算法策略

- 第1章 树的遍历与回溯法
- 第2章 贪心算法
- 第3章 分治策略
- 第4章 动态规划

算法策略
第1章 树的遍历与回溯法

参考书：

《数据结构》严蔚敏

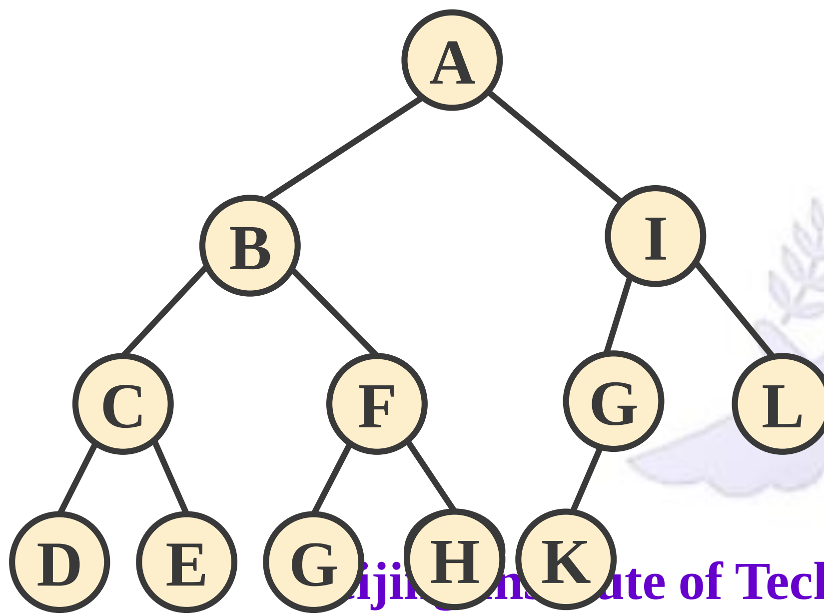
《计算机算法设计与分析》王晓东

高春晓

Contents

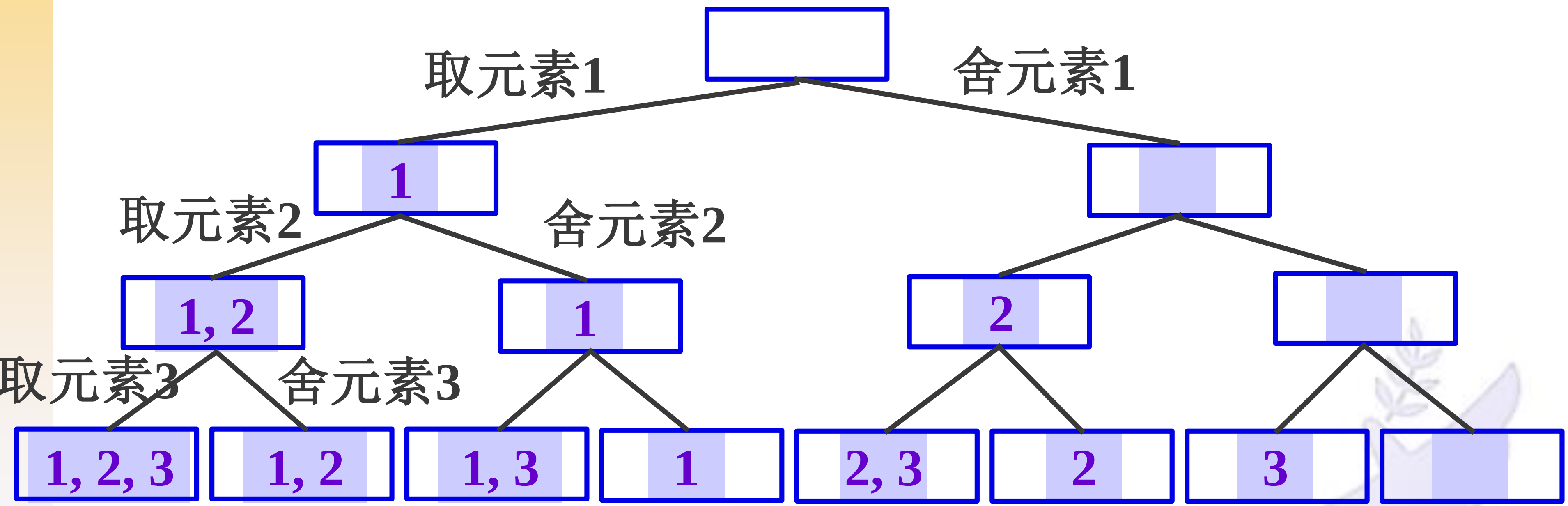
1. 树的遍历与回溯法

- 解空间
  - 问题求解是在其解的全部空间中找最优解或满足特定条件的解。
- 状态树
  - 求解过程实质是一个先序遍历一棵“状态树”的过程，只是这棵树不是预先建立的，而是隐含在遍历过程中。
- 例：
  - 幂集问题
  - TSP问题
  - n皇后问题
  - 最大团问题...



幂集问题

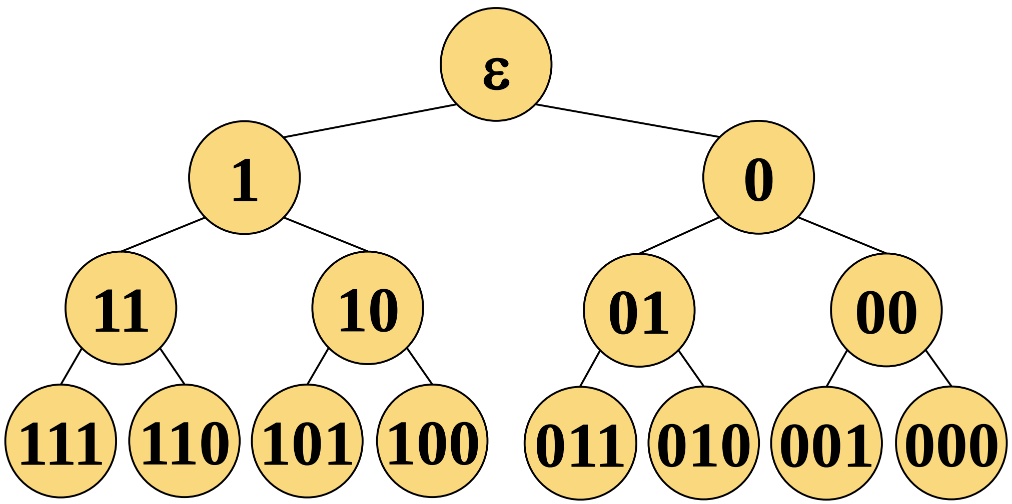
- 例：求含n个元素的集合的幂集。
  - 如 n = 3， A={1, 2, 3}
  - A的幂集为{{1, 2, 3}, {1, 2}, {1, 3}, {2, 3}, {1}, {2}, {3}, { }}



问题求解过程：遍历状态树，输出所有叶子节点

幂集问题

- 例：求含n个元素的集合的幂集。
  - 如 n = 3， A={1,2,3}
  - A的幂集为{{1, 2, 3}, {1, 2}, {1, 3}, {2, 3}, {1}, {2}, {3}, { }}



