搜索算法

湖南师大附中 许力

目录

深度优先搜索

Depth First Search

全排列问题

- 编号为1~3的三张扑克牌，放到编号为1~3的三个盒子中，有多少种不同的摆放方案？

Sample input	Sample output
	123 132 213 231 312 321

暴力秒

```
123 132 213 231 312 321
--------------------------------
Process exited after 0.1291 seconds w
ith return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    for (int a = 1; a <= 3; a ++)
    for (int b = 1; b <= 3; b ++)
    for (int c = 1; c <= 3; c ++)
        if (a != b && a != c && b != c)
            printf("%d%d%d ", a, b, c);
    return 0;
 }
```

继续

- 编号为1~n的n张扑克牌，放到编号为1~n的n个盒子中，有多少种不同的摆放方案？

$n \le 10$

Sample input	Sample output
3	123 132 213 231 312 321

继续

- 显然无法穷举了，因为我们要写n层循环，而n在输入前是未知的

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
	int n;
	scanf("%d", &n);

	?

	return 0;
 }
```

递归的本质

- 我们再回顾一下递归的本质：

递归相当于这样一件事：

- 每递归调用一次，相当于在栈顶压入新的元素
- 而每次递归调用结束返回上一层，相当于释放当前栈顶的元素

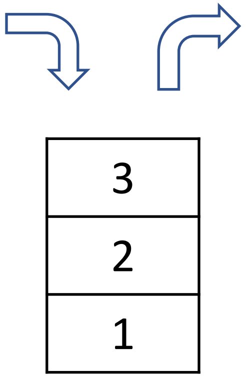
递归的本质

- 我们再回顾一下递归的本质：

- 有什么联系？

联想到我们试探性把一张张扑克牌放入盒子的过程

并且把已放好的扑克牌取回来，进行新一轮试探的过程



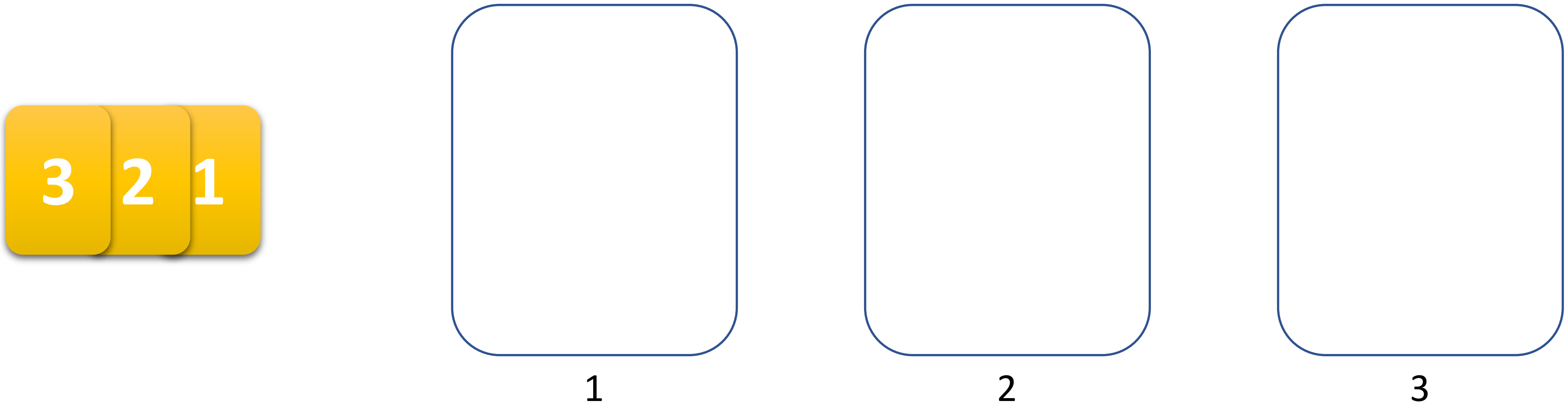
- 这个过程和递归调用的过程是合拍的

分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

1. 逐个尝试把一张扑克牌放入盒子

```
a[j] = i;     //将编号 i的扑克牌放入第 j个盒子
```



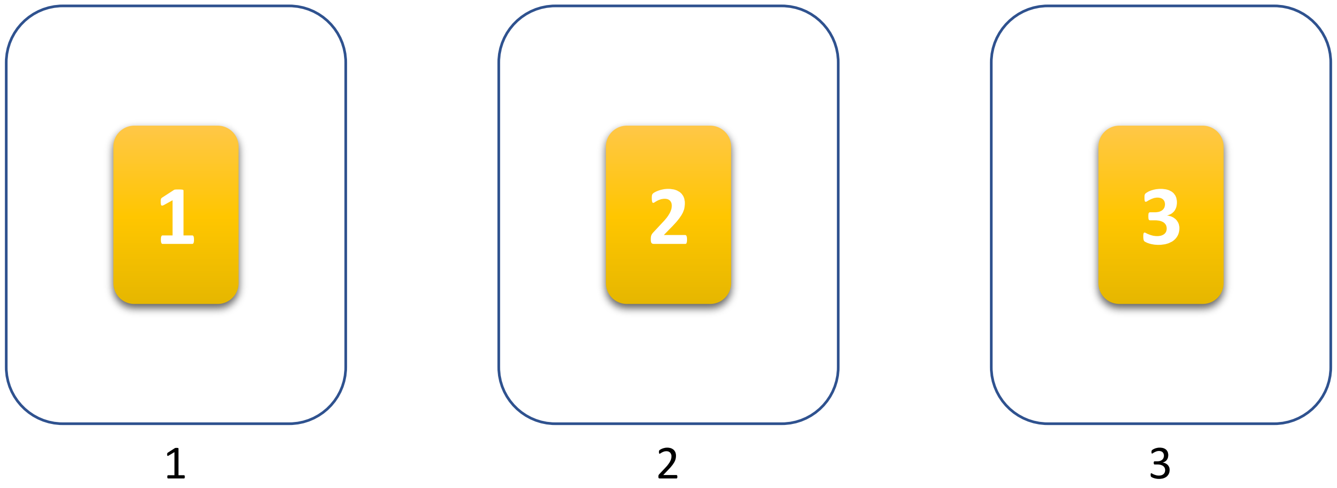
分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

2. 生成一种方案后，就直接输出：123

怎么判断生成了一种方案？

