回顾

第1章　算法概述

第2章　递归与分治策略

第3章　动态规划

第4章　贪心算法

第5章　回溯法

第6章　分支限界法

第7章　随机化算法

第8章　线性规划与网络流

第9章　串与序列算法

二、算法总览

经典问题、经典算法

2.1 经典问题和算法

排序问题： 快速排序、合并排序、排列问题等

搜索问题： 数组结构（二分搜索）等

图论问题： 单源最短路径距离、着色问题、旅行售货员问题等

组合数学问题： 背包问题、装载问题、活动安排问题、n后问题等

经典问题

分治法： 合并排序、快速排序、二分搜索技术、Strassen矩阵乘法、棋盘覆盖

动态规划算法： 0/1背包问题、最长公共子序列、最大字段和、矩阵连乘问题

贪心算法： 单源最短路径、活动安排问题、最优装载、哈夫曼编码

回溯法： 0-1背包问题、装载问题、n后问题、旅行售货员问题

分支限界法： 0-1背包问题、单源最短路径、装载问题、旅行售货员问题

经典算法

2.1.1经典算法：分治法

主要思想： 将问题规模为n的问题分解为k个较小的子问题，这些子问题互相独立且与原问题相同，递归地解这些子问题，然后将各个子问题的解合并得到原问题的解。

解决的问题满足以下特征：

① 问题的规模缩小到一定的程度就可以容易地解决；

② 问题可以分解为若干个规模较小的相同问题;

③ 子问题的解可以合并为该问题的解；

④ 各个子问题是**相互独立**的;

代码:

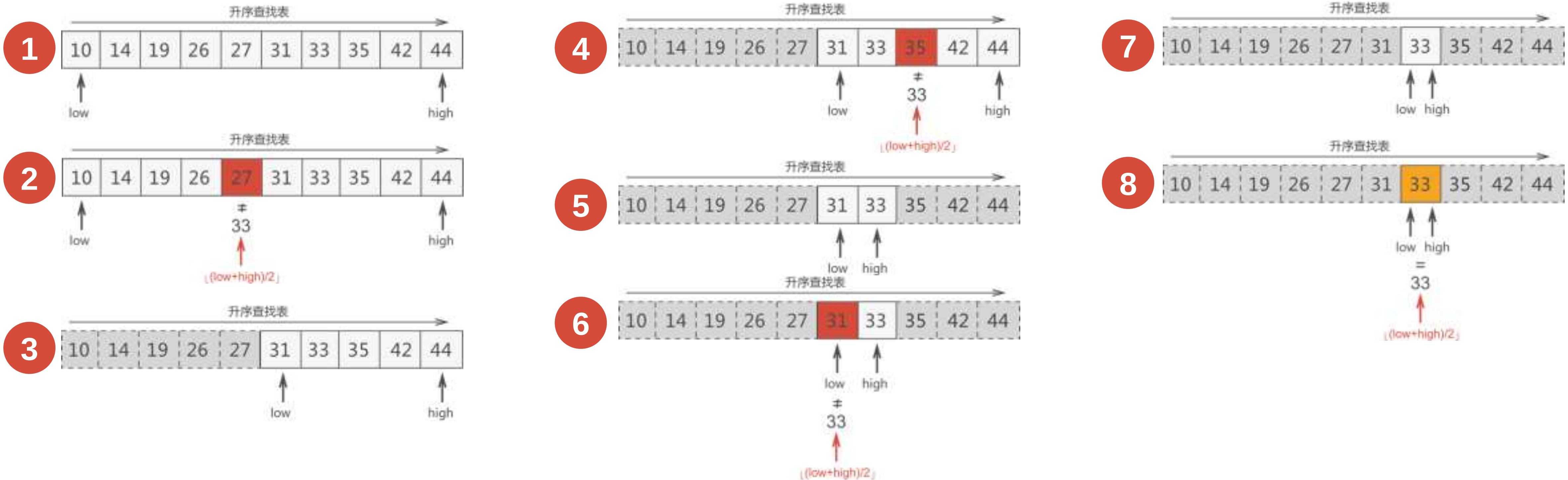
```
divide-and-conquer(P)
 {
  if ( | P | <= n0)  adhoc(P);    //解决小规模的问题
  //分解问题
  divide P into smaller sub instances P1,P2,...,Pk;
  for (i=1,i<=k,i++)
    yi=divide-and-conquer(Pi);  //递归的解各子问题
  //将各子问题的解合并为原问题的解
  return merge(y1,...,yk);
 }
```

2.1.1经典算法：分治法

（1）二分搜索

问题：给定已按升序排好序的n个元素$a[0:n-1]$，要在这n个元素中找出特定元素x。

思路：① 该问题的规模缩小到一定的程度就可以容易地解决；② 该问题可以分解为若干个规模较小的相同问题；③ 分解出的子问题的解可以合并为原问题的解；④ 分解出的各个子问题是相互独立的，即在$a[i]$的前面或后面查找x是独立的子问题。



2.1.1经典算法：分治法

$$T(n)=\begin{cases} O(1) & n\le 1 \\ T(n/2)+O(1) & n>1 \end{cases}$$

（1）二分搜索

时间复杂度：O(logn)

代码（递归）：

```
int BinarySearch_Rec(int [] a, int x, int left, int right)
{
   while (left <= right){
      int middle = left + (right - left) / 2;
      if (x==a[middle])
          return middle;
      if (x > a[middle])
           return BinarySearch_Rec(a, x, mid + 1, right);
       else
          return BinarySearch_Rec(a, x, left, middle-1);
      }
   return -1; // 未找到x

}
```

代码（非递归）：

```
int binarySearch(int [] a, int x, int n)
  {
    // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
    // 找到x时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
  }
```

