An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi, is defined as follows: Roughly speaking:

* An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
* An algorithm is a finite step-by-step procedure to achieve a required result.
* An algorithm is a sequence of computational steps that transform the input into the output.
* An algorithm is a sequence of operations performed on data that have to be organized in data structures.
* An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is the Euclid's algorithm for calculating the greatest common divisor of two integers. This theorem appeared as the solution to the Proposition II in the Book VII of Euclid's "Elements." Euclid's "Elements" consists of thirteen books, which contain a total number of 465 propositions.

Algorithmic is a branch of computer science that consists of **designing** and a**nalysing** computer algorithms.

1. The design pertains to

(a) Descript ion of algorithm at an abstract level by means of psuedo code languages and

(b) Proof of correctness i.e, the algorithm solves a given problem in all cases.

1. The “analysis” deals with performance evaluation( complexity analysis)

Analysis of Algorithm:

1. Theorital study of computer program targets for (a) performance and (b) resource usage ( ram and disk memory) of the program behaviour. How do we make the performance fast?
2. What is more important than performance?

Correctness, Simplicity, maintainability, cost ( program time – billing :) ) , stability, robustness of the software, modularity, security, scalibality, and userfriendlyness...

1. Why do we study algorithms and performances?
2. Performance is co-related with user-friendlyness
3. There are real time contraints that we need to compromise with performances..
4. Performance measures the line between the feasible and in-feasible
5. Real time scinerio ( not fast enough, uses too much of resources ( memory), thus time consuming and where things donot scale.)
6. That's why Algorithm are there in the cutting edge of enterprenuership.
7. Algorithm gives us a language of talking about programming behaviour.
8. Anything else?
9. Performance is like money. What good does a good stack of $100 bills do for you? Wouldn't you rather have water, food, shelter that we are willing to pay those $100 bills.
10. Performance is that we pay for user-friendlyness and security.
11. People would say program in “Java” rather in “C” , as it has more functionality( object oriented feature and has expection handling mechanism). So smart invester lands up paying a factor of 3 times because of performance and speed.

Problem: Sorting ( Many algorithm technique )

Input : Sequence < a1, a2, ..... , an > of numbers in some order

Output : In order sequence of < a1', a2' ...... an' > such that ( $ )

$ a1' <= a2' <= ...... <= an' ( monotonically increasing in size)

Use an algorithm to do this output sequence : This can be done by using INSERTION SORT.

Insertion Sort(A, n) : Wrtite a psuedo code that sorts an array A with size (1,....,n)

Psuedo code:

FOR ( j ← 2 ) TO length[A] or “n” // We are running an outer loop ( j from 2 to n )   
 DO key ← A[j]  
 {Put A[j] into the sorted sequence A[1 . . [j − 1]}  
 i← ( j − 1) // the inner loop starts at ( j - 1)   
 WHILE ( i > 0 and A[i] > key  
 DO A[i + 1] ← A[i]   
 i ← (i − 1) // the inner loop goes down as ( i < - ( 1 – 1) ) until it goes down to “0” while ( i > 0)  
 A[i + 1] ← key // This psuedo code has induction techniques followed

Explaination Notes :

1. There is an Array which is partitioned at jth position. We introduce a key and start running an outer loop ( j < - 2 to n ) and
2. There are inner loop start at from i at ( j – 1) and then goes down to ( i < - ( i -1 ) until it goes to '0' ( while i > 0)
3. The elements below jth position are sorted using the key value , we pull the value out here and then we call the key which we can insert in the same position
4. There is an invariant which is being maintained by this while loop each time its through.
5. The invariant is that each time the element before jth element are sorted.
6. The goal for each time through the loop is to increase ( add 1) to the length of things that are being sorted.
7. We pull out the key and copy the values, keep copying up, until we find the place where the key goes and inserted the key into that position.
8. Like we have just sorted from 1 to j and next we can start sorting ( j + 1 )

Note:

**Algorithm's Performance**

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation.

The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

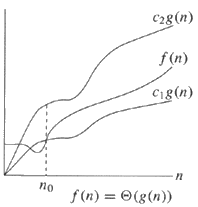
The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

**Θ-Notation (Same order)**

This notation bounds a function to within constant factors. We say f(n) = Θ(g(n)) if there exist positive constants n0, c1 and c2 such that to the right of n0 the value of f(n) always lies between c1 g(n) and c2\ g(n) inclusive.  
  
