

北航机试复习指南

2021 版

目录

序.....	3
RENEWAL LIST	4
2021 年北京航空航天大学复试上机试题	5
2021 年北京航空航天大学复试上机试题解析及参考答案.....	11
2019 年北京航空航天大学复试上机试题	16
2019 年北京航空航天大学复试上机试题解析及参考答案.....	19
2018 年北京航空航天大学复试上机试题	29
2018 年北京航空航天大学复试上机试题解析及参考答案.....	31
2017 年北京航空航天大学复试上机试题	37
2017 年北京航空航天大学复试上机试题解析及参考答案.....	39
2016 年北京航空航天大学复试上机试题	51
2016 年北京航空航天大学复试上机试题解析及参考答案.....	53
2015 年北京航空航天大学复试上机试题	58
2015 年北京航空航天大学复试上机试题解析及参考答案.....	61

序

因为笔者目前搜集到的资料（主要是王道论坛几位学长的帖子，红果研、400+的复试资料和各种群文件）中，在历年机试真题这部分整合得普遍都比较难受，要么是叙述不够完善，缺少各种题目需考虑的细节，要么是没有测试用例（在这里感谢阿奇大佬的资料提供了大量的测试用例），以及大部分代码缺少解释、可读性较差、方法过于繁琐。不过还是很感谢这些学长学姐提供了非常宝贵的真题回忆！

所以笔者打算在此基础上，尽量地完善题目，对叙述不足的按个人理解补足（大部分都比较完整，并且补充的地方和原题目不会有太差的差别，核心思想不会变），并提供一定的测试用例，形成一道可以掐时间完成的、高度还原真题的试题。

北航机试时间是两小时，根据往年学长的经验，一般会提供 2~3 个测试用例。所以我在题目中也会给出 2~3 个很基本的测试用例，并在解析和代码之后给出自测分数用的一些测试用例（主要是边界条件、特殊情况等），建议完整地做完题目后再使用自测用例核算分数。由于这部分用例需要用心设计，所以并不一定能做到全面地覆盖，大家做参考即可。

分数是由机器评判和人工查阅的两部分综合而成，一般来讲应该是对一部分测试用例给一部分分数，而人工查阅也有分数意味着，只要代码思想表达出来了，大概框架也出来了，哪怕测试没有通过，也还是能捞到不少分数的。所以大家在估分的时候，可以参考这些规则综合评估自己的机试分数。

此外，对每道题目也会做一定的题目解析和参考代码，这些代码都是优化过后的，所以大家如果掐时间做题或在考场上，不必太过追求代码的完美，达到要求拿分了才是最关键的，不过平时练习的时候可以根据这些参考代码进行优化。如果做起来时间相对充裕的话，也建议养成写注释的习惯，一是可以很好地梳理自己的思路（类似于先写伪代码），二是也便于给人工查阅一个很好的体验（肯定是有用的，实在没时间至少在开头写一下大思想，比如 Dijkstra、DFS 什么的）。

所有代码都是针对单点测试的方式编写的，即每一个测试用例是一个文件，多次输入。当然了多点测试是最稳妥的方式，如果你习惯了多点测试的方式，用于单点肯定也是不会错的。在这一问题上大家可以根据自己的喜好自主选择，不过必须至少知道多点测试是怎么回事儿，因为 15 年是曾考察过多点输入的。

水印什么的我不打算加，感觉很影响阅读体验，反正这份文档是免费开源的，我欢迎各考研机构对此不遗余力的宣传（笑）。我是真的希望这个文件能传承下去，每年都有人来维护（希望每年能有人加上自己年份的机试试题并四处上传吧），不断地加入更多测试用例或题目方法，形成一个良好的氛围，真正造福更多想花时间准备、肯下功夫的人，也算是为提升北航计算机整体质量做一些绵薄的贡献了。

由于 20 年最终并未机试，以及笔者时间和水平有限，所以最终只写到了 15 年的，并且难免会有一些疏漏。望各位勤加批判，并对本文档做出相应的修正，共同完善好这份我们北航人的机试指南。

如果对本文内容有任何疑问，或需补充测试用例，欢迎在本文对应的 [GitHub Repository](#) 中 [Issues](#) 中做补充。对于新试题的添加，欢迎自行添加后并广泛扩散，或直接发邮件给我：gt3115a@163.com，我会替为上传到 [GitHub](#) 中。如有愿意帮助管理这个 [Repo](#) 的同学也欢迎私信邮箱。

本文对应的 [GitHub](#) 地址：<https://github.com/Muyiyunzi/BUAA-CS-Codes>

By 慕弋云子

2020/4/17

Renewal List

2020/4/17 by 慕弋云子 创建了本文档

2020/7/6 by 慕弋云子 最终修改全文形成 2020 初版

2021/3/4 by 慕弋云子 将 renewal 提至文档最前列

2021/3/4 by wujinxuan008@163.com 增添了 2019-2 的新解法

2021/6/15 by migu4917 添加了 2021 年复试机试的相关题目和解法

(e.g.) 2021/3/21 by xxx 添加了 xxxx/修改了 xxxx（后续更新者请保留此行）

年中的小改动在 github 上提 issues 即可，新一年复试后的大更新，请修改末尾年份并扩散

2021年北京航空航天大学复试上机试题

题目一：

在操作系统中，空闲存储空间通常以空闲块链表组织，每个块包含块起始位置、块长度及一个指向下一块的指针。空闲块按照存储位置升序组织，最后一块指向第一块（循环链表）。当有空间申请请求时，按照如下原则在空闲块循环链表中寻找并分配空闲块：

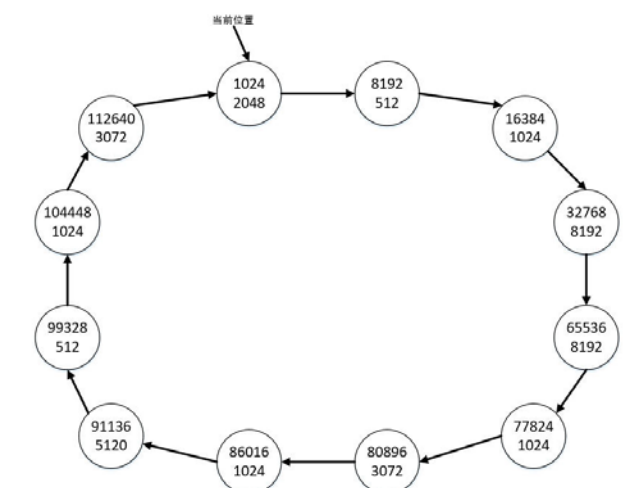
1) 从当前位置开始遍历空闲块链表（初始是从地址最小的第一个空闲块开始），寻找满足条件的最小块（即：大于等于请求空间的最小空闲块，如果有多个大小相同的最小空闲块，则选择遍历遇到的第一个空闲块）（最佳适应原则）；

2) 如果选择的空闲块恰好与请求的大小相符合，则将它从链表中移除并返回给用户；这时当前位置变为移除的空闲块指向的下一空闲块；

3) 如果选择的空闲块大于所申请的空间大小，则将大小合适的空闲块返回给用户，剩下的部分留在空闲块链表中；这时当前位置仍然为该空闲块；

4) 如果找不到足够大的空闲块，则申请失败；这时当前位置不变。

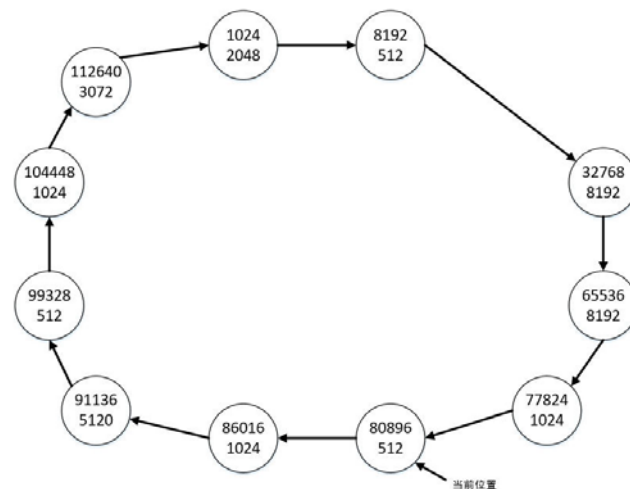
例如：下图示例给出了空闲块链表的初始状态，每个结点表示一个空闲块，结点中上面的数字指空闲块的起始位置，下面的数字指空闲块的长度，位置和长度都用正整数表示，大小不超过 int 表示范围。当前位置为最小地址为 1024 的空闲块。



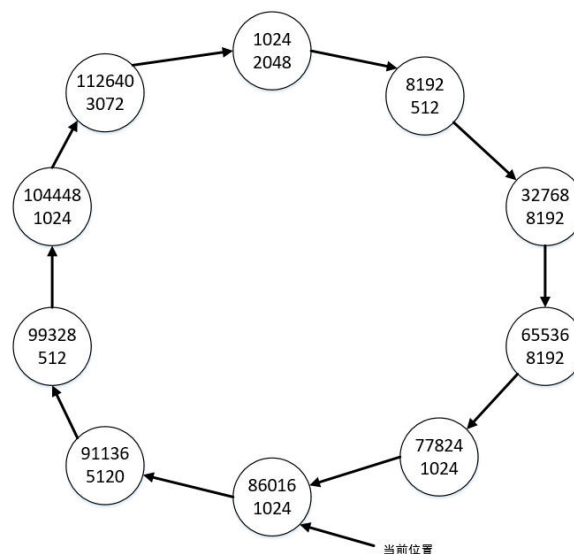
若有 4 个申请空间请求，申请的空间大小依次为：1024、2560、10240 和 512。则从当前位置开始遍历上图的链表，按照上述原则寻找到满足条件的最小块为地址是 16384 的空闲块，其长度正好为 1024，所以将其从链表中删除，这时链表状态如下图所示，当前位置变成地址为 32768 的空闲块。



从当前位置开始为第二个空间请求（大小为 2560）遍历链表，按照上述原则寻找到满足条件的最小块为地址是 80896 的空闲块，其长度为 3072，大于请求的空间大小，于是申请空间后该空闲块剩余的的长度为 512，当前位置为地址是 80896 的空闲块，链表状态如下图所示：



从当前位置开始为第三个空间请求（大小为 10240）遍历链表，遍历一圈后发现找不到足够大的空闲块，则忽略该请求，当前位置不变。下面继续为第四个空间请求（大小为 512）遍历链表，按照上述原则寻找到满足条件的最小块为当前位置的空闲块，其长度等于请求的空间大小，于是将该空闲块删除后，链表状态变为下图所示：



编写程序，模拟上述空闲空间申请。

测试用例 1:

输入:

12

1024 2048

8192 512

16384 1024

32768 8192

65536 8192

77824 1024

80896 3072
86016 1024
91136 5120
99328 512
104448 1024
112640 3072
1024 2560 10240 512 1024 6400 512 -1

输出：

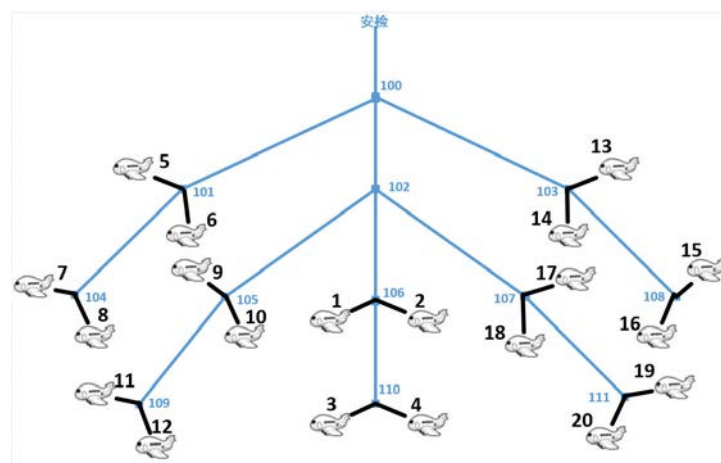
104448 1024
112640 3072
1024 2048
8192 512
32768 1792
65536 8192
77824 1024
91136 5120

样例说明：

样例输入了 12 个空闲块的信息，形成了如上述第一个图所示的空闲块链表；然后读取了 7 个空间申请请求，为前 4 个申请请求分配空间后，空闲块链表状态为上述最后一张图所示。满足第五个请求后，将删除地址为 86016 的空闲块；满足第六个请求后，地址为 32768 的空闲块剩余长度为 1792；满足第七个请求后，将删除地址为 99328 的空闲块，这时链表中剩余 8 个空闲块，当前位置为地址是 104448 的空闲块，从该空闲块开始依次遍历输出所有剩余空闲块的起始位置和长度。

题目二：

假设某机场所有登机口（Gate）呈树形排列（树的度为 3），安检处为树的根，如下图所示。图中的分叉结点（编号 ≥ 100 ）表示分叉路口，登机口用小于 100 的编号表示（其一定是叶结点）。通过对机场所有出发航班的日志分析，得知每个登机口每天的平均发送旅客流量。作为提升机场服务水平的一个措施，在不改变所有航班相对关系的情况下（出发时间不变，原在同一登机口的航班不变），仅改变登机口（如将 3 号登机口改到 5 号），使得整体旅客到登机口的时间有所减少（即从安检口到登机口所经过的分叉路口最少）。



