1. The expression which is invariant after each iteration of the for loop is $\boxed{i - j == k}$

Explanation :-

The for loop is running through all the elements from start to end - 1.
If statement is dividing the loop in two parts one part contain the squares another part contains non squares.

In the if square(m) condition,
i is getting ~~decreas~~ decreased by 1 for each square values in the rang.
And
k is getting decreased by 1

If else :
j is getting increased by 1 when we are finding a non square value
And
k is getting decreased by 1.

So, It doesn't matter if we get square or not,
k is always decreasing by 1 for each iteration of the loop.

And i will be ~~increased~~ decreased ~~for~~ if we get a square, otherwise j will keep increasing by 1.

Now. we are taking -j and adding the value with i.

So, we can consider in each iteration,

- i = is a set containing square elements ~~till~~ of cardinality n in the range of (start, start# - (k-1)).

+ j = is cardinality of set containing non square elements in the range of (start, start - (k-1)).

and -k is the cardinality of all elements in the range (start, start - (k-1)).

So, -i and j are disjoint sets. they don't have any intersection point.

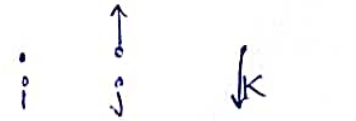
Hence $-i + j == -k$

So, $\boxed{i - j == k}$ is loop invariant.

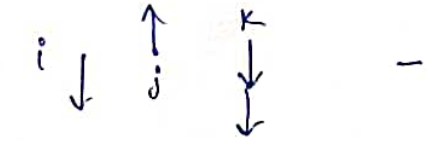
Example

3 4 5 →

3 — i = 0, j = 1, k = -1



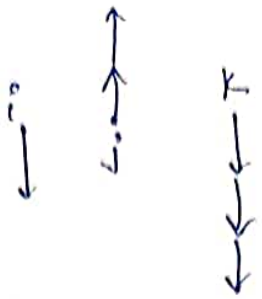
4 — i = -1, j = 1, k = -2



5 — $i = -1$

$j = 2$

$k = -3.$



$-1-2 = -3.$

Hence $i - j = k$ is loop invariant.

2. $(L1, L2, L3) = ([73], [82], [91])$

$(x1, x2) = (20, 21)$

$L_1 = [73]$

$L2 = [82, 80]$

$L3 = [\quad]$

} output of the function.

$x_1 = 20$

$x_2 = 63.$

here in the code L3 and x_2 are global variable.

L_1 is not a global variable so, the operation $L1 = L3 + [74]$ inside the function updates the list $L1 = [91, 74]$ But as this operation is immutable the outside function L1 doesn't change so, $L1 = [73]$.

L2 is not a global variable But the operation L2.append(80) is mutable in nature so, the outside function will also update the value of L2 to $[82, 80]$.

L3 is global variable mentioned in the code. so the operation $L3 = L3 + [90]$ will update its value to $[91, 90]$.

x_1 is not global variable, so, inside the function

a new variable x1 is created but due to immutable in nature, x1 is value outside the function doesnot change so, x1 = 20.

But x2 is global variable. so, x2 = x2+42
will change its value 21+42 = 63.

so, x2 = 63.

3

(3)

```
def f(.....):
    print("a", a, "b", b, "c", c, "d", d)
```

f(d=4, a=3)

a 3 b 20 c 35 d 4

b and c are getting values from the argument of the function.

So, the default values of b=20
c=35.

f(3,5,7)

a 5 b 7 c 35 d 3.

Here d=3.
a=5
b=7
and c is taking its default argument

So, the ordering is important.

d comes first, a comes second and b comes and then c comes.

d → a → b → c f(d a b c)

f(3, b=7) because no values of a is mentioned. and a has no default Argument.

So, the error is coming.

So, f(d, a, b=20, c=35). (definition).

6.

```
def dosasort(A):
    n = len(A)
    while n > 1:
        max = max_val(A)
        if max != n-1:
            flip(A, n)
            flip(A, n-1)
        n = n-1

def max_val(A):
    return(A.index(max(A))).
```

6.

```
def flip(A, i):
    A[i:] = reversed(A[i:]) # given
    return

def dosasort(A):
    if A == []:
        return []
    else:
        max_dossa = max(A)
        i = A.index(max(A))
        flip(A, i)
        flip(A, 0)
    return([max_dossa] + dosasort(A[1:]))
```

7.

```
def mymap(self, f):
    if self.isempty():
        return
    else:
        temp = self
        While temp != None :
            k = temp.value
            t = f(k)
            temp.value = t
            temp = temp.next
```

