

**알고리즘 과제 4**

**과목명 알고리즘**

**담당교수 김희철교수님**

**제출일 20211203**

**전공 컴퓨터전자시스템**

**학번 201904458**

**이름 이준용**

* **문제 1-1번**

다음과 같이 n(1이상 5,000이하 정수)개의 셀들로 구성된 하나의 행이 있다. 각 셀은 왼쪽부터 차례대로 0부터 n-1까지 번호가 붙여져 있고, 셀 안에는 음이 아닌 정수가 주어져 있다. 이들 셀 중 하나(아래 그림의 빗금 친 셀)에 마커가 있다.

각 단계에서 마크가 있는 셀 안의 정수만큼 왼쪽 혹은 오른쪽으로 마크를 이동할 수 있다. 단 왼쪽 끝 혹은 오른쪽 끝을 벗어날 수 없다. 아래 그림은 0번 셀로부터 7번 셀로 이동하는 과정을 보여준다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 마커 시작위치 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 오른쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 왼쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 오른쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 왼쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 오른쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 오른쪽 이동 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 마커 시작위치 | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

마커가 7번 셀에 있을 경우, 이 마커를 0번 셀로 이동하는 것은 불가능하다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 셀 번호 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  | 3 | 6 | 4 | 1 | 3 | 4 | 2 | 5 | 3 | 0 |

어느 위치에 마커를 두면, 가장 많은 셀에 이 마크가 도달할 수 있는지를 구하는 프로그램을 작성하시오. 가장 많은 셀에 도달하게 되는 셀(마크가 있는 셀)이 여러 개일 경우 이들 셀을 번호 순서(오름차순)대로 출력한다.

입력 예 1

10 // n (1이상 100,000이하 정수)

3 6 4 1 3 4 2 5 3 0

출력 예 1

0

입력 예 2

10 // n (1이상 100,000이하 정수)

1 1 1 1 1 1 1 1 1 1

출력 예 2

0 1 2 3 4 5 6 7 8 9

* **알고리즘 및 자료구조**

깊이탐색우선을 사용한 1차원 셀 방문하기 문제는 recursion을 이용하여 풀었습니다.

Has\_duplicates함수는 가장 많은 셀에 도착한 index가 중복인지 아닌지 판별해주는 함수입니다.

0부터 입력받은 n까지 시작위치로서 bfs함수를 돌게 됩니다.

Dfs()함수는 시작위치 – I 번째 배열값 >=0 과 i+arr[i] <n에 따라 recursion을 통해 현재 방문하고 있는 위치에 따라서 append합니다.

Visited배열에서 True 일 경우 방문처리 된 index이고 False일 경우 방문처리가 안된 index를 뜻합니다.

* **시간복잡도 분석**

Arr list를 만들고 1차원 배열 입력 받는데 드는 시간복잡도 O(n)

dfs함수를 실행 하기전에 시작위치부터 n까지 for구문을 돌면서 dfs()를 실행하게 됩니다. -> O(n)

dfs함수에서는 평균적으로 O(n)보다는 적게 들지만 최대 O(n)만큼의 시간복잡도가 발생합니다.

이는 n^2 + n이므로 모든 시간 복잡도는 O(n^2)입니다.

* **느낀 점**

10   
3 6 4 1 3 4 2 5 3 0

의 테스트 케이스는 해결했지만 지금의 코드로는

10   
1 1 1 1 1 1 1 1 1 1

을 해결 못했습니다. 위의 테스트 케이스를 해결하려면 if -else 구문으로 해결 가능하지만 if – else로 바꿀 경우 첫번째 케이스를 해결이 안됐습니다. 결국 해결 못한 채로 제출했습니다.

