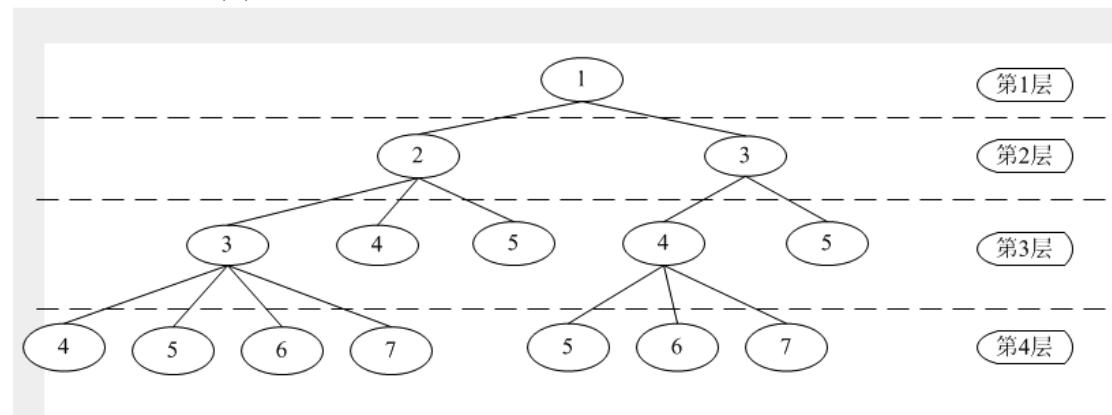
12 个高矮不同的人,排成两排,每排必须是从矮到高排列,而且第二排比对应的第一排的人高,问排列方式有多少种?

我们可以从以下几个方面来分析这个问题:

- a. 先把这 12 个高矮不同的人按身高递增的顺序排成 1 列, 并依次从 1 到 12 编号来简化不同身高的表示方法;
- b. 如果第一排的排列次序已确定的话, 那么第二排的排列就自然确定;

- c. 显然，如果只有编号为 1 和 2 的 2 个人的话，排列方式只有一种 (1) (2)；
- d. 对于编号为 1 - n 的每一种符合条件的排列 (n 为偶数，并且假设第一排末尾的编号为 x)，在第一排加入 n+1 和在第二排加入 n+2 可以产生一种符合条件的编号为 1 - (n+2) 的排列；
- e. 在 d 条件下，可以把新加入第一排的 n+1 替换成 大于 x 且小于 n+1 的编号（即第一排新末尾的编号值 x' 的取值范围为大于 x 且小于等于 n+1），因为这样的替换保证了新的第一排的排列都是从原来的第一排生成的（即第一排的前 n/2 个排列与原来的相同），而且在第二排中加入编号 n+1 后会使第二排的某个编号值变大（因为第二排中的 n+2 不会被换出）。显然，无论第二排中哪个编号变大，只要重新按编号递增排列第二排，都会保证原来 第二排比对应的第一排的人高（即编号较大），而第二排末尾的编号 n+2 总是能够确保大于第一排的末位；

因此，我们可以根据以上分析来画出从 **编号为 1 - n 的排列 到 编号为 1 - (n+2) 的排列** 的状态转换树 (n=2,4,6 的情况)：



以上状态转换树只画出了第一排的状态转换，对于第 3 层，它的 $n+1=5$, $n+2=6$ ，显然 $n+2$ 的值即为节点层序值的 2 倍

所以，求编号为 **1 - 12** 的排列总数，就是求该树的第 **6** 层的节点数，而从根节点 **1** 到第 **6** 层的每个节点的路径就是一个符合条件的第一排排列（可以用树的遍历算法求得）。对于每个节点我们可以赋它的编号值为 **order** 和层序值为 **level**，以下是简单计算第 6 层节点数的方法：

Java 代码



```

1. private void firstOrder(int order, int level) {
2.     place.add(order);

```

```
3.     if (level >= 6) {  
4.         count++;  
5.         place.pop();  
6.         return;  
7.     }  
8.  
9.     for (int i = order + 1; i < 2 * (level + 1); i++)  
10.        firstOrder(i, level + 1);  
11.        place.pop();  
12.    }
```

[java] [view plain](#) [copy print?](#)

```
1.  private void firstOrder(int order, int level) {  
2.      place.add(order);  
3.      if (level >= 6) {  
4.          count++;  
5.          place.pop();  
6.          return;  
7.      }  
8.  
9.      for (int i = order + 1; i < 2 * (level + 1); i++)  
10.         firstOrder(i, level + 1);  
11.         place.pop();  
12.     }
```

以下是计算并打印第一排排列的完整代码:

Java 代码   

```
1.  import java.util.Stack;  
2.  
3.  public class TwoOrderQueue {  
4.  
5.      private int count = 0;
```

```
6.     private int total = 0;
7.     private Stack<Integer> place = new Stack<Integer>();
8.
9.     public TwoOrderQueue(int total) {
10.         this.total = total;
11.     }
12.
13.     private void firstOrder(int order, int level) {
14.         place.add(order);
15.         if (level >= total / 2) {
16.             //计数叶结点并打印路径
17.             count++;
18.             for (Integer i : place)
19.                 System.out.print(i + " ");
20.             System.out.println();
21.             place.pop();
22.             return;
23.         }
24.
25.         //展开编号为 order 的节点的子树，并进入下一层 level+1
26.         for (int i = order + 1; i < 2 * (level + 1); i++)
27.             firstOrder(i, level + 1);
28.         place.pop();
29.     }
30.
31.     public void firstOrder() {
32.         firstOrder(1, 1);
33.         System.out.println("第一排一共有" + count + "种排列");
34.     }
35.
```

```
36. public static void main(String[] args) {  
37.     TwoOrderQueue q = new TwoOrderQueue(12);  
38.     q.firstOrder();  
39. }  
40. }
```

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. import java.util.Stack;  
2.  
3. public class TwoOrderQueue {  
4.  
5.     private int count = 0;  
6.     private int total = 0;  
7.     private Stack<Integer> place = new Stack<Integer>();  
8.  
9.     public TwoOrderQueue(int total) {  
10.         this.total = total;  
11.     }  
12.  
13.     private void firstOrder(int order, int level) {  
14.         place.add(order);  
15.         if (level >= total / 2) {  
16.             //计数叶结点并打印路径  
17.             count++;  
18.             for (Integer i : place)  
19.                 System.out.print(i + " ");  
20.             System.out.println();  
21.             place.pop();  
22.             return;  
23.         }  
24.  
25.         //展开编号为 order 的节点的子树，并进入下一层 level+1  
26.         for (int i = order + 1; i < 2 * (level + 1); i++)  
27.             firstOrder(i, level + 1);  
28.         place.pop();
```

```
29.    }
30.
31.    public void firstOrder() {
32.        firstOrder(1, 1);
33.        System.out.println("第一排一共有" + count + "种排列");
34.    }
35.
36.    public static void main(String[] args) {
37.        TwoOrderQueue q = new TwoOrderQueue(12);
38.        q.firstOrder();
39.    }
40. }
```

小结：

我们把 12 个高矮不同的人按身高递增的顺序排成 1 列，并依次从 1 到 12 编号，这里面除了简化问题之外，还存在什么样的思维切入点呢？我们把 12 个人分为 2 列，这里隐含了一个“有序”的方法，即假设有 2 个空队列，12 个人依次选择是进入第 1 个队列还是第 2 个队列，最后只要两队人数相同即可。显然，这 12 个人进入队列的次序是可以任意的，那么我们为什么不选择一种有序的次序呢？而最明显的一种有序的次序就属身高递增的次序了，把他们依次编号的话不仅可以区分不同的身高，还表示了依次进入队列的次序，这样就最容易在加入队列的过程中找出规律。

我们都知道，**计算机最擅长的就是使用规则来求解规范的问题**。比如单纯形法，第一步就是要把线性方程组规范化，而这里问题最好的规范化就是问题分析中的 a 步骤。

使用排列组合的方法求解（参见 think365 帖子中 BenArfa 的解答）：

如果要满足题意，只要从 12 个人中挑选 6 个人放在第一排，那么所有人的位置就都确定了，因为是要求按身高排序的。

这里我们的挑选方法以及限制条件是这样的：12 个人先从矮到高排序，然后每个人被选到第一排或者第二排，如果要满足题意，当且仅当每挑选完一次之后，第二排中的人数不多于第一排中的人数，而这个条件的排列就是 $C(12,6) - C(12,5) = 132$ ，但这样并不能求得具体的排列次序。

1.17.2. 毒酒

有 1000 桶酒，其中 1 桶有毒。而一旦吃了，毒性会在 1 周后发作。

现在我们用小老鼠做实验，要在 1 周内找出那桶毒酒，问最少需要多少老鼠。

10 只老鼠按顺序排好每桶酒按照编号转换成二进制，给相应位置上是 1 的老鼠喝。最后按死掉的老鼠是哪几只，然后排成二进制，再转成十进制就是第几桶酒。比如：

第 70 桶酒，70 转换成二进制就是 0001000110，那么就给第四、八、九只老鼠喝。如果最后死掉第三、七、八只老鼠，那么就是 0010001100，转换成十进制就是 140，即 140 桶酒有毒。

1.17.3. 用代码验证阿里巴巴的一道关于男女比例的面试题

其中有一个问题，比较有意思：

说澳大利亚的父母喜欢女孩，如果生出来的第一个女孩，就不再生了，如果是男孩就继续生，直到生到第一个女孩为止，问若干年后，男女的比例是多少？

刚看到问题是的思维逻辑：用递推法，假设一对夫妻，生了个女儿，就不再要了；另外一对夫妻，生了个儿子，再要一个，是女儿，然后也就不要了。第一感觉，应该是女的比男的多。然后思考如何证明这个结论。用数学可以证明，比如用归纳法，看看男女到底比例应该是多少。

后来用仔细考虑了一下：一个家庭如果一直没有生女孩，不是要生多个男孩才对一个女孩吗？女多于男的结论未必正确。我的一个同事坚持认为，女孩的个数一定多余男孩的个数。哈哈，真是这样吗？

有一点我后来考虑到了，就是生男和生女的几率都是一样的，是不是应该是 1: 1 啊？

数学退化的差不多了，还是用编程来验证一下。我写了一个下面的小程序来验证我的程序，编译环境是 linux AS5.0。

[cpp] [view plain](#) [copy print?](#)

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define MAN 1 //男孩
4. #define WOMAN 0 //女孩
5. static int g_iManNum = 0; //男孩个数
```

```
6. static int g_iWoManNum = 0; //女孩个数
7. static void AddOneMan()
8. {
9.     g_iManNum++;
10. }
11. static void AddOneWoman()
12. {
13.     g_iWoManNum++;
14. }
15. static int GetOneChild()
16. {
17.     return (rand()%2);
18. }
19. static void PrintResult()
20. {
21.     printf("Man's number is %d, woman's number is %d./n", g_iManNum,
22. g_iWoManNum);
23.
24.     float fResult = (float)g_iManNum / (float)g_iWoManNum;
25.
26.     printf("Man / Woman is [%f]/n", fResult);
27. }
28. static void OneFamilyGetChild()
29. {
30.     int iChildType = GetOneChild();
31.     if (MAN == iChildType)
32.     {
33.         //如果是男孩，则递归调用，再生一个小孩，直到生出一个女孩为止
34.         AddOneMan();
35.         OneFamilyGetChild();
36.     }
37.     else if (WOMAN == iChildType)
38.     {
39.         AddOneWoman();
40.     }
```

```
41.     else
42.     {
43.         printf("child type is error!/n");
44.         exit(0);
45.     }
46. }
47. static void StatisticsOfAllFamily(int iFamilyCount)
48. {
49.     printf("Now family count is %d./n", iFamilyCount);
50.
51.     int i = 0;
52.     for (i = 0; i < iFamilyCount; i++)
53.     {
54.         OneFamilyGetChild();
55.     }
56.
57.     PrintResult();
58. }
59. int main ()
60. {
61.     srand(time(NULL));
62.     StatisticsOfAllFamily(1000000);
63.
64.     return 1;
65. }
```

最终的结果是 0.9996 约等于 1，男孩小于女孩。我运行了多次，都是这个结果。从理论上来
说，不一定“男孩个数一定小于女孩个数”，但是结果每次都是这样，可能是“男孩个数小于
女孩个数”出现的几率比较高。

