

作业2

202208010512 计科2205 刘志垚

算法实现题 2-2

2-2 马的 Hamilton 周游路线问题。

问题描述： 8×8 的国际象棋棋盘上的一只马，恰好走过除起点外的其他 63 个位置各一次，最后回到起点。这条路线称为马的一条 Hamilton 周游路线。对于给定的 $m \times n$ 的国际象棋棋盘， m 和 n 均为大于 5 的偶数，且 $|m-n| \leq 2$ ，试设计一个分治算法找出马的一条 Hamilton 周游路线。

算法设计：对于给定的偶数 $m, n \geq 6$ ，且 $|m-n| \leq 2$ ，计算 $m \times n$ 的国际象棋棋盘上马的一条 Hamilton 周游路线。

数据输入：由文件 input.txt 给出输入数据。第 1 行有两个正整数 m 和 n ，表示给定的国际象棋棋盘由 m 行，每行 n 个格子组成。

结果输出：将计算出的马的 Hamilton 周游路线用下面的两种表达方式输出到文件 output.txt。

第 1 种表达方式按照马步的次序给出马的 Hamilton 周游路线。马的每一步用所在的方格坐标 (x, y) 来表示。 x 表示行坐标，编号为 $0, 1, \dots, m-1$ ； y 表示列坐标，编号为 $0, 1, \dots, n-1$ 。起始方格为 $(0, 0)$ 。

第 2 种表达方式在棋盘的方格中标明马到达该方格的步数。 $(0, 0)$ 方格为起跳步，并标明为第 1 步。

1.思路

本题很容易用 dfs+回溯实现，但算法的复杂度较高。题目要求使用分治法，那么如何划分呢？

在 $n \times n$ 的国际象棋棋盘上的一只马，可按 8 个不同方向移动。定义 $n \times n$ 的国际象棋棋盘上的马步图为 $G=(K,E)$ 。棋盘上的每个方格对应于图 G 中的一个顶点， $V = \{(i, j) | 0 \leq i, j < n\}$ 。从一个顶点到另一个马步可跳达的顶点之间有一条边 $E = \{(u, v), (s, t) | \{|u-s|, |v-t|\} = \{1, 2\}\}$ 。图 G 有 n^2 个顶点和 $4n^2 - 12n + 8$ 条边。马的 Hamilton 周游路线问题即是图 G 的 Hamilton 回路问题。容易看出，当 n 为奇数时，该问题无解。事实上，由于马在棋盘上移动的方格是黑白相间的，如果有解，则走到的黑白格子数相同，因此棋盘格子总数应为偶数，然而 n 为奇数，此为矛盾。下面给出的算法可以证明，当 $n \geq 6$ 是偶数时，问题有解，而且可以用分治法在线性时间内构造出一个解。

$|m-n| \leq 2, m, n > 5$ ，所以说最小的解是 6×6 ，按照 $|m-n| \leq 2$ 规则可以继续扩展 $m \times m$ 、 $m \times (m-2)$ 、 $m \times (m+2)$... 当可以构造出 6 的 2 倍数也就是 12 的时候，结构化的解构造完毕。因为问题的可行域在 $|m-n| \leq 2, m, n > 5$ 范围内，当我们已经构造了结构化解，那么所有的解都能用结构化解构造出，这是分治法核心。

可以先求出： $6 \times 6, 6 \times 8, 8 \times 8, 8 \times 10, 10 \times 10, 10 \times 12$ 的结构化棋盘。

1	30	33	16	3	24
32	17	2	23	34	15
29	36	31	14	25	4
18	9	6	35	22	13
7	28	11	20	5	26
10	19	8	27	12	21

(a) 6×6 棋盘上的结构化 Hamilton 回路

1	10	31	40	21	14	29	38
32	41	2	11	30	39	22	13
9	48	33	20	15	12	37	28
42	3	44	47	6	25	18	23
45	8	5	34	19	16	27	36
4	43	46	7	26	35	24	17

(b) 6×8 棋盘上的结构化 Hamilton 回路

1	46	17	50	3	6	31	52
18	49	2	7	30	51	56	5
45	64	47	16	27	4	53	32
48	19	8	29	10	55	26	57
63	44	11	22	15	28	33	54
12	41	20	9	36	23	58	25
43	62	39	14	21	60	37	34
40	13	42	61	38	35	24	59