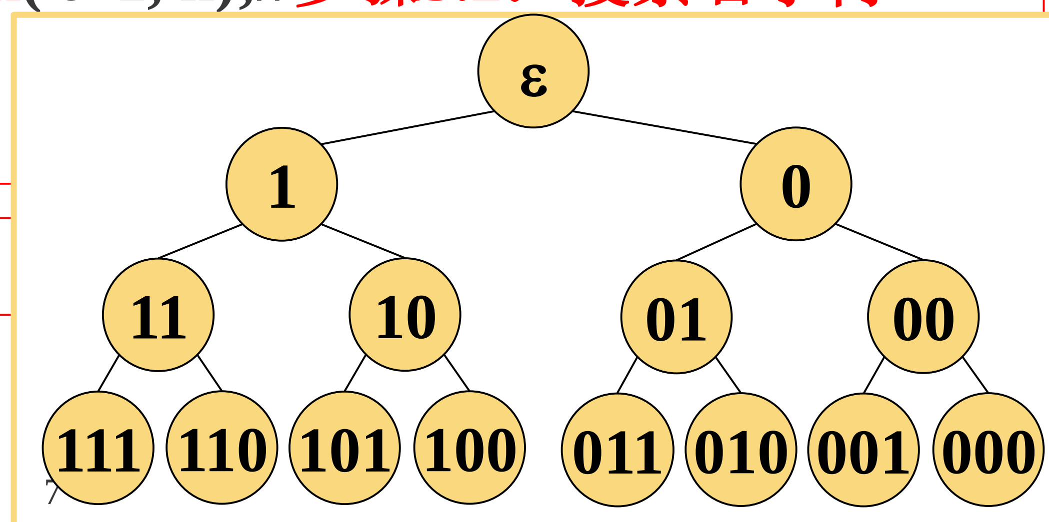
问题求解过程：遍历状态树，输出所有叶子节点

幂集问题-算法

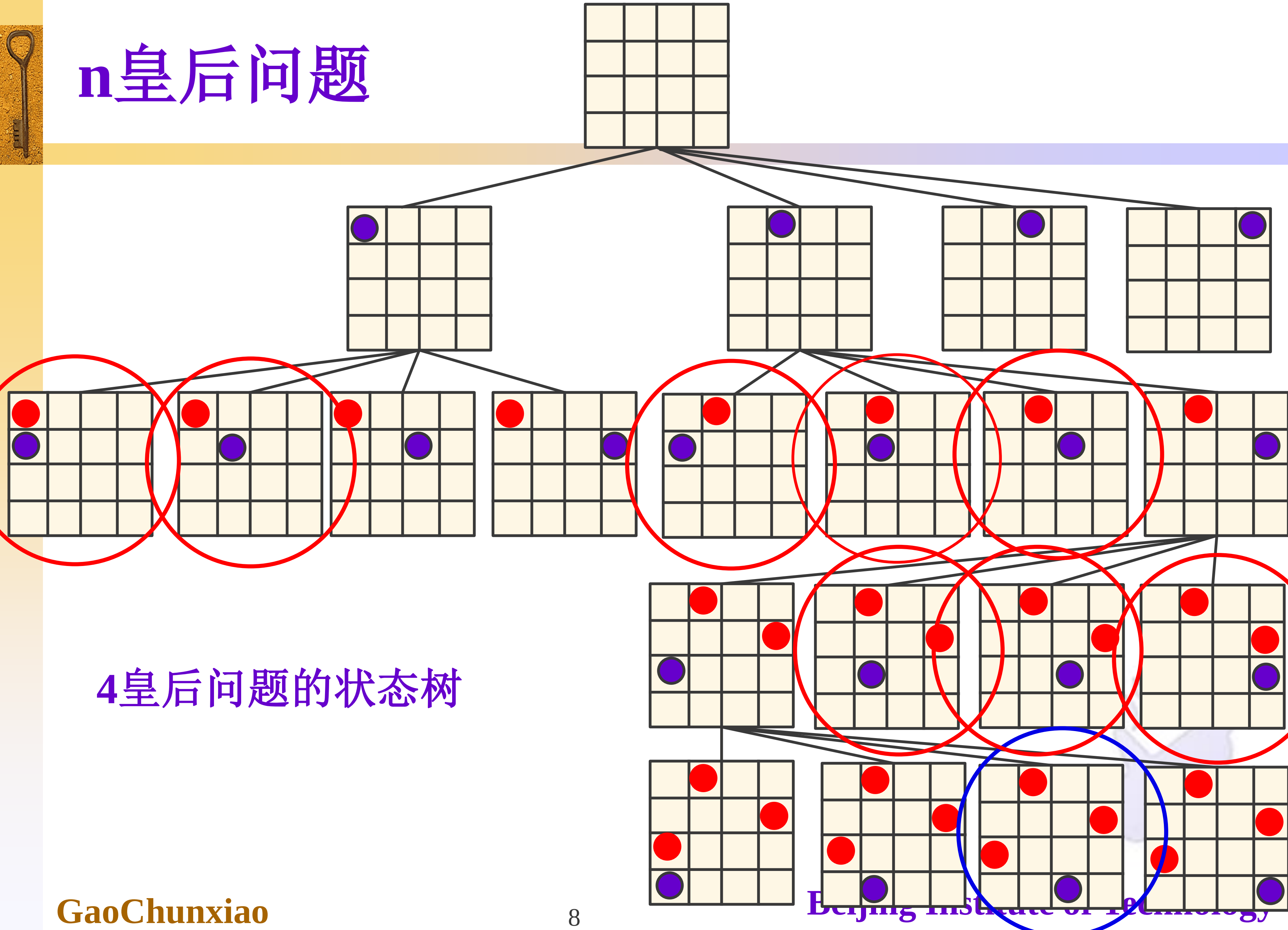
- 用x[ ] 表示棋局，例x = (1, 0, 1)表示集合{1, 3}

```
void backtrack( int t, int n){
        if (t > n ) Output(x);//步骤1：输出当前解(叶子节点)
        else{
                x[t] =1;//步骤2.1： “取”第t个元素
                backtrack( t+1, n);//步骤2.2：搜索左子树
                x[t] =0;//步骤3.1： “舍”第t个元素
                backtrack( t+1, n);//步骤3.2：搜索右子树
        }//else
}//backtrack
```

```
backtrack( 1, 3);
```



n皇后问题



4皇后问题的状态树

n皇后问题-分析

- 问题空间：4个棋子在棋盘上所有可能的摆放方式
- 求解过程：先根遍历状态树
- 访问当前节点操作：
  - 判断是否得到一个完整的布局（已摆了4个棋子）
  - 若是则输出该布局
  - 否则依次先根遍历各棵子树
    - 判断子树根节点布局是否合法
    - 若合法则遍历该子树
    - 否则剪枝

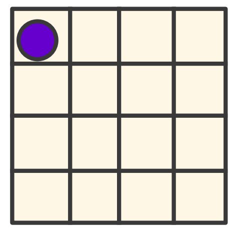
n皇后问题-算法

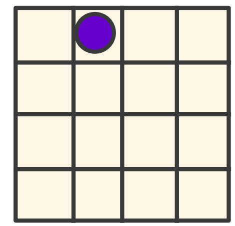
- 棋盘用p[ ][ ]表示，有棋子的位置设为1，否则设为0

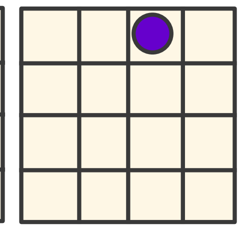
```
void Trial( int t, int n){
        if (t > n ) output( p);//步骤1：输出当前解(叶子节点)
        else for( j =1; j<=n; j++){ //步骤2：一行行放置棋子
        //即依次遍历各棵子树
                p[t][j] = 1;     //在第t行第j列放一个棋子;
                if( Place(p) ) //当前棋局不合法则剪枝
                        Trial( t+1, n);//合法，继续遍历
                p[t][j] = 0;//移走第t行第j列的棋子;
            }//for
}//Trial
```

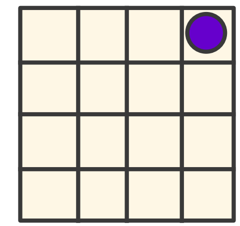
```
Trial(1, 4);
```

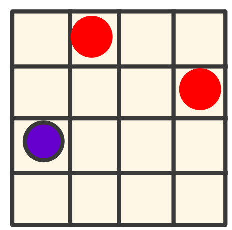
◆ 用x[ ] 表示棋局

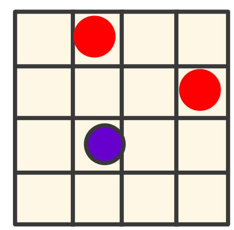

(1, 0, 0, 0)

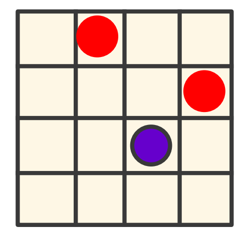

(2, 0, 0, 0)

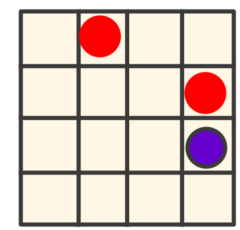

(3, 0, 0, 0)

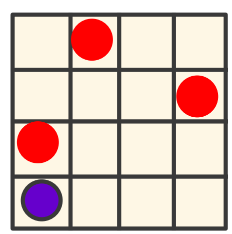

(4, 0, 0, 0)

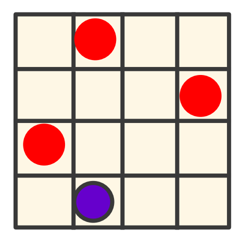

(2, 4, 1, 0)

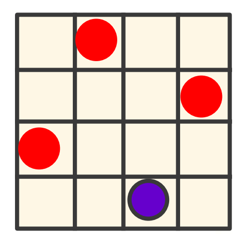

(2, 4, 2, 0)

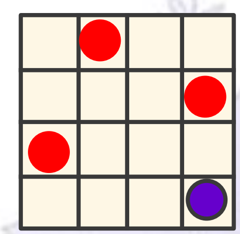

(2, 4, 3, 0)


(2, 4, 4, 0)


(2, 4, 1, 1)


(2, 4, 1, 2)


(2, 4, 1, 3)


(2, 4, 1, 4)

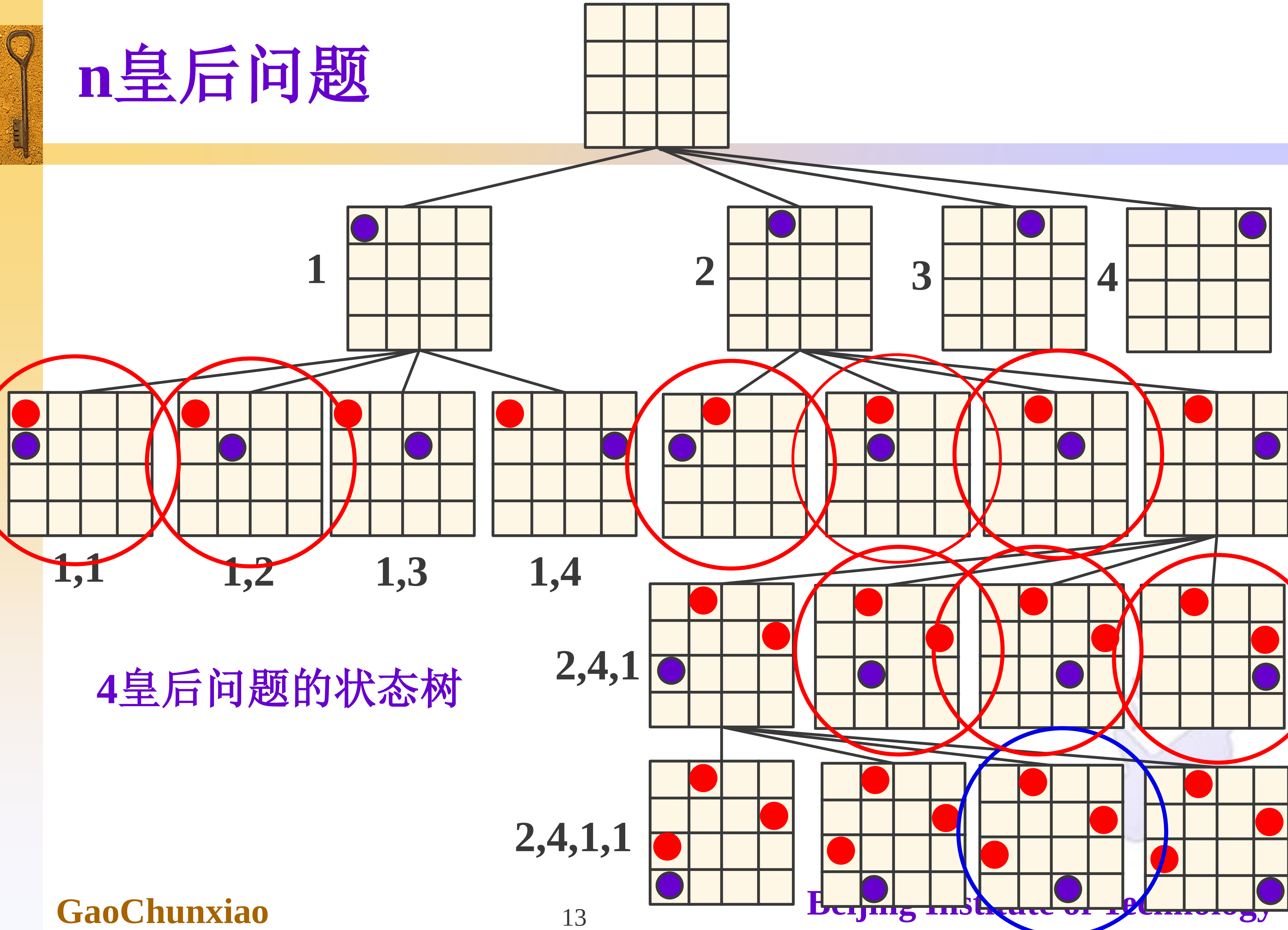
n皇后问题-算法

- 棋盘用x[ ]表示

```
void backtrack( int t, int n){
        if (t > n )
                output( x );// 步骤1：输出当前解(叶子节点)
        else for( j =1; j<=n; j++){//步骤2：一行行放置棋子
        //即依次遍历各棵子树
                x[t] = j;        //在第t行第j列放一个棋子;
                if( Place( x )) //当前棋局不合法则剪枝
                        backtrack( t+1, n);//合法，继续遍历
          }//for
}//backtrack
```

