

**Problem 1. Time Complexity Analysis**

Analyse the following code snippets and find the best asymptotic bound for the time complexity of the following functions with respect to  $n$ .

(a) 

```
public int niceFunction(int n) {  
    for (int i = 0; i < n; i++) {  
        System.out.println("I am nice!");  
    }  
    return 42;  
}
```

**Solution:**  $O(n)$

This should be straightforward; it's a for-loop that runs for a total of  $n$  iterations.

(b) 

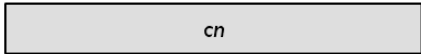
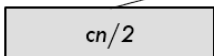
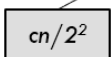

```
public int meanFunction(int n) {  
    if (n == 0) return 0;  
    return 2 * meanFunction(n / 2) + niceFunction(n);  
}
```

**Solution:**  $O(n)$

For a recursive function, the first step is to construct the recurrence relation.

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

There are many ways to solve this recurrence relation. One of the simplest ways is to draw the recurrence tree:

Level	Function Call	Work done	Total Work Done
0	$T(n)$		$cn$
1	$T(n/2)$		$cn/2$
2	$T(n/2^2)$		$cn/2^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$h$	$T(1)$		$c$

To solve the recurrence relation, simply sum up the total work done over all the levels. Additionally, when we encounter a geometric series, our lives are made simpler; a geometric series is upper bounded by the largest term.

$$\begin{aligned}
 T(n) &= cn + T\left(\frac{n}{2}\right) \\
 &= cn + \frac{cn}{2} + T\left(\frac{n}{2^2}\right) \\
 &= cn + \frac{cn}{2} + \frac{cn}{2^2} + T\left(\frac{n}{2^3}\right) \\
 &= \left(cn + \frac{cn}{2} + \frac{cn}{2^2} + \dots\right) + T(1) \\
 &\leq cn + \frac{cn}{2} + \frac{cn}{2^2} + \dots && \text{(Upper bounded by the infinite sum)} \\
 &= \sum_{r=0}^{\infty} \frac{cn}{2^r} \\
 &= \frac{cn}{1 - \frac{1}{2}} && \text{(Formula for infinite geometric series)} \\
 &= O(n)
 \end{aligned}$$

```
(c) public int strangerFunction(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            System.out.println("Execute order?");
        }
    }
}
```

```

    }
    return 66;
}

```

**Solution:**  $O(n^2)$

This is a standard nested for-loop. Roughly speaking, during the  $i$ -th iteration of the outer for-loop, the inner for-loop runs for  $i$  iterations. If we add them all up, we get:

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c \\
 &= \sum_{i=0}^{n-1} ic \\
 &= c(0 + 1 + 2 + 3 + \cdots + (n-1)) \\
 &= \frac{cn(n-1)}{2} \\
 &= O(n^2)
 \end{aligned}$$

```

(d) public int suspiciousFunction(int n) {
    if (n == 0) return 2040;

    int a = suspiciousFunction(n / 2);
    int b = suspiciousFunction(n / 2);
    return a + b + niceFunction(n);
}

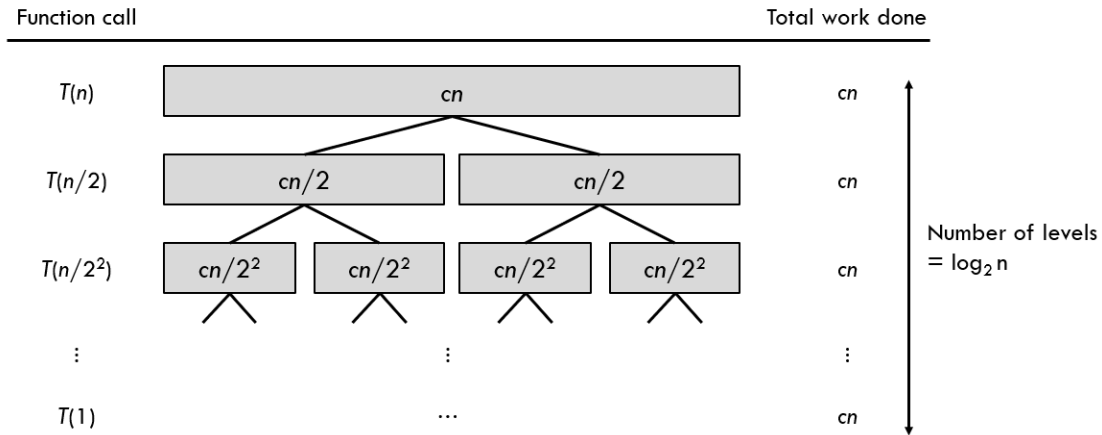
```