```
if (j > n)    //此时全部 n个盒子已摆放完毕
{
      输出;
}
```



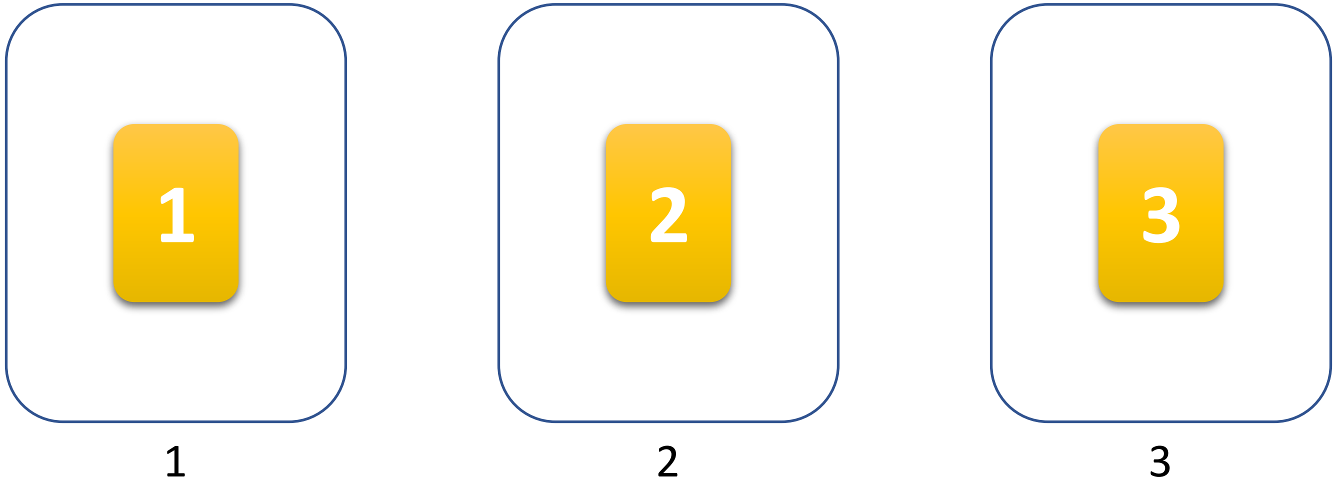
分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

2.　生成一种方案后，就直接输出：123

怎么判断生成了一种方案？

```
for (int i = 1; i <= n; i ++) printf("%d", a[i]);
printf(" ");
```

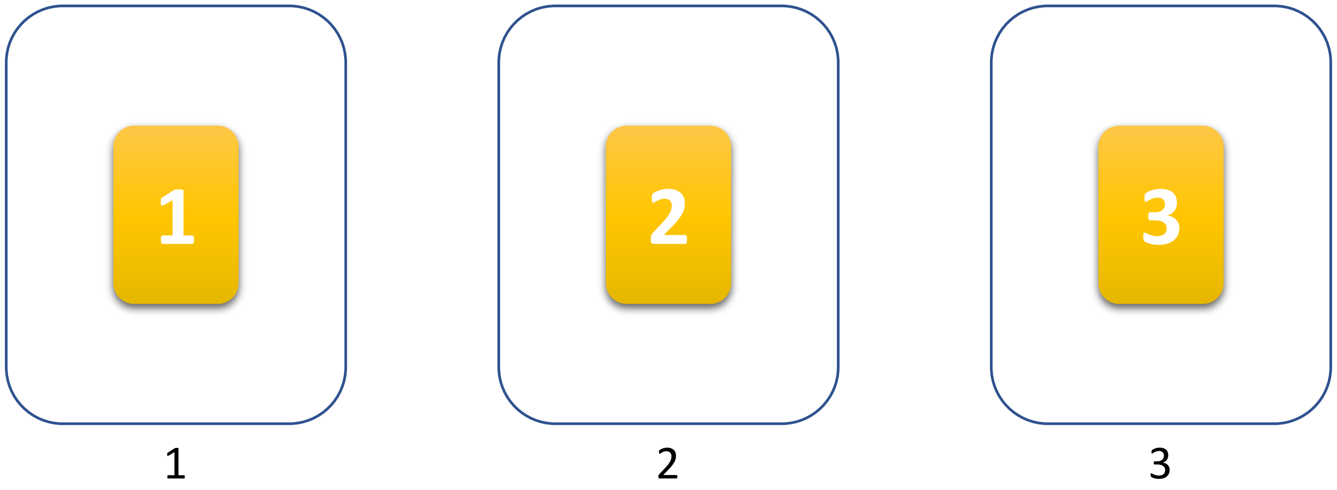


分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

3. 接下来要尝试新的摆放方案。

但此时所有盒子被占满，我们需要考虑把3号牌收回来



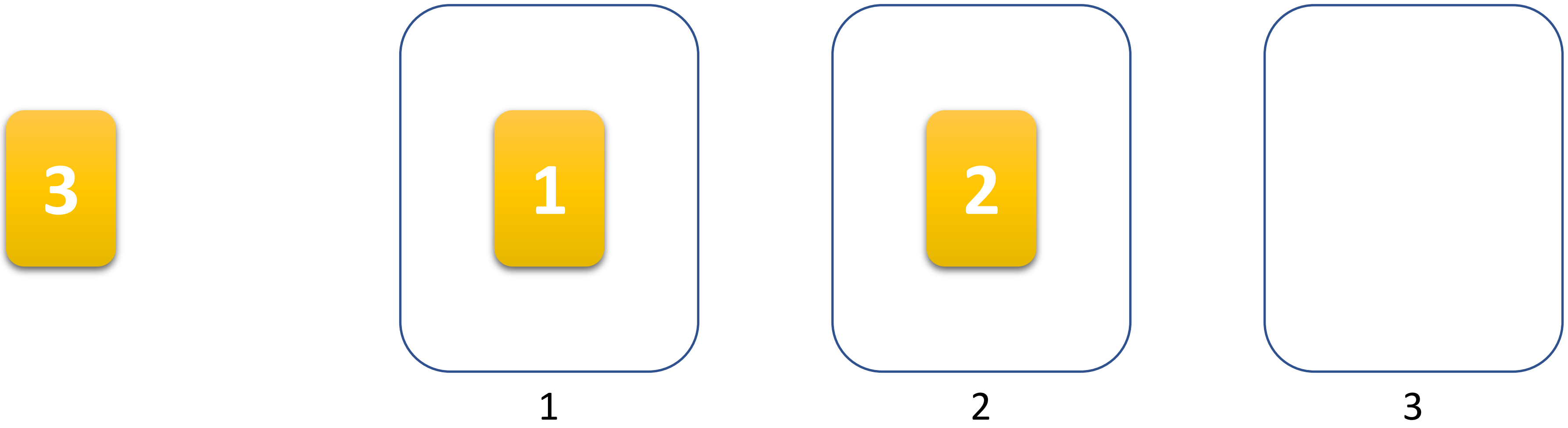
分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

2. 接下来要尝试新的摆放方案。

但此时所有盒子被占满，我们需要考虑把3号牌收回来

这样还不够，因为我们手上只有3号牌，只有3号盒子为空，于是把2也收回



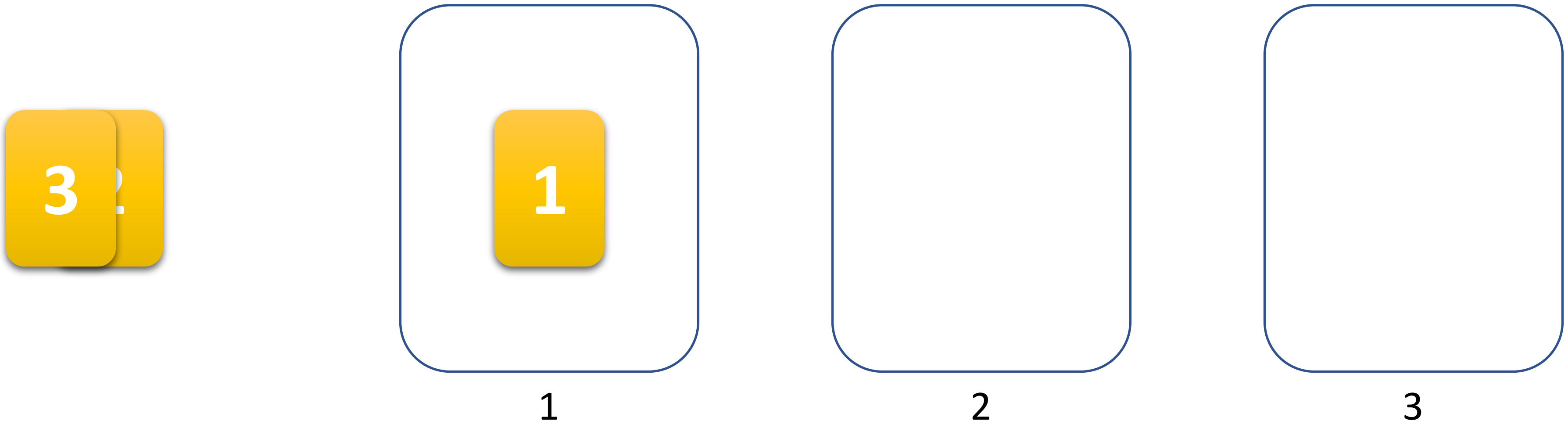
分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

3. 接下来要尝试新的摆放方案。

在这种情况下，我们怎么避免出现又把2号牌放入2号盒子的~~傻逼~~行为？

我们必须给每张牌打上标记，标记它是放了还是没放

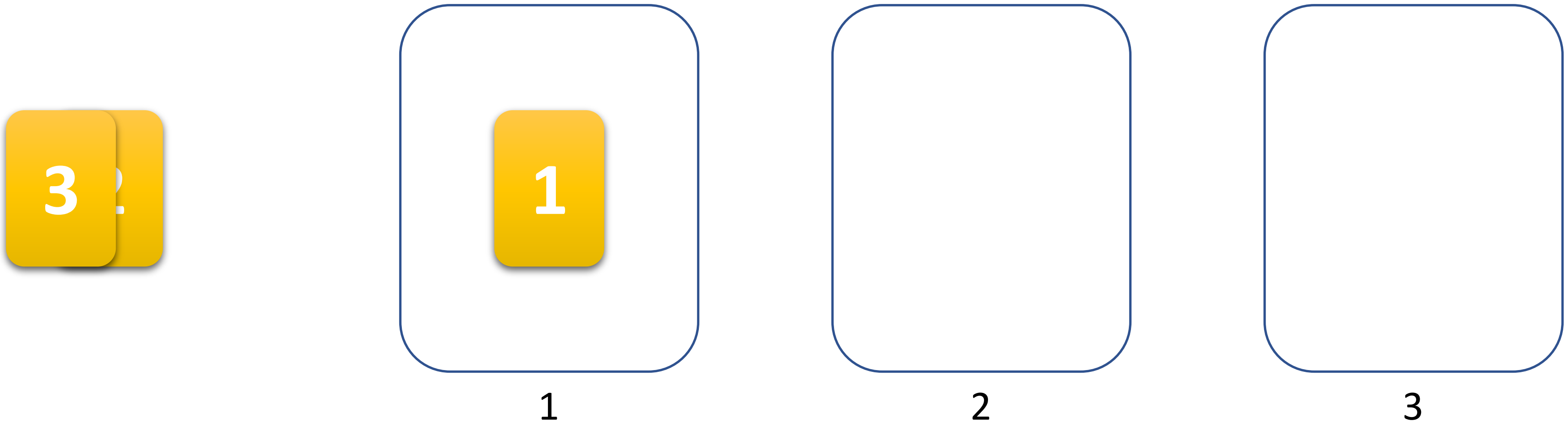


分析

- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

3. 接下来要尝试新的摆放方案。

