搜索

信息竞赛组

广州大学附属中学

2024 年 8 月

目录

搜索算法

- 搜索算法是利用计算机高性能、快速运算的特点有目的地穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。

搜索算法

- 搜索算法是利用计算机高性能、快速运算的特点有目的地穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。
- 搜索算法从问题的**初始状态出发**，根据其中的约束条件，按照一定的策略**有序推进**，不断深入，对于达到的所有状态空间进行的一一验证，直到找到符合条件的**目标状态**（解空间），这个目标状态就是问题的**可行解或可行解中的最优解**。

目录

顺序搜索

- 从第一个数据开始，按数据的顺序逐个将数据与给定的值进行比较。若某个数据和给定的值相等，则查找成功，找到所查数据的位置；反之，查找不成功。

顺序搜索

- 从第一个数据开始，按数据的顺序逐个将数据与给定的值进行比较。若某个数据和给定的值相等，则查找成功，找到所查数据的位置；反之，查找不成功。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`

顺序搜索

- 从第一个数据开始，按数据的顺序逐个将数据与给定的值进行比较。若某个数据和给定的值相等，则查找成功，找到所查数据的位置；反之，查找不成功。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`
- 查找目标：`target = 15`

顺序搜索

- 从第一个数据开始，按数据的顺序逐个将数据与给定的值进行比较。若某个数据和给定的值相等，则查找成功，找到所查数据的位置；反之，查找不成功。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`
- 查找目标：`target = 15`
- 如何查找到 target 在数组 a 中的位置？

给定数组：int a[] = {1, 3, 4, 11, 15, 25, 63, 22}，返回 target 所在位置，若找不到，返回 −1。

```
#include <bits/stdc++.h>
using namespace std;
int a[] = {1, 3, 4, 11, 15, 25, 63, 22};
int target;
int ans = -1;
int main() {
    cin >> target;
    for (int i = 0; i < 8; i++) {
        if (target == a[i]) {
            ans = i;
        }
    }
    cout << ans << endl;
    return 0;
}
```

目录

二分搜索

- 二分搜索是一种效率很高的搜索算法，但是被搜索的数据必须是有序的。

二分搜索

- 二分搜索是一种效率很高的搜索算法，但是被搜索的数据必须是有序的。
- 基本思想：首先把待查数据与数组中间位置的数据进行比较，待查数据比中间位置的数据大，在数组的后半部分继续搜索，否则，在数组的前半部分搜索，继续二分搜索，直到找到待查数据在数组中的位置或数组已无法对分。

二分搜索

- 二分搜索是一种效率很高的搜索算法，但是被搜索的数据必须是有序的。
- 基本思想：首先把待查数据与数组中间位置的数据进行比较，待查数据比中间位置的数据大，在数组的后半部分继续搜索，否则，在数组的前半部分搜索，继续二分搜索，直到找到待查数据在数组中的位置或数组已无法对分。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`

二分搜索

- 二分搜索是一种效率很高的搜索算法，但是被搜索的数据必须是有序的。
- 基本思想：首先把待查数据与数组中间位置的数据进行比较，待查数据比中间位置的数据大，在数组的后半部分继续搜索，否则，在数组的前半部分搜索，继续二分搜索，直到找到待查数据在数组中的位置或数组已无法对分。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`
- 查找目标：`target = 15`

二分搜索

- 二分搜索是一种效率很高的搜索算法，但是被搜索的数据必须是**有序**的。
- 基本思想：首先把待查数据与**数组中间位置的数据进行比较**，待查数据比**中间位置的数据大，在数组的后半部分继续搜索，否则，在数组的前半部分搜索**，继续二分搜索，**直到找到待查数据在数组中的位置或数组已无法对分**。
- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`
- 查找目标：`target = 15`
- 如何使用**二分搜索**查找到 target 在数组 a 中的位置？

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序
  2. 规定左边界（left）和右边界（right）范围

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序
  2. 规定左边界（left）和右边界（right）范围
  3. 循环查找，若中间值为结果，结束循环；若待查数据大于中间值，更新左边界（left）；若待查数据小于中间值，更新右边界（right）

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序
  2. 规定左边界（left）和右边界（right）范围
  3. 循环查找，若中间值为结果，结束循环；若待查数据大于中间值，更新左边界（left）；若待查数据小于中间值，更新右边界（right）
- 核心代码：

- 给定数组：`int a[] = {1, 3, 4, 11, 15, 25, 63, 22}`，返回 target 所在位置，若找不到，返回 -1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序
  2. 规定左边界（left）和右边界（right）范围
  3. 循环查找，若中间值为结果，结束循环；若待查数据大于中间值，更新左边界（left）；若待查数据小于中间值，更新右边界（right）
- 核心代码：

- 给定数组：int a[] = {1, 3, 4, 11, 15, 25, 63, 22}，返回 target 所在位置，若找不到，返回 −1。
- 要求使用**二分搜索**算法。
- 算法框架：
  1. 输入数组和待查数据，数组必须有序
  2. 规定左边界（left）和右边界（right）范围
  3. 循环查找，若中间值为结果，结束循环；若待查数据大于中间值，更新左边界（left）；若待查数据小于中间值，更新右边界（right）
- 核心代码：

```
while(left <= right) {
    int mid = (left + right) / 2;
    if (a[mid] == target) return mid;
    else if (a[mid] < target) left = mid + 1;
    else right = mid - 1;
}
```

二分搜索

————算法实现

```cpp
#include <bits/stdc++.h>
using namespace std;
int a[] = {1, 3, 4, 11, 15, 25, 63, 22};
int target;
int ans = -1;
int main() {
    cin >> target;
    int left = 0, right = 7;
    while(left <= right) {
        int mid = (left + right) / 2;
        if (a[mid] == target) {
            ans = mid;
            break;
        }
        else if (a[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    cout << ans << endl;
    return 0;
}
```

目录

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。
- 深度优先搜索的步骤：

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。
- 深度优先搜索的步骤：
  1. 访问顶点 v

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。
- 深度优先搜索的步骤：
  1. 访问顶点 v
  2. 依次从 v 未被访问的邻接点出发，对图进行深度优先遍历

深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。
- 深度优先搜索的步骤：
  1. 访问顶点 v
  2. 依次从 v 未被访问的邻接点出发，对图进行深度优先遍历
  3. 若有节点未被访问，则回溯到该节点，继续进行深度优先遍历

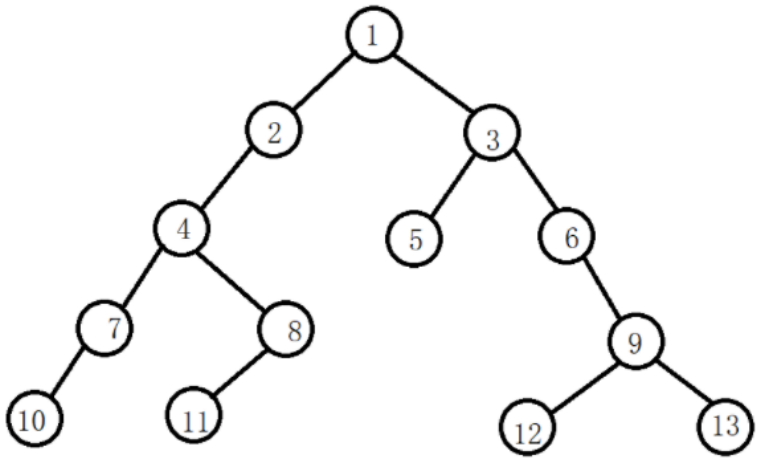
深度优先搜索

- 深度优先搜索算法（Depth-First-Search，dfs），是最简便的图的搜索算法之一。
- 其核心思想是：从一个**顶点 V**开始，沿着一条路**一直走到底**，如果发现**不能**到达目标解，那就**返回**到上一个节点，然后从另一条路开始走到底，这种**尽量往深处走的概念**即是深度优先的概念。
- 深度优先搜索的步骤：
  1. 访问顶点 v
  2. 依次从 v 未被访问的邻接点出发，对图进行深度优先遍历
  3. 若有节点未被访问，则回溯到该节点，继续进行深度优先遍历
  4. 直到所有与顶点 A 路径想通的节点都被访问过一次

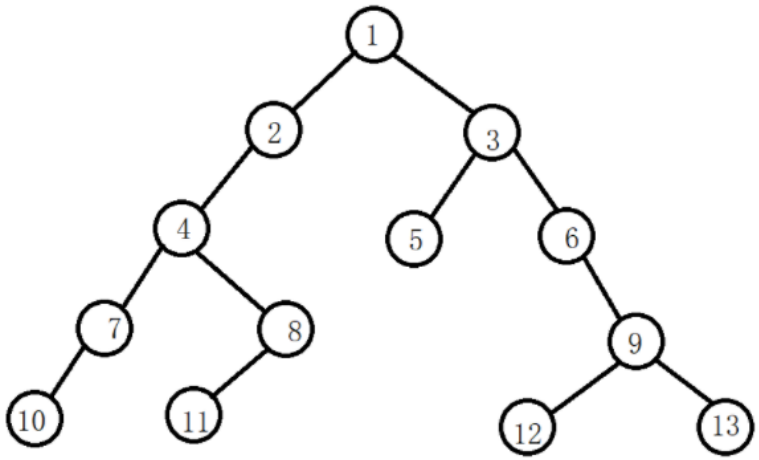
深度优先搜索

- 使用深度优先搜索对下面这幅图进行遍历，假设我们规定先遍历左边再遍历右边。从根节点 1 开始遍历。

- 使用深度优先搜索对下面这幅图进行遍历，假设我们规定先遍历左边再遍历右边。从根节点 1 开始遍历。



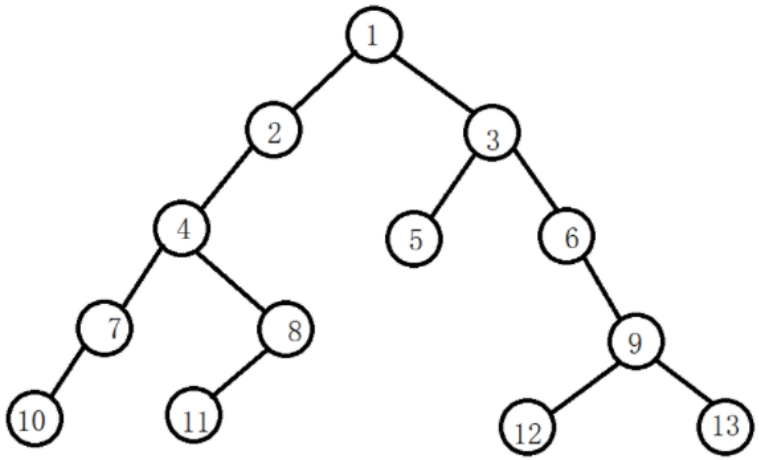
- 使用深度优先搜索对下面这幅图进行遍历，假设我们规定先遍历左边再遍历右边。从根节点 1 开始遍历。



- 它的遍历顺序为？

深度优先搜索

- 使用深度优先搜索对下面这幅图进行遍历，假设我们规定先遍历左边再遍历右边。从根节点 1 开始遍历。

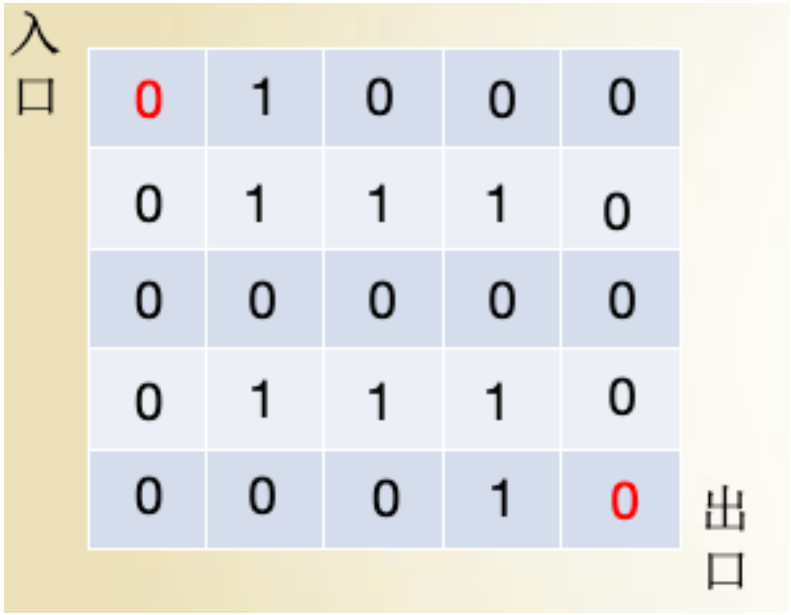


- 它的遍历顺序为？
- $1 \to 2 \to 4 \to 7 \to 10 \to 8 \to 11 \to 3 \to 5 \to 6 \to 9 \to 12 \to 13$

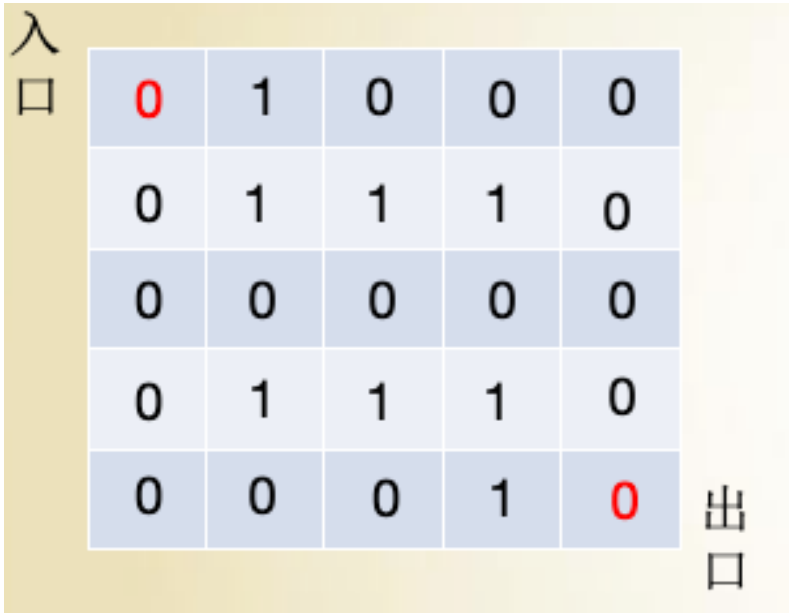
深度优先搜索

- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。

- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。



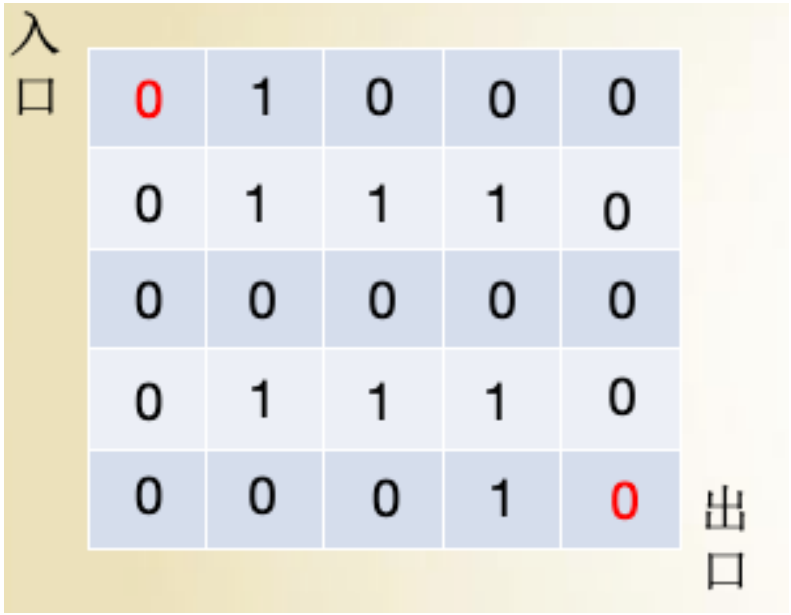
- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。



- 入口、出口坐标 → 基本变量 (int)

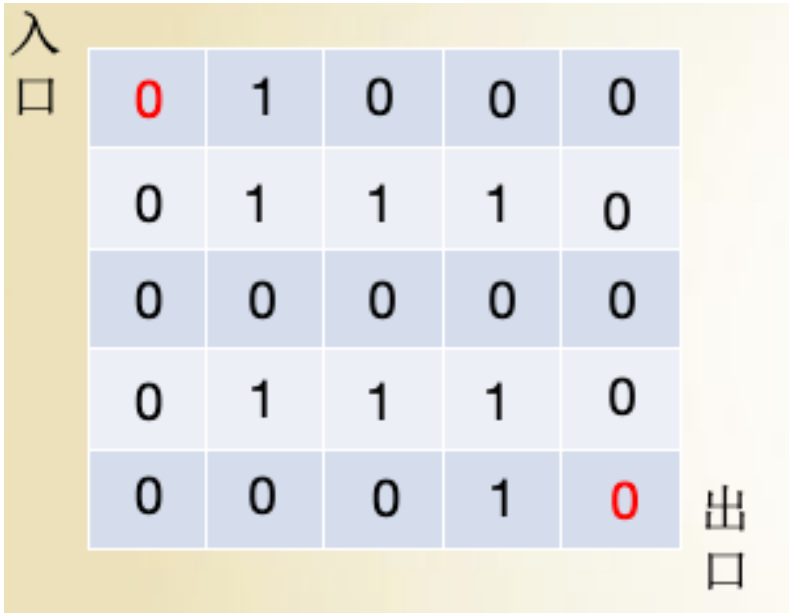
深度优先搜索

- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。



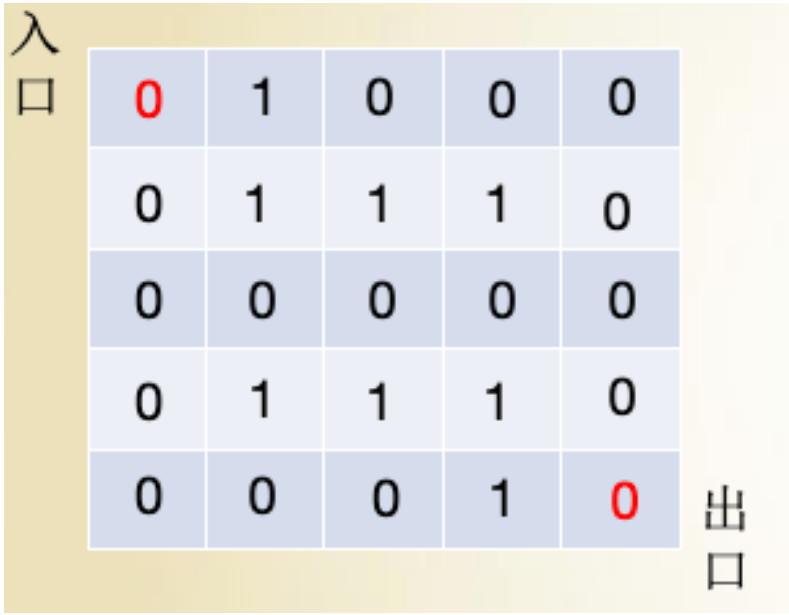
- 入口、出口坐标 → 基本变量 (int)
- 迷宫的环境 → 二维数组

- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。



- 入口、出口坐标 → 基本变量 (int)
- 迷宫的环境 → 二维数组
- 经过哪些位置 → 二维数组

- 下面是一个迷宫，0 表示可通过，1 表示障碍，不可通行。左上角为入门，右下角为出口。



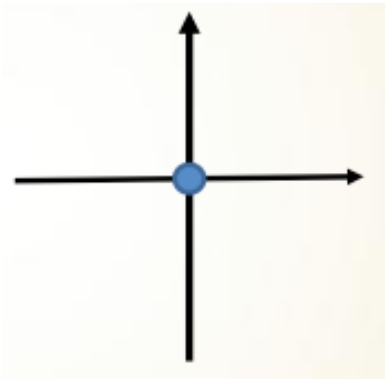
- 入口、出口坐标 → 基本变量 (int)
- 迷宫的环境 → 二维数组
- 经过哪些位置 → 二维数组
- 当前所在的位置 → 基本变量 (int)