哈哈，有没有人有兴趣，继续研究一下，“男孩个数小于女孩个数”出现的几率有多大？或者
是一个随机数？

1.17.4. 金币

题目： 10 个房间里放着随机数量的金币。每个房间只能进入一次，并只能在一个房间中拿金币。一个人采取如下策略：前四个房间只看不拿。随后的房间只要看到比前四个房间都多的金币数，就拿。否则就拿最后一个房间的金币。编程计算这种策略拿到最多金币的概率。

```
int genrand(int a, int b)
{
    return rand()% (b-a+1)+a;
}

void gennum(int *a, int size)
{
    for (int i=0;i<size;i++)
    {
        a[i] = genrand(1,10000);
    }
}

int getnum(int *a, int size)
{
    int max4 = 0;
    for (int i=0;i<4;i++)
    {
        if (a[i]>max4) max4 = a[i];
    }
    for (i=4;i<size-1;i++)
    {
        if (a[i]>max4) return a[i];
    }
    return a[size-1];
}

int getmax(int *a, int size)
{
    int max = 0;
    for (int i=0;i<size;i++)
    {
```

```

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

int success(int *a, int size)
{
gennum(a,size);
if (getnum(a,size)==getmax(a,size))
return 1;
return 0;
}

```

主程序调用：

```

srand(LOWORD(GetCurrentTime()));

int a[10];

int total = 10000000;

int count = 0;

for (int i=0;i<total;i++)
{
    if (success(a,10)) count++;
}

cout<<(double)count/(double)total<<endl;

```

最后得到结果是：成功的概率接近 40%，例如 39.826%

1.17.5. 海盗

其实任何推理的源泉都在于简化。所以推理过程是这样的：

从后向前推，如果 1 – 3 号 强盗都喂了鲨鱼，只剩 4 号和 5 号的话，5 号一定投 反对票让 4 号喂鲨鱼，以独吞全部金币。所以，4 号惟有支持 3 号才能保命。

3 号知道这一点，就会提（1 0 0，0，0）的分配方案，对 4 号、5 号一毛不拔而将全部金币归为已有，因为他知道 4 号一无所获但还是会投赞成票，再加上自己一票，他的方案即可通过。

不过，2 号推知到 3 号的方案，就会提出（9 8，0，1，1）的方案，即放弃 3 号，而给予 4 号和 5 号各一枚金币。

由于该方案对于 4 号和 5 号来说比在 3 号分配时更为有利，他们将支持他而不希望他出局而由 3 号来分配。这样，2 号将拿走 9 8 枚金币。

不过，2号的方案会被1号所洞悉，1号并将提出（97，0，1，2，0）或（97，0，1，0，2）的方案，即放弃2号，而给3号一枚金币，同时给4号（或5号）2枚金币。

由于1号的这一方案对于3号和4号（或5号）来说，相比2号分配时更优，他们将投1号的赞成票，再加上1号自己的票，1号的方案可获通过，97枚金币可轻松落入囊中。这无疑是1号能够获取最大收益的方案了！

正确分配方法为（97，0，1，0，2）或者是（97，0，1，2，0）

第一个版本：

有5个海盗，按照等级从5到1排列。最大的海盗有权提议他们如何分享100枚金币。但其他人要对此表决，如果多数反对，那他就会被杀死。他应该提出怎样的方案，既让自己拿到尽可能多的金币又不会被杀死？

答案：98,0,1,0,1

你反向思考就行了，从倒数第2个开始。

给人员编号

1，2，3，4，5

5 我要是能拿到1个就满足了

4 当然是100%了，因为自己肯定是支持，，所以5关心自己能拿到至少一个金币的方案
3 99个给5一个，这样5肯定会支持，否则4就会拿到所有的，他就一个都拿不到了。4则关心自己能拿到金币的方案

2 99个，给4一个，这样2:2,4如果不支持，就只能一个都拿不到，所以会支持你。而3和5则关心能拿到一个的方案

1 需要2个支持，给3和5，他们肯定会支持，否则他们将一个都拿不到。

据说有一个原则，相邻是冤家，冤家的冤家就是朋友，

第二个版本

有 5 个海盗，按照等级从 5 到 1 排列。最大的海盗有权提议他们如何分享 100 枚金币。但其他人要对此表决，如果方案得到超过半数的人同意，则按照他的方案进行，否则那他就会被杀死。他应该提出怎样的方案，既让自己拿到尽可能多的金币又不会被杀死？

给人员编号

1, 2, 3, 4, 5

5 不说了，就他一个人了

4 无论怎么分，5 都会反对，所以他为了活命，必须支持 3 号的任何分发，否则就会因为无法得到半数以上的支持而死掉

3 自己 100，因为 4 肯定会支持他，所以会被通过，4 和 5 则关心能否拿到一个的方案

2 无论怎么分，3 都会反对，所以他必须给 4 和 5 每个人一个，否则他们任何一个人反对，都会被杀。98,0,1,1,而 3 则关心自己能拿到一个的方案，而 4, 5 则关心能拿到 2 个的方案，因为他们肯定能拿到一个。

1 需要得到 2 个支持，从利益最大化看，可以给 3 号 1 个，给 4 或者 5 号 2 个，97,0,1,2,0 或者 97,0,1,0,2

3 肯定支持，否则 2 号分配，他啥都得不到

4/5 肯定支持，否则 2 分配，他们最多拿到一个

1.17.6. 1024

末尾 0 的个数取决于乘法中因子 2 和 5 的个数。显然乘法中因子 2 的个数大于 5 的个数，所以我们只需统计因子 5 的个数。

是 5 的倍数的数有： $1024 / 5 = 204$ 个

是 25 的倍数的数有： $1024 / 25 = 40$ 个

是 125 的倍数的数有： $1024 / 125 = 8$ 个

是 625 的倍数的数有： $1024 / 625 = 1$ 个

所以 $1024!$ 中总共有 $204+40+8+1=253$ 个因子 5。

也就是说 $1024!$ 末尾有 253 个 0。

“弱弱地算了一下， $1024/5 + 1024/25 + 1024/125 + 1024/625$ ”这个是对的，计算步骤如下：首先考虑有几个 5, $1024/5=204$ 个。那么，为什么还要算 $1024/25$ 呢？因为 $25*4$ 或 $25*2*12$ 会有两个 0、100 也有 2 个 0，因为已经在 $1024/5$ 中算过一次了，所以此处只加一次。同理，125 有 3 个 0, 625 有 4 个 0。相加共 253 个 0。

1.17.7. 最少零钱问题 最少硬币问题

```
*****问题描述*****  
/* 设有 n 种不同面值的硬币，现要用这些面值的硬币来找开待凑钱数 m，可以使用的各种面值的硬币个数不限。  
找出最少需要的硬币个数，并输出其币值。  
数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n (n<=13) ，  
表示有 n 种硬币可选。接下来的一行是每种硬币的面值。由用户输入待找钱数 m。  
结果输出：程序运行结束时，将计算出的所需最少硬币个数和币值输出到文件 output.txt 中。
```

```
*****动态规划实现*****  
/*长度为 m 的数组 c[1...m]中存放一系列子结果，即 c[i]为要凑的钱数为 i 时所需的最少硬币数，则 c[m]为所求  
当要找的钱数 i(1<i<m)与当前所试探的硬币面值 k 相等时，结果为 1，即 c[i]=1  
当 i 大于当前所试探硬币面值 k 时，若 c[i] 为 0，即还未赋过值，且 c[i-k] 不为 0，即从 i 元钱中刨去 k 元后剩下的钱数可以找开，  
则 c[i]=c[i-k]+1  
若 c[i] 不为 0，即已赋过值，则 c[i] 为 c[i-k]+1 和 c[i] 中较小的  
*/  
  
#include<iostream.h>  
void main()  
{  
    int i,j;  
    int m,n;//m 为要凑的钱数，n 为不同面值的硬币个数  
    cout<<"请输入要凑的钱数:";  
    cin>>m;  
    int *c=new int[m+1];//动态开辟长度为 m+1 的数组 c[],c[0]不用,其中存放各个子结果  
    for(i=0;i<=m;i++)//赋初值为 0  
        c[i]=0;  
    cout<<"请输入不同面值的硬币个数:";  
    cin>>n;  
    int *a=new int[n];//动态开辟长度为 n 的数组 a[],其中存放 n 个硬币的面值  
    cout<<"请输入 "<<n<<" 个不同的硬币面值:";  
    for(j=0;j<n;j++)  
        cin>>a[j];
```

```

for(j=0;j<n;j++)//当要找的钱数 i(1<i<m)与当前所试探的硬币面值 k 相等时, 结果为 1, 即 c[i]=1
c[a[j]]=1; //即把与硬币面值相等的数组 c 中的位置赋 1
for(i=1;i<=m;i++)//要凑的钱数为 i
for(j=0;j<n;j++)//试探面值为 a[j]的第 j+1 个硬币
if(i>a[j])//钱数 i 大于当前硬币面值时(等于的已处理过;小于的当前找不开,跳过)
{
    if(c[i]==0 && c[i-a[j]]!=0)
        c[i]=c[i-a[j]]+1;
    else
        if(c[i-a[j]]!=0) //第一次没考虑到此时 c[i-a[j]]应大于零, 感谢 csdn 的 vasile010 指正
            c[i]=c[i-a[j]]+1 < c[i] ? c[i-a[j]]+1 : c[i];
    }
if(c[m]==0)
cout<<"找不开!"<<endl;
else
cout<<c[m]<<endl;
}

/*注：贪心算法可能找不到最优解或者找不到解、例如：要找开 15 元，可用硬币为 1 元，5 元，11 元，最优解为 3 个 5 元，而用贪心算法得到的解是 11 元和 4 个 5 元。再如：要找开 4 元，可用硬币为 2 元和 3 元的，最优解为 2 个 2 元，而用贪心算法则找不到解*/

```

1.17.8. 石子合并

【石子合并】

在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

试设计一个算法，计算出将 n 堆石子合并成一堆的最小得分和最大得分。

【输入文件】

包含两行，第 1 行是正整数 n ($1 \leq n \leq 100$)，表示有 n 堆石子。

第 2 行有 n 个数，分别表示每堆石子的个数。

【输出文件】

输出两行。

第 1 行中的数是最小得分；第 2 行中的数是最大得分。

【输入样例】

4 4 5 9

【输出样例】

43

54

【分析】

本题初看以为可以使用贪心法解决问题，但是事实上因为有必须相邻两堆才能合并这个条件在，用贪心法就无法保证每次都能取到所有堆中石子数最多的两堆。例如下面这个例子：

6

3 4 6 5 4 2

如果使用贪心法求最小得分，应该是如下的合并步骤：

第一次合并 3 4 6 5 4 2 2,3 合并得分是 5

第二次合并 5 4 6 5 4 5,4 合并得分是 9

第三次合并 9 6 5 4 5,4 合并得分是 9

第四次合并 9 6 9 9,6 合并得分是 15

第五次合并 15 9 15,9 合并得分是 24

总得分 = 5 + 9 + 9 + 15 + 24 = 62

但是如果采用如下合并方法，却可以得到比上面得分更少的方法：

第一次合并 3 4 6 5 4 2 3,4 合并得分是 7

第二次合并 7 6 5 4 2 7,6 合并得分是 13

第三次合并 13 5 4 2 4,2 合并得分是 6

第四次合并 13 5 6 5,6 合并得分是 11

第五次合并 13 11 13,11 合并得分是 24

总得分 = 7 + 13 + 6 + 11 + 24 = 61

由此我们知道本题是不可以使用贪心法求解的，上例中第五次合并石子数分别为 13 和 11 的相邻两堆。这两堆石头分别由最初 的第 1, 2, 3 堆（石头数分别为 3, 4, 6）和第 4, 5, 6 堆（石头数分别为 5, 4, 2）经 4 次合并后形成的。于是问题又归结为如何使得这两个子序列的 N-2 次合并的得分总和最优。为了实现这一目标，我们将第 1 个序列又一分为二：第 1、2 堆构成子序列 1，第 3 堆为子序列 2。第一次合并子序列 1 中的两堆，得分 7；第二次再将之与子序列 2 的一堆合并，得分 13。显然对于第 1 个子序列来说，这样的合并方案是最优的。同样，我们将第 2 个子序列也一分为二；第 4 堆为子序列 1，第 5, 6 堆构成子序列 2。第三次合 并子序列 2 中的 2 堆，得分 6；第四次再将之与子序列 1 中的一堆合并，得分 13。显然对于第二个子序列来说，这样的合并方案也是最优的。由此得出一个结论——6 堆石子经过这样的 5 次合并后，得分的总和最小。

动态规划思路：

阶段 i：石子的每一次合并过程，先两两合并，再三三合并，...最后 N 堆合并

状态 s：每一阶段中各个不同合并方法的石子合并总得分。

决策：把当前阶段的合并方法细分成前一阶段已计算出的方法，选择其中的最优方案

具体来说我们应该定义一个数组 $s[i,j]$ 用来表示合并方法， i 表示从编号为 i 的石头开始合并， j 表示从 i 开始数 j 堆进行合并， $s[i,j]$ 为合并的最优得分。

对于上面的例子来说，初始阶段就是 $s[1,1], s[2,1], s[3,1], s[4,1], s[5,1], s[6,1]$ ，因为一开始还没有合并，所以这些值应该全部为 0。

