

# 回溯算法

问题描述：输出n个数字的全部排列,数字可以重复

输入: 1,2,3

输出: 111 112 113 121 122 123 131 132 133 ... 333



解空间 (搜索空间)

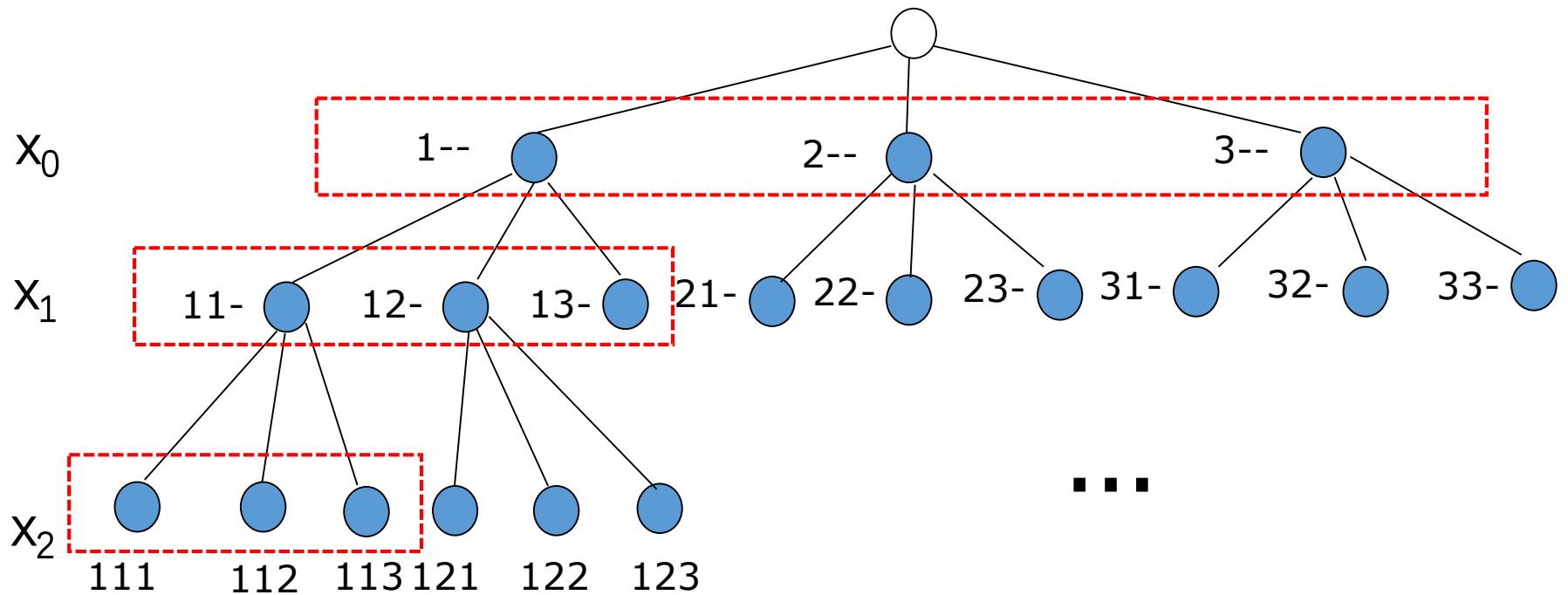
一个解 ( $x_0, x_1, x_2$ )

回溯：按照深度优先顺序遍历整个解空间

# 解空间（搜索空间）

每个元素都有三个候选值{1,2,3}

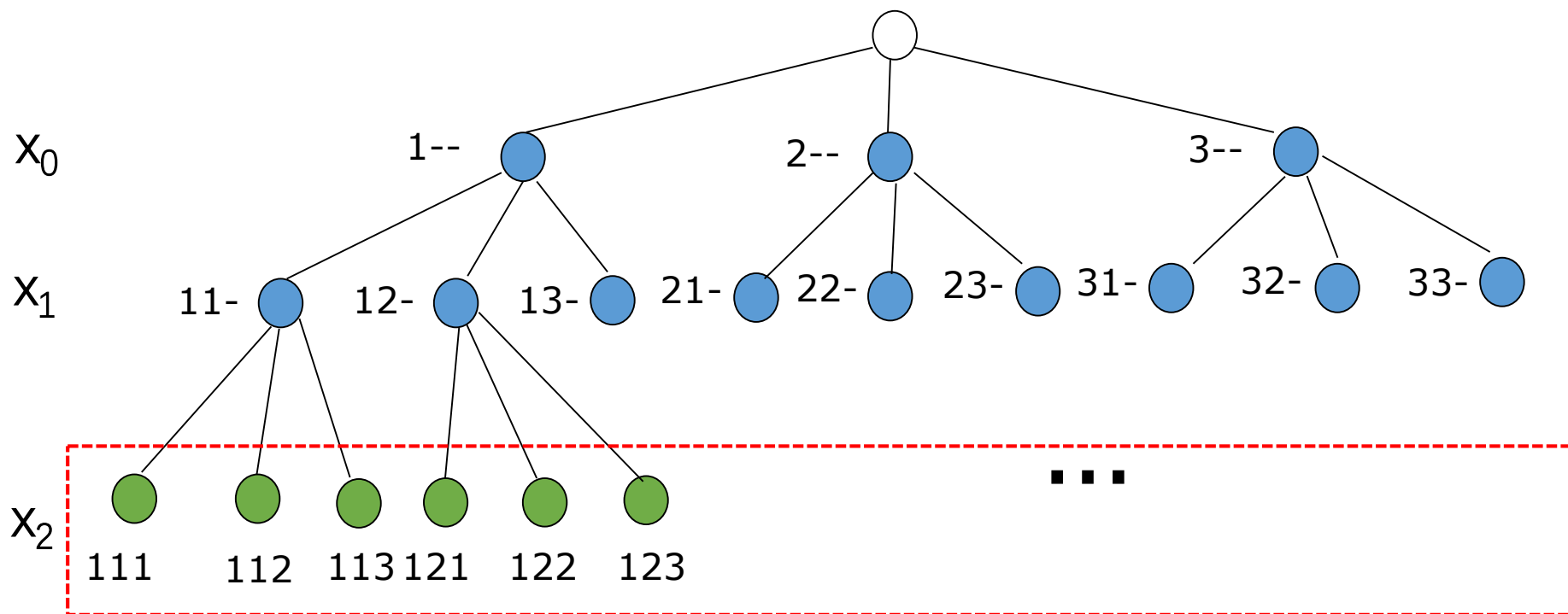
$(x_0, x_1, x_2)$



# 解空间 (搜索空间)

每个元素都有三个候选值{1,2,3}

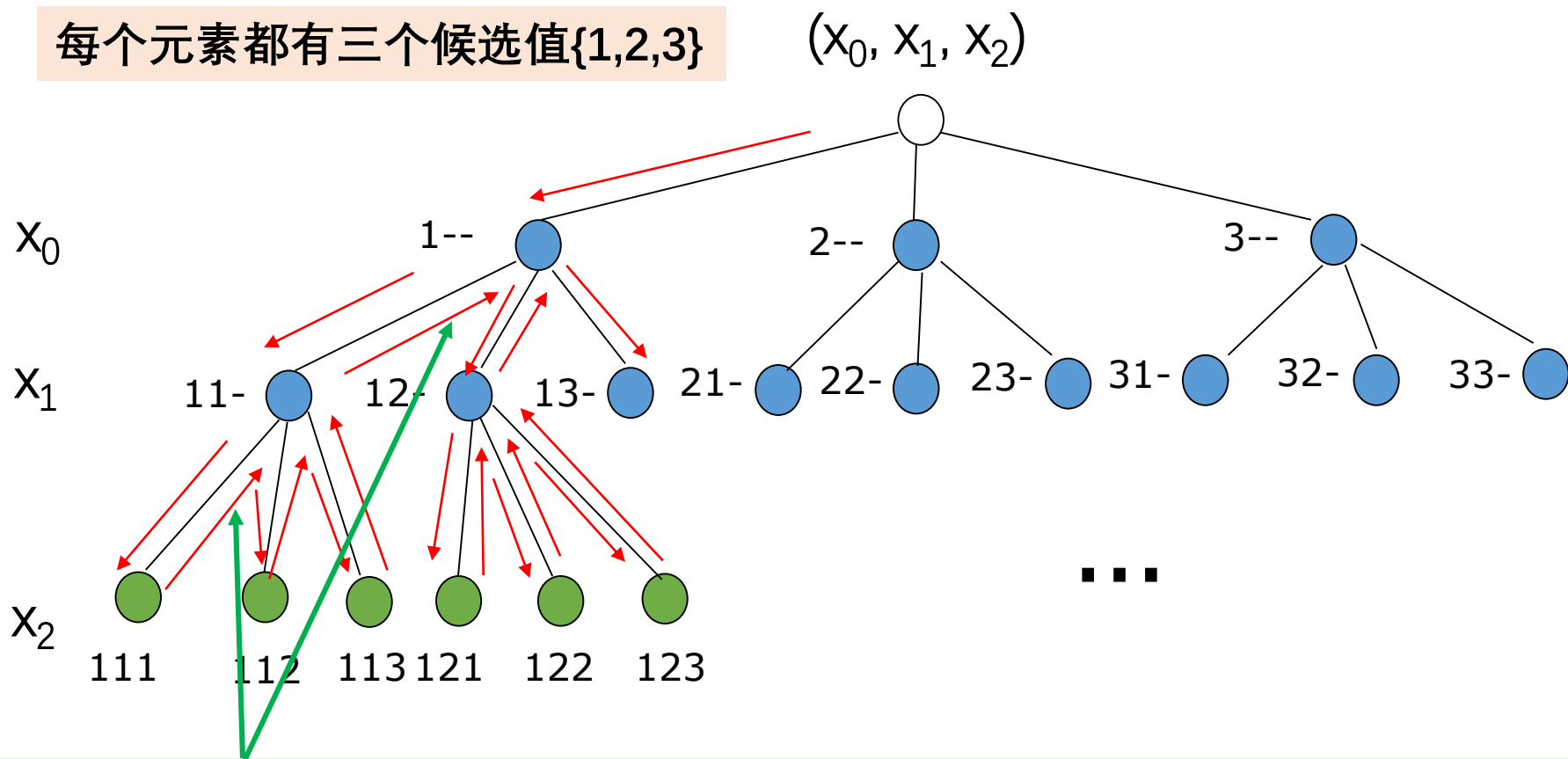
$(x_0, x_1, x_2)$



解空间 (搜索空间)

**深度优先：** 每次决定解的一个元素，先决定 $x_0$ ，再决定 $x_1, \dots$

每个元素都有三个候选值{1,2,3}



**回溯：** 深度优先，从一条路往前走，能进则进，不能进退回来，换一条路再试，直到走完搜索空间

## 回溯算法：递归实现

`int x[3];` //数组存放解, `x[0]` 代表  $x_0$ , `x[1]` 代表  $x_1$  ...