**Solution:**  $O(n \log n)$

Constructing the recurrence relation, we have:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

You might have noticed that this is exactly the recurrence relation for Merge Sort, and hence, can immediately declare the answer to be  $O(n \log n)$ . Otherwise, we can draw the recurrence tree:



Notice how the total amount of work done in each level sums up to  $cn$ . Therefore, we can simply multiply  $cn$  by the height of this tree. The function call at the  $k$ -th level is  $T(n/2^k)$ . To determine the height of the tree, we have to determine the value of  $h$  for which  $n/2^h = 1$  (i.e. the base case).

$$\begin{aligned} \frac{n}{2^h} &= 1 \\ 2^h &= n \\ h &= \log_2 n \end{aligned}$$

Therefore, the total amount of work done across all levels is  $(cn)(\log_2 n + 1) = O(n \log n)$ .

```
(e) public int badFunction(int n) {
    if (n <= 0) return 2040;
    if (n == 1) return 2040;
    return badFunction(n - 1) + badFunction(n - 2) + 0;
}
```

Note: The  $n$ th Fibonacci number can be expressed as  $F_n = \frac{\Phi^n - (1-\Phi)^n}{\sqrt{5}}$ , where  $\Phi = \frac{1+\sqrt{5}}{2}$ .  
 E.g.  $F_0 = 0$ ,  $F_1 = 1$ .

**Solution:**  $O(\phi^n)$

The trivial bound is  $O(2^n)$ . This is actually similar to the Fibonacci sequence and the tighter bound is  $O(\phi^n)$  where  $\phi$  is the golden ratio.

**Take note:** A faulty argument is to say “this program looks like Fibonacci, so it takes  $F(n)$  time, where  $F(n)$  is the  $n^{th}$  Fibonacci number”. This is because the recurrence for the work done is actually  $T(n) = T(n-1) + T(n-2) + O(1)$ , which is not the same as  $F(n) = F(n-1) + F(n-2)$  (there’s an additional +1 term).

Instead, we can prove it using induction, with  $F(0) = F(1) = 1, T(0) = T(1) = 0$ . Suppose  $n$  is a non-negative integer such that  $T(i) = F(i) - 1, \forall i < n$ . Then, by basic algebra, we can also say that:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &= F(n-1) - 1 + F(n-2) - 1 + 1 \\ &= F(n-1) + F(n-2) - 1 \\ &= F(n) - 1 \end{aligned}$$

Hence,  $T(n-1) = F(n-1) - 1$  implies  $T(n) = F(n) - 1$ .

There are several points that you need to take note of:

- First, notice that the base case for the relation  $T$  ( $T(0) = T(1) = 0$ ), given the context that we are analysing computational cost of the algorithm, does not make sense. The computational cost can’t be 0. This contradiction can be resolved by the fact that, this bound on our runtime applies on the algorithm on problem sizes 2 and above. Whilst this might feel clunky, this argument is actually acceptable in the context of algorithms. Why?
- Second, notice that it does not make intuitive sense that  $T(n) < F(n)$  as  $T(n)$  is the relation that has the additional +1 term. The relation between the relations is only true because the base cases of  $T(n)$  and  $F(n)$  are different. We are actually able to prove an alternative relation  $T(n) = F(n+1) - 1$  with the base cases  $T(0) = 0, T(1) = 1, F(0) = 1, F(1) = 1$ , where  $T(n) = F(n+1) - 1$  makes more intuitive sense. Just take note that we only decided to prove  $T(n) = F(n) - 1$  because it’s the most convenient.

```
(f) public int metalGearFunction(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < i; j *= 2) {
            System.out.println("!");
        }
    }
    return 0;
}
```

