如何求n个数全排列和所有子集

一、回溯法：深度优先搜素

简单概述

回溯法思路的简单描述是：把问题的解空间转化成了图或者树的结构表示，然后使用深度优先搜索策略进行遍历，遍历的过程中记录和寻找所有可行解或者最优解。

基本思想类同于：

- 图的深度优先搜索
- 二叉树的后序遍历
    - 分支限界法：广度优先搜索。
    - 思想类同于：图的广度优先遍历、二叉树的层序遍历。

详细描述

详细的描述则为：

回溯法按深度优先策略搜索问题的解空间树。首先从根节点出发搜索解空间树，当算法搜索至解空间树的某一节点时，先利用剪枝函数判断该节点是否可行（即能得到问题的解）。如果不可行，则跳过对该节点为根的子树的搜索，逐层向其祖先节点回溯；否则，进入该子树，继续按深度优先策略搜索。

回溯法的基本行为是搜索，搜索过程使用剪枝函数来为了避免无效的搜索。剪枝函数包括两类：

1. 使用约束函数，剪去不满足约束条件的路径。
2. 使用限界函数，剪去不能得到最优解的路径。

问题的关键在于如何定义问题的解空间，转化成树（即解空间树）。解空间树分为两种：子集树和排列树。两种在算法结构和思路上大体相同。

回溯法应用

当问题是要求满足某种性质（约束条件）的所有解或最优解时，往往使用回溯法。

它有“通用解题法”之美誉。

二、回溯法实现：递归和递推（迭代）

回溯法的实现方法有两种：递归和递推（也称迭代）。一般来说，一个问题两种方法都可以实现，只是在算法效率和设计复杂度上有区别。

类比于图深度遍历的递归实现和非递归（递推）实现。

递归

思路简单，设计容易，但效率低，其设计范式如下：

针对N叉树的递归回溯方法

```
void backtrack (int t)
{
if (t>n) output(x); //叶子节点，输出结果，x是可行解
else
 for i = 1 to k//当前节点的所有子节点
{
 x[t]=value(i); //每个子节点的值赋值给x
 //满足约束条件和限界条件
if (constraint(t)&&bound(t))
backtrack(t+1);  //递归下一层
}
```


递推

算法设计相对复杂，但效率高。

```
//针对N叉树的迭代回溯方法
void iterativeBacktrack ()
{
   int t=1;
   while (t>0) {
        if(ExistSubNode(t)) //当前节点的存在子节点
        {
           for i = 1 to k  //遍历当前节点的所有子节点
            {
                x[t]=value(i);//每个子节点的值赋值给x
                if (constraint(t)&&bound(t))//满足约束条件和限界条件
                {
                    //solution表示在节点t处得到了一个解
```

```
                    if (solution(t)) output(x);//得到问题的一个可行
解，输出
                    else t++;//没有得到解，继续向下搜索
                }
            }
        }
        else //不存在子节点，返回上一层
        {
            t--;
        }
    }
}
```

三、经典问题

1. 装载问题
2. 0-1背包问题
3. 旅行售货员问题
4. 八皇后问题
5. 迷宫问题
6. 图的m着色问题

0-1背包问题

问题：给定n种物品和一背包。物品i的重量是wi，其价值为pi，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

分析：问题是n个物品中选择部分物品，可知，问题的解空间是子集树。比如物品数目n=3时，其解空间树如下图，边为1代表选择该物品，边为0代表不选择该物品。使用x[i]表示物品i是否放入背包，x[i]=0表示不放，x[i]=1表示放入。回溯搜索过程，如果来到了叶子节点，表示一条搜索路径结束，如果该路径上存在更优的解，则保存下来。如果不是叶子节点，是中点的节点（如B），就遍历其子节点（D和E），如果子节点满足剪枝条件，就继续回溯搜索子节点。

