Time complexity

Iterative and recursive algorithms

How to measure running time of an algorithm?

- Experimental study –
  - **Implement** the algorithm in a **programming language**
  - **Run** it with **different input sets**
  - Use **system time** (clock() function, time(), data()) to **measure** actual running **time**.

- Drawbacks –
  - **Implementation** of algorithm in a **preferred language** – time required
  - Only **finite input sets can be verified** – Not all input sizes are considered
  - For **comparing** two **algorithms same hardware** and **software** is required

Algorithm analysis

- Use **high-level description** of the algorithm **instead** of **testing its implementations.**
- Consider **all possible inputs**
- **Analysis** algorithm running time **irrespective** of **software and hardware requirements**.

Pseudo-code

- **Pseudocode** is an informal high-level description of the operating principle of a computer program or other algorithm.

- It uses the **structural conventions** of a normal programming language, but is intended for **human reading** rather than machine reading.

- Pseudocode typically omits details that are essential for machine understanding of the algorithm, such as variable declarations, system-specific code and some subroutines.

- No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program.

Pseudo-code

- Decision structures – **If .. Else .. End**
- While loop – **While …. End**
- For loop – **For … End**
- Array indexing – **A[i] .. A[i, j]**
- Methods – **methodname(Arguments)**

Time complexity – Big Oh notation

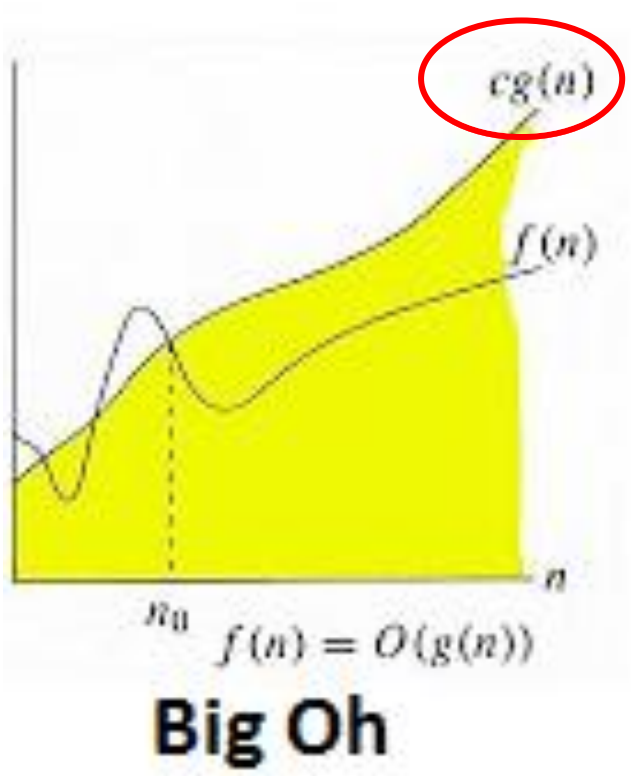
- Total **time** required by the **program** to run till its **completion**.

- It is estimated by **counting** the **number** of **elementary** steps **performed** by any **algorithm(code)** to **finish execution**.

- **Algorithm's** (**code**) **performance** varies with different types of input data.

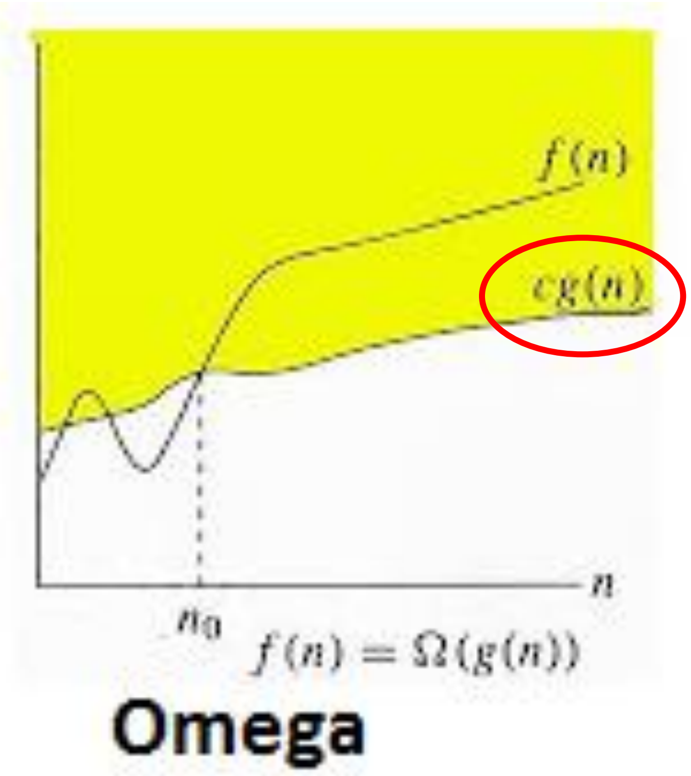
- Usually, the **worst-case time complexity** of an algorithm is of interest.
  - **Maximum time** taken for any input size.

