Program Structures and Algorithms

Assignment - 4

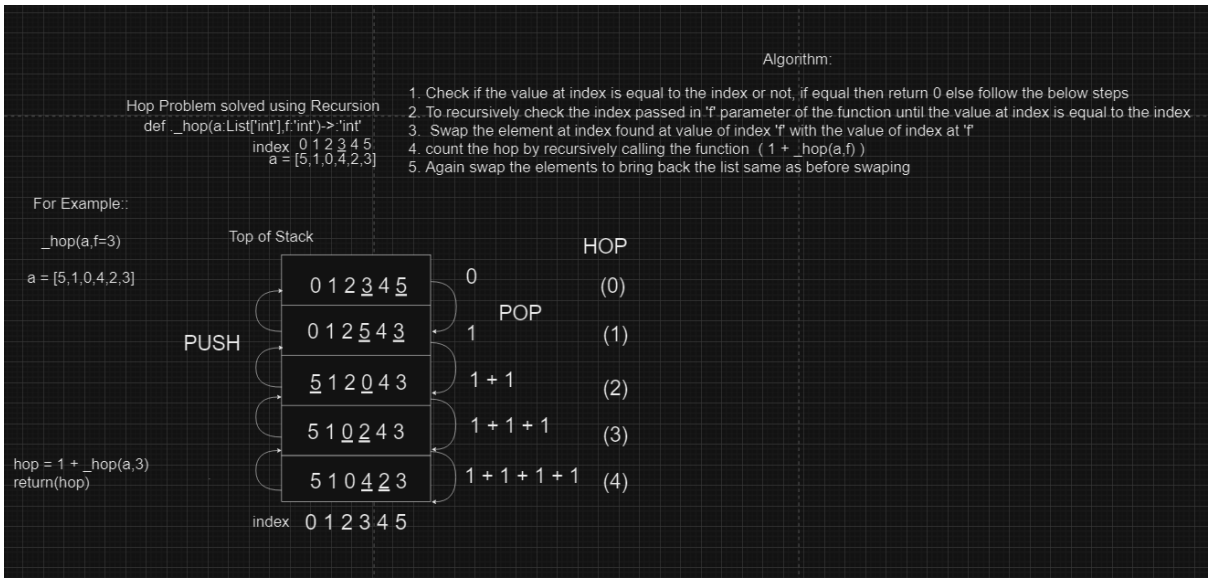

Figure -1

```
##########################################
def _hop(self, a:List[int],f:'int') -> 'int':
  ## n = len(a)  ##YOU CANNOT CALL len
  if (a[f] == f):
    return 0
  else:
    val1 = a[f] #val1 = 4
    val2 = a[val1] #val2 = 2
    a[f] = val2
    a[val1] = val1
    #print(a)
    hop = 1 + self._hop(a,f)
    a[f] = val1
    a[val1] = val2
    #print(a)
    return(hop)
```

Algorithm:

1. Check if the value at index is equal to the index or not, if equal then return 0 else follow the below steps
2. Swap the element at index found at value of index 'f' with the value of index at 'f'
3. Recursively check the index passed in 'f' parameter of the function until the value at index is equal to the index and count the hop by recursively calling the function (1 + _hop(a,f))

4. Again swap the elements to bring back the list same as before swapping

In above example, list a has six elements [5,1,0,4,2,3] and f parameter value of 3 is passed to _hop() function and number of hops is calculated using recursion
Base condition is the function will return 0 if element at index 'f' I.e a[f] is equal to f (a[f] == f)
Otherwise, it will swap the elements at the index and value present at the other index.
So, in this example value at index 3 which is 4 is swapped with value at index 4 or vice versa.

After the elements are swapped, we get list elements as a = [5,1,0,2,4,3] as shown in above image. Now we calculate the recursive expression hop = 1 + self._hop(a,3)

self._hop(a,3) is pushed on top of stack and this operations are performed until base condition is satisfied and zero value is returned.

After the base condition is meet and value is returned, the pop operation is done in which we re-swap the array to bring the original array back as shown in the figure-1

Time complexity: (Number of times the function is called * time complexity of each function run)
As the function calls itself, for worst case it will take N-1 hops to complete so the time complexity is O(N)
Space Complexity:
The list a is stored in heap and a reference variable is used to access the list, so the space complexity is O(N)