* **프로그램 코드**

import sys  
sys.setrecursionlimit(10000)  
def has\_duplicates(seq): # 중복인지 아닌지 판별해주는 함수  
 return len(seq) != len(set(seq))  
  
def dfs(i, n, arr,visited): # 탐색 방법: 깊이 우선 탐색  
 visited[i] = True # 0부터 n까지 입력 받은 시작 위치 i, 시작위치는 방문했다고 표시 True  
 result.append(i) # result 배열에 현재 방문하고 있는 위치 append  
 #print(result)  
 if i - arr[i] >=0: # 시작위치 - i번째 배열값 >=0  
 if i == n-1 and arr[-1] > 0: # 1차원 배열에서 마지막 배열 값이 0보다 큰경우 -> 왼쪽 으로 이동 가능  
 result.append(i) # result배열에 append  
 return result  
 if visited[i - arr[i]] is False: # visited배열에서 이미 방문한 배열인지 아닌지 판단  
 return dfs(i-arr[i],n,arr,visited) # 깊이우선탐색 recursion  
 if i+arr[i] < n: # 시작위치 + i번째 배열값 < n  
 if i == n-1 and arr[-1] > 0: # 1차원 배열에서 마지막 배열 값이 0보다 큰경우 -> 왼쪽 으로 이동 가능  
 result.append(i) # result배열에 append  
 return result  
 if visited[i + arr[i]] is False: # visited배열에서 이미 방문한 배열인지 아닌지 판단  
 return dfs(i+arr[i],n,arr,visited) # 깊이우선탐색 recursion  
 return False # 위의 if 구문 모두 해당 안될 경우 break  
  
n = int(sys.stdin.readline()) # 1차원 배열 길이 입력  
result=[] # 검사용 # 방문하고 있는 위치를 담는 list  
result\_num =[0] \*n # 마지막으로 방문한 index의 배열값을 시작위치 index에 입력 받는 0으로 구성된 list  
num = list(map(int, sys.stdin.readline().split())) # 1차원 셀 각각의 셀값 입력  
arr = [0 for i in range(n)] # arr에는 최소값이 들어감  
for i in range(n):  
 arr[i] = num[i] # num값을 arr배열에 대입  
  
for startPosition in range(n):  
 result = [] # result배열 초기화  
 visited = [False] \* n # visited 방문처리 확인하는 배열 False로 초기화 -> 방문처리될 경우 True  
 for i in range(startPosition): # 0부터 n까지 시작위치  
 visited[i] = True # 시작위치는 함수 방문하기 전에 방문한것으로 표시  
 dfs(startPosition, n, arr, visited) # 깊이 우선 탐색 함수  
 if len(result) != 1: # bfs 함수를 1번이상 돌경우 배열의 길이는 1이상  
 result\_num[startPosition] = result[-1] # 시작위치의 가장 많은 간 셀 값을 반환  
 else: # bfs함수를 안돌 경우 result 배열의 길이는 0  
 result\_num[startPosition] = 0  
 #print("최종답안",result\_num)  
  
if has\_duplicates(result\_num) is False: # list 중복되는 값 없을 경우-> False  
 print(result\_num.index(max(result\_num)))  
elif has\_duplicates(result\_num) is True: # list에 중복 되는 값 있을 경우-> True  
 pos = []  
 for i in range(len(result\_num)):  
 if result\_num[i] == max(result\_num): # 최대값일 경우 append  
 pos.append(i)  
 for i in pos:  
 print(i, end=' ') # 한 줄로 print

* **문제 2-1번**

m×n 크기(m은 행의 개수로 500이하 양의 정수, n은 열의 개수로 500이하 양의 정수)의 배열로 표현되는 미로가 있다. **1은 갈 수 있는 곳을 나타내고, 0은 갈 수 없는 곳을 나타낸다.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

(2-1) 미로의 가장 위에서 가장 아래로 갈 수 있는 경로가 존재하는지를 판별하는 프로그램을 작성하시오. 미로의 각 위치에서 상하좌우로 인접한 곳으로만 갈 수 있다. 가는 길이 있으면 1을 출력하고 없으면 –1을 출력한다.

**요구조건: 깊이우선탐색을 이용한다.**

입력 예

6 10 // m n은 각각 2이상 500이하 정수

0110000011

1101111101

1101010111

1111010111

0100111000

1011110111

출력 예

(1) 1

(2) 12

* **알고리즘 및 자료구조**

DFS(Depth First Search) 보통 stack을 이용해서 갈 수 있는 만큼 최대한 깊이 가고, 더 이상 갈 곳이 없을 경우 이전 정점으로 돌아간다는 방식이다.

