Lecture 03-04

Program Efficiency & Complexity Analysis

By: Dr. Zahoor Jan

Algorithm Definition

A <u>finite</u> set of statements that <u>guarantees</u> an <u>optimal</u> 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.
  - The resource we are most interested in is time
  - We can use the same techniques to analyze the consumption of other resources, such as memory space.

- It would seem that the most obvious way to measure the efficiency of an algorithm is to run it and measure how much processor time is needed

- Is it correct ?

Factors

- Hardware
- Operating System
- Compiler
- Size of input
- Nature of Input
- Algorithm

Which should be improved?

RUNNING TIME OF AN ALGORITHM

- Depends upon
  - Input Size
  - Nature of Input
- Generally time grows with size of input, so running time of an algorithm is usually measured as function of input size.
- Running time is measured in terms of number of steps/primitive operations performed
- Independent from machine, OS

Finding running time of an Algorithm / Analyzing an Algorithm

- Running time is measured by number of steps/primitive operations performed

- Steps means elementary operation like
  - ,+, *,<, =, A[i] etc

- We will measure number of steps taken in term of size of input

Simple Example (1)

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

int Sum(int A[], int N)
{
  int s=0;
  for (int i=0; i< N; i++)
    s = s + A[i];
  return s;
}
```

How should we analyse this?

Simple Example (2)

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

int Sum(int A[], int N){
    int s=0;

    for (int i=0; i< N; i++)

        s = s + A[i];

    return s;
}
```

1,2,8: Once

3,4,5,6,7: Once per each iteration of for loop, N iteration

Total: 5N + 3

The *complexity function* of the algorithm is : $f(N) = 5N + 3$

Simple Example (3) Growth of 5n+3

Estimated running time 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

As N grows, the number of steps grow in *linear* proportion to N for this function "*Sum*"

What Dominates in Previous Example?

What about the +3 and 5 in 5N+3?

- 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.

<u>Asymptotic Complexity</u>: As N gets large, concentrate on the highest order term:

- Drop lower order terms such as +3
- Drop the constant coefficient of the highest order term i.e. N

Asymptotic Complexity

- The 5N+3 time bound is said to "grow asymptotically" like N
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of (machine dependent) details, concentrate on the bigger picture

COMPARING FUNCTIONS: ASYMPTOTIC NOTATION

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

Big Oh Notation [1]

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

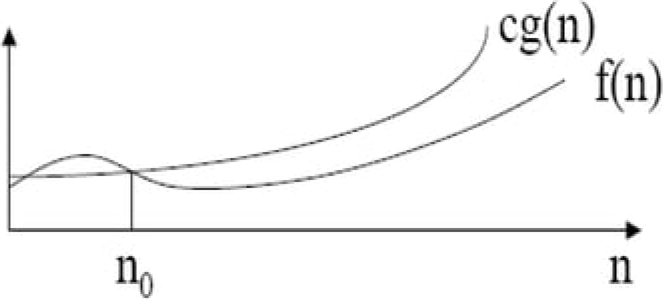
$$f(N) = O(g(N))$$

(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 $N > N_0$,

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

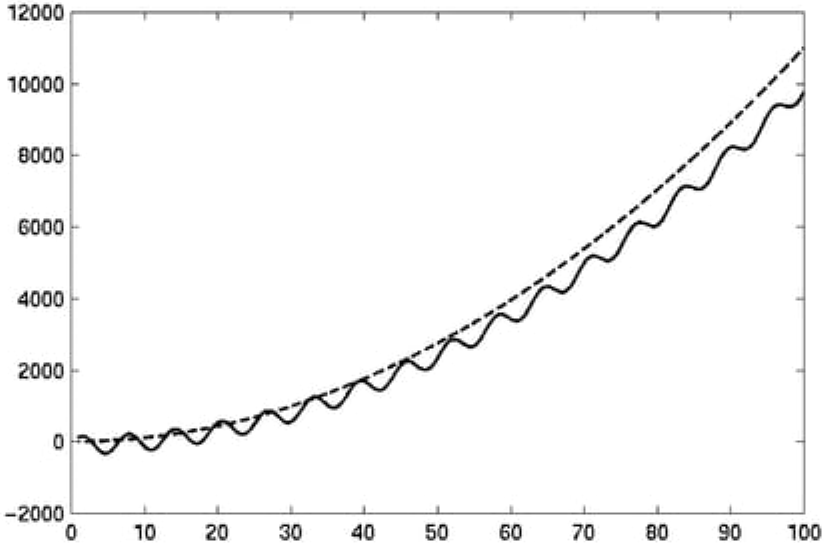
for all sufficiently large N.

