گزارش تمرین عملی اول – طراحی الگوریتم

سوال 1)

ابتدا تا زمانی که کاربر ورودی وارد کند ورودی هارا در آرایه keys ذخیره میکنیم:

```
# Getting input keys
keys = []
while True:
    user_input = input()
    if len(user_input) > 0:
        keys.append(int(user_input))
    else:
        break
```

در قدم بعدی تابع constructBST را فراخوانی میکنیم که دنباله ای از اعداد را ورودی میگیرد و به صورت زیر است:

```
# Function to construct balanced BST from the given unsorted list
def constructBST(keys):
    # sort the keys first
    keys.sort()

    # construct a balanced BST and return the root node of the tree
    return construct(keys, 0, len(keys) - 1, None)
```

در این تابع ابتدا آرایه ورودی را سورت میکنیم سپس تابع دیگری را فراخوانی میکنیم به نام construct که در نهایت گره ریشه (root) را باز میگرداند

این تابع بصورت زیر خواهد بود:

```
# Function to construct balanced BST from the given sorted list
def construct(keys, low, high, root):
    # base case
    if low > high:
        return root

    # find the middle element of the current range
    if (low + high) % 2 == 0:
        mid = (low + high) // 2
    else:
        mid = ((low + high) // 2) + 1

    # construct a new node from the middle element and assign it to the root
    root = Node(keys[mid])

    # left subtree of the root will be formed by keys less than middle
element
    root.left = construct(keys, low, mid - 1, root.left)

    # right subtree of the root will be formed by keys more than the middle
element
    root.right = construct(keys, mid + 1, high, root.right)

    return root
```

در این تابع ورودی ها آرایه وارد شده کاربر که سورت شده و دو مقدار اشاره کننده به ابتدا و انتهای آرایه است که در ابتدای کار low برابر با صفر و high برابر اخرین خانه است len(keys)-1.

در مراحل بعد با مقایسه این دوتا مقدار انقدری میانگین گیری میکنیم تا به زمانی برسیم که مقدار low بیشتر یا مساوی high شود. در هرمرحله میانگین گیری اگر پیدا کردن عدد وسط ممکن بود که مشکلی نداریم وگرنه دست بالا را انتخاب میکنیم که شرایط درختمان به صورتی باشد که مسئله خواسته (حالت اول این است که برگ هایی که null هستند رو در طرف راست نود های والد درنظر بگیریم. در این حالت برگ ها در طرف چپ والد خود قرار میگیرند. در اینصورت مقدار والد باید از مقدار برگ ها بیشتر باشد) پس از پیدا کردن مقدار وسط(مثلا اگر فرض کنیم مقادیر 10,-3,0,5,9- را داریم و حالا مقدار 0 انتخاب میشود) در اولین مرحله نتیجه بعنوان ریشه انتخاب میشود و حال برای گره های راست و چپ آن تابع مجددا فراخوانی میشود اما با ورودی های متفاوتی که نشانگر چگونگی اشاره به ابتدا و انتها (بازه بندی مدنظر) در ارایه هستند. با فراخوانی برای سمت چپ میدانیم که دنبال گره ای با مقدار کمتر از ریشه هستیم

بنابراین بازه low = 0تا اخرین عدد قبل از ریشه mid-1=1 انتخاب میشود (یعنی 3-,10-) حال با اجرای دوباره مقدار mid =1 میشود و بزرگترین مقدار کوچکتر از ریشه انتخاب میشود و به عنوان ریشه فعلی درنظر گرفته شده و دوباره تابع را برای گره های سمت راست و چپ خود فراخوانی میکند. در این مرحله اگر گره ای موجود باشد تا زمانی که به انتهای این شاخه برسیم جلو میرود وگرنه مقدار low > high شده و اخرین گره ای که به عنوان ریشه نگهداری شده بود را برمیگرداند و سراغ گره سمت راست میرود. لازم به ذکر است که در ابتدای کار کلاسی تحت عنوان Node تعریف شده بود که شامل دیتای گره و گره سمت راست و چپ آن است.

با اجرای این الگو درخت تکمیل شده و نهایتا ریشه اصلی برگردانده میشود

در قدم بعد فراخوانی تابع levelOrder انجام میشود که ریشه اصلی را ورودی میگیرد و مرتب سازی سطح به سطح را انجام میدهد و به ارایه answer اضافه میکند

```
# Function to  print level order traversal of tree
def levelOrder(root):
    h = height(root)
    for i in range(1, h + 1):
        currentLevel(root, i)
```

برای این کار ابتدا ارتفاع درخت را با کمک تابع height پیدا میکنیم:

```
""" Compute the height of a tree--the number of nodes
    along the longest path from the root node down to
    the farthest leaf node
"""
def height(node):
    if node is None:
        return 0
    else:
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        # Use the larger one
        if lheight > rheight:
            return lheight + 1
        else:
            return rheight + 1
```

برای پیدا کردن ارتفاع درخت زیر درخت هارا انقدری پیمایش میکنیم تا به مقدار حداکثری برسیم و آن را برمیگردانیم.

