

---

# 第十一课      宽度优先搜索

林沐

# 内容概述

---

## 1.5道经典**宽度优先搜索**相关题目

例1:岛屿数量(medium)(深搜、宽搜)

例2:骑士移动(medium)(宽搜)

例3:词语阶梯(medium)(宽搜、图、哈希表)

例4:词语阶梯2(hard)(记录路径的宽搜、图、哈希表)

例5:收集雨水2(hard)(带优先级的宽度优先搜索、堆)

## 2.详细讲解题目**解题方法**、**代码实现**

# 例1:岛屿数量

用一个二维数组代表一张**地图**，这张地图由字符“0”与字符“1”组成，其中“0”字符代表**水域**，“1”字符代表**小岛土地**，小岛“1”被水“0”所**包围**，当小岛土地“1”在**水平和垂直**方向相连接时，认为是同一块土地。求这张地图中小岛的**数量**。

```
class Solution {
public:
    int numIslands(std::vector
        <std::vector<char> >& grid) {
    }
};
```

1个小岛:      3个小岛:

1 1 1 1 0      1 1 1 0 0

1 1 0 1 0      1 1 0 0 0

1 1 0 0 0      0 0 1 0 0

0 0 0 0 0      0 0 0 1 1

选自 **LeetCode 200. Number of Islands**

<https://leetcode.com/problems/number-of-islands/description/>

难度:**Medium**

# 例1:思考

```
class Solution {  
public:  
    int numIslands(std::vector  
        <std::vector<char> >& grid) {  
    }  
};
```


1个小岛:	3个小岛:
1 1 1 1 0	1 1 1 0 0
1 1 0 1 0	1 1 0 0 0
1 1 0 0 0	0 0 1 0 0
0 0 0 0 0	0 0 0 1 1

## 思考:

- 1.统计岛屿的数量首先要能够探索**相连接**的小岛，如何对一个**完整**的小岛进行探索？
- 2.在探索过程中，使用什么样的数据结构对**已到达的位置**进行记录？
- 3.若已知一个**起始点**，如何设计**深搜(DFS)**或**宽搜(BFS)**对某个位置进行遍历，将与该位置**相连**的位置都进行标记？
- 4.**整体的算法**是怎样的？

# 例1:分析，搜索独立小岛

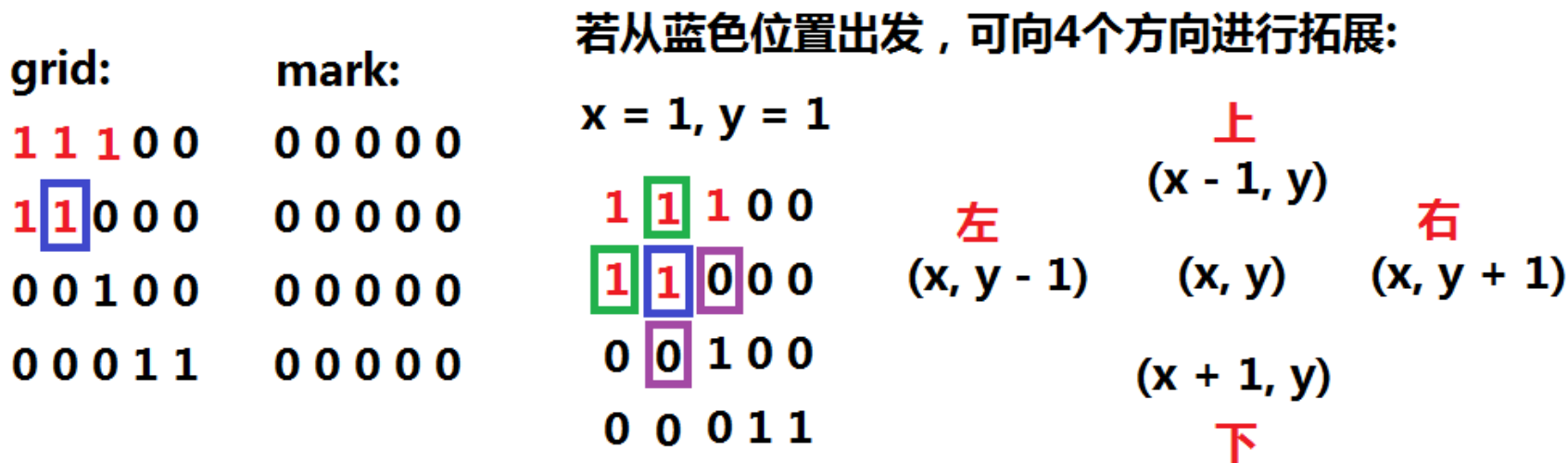
给定该二维地图grid，与一个二维标记数组mark(初始化为0)，mark数组的每个位置都与grid对应，设计一个搜索算法，从该地图中的某个岛的某个位置出发，探索该岛的全部土地，将探索到的位置在mark数组中标记为1。

grid:	mark:		grid:	mark:
1 1 1 0 0	0 0 0 0 0		1 1 1 0 0	1 1 1 0 0
1 1 0 0 0	0 0 0 0 0		1 1 0 0 0	1 1 0 0 0
0 0 1 0 0	0 0 0 0 0		0 0 1 0 0	0 0 0 0 0
0 0 0 1 1	0 0 0 0 0		0 0 0 1 1	0 0 0 0 0

```
void DFS (std::vector<std::vector<int> > &mark,  
          (BFS) std::vector<std::vector<char> > &grid,  
            int x, int y) {  
}
```

# 例1:算法设计(DFS)

1. 标记当前搜索位置**已被搜索**(标记当前位置的mark数组为1)。
2. 按照方向数组的4个方向，**拓展**4个新位置newx、newy。
3. 若新位置**不在地图范围内**，则**忽略**。
4. 如果新位置**未曾到达过**(mark[newx][newy]为0)、**且是陆地**(grid[newx][newy]为"1")，**继续DFS**该位置。



绿色位置代表可由蓝色位置拓展至，紫色代表无法进入(水域)。

# 例1:算法设计(DFS)

若按照上、下、左、右的顺序深度搜索 从(1, 1)位置出发

图1: grid: mark:

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

图3: grid: mark:

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

图5: grid: mark:

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

图2: grid: mark:

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

图4: grid: mark:

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

图6: grid: mark:

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

回到了图3，继续递归

# 例1:课堂练习(DFS)

```
void DFS(std::vector<std::vector<int> > &mark,  
        std::vector<std::vector<char> > &grid,  
        int x, int y){
```

1

```
static const int dx[] = {-1, 1, 0, 0}; //方向数组  
static const int dy[] = {0, 0, -1, 1};  
for (int i = 0; i < 4; i++){
```

```
    int newx =
```

2

```
    int newy =
```

3

```
    if (newx < 0 || newx >= mark.size() ||  
        newy < 0 || newy >= mark[newx].size()){
```

//超过数组边界时

4

```
}
```

```
if (
```

5

```
    DFS(mark, grid, newx, newy);
```

//??情况时 继续深搜

```
}
```

```
}
```

```
}
```

3分钟填写代码，  
有问题随时提出！



# 例1:实现(DFS)

```
void DFS (std::vector<std::vector<int> > &mark,  
         std::vector<std::vector<char> > &grid,  
         int x, int y) {  
    mark[x][y] = 1; //标记已搜寻的位置  
  
    static const int dx[] = {-1, 1, 0, 0}; //方向数组  
    static const int dy[] = {0, 0, -1, 1};  
    for (int i = 0; i < 4; i++) {  
        int newx = dx[i] + x;  
        int newy = dy[i] + y;  
  
        if (newx < 0 || newx >= mark.size() ||  
            newy < 0 || newy >= mark[newx].size()) {  
            continue; //超过数组边界时  
        }  
        if (mark[newx][newy] == 0 && grid[newx][newy] == '1') {  
            DFS (mark, grid, newx, newy); //??情况时 继续深搜  
        }  
    }  
}
```

# 例1:算法设计(BFS)

1. 设置搜索**队列Q**，标记 $make[x][y] = 1$ ，并将**待搜索的位置** $(x, y)$  push进入队列Q。
2. 只要队列不空，即**取队头元素**，按照方向数组的4个方向，**拓展**4个新位置 $newx$ 、 $newy$ 。
3. 若新位置**不在地图范围内**，则**忽略**。
4. 如果新位置**未曾到达过**( $mark[newx][newy]$ 为0)、且**是陆地**( $grid[newx][newy]$ 为“1”)；将该新位置push进入队列，并标记 $mark[newx][newy] = 1$ 。

grid:

1 1 1 0 0

1 1 0 0 0

0 0 1 0 0

0 0 0 1 1

mark:

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

若从蓝色位置出发，可向4个方向进行拓展：

$x = 1, y = 1$

1 1 1 0 0

1 1 0 0 0

0 0 1 0 0

0 0 0 1 1

上  
 $(x - 1, y)$   
左  
 $(x, y - 1)$   $(x, y)$   $(x, y + 1)$  右  
 $(x + 1, y)$   
下

绿色位置代表可由蓝色位置拓展至，紫色代表无法进入(水域)。

# 例1:算法设计(BFS)

若按照上、下、左、右的顺序深度搜索 从(1, 1)位置出发

图1: grid: mark:

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

图3: grid: mark:

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

图5: grid: mark:

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

图2: grid: mark:

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

图4: grid: mark:

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

图6: grid: mark:

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

# 例1:课堂练习(BFS)

```
void BFS(std::vector<std::vector<int> > &mark,  
        std::vector<std::vector<char> > &grid,  
        int x, int y){  
    static const int dx[] = {-1, 1, 0, 0}; //方向数组  
    static const int dy[] = {0, 0, -1, 1};  
    std::queue<std::pair<int, int> > Q; //宽度优先搜索队列  
    Q.push(std::make_pair(x, y)); //将(x, y)push进入队列，并做标记  
    mark[x][y] = 1;
```

```
    while(!Q.empty()){  
        x = Q.front().first;  
        y = Q.front().second; //取队列头部元素
```

1

```
    for (int i = 0; i < 4; i++){  
        int newx = dx[i] + x; //拓展四个方向  
        int newy = dy[i] + y; //忽略越过地图边界的位置  
        if (newx < 0 || newx >= mark.size() ||  
            newy < 0 || newy >= mark[newx].size()){  
            continue;  
        } //如果当前位置未搜索，且为陆地  
        if (mark[newx][newy] == 0 && grid[newx][newy] == '1'){
```

2

3

```
    }
```

```
}
```

```
}
```

```
}
```

3分钟填写代码，  
有问题随时提出！

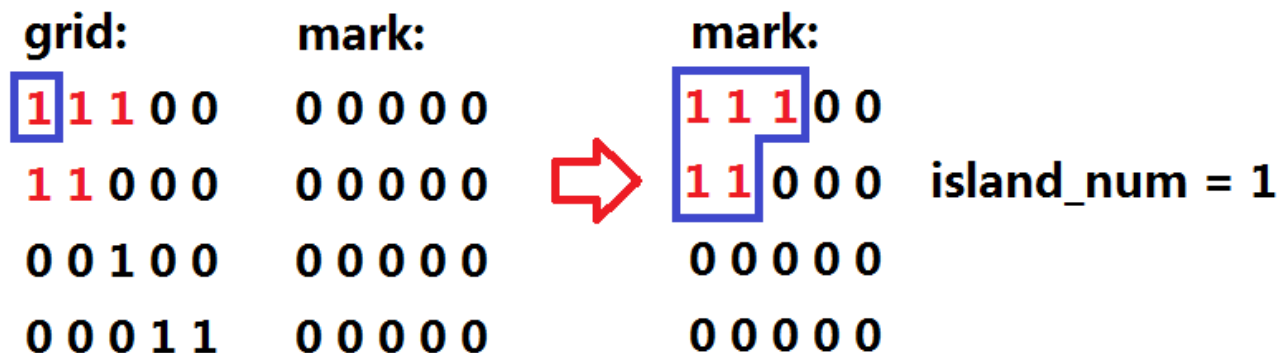
# 例1:实现(BFS)

```
void BFS(std::vector<std::vector<int> > &mark,
        std::vector<std::vector<char> > &grid,
        int x, int y) {
    static const int dx[] = {-1, 1, 0, 0}; //方向数组
    static const int dy[] = {0, 0, -1, 1};
    std::queue<std::pair<int, int> > Q; //宽度优先搜索队列
    Q.push(std::make_pair(x, y)); //将(x, y)push进入队列，并做标记
    mark[x][y] = 1;
    while (!Q.empty()) {
        x = Q.front().first; //取队列头部元素
        y = Q.front().second;
        Q.pop(); //弹出队头元素
        for (int i = 0; i < 4; i++) {
            int newx = dx[i] + x; //拓展四个方向
            int newy = dy[i] + y; //忽略越过地图边界的位置
            if (newx < 0 || newx >= mark.size() ||
                newy < 0 || newy >= mark[newx].size()) {
                continue;
            } //如果当前位置未搜索，且为陆地
            if (mark[newx][newy] == 0 && grid[newx][newy] == '1') {
                Q.push(std::make_pair(newx, newy)); //将新位置push进入队列
                mark[newx][newy] = 1; //并做标记
            }
        }
    }
}
```

# 例1:算法设计(整体)

求地图中**岛屿的数量**:


1. 设置岛屿数量 **island\_num** = 0;
2. 设置mark数组, 并**初始化**。
3. 遍历**地图grid**的所有点, 如果当前点是**陆地**, 且**未被访问**过, 调用**搜索**接口 `search(mark, grid, i, j)`; `search`可以是DFS或BFS; **完成**搜索后 `island_num++`。




grid:	grid:	grid:	grid:
1 <b>1</b> 1 0 0	1 1 <b>1</b> 0 0	1 1 1 0 0	1 1 1 0 0
1 1 0 0 0	1 1 0 0 0	<b>1</b> 1 0 0 0	1 <b>1</b> 0 0 0
0 0 1 0 0	0 0 1 0 0	0 0 1 0 0	0 0 1 0 0
0 0 0 1 1	0 0 0 1 1	0 0 0 1 1	0 0 0 1 1

绿色的点都不会被搜索!

