算法

题型	题数	分值
选择	10	30
分析（算法思想，区别）	3	20
程序(程序选择，读程序，复杂度，算法)	7	30
综合设计（简答，过程填空）	2	20

递归

递归模型

递归模型是递归算法的抽象,它反映一个递归问题的递归结构,例如,前面的递归算法对应的递归模型：

- fun(1)=1 (1)
- fun(n)=n*fun(n-1) n>1 (2)

其中,第一个式子给出了递归的终止条件,第二个式子给出了fun(n)的值与fun(n-1)的值之间的关系,我们把第一个式子称为递归边界,把第二个式子称为递归体。

优点：

- 递归过程结构清晰
- 程序易读

缺点：

- 时间效率低
- 空间开销大，问题规模扩大时，噩梦来临。
- 算法不容易优化

对于频繁使用的算法，或不具备递归功能的程序设计语言，需要把递归算法转换为非递归算法。

二分查找(折半查找)

```
int bsearch(int b[], int x, int L, int R)
{
    int mid;

    if(L > R) return(-1);
    mid = (L + R)/2;
    if(x == b[mid])
        return mid;
    else if(x < b[mid])
```

```
        return bsearch(b, x, L, mid-1);
    else
        return bsearch(b, x, mid+1, R);
```

阶乘函数

```
void   main( )
{
        int   n;
        printf("请输入一个整数：");
        scanf("%d",   &n);
        printf("%d!  =  %d \n",   n,   fact(n));
}
int   fact(int  n)
{
        if(n  ==  1)    return(1);
        else    return(n * fact(n-1));
}
```

斐波那契序列O(2^n)

```
long Fib(int n)
{        if(n == 0 || n == 1) return n; //递归出口
         else return Fib(n-1) + Fib(n-2); //递归调用
}
```

汉诺塔问题O(2^n)

```
#include <stdio.h>
int cnt;
void hanoi(int n, char a, char b, char c) {
    if (n == 0)
        return;
    hanoi(n - 1, a, c, b);//将n-1个盘子由A经过C移动到B
    printf ("step %d: move %d from %c->%c\n", cnt++, n, a, c);
    hanoi(n - 1, b, a, c);//剩下的n-1盘子，由B经过A移动到C
}

int main() {
    int n;
    while( scanf ("%d", &n)) {
        cnt=1;
        hanoi(n, 'A', 'B', 'C');
    }
    return 0;
}
```

寻找线性表最大元素

```
#include <stdio.h>
int Max(int L[],unt i,int j){
int mid,max,max1,max2;
if(i==j)
   max=L[i];
else{
mid=(i+j)/2;
```

```
max1=Max(L,i,mid);
max2=Max(L,mid+1,j);
max=(max1>max2)?max1:max2;//取大的
}return max;
}
```

分治

分治算法分三步

1. 首先，它将一个复杂的问题分成若干个简单的子问题并进行解决，这一步被称为分割（divide）.
2. 然后解决每一个子问题，这一部被称为征服或解决(conquer)，也就是分治这个词中“治”的来源。在这一步中，如果子问题非常简单，就直接解决了。如果子问题依然很大，那么还需要递归调用分治算法，把子问题分成更小一级的问题来解决，直到那些被分出的子问题能够直接解决为止。
3. 最后对子问题的结果进行合并(combine)，得到原有问题的解。

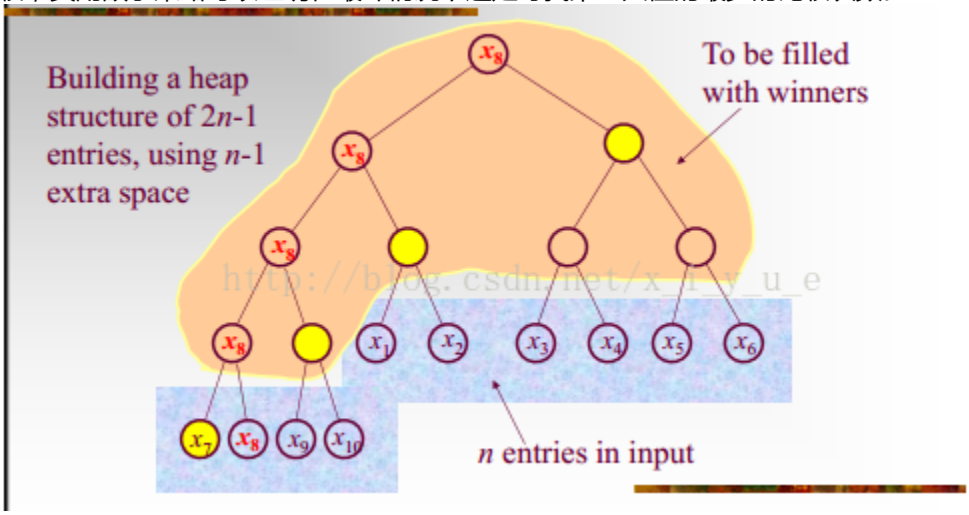
二分搜索

```
BinarySearch(int [] a, int x, int n)
   {
   // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
   // 找到x返回其在数组中的位置,否则返回-1
     int left = 0; int right = n - 1;
     while (left <= right) {
       int middle = (left + right)/2;
       if (x == a[middle]) return middle;
       if (x > a[middle]) left = middle + 1;
       else right = middle - 1;
      }
     return -1; // 未找到x
   }
```

锦标赛算法

使用锦标赛算法，设置一个额外的大小为2n的数组(在优化的算法中额外数组的大小为n就可以)b，首先将原数组的n个数复制到额外数组的后n个位置，将额外数组b理解为一个类似于最大堆的结构，然后从倒数第二位置开始，以此选择两个数的较大者放到父亲节点中，依次进行，直到到根节点（b[1]），那么b[1]一定是这n个数中的最大值，第二大值一定同最大值比较过且输给了最大值，因此找第二大值时只需要从直接输给最大值的元素中去找，直接输给最大值的元素的个数有“lgn

的上界”个；找最大值需要n-1次比较，找第二大值需要“lgn的上界”-1次比较，所以一共需要n+“lgn的上界”-2次比较，实用敌手策略可以证明在最坏情况下这是寻找第二大值的最少的比较次数。

