- 无优先级运算
  - 最多需要多少个数字
  - 输出所有结果

贪心算法

- 神奇的口袋
- 搬桌子*
- 堆石头*
- 活动安排*
- 马踏棋盘*
  - 深度优先搜索
  - 递归解法+深度优先搜索
  - 递归深度优先搜索+贪心解法

图

- 哈密顿图
- 简单回路
- 并查集
  - 求能够连通需要的边数
  - 求是否为无向连通图
  - 求是否为有向树
- 求有向图强连通图
- 连通图着色*
- 广义表->二叉链表*
- 最小生成树
  - 未修好路
  - 修部分路
- 最短路径
  - Dijkstra
    - 单纯最短路径
    - 最短路径下的最少花费
  - Floyd多点最短路径
  - TSP单源最短路径*
  - 中国邮递员问题
  - 选址问题
- 拓扑排序*
  - 判断能否产生拓扑序列
  - 产生拓扑序列
- 关键路径
  - 最早开始时间与最晚开始时间的计算

搜索

- 八皇后问题
- 迷宫问题（广度优先搜索）
- ROAD
- 八数码*

动态规划

- 凑硬币问题
- 0-1背包问题
- 完全背包问题
- 多重背包
- 最长上升子序列
- 最长公共子序列
- 最大连续子序列和
  - 直接找最大

C++文件操作

简单C++文件操作

```cpp
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
#include <fstream>
using namespace std;
int main()
{
    //const char* ifile ="/Users/yangshucheng/Desktop/C++/fushi/fushi/in.txt";
    const char* ofile ="/Users/yangshucheng/Desktop/C++/fushi/fushi/out.txt";
    char name[100];
    string age;
    ofstream outfile;
    outfile.open(ofile);
    cout << "Write" << endl;
    cout << "name:";
    cin.getline(name, 100);
    outfile << name <<endl;
    cout << "age:";
    cin >> age;
    cin.ignore();
    outfile << age <<endl;
    outfile.close();
    ifstream infile;
    infile.open(ofile);
    cout << "Reading from the file" << endl;
    infile >> name;
    cout << name << endl;
    infile >> age;
    cout << age << endl;
    infile.close();
    return 0;
}
```

```
Write
name:zara
age:12
Reading from the file
zara
```

约瑟夫环+文件处理

```cpp
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
#include <fstream>
#include <queue>
using namespace std;
int arr[100][3];
int main()
{
    const char* ifile ="/Users/yangshucheng/Desktop/C++/fushi/fushi/in.txt";
    const char* ofile ="/Users/yangshucheng/Desktop/C++/fushi/fushi/out.txt";
    ifstream in(ifile);
    ofstream out(ofile);
    int length = 0;
    for (int i = 0; i < 100; ++i) {
        for (int j = 0; j < 3; ++j) {
            in >> arr[i][j];
        }
        if(arr[i][0] == 0 && arr[i][0] == 0 && arr[i][0] == 0){
            length = i + 1;
            break;
        }
    }
    int m,n,p;
    for (int i = 0; i < length; i++) {
        n = arr[i][0];
        p = arr[i][1];
        m = arr[i][2];
        if (m == 0 && n == 0 && p == 0) {
            break;
        }
        queue<int> children;
        for (int i = 1; i <= n; ++i) {//将1，2，3...n入队
            children.push(i);
        }
        for (int i = 1; i < p; ++i) {//for循环线判断中间的判断式子是否为真再决定是否继续执行
            children.push(children.front());//从编号p开始计算，需要编号p放到队头
            children.pop();
        }
        while (!children.empty()) {
            for (int i = 1; i < m; ++i) {//m-1个小孩依次重新入队
                children.push(children.front());
                children.pop();
            }
            int x = children.front();
            if (children.size() == 1) {//最后一个小孩的输出不同
                out << x <<endl;
                //printf("%d\n",x);
            } else{
                out << x <<",";
            }
                //printf("%d,",x);}
            children.pop();
        }
    }
```

```
    return 0;

}

```

经典基础

反序数

```
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
int converseNum(int num){
    int a = 0;
    while (num!=0) {
        a *= 10;
        a += num % 10;
        num /= 10;
    }
    return a;
}
int main(){
    int x;
    for (int n = 1; n<=9999; n++) {
        x = converseNum(n);
        if(x == 9 * n){
         printf("%d\n",n);
    }
    }
    return 0;
}
```

对称平方数

```
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
int converseNum(int num){
    int a = 0;
    while (num!=0) {
        a *= 10;
        a += num % 10;
        num /= 10;
    }
    return a;
}
int main(){
    int x,y;
    for (int n = 0; n<=256; n++) {
        x = n * n;
        y = converseNum(x);
        if(x == y){
```

```
        printf("%d\n",n);
    }
    }
    return 0;
}
```

叠框

```
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
char G[100][100];
void change(char* x,char* y){
    char* tmp = x;
    x = y;
    y = tmp;
}
int main(){
    int n;
    char a,b;
    bool flag = true;
    char c;//当前填充的字符
    int len;//当前填充的长度
    while (scanf("%d %c %c",&n,&a,&b)!=EOF) {
        for (int i = 0; i < 99; ++i) {
            for (int j = 0; j < 99; ++j) {
                G[i][j] = ' ';
            }
        }
        if (flag) {
            flag = false;
        } else{
            printf("\n");
        }
        for (int i = 0; i <= (n/2); i++) {
            int j = n - i - 1;
                if ((n/2 - i) % 2 == 0 ) {
                    c = a;
                } else{
                    c = b;
                }
                len = n - 2*i;
                for (int k = 0; k < len; k++) {
                    G[i][i + k] = c;//上方
                    G[i + k][i] = c;//左方
                    G[j][j - k] = c;//下方
                    G[j - k][j] = c;//右方
                }


        }
        G[0][0] = G[0][n-1] = G[n-1][0] = G[n-1][n-1] = ' ';
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                printf("%c",G[i][j]);
            }
            printf("\n");
        }

```

```
    }
    return 0;
}

```

日期累加

```
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
int daytab[2][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};
bool isLeapYear(int year){
    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
};
int NumOfYear(int year){
    if (isLeapYear(year)) {
        return 366;
    } else{
        return 365;
    }
};
int main(){
    int caseNum;
    int year,month,day,dayNum;
    scanf("%d",&caseNum);
    while (caseNum--) {
        scanf("%d %d %d %d",&year,&month,&day,&dayNum);
        int row = isLeapYear(year);
        for (int i = 0; i < month ; i++) {
            dayNum += daytab[row][i];
        }
        dayNum += day;
        while (dayNum > NumOfYear(year)) {
            dayNum -= NumOfYear(year);
            year++;
        }
        int j;
        row = isLeapYear(year);
        for (j = 0; dayNum > daytab[row][j]; ++j) {
            dayNum -= daytab[row][j];
        }
        printf("%04d-%02d-%02d\n",year,j,dayNum);

    }
    return 0;
}

```

排序

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
```

```cpp
using namespace std;
int a[100];
int main(){
    int num;
    while(scanf("%d",&num)!=EOF){
        for (int i = 0; i < num; i++) {
            scanf("%d",&a[i]);
        }
        sort(a, a + num);
        for (int i = 0; i < num; i++) {
            printf("%d ",a[i]);
        }
        printf("\n");

    }
        return 0;
}

```

- 快速排序

```cpp
#include <stdio.h>
#include <iostream>
#include <algorithm>
using namespace std;
int a[100];
void QuickSort(int a[],int low,int high){
    if(low<high){
        int  i =low,j = high;
        int temp;
        temp = a[i];
        while(i<j){
            while(i<j&&a[j]>=temp) j--;
            if(i<j){
                a[i] = a[j];
                i++;
            }
            while(i<j&&a[i]<=temp) i++;
            if(i<j){
                a[j] = a[i];
                j--;
            }
        }
        a[i] = temp;
        QuickSort(a,low,i-1);
        QuickSort(a,i+1,high);
    }
}
int main(){
    int num;
    while(scanf("%d",&num)!=EOF){
        for (int i = 0; i < num; i++) {
            scanf("%d",&a[i]);
        }
        QuickSort(a, 0, num - 1);
        for (int i = 0; i < num; i++) {
            printf("%d ",a[i]);
        }
        printf("\n");
    }
```

```
    return 0;
}
```

- 快速排序，mid

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
using namespace std;
int a[110];
void quicksort(int a[], int left, int right) {
    if (left >= right) {
        return;
    }//递归结束条件
    int i = left, j = right;
    int mid = left + (right - left) / 2;
    swap(a[left], a[mid]);
    while (i != j) {
        while (i < j && a[i] < a[left]) i++;
        while (i < j && a[j] >= a[left]) j--;
        swap(a[i], a[j]);
    }
    mid = j;
    swap(a[left], a[mid]);
    quicksort(a, left, mid - 1);//i左边递归
    quicksort(a, mid + 1, right);//j右边递归
}
int main() {
    int num;
    while (scanf("%d", &num) != EOF) {
        for (int i = 0; i < num; i++) {
            scanf("%d", &a[i]);
        }
        quicksort(a, 0, num - 1);
        for (int i = 0; i < num; i++) {
            printf("%d ", a[i]);
        }
        printf("\n");
    }

    return 0;
}
```

成绩排序

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
struct Student{
    int grade;
    int id;
    string name;
};
bool Ascend(Student a,Student b){
```

```
    if (a.grade == b.grade) {
        return a.id < b.id;
    } else{
        return a.grade < b.grade;
    }
}
bool Descend(Student a,Student b){
    if (a.grade == b.grade) {
        return a.id < b.id;
    } else{
        return a.grade > b.grade;
    }
}
int main() {
    int number;
    int way;
    while(scanf("%d %d",&number,&way)!=EOF){
    Student Stu[number];//在这里产生学生数组，防止数组溢出
    for (int i = 0; i < number; i++) {
        cin>>Stu[i].name>>Stu[i].grade;
        Stu[i].id = i;
    }
    if (way == 0) {
        sort(Stu, Stu + number, Descend);

    } else{
        sort(Stu, Stu + number, Ascend);
    }
        for (int j = 0; j < number; j++ ) {
            cout<<Stu[j].name<<" "<<Stu[j].grade<<endl;
        }
    }
    return 0;
}
```

二分查找

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
bool BinarySearch(int a[],int n,int x){
    int left = 0,right = n - 1;
    while (left <= right) {//left = right时也要确定是否a[left] = x
        int mid = left + (right - left) / 2;
        if (a[mid] < x) {
            left = mid + 1;
        }else if (a[mid] > x){
            right = mid - 1;
        }else{
            return true;
        }
    }
    return false;
}
int main() {
    int num,m;
    while(scanf("%d",&num)!=EOF){
        int arr[num];
```

```
        for (int i = 0; i < num; i++) {
            scanf("%d",&arr[i]);
        }
        sort(arr,arr + num);
        scanf("%d",&m);
        int m_arr[m];
        for (int i = 0; i < m; i++) {
            scanf("%d",&m_arr[i]);
            if(BinarySearch(arr, num,m_arr[i]) == true){
                printf("YES\n");
            } else {
                printf("NO\n");
            };
        }
    }
    return 0;
}
```

特殊乘法

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main() {
    string a,b;
    while(cin>>a>>b){
    int sum = 0;
    for (int i = 0; i < a.length(); i++) {
        for (int j = 0; j < b.length(); j++) {
            sum += (a[i] - '0')*(b[j] - '0');
        }
    }
    cout<<sum;
    }
    return 0;
}
```

简单密码

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main() {
    string str;
    while (getline(cin,str)) {
        if (str == "ENDOFINPUT") {
            break;
        }
        getline(cin, str);
        for (int i = 0; i < str.size(); ++i) {
            if (str[i] >='A' && str[i] <='Z' ) {
```

```
                str[i] = 'A' + (str[i] - 'A' - 5 + 26) % 26;//原来字母往前找5位为正确字母
            }
        }
        cout<<str<<endl;
        getline(cin, str);
    }

    return 0;
}
```

