Question 1

Generally, students did well on this question. We expected a somewhat rigorous solution for this question as this question was relatively straightforward to solve. Hence, students were expected to carefully analyse the applicability of the techniques that they deploy for solving the problem. Also, as mentioned in the class and the description before the submission link, **your algorithm should be described in plain English**. Pseudo code or any code written using a programming language should **only** be used to complement the understanding of the proposed algorithm design (i.e., the answer should be sensible even if the code snippets are removed from the answer).

1.1

- Most students received 9 to 12 marks for this question.
- The most common mistake observed was the lack of analysis of prerequisites to run a binary search. Note that binary search works only on a sorted list of elements. Hence it is important to identify that V in terms of x is a strictly increasing function.
- For students who decided to use a linear search on the range $[1, \log(V)]$, we expected an explanation of the selection of the upper bound.

1.2

- A fair number of students managed to get 18 marks for this question!
- Among some of the correct answers, some answers did not provide an explanation for the selection of $\log(V)$ as the upper bound. They would have lost around 6 marks for the missing explanation.
- Some students referred to materials that are not covered in this course or its prerequisite courses. Such material, as outlined in the course Moodle page should be reproduced by the student in their own words, along with a citation in any format.
- In this course, we are analysing the worst-case performance. Hence, techniques that improve amortised performance should not be proposed as valid solutions.
- Some students proposed "binary search-like" searching algorithms that are based on the divide-and-conquer approach with different methods to compute the pivot point (taking the square root, interpolating the boundary values, etc). Though these approaches seem to be very convincing, almost all the time, their time complexities are logarithmic in the size of the input. So, any student who attempted such a searching algorithm on the input $[1, V]$ have likely failed to meet the expected time complexity and therefore would have received a mark in the range of 2 to 8 based on the merit of their answer.

Question 2

Please note that requesting a mark review means your mark may be adjusted downwards. Marks given were generally very generous for this question.

Overall students performed well on this question.

2.1

We firstly note that the array is *sorted*. It is then apparent that all occurrences x_i will appear as a single contiguous sub-array of X.

Iterate through X. We can keep track of precisely how many times an element appears by keeping track of where a new distinct integer appears. If a new distinct integer starts at index i and the next new integer begins at index j, then we know there are precisely $j - i$

many occurrences of integer x_i as we have just encountered all the possible occurrences of x_i since it appears in a single contiguous sub-array. We can find $j - i$ by doing a linear scan over X, such that we increment a variable count each time we see the same value and upon encountering a different value, count will contain $j - i$ and we can reset count and continue. The number of occurrences for each distinct integer computed with a single linear scan of X; since there are n many comparisons to make, the algorithm runs in $O(n)$ time.

Overall students performed well on this sub-question. Most students who had a correct idea achieved 5 or 6.

- Many students lost marks for failing to justify the correctness of the algorithm. We were particularly lenient on this. To justify why the linear scan works, one must express the following logic or equivalent. "We firstly note that the array is *sorted*. It is then apparent that all occurrences x_i will appear as a single contiguous sub-array of X." A more casual expression of this might look like "since it is sorted, all of the same element appear next to each other", which was sufficient for our standards.
- Some students assumed that the values were bound by n or some equivalent. This is not true and reduces the problem to something almost trivial. Little to no marks were awarded.
- Some students assumed hash-tables had a O(1) worst case. This also reduces the problem to something almost trivial. Little to no marks were awarded.
- Some students proposed defining a new array and using the scanned values as keys. E.g. create array A and then put the count of X[i] into A[X[i]]. This is effectively defining a hash-table which has no collisions and is only possible with unbounded memory. On multiple occasions in the forums we have stated this is not allowed, and should be deduced by the worst case of hash-tables being O(n). These solutions were either incorrect, or made the assumption that the given values were bound by n which again, forms a reduction to a trivial problem. Hence little to no marks were awarded.

2.2

Define and denote our hash table as H and begin by initialising it to be empty. Scan each element y_i in Y through a linear scan and increment $H[y_i]$ by one. (Creating an entry if it does not exist) Once we finish performing the linear scan of Y, each key of H will contain the number of occurrences of that particular key in Y.

This runs in $O(n)$ *expected* time because, for each element in Y, we perform an expected $O(1)$ insertion and possibly an expected $O(1)$ creation. Since there are n elements, we perform n $O(1)$ expected time operations and so, our final time complexity is $O(n)$ in the expected time.

Overall students performed well on this sub-question. Most students who had a correct idea achieved 12.

- Some students again assumed that the values were bound by n or some equivalent. This is not true and reduces the problem to something almost trivial. HOWEVER if the solution generalised to cases which did not require the aforementioned assumption, we were lenient and did not penalise this. Otherwise, this was awarded little to no marks.
- Some students again proposed defining a new array and using the scanned values as keys. E.g. create array A and then put the count of X[i] into A[X[i]]. This is effectively defining a hash-table which has no collisions and is only possible with unbounded memory. On multiple occasions in the forums we have stated this is not allowed, and should be deduced by the worst case of hash-tables being O(n). These solutions were either incorrect, or made the

assumption that the given values were bound by n which again, forms a reduction to a trivial problem. However, we recognise that such a solution contains relevant logic and progress to a correct solution. These solutions were awarded some but very few marks.