In the set notation, we write as follows:

Θ(g(n)) = {f(n) : there exist positive constants c1, c1, and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n) for all n ≥ n0}

We say that is g(n) an asymptotically tight bound for f(n).



Graphically, for all values of n to the right of n0, the value of f(n) lies at or above c1 g(n) and at or below c2 g(n). In other words, for all n ≥ n0, the function f(n) is equal to g(n) to within a constant factor. We say that g(n) is an asymptotically tight bound for f(n).

In the set terminology, f(n) is said to be a member of the set Θ(g(n)) of functions. In other words, because O(g(n)) is a set, we could write

f(n) ∈ Θ(g(n))

to indicate that f(n) is a member of Θ(g(n)). Instead, we write

f(n) = Θ(g(n))

to express the same notation.

Historically, this notation is "f(n) = Θ(g(n))" although the idea that f(n) is equal to something called Θ(g(n)) is misleading.

Example: n2/2 − 2n = (n2), with c1 = 1/4, c2 = 1/2, and n0 = 8.

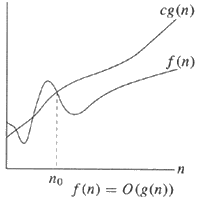
Ο-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write f(n) = O(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or below c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

Ο(g(n)) = {f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ c g(n) for all n ≥ n0}

We say that the function g(n) is an asymptotic upper bound for the function f(n). We use Ο-notation to give an upper bound on a function, to within a constant factor.



Graphically, for all values of n to the right of n0, the value of the function f(n) is on or below g(n). We write f(n) = O(g(n)) to indicate that a function f(n) is a member of the set Ο(g(n)) i.e.

f(n) ∈ Ο(g(n))

Note that f(n) = Θ(g(n)) implies f(n) = Ο(g(n)), since Θ-notation is a stronger notation than Ο-notation.

Example: 2n2 = Ο(n3), with c = 1 and n0 = 2.

Equivalently, we may also define f is of order g as follows:

If f(n) and g(n) are functions defined on the positive integers, then f(n) is Ο(g(n)) if and only if there is a c > 0 and an n0 > 0 such that

f(n) | ≤ g(n) | for all n ≥ n0

*Historical Note: The notation was introduced in 1892 by the German mathematician Paul Bachman.*

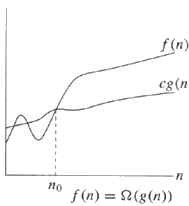
Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write f(n) = Ω(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or above c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

Ω(g(n)) = {f(n) : there exist positive constants c and n0 such that 0 ≤ c g(n) ≤ f(n) for all n ≥ n0}

We say that the function g(n) is an asymptotic lower bound for the function f(n).



The intuition behind Ω-notation is shown above.

Example: √n = (lg n), with c = 1 and n0 = 16.

**Algorithm Analysis**:

1. Running time depends on whether Input order ( eg: if already sorted, insertion sort has less work or things to do )
2. Runing time depends on Input size . How big is the array? ( eg: 6 elements v/s 6 \* elements )

How to handle this?

Parameterize in input size as a function of the size we are sorting.

1. Wants upper bounds

Program that runs in 3 sec. ( Real time settings)

Programs that takes at least 3 secs ( grantueed to the user)

1. Normally , kind of analysis that we do for any algorithm :
2. Worst case analysis (usually)

max time on any input size n.

1. Average case analysis

expected time over all input of size n. This is the probability of every input occurence in a given situation. So we need assumption of statistical distribution.

1. Best case analysis ( Bogus)

We do some cheating based on some known input.

1. Why is INSERTION SORT worst case time?

It depends on what computer with different computational ability we use – relative speed ( on some machine), absolute speed ( on different machine)

1. BIG IDEA of algorithms are done using ASYMTOTIC analysis.

Asymtotic analysis is to :

1. Ignore machine dependent constants
2. Look at the growth of as

ASYMTOTIC NOTATION:

1. Asymtotic notation satisfies our issues of being able to compare both relative and absolute speed.
2. Drop low order terms and ignore leading constants.

Example:

As , algorithm always beats algorithm.

Here, will be sufficiently faster than . It doesn't matter what the low order terms are, and also does not matter what the leading constants were,

even though we run algo on slower computer and algo on a fast computer.

