算法设计与分析

回溯法

讲师：杨根晖 博士

本讲内容

5.1 回溯法概述

5.2 求解0/1背包问题

5.3 求解装载问题

5.4 求解子集和问题

5.5 求解*n*皇后问题

5.6 求解图的*m*着色问题

5.7 求解任务分配问题

5.8 求解活动安排问题

5.9 求解流水作业调度问题

5.1 回溯法概述

回溯法实际上一个类似穷举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”（即回退），尝试别的路径。

5.1.1 问题的解空间

一个复杂问题的解决方案是由若干个小的决策步骤组成的决策序列，解决一个问题的所有可能的决策序列构成该问题的**解空间**。

应用回溯法求解问题时，首先应该明确问题的解空间。解空间中满足约束条件的决策序列称为**可行解**。

一般来说，解任何问题都有一个目标，在约束条件下使目标达到最优的可行解称为该问题的**最优解**。

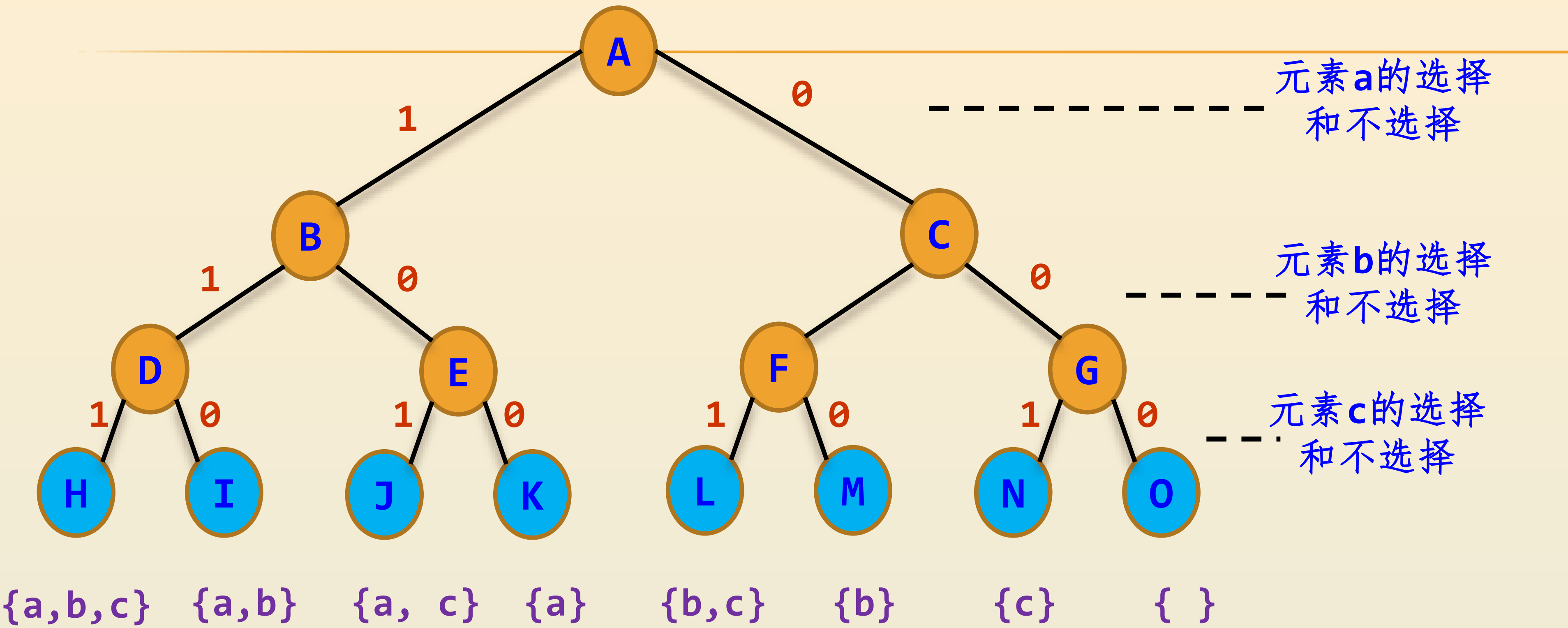
问题的解由一个不等长或等长的解向量$X=\{x_1，x_2，…，x_n\}$组成，其中分量x_i表示第i步的操作。

所有满足约束条件的解向量组构成了问题的解空间。

问题的解空间一般用树形式来组织，也称为解空间树或状态空间，树中的每一个结点确定所求解问题的一个问题状态。

树的根结点位于第1层，表示搜索的初始状态，第2层的结点表示对解向量的第一个分量做出选择后到达的状态，以此类推。

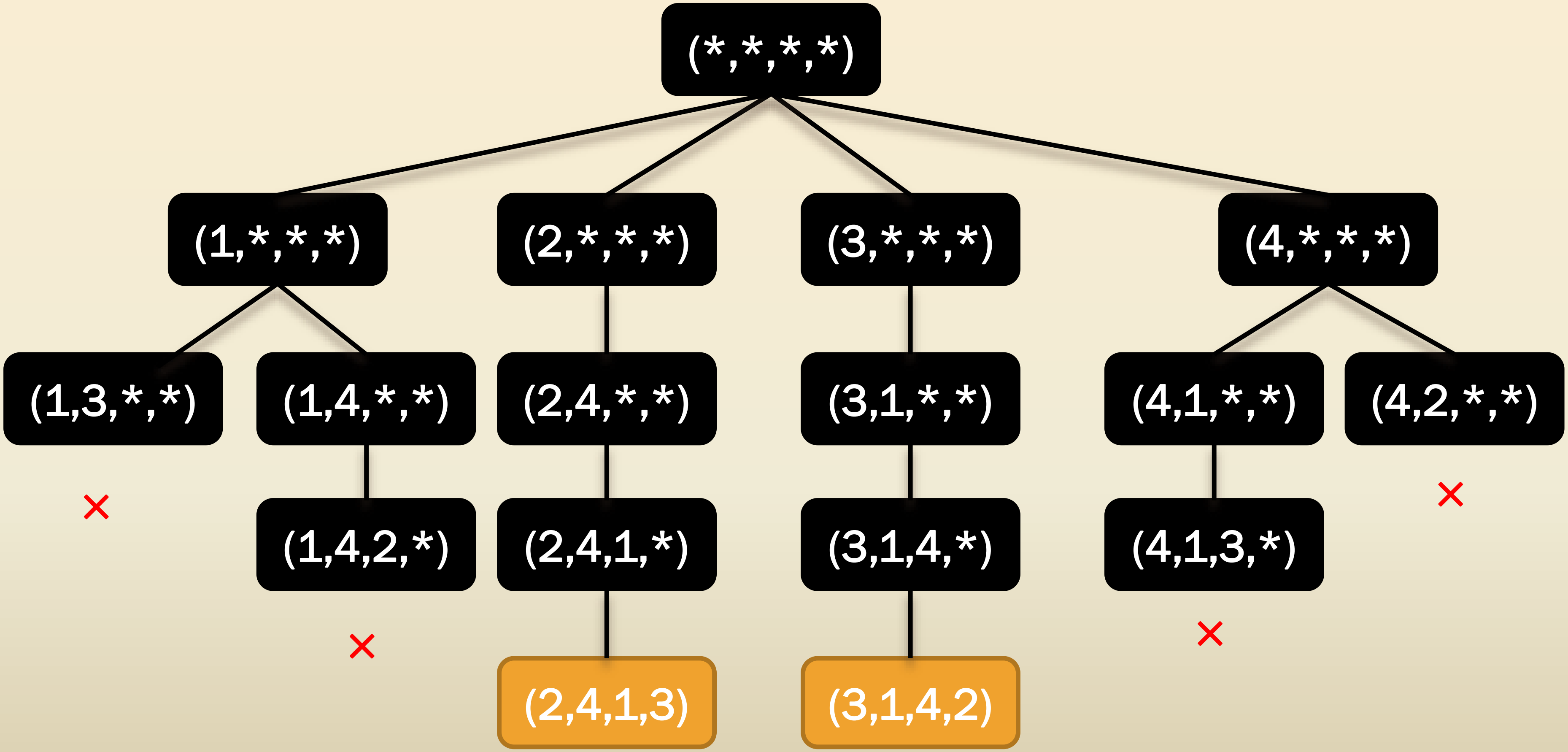
例如，以下求集合{*a*，*b*，*c*}的幂集的解空间树：



求解过程分为3步，分别对*a*、*b*、*c*元素做决策，该解空间的每个叶子结点都构成一个解（很多情况并非如此）。

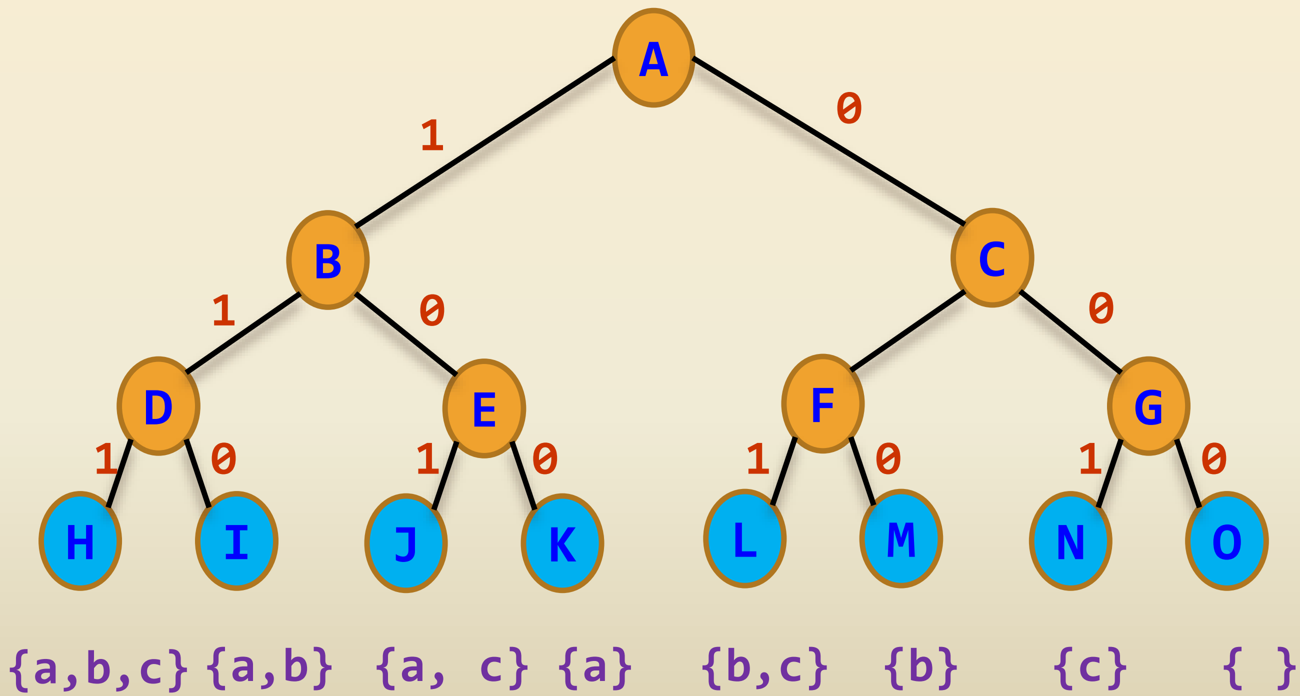
在一个解空间中搜索解的过程构成**搜索空间**，上图中所有叶子结点都是解，所以该问题的解空间和搜索空间相同。

