时间复杂性和空间复杂性

当给出合法输入时，为了得到输出，该算法所需要的时间和空间。算法复杂度分为时间复杂度和空间复杂度。其作用：时间复杂度是度量算法执行的时间长短；而空间复杂度是度量算法所需存储空间的大小。

分治法(Divide and Conquer)

将一个规模为 n 的问题分解为 k 个规模较小且容易求解的子问题，这些子问题互相独立且与原问题相同，将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

动态规划(Dynamic planning)

通过组合子问题的解而解决整个问题。将待求解问题分解成若干个子问题，先求解子问题的解，然后从子问题的解得到原问题的解。适用于子问题不独立的情况，即各子问题包含公共的子问题。保存已求解子问题的答案，在需要时再找出已求得的答案，可以避免大量重复计算。步骤：1、找出最优解的性质，刻画其结构特征 2、递归的定义最优值 3、以自底向上的方式计算最优值 4、根据计算最优值时得到的信息，构造最优解。

分治法和动态规划的区别

共同点 ：二者都要求原问题具有最优子结构性质，都是将原问题分而治之，分解成若干个规模较小(小到很容易解决的程序)的子问题。然后将子问题的解合并,形成原问题的解.

不同点：分治法将分解后的子问题看成相互独立的，通过用递归来做；动态规划将分解后的子问题理解为相互间有联系,有重叠部分，需要记忆，通常用迭代来做。

备忘录方法(Memo method)

备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法采用自顶向下，为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。一般来说，当一个问题的所有子问题都至少要解一次时，用动态规划算法比用备忘录方法好。此时，动态规划算法没有任何多余的计算，还可以利用其规则的表格存取方式来减少在动态规划算法中的计算时间和空间需求。当子问题空间中部分子问题可以不必求解时，易用备忘录方法则较为有利，因为从其控制结构可以看出，该方法只解那些确实需要求解的子问题。

贪心算法(greedy algorithm)

贪心算法总是作出在当前看来最好的选择。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。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。贪心算法则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。可以用贪心算法求解的问题一般满足贪心选择性质和最优子结构性质。

决策树(Decision Tree)

决策树是在已知各种情况发生概率的基础上，通过构成决策树来求取净现值的期望值大于等于零的概率，评价项目风险，判断其可行性的决策分析方法，是直观运用概率分析的一种图解法。由于这种决策分支画成图形很像一棵树的枝干，故称决策树。

回溯法(Back Tracking Method)

回溯法（探索与回溯法）是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，是具有限界函数的深度优先算法。

分支限界法(Branch and Bound Method)

分支限界法按广度优先策略搜索问题的解空间树，对待处理的节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（极大或极小）的结点优先进行广度优先搜索，从而不断调整搜索方向，尽快找到问题的解。分支限界法适合求解最优化问题。

分支限界和回溯法的区别

1.求解目标不同：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。3.存储结点的常用数据结构：回溯法常采用堆栈；而分支限界法常采用队列。

随机算法（Random algorithm）

分为两类，第一类称为 Las Vegas 算法，它建立的那些随机算法总是或者给出正确的解，或者无解。而另一类 Monte Carlo 算法与之相反，总是给出解，但是偶尔可能会产生非正确的解。然而，可以通过多次运行原算法，并且满足每次运行时的随机选择都相互独立，使产生非正确解的概率可以减到任意小。

近似算法评价标准

近似算法的性能分析包括时间复杂度分析、空间复杂度分析和近似精度分析，其中时间（空间）复杂度的分析同精确复杂度相同。近似精度分析是近似算法特有的，它主要用于刻画近似算法给出的近似解相比于问题优化解的优劣程度。目前，存在三种刻画近似精度的度量，即近似比、相对误差界和1+ε近似。

求二叉树叶子节点数

```
public int getLeaf(TreeNode root) {
if (root == null) {return 0;}
if (root.left == null && root.right == null)
{return 1;}
return getLeaf(root.left)+getLeaf(root.right);}
```

求二叉树高度

```
public int getHeight(TreeNode root){
if(root==null){return 0;}
int leftHeight=getHeight(root.left);
int rightHeight=getHeight(root.right);
return Math.max(leftHeight, rightHeight)+1;}
```

