



# 数据结构和算法 (Python描述)

郭 炜

微信公众号



微博: <http://weibo.com/guoweiofpku>

**学会程序和算法，走遍天下都不怕!**

讲义照片均为郭炜拍摄



北京大学  
PEKING UNIVERSITY

信息科学技术学院

# 图的遍历和搜索



北京大学  
PEKING UNIVERSITY

信息科学技术学院

## 图的概念



黄山

# 图的概念

图由顶点集合和边集合组成

每条边连接两个不同顶点

有向图：边有方向(有起点和终点)

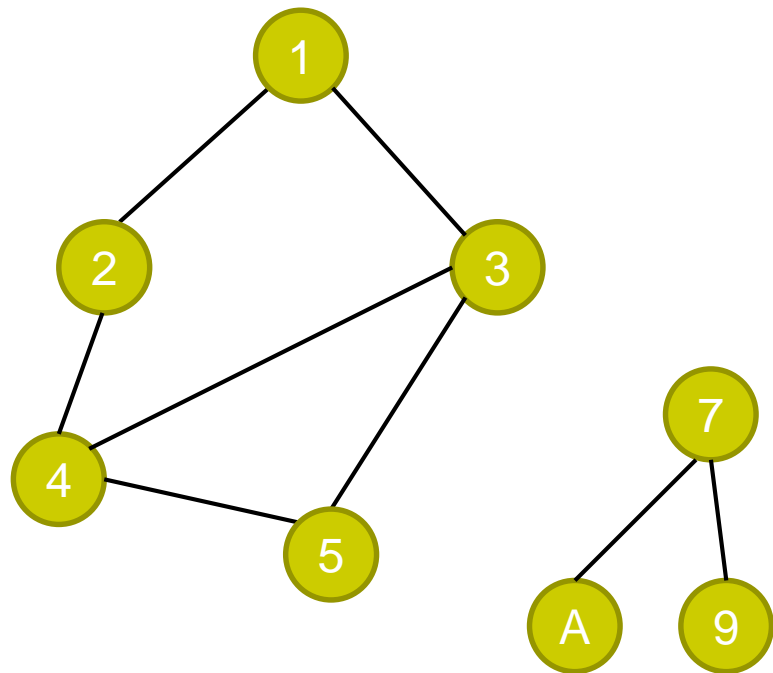
无向图：边没有方向

边只是逻辑上表示两个顶点有直接关系，边是直的还是弯的，边有没有交叉，都没有意义。

无向图两个顶点之间最多一条边

有向图两个顶点之间最多两条方向不同的边

无向图



# 图的概念

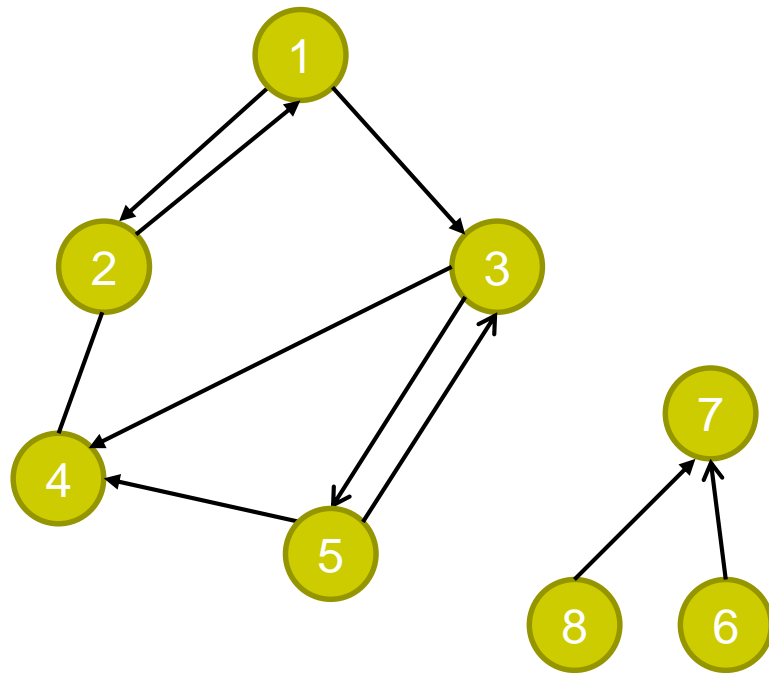
无向图连接顶点 $u, v$ 的边, 记为 $(u, v)$

有向图连接顶点 $u, v$ 的边, 记为 $\langle u, v \rangle$

无向图中边 $(u, v)$ 存在, 称 $u, v$ 相邻,  $u, v$ 互为邻点

有向图中边 $\langle u, v \rangle$ 存在, 称 $v$ 是 $u$ 的邻点

有向图



# 图的相关概念

- 1) 顶点的度数：和顶点相连的边的数目。
- 2) 顶点的出度：有向图中，以该顶点作为起点的边的数目
- 3) 顶点的入度：有向图中，以该顶点作为终点的边的数目
- 4) 顶点的出边：有向图中，以该顶点为起点的边
- 5) 顶点的入边：有向图中，以该顶点为终点的边
- 6) 路径：对于无向图，如果存在顶点序列 $V_{i0}, V_{i1}, V_{i2}, \dots, V_{im}$ ，使得 $(V_{i0}, V_{i1}), (V_{i1}, V_{i2}), \dots, (V_{im-1}, V_{im})$ 都存在，则称 $(V_{i0}, V_{i1}, \dots, V_{im})$ 是从 $V_{i0}$ 到 $V_{im}$ 的一条路径。（对于有向图，把 $()$ 换成 $<>$ ）
- 5) 路径的长度：路径上的边的数目
- 6) 回路（环）：起点和终点相同的路径
- 7) 简单路径：除了起点和终点可能相同外，其它顶点都不相同的路径

# 图的相关概念

## 8) 完全图:

完全无向图: 任意两个顶点都有边相连

完全有向图: 任意两个顶点都有两条方向相反的边

9) 连通: 如果存在从顶点 $u$ 到顶点 $v$ 的路径, 则称 $u$ 到 $v$ 连通, 或 $u$ 可达 $v$ 。无向图中,  $u$ 可达 $v$ , 必然 $v$ 可达 $u$ 。有向图中,  $u$ 可达 $v$ , 并不能说明 $v$ 可达 $u$ 。

10) 连通无向图: 图中任意两个顶点 $u$ 和 $v$ 互相可达。

11) 强连通有向图: 图中任意两个顶点 $u$ 和 $v$ 互相可达。

12) 子图: 从图中抽取部分或全部边和点构成的图

13) 连通分量 (极大连通子图): 无向图的一个子图, 是连通的, 且再添加任何一些原图中的顶点和边, 新子图都不再连通。

## 图的相关概念

- 14) 强连通分量：有向图的一个子图，是强连通的，且再添加任何一些原图中的顶点和边，新子图都不再强连通。
- 15) 带权图：边被赋予一个权值的图
- 16) 网络：带权无向连通图



# 图的性质

1. 图的边数等于顶点度数之和的一半
2.  $n$ 个顶点的连通图至少有 $n-1$ 条边
3.  $n$ 个顶点的，无回路的连通图就是一棵树，有 $n-1$ 条边



北京大学  
PEKING UNIVERSITY

信息科学技术学院

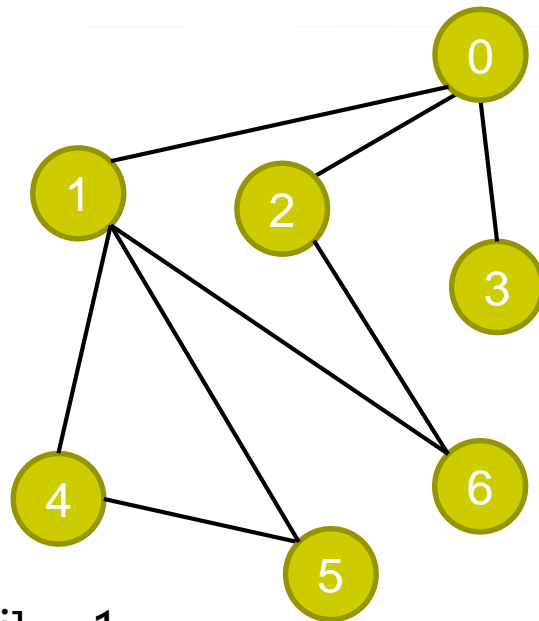
## 图的表示方法



云岗石窟

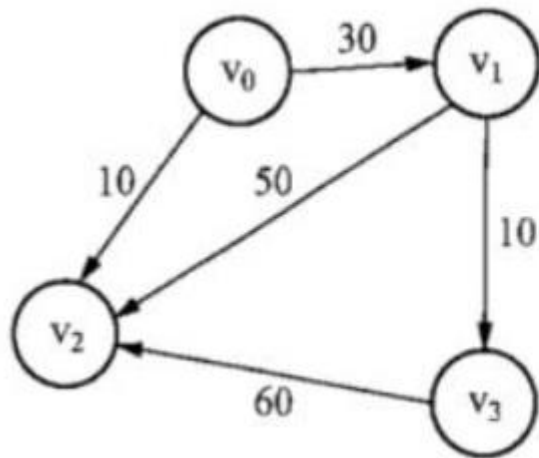
# 用邻接矩阵表示无向图

	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	0	0	1	1	1
2	1	0	0	0	0	0	1
3	1	0	0	0	0	0	0
4	0	1	0	0	0	1	0
5	0	1	0	0	1	0	0
6	0	1	1	0	0	0	0



- $G[i][j] = 1 \Leftrightarrow$  点 $i$ 和点 $j$ 之间有边  $\Leftrightarrow G[j][i] = 1$
- $G[i][j] = 0 \Leftrightarrow$  点 $i$ 和点 $j$ 之间无边  $\Leftrightarrow G[j][i] = 0$ 或无穷大
- 矩阵是对称的。对角线元素根据需要设置成0或无穷大或其它值
- 如果是带权图, 若 $(i,j)$ 存在, 可以将 $G[i][j]$ 和 $G[j][i]$ 设置为权值,  $(i,j)$ 不存在, 则 $G[i][j]$ 和 $G[j][i]$ 设为无穷大或0或其它需要的值

# 用邻接矩阵表示有向图



$\infty$	30	10	$\infty$
$\infty$	$\infty$	50	10
$\infty$	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	60	$\infty$

- $G[i][j] = 1$  或权值  $\Leftrightarrow$  点  $i$  和点  $j$  之间有边
- $G[i][j] = 0$  或无穷大  $\Leftrightarrow$  点  $i$  和点  $j$  之间无边
- 矩阵未必是对称的。对角线元素根据需要设置成 0 或无穷大或其它值

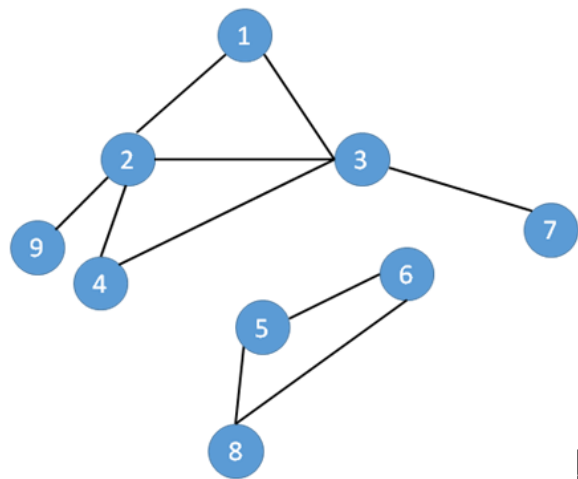
# 邻接矩阵表示法的适用场景

- $n$ 个顶点的图，需要一个有 $n^2$ 个元素的矩阵,比较费空间
- 查找和一个顶点的邻点，需要 $O(n)$ 时间
- 对于边的数目只有 $O(n)$ 量级的稀疏图，邻接矩阵既浪费空间也浪费时间
- 适用于边的数目达到 $O(n^2)$ 量级的稠密图