下图是四皇后问题的搜索空间，图中每个状态由当前放置的皇后的行列号构成。它给出了四皇后问题的全部搜索过程，只有18个结点，其中标有☒号的结点无法继续扩展。

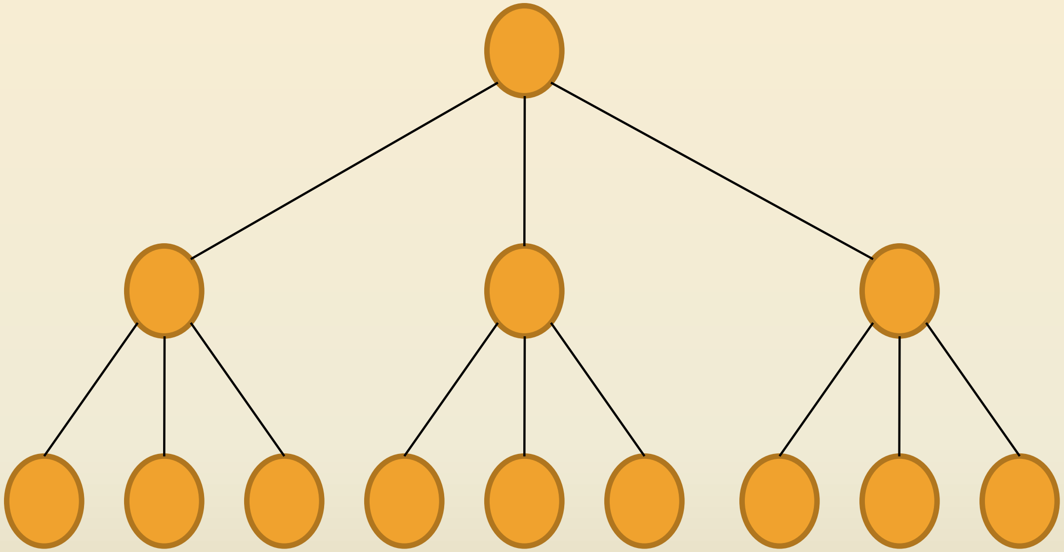


解空间树通常有两种类型：

- **子集树：** 当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树称为子集树。



- **排列树：** 当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间树称为**排列树**。



注意：问题的解空间树是虚拟的，并不需要在算法运行时构造一棵真正的树结构，然后再在该解空间树中搜索问题的解，而是只存储从根结点到当前结点的路径。

实际上，有些问题的解空间因过于复杂或状态过多难以画出来。

5.1.2 什么是回溯法

在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点（开始结点）出发搜索解空间树。

首先根结点成为活结点（活结点是指自身已生成但其孩子结点没有全部生成的结点），同时也成为当前的扩展结点（扩展结点是指正在产生孩子结点的结点）。

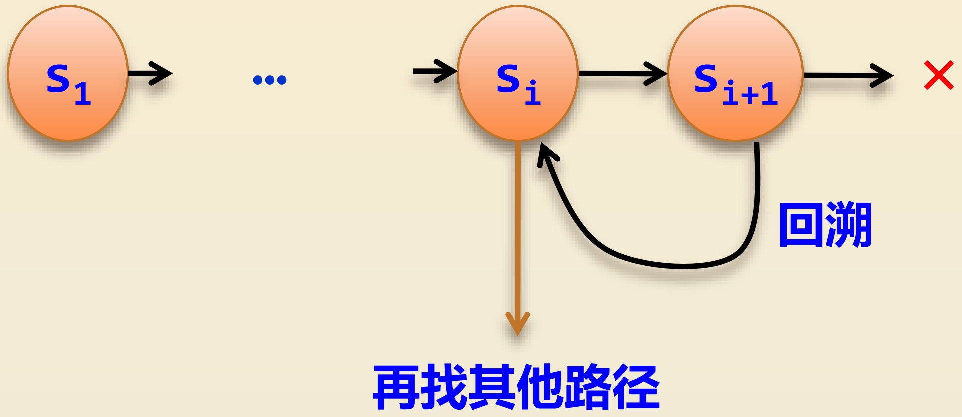
在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。

如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点（死结点是指由根结点到该结点构成的部分解不满足约束条件，或者其子结点已经搜索完毕）。

此时应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。

回溯法以这种方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点为止。

当从状态s_i搜索到状态s_{i+1}后，如果s_{i+1}变为死结点，则从状态s_{i+1}回退到s_i，再从s_i找其他可能的路径，所以回溯法体现出走不通就退回再走的思路。



若用回溯法求问题的所有解时，需要回溯到根结点，且根结点的所有可行的子树都要已被搜索完才结束。而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

回溯法搜索解空间时，通常采用两种策略避免无效搜索，提高回溯的搜索效率：

- 用约束函数在扩展结点处剪除不满足约束的子树；
- 用限界函数剪去得不到问题解或最优解的子树。

这两类函数统称为剪枝函数。

归纳起来，用回溯法解题的一般步骤如下：

① 针对所给问题，确定问题的解空间树，问题的解空间树应至少包含问题的一个（最优）解。

② 确定结点的扩展搜索规则。

③ 以深度优先方式搜索解空间树，并在搜索过程中可以采用剪枝函数来避免无效搜索。

5.1.3 回溯法的算法框架

1. 非递归回溯框架

```
int x[n];                                    //x存放解向量，全局变量
void backtrack(int n)                        //非递归框架
{   int i=1;                                 //根结点层次为1
    while (i>=1)                             //尚未回溯到头
    {   if(ExistSubNode(t))                  //当前结点存在子结点
        {   for (j=下界;j<=上界;j++)         //对于子集树，j=0到1循环
            {   x[i]取一个可能的值;
                if (constraint(i) && bound(i))
                                             //x[i]满足约束条件或界限函数
                {   if (x是一个可行解)
                        输出x;
                    else   i++;              //进入下一层次
                }
            }
        }
        else  i--;                           //回溯：不存在子结点，返回上一层
    }
}
```

2. 递归的算法框架

(1) 解空间为子集树

```
int x[n];                                    //x存放解向量, 全局变量
void backtrack(int i)                        //求解子集树的递归框架
{   if(i>n)                                  //搜索到叶子结点,输出一个可行解
        输出结果;
    else
    {   for (j=下界;j<=上界;j++)             //用j枚举i所有可能的路径
        {   x[i]=j;                          //产生一个可能的解分量
            …                                //其他操作
            if (constraint(i) && bound(i))
                backtrack(i+1);              //满足约束条件和限界函数,继续下一层
        }
    }
}
```

【例5.3】有一个含n个整数的数组a，所有元素均不相同，设计一个算法求其所有子集（幂集）。

例如，a[]={1，2，3}，所有子集是：{}，{3}，{2}，{2，3}，{1}，{1，3}，{1，2}，{1，2，3}（输出顺序无关）。

解： 显然本问题的解空间为子集树，每个元素只有两种扩展，要么选择，要么不选择。

采用深度优先搜索思路。解向量为x[]，x[i]=0表示不选择a[i]，x[i]=1表示选择a[i]。

用i扫描数组a，也就是说问题的初始状态是（i=0，x的元素均为0），目标状态是（i=n，x为一个解）。从状态（i，x）可以扩展出两个状态：

- 不选择a[i]元素 ⇨ 下一个状态为（i+1，x[i]=0）。
- 选择a[i]元素 ⇨ 下一个状态为（i+1，x[i]=1）。

```
void dfs(int a[],int n,int i,int x[])
//回溯算法求解向量x
{  if (i>=n)
      dispasolution(a,n,x);
   else
   {  x[i]=0; dfs(a,n,i+1,x);        //不选择a[i]
      x[i]=1; dfs(a,n,i+1,x);        //选择a[i]
   }
}
```

【例5.4】设计一个算法在1，2，…，9（顺序不能变）数字之间插入+或-或什么都不插入，使得计算结果总是100的程序，并输出所有的可能性。

例如：1+2+34–5+67–8+9=100。

解： 用数组*a*存放1～9的整数，用字符数组op存放插入的运算符，op[*i*]表示在*a*[*i*]之前插入的运算符。

采用回溯法产生和为100的表达式，op[*i*]只能取值+、-或者空格（不同于上一个示例，这里是三选一）。设计函数：

fun(op, sum, prevadd, *a*, *i*)

其中：sum记录考虑整数*a*[*i*]时前面表达式计算的整数和（初始值为*a*[0]），prevadd记录前面表达式中的一个数值（初始值为*a*[0]），*i*从1开始到9结束，如果sum=100，得到一个解。

```c
void fun(char op[],int sum,int prevadd,int a[],int i)
{  if (i==N)                        //扫描完所有位置
   {  if (sum==100)                 //找到一个解
      {  printf("  %d",a[0]);   //输出解
         for (int j=1;j<N;j++)
         {  if (op[j]!=' ')
                 printf("%c",op[j]);
            printf("%d",a[j]);
         }
         printf("=100\n");
      }
      return;
   }
```

```
    op[i]='+';                      //位置i插入'+'
    sum+=a[i];                      //计算结果
    fun(op,sum,a[i],a,i+1);         //继续处理下一个位置
    sum-=a[i];                      //回溯

    op[i]='-';                      //位置i插入'-'
    sum-=a[i];                      //计算结果
    fun(op,sum,-a[i],a,i+1);        //继续处理下一个位置
    sum+=a[i];                      //回溯

    op[i]=' ';                      //位置i插入' '
    sum-=prevadd;                   //先减去前面的元素值
    int tmp;                        //计算新元素值
    if (prevadd>0)
      tmp=prevadd*10+a[i];          //如prevadd=5,a[i]=6,结果为56
    else
      tmp=prevadd*10-a[i];          //如prevadd=-5,a[i]=6,结果为-56
    sum+=tmp;                       //计算合并结果
    fun(op,sum,tmp,a,i+1);          //继续处理下一个位置
    sum-=tmp;                       //回溯sum
    sum+=prevadd;
}
```

```c
void main()
{   int a[N];
    char op[N];                       //op[i]表示在位置i插入运算符
    for (int i=0;i<N;i++)             //为a赋值为1,2, ...,9
        a[i]=i+1;
    printf("求解结果\n");
    fun(op,a[0],a[0],a,1);            //插入位置i从1开始
}
```