8.

```
class Dset:
    def __init__(self, l=[]):
        if l != None:
            for i in l:
                if i not in self.set.keys():
                    self.set[i] = -1

    def size(self):
        flag = 0
        for j in self.set.keys():
            flag = flag + 1
        return(flag)

    def member(self, x):
        return(x in self.set)

    def union(self, new):
        l1 = sorted(list(self.set.keys()))
        l2 = sorted(list(new.set.keys()))
        l3 = dict()
        (i, j, m, n) = (0, 0, len(l1), len(l2))
        while (i+j) < (m+n):
            if i == m:
                l3[l2[j]] =
                j = 1
                j+1

            elif j == n:
                l3[l1[i]]
                i = 1
                i+1
```

```
While i < m and j < n :
    if (l1[i] < L2[j]):
        L3[L1[i]]
        i = 1
        i + 1
    elif (l1[i] > L2[j]):
        L3[L2[j]]
        j = 1
        j+1
    elif (L1[i] == L2[j]):
        L3[L1[i]]
        i = 1
        j = 1
        i+1
        j+1
← return(L3).

def intersection(self, new):
    L1 = sorted(list(self.set.keys()))
    L2 = sorted(list(new.set.keys()))
    L3 = dict
    (i,j,m,n) = (0,0, len(L1), len(L2)).
    While (i+j) < (m+n):
        if i == m:
            j = n
        elif j == n:
            i = m
```

```
while i < m and j < n :
    if L1[i] < L2[j] :
        i += 1
    elif L1[i] > L2[j] :
        j += 1
    elif L1[i] == L2[j] :
        L3[L1[i]]
        i = 1
        j = 1
        i+1
        j+1.

← return(L3).
```

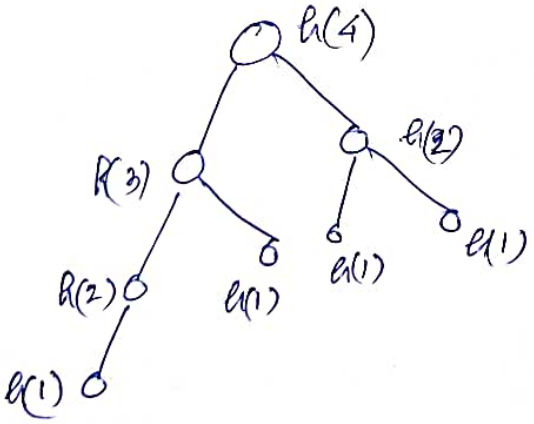
5. The function mytree.foo() will compute the height of the tree.

```
def foo(self):
    if self.isempty()
        return(0)
    elif self.isleaf():
        return(1)
    else:
        return(1 + max(self.left.foo(), self.right.foo()))
```

From return function it is clear that it function foo will deal with some length.

Because for empty node, the function returns zero.

If the node is the leaf node, the height will be 1



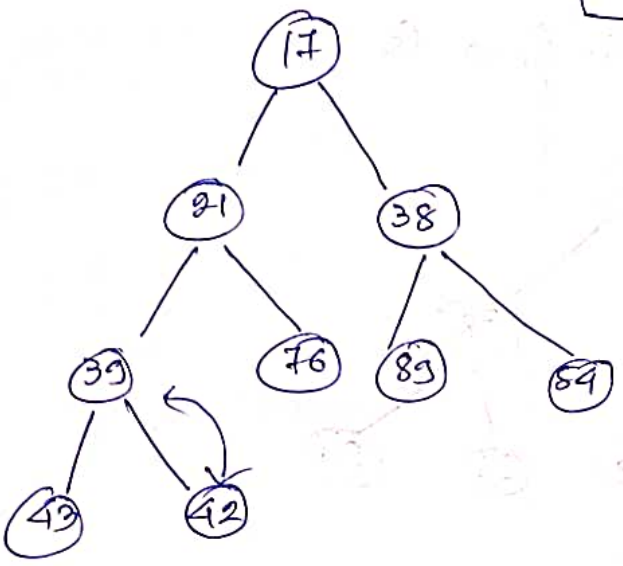
the recursive function, self.left.foo() or self.right.foo() will return the lengths recursively. and max takes the max of the left child and right child and add 1 to take account of the parent node.

That is why, if mytree.foo() gives the height of the tree which is log(n) for n elements.

④. The heap structure is.

17, 21, 39, 42

can be inserted at last.