DFS는 stack과 recursive function을 이용하여 구현한다. 2-1번 문제는 class을 활용한 recursive fucntion을 이용 하여 구현 했습니다.

class maze에서 행과 열 좌표를 반환해주는 클래스 구현

dfs\_maze함수에서 좌표를 받아서 위쪽, 아래쪽, 왼쪽, 오른쪽을 검사해서 있을 경우 1을 대입하는 동시에 visit\_road하는 행렬을 생성해서 False일경우 방문처리 안된 좌표이고 True일경우 방문처리 된 좌표인 행렬을 구성해 하나씩 검사하는 함수

이를 입력받은 n만큼 dfs\_maze()함수를 돌려서 결과값은 리턴받음.

* **시간복잡도 분석**

DFS함수 한 번에 for loop를 n 만큼 돌음 -> O(n)

정점을 방문할 때 마다 DFS호출, n개의 정점을 모두 방문하므로

DFS의 전체 시간 복잡도는 O(n^2)

* **느낀 점**

처음 문제를 읽고 class를 이용해서 좌표를 이용한 문제를 풀었던 경험을 바탕으로 바로 구현 할 수 있었습니다.

* **프로그램 코드**
* import sys  
  sys.setrecursionlimit(10\*\*7)  
    
  class maze:  
   def \_\_init\_\_(self, x=0, y=0):  
   self.x = x  
   self.y = y  
    
  m, n = map(int, input().split()) # m \* n 크기 행렬 수 입력  
  maze\_map = [[0 for \_ in range(n)] for \_ in range(m)] # 미로  
  visit\_road = [[False for \_ in range(n)] for \_ in range(m)] # True는 방문 False는 미방문  
  # visit\_road: 미로의 행과 열의 각 위치가 방문되었는지 아닌지를 확인하기 위한 2차원 배열  
  for i in range(m):  
   x = input() # 01입력 받아서  
   maze\_map[i] = list(map(int, list(str(x)))) # maze\_map 행렬에 대입  
    
  # map: 미로 정보를 저장하는 2차원 배열  
  # 갈 수 있는 곳 1, 없는 곳 0  
  def bfs\_maze(map, n, m, start, dest, visit\_road): # start, dest: 출발지와 목적지  
   next = maze() # class maze에서 행열 대입  
   if map[start.x][start.y] == 1: # 시작점이 1이라면  
   visit\_road[start.x][start.y] = True # 시작 위치를 방문 했음 -> True  
   else: # 방문안했으면 -1  
   return False # 종료  
   if start.x == dest.x: # 시작 좌표가 목적지와 같다면 1 리턴  
   return True # 종료  
   if start.x > 0: # 좌표에서 위쪽 이동 할 경우  
   if map[start.x-1][start.y] == 1 and not visit\_road[start.x - 1][start.y]:  
   # 현재 좌료에서 위 행 이동 또는 방문처리 안된 경우  
   next.x = start.x - 1 # 다음행 = 행 - 1  
   next.y = start.y # 다음행 = 열 + 0  
   if bfs\_maze(map, n, m, next, dest, visit\_road): # 함수실행가능 -> True일 때  
   return True  
   if start.y > 0: # 좌표에서 왼쪽 이동 할 경우  
   if map[start.x][start.y-1] == 1 and not visit\_road[start.x][start.y - 1]:  
   # 현재 좌료에서 왼쪽 열 이동 또는 방문처리 안된 경우  
   next.x = start.x # 다음행 = 행 + 0  
   next.y = start.y - 1 # 다음행 = 열 - 1  
   if bfs\_maze(map, n, m, next, dest, visit\_road): # 함수실행가능 -> True일 때  
   return True  
   if start.y < n-1: # 좌표에서 오른쪽 이동 할 경우  
   if map[start.x][start.y+1] == 1 and not visit\_road[start.x][start.y + 1]:  
   # 현재 좌료에서 오른쪽 열 이동 또는 방문처리 안된 경우  
   next.x = start.x # 다음행 = 행 + 0  
   next.y = start.y + 1 # 다음 열 = 열 +1  
   if bfs\_maze(map, n, m, next, dest, visit\_road): # 함수실행가능 -> True일 때  
   return True # True반환 종료  
   if start.x < m-1: # 좌표에서 아래 이동 할 경우  
   if map[start.x+1][start.y] == 1 and not visit\_road[start.x + 1][start.y]:  
   # 현재 좌료에서 아래 행 이동 또는 방문처리 안된 경우  
   next.x = start.x + 1 # 다음행 = 행 + 1  
   next.y = start.y # 다음 열 = 열 +0  
   if bfs\_maze(map, n, m, next, dest, visit\_road):  
   return True  
   return False # 동서남북 이동 불가 -> False  
    