```
if (!flag[i])    //如果扑克牌还在手上
{
    a[j] = i;     //将编号 i的扑克牌放入第 j个盒子
    flag[i] = 1;   //标记已放入
}
```



分析

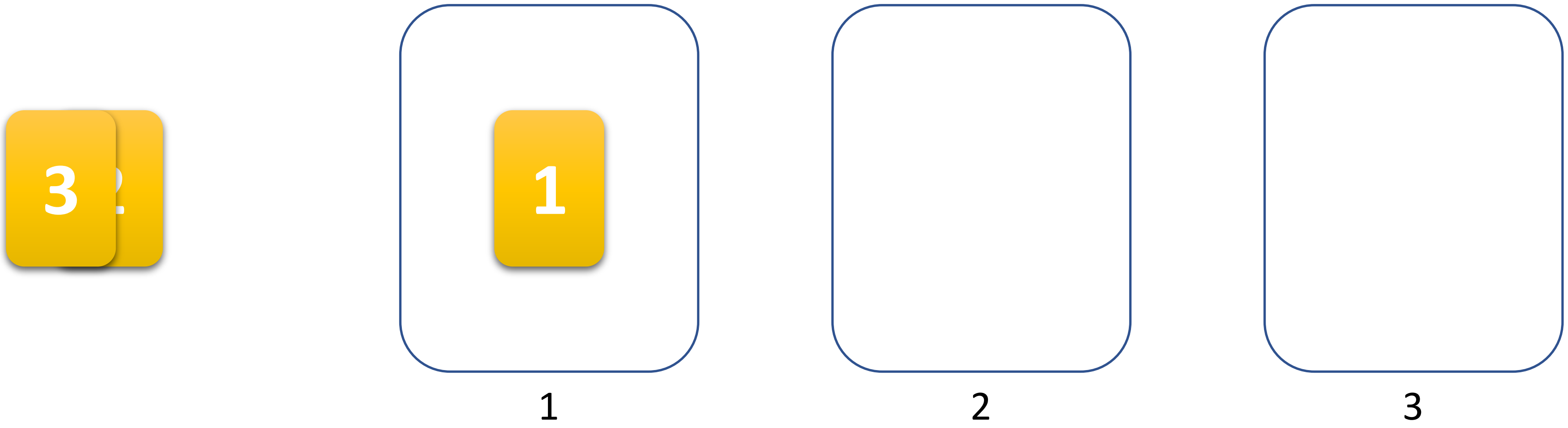
- 我们还是以三张扑克牌为例，来手工模拟一下全过程：

3. 接下来要尝试新的摆放方案。

什么时候释放标记？这是整个程序中最难理解的

答案是：递归返回到上一个盒子的时候释放标记