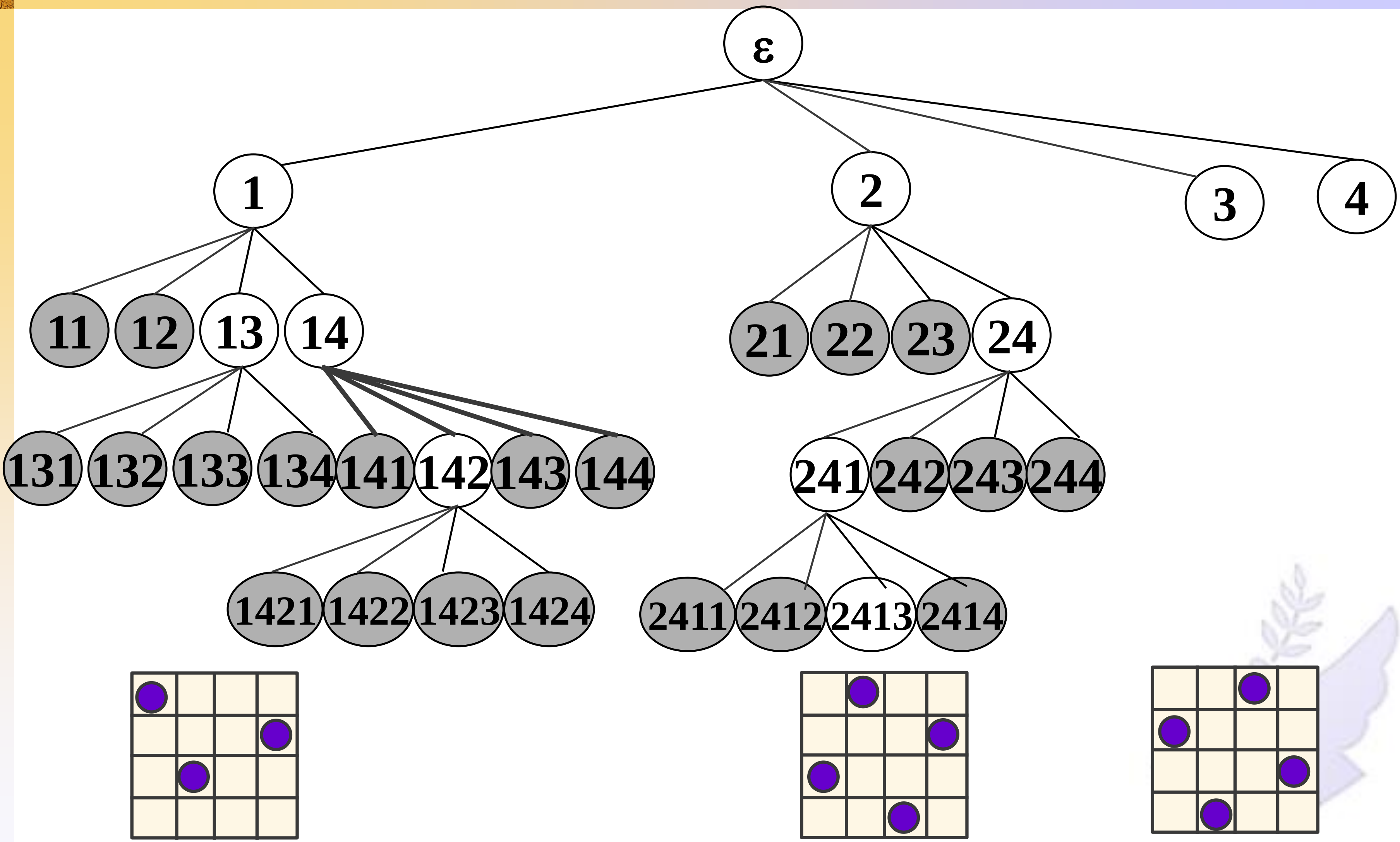
```
backtrack(1, 4);
```

n皇后问题



4皇后问题的状态树

n皇后问题


ε
1
2
3
4
11
12
13
14
21
22
23
24
131
132
133
134
141
142
143
144
241
242
243
244
1421
1422
1423
1424
2411
2412
2413
2414

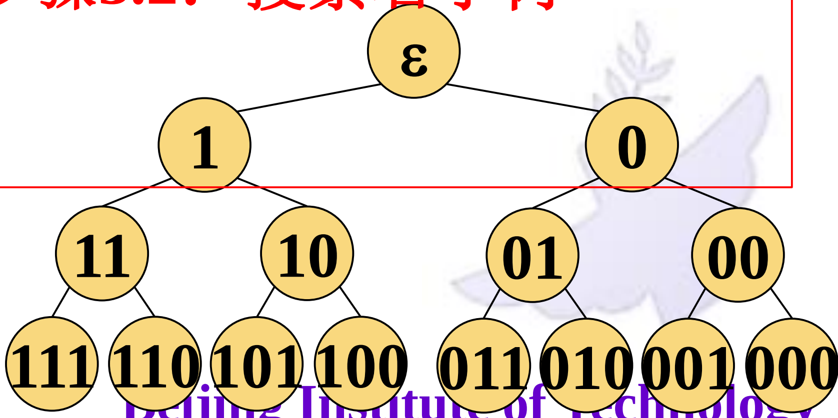
4、回溯算法的基本思想

- 回溯法的基本步骤：
  - (1) 针对所给问题，定义问题的解空间；
  - (2) 确定易于搜索的解空间结构；
  - (3) 设计约束函数和限界函数
  - (4) 以深度优先方式搜索解空间（先根遍历状态树），并在搜索过程中用剪枝函数避免无效搜索。
- 常用剪枝函数：
  - 用约束函数在扩展结点处剪去不满足约束的子树；
  - 用限界函数剪去得不到最优解的子树。

递归回溯

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack( int t, int n){
        if (t > n ) Output(x);//步骤1：输出当前解(叶子节点)
        else{
                x[t] =1;//步骤2.1：“取”第t个元素
                backtrack( t+1, n);//步骤2.2：搜索左子树
                x[t] =0;//步骤3.1：“舍”第t个元素
                backtrack( t+1, n);//步骤3.2：搜索右子树
        }//else
}//backtrack
```



递归回溯

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
    if (t>n) output(x);//步骤1：输出当前解(叶子节点)
    else for (int i=f(n, t);i<=g(n, t); i++)
        {//步骤2：依次先根遍历各棵子树
            x[t]=h(i); //加入的元素h(i)
            if (constraint(t)&&bound(t))
                  backtrack(t+1);
            x[t]=0; //必要时舍元素h(i)
        }
}
```

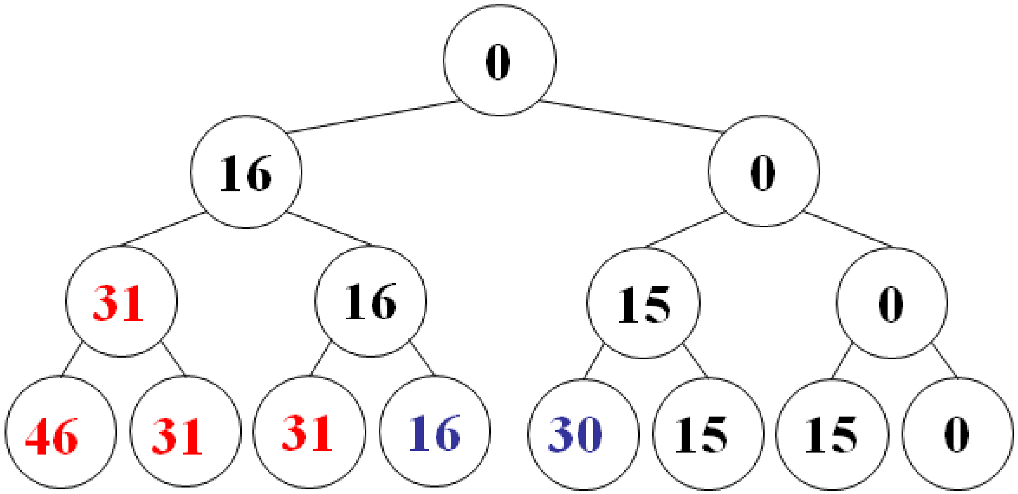
装载问题

- 装载问题定义
- n件货物(重w[1:n])装两艘船(载重量c_1,c_2), $\Sigma_{i=1}^{n}$ w[i] ≤ c_1+c_2, 是否有装载方案.
- 装载方案: 尽可能装满第1艘, 剩余的装第2艘
- 尽可能装满第1艘等价于下面变形的0-1背包
- 例: w=[16,15,15], c=30

$$\max \sum_{i=1}^{n} w[i]x[i]$$

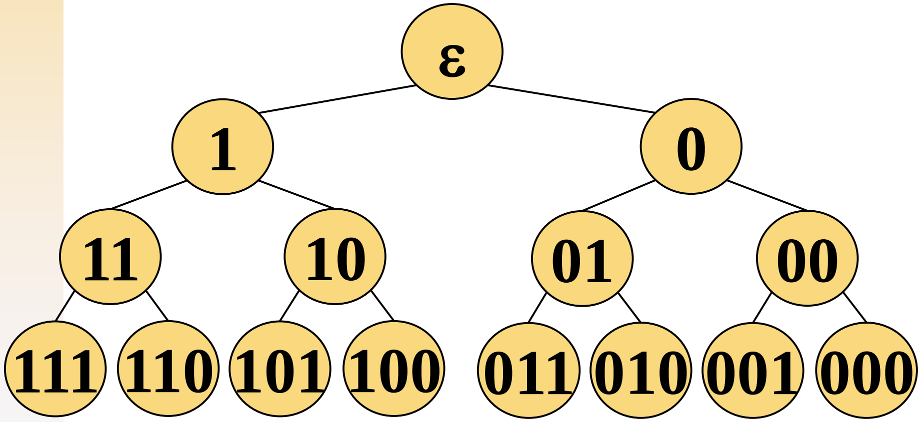
$$\text{s.t.} \sum_{i=1}^{n} w[i]x[i] \le c$$

