COMP1004: Analysis of Algorithms

This part of the course deals with assessing the time-demand of algorithmic procedures with the aim, where possible, of finding efficient solutions to problems. We will not be considering issues related to demands on memory space; for those interested, these are dealt with for example one or other of the references below.

Background reading

The material in these books is supplementary to the notes – the books are not essential for this part of the 1004 course.

ALGORITHMICS: The Spirit of Computing - David Harel (Addison Wesley) *A very readable introduction to the subject, covering most of the areas dealt with in these lectures and also many further topics (some relating to later modules such as COMP3004) – highly recommended.*

INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS - Anany Levitin (Pearson International) *Clear and well written and about the right level. This course doesn't follow the book closely but uses a similar style of pseudocode and some of its examples.*

ALGORITHMS: Theory and Practice - Giles Brassard and Paul Bratley (Prentice-Hall) *Detailed mathematical treatment, goes much further than this course – recommended if you find the material interesting and want to learn more.*

1. INTRODUCTION

What is an algorithm?

An algorithm is a procedure composed of a sequence of well-defined steps, specified either in a natural language (a recipe can be regarded as an algorithm), or in appropriate code or *pseudocode*. In these lectures algorithms will be presented in a simplified pseudocode.

An algorithm is able to take a 'legal' input – eg for multiplication a pair of numbers is legal, but a pair of text files is not – carry out the specified sequence of steps, and deliver an output.

Algorithms are procedural solutions to problems.

Problems to which algorithmic solutions may be sought fall into four basic classes:

- Those that admit solutions that run in 'reasonable time' – the class of **tractable** problems (eg *sorting* and *searching*).
- Those that *probably* don't have reasonable-time algorithmic solutions (eg the *Travelling Salesman problem*).
- Those that *definitely* don't have such solutions (eg the *Towers of Hanoi*).
- Those that can't be solved algorithmically at all – the class of **non-computable** problems (eg the *halting problem*).

The last three of these will be the subject of later courses; this course will deal mainly with methods for evaluating the time-complexity of 'reasonable time' algorithms, for those tractable problems which do admit such solutions.

A tractable problem may however also have an algorithmic solution which does *not* have reasonable time demands. Often such an algorithm comes directly from the definition of the problem – a 'naïve' algorithm – but a more subtle approach will yield a more practically useful solution.

Here's an example of a problem which *is* tractable, but which has also has a naive algorithm that has a very fast growing time-demand that makes this algorithm useless for all but very small input instances.

Example: evaluating determinants

A determinant is a number that can be associated with a square matrix, used for example in the calculation of the inverse of a matrix (you will learn about matrices in the MATH6301 Discrete Mathematics course this term).

It has a recursive definition such that det(M), for an nxn (n rows and n columns) matrix, is a weighted sum of n determinants of (n-1)x(n-1) sub-matrices. These in turn can be expressed as the weighted sums of the determinants of (n-1) determinants of (n-2)x(n-2) matrices, and so on.

$$\det(M) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det(M^{(n)}[1,j]), \quad n>1$$
$$= a_{11}, \quad n=1$$

($M^{(n)}[i,j]$ is the (n-1)×(n-1) matrix formed by deleting the ith row and jth column of the n×n matrix M.)

This definition can be used as the basis for a simple algorithm, referred to here as the 'recursive algorithm'.

Using it for example for a 3x3 matrix is easy; the calculation takes only a few minutes.

For a 4x4 or 5x5 it begins to get messy and time-consuming...

...but for a 10x10 matrix, forget it. Unless you are exceptionally patient you would not want to do this by hand. And even using a computer it takes a significant amount of time.

What isn't immediately apparent from the 'to calculate the quantity for an instance of size n, calculate it for n instances of sizes n-1' recursive definition is how much work this implies.

The recursive algorithm takes a time **in the order of n!,** written **O(n!),** where n! = n×(n-1)×(n-2)...×3×2×1 (later in the course we will show this).

('In O(...)' means 'roughly like (...)' at an intuitive level – later we will formalise the definition.)

n! is an extremely fast-growing function, making it unfeasible to use the recursive algorithm for all but very small matrices ($n \leq 5$).

However there is an alternative algorithm for evaluating a determinant, based on *Gaussian elimination*, that only takes time in **$O(n^3)$.**

The difference between the time-demand of the two algorithms as the input size grows is startling:

size of matrix	**recursive algorithm**	**Gaussian elimination**
5 x 5	20 secs	
10 x 10	10 minutes	0.01 secs
20 x 20	>10 million years	
100 x 100	!!	5.5 secs

Two ways to approach analysis of algorithms:

Empirical: repeatedly run algorithm with different inputs – get some idea of behaviour on different sizes of input
→ can we be sure we have tested the algorithm on a sufficiently wide range of inputs?
→ this consumes the very resource (time) we are trying to conserve!

