

Problems-for-AST

This repo contains the problems and solutions for the NJUSE Academic Corner.

算法天梯解析

筒子们是否觉得算法天梯很难呢？

（dlA：我每天早上 8 点就把这一天的题 A 了。

（rubbishB：我就是那个排名在最前面的 Rubbish。

（caijiC：我从第二天开始就不会做。。。)

无论你是因为什么原因不会做，相信看完这些题解之后都会大有收获！

那么我们开始吧！

- 算法天梯解析

- [Day 1](#)
- [Day 2](#)
- [Day 3](#)
- [Day 4](#)
- [Day 5](#)
- [Day 6](#)
- [Day 7](#)
- [Day 8](#)
- [Day 9](#)
- [Day 10](#)
- [Day 11](#)
- [Day 12](#)
- [Day 13](#)
- [Day 14](#)
- [Day 15](#)
- [Day 16](#)
- [Day 17](#)
- [Day 18](#)
- [Day 19](#)
- [Day 20](#)

Day 1

我们的目的就是排序，常见的排序方法有快速排序，归并排序等

这里说一下快速排序，方法其实很简单

1、找一个 key 值（一般为第一个元素），经过合适的移动将所有比 key 值小的值放到数组左边，大的放右边

2、对两边分别进行快速排序

可以明显地看出这是递归的方式

C/C++代码如下

```
void sort(int a[],int b,int e){//a为待排序数组, b=begin, e=end
    int i=b,j=e;
    int k=a[b];//k就是key值
    if(b>=e)
        return;
    while(i<j){

        while(a[i]<k&&i<e){//从前往后找到第一个不小于key值的（第一次一定是第一个）
            i++;
        }
        while(a[j]>=k&&j>b){//从后往前找第一个小于key值的
            j--;
        }
        if(i<j){//只要i比j小说明这两个是逆序，应该互换
            int temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    sort(a,b,j);//对前面快排
    sort(a,j+1,e);//对后面快排
    return;
}
```

注：1、该方法平均复杂度为 $O(n\log n)$ ，最坏复杂度为 $O(n^2)$ ，常用的冒泡排序等复杂度为 $O(n^2)$

2、因为本渣渣是用 c++写的，所以只能提供 C++的版本

3、对于某些没有使用 vector 的代码可以很容易地转成同学们熟悉的 C，有些不容易转，请多包涵（所有%3==1 的都是我的）

Day 2

（水题 😊）

这题我们把 1000 以内的整数一个一个遍历一遍就好了，然后每个整数进行检验，如果满足条件就输出。

至于具体的检验方法，我们可以把这个数分解成个位、十位和百位，然后对 0~9 的每个数字设置一个标志，当这个数字出现的时候就把标志置为一。如果某一个标志已经被置为一了说明对应的数字已经出现了两次，那么这个数就不满足要求。

具体代码见下：

```

#include <iostream>
using namespace std;
bool check(int num)
{
    bool used[10] = {false};    // 标志
    for (int i = 1; i < 4; ++i)
    {
        int n = num * i;
        for (int j = 0; j < 3; ++j) // 依次检查个位、十位和百位
        {
            int m = n % 10;
            if (m == 0 || used[m]) // 0 不允许出现在数字中
                return false;
            used[m] = true;
            n /= 10;
        }
    }
    return true;
}

int main()
{
    for (int i = 123; i < 333; ++i)
        if (check(i)) // 检查这个数是否符合要求
            cout << i << ' ' << i * 2 << ' ' << i * 3 << '\n';
    return 0;
}

```

Day 3

提示正解：[约瑟夫环问题](#)。

当然，事实上不用管这是什么问题—如果用数组，需要解决的是数组到了头怎么继续循环，即一个判断；如果用链表，那么用循环链表也可以很方便解决。

解决了框架，细节就是怎么把已经被抛弃的学生踢出去，注意顺序。

下面是一个 caiji 的代码(大家尽情骂他吧)。

```

#include <stdio.h>

int main() {
    int len = 0, k = 0, m = 0;
    scanf("%d %d %d", &len, &k, &m);
    int store[len]; // 存储编号
    for (int i = 0; i < len; i++) {
        store[i] = i + 1;
    }
    int currentk = 0, currentm = len - 1, count = 0;
    // currentk指以k为单位的循环指向哪个学生，currentm类似
    // count指已经踢掉多少个学生

```

```

while (1) {
    if (count == len) {
        break;
    }
    int countk = 1, countm = 1;
    // countk指本次循环已经数了多少个同学
    while (1) {
        if (countk == k) { // 数了k个
            if (store[currentk] < 0) { // 指向的学生是已经被踢掉的
                if (currentk == len - 1) {
                    currentk = 0;
                } else {
                    currentk++;
                }
                continue; // 继续循环
            } else {
                break; // 找到本次循环目标
            }
        }
        if (store[currentk] > 0) { // 指向的学生未被踢出
            countk++;
        }
        if (currentk == len - 1) { // 数组到头了
            currentk = 0;
        } else {
            currentk++;
        }
    }
    while (1) { // 和上面循环类似
        if (countm == m) {
            if (store[currentm] < 0) {
                if (currentm == 0) {
                    currentm = len - 1;
                } else {
                    currentm--;
                }
                continue;
            } else {
                break;
            }
        }
        if (store[currentm] > 0) {
            countm++;
        }
        if (currentm == 0) {
            currentm = len - 1;
        } else {
            currentm--;
        }
    }
    if (currentk == currentm) { // 找到班长了
        printf("%d", store[currentk]);
        break;
    }
}

```

```

    } else {
        printf("%d %d", store[currentk], store[currentm]);
        store[currentk] = -store[currentk];
        store[currentm] = -store[currentm]; // 踢人
        count = count + 2; // 踢了俩
        if (count != len) {
            printf(" ");
        }
        if (currentk == len - 1) {
            currentk = 0;
        } else {
            currentk++;
        }
        if (currentm == 0) {
            currentm = len - 1;
        } else {
            currentm--;
        }
    }
}
return 0;
}

