1. 算法基础

算法复杂度的度量

1) 改进算法和提高计算机处理能力对算法速度的影响（课堂上讲过相关提高算法效率的实例）

2) 渐进意义下，算法的复杂度的同阶度量：

$f(N)=O(g(N))$的定义，以及O的运算性质证明。

3) 给出一个表达式，证明n^2是n^2+2n+6的上界

下面的讨论中，对所有 n，$f(n)\geq 0$，$g(n)\geq 0$。

渐近上界记号 O

$O(g(n))=\{ f(n) \mid$ 存在正常数 c 和 n_0 使得对所有 $n\geq n_0$ 有：$0\leq f(n)\leq cg(n)\}$

渐近下界记号 Ω

$\Omega(g(n))=\{ f(n) \mid$ 存在正常数 c 和 n_0 使得对所有 $n\geq n_0$ 有：$0\leq cg(n)\leq f(n)\}$

非紧上界记号 o

$o(g(n))=\{ f(n) \mid$ 对于任何正常数 $c>0$，存在正数 $n_0>0$ 使得对所有 $n\geq n_0$ 有：$0\leq f(n)<cg(n)\}$

等价于 $f(n)/g(n)\to 0$ ， as $n\to\infty$。

非紧下界记号 ω

$\omega(g(n))=\{ f(n) \mid$ 对于任何正常数 $c>0$，存在正数 $n_0>0$ 使得对所有 $n\geq n_0$ 有：$0\leq cg(n)<f(n)\}$

等价于 $f(n)/g(n)\to\infty$ ， as $n\to\infty$。

$f(n)\in\omega(g(n)) \Leftrightarrow g(n)\in o(f(n))$

紧渐近界记号 Θ

$\Theta(g(n))=\{ f(n) \mid$ 存在正常数 c_1, c_2 和 n_0 使得对所有 $n\geq n_0$ 有：$c_1g(n)\leq f(n)\leq c_2g(n)\}$

2. 递归与分治策略

分治法的设计思想、递归的实质、排列问题、整数划分问题、二分搜索、大数乘法、strassen 矩阵乘、棋盘覆盖、合并排序、快速排序、线性时间选择（寻找中位数）、最近点对问题、循环赛日程安排问题。

例：分析二分搜索的时间复杂性，改写二分搜索算法使得搜索元素 x 不在数组中时，返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j。如搜索元素在数组中，i 和 j 相同，均为 x 在数组中的位置。

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如果问题可分割成 k 个子问题，且这些子问题都可解，利用这些子问题可解出原问题的解，此分治法是可行的。

由分治法产生的子问题往往是原问题的较少模式，为递归提供了方便。

例如：快速排序：

```
 template<class Type>
int Partition (Type a[], int p, int r)
{
```

```
        int i = p, j = r + 1;
        Type x=a[p];
        // 将< x 的元素交换到左边区域
        // 将> x 的元素交换到右边区域
        while (true) {
            while (a[++i] <x);
            while (a[- -j] >x);
            if (i >= j) break;
            Swap(a[i], a[j]);
            }
        a[p] = a[j];
        a[j] = x;
        return j;
}
```

3. 动态规划

矩阵连乘问题、系列赛、最长公共子序列、最大子段和、凸多边形最优三角剖分、图像压缩、电路布线、0-1 背包问题。

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题

与分治区别：经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次

用分治法求解，子问题的数目常有多项式量级，有些子问题被重复计算了多次，如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

动态规划即是用一个表来记录所有已解决的子问题。

动态规划算法常用于求解具有某种最优性质的问题。可能会有多个解，每个解都对应于一个值，希望找到最优解。

例如：矩阵连乘问题

```
void matrixmultiply(int **a, int **b, int **c, int ra, int ca,
                                                    int rb, int cb)
{
    if (ca!=rb) error(“不可乘!”);
    for (int i=0; i<ra; i++)
        for (int j=0; j<cb; j++){
            int sum=a[i][0]*b[0][j];
            for (int k=1; k<ca; k++)
                sum+=a[i][k]*b[k][j];
                c[i][j]=sum;
        }
}
```

