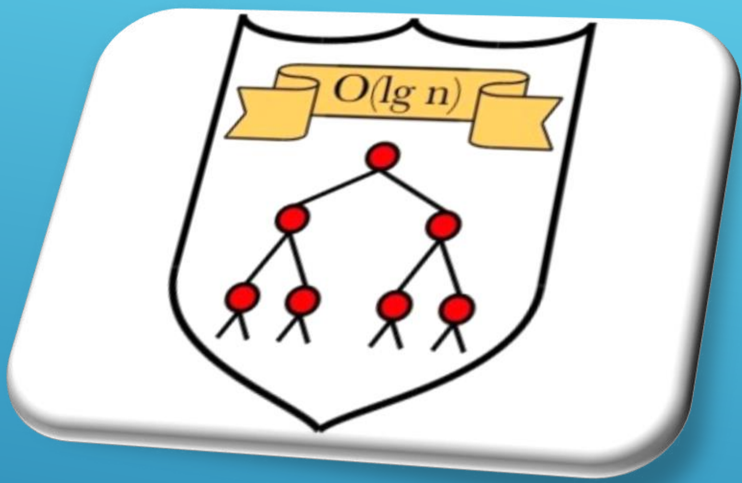


Algorithm Efficiency

Big O

To design and implement algorithms, programmers must have a basic understanding of what constitutes good, efficient algorithms.

Linear Loops
Logarithmic Loops
Nested Loops
Big-O Notation
Standard Measurement of Efficiency

Algorithm efficiency is generally defined as a function of the numbers to be processed.

$$f(n) = \frac{n(n+1)}{2}$$

The simplification of efficiency is known as

big-O notation.

$$O(n^2)$$

$$f(n) = \frac{n(n+1)}{2}$$

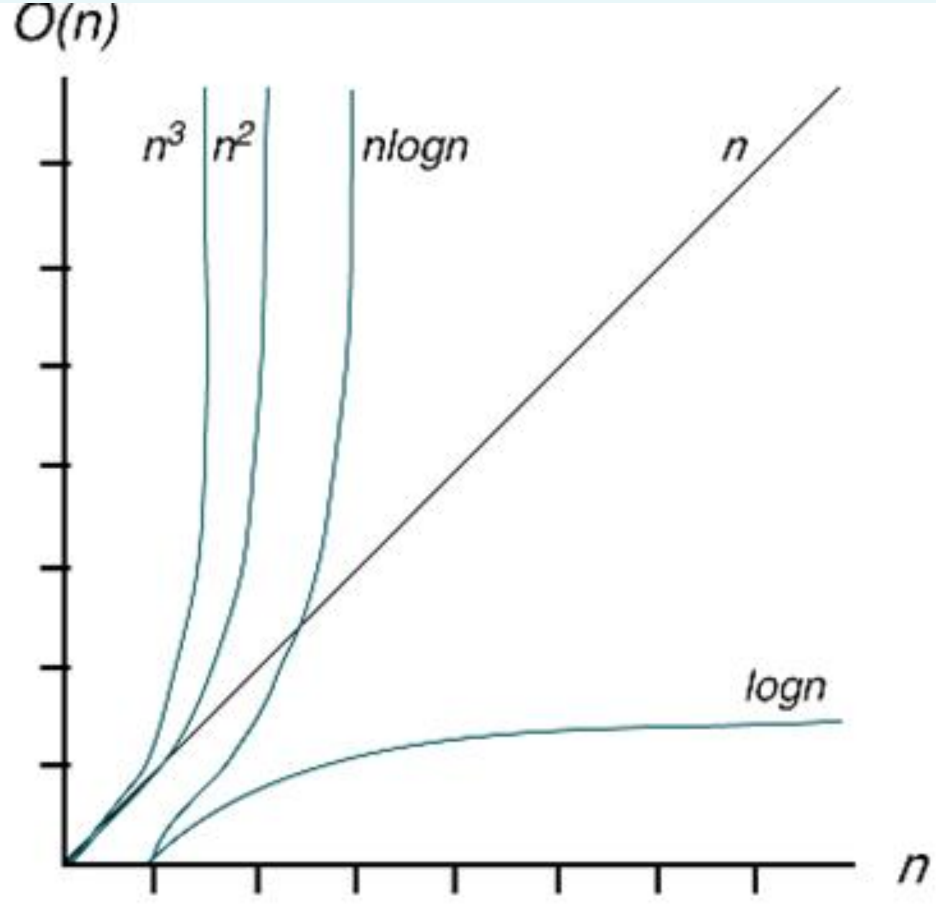
- **As *n* grows large, the n^2 term will come to dominate, so that all other terms can be neglected.**
- **The constants will depend on the precise details of the implementation and the hardware it runs on, so they should also be neglected.**