(c) 8×8 棋盘上的结构化 Hamilton 回路

1	46	37	66	3	48	35	68	5	8
38	65	2	47	36	67	4	7	34	69
45	80	39	24	49	18	31	52	9	6
64	23	44	21	30	15	50	19	70	33
79	40	25	14	17	20	53	32	51	10
26	63	22	43	54	29	16	73	58	71
41	78	61	28	13	76	59	56	11	74
62	27	42	77	60	55	12	75	72	57

(d) 8×10 棋盘上的结构化 Hamilton 回路

1	54	69	66	3	56	39	64	5	8
68	71	2	55	38	65	4	7	88	63
53	100	67	70	57	26	35	40	9	6
72	75	52	27	42	37	58	87	62	89
99	30	73	44	25	34	41	36	59	10
74	51	76	31	28	43	86	81	90	61
77	98	29	24	45	80	33	60	11	92
50	23	48	79	32	85	82	91	14	17
97	78	21	84	95	46	19	16	93	12
22	49	96	47	20	83	94	13	18	15

(e) 10×10 棋盘上的结构化 Hamilton 回路

1	4	119	100	65	6	69	102	71	8	75	104
118	99	2	5	68	101	42	7	28	103	72	9
3	120	97	64	41	66	25	70	39	74	105	76
98	117	48	67	62	43	40	27	60	29	10	73
93	96	63	44	47	26	61	24	33	38	77	106
116	51	94	49	20	23	46	37	30	59	34	11
95	92	115	52	45	54	21	32	35	80	107	78
114	89	50	19	22	85	36	55	58	31	12	81
91	18	87	112	53	16	57	110	83	14	79	108
88	113	90	17	86	111	84	15	56	109	82	13

(f) 10×12 棋盘上的结构化 Hamilton 回路

如何划分？

将棋盘尽可能平均地分割成4块。

1. 当 $m, n=4k$ 时，分割为2个 $2k$ ；
2. 当 $m, n=4k+2$ 时，分割为1个 $2k$ 和1个 $2k+2$ 。

两个原因，子问题必须是偶数，所以取模4，原问题是偶数所以只有 $4k$ 和 $4k+2$ 两种规模。

2.代码

```
#include <iostream>
#include <fstream>
using namespace std;
struct grid
{
    //表示坐标
    int x;
    int y;
};
class Knight{
public:
    knight(int m,int n);
    ~knight(){};
    void out0(int m,int n,ofstream &out);
    grid *b66,*b68,*b86,*b88,*b810,*b108,*b1010,*b1012,*b1210,link[20][20];
    int m,n;
    int pos(int x,int y,int col);
};
```

```

    void step(int m,int n,int a[20][20],grid *b);
    void build(int m,int n,int offx,int offy,int col,grid *b);
    void base0(int mm,int nn,int offx,int offy);
    bool comp(int mm,int nn,int offx,int offy);
};

knight::knight(int mm,int nn){
    int i,j,a[20][20];
    m=mm;
    n=nn;
    b66=new grid[36];b68=new grid[48];
    b86=new grid[48];b88=new grid[64];
    b810=new grid[80];b108=new grid[80];
    b1010=new grid[100];b1012=new grid[120];
    b1210=new grid[120];
    //cout<<"6*6"<<"\n";
    ifstream in0("66.txt",ios::in); //利用文件流读取数据
    ifstream in1("68.txt",ios::in); //利用文件流读取数据
    ifstream in2("88.txt",ios::in); //利用文件流读取数据
    ifstream in3("810.txt",ios::in); //利用文件流读取数据
    ifstream in4("1010.txt",ios::in); //利用文件流读取数据
    ifstream in5("1012.txt",ios::in); //利用文件流读取数据
    for(i=0;i<6;i++)
    {
        for(j=0;j<6;j++)
        {
            in0>>a[i][j];
        }
    }
    step(6,6,a,b66);
    //cout<<"6*8"<<"\n";
    for(i=0;i<6;i++)
    {
        for(j=0;j<8;j++)
        {
            in1>>a[i][j];
        }
    }
    step(6,8,a,b68);
    step(8,6,a,b86);
    //cout<<"8*8"<<"\n";
    for(i=0;i<8;i++)
    {
        for(j=0;j<8;j++)
        {
            in2>>a[i][j];
        }
    }
    step(8,8,a,b88);
    for(i=0;i<8;i++)
    {
        for(j=0;j<10;j++)
        {
            in3>>a[i][j];
        }
    }
    step(8,10,a,b810);