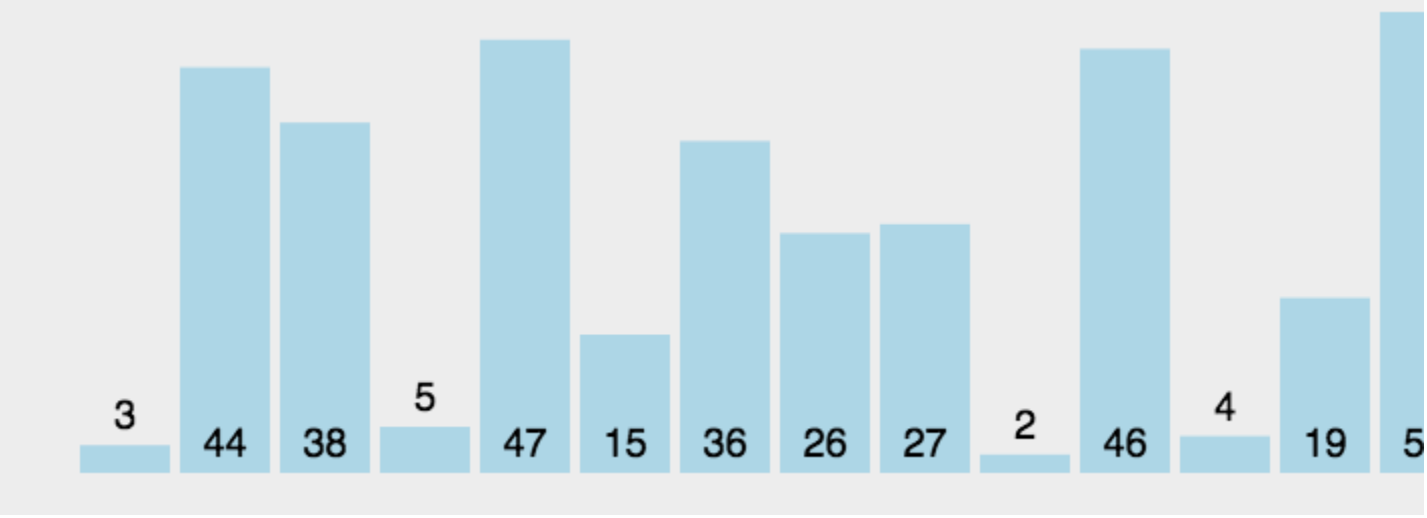



```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        int a[]=new int[11];
        Random random=new Random();
        for(int i=0;i<a.length;i++)
            a[i]=random.nextInt(101);  //随机生成10个0到100的数字
        System.out.println("生成的结果是："+Arrays.toString(a));
        int secondmaxi = FindsecondMax2(a);
        System.out.println("第二大的数是："+secondmaxi);
    }

//锦标赛算法
    public static int ChampionShip(int a[]) {
        int temp[]=new int[2*a.length];
        int secondmax=-0x3f3f3f3f;

        //将数组里面的数据填在临时数组中的temp[a.length-1]的后面
        for(int i=temp.length-1;i>=a.length;i--)
            temp[i]=a[i-a.length];

        //从后往前进行比较，将最大值以此填在temp[1]到temp[a.length-1]之间，则temp[1]是数组中的最大值
        for(int i=temp.length-2;i>=2;i-=2)
            temp[i/2]=Math.max(temp[i], temp[i+1]);

        //我们将从直接输给最大值的数中找到第二大的数
        for(int i=1;i<a.length;) {
            if(temp[i]==temp[2*i]) {
                if(secondmax<temp[2*i+1])
                    secondmax=temp[2*i+1];
                i=i*2;
            }


            else {
                if(secondmax<temp[2*i])
                    secondmax=temp[2*i];
                i=2*i+1;
            }
        }
        return secondmax;
    }
```

排序算法

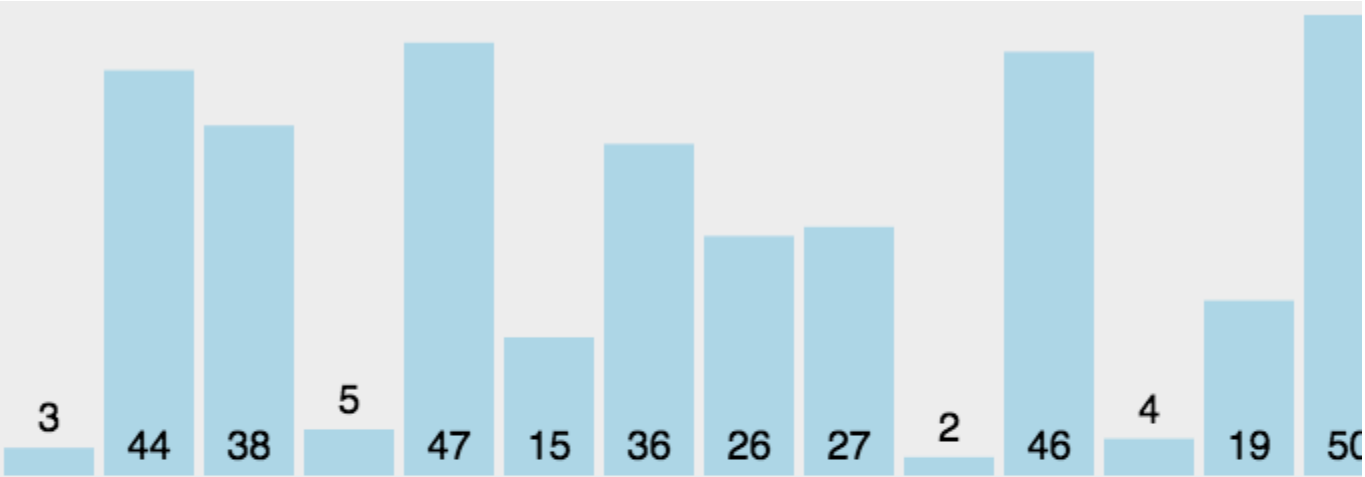
排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	

冒泡排序$O(n^2)$



```
//按照刚才那个动图进行对应
//冒泡排序两两比较的元素是没有被排序过的元素--->
public void bubbleSort(int[] array){
    for(int i=0;i<array.length-1;i++){//控制比较轮次，一共 n-1 趟
        for(int j=0;j<array.length-1-i;j++){//控制两个挨着的元素进行比较
            if(array[j] > array[j+1]){
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
}
```

