算法设计与分析

蛮力法

讲师：杨根晖 博士

本讲内容

4.1 蛮力法概述

4.2 蛮力法的基本应用

4.3 递归在蛮力法中的应用

4.4 图的深度优先和广度优先遍历

4.1 蛮力法概述

蛮力法是一种简单直接地解决问题的方法，通常直接基于问题的描述和所涉及的概念定义，找出所有可能的解。

然后选择其中的一种或多种解，若该解不可行则试探下一种可能的解。

使用蛮力法通常有如下几种情况：

- 搜索所有的解空间：问题的解存在于规模不大的解空间中。
- 搜索所有的路径：这类问题中不同的路径对应不同的解。
- 直接计算：按照基于问题的描述和所涉及的概念定义，直接进行计算。往往是一些简单的题，不需要算法技巧的。
- 模拟和仿真：按照求解问题的要求直接模拟或仿真即可。

4.2 蛮力法的基本应用

4.2.1 直接采用蛮力法的一般格式

在直接采用蛮力法设计算法中，主要是使用循环语句和选择语句，循环语句用于穷举所有可能的情况，而选择语句判定当前的条件是否为所求的解。

基本格式

```
for (循环变量x取所有可能的值)
{        ┆
         if (x满足指定的条件)
              输出x;
         ┆
}
```

【例4.1】编写一个程序，输出2～1000之间的所有完全数。所谓完全数，是指这样的数，该数的各因子（除该数本身外）之和正好等于该数本身，例如：

6=1+2+3

28=1+2+4+7+14

解： 先考虑对于一个整数*m*，如何判断它是否为完全数。

从数学知识可知：一个数*m*的除该数本身外的所有因子都在1～*m*/2之间。算法中要取得因子之和，只要在1～*m*/2之间找到所有整除*m*的数，将其累加起来即可。如果累加和与*m*本身相等，则表示*m*是一个完全数，可以将*m*输出。

```
for (m=2;m<=1000;m++)
{   求出m的所有因子之和s;
    if (m==s) 输出s;
}
```

对应的程序如下：

```
void main()
{  int m,i,s;
   for (m=2;m<=1000;m++)
   {  s=0;
      for (i=1;i<=m/2;i++)
         if (m%i==0) s+=i;   //i是m的一个因子
      if (m==s)
         printf("%d ",m);
   }
   printf("\n");
}
```

【例4.3】在象棋算式里，不同的棋子代表不同的数，有以下算式，设计一个算法求这些棋子各代表哪些数字。

	兵	炮	马	卒
+	兵	炮	车	卒
车	卒	马	兵	卒

		兵	炮	马	卒
+		兵	炮	车	卒
	车	卒	马	兵	卒

解： 采用蛮力法时，设兵、炮、马、卒和车的取值分别为a、b、c、d、e。则有：

a、b、c、d、e的取值范围为0～9且均不相等（即$a==b$ || $a==c$ || $a==d$ || $a==e$ || $b==c$ || $b==d$ || $b==e$ || $c==d$ || $c==e$ || $d==e$不成立）。

设：

$$m=a\times1000+b\times100+c\times10+d$$

$$n=a\times1000+b\times100+e\times10+d$$

$$s=e\times10000+d\times1000+c\times100+a\times10+d$$

则有：$m+n==s$

```
void fun()
{   int a,b,c,d,e,m,n,s;
    for (a=1;a<=9;a++)
      for (b=0;b<=9;b++)
        for (c=0;c<=9;c++)
          for (d=0;d<=9;d++)
            for (e=0;e<=9;e++)
              if (a==b || a==c || a==d ||
                  a==e || b==c || b==d ||
                  b==e || c==d || c==e ||  d==e)
                     continue;
              else
              {  m=a*1000+b*100+c*10+d;
                 n=a*1000+b*100+e*10+d;
                 s=e*10000+d*1000+c*100+a*10+d;
                 if (m+n==s)
                    printf("兵:%d 炮:%d 马:%d卒:%d 车:%d\n",
                               a,b,c,d,e);
              }
}
```

4.2.2 简单选择排序和冒泡排序

【问题描述】对于给定的含有n个元素的数组a，对其按元素值递增排序。

1. 简单选择排序

例如，i=3的一趟简单选择排序过程，其中a[0..2]是有序的，从a[3..9]中挑选最小元素a[5]，将其与a[3]进行交换，从而扩大有序区，减小无序区。

```
void SelectSort(int a[],int n)
//对a[0..n-1]元素进行递增简单选择排序
{  int i,j,k;

   for (i=0;i<n-1;i++)         //进行n-1趟排序
   {   k=i;                    //用k记录每趟无序区中最小元素的位置
       for (j=i+1;j<n;j++)     //在a[i+1..n-1]中穷举找最小元素a[k]
          if (a[j]<a[k])
             k=j;
       if (k!=i)               //若a[k]不是最小元素,将a[k]与a[i]交换
          swap(a[i],a[k]);
   }
}
```

2. 冒泡排序

例如，i=3的一趟冒泡排序过程，其中a[0..2]是有序的，从a[3..9]中通过交换将最小元素放在a[3]处，从而扩大有序区，减小无序区。

```
void BubbleSort(int a[],int n)
//对a[0..n-1]按递增有序进行冒泡排序
{  int i,j; int tmp;
   bool exchange;

   for (i=0;i<n-1;i++)              //进行n-1趟排序
   {    exchange=false;             //本趟排序前置exchange为false
        for (j=n-1;j>i;j--)         //无序区元素比较,找出最小元素
           if (a[j]<a[j-1])         //当相邻元素反序时
           {  swap(a[j],a[j-1]);     //a[j]与a[j-1]进行交换
              exchange=true;        //发生交换置exchange为true
           }
        if (exchange==false)        //本趟未发生交换时结束算法
          return;
   }
}
```

4.2.3 字符串匹配

【问题描述】对于字符串*s*和*t*，若*t*是*s*子串，返回*t*在*s*中的位置（*t*的首字符在*s*中对应的下标），否则返回-1。

【问题求解】采用直接穷举法求解，称为BF算法。该算法从s的每一个字符开始查找，看t是否会出现。例如，s=“*aababcde*”，t=“*abcd*”：

第1趟

s: *a a b a b c d e*

t: *a b c d*

第2趟

s: *a a b a b c d e*

t: *a b c d*

第3趟

s: *a a b a b c d e*

t: *a b c d*

第4趟

s: *a a b a b c d e*

t: *a b c d*

成功：返回7-4=3

```
int BF(string s,string t)    //字符串匹配
{  int i=0,j=0;
   while (i<s.length() && j<t.length())
   {  if (s[i]==t[j])         //比较的两个字符相同时
      {  i++;
         j++;
      }
      else                    //比较的两个字符不相同时
      {  i=i-j+1;             //i回退
         j=0;                 //j从0开始
      }
   }
   if (j==t.length())         //t的字符比较完毕
      return i-j;             //t是s的子串,返回位置
   else                       //t不是s的子串
      return -1;              //返回-1
}
```