  for i in range(n):  
   result = bfs\_maze(maze\_map, n, m, maze(0, i), maze(m - 1, None), visit\_road)  
   if result:  
   break  
    
  if result: # result True, False 검사  
   print(1) # 가는길 있을 경우  
  else:  
   print(-1) # 가는 길 없을 경우
* **문제 2-2번**

m×n 크기(m은 행의 개수로 500이하 양의 정수, n은 열의 개수로 500이하 양의 정수)의 배열로 표현되는 미로가 있다. **1은 갈 수 있는 곳을 나타내고, 0은 갈 수 없는 곳을 나타낸다.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

(2-2) 미로의 가장 위에서 가장 아래로 내려가는 최단 경로의 길이 (지나는 1의 개수)를 구하는 프로그램을 작성하시오. 미로의 각 위치에서 상하좌우로 인접한 곳으로만 갈 수 있다. 가는 길이 없으면 -1을 출력한다.

**요구조건: 너비우선탐색을 이용한다.**

* **알고리즘 및 자료구조**

BFS(Breadth First Search) 기본 작동 방식은 queue를 이용하여 지금 위치에서 갈 수 있는 것들을 모두 큐에 넣는 방식이다. 이때, queue에 넣을 시점에 해당 노드를 방문했다고 체크해야 됩니다,

BFS는 queue와 while loop을 이용하여 구현한다.

2-1번 문제는 class을 활용한 recursive fucntion을 이용하여 구현 했습니다.

class maze에서 행과 열 좌표를 반환해주는 클래스 구현

bfs\_maze함수에서 좌표를 받아서 위쪽, 아래쪽, 왼쪽, 오른쪽을 검사해서 있을 경우 1을 대입하는 동시에 visit\_road하는 행렬을 생성해서 False일경우 방문처리 안된 좌표이고 True일경우 방문처리 된 좌표인 행렬을 구성해 하나씩 검사하는 함수

이를 입력받은 n만큼 bfs\_maze()함수를 돌려서 결과값은 리턴받음.

* **시간복잡도 분석**

bfs함수를 돌기 전에 i가 n만큼 열 개수를 검사합니다. 열의 원소에서 0인지 1인지 구분합니다. 1일 경우 bfs함수를 돕니다. -> O(n)

bfs함수안에서 dx와 dy 동서남북 방향에 대해서 갈 수 있는지 없는지 검사합니다. -> O(4)

while문에서 n+m번 만큼 돌게 됩니다.

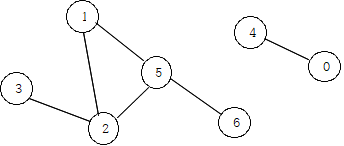
모두 곱하면 시간 복잡도는 O((n+m)\*n)입니다.

* **느낀 점**

테스트 케이스 11번부터 20까지의 timeout으로 시간 복잡도를 O(n\*m)으로 고쳐보려고 했지만 visit\_arr변수를 bfs함수를 돌때마다 초기화 하지 않을 경우 계속 답이 틀려서 해결을 못하고 제출해서 아쉬웠습니다.

* **프로그램 코드**
* from collections import deque  
  class maze:  
   def \_\_init\_\_(self, x=0, y=0):  
   self.x = x  
   self.y = y  
   self.visit\_arr = [[False] \* n for \_ in range(m)] # True는 방문 False는 미방문  
    
  m, n = map(int, input().split()) # m \* n 크기 행렬 수 입력  
  maze\_map = [[0 for \_ in range(n)] for \_ in range(m)] # 미로  
  road = [[0 for \_ in range(n)] for \_ in range(m)] # 01도로 입력  
  # visit\_arr = [[False for \_ in range(n)] for \_ in range(m)] # True는 방문 False는 미방문  
  dx = [1, 0, -1, 0]  
  dy = [0, 1, 0, -1]  
  for i in range(m):  
   x = input() # 01입력 받아서  
   maze\_map[i] = list(map(int, list(str(x)))) # maze\_map 행렬에 대입  
  queue = deque() # 너비우선탐색은 큐 이용  
    