深度优先搜索

入口

0	1	0	0	0
0	1	1	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

出口



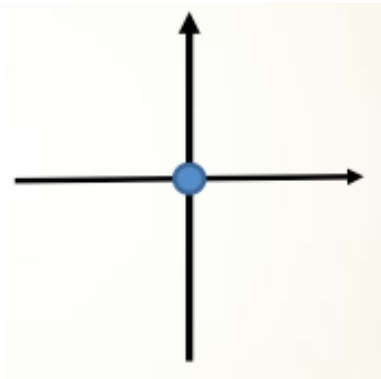
```
1  // 方向数组
2  int dx[4][2] = {
3      {-1,  0}, // 上
4      {1,   0}, // 下
5      {0,  -1}, // 左
6      {0,   1}, // 右
7  };
```

```
1  int x = 0, y = 0;
2  // 遍历四个方向
3  for (int i = 0; i < 4; i++) {
4      int nx = x + dx[i][0];
5      int ny = y + dy[i][1];
6  }
```

入口

0	1	0	0	0
0	1	1	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

出口



- 如何表示我们移动的方向？

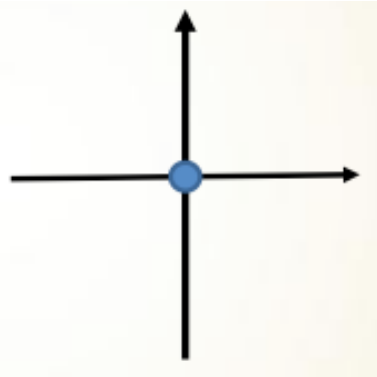
```
1  // 方向数组
2  int dx[4][2] = {
3      {-1,  0}, // 上
4      {1,   0}, // 下
5      {0,  -1}, // 左
6      {0,   1}, // 右
7  };
```

```
1  int x = 0, y = 0;
2  // 遍历四个方向
3  for (int i = 0; i < 4; i++) {
4      int nx = x + dx[i][0];
5      int ny = y + dy[i][1];
6  }
```

入口

0	1	0	0	0
0	1	1	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

出口



- 如何表示我们移动的方向?
- (x, y) 表示当前位置, (nx, ny) 表示下一步位置

```
1  // 方向数组
2  int dx[4][2] = {
3      {-1,  0}, // 上
4      {1,   0}, // 下
5      {0,  -1}, // 左
6      {0,   1}, // 右
7  };
```

```
1  int x = 0, y = 0;
2  // 遍历四个方向
3  for (int i = 0; i < 4; i++) {
4      int nx = x + dx[i][0];
5      int ny = y + dy[i][1];
6  }
```

深度优先搜索初始化 tips:

- 图：存储需要遍历的数据，目前一般为一维数组或二维数组

深度优先搜索初始化 tips:

- 图：存储需要遍历的数据，目前一般为一维数组或二维数组
- 已访问节点：存储已经访问的节点，一般为布尔类型的数组

深度优先搜索初始化 tips:

- 图：存储需要遍历的数据，目前一般为一维数组或二维数组
- 已访问节点：存储已经访问的节点，一般为布尔类型的数组
- 移动方向：存储可能的移动方向，一般为二维数组

深度优先搜索初始化 tips:

- 图：存储需要遍历的数据，目前一般为一维数组或二维数组
- 已访问节点：存储已经访问的节点，一般为布尔类型的数组
- 移动方向：存储可能的移动方向，一般为二维数组

深度优先搜索初始化 tips:

- 图：存储需要遍历的数据，目前一般为一维数组或二维数组
- 已访问节点：存储已经访问的节点，一般为布尔类型的数组
- 移动方向：存储可能的移动方向，一般为二维数组

算法描述:

```
1  void dfs() { // 参数视具体情况而定，一般为可访问的节点
2      if (访问到终点) { 结束 }
3      for (遍历所有可能的状态) {
4          if (该节点未访问) {
5              标记该节点已访问;
6              dfs(进入到下一个可访问节点)
7              恢复该节点状态
8          }
9      }
10 }
```

深度优先搜索 I

给定以下迷宫，0 表示可通过，1 表示不可通过，请设计程序求出可通过的路径数。

```cpp
#include <bits/stdc++.h>
using namespace std;
// 方向数组
int dx[4][2] = {
    {-1,  0}, // 上
    {1,   0}, // 下
    {0,  -1}, // 左
    {0,   1}, // 右
};
// 存储迷宫
int migong[5][5] = {
    0, 1, 0, 0, 0,
    0, 1, 1, 1, 0,
    0, 0, 0, 0, 0,
    0, 1, 1, 1, 0,
    0, 0, 0, 1, 0
};
bool vis[5][5];
int ans = 0;
void dfs(int x, int y) {
    if (x == 4 && y == 4) {
        ans++;
        return;
    }
    for (int i = 0; i < 4; i++) {
```

深度优先搜索 II

```
26          int nx = x + dx[i][0];
27          int ny = y + dx[i][1];
28          if (nx < 0 || nx >= 5 || ny < 0 || ny >= 5) continue; // 越界
29          if (migong[nx][ny] || vis[nx][ny]) continue; // 迷宫不可走或走过
30          vis[nx][ny] = 1;
31          dfs(nx, ny);
32          vis[nx][ny] = 0;
33      }
34  }
35  int main() {
36      vis[0][0] = 1;
37      dfs(0, 0);
38      cout << ans << endl;
39      return 0;
40  }
```

目录

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即
  1. 从图中的某一顶点 V_0 开始，先访问 V_0；

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即
  1. 从图中的某一顶点 V_0 开始，先访问 V_0；
  2. 访问所有与 V_0 相邻接的顶点 $V_1, V_2, \ldots\ldots, V_t$；

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即
  1. 从图中的某一顶点 V_0 开始，先访问 V_0；
  2. 访问所有与 V_0 相邻接的顶点 $V_1, V_2, \ldots\ldots, V_t$；
  3. 依次访问与 $V_1, V_2, \ldots\ldots, V_t$ 相邻接的所有未曾访问过的顶点；

广度优先搜索算法

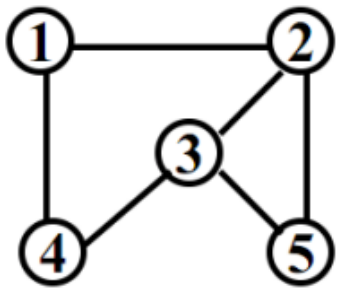
- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即
  1. 从图中的某一顶点 V_0 开始，先访问 V_0；
  2. 访问所有与 V_0 相邻接的顶点 $V_1, V_2, \ldots\ldots, V_t$；
  3. 依次访问与 $V_1, V_2, \ldots\ldots, V_t$ 相邻接的所有未曾访问过的顶点；
  4. 循此以往，直至所有的顶点都被访问过为止。

广度优先搜索算法

- **广度优先搜索算法**（又称宽度优先搜索）是最简便的图的搜索算法之一。
- 广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目节点为止。即
  1. 从图中的某一顶点 V_0 开始，先访问 V_0；
  2. 访问所有与 V_0 相邻接的顶点 $V_1, V_2, \ldots\ldots, V_t$；
  3. 依次访问与 $V_1, V_2, \ldots\ldots, V_t$ 相邻接的所有未曾访问过的顶点；
  4. 循此以往，直至所有的顶点都被访问过为止。
- 这种搜索的次序体现沿层次向横向扩长的趋势，所以称之为广度优先搜索。

- 路径记录：每生成一个子结点，就要提供指向它们父亲结点的指针。当解出现时候，通过逆向（递归）跟踪，找到从根结点到目标结点的一条路径。

例：



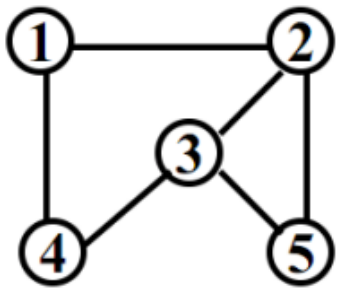
广度优先搜索访问序列：

1 → 2 → 4 → 3 → 5

2 → 1 → 3 → 5 → 4 ……

- 路径记录：每生成一个子结点，就要提供指向它们父亲结点的指针。当解出现时候，通过逆向（递归）跟踪，找到从根结点到目标结点的一条路径。
- 判重：生成的结点要与前面所有已经产生结点比较，以免出现重复结点，浪费时间，还有可能陷入死循环。

例：



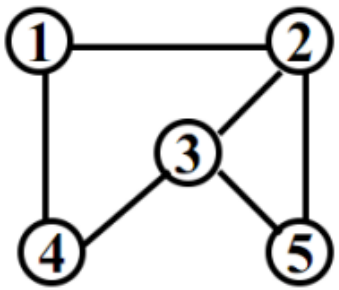
广度优先搜索访问序列：

1 → 2 → 4 → 3 → 5

2 → 1 → 3 → 5 → 4 ……

- 路径记录：每生成一个子结点，就要提供指向它们父亲结点的指针。当解出现时候，通过逆向（递归）跟踪，找到从根结点到目标结点的一条路径。
- 判重：生成的结点要与前面所有已经产生结点比较，以免出现重复结点，浪费时间，还有可能陷入死循环。
- 时间效率：广度优先搜索的效率有赖于目标结点所在位置情况，如果目标结点深度处于较深层时，需搜索的结点数基本上以指数增长。

例：



广度优先搜索访问序列：

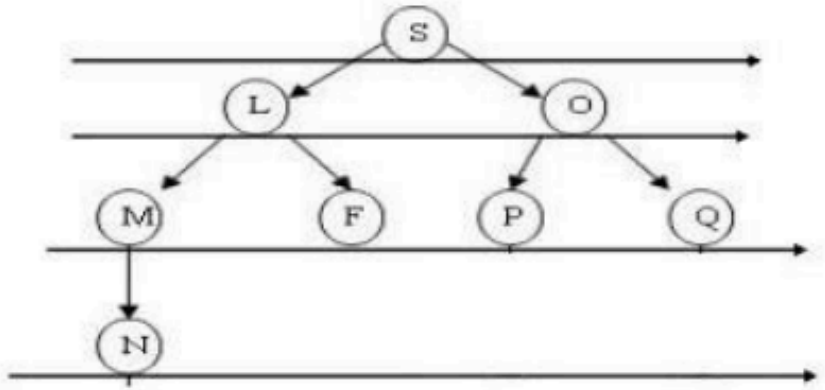
1 → 2 → 4 → 3 → 5

2 → 1 → 3 → 5 → 4 ……

bfs 算法描述

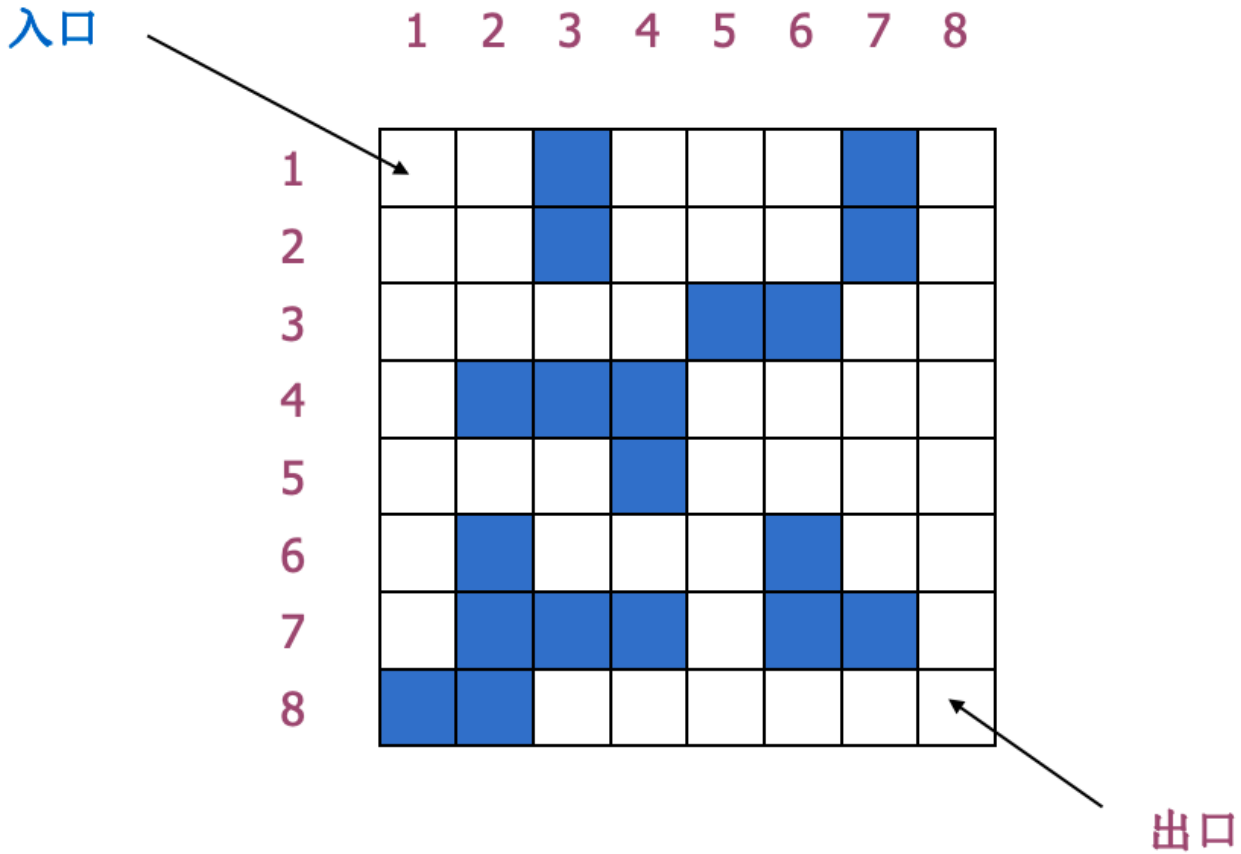
```
1  void bfs() {
2      初始化，初始状态存入队列;
3      队列首指针head=1; 尾指针tail=1;
4      While (head < tail) { // 队列不为空
5          for i = 1 to max { // max为产生子结点的规则数
6              if (子结点符合条件 && 与队列中结点不重复) {
7                  把新结点存入列尾,tail指针增1
8                  if (新结点是目标结点) then 输出并退出
9              }
10         }
11         首指针head后移一位，指向待扩展结点;
12     }
13 }
```

原理解析



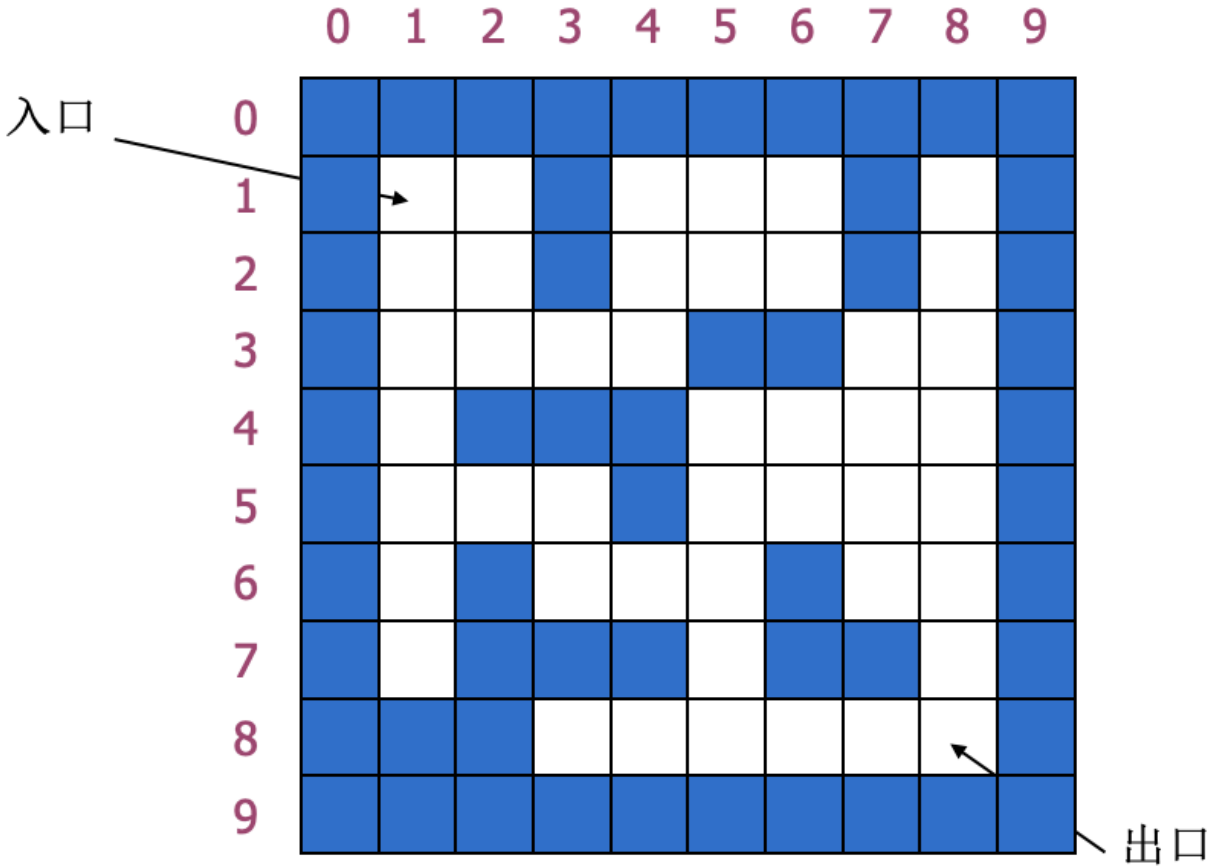
下标	1	2	3	4	5	6	7	8
father	0	1	1	2	2	3	3	4
state	S	L	O	M	F	P	Q	N

