第3章 穷举法

【案例引入】

黑客经常采用穷举法破译用户口令

- 口令字符有 10 个数字、33 个标点符号、26 个大写字母、26 个小写字母，共 95 个。
- 如果采用 8 位口令，这样一次破译的概率是四十四万亿分之一。

本章学习目标：

（1）理解穷举法的定义和各种列举方法。

（2）掌握前缀和数组以及并查集在穷举算法中的优化方法。

（3）掌握采用穷举法求解回文问题、幂集、全排列、0/1 背包和旅行商问题等经典问题的算法设计方法。

（4）灵活运用穷举法解决复杂问题。

3.1 穷举法概述

3.1.1 什么是穷举法

穷举法又称枚举法或者列举法，其基本思想是先确定有哪些穷举对象和穷举对象的顺序，按穷举对象的顺序逐一列举每个穷举对象的所有情况，再根据问题提出的约束条件检验哪些是问题的解，哪些应予以排除。穷举法主要用于解决“是否存在”和“有多少可能性”等类型问题。采用穷举法设计的算法称为**穷举算法**。

穷举法的关键是如何列举所有的情况，如果遗漏了某些情况可能得不到正确的解。往往不同求解问题的列举方法也是不同的，归纳起来常用的列举方法如下。

(1) 顺序列举：顺序列举是指问题解范围内的各种情况很容易与自然数对应甚至就是自然数，可以按自然数的变化顺序去列举。这是一种最简单也是常用的列举方法。

(2) 组合列举：组合列举是指问题解表现为一些元素的组合，可以通过组合列举方式枚举所有的组合情况，通常情况下组合列举是无序的。例如在 n 个元素中选择 m 个满足条件的元素通常采用组合列举。

(3) 排列列举：排列列举是指问题解表现为一组元素的排列，可以通过排列列举方式枚举所有的排列情况，针对不同的问题有些排列列举是无序的，有些是有序的。例如在 n 个元素中选择某种满足条件的元素顺序通常采用排列列举。

穷举法的优点如下：

(1) 理论上讲，穷举法可以解决可计算领域中的各种问题，尤其是在计算机速度非常快的今天，穷举法的应用领域是非常广阔的。

(2) 在实际应用中，通常要解决的问题规模不大，用穷举法设计的算法运算速度是人们可以接受的，此时设计一个更高效率的算法不值得。

(3) 穷举法算法一般逻辑清晰，编写的程序简洁明了。

(4) 穷举法算法一般不需要特别证明算法的正确性。

(5) 穷举法可作为某类问题时间性能的底限，用来衡量同样问题的更高效率的算法。

穷举法的主要缺点是设计的大多数算法的效率都不高，主要适合规模比较小的问题的求解。为此在采用穷举法求解时应根据问题的具体情况分析归纳，寻找简化规律，精简穷举循环，优化穷举过程。

3.1.2 穷举算法的框架

穷举算法一般使用循环语句和选择语句实现，其中循环语句用于枚举穷举对象所有可能的情况，而选择语句判定当前的条件是否为所求的解。在枚举时对可能的情况不能遗漏，一般也不应重复。其基本流程如下：

(1) 根据问题的具体情况确定穷举变量(简单变量或数组)。

(2) 根据确定的范围设置穷举循环。

(3) 根据问题的具体要求确定解满足的约束条件。

(4) 设计穷举算法,编写程序并执行和调试,对执行结果进行分析与讨论。

这里以顺序列举为例,假设某个问题的枚举变量是 x 和 y,穷举次序是先 x 后 y,均为顺序列举方式,它们的取值范围(即值域)分别是 $x \in \{x_1, x_2, \cdots, x_n\}$,$y \in \{y_1, y_2, \cdots, y_m\}$,约束条件为 $p(x_i, y_j)$,对应的穷举算法基本框架如下:

```
void exhaustive(x,n,y,m) {                    //穷举算法框架
    for (int i=1;i<=n;i++) {                  //枚举 x 的所有可能的值
        for (int j=1;j<=m;j++) {              //枚举 y 的所有可能的值
            ...
            if (p(x[i],y[j]))                 //检测是否满足约束条件
                输出一个解;
            ...
        }
    }
}
```

从中看出,x 和 y 所有可能的搜索范围是笛卡儿积,即($[x_1, y_1]$,$[x_1, y_2]$,…,$[x_1, y_m]$,…,$[x_n, y_1]$,$[x_n, y_2]$,…,$[x_n, y_m]$),这样的搜索范围可以用一棵树表示,称为解空间树(或者解空间),其中每个结点对应一个状态$[x_i, y_i]$。解空间包含求解问题的所有解,求解过程就是在整个解空间中搜索满足约束条件 $p(x_i, y_j)$的解。

【例 3.1】 在象棋算式中,不同的棋子代表不同的数字,现有如图 3.1 所示的算式,设计一个算法求这些棋子各代表哪些数字。

扫一扫

视频讲解

兵炮马卒
+ 兵炮车卒
车卒马兵卒

图 3.1 象棋算式

解 采用穷举法求解,显然该问题属于顺序列举。设兵、炮、马、卒和车的枚举变量分别为 a、b、c、d、e,则 a、b、c、d、e 的取值范围均为 0～9 且均不相等,即!($a==b$ || $a==c$ || $a==d$ || $a==e$ || $b==c$ || $b==d$ || $b==e$ || $c==d$ || $c==e$ || $d==e$)成立。

用 m、n 和 s 分别表示象棋算式中 3 行对应的整数,则 $m=a\times1000+b\times100+c\times10+d$,$n=a\times1000+b\times100+e\times10+d$,$s=e\times10000+d\times1000+c\times100+a\times10+d$,根据题目的约束条件有 $m+n=s$ 成立。对应的穷举算法如下:

```
void chess() {                                //求象棋算式问题
    for (int a=1;a<=9;a++) {
        for (int b=0;b<=9;b++) {
            for (int c=0;c<=9;c++) {
                for (int d=0;d<=9;d++) {
                    for (int 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;     //避免重复
                        int m=a*1000+b*100+c*10+d;
                        int n=a*1000+b*100+e*10+d;
                        int 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);
                    }
                }
            }
        }
    }
}
```

执行上述算法的输出结果如下:

兵:5 炮:2 马:4 卒:0 车:1

一般地,穷举算法的执行时间可以用解空间中的状态总数×单个状态的检测代价(检测单个状态是否满足约束条件)表示。在例 3.1 的算法中有 5 个枚举变量,每个枚举变量的取值范围是 0～9,所有的可能情况或者说解空间树中的结点个数大约为 10^5,单个状态的检测代价为 $O(1)$,显然该算法是低效的。

尽管穷举算法的性能较差,但可以以它为基础进行优化从而得到较高性能的算法,优化点主要是减少状态总数(即减少枚举变量的个数和缩小枚举变量的值域)和降低单个状态的检测代价,具体来讲包含以下几个方面:

(1) 选择高效的数据结构。

(2) 减少重复计算。

(3) 将原问题转化为更小的问题。

(4) 根据问题的性质进行剪枝。

(5) 引进其他算法。

在穷举算法的优化中不同问题的优化点是不相同的,大家需要通过大量实训掌握一些基本的优化算法设计技巧。例如对于例 3.1,从象棋算式看出“卒+卒=卒”,这样卒只能取 0,从而减少了一个枚举变量,其他类推。

3.2 算法优化中常用的数据结构

从数据结构的角度看,选择合适、高效的数据结构可以优化穷举算法的性能,本节主要讨论前缀和数组与并查集。

3.2.1 前缀和数组

前缀和数组主要用于高效地求一个数组的全部子数组(由数组中的若干连续元素组成)的元素和。假设有一个数组 a,其前缀和数组用 psum 表示,其中 psum[i]($0\leqslant i\leqslant n$)为 a 中的前 i 个元素和,即 psum[i]=a[0]+…+a[$i-1$]。求 psum 数组的递推关系如下:

psum[i]=0　　当 $i=0$ 时

psum[i]=psum[$i-1$]+a[$i-1$]　　当 $i\geqslant 1$ 时

因此对于 $j\geqslant i$(即 $1\leqslant i\leqslant j\leqslant n$),有:

psum[i]=a[0]+…+a[$i-1$]

psum[$j+1$]=a[0]+…+a[$i-1$]+a[i]+…+a[j]

两式相减得到 psum[$j+1$]−psum[i]=a[i]+…+a[j],或者 a[i]+…+a[j]=psum[$j+1$]−psum[i]。

也就是说求出 psum 数组后,a[$i..j$]的元素和等于 psum[$j+1$]−psum[i]。显然 a[$i..j$]区间的下标满足 $0\leqslant i\leqslant j\leqslant n-1$,通过枚举 i 和 j 可以求出 a 中任意子数组元素和,用 ans 数组表示。对应的代码如下:

```
vector<vector<int>> subarr(vector<int> & a) {                    //求子数组和
    int n=a.size();
```

```
        vector<vector<int>> ans(n,vector<int>(n,0));
        int psum[n+1];
        psum[0]=0;                                    //求 psum
        for(int i=1;i<=n;i++)
            psum[i]=psum[i-1]+a[i-1];
        for(int i=0;i<n;i++) {
            for(int j=i;j<n;j++) {
                ans[i][j]=psum[j+1]-psum[i];//求 nums[i..j]元素之和
            }
        }
        return ans;
    }
```

subarr 算法分析：该算法的时间复杂度为 $O(n^2)$，如果不采用前缀和数组，对于每个 $a[i..j]$都采用一重循环求其元素和，对应的时间复杂度为 $O(n^3)$。

【例 3.2】 子数组之和(LintCode138★)。给定一个整数数组 nums，设计一个算法找到和为 0 的子数组，返回满足要求的子数组的起始位置和结束位置，测试数据保证至少有一个子数组的和为 0，如果有多个子数组的和为 0，返回其中任意一个子数组即可。例如，nums={−3,1,2,−3,4}，答案为{0,2}或{1,3}。要求设计如下成员函数：

```
vector<int> subarraySum(vector<int> &nums) { }
```