WORST CASE



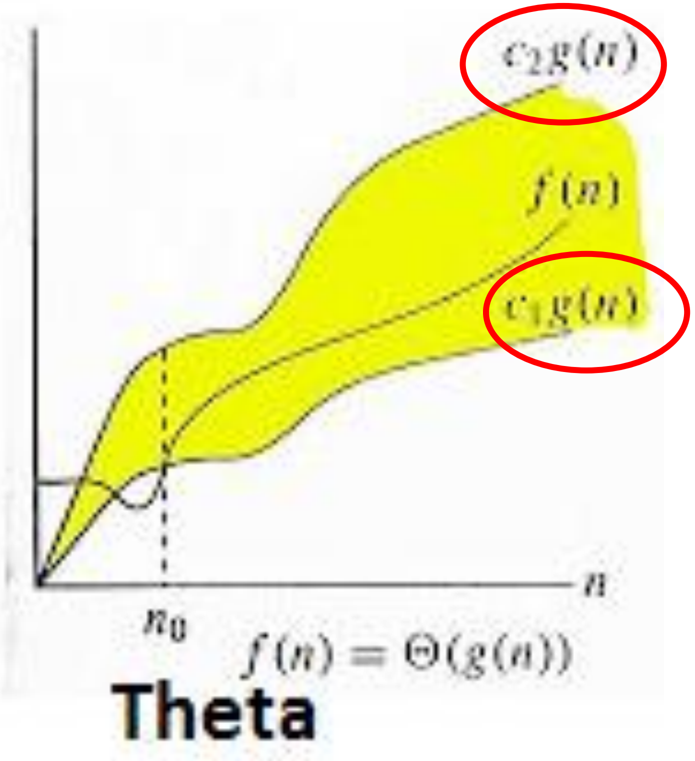
Worst case : Input which takes long time or algorithm is slower

BEST CASE



Best case : Input for which algorithm takes lowest time or algorithm is faster

AVERAGE CASE



Average case : Predicts running time of algorithm for a random input

Order of growth

- The rate at which the running time increases as a function of input is called “**Order of growth**”

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log n)**	**O(n)**	**O(n log n)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Question

- Write an algorithm to find a factorial of a number.
- Express its running time in terms of input size.

Iterative and recursive algorithms

Iterative

- Factorial
- For i=1:n
  - Fact = Fact * i;
- End
- Return Fact

Recursive

- **Factorial** (n)
- If n==0
  - Return 1
- Else
  - Return n***Factorial**(n-1)
- End

Iterative and recursive algorithms

Iterative

- keep repeating until a task is "done"

Recursive

- Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.

Which is Better? **No clear answer, but there are known trade-offs.**

Iterative and recursive algorithms

- Which approach to choose? Depends on the problem

- Algorithms with Abstract Data Types (ex: Trees) can be easily implemented **recursively**.
-
- “Mathematicians” often prefer recursive approach.
  - Solutions often shorter
  - Good recursive solutions may be more difficult to design and test.
-
- “Programmers”, often prefer iterative solutions.
  - Easy to implement
  - Control stays local to loop

Master theorem for subtract and conquer recurrences

- If the recurrence is of the form $T(n) = aT(n-b) + O(n^K)$
- $T(n) = O(n^K)$ if $a<1$
- $T(n) = O(n^{K+1})$ if $a=1$
- $T(n) = O(n^K a^{n/b})$, if $a>1$

Recursive linear search

- Search(A,i,x)
- If A[i] == x
  - Return i
- Else
  - Return Search(A,i+1,x)
- End

- Time complexity – O(n)

Solution :
$T(n) = O(1) + T(n-1)$
$= O(n^0) + T(n-1)$

a=1 K=0

$T(n) = O(n^{K+1})$ if a=1
$=O(n)$

Question

- What is the time complexity of the following code?
- Function(n)
- If n<= 1
  - Return
- End
- For i=1:3
  - Function(n-1)
- End

Solution :

$T(n) = O(1) + 3T(n-1)$

$= O(n^0) + 3T(n-1)$

a=3 K=0 b=1

$T(n) = O(n^K a^{n/b})$, if a>1

$=O(n^0\ 3^n) = O(3^n)$

Guidelines for algorithm analysis

- Loops – O(n)
- Nested loops – Total running time is product of sizes of all the loops
- Consecutive statements – Add the time complexities of each statement
- If-then-else : Worst-case running time of either ‘then’ part or the ‘else’ part (whichever is the larger)

Logarithmic complexity

- An algorithm is O(log n) if it takes a constant time to divide the problem size by a fraction.
- Example :
- For i=1:n
  - i=i*2
- End
- Let us assume loop ends after ‘k’ times that is $2^k=n$
- $n = 2^k$ therefore $k = \log n$

Master Theorem for Divide and Conquer

- If the recurrence is of the form
- $T(n) = a\,T(n/b) + O(n^K)$, where $a \geq 1$, $b > 1$, $K \geq 0$ then

- If $a < b^K$, $T(n) = O(n^K)$

- If $a = b^K$, $T(n) = O(n^K \log n)$

- If $a > b^K$, $T(n) = O(n^{\wedge}(\log_b a))$