第二阶段：两两合并过程如下，其中 $\text{sum}(i,j)$ 表示从 i 开始数 j 个数的和

$$s[1,2]=s[1,1]+s[2,1]+\text{sum}(1,2)$$

$$s[2,2]=s[2,1]+s[3,1]+\text{sum}(2,2)$$

$$s[3,2]=s[3,1]+s[4,1]+\text{sum}(3,2)$$

$$s[4,2]=s[4,1]+s[5,1]+\text{sum}(4,2)$$

$$s[5,2]=s[5,1]+s[6,1]+\text{sum}(5,2)$$

$$s[6,2]=s[6,1]+s[1,1]+\text{sum}(6,2)$$

第三阶段：三三合并可以拆成两两合并，拆分方法有两种，前两个为一组或后两个为一组

$$s[1,3]=s[1,2]+s[3,1]+\text{sum}(1,3) \text{ 或 } s[1,3]=s[1,1]+s[2,2]+\text{sum}(1,3), \text{ 取其最优}$$

$$s[2,3]=s[2,2]+s[4,1]+\text{sum}(2,3) \text{ 或 } s[1,3]=s[2,1]+s[3,2]+\text{sum}(2,3), \text{ 取其最优}$$

·
·
·

第四阶段：四四合并的拆分方法用三种，同理求出三种分法的得分，取其最优即可。以后第五阶段、第六阶段依次类推，最后在第六阶段中找出最优答案即可。

由此得到算法框架如下：

```
For j←2 to n do {枚举阶段，从两两合并开始计算}  
  For i←1 to n do {计算当前阶段的 n 种不同状态的值}  
    For k←1 to j-1 do {枚举不同的分段方法}  
      begin  
        If i+k>n then t←(i+k) mod n else t←i+k {最后一个连第一个的情况处理}  
        s[i,j]←最优{s[i,k]+s[t,j-k]+\text{sum}[1,3]} {sum[i,j] 表示从 i 开始数 j 个数的和}  
      end;
```

【参考程序】

```
var  
n:integer;
```

```

a:array[1..100] of longint;
s:array[1..100,1..100] of longint;
t:array[0..100,0..100] of longint;
i,j,k,temp,max,min:longint;

begin
  assign(input,'shizi.in');
  reset(input);
  readln(n);
  fillchar(t,sizeof(t),0); { 计算和数组 }
  for i:=1 to n do
    read(a[i]);
  for i:=1 to n do
    for j:=1 to n do
      for k:=i to i+j-1 do
        begin
          if k>n then temp:=k mod n else temp:=k;
          t[i,j]:=t[i,j]+a[temp];
        end;
  {动态规划求最大得分}
  fillchar(s,sizeof(s),0);
  for j:=2 to n do
    for i:=1 to n do
      for k:=1 to j-1 do
        begin
          if i+k>n then temp:=(i+k) mod n else temp:=i+k; { 处理环形问题 }
          max:=s[i,k]+s[temp,j-k]+t[i,j];
          if s[i,j]<max then s[i,j]:=max;
        end;
  max:=0; { 在最后的阶段状态中找最大得分 }
  for i:=1 to n do
    if max<s[i,n] then max:=s[i,n];
  {动态规划求最小得分}
  fillchar(s,sizeof(s),0);
  for j:=2 to n do
    for i:=1 to n do

```

```

begin
    min:=max long int;
    for k:=1 to j-1 do
        begin
            if i+k>n then temp:=(i+k) mod n else temp:=i+k; {处理环形问题}
            s[i,j]:=s[i,k]+s[temp,j-k]+t[i,j];
            if min>s[i,j] then min:=s[i,j];
        end;
        s[i,j]:=min;
    end;
    min:=max long int; {在最后的阶段状态中找最小得分}
    for i:=1 to n do
        if min>s[i,n] then min:=s[i,n];
    writeln(max);
    writeln(min);
end.

```

1.18.面试题集合（十七）

1.18.1. 生产者-消费者模式

生产者-消费者模式实现概述

生产者与消费者模式是我们在编程过程中经常会遇到的，就像我们生活那样，生产者生产出产品，消费者去购买产品。在这里我们创建三个线程，一个主控线程 main，用于创建各辅助线程；一个生产者线程，用于生产产品；一个消费者线程，用于购买产品。另外，我们创建一个队列类 Queue，生产线程生产的产品将放置到该队列中，然后消费者线程在该队列中取走产品。

2. 实现该模式的要点：

首先必须让生产者线程与消费者线程达到同步，也就是说，当生产者线程生产出产品后，消费者才能去取，依此轮回。当生产者线程放置产品到队列中时，队列要检查队列是否已满，如已满，则等待消费者线程将产品取走，否则放置产品到队列中。当消费者线程在队列中取产品时，队列也要检查队列是否为空，如果为空，则等待生产者线程放置产品到队列，否则在队列中取走产品。

3. 源代码：

[java] [view plain](#) [copy print?](#)

```
1. class MainThread
2.
3. {
4.
5.     public static void main(String[] args)
6.
7.     {
8.
9.         Queue queue=new Queue();
10.
11.        Producer producer=new Producer(queue);
12.
13.        Consumer consumer=new Consumer(queue);
14.
15.        new Thread(producer).start();
16.
17.        new Thread(consumer).start();
18.
19.    }
20.
21. }
22.
23. /*注意:wait notify notifyAll 只能在同步方法或内步块中调用*/
24.
25. class Queue
26.
27. {
28.
29.     int product=0;
30.
31.     boolean bfull=false;
32.
33.     public synchronized void setProduct(int product)
34.
```

```
35. {
36.
37. if(bfull)//如果队列已满，则调用 wait 等待消费者取走产品
38.
39. {
40.
41. try
42.
43. {
44.
45. wait();
46.
47. }
48.
49. catch(Exception e)
50.
51. {
52.
53. e.printStackTrace();
54.
55. }
56.
57. }
58.
59. /*开始放置产品到队列中*/
60.
61. this.product=product;
62.
63. System.out.println(Producer set product:+product);
64.
65. bfull=true;
66.
67. notify();//生产产品后通知消费者取走产品
68.
69. }
```

```
70.  
71. public synchronized void getProduct()  
72.  
73. {  
74.  
75. if(!bfull)//如果队列是空的，则调用 wait 等待生产者生产产品  
76.  
77. {  
78.  
79. try  
80.  
81. {  
82.  
83. wait();  
84.  
85. }  
86.  
87. catch(Exception e)  
88.  
89. {  
90.  
91. e.printStackTrace();  
92.  
93. }  
94.  
95. }  
96.  
97. /*开始从队列取走产品*/  
98.  
99. System.out.println(Consumer get product:+product);  
100.  
101. bfull=false;  
102.  
103. notify();//取走产品后通知生产者继续生产产品  
104.
```

```
105. }
106.
107. }
108.
109. class Producer implements Runnable
110.
111. {
112.
113. Queue queue;
114.
115. Producer(Queue queue)
116.
117. {
118.
119. this.queue=queue;
120.
121. }
122.
123. public void run()//生产线程
124.
125. {
126.
127. for(int i=1;i=10;i++)
128.
129. {
130.
131. queue.setProduct(i);
132.
133. }
134.
135. }
136.
137. }
138.
139. class Consumer implements Runnable
```

```
140.  
141. {  
142.  
143. Queue queue;  
144.  
145. Consumer(Queue queue)  
146.  
147. {  
148.  
149. this.queue=queue;  
150.  
151. }  
152.  
153. public void run()//消费线程  
154.  
155. {  
156.  
157. for(int i=1;j=10;j++)  
158.  
159. {  
160.  
161. queue.getProduct();  
162.  
163. }  
164.  
165. }  
166.  
167. }
```

输出结果如下：

```
Producer set product:1  
Consumer get product:1  
Producer set product:2  
Consumer get product:2
```

Producer set product:3
Consumer get product:3
Producer set product:4
Consumer get product:4
Producer set product:5
Consumer get product:5
Producer set product:6
Consumer get product:6
Producer set product:7
Consumer get product:7
Producer set product:8
Consumer get product:8
Producer set product:9
Consumer get product:9
Producer set product:10
Consumer get product:10
完。 . 。

1.18.2. 动态规划

最优化原理

1951年美国数学家 R. Bellman 等人，根据一类多阶段问题的特点，把多阶段决策问题变换为一系列互相联系的单阶段问题，然后逐个加以解决。一些静态模型，只要人为地引进“时间”因素，分成时段，就可以转化成多阶段的动态模型，用动态规划方法去处理。与此同时，他提出了解决这类问题的“最优化原理”(Principle of optimality)：

“一个过程的最优决策具有这样的性质：即无论其初始状态和初始决策如何，其今后诸策略对以第一个决策所形成的状态作为初始状态的过程而言，必须构成最优策略”。简言之，一个最优策略的子策略，对于它的初态和终态而言也必是最优的。

这个“最优化原理”如果用数学化一点的语言来描述的话，就是：假设为了解决某一优化问题，需要依次作出 n 个决策 D_1, D_2, \dots, D_n ，如若这个决策序列是最优的，对于任何一个整数 k ， $1 < k < n$ ，不论前面 k 个决策是怎样的，以后的最优决策只取决于由前面决策所确定的当前状态，即以后的决策 $D_{k+1}, D_{k+2}, \dots, D_n$ 也是最优的。

最优化原理是动态规划的基础。任何一个问题，如果失去了这个最优化原理的支持，就不可能用动态规划方法计算。能采用动态规划求解的问题都需要满足一定的条件：

- (1) 问题中的状态必须满足最优化原理；

(2) 问题中的状态必须满足无后效性。

所谓的无后效性是指：“下一时刻的状态只与当前状态有关，而和当前状态之前的状态无关，当前的状态是对以往决策的总结”。

问题求解模式

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

初始状态→| 决策 1 |→| 决策 2 |→...→| 决策 n |→结束状态

图 1 动态规划决策过程示意图

(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两段各状态之间的关系来确定决策。

(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

算法实现

动态规划的主要难点在于理论上的设计，也就是上面 4 个步骤的确定，一旦设计完成，实现部分就会非常简单。使用动态规划求解问题，最重要的就是确定动态规划三要素：问题的阶段，每个阶段的状态以及从前一个阶段转化到后一个阶段之间的递推关系。递推关系必须是从次小的问题开始到较大的问题之间的转化，从这个角度来说，动态规划往往可以用递归程序来实现，不过因为递推可以充分利用前面保存的子问题的解来减少重复计算，所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，从 1 行 1 列开始，以行或者列优先的顺序，依次填写表格，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。下面分别以求解最大化投资回报问题和最长公共子序列问题为例阐述用动态规划算法求解问题的一般思路。

1. 最大化投资回报问题：某人有一定的资金用来购买不同面额的债券，不同面额债券的年收益是不同的，求给定资金，年限以及债券面额、收益的情况下怎样购买才能使此人获得最大投资回报。

程序输入约定：第一行第一列表示资金（1000 的倍数）总量，第二列表示投资年限；第二行表示债券面额总数；从第三行开始每行表示一种债券，占用两列，前一列表示债券面额，后一列表示其年收益，如下输入实例，

10000 1

2

4000 400

3000 250

程序实现如下，注释几乎说明了一切，所以不再另外分析。

/// 此数组是算法的关键存储结构，用来存储不同阶段各种债券

/// 组合下对应可获取的最大利息。

```
int saifa[80005];
```

/// 此函数用于计算当前债券在不同购买额下的最优利息情况，

/// 注意此时的利息情况是基于上一次债券的情况下计算得到的，

/// 也就是说当前利息最优是基于上一次利息最优的基础上计算出来的，

/// 这也正好体现了动态规划中“最优化原则”：不管前面的策略如何，

/// 此后的决策必须是基于当前状态（由上一次决策产生）的最优决策。

```
/*
```

动态规划的求解过程一般都可以用一个最优决策表来描述，

对于本程序，以示例输入为例，对于第一年，其最优决策表如下：

0 1 2 3 4 5 6 7 8 9 10(*1000) -- (1)

0 0 0 0 400 400 400 400 800 800 800 -- (2)

0 0 0 250 400 400 500 650 800 900 900 -- (3)

(1) -- 表示首先选利息为 400 的债券在对应资金下的最优利息。

(2) -- 表示可用来购买债券的资金。

(3) -- 表示在已有状态下再选择利息为 300 的债券在对应资金下的最优利息。

注意上面表格，在求购买利息为 300 的债券获得的最优收益的时候，

参考了以前的最优状态，以 3 行 8 列的 650 为例，7(*1000) 可以

在以前购买了 0 张 4000 的债券的基础上再 2 张 3000 的，也可以在以前购买了 1 张 4000 的基础上再买 1 张 3000，经比较取其收益大的，这就是典型的动态规划中的当前最优状态计算。