```

Day 4

关于这题，题目的提示其实已经很明显了，就是只计次数，用一个 10001 大小的数组存一下每个数字出现了几次就好了
话不多说上代码

```

#include<iostream>
int t1[10001]={0}; //记录餐厅1和2的和
int t2[10001]={0}; //记录餐厅3和4的和
int main(){
    int n;
    std::cin>>n;
    int a[4][n];
    //输入，不做解释
    for(int i=0;i<4;++i){
        for(int j=0;j<n;++j){
            std::cin>>a[i][j];
        }
    }
    //通过两个循环把所有两个餐厅的和记录下来
    for(int i=0;i<n;++i){
        for(int j=0;j<n;++j){
            if(a[0][i]+a[1][j]<=10000){
                ++t1[a[0][i]+a[1][j]];
            }
            if(a[2][i]+a[3][j]<=10000){
                ++t2[a[2][i]+a[3][j]];
            }
        }
    }
}

```

```

    }
    int sum=0;
    //这里直接循环把乘积算出来
    for(int i=1;i<=10000;++i){
        sum+=(t1[i]*t2[10000-i]);
    }
    std::cout<<sum;
}

```

Day 5

（讨厌的高精度水题 😊）

这题没什么好说的，直接写一个高精度乘法就行了。

（朴素）高精度乘法的本质就是把多位数相乘分解为每一位的数相乘再乘上位数，和我们手算多位数乘法的原理相同。

具体而言，假设两个两位数相乘：

$$\begin{aligned}
 & \overline{a_1 a_2} \times \overline{b_1 b_2} \\
 &= (a_1 \times 10 + a_2)(b_1 \times 10 + b_2) \\
 &= a_1 b_1 \times 10^2 \\
 &+ (a_1 b_2 + a_2 b_1) \times 10^1 \\
 &+ a_2 b_2 \times 10^0
 \end{aligned}$$

而当我们使用数组来存储一个数的时候，这个数每一位的位数则刚好可以和数组的下标对应起来了，由此我们就可以相对简单地实现高精度乘法。

需要注意的是，我们把一个数反过来存，比如对于 $123 = 3 \times 10^0 + 2 \times 10^1 + 1 \times 10^2$ ，我们使 $a_0 = 3, a_1 = 2, a_2 = 1$ ，这样就可以刚好对应起来了。

在进行乘法时，我们分别把每一位上的数相乘，把结果加到对应的位上。同时注意当某一位上的数大于 10 之后就需要进位。

关于高精度数四则运算的详解可自行到 OI Wiki 食用：<https://oi-wiki.org/math/bignum/>

本题具体代码如下：

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <iterator>
using namespace std;
string multiply(string a, string b)
{
    string ret;
    ret.resize(a.size() + b.size()); // 调整字符串大小，防止数组越界
    for (int i = 0; i < a.size(); ++i)
        for (int j = 0; j < b.size(); ++j)
        {
            ret[i + j] += (a[i] - '0') * (b[j] - '0');

```

```

        // 把两位的乘积加到对应位上
        // 我是用的 std::string 存的这个数对应的字符串，因此需要把一位上的字符源
        if (ret[i + j] >= 10)    // 进位
        {
            ret[i + j + 1] += ret[i + j] / 10;
            ret[i + j] %= 10;
        }
    }
    int len = 0;
    for (int i = 0; i < ret.size(); ++i)
    {
        if (ret[i])
            len = i + 1;
        ret[i] += '0';
    }
    ret.resize(len); // 调整字符串的大小，去除前导零
    return ret;
}
int main()
{
    int n;
    string num1, num2;
    cin >> n;
    cin >> num1;
    reverse(num1.begin(), num1.end()); // 把一个数反过来
    while (--n)
    {
        cin >> num2;
        reverse(num2.begin(), num2.end());
        num1 = multiply(num1, num2);
    }
    copy(num1.rbegin(), num1.rend(), ostream_iterator<char>{cout}); // 输出
}

```

Day 6

这个长生不老的筒子看漫画很有特点——一本漫画必须在整数个小时内看完，而且非常专一——在单位时间内(小时)内不会看第二本书。

此外，这位筒子奢望可以享受阅读，尽可能要读得慢，于是最慢的速度是 1，最快的速度是 $\max(\text{page}[i])$ ，所有答案不过是在这两个数之间，包括这两个数的整数。

于是可以想到用搜索算法找出最小正解即可。因为时间复杂度的要求，时间要尽可能缩短，所以二分搜索是好朋友。

下面是一位 caiji 的垃圾代码(这份代码的二分搜索其实很成问题，如果 $\text{begin} > \text{end}$, return 回来的无法确定是比答案大还是小，于是处理复杂了点)。

```

#include <stdio.h>

int num = 0, leavetime = 0, page[10001] = {0};

int binary_search(int begin, int end);
int judge(int minpage);

int main() {
    scanf("%d %d", &num, &leavetime);
    int max = 0;
    for (int i = 1; i < num + 1; i++) {
        scanf("%d", &page[i]);
        if (page[i] > max) {
            max = page[i]; // 最多的书页
        }
    }
    int result = binary_search(1, max);
    // 在已知书页中寻找最佳页数, binarysearch如果能找到, 找到也不一定最小
    if (result > 0) { // 所以以防万一加个判断
        while (result > 1 && !judge(result - 1)) {
            result--;
        } // 确保找到最小的那个
        printf("%d\n", result);
    } else { // 没找到, 说明答案是不是书页中的某个数
        int trial = -result;
        int count1 = judge(trial + 1), count2 = judge(trial);
        while (trial >= 1 && !(count1 > 0 && count2 < 0)) { // >=1是防止出现1这种
            if (count2 > 0) { // 暴力搜索
                trial--;
            } else {
                trial++;
            }
            if (trial < 1) {
                break;
            }
            count1 = judge(trial + 1);
            count2 = judge(trial); // 一个大于0一个小于0说明已经夹逼到了确定的数值
        }
        printf("%d\n", trial + 1);
    }
    return 0;
}

int binary_search(int begin, int end) {
    int medium = begin + (end - begin) / 2;
    if (begin > end) {
        return -(end + 1);
    }
    int result = judge(medium);
    if (result == 0) {
        return medium;
    } else if (result < 0) {

```