Theoretical: analysis of a 'paper' version of the algorithm
→ can deal with all cases (even impractically large input instances)
→ machine-independent

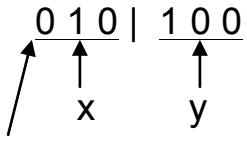
The aim is to obtain some measure of how the time demand of an algorithm grows with the size of its inputs, and to express the result in a simplified way, using order notation, so that the implications can be more easily visualized.

Time complexity function	**Problem size n**			
	10	**10^2**	**10^3**	**10^4**
$\log_2 n$	3.3	6.6	10	13.3
n	10	100	10^3	10^4
$n \log_2 n$	33	700	10^4	1.3×10^5
n^2	100	10^4	10^6	10^8
n^3	1000	10^6	10^9	10^{12}
2^n	1024	1.3×10^{30}	$>10^{100}$	$>10^{100}$
n!	3×10^6	$>10^{100}$	$>10^{100}$	$>10^{100}$

Measuring 'size of an instance'

Formally, the size |x| of an input instance x is the number of bits needed to encode it, using some easily-decoded format.

e.g. for multiplying 2 numbers x & y



smaller number padded
with leading zeros

'Size of input' = $2 \times \max (1+\lfloor \log_2 x \rfloor, 1+\lfloor \log_2 y \rfloor)$

[We will use the functions

ceiling(x) = $\lceil x \rceil$ = smallest integer $\geq x$,

floor(x) = $\lfloor x \rfloor$ = largest integer $\leq x$]

But normally a much more informal definition is used which depends on the context, eg

problem	'size of an input instance'
sorting	number of items to be sorted
calculating a determinant	number of rows and columns in matrix
finding a minimal spanning tree	number of nodes in the graph

Measuring 'time taken'

The objective is to make the time-cost analysis machine-independent. The difference between running the same algorithm on two different machines is only going to be some constant factor (eg. "this machine is twice as fast as that one") which is the same for all input sizes.

The kind of difference that really counts is the sort that itself increases with size – the difference between n log n and n^2, or between n^3 and n!.

A machine-independent measure of time is given by counting **elementary operations.**

These are simple operations used as primitives by all the candidate algorithms – for example when we say that the cost of a sorting algorithm "grows like n^2" we will usually be counting the number of comparisons done as a function of n, the number of things to be sorted.

Other operations that can be used as 'elementary' time-counters are Boolean operations (AND, OR, etc.), assignments, and mathematical operations such as addition, subtraction, multiplication and division.

Elementary operations are considered to themselves be of negligible cost, and are sometimes – for simplicity – referred to as being of 'unit cost' or taking 'unit time'.

Note: operations which are 'primitive' and considered to be of trivial intrinsic cost – so that they can be used as time-counters – in some contexts may not be so lightly dismissed in others.

(For example multiplying 2 numbers can almost always be thought of as an elementary operation, but there are some applications, such as cryptology, using very large numbers (100s or 1000s of decimal digits), where the cost of multiplication is *not* trivial! Then multiplication itself needs to be broken down into simpler operations (single-bit ones) and better algorithms (like Strassen's algorithm – see later) looked for.)

Forms of time-complexity analysis

Worst case

This is the easiest form of analysis and provides an upper bound on the efficiency of an algorithm (appropriate when it is necessary to ensure a system will respond fast enough under *all* possible circumstances- eg. controlling a nuclear power plant)

Average case

There may be situations where we are prepared to put up with bad performance on a small proportion of inputs if the 'average performance' is favourable.

What does 'average performance' mean?

Either sum the times required for every instance of a particular size, divide by the number of instances, *or* evaluate performance with respect to an 'average instance'. (For a sorting algorithm this might well be a randomly ordered file- but what is an average instance for a program processing English-language text?)

Average case analysis is mathematically much more difficult- many algorithms exist for which no such analysis has been possible.

Best case

This kind of analysis differs from the other two in that we consider not the algorithm but the **problem itself**. It should really be referred to as **'best worst case'** analysis, because we aim to arrive at bounds on the performance of **all possible algorithmic solutions**, assuming their worst cases.

Best case analysis is based on an consideration of the logical demands of the problem at hand – what is the very minimum that any algorithm to solve this problem would need to do, in the worst case, for an input of size n?

Example: Consider the multiplication of two n-bit numbers. Any algorithm to solve this problem must at least *look* at each bit of each number, in the worst case – since otherwise we would be assuming that the product could in general be independent of some of the 2n bits – and so we can conclude that multiplication is bounded below by a **linear function of n**.

