

6205-HW9

HackerRank Prepare > Data Structures > Disjoint Set > Merging Communities

Problem
People connect with each other in a social network. A connection between Person i and Person j is represented as Mij . When two persons belonging to different communities connect, the net effect is the merge the communities which i and j belong to.
At the beginning, there are n people representing n communities. Suppose person 1 and 2 connected and later 2 and 3 connected, then 1, 2 and 3 will belong to the same community.
There are two types of queries:
1. $Mij \implies$ communities containing persons i and j are merged if they belong to different communities.
2. $Qi \implies$ print the size of the community to which person i belongs.

Input Format
The first line of input contains 2 space-separated integers n and q the number of people and the number of queries.
The next q lines will contain the queries.

Constraints
 $1 \leq n \leq 10^5$
 $1 \leq q \leq 2 \times 10^5$

Output Format
The output of the queries.

Sample Input
STDIN Function
3 6 n = 3, q = 6 Q 1 print the size of the community containing person 1 M 1 2 merge the communities containing persons 1 and 2 ... Q 2 M 2 3 Q 3 Q 2

Sample Output
1
2
3
3

Explanation
Initial size of each of the community is 1.

```
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
# NOTHING CAN BE CHANGED
if (self._n < 20):
    u = util()
    u.print_index(self._n)
    u.print_list(self._id)
    print("U = ",self._numUnion,"F = ",self._numFind, "H = ",self._maxHeight) ;

def H(self):
    # NOTHING CAN BE CHANGED
    return self._maxHeight

#####
# NOTHING CAN BE CHANGED BELOW
# At the end Find the max height
# Find the owner of each object
# The max HOP will be recorded in self._maxHeight
#####
def max_height(self)->int:
    for i in range(self._n):
        self.fi(i)
```

Upload Code as File ☐ Test against custom input Run Code Submit Code

Congratulations
You solved this challenge. Would you like to challenge your friends? [f](#) [t](#) [m](#) [Next Challenge](#)

Test case 3 ☒ Test case 4 ☒ Test case 5 ☒ Test case 6 ☒ Test case 7 ☒ Test case 8 ☒ Test case 9 ☒

Expected Output
1 1
2 2
3 3
4 3

```
def max_height(self)->int:
```

Time complexity

for the `max_height` method, since we need to iterate all the elements, and for each element, the worst case is that it needs to hop `max_height` times to find the parent node. Therefore, time complexity is $O(n \cdot \text{max_height})$

Space complexity

the space complexity is $\theta(n)$

```
def num(self,a:'int')->'int':
```

Time complexity

for the `num` method, the time complexity is $O(\text{max_height})$, since the `max_height` ≤ 6 , we can say that the time complexity is a constant

Space complexity

the space complexity is a constant

```
def F1(self,a:'int')->'int':
```

for the F1 method, since the time complexity is the hop times, which is $O(\text{max_height})$, we can say the time complexity is a constant

space complexity is a constant

```
def F2(self,a:'int',b:'int')->'bool':
```

for the F2 method, the time complexity is the same as the F1 method, therefore the time complexity is a constant, and the space complexity is a constant

```
def U(self,a:'int',b:'int')->'bool':
```

for the U method, the time complexity is the same as F2 as the comparison following has complexity of a constant. therefore the time complexity is a constant, and the space complexity is a constant.

In conclusion, the time complexity of the SUSF function is:

$O(n \cdot \text{max_height}) + O(\text{max_height}) + O(\text{max_height}) + C + C = O(n \cdot \text{max_height})$

Since max_height is no larger than 6, therefore the time complexity is $O(n)$

the space complexity is $\Theta(n)$