编写程序模拟上述登机口的调整，登机口调整规则如下：

1) 首先按照由大到小的顺序对输入的登机口流量进行排序，流量相同的按照登机口编号由小到大排序；

2) 从上述登机口树的树根开始，按照从上到下（安检口在最上方）、从左到右的顺序，依次放置上面排序后的登机口。

例如上图的树中，若只考虑登机口，则从上到下有三层，第一层从左到右的顺序为：5、6、14、13，第二层从左到右的顺序为：7、8、9、10、1、2、18、17、16、15，第三层从左到右的顺序为：11、12、3、4、20、19。若按规则 1 排序后流量由大至小的前五个登机口为 3、12、16、20、15，则将流量最大的 3 号登机口调整到最上层且最左边的位置（即：5 号登机口的位置），12 号调整到 6 号，16 号调整到 14 号，20 号调整到 13 号，15 号调整到第二层最左边的位置（即 7 号登机口的位置）。

输入形式：

1) 首先输入一个整数表示树结点关系的条目数，接着在下一行开始，按层次从根开始依次输入树结点之间的关系。其中分叉结点编号从数字 100 开始（树根结点编号为 100，其它分叉结点编号没有规律但不会重复），登机口为编号小于 100 的数字（编号没有规律但不会重复，其一定是一个叶结点）。树中结点间关系用下面方式描述：

R S1 S2 S3

其中 R 为分叉结点，从左至右 S1, S2, S3 分别为树叉 R 的子结点，其可为树叉或登机口，由于树的度为 3，S1, S2, S3 中至多可以 2 个为空，该项为空时用 -1 表示。各项间以一个空格分隔，最后有一个回车。如：

100 101 102 103

表明编号 100 的树根有三个子叉，编号分别为 101、102 和 103，又如：

104 7 8 -1

表明树叉 104 上有 2 个编号分别为 7 和 8 的登机口。

假设分叉结点数不超过 100 个。分叉结点输入的顺序不确定，但可以确定：输入某个分叉结点信息时，其父结点的信息已经输入。

2) 在输入完树结点关系后，接下来输入登机口的流量信息，每个登机口流量信息分占一行，分别包括登机口编号（1~99 之间的整数）和流量（大于 0 的整数），两整数间以一个空格分隔。

输出形式：

按照上述调整规则中排序后的顺序（即按旅客流量由大到小，流量相同的按照登机口编号由小到大）依次分行输出每个登机口的调整结果：先输出调整前的登机口编号，再输出要调整到的登机口编号。编号间均以空格分隔。

测试用例 1：

输入：

12

100 101 102 103

103 14 108 13

101 5 104 6

104 7 8 -1

102 105 106 107

106 1 110 2

108 16 15 -1

107 18 111 17
110 3 4 -1
105 9 109 10
111 20 19 -1
109 11 12 -1
17 865
5 668
20 3000
13 1020
11 980
8 2202
15 1897
6 1001
14 922
7 2178
19 2189
1 1267
12 3281
2 980
18 1020
10 980
3 1876
9 1197
16 980
4 576

输出:

12 5
20 6
8 14
19 13
7 7
15 8
3 9
1 10
9 1
13 2
18 18
6 17
2 16
10 15
11 11
16 12
14 3

17 4
5 20
4 19

样例说明：

样例输入了 12 条树结点关系，形成了如上图的树。然后输入了 20 个登机口的流量，将这 20 个登机口按照上述调整规则 1 排序后形成的顺序为：12、20、8、19、7、15、3、1、9、13、18、6、2、10、11、16、14、17、5、4。最后按该顺序将所有登机口按照上述调整规则 2 进行调整，输出调整结果。

2021 年北京航空航天大学复试上机试题解析及参考答案

题目一：

这道题还是很简单的，表述的是操作系统对空闲空间管理的 Best Fit 算法，按照该算法使用循环链表模拟即可。

手写双向链表时，注意使用头节点(dummy node)来简化操作。同时，注意删除节点时，指针的操作过程，这里想不清楚画个图就好了。当然，也可以使用数组来做，只是时间复杂度可能会高一些（但是用 Vector 写起来快啊）。

以下是代码部分：

```
#include <cstdio>

struct ListNode {
    int start, length;
    ListNode* next, * prev;
    ListNode(): start(0), length(0), next(nullptr), prev(nullptr) {}
    ListNode(int start, int length): start(start), length(length),
                                     next(nullptr), prev(nullptr) {}
};

ListNode* findBestFit(ListNode* now, int query) {
    ListNode* temp = now;
    int best = 0x7fffffff;
    ListNode* chosen = nullptr;
    do {
        if(now->length >= query && now->length < best) {
            chosen = now;
            best = now->length;
        }
        now = now->next;
    } while(now != temp);
    return chosen;
}

void print_list(ListNode* head, int size) {
    ListNode* temp = head;
    for (int i = 0; i < size; i++) {
        printf("%d %d\n", temp->start, temp->length);
        temp = temp->next;
    }
}

int main() {
    int n;
    scanf("%d", &n);
    int start, length;
```

```

ListNode* dummyNode = new ListNode(); // 头节点，简化链表操作
ListNode* last = dummyNode;
for(int i = 0; i < n; i++) {
    scanf("%d%d", &start, &length);
    ListNode* node = new ListNode(start, length);
    node->prev = last;
    last->next = node;
    last = node;
}
last->next = dummyNode->next;
dummyNode->next->prev = last; // 处理输入并构建链表结束
int query;
ListNode* now = dummyNode->next;
int list_size = n;
while(true) {
    scanf("%d", &query);
    if(query == -1) {
        break;
    }
    ListNode* bestFit = findBestFit(now, query);
    if(bestFit != nullptr) {
        bestFit->length -= query;
        if(bestFit->length == 0) {
            // 注意双向链表删除时的指针操作
            ListNode* prevNode = bestFit->prev;
            ListNode* nextNode = bestFit->next;
            prevNode->next = nextNode;
            nextNode->prev = prevNode;
            delete bestFit;
            now = nextNode;
            list_size--;
        } else {
            now = bestFit;
        }
    }
}
print_list(now, list_size);
return 0;
}

```

题目二：

这道题的题目又臭又长，读出来就这么几个关键信息：输入分为登机口的树结构信息和登机口流量信息。要求调整登机口，使整体旅客到登机口经过的分叉减少。输出是调整前后的登机口对应关系。

分析：对于输入处理，可以要建一个二叉树，来保存登机口的树结构信息。要使得总体登机走的路最少，那么就把客流量大的安排在靠近安检口（即根节点的位置）。树的同一层到根节点的距离相同。由贪心算法的思想，要让整体代价最小，就把客流量大的放在近处，客流量小的放在远处。

那么本题就是两个排序：对树做深度优先排序，获取距离安检口由近到远的序列；对登机口的客流量做排序，由大到小；再将二者一一对应即可。

细节：一是输入没有给出有多少个登机口，所以建树的时候注意记录登机口的个数（题目中给出登机口的编号小于 100）。二是广度优先遍历（即层次遍历）的算法一定要熟悉，可以用 STL 加快编码速度，并且从左往右遍历。三是对登机口排序时，注意按旅客流量由大到小，流量相同的按照登机口编号由小到大。

以下是代码部分。

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <map>
#include <queue>
using namespace std;

const int MAXN = 110;
const int MAX_SON = 3;

struct Gate {
    int index, strength;
    Gate(): index(0), strength(0) {}
    Gate(int index, int strength)
        : index(index), strength(strength) {}
};

struct TreeNode {
    int index;
    TreeNode* children[MAX_SON];
    TreeNode() {}
    TreeNode(int index): index(index) {
        for(int i = 0; i < MAX_SON; i++) {
            children[i] = nullptr;
        }
    }
};

bool isGate(int index) {
```

```

        return 1 <= index && index <= 99;
    }
    // 排序的比较函数
    bool cmpGate(Gate a, Gate b) {
        if(a.strength != b.strength) {
            return a.strength > b.strength;
        }
        return a.index < b.index;
    }
    // 广度优先遍历（层次遍历）
    vector<int> BFS(TreeNode* root) {
        vector<int> res;
        queue<TreeNode*> que;
        que.push(root);
        while(!que.empty()) {
            TreeNode* node = que.front();
            que.pop();
            if(isGate(node->index)) {
                res.push_back(node->index);
            }
            for(int j = 0; j < MAX_SON; j++) {
                if(node->children[j] != nullptr) {
                    que.push(node->children[j]);
                }
            }
        }
        return res;
    }

    int main() {
        int n;
        scanf("%d", &n);
        map<int, TreeNode*> getFather; // 用这个 map 来查找父节点
        const int rootIndex = 100;
        TreeNode* root = new TreeNode(rootIndex); // 根节点
        getFather[rootIndex] = root;
        // build tree 开始建树
        int father, ch[MAX_SON], gateCnt = 0;
        for(int i = 0; i < n; i++) {
            scanf("%d%d%d", &father, ch + 1, ch + 2);
            for(int j = 0; j < MAX_SON; j++) {
                TreeNode* f = getFather[father];
                if(ch[j] == -1) {

```

```

        f->children[j] = nullptr;
    } else {
        TreeNode* node = new TreeNode(ch[j]);
        f->children[j] = node;
        if(isGate(ch[j])) { // gate 登机口
            gateCnt++; // 记录数量
        } else { // cross point 分岔口
            getFather[ch[j]] = node;
        }
    }
}

}

vector<Gate> gateList(gateCnt); // 登机口和客流量的结构体数组
for(int i = 0; i < gateCnt; i++) {
    scanf("%d%d", &gateList[i].index, &gateList[i].strength);
}
sort(gateList.begin(), gateList.end(), cmpGate); // 排序
vector<int> cost2gate = BFS(root); // 获取层次遍历的结果
for(int i = 0; i < gateCnt; i++) {
    printf("%d %d\n", gateList[i].index, cost2gate[i]);
}
return 0;
}

```

2019 年北京航空航天大学复试上机试题

题目一：

输入为两个数 a, b 。要求输出闭区间 $[a, b]$ 之间所有相邻素数组成的等差数列，三个以上才算是序列，且对多于三个的序列，不必再输出其子序列。其中 $1 \leq a, b < 100000$ 。求素数的步骤必须优化。输出的每个等差数列占一行，每个数之间有空格，一行最后一个数也有空格。输入的数据规模不会超过 `int` 型变量范围。

测试用例 1：

输入：

100 200

输出：

151 157 163

167 173 179

测试用例 2：

输入：

200 500

输出：

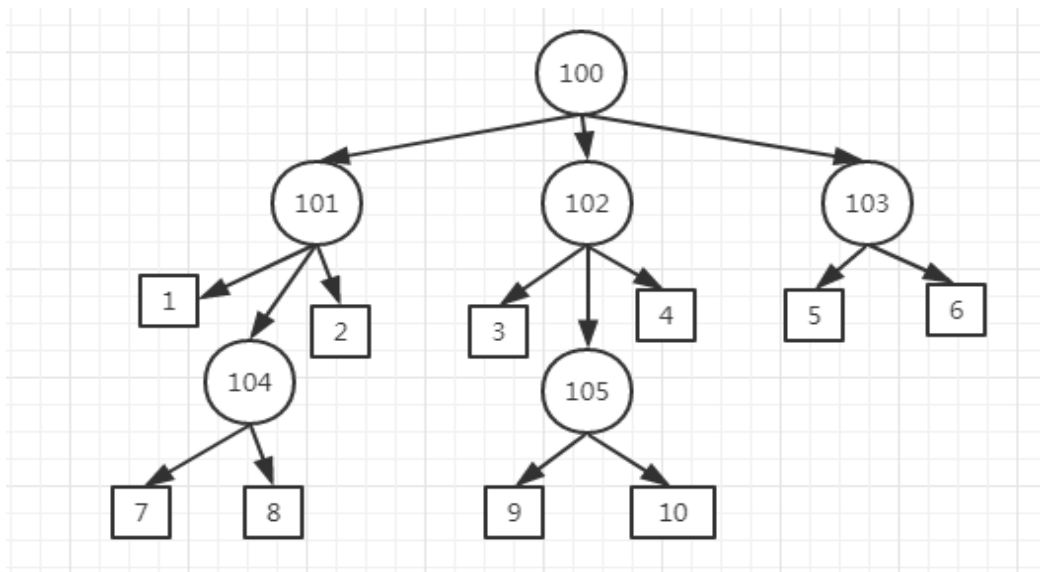
251 257 263 269

367 373 379

题目二：

先输入 n ($n < 10000$)，代表有 n 组数据，接下来 n 行每行输入四个数，第一个输入为分支节点，后三个输入为其三个孩子节点（-1 代表子节点不存在）。输入的 n 行中第一行第一个节点为根节点，作为起点。保证输入的结点至多有一个父节点。

然后输入 m ($m < n$)，接下来输入 m 个客户，每行输入他的目标（叶节点）和优先级（数字越小越优先，保证不会有相同优先级），要求从起点出发，按照优先级把客户送到目标节点，然后从该节点继续运送下一个客户至其目标（叶节点），全部运送完毕后返回起点，记录每段经过的路径，并输出。输入的数据规模不会超过 `int` 型变量范围。



测试用例 1:

输入:

6

100 101 102 103

101 1 104 2

102 3 105 4

103 5 6 -1

104 7 8 -1

105 9 10 -1

5

7 7

9 3

6 2

8 1

4 5

输出:

100 101 104 8

104 101 100 103 6

103 100 102 105 9

105 102 4

102 100 101 104 7

104 101 100

测试用例 2:

输入:

10

100 101 108 107

101 1 102 2

108 103 104 105

107 17 109 18

102 3 4 5

103 7 8 9

104 10 106 11

105 15 16 -1

109 19 20 21

106 12 13 14

5

8 1

14 3

16 2

5 0

19 4

输出:

100 101 102 5

102 101 100 108 103 8

103 108 105 16

105 108 104 106 14

106 104 108 100 107 109 19

109 107 100

2019 年北京航空航天大学复试上机试题解析及参考答案

题目一：

这道题是比较简单的数学问题，建议在 40 分钟内完成（我一开始被“相邻”给坑了，红果研的书没说相邻）。而关于素数的判断，会就是会，不会就是不会。可以选择用 sqrt 那样判断，我这里使用了埃式筛法，并且提前“打表”，可以大大地加快判断时间。尤其这种要求相邻素数的问题，提前打出素数表，建立一个 prime[] 数组存放所有的质数是非常舒服的。建议两种质数的判断方式都要掌握。

最后注意一下在大于三个的情况下不要重复打印，以及 1 的特判问题。有部分回忆试题是说 a 必然大于 2，不过也有说需要特判 1 的学长，并且因为这事儿在考场最后手忙脚乱。我在这里就取需要特判的情况，锻炼一下大家的考虑特殊情况的思维。

据我了解的情况，从 2018 年以后机试因为人数原因一般来讲是两场，像这道题就是一场质数一场合数，还是相对公平的，大家感兴趣也可以顺手练一下合数的，总之知道这种差异就可以了。

以下是代码部分：

