Big O Notation

1. Describe what Big O Notation is ?
2. Simplyfy Big O Expressions ?
3. Define time complexity and Space complexity ?
4. Evaluate the time complexity and space complexity of different algorithms using Big O Notation
5. Describe what a logarithm is ?

Intro to Big O Notation

▼ Describe What Big O Notation is ?

- It's a system or a way of generalizing code and talking about it and comparing it's performace to others pieces of code.
- It's a numeric representation of the performace of our code. With Big O Notation, you can actually pinpoint where some problems might arise.
- It's helps us understand things that are slowing our code, it helps us identify ineffiencies in our code.

Timing our code

▼ Suppose you want to write a function that calculate the sum of all numbers from 1 up to (and including) some numbers n.

- The easiest solution is to create a variable total which works like an accumulator and then loop through all those numbers and add them in one at a time, starting at one all the way up until n.
- we have another function addUpTo2() in which we have derived a mathematical formula which does the same thing the above function does.
- So now we have to determine which one is better, by better we mean

- which code is faster in milliseconds(How long does the code take to execute) for this we will use function called performace.now(), it determines how much time has elapsed since a particular point in your code. So first we'll check right before our function addUpTo() is executed and then right after the exection has completed. Then we find the time difference which is in milliseconds. The first function takes 0.063496 milliseconds and the 2nd function takes 0.019789 milliseconds which is much less then the 1st function, it seems alot more efficient. But this process is not the most reliable of manually timing things like this before and after and comparing. What we want to do is give it a label of how efficient this one is comapred to the other one, which brings us to the problem with time becasue different machines will record different time, so it's not reliable depending on the specifications of a machine because we dont have precision. So how can we walk through our code and acutally talk in general terms about which code is better without having to go and time it.
- it's about how much memory is used(number of variables created, the data that is stored each time that function is called).
- It's about readability

```
function addUpTo(n) {
    let total = 0;
    for (let i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}

function addUpTo(n) {
    return n * (n + 1) / 2;
}
```

Counting Operations

▼ So how can we walk through our code and acutally talk in general terms about which code is better without having to go and time it ?

- We can count number of simple operations that a computer has to perform. E.g. in the function below, we have three operations. First we have multiplication, then addition and at last we have the division operation. It doesn't matter what n is, there are only three calculations that are happening regardless of the size of n

```
function addUpTo(n) {
    return n * (n + 1) / 2;
}
```

- Now if we count the number of operations in the function below, inside the loop we have `total += 1` e.g. if `n` is equal to 5 then the operation will be performed 5 times, if `n` is equal to 10 then the operation will be performed 10 times, then there is the assignment operation, then the increment operation which involves addition and assignment however they happen one time each per `n` and as n grow we have n additions and n assignments. Then inside the loop we have variable intializtion clause which happens once when the loop is executed. Then we have a conditional expression which is checked `n` times. Then at the start of the function we have a variable `total` to which we are assigning the vaue `0`, which only happens once. Now we have alot of operations and counting them would be hard. So we need to generalize it. There are alot of ways of counting but it could be as high as `5n + 2` and as low as 2n.
    - `let total = 1` assignment at the start of the loop which counts as 1 operation.
    - Then variable intialization statement `let i = 0` which counts as 1 operation it only happens once.
    - Then we have `i <= n` this happens `n` times.

- Then we have `i++` this also happens `n` times, it contains two operations first is addition which happens `n` times and the second one is assignment operation which also happens `n` times.
- Then inside the body of the loop we have `total += 1` which also has two operations, the first one is addition operations which happens `n` times and the second one is assignment operation which also happens `n` times.
- Now if we count the total number of operations, we have 5n + 2 operations

```
function addUpTo(n) {
    let total = 0;
    for (let i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}
```

Official Into to Big O

▼ What is Big 0 generally?

- It's a way to formalize fuzzy counting of operations. It allows us to formally talk about how the runtime of an algorithm grows as the input grow. It's a way of describing the relationship between the input to a function as it grows and how that changes the runtime of that function, the relationship between the input size and then the time relative to that input.

▼ Definition of Big O ?