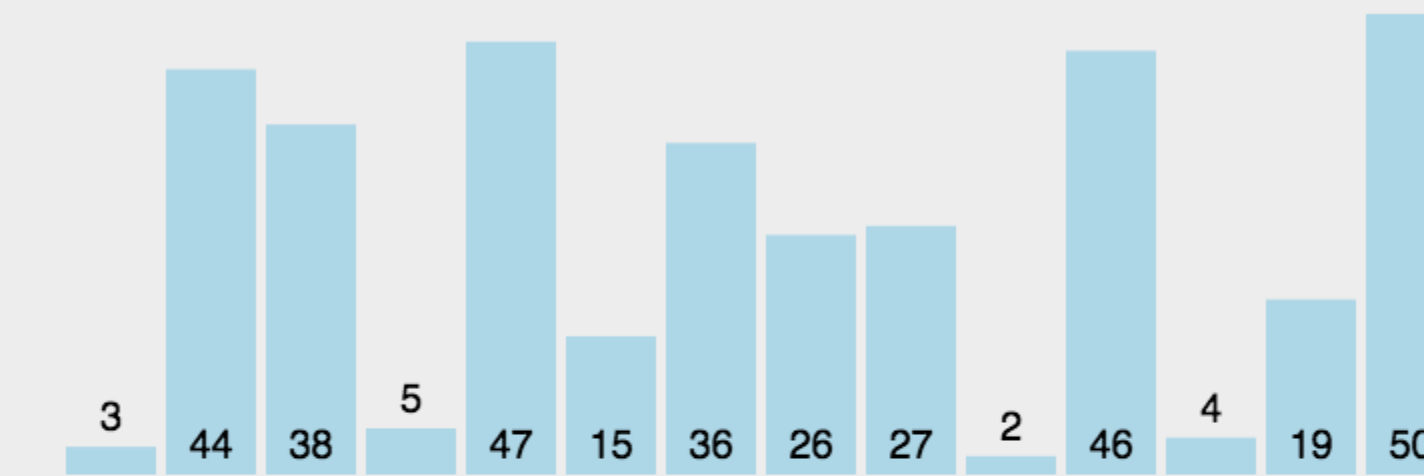
插入排序O(n^2)




```
#include <bits/stdc++.h>
#define N 1550
using namespace std;
int a[N], n;

int main() {
    // 输入
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
```

```
    // 插入排序
    for (int i = 2; i <= n; ++i) {    // 按照第2个到第n个的顺序依次插入
        int j, x = a[i];    // 先将i号元素用临时变量保存防止被修改。

        // 插入过程，目的是空出分界线位置j，使得所有<j的部分<=x，所有>j的部分>x。
        // 循环维持条件，j>1，并且j前面的元素>x。
        for (j = i; j > 1 && a[j - 1] > x; --j) {
            // 满足循环条件，相当于分界线应向前移，
            // 分界线向前移，就等于将分界线前面>x的元素向后移
            a[j] = a[j - 1];

        }
        // 找到分界线位置，插入待插入元素x
        a[j] = x;
    }

    // 输出
    for (int i = 1; i <= n; ++i) cout << a[i] << ' ';
    cout << endl;
    return 0;
}
```

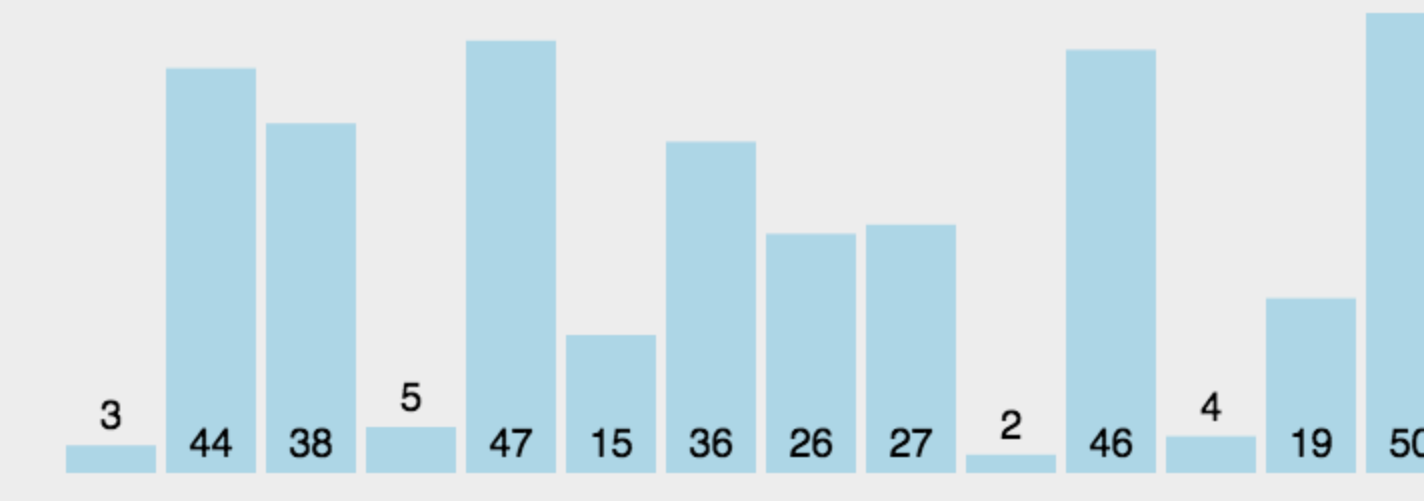
归并排序（递归）O(nlogn)




```
#include <iostream>
#include <cstdio>

using namespace std;
```

```
const int N = 1e5 + 5;
int a[N], temp[N];

void merge_Sort(int q[], int l, int r) {
    if (l >= r) return;
    int mid = (l + r) >> 1;
    merge_Sort(q, l, mid);
    merge_Sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;

    //两组数比较，小的先暂存temp
    while (i <= mid && j <= r) {
        if (q[i] <= q[j]) {
            temp[k++] = q[i++];
        } else {
            temp[k++] = q[j++];
        }
    }
    /*
    3 4(1) 5 8
    1 2 4(2) 5
    1 2 3 4(1) 4(2) 5(1) 5(2)
    */
    //当一组数全部排入temp后，剩下的一次加入temp中
    while (i <= mid) {
        temp[k++] = q[i++];
    }
    //1 2 3 4(1) 4(2) 5(1) 5(2) 8
    while (j <= r) {
        temp[k++] = q[j++];
    }

    //把temp中排好序的数据重新存入q[]
    for(i=l,j=0;i<=r;i++,j++){
        q[i]=temp[j];
    }

}

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    merge_Sort(a, 0, n - 1);

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

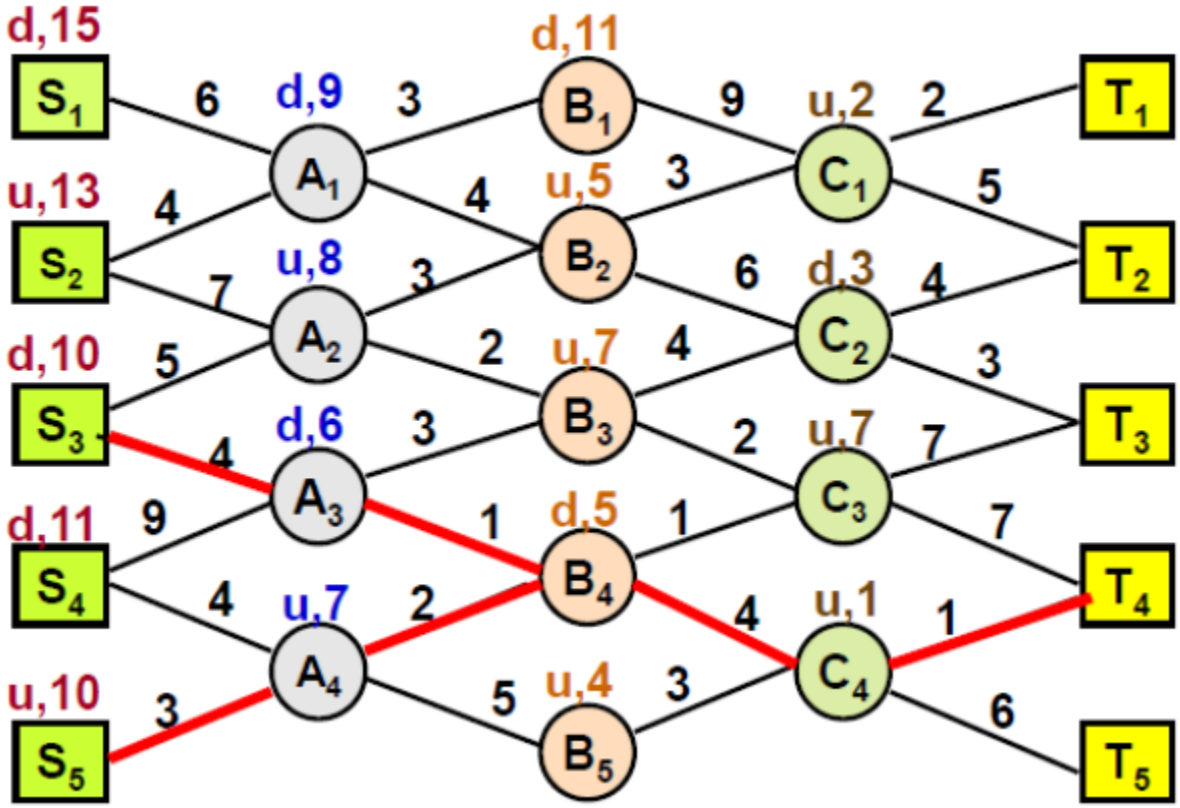
快速排序O(nlogn)




