2) Our key idea for solving this problem is to construct a new graph, where vertices of it represent the edges of original graph, and we connect vertices of new graph v_i and v_j by an edge if and only if these vertices represent two edges with the same endpoints. So, basically taking our edges from original graph, we construct vertices from them (correspond each edge to a vertice) such that v_i and v_j are adjacent $\Leftrightarrow$ edges from original graph shared same endpoints.

From this construction, we observe that assigning the smallest # of possible colors to edges of a given graph is equivalent to assigning smallest # of possible colors to vertices of our new graph, such that no two edges sharing the same endpoint are same colored, or equivalently, no two vertices v_i and v_j, which are adjacent, are same colored. Therefore, we reduced original edge-coloring problem to the problem $\rightarrow$ finding the smallest # of possible colors to vertices of a graph so that no two adjacent vertices v_i and v_j are same colored which is obviously vertex-coloring problem ■

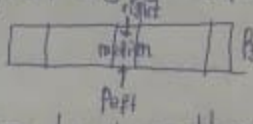
1) Before starting, we should make clear that it is okay to accept that mysterious machine returns median m of given set S and S\{m} in constant time (i.e. O(1)) Our another assumption also will be about numbers in the given list → consider that there are no duplicated numbers in given list. The key idea is firstly to create a new array with capacity=n, and we'll provide code snippets in Python which guarantees that it's possible to return a sorted output of these distinct n numbers using machine in linear time

```
lst = [None] * n          // We create an empty array, size=n
left = (n+1)//2           // Initialize two pointers, left & right
right = (n+1)//2
updateSet = S             // Create a variable which updates sets
median, S1 = machine(updateSet)   // Returns median, S\{median}
lst[left] = median        // Assign middle element to median
updateSet = S1            // Update the variable
for i in range(n-1):
    a, b = machine(updateSet)
    if a > lst[right]:
        lst[right+1] = a
        right = right+1
    else
        ...
    else:
        lst[left-1] = a
        left = left-1
    updateSet = b
```

Let's analyse on what happens in these codes → Firstly, we initialise a new empty array with length=n (it could also be possible to write lst=[0]*n, but essential idea was to create an array) Then, we initialise two pointers, left & right, which will keep track of a "for-loop" and they both start at the middle (I assume array indices are from 1 to n, so middle one will be $\left[\frac{n+1}{2}\right]$, which is (n+1)//2 in Python

Before the "for-loop", I want to mention that "left" & "right" variables will move to the left part or to the right part of the "lst" array in each iteration, but I will explain more detailed in future codes. "updateSet" will keep track of sets when machine deletes median of some set and returns the remaining set. So, basically we start with "updateSet=S" as our original set S, and then use "machine(input)" function for set S (original set). This function returns median of given n numbers, and S\{median}, which is S_1 in our case. As we know, median's correct position should be in the middle, since its definition requires for middle. As "left" was the middle, we assigned lst[left] to be median, which gives us correct position for "median" element. And then, we update "updateSet" to be S\{median}, or just S_1 in this case

It's easily seen that until the latter code (inclusive), every code was running in constant time, because "machine(input)" function returns median and remaining set in O(1) time, therefore what we did just costs O(1) time, with space complexity = Θ(n)

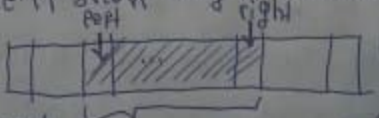


Now, we'll provide a claim to show the correctness of our algorithm, with providing Initialization, Maintanence, Termination stages in detailed way

Claim: lst[left .. right] consists of numbers in their own correct position, meaning that lst[left] is (left)th order of statistics, ..., lst[k] is k^{th} order of statistics for each left ≤ k ≤ right

Proof: Let's see why loop invariant in our code applies this claim to be true. For "Initialization", lst[left .. right] = median before first iteration of the loop, and we know lst[left] = median is in correct position, since left = (n+1)//2

Now, for "Maintanence", assume it's true before an iteration of the loop, then we'll show why it remains true after the next iteration →



each number in array's

Basically, [left, right] interval is located in correct position
For each k ∈ [left, right] ⇒ lst[k] is k^{th} order of statistics