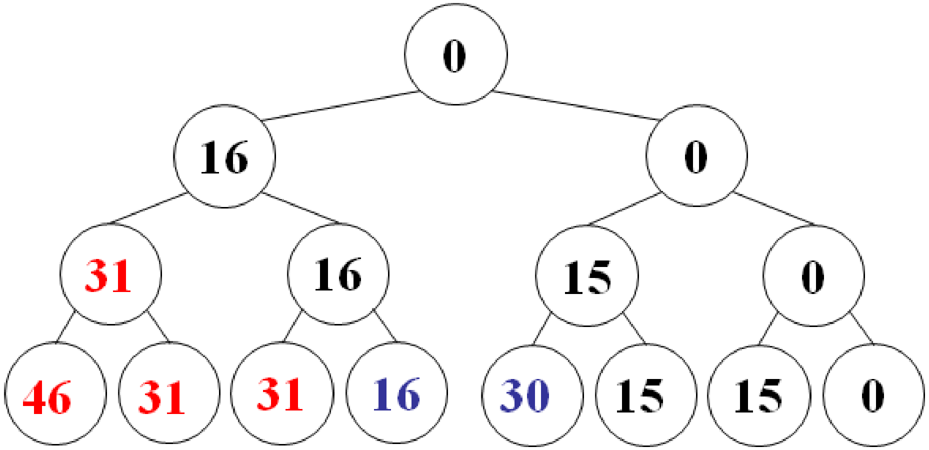
$$x[i] \in \{0,1\}, 1 \le i \le n$$



装载问题—分析

- 解空间：状态树
- 解的表示：*x*[ ]
- 约束函数：解*x*的总重量小于船的载重量*c*
- 求解过程：在树上进行深度优先搜索





w=[16,15,15], c=30

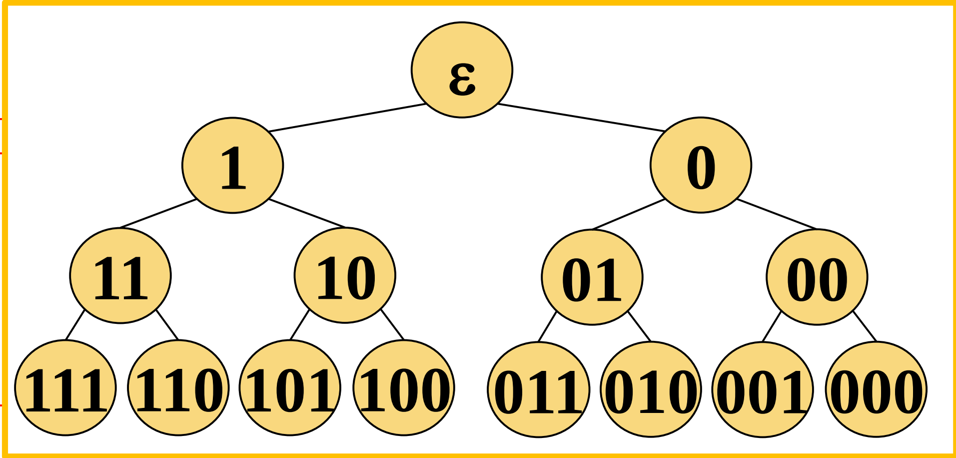
装载问题-算法

w=[16,15,15]，c=30

- 用x[ ] 表示解，例x = (1, 0, 1)表示集合{16, 15}

```
void backtrack( int t, int n){
        if (t > n )  //cw:当前重量，bestw:最优重量
                     if( cw<=c && cw>bestw) bestw=cw;
        else{
                x[t] =1;  cw+=w[t];
                backtrack( t+1, n);  cw-=w[t];
                x[t] =0; // "舍" 第t个元素
                backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

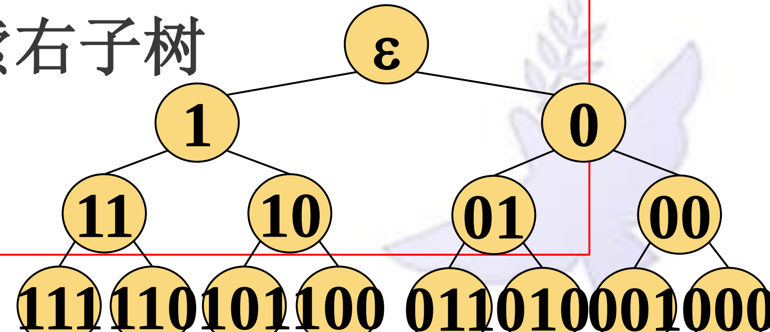
```
bestw=cw=0;

backtrack( 1, 3);
```



装载问题-算法

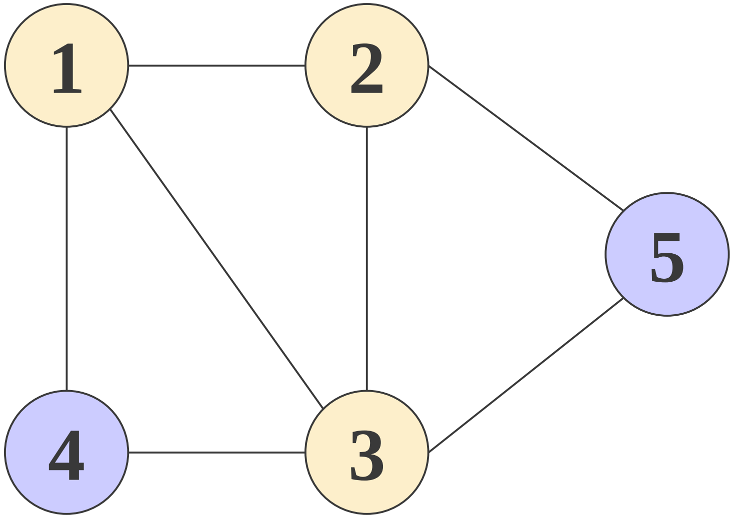
w=[16,15,15], c=30

```
void backtrack( int t, int n){
        if (t > n ) //cw:当前重量, bestw:最优重量
                if( cw<=c && cw>bestw) bestw=cw;
        else{
                if( cw+w[t]<=c ) {//若重量不超限则搜索，否则剪枝
                        x[t] =1; cw+=w[t];
                        backtrack( t+1, n);//搜索左子树
                        cw-=w[t];}
                x[t] =0; // “舍” 第i个元素
                backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```



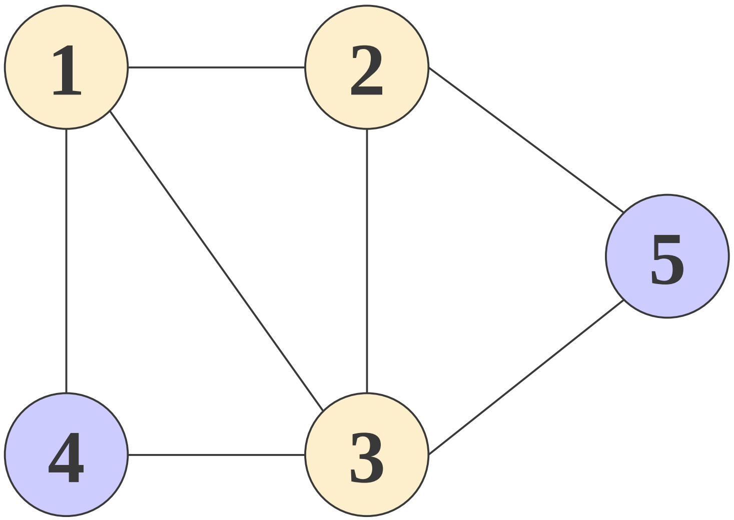
5、最大团问题

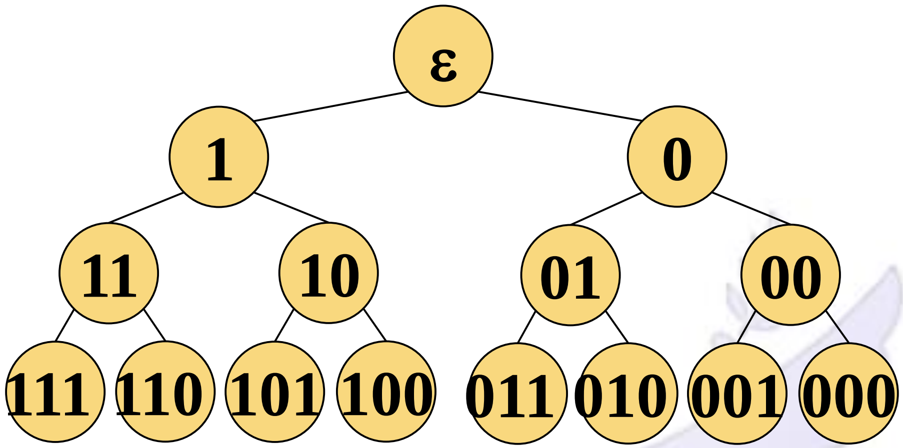
- **完全子图**：给定无向图G=(V, E)。如果U⊆V，且对任意u，v∈U有(u，v)∈E，则称U是G的完全子图。
- **团**：G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。
- **G的最大团**：是指G中所含顶点数最多的团。



