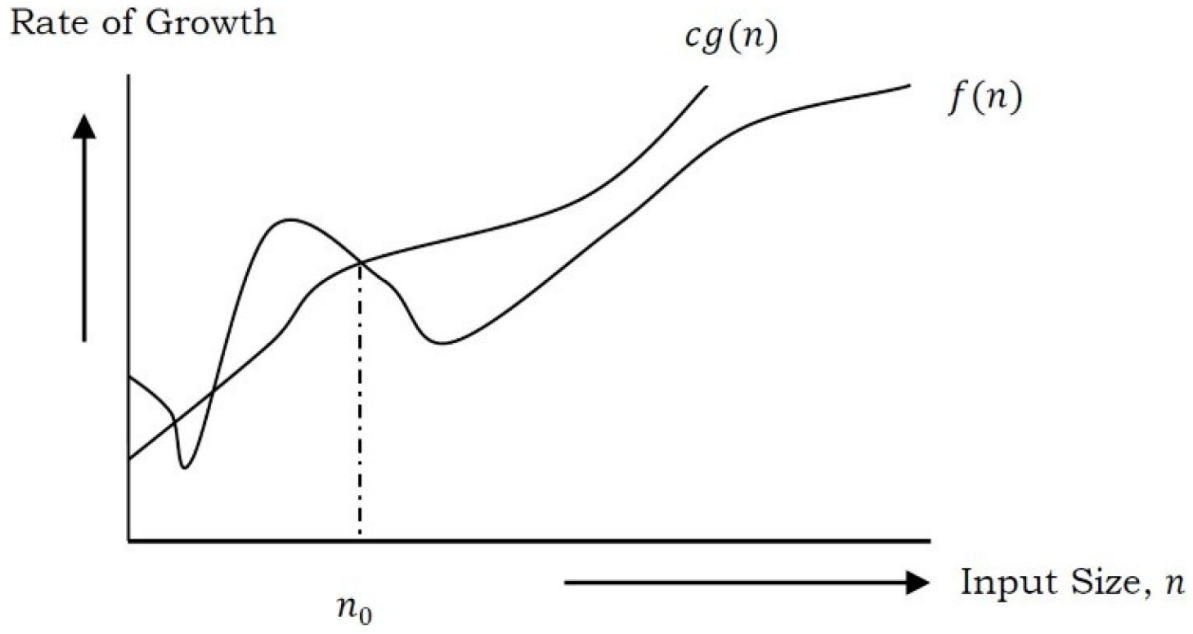
Big O Notation-Observations

- With big O notation we express the runtime in terms of—brace yourself—how quickly it grows relative to the input, as the input gets arbitrarily large.
- Be careful to differentiate between:
    - Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
    - Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

- Complexity affects performance but not the other way around.
- The time required by a function/procedure is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:
    - one arithmetic operation (e.g., +, *).
    - one assignment (e.g. x := 0) • one test (e.g., x = 0)
    - one read (of a primitive type: integer, float, character, boolean)
    - one write (of a primitive type: integer, float, character, boolean)

