第 2 章 递归与分治策略

分治策略是有效算法设计中普遍采用的一种技术，是“分而治之”(Divide and conquer)的意思。分治策略主要包含两个步骤：分、治。分：就是递归地将原问题分解成子问题；治：则是分别解决各个子问题后再合并子问题的解，从而得到整个问题的解。先分后治的解决问题的过程就是分治策略的核心。软件的体系结构设计、模块化设计等就是分治策略的具体体现。

2.1 分治策略基本思想

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关，问题的规模越小，越容易直接求解。直接解决一个规模较大的问题很困难时，可以考虑分治策略。分治策略的基本思想是把一个规模较大的复杂的问题，拆分成多个规模较小的子问题，解决子问题的难度比解决原问题难度简单的多。所以分治策略的难点就在于如何拆分原问题，因为拆分出来的子问题必须和原问题是同样结构，或者说是相同的问题，只是规模更小一些。拆分出来的子问题如果不是足够小，还可以用同样的方式进一步的拆分。将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

在计算机科学中，分治策略是一种很重要的算法，是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换(快速傅立叶变换)等。

所以，分治策略先做分解，分解之后得到一系列的子问题的解，然后再把子问题的解一步一步整合，得到最终的原问题的解。利用分治策略解决问题对问题本身有一定的要求，首先，问题的规模缩小到一定的程度要能够容易的解决，否则就失去了拆分的意义；其次，问题必须要能够分解成若干个规模较小的相同子问题，如果分出来的子问题不是原问题也不行；最后分解出来的子问题的解要能够合并为原来问题的解。同时，各个子问题要求是相互独立的，不相互独立也不行。这就是分治策略的基本思想以及对分治策略应用的基本要求。

比如一个大的问题，可以分割为对 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。然后将求出的小规模的问题的解自底向上合并为一个更大规模的问题的解，逐步求出原来问题的解。

在分治策略中要解决的规模较大的复杂问题可以拆分为多个同类型的子问题，如果有一

个算法可以解决这个规模较大的复杂问题，那么同质的子问题也能够用这个算法解决，这里就涉及到了递归的概念。当然，实际问题中总是先解决子问题然后抽象出递归算法的。

2.2 分治策略的适用条件

分治策略所能解决的问题一般具有以下几个特征：

该问题的规模缩小到一定的程度就可以容易地解决；

该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质（一个问题的最优解包含其子问题的最优解，这个问题就具有最优解）。

利用该问题分解出的子问题的解可以合并为该问题的解；

该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

这条特征涉及到分治策略的效率，如果各子问题是不独立的，则分治策略要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治策略，但一般用动态规划较好。

分治策略的基本步骤：

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

通过大量实践发现，用分治策略设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的，使子问题规模大致相等的做法是出自一种平衡问题的思想，它几乎总是比子问题规模不等的做法要好。

2.3 分治策略的复杂性分析

一个分治策略将规模为 n 的问题分成 k 个规模为 n / m 的子问题去解。设分解阈值 n0=1，且 adhoc 解规模为 1 的问题耗费 1 个单位时间。再设将原问题分解为 k 个子问题以及用 merge

将 k 个子问题的解合并为原问题的解需用 f(n)个单位时间。用 T(n)表示该分治策略解规模为|P|=n 的问题所需的计算时间，则有：

$$T(n)=\begin{cases} O(1) & n=1 \\ kT(n/m)+f(n) & n>1 \end{cases}$$

通过迭代法求得方程的解：

$$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$$

注意：递归方程及其解只给出 n 等于 m 的方幂时 T(n)的值，但是如果认为 T(n)足够平滑，那么由 n 等于 m 的方幂时 T(n)的值可以估计 T(n)的增长速度。通常假定 T(n)是单调上升的，从而当 mi≤n<mi+1 时，T(mi)≤T(n)<T(mi+1)。

2.4 递归及应用

2.4.1 递归的概念

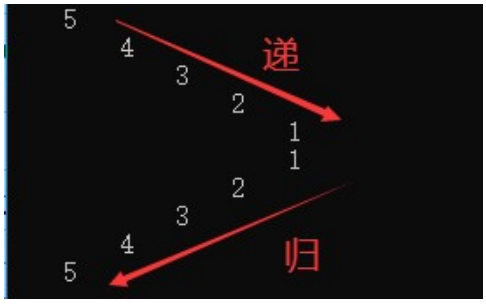
直接或间接地调用自身的算法称为递归算法。用函数自身给出定义的函数称为递归函数。由分治策略产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解，相当于递归过程的产生。

【例 0】递归概念的引入。

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

class RecuDemo {
private:
  int   _dep;

  void  blank(int dif=0)  { printf("%*s",_dep*4+dif,"");          }
  void  dispush(int it)   { _dep++; blank();  printf("%5d\n",it);   }
  void  dispop(int it)    { blank(); _dep--;  printf("%5d\n",it);   }

public:
  RecuDemo():_dep(-1) {}

  void recu(int cnt) {
    dispush(cnt);