```cpp
#include <iostream>
#include <cstdio>

using namespace std;

const int N = 1e6+10;

int q[N];
int n;

void quick_sort(int q[],int l,int r){

    if(l>=r)return;//没有数或者只有一个数，则不用排序

    int x=q[l],i=l-1,j=r+1;

    while(i<j){
        do i++; while(q[i]<x);
        do j--; while(q[j]>x);
        if(i<j){
            swap(q[i],q[j]);
        }
    }

    quick_sort(q,l,j);
    quick_sort(q,j+1,r);

}

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

    for(int i=0;i<n;i++)
        scanf("%d",&q[i]);

    //快排
    quick_sort(q,0,n-1);

    for(int i=0;i<n;i++)
        printf("%d ",q[i]);

    return 0;

}
```

动态规划

动态规划技术把求解过程变成一个多步判断的过程，每一步都对应于某个子问题。算法细心地划分子问题的边界，从小的子问题开始，逐层向上求解。通过子问题之间的依赖关系，有效利用前面已经得到的结果，最大限度减少重复工作，以提高算法效率。



矩阵连乘问题

设计算A[i:j]（矩阵A从i乘到j），1≤i≤j≤n，所需要的最少数乘次数m[i,j]，则原问题的最优值m[1,n] 当i=j时，A[i:j]=Ai，因此，m[i][i]=0，i=1,2,...,n 当i<j时，若A[i:j]的最优次序在Ak和Ak+1之间断开，i<=k<j,则：m[i][j]=m[i][k]+m[k+1][j]+pi1pkpj。由于在计算是并不知道断开点k的位置，所以k还未定。不过k的位置只有j-i个可能。因此，k是这j-i个位置使计算量达到最小的那个位置。

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

例如，在计算 m[2][5] 时，依递归式有

$$m[2][5]=\min\begin{cases}m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+\\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\end{cases}$$

$$=7125$$

且 $k=3$，因此，s[2][5] = 3。

```
#include<iostream>
using namespace std;

#define N 7  //N为7，实际表示有6个矩阵
/*
*矩阵链构造函数：构造m[][]和s[][]
*m中存储的值是计算出来的最小乘法次数，比如m[1][5]就是A1A2A3A4A5的最小乘法次数
*s中存储的是获取最小乘法次数时的断链点，s[1][5]对应的就是如何拆分A1A2A3A4A5,
*比如S[1][5]=3可表示：(A1A2A3)(A4A5)，当然内部断链还会继续划分A1A2A3
*/
int MatrixChain(int *p, int n, int m[][N], int s[][N]){
    for(int i=1;i<=n;i++){     //矩阵链中只有一个矩阵时，次数为0，注意m[0][X]时未使用的
        m[i][i]=0;
    }
    for(int r=2;r <= n;r++){    //矩阵链长度,从长度为2开始
        for(int i=1;i <= n-r+1;i++){    //根据链长度，控制链最大的可起始点
            int j = i+(r-1);  //矩阵链的末尾矩阵，注意r-1，因为矩阵链为2时，实际是往右+1
            m[i][j] = m[i][i]+m[i+1][j]+p[i-1]*p[i]*p[j];   //先设置最好的划分方法就是直接右边开刀，后续改正，也可合并到
            s[i][j]=i;
            for(int k=i+1;k < j;k++){   //这里面将断链点从i+1开始，可以断链的点直到j-1为止
                int t = m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
                if(t<m[i][j]){
                    m[i][j] = t;
                    s[i][j] = k;
                }

            }

        }
    }
}
/*
*追踪函数：根据输入的i,j限定需要获取的矩阵链的始末位置，s存储断链点
*/
void Traceback(int i,int j, int s[][N]){
    if(i==j)       //回归条件
    {
        cout<<"A"<<i;
    }
    else       //按照最佳断点一分为二，接着继续递归
```

```
    {
        cout<<"(";
        Traceback(i,s[i][j],s);
        Traceback(s[i][j]+1,j,s);
        cout<<")";
    }
}
int main(){
    int p[N]={30,35,15,5,10,20,25};
    int m[N][N],s[N][N];

    MatrixChain(p,N-1,m,s);//N-1因为只有六个矩阵
    Traceback(1,6,s);
    return 0;
}
```

投资问题