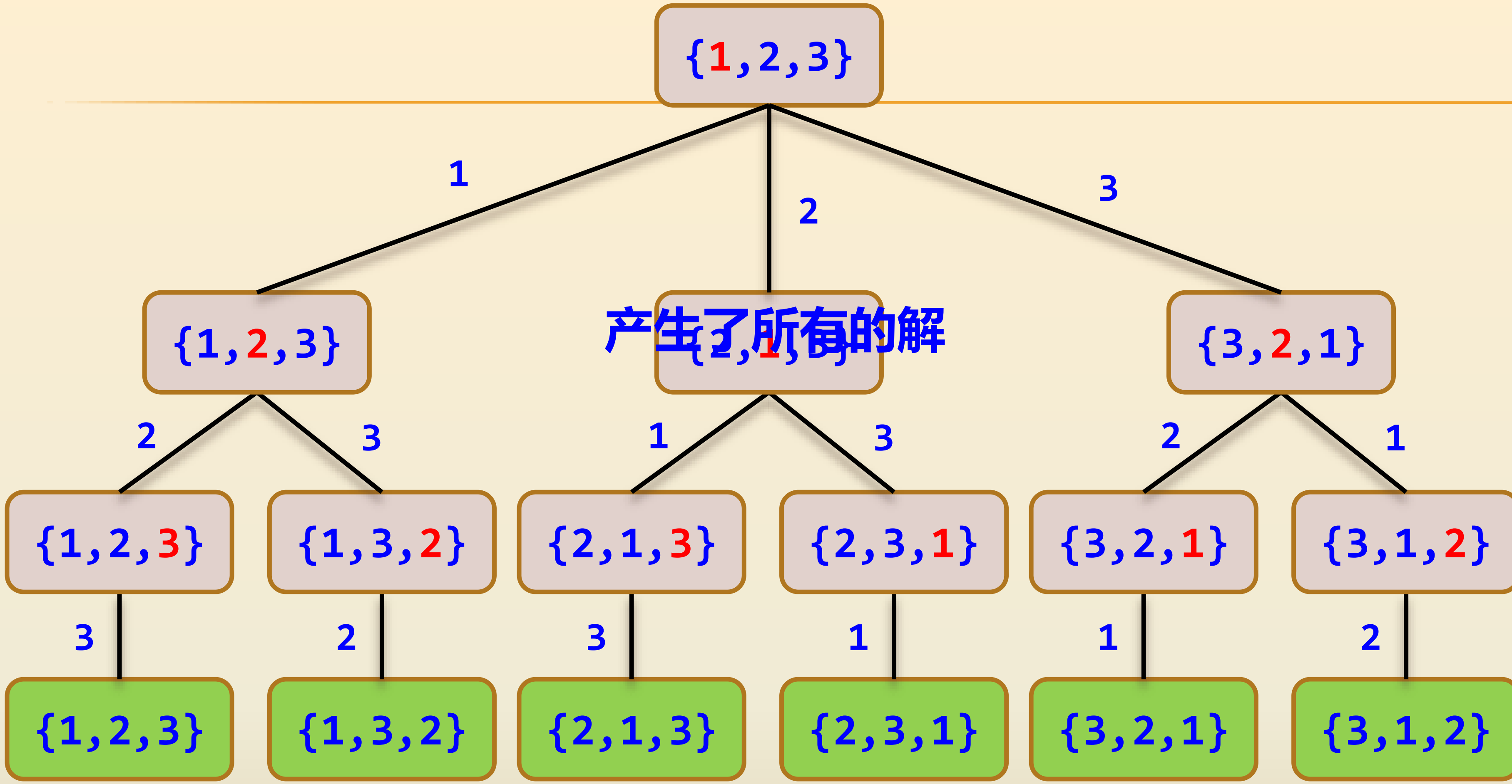
```
求解结果
     1+2+3-4+5+6+78+9=100
     1+2+34-5+67-8+9=100
     1+23-4+5+6+78-9=100
     1+23-4+56+7+8+9=100
     12+3+4+5-6-7+89=100
     12+3-4+5+67+8+9=100
     12-3-4+5-6+7+89=100
     123+4-5+67-89=100
     123+45-67+8-9=100
     123-4-5-6-7+8-9=100
     123-45-67+89=100
```

（2）解空间为排列树

```
int x[n];                       //x存放解向量，并初始化
void backtrack(int i)           //求解排列树的递归框架
{   if(i>n)                     //搜索到叶子结点,输出一个可行解
        输出结果;
    else
    {   for (j=i;j<=n;j++)      //用j枚举i所有可能的路径
        {   …                   //第i层的结点选择x[j]的操作
            swap(x[i],x[j]);    //为保证排列中每个元素不同,通过交换来实现
            if (constraint(i) && bound(i))
                backtrack(i+1); //满足约束条件和限界函数，进入下一层
            swap(x[i],x[j]);    //恢复状态
            …                   //第i层的结点选择x[j]的恢复操作
        }
    }
}
```

【例5.5】 有一个含n个整数的数组a，所有元素均不相同，求其所有元素的全排列。

例如，a[ ]={1，2，3}，得到结果是（1，2，3）、（1，3，2）、（2，3，1）、（2，1，3）、（3，1，2）、（3，2，1）。



{1,2,3}
1
2
3
{1,2,3}
产生了所有的解
{3,2,1}
2
3
1
3
2
1
{1,2,3}
{1,3,2}
{2,1,3}
{2,3,1}
{3,2,1}
{3,1,2}
3
2
3
1
1
2
{1,2,3}
{1,3,2}
{2,1,3}
{2,3,1}
{3,2,1}
{3,1,2}

```
void dfs(int a[],int n,int i)          //求a[0..n-1]的全排列
{  if (i>=n)                            //递归出口
      dispasolution(a,n);
   else
   {  for (int j=i;j<n;j++)
      {  swap(a[i],a[j]);               //交换a[i]与a[j]
         dfs(a,n,i+1);
         swap(a[i],a[j]);               //交换a[i]与a[j]：恢复
      }
   }
}
```

5.1.4 回溯法与深度优先遍历的异同

两者的相同点：

回溯法在实现上也是遵循深度优先的，即一步一步往前探索，而不像广度优先遍历那样，由近及远一片一片地搜索。

两者的不同点：

（1）访问序不同：深度优先遍历目的是“遍历”，本质是无序的。而回溯法目的是“求解过程”，本质是有序的。

（2）访问次数的不同：深度优先遍历对已经访问过的顶点不再访问，所有顶点仅访问一次。而回溯法中已经访问过的顶点可能再次访问。

（3）剪枝的不同：深度优先遍历不含剪枝，而很多回溯算法采用剪枝条件剪除不必要的分枝以提高效能。

5.1.5 回溯法算法的时间分析

通常以回溯算法的解空间树中的结点数作为算法的时间分析依据，假设解空间树共有n层。

第1层有m_0个满足约束条件的结点，每个结点有m_1个满足约束条件的结点；

第2层有m_0m_1个满足约束条件的结点，同理，第3层有$m_0m_1m_2$个满足约束条件的结点。

第n层有$m_0m_1 \ldots m_{n-1}$个满足约束条件的结点，则采用回溯法求所有解的算法的执行时间为

$T(n)=m_0+m_0m_1+m_0m_1m_2+\ldots+m_0m_1m_2 \ldots m_{n-1}$。

通常情况下，回溯法的效率会高于蛮力法。

5.2 求解0/1背包问题

【问题描述】有n个重量分别为$\{w_1, w_2, …, w_n\}$的物品，它们的价值分别为$\{v_1, v_2, …, v_n\}$，给定一个容量为W的背包。

设计从这些物品中选取一部分物品放入该背包的方案，每个物品要么选中要么不选中，要求选中的物品不仅能够放到背包中，而且**重量和恰好为W具有最大的价值。**

【问题求解】第4章采用蛮力法求解，这里采用回溯法求解该问题。

用$x[1..n]$数组存放最优解，其中每个元素取1或0，$x[i]=1$表示第i个物品放入背包中，$x[i]=0$表示第i个物品不放入背包中。

为了更清楚地描述算法，将这些给定的算法输入设计成全局变量。

这是一个求最优解问题。

对第i层上的某个分枝结点，对应的状态为dfs(i, tw, tv, op)，其中tw表示装入背包中的物品总重量，tv表示背包中物品总价值，op记录一个解向量。该状态的两种扩展如下：

（1）选择第i个物品放入背包：op[i]=1，tw=tw+w[i]，tv=tv+v[i]，转向下一个状态dfs(i+1, tw, tv, op)。该决策对应左分枝。

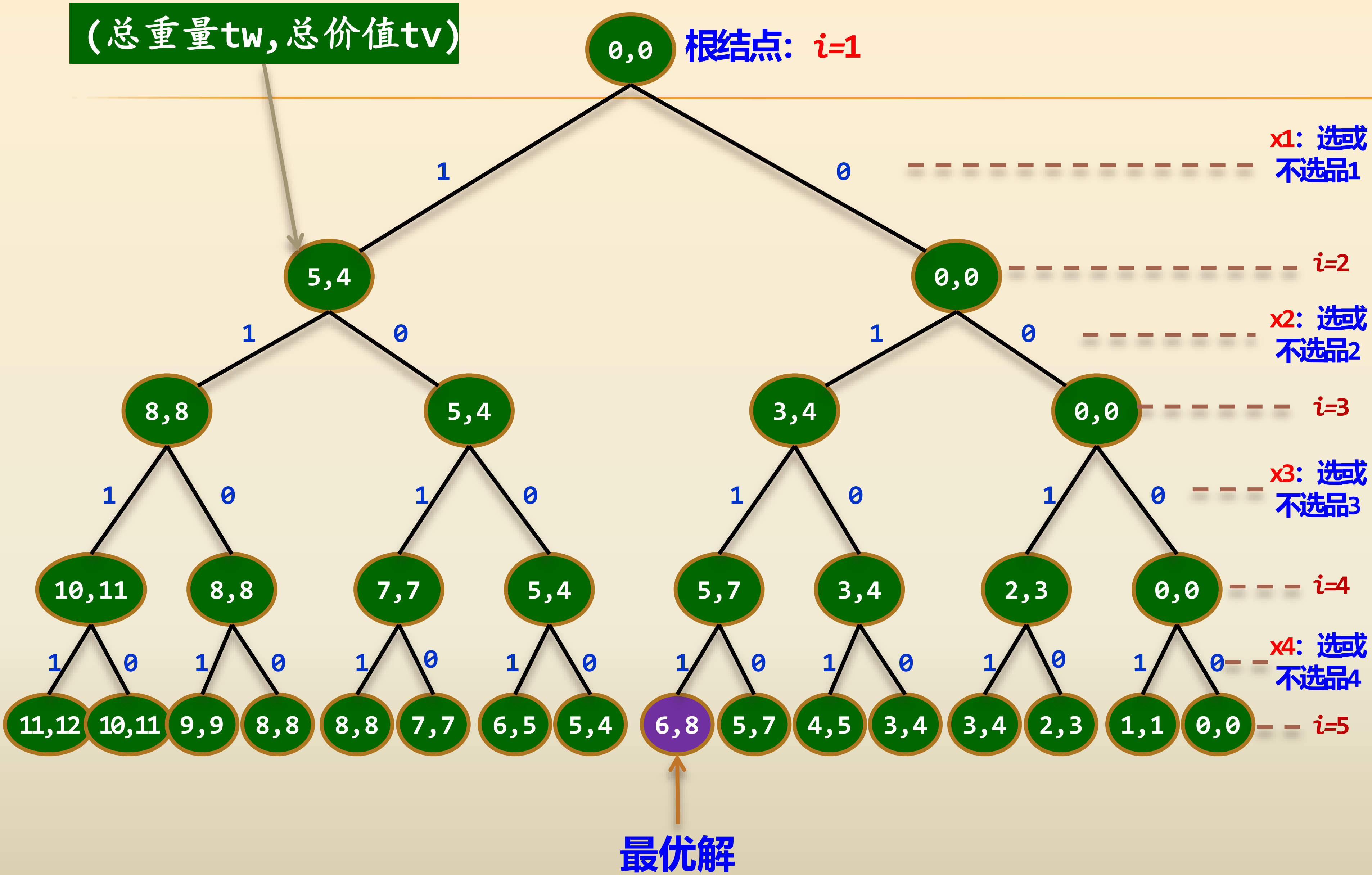
（2）不选择第i个物品放入背包：op[i]=0，tw不变，tv不变，转向下一个状态dfs(i+1, tw, tv, op)。该决策对应右分枝。

