

# CS2040S Tutorial 2

Recap

# Sorting

Stability : whether same values gets swapped from its original order.

Generally algorithms are not stable when your algorithm swaps elements that are not next to each other.

# Sorting

<b>Algo</b>	Bubble sort
<b>In-place?</b>	Yes
<b>Stable?</b>	Yes - only swap when elements are not equal
<b>Best runtime</b>	$O(n)$ - already sorted
<b>Average runtime</b>	$O(n^2)$
<b>Worst runtime</b>	$O(n^2)$
<b>Loop Invariant</b>	At the end of iteration $j$ , the biggest $j$ items are correctly sorted in the final $j$ positions of the array.
<b>Remarks</b>	

5 4 3 1 2 | 

# Sorting

<b>Algo</b>	Insertion sort
<b>In-place?</b>	Yes
<b>Stable?</b>	Yes
<b>Best runtime</b>	$O(n)$ - Already sorted
<b>Average runtime</b>	$O(n^2)$
<b>Worst runtime</b>	$O(n^2)$ - inversely sorted
<b>Loop Invariant</b>	At the end of iteration $j$ : the first $j$ items in the array are in sorted order.
<b>Remarks</b>	Idea is to <u>insert</u> the first element in the unsorted subarray into the sorted subarray

6 5 3 1 8 7 2 4

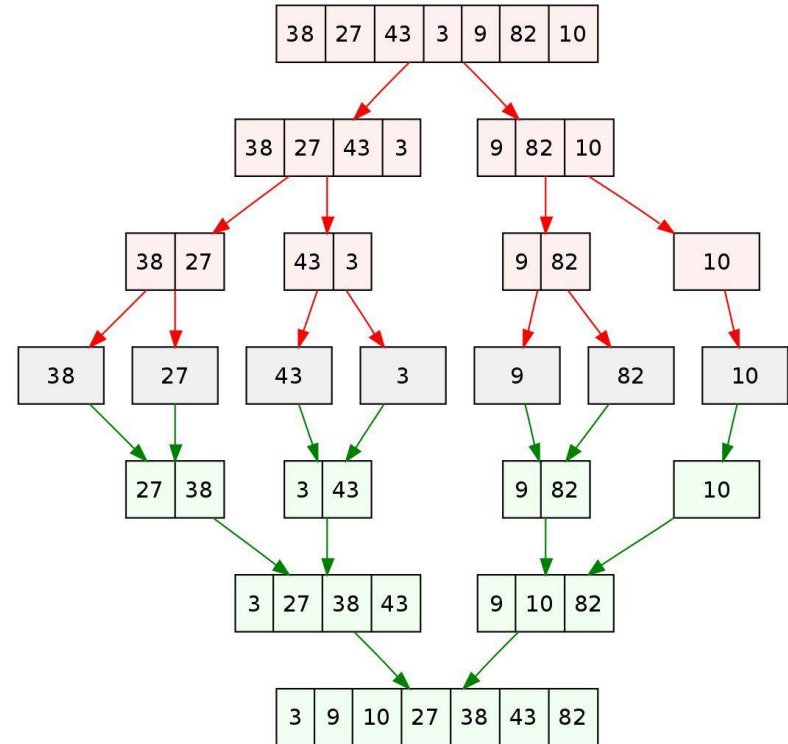
# Sorting

<b>Algo</b>	Selection sort
<b>In-place?</b>	Yes
<b>Stable?</b>	No
<b>Best runtime</b>	$O(n^2)$
<b>Average runtime</b>	$O(n^2)$
<b>Worst runtime</b>	$O(n^2)$
<b>Loop Invariant</b>	At the end of iteration $j$ : the <u>smallest</u> $j$ items are correctly sorted in the first $j$ positions of the array.
<b>Remarks</b>	Idea is to <u>select</u> minimum element in the unsorted sub array

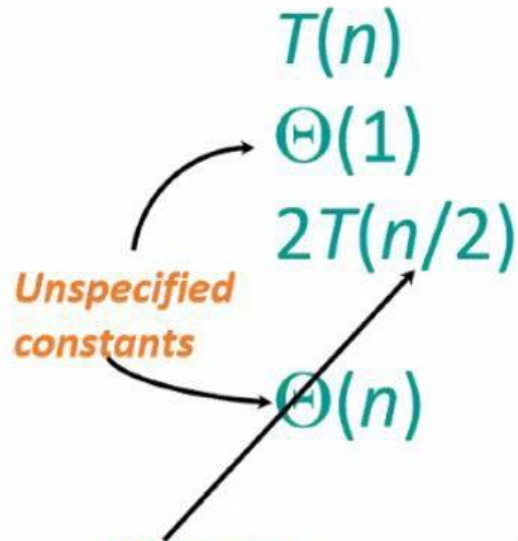
	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Sorting