Big Oh Notation [2]



- Function cg(n) always dominates f(n) to the right of n_0

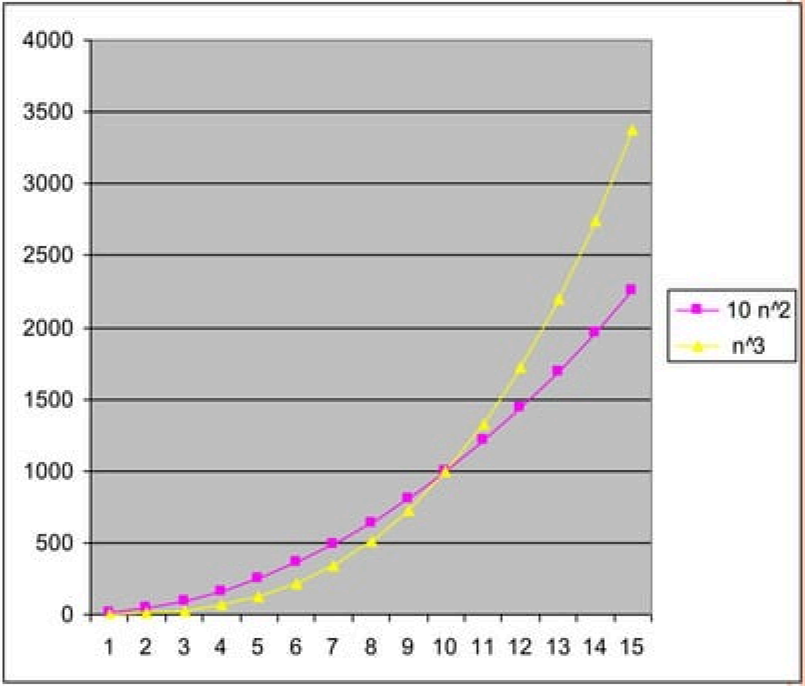
O(F(N))


12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

Example (2): Comparing Functions

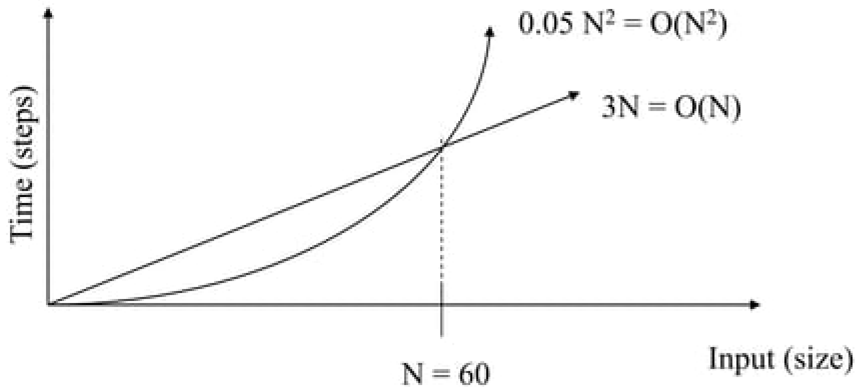
- Which function is better?

  $10\,n^2$ Vs n^3



Comparing Functions

- As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order



Big-Oh Notation

- 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

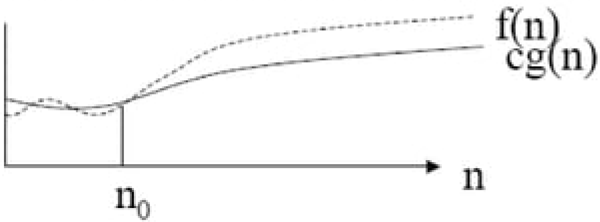
  $7n-3$ is $O(n)$

  $8n^2 \log n + 5n^2 + n$ 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) is $\Omega(g(n))$ if there exist positive numbers c and n_0 such that $0<=f(n)>=c\Omega$ (n) for all $n>=n_0$