字符统计

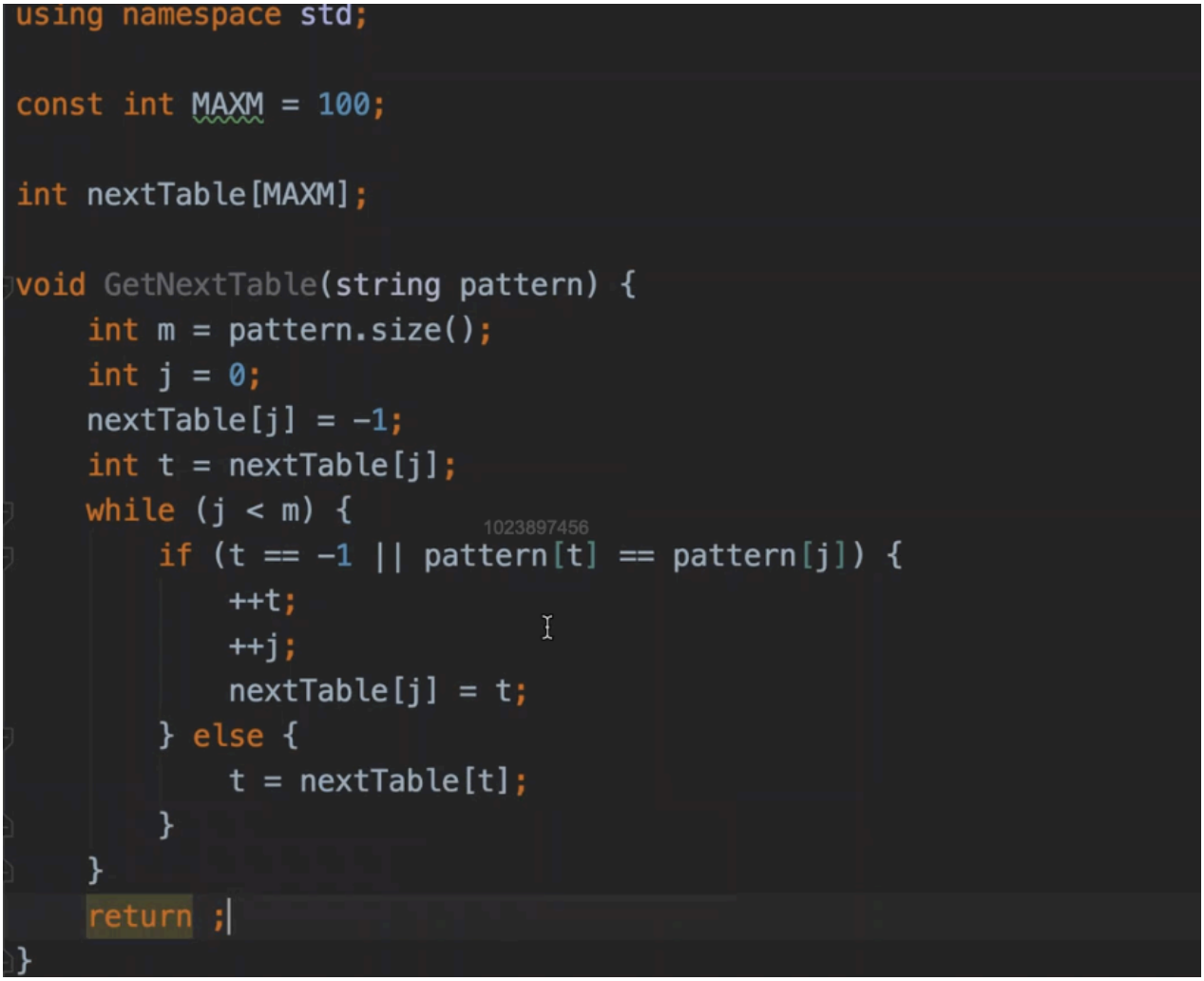
```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int number[128];
int main() {
    string Fstr,str;
    while (getline(cin, Fstr)) {
        if (Fstr == "#") {
            break;
        }
        memset(number, 0, sizeof(number));
        getline(cin, str);
        for (int j = 0; j < str.size(); ++j) {
                number[str[j]]++;
            }
        for (int i = 0; i < Fstr.size(); ++i) {
            cout<<Fstr[i]<<" "<<number[Fstr[i]]<<endl;
        }
    }

    return 0;
}
```

字母统计

```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int number[26];
int main() {
    string str;
    while (getline(cin, str)) {
        memset(number, 0, sizeof(number));
        for (int j = 0; j < str.size(); ++j) {
            if ('A' <= str[j] && str[j] <= 'Z') {
                number[str[j]-'A']++;
            }
            }
        for (int i = 0; i < 26; i++) {
```

```
            printf("%c:%d\n",'A' + i,number[i]);
        }
    }

    return 0;
}
```

KMP算法



```
using namespace std;

const int MAXM = 100;

int nextTable[MAXM];

void GetNextTable(string pattern) {
    int m = pattern.size();
    int j = 0;
    nextTable[j] = -1;
    int t = nextTable[j];
    while (j < m) {
        if (t == -1 || pattern[t] == pattern[j]) {
            ++t;
            ++j;
            nextTable[j] = t;
        } else {
            t = nextTable[t];
        }
    }
    return ;
}
```

```
int KMP(string text, string pattern) {
    GetNextTable(pattern);
    int n = text.size();
    int m = pattern.size();
    int i = 0;
    int j = 0;
    while (i < n && j < m) {
        if (j == -1 || text[i] == pattern[j]) {
            ++i;
            ++j;
        } else {
            j = nextTable[j];
        }
    }
    if (j == m) {
        return i - j;
    } else {
        return -1;
    }
}
```

```
int main() {
    string text;
    string pattern;
    text = "I love you";
    pattern = "love";
    int position = KMP(text, pattern);
    cout << position << endl;
    return 0;
}
```

KMP简单匹配

```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
//const int MAXN = 10000;
//const int MAXM = 1000000;
```

```cpp
int nextTable[20];
void GetNextTable(string pattern){
    int n = pattern.size();
    int j = 0;
    nextTable[j] = -1;
    int t = nextTable[j];//t保存当前已经算好的next数组的值（next数组存的是pattern的下标）
    while(j < n) {
        if (t == -1 || pattern[t] == pattern[j]) {
            ++j;
            ++t;
            nextTable[j] = t;
        } else{
            t = nextTable[t];//不匹配t返回上一个next数组的值继续进行比较
        }
    }
    return;
}
int KMP(string pattern,string text){
    GetNextTable(pattern);
    int m = text.size();//必须把数组长度先赋值给int型变量，不然后面循环判断会出错
    int n = pattern.size();
    int i = 0;
    int j = 0;
    while (j < n && i < m) {//一直匹配直到匹配结束
        if (j == -1 || pattern[j] == text[i]) {
            ++i;
            ++j;
        } else{
            j = nextTable[j];
        }
    }
    if (j == pattern.size() ) {//完全匹配成功后 j 小标为数组长度
        return i-j;
    }else{
        return -1;
    }
}
int main() {
    string a = "i love you ";
    string b = "you";
    int x = KMP(b, a);
    printf("%d",x);
    return 0;
}
```

记录字符串匹配次数

```cpp
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
//const int MAXN = 10000;
//const int MAXM = 1000000;
int nextTable[1000];
void GetNextTable(string pattern){
    int n = pattern.size();
    int j = 0;
    nextTable[j] = -1;
```

```
    int t = nextTable[j];
    while(j < n) {
        if (t == -1 || pattern[t] == pattern[j]) {
            ++j;
            ++t;
            nextTable[j] = t;
        } else{
            t = nextTable[t];//不匹配返回上一个next数组的下标继续进行比较
        }
    }
    return;
}
int KMP(string pattern,string text){
    GetNextTable(pattern);
    int m = text.size();//必须把数组长度先赋值给int型变量，不然后面循环判断会出错
    int n = pattern.size();
    int i = 0;
    int j = 0;
    int times = 0;
    while (i < m) {//记录匹配次数，需要一直匹配直到i到了文本的最后一个字符
            if (j == -1 || pattern[j] == text[i]) {
                ++i;
                ++j;
            } else{
                j = nextTable[j];
            }
        if (j == n ) {//完全匹配成功一次后 j 小标为数组长度，统计成功数目
            j = nextTable[j];
            times++;
        }
        }
    return times;
}
int main() {
    int caseNumber;
    scanf("%d",&caseNumber);
    while (caseNumber--) {
        string text, pattern;
        cin >> pattern >> text;
        printf("%d\n",KMP(pattern, text));
    }
    return 0;
}
```

完数与盈数

```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
vector<int> numG;
vector<int> numE;
int Sum(int n){
    int sum = 0;
    for (int i = 1; i < n; ++i) {
        if ((n % i) == 0) {
            sum += i;
```

```
        }
    }
    return sum;
}
int main() {
    for (int i = 2; i <= 60; ++i) {
        if (Sum(i) == i) {
            numE.push_back(i);
        } else if (Sum(i) > i){
            numG.push_back(i);
        }
    }
    printf("E: ");
    for (int i = 0; i < numE.size(); ++i) {
        printf("%d ",numE[i]);
    }
    printf("\n");
    printf("G: ");
    for (int i = 0; i < numG.size(); ++i) {
        printf("%d ",numG[i]);
    }
    return 0;
}
```

约瑟夫问题

- 队列解法

```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
#include <queue>
using namespace std;

int main() {
    int n,m,p,x;
    while(scanf("%d%d%d",&n,&p,&m)!=EOF){
        if (m == 0 && n == 0 && p == 0) {
            break;
        }
        queue<int> children;
        for (int i = 1; i <= n; ++i) {//将1, 2, 3...n入队
            children.push(i);
        }
        for (int i = 1; i < p; ++i) {//for循环线判断中间的判断式子是否为真再决定是否继续执行
            children.push(children.front());//从编号p开始计算，需要编号p放到队头
            children.pop();
        }
        while (!children.empty()) {
            for (int i = 1; i < m; ++i) {//m-1个小孩依次重新入队
                children.push(children.front());
                children.pop();
            }
            x = children.front();
            if (children.size() == 1) {//最后一个小孩的输出不同
                printf("%d\n",x);
```

```
            } else{
                printf("%d,",x);}
            children.pop();
        }
}
    return 0;
}
```

- 数组解法

```
#include <stdio.h>
using namespace std;
#define N 100
int main()
{
    int n,m,s;
    scanf("%d%d%d",&n,&m,&s);//s是数组起点
    int a[N] = {0};//数组初始化
    int i,j;
    for(i = 0; i < n; i++)//数组遍历
    {
        a[i] = i+1;
    }
    i=s-1;//数组起点
    while (n > 1)
    {
        i = (i + m - 1) % n;// 淘汰的人
        printf("%d\n",a[i]);
        for(j = i+1; j < n; j++)//淘汰后将后面的数字往前移动
        {
            a[j-1] = a[j];
        }
        n--;
        if(i == n)//终点后，开始起点
        {
            i = 0;
        }
    }
    printf("%d\n", a[i]);
    return 0;
}
```

文件操作

- 简单打开文件，写入char数组

```
#include <stdio.h>
#include <cstring>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    const char* filename ="/Users/yangshucheng/Desktop/C++/fushi/fushi/test.txt";
    FILE* fp = fopen(filename, "wb");
    char a[] = "hello number";
```

```
    if (fp == NULL) {
        cout << "文件打开失败！" << endl;
        exit(0);
    }
    fwrite(a, 1, sizeof(a), fp);
    fclose(fp);

    return 0;
}
```

猫狗收容所

回文数组判断

```
#include <iostream>
#include <stack>
#include <string>
using  namespace std;
string match(string str){
    string pa = str;
    for (int i = 0; i < str.size(); i++) {
        pa[i] = str[str.size() - 1 - i];
    }
    return pa;
}
int main()
{
    string str;//size_t 为unsigned int
    cout << "请输入需要判断的字符串"<<endl;
    cin >> str;
    if( str == match(str)  ){
        cout << "YES";
    }else{
        cout << "NO";
    }
    return 0;
}
```

- 利用栈的方法

```
#include <iostream>
#include <stack>
#include <string>
using  namespace std;
int main()
{
    string str;//size_t 为unsigned int
    cout << "请输入需要判断的字符串"<<endl;
    cin >> str;
    stack<char> pa ;
    string paa;
    for (int i = 0; i < str.size(); i++) {
        pa.push(str[i]);
    }
    for (int i = 0; i < str.size(); i++) {
        if (pa.top() != str[i]) {
```

```
            cout << "NO";
            return 0;
        } else{
            pa.pop();
        }
    }
    cout << "YES";
    return 0;
}
```

表达式括号匹配判断

```
#include <iostream>
#include <stack>
#include <string>
using  namespace std;
bool match(string str){
    stack<char> sta;
    for (int i = 0; i < str.size(); i++) {
        if (str[i] == '(' || str[i] == '[' || str[i] == '{') {
            sta.push(str[i]);
        }else if (str[i] == ')' || str[i] == ']' || str[i] == '}'){
            char a = sta.top();
            if ( (a == '(' && str[i] == ')') ||
                (a == '{' && str[i] == '}') ||
                (a == '[' && str[i] == ']')) {
                sta.pop();
            } else{
                return false;
            }
        }

    }
    if (!sta.empty()) {
        return false;
    }
    return true;
}
int main()
{
    string str;//size_t 为unsigned int
    cout << "请输入需要配对的字符串"<<endl;
    cin >> str;
    if(match(str)){
        cout << "YES";
    } else{
        cout << "NO";
    }
    return 0;
}
```

转换为后缀表达式求值

- 转换为后缀表达式(字母+括号+四则运算)

```
#include <iostream>
```

```cpp
#include <map>
#include <stack>
#include <string>
#include <ctype.h>
using namespace std;
int main(){
    string s;
    cin>>s;
    stack<char> S;
    map<char,int> P;
    P['+'] = P['-'] = 1;
    P['*'] = P['/'] = 2;
    for(int i=0;i<s.length();i++){
        if(s[i] == '('){
            S.push(s[i]);
            continue;
        }
        if (s[i] == ')') {
            while(S.top() != '('){
                cout<<S.top();
                S.pop();
            }
                S.pop();
            }
        if(isalpha(s[i]))
            cout<<s[i];
        else{
            while(!S.empty() && P[s[i]]<=P[S.top()]  && s[i]!= ')'){
                cout<<S.top();
                S.pop();
            }
            if (s[i] != ')') {
                S.push(s[i]);
            }
        }
    }
    while(!S.empty()){
        cout<<S.top();
        S.pop();
    }
    cout<<endl;
    return 0;
}
```