本程序中把上面的最优决策二维表用一个一维数组表示，值得借鉴。

```
*/
```

```
void add(int a,int b)
{ cout << a << " " << b << endl; // for debug
for(int i=0;i<=80000;i++)
{
if(i+a > 80000)
{
break;
}
if(saifa[i]+b > saifa[i+a]) // 累计同时购买多种债券时的利息
{
saifa[i+a] = saifa[i] + b;
}
if(i<200) // for debug
cout << i << "-" << saifa[i] << " ";
}
cout << endl; // for debug
}

int main(void)
{
int n,d,money,year,pay,bond;
int ii,i;
scanf("%d",&n);
for(ii=0;ii<n;ii++)
{
memset(saifa,0,sizeof(saifa));
scanf("%d%d",&money,&year);
scanf("%d",&d);
for(i=0;i<d;i++)
{
scanf("%d%d",&pay,&bond);
add(pay/1000,bond);
}
// 计算指定年限内最优组合的本金利息总额
for(i=0;i<year;i++)
{ cout << saifa[money/1000] << " "; // for debug
```

```

    money += saifa[money/1000];
}

cout << endl; // for debug
printf("%d/n",money);
}

return 0;
}

```

上述程序实现方法同样适合于背包问题，最优库存问题等，只是针对具体情况，最优决策表的表示和生成会有所不同。

2. 最长公共子串问题：一个给定序列的子序列是在该序列中删去若干元素后得到的序列。给定两个序列 X 和 Y ，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。最长公共子串就是求给定两个序列的一个最长公共子序列。例如， $X=“ABCBDAB”， Y=“BCDB”$ 是 X 的一个子序列。

问题分析：

给定两个序列 A 和 B ，称序列 Z 是 A 和 B 的公共子序列，是指 Z 同是 A 和 B 的子序列。问题要求已知两序列 A 和 B 的最长公共子序列。如采用列举 A 的所有子序列，并一一检查其是否又是 B 的子序列，并随时记录所发现的子序列，最终求出最长公共子序列。这种方法因耗时太多而不可取。

考虑最长公共子序列问题如何分解成子问题，设 $A=“a_0, a_1, \dots, a_{m-1}”$, $B=“b_0, b_1, \dots, b_{m-1}”$, 并 $Z=“z_0, z_1, \dots, z_{k-1}”$ 为它们的最长公共子序列。不难证明有以下性质：

- (1) 如果 $a_{m-1}=b_{n-1}$, 则 $z_{k-1}=a_{m-1}=b_{n-1}$, 且“ z_0, z_1, \dots, z_{k-2} ”是“ a_0, a_1, \dots, a_{m-2} ”和“ b_0, b_1, \dots, b_{n-2} ”的一个最长公共子序列；
- (2) 如果 $a_{m-1}!=b_{n-1}$, 则若 $z_{k-1}=a_{m-1}$, 蕴涵“ z_0, z_1, \dots, z_{k-1} ”是“ a_0, a_1, \dots, a_{m-2} ”和“ b_0, b_1, \dots, b_{n-1} ”的一个最长公共子序列；
- (3) 如果 $a_{m-1}!=b_{n-1}$, 则若 $z_{k-1}=b_{n-1}$, 蕴涵“ z_0, z_1, \dots, z_{k-1} ”是“ a_0, a_1, \dots, a_{m-1} ”和“ b_0, b_1, \dots, b_{n-2} ”的一个最长公共子序列。

这样，在找 A 和 B 的公共子序列时，如有 $a_{m-1}=b_{n-1}$, 则进一步解决一个子问题，找“ a_0, a_1, \dots, a_{m-2} ”和“ b_0, b_1, \dots, b_{m-2} ”的一个最长公共子序列；如果 $a_{m-1}!=b_{n-1}$, 则要解决两个子问题，找出“ a_0, a_1, \dots, a_{m-2} ”和“ b_0, b_1, \dots, b_{n-1} ”的一个最长公共子序列 和 找出“ a_0, a_1, \dots, a_{m-1} ”和“ b_0, b_1, \dots, b_{n-2} ”的一个最长公共子序列，再取两者中较长者作为 A 和 B 的最长公共子序列。

为了节约重复求相同子问题的时间，引入一个数组，不管它们是否对最终解有用，把所有子问题的解存于该数组中，这就是动态规划法所采用的基本方法，具体说明如下。

定义 $c[i][j]$ 为序列“ a_0, a_1, \dots, a_{i-2} ”和“ b_0, b_1, \dots, b_{j-1} ”的最长公共子序列的长度，计算 $c[i][j]$ 可递归地表述如下：

- (1) $c[i][j] = 0$ 如果 $i=0$ 或 $j=0$;
- (2) $c[i][j] = c[i-1][j-1]+1$ 如果 $i,j>0$, 且 $a[i-1] = b[j-1]$;
- (3) $c[i][j] = \max\{c[i][j-1], c[i-1][j]\}$ 如果 $i,j>0$, 且 $a[i-1] \neq b[j-1]$ 。

按此算式可写出计算两个序列的最长公共子序列的长度函数。由于 $c[i][j]$ 的产生仅依赖于 $c[i-1][j-1]$ 、 $c[i-1][j]$ 和 $c[i][j-1]$, 故可以从 $c[m][n]$ 开始, 跟踪 $c[i][j]$ 的产生过程, 逆向构造出最长公共子序列。细节见程序。

```
#include <stdio.h>
#include <string.h>

#define N 100

char a[N], b[N], str[N];
int c[N][N];

int lcs_len(char* a, char* b, int c[][N])
{
    int m = strlen(a), n = strlen(b), i, j;

    for( i=0; i<=m; i++ )
        c[i][0]=0;
    for( i=0; i<=n; i++ )
        c[0][i]=0;

    for( i=1; i<=m; i++ )
    {
        for( j=1; j<=n; j++ )
        {
            if (a[i-1]==b[j-1])
                c[i][j]=c[i-1][j-1]+1;
            else if (c[i-1][j]>=c[i][j-1])
                c[i][j]=c[i-1][j];
            else
                c[i][j]=c[i][j-1];
        }
    }
}
```

```

    return c[m][n];
}

char* build_lcs(char s[], char* a, char* b)
{
    int i = strlen(a), j = strlen(b);
    int k = lcs_len(a,b,c);
    s[k] = '/0';
    while( k>0 )
    {
        if (c[i][j]==c[i-1][j])
            i--;
        else if (c[i][j]==c[i][j-1])
            j--;
        else
        {
            s[--k]=a[i-1];
            i--; j--;
        }
    }

    return s;
}

void main()
{
    printf("Enter two string (length < %d) :\n",N);
    scanf("%s%s",a,b);
    printf("LCS=%s\n",build_lcs(str,a,b));
}

```

1.18.3. 01 背包

问题描述

求出获得最大价值的方案。

注意：在本题中，所有的体积值均为整数。

算法分析

对于背包问题，通常的处理方法是搜索。

用递归来完成搜索，算法设计如下：

```
function Make( i {处理到第 i 件物品} , j{剩余的空间为 j}:integer) :integer;
```

```
初始时 i=m , j=背包总容量
```

```
begin
```

```
if i=0 then
```

```
    Make:=0;
```

```
if j>=wi then (背包剩余空间可以放下物品 i)
```

```
    r1:=Make(i-1,j-wi)+vi; (第 i 件物品放入所能得到的价值 )
```

```
    r2:=Make(i-1,j) (第 i 件物品不放所能得到的价值 )
```

```
    Make:=max{r1,r2}
```

```
end;
```

这个算法的时间复杂度是 $O(2^n)$ ，我们可以做一些简单的优化。

由于本题中的所有物品的体积均为整数，经过几次的选择后背包的剩余空间可能会相等，在搜索中会重复计算这些结点，所以，如果我们把搜索过程中计算过的结点的值记录下来，以保证不重复计算的话，速度就会提高很多。这是简单的“以空间换时间”。

我们发现，由于这些计算过程中会出现重叠的结点，符合动态规划中子问题重叠的性质。

同时，可以看出如果通过第 N 次选择得到的是一个最优解的话，那么第 N-1 次选择的结果一定也是一个最优解。这符合动态规划中最优子问题的性质。

解决方案

考虑用动态规划的方法来解决，这里的：

阶段是：在前 N 件物品中，选取若干件物品放入背包中； **状态是：**在前 N 件物品中，选取若干件物品放入所剩空间为 W 的背包中的所能获得的最大价值；

决策是：第 N 件物品放或者不放；

由此可以写出动态转移方程：

我们用 $f[i,j]$ 表示在前 i 件物品中选择若干件放在所剩空间为 j 的背包里所能获得的最大价值

$$f[i,j] = \max\{f[i-1,j-Wi] + vi \ (j \geq Wi), f[i-1,j]\}$$

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 $i-1$ 件物品的问题。如果不放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入容量为 v 的背包中”，价值为 $f[v]$ ；如果放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入剩下的容量为 $v-c$ 的背包中”，此时能

获得的最大价值就是 $f[v-c]$ 再加上通过放入第 i 件物品获得的价值 w 。

这样，我们可以自底向上地得出在前 M 件物品中取出若干件放进背包能获得的最大价值，也就是 $f[m,w]$

算法设计如下：

```
procedure Make;  
begin  
for i:=0 to w do  
f[0,i]:=0;  
for i:=1 to m do  
for j:=0 to w do begin  
f[i,j]:=f[i-1,j];  
if (j>=w) and (f[i-1,j-w]+v>f[i,j]) then  
f[i,j]:=f[i-1,j-w]+v;  
end;  
writeln(f[m,wt]);  
end;
```

由于是用了一个二重循环，这个算法的时间复杂度是 $O(n*w)$ 。而用搜索的时候，当出现最坏的情况，也就是所有的结点都没有重叠，那么它的时间复杂度是 $O(2^n)$ 。看上去前者要快很多。但是，可以发现在搜索中计算过的结点在动态规划中也全都要计算，而且这里算得更多（有一些在最后没有派上用场的结点我们也必须计算），在这一点上好像是矛盾的。

事实上，由于我们定下的前提是：所有的结点都没有重叠。也就是说，任意 N 件物品的重量相加都不能相等，而所有物品的重量又都是整数，那末这个时候 W 的最小值是：

$$1+2+2^2+2^3+\dots+2^{n-1}=2^n - 1$$

此时 $n*w > 2^n$ ，动态规划比搜索还要慢~所以，其实背包的总容量 W 和重叠的结点的个数是有关的。

考虑能不能不计算那些多余的结点.....

优化空间复杂度

以上方法的时间和空间复杂度均为 $O(N*V)$ ，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到 $O(V)$ 。

先考虑上面讲的基本思路如何实现，肯定是有两个主循环 $i=1..N$ ，每次算出来二维数组 $f[0..V]$ 的所有值。那么，如果只用一个数组 $f[0..V]$ ，能不能保证第 i 次循环结束后 $f[v]$ 中表示的就是我们定义的状态 $f[v]$ 呢？ $f[v]$ 是由 $f[v]$ 和 $f[v-c]$ 两个子问题递推而来，能否保证在推 $f[v]$ 时（即在第 i 次主循环中推 $f[v]$ 时）能够得到 $f[v]$ 和 $f[v-c]$ 的值呢？事实上，这要求在每次主循环中我们以 $v=V..0$ 的顺序推 $f[v]$ ，这样才能保证推 $f[v]$ 时 $f[v-c]$ 保存的是状态 $f[v-c]$ 的值。伪代码如下：

```
for i=1..N  
for v=V..0  
f[v]=max{f[v],f[v-c]+w};
```

其中的 $f[v]=\max\{f[v], f[v-c]\}$ 一句恰就相当于我们的转移方程 $f[v]=\max\{f[v], f[v-c]\}$, 因为现在的 $f[v-c]$ 就相当于原来的 $f[v-c]$ 。如果将 v 的循环顺序从上面的逆序改成顺序的话, 那么则成了 $f[v]$ 由 $f[v-c]$ 推知, 与本题意不符, 但它却是另一个重要的[背包问题 P02](#) 最简捷的解决方案, 故学习只用一维数组解[01 背包](#)问题是十分必要的。

事实上, 使用一维数组解 01 背包的程序在后面会被多次用到, 所以这里抽象出一个处理一件 01 背包中的物品过程, 以后的代码中直接调用不加说明。

过程 ZeroOnePack, 表示处理一件 01 背包中的物品, 两个参数 cost、weight 分别表明这件物品的费用和价值。

```
procedure ZeroOnePack(cost,weight)  
for v=V..cost  
f[v]=max{f[v],f[v-cost]+weight}
```

注意这个过程里的处理与前面给出的[伪代码](#)有所不同。前面的示例程序写成 $v=V..0$ 是为了在程序中体现每个状态都按照方程求解了, 避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了, 就可以加入优化。费用为 cost 的物品不会影响状态 $f[0..cost-1]$, 这是显然的。

有了这个过程以后, 01 背包问题的伪代码就可以这样写:

```
for i=1..N  
ZeroOnePack(c,w);  
初始化的细节问题
```

我们看到的求最优解的[背包问题](#)题目中, 事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解, 有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法, 要求恰好装满背包, 那么在初始化时除了 $f[0]$ 为 0 其它 $f[1..V]$ 均设为 $-\infty$, 这样就可以保证最终得到的 $f[N]$ 是一种恰好装满背包的最优解。

