Algorithms and Complexity (Lecture I)

Introduction

- Frequently, there may be a number of algorithms to solve the same problem. The question arises as to which one should be used. Is one algorithm better than another? This is a non-trivial question and leads to the analysis of algorithms and the means by which they can be compared. This area of study is sometimes called Complexity Theory in Computer Science.
- One way to compare algorithms is to compare their performance in terms of how quickly they solve the problem. Some algorithms arrive at a solution faster then others. We often choose the fastest algorithm when solving a problem.

- Another way of comparing algorithms is to look at the amount of space (memory) they require Some algorithms require large amounts of space but arrive at a solution quickly. Others require small amounts of space but arrive at a solution less quickly.
- You are often presented with a space/time trade-off. You will find time and time again in comparing algorithms and computing systems (and in many aspects of life!) that you have a trade-off situation. Improvement in one aspect leads to degradation in another eg Fast algorithm may require a lot of space Slow algorithm may require small amount of space

Analysis of Algorithms: Efficiency of an algorithm can be measured in terms of

- Execution time (time complexity)
- The amount of memory required (space complexity)

Time Complexity

The actual absolute time taken to solve a problem depends on a number of factors:

- how fast the computer is
- RAM capacity of the computer
- OS the computer uses
- quality of code generated by the compiler
- etc...

Time complexity: A measure of the amount of time required to execute an algorithm

- Analysis is based on the amount of work done by the algorithm
- Time complexity expresses the relationship between the size of the input and the run time for the algorithm is

- One reason for computing complexity is to compare algorithms. We need a measure which will allow us compare two algorithms. Each algorithm consists of a finite sequence of instructions. The more instructions in an algorithm the longer it will take to execute.
- One way to compare algorithms would be to count the instructions that the algorithm requires to solve a problem.
- The number of instructions will vary depending on the input. A payroll program for 100 people will repeat more instructions than one for 10 people.
- Thus we compute the number of instructions as a function of the input size.

Simplified analysis can be based on:

- Number of arithmetic operations performed
- Number of comparisons made
- Number of times through a critical loop
- Number of array elements accessed
- etc

Big-O Notation

- Big-O Notation Examples:
- $1 = \mathrm{O(n)}$
- $n = O(n^2)$
- $\log(\mathrm{n}) = \mathrm{O(n)}$
- $2\ \mathrm{n} + 1 = \mathrm{O(n)}$

Constant Time: O(1)

Constant Time: O(1)
An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: O(n)

Linear Time: O(n) =¿ An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- Array List: contains method
- queue: contains method

Logarithm Time

- Logarithmic Time: O(log n)
- An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size.
- Example: binary search

Quadratic Time

- Quadratic Time: $O(n^2)$
- An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size.
- Examples: bubble sort, selection sort, insertion sort

Definition of "big Omega"

We need the notation for the lower bound. A capital omega notation is used in this case. We say that f(n) = (g(n)) when there exist constant c that f(n) c*g(n) for all sufficiently large n.
Examples

- n = (1)
- $n^2 = (n)$
- $n^2 = (nlog(n))$
- 2 n + 1 = O(n)

Definition of "big Theta"

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that f(n) = (g(n)), if and only f(n) = O(g(n)) and f(n) = (g(n)). Examples

- 2 n = (n)
- $n^2 + 2n + 1 = (n^2)$

Big-O Analysis in General

- A computer executes a million instructions a second.
- The chart below summarizes the amount of time required to execute f(n)instructions on a machine for various values of n.

Table: **Big-O Analysis in General.**

$f(n)$	$n = 10^3$	$n = 10^5$	$n = 10^6$
$log_2(n)$	10^{-5}sec	$1.7 * 10^{-5}$ sec	$2 * 10^{-5}$ sec
n	10^{-3} sec	0.1 sec	1 sec
$n * log_2(n)$	0.01 sec	1.7 sec	20 sec
n^2	1 sec	3 hr	12 days
n^3	17 min	32 yr	317 centuries
2^n	10^{285} centuries	10^{10000} yr	10^{10000}yr

Intractable problems

- A problem is said to be intractable if solving it by computer is impractical
- Example: Algorithms with time complexity O(2n)take too long to solve even for moderate values of n; a machine that executes 100 million instructions per second can execute 260 instructions in about 365 years

Space Complexity

Space Complexity is concerned with the use of main memory (often RAM) while the algorithm is being carried out. As for time analysis above, analyse the algorithm, typically using space complexity analysis to get an estimate of the run-time memory needed as a function as the size of the input data. The result is normally expressed using Big O notation.

Memory Usage

There are up to four aspects of memory usage to consider:

- The amount of memory needed to hold the code for the algorithm.
- The amount of memory needed for the input data.
- The amount of memory needed for any output data (some algorithms, such as sorting, often just rearrange the input data and don't need any space for output data).
- The amount of memory needed as working space during the calculation (this includes both named variables and any stack space needed by routines called during the calculation; this stack space can be significant for algorithms which use recursive techniques).

Categories of Memory

Current computers can have relatively large amounts of memory (possibly Gigabytes)with three different categories of memory.

- Cache memory (often static RAM) - this operates at speeds comparable with the CPU.
- Main physical memory (often dynamic RAM) - this operates somewhat slower than the CPU.
- Virtual memory (often on disk) - this gives the illusion of lots of memory, and operates thousands of times slower than RAM.

- An algorithm whose memory needs will fit in cache memory will be much faster than an algorithm which fits in main memory, which in turn will be very much faster than an algorithm which has to resort to virtual memory. To further complicate the issue, some systems have up to three levels of cache memory, with varying effective speeds. Different systems will have different amounts of these various types of memory, so the effect of algorithm memory needs can vary greatly from one system to another.
- In the early days of electronic computing, if an algorithm and its data wouldn't fit in main memory then the algorithm couldn't be used. Nowadays the use of virtual memory appears to provide lots of memory, but at the cost of performance. If an algorithm and its data will fit in cache memory, then very high speed can be obtained; in this case minimising space will also help minimise time. An algorithm which will not fit completely in cache memory but which exhibits locality of reference may perform reasonably well.