Week03 Tutorial

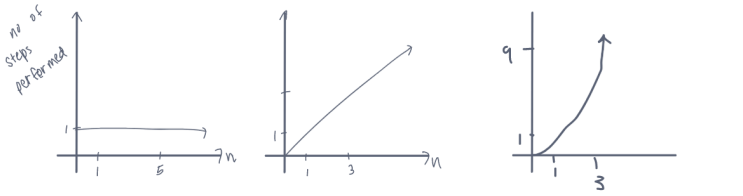
Time complexity: the amount of time taken by an algorithm to run as a function of the length of input
Primitive operation: basic computations that takes "constant time" to perform

- assigning variable
- evaluating an expression
- calling/returning from a function
- indexing into an array

```
int i = 0;
```

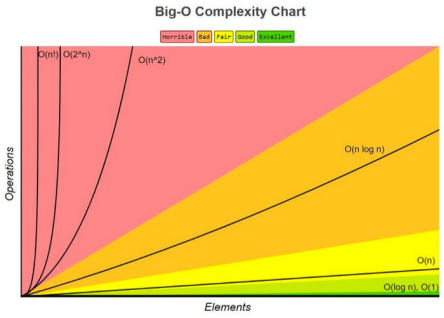
```
for (int i = 0; i < n; i ++) {
    printf("a line\n");
}
```

```
for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j++) {
        printf("a line\n");
    }
}
```



Big O Notation

- Describes the efficiency or performance of an algorithm
- Specifically describes how much time an algorithm would take to execute in the worst-case scenario



To find Big O complexity: keep the most dominant term and drop any constants

$6n^3 + 7n\log(n) - 2n + 3 = O(n^3)$

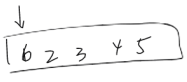
```
void fn(n) {
    for (int i = 0; i < n; i++) {
        print("hi")
    }
    fn(n / 2)
}
```

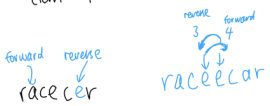
$n + n/2 + n/4 + n/8 + \dots$

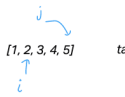
$O(n \log n)$

$n = 4 \rightarrow fn(4), fn(2), fn(1)$

$n = 16 \rightarrow fn(16), fn(8), fn(4) \dots$







QueueNew:
    initialise two empty stacks s1 and s2

QueueEnqueue:
    push given item onto s1

QueueDequeue:
    if s2 is empty:
        while s1 is not empty:
            pop an item from s1 and push it onto s2

    front = pop item from s2

    return front

the amortised time complexity of dequeue is O(1)