叶子结点表示已经对n个物品做了决策，对应一个解。对所有叶子结点进行比较求出满足tw≤W的最大tw，用maxv表示，对应的最优解op存放到x中。

0/1背包问题（W=6）：

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

解空间树



(总重量tw,总价值tv)
0,0
根结点: i=1
x1: 选或不选品1
1
0
5,4
0,0
i=2
x2: 选或不选品2
8,8
5,4
3,4
0,0
i=3
x3: 选或不选品3
10,11
8,8
7,7
5,4
5,7
3,4
2,3
0,0
i=4
x4: 选或不选品4
11,12
10,11
9,9
8,8
8,8
7,7
6,5
5,4
6,8
5,7
4,5
3,4
3,4
2,3
1,1
0,0
i=5
最优解

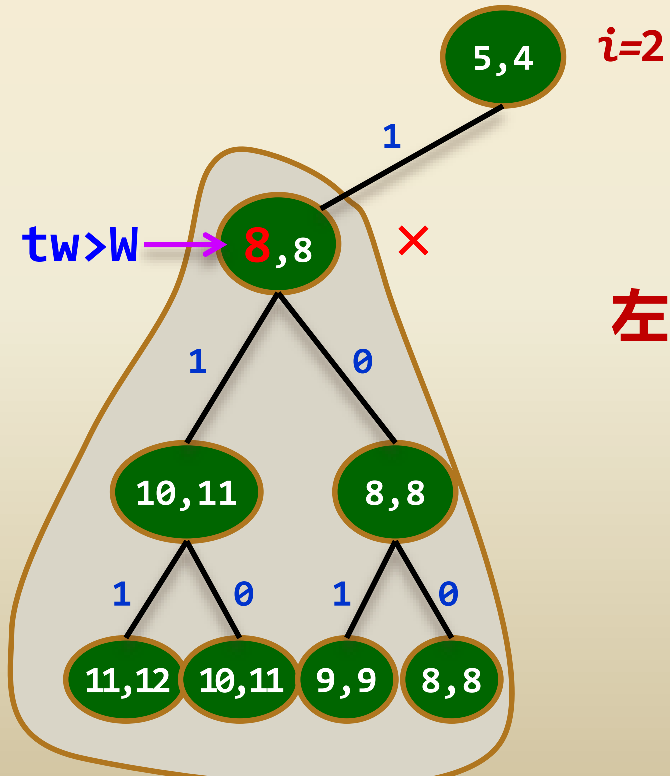
```
//问题表示
int n=4;                        //4种物品
int W=6;                        //限制重量为6
int w[]={0,5,3,2,1};            //存放4个物品重量,不用下标0元素
int v[]={0,4,4,3,1};            //存放4个物品价值,不用下标0元素
//求解结果表示
int x[MAXN];                    //存放最终解
int maxv;                       //存放最优解的总价值
```

采用解空间为子集树递归的算法框架

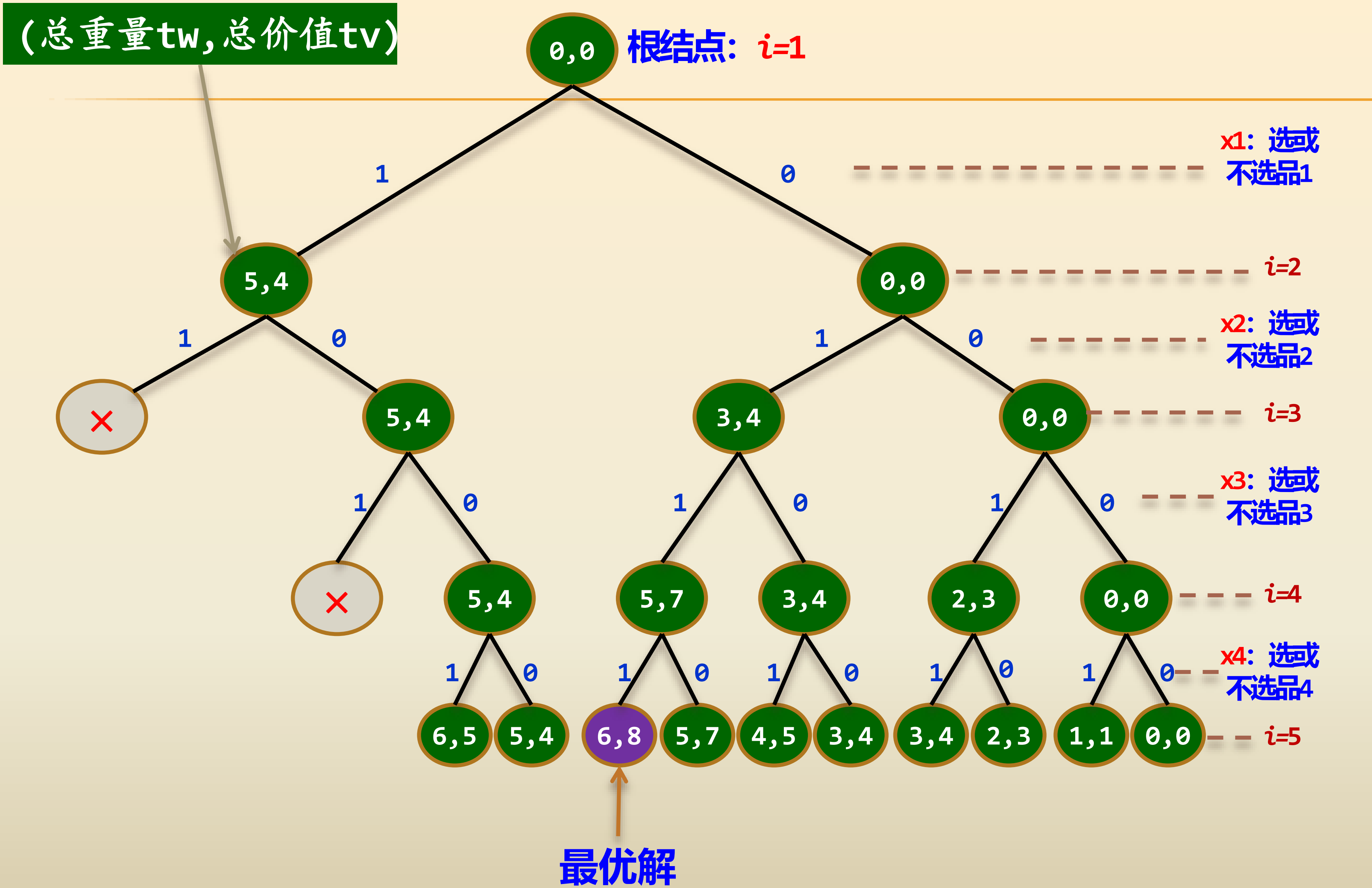
```
void dfs(int i,int tw,int tv,int op[]) //求解0/1背包问题
{  if (i>n)                            //找到一个叶子结点
   {  if (tw<=W && tv>maxv)            //找到一个满足条件的更优解,保存
      {  maxv=tv;
         for (int j=1;j<=n;j++)
            x[j]=op[j];
      }
   }
   else                                //尚未找完所有物品
   {  op[i]=1;                         //选取第i个物品
      dfs(i+1,tw+w[i],tv+v[i],op);
      op[i]=0;                         //不选取第i个物品,回溯
      dfs(i+1,tw,tv,op);
   }
}
```

改进：左剪枝

对于第i层的有些结点，tw+a[i]已超过了W，显然再选择a[i]是不合适的。如第2层的（5，4）结点，tw=5，a[2]=3，而tw+a[2]>W，选择物品2进行扩展是不必要的，可以增加一个限界条件进行剪枝，如若选择物品i会导致超重即tw+w[i]>W，就不再扩展该结点，也就是仅仅扩展tw+w[i]≤W的左孩子结点。