<b>Algo</b>	Merge sort
<b>In-place?</b>	Usually no
<b>Stable?</b>	yes
<b>Best runtime</b>	$O(n \log n)$
<b>Average runtime</b>	$O(n \log n)$
<b>Worst runtime</b>	$O(n \log n)$
<b>Loop Invariant</b>	
<b>Remarks</b>	Divide and conquer.



# Analyzing merge sort



The diagram shows the recurrence relation  $T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$ . A curved arrow labeled "Unspecified constants" points from the  $\Theta(1)$  and  $\Theta(n)$  terms. A straight arrow points from the  $\Theta(n)$  term to the "Sloppiness" note below.

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$$

**Sloppiness:** Should be  $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ , but it turns out not to matter asymptotically.

## MERGE-SORT $A[1 \dots n]$

1. If  $n = 1$ , done.
2. Recursively sort  $A[1 \dots \lceil n/2 \rceil]$  and  $A[\lceil n/2 \rceil + 1 \dots n]$ .
3. **"Merge"** the 2 sorted lists



# Sorting(Quicksort)

<b>Algo</b>	Quick sort
<b>In-place?</b>	Yes
<b>Stable?</b>	No
<b>Best runtime</b>	$O(n \log n)$
<b>Average runtime</b>	$O(n \log n)$
<b>Worst runtime</b>	$O(n^2)$
<b>Loop Invariant</b>	All elements on the left of the pivot are smaller than the pivot, vice versa for the right side
<b>Remarks</b>	

6 5 3 1 8 7 2 4

# Tutorial

## Qn2(a)

How would you implement insertion sort recursively?

Analyse the time complexity by formulating a recurrence relation.

## Qn2(a) (Answer)

Recursively sort the array from index 0 to  $(n - 1)$ . Then, insert the last element into the sorted subarray.

Recurrence relation:  $T(n) = T(n - 1) + O(n)$ .

Time complexity  $O(n^2)$ .

## Qn2(b)

Consider an array of pairs  $(a,b)$ . Your goal is to sort them by  $a$  in ascending order first, and then by  $b$  in ascending order.

Do you Merge sort first or Selection Sort

## Qn2 (b) (Answer)

Consider the stability of each sort.

Merge Sort: Stable

Selection Sort: Not stable

Selection Sort to sort pairs by their  $b$ , then Merge Sort to sort pairs by their  $a$ .

Selection Sort is unstable  $\rightarrow$  if we used it to sort pairs by their  $a$ , we will not necessarily get ascending  $b$  for pairs with same  $a$  value.

## Qn2(c)

We have learned how to implement MergeSort recursively. How would you implement Merge-Sort iteratively? Analyse the time and space complexity.

## Qn2 (c) (Answer)

Increment sorting by power of 2

1st iteration = 2

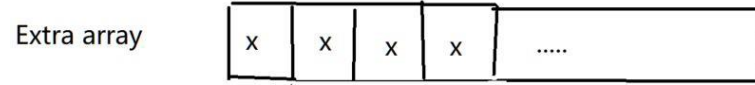
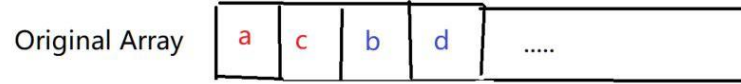
2nd iteration = 4

...