Engineering Notes:

SO, ASYMTOTIC NOTATION satisfies our issues of being able to compare both relative and absolute speed. As we go more larger ,is going to be cheaper than algorithm, no matter

how much of advantages we give it in the begining in terms of the speed of the computer that it is running on.

ENGINEERING POINT OF VIEW is that some issues we have to deal with because sometimes that ( common point) is so large that the computer aren't big enough to run the problem faster and

that's why we are neverthless interested in slower algorithm even though they are not asymtotically slower, they may be still faster on reasonable size of things.

So, we have to both balance our “Mathematical understanding” with our “Engineering common sense” in order to do good programming. So , just having done analysis and algorithm doesn't automatically

make you a good programmer. We also need to know atleast how to use these tools in practice to understand when they are revelent and non-relevant.

To be a world class programmer, you can program everyday for 2 years and take algorithm classes or program everyday for 10 years.

1. Identifying in terms of counting operations is counting memory references.
2. How many times do you access same variables?
3. “j” is going down from 2 to n and
4. then we are going to add up the work done within the do loop.
5. Work done in the loop ( work varies, but in the worst case, how many operations is going on here for each value of “j” ?)

INSERTION SORT ANALYSIS:

1. Worst case : Input reverse sort. This will take the longest possible time.

// ARITHMETRIC SERIES

For a given value of j, how much of work done in terms of whole while loop asymptotically? Is this insertion sort fast enough?

1. Moderately fast for smaller size of “n”
2. Not at all fast for smaller “n”.

This gives us a need to improvise on insertion sort and work out all for MERGE SORT.

MERGE SORT: ( RECURSION PROGRAM): Array A [ 1, ...., n ]

1. CONSTANT : if n = 1, done .

How long does this step (1) take?

This is constant time . It just check and return which is independent of the size of what we are doing?

CONSTANT

1. RECURSIVE SORT : Divide them into 2 equal lists as below:

A [ 2 ..... n/2 ] and A [ (n/2 + 1) .... n]

How long does this type of step take?

And

Here time taken is ok to be sloppy as long as linear regression are precise, we can be sloppy.

Sloppy though...

1. MERGE this 2 sorted lists

Hence this 2 sorted list will have a total of “n” elements.

MERGE SORT:

Key sub-routing here:

L1 L2 LIST

1. Observe where is the smallest element of any of the 2 list that are bieng sorted.
2. So, One has to place this ahead of the 1st or the 2nd lists.

So, Total linear time ( as it is not quadratic) on “n” total elements

This allows us to write a recurrence for the performance of merge sort.

**RECURRENCE:**

Running time of Merge Sort:

T(n) = Θ(1), if n = 1,

T(n) = 2T( n 2 ) + Θ(n) otherwise when n > 1



Rewriting the recurence Recursion tree:

T(n) = 2T( n 2 ) + c \* n , constant c > 0

*T*(*n*) *=* ***c***if *n =* 1

*T*(*n*) *=* 2*T*(*n*/2) *+* ***cn***if *n >* 1

***c* > 0**: Running time for the base case and time per array element for the

divide and combine steps.

For the original problem, we have a cost of *cn* , plus two subproblems each of size (*n* /2) and running time *T* (*n* /2).

Each of the size *n* /2 problems has a cost of *cn* /2 plus two subproblems, each costing *T* (*n* /4).

Cost of divide and merge ( total = cn , incremental steps = cn/2 )

Cost of sorting sub-problem ( each subproblem = cn/2, incremental steps = T(n/2), T(n/4))

Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1

***cn***

***cn***

***cn***

***cn***

**lg *n***

Total = cn(lg n ) + cn

Here,

height of the tree = (lg n)

number of leaves = n ( if we go till height h = lg n , we have 2^h leaves, hence 2^ lg n = n

Total = cn lg n + Θ(n) = Θ( n lg n)

Now , Θ( n lg n) is asymtotically faster than Θ( n^2)

Merge sort (Θ( n lg n)) on a large enough input size ( huge array ) is going to be faster and asymtotically

beats insertion sort .( Θ( n^2))

We can run insertion sort on a super computer or some one running on a PC with merge sort for

sufficiently large input will clobber up. This is because Θ( n^2) is a way bigger than Θ( n lg n) when n is

large.

In practice, merge sort is tends to be the winner for cases n > 30 or so.

If n < 30, insertion sort is fine to choose but merge sort is going to be more faster even for a few dozen

of elements.