【例4.5】有两个字符串*s*和*t*，设计一个算法求*t*在s中出现的次数。例如，*s*="abababa"，*t*="aba"，则*t*在s中出现2次。

解： 采用BF算法思路。用num记录*t*在*s*中出现的次数（初始时为0）。

当在*s*中找到*t*的一次出现时，num++，此时*j*=*t*的长度，*i*指向*s*中本次出现*t*子串的下一个字符，所以为了继续查找*t*子串的下一次出现，只需要置*j*=0。

```
int Count(string s,string t)            //求t在s中出现的次数
{   int num=0;                          //累计出现次数
    int i=0,j=0;

    while (i<s.length() && j<t.length())
    {   if (s[i]==t[j])                 //比较的两个字符相同时
        {   i++;
            j++;
        }
        else                            //比较的两个字符不相同时
        {  i=i-j+1;                     //i回退
           j=0;                         //j从0开始
        }
        if(j==t.length())
        {   num++;                      //出现次数增1
            j=0;                        //j从0开始继续
        }
    }
    return num;
}
```

4.2.4 求解最大连续子序列和问题

【问题描述】给定一个有*n*（*n*≥1）个整数的序列，要求求出其中最大连续子序列的和。

例如：

序列（-2，11，-4，13，-5，-2）的最大子序列和为20

序列（-6，2，4，-7，5，3，2，-1，6，-9，10，-2）的最大子序列和为16。

规定一个序列最大连续子序列和至少是0，如果小于0，其结果为0。

解法1：设含有n个整数的序列$a[0..n-1]$，其中任何连续子序列$a[i..j]$（$i \le j$，$0 \le i \le n-1$，$i \le j \le n-1$）求出它的所有元素之和thisSum。通过比较将最大值存放在maxSum中，最后返回maxSum。

$$a[0]\ \ a[1]\ \dots\ \underbrace{a[i]\ \ a[i+1]\ \dots\ a[j]}_{\text{thisSum}}\ \ a[j+1]\ \dots\ a[n-1]$$

$$\text{thisSum} \ \Big\downarrow \text{MAX} \ \ \text{maxSum}$$

$$\left.\begin{array}{l} i:\ 0\sim n-1 \\ j:\ i\sim n-1 \end{array}\right\}\ k:\ i\sim j$$

```
int maxSubSum1(int a[],int n)
{   int i,j,k;
    int maxSum=a[0],thisSum;
    for (i=0;i<n;i++)             //两重循环穷举所有的连续子序列
    {   for (j=i;j<n;j++)
        {   thisSum=0;
            for (k=i;k<=j;k++)
                 thisSum+=a[k];
            if (thisSum>maxSum)   //通过比较求最大连续子序列之和
                 maxSum=thisSum;
       }
    }
    return maxSum;
}
```

maxSubSum1(a,n)算法中用了三重循环，所以有：

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\frac{1}{2}\sum_{i=0}^{n-1}(n-i)(n-i+1)=O(n^3)$$ 。

解法2： 改进前面的解法，在求两个相邻子序列和时，它们之间是关联的。

例如a[0..3]子序列和=a[0]+a[1]+a[2]+a[3]，a[0..4]子序列和=a[0]+a[1]+a[2]+a[3]+a[4]，在前者计算出来后，求后者时只需在前者基础上加以a[4]即可，没有必须每次都重复计算。从而提高了算法效率。

a[0] a[1] … a[i] a[i+1] … a[j-1] a[j] … a[n-1]

MAX

thisSum

maxSum

thisSum+a[j]

i：0～n-1

j：i～n-1

```
int maxSubSum2(int a[],int n)
{  int i,j;
   int maxSum=a[0],thisSum;

   for (i=0;i<n;i++)
   {  thisSum=0;
      for (j=i;j<n;j++)
      {  thisSum+=a[j];  //maxSum已经包含了a[i..j-1]的最大和
         if (thisSum>maxSum)
            maxSum=thisSum;
      }
   }
   return maxSum;
}
```

maxSubSum2(a,n)算法中只有两重循环，容易求出T(n)=O(n^2)。

解法3：更一步改进解法2。

如果扫描中遇到负数，当前子序列和thisSum将会减小，若thisSum为负数，表明前面已经扫描的那个子序列可以抛弃了，则放弃这个子序列，重新开始下一个子序列的分析，并置thisSum为0。

若这个子序列和thisSum不断增加，那么最大子序列和maxSum也不断增加。

```
int maxSubSum3(int a[],int n)
{   int i,maxSum=0,thisSum=0;
    for (i=0;i<n;i++)
    {   thisSum+=a[i];
        if (thisSum<0)          //若当前子序列和为负数，重新开始下一子序列
            thisSum=0;
        if (maxSum<thisSum)     //比较求最大连续子序列和
            maxSum=thisSum;
    }
    return maxSum;
}
```

显然该算法中仅扫描*a*一次，其算法的时间复杂度为O(n)。

3.2.5 求解幂集问题

【问题描述】 对于给定的正整数n（$n \geq 1$），求1～n构成的集合的所有子集（幂集）。

解法1：采用直接蛮力法求解，将1～n的存放在数组a中，求解问题变为构造集合a的所有子集。设集合a[0..2]={1，2，3}，其所有子集对应的二进制位及其十进制数如下。

子集	对应的二进制位	对应的十进制数
{}	—	—
{1}	001	1
{2}	010	2
{1,2}	011	3
{3}	100	4
{1,3}	101	5
{2,3}	110	6
{1,2,3}	111	7

对于含有*n*（*n*≥1）个元素的集合*a*，求幂集的过程如下：

```
for (i=0;i<2ⁿ;i++)        //穷举a的所有子集并输出
{  将i转换为二进制数b;
   输出b中为1的位对应的a元素构成一个子集;
}
```

显然该算法的时间复杂度为O(n×2^n)，属于指数级的算法。

首先b[0..2]=000，每调用一次inc，b表示十进制数的增加1

```
void inc(int b[], int n)     //将b表示的二进制数增1
{  for(int i=0;i<n;i++)      //遍历数组b
   {  if (b[i])              //将元素1改为0
        b[i]=0;
      else                   //将元素0改为1并退出for循环
      {  b[i]=1;
         break;
      }
   }
}
```

例如:

b[0..2]=0 0 0

↓ 第1次调用：b[0]=0，改为b[0]=1

b=1 0 0

↓ 第2次调用：第1个1改为0，第2个0改为=1

b=0 1 0

↓ 第3次调用：第1个0改为1

b=1 1 0

↓ 第4次调用：前2个1改为0，第3个0改为1

b=0 0 1

↓ 第5次调用：前第1个0改为1

b=1 0 1

…

```
void PSet(int a[],int b[],int n)     //求幂集
{  int i,k;
   int pw=(int)pow(2,n);             //求2^n
   printf("1～%d的幂集:\n  ",n);

   for(i=0;i<pw;i++)                 //执行2^n次
   {    printf("{ ");
        for (k=0;k<n;k++)            //执行n次
           if(b[k])
              printf("%d ",a[k]);
        printf("} ");
        inc(b,n);                    //b表示的二进制数增1
   }
   printf("\n");
}
```