如果没有要求必须把背包装满, 而是只希望价格尽量大, 初始化时应该将 $f[0..V]$ 全部设为 0。

为什么呢? 可以这样理解: 初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满, 那么此时只有容量为 0 的背包可能被价值为 0 的 nothing“恰好装满”, 其它容量的背包均没有合法的解, 属于未定义的状态, 它们的值就都应该是 $-\infty$ 了。如果背包并非必须被装满, 那么任何容量的背包都有一个合法解“什么都不装”, 这个解的价值为 0, 所以初始时状态的值也就全部为 0 了。

这个小技巧完全可以推广到其它类型的背包问题, 后面也就不再对进行状态转移之前的

初始化进行讲解。

小结

01 背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的[空间复杂度](#)。

[编辑本段](#)装箱问题

论述

有一个箱子容量为 V (正整数, $0 \leq V \leq 20000$)，同时有 n 个物品 ($0 < n \leq 30$)，每个物品有一个体积 (正整数)。要求从 n 个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入 v, n ，在输入 n 个物品。

输出箱子的剩余空间为最小。

Input:

24 一个整数，表示箱子容量

6 一个整数，表示有 n 个物品

8 接下来 n 行，分别表示这 n 个物品的各自体积。

3

12

7

9

7

Output:

0 一个整数，表示箱子剩余空间。

```
var v,n,i,j,k:longint;
f:array[0..20000]of boolean;
a:array[1..30]of longint;
begin
read(v,n);
for i:=1 to n do read(a[i]);
f[0]:=true;
for i:=1 to n do
for j:=v downto a[i] do
if not f[j] and f[j-a[i]] then
f[j]:=true;
k:=v;
while (k>1)and(not f[k]) do dec(k);
```

```
writeln(v-k);
```

```
end.
```

二次背包问题

二次背包问题是背包问题的一种推广形式:

1.18.4. 贪心算法

一. 贪心算法的基本概念

当一个问题具有最优子结构性质时，我们会想到用动态规划法去解它。但有时会有更简单有效的算法。我们来看一个找硬币的例子。假设有四种硬币，它们的面值分别为二角五分、一角、五分和一分。现在要找给某顾客六角三分钱。这时，我们会不假思索地拿出 2 个二角五分的硬币，1 个一角的硬币和 3 个一分的硬币交给顾客。这种找硬币方法与其他的找法相比，所拿出的硬币个数是最少的。这里，我们下意识地使用了这样的找硬币算法：首先选出一个面值不超过六角三分的最大硬币，即二角五分；然后从六角三分中减去二角五分，剩下三角八分；再选出一个面值不超过三角八分的最大硬币，即又一个二角五分，如此一直做下去。这个找硬币的方法实际上就是贪心算法。顾名思义，贪心算法总是作出在当前看来是最好的选择。也就是说贪心算法并不从整体最优上加以考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，我们希望贪心算法得到的最终结果也是整体最优的。上面所说的找硬币算法得到的结果就是一个整体最优解。找硬币问题本身具有最优子结构性质，它可以用动态规划算法来解。但我们看到，用贪心算法更简单，更直接且解题效率更高。这利用了问题本身的一些特性。例如，上述找硬币的算法利用了硬币面值的特殊性。如果硬币的面值改为一分、五分和一角一分 3 种，而要找给顾客的是一角五分钱。还用贪心算法，我们将找给顾客 1 个一角一分的硬币和 4 个一分的硬币。然而 3 个五分的硬币显然是最好的找法。虽然贪心算法不是对所有问题都能得到整体最优解，但对范围相当广的许多问题它能产生整体最优解。如图的单源最短路径问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，但其最终结果却是最优解的很好的近似解。

二. 求解活动安排问题算法

活动安排问题是可以用贪心算法有效求解的一个很好的例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

设有 n 个活动的集合 $e=\{1, 2, \dots, n\}$ ，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i ，且 $s_i < f_i$ 。如果选择了活动 i ，则它在半开时间区间 $[s_i, f_i]$ 内占用资源。若区间 $[s_i, f_i]$ 与区间 $[s_j, f_j]$ 不相交，则称活动 i 与活动 j 是相容的。也就是说，当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 i 与活动 j 相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

在下面所给出的解活动安排问题的贪心算法 `greadyselector` 中，各活动的起始时间和结束时间存储于数组 `s` 和 `f` 中且按结束时间的非减序： $f_1 \leq f_2 \leq \dots \leq f_n$ 排列。如果所给出的活动未按此序排列，我们可以用 $O(n \log n)$ 的时间将它重排。

```
template< class type>
void greadyselector(int n, type s[ 1, type f[ ], bool a[ ] ]
{ a[ 1 ] = true;
int j = 1;
for (int i=2;i<=n;i+ + ) {
    if (s[i]>=f[j]) {
        a[i] = true;
        j=i;
    }
    else a[i]= false;
}
}
```

算法 `greadyselector` 中用集合 `a` 来存储所选择的活动。活动 `i` 在集合 `a` 中，当且仅当 `a[i]` 的值为 `true`。变量 `j` 用以记录最近一次加入到 `a` 中的活动。由于输入的活动是按其结束时间的非减序排列的， f_j 总是当前集合 `a` 中所有活动的最大结束时间，即：

贪心算法 `greadyselector` 一开始选择活动 1，并将 `j` 初始化为 1。然后依次检查活动 `i` 是否与当前已选择的所有活动相容。若相容则将活动 `i` 加入到已选择活动的集合 `a` 中，否则不选择活动 `i`，而继续检查下一活动与集合 `a` 中活动的相容性。由于 f_i 总是当前集合 `a` 中所有活动的最大结束时间，故活动 `i` 与当前集合 `a` 中所有活动相容的充分且必要的条件是其开始时间 `s` 不早于最近加入集合 `a` 中的活动 `j` 的结束时间 `f_j`， $s \geq f_j$ 。若活动 `i` 与之相容，则 `i` 成为最近加入集合 `a` 中的活动，因而取代活动 `j` 的位置。由于输入的活动是以其完成时间的非减序排列的，所以算法 `greadyselector` 每次总是选择具有最早完成时间的相容活动加入集合 `a` 中。直观上按这种方法选择相容活动就为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。算法 `greadyselector` 的效率极高。当输入的活动已按结束时间的非减序排列，算法只需 $O(n)$ 的时间来安排 `n` 个活动，使最多的活动能相容地使用公共资源。

例：设待安排的 11 个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

算法 `greadyselector` 的计算过程如图所示。

图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合 a 中的活动，而空白长条表示的活动是当前正在检查其相容性的活动。若被检查的活动 i 的开始时间 s_i 小于最近选择的活动了的结束时间 f_j ，则不选择活动 i ，否则选择活动 i 加入集合 a 中。

三. 算法分析

贪心算法并不总能求得问题的整体最优解。但对于活动安排问题，贪心算法 greedyselector 却总能求得的整体最优解，即它最终所确定的相容活动集合 a 的规模最大。我们可以用数学归纳法来证明这个结论。

事实上，设 $e=\{1, 2, \dots, n\}$ 为所给的活动集合。由于正中活动按结束时间的非减序排列，故活动 1 具有最早的完成时间。首先我们要证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动 1。设 a 是所给的活动安排问题的一个最优解，且 a 中活动也按结束时间非减序排列， a 中的第一个活动是活动 k 。若 $k=1$ ，则 a 就是一个以贪心选择开始的最优解。若 $k>1$ ，则我们设 b 是 a 去掉活动 k 后的活动安排。由于 $f_1 \leq f_k$ ，且 a 中活动是互为相容的，故 b 中的活动也是互为相容的。又由于 b 中活动个数与 a 中活动个数相同，且 a 是最优的，故 b 也是最优的。也就是说 b 是一个以贪心选择活动 1 开始的最优活动安排。因此，我们证明了总存在一个以贪心选择开始的最优活动安排方案。

进一步，在作了贪心选择，即选择了活动 1 后，原问题就简化为对 e 中所有与活动 1 相容的活动进行活动安排的子问题。即若 a 是原问题的一个最优解，则 $a'=a-\{1\}$ 是活动安排问题的一个最优解。事实上，如果我们能找到 a' 的一个解 b' ，它包含比 a' 更多的活动，则将活动 1 加入到 b' 中将产生 e 的一个解 b ，它包含比 a 更多的活动。这与 a 的最优性矛盾。因此，每一步所作的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。对贪心选择次数用数学归纳法即知，贪心算法 greedyselector 最终产生原问题的一个最优解。

四. 贪心算法的基本要素

贪心算法通过一系列的选择来得到一个问题的解。它所作的每一个选择都是当前状态下某种意义的最好选择，即贪心选择。希望通过每次所作的贪心选择导致最终结果是问题的一个最优解。这种启发式的策略并不总能奏效，然而在许多情况下确能达到预期的目的。解活动安排问题的贪心算法就是一个例子。下面我们着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题，我们怎么知道是否可用贪心算法来解此问题，以及能否得到问题的一个最优解呢？这个问题很难给予肯定的回答。但是，从许多可以用贪心算法求解的问题中

我们看到它们一般具有两个重要的性质：贪心选择性质和最优子结构性质。

1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区

别。在动态规划算法中，每步所作的选择往往依赖于相关子问题的解。因而只有在解出相关子问题后，才能作出选择。而在贪心算法中，仅在当前状态下作出最好选择，即局部最优选择。然后再去解作出这个选择后产生的相应的子问题。贪心算法所作的贪心选择可以依赖于以往所作过的选择，但决不依赖于将来所作的选择，也不依赖于子问题的解。正是由于这种差别，动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为一个规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，我们必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。通常可以用我们在证明活动安排问题的贪心选择性质时所采用的方法来证明。首先考察问题的一个整体最优解，并证明可修改这个最优解，使其以贪心选择开始。而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步作贪心选择，最终可得到问题的一个整体最优解。其中，证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

2. 最优子结构性质

当一个问题的最优解包含着它的子问题的最优解时，称此问题具有最优子结构性质。问题所具有的这个性质是该问题可用动态规划算法或贪心算法求解的一个关键特征。在活动安排问题中，其最优子结构性质表现为：若 a 是对于正的活动安排问题包含活动 1 的一个最优解，则相容活动集合 $a' = a - \{1\}$ 是对于 $e' = \{i \in e : s_i \geq f_1\}$ 的活动安排问题的一个最优解。

3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。但是，对于一个具有最优子结构的问题应该选用贪心算法还是动态规划算法来求解？是不是能用动态规划算法求解的问题也能用贪心算法来求解？下面我们来研究两个经典的组合优化问题，并以此来说明贪心算法与动态规划算法的主要差别。

五. 0-1 背包问题

给定 n 种物品和一个背包。物品 i 的重量是 w_i ，其价值为 v_i ，背包的容量为 c 。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i 。

此问题的形式化描述是，给定 $c > 0$, $w_i > 0$, $v_i > 0$, $1 \leq i \leq n$, 要求找出一个 n 元 0-1 向量 (x_1, x_2, \dots, x_n) , 使得 $\sum w_i x_i \leq c$, 而且达到最大。

背包问题：与 0-1 背包问题类似，所不同的是在选择物品 i 装入背包时，可以选择物品 i 的一部分，而不一定要全部装入背包。

此问题的形式化描述是，给定 $c > 0$, $w_i > 0$, $v_i > 0$, $1 \leq i \leq n$, 要求找出一个 n 元向量 (x_1, x_2, \dots, x_n) , $0 \leq x_i \leq 1$, $1 \leq i \leq n$ 使得 $\sum v_i x_i \leq c$, 而且达到最大。

这两类问题都具有最优子结构性质。对于 0—1 背包问题，设 a 是能够装入容量为 c 的背包的具有最大价值的物品集合，则 $a_j = a - \{j\}$ 是 $n-1$ 个物品 $1, 2, \dots, j-1, j+1, \dots, n$ 可装入容量为 $c-w_i$ 叫的背包的具有最大价值的物品集合。对于背包问题，类似地，若它的一个最优解包含物品 j ，则从该最优解中拿出所含的物品 j 的那部分重量 w_i ，剩余的将是 $n-1$ 个原重物品 $1, 2, \dots, j-1, j+1, \dots, n$ 以及重为 w_j-w_i 的物品 j 中可装入容量为 $c-w$ 的背包且具有最大价值的物品。

虽然这两个问题极为相似，但背包问题可以用贪心算法求解，而 0-1 背包问题却不能用贪心算法求解。用贪心算法解背包问题的基本步骤是，首先计算每种物品单位重量的价值 v_j / w_i 然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过 c ，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直进行下去直到背包装满为止。具体算法可描述如下：

```
void knapsack(int n, float m, float v[], float w[], float x[])
{
    sort(n, v, w);
    int i;
    for(i= 1;i<= n;i++) x[i] = 0;
    float c = m;
    for (i = 1;i <= n;i++) {
        if (w[i] > c) break;
        x[i] = 1;
        c -= w[i];
    }
    if (i <= n) x[i] = c/w[i];
}
```