- 转换为后缀表达式(数字+括号+四则运算)

```cpp
#include <iostream>
#include <map>
#include <stack>
#include <string>
#include <ctype.h>
using namespace std;
int main(){
    string s;
    cin>>s;
    stack<char> S;
    map<char,int> P;
    P['+'] = P['-'] = 1;
    P['*'] = P['/'] = 2;
    for(int i=0;i<s.length();i++){
```

```
        if(s[i] == '('){
            S.push(s[i]);
            continue;
        }
        if (s[i] == ')') {
            while(S.top() != '('){
                cout<<S.top();
                S.pop();
            }
                S.pop();
            }
        if(isdigit(s[i])){
            int num = 0;
        do {
            num = num * 10 + (s[i] - '0'); // ch - 48根据ASCAII码，字符与数字之间的转换关系
            i++; // 下一个字符
        }while(isdigit(s[i]));
            cout << num;
            i--;
        }else{
            while(!S.empty() && P[s[i]]<=P[S.top()]   && S.top() != '(' && s[i]!= ')'){
                cout<<S.top();
                S.pop();
            }
            if (s[i] != ')') {
                S.push(s[i]);
            }
        }
    }
    while(!S.empty()){
        cout<<S.top();
        S.pop();
    }
    cout<<endl;
    return 0;
}

```

- 中缀转后缀，后缀再求值

```
#include <iostream>
#include <map>
#include <stack>
#include <string>
#include <ctype.h>
using namespace std;
string infixTosuffix(string s){//中缀表达式转后缀表达式
    stack<char> S;
    string algo;
    map<char,int> P;//优先级表
    P['+'] = P['-'] = 1;
    P['*'] = P['/'] = 2;
    for(int i=0;i<s.length();i++){
        if(s[i] == '('){//遇到‘（’直接入栈
            S.push(s[i]);
            continue;
        }
        if (s[i] == ')') {//遇到‘）’，把栈中‘（’前的运算字符依次输出
            while(S.top() != '('){
                algo += S.top();
```

```
                algo += " ";//空格为了之后后缀表达式求值能够确定多位数的边界
                S.pop();//依次弹出栈中运算字符
            }
            S.pop();//弹出‘（’符号
        }
        if(isdigit(s[i])){//遇到数字，则需要把这个数字直接保存在后缀表达式中
            int num = 0;
            do {
                num = num * 10 + (s[i] - '0'); // ch - 48根据ASCAII码，字符与数字之间的转换关系
                algo += s[i];
                i++; // 下一个字符
            }while(isdigit(s[i]));
            algo += " ";//确定数字的边界
            i--;//注意要变回当前扫描的字符
        }else{
            while(!S.empty() && P[s[i]]<=P[S.top()]   && S.top() != '(' && s[i]!= ')'){
                algo += S.top();
                algo += " ";
                S.pop();//扫描到符号的时候，如果当前扫描的符号优先级小于等于栈中符号，则出栈
            }
            if (s[i] != ')') {//当前扫描的符号优先级大于栈中符号，则入栈
                S.push(s[i]);
            }
        }
    }
    while(!S.empty()){//若栈中还有运算符号，则依次出栈
        algo += S.top();
        algo += " ";
        S.pop();
    }
    return algo;
}
int calculate(int a,int b,char ch){//根据符号来确定运算值
    switch (ch) {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            return a / b;
    }
    return 0;
}
int result(string str){
    stack<int> S;
    for (int i = 0; i < str.size(); ++i) {//依次扫描后缀表达式
        if (isdigit(str[i])) {//遇到数字，则压入栈
            int num = 0;
            while (isdigit(str[i])) {
                num = num * 10 ;
                num += (str[i] - '0');
                ++i;
            }
            --i;
            S.push(num);
        }else if(str[i] == '+' || str[i] == '-' || str[i] == '*' || str[i] == '/' ){
            int x = S.top();//遇到运算符号，则把栈中前两个数字拿出来进行相应的运算，得到结果后放回栈中
            S.pop();
            int y = calculate(S.top(), x, str[i]);
            S.pop();
            S.push(y);
```

```
        }
    }
    int x = S.top();
    S.pop();//栈中最后一个元素就是后缀表达式的运算结果
    return x;
}
int main(){
    string s;
    cin>>s;
    s = infixTosuffix(s);
    cout << s <<endl;
    cout <<result(s);
    return 0;
}
//16+2*30/4-(26-12*3)

```

- 前置表达式求值

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
double exp(){
    char s[20];
    cin >> s;
    switch (s[0]) {
        case '+':
            return exp()+exp();
        case '-':
            return exp()-exp();
        case '*':
            return exp()*exp();
        case '/':
            return exp()/exp();
        default:
            return atof(s);
        break;
    }
}
int main() {
    printf("%lf",exp());
    return 0;
}
```

> 样例输入* + 11.0 12.0 + 24.0 35.0
>
> 样例输出1357.000000
>
> // 提示：(11.0+12.0)*(24.0+35.0)

求众数以及它的重数*

问题：在一个由元素组成的表中，出现次数最多的元素称为众数，试写一个寻找众数的算法

方法：1）对输入数据进行排序

2）从每个数字出现的第一个位置开始计数，计算出现的次数，记为count，如果大于最大众数MaxCount则更新MaxCount，并记下索引位置index

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int* a = new int[n];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];//输入数据
    }
    sort(a, a + n);//数据由低到高进行排序
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
    int i = 0;
    int MaxCount = 1;
    int index = 0;
    while (i < n - 1)//遍历
    {
        int count = 1;
        int j ;
        for (j = i; j < n - 1; j++)
        {
            if (a[j] == a[j + 1])//存在连续两个数相等，则众数+1
            {
                count++;
            }else
            {
                break;
            }
        }
        if (MaxCount < count)
        {
            MaxCount = count;//当前最大众数
            index = j;//当前众数标记位置
        }
        ++j;
        i = j;//位置后移到下一个未出现的数字
    }
    cout << a[index] << " " << MaxCount << endl;
    return 0;
}
```

数制转换

递归相关

走楼梯

树老师爬楼梯，他可以每次走1级或者2级，输入楼梯的级数，求不同的走法数

思路：

n级台阶的走法 =先走一级后， n−1级台阶的走法 +先走两级后， n−2级台阶的走法

f(n) = f(n−1)+f(n−2)

边界条件：

n < 0　0　　n = 0　1　　n = 1　1

n = 0　1　　n = 1　1　　n = 2　2

```cpp
#include <iostream>
#include <cstdio>
using namespace std;
int staris(int n){
    if (n < 0) {
        return 0;
    }else if (n == 0) {
        return 1;
    }else{
        return staris(n - 1) + staris(n - 2);
    }
}
int main(){
    int n;
    scanf("%d",&n);
    cout << staris(n);
    return 0;
}
```

算24

输入

输入数据包括多行，每行给出一组测试数据，包括4个小于10个正整数。最后一组测试数据中包括4个0，表示输入的结束，这组数据不用处理。

输出

对于每一组测试数据，输出一行，如果可以得到24，输出“YES”；否则，输出“NO”。

样例输入

5 5 5 1

1 1 4 2

0 0 0 0

样例输出

YES

NO

```cpp
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;
#define EPS 1e-6
bool isZero(double x){
    return fabs(x) <= EPS;
}
double a[5];
bool count24(double a[],int n){//用数组a里的n个数，计算24
    if (n == 1) {
        if (isZero( a[0] - 24)) {//判断是否为24
            return true;
        }else{
            return false;
```

```
        }
    }
    double b[5];
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {//枚举两个数的组合
            int m = 0;//还剩下m个数
            for (int k = 0; k < n; ++k) {
                if (m != i && m != j) {
                    b[m++] = a[k];//把其余数放入b中
                }
            }
            b[m] = a[i] + a[j];
            if (count24(b, m + 1)) {
                return true;
            }
            b[m] = a[i] - a[j];
            if (count24(b, m + 1)) {
                return true;
            }
            b[m] = a[i] * a[j];
            if (count24(b, m + 1)) {
                return true;
            }
            if (!isZero(a[i])) {
                b[m] = a[j] / a[i];
                if (count24(b, m + 1)) {
                    return true;
                }
            }
            if (!isZero(a[j])) {
                b[m] = a[i] / a[j];
                if (count24(b, m + 1)) {
                    return true;
                }
            }
        }
    }
    return false;
}
int main(){
    while (true) {
        for (int i = 0; i < 4; ++i) {
            cin >> a[i];
        }
        if (isZero(a[0])) {
            break;//第一个数为0则跳出
        }
        if (count24(a, 4)) {
            cout << "YES\n";
        }else{
            cout << "NO\n";
        }
    }
    return 0;
}

```

寻找第k大的数

```
#include<iostream>
```

```cpp
#include<vector>
#include <algorithm>
using namespace std;
int QuickSortK(vector<int>arr, int begin, int end,int k)
{
    if (begin == end)return arr[begin];
    int first = begin;
    int last = end;
    int key = arr[first];
    while (first < last)
    {
        while (first < last && key <= arr[last])
            last--;
        arr[first] = arr[last];
        while (first < last && key >= arr[first])
            first++;
        arr[last] = arr[first];
    }
    arr[first] = key;
    if (first+1 == k)return arr[first];
    else if (first+1 < k)return QuickSortK(arr, first + 1, end, k);
    else return QuickSortK(arr, begin, first - 1, k);
}
int findk(vector<int>arr,int k)
{
    int len = arr.size();
    return QuickSortK(arr, 0, len - 1, k);
}
int main()
{
    vector<int>a = { 6,5,7,8,1,3,4,9,2,0 };
    int No = findk(a, 9);
    sort(a.begin(), a.end());//对vector的排序方法
    cout << No << endl;
    cout << a[8];
    return 0;
}
```

汉诺伊塔

- 经典汉诺伊塔

```cpp
#include <iostream>
using namespace std;
void Hanoid(int n,char a,char b,char c){
    if(n == 1){//只剩下一个盘子，从A移动到C
        cout << a << " move to " << c <<endl;
    }else{
    Hanoid(n - 1, a, c, b);//还有些盘子，先把前n - 1个盘子从A经过C移动到B
    cout << a << " move to " << c << endl;//将A上的最后一个盘子移动到C上
    Hanoid(n - 1, b, a, c);//将B上的盘子经过A移动到C
        return;}
}
int main(){
    int n;
    cout << "n : ";
    cin >> n;
    Hanoid(n, 'A', 'B', 'C');
```

```
    return 0;
}

```

斐波那契非递归

```
#include <iostream>
#include <cstdio>
using namespace std;
int Fib(unsigned int n){
    unsigned int f,f1,f2;
    f = f1 = f2 = 1;
    if (n == 1 || n == 2) {
        return f;
    }
    for (int i = 3; i <= n; ++i) {
        f = f1 + f2;
        f2 = f1;
        f1 = f;
    }
    return f;
}
int main(){
    unsigned int n;
    cin >> n;
    cout << Fib(n) << endl;
    return 0;
}
```

杨辉三角

- 递归算法

```
#include <iostream>
#include <cstdio>
#include <cmath>
#include <iomanip>
using namespace std;
int yanghui(int i,int j){
    if (j > i){
        return 0;
    }else if (j == 0) {
        return 1;
    }else {
        return yanghui(i - 1, j - 1) + yanghui(i - 1, j);
    }
}
int main(){
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cout << setw(n-i);
        for (int j = 0; j <= i; ++j) {
```

```
            cout << yanghui(i, j) <<" ";
        }
        cout << endl;
    }
    return 0;
}

```

队列算法

```
#include <iostream>
#include <queue>
#include <iomanip>
using  namespace std;
void yanghuitriangle(int n)
{
    queue<int> Q;
    int space = n;//空格
    Q.push(1);//第一行元素
    for (int j = 1; j <= n; j++) {//打印第n行数列，产生第n+1行数列
        Q.push(1);//第n+1行的第一个元素
        cout << setw(space--);//输出每行前面的空格
        for (int i = 1; i < j ; i++) {//第n行输出前n-1个数据
            cout << Q.front()<<" ";//输出队列第一个元素
            int fro = Q.front();//保存队列第一个元素的数据
            Q.pop();//弹出队列第一个数据
            Q.push(fro + Q.front());//将每次循环时的最前面的两个数据的和放入队列末尾
        }
        cout << Q.front() <<endl;//输出最后一个元素
        Q.pop();
        Q.push(1);//最后一个元素永远是1
    }
}

int main()
{
    int N;//size_t 为unsigned int
    cout << "请输入杨辉三角的行数：";
    cin >> N;
    yanghuitriangle(N);
    return 0;
}