扫一扫

视频讲解

解法 1：用 vector<int>容器 ans 存放答案，即和为 0 的子数组的起始位置和结束位置。采用穷举法，用 i 和 j 枚举所有的 nums$[i..j]$，求出元素和 sum，若 sum=0，则置 ans=$\{i,j\}$，返回 ans。对应的代码如下：

```
class Solution {
public:
    vector<int> subarraySum(vector<int> &nums) {              //求子数组和
        vector<int> ans;
        int n=nums.size();
        for(int i=0;i<n;i++) {
            for(int j=i;j<n;j++) {
                int sum=0;
                for(int k=i;k<=j;k++)
                    sum+=nums[k];
                if(sum==0) {
                    ans.push_back(i);
                    ans.push_back(j);
                    return ans;
                }
            }
        }
        return {0,0};                                         //没有找到
    }
};
```

上述算法的时间复杂度为 $O(n^3)$，代码提交后超时。

解法 2：采用前缀和数组 psum 实现，设 psum$[i]$为 nums 中前 i 个元素(即 nums$[0..i-1]$)的和，在求出 psum 数组后，$a[i..j]$的元素和等于 psum$[j+1]$−psum$[i]$，通过枚举 i 和 j 求出元素和为 0 的 nums$[i..j]$即可。对应的代码如下：

```
class Solution {
public:
```

```
    vector<int> subarraySum(vector<int> &nums) {                    //求子数组和
        int n=nums.size();
        int psum[n+1];
        psum[0]=0;
        for(int i=1;i<=n;i++)
            psum[i]=psum[i-1]+nums[i-1];
        vector<int> ans;
        for(int i=0;i<n;i++) {
            for(int j=i;j<n;j++) {
                int sum=psum[j+1]-psum[i];                          //求 nums[i..j]元素之和
                if(sum==0) {
                    ans.push_back(i);
                    ans.push_back(j);
                    return ans;
                }
            }
        }
        return {0,0};
    }
};
```

上述算法的时间复杂度为 $O(n^2)$。程序提交时通过，执行用时为 904ms，内存消耗为 10.47MB。

扫一扫

视频讲解

解法 3：采用前缀和数组＋哈希表进一步提高时间性能。前缀和数组仍然用 psum 表示，但将 psum[i]改为 nums[0..i]共 $i+1$ 个元素和，即 psum[i]＝nums[0]＋…＋nums[$i-1$]＋nums[i]，这样求 psum 如下：

psum[0]＝nums[0]

psum[i]＝psum[$i-1$]＋nums[i]　当 $i>0$ 时

因此对于 $j\geqslant i$（即 $0\leqslant i\leqslant j<n$），有 psum[$i$]＝nums[0]＋…＋nums[$i$]，psum[$j$]＝nums[0]＋…＋nums[$i$]＋nums[$i+1$]＋…＋nums[$j$]。两式相减得到 psum[$j$]－psum[$i$]＝nums[$i+1$]＋…＋nums[$j$]，或者 nums[$i+1$]＋…＋nums[$j$]＝psum[$j$]－psum[$i$]。

说明：在前缀和数组 psum 中元素设置有两种，一是用 psum[i]表示前 $i+1$ 个元素和，二是用 psum[i]表示前 i 个元素和，两种方法的含义类似，用户可以根据需要选择，但要注意细节上的差异。

再设计一个哈希表 unordered_map<int,int>类型的容器 hmap，用于存放以每个 psum[i]为关键字的序号 i，如图 3.2 所示。

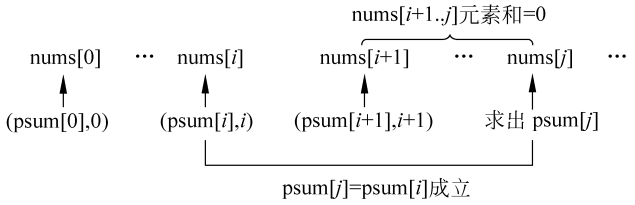

图 3.2　用 hmap 存放(psum[i],i)

用 j 遍历 nums，依次求出 nums[j]对应的前缀和 psum[j]，然后在 hmap 中查找 psum[j]，若找到了该关键字的元素，不妨假设 hmap 中找到的相同关键字元素是(psum[i],i)（即 hmap[psum[i]]＝i），显然 psum[j]－psum[i]＝nums[$i+1$]＋…＋nums[j]＝0，从而说明 nums[$i+1$..j]的元素和为 0，则(hmap[psum[j]]＋1,j)就是答案，否则说明在 hmap 中

没有找到 psum[j],则将(psum[j],j)插入 hmap 中。另外需要注意以下两点:

① 答案中起始位置为 0 是一种特殊情况,例如 nums={1,-1,2},求得 psum[0]=1(在 hmap 中插入(1,0)),psum[1]=0,此时应该找到一个和为 0 的子数组 nums[0..1],为了保证找到这样的答案,需要事先在 hmap 中插入元素(0,-1),即 hmap[0]=-1。

② 不必采用前缀和数组,将 psum 改为单个变量即可,也就是说 nums[j]对应的前缀和 psum 等于 psum+nums[j],在 hmap 中找到的相同关键字元素值是 hmap[psum],则(hmap[psum]+1,j)就是答案。

对应的代码如下:

```
class Solution {
public:
    vector<int> subarraySum(vector<int> &nums) {                    //求子数组和
        unordered_map<int,int> hmap;
        vector<int> ans;
        int n=nums.size();
        hmap[0]=-1;
        int psum=0;
        for (int j=0;j<n;j++) {
            psum+=nums[j];
            if (hmap.count(psum)>0) {
                ans.push_back(hmap[psum]+1);
                ans.push_back(j);
                return ans;
            }
            hmap[psum]=j;
        }
        return ans;
    }
};
```

由于哈希表查找和插入操作的时间复杂度接近 $O(1)$,上述算法的时间复杂度为 $O(n)$。程序提交时通过,执行用时为 806ms,内存消耗为 10.49MB。

3.2.2 并查集

扫一扫

视频讲解

1. 什么是并查集

给定 n 个结点的集合 U,结点编号为 1～n,再给定一个等价关系 R(满足自反性、对称性和传递性的关系称为等价关系,像图中顶点之间的连通性、亲戚关系等都是等价关系),由等价关系产生所有结点的一个划分,每个结点属于一个等价类,所有等价类是不相交的。

例如,U={1,2,3,4,5},R={<1,1>,<2,2>,<3,3>,<4,4>,<5,5>,<1,3>,<3,1>,<1,5>,<5,1>,<3,5>,<5,3>,<2,4>,<4,2>},从 R 看出它是一种等价关系,这样得到划分 U/R={{1,3,5},{2,4}},可以表示为$[1]_R=[3]_R=[5]_R$={1,3,5},$[2]_R=[4]_R$={2,4}。如果省略 R 中的<a,a>,用(a,b)表示对称关系,可以简化为 R={(1,3),(1,5),(2,4)}。

针对上述(U,R),现在的问题是求一个结点所属的等价类,以及合并两个等价类。该问题对应的基本运算如下。

① Init():初始化。

② Find(x):查找 x($x\in U$)结点所属的等价类。

③ Union(x,y):将 x 和 y($x\in U$,$y\in U$)所属的两个等价类合并。

上述数据结构就是并查集(因主要的运算为查找和合并而得名),所以并查集是支持一组互不相交的集合的数据结构。

2. 并查集的实现

并查集的实现方式有多种,这里采用树结构来实现。将并查集看成一个森林,每个等价类用一棵树表示,包含该等价类的所有结点即结点子集,称为子集树,每个子集树通过根结点标识,如图 3.3 所示的子集的根结点为 A 结点,称为以 A 为根的子集树。

并查集的基本存储结构(实际上是森林的双亲存储结构)如下:

```
int parent[MAXN];                    //并查集存储结构
int rnk[MAXN];                       //存储结点的秩(近似于高度)
```

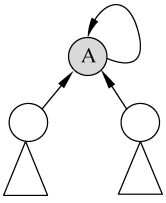

图 3.3 一个以 A 为根的子集树

其中,parent[i]=j 时,表示结点 i 的双亲结点是 j,初始时可以将每个结点看成一棵树,置 parent[i]=i(实际上置 parent[i]=−1 也是可以的,只是人们习惯采用前一种方式),当结点 i 是对应子树的根结点时,用 rnk[i]表示子树的高度,即秩,秩并不与高度完全相同,但它与高度成正比,初始化时置所有结点的秩为 0。

初始化算法如下(该算法的时间复杂度为 $O(n)$):

```
void Init() {                        //并查集初始化
    for (int i=1;i<=n;i++) {
        parent[i]=i;
        rnk[i]=0;
    }
}
```

所谓查找就是查找 x 结点所属子集树的根结点(根结点 y 满足条件 parent[y]=y),这是通过 parent[x]向上找双亲实现的,显然树的高度越小查找性能越好。为此在查找过程中进行路径压缩(即在查找过程中把查找路径上的结点逐一指向根结点),如图 3.4 所示,查找 x 结点根结点为 A,查找路径是 x→C→B→A,找到根结点 A 后,将路径上所有结点的双亲置为 A 结点,这样以后再查找 x、B 或者 C 结点的根结点时效率更高。

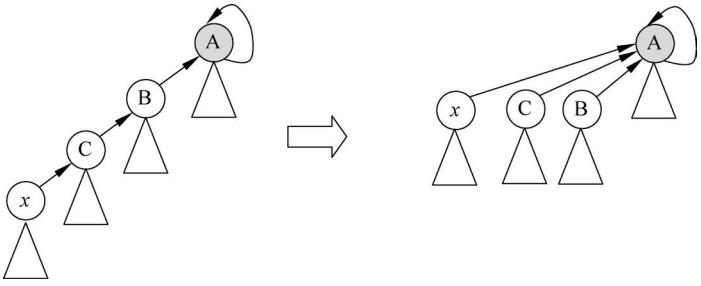

图 3.4 在查找中进行路径压缩

那么为什么不直接将一棵树中所有子结点的双亲都置为根结点呢?这是因为还有合并运算,合并运算可能破坏这种结构。

查找运算的递归算法如下:

```
int Find(int x) {                    //递归算法:在并查集中查找 x 结点的根结点
    if (x!=parent[x])
        parent[x]=Find(parent[x]); //路径压缩
    return parent[x];
}
```

查找运算的非递归算法如下：

```
int Find(int x) {                          //非递归算法：在并查集中查找 x 结点的根结点
    int rx=x;
    while (parent[rx]!=rx)                 //找到 x 的根 rx
        rx=parent[rx];
    int y=x;
    while (y!=rx) {                        //路径压缩
        int tmp=parent[y];
        parent[y]=rx;
        y=tmp;
    }
    return rx;                             //返回根
}
```