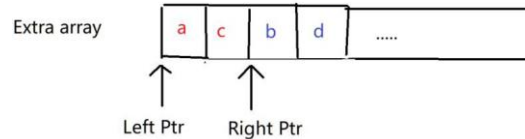
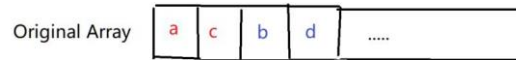
Nth iteration =  $(2^n)$



## Qn2(c) (Answer)

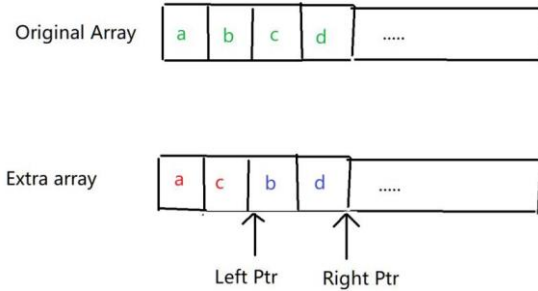


Copy first half of the array into left array, second half into right array and set a left and right pointer to keep track



## Qn2 (c) Answer

Compare elements with the pointer, writing the smaller element of the two into the original array



Repeat till the end of the array

After than repeat for size \* 2 ...

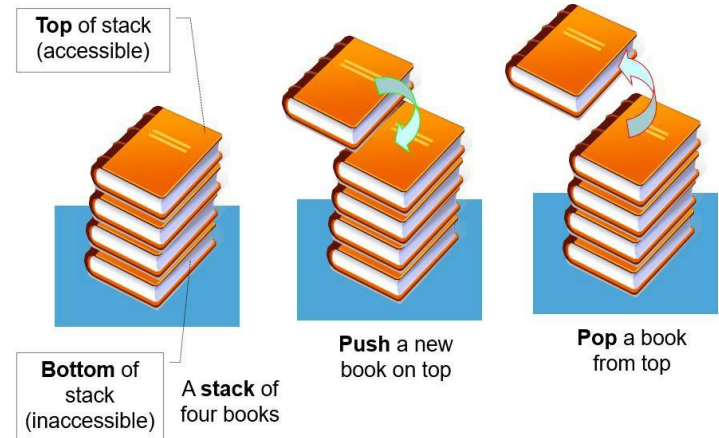
# Stack

Last in first out

`push(element)` -> pushes an element to the top of the stack.

`pop()` -> removes the top element of the stack.

`peek()` -> returns the last element added to the stack



# Queues

First in first out

2 main functions

enqueue(element) -> Adds an element to the back of the queue.

dequeue() -> Removes the element at the front of the queue.

peek() -> Returns the next item to be dequeued



## Qn3 a)

How would you implement a stack and queue with a fixed-size array in Java?  
(Assume that the number of items in the collections never exceed the array's capacity.)

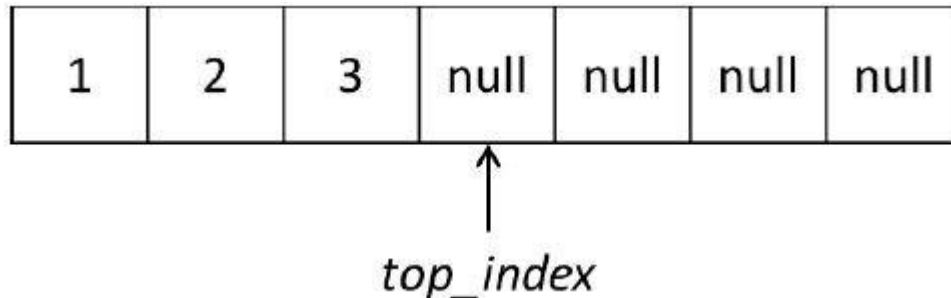
## Qn3 a)

Maintain a variable `top_index` for the top of the stack (with initial value 0):

`push` → add element at `top_index` and increment `top_index`

`peek` → return element at `top_index - 1`

`pop` → decrement `top_index` then return element at `top_index`



## Qn3 a)

Queue: Maintain an head and tail index on the array.

On enqueue, set element to the  $\text{arr}[\text{tailIdx}]$  ,  $\text{tailIdx} + 1$

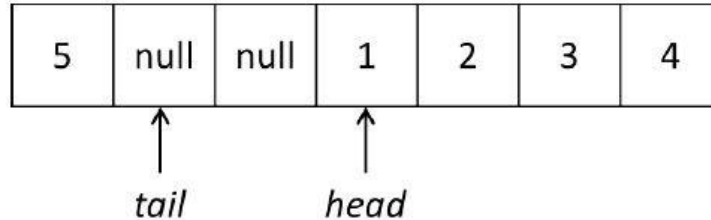
On dequeue, return value at  $\text{arr}[\text{headIdx}]$  , set  $\text{arr}[\text{headIdx}] = \text{null}$ , index -1

## Qn3 a)

But if we increment head for each enqueue  $\rightarrow$  we will eventually reach index out of bounds!