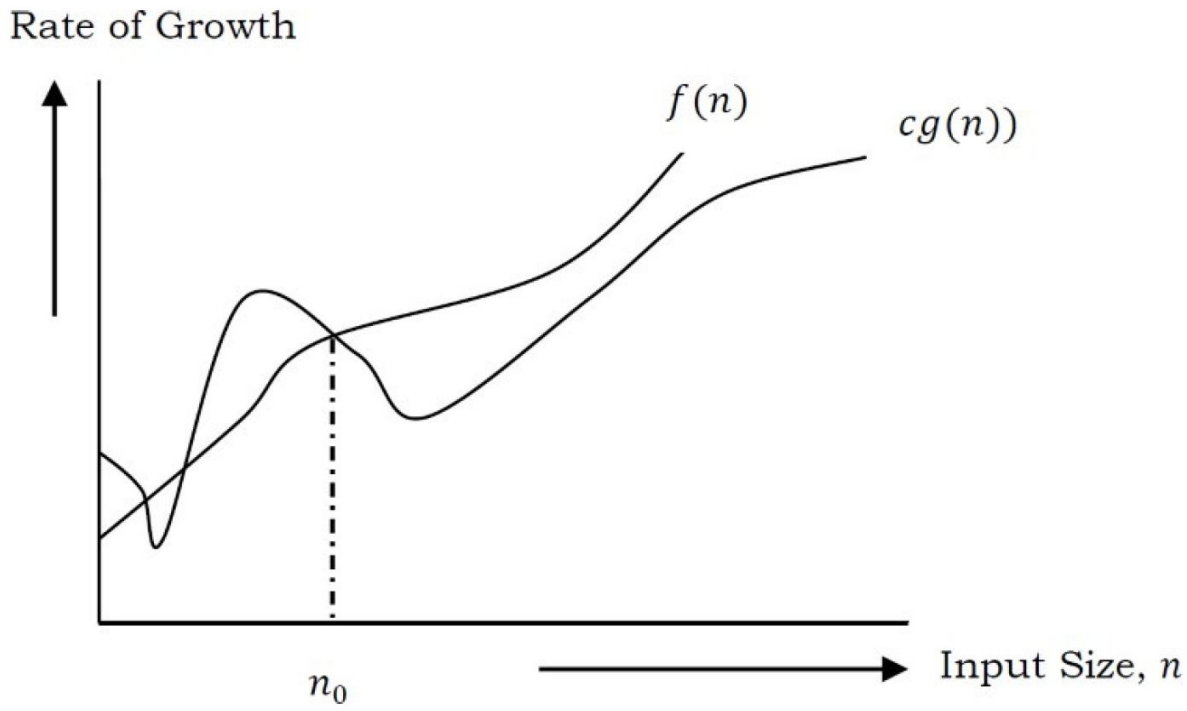
Big-O Notation [Upper Bounding Function]

- O(g(n)) is the set of functions with smaller or the same order of growth as g(n). For example;
  O(n2) includes O(1), O(n), O(nlogn), etc.



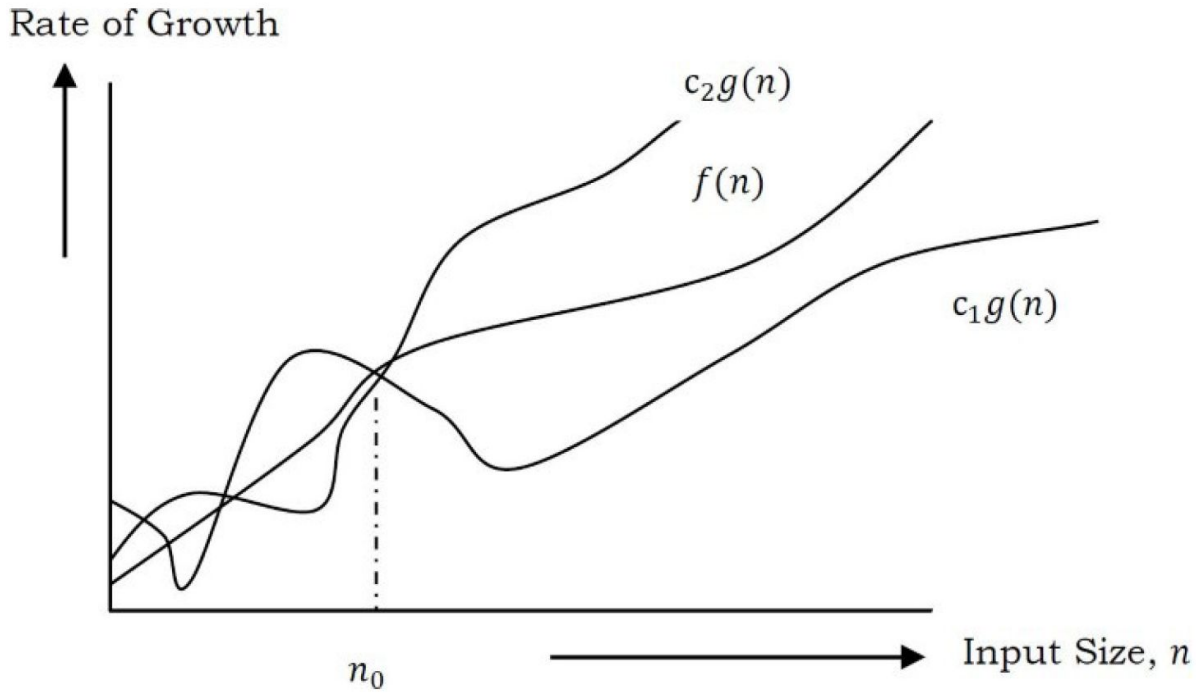
Omega Notation [Lower Bounding Function]

