Find a specific index integer.
Quick Select / Algo on unsorted Array:
Find Median => (The ((N+1)/2) Integer)
in O(N)

Sorting

Sorting

Sorting Algorithm	Best Case		Worst Case	
	Swaps	Comparisons	Swaps	Comparisons
Selection Sort Not stable	0 – already sorted	$O(n^2)$ – Still need to check	O(n) – always n-1 swaps	$O(n^2)$ – always n
Insertion Sort	0 – already sorted	O(n) – compare O(n) times and break	$O(n^2)$ – reverse order hence constant shifting	$O(n^2)$ – constant comparing
Bubble Sort w/o flag	0 – already sorted	$O(n^2)$ – no flag hence need to check all	$O(n^2)$ – reverse order	$O(n^2)$ – check all
Bubble Sort w/ flag	0 – already sorted	O(n) – early termination with flag	$O(n^2)$ – reverse order	$O(n^2)$ – reverse order or smallest element at the back
Merge Sort Not in-place	0 – all copying, which takes O(n log n), but no swaps	≤O(n log n) – comparisons stop once one sublist is fully copied in	0 – all copying, which takes O(n log n), but no swaps	≤O(n log n) – comparisons stop once one sublist is fully copied in
Quick Sort Not stable	O(n log n) - ~log n levels of n/2 swaps	O(n log n) – log n levels of n comparisons	$O(n^2)$ – reverse order with O(n) swaps for n levels O(n) – if we are talking about the worst case of an already sorted array, where we have n levels of 1 swap (with itself).	$O(n^2)$ – n levels of comparing n elements
Radix Sort Not in-place	0 – O(kn) copying	0 – non-comparison based sort	0 – O(kn) copying	0 – non-comparison based sort

K = no of digits in largest number
If number large, cannot do in O(N)
↓
good for bounded integers (non -ve and within a small range)

---

6. An array is **k**-*sorted* if each element is at most **k** positions away from its sorted position. Which of the following sorting algorithms (as given in the lecture notes) can sort a **k**-*sorted* array of size **N** in **O(kN)** time?

- Insertion sort
- Optimised bubble sort
- Selection sort

a) (i) and (ii)
b) (i) and (iii)
c) (i), (ii), and (iii)
d) None of the algorithms will run in *O(kN)* time.

For insertion sort, each element will move at most k times (consecutive swaps) within the algorithm. This is the same for Optimised bubble sort, which will run for k iterations, followed by the (k + 1) "check" iteration. (In fact, if you know it is k-sorted there is no need for the (k + 1) iteration).

This question tests if students understand the underlying mechanisms of how the sorting algorithms work. For students who have difficulty with this problem, you are advised to revisit the invariants of each algorithm, i.e. how they "grow" the sorted area during the algorithm.

→ O(N) for K-sorted Array,
Use Insertion / Bubble Sort

Radix Sort Optimisation
If we want know the range of numbers to be between [1, N^3], we can actually use radix sort to get an O(N) sort, with fewer 'digits' d and more buckets instead.

Traditionally, we use 10 buckets, requiring close to 3 $\log_{10} N$ passes, one for each digit. Each pass places all N elements into the buckets and then extracts them. This results in an O(N log N) time sort, as d = O(log N).

We can instead reduce the number of 'digits' till d = 3. Each 'digit' of increased size must be able to take N values, in [0, N-1]. This means that radix sort now needs N buckets and 3 passes, resulting in an O(N) sort.

Why should we not reduce the number of 'digits' till d = 1? If we do so, There are now b = N^3 buckets, increasing the time complexity of radix sort. To be precise, the time complexity of radix sort is O(d(N+b)). We usually write O(dN) or O(N) when we can be certain that d and b are bounded by some small constant.