DSA Course Notes

INTRODUCTION TO ALOGORITHM AND ANALYSIS

Analysis of Algorithm

Example Problem: Sum of n natural numbers

Input: n=3

Output: 6 //1+2+3

Which code is faster depending upon the which system you are running the program one, which type programming language you used? Java and python program are interpreted language so they are generally slower than other programming language. System load is also the resistance toward fair measurement of efficiency of the code.

So, we use asymptotic notations where we use theoretical and mathematical calculations to check the real efficiency of the code. Avoiding the natural resistance like system load, programming language and system configuration.

Asymptotic Notation

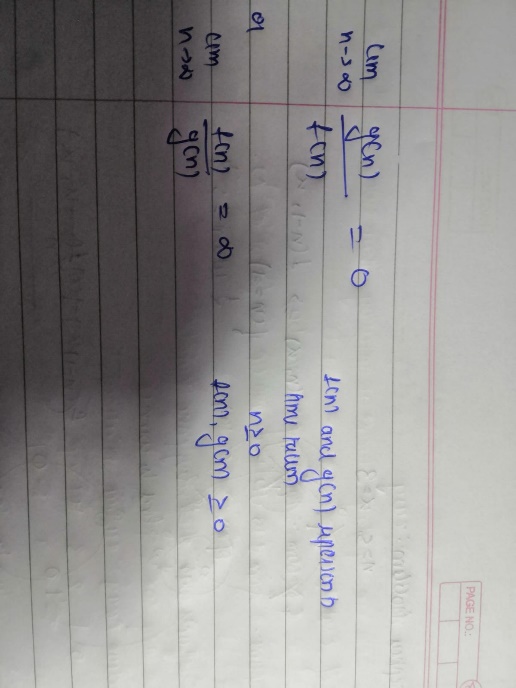
* The idea is to measure order of growth
* Does not depend upon machine, programming language etc.
* No need to implement, we can analyze algorithm.

N: depends upon the system configuration.

In Order of Growth, we have to take consideration of intersection point after which non-linear function takes more time than linear function.

Fun3() is Quadratic function which grows even more faster than any other type of function.

We will always 2-D Quadrant System.

Order Of Growth

A function f(n) is said to be growing faster than g(n) if

We always talk about larger-larger value of n.

Which is basically n🡪∞

f(n)=n+1 //Bad algorithm value not more 999 it will become good algorithm and it doesn’t matter.

g(n) =1000 //constant and good algorithm

Direct Way to Find and Compare Growths

1. Ignore lower order terms
2. Ignore leading term constant

Example: f(n)= 2n^2 + n + 6 //Order of Growth: n^2 (Quadratic)

g(n)= 100n + 3 //Order of Growth: n (Linear)

Comparison:

C< log (log n) < log n < n^1/3 < n^1/2 < n < n^2 < n^3 < n^4 < 2^n < n^n

Slowest ---------------------------------------------- fastest

Final order of growth always be single term

Best, Average and Worst Cases

Ex: sum of all elements of array

OOG: c1n + c2, O(n) (Order of Growth)

When condition or deciding factors comes in, we take three statements in-hands Best Case, Average Case and Worst Case.

Average Case is quite tricky we need to make some assumptions in order to calculate. This is the reason we mainly avoid this for measurement of efficiency. Same for the best case we don’t use this for measurement too.

We use worst case, how our code reacts when condition become worst and it gives accurate form of efficiency of the code. ‘O’ notation.

Asymptotic Notation

Big ‘O’: exact or upper order growth

Theta: exact growth

Omega: exact or lower

Example: C1n + C2 🡪 O(n) //remove the leading term constants.

O(1), Theta(1) is constant.

