前言：需要大量刷题掌握的，字符串，树，链表，基础数学知识

飞机场模拟

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <cstring>

using namespace std;

const int N = 310;

int n;
int tr[N][3]; //用二维数组来存整个树
int w[N]; // 表示每个节点的权重，即客流量
int p[N]; // 用来存储初始的时候，所有登机口按照人流量从大到小排序的结果
int cnt;

void bfs()
{
    queue<int> q;
    q.push(100);
    int k = 0;

    while(!q.empty())
    {
        auto t = q.front();
        q.pop();

        if (t < 100) printf("%d %d\n", p[k ++], t);
        else //否则就是内部节点，遍历一下它的三个儿子
        {
            for (int i = 0; i < 3; i ++)
            {
                int j = tr[t][i];
                if (j != -1) q.push(j);
            }
        }
    }
}

int main()
{
    scanf("%d", &n); //输入分叉节点的数量
    memset(tr, -1, sizeof tr); // 由于所有空位都是用-1来表示的，所以先把所有儿子节点初始化为-1

    // 接下来读入n个节点的信息
    for (int i = 0; i < n; i ++)
    {
        // 先读入当前节点的编号
        int t;
        scanf("%d", &t);
        // 接下来依次读入当前节点的3个儿子
        for (int j = 0; j < 3; j ++)
            scanf("%d", &tr[t][j]);
    }

    // 在bfs之前要输入所有登机口
    int num, flow;
    while (scanf("%d%d", &num, &flow) != -1) //由于没有告诉说要输入多少项，所以这里要判断是不是读到文件结束符
    {
        w[num] = flow;
        p[cnt ++] = num;
    }

    //读完所有登机口，按照客流量从大到小的顺序排个序
    sort(p, p + cnt, [&](int a, int b){
        if (w[a] != w[b]) return w[a] > w[b];
        return a < b;
    });
    // 接下来从根节点开始bfs
    bfs();
    return 0;
}
```

八皇后

```
#include <iostream>
#include <algorithm>

using namespace std;
```

```
const int N = 20;
int n;
int b;
int q[N];
int res[100]; // 假设最多有100个解
bool col[N], dp[2 * N], udp[2 * N];
int cnt = 0;

int convert(int digits[], int size)
{
    int number = 0;
    for (int i = 0; i < size; ++i) {
        number = number * 10 + digits[i];
    }
    return number;
}

void dfs(int u)
{
    if (u == n)
    {
        res[cnt++] = convert(q, n);
        return;
    }

    for (int i = 0; i < n; i++)
    {
        if (!col[i] && !dp[u - i + n] && !udp[u + i])
        {
            q[u] = i + 1; // 列号从1开始
            col[i] = dp[u - i + n] = udp[u + i] = true;
            dfs(u + 1);
            col[i] = dp[u - i + n] = udp[u + i] = false;
        }
    }
}

int main()
{
    n = 8;
    dfs(0);
    sort(res, res + cnt); // 对所有解进行排序

    while (cin >> b)
    {
        cout << res[b - 1] << endl;
    }

    return 0;
}
```

字符串匹配

```cpp
#include <iostream
#include <algorithm>
#include <cstring>

using namespace std;

const int N = 1010;

int n;
string strs[N], p;

string filter(string str)
{
    string res;
    for (auto c : str)
        res += tolower(c);
    return res;
}

bool match(string a, str p)
{
    for (int i = 0, j = 0; i < a.size() || j < p.size(); i ++)
    {
        if (i == a.size || j == p.size()) return false;

        if (p[j] != '[')
        {
            if (a[i] != p[j]) return false;
            j ++;
        }

        else
        {
            string s;
            j ++;
            while(p[j] != ']') s += p[j ++];
            if (s.find(a[i] == -1)) return false;

        }
    }

    return true;
}

int main()
{
    cin >> n;
    for (int i = 0; i < n; i ++) cin >> strs[i];

    cin >> p;
    p = filter(p);

    for (int i = 0; i < n; i ++)
        if (match(filter(strs[i], p)))
           cout << i + 1 << ' ' << strs[i] << endl;
    return 0;
}
```

老鼠回家路

```
#include <iostream>
#include <stack>
#include <cstring>
#include <algorithm>
#include <cmath>
#include <sstream>

using namespace std;

typedef pair<int, int> PII;

int change(int a)
{
    if (a == 1) a = 2;
    else if (a == 2) a = 1;
    else if (a == 3) a = 4;
    else if (a == 4) a = 3;

    return a;
}

int main()
{
    stack<PII> route;

    string step;

    while(cin >> step && step != "0-0")
    {
        int first, second;
        sscanf(step.c_str(), "%d-%d", &first, &second);
        if (route.empty())
        {
            route.push({first, second});
            //cout <<  route.top().first << "-" << route.top().second << " ";
        }

        else
        {
            if ((abs(route.top().first - first) == 1 && route.top().first +
first != 5))
            {
                if (route.top().second < second)
                {
                    second -= route.top().second;
                    route.pop();
                    route.push({first, second});
                }
                else if (route.top().second > second)
                {
                    first = route.top().first, second = route.top().second - 
second;
                    route.pop();
                    route.push({first, second});
                }
                else if (route.top().second == second) route.pop();