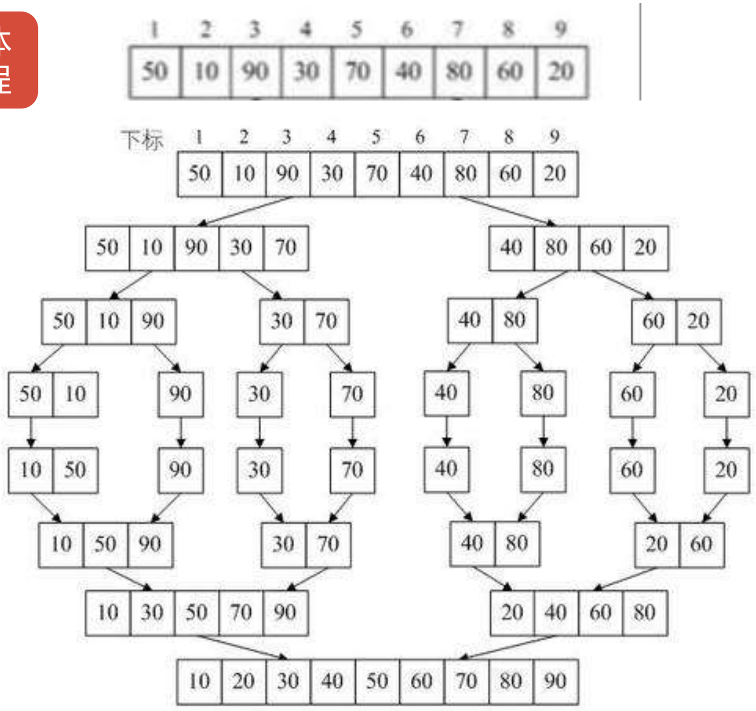
2.1.1经典算法：分治法

（2）合并排序

问题： 用分治策略对n个元素进行排序。

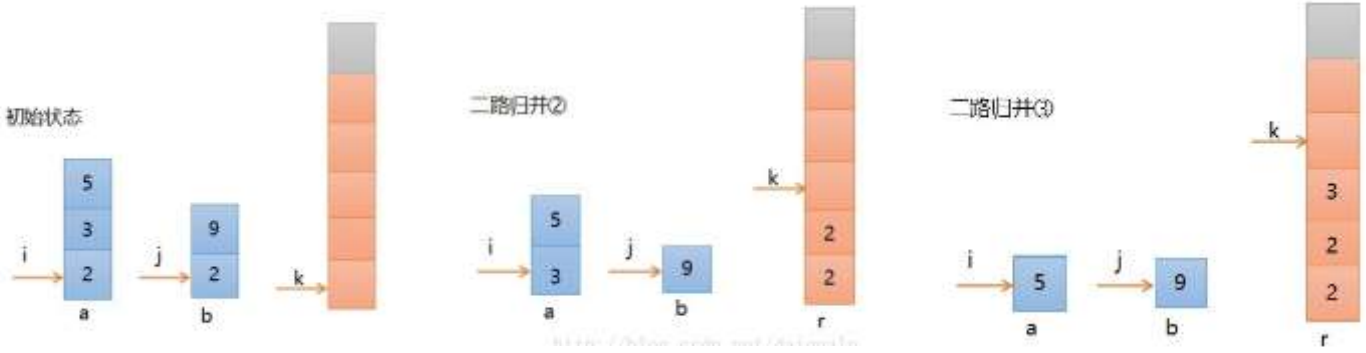
思路： 将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

整体流程



二路归并

P23 void merge()



2.1.1经典算法：分治法

（2）合并排序

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

时间复杂度：*O(nlogn)*
辅助空间：*O(n)*

代码（递归）：

```
void MergeSort(Type a[], int left, int right)
  {
    if (left<right) {
   //至少有2个元素
    int i=(left+right)/2;  //取中点
    MergeSort(a, left, i);
    MergeSort(a, i+1, right);
    //二路归并合并到数组b
    merge(a, left, i, right);  }
  }
```

代码（非迭代）：

```
void MergeSort(Type a[], int n)
  {
    Type *b = new Type[n];
    int s = 1;
    while (s<n)  {
      MergePass(a,b,s,n); //合并到数组b
       s+=s;
       MergePass(b,a,s,n); //合并到数组a
       s+=s;   }
  }
```

P23 void MergePass()

P23 void Merge()

```
void merge(int arr[], int low, int mid, int high){  //二路归并过程
    int i, k;
    int *tmp = (int *)malloc((high-low+1)*sizeof(int));   //申请空间，使其大小为两个
    int left_low = low;
    int left_high = mid;
    int right_low = mid + 1;
    int right_high = high;

    for(k=0; left_low<=left_high && right_low<=right_high; k++){  // 比较两个指针所指向的元素
        if(arr[left_low]<=arr[right_low]){
            tmp[k] = arr[left_low++];
        }else{
            tmp[k] = arr[right_low++];
        } }

    if(left_low <= left_high){  //若第一个序列有剩余，直接复制出来粘到合并序列尾
    for(i=left_low;i<=left_high;i++)
        tmp[k++] = arr[i];
    }

    if(right_low <= right_high){
    //若第二个序列有剩余，直接复制出来粘到合并序列尾
        for(i=right_low; i<=right_high; i++)
            tmp[k++] = arr[i];
    }

    for(i=0; i<high-low+1; i++)
        arr[low+i] = tmp[i];
    free(tmp);
    return;
}
```

2.1.1经典算法：分治法

（3）快速排序

问题：用分治策略对n个元素进行排序。

思路：在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

算法的基本思想

1. 分解(Divide)：将输入的序列L[p..r]（其中L(p)=x）划分成两个非空子序列L[p..q-1]、L[q]=x和L[q+1..r]，使**L[p..q-1]的值，小于等于**x；L[q+1..r]**中任一元素**的值大于等于x 。

2. 递归求解(Conquer)：通过递归调用快速排序算法分别对L[p..q-1]和L[q+1..r]进行排序。

3. 合并(Merge)：由于对分解出的两个子序列的排序是就地进行的，所以在L[p..q]和L[q+1..r]都排好序后不需要执行任何计算L[p..r]就已排好序。

2.1.1经典算法：分治法

（3）快速排序

$$T(n)=\begin{cases} O(1) & n<=1 \\ T(n\text{-}1)+O(n) & n>1 \end{cases}$$

$$T(n)=\begin{cases} O(1) & n<=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

最差：$O(n^2)$ 最好：$O(nlogn)$

时间复杂度

代码（递归）：

```
void qSort(Type a[], int p, int r)
{
    if (p<r) {
        int q=partition(p,r);
        qSort (p,q-1); //对左半段排序
        qSort (q+1,r); //对右半段排序
    }
}
```

```
int partition (type a[],  int p, int r){
    int i = p, j=r+1;
    Type x = a[p];
    while(true){
        while (a[++i] < x&&i<r);
        while (a[--j] > x);
        if (i >= j) break;
        swap(a[i], a[j]);
    }
    a[p] = a[j]; //给起点a[p]赋值
    a[j] = x;//将循环到j作为q位置
    return j;
}
```

2.1.1经典算法：分治法

- 时间复杂度对比

$$T(n)=\begin{cases} O(1) & n\le 1 \\ T(n/2)+O(1) & n>1 \end{cases}$$

时间复杂度：$O(\log n)$

二分搜索

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

时间复杂度：$O(n\log n)$

合并排序

$$T(n)=\begin{cases} O(1) & n<=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

最好时间复杂度：$O(n\log n)$

快速排序

- 时间复杂度计算:Master定理

Master定理 对规模为n的问题，通过分治将其分解成k个规模为n/m的子问题，每次递归带来的额外运算为$f(n)=n^d$(d≥0)，则该问题的时间复杂度关系$T(n)=kT(n/m)+f(n)$可如下求解：

$$T(n)=\begin{cases} O(n^d) & k<m^d \\ O(n^d\log_m n) & k=m^d \\ O(n^{\log_m k}) & k>m^d \end{cases}$$

2.1.2经典算法：动态规划

主要思想： 将待求解的问题分解成若干子问题，先求解子问题，再结合子问题的解得到原问题的解。其中，分解得到的子问题往往不是相互独立。

解决的问题满足以下特征：

① 子问题**不独立**；

② 存在**迭代递归关系**；

③ 存在**最优子结构**；（如果总问题是最优解，则所有子问题都是最优解）

④ 利用分治求解，部分子问题被重复计算。

对比项目	分治法	动态规划
分解成子问题	√	√
子问题相互独立	√	×
求解顺序	自顶向下	自底向上
求解部分适用于动态规划的问题	部分子问题重复计算	可不重复计算

算法的基本步骤

1、找出最优解的性质，并刻划其结构特征。——找出最优问题特点

2、递归地定义最优值。——定义递归关系

3、以自底向上的方式计算出最优值。——根据递归关系计算每一步结果

4、根据计算最优值时得到的信息，构造最优解。

2.1.2经典算法：动态规划

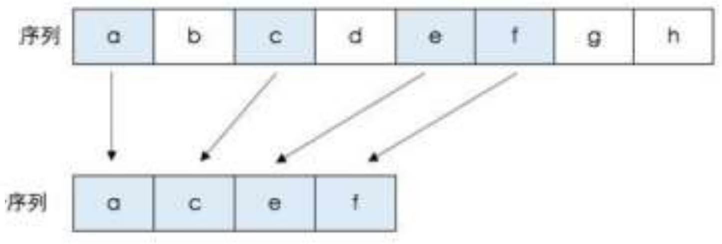
（1）最长公共子序列（Longest Common Subsequence， LCS ）

问题： 给定2个序列$X=\{x_1, x_2,\ldots, x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$，找出X和Y的最长公共子序列。