解法2： 采用增量蛮力法求解1～n的幂集，当n=3时的求解过程。

{ }

添加1

{ 1 }

1的幂集：{{ }，{1}}

添加2

{{ 2 }，{1，2}}

1~2的幂集：{ { }，{1}，{ 2 }，{1，2}}

添加3

{{ 3 }，{1，3}，{2，3}，{1，2，3}}

1~3的幂集：
{ { }，{1}，
{ 2 }，{1，2}
{3}，{1，3}，
{2，3}，
{1，2，3}
}

这种思路也是**蛮力法**方法：即穷举1～n的所有子集。先建立一个空子集，对于i（$1 \le i \le n$），每次都是在前面已建立的子集上添加元素i而构成若干个子集，对应的过程如下：

```
void f(int n)            //求1～n的幂集ps
{  置ps={{}};            //在ps中加入一个空子集元素
   for (i=1;i<=n;i++)
       在ps的每个元素中添加i而构成一个新子集;
}
```

在实现算法时，用一个vector<int>容器表示一个集合元素，用vector<vector<int> >容器存放幂集（即集合的集合）。

```
#include <stdio.h>
#include <vector>
using namespace std;
vector<vector<int> > ps;                //存放幂集
```

```
void PSet(int n)                          //求1～n的幂集ps
{   vector<vector<int> > ps1;             //子幂集
    vector<vector<int> >::iterator it;//幂集迭代器
    vector<int> s;
    ps.push_back(s);                      //添加{}空集合元素

    for (int i=1;i<=n;i++)                //循环添加1～n
    {   ps1=ps;                           //ps1存放上一步得到的幂集
        for (it=ps1.begin();it!=ps1.end();++it)
          (*it).push_back(i);             //在ps1的每个集合元素末尾添加i
        for (it=ps1.begin();it!=ps1.end();++it)
          ps.push_back(*it);              //将ps1的每个集合元素添加到ps中
    }
}
```

【算法分析】对于给定的n，每一个集合元素都要处理，有2^n个，所以上述算法的时间复杂度为$O(2^n)$。

4.2.6 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，而且具有最大的价值。

并对下表所示的4个物品求出W=6时的所有解和最佳解。

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

【问题求解】对于n个物品、容量为W的背包问题，采用前面求幂集的方法求出所有的物品组合。

对于每一种组合，计算其总重量sumw和总价值sumv，当sumw小于等于W时，该组合是一种解，并通过比较将最佳方案保存在maxsumw和maxsumv中，最后输出所有的解和最佳解。

```
#include <stdio.h>
#include <vector>
using namespace std;
vector<vector<int> > ps;                    //存放幂集
void PSet(int n)                            //求1～n的幂集ps
{  vector<vector<int> > ps1;                //子幂集
   vector<vector<int> >::iterator it;//幂集迭代器
   vector<int> s;
   ps.push_back(s);                         //添加{}空集合元素

   for (int i=1;i<=n;i++)                   //循环添加1～n
   {    ps1=ps;                             //ps1存放上一步得到的幂集
        for (it=ps1.begin();it!=ps1.end();++it)
          (*it).push_back(i);               //在ps1的每个集合元素末尾添加i
        for (it=ps1.begin();it!=ps1.end();++it)
          ps.push_back(*it);                //将ps1的每个集合元素添加到ps中
   }
}
```

```
void Knap(int w[],int v[],int W)      //求所有的方案和最佳方案
{   int count=0;                      //方案编号
    int sumw,sumv;                    //当前方案的总重量和总价值
    int maxi,maxsumw=0,maxsumv=0;     //最佳方案的编号、总重量和总价值
    vector<vector<int> >::iterator it;       //幂集迭代器
    vector<int>::iterator sit;               //幂集集合元素迭代器
    printf("  序号\t选中物品\t总重量\t总价值\t能否装入\n");

    for (it=ps.begin();it!=ps.end();++it)    //扫描ps中每一个集合元素
    {    printf("  %d\t",count+1);
         sumw=sumv=0;
         printf("{");
         for (sit=(*it).begin();sit!=(*it).end();++sit)
         {    printf("%d ",*sit);
              sumw+=w[*sit-1];               //w数组下标从0开始
              sumv+=v[*sit-1];               //v数组下标从0开始
         }
```

```
        printf("}\t\t%d\t%d  ",sumw,sumv);
        if (sumw<=W)
        {  printf("能\n");
           if (sumv>maxsumv)                 //比较求最优方案
           {  maxsumw=sumw;
              maxsumv=sumv;
              maxi=count;
           }
        }
        else printf("否\n");
        count++;                             //方案编号增加1
     }
     printf("最佳方案为: ");
     printf("选中物品");
     printf("{ ");
     for (sit=ps[maxi].begin();sit!=ps[maxi].end();++sit)
         printf("%d ",*sit);
     printf("},");
     printf("总重量:%d,总价值:%d\n",maxsumw,maxsumv);
}
```

```
void main()
{   int n=4,W=6;
    int w[]={5,3,2,1};
    int v[]={4,4,3,1};
    PSet(n);
    printf("0/1背包的求解方案\n",n);
    Knap(w,v,W);
}
```

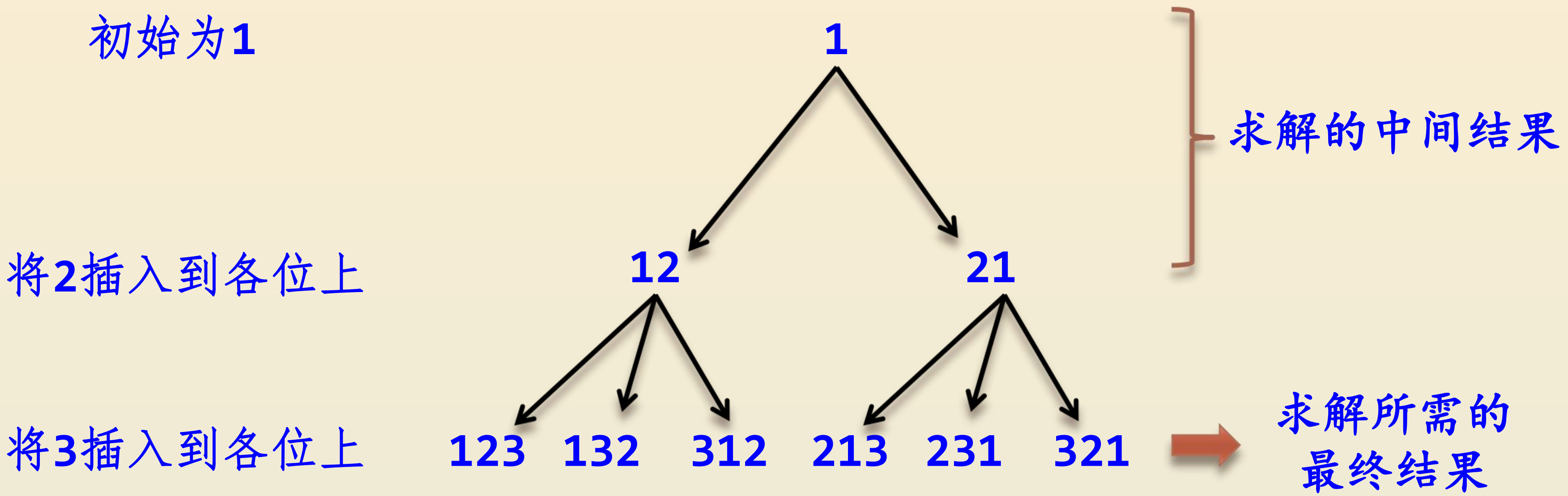
程序执行结果如下:

```
0/1背包的求解方案
  序号  选中物品          总重量          总价值          能否装入
  1          { }            0              0                能
  2          { 1 }          5              4                能
  3          { 2 }          3              4                能
  4          { 1 2 }        8              8                否
  5          { 3 }          2              3                能
  6          { 1 3 }        7              7                否
  7          { 2 3 }        5              7                能
  8          { 1 2 3 }      10             11               否
  9          { 4 }          1              1                能
  10         { 1 4 }        6              5                能
  11         { 2 4 }        4              5                能
  12         { 1 2 4 }      9              9                否
  13         { 3 4 }        3              4                能
  14         { 1 3 4 }      8              8                否
  15         { 2 3 4 }      6              8                能
  16         { 1 2 3 4 }    11             12               否
最佳方案为 选中物品:{ 2 3 4 },总重量:6,总价值:8
```