# 用邻接表表示图

每个顶点V对应一个列表。对无向图，列表里存放所有和V相连的边；对有向图，列表里存放所有V的出边。边的信息包括邻点，边权值等。**对稀疏图特别适用。**

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







北京大学  
PEKING UNIVERSITY

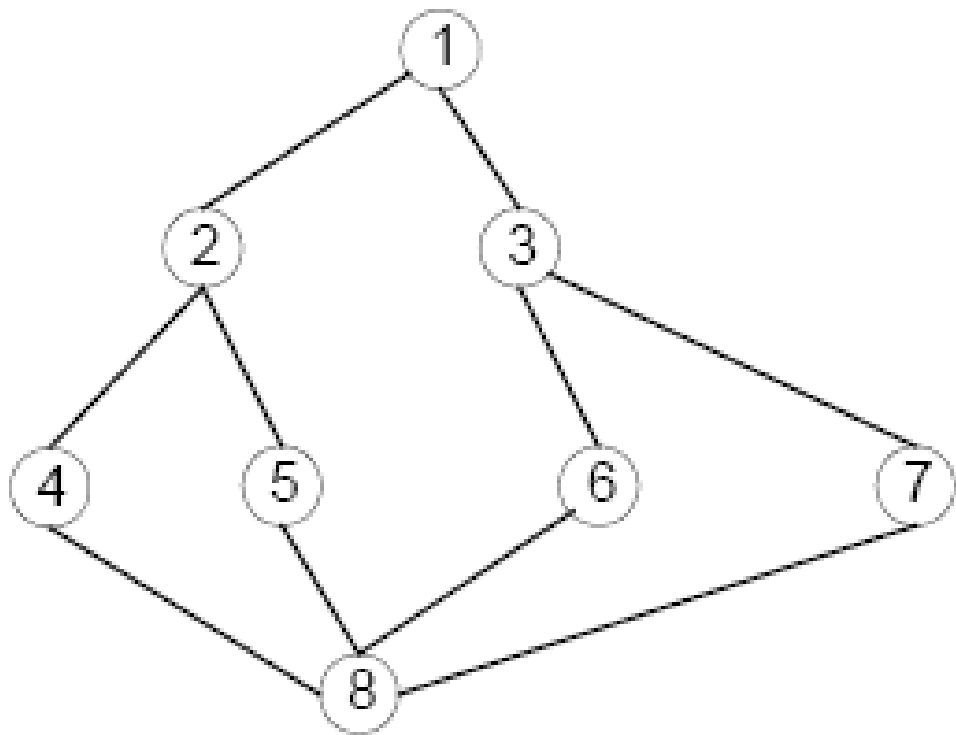
信息科学技术学院

## 图的遍历



云岗石窟

# 深搜 vs. 广搜



若要遍历所有顶点：

□ 深搜（能往前走就往前走）

1-2-4-8-5-6-3-7

□ 广搜（按距离起点的距离，  
即最小边数从小到大遍历）

1-2-3-4-5-6-7-8



# 深度优先搜索 (Depth-First-Search)

从起点出发，走过的点要做标记，发现有没走过的点，就随意挑一个往前走，走不了就回退，此种路径搜索策略就称为“深度优先搜索”，简称“深搜”。

其实称为“远度优先搜索”更容易理解些。因为这种策略能往前走一步就往前走一步，总是试图走得更远。所谓远近(或深度)，就是以距离起点的步数来衡量的。

# 图的深度优先遍历

```
def Dfs (V) :
```

```
    将v标记为旧点
```

```
    对和v相邻的每个点 U:
```

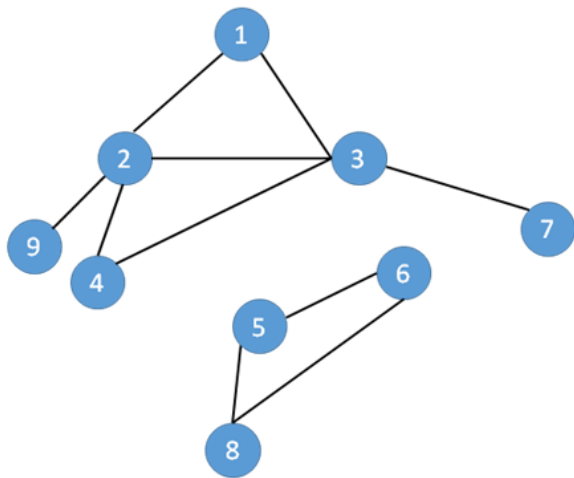
```
        Dfs (U)
```

```
def main() :
```

```
    将所有点都标记为新点
```

```
    while (在图中能找到新点k)
```

```
        Dfs (k)
```



## 图的深度优先遍历 —— 邻接表形式

```
def dfsTravel(G, op): #G是邻接表
    def dfs(v):
        visited[v] = True
        op(v)
        for u in G[v]:
            if not visited[u]:
                dfs(u)
    n = len(G) # 顶点数目
    visited = [False for i in range(n)]
    for i in range(n): # 顶点编号0到n-1
        if not visited[i]:
            dfs(i)
```

- 每个顶点看过一遍，每条边看过一遍（无向图两遍） 复杂度 $O(E+V)$   $E$ 是边数， $V$ 是顶点数

## 图的深度优先遍历 —— 邻接矩阵形式

```
def dfsTravel2(G,op): #G是邻接矩阵
    def dfs(v): #从顶点v开始进行深度优先遍历
        visited[v] = True
        op(v)
        for i in range(n):
            if G[v][i] and not visited[i]:
                dfs(i)
    n = len(G) # 顶点数目
    visited = [False for i in range(n)]
    for i in range(n): # 顶点编号0到n-1
        if not visited[i]:
            dfs(i)
```

➤ 对每个顶点 $v$ 要做一次 $\text{dfs}(v)$ ，做一次 $\text{dfs}(v)$ 时要 $O(V)$ 时间查看它和另外所有顶点的关系，因此复杂度 $O(V^2)$   $V$ 是顶点数

# 图的深度优先遍历 —— 邻接表形式

## ➤ 非递归写法

```
def dfsTravel3(G,op): #顶点编号从0开始, G是邻接表
```

```
    n = len(G)    # 顶点数目
```

```
    visited = [False for i in range(n)]
```

```
    for x in range(n):
```

```
        if not visited[x]:
```

```
            stack = [[x,0]] #0表示只看了0个邻点
```

```
            visited[x] = True
```

```
            while len(stack) > 0:
```

```
                nd = stack[-1] #nd[1]表示已经看过nd[1]个邻点
```

```
                v = nd[0]
```

```
                if nd[1] == 0:
```

```
                    op(v)
```

```
                if nd[1] == len(G[v]): #最后一个邻点已经看过
```

```
                    stack.pop()
```

➤ 每个顶点出栈一次。对栈顶顶点，查看它的所有边，因此复杂度 $O(V+E)$   $V$ 是顶点数

# 图的深度优先遍历 —— 邻接表形式

## ➤ 非递归写法

```
else: #对应if nd[1] == len(G[v]):  
    for i in range(nd[1], len(G[v])):  
        u = G[v][i]  
        nd[1] += 1 #看过的邻点多了一个  
        if not visited[u]:  
            stack.append([u, 0])  
            visited[u] = True  
            break
```

# 图的广度优先遍历

- 1) 选一个没有访问过的顶点入队列,并标记其为访问过。如果找不到, 遍历结束
- 2) 若队列不为空, 取出队头顶点x, goto 3)。若队列为空, goto 1)
- 3) 找出x的所有未访问过的邻点, 将它们标记为访问过, 并入队列
- 4) goto 2)

因为图可能不连通（有向图），或不是强连通（无向图），因此队列为空时，可能还有顶点未曾访问过

## 图的广度优先遍历 —— 邻接表形式

```
def bfsTravel(G, op): #G是邻接表形式的图, op是访问操作
```

```
    import collections
```

```
    n = len(G) #顶点数目
```

```
    q = collections.deque() #队列, 即Open表
```

```
    visited = [False for i in range(n)]
```

```
    for i in range(n): #顶点编号0到n-1
```

```
        if not visited[i]:
```

```
            q.append(i)
```

```
            visited[i] = True
```

```
            while len(q) > 0:
```

```
                v = q.popleft() #弹出队头顶点
```

```
                op(v) #访问顶点v
```

```
                for e in G[v]: #G[v]是点v的边的列表, e是Edge对象
```

```
                    if not visited[e.v]: #e.v是边e的另一个顶点, 还有一个是v
```

```
                        q.append(e.v)
```

```
                        visited[e.v] = True
```

➤ 每条边看过一遍或两遍（对无向图可能两遍），每个顶点看过一遍，因此  
复杂度 $O(E+V)$   $E$ 是边数， $V$ 是顶点数



## 图的广度优先遍历 —— 邻接表形式

```
class Edge:
    def __init__(self, v, w):
        self.v, self.w = v, w    #v是顶点, w是权值
```

## 图的广度优先遍历 —— 邻接矩阵形式

```
def bfsTravel2(G, op):  
    import collections  
    n = len(G) #顶点数目  
    q = collections.deque() #队列, 即Open表  
    visited = [False for i in range(n)]  
    for x in range(n): #顶点编号0到n-1  
        if not visited[x]:  
            q.append(x)  
            visited[x] = True  
            while len(q) > 0:  
                v = q.popleft()  
                op(v) #访问顶点v  
                for i in range(n):  
                    if G[v][i]: #G[v][i]不为0说明有边(v,i) 或<v,i>  
                        if not visited[i]:  
                            q.append(i)  
                            visited[i] = True
```

➤ 每个顶点出队一次。对队头顶点，要  $O(V)$  时间查看它和另外所有顶点的关系，因此复杂度  $O(V^2)$   $V$  是顶点数

# 图的遍历的复杂度

用邻接矩阵存储图： $O(V^2)$

用邻接表存储图： $O(E+V)$



北京大学  
PEKING UNIVERSITY

信息科学技术学院

## 图的深度优先搜索



长城入海处：老龙头

# 在图上寻找路径(搜索)

在图上如何寻找从1到8的路径?

➤ 广度优先搜索:

1 2 3 4 8

找到的路径就是最短的

➤ 深度优先搜索

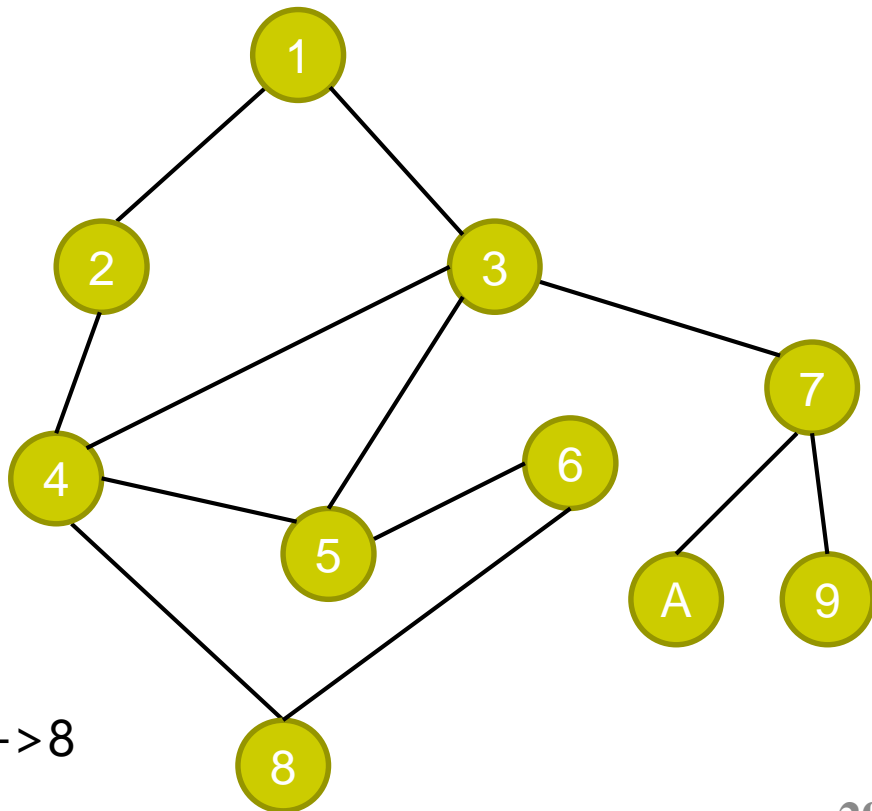
运气最好: 1->2->4->8

运气稍差: 1->2->4->5->6->8

运气坏:

1->3->7->9=>7->A=>7=>3->5->6->8

(双线箭头表示回退)



# 在图上寻找路径

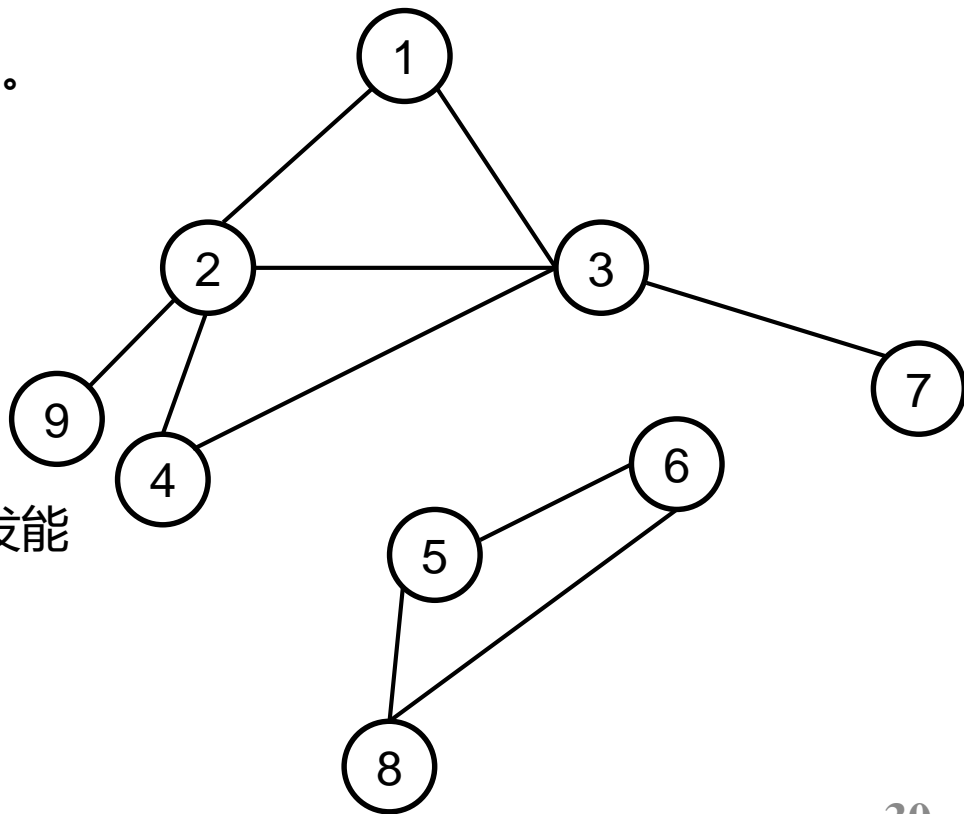
不连通的图，无法从顶点1走到顶点8。

完整的尝试过程可能如下：

1->2->4->3->7=>3=>4=>2->9=>2=>1

结论：不存在从1到8的路径

得出这个结论之前，一定会把从1出发能走到的点全部都走过。



# 在图上寻找路径(深搜)

➤判断从V出发是否能走到终点:

```
def dfs(V):  
    if V为终点:  
        return True  
    将V标记为走过  
    对每个V的没有走过的邻点U:  
        if dfs(U) == True  
            return True  
    return False
```

# 在图上寻找路径(深搜)

```
def main():  
    将所有点都标记为没走过  
    指定终点  
    print(dfs(起点))
```



# 在图上寻找路径(深搜)

➤判断从V出发是否能走到终点,如果能, 要记录所有路径:

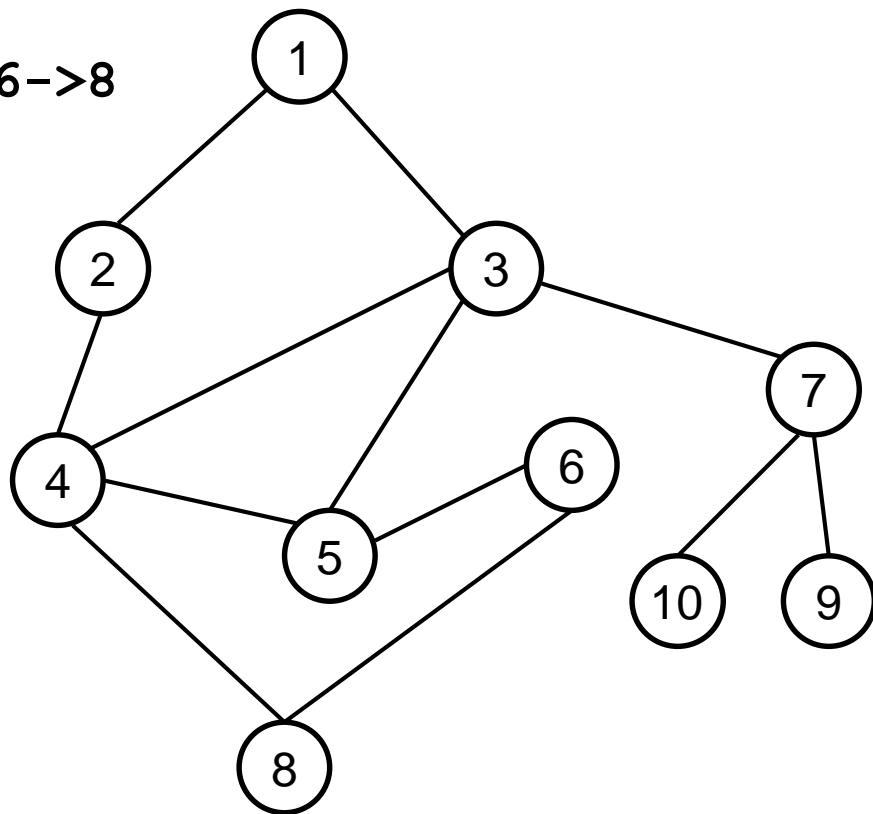
```
path = []    #记录路径上的顶点
def dfs(V):  #寻找从v到终点的所有路径
    path.append(V)
    if V为终点:
        print(path)    #输出一条路径
        path.pop()     #从path中删除v
        return
    将v标记为走过
    对v的每个没走过的邻点U:
        dfs(U)
    path.pop()
    将v标记为没有走过
    return
```

将所有点都标记为没走过  
指定终点  
dfs(起点)

# 在图上寻找路径(深搜)

1->3->7->9=>7->10=>7=>3->5->6->8

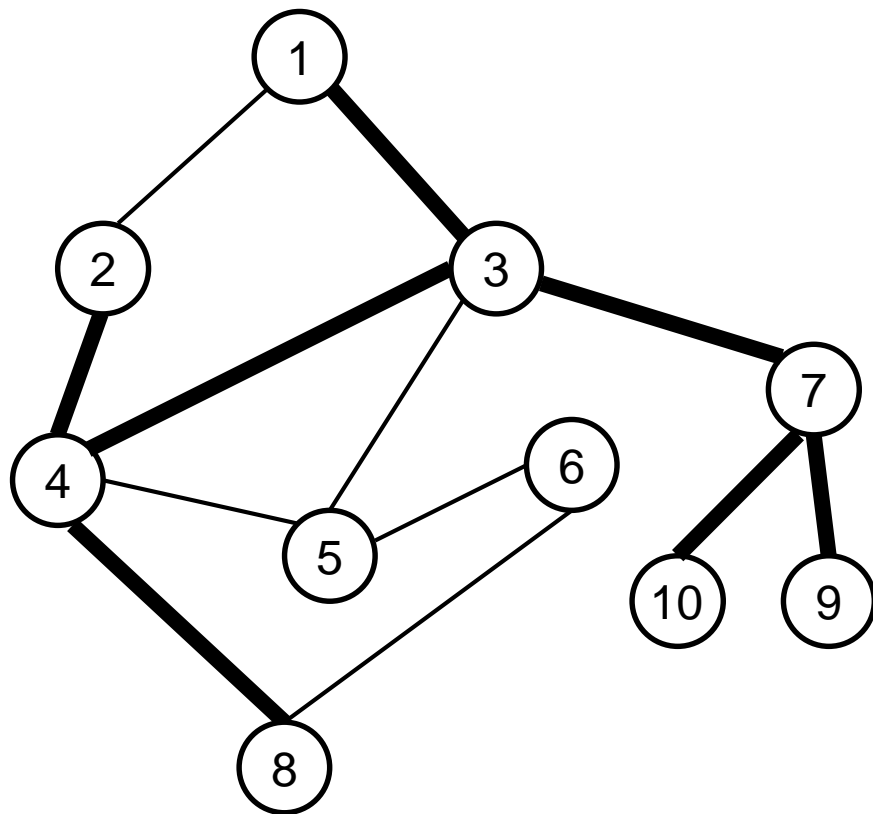
path: 1,3,5,6,8



# 在图上寻找路径(深搜)

1->3->7->9=>7->10=>7=>3->4->2=>4->8

**path:** 1,3,4,8



# 在图上寻找路径(深搜)

## ➤ 许多问题都能转换为在图上寻找路径

状态对应顶点，状态A可以通过一步操作转换到状态B，则状态A到状态B就有边  
求如何将初始状态转换为目标状态，就是在图上找从初始状态到目标状态的路径

例如，给定4个数，要用加减乘除算出24

初始状态：4个数

目标状态：1个数，即24

状态转移：经过一次运算，减少两个原有的数，新增一个数(运算结果)

解决问题未必需要用邻接表或邻接矩阵建图



北京大学  
PEKING UNIVERSITY

信息科学技术学院

深搜寻找最优路径



美国黄石公园大棱镜温泉

# 深搜在图上寻找最优(步数最少)路径

```
bestPath = [0] * MAX_LEN #存放最优路径, MAX_LEN取顶点总数即可
minSteps = INFINITE      #最优路径步数
path = [0] * MAX_LEN     #正在探索的路径
def Dfs(V): #从v出发进行深搜
    global depth
    if v为终点:
        path[depth] = v
        if depth < minSteps:
            minSteps = depth
            拷贝path到bestPath
        return;
    if v为旧点:
        return
    if depth >= minSteps:
        return //最优性剪枝
    将v标记为旧点
    path[depth]=v
    depth+=1
```

# 在图上寻找最优(步数最少)路径

对和v相邻的每个顶点u:

**Dfs (U)**

将v恢复为新点

depth-=1

```
def main():
```

```
    将所有点都标记为新点
```

```
    depth = 0
```

```
    Dfs(起点)
```

```
    if minSteps != INFINITE):
```

```
        for i in range(minSteps+1):
```

```
            print(bestPath[i], end = ",")
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院

## 例题：城堡问题



大连金石滩



## 例题：百练2815 城堡问题

- 右图是一个城堡的地形图。请你编写一个程序，计算城堡一共有多少房间，最大的房间有多大。城堡被分割成 $m \times n$  ( $m \leq 50$ ,  $n \leq 50$ ) 个方块，每个方块可以有0~4面墙。

```

      1      2      3      4      5      6      7
#####
1 #   |   #   |   #   |   |   #
#####-----#####-----#-----#
2 #   #   |   #   #   #   #   #
#-----#####-----#####-----#
3 #   |   |   #   #   #   #   #
#-----#####-----#####-----#
4 #   #   |   |   |   |   #   #
#####

```

(圖 1)