左剪枝条件：tw+w[i]<W



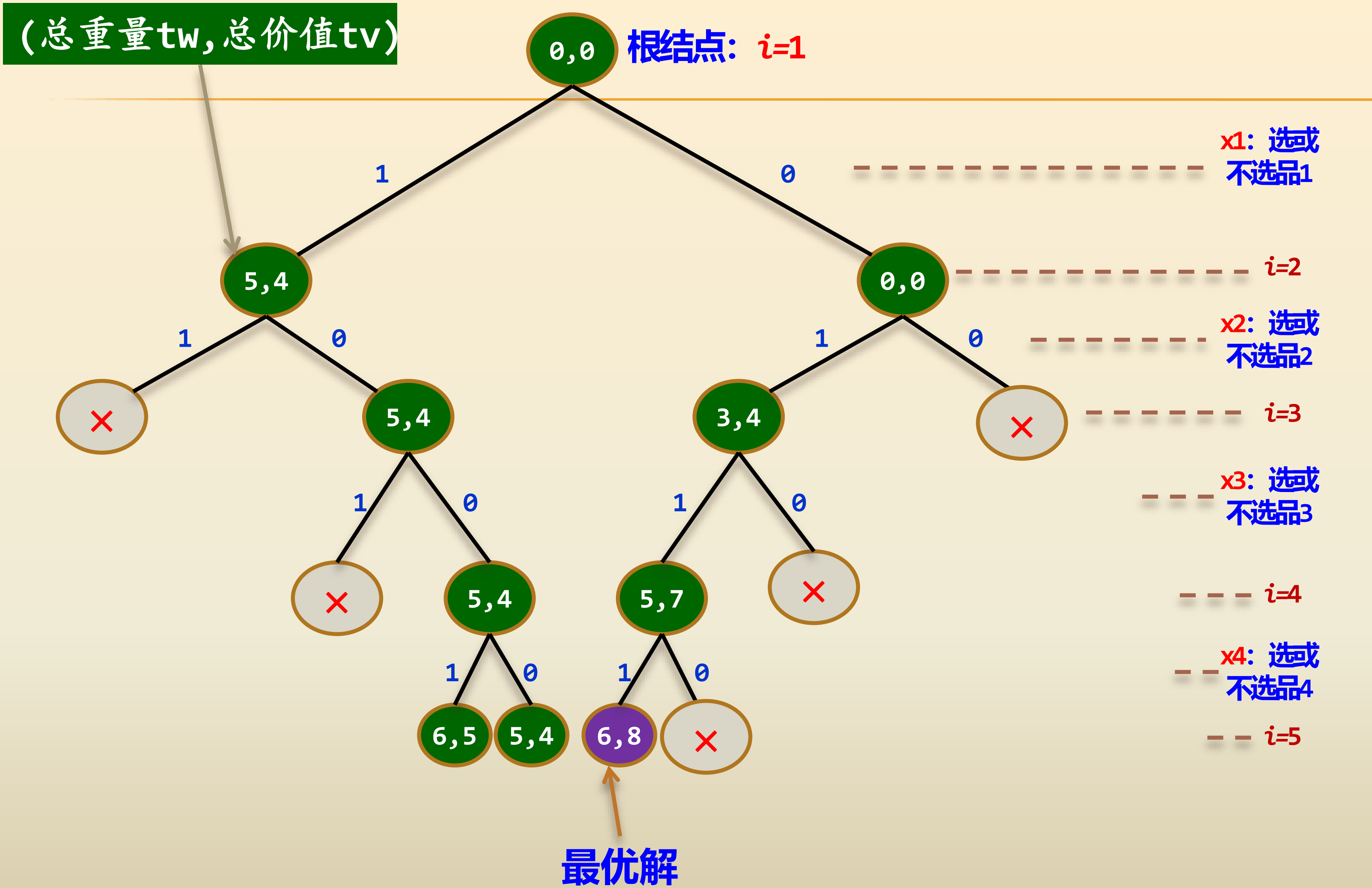
(总重量tw,总价值tv)
0,0
根结点：i=1
1
0
x1：选或不选品1
5,4
0,0
i=2
x2：选或不选品2
5,4
3,4
0,0
i=3
x3：选或不选品3
5,4
5,7
3,4
2,3
0,0
i=4
x4：选或不选品4
6,5
5,4
6,8
5,7
4,5
3,4
3,4
2,3
1,1
0,0
i=5
最优解

```
void dfs(int i,int tw,int tv,int op[])  //求解0/1背包问题
{  if (i>n)                             //找到一个叶子结点
   {  if (tw<=W && tv>maxv)             //找到一个满足条件的更优解,保存
      {  maxv=tv;
         maxw=tw;
         for (int j=1;j<=n;j++)
           x[j]=op[j];
      }
   }
   else                                 //尚未找完所有物品
   {  if (tw+w[i]<W)                    //左孩子结点剪枝
      {  op[i]=1;                       //选取第i个物品
         dfs(i+1,tw+w[i],tv+v[i],op);
      }
      op[i]=0;                          //不选取第i个物品,回溯
      dfs(i+1,tw,tv,op);
   }
}
```

改进：右剪枝

用rw表示考虑第i个物品时剩余物品的重量。

当不选择物品i时，若tw+rw≤W（注意rw中包含w[i]）时，也就是说即使选择后面的所有物品，重量也不会达到W，因此不必要再考虑扩展这样的结点，也就是说，对于右分枝仅仅扩展tw+rw>W的结点。



(总重量tw,总价值tv)
0,0
根结点：i=1
x1：选或
不选品1
1
0
5,4
0,0
i=2
x2：选或
不选品2
1
0
1
0
5,4
3,4
i=3
x3：选或
不选品3
1
0
1
0
5,4
5,7
i=4
x4：选或
不选品4
1
0
1
0
6,5
5,4
6,8
i=5
最优解

```
void dfs(int i,int tw,int tv,int rw,int op[]) //求解0/1背包问题
{  //初始调用时rw为所有物品重量和
   int j;
   if (i>n)                          //找到一个叶子结点
   {  if (tw==W && tv>maxv)          //找到一个满足条件的更优解,保存
      {  maxv=tv;
         for (j=1;j<=n;j++)          //复制最优解
            x[j]=op[j];
      }
   }
   else                              //尚未找完所有物品
   {  if (tw+w[i]<=W)                //左孩子结点剪枝
      {  op[i]=1;                    //选取第i个物品
         dfs(i+1,tw+w[i],tv+v[i],rw-w[i],op);
      }
      op[i]=0;                       //不选取第i个物品,回溯
      if (tw+rw>W)                   //右孩子结点剪枝
         dfs(i+1,tw,tv,rw-w[i],op);
   }
}
```

【算法分析】该算法不考虑剪枝时解空间树中有$2^{n+1}-1$个结点，对应的算法时间复杂度为$O(2^n)$。

5.3 求解装载问题

5.3.1 求解简单装载问题

【问题描述】有n个集装箱要装上一艘载重量为W的轮船，其中集装箱i（$1 \le i \le n$）的重量为w_i。不考虑集装箱的体积限制，现要这些集装箱中选出若干装上轮船，使它们的重量之和等于W，当总重量相同时要求选取的集装箱个数尽可能少。

例如，n=5，W=10，w={5，2，6，4，3}时，其最佳装载方案是（0，0，1，1，0），即装载第3、4个集装箱。

【问题求解】采用带剪枝的回溯法求解。问题的表示如下：

```
int w[]={0, 5, 2, 6, 4, 3}; //各集装箱重量，不用下标0的元素
int n=5, W=10;
```

求解的结果表示如下：

```
int maxw;                //存放最优解的总重量
int x[MAXN];             //存放最优解向量
int minnum=999999;       //存放最优解的集装箱个数,初值为最大值
```

将上述数据设计为全局变量。求解算法如下：

```
void dfs(int num, int tw, int rw, int op[], int i)
```

其中num表示选择的集装箱个数，tw表示选择的集装箱重量和，rw表示剩余集装箱的重量和，op表示一个解，即一个选择方案，i表示考虑的集装箱i。

```
void dfs(int num,int tw,int rw,int op[],int i) //考虑第i个集装箱
{  if (i>n)                          //找到一个叶子结点
   {  if (tw==W && num<minnum)
      {  maxw=tw;                    //找到一个满足条件的更优解,保存它
         minnum=num;
         for (int j=1;j<=n;j++)
            x[j]=op[j];               //复制最优解
      }
   }
   else                              //尚未找完所有集装箱
   {  op[i]=1;                       //选取第i个集装箱
      if (tw+w[i]<=W)                //左孩子结点剪枝：装载满足条件的集装箱
         dfs(num+1,tw+w[i],rw-w[i],op,i+1);
      op[i]=0;                       //不选取第i个集装箱,回溯
      if (tw+rw>W)                   //右孩子结点剪枝
        dfs(num,tw,rw-w[i],op,i+1);
   }
}
```

```
int w[]={0, 5, 2, 6, 4, 3};    //各集装箱重量，不用下标0的元素
int n=5, W=10;
```

```
最优方案
  选取第3个集装箱
  选取第4个集装箱
总重量=10
```

5.3.2 求解复杂装载问题

【问题描述】有一批共n个集装箱要装上两艘载重量分别为c1和c2的轮船，其中集装箱i的重量为w_i，且$w_1+w_2+\ldots+w_n \le c1+c2$。

装载问题要求确定是否有一个合理的装载方案可将这些集装箱装上这两艘轮船。如果有，找出一种装载方案。

例如，当n=3，c1=c2=50，w={10，40，40}时，则可以将集装箱1和2装到第一艘轮船上，而将集装箱3装到第二艘轮船上。如果w={20，40，40}，则无法将这3个集装箱都装上轮船。

【问题求解】如果一个给定的复杂装载问题有解，则可以采用如下方式得到一个装载方案：

首先将第一艘轮船尽可能装满，然后将剩余的集装箱装在第二艘轮船上。

可以用反证法证明其正确性。

```
void dfs(int tw,int rw,int op[],int i) //求第一艘轮船的最优解
{  if (i>n)                             //找到一个叶子结点
   {  if (tw<=c1 && tw>maxw)
      {  maxw=tw;                       //找到一个满足条件的更优解
         for (int j=1;j<=n;j++)         //复制最优解
           x[j]=op[j];
      }
   }
   else                        //尚未找完所有集装箱
   {
      op[i]=1;                 //选取第i个集装箱
      if (tw+w[i]<=c1)         //左孩子结点剪枝
         dfs(tw+w[i],rw-w[i],op,i+1);
      op[i]=0;                 //不选取第i个集装箱,回溯
      if (tw+rw>c1)            //右孩子结点剪枝
         dfs(tw,rw-w[i],op,i+1);
   }
}
```

```
bool solve()                    //求解复杂装载问题
{  int sum=0;                   //累计第一艘轮船装完后剩余的集装箱重量
   for (int j=1;j<=n;j++)
     if (x[j]==0)
        sum+=w[j];
   if (sum<=c2)                 //第二艘轮船可以装完
      return true;
   else                         //第二艘轮船不能装完
      return false;
}
```

```
void main()
{   int op[MAXN];                          //存放临时解
    memset(op,0,sizeof(op));
    int rw=0;
    for (int i=1;i<=n;i++)
       rw+=w[i];

    dfs(0,rw,op,1);                        //求第一艘轮船的最优解

    printf("求解结果\n");

    if (solve())                           //输出结果

    {  printf("    装载方案\n");
       dispasolution(n);
    }
    else printf("    没有合适的装载方案\n");
}
```

```
//问题表示
int w[]={0,10,40,40}; //各集装箱重量,不用下标0的元素
int n=3;
int c1=50,c2=50;
```

