Introduction to algorithms

Problem Solving ------> Computer Programs -----> Algorithms

There can be many solutions or algorithms for one problem.

P1 ------> {A1, A2, A3, … }

Before implementing any algorithm as the program, we have to find out which algorithm among these is good in terms of performance measures time and memory.

Analyse an algorithm in terms of performance measures.

Time : Which algorithm would execute faster
Memory : Which algorithm will take less memory

First we design all possible algorithms which can use, then analyse them to choose the best algorithm, then the best algorithm will be implemented as our program, or the solution.

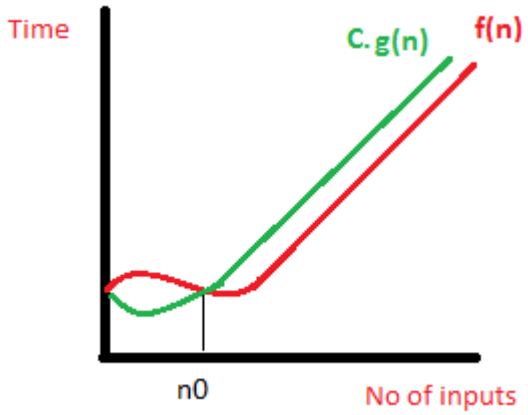
Asymptotic Notations - notations used in algorithm

Big O notations

To measure how well computer algorithms scales as the amount of data involved increases.

As no of input giving to the algorithm is increasing, the time is increasing according to the following graph.

The function of the curve : f(n) where n is no of inputs
Worst case or the upper bound for this function?
Another function C.g(n) greater than the given f(n)
After some limit n0, if C.g(n) can bound the function f(n), the value of C.g(n) is greater than f(n).



f(n)<=C.g(n) | after some n>n0 ; C>0 and n0>=1
If you can satisfy the above condition; we can say,
f(n) is big O of g(n) ----------> f(n) = O (g(n)) which means that f(n) is smaller than g(n)

Example 01

f(n) = 3n+2 and g(n) = n
Can we say, f(n)=O(g(n)) is true?
Find C and n0 first.
f(n)<=C.g(n) ; for some C>0 and n>=n_0
3n+2 <= C.n
When can 3n+2 be less than or equal to C.n ?
Can have any value for C, the constant.
The best option is C=4, but anything above 3 is good.
So, 3n+2<=4n ; so we found C. What about n?
3n+2<=4n, from this
n>=2
So, every n>=2 and C=4; f(n)<=C.g(n)
Therefore; f(n)=O(g(n)) ; f(n) is big O of g(n)
O(3n+2) = n

We found out that, if f(n) = 3n+2 and g(n)
Then f(n) is big O of g(n) ; f(n)=O(g(n))
That means, 3n+2 can be written as a constant multiply by n.

Let f(n) and g(n) be functions matching non-negative integers just to real numbers. We say that f(n) is big O of g(n). If there is a real constant C>0 and an integer constant n0>=1 such that,

f(n) <= C.g(n) for n>=n0

Example 02

f(n)=8n-2
Linear constant

$f(n) <= C.g(n)$ for $n>=n_0$

$g(n)=n$
$8n-2 <= C.n$
$8n-2 <= 8n$
So, $C=8$

$f(n)<=8n$ for $n>=1$
$O(8n - 2) = n$

Example 03

$f(n)=8n^2 + 3n + \log n$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = n^2$
$8n^2 + 3n + \log n <= C.n^2$
$8n^2 + 3n + \log n <= 8.n^2$
So, $C=8$

$f(n) <= 8.n^2$ for $n>=8$
$O(8n^2 + 3n + \log n) = n^2$

Example 04

$f(n) = 3n^5 + n\log n + 3$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = n^5$

$3n^5 + nlogn + 3 <= C.n^5$
$3n^5 + nlogn + 3 <= 3.n^5$
So, C=3

$f(n) <= 3.n^5$ for n>=8
$O(3n^5 + nlogn + 3) = n^5$

Example 05

f(n) = 3nlogn + 3n
$f(n) <= C.g(n)$ for $n>=n_0$

g(n) = nlogn
3nlogn + 3n <= C.nlogn
3nlogn + 3n <= 3.nlogn
So, C=3

f(n) <= 3nlogn for n>=3
O(3nlogn + 3n) = nlogn

Example 06

$f(n) = 2^n + 3$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = 2^n$
$2^n + 3 <= C.2^n$
$2^n + 3 <= 1*2^n$