4.2.7 求解全排列问题

【问题描述】对于给定的正整数n（$n \geq 1$），求1～n的所有全排列。

【问题求解】这里采用增量蛮力法求解。产生1～3全排列的过程如下：



```
void Insert(vector<int> s,int i,vector<vector<int> > &ps1)
//在每个集合元素中间插入i得到ps1
{   vector<int> s1;
    vector<int>::iterator it;
    for (int j=0;j<i;j++)     //在s(含i-1个整数)的每个位置插入i
    {   s1=s;
        it=s1.begin()+j;      //求出插入位置
        s1.insert(it,i);      //插入整数i
        ps1.push_back(s1);    //添加到ps1中
    }
}
```

例如： 1 2 插入3

312 132 123

```
void Perm(int n)                          //求1～n的所有全排列
{  vector<vector<int> > ps1;              //临时存放子排列
   vector<vector<int> >::iterator it;//全排列迭代器
   vector<int> s,s1;
   s.push_back(1);
   ps.push_back(s);                       //添加{1}集合元素

   for (int i=2;i<=n;i++)                 //循环添加1～n
   {  ps1.clear();                        //ps1存放插入i的结果
      for (it=ps.begin();it!=ps.end();++it)
        Insert(*it,i,ps1);                //在每个集合元素中间插入i得到ps1
      ps=ps1;
   }
}
```

【算法分析】对于给定的正整数n，每一种全排列都必须处理，有$n!$种，所以上述算法的时间复杂度为$O(n*n!)$。

4.2.8 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】所谓一种分配方案就是由第i个人执行第j个任务，用（a_1，a_2，…，a_n）表示，即第1个人执行第a_1个任务，第2个人执行第a_2个任务，以此类推。全部的分配方案恰好是1～n的全排列。

这里采用增量穷举法求出所有的分配方案ps（全排列），再计算出每种方案的成本，比较求出最小成本的方案，即最优方案。以n=4，成本如下表所示为例讨论。

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

```
//问题表示
int n=4;
int c[MAXN][MAXN]={ {9,2,7,8},{6,4,3,7},{5,8,1,8},{7,6,9,4} };
vector<vector<int> > ps;      //存放全排列

void Insert(vector<int> s,int i,vector<vector<int> > &ps1)
//在每个集合元素中间插入i得到ps1
{  vector<int> s1;
   vector<int>::iterator it;
   for (int j=0;j<i;j++)     //在s(含i-1个整数)的每个位置插入i
   {    s1=s;
        it=s1.begin()+j;     //求出插入位置
        s1.insert(it,i);     //插入整数i
        ps1.push_back(s1);   //添加到ps1中
   }
}
```

```
void Perm(int n)                                  //求1~n的所有全排列
{   vector<vector<int> > ps1;                     //临时存放子排列
    vector<vector<int> >::iterator it;            //全排列迭代器
    vector<int> s,s1;
    s.push_back(1);
    ps.push_back(s);                              //添加{1}集合元素

    for (int i=2;i<=n;i++)                        //循环添加1~n
    {   ps1.clear();                              //ps1存放插入i的结果
        for (it=ps.begin();it!=ps.end();++it)
            Insert(*it,i,ps1);   //在每个集合元素中间插入i得到ps1
        ps=ps1;
    }
}
```

```
void Allocate(int n,int &mini,int &minc)
//求任务分配问题的最优方案
{  Perm(n);                              //求出全排列ps

   for (int i=0;i<ps.size();i++)    //求每个方案的成本
   {  int cost=0;
      for (int j=0;j<ps[i].size();j++)
         cost+=c[j][ps[i][j]-1];
      if (cost<minc)                    //比较求最小成本的方案
      {  minc=cost;
         mini=i;
      }
   }
}
```

```
void main()
{   int mincost=INF,mini;
               //mincost为最小成本,mini为ps中最优方案编号
    Allocate(n,mini,mincost);
    printf("最优方案:\n");
    for (int k=0;k<ps[mini].size();k++)
        printf("   第%d个人安排任务%d\n",k+1,ps[mini][k]);
    printf("   总成本=%d\n",mincost);
}
```

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

4.3 递归在蛮力法中的应用

蛮力法所依赖的基本技术是遍历技术，采用一定的策略将待求解问题的所有元素依次处理一次，从而找出问题的解。

而在遍历过程中，很多求解问题都可以采用递归方法来实现，如二叉树的遍历和图的遍历等。

4.3.1 用递归方法求解幂集问题

前面介绍了两种采用蛮力法求解由1～*n*整数构成的集合的幂集的方法，这里以解法2为基础。

同样采用vector<vector<int> >容器ps存放幂集，并作为全局变量。初始时ps={{}}。

小问题：$f(i, n)$用于添加i～n整数（共需添加$n-i+1$个整数）产生的幂集ps。

大问题：$f(1, n)$就是生成1～n的整数集合对应的幂集ps。

$f(i,n,\mathrm{p}) \equiv$ 输出幂集p　　当$i>n$

$f(i,n,\mathrm{p}) \equiv$ 将整数i添加到p中原有每个子集中产生新子集；　　否则

并将所有新子集加入到p中；

$f(i+1,n,\mathrm{p})$;

```
vector<vector<int> > ps;                          //存放幂集
void Inserti(int i)
//向幂集ps中每个集合元素添加i并插入到ps中
{  vector<vector<int> > ps1;                      //子幂集
   vector<vector<int> >::iterator it;             //幂集迭代器
   ps1=ps;                                        //ps1存放原来的幂集
   for (it=ps1.begin();it!=ps1.end();++it)
     (*it).push_back(i);      //在ps1的每个集合元素末尾添加i
   for (it=ps1.begin();it!=ps1.end();++it)
     ps.push_back(*it);       //将ps1的每个集合元素添加到ps中
}
```

```
void PSet(int i,int n)    //求1～n的幂集ps
{  if (i<=n)
   {    Inserti(i);
                  //将i插入到现有子集中产生新子集
        PSet(i+1,n);
                  //递归调用
   }
}
```