در قدم بعدی با حرکت روی درخت به اندازه ارتفاع آن تابع currentLevel را فراخوانی میکنیم:

```
# Print nodes at a current level
def currentLevel(root, level):
    if root is None:
        return
    if level == 1:
        answer.append(root.data)
    elif level > 1:
        currentLevel(root.left, level - 1)
        currentLevel(root.right, level - 1)
```

در این تابع در هرسطحی که به عنوان ورودی وارد شده انقدری به عمق میرویم که به برگ ها برسیم و پس از ان از چپ به راست آنهارا به ارایه answer اضافه میکنیم.

در نهایت ارایه answer را به عنوان جواب نهایی نمایش میدهیم.

سوال2)

در این سوال به دنبال حداقل تغییرات ممکن برای یکسان سازی تمامی رشته های باینری ورودی هستیم. در اولین قدم کاربر اعلام میکند که چند درخواست دارد و بعد در هر درخواست اعلام میکند که چند رشته (n) و به چه اندازه (m) وارد خواهد کرد. سپس رشته های مدنظر را وارد میکند.

```
result =[]
number = int(input())
for i in range(0, number):
    nm = input()
    nm2 = nm.split(" ")
    n = int(nm2[0])
    m = int(nm2[1])
    arr = []
    for j in range(0, n):
        a = input()
        arr.append(a)
    result.append(checkStrings(arr,n,m))
```

پس از ذخیره سازی متغیرهای لازم و رشته های ورودی برای هردرخواست کاربر تابع checkStrings فراخوانی میشود.

```
#checking strings in order to equalize them with least possible changes
def checkStrings(arr, n, m):
    t = []
    for p in range(0, n):
        t.append(changeStr(p, arr, n, m))
    return min(t)
```

در این مرحله هربار یک رشته را به عنوان مرجع درنظر گرفته و محاسبه میکنیم چند تغییر لازم است تا در نهایت باقی رشته ها به آن تبدیل شوند و پس از محاسبه همه حالات مقدار حداقلی را برمیگردانیم که جواب مسئله خواهد بود. (مثلا تست اخر را درنظر بگیرید که 4 رشته 5 تایی را وارد میکردیم، در هر بار محاسبه یکی از رشته ها مرجع درنظر گرفته میشود و محاسبه میشود چند تغییر لازم است تا باقی رشته ها به آن تبدیل شود پس در نهایت یک آرایه چهارتایی از اعداد داریم که با مینیموم گرفتن از ان به جواب مسئله میرسیم.) برای انجام این محاسبه پس از انتخاب مرجع تابع changeStr را فراخوانی میکنیم:

```
#Estimates needed changes to equalize strings to the source one
def changeStr(p, arr, n, m):
    arra = arr.copy()
    forThis = 0
    for i in range(0, n):
        if i != p:
            t = m - 1
            f = 0
            while t > -1:
                if arra[i][t] == arra[p][t]:
                    t = t - 1
                else:
                    arra[i] = flip(arra[i], t, m)
                    f = f + 1
            forThis = forThis + f
    return forThis
```

در این آرایه ابتدا یک کپی از ارایه ورودی میگیریم و متغیری را تعریف میکنیم که مجموع تغییر ها برای تبدیل رشته ها به مرجع را ذخیره کند. در قدم بعد برای هررشته (مشخصا بغیر از خود رشته مرجع) شروع به مقایسه میکنیم، اگر برابری در کاراکترهای رشته مدنظر و مرجع وجود داشت نیاز به تغییر نداریم وگرنه بزرگترین پیشوند ممکن فلیپ میشود به کمک تابع فلیپ.

این تابع به صورت زیر است:

```
def flip(str, t, m):
    newStr = ''
    for s in range(0, t + 1):
        if str[s] == '0':
            newStr = newStr + '1'
        else:
            newStr = newStr + '0'

    if t + 1 < m:
        for x in range(t + 1, m):
            newStr = newStr + str[x]

    return newStr
```

در این تابع تا اندیسی که در ورودی داده شده فلیپ میشود و باقی رشته کپی میشود.

پس از انجام فرایند بالا برای هر درخواست کاربر و اضافه کردن آن به ارایه result جواب نهایی نمایش داده میشود.

```
for r in result:
    print(r)
```