代码如下:

```
#include <stdio.h>
#define N 3           //物品的数量
#define C 16          //背包的容量
```

```
int w[N]={10,8,5};  //每个物品的重量
int v[N]={5,4,1};   //每个物品的价值
int x[N]={0,0,0};   //x[i]=1代表物品i放入背包，0代表不放入

int CurWeight = 0;  //当前放入背包的物品总重量
int CurValue = 0;   //当前放入背包的物品总价值

int BestValue = 0;  //最优值；当前的最大价值，初始化为0
int BestX[N];       //最优解；BestX[i]=1代表物品i放入背包，0代表不放入

//t = 0 to N-1
void backtrack(int t)
{
   //叶子节点，输出结果
    if(t>N-1)
   {
        //如果找到了一个更优的解
        if(CurValue>BestValue)
        {
           //保存更优的值和解
            BestValue = CurValue;
           for(int i=0;i<N;++i) BestX[i] = x[i];
       }
   }
   else
    {
       //遍历当前节点的子节点：0 不放入背包，1放入背包
        for(int i=0;i<=1;++i)
       {
           x[t]=i;

           if(i==0) //不放入背包
           {
               backtrack(t+1);
           }
           else //放入背包
           {
              //约束条件：放的下
               if((CurWeight+w[t])<=C)
                {
                  CurWeight += w[t];
                   CurValue += v[t];
                    backtrack(t+1);
                    CurWeight -= w[t];
                    CurValue -= v[t];
                }
            }
       }
        //PS:上述代码为了更符合递归回溯的范式，并不够简洁
    }
}
```

```
int main(int argc, char* argv[])
{
   backtrack(0);

    printf("最优值：%d\n",BestValue);

   for(int i=0;i<N;i++)
    {
       printf("最优解：%-3d",BestX[i]);
    }
    return 0;
}
```

详细描述N皇后问题

问题：在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

N皇后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

分析：从n×n个格子中选择n个格子摆放皇后。可见解空间树为子集树。

使用Board[N][N]来表示棋盘，Board[i][j]=0 表示(I,j)位置为空，Board[i][j]=1 表示(I,j)位置摆放有一个皇后。

全局变量way表示总共的摆放方法数目。

使用Queen(t)来摆放第t个皇后。Queen(t) 函数符合子集树时的递归回溯范式。当t>N时，说明所有皇后都已经摆 放完成，这是一个可行的摆放方法，输出结果；否则，遍历棋盘，找皇后t所有可行的摆放位置，Feasible(i,j) 判断皇后t能否摆放在位置(i,j)处，如果可以摆放则继续递归摆放皇后t+1，如果不能摆放，则判断下一个位置。