```

全排练问题*

```
#include <cstdio>
#include <iostream>
using namespace std;
const int MAXN = 20;
int P[MAXN],n;
bool hashmap[MAXN];
void generateP(int index){
    if (index == n + 1) {//递归边界，已经处理完排列的1~n位
        for (int i = 1; i <= n; ++i) {//输出当前排列
            cout << P[i] << " ";
        }
```

```
        cout << endl;
    }else{
        for (int x = 1; x <= n; ++x) {//枚举1~n，试图将x填入到P[index]
            if (hashmap[x] == false) {//如果x不在P[0]~P[index - 1]中
                P[index] = x;//令P的第index位为x，即把x加入到当前排列
                hashmap[x] = true;//记x已经在P中
                generateP(index + 1);//处理排列的第index+1号位置
                hashmap[x] = false;//已经处理完P[index]为x的子问题，还原状态
            }
        }
    }
}
int main(){
    cin >> n;
    generateP(1);//从P[1]开始填充
    return 0;
}
```

归并排序

```
#include <iostream>
#include "cstdio"
using namespace std;
const int MAXN = 100;
void Mergesort(int A[],int L1,int R1,int L2,int R2){
    int temp[MAXN];
    int idx = 0;
    int i,j;//分别指向开头
    for ( i = L1,j = L2; i <= R1 && j <= R2 ;) {
        if (A[i] <= A[j]) {//小的那个加入临时数组
            temp[idx++] = A[i++];
        } else {
            temp[idx++] = A[j++];
        }
    }
    //L1,R1,L2,R2指的是数组的下标
//有可能i等于前列的最后一个元素的数组下标，放入临时数组中
    while(i <= R1) {temp[idx++] = A[i++];}
    while(j <= R2) {temp[idx++] = A[j++];}
    for (int i = 0; i < idx; i++) {
        A[i + L1] = temp[i];
    }
}
void Merge(int A[],int left,int right){
    if (left < right) {//递归跳出条件
        int mid = left + (right - left) / 2;
        Merge(A,left,mid);//前半部分排好序
        Merge(A,mid+1,right);//后半部分排好序
        Mergesort(A, left, mid, mid+1, right);//归并d
    }
}
int main(){
    int A[8] = {2,24,76,102,44,15,59,13};
    Merge(A, 0, 7);
    for (int i = 0; i < 8; i++) {
        cout << A[i] << " ";
    }
    return 0;
}
```

分治法求最大子序列和

```cpp
#include <iostream>
#include <cstdio>
#include <algorithm>
#define INF 0x3f3f3f3f
using namespace std;
const int MAXN = 99999999;
int a[MAXN];
int MaxSequ(int low,int high){
    if (high <= low ) {//分治到最小单位，返回其中任意一个
        return a[low];
    }
    int mid = (low + high)>>1;//找中间数
    int tmp1 = -INF,tmp2 = -INF;//一开始为负无穷大，方便后面的比较
    int Midleft = MaxSequ(low,mid);//左边最大子序列和
    int Midright = MaxSequ(mid + 1 ,high);//右边最大子序列和
    int sum = 0;
    for (int i = mid; i >= low; i--) {//从中间开始，往前计算最大子序列和，实现跨中间元素相加
        sum += a[i];//计算当前扫描到的数与中间数之间的序列和
        tmp1 = sum > tmp1 ? sum : tmp1;//选出 原先的子序列的最大值 与 现在的 更大的一个
    }
    sum = 0;
    for (int i = mid + 1; i <= high; i++) {//从中间开始，往后计算最大子序列和，实现跨中间元素相加
        sum += a[i];//计算当前扫描到的数与中间数之间的序列和
        tmp2 = sum > tmp2 ? sum : tmp2;//选出 原先的子序列的最大值 与 现在的 更大的一个
    }
    sum = tmp1 + tmp2;//sum是跨越中间序列的最大值
    int x = max(Midleft, sum);//选出三者最大值
    return max(x,Midright);
}
int main(){
    int n;
    while ( cin >> n )
    {
        for ( int i = 0; i < n; i ++ )
            cin >> a[i];
        cout << MaxSequ(0, n-1)<< endl;
    }
    return 0;
}
```

无优先级运算

- 最多需要多少个数字

△小明玩一个游戏，小明手上有一张牌，桌子上有六张牌，游戏规则是，求用最多的桌上卡牌与小明手中的卡牌进行加减乘除四则运算 最后使果达到 13. 如小明卡牌是 5，桌上卡牌是 1 3 9 4 2 2，此题答案即 5+1/3+9+4-2 即最多可容纳 5 张牌（此题不考乘除优先级，先到优先，桌上卡牌可重）。

从 input.txt 中读入数据	往 output.txt 中写入数据
5 1 3 9 4 2 2	5 5+1/3+9+4-2

```cpp
#include <vector>
#include <cstdio>
#include <iostream>
#define N 4
using namespace std;
int n[100];
vector<int> num_v, renum;
vector<char> char_v, rechar;
int max_len = 0;
double re;
char oper[N] = { '/', '*', '-', '+'};
void seq(int k, int len, double sum) {//k为利用的数字的位数
    if (sum == re && k > max_len) {//如果当前计算结果sum=re要求的结果并且第一次满足，则跳出递归，保存结果
        renum = num_v;          //数字结果存入renum
        rechar = char_v;        //字符结果存入rechar
        max_len = k;            //递归一次就好了
    }
    if (k >= len) return;       //k大于总长度时，退出递归，表示找不到正确的结果
    num_v.push_back(n[k]);      //将第k位数字加入num_v
    for (int i = 0; i < N; i++) {//对字符进行选择，先往符号数组加入，再递归调用
        char op = oper[i];
        if (op == '+') {
            char_v.push_back('+');
            seq(k + 1, len, sum + n[k]);
        }
        else if (op == '*') {
            char_v.push_back('*');
            seq(k + 1, len, sum * n[k]);

        }
        else if (op == '-') {
            char_v.push_back('-');
            seq(k + 1, len, sum - n[k]);
        }
        else if (op == '/') {
            char_v.push_back('/');
            seq(k + 1, len, sum / n[k]);
        }
        char_v.pop_back();//都试过却不可以得出正确结果则把符号弹出
    }
    num_v.pop_back();
```

```
}
int main() {
    int len;
    double m;//m 小明手上牌的数字，  re 运算结果的要求  ，len  桌上牌的个数
    scanf("%lf %lf %d", &m, &re, &len);
    for (int i = 0; i < len; i++)// 桌上牌的数字
        scanf("%d", &n[i]);
    int k = 0;
    double sum = m;//sum为当前的计算结果
    seq(k, len, sum);
    cout << m <<" ";
    for (int i = 0; i < rechar.size() ; ++i) {
        if( i < renum.size()){
            cout << rechar[i] << " ";
        }
        cout << renum[i] << " ";
    }
    return 0;
}

```

- 输出所有结果

```
#include <vector>
#include <cstdio>
#include <iostream>
#define N 4
using namespace std;
int n[100];
vector<int> num_v, renum;
vector<char> char_v, rechar;
int max_len = 0;
double re;
char oper[N] = { '/', '*', '-', '+'};
void seq(int m,int k, int len, double sum) {//k为利用的数字的位数
    if (sum == re ) {//如果当前计算结果sum=re要求的结果并且第一次满足，则跳出递归，保存结果
        renum = num_v;          //数字结果存入renum
        rechar = char_v;        //字符结果存入rechar
        cout << m <<" ";
        for (int i = 0; i < rechar.size() ; ++i) {
            if( i < renum.size()){
                cout << rechar[i] << " ";
            }
            cout << renum[i] << " ";
        }
        cout << endl;
        return;
    }
    if (k >= len) return;       //k大于总长度时，退出递归，表示找不到正确的结果
    num_v.push_back(n[k]);      //将第k位数字加入num_v
    for (int i = 0; i < N; i++) {//对字符进行选择，先往符号数组加入，再递归调用
        char op = oper[i];
        if (op == '+') {
            char_v.push_back('+');
            seq(m,k + 1, len, sum + n[k]);
        }
        else if (op == '*') {
            char_v.push_back('*');
            seq(m,k + 1, len, sum * n[k]);

```

```
        }
        else if (op == '-') {
            char_v.push_back('-');
            seq(m,k + 1, len, sum - n[k]);
        }
        else if (op == '/') {
            char_v.push_back('/');
            seq(m,k + 1, len, sum / n[k]);
        }
        char_v.pop_back();//都试过却不可以得出正确结果则把符号弹出
    }
    num_v.pop_back();
}
int main() {
    int len;
    double m;//m 小明手上牌的数字， re 运算结果的要求 ，len 桌上牌的个数
    scanf("%lf %lf %d", &m, &re, &len);
    for (int i = 0; i < len; i++)// 桌上牌的数字
        scanf("%d", &n[i]);
    int k = 0;
    double sum = m;//sum为当前的计算结果
    seq(m, k, len, sum);
    return 0;
}
```

贪心算法

神奇的口袋

搬桌子*

某教学大楼一层有n个教室，从左到右依次编号为1、2、…、n。现在要把一些课桌从某些教室搬到另外一些教室，每张桌子都是从编号较小的教室搬到编号较大的教室，每一趟，都是从左到右走，搬完一张课桌后，可以继续从当前位置或往右走搬另一张桌子。输入数据：先输入n、m，然后紧接着m行输入这m张要搬课桌的起始教室和目标教室。输出数据：最少需要跑几趟。
Sample Input
10 5
1 3
3 9
4 6
6 10
7 8
Sample Output
3

分析：贪心算法，把课桌按起点从小到大排序，每次都是搬离当前位置最近的课桌。

```
#include<stdio.h>
int main()
{
    struct
    {
        int start;
```

```
        int end;
    }a[100];
    int i,j;
    int n,m,min,num,temp,used[100]={0};
    scanf("%d%d",&m,&n);
    for(i=0;i<n;i++)
        scanf("%d%d",&a[i].start,&a[i].end);
    sort(a,a + n);          //按start从小到大对数组a排序
    min=0;
    num=0;
    while(num<n)
    {
        temp=0;
        for(i=0;i<n;i++)
            if(used[i]==0&&a[i].start>=temp)
            {
                temp=a[i].end;
                used[i]=1;
                num++;
            }
        min++;
    }
    printf("%d",min);
}
```

堆石头*

【问题】
抓石子游戏又称为巴什博奕，简单描述一下，有一堆n个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取m个。取到最后一个物品者得胜。
【解法】
显然，如果n=m+1，那么由于一次最多只能取m个，所以，无论先取者拿走多少个，后取者都能够一次拿走剩余的物品，后者取胜。因此我们发现了如何取胜的法则：分段来进行讨论，如果n<=m，先手的赢，在n>m中再分情况讨论，如果n为m+1的整数i倍，假想一下，把物品平均分为i堆，一堆m+1个，不就是把前面提到的过程重复i次嘛，所以，后手者是肯定赢的。如果n=(m+1)*i+j，当然可以知道 j < m + 1 ，j的范围是[1,m]中的整数，就是把物品按一堆 m + 1 个分，又多出来 j 个。结果就相反，先手的人可以直接把 j 个东西全部拿走，剩下(m + 1) * i 个，就到上面分析的那种情况了，于是乎这种情况下后手者就被动了。
所以要判断这个游戏的结果，只需知道石子的开始个数、一次最多能拿多少和谁先手就可以了。
【实现】
由于和先手有关，为了简化实现过程，在这里按照自己先手来编写。

```
#include<iostream>
using namespace std;
void myfunction(int m,int n)
{
    if (n % (m + 1) == 0)
        cout << "你将失败" << endl;
    else
        cout << "你将胜利" << endl;
}
int main()
{
    int m, n;
    cout << "请分别输入最初石子数和一次最多拿的石子数：";
    cin >> n >> m;
    myfunction(m, n);
}
```

活动安排*

```cpp
#include <iostream>
#include <cstdio>
#include <algorithm>
using namespace std;
struct Program{
    int starttime;
    int endtime;
};
Program arr[100];
bool compare(Program A,Program B){
    return A.endtime < B.endtime;
}
int main(){
    int n;
    while(scanf("%d",&n)!=EOF){
        if(n == 0){break;}
        for (int i = 0; i < n; ++i) {
            scanf("%d%d",&arr[i].starttime,&arr[i].endtime);
        }
        sort(arr, arr + n, compare);
        int currentTime = 0;
        int count = 0;
        for (int i = 0; i < n; ++i) {
            if (currentTime <= arr[i].starttime) {
                currentTime = arr[i].endtime;
                count++;
            }
        }
        cout << count;

    }
    return 0;
}
```