- 寻找一条从入口到出口的通路

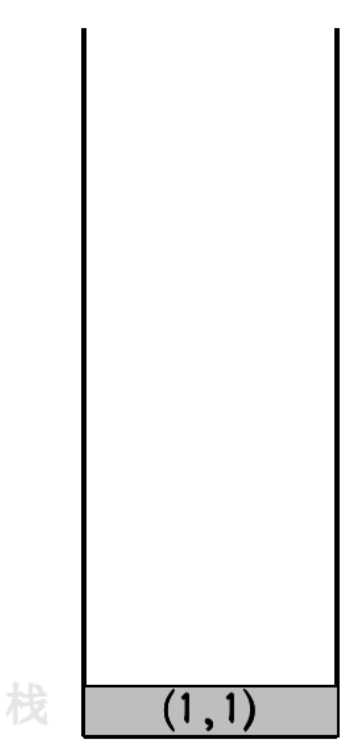


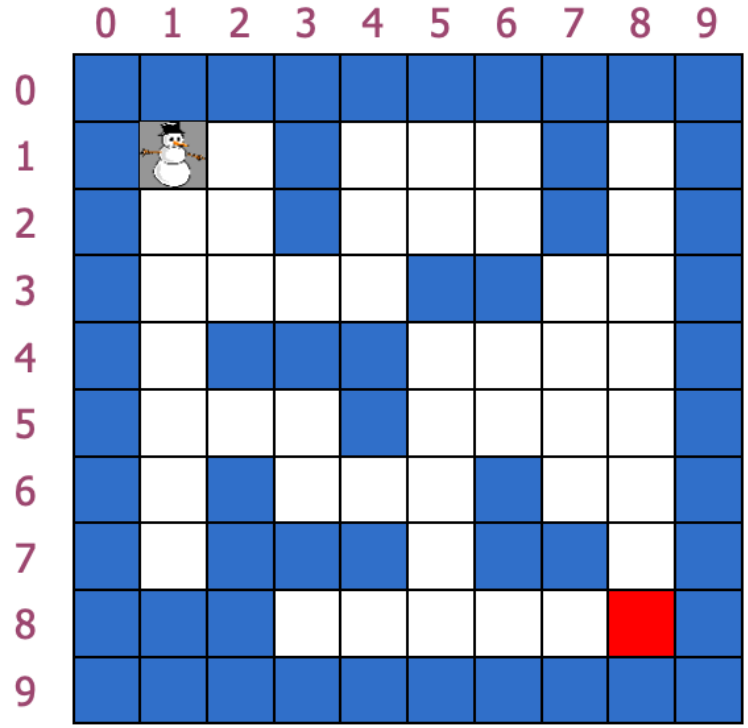
迷宫问题

- 在迷宫周围加墙，避免判断是否出界

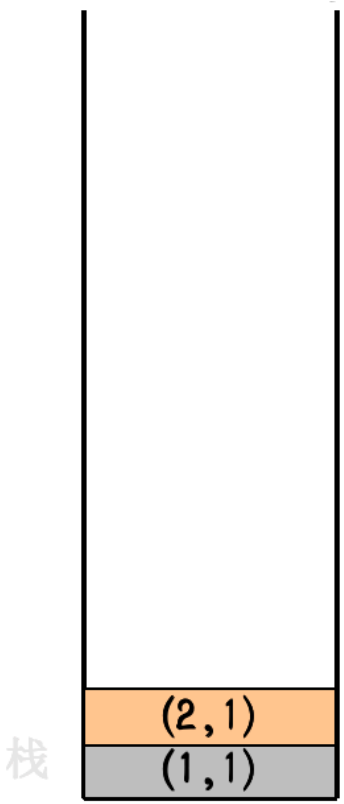


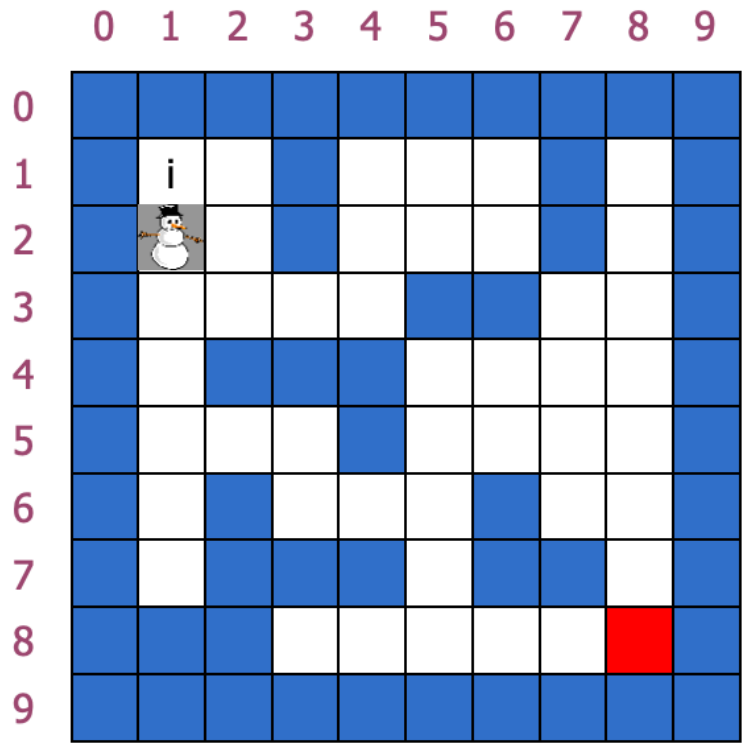
- 在寻找出口的过程中，每前进一步，当前位置入栈；每回退一步，栈顶元素出栈



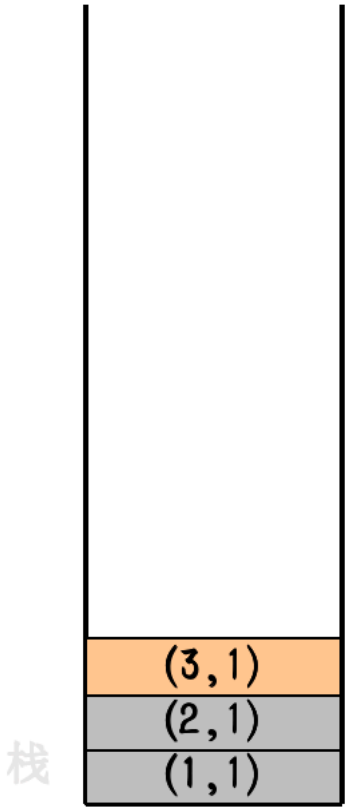


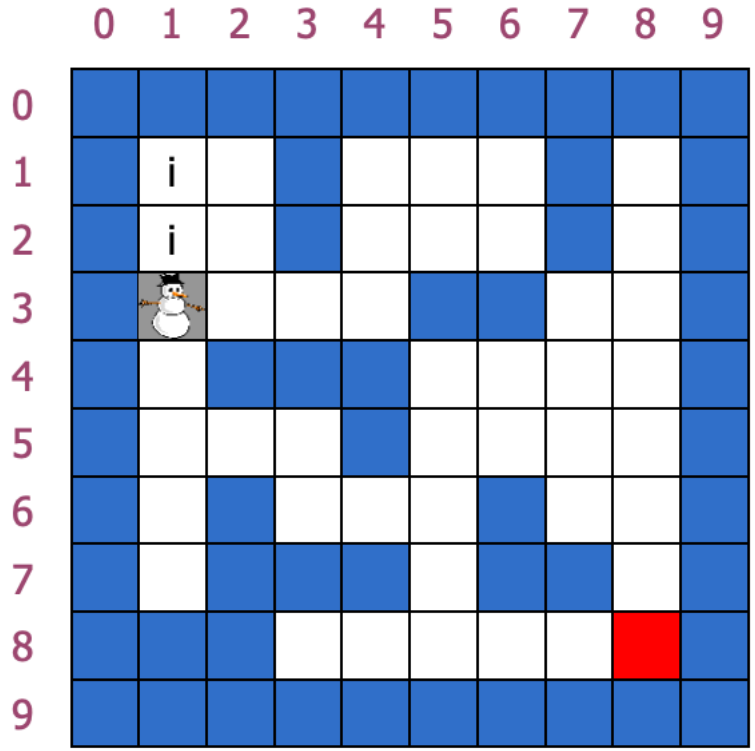
- 向下方前进一步



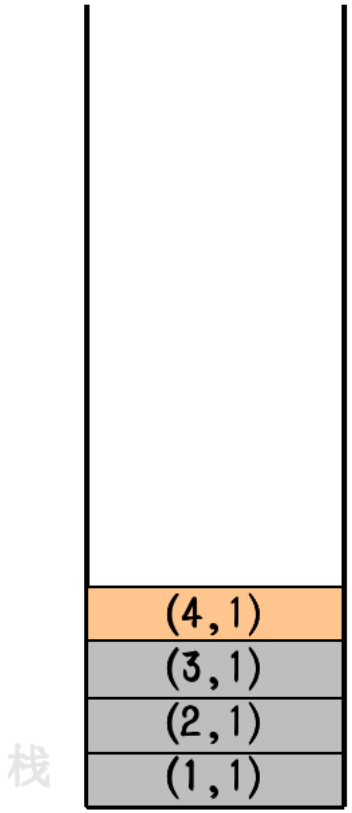


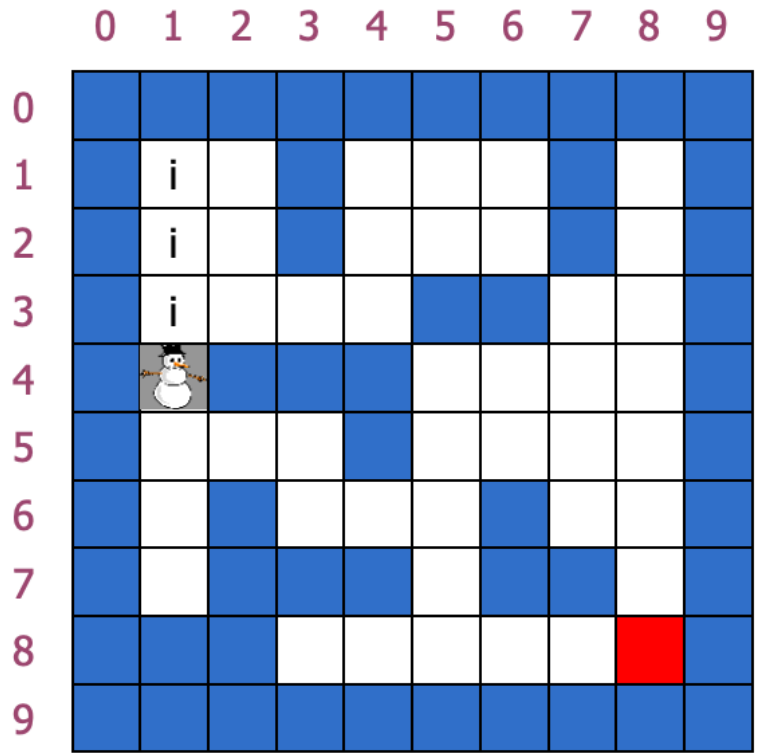
- 向下方前进一步



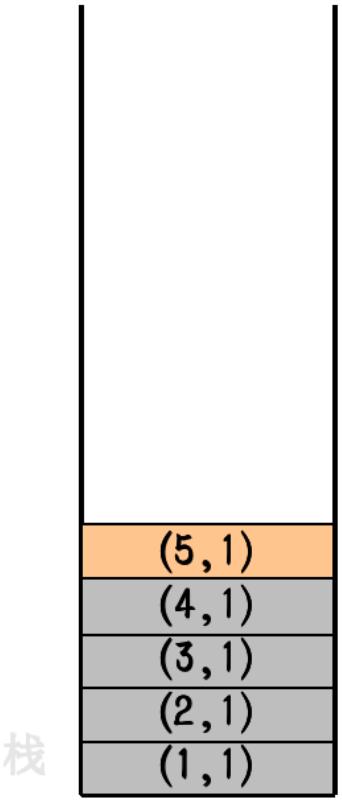


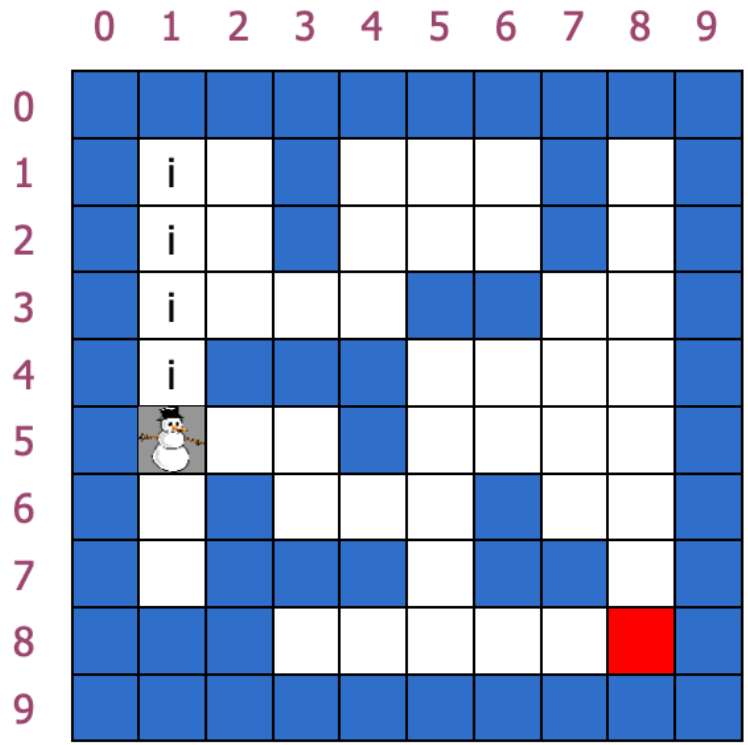
- 向下方前进一步



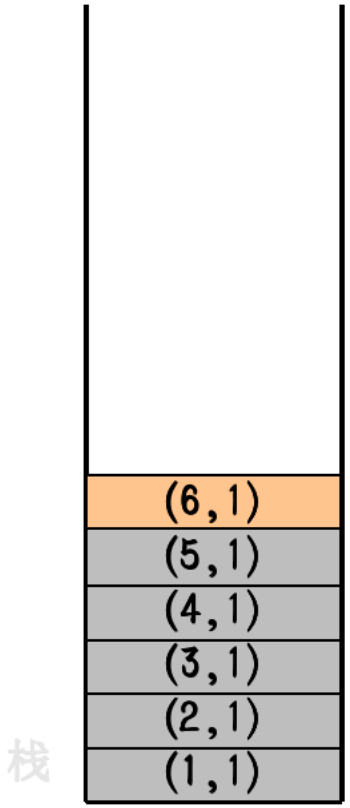


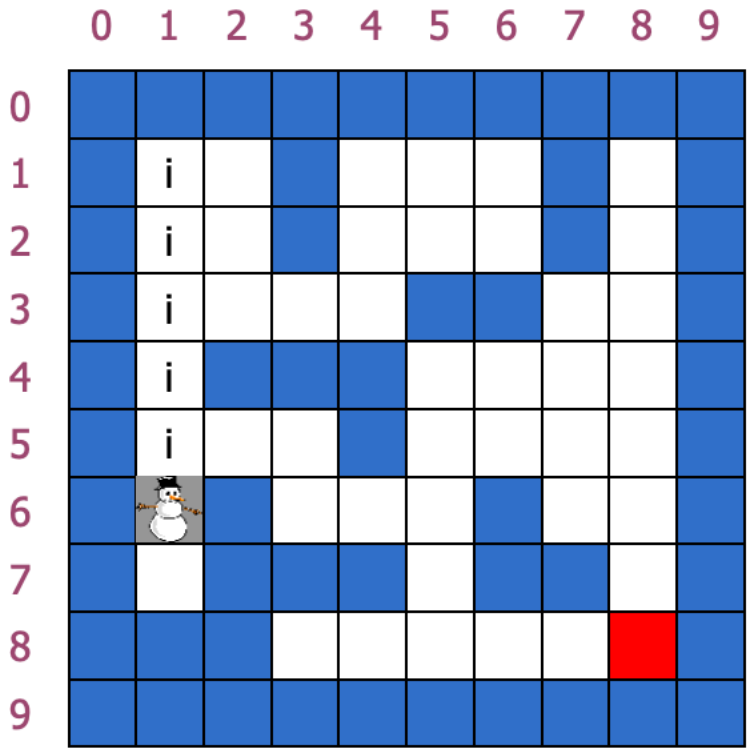
- 向下方前进一步



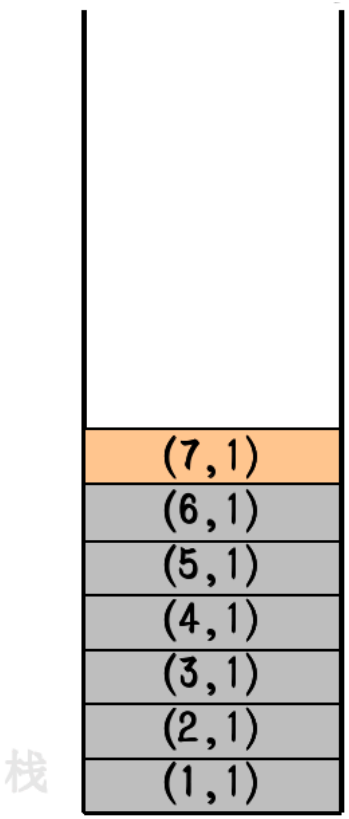


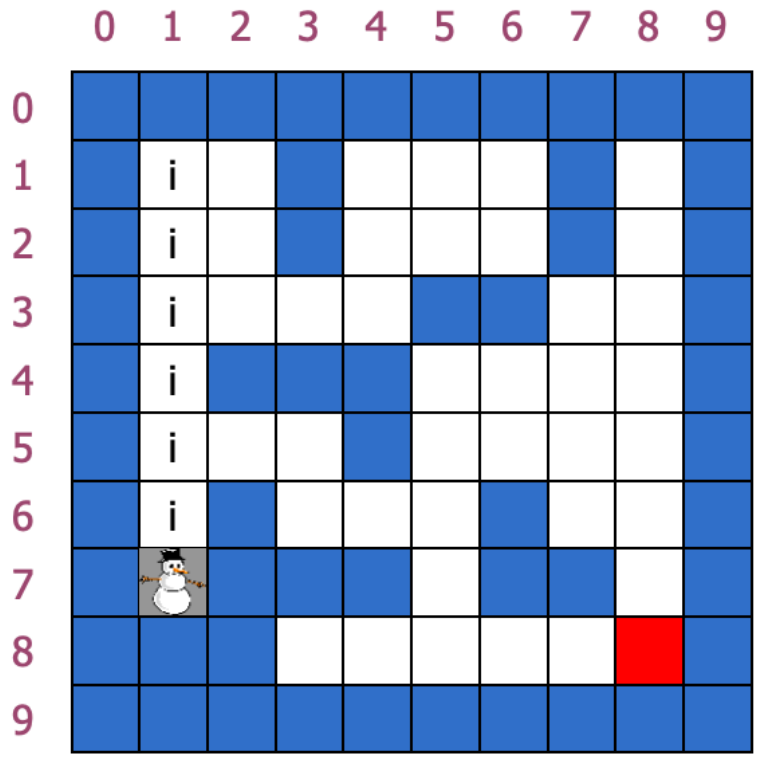
- 向下方前进一步



