500509944

COMP2123 Assignment 1

1. Line 2 takes O(1) time for initializing a primitive variable and line 3 takes O(n – k) to initialize an array. For lines 4-7, line 4 is a for loop which takes n – k iterations, line 5 takes O(1) time, line 6 takes k iterations and line 7 takes O(1) time. Therefore, since the algorithm utilizing a nested for loop,
   It takes $n – k + (n – k) * k = n – k + nk – k^2 = O(nk)$ time since nk dominates all terms here ($k < n / 2$).

2. a) We simply use an array implementation of a circular queue, with the addition of storing another variable to track the current seesaw sum (**seeSaw** = 0). Instead of storing the elements in the queue however, we store the elements as represented when used to calculate the **seeSaw** sum. We make modifications to both the enqueue and dequeue operations and add a new operation called see_saw() for calculating the seesaw sum.

   For enqueuing an element **x**, we first check if the queue is full in which case we return. We now check if length of the queue modulus 2 is equal to 0. If it is equal to 0, we add **x** to **seeSaw** and enqueue it. Else, we add $\frac{1}{x}$ to **seeSaw** and enqueue $\frac{1}{x}$.

   For dequeuing, we check if the queue is empty in which case we return, else we dequeue the element x and check if it is an integer. If it is we return it, else we return x^{-1} to get the correct element. We also decrement **x** from **seeSaw**.

   For our new operation see_saw(), we simply return **seeSaw** as our sum.

   b) The correctness of our new operation see_saw() depends on our variable **seeSaw** being correct after every enqueue or dequeue operation .
   Now initially, our variable **seeSaw** is 0 which is correct as our queue is empty.
   Now, when enqueuing, if the queue is full, we do not change **seeSaw** which is correct.
   No, we can have two cases. If the length of the queue modulus 2 is 0, we add the element to **seeSaw,** else we add $\frac{1}{element}$ to **seeSaw** which would result in the correct seesaw sum.
   When dequeuing, if our queue is empty, we do not change **seeSaw** which is correct.
   Now, we again have two cases, if our element is an integer, we return it, else we return it to the power of -1 which is correct since our queue should only return integer elements. Now we decrement **seeSaw** by element which is correct.
   Now, since our variable **seeSaw** correctly stores the seesaw sum before and after every enqueue/dequeue operation, our see_saw() operation will always return the correct value.

   c) Our data structure stores 4 variables which takes O(1) space and our array stores n elements which is O(n) space. Therefore, our data structure takes O(n) space.

Our see_saw() operation takes O(1) time since we just return a variable. Enqueuing and dequeuing both take O(1) time, since we only add O(1) operations in both operations.

3. a) Our elements in our data structure in addition to having a colour and value attribute, will have a new attribute which stores the running monochromatic stretch for that colour (**mc = 1).** We use two arrays which both serve as stacks, one to store our elements and the other which is used to keep track of the maximum monochromatic stretch at the corresponding index of the array (**stretches)**. Our data structure makes changes to stack operations push and pop, also adding an operation called maxMono().

   For our push operation on an element **x**, we first check if the stack is empty, if it is we do not change **x's** mc value . If it is not and stack is also not full, then we first get the element at the top of the stack **y**. If the colour of **y** is equal to the colour of **x**, we set **x's** mc to **y's** mc plus 1. Now we check if **x's** mc is greater than the top element in **stretches** . If it is greater than that element, we push **x's** mc onto **stretches**. If it is not, then we push the top element onto **stretches** again**.** Finally, we push **x** on our stack.

   For our pop operation. We simply just call pop on both **stretches** and our stack.

   For our maxMono() operation, we simply return the top element in **stretches.** If, however **stretches** is empty then we just return 0.

   b) The correctness of our new function maxMono() relies on our stack **stretches** storing the correct max monochromatic stretch at each index.
   Now initially, our stack is empty and so is **stretches**, therefore maxMono() returns 0 which is correct.

   Now if we push an element, we can have two cases, the element being pushed has a greater mc value than the top element in **stretches** or not.
   If it has a greater mc value, then we push that value onto **stretches** which is correct. If it does not, then we just push the top element onto **stretches** again which is also correct as our max stretch has not changed.

   If we pop an element, we also pop from **stretches** which also updates our maxMono() operation correctly.

   Since we have shown that **maxStretch** is correct before and after every push and pop operations, our maxMono() operation will clearly return the max monochromatic stretch which is always stored at the top of our stack **stretches.**

   c) Our data structure two arrays of size n which takes 2n space. Therefore, our data structure takes O(n) space.

   Our maxMono() operation takes O(1) time as we simply just call peek at **stretches**, our push and pop operations only add O(1) operations and regular stack operations take O(1) time so they each only take O(1) time.