马踏棋盘*

- 深度优先搜索

```cpp

#include <iostream>
#include <cstdio>
#include <string>
#include <cstring>
using namespace std;
const int MAXN = 30;
int p,q;
bool visit[MAXN][MAXN];
int direction[8][2]{
    {-1,-2},{1,-2},
    {-2,-1},{2,-1},
    {-2,1},{2,1},
    {-1,2},{1,2}
```

```
};
bool DFS(int x,int y,int step,string ans){
    if (step == p * q) {
        cout << ans << endl <<endl;
        return true;
    }else{
        for (int i = 0; i < 8; ++i) {
            int nx = x + direction[i][0];
            int ny = y + direction[i][1];
            char col = ny + 'A';
            char row = nx + '1';
            if (nx < 0 || nx >= p || ny < 0 || ny >= q || visit[nx][ny]) {
            //边界之外或者已经走过的格子则跳过
                continue;
            }
            visit[nx][ny] = true;
            if (DFS(nx,ny,step + 1,ans + col + row)) {
                return true;
            }
            visit[nx][ny] = false;
        }
    }
    return false;
}
int main(){
    int n;
    scanf("%d",&n);
    int caseNumber = 0;
    while (n--) {
        scanf("%d%d",&p,&q);
        memset(visit, false, sizeof(visit));
        cout << "Scenario #" << ++caseNumber << ":" << endl;
        visit[0][0] = true;
        if (!DFS(0,0,1,"A1")) {
            cout << "impossible" <<endl<<endl;
        }
    }
    return 0;
}
```

- 递归解法+深度优先搜索

```
#include <iostream>
#include <cstdio>
#include <string>
#include <cstring>
#include <iomanip>
using namespace std;
int visit[8][8];//标记矩阵
int direction[8][2]{//下一步走的位置
    {-1,-2},{1,-2},
    {-2,-1},{2,-1},
    {-2,1},{2,1},
    {-1,2},{1,2}
};
void printChess(){
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            cout <<setw(3)<< visit[i][j] ;
        }
```

```
        cout << endl;
    }
}
bool DFS(int x,int y,int step){
    if (step > 64) {//搜索成功
        printChess();
        return true;
    }else{
        for (int i = 0; i < 8; ++i) {//遍历邻居结点
            int nx = x + direction[i][0];//扩展坐标状态
            int ny = y + direction[i][1];
            if (nx < 0 || nx >= 8 || ny < 0 || ny >= 8 || visit[nx][ny] != 0) {
                //边界之外或者已经走过则跳过
                continue;
            }
            visit[nx][ny] = step;//标记该点
            if (DFS(nx,ny,step + 1)) {
                return true;
            }
            visit[nx][ny] = 0 ;//取消标记
        }
    }
    return false;
}
int main(){
    memset(visit, 0, sizeof(visit));
    visit[0][0] = 1;//标记A1点
    if (DFS(0,0,2)) {
        cout << "OK" <<endl<<endl;
    }else{
        cout << "NO" <<endl<<endl;
    }
    return 0;
}
```

- 递归深度优先搜索+贪心解法

```
#include <iostream>
#include <iomanip>
#include <algorithm>
using namespace std;
typedef struct Node {
    int x, y, w;
    Node(){}
    Node(int x, int y, int w) :x(x), y(y), w(w){}
};
int visit[8][8];//标记矩阵
Node nextxy[8];//下一点要走的位置以及下一点的可能走的下一位置的个数
int direction[8][2]{//下一步走的位置
    {-1,-2},{1,-2},
    {-2,-1},{2,-1},
    {-2,1},{2,1},
    {-1,2},{1,2}
};
void printChess() {
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            cout << setw(3) << visit[i][j];
        }
        cout << endl;
```

```
    }
}
bool cmp(Node a, Node b) {
    return a.w < b.w;
}
void next(int x, int y) {
    for (int i = 0; i < 8; ++i) {
        nextxy[i] = Node(-1, -1, 100);
        // 将初始值的坐标定位(-1,-1)，即不合法的位置
        //w为下一位置可能的落点数，w定很大另它排序到后面
    }
    for (int i = 0; i < 8; ++i) {// 遍历邻居结点
        int nx = x + direction[i][0];// 扩展坐标状态
        int ny = y + direction[i][1];
        if (nx < 0 || nx >= 8 || ny < 0 || ny >= 8 || visit[nx][ny] != 0) {
            // 边界之外或者已经走过则跳过
            continue;
        }else {
            nextxy[i].x = nx;
            nextxy[i].y = ny;
            nextxy[i].w = 1 ;
        }
        for (int j = 0; j < 8; ++j) {
            int nx2 = nx + direction[j][0];// 扩展坐标状态
            int ny2 = ny + direction[j][1];
            if (nx2 < 0 || nx2 >= 8 || ny2 < 0 || ny2 >= 8 || visit[nx2][ny2] != 0) {
                // 边界之外或者已经走过则跳过
                continue;
            }
            nextxy[i].w++;
        }
    }
    sort(nextxy, nextxy + 8, cmp);
}
bool DFS(int x, int y, int step) {
    if (step > 64) {//搜索成功
        printChess();
        return true;
    }else {
        next(x, y);
        int i = 0;
        while (i < 8 && (nextxy[i].x != -1 && nextxy[i].y != -1) ) // 位置可行
        {
            visit[nextxy[i].x][nextxy[i].y] = step;
            if (DFS(nextxy[i].x, nextxy[i].y, step + 1)) // 进入新位置
                return true;
            i++;
        }
            visit[nextxy[i].x][nextxy[i].y] = 0; // 此路不可走，回退
        return false;
    }
}
int main() {
    memset(visit, 0, sizeof(visit));
    visit[1][1] = 1;//标记起点
    if (DFS(1, 1, 2)) {
        cout << "OK" << endl << endl;
    }else {
        cout << "NO" << endl << endl;
    }
    return 0;
}
```

图

哈密顿图

简单回路

并查集

- 求能够连通需要的边数

```
#include <iostream>
#include <cstdio>
using namespace std;
//连通问题：并查集，观察是否在同一个集合当中
const int MAXN = 1000;
int father[MAXN];
int height[MAXN];
void initial(int n){
    for (int i = 1; i <= n; ++i) {
        father[i] = i;
        height[i] = 0;
    }
}
int findFather(int x){
    if (x != father[x]) {
        x = findFather(father[x]);
    }
    return x;
}
void UnionP(int a,int b){
    int x = findFather(a);
    int y = findFather(b);
    if (height[x] < height[y]) {
        father[x] = father[y];
    }else if (height[x] > height[y]){
        father[y] = father[x];
    }else{
        father[y] = father[x];
        height[y]++;
    }
}
int main(){
    int n,m;
    while (scanf("%d",&n)!=EOF) {
        initial(n);
        scanf("%d",&m);
        while (m--) {
            int x,y;
            scanf("%d %d",&x,&y);
            UnionP(x, y);
```

```
        }
        int ans = -1;
        for (int i = 1; i <= n; ++i) {
            if (i == father[i]) {
                ans++;
            }
        }
        printf("%d",ans);
    }
}
```

- 求是否为无向连通图

```
#include <iostream>
#include <cstdio>
using namespace std;
//连通问题：并查集，观察是否在同一个集合当中
const int MAXN = 1000;
int father[MAXN];
int height[MAXN];
void initial(int n){
    for (int i = 1; i <= n; ++i) {
        father[i] = i;
        height[i] = 0;
    }
}
int findFather(int x){
    if (x != father[x]) {
        x = findFather(father[x]);
    }
    return x;
}
void UnionP(int a,int b){
    int x = findFather(a);
    int y = findFather(b);
    if (height[x] < height[y]) {
        father[x] = father[y];
    }else if (height[x] > height[y]){
        father[y] = father[x];
    }else{
        father[y] = father[x];
        height[y]++;
    }
}
int main(){
    int n,m;
    while (scanf("%d",&n)!=EOF) {
        initial(n);
        scanf("%d",&m);
        while (m--) {
            int x,y;
            scanf("%d %d",&x,&y);
            UnionP(x, y);
        }
        int ans = -1;
        for (int i = 1; i <= n; ++i) {
            if (i == father[i]) {
                ans++;
            }
        }
```

```
        if (ans == 0) {
            printf("YES\n");
        }else{
            printf("NO\n");
        }
    }
}
```

- 求是否为有向树

```
#include <iostream>
#include <cstdio>
using namespace std;
//无向连通图问题：并查集，观察是否在同一个集合当中
const int MAXN = 1000;
int father[MAXN];   //父亲节点
int height[MAXN];   //结点高度
bool visited[MAXN]; //标记
int indegree[MAXN]; //入度
void initial(int n){//初始化
    for (int i = 1; i <= n; ++i) {
        father[i] = i;
        height[i] = 0;
        visited[i] = false;
        indegree[i] = 0;
    }
}
int findFather(int x){  //查找根节点
    if (x != father[x]) {
        x = findFather(father[x]);
    }
    return x;
}
void UnionP(int a,int b){       // 合并集合
    int x = findFather(a);      // 找最原始结点
    int y = findFather(b);
    if (x != y) {   //原始结点不一样
        if (height[x] < height[y]) {//短的并上长的，高度不变
            father[x] = father[y];
        }else if (height[x] > height[y]){
            father[y] = father[x];
        }else{
            father[y] = father[x];
            height[y]++;//两个长度相同结点并为一个，高度+1
        }
    }
    return;
}
bool isTree(int n){
    bool flag = true;
    int component = 0;  //连通分量数目
    int root = 0;       //根结点数
    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) {  //如果没有被访问过，则逃过
            continue;
        }
        if (i == father[i]) {   //找到了最原始结点
            component++;        //连通分量+1
        }
        if (indegree[i] == 0){  //入度为0，则是根节点
```

```
            root++;
        }else if (indegree[i] > 1) {    //入度大于1，不是树
            flag = false;
        }
        if (root != 1 || component != 1) {  //不符合树的定义
            flag = false;
        }
        if (root == 0 && component == 0) {  //空集也是树
            flag = true;
        }
    }
    return flag;
}
int main(){
    int n,m,caseNum = 1;
    initial(MAXN);
    while (scanf("%d %d",&n,&m)!=EOF) {
        if (n == -1 && m == -1) {
            break;
        }
        if (n == 0 && m == 0) {
            if (isTree(MAXN)) {
                printf("Case %d is a tree.\n",caseNum++);
            }else{
                printf("Case %d is not a tree.\n",caseNum++);
            }
            initial(MAXN);//本次结果完毕，开始新的判断，需要重置所有数据
        }else{
            UnionP(n, m);
            indegree[m]++;
            visited[n] = true;
            visited[m] = true;
        }
    }
    return 0;
}
```

求有向图强连通图

连通图着色*

广义表->二叉链表*

最小生成树

- 未修好路

输入描述:

测试输入包含若干测试用例。每个测试用例的第1行给出村庄数目N（< 100）；随后的N(N-1)/2行对应村庄间的距离，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间的距离。为简单起见，村庄从1到N编号。当N为0时，输入结束，该用例不被处理。

输出描述:

对每个测试用例，在1行里输出最小的公路总长度。

```cpp
#include <iostream>
#include <algorithm>
#include <cstdio>
using namespace std;
struct Edge{
    int from;
    int to;
    int length;
    bool operator < (const Edge& a)const{
        return length < a.length;
    }
};
const int MAXN = 100;
Edge edge[MAXN];
int father[MAXN];
int height[MAXN];
void initialA(int n){
    for (int i = 0; i <= n; ++i) {
        father[i] = i;//一开始单个结点的父亲是自己
        height[i] = 0;
    }
}
int Find(int x){
    if (x != father[x]) {
        father[x] = Find(father[x]);//查找根节点
    }
    return father[x];
}
void UnionP(int a,int b){
    int x = Find(a);
    int y = Find(b);
    if (x!=y) {//当两个结点不同的时候
        if (height[x] < height[y]) {
            father[x] = y;
        }else if(height[x] > height[y]){
            father[y] = x;
        }else{
            father[y] = x;
            height[x]++;
        }
    }
    return;
}
int Kruskal(int n,int edgeNum){
    initialA(n);//先初始化
    int sum = 0;
    sort(edge, edge + edgeNum);//按权值排序
    for (int i = 0; i < edgeNum; ++i) {
        Edge current = edge[i];//确定当前边
        if (Find(current.from) != Find(current.to)) {//确定当前边起始结点的根节点
            UnionP(current.from, current.to);//若短边没有在最小生成树集合中，则将他们归并其中
            sum += current.length;
        }
    }
    return sum;
}
int main(){
    int n;
    while (scanf("%d",&n) != EOF) {
        if (n == 0) {
            break;
        }
```

```
        int edgeNum = n * (n - 1) / 2;
        for (int i = 0; i < edgeNum; ++i) {
            scanf("%d%d%d",&edge[i].from,&edge[i].to,&edge[i].length);
        }
        int ans = Kruskal(n, edgeNum);
        printf("%d\n",ans);
    }
    return 0;
}
```

