Q 1)

$f(\text{schedule}, n) = g(\cancel{\text{schedule}, n}, 1, \cancel{\text{sundays}}, \cancel{\text{sum schedule}}$

$g(\text{schedule}, n, 1, 0, 0, 0)$

$g(\text{schedule}, n, i, \text{sundays}, \text{sum}, \text{count})$

$$= \begin{cases} (\text{sundays} + i) = (\text{sum} + \text{schedule}(i)) & i = N \\ g(\text{schedule}, n, i+1, \text{sundays}+i, \text{sum} + \text{schedule}(i), \text{ if } (\text{sundays}+i) = \text{sum} + \text{schedule}(i) \text{ then count} + 1 \text{ else count}) & i < n \end{cases}$$

if the sum of the days till any k = the ~~sum of~~ $\sum_{i=1}^{k} \text{schedule}(i)$ then that i has all topics well studied.

so that is what this algorithm does, and iterates from $i=1$ to $i = n$, therefore its of $O(n)$.

Q2)

a = stk.top()

b = stk.pop()

c = x in op1

d = op1[index(op2, x)]

e = stk.size() == 0

~~3) def remove_all (l, x):~~
~~if l.pop() == x~~
~~remove_all (l, x)~~
~~else~~

3) 1)

```
def remove_all (l, x):
    a = l.pop()
    if l == []:
        return []
    elif a == x:
        return (remove_all (l, x))
    else
        return (remove_all (l, x).append (a))
```

2) Time complexity analysis:

let $n = len(l)$

Base Case: For $n = 0$

$T(0) = k$ (some constant)

Induction hypothesis:

~~Let $T_n = T_{n-1} + k$ be true for n~~

We claim that $T(n) = kn + c = O(n)$

Let $T(n)$ be true for n

Induction step:

We know that during recursion, when we receive the output from the function called before, we just append 'a' to the list or not, both taking $O(1)$ operation

Therefore

$$T_{n+1} = T_n + k$$

$$= kn + c + k$$

$$T_{n+1} = k(n+1) + c = O(n+1)$$

Hence Proved, Time complexity is of ~~ord~~

$O(n)$

Q4) We know that $\sqrt{b^2-4ac}$ (which is used in both cases is positive (assuming real roots)

$\therefore$ We have to take caution while we are subtracting either two positive number or two negative number both of which result in a high relative error.

Case 1: $b > 0$

then $-b < 0$

In this case $-b + \sqrt{b^2-4ac}$ is like subtracting two positive numbers, so we need to avoid this & while we dont need to ~~avoid~~ avoid $-b - \sqrt{b^2-4ac}$.

$\therefore$ when $b > 0$

We find the roots using the formulas

$$\frac{2c}{-b-\sqrt{b^2-4ac}} \quad \text{and} \quad \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Case 2: $b < 0$

then $-b > 0$

So in this case we have to reverse the operation and therefore we use

$$\frac{-b+\sqrt{b^2-4ac}}{2a}, \quad \frac{2c}{-b+\sqrt{b^2-4ac}}$$ (since both are additional function)

Scanned with CamScanner

And these cases will give us ~~[illegible]~~ stable roots.

1)

[illegible]

Q5) 1) def isReachable(x1, y1, x2, y2):

```
if x1 == x2 and y1 == y2:
    return True
elif x1 > x2 or y1 > y2:
    return False
else
    return isReachable(x1+y1, y1, x2, y2) or
           isReachable(x1, x1+y1, x2, y2)
```

2) Correctness proof

Base case:

$x_1 > x_2$ or $y_1 > 2$ = False

$x_1 == x_2$ and $y_1 == y_2$ = True

~~Induction~~

We use recursion to show that if Path the base cases are not the current case, we go along both the paths $(x+y, y)$ and $(x, x+y)$ so finally we will have output of all paths separated by (or)

i.e. Path1 or Path2 or Path3 or.... or Path(N)

if even 1 of these Paths are true (can make us reach airport, we return true.

If none of the Paths are true, we return false. Hence we get correct output

6) 1)

```
def nonOverlappingIntervals(intervals):
    a = [intervals[0]]
    for i ~~intervals~~ in range(1, len(intervals)):
        if intervals[i][0] <= a[-1][1] :
            ~~a[-1][1] = intervals[i][1]~~
            if intervals[i][1] > a[-1][1] :
                a[-1][1] = intervals[i][1]
        else:
            a.append(intervals[i])
    return (a)
```

2) Correctness Proof:

We first initialise a with first element of interval.

We know that intervals is sorted according to the start i.

If the start of i^{th} element < end of $i-1^{th}$ element, we merge them.

So we iterate i from (1 to $\underbrace{\text{len(intervals)}-1}_{\text{Indices are 1 less}}$)

Now if the last element of a = c (i.e: a[-1] = c)

if c[1] ≥ intervals[i][0] we have to merge them, Now if intervals[i] is totally included in c, we dont need to alter c

but if the intervals[i]'s end is larger that [c]'s end (intervals[i][1] > c[1])
we have to increase c[1] to intervals[i][1], so thats what we do.

We do this ~~for~~ till i = len(l), and we get a ~~merged~~ non overlapping interval list.

Hence Proved.

Q7)

```
def getLargest(self):
    x = None
    y = None
    if isInstance(self.left, TreeNode):
        x = self.left.getLargest()
    elif isinstance(self.left, int)
        x = self.left

    if isinstance(self.right, Treenode):
        y = self.right.getLargest()
    elif isinstance(self.right, int):
        y = self.right

    if x != none and y != none:
        return max(x, y)
    elif x == none and y != none:
        return y
    elif x != none and y == neone:
        return x
    else
        return None
```

2)

Correctness Proof :

First we check that if ~~either of~~ self.left is a node or not, if it is, we find the max of that node and assign x = self.left.getLargest().
If its an integer we assign x = self.left.
else x = None

similarly we get values for y recursively

Now having both the value of max of left and right node,

We check that if Both are not None,
we return max(x, y)
if one of them is None,
we return the other one,
else we return None.

So we recursively obtain the maximum of all the nodes as x and y themselves will the max of left branch and right branch respectively.