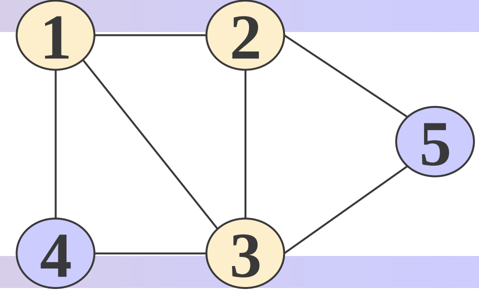
最大团问题 问题分析

- 解空间：状态树（子集树）
- 可行性约束函数：
  - 顶点*i*到已选入的顶点集中每一个顶点都有边相连。
- 限界函数：
  - 有足够多的可选择顶点使得算法有可能在右子树中找到更大的团





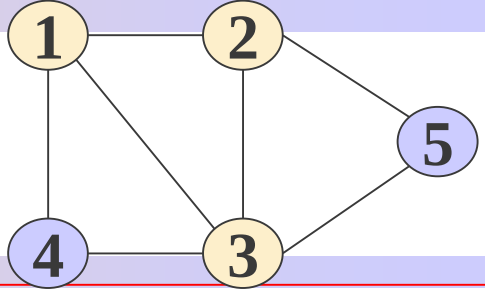
最大团问题-算法



- x[ ] 表示解，例x = (1, 1, 0, 0, 0)表示当前的团是{1, 2}
- bestx[ ]：当前最大团

```
void backtrack( int t, int n){
        if (t > n )  //cn: 当前团顶点数, bestn: 最大团顶点数
                     if(cn > bestcn){ bestn = cn; bestx[ ] = x[ ];}
        else{
          if( Clique(x, t)) {//若结点t可增大当前团则搜索,否则剪
                x[t] =1; cn++;
                backtrack( t+1, n);//搜索左子树
                cn--;}
              x[t] =0; // “舍” 第t个元素
              backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

最大团问题-算法



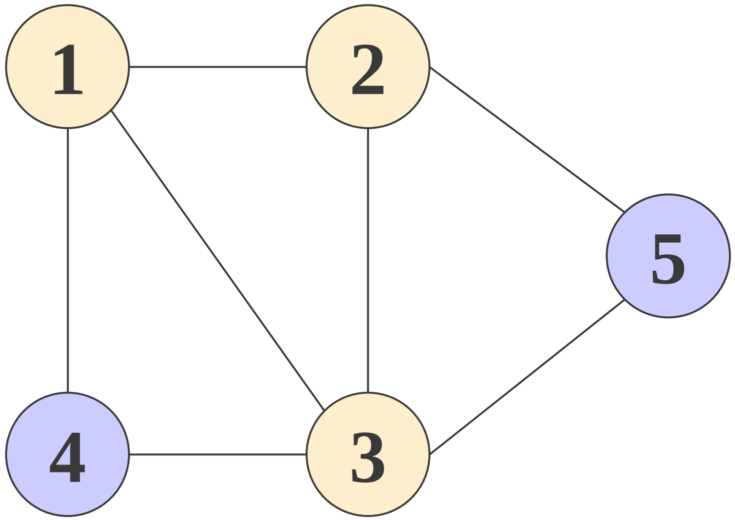

```
void backtrack( int t, int n){
   if (t > n ) {//cn: 当前团顶点数, bestn: 最大团顶点数
      if(cn > bestcn){ bestn = cn; bestx[ ] = x[ ]; }
      else{
         if( Clique(x, t)) {//若结点t可加入当前团则搜索，否则剪枝
            x[t] =1; cn++;
            backtrack( t+1, n);//搜索左子树
            cn--;
         }
         if (cn + n - t > bestn) {// 剪枝或进入右子树
                  x[t] = 0;
                  Backtrack(t+1);}
}//backtrack
```

//限界函数：有足够多的可选择顶点使得算法

最大团问题-算法

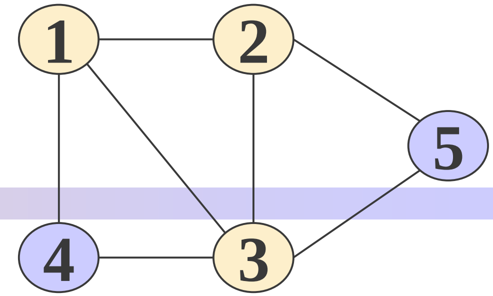
- 图如何表示？图的邻接矩阵

$$A_{ij} = \begin{cases} 0, (i, j) \notin VR \\ 1, (i, j) \in VR \end{cases}$$



0	1	1	1	0
1	0	0	0	1
1	1	0	1	1
1	0	1	0	0
0	1	1	0	0

最大团问题-算法



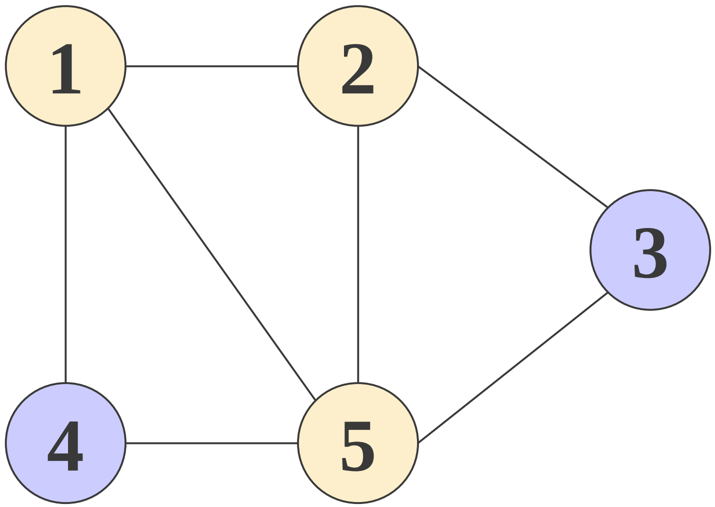
- Clique(x, t) //若结点t是否可加入当前团
- a[ ][ ]:图的邻接矩阵

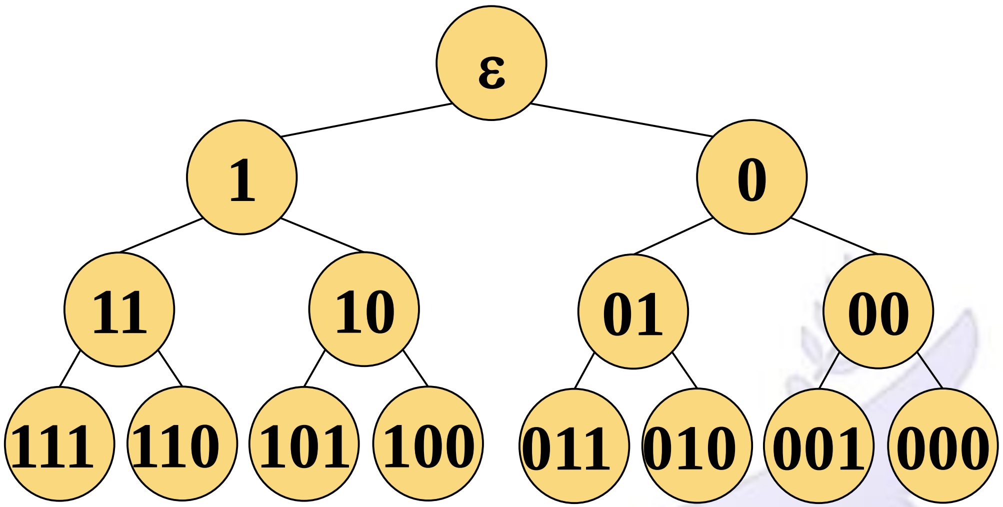
```
status Clique(x, t){
    int OK = true; // 检查v_t 与当前团的连接
    for (int j = 1; j < t && OK; j++)
        if (x[j] && a[t][j] == 0) // i与j不相连
            OK = false;
    return OK;
}
```

0	1	1	1	0
1	0	0	0	1
1	1	0	1	1
1	0	1	0	0
0	1	1	0	0

最大团问题-复杂度分析

- 在最坏情况下有 $O(2^n)$个结点需要计算可行性约束
- 计算可行性约束需要$O(n)$时间
- 回溯算法backtrack所需的计算时间为$O(n2^n)$。





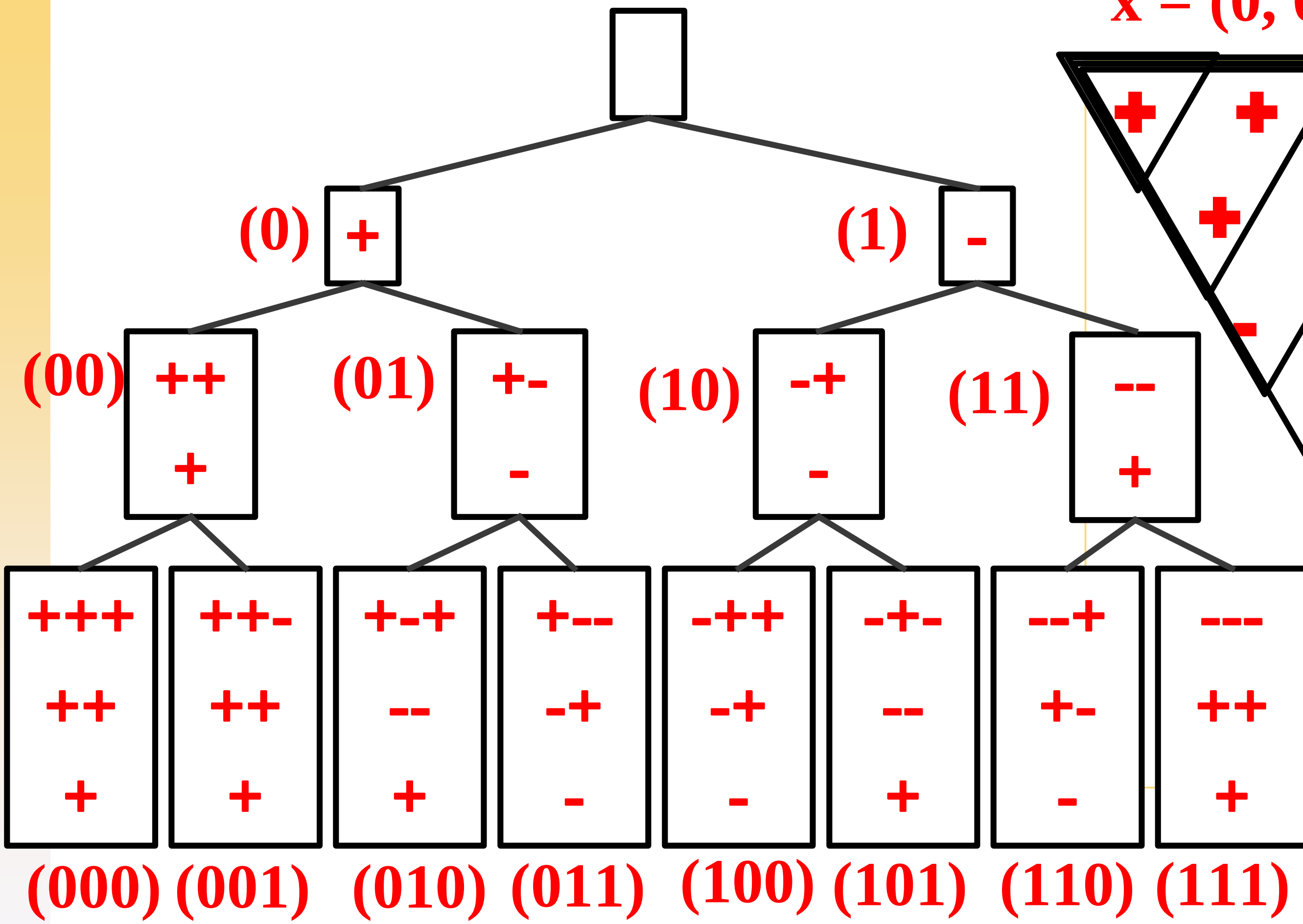
6. 符号三角形

- **问题**:
  - 设第一行有n个符号
  - 2个同号下面都是“+”，2个异号下面都是“-”。
  - 给定的n，求“+”“-”个数相同的符号三角形个数。
- **解向量**：用n元组x[1:n]表示符号三角形的第一行
- **无解的判断**:n*(n+1)/2为奇数
- **限界条件**: 0,1的个数都不能超过n(n+1)/4

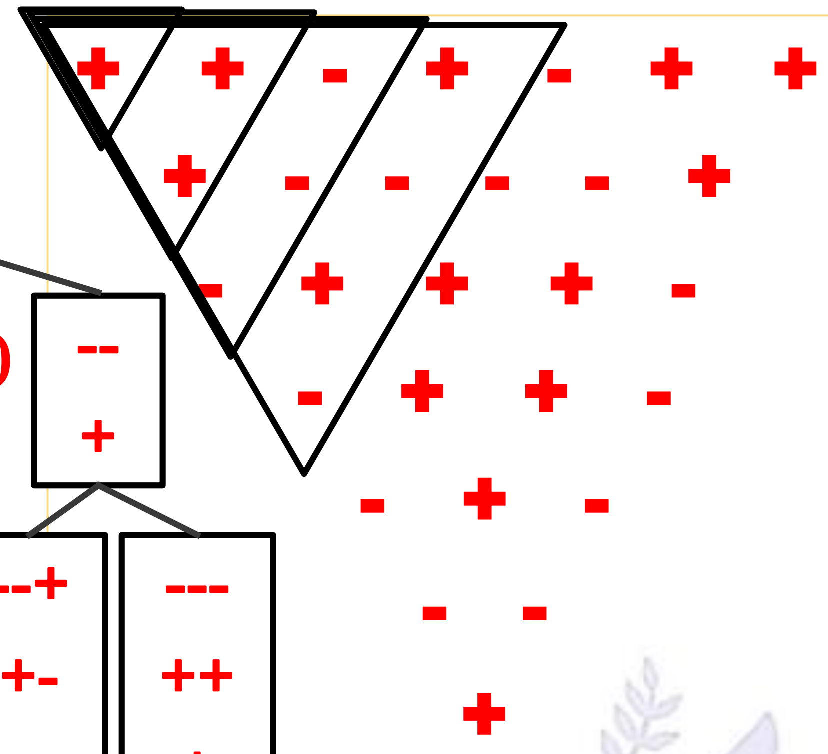
```
+   +   -   +   -   +   +
  +   -   -   -   -   +
    -   +   +   +   -
      -   +   +   -
        -   +   -
          -   -
            +
```

n = 7;
14个“+”，14个“-”

符号三角形-状态树



$x = (0, 0, 1, 0, 1, 0, 0)$



$n = 7;$

$n(n+1)/4 = 14$

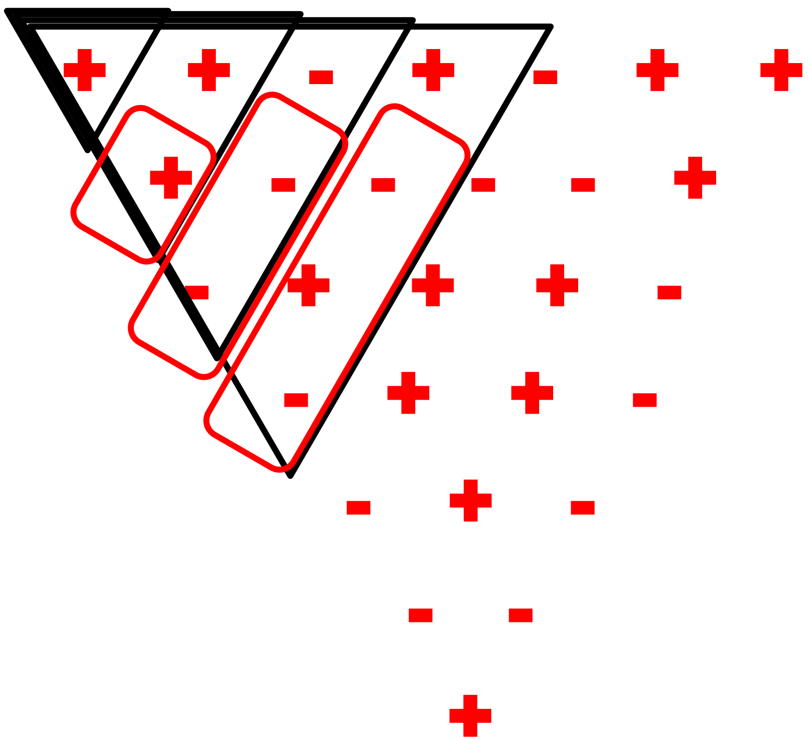
符号三角形-算法

- p[ ][ ]当前三角形

