# Complexity Analysis of an Algorithm

# Algorithm Complexity-Why?

- Resources required for running that algorithm
- To estimate how long a program will run.
- To estimate the largest input that can reasonably be given to the program.
- To compare the efficiency of different algorithms.
- To help focus on the parts of code that are executed the largest number of times.
- To choose an algorithm for an application

# Memory Issue

- It is true that memory capacity is increasing at cache, primary and secondary level, but the complexity, sophistication and volume of applications that are required to be accommodated is also increasing
- Memory now mainly concerned with bandwidth, small gadgets, and smart cards. Vista requires 1 GB of RAM. Running in RAM or Cache is still a big issue
- Small programs more efficient (Concept of time space trade off does not hold good always)

# Empirical Study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock(), system.currentTimeMiilis() function, to get an accurate measure of the actual running time
- Plot the results
- Time in NSec/Size
- Good for embedded/small devices or where the product is to be manufactured in millions of units

# Difficulties in Empirical Study

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.
- Even in same hardware and software environments the results may vary depending upon the processor load, sharing of resources, No. of background processes, Actual status of Primary and Secondary Memory at the time of running the program, Compiler, Network Architecture, programming language

# Apriori Vs Posterior Analysis

- Apriori – Designing then making
- Posterior - Making then waking up after the problem crops up
- Posterior Analysis refers to the technique of coding a given solution and then measuring its efficiency. it provides the actual time taken by the program. This is useful in practice but Apriori is always better.
- There is corresponding guarantee that any algorithm that is better in performance in its apriori analysis will be always better in performance in its posterior analysis. Vice-Versa may not be true.

# Instruction Count as a measure

- Is not related to type of input (input size in terms of number of bytes like in factoring, or the number of inputs or the type of input like in sorting)
- Intelligent compilers can make your count useless
- Language/ Compiler/interpreter issues will also play a role
- Parallel/ pipelining/superscalar executions
- Different instructions may be very differently loaded in terms of resource requirements
- Can be a first preliminary indication to compare the size of two algorithms but should not cheat you
- Only 10% of the instructions may be actually responsible for the 90% of resource usage

# Micro Analysis

- To count each and every operation of the program.
- Detailed, Takes more time and is complex and tedious
- Average lines of codes are in the range of 3 to 5 million lines
- Those operations which are not dependent upon the size or number of the input will take constant time and will not participate in the growth of the time or space function, So they need not be part of our analysis

# Macro Analysis

- Deals with selective instructions which are dominant & costliest. Selection of right instructions is very important
- Comparisons and Swapping are basic operations in sorting algorithms
- Arithmetic operations are basic operations in math algorithms
- Comparisons are basic operations in searching algorithms
- Multiplication and Addition are basic operations in matrix multiplication

# Worst Case Analysis

- Goodness of an algorithm is most often expressed in terms of its worst-case running time.
- Need for a bound on one's pessimism, Every Body needs a guarantee. This is the maximum time an algorithm will take on a given input size
- Ease of calculation of worst-case times
- In case of critical systems we cannot rely on average or best case times
- Worst Case for all sorting problems is when the inputs are in the reverse order

# Average Case Analysis

- Very difficult to compute
- Average-case running times are calculated by first arriving at an understanding of the average nature of the input, and then performing a running-time analysis of the algorithm for this configuration
- Needs assumption of statistical and probabilistic distribution of input e.g. uniform probability distribution
- It is supposed that all inputs are equally likely to occur
- If we have the same worst case time for two algorithms then we can go for average case analysis
- If the average case is also same then we can go for micro analysis or empirical analysis