对于查找运算可以证明，若使用了路径压缩的优化方法，其平均时间复杂度为 Ackerman 函数的反函数，而 Ackerman 函数的反函数是一个增长速度极为缓慢的函数，在实际应用中可以粗略地认为是一个常量，也就是说查找运算的时间复杂度可以看成 $O(1)$。

所谓合并，就是给定一个等价关系(x,y)后，需要将 x 和 y 所属的两棵子集树合并为一棵树。首先查找 x 和 y 所属子集树的根结点 rx 和 ry，若 rx=ry，说明它们属于同一棵子集树，不需要合并，否则需要合并。注意合并是根结点 rx 和 ry 的合并，并且希望合并后的树的高度尽可能小，基本手段是将较小高度的根结点作为较大高度的根结点的孩子，具体合并过程如下：

① 若 rnk[rx]<rnk[ry]，将高度较小的 rx 结点作为 ry 的孩子结点，ry 树的高度不变。

② 若 rnk[rx]>rnk[ry]，将高度较小的 ry 结点作为 rx 的孩子结点，rx 树的高度不变。

③ 若 rnk[rx]=rnk[ry]，将 rx 结点作为 ry 的孩子结点或者将 ry 结点作为 rx 的孩子结点均可，但此时合并后的树的高度增 1。

对应的合并算法如下：

```
void Union(int x,int y) {                  //并查集中 x 和 y 的两个集合的合并
    int rx=Find(x);
    int ry=Find(y);
    if (rx==ry)return;                     //x 和 y 属于同一棵树的情况
    if (rnk[rx]< rnk[ry])
        parent[rx]=ry;                     //rx 结点作为 ry 的孩子
    else {
        if (rnk[rx]==rnk[ry])              //秩相同，合并后 rx 的秩增 1
            rnk[rx]++;
        parent[ry]=rx;                     //ry 结点作为 rx 的孩子
    }
}
```

合并运算的时间主要花费在查找上，查找运算的时间复杂度为 $O(1)$，则合并运算的时间复杂度也是 $O(1)$。

例如，$n=5$，执行并查集操作如下：

(1) 执行 Init()，构造 5 棵子集树，每棵树只有一个结点，结果如图 3.5(a)所示。

(2) 执行 Union(1,2)，将 1 和 2 所属的子集树合并，假设结点 1 作为合并树的根结点，其秩增 1，结果如图 3.5(b)所示。

(3) 执行 Union(2,3)，将根结点 3 作为根结点 1 的孩子，结果如图 3.5(c)所示。

(4) 执行 Union(4,5)，将根结点 5 作为根结点 4 的孩子，根结点 4 的秩增 1，结果如图 3.5(d)所示。

(5) 执行 Union(4,1)，假设将根结点 4 作为根结点 1 的孩子，根结点 1 的秩增 1，结果

如图 3.5(e)所示。

(6) 执行 Find(5),将结点 5 作为根结点 1 的孩子,结果如图 3.5(f)所示。

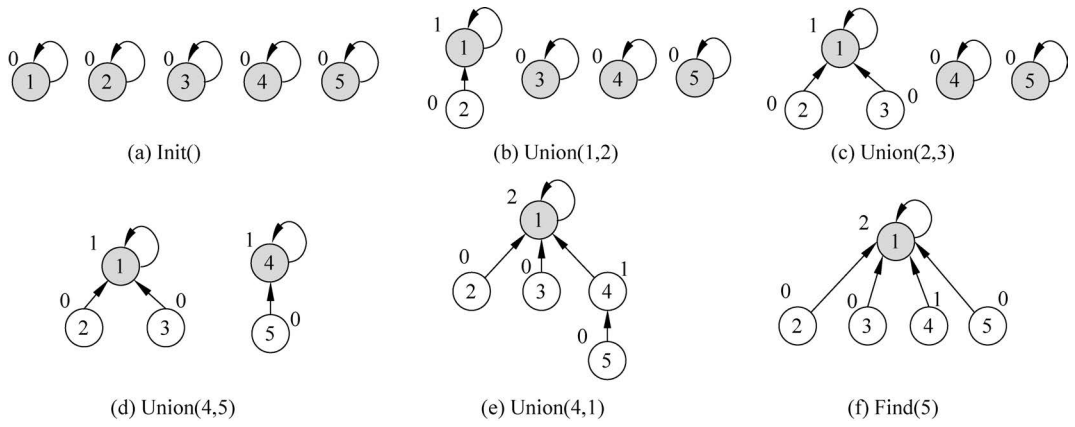

图 3.5 并查集操作及其结果

扫一扫 视频讲解

【例 3.3】 寻找朋友圈数(LintCode1857★★)。班上有 $n(1\leqslant n\leqslant 200)$名学生,其中有些人是朋友,有些则不是。学生的友谊具有传递性,如果已知 A 是 B 的朋友,B 是 C 的朋友,那么 A 也是 C 的朋友。朋友圈是指所有朋友的集合。给定一个 $n\times n$ 的对称矩阵 $\boldsymbol{M}$,表示班级中学生之间的朋友关系,如果 $M[i][j]=1$,表示已知学生 i 和学生 j 互为朋友关系,否则为不知道。设计一个算法求所有学生中已知的朋友圈总数。例如,$n=3$,$M=\{\{1,1,0\},\{1,1,0\},\{0,0,1\}\}$,答案是 2,两个朋友圈是{0,1}和{2}。要求设计如下成员函数:

```
int findCircleNum(vector<vector<int>>&M){}
```

扫一扫 源程序

解法 1:n 名学生的编号是 0～$n-1$。采用基本穷举法,用 fno 数组表示顶点所在的连通分量编号,首先将 n 个顶点看成 n 个连通分量,初始化顶点 i 的连通分量编号为 i。遍历 M,若 $M[i][j]=1$,在顶点 i 和顶点 j 之间连一条边,同时将顶点 j 所在连通分量的全部顶点的连通分量编号改为 fno[i]。遍历完毕,fno 中不同连通分量编号的个数即为朋友圈总数。

扫一扫 源程序

解法 2:同一个朋友圈的所有学生具有朋友关系,依题意朋友关系是一种等价关系,题目就是求按朋友关系划分后得到的等价类的个数,为此采用并查集,每棵子集树表示一个朋友圈,遍历 M,若 $M[i][j]=1$,将顶点 i 和顶点 j 所在的子集树合并。遍历完毕,每棵子集树的根结点 i 满足 parent[i]=i,所求根结点的个数即为朋友圈总数。

3.3 求回文串的个数

扫一扫 视频讲解

问题描述:回文子串(LintCode1856★)。小明喜欢玩文字游戏,今天他希望在一个字符串的子串中找到回文串。回文串是从左往右读和从右往左读相同的字符串,例如"121"和"tacocat"是回文串。子串是一个字符串中任意几个连续的字符构成的字符串。现在给定一个字符串 s,设计一个算法求出 s 的回文串个数。例如,s="mokkori",它的一些子串是"m"、"o"、"k"、"r"、"i"、"mo"、"ok"、"mok"、"okk"、"kk"和"okko",其中"m"、"o"、"k"、"r"、

"i"、"kk"和"okko"是回文子串，共有 7 个不同的回文子串，答案是 7。要求设计如下成员函数：

```
int countSubstrings(string &s) { }
```

解法 1：这里是求 s 中不重复的回文子串的个数，由于 s 中存在重复的字符，所以可能存在两个位置不同的相同子串。为了达到子串除重的目的，设计一个 set < string >容器 myset。采用穷举法枚举所有 s[$i..j$](0≤i≤j≤n－1)，若 s[$i..j$]是回文，将其插入 myset 中，最后返回 myset 的大小。对应的算法如下：

```
class Solution {
public:
    int countSubstrings(string &s) {
        int n=s.size();
        set<string> myset;
        for(int i=0;i<n;i++) {
            for(int j=i;j<n;j++) {
                if(ispal(s,i,j))
                    myset.insert(s.substr(i,j-i+1));
            }
        }
        return myset.size();
    }
    bool ispal(string &s,int low,int high) {                    //判断 s[i..j]是否为回文
        int i=low,j=high;
        while(i<j) {
            if(s[i]!=s[j]) return false;
            i++; j--;
        }
        return true;
    }
};
```

上述程序提交时通过，执行用时为 507ms，内存消耗为 2.07MB。

解法 2：上述算法在回文判断中存在冗余，例如 s="abba"，其中"bb"和"abba"的回文判断是独立的，实际上在确定"bb"为回文后，左、右两边均为'a'，则可以得出"abba"也是回文，以此作为优化点提高算法的性能。

对于长度为 n 的字符串 s，显然每个字符的位置可能是回文子串的中心点(共 n 个，这样的回文长度为奇数)，称为中心点 1，如图 3.6(a)所示，对应的位置 c 取值为 0～n－1，从 s[c..c]开始，若 s[l]=s[r]，则 s[l..r]为回文，执行 l－－和 r＋＋继续向两边扩展。例如，s="abba"，这样的中心点 c 可能是 0～3。

(1) c=0，s[0]=s[0]，则"a"是回文。

(2) c=1，s[1]=s[1]，则"b"是回文。后面 s[0]≠s[2]。

(3) c=2，s[2]=s[2]，则"b"是回文。后面 s[1]≠s[3]。

(4) c=3，s[3]=s[3]，则"a"是回文。

另外，每两个字符中间的位置可能是回文子串的中心点(共 n－1 个，这样的回文长度为偶数)，称为中心点 2，如图 3.6(b)所示，对应的位置 c 取值为 0～n－2，从 s[$c..c$＋1]开始，同样若 s[l]=s[r]，则 s[l..r]为回文，执行 l－－和 r＋＋继续向两边扩展。例如，s="abba"，这样的中心点 c 可能是 0～2。