- 修部分路

```
#include <iostream>
#include <cstdio>
#include <algorithm>
using namespace std;
const int MAXN = 100;
struct Edge{
    int from;
    int to;
    int length;
    bool operator <(const Edge &a)const{
        return length < a.length;
    }
};
int father[MAXN];
int height[MAXN];
Edge edge[MAXN*MAXN];
void initial(int n){
    for (int i = 0; i < n; ++i) {
        father[i] = i;
        height[i] = 0;
    }
}
int findf(int x){//寻找根节点
    if (x != father[x]) {
        father[x] = findf(father[x]);
    }
    return father[x];
}
void Union(int x,int y){
    x = findf(x);
    y = findf(y);
    if (x != y) {
        if (height[x] < height[y]) {
            father[x] = y;
        }else if (height[x] > height[y]){
            father[y] = x;
        }else{
            father[y] = x;
            height[x]++;
        }
    }
}
int Kruskal(int n,int edgeNumber){
    initial(n);
    int sum = 0;
    sort(edge, edge + edgeNumber);
    for (int i = 0; i < edgeNumber; ++i) {
        Edge current = edge[i];
```

```
        if (findf(current.from) != findf(current.to)) {
            Union(current.from, current.to);
            sum += current.length;
        }
    }
    return sum;
}
int main(){
    int n;
    while (scanf("%d",&n)!=EOF) {
        if (n == 0) {
            break;
        }
        int edgeNumber = n * (n - 1) / 2;
        for (int i = 0; i < edgeNumber; ++i) {
            int status;
            scanf("%d%d%d%d",&edge[i].from,&edge[i].to,&edge[i].length,&status);
            if (status == 1) {
                edge[i].length = 0;
            }
        }
        int ans = Kruskal(n, edgeNumber);
        printf("%d\n",ans);
    }
    return 0;
}
```

最短路径

Dijkstra

单纯最短路径

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;
const int INF = 99999999;
const int MAXN = 200;
struct Edge{
    int to;//边的终点
    int length;//边的权值
    Edge(int t,int l):to(t),length(l){}
};
vector<Edge> graph[MAXN];
int dis[MAXN];
struct Point{
    int number;//点的编号
    int distance;//点到源点的距离
    Point(int n,int d):number(n),distance(d){}
    bool operator <(const Point& a)const{
        return distance>a.distance;
    }
};
void Dijkstra(int n){
    priority_queue<Point> myQueue;
```

```
    dis[n] = 0;
    myQueue.push(Point(n,dis[n]));
    while (!myQueue.empty()) {
        int num = myQueue.top().number;//找出与当前最短路径点集合中相连边权值最小的点序号num
        myQueue.pop();//弹出
        for (int i = 0; i < graph[num].size(); ++i) {
//遍历与num相连的边的点v，如果包含最短路径，则更新v的最短距离
            int v = graph[num][i].to;
            int w = graph[num][i].length;
            if (dis[v] > dis[num] + w) {
                dis[v] = dis[num] + w;
                myQueue.push(Point(v, dis[v]));//如果有更短的路径，则加入最短路径点集合
            }
        }
    }
    return;
}
int main(){
    int n,m;
    while (scanf("%d%d",&n,&m)!=EOF) {
        memset(graph, 0, sizeof(graph));//图初始化
        fill(dis, dis + n, INF);//初始化距离为正无穷
        while (m--) {
            int a,b,x;
            scanf("%d%d%d",&a,&b,&x);
            graph[a].push_back(Edge(b,x));//加入边的数据
            graph[b].push_back(Edge(a,x));
        }
        int from,to;
        scanf("%d%d",&from,&to);
        Dijkstra(from);
        if(dis[to] == INF){
            dis[to] = -1;
        }
        printf("%d\n",dis[to]);
    }
    return 0;
}

```

最短路径下的最少花费

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;
const int INF = 99999999;
const int MAXN = 1001;
struct Edge{
    int to;//边的终点
    int length;//边的权值
    int price;//边的花费
    Edge(int t,int l,int p):to(t),length(l),price(p){}//构造函数
};
vector<Edge> graph[MAXN];//图的h数组
int dis[MAXN],cost[MAXN];//距离数组，花费数组
struct Point{
    int number;//点的编号
```

```
    int distance;//点到源点的距离
    Point(int n,int d):number(n),distance(d){}
    bool operator <(const Point& a)const{//距离小的优先级高
        return distance>a.distance;
    }
};
void Dijkstra(int n){
    priority_queue<Point> myQueue;//存储起点到各个点的最短路径数据（编号与距离）
    dis[n] = 0;
    cost[n] = 0;
    myQueue.push(Point(n,dis[n]));//放入起点
    while (!myQueue.empty()) {
        int num = myQueue.top().number;//找出与当前最短路径点集合中相连边权值最小的点序号num
        myQueue.pop();//弹出
        for (int i = 0; i < graph[num].size(); ++i) {
            //遍历与num相连的边的点v，如果包含最短路径，则更新v的最短距离
            int v = graph[num][i].to;
            int w = graph[num][i].length;
            int p = graph[num][i].price;
            if (dis[v] == dis[num] + w && cost[v] > cost[num] + p
                || dis[v] > dis[num] + w) {
                //优先选择距离短的，在距离相等时选择花费更少的
                dis[v] = dis[num] + w;
                cost[v] = cost[num] + p;
                myQueue.push(Point(v, dis[v]));//如果有更短的路径，则加入最短路径点集合
            }
        }
    }
    return;
}
int main(){
    int n,m;
    while (scanf("%d%d",&n,&m)!=EOF) {
        if (n == 0 && m == 0) {
            break;
        }
        memset(graph, 0, sizeof(graph));//图初始化
        fill(dis, dis + n + 1, INF);//初始化距离为正无穷
        fill(cost, cost + n + 1, INF);//初始化花费为正无穷
        while (m--) {
            int a,b,x,p;
            scanf("%d%d%d%d",&a,&b,&x,&p);
            graph[a].push_back(Edge(b,x,p));//加入边的数据
            graph[b].push_back(Edge(a,x,p));
        }
        int from,to;
        scanf("%d%d",&from,&to);
        Dijkstra(from);
        if(dis[to] == INF){
            dis[to] = -1;
        }
        printf("%d%d\n",dis[to],cost[to]);
    }
    return 0;
}

```

Floyd多点最短路径

当dis[i][j] < dis[i][k] + dis[k][j]时

dis[i][j] = dis[i][k] + dis[k][j];//找到更短的路径

```cpp
#include <cstdio>
#include <iostream>
#include <algorithm>
using namespace std;
const int INF = INT_FAST32_MAX;
const int MAXN = 200;//最大顶点数目
int dis[MAXN][MAXN];//dis[i][j]表示顶点i到顶点j的最短距离
void Floyd(int n){
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dis[i][k] != INF && dis[k][j] != INF && dis[i][j] > dis[i][k] + dis[k][j]) {
                    dis[i][j] = dis[i][k] + dis[k][j];//找到更短的路径
                }
            }
        }
    }
}
int main(){
    int n,m;//n为定点数，m为边数
    int u,v,w;//分别为边的起点，终点，权值
    fill(dis[0],dis[0] + MAXN * MAXN,INF);//初始化dis数组
    scanf("%d%d",&n,&m);//顶点数n，边数m
    for (int i = 0; i < m; ++i) {
        scanf("%d%d%d",&u,&v,&w);
        dis[u][v] = w;//以有向图输出为例
    }
    for (int i = 0; i < n; ++i) {//顶点i到顶点i的距离为0
        dis[i][i] = 0;
    }
    Floyd(n);
    for (int i = 0; i < n; ++i) {//输出dis数组
        for (int j = 0; j < n; ++j) {
            printf("%d ",dis[i][j]);
        }
        printf("\n");
    }
    return 0;
}

```

- TSP单源最短路径*
- 中国邮递员问题
- 选址问题

拓扑排序*

- 判断能否产生拓扑序列

```cpp
#include <iostream>
#include <cstdio>
#include <queue>
#include <cstring>
#include <vector>
using namespace std;
const int MAXN = 500;
int indegree[MAXN];//存储入度的数组
vector<int> graph[MAXN];//邻接表
bool TopologicalSort(int n){//拓扑排序
    queue<int> node;//此队列存放入度为0的点序号
    for (int i = 0; i < n; ++i) {//遍历n个点
        if (indegree[i] == 0) {//入度为0则入队
            node.push(i);
        }
    }
    int number = 0;//拓扑序列定点个数
    while (!node.empty()) {//如果队列中含有入度为0的元素
        int x = node.front();//保存队头元素并且出队
        node.pop();
        number++;//拓扑序列定点加1
        for (int i = 0; i < graph[x].size(); ++i) {
            int y = graph[x][i];
            indegree[y]--;//后继顶点入度减1
            if (indegree[y] == 0) {//新来个入度为0的点，入队
                node.push(y);
            }

        }
    }
    return number == n;//判断能否产生拓扑序列
}
int main(){
    int n,m;
    while (scanf("%d%d",&n,&m)!=EOF) {
        if (n == 0 && m == 0) {
            break;
        }
        memset(indegree, 0, sizeof(indegree));//初始化入度
        memset(graph, 0, sizeof(graph));//初始化图
        while (m--) {//建立邻接表
            int a, b;
            scanf("%d%d",&a,&b);
            graph[a].push_back(b);
            indegree[b]++;
        }
        if (TopologicalSort(n)) {
            printf("YES\n");
        }else{
            printf("NO\n");
        }
    }
    return 0;
}

```

- 产生拓扑序列

主要思想是引入priority_queue，每次出队都记录在中

```cpp
#include <iostream>
#include <cstdio>
#include <queue>
#include <cstring>
#include <vector>
using namespace std;
const int MAXN = 500;
int indegree[MAXN];//存储入度的数组
vector<int> graph[MAXN];//邻接表
vector<int> TopologicalSort(int n){//拓扑排序
    vector<int> Topo;
    priority_queue<int,vector<int>,greater<int>> node;//此队列存放入度为0的点序号,以编号小为优先的优先队列
    for (int i = 1; i <= n; ++i) {//遍历n个点x，序号从1~n
        if (indegree[i] == 0) {//入度为0则入队
            node.push(i);
        }
    }
    while (!node.empty()) {//如果队列中含有入度为0的元素
        int x = node.top();//保存队头元素并且出队,优先队列priority_queu的首元素是top ()
        node.pop();
        Topo.push_back(x);//加入拓扑序列
        for (int i = 0; i < graph[x].size(); ++i) {
            int y = graph[x][i];
            indegree[y]--;//后继顶点入度减1
            if (indegree[y] == 0) {//新来个入度为0的点，入队
                node.push(y);
            }

        }
    }
    return Topo;//判断能否产生拓扑序列
}
int main(){
    int n,m;
    while (scanf("%d%d",&n,&m)!=EOF) {
        memset(indegree, 0, sizeof(indegree));//初始化入度
        memset(graph, 0, sizeof(graph));//初始化图
        while (m--) {//建立邻接表
            int a, b;
            scanf("%d%d",&a,&b);
            graph[a].push_back(b);
            indegree[b]++;
        }
        vector<int> ans = TopologicalSort(n);//产生拓扑序列
        for (int i = 0; i < ans.size(); ++i) {
            if (i == 0) {
                printf("%d",ans[i]);
            }else{
                printf(" %d",ans[i]);
            }
        }
        printf("\n");
    }
    return 0;
}

```

关键路径

最早开始时间与最晚开始时间的计算

求关键路径长度：

- 求最早开始时间：根据拓扑序列逐一求出每个活动的最早开始时间
- 求最晚开始时间：根据拓扑序列的**逆序**求出每个活动的最晚开始时间
- 关键活动：所有活动中最早开始时间和最晚开始时间相同的活动为关键活动
- 关键路径长度：所有活动中最早开始时间的最大值便是关键路径长度

初始化：所有结点初始化为0，汇点的最晚开始时间初始化为它的最早开始时间，其余结点初始化INF

```cpp
#include <iostream>
#include <cstring>
#include <cstdio>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
const int INF = INT_FAST16_MAX;
const int MAXN = 1001;
struct Edge{
    int to;//终点
    int length;//长度
    Edge(int t,int l):to(t),length(l){}
};
vector<Edge> graph[MAXN];
int indegree[MAXN];
int earliest[MAXN];//最早开始时间
int latest[MAXN];//最晚开始时间
void CriticalPath(int n){
    vector<int> Topo;//拓扑排序
    queue<int> node;
    for (int i = 0; i < n; ++i) {//将入度为0的结点入队
        if (indegree[i] == 0) {
            node.push(i);
            earliest[i] = 1;//初始化最早开始活动时间为1
        }
    }
    while (!node.empty()) {//计算出拓扑序列并且算出最早开始时间earliest[]
        int x = node.front();//将队列首元素找出来
        Topo.push_back(x);
        node.pop();
        for (int i = 0; i < graph[x].size(); ++i) {//从队首元素开始，遍历其相连的每一条边
            int v = graph[x][i].to;
            int l = graph[x][i].length;
            earliest[v] = max(earliest[v], earliest[x] + l);//最早开始时间
            indegree[v]--;//相邻点的入度减1
            if (indegree[v] == 0) {//入度为0则入队
                node.push(v);
            }
        }
    }
        for (int i = Topo.size() - 1; i >= 0; --i) {
            int u = Topo[i];//拓扑序列的最后一个元素开始，从后往前推算
            if (graph[u].size() == 0) {//如果是汇点，则最晚时间与最早时间相同
                latest[u] = earliest[u];
            } else {//非汇点的最晚开始时间初始化
                latest[u] = INF;
```

```
            }
            for (int j = 0; j < graph[u].size(); ++j) {//从u结点的邻接边开始计算
                int v = graph[u][j].to;
                int l = graph[u][j].length;
                latest[u] = min(latest[u], latest[v] - l);
            }
        }
}
int main(){
    int n,m;
    while (scanf("%d%d",&n,&m)) {
        memset(graph, 0, sizeof(graph));
        memset(indegree, 0, sizeof(indegree));
        memset(earliest, 0, sizeof(earliest));
        memset(latest, 0, sizeof(latest));
        while (m--) {
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            graph[x].push_back(Edge(y,z));
            indegree[y]++;
        }
        CriticalPath(n);
        int ans = 0;
        for (int i = 0; i < n; ++i) {
            ans = max(ans,earliest[i]);
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

搜索

八皇后问题