```
#include <iostream>
#include <math.h>
using namespace std;
const int M = 5;
const int N = 6;

int maxprofit(int dp[M][N],int f[M][N],int money,int number)
{//i表示投资项目的序号，j表示投资了多少钱
    for (int i=1;i<=number;i++)
    {
        for (int j=0;j<=money;j++)
        {
            dp[i][j]=0;//全部初始化为0
            for (int k=0;k<=j;k++)
            {
                if (dp[i][j]<f[i][k]+dp[i-1][j-k])
                    dp[i][j]=f[i][k]+dp[i-1][j-k];
            }
        }
    }
    return dp[number][money];
}

int main()
{
    int dp[M][N]={0};
    int f[M][N] = {0,0,0,0,0,0,
                   0,11,12,13,14,15,
                   0,0,5,10,15,20,
                   0,2,10,30,32,40,
                   0,20,21,22,23,24};
    cout<<"最大利益为："<<maxprofit(dp,f,5,4)<<endl;
    return 0;
}
```

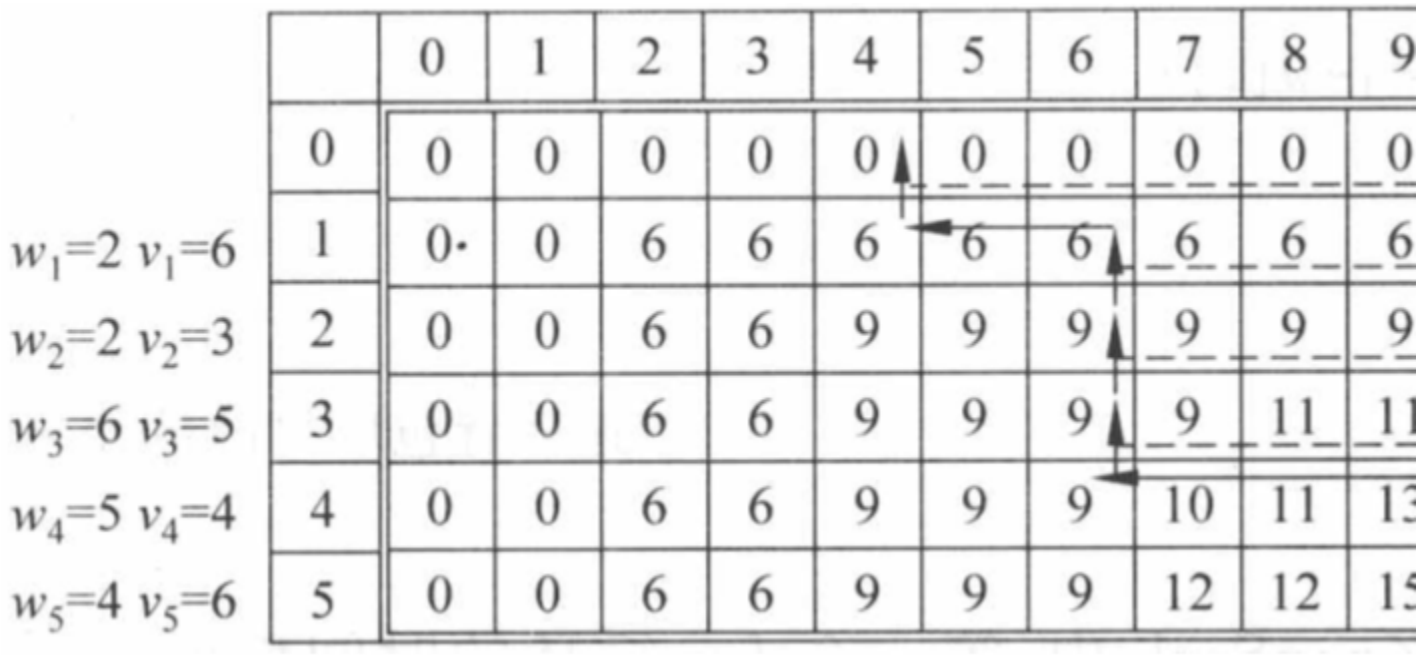
0-1背包问题

有n个物品，它们有各自的体积和价值，现有给定容量的背包，如何让背包里装入的物品具有最大的价值总和？

为方便讲解和理解，下面讲述的例子均先用具体的数字代入，即：eg：number = 5，capacity = 10

i（物品编号）	1	2	3	4	5
w（体积）	2	2	6	5	4
v（价值）	6	3	5	4	6

- 包的容量比该商品体积小，装不下，此时的价值与前i-1个的价值是一样的，即V(i,j)=V(i-1,j)；
- 还有足够的容量可以装该商品，但装了也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即V(i,j) =max { V(i-1,j)，V(i-1,j-w(i))+v(i) } 。



		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
w_1=2 v_1=6	1	0	0	6	6	6	6	6	6	6	6
w_2=2 v_2=3	2	0	0	6	6	9	9	9	9	9	9
w_3=6 v_3=5	3	0	0	6	6	9	9	9	9	11	11
w_4=5 v_4=4	4	0	0	6	6	9	9	9	10	11	13
w_5=4 v_5=6	5	0	0	6	6	9	9	9	12	12	15

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

int w[5] = { 0 , 2 , 3 , 4 , 5 };           //商品的体积2、3、4、5
int v[5] = { 0 , 3 , 4 , 5 , 6 };           //商品的价值3、4、5、6
int bagV = 8;                              //背包大小
int dp[5][9] = { { 0 } };                   //动态规划表
int item[5];                               //最优解情况

void findMax() {                        //动态规划
    for (int i = 1; i <= 4; i++) {
        for (int j = 1; j <= bagV; j++) {
            if (j < w[i])
                dp[i][j] = dp[i - 1][j];
//背包容量不足以装人物品i,则装入前i个物品得到的最大价值和装入前i一1个物品得到的最大价值是相同的，即xi=0，背包不增加价值。
            else
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);
//背包容量可以装入物品i，如果把第i个物品装入背包,则背包中物品的价值等于把前i-1个物品装入容量为j-wi的背包中的价值加上第i个物品
        }
    }
}

void findWhat(int i, int j) {//最优解情况，i表示物品编号，j表示背包剩余重量
    if (i >= 0) {
```

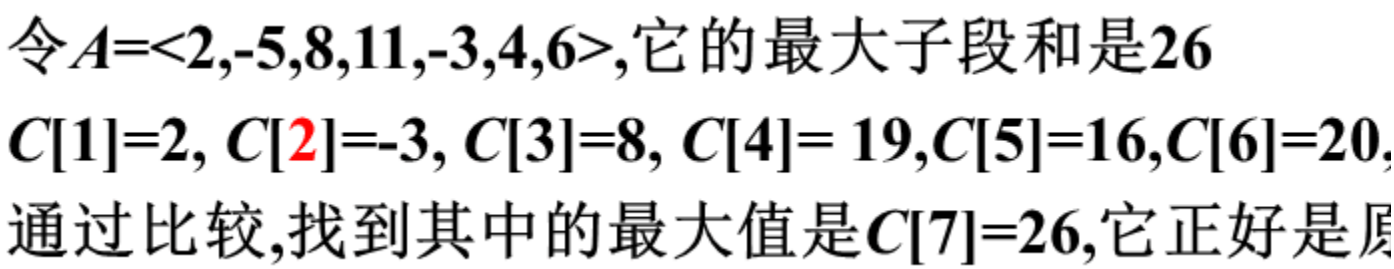
```
        if (dp[i][j] == dp[i - 1][j])//价值未变说明没有放入背包 {
            item[i] = 0;//未装入标记为0
            findWhat(i - 1, j);//回溯到上一个物品，并且背包重量不变
        }
        else if (j - w[i] >= 0 && dp[i][j] == dp[i - 1][j - w[i]] + v[i])//背包空间足够并且价值能对应 {
            item[i] = 1;//装入标记为1
            findWhat(i - 1, j - w[i]);//回溯到上一个物品，并且背包重量发生变化
        }
    }
}

void print() {
    for (int i = 0; i < 5; i++) {           //动态规划表输出
        for (int j = 0; j < 9; j++) {
            cout << dp[i][j] << ' ';
        }
        cout << endl;
    }
    cout << endl;

    for (int i = 0; i < 5; i++)         //最优解输出
        cout << item[i] << ' ';
    cout << endl;
}

int main()
{
    findMax();
    findWhat(4, 8);
    print();

    return 0;
}
```

