ALGOMANIAX LECTURES SERIES - II

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

ALGORITHM DESIGN
AND
TIME COMPLEXITY

ALGORITHM DEFINITION

A finite set of statements that guarantees an optimal solution in finite interval of time.

GOOD ALGORITHMS ?

- Run in less time

- Consume less memory

But computational resources (time complexity) is usually more important.

MEASURING EFFICIENCY

- The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size n.
  - Can be understood as the number of operations done in order to solve that problem.
- Time complexity mainly comes into picture when we deal with bigger inputs.
  - Small inputs which can be entered manually generally run in highly inefficient solutions too !

ANALYZING AN ALGORITHM

- Running time is measured by the number of steps / primitive operations performed.
- Steps mean elementary operations like :
  - + , - , * , / , = , == , A[i]
- We will measure number of steps taken in terms of size of the input.

EXAMPLE 1 :

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int s = 0;
for (int i=0; i<N; i++)
    s = s + A[i];
```

GROWTH OF THE COMPLEXITY

Complexity function of the previous example : 5N + 3.

Number of steps for different values of N :

- N = 10 → 53 steps
- N = 100 → 503 steps
- N = 1,000 → 5003 steps
- N = 1,000,000 → 5,000,003 steps

WHAT DOMINATES IN THE GROWTH ?

- As N gets large, the +3 becomes insignificant.
- 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance.

What is fundamental is that the time is *LINEAR* in N.

ASYMPTOTIC COMPLEXITY

- As N gets large, concentrate on the highest order term or Dominant term (for polynomials)
  - Drop lower order terms such as +3
  - Drop the constant coefficient of the highest order term i.e. N.
- The 5N+3 time bound is said to “grow asymptotically” like N.
- This gives us an approximation of the complexity of the algorithm.

EXAMPLE 2 :

```
// Input: int A[N], array of N integers
// Output: Sum of products of all pairs in array.

int sum = 0;