# 例1:算法设计(整体)

grid:	mark:		mark:	
1 1 1 0 0	1 1 1 0 0		1 1 1 0 0	island_num = 2
1 1 0 0 0	1 1 0 0 0		1 1 0 0 0	
0 0 1 0 0	0 0 0 0 0		0 0 1 0 0	
0 0 0 1 1	0 0 0 0 0		0 0 0 0 0	

grid:	mark:		mark:	
1 1 1 0 0	1 1 1 0 0		1 1 1 0 0	island_num = 3
1 1 0 0 0	1 1 0 0 0		1 1 0 0 0	
0 0 1 0 0	0 0 1 0 0		0 0 1 0 0	
0 0 0 1 1	0 0 0 0 0		0 0 0 1 1	

# 例1:课堂练习

```
class Solution {
public:
    int numIslands(std::vector<std::vector<char>> & grid) {
        int island_num = 0;
        std::vector<std::vector<int>> mark;
        for (int i = 0; i < grid.size(); i++){
            mark.push_back(std::vector<int>());
            for (int j = 0; j < grid[i].size(); j++){
                1
            }
        }
        for (int i = 0; i < grid.size(); i++){
            for (int j = 0; j < grid[i].size(); j++){
                if ( 2 ) {
                    //或BFS DFS(mark, grid, i, j);
                    3
                }
            }
        }
        return island_num;
    }
};
```

3分钟填写代码，  
有问题随时提出！



# 例1:实现

```
class Solution {
public:
    int numIslands(std::vector<std::vector<char> >& grid) {
        int island_num = 0;
        std::vector<std::vector<int> > mark;
        for (int i = 0; i < grid.size(); i++){
            mark.push_back(std::vector<int>());
            for (int j = 0; j < grid[i].size(); j++){
                mark[i].push_back(0);
            }
        }
        for (int i = 0; i < grid.size(); i++){
            for (int j = 0; j < grid[i].size(); j++){
                if (mark[i][j] == 0 && grid[i][j] == '1') {
                    //或BFS DFS(mark, grid, i, j);
                    island_num++;
                }
            }
        }
        return island_num;
    }
};
```

# 例1:测试与leetcode提交结果

```
int main() {
    std::vector<std::vector<char> > grid;
    char str[10][10] = {"11100", "11000", "00100", "00011"};
    for (int i = 0; i < 4; i++) {
        grid.push_back(std::vector<char>());
        for (int j = 0; j < 5; j++) {
            grid[i].push_back(str[i][j]);
        }
    }
    Solution solve;
    printf("%d\n", solve.numIslands(grid));
    return 0;
}
```

Number of Islands

Submission Details

47 / 47 test cases passed.

Status: **Accepted**

Runtime: 16 ms

Submitted: 0 minutes ago

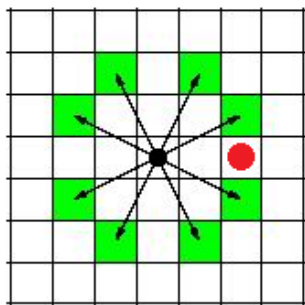
3

请按任意键继续 . . .

# 例2:骑士移动

已知一个 $n*n$ 的棋盘，在这个棋盘上设置两个位置，**起始位置**(beginx, beginy)与**终点位置**(endx, endy)，x代表是第x行，y代表是第y列，骑士每次可以按照下图的8个方向进行**跳跃**，求起始位置跳跃至终点位置**最少用几步**？(n最大不超过300)。思考该问题可否使用**DFS**？

**7\*7的棋盘，骑士在某处可以跳跃的8个位置**



骑士的当前位置是(4, 4) ,  
若要跳跃到(4, 6) , 至少需要2步。

**//起始位置的行与列**

**//终点位置的行与列**

```
int BFS_move(int beginx, int beginy, int endx, int endy, int n) {  
    //函数返回从起始位置到终点位置最少要用多少步  
    //棋盘是n*n的大小
```

选自 **poj1915**

<http://poj.org/problem?id=1915>

难度:**Medium**

# 例2:思考

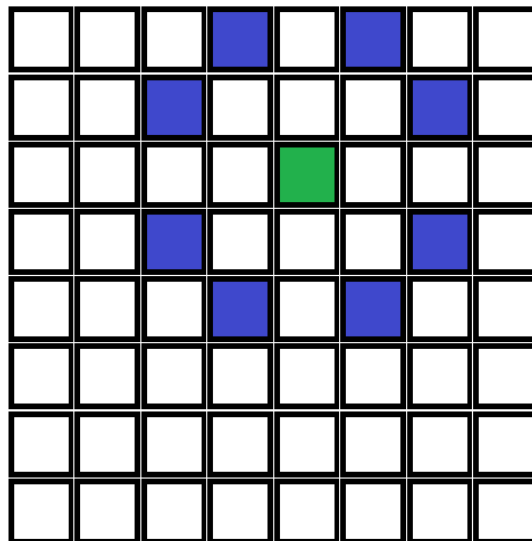
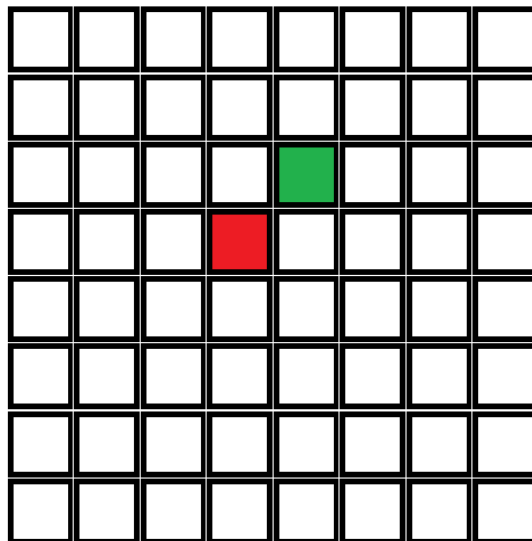
一个8\*8的棋盘，绿色标记了骑士**起始位置**为(3, 5)，即beginx = 3, beginy = 5；红色标记了骑士的**目标位置**为(4, 4)，即endx = 4, endy = 4；求，从**绿色位置**到**红色位置**，骑士需要**至少**跳跃多少步？**蓝色位置**标记了骑士第一次跳跃可以达到的位置。

**思考**如下问题：

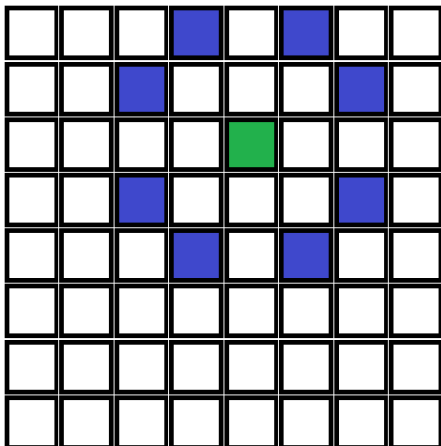
1. 棋盘使用怎样的**数据结构**进行表示？是否需要对**已经跳跃过的地方**进行标记，如何进行记录。
2. 根据骑士的**初始位置**，如何计算骑士进行一次跳跃后的**8个位置坐标**？
3. 是否有可能有些位置**永远达不到**？如何判断某位置是否在**棋盘内部**？
4. **整体算法**是怎样的，如何在BFS时记录跳跃的步数，设计整体BFS算法解决该问题。

**8\*8的棋盘：**

从**绿色位置**向**红色位置**行走，  
至少需要多少步？



# 例2:分析



骑士从绿色位置到蓝色位置，设绿色位置(x, y)  
蓝色位置:

(x - 2, y - 1)      (x + 2, y + 1)

(x - 1, y - 2)      (x + 1, y + 2)

(x + 1, y - 2)      (x - 1, y + 2)

(x + 2, y - 1)      (x - 2, y + 1)

故设置方向数组，来辅助坐标更新：

**static const int dx[] = {-2, -1, 1, 2, 2, 1, -1, -2};**

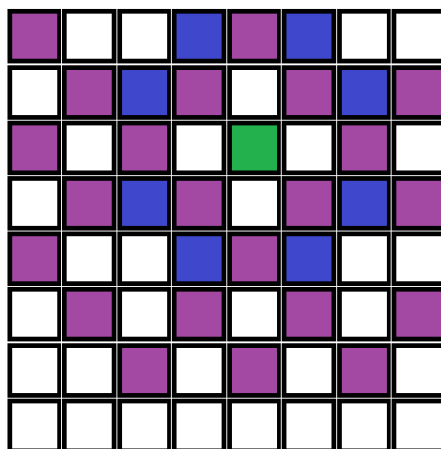
**static const int dy[] = {-1, -2, -2, -1, 1, 2, 2, 1};**

队列节点数据结构，包括位置(x, y)，  
和到达这个位置所需要的步数cnt

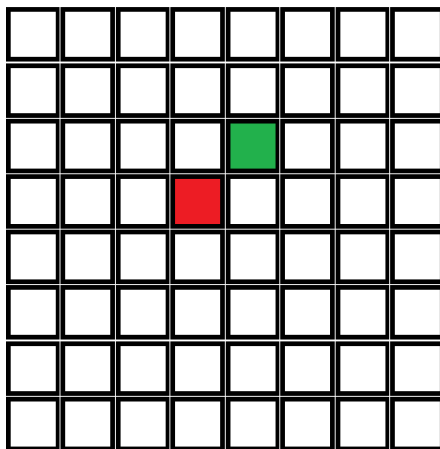
```
struct location{
    int x;
    int y;
    int cnt;
    location (int _x, int _y, int _cnt) :
        x(_x), y(_y), cnt(_cnt){
    }
};

std::queue<location> Q;
```

紫色位置是第二次跳跃后达到的位置



故从绿色位置达到红色位置需要1步：



mark数组:

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

# 例2:算法设计

1. 设置记录哪个位置**已跳跃的mark数组**，`int mark[MAXN][MAXN]`，`MAXN = 310`；
2. 设置队列`std::queue<location> Q`，队列节点中有**坐标(x, y)**与到达这个坐标的**步数cnt**。
3. 将**初始节点**放入队列，到达初始节点的步数是0，标记初始节点的位置**mark数组**为1。
4. **循环**，只要队列不空：

取队头元素，包括**位置(locx, locy)**，与到达该位置的**步数loccnt**，弹出队头；  
如果当前取出的节点位置即为最终**目标位置**，返回到达当前位置的**步数**。

根据当前(`locx, locy`)，按照方向数组**拓展8个方向**的新坐标(`newx, newy`)