(1) c=0，s[0]≠s[1]。

(2) $c=1$,$s[1]=s[2]$,则"bb"是回文。$s[0]=s[3]$,则"abba"是回文。

(3) $c=2$,$s[2]\neq s[3]$。

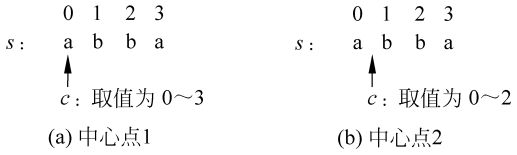

图 3.6 两种类型的中心点

这样共枚举 $2n-1$ 个中心点,将求出的回文子串插入 myset 中,最后返回 myset 的大小。对应的算法如下:

```
class Solution {
    int n;
    set<string> myset;
public:
    int countSubstrings(string &s) {
        n=s.size();
        for(int c=0;c<n;c++)                    //考虑每个字符的位置为回文的中心点
            cnt(s,c,c);
        for(int c=0;c<n-1;c++)                  //考虑每两个字符的中间位置为回文的中心点
            cnt(s,c,c+1);
        return myset.size();
    }
    void cnt(string &s,int l,int r) {           //求回文子串
        while (l>=0 && r<n && s[l]==s[r]) {
            myset.insert(s.substr(l,r-l+1));
            l--; r++;
        }
    }
};
```

上述程序提交时通过,执行用时为 122ms,内存消耗为 5.45MB。大家通过对比可以看出优化后的算法在时间性能上得到大幅度提高。

3.4 求最大连续子序列和

扫一扫

视频讲解

问题描述:给定一个含 $n(n\geqslant 1)$ 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)的最大连续子序列和为 20,而序列(-6,2,4,-7,5,3,2,-1,6,-9,10,-2)的最大连续子序列和为 16。规定一个序列的最大连续子序列和至少是 0,如果小于 0,其结果为 0(或者说最大连续子序列可以为空)。

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为 $a[i..j](i\leqslant j,0\leqslant i\leqslant n-1,i\leqslant j\leqslant n-1)$,求出它的所有元素之和 cursum,并通过比较将最大值存放在 maxsum 中,最后返回 maxsum。这种解法是通过穷举所有连续子序列(一个连续子序列由起始下标 i 和终止下标 j 确定)得到,采用的是典型的穷举法思想。

例如,对于 $a[0..5]=\{-2,11,-4,13,-5,-2\}$,求出的 $a[i..j](0\leqslant i\leqslant j\leqslant 5)$ 所有元素和如图 3.7 所示(行号为 i,列号为 j),其过程如下:

(1) $i=0$,依次求出 $j=0$、1、2、3、4、5 的子序列和分别为-2、9、5、18、13、11。

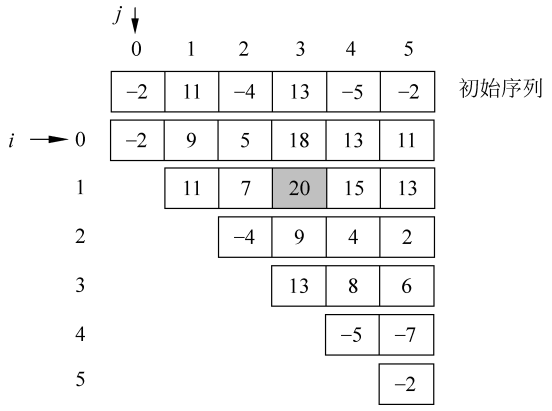

图 3.7 所有 $a[i..j]$子序列($0 \leqslant i \leqslant j \leqslant 5$)的元素和

(2) $i=1$,依次求出 $j=1$、2、3、4、5 的子序列和分别为 11、7、20、15、13。

(3) $i=2$,依次求出 $j=2$、3、4、5 的子序列和分别为 −4、9、4、2。

(4) $i=3$,依次求出 $j=3$、4、5 的子序列和分别为 13、8、6。

(5) $i=4$,依次求出 $j=4$、5 的子序列和分别为 −5、−7。

(6) $i=5$,求出 $j=5$ 的子序列和为 −2。

其中 20 是最大值,即最大连续子序列和为 20。对应的算法如下:

```
int maxsubsum1(int a[],int n) {                    //解法 1
    int maxsum=0,cursum;
    for (int i=0;i<n;i++){                         //通过两重循环穷举所有的连续子序列
        for (int j=i;j<n;j++) {
            cursum=0;
            for (int k=i;k<=j;k++)
                cursum+=a[k];
            maxsum=max(maxsum,cursum);             //通过比较求最大 maxsum
        }
    }
    return maxsum;
}
```

maxsubsum1 算法分析:在该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:采用前缀和方法,用 psum[i]表示子序列 $a[0..i-1]$元素和,即 a 中前 i 个元素和,显然有如下递推关系。

psum[0]=0

psum[i]=psum[$i-1$]+$a[i-1]$　当 $i>0$ 时

假设 $j \geqslant i$,则有:

psum[i]=$a[0]+a[1]+\cdots+a[i-1]$

psum[j]=$a[0]+a[1]+\cdots+a[i-1]+a[i]+\cdots+a[j-1]$

两式相减得到 psum[j]−psum[i]=$a[i]+\cdots+a[j-1]$,这样 i 从 0 到 $n-1$、$j-1$ 从 i 到 $n-1$(即 j 从 $i+1$ 到 n)循环可以求出所有连续子序列 $a[i..j]$之和,通过比较求出最大 maxsum 即可。对应的算法如下:

```
int maxsubsum2(int a[],int n) {                    //解法 2
    int psum[n+1];
```

```
    psum[0]=0;
    for(int i=1;i<=n;i++)
        psum[i]=psum[i-1]+a[i-1];
    int maxsum=0,cursum;
    for (int i=0;i<n;i++) {
        for (int j=i+1;j<=n;j++) {
            cursum=psum[j]-psum[i];
            maxsum=max(maxsum,cursum);        //通过比较求最大 maxsum
        }
    }
    return maxsum;
}
```

maxsubsum2 算法分析：在该算法中主要用了两重循环，所以有

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

解法 3：对前面的解法 1 进行优化。当 i 取某个起始下标时，依次求 $j=i$、$i+1$、…、$n-1$ 对应的子序列和，实际上这些子序列是相关的。用 Sum($a[i..j]$)表示子序列 $a[i..j]$ 元素和(即表示以 $a[i]$为起始元素的前缀和)，显然有如下递推关系：

Sum($a[i..j]$)=0　　当 $j<i$ 时

Sum($a[i..j]$)=Sum($a[i..j-1]$)+$a[j]$　　当 $j\geqslant i$ 时

这样在连续求 $a[i..j]$子序列和($j=i$、$i+1$、…、$n-1$)时没有必要使用循环变量为 k 的第三重循环，优化后的算法如下：

```
int maxsubsum3(int a[],int n) {                    //解法 3
    int maxsum=0,cursum;
    for (int i=0;i<n;i++) {
        cursum=0;
        for (int j=i;j<n;j++) {
            cursum+=a[j];
            maxsum=max(maxsum,cursum);    //通过比较求最大 maxsum
        }
    }
    return maxsum;
}
```

maxsubsum3 算法分析：在该算法中只有两重循环，所以有

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

解法 4：对前面的解法 2 继续优化。将 maxsum 和 cursum 初始化为 0，用 i 遍历 a，置 cursum+=$a[i]$，也就是说 cursum 累计到 $a[i]$时的元素和，分为以下两种情况。

① 若 cursum≥maxsum，说明 cursum 是一个更大的连续子序列和，将其存放在 maxsum 中，即置 maxsum=cursum。

② 若 cursum<0，说明 cursum 不可能是一个更大的连续子序列和，从下一个 i 开始继续遍历，所以置 cursum=0。

在上述过程中先置 cursum+=$a[i]$，后判断 cursum 的两种情况。在 a 遍历完后返回 maxsum 即可。对应的算法如下：

```
int maxsubsum4(int a[],int n) {                    //解法 4
    int maxsum=0,cursum=0;
    for (int i=0;i<n;i++) {
```

```
        cursum+=a[i];
        if(cursum>=maxsum)              //通过比较求最大 maxsum
            maxsum=cursum;
        if(cursum<0)                    //若 cursum<0,最大连续子序列从下一个位置开始
            cursum=0;
    }
    return max(maxsum,0);
}
```

maxsubsum4 算法分析：在该算法中只有一重循环，所以时间复杂度为 $O(n)$。

可以看出，尽管仍采用穷举法思路，但可以通过各种优化手段降低算法的时间复杂度。解法 2 的优化点是采用前缀和数组，解法 3 的优化点是找出 $a[i..j-1]$和 $a[i..j]$子序列的相关性，解法 4 的优化点是进一步判断 cursum 的两种情况。

思考题：对于给定的整数序列 a，不仅要求出其中最大连续子序列的和，还需要求出这个具有最大连续子序列和的子序列(给出其起始下标和终止下标)，如果有多个具有最大连续子序列和的子序列，求其中任意一个子序列。

【例 3.4】 求最大子序列和(LeetCode53★)。给定一个含 $n(1\leqslant n\leqslant 10^5)$个整数的数组 nums，设计一个算法找到一个具有最大和的连续子数组(子数组中最少包含一个元素)，返回其最大和。例如，nums={−2,1,−3,4,−1,2,1,−5,4}，答案为 6，对应的连续子数组是{4,−1,2,1}。要求设计如下成员函数：

扫一扫

视频讲解

```
int maxSubArray(vector<int>&nums){ }
```

解 本例是求 nums 的最大连续子序列和，这里最大连续子序列至少包含一个元素，所以最大连续子序列和可能为负数。例如 nums={−1,−2}，答案为−1，因此不能简单地采用 3.4 节中解法 4 的优化点，需要做以下两点修改：

