22ND MAY, 2019
COURSE CODE: CSC 213
COURSE TITLE: ALGORITHM AND ITS APPLICATION COMPLEXITY
DESIGNING AND ANALYSIS OF AN ALGORITHM

Algorithm is a sequence of getting some inputs to get an output (result)

A well-defined procedure in which some inputs are transformed into outputs. (Another definition)

Note that for an algorithm to exist, there must be a well-defined procedure in order to get a desired/pre-determined output/result.

An algorithm can be implemented in several ways. An algorithm cannot be implemented if there is no design of framework.

*Improving time and reducing cost

When is an algorithm effective?

It is effective, if the input(s) brings out a useable output (saves time and cost)

In designing an algorithm, the content must be considered.

Analysis can be abstract or mathematical.

The abstract and mathematical comparism of an algorithm is called an analysis.

Mathematics is needed for algorithm;

(1) It is the best way to do formal specification of a problem

(2) It is used in the analysis of correctness.

(3) Mathematics is required in carrying out the analysis of efficiency (i.e how much time used in the analysis of the algorithm, also the inputs)

In analysis, the interest is to predict and analyze the resources that are used for the algorithm. Many resources can be used but running time is most concerned

To analyse R.T of an algorithm mathematical models of a computer is needed

There are several models involved in determining R.T of an algorithm.

Running Time is basically the number of RAM instructions that is executed. RAM has a model

NEXT WEEK

10TH JULY 2019 $f(n) = O(g(n))$ (Running time; order of growth)

GROWTH OF FUNCTIONS

When we study algorithm, we are interested in their efficiency and how to characterize them according to their efficiency. We are actually concerned about the order of growth for the running time of the given algorithm. This is also referred to the asymptotic running time. We need to discover that running time is

Order of growth is called asymptotic

Asymptotic notation gives us a method for classifying algorithm according to their order growth.

If $f(n)$ is non-negative, we can simply find the condition that O can still be re-expressed

$$0 \leq f(n) \leq c(g(n))$$

(For all values of N [$\forall N \exists$ no exist]

From the above, we can say

$f(n)$ is Big O $g(n)$

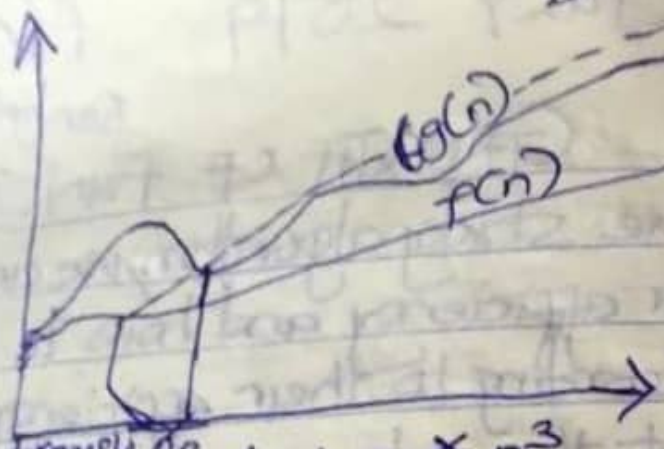
In other words, we can interpret that as n increases $f(n)$ grows no faster than $g(n)$

$\therefore g(n)$ is the asymptotic upper bound of $f(n)$

$f(n) = n^2 + n = O(n^3)$, Prove

Proof; Here, we have that $f(n) = n^2 + n$

$g(n) = n^3$



Example no

If $n >= 1$, $\therefore n \leq n^3$

$f(n) = 1+1 = 2$

~~$n^3 = 1^3 = 1$~~

If $n >= 1$ $\therefore n^2 \leq n^3$

In general, if $A \leq B$, $\therefore n^A \leq n^B$

Whenever $n \geq 1$

This is a fact that is often used in this type of proof. Therefore, for the above, we have $n^2+n \leq n^3+n^3$. But for the function of concern, the expression is equal to $2n^3$. $n^2+n \leq n^3+n^3 = 2n^3 \quad \forall n > 1$