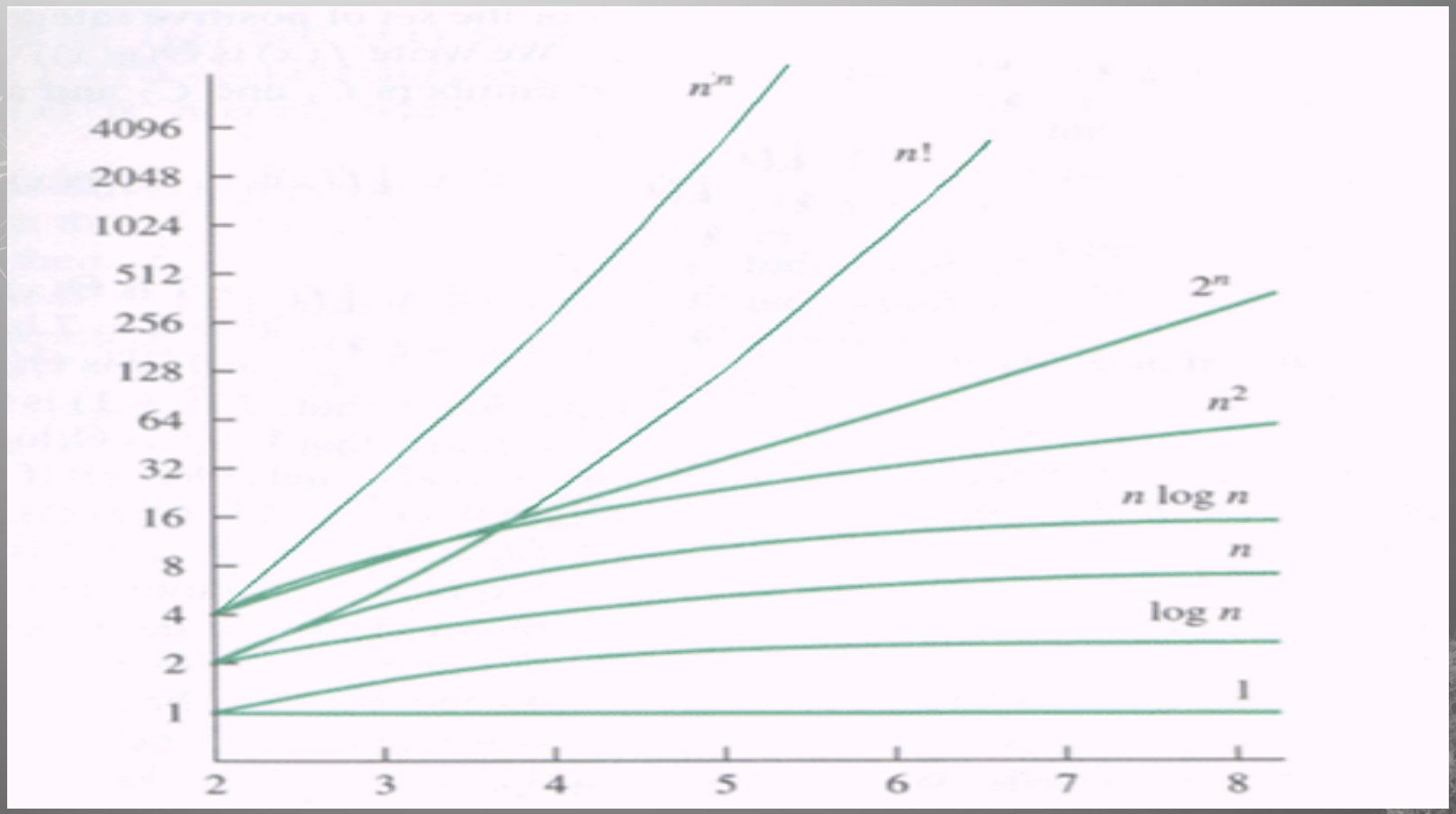
# Best Case Analysis

- Not used in general
- Best case may never occur
- Can be a bogus or cheat algorithm that is otherwise very slow but works well on a particular input
- A particular input is likely to occur more than 90% of the time then we can go for a customized algorithm for that input
- Best Case for all sorting problems is that sequence is already in the sorted sequence

