Learning Aims

- Measures and methods to determine the efficiency of different algorithms
- Understand the Big O notation (constant, linear, polynomial, exponential and logarithmic complexity)

What is Big O?

- Evaluate the complexity of the algorithm
- Show how the time / memory / resources increase as the data size increases
- Evaluate worst case scenario for the algorithm

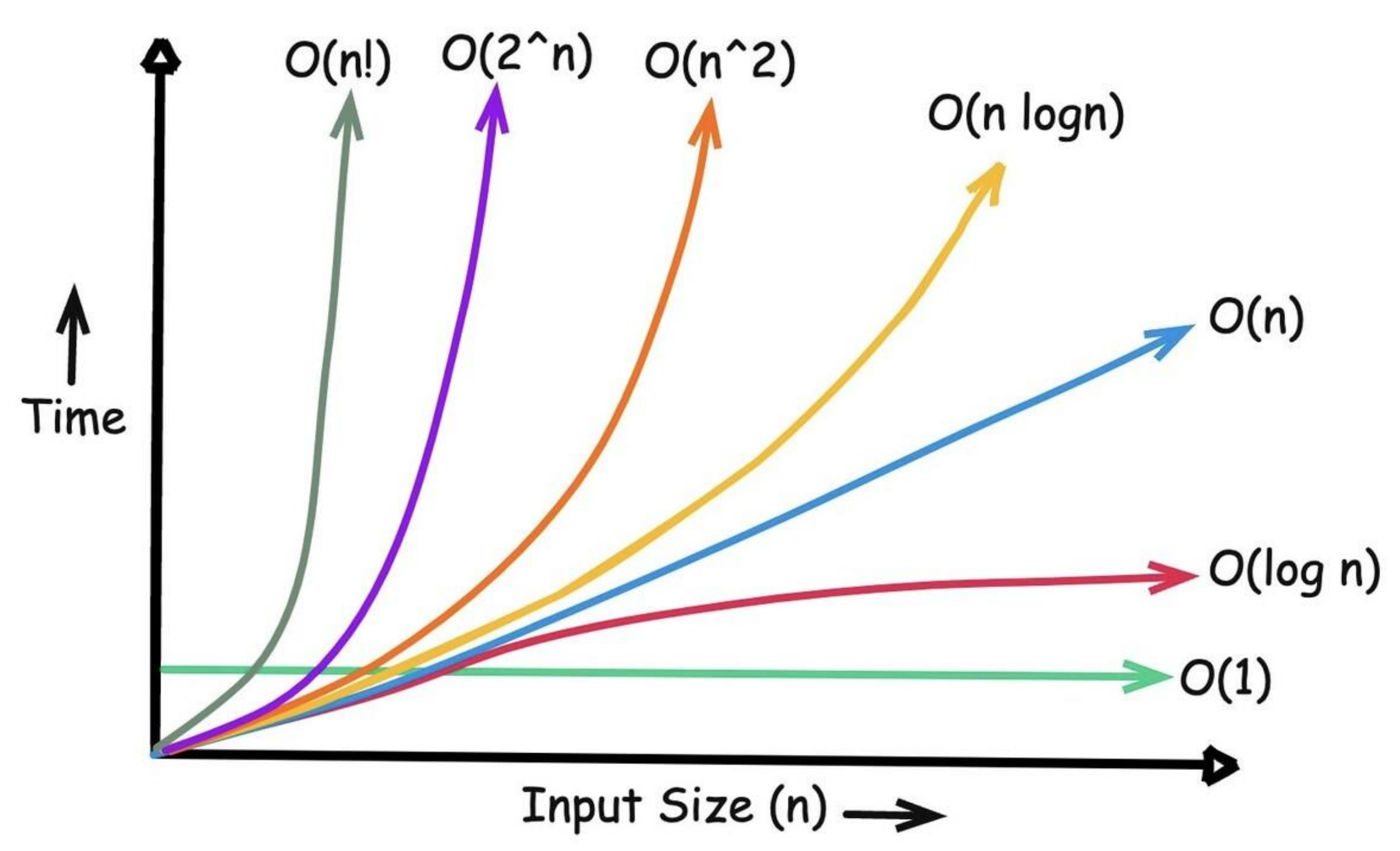
Time Complexity - How the time scales as data size increases