```
#include <stdio>

const int maxn = 100010;

bool hashTable[maxn] = {false};
int prime[maxn], pNum = 0;

//使用埃式筛法打印质数表
void FindPrime() {
    hashTable[1] = true; //特判认为 1 不是质数
    for(int i = 2; i < maxn; i++) {
        if(hashTable[i] == false) {
            prime[pNum++] = i;
            for(int j = 2 * i; j < maxn; j += i) {
                hashTable[j] = true;
            }
        }
    }
}

int main() {
    FindPrime(); //初始化素数表
    int a, b;
    scanf("%d %d", &a, &b);
    //找到从 a 开始的第一个质数
    int first;
    for(int i = 0; ; i++) {
        if(prime[i] >= a) {
            first = i;
        }
    }
}
```

```

        break;
    }
}
//从 first 开始讨论，其开始的相邻三个是否为等差数列，当 prime[i + 2] > b 时或 i+2 大于了最大质数个数时退出循环
for(int i = first; i + 2 <= pNum && prime[i + 2] <= b; i++) {
    if(prime[i] + prime[i + 2] == 2 * prime[i + 1]) {
        printf("%d %d %d ", prime[i], prime[i + 1], prime[i + 2]);
        //接下来判断是否多于三个，若多于三个，提前对 i 做++，以循环判断并避免重复输出
        while(prime[i + 1] + prime[i + 3] == 2 * prime[i + 2] && i + 3 <= pNum && prime[i + 3] <= b) {
            printf("%d ", prime[i + 3]);
            i++;
        }
        printf("\n");
    }
}
return 0;
}

```

自测用例 1:

输入:

1 100

输出:

3 5 7

47 53 59

自测用例 2:

输入:

95000 100000

输出:

95261 95267 95273 95279

95701 95707 95713

95911 95917 95923 95929

95989 96001 96013

96353 96377 96401

96419 96431 96443

96757 96763 96769

97429 97441 97453

97961 97967 97973

98467 98473 98479

98573 98597 98621
99817 99823 99829

自测用例 3:

输入:

151 179

输出:

151 157 163
167 173 179

自测用例 4:

输入:

200 268

输出:

251 257 263

用例 1 考虑了 a 的边界条件（是否特判 1 的情况，未特判会输出 1 3 5 7），用例 2 考虑了 b 的边界条件（素数表是否打印到位，是否会溢出），用例 3 考虑了 a 、 b 闭区间的问题，是否可以正常打印出闭区间边界上的数字。用例 4 考虑了多于 3 个时，期间在中间阶段的情况，即判断多于三个的情况时，有没有对判断条件加入 b 的限制。

题目二:

这道题有很多人都是用树做的，很多人在考场上写了一堆东西，到最后 debug 不出来也无所谓就交了，包括红果研上给的也是用树做的代码，足足写了六页纸。事实上这题用图做是非常简单的，这也告诉我们，当题目中出现“路径”这种词的时候，还是应该优先考虑用图来做，更何况根据这道题三叉树没有环的特点，实际上路径是唯一的（当你发现题目中并没有要求最短，或是某个指标最大最小时，就应该反应过来实际上路径是唯一的），所以这里给出用图做的方式。如果有用树做的还很简单的方式，可以在之后补充上。

具体代码的思路就是，因为数据规模大于 1000，所以使用邻接表的方式存储图，然后使用 DFS 对图进行遍历。然后开了一个 `pre` 数组记录前驱结点，再 DFS 打印数据（实际上就是堆栈，用一个 `stack` 亦可）。对输入的客户数据使用结构体存储，可以很方便的使用 `sort` 进行排序，其他的没有太多技术难度，可以说只要想到用图，基本上这题可以很轻松地做出来。

以下是代码部分。大约写了半个小时，然后 debug 了二十分钟，一个小时内足够拿下了。

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;
```

```

const int maxn = 10010;
const int inf = 0x3fffffff;

int n, m, s;
int pre[maxn];
bool vis[maxn] = {false};
bool flag; //flag=true 时表示已找到
vector<int> Adj[maxn]; //邻接表存储图

struct client {
    int num, prior;
}c[maxn];

bool cmp(client a, client b) {
    return a.prior < b.prior;
}

void printDFS(int v) {
    if(pre[v] != -1) {
        printDFS(pre[v]);
        printf("%d ", v);
    }
}

void DFS(int s, int ed) {
    vis[s] = true; //访问时先标记已访问，防止重复访问
    for(int i = 0; i < Adj[s].size(); i++) {
        int v = Adj[s][i];
        if(flag == true) return; //若已找到，直接返回
        if(vis[v] == false) { //当未找到时且当前 v 结点未被访问过
            pre[v] = s; //进入并设定 v 的前驱结点为 s
            if(v != ed) { //若此时 v 不是终点，继续 DFS 遍历
                DFS(v, ed);
            }
            else { //到达终点
                printDFS(pre[v]); //打印先前的所有结点并空格
                printf("%d\n", v); //打印最后一个结点并换行
                flag = true; //并标记已找到
            }
        }
    }
}

int main() {

```

```

int f, c1, c2, c3; //father & child 1~3
scanf("%d", &n);
for(int i = 0; i < n; i++) {
    scanf("%d %d %d %d", &f, &c1, &c2, &c3);
    if(i == 0) s = f;
    if(c1 != -1) {
        Adj[f].push_back(c1);
        Adj[c1].push_back(f);
    }
    if(c2 != -1) {
        Adj[f].push_back(c2);
        Adj[c2].push_back(f);
    }
    if(c3 != -1) {
        Adj[f].push_back(c3);
        Adj[c3].push_back(f);
    }
}
scanf("%d", &m);
for(int i = 0; i < m; i++) {
    scanf("%d %d", &c[i].num, &c[i].prior);
}
sort(c, c + m, cmp);
//假设不存在环，那么路径唯一，直接使用 DFS 即可
printf("%d ", s); //打印起点，此后遍历时均不打印起点
for(int i = 0; i < m; i++) {
    flag = false;
    memset(pre, -1, sizeof(pre));
    memset(vis, 0, sizeof(vis));
    if(i == 0) {
        DFS(s, c[i].num);
    }
    else DFS(c[i - 1].num, c[i].num);
}
flag = false;
memset(pre, -1, sizeof(pre));
memset(vis, 0, sizeof(vis));
DFS(c[m - 1].num, s);
return 0;
}

```

以下为 wujinxuan008@163.com 增添的使用树的解法，ta 表示使用树也是比较简单的，在此附上给大家参考：

```

#include<stdio.h>
#include<stdlib.h>
#include<vector>
#include<algorithm>
#include<queue>
using namespace std;
#define MaxN 10000
struct Node
{
    int value;
    Node* child[3];
    Node* parent;
    int ifVisited;
};

int g_Path[MaxN];
int g_pathIndex = 0;

struct Customer
{
    int leafValue;
    int weight;
};
Customer g_customer[MaxN];
int g_customerCnt = 0;
int g_found = 0;

bool cmp(Customer a, Customer b)
{
    return a.weight < b.weight;
}

Node* InitNode(int value)
{
    Node* node = new Node;
    node->value = value;
    node->parent = NULL;
    int i = 0;
    for (i = 0; i < 3; i++)
    {
        node->child[i] = NULL;
    }
    return node;
}

```



```

Node* findWithValue(int value, Node* root)
{
    int i = 0;
    if (root == NULL)
        return NULL;
    if (root->value == value)
        return root;
    for (i = 0; i < 3; i++)
    {
        Node *result = findWithValue(value, root->child[i]);
        if (result != NULL)
            return result;
    }
    return NULL;
}

void InitVisited(Node* root)
{
    int i = 0;
    if (root == NULL)
        return;

    root->ifVisited = 0;

    for (i = 0; i < 3; i++)
    {
        InitVisited(root->child[i]);
    }
    return;
}

int DFS_Search(Node *current, int value)
{
    g_Path[g_pathIndex++] = current->value;
    current->ifVisited = 1;
    if (current->value == value)
    {
        return 1;
        g_found = 1;
    }
}

```

```

int i = 0;
for (i = 0; i < 3; i++)
{
    if (current->child[i] != NULL && current->child[i]->ifVisited==0)
    {
        if (DFS_Search(current->child[i], value))
        {
            return 1;
        }
    }
}
if (current->parent->ifVisited == 0)
{
    if (DFS_Search(current->parent, value))
    {
        return 1;
    }
}

if (!g_found)
{
    g_pathIndex--;
    return 0;
}
else
    return 1;
}

```

```

int main()
{
    int dataLine = 0;
    scanf("%d", &dataLine);

    int i = 0;
    int childIndex = 0;
    int value;
    Node* root = NULL;

    for (i = 0; i < dataLine; i++)
    {
        scanf("%d", &value);
        Node* searchedNode = findWithValue(value, root);
        if (searchedNode == NULL)

```

```

    {
        searchedNode = InitNode(value);
        if (i == 0)
            root = searchedNode;
    }
    for (childIndex = 0; childIndex < 3; childIndex++)
    {
        scanf("%d", &value);
        if (value != -1)
        {
            searchedNode->child[childIndex] = InitNode(value);
            searchedNode->child[childIndex]->parent = searchedNode;
        }
    }
}

scanf("%d", &g_customerCnt);
for (i = 0; i < g_customerCnt; i++)
{
    scanf("%d %d", &g_customer[i].leafValue, &g_customer[i].weight);
}

sort(g_customer, g_customer + g_customerCnt, cmp);

for (i = 0; i < g_customerCnt; i++)
{
    g_found = 0;
    InitVisited(root);
    g_pathIndex = 0;
    if (i == 0)
        DFS_Search(root, g_customer[i].leafValue);
    else
    {
        Node* tmp = findWithValue(g_customer[i-1].leafValue, root);
        DFS_Search(tmp, g_customer[i].leafValue);
    }

    for (int j = 0; j < g_pathIndex; j++)
    {
        printf("%d ", g_Path[j]);
    }
}

```

```
        printf("\n");  
  
    }  
  
    system("Pause");  
    return 0;  
}
```

2018 年北京航空航天大学复试上机试题

题目一：

在二维空间中给定一定数量的线段，线段的右端点的横坐标必然比左端点的横坐标大。第一行输入 n ($n < 10000$)，表示一共有 n 条线段。之后输入 n 行，每一行依次输入左端点的横、纵坐标，右端点的横、纵坐标；一些线段可以连在一起，规定连接的规则只能是一条线段的左端点和另一条线段的右端点相连，要求输出连在一起的大线段最多有多少条线段，以及最大线段起点（左端点）的横、纵坐标。输入的数据规模不会超过 `int` 型变量范围。

测试用例 1：

输入：

```
1
1 2 2 3
```

输出：

```
1 1 2
```

测试用例 2：

输入：

```
4
1 2 2 3
2 3 3 4
3 4 4 1
2 3 5 6
```

输出：

```
3 1 2
```

题目二：

燕子会将巢穴建立在最稳定的地方，对于二叉树而言，一个结点的子分支越多越稳定。

第一行输入 n ($n < 10000$)，之后输入 n 行，每一行输入 4 个数字，分别为根节点，和其左孩子、中孩子、右孩子（确保每个根已经在之前出现过）的节点编号。求前序遍历（根，左，中，右）中子分支最多而且深度最深（高度最低）的节点编号，以及第几次遍历到该节点。若有多个满足题目要求的结点，取最先遍历到的结点。若输入的孩子节点编号为 -1，则代表不存在此孩子分支。输入的数据规模不会超过 `int` 型变量范围。

测试用例 1：

输入：

```
2
20 2 -1 7
7 6 4 -1
```

输出：

```
7 3
```

测试用例 2：

输入：

4

10 9 -1 7

9 6 5 -1

7 -1 3 11

11 -1 14 -1

输出:

9 2

2018 年北京航空航天大学复试上机试题解析及参考答案

题目一：

这道题的算法本质应该属于贪心（不过也有说是是什么算法都没有的），我认为这是一道非常标准的北航机试题目一风格，不过难度稍难了些，18 年很多学长都叫苦不迭，主要是难度突然比之前上了一个档次，很多人捣鼓了两个小时才把第一题做的差不多（还不一定全对）。我个人认为这道题的时间应该控制在一个小时左右，一个小时的时候做的差不多了就赶紧开始写第二道（事实上第二题我认为反而比第一题简单，做完了第二题还可以反过来检查第一题），总之写出了第二题的主体部分，这样就可以拿到大部分分数，哪怕一些特殊用例通过不了。在考场上，如果有时间考虑这些东西就去考虑，没时间自然不能因小失大。

而贪心算法我认为是一件很玄学的东西，在考场上不可能去证明你算法的正确性，所以就是感觉上差不多就去写了。但如果你写完了发现自己想的有问题，可能就没时间再去修改调整了，所以一定要在最初想代码逻辑的时候就谨慎一些，免得到最后要调整大框架，那是根本没有时间的。

我受到递归思想的启发（比如当时 codeup 在递归的章节就让我去做八皇后问题了，还有神奇的口袋），这些启发我一种“从头一个个递归枚举”的思想，你也可以说是一种 DFS 吧（DFS 的本质也是递归，而递归的本质是循环）。而最关键的一点是题目中给出了“线段的右端点的横坐标必然比左端点的横坐标大”这句话，让我非常警觉（想想为什么只给了横坐标而纵坐标无所谓？），于是这成为了我顺序枚举的一种标准，先对所有线段按起点（左端点）的横坐标（sx）排序，这样就不必判断左端点会和其他右端点重合的情况了（若存在这样的线段，它的左端点必然小于右端点，也就是小于当前线段的左端点，这意味着之前就已经判断过了）。于是代码和递归思想就大大地被简化了。虽然还是会重复判断很多情况，但是当运行时间完全够用的情况，我们就不“贪心”了，还是简单地完成题目就好了。以下是我的代码。

```
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;

const int maxn = 10010;
int cnt[maxn], temp = 0;
int n, s, t;

struct line {
    int sx, sy, tx, ty;
}l[maxn];

void check(int i, int now) { //对线段 i 进行讨论，查看 now 之后的线段是否与线段 i 重合
    for(int j = now + 1; j < n; j++) {
        if(l[j].sx == l[now].tx && l[j].sy == l[now].ty) { //若此时重合
            temp++;
            check(i, j);
            temp--;
        }
    }
}
```

```

        else if(l[j].sx > l[now].tx) { //因已按横坐标排序，此句用于加速判断，可以不写
            return;
        }
    }
    //当未找到退出循环后，比较 temp 和对 i 的计数器，若更大则赋值
    //虽然每次从 check 退出都会判断一下，但是如果开 flag 标记找到，也会判断一下所以
    就这样吧
    if(temp > cnt[i]) {
        cnt[i] = temp;
    }
    return;
}

```

```

bool cmp(line a, line b) {
    return a.sx < b.sx;
}

```

```

int main() {
    scanf("%d", &n);
    for(int i = 0; i < n; i++) {
        scanf("%d%d%d%d", &l[i].sx, &l[i].sy, &l[i].tx, &l[i].ty);
    }
    sort(l, l + n, cmp); //对所有线段按起点横坐标从小到大排序
    memset(cnt, 0, sizeof(cnt));
    for(int i = 0; i < n; i++) {
        check(i, i);
    }
    int MAX = -1, num;
    for(int i = 0; i < n; i++) {
        if(cnt[i] > MAX) {
            MAX = cnt[i];
            num = i;
        }
    }
    printf("%d %d %d\n", MAX + 1, l[num].sx, l[num].sy);
    return 0;
}

```

以下是自测用例部分

自测用例 1:

输入:

6

1 3 2 4

2 4 3 3

3 3 7 2
2 4 4 1
4 1 5 6
5 6 6 5

输出：
4 1 3

自测用例 2:

输入：
6
2 4 3 3
1 3 2 4
3 3 7 2
2 4 4 1
4 1 5 6
5 6 6 5

输出：
4 1 3

自测用例 3:

输入：
3
1 3 2 4
2 4 7 2
3 3 7 2

输出：
2 1 3

自测用例 4:

输入：
3
1 3 2 4
3 3 7 2
2 4 3 3

输出：
3 1 3

自测用例 1 直接干掉了某年学硕第一学长提供的代码，他用的是线段融合，对相连的两条线段做融合，将他们的起点和终点都改为融合后的起点和终点，并且对线段长度加和（相当于忽略掉了中间那个点），那么这样做的弊端自然是对中间那个点加分支，他就 GG 了。

自测用例 2 干掉了红果研给的代码，巧的是他也用的线段融合，这次他比较机智，边扫描边融合，还分了左端点融合和右端点融合两种情况讨论；如果可以连接，把新起点（或终点），也就是本该融合的点，赋给连接线段的起点（或终点），另外一点就按扫描输入的保持不变。新线段长度对应相加。也就是说他保留了原线段，以便之后输入再融合，还新生成了新线段。那么干掉这样代码的方式自然是让他“融合掉”分支点，所以就有了我先输入 2 4 3 3，再输入 1 3 2 4，让他融合掉 (2,4) 这个点，那么后边的输入他就都 GG 了。

自测用例 3 和 4 都是比较常规的，主要测试在右端点相连的情况下会不会被判定相连，以及先输入左边和右边的线段，再输入“中间的”线段，长度记录会不会正常，主要是卡掉一些只判断左、右的情况。

另外，我后来在群里讨论的时候，有说这题原题说了一个端点至多有多条线段相连的情况，这样确实就不会出现以上问题了，并且可能直观的思路会想到线段融合吧。无论如何，大家就当锻炼思维了，这种难度上的浮动我认为也是合理的。

题目二：

相比上一题来讲，这一题的坑就非常少了，可能因为这是北航机试第二年考察树、数据结构的部分，所以难度没有太做提升（相对 17 年可以认为就是二叉变三叉，而且这次明确说了树还要求遍历，这里就不做对比了，以免有些同学 17 年的还没做）。基本的建树、遍历，只要二叉树学的没问题，三叉树就也不是问题了，都是很基本的操作。而我认为这题的难点在于建树（红果研给的建树代码我认为有问题），我想了想我好像也没有练过二叉树这样输入数据建树的代码。因为每次输入四个数，孩子很好说，可以轻松挂在父节点下边，可是父节点的地址咋办呢。如果通过类似并查集的方式建立一个父节点的数组，那么父节点的父节点的地址又在哪儿呢？所以我认为这也是我比较巧妙的地方，即建立一个 `map<int, node*>` 的映射关系，建立题意中的结点编号与结点指针的映射关系，这样能通过结点编号直接找到它的地址，从而可以边扫描边建树，代码就十分简洁了。

此外，应该也可以考虑使用静态数组来建树，对不存在的分支也赋值 -1 进去而达到一个完全二叉树的情况，并根据完全二叉树的特性进行查找和存储，同时对深度来讲也好计算，只是在遍历的过程会比较繁琐一些（可能这也是题目设置一个第几次遍历到该结点而防止一些偷鸡的做法），这里只提供一个思路，如果正好有这样写的，可以放上来做比较。

然后回到我的做法上来讲，自己写一个 `newnode` 函数会比较好，赋值 `data` 部分，并且对左中右孩子都挂上空指针，这样边扫描边建树就方便了许多；同时对树设置深度信息，这样在遍历的同时也方便进行深度的比较。以下是代码部分。

```
#include <cstdio>
#include <map>
using namespace std;

const int maxn = 10010;
int n, num, time, numtime, dmax = -1, cntmax = -1;
//n 为输入行数，num 为满足题意结点，time 为遍历次数，numtime 为满足题意的遍历次数
//dmax 记录遍历过程中满足题意的最大深度，cntmax 为最大子结点数

struct node {
    int data, depth;
    node *lchild, *mchild, *rchild;
```

```
};
```

map<int, node*> m; //使用 map 建立结点数字与结点指针的映射

```
node* newnode(int x) {
    if(x == -1) { //如果输入值为-1，则返回空指针
        return NULL;
    }
    node* root = new node;
    root->data = x;
    root->lchild = NULL;
    root->mchild = NULL;
    root->rchild = NULL;
    return root;
}
```

```
void preOrder(node* root, int d) { //前序遍历输入根节点和深度
    if(root == NULL) {
        return;
    }
    root->depth = d;
    time++;
    int cnt = 0;
    if(root->lchild != NULL) cnt++;
    if(root->mchild != NULL) cnt++;
    if(root->rchild != NULL) cnt++;
    if(cnt >= cntmax && d > dmax) {
        dmax = d;
        cntmax = cnt;
        num = root->data;
        numtime = time;
    }
    preOrder(root->lchild, d + 1);
    preOrder(root->mchild, d + 1);
    preOrder(root->rchild, d + 1);
}
```

```
int main() {
    scanf("%d", &n);
    int n1, n2, n3, n4;
    node* root;
    for(int i = 0; i < n; i++) {
        scanf("%d %d %d %d", &n1, &n2, &n3, &n4);
        if(i == 0) {
```

```

        root = newnode(n1);
        m[n1] = root;
    }
    node* rtemp = m[n1];
    rtemp->lchild = newnode(n2);
    rtemp->mchild = newnode(n3);
    rtemp->rchild = newnode(n4);
    m[n2] = rtemp->lchild;
    m[n3] = rtemp->mchild;
    m[n4] = rtemp->rchild;
}
preOrder(root, 1);
printf("%d %d\n", num, numtime);
return 0;
}

```

自测用例 1:

输入:

1
10 -1 -1 -1

输出:

10 1

自测用例 2:

输入:

4
10 9 -1 7
7 -1 3 11
11 -1 14 16
9 6 5 -1

输出:

11 7

一般这种建树比较复杂的题，给的测试用例“规模”还会是比较大的，基本能建到三层四层，所以考虑一下只有根节点的情况会不会进行判断，以及对题目给的测试用例进行拓展，比如题目给的用例 1 表明当子分支数目相同时取深度更深者；用例 2 表明当有深度更深但子分支更少时，取子分支更多者做优先考虑。那么对用例 2 稍加修改就得到了子分支数目相同时取深度更深者的自测用例 2，同时调换一下输入顺序，以卡掉某些边输入边加深度的代码。

2017 年北京航空航天大学复试上机试题

题目一：中位数的位置

先输入一个整型数字 N ($1 \leq N < 1000000$)，接着输入 N 个无序的数字。要求输出非递减顺序排列后的中位数，以及该中位数输入的次序。如果 N 为偶数，则分两行输出有两个中位数，如果 N 为奇数，输出最中间的数即可。输入的数据可能重复，当输入数据相同时，优先输入的排在前面。所有输入的数据不会超过 `int` 型变量的范围。

测试用例 1:

输入:

5

9 2 7 1 6

输出:

6 5

测试用例 2:

输入:

6

9 6 7 1 2 3

输出:

3 6

6 2

题目二：查找未定义的变量

输入两个 C 语言语句，第一句为正常的 C 语言变量定义语句，符合 C 语言的语法要求，变量间可以有多个空格，包含数组，指针定义等，第二句为基本的变量运算语句（不存在强制转换），要求输出第二个 C 语言语句中未定义的变量。定义的变量数目不会超过 1000 个，每个变量的长度不会超过 1000 个字符。不考虑 `long long` 等多词变量的情况，输入保证定义的变量只有一个词。

输出的每一个未定义的变量后都有一个空格，然后换行；若没有未定义的变量，则只输出一个换行。此外，不考虑数组超限造成的未定义情况。

测试用例 1:

输入:

`double x12, y=1, num_stu=89, a[30], *p;`

`Sum=num+x12*y`

输出:

Sum num

测试用例 2:

输入:

`int int123 =456;`

`in = ab_ * (int123++)`

输出:

in ab_

题目三：找家谱成员

输入若干行（总行数不超过），每一行的第一个输入为家谱中的某成员，该行接着输入的信息为该成员两个孩子的姓名（保证除第一行的第一个成员外，每行第一个成员都在之前输入过）。

输入完毕后，最后一行输入两个成员（保证已输入过），为要求查找的两个家谱成员的关系，输出内容包括他们最近邻的共同祖先的名字以及在家谱中相差的层次数。

测试用例 1:

输入:

Ye Shu Ba

Shu Ge Mei1

Ba Self Mei2

Ge Son1 Son2

Son2 Mei1

输出:

Shu 1

2017 年北京航空航天大学复试上机试题解析及参考答案

17 年是三道题目，因为当时好像就一场考试，有可能时间是两个半小时，风格也和之后 18 年差异很大，可以说是一道分水岭，所以并不是很具有参考价值，但也说不准会杀个回马枪，作为丰富解决字符串处理这些问题的经验来讲，是很好的练习题。我个人认为第二题是最难的，坑非常多（不知道是不是因为试题回忆的原因，原题很多细节没有给到，我在完善的时候也没有去补这些细节，那么就需要在代码中考虑很多判定），所以正常来讲，应该能做到第一题轻松解决，第三题稍微思考下，第二题写个大概，这样保守估计也能在 70-80 分左右了。

另外就是，貌似是从 18 年开始逐步加入 codeblocks（毕竟开始上树上图了，再不让用 C++，尤其是容器函数有点儿过分为 hhh），所以 17 年之前貌似是不太建议用 C++ 的，那么像第一题这种不用 sort 确实是稍微费点儿时间。用 sort 可能十分钟就写完了第一题。

题目一：

这道题是非常常规的排序题，可以做的非常百花齐放，插入排序归并排序什么的。这题我唯一能想到的坑点就是会不会给的数据规模非常大而没法开数组去存储，不过好像也没法不存储就做排序，所以在完善的时候加入了对 n 的限制。