Big O notation

Captures what remains: O(n^2) says that the algorithm has *order of* n^2 time complexity.

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

Measures of Efficiency for $n = 10{,}000$



Plot of Effeciency Measures

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

. . .

$2^n = x$

$2^0 = 1$ $\log_2 1 = 0$

$2^1 = 2$ $\log_2 2 = 1$

$2^2 = 4$ $\log_2 4 = 2$

$2^3 = 8$ $\log_2 8 = 3$

$2^4 = 16$ $\log_2 16 = 4$

. . .

$2^n = x$ $\log_2 x = n$

$2^0 = 1$ $\quad$ $\log_2 1 = 0$

$2^1 = 2$ $\quad$ $\log_2 2 = 1$

$2^2 = 4$ $\quad$ $\log_2 4 = 2$

$2^3 = 8$ $\quad$ $\log_2 8 = 3$

$2^4 = 16$ $\quad$ $\log_2 16 = 4$

. . .

$2^n = x$ $\quad$ $\log_2 x = n$

$\log_2 40 = ?$

$2^0 = 1$ $\qquad\qquad$ $\log_2 1 = 0$

$2^1 = 2$ $\qquad\qquad$ $\log_2 2 = 1$

$2^2 = 4$ $\qquad\qquad$ $\log_2 4 = 2$

$2^3 = 8$ $\qquad\qquad$ $\log_2 8 = 3$

$2^4 = 16$ $\qquad\qquad$ $\log_2 16 = 4$

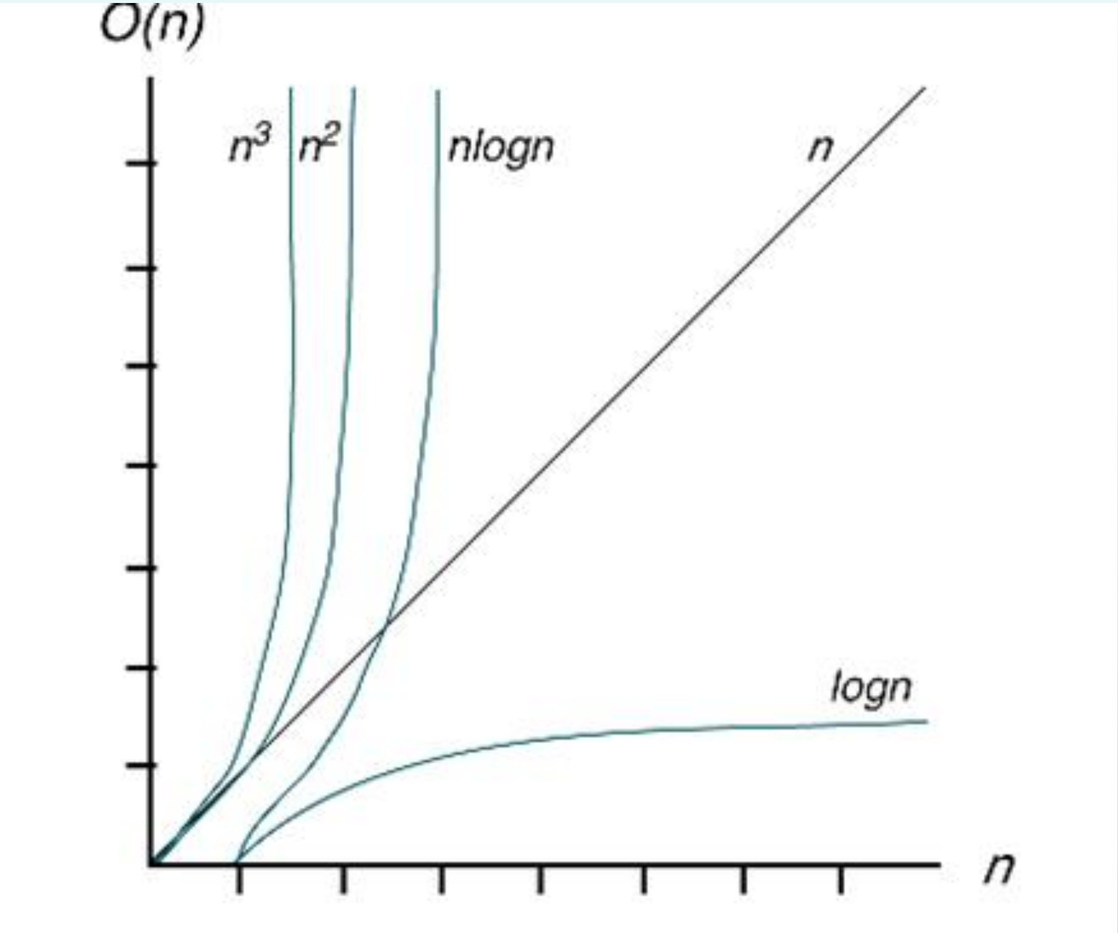
. . .

$2^n = x$ $\qquad\qquad$ $\log_2 x = n$

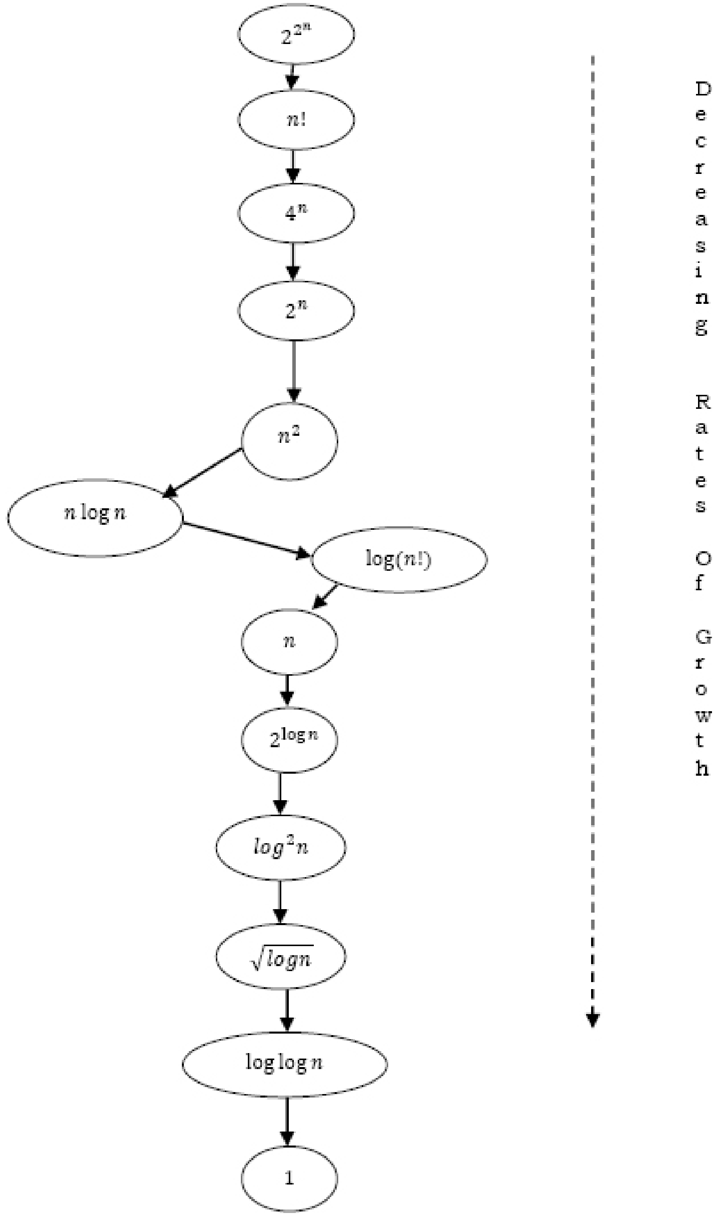
$$5 = \log_2 32 < \log_2 40 < \log_2 64 = 6$$