三重循环中总共需要 *pqr* 次数乘。

$A=50\times10$ $B=10\times40$ $C=40\times30$ $D=30\times5$

$(A((BC)D))$ 16000

特征：计算 $A[i:j]$的最优次序所包含的计算矩阵子链 $A[i:k]$和 $A[k+1:j]$的次序也是最优的。

矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**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求解的显著特征。

用动态规划法求最优解：

```
void MatrixChain(int *p，int n，int **m，int **s)
{
        for (int i = 1; i <= n; i++) m[i][i] = 0;
        for (int r = 2; r <= n; r++)
            for (int i = 1; i <= n - r+1; i++) {
                int j=i+r-1;
                m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
                s[i][j] = i;
                for (int k = i+1; k < j; k++) {                    //
                    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;}
                }
           }
}
```

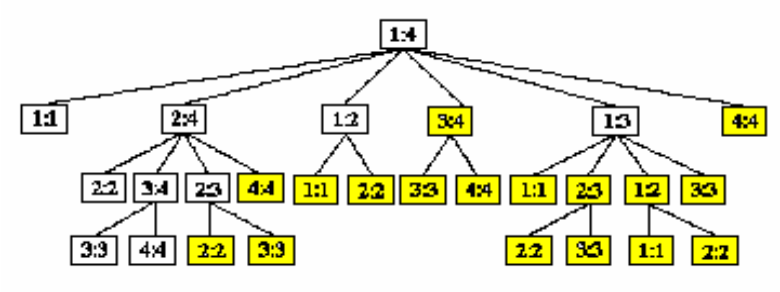
算法复杂度分析：

算法 **matrixChain** 的主要计算量取决于算法中对 r，i 和 k 的 3 重循环。循环体内的计算量为 O(1)，而 3 重循环的总次数为 $O(n^3)$。因此算法的计算时间上界为 $O(n^3)$。算法所占用的空间显然为 $O(n^2)$。

利用递归直接计算 A[*i*,*j*]的递归算法 RecurMatrixChain：

```
int RecurMatrixChain(int i，int j)
{
        if (i==j) return 0;
        int u=RecurMatrixChain(i,i)+RecurMatrixChain(i+1,j)*p[i-1]*p[i]*p[j];
        s[i][j]=i;
            for (int k = i+1; k < j; k++) {
                int
t=RecurMatrixChain(i,k)+RecurMatrixChain(k+1,j)*p[i-1]*p[k]*p[j];
                if (t < u) { u = t; s[i][j] = k;}
            }
}
```

用上述算法 RecurMatrixChain(1,4)计算 A[1:4]的递归树：P22



0-1 背包问题

给定 n 种物品和一背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1 背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

设所给0-1背包问题的子问题 $\max \sum_{k=i}^{n} v_k x_k$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优值为 $m(i,j)$，即 $m(i,j)$ 是背包容量为 j，可选择物品为 $i, i+1, \ldots, n$ 时0-1背包问题的最优值。由0-1背包问题的最优子结构性质，可以建立计算 $m(i,j)$ 的递归式如下。

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$

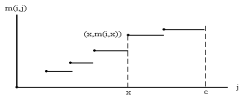
$$m(n,j) = \begin{cases} v_n & j \ge w_n \\ 0 & 0 \le j < w_n \end{cases}$$

