**Introduction:**

Worst-case time complexity denoted as which defined as the maximum amount of time taken on any input of size

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform.

The **order of magnitude** function describes the part of  that increases the fastest as the value of n increases. Order of magnitude is often called **Big-O** notation (for “order”) and written as

It provides a useful approximation to the actual number of steps in the computation.

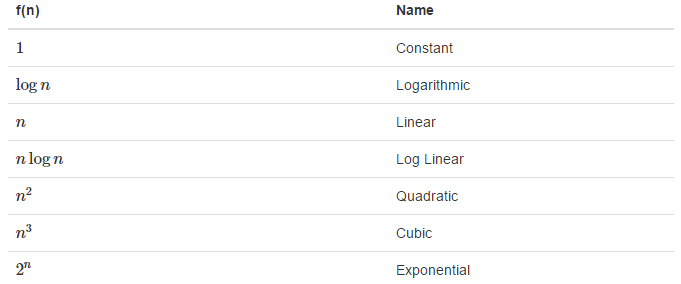
The function  provides a simple representation of the dominant part of the original

For example:

As n gets large, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for  then we can drop the 1 and simply say that the running time is .

suppose that for some algorithm, the exact number of steps is . When  is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as  gets larger, the  term becomes the most important.

Table of common time complexities



a=5

b=6

c=10

**for** i **in** range(n):

**for** j **in** range(n):

x = i \* i

y = j \* j

z = i \* j

**for** k **in** range(n):

w = a\*k + 45

v = b\*b

d = 33

The number of assignment operations is the sum of four terms.

The first term is the constant 3, representing the three assignment statements at the start of the fragment.

The second term is , since there are three statements that are performed  times due to the nested iteration.

The third term is , two statements iterated  times.

Finally, the fourth term is the constant 1, representing the final assignment statement.

This gives us

By looking at the exponents, we can easily see that the  term will be dominant and therefore this fragment of code is

Note that all of the other terms as well as the coefficient on the dominant term can be ignored as n grows larger.

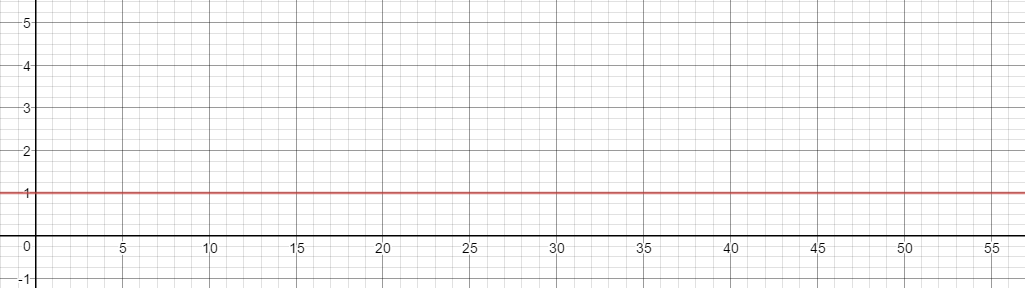
**The Constant Function (Constant Time):**

Constant function is the function whose value is the same for every input value is constant because no matter what the value of the output will always be 2.

An algorithm is said to be constant also written if the value of doesn’t depend on size of , an example is adding two 32 bit-integer, this single arithmetic operation will take the same amount of time as adding two 64 bit-integer.

Other examples of the constant are assigning values and comparing two numbers.

Graph of a constant function is always a horizontal line



**The Logarithmic Function (Logarithmic Time):**

An algorithm is said to take Logarithmic time if , in another word for some constant this function is defined as follow:

if and only if

Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search

Notice: the most common base for the logarithm function in computer since is 2, base is in , and this is due the use of the binary system by computers.

A very good programming example is an algorithm that tries to cut a list in half and then cut the first half in two and so until we only have one element left in this list, notice that a list with 10 elements will take less operations to finish then a list with 20 elements, the question is how many operations will each of these list will take to finish is job?

The first list has 10 elements:

The first list will take 4 operations specifically iterations to cut down in halves.

The second list with 20 elements:

The second list will take 5 iterations.

Notice the in represents the number of elements in a list or the number of characters in string.