    if(cnt>1) {
      recu(cnt-1);
    }

    dispop(cnt);
  }
};
```

```
int main()   {
  RecuDemo rd;
  rd.recu(5);
}
```

比如上面代码的运行结果如下图，很明显有递和归的过程。图中每递归一个层次则向右缩进一次，看图显然递归了 5 次。



递归就是有去（递）有回（归）。递可以理解成：大问题可以分解为若干个规模较小，与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决；归则是：问题的处理过程从大到小后达到一个明确的终点，一旦到达了这个临界点，则从这个临界点开始，原路返回到原点，原问题解决。

上面代码的运行结果也可以用循环实现，递归和循环是两种不同的解决问题的思路。递归通常很直白地描述了一个问题的求解过程，也是最容易被想到的解决问题的方式。就重复任务而言，循环和递归具有相同的特性，但使用循环的算法可能难以清晰地描述解决问题的步骤。

从算法设计上看，递归和循环并无优劣之分。在实际开发中，特别是在求解规模不确定的情况下，考虑到函数调用的开销，递归常常会带来性能问题；而循环没有函数调用的开销，所以效率比递归高。在大多数情况下，递归求解方式和循环求解方式可以互换。比如。用到递归的地方可以使用循环替换而不影响程序的阅读，同时也会带来性能的改善。

2.4.2 递归的要素及应用场景

理解了递归的基本思想，利用递归写程序一般需要考虑三个方面：

1. 明确递归终止条件

递归是有去有回的，所以必然有一个明确的临界点即递归终止条件。程序一旦到达这个临界点，就不用继续往下递而是逐步的回来。换句话说，该临界点可以防止无限递归。

2. 给出递归终止时的处理办法

在递归的临界点应该直接给出问题的解决方案。一般情况下，问题的解决方案是直观的、

容易的。

3．提取重复逻辑缩小问题规模

递归思想的内涵时在于递归问题必须可以分解为若干个规模较小、与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决。从程序实现的角度，必须抽象出一个干净利落的重复的逻辑，以便使用相同的方式解决子问题。

递归算法一般用于解决三类应用场景的问题：

1．问题的定义是递归的（Fibonacci 函数，阶乘，…）；

2．问题的解法是递归的（有些问题只能使用递归方法来解决，例如，汉诺塔问题，…）；

3．数据结构是递归的（链表、树等的操作，包括树的遍历，树的深度，…）。

2.4.3 递归应用的第一类问题 问题的定义是递归的

【例 1】阶乘函数可递归地定义为：

```
int factorial(int n)    {
  if (n== 0) return 1;
  return n*factorial(n-1);
}
```

$$n!=\begin{cases}1 & n=0\\ n(n-1)! & n>0\end{cases}$$

上式中的 n=0 称为边界条件即递归终止条件，同时也给出了递归终止时的处理方法，即递归过程结束，立即返回；n(n-1)！称为递归方程，这是提取出来的重复逻辑，边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

【例 2】 Fibonacci 数列

无穷数列 1，1，2，3，5，8，13，21，34，55，……，称为 Fibonacci 数列。它可以递归地定义。第 n 个 Fibonacci 数可递归地计算如下：

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

```
int fibonacci(int n)        {
   if(n<=1) return 1;
   return fibonacci(n-1)+fibonacci(n-2);
}
```

上面的例子可以看出递归的优缺点。

优点：结构清晰，可读性强，容易用数学的方法证明算法的正确性。为设计算法、调试程序带来了很大方便。

缺点：递归算法的运行效率较低。无论时间复杂性还是空间复杂性都比非递归算法要高的多。

因此，某些场合下可以设法在递归算法中消除递归调用，使其转化为非递归算法。

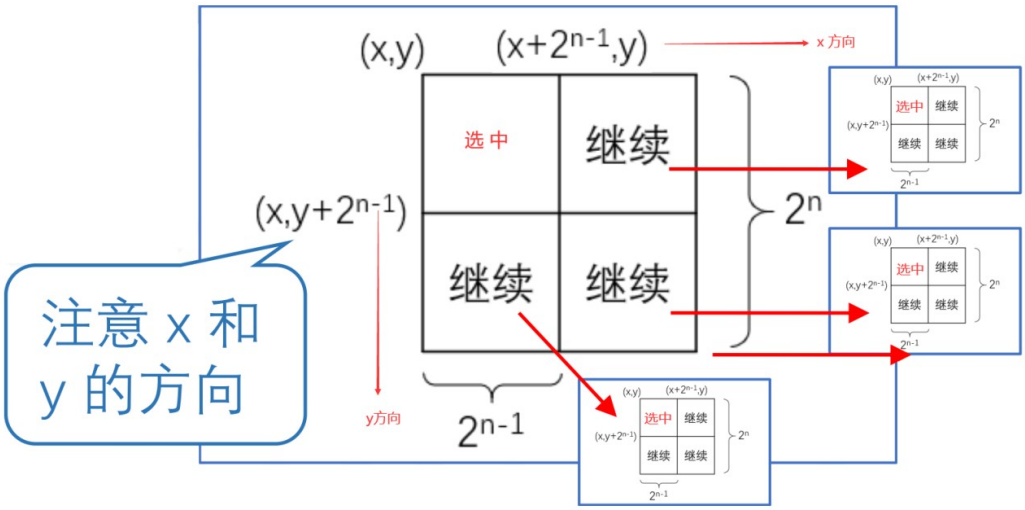
2.4.4 递归应用的第二类问题　问题的解法是递归的

【例 3】　后面讲到的全排列问题。

具体见后面的内容。

【例 4】小 A 是美食爱好者，到旅行目的地都会品尝当地的各式各样的佳肴。已知目的地有 $2^n \times 2^n$ 种美食，但是小 A 无法在一次旅行中全部品尝。程序员朋友给小 A 想了个方法：把 $2^n \times 2^n$ 种美食做成 $2^n \times 2^n$ 正方形的选项表，正方形可以均分成 4 个小正方形，每个小正方形边长是原正方形边长的一半。选择美食的方法是：左上角正方形的所有美食都被选中，剩下的 3 个正方形中，每一个正方形继续分为 4 个更小的正方形，然后通过同样方法选中美食……直到正方形无法再分下去为止。

下图可以看出，选中美食的过程是递归的过程。假设用二维数组表示美食的选中状态，0 表示选中，1 表示未选中。只要 n!=0，则总是对正方形中除左上角以外的三个正方形递归。当 n=1 时，此时的正方形是 $2^1 \times 2^1$，再次对该正方形中除左上角以外的三个正方形递归时，要注意结束条件，因为其余的三个正方形已经是 $2^0 \times 2^0$ 正方形了，即这三个正方形边长已经是 1，且不是左上角，所以这三个正方形的数组相应坐标置 1，表示未选中。



具体代码如下，其中 1<<n 表示 2^n。

