Big O 1

Date ´
Page 1

* - Big O notation.

* - Analyzing Performance of arrays and objects.

* - Problem solving approach and pattern

* - Recursion
- Searching Algorithm.
- Bubble sort, Selection sort, insertion sort
merge sort, quick sort, Radix Sort.

div - intro to Data structure.

- Singly linked list, doubly linked list.
- Stacks and queues
- Binary search trees.
- Tree traversal
- Binary Heaps
- Hash tables.
- Graphs.
- Graphs Traversal
- ~~Dijkstra~~ Dijkstra's algorithm.

Big O notation

Suppose we have to add the total number. Passed like if we pass 5 so it will be
$1+2+3+4+5 = 15$ factorial.

1st was using for loop
we

~~function (~~

```
Const addupto=(n) => {
  let total = 0;
for (let i = 0; i<=n; i++) {
    total = total + i // Shorthand total += i;
console.log (total)
  } };
addupto ~~function~~ (5)
```

~~It~~ will 0, 1, 3, 6, 10, 15.

2nd way

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

Both are same

what does better mean?

- first ?
- Less memory - intensive? → more important.
- More redable ?

To check performance we will use built in performance.now() function.

```
var t1 = performance.now();
addupto1(10000000);
var t2 = performance.now();
console.log(`time Elapsed1: ${(t2-t1)/1000} seconds`)
```

to get time in second.

Same wit addupto2()

op is

time Elapsed1: 0.01166866 seconds.
time Elapsed2: 0.000005874 seconds.

Second one is way faster.

Problem with time.

- Different machine will record different time
- Same machine will record different time.
- for fast algorithms, speed measurement may not be precise enough.

To handle this Big O notation comes in picture.

If not time, then what?

Rather than counting seconds, which are so variable...

lets count the number of simple operations the computer has to perform

we can use this

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

1 operation 2 3

Here we have 3 operation.

3 operation

constant 3

3 simple operation regardless the size of the ~~operations~~ n

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

1 assignment.
1 assignment.

n additions and n assignments.

one operation but it depends on n

if n = 10 then it's 10 operation

if n = 10000 the it's 10000 operation

So it is n operations

assignment operation also

it's not a static number it could be 5n + 2

as n grows calculation grows as n

Big O

Big O Notation is a way to formalize fuzzy counting.

it allows us to talk formally about how the runtime of an algorithm grows as the input grow.

Big O defination:

we say that an algorithm is $O(f(n))$ if the number ~~its~~ of simple operations the computer has to do is eventually less than a constant time $f(n)$, as n increses.

- $f(n)$ could be linear ($f(n) = n$)
- $f(n)$ could be quadratic ($f(n) = n^2$)
- $f(n)$ could be constant ($f(n) = 1$)
- $f(n)$ could be something entirely different.

when we are talking about big O we are talking about upper bond.

add up to 1 = $O(1)$ Always 3 operation

add up to 2 = $O(n)$ number of operation is (eventually) bounded by a multiply of n (say, 10n) count upand down

```
function countupanddown (n) {
  for (let i=0, i<=n; i++){
    console.log (i)
  }
  for (let j=n, i>=0; j--){
    console.log (j)
  }
}
  countupanddown (5)
```

Print all pairs.

```
function Printallpairs (n) {
  for (let i=0; i<n; i++){        ← O(n)
    for (let j=0; j<n; j++){      ← O(n) * O(n)
      console.log (i, j)
    }
  }
}
Printallpairs (5)
```

$O(n^2)$

Simplifying Big O expressions:

Constants Don't matter

$O(2n)$ $O(n)$
$O(500)$ $O(1)$
$O(13n^2)$ $O(n^2)$

Smaller Terms Don't matter

$O(n+10)$ $O(n)$

$O(1000n+50)$ $O(n)$

$O(n^2+5n+8)$ $O(n^2)$

Big O Shorthands.

- Analyzing complexity with big O can get complicated.
- There are several rules of thumbs that can help
- these rules won't Always work, but are helpful starting point.

① Arithmetic operation are constant
② variable assignment is constant.
③ Accessing elements in an array (by index) or object (by key) is constant

4. In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop

Example

```
function logAtLeast5(n) {
  for (var i = 1; i < Math.max(5, n); i++)
  { console.log(i); }
}
```