`void Search (int k) {` //参数`k`表示正在决定`x[k]`

`if (k > 2)`

`cout<<x[0]<<x[1]<<x[2];` ← 找到一组解, 输出

`else`

`for (int i = 1; i <= 3; i ++)` {

`x[k] = i;`

`Search(k+1);`

}

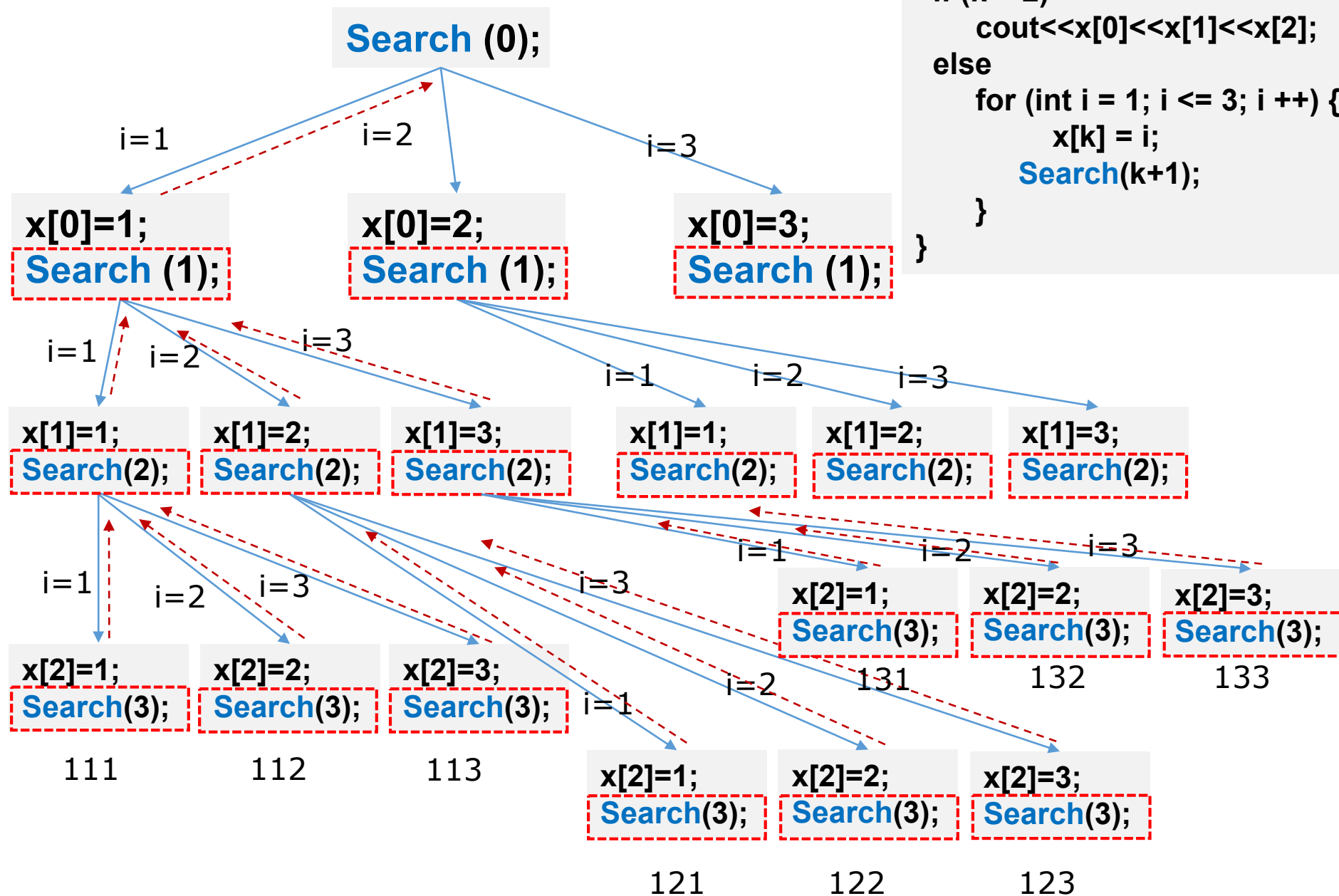
}

`x[k]`有三种选择 {1,2,3},  
逐个尝试

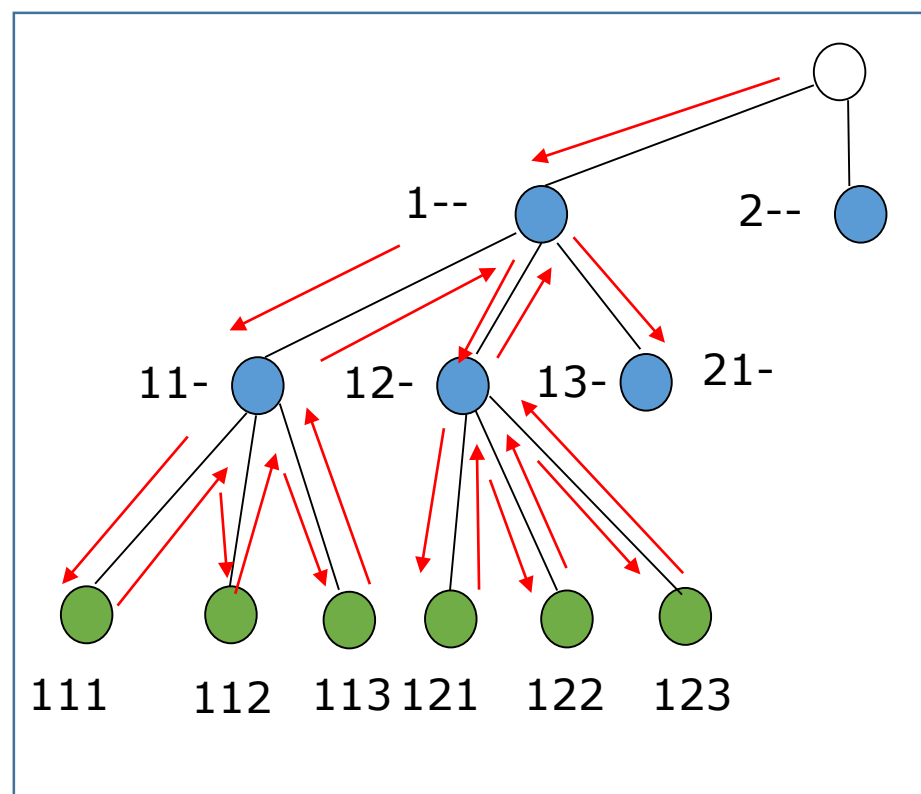
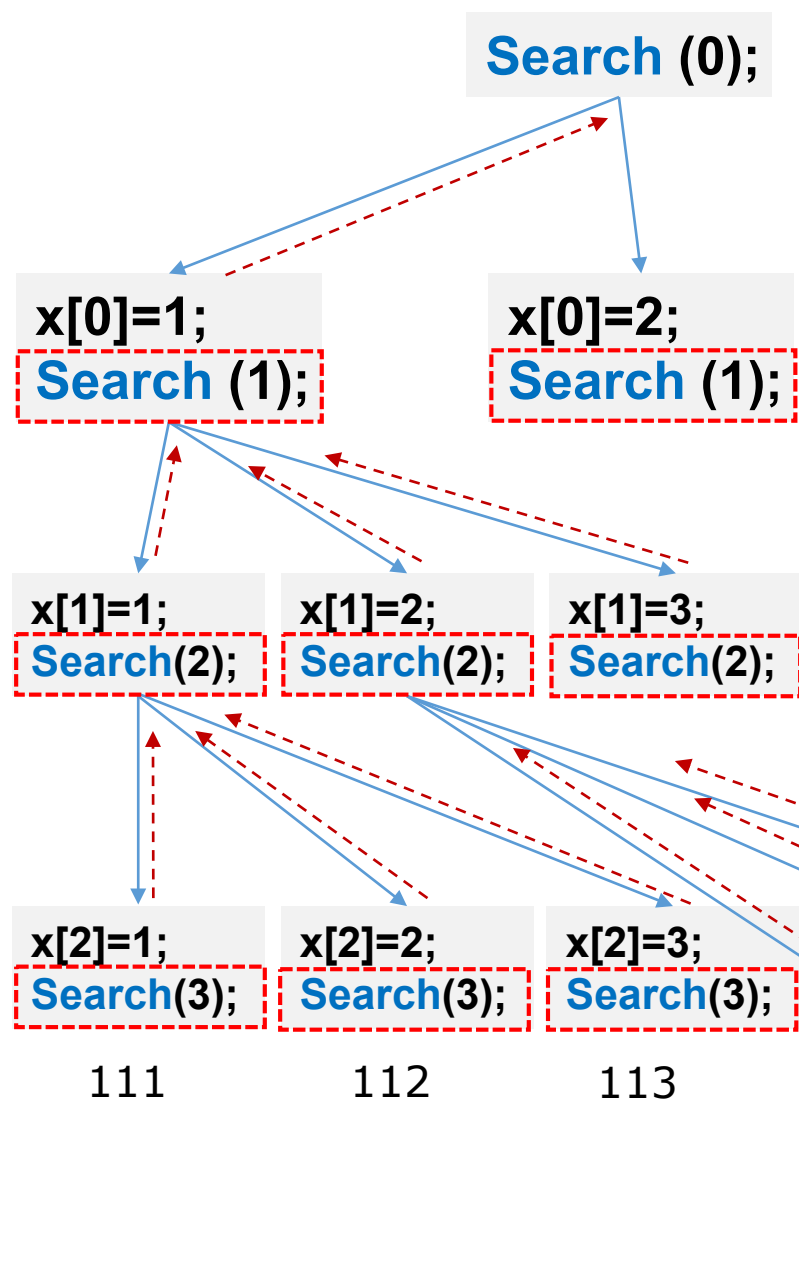
`x[k]` 确定后, 决定 `x[k+1]`

# 回溯算法：递归实现

```
int x[3];  
void Search (int k) {  
    if (k > 2)  
        cout<<x[0]<<x[1]<<x[2];  
    else  
        for (int i = 1; i <= 3; i++) {  
            x[k] = i;  
            Search(k+1);  
        }  
}
```