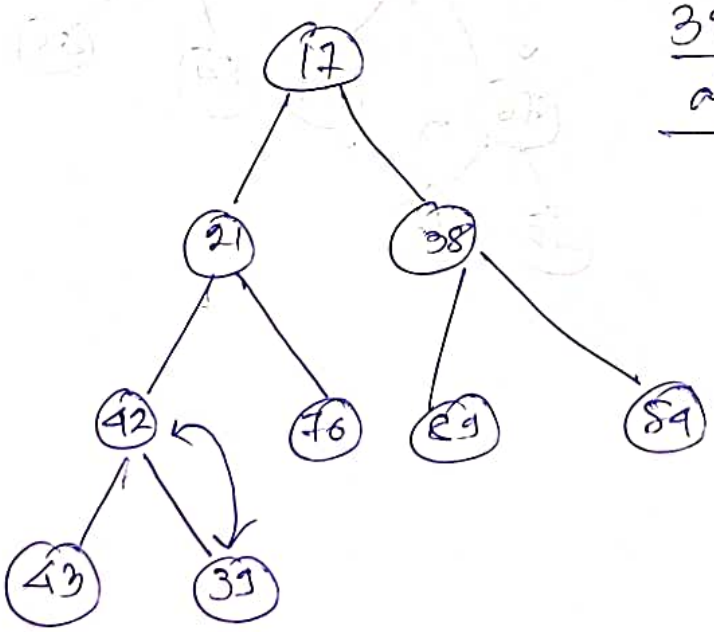


Obviously 42 can be inserted at last.

now, we can ~~insert~~. swap the last element with the previous elements liked to the last element to get the same heap, like swapping 39 and 42,

[17, 21, 38, 42, 76, 89, 84, 43, 39]

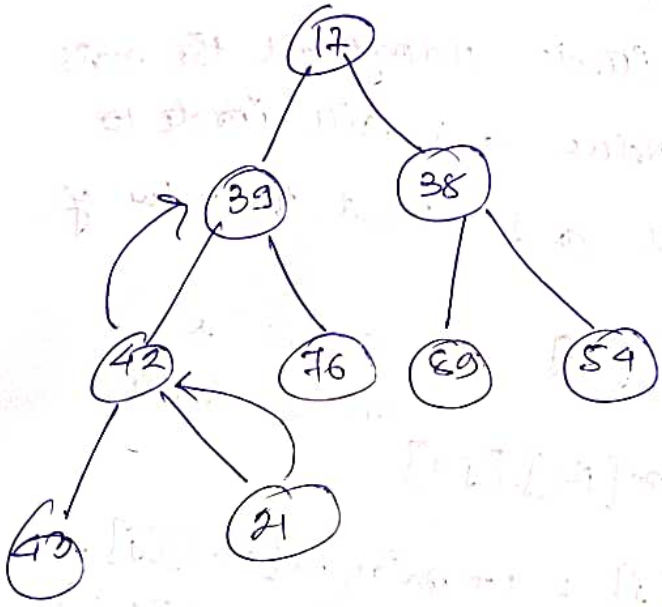
39 can be inserted at last



now, swapping 21 with 39 in the initial list and 39 with 42. (first)

then [17, 39, 38, 42, 76, 89, 54, 43, 21]

the heap will be

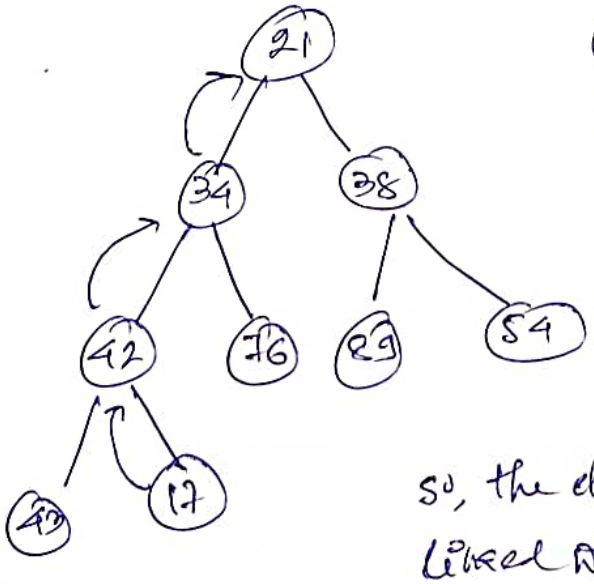


it will restore the heap structure to its initial.

we can insert 21 in the end.

now, lastly, [21, 34, 38, 42, 76, 89, 54, 43, 17]

this will create



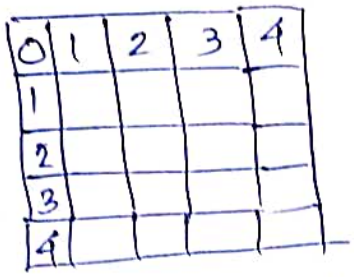
like this we can restore the heap property

so, we can insert 17 at last

so, the elements which are directly linked with the last element can be inserted last.

9.(a) The row number and column number s are initialised.

Mismatch table is m.



now we will iterate through all the rows and will count x value and will iterate to all the columns w and count the value of.

if $s[i-1] = t[y-1]$ s and t are two words.

$m[i][j] = m[i-1][j-1]$

otherwise, $m[i][j] = 1 + \min(m[i-1][j], m[i][j-1])$