```

        return binary_search(medium + 1, end);
    } else {
        return binary_search(begin, medium - 1);
    }
}

int judge(int minpage) {
    int count = 0;
    for (int i = 1; i < num + 1; i++) {
        if (page[i] % minpage == 0) {
            count = count + page[i] / minpage;
        } else {
            count = count + page[i] / minpage + 1;
        }
    }
    return leavetime - count; // 返回用minpage算出来的时间与班长返回时间的差值
}

```

Day 7

这题考验的不是筒子们的算法功底，感觉更像是数学，或者说映射
我们举个例子，就以 5 个数字为例

1 2 3 4 5 -> 1

1 2 3 5 4 -> 2 (1->2, 4 和 5 互换)

1 2 4 3 5 -> 3 (2->3, 3 和 4 互换, 3 和 5 排序)

1 2 4 5 3 -> 4 (3->4, 3 和 5 互换)

1 2 5 3 4 -> 5 (4->5, 4 和 5 互换, 3 和 4 排序)

1 2 5 4 3 -> 6 (5->6, 3 和 4 互换)

1 3 2 4 5 -> 7 (6->7, 2 和 3 互换, 2、4、5 排序)

所以能看出什么？每次+1 都伴随着一个交换和一次排序（没写排序的是因为后面其实是 1 个元素的排序）

交换是较小数下标最大且较大数的数值最小的顺序组（即按从小到大排列的两个数）

排序是交换后从原较小数的下标到最后的排序，那么逻辑就很清楚了

上代码

```

int main(){
    int n, add;
    std::cin >> n >> add;
    int a[n];
    //输入，不做解释
    for(int i=0; i<n; ++i){
        std::cin >> a[i];
    }
    int count=0;
    for(int i=n-1; i>0 && count<add; --i){ //从后往前找下标最大的较小数
        if(a[i-1]<a[i]){ //其实只要判断相邻两数即可，只要相邻的数一直是逆序整体一定逆
            int min=i;
            for(int j=i; j<n; ++j){ //找后面的最小值，没什么好说的


```

```

        if(a[j]<a[min]&&a[j]>a[i-1]){
            min=j;
        }
    }
    int temp=a[i-1];
    a[i-1]=a[min];
    a[min]=temp;
    sort(a,i,n-1);
    ++count;
    i=n;
}
}
for(int i=0;i<n-1;++i){
    std::cout<<a[i]<<" ";
}
std::cout<<a[n-1]<<std::endl;
return 0;
}

```

Day 8

（不这么讨厌的 dfs 水题 )

本讲解默认读者具有 dfs 的基础知识，还不会 dfs 的人请到 OI Wiki 自行食用：<https://oi-wiki.org/search/dfs/>

由于这题的数据规模非常小（ $n \leq 20$ ），我们可以直接考虑暴搜。我们先把每两个字符串拼接后减少的长度算出来（如果无法拼接则设为一个不可能的值，比如我设为 0），之后直接 dfs 找最大的长度就行了。

由于每个字符串最多只能用两次，我们也设置一个数组来存储使用的次数，当碰到已经用过两次的就直接跳过。同时，在 dfs 过程中也要维护这个数组的内容。

具体代码如下：

```

#include <iostream>
#include <string>
using namespace std;
int n;
string str[20];
int cat_len[20][20];
int mlen(string str1, string str2) // 计算两个字符串拼接后减少的长度，如果可以拼接
{
    int minlen = min(str1.length(), str2.length());
    for (int i = 1; i < minlen; ++i)
        if (str1.compare(str1.length() - i, i, str2, 0, i) == 0)
            // 比较第一个字符串的末尾 length - i 个字符与第二个字符串的前 length - i 个字符
            return -i;
    return 0;
}
int usecnt[20]; // 每个字符串使用的次数，dfs 过程中会更新

```

```

int dfs(int curr)
{
    ++usecnt[curr]; // 表示该字符串使用了一次
    int maxlen = 0; // 以当前字符串开头的最大长度
    for (int i = 0; i < n; ++i)
        if (usecnt[i] < 2 && cat_len[curr][i] != 0)
            maxlen = max(maxlen, dfs(i) + cat_len[curr][i]);
        // 更新最大长度
    --usecnt[curr]; // 退出一层递归的时候要把对应字符串的使用次数 - 1
    return maxlen + str[curr].length();
}

int main()
{
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> str[i];
    char first;
    cin >> first;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            cat_len[i][j] = mlen(str[i], str[j]);
    int maxlen = 0;
    for (int i = 0; i < n; ++i)
        if (str[i].front() == first)
            maxlen = max(maxlen, dfs(i)); // 计算以第 i 个字符串开头的最大长度
    cout << maxlen;
    return 0;
}

```

Day 9

难得一见的水题。

通过前序遍历可以求出整颗树，在改变一下前序遍历中左子树和右子树输出顺序再比较一不一样即可。当然这种方法需要将空结点也储存一下。

下面是一个 caiji 的代码。

```

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

typedef struct tree {
    int val;
    struct tree *left;
    struct tree *right;
} node;

int flag = 0, store1[200] = {0}, store2[200] = {0};

void create_tree(node **pnode);

```

```
void print_tree_1(node *pnode, int *index);  
void print_tree_2(node *pnode, int *index);
```

```
int main() {  
    node *head1 = NULL;  
    node *head2 = NULL;  
    int index1 = 0, index2 = 0;  
    create_tree(&head1);  
    create_tree(&head2);  
    print_tree_1(head1, &index1);  
    print_tree_2(head2, &index2);  
    int flag1 = 1;  
    for (int i = 0; i < 200; i++) {  
        if (store1[i] != store2[i]) {  
            flag1 = 0;  
            break;  
        }  
    }  
    if (flag1) {  
        printf("YES\n");  
    } else {  
        printf("NO\n");  
    }  
    return 0;  
}
```

```
void create_tree(node **pnode) {  
    int realdata = 0;  
    flag = scanf("%d", &realdata);  
    if (flag == EOF) {  
        return;  
    }  
    if (realdata == -1) {  
        return;  
    }  
    *pnode = (node *)malloc(sizeof(node));  
    (*pnode)->val = realdata;  
    (*pnode)->left = NULL;  
    (*pnode)->right = NULL;  
    create_tree(&((*pnode)->left));  
    create_tree(&((*pnode)->right));  
}
```