最长公共子序列 LCS

```
#include<iostream>
#include<string.h>
using namespace std;
void  Structure Sequence(int S[][],int i,int j)
{
    if ( i == 0  ||  j == 0 )   return ; //一个序列为空
    if  (S[i, j] == 1)
    {
        cout<<x[i];
        Structure Sequence (S, i - 1, j - 1) ;
    }
    else if(S[i, j]=2) Structure Sequence (S, i, j - 1) ;
    else Structure Sequence (S, i - 1, j);
}

//循环打表 的方式求LCS[x][y]
int main()
{
    string A = "BCBACABBACDX";
    string B = "BCXYDADJL";     // A 和 B的LCS为BCAD
    int x = A.length();
    int y = B.length();
    A = ' ' + A;
    B = ' ' + B;

    int i, j;
    int LCS[x+1][y+1];
    int S[x+1][y+1];
```

```
    for(i=0;i<=x;i++)
    {
          for (j=0;j<=y;j++)
            LCS[i][j] = 0;//LCS[i][j]用来存储
            S[i][j] = 0;
    }

    for(i=0;i<=x;i++)
        for(j=0;j<=y;j++)
            if(i==0 || j==0)
                LCS[i][j] = 0;

            else
            {
                if(A[i] == B[j]){
                    LCS[i][j] = LCS[i-1][j-1] + 1;
                    S[i][j] = 1;}
                else if(LCS[i-1][j]<LCS[i][j-1]){
                     S[i][j] = 2;
                     LCS[i][j] = LCS[i][j-1];}
                else if(LCS[i-1][j]>LCS[i][j-1]){
                     S[i][j] = 3;
                     LCS[i][j] = LCS[i-1][j];}

            }

    int result = LCS[x][y];
    Structure Sequence(S,x,y);
    cout << result << endl;
    return 0;
}
```

最大字段和




```
#include<cstdio>
#include<algorithm>
using namespace std;
const int maxn = 10010;
int A[maxn], dp[maxn];
int main() {
    int n;
    scanf("%d", &n);
    for(int i=0;i<n;i++){
        scanf("%d",&A[i]);
    }
    dp[0] = 0;
    for (int i = 1; i < n; i++)
        dp[i] = max(A[i], dp[i - 1] + A[i]);
    int k = 0;
    for (int i = 1; i < n; i++) {
        if (dp[i] > dp[k])
            k = i;
```

```
    }
    printf("%d\n", dp[k]);
    return 0;
}
```

贪心算法

- 贪心法与动态规划算法不一样，动态规划
- 贪心法的求解过程也是多步判断,每步判断

可以用贪心算法求解的问题一般具有2个重要的性质：

1.最优子结构性质。

- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

2.贪心选择性质

- 贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。
- 贪心选择每次选取当前最优解，因此它依赖以往的选择，而不依赖于将来的选择。
- 贪心算法通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。
- 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

活动选择问题

把活动按照截止时间从小到大排序,使得 $f_1 \le f_2 \le ..$

以上策略中的挑选都要注意满足相容性条件

i	1	2	3	4	5	6	7	
s_i	1	3	0	5	3	5	6	
f_i	4	5	6	7	8	9	10	

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

public class test{
    static class myComparator implements Comparator<int[]> {
        //按起点递增排序
        public int compare(int[] a,int[] b){
            return a[1]-b[1];
        }
    }
    public static int greedyActivitySelector(int[][] act){
        //每次选择最早结束的活动
        int num=1;
        int i=0;
        for (int j = 1; j <act.length ; j++) {
            if(act[j][0] >=  act[i][1]){
                i=j;
                num++;
            }
        }
        return num;
    }

    public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        int number=scanner.nextInt();
        int[][] act=new int[number][2];
        int idx=0;
        for (int i = 0; i <act.length ;i++) {
            act[i][0]=scanner.nextInt();
            act[i][1]=scanner.nextInt();
        }
        //按照活动截止时间从小到大排序
        myComparator mycomparator=new myComparator();
        Arrays.sort(act,mycomparator);
        int ret=greedyActivitySelector(act);
        System.out.println(ret);
    }
}
```

最优装载问题

输人:集装箱集合N={1,2，…,n} ,集装箱i的重量w_i

1.对集装箱重量排序,使得$w_1 \le w_2 \le \ldots \le w_n$

2.I←{1}

3. W←w_1

4. for j←2 to n do

5. if $W+w_j \le C$

6. then W← $W+w_j$

7. I←I∪U{j}

8. else Return I，W

最小延迟调度

客户集合A={1, 2, 3, 4, 5}

服务时间集合T = <5, 8, 4, 10, 3>

截至时间D=<10, 12, 15, 11, 20 >

按截至时间D从前往后安排

$f_2(1)=0, f_2(2)=15, f_2(3)=23, f_2(4)=$

各任务延迟： 0, 11, 12, 4, 10; 最大延迟：12

Schedule //按照截至时间从小到大选择任务，安排时不留

输入：A, T, D

输出：f

1. 排序A使得$d_1 \le d_2 \le \ldots \le d_n$ //O(nlogn)
2. $f(1) \leftarrow 0$
3. $i \leftarrow 2$
3. while $i \le n$ do
4. $f(i) \leftarrow f(i-1)+t_{i-1}$ //任务i-1结束时刻是任务i开始时刻
5. $i \leftarrow i+1$

设计思想：按完成时间从早到晚安排任务，没有空闲

时间复杂度：$O(nlogn)$

```
public class Scheduling {
    int index;  //任务编号
    int time;  //服务时间
    int deadTime;  //截止时间

    public Scheduling() {
        // TODO Auto-generated constructor stub
    }
```

```
    public Scheduling(int index, int time, int deadTime) {
        super();
        this.index = index;
        this.time = time;
        this.deadTime = deadTime;
    }


}

public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scheduling scheduling[]=new Scheduling[5];
        scheduling[0]=new Scheduling(1,5,10);
        scheduling[1]=new Scheduling(2,8,12);
        scheduling[2]=new Scheduling(3,4,15);
        scheduling[3]=new Scheduling(4,10,11);
        scheduling[4]=new Scheduling(5,3,20);