Space Complexity – how much memory is required

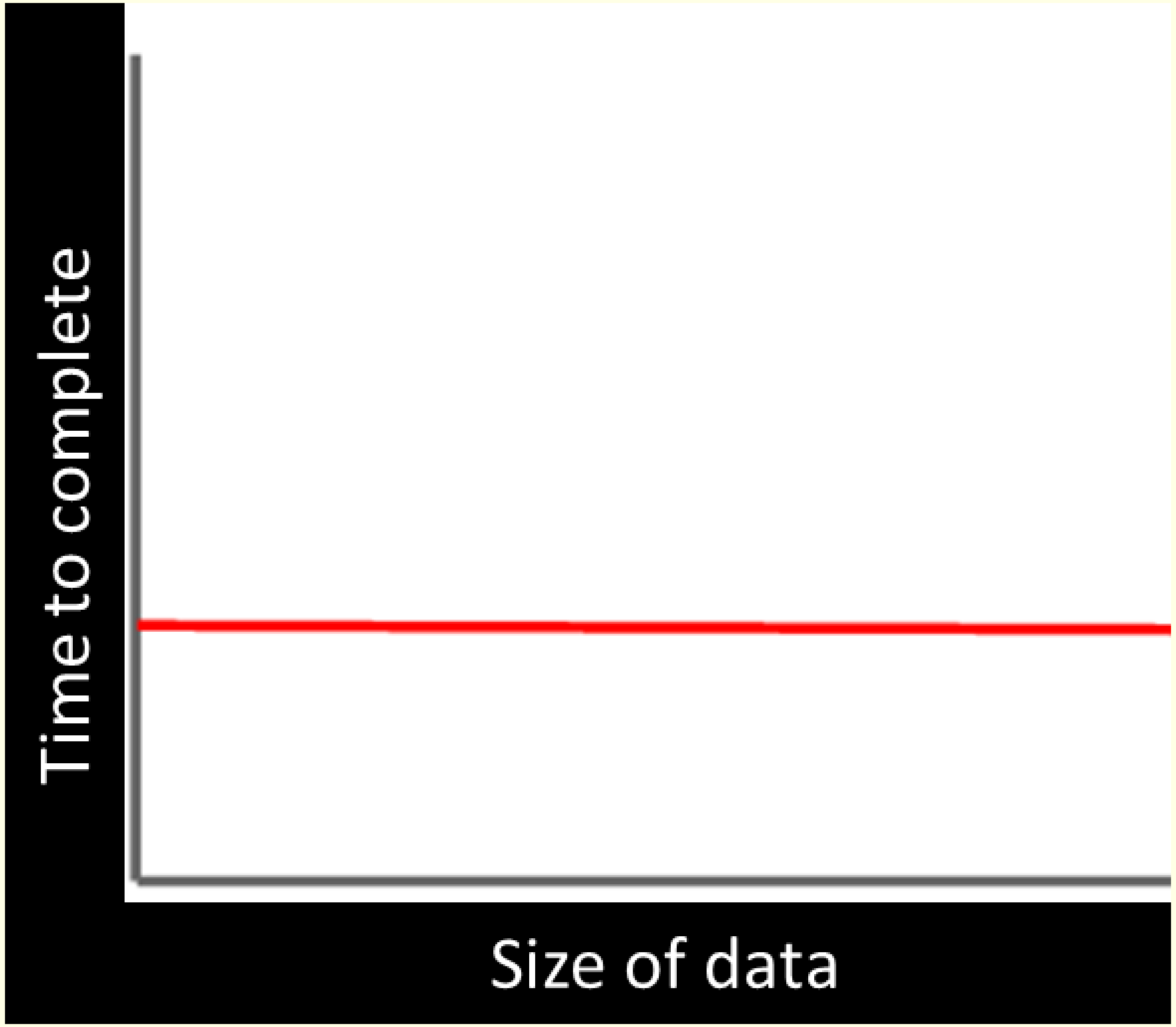
Types of complexity

- constant
- logarithmic
- linear
- linearithmic
- polynomial
- exponential

Notation	Description	Name	Example
$O(1)$	**Same time** regardless of the size of the data set.	constant	Determining if a number is even or odd Stack Push or Pop
$O(\log N)$	Halves each time and scales well (meaning increasing the data size will only result in a slight increase in time)	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree
$O(N)$	The time will increase in **direct proportion** to the number of items	linear	Finding an item in an unsorted list
$O(n \log N)$	Every item is processed (n), but also divided or merged (log n).	linearithmic	Merge & Quick sort (optimised pivot)
$O(n^2)$	proportional to the square of the size of the data set.	polynomial	Nested loops Bubble and Insertion sorts Quick sort with a poorly chosen pivot