So, C =1

$f(n) <= 1*2^n$ for n>=1
$O(2^n + 3) = 2^n$

Example 07

$f(n) = 2^{n+3} + n^2$
$f(n) = 2^3.2^n + n^2$
$f(n) = 8.2^n + n^2$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = 2^n$
$8.2^n + n^2 <= C. 2^n$
$8.2^n + n^2 <= 8* 2^n$
So, C=8

$f(n) <= 8* 2^n$ for n>=8
$O(2^{n+3} + n^2) = 2^n$

Example 08

$f(n)= 3n + \log n + 7$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = n$
$3n + \log n + 7 <= C.n$
$3n + \log n + 7 <= 3.n$
So, C = 3

$f(n) <= 3.n$ for $n>=3$
$O(3n + \log n + 7) = n$

Example 09

$f(n) = n^3 + n^2 + 100n$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = n^3$
$n^3 + n^2 + 100n <= C.n^3$
$n^3 + n^2 + 100n <= 1.n^3$
So, C=1

$f(n) <= 1.n^3$ for $n>=1$
$O(n^3 + n^2 + 100n) = n^3$

Example 10

$f(n) = 2^n + n^3$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = 2^n$
$2^n + n^3 <= C.2^n$
$2^n + n^3 <= 1.2^n$
So, C=1

$f(n) <= 1.2^n$ for $n>=1$
$O(2^n + n^3) = 2^n$

Example 11

f(n) = 3n - 2
f(n) <= C.g(n) for n>=n_0

g(n) = n
3n - 2 <= C.n
3n - 2 <= 3.n
So, C=3

f(n) <= 3.n for n>=3
O(3n - 2) = n

Example 12

f(n) = nlogn -n
f(n) <= C.g(n) for n>=n_0

g(n) = nlogn
nlogn - n <= C.nlogn
nlogn - n <= 1.nlogn
So, C=1

f(n) <= 1.nlogn for n>=1
O(nlogn -n) = nlogn

Example 13

$f(n) = 3n^3 + 10n^2 - 21n\log n$
$f(n) <= C.g(n)$ for $n>=n_0$

$g(n) = n^3$
$3n^3 + 10n^2 - 21n\log n <= C.n^3$
$3n^3 + 10n^2 - 21n\log n <= 3.n^3$
So, C=3

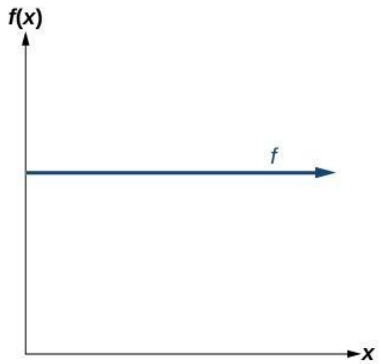
$f(n) <= 3n^3$ for $n>=3$
$O(3n^3 + 10n^2 - 21n\log n) = n^3$

Note:
$2^n > n^5 > n^4 > n^3 > n^2 > n\log n > n > \log n$

Different Functions

1. **Constant Function**

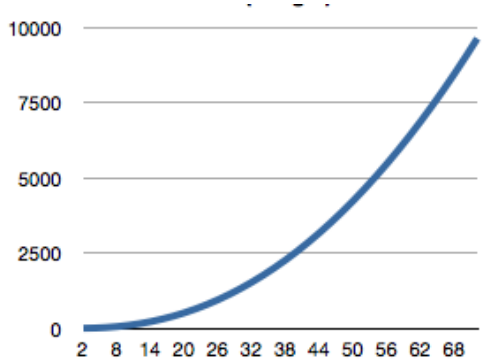
f(n) = C



One of the application in constant function is the time required to execute the primitive operations will be the constant time.
Example : comparing a 8 bit register with another 8 bit register. (It will take same time for each bit)