```
if (!flag[i])    //如果扑克牌还在手上
{
    a[j] = i;     //将编号 i的扑克牌放入第 j个盒子
    flag[i] = 1;   //标记已放入
    dfs(j + 1);   //递归尝试下一个盒子
    flag[i] = 0;   //递归返回时，将刚才尝试的牌收回
}
```



参考代码

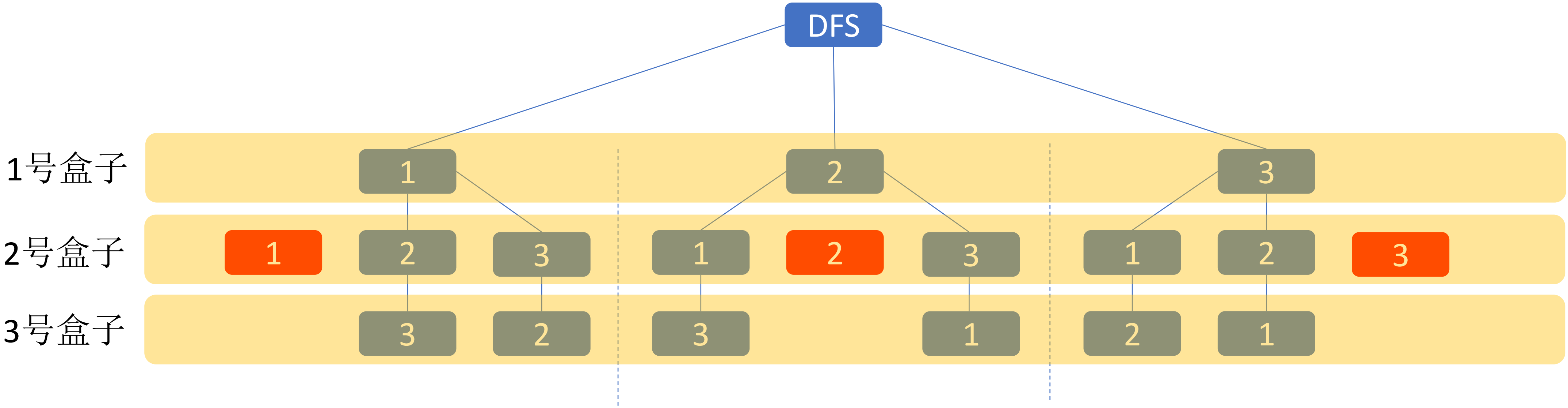
```
4
1234 1243 1324 1342 1423 1432 2134 2143
2314 2341 2413 2431 3124 3142 3214 3241
3412 3421 4123 4132 4213 4231 4312 4321
--------------------------------
Process exited after 1.428 seconds with
return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[20], flag[20], n;
void dfs(int j) // j代表尝试摆放第 j个盒子
{
    if (j > n)    //此时全部 n个盒子已摆放完毕
    {
        for (int i = 1; i <= n; i ++) printf("%d", a[i]);
        printf(" ");
     }
    else for (int i = 1; i <= n; i ++)
        if (!flag[i])    //如果 i号扑克牌还在手上
        {
            a[j] = i;     //将 i号扑克牌放入第 j个盒子
            flag[i] = 1;   //标记 i号牌已放入
            dfs(j + 1);   //递归尝试第 j+1个盒子
            flag[i] = 0;   //递归返回时，将刚才尝试的 i号牌收回
        }
     return;
}
```

```cpp
int main()
{
    scanf("%d", &n);
    dfs(1);
    return 0;
}
```

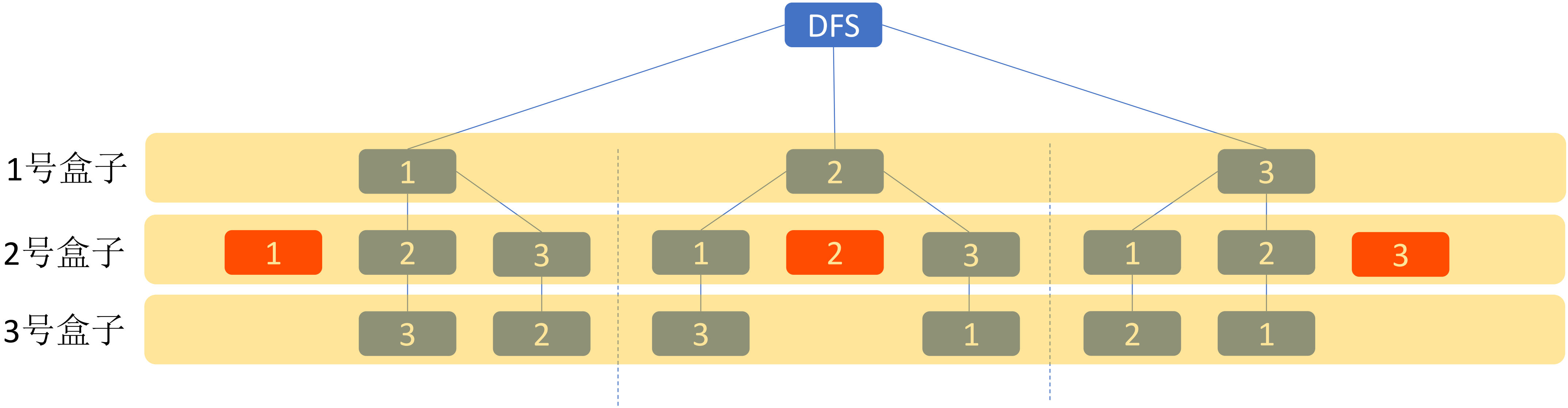
深度优先搜索

- 刚才全排列问题的思路，形如下图：
- 数字代表扑克牌的编号



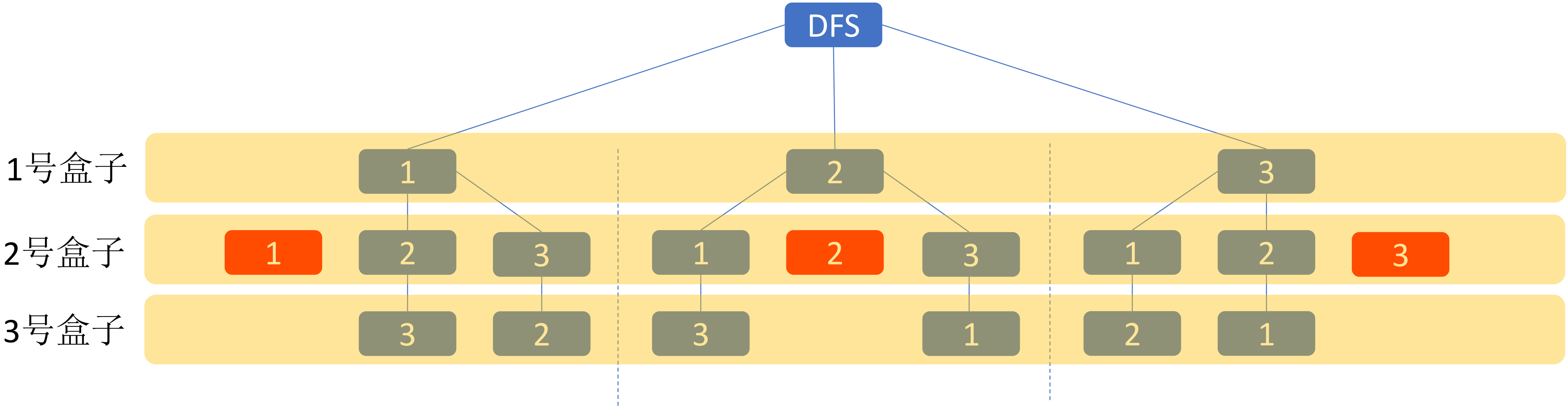
深度优先搜索

- 所有可能的解构成的解空间形如一棵树，称为搜索树
- 整个过程类似于不断往叶节点拓展，如无法拓展则返回其父节点，直到访问完所有可访问的节点为止



深度优先搜索

- 因此这种搜索算法，被称为深度优先搜索（DFS）



DFS的基本模型

