Algorithms

(& Big O Notation)

What is an algorithm?

- AKA, Data Structures and Algorithms (sometimes referred to as DSA)
- A step-by-step procedure for solving a problem
- The procedures are reusable (functions/methods)
- Algorithms refer to virtually any computational task
- DSA knowledge can be applied to ANY language or framework
- This knowledge is often tested in technical interview - so worth understanding!

Why do we care about Algorithms?

- A valid approach: ‘if my code works, it works’
- ☝️often helpful to assess how your program is working
- “How good is my code?” “Is this the best solution?”
- How performant is the solution?
- Scalability
  - How can a start up serving a customer base of 10,000 move to a customer base of 2 million?
  - How many hits per day could the API handle?
- Assumed:
  - Thoughtful variable naming
  - Tidy code, formatted correctly, attention to spacing

Comparing algorithms

- Possible to ‘time’ an algorithm…
- …but there are problems with this, not least:
  - Tiny changes are hard to measure - so for comparing fast algorithms this is especially tricky
  - Different machines will record different times (for the same code)
  - Equally, the *same* machine will record different times!
- We need a way to *objectively* distinguish between one algorithm and another
- How can we analyse and compare their performance?
- Enter: **Big O Notation**
- (O = Order of)

Big O Notation

- Every algorithm has an associated cost of execution
- Cost of execution can be expressed in:
  - TIME (how many operations are performed during execution)
  - SPACE (how much memory is utilised during execution)
- **Big O lets us express the efficiency of the algorithm in relation to the size of the input it takes**
- *How does the time/space complexity of this algorithm grow as the magnitude (size) of input increases?*
- *How does the performance of this algorithm scale as the magnitude (size) of input grows?*

How does Big O help us?

- Good to have a vocabulary to talk about how our code is performing
- Which is better - this one or that one?
- If you encounter a problem - e.g. code performing very slowly - can help identify the parts of the code that are inefficient

Big O (Time Complexity)

- We’ve covered some of the issues of using timers ❌
- Rather than counting seconds, we can count the number of simple operations the computer has to perform

What is meant by ‘simple operations’?

- multiplication, division, assignments, comparators
- Counting the number of operations is tricky!
- Remember, with Big O we’re focussing on the bigger picture, on the **general trend**

Example 1a

A function/method to add all the numbers up to *n*. If n = 5 → 15. If n = 3 → 6.

Option 1: we could use a loop

```
public int addAllNumbersTo(int n) {
     int total = 0;
     for (int i = 0; i <= n; i++) {
          total += i;
     }
     return total;
}
```

Counting Operations

assignment x 1

increment x *n*
assignment x *n*

```
public int addAllNumbersTo(int n) {
     int total = 0;
     for (int i = 0; i <= n; i++) {
          total += i;
     }
     return total;
}
```

assignment x 1

comparison x *n*

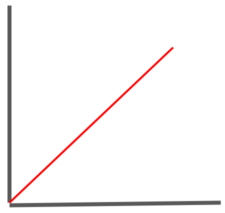
addition x *n*
assignment x *n*

- We can conclude that the no. of operations is not static…
- …**it depends on n, as n increases so does the no. of operations**
- It could be `5n + 2`, or `5n`, or `2n` (if we're only counting `total += i`)
- Doesn't matter, we just want the BIG PICTURE

Counting Operations

```
public int addAllNumbersTo(int n) {
     int total = 0;
     for (int i = 0; i <= n; i++) {
          total += i;
     }
     return total;
}
```

- The general trend is:
- A function(method) with an input of `n`, has a runtime output of `n`
- `f(n) = n`
- `O(n)`, expressed as ‘O of N’, aka **Linear Time**



Example 1b

Option 2: we could use a mathematical formula.

For now, just accept that `n * (n + 1) / 2` adds all numbers up to and including *n*.

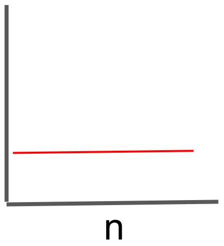
multiply x 1

```
public int addAllNumbersTo(int n) {
    return n * (n + 1) / 2;
}
```

add x 1

divide x 1

- Doesn't change if `n` increases
- Same no. of operations whether `n = 5`, `n = 50,000`, etc
- O(1), expressed as 'O of 1', aka **Constant Time**



Example 2

Suppose you have an array of elements and you want to check if there are any duplicates.