Feasible(row,col)函数首先判断位置(row,col)是否合法，继而判断(row,col)处是否已有皇后，有则冲突，返回0，无则继续判断行、列、斜方向是否冲突。斜方向分为左上角、左下角、右上角、右下角四个方向，每次从（row,col）向四个方向延伸一个格子，判断是否冲突。如果所有方向都没有冲突，则返回1，表示此位置可以摆放一个皇后。

```
/*************************************************************
********
 * 名  称：NQueen.cpp
 * 功  能：回溯算法实例：N皇后问题
 * 作  者：JarvisChu
 * 时  间：2013-11-13
 *************************************************************
********/
```

```c
#include <stdio.h>

#define N 8

int Board[N][N];  //棋盘 0表示空白 1表示有皇后
int way;//摆放的方法数


//判断能否在(x,y)的位置摆放一个皇后；0不可以，1可以
int Feasible(int row,int col)
{
    //位置不合法
    if(row>N || row<0 || col >N || col<0)
        return 0;

    //该位置已经有皇后了，不能
    if(Board[row][col] != 0)
    {   //在行列冲突判断中也包含了该判断，单独提出来为了提高效率
        return 0;
   }

    ////////////////////////////////////////////////
    //下面判断是否和已有的冲突

    //行和列是否冲突
    for(int i=0;i<N;++i)
    {
        if(Board[row][i] != 0 || Board[i][col]!=0)
           return 0;
    }

   //斜线方向冲突

    for(int i=1;i<N;++i)
    {
     /* i表示从当前点(row,col)向四个斜方向扩展的长度

        左上角 \   / 右上角    i=2
         \/            i=1
         /\            i=1
       左下角 /   \ 右下角    i=2
     */
        //左上角
       if((row-i)>=0 && (col-i)>=0)     //位置合法
        {
            if(Board[row-i][col-i] != 0)//此处已有皇后，冲突
               return 0;
        }

        //左下角
       if((row+i)<N && (col-i)>=0)
       {
```

```
            if(Board[row+i][col-i] != 0)
               return 0;
        }

        //右上角
        if((row-i)>=0 && (col+i)<N)
        {
            if(Board[row-i][col+i] != 0)
                return 0;
        }

        //右下角
       if((row+i)<N && (col+i)<N)
        {
            if(Board[row+i][col+i] != 0)
               return 0;
        }
   }

  return 1; //不会发生冲突，返回1
}


//摆放第t个皇后 ；从1开始
void Queen(int t)
{
    //摆放完成，输出结果
    if(t>N)
   {
        way++;
        /*如果N较大，输出结果会很慢；N较小时，可以用下面代码输出结果
        for(int i=0;i<N;++i){
            for(int j=0;j<N;++j)
               printf("%-3d",Board[i][j]);
            printf("\n");
        }
        printf("\n----------------------\n\n");
        */
    }
    else
    {
      for(int i=0;i<N;++i)
        {
            for(int j=0;j<N;++j)
            {
                //（i,j）位置可以摆放皇后，不冲突
               if(Feasible(i,j))
               {
                   Board[i][j] = 1;  //摆放皇后t
                    Queen(t+1);        //递归摆放皇后t+1
                    Board[i][j] = 0;  //恢复
               }
```

```
            }
        }
    }
}

//返回num的阶乘,num!
int factorial(int num)
{
    if(num==0 || num==1)
       return 1;
    return num*factorial(num-1);
}


int main(int argc, char* argv[])
{
    //初始化
    for(int i=0;i<N;++i)
    {
        for(int j=0;j<N;++j)
        {
            Board[i][j]=0;
        }
    }

    way = 0;

    Queen(1);   //从第1个皇后开始摆放

    //如果每个皇后都不同
    printf("考虑每个皇后都不同，摆放方法：%d\n",way);//N=8时，
way=3709440 种

    //如果每个皇后都一样，那么需要除以 N! 出去重复的答案（因为相同，则每个皇
后可任意调换位置）
    printf("考虑每个皇后都不同，摆放方法：
%d\n",way/factorial(N));//N=8时, way=3709440/8! = 92种

    return 0;
}
```


回溯法是设计递归的一种常用方法，它的求解过程实质上就是一个先序遍历一棵”状态树”的过程,只是这棵树不是遍历前预先建立的而是隐含在遍历过程中的。

四、n个数的全排列

方法一：递归回溯

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。从集合中依次选出每一个元素，作为排列的第一个元素，然后对剩余的元素进行全排列，如此递归处理，从而得到所有元素的全排列。

以对字符串abc进行全排列为例，我们可以这么做：以abc为例，

固定a，求后面bc的排列：abc，acb，求好后，a和b交换，得到bac。

固定b，求后面ac的排列：bac，bca，求好后，c放到第一位置，得到cba。

固定c，求后面ba的排列：cba，cab。

这个思想和回溯法比较吻合。代码可如下编写所示：

```
// 回溯法搜索全排列树
  #include<stdio.h>
  #define M 20
  int n;
  int a[M];
  int cnt = 0;// 记录全排列个数

  void swap(int *a, int *b)//交换a,b
  {
      char t;
      t = *a;
      *a = *b;
      *b = t;
  }

  void dfs(int cur)
  {
       int i;
       if(cur == n)// 找到 输出全排列
          {
              ++cnt;
              for(i=0; i<n; i++)
                 printf("%d ", a[i]);
                 printf("\n");
          }
       else
       {
                // 将集合中的所有元素分别与第一个交换，这样就总是在
               // 处理剩下元素的全排列(即用递归)
               for(i=cur; i<n; i++)
                 {
                     swap(&a[cur], &a[i]);
                     dfs(cur+1);
                     swap(&a[cur], &a[i]);//回溯
                 }
        }
```

```
}


int main()
{
    while(scanf("%d", &n) != EOF)
    {
          for(int i=0; i<n; i++)
             a[i] = i+1;// 假设集合S为:1 2 3 ... n
          cnt = 0;
          dfs(0);
          printf("count:%d\n", cnt);
     }
    return 0;
}
```