4.3.2 用递归方法求解全排列问题

同样采用vector<vector<int> >容器ps存放全排列，并作为全局变量。首先初始化ps={{1}}。

大问题：$f(i,n)$用于添加i～n整数（共需添加$n-i+1$个整数）产生的全排列ps。显然$f(2,n)$就是生成1～n的整数集合对应的全排列ps。

小问题：$f(i+1,n)$用于添加$i+1$～n整数（共需添加$n-i$个整数）产生的全排列。

$f(i,n) \equiv$ 产生全排序ps	当$i>n$时
$f(i,n) \equiv$ 置ps为{{1}}，取出ps的每个集合元素s， 在s的每个位置插入i; 将插入i后的新集合元素添加的ps1中; 置ps=ps1; $f(i+1,n)$;	否则

```
vector<vector<int> > ps;                //存放全排列
void Insert(vector<int> s,int i,vector<vector<int> > &ps1)
//在每个集合元素中间插入i得到ps1
{  vector<int> s1;
   vector<int>::iterator it;

   for (int j=0;j<i;j++)     //在s(含i-1个整数)的每个位置插入i
   {    s1=s;
        it=s1.begin()+j;      //求出插入位置
        s1.insert(it,i);      //插入整数i
        ps1.push_back(s1);    //添加到ps1中
   }
}
```

```
void Perm(int i,int n)              //求1～n的全排列ps
{  vector<vector<int> >::iterator it; //全排列迭代器
   if (i<=n)
   {  vector<vector<int> > ps1; //临时存放子排列
      for (it=ps.begin();it!=ps.end();++it)
        Insert(*it,i,ps1);      //在每个集合元素中间插入i得到ps1
      ps=ps1;
      Perm(i+1,n);              //继续添加整数i+1
   }
}
```

4.3.3 用递归方法求解组合问题

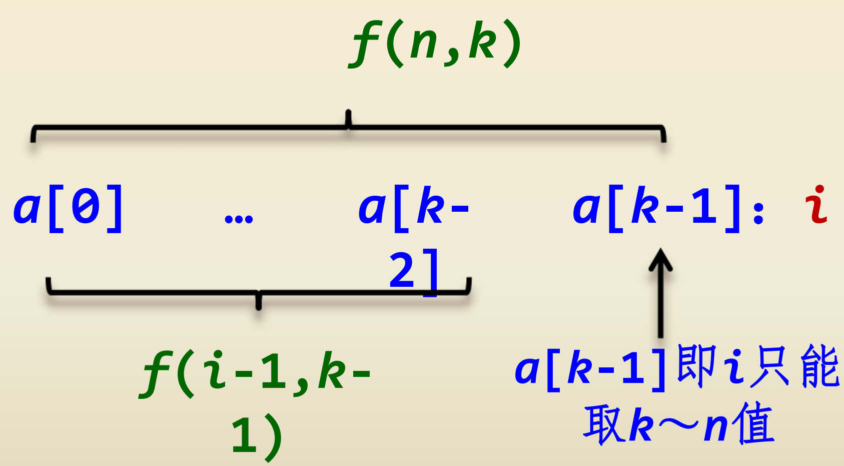
【问题描述】 求1～n的正整数中取k（$k \le n$）个不重复整数的所有组合。

【问题求解】 用数组元素$a[0..k-1]$来保存一个组合，由于一个组合中所有元素不会重复出现，规定a中所有元素按递增排列。

大问题：$f(n,k)$从$1\sim n$中任取k个数的所有组合。

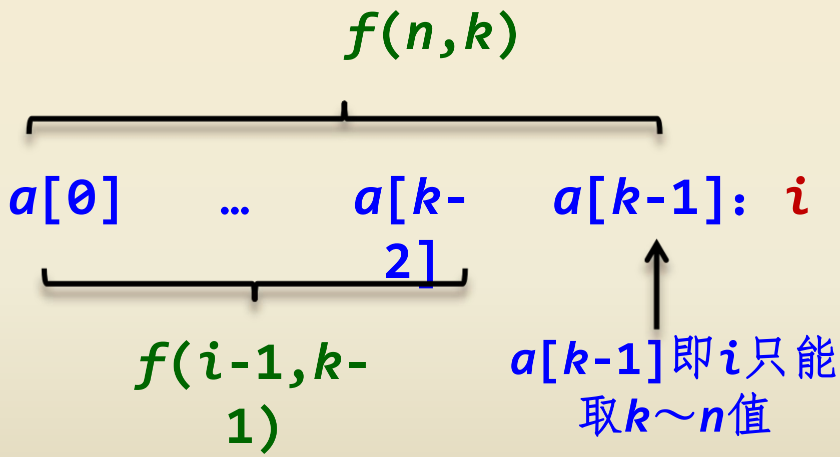
小问题：$f(m,k-1)$从$1\sim m$中任取$k-1$个数的所有组合。

因为a中元素递增排列，所以$a[k-1]$的取值范围只能为$k\sim n$，当$a[k-1]$确定为i后，合并$f(i-1,k-1)$的一个结果便构成$f(n,k)$的一个组合结果。