- give the largest rate of growth g(n) which is less than or equal to the given algorithm's rate of growth f(n).



Theta-Θ Notation [Order Function]

- This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth.



Guidelines for Asymptotic Analysis

- Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
//executes n times
for(i=1;i<=n;i++)
{
      m=m+2;//constant time c
}
```

- Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//executes n times
```

```
for(i=1;i<=n;i++)
{
//inner loop executes n times
     for (j=1;j<=n;j++)
     {
            m=m+2;//constant time c
     }
}
```

- Consecutive statements: Add the time complexities of each statement

```
x=x+1;//constant time
```

```
//executes n times
for (i=1;i<=n;i++)
      m=m+2;//constant time
//outer loop executes n times
for(i=1;i<=n;i++)
{
      //inner loop executed n times
      for(j=i;j<=n;j++)
            k=k+1;//constant time
}
```

- If-then-else statements: Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).
- Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½).

Constant time: O(1)

The following operations take *constant time*:

- Assigning a value to some variable
- Inserting an element in an array
- Determining if a binary number is even or odd.
- Retrieving element i from an array
- Retrieving a value from a hash table(dictionary) with a key

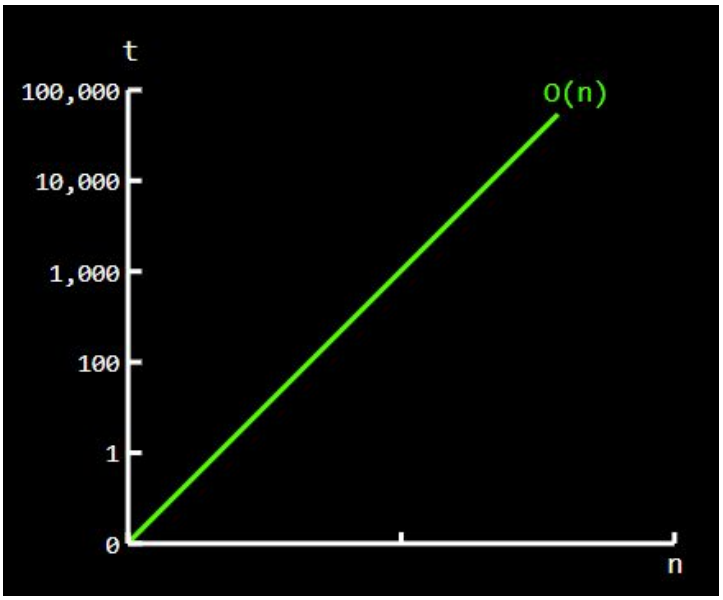
They take constant time because they are "simple" statements. In this case we say the statement time is O(1)

Linear time: O(n)

The next loop executes N times, if we assume the statement inside the loop is O(1), then the total time for the loop is N*O(1), which equals O(N) also known as *linear time*:

```
      for (int i = 0; i < N; i++) {
            //do something in constant time...
      }
```