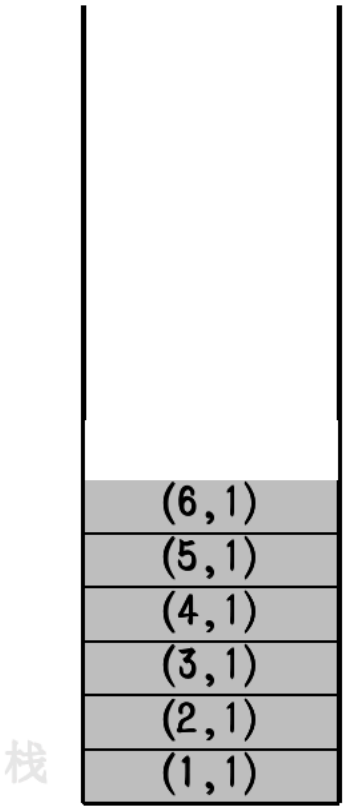


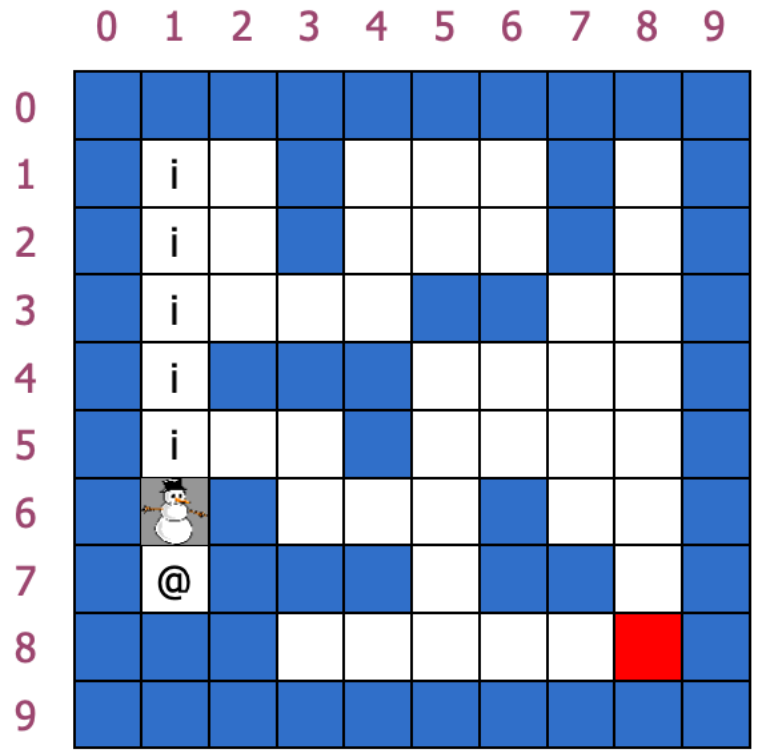
- 向下方前进一步



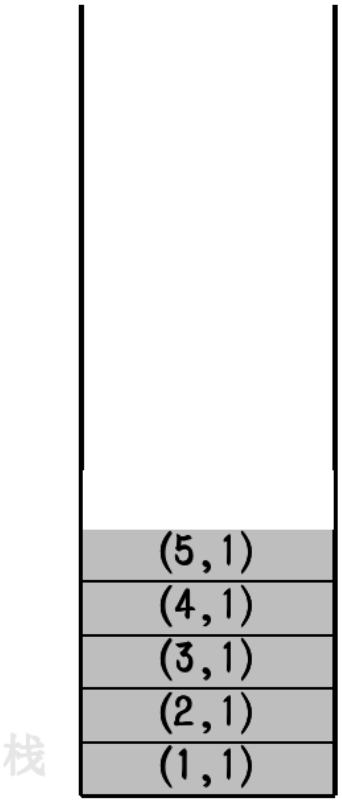


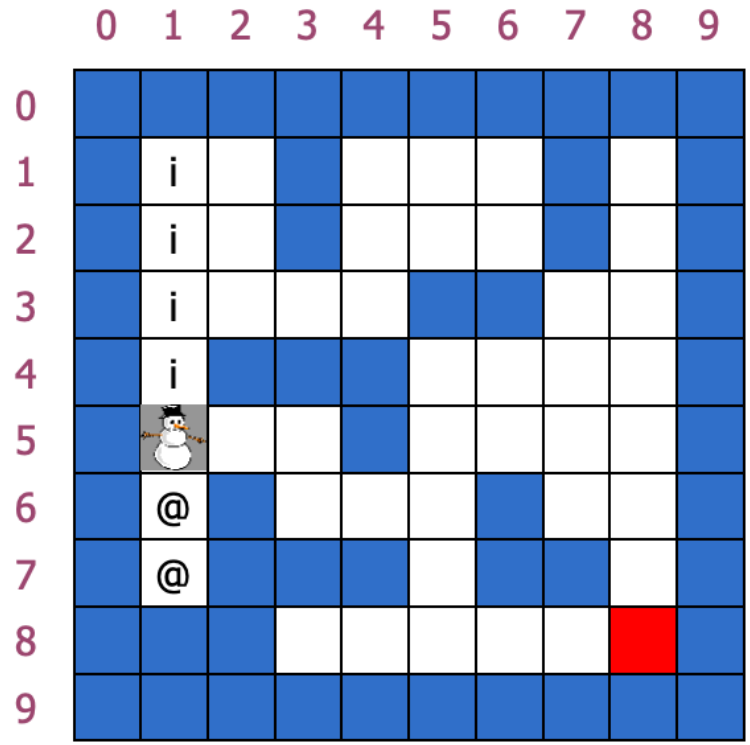
- 向下方、右方、左方均不能前进，上方是来路，则后退



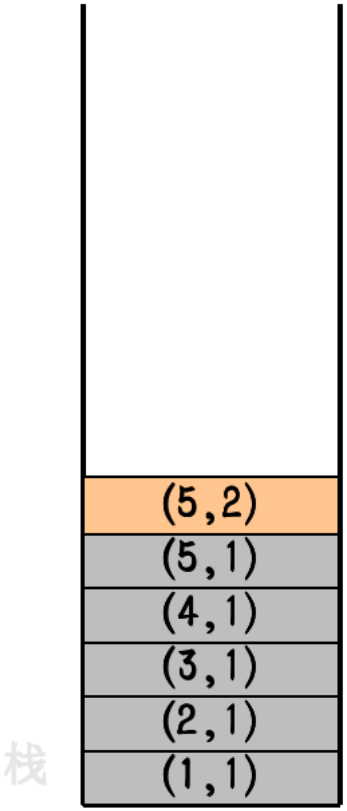


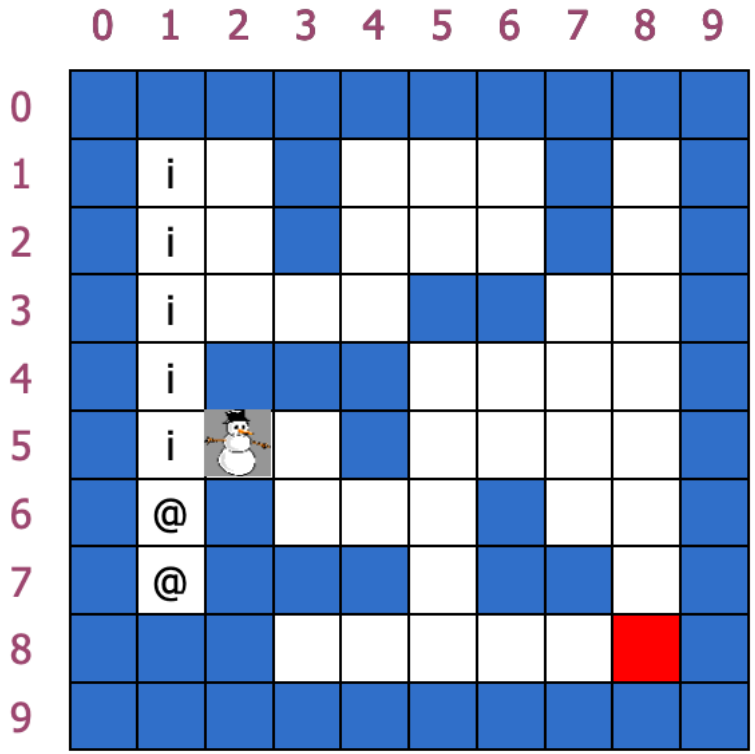
- 向右方、左方均不能前进，下方路不通，上方是来路，则后退



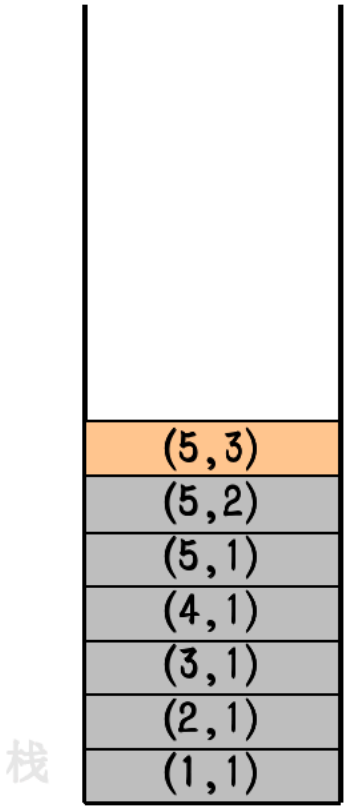


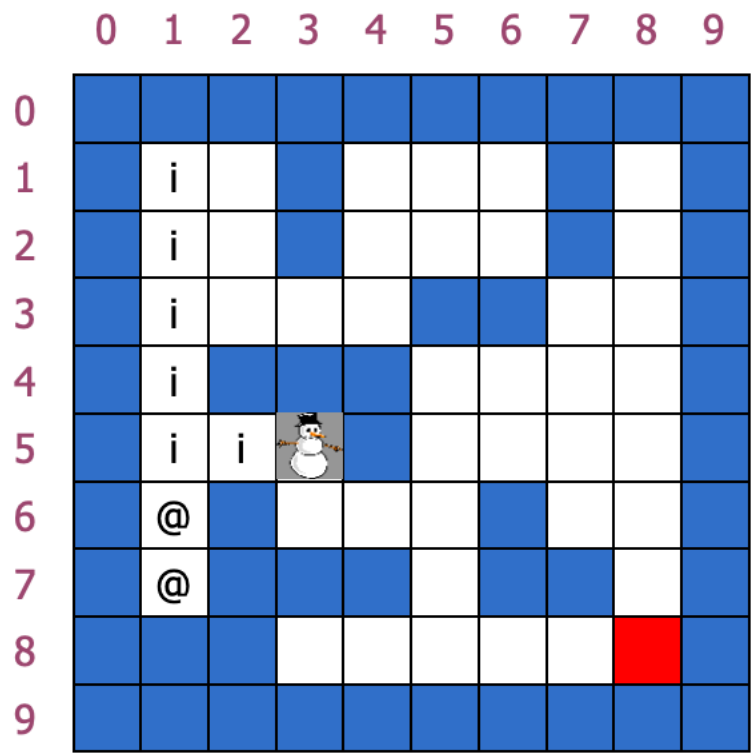
- 向右方前进一步



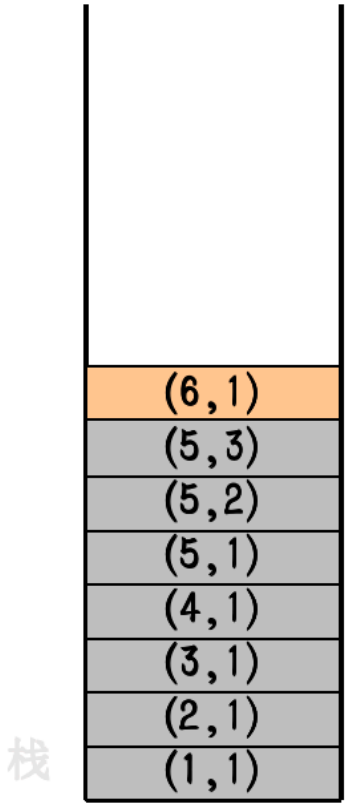


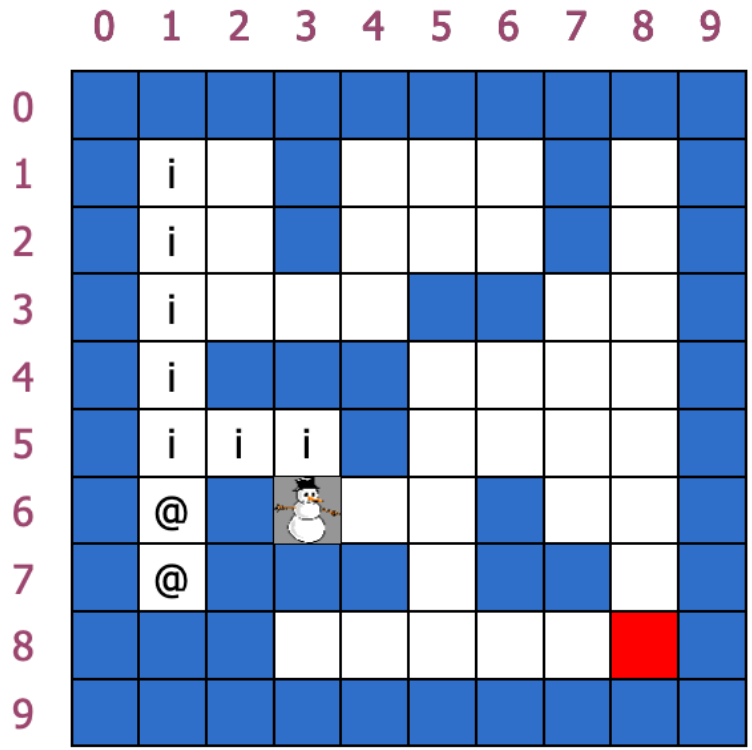
- 下方路不通，向右方前进一步



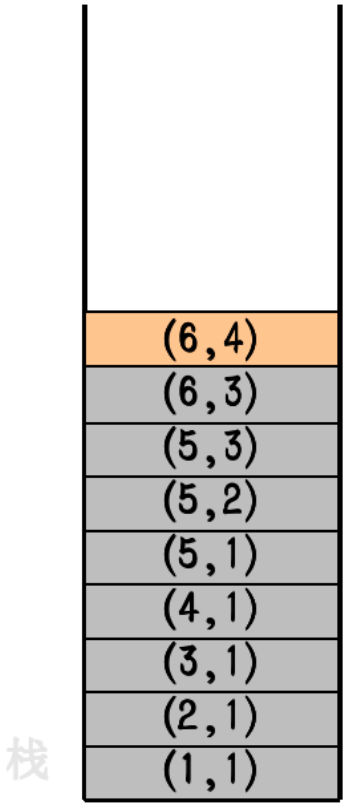


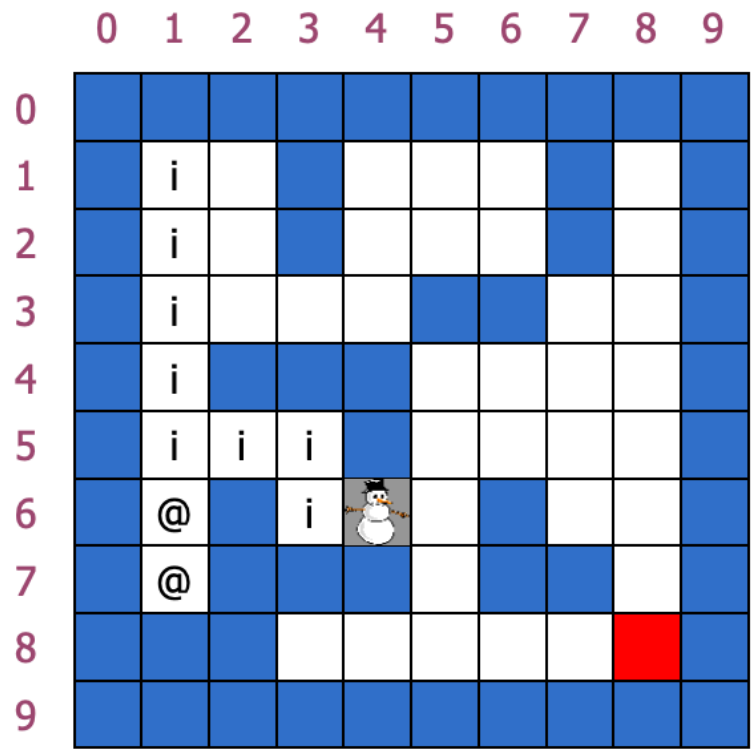
- 向下方前进一步



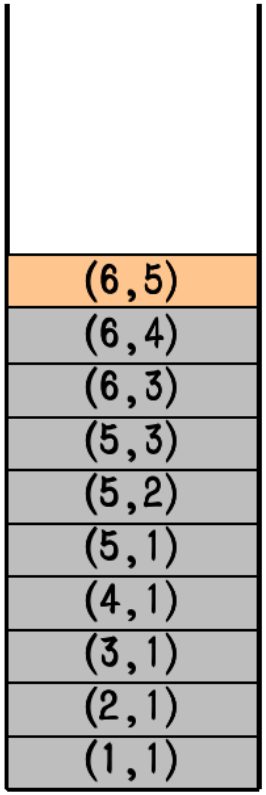


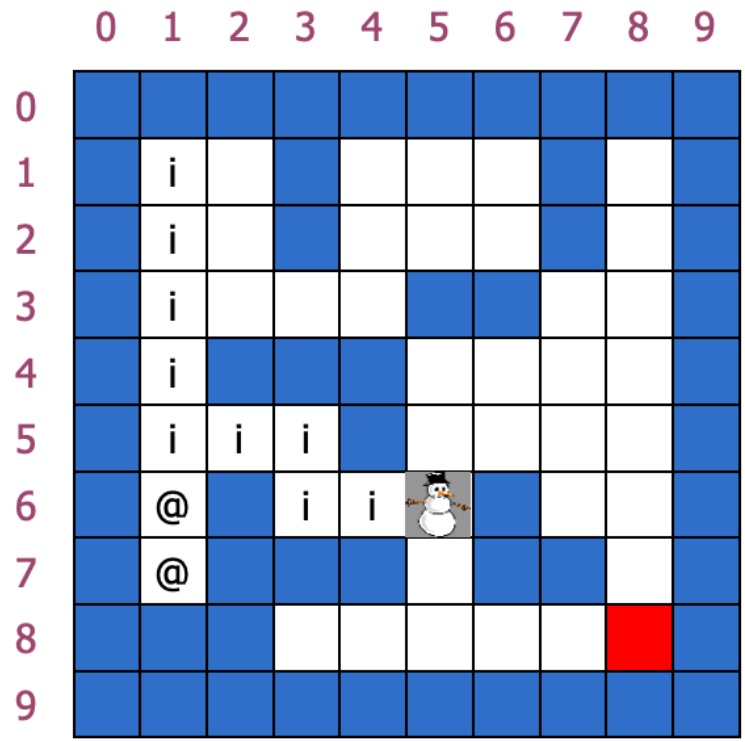
- 下方路不通，向右方前进一步



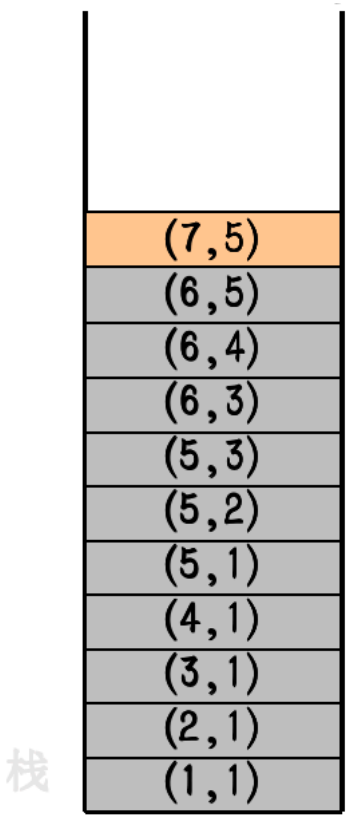