【子序列概念】 若序列$Z=\{z_1,z_2,\ldots,z_k\}$中元素是按照递增顺序从序列 $X=\{x_1,x_2,\ldots,x_m\}$ 中选取的元素合集，则Z是X的子序列。

【公共子序列概念】 给定序列X和Y，若序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。

思路： ①最优子结构性质分析；② 分析子问题的递归结构建立递归关系；③ 以自底向上的方法计算最优值；④ 构造最优解 。



2.1.2经典算法：动态规划

（1）最长公共子序列（Longest Common Subsequence， LCS ）

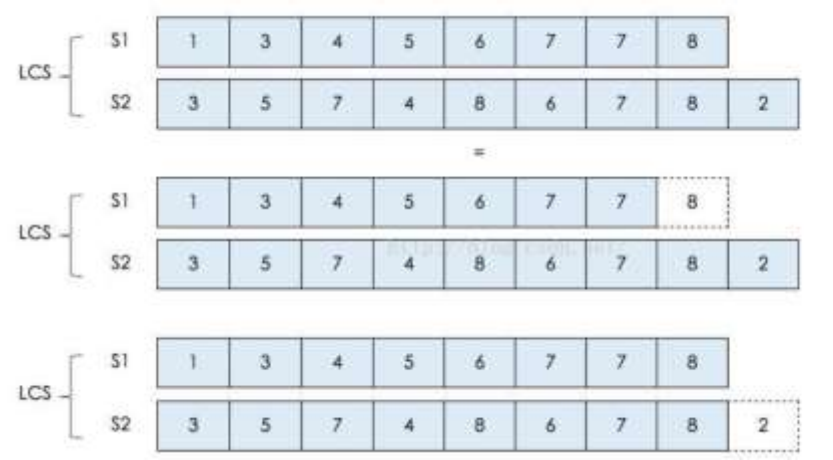
①最优子结构性质分析

设序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z=\{z_1,z_2,\ldots,z_k\}$，则

1)若$x_m=y_n$，则$z_k=x_m=y_n$，且z_{k-1}是x_{m-1}和y_{n-1}的最长公共子序列。

2)若$x_m \neq y_n$且$z_k \neq x_m$，则Z是x_{m-1}和Y的最长公共子序列。

3)若$x_m \neq y_n$且$z_k \neq y_n$，则Z是X和y_{n-1}的最长公共子序列。



结论：2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列，该问题具有最优子结构性质

② 建立递归关系

用c[i][j]记录序列X_i和Y_j的最长公共子序列的长度，当i=0或j=0时，空序列是最长公共子序列，此时C[i][j]=0。

结论：

$$c[i][j]=\begin{cases} 0 & i=0, j=0 \\ c[i-1][j-1]+1 & i,j>0; x_i=y_j \\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0; x_i \neq y_j \end{cases}$$

③ 计算最优值

以自底向上的方法：基础的动态规划

以自顶向下的方法：备忘录方法

④ 构造最优解

输出最长公共子序列

2.1.2经典算法：动态规划

（1）最长公共子序列（Longest Common Subsequence， LCS ）

动态规划代码(P55 LCSLength函数):

```
void LOOKUP_CHAIN (int m,int n ,char []x,char []y,int [][]c,int [][]b){
    int i,j;
    for (i=1;i<=m;i++)        c[i][0]=0;
    for(i=1;i<=n;i++)        c[0][i]=0;
    for(i=1;i<=m;i++)
    for(j=1;j<=n;j++){
     if(x[i]==y[j]){c[i][j]=c[i-1][j-1]+1;      b[i][j]=1;}
     else if (c[i-1][j]>=c[i][j-1]){c[i][j]=c[i-1][j];b[i][j]=2;}
     else{ c[i][j]=c[i][j-1]; b[i][j]=3; }
    }}}
```

时间复杂度：*O(mn)*

备忘录方法代码：

```
int LOOKUP_CHAIN(char *x,char *y,int i,int j){
  if(c[i][j]>-1)  return c[i][j];
  if(i==0||j==0) c[i][j]=0;
  else{if(x[i-1]==y[j-1])c[i][j]=LOOKUP_CHAIN(x,y,i1,j1)+1;
    else
      c[i][j]=max(LOOKUP_CHAIN(x,y,i,j1), LOOKUP
      _CHAIN (x,y,i-1,j));}
    return c[i][j];}
```

示例题目

X=abcf, Y=adct，求最优子结构c[i][j]的值。

	j=0	j=1	j=2	j=3	j=4
i=0	0	0	0	0	0
i=1	0	1	1	1	1
i=2	0	1	1	1	1
i=3	0	1	1	2	2
i=4	0	1	1	2	2

2.1.2经典算法：动态规划

（2）最大子段和问题

问题： 给定由n个整数(可能为负整数)组成的序列a_1, a_2，..an,求该序列形如$\sum_{k=i}^{j} a_k$的子段和的最大值。

【情况说明】 当所有整数均为负整数时定义其最大子段和为0，依此定义,所求的最优值为max {0, max $\sum_{k=i}^{j} a_k$(1≤i≤j≤n)}

思路： ①最优子结构性质分析；②分析子问题的递归结构建立递归关系；③以自底向上的方法计算最优值；④构造最优解。

对比蛮力法： 通过双层 for 循环，遍历所有可能存在的连续子序列，并计算每个子序列的和；比较所有子序列的和，选取序列值最大的那个子序列。

（a_1，a_2，a_3，a_4，a_5）=（-2，11，-4，13，-5，-2）时，

最大子段和为$\sum_{k=2}^{4} a_k$=20。

2.1.2经典算法：动态规划

（2）最大子段和问题

①最优子结构性质分析和②递归关系建立

设有数组a,子段和数组b，其b[j]为：以a[j]做结尾，且包含a[j]的子段所能得到最大和。则b[j]为：

$$b[j] = max_{1\le i\le j}\{\sum_{k=i}^{j} a[k]\}, 1\le j\le n$$

由b[j]的定义可知： 当b[j-1]>0 时：b[j] = b[j-1]+a[j]，否则 b[j] = a[j]。由此可得b[j]的动态规划递归式：

$$b[j] = max\{b[j-1]+a[j], a[j]\}\ (1\le j\le n)$$

③ **计算最优值：** 动态规划

若b[j-1]<0或b[j]=a[j]，说明起始位置为j；

若b[j-1]>0或b[j]≠a[j]，则说明起始位置来源b[j-1]的起始位置，再根据b[j-1-1]是否大于0进行判断。

④ **构造最优解：** 输出最大子段和和起始位置

index	j=1	j=2	j=3	j=4	j=5	j=6
数组 a	-2	11	-4	13	-5	-2
子段和 b	-2	11	7	20	15	13

动态规划代码(P59 MaxSum函数):

```
    int *b;    //定义一个动态全局数组b，b[j]存储的以a[j]做结尾且包含a[j]的子段的
    int maxsum;
    int bestj=0;  //存储具有最大子段和的结束位置j
//动态规划求解最大子段和的最优值
//数组a是序列
int MaxSum_Dp(int n, int *a) {
     b=new int[n];    //动态数组b的长度与a相同
     b[0]=a[0];        //b[j]存储的以a[j]做结尾且包含a[j]的子段的能达到的最大和
     maxsum=a[0];     //maxsum为 b数组中最大元素

     for(int j = 1; j < n; j++) {
        if (b[j-1] > 0)        //若b[j-1]>0,则b[j]=b[j-1]+a[j]
           b[j] =b[j-1]+a[j];
        else
           b[j] =a[j];         //否则, b[j]=a[j]

        if (b[j]>b[j-1])
          {maxsum=b[j];
      bestj=j;
            }
        else
           {maxsum=maxsum; //不更新
          bestj=bestj;
          }
}
       return maxsum;
       return bestj;
}
```

2.1.2经典算法：动态规划

（3）0/1背包

问题： 给定n种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为C。若x是一组解，且x的取值只能是0或者1；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?