总之对于这种问题就非常典型的结构体+sort，如果你对这个思想还不够条件反射，可能这样的题还做的不够多。建议在 15 分钟内完成。以下是代码部分：

```
#include <cstdio>
#include <algorithm>
using namespace std;

const int maxn = 1000010;

struct node {
    int x, t; //输入的数值和次序
} num[maxn];

bool cmp(node a, node b) {
    if(a.x == b.x) return a.t < b.t;
    return a.x < b.x;
}

int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n; i++) {
        scanf("%d", &num[i].x);
        num[i].t = i + 1;
    }
    sort(num, num + n, cmp);
    if(n % 2 == 1) {
        printf("%d %d\n", num[n / 2].x, num[n / 2].t);
    }
```

```

    }
    else {
        printf("%d %d\n", num[n / 2 - 1].x, num[n / 2 - 1].t);
        printf("%d %d\n", num[n / 2].x, num[n / 2].t);
    }
    return 0;
}

```

自测用例 1:

输入:

1
2

输出:

2 1

自测用例 2:

输入:

6
9 3 7 1 3 3

输出:

3 5
3 6

用例 1 主要是边界条件，一般来讲不会有什么问题；用例 2 是按题目所说，当数字重复时会如何排序，一般来讲也不会出错，主要卡掉一些，先获得数字再对次数排序的人（这样的会输出 3 2 3 5），总的来说没什么坑，稍微注意下细节就好。

题目二:

我认为这是三道题里最难的一道。不知道是不是因为考场回忆细节不全的缘故，不过我看阿奇大佬也参考了网上的答案。我个人做这道题的时候，没考虑到指针和数组，差不多时间就到了，因为当时题目也没说数组如果超限会怎么样（比如定义到 30，后边却用了 50），也没说会不会强制转换（这意味着你要输入很多变量类型的名字进去），也没说 long long, unsigned int 这种情况怎么办（这意味着开头跳过的部分，不能找到空格就停止，而必须检查很多变量类型的名字），也没有说变量长度（这关系到如何存储数据），这些都是我后来完善上去的。不过这道题我认为也是非常优秀的，它非常能甄别被试者的基本功，以及是否有关注意到变量声明规则这些基本的细节，同时我想也对将来自己写软件帮助很多（相当于假如你是 IDE，你怎么识别这些东西定义了没有，这道题就在做这样的事情）。

能否做出这道题，或者说很简洁地做出这道题，很关键的一点是定义的规则（按说试题应该给出的，但不给就是考基本功了），这里摘自《算法笔记》上的相关内容：

变量名一般来说可以任意取，只是需要满足几个条件：

① 不能是 C 语言标识符（标识符不多，比如 `for`、`if`、`or` 等都不能作为变量名，因为它们在 C 语言中本身有含义）。所以 `ZJU`、`Love` 等都可以用作变量名，但还是建议取有实际意义的变量名，这样可以提高程序的可读性。

② 变量名的第一个字符必须是字母或下划线，除第一个字符之外的其他字符必须是字母、数字或下划线。因此 `abc`、`_zju123_ujz` 是合法的变量名，`6abc` 是不合法的变量名。

③ 区分大小写，因此 `Zju` 和 `zju` 可以作为两个不同的变量名。

根据这些规则，基本就可以划分成两个思路，一个是扫描“合法”字符（字母、数字、下划线，为方便后面也这样叫了），一个是扫描“非法”字符（与合法相对，比如逗号、空格、加减乘除（也就包括了指针的*号）、取地址符号、分号等等）。显然扫描合法字符的做法要更加简单些。

但无论如何有两点是比较关键的，也往往是答题者容易忽视的：数字不能作为变量名的起始——实际上第二点正与此联系，在非法字符中夹合法字符却不该被标记的情况，就只有这一种——常数。

考虑下面这个用例的输入：

```
int x, y=1, *p = &sum;
```

```
*p = 2 + y;
```

有的人会输出就会输出 2（事实上应该什么也不输出的）。因为他的判断条件是，在扫描到合法字符之后就开始 `while` 记录、然后遇到了非法字符就继续++，直到再次遇到合法字符，这样循环判断。我认为考场上能想到这层判断的都已经很不容易了，不过就是要考虑清楚细节，就是这种情况有没有什么特例，也就是常数的情况。

所以我们的算法思路就比较清晰了，先写一个 `bool` 函数，判断合法字符：

```
bool judge(char c) { //判断字符是否为合法变量名，即只能是字母、下划线、数字
    if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
        || (c >= '0' && c <= '9') || c == '_') return true;
    else return false;
}
```

然后对输入的 `char` 数组进行处理。这里形而上的简单讲一下思想，具体代码可以看后面的代码部分。

①首先通过 `while(str[i] != '\0') i++`；跳过变量名，注意这里不建议写成 `while(str[i++] != '\0')`；虽然这样看起来更加简洁了，但是造成的影响是退出时 `i` 会大 1（比如你想进一步处理这个空格，但实际上 `i` 已经在这个空格的后面一个了）。

②进入一个大循环，直到 `i` 超过数组长度时退出，在这个大循环里完成以下操作：

（1）先跳过所有的不合法字符，直至 `i` 超出范围或遇到“合法”字符。无论是变量类型后边空多少空格，或是指针的*打了头阵（比如 `int *p`）都可以被我们很好地忽略掉；

（2）检查遇到的是不是数字，如果是数字，认为是常数，用 `while` 扫描掉所有后续的数字；然后如果遇到合法字符就进入（3），如果还是不合法的就重新开始大循环进入（1）；

（3）对遇到的合法字符，循环记录到一个 `char` 数组里；

以上就完成了对第一句话的处理，事实上是非常简洁的。之后对第二句的处理就大同小异了，这里就不继续展开说了，大家直接看代码，也有注释写的比较清楚了。具体我想到的坑例子都写在自测用例里了。

另外细节上需要注意的就是第二句话不一定有分号（我看的是阿奇学长给的题目，所以用 `i < strlen()` 做退出的判断，比分号判断会更好些），以及 `while(i < len && !judge(str[i]))` 类似

这样的双重判断时，尽量把 `i < len` 写在前边，如果 `i++` 后超出了上限而判断 `str[i]` 很可能会出问题（一般来讲会存储 `'\0'`），以及养成字符串处理的好习惯，在处理完毕后加上 `'\0'`。

数组和指针其实都不用特判的（很多人应该一开始会想到要特判数组的，至少我就是这样的），因为数组[（数字）]，实际上也是在非法字符后直接写了数字，会在（2）的判断中被忽略掉，而对于指针来讲，*再多它也是属于变量类型的，都可以统统被忽略掉，就算需要特判，你也无法区分他和乘法的区别。

此外对于浮点数，如果有小数点也是一样的，会被（2）判断掉。其实想了这么多，就大约能明白为什么变量名不能用数字做开头了（会给判断变量增加很大的麻烦），算是一个题目之外的小收获了。

```
#include <stdio>
#include <cstring>

const int maxn = 1010;
char str[maxn * maxn];
char sub[maxn][maxn];
int cnt1 = 0, cnt2 = 0;

bool judge(char c) { //判断字符是否为合法变量名，即只能是字母、下划线、数字
    if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
        || (c >= '0' && c <= '9') || c == '_') return true;
    else return false;
}

void deal() { //对 str 进行处理，将所有的合法变量名按顺序存入 sub 中
    int i = 0, len = strlen(str);
    while(str[i] != '\0') i++; //无论输入的变量类型，先忽略变量类型
    while(i < len) { //当 i 超出范围时退出循环（也可以判断分号）
        while(i < len && !judge(str[i])) i++; //跳过所有的不合法字符，直至 i 超出范围或遇到“合法”字符
        while(str[i] >= '0' && str[i] <= '9') i++; //当在非法字符后遇到数字，必为常数，不予记录
        while(judge(str[i])) { //对满足变量名称类型的字符，循环记录到 sub 里
            sub[cnt1][cnt2++] = str[i++];
        }
        //完善字符串的末尾，并对 cnt1 自增，cnt2 置零，准备处理下一串字符
        sub[cnt1][cnt2] = '\0';
        cnt1++, cnt2 = 0;
    }
}

bool defined(char temp[]) { //检查变量名是否已经定义过
    for(int j = 0; j < cnt1; j++) {
        if(strcmp(temp, sub[j]) == 0) return true;
    }
}
```

```

    }
    return false;
}

void deal2() {
    int i = 0, len = strlen(str);
    while(i < len) {
        char temp[maxn]; //类似地，处理合法变量名保存至 temp，在循环中定义可以避免
        清零的麻烦
        while(i < len && !judge(str[i])) i++; //当 i 超出范围时退出循环，不一定有分号
        while(str[i] >= '0' && str[i] <= '9') i++; //当在非法字符后遇到数字，必为常数，不予
        记录
        while(judge(str[i])) { //对满足变量名称类型的字符，循环记录到 temp 里
            temp[cnt2++] = str[i++];
        }
        temp[cnt2] = '\0', cnt2 = 0;
        if(!defined(temp)) { //若 temp 未定义，打印 temp
            printf("%s ", temp);
        }
    }
    printf("\n");
}

int main() {
    gets(str);
    deal();
    gets(str);
    deal2();
    return 0;
}

```

自测用例 1:

输入:

```

int x12, y=1, num_stu=89,a[30],*p;
Sum=num+ (x12*y) -( *p)

```

输出:

Sum num

自测用例 2: (*比较坑的情况，存疑)

输入:

```

int *p, sum = 10;
p = &abc + &sum;

```

输出：

abc

自测用例 3：

输入：

```
int x, y=1, *p = &sum;
```

```
*p = 2 + y = x;
```

输出：

（换行）

自测用例 4：

输入：

```
double double123 = 456.78, f[20][30][10];
```

```
dou = c[1][2][3]
```

输出：

dou c

自测用例 1 考虑的是括号和指针并存的情况，卡掉一些扫描完括号就直接开始扫描变量的人，坑不是很大；自测用例 2 考虑的是指针和取地址运算符的问题，因为题目对基本运算定义的非常模糊，到底取地址运算符行不行，以及甚至 `pow` 函数那些的算不算，如果认为取地址运算符算的话，那么这条是应该考虑上的。事实上，这也是扫描合法字符的好处，因为非法字符总是很难想全的，所以合法取反就可以很好地规避掉这一点，如果题目没说，这种我们能考虑还是应该考虑上的，能多拿点分是一点嘛。

自测用例 3 同时考虑了连等、常数和空输出的问题，在第一句话中倒无所谓，如果把 1 也扫成了一个变量事实上也不影响，但如果出题老师非常心机婊的话，就会设置几千个常数，如果你把常数也扫进去，就很可能超出了你定义的数组范围，爆内存，毕竟题目说的是不超过 1000 个变量（虽然这句话是我后加的），但人家变量指的是合法的变量啊，合法 999 个，常数再来个几百个，那不就 GG 了嘛。

自测用例 4 则考虑的是多维数组的问题。如果有的代码只进行一次 '[' 和 ']' 的扫描，就有可能输出 2、3，这个用例用于卡掉这些人。同样地卡掉一些对非法字符做判断的人，看他们能不能想到小数点。

题目三：

这是一道非常常规的二叉树问题（甚至可以操作为完全二叉树问题），可能是因为突然第一年加入树，所以说难度不大，也没什么坑。我认为这道题最好的地方在于，没有一开始给 `N`，这样的输入方式可能打了很多人的措手不及，突然不知道怎么判断结束了（我拿到的版本是没有给 `N` 的，红果研是给了 `N` 的，我们这里就权当一种练手了，习惯下不给 `N` 的情况该如何处理问题）。

以及这种输入父亲、孩子的输入方式，和 18 年的题目是非常类似的（这里不做过多对比，以免很多同学还没有做到 18 年的题目），所以 18 年的考生上考场应该就轻松很多了（考虑到他们应该做了 17 年的题目）。

对于这种输入方式，我认为比较常规的做法应该是使用 `map<int, *node>` 这样的映射建

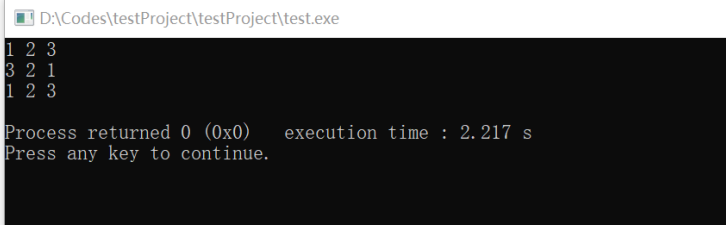
立二叉树，无论动态还是静态都非常适用，可以省去查找结点带来的很多麻烦。18 年的题目给了映射建动态二叉树的做法（大家如果先做了 17 年的题目，18 年不妨可以考虑换个方式），这里我之后会给出静态**完全**二叉树的做法（可以/2 就得到了父亲信息），但对于**完全**二叉树来讲存在风险，因为数据规模是个问题，除非是像 AVL 这种问题，或者题目给出了 n 的最大上限，或是层数（深度）的上限（对于完全二叉树，层数决定规模），如果 $2^{\text{层数}}$ 很大，就不推荐建完全二叉树了，可以退而常规地建静态二叉树，并对数添加父节点的指针。

而这道题目我用的是并查集，要比静态二叉树简单许多，省去了建树的麻烦（不知道是老师故意留有余地，还是真的可以偷鸡），因为像这种共同祖先问题，用并查集是非常好向上追溯的，如果建树还需要对树添加父节点的指针，就大大增加了复杂度。我认为并查集应该是最简洁也最快捷的方法了。