            }

            else if ((abs(route.top().first - first) != 1))
            {
                route.push({first, second});
            }
        }

    }

    PII res = {0, 0};
    while(!route.empty())
    {
        int first = route.top().first;
        int second = route.top().second;

        route.pop();
        res.first = change(first);
        if (res.second == 0) res.second = second;

        if (!route.empty())
        {
            int first_next = route.top().first;
            int second_next = route.top().second;

            if (first == first_next)
            {
                res.second = second;
                res.second += second_next;
            }
            else
            {

                if (res.second != second)
                {
                    cout << res.first << "-" << res.second << " ";
                    res.second = 0;
                }
                else cout << res.first << "-" << res.second << " ";

            }
        }

        else
        {
            res.second = second;
            cout << res.first << "-" << res.second << " ";
        }
    }
}
```

迭代求立方根

```
#include <iostream>
#include <iomanip>

double cubeRoot(double x, int n) {
    double y = x;  // 初始值 y0 = x
    for (int i = 0; i < n; ++i) {
        y = y * 2.0 / 3.0 + x / (3.0 * y * y);
    }
    return y;
}

int main() {
    double x;
    int n;
    while (std::cin >> x >> n) {
        double result = cubeRoot(x, n);
        std::cout << std::fixed << std::setprecision(6) << result << std::endl;
    }
    return 0;
}
```

std::cout << std::fixed << std::setprecision(6) << result << std::endl;

这行代码用于格式化输出 `result` 变量的值，并确保输出结果保留六位小数。让我们逐步解释这行代码的各个部分：

1. `std::cout`：这是C++中标准输出流的对象，用于在控制台上输出信息。
2. `std::fixed`：这是一个流操作符，用于设置输出为固定小数位格式。默认情况下，`std::cout` 输出浮点数时可能会使用科学记数法（如 `1.23e+3` 表示 1230）。使用 `std::fixed` 可以确保数字以正常的小数形式输出，而不是科学记数法。
3. `std::setprecision(6)`：这也是一个流操作符，用于设置输出流中浮点数的精度。在这里，它将数字的精度设置为6位小数。这意味着无论原始数字有多少位小数，输出时都会截断或补零至六位小数。
4. `<< result`：这个操作符用于将 `result` 变量的值插入到输出流中。`result` 是我们计算得到的数值，也就是 `x` 的立方根的近似值。
5. `<< std::endl`：这是输出流中的另一个操作符，表示输出一个换行符并刷新输出缓冲区。这意味着在输出结果后，会换到下一行，并确保所有输出都立即显示到控制台上。

```
#include <iostream>

#include <cstdio>


```

```
using namespace std;



int main(){

    // x是被求立方根的数

    double x;

    int iteration;

    double y0;

    while(cin >> x >> iteration){

        y0 = x;

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

            y0 = 2.0/3.0*y0 + x/(3*y0*y0);

        }

        // 使用 printf 函数设置小数点后的位数

        printf("%.6f\n", y0);

    }



    return 0;

}
```

旋转矩阵

```
#include <iostream>

using namespace std;

const int N = 10;

int s[N][N], t[N][N];
int n;

int main()
{
    cin >> n;
    for (int i = 0; i < n; i ++)
        for (int j = 0; j < n; j ++)
            cin >> s[i][j];
    for (int i = 0; i < n; i ++)
        for (int j = 0; j < n; j ++)
            cin >> t[i][j];

    if (t[0][0] == s[0][0])
    {
        bool out = false;
        for (int i = 0; i < n; i ++)
        {
            for (int j = 0; j < n; j ++)
                if (t[i][j] != s[i][j])
                {
                    cout << "-1" << endl;
                    out = true;
                    break;
                }
            if (out) break;
        }

        if (!out) cout << "0" << endl;
    }

    else if (t[0][0] == s[n - 1][0])
    {
        bool out = false;
        for (int i = 0; i < n; i ++)
        {
            for (int j = 0; j < n; j ++)
                if (t[i][j] != s[n - 1 - j][i])
                {
                    cout << "-1" << endl;
                    out = true;
                    break;
                }
            if (out) break;
        }

        if (!out) cout << "90" << endl;
    }

    else if (t[0][0] == s[n - 1][n - 1])
    {
        bool out = false;
        for (int i = 0; i < n; i ++)
        {
            for (int j = 0; j < n; j ++)
                if (t[i][j] != s[n - 1 - i][n - 1 - j])
                {
                    cout << "-1" << endl;
                    out = true;
                    break;
                }
            if (out) break;
        }
        if (!out) cout << "180" << endl;
    }