(1) 将表示结果的 maxsum 初始化为 nums[0]而不是 0。

(2) 在求出 maxsum 后直接返回该值，而不是 max(maxsum,0)。

对应的算法如下：

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n=nums.size();
        if(n==1) return nums[0];
        int maxsum=nums[0],cursum=0;
        for (int i=0;i<n;i++) {
            cursum+=nums[i];
            if(cursum>=maxsum)              //求 maxsum
                maxsum=cursum;
            if(cursum<0)                    //若 cursum<0,重置其为 0
                cursum=0;
        }
        return maxsum;
    }
};
```

上述程序提交时通过，运行时间为 92ms，内存消耗为 66.1MB。

3.5 求 幂 集

扫一扫

视频讲解

穷举算法中的组合列举通常与求幂集相关，本节介绍求幂集的算法设计。

问题描述：对于给定的正整数 $n(n\geqslant 1)$，求由 $1\sim n$ 构成的集合的幂集 ps。例如，$n=3$ 时 ps=$\{\{\},\{1\},\{2\},\{1,2\},\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$(子集的顺序可以任意)。

解法 1：采用二进制表示法求 $1\sim n$ 集合的幂集。所谓 $1\sim n$ 集合的幂集就是由 $1\sim n$ 集合中所有子集构成的集合，包括全集和空集。例如 $n=3$ 时所有子集对应的二进制数/十进制数如表 3.1 所示，从中看出，$1\sim n$ 集合的幂集恰好与 2^n 个二进制数(转换为 $0\sim 2^n-1$ 的十进制数)相对应，每个二进制数对应唯一的子集。二进制数恰好 n 位，从低位到高位编号为 $0\sim n-1$，二进制位和 $1\sim n$ 集合元素的对应关系如图 3.8 所示，若二进制位 j 是 1，则对应的子集中包含整数 $j+1$，否则不包含整数 $j+1$。

表 3.1 所有子集对应的二进制数/十进制数

子 集	对应的二进制数	对应的十进制数
{}	000	0
{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

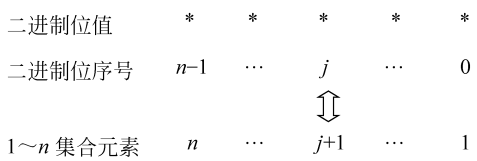

图 3.8 二进制位和 $1\sim n$ 集合元素的对应关系

用一个 vector<int>容器 e 表示一个子集，用 vector<vector<int>>容器 ps 存放幂集(即子集的集合)。算法的过程是用 i 枚举 $0\sim 2^n-1$ 的十进制数，每个 i 产生一个子集，由 i 产生子集 e 的过程是用 j 采用位运算符枚举 i 的二进制位，若 i 中编号为 j 的位是 1，则将 $j+1$ 添加到 e 中，j 枚举结束后将 e 添加到 ps 中。i 枚举结束后返回 ps。对应的算法如下：

```
vector<vector<int>> pset1(int n) {                    //求幂集算法 1
    vector<vector<int>> ps;                           //存放幂集
    for(int i=0;i<(1<<n);i++) {                       //执行 2^n 次
        vector<int> e;
        for(int j=0;j<n;j++) {
            if(i&(1<<j))                              //i 的二进制位 j 的值是 1
                e.push_back(j+1);                     //选取整数 j+1
        }
        ps.push_back(e);                              //将子集 e 添加到 ps 中
```

```
    }
    return ps;
}
```

pset1 算法分析：在该算法中外层 for 循环执行 2^n 次，内层 for 循环执行 n 次，所以上述算法的时间复杂度为 $O(n\times 2^n)$，属于指数级的算法。

说明：在 pset1 算法中用一个十进制数表示一个子集，当 n 较大时会发生溢出。

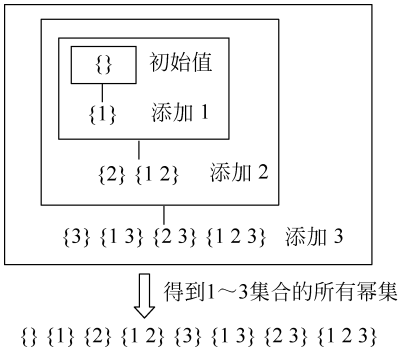

图 3.9 求 1～3 集合的幂集的过程

解法 2：采用增量穷举法求 1～n 集合的幂集，当 $n=3$ 时的求解如图 3.9 所示，其过程如下：

(1) 产生一个空集合元素{}添加到 ps 中，即 ps={{}}。

(2) 在(1)得到的 ps 的每一个集合元素的末尾添加 1 构成新集合元素{1}，将其添加到 ps 中，即 ps={{},{1}}。

(3) 在(2)得到的 ps 的每一个集合元素的末尾添加 2 构成新集合元素{2}、{1,2}，将其添加到 ps 中，即 ps={{},{1},{2},{1,2}}。

(4) 在(3)得到的 ps 的每一个集合元素的末尾添加 3 构成新集合元素{3}、{1,3}、{2,3}、{1,2,3}，将其添加到 ps 中，即 ps={{},{1},{2},{1,2},{3},{1,3},{2,3},{1,2,3}}。

最后的 ps 构成{1,2,3}的幂集，返回 ps 即可。

采用迭代方法，假设前面求出 1～$i-1$ 集合的幂集为 ps，先置 ps1 为 ps，在 ps1 的每个集合元素中添加 i，再将 ps1 的全部集合元素添加到 ps 中，这样得到 1～i 集合的幂集 ps，以此类推，直到 $i=n$ 为止。对应的迭代算法如下：

```
vector<vector<int>> pset2(int n) {                    //求幂集算法 2
    vector<vector<int>> ps;                           //存放幂集
    vector<int> e;
    ps.push_back(e);                                  //添加空集合元素{}
    for (int i=1;i<=n;i++) {                          //循环添加 1～n
        vector<vector<int>> A=ps;                     //A 存放上一步得到的幂集
        for (auto it=A.begin();it!=A.end();++it)
            (*it).push_back(i);                       //在 A 的每个集合元素的末尾添加 i
        for (auto it=A.begin();it!=A.end();++it)
            ps.push_back(*it);                        //将 A 的每个集合元素添加到 ps 中
    }
    return ps;
}
```

实际上可以不用 A，当前面求出 1～$i-1$ 集合的幂集 ps 后，置 m=ps.size()，在 ps[0..$m-1$]的每个集合元素中添加 i 并添加到 ps 中，同样得到 1～i 集合的幂集 ps，以此类推，直到 $i=n$ 为止。对应的迭代算法如下：

```
vector<vector<int>> pset3(int n) {                    //求幂集算法 3
    vector<vector<int>> ps;                           //存放幂集
    vector<int> e;
    ps.push_back(e);                                  //添加空集合元素{}
    for (int i=1;i<=n;i++) {                          //循环添加 1～n
        int m=ps.size();
        for(int j=0;j<m;j++) {
```

```
            vector<int> e=ps[j];                    //取出 e=ps[j]
            e.push_back(i);                         //在 e 的末尾添加 i
            ps.push_back(e);                        //再将 e 添加到 ps 中
        }
    }
    return ps;
}
```

上述迭代算法很容易采用递归算法实现，对应的递归算法如下：

```
vector<vector<int>> pset4(int n) {                  //求幂集算法 4:递归算法
    if(n==0) return {{}};
    vector<vector<int>> ps;                         //存放幂集
    ps=pset4(n-1);                                  //求出 1~n-1 的幂集 ps
    int m=ps.size();
    for(int j=0;j<m;j++) {
        vector<int> e=ps[j];                        //取出 e=ps[j]
        e.push_back(n);                             //在 e 的末尾添加 n
        ps.push_back(e);                            //再将 e 添加到 ps 中
    }
    return ps;
}
```

pset2～pset4 算法分析：这 3 个算法的过程类似，时间复杂度相同。以 pset2 为例，外层 for 循环 i 从 1 到 n，对于 i 循环，ps1 中存放 $1\sim i-1$ 集合的幂集，共 2^{i-1} 个集合元素，两个内层 for 循环每个执行 2^{i-1} 次，合起来执行 2^i 次，所以 $T(n)=\sum_{i=1}^{n}2^i=O(2^n)$。

扫一扫

视频讲解

【例 3.5】 子集(LintCode17★★)。给定一个含不同整数的集合 nums，设计一个算法求其所有的子集，要求子集中的元素不能以降序排列，解集中不能包含重复的子集，但全部的子集可以按任意顺序输出。例如，nums={2,3,1}，答案是{{3},{1},{2},{1,2,3},{1,3},{2,3},{1,2},{}}。要求设计如下成员函数：

```
vector<vector<int>> subsets(vector<int> &nums) { }
```

解法 1：算法思路与前面求幂集的解法 1 相同，仅将求 $1\sim n$ 集合的幂集改为求 nums[0..$n-1$]集合的幂集，另外由于要求子集中的元素不能以降序排列，所以先对 nums 递增排序。排序后求幂集的过程是用 i 枚举 $0\sim 2^n-1$ 的十进制数，每个 i 产生一个子集，由 i 产生子集 e 的过程是用 j 采用位运算符枚举 i 的二进制位，若 i 中编号为 j 的位是 1，则将 nums[j]添加到 e 中，j 枚举结束后将 e 添加到 ps 中。i 枚举结束后返回 ps。对应的代码如下：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int> &nums) {
        sort(nums.begin(),nums.end());
        vector<vector<int>> ps;                     //存放幂集
        int n=nums.size();
        for(int i=0;i<(1<<n);i++) {                 //执行 2^n 次
            vector<int> e;
            for(int j=0;j<n;j++) {
                if(i&(1<<j)) e.push_back(nums[j]);
            }
            ps.push_back(e);
```

```
        }
        return ps;
    }
};
```

上述程序提交时通过，执行用时为 41ms，内存消耗为 3.82MB。

解法 2：算法思路与前面求幂集的解法 2 相同，先对 nums 递增排序，初始化 ps={{}}，i 从 0 到 $n-1$ 循环，即置 A=ps，在 A 的每个集合元素中添加 nums[i]，再将 A 的元素添加到 ps 中。对应的代码如下：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int> &nums) {
        int n=nums.size();
        if(n==0) return {{}};
        sort(nums.begin(),nums.end());
        vector<vector<int>> ps;                    //存放幂集
        vector<int> e;
        ps.push_back(e);                           //添加空集合元素{}
        for (int i=0;i<n;i++) {                    //循环添加 nums[0..n-1]
            vector<vector<int>> A=ps;              //A 存放上一步得到的幂集
            for (auto it=A.begin();it!=A.end();++it)
                (*it).push_back(nums[i]);          //在 A 的每个集合元素的末尾添加 nums[i]
            for (auto it=A.begin();it!=A.end();++it)
                ps.push_back(*it);                 //将 A 的每个集合元素添加到 ps 中
        }
        return ps;
    }
};
```

上述程序提交时通过，执行用时为 41ms，内存消耗为 3.89MB。

3.6 0/1 背包问题

问题描述：问题描述见 2.4 节。

解 0/1 背包问题的物品选择情况用解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$ 表示，$x_i=1$ 表示选择物品 i，$x_i=0$ 表示不选择物品 i。其形式化描述如下。

约束条件：

$$\sum_{i=0}^{n-1} x_i \times w_i \leqslant W$$

目标函数：

$$\mathrm{MAX}\left\{\sum_{i=0}^{n-1} x_i \times v_i\right\}$$

0/1 背包问题转换为求出 n 个物品的一个子集满足上述条件。采用组合列举的穷举法求解，用 maxv 表示最优解的总价值，用 maxw 表示最优解的总重量，用 maxi 表示最优解的选择物品方案，求出 0～$n-1$ 的幂集 ps，然后枚举 ps 中的每个子集，将每个子集看成一种背包装入方案，求出所选物品总重量 sumw 和总价值 sumv，若 sumw≤W 且 sumv＞maxv 成立，说明找到一个更优解，用 maxi 存放该解的编号，最后输出解。

求幂集采用 3.5 节中的解法 1，用 ps 枚举 0～2^n-1 的十进制数，不必求出子集。对应

的穷举算法如下：