```
void Backtrack( int t){
    if (t>n) sum++;//得到一个符合条件三角形
    else
        for (int i=0;i<2;i++) {  //i=0或者1，即“+”或“-”
                p[1][t]=i; //p[ ][ ]当前三角形，1行t列取符号i
                count+=i; //累计p中1的个数
                计算p并累计p中新增1的个数count1;
                count += count1;
                Backtrack( t+1);
                count -= count1; count-=i;
        }//for (int i=0;i<2;i++)
}//Backtrack
```

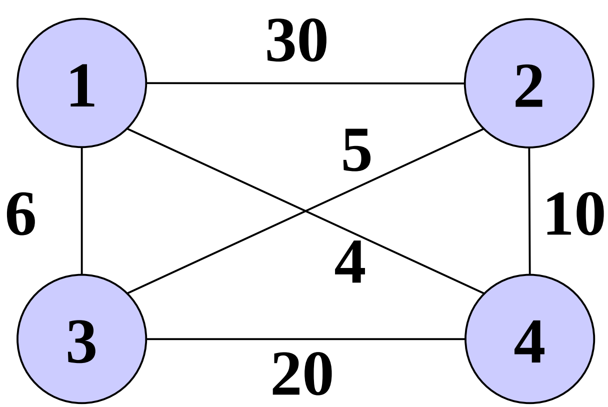
符号三角形-复杂度分析

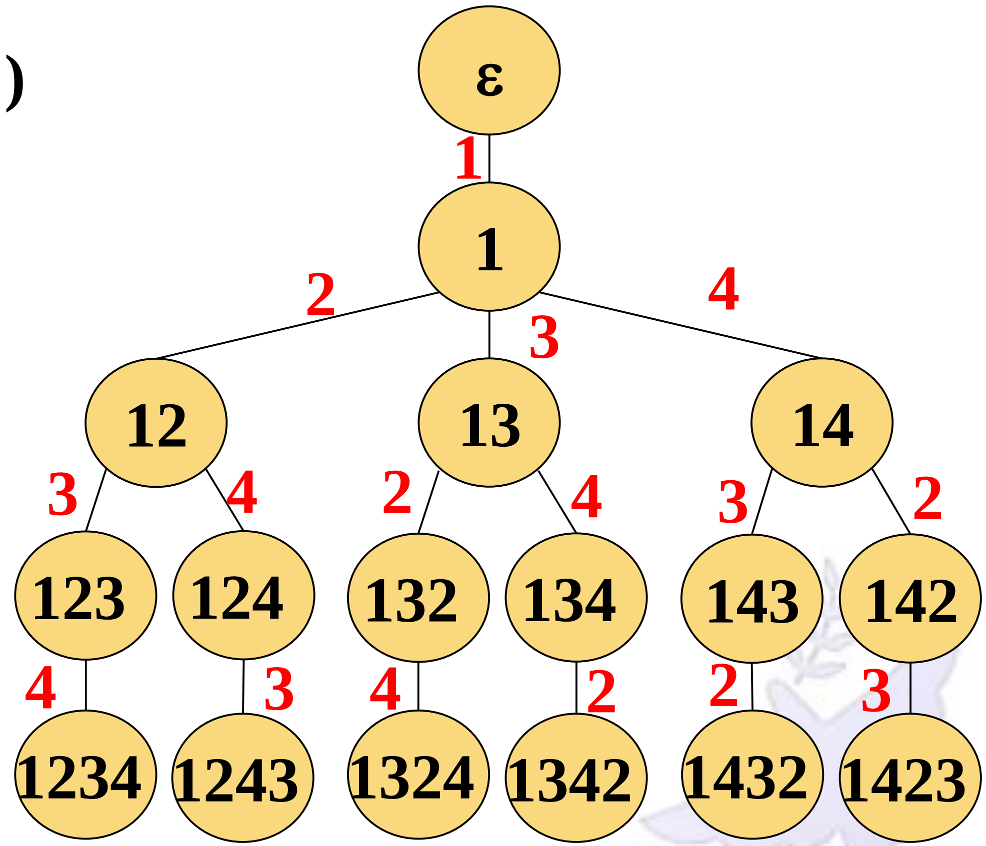
- 计算可行性约束需要O(n)时间
- 在最坏情况下有 $O(2^n)$个结点需要计算可行性约束
- 解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$。



旅行商问题(TSP)-排列树

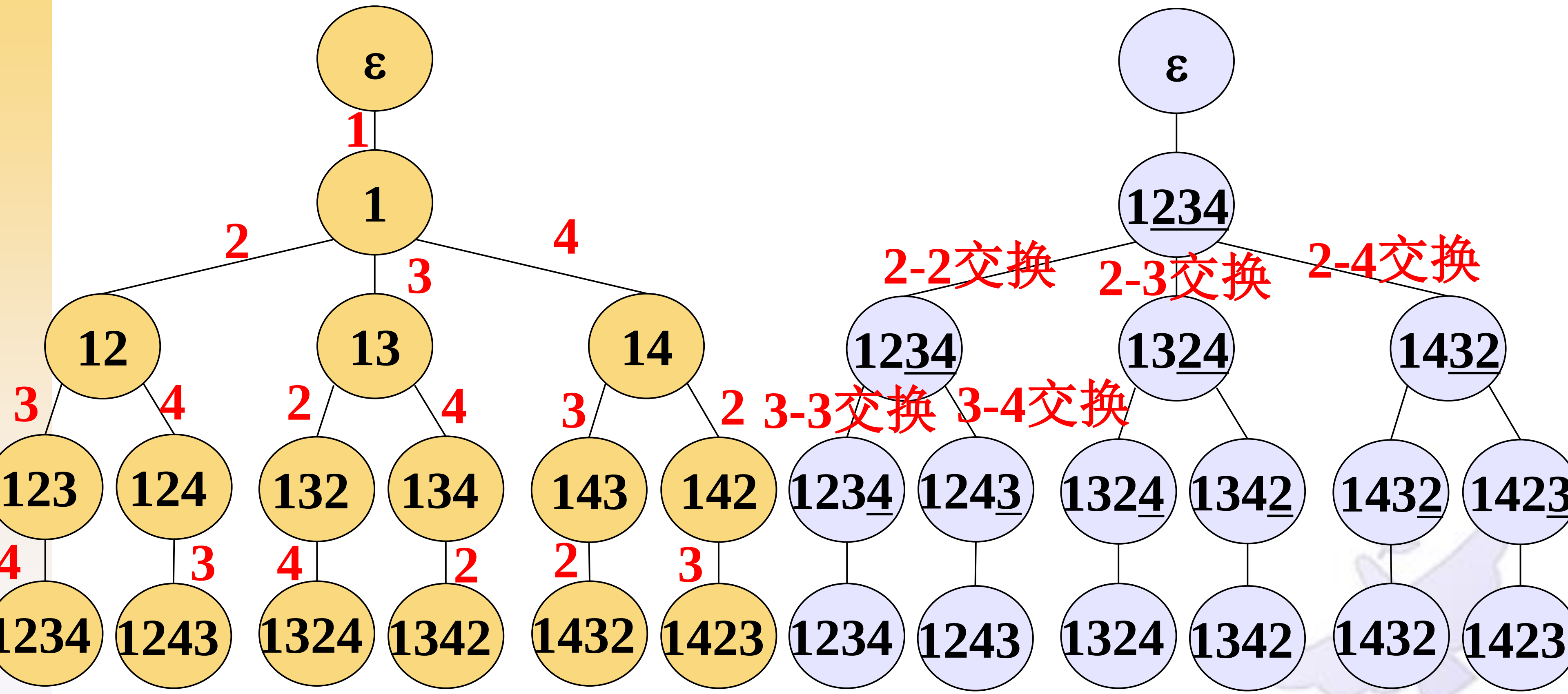
- 问题：求一条从某城市出发, 经过每个城市最后回到起点的回路, 使总距离最小
- 解空间: 全体排列((n-1)!)
- 解空间结构: 排列树





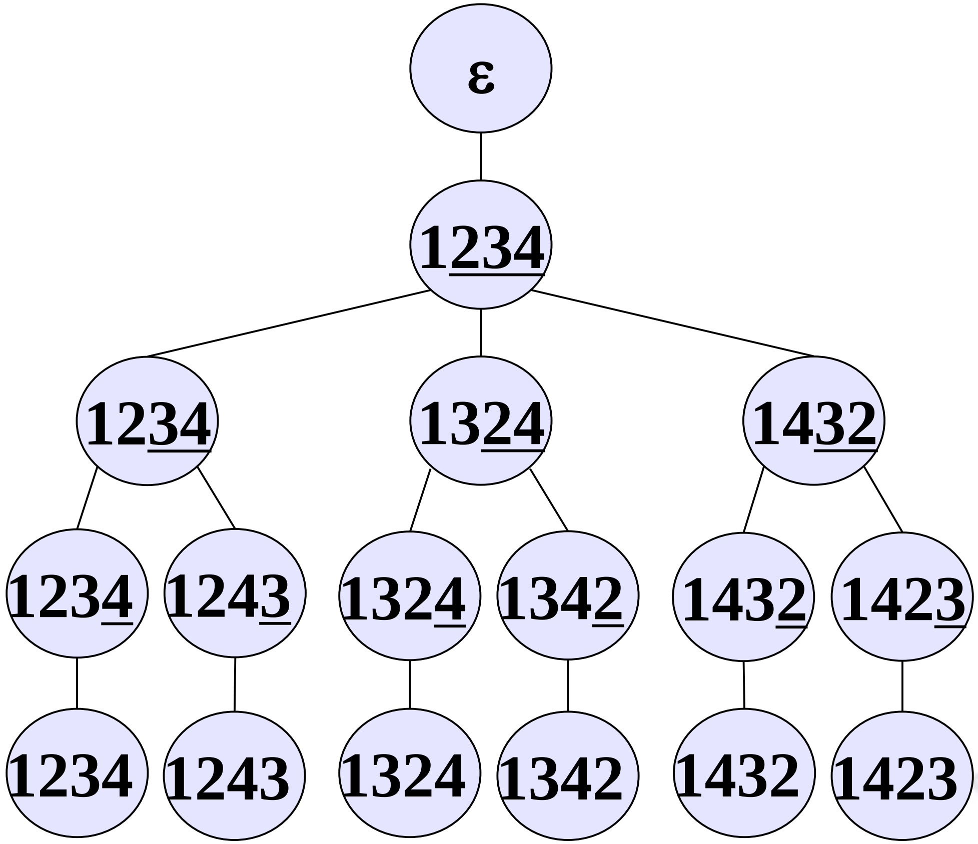
旅行商问题(TSP)

- 解向量：用n元组x[1:n]表示当前排列



旅行商问题(TSP)

- 约束函数：无
- 限界条件:当前解的长度超过了已知最优解的长度



TSP-算法

