# 生成组合数C(n,k)

## 回溯法之二叉子集树

这是比较笨的一种方法，在最原始的n层二叉集树中，共有2的n次方个叶节点，也就是有2的n次方种可能性，在这2的n次方种可能性中包含了C(n,0),C(n,1),C(n,2)…..C(n,k) (0<=k<=n)这n种组合的所有情况。因此当我使用回溯法来生成C(n,k)的时候，其实是一个对这n层二叉树的一个剪枝过程，我只选取C(n,k)的组合数，我现在的实现思路是设置一个countK计数器来记录已经选取的元素的个数，一旦元素的个数达到k的话，就停止进一步的递归。

这种算法是非常不好的，因为n层二叉树中包含很多我根本不需要的组合的情况，这样的成本太大。

## 枚举法

#include<iostream>

#include<cstdio>

Int main(){

Int n=5;

Int m=0;

For(int i=1;i<=n-m+1;i++)

For(int j=i+1;j<=n-m;j++)

For(int k=j+1;k<=n;k++)

Printf(“%d%d%d\n”,x,y,z);

Return 0;

}

这样它就生成了所有以1打头的组合

所有以2打头的组合

所有以3打头的组合

点评：这种方法是非常不灵活的，因为它只能够计算C(n,3),因为只有三层循环，所以它是不灵活的。下面的回溯法就非常地灵活，它通过k来控制递归的层次，只有就可以求任意C(n,k)了，显然下面的算法是对这个算法的一个提升。

## 递归法

设函数comb(int n,k)是从n个数中取k个数的组合，每个组合的第一个数可以是{n,n-1,n-2….k}中的一个，当组合的第一个数选定后，其后的数字就是从余下的n-1个数中选取k-1个数，这样就将问题转换为了comb(n-1,k-1)。使用一个全局数组a来存放生成的组合。需要特别注意的是：每个组合的第一个数可以是{n,n-1,n-2….k}中的一个。函数确定了组合的一个数字后，如果还没有确定组合的其余元素，则继续递归。

程序如下：

#include<iostream>

#include<cstdio>

using namespace std;

const int MAX=10

int a[MAX];

void comb(int n,intk){

int i,int j;

for(int i=n;i>=k;i--){ //每个组合的第一个数可以是 n,n-1,n-2….,k

a[k]=i;

if(k>1){

comb(n-1,k-1);

++j;

}

else{

for(int j=a[0];j>0;j--)

printf(“%d”,a[j]);

printf(“\n”);

}

}

}

点评：对比这种方法和第二种枚举法，可以帮助对递归方法的理解。

显然两者算法的计算方向是完全不相同的，第一种算法是从低向上的计算，第二种算法是从顶向下的计算。

# 生成排列数的算法

## 生成全排列

### 回溯法

### Steinhaus-Johnson-Trotter算法

Steinhaus-Johnson-Trotter算法是一种基于最小变换的全排列生成算法，对于排列a[1...n]，该算法通过将a[i]，与a[i-1]（或a[i+1]）进行交换，生成下一个排列，直到所有排列生成完毕为止，这样，当前排列与其后继排列只是两个相邻位置的元素发生了调换。当然，为了防止重复生成某一个排列，算法并非随意调换某两个元素之间的位置，其生成全排列的具体规则如下。

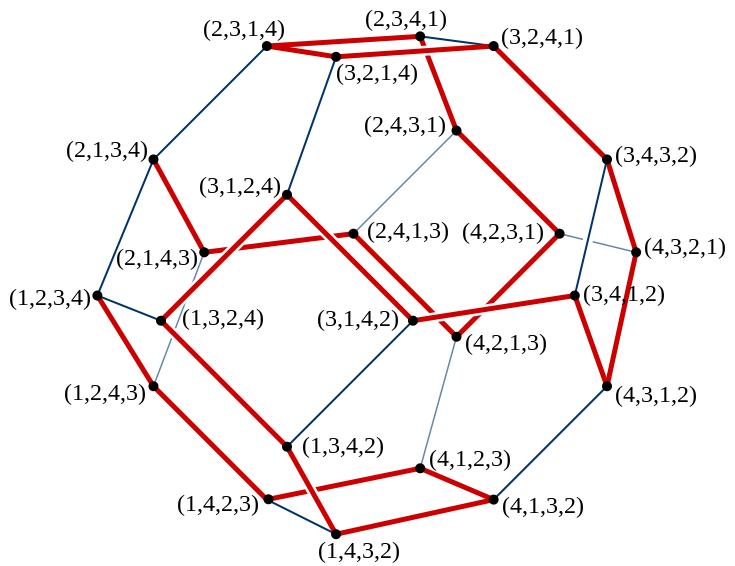
* 首先，以字典序最小的排列起始，并且为该排列的每个元素赋予一个移动方向，初始所有元素的移动方向都向左。
* 在排列中查找这样的元素，该元素按照其对应的移动方向移动，可以移动到一个合法位置，且移动方向的元素小于该元素，在所有满足条件的元素中，找到其中的最大者。
* 将该元素与其移动方向所对应的元素交换位置。
* 对于排列中，所有元素值大于该元素的元素，反转其移动方向。