    else if (t[0][0] == s[0][n - 1])
    {
        bool out = false;
        for (int i = 0; i < n; i ++)
        {
            for (int j = 0; j < n; j ++)
                if (t[i][j] != s[j][n - 1 - i])
                {
                    cout << "-1" << endl;
                    out = true;
                    break;
                }
            if (out) break;
        }

        if (!out) cout << "270" << endl;
    }

    else cout << "-1" << endl;
}
```

y总的代码（用两次对称来做：对角线，横纵轴）

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <vector> // 比较两个矩阵是否相等的话可以用vector来比较，因为vector自带一个比较函数

using namespace std;

int n;
typedef vector<vector<int>> VVI;
VVI a, b;

void rotate(VVI& c)
{
    //两次对称实现旋转，先沿着对角线，再是横轴（因为对于b要逆时针旋转回去）
    for (int i = 0; i < n; i ++)
        for (int j = 0; j < i; j ++)
            swap(c[i][j], c[j][i]);

    for (int j = 0; j < n; j ++)
        for (int i = 0, k = n - 1; i < k; i ++, k --)
            swap(c[i][j], c[k][j]);
}

int main()
{
    cin >> n;
    a = b = VVI(n, vector<int>(n));
    for (int i = 0; i < n; i ++)
        for (int j = 0; j < n; j ++)
            cin >> a[i][j];

    for (int i = 0; i < n; i ++)
        for (int j = 0; j < n; j ++)
             cin >> b[i][j];

    for (int i = 0; i < 4; i ++)
    {
        if (a == b)
        {
            cout << i * 90 << endl;
            return 0;
        }
        rotate(b);
    }

    cout << "-1" << endl;

    return 0;


}
```

三叉树

```
#include <iostream>
#include <vector>
#include <map>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 1000; // 假设节点数最大为1000
int tr[N][3]; // 存储树结构
map<int, vector<int>> paths; // 存储从根到叶子节点的路径

// DFS记录路径
void dfs(int node, vector<int> &path) {
    path.push_back(node);
    bool isLeaf = true;
    for (int i = 0; i < 3; i++) {
        if (tr[node][i] != -1) {
            isLeaf = false;
            dfs(tr[node][i], path);
        }
    }
    if (isLeaf) {
        paths[node] = path;
    }
    path.pop_back();
```

```
}

// 找到两个路径的最后一个公共节点
int findLastCommon(const vector<int> &path1, const vector<int> &path2) {
    int minLength = min(path1.size(), path2.size());
    int lastCommon = -1; // 用于标记最后一个公共节点的位置
    for (int i = 0; i < minLength; i++) {
        if (path1[i] != path2[i]) break;
        lastCommon = i;
    }
    return lastCommon;
}

int main() {
    memset(tr, -1, sizeof tr); // 初始化所有子节点为 -1

    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int t;
        cin >> t;
        for (int j = 0; j < 3; j++) {
            cin >> tr[t][j];
        }
    }

    vector<int> path;
    dfs(100, path); // 假设100是根节点

    int m;
    cin >> m;
    vector<pair<int, int>> leaves(m);
    for (int i = 0; i < m; i++) {
        cin >> leaves[i].first >> leaves[i].second;
    }
    sort(leaves.begin(), leaves.end(), [](pair<int, int> &a, pair<int, int> &b)
{
        return a.second < b.second; // 按优先级排序
    });

    int current = 100; // 从根节点开始
    for (auto leaf : leaves) {
        int nextLeaf = leaf.first;
        vector<int> &path1 = paths[current];
        vector<int> &path2 = paths[nextLeaf];
        int lastCommon = findLastCommon(path1, path2);

        // 输出从当前节点到nextLeaf的路径
        for (int i = path1.size() - 1; i > lastCommon; i--) {
            cout << path1[i - 1] << " ";
        }
        for (size_t i = lastCommon + 1; i < path2.size(); i++) {
            cout << path2[i] << " ";
        }
        cout << endl;
        current = nextLeaf;
    }

    // 最后从最后一个叶子节点回到根节点
    vector<int> &pathToRoot = paths[current];
    for (int i = pathToRoot.size() - 1; i > 0; i--) {
        cout << pathToRoot[i - 1] << " ";
    }

    return 0;
}
```

多叉树全路径

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

const int N = 200;
int tr[N][3]; // 存储树的结构
unordered_map<int, vector<int>> paths; // 存储从根节点到各个叶子节点的路径
vector<int> path; // 临时存储路径

// DFS函数遍历树并存储路径
void dfs(int node) {
    path.push_back(node);
    if (node < 100) { // 假设叶子节点编号小于100
        paths[node] = path;
    } else {
        for (int i = 0; i < 3; i++) {
            if (tr[node][i] != -1) {
                dfs(tr[node][i]);
            }
        }
    }
    path.pop_back();
}