迭代算法

```
1.  for k←1 to n
2.    c[k]←0
3.  end for
4.  flag←false
5.  k←1
6.  while k>=1
7.    While c[k]<= 2
8.       c[k]←c[k]+1
9.       if c为合法着色 then set flag←true且退出两层while循环
10.      else if c为部分解 then k←k+1 {前进}
11.   end while
12.   c[k]←0
13.   k←k-1  {回溯}
14. end while
15. if flag then output c
16. else output "no solution"
```

AB=C

```
bool verificationMatrix(int ** A, int ** B, int ** C, int n) {
    // 产生向量X， X的维度：n * 1
    int * X = new int[n];
    // 设置随机种子
    srand((unsigned)time(NULL));
    // 产生随机数
    for(int i = 0; i < n; i++) {
        X[i] = rand() % 10 + 1;
    }
    // 计算 B * X
    int *p1 = mul(B, X, n);
    // 计算 A * (B * X)
    int *p2 = mul(A, p1, n);
    // 计算 C * X
    int *p3 = mul(C, X, n);
    // 判断 A * (B * X) == C * X
    for(int i = 0; i < n; i++) {
        if(p2[i] != p3[i]) {
            return false;
        }
    }
    return true;
}
```

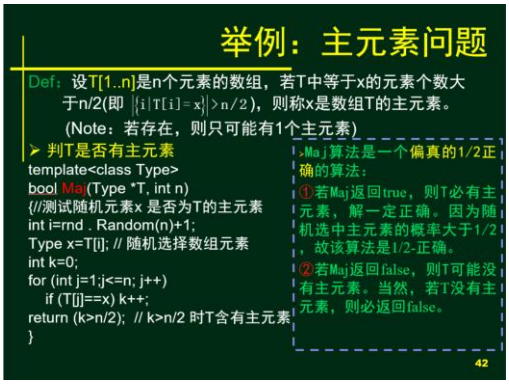
AB=E

```
Public static boolean product(double [][]a,double [][]b,int n){
      cnd=new Random();
      double []x=new double [n+1];
      double []y=new double [n+1];
      double []z=new double [n+1];
      Double []e=new double [n+1];
      for(int i=1;i<=n;i++){
           x[i]=cnd.random(2);
           if(x[i]==0)  x[i]=-1;
    }
      for(int i=1;i<=n:i++){
          for(int j=1;j<=n;j++)
              if(i==j)  e[i][j]=1
              else   e[i][j]=0;
    }
      mult(b,x,y,n);
      mult(a,y,z,n);
      mult(e,x,y,n);
      for(int i=1;i<=n;i++){
          if(y[i]!=z[i])   return false;
          else
              return ture;
    }
}
```

最短路径问题

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	∞	30	100
1	{1,2}	2		60		
2	{1,2,4}	4		50		90
3	{1,2,4,3}	3				60
4	{1,2,4,3,5}	5	10	50	30	60

主元素问题



矩阵连乘

```
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;}
      }
    }
}
```

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

A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

(a) 计算次序

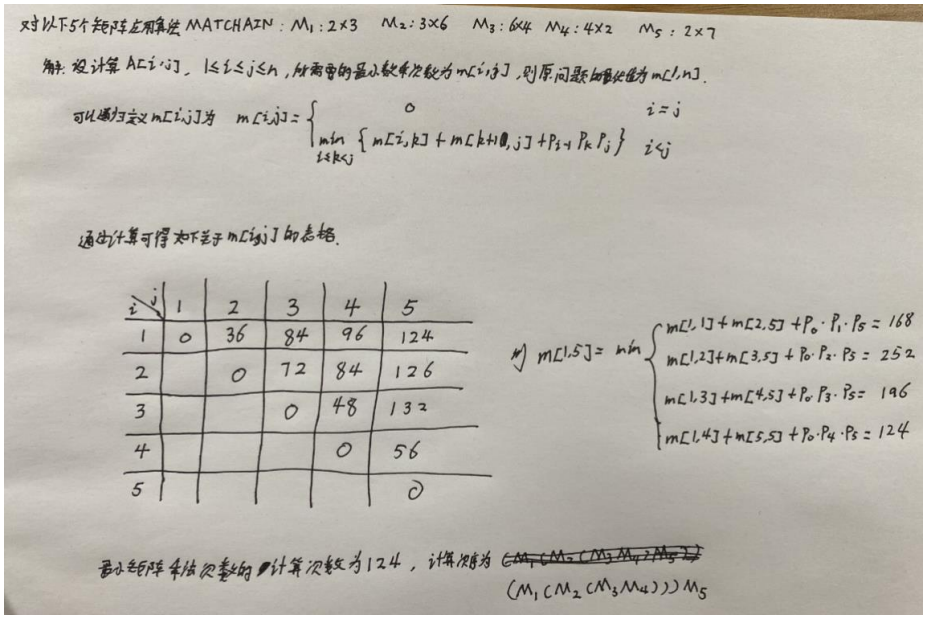
(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(c) s[i][j]