```
void Backtrack (int i) {
   if (i == n) //步骤1：比较记录当前最优解
      if ((cc + a[x[n-1]][x[n]] + a[x[n]][x[1]] < bestc || bestc == 0)) {
         for (int j = 1; j <= n; j++) bestx[j] = x[j];
         bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}//if ((cc+
   else {// 步骤2：搜索子树i到n
     for (int j = i; j <= n; j++) //先判断是否进入x[j]子树?
        if ((cc+a[x[i-1]][x[i]] < bestc || bestc == 0)) {//不满足则剪枝
            Swap(x[i], x[j]);
            cc += a[x[i-1]][x[i]];
            Backtrack(i+1);
            cc -= a[x[i-1]][x[i]];
            Swap(x[i], x[j]);   }
   }//else
}//Backtrack
```

x[]: 当前解
cc：当前解的距离
bestx[]: 当前最优解
bestc: 当前最短距离

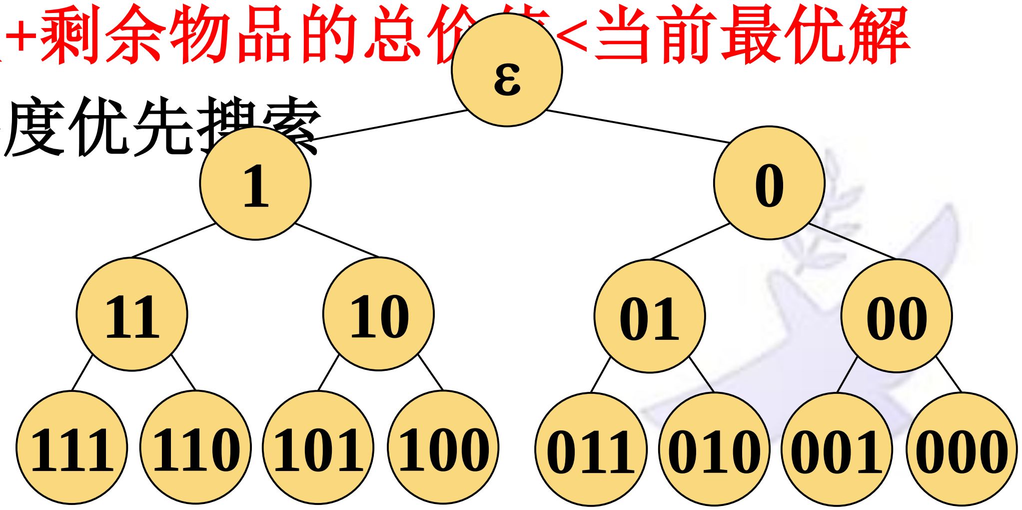
TSP-算法复杂度

- 复杂度分析：
- 排列树有(n-1)!个叶子节点
  - backtrack在最坏情况下需要更新当前最优解O((n-1)!)次
  - 每次更新bestx需计算时间O(n)
  - 从而整个算法的计算时间复杂性为O(n!)。

7. 0-1背包回溯 $O(2^n)$

- 输入: n物品重w[1:n], 价值v[1:n], 背包容量C
- 输出: 装包使得价值最大.

- 解空间：状态树
- 解的表示：*x*[ ]
- 约束函数：解*x*的总重量小于背包容量C
- 限制函数：当前解的价值+剩余物品的总价值<当前最优解
- 求解过程：在树上进行深度优先搜索



0-1背包回溯 O(2^n)

r:当前剩余价值
初始:r=$sum_{t=1}^{n}$ v[t]

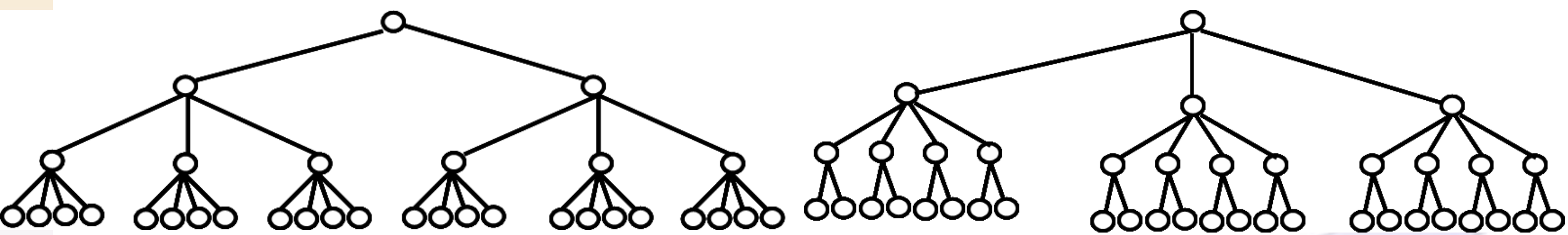
```
void backtrack( int t, int n){
        if (t > n ) //cw:当前重量， cv:当前价值, bestv:最大价值,
                if( cw<=c && cv>bestv) bestv=cv;
        else{   if( cw+w[t]<=c ) {//若重量不超限则搜索， 否则剪枝
                x[t] =1; cw+=w[t];  cv+=v[t];r -= v[t];
                backtrack( t+1, n);//搜索左子树
                cw-=w[t];  r += v[t];  }
                x[t] =0; // “舍” 第i个元素
                if( cv + r ) > bestv
                    backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