        //将截止时间按升序排序
        Arrays.sort(scheduling,new Comparator<Scheduling>() {
            public int compare(Scheduling o1, Scheduling o2) {
                return o1.deadTime-o2.deadTime;
            };
        });

        int minimumdelayScheduling = MinimumdelayScheduling(scheduling);
        System.out.println("最小延迟调度时间是："+minimumdelayScheduling);
    }
    //序号       N = <1 , 4 , 2 , 3 , 5>
    //服务时间集合T = <5, 10, 8, 4, 3>
    //截至时间D=<10, 11, 12, 15, 20 >
    //贪心法解最小延迟调度问题
    public static int MinimumdelayScheduling(Scheduling scheduling[]) {
        int time=0;
        int delayTime=0;
        for(int i=0;i<scheduling.length;i++) {
            time+=scheduling[i].time;
            delayTime=Math.max(delayTime, time-scheduling[i].deadTime);
        }
        return delayTime;
    }
```

找零钱问题

最优前缀码（哈夫曼编码）

例 最优前缀码 给定字符集$C=\{x_1, x_2, \ldots, x_n\}$和每
的频率$f(x_i), i=1,2,\ldots,n$, 求关于C的一个最优前缀码

算法 Huffman(C)
输入： $C=\{x_1, x_2, \ldots, x_n\}$， $f(x_i), i=1, 2, \ldots, n$.
输出： Q //队列

```
1. n←|C|
2. Q←C        //按频率递增构成队列Q
3. for i←1 to n−1 do
4.     z←Allocate-Node()     // 生成结点z
5.     z.left←Q中最小元      // 取出Q最小元作z的
6.     z.right←Q中最小元     // 取出Q最小元作z的右
7.     f(z)←f(x)+f(y)
8.     Insert(Q,z)           // 将 z 插入Q
9. return Q
```

最小生成树

最小生成树

Kruskal（克鲁斯卡尔）算法（不能画圈）

- 归并边，适于稀疏网
- 时间复杂度：O(eloge)
- 存储表示: 邻接表

算法 Kruskal伪码

输入：连通图$G=<V,E,W>$　　//顶点数n，边数m

输出：G的最小生成树

1. 按权从小到大顺序排序G中的边，使得$E=\{e_1,$
2. $T\leftarrow\varnothing$
3. repeat
4. 　　$e\leftarrow E$中的最短边
5. 　　if e的两端点不在同一个连通分支
6. 　　then $T\leftarrow T\cup\{e\}$　　//把e加入树中
7. 　　$E\leftarrow E-\{e\}$
8. until T包含了$n-1$条边

Prim（普里姆）算法 (画圈)

- 归并顶点，与边数无关，适于稠密网
- 时间复杂度：$O(n^2)$
- 存储表示: 邻接矩阵

算法 Prim伪码

输入：连通图 $G=<V, E, W>$

输出：G 的最小生成树 T

1. $S \leftarrow \{1\}$；$T \leftarrow \varnothing$
2. while $V-S \neq \varnothing$ do
3. 从 $V-S$ 中选择 j 使得 j 到 S 中顶点的边 e 的权最小；$T \leftarrow T \cup \{e\}$
4. $S \leftarrow S \cup \{j\}$

算法名	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	O(eloge)
适应范围	稠密图	稀疏图

最短路径Dijkstra算法

先找出从源点v0到各终点vk的直达路径（v0,vk），即通过一条弧到达的路径。 从这些路径中找出一条长度最短的路径（v0,u）,然后对其余各条路径进行适当调整： 若在图中存在弧（u,vk），且（v0,u）+（u,vk）<（v0,vk）,则以路径（v0,u,vk）代替（v0,vk）。 在调整后的各条路径中，再找长度最短的路径，依此类推

算法4.7 Dijkstra伪码

输入：带权有向图$G=<V,E,W>$，源点$s \in V$

输出：数组L,对所有$j \in V-\{s\}$,$L[j]$表示s到j的最短结点的标号

```
1. S←{s}
2. dist[s]←0
3.  for i∈V–{s}  do
4.       dist[i]←w(s,i)        // 如果 s 到 i 没有边, w(s,
5. while  V–S ≠ ∅  do
6.      从V–S 中取出具有相对S的最短路径的结点
        上连接j的结点
7.      S←S∪{j}; L[j]←k
8.      for  i∈V–S  do
9.          if  dist[j] + w(j,i) < dist[i]
10.        then  dist[i]← dist[j]+w(j,i)   // 更新i相对S
```

回溯法（DFS+剪枝）

分支限界法与回溯法的比较

相同点：

1. 均需要先定义问题的解空间，确定的解空间组织结构一般都是树或图。
2. 在问题的解空间树上搜索问题解。
3. 搜索前均需确定判断条件，该判断条件用于判断扩展生成的结点是否为可行结点。
4. 搜索过程中必须判断扩展生成的结点是否满足判断条件，如果满足，则保留该扩展生成的结点，否则舍弃。

不同点：

1. 搜索目标：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。

2. 搜索方式不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树

3. 扩展方式不同：在回溯法搜索中，扩展结点一次生成一个孩子结点，而在分支限界法搜索中，扩展结点一次生成它所有的孩子结点

n皇后问题

```
#include<stdio.h>
#include<iostream>
#include<cmath>
using namespace std;
//用一维数组存储，解决行冲突，数组下标为行数，数组值为列数
int x[10]={0};
int count=0;
//剪枝
bool judge(int k)//判断第k行某个皇后是否发生冲突 ，发生冲突就跳过
{
    int i=1;
    while(i<k)  //循环i到k-1之前的皇后；
    {
        if(x[i]==x[k]||abs(x[i]-x[k])==abs(i-k))//存在列冲突或者存在对角线冲突（实际上就是一种剪枝）
            return false;//表示无法放入
        i++;
    }
    return true;//表示可以放入
}
//回溯
void queen(int n)
{
    int i,k=1; //k为当前行号，从第一行开始
    x[1]=0;//x[k]为第k行皇后所放的列号
    while(k>0)
    {
        x[k]++;  //首先从第一列开始判断
        while(x[k]<=n&&!judge(k))   x[k]++;
       //推导k行到底选择哪一列，如果当前该列不行则下一列，直到成立即可； 超过列数或者可以放入循环结束

        if(x[k]<=n)
        {
            if(k==n)//输出所有解 判断到最后一行，皇后放完可以输出结果
            {
                for(i=1;i<=n;i++)
                    printf("第%d行的皇后：在第%d列  ",i,x[i]);
                count++;
                printf("-------这是其中第%d个解",count);
                printf("\n");
            }
            else//判断下一行
            {
                k++; x[k]=0;
            }
        }
        else k--;//没找到，回溯再让上一行的到下一列进行判断
    }
    return ;
}
int main()
{
    int n;
    cout<<"请输入你的n皇后：      ";
```

