Solution 1. Line 2 takes $O(1)$ time, while Line 3 takes $O(n-k)$ time, which is $O(n)$. The inner for loop (lines 6-7) is always executed k times. Each iteration does $O(1)$ work, so the overall running time of the inner loop is $O(k)$. The outer loop (lines 4-7) is executed $n-k$ times. Each iteration does $O(k)$ work, so the overall running time of the outer loop is $O((n-k)k)$, since $k < n/2$ this amounts to $O(nk)$ time. Therefore, the overall running time is $O(nk)$.

Solution 2.
a) We modify the linked-list implementation of queues covered in lectures. In order to implement SEESAW in $O(1)$ we need to keep track of the the sum of even positions and the sum of reciprocals of odd positions:

$$\text{sumEven} = \sum_{i \text{ is even}} Q_i \quad \text{and} \quad \text{sumRecipOdd} = \sum_{i \text{ is odd}} \frac{1}{Q_i}. \tag{1}$$

With this information at hand the implementation of SEESAW is trivial: Simply return sumEven + sumRecipOdd.

Initially, when we create an empty queue, both attributes are set to 0. Maintaining (1) when we enqueue an element is straightforward: we would need to update either sumEven or sumReciprocalOdd depending on the parity of the index of the newly inserted element. This alone won't work, however, as dequeuing an element flips the parity of the remaining elements[1] . To overcome this hurdle we need to keep the plain sum and the reciprocal sum for both even and odd elements, namely,

$$\text{sumOdd} = \sum_{i \text{ is odd}} Q_i \quad \text{and} \quad \text{sumRecipEven} = \sum_{i \text{ is even}} \frac{1}{Q_i}. \tag{2}$$

Enqueue is still straightforward to implement: Just modify the attributes to account for the incoming element according to its parity.

```
1: function NEWENQUEUE(e)
2:     ENQUEUE(e)
3:     if SIZE( ) − 1 is even then            ▷ element e has index SIZE() − 1
4:         sumEven += e
5:         sumRecipEven += 1/e
6:     else
7:         sumOdd += e
8:         sumRecipOdd += 1/e
```

Dequeue also needs to update the even attributes to account for the departing element, and flip the roles that even and odd positions play.

```
1: function NEWDEQUEUE(e)
2:     DEQUEUE(e)
3:     sumEven = sumEven - e
4:     sumRecipEven = sumRecipEven - 1/e
5:     sumEven, sumOdd = sumOdd, sumEven
```

[1]If we dequeue Q_0 going from queue state Q to queue state Q', then Q_1 becomes the new Q'_0, Q_2 becomes the new Q'_1, etc.

```
6:      sumRecipEven, sumRecipOdd = sumRecipOdd, sumRecipEven
```

b) The correctness of SEESAW hinges on the invariant that equations (1) and (2) hold throughout the execution. It is clear that NEWENQUEUE preserves this invariant (updating the even or odd attributes depending on the parity of the new element). Similarly, NEWDEQUEUE first removes the contribution of the outgoing element to the even attributes (Q_0, the first item of the queue, contributes towards SumEven and SumRecipEven) and then flips the roles of even and odd attributes.

c) Clearly SEESAW runs in $O(1)$ time and the additional work done by NEWENQUEUE and NEWDEQUEUE does not add to the $O(1)$ time complexity of the underlying ENQUEUE and DEQUEUE operations.

Finally, we note that we only use additional $O(1)$ space (the four attributes) and the underlying queue takes $O(n)$ space, so $O(n)$ space overall.

Solution 3.

a) We augment the linked list implementation of a stack from the lectures. First we augment the stack entries to maintain a currStreak attribute, which captures for entry e the maximum length of a monochromatic stretch starting before e and ending at e. Just before pushing a new entry e into the stack, we compare its colour to the top entry t, if they are different we set $e.currStreak \leftarrow 1$ otherwise, if they have the same colour, we set $e.currStreak \leftarrow t.currStreak + 1$.

In addition to currStreak, we also maintain for each entry e an additional maxStreak attribute that holds the maximum currStreak value of stack entries from the bottom of the stack up to e. With this information in hand, implementing MAXMONO is trivial: If the stack is empty return 0, otherwise, return TOP().maxStreak.

All that is left is to further modify the push operation to maintain these attributes.

```
1:  function NEWPUSH(v)
2:      if ISEMPTY() then
3:          $e' \leftarrow$ NEWENTRY($v, 1, 1$)                ▷ $(value, currStreak, maxStreak)$
4:      else
5:          if TOP().value.colour = v.colour then
6:              $nextStreak \leftarrow$ TOP()$.currStreak + 1$
7:          else
8:              $nextStreak \leftarrow 1$
9:          $nextMaxStreak \leftarrow \max(nextStreak, f.maxStreak)$
10:         $e' \leftarrow$ NEWENTRY($v, nextStreak, nextMaxStreak$)
11:     PUSH($e'$)
```

Finally, NEWPOP() is trivial: Ignore the additional attributes and return POP()$.value$

b) We maintain the invariant that the maxStreak attribute at the top of the stack is the maximum length monochromatic stretch in the entire stack and currStreak the maximum length of the monochromatic stretch ending at that entry. When the first element is pushed, this indeed holds, since that element is by definition a stretch of length 1. Assuming that the invariant holds before a push into a non-empty stack, we observe that it also holds afterwards, as we increment the previous currStreak value if needed, and compare the previous maxStreak value to the length of currStreak ending at the new element.

The invariant ensures that MAXMONO returns the correct value when the stack is non-empty, while the routine is trivially correct when the stack is empty.

c) All the underlaying stack operations take $O(1)$ time as per the lecture. On top of that, we only carry out a constant number of basic operations (i.e., assignments, comparisons, arithmetic operation, etc) per function. Therefore, the overall running time of each function is $O(1)$ as required.

Since we use a linked list and constant additional space per entry (storing two additional integers per entry), we ensure that the size of our stack is proportional to the number of elements we have pushed. In other words, when our data structure contains n elements, it uses $O(n)$ space.