```

```

step(10,8,a,b108);
//cout<<"10*10"<<"\n";
for(i=0;i<10;i++)
{
    for(j=0;j<10;j++)
    {
        in4>>a[i][j];
    }
}
step(10,10,a,b1010);
for(i=0;i<10;i++)
{
    for(j=0;j<12;j++)
    {
        in5>>a[i][j];
    }
}
step(10,12,a,b1012);
step(12,10,a,b1210);
}
//将读入的基础棋盘的数据转换为网格数据
void Knight::step(int m,int n,int a[20][20],grid *b)
{
    int i,j,k=m*n;
    if(m<n)
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                int p=a[i][j]-1;
                b[p].x=i;
                b[p].y=j;
            }
        }
    }
    else
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                int p=a[j][i]-1;
                b[p].x=i;
                b[p].y=j;
            }
        }
    }
}
//分治法的主体部分
bool Knight::comp(int mm,int nn,int offx,int offy)
{
    int mm1,mm2,nn1,nn2;
    int x[8],y[8],p[8];
    if(mm%2||nn%2||mm-nn>2||nn-mm>2||mm<6||nn<6) return 1;
    if(mm<12||nn<12)

```

```

{
    base0(mm,nn,offx,offy);
    return 0;
}
mm1=mm/2;
if(mm%4>0)
{
    mm1--;
}
mm2=mm-mm1;
nn1=nn/2;
if(nn%4>0)
{
    nn1--;
}
nn2=nn-nn1;
//分割
comp(mm1,nn1,offx,offy); //左上角
comp(mm1,nn2,offx,offy+nn1); //右上角
comp(mm2,nn1,offx+mm1,offy); //左下角
comp(mm2,nn2,offx+mm1,offy+nn1); //右下角
//合并
x[0]=offx+mm1-1; y[0]=offy+nn1-3;
x[1]=x[0]-1;      y[1]=y[0]+2;
x[2]=x[1]-1;      y[2]=y[1]+2;
x[3]=x[2]+2;      y[3]=y[2]-1;
x[4]=x[3]+1;      y[4]=y[3]+2;
x[5]=x[4]+1;      y[5]=y[4]-2;
x[6]=x[5]+1;      y[6]=y[5]-2;
x[7]=x[6]-2;      y[7]=y[6]+1;
for(int i=0;i<8;i++)
{
    p[i]=pos(x[i],y[i],n);
}
for(int i=1;i<8;i+=2)
{
    int j1=(i+1)%8,j2=(i+2)%8;
    if(link[x[i]][y[i]].x==p[i-1])
        link[x[i]][y[i]].x=p[j1];
    else
        link[x[i]][y[i]].y=p[j1];
    if(link[x[j1]][y[j1]].x==p[j2])
        link[x[j1]][y[j1]].x=p[i];
    else
        link[x[j1]][y[j1]].y=p[i];
}
return 0;
}
//根据基础解构造子棋盘的Hamilton回路
void Knight::base0(int mm,int nn,int offx,int offy)
{
    if(mm==6&&nn==6)
        build(mm,nn,offx,offy,n,b66);
    if(mm==6&&nn==8)
        build(mm,nn,offx,offy,n,b68);
    if(mm==8&&nn==6)