We have shown that; $n^2+n = O(g(n))$

~~$n^2+n \leq 2n^3 \quad \forall n > 1$~~

We just proved that $n^2+n \leq 2n^3$ [i.e $c=2$ and $O(g(n)) = n^3$]

~~The~~ $f(n) = \Omega(g(n))$ ~~BIG(n^2) Notation~~ iff there ~~are~~ exist two positive constants c and n_0, such that the bounded $|f(n)| \geq c|g(n)| \quad \forall n \geq n_0$

If $f(n)$ is non-negative, we can simplify the condition by saying that;

~~$0 \leq f(n)$~~ $0 \leq c(g(n)) \leq f(n)$.

Then Big Ω
$f(n) \geq g(n)$

Big Θ Notation

$f(n) = \Theta(g(n))$ if there are some positive constant values c_1, c_2 and n_0 such that $c_1|g(n)| \leq c_2|g(n)| \leq n_0|g(n)|$
$\Rightarrow c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ Because c_1 and c_2 are positive constant for all +ve nos in N where $n \geq n_0$

If $f(n)$ is non-negative, we can now simplify the notation

$0 \leq c_1(g(n)) \leq f(n) \leq c_2(g(n))$ for $n \geq n_0$

$F(n)$ i.s $\Theta(g(n))$

So as n increases, $f(n)$ grows at the same rate as $g(n)$. Therefore $g(n)$ is asymptotically tight bound on $f(n)$

eg $n^2+5n+7 = \Theta(n^2)$

Proof; $f(n) = \Theta(g(n))$

$f(n) = n^2+5n+7$

When $n \geq 1$, $g(n) = n^2$

$n^2+5n+7 \leq n^2+5n+7 \leq n^2$

$n^2+5n+7 \leq n^2+5n^2+7n^2 \leq 13n^2$

When $n \geq 0$

$f(n)$ can exist in any form

n, $n+n$, $n+n+n$, 2^n

As n increases, $f(n)$ grows at the same rate with $g(n)$

Following from above, when $n \geq 1$, $n_0 = 1$, $c_1 = 1$ and $c_2 = 13$

That is also positive. It changes Initial/ starting Point

$$c_1(g(n)) \leq f(n) \leq c_2(g(n))$$

$$n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Arithmetic of Big O, Ω, Θ

(1) Transitivity; $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$

There is a transit $f(n) \in O(h(n))$

(2) $F(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$

$\Rightarrow F(n) \in \Theta(h(n))$

(3) $F(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$

$\Rightarrow F(n) \in \Omega(h(n))$

Scaling

If $f(n)$ is given, as $f(n)$ an element of $\Theta\, g(n)$. Then, if there exist a K that is greater than zero, we can re-formulate that, $f(n) \in O(Kg(n))$

As K grows, the value of $g(n)$ increases

Sum; If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$

$\Rightarrow f_1 + f_2(n) \in O(g_1 + g_2(n))$ - Growth

$\Rightarrow f_1 + f_2(n) \in \max(O g_1(n)) O(g_2(n))$

$F(n) = n^2 \cdot n^2 + n^2 + 5n + 7 \in Max(g_1(n), g_2(n))$

$F(n) = n^2 + 5n + 7$ //

24th July 2019

Analysis of Algorithms;

* Algorithmic strategies are the various approaches ~~designed~~ adopted/designed/defined for the purpose of increasing the efficiency of an algorithm.

The purpose of these approaches is to optimize time and space.

Some of the approaches include;

(1) Bruteforce approach involves on the rely power of the computer. If there is solution to the problem, definitely

e.g Linear search algorithm of Bruteforce algorithm, Bubble sort algorithm

(2) Divide and conquer approach; No matter how the data size in increased, it doesn't affect. e.g Merge sort, Binary search

$T(n) = O(\log_n)$

Assignment → Linear Search Algorithm, Bubble sort algorithm, merge sort, Binary Search, Travelling sales man algorithm

(3) ~~Branch and~~ Backtracking - The solution with least amount of space and time is selected for the algorithm eg travelling salesman Algorithm

(4) Branch and Bound

(5) Heuristics Algorithms - These are approximate algorithms that are based on ~~the~~ experience of the designer eg

19TH JUNE 2019

Expressing Algorithm;

E.g To sort out some numbers [32, 15, 12, 8, 17]

→ The first thing is to look at the numbers

(1) Look at the first number; Assume that the first number is the largest

(2) Look at the remaining items in the list and check if they are bigger than the first item.