思路： ①最优子结构性质分析；②分析子问题的递归结构建立递归关系；③以自底向上的方法计算最优值；④构造最优解。

目标函数： $\max \sum_{i=1}^{n} v_i x_i$

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

	商品1	商品2	商品3	商品4	商品5
价值	v1	v2	v3	v4	v5
重量	w1	w2	w3	w4	w5
是否取	**0**	**1**	**1**	**1**	**0**

2.1.2经典算法：动态规划

（3）0/1背包

①最优子结构性质分析

若(y1,y2……yn)，是背包容量为C时的一组最优解；则（ y2……yn)是背包容量为C-w1y1的最优解，因此满足最优子结构。

考虑前i个物品装入容量为j的背包所能获得的最大价值为m[i][j]表示：

1）当第i个物品的重量wi超过背包容量j时，第i个物品不能装入；则有：

$$m[i][j]=m[i-1][j] \quad //i个商品不装入，0$$

2）当第i个物品的重量wi不超过背包容量j时，第i个物品可装入可不装入，则有：

$$m[i][j]=max(\ m[i-1][j]\ ,\ \ m[i-1][j-wi]+vi)$$

③ 计算最优值： 动态规划(或P75 knapsack函数):

```
//动态规划求解0/1背包
//子结构dp[i][j]表示前i个商品为可选商品，背包容量为j的能获得的最大价值
    for(int i=1; i<=m; i++) //物品
        for(int j=c; j>=0; j--) //容量
        {
            if(j >= w[i])
                dp[i][j] = max(dp[i-1][j-w[i]]+val[i], dp[i-1][j]);  //dp[1][c] 当前最优解
            else        //只是为了好理解
                dp[i][j] = dp[i-1][j];   //第i个商品不选
        }
        cout<<"最大价值:" ;
    cout << dp[m][c] << endl;
```

④ 构造最优解： ： 输出哪些商品取哪些商品不取

```
//构造最大价值的解
    for(int i=m;i>=1;i--)
    { if(dp[i][c]==dp[i-1][c]) //说明从前i个商品中选，与从前i-1个商品中选，最大价值一样，即第i个商品不拿
        x[i]=0;
        else
        x[i]=1;
    }
    cout<<"输出m个商品的取舍方案:" ;

    for(int i=1; i<=m; i++)
    cout<<x[i]<< " ";
```

2.1.3经典算法：贪心算法

主要思想： 在对问题求解时总是做出在当前看来是最好的选择，也就是说贪心法不从整体最优上加以考虑，所做出的仅是在某种意义上的局部最优解。

解决的问题满足以下特征：

① **贪心选择性质：** 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

② **最优子结构性质；** 如果一个问题的最优解包含其子问题的最优解，则称此问题具有最优子结构性质。最优子结构性质是可用动态规划算法或贪心法求解的关键特征

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。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

2.1.3经典算法：贪心算法

（1）单源最短路径

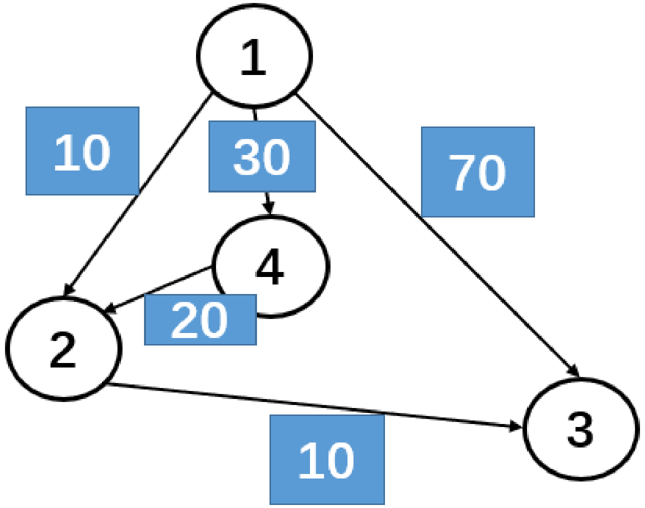
问题：给定带权有向图G =(V,E)，其中每条边的权是非负实数。给定V中的一个顶点，称为源。计算从源到所有其它各顶点的最短路长度。即通路上各边权之和。

【使用贪心算法解决】Dijkstra算法　　狄克斯特拉/迪杰斯特拉算法

思路：①1到j的最短距离只有两种来源：来源于1→j(直连)或者1→？→j(经过某个结点，再连到j)；

② 对于点2而言：由于1与2的距离为10，1与3的距离为70，1到4最短距离30，所以对于2，其最短距离不可能来源于1经过其他点达到2。所以1→2的最短距离应为其直连距离10。

③ 对于点3而言：最短距离，来源于1直连3，或者1经过2到3的距离所以min(1直连3,1到2的最短距离+a[2][3])



2.1.3经典算法：贪心算法

（1）单源最短路径

算法基本思想：

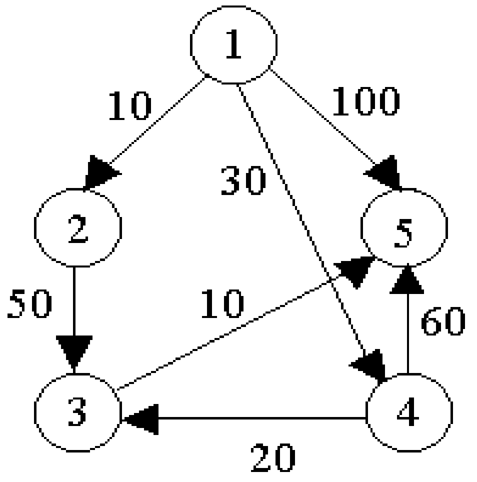
①、设置顶点集合S并不断地作贪心选择来扩充这个集合。 最开始顶点集合只有源v；

②、每次计算源V经过集合S中的点（包括直达及经过其他点达到），到其他所有顶点的最短路径dist[i]；（其中不直接相连的点，认为线上的权是很大的值；）

③、选择此时具有最短距离dist[i]的顶点，被贪心选择进顶点集合S，更新S；（被更新到S中的点，说明v到其的最短距离已经被确定）

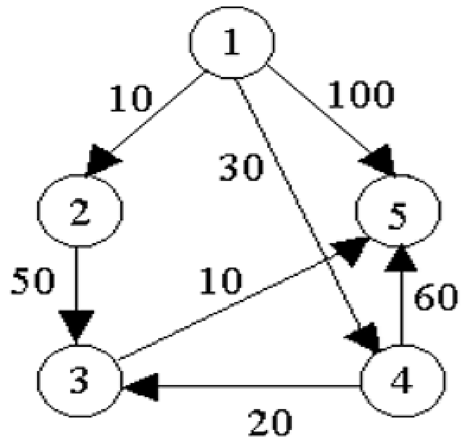
④、再次执行②、③；（后续迭代时，v到顶点i的距离，计算方法，对比dist[u]+a[u][i] 与 dist[i]大小对比）

⑤ 、直到所有的顶点都被更新进S即可，此时计算的dist[i]，即为当前带权有向图的最小距离；



2.1.3经典算法：贪心算法

（1）单源最短路径



Dijkstra算法的迭代过程：

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10 (最短距离确定不更新)	60 (更新为1-2-3)	30 (1-2-4距离更长，不需更新)	100 (1-2-5更长，不需要更新)
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

