Project 1 algo eng

Note

Need to find a leader or a parent | something to have another pair with

```
function find(a):
        if parent[a] != a:
                parent[a] = find(a)
        return parent[a]
```

Summary

A recursive function in order to find a leader or a origin to the first pair to later find the next one for it.

Note

We need a function in order to fing the leaders of a group such that a and b belong to. If they don't match each other then we can merge them. By doing this we can make the rootA a child of rootY

```
def union(a, y):
        rootA = find(a)
        rootB = find(b)
        if rootA != rootB:
                parent[rootA] = rootB
```

Note

We need to declare the size of our row. Such as n. and we need to set a parent list. This parent list will store such that each person has it's own parent.

```
n = len(row) // 2 # this is so the len = 6 / 2 = 3 couples
parent[i for i in range(n)]
```

Note

We then need to iterate through the rows but we need to so in pairs. as we find the pairs we union them.

```
for i in range(0, len(row), 2):
        union(row[i] // 2, row[i+1] // 2
```

Summary

row[i] // 2 = 0 / 2 = 0
row[i+1] // 2 = 2/2 = 0 ##This is assuming that the second index of the row list is 1

Note

Then we want to count how many "people" are their own. then this shows the number of couples are there.. We then can calculate the total number of swaps needed

```
count = sum([1 for i, a in enumerate(parent) if i == find(a)])
return n - count
```

Summary

1 for i essentially builds a list of 1's bu only for the elements where the index of i is equal to find(a). it adds a 1 to the list for each element that is its own representative.

a in enumerate essentially is an iterator that is 0,0 or 1, 1 or 2, 3. The i is the index and the a is the value.

when comparing to i == find(a)
if we are assuming the parent is [0, 1, 2, 0, 1]

find(a):
find(0) -> 0
find(1) -> 1
find(2) -> 2
find(3) -> 0
find(4) -> 1

Code Section.	Description	Time complexity	Space Complexity

def find(a):
  if root[a] != a:
    root [a] = find(root[a])
  return root[a]
	A recursive function in order to find a leader or a origin to the first pair to later find the next one for it.	O(α(n)) amortized	O(log n)

def union(a, b):
  rootA = find(a)
  rootB = find(b)
  if rootA != rootB:
parent[rootA] = rootB
	A function to find a the leaders of a group that x and y belong to, if they don't match we merge them	**O(α(n))** (amortized)	O(1)

n = len(row) // 2	n will be the length of the row divided by 2 in order to find how many couples we have in the list	O(1)	O(1)
parent = [i for i in range(n)]	Initializes an array where each parent elements is its own parent	O(n)	O(n)
for i in range(0, len(row), 2): union(row[i] // 2, row [i + 1] //	Looping through couples whilst making them pairs and then	O(n * α(n))	O(1) per iteration

Code Section.	Description	Time complexity	Space Complexity
2)	performs a union to pair them.		
count = sum([1 for i, a in enumerate(parent) if i == find(a)])	Counts the numbers of disjoints (non couple) by checking each element of its own parent	O(n * α(n))	O(1) per iteration
return n - count	returning the final statement to justify how many swaps have to be made to pair all the couples	O(1)	O(1)

Calculating the Big O efficiency class. using step counts

find(a)

```
function find(a):
        if root[a] != a:
                root[a] = find(a)
        return root[a]
```

1. if parent[a] != a, we call find(root(a))
    1. When we call the find(a) it will check root[a] != a every time
    2. This is a recursive call until we find the root
2. return root[a]
    1. 1 step

It all really depends on the how close the desired number is so O(n)

Big O: O(α(n))

Since it is an amortized due to recursive and the legnth of the root

---

union(a, b)

```
def union(a, b):
        rootA = find(a)
        rootB = find(b)
        if rootA != rootB:
                parent[rootA] = rootB
```

1. rootA = find(a)
    1. since it is calling find(a) | we know that find(a) is $O(\alpha(n))$
2. rootB = find(a)
    1. same goes here | $O(\alpha(n))$
3. if rootA != rootB:
    1. 1 step
4. parent[rootA] = rootB
    1. 1 step

$O(\alpha(n))$

since the steps for finding roots is $O(\alpha(n)) + O(\alpha(n)) = O(2\alpha(n)) = O(\alpha(n))$

Steps are 1 + 1 = 2

---

total is
$O(\alpha(n)) + 1 + 1 = O(\alpha(n))$

---

n = len(row) // 2

```
n = len(row) // 2
```

1. len(row)

    1. finding length of a list is O(1)
2. // 2
    1. basic arithmetic operation so O(1)
3. This is 2 step

O(1)

Since both of these produce only 2 steps

1 + 1 = 2 => O(1) which is constant time

---

parent = [i for i in range(n)]

```
parent = [i for i in range(n)]
```

1. range(n)
    1. sets up an iterable number so O(1) | doesn't make a list yet
2. i for i in range(n)
    1. iterating n times to get each integer from 0 to n - 1
    2. then, for each integer i, adding i to the new list
    3. this loops runs n times

O(n)

Total steps = 1 + n = n + 1 => O(n)

---

for i in range(0, len(row), 2): union(row[i] // 2, row[i+1] // 2

1. range(0, len(row), 2)
    1. as this creates an iterator from 0 to len(row) with a step of 2
    2. this is just O(1)
2. Since this is a pair, we do run the len(row) two times since we take a two step
    1. the input list is 2n
    2. loop increments i by 2 each time resulting in n iteration
    3. the loop runs n time
3. inside the loop | union row[i] // 2
    1. calls find and that is like rootA = find(row[i] // 2)
    2. we know that find = O(α(n))
4. same goes for the row[i + 1] // 2
    1. calls find rootB = find(row[i + 1] // 2)
    2. we know that find = O(α(n))

O(n * α(n))

Total steps = O(α(n)) + O(α(n)) + 1 + 1 = O(2α(n)) + 2 = O(α(n))

---

entire loop
total steps including the for loop

total = n *O(α(n))* = *O(n* α(n))

---

count = sum([1 for i, a in enumerate(parent) if i == find(a)])

```
count = sum([1 for i, a in enumerate(parent) if i == find(a)])
```

1. enumerate(parent)
    1. creates numerator which is O(1)
2. 1 for i, a in enumerate(parent) if i == find(a)
    1. iterates through parent

    2. if the i == find(a) | we run for n iterations
3. The i == find(a) calls find(a)
    1. we know that find(a) = O(α(n))
4. adding to the list
    1. when condition is met, we add 1 to the list
    2. O(1)

O(α(n))

total steps: O(α(n)) + 1 + 1 = O(α(n)) + 2 = O(α(n))

for the complete line of code
O(n *α(n)) + O(n) = O(n* α(n))

This is because of the for loop that we have.

---

return n - count

```
return n - count
```

1. Basic arithmetic operation
    1. O(1) time | 1 step

O(1)

Total steps: 1 = O(1)

Overall Complexity

Total time complexity: $O(1) + O(n) + O(n\ \alpha(n)) + O(n\ \alpha(n)) + O(1) = O(n * \alpha(n))$