**Solution:**  $O(n \log n)$

We have another nested for-loop. But this time, during the  $i$ -th iteration of the outer for-loop, the inner for-loop runs for about  $\log(i)$  iterations. If we add them up, we get:

$$\begin{aligned}\log(1) + \log(2) + \cdots + \log(n) &= \log(1 \cdot 2 \cdots n) \\ &= \log(n!)\end{aligned}$$

For a simple way to upper bound  $\log(n!)$ ,

$$\begin{aligned}\log(n!) &= \log(1) + \log(2) + \cdots + \log(n) \\ &\leq \log(n) + \log(n) + \cdots + \log(n) \\ &= n \log n \\ &= O(n \log n)\end{aligned}$$

To prove that  $O(n \log n)$  is actually the best upper bound, we can prove that it is a lower bound as well.

$$\begin{aligned}\log(n!) &= \log(1) + \log(2) + \log(3) + \cdots + \log(n) \\ &\geq \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2} + 1\right) + \cdots + \log(n) \\ &\geq \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}\right) + \cdots + \log\left(\frac{n}{2}\right) \\ &= \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - \log 2) = \Omega(n \log n)\end{aligned}$$

```
(g) public String simpleFunction(int n) {  
    String s = "";  
    for (int i = 0; i < n; i++) {  
        s += "?";  
    }  
    return s;  
}
```

**Solution:**  $O(n^2)$

We have string concatenation in a loop here. On each concatenation, a new copy of the string is created, which takes  $O(\text{length of string})$  time. During the  $i$ -th iteration of the loop, the string is  $i - 1$  characters long. The time complexity is very similar to that in part (c). If we add them all up, we get:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} i \\ &= 0 + 1 + 2 + 3 + \cdots + (n - 1) \\ &= \frac{n(n - 1)}{2} \\ &= O(n^2) \end{aligned}$$

## Problem 2. Sorting Review

- (a) How would you implement insertion sort recursively? Analyse the time complexity by formulating a recurrence relation.

**Solution:** Recursively sort the array from index 0 to  $(n - 1)$ . Then, insert the last element into the sorted subarray. The recurrence relation will be  $T(n) = T(n - 1) + O(n)$ . Solving it results in the time complexity  $O(n^2)$ .

- (b) Consider an array of pairs  $(a, b)$ . Your goal is to sort them by  $a$  in ascending order. If there are any ties, we break them by sorting  $b$  in ascending order. For example,  $[(2, 1), (1, 4), (1, 3)]$  should be sorted into  $[(1, 3), (1, 4), (2, 1)]$ .

You are given 2 sorting functions, which are a MergeSort and a SelectionSort. You can use each sort at most once. How would you sort the pairs? Assume you can only sort by one field at a time.

**Solution:** You should use the SelectionSort to sort the pairs by  $b$  first, and then use MergeSort to sort the pairs by  $a$ . This is because, MergeSort is stable and would not affect the ordering of  $b$  (which is already sorted) when  $a$  has the same value.

Original Array:  $[(1, 3), (4, 2), (2, 1), (3, 8), (2, 5), (5, 2)]$

SelectionSort by  $b$ :  $[(2, 1), (\mathbf{5, 2}), (\mathbf{4, 2})^a, (1, 3), (2, 5), (3, 8)]$

MergeSort by  $a$ :  $[(1, 3), (2, 1), (2, 5), (3, 8), (4, 2), (5, 2)]$

---

<sup>a</sup>Note that the order is inverted

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

**Solution:** For the iterative MergeSort, we will sort the array in groups of power of 2. In other words, we will first sort the arrays in pairs, then merge into 4's, 8's and so on, until we have merged the entire array.

For each given size  $i$  (that is a power of 2),

- Copy the first  $i/2$  elements into a left array and the next  $i/2$  elements into the right array
- Set the *left\_pointer* and *right\_pointer* to the first element of each array, and *array\_pointer* to the first element of the original array
- Repeat until there is no more element in both arrays
  - Check the first element of the left and right array, and place the smaller element at index *array\_pointer* in the original array
  - Increment the pointer for the array containing the smaller number, and
  - Increment the *array\_pointer*
- Repeat the above for the next power of 2

In the iterative MergeSort, each size requires  $O(n)$  operation to perform the merge operation, where  $n$  is the length of the array. The possible number of sizes is  $\log n$ . And hence, the runtime complexity is  $O(n \log n)$ .