```
void knap(int w[],int v[],int W,int n) {                    //求 0/1 背包问题
    int maxv=0,maxw=0,maxi;
    for(int ps=0;ps<(1<<n);ps++) {                          //执行 2^n 次
        int sumw=0,sumv=0;
        for(int i=0;i<n;i++) {
            if(ps&(1<<i)) {                                 //选择了物品 i
                sumw+=w[i];
                sumv+=v[i];
            }
        }
        if(sumw<=W && maxv<sumv) {                          //找到更优解
            maxw=sumw;
            maxv=sumv;
            maxi=ps;
        }
    }
    printf("最佳方案：\n");                                  //找到最优解
    printf(" 选中物品：");
    printf("{ ");
    for (int i=0;i<n;i++) {
        if(maxi&(1<<i)) printf("%d ",i);
    }
    printf("}\n");
    printf(" 总重量=%d, 总价值=%d\n",maxw,maxv);
}
```

对于表 2.1 所示的 4 个物品，当 $W=6$ 时，求解结果如下：

```
最佳方案：
    选中物品：{ 1 2 3 }
    总重量=6, 总价值=8
```

knap 算法分析： 对于 n 个物品，主要时间花费在求幂集上，所以算法的时间复杂度为 $O(2^n)$。

扫一扫

视频讲解

【例 3.6】 背包问题Ⅱ(LintCode125★★)。有 $n(n\leqslant 100)$个物品和一个容量为 $m(m\leqslant 1000)$的背包，给定数组 A 表示物品的重量，数组 V 表示物品的价值。设计一个算法求最多能装入背包的总价值，要求物品不能被切分，每个物品只能取一次，同时装入背包的物品的总重量不能超过 m。例如，$m=10$，$A=\{2,3,5,7\}$，$V=\{1,5,2,4\}$，答案是 9，装入 $A[1]$ 和 $A[3]$可以得到最大价值，对应的价值为 $V[1]+V[3]=9$。要求设计如下成员函数：

```
int backPackII(int m, vector<int> &A, vector<int> &V) { }
```

扫一扫

源程序

解 采用前面求解 0/1 背包问题的穷举思路，由于在测试数据中 n 最大为 100，难以表示 2^n(尽管可以采用多个 int 整数表示 2^n，但实现起来较复杂)，为此采用 3.5 节中的解法 2 求幂集，这样设计的程序在提交时内存超过限制(memory limit exceeded)，例如，$n=100$ 时 ps 集合中包含 2^n 个子集元素，远超正常内存容量，所以本问题采用穷举法不可行。

3.7 求全排列

穷举算法中的排列列举与求排列相关，本节介绍求全排列的相关算法设计，后面讨论排列列举的几个应用示例。

问题描述：对于给定的正整数 $n(n \geqslant 1)$，求 $1 \sim n$ 的全排列 pm。例如，$n=3$ 时 pm=｛｛1,2,3｝,｛1,3,2｝,｛3,1,2｝,｛2,1,3｝,｛2,3,1｝,｛3,2,1｝｝(排列顺序可以任意)。

解法 1：这里采用增量穷举法求解。产生 1～3 全排列的过程如图 3.10 所示，这里 $n=3$，用 pm 表示全排列结果(它的每一个元素是一个整数集合)，其求解过程如下：

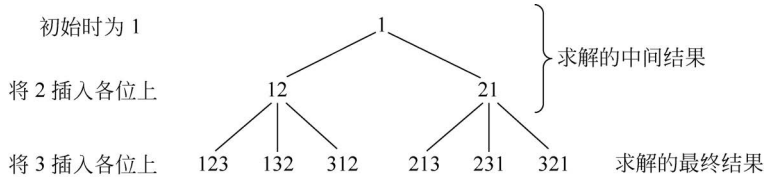

图 3.10 产生 1～3 全排列的过程

(1) 产生一个｛1｝集合元素添加到 pm 中，即 pm=｛｛1｝｝。

(2) $i=2$，置 pm1=pm，将 pm 清空，对于 pm1 的每个集合元素 e，依位置从后向前的次序在 e 的每个位置插入整数 i 产生新集合元素 $e1$，将 $e1$ 添加到 pm 中，即 pm=｛｛1,2｝,｛2,1｝｝。

(3) $i=3$，置 pm1=pm，将 pm 清空，对于 pm1 的每个集合元素 e，依位置从后向前的次序在 e 的每个位置插入整数 i 产生新集合元素 $e1$，将 $e1$ 添加到 pm 中，即 pm=｛｛1,2,3｝,｛1,3,2｝,｛3,1,2｝,｛2,1,3｝,｛2,3,1｝,｛3,2,1｝｝。

与存放幂集一样，用 vector < vector < int >>类型的容器 pm 存放全排列。按照上述过程对应的迭代算法如下：

```
vector<vector<int>> perm1(int n) {                    //算法 1:迭代求 1～n 的全排列
    vector<vector<int>> pm={{1}};                     //存放全排列
    for (int i=2;i<=n;i++) {                          //循环添加 2～n
        vector<vector<int>> pm1=pm;
        pm.clear();
        for (auto it=pm1.begin();it!=pm1.end();it++) {
            vector<int> e=(*it);                      //取出 pm1 中的一个元素 e
            for (int j=e.size();j>=0;j--) {           //在 e 的每个位置插入 i
                vector<int> e1=e;
                auto it=e1.begin()+j;                 //求出插入位置
                e1.insert(it,i);                      //插入整数 i
                pm.push_back(e1);                     //添加到 pm 中
            }
        }
    }
    return pm;
}
```

对应的递归算法如下：

```
vector<vector<int>> perm2(int n) {                    //算法 2:递归求 1～n 的全排列
    if(n==1) return {{1}};
    vector<vector<int>> pm1=perm2(n-1);               //求 1～n-1 的全排列
    vector<vector<int>> pm;
    for (auto it=pm1.begin();it!=pm1.end();it++) {    //循环(n-1)!次
        vector<int> e=(*it);                          //取出 pm1 中的一个元素 e
        for (int j=e.size();j>=0;j--) {               //在 e 的每个位置插入 i,循环 n 次
            vector<int> e1=e;
            auto it=e1.begin()+j;                     //求出插入位置
            e1.insert(it,n);                          //插入整数 i
            pm.push_back(e1);                         //添加到 pm 中
```

扫一扫

视频讲解

```
            }
        }
        return pm;
    }
```

perm1 和 perm2 算法分析：两个算法的时间复杂度相同。以 perm2 算法为例，对应的时间递推式如下：

$T(1)=1$

$T(n)=T(n-1)+n!$　当 $n>1$ 时

可以推出 $T(n)=O(n\times n!)$。

解法 2：在 STL 中提供了求全排列的相关函数，例如 next_permutation(beg,end)，其功能是将[beg,end)置为一个排列，若没有下一个排列，返回 false。利用该函数求 1～n 的全排列的算法如下：

```
vector<vector<int>> perm3(int n) {                    //算法 3:求 1～n 的全排列
    vector<vector<int>> pm;                           //存放 1～n 的全排列
    vector<int> e;
    for(int i=1;i<=n;i++)                             //在 e 中添加 1～n
        e.push_back(i);
    do {
        pm.push_back(e);
    } while(next_permutation(e.begin(),e.end()));     //取 e 的下一个排列
    return pm;
}
```

perm3 算法分析：该算法中调用一次 next_permutation()的时间为 $O(n)$，do-while 循环 $n!$次，所以算法的时间复杂度为 $O(n\times n!)$。

说明：使用 next_permutation 求全排列的好处是可以按字典顺序返回所有的排列。

扫一扫 视频讲解

【例 3.7】 全排列(LeetCode46★★)。给定一个不含重复数字的数组 nums，设计一个算法求其所有可能的全排列，可以按任意顺序返回答案。要求设计如下成员函数：