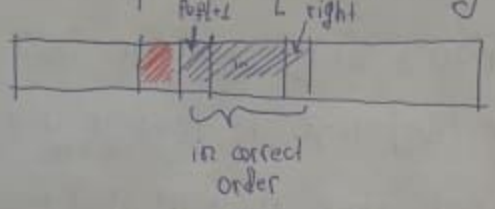
When we run "for-loop", we take the remaining set S_k, which consists of numbers except $\{Pst[left], \ldots, Pst[right]\}$ and machine returns median $\in S_k \setminus \{median\}$ in $O(1)$ time. Let right-left+1 = X and $|S_k| = n-X$. If "median" (variable name) is the middle element of "updateSet", then median will be the middle element of $S_k = \{Pst[1], \ldots, Pst[left-1], Pst[right+1], \ldots, Pst[n]\}$

We prematurely know that median can not be between $Pst[left]$ and $Pst[right]$, since it violates our inductive hypothesis. ($Pst[left], \ldots, Pst[right]$ are in correct places) There can be two cases: median < $Pst[left]$ or median > $Pst[right]$ (here, median = middle element of S_k)

If median > $Pst[right]$, it's implying that median is at least $(right+1)^{th}$ order of statistics; however, we'll show that it's exactly $(right+1)^{th}$ order of statistics → From definition, median was middle element of S_k, then median is greater than exactly $\left\lceil \frac{n-X-1}{2} \right\rceil$ # of elements in S_k.

We know that in previous iteration, either left was decremented or right was incremented. Our first consideration was about median > $Pst[right]$, now let's see what was previous iteration's possibility:

Assume "left" was added to the left, meaning Ast[left] was median of $S_k \cup \{Ast[left]\}$



"median" is middle of S_k and in our first case, median > Ast[right]

So, median > Ast[left] and $S_{k+1} = S_k \cup \{Ast[left]\} =$

$= \{..., Ast[left], ..., median, ...\}$ has middle element Ast[left]

When we drop that, median becomes middle of S_k

$\underline{\quad N \quad}$ Ast[left] ① $\underline{\quad N \quad}$ or $\underline{\quad N \quad}$ ② Ast[left] $\underline{\quad N+1 \quad}$

So basically, 2 cases for Ast[left] to be median element

If we drop that ① $\underline{\quad N \quad}$ □ $\underline{\quad N \quad}$ □ becomes "median"

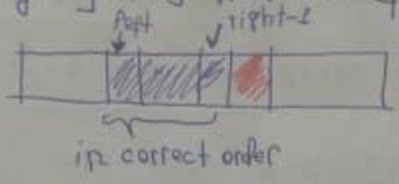
② $\underline{\quad N \quad}$ □ $\underline{\quad N+1 \quad}$ □ becomes "median"

In the ① median < Ast[left] would be satisfied ✗

Hence, ② is the only choice and $S_{k+1} = \{..., Ast[left], median, ...\}$

becomes satisfied → Ast[left] and median were consecutives. As Ast[right+1] is the first element not flipped after Ast[left], median will be put on Ast[right+1]

Assume "right" was incremented from Pst[right-1] to Pst[right] and Pst[right] was median of $S_K \cup \{Pst[right]\}$



median is middle element of S_K and we had median > Pst[right], $S_{K+1} = S_K \cup \{Pst[right]\} = \{..., Pst[right], median, ...\}$

has middle element $\Rightarrow$ ___ N ___ Pst[right] ___ N ___ ①

___ N ___ Pst[right] ___ N+1 ___ ②

when we drop Pst[right], "median" becomes middle of S_K

① ___ N ___ ___ N ___ → this is "median"

② ___ N ___ ___ N+1 ___ → this is "median"

In ① median < Pst[right] becomes true ⊗

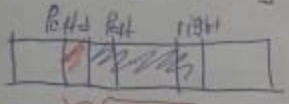
In ② $S_{K+1} = \{..., Pst[right], median, ...\}$ is becoming true

and we plug in "median" immediately after Pst[right] since it was middle element of S_K and it should come first after Pst[right] appeared on the array. Therefore

median will be put on Pst[right+1]

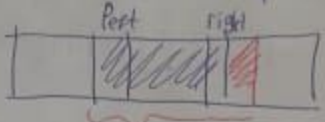
Hence, for median > Pst[right] we considered all previous possible cases (where cell was incremented or decremented) and found that **median = Pst[right+1]**

