ONUR
SEZER
121044074

CSE 321 - HW03

1) a) $gcd(m,n) = gcd(n, m \bmod n)$

yeni çiftin size'ı $m \bmod n$ olacak.

Bu nedenle 0 ile $n-1$ arasında bir değer olabilir. n size'ı 1 ve n arasındaki bir sayı kadar azalma gösterecek.

b)

$gcd(m,n) = gcd(n,r) = gcd(r, n \bmod r)$

Burda $r = m \bmod n$ 'dir.

Göstermemiz gereken → $\boxed{n \bmod r \leq n/2}$

$r \leq n/2$, $n/2 < r < n$ bu durumları dikkate alalım. Eğer $r \leq n/2$ ise;

→ $n \bmod r < r \leq n/2$

Eğer $n/2 < r < n$ ise;

→ $\boxed{n \bmod r = n - r < n/2}$ olur

2)

```
def permute(num):
    if len(num) == 2:
        # get the permutations of the last 2 numbers
        # by swapping them
        yield num
        num[0], num[1] = num[:], num[0]
        yield num
    else:
        for i in range(0, len(num)):
            # fix the first number and get the permutations of
            # the rest of numbers
            for perm in permute(num[0:i] + num[i+1:len(num)]):
                yield [num[i]] + perm


for p in permute([1,2,3]):  # yukarda yazilan fonksiyon
    print p                 # calistiriliyor

>
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

3)

```
def delete(root, key):
    if root is None: # Base Case
        return root
    if key < root.key: # key root'tan kucukse sol subtree'ye bakil
        root.left = delete(root.left, key)
    elif key > root.key: # key root'tan buyukse sag subtree'ye bakil
        root.right = delete(root.right, key)
    else: # key root'ta ise silinir
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = minValueNode(root.right) # sag subtree'de en kucugu bulur
        root.key = temi.key
        root.right = deleteNode(root.right, temp.key)
    return root
```

a) Variable-size-decrease algoritmasi yukarda kurdugumuz algoritmayi saglamaz. Çünkü daha küçük binary tree'lerde key silinirken problem azalmaz.

b) Complexity → O(n) olur

En kötü durumda tree nin en dibindeki leaf'ine bakabilir

4) O(n) zamanda sıralanabilir. Bunun için counting algoritması kullanılarak, Sıralanacak olan dizide her sayının kaç tane olduğunu farklı bir dizide sayarız. Dizimizde -1 olduğu için -1 sıfırıncı indexte, 0 birinci indexte 1 de ikinci indexte kaç tane olduklarını tutulur. Yani her eleman kendi değerinin 1 fazlası olan indexte temsil edilmiş oldu. Counting yapılan dizinin üstünden geçilerek indexlerin değeri kadar index-1 elemanı sıralanacak arraye yazılır.

ÖR:

-1	1	1	1	0	-1	0

arrayi input olarak gelsin.

ilk önce counting arrayi tutuyoruz.

-1 → 0.
0 → 1.
1 → 2. indexlerde kaç tane geçtiği tutulur

0	1	2
2	2	3

→ Counting arrayi

→ -1 den 2 tane
→ 0 dan 2 tane
→ 1 den 3 tane olduğu bulundu

Daha sonra input olarak alınan arrayin size'ı kadar sıralama array'i oluşturulur.
Counting arrayi üzerinden geçilerek sıralama arrayi doldurulur.

1. adım

-1	-1					

→ 0. index in değeri kadar -1 yazıldı.

2. adım

-1	-1	0	0			

→ 1. index'in değeri kadar 0 yazıldı.

3. adım

-1	-1	0	0	1	1	1

→ 2. index'in değeri kadar 1 yazıldı.

İki tane arrayin üstünden geçildiği için toplam zaman n+k'dır. (Complexity bu durumda O(n) çıkar)

5)

```
def binarySearch(list)
    if len(list) == 0:
        return -1
    mid = len(list)/2
    if mid == list[mid]:
        return mid
    elif mid > list[mid]:
        return binarySearch(list[mid+1:])
    else:
        return binarySearch(list[:mid])
```

// Binary search algoritmasi kullanilarak A[i] == i olup olunmadigina bakildi. Eger bulursa buldugu index'i return edur. Eger recursion'in ilk kosuluna takilirsa o zaman listenin tum elemanlarina bakildigi anlasilir ve -1 return eder.

Complexity → O(logn)