$O(k^n)$	Time doubles with each new element. Extremely inefficient.	exponential	Recursive algorithms


O(n!)
O(2^n)
O(n^2)
O(n logn)
O(n)
O(log n)
O(1)
Time
Input Size (n)

Constant complexity O(1)



- Algorithms that show a constant complexity take the same time to run regardless of the size of a data set.
- An example of this is pushing an item onto, or popping an item off, a stack.
- No matter how big the stack, the time to push or pop remains constant.
- Thus this action gets a Big O notation of O(1).

Example O(1)

```
def func(arr):
  return arr[0]
```

- This function returns the first element of the input array and will always take a single operation to complete, regardless of the size of the input.

Logarithms O(log N)

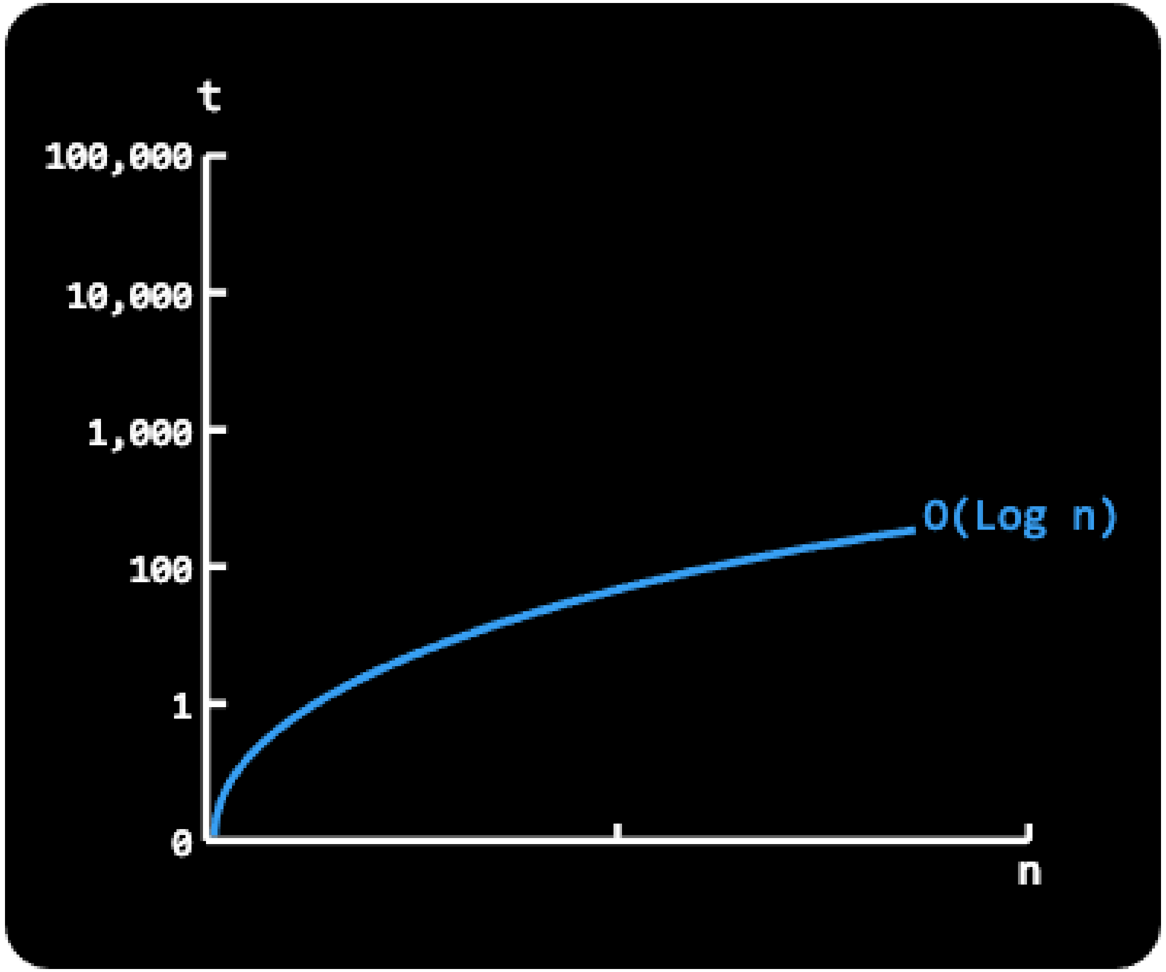
Logarithmic time complexities usually apply to algorithms that divide problems in half every time, like any **Divide and conquer algorithms.**