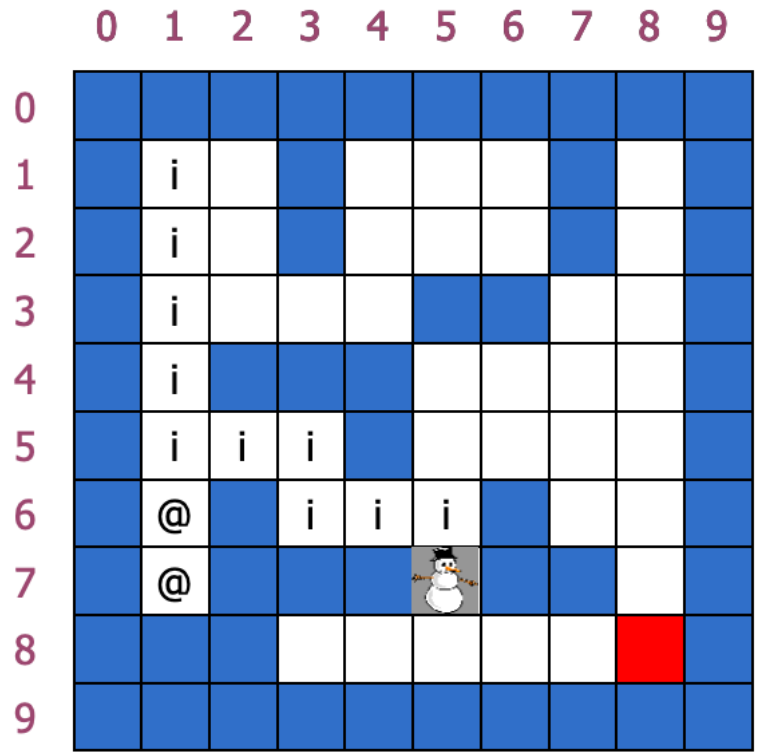
- 下方路不通，向右方前进一步



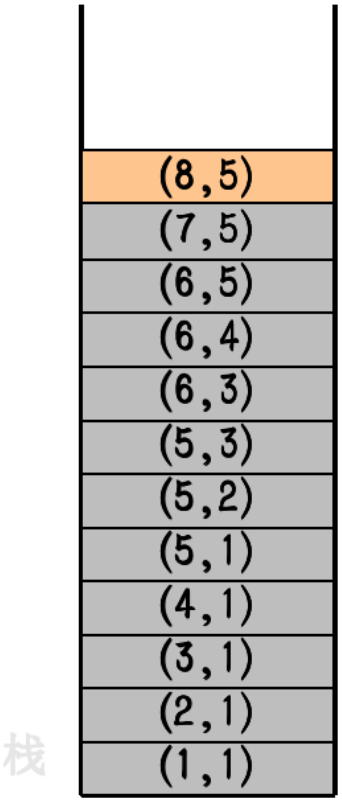


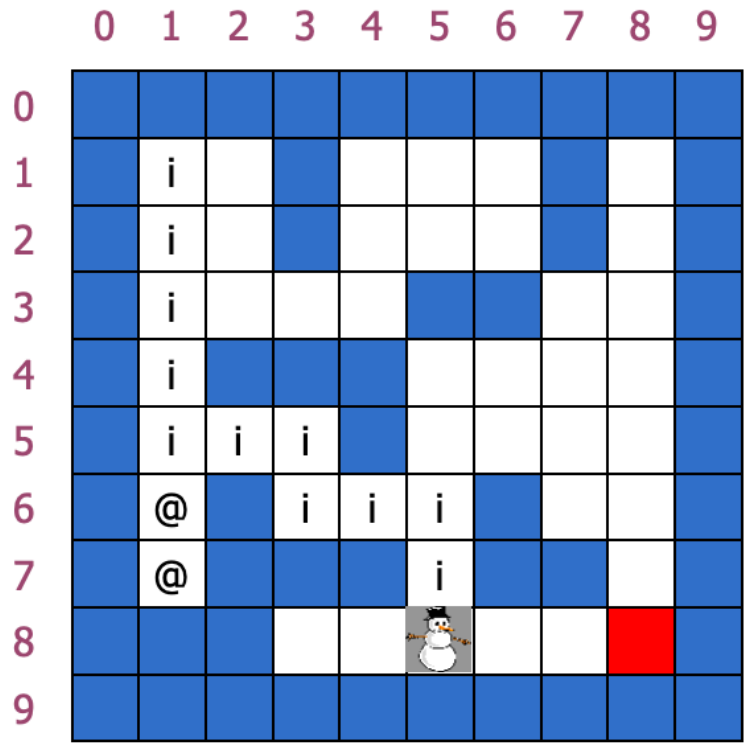
- 向下方前进一步


(7,5)
(6,5)
(6,4)
(6,3)
(5,3)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)
栈

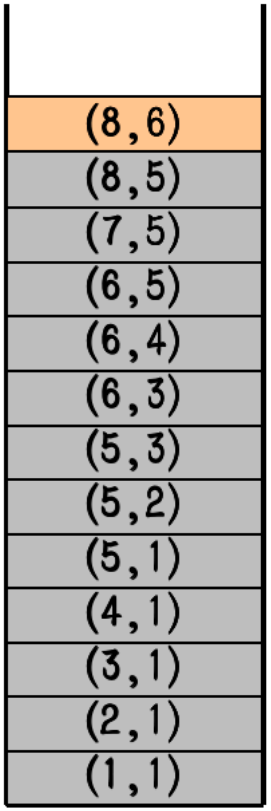

0 1 2 3 4 5 6 7 8 9
0
1 i
2 i
3 i
4 i
5 i i i
6 @ i i i
7 @
8
9

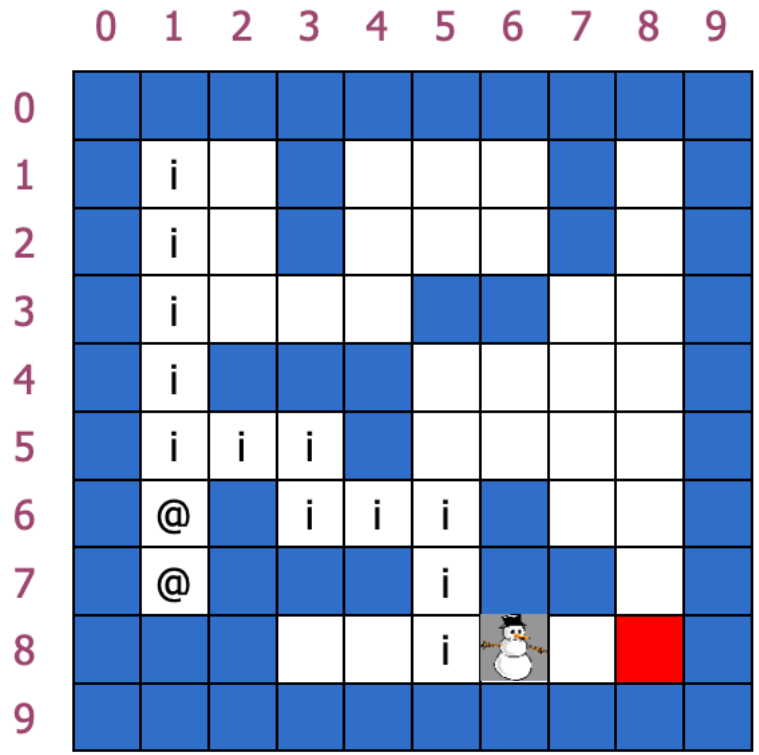
- 向下方前进一步


(8,5)
(7,5)
(6,5)
(6,4)
(6,3)
(5,3)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)
栈

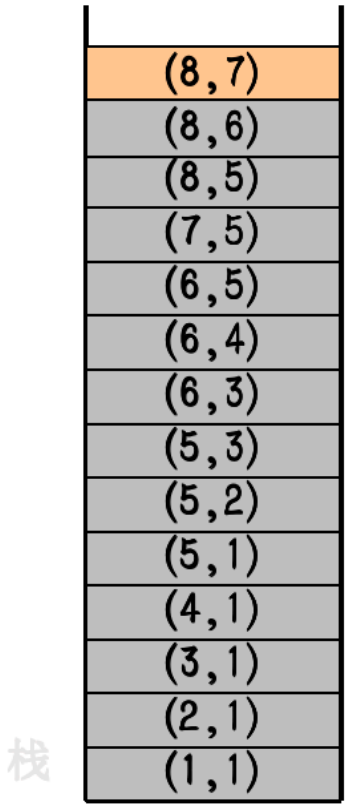

0 1 2 3 4 5 6 7 8 9
0
1 i
2 i
3 i
4 i
5 i i i
6 @ i i i
7 @ i
8
9

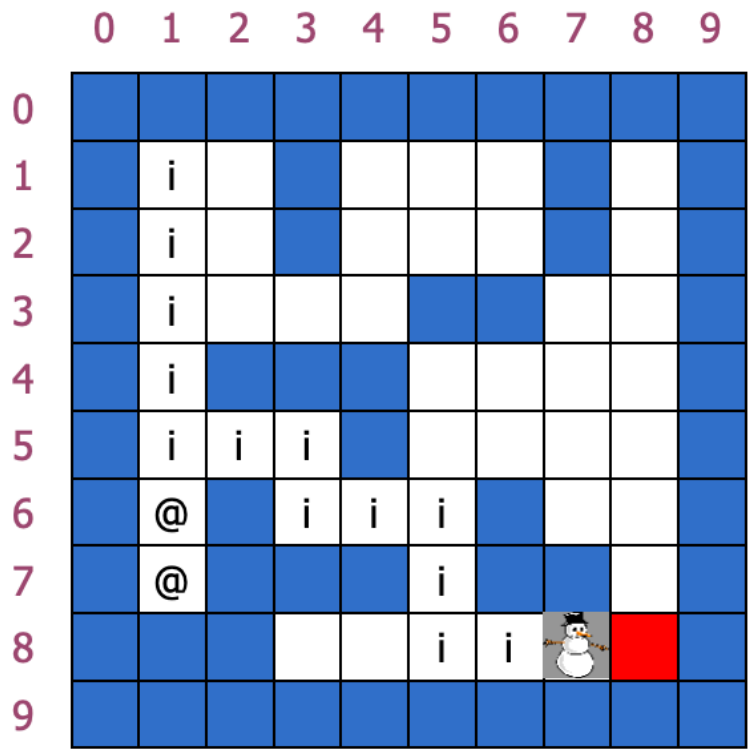
- 向右方前进一步


(8,6)
(8,5)
(7,5)
(6,5)
(6,4)
(6,3)
(5,3)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)
栈