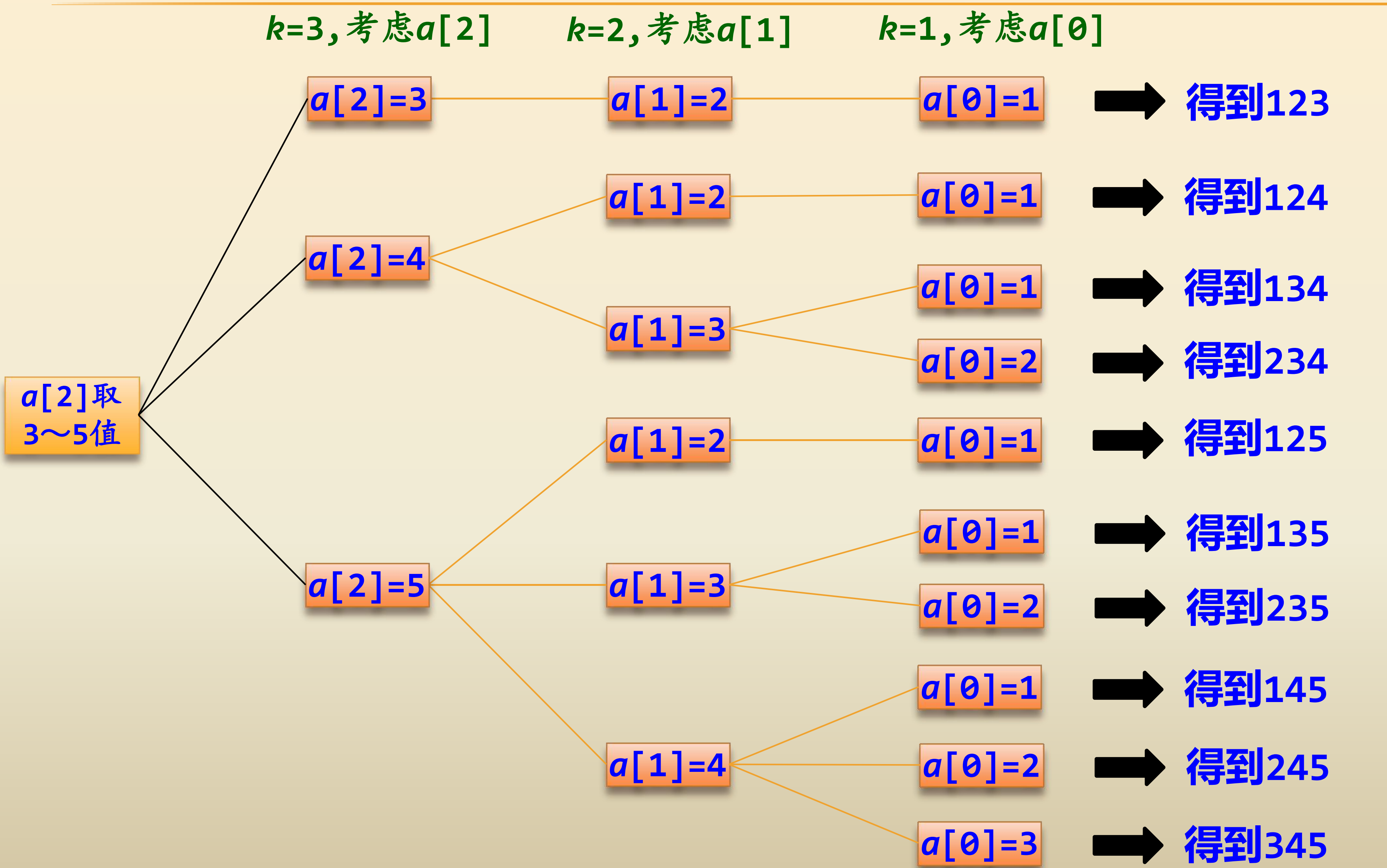


对应的递归模型如下：

$f(n, k) \equiv$ 输出a中的一种组合　　　当k=0
$f(n, k) \equiv$ i取值从k到n:　　　当k>0
　　　　{ $a[k-1]=i$; $f(i-1, k-1)$}



求1～5中取3个整数组合的过程如下所示（a[0..2]放一个组合）



```
void comb(int n,int k)          //求1..n中k个整数的组合
{   if (k==0)                   //k为0时输出一个组合
        dispacomb();

    else
    {    for (int i=k;i<=n;i++)
         {    a[k-1]=i;          //a[k-1]位置取k～n的整数
              comb(i-1,k-1);
         }
    }
}
```

4.4 图的深度优先和广度优先遍历

4.4.1 图的存储结构

- **邻接矩阵**
- **邻接表**

1. 邻接矩阵存储方法

邻接矩阵是表示顶点之间相邻关系的矩阵。设G=(V,E)是含有n（n>0）个顶点的图，各顶点的编号为0～(n-1)，则G的邻接矩阵A是n阶方阵，其定义如下：

（1）如果G是不带权无向图，则：

A[i][j]=1	若(i,j)∈E(G)
A[i][j]=0	其他

（2）如果G是不带权有向图，则：

A[i][j]=1	若<i,j>∈E(G)
A[i][j]=0	其他

1. 邻接矩阵存储方法

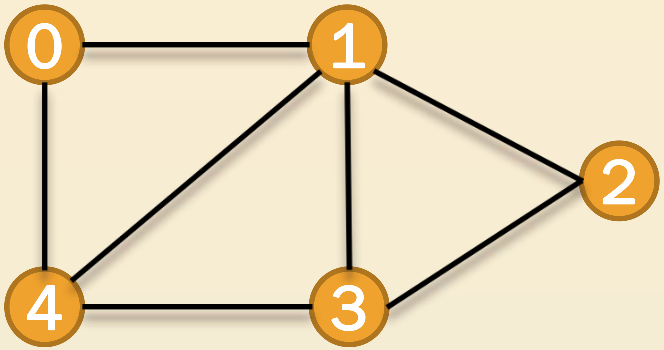
（3）如果G是带权无向图，则：

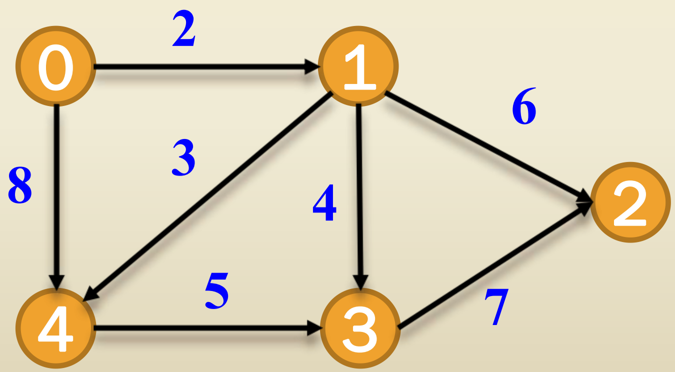
$A[i][j]=w_{ij}$	若 $i \neq j$ 且 $(i,j) \in E(G)$
$A[i][j]=0$	$i=j$
$A[i][j]=\infty$	其他

（4）如果G是带权有向图，则：

$A[i][j]=w_{ij}$	若 $i \neq j$ 且 $\langle i,j \rangle \in E(G)$
$A[i][j]=0$	$i=j$
$A[i][j]=\infty$	其他

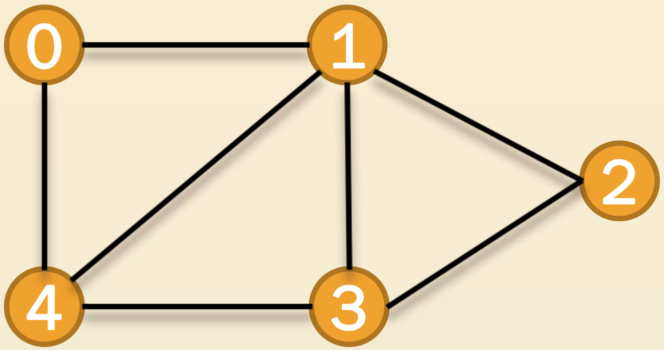
1. 邻接矩阵存储方法



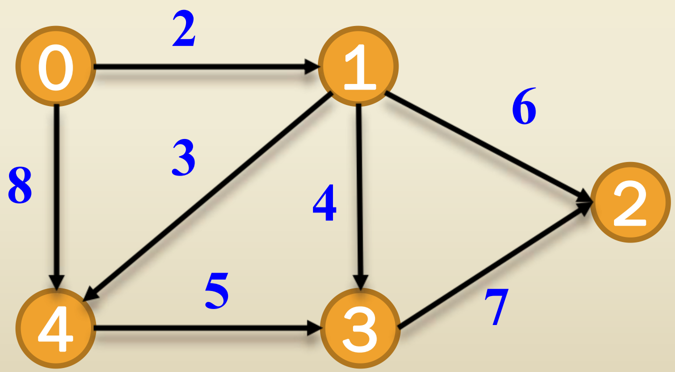


课堂练习：请计算上述两图对应的邻接矩阵。

1. 邻接矩阵存储方法



$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0
\end{array}
$$



$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 2 & \infty & \infty & 8 \\
1 & \infty & 0 & 6 & 4 & 3 \\
2 & \infty & \infty & 0 & \infty & \infty \\
3 & \infty & \infty & 7 & 0 & \infty \\
4 & \infty & \infty & \infty & 5 & 0
\end{array}
$$