这里有几个概念需要说明一下，所谓合法位置，是指该元素按照其移动方向移动，不会移动到排列数组之外，例如对于<4，<1，<2，<3，此时对于元素4，如果继续向左移动，就会超过数组范围，所以4的下一个移动位置是非法位置。而且，所有元素，都只能向比自己小的元素的方向移动，如上面例子中的元素2，3，而元素1是不能够移动到元素4的位置的。每次移动，都要对可以移动的所有元素中的最大者进行操作，上例中元素1，4不能移动，2，3都存在合法的移动方案，此时需要移动3，而不能移动2。合法移动之后，需要将所有大于移动元素的元素的移动方向反转，上例中的元素3移动后的结果是4>，1<，<3，<2，可以看到，元素4的移动方向改变了。再如此例子<2，<1，3>，4>，对于其中的元素2，4，其对应的下一个移动位置都是非法位置，而对于元素1，3，其下一个移动位置的元素，都比他们要大，对于该排列就找不到一个可以的移动方案，这说明该算法已经达到终态，全排列生成结束。下面是该算法的代码

**[cpp]** [view plain](http://blog.csdn.net/joylnwang/article/details/7074066) [copy](http://blog.csdn.net/joylnwang/article/details/7074066)

1. **inline** **int** SJTNext(unsigned **int**\* index, **size\_t** array\_size, **int**\* move)
2. {
3. unsigned **int** i, j, t;  //j记录的是待交换位置的元素的下标
5. //找到最大合法移动的元素索引
6. **for**(i = array\_size - 1, j = array\_size; i != UINT\_MAX; --i)
7. {
8. **if**(i + move[i] < array\_size && index[i] > index[i + move[i]])
9. {
10. **if**(j == array\_size)
11. {
12. j = i;
13. **continue**;
14. }
16. **if**(index[i] > index[j])
17. {
18. j = i;
19. }
20. }
21. }
23. //未发现合法的移动策略
24. **if**(j == array\_size)
25. {
26. **return** 1;
27. }
29. t = index[j];//要交换位置的元素
30. i = j + move[j];//发生交换的位置
31. swap(index, i, j);
32. swap(move, i, j);
34. //将所有比t大的元素的移动方向反转
35. **for**(i = 0; i < array\_size; ++i)
36. {
37. **if**(index[i] > t)
38. {
39. move[i] = -move[i];
40. }
41. }
43. **return** 0;
44. }
46. /\*
47. \* 基于最小变换的Steinhaus–Johnson–Trotter算法
48. \*/
49. **void** FullArray(**char**\* array, **size\_t** array\_size)
50. {
51. unsigned **int** index[array\_size];
52. **int** move[array\_size];
54. **for**(unsigned **int** i = 0; i < array\_size; ++i)
55. {
56. index[i] = i;
57. move[i] = -1;
58. }
60. ArrayPrint(array, array\_size, index);
62. **while**(!SJTNext(index, array\_size, move))
63. {
64. ArrayPrint(array, array\_size, index);
65. }
66. }

代码使用了一个伴随数组move标记对应位置元素的移动方向，在元素移动时，move数组中的对应元素也要相应移动。该算法从初始排列<1,<2,<3,<4开始，可以生成4元素的所有排列，直至最终排列<2，<1，3>，4>为止，其状态转移如下图所示，该图片来自于Wiki百科。



实际上该算法是Shimon Even对于Steinhaus-Johnson-Trotter三人提出的全排列生成算法的改进算法，在算法中实际上还有一个问题需要解决，就是对于给定的排列，如何判断其所有元素的移动方向，如果上面所谓终态的移动方向是<2，<1，3>，<4，那么这个状态就还存在可行的移动方案。Johnson(1963)给出了判断当前排列各元素移动方向的方法，对于排列中的每个元素，判断所有比该元素小的元素所生成序列的逆序数，如果逆序数为偶，则该元素的移动方向为向左，否则移动方向向右，我们用这条原则来看一下上面的终态2，1，3，4。对于元素1，没有比1小的元素，此时我们认为，空序列的逆序数为偶，所以元素1的移动方向向左；对于元素2，比2小的元素形成的序列为1，单元素序列的逆序数为偶，所以2的移动方向向左；对于元素3，小于3的元素组成的序列为21，逆序数为1，奇数，所以3的移动方向向右；对于元素4，对应序列为213，逆序数为奇数，所以4的移动方向向右。根据该规则就可以知道，给定某一排列，其对应元素的移动方向是确定的。

void SJT(int \*a, int n)

{

long len=count\_factorial(n);

int b[100] = {0};

int direction[100];

for (int i = 0; i < n; i++) direction[i] = -1;

int pos = n - 1;

for (int i = 0; i < len; i++){

permutation\_print(a, n);

if (direction[pos] == -1 && pos > 0){

int next\_pos = direction[pos] + pos;

swap(a[pos], a[next\_pos]);

swap(direction[pos], direction[next\_pos]);

pos = next\_pos;

}

else if (direction[pos] == 1 && pos < n - 1){

int next\_pos = direction[pos] + pos;

swap(a[pos], a[next\_pos]);

swap(direction[pos], direction[next\_pos]);

pos = next\_pos;

}

else{

int max\_pos = -1, max\_num = -1;

for (int j = 0; j < n; j++){

int next\_pos = j + direction[j];

if (next\_pos < 0 || next\_pos >= n) continue;

if (a[next\_pos] > a[j]) continue;

if (max\_pos == -1 || a[max\_pos] < a[j]){

max\_pos = j;

max\_num = a[max\_pos];

}

}

if (max\_pos == -1) break;

int next\_pos = max\_pos + direction[max\_pos];

swap(a[max\_pos], a[next\_pos]);

swap(direction[max\_pos], direction[next\_pos]);

for (int j = 0; j < n; j++){

if (a[j] > max\_num) direction[j] = -direction[j];

}

}

}

cout<<"Total 6:"<<len<<endl;

}