```
void dfs(int step)
{
    if (符合输出条件) 输出;
    else (枚举所有的可能项)
    {
        尝试一种可能;
        给它打上标记;
        递归到下一步 dfs(step + 1);
        (回溯时)释放它的标记;
    }
}
```

素数环

- 数字1~n组成一个数字序列，不要求有序，但是要求相邻的两个数的和是一个素数，这样的一个序列称为素数环
- 给定n，要求输出所有的素数环

n≤20

Sample input	Sample output
8	1 2 3 8 5 6 7 4 1 2 5 8 3 4 7 6 1 4 7 6 5 8 3 2 1 6 7 4 3 8 5 2

分析

- 首先，1~n要组成数字序列，标准的全排列问题
- 然后在尝试摆放数字时，加上素数判定的条件即可

```
if (!flag[i] && prime(当前数i和前一个数))
if (!flag[i] && prime(当前数i和后一个数))
```

- 附带的，搜索要从2开始

分析

- 首先，1~n要组成数字序列，标准的全排列问题

- 注意输出的时候，首尾也要符合条件：

```
if (j > n && prime(队首和队尾))
```

自然数的拆分问题

- 将自然数n拆分为若干个小于n的自然数之和，求所有拆分方案

n≤50

Sample input	Sample output
5	1+1+1+1+1 1+1+1+2 1+1+3 1+2+2 1+4 2+3

分析

- 这题依然是全排列问题，所以可以通过搜索求解
- 我们从1开始搜索，然后逐个尝试下一个数

- 拆分出几个数，这种尝试就停下来？
- 这里和前两题不一样的地方是：我们不能预知答案的个数，但可以通过“已经拆出多少和”/“还剩多少和没有拆”来控制循环的进行或者结束

```
void dfs(int j, int sum)    // 已拆出 j 个数字，已经拆出的数字和为 sum
{



}
```

分析

- 这题依然是全排列问题，所以可以通过搜索求解

- 不过这题的输出格式：5=1+1+3，我们可以拆成1+1+和3输出
- 也就是单独处理尾项，这个相信大家自己可以应付

分析

- 这题不再是全排列问题，但依然可以通过搜索求解
- 输出的问题解决了，接下来我们处理递归与回溯

```
for (i = 1; i <= n; i ++)
{
    标记访问;
    递归尝试下一个数;
    回溯;
}
```

分析

```
for (i = 1; i <= n; i ++)
{
      标记访问;
      递归尝试下一个数;
      回溯;
}
```

- 如果我们还是按标准的DFS模型这样处理，肯定会在搜出1+1+3之后，再搜出1+3+1这样的方案的
- 如果还要特地为此去判重，不仅麻烦而且低效

分析

- 我们仔细观察一下：

1+1+1+1+1

1+1+1+2

1+1+3

1+2+2

1+4

2+3

- 有什么办法可以规避重复？
- 如果我们能保证后面拆分的数不小于前面拆分的数，就不必担心会搜出1+3+1这样的方案

分析

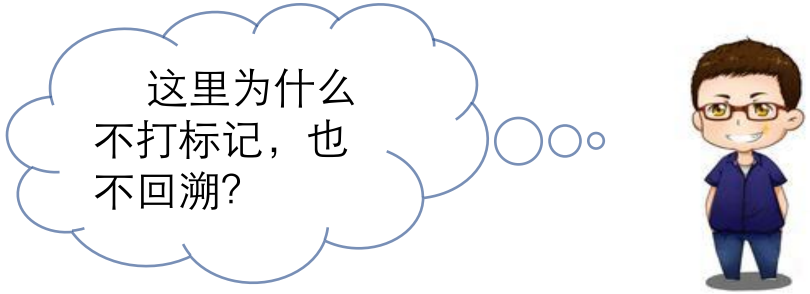
```
for (i = 1; i <= n; i ++)
{
    标记访问;
    递归尝试下一个数;
    回溯;
}
```

- 那么要做怎样的改动，就可以保证后续拆分的数不小于前面的数？
- 注意这些被搜出来的数，都是i的当前值
- 只要我们修改i的范围：下限是之前拆出来的最大数，上限是还剩的和，就可以保证只会搜出1+1+3

分析

- 所以我们为dfs函数增加一个变量，用于记录之前搜出来的最后一个数

```
void dfs(int j, int x, int sum)
// 已拆出 j个数字, 最后一个是 x, 目前拆出的数字和为 sum
{
    if (sum == n)    //拆分出来的数之和等于n就输出
    {
        输出;
    }
    else for (i = x; i <= n - sum; i ++)
    //余下拆出的数字最小为 x, 保证了序列不降, 最大到 n-sum, 这样最后的和不会超过 n
    {
        尝试拆出一个数 i;
        递归尝试拆出下一个数（也许还是 i）;
    }
}
```



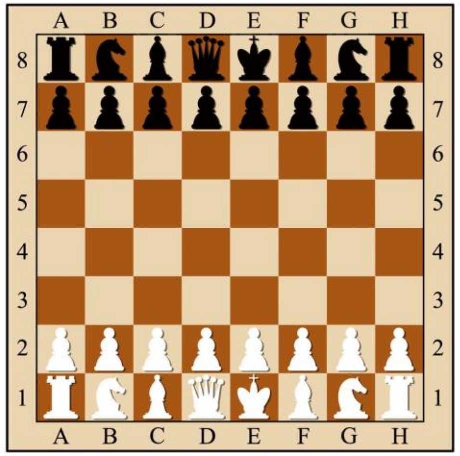
修正

- 修正的办法很多：
- 比如我们要求输出的数<n
- 或者要求至少拆出2个数