```
#include <iostream>
#include <cmath>
using namespace std;
int N;
int queen[100];  // 用来存放算好的皇后的位置，最左上角是(0,0),最多有100个皇后
void NQueen(int k){ // 在0~k-1行皇后已经摆好的情况下，摆第k行及其后的皇后
    int i;
    if (k == N) {
        for (i = 0; i < N; i++) {
            cout << queen[i] + 1 << " ";
        }
        cout << endl;
        return;
    }
    for ( i = 0; i < N ; i++ ) {// i表示列
        int j;
```

```
        for ( j = 0; j < k; j++) {
            // 和已经摆好的k个皇后位置比较，看是否冲突
            if ( queen[j] == i // 避免同列，queen[j]存第j+1行皇后所在的列
                || abs(queen[j] - i) == abs(k - j)) { // 避免在一条斜线上,即 |queen[j] - i| !=|k-j|
                break;// 冲突，测试下一个位置
            }
        }
        if (j == k) {// 当前选的位置i不冲突
            queen[k] = i;// 将第k个皇后摆放在位置i
            NQueen(k+1);// 摆放后续的皇后
        }
    }// for ( i = 0; i < N ; i++ )
}
int main() {
    cin >> N;
    NQueen(0);   // 从第0行开始摆皇后
    return 0;
}
```

迷宫问题（广度优先搜索）

```
#include <cstdio>
#include <cstring>
using namespace std;
int map[6][6]; //地图
int vis[6][6]; //判重
int dir[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
//坐标优先级：往下走>往上走>往左走>往右走
//能往下走就往下走，如果下走遇到墙壁，往右走
struct node{
    int x,y; //坐标
    int c;   //上一点
}queue[6*6];
void print(int head) //输出坐标点,一开始的head值为终点坐标点的head,
{                                //它往上找的点，head指向的都是最短路径的坐标
    while(queue[head].c != -1)  //上一点存在时
    {
        print(queue[head].c);//先输出上一点坐标
        printf("(%d, %d)\n",queue[head].x,queue[head].y);//输出该点坐标
        return;
    }
    printf("(0, 0)\n");
}
void bfs(int x,int y)
{
    int head = 0,tail = 1;//一开始队列为空
    int i;
    queue[0].x = 0;
    queue[0].y = 0;
    queue[0].c = -1;
    struct node now;//当前点
    while(head < tail)
    {
        if(queue[head].x == 4 && queue[head].y == 4)
        {//到达终点，直接输出
            print(head);
            return;
        }
        for(i = 0;i < 4;i++)
        {//先确定当前的可能坐标
```

```
            now.x = queue[head].x + dir[i][0];
            now.y = queue[head].y + dir[i][1];
            now.c = head;//上一点为head
            if(now.x >= 0 && now.x <= 4 && now.y >= 0 && now.y <= 5)
            {// 当前点位置符合要求
                if(!vis[now.x][now.y] && !map[now.x][now.y])
                {//如果没有被访问过，并且地图上也是可以走的
                    vis[now.x][now.y] = 1;//置访问为为1,标记访问过，[0][0]没能有机会被标记
                    queue[tail] = now;//加入路径队列队尾
                    tail++;
                }
            }
        }
        head++;
    }
}
int main()
{
    int i,j;
    for(i = 0;i < 5;i++)
    {
        for(j = 0;j < 5;j++)
        {
            scanf("%d",&map[i][j]);
        }
    }
    memset(vis,0,sizeof(vis));
    bfs(0,0);
    return 0;
}
```

ROAD

N个城市，编号1到N。城市间有R条单向道路。
每条道路连接两个城市，有长度和过路费两个属性。
Bob只有K块钱，他想从城市1走到城市N。问最短共需要走多长的路。如果到不了N，输出-1
2<=N<=100
0<=K<=10000
1<=R<=10000
每条路的长度 L, 1 <= L <= 100
每条路的过路费T , 0 <= T <= 100

> 输入：
> K N R s1
> e1 L1 T1
> s1 e2 L2 T2
> ...
> sR eR LR TR
> s e是路起点和终点

```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;
int K,N,R;
struct Road {
    int d,L,t;
};
vector<vector<Road> > cityMap(110); //邻接表。 cityMap[i]是从点i有路连到的城市集合
int minLen = 1 << 30; //当前找到的最优路径的长度
```

```
int totalLen; //正在走的路径的长度
int totalCost ; //正在走的路径的花销
int visited[110]; //城市是否已经走过的标记
int minL[110][10100]; //minL[i][j]表示从1到i点的，花销为j的最短路的长度11
void Dfs(int s) //从 s开始向N行走
{
    if( s == N ) {
        minLen = min(minLen,totalLen);
        return ;
    }
    for( int i = 0 ;i < cityMap[s].size(); ++i ) {
        int d = cityMap[s][i].d; //s 有路连到d
        if(! visited[d] ) {
            int cost = totalCost + cityMap[s][i].t;
            if( cost > K)
                continue;
            if( totalLen + cityMap[s][i].L >= minLen ||
               totalLen + cityMap[s][i].L >= minL[d][cost])
                continue;
            totalLen += cityMap[s][i].L;
            totalCost += cityMap[s][i].t;
            minL[d][cost] = totalLen;
            visited[d] = 1;
            Dfs(d);
            visited[d] = 0;
            totalCost -= cityMap[s][i].t;
            totalLen -= cityMap[s][i].L;
        }
    }
}
int main()
{
    cin >>K >> N >> R;
    for( int i = 0;i < R; ++ i) {
        int s;
        Road r;
        cin >> s >> r.d >> r.L >> r.t;
        if( s != r.d )
            cityMap[s].push_back(r);
    }
    for( int i = 0;i < 110; ++i )
        for( int j = 0; j < 10100; ++ j )
            minL[i][j] = 1 << 30;
    memset(visited,0,sizeof(visited));
    totalLen = 0;
    totalCost = 0;
    visited[1] = 1;
    minLen = 1 << 30;
    Dfs(1);
    if( minLen < (1 << 30))
        cout << minLen << endl;
    else
        cout << "-1" << endl;
}
```

八数码*

在3×3的棋盘上，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘中留有一个空格，空格用0来表示。空格周围的棋子可以移到空格中。要求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单,设目标状态为123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

```cpp
#include<cstdio>
#include<cstring>
#include<set>
using namespace std;
typedef int State[9];//定义一个数组，专门存储九宫格的数字变成一维后的数组
const int MAXSTATE=1000000;
State st[MAXSTATE],goal={1,2,3,8,0,4,7,6,5};
int dist[MAXSTATE];
set<int> vis;//vis存储每一步的移动过程
void init_lookup_table() { vis.clear(); }//初始化
int try_to_insert(int s)
{//第s种方案是否满足要求
    int v=0;//将st中的第s中方案的数组组合变为数字
    for(int i=0;i<9;i++) v=v*10+st[s][i];
    if(vis.count(v)) return 0;//如果vis里面有v这个数字，则返回
    vis.insert(v);//如果没有v这种方案，则插入
    return 1;
}
const int dx[]={-1,1,0,0};
const int dy[]={0,0,-1,1};//dx,dy分别存储移动一格时的x轴y轴变化
int bfs()
{
    init_lookup_table();//初始化vis集合
    int front=1,rear=2;
    while(front<rear)
    {
        State& s=st[front];//s为st的当前状态
        if(memcmp(goal,s,sizeof(s))==0) return front;
        //对目标goal状态与当前s状态进行比较，如果相同则说明完成移动状态，返回front
        int z;//如果不相同，说明需要继续调整
        for(z=0;z<9;z++) if(!s[z]) break;//遍历当前状态，找到空格的坐标
        int x=z/3,y=z%3;//将空格坐标转化为3*3九宫格的x,y坐标
        for(int d=0;d<4;d++)
        {//移动新的位置，每个方向都试一试
            int newx=x+dx[d];
            int newy=y+dy[d];
            int newz=newx*3+newy;//空格移动到的新位置
            if(newx>=0 && newx<3 && newy>=0 && newy<3)
            {//如果移动的位置满足要求，则记录当前的状态到st[rear]
                State& t=st[rear];//利用指针改变st[rear]存储一维数字，改变的只是空格的移动位置关系
                memcpy(&t,&s,sizeof(s));
                //必须先复制front位置s到st[rear]上，之后的t指针的复制操作才有意义
                t[newz]=s[z];//新空格的位置为原来空格位置的值即   t[new] = 0;
                t[z]=s[newz];//新位置的原来空格位置是原来位置的数字
                dist[rear]=dist[front]+1;//rear代表前一个位置front改变到当前状态时的移动次数
                if(try_to_insert(rear)) rear++;//将当前可移动状态加入到set中
            }
        }
        front++;
    }
    return 0;
}
int main()
{
    char s[15];
    scanf("%s",s);
    for(int i=0;i<9;i++)
        st[1][i]=s[i]-'0';//存储初始状态
    //  for(int i=0;i<9;i++) printf(" %d ",st[1][i]);
    int ans = bfs();
    printf("%d\n", dist[ans]);
    return 0;
```

```
}

```

动态规划

凑硬币问题

问题描述

现有面值为*c*1,*c*2,...,*c***m*

c1,c2,...,cm元的m种硬币，求支付n元时所需硬币的最少枚数。各面值的硬币可重复使用任意次。

输入：
n　m
*c*1 *c*2 … cm
第1行输入整数n和整数m，用1个空格隔开。第2行输入各硬币的面值，相邻面值间用1个空格隔开。
输出：
输出所需硬币的最少枚数，占1行。
限制：
1 ⩽ n ⩽ 50000
1 ⩽ m ⩽ 20
1 ⩽ 面值 ⩽ 10000
各面值均不相同，其中必包含1

解法：

C[m]：一维数组，C[i]表示第 i 种硬币的面值
T[j]：一维数组，T[j]表示使用第0至第 i 种硬币支付 j 元时的最少硬币枚数。

支付 j 元时的最少枚数可以作为一维数组元素T[j]，由下面的式子求得：

T[j]=min(T[j],T[j−C[i]]+1)

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int MMAX = 20;
const int NMAX = 50000;
const int INFTY = (1<<29);
int main(){
    int n, m;//n为总找钱值，m为硬币种类数目
    int w[MMAX + 1];//C存储着硬币的面值
    int dp[NMAX + 1];//dp[j]表示找钱价值为j时，所需要的最少纸币数目
    cin>>n>>m;
    for(int i = 1; i <= m; i++) {
        cin>>w[i];
    }
    for(int i = 0; i <= NMAX; i++) dp[i] = INFTY;//初始化
    dp[0] = 0;
    for(int i = 1; i <= m; i++) {
        for(int j = w[i]; j <= n; j++) {
//可以看成是总容量为找零总额，物品重量为硬币面值，其物品价值为1 的 0-1背包问题
            dp[j] = min(dp[j], dp[j-w[i]] + 1);
        }
    }
    cout<<dp[n]<<endl;
    return 0;

```

```
}

```

0–1背包问题

有 *N* 种物品和一个容量是 *V *的背包，每种物品只有**一件可用**。

第 i 种物品的体积是 v[i]，价值是 w[i]

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
int main()
{
    int N,M;// N为物品个数，M为背包总容量
    scanf("%d%d",&N,&M);
    int i,j;
    int dp[15000]={0};// dp[x]存储容量为x时的最大价值
    int w[5000],v[5000];// 第i件物品的体积w[i]，价值是v[i]
    for (i=0;i<=N-1;i++)
        scanf("%d%d",&w[i],&v[i]);// 输入每件物品的体积、价值
    for (i=0;i<=N-1;i++)
    {
       for (j=M;j>=w[i];j--)// 往背包放东西,剩余空间大于所选择的物品才可以放
        {
            dp[j] = max(dp[j], dp[j-w[i]] + v[i]);
//   如果放入背包后的价值大于当前容积的价值，则放入背包，更新dp[j]
        }
    }
    printf("%d\n",dp[M]);
    return 0;
}
```

完全背包问题

有 *N* 种物品和一个容量是 *V *的背包，每种物品都有**无限件可用**。

第 i 种物品的体积是 v[i]，价值是 *w[i]*

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
int main()
{
    int N,M;// N为物品个数，M为背包总容量
    scanf("%d%d",&N,&M);
    int i,j;
    int dp[15000]={0};// dp[x]存储容量为x时的最大价值
    int w[5000],v[5000];// 第i件物品的体积w[i]，价值是v[i]
    for (i=0;i<=N-1;i++)
        scanf("%d%d",&w[i],&v[i]);// 输入每件物品的体积、价值
    for (i=0;i<=N-1;i++)
    {
```

```
        for (j=w[i];j<=M;j++)// 往背包放东西,剩余空间大于所选择的物品才可以放
         {// 从选择的物品的体积开始选择物品（从前往后）
             dp[j] = max(dp[j], dp[j-w[i]] + v[i]);
//   如果放入背包后的价值大于当前容积的价值，则放入背包，更新dp[j]
         }
    }
    printf("%d\n",dp[M]);
    return 0;
}
```