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

Measures of Efficiency for $n = 10{,}000$



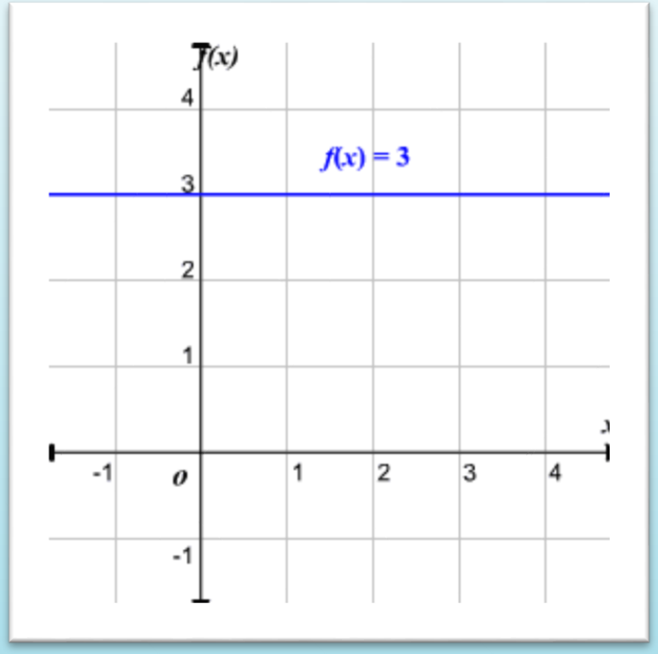
Plot of Effeciency Measures



2^{2^n}
$n!$
4^n
2^n
n^2
$n \log n$
$\log(n!)$
n
$2^{\log n}$
$log^2 n$
$\sqrt{log n}$
$\log \log n$
1
Decreasing Rates Of Growth