2. **Logarithmic Function (Log function)**

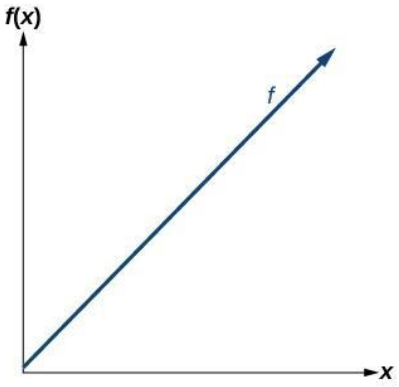
f(n) = logn ; O(f(n))= logn



Even for a very large input, the output will be coming in a very small time.

3. **Linear Function**

f(n) = n ; O(f(n))= n

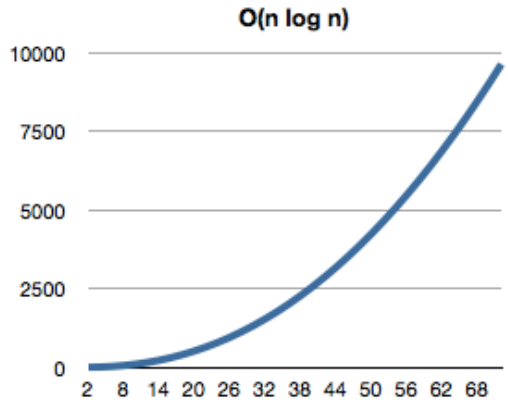


A linear function will arise in algorithm analysis, when we have to do a single basis operation for each of n elements.

Example : Comparing a number x to each element of an array of size n will require n comparisons.

4. **nlogn Function**

f(n) = nlogn ; O(f(n))= nlogn

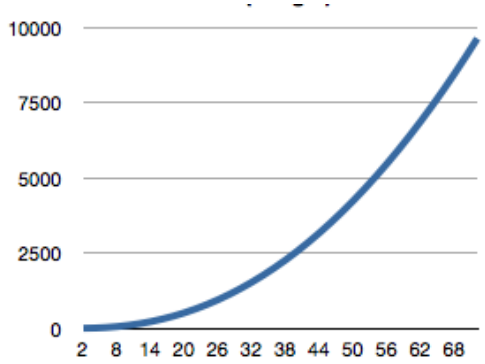


The function goes a little faster than the Linear function and a lot slower than the Quadratic function. ($f(n) = n^2$)

n	logn	nlogn
10	1	10
100	2	200
1000	3	3000
10000	4	40000

5. **Quadratic Function**

$f(n) = n^2$; $O(f(n))= n^2$

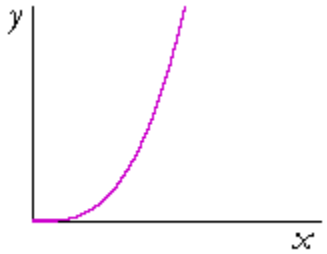


The quadratic function appears in many algorithms that have “nested loops”, where the “inner loop” perform a linear no of operations and the “outer loop” performs another linear no of operations.
In such cases, the algorithm will perform n*n = n^2 no of operations.
Example:

```
#include<stdio.h>
int main()
{
    for(int i=0; i<5; i++)
    {
        for(int j=0; j<=i; j++)
        {
            printf("* ");
        }
        printf("\n");
    }
}
```


6. Cubic Function (and other polynomials)

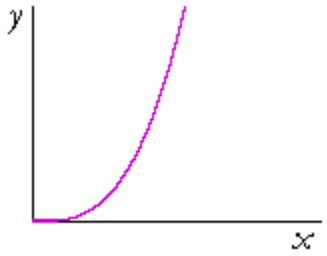
$f(n) = n^3$; $O(f(n)) = n^3$



Polynomial :
$f(n) = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n$
Here, $a_1, a_2, a_3, \ldots, a_n$ are constants

