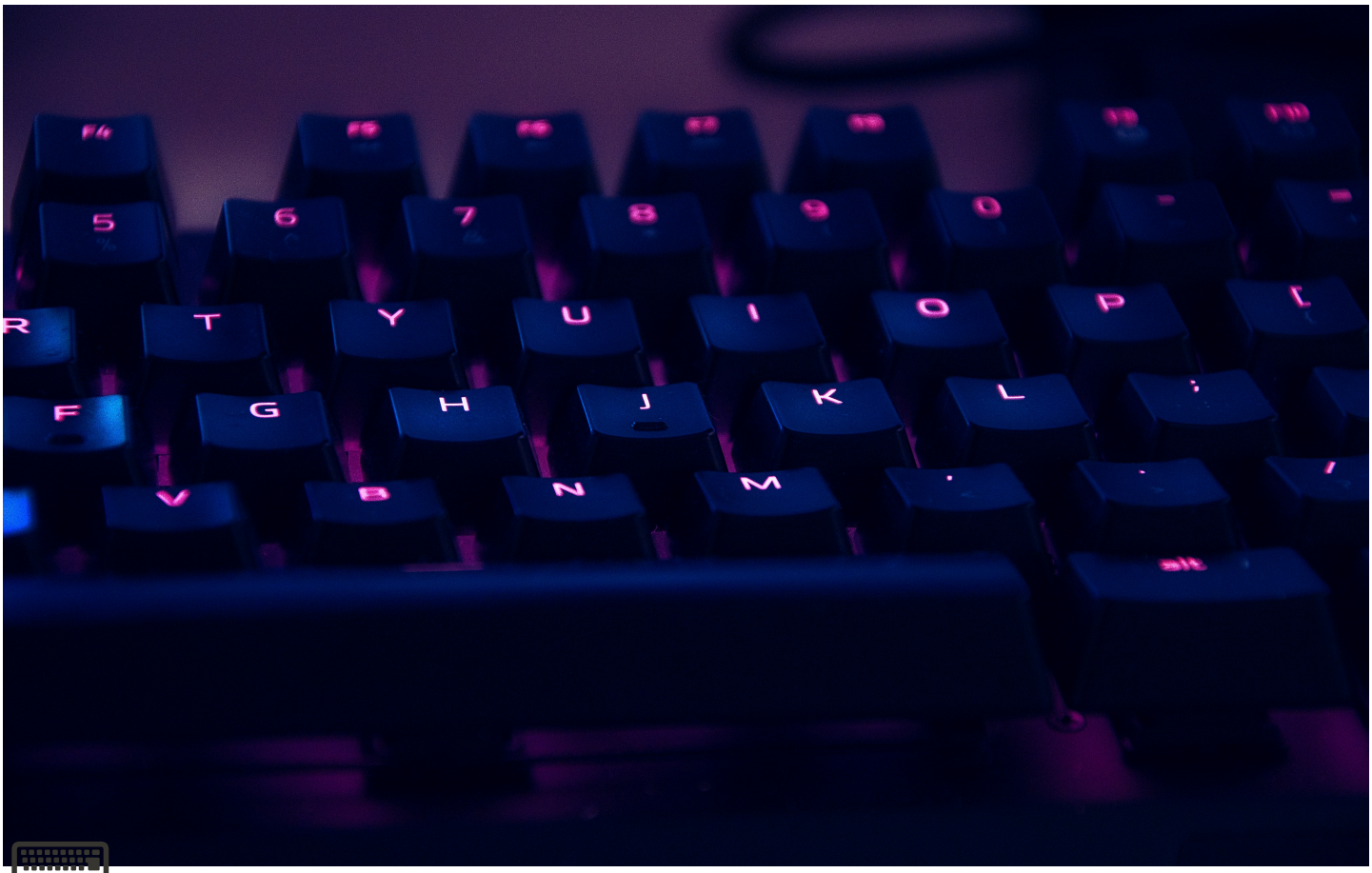

⌨

DAY 3 - 4

Easy

▼ Sum of the branches of a Tree:

```
class BinaryTree:
 def __init__():
  self.value = value
  self.left = left
  self.right = right

def branchsums(root):
 sums = []
 calculateBranchSums( root, 0 , sums)
 return sums

def calculateBranchSums(node, runningSum, sus):
 if node is None:
  return

 newRunningsum = node.value + runningSum
 if node.left is None and node.right is None:
 sums.append(newRunningSum)
 return

 calculateBranchSums(node.left, newRunningSum, sums)
 calculateBranchSums(node.right, newRunningSum, sums)
```

▼ Depth for first Search

```
class Node:
 def __init__(self, name):
  self.name = name
```

```
  self.children = children[]

 def addnode(self, name):
  self.children.append(NODE(name)

 def depthfirstsearch(self, array):
  array.append(self.name)
  for child in children:
   child.depthfirstsearch(array)
  return array
```

▼ Construction Linked list

Lets start with making the class and initializing the values

```
class DoublyLinkedList:
 def __init__(self):
  self.head = None
  self.tail = None
```