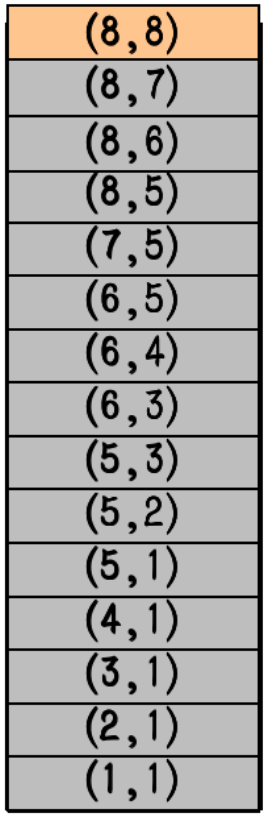

0 1 2 3 4 5 6 7 8 9
0
1 i
2 i
3 i
4 i
5 i i i
6 @ i i i
7 @ i
8 i
9

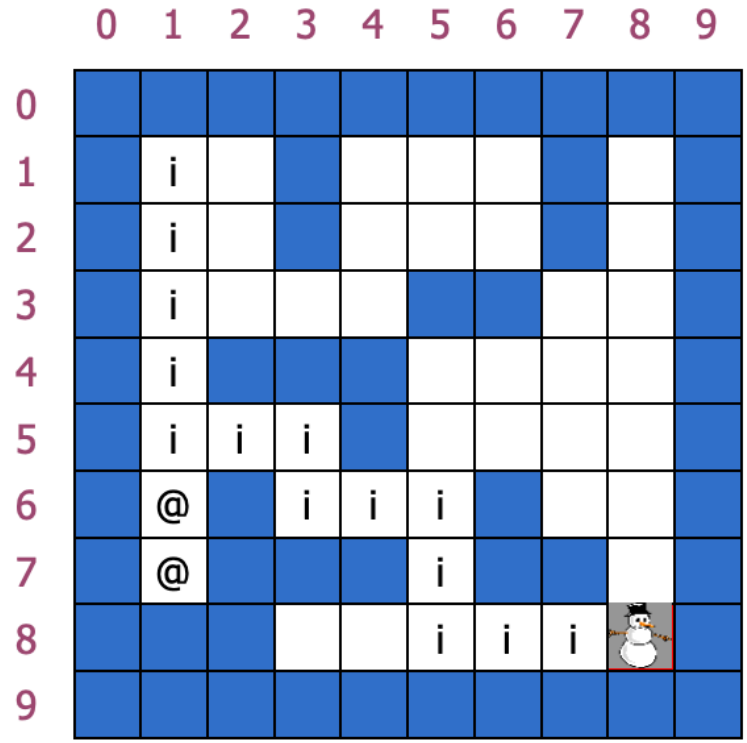
- 下方路不通，向右方前进一步





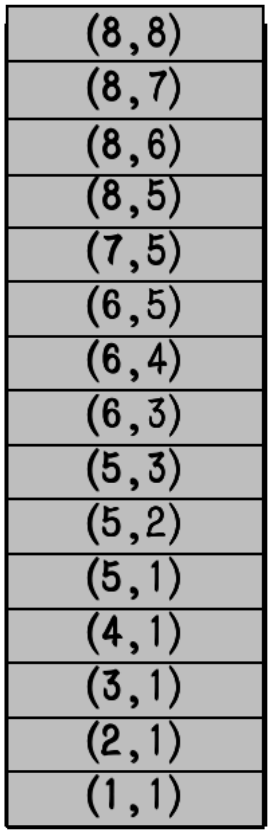
- 下方路不通，向右方前进一步，到达出口

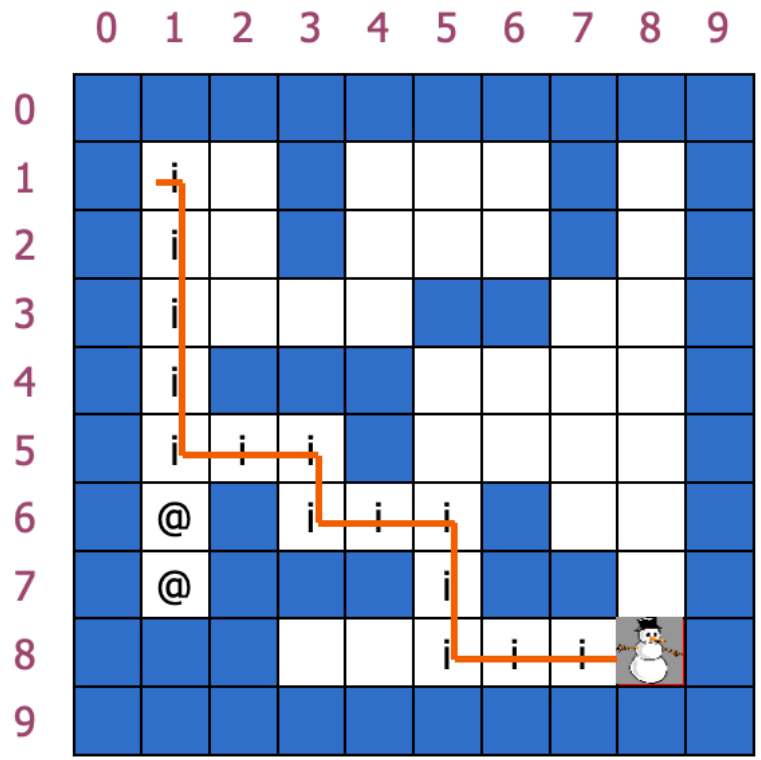




迷宫问题

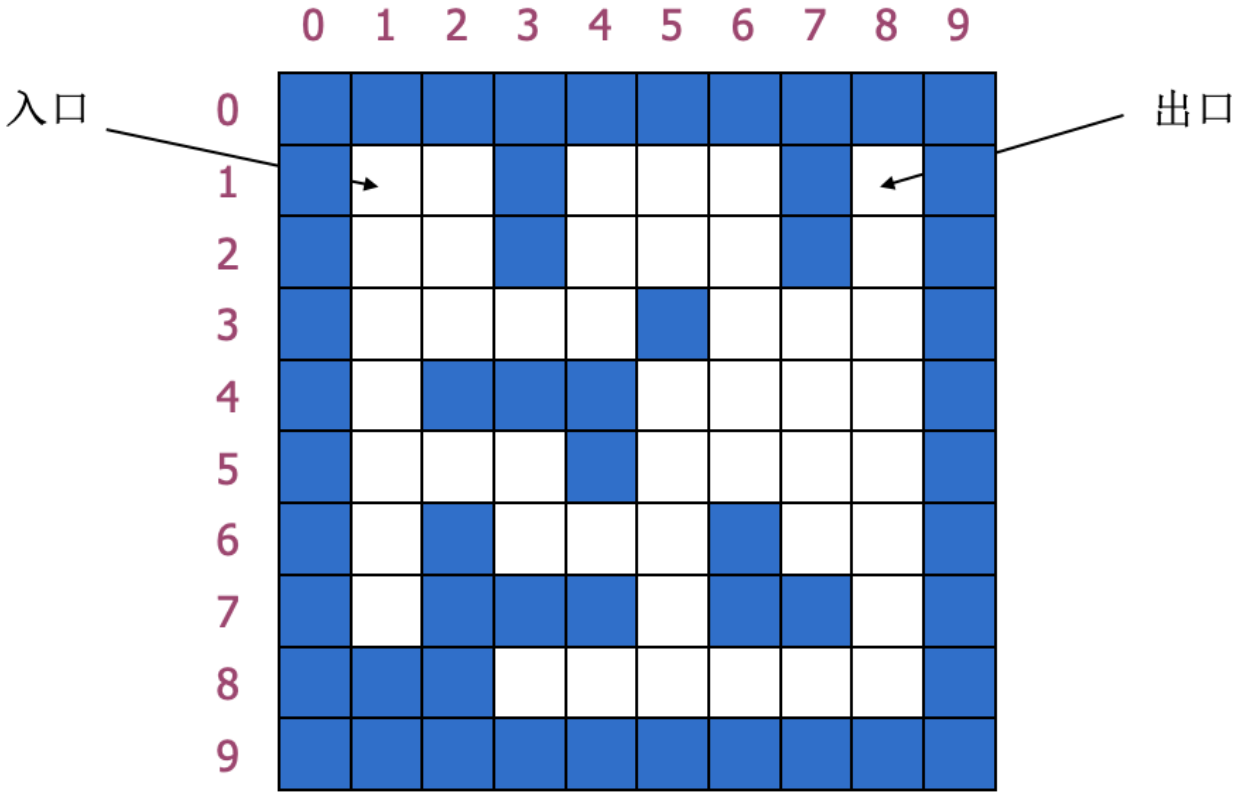
- 用栈保存了路径





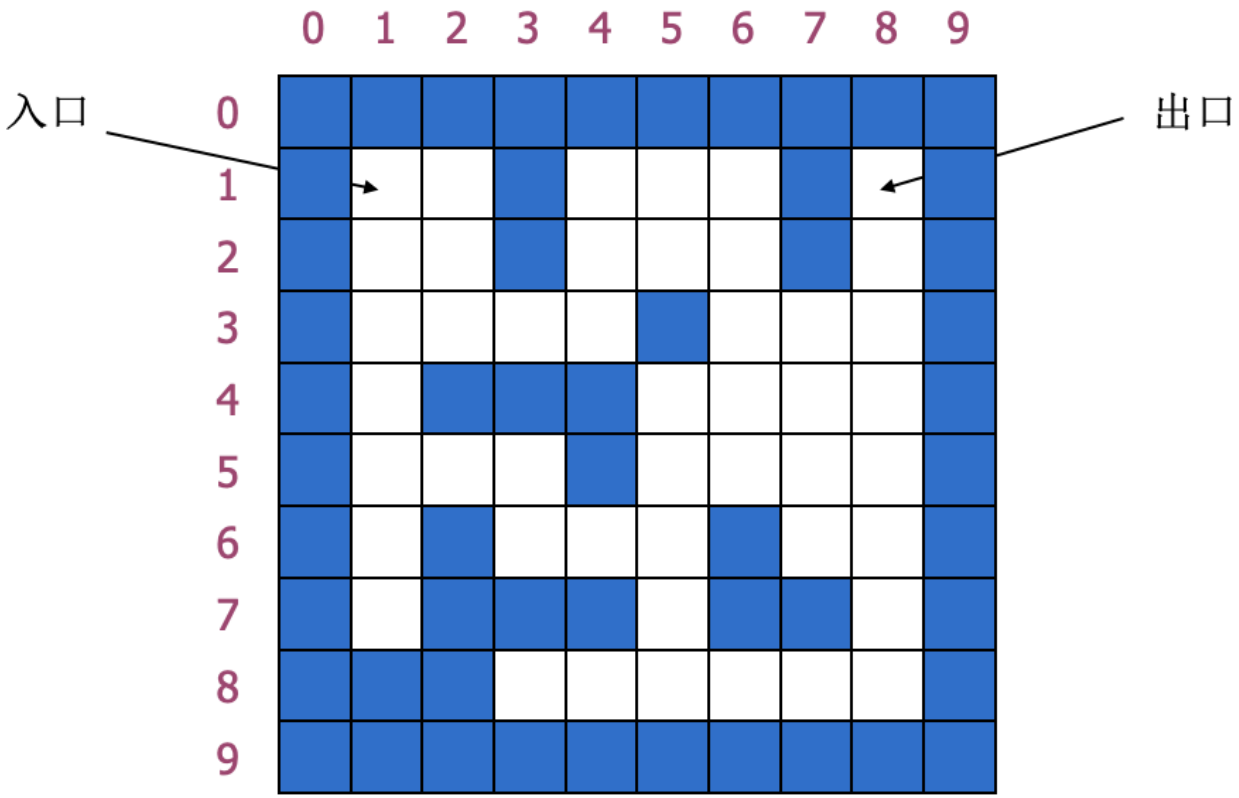
迷宫问题

- 求迷宫入口到出口的最短路径

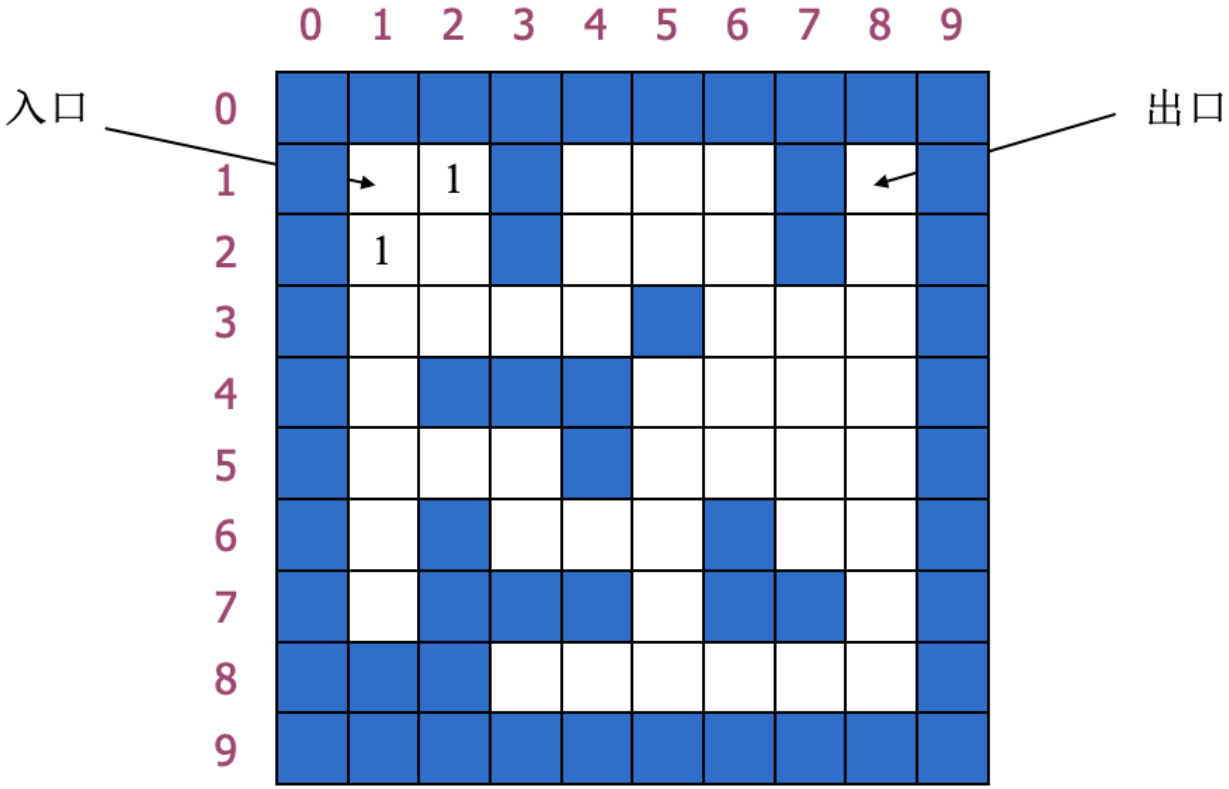


迷宫问题

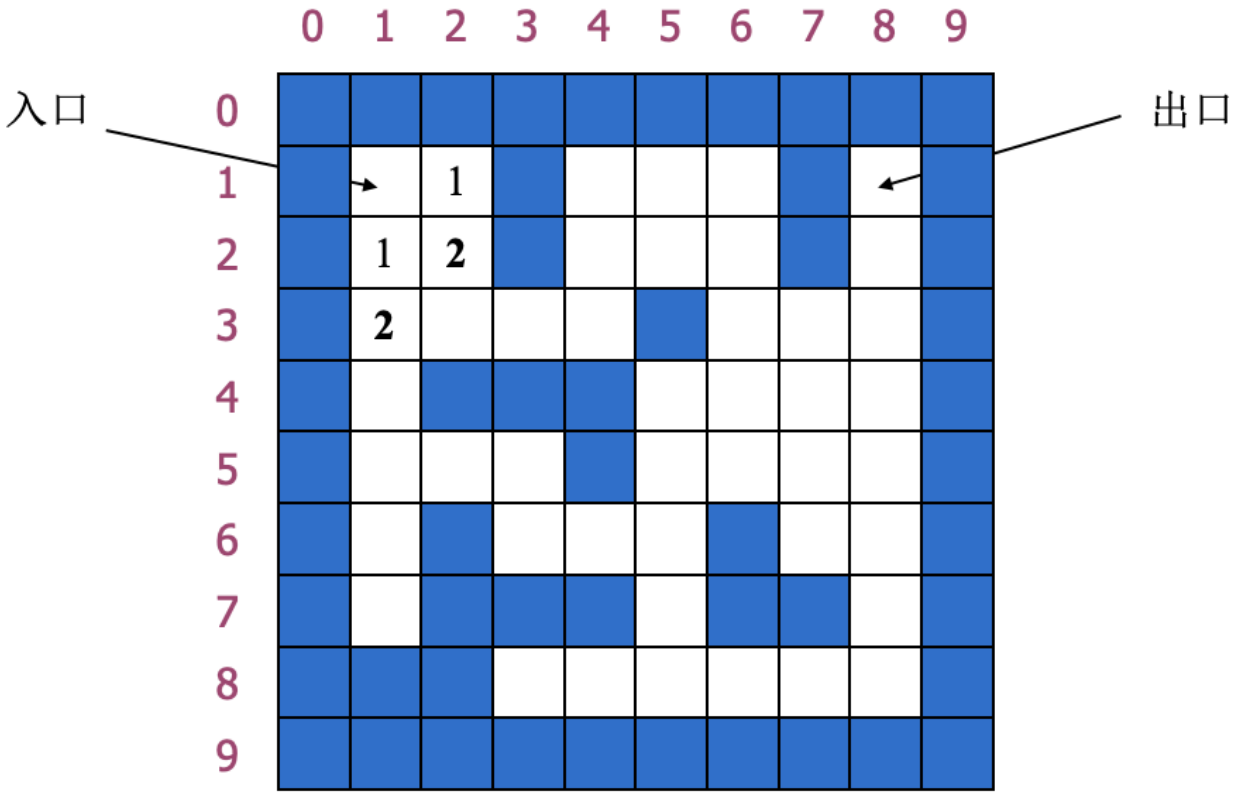
- 0



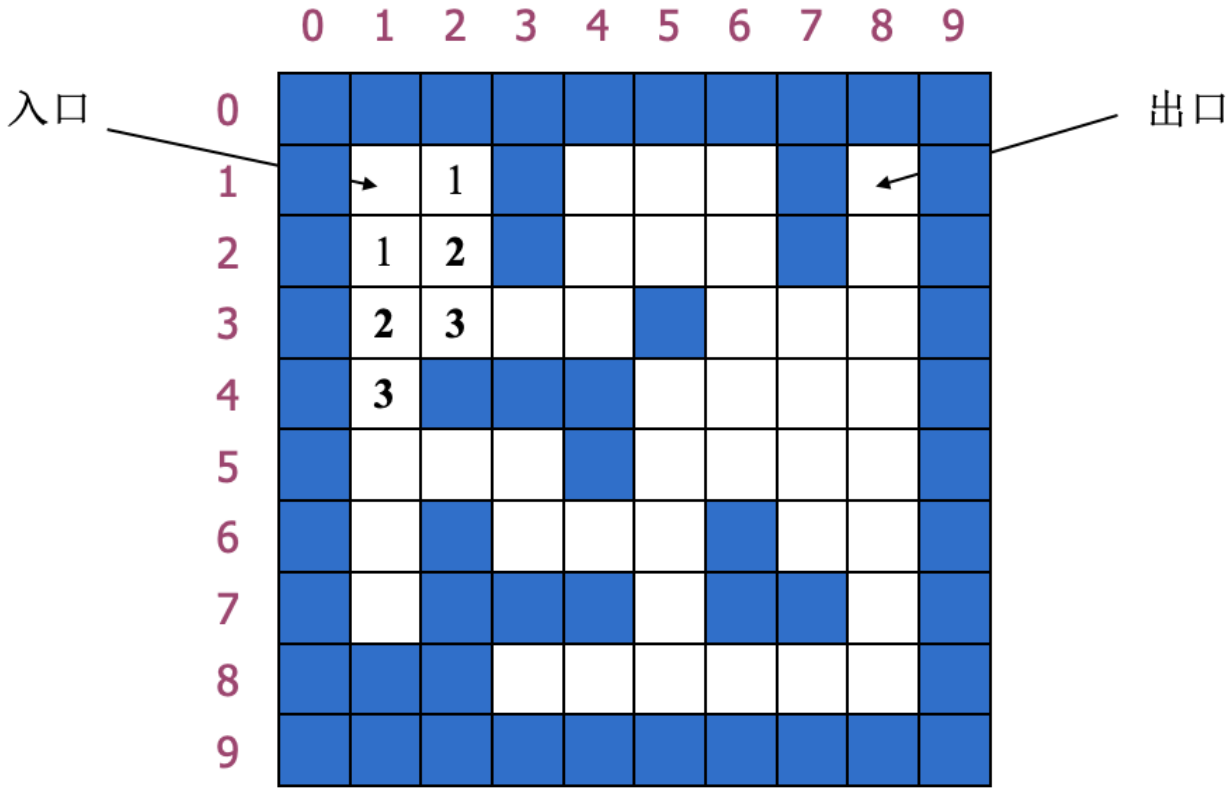
- 1 步可以到的点



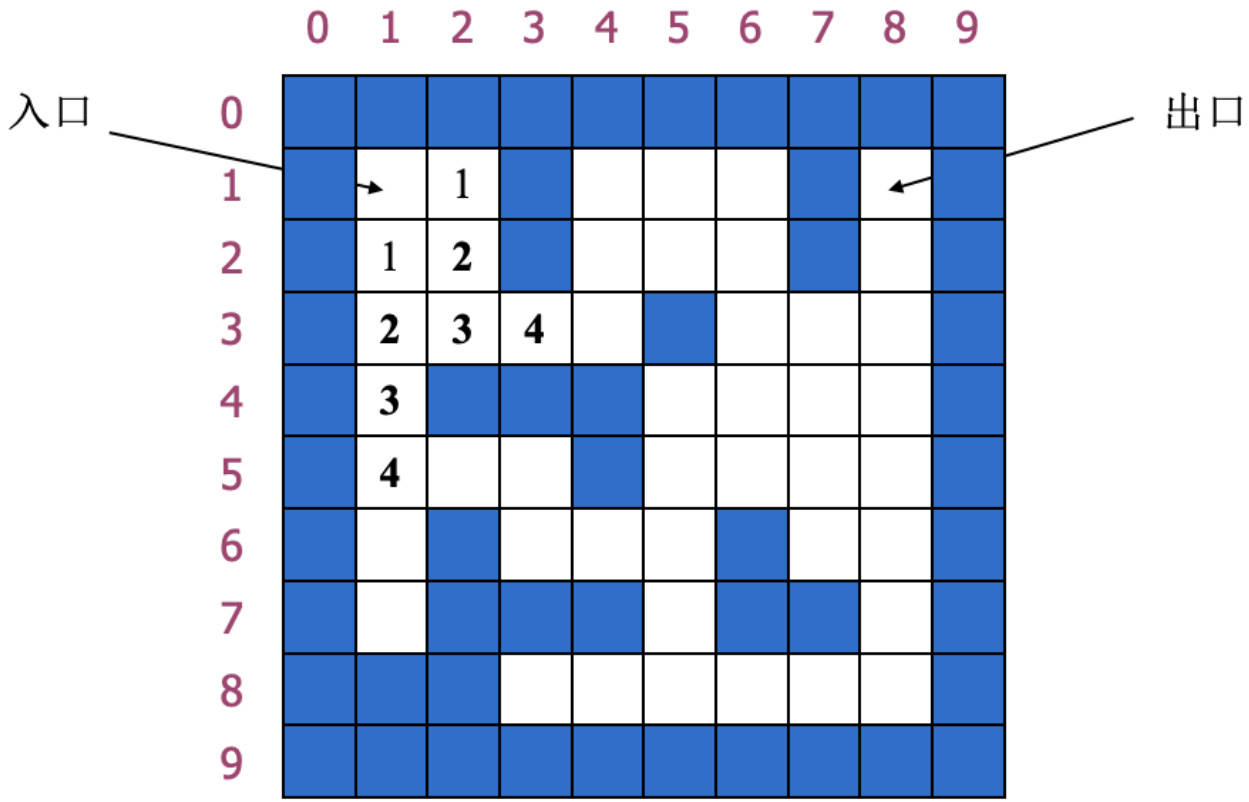
- 2 步可以到的点



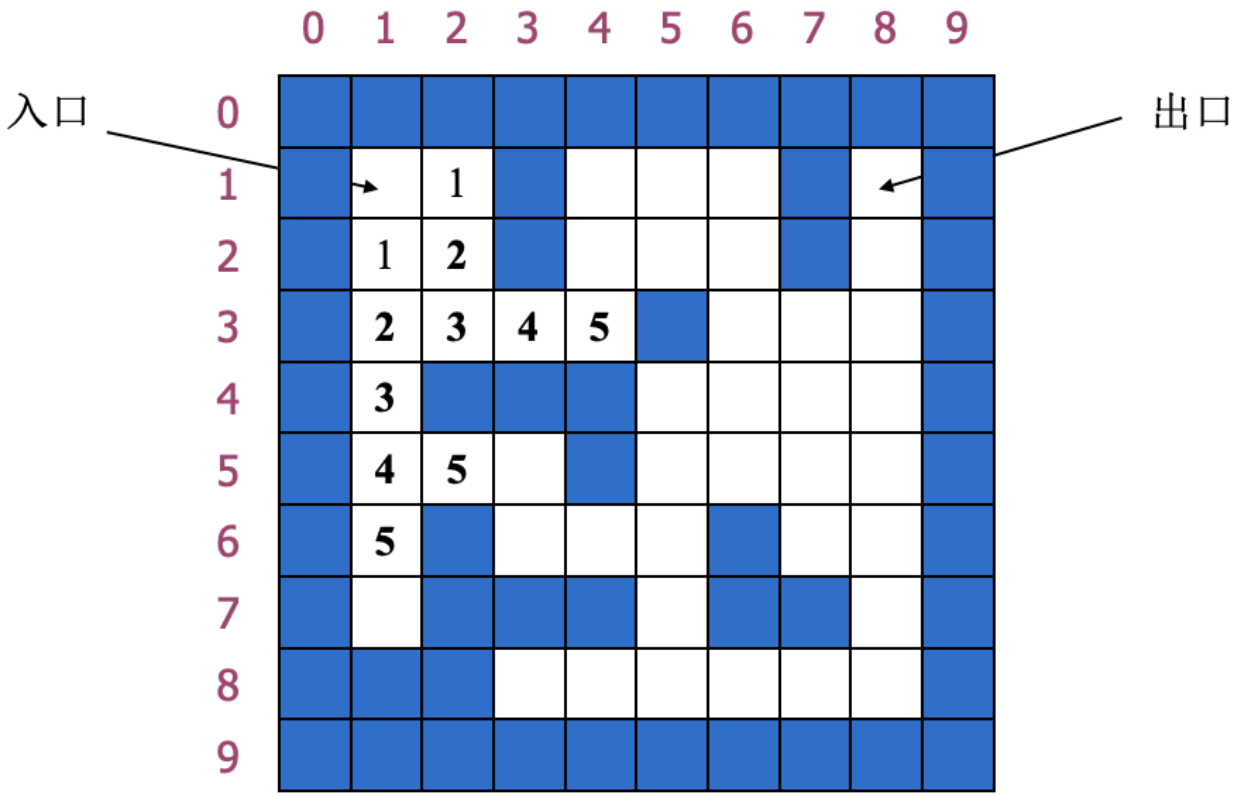
- 3 步可以到的点



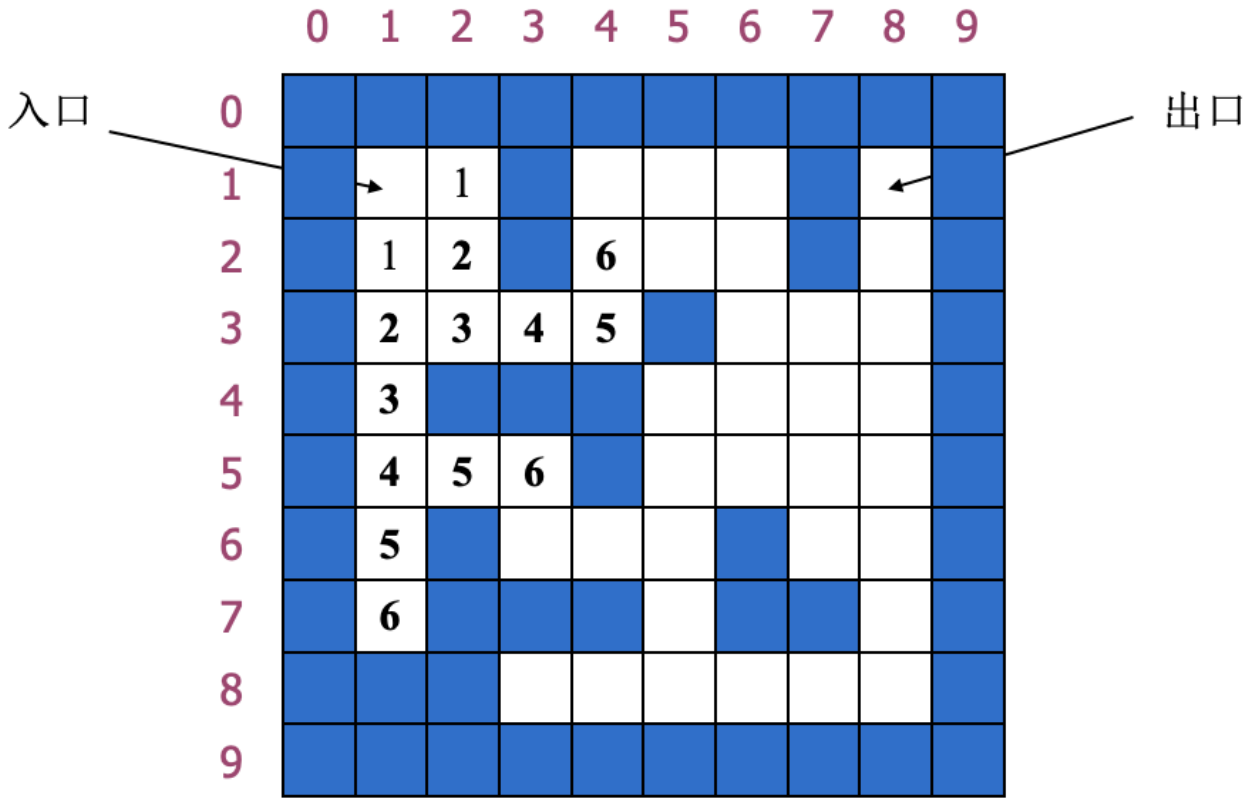
- 4 步可以到的点



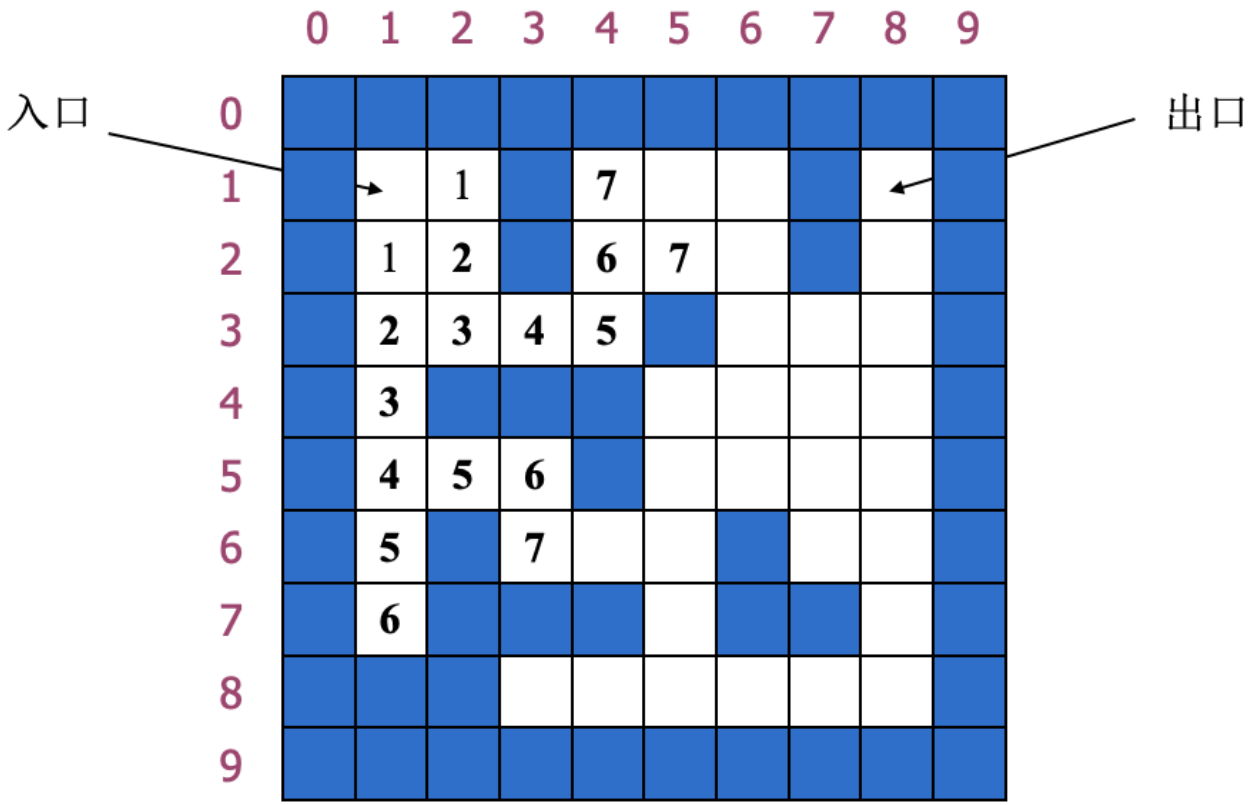
- 5 步可以到的点



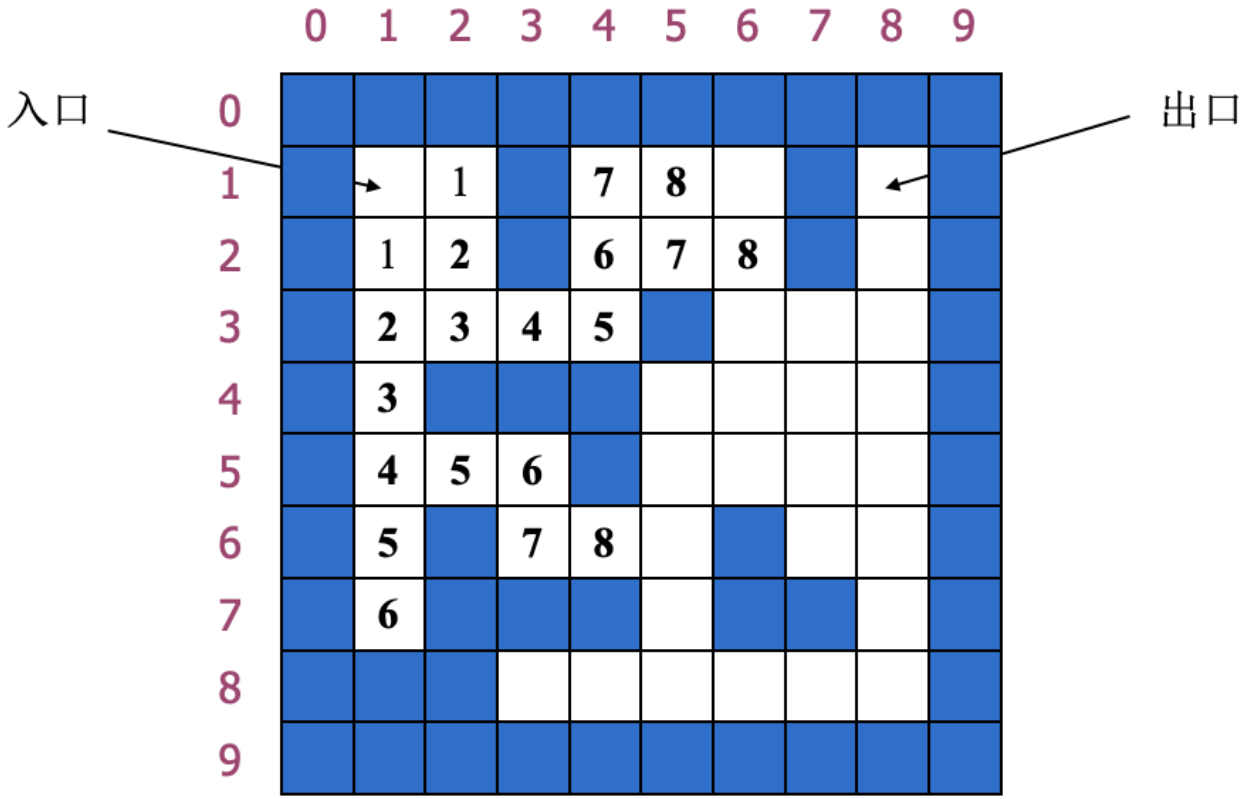
- 6 步可以到的点



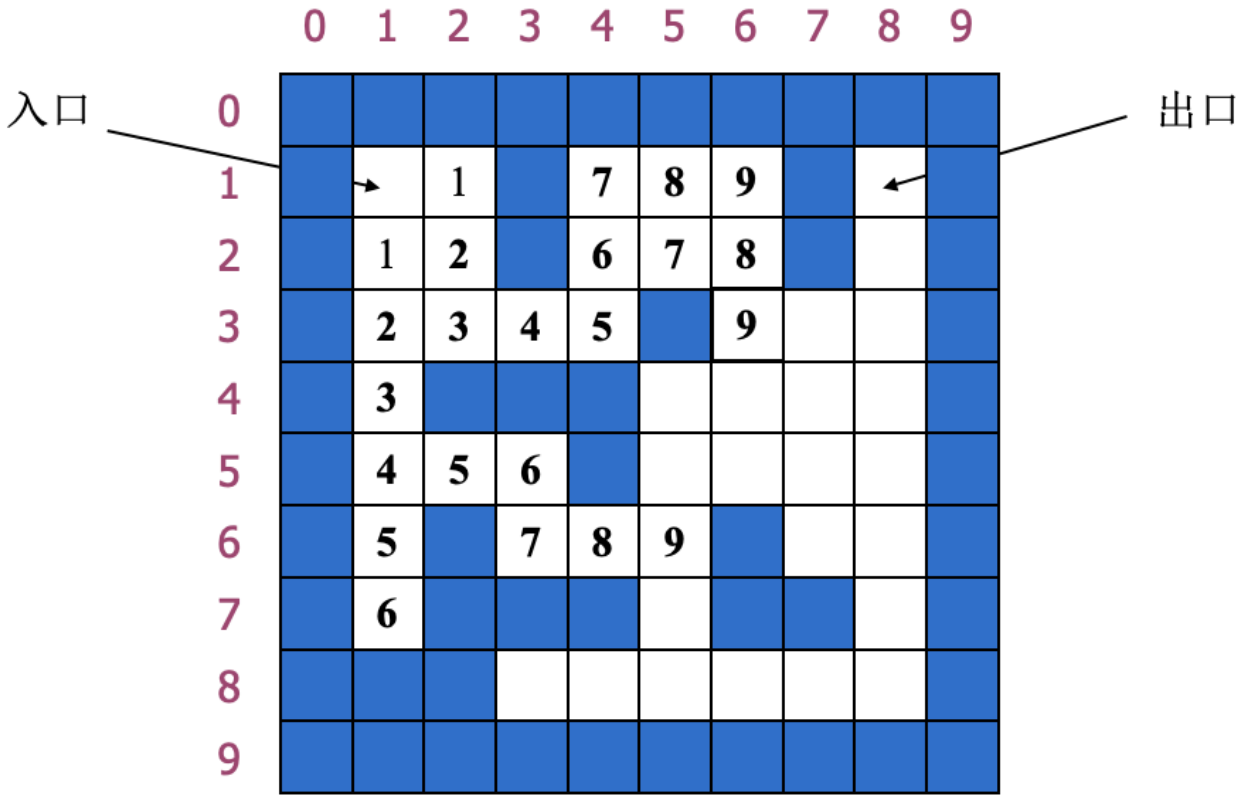
- 7 步可以到的点



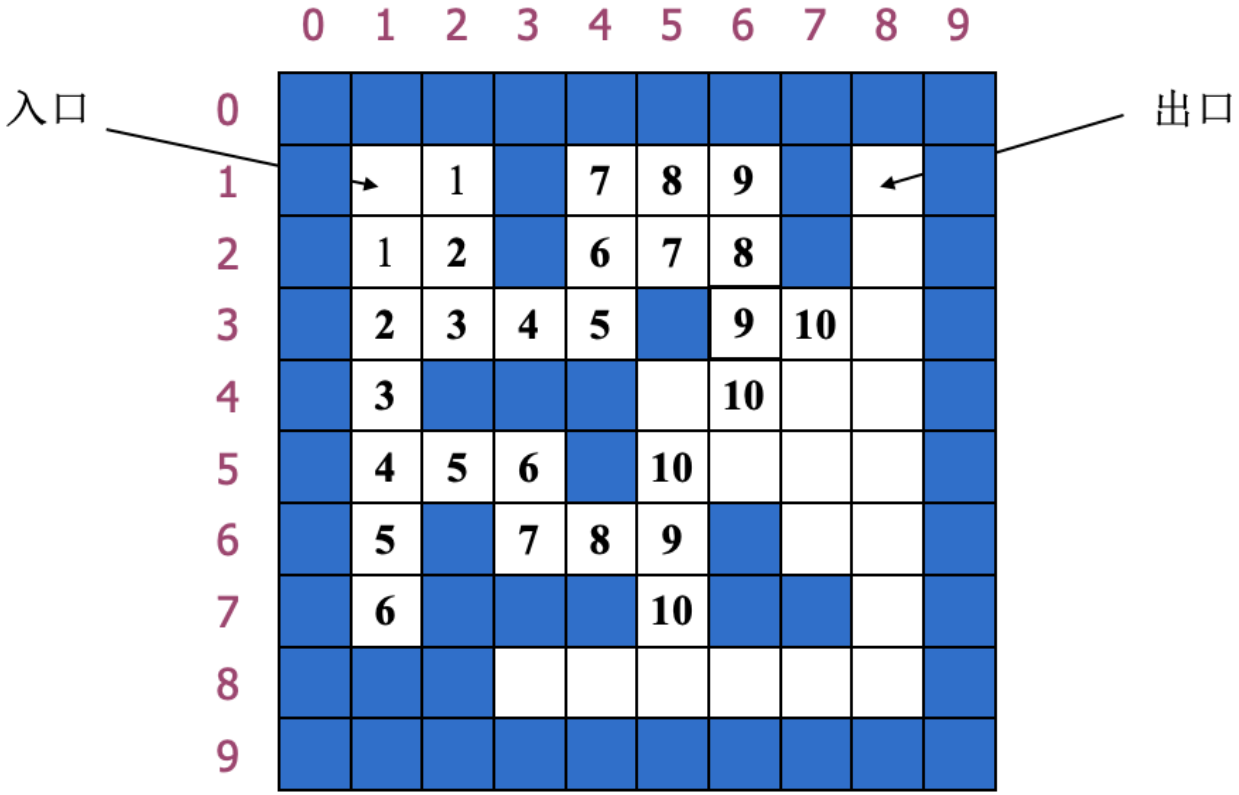
- 8 步可以到的点



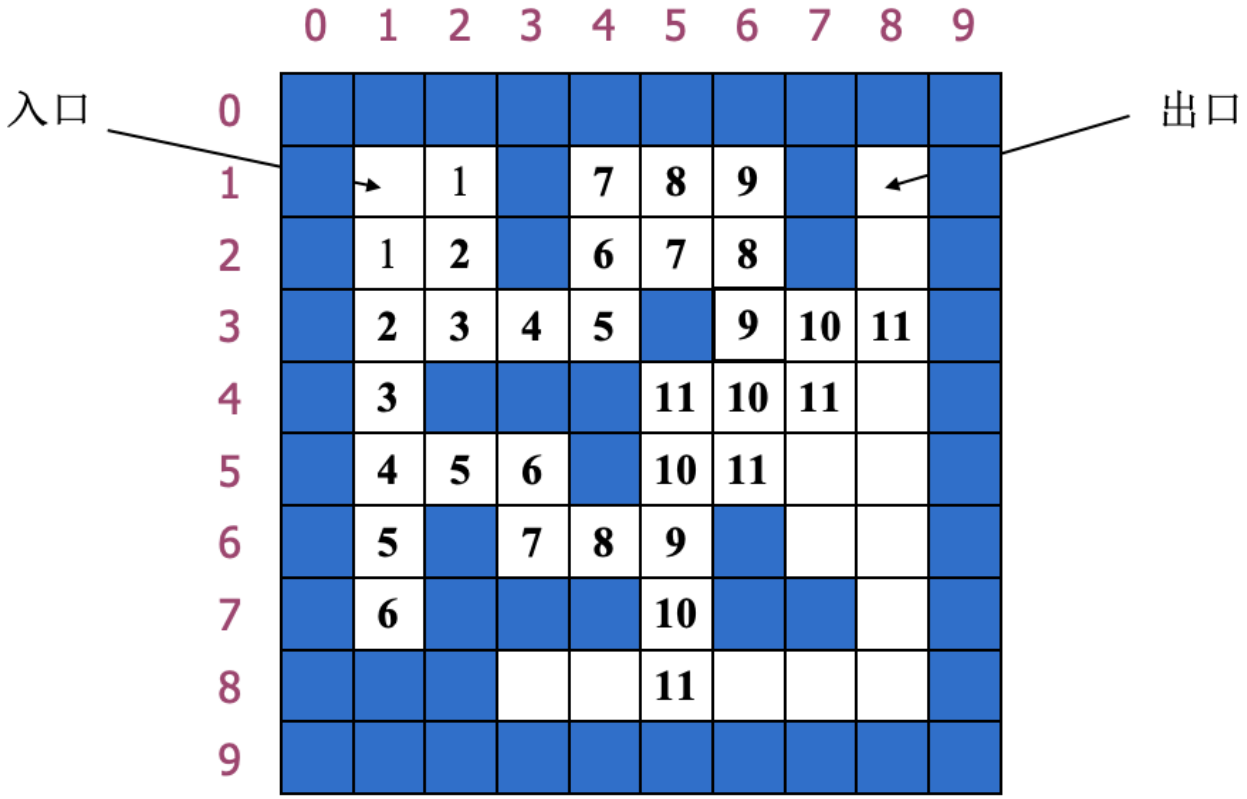
- 9 步可以到的点



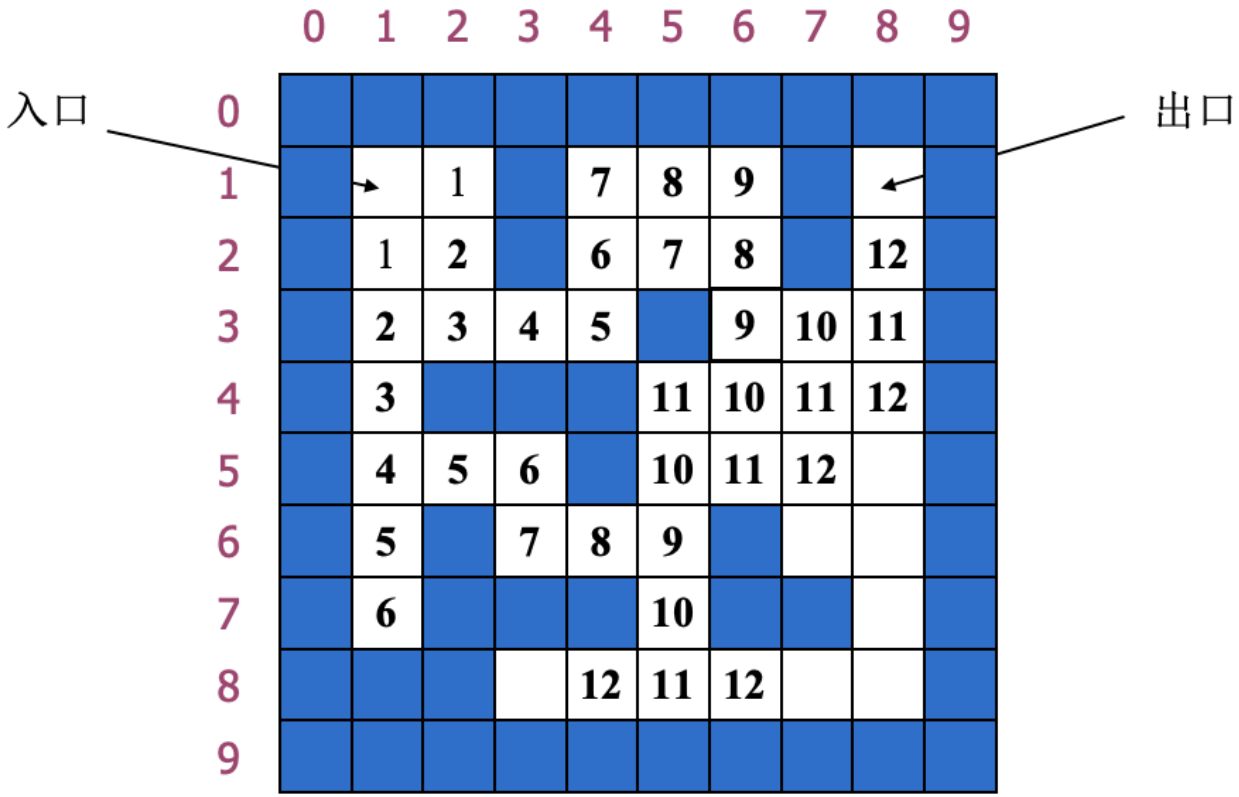
- 10 步可以到的点


0 1 2 3 4 5 6 7 8 9
入口
出口
0
1 1 7 8 9
2 1 2 6 7 8
3 2 3 4 5 9 10
4 3 10
5 4 5 6 10
6 5 7 8 9
7 6 10
8
9

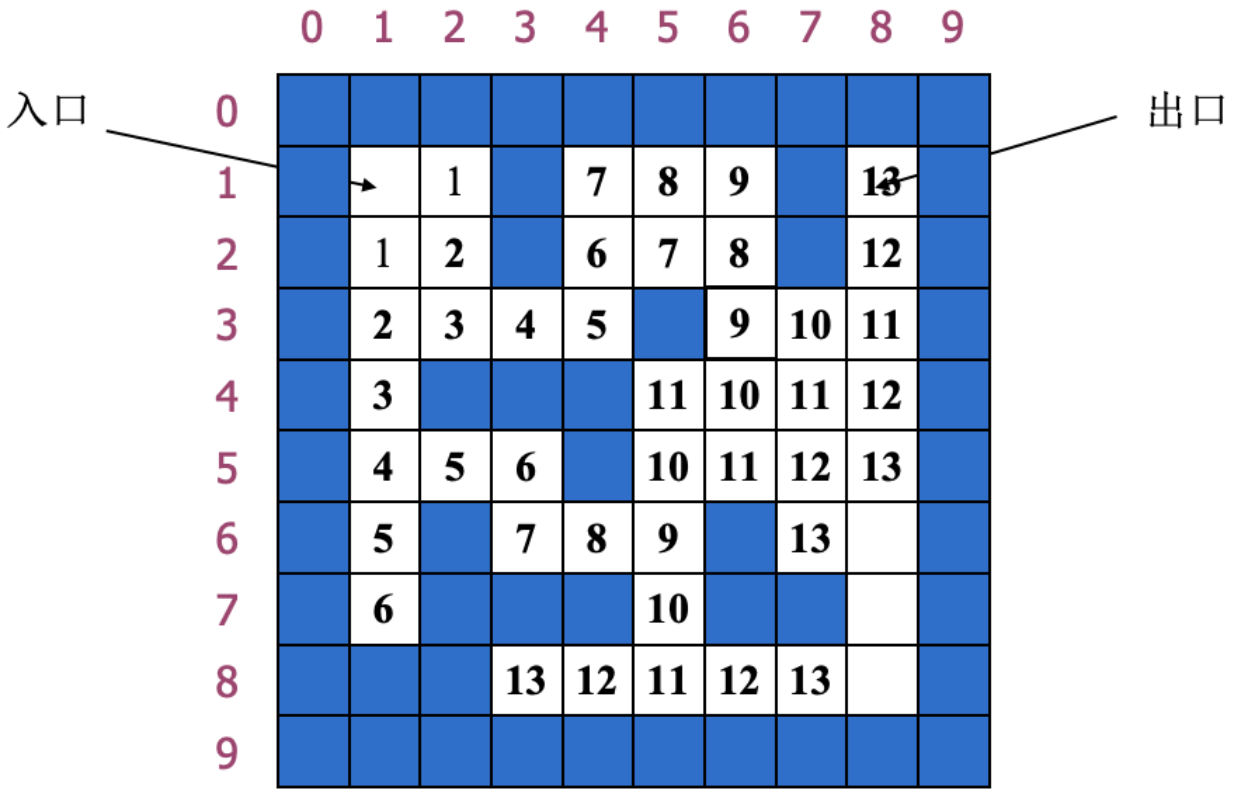
- 11 步可以到的点


入口
出口
0 1 2 3 4 5 6 7 8 9
0
1 1 7 8 9
2 1 2 6 7 8
3 2 3 4 5 9 10 11
4 3 11 10 11
5 4 5 6 10 11
6 5 7 8 9
7 6 10
8 11
9

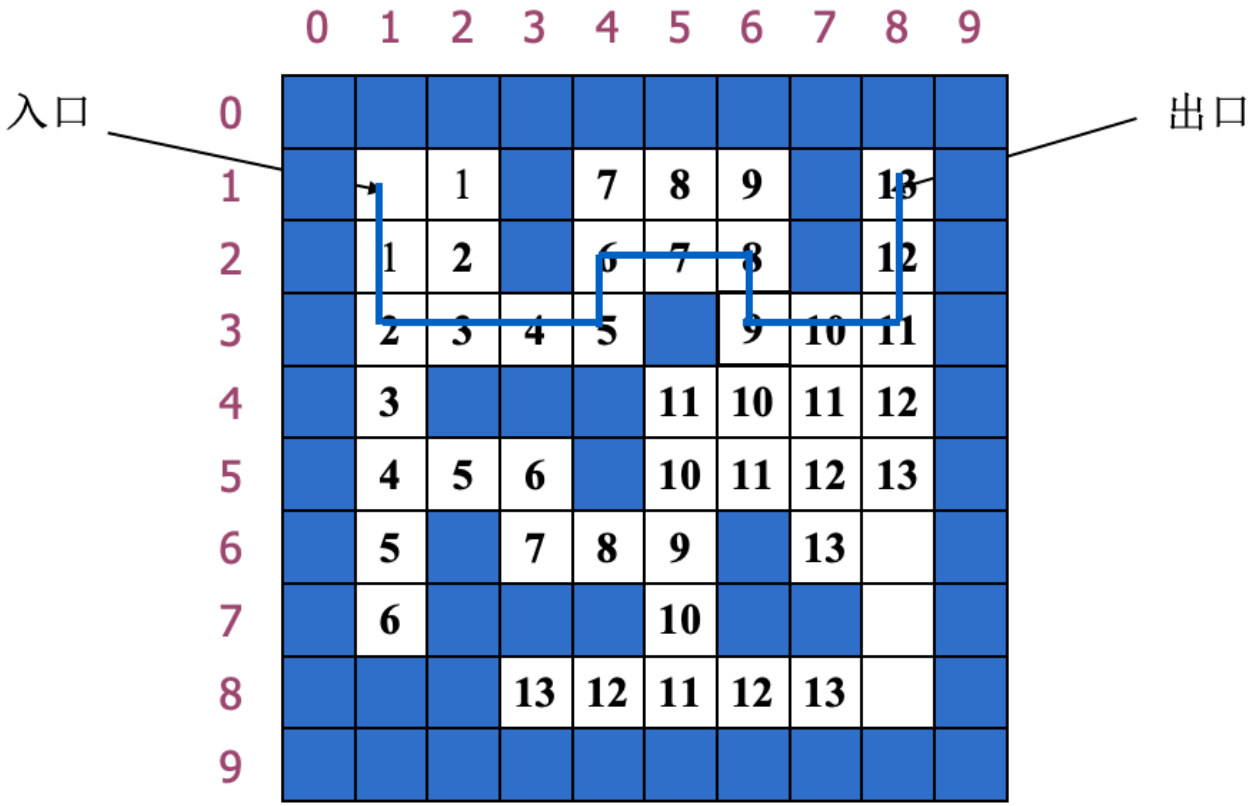
- 12 步可以到的点


入口
出口
0 1 2 3 4 5 6 7 8 9
0
1 1 7 8 9
2 1 2 6 7 8 12
3 2 3 4 5 9 10 11
4 3 11 10 11 12
5 4 5 6 10 11 12
6 5 7 8 9
7 6 10
8 12 11 12
9

- 13 步可以到的点


入口
出口
0 1 2 3 4 5 6 7 8 9
0
1 1 7 8 9 13
2 1 2 6 7 8 12
3 2 3 4 5 9 10 11
4 3 11 10 11 12
5 4 5 6 10 11 12 13
6 5 7 8 9 13
7 6 10
8 13 12 11 12 13
9

- 入口到出口的最短路径


0 1 2 3 4 5 6 7 8 9
入口
出口
0
1 1 7 8 9 13
2 1 2 6 7 8 12
3 2 3 4 5 9 10 11
4 3 11 10 11 12
5 4 5 6 10 11 12 13
6 5 7 8 9 13
7 6 10
8 13 12 11 12 13
9

BFS 算法描述

```
1  void bfs() {
2      初始化，初始状态存入队列；
3      While (队列不为空) {
4          取队头；
5          队头出队；
6          扩展下一层的子结点：
7              if (子结点符合条件 && 与队列中结点不重复) {
8                  把新结点存入列尾
9                  if (新结点是目标结点) then 输出并退出
10             }
11         }
12     }
13 }
```

【题目描述】

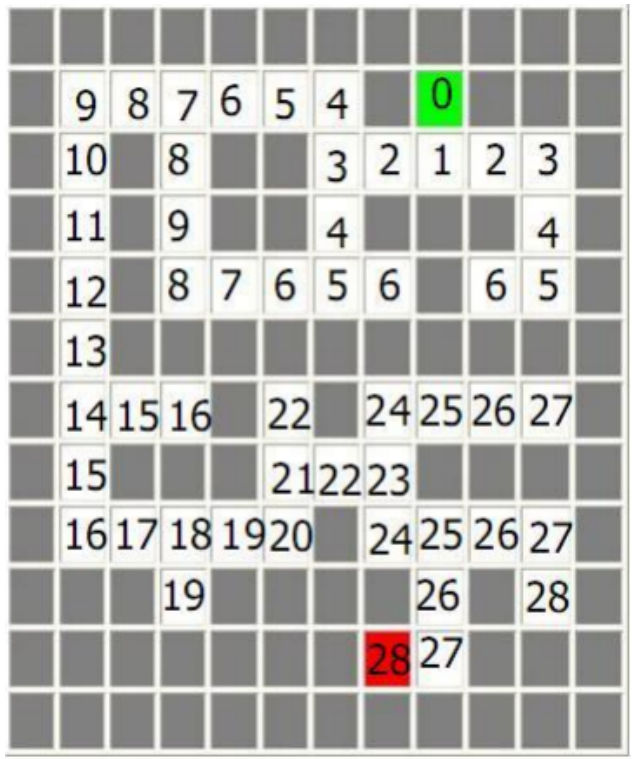
- 如下图 12×12 方格图，找出一条自入口 $(2,9)$ 到出口 $(11,8)$ 的最短路径。

【输入样例】

```
12 //迷宫大小
2 9 11 8 //起点和终点
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1 0 1 1 1