Big ‘O’ Notation

Direct way:

1. Ignore lower order terms
2. Ignore leading term constants

Example:

3n^2 + 5n + 6 🡪 O(n^2)

3n + 10nlog(n) + 3 🡪 O(nlog(n))

10n^3 + 40n + 10 🡪 O(n^3)

Mathematical Explanation of Big ‘O’

We say f(n) = O(g(n)), if there exist constants c and n˚ such that f(n) <= g(n) for all

n >= n˚

Example: f(n) = 2n + 3 can be written as O(n) [g(n)=n]

Let us take C=3

2n + 3 <= 3n, 3<=n we get n˚=3

Big ‘O’ Notations works for multiple variables also

100n^2 + 100m + n: O(n^2 + m)

100m^2 + 200mn + 30m + 20n: O (m^2 + mn)

Applications of Big ‘O’ notation

When you know we have exact iteration or growth do not use Big ‘O’ notation.

Example: To check the prime number algorithm: Sieve of Erasthonese

Bool isPrime (int n) {

If(n==1) return false;

If (n==2 || n==3) return true;

If (n%2==0 || n%3==0) return false;

For (int i=5; i\*i<=n; i=i+6) if (n%i == 0 || n%(i%2) == 0) return false;

Return true;

} Time Complexity: O(sqrt(n))

Omega Notation

f(n) = Ω(g(n)) if there exist constants C (where c>0) and n˚ (where n˚>=0) such that (g(n)<=f(n)) for all n>=n˚

Example:

f(n) = 2n + 3 = Ω(n)

c=1, n<= 2n +3; -3 <= n; n˚ = 0

if f(n) = O(g(n)), g(n) = Ω(f(n))

Theta Notation

F(n) = theta (g(n)) if there exist constants c1, c2 (where c1>0 and c2>0) and n˚ where n˚>=0 such that c1g(n) <= f(n) <= c2(g(n)) for all n>=n˚

Example: f(n) = 2n+3: theta(n) C1=1, C2=3

n <= 2n+3 <= 3n

Direct Method: 1000n^2 + 100nlog(n) + 2n: theta(n^2)

If f(n) = theta(g(n)) then f(n)=O(g(n)) and f(n) = Ω(g(n)) and g(n) = O(f(n)) and g(n)=Ω(f(n)), simple theta represents exact growth of algorithm and its is between upper and lower bound.

Analysis of Codes

In case of Recursive Function, we use the method called Recursive relations. Which is basically combination of base case and general case.

Recursion Tree Method for Solving Recurrences

1. We consider the recursion tree and compute total work done.
2. We write non-recursive part as root of the root of the tree and write the recursive part as children.
3. We keep expanding until we see a pattern.

Same Complexity

Order of growth of memory (or RAM) space in terms of input size. We same Asymptotic notation.

Example:

Int sumofnatural (int n) {

Return (n\* (n+1))/2;

} space complexity= theta (1)

Auxiliary Space: Order of growth of extra space or temporary space in terms of input size. We mainly use auxiliary space to measure the space efficiency of the code.

MATHEMATIC FOR PROGRAMMING

Prime Numbers: 2,3,5,7,11\_\_\_\_\_\_\_

Facts About Prime Number:

1. Every prime number can be represented in form of 6n+1 or 6n-1 except 2 and 3, where n is the natural number.
2. 2 and 3 are only prime number that are consecutive to each other.

LCM & HCF:

1. Factors: All the numbers that divide a number completely without leaving any reminder, are called factors of that number.
2. The Least common multiple, or LCM, of two or more numbers is the smallest number of other than zero that’s a multiple of each number.
3. Highest common factor/ Greatest Common divisor of two or more given numbers is the highest number which exactly divides all the numbers.
4. Factorials of a positive integer n, denoted by n!, is the product of all positive integer less than or equal to n.

Permutation & Combination

1. Permutation: Permutation is defined as arrangement of r things that can be done out of total n things.
2. Combination: Combination is defined as selection of r things that can be done out of total n things.

Modular Arithmetic

1. The reminder obtained after the division operation on two operands is known as modulo operation
2. Operator for doing modulus operation is ‘%’.

Iterative Power Binary Exponentiation

1. Every number can be written as sum of power of 2 (Set bits in binary)
2. We can traverse through all bits of a number (LSB to MSB) in O(logn) time