This is for algorithms that halves each time and scales well (meaning increasing the data size will only result in a slight increase in time)

Because the algorithm halves each time from a large data it starts off with a really large search time then flattens out over time.

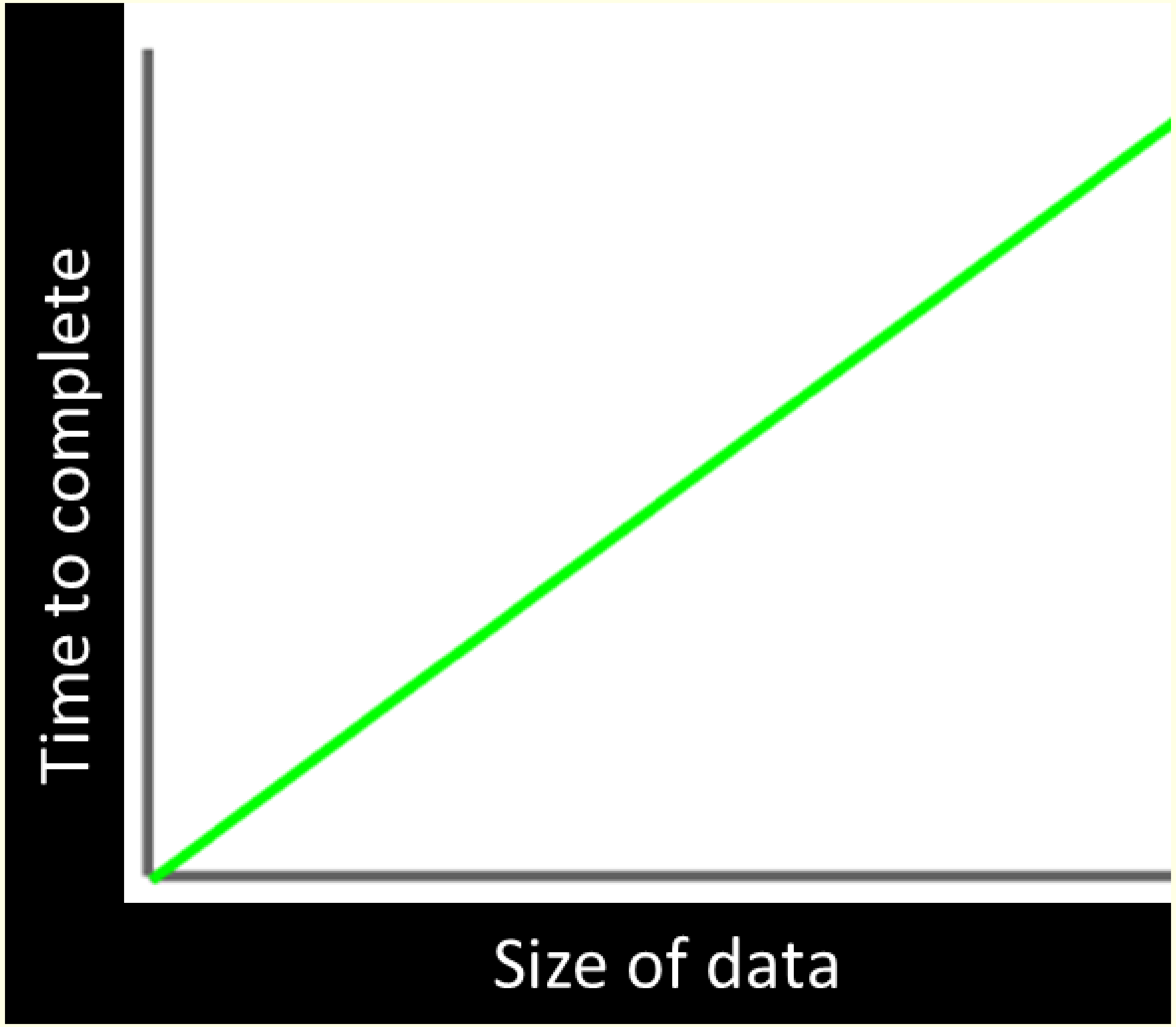
A data set containing 10 items will take 1 second
A data set containing 100 items takes 2 seconds.
A data set containing 1000 items takes 3 seconds.

A good example is a **binary search.** As the size of the data set doubles, the number of items to be checked only increases by 1 (scalable)



Linear Complexity O(N)

- Algorithms with linear complexity increase at the same rate as the input size increases.
- If the input size doubles, the time taken for the algorithm to complete doubles.
- An example of this is the average time taken to find an element using a linear search, the bigger the dataset the longer it takes to search.
- Thus linear complexity gets a Big O notation of O(n).