```
void print_tree_1(node *pnode, int *index) {// 前序遍历  
    if (!pnode) {  
        store1[*index] = -1;  
        (*index)++;  
        return;  
    }  
    store1[*index] = pnode->val;  
    (*index)++;  
    print_tree_1(pnode->left, index);  
    print_tree_1(pnode->right, index);  
}
```

```

}

void print_tree_2(node *pnode, int *index) { // 后两个反过来的前序遍历
    if (!pnode) {
        store2[*index] = -1;
        (*index)++;
        return;
    }
    store2[*index] = pnode->val;
    (*index)++;
    print_tree_2(pnode->right, index);
    print_tree_2(pnode->left, index);
}

```

Day 10

从这个题就能看出来出题人满满的恶意啊，不要使用递归方法解题，多么可恶
不用递归当然就是循环咯，找到第一个右括号，找到和它匹配的左括号，然后把里面的东西展开，重复进行直到没有右括号为止
不说废话了上代码

```

#include<iostream>
#include<string>
int main(){
    std::string str;
    std::cin>>str;
    std::string str_b="",str_m="",str_e=""; //b表示前半部分，m是要重复的部分，e是
    bool count_r;
    while(true){
        str_b="";str_m="";str_e="";
        count_r=false;
        int l=0,r,num=0;
        int i;
        for(r=0;r<str.size();++r){
            //从前往后找右括号
            if(str[r]==']'){
                count_r=true;
                //从右括号往前找左括号
                for(l=r;l>0;--l){
                    if(str[l]=='['){
                        break;
                    }
                }
                break;
            }
        }
        //没找到右括号就结束了
        if(!count_r){
            break;
        }
        //算一下重复次数

```

```

for(i=1;i<=l&&str[l-i]>='0'&&str[l-i]<='9';++i){
    int temp=1;//temp是表示位数，就是个十百千万之类的
    for(int j=1;j<i;++j){
        temp*=10;
    }
    num+=(((str[l-i])-48)*temp);
}
//偷懒用了分割字符串的函数
str_b=str.substr(0,l-i+1);
str_m=str.substr(l+1,r-l-1);
str_e=str.substr(r+1,str.size()-r-1);
std::string str4="";//临时用来存重复之后的被重复部分
for(i=0;i<num;++i){
    str4=str4+str_m;
}
str=str_b+str4+str_e;
}
std::cout<<str<<std::endl;
return 0;
}

```

Day 11

（并不讨厌的队列模拟水题 ☺）

这题就是模拟，我们按照列车出站的顺序考虑，具体的模拟逻辑如下：

1. 如果当前出站的列车恰好是进站的列车，那么就让这辆车直接过。
2. 如果当前出站的列车恰好是站内的第一辆车，就让这辆车出站。
3. 排除以上两种情况，则让当前进站的列车进站。

而当一下情况出现的时候，那么就不能实现相应的顺序：

1. 最后一辆列车已经进站，但是站内的第一辆列车并不是需要出站的车。
2. 需要某辆列车进站时，车站已满。

根据上面的逻辑，我们就可以写出相应的代码了：

```

#include <iostream>
#include <queue>
using namespace std;
int out[1000];
int main()
{
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        cin >> out[i];
    queue<int> qu; // 车站内停靠的列车，用队列表示
    int i = 0, j = 0; // 进、出站的车
}

```

```

bool sgn = true;    // 表示能否满足给定的顺序
while (i < n || !qu.empty())
{
    while (!qu.empty() && qu.front() == out[j]) // 站内第一辆车是需要出站的
    {
        qu.pop();
        ++j;
    }
    if (i < n)
    {
        if (i == out[j])    // 正要进站的车是需要出站的车
        {
            ++i;
            ++j;
        }
        else    // 不满足上述情况
            qu.push(i++);
    }
    if (qu.size() > m)    // 站内的车总数大于车站容量
    {
        sgn = false;
        break;
    }
    if (i == n && !qu.empty() && qu.front() != out[j])    // 列车已经全部进站
    {
        sgn = false;
        break;
    }
}
cout << (sgn ? "YES" : "NO");
return 0;
}

```

Day 12

这道题有很多种做法，但是很遗憾，作为小白只能使用最简单最无脑的做法—直接按层遍历。

如果不会二叉树按层遍历，google 欢迎您。

这道题别的做法有的突破点是结点是按顺序编号的，并且可以通过计算深度来判断层数。

下面是一个 caiji 的代码(按层遍历那个，c 语言选手什么都得自己实现，体谅一下，那一大堆函数除了 BFS 别的都不重要)。

```

#include <stdio.h>

typedef struct tree {
    int self;
    int left;

```

```

    int right;
} node; // 自己, 左子树, 右子树

typedef struct queue {
    node data[64];
    int front;
    int rear;
    int size;
} queue;

int maxsize = 0, real[30][15] = {0};

void BFS(node store[], queue aqueue);
void create_queue(queue *target, int max);
void enter_queue(queue *target, node thedata);
void delete_queue(queue *target, node *store);
int is_full(queue target);
int is_empty(queue target);

int main() {
    int sum = 0, count = 0, realresult[31] = {0};
    scanf("%d", &sum);
    queue aqueue;
    create_queue(&aqueue, 64);
    node store[sum + 1];
    for (int i = 1; i < sum + 1; i++) {
        store[i].self = i;
        scanf("%d %d", &store[i].left, &store[i].right);
    }
    store[0].self = 0;
    BFS(store, aqueue); // real[]里边已经存有各行的结点
    for (int i = 0; i < 30; i++) {
        int store = -1;
        for (int j = 0; j < 15; j++) {
            if (real[i][j] != 0) {
                store = real[i][j];
                realresult[count] = store;
                count++;
                break;
            }
        }
        for (int j = 14; j >= 0; j--) {
            if (real[i][j] != 0) {
                if (real[i][j] != store) {
                    realresult[count] = real[i][j];
                    count++;
                }
                break;
            }
        }
    }
} // 寻找边界结点
for (int i = 0; i < 30; i++) {
    if (!realresult[i + 1]) {

```