Order Notation

The result of a time-complexity analysis may be some long and complicated function which describes the way that time-demand grows with input size.

What we really want to know is how, roughly, these time-demand functions behave – like n? log n? n^3?

The objective of using order notation is to simplify results of complexity analysis so that the overall shape – and in particular, the behaviour as $n \to \infty$ (**asymptotic behaviour**) – of the time-demand functions are more clearly apparent.

O-notation

'O' can provide an **upper bound** to time-demand in either **worst or average cases**.

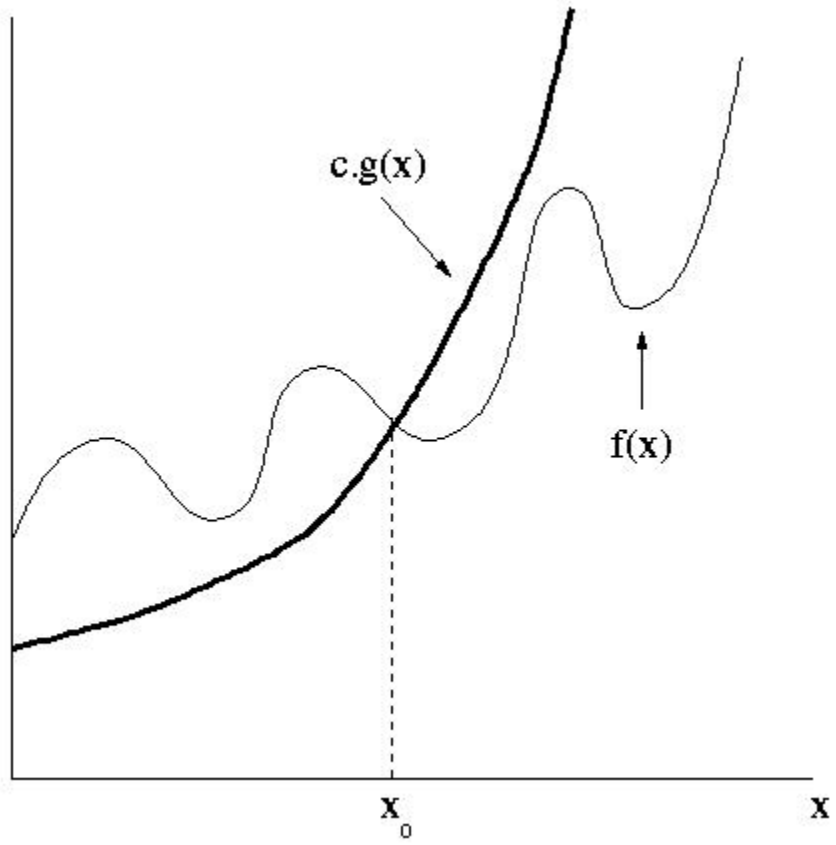
Intuitively, 'f(x) is O(g(x))' means that f(x) grows no faster than g(x) as x gets larger.

Formally,

The positive-valued function f(x) $\in$O(g(x)) if and only if there is a value x_0 and a constant c>0 such that

$$\text{for all } x \geq x_0, \quad f(x) \leq c.g(x)$$

(Note: the restriction in the definition here – for simplicity – that f(x) be 'positive-valued' isn't likely to cause problems in algorithmic applications since functions will represent 'work done' and so will always return positive values in practice.)



Useful properties of ‘O’

1. **O(k.f(n)) = O(f(n)), for any constant k**

This is because multiplication by a constant just corresponds to a re-adjustment of the value of the arbitrary constant ‘k’ in the definition of ‘O’.

This means that under O-notation, we can forget constant factors (though these ‘hidden constants’ might be important in practice, the don’t change the *order* of the result).

Note that as a consequence of this, since $\log_a n = \log_a b \times \log_b n$ there is no effective difference between logarithmic bases under O-notation; conventionally we just use O(log n), forgetting the (irrelevant) base.

2. **O(f(n) + g(n)) = O(max(f(n),g(n)))**
(for those interested the proof is on p.56 of Levitin)

'max' here is a shorthand way of saying 'the part that grows the fastest as $n \to \infty$'. This result enables us to simplify the result of a complexity analysis, for example

$$n^3 + 3n^2 + n + 8 \in O(n^3 + (3n^2 + n + 8))$$
$$= O(\max(n^3, 3n^2 + n + 8)) = O(n^3)$$

3. O(f(n)) U O(g(n)) = O(f(n) + g(n))
(not so easy to prove!)
= O(max(f(n),g(n))), by 2. above.

This last means that where an algorithm consists of a *sequence* of procedures, of different time-complexities, the *overall* complexity is just that of the most time-demanding part.