TC: O(1) SC:O(1)

To set the head of the linked list

```
def sethead(self):
 if self.head is None:
  self.head = node
  self.tail = node
  return
self.insertbefore(self.head, node)
```

TC: O(1) SC:O(1)

To set the tail of the linked list

```
def sethtail(self):
 if self.tail is None:
  self.sethead(node)
  return
self.insertafter(self.tail, node)
```

TC: O(1) SC:O(1)

Insert before a node

- First to check if we are dealing a node that is a head or tail
- Lets check if the node we are inserting the nodetoinsert is not the head

```
def insertBefore(self, node, Nodetoinsert):
  if nodetoInsert == self.head and nodetoInsert == self.tail:
   return
  self.remove(nodetoinsert)
  nodetoinsert.prev = node.prev
  nodetoinsert.next = node
  if node.prev is None:
   self.head = nodetoinsert
  else:
   node.prev.next = nodetoinsert
  node.prev = nodetoinsert
```

TC: O(1) SC:O(1)

insert After a node

- gonna apply the same checks as above

```
def insertafter(self, node, Nodetoinsert):
  if nodetoInsert == self.head and nodetoInsert == self.tail:
   return
  self.remove(nodetoinsert)
  nodetoinsert.next= node.next
  nodetoinsert.prev= node
  if node.next is None:
   self.tail= nodetoinsert
  else:
   node.next.prev= nodetoinsert
  node.next = nodetoinsert
```

TC: O(p) SC:O(1)

Insert at a Position

```
def insertAtPosition(self, Position, nodetoinsert):
 if position = 1:
  self.head = nodetoinsert
 node = self.head
 currentposition = 1
 while node is not None and currentposition = position:
  node = node.next
  currentposition += 1
 if node is not None:
  self.insertBefore(node, nodetoinsert)
 else:
  self.settail(nodetoinsert)
```

TC: O(n) SC:O(1)

To remove a node with a value

```
def RemoveNodeswWitvalue(self, Value):
 node == self.head
 while node is not None:
  nodetoremove == node
  node == node.next
   if nodetoremove == value:
    se  lf.remove(nodetoremove)
```

TC: O(1) SC:O(1)

To remove a node

- Here to check if we are dealing with a head or tail of linked list if this both condition does not satisfies then all good we will proceed further with and if not we will make the value the head or the tail of the linked list.

```
def remove(self, node):
 if(node == self.head):
  self.head == self.head.next
 if(node == self.tail)
  self.tail ==  self.tail.prev
 self.removeNodebinding(node)
```

TC: O(n) SC:O(1)

To check if our linked list has a particular value.

```
def containsavalue(self, value):
 node = self.head
 while node is not None and node.value != value:
  node = node.next
 return node is not None
```

As we have checked for the pre conditions in the remove() method above its time for us to remove the node now

```
def removeNodebinding(self, node):
 if node.prev is not None:
  node.prev.next = node.next
 if node.next is not None:
  node.next.prev = node.prev
 node.next = None
 node.prev = None
```

- Nth Fibonnaci

  Take a number → N and return the Nth fiboncaci number in the fibonacci series

  ```

  ```

- Product Sum

  Sum and special array

  ```
  def ProductSum(array, multiplier=1):
  sum = 0
  for element in array:
   if type(element) is list:
    ProductSum(element, multiplier + 1)
   else:
    sum = sum + element
  return sum * multiplier
  ```

- Binary Search

  Recursively

  ```
  # O(log(n)) time | O(log(n)) space
  def binarySearch(array, target):
   return binarySearchHelper(array, target, 0, len(array) - 1)
  def  binarySearchHelper(array, target, left, right):
   if left > right:
    return -1
   middle = (left + right) // 2
   potentialMatch = array[middle]
   if target == potentialMatch:
    return middle
  elif target < potentialMatch:
    return binarySearchHelper(array, target, left, middle - 1
   else:
    return binarySearchHelper(array, target, middle + 1, right
  ```

  Itreteaviley

  ```
  # O(log(n)) time | O(1) space
  def binarySearch(array, target):
   return binarySearchHelper(array, target, 0, len(array) - 1)
  def binarySearchHelper(array, target, left, right):
   while left <= right:
    middle = (left + right) // 2
    potentialMatch = array[middle]
    if target == potentialMatch:
     return middle
    elif target < potentialMatch:
     right = middle - 1
    else: left = middle + 1
     return -1
  ```

- Find Three Largest Number

  Find Three Largest Number in an array.

```
# O(n) time | O(1) space
def findThreeLargestNumbers(array):
 threeLargest = [None, None, None]
 for num in array:
 updateLargest(threeLargest, num)
  return threeLargest

def updateLargest(threeLargest, num):
 if threeLargest[2] is None or num > threeLargest[2]:
  shiftAndUpdate(threeLargest, num, 2)
 elif threeLargest[1] is None or num > threeLargest[1]:
  shiftAndUpdate(threeLargest, num, 1)
 elif threeLargest[0] is None or num > threeLargest[0]:
  shiftAndUpdate(threeLargest, num, 0)
```