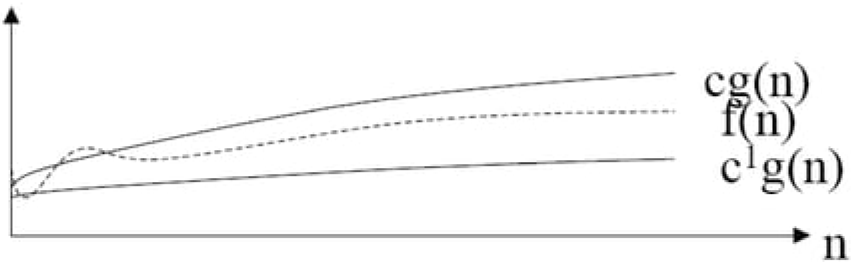


- In this instance, function cg(n) is dominated by function f(n) to the right of n_0

- Example : $3n + 2 = \Omega(n)$

Big Theta Notation

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



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?

- To express the efficiency of our algorithms which of the three notations should we use?
- As computer scientist we generally like to express our algorithms as big O since we would like to know the upper bounds of our algorithms.
- Why?
- If we know the worse case then we can aim to improve it and/or avoid it.

Performance Classification

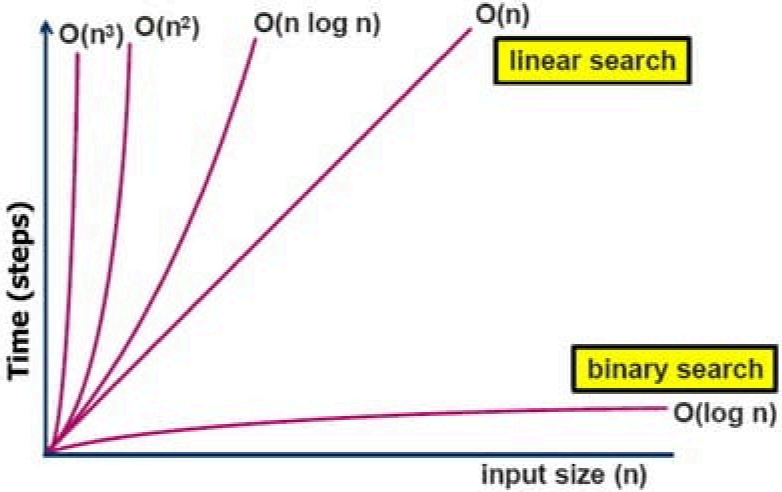
f(*n*)	Classification
1	*Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
log n	*Logarithmic*: when *n* increases, so does run time, but much slower. Common in programs which solve large problems by transforming them into smaller problems. Exp : binary Search
n	*Linear*: run time varies directly with *n*. Typically, a small amount of processing is done on each element. Exp: Linear Search
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. Exp: Merge
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). Exp: Insertion Search
n^3	*Cubic*: when n doubles, runtime increases eightfold. Exp: Matrix
2^n	*Exponential*: when n doubles, run time squares. This is often the result of a natural, "brute force" solution. Exp: Brute Force. Note: logn, n, nlogn, n^2>> less Input>>Polynomial n^3, 2^n>>high input>> non polynomial

Size does matter[1]

What happens if we double the input size N?

N	$\log_2 N$	5N	$N \log_2 N$	N^2	2^N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

COMPLEXITY CLASSES



SIZE DOES MATTER[2]

- Suppose a program has run time O(n!) and the run time for n = 10 is 1 second

  For n = 12, the run time is 2 minutes

  For n = 14, the run time is 6 hours

  For n = 16, the run time is 2 months

  For n = 18, the run time is 50 years

  For n = 20, the run time is 200 centuries

Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

Constant time statements

- Simplest case: O(1) time statements
- Assignment statements of simple data types
  int x = y;
- Arithmetic operations:
  x = 5 * y + 4 - z;
- Array referencing:
  A[j] = 5;
- Array assignment:
  $\forall$ j, A[j] = 5;
- Most conditional tests:
  if (x < 12) ...