- We say than an algorithm is O(f(n)) if the number of simple operations the computer has to do is eventually less than a

constant times f(n), as n increases. What this definition means is that:

- f(n) could be linear (f(n) = n) (f(n) = n means a function with an input of n and then the output, we are describing the relationship between the input and then the runtime). This means that as the input scales the runtime scales lineraly as well.

```
// Here we have 5n + 2 operations or we can say t
// of operations is bounded by a multiple of n. I
// if it's 10n or 50n because we are going to sir
// so the big O notation will be O(n).
//
function addUpTo(n) {
    let total = 0;
    for (let i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}

// Now if we try to figure out the big O for this
// that in the first loop as n grows we have roug
// that means we have O(n) here. Then we have and
// we have rougly n operations. So here we might
// remember that we dont care about that. WE care
// simplify it to O(n)
function countUpAndDown(n) {
  console.log("Going Up!");
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
  console.log("At the top!\nGoing down...");
  for (let j = n - 1; j >= 0; j--) {
    console.log(j);
```

```
  }
  console.log("Back down. Bye!");
}
```

- f(n) could be quadratic (f(n) = n^2). This means as the input scales the runtime scales quadractically

```
// Here we have a nested loop, if we try to figu
// we start by saying that in the outer loop as
// so it's big O will be O(n)but then we have a
// have n operations so it's Big O will be O(n).
// have simplified to O(n) becasue it's nested,
// O(n) operation is O(n * n) which is O(n^2). So
//, the runtime roughly grows at the rate of n so
// Big O will also we
function printAllPairs(n) {
  for (var i = 0; i < n; i ++) {
    for (var j = 0; j < n; j++) {
      console.log(i, j);
    }
  }
}
```

- f(n) could be constant (f(n) = 1). This means as the input scales it doesn't have an impact on the runtime, it always remains constant.

```
// Here we have 3 operations and the Big O notati
// O(1).What this means is that as n grows(the in
// it has no change on the runtime, it's not refl
function addUpTo(n) {
    return n * (n + 1) / 2;
}
```

- f(n) could be something entirely different!

- When we talk about Big O, we talk about the worst case scenrio

Simplifying Big O Expression

- When determining the time complexity of an algorithm, there are some helpful rules of thumb for Big O expressions.
  - constants don't matter e.g. if we have something like O(2n) we would simply it to O(n).
  - If we have O(500)e.g. there are 500 operations. we would simplfy it to O(1), it's how we say something is constant, constant runtime.
  - if we have O(13n^2) we would simply it to O(n^2)
  - Smaller terms e.g. O(n + 10) donot matter, so get rid of the 10, it'll just be 0(n) or if it's O(1000n + 50) it will be simplified to O(n).
  - If we have O(n^2 + 5n + 8) we would simply it to O(n^2) because in comparsion to n^2 ⇒ 5n + 8 is very small e.g. if n is 1000 then 5n is equal to 5000 and if we do n^2 it would equal to 10 million.

Big O shorthands

1. Arithmetic operations are constant ⟶ if we are adding something or subtracting something it's going to have constant runtime e.g. a computer takes the same time to add 2+2 or 100000+100000.
2. Variable assignment is constant ⟶ your computer takes the same amount of time to create and assign a variable the value 10 or 100000.
3. Accessing elements in an array(by index) or object's value (by key) is constant. ⟶ Accessing an element in an array using index number or accessing a value in an object using a key thats constant runtime as well.
4. In a loop, the complexity is the length of the loop times the complexity of whatever happens inside the loop.

```
// Inside the loop we have a condtional expression that
// i <= Math.max(5, n), Math.max() returns the largest
// to it as argument
// Whats the Big O here ?
// we have a loop and this loop is going to go from 1 t
//  which ever one is larger. IF n is smaller than 5 th
// going to execute 5 times but if n is larger than 5 e
// this loop is going to run 10 million times, so we ca
// say this loop has a Time comlexity of O(n)

function logAtleast5(n) {
  for (var i = 1; i <= Math.max(5, n); i++) {
    console.log(i);
  }
}
```

```
// Inside the loop we have a condtional expression that
// i <= Math.min(5, n), Math.min() returns the smallest
// to it as argument.
// Whats the Big O here ?
// we have a loop and this loop is going from 1 to eith
// which ever one is smaller. If n is approaching a mil
// to run 5 times, if n is 2 then the loop is going to
// can simply that as n grows the Big O of this is just
//
function logAtMost5(n) {
  for (var i = 1; i <= Math.min(5, n); i++) {
    console.log(i);
  }
}
```

Big O Time Complexity Quiz

▼ Simply the big O expression as much as possible O(n + 10) ?

- O(n)

▼ Simply the big O expression as much as possible O(100 * n) ?

- O(n)

▼ Simply the following big O expression as much as possible O(25) ?

- O(1)

▼ Simply the following big O expression as much as possible O(n^2 + N^3) ?