Solution: “wrap around” the array  $\rightarrow$  incrementing at `array.length` gives 0

(i.e. instead of `x++`, we use `(x+1) % array.length`)





## Qn3 (b)

A Deque (double-ended queue) is an extension of a queue, which allows en-queueing and de-queueing from both ends of the queue. So the operations it would support are enqueue front, dequeue front, enqueue back, dequeue back. How can we implement a Deque with a fixed-size array in Java? (Again, assume that the number of items in the collections never exceed the array's capacity.)

## Qn3 (b) Answer

Same as before,

`_front` ops will manipulate front index, but `_back` ops will manipulate tail index.

## Qn3 c)

What sorts of error handling would we need, and how can we best handle these situations?

## Qn3 c) Answer

For enqueue and push:

- Edge case is when the capacity of the array is reached.
- Can return false upon reaching capacity

For dequeue and pop:

- Edge case is operating on empty collection (in this case, queue).
- Can throw exception

## Qn3 d)

A set of parentheses is said to be balanced as long as every opening parenthesis "(" is closed by a closing parenthesis ")". So for example, the strings "()()" and "(())" are balanced but the strings ")()()" and "(" are not. Using a stack, determine whether a string of parentheses are balanced.

## Qn3 d) Answer

Traverse the string, push the opening brackets '(' onto the stack, then pop one out whenever a closing parenthesis ')' is seen.

String is unbalanced if:

There's attempt at popping on empty stack, OR

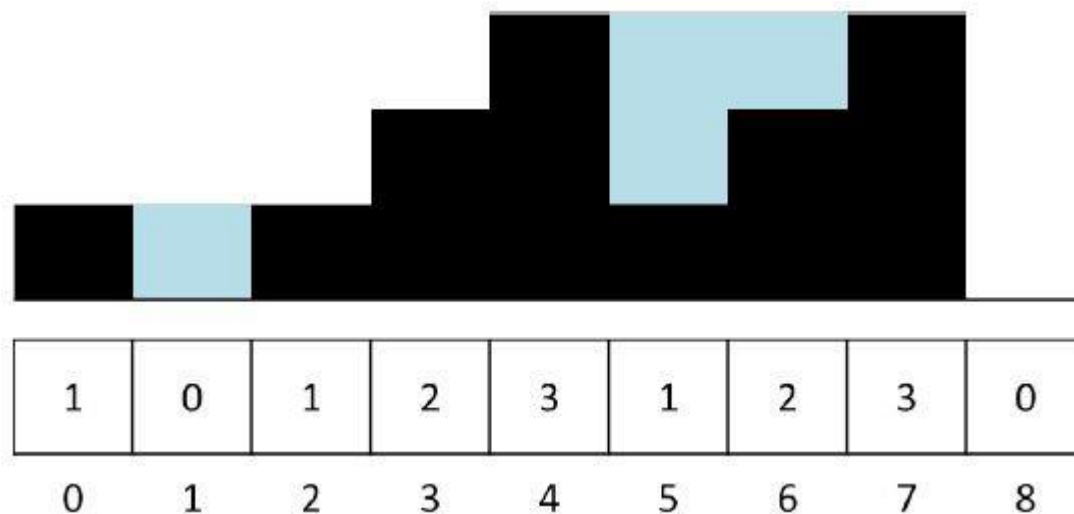
The stack is non-empty after reading the last character.

Alternate way without using a stack: using an integer.

Bonus: what if there are multiple different type of parentheses such as {} and []?

## Qn4

Given array of the height of the houses in the village, determine the number of houses that will be flooded.



## Qn4 Answer

Maintain a monotonic stack (either increasing or decreasing)

In this case, we are using a decreasing monotonic stack.