// 主函数
int main() {
    memset(tr, -1, sizeof tr); // 初始化树的结构
    int n;
    cin >> n; // 读取节点数
    for (int i = 0; i < n; i++) {
        int t;
        cin >> t; // 读取节点编号
        for (int j = 0; j < 3; j++) {
            cin >> tr[t][j]; // 读取子节点
        }
    }

    dfs(100); // 假设根节点为100，开始DFS

    int m;
    cin >> m; // 读取查询数
    vector<pair<int, int>> queries(m);
    for (int i = 0; i < m; i++) {
        cin >> queries[i].first >> queries[i].second; // 读取叶子节点和优先级
    }

    // 根据优先级排序查询
    sort(queries.begin(), queries.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
        return a.second < b.second;
    });

    // 直接打印从根节点到叶子节点的路径
    for (const auto& query : queries) {
        const vector<int>& current_path = paths[query.first];
        for (int i = 0; i < current_path.size(); ++i) {
            if (i > 0) cout << " "; // 添加空格分隔节点
            cout << current_path[i];
        }
        cout << endl; // 每条路径打印后换行
    }

    return 0;
}
```

lambda表达式的使用

在飞机场和三叉树中这种树的权值排序问题当中，lambda表达式是蛮常用的，下面结合几个例子讲讲使用的注意事项：

```
#### 首先看看飞机场这道题
//读完所有登机口，按照客流量从大到小的顺序排个序
    sort(p, p + cnt, [&](int a, int b){
        if (w[a] != w[b]) return w[a] > w[b];
        return a < b;
    });