Analyzing Loops[1]

- Any loop has two parts:
  - How many iterations are performed?
  - How many steps per iteration?

```
int sum = 0,j;
for (j=0; j < N; j++)
  sum = sum +j;
```

  - Loop executes N times (0..N-1)
  - 4 = O(1) steps per iteration
- Total time is N * O(1) = O(N*1) = O(N)

Analyzing Loops[2]

- What about this **for** loop?

```
int sum =0, j;
for (j=0; j < 100; j++)
   sum = sum +j;
```

- Loop executes 100 times

- 4 = O(1) steps per iteration

- Total time is 100 * O(1) = O(100 * 1) = O(100) = O(1)

Analyzing Loops – Linear Loops

- Example (have a look at this code segment):

```
int i = 1;
while (i < n)
{
    <<list of sequence statements>>
    i++;
}
```

Executed 1 times

n comparisons

Lets presume there is constant c steps present here; so we have c*n steps

Executed n times

- Efficiency is proportional to the number of iterations.
- Efficiency time function is :

$$f(n) = 1 + (n-1) + c^*(n-1) +(n-1)$$
$$= (c+2)^*(n-1) + 1$$
$$= (c+2)n – (c+2) +1$$

- Asymptotically, efficiency is : O(n)

ANALYZING NESTED LOOPS[1]

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;
for (j=0; j<N; j++)
  for (k=N; k>0; k--)
    sum += k+j;
```

- Start with outer loop:
  - How many iterations? N
  - How much time per iteration? Need to evaluate inner loop
- Inner loop uses O(N) time
- Total time is N * O(N) = O(N*N) = $O(N^2)$

Analyzing Nested Loops[2]

- What if the number of iterations of one loop depends on the counter of the other?

```
int j,k;
for (j=0; j < N; j++)
   for (k=0; k < j; k++)
      sum += k+j;
```

- Analyze inner and outer loop together:
- Number of iterations of the inner loop is:
- $0 + 1 + 2 + ... + (N-1) = O(N^2)$

How Did We Get This Answer?

- When doing Big-O analysis, we sometimes have to compute a series like: 1 + 2 + 3 + ... + (n-1) + n

- i.e. Sum of first n numbers. What is the complexity of this?

- Gauss figured out that the sum of the first n numbers is always:

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2} = \frac{n^2 + n}{2} = O(n^2)$$

Sequence of Statements

- For a sequence of statements, compute their complexity functions individually and add them up

```
for (j=0; j < N; j++)
    for (k =0; k < j; k++)
        sum = sum + j*k;
for (l=0; l < N; l++)
    sum = sum -l;
System.out.print("sum is now"+sum);
```

$O(N^2)$

$O(N)$

$O(1)$

- Total cost is $O(n^2) + O(n) + O(1) = O(n^2)$

Conditional Statements

- What about conditional statements such as

```
if (condition)
        statement1;
else
        statement2;
```

- where statement1 runs in $O(n)$ time and statement2 runs in $O(n^2)$ time?
- We use "worst case" complexity: among all inputs of size n, what is the maximum running time?
- The analysis for the example above is $O(n^2)$

DERIVING A RECURRENCE EQUATION

- So far, all algorithms that we have been analyzing have been non recursive

- Example : Recursive power method

```
double power( double x, int n) {
    if ( n == 0)
        return 1.0;                    // base calse
    //else
        return power(x, n-1)*x;        // recursive case
}
```

- If N = 1, then running time T(N) is 2

- However if N ≥ 2, then running time T(N) is the cost of each step taken plus time required to compute power(x,n-1). (i.e. T(N) = 2+T(N-1) for N ≥ 2)

- How do we solve this? One way is to use the iteration method.

ITERATION METHOD

- This is sometimes known as "Back Substituting".
- Involves expanding the recurrence in order to see a pattern.
- Solving formula from previous example using the iteration method :
- ***Solution*** : Expand and apply to itself :

  Let T(1) = n0 = 2

  $$
  \begin{aligned}
  T(N) &= 2 + T(N-1) \\
  &= 2 + 2 + T(N-2) \\
  &= 2 + 2 + 2 + T(N-3) \\
  &= 2 + 2 + 2 + \ldots\ldots + 2 + T(1) \\
  &= 2N + 2 \text{ remember that } T(1) = n0 = 2 \text{ for } N = 1
  \end{aligned}
  $$

- So T(N) = 2N+2 is O(N) for last example.

Summary

- Algorithms can be classified according to their complexity => O-Notation
  - only relevant for large input sizes
- "Measurements" are machine independent
  - worst-, average-, best-case analysis

REFERENCES

Introduction to Algorithms by Thomas H. Cormen
Chapter 3 (Growth of Functions)