```
# = Wall
| = No wall
- = No wall
```

# 输入输出

- 输入

- 程序从标准输入设备读入数据。
- 第一行是两个整数，分别是南北向、东西向的方块数。
- 在接下来的输入行里，每个方块用一个数字( $0 \leq p \leq 50$ )描述。用一个数字表示方块周围的墙，1表示西墙，2表示北墙，4表示东墙，8表示南墙。**每个方块用代表其周围墙的数字之和表示。**城堡的内墙被计算两次，方块(1,1)的南墙同时也是方块(2,1)的北墙。
- 输入的数据保证城堡至少有两个房间。

- 输出

- 城堡的房间数、城堡中最大房间所包括的方块数。
- 结果显示在标准输出设备上。

- 样例输入

4

7

11 6 11 6 3 10 6

7 9 6 13 5 15 5

1 10 12 7 13 7 5

13 11 10 8 10 12 13

- 样例输出

5

9

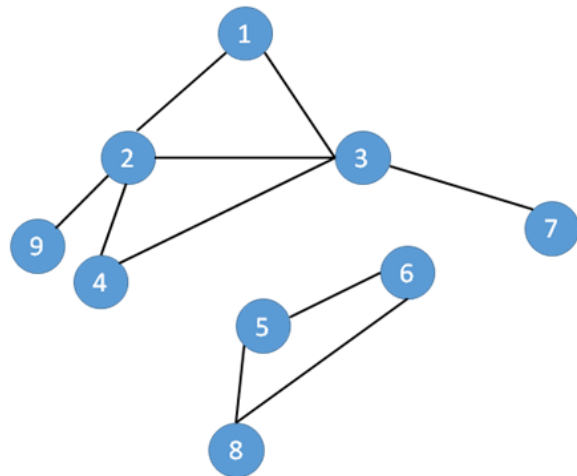
数据保证城堡  
四周都是墙

```
      1   2   3   4   5   6   7
#####
1 #   |   #   |   #   |   #
#####---#####---#---#####---#
2 #   #   |   #   #   #   #   #
#---#####---#####---#####---#
3 #   |   |   #   #   #   #   #
#---#####---#####---#---#
4 #   #   |   |   |   |   #   #
#####
      (图 1)

#   = Wall
|   = No wall
-   = No wall
```

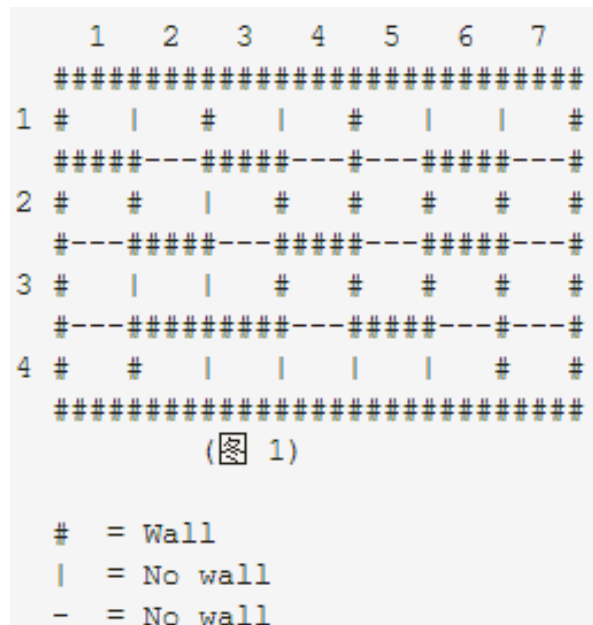
## 解题思路

- 把方块看作是顶点，相邻两个方块之间如果没有墙，则在方块之间连一条边，这样城堡就能转换成一个图。
- 求房间个数，实际上就是在求图中有多少个极大连通子图。
- 一个连通子图，往里头加任何一个图里的其他点，就会变得不连通，那么这个连通子图就是极大连通子图。（如：(8,5,6)）



## 解题思路

- 对每一个房间，深度优先搜索，从而给这个房间能够到达的所有位置染色。最后统计一共用了几种颜色，以及每种颜色的数量。
- 比如  
1 1 2 2 3 3 3  
1 1 1 2 3 4 3  
1 1 1 5 3 5 3  
1 5 5 5 5 5 3  
从而一共有5个房间，最大的房间（1）占据9个格子



```

maxRoomArea = roomNum = roomArea = 0
def Dfs(i, k):
    global roomNum, roomArea
    if color[i][k]:
        return
    roomArea = roomArea + 1
    color[i][k] = roomNum
    if (rooms[i][k] & 1) == 0: Dfs(i, k-1)    # 向西走
    if (rooms[i][k] & 2) == 0: Dfs(i-1, k)    # 向北
    if (rooms[i][k] & 4) == 0: Dfs(i, k+1)    # 向东
    if (rooms[i][k] & 8) == 0: Dfs(i+1, k)    # 向南

```

```

RC = list( map( int, input().split() ) ) #输入格式和题目描述不一致
if len(RC) == 1:
    R = RC[0]
    C = int(input())
else:
    R, C = RC

```

```
rooms = [ [ ] ] #第0行没用
color = [ [ 0 for i in range(C+2)] for i in range(R+2) ]
for i in range(R):
    rooms.append( [0] + list( map( int, input().split() ) ) )
for i in range(1,R+1):
    for k in range(1,C+1):
        if not color[i][k]:
            roomNum += 1
            roomArea = 0
            Dfs(i,k)
            maxRoomArea = max(roomArea,maxRoomArea)

print( roomNum )
print( maxRoomArea )
```

复杂度:  $O(R*C)$



北京大学  
PEKING UNIVERSITY

信息科学技术学院

例题：踩方格



阳朔遇龙河



## 例题：百练4982 踩方格

有一个方格矩阵，矩阵边界在无穷远处。我们做如下假设：

- a. 每走一步时，只能从当前方格移动一格，走到某个相邻的方格上；
- b. 走过的格子立即塌陷无法再走第二次；
- c. 只能向北、东、西三个方向走；

请问：如果允许在方格矩阵上走 $n$ 步( $n \leq 20$ )，共有多少种不同的方案。  
2种走法只要有一步不一样，即被认为是不同的方案。

## 例题：百练4982 踩方格

思路：

递归

从  $(i,j)$  出发，走  $n$  步的方案数，等于以下三项之和：

从  $(i+1,j)$  出发，走  $n-1$  步的方案数。前提： $(i+1,j)$  还没走过

从  $(i,j+1)$  出发，走  $n-1$  步的方案数。前提： $(i,j+1)$  还没走过

从  $(i,j-1)$  出发，走  $n-1$  步的方案数。前提： $(i,j-1)$  还没走过

```

visited = [ [ 0 for i in range(50) ] for i in range(30) ]
def ways( i, j, n):
    if n == 0:
        return 1
    visited[i][j] = 1
    num = 0
    if not visited[i][j-1]:
        num+= ways(i,j-1,n-1)
    if not visited[i][j+1]:
        num+= ways(i,j+1,n-1)
    if not visited[i+1][j]:
        num+= ways(i+1,j,n-1)
    visited[i][j] = 0
    return num

n = int( input() )
print( ways(0,25,n) )

```

	i,j,n	
i-1,j-1,n	i-1,j,n+1	i-1,j+1,n



北京大学  
PEKING UNIVERSITY

信息科学技术学院

例题：算 24



银川沙湖(航拍)

## 例题：算24

给出4个小于10个正整数，你可以使用加减乘除4种运算以及括号把这4个数连接起来得到一个表达式。现在的问题是，是否存在一种方式使得得到的表达式的结果等于24。

样例输入

5 5 5 1

1 1 4 2

0 0 0 0

样例输出

YES

NO

## 例题：算24

思路：

先做一步，即拿两个数来算以下，剩下的问题就变成了3个数算24

```
import math
EPS = 1e-6
def isZero( x ):
    return math.fabs(x) <= EPS
def count24( a , n ): #用列表a里面的头n个元素算24, 返回能否成功
    if n == 1:
        if isZero( a[0] - 24 ):
            return True
        else:
            return False
    b = [ float() for i in range(5) ]
    for i in range( n-1 ):
        for j in range( i+1 , n ): #选 a[i]和a[j]算一下
            m = 0
            for k in range(n): #把a[i],a[j]以外的数放到b开头
                if k != i and k != j:
                    b[m] = a[k]
                    m = m+1
```

```
b[m] = a[i] + a[j]
if count24(b , m+1):
    return True
b[m] = a[i] - a[j]
if count24(b, m + 1):
    return True
b[m] = a[j] - a[i]
if count24(b, m + 1):
    return True
b[m] = a[i] * a[j]
if count24(b, m + 1):
    return True
if not isZero(a[j]):
    b[m] = a[i] / a[j]
    if (count24(b, m+1)): return True
if not isZero(a[i]):
    b[m] = a[j] / a[i]
    if (count24(b, m + 1)):
        return True
```

```
return False
```



```
#main
```

```
while True:
```

```
    a = input().split()
```

```
    a = [ int(c) for c in a ]
```

```
    if isZero( a[0] ):
```

```
        break
```

```
    if count24( a , 4 ):
```

```
        print( "YES" )
```

```
    else:
```

```
        print( "NO" )
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院

例题: Roads



美国黄石公园

## ROADS (POJ1724)

N个城市，编号1到N。城市间有R条单向道路。

每条道路连接两个城市，有长度和过路费两个属性。

Bob只有K块钱，他想从城市1走到城市N。问最短共需要走多长的路。如果到不了N，输出-1

$2 \leq N \leq 100$

$0 \leq K \leq 10000$

$1 \leq R \leq 10000$

每条路的长度  $L$ ,  $1 \leq L \leq 100$

每条路的过路费  $T$ ,  $0 \leq T \leq 100$

输入：

K

N

R

$s_1 e_1 L_1 T_1$

$s_1 e_2 L_2 T_2$

...

$s_R e_R L_R T_R$

s e是路起点和终点

## 解题思路

从城市 1 开始深度优先遍历整个图，找到所有能过到达  $N$  的走法，选一个最优的。

# 解题思路

从城市 1 开始深度优先遍历整个图，找到所有能过到达  $N$  的走法，选一个最优的。

最优性剪枝：

1) 如果当前已经找到的最优路径长度为  $L$  ,那么在继续搜索的过程中，总长度已经大于等于  $L$  的走法，就可以直接放弃，不用走到底了

# 解题思路

从城市 1 开始深度优先遍历整个图，找到所有能到达 N 的走法，选一个最优的。

## 最优性剪枝：

- 1) 如果当前已经找到的最优路径长度为 $L$ ，那么在继续搜索的过程中，总长度已经大于等于 $L$ 的走法，就可以直接放弃，不用走到底了

## 保存中间计算结果用于最优性剪枝：

- 2) 用 $midL[k][m]$  表示：走到城市 $k$ 时总过路费为 $m$ 的条件下，最优路径的长度。若在后续的搜索中，再次走到 $k$ 时，如果总路费恰好为 $m$ ，且此时的路径长度已经不小于 $midL[k][m]$ ，则不必再走下去了。

# 解题思路

另一种通用的最优性剪枝思想 ---保存中间计算结果用于最优性剪枝:

- 2) 如果到达某个状态A时, 发现前面曾经也到达过A, 且前面那次到达A所花代价更少, 则剪枝。这要求保存到达状态A的到目前为止的最少代价。

用 $midL[k][m]$  表示: 走到城市k时总过路费为m的条件下, 最优路径的长度。若在后续的搜索中, 再次走到k时, 如果总路费恰好为m, 且此时的路径长度已经不小于 $midL[k][m]$ , 则不必再走下去了。

```
MX = 110
INF = 1 << 30
class Road:
    def __init__(self,d,L,t):
        self.d,self.L,self.t = d,L,t

cityMap = [[] for i in range(MX)] #邻接表。cityMap[i]是从点i有路连
到的城市集合
minLen = INF #当前找到的最优路径的长度
totalLen = 0 #正在走的路径的长度
totalCost = 0 #正在走的路径的花销
visited = [0] * MX #城市是否已经走过的标记
minL = [[INF for j in range(10100)] for i in range(MX)]
#minL[i][j]表示从1到i点的，花销为j的最短路的长度
```



```
def Dfs(s): #从 s开始向N行走
    global N,minLen,totalLen,totalCost,cityMap,K
    if s == N :
        minLen = min(minLen,totalLen)
        return
    for i in range( len(cityMap[s])):
        d = cityMap[s][i].d #s 有路连到d
        if visited[d] == 0:
            cost = totalCost + cityMap[s][i].t
            if cost > K:
                continue
            if totalLen+cityMap[s][i].L >= minLen or \
                totalLen + cityMap[s][i].L >= minL[d][cost]:
                continue
            totalLen += cityMap[s][i].L
            totalCost += cityMap[s][i].t
            minL[d][cost] = totalLen
            visited[d] = 1
            Dfs(d)
```

```
visited[d] = 0
totalCost -= cityMap[s][i].t
totalLen -= cityMap[s][i].L
```

```
#main
```

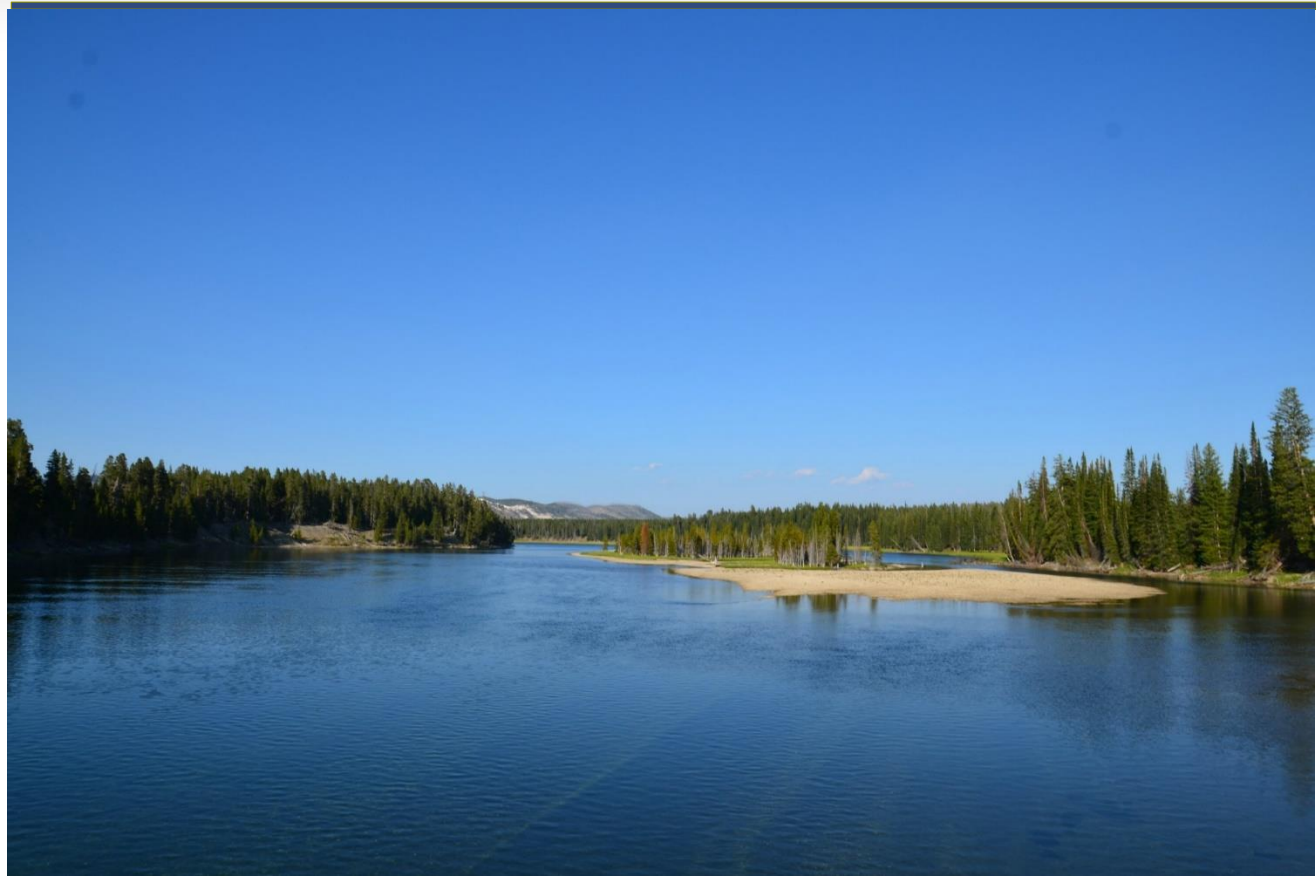
```
K = int(input())
N = int(input())
R = int(input())
for i in range(R):
    r = Road(0,0,0)
    s,r.d,r.L,r.t = map(int,input().split())
    if s != r.d:
        cityMap[s].append(r)
totalLen = totalCost = 0
visited[1] = 1
Dfs(1)
if minLen < INF:
    print(minLen)
else:
    print(-1)
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院

例题:生日蛋糕



美国黄石公园

## 生日蛋糕 (POJ1190)

要制作一个体积为 $N\pi$ 的 $M$ 层生日蛋糕，每层都是一个圆柱体。

设从下往上数第 $i$  ( $1 \leq i \leq M$ )层蛋糕是半径为 $R_i$ ，高度为 $H_i$ 的圆柱。当 $i < M$ 时，要求 $R_i > 1 + R_{i+1}$ 且 $H_i > 1 + H_{i+1}$ 。

由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积 $Q$ 最小。

$$\text{令 } Q = S\pi$$

请编程对给出的 $N$ 和 $M$ ，找出蛋糕的制作方案（适当的 $R_i$ 和 $H_i$ 的值），使 $S$ 最小。

（除 $Q$ 外，以上所有数据皆为正整数）

# 解题思路

- 深度优先搜索，枚举什么？

# 解题思路

- 深度优先搜索，枚举什么？  
枚举每一层可能的高度和半径。
- 如何确定搜索范围？

# 解题思路

- 深度优先搜索，枚举什么？  
枚举每一层可能的高度和半径。
- 如何确定搜索范围？  
底层蛋糕的最大可能半径和最大可能高度
- 搜索顺序，哪些地方体现搜索顺序？

# 解题思路

- 深度优先搜索，枚举什么？  
枚举每一层可能的高度和半径。
- 如何确定搜索范围？  
底层蛋糕的最大可能半径和最大可能高度
- 搜索顺序，哪些地方体现搜索顺序？  
从底层往上搭蛋糕，而不是从顶层往下搭  
在同一层进行尝试的时候，半径和高度都是从大到小试
- 如何剪枝？



# 剪枝

- 剪枝1：搭建过程中发现已建好的面积已经**不小于**目前求得的最优表面积，或者预见搭完后面积一定会**不小于**目前最优表面积,则停止搭建  
(最优性剪枝)

# 剪枝

- 剪枝1：搭建过程中发现已建好的面积已经**不小于**目前求得的最优表面积，或者预见到搭完后面积一定会**不小于**目前最优表面积,则停止搭建  
(**最优性剪枝**)
- 剪枝2：搭建过程中预见到再往上搭，高度已经无法安排，或者半径已经无法安排，则停止搭建(**可行性剪枝**)

# 剪枝

- 剪枝1：搭建过程中发现已建好的面积已经**不小于**目前求得的最优表面积，或者预见到搭完后面积一定会**不小于**目前最优表面积,则停止搭建  
(**最优性剪枝**)
- 剪枝2：搭建过程中预见到再往上搭，高度已经无法安排，或者半径已经无法安排，则停止搭建(**可行性剪枝**)
- 剪枝3：搭建过程中发现还没搭的那些层的体积，一定会超过还缺的体积，则停止搭建(**可行性剪枝**)

# 剪枝

- 剪枝1：搭建过程中发现已建好的面积已经**不小于**目前求得的最优表面积，或者预见到搭完后面积一定会**不小于**目前最优表面积,则停止搭建  
(**最优性剪枝**)
- 剪枝2：搭建过程中预见到再往上搭，高度已经无法安排，或者半径已经无法安排，则停止搭建(**可行性剪枝**)
- 剪枝3：搭建过程中发现还没搭的那些层的体积，一定会超过还缺的体积，则停止搭建(**可行性剪枝**)
- 剪枝4：搭建过程中发现还没搭的那些层的体积，最大也到不了还缺的体积，则停止搭建(**可行性剪枝**)

```

import math
minArea = 1 << 30 #最优表面积
area = 0 #正在搭建中的蛋糕的表面积
minV = [0]*30 # minV[n]表示n层蛋糕最少的体积
minA = [0]*30 # minA[n]表示n层蛋糕的最少侧表面积
def MaxVforNRH(n,r,h):
#求在n层蛋糕，底层最大半径r，最高高度h的情况下，能凑出来的最大体积
    v = 0
    for i in range(n):
        v += (r - i ) *(r-i) * (h-i)
    return v
def Dfs(v, n, r, h):
#要用n层去凑体积v,最底层半径不能超过r,高度不能超过h
#求出最小表面积放入 minArea
    global minArea,area,M
    if n == 0:
        if v != 0: return
    else:
        minArea = min(minArea,area)
        return

```

```
if v <= 0:
    return
if minV[n] > v: #剪枝3
    return
if area + minA[n] >= minArea: #剪枝1
    return
if h < n or r < n: #剪枝2
    return

if MaxVforNRH(n,r,h) < v: #剪枝4
    #这个剪枝最强!
    return
#for rr in range(n,r+1): 这种写法比从大到小慢5倍
for rr in range(r,n-1,-1):
    if n == M : #底面积
        area = rr * rr
    for hh in range(h,n-1,-1):
        area += 2 * rr * hh
        Dfs(v-rr*rr*hh,n-1,rr-1,hh-1)
        area -= 2 * rr * hh
```

```

#main
N = int(input())
M = int(input())    #M层蛋糕, 体积N
minV[0] = minA[0] = 0
for i in range(1,M+1):
    minV[i] = minV[i-1] + i * i * i
    minA[i] = minA[i-1] + 2 * i * i
if minV[M] > N:
    print(0)
else:
    maxH = int((N - minV[M-1]) / (M*M)) + 1 #底层最大高度
    #最底层体积不超过 (N-minV[M-1]), 且半径至少M
    maxR = int(math.sqrt((N-minV[M-1]) / M)) + 1 #底层高度至少M
    area = 0
    minArea = 1 << 30
    Dfs( N,M,maxR,maxH)
    if minArea == 1 << 30:
        print(0)
    else:
        print(minArea)

```

还有什么可以改进



# 还有什么可以改进

- 1) 用数组存放 `MaxVforNRH(n,r,h)` 的计算结果，避免重复计算

# 还有什么可以改进

1) 用数组存放 `MaxVforNRH(n,r,h)` 的计算结果，避免重复计算

2)

```
for rr in range(r,n-1,-1):  
    if n == M : #底面积  
        area = rr * rr  
    for hh in range(h,n-1,-1):  
        area += 2 * rr * hh  
        Dfs(v-rr*rr*hh,n-1,rr-1,hh-1)
```

#加上对本次Dfs失败原因的判断。如果是因为剩余体积不够大而失败，那么就用不着试下一个高度，直接break；或者由小到大枚举 h.....