Constant Function

f(n) = c



where c is a fixed constant such as

c = 5

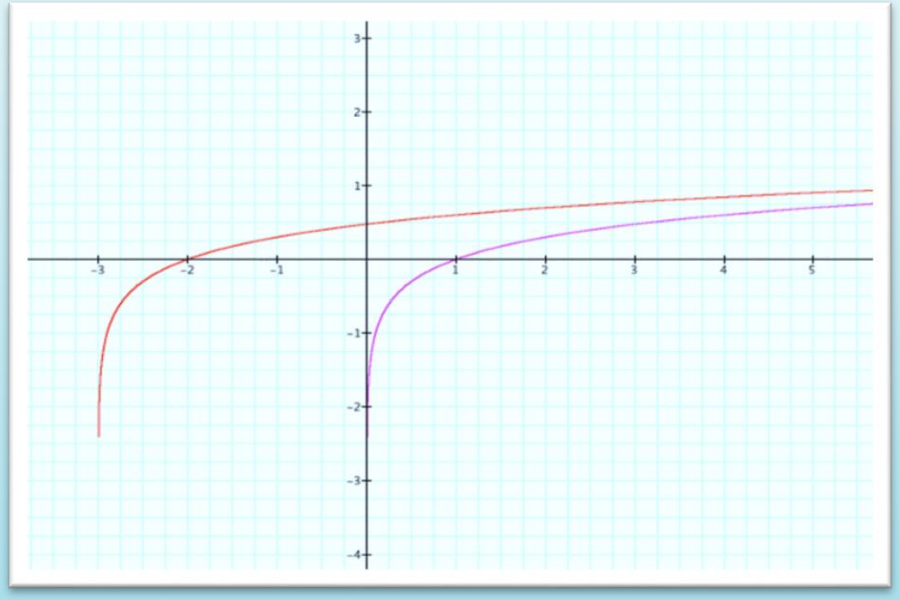
c = 1

c = 510

The variable n is the size of the data that needs to be evaluated.

Logarithmic Function

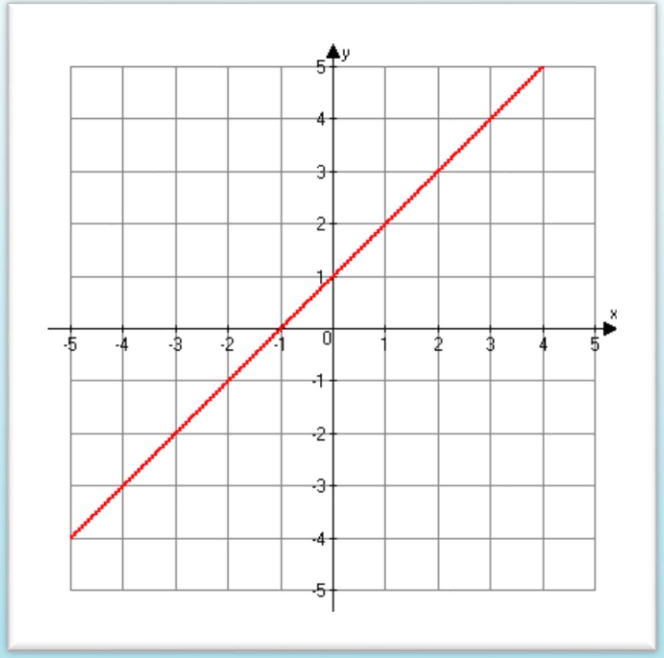
f(n) = log [***base***]n



where [base] is the base of the logarithm. In computer science, log2n is used that the 2 is often left off and log2n is written as log n.