方法二：回溯标记访问

利用一个vis数组标识每个元素是否已经被访问，代码如下：

```
#include <stdio.h>

   int a[10];
   bool vis[10];
   int n;//排列个数 n
   void dfs(int dep)  //打印所有的全排列，穷举每一种方案
   {
          if(dep == n)
          {
               for(int i = 0; i < n; i++)
              {
                  printf("%d ",a[i]);
              }
               printf("\n");
               return ;
           }
          for(int i = 0; i < n; i++)// 找一个最小的未标记的数字，保证
了字典序最小
             {
                 if(!vis[i])
                   {
                        a[dep] = i+1;
                        vis[i] = true;// 找到了就标记掉，继续下一层
                        dfs(dep + 1);
                        vis[i] = false;
                   }
             }
       }
   int main()
```

```
{
        while(scanf("%d",&n) != EOF)
        {
              dfs(0);
          }
        return 0;
}
```

方法三：利用图的深度优先遍历

要求按照字典序输出。这是最典型的深搜问题。我们可以把N个数两两建立无向边（即任意两个结点之间都有边，也就是一个N个结点的完全图），然后对每个点作为起点，分别做一次深度优先遍历，当所有点都已经标记时输出当前的遍历路径，就是其中一个排列，这里需要注意，回溯的时候需要将原先标记的点的标记取消，否则只能输出一个排列。如果要按照字典序，则需要在遍历的时候保证每次遍历都是按照结点从小到大的方式进行遍历的。

如下图所有：

代码如下:


```
#include<iostream>
  #include<vector>

  using namespace std;

  #define MAX 20

  vector<int>index;
  int visited[MAX]={0};

   void dfs(int arr[],int len,int num,int k)//len代表数组长度，num
表示当前求到第几个，k表示多少数的全排列
   {
        if(k==num){
              for(int i=0;i<num;i++){
                    cout<<arr[index[i]]<<" ";
                }
              cout<<endl;
              return;
        }
        for(int i=0;i<len;i++){
              if(visited[i]==0){
                    index.push_back(i);
                    visited[i] = 1;
                    num++;
                    dfs(arr,len,num,k);
                    index.pop_back();
```

```
                    num--;
                    visited[i]=0;
                }
            }
    }

    int main()
    {
        int arr[]={1,2,3,4};
        dfs(arr,4,0,3);//求3个数的全排列
        system("pause");
        return 0;
    }
```

方法四：利用next_permutation函数

C++ algorithm中有全排列函数，着实简单，而且包含元素可重复情况。

```
#include <iostream>
  #include <cstdio>
  #include <algorithm>
  #define maxN 4

  using namespace std;

  int main()
  {
        int p[maxN] = {-1};
        for(int i=0; i<maxN; ++i)
          {
                p[i] = i+1;
          }

        do
          {
                for(int i=0; i<maxN; ++i)
                  {
                        printf("%d\t",p[i]);
                  }
                printf("\n");
          }while(next_permutation(p,p+maxN) );
        return 0;
  }
```