  def bfs(maze\_map, start, dest): # 탐색 방법 : 너비우선탐색  
   next = maze()  
   # start.x, start.y i, 0  
   # visit\_arr = [[False] \* n for \_ in range(m)] # True는 방문 False는 미방문  
   if maze\_map[0][start.x] == 1: # 시작점이 1이라면  
   # visit\_arr[start.x][start.y] = True  
   queue.append([start.x, start.y]) # 큐에 시작 좌표 append  
   road[start.y][start.x] = 1 # 시작 좌표 방문했으므로 1로 처리  
   else:  
   return False  
   # if start.x == dest.x: # 시작 좌표가 목적지와 같다면 1 리턴  
   # return True # 종료  
   while queue:  
   start.x, start.y = queue.popleft() # row와 col에 maze\_map[0][i]를 넣음  
   for i in range(4): # 좌표계 위아래양옆을 동서남북으로 생각  
   # 0 동, 1 북, 2서, 3남  
   next.x = start.x + dx[i] # 이동한 x 좌표  
   next.y = start.y + dy[i] # 이동한 y 좌표  
   if 0 <= next.x < n and 0 <= next.y < m and not next.visit\_arr[next.y][next.x]:  
   # 만약 이동한 좌표가 입력 한 m\*n안에 있고 visit\_arr에 방문처리가 안되어있는 좌표라면  
   if maze\_map[next.y][next.x] == 1:  
   next.visit\_arr[next.y][next.x] = True # 방문처리  
   queue.append([next.x, next.y]) # 새로운 좌표 큐에 append  
   road[next.y][next.x] = road[start.y][start.x] + 1 # 위에서 방문처리한 그 지점까지의 거리를 원래 거리 +1  
   if next.y == dest.y: # 좌표에서 위쪽 이동 할 경우,y좌표가 m-1일 경우는 맨 밑줄 위치  
   queue.clear() # 큐 비우고  
   return road[next.y][next.x] # 거리값 리턴  
    
    
  new\_list = []  
  for i in range(n):  
    
   new\_list.append(bfs(maze\_map, maze(i, 0), maze(None, m-1)))  
  res = list(filter(None, new\_list)) # bfs함수를 돌은 거리값들을 새로운 결과변수에 filter 함수 이용해서 정리  
  # print(res)  
  if len(res) == 0: # res가 0이라면 갈수 없는 좌표이다.  
   print(-1) # 가는 길이 없는 경우 -1출력  
  else:  
   print(min(res)) # 가는 길이 있으면 최소 결과값 출력
* **문제 3-1번**

0부터 n-1까지 번호가 붙여진 n명의 사람들의 친구관계가 주어져 있다. 다음은 6명의 사람들에 대한 친구관계를 보여준다.



친구관계로 연결되어 있는 집단의 개수와 가장 큰 집단의 크기 (사람 수), 가장 작은 집단의 크기를 구하는 프로그램을 작성하시오. 위의 그림에서 집단은 2개이고 가장 큰 집단은 {1, 2, 3, 5, 6}으로서 크기는 5이고 가장 작은 집단은 {0, 4}로서 크기는 2이이다.

**(3-1) 요구조건: 친구관계를 인접행렬로 표현하고, 탐색방법은 깊이우선탐색을 이용한다.**

첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(**0 이상 300,000 이하 정수**)가 주어진다. 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.

출력

첫 번째 줄에 친구관계로 연결되어 있는 집단의 개수를 출력한다.

두 번째 줄에 가장 큰 집단의 크기 (사람 수), 가장 작은 집단의 크기를 출력한다.

입력 예

7 6

1 2

1 5

5 6

5 2

0 4

2 3

출력 예

2

5 2

* **알고리즘 및 자료구조**

인접행렬로 만들고 탐색방법은 DFS방법으로 도는 알고리즘의 문제입니다.

인접 행렬이란, 이름 그대로 그래프의 연결 관계를 행렬로 표현하여 이차원 배열로 나타내는 방식입니다.

인접 행렬 arr[i][j]는 노드 i에서 j로 가는 간선이 존재할 경우 1, 아니면 0로 정의 할 수 있습니다.

무향 그래프의 경우 j에서 i로 가는 길이 존재하면 i에서 j로 가는 길 또한 존재합니다.