For space complexity, we only require an additional auxillary array, and thus it only requires  $O(n)$  space.

### Problem 3. Queues and Stacks Review

Consider the Stack and Queue Abstract Data Types (ADTs). Just a quick explanation, a Stack is a “LIFO” (Last In First Out) collection of elements that supports the following operations in  $O(1)$  time:

- **push:** Adds an element to the stack
- **pop:** Removes the **last** element that was added to the stack
- **peek:** Returns the last element added to the stack (without removing it)

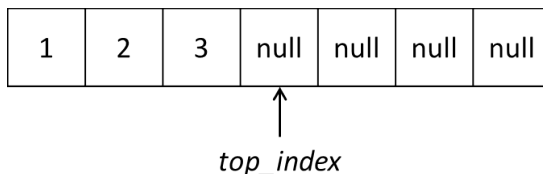
A Queue is a “FIFO” (First In First Out) collection of elements that supports these operations in  $O(1)$  time:

- **enqueue:** Adds an element to the queue front
- **dequeue:** Removes the **first** element that was added to the queue
- **peek:** Returns the next item to be dequeued (without removing it)



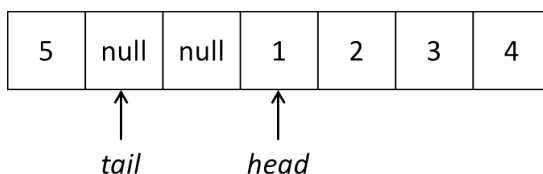
- (a) How would you implement a stack and queue with a fixed-size array in Java? (Assume that the number of items in the collections never exceed the array's capacity.)

**Solution: Stack:** Maintain *top\_index* that represents the index of the top of the stack. To perform a *push*, place the new element at *top\_index* and increment *top\_index*. To perform a *pop* or *peek*, return *arr[top\_index - 1]* (and subsequently decrement *top\_index* in a *pop* operation). All of these operations are achievable in  $O(1)$  time.



**Queue:** We need to maintain two indices, *head* and *tail* to represent the head and tail of the queues respectively (both starting at 0). To perform *enqueue*, place the new element at *arr[tail]* and increment *tail*. To perform *dequeue* or *peek*, return the element at *head* (and increment *head* in a *dequeue* operation). Again, all the operations would be in  $O(1)$ .

We might find that after performing  $l - 1$  enqueues and  $k < l$  dequeues (where  $l$  is the array's capacity), we would be enqueueing at index  $l$  which is out of the array's index bounds. At the same time, we find that array indices 0 to  $k - 1$  are all unused. The idea here is to “wrap around” and use 0 as the next index instead of  $l$ , so we can reuse the unused spaces at the front of the array. A simple way to achieve this is to replace increments  $tail + 1$  with  $(tail + 1) \% l$ .



- (b) A Deque (double-ended queue) is an extension of a queue, which allows enqueueing and dequeueing 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.)

**Solution:** The idea is the same with the queue implementation, except any *\_front* operations would manipulate the *head* index, while *\_back* operations would manipulate the *tail* index.

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

**Solution:** For *enqueue* and *push* operations, we need to be mindful of our underlying array's capacity. If the current collection size is equal to the array's capacity, we can report that the operation is invalid (by returning `false`, for example). Alternatively, we can think of a strategy to move to an array with a bigger capacity (this might involve an expensive operation to copy everything in this array, but as we will learn later on in CS2040S, it may not be that costly after all). Detecting this error is relatively simple for our stack implementation (we just need to check if *top\_index* is equal to array's length), but might be a bit trickier for our "circular" array queue.

For *dequeue* or *pop* operations, the main error to check for is that we are not performing on an empty collection - if we find that this is so, we would need to throw an exception again.

- (d) A sequence 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.

