Algorithm Analysis

See: Koffman & Wolfgang section 2.6

Efficiency of algorithms

- Most algorithms are used to process many data items
  - For example, all the data items held in a data structure such as an array, a vector or a linked list.

- To analyse the efficiency of an algorithm, we would like to measure the amount of time it takes to run the program with many different amounts of data.

- For many problems, there are algorithms that are relatively obvious, but very inefficient
  - They may seem OK with the amount of data we use in a test, but would become totally impractical if the amount of data increased by even a small amount.
  - It may be quite fast to run the algorithm with a reasonably small amount of data (say 100000 items) but what happens when we double the amount of data items?
  - Will the time double?   Or quadruple?
  - Or increase exponentially?
    - which means a huge amount more time would be taken
  - Or just increase by a little bit?
  - Or not change at all?
  - Or something else? …

Efficiency of algorithms cont.

- Although it is quite hard to measure precisely how much time an algorithm takes to perform, we can analyse the algorithm to see how the execution time will change as the number of data items increases.

- So measuring the rate of change in execution time as the amount of data increases is fundamental to choosing an appropriate algorithm.
  - Of course, we would also have to check that the data structures used by the algorithms were within the bounds of the memory resources we have available to store them.
  - And maybe there are two algorithms which always increase at the same rate, but, within that, one may be always better than the other.

Algorithm Analysis – the Big-O notation

- We can analyse an algorithm to see how the execution time grows as the size of the input increases. Some examples:
- If the execution time is doubled when the number of inputs, n, is doubled, we can say that the algorithm grows at a linear rate.
  - The growth rate is directly proportional to n, or *has order n*

In Big-O notation, the algorithm is O(n)

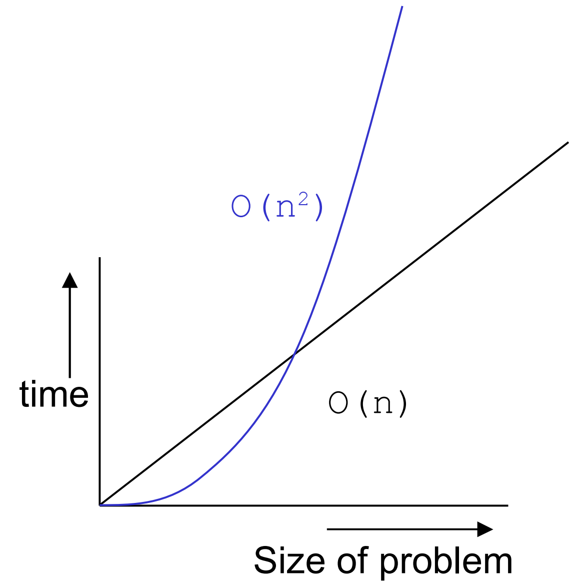
- But if the execution time is quadrupled every time the number of inputs, n, is doubled, then the algorithm grows at a quadratic rate
  - The growth rate is directly proportional to n^2, or *has order* n^2
  - This would obviously make the algorithm very much slower than a linear algorithm when n is very large!

In Big-O notation, the algorithm is $O(n^2)$

for example, if the algorithm involves a double loop, it might well be `O(n²)`

Comparing O(n) and O(n^2) algorithms

- An O(n) algorithm
  - If size of problem is multiplied by 1000, time taken is multiplied by 1000



- An O(n^2) algorithm
  - If size of problem is multiplied by 1000, time taken is multiplied by 1 million (1000000)!

- Suppose that the number of execution steps is related to the size of the problem like this:

  Number of steps = $3n^2 + 2n + 6$

  - When n gets very large, the no. of steps is far more affected by the n^2 factor than by the 2n (and the extra 6 steps is making no impact at all)
  - For our big-O analysis we disregard the factors that make least impact – this is still an O(n^2) algorithm.