求解结果

装载方案

将第1个集装箱装上第一艘轮船
将第2个集装箱装上第一艘轮船
将第3个集装箱装上第二艘轮船

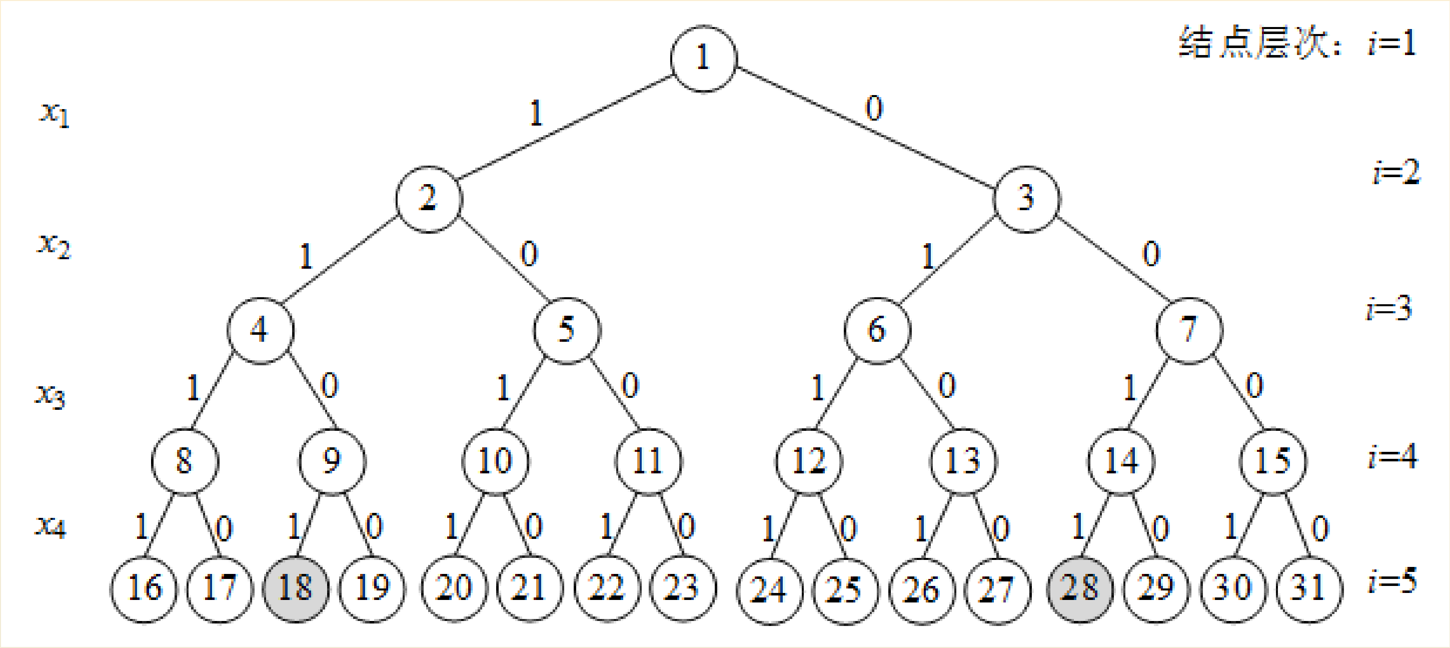
5.4 求解子集和问题

5.4.1 求子集和问题的解

【问题描述】给定n个不同的正整数集合$w=(w_1, w_2, …, w_n)$和一个正数W，要求找出w的子集s，使该子集中所有元素的和为W。

例如，当$n=4$时，$w=(11, 13, 24, 7)$，$W=31$，则满足要求的子集为（11，13，7）和（24，7）。

【问题求解】 当n=4时的解空间树如下：



从i层到i+1层（1≤i≤n）的每一条边标有x_i的值，x_i或者为1或者为0，x_i为1时表示取w_i整数，x_i为0时表示不取w_i整数，从根结点到叶子结点的所有路径定义了解空间。

用tw表示选取的整数和，rw表示余下的整数和，设置相关的剪枝函数如下：

- **约束函数**：通过检查当前整数$w[i]$加入后子集和是否超过W，若是则不能选择该路径。这用于左孩子结点的剪枝。
- **限界函数**：如果一个结点满足$tw+rw<W$，也就是说即便选择剩余所有整数，也不可能找到一个解。这用于右孩子结点的剪枝。

```
void dfs(int tw,int rw,int x[],int i) //求解子集和
{  //tw为考虑第i个整数时选取的整数和, rw为剩下的整数和
   if (i>n)                             //找到一个叶子结点
   {  if (tw==W)                        //输出一个满足条件的解
        dispasolution(x);
   }

   else                                 //尚未找完所有整数
   {  if (tw+w[i]<=W) //左孩子结点剪枝：选取满足条件的整数w[i]
      {  x[i]=1;                        //选取第i个整数
         dfs(tw+w[i],rw-w[i],x,i+1);
      }

      if (tw+rw>W)    //右孩子结点剪枝：剪除不可能存在解的结点
      {  x[i]=0;                        //不选取第i个整数,回溯
           dfs(tw,rw-w[i],x,i+1);
      }
   }
}
```

```
int n=4,W=31;
int w[]={0,11,13,24,7};        //存放所有整数,不用下标0的元素
```

```
第1个解:
  选取的数为11 13 7
第2个解:
  选取的数为24 7
```

【算法分析】算法的解空间树中有$2^{n+1}-1$个结点，对应的算法时间复杂度为$O(2^n)$。

5.4.2 判断子集和问题是否存在解

采用回溯法一般是针对问题存在解时求出相应的一个或多个解，或者最优解。

如果要判断问题是否存在解（一个或者多个），可以将求解函数改为bool类型，当找到任何一个解时返回true，否则返回false。需要注意的是当问题没有解时需要搜索所有解空间。

```
bool dfs(int tw,int rw,int i)        //求解子集和
{  if (i>n)                          //找到一个叶子结点
   {  if (tw==W)                     //找到一个满足条件的解,输出
        return true;
   }

   else                              //尚未找完所有物品

   {  if (tw+w[i]<=W)                //左孩子结点剪枝
        return dfs(tw+w[i]-w[i],rw,i+1);    //选取第i个整数

      if (tw+rw>W)                   //右孩子结点剪枝

        return dfs(tw,rw-w[i],i+1);  //不选取第i个整数,回溯
   }
   return false;
}
```

另外一种方法是通过解个数来判断，如设置全局变量count表示解个数，初始化为0，调用搜索解的回溯算法，当找到一个解时置count++。

最后判断count>0算法成立，若为真，表示存在解，否则表示不存在解。

5.5 求解*n*皇后问题

第2讲采用递归技术求解，这里采用回溯法求解。实际上，2.3.2小节的递归算法对应的就是回溯法的递归框架，这里讨论采用非递归框架求解皇后问题。

非递归回溯算法对应的算法:

```
void Queens(int n)      //求解n皇后问题
{  int i=1;             //i表示当前行,也表示放置第i个皇后
   q[i]=0;              //q[i]是当前列,每个新考虑的皇后初始位置置为0列
   while (i>=1)         //尚未回溯到头, 循环
   {  q[i]++;           //原位置后移动一列
      while (q[i]<=n && !place(i))  //试探一个位置(i,q[i])
        q[i]++;

      if (q[i]<=n)      //为第i个皇后找到了一个合适位置(i,q[i])
      {  if (i==n)      //若放置了所有皇后,输出一个解
           dispasolution(n);
         else           //皇后没有放置完
         {  i++;        //转向下一行,即开始下一个新皇后的放置
            q[i]=0;     //每个新考虑的皇后初始位置置为0列
         }
      }
      else i--;         //若第i个皇后找不到合适的位置,则回溯到上一个皇后
   }
}
```

```
bool place(int i)      //测试第i行的q[i]列上能否摆放皇后
{  int j=1;
   if (i==1) return true;
   while (j<i)                    //j=1～i-1是已放置了皇后的行
   {  if ((q[j]==q[i]) || (abs(q[j]-q[i])==abs(j-i)))
        //该皇后是否与以前皇后同列，位置(j,q[j])与(i,q[i])是否同对角线
         return false;
      j++;
   }
   return true;
}
```

【算法分析】 该算法中每个皇后都要试探n列，共n个皇后，其解空间是一棵子集树，不同于前面一般的二叉树子集树，这里每个结点可能有n棵子树。

对应的算法时间复杂度为$O(n^n)$。

5.6 求解图的*m*着色问题

【问题描述】给定无向连通图*G*和*m*种不同的颜色。用这些颜色为图*G*的各顶点着色，每个顶点着一种颜色。如果有一种着色法使*G*中每条边的两个顶点着不同颜色，则称这个图是*m*可着色的。图的*m*着色问题是对于给定图*G*和*m*种颜色，找出所有不同的着色法。

【输入格式】第1行有3个正整数*n*、*k*和*m*，表示给定的图*G*有*n*个顶点和*k*条边，*m*种颜色。顶点编号为1，2，…，*n*。接下来的*k*行中，每行有两个正整数*u*、*v*，表示图*G*的一条边（*u*，*v*）。

【输出格式】程序运行结束时，将计算出的不同的着色方案数输出。如果不能着色，程序输出-1。

【输入样例】

5 8 4

1 2

1 3

1 4

2 3

2 4

2 5

3 4

4 5

【输出样例】

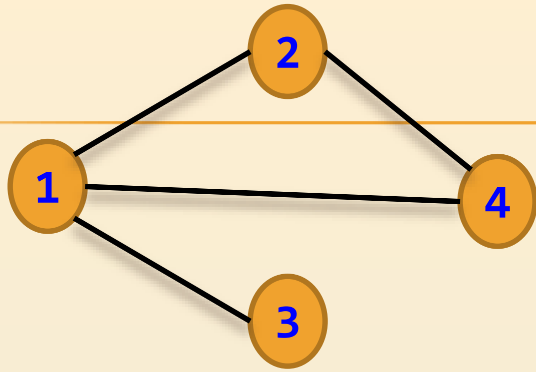
48

【问题求解】对于图G，采用邻接矩阵a存储，根据求解问题需要，这里a为一个二维数组（下标0不用），当顶点i与顶点j有边时，置$a[i][j]=1$，其他情况置$a[i][j]=0$。

图中的顶点编号为1～n，着色编号为1～m。对于图G中的每一个顶点，可能的着色为1～m，所以对应的解空间是一棵m叉树，高度为n，层次i从1开始。

```
bool Same(int i)      //判断顶点i是否与相邻顶点存在相同的着色
{  for (int j=1;j<=n;j++)
     if (a[i][j]==1 && x[i]==x[j])
        return false;
   return true;
}

void dfs(int i)                //求解图的m着色问题
{  if (i>n)                    //达到叶子结点
      count++;                 //着色方案数增1
   else
   {  for (int j=1;j<=m;j++)  //试探每一种着色
      {  x[i]=j;               //试探着色j
         if (Same(i))          //可以着色j，进入下一个顶点着色
            dfs(i+1);
         x[i]=0;               //回溯
      }
   }
}
```



n=4，*k*=4，*m*=3，其着色方案有12个。

```
第1个着色方案:  1 2 2 3
第2个着色方案:  1 2 3 2
第3个着色方案:  1 3 2 3
第4个着色方案:  1 3 3 2
第5个着色方案:  2 1 1 3
第6个着色方案:  2 1 3 1
第7个着色方案:  2 3 1 3
第8个着色方案:  2 3 3 1
第9个着色方案:  3 1 1 2
第10个着色方案:  3 1 2 1
第11个着色方案:  3 2 1 2
第12个着色方案:  3 2 2 1
```