多重背包

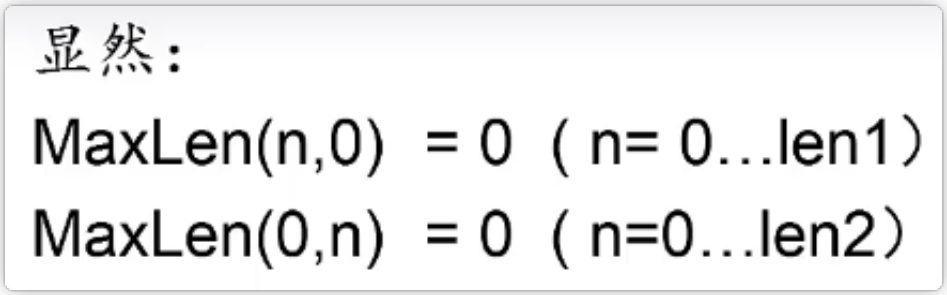
总体思路：多重背包主要把它转化成0-1背包问题，同一个种类的物品有多个，那就在数组里面重复存储就好，其他和0-1背包一样，也是从后往前推

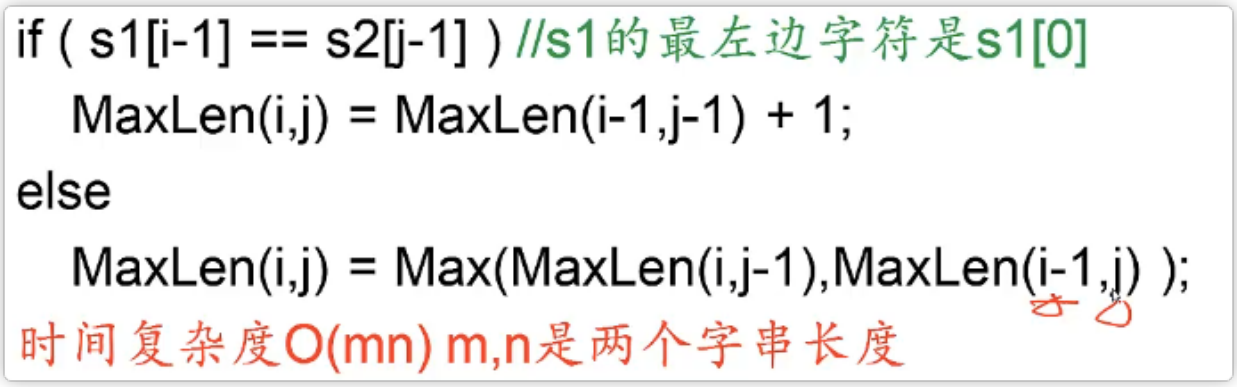
```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAXN = 10000;
int dp[MAXN];
int v[MAXN];
int w[MAXN];
int k[MAXN];
int weight[MAXN];
int value[MAXN];
int main(){
    int CaseNumber;
    scanf("%d",&CaseNumber);
    while (CaseNumber--) {
        int n,m;
        scanf("%d%d",&m,&n);
        int number = 0;
        for (int i = 0; i < n; ++i) {
            scanf("%d%d%d",&w[i],&v[i],&k[i]);
            for (int j = 1; j <= k[i]; ++j) {
                weight[number] = w[i];
                value[number] = v[i];
                number++;
            }
        }
        for (int i = 0; i <= m; ++i) {
            dp[i] = 0;
        }
        for (int i = 0; i < number; ++i) {
            for (int j = m; j >= weight[i] ; --j) {
                dp[j] = max(dp[j], dp[j - weight[i]]+value[i]);
            }
        }
        printf("%d\n",dp[m]);
    }
    return 0;
}

```

最长上升子序列

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int MAXN = 1010;
int a[MAXN];
int maxLen[MAXN];
int ans = -1;
int main()
{
    int N;// 序列的长度
    cin >> N;
    for (int i = 1; i <= N; ++i) {
        cin >> a[i];
        maxLen[i] = 1;
    }// 输入序列
    for (int i = 2; i <= N; ++i) {
    // 每次求以第i个数为终点的最长上升子序列
        for (int j = 1; j < i; ++j){
    // 查看以第j个数为终点的最长上升子序列
            if (a[i] > a[j]){// 若终点i比之前的最长序列的终点大
                maxLen[i] = max(maxLen[i], maxLen[j]+1);// 则最长序列+1
            }
        }
    }
    for(int i = 1; i <= N; i++)
        ans = max(ans, maxLen[i]);
    printf("%d\n",ans);
    //cout << * max_element(maxLen, maxLen + N + 1);
    //max_element是用来来查询最大值所在的第一个位置。
    return 0;
}
```

最长公共子序列






```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
char sz1[1000];
char sz2[1000];
int maxLen[1000][1000];
int main()
{
    while (cin >> sz1 >> sz2) {
        int length1 = strlen(sz1);
        int length2 = strlen(sz2);
        int i,j;// 设置边界条件
        for (i = 0; i <= length1; ++i) {
            maxLen[i][0] = 0;
        }
        for (j = 0; j <= length2; ++j) {
            maxLen[0][j] = 0;
        }
        for (i = 1; i <= length1; ++i) {
            for (j = 1; j <= length2; ++j) {
                if (sz1[i-1] == sz2[j-1]) {
                    maxLen[i][j] = maxLen[i-1][j-1] + 1;
                }else{
                       maxLen[i][j] = max(maxLen[i][j-1],
                                          maxLen[i-1][j]);
                    }
            }
        }
        cout << maxLen[length1][length2] << endl;
    }
    return 0;
}
```

最大连续子序列和

- 直接找最大

```cpp
#include<stdio.h>
#include<algorithm>
using namespace std;
int main(){
    int N,i;
    while(scanf("%d",&N)!=EOF){
```

```
        long sum=0,Max=-999999999,x;
        for(i=0;i<N;i++){
            scanf("%ld",&x);
            sum=max(sum+x,x);
            Max=max(Max,sum);
        }
        printf("%ld\n",Max);
    }
}
```

- 简单动态规划

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
int dp[1000010];
int a[1000010];
long long maxx;
int main()
{
    int n ;
    while(cin>>n){
        for(int i=0;i<n;i++){
            cin>>a[i];
        }
        dp[0] = a[0];
        maxx = 0;
        for(int i=1;i<n;i++){
            dp[i] = max(dp[i-1]+a[i],a[i]);
            if(maxx<dp[i]){
                maxx = dp[i];
            }
        }
        cout<<maxx<<endl;
    }
   return 0;
}
```

最大上升子序列和

动态规划求解
开始假设所有的数自成最大递增子序列，也就是sum[i]=num[i];
后面再从前向后遍历，如果前面某个数小于当前的数，那么那个数的最大递增子序列的和加上当前的数会构成更大的递增子序列和，找出所有这样的和的最大值作为以当前数为尾的最大递增子序列的和。

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    int num[1001];
    int sum[1001];
    while(cin>>n)
    {
        int  maxsum = 0;
```

```
        for(int i = 0; i<n; i++)
        {
            cin>>num[i];
            sum[i] = num[i]; //初始化，所有数自成一个最大递增子序列
        }
        sum[0] = num[0];
        for(int i = 1; i<n; i++)
        {
            for(int j = i-1; j>=0; j--)
            {
                if(num[j]<num[i])   //符合递增子序列
                    sum[i] = max(sum[j]+num[i], sum[i]);
            }
        }
        maxsum = sum[0];
        for(int i = 1; i<n; i++)
        {
            if(maxsum<sum[i]) maxsum = sum[i];
        }
        cout<<maxsum<<endl;
    }
    return 0;
}
```

拦截导弹

```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAXN = 25;
int height[MAXN];   //导弹高度
int dp[MAXN];
int main(){
    int n;
    while (scanf("%d",&n)!=EOF) {
        for (int i = 0; i < n; ++i) {
            scanf("%d",&height[i]);
        }
        int answer = 0;
        for (int i = 0; i < n; ++i) {
            dp[i] = 1;  //初始化为1
            for (int j = 0; j < i ; ++j) {
                if (height[i] <= height[j]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            answer = max(answer, dp[i]);    //dp数组最大值
        }
        printf("%d\n",answer);
    }
    return 0;
}
```

最大子矩阵

```
#include <stdio.h>
#define N 100
```

```
#define INF 1E9
int martix[N][N];//存储矩阵
int buf[N];//将相邻若干行合并成一行以后的结果
int n, max;//n是矩阵大小，max是最大矩阵和

void FindMax(int from, int m)
{//从from行开始的连续m行合并，得到其最大连续序列和
    for(int i=0; i<n; i++) buf[i]=0;
    for(int j=0; j<n; j++)
    {
        for(int i=0; i<m; i++)
        {
            buf[j]+=martix[from+i][j];
        }
    }
    int sum=0;
    int maxHere=buf[0];
    for(int i=0; i<n; i++)
    {
        sum+=buf[i];
        if(sum>maxHere) maxHere=sum;
        if(sum<0) sum=0;//和为负，则全部丢弃，从下一个开始新的序列
    }
    if(maxHere>max) max=maxHere;//有必要的话修改max值
    return;
}

int main()
{
    while(scanf("%d", &n)!=EOF)
    {
        if(n<=0) break;
        max=-INF;
        for(int i=0; i<n; i++)
        {//读取矩阵
            for(int j=0; j<n; j++)
            {
                scanf("%d", &martix[i][j]);
            }
        }
        for(int m=1; m<=n; m++)
        {//m是要合并连续的几行
            for(int from=0; from+m-1<n; from++)
            {//从from行开始合并连续的m行，并修改最大值
                FindMax(from, m);
            }
        }
        printf("%d\n", max);//输出结果
    }
}
```

数塔问题

```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAXN = 100;
int dp[MAXN][MAXN];
int matrix[MAXN][MAXN];
int main(){
```

```
    int n;
    scanf("%d",&n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= i ; ++j) {
            scanf("%d",&matrix[i][j]);
            dp[i][j] = matrix[i][j];
        }
    }//dp[i][j]i表示(i,j)出发到底部路径所有值之和的最大值，dp[0][0]是问题答案
//dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + dp[i][j]
//(i,j)的值为下方路径与右下方路径最大值加上(i,j)自身的权值
    for (int i = n - 1; i >= 0; --i) {
        for (int j = 0; j <= i; ++j) {
            dp[i][j] += max(dp[i+1][j], dp[i+1][j+1]);
        }
    }
    cout << dp[0][0];
    return 0;
}

```

神奇的口袋

有一个神奇的口袋，总的容积是40，用这个口袋可以变出一些物品，这些物品的总体积必须是40。

John现在有n（1≤n ≤ 20）个想要得到的物品，每个物品的体积分别是a1， a2……an。 John可以从这些物品中选择一些，如果选出的物体的总体积是40，那么利用这个神奇的口袋， John就可以得到这些物品。现在的问题是， John有多少种不同的选择物品的方式。

输入
输入的第一行是正整数n (1 <= n <= 20)，表示不同的物品的数目。接下来的n行，每行有一个1到40之间的正整数，分别给出a1， a2……an的值。

输出
输出不同的选择物品的方式的数目

```
#include <iostream>
#include <cstdio>
using namespace std;
const int MAXN = 100;
int dp[MAXN][MAXN];//dp[i][j]表示体积为i时，用了前面j种物品时最多的方案
int a[MAXN];
int main(){
    int n;
    scanf("%d",&n);
    memset(dp, 0, sizeof(dp));
    for (int i = 1; i <= n; ++i) {//物品从a[1]开始放，a[i]表示第i件物品的体积
        scanf("%d",&a[i]);
        dp[0][i] = 1;//总体积为0的放法只有一种
    }
    dp[0][0] = 1;//补上漏掉的dp[0][0]
    for (int i = 1; i <= 40 ; ++i) {
        for (int j = 1; j <= n; ++j) {
            dp[i][j] = dp[i][j-1];//体积为i，用前j种方案时，至少有体积为i用前j-1种方案的次数
            if (i - a[j] >= 0) {
//如果第j件物品可以选下来（选择它体积不会溢出），则再加上没选上这件物品时的体积且用前j-1种方案的数量
                dp[i][j] += dp[i - a[j]][j-1];
            }
        }
    }
    cout << dp[40][n];

```

```
    return 0;
}
```