```
#include <iostream>
using namespace std;

int a[1050][1050]={0};

void recu(int x,int y,int n) {
  if(n==0) a[x][y] = 1;
  else {
    recu(x+(1<<n-1), y, n-1);     // 右上方矩阵
    recu(x, y+(1<<n-1), n-1);     // 左下角矩阵
    recu(x+(1<<n-1), y+(1<<n-1), n-1); // 右下角
  }
}
int main() {
  int n=3;
  scanf("%d",&n);

  recu(0,0,n);
  for(int i=0; i<(1<<n); i++)  {
    for(int j=0; j<(1<<n); j++)  {
      printf("%d%c",a[i][j],j==(1<<n)-1?'\n':' ');
    }
  }
  return 0;
}
```

输出结果中，0 表示选中，1 表示未选中。

2.4.5 递归应用的第三类问题 数据结构是递归的

【例 5】给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

输入： n = 4, k = 2

输出：[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]。

本题是回溯法的经典题目，最直接的解法是使用 for 循环，例如 k=2，就用两个 for 循环，如函数 f1()。

```
//4 个里面选 2 个
void f1() {
  int n = 4;
  for(int i=1; i<=n; i++) {
    for(int j=i+1; j<=n; j++) {
      printf("%5d%5d\n",i,j);
    }
  }
}
```

如果不是 4 选 2 而是 10 选 3 也简单，就用三个 for 循环，如函数 f2()。

```
////10 个里面选 3 个
void f2() {
  int n = 10;
  for(int i=1; i<=n; i++) {
    for(int j=i+1; j<=n; j++) {
      for(int k=j+1; k<=n; k++) {
        printf("%5d%5d%5d\n",i,j,k);
      }
    }
  }
}
```

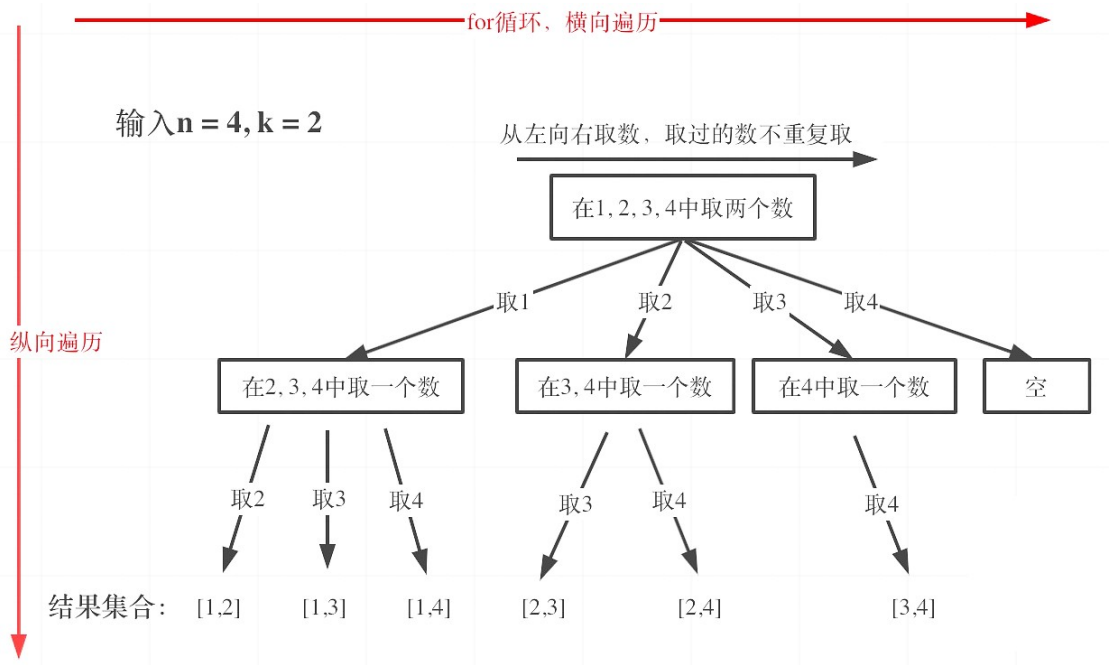
假如 n 为 100，k 为 50 的话就不太好玩了，总不能写嵌套 50 个 for 循环这样的暴力算法吧。当然这还算好的，如果题目的 n 和 k 可变的话，用这种方法就没法写了，因为无法控制写多少个 for 循环。在递归的概念小节中曾经提到：递归和循环是两种不同的解决问题的思路，也就是说，循环的问题也可以用递归解决。

对于解决 n 为 100，k 为 50 的情况，为避免暴力嵌套 50 层 for 循环，可以用递归解决嵌套层数的问题，相当于用递归做到嵌套 k 层 for 循环的效果。对于给定两个整数 n 和 k 的问题，每一次的递归中嵌套一个 for 循环，就解决多层嵌套循环的问题了，此时递归的层数在 k 为 50 的情况下，就是递归 50 层。

为便于理解给定两个整数 n 和 k 问题的递归过程，需要借助抽象图形结构来理解。实际上，能够递归回溯解决的问题都可以抽象为 N 叉树的树形结构，用树形结构理解递归相对比较容易。当 n=4，k=2 时，该组合问题抽象为如下树形结构：这棵树一开始集合是 1，2，3，4， 从左向右取数，取过的数不再重复取。第一次取 1，集合变为 2，3，4 ，因为 k 为 2，只需要再取一个数就可以了，分别取 2，3，4，得到集合[1,2] [1,3] [1,4]，以此类推。每次从集合中选取元素的可选择的范围随着选取的进行而收缩。

递归的搜索过程就是 N 叉树结构的遍历过程，下图可以看出 for 循环用来横向遍历，

递归的过程是纵向遍历。



图中显示 n 相当于树的宽度，k 相当于树的深度。通过对 N 叉树的遍历就能够组合到需要的结果集。每当搜索到叶子节点就找到了一个结果，相当于只需要把达到叶子节点的结果收集起来，就求得 n 个数中 k 个数的组合集合。

具体实现代码如下所示。

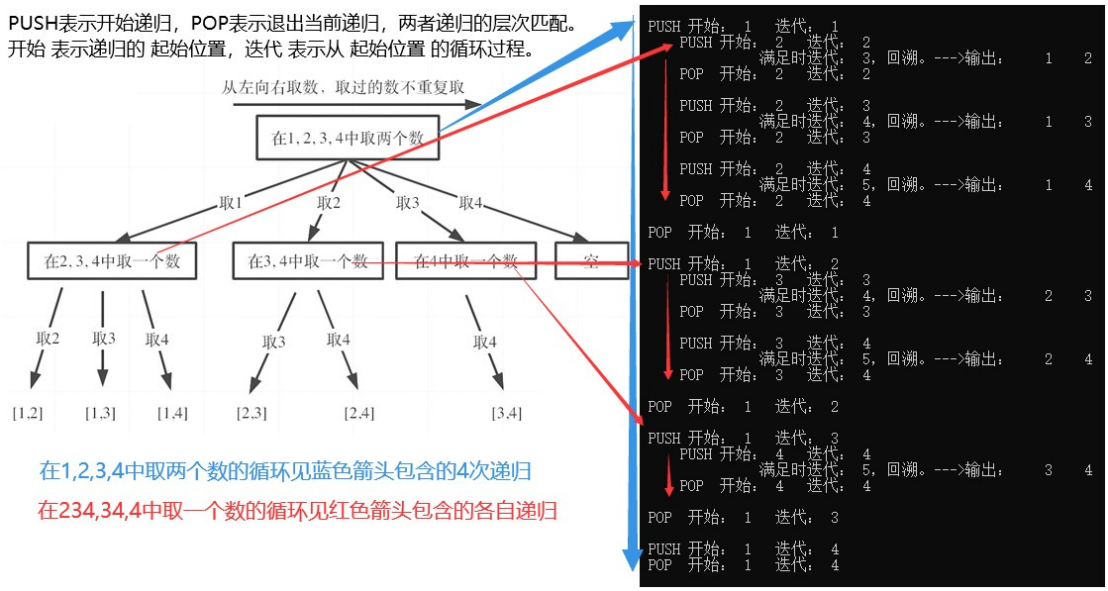
```
#include <iostream>
#include <vector>
using namespace std;