```

```

        build(mm,nn,offx,offy,n,b86);
    if(mm==8&&nn==8)
        build(mm,nn,offx,offy,n,b88);
    if(mm==8&&nn==10)
        build(mm,nn,offx,offy,n,b810);
    if(mm==10&&nn==8)
        build(mm,nn,offx,offy,n,b108);
    if(mm==10&&nn==10)
        build(mm,nn,offx,offy,n,b1010);
    if(mm==10&&nn==12)
        build(mm,nn,offx,offy,n,b1012);
    if(mm==12&&nn==10)
        build(mm,nn,offx,offy,n,b1210);
}

void Knight::build(int m,int n,int offx,int offy,int col ,grid *b)
{
    int i,p,q,k=m*n;
    for(i=0;i<k;i++)
    {
        int
x1=offx+b[i].x,y1=offy+b[i].y,x2=offx+b[(i+1)%k].x,y2=offy+b[(i+1)%k].y;
        p=pos(x1,y1,col);
        q=pos(x2,y2,col);
        link[x1][y1].x =q;
        link[x2][y2].y =p;
    }
}

//计算方格的编号
int Knight::pos(int x,int y,int col)
{
    return col*x+y;
}

void Knight::out0(int m,int n,ofstream &out)
{
    int i,j,k,x,y,p,a[20][20];
    if(comp(m,n,0,0))
        return;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            a[i][j]=0;
        }
    }
    i=0;j=0;k=2;
    a[0][0]=1;
    out<<"(0,0)"<<" ";
    for(p=1;p<m*n;p++)
    {
        x=link[i][j].x;
        y=link[i][j].y;
        i=x/n;j=x%n;
        if(a[i][j]>0)
        {
            i=y/n;
            j=y%n;
        }
    }
}

```

```

    }
    a[i][j]=k++;
    out<<"("<<i<<" "<<j<<")";
    if((k-1)%n==0)
    {
        out<<"\n";
    }
}
out<<"\n";
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        out<<a[i][j]<<" ";
    }
    out<<"\n";
}
}

int main()
{
    int m,n;
    ifstream in("input.txt",ios::in); //利用文件流读取数据
    ofstream out("output.txt",ios::out); //利用文件流将数据存到文件中
    in>>m>>n;
    Knight k(m,n);
    k.comp(m,n,0,0);
    k.out0(m,n,out);
    in.close();
    out.close();
}

```

3.运行结果

6.pdf	input.txt	output.txt
code > hw > hw2 > input.txt	code > hw > hw2 > output.txt	
1 6 6	1 (0,0)(2,1)(4,0)(5,2)(4,4)(2,3)	
	2 (0,4)(2,5)(1,3)(0,5)(2,4)(4,5)	
	3 (5,3)(3,2)(5,1)(3,0)(1,1)(0,3)	
	4 (1,5)(3,4)(5,5)(4,3)(3,1)(5,0)	
	5 (4,2)(5,3)(3,5)(1,4)(0,2)(1,0)	
	6 (2,2)(0,1)(2,0)(4,1)(3,3)(1,1)	
	7 1 32 29 18 7 10	
	8 30 17 36 9 28 19	
	9 33 2 31 6 11 8	
	10 16 23 14 35 20 27	
	11 3 34 25 22 5 12	
	12 24 15 4 13 26 21	

4.分析

时间复杂度分析:

用分治法计算 Hamilton 回路所需计算时间为 $T(n)$, 则 $T(n)$ 满足如下递归式:

$$T(n) = \begin{cases} O(1), & n < 12 \\ 4T(\frac{n}{2}) + O(1), & n \geq 12 \end{cases}$$

因此时间复杂度为:

$$O(n^2)$$

空间复杂度分析:

算法要存储棋盘, 需要 $O(m \times n)$ 的空间; link 存储棋盘的时候要将二维的棋盘线性化, 其规模为 $O(n^2 \times n^2) = O(n^4)$ 。因此空间复杂度为:

$$O(n^4)$$

算法实现题 2-3

2-3 半数集问题。

问题描述: 给定一个自然数 n , 由 n 开始可以依次产生半数集 $\text{set}(n)$ 中的数如下:

- (1) $n \in \text{set}(n)$;
- (2) 在 n 的左边加上一个自然数, 但该自然数不能超过最近添加的数的一半;
- (3) 按此规则进行处理, 直到不能再添加自然数为止。

例如, $\text{set}(6) = \{6, 16, 26, 126, 36, 136\}$ 。半数集 $\text{set}(6)$ 中有 6 个元素。注意, 该半数集是多重集。