0-1背包-提前更新最优解

```
void backtrack( int t, int n){
        if (t > n ) //cw:当前重量，cv:当前价值，bestv:最大价值,
                if( cw<=c && cv>bestv) bestv=cv;
        else{   if( cw+w[t]<=c ) {//若重量不超限则搜素，否则剪枝
                x[t] =1; cw+=w[t]; cv+=v[t];r -= v[t];
                if( cv>bestv) bestv=cv;
                backtrack( t+1, n);//搜索左子树
                cw-=w[t];  r += v[t];  }
                x[t] =0; // "舍"第i个元素
                if( cv + r ) > bestv
                    backtrack( t+1, n);//搜索右子树

        }//else
}//backtrack
```

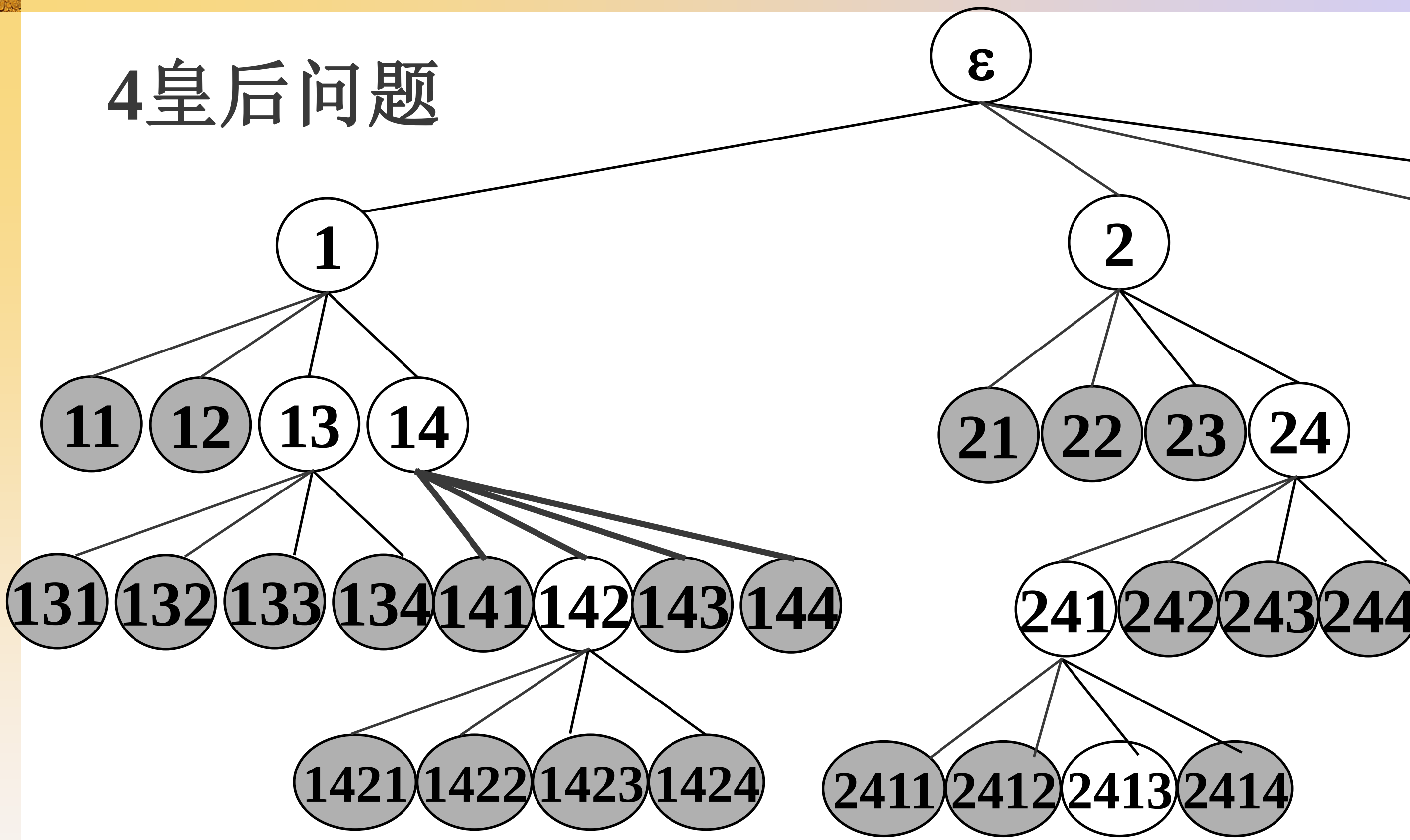
8. 回溯法的效率分析

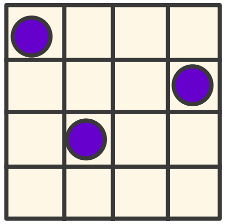
- 影响回溯算法效率的因素
  - 1. 每个顶点的产生时间
  - 2. 计算剪枝函数的时间
  - 3. 剪枝后剩余顶点个数
- 剪枝函数的设计与平衡? 更好的剪枝会增加计算时间
- 重排原理：优先搜索取值最少的x[i]



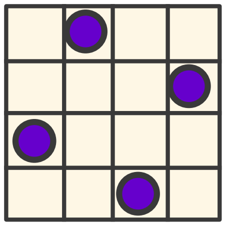
回溯法的效率举例-n皇后问题

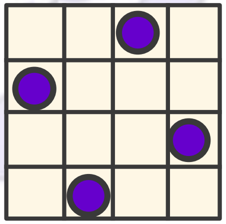
4皇后问题





扩展节点： 61
叶子节点： 16





回溯法的效率举例-n皇后问题

- 问题的规模：C(16, 4) =43680

- 状态树：约束1：棋子不放在同行中
- 不剪枝：
  - 叶子节点 4^4 = 256
  - 状态树的扩展节点：$\sum_{i=0}^{4} 4^{i}$=341
- 剪枝：约束2：棋子不同列且不同斜线
  - 叶子节点: 16，解的个数:2
  - 状态树的扩展节点：61

回溯法的效率举例-n皇后问题

- **8皇后问题**
- 教材中对8皇后问题的效率进行了概率估计
- 假设只需要求出一个合法的解,扩展节点/总节点 大约为1.55%
- 搜索节点数估计:
- 记第i层x[i]的可选列数为m_i,
- 随机选一可选列进入下一层
- 得节点数$1+m_1+m_1m_2+m_1m_2m_3+\ldots$
- 多次取平均得1702
- **m_1=8; m_2=5: 因为2可选4-8列; m_3=4: 因为3可选1,6,7,8; m_4=3: 因为4可选3,7,8; m_5=2: 因为5可选5,8;**

		1					
					2		
	3						
						4	
5							
			6				
							7
				8			

2329

END

思考题：运动员最佳配对问题

- 问题描述: 羽毛球队有男女运动员各n人. 给定2个n×n矩阵P和Q.
- P[i][j]是男运动员i与女运动员j配混合双打的男运动员竞赛优势;
- Q[i][j]是女运动员i与男运动员j配混合双打的女运动员竞赛优势.
- 由于技术配合和心理状态等各种因素影响, P[i][j]不一定等于Q[j][i]. 男运动员i和女运动员j配对的竞赛优势是P[i][j]*Q[j][i]. 设计一个算法, 计算男女运动员最佳配对法, 使得各组男女双方竞赛优势的总和达到最大.
- 数据输入: 正整数n(1≤n≤20)，P和Q
- 结果输出: 最佳配对的各组男女双方竞赛优势总和
- 例如：n=3，
- P = 10 2 3   Q= 2 2 2
  2 3 4   3 5 3
  3 4 5   4 5 1

$$P = \begin{matrix} 10 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{matrix} \qquad Q = \begin{matrix} 2 & 2 & 2 \\ 3 & 5 & 3 \\ 4 & 5 & 1 \end{matrix}$$

- 结果为52.