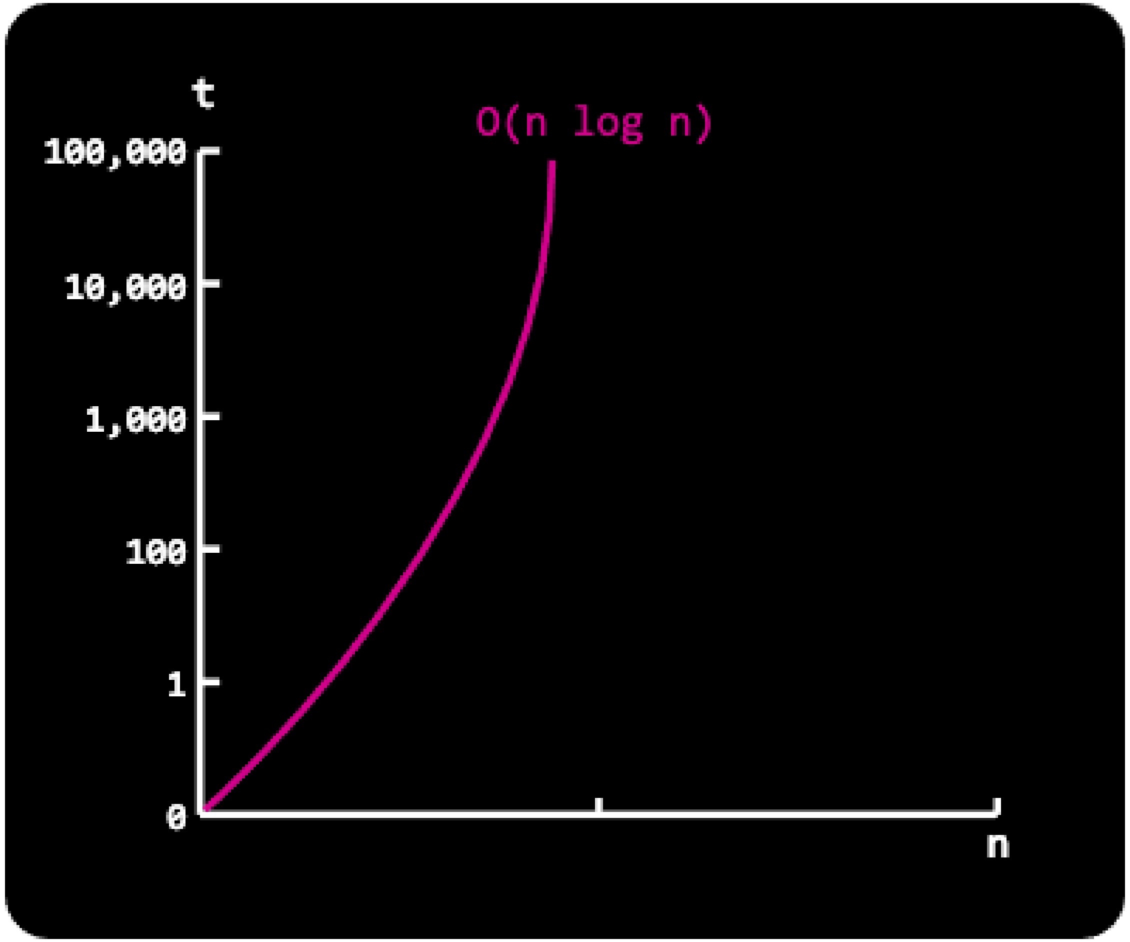


Linearithmic O(n log n)

Every item is processed (n), but also divided or merged (log n).