(3) The last noted item is the largest in the list when the process is complete

The steps above used a natural language to solve a problem [sorting a set of numbers]

Using a Pseudocode;

(1) Form a name [Algorithm Largest Number]

(2) Input: A non-empty list of number L

(3) Output: The largest number in the list L

Largest ← L_0

(4) For each item in the list, DO

If item is greater than the largest (ie item > Largest) then, put the item as largest

(5) Return Largest

Assignment;
Represent the same Algorithm as a flow chart;
Definiteness - Instructions are clear and has a Unique Meaning.
Finiteness - This means that the algorithm must terminate and must produce an output.

Analysis of an Algorithm

An algorithm can be analysed by taking note of the following;

(1) We study the complexity of the algorithm
Algorithms are required for strong analysis of

(2) Check out how good your algorithm is for the problem to be solved.

(3) Compare the algorithm with other algorithm.

(4) At every time, you check the best algorithm.

Complexities as a key factor in the analysis of an algorithm;

Complexities are of 2 types -
Complexity of Space and Time
Complexity of Space - This is the number of bits and the number of elements.

No 5 - Practical : 37, 15, 17, 8, 17, 3, 24, 85

complexity of time - The number of operations that depend on the model.

RAM is also a factor under complexity of time.

Turing machine.

~~Dope~~

27TH JUNE 2019

BIG O [Order of growth

The BIG O comes with a capital notation is called the Landau's symbol. It is used in complexity theory, mathematics and computer science to describe the asymptotic behaviour of functions. Basically, It is us to describe how fast a function grow or declines. The Landau symbol is from a German name of the person that orig the growing symbol. hon

The O is used because this symbol repre Edmond Landau used this symbol t represent the order of growth, the O c from the word ~~de~~ order.

The BIG O notation is used when analyzing some algorithms. One might find the time or the numbers of steps it takes to complete a process of size N. Given in the following example

$T(n) = 4n^2 - 2n + 2$

$T(n)$ is growing at the order of n^2

$T(n) = O(n^2)$

~~In mathematics, it is important to~~

In $e^x = 1 + x + x^2/2$ [$O(x^2)$ - This is the order

This is to express the fact that the error is smaller in absolute value than x^2. Formally, suppose $f(x)$ and $g(x)$ are two functions defined on some subset of real numbers then

$f(x) = O(g(x))$

For all x defined on the space

i.e $x \to \infty$

This exists if and only if, there exist a constant N and C such that

$|f(x)| \leq c|g(x)|$ — ⊛

This will exist if and only if, $x \geq N$

This is a representation of the function and its growth rate and intuitively, it means that f does not grow faster than g.

If A is some real number

$f(x) = O(g(x))$ [O - For all $x > A$]

When a constant is introduced, the relation changes.

This can exist if and only if, there is a constant $D > 0$, such that the absolute value $|f(x)|$ is less than the constant of $g(x)$

$|F(x)| = c|g(x)|$

Dealing with positive Numbers of Natural Numbers N, the absolute value can be ignored if it is in mathematics.

Below is a list of classes of functions that are commonly encountered in the process of analysis of an algorithm.

Notation	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic

$O(\log(n)^c)$	Poly-logarithmic
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n)$	Exponential

Note that the last two are very difficult to analyze, the later grows much more faster no matter how big C is, a function that grows faster than any power of n is called super-Polynomial

One that grows slower than any exponential function of n is called Sub-Exponential

Question;

What is the difference between the super polynomial and subexponential

An algorithm can require time that is both super polynomial and subexponential An example is the fastest algorithm not for integer

Note that $O(\log n)$ is $O(\log(n)^c)$

The logarithmic and Polynomial

This is because the constant is ignored

i.e the hardware does not recognise the constant.