# 回溯算法：递归实现





# 完整程序

```
#include<iostream>
using namespace std;
int x[3];
void Search (int k) {
    if (k > 2)
        cout<<x[0]<<x[1]<<x[2];
    else
        for (int i = 1; i <= 3; i ++) {
            x[k] = i;
            Search(k+1);
        }
}

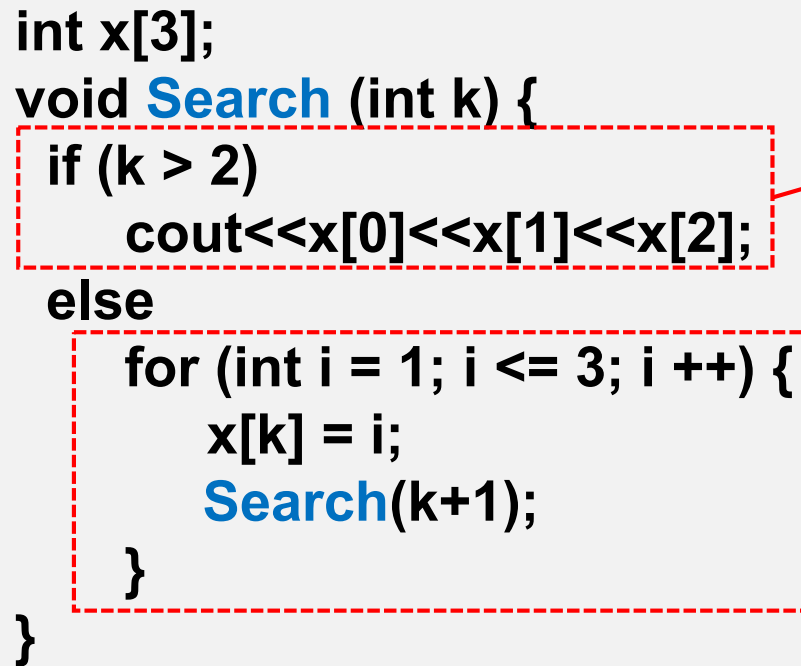
int main() {
    Search (0);
    return 0;
}
```

**思考：**为什么不直接用三重for循环？  
如果有n个数字，需要多少重循环？

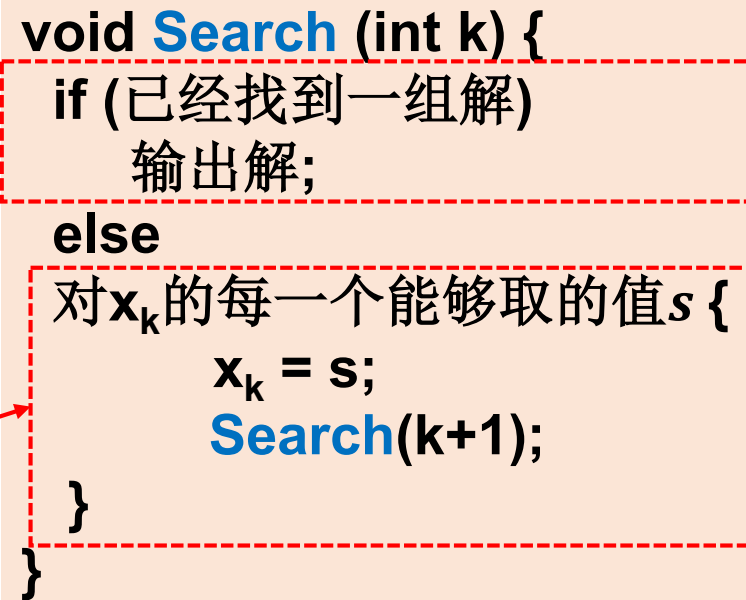
# 回溯算法：递归实现

## 回溯算法的通式

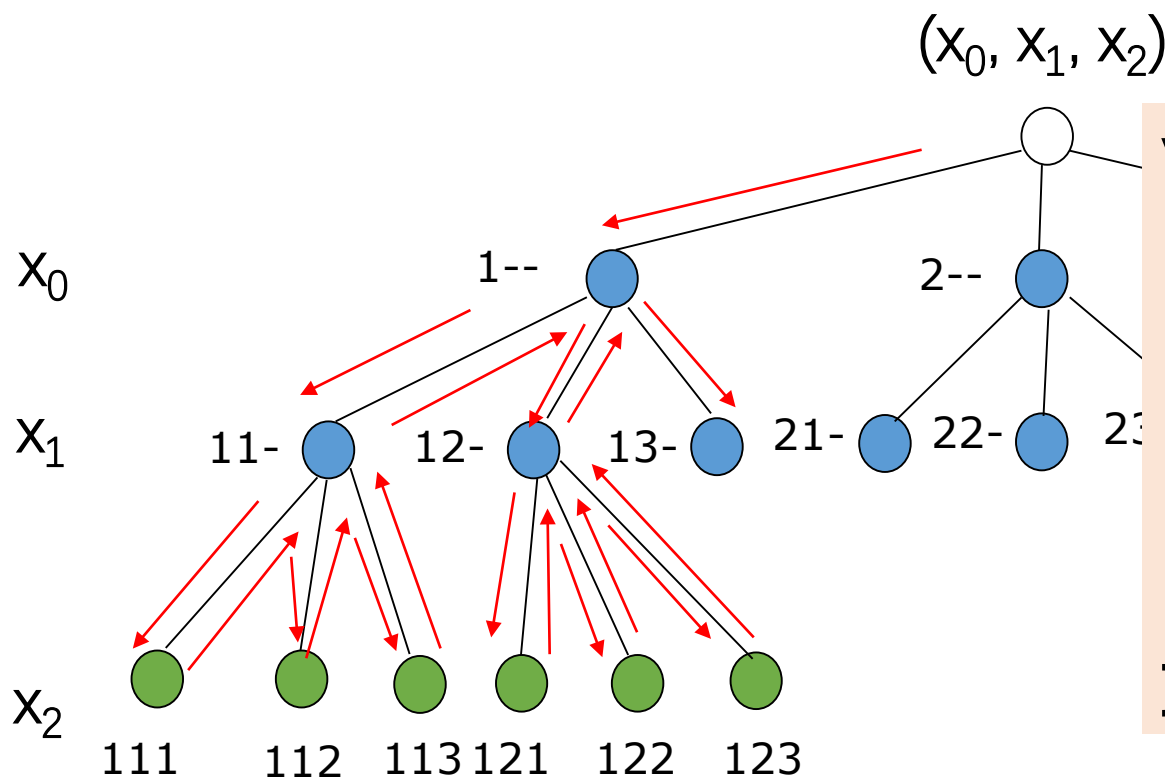
```
int x[3];  
void Search (int k) {  
    if (k > 2)  
        cout<<x[0]<<x[1]<<x[2];  
    else  
        for (int i = 1; i <= 3; i++) {  
            x[k] = i;  
            Search(k+1);  
        }  
}
```



```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 的每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            Search(k+1);  
        }  
}
```



# 回溯算法通式



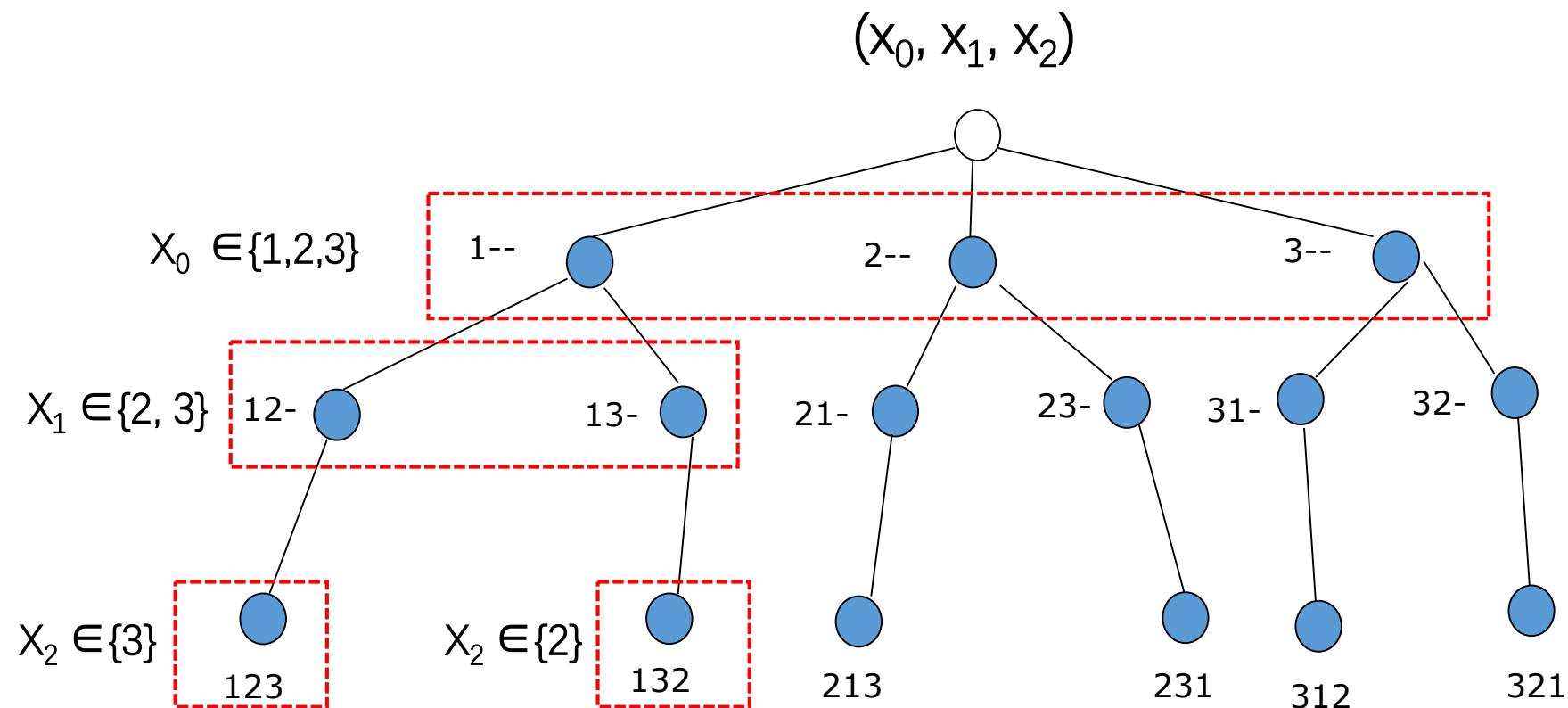
```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 的每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            Search(k+1);  
        }  
}
```