7. Exponential Function

$f(n) = a^n$; $O(f(n))= a^n$

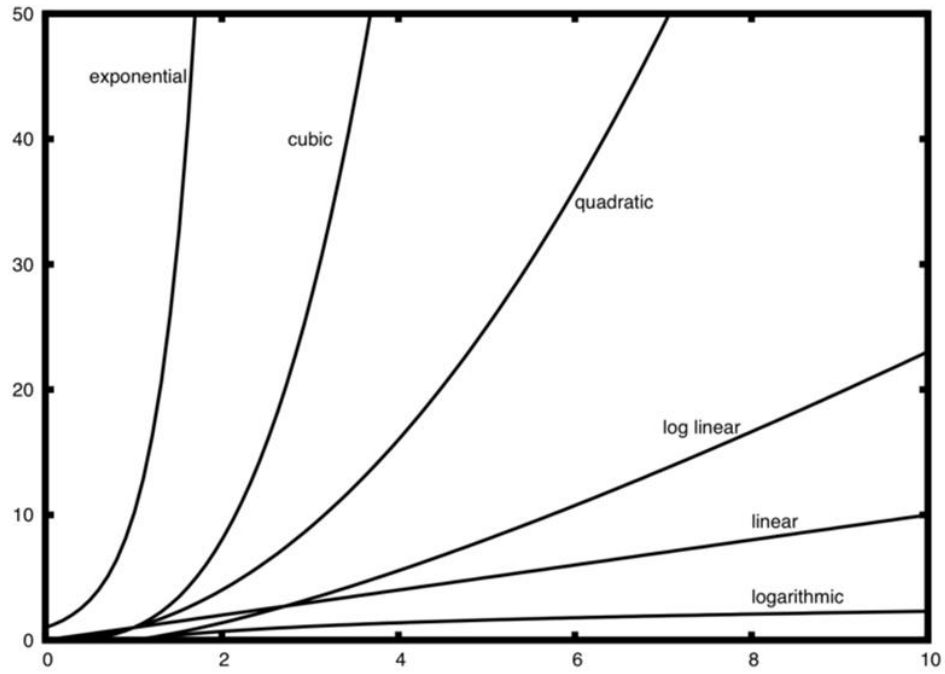


In algorithm analysis, he most common based for exponential function is a = 2 , that is $f(n) = 2^n$.

Example : if we have a loop that starts by performing one operation and then doubles no of operations performed with each iteration. Then the total no of operations performed in the n^{th} iteration is 2^n.

No of iteration	1	2	3	4	...	n
Total operations	2	4	8	16	...	2^n

Time vs Problem Size (n - no of inputs) for different functions (different types of algorithms)



Big O of some algorithms
(Time Complexity)

Algorithm	**Big O**
Bubble Sort	n^2
Insertion Sort	n^2
Selection Sort	n^2
Quick Sort	n^2
Merge Sort	nlogn
Search (array, queue, stack, binary search tree, singly linked list, doubly linked list)	n
Search (Red black tree)	logn

Q1) Determine the Big-O values of the following:

1. Find an item in an arbitrary array of size n. ----------> n

2. Finding the maximum from an array of values. ----------> n
Think about it mathematically. Unless the array is sorted, there is nothing to "cut in half" to give you the `log(n)` behavior.

3. Finding the sum of an array of items.

```
temp_sum = 0
for i in 1 ...array.length
temp_sum = temp_sum + array[i]
```

4. Adding a value to an unsorted array.

5. Adding values to a sorted array.

	Insert	Delete	Search	Space Usage
Unsorted array	O(1)	O(1)	O(n)	O(n)
Value-indexed array	O(1)	O(1)	O(1)	O(n)
Sorted array	O(n)	O(n)	O(log n)	O(n)
Unsorted linked list	O(1)*	O(1)*	O(n)	O(n)
Sorted linked list	O(n)*	O(1)*	O(n)	O(n)
Balanced binary tree	O(log n)	O(log n)	O(log n)	O(n)
Heap	O(log n)	O(log n)**	O(n)	O(n)
Hash table	O(1)	O(1)	O(1)	O(n)

 * The cost to add or delete an element into a known location in the list (i.e. if y
** The deletion cost is O(log n) for the minimum or maximum, O(n) for an arbitrary e

6. Input n no. of values to an array. ----------> n

7. Finding a value in an array of size n using Binary search. ----------> n