【算法分析】该算法中每个顶点试探1～m种着色，共n个顶点，对应解空间树是一棵m叉树（子集树），算法的时间复杂度为$O(m^n)$。

5.7 求解任务分配问题

【问题描述】有n（$n≥1$）个任务需要分配给n个人执行，每个任务只能分配给一个人，每个人只能执行一个任务。

第i个人执行第j个任务的成本是$c[i][j]$（$1≤i$，$j≤n$）。求出总成本最小的分配方案。

【问题求解】这里采用回溯法求解。问题表示如下：

```
int n=4;
int c[MAXN][MAXN]={{0},{0,9,2,7,8},{0,6,4,3,7},
                   {0,5,8,1,8},{0,7,6,9,4} };
    //下标0的元素不用，c[i][j]表示第i个人执行第j个任务的成本
```

4个人员、4个任务的信息

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

考虑为第i个人员分配任务（i从1开始），由于每个任务只能分配给一个人员，为了避免重复分配，设计一个worker布尔数组，初始时均为false，当任务j分配后置worker[j]=true。求解结果表示如下：

```
int x[MAXN];            //临时解
int cost=0;             //临时解的成本
int bestx[MAXN];        //最优解
int mincost=INF;        //最优解的成本
bool worker[MAXN];      //worker[j]表示任务j是否已经分配人员
```

```
void dfs(int i)                          //为第i个人员分配任务
{  if (i>n)                              //到达叶子结点
   {  if (cost<mincost)                  //比较求最优解
      {  mincost=cost;
         for (int j=1;j<=n;j++)
           bestx[j]=x[j];
      }
   }

   else
   {  for (int j=1;j<=n;j++)         //为人员i试探任务j:1到n
        if (!worker[j])                  //若任务j还没有分配
        {  worker[j]=true;
           x[i]=j;                       //任务j分配给人员i
           cost+=c[i][j];
           dfs(i+1);                     //为人员i+1分配任务
           worker[j]=false;              //回退
           x[j]=0;
           cost-=c[i][j];
        }
   }
}
```

程序的执行结果：

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

最优方案：

第1个人安排任务2
第2个人安排任务1
第3个人安排任务3
第4个人安排任务4

总成本=13

【算法分析】该算法中每个人员试探1～n个任务，对应解空间树是一棵n叉树（子集树），算法的时间复杂度为$O(n^n)$。

5.8 求解活动安排问题

【问题描述】 假设有一个需要使用某一资源的n个活动所组成的集合S，$S=\{1，…，n\}$。该资源任何时刻只能被一个活动所占用，活动i有一个开始时间b_i和结束时间e_i（$b_i<e_i$），其执行时间为e_i-b_i，假设最早活动执行时间为0。

一旦某个活动开始执行，中间不能被打断，直到其执行完毕。若活动i和活动j有$b_i \ge e_j$或$b_j \ge e_i$，则称这两个活动兼容。

设计算法求一种最优活动安排方案，使得所有安排的活动个数最多。

【问题求解】这里采用回溯法求解，相当于找到$S=\{1, …, n\}$的某个排列即调度方案，使得其中所有兼容活动的执行时间和最大，显然对应的解空间是一个是排列树。

直接采用**排列树递归框架**实现，对于每一种调度方案求出所有兼容活动个数，通过比较求出最多活动个数，对应的调度方案就是最优调度方案，即为本问题的解。

对于一种调度方案，如何计算所有兼容活动的个数呢？因为其中可能存在不兼容的活动。

例如，有如表5.1所示的4个活动，若调度方案为（1，2，3，4），求所有兼容活动个数的过程如下：

活动编号	1	2	3	4
开始时间	1	2	4	6
结束时间	3	5	8	10

① 置当前活动最大结束时间laste=0，所有兼容活动个数sum=0。

② 活动1：其开始时间为1，大于等于laste，属于兼容活动，选取它，sum增加1，sum=1，置laste=其结束时间=3。

③ 活动2：其开始时间为2，小于laste，属于非兼容活动，不选取它。

④ 活动3：其开始时间为4，大于等于laste，属于兼容活动，选取它，sum增加1，sum=2，置laste=其结束时间=8。

⑤ 活动4：其开始时间为6，小于laste，属于非兼容活动，不选取它。

⑥ 该调度方案的所有兼容活动个数sum为2。

问题表示

```
struct Action
{   int b;                                  //活动起始时间
    int e;                                  //活动结束时间
};
int n=4;
Action A[]={{0,0},{1,3},{2,5},{4,8},{6,10}}; //下标0不用
```

问题的求解结果表示:

```
int x[MAX];                //临时解向量
int bestx[MAX];            //最优解向量
int laste=0;               //一个调度方案中最后兼容活动的结束时间,初值为0
int sum=0;                 //一个调度方案中所有兼容活动个数,初值为0
int maxsum=0;
```

```
void dfs(int i)          //搜索活动问题最优解
{  if (i>n)              //到达叶子结点,产生一种调度方案
   {  if (sum>maxsum)
      {  maxsum=sum;
         for (int k=1;k<=n;k++)
            bestx[k]=x[k];
      }
   }
```

```
   else
   {  for(int j=i; j<=n; j++)          //没有到达叶子结点,考虑i到n的活动
      {    //第i层结点选择活动x[j]
         int sum1=sum;                 //保存sum, laste以便回溯
         int laste1=laste;
         if (A[x[j]].b>=laste)         //活动x[j]与前面兼容
         {  sum++;                     //兼容活动个数增1
            laste=A[x[j]].e;           //修改本方案的最后兼容时间
         }
         swap(x[i],x[j]);    //排序树问题递归框架:交换x[i],x[j]
         dfs(i+1);           //排序树问题递归框架:进入下一层
         swap(x[i],x[j]);    //排序树问题递归框架:交换x[i],x[j]
         sum=sum1;           //回溯
         laste=laste1;       //即撤销第i层结点对活动x[j]的选择
      }
   }
}
```

活动编号	1	2	3	4
开始时间	1	2	4	6
结束时间	3	5	8	10

最优调度方案

选取活动1: [1,3)

选取活动3: [4,8)

安排活动的个数=2

【算法分析】该算法对应解空间树是一棵排列树，与求全排列算法的时间复杂度相同，即为O(n!)。

5.9 求解流水作业调度问题

【问题描述】有n个作业（编号为1～n）要在由两台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业i所需的时间分别为a_i和b_i（$1≤i≤n$）。

流水作业调度问题要求确定这n个作业的最优加工顺序，使得从第一个作业在机器M1上开始加工，到最后一个作业在机器M2上加工完成所需的时间最少。可以假定任何作业一旦开始加工，就不允许被中断，直到该作业被完成，即非优先调度。

【输入格式】输入包含若干个用例。每个用例第一行是作业数n（$1 \le n \le 1000$），接下来n行，每行两个非负整数，第i行的两个整数分别表示在第i个作业在第一台机器和第二台机器上加工时间。以输入n=0结束。

【输出格式】每个用例输出一行，表示采用最优调度所用的总时间，即从第一台机器开始到第二台机器结束的时间。

【输入样例】

4
5 6
12 2
4 14
8 7
0

作业编号	1	2	3	4
M1时间a	5	12	4	8
M2时间b	6	2	14	7

【输出样例】

33

【问题求解】采用回溯法求解，对应的解空间是一个是**排列树**，相当于求出n个作业的一种排列使完成时间最少。

作业的编号是1～n，用数组x[]作为解向量即调度方案，即x[i]表示第i顺序执行的作业编号，初始时数组x的元素分别是1～n，最优解向量用bestx[]存储，对应的最优调度时间用bestf表示。

求作业的所有排列可以直接采用**排列树递归框架**实现。对于每一种调度方案求出其所有作业执行的总时间，通过比较求出最小的总时间，对应的调度方案就是最优调度方案，即为本问题的解。

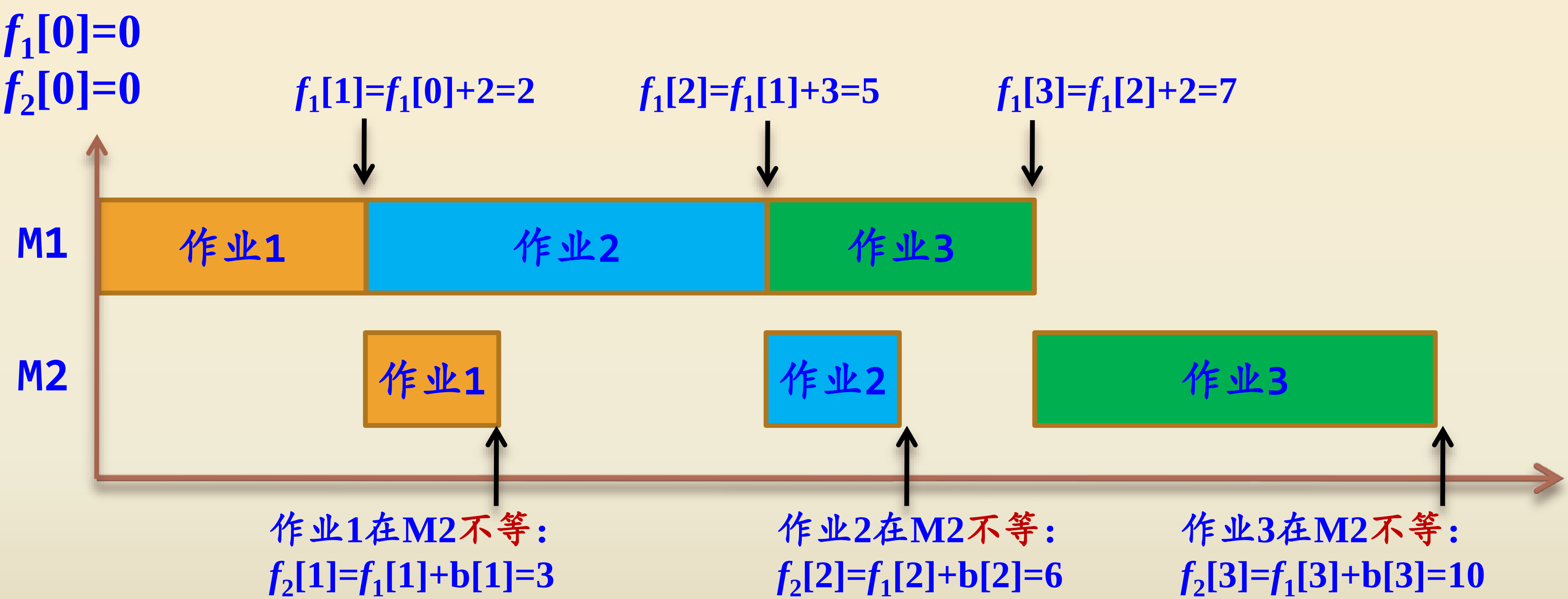
用f_1数组表示在M1上执行完当前作业i的总时间（含前面作业的执行时间），f_2数组表示在M2上执行完当前作业i的总时间（含前面作业的执行时间）。