按照通式程序，就可以完成对解空间的深度优先搜索

问题描述：输出n个数的全部排列,数字**不能重复**

输入: 1,2,3

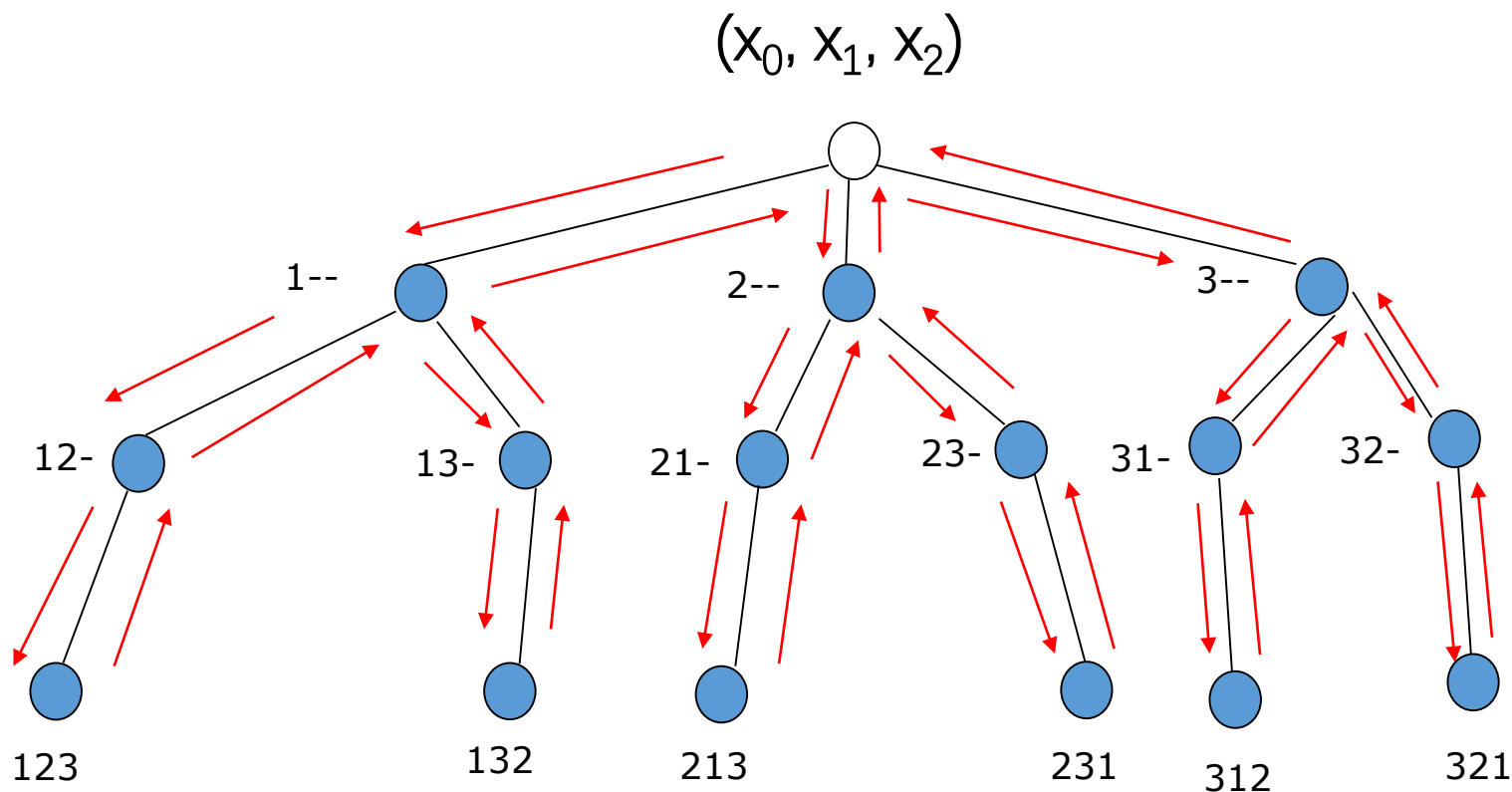
输出: 123 132 213 231 312 321



问题描述：输出n个数的全部排列,数字**不能重复**

输入: 1,2,3

输出: 123 132 213 231 312 321



# 回溯算法：递归实现

## 回溯算法的通式

```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            Search(k+1);  
        }  
}
```

```
int x[3]; //  $x[k]$ 代表 $x_k$   
void Search (int k) { // 决定 $x_k$   
    if (k > 2) // 找到一组解, 输出结果  
        cout<<x[0]<<x[1]<<x[2];  
    else  
        for (int i = 1; i <= 3; i++) {  
            if (check_feasible(i, k)) {  
                x[k] = i;  
                Search(k+1);  
            }  
        }  
}
```

判断 $i$ 是否可以作为 $x_k$ 的值

# 回溯算法：递归实现

//判断i是否可以作为 $x_k$ 的值

```
bool check_feasible(int i, int k) {  
    bool flag = true;  
    for(j=k-1; j>=0; j--) {  
        if(i == x[k-1]) {  
            flag = false;  
            break;  
        }  
    }  
    return flag;  
}
```

i如果和 $x[0] \dots x[k-1]$ 中的某个值重复，则不能作为 $x_k$ 的值

复杂度较高！

# 回溯算法：递归实现

```
int b[100]={0}; //b[i]用来标记i是否已经被用过, 0表示没用过, 1表示被用过
int x[3]; //x[k]代表 $x_k$ 
void Search (int k) { //决定 $x_k$ 
    if (k > 2) //找到一组解, 输出结果
        cout<<x[0]<<x[1]<<x[2];
    else
        for (int i = 1; i <= 3; i++) {
            if(b[i] == 0) { //0表示没用过
                x[k] = i;
                b[i] = 1; //标记i已经被 $x_k$ 用了
                Search(k+1);
                b[i] = 0; //回溯, 恢复b[i]的值
            }
        }
}
```

## 回溯算法的通式

```
void Search (int k) {
    if (已经找到一组解)
        输出解;
    else
        对 $x_k$ 每一个能够取的值s {
             $x_k = s$ ;
            添加相应的标记;
            Search(k+1);
            删除标记;
        }
}
```



# 回溯过程

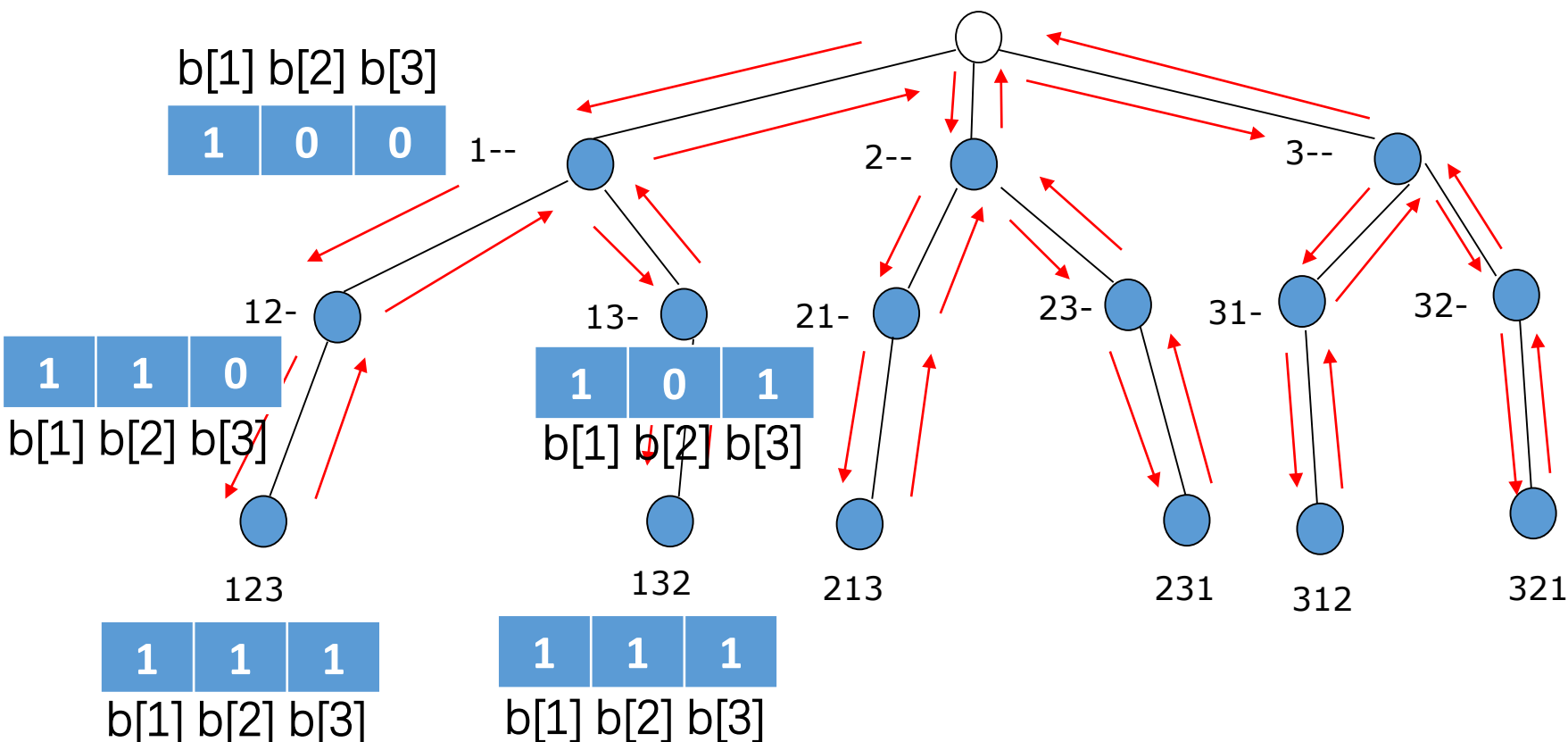
0	0	0
---	---	---

  
b[1] b[2] b[3]

数组b的初始值都为0，表示数字1,2,3都没被用过

b[1]	b[2]	b[3]
0	0	0

 ( $x_0, x_1, x_2$ )