Python example:

**import** random  
  
**def** print\_half():  
 x = [random.randrange(0, 100) **for** i **in** range(21)]  
 **print** x  
 counter = 0  
 **while** len(x) != 1:  
 x = [x[i] **for** i **in** range(0, len(x) / 2)]  
 **print** x  
 counter += 1  
  
 **print** counter  
  
print\_half()

output:

[1, 8, 96, 46, 16, 34, 14, 28, 99, 60, 96, 38, 18, 73, 58, 77, 49, 65, 55, 42, 19]

[1, 8, 96, 46, 16, 34, 14, 28, 99, 60]

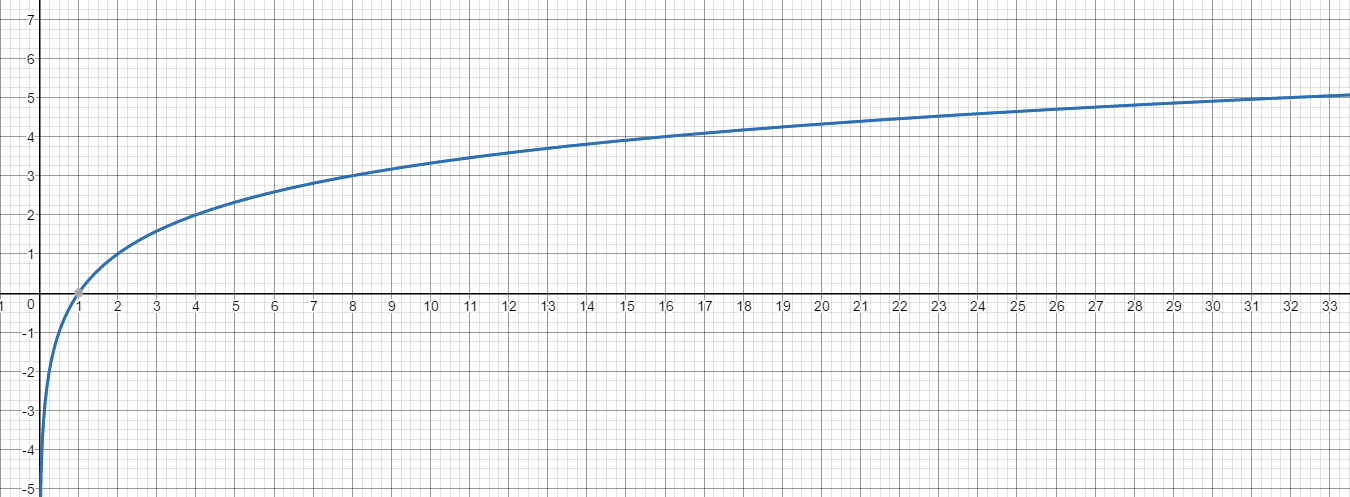
[1, 8, 96, 46, 16]

[1, 8]

[1]

4

Graph of the logarithmic function looks like this:



**Linear Function (Linear Time):**

An algorithm is said to be Linear time or

This function arises in algorithm analysis any time we have to do a single basic operation number of times, for example comparing the number x with all elements in list y.

Example of linear time is:

**import** random  
  
x = [random.randrange(0, 100) **for** i **in** range(10)]  
**for** e **in** x:  
 **print 'current element is {}'**.format(e)

output:

current element is 27

current element is 22

current element is 67

current element is 35

current element is 43

current element is 64

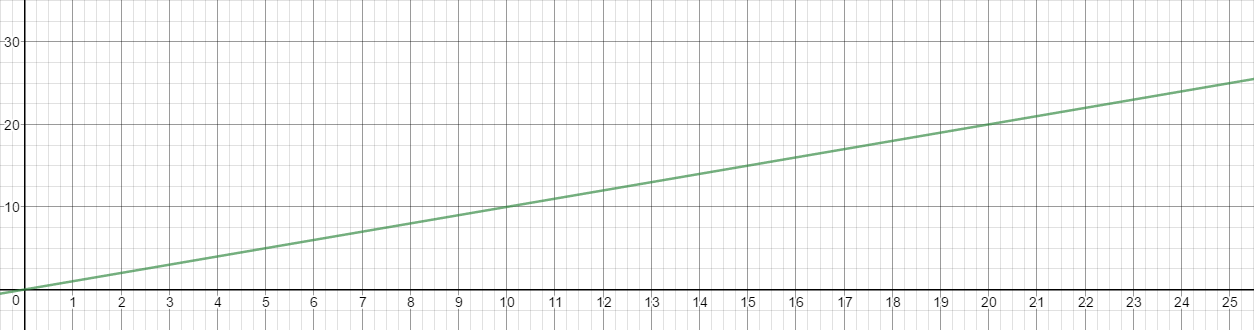
current element is 51

current element is 78

current element is 46

current element is 86

Linear function can be graphically represented as follow:



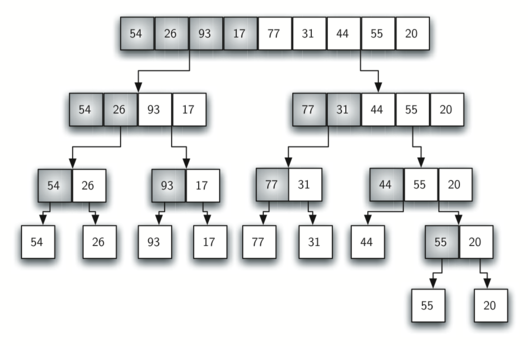
**The N-Log-N function (Linearithmic time – log linear time):**

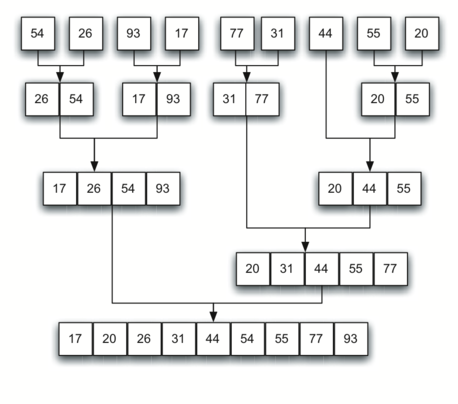
An algorithm us said to run in Linearithmic time if

This function assign to an input the value of times the logarithm base-2 of , this function grows more rapidly than the linear function and a lot less than the quadratic function- will be discussed next-

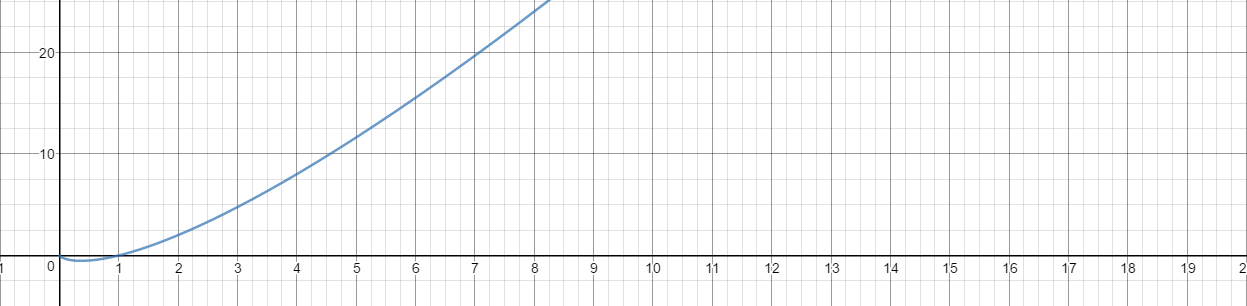
Comparison sorts require at least Linearithmic number of comparison in the worst case.

A good example of the log linear time is Merge sort algorithm





Linearithmic time can be graph as follow:



**Quadratic Function (Quadratic Run Time)**

An algorithm is said to be quadratic if ,

the main reason why the quadratic function appears is the nested loops, where the inner loop is performing linear number of operations and the outer loop is also performing a linear number of operations , thus in this case the whole algorithm perform operations.

A good example is to check how many a single item occurred in a list, this required comparing each single element in the list with all the element in the list including itself,

**import** random  
  
alist = [random.randrange(1, 10) **for** x **in** range(10)]  
**print** alist  
counter = 0  
**for** o **in** alist:  
 **for** i **in** alist:  
 counter += 1  
 **print 'comparing {} with {} operation number {}'**.format(o, i, counter)

output:

[9, 8, 7, 8, 6, 9, 8, 5, 5, 9]