算法 knapsack 的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为 $O(n \log n)$ 。当然，为了证明算法的正确性，我们还必须证明背包问题具有贪心选择性质。

这种贪心选择策略对 0—1 背包问题就不适用了。看图 2(a) 中的例子，背包的容量为 50 千克；物品 1 重 10 千克，价值 60 元；物品 2 重 20 千克，价值 100 元；物品 3 重 30 千克，价值 120 元。因此，物品 1 每千克价值 6 元，物品 2 每千克价值 5 元，物品 3 每千克价值 4 元。若依贪心选择策略，应首选物品 1 装入背包，然而从图 2(b) 的各种情况可以看出，最优的选择方案是选择物品 2 和物品 3 装入背包。首选物品 1 的两种方案都不是最优的。对于背包问题，贪心选择最终可得到最优解，其选择方案如图 2(c) 所示。

对于 0—1 背包问题，贪心选择之所以不能得到最优解是因为它无法保证最终能将背包装满，部分背包空间的闲置使每千克背包空间所具有的价值降低了。事实上，在考虑 0—1 背

包问题的物品选择时，应比较选择该物品和不选择该物品所导致的最终结果，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。动态规划算法的确可以有效地解 0—1 背包问题。

1.18.5. 装箱问题

装箱问题

有一个箱子容量为 v （正整数， $0 \leq v \leq 20000$ ），同时有 n 个物品（ $0 < n \leq 30$ ），每个物品有一个体积（正整数）。

要求从 n 个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入：

箱子的容量 v

物品数 n

接下来 n 行，分别表示这 n 个物品的体积

输出：

箱子剩余空间

输入输出样例

输入： 24

6

8

3

12

7

9

7

输出： 0

题解：

1. 使用回溯法计算箱子的最小剩余空间

容量为 v 的箱子究竟应该装入哪些物品可使得剩余空间最小。显然在 n 个物品的体积和小于等于 v 的情况下，剩余空间为 $v-n$ 个物品的体积和。但在 n 个物品的体积和大于 v 的情况下，没有一种可以直接找到问题解的数学方法。无奈之下，只能采用搜索的办法。设 a 和 s 为箱子的体积序列。其中 $a[i]$ 为箱子 i 的体积， $s[i]$ 为前 i 个箱子的体积和 ($1 \leq i \leq n$)；
 $best$ 为目前所有的装箱方案中最小的剩余空间。初始时 $best=v$ ；

确定搜索的几个关键因素：

状态 (k, v') : 其中 k 为待装入箱子的物品序号， v' 为箱子目前的剩余空间。

目标 $v' < best$: 若箱子的剩余空间为目前为止最小，则 $best$ 调整为 v' ($best \leftarrow v'$)；

边界条件 $(v' - (s[n] - s[k-1])) \geq best$: 即便剩下的物品全部装入箱子(未装物品的体积和为 $s[n] - s[k-1]$)，其剩余空间仍不小于 $best$ ，则放弃当前方案，回溯；

搜索范围: 在未装入全部物品的前提下 ($k \leq n$)，搜索两种可能情况：

- 1 若剩余空间装得下物品 k ($v' \geq a[k]$)，则物品 k 装入箱子，递归计算子状态 $(k+1, v' - a[k])$ ；
- 1 物品 k 不装入箱子，递归计算子状态 $(k+1, v')$ ；

我们用递归过程 $search(k, v)$ 描述这一搜索过程：

```
procedure search(k, v:integer); {从状态(k, v)出发，递归计算最小剩余空间}
```

```
begin
  if  $v < best$  then  $best \leftarrow v$ ; {若剩余空间为目前最小，则调整 best}
  if  $v - (s[n] - s[k-1]) \geq best$  {若箱子即便装下全部物品，其剩余空间仍不小于 best，则回溯}
    then exit;
  if  $k \leq n$  {在未装入全部物品的前提下搜索两种可能情况}
    then begin
      if  $v \geq a[k]$  {若剩余空间装得下物品 k，则物品 k 装入箱子，递归计算子状态}
        then search( $k+1, v - a[k]$ );
      search( $k+1, v$ ); {物品 k 不装入箱子，递归计算子状态}
    end; {then}
  end; {search}
```

主程序如下：

读箱子体积 v ；

读物品个数 n ；

```
s[0]  $\leftarrow 0$ ; {物品装入前初始化}
for i  $\leftarrow 1$  to n do {输入和计算箱子的体积序列}
begin
  读第 i 个箱子的体积  $a[i]$ ;
   $s[i] \leftarrow s[i-1] + a[i]$ ;
end; {for}
best  $\leftarrow v$ ; {初始时，最小剩余空间为箱子体积}
if  $s[n] \leq v$  then  $best \leftarrow v - s[n]$  {若所有物品能全部装入箱子，则剩余空间为问题解}
else search(1, v); {否则从物品 1 出发，递归计算最小剩余空间}
输出最小剩余空间 best;
```

2. 使用动态程序设计方法计算箱子的最小剩余空间

如果按照物品序号依次考虑装箱顺序的话，则问题具有明显的阶段特征。问题是当前阶段的剩余空间最小，并不意味下一阶段的剩余空间也一定最小，即该问题并不具备最优子结构的特征。但如果将装箱的体积作为状态的话，则阶段间的状态转移关系顺其自然，可使得最优化问题变为判定性问题。设状态转移方程

$f[i, j]$ ——在前 i 个物品中选择若干个物品（必须包括物品 i ）装箱，其体积正好为 j 的标志。显然 $f[i, j] = f[i-1, j - \text{box}[i]]$ ，即物品 i 装入箱子后的体积正好为 j 的前提是 $f[i-1, j - \text{box}[i]] = \text{true}$ 。初始时， $f[0, 0] = \text{true}$ ($1 \leq i \leq n, \text{box}[i] \leq j \leq v$)。

由 $f[i, j] = f[i-1, j - \text{box}[i]]$ 可以看出，当前阶段的状态转移方程仅与上一阶段的状态转移方程攸关。因此设 f_0 为 $i-1$ 阶段的状态转移方程， f_1 为 i 阶段的状态转移方程，这样可以将二维数组简化成一维数组。我们按照下述方法计算状态转移方程 f_1 ：

```
fillchar(f0, sizeof(f0), 0);           {装箱前，状态转移方程初始化}
f0[0]←true;
for i←1 to n do          {阶段 i:按照物品数递增的顺序考虑装箱情况}
begin
    f1←f0;                  {i 阶段的状态转移方程初始化}
    for j←box[i] to v do    {状态 j: 枚举所有可能的装箱体积}
        if f0[j-box[i]] then f1[j]←true; {若物品 i 装入箱子后的体积正好为 j, 则物品 i 装入箱子}
        f0←f1;                  {记下当前装箱情况}
    end; {for}
```

经过上述运算，最优化问题转化为判定性问题。再借用动态程序设计的思想，计算装箱的最大体积 $k = \dots$ 。显然最小剩余空间为 $v-k$ ：

```
for i←v downto 0 do      {按照递减顺序枚举所有可能的体积}
    if f1[i] then begin {若箱子能装入体积为 i 的物品，则输出剩余空间 v-i，并退出程序}
        writeln(v-i);   halt
    end; {then}
end. {for}
writeln(v);      {在未装入一个物品的情况下输出箱子体积}
```

1.19. 教你如何迅速秒杀掉：99%的海量数据处理面试题

作者：July

出处：结构之法算法之道 blog

前言

一般而言，标题含有“秒杀”，“99%”，“史上最全/最强”等词汇的往往都脱不了哗众取宠之嫌，但进一步来讲，如果读者读罢此文，却无任何收获，那么，我也甘愿背负这样的罪名，:-)，同时，此文可以看做是对这篇文章：[十道海量数据处理面试题与十个方法大总结的一般抽象性总结](#)。

毕竟受文章和理论之限，本文将摒弃绝大部分的细节，只谈方法/模式论，且注重用最通俗最直白的语言阐述相关问题。最后，有一点必须强调的是，全文行文是基于面试题的分析基础之上的，具体实践过程中，还是得具体情况具体分析，且场景也远比本文所述的任何一种情况复杂得多。

OK，若有任何问题，欢迎随时不吝赐教。谢谢。

何谓海量数据处理？

所谓海量数据处理，无非就是基于海量数据上的存储、处理、操作。何谓海量，就是数据量太大，所以导致要么是无法在较短时间内迅速解决，要么是数据太大，导致无法一次性装入内存。

那解决办法呢？针对时间，我们可以采用巧妙的算法搭配合适的数据结构，如 Bloom filter/Hash/bit-map/堆/数据库或倒排索引/trie 树，针对空间，无非就一个办法：大而化小：分而治之/hash 映射，你不是说规模太大嘛，那简单啊，就把规模大化为规模小的，各个击破不就完了嘛。

至于所谓的单机及集群问题，通俗点来讲，单机就是处理装载数据的机器有限(只要考虑 cpu，内存，硬盘的数据交互)，而集群，机器有多辆，适合分布式处理，并行计算(更多考虑节点和节点间的数据交互)。

再者，通过本 blog 内的有关海量数据处理的文章：[Big Data Processing](#)，我们已经大致知道，处理海量数据问题，无非就是：

1. 分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序；
2. 双层桶划分
3. Bloom filter/Bitmap；
4. Trie 树/数据库/倒排索引；
5. 外排序；
6. 分布式处理之 Hadoop/Mapreduce。

下面，本文第一部分、从 set/map 谈到 hashtable/hash_map/hash_set，简要介绍下 set/map/multiset/multimap，及 hash_set/hash_map/hash_multiset/hash_multimap 之区别(万丈高楼平地起，基础最重要)，而本文第二部分，则针对上述那 6 种方法模式结合对应的海量数据处理面试题分别具体阐述。

第一部分、从 set/map 谈到 hashtable/hash_map/hash_set

稍后本文第二部分中将多次提到 hash_map/hash_set，下面稍稍介绍下这些容器，以作为基础准备。一般来说，STL 容器分两种，

- 序列式容器(vector/list/deque/stack/queue/heap)，
- 关联式容器。关联式容器又分为 set(集合)和 map(映射表)两大类，以及这两大类的衍生体 multiset(多键集合)和 multimap(多键映射表)，这些容器均以 RB-tree 完成。此外，还有第 3 类关联式容器，如 hashtable(散列表)，以及以 hashtable 为底层机制完成的 hash_set(散列集合)/hash_map(散列映射表)/hash_multiset(散列多键集合)/hash_multimap(散列多键映射表)。也就是说，set/map/multiset/multimap 都内含一个 RB-tree，而 hash_set/hash_map/hash_multiset/hash_multimap 都内含一个 hashtable。

所谓关联式容器，类似关联式数据库，每笔数据或每个元素都有一个键值(key)和一个实值(value)，即所谓的 Key-Value(键-值对)。当元素被插入到关联式容器中时，容器内部结构(RB-tree/hashtable)便依照其键值大小，以某种特定规则将这个元素放置于适当位置。

包括在非关联式数据库中，比如，在 MongoDB 内，文档(document)是最基本的数据组织形式，每个文档也是以 Key-Value(键-值对)的方式组织起来。一个文档可以有多个 Key-Value 组合，每个 Value 可以是不同的类型，比如 String、Integer、List 等等。

```
{ "name" : "July",  
  "sex" : "male",  
  "age" : 23 }
```

set/map/multiset/multimap

set，同 map 一样，所有元素都会根据元素的键值自动被排序，因为 set/map 两者的所有各种操作，都只是转而调用 RB-tree 的操作行为，不过，值得注意的是，两者都不允许两个元素有相同的键值。

不同的是：set 的元素不像 map 那样可以同时拥有实值(value)和键值(key)，set 元素的键值就是实值，实值就是键值，而 map 的所有元素都是 pair，同时拥有实值(value)和键值(key)，pair 的第一个元素被视为键值，第二个元素被视为实值。

至于 multiset/multimap，他们的特性及用法和 set/map 完全相同，唯一的差别就在于它们允许键值重复，即所有的插入操作基于 RB-tree 的 insert_equal()而非 insert_unique()。

hash_set/hash_map/hash_multiset/hash_multimap

hash_set/hash_map，两者的一切操作都是基于 hashtable 之上。不同的是，hash_set 同 set 一样，同时拥有实值和键值，且实质就是键值，键值就是实值，而 hash_map 同 map 一样，每一个元素同时拥有一个实值(value)和一个键值(key)，所以其使用方式，和上面的 map 基本相同。但由于 hash_set/hash_map 都是基于 hashtable 之上，所以不具备自动排序功能。为什么？因为 hashtable 没有自动排序功能。

至于 `hash_multiset`/`hash_multimap` 的特性与上面的 `multiset`/`multimap` 完全相同，唯一的差别就是它们 `hash_multiset`/`hash_multimap` 的底层实现机制是 `hashtable`(而 `multiset`/`multimap`，上面说了，底层实现机制是 `RB-tree`)，所以它们的元素都不会被自动排序，不过也都允许键值重复。