由于一个作业总是先在M1上执行后在M2执行，所以$f_2[n]$就是执行全部作业的总时间。

一个示例，假设有3个作业：

作业编号	1	2	3
M1时间a	2	3	2
M2时间b	1	1	3

现在的调用方案为（1，2，3），即按作业1、2、3的顺序执行。首先将$f1$和$f2$数组所有元素初始化为0。该调度方案的总时间计算：

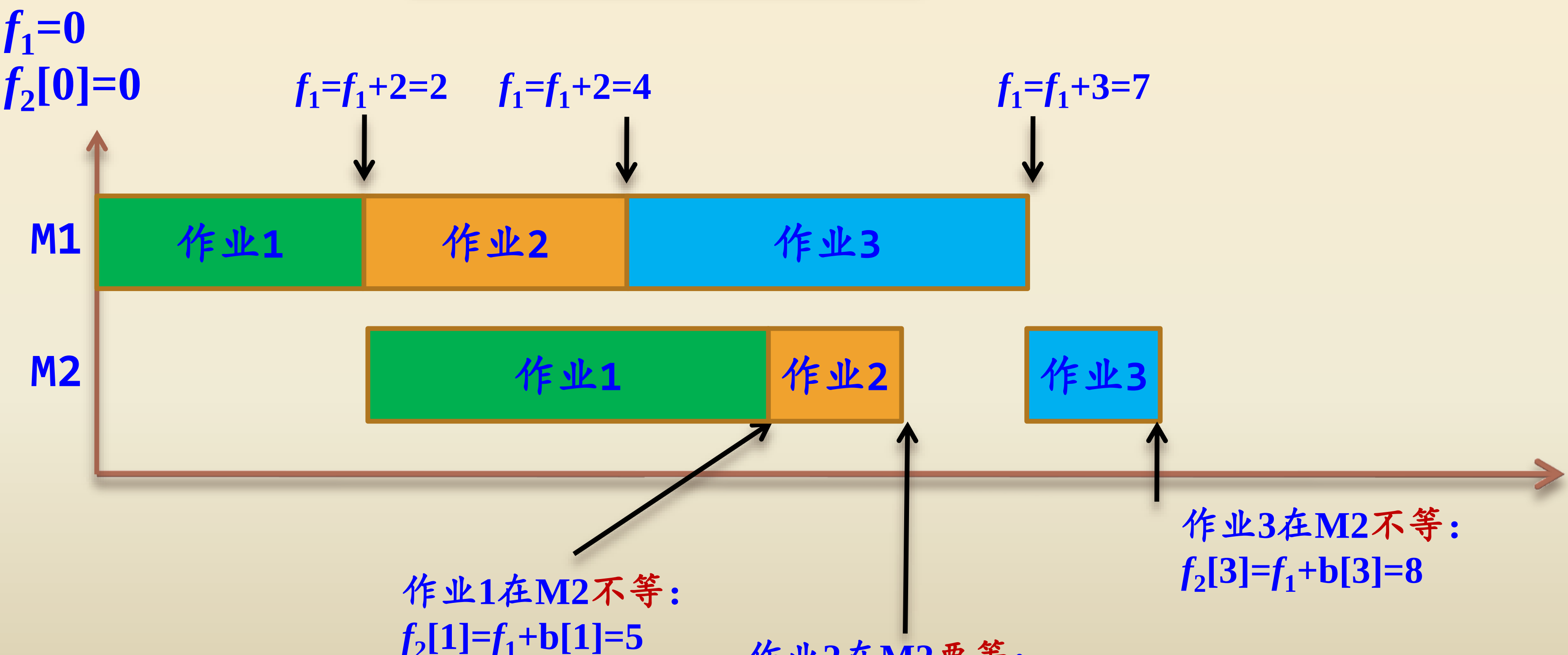


由于每个作业都是从M1开始的，即M1上各个作业是连续执行的，不需要等待，所以f_1不需要用数组表示，直接用单个变量f_1表示，

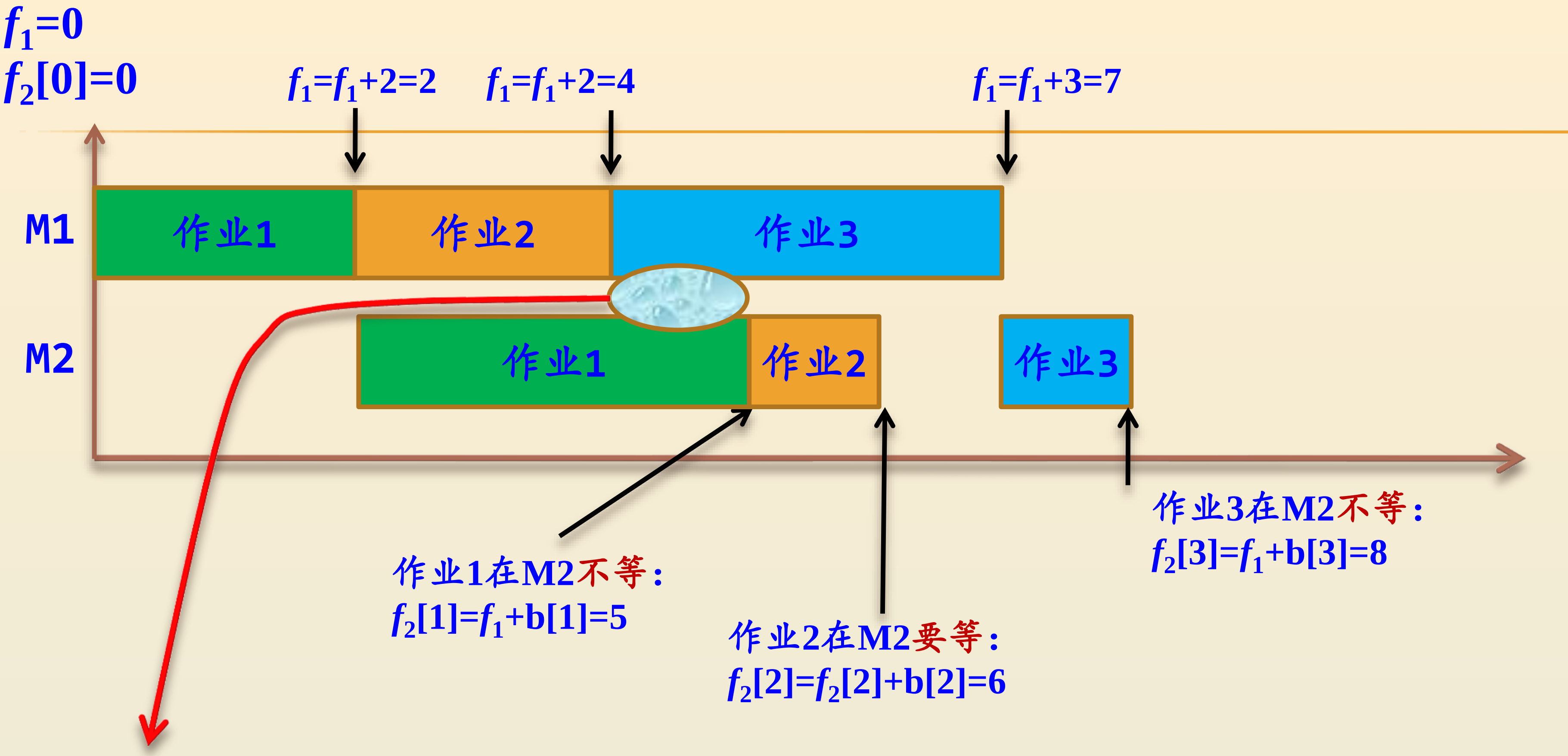
该调度方案的总时间：10

再看看另外一种调用方案，假设3个作业如表5.3所示，调用方案仍然是（1，2，3）。该调度方案的总时间计算：

作业编号	1	2	3
M1时间a	2	2	3
M2时间b	3	1	1



该调度方案的总时间：8



当前作业i在M2上需要等待的条件：**f_2[i-1]>f_1**；否则不需要等待

```
f1 += a[x[j]];          //在第i层选择执行作业x[j],在M1上执行完的时间
f2[i]=max(f1,f2[i-1])+b[x[j]];
```

```
void dfs(int i)                 //从第i层开始搜索
{  if (i>n)                     //到达叶子结点,产生一种调度方案
   {  if (f2[n]<bestf)                    //找到更优解
      {  bestf=f2[n];
         printf("   一个解: bestf=%d",bestf);
         printf(", 调度方案: "); disparr(x);
         printf(", f2: "); disparr(f2);
         printf("\n");
         for(int j=1; j<=n; j++)            //复制解向量
            bestx[j] = x[j];
      }
   }
```

```
    else
    {  for(int j=i; j<=n; j++)  //没有到达叶子结点,考虑i到n的作业
       {  f1 += a[x[j]];
             //在第i层选择执行作业x[j],在M1上执行完的时间
          f2[i]=max(f1,f2[i-1])+b[x[j]];
          if (f2[i]<bestf)
             //剪枝:仅仅扩展当前总时间小于bestf的结点
          {  swap(x[i],x[j]);
             dfs(i+1);
             swap(x[i],x[j]);
          }
          f1 -= a[x[j]];
             //回溯, 即撤销第i层对作业x[j]的选择,以便再选择其他作业
      }
  }
}
```

```
int n=4;                       //作业数
int a[MAX]={0,5,12,4,8};       //M1上的执行时间,不用下标0的元素
int b[MAX]={0,6,2,14,7};       //M2上的执行时间,不用下标0的元素
```

作业编号	1	2	3	4
M1时间a	5	12	4	8
M2时间b	6	2	14	7

求解过程：

一个解：bestf=42，调度方案：1 2 3 4，f2：11 19 35 42

一个解：bestf=36，调度方案：1 3 2 4，f2：11 25 27 36

一个解：bestf=34，调度方案：1 3 4 2，f2：11 25 32 34

一个解：bestf=33，调度方案：3 1 4 2，f2：18 24 31 33

求解结果：

最少时间：33，最优调度方案：3 1 4 2

【算法分析】该算法的解空间树是一棵高度为n的**排列树**，对应算法的时间复杂度为$O(n!)$。