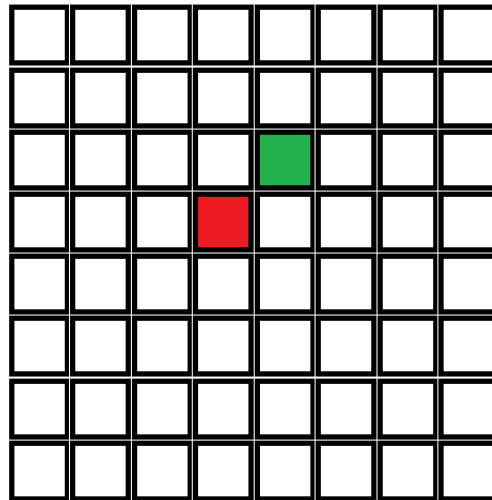
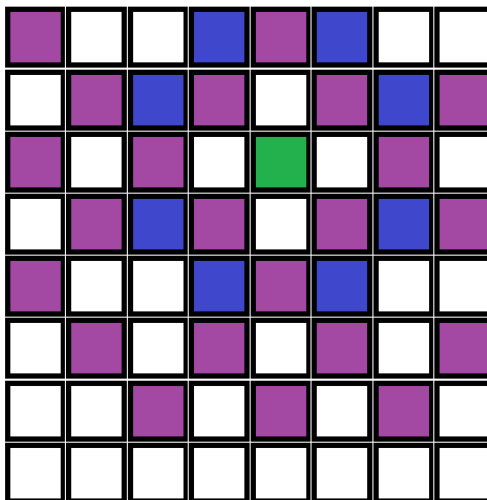
如果(`newx, newy`)**超出地图范围**或**曾经到达过**，忽略该位置；

将新位置(`newx, newy`)与到达新位置的步数push进入队列；**标记该位置已到达**。

队列节点数据结构，包括位置(x, y)，**紫色位置是第二次跳跃后达到的位置** 故从绿色位置达到红色位置需要1步：  
和到达这个位置所需要的步数cnt

```
struct location{
    int x;
    int y;
    int cnt;
    location (int _x, int _y, int _cnt) :
        x(_x), y(_y), cnt(_cnt){
    }
};

std::queue<location> Q;
```



# 例2:课堂练习

```
int BFS_move(int beginx, int beginy, int endx, int endy, int n){
    int mark[MAXN][MAXN] = {0};
    std::queue<location> Q;
    location loc(beginx, beginy, 0);
    Q.push(loc);
```

1

```
while(!Q.empty()){
    loc = Q.front();
    Q.pop();
    if (loc.x == endx && loc.y == endy){
```

2

```
    }
    for (int i = 0; i < 8; i++){
        int newx = dx[i] + loc.x;
        int newy =
```

3

```
        if (newx < 0 || newy < 0 ||
            newx >= n || newy >= n || mark[newx][newy]){
```

4

```
        }
        Q.push(location(newx, newy,
```

5

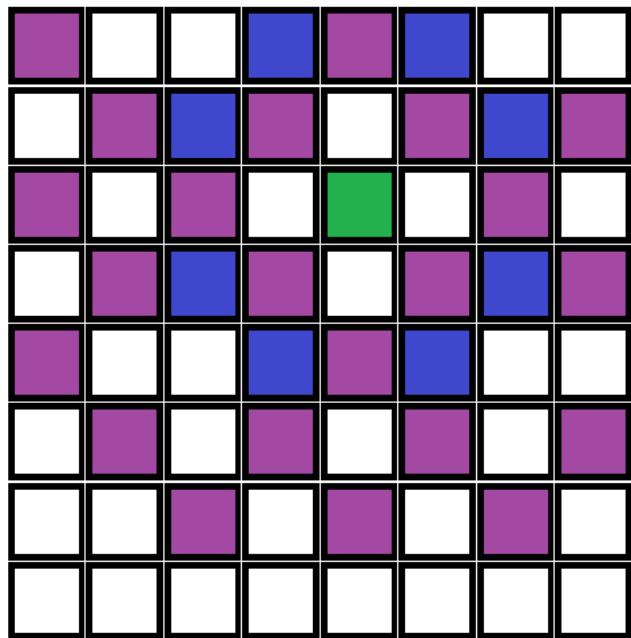
```
        mark[newx][newy] = 1;
    }
}
return -1;
```

3分钟填写代码，  
有问题随时提出！



# 例2:实现

```
int BFS_move(int beginx, int beginy, int endx, int endy, int n){
    int mark[MAXN][MAXN] = {0};
    std::queue<location> Q;
    location loc(beginx, beginy, 0);
    Q.push(loc);
    mark[beginx][beginy] = 1;
    while(!Q.empty()){
        loc = Q.front();
        Q.pop();
        if (loc.x == endx && loc.y == endy){
            return loc.cnt;
        }
        for (int i = 0; i < 8; i++){
            int newx = dx[i] + loc.x;
            int newy = dy[i] + loc.y;
            if (newx < 0 || newy < 0 ||
                newx >= n || newy >= n || mark[newx][newy]){
                continue;
            }
            Q.push(location(newx, newy, loc.cnt + 1));
            mark[newx][newy] = 1;
        }
    }
    return -1;
}
```





# 例2:poj测试与提交

linmu89

1915

Accepted

632K

485MS

C++

1113B

## Sample Input

```
3
8
0 0
7 0
100
0 0
30 50
10
1 1
1 1
```



## Sample Output

```
5
28
0
```

```
int main() {
    int t;
    int n;
    int beginx;
    int beginy;
    int endx;
    int endy;
    scanf("%d", &t);
    while(t--) {
        scanf("%d %d %d %d %d", &n, &beginx, &beginy, &endx, &endy);
        printf("%d\n", BFS_move(beginx, beginy, endx, endy, n));
    }
    return 0;
}
```

# 例3:词语阶梯

已知两个单词(分别是起始单词与结束单词)，一个**单词词典**，根据**转换规则**计算从起始单词到结束单词的**最短转换步数**。

转换规则如下：

- 1.在转换时，只能转换单词中的**1个字符**。
- 2.转换得到的**新单词**，必须在单词词典中。

例如: beginWord = “hit”； endWord = “cog”； wordList = ["hot","dot","dog","lot","log","cog"]  
最短转换方式: "hit" -> "hot" -> "dot" -> "dog" -> "cog", 结果为5。

```
class Solution {  
public:  
    int ladderLength(std::string beginWord, std::string endWord,  
                    std::vector<std::string>& wordList) {  
    }  
};
```

1.若无法转换到endWord，返回0

2.所有的单词长度相同

4.wordList中无重复单词

3.只包含小写字符

5.beginWord与endWord非空，且不相同

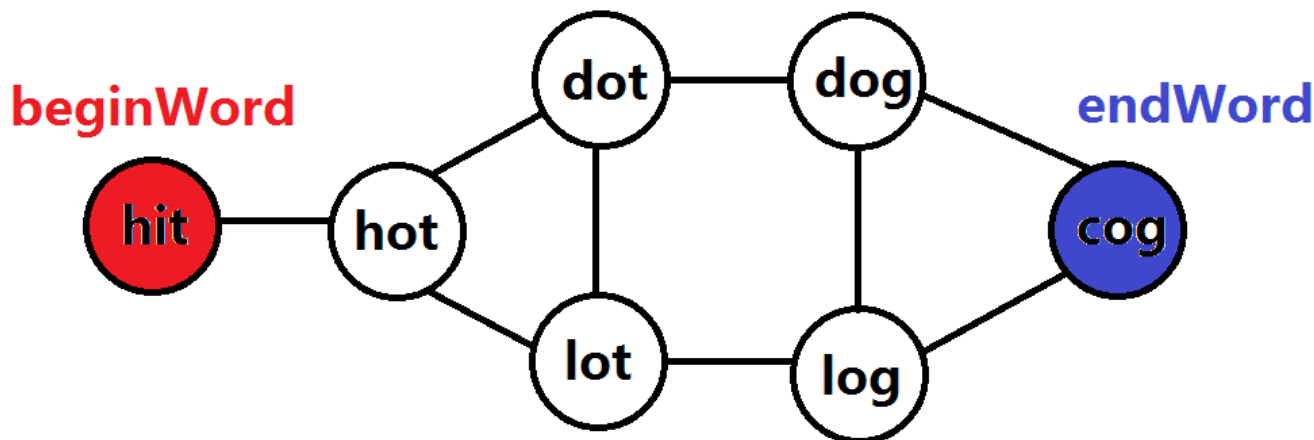
选自 **LeetCode 127. Word Ladder**

<https://leetcode.com/problems/word-ladder/description/>

难度:**Medium**

# 例3:思考与分析

单词与单词之间的**转换**，可以理解为一张**图**，图的**顶点**为单词，若两单词之间可以**互相转换**，则这两个单词所代表的**顶点间有一条边**，求图中节点hit(beginWord)到节点cog(endWord)的**所有路径**中，**最少**包括多少个节点。即图的**宽度优先**搜索。

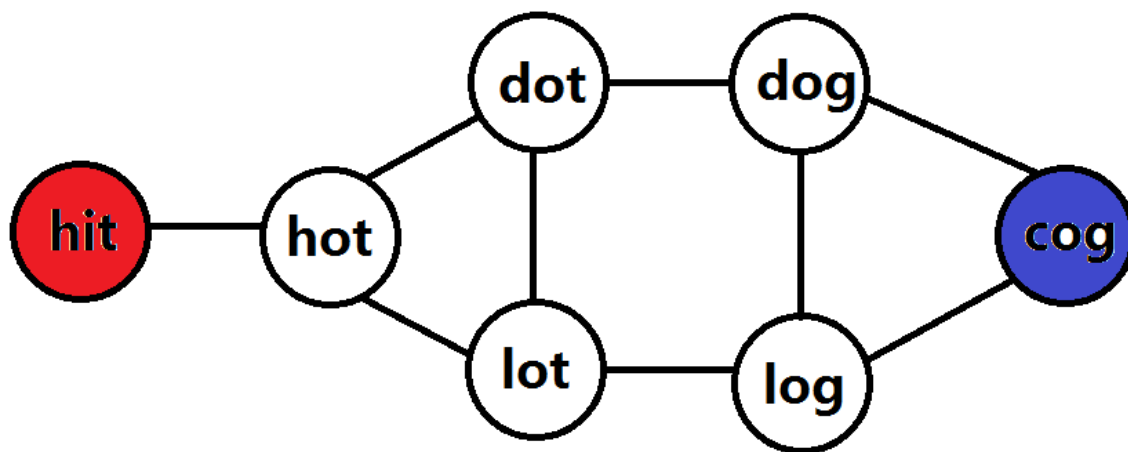


路径1: hit hot dot dog cog

路径2: hit hot lot log cog

# 例3:算法设计，图的表示与构造

使用map构造邻接表表示的图，map定义为以string为key(代表图的顶点)，vector<string>为value(代表图的各个顶点临接的顶点)。如下图所示：



Graph:

hot	→	hit	dot	lot
dot	→	hot	log	dog
dog	→	dot	log	cog
lot	→	hot	dot	log
log	→	lot	dog	cog
cog	→	dog	log	
hit	→	hot		

将beginWord push进入wordList。遍历wordList，对任意两个单词wordList[i]与wordList[j]，若wordList[i]与wordList[j]只相差1个字符，则将其相连。

## 例3:课堂练习

```
bool connect(const std::string &word1, const std::string &word2){
    int cnt = 0; //记录word1与word2不相等字符的个数
    for (int i = 0; i < word1.length(); i++){
        if ( 1 ){
            cnt++;
        }
    }
    return cnt == 1;
}

void construct_graph(std::string &beginWord,
                    std::vector<std::string>& wordList,
                    std::map<std::string, std::vector<std::string> > &graph){
    wordList.push_back( 2 );
    for (int i = 0; i < wordList.size(); i++){
        3 = std::vector<std::string>();
    }
    for (int i = 0; i < wordList.size(); i++){
        for (int j = 4; j < wordList.size(); j++){
            if (connect(wordList[i], 5)){
                graph[wordList[i]].push_back(wordList[j]);
                graph[wordList[j]].push_back(wordList[i]);
            }
        }
        //对任意两个单词wordList[i]与wordList[j]，若
        //wordList[i]与wordList[j]只相差1个字符，则将其相
        //连。
    }
}
```

3分钟填写代码，  
有问题随时提出！

```

bool connect(const std::string &word1, const std::string &word2){
    int cnt = 0; //记录word1与word2不相等字符的个数
    for (int i = 0; i < word1.length(); i++){
        if (word1[i] != word2[i]){
            cnt++;
        }
    }
    return cnt == 1;
}

void construct_graph(std::string &beginWord,
                    std::vector<std::string>& wordList,
                    std::map<std::string, std::vector<std::string> > &graph){
    wordList.push_back(beginWord);

    for (int i = 0; i < wordList.size(); i++){
        graph[wordList[i]] = std::vector<std::string>();
    }
    for (int i = 0; i < wordList.size(); i++){
        for (int j = i + 1; j < wordList.size(); j++){
            if (connect(wordList[i], wordList[j])){
                graph[wordList[i]].push_back(wordList[j]);
                graph[wordList[j]].push_back(wordList[i]);
            }
        }
    }
}