**Solution:** The idea behind this is to simply push opening parentheses "(" onto a stack, then pop them out whenever you encounter a closing parenthesis ")". The string is unbalanced if there is an attempt at:

- popping an empty stack, or
- if the stack is non-empty after reading the last character.

- (e) A sequence of opening and closing parentheses of three different types {}, (), and [] is given. We will define a hyperbalanced parenthesis sequence in a recursive way. For the recursion base, we say that empty parentheses sequence is a hyperbalanced one. Next, if "X" and "Y" are hyperbalanced sequences, then any of the sequences "{X}", "[X]", "(X)", and "XY" is hyperbalanced. For example, sequences "{}({{}})", "{ }", "((( )))" are hyperbalanced, but sequences "{( )}", "]", "{(}" are not. Determine whether a given sequence of parentheses is hyperbalanced.

**Solution:** The idea is similar to the previous task: we push an opening parenthesis to the stack, keeping track of its type. Whenever we encounter a closing parenthesis, we check whether the stack is not empty and whether the parenthesis at the back of the stack has the same type as the closing one. If these conditions are satisfied, we pop the stack and continue; otherwise, the sequence is not hyperbalanced. Also ensuring that the stack becomes empty after all the operations.

#### Problem 4. Mountain stack

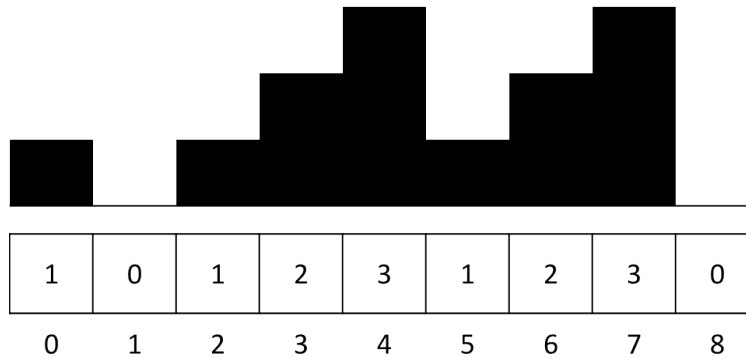
Adventure flows in your veins, and today, you embark on your journey through the mountains! The range consists of  $n$  hills, aligned linearly and numbered from 0 to  $n - 1$ . Each hill, indexed at  $i$ , stands at a height of  $a[i]$  meters and spans a length of 1 kilometer. As you plan your journey, you're keen to understand how far you can see to your left (you choose not to look right due to the prevailing winds in the mountains). When you stand atop hill  $i$  and look leftward, you'll see a point  $j$  kilometers away if and only if all hills from  $i - j$  to  $i$  are no taller than  $a[i]$  meters. For each

hill  $i$ , determine the maximum distance you can see from it. If there are no hills  $j < i$  taller than  $i$  then the answer for  $i$  is "infinity". Your goal is to solve this problem as effectively as possible. (Hint: try using a stack to achieve  $O(n)$  time complexity).

Example:

[1, 0, 1, 2, 3, 1, 2, 3, 0]

The array above can be represented with the diagram below.



For hills numbered 0, 2, 3, 4, and 7, there are no taller hills to their left; therefore, the answer for them is "infinity". For the hill with index 1, it is impossible to see even 1 km to the left, as the hill at index 0 is taller, and thus the answer is 0. The same applies to the hill numbered 5, as  $a[4] > a[5]$ . However, when standing on top of hill 6, one can observe 1 km to the left, since  $a[5] \leq a[6]$  and does not limit the sight. But the view is limited beyond that, as  $a[4] > a[6]$ .

**Solution:** One possible solution would be to use a monotonic stack. In a monotonic stack, the elements can be either monotonic decreasing or increasing (i.e. the elements are in order). In this case, we will use a monotonic decreasing stack (e.g. [5, 3, 2, 1]).