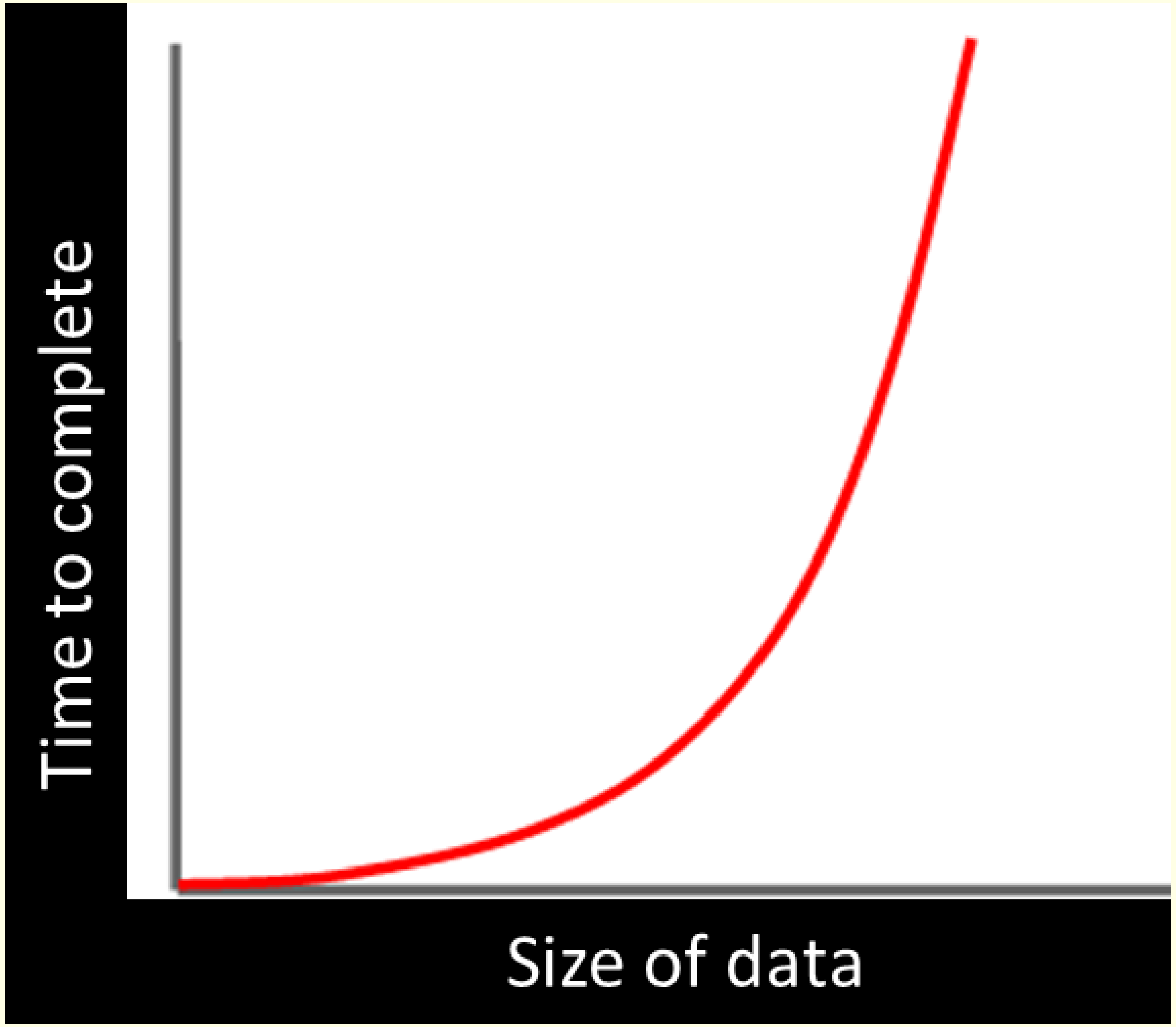
Linearithmic time complexity it's slightly slower than a linear algorithm. However, it's still much better than a quadratic algorithm

Efficient sorting algorithms like merge sort & quicksort (with well chosen pivot value)



Polynomial Complexity O(n²)