2.1.4经典算法：回溯法

主要思想： 类似穷举的搜索尝试：在解空间树中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。类似于深度优先的思路搜索整个**解空间**，同时在求解过程中，降低无用搜索，加入**剪枝函数**。

问题的解空间:

【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。

【可行解】解空间中满足约束条件的决策序列。

【最优解】在约束条件下使目标达到最优的可行解。

【溯法搜索过程】

具剪枝函数的深度优先生成法称为回溯法。构造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**，求解过程中根据**剪枝函数**减少无效搜索。

【求解步骤】

(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；

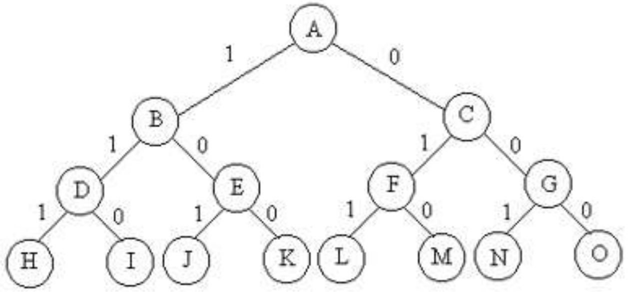
(3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

2.1.4经典算法：回溯法

解空间树: 问题的解空间一般用树或图的形式来组织，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。解空间树一般有如下两种。

子集树：满足某种性质的集合

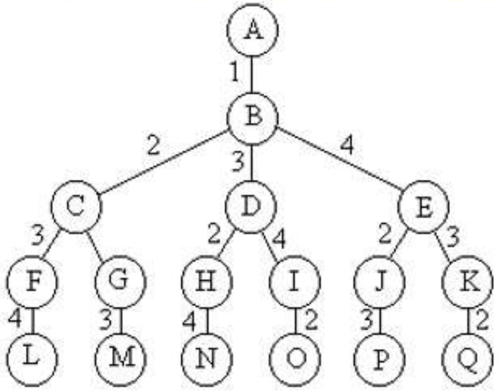
解空间为子集树： 0/1背包、复杂装载问题、*m*着色



遍历子集树需$O(2^n)$计算时间

排列树：满足某种性质的排列

解空间为排列树： 旅行售货员问题



遍历排列树需要$O(n!)$计算时间

剪枝函数： 避免无效搜索，提高回溯的搜索效率，一般有如下两类

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。

2.1.4经典算法：回溯法

（1）装载问题

问题： 有n个货物,要装上两艘容量为c1和c2的轮船，其中货物i的重量为w_i，且$w_1+w_2+\ldots+w_n \le c1+c2$。如何成功将所有货物装上这两艘轮船。如：当$n=3$，c1=50，c2=48，w={5，45，47}，应该怎么装？

思路： 将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，在承载范围内，使该子集中集装箱重量之和最大。由此可知，装载问题等价于特殊的0-1背包问题。

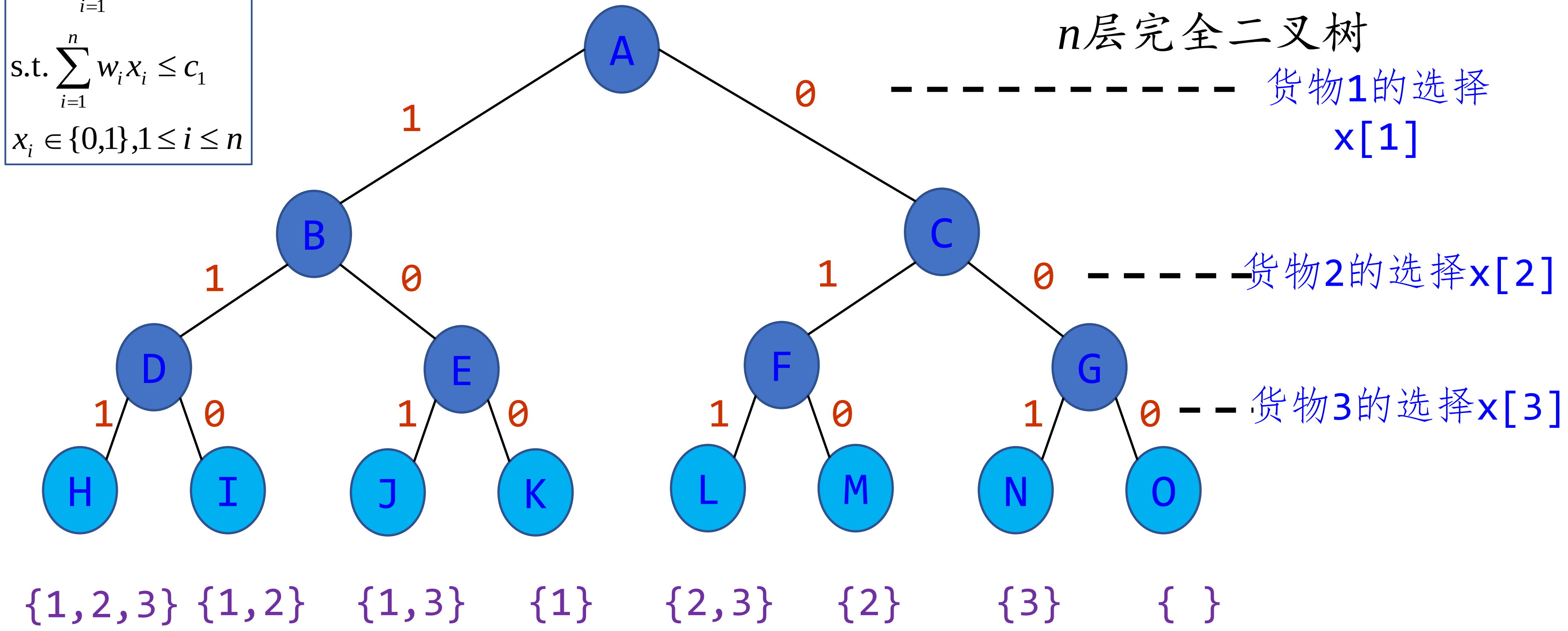
$$\max \sum_{i=1}^{n} w_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