邻接矩阵的类型定义如下：

```
#define MAXV <最大顶点个数>
typedef struct
{   int no;                        //顶点编号
    char data[MAXL];               //顶点其他信息
} VertexType;                      //顶点类型

typedef struct
{   int edges[MAXV][MAXV];         //邻接矩阵的边数组
    int n,e;                       //顶点数，边数
    VertexType vexs[MAXV];         //存放顶点信息
} MGraph;                          //完整的图邻接矩阵类型
```

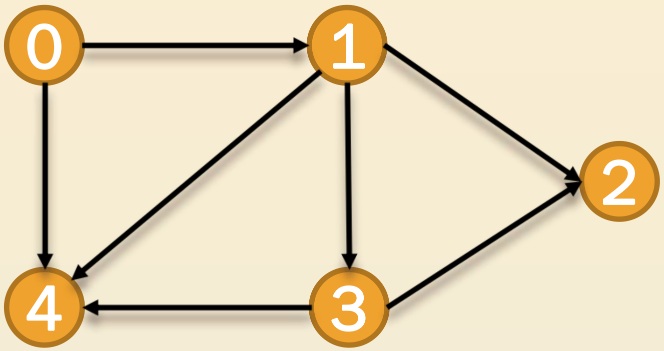
2．邻接表存储方法

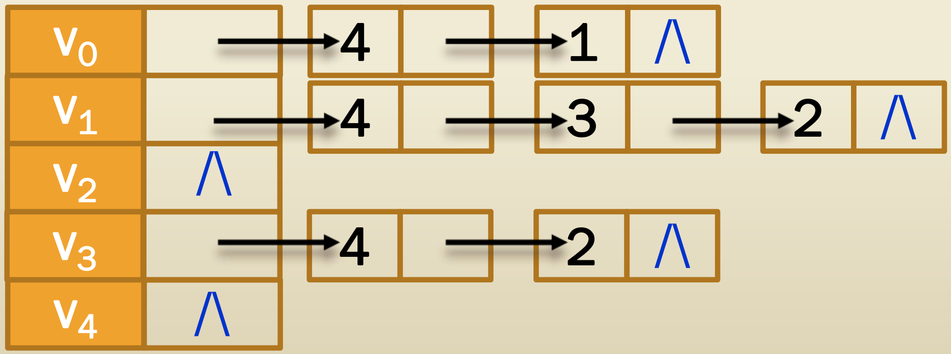
图的邻接表存储方法是一种链式存储结构。

图的每个顶点建立一个单链表，第i（$0 \le i \le n-1$）个单链表中的结点表示依附于顶点i的边。

每个单链表上附设一个表头结点，将所有表头结点构成一个表头结点数组。

2．邻接表存储方法





```c
typedef struct ANode
{   int adjvex;                        //该边的终点编号
    int weight;                        //该边的权值
    struct ANode *nextarc;             //指向下一条边的指针
} ArcNode;                             //边结点类型

typedef struct Vnode
{   char data[MAXL];                   //顶点其他信息
    ArcNode *firstarc;                 //指向第一条边
} VNode;                               //邻接表头结点类型

typedef VNode AdjList[MAXV];           //AdjList是邻接表类型

typedef struct
{   AdjList adjlist;                   //邻接表
    int n,e;                           //图中顶点数n和边数e
} ALGraph;
```

4.4.2 深度优先遍历

从给定图中任意指定的顶点（称为初始点）出发，按照某种搜索方法沿着图的边访问图中的所有顶点，使每个顶点仅被访问一次，这个过程称为图的遍历。

为了避免同一个顶点被重复访问，必须记住每个被访问过的顶点。为此，设置一个访问标志数组visited[]，当顶点i被访问过时，数组中元素visited[i]置为1；否则置为0。

深度优先搜索的过程是：

（1）从图中某个初始顶点*v*出发，首先访问初始顶点*v*。

（2）然后选择一个与顶点*v*相邻且没被访问过的顶点*w*为初始顶点，再从*w*出发进行深度优先搜索。

（3）重复直到图中与当前顶点*v*邻接的所有顶点都被访问过为止。显然，这个搜索过程是个递归过程。

以邻接矩阵为存储结构的深度优先搜索算法

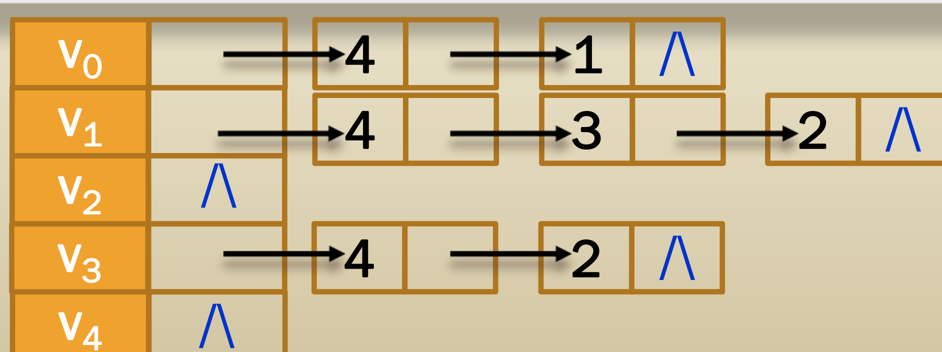
```
void DFS(MGraph g,int v)        //邻接矩阵的DFS算法
{  int w;
   printf("%3d",v);             //输出被访问顶点的编号
   visited[v]=1;                //置已访问标记

   for (w=0;w<g.n;w++)          //找顶点v的所有相邻点
     if (g.edges[v][w]!=0 && g.edges[v][w]!=INF
              && visited[w]==0)
        DFS(g,w);               //找顶点v的未访问过的相邻点w
}
```

以邻接表为存储结构的深度优先搜索算法：

```
void DFS(ALGraph *G,int v)     //邻接表的DFS算法
{   ArcNode *p;
    printf("%3d",v);           //输出被访问顶点的编号
    visited[v]=1;              //置已访问标记

    p=G->adjlist[v].firstarc;  //p指向顶点v的第一个邻接点
    while (p!=NULL)
    {   if (visited[p->adjvex]==0)  //若p->adjvex顶点未访问,递归访问它
            DFS(G,p->adjvex);
        p=p->nextarc;          //p指向顶点v的下一个邻接点
    }
}
```



【例4.6】假设图G采用邻接表存储，设计一个算法判断图G中从顶点*u*到*v*是否存在简单路径。

解：所谓简单路径是指路径上的顶点不重复。采用深度优先遍历的方法，从顶点*u*出发搜索到顶点*v*的过程如下：

DFS(G,u,v) → DFS(G,u1,v) → … → DFS(G,v,v)

```
bool ExistPath(ALGraph *G,int u,int v)
//判断G中从顶点u到v是否存在简单路径
{   int w; ArcNode *p;
    visited[u]=1;                //置已访问标记
    if (u==v)                    //找到了一条路径，返回true
       return true;

    p=G->adjlist[u].firstarc; //p指向顶点u的第一个相邻点
    while (p!=NULL)
    {    w=p->adjvex;            //w为顶点u的相邻顶点
         if (visited[w]==0)      //若w顶点未访问,递归访问它
         {   bool flag=ExistPath(G,w,v);
             if (flag) return true;
         }
         p=p->nextarc;           //p指向顶点u的下一个相邻点
    }
    return false;                //没有找到v，返回false
}
```