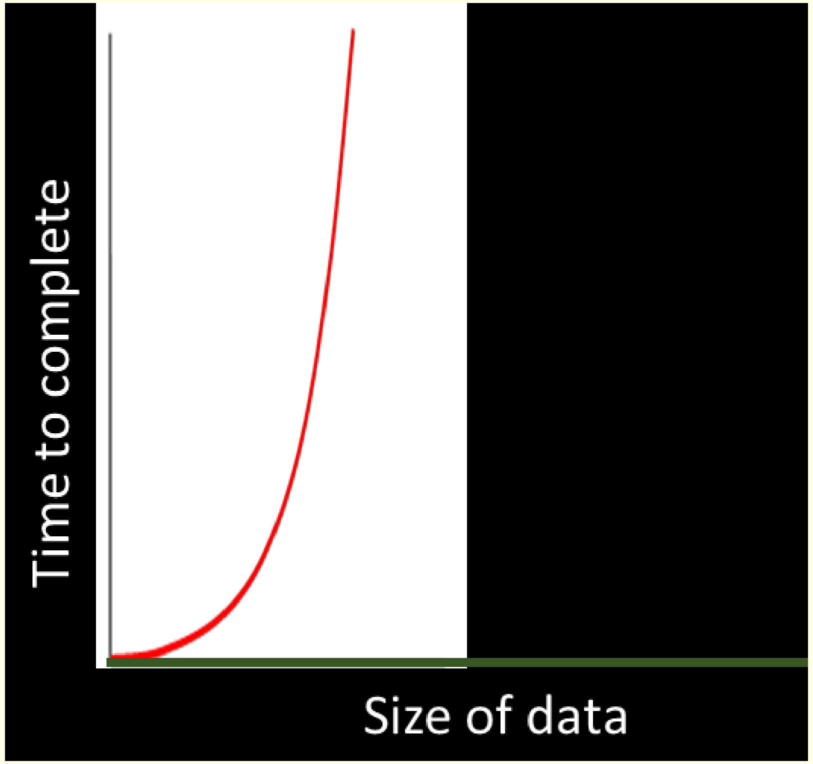
- This represents an algorithm whose performance is directly linked to the square of the size of data input set.
- Algorithms that use nested loops.
- Bubble and Insertion sorts are common examples of this because if you **double the amount of items it will quadruple the runtime.**