Linear Function

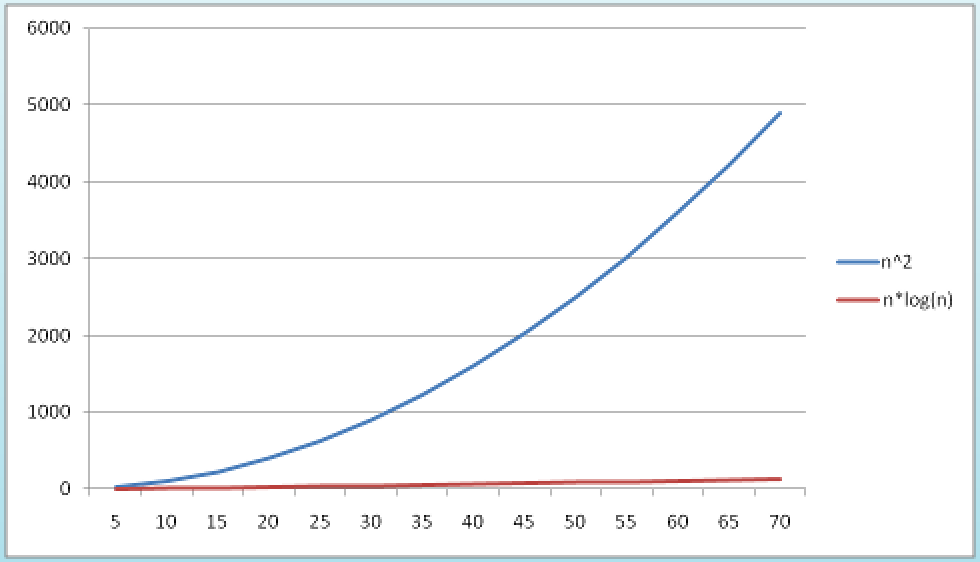
f(n) = n



The output of this linear function example is the value of n itself. A function that includes a constant, such as **f(n) = 2n** or **f(n) = n + 2**, also is a linear function.

N-Log-N Function

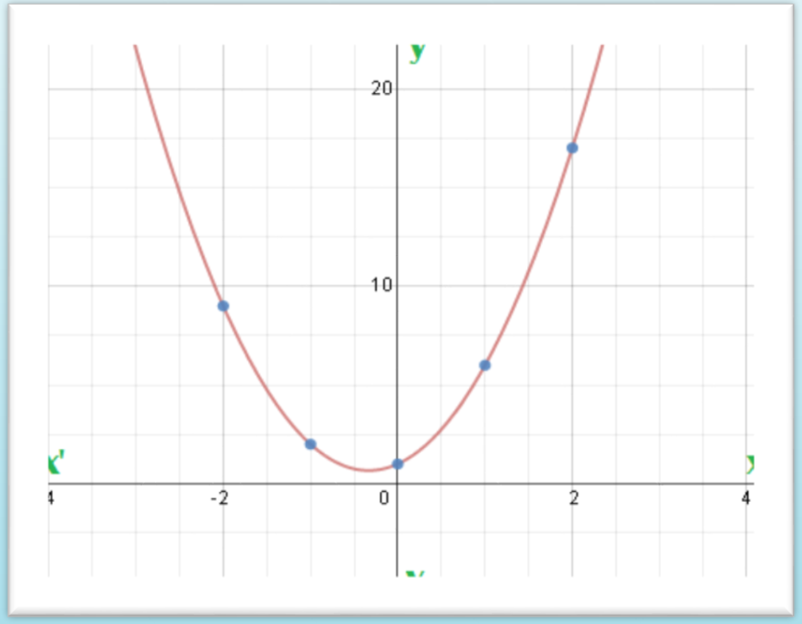
f(n) = n log(n)



In an **n-log-n** function, the log(n) calculation is repeated n times.

Quadratic Function

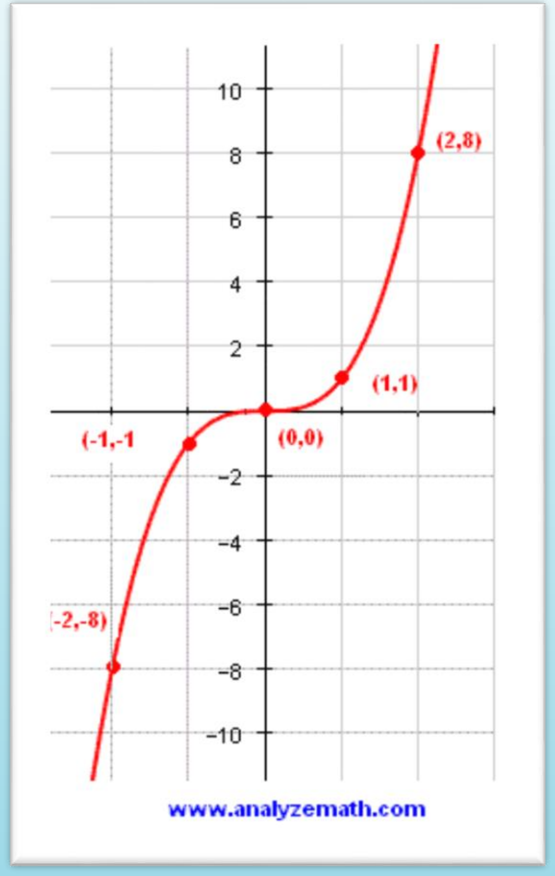
$f(n) = n^2$



Example: Algorithms with two **for** loops, where one for loop nested in the other **for** loop