# 完整程序

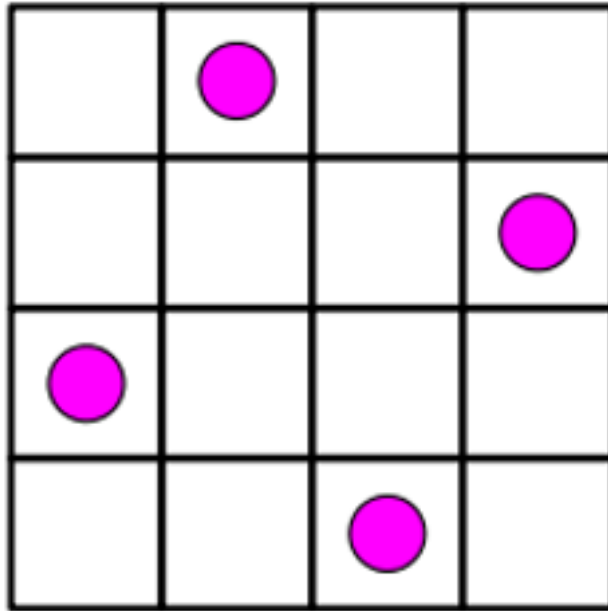
```
#include<iostream>
using namespace std;

int b[100]={0};
int x[3];
void Search (int k) {
    if (k > 2)
        cout<<x[0]<<x[1]<<x[2];
    else
        for (int i = 1; i <= 3; i ++) {
            if(b[i] == 0) {
                x[k] = i;
                b[i] = 1;
                Search(k+1);
                b[i] = 0;
            }
        }
}
```

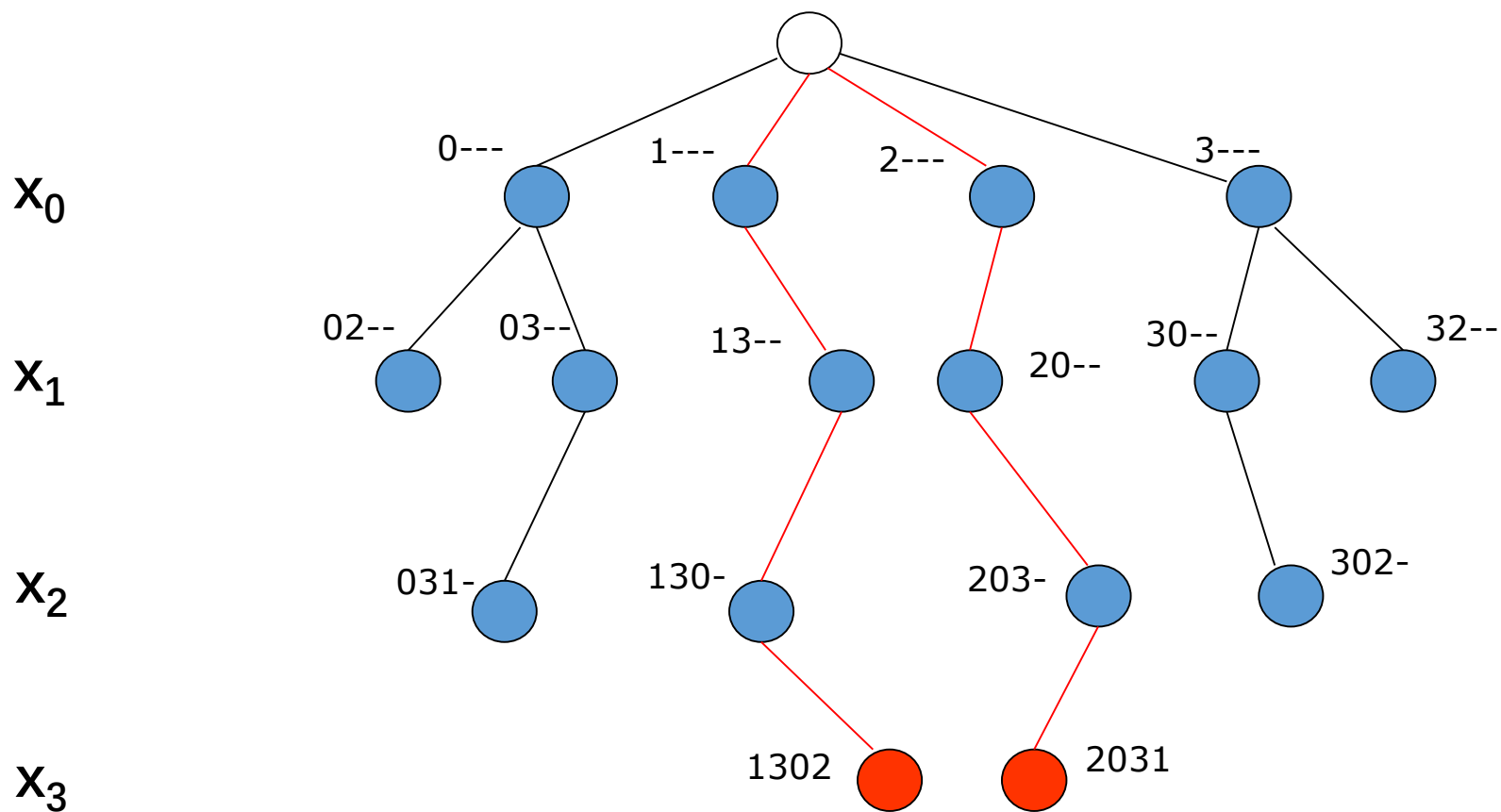
```
int main() {
    Search (0);
    return 0;
}
```

## 4皇后问题

在 $4 \times 4$ 的国际象棋棋盘上，放置4个皇后，使任何一个皇后都不能吃掉另一个，需满足的条件是：同一行、同一列、同一对角线上只能有一个皇后。求所有满足要求的放置方案。



**回溯解法：**  $(x_0, x_1, x_2, x_3)$  //  $x_i$  表示第  $i$  行的皇后放在哪一列



# 4皇后问题的回溯程序

## 回溯算法的通式

```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            Search(k+1);  
        }  
}
```

```
int x[4]; //x[i]表示第i行皇后放在哪一列  
void Search (int k) {  
    if (k > 3)  
        print(); //找到可行解, 打印棋盘  
    else  
        for (int i = 0; i < 4; i++) {  
            if(able_to_place(i, k)) {  
                x[k] = i;  
                Search(k+1);  
            }  
        }  
}
```

检查如果第 $k$ 行皇后放在第 $i$ 列，  
是否可以

## 4皇后问题的回溯程序

```
bool able_to_place(int i, int k) {  
    //遍历第k行之前的所有行，即x[0],...,x[k-1]  
    for (int j=0; j<k; j++) {  
        if (i==x[j]) { //如果在同一列，该位置不能放  
            return false;  
        }  
        //如果当前位置的右上/左下方有皇后，也不行  
        if ((j+x[j])== (k+i)) {  
            return false;  
        }  
        //如果当前位置的左上/右下方有皇后，也不行  
        if ((j-x[j])== (k-i)) {  
            return false;  
        }  
    }  
    return true; //如果以上情况都不是，当前位置就可以放皇后  
}
```

## 4皇后问题的回溯程序

```
void print() {  
    for (int line = 0; line < 4; line++) {  
        int i;  
        for (i = 0; i < x[line]; i++)  
            cout<<"0"; //没放皇后的位置打印字符0  
        cout<<"#"; //放皇后的位置打印字符#  
        for (i = x[line] + 1; i < 4; i++) {  
            cout<<"0"; //没放皇后的位置打印字符0  
        }  
        cout<<endl;  
    }  
    cout<<"=====\n";  
}
```

## 4皇后问题的回溯程序

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

int main() {
    Search(0); //从第0行开始放
    return 0;
}
```

主函数

```
0#00
000#
#000
00#0
=====
00#0
#000
000#
0#00
=====
```

输出结果



# 简化版本

## 回溯算法的通式

```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            添加相应的标记;  
            Search(k+1);  
            删除标记;  
        }  
}
```

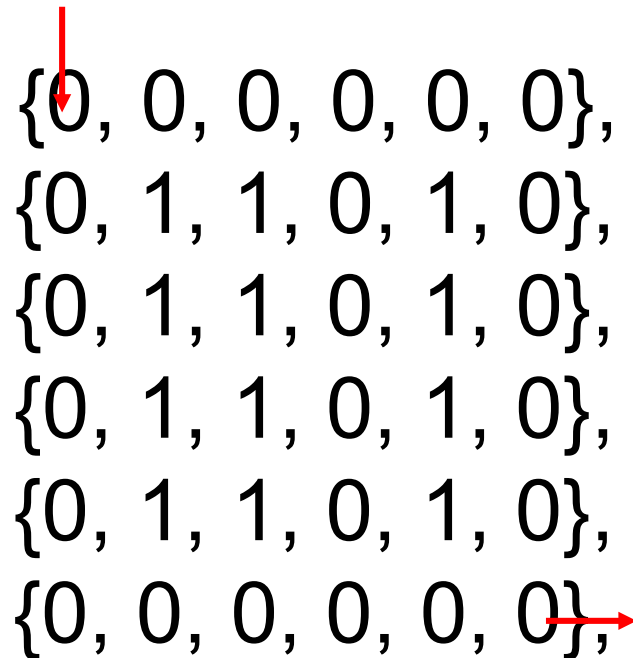
```
int x[4]; //x[i]表示第i行皇后放在哪一列  
int b[100]={0}; //b[j]标记第j列是否已被占用  
int c[100]={0}; //标记右上/左下对角线是否被占  
int d[100]={0}; //标记左上/右下对角线是否被占
```