class Solution {
protected:
  typedef vector<int>     vint1;  //相当于一维整数数组类型
  typedef vector<vint1>   vint2;  //相当于二维整数数组类型

  void disp1(vint1 v1) {          //输出一维整数数组
    for(int i=0;i<v1.size();i++)  printf("%5d",v1[i]);
    puts("");
  }

  void disp2(vint2 v2) {          //输出二维整数数组
    for(int i=0;i<v2.size();i++)  disp1(v2[i]);
  }

private:
  int     n,k;
  vint2   ans;                  //组合的集合
  vint1   pth;                  //每次的搜索路径 即每一次的组合
  int     _dep;                 //递归时控制输出左边界距离

  void  blank(int dif=0) {   printf("%*s",_dep*4+dif,"");  }

  void  dispbt(int pos)  {        //输出回溯结果
    blank(10);
    printf("满足时迭代：%2d，回溯。--->输出：",pos);
    disp1(pth);
  }

  void  dispush(int pos,int it) {
```

```
    _dep++;    blank();
    printf("PUSH 开始：%2d   迭代：%2d\n",pos,it);
  }

  void  dispop(int pos,int it) {
    blank();    _dep--;
    printf("POP  开始：%2d   迭代：%2d\n\n",pos,it);
  }

public:
  Solution(int nn,int kk):n(nn),k(kk),_dep(-1) {}

  void backtrack(int pos) {
    if(pth.size() == k) {
      ans.push_back(pth);
      dispbt(pos);
      return;
    }

    for(int i=pos; i<=n; i++) {
      dispush(pos,i);

      pth.push_back(i);   //保存当前路径，即组合
      backtrack(i+1);     //递归
      pth.pop_back();     //剔除最后加入的元素

      dispop(pos,i);
    }
  }

  void disp() { disp2(ans);  }
};