Option 1: loop through the entire array for each element, checking for a match each time.

```
public boolean hasDuplicates(int[] arr) {
     // loop through array
     for(int i = 0; i < arr.length; i++) {
          // track the index from the end of the array
          for(int j = arr.length - 1; j > 0; j--) {
               // skip if index is same
               if(i == j) continue;
               else if(arr[i] == arr[j]) {
                    return true;
               }
          }
     }

     return false;
  }
```

value:	[10,	6,	7,	9,	5,	9]
index:	0	1	2	3	4	5

Iteration 1	**i = 0**	**j = 5**
no match	0 = 10	5 = 9
no match	0 = 10	4 = 5
no match	0 = 10	3 = 9
no match	0 = 10	2 = 7
no match	0 = 10	1 = 6
i == j so skipped	0 = 10	0 = 10

Iteration 2	**i = 1**	**j = 5**
no match	1 = 6	5 = 9
no match	1 = 6	4 = 5
no match	1 = 6	3 = 9
no match	1 = 6	2 = 7
i == j so skipped	1 = 6	1 = 6
no match	1 = 6	0 = 10

& so on…

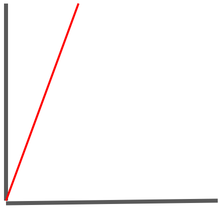
value:	[10,	6,	7,	9,	5,	9]
index:	0	1	2	3	4	5

Iteration 4	**i = 3**	**j = 5**
Match found!	3 = 9	5 = 9
	3 = 9	4 = 5
	3 = 9	3 = 9
	3 = 9	2 = 7
	3 = 9	1 = 6
	3 = 9	0 = 10

```
public boolean hasDuplicates(int[] arr) {
      // loop through array
    for(int i = 0; i < arr.length; i++) {
         // track the index from the end of the array
        for(int j = arr.length - 1; j >= 0; j--) {
            // skip if index is same
            if(i == j) continue;
            else if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
  }
```

O(n)

O(n)



- With Big O, always consider worst case scenario
- We could find the matching pair immediately, but could also work through entire array and find no matches
- With both loops, as `n` grows, so do the no. of operations
- `O(n^2)`, expressed as ‘O of N Squared’, aka **Quadratic Time**

Example 3

Logarithm is the inverse of exponentiation

roughly measures the number of times you can divide that number by 2 before you get to a value that's less than (or equal to) 1.