```
vector<vector<int>> permute(vector<int>& nums) { }
```

解 采用前面求 1～n 全排列的解法 1 思路，仅将初始序列由 1～n 改为 nums[0..$n-1$]。

扫一扫 源程序

3.8 n 皇后问题

扫一扫 视频讲解

问题描述：在 $n\times n$ 的方格棋盘上放置 n 个皇后，要求所有皇后不同行、不同列、不同左右对角线。如图 3.11 所示为 6 皇后问题的一个解。设计一个算法求出 n 皇后问题的全部解。

解 n 个皇后编号为 0～$n-1$，采用整数数组 $q[n]$存放 n 皇后问题的求解结果，因为每行只能放一个皇后，$q[i]$(0≤i≤$n-1$)表示皇后 i 所在的列号，即皇后 i 放在(i,$q[i]$)位置。对于图 3.11 所示的解，$q[0..5]=\{1,3,5,0,2,4\}$。

由于 n 个皇后的列号的取值范围是 0～$n-1$，每个皇后的列号是唯一的，所以 n 皇后的一个解 q 一定是 0～$n-1$ 的某个排列，并且 n 个皇后位置(i,$q[i]$)(0≤i≤$n-1$)相互之间

没有冲突。为此利用穷举法求解，列举方法采用排列列举，也就是说枚举 0～$n-1$ 的全排列，检测一个排列是否为一个 n 皇后问题的解，若是，则输出对应的解。

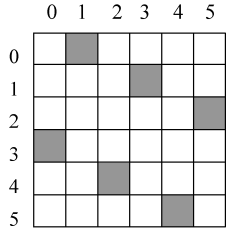

图 3.11 6 皇后问题的一个解

当 q 是 0～$n-1$ 的某个排列时，如何确定 n 个皇后位置(i，$q[i]$)相互之间没有冲突呢？假设前面 i 个皇后(即皇后 0～皇后 $i-1$)之间没有冲突，现在考虑皇后 i 是否与前面的 i 个皇后存在冲突，皇后 i 的位置为(i，$q[i]$)，前面 i 个皇后的位置是(k，$q[k]$)($0 \leqslant k \leqslant i-1$)。

(1) 皇后 i 不能与皇后 k($0 \leqslant i \leqslant n-1$)同列，若同列，则有 $q[k]=q[i]$成立。

(2) 皇后 i 不能与皇后 k($0 \leqslant i \leqslant n-1$)同左右对角线。如图 3.12 所示，若皇后 i 与皇后 k 在一条对角线上，则构成一个等腰直角三角形，即$|q[k]-q[i]|=|i-k|$。

也就是说，若皇后 i 的位置(i，$q[i]$)与任意一个皇后 k 的位置(k，$q[k]$)满足条件(q[k]==q[i]) || (abs(q[k]-q[i])==abs(i-k))，说明皇后 i 与皇后 k 存在冲突。

如果皇后 i 与任意一个皇后 k($0 \leqslant i \leqslant n-1$)均不满足上述条件，则说明皇后 i 与前面已经放置的 i 个皇后没有冲突，或者说这样的 $i+1$ 个皇后是没有冲突的，如果 n 个皇后没有冲突，则对应的 q 就是一个解。

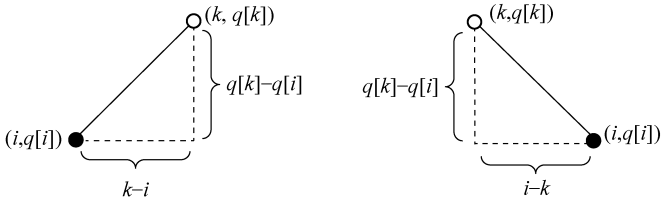

图 3.12 两个皇后构成对角线的情况

为了简单，在枚举 n 皇后问题的解 q 时先置 q 为{0，1，…，$n-1$}，当判定 q 是否为一个解后再直接利用 STL 的 next_permutation(q，$q+n$)函数得到下一个排列，直到枚举完所有的排列。对应的输出 n 皇后问题的全部解的穷举法算法如下：

```
int cnt;                                    //累计解个数
void disp(int n,int q[]) {                  //输出 n 皇后问题的一个解
    printf(" 第%d 个解:",++cnt);
    for (int i=0;i<n;i++)
        printf("(%d,%d) ",i,q[i]);
    printf("\n");
}
bool valid(int i,int q[]) {                 //测试(i,q[i])位置是否与前面的皇后不冲突
    if (i==0) return true;                  //皇后 0 一定没有冲突
    int k=0;
    while (k<i) {                           //k=0～i-1 是已放置了皇后的行
        if ((q[k]==q[i]) || (abs(q[k]-q[i])==abs(k-i)))
            return false;                   //(i,q[i])与皇后 k 有冲突
        k++;
    }
    return true;
}
bool isaqueen(int n,int q[]) {              //判断 q 是否为 n 皇后问题的一个解
    for(int i=1;i<n;i++) {
```

```
        if(!valid(i,q))                          //若皇后 i 的位置不合适,则返回 false
            return false;
    }
    return true;
}
void queen(int n) {                              //求解算法
    cnt=0;
    int q[20];                                   //q[i]存放皇后 i 的列号
    for(int i=0;i<n;i++)                         //初始化 q 为 0~n-1
        q[i]=i;
    do {
        if(isaqueen(n,q))                        //q 是一个解时输出
            disp(n,q);
    } while(next_permutation(q,q+n));            //取 a 的下一个排列
}
```

调用上述算法求出的 6 皇后问题的 4 个解如下,对应的图示如图 3.13 所示。

```
第 1 个解:(0,1) (1,3) (2,5) (3,0) (4,2) (5,4)
第 2 个解:(0,2) (1,5) (2,1) (3,4) (4,0) (5,3)
第 3 个解:(0,3) (1,0) (2,4) (3,1) (4,5) (5,2)
第 4 个解:(0,4) (1,2) (2,0) (3,5) (4,3) (5,1)
```

queen 算法分析:其中 valid(n,q)算法的执行时间为 $O(n)$,isaqueen(n,q)算法的最坏时间复杂度为 $O(n^2)$,调用一次 next_permutation(q,$q+n$)的时间为 $O(n)$,queen(n)算法中 do-while 循环的次数为 $n!$,所以执行总时间为 $2^n(O(n^2)+O(n))$,即 $O(n^2\times n!)$。

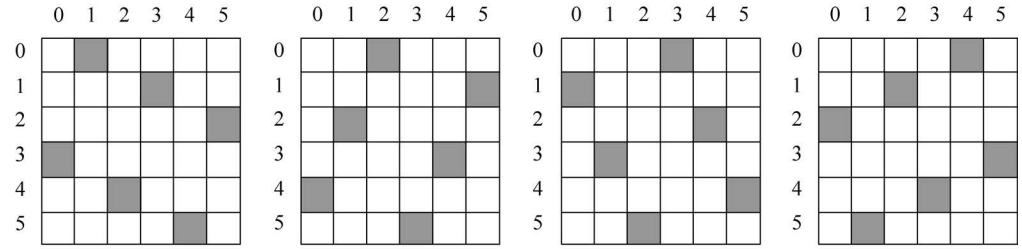

图 3.13 6 皇后问题解的描述

3.9 任务分配问题

扫一扫

视频讲解

问题描述:有 $n(n\geqslant 1)$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,人员和任务的编号均为 0~$n-1$。第 i 个人执行第 j 个任务的成本是 $c[i][j](0\leqslant i,j\leqslant n-1)$。设计一个算法求出总成本最小的一种分配方案,并结合如表 3.2 所示的问题求解。

解 考虑为人员分配唯一的任务,用 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$表示,$x_i$ 表示人员 i 分配的任务编号,由于每个人有且仅有一个分配的任务,而任务编号是 0~$n-1$,所以每种分配方案一定是 0~$n-1$ 的一个排列,全部可能的分配方案恰好是 0~$n-1$ 的全排列。

通过枚举 0~$n-1$ 的全排列,计算出每种分配方案的成本,比较求出最小成本的方案,即最优方案。对于表 3.2 所示的示例,$n=4$,共 24 种分配方案:

分配方案 1：$x=(0,1,2,3)$，对应总成本＝18

分配方案 2：$x=(0,1,3,2)$，对应总成本＝30

分配方案 3：$x=(0,3,1,2)$，对应总成本＝33

……

分配方案 24：$x=(3,2,1,0)$，对应总成本＝26

表 3.2　4 个人员、4 个任务的信息

人　员	任务 0	任务 1	任务 2	任务 3
0	9	2	7	8
1	6	4	3	7
2	5	8	1	8
3	7	6	9	4

在全部方案中比较求出最优方案是 $x=(1,0,2,3)$，即人员 0 分配任务 1，人员 1 分配任务 0，人员 2 分配任务 2，人员 3 分配任务 3，最优成本为 13。对应的穷举算法如下：

```
vector<vector<int>> perm1(int n) {                    //迭代求 0～n-1 的全排列
    vector<vector<int>> pm={{0}};                     //存放全排列
    for (int i=1;i<n;i++) {                           //循环添加 1～n-1
        vector<vector<int>> pm1=pm;
        pm.clear();
        for (auto it=pm1.begin();it!=pm1.end();it++) {
            vector<int> e=(*it);                      //取出 pm1 中的一个元素 e
            for (int j=e.size();j>=0;j--) {           //在 e 的每个位置插入 i
                vector<int> e1=e;
                auto it=e1.begin()+j;                 //求出插入位置
                e1.insert(it,i);                      //插入整数 i
                pm.push_back(e1);                     //添加到 pm 中
            }
        }
    }
    return pm;
}
void allocate(vector<vector<int>> & c) {              //求任务分配问题的最优方案
    int n=c.size();
    vector<vector<int>> pm=perm1(n);
    vector<int> bestx;                                //最优分配方案
    int mincost=INF;                                  //最小成本(初始化为∞)
    for (int f=0;f<pm.size();f++) {                   //求每个分配方案的成本
        vector<int> x=pm[f];                          //取当前分配方案 x
        int cost=0;
        for (int i=0;i<x.size();i++)                  //人员 i 分配任务 x[i]
            cost+=c[i][x[i]];
        printf("分配方案: x=(");
        for(int k=0;k<x.size();k++)
            printf("%d ",x[k]);
        printf(") 总成本=%d\n",cost);
        if (cost<mincost) {                           //通过比较求最小成本的方案
            mincost=cost;
            bestx=x;
        }
    }
    printf("最优方案:\n");                             //输出结果
    for (int i=0;i<bestx.size();i++)
        printf(" 人员%d 分配任务%d\n",i,bestx[i]);
```

```
        printf(" 总成本=%d\n",mincost);
    }
```

allocate 算法分析：perm1 求全排列的时间复杂度为 $O(n\times n!)$，后面求最优方案的时间复杂度也是如此，所以总的时间复杂度为 $O(n\times n!)$。

扫一扫 视频讲解

【例 3.8】 订单分配(LintCode1909★★)。假定有 $n(1\leqslant n\leqslant 8)$个订单，待分配给 n 个司机，每个订单在匹配司机前会对候选司机进行打分，打分的结果保存在 $n\times n$ 的矩阵 score 中，其中 score[i][j](0≤score[i][j]≤100)代表订单 i 派给司机 j 的分值。假定每个订单只能派给一位司机，司机只能分配到一个订单。设计一个算法求最终的派单结果，使得匹配的订单和司机的分值累加起来最大，并且所有订单都得到分配。题目保证每组数据的最大分数的分配方案都是唯一的。例如，$n=3$，score={{1,2,4},{7,11,16},{37,29,22}}，答案是{1,2,0}，即 3 个订单分别分派给司机 1、2 和 0，对应的最大分值是 55。要求设计如下成员函数：

扫一扫 源程序