for (int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        sum = sum + A[i]*A[j];
```

COMPARING FUNCTIONS : ASYMPTOTIC NOTATIONS

- Big Oh Notation : Upper Bound
- Omega Notation : Lower Bound
- Theta Notation : Tighter Bound

BIG-OH NOTATION

If $f(N)$ and $g(N)$ are two complexity functions, we say

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

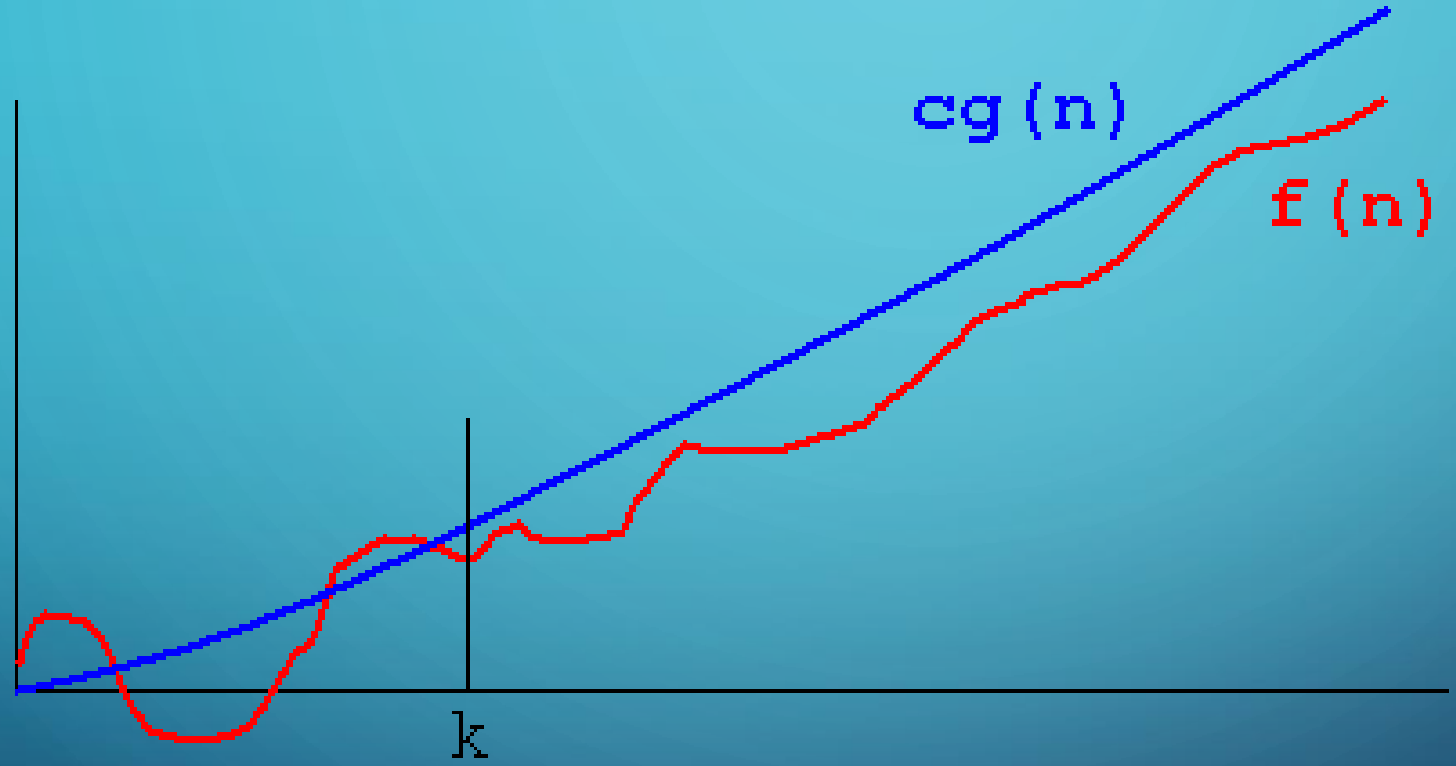
(read "f(N) is order g(N)", or "f(N) is big-O of g(N)")

If there are constants c and N_0 such that for all $N > N_0$,

$$f(N) \leq c * g(N)$$

for all sufficiently large N.

BIG-OH NOTATION



COMPARING FUNCTIONS

- As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order.
  - $f(n) = 7n - 3$ will always be lesser than $g(n) = 4n^3 + 2$
- Even though it is correct to say "$7n - 3$ is $O(n^3)$", a better statement is "$7n - 3$ is $O(n)$", that is, one should make the approximation as tight as possible.
- Simple Rule: Drop lower order terms and constant factors
  - $\mathbf{7n-3}$ is $O(n)$
  - $\mathbf{8n^2 \log n + 5n^2 + n}$ is $O(n^2 \log n)$

BIG OMEGA NOTATION

- If we wanted to say "running time is at least…" we use Ω
- Big Omega notation, Ω, is used to express the lower bounds on a function.
- If f(n) and g(n) are two complexity functions then we can say:

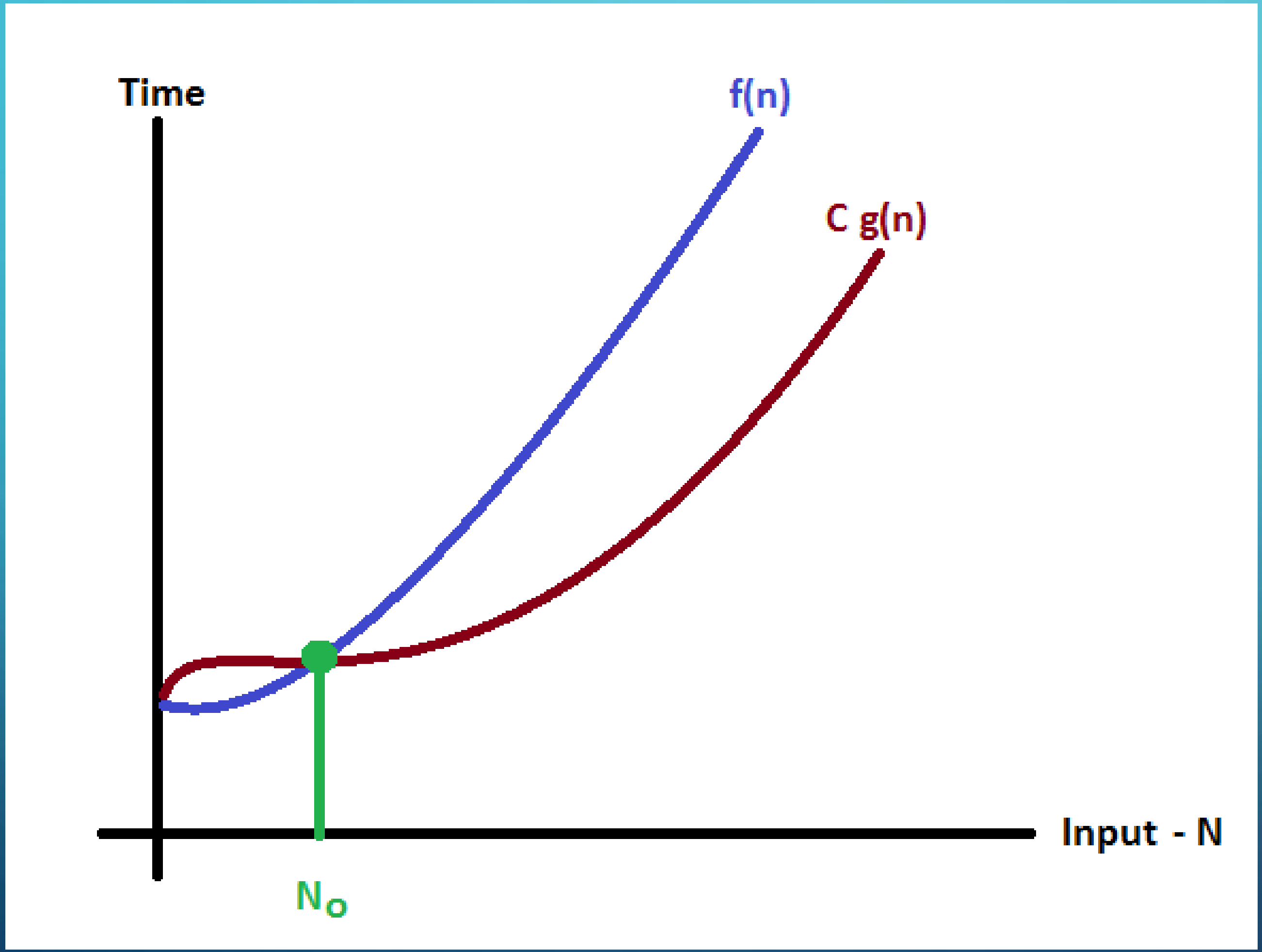
$$f(N) = \Omega(g(N))$$

If there are constants c and N_0 such that for all $N > N_0$,

$$f(N) \geq c * g(N)$$

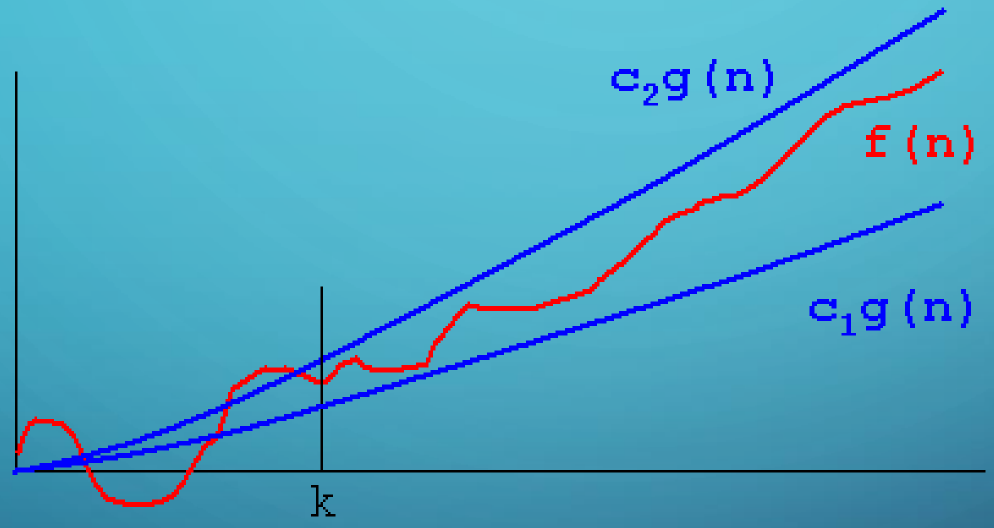