`log(8) = 3`

- 8 / 2 = 4 $\boxed{1}$
- 4 / 2 = 2 $\boxed{2}$
- 2 / 2 = 1 $\boxed{3}$

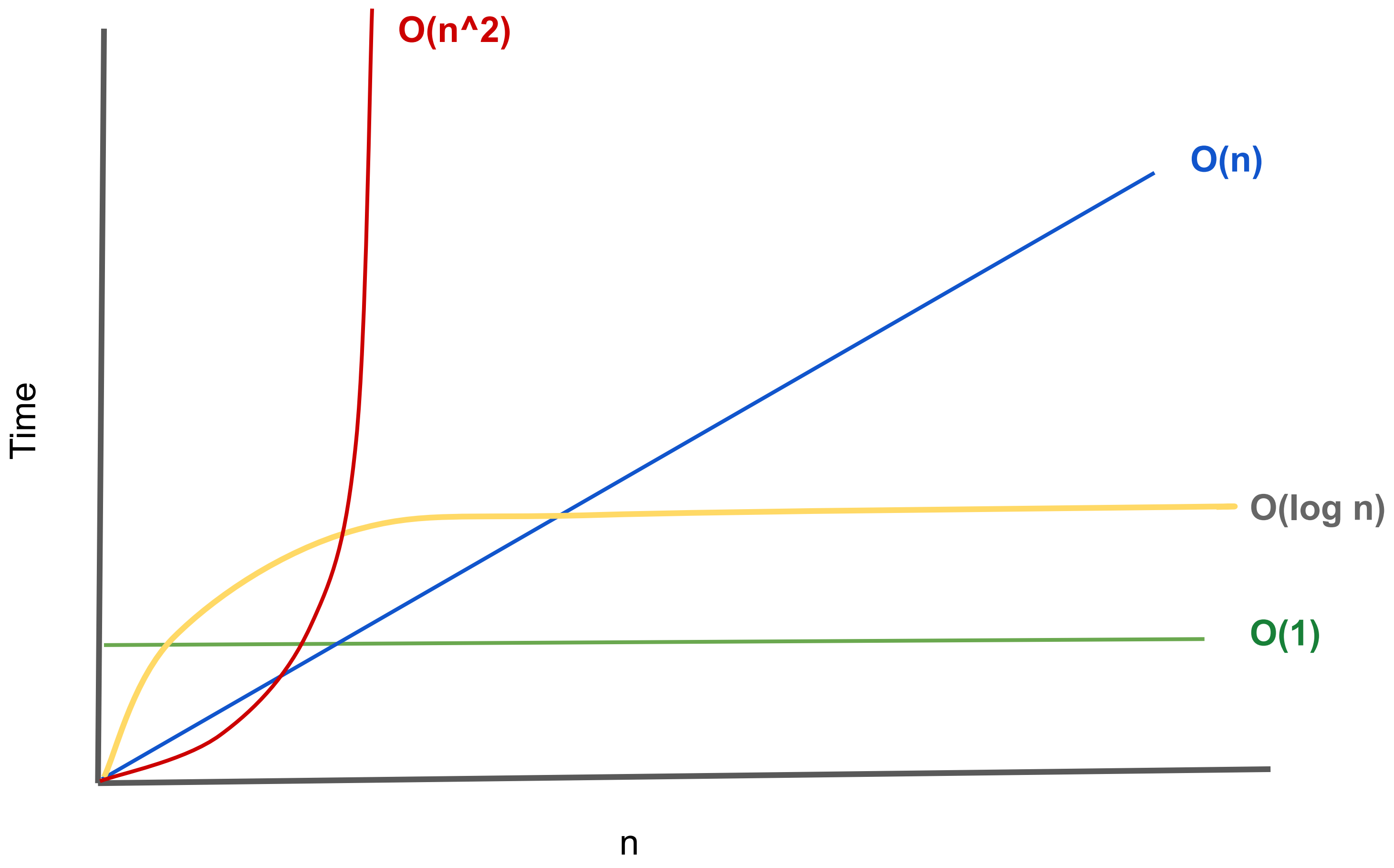
Example 3 (cont'd)

```
public int logLookUp(int n) {
    for (int i = 1; i < n; i = i * 2) {
        System.out.println("looking up element at index " + i;
    }
}
```

- Loop is constructed in such a way that we effectively divide the number of remaining elements by 2 on each iteration, creating logarithmic behaviour
- Suppose n = 10
  - Iteration 1 // `"looking up element at index 1" (i = 2)`
  - Iteration 2 // `"looking up element at index 2" (i = 4)`
  - Iteration 3 // `"looking up element at index 4" (i = 8) etc.`
- No. of operations is related to magnitude of n, but some improvement on O(n)
- O(log n), expressed as 'O of Log N', aka **Logarithmic Time**
- Famous example: **Binary Search** (https://www.youtube.com/watch?v=J3hM7xE9aFc)

A summary (in ascending order of runtime complexity)

- **O(1)**: the algorithm has the same (or, ***constant***) complexity, no matter what input you provide.
- **O(log n)**: having a **logarithmic** relationship with it's input
- **O(n)**: a **linear** growth relationship between input and complexity. It says that an algorithms efficiency is directly proportional to the magnitude of its input.
- **O(n^2)**: this notation depicts **quadratic** time. Every effort should be made to find a more efficient solution, as an algorithm with O(N2) is inefficient, and not very scalable.
- **O(n^3)**: **cubic** time is extremely inefficient and not at all scalable



O(n^2)
O(n)
O(log n)
O(1)
Time
n

Points to remember

- Big O == Big Picture
- You're trying to identify the overall trend
- Look for the worst case scenario
- In more complicated algorithms you may find multiple Big O terms, always go for the dominant term (weakest link)
- Practise! Codewars, Leetcode, Exercism
- https://www.learnhowtoprogram.com/computer-science/big-o-notation-and-binary-trees/big-o-practice
- It's not the be-all-and-end-all of what makes good code