# Asymptotic Analysis

- Asymptotic analysis means studying the behavior of the function when n→ infinity or very large
- Problems size will keep on increasing so asymptotic analysis is very important
- Limiting behavior
- Nested loops should be analyzed inside out. The total running time for a statement inside innermost loop is given by its running time multiplied by the product of the sizes of all for loops
- The running time of an if/else statement is not more than the running time of the test, plus the larger of the running times of statements contained inside if and else conditions
- We are more concerned with large input size because small size inputs will not very much in running time

# Relative Goodness of Functions



| Function/value | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | $2048=2^{11}$ | $2^{20}$ | $2^{30}$ | $2^{40}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lg n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 30 | 40 |
| N | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 | $2^{30}$ | $2^{40}$ |
| N logn | 2 | 8 | 24 | 64 | 160 | 384 | 896 | 2048 | 4608 | 10240 | 22528 | 49152 | $2^{35}$ | $2^{46}$ |
| $N^2$ | 4 | 16 | 64 | 256 | 1024 | 4096 | 16384 | 65536 | 262144 | 1048576 | 4194304 | $2^{40}$ | $2^{60}$ | $2^{80}$ |
| $N^3$ | 8 | 64 | 512 | 4096 | 32768 | 262144 | 2097152 | 16777216 | $2^{27}$ | $2^{30}$ | $2^{33}$ | $2^{60}$ | $2^{90}$ | $2^{120}$ |
| $N^{10}$ | $2^{10}$ | $2^{20}$ | $2^{30}$ | $2^{40}$ | $2^{50}$ | $2^{60}$ | $2^{70}$ | $2^{80}$ | $2^{90}$ | $2^{100}$ | $2^{110}$ | $2^{200}$ | $2^{300}$ | $2^{400}$ |
| $2^N$ | 4 | 16 | 256 | $2^{16}$ | $2^{32}$ | $2^{64}$ | $2^{132}$ | $2^{256}$ | $2^{512}$ | $2^{1024}$ | $2^{2048}$ | ∞ | ∞ | ∞ |
| N! | 2 | 24 | 40320 | 2092278988 8000 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

| Function/size of the problem | 1 | 10 | 50 | 60 | 100 | 1000 | $10^4$ | $10^5$ | $10^6$ | $10^{12}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Log n | $10^{-12}$ | $3*10^{-11}$ | $5*10^{-11}$ | $5*10^{-11}$ | $6*10^{-11}$ | $10^{-11}$ | 3.3*10-10 | $6.5*10^{-10}$ | $10^{-10}$ | $10^{-7}$ |
| N | $10^{-12}$ | $10^{-11}$ | $5*10^{-10}$ | $6*10^{-10}$ | $10^{-10}$ | $10^{-9}$ | $10^{-08}$ | $10^{-07}$ | $10^{-06}$ | 1 sec |
| N lgn | $10^{-12}$ | $3*10^{-10}$ | $5*10^{-9}$ | $6*10^{-08}$ | $6.64*10^{-9}$ | $9.965*10^{-8}$ | $1.3*10^{-7}$ | $1.5*10^{-04}$ | $1.8*10^{-03}$ | 40 sec |
| $N^2$ | $10^{-12}$ | $10^{-10}$ | $2.5*10^{-9}$ | $3.3*10^{-9}$ | $10^{-08}$ | $10^{-06}$ | $10^{-04}$ | $10^{-02}$ | 1 | 317 centuries |
| $N^3$ | $10^{-12}$ | $10^{-09}$ | $8*10^{-7}$ | $10^{-7}$ | $10^{-06}$ | $10^{-03}$ | 1 | 16.6 minutes | 12 days | ∞ |
| $N^{10}$ | $10^{-12}$ | $10^{-02}$ | 1.13 days | 7 days | 3.17 years | ∞ | ∞ | ∞ | ∞ | ∞ |
| $2^N$ | $0.5*10^{-11}$ | $10^{-09}$ | 16 min | 12 days | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| N! | $1*10^{-12}$ | $3.6*10^{-6}$ n=16 5 minus | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

The utility of log n and relative difficulty of the functions. What are the practical limits of every function?