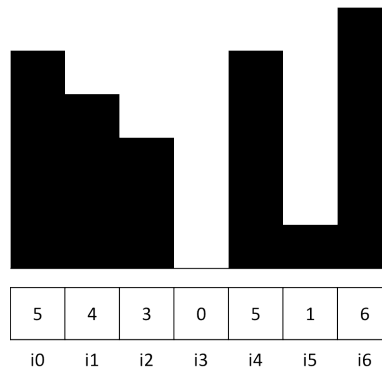
For each element in the array, we want to find an index on its left, where the elevation is higher than itself. If there exists such an index, the visibility of the current hill is limited.

Let's follow the following algorithm.

For every element at index  $i$  in the array:

- While the stack is non-empty, repeatedly pop indices off from the stack (which contains elements to the left of  $i$ ), that are smaller or equal than the elevation of index  $i$ 
  - If the stack eventually becomes empty, then there is no higher hill to the left, thus the answer for  $i$  is infinity.
  - Otherwise, if the index of the hill at the top of the stack is  $j$ , then the answer for  $i$  is  $i - j - 1$  km, as hill  $j$  is the first hill to the left of  $i$  that is higher than hill  $i$ .
- Push index  $i$  onto the stack

Example: [5, 4, 3, 0, 5, 1, 6]



Step 1: We will first push the indexes of heights 5, 4, 3 and 0 onto the stack one at a time. Answer for hill 0 is "infinity", as there are no hills to the left of it, and the answer for the rest of them is 0, as the first hill to the left of them is higher.

- Current stack state: [ $i_0, i_1, i_2, i_3$ ]

Step 2: When we reach  $i_4$ , we will pop all the four elements from the stack ( $i_0, i_1, i_2$  and  $i_3$ ), because 5, 4, 3, and 0 are less than or equal to 5. We update answer for  $i_4$  with "infinity", as the stack becomes empty. At the end,  $i_4$  is then pushed onto the stack.

- Current stack state: [ $i_4$ ]

**Solution:** (*Continued*)

Step 3: We then push  $i5$  onto the stack (and do not pop anything because  $i4$  is larger than  $i5$ ), and the answer for  $i5$  is 0.

- Current stack state:  $[i4, i5]$

Step 4: When we reach  $i6$ , we will pop all the elements from the stack ( $i4, i5$ ) as they are all less than 6. As everything is popped, we can conclude that answer for  $i6$  is "infinity".

- Current stack state:  $[i6]$

For the runtime analysis, each element is added and removed from the stack at most once, with each pop/push operations being  $O(1)$  cost. Hence, the runtime is  $O(n)$ .

**Problem 5.    Sorting with Queues**

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

**Solution:** The naïve way to do this is by simply cycling through the queue, picking the minimum element each time, and appending it to the second queue (this would take  $O(n^2)$ ). However, we can of course do better. The idea is to do it like merge sort! We will start by sorting by pairs, then by 4 elements, so on and so forth. For example, consider the following queue,  $Q1$ :

Head 19 8 16 5 10 15 18 9 Tail

In the first iteration, we group the elements into groups of  $2^1 = 2$  and sort within each group. To sort the first pair, we dequeue the first half of the pair (in this case, 19) from  $Q1$  and enqueue it in the second queue,  $Q2$ . Then, we compare the heads of both queues. The smaller element (in this case, 8) gets dequeued and enqueued in  $Q1$ . Then, the other element (in this case, 19) gets dequeued and enqueued in  $Q1$ . We do this for every pair of elements. At this point in time, the elements in  $Q1$  are:

Head 8 19 5 16 10 15 9 18 Tail

In the second iteration, we group the elements into groups of  $2^2 = 4$  and sort within each group. To sort the first group of 4, we dequeue the first half of the group (in this case, 8 and 19) from  $Q1$  and enqueue it in  $Q2$ . Then we compare the heads of both queues, dequeuing the smaller element and enqueueing into  $Q1$  **until both queues have had  $4/2 = 2$  dequeue operations**. By now, the state of  $Q1$  is:

Head 10 15 9 18 5 8 16 19 Tail

We continue this process and end the second iteration off with  $Q1$  being in the following state:

Head 5 8 16 19 9 10 15 18 Tail

After two more iterations,  $Q1$  will be sorted. This algorithm runs in  $O(n \log n)$  time because there are  $O(\log n)$  iterations, each with  $O(n)$  dequeue and enqueue operations.