```

        printf("%d", realresult[i]);
        break;
    }
    printf("%d ", realresult[i]);
}
return 0;
}

void BFS(node store[], queue aqueue) { // 最基本的按层遍历
    int count1 = 0, count2 = 0;
    enter_queue(&aqueue, store[1]); // root 结点
    enter_queue(&aqueue, store[0]); // 换层标志
    while (!is_empty(aqueue)) {
        node tmp;
        delete_queue(&aqueue, &tmp); // 出队列
        if (tmp.self) { // 不是换层标志
            real[count1][count2] = tmp.self; // count1层, 第count2个
            count2++;
        } else {
            count2 = 0;
            count1++;
            if (!is_empty(aqueue)) { // 队列没空
                enter_queue(&aqueue, store[0]); // 换层标志
            }
            continue;
        }
        if (tmp.left != -1) { // 左子树不空
            enter_queue(&aqueue, store[tmp.left]);
        }
        if (tmp.right != -1) { // 右子树不空
            enter_queue(&aqueue, store[tmp.right]);
        }
    }
}

void create_queue(queue *target, int max) {
    maxsize = max;
    target->front = 0;
    target->rear = 0;
    target->size = 0;
}

void enter_queue(queue *target, node thedata) {
    if (is_full(*target)) {
        return;
    }
    target->data[target->rear] = thedata;
    target->size++;
    target->rear++;
    target->rear = target->rear % maxsize;
}

void delete_queue(queue *target, node *store) {

```

```

    if (is_empty(*target)) {
        return;
    }
    *store = target->data[target->front];
    target->front++;
    target->front = target->front % maxsize;
    target->size--;
}

int is_full(queue target) {
    if (target.size == maxsize) {
        return 1;
    }
    return 0;
}

int is_empty(queue target) {
    if (target.size == 0) {
        return 1;
    }
    return 0;
}

```

Day 13

这个题，我是真的很烦啊，爆肝了一整天，巨恶心

连续学习最长时间，其实就是某段时间内认真学习天数的 2 倍大于某段时间

那我们假设从第一天到第 i 天认真学习天数为 i_0 ，到第 j 天为 j_0 ，($i < j$) 那么只要 $2*(j_0 - i_0) < j - i$ 就说明第 $i+1$ 天到第 j 天是认真学习的

也就是 $2*j_0 - j < 2*i_0 - i$ ，那我们开一个数组里面就存 $2*i_0 - i$ 即可

但是实践发现这样效率不行，这样遍历去比较会超时，那么就用空间换时间

再开一个新的数组，下标 k 对应的内容是满足 $2*i_0 - i < k - n$ 的最小的 i ，因为最小的天数对应最长的时间

我们只需要把每一个第 i 天的值加上 n 当作下标得到一个天数，用 i 和那个天数作差就能得到结果了

然后把最大的那个结果输出就行了

上代码

```

#include<iostream>
int main(){
    int n,cnt_s=0,temp,max=0;
    scanf("%d",&n);
    int v[n+1];
    int t[2*n+1];
    v[0]=0; //这里虚构了一个第0天，因为如果从第一天开始到第i天的话需要的应该是第0天的值
    //将后半部分置为0是因为k>n时k-n>0即第0天的值，不会有比第0天更小的天数了
    for(int i=0;i<=n;++i){
        t[i]=-1;
        t[n+i]=0;
    }
}

```

```

for(int i=1;i<n+1;++i){
    std::cin>>temp;
    if(temp>8){
        ++cnt_s;
    }
    v[i]=2*cnt_s-i;
    //这里因为天数是从小到大的，正好符合我们说的那个最小的天数，没赋值的赋值，赋值了
    for(int j=2*cnt_s-i+n+1;j<=2*n&&v[j]==-1;++j){
        t[j]=i;
    }
}
for(int i=1;i<n+1;++i){
    //找到最大的差值就行了
    if(t[v[i]+n]!=-1&&i-t[v[i]+n]>max){
        max=i-t[v[i]+n];
    }
}
std::cout<<max;
return 0;
}

```

这个方法从代码上看很像 $O(n^2)$ 但实际上应该是 $O(n)$ 的 注：感谢 czg 大佬对最后一步的提醒

Day 14

（还行的拓扑排序水题 🙄）

本讲解默认读者具有图论和拓扑排序的基本知识，否则请到 [OI Wiki](#) 或《离散数学结构》自行食用。

图论基本概念：<https://oi-wiki.org/graph/concept/>、《离散数学结构》4.2、4.3

拓扑排序：<https://oi-wiki.org/graph/topo/>

知道了上面的东西这题也就自然解决了吧？我们把每个人看作一个点，把一个人排在另一个人的前面这个关系作为一条有向边，之后拓扑排序即可得到结果。

不过由于需要字典序最小，我们可以使用一个优先队列来代替拓扑排序中原本所用的队列。

具体代码如下：

```

#include <queue>
#include <iostream>
using namespace std;
const int maxn = 600000;
vector<int> nodes[maxn];    // 某个学生所连的边
int in[maxn];    // 入度
int main()
{
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {

```

```

        int f, t;
        cin >> f >> t;
        nodes[f].emplace_back(t);
        ++in[t];
    }
    // 下面在拓扑排序的过程中输出结果
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i = 0; i < n; ++i)
        if (in[i] == 0)
            pq.push(i);
    while (!pq.empty())
    {
        int n = pq.top();
        pq.pop();
        for (int t : nodes[n])
        {
            --in[t];
            if (in[t] == 0)
                pq.push(t);
        }
        cout << n << (pq.empty() ? "" : " ");
    }
    return 0;
}

```

Day 15

这是一道深搜。只要不断地切割字符串，然后判断是否有不符合条件的数字存在，不符合条件就 return，最后自然会剩下一个解，输出即可。

切割字符串的长度是 1 或者 2，如果出现有数字>m 的，或者出现重复数字的，就是不符合要求。为了节省时间，设一个 flag，只要已经找到了，一路退出，不在别的查找上浪费时间。

下面是一个 caiji 的代码。