- O(n^2)

▼ Simply the following big O expression as much as possible O(n + n + n + n) ?

- O(n)

```
function logUpTo(n) {
    for (var i = 1; i <= n; i++) {
        console.log(i);
    }
}
```

▼ Determine the time complexity of the following function ?

- We have function which takes n as a parameter, then we have a loop which includes variable initialization clause which is 1 operation, then we have the comparsion expression which happens `n` times , then we have the increment operator which has two parts first one is addition and the second part is assignment operation so thses are 2 operations that happen n times which we can say are `2 * n`. So we have a total of `2n + n + 1` operations, and we can further simply it to `3n + 1` operation, if we simply it further it becomes `3n` and if we simply it further it beomes O(n).

```
function logAtMost10(n) {
    for (var i = 1; i <= Math.min(n, 10); i++) {
        console.log(i);
    }
}
```

▼ Determine the time complexity of the following function ?

- We have a function that takes n as a parameter. Inside the function we have a loop. Inside the loop we have variable intialization statement which happens once so that is `1` operation. Next we have a comparison expression which happens `n` times if n is less than 10 or 10 times if n is greater than 10, so we can simply it as n grows the bif O of this is just constant e.g. O(1). Thus the Big O of this function is O(1).

```
function logAtLeast10(n) {
    for (var i = 1; i <= Math.max(n, 10); i++) {
        console.log(i);
    }
}
```

▼ Determine the time complexity of the following function ?

- We have a function that takes n as a parameter. Inside the function we have a loop. Inside the loop we have variable intialization statement which happens once so that is 1 operation. Next we have a comparison expression which happens n times if n is greater than 10 or 10 times if n is less than 10, so we can simply it as n grows the big O of this is O(n). Thus the Big O of this function is O(n).

```
function onlyElementsAtEvenIndex(array) {
    var newArray = Array(Math.ceil(array.length / 2));
    for (var i = 0; i < array.length; i++) {
        if (i % 2 === 0) {
            newArray[i / 2] = array[i];
        }
    }
    return newArray;
}
```

▼ Determine the time complexity for the following function ?

- Inside the function we have a variable decleration statement and we are initializing it with with an array, so that is `1` operation. Next we have a for loop which includes a variable initialization statement which happens once so thats `1` operation, then we have a comparison expression which is checked `n` times e.g. if the length of the array is 10 it will be checked 10 times but if the length of the array is 10 million then it will be checked 10 million times, so we can say that as `n` grows the Big O will be `O(n)` , next inside the loop we have an incremenet operator which includes two operations first is addition and then assignment, so that counts as `2` operation that take place `n` times which makes it `2n`. Then further we have a conditonal expression which is checked on each iteration of the loop, so we can can say as n grows the Big O of the conditional expression will be `O(n)`.Next inside the if Block we have an array element assignment statement which is executed half the number of times the loop if exectued which will make it `n / 2` times. So if we sum all operations we have `1 + 1 + n + 2n + n + (n /2)`. If we simplify it, it becomes `2 + (17n / 2)`,So the Big O of this function will be `O(n)`.

```
function subtotals(array) {
    var subtotalArray = Array(array.length);
```

```
    for (var i = 0; i < array.length; i++) {
        var subtotal = 0;
        for (var j = 0; j <= i; j++) {
            subtotal += array[j];
        }
        subtotalArray[i] = subtotal;
    }
    return subtotalArray;
}
```

▼ Determine the time complexity for the following function ?

- First we have a variable decleration and initialization which counts as `1` operation.
- Then we have a loop which includes
  - variable initialization clause which only happens once so thats `1` operation.
  - Then we have a comparison expressions which happens `n` times.
  - Then we have an increment operator which includes 2 operation which happens `n` times which makes `2n`.
  - Then we have a variable decleration which happens each time the loop is executed so that makes it `n` times.
  - Then we have a nested loop which has a variable initialization clause which only happens once so thats `1` operation. Then we have a comparison expressions which happens `n` times. Then we have an increment operator which happens `n` times. Then inside the loop we have array element assignment which happens `n` times.
  - Then outside the loop we have another array element assignment statement which takes place `n` times
  - Now if we sum up all our operation it will become ` 1 + n + 2n + 2n + n(1 + n + 2n + n) + n, if we simply it, it becomes `24n^2 + 6n`

`+ 5` and if we further simplify it, it beomes O(n^2)

▼ So what is time complexity ?

- It's analyzing how the runtime of an algorithm scales as the size of the input increases

Space Complexity

▼ What is space complexity ?