```
area -= 2 * rr * hh
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院

# 广度优先搜索 例题: 抓住那头牛



美国黄石公园

## 抓住那头牛(百练习4001)

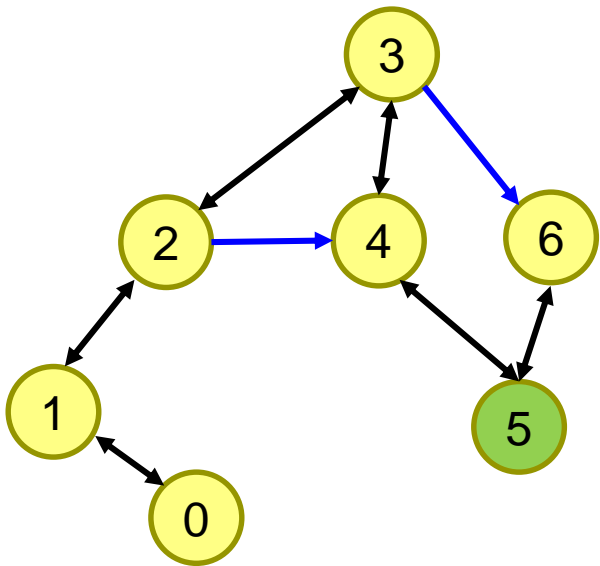
农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 $N$  ( $0 \leq N \leq 100000$ )，牛位于点 $K$  ( $0 \leq K \leq 100000$ )。农夫有两种移动方式：

- 1、从 $X$ 移动到 $X-1$ 或 $X+1$ ，每次移动花费一分钟
- 2、从 $X$ 移动到 $2 \times X$ ，每次移动花费一分钟

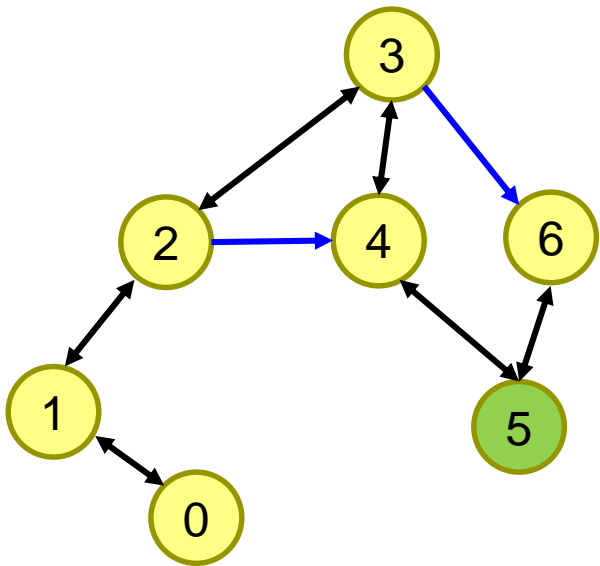


假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？

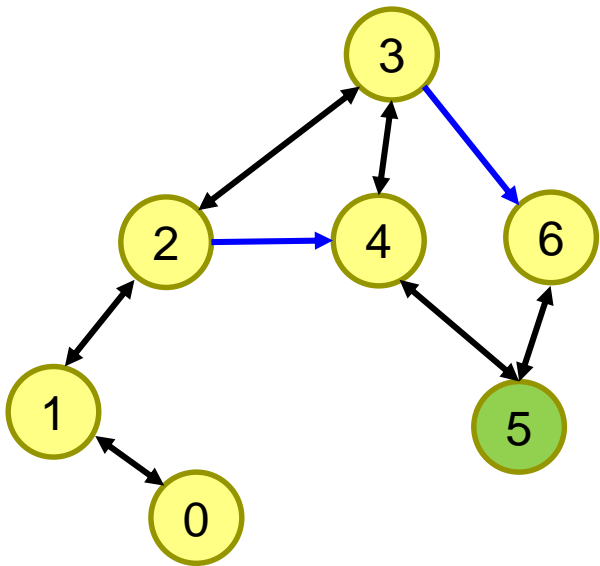


**策略1) 深度优先搜索：**从起点出发，随机挑一个方向，能往前走就往前走(扩展)，走不动了则回溯。不能走已经走过的点(要判重)。

假设农夫起始位于点3，牛位于5

$N=3, K=5$ ，最右边是6。

如何搜索到一条走到5的路径？



运气好的话：

3->4->5

或

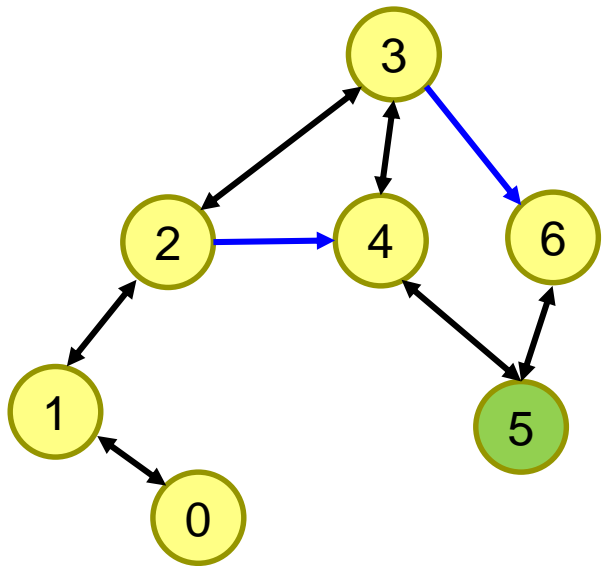
3->6->5

问题解决！

假设农夫起始位于点3，牛位于5

$N=3, K=5$ ，最右边是6。

如何搜索到一条走到5的路径？



运气不太好的话：

3->2->4->5

运气最坏的话：

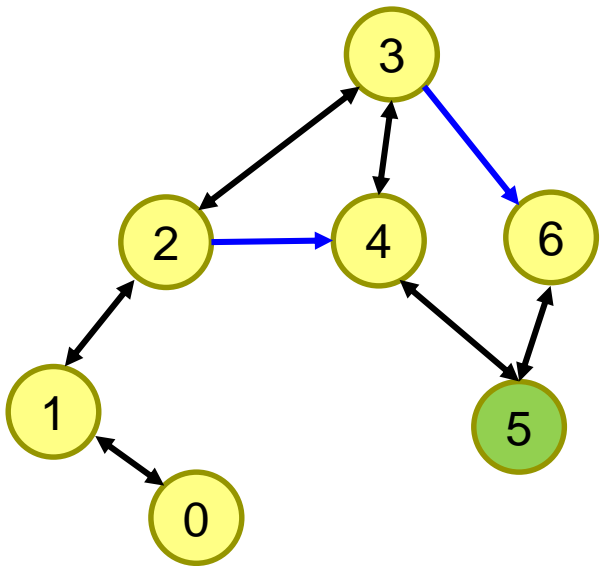
3->2->1->0->4->5

要想求最优(短)解，则要遍历所有走法。可以用各种手段优化，比如，若已经找到路径长度为 $n$ 的解，则所有长度大于 $n$ 的走法就不必尝试。

运算过程中需要存储路径上的顶点，数量较少。  
用栈存顶点。



假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



### 策略2) 广度优先搜索:

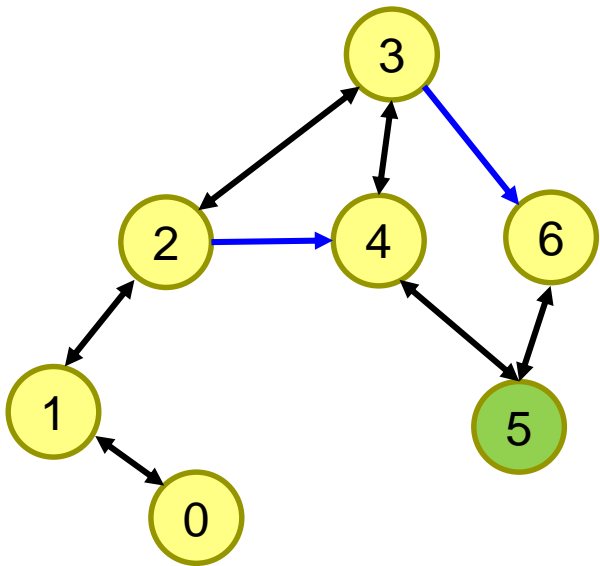
给顶点分层。起点是第0层。从起点最少需 $n$ 步就能到达的点属于第 $n$ 层。

第1层: 2,4,6

第2层: 1,5

第3层: 0

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？

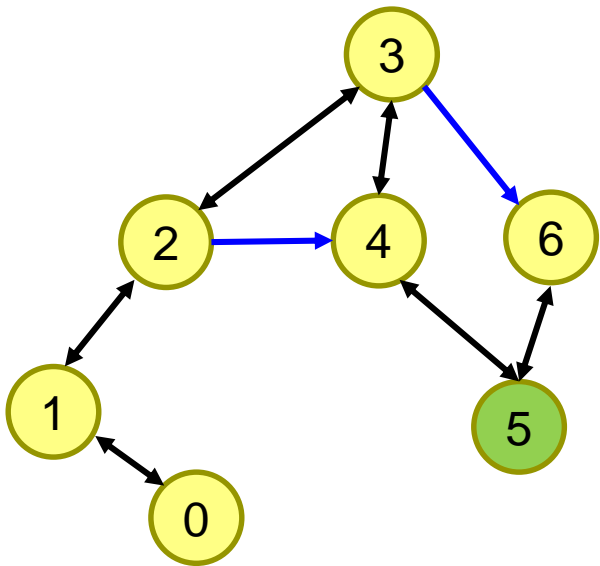


## 策略2) 广度优先搜索:

给顶点分层。起点是第0层。从起点最少需 $n$ 步就能到达的点属于第 $n$ 层。

依层次顺序，从小到大扩展顶点。  
把层次低的点全部扩展出来后，才会扩展层次高的点。

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



## 策略2) 广度优先搜索:

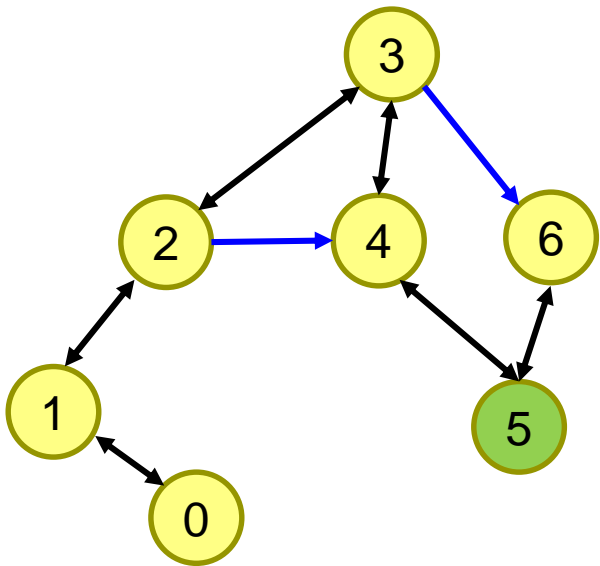
搜索过程 (顶点扩展过程) :

3  
2 4 6  
1 5

问题解决。

扩展时，不能扩展出已经走过的顶点(要判重)。

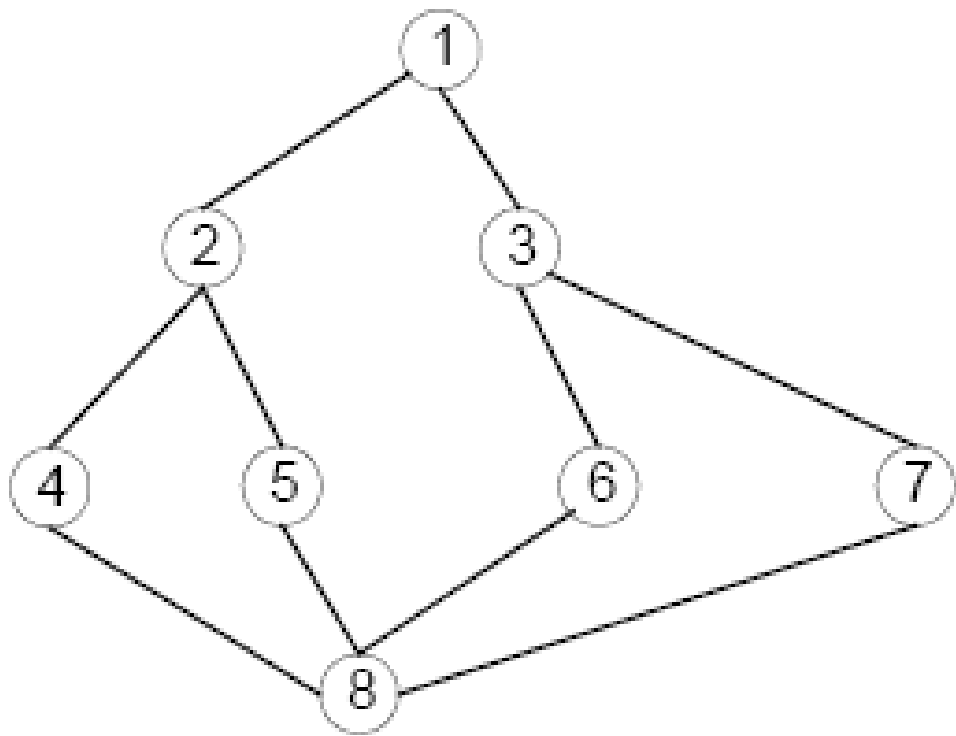
假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