For every element  $i$ ,

If stack is non empty, pop all elements less than  $i$

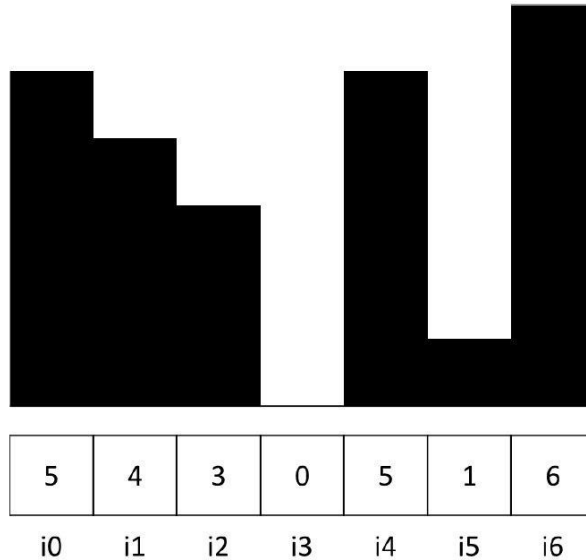
Add the number of elements popped into the result

Push  $i$  into the stack



## Qn4 Answer

Example: [5, 4, 3, 0, 2, 6]



res = 0, stack: {}

i0 -> push to stack

i1 -> push to stack

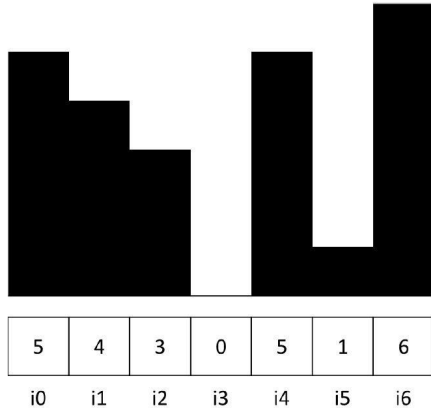
i2 -> push to stack

i3 -> push to stack

Stack at the moment: {5,4,3,0}

# Qn4 Answer

Example: [5, 4, 3, 0, 2, 6]



res = 0, stack: {5,4,3,0}

i4 -> 5 is larger than peek(0), pop, res++

res = 1, stack: {5,4,3}

5 > peek(3), pop, res ++

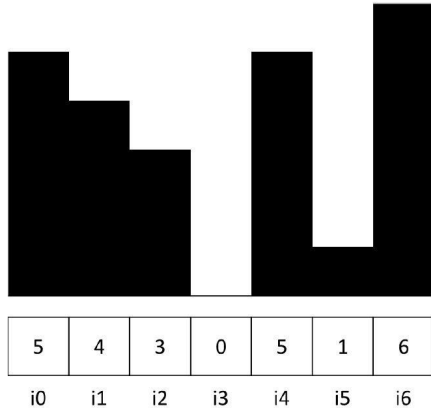
res = 2, stack: {5,4}

5 > peek(4), pop, res ++

res = 3, stack: {5}

# Qn4 Answer

Example: [5, 4, 3, 0, 2, 6]



res = 3, stack: {5}

push (5), stack: {5, 5}

i5 -> push to stack

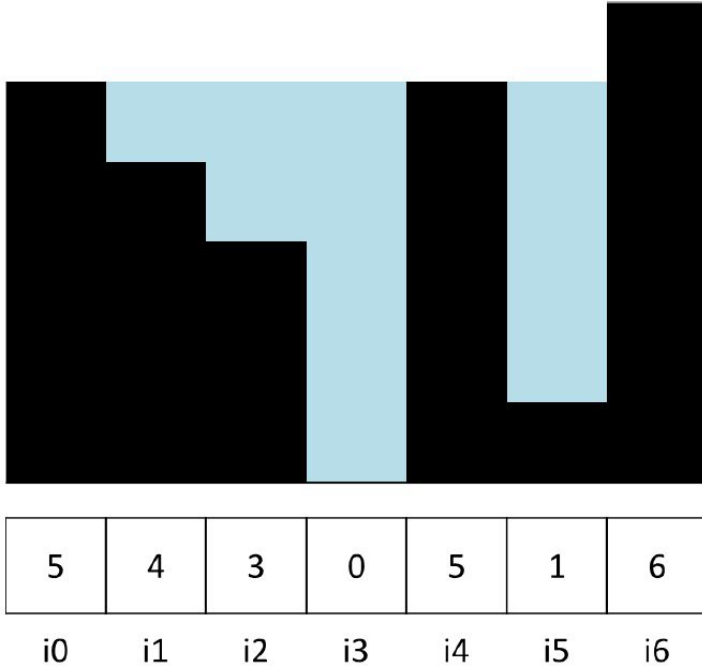
i6 -> Same as before pop all 3 from stack.

res = 6, stack : {}

Keep track of max, max\_count(number of max values popped)

Max = 5, max\_count = 2

## Qn4 Answer



Since stack is now the new max,  
Previous max will not be flooded

`res -= max_count`

`push()`

`res = 4, stack = {6}`

## Qn5 (Optional)

Sort a queue using another queue with  $O(1)$  additional space.