```

#include <stdio.h>

const int maxsize = 50, maxcapacity = 100;
int state[50], result[50], flag = 0, max = 0;
// state存储数字是否已经存在, result存储当前找到的
void slice(char store[], int index, int sum);

int main() {
    char store[100] = {0};
    scanf("%s", store);
    slice(store, 0, 0);从0开始, 当前一共0个
    return 0;
}

void slice(char store[], int index, int sum) {

```

```

    if (flag) {// 已经找到
        return;
    }
    if (max == sum && (index > maxcapacity - 1 || !store[index])) {// 找到了!
        for (int i = 0; i < sum - 1; i++) {
            printf("%d ", result[i]);
        }
        printf("%d", result[sum - 1]);
        flag = 1;
        return;
    }
    int count1 = 0;
    for (int i = index; i < index + 1 && store[i]; i++) {// 切一个, 即下一个数字
        count1 = count1 * 10 + store[i] - '0';
    }
    if (count1 && count1 <= maxsize && !state[count1]) {
        int tmp = max;// 存储当前max值
        if (count1 > max) {
            max = count1;
        }
        state[count1] = 1;// 改状态
        result[sum] = count1;// 结果数组里边存储这个值
        slice(store, index + 1, sum + 1);// 深搜
        if (flag) {// 找到了!
            return;
        }
        max = tmp;// max恢复
        state[count1] = 0;// 状态改回来
        result[sum] = 0;// 结果数组扔掉这个值
    }
    int count2 = 0;
    for (int i = index; i < index + 2 && store[i]; i++) {
        count2 = count2 * 10 + store[i] - '0';
    }
    if (count2 >= 10 && count2 <= maxsize && !state[count2]) {
        int tmp = max;
        if (count2 > max) {
            max = count2;
        }
        state[count2] = 1;
        result[sum] = count2;
        slice(store, index + 2, sum + 1);
        if (flag) {
            return;
        }
        max = tmp;
        state[count2] = 0;
        result[sum] = 0;
    }
}

```

Day 16

这个题就还可以吧，就是看能保证身高体重同时上升的最大长度问题

我们可以先按照某一要素（身高或者体重）排序，然后再看另一个要素

现在它们已经按某一个要素排好序了（假设是升序），那么最前面的一些只能当顶部，也就是最前面那些最大长度只能是 1

之后看后面的人，如果他能上面能放人，那他当底部对应的最大长度就是他上面能放的人中长度最长的那个再加 1

由于前面的每一个我们都处理过了，那么每个人的最大长度就确定了，找到最大的那个就行了

上代码

```
#include<iostream>
//定义了一个类，只是为了看起来方便，其实就是存了两个int
class people{
public:
    int h=0;//身高
    int w=0;//体重
    //这里我定义了一个对两个人的比较用的小于号方法，其实就是身高体重均小于
    bool operator<(const people &a){
        return this->h<a.h&&this->w<a.wreturn;
    }
};
//排序，详见Day1,只是把排序的变量变成了people类型，比较的方式变成的身高而已
void sort(people a[],int b,int e){
    int i=b,j=e;
    int k=a[b].h;
    if(b>=e)
        return;
    while(i<j){
        while(a[i].h<k&&i<e){
            i++;
        }
        while(a[j].h>=k&&j>b){
            j--;
        }
        if(i<j){
            people temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    sort(a,b,j);
    sort(a,j+1,e);
    return;
}
int main(){
    int n,max=0;
    std::cin>>n;
    people p[n];
```

```

int length[n];
for(int i=0;i<n;++i){
    std::cin>>p[i].h>>p[i].w;
    length[i]=0;
}
sort(p,0,n-1);
for(int i=0;i<n;++i){
    for(int j=0;j<i;++j){
        if(p[j]<p[i]){//这个比较就是我在class people里写的小于号方法
            //这里就是对每一个小于他的去比较长度，取最大的那个
            length[i]=(length[j]>length[i]?length[j]:length[i]);
        }
    }
    //加上他自己
    ++length[i];
    //判断最大值
    max=(max>length[i]?max:length[i]);
}
std::cout<<max<<std::endl;
return 0;
}

```

Day 17

（有点难的二分图水题 😊）

本题解假设读者具有图论、二分图和二分猜答案的基本知识，否则请到 OI Wiki、《离散数学结构》自行食用。

图论：参见 Day 14 题解

二分图：<https://oi-wiki.org/graph/bi-graph/>

二分猜答案：<https://oi-wiki.org/basic/binary/>

这题的描述再清楚不过了，你们就不用我说该怎么写了吧？

（显然如果某个比较小的破坏值可以满足要求，那么比较大的破坏值一定可以满足，于是我们可以猜答案然后用二分图来验证

具体代码如下：

```

#include <iostream>
#include <algorithm>
using namespace std;
struct Edge
{
    int to; // 所连另一位同学的编号
    int destruction; // 这一对的破坏值
};
vector<Edge> stu[20001]; // 班级里的同学
int n, m;
int sat[20001]; // 某个同学是在第一组还是第二组
bool chk(int max_des)
{

```

fill_n(sat, 20001, -1); // 所有同学最初都未分配

```
for (int i = 1; i <= n; ++i)
```

```
    if (sat[i] == -1)
```

```
    {
```

```
        sat[i] = 0; // 未分配的可以任意分配, 这里分配到第一组
```

```
        queue<int> qu;
```

```
        qu.push(i);
```

```
        while (!qu.empty())
```

```
        {
```

```
            int t = qu.front();
```

```
            qu.pop();
```

```
            for (Edge e : stu[t])
```

```
            {
```

```
                if (e.destruction <= max_des) // 由于每个同学的边已经按破坏值从大到小排序
                    break;
```

```
                if (sat[e.to] == 1 - sat[t]) // 所连的同学已经被分配到另一组
                    continue;
```

```
                if (sat[e.to] == sat[t]) // 两者分配到了同一个组, 不满足条件
                    return false;
```

```
                sat[e.to] = 1 - sat[t]; // 分配到另一个组
```

```
                qu.push(e.to);
```

```
            }
```

```
        }
```

```
    }
```

```
    return true;
```

```
}
```

```
int main()
```

```
{
```

```
    cin >> n >> m;
```

```
    int maxd = 0, mind = INT32_MAX; // 二分的上下界
```

```
    for (int i = 0; i < m; ++i)
```

```
    {
```

```
        int x, y, p;
```

```
        cin >> x >> y >> p;
```

```
        stu[x].push_back({y, p});
```

```
        stu[y].push_back({x, p});
```

```
        maxd = max(maxd, p);
```

```
        mind = min(mind, p);
```

```
    }
```

```
    for (int i = 1; i <= n; ++i)
```

```
        sort(stu[i].begin(), stu[i].end(), [](Edge a, Edge b) { return a.d > b.d; });
```