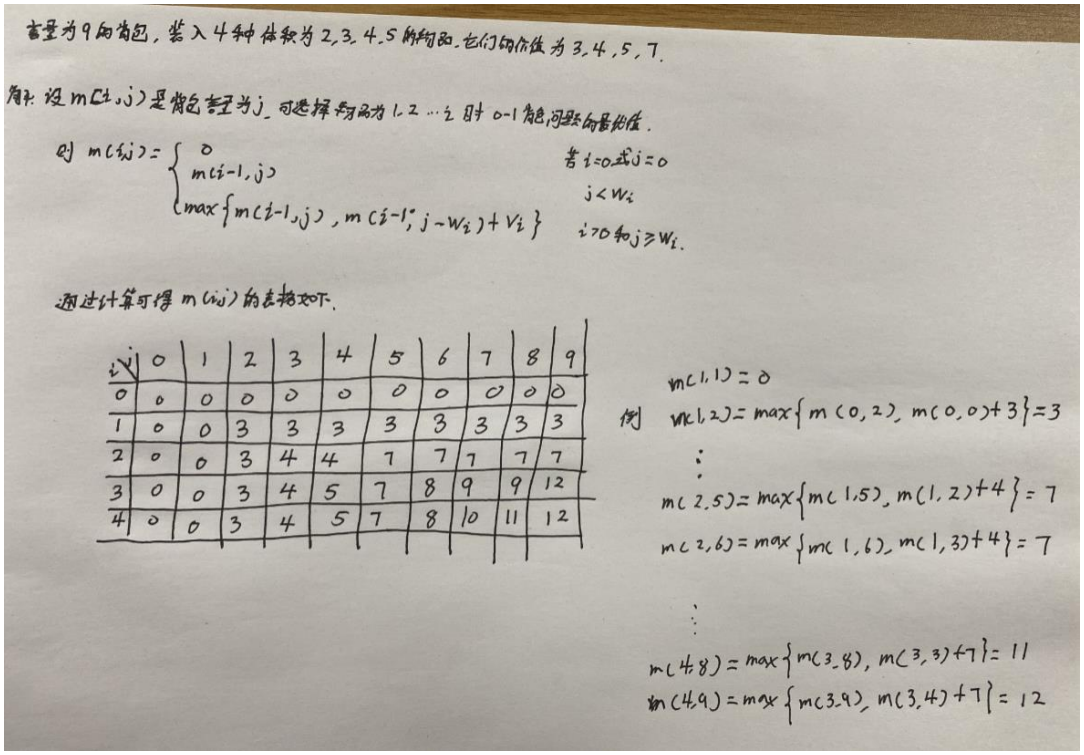
i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

$$m[2][5]=\min\begin{cases}m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000\\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125\\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375\end{cases}$$



0/1 背包

```
for i←0 to n
    V[i,0]←0
end for
for j←0 to C
    V[0,j]←0
end for
for i←1 to n
    for j←1 to C
        V[i,j]←V[i-1,j]
        if si<=j then V[i,j]←Max{V[i,j],V[i-1,j-si]+vi}
    end for
end for
return V[n,C]
```



i\j	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12
4	0	0	3	4	5	7	8	10	11	12

用贪心算法解背包问题的基本步骤

- 首先计算每种物品单位重量的价值V_i/W_i，然后，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];
}
```

算法knapsack的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为O(nlogn)。当然，为了证明算法的正确性，还必须证明背包问题具有贪心选择性质。