- It's the space that an algorithm takes up as the size of the input increases. And we can use the same Big O notation for describing what happens

▼ What are the rules of thumb for Space Complexity ?

- Most primitive values(numbers, boolean, undefined, null and NaN) except Strings occupy constant space, so it doesn't matter what the size of the input is e.g. if the number is 1 or 10000 we can consider it constant space.
- Strings require O(n) space Where n is the string length. e.g. if a string grows to 50 characters the string takes up 50 times more space.
- Reference types are generally O(n), where n is the length (for array) or the number of keys (for objects). e.g. if a length of an array is 4 compared to another one that is 2, it takes twice as much space as shorter array.

```
function sum(arr) {
    let total = 0;
    for (let i = 0; i < arr.length; i++) {
        total += arr[i];
    }
    return total;
}
```

▼ Whats the space complexity of the above code ?

- The function takes an array and sums all the items in the array.
- We have a variable total that is assigned a number literal so it occupies constant space that is O(1).
- Then inside the loop we have a variable intialization clause that is assigned a number literal so that takes up constant space as well, so it's space complexity is O(1).
- Then inside the body of the loop we are adding the elements of the array to the total variable.
- That means this function has space complexity of O(1).

```
function double(arr) {
    let newArray = [];
    for (let i = 0; i < arr.length; i++) {
        newArr.push(2 * arr[i]);
    }
    return newArr;
}
```

▼ Whats the space comlexity of the above code ?

- The function takes an array as an argument, loops over the new array and appends the double of the element which are number literal into the new array on each iteration.
- We have a variable that is assigned a reference to an array, so it's space complexity is O(n) as n grows(the array length) the space taken up also increases. So if the length of the array that was passes to the function was `10` we are also storing `10` items in the array, if there were 50 items in the array passed as argument then we will be appending 50 items in the new array. So the space taken up is directly propotional to the input n(which in this case is an array). So the space complexity is O(n).

Quiz on Space Complexity

```
function logUpTo(n) {
    for (var i = 1; i <= n; i++) {
        console.log(i);
    }
}
```

▼ Determine the **space** complexity for the following function ?

- The function has a loop that contains variable initialzation clause which has been assigned a number literal, so it's space complexity is O(1). So the space complexity is O(1).

```
function logAtMost10(n) {
    for (var i = 1; i <= Math.min(n, 10); i++) {
        console.log(i);
    }
}
```

▼ Determine the **space** complexity for the following function ?

- The function has a loop that contains variable initialzation clause which has been assigned a number literal, so it's space complexity is O(1). So the space complexity is O(1).

```
function logAtMost10(n) {
    for (var i = 1; i <= Math.min(n, 10); i++) {
        console.log(i);
    }
}
```

▼ Determine the **space** complexity for the following function ?

- The function has a loop that contains variable initialzation clause which has been assigned a number literal, so it's space complexity is O(1). So the space complexity is O(1).

```
function onlyElementsAtEvenIndex(array) {
    var newArray = Array(Math.ceil(array.length / 2));
    for (var i = 0; i < array.length; i++) {
        if (i % 2 === 0) {
            newArray[i / 2] = array[i];
        }
    }
    return newArray;
}
```

▼ Determine the **space** complexity for the following function ?

- The function takes an array as an argument. Then it declares a new variable and assigns it an array whoes length depends on the length of the array passed it the function as an argument, so as the length of the array increases the space complexity also increases this is variable has the space complexity of O(n). Further we have a loop which contains a variable initialization clause to which we are assigning a number literal so this has a space complexity of O(1). So after simplication the space complexity of the function is O(n).

```
function subtotals(array) {
    var subtotalArray = Array(array.length);
    for (var i = 0; i < array.length; i++) {
        var subtotal = 0;
        for (var j = 0; j <= i; j++) {
            subtotal += array[j];
        }
        subtotalArray[i] = subtotal;
    }
    return subtotalArray;
}
```

▼ Determine the **space** complexity for the following function ?