int main()   {
  system("chcp 65001");
  Solution sl(4,2);       //求从 4 个数里面选 2 个数的组合
  sl.backtrack(1);        //从第一个数开始作为递归的最初数
  sl.disp();
}
```

代码中的形参 int pos 表示本层递归的集合开始遍历的下标。对于从 4 个数里面选 2 个数的组合就是从集合[1,2,3,4]中选取 2 个数构成的。之所以需要 pos 参数是因为每次从集合中选取元素，可选择的范围随着选取的进行而收缩，调整可选择的范围就靠 pos。

比如从集合[1,2,3,4]取 1 之后，下一层递归就要在[2,3,4]中取数，下一层递归靠的就是 pos 才知道从[2,3,4]中取数，所以需要 pos 记录下一层递归搜索的起始位置。

什么时候到达所谓的叶子节点即已经取了 k 个数？私有变量 pth 的大小如果达到 k，说明找到一个子集大小为 k 的组合。代码中 pth 保存的就是从根节点到叶子节点的路径即选取出来的 k 个数。



2.5 全排列

2.5.1 问题描述

设计一个递归算法生成 n 个元素{r1,r2,…,rn}的全排列。

全排列是将一组数按一定顺序进行排列，如果这组数有 n 个，那么全排列数为 n!个。

(1) 当 N = 1 的时候，则直接打印数列即可。

(2) 当 N = 2 的时候，

设数组为 x[a, b]　　打印 x[0], x[1] ，即 a，b。

交换 x[0], x[1]的内容　　打印 x[0], x[1] ，此时已变成了 b， a。

(3) 当 N = 3 的时候，数组为 x [a, b, c]

把 a 放在 x [0] 的位置，原本也是如此，x[0] = x[0]。打印 b，c 的全排列，即 x [1], x [2]的全排列：

a b c

a c b

把 b 放在 x[0]的位置，即交换原数组的 x[0]和 x[1]，然后打印 a, c 的全排列：

b a c

b c a

打印完后再换回原来的位置，即 a 还是恢复到 x[0]，b 还恢复到 x[1]的位置。

把 c 放在 x[0]的位置，即交换是原数组的 x[0]和 x[2]，然后打印 a，b 的全排列：

c b a

c a b

打印完后再换回原来的位置，即 a 还是恢复到 x[0]，b 还恢复到 x[1]的位置，至此，全排列完成。

当 n = 4，5，6,……的时候，以此类推。

简单地说：就是第一个数分别与后面的数进行交换。

设 E=a,b,c)，则 prem（E）= a.perm（b,c）+b.perm（a,c）+ c.perm（b，a）。然后 a.perm（b,c）= ab.perm（c）+ ac.perm（b）= abc + acb 依次递归进行。

设 R={r1,r2,…,rn}是要进行排列的 n 个元素，Ri=R-{ri}。

设集合 X 中元素的全排列记为 perm(X)，(ri)perm(Xi)表示在全排列 perm(Xi)的每一个排列前加上前缀得到的排列。R 的全排列可归纳定义如下：

当 n=1 时，perm(R)=(r)，其中 r 是集合 R 中唯一的元素；

当 n>1 时，perm(R)由(r1)perm(R1)，(r2)perm(R2)，…，(rn)perm(Rn)构成。

可见这是一个递归的过程，把对整个序列做全排列的问题归结为对它的子序列做全排列的问题。

2.5.2 算法分析

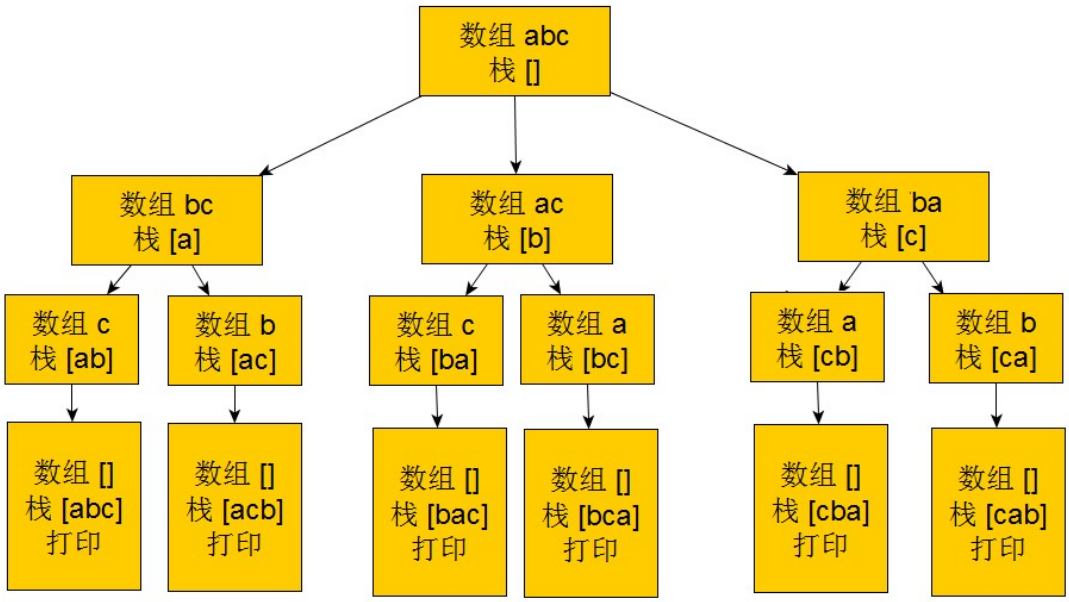
例如：使用分治算法求解全排列的过程如下

分解：将数组分为子数组 A[1..k-1] 和一个元素 A[k]。 (1≤k≤n)

解决：递归地求解每个子数组 A[1..k-1] 的全排列，直至子数组 A[1..k-1]为空时结束递归。

合并：将上一步的结果 A[1..k-1]的全排列（一个二维数组）与元素 A[k]合并，得出 A[1..k]的全排列。例如：

为了更容易理解，将整组数中的所有的数分别与第一个数交换，这样就总是在处理后 n-1 个数的全排列。



算法如下：

```
#include <iostream>
#include <vector>
using namespace std;

