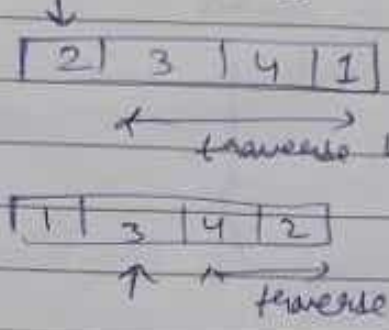
8)(a) As we have to sort DS 1 in place we can not select merge sort as merge sort is not in place.

In selection sort we will traverse the array ~~k times & find the index with the element~~ multiple times and compare all the elements in ~~this~~ the right of the current position for which we are ~~taking~~ finding the right element.

eg



So in worst case we swap the elements at max K times

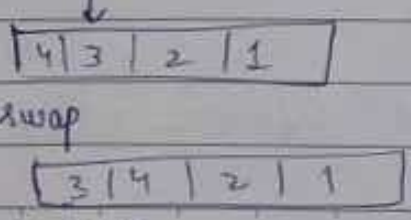
~~So, at m~~ For comparison it takes constant time, as stated in question it takes constant time to get element from index.

Selection Sort time complexity for this question = $K * (\text{time for swapping})$

$= K * (K \log K)$

$= O(K^2 \log K)$

In insertion sort worst case we will have to swap the elements ~~in~~ l ~~no~~ number of times where l represents the ~~no~~ number of elements in the left of position



So, number of swaps at most $= 1+2+\ldots+(k-1)$

$= \frac{(k-1)\,k}{2}$

$= O(k^2)$

Similarly as before comparison takes constant time.

Insertion sort time complexity $= k^2 \times$ (time for swapping)

$= k^2 \times (k \log k)$

$= O(k^3 \log k)$

So for this it is better to choose selection sort.

(b) In selection sort we will have worst case complexity as

Comparisons $= (n-1) + (n-2) + \ldots + 1 = \frac{(n-1)\,n}{2} = O(n^2)$ comparisons

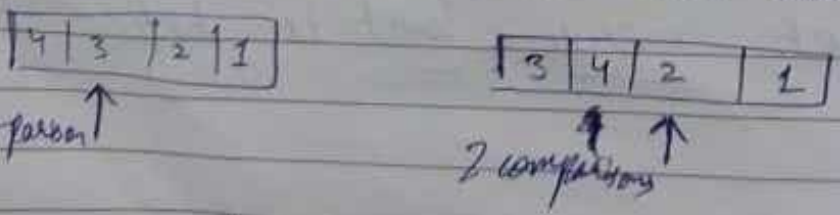
Time complexity $= O(n^2 \log n)$

(for selection sort)

n^2 ↓ number of comparisons

$\log n$ → time for a comparison

In insertion sort we will worst case complexity as,



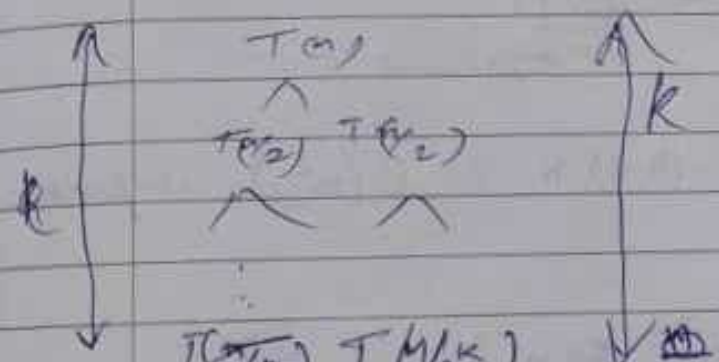
Comparisons $= 1+2+\ldots+n-1 = \frac{(n-1)\,n}{2} = O(n^2)$

For insertion sort
time complexity $= O(n^2 \log n)$

- In Merge Sort

breaking down still takes $\log n$ time but in joining (merge) function we compare n times & time for comparing is $\log n$ so, it takes $O(n \log n)$

So overall time complexity after taking n comparisons & $\log n$ height of recurrence tree we get time complexity $O(n(\log^2 n))$

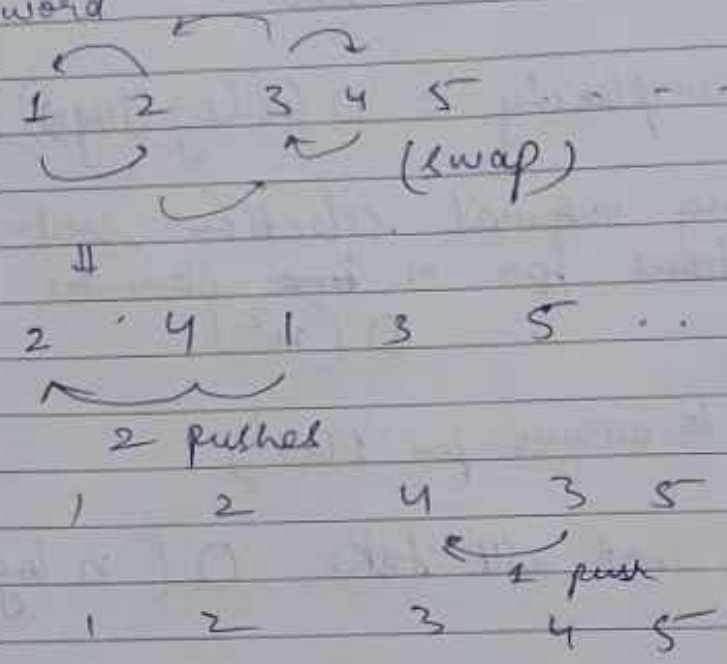


As $n \log^2 n < n^2 \log n$

we get merge sort is better for this case.

(c) For this case at (log(log n)) swaps made b/w adjacent pairs

In insertion sort worst case we would have to make (log(log n)) swaps while going through the whole word



(3 swaps ⇒ 3 pushes)

(tried different cases still got same result)

~~(time for traversing whole array)~~

So time complexity for insertion sort = $O(n + (\log(\log n)))$

$= O(n)$

time for traversing whole array

time for swaps.

Assuming selection sort updated for this case:

An element can at max be displaced by $(\log(\log n))$ of current position

So we will have to check $(\log(\log n))$ positions for n elements

time complexity = $O(n(\log(\log n)))$

Assuming normal selection sort: it will check for n elements

$O(n^2)$

(similar to answer for 1(a))

Merge sort will take $O(n \log n)$ time

So best is insertion sort for this particular case.

A2)(a) Stack functions implemented: push, pop, empty, top, size, display

⇓

shows whole stack

Queue functions implemented :-

Enque, deque, frontVal, size, isEmpty, display (comments in code)

A2)(b) First for loop finds maximum element & its index

$O(n)$ time for this

Next while loop travels n elements and at max pushes or pops n elements

$O(2n)$ time for this $O(n+2n)$

Whole code time complexity = $O(n)$

A3) In this the loop will not run more than 101 iterations in the worst case so as we increase n it doesnt effect the constant time.

So time complexity = $O(1)$

~~(doubt used, asked Q4 in 4th question)~~

A4) First ~~we~~ I have sorted radiant & dire teams using merge sort so $O(n \log n)$ time

Then fight function takes $O(n)$ time in worst case where each fighter fights at least once.

Again merge sort but this time on basis of order (index) so $O(n \log n)$ time

Then output function just prints all remaining heroes.
$O(n)$ time

Total time $= 2n \log n + 2n = O(n \log n)$

Time Complexity = $O(n \log n)$