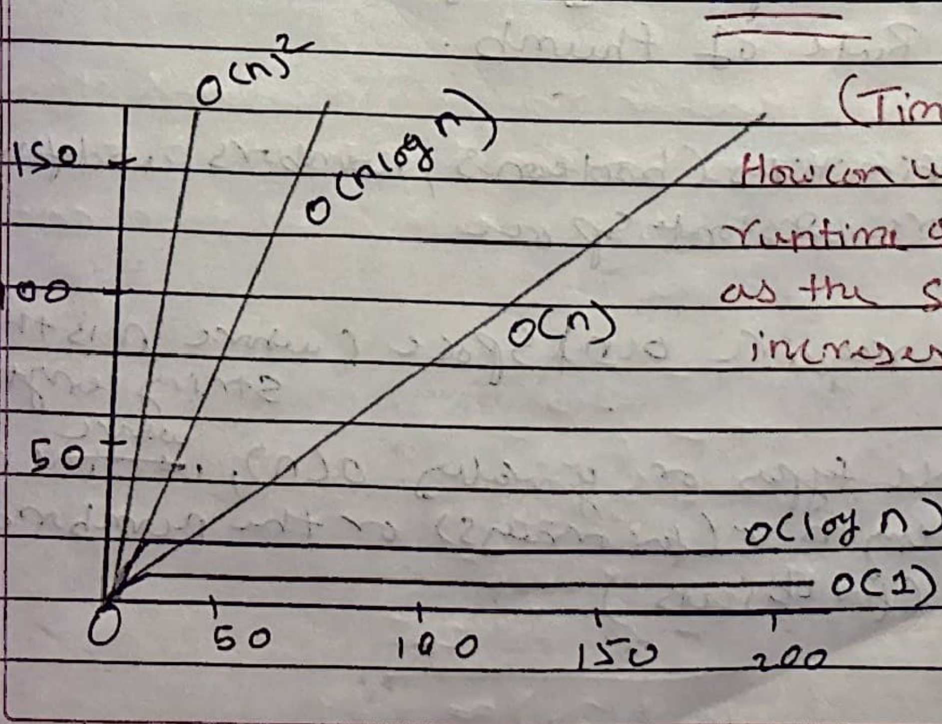
← $O(n)$

because n grows loop will grow.

same if it min

```
function logAtMost5(n) {
  for (var i = 1; i <= Math.min(5, n); i++)
  { console.log(i); }
}
```

$O(1)$



(Time complexity)

How can we analyze the runtime of an algorithm as the size of the input increases.

Space complexity.

We can also use big O notation to analyze space complexity

How much additional memory do we need to allocate in order to run the code in our algorithm.

What about inputs?
Some time you will have the terms auxiliary space complexity to refer to space required by the algorithm, not including space taken up by the inputs

auxiliary space complexity
we focus what happens what is inside of algorithm

space complexity in js
Rule of thumb.

Most primitives (booleans, numbers, undefined null) are constant space

Strings require O(n) space (where n is the string length)

Reference types are generally O(n), where ~~whether~~ n is the length (for arrays) or the number of keys (for objects)

An Example.

```
function sum (arr) {
  let total = 0;          ← one number
  for (let i = 0; i < arr.length; i++) {
2n number → total += arr[i]; (long hand total = total + arr[i])
  }
  return total;
}
```

it means we have Constant Space
O(1) Space

```
function double (arr) {
  let newArr = [];
  for (let i = 0; i < arr.length; i++) {
    newArr.push(2 * arr[i]);
  }                        ← Because this array is dependent
                             on input and if input grow
                             space grows.
  return newArr;  ← n numbers.
}

double ([1,2,3,4])
```

O(n) space

Logarithms

We have encountered some of the most common complexities : $O(1)$, $O(n)$, $O(n^2)$

Sometime big O expressions involves more ~~common~~ complex mathematical expressions are logarithm!

what is log again?

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

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

we'll omit the 2
$\log === \log_2$

we are going to see just general trend

Rule of thumb

The log of a number roughtly measured the number of times you can divide that number by 2 (before you get value that's less than or equal to one)

length
$\div 2$ (8)
$\div 2$ 4
$\div 2$ 2
1

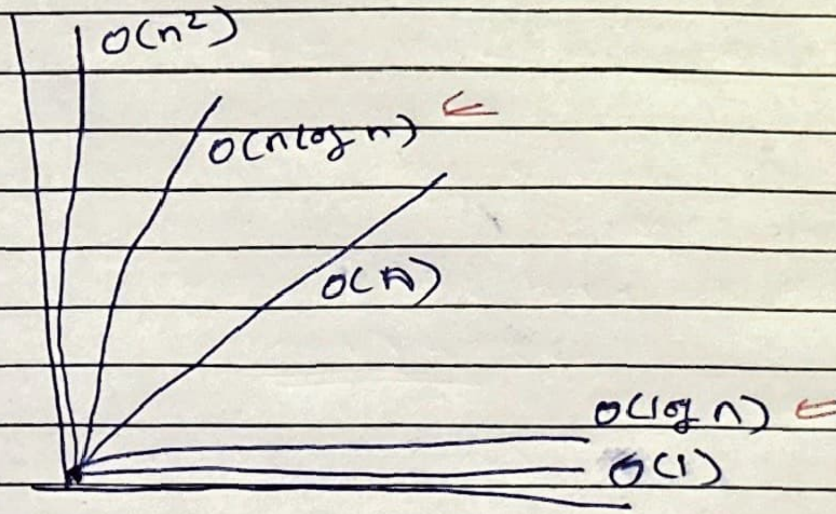
$\log(8) = 3$

25
$\div 2$ 12.5
$\div 2$ 6.25
$\div 2$ 3.125
$\div 2$ 1.5625
$\div 2$ 0.78125

$\log(25) \approx 4.64$

logarithm complexity

logarithmic time complexity is great.



who cares

Certain searching algorithm have logarithmic time complexity.

Efficient ~~st~~ sorting algorithm involves logarithms

Recursion sometimes involves logarithmic space complexity.

Recap
- To analyze the performance of an algorithm, we use Big O notation. ~~use~~
- Big O Notation can gives us a high level understanding of the time or space complexity of an algorithm.
- Big O Notation doesn't care about precision, only about general trends (linear? quadratic? constant?)
- The time or space complexity is measured by Big O