#### 再来康康三叉树这道题
    sort(leaves.begin(), leaves.end(), [](pair<int, int> &a, pair<int, int> &b) {
            return a.second < b.second; // 按优先级排序
        });
```

1. 为什么一个有引用？另一个没有？

   [capture]是捕获符，因为引用是为了让Lambda表达式能够用到外部变量

2. [&]表示所有外部变量可用，[&y]表示只能用外部变量y

pair<int, int> &a, pair<int, int> &b 这里用引用是为了避免拷贝，对于飞机场那里，int是基本类型，拷贝成本低可以不用引用传递的方式

3. vector &path1 = paths[current]; 这里为什么不直接定义vector path1 = paths[current];

   使用引用`&`来定义`path1`而不是直接定义新的`vector<int> path1`有几个原因：

1. **避免拷贝**

`vector<int> &path1 = paths[current];`定义了一个对`paths[current]`的引用。这意味着`path1`和`paths[current]`指向同一块内存，因此任何对`path1`的修改都会直接反映到`paths[current]`中。

如果使用`vector<int> path1 = paths[current];`，则会创建一个`paths[current]`的副本。这不仅占用额外的内存，而且如果需要修改`path1`中的数据，这些修改不会影响到原始的`paths[current]`。而使用引用，可以避免这种不必要的拷贝，节省内存和提高性能。

2. **保持数据的一致性**

使用引用时，`path1`和`paths[current]`是同一对象的别名，因此对`path1`的任何修改都是对`paths[current]`的直接修改。这确保了数据的一致性，尤其是在需要修改原始数据的场景中。例如，如果需要在`path1`中插入、删除或修改元素，这些操作会立即反映到`paths[current]`中。

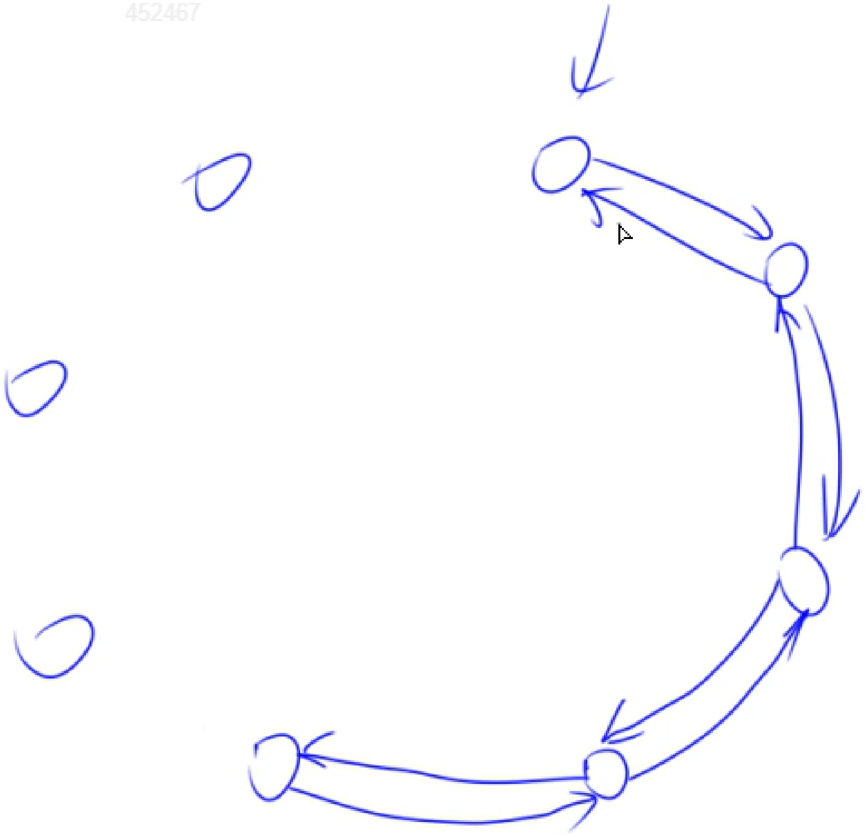
空闲块

自己的思路：

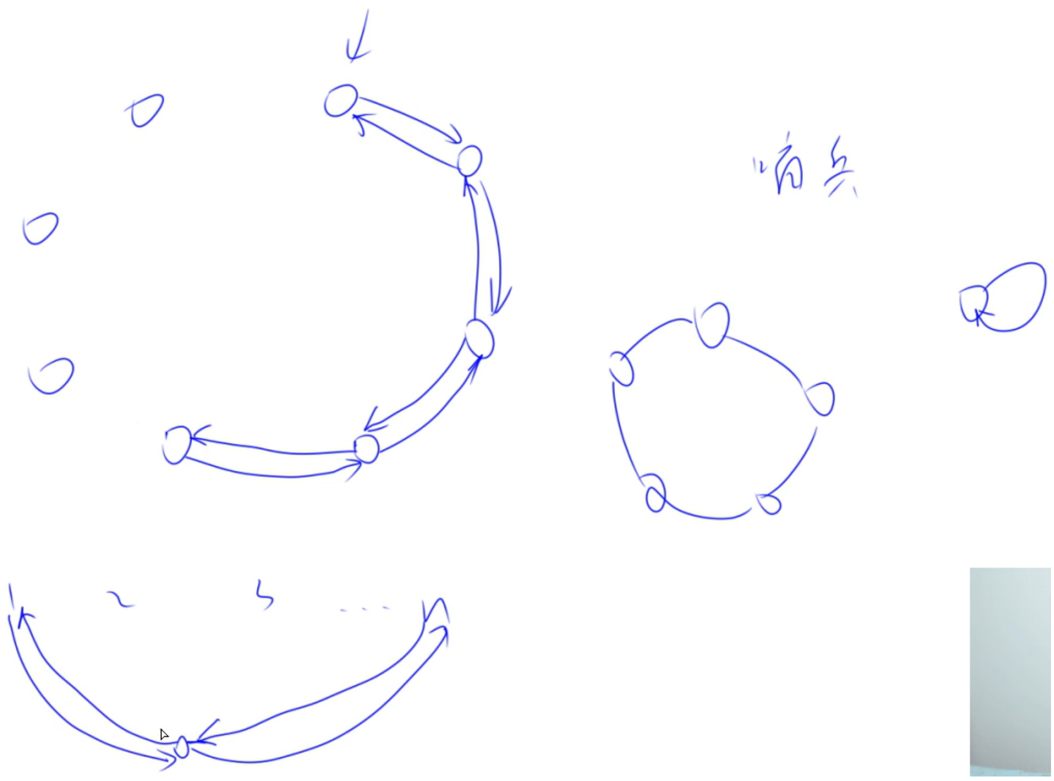
```
#include <iostream>
#include <vector>
#include <climits> // 包含INT_MAX定义
#include <algorithm>

using namespace std;

struct Block
{
    int start;   // 起始位置
    int length;  // 长度
    int next;    // 下一个节点的索引
};

int main()
{
    int N;
    cin >> N;

    // 创建一个数组来存储所有的空闲块
    vector<Block> blocks(N);

    // 读取每个空闲块的信息
    for (int i = 0; i < N; ++i)
    {
        cin >> blocks[i].start >> blocks[i].length;
        blocks[i].next = (i + 1) % N;  // 下一个节点的索引，最后一个指向第一个
    }

    int current = 0; // 当前节点索引

    int req;
    while (cin >> req && req != -1)
    {
        int min_index = -1;
        int min_length = INT_MAX;
        int start_index = current;
        bool found = false;

        do
        {
            if (blocks[current].length >= req && blocks[current].length <
min_length)
            {
                min_length = blocks[current].length;
                min_index = current;
                found = true;
            }
            current = blocks[current].next;
        } while (current != start_index);

        if (found)
        {
            if (blocks[min_index].length == req)
            {
                // 完全匹配，将其从链表中移除
                int prev = min_index;
                while (blocks[prev].next != min_index)
                {
                    prev = blocks[prev].next;
                }
                blocks[prev].next = blocks[min_index].next;
                current = blocks[min_index].next;
                blocks[min_index].length = 0; // 标记为空
            }
            else
            {
                // 部分匹配，减小块的大小
                blocks[min_index].length -= req;
                current = min_index;
            }
        }
    }