而对于不给 N 的输入方式，这就有点类似于多点测试了，这里我的判断方法可能有些复杂，我是用的“%s %s%c”（根节点那一行先扫描，之后的每一行先扫描两个，然后跟着一个 char 看是换行还是空格来决定是否进入检查），这个题目一开始是没有说一定有两个孩子的（这个信息是我后来加上去的），但是测试用例给的都是两个孩子，所以我就认为输入应该都是三个，检查的时候才是两个；如果不用这种偷鸡一点的方式，那自然就要对所有的第二个输入成员都 check 一遍存在与否，这其实是很繁琐的。但必须承认，这样写更加稳健，考场上如果时间允许、运行时间也允许，那么自然都检查一下是最稳的。

另外我因为这道题还发现了一个非常毁三观的事情：请看一下部分的代码和运行结果：

```
1
2 #include <stdio>
3
4 int main() {
5     int num[20], i = 0;
6     scanf("%d%d%d", &num[i++], &num[i++], &num[i++]);
7     printf("%d %d %d\n", num[0], num[1], num[2]);
8     i = 0;
9     printf("%d %d %d\n", num[i++], num[i++], num[i++]);
10 }
11
```



这意味着 printf 和 scanf 事实上是从后往前读入和打印数据的（据说与编译器有关系），所以这道题目本来我在第一句话是 `int i = 0; scanf("%s %s %s", m[i++].name, m[i++].name, m[i++].name);` 然后在测试一个用例的时候，就死活不对，然后开了 watches 才发现这个事情，简直惊呆了。所以我们拆分扫描其实也从一定程度上避免了这个问题。当然了如果用三个 temp 来扫描，不与自增运算符混用当然就不会出现这个问题，但就是时间空间复杂度都要多一点。

以下是我使用并查集做法的代码部分：

```
#include <stdio>
#include <cstring>
const int maxn = 1010;

int father[maxn];
```

```

struct member {
    char name[maxn];
    int depth;
} m[maxn], temp;

int main() {
    int i = 3, j, k;
    char check;
    scanf("%s %s %s", m[0].name, m[1].name, m[2].name);
    m[0].depth = 1;
    m[1].depth = m[2].depth = 2;
    father[1] = father[2] = 0;
    while(1) {
        scanf("%*c%s %s%c", temp.name, m[i++].name, &check);
        //找出 temp 所在的编号
        for(j = 0; j < i - 1; j++) {
            if(strcmp(temp.name, m[j].name) == 0) {
                break;
            }
        }
        if(check == '\n') break; //当输入只有两个就换行时，退出循环，开始检查
        scanf("%s", m[i++].name); //若不是，继续扫描第三个变量
        //标记其父亲和深度信息
        father[i - 2] = father[i - 1] = j;
        m[i - 2].depth = m[i - 1].depth = m[j].depth + 1;
    }
    //退出扫描后，要求检查 temp 和 m[i-1]深度差和共同的父节点
    //先找出 m[i-1]之前所在的编号，temp 所在编号为 j
    for(k = 0; k < i; k++) {
        if(strcmp(m[i - 1].name, m[k].name) == 0) {
            break;
        }
    }
    int d = m[k].depth - m[j].depth;
    if(d > 0) { //若 k 比 j 深度更深
        int dtemp = d;
        while(dtemp--) { //k 向上追溯至同一深度
            k = father[k];
        }
        while(strcmp(m[k].name, m[j].name) != 0) { //若同深度时不等，继续向上追溯
            k = father[k];
            j = father[j];
        }
    }
}

```

```

        printf("%s %d", m[j].name, d); //输出此时的共同结点，与原深度差
    }
    else { //若前者比后者深度更深
        int dtemp = -d;
        while(dtemp--) { //追溯至同一深度
            j = father[j];
        }
        while(strcmp(m[k].name, m[j].name) != 0) { //若同深度时不等，继续向上追溯
            k = father[k];
            j = father[j];
        }
        printf("%s %d", m[j].name, -d); //输出此时的共同结点，与原深度差
    }
    return 0;
}

```

接下来给出静态**完全**二叉树的做法，也还是比较简单的。我拿到的题目并没有给出数据的规模，所以还是再强调一下，使用静态**完全**二叉树一定要审慎，虽然完全二叉树可以简单地获得父亲信息，但如果数据规模极端的话（比如一条线）很容易爆内存，并且浪费空间。

对于这道题来讲，因为还需要深度信息，那么确实完全二叉树是非常方便快捷的。而这种方法最精髓的地方我认为在于 `string` 数组的使用，可以很好地存储姓名信息而不用开结构体。大家可以在此基础上，自行尝试正常建静态二叉树+`father` 数组的方法，和动态二叉树作为两种常规方法解决此类题目。

```

#include <iostream>
#include <cstdio>
#include <cstring>
#include <vector>
#include <string>
#include <map>
#include <cmath>
using namespace std;

```

```

string member[100];
map<string, int> m;

```

```

int checkDepth(int num) {
    int d = 0;
    while(num) {
        num /= 2;
        d++;
    }
    return d;
}

```

```

int main() {
    string s, s1, s2, s3;
    int i;
    cin >> s1 >> s2 >> s3;
    member[1] = s1;
    member[2] = s2;
    member[3] = s3;
    m[s1] = 1, m[s2] = 2, m[s3] = 3;
    getchar(); //在 cin 和 getline 前吸收掉换行
    while(1) {
        getline(cin, s);
        s1.clear();
        s2.clear();
        s3.clear();
        for(i = 0; s[i] != ' '; i++) {
            s1 += s[i];
        }
        for(i = i + 1; s[i] != ' ' && i < s.length(); i++) {
            s2 += s[i];
        }
        if(i == s.length()) break;
        for(i = i + 1; i < s.length(); i++) {
            s3 += s[i];
        }
        int f = m[s1];
        member[2 * f] = s2;
        member[2 * f + 1] = s3;
        m[s2] = 2 * f, m[s3] = 2 * f + 1;
    }
    //退出后查询 s1 和 s2 共同的父节点和深度差
    //先获得 s1 s2 在数组中的编号
    int num1 = m[s1], num2 = m[s2];
    int d1 = checkDepth(num1), d2 = checkDepth(num2);
    int d = abs(d1 - d2); //记录此时深度差信息
    while(d1 > d2) { //若 s1 比 s2 深度更深
        num1 /= 2;
        d1--;
    }
    while(d1 < d2) { //若 s2 比 s1 深度更深
        num2 /= 2;
        d2--;
    }
    while(num1 != num2) { //此时处于同一深度，若不相同则均向上追溯

```



```

        num1 /= 2;
        num2 /= 2;
    }
    cout << member[num1] << " " << d << endl;
    return 0;
}

```

自测用例 1:

输入:

Ye Shu Ba
 Shu Ge Mei1
 Ba Self Mei2
 Ge Son1 Son2
 Son2 Son1

输出:

Ge 0

自测用例 2:

输入:

Ye Shu Ba
 Shu Ge Mei1
 Ba Self Mei2
 Ge Son1 Son2
 Son2 Self

输出:

Ye 1

自测用例 3: (存疑)

输入:

Ye Shu Ba
 Shu Ge Mei1
 Ba Self Mei2
 Ge Son1 Son2
 Son2 Son2

输出:

Son2 0

以上这几个用例就是根据题目用例进行的修改,没有太多的坑。但是大家需要注意的是,像这种挂载父节点关系的问题,往往有的时候会出现错误挂载的情况,所以一定要在测试用例的基础上增加层数规模、多测试几遍,比如用例 1 是对同层测试(尽量选择底层的同层),题目的用例是对同一叉中的测试,那我们就设计用例 3 对交叉进行测试(跨得越远越能检查

两叉是不是都没问题)，比如我在没写方法二的 `getchar()` 吸收换行时，同叉就都没问题，跨叉却出现了问题，是很容易被我们忽视掉的。

自测 3 不一定用。因为题目没有说清楚输入相同成员时候怎么算，所以如果没有特判我觉得也没关系，应该不会给出这样的用例，但最好能想到这一层，并按照可能的逻辑设计。

2016 年北京航空航天大学复试上机试题

题目一：逆序数

给定一个数 n ($0 < n < 1000000$)，将这个数的各位顺序颠倒，成为逆序数 m 。例如 1234 的逆序数为 4321。如果 m 是 n 的 k 倍 (k 是整数)，那么输出 $n*k=m$ 。例如输入 1089，输出 $1089*9=9801$ 。如果 m 不是 n 的整数倍，那么输出 n 和 n 的逆序数，例如输入 1234，输出 1234 4321。再例输入 23200，输出 23200 00232。已知输入开头不包含多余的 0。

测试用例 1:

输入:

1089

输出:

1089*9=9801

测试用例 2:

输入:

1234

输出:

1234 4321

测试用例 3:

输入:

23200

输出:

23200 00232

题目二：enum 输出

enum 是枚举 (enumerate) 的缩写，在 C 语言中是一种基本数据类型，它可以让数据更简洁，更易读。规定各项的值只能是 int 型变量，若未定义第一项的值，则默认第一项的值从 0 开始；若已定义某项的值，则从该项后值递增加一。

输入一行 c 语言的 enum 定义语句，且符合 C 语言语法。各项在大括号之间，并保证大括号之间无空格。要求输出各项的名称和对应的值，占一行，且名称与值之间有一个空格。保证输入的长度不超过 1000000 个字符，且项至多为 1000 个，每个项的长度至多为 1000 个字符。

测试用例 1:

输入:

```
enum BOOL{true,false};
```

输出:

true 0

false 1

测试用例 2:

输入:

enum

date{ JAN=1,FEB,MAR,APR,MAY,JUN,JULY,AUG,SEP,OCT,NOV,DEC,MON=1,TUE,WED,THU,FRI,SAT,SUN,found=1949};

输出:

JAN1

FEB2

MAR3

APR4

MAY5

JUN6

JULY7

AUG8

SEP9

OCT10

NOV11

DEC12

MON1

TUE2

WED3

THU4

FRI5

SAT6

SUN7

found1949

2016 年北京航空航天大学复试上机试题解析及参考答案

可能因为 16 年以前都不太让用 C++ 的缘故，所以 16 年的题目比较简单，也没什么坑，正常来讲应该是一半个小时之内就写完了。

我手头搜集到的资料对题目二给的信息非常少，所以我加入了一些规范内容，应该是还原了题目的信息也保证了难度，如果有余力的话可以看看题目二可能增加各种空格的话如何处理。

题目一：

这道题考察字符串处理的基本功，题目没有太大难度，只要读懂题意应该问题不大。题目有一个小的坑就是必须是 m 是 n 的整数倍，而不能反过来，这一点其实在看到样例三的时候应该就反应过来这一点，以及题目其实说的相当精简了：“输入开头保证没有多余的零”，就意味着不会出现输入 00232，而输出 $00232*100=23200$ 这种情况了。

然后因为不涉及大整数运算，我就用了两种方法，一种是常规的字符串处理，因为正常整数转化为数组就是会逆序的，所以说非常方便，直接同时得到逆序数和逆序数的值。还有一种方法是利用了 C++ 的 `algorithm` 算法里的 `reverse`，这样直接通过扫描 `%s`，然后写一个数组转 `int` 的函数就行了（乘 10 并累加，非常好写），但因为 `reverse` 只能对单数组操作，所以说相对而言是用空间换了时间，应该不如第一种方法快。

这里给出两种方法的代码，并强调一些其中的小细节。

```
#include <cstdio>

const int maxn = 1000010;

int n, m, cnt = 0;
int a[maxn];

int revNum(int n) { //将输入 n 逆序，数值形式记录在 m 中，数组形式记录在 a[] 中
    int num = 0, temp;
    do{
        temp = n % 10;
        n /= 10;
        a[cnt++] = temp;
        num = num * 10 + temp;
    } while(n > 0);
    return num;
}

int main() {
    scanf("%d", &n);
    m = revNum(n);
    if(m % n == 0) {
        printf("%d*%d=%d\n", n, m / n, m);
    }
    else {
```

```

        printf("%d ", n);
        for(int i = 0; i < cnt; i++) {
            printf("%d", a[i]);
        }
        printf("\n");
    }
    return 0;
}

```

这种做法就是同时得到逆序数和逆序数的值，事实上因为定义的全局变量，可以不输入 `n[]` 和返回 `sum`，在函数内赋值 `m=sum` 亦可；然后需要注意的细节就是，整形数组不能用 `%s` 打印，比如最后打印 `a` 的这段，必须通过 `for` 循环打印整型数组：

```

printf("%d ", n);
for(int i = 0; i < cnt; i++) {
    printf("%d", a[i]);
}
printf("\n");

```

而不能是：

```
printf("%d %s\n", n, a);
```

因为有可能会有人测试个 1089 发现正确输出了（因为走的时候前边的 `if` 分支），然后咋也找不到后边的错误……

然后方法二就是利用 `algorithm` 的 `reverse`，简单易读，但在时效上可能没那么好。在这种情况下可以优先打印输入的 `n`（无论走哪个分支条件，最开始都是打印输入的 `n`），然后这个 `n` 就没用了，可以被用来 `reverse`。

```

#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int maxn = 1000010;

char str[maxn];

int toInt() { //将 str 转换为数值
    int num = 0;
    for(int i = 0; i < strlen(str); i++) {
        num = num * 10 + str[i] - '0';
    }
    return num;
}

int main() {