```
void Search (int k) {  
    if (k > 3)  
        print(); //找到可行解, 打印棋盘  
    else  
        for (int i = 0; i < 4; i++) {  
            if((!b[i]) && (!c[i+k]) && (!d[k-i+3])) {  
                x[k] = i;  
                b[i] = 1; c[i+k] = 1; d[k-i+3] = 1;  
                Search(k+1);  
                b[i] = 0; c[i+k] = 0; d[k-i+3] = 0;  
            }  
        }  
}
```

## 走迷宫 (Maze)

已知一 $N \times N$ 的迷宫，允许往上、下、左、右四个方向行走，现请你**找出所有**从左上角（起点）到右下角（终点）的路径

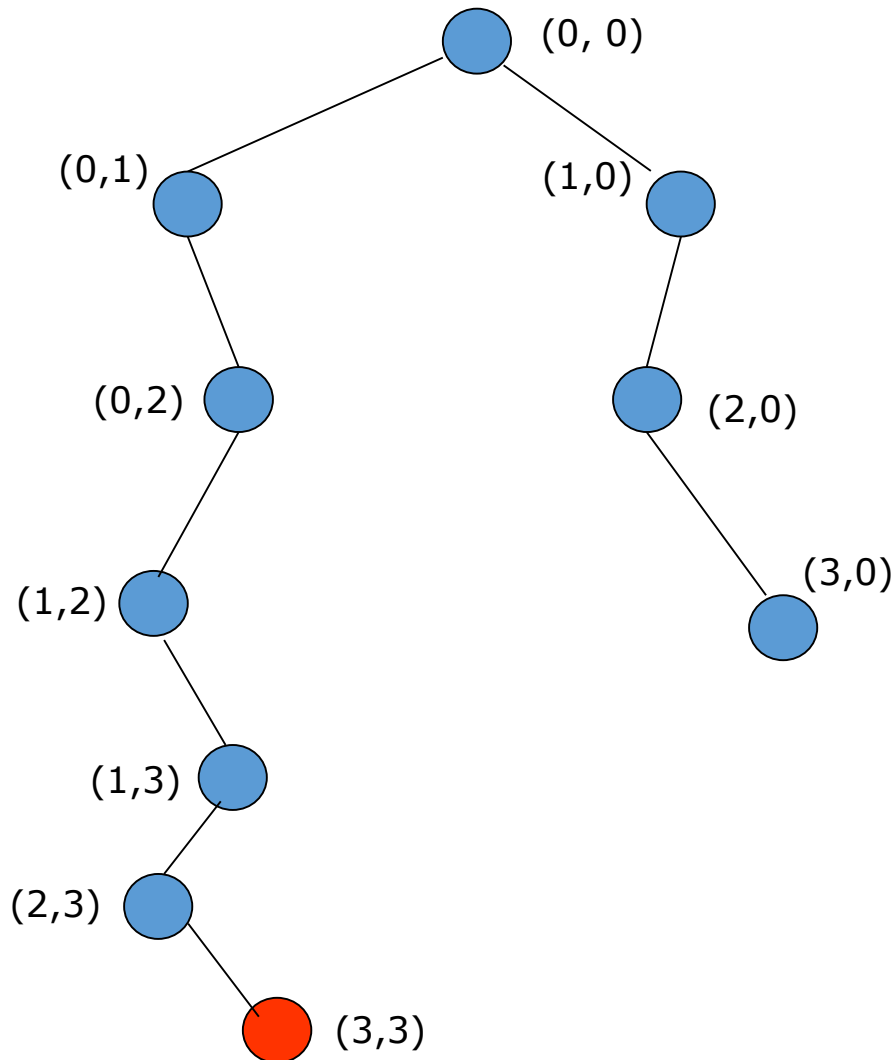
输入数据有 $N$ 行，每行有 $N$ 个0或1 (0表示可以通过，1表示不能通过)，用以描述迷宫地图。入口在左上角  $(0, 0)$  处，出口在右下角  $(N-1, N-1)$  处



```
{0, 0, 0, 0, 0, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 0, 0, 0, 0, 0}
```

## 回溯解法

$(x_0, x_1, x_2, x_3 \dots)$  //  $x_k$  表示第  $k$  步走到哪个位置，位置用  $(i, j)$  表示，  
 $i$  表示行， $j$  表示列



~~$\{0, 0, 0, 1\}$ ,~~  
 ~~$\{0, 1, 0, 0\}$ ,~~  
 $\{0, 1, 1, 0\}$ ,  
 $\{0, 1, 1, 0\}$  →

# 思路

从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止，如果所有可能的通路都试探过，还是不能走到终点，那就说明该迷宫不存在从起点到终点的通道。

**深度优先搜索解空间！**

## 回溯解法

$(x_0, x_1, x_2, x_3 \dots)$  //  $x_k$  表示第  $k$  步走到哪个位置，位置用  $(i, j)$  表示，  
 $i$  表示行， $j$  表示列

//定义结构体，表示迷宫一个具体位置

```
struct Pos {  
    int i; //表示所在的行  
    int j; //表示所在的列  
};
```

Pos x[1000]; //x[k]表示第k步走到哪个位置

# 迷宫问题的回溯程序

## 6X6迷宫

### 回溯算法的通式

```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            Search(k+1);  
        }  
}
```

```
void Search(int k) { // 决定第k步要走到哪个位置  
    if(x[k-1].i == 5 && x[k-1].j == 5) {  
        print(k); // 输出路径  
    } else {  
        // 四个方向分别用t = 1, 2, 3, 4表示  
        for(int t = 1; t <= 4; t++) {  
            if(able_to_go(k, t)) { // 如果可以往t方向  
                Pos cur = x[k-1]; // 第k步要走到哪  
                if(t == 1) // 往右  
                    cur.j += 1;  
                if(t == 2) // 往下  
                    cur.i += 1;  
                if(t == 3) // 往左  
                    cur.j -= 1;  
                if(t == 4) // 往上  
                    cur.i -= 1;  
                x[k] = cur; // 确定了第k步的位置  
                Search(k+1); // 决定第k+1步  
            }  
        }  
    }  
}
```

# 迷宫问题的回溯程序

//判断第k步是否可以往t方向走

```
bool able_to_go(int k, int t) {
```

```
    Pos last = x[k-1];//第k-1步的位置
```

```
    Pos cur = last;//第k步要走到的位置，从k-1步位置开始走
```

```
    if(t == 1) {//往右
```

```
        if (last.j+1 > 5 || maze[last.i][last.j+1] != 0) {  
            return false;
```

```
        }
```

```
        cur.j +=1;
```

```
    }
```

```
    if(t == 2) {//往下
```

```
        if ((last.i+1 > 5) || (maze[last.i+1][last.j] != 0)) {  
            return false;
```

```
        }
```

```
        cur.i += 1;
```

```
    }
```

# 迷宫问题的回溯程序

```
if(t == 3) {//往左
    if (last.j-1 < 0 || maze[last.i][last.j-1] != 0) {
        return false;
    }
    cur.j -= 1;
}
```

```
if(t == 4) {//往上
    if (last.i-1 < 0 || maze[last.i-1][last.j] != 0) {
        return false;
    }
    cur.i -= 1;
}
```

**//x[k]不能和x[0],x[1],...,x[k-1]重复**

```
for(int m = 0; m < k; m++) {
    if(cur.i == x[m].i && cur.j == x[m].j)
        return false;
}
```

**return true;****//满足上面所有条件，可以往i方向走**

```
}
```



# 迷宫问题的回溯程序

```
//打印找到的路径，即x[0],x[1],...,x[k-1]
void print(int k) {
    for(int m = 0; m < k; m++) {
        cout<<x[m].i<<" "<<x[m].j<<endl;
    }
    cout<<endl;
}
```

# 迷宫问题的完整程序

```
#include<iostream>
using namespace std;
//迷宫
int maze[6][6] = {
{0,0,0,0,0,1},
{0,1,1,0,1,1},
{0,1,1,0,1,0},
{0,1,1,0,1,0},
{0,1,1,0,0,0},
{0,0,0,0,1,0}
};
int main() {
    Pos start = {0,0};//起点
    x[0] = start;//把x[0]置为起点
    Search(1);//从第一步开始搜索
    return 0;
}
```



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

输出结果

# 简化版本

## 回溯算法的通式

```
void Search (int k) {  
    if (已经找到一组解)  
        输出解;  
    else  
        对 $x_k$ 每一个能够取的值 $s$  {  
             $x_k = s$ ;  
            添加相应的标记;  
            Search(k+1);  
            删除标记;  
        }  
}
```

```
void Search(int k) { // 决定第k步要走到哪个位置  
    if(x[k-1].i == 5 && x[k-1].j == 5) {  
        print(k); // 输出路径  
    } else {  
        for(int t = 1; t <= 4; t++) {  
            if(able_to_go(k, t)) { // 如果可以往t方向  
                Pos cur = x[k-1]; // 上一步的位置  
                if(t == 1) // 往右  
                    cur.j += 1;  
                if(t == 2) // 往下  
                    cur.i += 1;  
                if(t == 3) // 往左  
                    cur.j -= 1;  
                if(t == 4) // 往上  
                    cur.i -= 1;  
                x[k] = cur; // 确定了第k步的位置  
                maze[cur.i][cur.j] = 2;  
                Search(k+1);  
                maze[cur.i][cur.j] = 0;  
            }  
        }  
    }  
}
```

标记迷宫这个位置已经走过

递归回来删除标记

# 简化版本

//判断第k步是否可以往t方向走

```
bool able_to_go(int k, int t) {
```

```
    Pos last = x[k-1]; //上一步位置
```

```
    Pos cur = last; //这一步要走到的位置
```

```
    if(t == 1) { //往右... }
```

```
    if(t == 2) { //往下... }
```

```
    if(t == 3) { //往左... }
```

```
    if(t == 4) { //往上... }
```

```
    //x[k]不能和x[0],x[1],...,x[k-1]重复
```

```
    for(int m = 0; m < k; m++) {
```

```
        if(cur.i == x[m].i && cur.j == x[m].j)
```

```
            return false;
```

```
    }
```

```
    return true; //满足上面所有条件，可以往i方向走
```

```
}
```

可以删掉，因为已经把走过的路线标记为2，不会再走重复的路线

## 走迷宫 (Maze)

已知一 $N \times N$ 的迷宫，允许往上、下、左、右四个方向行走，**最短路径**的长度是多少？

```
{0, 0, 0, 0, 0, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 1, 1, 0, 1, 0},  
{0, 0, 0, 0, 0, 0},
```

# 回溯程序

```
int shortest_path = 6*6; //记录目前为止最短路径长度

void Search(int k) { //决定第k步要走到哪个位置
    if(x[k-1].i == 5 && x[k-1].j == 5) {
        //新找到的路径的长度为k
        if(shortest_path > k+1)
            shortest_path = k+1; //如果新找到的路径更短，更新
    } else {
        .....
    }
}
```

# 迷宫问题的完整程序

```
#include<iostream>
using namespace std;
int shortest_path = 36;
int maze[6][6] = {
    {0,0,0,0,0,1},
    {0,1,1,0,1,1},
    {0,1,1,0,1,0},
    {0,1,1,0,1,0},
    {0,1,1,0,0,0},
    {0,0,0,0,1,0} };
int main() {
    Pos start = {0,0}; // 起点
    x[0] = start; // 将x[0]置为起点
    Search(1); // 从第一步开始
    cout<<shortest_path<<endl;
    return 0;
}
```



11

输出结果

# 更高效的方案？