    // 输出剩余的空闲块信息
    int start_index = current; // 重新声明start_index
    do
    {
        if (blocks[current].length > 0)
        {
            cout << blocks[current].start << " " << blocks[current].length  <<
endl;
```

```
        }
        current = blocks[current].next;
    } while (current != start_index);

    return 0;
}
```

y总的思路：



为了便于删除操作，用一个双向链表来做



新增哨兵（都初始化为0，永远不可能被删掉），判断末尾节点为空的情况

```
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 110;

int n;
int l[N], r[N], pos[N], len[N];

void remove(int p)
{
    l[r[p]] = l[p];
    r[l[p]] = r[p];
}

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++)
    {
        scanf("%d%d", &pos[i], &len[i]);
        l[i] = i - 1, r[i] = i + 1;
    }

    l[0] = n, r[0] = 1, r[n] = 0;


    int cur = 1;
    while(ture)
    {
        int x;
        scanf("%d", &x);
        if (x == -1) break;

        int p = -1; //最优块下标
        for (int i = cur; ;i = r[i])
        {
            if (len[i] >= x && (p == -1 || len [i] < len[p]))
            {
                p = i;
            }
            if (r[i] = cur) break;
        }

        if (p != -1)
        {
            if (len[p] == x)
            {
                cur = r[p];
                remove(p);
            }

            else
            {
                len[p] -= x;
                cur = p;
            }
        }
    }

    for (int i = cur;  ; i = r[i])
    {
        if (i) printf("%d %d\n", pos[i], len[i]); //0是哨兵跳过
        if (r[i] == cur) break;
    }
}
```

阶乘和

用for循环

```
#include <iostream>
#include <algorithm>

using namespace std;
typedef long long LL;

const int N = 20;
LL fact[N];

int main()
{
    int n;
    cin >> n;
    fact[0] = 1;
    int res = 0;
    for (int i = 1; i <= n; i ++)
    {
        fact[i] = fact[i - 1] * i;
        res += fact[i];
    }

    cout << res << endl;

    return 0;

}
```

用递归

```
#include <iostream>
#include <algorithm>

using namespace std;
typedef long long LL;

const int N = 20;
LL res;

LL sum(int n)
{
    if (n == 1) return 1;
    LL res = sum(n - 1) + n * n;

    return res;
}

int main()
{
    int n;
    cin >> n;

    cout << sum(n) << endl;
    return 0;

}
```

```
#include <iostream>
#include <algorithm>

using namespace std;
typedef long long LL;


// 计算阶乘的递归函数
LL factorial(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

// 计算前n项阶乘和的递归函数
LL sumFactorials(int n)
{
    if (n == 0)
        return 0;
    else
        return factorial(n) + sumFactorials(n - 1);
}

int main()
{
    int n;
    cin >> n;
    cout << sumFactorials(n) << endl;
    return 0;

}
```

手机基站

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

struct Log {
    string phone;
    char baseStation;
    string loginTime;
    string logoutTime;
};

bool overlap(const Log& a, const Log& b) {
    return !(a.logoutTime <= b.loginTime || b.logoutTime <= a.loginTime);
}

bool compare(const Log& a, const Log& b) {
    if (a.loginTime == b.loginTime)
        return a.phone < b.phone;
    return a.loginTime < b.loginTime;
}

int main() {
    int N;
    cin >> N;
    vector<Log> logs(N);
    for (int i = 0; i < N; ++i) {
        cin >> logs[i].phone >> logs[i].baseStation >> logs[i].loginTime >> logs[i].logoutTime;
    }

    string targetPhone;
    cin >> targetPhone;

    vector<Log> targetLogs;
    vector<Log> resultLogs;

    for (const auto& log : logs) {
        if (log.phone == targetPhone) {
            targetLogs.push_back(log);
        }
    }

    for (const auto& log : logs) {
        if (log.phone != targetPhone) {
            for (const auto& targetLog : targetLogs) {
                if (log.baseStation == targetLog.baseStation && overlap(log, targetLog)) {
                    resultLogs.push_back(log);
                    break;
                }
            }
        }
    }

    sort(resultLogs.begin(), resultLogs.end(), compare);

    for (const auto& log : resultLogs) {
        cout << log.phone << " " << log.baseStation << " " << log.loginTime << " " << log.logoutTime << endl;
    }

    return 0;
}
```