for all sufficiently large N.

BIG OMEGA NOTATION



BIG THETA NOTATION

- If we wish to express tight bounds we use the theta notation, Θ
- **f(n) = Θ(g(n)) means that f(n) = O(g(n)) and f(n) = Ω(g(n))**



- **The Theta notation for two functions may or may not exist. We can safely say that if both the Big-Oh and Omega complexities of a function f(x) are the same, the Theta notation for that function exists and is equal to both O(f(x)) and Ω(f(x)).**

WHAT DOES THIS ALL MEAN?

- If $f(n) = \Theta(g(n))$ we say that $f(n)$ and $g(n)$ grow at the same rate, asymptotically
- If $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$, then we say that $f(n)$ is asymptotically slower growing than $g(n)$.
- If $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$, then we say that $f(n)$ is asymptotically faster growing than $g(n)$.

WHICH NOTATION DO WE USE?

"Expect for the best but prepare for the worst !"

- The Big-Oh notation gives the upper bound, i.e. the **Worst-Case Complexity** of our piece of code.
- The Big-Omega notation gives the lower bound, i.e. the **Best-Case Complexity** of our piece of code.

- If we know the worse case then we can aim to improve it and/or avoid it. Hence, we generally like to express our algorithms as **big Oh** since we would like to know the upper bounds of our algorithms.

EXAMPLES

EXAMPLE 1 :

Question : Find the Big-Oh Time Complexity of the following code snippet. Assume inputs to the array are already present.