- Finding an item in an unsorted collection or a unbalanced tree (worst case)
- Sorting an array via bubble sort



- ## Quadratic time: $O(n^2)$
- In this example the first loop executes N times. For each time the outer loop executes, the inner loop executes N times. Therefore, the statement in the nested loop executes a total of N * N times. Here the complexity is O(N*N) which equals $O(N^2)$. This should be avoided as this complexity grows in *quadratic time*

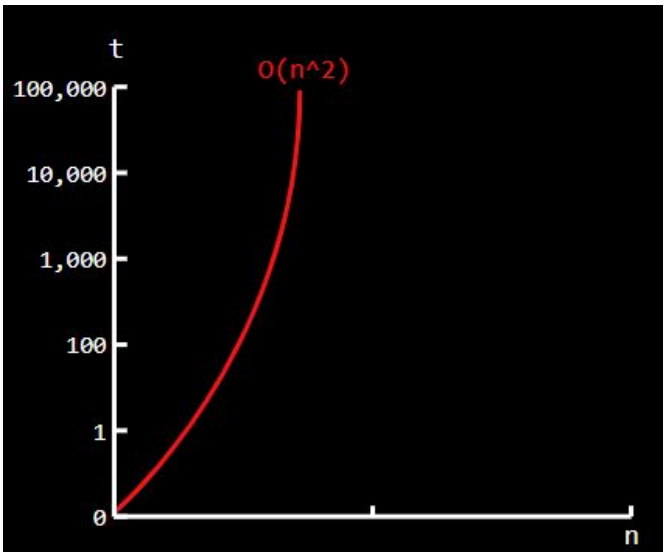
```
for (int i=0; i < N; i++) {
            for(int j=0; j< N; j++){
                    //do something in constant time...
            }
}
```

Some extra examples of quadratic time are:

- Performing linear search in a matrix
- Time complexity of quicksort, which is highly improbable as we will see in the Algorithms section of this website.
- Insertion sort

Algorithms that scale in quadratic time are better to be avoided. Once the input size reaches n=100,000 element it can take 10 seconds to complete. For an input size of n=1’000,000 it can take ~16 min to complete; and for an input size of n=10’000,000 it could take ~1.1 days

to complete...you get the idea.



Logarithmic time: O(Log n)

Logarithmic time grows slower as N grows. An easy way to check if a loop is log n is to see if the counting variable (in this case: i) doubles instead of incrementing by 1. In the following example int i doesn't increase by 1 (i++), it doubles with each run thus traversing the loop in log(n) time:

```
for(int i=0; i < n; i *= 2) {
            //do something in constant time...
}
```

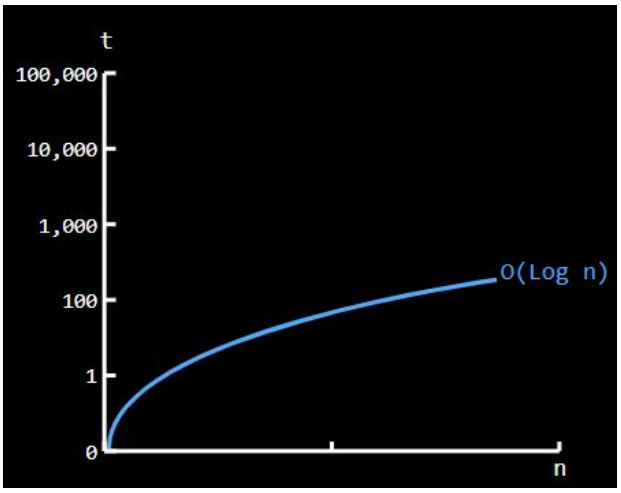
Some common examples of logarithmic time are:

- Binary search
- Insert or delete an element into a heap

Don't feel intimidated by logarithms. Just remember that logarithms are the inverse operation

of exponentiating something. Logarithms appear when things are constantly halved or doubled.