方法五：非递归算法

```
#include"stdio.h"
  void swap(int &a, int &b)
  {
        int temp;
        temp = a;
        a = b;
        b = temp;
    }


   /*
   根据当前的排列p，计算下一个排列。
   原则是从1234->4321，若p已经是最后一个排列，传回false，否则传回true。
   p是一个n维向量。
   */


   bool nextPermutation(int *p, int n)
   {
        int last=n-1;
        int i,j,k;
        //从后向前查找，看有没有后面的数大于前面的数的情况，若有则停在后一个
数的位置。
        i=last;
        while(i>0&&p[i]<p[i-1])
            i--;
        //若没有后面的数大于前面的数的情况，说明已经到了最后一个排列，返回
false。
        if(i==0)
              return false;
        //从后查到i，查找大于p[i - 1]的最小的数，记入k
        k=i;
        for(j=last;j>=i;j--)
              if(p[j]>p[i-1]&&p[j]<p[k])
                  k =j;
            //交换p[k]和p[i - 1]
            swap(p[k],p[i-1]);
            //倒置p[last]到p[i]
            for (j =last,k =i;j>k;j--,k++)
                  swap(p[j],p[k]);
            return true;
    }
   //显示一个排列
   void showPermutation(int *p, int n)
   {
         for(int i=0;i<n;i++)
             printf("%d ",p[i]);
         printf("\n");
   }
   int main(int argc, char *argv[])
   {
```

```
            int n;
            int *p;
            scanf("%d",&n);
            p = new int[n];
            for (int i = 0; i < n; i++)
                  p[i] = i + 1;
            showPermutation(p, n);
            while(nextPermutation(p, n))
              {
                  showPermutation(p, n);
              }
            //delete[] p;
            return 0;
      }
```

五、n个数的所有子集

非递归不去重

思路分析：n个元素的子集共有2^n个，其中包括空集。

1. 假设有3个元素 { a, b, c } ，那么此时有 2^3 个子集，即8个子集。
2. 因为有8个子集，而且包括空集，注意7对应的二进制形式为111，并且二进制数刚好3位；所以（ 000 ~ 111 ）可分别对应一个子集，若该位为1，则表示该元素出现在子集中，若该位为0，则表示该元素不出现在子集中。
3. 注意：001中的1在低位，对应的是a，即数组中的首个元素。
4. 举例：

111表示子集abc；

110表示子集bc；

101表示子集ac；

100表示子集c；

011表示子集ab

010表示子集b；

001表示子集a；

000则表示空集；

具体实现如下：

```
#include <iostream>
using namespace std;
typedef unsigned long DWORD; // DWORD 即double world，双字节。
// 求arr的子集，arr共有n个元素的所有子集，时间复杂度为2^n
void print_allSubSet(int arr[],int n) {
DWORD i,j,total,mask;
if (n > 30) {
 printf("%d is too big\n",n); return ;
}
total= (1 << n);
// 1 << n 即把1的二进制形式，左移n位；因为2^n不好表达，所以采用移位的方式；
(n个元素共有2^n个子集，包括空集)
for (j=0; j < total; j++) // 每循环一次选出一个子集 {
  printf("{ ");
i = 0; // 每一次循环，i都重新置0；对应原数组中的第一个数字。
mask = j; // 序号j对应的是第(j+1)个子集。
while (mask > 0) // 通过移位的方式，直至mask的二进制形式中，所有位都为0。
{
 if (mask & 1)
 // 若mask的二进制形式的最后一位非0，输出该位对应的数字。
  printf("%d ", arr[i]); mask >>= 1; // mask右移一位
  i++;
}
printf("}\n");
}
}
int main(int argc, const char * argv[]) {
int n=3; //求3个元素的所有子集。
int arr[32]; int i;
for (i=0;i<n;i++) arr[i]=i+1;
//arr表示一个集合，共有n个元素
print_allSubSet(arr, n);
return 0;
}
```


递归方法，不去重

思路分析：同上。

此处，我们添加一个标记数组tag，用于记录子集中对应的元素是否出现。每输出一个子集，结束当前步骤，并进入下一步，直至递归完毕。

具体实现如下：

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

```
// 递归
void allSubSet(int arr[], int tag[], int n) {
if(n == 3) {
cout<< "{ ";
for(int i = 0; i < 3; i++)
if(tag[i] == 1) cout << arr[i] << ' ';
cout<< "}" << endl; return;
}
tag[n] = 0;
allSubSet(arr, tag, n+1);
tag[n] = 1;
allSubSet(arr, tag, n+1);
}
int main(int argc, const char * argv[]) {
int a[3]={1,2,3};
int tag[3];
allSubSet(a,tag,0);
return 0;
}
```