算法设计: 对于给定的自然数 n , 计算半数集 $\text{set}(n)$ 中的元素个数。

数据输入: 输入数据由文件名为 `input.txt` 的文本文件提供。每个文件只有一行, 给出整数 n ($0 < n < 1000$)。

结果输出: 将计算结果输出到文件 `output.txt`。输出文件只有一行, 给出半数集 $\text{set}(n)$ 中的元素个数。

输入文件示例

`input.txt`

6

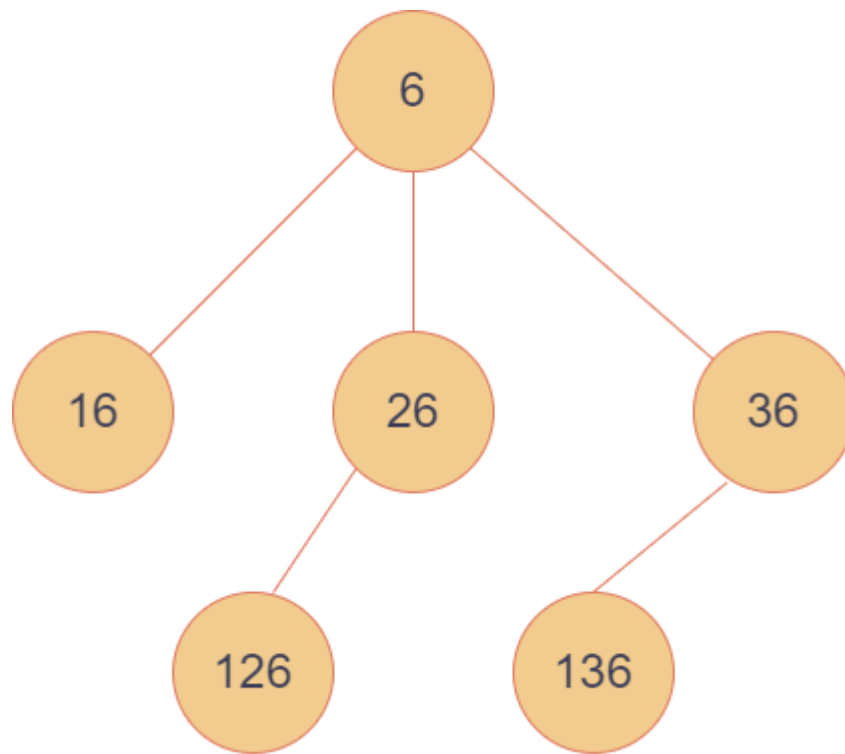
输出文件示例

`output.txt`

6

1.思路

先来了解一下半数集是如何生成的, 以 $\text{set}(6)$ 为例:



我们找到了3个满足条件的数，即1、2、3，他们分别与6构成了16、26、36，依照这三个数继续向集合中添加元素，对于16，1是最近添加的数，但因为比1的一半小的自然数只有0，所以这一步就结束了。对于26，2是最近添加的数，2的一半是1，所以将126也添加到了半数集中。同理，将136也添加到了半数集中。

设 $\text{set}(n)$ 的元素个数为 $f(n)$ ，经过以上例子不难发现， $f(6) = f(3) + f(2) + f(1) + 1$ 。因此，有递归表达式：

$$f(n) = \sum_{i=1}^{n/2} f(i)$$

2.代码

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1005;
typedef long long ll;
int n;
ll f[N];
void solution(int n) {
    ll sum = 1;
    if(f[n]) return;
    for(int i = 1; i <= n/2; ++i) {
        solution(i);
        sum += f[i];
    }
    f[n] = sum;
    return;
}
int main() {
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n;
    solution(n);
}
```