## 策略2) 广度优先搜索:

可确保找到最优解，但是因扩展出来的顶点较多，且多数顶点都需要保存，因此需要的存储空间较大。  
用队列存顶点。

## 深搜 vs. 广搜



若要遍历所有顶点：

□ 深搜

1-2-4-8-5-6-3-7

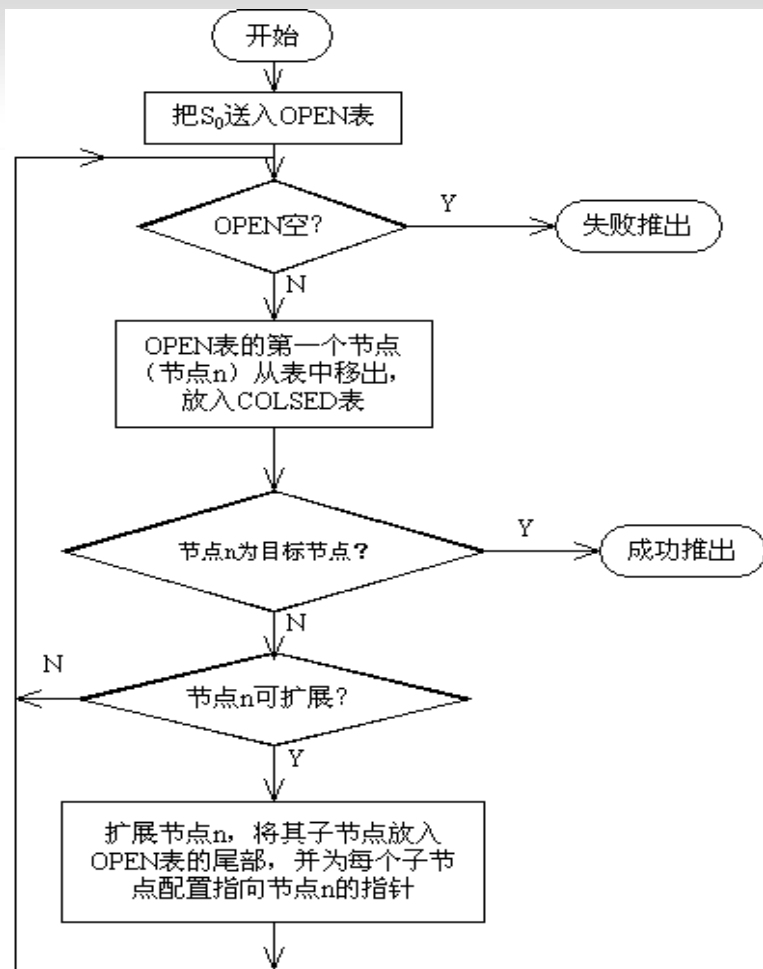
□ 广搜

1-2-3-4-5-6-7-8

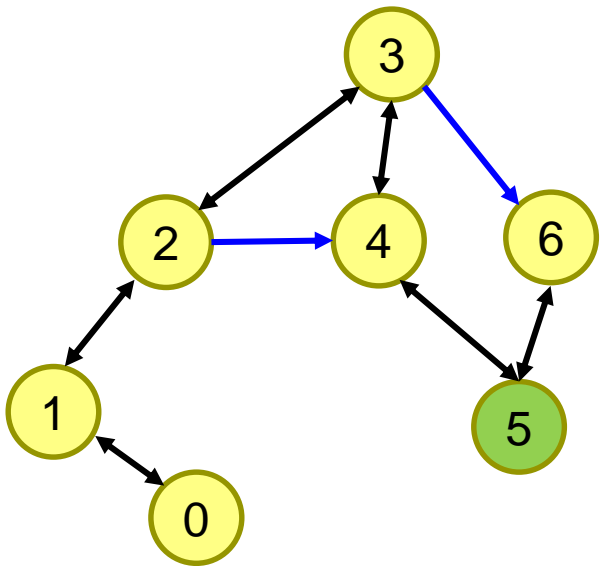
# 广搜算法

□ 广度优先搜索算法如下：（用QUEUE）

- (1) 把初始顶点 $S_0$ 放入Open表中；
- (2) 如果Open表为空，则问题无解，失败退出；
- (3) 把Open表的第一个顶点取出放入Closed表，并记该顶点为 $n$ ；
- (4) 考察顶点 $n$ 是否为目标顶点。若是，则得到问题的解，成功退出；
- (5) 若顶点 $n$ 不可扩展，则转第(2)步；
- (6) 扩展顶点 $n$ ，将其不在Closed表和Open表中的子顶点(判重)放入Open表的尾部，并为每一个子顶点设置指向父顶点的指针(或记录顶点的层次)，然后转第(2)步。