```
5
1+1+1+1+1
1+1+1+2
1+1+3
1+2+2
1+4
2+3
5
```

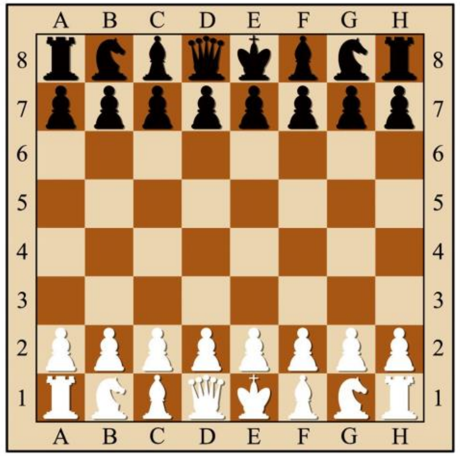
八皇后

- 在8×8的国际象棋棋盘中放置八个“皇后”，使任意两个“皇后”不能同处一行、一列、一条对角线，一共有多少种合法的方案？



分析

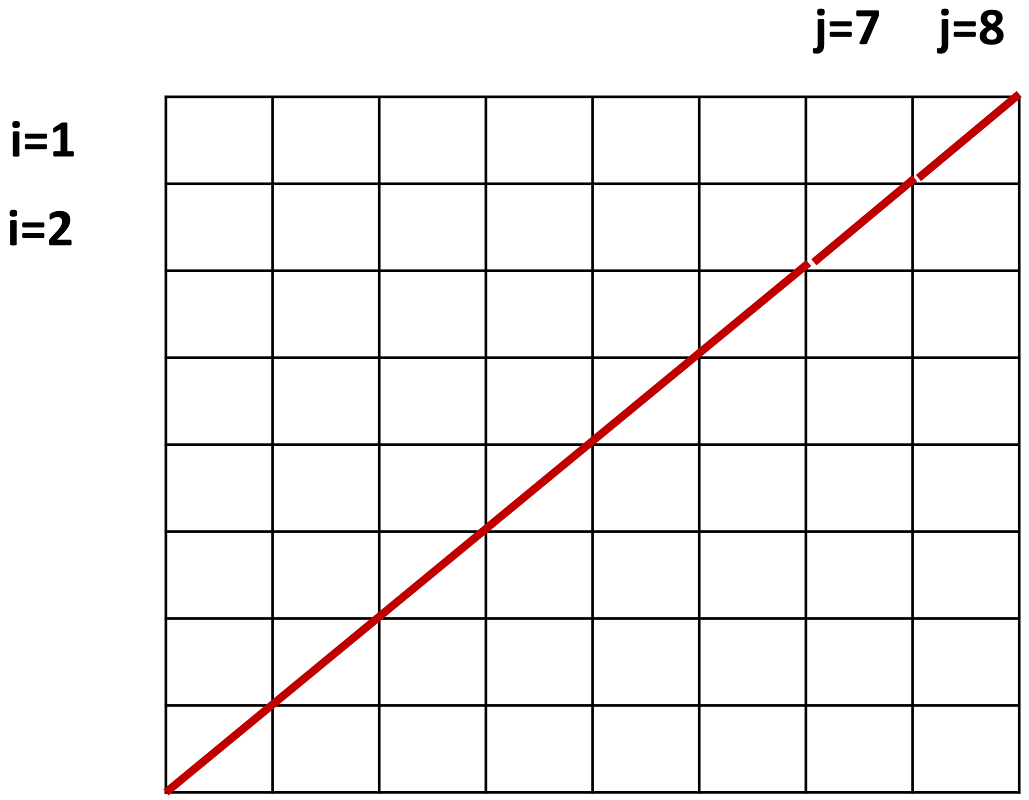
- 这题要求不同行、不同列，这是标准的全排列问题
- 同时还要求不同对角线，这是全排列问题的加强版



分析

- 对角线怎么表达？

flag[i+j]



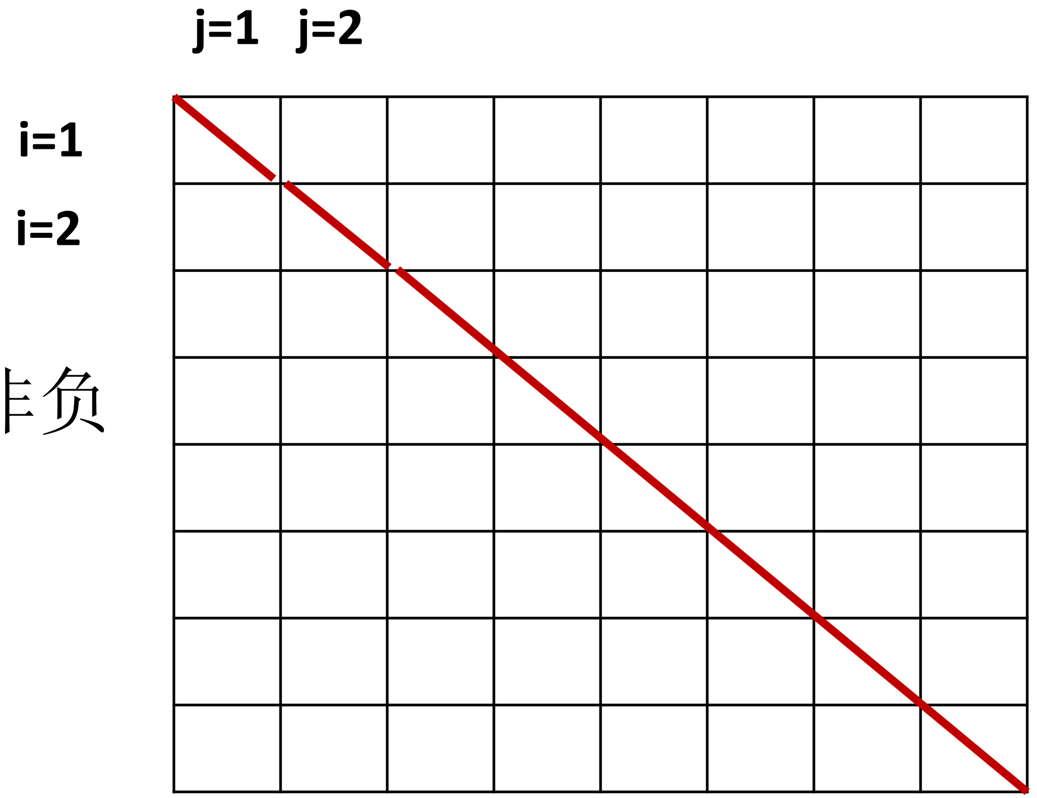
分析

- 对角线怎么表达？

flag[i-j]

- 但是i-j做了减法，而数组下标非负

flag[i-j+8]



分析

- 不过我们没有必要分别用两个标记数组来标记左、右对角线
- 我们开一个d[2][30]的数组来同时标记两条对角线