```

    cout << f[n];
    ofs << f[n];
    ifs.close();
    ofs.close();
    return 0;
}

```

关键代码及解释：

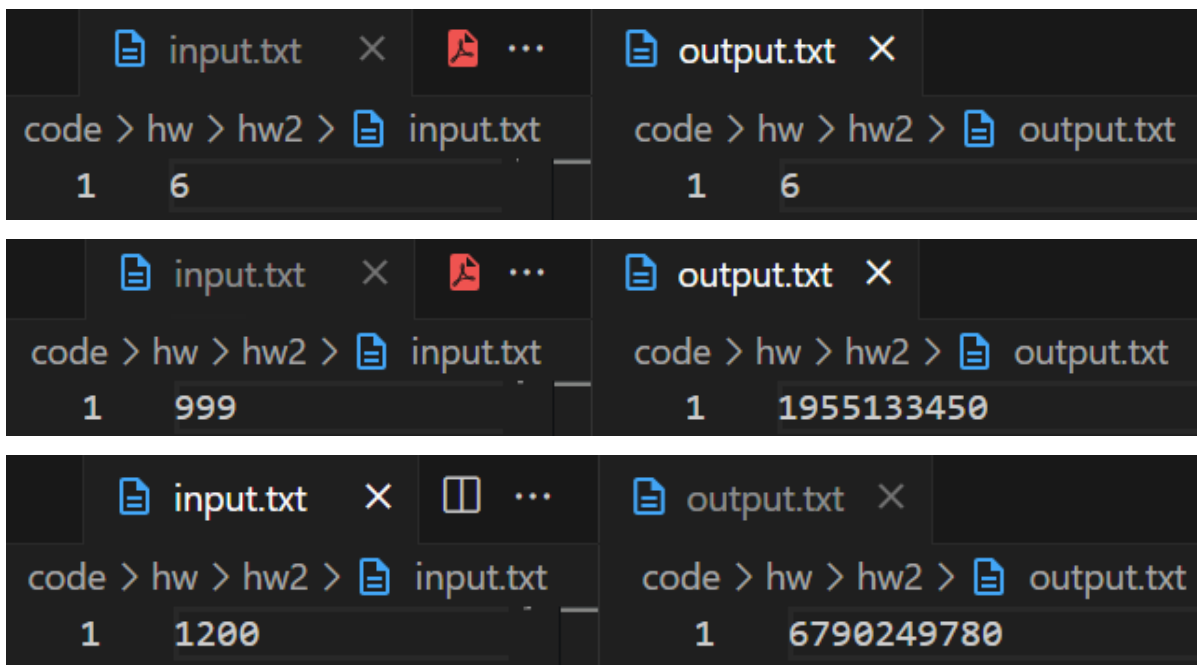
```

void solution(int n) {
    if sum = 1;
    if(f[n]) return; //如果f[n]不为0，表示已经计算过，直接返回
    for(int i = 1; i <= n/2; ++i) {
        solution(i);
        sum += f[i];
    }
    f[n] = sum; //存放结果
    return;
}

```

这里是递归的代码，这里使用 `记忆化数组f` 用来存放已经计算过的半数集，避免重复计算，提高算法效率。

3.运行结果



4.分析

时间复杂度分析：

如果 `f` 数组在未被初始化的情况下存储先前的计算结果，可以减少重复计算，从而降低实际的运行时间。使用记忆化后，实际时间复杂度会降到 $O(n)$ ，因为每个 `f[i]` 只会计算一次。

空间复杂度分析：

由于使用递归，空间复杂度主要由递归栈决定。最坏情况下，递归深度达到 n ，因此空间复杂度为 $O(n)$ 。

算法实现题 2-6

2-6 排列的字典序问题。
问题描述： n 个元素 $\{1, 2, \dots, n\}$ 有 $n!$ 个不同的排列。将这 $n!$ 个排列按字典序排列，并编号为 $0, 1, \dots, n!-1$ 。每个排列的编号为其字典序值。例如，当 $n=3$ 时，6 个不同排列的字典序值如下：

字典序值	0	1	2	3	4	5
排列	123	132	213	231	312	321

算法设计：给定 n 及 n 个元素 $\{1, 2, \dots, n\}$ 的一个排列，计算出这个排列的字典序值，以及按字典序排列的下一个排列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n 。接下来的 1 行是 n 个元素 $\{1, 2, \dots, n\}$ 的一个排列。

结果输出：将计算出的排列的字典序值和按字典序排列的下一个排列输出到文件 output.txt。文件的第 1 行是字典序值，第 2 行是按字典序排列的下一个排列。

输入文件示例	输出文件示例
input.txt	output.txt
8	8227
2 6 4 5 8 1 7 3	2 6 4 5 8 3 1 7

1.思路

计算字典序列：

设定排列 $P = [p_1, p_2, \dots, p_n]$ ，根据每个元素在剩余元素中的相对位置，计算该排列的字典序值。字典序值可以通过从左到右依次确定某一位的排列情况，并用阶乘计算剩余位置上的排列数来累加。

假设 $p_1 = x$ ，那么字典序中排在 x 前面的排列数等于：**第一位小于 x 的所有排列数**。其数量为： $(x - 1) \times (n - 1)!$ 。以此类推，对每一位 P_i 求贡献，累加得出字典序值。每一位的贡献是：该位在剩余元素中的相对位置乘以剩余位置上所有排列数的阶乘。

寻找下一个字典序排列：

使用 STL 中的 next_permutation 函数，可以直接得到排列 P 的下一个排列。

2.代码