解空间树：子集树

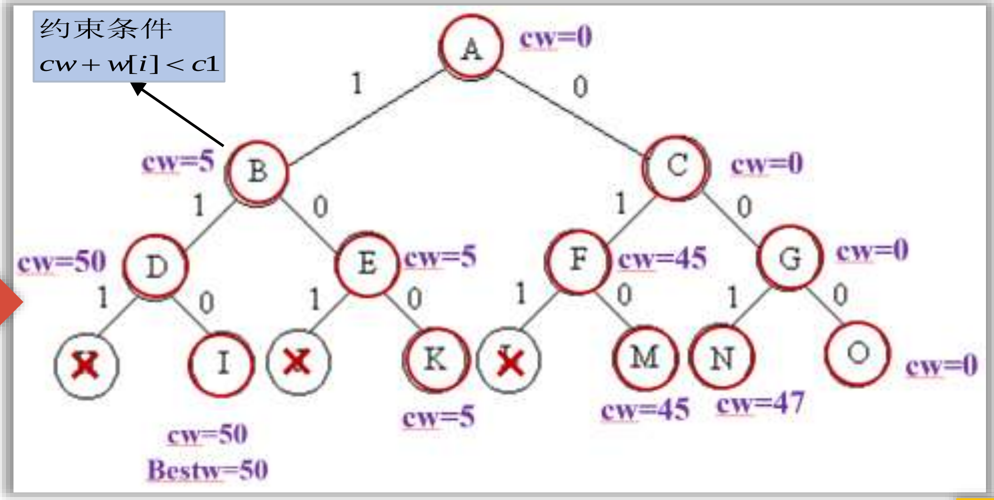
n层完全二叉树



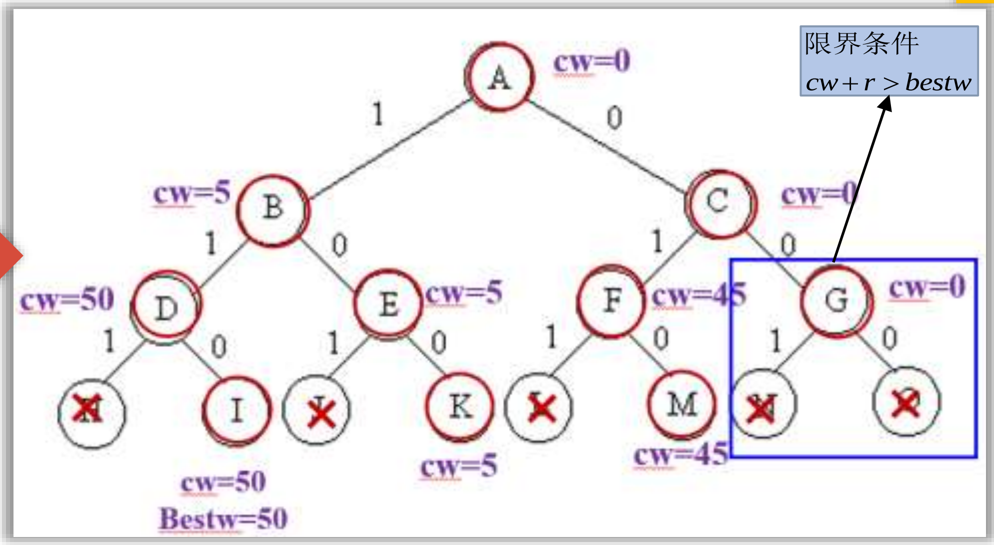
2.1.4经典算法：回溯法

（1）装载问题

剪枝函数：【约束函数】 $cw+w[i]<c1$，其中cw为当前节点装载量，$bestw$为当前最好的总装载。约束条件控制左子树是否搜索：满足约束条件则搜索左子树。



剪枝函数：【限界函数】 $cw+r \geq bestw$，其中r为剩余集装箱重量，$bestw$为当前最好的总装载。限界条件控制右子树是否搜索：满足限界条件则搜索右子树。



回溯法(P126 Backtrack函数):

```
void backtrack (int i)

{    ……
 //r为判断到第i个货物时，剩下货物重量和
        r -= w[i];
      // 约束条件，搜索左子树
      if (cw + w[i] <= c) {
          x[i] = 1;
           cw += w[i];
           backtrack(i + 1);
           cw -= w[i]; }
       // 限界条件，搜索右子树
        if (cw + r > bestw)    {
           x[i] = 0;
           backtrack(i + 1); }
           r += w[i];

 }
```

2.1.4经典算法：回溯法

（2）0/1背包问题

问题： 给定n种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为C。若x是一组解，且x的取值只能是0或者1；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?

思路： 总体思路与装载问题类似，但装载问题要求容量满足情况下，重量最大；0/1背包问题要求容量满足的情况下，价值最大。

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

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

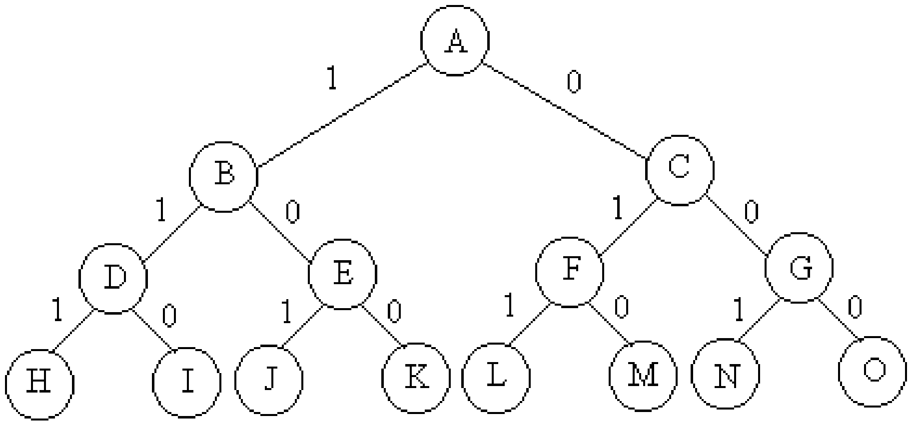
$$x_i \in \{0,1\}, 1 \le i \le n$$

剪枝函数

【约束函数】 $cw+w[i]<c$

【限界函数】 bound：商品价值

解空间树：子集树



回溯法(P138 Backtrack):

```
Void Backtrack(int i){
……
 if (cw + w[i] <= c)      {
        // 搜索左子树
         x[i] = 1;
         cw += w[i];
         cp += p[i];
         backtrack(i + 1);
         cw -= w[i];
         cp -= p[i]; }
if (Bound(i+1)>bestp)
       // 搜索右子树
       backtrack(i + 1); }
```

2.1.4经典算法：回溯法

（3）*m*着色问题

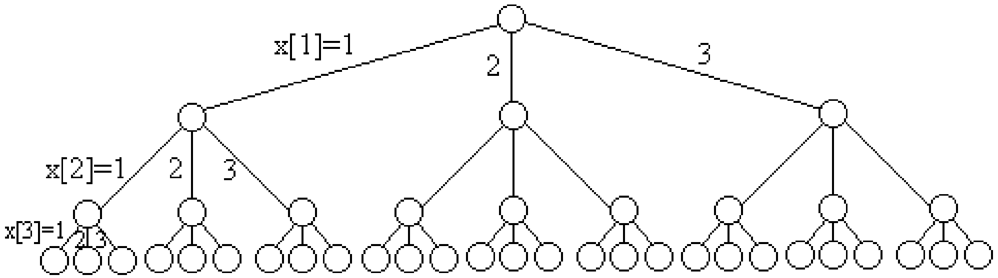
问题： 给定无向连通图*G*和*m*种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。如果有一种着色法使G中每条边的两个顶点着不同颜色，则称这个图是m可着色的。是对于给定图G和m种颜色，找出所有不同的着色法。

思路： 对于图G，采用邻接矩阵a存储，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。

剪枝函数【约束函数】 顶点i与已着色的相邻顶点颜色不重复。若a[i][j]=1，则x[i] ≠ x[j]，其中 x[i]表示顶点i所着颜色。

解空间树：子集树

*n*层完全二叉树



回溯法(P143 Backtrack函数):

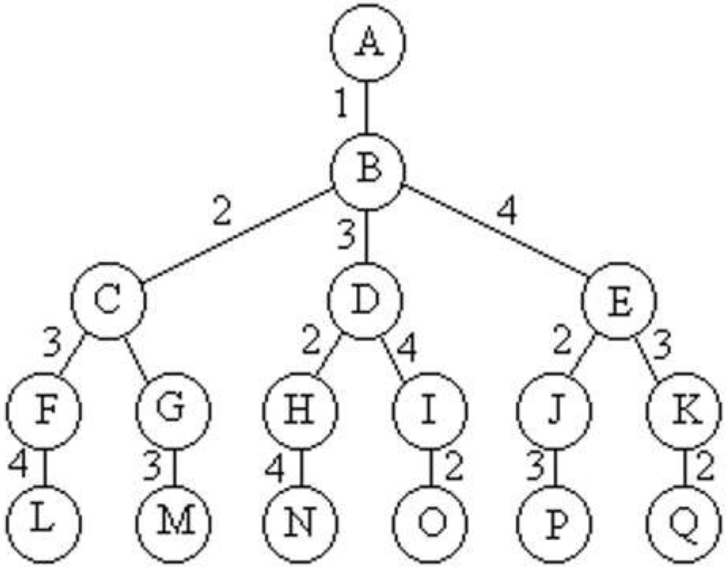
```
void backtrack(int t)    //递归回溯
  {
    if (t>n) sum++; //达到叶子结点，着色方案数增1
    else
      for (int i=1;i<=m;i++) {   //试探每一种着色
        x[t]=i;  //试探着色i
        if (ok(t)) // 检查是否满足约束
          backtrack(t+1); //可以着色i，进入下一个顶点着色
      }
  }
```