//全排列
template<typename T>  class Perm {
private:
  vector<T>   v;
  void swap(int lt,int rt) {  std::swap(v[lt],v[rt]); }

private:
  int   _dep;

  void  blank(int dif=0)  {   printf("%*s",_dep*4+dif,"");  }
  void  dispush(int lt,int rt) {
    _dep++; blank();
    printf("PUSH: %s\t 交换:%c %c\n",&v[0]+lt,v[rt],v[lt]);
  }
  void  dispop(int lt,int rt) {
    blank();_dep--;
    printf("POP:  %s\t 换回:%c %c\n\n",&v[0]+lt,v[lt],v[rt]);
  }
  void  disp()  {   blank();printf("----->%s\n",&v[0]);   }

public:
  Perm(const vector<T> &vec):v(vec),_dep(-1){}

  void perm(int fr,int to)  {
    if(fr==to) disp();
    else {
      for(int i=fr;i<=to;i++) {
        swap(fr,i);

        dispush(fr,i);
        perm(fr+1,to);

        swap(fr,i);
        dispop(fr,i);
      }
    }
  }
};

int main(int argc, char* argv[])  {
  system("chcp 65001");

  char s[30] = "abc";
  Perm<char> pm(vector<char>(s,s+sizeof(s)/sizeof(char)));
```

```
    pm.perm(0,2);  //闭区间

    return 0;
}
```

输出效果如下。其中 PUSH 属于同一层的是 for 循环的迭代，缩进的则是递归中的 for 迭代。以"abc"全排列为例，最右的三个 PUSH 相当于 abc，bac，cba 的三次变化。

```
PUSH: abc       交换:a a
    PUSH: bc    交换:b b
    ----->abc
    POP:  bc    换回:b b

    PUSH: cb    交换:b c
    ----->acb
    POP:  bc    换回:b c

POP:  abc       换回:a a

PUSH: bac       交换:a b
    PUSH: ac    交换:a a
    ----->bac
    POP:  ac    换回:a a

    PUSH: ca    交换:a c
    ----->bca
    POP:  ac    换回:a c

POP:  abc       换回:a b

PUSH: cba       交换:a c
    PUSH: ba    交换:b b
    ----->cba
    POP:  ba    换回:b b

    PUSH: ab    交换:b a
    ----->cab
    POP:  ba    换回:b a

POP:  abc       换回:a c
```

2.5.3 时间复杂度

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

$T(n) = \Theta(n!)$

2.6 循环赛日程表

2.6.1 问题描述

设有 $n=2^k$ 个运动员要进行羽毛球循环赛，现要设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其它 n-1 个选手各赛一次；

（2）每个选手一天只能比赛一次；

（3）循环赛一共需要进行 n-1 天。由于 $n=2^k$，显然 n 为偶数。

（1）如何分，即如何合理地进行问题的分解？

即将问题分解为若干个规模较小、相互独立、与原问题形式相同的子问题；

（2）如何治，即如何进行问题的求解？

1 求解各个子问题

2 合并

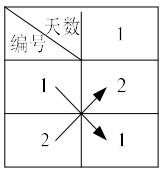
（3）问题的关键：发现循环赛日程表制定过程中存在的规律性。

2.6.2 算法分析

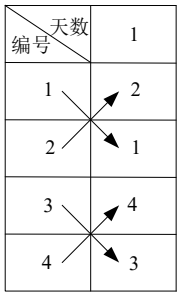
按分治策略，将所有的选手分为两半，则 n 个选手的循环赛日程表可以通过 n/2 个选手的循环赛日程表来决定。递归地用这种一分为二的策略对选手进行划分，直到只剩下两个选手时，循环赛日程表的制定就变得很简单。这时只要让这两个选手进行比赛就可以了。

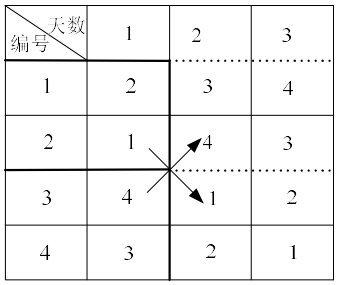
分治策略是自顶向下的，但是循环赛日程表的求解过程却是自底向上的迭代过程。求解过程中，把 2^k 个选手比赛日程问题分解成依次求解 2^1、2^2、…、2^k 个选手的比赛日程问题，换言之，2^k 个选手的比赛日程是在 2^{k-1} 个选手的比赛日程的基础上通过迭代的方法求得的。

第一步：$n=2^1$ 个选手的比赛日程表的制定。

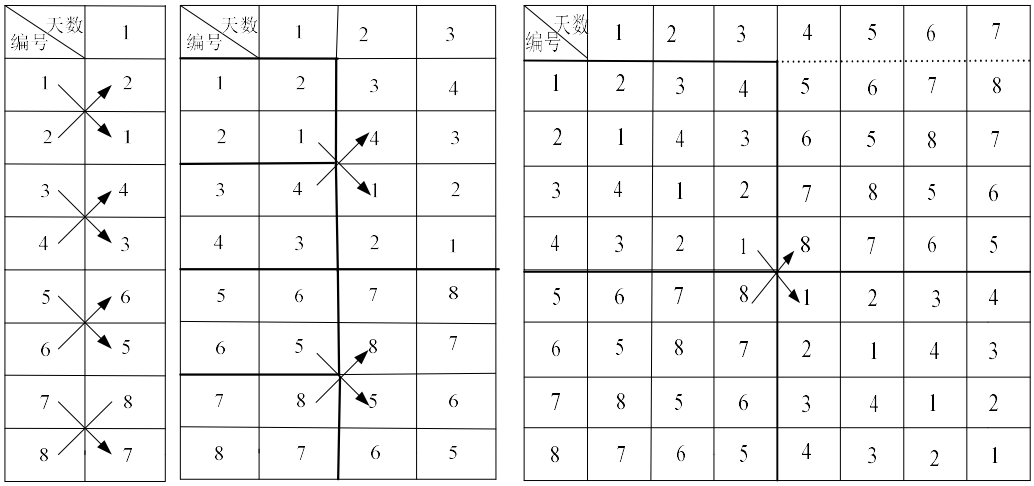


第二步：$n=2^2$ 个选手的比赛日程表的制定





第三步： $n=2^3$ 个选手的比赛日程表的制定



按分治策略，将所有的选手分为两半，n 个选手的比赛日程表就可以通过为 n/2 个选手设计的比赛日程表来决定。递归地对选手进行分割，直到只剩下 2 个选手时，循环赛日程表的制定就变得很简单。这时只要让这 2 个选手进行比赛就可以了。

下面的代码具体实现了循环赛日程表的全过程。