```
vector<int> orderAllocation(vector<vector<int>> &score) { }
```

解 与前面讨论的求解任务分配问题类似，将司机看成人员，将订单看成任务，仅将求最小成本改为求最大得分。

3.10 旅行商问题

扫一扫 视频讲解

问题描述：旅行商问题又称为货郎担问题(简称 TSP 问题)，是数学领域中著名的问题之一。假设一个旅行商要从某个城市 s 出发拜访 n 个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到出发城市。n 个城市的编号为 0～$n-1$，每个城市对应城市道路图中的一个顶点，假设城市道路图采用邻接矩阵 $\boldsymbol{A}$ 存储，起始点为 s，求出这样的路径中长度最短的路径，并结合如图 3.14 所示的城市道路图以 $s=0$ 为例讨论求解过程。

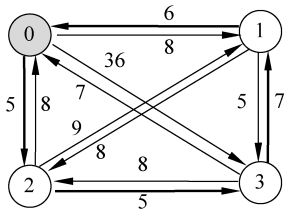

图 3.14 一个 4 城市的道路图

解 题目就是求从顶点 s 出发经过其他 $n-1$ 个顶点并且回到顶点 s 的最短路径，除了起始点和终点以外，路径中的其他顶点不重复，如图 3.15 所示，称这样的路径为 TSP 路径，对应的路径长度称为 TSP 路径长度。例如图 3.14 中用粗线标注的回路就是一条 TSP 路径，其路径长度为 23。

设候选 TSP 路径中除了起始点和终点以外 $x=\{x_0,x_1,\cdots,x_{n-2}\}$，则对应的路径长度=$A[s][x_0]+A[x_0][x_1]+\cdots+A[x_{n-3}][x_{n-2}]+A[x_{n-2}][s]$。依题意，$x_i$ 只能是 0～$n-1$ 中非 s 的顶点，为此采用穷举法，初始时置 x 为非 s 的 $n-1$ 个顶点，将其作为第一条路径，求出路径长度 curlen，然后将 x 的每一个排列看成不同的路径，同样求出路径长度 curlen，在所有的 curlen 中比较求最小值 bestlen，对应的路径用 bestx 表示，那么 TSP 路径就是 bestx。这里的求全排列直接利用 STL 的 next_permutation()函数。对应的算法如下：

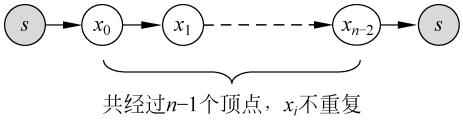

图 3.15 TSP 路径

```
void disppath(vector<int> &x,int plen,int s) {                    //输出一条路径 x
    printf("%d",s);
    for(int i=0;i<x.size();i++)
        printf("->%d",x[i]);
    printf("->%d 路径长度=%d\n",s,plen);
}
void TSP(vector<vector<int>> &A,int s) {                          //求解算法
    int n=A.size();
    int bestlen=INF;                                              //存放最短路径长度
    vector<int> bestx,x;                                          //bestx 存放最短路径
    for(int i=0;i<n;i++) {                                        //将非 s 的顶点添加到 x 中
        if(i!=s) x.push_back(i);
    }
    printf("TSP 求解\n");
    int cnt=0;                                                    //累计路径数
    do {
        int curlen=0,u=s,j=0;
        while(j<x.size()) {
            int v=x[j];
            curlen+=A[u][v];                                      //对应一条边<u,v>
            u=v;
            j++;
        }
        curlen+=A[u][s];
        printf(" 路径%d: ",++cnt); disppath(x,curlen,s);
        if(curlen<bestlen) {                                      //比较求最短路径
            bestlen=curlen;
            bestx=x;
        }
    } while(next_permutation(x.begin(),x.end()));
    printf(" 最短路径:"); disppath(bestx,bestlen,s);
}
```

以图 3.14 所示的一个 4 城市的道路图为例，起点 s 为 0 时调用上述算法的输出结果如下：

```
TSP 求解
    路径 1: 0->1->2->3->0 路径长度=28
    路径 2: 0->1->3->2->0 路径长度=29
    路径 3: 0->2->1->3->0 路径长度=26
    路径 4: 0->2->3->1->0 路径长度=23
    路径 5: 0->3->1->2->0 路径长度=59
    路径 6: 0->3->2->1->0 路径长度=59
    最短路径:0->2->3->1->0 路径长度=23
```

TSP 算法分析：算法中 do-while 循环执行$(n-1)!$次，一次调用 next_permutation()的时间为$O(n)$，一次循环求路径长度的时间为$O(n)$，合并总时间为$(O(n)+O(n))\times(n-1)!$，对应的时间复杂度为$O(n!)$。

【例 3.9】 旅行计划(LintCode1891★★)。有$n(n\leqslant 10)$个城市，城市编号为 0～$n-1$。给出邻接矩阵 arr 代表任意两个城市的距离，其中 arr[i][j](arr[i][j]$\leqslant$10 000)表示从城市 i 到城市 j 的距离。Alice 在周末制定了一个游玩计划，她从所在的城市 0 开始，游玩其他的 1～$n-1$ 的全部城市，最后回到城市 0。Alice 想知道她能完成游玩计划需要行走的最

扫一扫

视频讲解

小距离，返回这个最小距离。除了城市 0 以外每个城市都只能经过一次，且城市 0 只能是起点和终点，Alice 中途不可经过城市 0。例如，arr={{0,1,2},{1,0,2},{2,1,0}}，答案是 4，对应的路径是城市 0->城市 2->城市 1->城市 0。要求设计如下成员函数：

```
int travelPlan(vector<vector<int>> &arr) { }
```

扫一扫

源程序

解 Alice 最小距离的旅游路径就是一条 TSP 路径，这里仅求 TSP 路径长度。采用穷举法求解本例的思路与前面讨论的求解 TSP 路径完全相同，仅将起始点 s 设置为 0 即可。

3.11 练 习 题

扫一扫

自测题

1. 简述穷举法的基本思想。
2. 简述穷举法中有哪几种列举方法。
3. 举一个例子说明前缀和数组的应用。
4. 举一个例子说明并查集的应用。
5. 考虑下面的算法，用于求数组 a 中相差最小的两个元素的差。请对这个算法做尽可能多的改进。

```
int mindif(int a[],int n) {
    int dmin=INF;
    for (int i=0;i<=n-2;i++) {
        for (int j=i+1;j<=n-1;j++){
            int temp=abs(a[i]-a[j]);
            if (temp<dmin)
                dmin=temp;
        }
    }
    return dmin;
}
```

6. 为什么采用穷举法求解 n 皇后问题时列举方式是排列列举？
7. 给定一个整数数组 $a=(a_0,a_1,\cdots,a_{n-1})$，若 $i<j$ 且 $a_i>a_j$，称$<a_i,a_j>$为一个逆序对。例如数组(3,1,4,5,2)的逆序对有$<3,1>$、$<3,2>$、$<4,2>$、$<5,2>$。设计一个算法采用穷举法求 a 中逆序对的个数，即逆序数。
8. 求最长回文子串(LeetCode5★★)。给定一个字符串 s，设计一个算法求 s 中的最长回文子串，如果有多个最长回文子串，求出其中的任意一个。例如，s="babad"，答案为"bab"或者"aba"。要求设计如下成员函数：

```
string longestPalindrome(string s) { }
```

9. 求解涂棋盘问题。小易有一块 $n\times n$ 的棋盘，棋盘的每一个格子都为黑色或者白色，小易现在要用他喜欢的红色去涂画棋盘。小易会找出棋盘的某一列中拥有相同颜色的最大区域去涂画，帮助小易计算他会涂画多少个棋盘格。

输入格式：输入数据包括 $n+1$ 行，第一行为一个整数 $n(1\leqslant n\leqslant 50)$，即棋盘的大小，接下来的 n 行每行一个字符串，表示第 i 行棋盘的颜色，'W'表示白色，'B'表示黑色。

输出格式：输出小易会涂画的区域大小。

输入样例：

```
3
BWW
BBB
BWB
```

输出样例：

```
3
```

10. 电话号码的字母组合(LeetCode17★★)。给定一个仅包含数字2～9的长度为n(0≤n≤4)的字符串digits，设计一个算法求所有能表示的字母组合，答案可以按任意顺序返回。给出数字到字母的映射如图3.16所示(与电话的按键相同)，注意1不对应任何字母。例如，digits="23"，答案是{"ad","ae","af","bd","be","bf","cd","ce","cf"}。要求设计如下成员函数：

图3.16 电话按键表示的数字到字母的映射

```
vector<string> letterCombinations(string digits) { }
```

11. 求子数组的和为k的个数(LintCode838★)。给定一个整数数组nums和一个整数k，设计一个算法求该数组中连续子数组的和为k的总个数。例如，nums={2,1,-1,1,2}，k=3，和为3的子数组有(2,1)、(2,-1,1,2)、(2,1,-1,1)和(1,2)，答案为4。要求设计如下成员函数：

```
int subarraySumEqualsK(vector<int> &nums, int K) { }
```

12. 给定n个城市(城市编号为从1到n)，城市和无向道路成本之间的关系为三元组(A,B,C)，表示在城市A和城市B之间有一条路，成本是C，所有的三元组用tuple表示。现在需要从城市1开始找到旅行所有城市的最小成本。每个城市只能通过一次，可以假设能够到达所有的城市。例如，n=3，tuple={{1,2,1},{2,3,2},{1,3,3}}，答案为3，对应的最短路径是1→2→3。

13. n皇后问题Ⅱ(LintCode34★★)。给定一个整数n(n≤10)，设计一个算法求n皇后问题的解的数量。例如，n=4时答案为2，也就是说4皇后问题共有两个解。要求设计如下成员函数：

```
int totalNQueens(int n) { }
```

3.12 在线编程实验题

1. LintCode1068——寻找数组的中心索引★
2. LintCode1517——最大子数组★
3. LintCode1338——停车困境★
4. LintCode993——数组划分Ⅰ★

5. LintCode406——和大于 s 的最小子数组★★
6. LintCode1331——英语软件★
7. LintCode397——最长上升连续子序列★
8. LeetCode1534——统计好三元组★
9. LeetCode204——计数质数★★
10. LeetCode187——重复的 DNA 序列★★
11. LeetCode2018——判断单词是否能放入填字游戏内★★
12. LeetCode2151——基于陈述统计最多好人数★★★
13. POJ2000——金币
14. POJ1013——假币问题
15. POJ1256——字谜
16. POJ3187——倒数和