Show- The utility of log n and relative difficulty of the functions. What are the practical limits of every function?

| | 1 second | 1minute | 1 hour | 1 month | 1 year |
|---|---|---|---|---|---|
| Logn | $2^{1000000000000}$ | $2^{60000000000000}$ | ∞ | ∞ | ∞ |
| N | $10^{12}$ | $6*10^{13}$ | $3.6*10^{15}$ | $2.5*10^{18}$ | $3.1*10^{19}$ |
| Nlogn | $10^{12}$ | $6*10^{13}$ | $3.6*10^{15}$ | $2.5*10^{18}$ | $3.1*10^{19}$ |
| $N^2$ | $10^6$ | $7.74*10^6$ | $6.0*10^7$ | $1.6*10^9$ | $5.6*10^9$ |
| $N^3$ | $10^4$ | $3.9*10^4$ | $1.53*10^5$ | $1.37*10^6$ | $3.15*10^6$ |
| $N^{10}$ | 16 | 24 | 36 | 70 | 90 |
| $2^N$ | 40 | 46 | 52 | 61 | 65 |
| N! | 14 | 15 | 16 | 18 | 20 |

# Largest Growth Terms

- Why lower order terms are not required
- As seen in the previous tables if there is a function of higher growth term present then lower growth function becomes negligible for the large values of n, so all lower growth functions in the expression can be discarded. The constant in the highest growth term can be discarded because that does not participate in the growth of the function. This value can also change with change in implementation specifics. But anyhow the importance of it cannot be overlooked.

# Big Oh Notation

Independent of hardware and software and valid for any input

$f(n) = O(g(n))$ if there are positive constants c and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ and $c>0$ This notation is known as Big-Oh notation

| | |
|---|---|
| $n^2+3n+4 \leq 2n^2$ *for all* $n_o > 10$ *is* | $O(n^2)$ |
| $3 \log n + \log \log n \leq 4 \log n$ *for all* $n_o>2$ | $O(\log n)$ |
| $3n^3+20n^2+5 \leq 4n^3$ *for all* $n_o>21$ | $O(n^3)$ |
| $7n-2 \leq 7n$ *for all* $n_o>1$ | $O(n)$ |
| $a_0n^0+a_1n^1+a_2n^2+\ldots\ldots\ldots a_kn^K$ | $O(n^k)$ |