所以，综上，说白了，什么样的结构决定其什么样的性质，因为 `set`/`map`/`multiset`/`multimap` 都是基于 `RB-tree` 之上，所以有自动排序功能，而 `hash_set`/`hash_map`/`hash_multiset`/`hash_multimap` 都是基于 `hashtable` 之上，所以不含有自动排序功能，至于加个前缀 `multi_` 无非就是允许键值重复而已。

此外，

- 关于什么 `hash`，请看 blog 内此篇文章：
http://blog.csdn.net/v_JULY_v/article/details/6256463；
- 关于红黑树，请参看 blog 内系列文章：
http://blog.csdn.net/v_july_v/article/category/774945，
- 关于 `hash_map` 的具体应用：<http://blog.csdn.net/sdhongjun/article/details/4517325>，关于 `hash_set`：<http://blog.csdn.net/morewindows/article/details/7330323>。

OK，接下来，请看本文第二部分、处理海量数据问题之六把密匙。

第二部分、处理海量数据问题之六把密匙

密匙一、分而治之/Hash 映射 + Hash 统计 + 堆/快速/归并排序

1、海量日志数据，提取出某日访问百度次数最多的那个 IP。

既然是海量数据处理，那么可想而知，给我们的数据那就一定是海量的。针对这个数据的海量，我们如何着手呢？对的，无非就是分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序，说白了，就是先映射，而后统计，最后排序：

1. 分而治之/hash 映射：针对数据太大，内存受限，只能是：把大文件化成(取模映射)小文件，即 16 字方针：大而化小，各个击破，缩小规模，逐个解决
2. hash 统计：当大文件转化了小文件，那么我们便可以采用常规的 `hash_map(ip, value)` 来进行频率统计。
3. 堆/快速排序：统计完了之后，便进行排序(可采取堆排序)，得到次数最多的 IP。

具体而论，则是：“首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。注意到 IP 是 32 位的，最多有个 2^{32} 个 IP。同样可以采用映射的方法，比如模 1000，把整个大文件映射为 1000 个小文件，再找出每个小文中出现频率最大的 IP（可以采用 `hash_map` 进行频率统计，然后再找出

频率最大的几个)及相应的频率。然后再在这 1000 个最大的 IP 中,找出那个频率最大的 IP,即为所求。”--

十道海量数据处理面试题与十个方法大总结。

关于本题,还有几个问题,如下:

1、Hash 取模是一种等价映射,不会存在同一个元素分散到不同小文件中去的情况,即这里采用的是 mod1000 算法,那么相同的 IP 在 hash 后,只可能落在同一个文件中,不可能被分散的。

2、那到底什么是 hash 映射呢?简单来说,就是为了便于计算机在有限的内存中处理 big 数据,从而通过一种映射散列的方式让数据均匀分布在对应的内存位置(如大数据通过取余的方式映射成小树存放在内存中,或大文件映射成多个小文件),而这个映射散列方式便是我们通常所说的 hash 函数,设计的好 hash 函数能让数据均匀分布而减少冲突。尽管数据映射到了另外一些不同的位置,但数据还是原来的数据,只是代替和表示这些原始数据的形式发生了变化而已。

此外,有一朋友 quicktest 用 python 语言实践测试了下本题,地址如下:

<http://blog.csdn.net/quicktest/article/details/7453189>。谢谢。OK,有兴趣的,还可以再了解下一致性 hash 算法,见 blog 内此文第五部分: http://blog.csdn.net/v_july_v/article/details/6879101。

2、寻找热门查询: 搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来,每个查询串的长度为 1-255 字节。

假设目前有一千万个记录(这些查询串的重复度比较高,虽然总数是 1 千万,但如果除去重复后,不超过 3 百万个。一个查询串的重复度越高,说明查询它的用户越多,也就是越热门),请你统计最热门的 10 个查询串,要求使用的内存不能超过 1G。

由上面第 1 题,我们知道,数据大则划为小的,但如果数据规模比较小,能一次性装入内存呢?比如这第 2 题,虽然有一千万个 Query,但是由于重复度比较高,因此事实上只有 300 万的 Query,每个 Query255Byte,因此我们可以考虑把他们都放进内存中去,而现在只是需要一个合适的数据结构,在这里, Hash Table 绝对是我们优先的选择。所以我们放弃分而治之/hash 映射的步骤,直接上 hash 统计,然后排序。So,

1. hash 统计: 先对这批海量数据预处理(维护一个 Key 为 Query 字串, Value 为该 Query 出现次数的 HashTable, 即 hash_map(Query, Value), 每次读取一个 Query, 如果该字串不在 Table 中,那么加入该字串,并且将 Value 值设为 1; 如果该字串在 Table 中,那么将该字串的计数加一即可。最终我们在 O(N) 的时间复杂度内用 Hash 表完成了统计;
2. 堆排序: 第二步、借助堆这个数据结构,找出 Top K, 时间复杂度为 N'logK。即借助堆结构, 我们可以在 log 量级的时间内查找和调整/移动。因此,维护一个 K(该题目中是 10)大小的小根堆,然后遍历 300 万的 Query, 分别和根元素进行对比所以,我们最终的时间复杂度是: $O(N) + N'*O(\log K)$, (N 为 1000 万, N' 为 300 万)。

别忘了这篇文章中所述的堆排序思路：“维护 k 个元素的最小堆，即用容量为 k 的最小堆存储最先遍历到的 k 个数，并假设它们即是最大的 k 个数，建堆费时 $O(k)$ ，并调整堆（费时 $O(\log k)$ ）后，有 $k_1 > k_2 > \dots > k_{\min}$ (k_{\min} 设为小顶堆中最小元素)。继续遍历数列，每次遍历一个元素 x ，与堆顶元素比较，若 $x > k_{\min}$ ，则更新堆（用时 $\log k$ ），否则不更新堆。这样下来，总费时 $O(k \cdot \log k + (n-k) \cdot \log k) = O(n \cdot \log k)$ 。此方法得益于在堆中，查找等各项操作时间复杂度均为 \log 。”--第三章续、Top K 算法问题的实现。

当然，你也可以采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的最小堆来对出现频率进行排序。

3、有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M。返回频数最高的 100 个词。

由上面那两个例题，分而治之 + hash 统计 + 堆/快速排序这个套路，我们已经开始有了屡试不爽的感觉。下面，再拿几道再多多验证下。请看此第 3 题：又是文件很大，又是内存受限，咋办？还能怎么办呢？无非还是：

1. 分而治之/hash 映射：顺序读文件中，对于每个词 x ，取 $\text{hash}(x) \% 5000$ ，然后按照该值存到 5000 个小文件（记为 $x_0, x_1, \dots, x_{4999}$ ）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过 1M。
2. hash 统计：对每个小文件，采用 trie 树/hash_map 等统计每个文件中出现的词以及相应的频率。
3. 堆/归并排序：取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆），并把 100 个词及相应的频率存入文件，这样又得到了 5000 个文件。最后就是把这 5000 个文件进行归并（类似于归并排序）的过程了。

4、海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。

此题与上面第 3 题类似，

1. 堆排序：在每台电脑上求出 TOP10，可以采用包含 10 个元素的堆完成（TOP10 小，用最大堆，TOP10 大，用最小堆）。比如求 TOP10 大，我们首先取前 10 个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是 TOP10 大。
2. 求出每台电脑上的 TOP10 后，然后把这 100 台电脑上的 TOP10 组合起来，共 1000 个数据，再利用上面类似的方法求出 TOP10 就可以了。

上述第 4 题的此解法，经读者反应有问题，如举个例子如求 2 个文件中的 top2，照上述算法，如果第一个文件里有：

a 49 次

- b 50 次
- c 2 次
- d 1 次

第二个文件里有：

- a 9 次
- b 1 次
- c 11 次
- d 10 次

虽然第一个文件里出来 top2 是 b (50 次), a (49 次), 第二个文件里出来 top2 是 c (11 次), d (10 次), 然后 2 个 top2: b (50 次) a (49 次) 与 c (11 次) d (10 次) 归并, 则算出所有的文件的 top2 是 b(50 次), a(49 次), 但实际上是 a(58 次), b(51 次)。是否真是如此呢? 若真如此, 那作何解决呢?

正如老梦所述:

首先, 先把所有的数据遍历一遍做一次 hash(保证相同的数据条目划分到同一台电脑上进行运算), 然后根据 hash 结果重新分布到 100 台电脑中, 接下来的算法按照之前的即可。

最后由于 a 可能出现在不同的电脑, 各有一定的次数, 再对每个相同条目进行求和 (由于上一步骤中 hash 之后, 也方便每台电脑只需要对自己分到的条目内进行求和, 不涉及到别的电脑, 规模缩小)。

5、有 10 个文件, 每个文件 1G, 每个文件的每一行存放的都是用户的 query, 每个文件的 query 都可能重复。要求你按照 query 的频度排序。

直接上:

1. hash 映射: 顺序读取 10 个文件, 按照 $\text{hash(query)} \% 10$ 的结果将 query 写入到另外 10 个文件 (记为 ) 中。这样新生成的文件每个的大小大约也 1G (假设 hash 函数是随机的)。
2. hash 统计: 找一台内存 2G 左右的机器, 依次对用 `hash_map(query, query_count)` 来统计每个 query 出现的次数。注: `hash_map(query, query_count)` 是用来统计每个 query 的出现次数, 不是存储他们的值, 出现一次, 则 `count+1`。
3. 堆/快速/归并排序: 利用快速/堆/归并排序按照出现次数进行排序, 将排序好的 query 和对应的 `query_count` 输出到文件中, 这样得到了 10 个排好序的文件 (记为 )。最后, 对这 10 个文件进行归并排序 (内排序与外排序相结合)。

除此之外, 此题还有以下两个方法:

方案 2: 一般 query 的总量是有限的, 只是重复的次数比较多而已, 可能对于所有的 query, 一次性就可以加入到内存了。这样, 我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数, 然后按出现次数做快速/堆/归并排序就可以了。

方案 3：与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

6、给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url？

可以估计每个文件的大小为 $5G \times 64 = 320G$ ，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

1. 分而治之/hash 映射：遍历文件 a，对每个 url 求取 ，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为 ）中。这样每个小文件的大约 300M。遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 小文件中（记为 ）。这样处理后，所有可能相同的 url 都在对应的小文件（）中，不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。
2. hash 统计：求每对小文件中相同的 url 时，可以把其中一个小文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。

OK，此第一种方法：分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序，再看最后 4 道题，如下：

7、怎么在海量数据中找出重复次数最多的一个？

方案 1：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

8、上千万或上亿数据（有重复），统计其中出现次数最多的 N 个数据。

方案 1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用 hash_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前 N 个出现次数最多的数据了，可以用第 2 题提到的堆机制完成。

9、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析。

方案 1：这题是考虑时间效率。用 trie 树统计每个词出现的次数，时间复杂度是 $O(n*le)$ (le 表示单词的平均长度)。然后是找出出现最频繁的前 10 个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n*lg10)$ 。所以总的时间复杂度，是 $O(n*le)$ 与 $O(n*lg10)$ 中较大的哪一个。

10. 1000 万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

- 方案 1：这题用 trie 树比较合适，hash_map 也行。
- 方案 2：from xjbzju：1000w 的数据规模插入操作完全不现实，以前试过在 stl 下 100w 元素插入 set 中已经慢得不能忍受，觉得基于 hash 的实现不会比红黑树好太多，使用 vector+sort+unique 都要可行许多，建议还是先 hash 成小文件分开处理再综合。

上述方案 2 中读者 xjbzju 的方法让我想到了一些问题，即是 **set/map，与 hash_set/hash_map 的性能比较**?共计 3 个问题，如下：

- 1、hash_set 在千万级数据下，insert 操作优于 set？这位 blog：<http://t.cn/zOibP7t> 给的实践数据可靠不？
- 2、那 map 和 hash_map 的性能比较呢？谁做过相关实验？



- 3、那查询操作呢，如下段文字所述？



或者小数据量时用 map，构造快，大数据量时用 hash_map？

rbtree PK hashtable

据朋友邦卡猫做的红黑树和 hash table 的性能测试中发现：当数据量基本上 int 型 key 时，hash table 是 rbtree 的 3-4 倍，但 hash table 一般会浪费大概一半内存。

因为 hash table 所做的运算就是个%，而 rbtree 要比较很多，比如 rbtree 要看 value 的数据，每个节点要多出 3 个指针（或者偏移量）如果需要其他功能，比如，统计某个范围内的 key 的数量，就需要加一个计数成员。