```
#include <mem.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

//---------------------------------------------------------------------------
const int N = 8;
//---------------------------------------------------------------------------
//动态创建二维数组，n 代表行，c 代表列
template<typename T>
T **crt2d(int n,int c,T defval) {
  T **p = new T *[n];
       for(int i=0;i<n;i++) {
    p[i]=new int[c];
    for(int j=0;j<c;j++)  p[i][j] = defval;
  }
  return p;
}

template<typename T>
void  disp2d(T **p,int n,int c,const char *fmt="%8d")  {
  for(int i=0;i<n;i++) {
     for(int j=0;j<c;j++)  printf(fmt,p[i][j]);
     putchar('\n');
  }
}

void disp(int **tbl,int n,char *info){
  puts(info);
  disp2d(tbl,n,n,"%3d");
  printf("\n");
}
//---------------------------------------------------------------------------
void arrange(int **a,int start,int end){
  int m = end/2;
  if(end>=4){
     arrange(a,start,m);
     arrange(a,start+m,m);
  }

  // 复制左下角
```

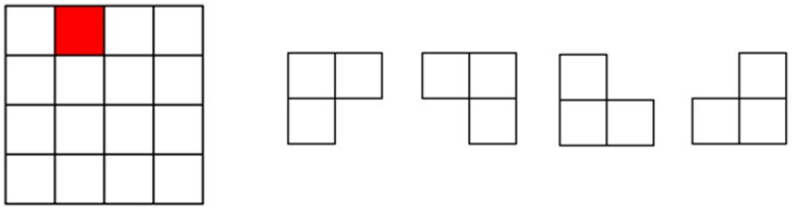
```
  for(int i=m;i<end;i++){
    for(int j=start;j<start+m;j++){
       a[i][j]=a[i-m][j+m];
    }
  }
  disp(a,N,"复制左下角:");

  // 复制右下角
  for(int i=m;i<end;i++){
    for(int j=start+m;j<start+end;j++){
      a[i][j]=a[i-m][j-m];
    }
  }
  disp(a,N,"复制右下角:");
}
//--------------------------------------------------------------------------
int main(){
  int **tbl= crt2d(N,N,0);

  for(int i=0;i<N;i++)  tbl[0][i] = i+1;

  arrange(tbl,0,N);
  disp(tbl,N,"最终输出:");

  system("pause");
  return 0;
}
//--------------------------------------------------------------------------
```

2.7 棋盘覆盖

2.7.1 问题描述

在一个 $2^k \times 2^k$ 个方格组成的棋盘中，如果有一个方格与其它方格不同，称称该方格为一特殊方格，且称该棋盘为一特殊棋盘。显然，特殊方格在棋盘上出现的位置有 4^k 种情形。所以对于任何 k>0，有 4^k 种不同的特殊棋盘。

棋盘覆盖问题是一种编程问题。即在一个 $2^k \times 2^k$ 个方格组成的棋盘中，任意给定一个特殊方格，用一种方案实现对除该特殊方格外的棋盘实现全覆盖。覆盖时，要用图示的 4 种不同形态的 L 型骨牌覆盖除给定的特殊棋盘上的特殊方格以外的所有方格，且任何 2 个 L 型骨牌不得重叠覆盖。可以知道，在 $2^k \times 2^k$ 棋盘覆盖中，用到的 L 型骨牌数恰为$(4^k - 1)/3$。



用分治策略解决棋盘覆盖问题时先将棋盘分成 4 个大小相等的子棋盘，如果特殊方块在子棋盘就递归该子棋盘；如果特殊方块不在子棋盘，就假设某方块为特殊方块，同样递归下去，直到全覆盖。大致的过程如下：

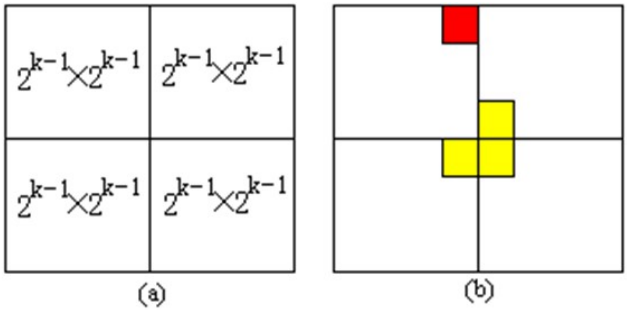
左上角的子棋盘若不存在特殊方格，则将该子棋盘右下角的那个方格假设为特殊方格；

右上角的子棋盘若不存在特殊方格，则将该子棋盘左下角的那个方格假设为特殊方格；

左下角的子棋盘若不存在特殊方格，则将该子棋盘右上角的那个方格假设为特殊方格；

右下角的子棋盘若不存在特殊方格：则将该子棋盘左上角的那个方格假设为特殊方格。

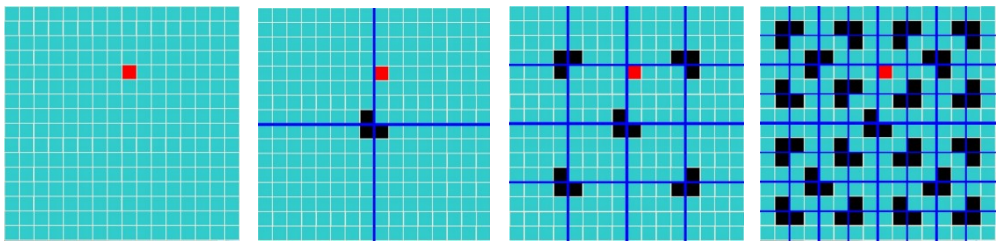
当 k>0 时，$2^k \times 2^k$ 棋盘可以分割为 4 个 $2^{k-1} \times 2^{k-1}$ 子棋盘，如下图所示。



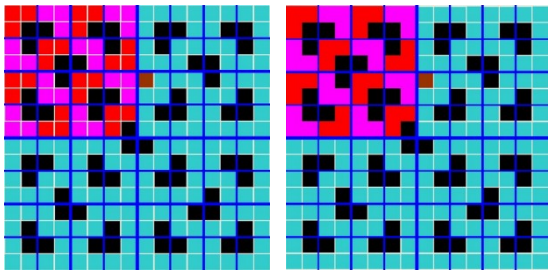
特殊方格必位于 4 个子棋盘之一中，其余 3 个子棋盘中无特殊方格。为了将这 3 个无特殊方格的子棋盘转化为特殊棋盘，可以用一个 L 型骨牌覆盖这 3 个较小棋盘的会合处，从而将原问题转化为 4 个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘 1×1。

2.7.2 算法分析

棋盘覆盖分治策略的覆盖过程大致可以通过下面步骤演示。



第一次　　第二次分割　　第三次分割　　第四次分割成 2X2 棋盘



子棋盘覆盖结果

下面的代码具体实现了棋盘覆盖的全过程。代码中的 tr，tc 表示棋盘的左上角的坐标即行号和列号，dr，dc 表示特殊方格的行号和列号，size 表示棋盘的大小。其中行列号从 0 开

始，size 是 2^n。

程序代码变量说明