60

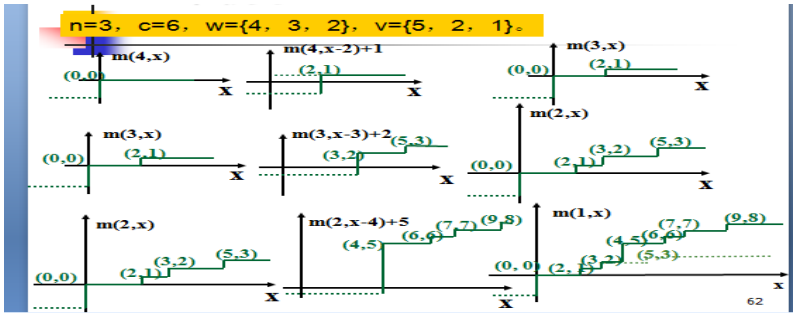
算法复杂度分析：

从 $m(i,j)$ 的递归式容易看出，算法需要 $O(nc)$ 计算时间。当背包容量 c 很大时，算法需要的计算时间较多。例如，当 $c>2^n$ 时，算法需要 $\Omega(n2^n)$ 计算时间。

由 $m(i,j)$ 的递归式容易证明，在一般情况下，对每一个确定的 $i(1 \le i \le n)$，函数 $m(i,j)$ 是关于变量 j 的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数 $m(i,j)$ 由其全部跳跃点唯一确定。如图所示。



对每一个确定的 $i(1 \le i \le n)$，用一个表 $p[i]$ 存储函数 $m(i, \mathrm{i})$ 的全部跳跃点。表 $p[i]$ 可依计算 $m(i,j)$ 的递归式递归地由表 $p[i+1]$ 计算，初始时 $p[n+1]=\{(0,0)\}$。 实例：



4. 贪心法

原理和设计思想、TSP 问题、图着色问题、最小生成树问题（最近顶点策略-Prim 算法、最短边策略-Krushal 算法）、背包问题（有别于 0-1 背包，可以找到最优解）、活动安排问题、多机调度问题。

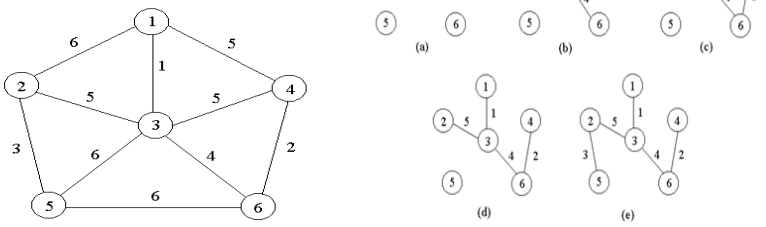
例：三个顾客需要的服务时间分别是 t_1、t_2、t_3，求解顾客的服务顺序， $\min\left(\sum_{i=1}^{n} t_i\right)$，

证明使用贪心法可以获得最优解。

例：最小生成树的生成步骤。

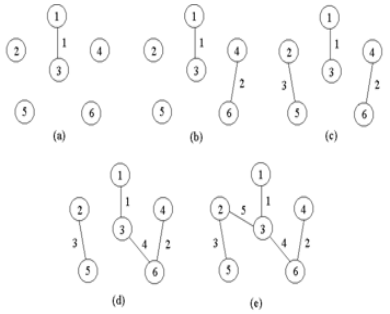
顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的**局部最优**选择。

最小生成树问题-Prim 算法



Krushal 算法：

使生成树以一种随意的方式生长，先让树随意生长，每长一次将两棵树合并 → 直到成一棵树。



背包问题中的贪心法

选择价值最大的物品，但虽然背包总价上升很快，背包容量也消耗太快，不能保证目标函数最大；

选择最轻的物品，装尽可能多的物品，虽然容量消耗慢，但背包价值没法保证迅速上升；

最后一种策略 → 选择单位重量价值最大的物品

如果其重量小于背包容量，就可以把它装入，并将容量减去该物品的重量，然后，转化为一个规模变小的背包问题。

背包问题具有最优子结构性质。

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。

但是，对于具有最优子结构的问题应该选用贪心算法还是动态规划算法求解?

是否能用动态规划算法求解的问题也能用贪心算法求解?