$n^2$ is not $O(n)$

$n^2 \leq cn$ *and* $n \leq c$ cannot be satisfied since *c* must be a constant

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

# Big Omega Notation

how to find the constant for the sake of argument

$n^2 + 3n + 4 \geq n^2$ *for all* $n_o > 1$ $\Omega(n^2)$,

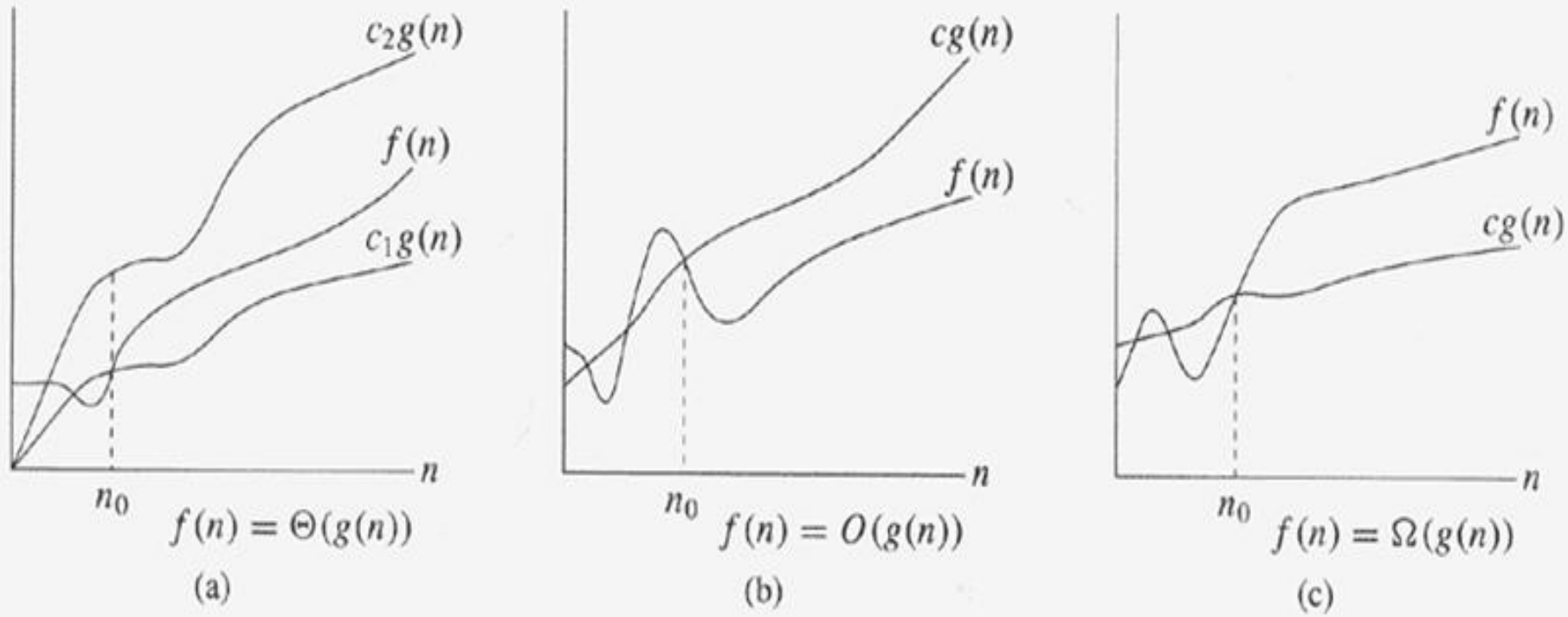
$3 \log n + \log \log n \geq \log n$ *for all* $n_o > 2$ $\Omega(\log n)$

$3n^3 + 20n^2 + 5 \geq n^3$ *for all* $n_o > 1$ $\Omega(n^3)$

$7n-2 \geq n$ *for all* $n_o > 1$ $\Omega(n)$

$a_0n^0+a_1n^1+a_2n^2+\ldots\ldots\ldots a_kn^K$ $\Omega(n^k)$

$f(n) = \Omega(g(n))$ if there are positive constants c and $n_0$ such that $f(n) >= cg(n)$ for all $n >= n_0$. This notation is known as Big-Omega notation


$c_2g(n)$
$f(n)$
$c_1g(n)$
$n$
$n_0$
$f(n) = \Theta(g(n))$
(a)
$cg(n)$
$f(n)$
$n$
$n_0$
$f(n) = O(g(n))$
(b)
$f(n)$
$cg(n)$
$n$
$n_0$
$f(n) = \Omega(g(n))$
(c)

# Big Theta Notation

$f(n) = \Theta(g(n))$ if there are positive constants $c_1$, $c_2$ and $n_0$ such that $c_1 g(n) <= f(n) <= c_2 g(n)$, for all $n >= n_0$. This notation is known as Big-Theta notation

$20n^2+17n+9$ belongs to $\Theta(n^2)$

$8n+2$ does not belong to $\Theta(n^2)$

$n^3$ does not belong to the $\Theta(n^2)$

Shrinking lower and upper bounds is a area of research

Instead of an instance we are giving T(n) for a function of n as a time to run an algorithm or goodness of an algorithm

$\Theta(g(n))=O(g(n)) \cap \Omega(g(n))$

Notions of upper bound and lower bound and the tightness of these bounds

Lack of the actual T(n) leads us to expressing these terms in terms of Oh or Omega notation