1 0 1 0 1 1 0 0 0 0 0 1
1 0 1 0 1 1 0 1 1 1 0 1
1 0 1 0 0 0 0 0 1 0 0 1
1 0 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 1 0 0 0 0 1
1 0 1 1 1 0 0 0 1 1 1 1
1 0 0 0 0 0 1 0 0 0 0 1
1 1 1 0 1 1 1 1 0 1 0 1
1 1 1 1 1 1 1 0 0 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
```

【输出样例】

```
28
```



代码实现

```cpp
const int N = 105;
struct Node {
    int x, y;
}; // 点的位置
int a[N][N]; // 表示地图，0可以走，1不可走
Node q[N * N]; // 表示队列
int d[N][N]; // 表示路径长度
int dx[4] = {-1, 1, 0, 0}; // 方向数组，上下左右
int dy[4] = {0, 0, -1, 1};
int px, py, qx, qy; // 起点(px, py)，终点(qx, qy)
void bfs() {
    int head = 1, tail = 1, tot = 0; // 初始化，初始状态存入队列;
    q[tail++] = {px, py};
    d[px][py] = 0;
    while (head < tail) { // 队列不为空
        Node t = q[head++]; // 取队头，队头出队
        for (int i = 0; i < 4; i++) {  // 扩展下一层的子结点
            int nx = t.x + dx[i];
            int ny = t.y + dy[i];
            if (check(nx, ny)) { // 检查是否合法
                q[tail++] = {nx, ny};
                a[nx][ny] = 1; // 判重标记
                d[nx][ny] = d[t.x][t.y] + 1;
                if (nx == qx && ny == qy) { // 是否为目标节点
                    cout << d[nx][ny] << endl;
                    return;
                }
            }
        }
    }
}
```

目录

康托展开

- 康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。

康托展开

- 康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。
- 什么是排列的排名呢?

康托展开

- 康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。
- 什么是排列的排名呢?
- 把 $1 \sim n$ 的所有排列按字典序排序，这个排列的位次就是它的排名。

康托展开

- 康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。
- 什么是排列的排名呢?
- 把 $1 \sim n$ 的所有排列按字典序排序，这个排列的位次就是它的排名。
- 因为排列是按字典序排名的，因此越靠前的数字优先级越高。也就是说如果两个排列的某一位之前的数字都相同，那么如果这一位如果不相同，就按这一位排序。

康托展开

- 康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。
- 什么是排列的排名呢?
- 把 $1 \sim n$ 的所有排列按字典序排序，这个排列的位次就是它的排名。
- 因为排列是按字典序排名的，因此越靠前的数字优先级越高。也就是说如果两个排列的某一位之前的数字都相同，那么如果这一位如果不相同，就按这一位排序。
- 比如 4 的排列，$[2,3,1,4] < [2,3,4,1]$，因为在第 3 位出现不同，则 $[2,3,1,4]$ 的排名在 $[2,3,4,1]$ 前面。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。
- 同理，第三位为 1，数量为 $1 \times 2!$，第四位为 1，数量为 $1 \times 1!$，第五位 0，数量为 $0 \times 0!$。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。
- 同理，第三位为 1，数量为 $1 \times 2!$，第四位为 1，数量为 $1 \times 1!$，第五位 0，数量为 $0 \times 0!$。
- 答案就是 $1 \times 4! + 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 45$ 。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。