```

```

scanf("%s", str);
int n = toInt();
printf("%d", n);
reverse(str, str + strlen(str));
int m = toInt();
if(m % n == 0) {
    printf("%d=%d\n", m / n, m);
}
else {
    printf(" ");
    for(int i = 0; i < strlen(str); i++) {
        printf("%d", str[i] - '0');
    }
    printf("\n");
}
return 0;
}

```

自测用例 1:

输入:

1

输出:

1*1=1

自测用例 2:

输入:

9801

输出:

9801 1089

这里我唯一能想到的坑就是个位数和 `nm` 反过来输入。我当时做的题目没有给 `n` 的范围，所以我还特别考虑了 0，发现会卡机，然后想了想 0 乘任何数都是 0，肯定卡机，遂在之后完善题目时特意强调了 0 的问题。

注意反过来输入大数的话是不会被判定为整除的。

题目二:

这个题目还是比较新颖的，但本质上也是一道输入合法语句而实现其功能的题目，会看到 17 年的题目也是同样的风格。相较 17 年第二题来讲这道题就良心多了（不做过多展开，以防还有人没做），因为有大括号在，输入变量的风格不用做过多的判定。

我认为基本的思路就是以结构体来存储数据（应该会很直观地就想到），主程序里先找到大括号作为起始，然后逐个分割变量，检查到等号则进行分支赋值，检查到右括号停止。

细节方面大家捋清楚自增运算符的使用起始就可以了，最简单最稳的方式当然是不在

while 循环里使用自增运算符，手动拉大括号自增，但这样看着自然是比较丑、繁琐；不过也不要过度使用自增运算符，因为受到编译器的影响会出现很多奇怪的问题（详情见 17 年的题目三中对于 printf 和 scanf 的讨论），有的时候还要稳妥起见分开写，比如下面这段：

```
e[j++].num = e[j - 2].num + 1;
```

编译器到底是先执行左边的 e[j++].num，让 j 已经自增然后再运算右边呢，还是说先调取 e[j - 2].num 的值、运算，再赋给左边呢？因为 j 是否先自增，设计了右边运算的值到底 j 实际上是 -1 了还是 -2 了，说的有点儿绕，大家如果反应不过来就多读几遍。

所以虽然这样写起来很简洁，但是最稳妥的方式应该是

```
else e[j].num = e[j - 1].num + 1;
j++;
```

虽然我在 codeblocks 里测试的两种语句效果是一样的，但是因为学校用的机判不知道是如何编译的（提交的又不是 .o 文件），所以还是稳一点比较好。

此外，这道题在我拿到的版本非常光秃秃，甚至关于 enum 的任何解释都没有，基本等同于直接上了两个测试用例。我进行了一定的背景完善和语法解释，增加了一些输入要求以便于更好地处理字符串，然后也应该会更加符合机试题的风味。

以下是代码部分：

```
#include <stdio>
```

```
char str[1000010]; //输入记录在 str 中
```

```
struct ENUM { //enum 类似于结构体，将变量名和值捆绑在一起
```

```
    char name[1010];
```

```
    int num;
```

```
}e[1010];
```

```
int main() {
```

```
    scanf("%s", str);
```

```
    int i = 3, j = 0, k = 0;
```

```
    while(str[++i] != '{'); //先找到定义所用的‘{’作为起始
```

```
    while(str[i++] != '}') { //若此时不为‘}’则一直循环记录定义项
```

```
        while(str[i] != '=' && str[i] != ',' && str[i] != '}') { //循环赋值项目名
```

```
            e[j].name[k++] = str[i++];
```

```
        }
```

```
        if(str[i] == '=') { //若因等号退出，则特殊记录该项的值
```

```
            int sum = 0;
```

```
            while(str[++i] >= '0' && str[i] <= '9') {
```

```
                sum = sum * 10 + str[i] - '0';
```

```
            }
```

```
            e[j].num = sum;
```

```
        }
```

```
    else { //否则为前一项值+1，第一项默认为 0
```

```
        if(j == 0) e[j].num = 0;
```

```
        else e[j].num = e[j - 1].num + 1;
```



```

    }
    j++; //无论何种情况，记录下一变量前统一进行 j++
    k = 0; //对计数器 k 归零
}
for(i = 0; i < j; i++) { //打印所有的 enum 项目和其值
    printf("%s %d\n", e[i].name, e[i].num);
}
return 0;
}

```

自测用例 1:

输入:

```
enum _B1O2O3L{true,false=0};
```

输出:

true 0

false 0

自测用例 2: (存疑)

输入:

```
enum _B1O2O3L{true,false=02};
```

输出:

true 0

false 2

我认为这道题应该没什么坑，我能想到的主要就是卡掉一些手动筛选变量名的人（其实完全没有必要，找左大括号其实就好了），然后还有就是等号赋值在最后的时候，与右大括号一起判定，可能会出现问题。

对于用例 2 存疑，因为不知道 02 这种输入是否合法（如果认为可以编译的就是合法的，那么这种就是合法的，就应该也被考虑在内），主要就是如何读入这个数值的问题，以及在使用例 1 里也测试了可不可以读入 0 的问题。虽然这里一般不会出问题，但大家一定要牢记，当整数转字符串的时候（除基取余），一定要用 `do-while` 循环，就可以避免特判 0 的问题。这里因为一般是字符串转整数，就不太会出问题。

2015 年北京航空航天大学复试上机试题

题目一：相亲数

相亲数，又称亲和数、友爱数、友好数，指两个正整数中，彼此的全部约数之和（本身除外）与另一方相等。毕达哥拉斯曾说：“朋友是你灵魂的倩影，要像 220 与 284 一样亲密。”

输入两个正整数 a 和 b ($1 < a, b < 1000000$)。若 a 的所有约数（包括 1，但不包括 a 本身）的和等于 b ，且 b 的所有约数（包括 1，但不包括 b 的本身）的和等于 a ，则两个数是相亲数。

输入保证 a, b 不相等，且 a, b 的因子数目均不超过 1000 个。要求用两行分别输出两个正整数 a, b 和其约数求和的式子，并求出其约数和。再换行输出 1 或者 0，表示这两个数是否为一对相亲数。要求 a, b 与和式之间有逗号和空格，且和式中的约数顺序是从小到大的。

测试用例 1:

输入:

220 284

输出:

220, 1+2+4+5+10+11+20+22+44+55+110=284

284, 1+2+4+71+142=220

1

测试用例 2:

输入:

1952 2015

输出:

1952, 1+2+4+8+16+32+61+122+244+488+976=1954

2015, 1+5+13+31+65+155+403=673

0

测试用例 3:

输入:

5 7

输出:

5, 1=1

7, 1=1

0

题目二：模拟鼠标点击窗口

先输入一个整数 n ($2 \leq n \leq 10000$)，表示桌面窗口的数量，再输入 n 行，每行 5 个数，分别为窗口 ID，窗口右下角横坐标，右下角纵坐标，左上角横坐标，左上角纵坐标（坐标均以屏幕左下角为原点，并假设屏幕无限大，窗口均为矩形窗口），先输入的窗口叠放在后输入的窗口的上面。

再输入 m 行，表示 m 次点击，每行两个数，分别表示点击的横坐标和纵坐标。

所有的输入内容保证都是整数，且均在 `int` 型表示的范围内。若窗口有重合，则认为点击到更靠上的窗口；若点击到窗口的边框上，也算点击到该窗口。一旦点击到某一窗口，则该窗口将会变为最上面的窗口。要求在 `m` 次点击后，按窗口叠放次序从上到下依次输出窗口的 `ID`。每个 `ID` 的后面都有一个空格，并在末尾换行。

测试用例 1:

输入:

```
2
1 5 1 1 5
2 7 1 3 5
3
1 2
4 3
6 4
```

输出:

```
2 1
```

测试用例 2:

输入:

```
3
3 5 1 1 4
4 7 3 3 6
2 8 2 4 5
3
5 3
8 4
4 4
```

输出:

```
2 3 4
```

题目三：统计词语

输入一段含标点的英文语段（若干行，以文件终止符 `EOF` 结束，在命令窗口中以 `ctrl+z` 表示），保证每行输入的单词都是完整的，词与词之间至少会出现空格或标点，且词不会被换行所分割。

保证输入的所有字母都是小写，且每行输入的长度不会超过 100000，输入单词总数不会超过 1000 个，每个单词的长度不超过 100。

统计这段话中出现的所有单词，要求按字典顺序输出词语，且不能重复，每输出一个词换一行。

测试用例 1:

输入:

```
we are family.
you and me are family, you are my sister.
```

输出:

and

are

family

me

my

sister

we

you

测试用例 2:

输入:

supercalifragilisticexpialidocious!

输出:

supercalifragilisticexpialidocious

2015 年北京航空航天大学复试上机试题解析及参考答案

15 年的题目难度适中，我认为是一套非常好的关于数学功底、字符串处理的题目，只不过和 18、19 年已经倾向于数据结构的风格偏差较大，但可以作为在字符串问题上很好的知识补充。

题目一：

这是一道基本的数学问题，很符合北航第一题的风格，难度稍微大了一些。我在做这道题的时候差点没做出来，主要是因为《算法笔记》上专门有一章在讲分解质因子，以及如何对所有的质因子求和的问题，不过这种讲法很不适合打印所有因子，我写到一半发现相亲数好判断，但打印很烦，只能递归枚举打印，然后又发现没法从小到大……

所以我只能想到用笨办法从头扫描到 $n/2$ 了（我去看了红果研给的做法和一些大佬学长的做法居然也都是扫描到 $n/2$ ，事实上扫描到 n 也并不会差太多），那么如果运行时间允许的话，这道题就非常普通且简单了。

最后需要注意一点的是，边扫描边求和边打印，一次循环干三件事应该是最优的代码，但是大家在考场上应该不会想到这么完美的代码，大部分人可能会将求和和打印放在一个函数里，然后两次调用函数，在时间上是并没有差太多的。

这里给出最优版本的代码：

```
#include<stdio>

int a, b, suma = 1, sumb = 1;

int main(){
    scanf("%d%d", &a, &b);
    printf("%d, 1", a);
    int i;
    for(i = 2; i <= a / 2; i++){
        if(a % i == 0) {
            suma += i;
            printf("+%d", i);
        }
    }
    printf("=%d\n", suma);
    printf("%d, 1", b);
    for(i = 2; i <= b / 2; i++){
        if(b % i == 0) {
            sumb += i;
            printf("+%d", i);
        }
    }
    printf("=%d\n", sumb);
    if(suma == b && sumb == a) printf("1\n");
    else printf("0\n");
    return 0;
}
```

```
}
```

自测用例 1:

输入:

898216 980984

输出:

898216,

1+2+4+8+11+22+44+59+88+118+173+236+346+472+649+692+1298+1384+1903+2596+3806+
5192+7612+10207+15224+20414+40828+81656+112277+224554+449108=980984

980984,

1+2+4+8+47+94+188+376+2609+5218+10436+20872+122623+245246+490492=898216

1

自测用例 2:

输入:

1952 1954

输出:

1952, 1+2+4+8+16+32+61+122+244+488+976=1954

1954, 1+2+977=980

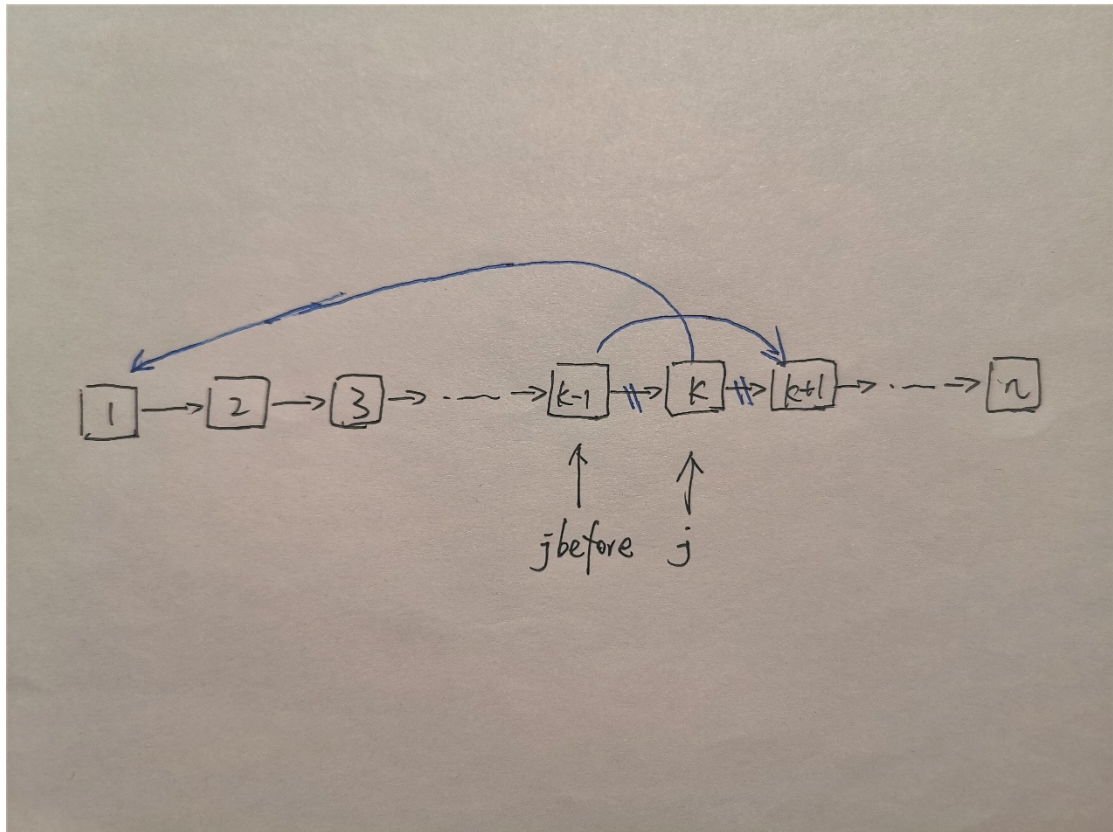
0

自测用例没有太多坑，用例 1 给出一个范围边界的极限相亲数，看看打印是否满足了题意要求。用例 2 是根据题目的测试用例修改的（虽然这个用例也是我后来编的，1952 是北航建校时间），这里修改主要是为了数字 2 等于数字 1 的因子和，而反之不等于，看有人会不会特判为 1；然后再讲两个数字反过来输入，即先输入 1954，再输入 1952，这样就可以反过来判断了，因为可能会有人误打误撞只判断 $\text{suma}=\text{b}$ ，而没有判断 $\text{sumb}=\text{a}$ ，这样仍然可以通过题目给的用例，但却通过不了自测用例，这里稍微注意下即可。