이러한 특성으로 인접행렬을 구현할 경우, 대각 성분을 기준으로 대칭인 성질을 갖게 됩니다,

첫번째로 n, m을 입력받습니다.

arr이라는 2차원 배열을 만들고 for 구문을 통해 번호를 입력받아 arr에 append

dfs는 위의 문제에서 설명했듯이 이번에는 재귀적 방법이 아닌 stack을 이용해서 풀었습니다.

dfs함수에서 시작 위치와 방문한 사람들의 행렬을 받고 stack을 이용해 dfs를 구현하였고

방문수와 사람들을 리턴합니다.

마지막으로 사람들 과의 관계수와 집단의 크기를 구해서 print합니다.

* **시간복잡도 분석**

인접 행렬로 구현했으므로 시간 복잡도는 O(n^2)입니다.

dfs함수를 돌기전에 입력받은 n만큰 for 구문을 돕니다. -> O(n)

시작위치에서 마지막 위치까지 방문했는지 안했는지 검사합니다. 이는 입력받은 m만큼 돌게 됩니다, -> O(m)

최종적으로 시간복잡도는 O(n^2)입니다.

* **느낀 점**

인접 행렬과 인접리스트를 파이썬으로 어떻게 구현하는지에 대해 알게 되었습니다.

* **프로그램 코드**
* from sys import stdin  
  # 행열 n, m 입력받기  
  # 첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(0 이상 300,000 이하 정수)가 주어진다.  
  n, m = map(int, stdin.readline().split())  
  # 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.  
  arr = [[0 for \_ in range(n)] for \_ in range(n)]  
  for \_ in range(m):  
   hang, yeol = map(int, stdin.readline().split()) # 번호 입력받고 arr에 append  
   arr[yeol].append(hang)  
   arr[hang].append(yeol)  
  # 현재 arr는 친구관계 인접행렬로 표현  
    
  def dfs(start, arr\_visit): # 탐색방법 : 깊이우선탐색  
   people = 0 # 사람들  
   stack = [start] # 스택 이용  
   while stack: # 스택을 끝까지 도는 동안  
   start = stack.pop() # 첫 번째 시작값을 스택에 잇을 경우 pop해줌  
   if start not in arr\_visit: # 만약 arr\_visit에 방문하지 않았을 경우 방문처리  
   arr\_visit.append(start) # arr\_visit에 start append  
   people += 1 # people변수에 사람들과 관계로 연결되어있으므로 people +1  
   for x in arr[start]: # arr에 관계 시작부분이 있을 경우 append  
   stack.append(x) # stack에 append  
   return arr\_visit, people # 방문수, 사람들 리턴  
    
  visited\_arr = [] # 방문한 사람 행렬  
  relship = 0 # 사람들과 연결되어 있는 집단의 수  
  ans = [] # 연결 되어있는 각 집단의 크기가 들어가있는 변수  
  if m !=0: # m이 0인 경우가 0이 아닌 경우로 나눔  
   for i in range(n): # 입력 받은 n만큼 for구문 돌음  
   visited\_arr, people = dfs(i, visited\_arr) # 깊이 우선 탐색 돌음  
   if people != 0: # 만약 people이 0이 아닐 경우 관계수는 무조건 하나 이상 있음  
   relship += 1  
   ans.append(people)  
  else: # 0인 경우  
   people = 1 # 집단을 이루는 사람들은 무조건 1  
   relship = n # 집단은 n  
   ans.append(people)  
  print(relship) # 관계수  
  print(max(ans), min(ans)) # 가장 큰 집단의 크기, 가장 작은 집단의 크기를 출력하는 변수
* **문제 3-2번**

**(3-2) 요구조건: 친구관계를 인접리스트(연결리스트)로 표현하고, 탐색방법은 깊이우선탐색을 이용한다.**

첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(**0 이상 300,000 이하 정수**)가 주어진다. 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.

출력

첫 번째 줄에 친구관계로 연결되어 있는 집단의 개수를 출력한다.

두 번째 줄에 가장 큰 집단의 크기 (사람 수), 가장 작은 집단의 크기를 출력한다.

* **알고리즘 및 자료구조**

인접리스트와 DFS를 이용한 탐색방법으로 구하는 문제입니다.

인접리스트는 각각의 노드에 연결된 노드들을 원소로 갖는 리스트들의 배열을 의미한다.

Arr[i] : i번째 노드에 연결된 노드들을 원소로 갖는 리스트 입니다.