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

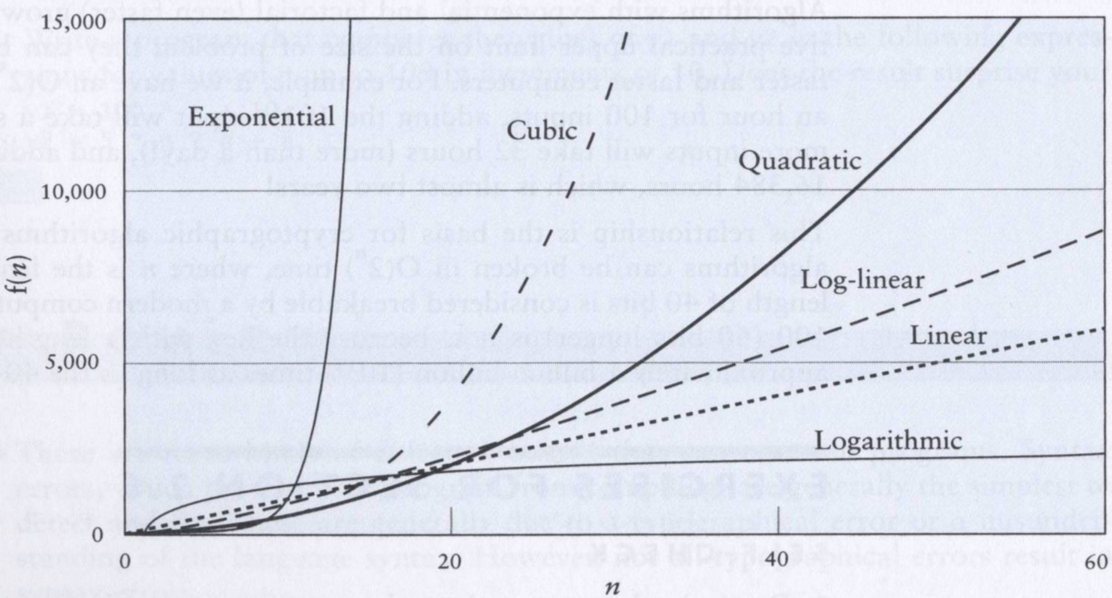
Some commonly found growth rates for algorithms

TABLE 2.5
Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

an O(1) algorithm is obviously the ultimate, but you can see that an O(log n) algorithm would grow very slowly as n increased, whereas an $O(2^n)$ algorithm would be a disaster!

FIGURE 2.8
Different Growth Rates



Best, worst and average cases

- For many algorithms, different inputs of the same size can give different results
- For example, consider searching an array of n integers to find one that matches a given key K.
- A sequential search will start at the first position in the array, and look at each value in turn until K is found.
- Each processing step is a comparison of an array entry with K
  - If K is found at the first entry (best-case) then only 1 processing step is needed
  - If K is found as the last entry, or not at all (the worst case), then all n entries have been looked at, so the number of processing steps is n.
  - But on running the algorithm many times with different data inputs, we would expect that on average it would search half way through the array to find K (half the time it would do better than this, and half the time it would do worse, but `n/2` is the *average* no of processing steps needed)
- Normally we would not base an algorithm on the best-case scenario
  - it would be very optimistic most of the time unless the best case has a high probability of occurring
- Often, the average case will be the most useful analysis
- What about the worst case?
  - We know that the algorithm must perform at least that well, which may be important for real-time applications
  - It is no good having an algorithm to use in an air traffic control system that can handle `n` aeroplanes efficiently *most of the time*, but spirals out of control in the worst case scenario!

- As we study various algorithms for the remainder of this course (and through SDEV6 next semester), we will analyse them using big-O notation.

- Our analysis may be quite informal (not using rigorous mathematics), but you should always be able to understand and explain the big-O analysis.

- Ultimately, you should be able to analyse algorithms yourself and use the analysis as a tool to help you choose appropriate algorithms for different situations.