- We have a variable subtotalArray to which we have assigned an array which has the same length of the array as the array passes to the function as an argument, thus as the length of the array increases the length of the subtotalArray also increases lineraly, so it's space complexity is `n`.
- Next we have a loop which has a variable initialization clause to which we have assigned a number literal, so it's space complexity is `1`. Then we have another variable declared inside the loop which has been assigned a number literal so it's space complexity is `1`.
- Next we have a nested loop which contains a variable initialization clause and this variable has been assigned a number literal so its space complexity is `1`.
- Now if we add up all the space complexities we get `n + 1(1)` which simplfies to n, so the space complexity is O(n)

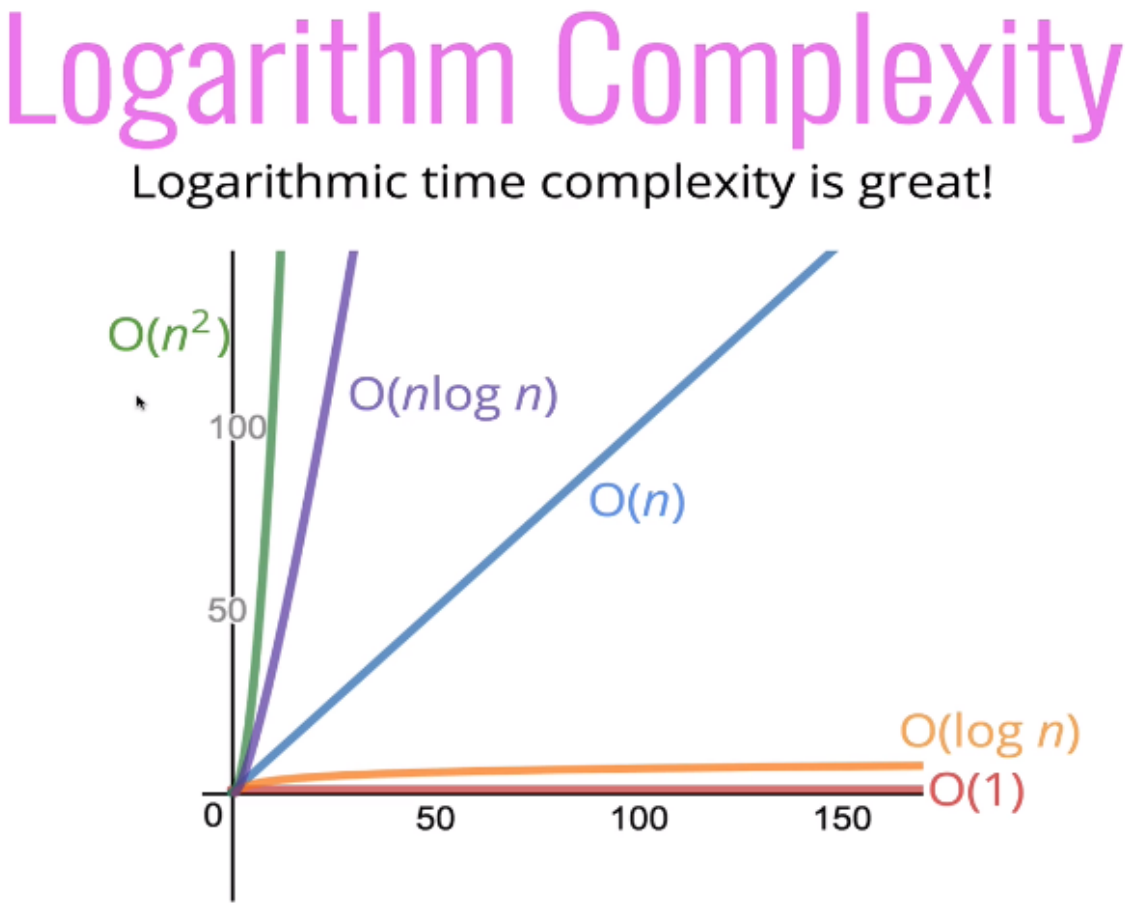
Logs and Section Recap

- A log is an inverse of an exponential action, just like multiplication and division are pairs, exponentiation and logs are pairs.
- So if we say log base 2 (8) = 3, we'd read this as log with base 2 of 8 is equal to 3. What we are calculating here is 2 to what power equals 8, so the answer is 3.

$$\log_2(8) = 3 \longrightarrow 2^3 = 8$$

$$\log_2(value) = exponent \longrightarrow 2^{exponent} = value$$

- Now logaritms aren't always working with base 2, we could have base 3, 5 or 10 and so on.
- When writing log with base 2 we can omit the base 2, which means the base is by default 2.



- If you have an algorithm with Log n time comlexity it is fantastic. It's better than O(n) and O(n^2)

Recap

- To analyze the performace of an algorithm we use Big O Notation. As the size of an input grows we want to know how the runtime changes or how the space complexity.
- Big O can give us a high level understanding of time and space complexity

- The time and space complexity depend only on the algorithm not on the hardware.