```

**例3:实现**

**//对任意两个单词wordList[i]与wordList[j]，若wordList[i]与wordList[j]只相差1个字符，则将其相连。**

# 例3:算法设计，图的宽度遍历

给定图的**起点**beginWord，**终点**endWord，**图**graph，从beginWord开始**宽度优先搜索**图graph，搜索过程中记录到达**步数**；

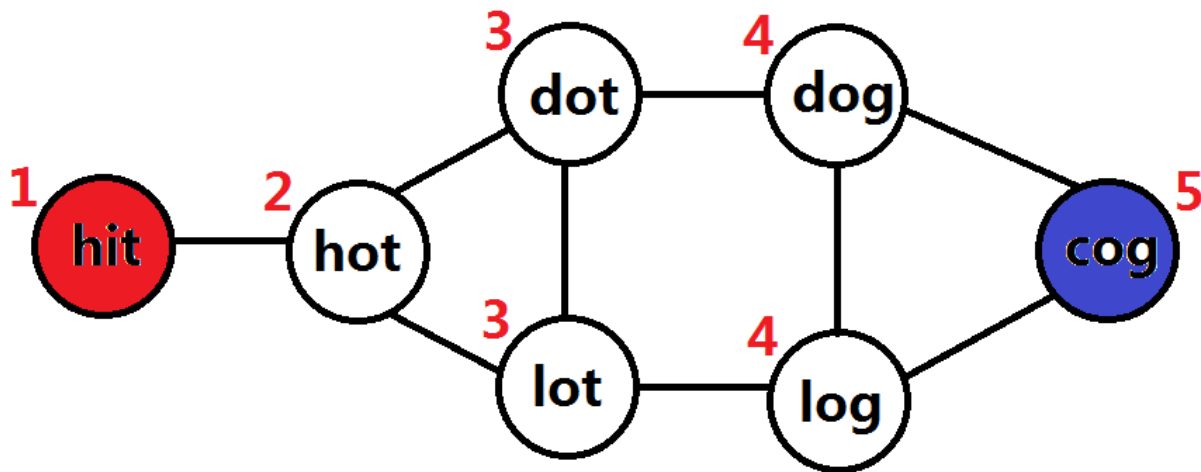
1.设置搜索**队列Q**，队列节点为pair<顶点，步数>；设置**集合visit**，记录搜索过的顶点；将<beginWord, 1>添加至队列；

2.只要队列**不空**，取出队列头部元素：

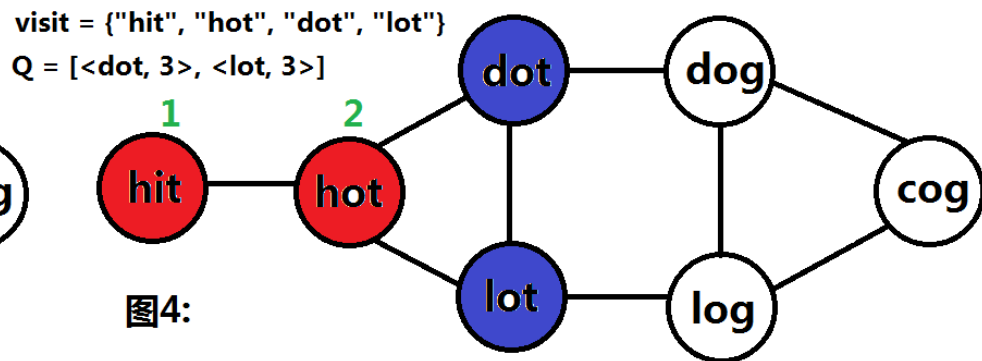
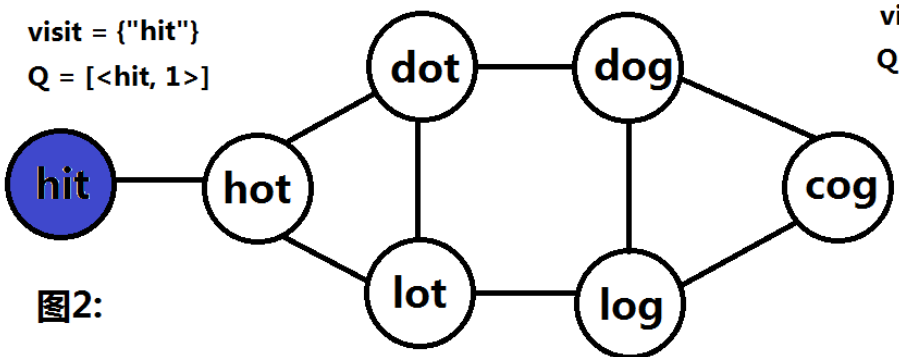
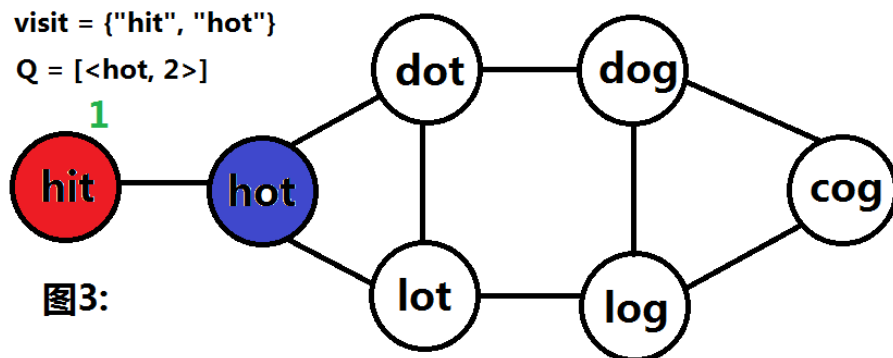
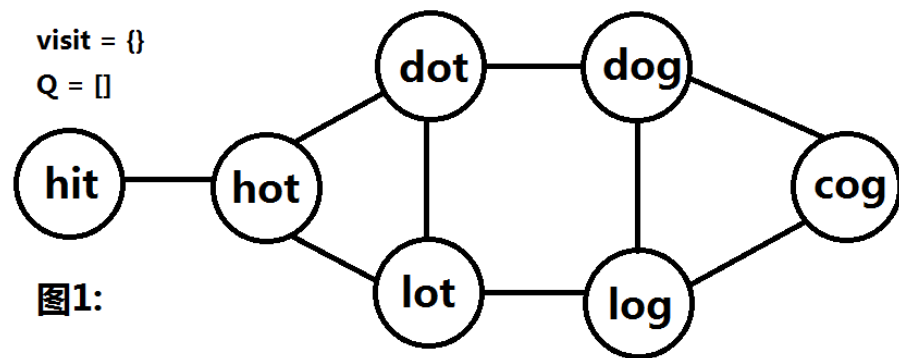
1)若取出的队列头部元素为endWord，返回**到达当前节点的步数**；

2)否则**拓展**该节点，将与该节点**相邻的**且**未添加**到visit中的节点与步数同时添加至队列Q，并将**拓展节点**加入visit；

3.若最终都无法搜索到endWord，**返回0**。



# 例3:算法设计，图的宽度遍历

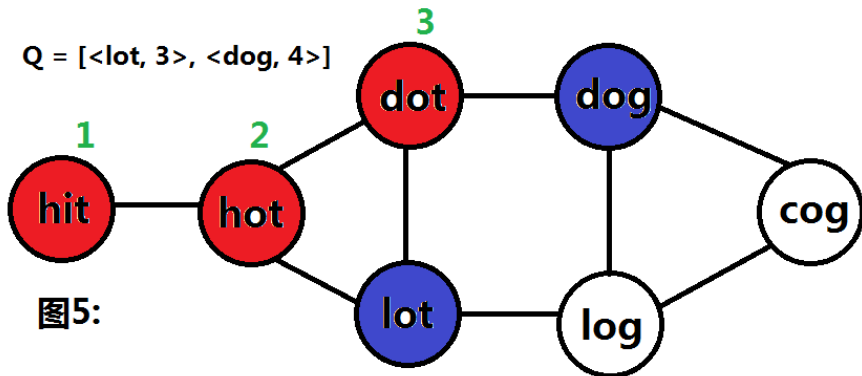




# 例3:算法设计，图的宽度遍历

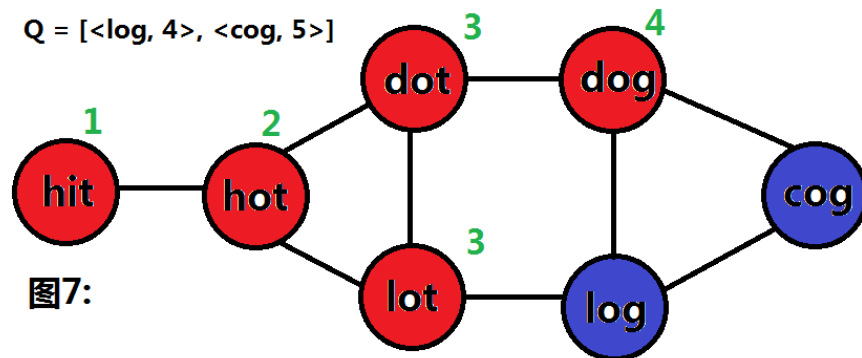
visit = {"hit", "hot", "dot", "lot", "dog"}

Q = [<lot, 3>, <dog, 4>]



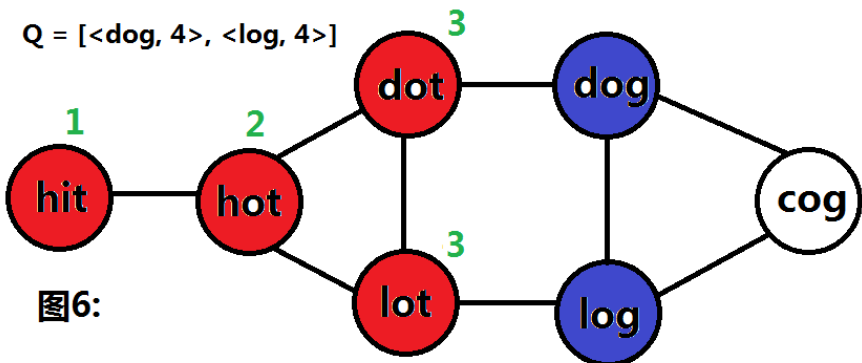
visit = {"hit", "hot", "dot", "lot", "dog", "log", "cog"}

Q = [<log, 4>, <cog, 5>]



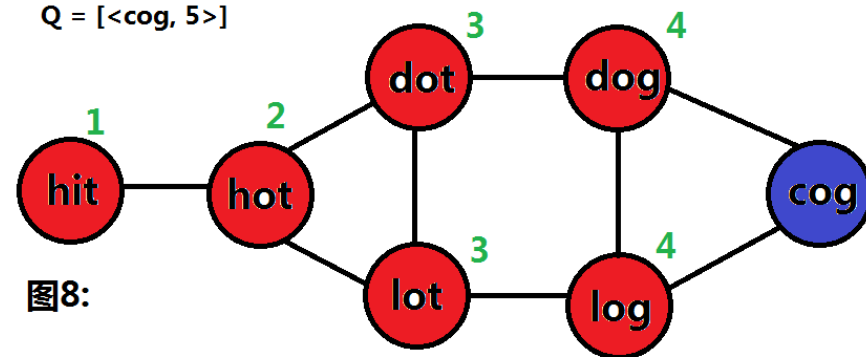
visit = {"hit", "hot", "dot", "lot", "dog", "log"}