【例4.7】假设图G采用邻接表存储，设计一个算法输出图G中从顶点*u*到*v*的一条简单路径（假设图G中从顶点*u*到*v*至少有一条简单路径）。

解：采用深度优先遍历的方法，*f*(G，*u*，*v*，apath，path)搜索图G中从顶点*u*到*v*的一条简单路径path。

通过顶点*u*在图G中搜索，当*u*=*v*时说明找到一条从*u*到*v*的简单路径，将apath复制到path中并返回。否则继续深度优先遍历。

```
void FindaPath(ALGraph *G,int u,int v,vector<int> apath,
       vector<int> &path)
{  int w;
   ArcNode *p;
   visited[u]=1;
   apath.push_back(u);                        //顶点u加入到apath路径中
   if (u==v)                                  //找到一条路径
   {   path=apath;                            //将apath复制到path
       return;                                //返回
   }

   p=G->adjlist[u].firstarc;                  //p指向顶点u的第一个相邻点
   while (p!=NULL)
   {   w=p->adjvex;                           //相邻点的编号为w
       if (visited[w]==0)
          FindaPath(G,w,v,apath,path);
       p=p->nextarc;                          //p指向顶点u的下一个相邻点
   }
}
```

4.4.4 广度优先遍历

广度优先搜索的过程是：

（1）首先访问初始顶点v。

（2）接着访问顶点v的所有未被访问过的邻接点v_1，v_2，…，v_t。

（3）然后再按照v_1，v_2，…，v_t的次序，访问每一个顶点的所有未被访问过的邻接点，依次类推，直到图中所有和初始顶点v有路径相通的顶点都被访问过为止。

以邻接矩阵为图的存储结构，采用广度优先搜索图时，需要使用一个队列。

```
void BFS(MGraph g,int v)                    //邻接矩阵的BFS算法
{  queue<int> qu;                           //定义一个队列qu
   int visited[MAXV];                       //定义存放结点的访问标志的数组
   int w,i;
   memset(visited,0,sizeof(visited));       //访问标志数组初始化
   printf("%3d",v);                         //输出被访问顶点的编号
   visited[v]=1;                            //置已访问标记
   qu.push(v);                              //v进队
   while (!qu.empty())                      //队列不空时循环
   {  w=qu.front(); qu.pop();               //出队顶点w
      for (i=0;i<g.n;i++)                   //找与顶点w相邻的顶点
        if (g.edges[w][i]!=0 && g.edges[w][i]!=INF && visited[i]==0)
        {               //若当前邻接顶点i未被访问
           printf("%3d",i);                 //访问相邻顶点
           visited[i]=1;                    //置该顶点已被访问的标志
           qu.push(i);                      //该顶点进队
        }
   }
   printf("\n");
}
```

以邻接表为图的存储结构，采用广度优先搜索图时，需要使用一个队列。

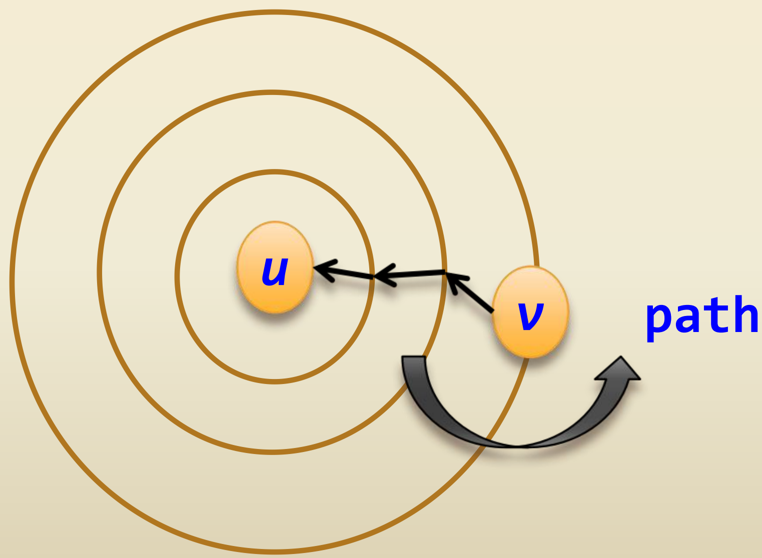
```
void BFS(ALGraph *G,int v)                    //邻接表的BFS算法
{  ArcNode *p;
   queue<int> qu;                             //定义一个队列qu
   int visited[MAXV],w;                       //定义存放顶点的访问标志的数组
   memset(visited,0,sizeof(visited));         //访问标志数组初始化
   printf("%3d",v);                           //输出被访问顶点的编号
   visited[v]=1;                              //置已访问标记
   qu.push(v);                                //v进队

   while (!qu.empty())                        //队列不空时循环
   {   w=qu.front(); qu.pop();                //出队顶点w
       p=G->adjlist[w].firstarc;              //找顶点w的第一个邻接点
       while (p!=NULL)
       {  if (visited[p->adjvex]==0)          //若当前邻接顶点未被访问
          {  printf("%3d",p->adjvex);         //访问相邻顶点
             visited[p->adjvex]=1;            //置该顶点已被访问的标志
             qu.push(p->adjvex);              //该顶点进队
          }
          p=p->nextarc;                       //找顶点w的下一个邻接点
       }
   }
}
```

【例4.8】假设图G采用邻接表存储，设计一个算法，求不带权无向连通图G中从顶点*u*到顶点*v*的一条最短路径。

解： **图G是不带权的无向连通图，一条边的长度计为1，因此，求顶点*u*和顶点*v*的最短路径即求距离顶点*u*到顶点*v*的边数最少的顶点序列。**

利用广度优先遍历算法，从*u*出发一层一层地向外扩展，扩展到某个顶点时记录其前驱顶点，当第一次找到顶点*v*时队列中便隐含从顶点*u*到顶点*v*最近的路径，再利用队列输出最短路径。



```
void ShortPath(ALGraph *G,int u,int v,vector<int> &path)
//求图G中从顶点u到顶点v的最短（逆）路径path
{  ArcNode *p; int w;
   queue<int> qu;                            //定义一个队列qu
   int pre[MAXV];                            //表示前驱关系
   int visited[MAXV];                        //定义存放顶点的访问标志的数组
   memset(visited,0,sizeof(visited));        //访问标志数组初始化
   qu.push(u);                               //顶点u进队
   visited[u]=1;
   pre[u]=-1;                                //起始顶点的前驱置为-1
   while (!qu.empty())                       //队不空时循环
   {  w=qu.front(); qu.pop();                //出队顶点w
      if (w==v)                              //找到v时输出路径之逆并退出
      {  Findpath(pre,v,path);
         return;
      }
      p=G->adjlist[w].firstarc;              //找w的第一个邻接点
      while (p!=NULL)
      {  if (visited[p->adjvex]==0)
         {  visited[p->adjvex]=1;            //访问w的邻接点
            qu.push(p->adjvex);              //将w的邻接点进队
            pre[p->adjvex]=w;                //设置p->adjvex顶点的前驱为w
         }
         p=p->nextarc;                       //找w的下一个邻接点
      }
   }
}
```