```
    cin>>n;//n最大值不超过10
    queen(n);
    return 0;
}
```

0-1背包

```
#include <iostream>
#include <stdio.h>
//#include <conio.h>
using namespace std;
int n;//物品数量
double c;//背包容量
double v[100];//各个物品的价值  value
double w[100];//各个物品的重量  weight
double cw = 0.0;//当前背包重量  current weight
double cp = 0.0;//当前背包中物品总价值  current value
double bestp = 0.0;//当前最优价值best price
double perp[100];//单位物品价值(排序后) per price
int order[100];//物品编号
int put[100];//设置是否装入，为1的时候表示选择该组数据装入，为0的表示不选择该组数据


//按单位价值排序
void knapsack()
{
    int i,j;
    int temporder = 0;
    double temp = 0.0;

    for(i=1;i<=n;i++)
        perp[i]=v[i]/w[i]; //计算单位价值（单位重量的物品价值）
    for(i=1;i<=n-1;i++)
    {
        for(j=i+1;j<=n;j++)
            if(perp[i]<perp[j])//冒泡排序perp[],order[],sortv[],sortw[]
        {
            temp = perp[i];  //冒泡对perp[]排序
            perp[i]=perp[j];
            perp[j]=temp;

            temporder=order[i];//冒泡对order[]排序
            order[i]=order[j];
            order[j]=temporder;

            temp = v[i];//冒泡对v[]排序
            v[i]=v[j];
            v[j]=temp;

            temp=w[i];//冒泡对w[]排序
            w[i]=w[j];
            w[j]=temp;
        }
    }
}

//回溯函数
void backtrack(int i)
{   //i用来指示到达的层数（第几步，从0开始），同时也指示当前选择玩了几个物品
    double bound(int i);
    if(i>n) //递归结束的判定条件
    {
        bestp = cp;
        return;
```

```
    }
    //如若左子节点可行，则直接搜索左子树;
    //对于右子树，先计算上界函数，以判断是否将其减去
    if(cw+w[i]<=c)//将物品i放入背包,搜索左子树
    {
        cw+=w[i];//同步更新当前背包的重量
        cp+=v[i];//同步更新当前背包的总价值
        put[i]=1;
        backtrack(i+1);//深度搜索进入下一层
        cw-=w[i];//回溯复原
        cp-=v[i];//回溯复原
    }
    if(bound(i+1)>bestp)//如若符合条件则搜索右子树
        backtrack(i+1);
}

//计算上界函数，功能为剪枝
double bound(int i)
{   //判断当前背包的总价值cp+剩余容量可容纳的最大价值<=当前最优价值
    double leftw= c-cw;//剩余背包容量
    double b = cp;//记录当前背包的总价值cp,最后求上界
    //以物品单位重量价值递减次序装入物品
    while(i<=n && w[i]<=leftw)
    {
        leftw-=w[i];
        b+=v[i];
        i++;
    }
    //装满背包
    if(i<=n)
        b+=v[i]/w[i]*leftw;
    return b;//返回计算出的上界

}



int main()
{
    int i;
    printf("请输入物品的数量和背包的容量：");
    scanf("%d %lf",&n,&c);
    /*printf("请输入物品的重量和价值：\n");
    for(i=1;i<=n;i++)
    {
        printf("第%d个物品的重量：",i);
        scanf("%lf",&w[i]);
        printf("第%d个物品的价值是：",i);
        scanf("%lf",&v[i]);
        order[i]=i;
    }*/
    printf("请依次输入%d个物品的重量:\n",n);
    for(i=1;i<=n;i++){
        scanf("%lf",&w[i]);
        order[i]=i;
    }

    printf("请依次输入%d个物品的价值:\n",n);
    for(i=1;i<=n;i++){
        scanf("%lf",&v[i]);
    }


    knapsack();
    backtrack(1);
```

```
    printf("最优价值为：%lf\n",bestp);
    printf("需要装入的物品编号是：");
    for(i=1;i<=n;i++)
    {
        if(put[i]==1)
            printf("%d ",order[i]);
    }
    return 0;
}
```

分支限界法（BFS）

多段图最短路径

```
#include<iostream>
#include<cstdio>
#include<queue>
using namespace std;

typedef struct ArcCell{
    int weight;//保存权值 weight=1说明一开始就输入了这条边，=0说明这两个点之间没有边
    int min_length;//存储最短路径长度
}ArcCell,AdjMaxtrix[100][100];

typedef struct{
    int data;//顶点编号
    int length;//起始顶点到data编号顶点的最短路径
}VerType;

typedef struct{
    VerType vexs[100];//顶点向量
    AdjMaxtrix arcs;
    int dingdian;//顶点数
    int bian;//边的数量
}Graph;

Graph G;
queue<int> q;

void CreateGraph()
{
    int m,n,t;
    printf("输入顶点数和边数:");
    scanf("%d%d",&G.dingdian,&G.bian);
    printf("输入顶点编号:");
    for(int i=1 ; i<=G.dingdian ; i++)
    {
        scanf("%d",&G.vexs[i].data);
        G.vexs[i].length=10000;
    }

    for(int i=1;i<=G.dingdian;i++)
        for(int j=1;j<=G.dingdian;j++)
        {
            G.arcs[i][j].weight=0;//先置零 weight=1说明一开始就输入了这条边，=0说明这两个点之间没有边
        }
    printf("输入所有边的起点、终点和边长（用空格隔开）:\n");
    for(int i=1;i<=G.bian;i++)
        {
            scanf("%d%d%d",&m,&n,&t);
            G.arcs[m][n].weight=1;//weight=1说明一开始就输入了这条边，=0说明这两个点之间没有边
            G.arcs[m][n].min_length=t;
        }
```

```
}

int Nextweight(int v,int w)
{
    for(int i=w+1;i<=G.dingdian;i++)
        if(G.arcs[v][i].weight)
            return i;//返回已有的边中序号最前的
    return 0;//未找到符合条件结点，就直接返回最初的结点;
}

void ShortestPaths(int v)
{
    int k=0;//从首个节点开始访问，k记录想要访问结点的位置
    int t;
    G.vexs[v].length=0;
    q.push(G.vexs[v].data);
    while(!q.empty())//队列q不为空的时候执行循环
    {
        t=q.front();//t为队列中第一个元素，也就是最先要被弹出的结点，返回值是其编号，第一次执行这一步时是 G.vexs[1].data，
        k=Nextweight(t,k);//k不断更新为已有的边中序号最前的
        while(k!=0)
        {
            if(G.vexs[t].length+G.arcs[t][k].min_length<=G.vexs[k].length)//减枝操作
            {
                G.vexs[k].length=G.vexs[t].length+G.arcs[t][k].min_length;

                q.push(G.vexs[k].data);
            }
            k=Nextweight(t,k);//k不断更新为已有的边中序号最前的
        }
        q.pop();
    }
}

void Print()
{
    for(int i=2;i<=G.dingdian;i++)
        printf("顶点1到顶点%d的最短路径长是：%d\n",G.vexs[i].data,G.vexs[i].length);
}

int main()
{
    CreateGraph();
    ShortestPaths(1);
    Print();
    return 0;
}
```