最简真分数

```
#include <iostream>

using namespace std;

const int N = 610;

int a[N];
int cnt;

// 辅助函数：判断两个数是否互质
bool is_primes(int x, int y)
```

```
{
    // 使用辗转相除法（欧几里得算法）计算最大公约数
    while (y != 0)
    {
        int temp = y;
        y = x % y;
        x = temp;
    }
    // 如果最大公约数等于1，则互质
    return x == 1;
}

int main()
{
    int n;

    while (cin >> n)
    {
        cnt = 0;  // 初始化计数器
        for (int i = 0; i < n; i++) cin >> a[i];

        for (int i = 0; i < n; i++)
            for (int j = 0; j < i; j++)
                if (i != j && a[i] > a[j] && is_primes(a[i], a[j])) cnt++;

        cout << cnt << endl;
    }

    return 0;
}
```

改进版本（主要是没有重复计算的部分了）

```
#include <iostream>

using namespace std;

const int N = 610;

int a[N];
int cnt;

// 辅助函数：判断两个数是否互质
bool is_primes(int x, int y)
{
    int u = (x < y)? x : y;
    int v = (x > y)? x : y;
    // 使用辗转相除法（欧几里得算法）计算最大公约数
    while (u != 0)
    {
        int temp = u;
        u = v % u;
        v = temp;
    }
    // 如果最大公约数等于1，则互质
    return v == 1;
}

int main()
{
    int n;

    while (cin >> n)
    {
        cnt = 0;  // 初始化计数器
        for (int i = 0; i < n; i++) cin >> a[i];

        for (int i = n - 1; i >= 0; i--)
            for (int j = 0; j < i; j++)
                if (a[i] != a[j] && is_primes(a[i], a[j])) cnt++;

        cout << cnt << endl;
    }

    return 0;
}
```

等差数列

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool is_prime(int n)
{
    if (n < 2) return false;
    for (int i = 2; i <= n / i; i ++)
        if (n % i == 0) return false;

    return true;
}

vector<int> prime;
vector<int> res;
bool used; //用来标记当前prime[i - 1]是否出现在上一组等差数列中


int main()
{
    int a, b;
    cin >> a >> b;

    for (int i = a; i <= b; i ++)
    {
        if (is_prime(i)) prime.push_back(i);
    }

    for (int i = 1; i + 1< prime.size(); i ++)
    {
        if (!used)
        {
            res.push_back(prime[i - 1]);
            res.push_back(prime[i]);
        }
        else
        {
            i ++;
            res.push_back(prime[i - 1]);
            res.push_back(prime[i]);
            used = false;
        }


        int u = prime[i] - prime[i - 1];
        while(i + 1 < prime.size() && prime[i + 1] - prime[i] == u)
        {
            res.push_back(prime[i + 1]);
            i ++;
        }
        if (res.size() >= 3)
        {
            used = true;
            for (auto r: res) cout << r << " ";
            puts("");
        }

        res.clear();

    }

    return 0;
}
```

字符串距离

这个题的思路很容易想，难点在于用什么存以及如何排序

```
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>

using namespace std;

vector<string> str;

struct Res {
    string small;
    string big;
    int distance;
};

int cnt_distance(string a, string b)
{
    int cnt = 0;
    for (int i = 0; i < a.size(); i ++)
    {
        if (a[i] != b[i]) cnt ++;
    }

    return cnt;
}

bool compare(const Res& a, const Res& b)
{
    if (a.distance == b.distance)
    {
        if (a.small == b.small)
            return a.big < b.big;
        return a.small < b.small;
    }

    return a.distance < b.distance;
}

int main()
{
    int N;
    cin >> N;

    vector<Res> results; //用来存最后要输出的六条数据

    string a;
    for (int i = 0; i < N; i ++) cin >> a, str.push_back(a);

    for (int i = 0; i + 1 < str.size(); i ++)
        for (int j = i + 1; j < str.size(); j ++)
        {
            Res tmp;
            int dis = cnt_distance(str[i], str[j]);

            if (str[i] < str[j])
            {
                tmp.small = str[i];
                tmp.big = str[j];
            }
            else
            {
                tmp.small = str[j];
                tmp.big = str[i];
            }
            tmp.distance = dis;
            results.push_back(tmp);
        }

    sort(results.begin(), results.end(), compare);

    for (int i = 0; i < 6 && i < results.size(); i ++)
        cout << results[i].small << " " << results[i].big << " " <<
results[i].distance << endl;