```
        // 将一个同学所连的边按照破坏值从大到小排序
```

```
    mind = -1;
```

```
    int mid;
```

```
    for (mid = (maxd + mind) / 2; mind < maxd - 1; mid = (maxd + mind) / 2)
```

```
        // 二分猜
```

```
    {
```

```
        if (chk(mid))
```

```
            maxd = mid;
```

```
        else
```

```
            mind = mid;
```

```
    }
```

```
    cout << maxd;
```



```
return 0;  
}
```

Day 18

关于 [dijkstra 算法](#)

这题没啥好说的，唯一需要注意的是驾驶员的限制，解决办法是把一个点分成两个点，第一个点表示由接下来第一个人开，第二个点表示由第二个人开。

然后直接 dijkstra 或 spfa 就行了。两个算法其实也差别不大，无非是一个用队列另一个用单调队列。

这是一个 caiji 的代码。

```
#include <queue>  
#include <iostream>  
#include <algorithm>  
using namespace std;  
struct Edge  
{  
    int to; // 边指向的节点  
    int len; // 边的长度  
};  
struct Point  
{  
    int minlen = INT32_MAX; // 该点距离出发地的最小距离，如果为 INT32_MAX 则表示无穷大  
    vector<Edge> edges; // 该点所连的边  
};  
Point arr[2000]; // 把原编号为 n 的点拆成编号为 2n 和 2n+1 的两个点  
int n, m;  
struct PointSave // 放在队列里面的点  
{  
    int id;  
    int len;  
};  
void spfa()  
{  
    queue<PointSave> qu;  
    qu.push({0, 0});  
    qu.push({1, 0});  
    arr[0].minlen = arr[1].minlen = 0; // 显然出发点最小距离为 0  
    while (!qu.empty())  
    {  
        auto p = qu.front();  
        qu.pop();  
        if (p.len > arr[p.id].minlen)  
            continue;  
        for (Edge e : arr[p.id].edges)  
            if (p.len + e.len < arr[e.to].minlen)
```

```

        {
            arr[e.to].minlen = p.len + e.len; // spfa 的松弛操作
            qu.push({e.to, p.len + e.len});
        }
    }
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        int s, e, l, c;
        cin >> s >> e >> l >> c;
        if (c == 1) // 根据驾驶员的不同选择连接到哪个点
            arr[2 * s].edges.push_back({2 * e + 1, l});
        else
            arr[2 * s + 1].edges.push_back({2 * e, l});
    }
    spfa();
    for (int i = 0; i < n; ++i)
    {
        int len = min(arr[2 * i].minlen, arr[2 * i + 1].minlen); // 因为被
        cout << (len == INT32_MAX ? -1 : len) << ' ';
    }
    return 0;
}

```

Day 19

这个题在我看来和 Day17 的捣蛋鬼那题是一样的，czg 大佬在 Day17 的题中使用的是二分图判断答案是否正确，而我是用的染色法

这题也可以染色，我们把每支军队看成一个点，每句话都可以得到一些关系，从而确定两个点的颜色关系，将两个点连线（放到同一个数组里）

已经连上线的一定不会是假话，但是如果某一句话涉及和之前的话（已连接出来的线）无关的东西怎么办呢？没关系，新建一条线就好了

新线上的点是什么颜色不重要，只要保证相对关系即可。当两个之前毫无关系的线相连的时候，如果颜色恰好和那句话的表述相同就没问题，如果不同就把其中一条线上的点颜色同时向后推移某一个定值，由于相互关系并没有改变所以这并不影响之前的话的成立性。

如果是同一根线上的两个点相连，如果颜色没有问题就是真话，否则就是假话，忽略本次连线

理论说完了，上代码

```

#include<iostream>
#include<vector>
//又是一个为了好看弄的类，其实就是两个int
class army{
public:
    int c=-1; //0胜1胜2, 其中0,1,2表示三个国家（三种颜色）
    int index=-1; //表示在哪一根线上

```

```

};
int main(){
    int n,m,count=0;
    std::cin>>n>>m;
    army a[n+1];
    std::vector<std::vector<int>> v;
    int kind=0,a1,a2;//kind是两个军队的关系，a1，a2是军队编号
    int index=0;
    for(int i=0;i<m;++i){
        std::cin>>kind>>a1>>a2;
        //军队编号超过n肯定是不对的
        if(a1>n||a2>n){
            ++count;
            continue;
        }
        if(kind==1){//同一国家
            //两军队是同一个那么同国家肯定是对的，不用做任何处理
            if(a1==a2){
                continue;
            }
            //如果两个军队都没被染过色
            if(a[a1].c== -1&&a[a2].c== -1){
                //新建一条线
                v.push_back({});
                v[index].push_back(a1);
                v[index].push_back(a2);
                a[a1].index=index;
                a[a2].index=index;
                //给这两个军队随便染个相同的颜色，这里选择0
                a[a1].c=0;
                a[a2].c=0;
                ++index;
                continue;
            }
            //一个染色一个未染色，直接放同一条线上染同一个色就行
            if(a[a1].c== -1&&a[a2].c!= -1){
                a[a1].index=a[a2].index;
                v[a[a2].index].push_back(a1);
                a[a1].c=a[a2].c;
                continue;
            }
            if(a[a2].c== -1&&a[a1].c!= -1){
                a[a2].index=a[a1].index;
                v[a[a1].index].push_back(a2);
                a[a2].c=a[a1].c;
                continue;
            }
            //两个都染过色了
            if(a[a1].c!= -1&&a[a2].c!= -1){
                if(a[a1].c!=a[a2].c){
                    //颜色不一样看是不是同一条线，同线就是假话，否则改掉其中一条线全
                    if(a[a1].index==a[a2].index){
                        ++count;

```