```
int a[n], i, j, sum = 0;

for(i = 0; i<n; i++)
{
    for(j = i+1; j<n; j++)
    {
        sum++;
    }
}
```

EXAMPLE 2 :

```
int i, sum = 0;

for(i=0; i<n; i++);
    for(j=0 j<n; j++)
        sum++;
```

EXAMPLE 3 :

```
int i, sum = 0, n, m;

for(i=0; i<n; i++)
    for(j=0 j<m; j++)
        sum++;
```

EXAMPLE 4 :

```
int sum = 0, n;

while( n )
{
    n /= 2;
    sum++;
}
```

EXAMPLE 5 :

Question : Find the exact number of times the loop runs for. Assume inputs are already available.

```
int t, sum = 0;
cin >> t;

while(t--)
{
    sum++;
}
```

EXAMPLE 6 :

Question : Find the exact number of times the loop runs for. Assume inputs are already available.

```
int t, sum = 0;
cin >> t;

while(--t)
{
    sum++;
}
```

EXAMPLE 7 :

Question : Find both the Best-Case and the Worst-Case time complexity of the given snippet of code.

```
int a[n], k, i;

for(i = 0; i<n; i++)
    if( a[i] == k )
        break;
```

EXAMPLE 8 :

```
int choice, i, j, n, sum = 0;

if(choice == 1)
{
    for(i=0; i<n; i++)
        sum++;
}
else if(choice == 2)
{
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            sum++;
}
```

EXAMPLE 9 :

Question : Find the time complexity of the given functions. Assume function is called once from main( ).

```
int foo(int n)
{
    int sum = 0;
    for(i=1; i<=n; i++)
        sum += i;

    return sum;
}
```

EXAMPLE 10 :

```
int foo(int n)
{
    if(n == 1)
        return 1;
    else
        return n + foo(n-1);
}
```

EXAMPLE 11 :

```
int pow(int a, int n)
{
    if(n == 0)
        return 1;
    else
        return a*pow(a, n-1);
}
```

EXAMPLE 12 :

```
int foo(int n)
{
    if(n == 0)
        return 1;
    else
        return foo(n-1) + foo(n-1);
}
```

EXAMPLE 13 :

```
int foo(int n)
{
    if(n == 0)
        return 1;
    else
    {
        int ans = foo(n-1);
        return 2*ans;
    }
}
```

EXAMPLE 14 :

```
int pow(int a, int n)
    {
        if(n==1)
            return a;
        else if( n&1 )
        {
            int pow2 = pow(a, n/2);
            return a*pow2*pow2;
        }
        else
        {
            int pow1 = pow(a, n/2);
            return pow1*pow1;
        }
    }
```

PERFORMANCE CLASSIFICATION

f(n)	Classification
1	*Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
log n	*Logarithmic*: when n increases, so does run time, but much slower. Common in programs which solve large problems by transforming them into smaller problems.
n	*Linear*: run time varies directly with n. Typically, a small amount of processing is done on each element.
n log n	When n doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions
n^2	*Quadratic*: when n doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
n^3	*Cubic*: when n doubles, runtime increases eightfold
2^n	*Exponential*: when n doubles, run time squares. This is often the result of a natural, "brute force" solution.

LINKING IT WITH CONSTRAINTS

- The "Constraints" in the questions on HackerRank, CodeChef are given mainly due to two reasons :
  - To give an idea of the data types to be used.
  - To get an idea of the complexity that would pass for that question.
- The compiler can only perform ~10^7-10^8 operations per second, so we need to write "optimal" and "efficient" code which passes in the given Constraints.
- Writing inefficient code on submission gives "*TLE (Time Limit Exceeded)*" or "*Terminated due to Timeout*" errors.

WHAT COMPLEXITY ALGORITHM TO WRITE ?

f(n)	Max Passing Constraints	Examples
1	Anything you can store !	Generally used when answer is a direct formula.
log n	2^{10^7} (Extremely HUGE)	Repeated division, exponentiation.
n	$\sim 10^7$	Normal array traversal , going from 1 to N etc.
n log n	$\sim 10^5$ - 10^6	Sorting an array, and used in a LOT of data structures.
n^2	~5000	Another very common complexity, used in a lot of array and DS questions.
n^3	~500	Not a very common time complexity, generally ad-hoc problems
2^n	~25-30	Brute-force solutions, finding all permutations of an array/string.