2.1.4经典算法：回溯法

（4）旅行售货员

问题：某旅行商要到若干城市去销售商品，已知各城市之间的路程，现在他要选择一条路线，该路线要求满足每个城市只拜访一次、最后要回到原来出发的城市、总路程最小。

解空间树：排列树



[*] 16-回溯法-旅行商.cpp

```
/**回溯法-旅行商(TSP)问题*/
#include<iostream>
#include<algorithm>
#define MAX 100
using  namespace std;
int n;                                  //城市个数
int a[MAX][MAX];                        //城市间距离，如果距离不为0，表示两个节点联通
int x[MAX];                             //记录路径，x[t]表示深度为t时，选择的城市
int bestx[MAX]  = {0};                  //记录最优路径
int bestp = 63355;                      //最短路径长
int cp = 0;                             //当前路径长
void backpack(int t){
    if(t>n){
        if((a[x[n]][1])&&(a[x[n]][1]+cp<bestp)){   //搜索到叶子结点，x[n]表示第n个到达的城市，a[x[n]][1]判定最后一个达到的城市与出发城市1的连通性；
                                                   //x[n]与城市1联通，且路径总和比之前的bestcp小的话，则更新
            bestp = a[x[n]][1]+cp;
            for(int i = 1;i<=n;i++){
                bestx[i] = x[i];     //具有bestp的路径上城市，为最佳路径的规划，输出
            }
        }
    }else{
        for(int i = t;i<=n;i++){
            /*约束为当前节点到下一节点的长度不为0,限界为走过的长度+当前要走的长度之和小于最优长度*/
            if((a[x[t-1]][x[i]])&&(cp+a[x[t-1]][x[i]]<bestp)){
                swap(x[t],x[i]);         // 通过swap交换构造每一层路径上的城市编号
                cp+=a[x[t-1]][x[t]];     // 对于深度t，如果x[t]选中某个城市，则路径长增加 a[x[t-1]][x[t]]
                backpack(t+1);           //递归
                cp-=a[x[t-1]][x[t]];     // 回溯，cp恢复
                swap(x[t],x[i]);         // 交换也恢复
            }
        }
    }
}

int main(){
    cout<<"输入城市个数:"<<endl;
```

2.1.5经典算法：分支限界法

主要思想：分支限界法类似于回溯法，也是一种在问题的解空间树上搜索问题解的算法。以广度优先或以最小耗费优先的方式搜索解空间树。

分类：根据从活结点表中选择下一扩展结点的不同方式，可分为**队列式分支限界法**、**优先队列式分支限界法**。

方法	解空间搜索方式	存储结点的数据结构	结点存储特性	常用应用
回溯法	深度优先	栈（先进后出）	活结点的所有可行子结点被遍历后才从栈中出栈	找出满足条件的所有解
分支限界法	广度优先	队列（先进先出，从队首删除，从队尾插入）优先队列（插入队列时，根据优先级插入；删除时从队首删除）	每个结点只有一次成为活结点的机会	找出满足条件一个解或者特定意义的最优解

【求解步骤】

① 活结点一旦成为扩展结点，就一次性产生其所有儿子结点。当前点从活动节点队列中删去；

② 儿子结点中，不可行解或非最优解结点删去，其余儿子结点被加入活结点列表中；

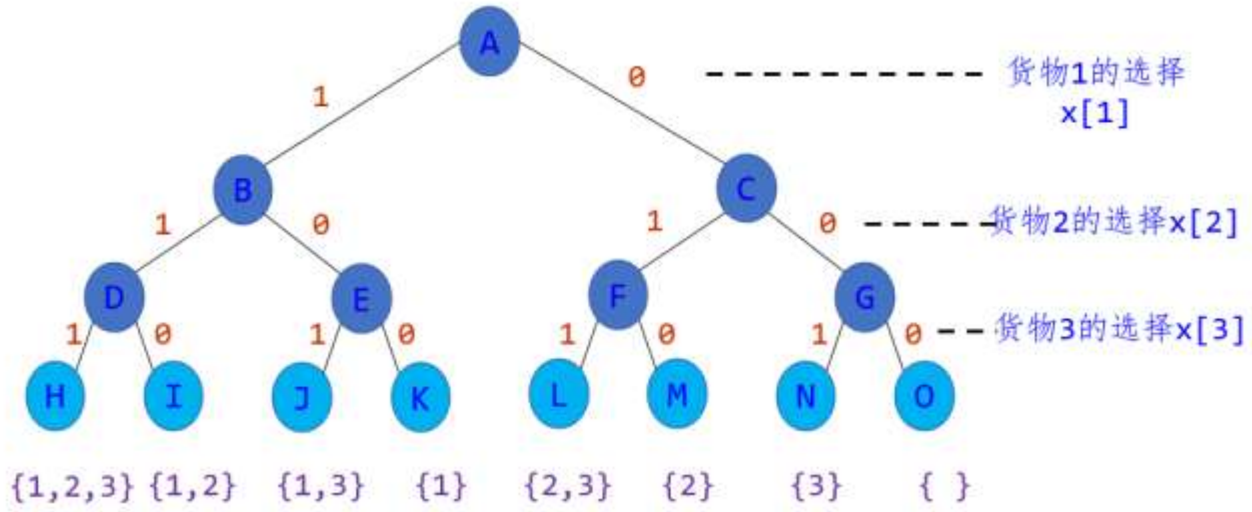
③ 从活结点表中取下一结点成为当前扩展结点；

④ 重复上述结点扩展过程。直至活动结点列表为空；

2.1.5经典算法：分支限界法

（1）0/1背包问题

问题： 给定*n*种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为C。若x是一组解，且x的取值只能是0或者1；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?