```

        continue;
    }
    else{
        int temp=a[a2].c-a[a1].c;
        int idx=a[a1].index;
        int index1=a[a2].index;
        for(int j=0;j<v[idx].size();++j){
            a[v[idx][j]].index=index1;
            a[v[idx][j]].c=(a[v[idx][j]].c+temp+3)%3;
            v[index1].push_back(v[idx][j]);
        }
        v[idx].clear();
    }
}
else{
    //颜色一样同一条线不做处理，不同的线直接合并
    if(a[a1].index==a[a2].index){
        continue;
    }
    else{
        int idx=a[a1].index;
        int index1=a[a2].index;
        for(int j=0;j<v[idx].size();++j){
            a[v[idx][j]].index=index1;
            v[index1].push_back(v[idx][j]);
        }
        v[idx].clear();
    }
}
}
else{//a1战胜a2
    //我杀我自己肯定是不行的
    if(a1==a2){
        ++count;
        continue;
    }
    //其实后面的和上半部分差不多，只是相同颜色变成了胜负关系要注意对3取模
    //两个未染色
    if(a[a1].c==1&&a[a2].c==1){
        v.push_back({});
        v[index].push_back(a1);
        v[index].push_back(a2);
        a[a1].index=index;
        a[a2].index=index;
        a[a1].c=0;
        a[a2].c=1;
        ++index;
        continue;
    }
    //一个染色一个没有
    if(a[a1].c==1&&a[a2].c!=1){
        a[a1].index=a[a2].index;

```

```

        v[a[a2].index].push_back(a1);
        a[a1].c=(a[a2].c+2)%3;
        continue;
    }
    if(a[a2].c==1&&a[a1].c!=1){
        a[a2].index=a[a1].index;
        v[a[a1].index].push_back(a2);
        a[a2].c=(a[a1].c+1)%3;
        continue;
    }
    //两个都染色
    if(a[a1].c!=1&&a[a2].c!=1){
        if((a[a1].c+1)%3!=a[a2].c){
            //不符合战胜关系
            if(a[a1].index==a[a2].index){
                ++count;
                continue;
            }
            else{
                int temp=a[a1].c-a[a2].c;
                int idx=a[a2].index;
                int index1=a[a1].index;
                for(int j=0;j<v[idx].size();++j){
                    a[v[idx][j]].index=index1;
                    a[v[idx][j]].c=(a[v[idx][j]].c+temp+4)%3;
                    v[index1].push_back(v[idx][j]);
                }
                v[idx].clear();
            }
        }
        else{
            //符合战胜关系
            if(a[a1].index==a[a2].index){
                continue;
            }
            else{
                int idx=a[a1].index;
                int index1=a[a2].index;
                for(int j=0;j<v[idx].size();++j){
                    a[v[idx][j]].index=index1;
                    v[index1].push_back(v[idx][j]);
                }
                v[idx].clear();
            }
        }
    }
}

std::cout<<count<<std::endl;
return 0;
}

```

这个代码又臭又长，主要是上下部分基本上一样但是我还没有写成函数的原因。。。
~~才不会告诉你们是我懒直接 CTRL+C/V 了~~

Day 20

（可能是全场最难的树的直径水题 🤔）

我们先从一个看起来和这道题毫不相关的点开始：

在一棵树中，从一个结点出发，遍历这棵树的所有节点，最后回到这个节点，所经过的边数最少是多少？

答案是 $2(n - 1)$ 条，其中 n 是这棵树的节点个数。

我们考虑从一个父节点到一个子节点再回来，那么这个父节点与子节点之间的那条边就经过了两次；逐层递归可得从开始的那个节点（根节点）经过这个过程，树的每一条边都经过了两次。

那么这和我们的这道题有什么关系呢？

这道题中我们依然要遍历所有的节点，唯一的差别是不需要回到开始的那个节点。容易证明这样我们最多可以少经过从开始节点到结束节点的那条路径上的所有的边一次。

于是这道题就转化成了找出合适的开始和结束节点使得他们之间的距离是一棵树上最长的一条简单路径（即树的直径）。

关于树的直径的求法，有以下定理：

到树上任意一点的距离最长的一点必定是直径的一个端点。

这个定理的具体证明见这个网页：

<https://ikely.me/2014/09/21/%E6%A0%91%E7%9A%84%E7%9B%B4%E5%BE%84/>

于是我们先从任意一点 dfs 或 bfs 找出距离最大的一个点，然后从这个点重复一次，就可以找出直径的长度了。

具体代码如下：

```
#include <iostream>
using namespace std;
struct Node
{
    int edges[3] = {-1, -1, -1};    // 与一个二叉树上的一个节点所连的点最多有三个
    int parent = -1;
    int depth = 0;    // 到根节点的距离
};
Node arr[100000];
int dfs(int curr, int depth, int parent)    // 第二次 dfs
{
    int maxd = depth;
    for (int i = 0; i < 3 && arr[curr].edges[i] != -1; ++i)
    {
        if (arr[curr].edges[i] == parent)
            continue;
        maxd = max(maxd, dfs(arr[curr].edges[i], depth + 1, curr));
    }
}
```

```

    return maxd;
}
int main()
{
    int n;
    cin >> n;
    int maxd = 0, maxdi;
    for (int i = 0; i < n; ++i)
    {
        int t;
        int k = 1;
        arr[i].edges[0] = arr[i].parent;    // 添加父节点与子节点之间连的边
        for (int j = 0; j < 2; ++j)
        {
            cin >> t;
            if (t != -1)
            {
                // 我在输入数据的同时也在进行第一次遍历，大家不要学我
                arr[i].edges[k++] = t;
                arr[t].parent = i;
                arr[t].depth = arr[i].depth + 1;
                if (arr[t].depth > maxd)    // 找出到根节点距离最大的节点
                {
                    maxd = arr[t].depth;
                    maxdi = t;
                }
            }
        }
    }
    swap(arr[0].edges[0], arr[0].edges[1]); // 两次魔法操作（其实是把根节点的两
    swap(arr[0].edges[1], arr[0].edges[2]);
    cout << 2 * (n - 1) - dfs(maxdi, 0, -1);
    // 以第一次找到的节点为根节点遍历，找到距离最大的节点；
    // 2 (n - 1) - 直径长度即为最短距离
    return 0;
}

```