题目二:

这道题我的方法横向对比之后应该还是比较简洁的了。我最开始就自然地想到，这样的问题应该建立结构体，设置好窗口的范围（红果研居然不开结构体，用二维数组记录窗口和判断？？），然后窗口编号信息、层数，这些都统统绑在一起，模拟移动的时候就不会乱。然而我写着写着发现这实际上是一道链表问题。

如下图，比如当你点到了一个窗口（pointer j 指向的那个窗口），你需要做的是把 $j\text{before}\rightarrow\text{next}$ 指向 $j\rightarrow\text{next}$ ，然后把 $j\rightarrow\text{next}$ 指向 head（注意这个操作不能反过来做，如果先把 $j\rightarrow\text{next}$ 指向 head，那么 j 的下一个就丢失了），而如果单纯通过记录“层数”，那么就需要修改 j 之前的所有层数，这显然是不行的。



所以在在大思路上我实际是用的 two pointers 加静态链表（有兴趣者可以自行尝试动态链表）。

然后再说算法细节部分，关于点击窗口的问题。我们可以按照层数从上到下来讨论，看是否在这个窗口范围内（即点击的 x 、 y 是否在窗口的 x 、 y 范围内），如果不在就继续讨论下一个窗口，这样做的好处是不必讨论窗口重叠的范围（如果在重叠部分点击了下层窗口，那么在之前讨论上层时应该已经被点击过）。然后设计一个布尔型 **flag** 表示是否点击到窗口，点击到桌面和点击了第一个窗口实际上都是等价的，所以这两种情况要排除掉，此外情况便做如上图的指针移动。

如下是代码部分：

```
#include <stdio>

struct window {
    int rx, ry, lx, ly;
    int num, next; //表示窗口编号和下一层窗口
}w[10010];

int main() {
    int n, m, click_x, click_y, top = 0; //top 表示此时的第一层窗口
    int j = top, jbefore; //j 表示此时检查的窗口层，jbefore 用于记录其上一层，便于之后修改层序
    scanf("%d", &n);
    for(int i = 0; i < n; i++) { //按序输入，并设置好下一层窗口，初始时下标越小越靠上
```

```

scanf("%d%d%d%d", &w[i].num, &w[i].rx, &w[i].ry, &w[i].lx, &w[i].ly);
if(i != n - 1) {
    w[i].next = i + 1;
}
else w[i].next = -1; //最后一个窗口，设置其下一窗口为-1，表示为空
}
scanf("%d", &m);
for(int i = 0; i < m; i++) {
    scanf("%d%d", &click_x, &click_y);
    bool flag = true;
    while(click_x < w[j].lx || click_x > w[j].rx || click_y < w[j].ry || click_y > w[j].ly) {
        //若不在当前窗口范围内，且还有未检查的窗口
        if(w[j].next > 0) { //若 j 的下一个窗口不为空，则向后移动 j 和 jbefore
            jbefore = j;
            j = w[j].next;
        }
        else {
            flag = false; //下一个窗口为空，也就是到了最后一个窗口，那么设置 flag
            //表示什么也没点到
            j = top; //重新将 j 赋值为 top，准备下一次操作
            break; //从 while 循环中跳出
        }
    }
    if(j != top && flag) { //此时 j 不为第一个窗口且确实点到了某窗口
        w[jbefore].next = w[j].next;
        w[j].next = top;
        top = j;
    }
    //除此之外，点到第一个窗口或者所有窗口都没点到，都可以认为什么都没做，不用 else
}
while(j != -1) { //循环打印，直至到最后一个窗口
    printf("%d ", w[j].num);
    j = w[j].next;
}
printf("\n");
return 0;
}

```

自测用例：

输入：

2

1 2 1 1 2

2 3 2 2 3

3
2 2
3 3
4 4

输出:

2 1

这个自测用例我想的是这样几点：1，第一次点击最上层窗口，看会不会有人乱掉；2，点击最后一层窗口，看会不会有人乱掉；3，点击桌面，看会不会有人乱掉；4，以上所有点击都是边边角角，看关于边界等于问题会不会有人乱掉。所以其实这一个自测用例就可能卡住绝大多数的代码了，其他的常规测试用例大家自己编几个检查一下就好了，这个题应该是没有太多坑的。

题目三：

这个题也是基本的字符串处理问题，相对前几年来讲坑还是很少的。扫描单词（字母）和非单词（非字母）应该是问题不大的，大家只要搞清楚 **while** 循环和自增运算符的使用即可。同样地，不建议过分使用，具体问题可以见 16、17 年的对应题目解析（大家做到了就知道了，这里不展开，以防还有没做过的）。

所以真正的难点应该在于如何按字典序排序和不重复。这里我会给出两种方法，第一种是基本的扫描，查重，扫描完所有的之后，写一个 **cmp** 函数统一 **sort**；第二种就非常非常的简单了，写完我都惊叹了怎么可以这么简单，基本上在 10 分钟左右就写出来了，也就是用了 **STL** 容器函数之 **set**，其内部已经用红黑树自动排序，对于这题来说简直就是“正中下怀”，代码十分简洁。

考虑到 15 年并没有放开 C++ 的使用，方法一自然更通用一些，方法二如果不会的当然也要掌握，这题用 **set** 简直就是砍瓜切菜，如果将来遇到这样的题目，自然不要放过使用 **STL** 的好机会。

对于方法一，要强调的一点是关于二维数组的 **sort** 问题。先看一种典型错误代码：

```
char str[1010][105]; //使用二维数组记录查到的单词

bool cmp(char a[], char b[]) { //用于 sort 的比较函数
    return strcmp(a, b) < 0;
}

int main() {
    sort(str, str + num, cmp); //num 为扫描到的单词数目
}
```

这就是我第一次做的时候的想法，对二维数组也像一维那样进行 **sort** 排序，我想我虽然没干过但应该也差不多吧，结果就报错了，直接给我弹到 **stl** 头文件里我连 **debug** 都没法 **debug**，就是语句错误了，不能这么写。

后来查了一下原因，也大概理解了，因为二维数组实际上是连续存储的，也就是 **str[0][0]**、**str[0][1]**、……、**str[0][1010]**、**str[1][0]**、**str[1][1]**、……**str[1][1010]**、**str[2][0]**、**str[2][1]**、……这样去存储的，自然不能只+**num** 长度来排序了，尤其是每个字符串的长度还不一定是相同的，所以说用二维数组+**sort** 基本上是 **GG** 了。

这时我能想到的有三种方法，供大家自主选择：

1.使用 struct，把二维数组改成结构体+一维数组，这样就可以很轻松地排序了（cmp 函数输入就是结构体了），这也是我给出代码的方法。

2.退而求其次，sort 的原身是 qsort，可以使用 qsort（我理解为指针版的 sort）完成任务。这里给出 qsort 部分的代码：

对于 qsort 的 cmp 函数，其返回值是 int，也就是直接返回 strcmp 函数的值即可。另外因为比较函数默认必须是 void* 的指针，那么在函数内要强转成 char*。

```
int cmp(const void* a, const void* b) { //使用 qsort 返回 int 型值
    return strcmp((char*)a, (char*)b);
}
```

然后 qsort 的语句要写成：

```
qsort(str, num, sizeof(str[0]), cmp);
```

其中第一个参数是要排序的数组的第一个元素的指针；然后是指向的数组中元素的个数，也即排序数组的个数；然后是数组中每个元素的大小，以字节为单位，这里可以填 sizeof(str[0])，也可以填 sizeof(char)*105（定义的第二维长度），总之就是第二维的长度；然后就是比较函数，根据返回的值大于、等于还是小于 0 进行相应的元素排序。

总之个中原理也不用掌握的太清楚，会用即可。

3.使用 vector 容器，这种类似于方法一，通过使用 vector<char>或者 vector<string>代替二维数组（后者应该比前者更好查重），比方法一可能会让人感觉比较蠢，这样做会显得更高级些（但是代码量肯定也相应的增加了，因为用不了 strcmp 等 string.h 里的函数了嘛）。

至于这种方式的代码就不给了，大家肯定都会写，我并不是很推荐这种做法，考试要紧嘛，哪儿有时间折腾 string 去。

接下来给出使用 struct 代替二维数组的整个题目的代码，对于 judgeChar 那个函数大家也可以不写，直接在 if 括号里判断。

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

struct STR { //使用结构体替代二维数组，便于排序
    char s[105];
}str[1010];

int num = 0;

bool judgeExist(char b[]) { //判断此单词是否已经记录过
    for(int i = 0; i < num; i++) {
        if(strcmp(str[i].s, b) == 0)
            return true;
    }
    return false;
}
```

```

bool judgeChar(char c) { //判断是否为英文字符
    if(c >= 'a' && c <= 'z')
        return true;
    else return false;
}

bool cmp(STR a, STR b) { //对结构体按字典序排序
    return strcmp(a.s, b.s) < 0;
}

int main() {
    char a[100010], temp[105];
    while(gets(a) != NULL) {
        int i = 0, j = 0;
        while(i < strlen(a)) { //当 i 不为末尾时重复循环
            while(i < strlen(a) && judgeChar(a[i])) { //当为英文字母时，记录单词
                temp[j++] = a[i++];
            }
            temp[j++] = '\0'; //养成习惯对字符串添加结尾符号
            if(!judgeExist(temp)) { //若未出现过，则将该单词 copy 到记录结构体里
                strcpy(str[num++].s, temp);
            }
            memset(temp, 0, sizeof(temp));
            j = 0;
            while(i < strlen(a) && !judgeChar(a[i])) i++;
        }
    }
    sort(str, str + num, cmp);
    for(int i = 0; i < num; i++) {
        printf("%s\n", str[i].s);
    }
    return 0;
}

```

然后给出使用 `set+string` 的做法。简单对比一下代码量就知道这种做法是真的简单……当然了，这样的做法需要大家对 `set`、`string` 的使用非常清楚（比如 `set` 的迭代器、`insert` 等，以及 `string` 打印必须要用 `iostream`，或者用 `c_str()` 等）。然后为了代码的简洁我就讲 `judgeChar` 的函数直接放到 `if` 括号里了。

```

#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <set>

```

```

using namespace std;

int main() {
    char a[100010];
    set<string> s;
    while(gets(a) != NULL) {
        int i = 0;
        while(i < strlen(a)) {
            string temp; //每次使用每次定义，相当于清空的操作
            while(i < strlen(a) && a[i] >= 'a' && a[i] <= 'z') {
                temp += a[i++];
            }
            s.insert(temp);
            while(i < strlen(a) && (a[i] < 'a' || a[i] > 'z')) i++;
        }
    }
    for(set<string>::iterator it = s.begin(); it != s.end(); it++) {
        cout << *it << endl;
    }
    return 0;
}

```

最后提醒大家在测试用例的时候千万不要忘了输入 `ctrl+z`，并且这也提醒我们，实际上北航的题目还是会考察多点测试的情况，大家要谨慎提防多点测试一些清空的情况，如果做惯了单点测试，转为多点很可能出现问题，以及也要对 `EOF` 有所了解，别到时候考试说 `EOF`，然后一脸懵逼 hhh

自测用例 1:

输入:

we are family.you and me are family, you are my sis

输出:

and
are
family
me
my
sis
we
you

自测用例 2:

输入:

“these violent delights have violent ends.” --shakespeare

输出：
delights
ends
have
shakespeare
these
violent

这道题目也没有太多的坑，我唯一想到的坑就是结尾不是标点和开头先出现标点的情况。自测用例 1 就是结尾不为标点，看看有没有一些代码的判断偷了懒，在对单词判断的部分不加上 `i<strlen(a)` 这个上位条件；而对于测试用例 2，其实是大有说法的！

一定要注意输出是上面那样的，而不是：

```
delights  
ends  
have  
shakespeare  
these  
violent
```

因为一般来讲，都是大循环里面套两个小循环，先扫描单词后扫描非单词，这样循环往复。那么在 `strcpy` 的判断条件处，就一定要判断字符串非空（在我的代码里是 `temp` 的 `j` 不等于 0，非空的话再进去赋结束符），否则就会出现上面这样的出现一个空行的情况，如果机判我认为这种多空行的情况是很可能扣分的。

此外应该没什么坑点了，多行且重复的情况题目用例已经帮忙测试了，只要通过了应该问题就不大了。