Similarly, logs with different constant bases are equivalent. If a function (f(x) is sum of functions, then the order is the one that grows faster.

For example, if $f(x) = 10\log n + 5\log n^3$

$f(x) = 10\log n + 5\log n^3 + 7n + 3n^2 + 6n^3$

Then $f(n) = n^3$

The order is $O(n^3)$

3RD JULY 2019

Understanding Complexities

- Interest is on large and complex algorithms which are usually determined by the complexity of the problem.
- To measure the complexities is by understanding the growth rate which is highly determined by the Running Time.

Interest to establish that an algorithm is efficient;

most efficient for a problem it has been designed or defined to solve. An algorithm is effective if it produces the desired output. It is efficient if it solves the problem within a minimal time.

An algorithm solves a problem considering the resource constraint. The Resource Constraint can exist as a relation to space or time ultimately an algorithm will be efficient if the resource constraints are well understood and clearly defined. Since algorithms are distinct, it follows that every algorithm must terminate. An algorithm can be expressed in many forms which are natural language, Pseudocode, mathematical expressions and relations for schematically, flowcharts.

Since algorithm describes a flow, diagrams based on flows essentially flowcharts are conventional used in the representation of an algorithm.

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$ If $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e, if there exists some positive constant c and some non-negative integer n_0 such that;

$t(n) \leq cg(n)$ for all $n \geq n_0$

n is extended to be a real number

e.g $100n+5 \in O(n^2)$

$t(n) \leq cg(n)$

$100n+5 \leq 100n+n$ (for all $n \geq 5$)

$= 101n \leq 101n^2$

Thus, as values of the constants c and n_0 required by the definition 101 and 5 respectively

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. e.g

$100n+5 \leq 100n+5n$ (for all $n \geq 1$) $= 105n$

to complete the proof with $c=105$

$n_0 = 1$

How to determine that an algorithm is efficient;
C.P.U usage
Data usage
Network Usage
Memory Usage
The CPU usage discusses about time complexity.
The performance of an algorithm;
This means how much time, memory, disk, that is actually used when the program is run.
~~The resources used~~ Performance of an algorithm;
How much of resources were used in the process of running a program to the end.

Complexities of an algorithm;
This determines how do the resources of a program or algorithm scales.
* Code determines performance but not complexities.
Complexities affect performance.

Complexity – How resources are scaled affect performance.

Performance – Is to determine the resources used in the cost of running a program (i.e the time, disc, codes)

Complexity is the measure of the scalability of the program. Therefore, Complexities affect performance. E.g If a 2GB memory compared to 6GB memory to run a program, the later

Complexity affects performance but performance does not affect complexity.

The time required by a function or a procedure is proportional to the number of basic operations it performs.

Examples of basic operations;

$f(x) = 2x^2 + x * 3$

The time taken for this function to be executed is directly proportional to the numbers of basic operation it performs.

$f(x) = 2x^2 + x \times 3$ Has two basic operation (+, x) and one assignment (=); also a text $f(x)$

Some functions procedure performs th

Stack(LIFO)

Stacksize — Tells you the number of elements in the stack. It returns some number of operations every time they are called example.

eg the stack size always return the number of elements in the stack.

The stack size is an operation that takes a constant time, other functions or operations may ~~take~~ perform a different number of operations depending on the values.

Eg The bubble sort algorithm considers a number of element in array and determines the nos of operations in the ~~prod~~ algorithm this problem is called problem size or input size

This parameter (i.e number of elements) is called problem size

When trying to find complexity, we are not interested in the exact number of ~~programs~~ operation performed but interested in the relation between the no of operations and problem size. In the case of complexity their interest is in the worst case.

[i.e maximum no of operations that might be performed for a given problem size)

Eg Inserting an element in an array

f(N) = O(g(N))

In the worst case, inserting at the beginning of an array.