In the very similar case, one can see that if median < < Pst[Pleft], then median's correct position will be on Pst[Pleft-1]



now, all are in correct position

or



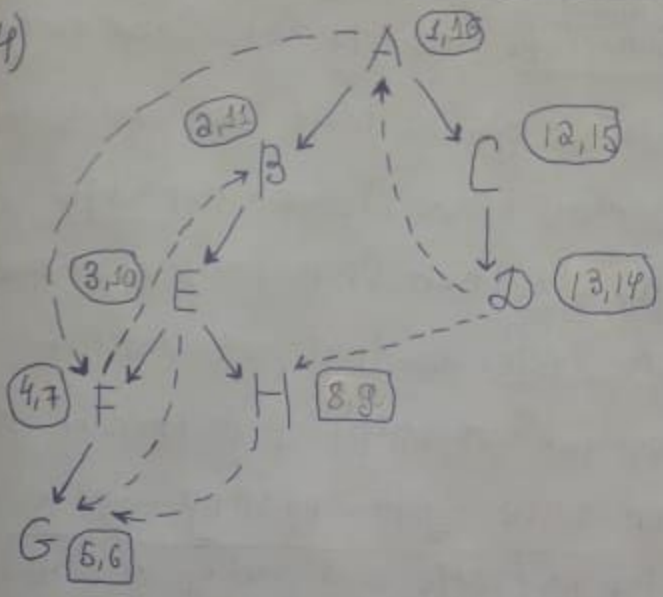
now, all of them are in correct position

This iteration will continue until when $i = n-2$ (from 0 to $n-2$) and process will stop adding new numbers into the array. So, basically, what we did was to add iteratively these n numbers onto correct position, and when we fulfilled all cells in the array, we will got an array of n numbers which are sorted in increasing order. As a termination for our claim, Pleft = 1 and right = n will enable us to obtain sorted n numbers in the "Pst" array.

Running Time → Linear time (all comparisons and usage of "machine" function were $O(1)$, only iterations (# of n iterations) enabled us to get Linear time. Space complexity → Linear

Rzayev Anar CS300-Homework #3 ID:20190188

4)



Here (a,b) denotes the order in which vertex was reached for the first time → a
the order in which vertex became dead-end → b

3) a) From the definition of simple cycle, we know that an edge (u,v) lies on a simple cycle ⇔ there exist at least one path from u to v such that it does not pass through the edge (u,v). (considering that given graph is undirected) However the latter statement is equivalent (⇔) to the fact that removing edge (u,v) will not disconnect our graph. Therefore, it is just saying that edge (u,v) will not be bridge ⇒
An edge (u,v) lies on a simple cycle ⇔ edge (u,v) is not a bridge
($x_1 \leftrightarrow x_2 \leftrightarrow \dots \leftrightarrow u \leftrightarrow v \leftrightarrow \dots \leftrightarrow x_k$) (another path was $u \leftrightarrow \dots \leftrightarrow x_c \leftrightarrow x_{c+1} \leftrightarrow \dots \leftrightarrow v$)
So, an edge of G is a bridge ⇔ that edge does not lie on any simple cycle of G
If cycle was $x_1 \leftrightarrow x_2 \leftrightarrow \dots \leftrightarrow u \leftrightarrow v \leftrightarrow \dots \leftrightarrow x_c \leftrightarrow x_1$, new path: $u \leftrightarrow \dots \leftrightarrow x_{c+1} \leftrightarrow x_c \leftrightarrow \dots \leftrightarrow v$ (not using edge (u,v))

b) From the definition, we can easily infer that cycle itself is biconnected component. Initially, let's show every edge which is not a bridge is in at least one of the biconnected components of G. Assume our edge (which is not bridge) is e, then from part a), we know e should lie on some cycle. If that edge was not part of any biconnected component of G, then it'll yield a contradiction since cycle containing edge e is itself biconnected ☒ edge e is part of at least 1 biconnected component of G

Now, our key claim will enable us to finish the problem:
Claim: If one biconnected component contains our edge, then it should also contain every cycle which is including our edge

From this claim, we infer that it would be impossible for 2 biconnected components to contain our edge, since every cycle from 1 biconnected becomes cycle of the other biconnected. In other words, we can say that let's suppose our edge e lies on 2 cycles from a different biconnected components

From our claim, biconnected component should have every cycle which includes our edge e. However, this's not true since cycles were from a different biconnected components ※ Hence, our claim finished the problem ⇒ every edge e (not bridge) is in exactly one of the biconnected components of G

■