下面研究 2 个经典的组合优化问题，并以此说明贪心算法与动态规划算法的主要差别。

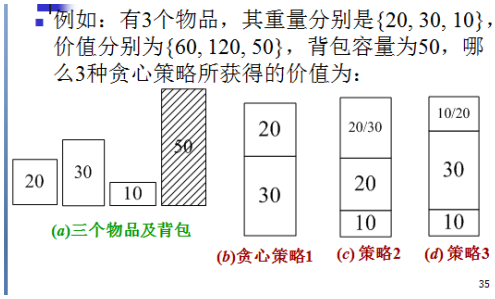
依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。

若将这种物品全部装入背包后，背包内的物品总重量未超过 *C*，则选择单位重量价值次高的物品并尽可能多地装入背包。

依此策略一直地进行下去，直到背包装满为止。

具体算法可描述如下：

```
void Knapsack(int n,float M,float v[],float w[],float x[])
{
    Sort(n,v,w);
    int i;
    for (i=1;i<=n;i++) x[i]=0;
    float c=M;
    for (i=1;i<=n;i++) {
        if (w[i]>c) break;
        x[i]=1;
        c-=w[i];
        }
    if (i<=n) x[i]=c/w[i];
}
```



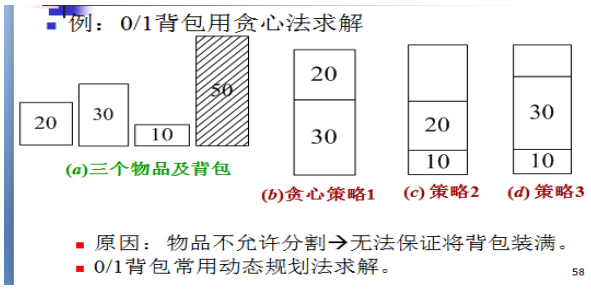
- 0/1 背包常用动态规划法求解。

对于 0-1 背包问题，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。

事实上，在考虑 0-1 背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。

由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。

实际上也是如此，动态规划算法的确可以有效地解 0-1 背包问题。



5. 回溯法

图着色问题、哈密顿回路、8-皇后问题、批处理作业调度问题。

例：设有 n=3 个正数的集合 $W=\{w_0,w_1,w_2\}=\{2,3,5\}$和整数 M=14，求 W 的所有满足条件的子集，使子集中的正数之和等于 M。请画出用回溯法求解的状态空间树。（采用固定长度 3-元组表示解）。

回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。

回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

问题的解向量：回溯法希望一个问题的解能够表示成一个 n 元式$(x_1,x_2,\ldots,x_n)$的形式。

显约束：对分量 x_i 的取值限定。

隐约束：为满足问题的解而对不同分量之间施加的约束。

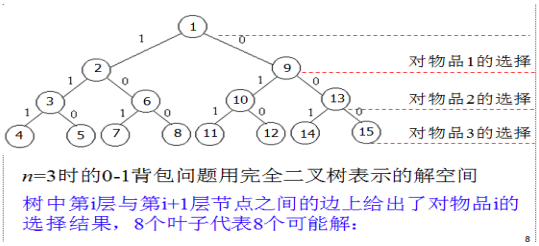
解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

例：对 n 个物品的 0/1 背包问题，其可能解的表示方式：

可能解由一个不等长向量组成，解向量的长度等于装入背包的物品个数，长度由 0~n 的解向量组成。

如 n=3，解空间{( ),(1),(2),(3),(1,2),(1,3),(2,3),(1,2,3)}

可能解由一个等长向量$\{x_1,\ldots, x_n\}$组成，则 n=3，解空间为{(0,0,0), (0,0,1), (0,1,0), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)}



n=3时的0-1背包问题用完全二叉树表示的解空间

树中第i层与第i+1层节点之间的边上给出了对物品i的选择结果，8个叶子代表8个可能解：

8

6. 分支限界法

0-1 背包问题、TSP 问题、组合问题（任务分配问题）、批处理作业调度问题

分支限界法与回溯法

（1）求解目标：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）搜索方式的不同：

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。

分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。对已处理的各结点根据限界函数估算目标函数的可能取值，从中选取使目标函数取得极值（极大/极小）的结点优先进行广度优先搜索 → 不断调整搜索方向，尽快找到解。