인접리스트 또한 무향 그래프의 경우에는 본인 노드 인덱스의 리스트 내의 서로를 원소로 갖게 됩니다.

인접 리스트를 활용하여 그래프의 연결 관계를 저장할 경우, 인접 리스트는 인접 행렬과 달리 실제로 연결된 노드에 대한 정보만 저장하기 때문에, 모든 벡터들의 원소의 개수의 합이 간선의 개수와 동일하다는 점입니다. 즉, 간선의 개수에 비례하는 메모리만 차지하여 구현이 가능하다.

Arr = [[] for \_ in range(n)]으로 인접 리스트를 구현하였습니다. 3-1 문제에서 DFS함수의 방식을 그대로 이용하였습니다.

* **시간복잡도 분석**

인접리스트로 구현했으므로 시간 복잡도는 O(n+m)입니다.

dfs함수를 돌기전에 입력받은 n만큰 for 구문을 돕니다. -> O(n)

시작위치에서 마지막 위치까지 방문했는지 안했는지 검사합니다. 이는 입력받은 m만큼 돌게 됩니다, -> O(m)

최종적으로 시간복잡도는 O(n+m)입니다.

* **느낀 점**

인접 행렬과 인접리스트를 파이썬으로 어떻게 구현하는지에 대해 알게 되었습니다

* **프로그램 코드**
* from sys import stdin  
  # 행열 n, m 입력받기  
  # 첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(0 이상 300,000 이하 정수)가 주어진다.  
  n, m = map(int, stdin.readline().split())  
  # 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.  
  arr = [[] for \_ in range(n)]  
  for \_ in range(m):  
   hang, yeol = map(int, stdin.readline().split()) # 번호 입력받고 arr에 append  
   arr[yeol].append(hang)  
   arr[hang].append(yeol)  
  # 현재 arr는 친구관계 인접리스트(연결리스트)로 표현  
  def dfs(start, arr\_visit): # 탐색방법 : 깊이우선탐색  
   people = 0 # 사람들  
   stack = [start] # 스택 이용  
   while stack: # 스택을 끝까지 도는 동안  
   start = stack.pop() # 첫 번째 시작값을 스택에 잇을 경우 pop해줌  
   if start not in arr\_visit: # 만약 arr\_visit에 방문하지 않았을 경우 방문처리  
   arr\_visit.append(start) # arr\_visit에 start append  
   people += 1 # people변수에 사람들과 관계로 연결되어있으므로 people +1  
   for x in arr[start]: # arr에 관계 시작부분이 있을 경우 append  
   stack.append(x) # stack에 append  
   return arr\_visit, people # 방문수, 사람들 리턴  
    
  visited\_arr = [] # 방문한 사람 행렬  
  relship = 0 # 사람들과 연결되어 있는 집단의 수  
  ans = [] # 연결 되어있는 각 집단의 크기가 들어가있는 변수  
  if m !=0: # m이 0인 경우가 0이 아닌 경우로 나눔  
   for i in range(n): # 입력 받은 n만큼 for구문 돌음  
   visited\_arr, people = dfs(i, visited\_arr) # 깊이 우선 탐색 돌음  
   if people != 0: # 만약 people이 0이 아닐 경우 관계수는 무조건 하나 이상 있음  
   relship += 1  
   ans.append(people)  
  else: # 0인 경우  
   people = 1 # 집단을 이루는 사람들은 무조건 1  
   relship = n # 집단은 n  
   ans.append(people)  
  print(relship) # 관계수  
  print(max(ans), min(ans)) # 가장 큰 집단의 크기, 가장 작은 집단의 크기를 출력하는 변수
* **문제 3-3번**

**(3-3) 요구조건: 친구관계를 인접리스트(연결리스트)로 표현하고, 탐색방법은 너비우선탐색을 이용한다.**

첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(**0 이상 300,000 이하 정수**)가 주어진다. 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.

출력

첫 번째 줄에 친구관계로 연결되어 있는 집단의 개수를 출력한다.

두 번째 줄에 가장 큰 집단의 크기 (사람 수), 가장 작은 집단의 크기를 출력한다.

* **알고리즘 및 자료구조**

3번째 문제는 인접리스트와 BFS의 탐색방법으로 해결하는 문제입니다. 인접리스트는 3-2문제에서 그대로 사용했으니 설명은 생략하겠습니다.