```
int shortest_path = 6*6; //记录目前为止最短路径长度
```

```
void Search(int k) { //决定第k步要走到哪个位置
```

```
    if(k+1 > shortest_path)
        return; //剪枝操作
```

```
    if(x[k-1].i == 5 && x[k-1].j == 5) {
```

```
        //新找到的路径的长度为k
```

```
        if(shortest_path > k+1)
```

```
            shortest_path = k+1;
```

```
    }
```

```
    else {
```

```
        .....
```

```
    }
```

```
}
```

如果搜索过程中，发现k+1已经大于shortest\_path了，可以跳过正在搜索的路径

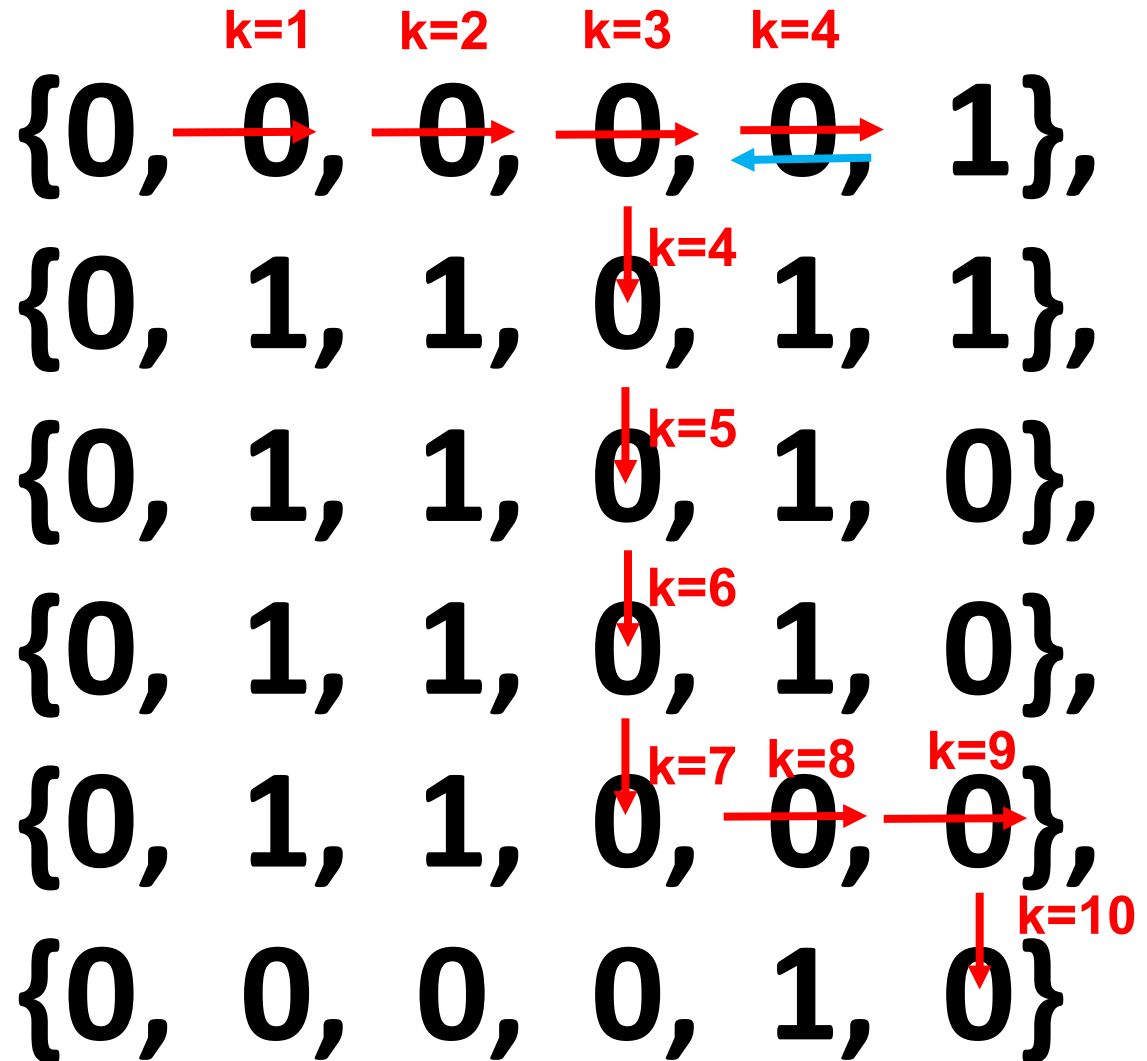


# 深度优先搜索过程

shortest\_path=36



shortest\_path=11

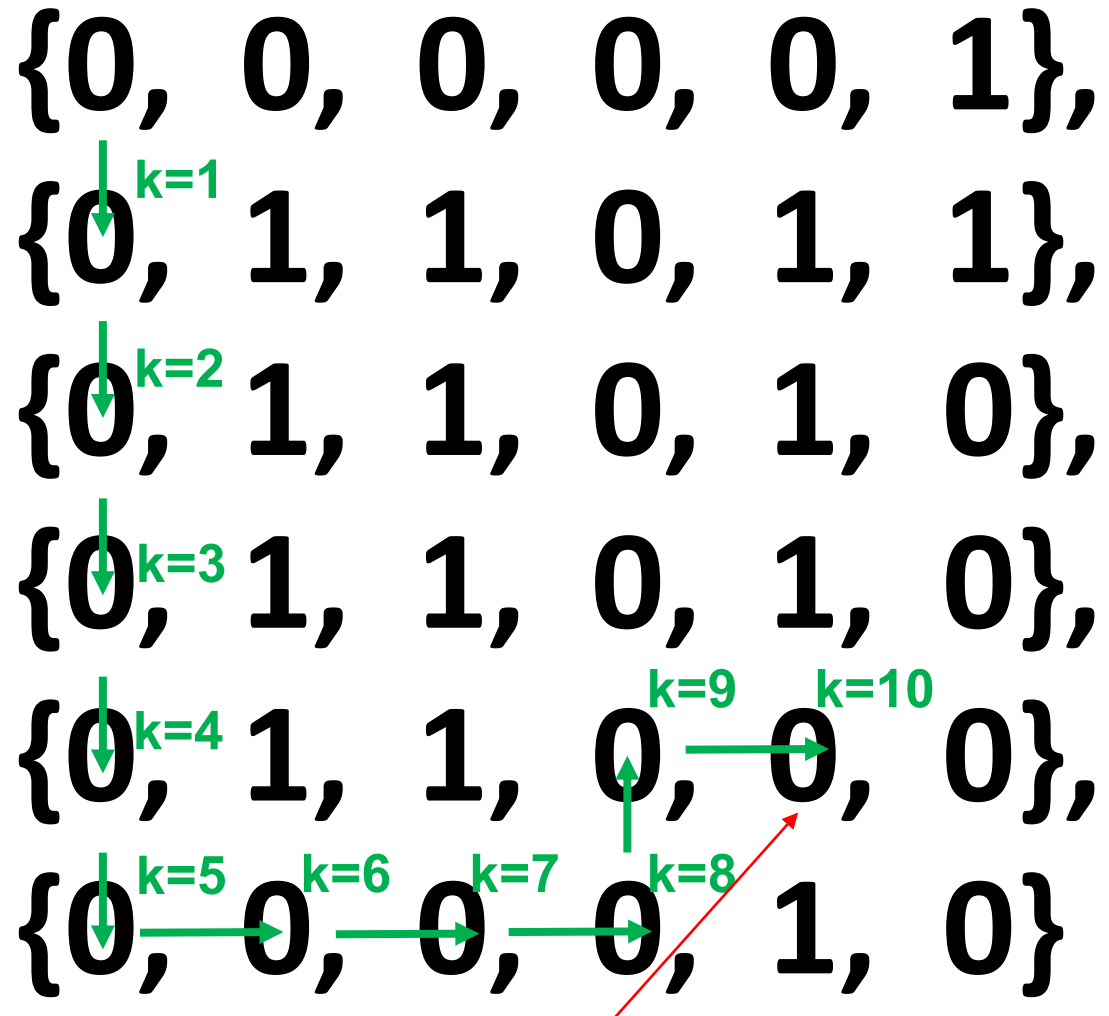


# 深度优先搜索过程

shortest\_path=36



shortest\_path=11



停止，因为 $k+1$ 已经超过了shortest\_path

# 深度优先(DFS)与广度优先(BFS)

迷宫最短路问题，m行n列迷宫，S表示起点，E表示终点，+表示不通，-表示通路，求S到E的最短路径长度

+	+	+	+	+	+
+	E	-	-	-	+
+	-	-	-	-	+
+	-	-	-	-	+
+	+	+	+	S	+

# 深度优先搜索(DFS)

## 思路

定义数组 $b[m][n]$ ， $b[i][j]$ 表示从S到 $(i,j)$ 的最短距离；初始时， $b[S_x][S_y]$ 为0（S到自己的最短路为0），其余 $b[i][j]$ 为无穷大；深度优先遍历所有路径，每走一步，更新所到达节点的 $b[i][j]$ （将路径长度+1），直到所有路径搜索完， $b[E_x][E_y]$ 就是S到E的最短路径。

+	+	+	+	+	+
+	E	-	-	-	+
+	-	-	-	-	+
+	-	-	-	-	+
+	+	+	+	S	+

$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	0	$\infty$

# 深度优先(DFS)搜索：下，上，右，左的顺序

+	-	-	-	$\infty$	$\infty$	$\infty$	$\infty$
+	E	+	-	$\infty$	$\infty$	$\infty$	$\infty$
+	-	-	-	$\infty$	$\infty$	$\infty$	$\infty$
+	+	+	S	$\infty$	$\infty$	$\infty$	0



+	-	-	-	$\infty$	$\infty$	$\infty$	$\infty$
+	E	+	-	$\infty$	$\infty$	$\infty$	$\infty$
+	-	-		$\infty$	$\infty$	$\infty$	1
+	+	+	S	$\infty$	$\infty$	$\infty$	0



+	-	-		$\infty$	$\infty$	$\infty$	3
+	E	+		$\infty$	$\infty$	$\infty$	2
+	-	-		$\infty$	$\infty$	$\infty$	1
+	+	+	S	$\infty$	$\infty$	$\infty$	0



+	-	-	-	$\infty$	$\infty$	$\infty$	$\infty$
+	E	+		$\infty$	$\infty$	$\infty$	2
+	-	-		$\infty$	$\infty$	$\infty$	1
+	+	+	S	$\infty$	$\infty$	$\infty$	0

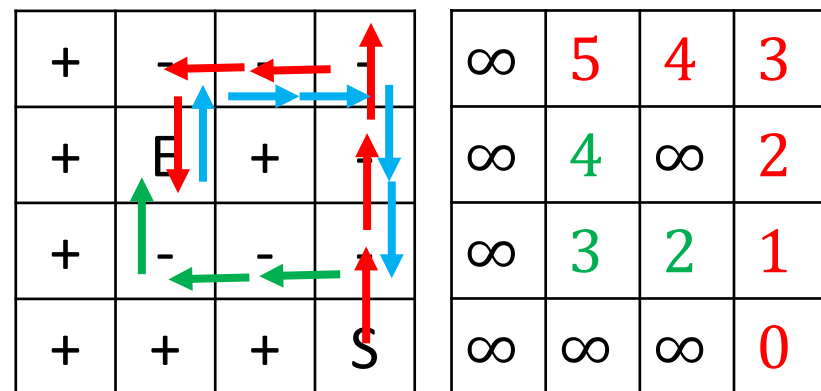
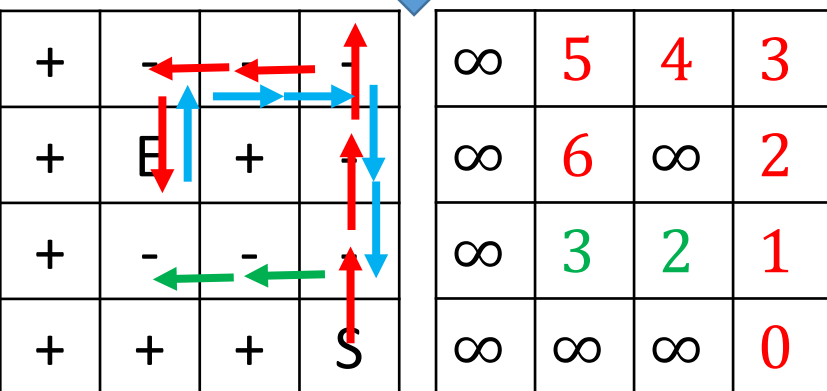
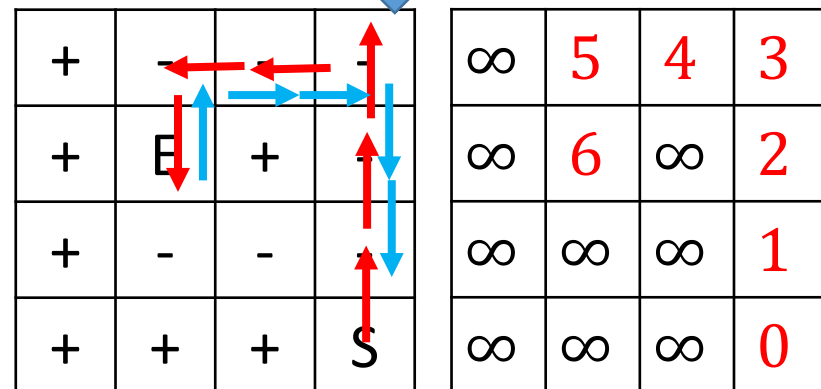
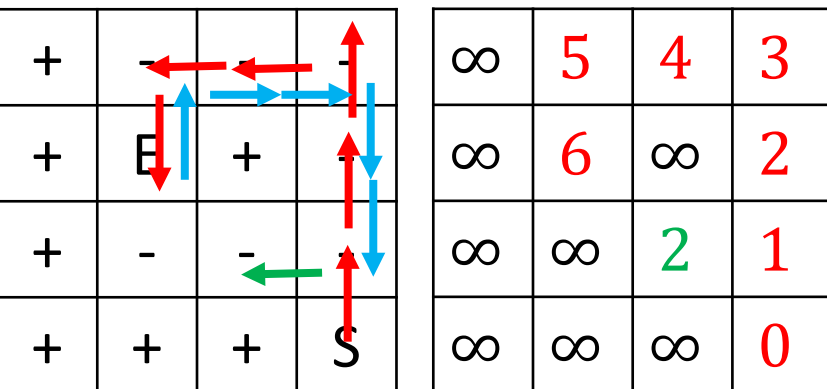
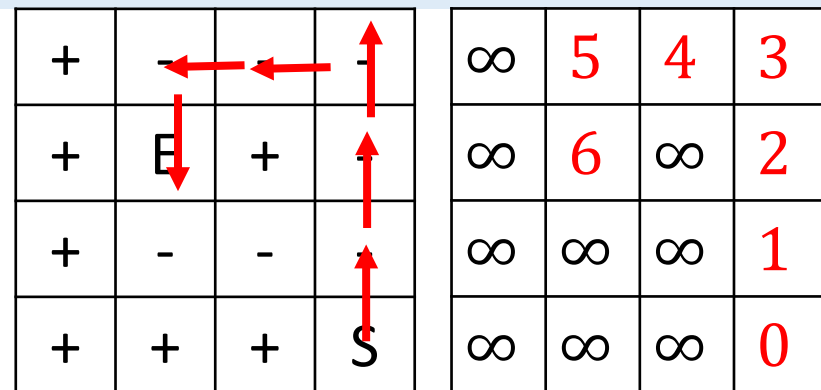
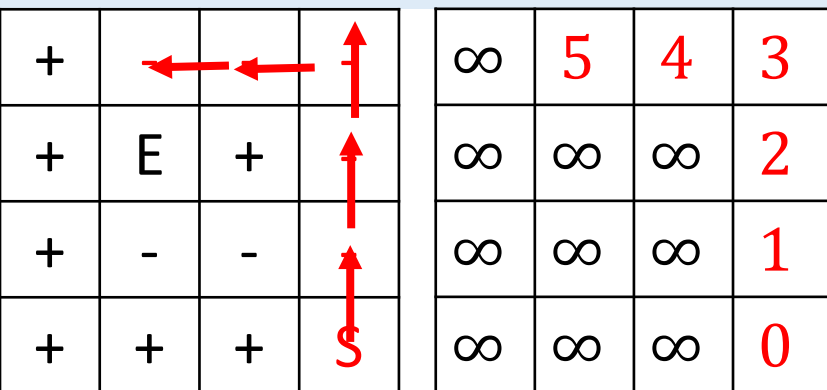


+	-			$\infty$	$\infty$	4	3
+	E	+		$\infty$	$\infty$	$\infty$	2
+	-	-		$\infty$	$\infty$	$\infty$	1
+	+	+	S	$\infty$	$\infty$	$\infty$	0



+				$\infty$	5	4	3
+	E	+		$\infty$	$\infty$	$\infty$	2
+	-	-		$\infty$	$\infty$	$\infty$	1
+	+	+	S	$\infty$	$\infty$	$\infty$	0

# 深度优先(DFS)搜索



# 深度优先搜索(DFS)