假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



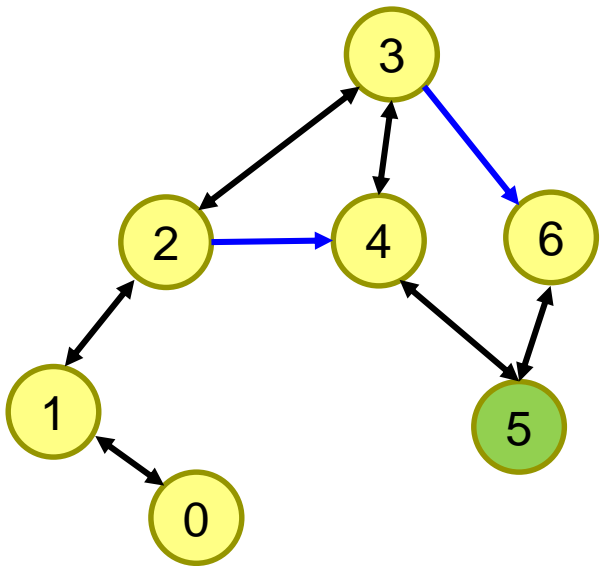
广度优先搜索队列变化过程：

3

Closed

Open

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

3

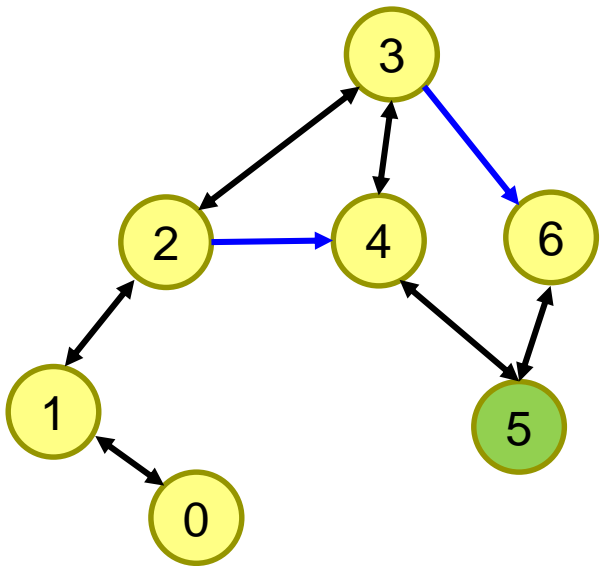
2 4 6

Closed

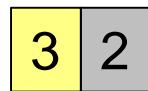
Open



假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

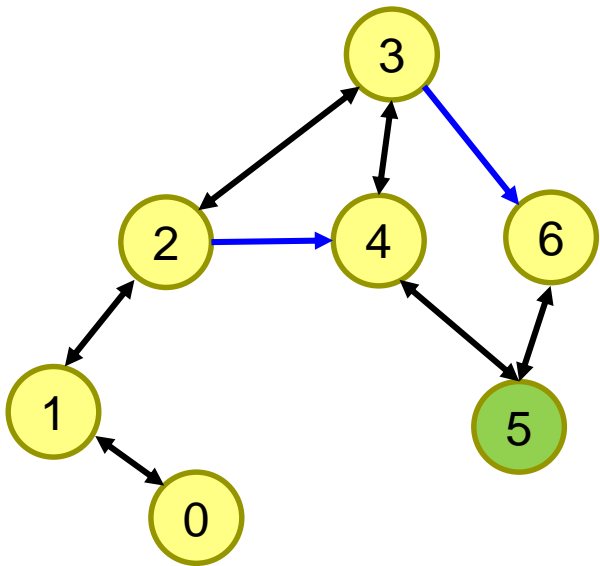


Closed



Open

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：

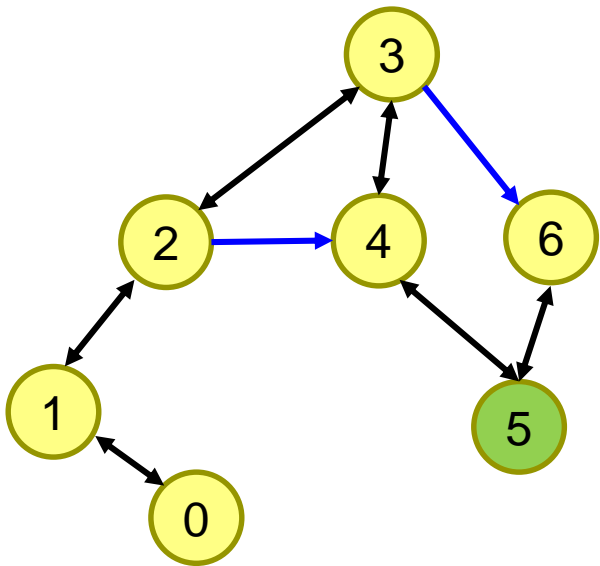


Closed

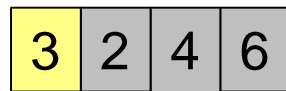


Open

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



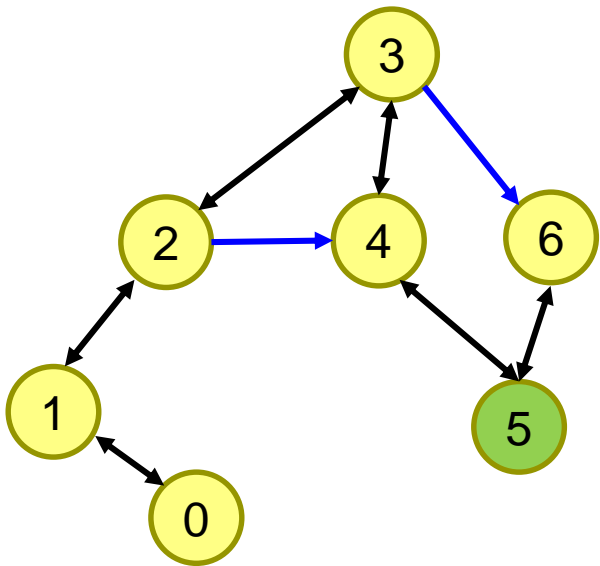
广度优先搜索队列变化过程：



Closed

Open

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



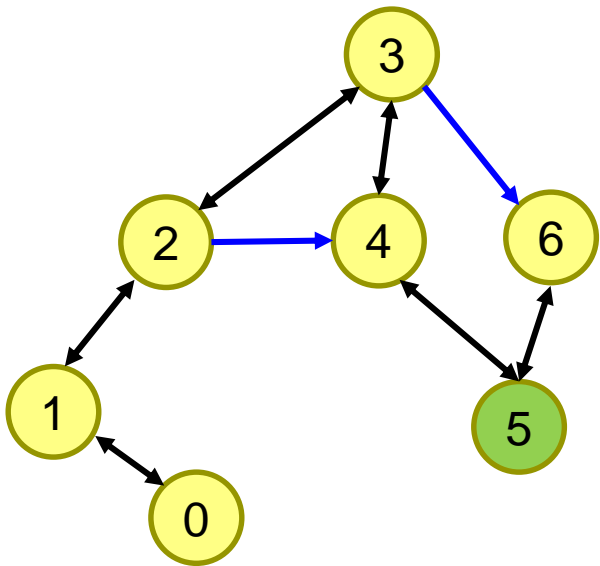
广度优先搜索队列变化过程：



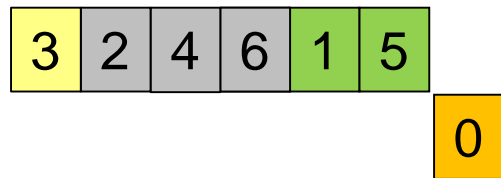
Closed

Open

假设农夫起始位于点3，牛位于5  
 $N=3, K=5$ ，最右边是6。  
如何搜索到一条走到5的路径？



广度优先搜索队列变化过程：



Closed

Open

目标顶点5出队列，问题解决！

## #poj3278 Catch That Cow

```
import collections
class step:
    def __init__(self,x,steps):
        self.x = x          #位置
        self.steps = steps  #到达x所需的步数
MAXN = 100000
N,K = map(int,input().split())
q = collections.deque()    #队列,即Open表
visited = [False] * (MAXN+10)
q.append(step(N,0))
visited[N] = True
```

```
while len(q) > 0:
    s = q.popleft()
    if s.x == K: #找到目标
        print(s.steps)
        break
    else:
        if s.x - 1 >= 0 and not visited[s.x-1]:
            q.append(step(s.x-1,s.steps+1))
            visited[s.x-1] = 1
        if s.x + 1 <= MAXN and not visited[s.x+1]:
            q.append(step(s.x+1,s.steps+1))
            visited[s.x+1] = 1
        if s.x * 2 <= MAXN and not visited[s.x*2]:
            q.append(step(s.x*2,s.steps+1))
            visited[s.x*2] = 1
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院

广度优先搜索  
例题: 迷宫问题



美国黄石公园



## 迷宫问题 (百练4127)

定义一个矩阵：

```
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

# 迷宫问题

基础广搜。先将起始位置入队列

每次从队列拿出一个元素，扩展其相邻的4个元素入队列(要用二维标志列表判重)，直到队头元素为终点为止。队列里的元素记录了指向父顶点（上一步）的指针

队列元素：(r,c,father)

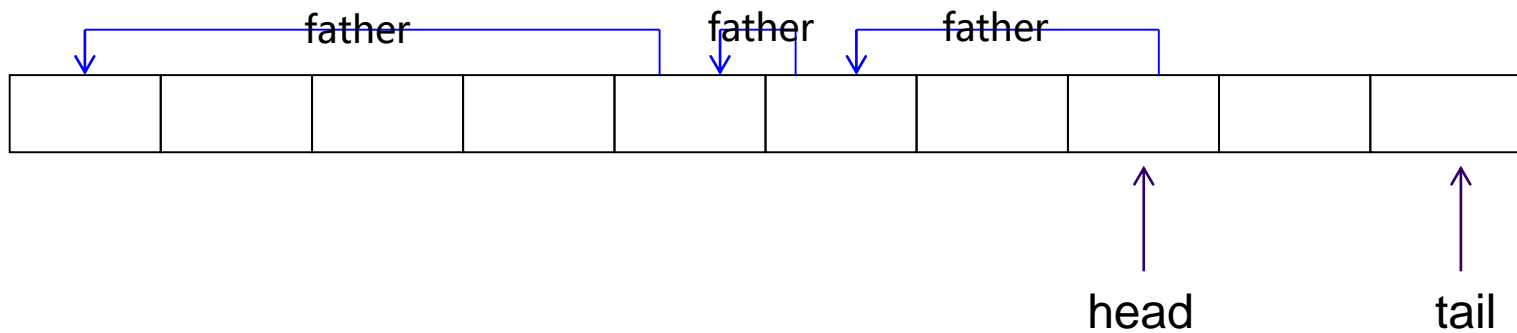
r,c: 顶点的坐标

father: 父顶点在队列中的下标(从a走到b,则a是b的父顶点)

判重的二维列表：flags[i][j]表示 (i,j)那个位置是否走过，即是否入过队列

# 迷宫问题

- 队列不能用collections.deque，要自己写。用一个足够大的列表实现，维护一个队头指针和队尾指针
- 足够大：能放下所有顶点



当队头元素为目标时，沿father指针链取出行走过程中的每个顶点的倒序



北京大学  
PEKING UNIVERSITY

信息科学技术学院

广度优先搜索  
例题: 鸣人和佐助



美国黄石公园

## 鸣人和佐助(百练6044)

已知一张地图（以二维矩阵的形式表示）以及佐助和鸣人的位置。地图上的每个位置都可以走到，只不过有些位置上有大蛇丸的手下(#)，需要先打败大蛇丸的手下才能到这些位置。

鸣人有一定数量的查克拉，每一个单位的查克拉可以打败一个大蛇丸的手下。假设鸣人可以往上下左右四个方向移动，每移动一个距离需要花费1个单位时间，打败大蛇丸的手下不需要时间。如果鸣人查克拉消耗完了，则只可以走到没有大蛇丸手下的位置，不可以再移动到有大蛇丸手下的位置。

佐助在此期间不移动，大蛇丸的手下也不移动。请问，鸣人要追上佐助最少需要花费多少时间？

```
4 4 1
#@##
**##
###+
****
```

# 鸣人和佐助

状态定义为：

$(r, c, k)$ ，鸣人所在的行，列和查克拉数量

如果队头顶点扩展出来的顶点是有大蛇手下的顶点，则其  $k$  值比队头的  $k$  要减掉 1。如果队头顶点的查克拉数量为 0，则不能扩展出有大蛇手下的顶点。

```
4 4 1
#@##
**##
###+
****
```

# 求钥匙的鸣人

不再有大蛇丸的手下。

但是佐助被关在一个格子里，需要依次集齐 $k$ 种钥匙才能打开格子里的门救出他。

$K$ 种钥匙散落在迷宫里。有的格子里放有一把钥匙。一个格子最多放一把钥匙。走到放钥匙的格子，即得到钥匙。

鸣人最少要走多少步才能救出佐助。

# 求钥匙的鸣人

状态:

$(r, c, \text{keys})$  : 鸣人的行, 列, 已经拥有的钥匙种数

目标状态  $(x, y, K)$   $(x, y)$  是佐助呆的地方

如果队头顶点扩展出来的顶点上面有不曾拥有的某种钥匙, 则该顶点的  $\text{keys}$  比队头顶点的  $\text{keys}$  要加1



# 迷宫问题变形一(百练4980, 拯救行动)

鸣人要从迷宫中的起点 r 走到终点 a, 去营救困在那里的佐助。迷宫中各个字符代表道路 (@)、墙壁 (#)、和守卫 (x)。

能向上下左右四个方向走。不能走到墙壁。  
每走一步需要花费1分钟

行走过程中一旦遇到守卫, 必须杀死守卫才能继续前进。杀死一个守卫需要花费额外的1分钟

求到达目的地最少用时

```
# @ # # # # @  
# @ a # @ @ r @  
# @ @ # x @ @ @  
@ @ # @ @ # @ #  
# @ @ @ # # @ @  
@ # @ @ @ @ @ @  
@ @ @ @ @ @ @ @
```



# 迷宫问题变形一(百练4980, 拯救行动)

解法一:

队列里放以下结构:

```
struct Position
{
    int r,c;
    int steps;
};
```

将 'x' 对应的节点放入队列时, 直接将其steps多加1

```
2
7 8
#####@
#@a#@@r@
###x@@@
@@#@@#@#
#####@@
#####@
@@@@@@@@@
```

# 迷宫问题变形一(百练4980, 拯救行动)

解法一:

队列里放以下结构:

```
struct Position
{
    int r,c;
    int steps;
};
```

将 'x' 对应的节点放入队列时, 直接将其steps多加1

队列是优先队列, steps最小的在队头

广搜一定要确保步数少的先于步数多的出队列!!!

```
2
7 8
#####@
#a#@@r@
###x@@@
@@#@#@#
#####@@
#####@
@@@@@@@@
```

# 迷宫问题变形一(百练4980, 拯救行动)

解法二:

●状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫  
    int t; //走到本节点花的时间  
};
```

# 迷宫问题变形一(百练4980, 拯救行动)

- 状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫, 没守卫的地方就算已经杀死  
    int t; //走到本节点花的时间  
};
```

- 若(r,c)处没有守卫, 则进入(r,c)时的状态是 (r,c,1,t)可以扩展出 (r+1,c,?,t+1), (r-1,c, ?,t+1), (r,c+1, ?,t+1), (r,c-1, ?,t+1)

如果有守卫, ? 就是0,否则就是1

# 迷宫问题变形一(百练4980, 拯救行动)

- 状态表示:

```
struct Pos {  
    int r,c; //本节点的位置  
    bool kill; //是否杀死过守卫, 没守卫的地方就算已经杀死  
    int t; //走到本节点花的时间  
};
```

- 若(r,c)处没有守卫, 则进入(r,c)时的状态是 (r,c,1,t), 可以扩展出 (r+1,c,?,t+1), (r-1,c, ?,t+1), (r,c+1, ?,t+1), (r,c-1, ?,t+1)

如果有守卫, ? 就是0,否则就是1

- 若(r,c)处有守卫, 则进入(r,c)时的状态是 (r,c,0,t)只能扩展出 (r,c,1,t+1)

# 迷宫问题变形一(百练4980, 拯救行动)

判重数组:

```
int flag[M][N][2];
```

flag[r][c][0]表示在坐标(r,c), 尚未杀死守卫的情况

flag[r][c][1]表示在坐标(r,c), 已经杀死守卫的情况

## 最倒霉的鸣人(百练8436)

要从迷宫中的起点  $r$  走到终点  $a$ , 迷宫中各个字符代表道路 (@)、墙壁 (#)、和守卫 (x), 放有钥匙的道路 (1--9, 表示有9种钥匙)

行走过程中一旦遇到守卫, 必须杀死守卫才能继续前进。杀死一个守卫需要花费额外1分钟。最多5个守卫。

走到终点时, 必须要每种钥匙至少有一把才算完成任务。钥匙不全, 也可以经过终点。

想拿第  $k$  种钥匙, 必须手里已经有第  $k-1$  种钥匙。拿不了钥匙, 也可以经过放钥匙的地方

求完成任务最少用时

```
# @ # # # # # @  
# @ a # @ @ r @  
# @ @ # x @ @ @  
@ @ # @ @ # 1 #  
# @ @ 2 # # @ @  
@ # @ @ 5 @ @ @  
@ @ @ @ @ @ @ @
```



```
struct Status
{
    short r,c;
    short keys;
    short foughted;           //守卫是否打过
    int steps;
    short layout;             //守卫的局面(哪些被杀, 哪些还没被杀)
};

char flags[100][100][10][33][2]; //判重, 也可以用set存放扩展过德
状态(用元组表示)来判重
```

`flags[r][c][k][x][f]` 对应的状态是:

在位置  $(r,c)$ , 手里有  $k$  把钥匙, 在位置  $(r,c)$  是否打过首卫的情况是  $f$ , 守卫的局面是  $x$

一共只有5个守卫, 他们被杀或没被杀的情况一共有32种, 可以用5个bit表示