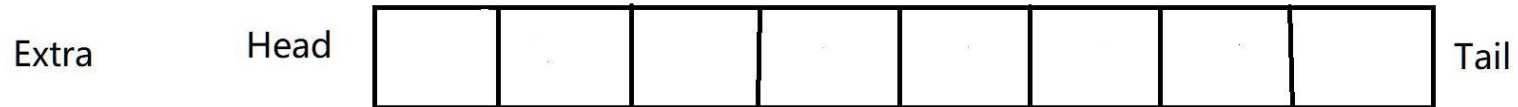
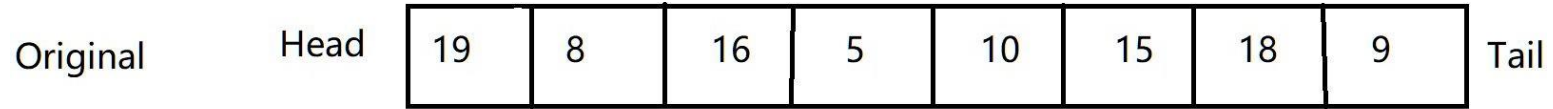
## Qn5

$O(n^2)$  way: cycling through the queue, picking the minimum element each time, and appending it to the second queue.

$O(n \log n)$  way: merge sort

## Qn5 Answer

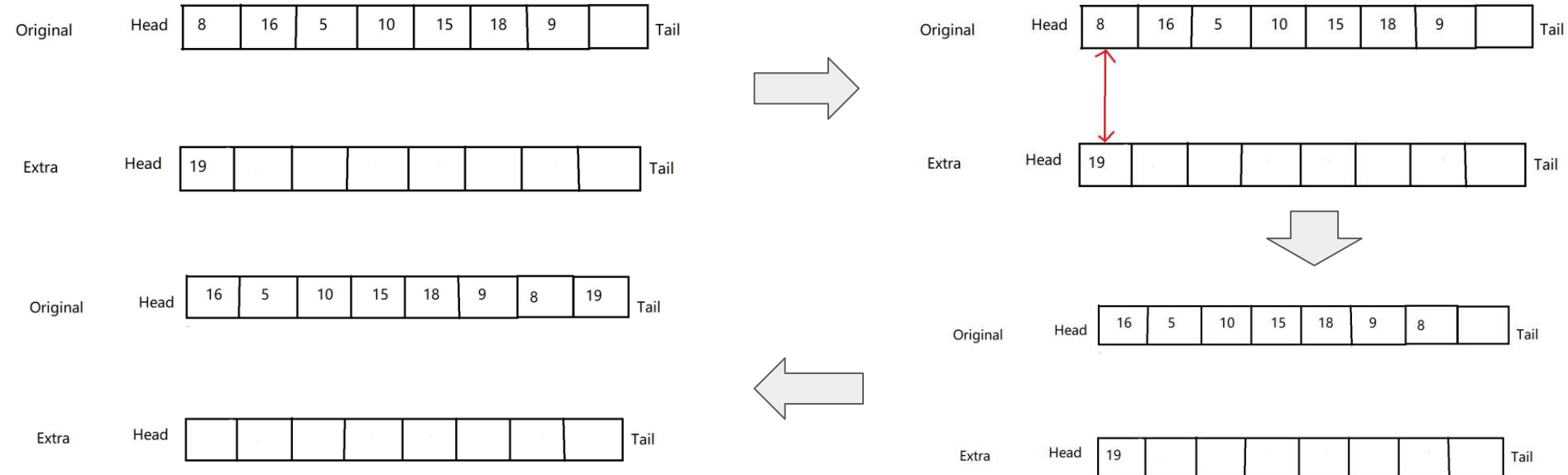
Same idea as iterative merge sort, check from size 2, 4 ....  $n^2$



# Qn5 Answer

Deque first half of the size, enqueue into extra queue

Compare heads, then enqueue smaller element into original array, followed by leather element





## Qn5 Answer

Repeat the process for larger sizes unless the queue is sorted

End of size 2      Head 8 19 5 16 10 15 9 18 Tail

End of size 4      Head 5 8 16 19 9 10 15 18 Tail