且 1s rbtree 能进行大概 50w+ 次插入，hash table 大概是差不多 200w 次。不过很多时候，其速度可以忍了，例如倒排索引差不多也是这个速度，而且单线程，且倒排表的拉链长度不会太大。正因为基于树的实现其实不比 hashtable 慢到哪里去，所以数据库的索引一般都是用的 B/B+ 树，而且 B+ 树还对磁盘友好（B 树能有效降低它的高度，所以减少磁盘交互次数）。比如现在非常流行的 NoSQL 数据库，像 MongoDB 也是采用的 B 树索引。关于 B 树系列，请参考本 blog 内此篇文章：[从 B 树、B+ 树、B* 树谈到 R 树](#)。

OK，更多请待后续实验论证。接下来，咱们来看第二种方法，双层桶划分。
密匙二、双层桶划分

双层桶划分----其实本质上还是分而治之的思想，重在“分”的技巧上！

适用范围：第 k 大，中位数，不重复或重复的数字

基本原理及要点：因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子。

扩展：

问题实例：

11、2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

有点像鸽巢原理，整数个数为 2^{32} ，也就是，我们可以将这 2^{32} 个数，划分为 2^8 个区域（比如用单个文件代表一个区域），然后将数据分离到不同的区域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

12、5亿个int找它们的中位数。

这个例子比上面那个更明显。首先我们将int划分为 2^{16} 个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成 2^{24} 个区域，然后确定区域的第几大数，在将该区域分成 2^{20} 个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有 2^{20} ，就可以直接利用direct addr table进行统计了。

密匙三：Bloom filter/Bitmap

Bloom filter

关于什么是**Bloom filter**，请参看blog内此文：

- [海量数据处理之 Bloom Filter 详解](#)

适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集

基本原理及要点：

对于原理来说很简单，位数组+k个独立hash函数。将hash函数对应的值的位数组置1，查找时如果发现所有hash函数对应位都是1说明存在，很明显这个过程并不保证查找的结果是100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是counting Bloom filter，用一个counter数组代替位数组，就可以支持删除了。

还有一个比较重要的问题，如何根据输入元素个数n，确定位数组m的大小及hash函数个数。当hash函数个数 $k=(\ln 2) * (m/n)$ 时错误率最小。在错误率不大于E的情况下，m至少要等于 $n \cdot \lg(1/E)$ 才能表示任意n个元素的集合。但m还应该更大些，因为还要保证bit数组里至少一半为0，则m应该 $\geq n \cdot \lg(1/E) \cdot 1.44$ 倍(lg表示以2为底的对数)。

举个例子我们假设错误率为 0.01，则此时 m 应大概是 n 的 13 倍。这样 k 大概是 8 个。

注意这里 m 与 n 的单位不同，m 是 bit 为单位，而 n 则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多 bit 的。所以使用 bloom filter 内存上通常都是节省的。

扩展：

Bloom filter 将集合中的元素映射到位数组中，用 k (k 为哈希函数个数) 个映射位是否全 1 表示元素在不在这个集合中。Counting bloom filter (CBF) 将位数组中的每一位扩展为一个 counter，从而支持了元素的删除操作。Spectral Bloom Filter (SBF) 将其与集合元素的出现次数关联。SBF 采用 counter 中的最小值来近似表示元素的出现频率。

13、给你 A,B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4G，让你找出 A,B 文件共同的 URL。如果是三个乃至 n 个文件呢？

根据这个问题我们来计算下内存的占用， $4G=2^{32}$ 大概是 40 亿*8 大概是 340 亿，n=50 亿，如果按出错率 0.01 算需要的大概是 650 亿个 bit。现在可用的是 340 亿，相差并不多，这样可能会使出错率上升些。另外如果这些 urlip 是一一对应的，就可以转换成 ip，则大大简单了。

同时，上文的第 5 题：给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url？如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。

Bitmap

- 关于什么是 Bitmap，请看 blog 内此文第二部分：

http://blog.csdn.net/v_july_v/article/details/6685962。

下面关于 Bitmap 的应用，直接上题，如下第 9、10 道：

14、在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数。

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存 $2^{32} * 2 \text{ bit}=1 \text{ GB}$ 内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。所描完事后，查看 bitmap，把对应位是 01 的整数输出即可。

方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

15、腾讯面试题：给 40 亿个不重复的 unsigned int 的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 40 亿个数当中？

方案 1: frome oo, 用位图(Bitmap)的方法, 申请 512M 的内存, 一个 bit 位代表一个 unsigned int 值。读入 40 亿个数, 设置相应的 bit 位, 读入要查询的数, 查看相应 bit 位是否为 1, 为 1 表示存在, 为 0 表示不存在。

密匙四、Trie 树/数据库/倒排索引

Trie 树

适用范围: 数据量大, 重复多, 但是数据种类小可以放入内存

基本原理及要点: 实现方式, 节点孩子的表示方式

扩展: 压缩实现。

问题实例:

1. 上面的**第 2 题**: 寻找热门查询: 查询串的重复度比较高, 虽然总数是 1 千万, 但如果除去重复后, 不超过 3 百万个, 每个不超过 255 字节。
2. 上面的**第 5 题**: 有 10 个文件, 每个文件 1G, 每个文件的每一行都存放的是用户的 query, 每个文件的 query 都可能重复。要你按照 query 的频度排序。
3. 1000 万字符串, 其中有些是相同的(重复), 需要把重复的全部去掉, 保留没有重复的字符串。请问怎么设计和实现?
4. 上面的**第 8 题**: 一个文本文件, 大约有一万行, 每行一个词, 要求统计出其中最频繁出现的前 10 个词。其解决方法是: 用 trie 树统计每个词出现的次数, 时间复杂度是 $O(n*le)$ (le 表示单词的平均长度), 然后是找出出现最频繁的前 10 个词。

更多有关 Trie 树的介绍, 请参见此文: [从 Trie 树\(字典树\)谈到后缀树](#)。

数据库索引

适用范围: 大数据量的增删改查

基本原理及要点: 利用数据的设计实现方法, 对海量数据的增删改查进行处理。

- 关于数据库索引及其优化, 更多可参见此文:
<http://www.cnblogs.com/pkuoliver/archive/2011/08/17/mass-data-topic-7-index-and-optimize.html>;
- 关于 MySQL 索引背后的数据结构及算法原理, 这里还有一篇很好的文章:
<http://www.codinglabs.org/html/theory-of-mysql-index.html>;
- 关于 B 树、B+ 树、B* 树及 R 树, 本 blog 内有篇绝佳文章:
http://blog.csdn.net/v_JULY_v/article/details/6530142。

倒排索引(Inverted index)

适用范围: 搜索引擎, 关键字查询

基本原理及要点: 为何叫倒排索引? 一种索引方法, 被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。

以英文为例，下面是要被索引的文本：

T0 = "it is what it is"

T1 = "what is it"

T2 = "it is a banana"

我们就能得到下面的反向文件索引：

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

检索的条件"what","is"和"it"将对应集合的交集。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

扩展：

问题实例：文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索。

关于倒排索引的应用，更多请参见：

- 第二十三、四章：杨氏矩阵查找，倒排索引关键词 Hash 不重复编码实践,
- 第二十六章：基于给定的文档生成倒排索引的编码与实践。

密匙五、外排序

适用范围：大数据的排序，去重

基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树

扩展：

问题实例：

1).有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 个字节，内存限制大小是 1M。返回频数最高的 100 个词。

这个数据具有很明显的特点，词的大小为 16 个字节，但是内存只有 1M 做 hash 明显不够，所以可以用来排序。内存可以当输入缓冲区使用。

关于多路归并算法及外排序的具体应用场景，请参见 blog 内此文：

- 第十章、如何给 10^7 个数据量的磁盘文件排序

密匙六、分布式处理之 Mapreduce

MapReduce 是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。但如果你要我再通俗点介绍，那么，说白了，Mapreduce 的原理就是一个归并排序。

适用范围：数据量大，但是数据种类小可以放入内存

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

扩展：

问题实例：

1. The canonical example application of MapReduce is a process to count the appearances of each different word in a set of documents:
2. 海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。
3. 一共有 N 个机器，每个机器上有 N 个数。每个机器最多存 O(N) 个数并对它们操作。
如何找到 N^2 个数的中数(median)?

更多具体阐述请参见 blog 内：

- 从 [Hadoop 框架与 MapReduce 模式中谈海量数据处理](#),
- 及 [MapReduce 技术的初步了解与学习](#)。

其它模式/方法论，结合操作系统知识

至此，六种处理海量数据问题的模式/方法已经阐述完毕。据观察，这方面的面试题无外乎以上一种或其变形，然题目为何取为是：秒杀 99% 的海量数据处理面试题，而不是 100% 呢。OK，给读者看最后一道题，如下：

非常大的文件，装不进内存。每行一个 int 类型数据，现在要你随机取 100 个数。

我们发现上述这道题，无论是以上任何一种模式/方法都不好做，那有什么好的别的方法呢？我们可以看看：操作系统内存分页系统设计(说白了，就是映射+建索引)。

Windows 2000 使用基于分页机制的虚拟内存。每个进程有 4GB 的虚拟地址空间。基于分页机制，这 4GB 地址空间的一些部分被映射了物理内存，一些部分映射硬盘上的交换文件，一些部分什么也没有映射。程序中使用的都是 4GB 地址空间中的虚拟地址。而访问物理内存，需要使用物理地址。关于什么是物理地址和虚拟地址，请看：

- 物理地址 (physical address): 放在寻址总线上的地址。放在寻址总线上，如果是读，电路根据这个地址每位的值就将相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个 地址每位的值就将相应地址的物理内存中放入数据总线上内容。物理内存是以字节(8 位)为单位编址的。

- 虚拟地址 (virtual address): 4G 虚拟地址空间中的地址，程序中使用的都是虚拟地址。使用了分页机制之后，4G 的地址空间被分成了固定大小的页，每一页或者被映射到物理内存，或者被映射到硬盘上的交换文件中，或者没有映射任何东西。对于一般程序来说，4G 的地址空间，只有一小部分映射了物理内存，大片大片的部分是没有映射任何东西。物理内存也被分页，来映射地址空间。对于 32bit 的 Win2k，页的大小是 4K 字节。CPU 用来把虚拟地址转换成物理地址的信息存放在叫做页目录和页表的结构里。

物理内存分页，一个物理页的大小为 4K 字节，第 0 个物理页从物理地址 0x00000000 处开始。由于页的大小为 4KB，就是 0x1000 字节，所以第 1 页从物理地址 0x00001000 处开始。第 2 页从物理地址 0x00002000 处开始。可以看到由于页的大小是 4KB，所以只需要 32bit 的地址中高 20bit 来寻址物理页。

返回上面我们的题目：非常大的文件，装不进内存。每行一个 int 类型数据，现在要你随机取 100 个数。针对此题，我们可以借鉴上述操作系统中内存分页的设计方法，做出如下解决方案：

操作系统中的方法，先生成 4G 的地址表，在把这个表划分为小的 4M 的小文件做个索引，二级索引。30 位前十位表示第几个 4M 文件，后 20 位表示在这个 4M 文件的第几个，等等，基于 key value 来设计存储，用 key 来建索引。

但如果现在只有 10000 个数，然后怎么去随机从这一万个数里面随机取 100 个数？请读者思考。

1.20.附录

作为一个 android 开发者，是孤单的，无助的，作为一个团队的 android 开发者，可能也是无助的，孤单的。

我们一直在做刀疤鸭系列软件，其实就是希望把孤立的软件，孤独的开发者联合起来，形成刀疤效应。

最近有很多的网友鼓励，我们支持我们，找我们要 QQ，我们想，其实我们也只是在学习，在成长，您也是在学习，在成长。

我们构建了一个刀疤鸭 QQ，在这个群里，大家都是无私的，相互帮助的，共同进步，共同成长，把你的经验介绍给其他人，帮助其他人，把我的经验介绍给别人，帮助别人。

在这个群里，我们都是友爱的，相互帮助的，我们构建共同成长。

一个人技术总是有限的，大家的力量才是无穷的。

或许您今天是技术小菜，但明天您或许就成长为技术大牛，希望技术大牛能够帮助正在成长的技术小菜们，少走些弯路。

或许您擅长 android 应用编程，但是不擅长美工；

或许您擅长美工，但是不擅长编程；
或许您有好的想法，但是一个人无法实现；
或许您有很好的技术，但是没有好的想法；
在这里，您会找到志同道合的朋友。

用战友，这词，我想更加的恰当，我们都是为美好的应用做努力的战友们。我们祝福每个战友！

我们提供我们所有之前的刀疤鸭应用源码，刀疤鸭应用图片，希望我们所有的战友都是无私的，都能够把自己的技术，自己的经验，自己的一切都奉献出来。

如果您想用刀疤鸭，可爱的形象去做软件，我们欢迎，如果您想修改我们的刀疤鸭系列软件，我们欢迎，因为我们都是刀疤鸭部落的战友！我们都是刀疤鸭族，我们有一颗坚毅的心，不放弃的信念。

希望您真诚的加入到刀疤鸭部落群，希望您在自己成长的时候，真诚的也帮助其他人。

刀疤鸭部落群：231757205