Q = [<dog, 4>, <log, 4>]



visit = {"hit", "hot", "dot", "lot", "dog", "log", "cog"}

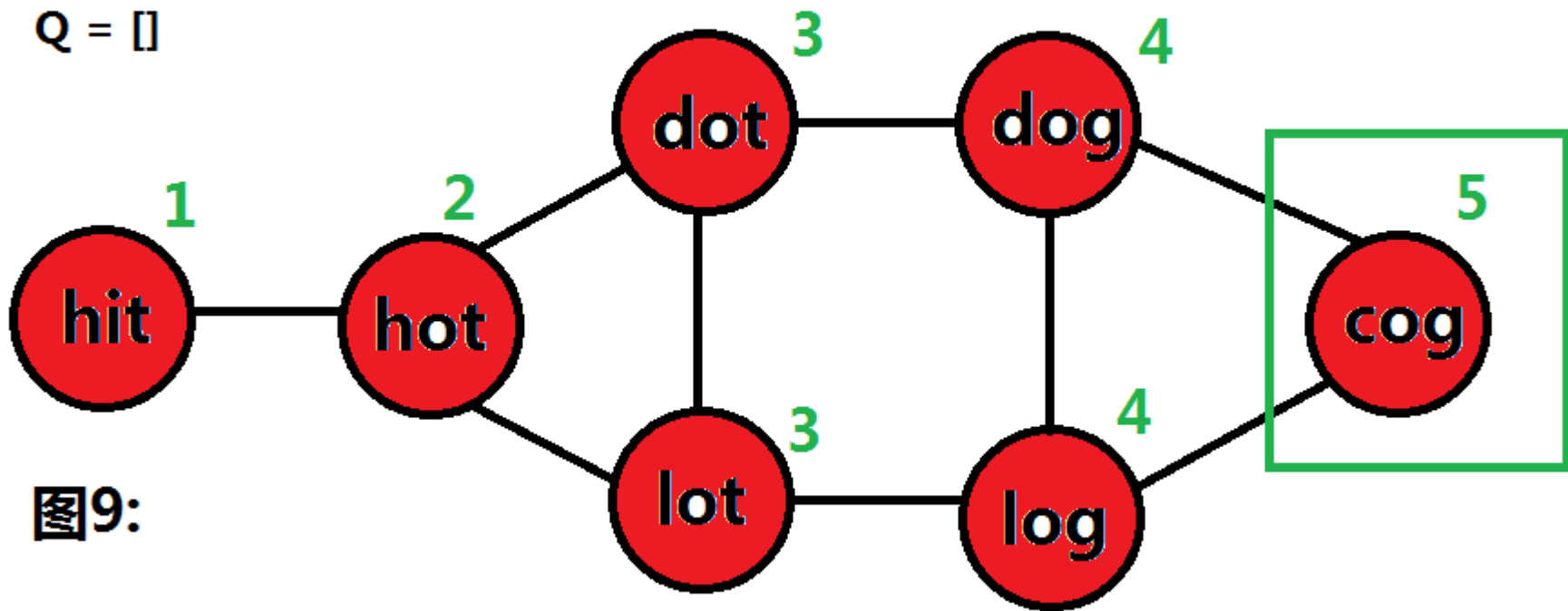
Q = [<cog, 5>]



# 例3:算法设计，图的宽度遍历

visit = {"hit", "hot", "dot", "lot", "dog", "log", "cog"}

Q = []



```

int BFS_graph(std::string &beginWord, std::string &endWord,
              std::map<std::string, std::vector<std::string> > &graph) {
    std::queue<std::pair<std::string, int> > Q; //搜索队列<顶点, 步数>
    std::set<std::string> visit; //记录已访问的顶点
    Q.push(std::make_pair(beginWord, 1)); //添加起始点, 起始点步数为1
    visit.insert(beginWord); //标记起点已访问
    while (1) {
        std::string node = Q.front().first; //取队列头部节点与步数
        int step = Q.front().second;
        2
        if (3) {
            return step;
        } //取node的全部临接点
        const std::vector<std::string> &neighbors = graph[node];
        for (int i = 0; i < neighbors.size(); i++) {
            if (visit.find(4) == visit.end()) {
                Q.push(std::make_pair(neighbors[i], 5));
                visit.insert(neighbors[i]);
            } //标记neighbors[i]已添加至队列
        }
    }
    return 0;
}

```

## 例3:课堂练习

5分钟填写代码,  
有问题随时提出!

```

int BFS_graph(std::string &beginWord, std::string &endWord,
              std::map<std::string, std::vector<std::string> > &graph) {
    std::queue<std::pair<std::string, int> > Q; //搜索队列<顶点, 步数>
    std::set<std::string> visit; //记录已访问的顶点
    Q.push(std::make_pair(beginWord, 1)); //添加起始点, 起始点步数为1
    visit.insert(beginWord); //标记起点已访问
    while (!Q.empty()) { //只要队列不空, 即不断进行搜索
        std::string node = Q.front().first; //取队列头部节点与步数
        int step = Q.front().second;
        Q.pop(); //每搜索完成一个节点, 即从队列弹出
        if (node == endWord) { //找到终点, 返回步数
            return step;
        }
        //取node的全部临接点
        const std::vector<std::string> &neighbors = graph[node];
        for (int i = 0; i < neighbors.size(); i++) { //若相邻节点还未添加至队列
            if (visit.find(neighbors[i]) == visit.end()) { //到达该节点的步数
                Q.push(std::make_pair(neighbors[i], step + 1)); //为当前步数+1
                visit.insert(neighbors[i]);
            }
            //标记neighbors[i]已添加至队列
        }
    }
    return 0;
}

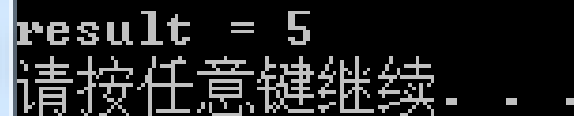
```

## 例3:实现

# 例3:测试与leetcode提交结果

```
class Solution {
public:
    int ladderLength(std::string beginWord, std::string endWord,
                    std::vector<std::string>& wordList) {
        std::map<std::string, std::vector<std::string> > graph;
        construct_graph(beginWord, wordList, graph);
        return BFS_graph(beginWord, endWord, graph);
    }
};

int main() {
    std::string beginWord = "hit";
    std::string endWord = "cog";
    std::vector<std::string> wordList;
    wordList.push_back("hot");
    wordList.push_back("dot");
    wordList.push_back("dog");
    wordList.push_back("lot");
    wordList.push_back("log");
    wordList.push_back("cog");
    Solution solve;
    int result = solve.ladderLength(beginWord, endWord, wordList);
    printf("result = %d\n", result);
    return 0;
}
```



```
result = 5
请按任意键继续. . .
```

Word Ladder

Submission Details

39 / 39 test cases passed.

Status: **Accepted**

Runtime: 515 ms

Submitted: 0 minutes ago

# 课间休息10分钟

---

## 有问题提出！

# 例4:词语阶梯2

已知两个单词(分别是起始单词与结束单词)，一个**单词词典**，根据**转换规则**计算**所有**的从起始单词到结束单词的**最短转换路径**。

转换规则如下：

1.在转换时，只能转换单词中的**1个字符**；2.转换得到的**新单词**，必须在单词词典中。

例如: beginWord = “hit”； endWord = “cog”； wordList = ["hot","dot","dog","lot","log","cog"]

**最短转换路径**:["hit","hot","dot","dog","cog"], ["hit","hot","lot","log","cog"]

```
class Solution {  
public:  
    std::vector<std::vector<std::string> > findLadders(  
        std::string beginWord, std::string endWord,  
        std::vector<std::string>& wordList) {  
    }  
};
```

1.若无法转换到endWord，返回0

2.所有的单词长度相同

4.wordList中无重复单词

3.只包含小写字符

5.beginWord与endWord非空，且不相同

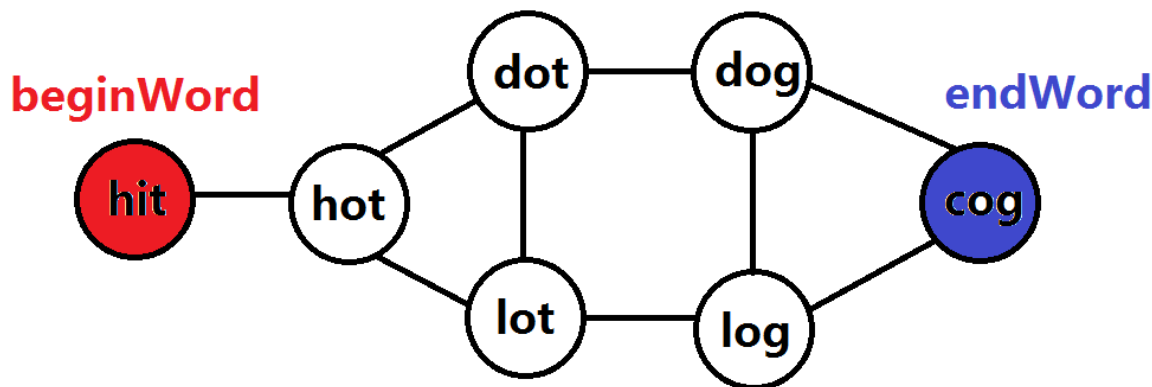
选自 **LeetCode 126. Word Ladder II**

<https://leetcode.com/problems/word-ladder-ii/description/>

难度:**Hard**

# 例4:思考

- 1.在宽度优先搜索时，如何**保存**宽度搜索时的**路径**？
- 2.如果起始点与终点间有**多条路径**，如何将多条路径**全部搜索出**？
- 3.在建立beginWord与wordList的**连接**时，若单词表中**已包含**beginWord，按照例3的方法建立图，会**出现什么问题**？



路径1: hit hot dot dog cog

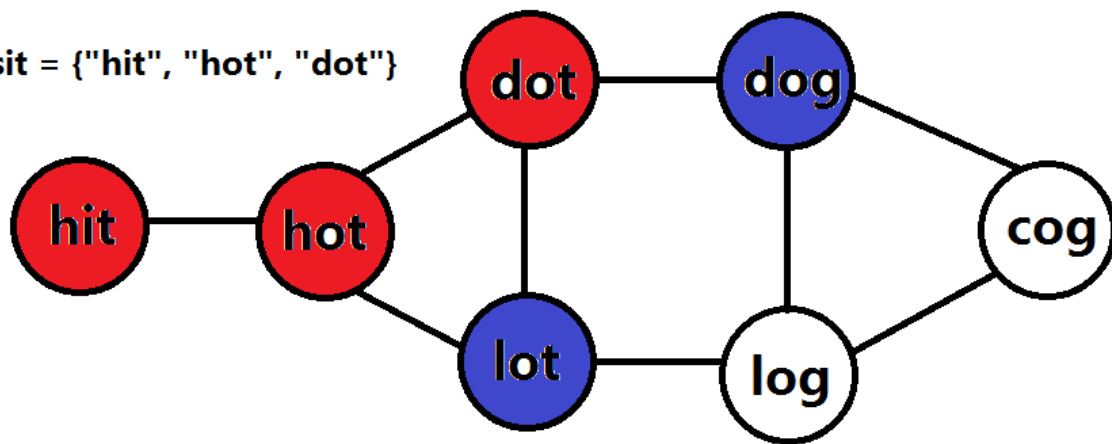
路径2: hit hot lot log cog



# 例4:算法设计，记录路径的宽搜

```
struct Qitem{  
    std::string node; //搜索节点  
    int parent_pos; //前驱节点在队列中的位置  
    int step; //到达当前节点的步数  
    Qitem(std::string node, int parent_pos, int step)  
        : node(_node), parent_pos(_parent_pos), step(_step) {  
    }  
};
```

visit = {"hit", "hot", "dot"}



Q = [<lot, 3>, <dog, 4>]

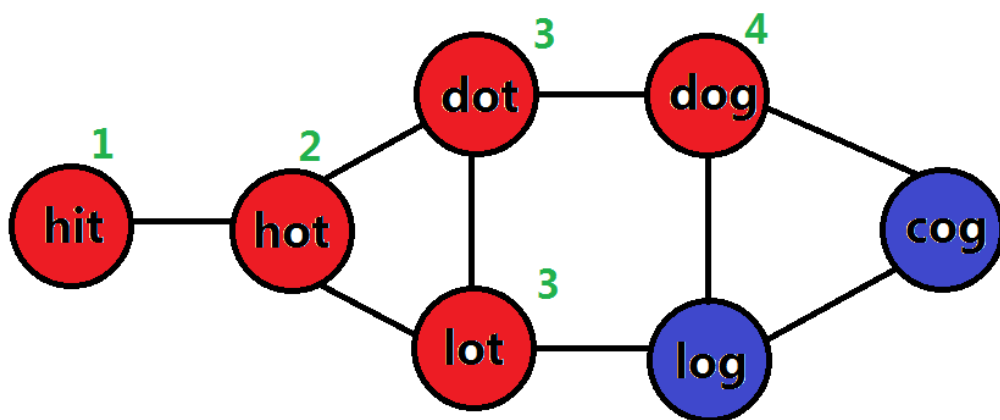
Q = [ <hit, -1, 1>, <hot, 0, 2>, <dot, 1, 3>, <lot, 1, 3>, <dog, 2, 4> ]

front

- 1.将普通队列更换为**vector实现队列**，保存**所有**的搜索节点，即在pop节点时**不会丢弃**队头元素，只是**移动front**指针。
- 2.在队列节点中增加该节点的**前驱节点**在队列中的**下标信息**，可通过该下标找到是队列中的**哪个节点**搜索到的当前节点。