Visited\_arr = []리스트는 방문했던 노드들의 목록을 순차적으로 저장할 리스트입니다.

Queue = deque()는 노드의 목록을 차례대로 저장하기 위한 변수입니다.

**queue.append(start)**

탐색을 시작할 시작 노드를 queue에 append합니다,

Queue를 전부 순회할 때까지 (탐색하지 못한 노드가 없을때까지)반복을 진행합니다.

Queue에서 제일 앞에 있는 node를 popleft()메서드를 통해 node 변수로 꺼내옵니다.

Pop(0)메서드르 사용 안하는 이유는 list자료구조를 써서 제일 앞에 있는 노드를 가져오기 위해서는 시간 복잡도가 O(N)으로 매우 비효율적이기 때문입니다. 그 이유는 제일 앞 원소를 pop한 뒤에 이후에 있는 원소들을 모두 한 칸씩 땡겨줘야 하기 때문입니다.

Deque 자료구조에 들어있는 원소중 제일 앞에 있는 노드를 꺼내서 해당 노드가 visited\_arr배열에 있는지 확인합니다.

없다면 append해주고 방문 수와 사람들을 리턴합니다.

* **시간복잡도 분석**

bfs함수를 돌기 전에 i가 n만큼 열 개수를 검사합니다. 열의 원소에서 0인지 1인지 구분합니다. 1일 경우 bfs함수를 돕니다. 이는 while문에서 n+m번 만큼 수행합니다. -> O(n+m)

bfs함수안에서 popleft를 이용하므로 시간 복잡도는 O(n+m)입니다.

* **느낀 점**

인접 행렬과 인접리스트를 파이썬으로 어떻게 구현하는지에 대해 알게 되었습니다. 또한 1-1번과 2-2번에서의 너비우선탐색(bfs)를 testcase fail로 인해 힘들었지만 3-3번은 운좋게 제가 설계한대로 프로그램이 잘 실행 되었습니다.

* **프로그램 코드**
* from collections import deque  
  from sys import stdin  
  # 행열 n, m 입력받기  
  # 첫 번째 줄에 사람들의 수 n(1이상 1,000이하 정수)와 친구 관계 수 m(0 이상 300,000 이하 정수)가 주어진다.  
  n, m = map(int, stdin.readline().split())  
  # 다음 m개의 각 줄에 친구 관계를 나타내는 두 사람의 번호가 주어진다.  
  arr = [[] for \_ in range(n)]  
  for \_ in range(m):  
   hang, yeol = map(int, stdin.readline().split()) # 번호 입력받고 arr에 append  
   arr[yeol].append(hang)  
   arr[hang].append(yeol)  
  # 현재 arr는 친구관계 인접리스트(연결리스트)로 표현  
    
  def bfs(start, arr\_visit): # 탐색 방법 : 너비 우선 탐색  
   people = 0 # 사람들  
   queue =deque() # 너비 우선 탐색에서는 큐 이용  
   queue.append(start)  
   while queue:  
   start = queue.popleft() # 첫 번째 시작값이 큐에 잇을 경우 pop해줌  
   if start not in arr\_visit: # 만약 arr\_visit에 방문하지 않았을 경우 방문처리  
   arr\_visit.append(start) # arr\_visit에 start append  
   people += 1 # people변수에 사람들과 관계로 연결되어있으므로 people +1  
   for w in arr[start]: # arr에 관계 시작부분이 있을 경우 append  
   queue.append(w) # stack에 append  
   return arr\_visit, people # 방문수, 사람들 리턴  
  visited\_arr = [] # 방문한 사람 행렬  
  relship = 0 # 사람들과 연결되어 있는 집단의 수  
  ans = [] # 연결 되어있는 각 집단의 크기가 들어가있는 변수  
  for i in range(n): # 입력 받은 n만큼 for구문 돌음  
   visited\_arr, people = bfs(i, visited\_arr) # 깊이 우선 탐색 돌음  
   if people != 0: # 만약 people이 0이 아닐 경우 관계수는 무조건 하나 이상 있음  
   relship += 1  
   ans.append(people)  
  print(relship) # 관계수  
  print(max(ans), min(ans)) # 가장 큰 집단의 크기, 가장 작은 집단의 크기를 출력하는 변수