特点：限界函数常基于问题的目标函数，适用于求解最优化问题。

在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。

此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

常见的两种分支限界法

（1）队列式(FIFO)分支限界法

按照队列先进先出（FIFO）原则选取下一个结点为扩展结点。

（2）优先队列式分支限界法

按照优先队列中规定的优先级选取优先级最高的结点成为当前扩展结点。

例子：0/1 背包

（1）回溯求解 0/1 背包问题，虽剪枝减少了搜索空间，但整个搜

索按深度优先机械进行，是盲目搜索（不可预测本结点以下的结点进行的如何）。

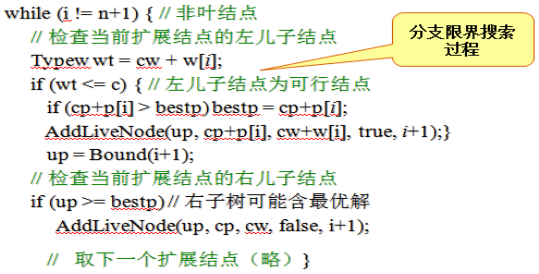
分支限界法首先确定一个合理的限界函数，并根据限界函数确定目标函数的界[down, up]；

然后按照广度优先策略遍历问题的解空间树，在某一分支上，依次搜索该结点的所有孩子结点，分别估算这些孩子结点的目标函数的可能取值（对最小化问题，估算结点的 down，对最大化问题，估算结点的 up）。

如果某孩子结点的目标函数值超出目标函数的界，则将其丢弃（从此结点生成的解不会比目前已得的更好），否则入待处理表。

上界函数

```
// n 表示物品总数，cleft 为剩余空间
while (i <= n && w[i] <= cleft)
{
        cleft -= w[i];                          //w[i]表示 i 所占空间
        b += p[i];                              //p[i]表示 i 的价值
        i++;
}
if (i <= n) b+=p[i]/w[i] * cleft;       // 装填剩余容量装满背包
return b;                                       //b 为上界函数
```



从 0/1 背包问题的搜索过程可看出：与回溯法相比，分支限界法可根据限界函数不断调整搜索方向，选择最可能得最优解的子树优先进行搜索 → 找到问题的解。

7. 概率算法

概率算法的设计思想

概率算法(probabilistic algorithm)，把“对于所有合理的输入都必须给出正确的输出”这一求解问题的条件放宽：

允许算法在执行过程中随机选择下一步该如何进行；

允许结果以较小概率出现错误，并以此为代价，获得算法运行时间的大幅度减少。

1)、如一个问题无有效确定性算法可在一个合理时间内给出解答，且该问题能接受小概率的错误，用概率算法可快速找到这个问题的解。

2)、这种测试的随机向量越多，结论出错可能就越小。

3)、不难看出：在算法中增加这种随机性的因素，常可引导算法快速求解，概率算法所需的时间，常小于其它确定算法。

4） 、对确定性算法，常可分析平均情况下，及最坏情况下的时间复杂性；

5）、对概率算法，常可分析平均情况下，及最坏情况下的期望(expected)时间复杂性，即由概率算法反复运行同一输入实例所得的平均运行时间。

舍伍德（Sherwood）型概率算法（随机洗牌、如何改造一个算法→舍伍德算法、舍伍德算法的目的、选择问题）

如一个确定性算法无法直接改造成舍伍德算法，可用随机预处理技术，不改变原有的确定性算法，仅对其输入实例随机排列（洗牌）。

例如：快速排序 随机选择中间轴

```
template<class Type>
void QuickSort(int r[], int low, int high)
{// 随机洗牌算法
  if (low<high){
      i=Random(low, high);
      r[low]←→r[i];
      k=Pratition(r, low, high);
      QuickSort(r, low, k-1);
      QuickSort(r, k+1, high);
  }
}
```