```
#include <mem.h>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

int   tile  = 0;                    //棋牌编号
int   zsize = 4;                    //棋盘大小
int   **board = NULL;   //定义指向指针的指针用于动态的创建用于存储骨牌号的数组
//-----------------------------------------------------------------------------
//动态创建二维数组，n 代表行，c 代表列
template<typename T>
T **crt2d(int n,int c,T defval) {
  T **p = new T *[n];
     for(int i=0;i<n;i++) {
    p[i]=new int[c];
    for(int j=0;j<c;j++)  p[i][j] = defval;
  }
  return p;
}

template<typename T>
void  disp2d(T **p,int n,int c)  {
     for(int i=0;i<n;i++) {
    for(int j=0;j<c;j++)  printf("%8d",p[i][j]);
    putchar('\n');
  }
}
//-----------------------------------------------------------------------------
void  dispboard(int tt,int tr,int tc,int dr,int dc,int size) {
  printf("=========牌号:%d 棋盘大小:%d  棋盘坐标[%d,%d],特殊方块[%d,%d]\n",tt,size,tr,tc,dr,dc);
     disp2d(board,zsize,zsize);
  printf("\n");
}
//-----------------------------------------------------------------------------
void chessBoard(int tt,int tr, int tc, int dr, int dc, int size) {
  dispboard(tt,tr,tc,dr,dc,size);    getchar();
  if(1==size) {    printf("*********递归到底了。\n");    getchar(); return;  }

  int t = tile++;
  int s = size/2;

  if(dr < tr+s && dc < tc+s)  {
    printf("  左上\n");
    chessBoard(t,tr,tc,dr,dc,s);
  }
  else  {
    printf("非左上\n");    board[tr+s-1][tc+s-1] = t;
    chessBoard(t,tr,tc,tr+s-1,tc+s-1,s);
  }

  if(dr < tr+s && dc >= tc+s ) {
    printf("  右上\n");
    chessBoard(t,tr,tc+s,dr,dc,s);
  }
  else  {
    printf("非右上\n");    board[tr+s-1][tc+s] = t;
    chessBoard(t,tr,tc+s,tr+s-1,tc+s,s);
  }

  if(dr >= tr+s && dc < tc+s)   {
```

```
    printf("  左下\n");
    chessBoard(t,tr+s,tc,dr,dc,s);
  }
  else  {
    printf("非左下\n");    board[tr+s][tc+s-1] = t;
    chessBoard(t,tr+s,tc,tr+s,tc+s-1,s);
  }

  if(dr >= tr+s && dc >= tc+s)  {
    printf("  右下\n");
    chessBoard(t,tr+s,tc+s,dr,dc,s);
  }
  else  {
    printf("非右下\n");    board[tr+s][tc+s] = t;
    chessBoard(t,tr+s,tc+s,tr+s,tc+s,s);
  }
}
//---------------------------------------------------------------------------
int main()  {
  //定义棋盘的左上角方格、特殊方格的行号和列号以及棋盘大小
     int tx=0,ty=0,dx=0,dy=0;

  board = crt2d<int>(zsize,zsize,99);
     board[dx][dy]=0;
     chessBoard(tile,tx,ty,dx,dy,zsize);

     system("pause");
     return 0;
}
```

2.7.3 时间复杂度

$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

T(n)=O(4^k) 渐进意义下的最优算法

2.8 本章小结

本章主要介绍了分治策略，概述了递归的基本概念，比较详细的讲解了全排列、循环赛日程表以及棋盘覆盖等三个问题。重点需要理解分治策略的适用条件以及实现方法。对于棋盘覆盖的例子要特别理解如何改变不符合分治策略条件下的解决问题的思路。