    return 0;
}
```

模拟编译

中缀表达式转后缀表达式（板子题）

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
#include <sstream>
#include <map>

using namespace std;

// 判断运算符的优先级
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// 进行简单的算术运算
int applyOp(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
    return 0;
}

// 将中缀表达式转换为后缀表达式
string infixToPostfix(const string &infix) {
    stack<char> operators;
    string postfix;
    for (char ch : infix) {
        if (isspace(ch)) continue;  // 跳过空格
        if (isdigit(ch)) {
            postfix += ch;
        } else if (ch == '(') {
            operators.push(ch);
        } else if (ch == ')') {
            while (!operators.empty() && operators.top() != '(') {
                postfix += operators.top();
                operators.pop();
            }
            operators.pop();  // 弹出左括号
        } else {
            while (!operators.empty() && precedence(operators.top()) >= precedence(ch)) {
                postfix += operators.top();
                operators.pop();
            }
            operators.push(ch);
        }
    }
    while (!operators.empty()) {
        postfix += operators.top();
        operators.pop();
    }
    return postfix;
}

// 计算后缀表达式
int evaluatePostfix(const string &postfix) {
    stack<int> values;
    for (char ch : postfix) {
        if (isdigit(ch)) {
            values.push(ch - '0');
        } else {
            int b = values.top(); values.pop();
            int a = values.top(); values.pop();
            values.push(applyOp(a, b, ch));
        }
    }
    return values.top();
}

// 计算中缀表达式
int evaluateInfix(const string &infix) {
    string postfix = infixToPostfix(infix);
    return evaluatePostfix(postfix);
}

int main() {
    string infix = "3 + 5 * (2 - 8)";
    cout << "Infix: " << infix << endl;
    int result = evaluateInfix(infix);
    cout << "Result: " << result << endl;
    return 0;
}
```

模拟编译题目

```
#include <iostream>
```

```
#include <string>
#include <sstream>
#include <map>
#include <iomanip>
#include <stack>
#include <cctype>

using namespace std;

map<char, double> variables;

double evaluateExpression(string expr);
// 读取变量的值
void readCommand() {
    string line;
    getline(cin, line);  // 读取一行输入
    stringstream ss(line);
    double value;
    for (auto& pair : variables) {
        ss >> value;
        variables[pair.first] = value;
    }
}

// 赋值命令处理函数
void assignCommand(string line) {
    char var = line[0];
    string expr = line.substr(2);
    double value = evaluateExpression(expr);
    variables[var] = value;
}

// 打印变量的值
void printCommand(string line) {
    stringstream ss(line);
    char var;
    bool first = true;
    while (ss >> var) {
        if (!first) cout << " ";
        cout << fixed << setprecision(2) << variables[var];
        first = false;
    }
    cout << endl;
}

// 计算表达式的值
double evaluateExpression(string expr) {
    // 将表达式转换为后缀表达式，然后求值
    stack<char> ops; // 操作符栈
    stack<double> vals; // 操作数栈
    stringstream ss;
    ss << '(' << expr << ')'; // 将表达式两端加上括号，方便处理

    while (!ss.eof()) {
        char ch = ss.peek();
        if (isspace(ch)) {
            ss.get(); // 忽略空格
            continue;
        }
        if (isdigit(ch) || ch == '.') {
            double num;
            ss >> num;
            vals.push(num); // 将数字压入操作数栈
        } else if (isalpha(ch)) {
            ss.get();
            vals.push(variables[ch]); // 将变量值压入操作数栈
        } else if (ch == '(') {
            ss.get();
            ops.push('('); // 将左括号压入操作符栈
        } else if (ch == ')') {
            ss.get();
            // 遇到右括号，处理括号内的所有操作符
            while (ops.top() != '(') {
                char op = ops.top(); ops.pop();
                double b = vals.top(); vals.pop();
                double a = vals.top(); vals.pop();
                if (op == '+') vals.push(a + b);
                else if (op == '-') vals.push(a - b);
                else if (op == '*') vals.push(a * b);
                else if (op == '/') vals.push(a / b);
            }
            ops.pop(); // 弹出左括号
        } else {
            ss.get();
            // 遇到运算符，处理优先级
            while (!ops.empty() && ops.top() != '(' &&
                   (ops.top() == '*' || ops.top() == '/' || ops.top() == '+' ||
ops.top() == '-')) {
                char op = ops.top(); ops.pop();
                double b = vals.top(); vals.pop();
                double a = vals.top(); vals.pop();
                if (op == '+') vals.push(a + b);
                else if (op == '-') vals.push(a - b);
                else if (op == '*') vals.push(a * b);
                else if (op == '/') vals.push(a / b);
            }
            ops.push(ch); // 将当前操作符压入栈
        }
    }
    return vals.top(); // 返回最终结果
}

int main() {
    string line;
    while (getline(cin, line)) {
        if (line.find("read") == 0) {
            string vars = line.substr(5);
            stringstream ss(vars);
            char var;
            while (ss >> var) {
                variables[var] = 0;
            }
            readCommand();
        } else if (line.find("print") == 0) {
            printCommand(line.substr(6));
        } else if (line.find("exit") == 0) {
            break;
        } else {
            assignCommand(line);
        }
    }
    return 0;
}
```

这个题的输入输出部分属于进阶版的字符串操作