2.3

Note that Z is sorted, so our approach here will be to binary search for the last occurrences of each distinct integer. It is also apparent that all occurrences z_i will appear as a single contiguous sub-array of Z. The first distinct integer occurs at index 1. To find the last occurrence of any integer, we perform a binary search on Z.

The function on which we binary search will be as follows.

$$f(i) = \begin{cases} 0 & \text{if } Z[i] < X \\ 1 & \text{if } Z[i] = X \text{ AND } Z[i+1] \neq X \\ 2 & \text{otherwise} \end{cases}$$

Given that Z is sorted, this function $f(i)$ will be monotonic as elements at indices larger than the last occurrence of a particular integer will be greater. (And the reverse will be smaller). Therefore searching for $f(i) = 1$ would return us the last occurrence of any integer X.

Using this, we can begin by binary searching for the last occurrence of Z_1. The index of the last occurrence minus 0 must be the number of occurrences of Z_1. We know the next distinct element then appears at the next index and we can repeat for all k contiguous sub-arrays of distinct elements.

We observe that we are performing a binary search on Z for *each* distinct contiguous sub-array of like elements. The time complexity for each binary search is $O(\log n)$ and since there are k distinct integers, we perform k $O(\log n)$ operations which gives us the desired time complexity of $O(k \log n)$.

Some students struggled in this section.

- Many students considered the idea of finding the last occurrence of any particular integer, but did not state how the non-trivial binary search worked. This constituted 4 marks and little to none of those would be awarded if the binary search step was not detailed.
- Many students also do not state why we can binary search this in the first place. To justify the correctness of your algorithm constitutes a considerable part of the assessment of this course.

Question 3

This question was found to be difficult for many students, with the average mark sitting at around 11/20.

- Many students correctly identified that the inversion counting algorithm from lectures was useful here.
- Many students correctly identified that they could not simply apply the inversion counting algorithm, since it only counts inversions in one array. However, there were several attempts to use the algorithm on both array A and array B separately and combine the results somehow – doing this completely ignores the relationship between the arrays and cannot properly count inversions.
- The area where most students lost marks was justification – it is expected that you justify why your algorithm finds the correct answer. In this question, we require both that you justify

why your pre-processing doesn't affect the number of inversions, and also why your divide and conquer approach doesn't miscount. Note that if you referenced the inversion counting from lectures, then you may assume it is correct and not justify it, but pre-processing still required justification.

- There were many solutions using hashmaps and hashtables. These data structures have poor worst case guarantees ($O(n)$ lookup and insertion), so realistically should be avoided for this question, since we don't ask for average complexity. If you did include these data structures, you were required to explain why the lookup/insertion operations remain $O(1)$ (i.e. why no collisions occur).
- The number of inversions between two arrays can possibly be $O(n^2)$ (e.g. A is sorted, B is reverse sorted) – this is a good sanity check for your algorithm. If you increment a counter for every inversion you find then either you don't count all of them, or your algorithm could be $O(n^2)$. Several solutions did so.
- It is not sufficient to state time complexity at the end of your solution. We gave you the time bound in the question, after all, so expect you to explain why your program hits this time bound. Several solutions failed to include time complexity justifications.
- Many students attempted to get around the $O(n^2)$ nature of checking for every inversion by binary searching array B for the element in array A. Binary search can only be used on sorted data.

Question 4

Overall, this was a pretty difficult question. Students did okay on part 1 but many struggled with part 2. Quite a few students attempted part 1 but did not attempt or did not make a serious attempt at part 2. Overall average marks were around 9/20

4.1 Most students who attempted this question got most or all of the marks, using a prefix sum approach. No other valid approaches were used

- one common error was using a hash map instead of an array. Hash maps that map from index to value are strictly worse than arrays, as they have far more overhead and only average case $O(1)$ access. Students are encouraged to only use hash maps when a key-value pair is necessary, and to describe what the keys and values are.
- The other most common error was failing to justify time complexity of their approach.

4.2 This question was significantly more difficult. It required a fair amount of insight into the nature of the problem. It also had an $O(n)$ solution that a fair number of students did manage to find.

- The most common incorrect solution, by far, was an attempt to use a variation of merge sort/divide and conquer to solve this problem. I think the only reason for that it is that they saw $O(nlog(n))$ and immediately thought of merge-sort, then tried to force this problem into a merge-sort shaped problem. Not only does this approach not solve the issue, but sorting the array would destroy the structure that is being observed, meaning that merge-sort cannot be part of a solution that works.
- Another common issue was approaches that tried to consider sub-arrays one by one, which meant that not all sub-arrays could be considered, as there are $O(n^2)$ sub-arrays, so there is no way to consider them all in less than $O(n^2)$ time individually. All these approaches either had incorrect complexity analysis, or were non-exhaustive in their search.

- Finally, a disappointingly high proportion of students clearly misread the question. There were many approaches that were trying to find sub-arrays with more red flowers **inside** the sub-array than yellow flowers **inside** the sub-array, when the question pertained to yellow flowers **outside** the sub-array. It is a subtle difference when skim reading, but assignment questions should be read carefully.
- Of the students who correctly solved the problem, a roughly equal number used solution 1 (binary search) and solution 2 (counting sub-arrays based on size alone).
- The most common errors among those who used a binary search solution were forgetting to update the actual count of qualifying sub-arrays, and not stating what their algorithm would return.
- the most common mistake for students using the $O(n)$ approach was failing to justify the claim that $size > yellow\_count$ is equivalent $red\_inside > yellow\_outside$.