# 例4:算法设计，多条路径的保存

到达某一位置可能存在**多条路径**，使用**映射**记录到达每个位置的**最短需要的步数**，新拓展到的位置只要**未曾到达**或到达步数**与最短步数相同**，即将该位置**添加**到队列中，从而存储了**从不同前驱到达该位置的情况**。



```
visit[hit] = 1;  
visit[hot] = 2;  
visit[dot] = 3;  
visit[lot] = 3;  
visit[dog] = 4;  
visit[log] = 4;  
visit[cog] = 5;
```

Q = [ <sup>0</sup><hit, -1, 1>, <sup>1</sup><hot, 0, 2>, <sup>2</sup><dot, 1, 3>, <sup>3</sup><lot, 1, 3>, <sup>4</sup><dog, 2, 4>, <sup>5</sup><log, 3, 4>, <sup>6</sup><cog, 4, 5>]

拓展  
↓  
<log, 3, 4> → <cog, 5, 5>

↑  
front

Q = [ <sup>0</sup><hit, -1, 1>, <sup>1</sup><hot, 0, 2>, <sup>2</sup><dot, 1, 3>, <sup>3</sup><lot, 1, 3>, <sup>4</sup><dog, 2, 4>, <sup>5</sup><log, 3, 4>, <sup>6</sup><cog, 4, 5>, <sup>7</sup><cog, 5, 5>]

最终得到以vector实现的**队列Q**，**终点下标6与7**，即可输出全部搜索路径。

```
void BFS_graph(std::string &beginWord, std::string &endWord,
              std::map<std::string, std::vector<std::string> > &graph,
              std::vector<Qitem> &Q, //使用vector实现的队列，可保存所有信息
              std::vector<int> &end_word_pos) { //终点endWord所在队列的位置下标
```

```
std::map<std::string, int> visit; //<单词，步数>
```

```
int min_step = 0; //到达endWord的最小步数
```

```
Q.push_back(Qitem(beginWord.c_str(), -1, 1)); //起始单词的前驱为-1
```

```
visit[beginWord] = 1; //标记起始单词步数为1
```

```
int front = 0; //队列头指针front，指向vector表示的队列头
```

```
while( 1 ) {
```

```
    const std::string &node = Q[front].node;
```

```
    int step = Q[front].step; //取队头元素
```

```
    if (min_step != 0 && 2) {
        break;
    }
```

```
    if (node == endWord) {
```

```
        3
```

```
        end_word_pos.push_back(front);
    }
```

```
    const std::vector<std::string> &neighbors = graph[node];
```

```
    for (int i = 0; i < neighbors.size(); i++) {
```

```
        if (visit.find(neighbors[i]) == visit.end() ||
```

```
            4) {
```

```
            Q.push_back(Qitem(neighbors[i], 5, step + 1));
```

```
            visit[neighbors[i]] = step + 1; //标记到达临接点neighbors[i]
```

```
        }
    }
    front++;
}
```

的最小步数

## 例4:课堂练习

5分钟填写代码，  
有问题随时提出！

# 例4:实现

```
void BFS_graph(std::string &beginWord, std::string &endWord,
               std::map<std::string, std::vector<std::string>> &graph,
               std::vector<Qitem> &Q, //使用vector实现的队列, 可保存所有信息
               std::vector<int> &end_word_pos) { //终点endWord所在队列的位置下标

    std::map<std::string, int> visit; //<单词, 步数>

    int min_step = 0; //到达endWord的最小步数

    Q.push_back(Qitem(beginWord.c_str(), -1, 1)); //起始单词的前驱为-1

    visit[beginWord] = 1; //标记起始单词步数为1

    int front = 0; //队列头指针front, 指向vector表示的队列头

    while (front != Q.size()) { //front指向Q.size()即vector尾步时, 队列为空

        const std::string &node = Q[front].node;
        int step = Q[front].step; //取队头元素

        if (min_step != 0 && step > min_step) { //step > min_step时, 代表所有到终点的路径都搜索完成
            break;
        }

        if (node == endWord) { //当搜索到结果时, 记录到达终点的最小步数
            min_step = step;
            end_word_pos.push_back(front);
        }

        const std::vector<std::string> &neighbors = graph[node];
        for (int i = 0; i < neighbors.size(); i++) {
            if (visit.find(neighbors[i]) == visit.end() || //节点没被访问, 或另一条最短路径
                visit[neighbors[i]] == step + 1) {
                Q.push_back(Qitem(neighbors[i], front, step + 1));
                visit[neighbors[i]] = step + 1; //标记到达临接点neighbors[i]的最小步数
            }
        }
        front++;
    }
}
```

# 例4:图的建立问题修改实现

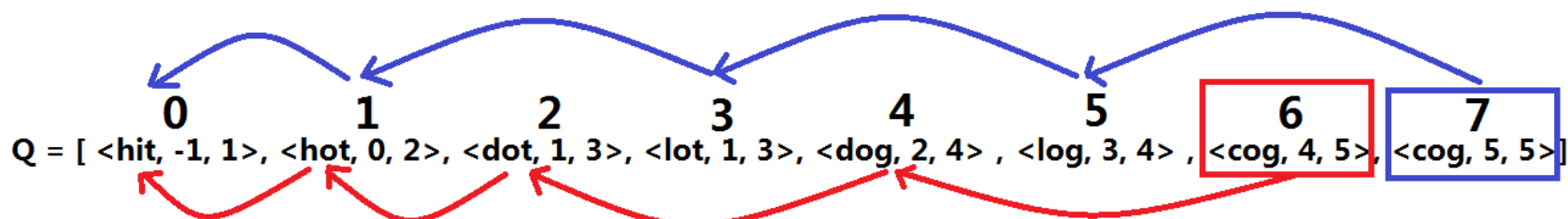
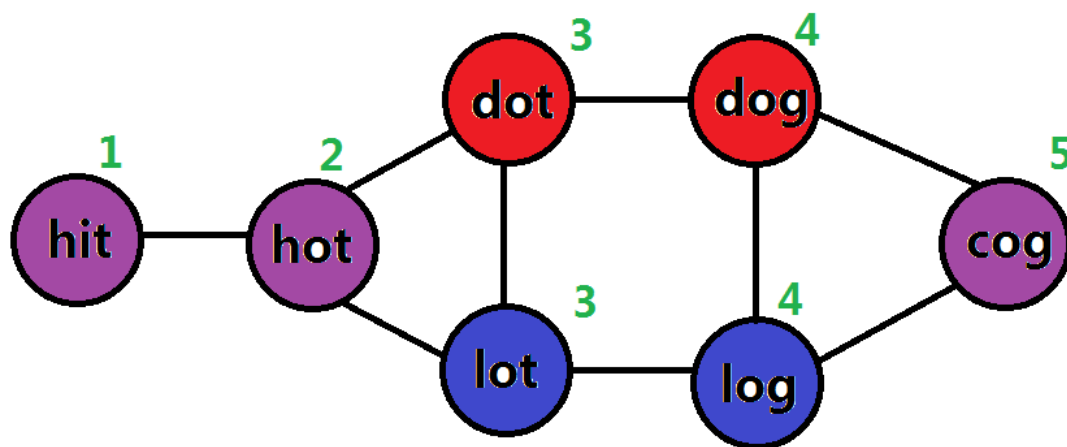
```
void construct_graph(std::string &beginWord,
                    std::vector<std::string>& wordList,
                    std::map<std::string, std::vector<std::string> > &graph) {
    int has_begin_word = 0;
    for (int i = 0; i < wordList.size(); i++) {
        if (wordList[i] == beginWord) {
            has_begin_word = 1;
        }
        graph[wordList[i]] = std::vector<std::string>();
    }
    for (int i = 0; i < wordList.size(); i++) {
        for (int j = i + 1; j < wordList.size(); j++) {
            if (connect(wordList[i], wordList[j])) {
                graph[wordList[i]].push_back(wordList[j]);
                graph[wordList[j]].push_back(wordList[i]);
            }
        }
        if (has_begin_word == 0 && connect(beginWord, wordList[i])) {
            graph[beginWord].push_back(wordList[i]);
        }
    }
}
```

//由于wordList中可能有beginWord，直接将beginWord push进入wordList，会出现重复的结果

**beginWord** → **word1** **word1**

# 例4:算法设计，遍历搜索路径

从所有结果(endWord)所在的**队列位置**(end\_word\_pos)，**向前遍历**直到起始单词(beginWord)，遍历过程中，**保存**路径上的单词。如此遍历得到的**路径**为endWord到beginWord的路径，将其按**从尾到头**的顺序存储到最终结果中即可。



路径1:  $\langle \text{cog}, 4, 5 \rangle \langle \text{dog}, 2, 4 \rangle \langle \text{dot}, 1, 3 \rangle \langle \text{hot}, 0, 2 \rangle \langle \text{hit}, -1, 1 \rangle$

路径2:  $\langle \text{cog}, 5, 5 \rangle \langle \text{log}, 3, 4 \rangle \langle \text{lot}, 1, 3 \rangle \langle \text{hot}, 0, 2 \rangle \langle \text{hit}, -1, 1 \rangle$

最短转换路径:  $[\text{"hit"}, \text{"hot"}, \text{"dot"}, \text{"dog"}, \text{"cog"}], [\text{"hit"}, \text{"hot"}, \text{"lot"}, \text{"log"}, \text{"cog"}]$

## 例4:课堂练习

```
class Solution {
public:
    std::vector<std::vector<std::string> > findLadders(
        std::string beginWord, std::string endWord,
        std::vector<std::string>& wordList) {

        std::map<std::string, std::vector<std::string> > graph;
        construct_graph(beginWord, wordList, graph);

        std::vector<Qitem> Q; //使用vector实现的队列
        std::vector<int> end_word_pos; //endWord在搜索队列的位置
        BFS_graph(beginWord, endWord, graph, Q, end_word_pos);

        std::vector<std::vector<std::string> > result; //最终结果
        for (int i = 0; i < end_word_pos.size(); i++){
            int pos = end_word_pos[i];
            std::vector<std::string> path;
            while (1) { //从endWord到beginWord将路径上的节点值push进入path
                path.push_back(Q[pos].node);
                2
            }
            result.push_back(std::vector<std::string>());
            for (int j = path.size() - 1; j >= 0; j--){
                3
            }
        }
        return result;
    }
};
```

3分钟填写代码，  
有问题随时提出！

## 例4:实现

```
class Solution {
public:
    std::vector<std::vector<std::string> > findLadders(
        std::string beginWord, std::string endWord,
        std::vector<std::string>& wordList) {

        std::map<std::string, std::vector<std::string> > graph;
        construct_graph(beginWord, wordList, graph);

        std::vector<Qitem> Q; //使用vector实现的队列
        std::vector<int> end_word_pos; //endWord在搜索队列的位置
        BFS_graph(beginWord, endWord, graph, Q, end_word_pos);
        std::vector<std::vector<std::string> > result; //最终结果

        for (int i = 0; i < end_word_pos.size(); i++){
            int pos = end_word_pos[i];
            std::vector<std::string> path;
            while (pos != -1) { //从endWord到beginWord将路径上的节点值push进入path
                path.push_back(Q[pos].node);
                pos = Q[pos].parent_pos;
            }
            result.push_back(std::vector<std::string>());
            for (int j = path.size() - 1; j >= 0; j--){
                result[i].push_back(path[j]);
            }
        }
        return result;
    }
};
```



# 例4:测试与leetcode提交结果

```
int main() {
    std::string beginWord = "hit";
    std::string endWord = "cog";
    std::vector<std::string> wordList;
    wordList.push_back("hot");
    wordList.push_back("dot");
    wordList.push_back("dog");
    wordList.push_back("lot");
    wordList.push_back("log");
    wordList.push_back("cog");
    Solution solve;
    std::vector<std::vector<std::string> > result
        = solve.findLadders(beginWord, endWord, wordList);
    for (int i = 0; i < result.size(); i++) {
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%s] ", result[i][j].c_str());
        }
        printf("\n");
    }
    return 0;
}
```

Word Ladder II

Submission Details

39 / 39 test cases passed.