Cubic Function

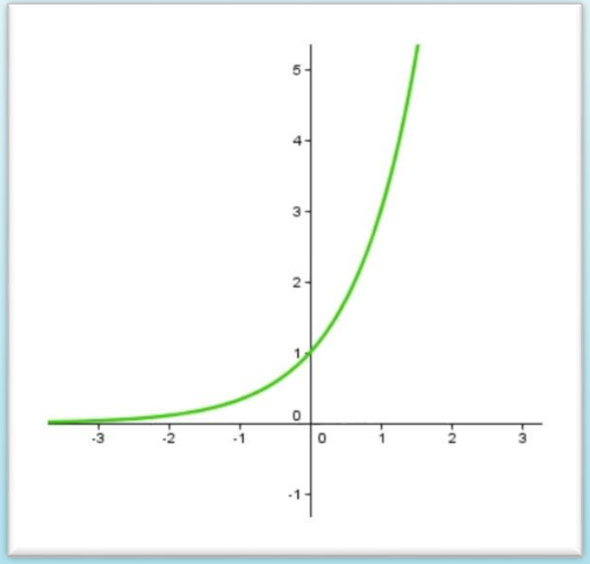
$f(n) = n^3$



Example: Algorithm with 3 nested for loops

Exponential Function

$f(n) = b^n$



where b is a positive constant called the base. In computer science, the most common base is 2, which means that an algorithm can be described on the order of $f(n) = 2^n$

Example: Given a binary string of 8 bits, there will be 2^8 combinations (256) of those bits.

Polynomial Example

T(n) = $5n^3 + 3n^2 + \mathbf{n} + 5$.

The term $5n^3$ has the largest growth rate and will dominate the other terms as **n** grows sufficiently large.

The $3n^2$ and **n** terms can be discarded.

$5n^3$ has a constant of 5 that can be omitted, leaving n^3.

T(n) = $5n^3 + 3n^2 + \mathbf{n} + 5 = O(n^3)$.

Give the BigO notation for each of the following pseudocode fragments:

```
(A)   a = n
      b = n + 1
```

Give the BigO notation for each of the following pseudocode fragments:

```
(B)   a = n
      if( a >= 0 )
          b = n + 1
      else
          b = (-1)*n + 1
      end if
```

Give the BigO notation for each of the following pseudocode fragments:

```
(C)  i = 1
     loop( i <= n )
        print( i )
        i = i + 1
     end loop
```

Give the BigO notation for each of the following pseudocode fragments:

```
(D)  i = n
     loop( i > 0 )
        print( i )
        i = i - 1
     end loop
     j = 1
     loop( j <= n )
        print( j )
        j = j + 2
     end loop
```

Calculate the run-time complexity of the following algorithm segment:

```
i = 1
loop( i <= n )
  j = 1
  loop( j <= n )
     k = 1
     loop( k <= n )
        print( i, j, k )
        k = k + 1
     end loop
     j = j + 1
  end loop
  i = i + 1
end loop
```

```
(E)  i = 1
     loop( i <= n )
         print( i )
         i = i * 2
     end loop
```

```
(F)  i = n
     loop( i > 0 )
         print( i )
         i = i / 2
     end loop
     j = 1
     loop( j <= n )
         print( j )
         j = j + 2
     end loop
```

```
Node lookup(node, n)
      for Node p = node; p != null; p = p.next
            if( p.item = = n )
                  return p
      return null
```

```
func bubblesort(array )
      for i from 2 to N
            swaps = 0
            for j from 0 to N - 2
                  if a[j] > a[j + 1]
                        swap( a[j], a[j + 1] )
                        swaps = swaps + 1
                  if swaps = 0
                        break
end func
```