```
#include<bits/stdc++.h>
#define endl '\n'
typedef long long ll;
const int N = 100;
using namespace std;
int n;
ll factorial[N];
int perm[N];
void calcu_factorial(int n) {
    factorial[1] = 1;
    for(int i = 2; i <= n; i++) {
        factorial[i] = i * factorial[i - 1];
    }
}
```

```

        return;
    }
    11 order() {
        11 order = 0;
        vector<int> elems;
        for(int i = 1; i <= n; ++i) elems.push_back(i);

        for(int i = 0; i < n; ++i) {
            int pos = find(elems.begin(), elems.end(), perm[i]) - elems.begin();
            order += pos * factorial[n - i - 1];
            elems.erase(elems.begin() + pos);
        }

        return order;
    }
    int main() {
        ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
        ifstream ifs("input.txt");
        ofstream ofs("output.txt");
        ifs >> n;
        for(int i = 0; i < n; ++i) ifs >> perm[i];

        calcu_factorial(n);

        ofs << Order() << endl;

        //计算下一个字典序排列
        next_permutation(perm, perm + n);

        for(int i = 0; i < n; ++i) ofs << perm[i] << ' ';

        ifs.close();
        ofs.close();
        return 0;
    }
}

```

数组 `factorial` 用来存放阶乘。


在计算字典序的函数 `order()` 中，有序列表 `elems` 用来存放初始元素，便于计算 `perm[i]` 在剩余元素中的相对位置。

对于每一个 `perm[i]`，我们查找它在 `elems` 列表中的位置 `pos`。这个 `pos` 告诉我们当前元素在剩下的可选元素中排第几。我们利用这个位置 `pos` 来更新字典序值。字典序的贡献是：`pos` 乘以剩下元素的阶乘数，即 `pos * factorial(n - i - 1)`。

处理完 `perm[i]` 后，我们将该元素从 `elements` 列表中移除，保证下一次迭代时只考虑剩余的元素。

3.运行结果

File	Line 1	Line 2
input.txt	1 8	2 2 6 4 5 8 1 7 3
output.txt	1 8227	2 2 6 4 5 8 3 1 7

```
code > hw > hw2 >  output.txt
```

```
1 148011744958480169
2 2 6 4 5 8 1 7 3 10 9 13 11 12 15 14 19 20 16 17 18
```

$$S(n, k) = k \cdot S(n - 1, k) + S(n - 1, k - 1)$$

(1)第一部分考虑的是将第 n 个元素加入到已经划分好的 k 个子集中。我们可以把新元素放入任意一个已经存在的 k 个子集中的 k 种选择。因此, 这部分的结果是 $k \cdot S(n-1, k)$, 表示将 $n-1$ 个元素划分为 k 个非空子集的方式, 随后选择一个子集来放入新元素。

(2)第二部分考虑的是将第 n 个元素单独放入一个新的子集中。因此这部分的结果是 $S(n-1, k-1)$ 。

贝尔数 $B(n)$:表示将 n 个元素的集合划分为非空子集的总数, 可以通过斯特林数计算:

$$B(n) = \sum_{k=1}^n S(n, k)$$

2.代码

```
#include<bits/stdc++.h>
#define endl '\n'
typedef unsigned long long ull;
const int N = 105;
using namespace std;
ull S[N][N];
int n;
void StirlingNumbers() {
    S[0][0] = 1; //s(0, 0) = 1
    for(int i = 1; i <= n; ++i) {
        for(int k = 1; k <= n; k++) {
            S[i][k] = k * S[i-1][k] + S[i-1][k-1];
        }
    }
}
ull BellNumber(int n) {
    ull number = 0;
    for(int k = 1; k <= n; ++k) number += S[n][k]; //计算bell数
    return number;
}
int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n;
    StirlingNumbers();
    ofs << BellNumber(n);
    ifs.close();
    ofs.close();
    return 0;
}
```

关键代码及解释:

```

void StirlingNumbers() {
    s[0][0] = 1; //s(0, 0) = 1
    for(int i = 1; i <= n; ++i) {
        for(int k = 1; k <= n; k++) {
            s[i][k] = k * s[i - 1][k] + s[i - 1][k - 1];
        }
    }
}

```

使用递推 $S(n, k) = k \cdot S(n - 1, k) + S(n - 1, k - 1)$ 求斯特林数。

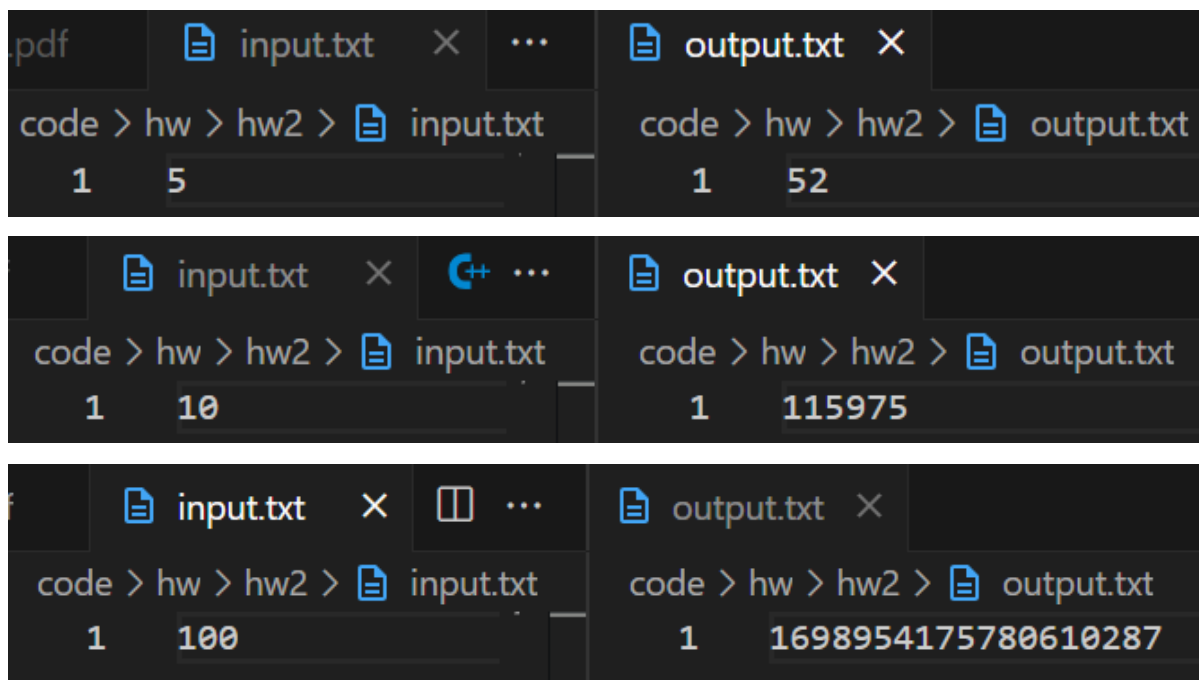
```

ull BellNumber(int n) {
    ull number = 0;
    for(int k = 1; k <= n; ++k) number += s[n][k]; //计算bell数
    return number;
}

```

通过斯特林数求出贝尔数 $B(n) = \sum_{k=1}^n S(n, k)$, 即划分的非空子集的总数。

3.运行结果



4.分析

时间复杂度分析:

计算斯特林数的嵌套循环中, 外层循环迭代 n 次, 内层循环迭代最多 n 次。因此, 构建斯特林数表的时间复杂度为 $O(n^2)$ 。计算贝尔数时, 循环遍历从 1 到 n , 并且对每个 k 值从斯特林数表中获取值, 时间复杂度为 $O(n)$ 。因此, 时间复杂度为:

$$O(n^2)$$

空间复杂度分析：

需要一个二维数组 S 构建斯特林数表，因此空间复杂度为：

$$O(n^2)$$