Status: **Accepted**

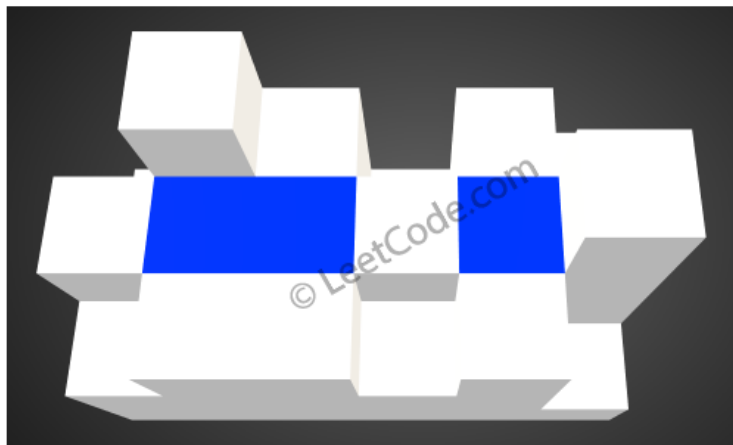
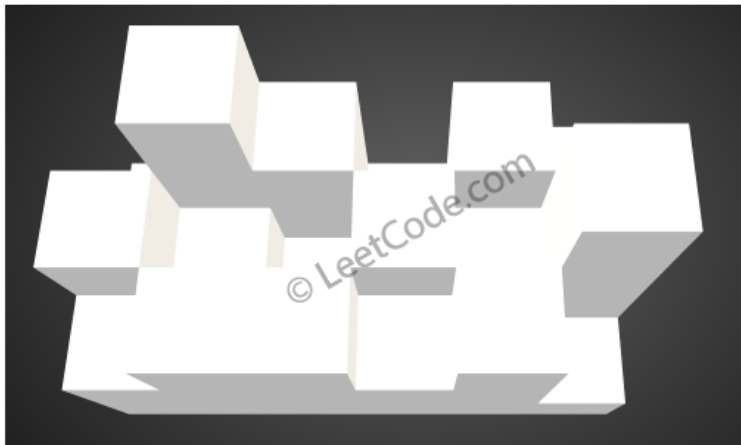
Runtime: 638 ms

Submitted: 0 minutes ago

```
[hit] [hot] [dot] [dog] [cog]
[hit] [hot] [lot] [log] [cog]
请按任意键继续. . .
```

# 例5:收集雨水2

已知一个 $m*n$ 的**二维数组**，数组存储正整数，代表一个个单元的高度(立方体)，将这些立方体想象成**水槽**，问如果下雨这些立方体中会有多少**积水**。



[1,4,3,1,3,2],  
[3,2,1,3,2,4],  
[2,3,3,2,3,1]

↓  
4

```
class Solution {  
public:  
    int trapRainWater(std::vector<std::vector<int>> & heightMap) {  
    }  
};
```

选自 **LeetCode 407. Trapping Rain Water II**

<https://leetcode.com/problems/trapping-rain-water-ii/description/>

难度:**Hard**

# 例5:思考

1.观察如下立方体，会有**多少**积水？

2, 3, 4

9, 1, 5

8, 7, 6



积水？

2, 3, 4, 1

2, 1, 1, 5

9, 6, 7, 8



积水？

2.能积水的底面有什么**特点**，积水的**多少**和什么直接相关？

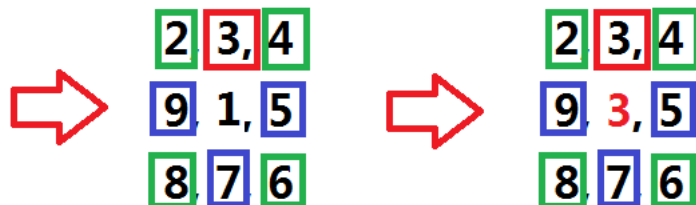
3.找出四周(边界)上与**中间底面**相连**最低**的点，设高度为 $h$ ，用 $h$ 与**中间底面**的高度做**差**，这些差的和即为最终结果？

# 例5:分析

1.能积水的底面一定**不在四周**，积水多少与周围**最矮的**立方体相关。

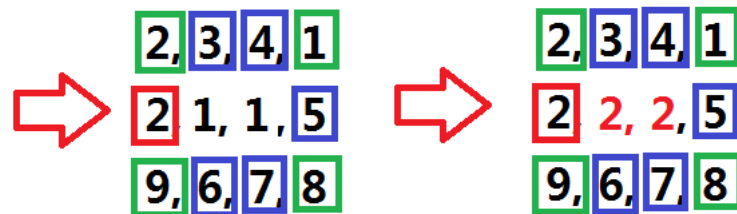
2.**围住**中间积水的边界位置**不一定**在四周，所以"找出四周(边界)上最低的点"求直接差的点，**不可行**。

2, 3, 4  
9, 1, 5  
8, 7, 6



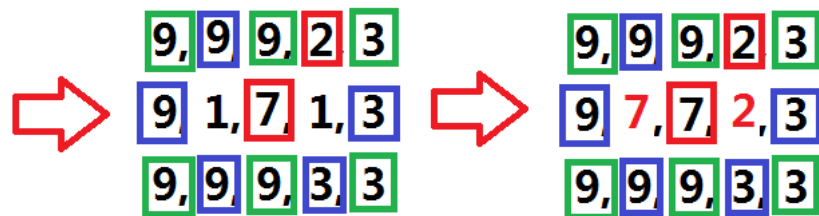
积水量为  $3 - 1 = 2$

2, 3, 4, 1  
2, 1, 1, 5  
9, 6, 7, 8



积水量为  $2 - 1 + 2 - 1 = 2$

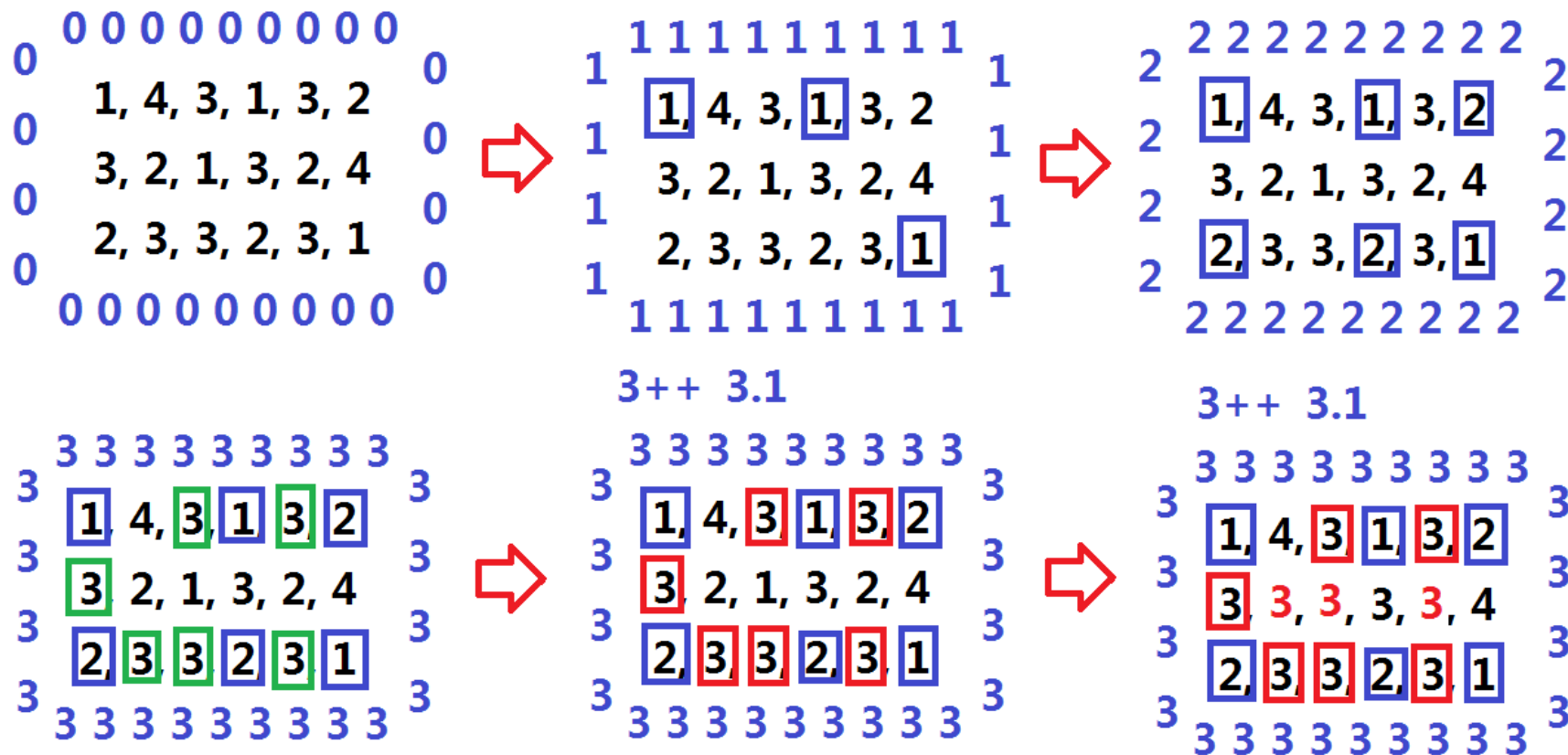
9, 9, 9, 2, 3  
9, 1, 7, 1, 3  
9, 9, 9, 3, 3



积水量为  $7 - 1 + 2 - 1 = 7$

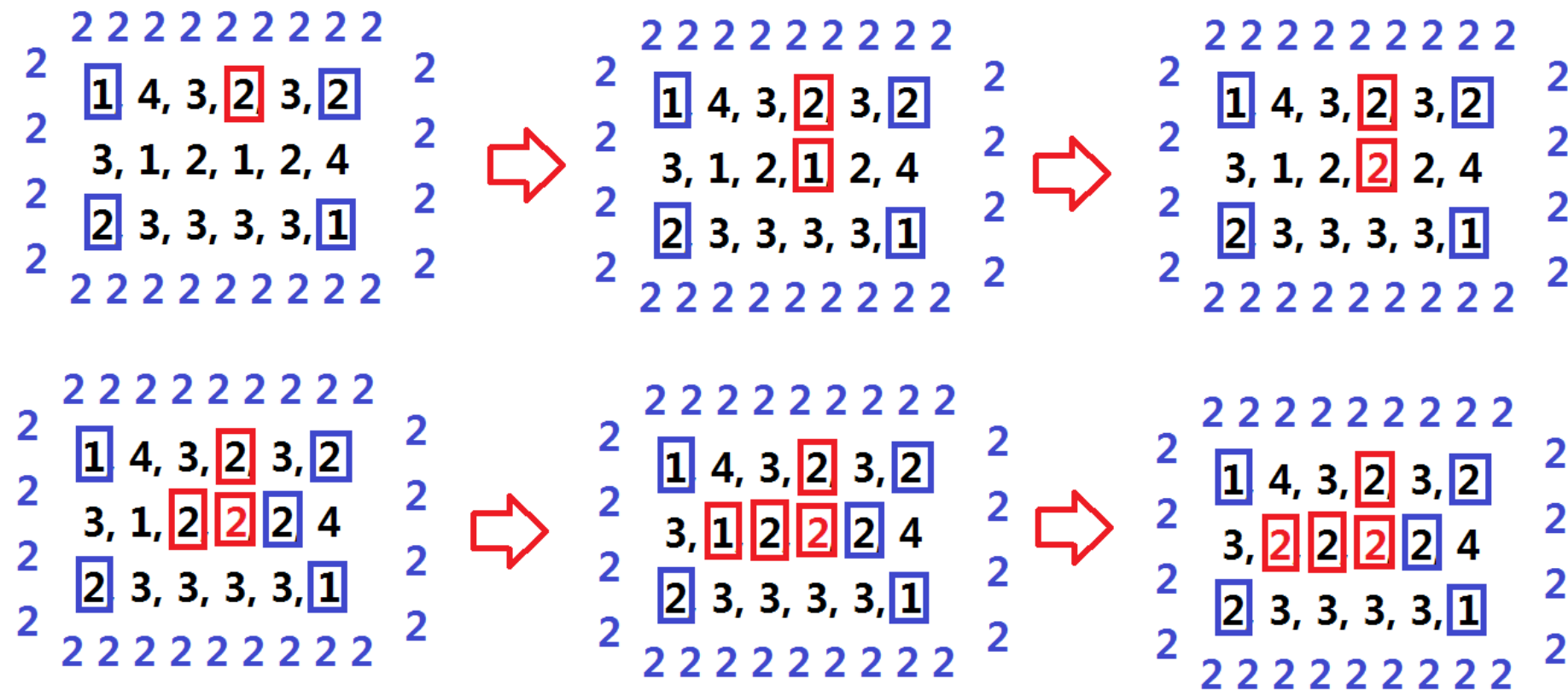
# 例5:分析2

想像水流在最外层，慢慢上升，过程如下:



积水量为  $3 - 2 + 3 - 1 + 3 - 2 = 4$

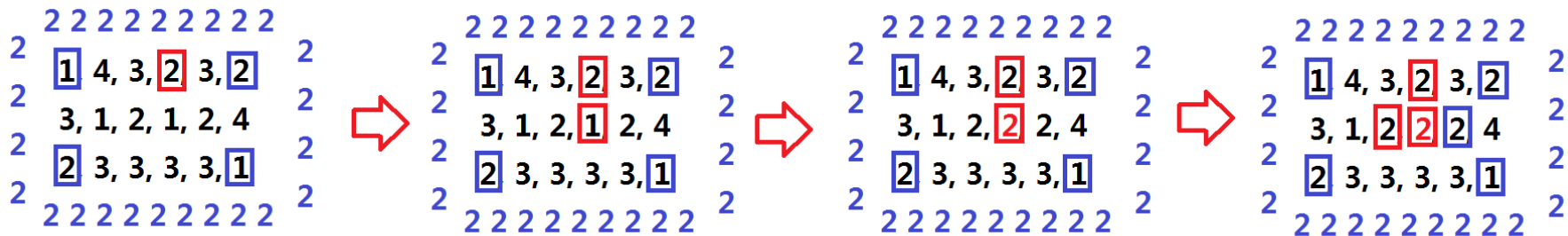
# 例5:分析3



积水量为  $2 - 1 + 2 - 1 = 2$

# 例5:算法设计

1. 搜索队列使用**优先级队列(堆)**，**越低矮**的点**优先级越高**(最小堆)，**越优先**进行搜索。
2. 以矩形**四周的点**作为**起始点**进行广度优先搜索(这些点要**最初**push进入队列)。
3. 使用一个二维数组对push进入队列的点进行**标记**，之后搜索到该点后，**不再**push到队列中。
4. 只要优先级队列**不空**，即取出优先级队列队头元素进行搜索，按照**上下左右**四个方向进行拓展，拓展过程中**忽略超出边界与已入队列**的点。
5. 当对某点(x, y, h)进行**拓展**时(h即为(x, y)位置的高度，heightMap[x][y]):  
得到的**新点**为(newx, newy)，**高度**为heightMap[newx][newy]，  
**如果** h 大于 heightMap[newx][newy]:  
    最终结果 += h - heightMap[newx][newy];  
    将heightMap[newx][newy]**赋值**为h(即升高该位置的水面)。  
将(newx, newy, heightMap[newx][newy]) **push进入** 优先级队列，并做标记。



# 例5:算法设计

初始化:

1, 4, 3, 1, 3, 2	0, 0, 0, 0, 0, 0
3, 2, 1, 3, 2, 4	0, 0, 0, 0, 0, 0
2, 3, 3, 2, 3, 1	0, 0, 0, 0, 0, 0

将四周的点添加至优先级队列Q,并做标记

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

优先级队列中存储<行, 列, 高>

$Q = [<0, 0, 1>, <0, 4, 1>, <2, 5, 1>, <0, 5, 2>, \dots, <1, 5, 4>]$

1.搜索:  $<0, 0, 1>$

蓝色:正在搜索

1	4	3	1	3	2
3	2	1	3	2	4
2	3	3	2	3	1

绿色:队列中

紫色:已完成

红色:拓展的新位置

2.搜索:  $<0, 4, 1>$  并拓展  $<1, 3, 3>$  push进入队列

1	4	3	1	3	2
3	2	1	3	2	4
2	3	3	2	3	1

1, 1, 1, 1, 1, 1

1, 0, 0, 1, 0, 1

1, 1, 1, 1, 1, 1

3.搜索:  $<2, 5, 1>$ ,  $<0, 5, 2>$ ,  $<2, 0, 2>$ ,  $<2, 3, 2>$

1	4	3	1	3	2
3	2	1	3	2	4
2	3	3	2	3	1



# 例5:算法设计

4.搜索:  $\langle 0, 2, 3 \rangle$  并拓展  $\langle 1, 2, 1 \rangle$

1	4	3	1	3	2	1, 1, 1, 1, 1, 1
3	2	1	3	2	4	1, 0, 1, 1, 0, 1
2	3	3	2	3	1	1, 1, 1, 1, 1, 1

由于 $\langle 0, 2, 3 \rangle$ 的h 高于  $\langle 1, 2, 1 \rangle$ 的h

积水量  $+= 3 - 1$ , 积水量 = 2

push  $\langle 1, 2, 3 \rangle$

1	4	3	1	3	2
3	2	3	3	2	4
2	3	3	2	3	1

5.无论搜索 $\langle 1, 0, 3 \rangle$ 、 $\langle 2, 1, 3 \rangle$ 、 $\langle 1, 2, 3 \rangle$ 的哪一个 均可拓展到 $\langle 1, 1, 2 \rangle$

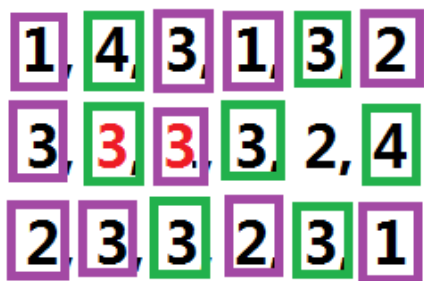
1	4	3	1	3	2	1, 1, 1, 1, 1, 1
3	2	3	3	2	4	1, 1, 1, 1, 0, 1
2	3	3	2	3	1	1, 1, 1, 1, 1, 1

由于 蓝色位置 的h 高于  $\langle 1, 1, 2 \rangle$  的h

积水量  $+= 3 - 2$ , 积水量 = 3

1	4	3	1	3	2	push $\langle 1, 1, 3 \rangle$
3	3	3	3	2	4	
2	3	3	2	3	1	

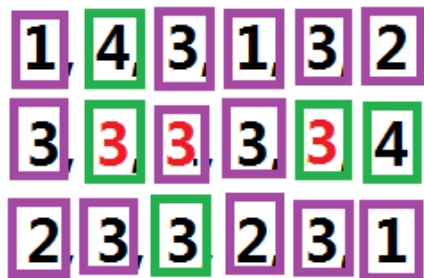
# 例5:算法设计



6. 无论搜索  $\langle 0, 4, 3 \rangle$ 、 $\langle 1, 3, 3 \rangle$ 、 $\langle 2, 4, 3 \rangle$  的哪一个 均可拓展到  $\langle 1, 4, 2 \rangle$



由于 蓝色位置 的  $h$  高于  $\langle 1, 4, 2 \rangle$  的  $h$   
积水量  $+= 3 - 2$ , 积水量 = 4



剩余的绿色点不再会拓展新的点，  
故最终积水量 为4

# 例5:预备知识:结构体的STL优先级队列

```
#include <stdio.h>
#include <queue>
struct Qitem{
    int x;
    int y;
    int h;
    Qitem(int _x, int _y, int _h) :
        x(_x), y(_y), h(_h){
    }
};
struct cmp{
    bool operator()(const Qitem &a, const Qitem &b){
        return a.h > b.h;
    }
};
int main(){
    std::priority_queue<Qitem, std::vector<Qitem>, cmp> Q;
    Q.push(Qitem(0, 0, 5));
    Q.push(Qitem(1, 3, 2));
    Q.push(Qitem(5, 2, 4));
    Q.push(Qitem(0, 1, 8));
    Q.push(Qitem(6, 7, 1));
    while (!Q.empty()){
        int x = Q.top().x;
        int y = Q.top().y;
        int h = Q.top().h;
        printf("x = %d y = %d h = %d\n", x, y, h);
        Q.pop();
    }
    return 0;
}
```

```
x = 6 y = 7 h = 1
x = 1 y = 3 h = 2
x = 5 y = 2 h = 4
x = 0 y = 0 h = 5
x = 0 y = 1 h = 8
请按任意键继续. . .
```

# 例5:初始化代码实现

```
class Solution {
public:
    int trapRainWater(std::vector<std::vector<int>> & heightMap) {
        std::priority_queue<Qitem, std::vector<Qitem>, cmp> Q;
        if (heightMap.size() < 3 || heightMap[0].size() < 3){
            return 0;
        }
        //行数或列数小于3，必然无法积水
        int row = heightMap.size();
        int column = heightMap[0].size();
        std::vector<std::vector<int>> mark;
        for (int i = 0; i < row; i++){
            mark.push_back(std::vector<int> ());
            for (int j = 0; j < column; j++){
                mark[i].push_back(0);
            }
        }
        for (int i = 0; i < row; i++){
            Q.push(Qitem(i, 0, heightMap[i][0]));
            mark[i][0] = 1;
            Q.push(Qitem(i, column - 1, heightMap[i][column - 1]));
            mark[i][column - 1] = 1;
        }
        for (int i = 1; i < column - 1; i++){
            Q.push(Qitem(0, i, heightMap[0][i]));
            mark[0][i] = 1;
            Q.push(Qitem(row - 1, i, heightMap[row - 1][i]));
            mark[row - 1][i] = 1;
        }
    }
};
```

初始化:

mark:

1, 4, 3, 1, 3, 2	0, 0, 0, 0, 0, 0
3, 2, 1, 3, 2, 4	0, 0, 0, 0, 0, 0
2, 3, 3, 2, 3, 1	0, 0, 0, 0, 0, 0

将四周的点添加至优先级队列Q,并做标记

	0	1	2	3	4	5	mark:
0	1, 4, 3, 1, 3, 2	1, 1, 1, 1, 1, 1					
1	3, 2, 1, 3, 2, 4	1, 0, 0, 0, 0, 1					
2	2, 3, 3, 2, 3, 1	1, 1, 1, 1, 1, 1					

# 例5:课堂练习

```
static const int dx[] = {-1, 1, 0, 0}; //方向数组
static const int dy[] = {0, 0, -1, 1};
int result = 0; //最终积水量
```

```
while(!Q.empty()){
```

```
    int x = Q.top().x; //取队列头部信息
    int y = Q.top().y;
    int h = Q.top().h;
```

1

```
    for (int i = 0; i < 4; i++){
        int newx = x + dx[i]; //拓展4个方向
        int newy = y + dy[i];
```

```
        if(newx < 0 || newx >= row ||
            newy < 0 || newy >= column || mark[newx][newy]){
```

2

//当新拓展的点超出边界或已加入队列

```
        }
        if (h > heightMap[newx][newy]){ //当前点的高度高于拓展点时
```

```
            result +=
```

3

4

```
        }
        Q.push(Qitem(newx, newy, heightMap[newx][newy]));
```

5

```
    }
    return result;
```

5分钟填写代码，  
有问题随时提出！



# 例5:实现

```
static const int dx[] = {-1, 1, 0, 0}; //方向数组
static const int dy[] = {0, 0, -1, 1};
```

```
int result = 0; //最终积水量
```

```
while(!Q.empty()){
```

```
    int x = Q.top().x; //取队列头部信息
```

```
    int y = Q.top().y;
```

```
    int h = Q.top().h;
```

```
    Q.pop();
```

```
    for (int i = 0; i < 4; i++){
```

```
        int newx = x + dx[i];
```

```
        int newy = y + dy[i];
```

```
        if (newx < 0 || newx >= row ||
            newy < 0 || newy >= column || mark[newx][newy]){
```

```
            continue;
```

```
        }
```

```
        if (h > heightMap[newx][newy]){
```

```
            result += h - heightMap[newx][newy];
```

```
            heightMap[newx][newy] = h;
```

```
        }
```

```
        Q.push(Qitem(newx, newy, heightMap[newx][newy]));
```

```
        mark[newx][newy] = 1;
```

```
    }
```

```
    return result;
```

```
}
```

```
};
```

搜索: <0, 2, 3> 并拓展 <1, 2, 1>

1	4	3	1	3	2
3	2	1	3	2	4
2	3	3	2	3	1

1, 1, 1, 1, 1, 1

1, 0, 1, 1, 0, 1

1, 1, 1, 1, 1, 1

由于<0, 2, 3>的h

高于 <1, 2, 1>的h

积水量 += 3 - 1, 积水量 = 2

push < 1, 2, 3>

1	4	3	1	3	2
3	2	3	3	2	4
2	3	3	2	3	1

# 例5:测试与leetcode提交结果

## Trapping Rain Water II

### Submission Details

```
int main() {
    int test[][10] = {
        {1, 4, 3, 1, 3, 2},
        {3, 2, 1, 3, 2, 4},
        {2, 3, 3, 2, 3, 1}
    };
    std::vector<std::vector<int> > heightMap;
    for (int i = 0; i < 3; i++) {
        heightMap.push_back(std::vector<int> ());
        for (int j = 0; j < 6; j++) {
            heightMap[i].push_back(test[i][j]);
        }
    }
    Solution solve;
    printf("%d\n", solve.trapRainWater(heightMap));
    return 0;
}
```

40 / 40 test cases passed.

Status: **Accepted**

Runtime: 19 ms

Submitted: 1 minute ago

4

请按任意键继续. . .

# 结束

---

非常感谢大家！

林沐