Logarithmic algorithms have excellent performance in large data sets:



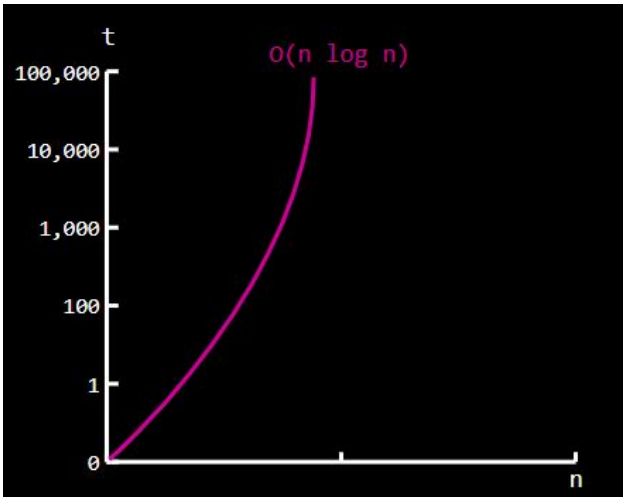
Linearithmic time: O(n*Log n)

Linearithmic algorithms are capable of good performance with very large data sets. Some examples of linearithmic algorithms are:

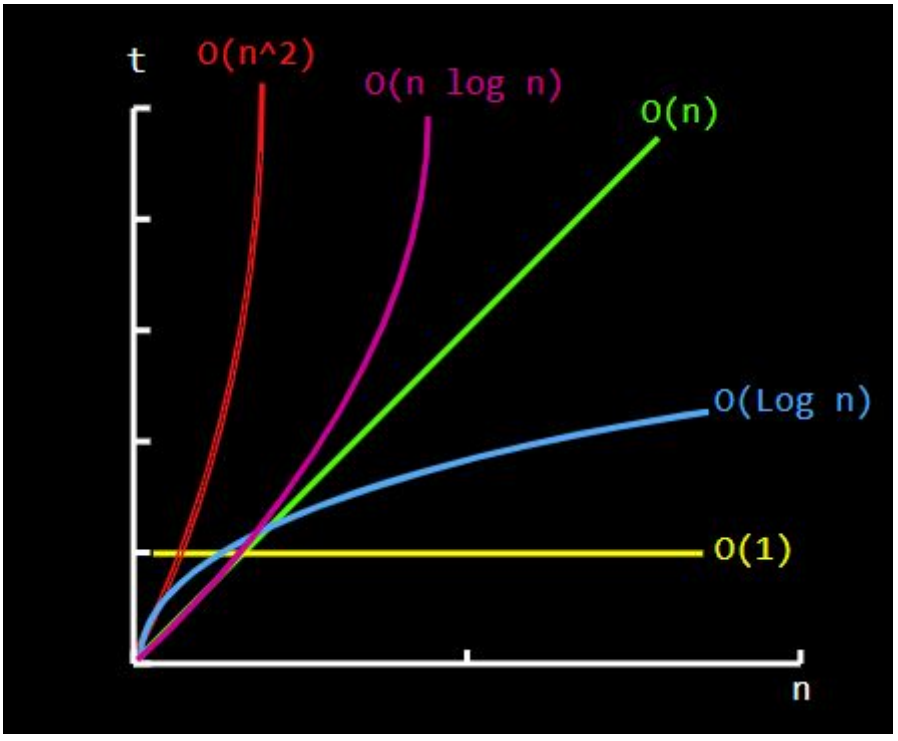
- heapsort
- merge sort
- Quick sort

We'll see a custom implementation of Merge and Quicksort in the algorithms section. But for now the following example helps us illustrate our point:

```
for(int i= 0; i< n; i++) { // linear loop  O(n) * ...
          for(int j= 1; j< n; j *= 2){ // ...log (n)
                  //do something in constant time...
          }
}
```



In summary:



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)