- 同理，第三位为 1，数量为 $1 \times 2!$，第四位为 1，数量为 $1 \times 1!$，第五位 0，数量为 $0 \times 0!$。
- 答案就是 $1 \times 4! + 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 45$ 。
- 排列从 0 开始计数。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。
- 同理，第三位为 1，数量为 $1 \times 2!$，第四位为 1，数量为 $1 \times 1!$，第五位 0，数量为 $0 \times 0!$。
- 答案就是 $1 \times 4! + 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 45$ 。
- 排列从 0 开始计数。

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3 \times 3!$。
- 同理，第三位为 1，数量为 $1 \times 2!$，第四位为 1，数量为 $1 \times 1!$，第五位 0，数量为 $0 \times 0!$。
- 答案就是 $1 \times 4! + 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 45$ 。
- 排列从 0 开始计数。

总结一下，康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

康托展开

尝试计算排列 $[2,5,3,4,1]$ 的排名。

- 排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。
- 对第二位的 5 而言，它大于第一位与这个排列相同的，而这一位比 5 小的所有排列。
- 第二位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了，因此第二位为 $1,3$ 或 4。
- 第一位为 2 的所有排列都比它要小，数量为 $3\times 3!$。
- 同理，第三位为 1，数量为 $1\times 2!$，第四位为 1，数量为 $1\times 1!$，第五位 0，数量为 $0\times 0!$。
- 答案就是 $1\times 4!+3\times 3!+1\times 2!+1\times 1!+0\times 0!=45$ 。
- 排列从 0 开始计数。

总结一下，康托展开的公式为：

$$X=a_1(n-1)!+a_2(n-2)!+\cdots+a_n\cdot 0!$$

a_i 的意思是有 a_i 个数比 p_i 小并且未在前 $i-1$ 个数中出现。

康托展开

- 康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

康托展开

- 康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

- a_i 的意思是有 a_i 个数比 p_i 小并且未在前 $i-1$ 个数中出现。

康托展开

- 康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

- a_i 的意思是有 a_i 个数比 p_i 小并且未在前 $i-1$ 个数中出现。
- 尝试使用公式计算排列 $[5, 2, 4, 1, 3]$ 的康托值。

康托展开

- 康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

- a_i 的意思是有 a_i 个数比 p_i 小并且未在前 $i-1$ 个数中出现。
- 尝试使用公式计算排列 $[5, 2, 4, 1, 3]$ 的康托值。
- 答案：$4 \times 4! + 1 \times 3! + 2 \times 2! + 0 \times 1! + 0 \times 0! = 106$

- 康托展开的公式为：

$$X = a_1(n-1)! + a_2(n-2)! + \cdots + a_n \cdot 0!$$

- a_i 的意思是有 a_i 个数比 p_i 小并且未在前 $i-1$ 个数中出现。
- 尝试使用公式计算排列 $[5,2,4,1,3]$ 的康托值。
- 答案：$4 \times 4! + 1 \times 3! + 2 \times 2! + 0 \times 1! + 0 \times 0! = 106$

```
1  int Cantor(int p[], int n) {
2      int sum = 0;
3      for (int i = 1; i <= n; i++) {
4          int cnt = 0;
5          for (int j = i + 1; j <= n; j++) {
6              if (p[j] < p[i]) {
7                  cnt++;
8              }
9          }
10         sum += cnt * fact[n - i];
11     }
12     return sum + 1;
13 }
```