comparing 9 with 9 operation number 1

comparing 9 with 8 operation number 2

comparing 9 with 7 operation number 3

comparing 9 with 8 operation number 4

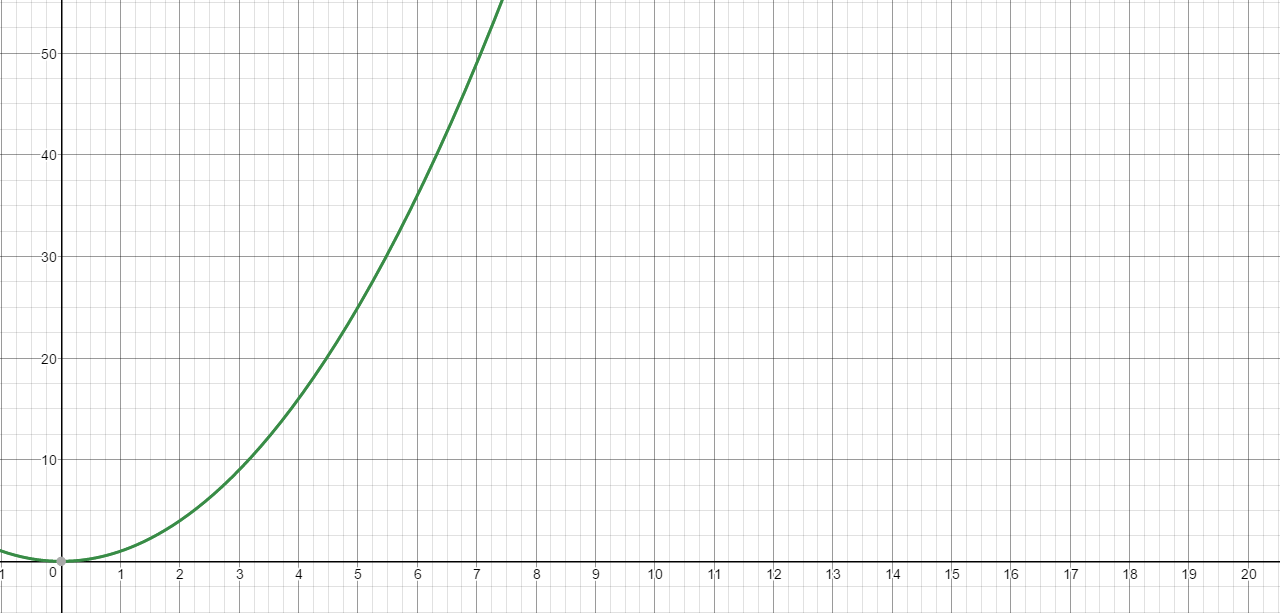
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

comparing 9 with 5 operation number 99

comparing 9 with 9 operation number 100

quadratic function graph:



**Cubic function**

An algorithm is said to be cubic if ,

Which assigns to an input value the product of with itself three times.

This function appears less frequently in the context of algorithm analysis than the Constant, Linear and Quadratic.

Example:

**def** counter3(n):  
 counter = 0  
 **for** i **in** range(0, n):  
 **print 'i ='**, i  
 **for** j **in** range(0, n):  
 **print 'j ='**, j  
 **for** k **in** range(0, n):  
 **print 'k ='**, k  
 counter += 1  
  
 **return** counter  
  
  
**print** counter3(2) *# 8*

output

i = 0

j = 0

k = 0

k = 1

j = 1

k = 0

k = 1

i = 1

j = 0

k = 0

k = 1

j = 1

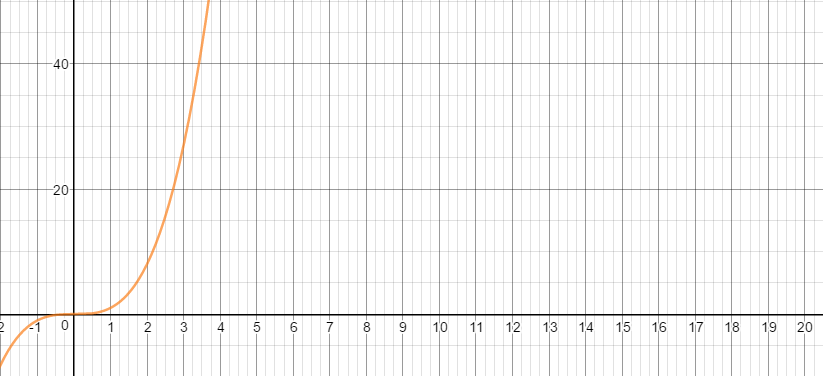
k = 0

k = 1

8

For the previous example it required iterations,

Graphing the cubic function looks like this:



**The Exponential Function:**

A function is said to be exponential if where is a positive constant, called **base** and is the **exponent**, as the logarithmic function the most common base to use is 2, so most of the time the base for the exponential function is base 2.

A program or function that has exponential running time is a bad news because such a program run extremely slow.

Example:

Suppose the running time of a function is and each operation can be executed in one micro second the solving the problem for = 100 will take this is more than 10,000 trillion year.