A	B	C	D	E	F	G	H

The worst case above is inserting at A, which will make all other items to move.
The worst case time is linear (because it takes the same time to move an element at a time)
For a linear time algorithm, if the problem size doubles, the operation also doubles.
Formally; T(N) is O(f(N))
T(N) = O(f(N)); If for some contant C and for the values of N, N gets greater than n_0 (i.e $N > n_0$)
Increment exists for the increase in O
This is given as;
$T(N) <= C * f(N)$
$\forall C$ and values of $N > n_0$
Then $T(N) <= C * f(N)$
The idea is that T(N) is the exact complexity of a procedure function of an algorithm as a function of the proble

RT - simply Number of instructions (Also steps)

given as follows;

Memory consisting of infinite Arrays

And they

Instructions are executed, sequentially one at a time.

All instructions are executed over a unit time. For each instruction, there is a fixed time.

Instruction is kept on a temporary load ~~cache~~.

* Load/store
* Arithmetic instruction
* Logics

The RAM model is not completely realistic.

Memory is distinct, but when instructions are to be run on a memory arbitrarily memory seems to be infinite. (a phone a Portugal guy

- sms can be countless ~~host~~

Not all memory access takes the same time.

Instruction pipe-lining (parallel processing

- Takes a long time to run bulk instructio

Execution one after the other

Not all arithmetic operations take 1 same time.

List Arithmetic operators in the order of execution

Running Time of an Insertion sort of an Algorithm

It depends on:

(1) How ~~sor~~ the list is ordered (How sorted is the input)

1 4 7 3

2 6 (picking the inputs in the correct order)

(2) How big is the input size (For the running time to be determined, you focus on input size)

Insertion sort - Input size is "n"

assuming the line i used for instructions in the program

for J = 2 to n, do

Step 1 for J = 2 - n do

Key = A[J]

I = J - 1

While I > 0 and A(I) > Key

A[I+1] = A(I)

For every I - 1 I = i - 1

A[I+1] = Key (A[J])

OA3) insertion algorithm sort a pascal like pseudocode

This pseudocode is an invariate method of representing the insertion sort algorithm

To understand how many times each of the line of the program is executed

R_T [Number of instruction executed by RAM]

Let t_j be the number of time the line 4 is executed

	Cost	Time
For J = 2 to n,	c_1	n
Key = A[J]	c_2	n-1
I = J-1	c_3	n-1
While I > 0 and A[I] > Key do	c_4	$\sum_{j=2}^{n} t_j$
A[I+1] = A[I]	c_5	$\sum_{j=2}^{n} t_j - 1$

12TH JUNE 2019

Level of Ambiguity is a solid part of impriciseness; level of impricision are;

(i) Vagueness (ii) Ambiguity

(iii) Incompletness (iv) falseness

An algorithm is to ensure processes are efficient: for any given problem, it is true that an algorithm or the other algorithm c solve the problem because an algorith

Then;
As n increases, f(n) grows no slower than g(n).

Asymptotic lower bound ⟷

Highest of the upper bound

16th July 2019 Big Ω

Eg $n^3+4n^2 = \Omega(n^2)$ $f(n)=g(n)$

$f(n)=n^3+4n^2$ $g(n)=\Omega(n^2)$

f(n) grows no slower than g(n) $g(n)=n^2$

It is not hard to note that $n \geq 0$

$n^3 \leq n^3+4n^2$

If $n \geq 1$:

$n^2 \leq n^3$

Whenever you have n of a value greater than 1

Then; $n^2 \leq n^3+4n^2$ (for all value of $n \geq 1$)

$\Rightarrow \Omega(n^2)$ → the order of growth of n. Asymptotically lower bound. This analysis is to prove about order of growth. We have shown that

$n^3+4n^2 = \Omega(n^2)$

where $n_0=1$ and C is also given 1.

Size N and that f(N) is an upper bound of that complexity.
Because every other item is lesser than or equal to.
Every other operation will be lesser than the upper bound.
The smallest f(N) is the least upper bound of the actual complexity

show the complexity of the following

$T(N) = 3*N^2+5$

$T(n) = O(g(N))$

$T(n) = O(n^2)$

$T(n) = O(n^2)$ —

↓ order of growth