```
d[0][j+i]   //右倾对角线
d[1][i-j+8])  //左倾对角线
```

分析

- 解决了对角线的问题，我们可以考虑这样的全排列模型：
- 把8个皇后，逐个放在不同的八列中，要求互相之间不能冲突

- 图示这种不成功，最后回溯到第二列

1							
	1						
		1					
			1				
			1				

分析

- 解决了对角线的问题，我们可以考虑这样的全排列模型：
- 把8个皇后，逐个放在不同的八列中，要求互相之间不能冲突

- 图示这种不成功，最后回溯到第二列
- 成功！

1							
						1	
				1			
							1
	1						
			1				
					1		
		1					

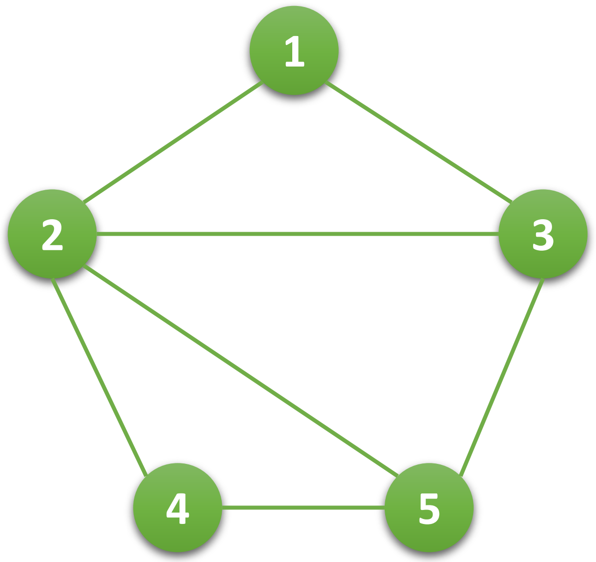
课外加练

- luogu 1706　　全排列问题
- luogu uva524　　素数环
- luogu 2404　　自然数的拆分问题
- luogu 1219　　八皇后
- luogu 1036　　选数
- luogu 1657　　选书
- luogu 1019　　单词接龙
- luogu 1101　　单词方阵

图上的DFS

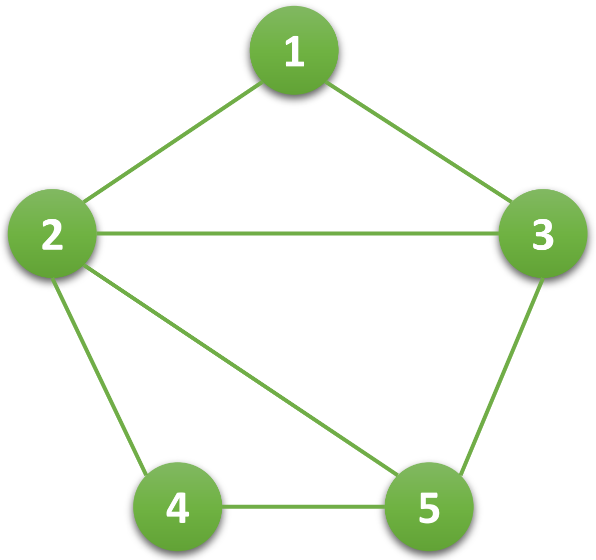
- 之前的DFS搜索都是在树上展开的，除此之外DFS搜索也可以在图上展开

- 比如右图，从1号点出发，每个点访问一次，直到访问完所有的点为止。给出访问的DFS先后顺序



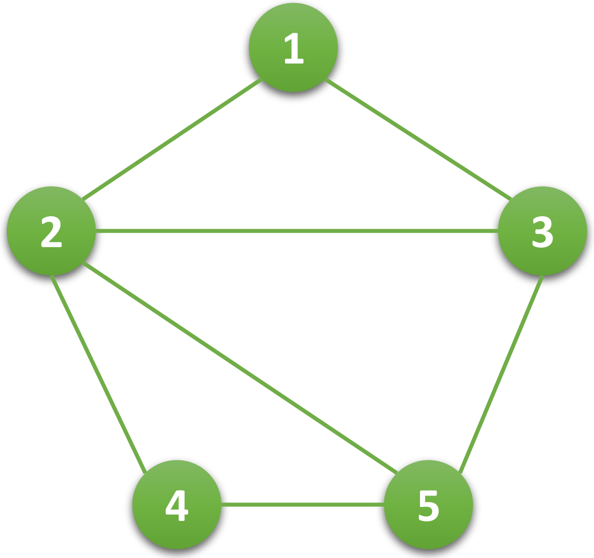
邻接矩阵

- 要在图上DFS，首先需要解决一个问题：存图
- 我们的办法是：如果一对顶点(u,v)有边相连，则在二维数组里对应的位置存1，表示可以达到/访问
- 否则在对应位置存INF无穷大，表示无法到达



邻接矩阵

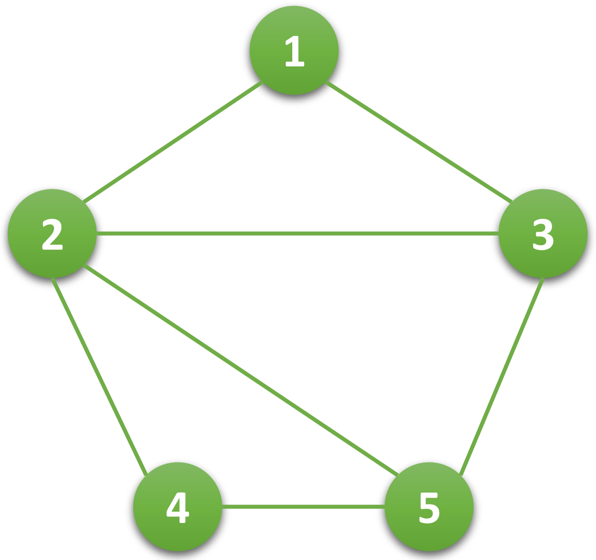
- 初始化：刚开始除了每个点可以到达自己，其他都是INF



	1	2	3	4	5
1	0	INF	INF	INF	INF
2	INF	0	INF	INF	INF
3	INF	INF	0	INF	INF
4	INF	INF	INF	0	INF
5	INF	INF	INF	INF	0

初始化邻接矩阵

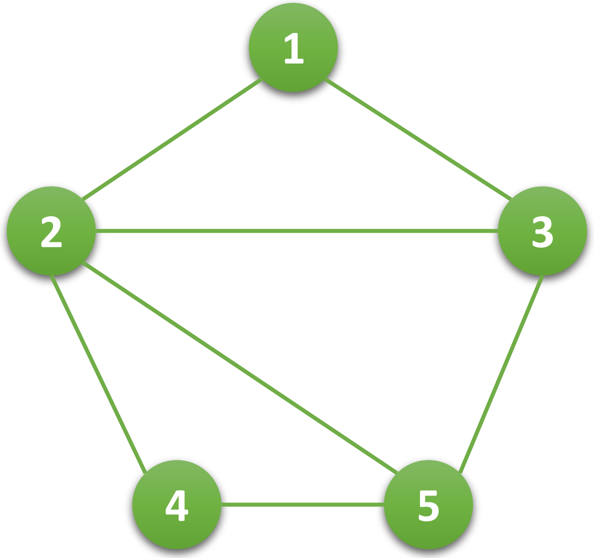
```
for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= n; j ++)
        if (i == j) edge[i][j] = 0;
        else edge[i][j] = INF;
```



邻接矩阵

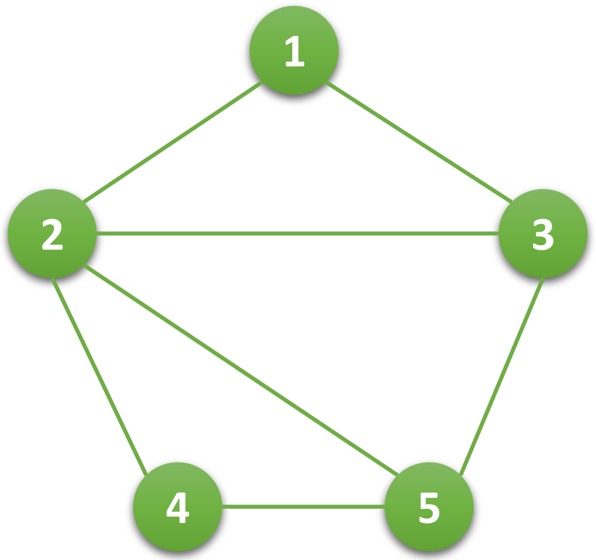
- 接下来就可以每读入一条边，存入一条了

	1	2	3	4	5
1	0	1	1	INF	INF
2	1	0	1	1	1
3	1	1	0	INF	1
4	INF	1	INF	0	1
5	INF	1	1	1	0



邻接矩阵

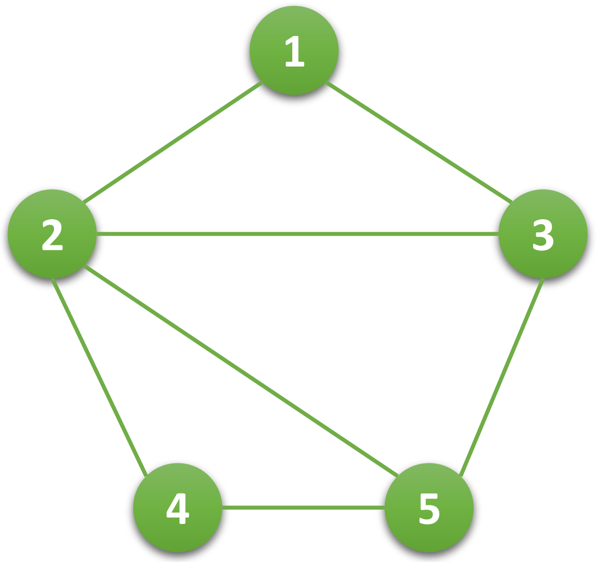
- 我们不难发现：如果是无向图，那么该矩阵关于左对角线对称



	1	2	3	4	5
1	0	1	1	INF	INF
2	1	0	1	1	1
3	1	1	0	INF	1
4	INF	1	INF	0	1
5	INF	1	1	1	0

邻接矩阵存图