【队列式分支限界】P168

A→BC→CE→EFG→KFG→GLM →NO →空

【优先队列式分支限界】P168

优先级可选用：

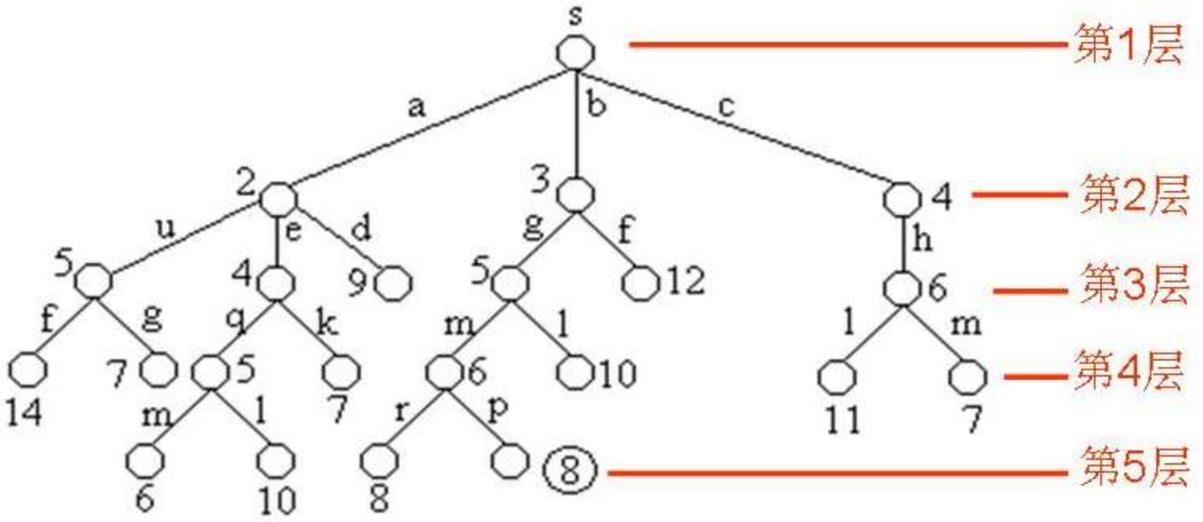
① 结点当前的价值

②可行结点相应子树可能获得最大价值的上界，该上界函数也可同时作为**剪枝函数**。

2.1.5经典算法：分支限界法

（2）单源最短路径

问题： 给定带权有向图G =(V,E)，其中每条边的权是非负实数。给定V中的一个顶点，称为源。计算从源到所有其它各顶点的最短路长度。即通路上各边权之和。



详细分析和代码见课本P170~P173

【剪枝函数】下界函数P168： 结点的霞姐不小于当前找到最短路径，则剪去以该结点为跟的子树。

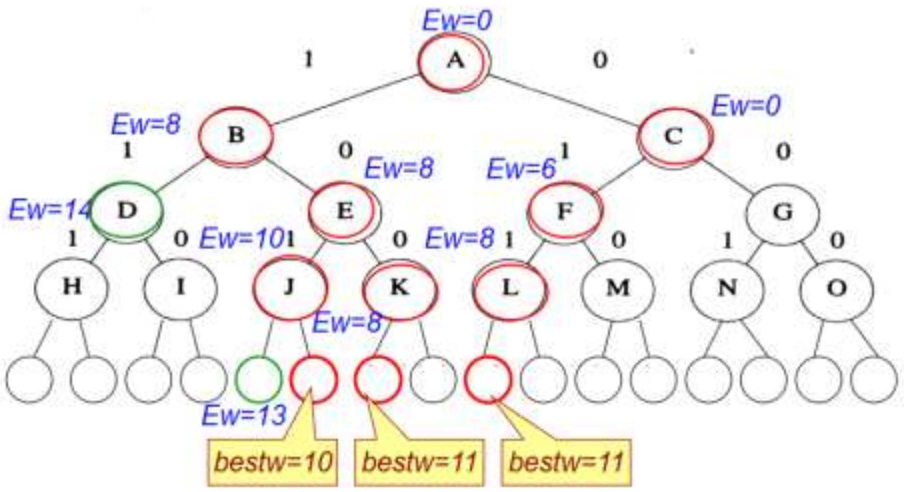
2.1.5经典算法：分支限界法

（3）装载问题

问题：有n个货物,要装上两艘容量为c1和c2的轮船，其中货物i的重量为w_i，且$w_1+w_2+\ldots+w_n \le c1+c2$。如何成功将所有货物装上这两艘轮船。如：当$n$=3，c1=50，c2=48，w={5, 45，47}，应该怎么装？

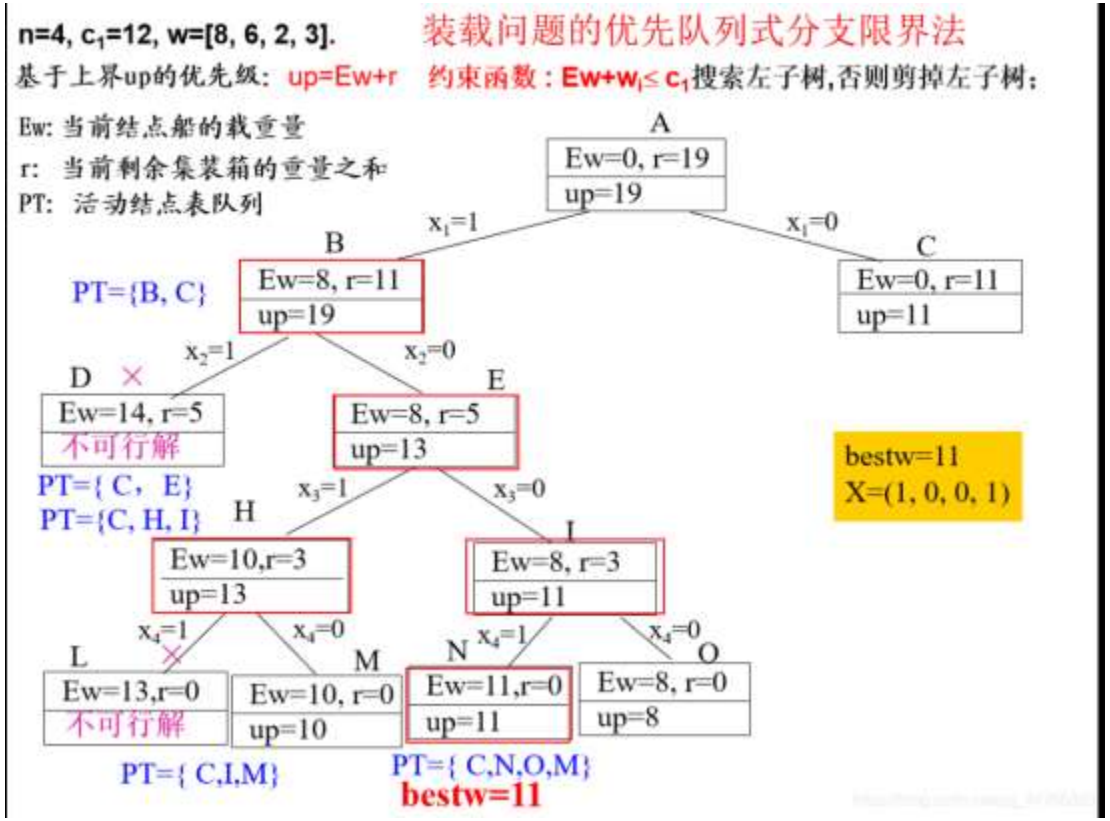
【队列式分支限界】P168

A→BC→CE→EFG→FGJK→GJKLM→JKLMNO→...



【优先队列式分支限界】P168

优先级可选用：①结点当前的价值；②可行结点相应子树可能获得最大价值的上界，该上界函数也可同时作为剪枝函数。



2.2 经典问题索引

0/1背包

问题：给定n种物品和一背包。物品i的重量是wi，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?

解决方法

动态规划 PPT-P18、课本-P74

回溯法 PPT-P28、课本-P137

分支限界 PPT-P32、课本-P181

单源最短路径

问题：给定带权有向图G =(V,E)，其中每条边的权是非负实数。给定V中的一个顶点，称为源。计算从源到所有其它各顶点的最短路长度。即通路上各边权之和。

解决方法

贪心法 PPT-P21、课本-P105

分支限界 PPT-P33、课本-P170

装载问题

问题：有n个货物,要装上两艘容量为c1和c2的轮船，其中货物i的重量为w_i，且$w_1+w_2+\ldots+w_n \le c1+c2$。如何成功将所有货物装上这两艘轮船。

解决方法

贪心法 课本-P100 1艘船版本

回溯法 PPT-P26、课本-P125

分支限界 PPT-P34、课本-P172

2.2 经典问题索引

搜索排序问题

问题：二分搜索、合并排序、快速排序……

解决方法

分治法 PPT-P5~10、课本-第2章

最X问题

问题：最长公共子序列、最大字段和……

解决方法

动态规划 PPT-P13~17、课本-第3章

组合数学问题

问题：*m*着色、旅行售货员……

解决方法

回溯法 PPT-P29~30、课本-第5章

分支限界 课本-第6章

……

问题：Master定理……

解决方法

PPT-P11……