```
#include<iostream>
#include<cstring>
using namespace std;
char a[101][101];//存迷宫，+号表示障碍，-表示通路，S表示入口，
E表示出口
int b[101][101];//b[i][j]存储从入口到a[i][j]的最短路长度
int m,n,x_e,y_e,x_s,y_s;//m行n列；xe: 出口行坐标，ye: 出口
列坐标，xs: 入口行坐标，ys: 入口列坐标
void dfs(int x, int y, int z)//DFS，z表示从起点到(x,y)的最短路径
{
    if(x>m||x<1||y>n||y<1||a[x][y]=='+'||b[x][y]<=z) return;
    b[x][y] = z;//更新最短路
    dfs(x+1,y,z+1);//往下走
    dfs(x-1,y,z+1);//往上走
    dfs(x,y+1,z+1);//往右走
    dfs(x,y-1,z+1);//往左走
}
```

# 深度优先搜索(DFS)

```
int main() {  
    cin>>m>>n;  
    for(int i=1; i<=m; i++)  
        for(int j=1; j<=n; j++){  
            cin>>a[i][j];  
            if(a[i][j] == 'S') x_s=i, y_s=j;  
            if(a[i][j] == 'E') x_e=i, y_e=j;  
        }//输入迷宫  
    for(int i=1; i<=m; i++)  
        for(int j=1; j<=n; j++)  
            b[i][j]=1e9;//将每个点与起点的最短路置为无穷大  
    dfs(xs,ys,0);//从起点开始走  
    if(b[x_e][y_e]==1e9) cout<<"WRONG";  
    else  
        cout<<b[x_e][y_e];//输出终点到起点的最短路  
    return 0;  
}
```



# 广度优先搜索(BFS)

## 思路

定义数组 $b[m][n]$ ,  $b[i][j]$ 表示从S到 $(i,j)$ 的最短距离; 初始时,  $b[S_x][S_y]$ 为0 (S到自己的最短路为0)

对于起点每个可达的位置 $(i,j)$ , 更新 $b[i][j]$ 为1

对于每个位置 $(i,j)$ 每个可达的位置 $(i1,j1)$ , 更新 $b[i1][j1]$ 为2

...

以此类推, 直到到达终点E

+	+	+	+	+	+
+	E	-	-	-	+
+	-	-	-	-	+
+	-	-	-	-	+
+	+	+	+	S	+

$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	$\infty$	0	$\infty$

# 广度优先搜索(BFS)

+	-	-	-	∞	∞	∞	∞
+	E	-	-	∞	∞	∞	∞
+	-	-	-	∞	∞	∞	∞
+	+	+	S	∞	∞	∞	0



+	-	-	-	∞	∞	∞	∞
+	E	-	-	∞	∞	∞	∞
+	-	-		∞	∞	∞	1
+	+	+	S	∞	∞	∞	0



+	-	-		∞	∞	∞	3
+	E			∞	∞	3	2
+	-			∞	∞	2	1
+	+	+	S	∞	∞	∞	0



+	-	-	-	∞	∞	∞	∞
+	E	-		∞	∞	∞	2
+	-			∞	∞	2	1
+	+	+	S	∞	∞	∞	0



+	-	-		∞	∞	∞	3
+	E			∞	∞	3	2
+				∞	3	2	1
+	+	+	S	∞	∞	∞	0



+	-			∞	∞	4	3
+	E			∞	∞	3	2
+				∞	3	2	1
+	+	+	S	∞	∞	∞	0

# 广度优先搜索(BFS)

```
定义数组b[m][n]; //b[i][j]表示S到(i,j)的最短距离
将b[S.x][S.y]初始化为0, 其余的b[i][j]置无穷大
定义队列Q;
将S放入Q;
while(Q不空) {
    取出队列首元素q;
    如果q是终点E, 程序结束, 输出b[E.x][E.y];
    对于每个q能到达的位置a //更新b[a.x][a.y]
        b[a.x][a.y] = min{b[q.x][q.y]+1, b[a.x][a.y]};
}
```