```
while (m --)
{
    scanf("%d%d", &u, &v);
    edge[u][v] = edge[v][u] = 1;
    //无向图对称
 }
```

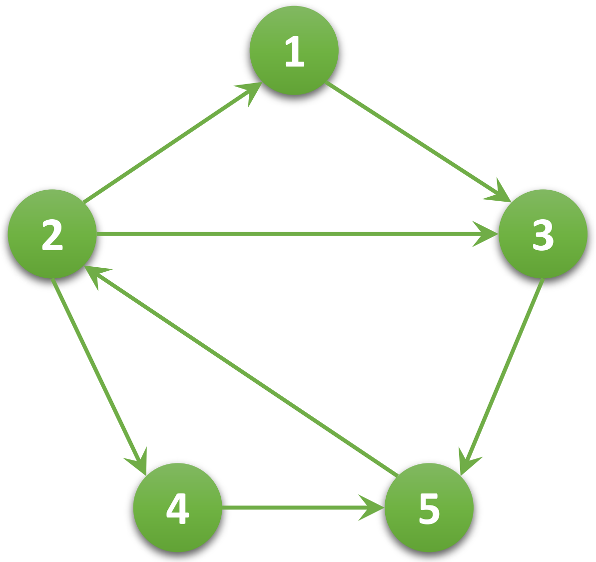


邻接矩阵

- 如果是有向图：

```
while (m --)
{
    scanf("%d%d", &u, &v);
    edge[u][v] = 1;
 }
```

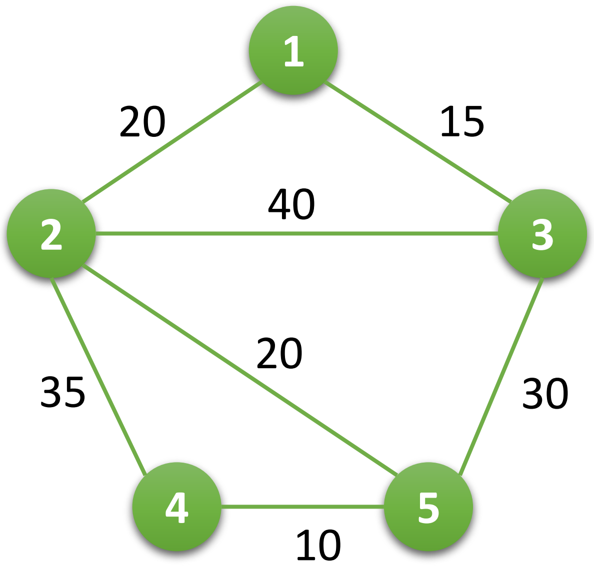
	1	2	3	4	5
1	0	INF	1	INF	INF
2	1	0	1	1	INF
3	INF	INF	0	INF	1
4	INF	INF	INF	0	1
5	INF	1	INF	INF	0



邻接矩阵

- 如果图中带边权值：

```
while (m --)
{
    scanf("%d%d%d", &u, &v, &w);
    edge[u][v] = edge[v][u] = w;
}
```

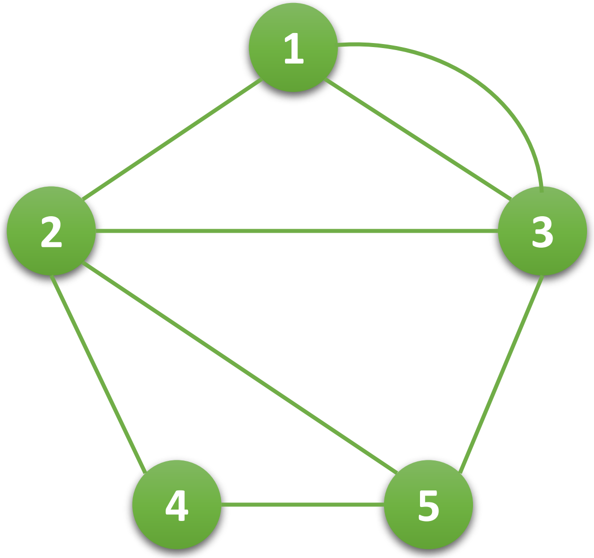


	1	2	3	4	5
1	0	20	15	INF	INF
2	20	0	40	35	20
3	15	40	0	INF	30
4	INF	35	INF	0	10
5	INF	20	30	10	0

邻接矩阵

- 邻接矩阵存图的缺陷：

1. 空间开销大
2. 无法存储重边

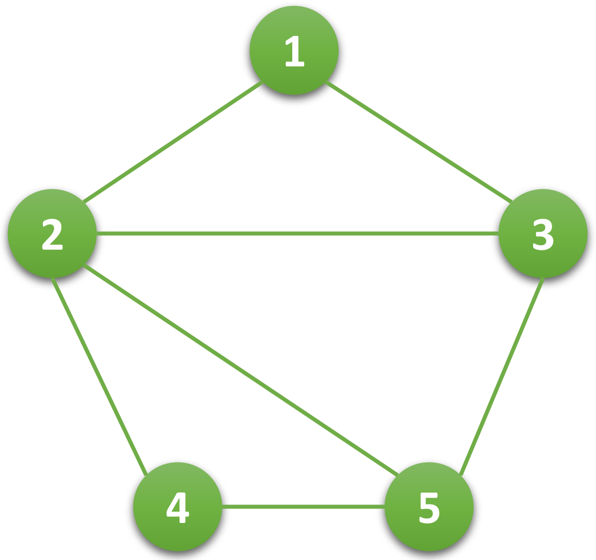


<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>…</th><th>1w</th></tr>
<tr><td>1</td><td colspan="5" rowspan="5">无法存储</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>…</td></tr>
<tr><td>1w</td></tr>
</table>

图上的DFS

- 解决了存图的问题，再来看图上的DFS

- 我们的搜索策略是：一旦发现从当前顶点到下一个顶点有边相连，且下一个顶点未被访问，则向下搜索，否则回溯



图上的DFS

```
5 7
1 2
1 3
2 3
2 4
2 5
3 5
4 5
1 2 3 5 4
--------------------------------
```

```cpp
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
int flag[1010], n, m, edge[1010][1010];
void dfs(int u)
{
    printf("%d ", u);
    flag[u] = 1;
    for (int v = 1; v <= n; v ++)
        if (edge[u][v] == 1 && !flag[v]) dfs(v);
    //如果 u到 v有边相连，且 v未访问过
 }
```

```cpp
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++)
        for (int j = 1; j <= n; j ++)
            if (i == j) edge[i][j] = 0;
            else edge[i][j] = INF;
    int u, v;
    while (m --)
    {
        scanf("%d%d", &u, &v);
        edge[u][v] = edge[u][v] = 1;
    }
    dfs(1);
    return 0;
}
```