Exponential Complexity $O(k^n)$

- Algorithms that do not scale well as the input (n) gets larger the time taken increases at a rate of k^n Where is K is the constant value

- Looking at the graph it would appear that exponential and polynomial are very similar.
- As can be seen on in the table the growth is much faster.
- Examples of this is recursive algorithms that calls itself twice (for example Fibonacci algorithm)



n	n^2 (Polynomial)	2^n (Exponential)
1	1	2
10	100	1,024
20	400	1,0485,76
30	900	1,073,741,824

In this recursive algorithm it calls itself twice – so it is Exponential Complexity $O(k^n)$

```
def Fibonacci(n):
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
print(Fibonacci(9))
```

To calculate a Fibonacci number can be done using iteration and would be much better in terms of time complexity (linear O(n))

Working out the complexity of a algorithm

$$steps = 7n^3+n^2+4n+1$$

constant operation (+1)

4n → linear part

n^2 → quadratic part

7n^3 → cubic part

```
def complexity_demo(n):
    count = 0  #

for i in range(4 * n ):
      count += 1

for i in range(n):
      for j in range(n):
         count += 1


    for k in range(7):  # constant 7 repetitions
       for i in range(n):
          for j in range(n):
             for l in range(n):
                count += 1

    return count
```

Simplifying Big O

When writing Big O, we ignore:
- *constant factors (for example, 3n to O(n)*
- *less significant terms*

This helps focus on the most important factor that affects how the algorithm

- $7n^3+n^2+4n+1$ steps.
- Now look at how n increases.

n	$7n^3$	n^2	4n	1	Total
1	7	1	4	1	13
10	7,000	100	40	10	7,141
100	7,000,000	10,000	400	100	7,010,401
1000	7,000,000,000	1,000,000	4,000	1000	7,001,004,001

- The larger n gets the less impact n^2+4n+1 has on the total compared to $7n^3$
- Meaning our algorithm $7n^3+n^2+4n+1$ becomes
- $O(n^3)$.

Another Worked Example

- **An algorithm has a complexity of $3n^5 + 2n^3 + 2^n + 7$ steps to run – let's work that into Big O.**

- First spilt each term into types of complexity.
- $3n^5$ & $2n^3$ both have polynomial complexity
- 2^n is exponential
- And 7 is constant
- Therefore 2^n is the most complex
- **So the complexity as Big O is $O(2^n)$**

Working it out

Here is an algorithm – lets workout its steps and then notate it.

Runs **n times.**

```
for x = 1 to n
  for y = 1 to n
    z = data[y]
    print z
  next
next
print 'finished'
```

Runs **n times.**

n^2 $(n * n) \rightarrow$ quadratic part as nested.

2 constant-time operations so $2n^2$

Runs **once**

$$\mathbf{n * 2n + 1 \rightarrow n2 \quad + 1}$$

The Big O notation is $O(n^2)$.