拉斯维加斯（Las Vegas）型概率算法（8-皇后问题）。

拉斯维加斯算法的一个显著特征是它所作的随机性决策有可能导致算法找不到问题所需的解，即算法运行一次或者得到一个正确解，或者无解。

void **obstinate**(Object *x*, Object *y*)

{ // 反复调用拉斯维加斯算法 LV(*x*,*y*)，直到找到问题的一个解 *y*

bool success= false;

while (!success) success=LV(*x*, *y*);

}

例如：

对于 *n* 后问题的任何一个解而言，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更象是随机放置的。由此容易想到下面的拉斯维加斯算法。

n 后问题的 LV 思路：各行随机放置皇后，使新放的与已有的互不攻击，until(8 皇后放好||无可供下一皇后放置的位置)。

1) *x*[8]←0; count←0;

2) for (*i*=1;*i*<=8; *i*++)

2.1) 产生一个[1,8]的随机数 *j*;

2.2) count=count+1; //第 count 试探

2.3) *if* (皇后 *i* 放在位置 *j* 不发生冲突)

则 $x[i]=j$;　count=0;　转 2)放下一个皇后；

if (count==8)　算法失败；

else　转 2.1 重新放皇后 i；

3) $x[1]$~$x[8]$作为一个解输出；

可得：

1. 随机放两个皇后，再回溯比完全用回溯快大约两倍；
2. 随机放 3 个皇后，再回溯比完全用回溯快大约一倍；
3. 随机放所有皇后，再回溯比完全用回溯慢大约一倍；

产生随机数所需的时间导致。

蒙特卡罗（Monte Carlo）型概率算法（主元素问题）

- 在实际应用中常会遇到一些问题，不论采用确定性算法或概率算法都无法保证每次都能得到正确的解答。蒙特卡罗算法则在一般情况下可以保证对问题的所有实例都以高概率给出正确解，但是通常无法判定一个具体解是否正确。
- MC 算法偶尔会出错，无论任何输入实例，总可以高概率找一正确解。即 MC 算法总是给出解，此解偶尔可能是不正确，也无法有效判定该解是否正确。

设 p 是一个实数，且 $1/2<p<1$。如果一个 MC 算法对于问题的任一实例得到正确解的概率不小于 p，则称该 MC 算法是 p 正确的，且称 $p-1/2$ 是该算法的优势。

对于一个一致的 p 正确蒙特卡罗算法，要提高获得正确解的概率，只要执行该算法若干次，并选择出现频次最高的解即可。

例如：T[7]={3,2,3,2,3,3,5}

传统就是统计各元素在数组中出现的次数，时间复杂度 $O(n^2)$。

MC 算法：随机选数组中的元素 $T[i]$统计，如果元素出现次数大于 $n/2$，即主元素，返回 *true*，否则 *false*。

设 T[1:n]是一个含有 n 个元素的数组。当|{i|T[i]=x}|>n/2 时，称元素 x 是数组 T 的主元素。

```
template<class Type>
bool Majority(Type *T, int n)
{//  判定主元素的蒙特卡罗算法
     int i=rnd.Random(n)+1;
     Type x=T[i];       //  随机选择数组元素
     int k=0;
     for (int j=1;j<=n;j++)
        if (T[j]==x) k++;
     return (k>n/2);    // k>n/2  时 T 含有主元素
}
template<class Type>
bool MajorityMC(Type *T, int n, double e)
{//  重复调用算法 Majority
     int k=ceil(log(1/e)/log(2));
```

```
        for (int i=1;i<=k;i++)
            if (Majority(T,n)) return true;
        return false;
    }
```

对于任何给定的 ε>0，算法 **majorityMC** 重复调用⌈log(1/ε)⌉次算法 **majority**。它是一个偏真蒙特卡罗算法，且其错误概率小于 ε。算法 **majorityMC** 所需的计算时间显然是 O(nlog(1/ ε))。