Examples of proofs using O-notation

[Note: You can assume in all such proofs that n>0, as in this course n will represent 'size of an input'.]

For example, is it true that

(i) $n^2 \in O(n^3)$?
(ii) $n^3 \in O(n^2)$?

The general way to proceed is as follows:

- Assume the assertion is true.
- Work from the definition of 'O', and try to find suitable values of c and n_o
- If you can find ***any*** pair of values (there's no *unique* pair) the assertion *is*, in fact, ***true***. If there is some fundamental reason why ***no*** pair c,n_0 could be found, then the original hypothesis was wrong and the assertion is ***false***.

(i) Is $n^2 \in O(n^3)$**?**

Assume it is. Then

$$\begin{aligned} & n^2 - cn^3 \le 0, \quad \text{for all } n \ge n_0 \\ \Rightarrow\ & n^2(1 - cn) \le 0, \quad \text{for all } n \ge n_0 \\ \Rightarrow\ & cn \ge 1, \quad \text{for all } n \ge n_0 \\ \Rightarrow\ & n \ge \frac{1}{c}, \quad \text{for all } n \ge n_0 \end{aligned}$$

Choosing (for example) c=2, n_0=1 is satisfactory and so it's TRUE that $n^2 \in O(n^3)$.

(ii) Is $n^3 \in O(n^2)$**?**

Again, assume it is. Then

$$\begin{aligned} & n^3 - cn^2 \le 0, \quad \text{for all } n \ge n_0 \\ \Rightarrow\ & n^2(n - c) \le 0, \quad \text{for all } n \ge n_0 \\ \Rightarrow\ & n - c \le 0, \quad \text{for all } n \ge n_0 \end{aligned}$$

But c has to have a *fixed* value. There is no way to satisfy n≤c, for all $n \ge n_0$ for a fixed c. Hence the original assumption was FALSE, and $n^3 \notin O(n^2)$.

Notes

- When answering the question 'Is $f(n) \in O(g(n))$?' it is not sufficient to draw a picture showing the curves f(n) and g(n) -- that can illustrate your argument, but isn't in itself a proof, as the question is about what happens as $n \to \infty$, so can't be resolved by looking at any finite range of n.
- If you are asked to base a proof on 'the formal definition of O-notation' don't base your argument on the three properties listed on pp.10-11. Argue from the definition of O-notation, as above.

Hierarchies of complexity

Let n be the 'size' of an input instance, in the usual informal definition (eg degree of a polynomial, length of a file to be sorted or searched, number of nodes in a graph).

Complexity	
O(1)	**Constant time**: all instructions are executed a fixed number of times, regardless of the size of the input. Example: taking the head of a list.
O(log n)	**Logarithmic**: program gets only slightly slower as n grows (typically by using some transformation that progressively cuts down the size of the problem). Example: binary search.
O(n)	**Linear**: a constant amount of processing is done on each input element. Example: searching an unordered list.
O(n log n)	Typical of **'divide and conquer'** algorithms, where the problem is solved by breaking it up into smaller subproblems, solving them independently, then combining the solutions. Example: quicksort.
$O(n^k)$	**Polynomial**: most often arises from the presence of k nested loops (examples: insertionsort (k=2); Gaussian elimination method for a calculating a determinant (k=3).
$O(a^n)$	**Exponential**: very fast-growing (assuming a>1), essentially unusable for all but very small instances. Example: Towers of Hanoi (a=2).
$O(n!)$	**Factorial**: even worse! Example: recursive evaluation of a determinant.

Only algorithms running in *polynomial time* (those which are in $O(n^k)$ for some k) are effectively usable; only problems which admit such algorithms are effectively soluble (tractable). Thus finding a determinant is soluble in reasonable time because it has a 'good' algorithm running in $O(n^3)$ as well as an unusable O(n!) one, but the Towers of Hanoi puzzle isn't, because it can be demonstrated that there are *no* 'good' algorithms possible in this case. (More about such *intractable problems* in the 3rd year COMP3004 Computational Complexity course.)

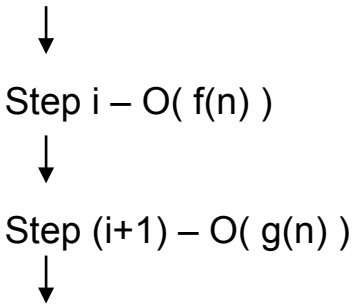
2. ANALYSIS OF NONRECURSIVE ALGORITHMS

There is not a clear set of rules by which algorithms can be analysed. There are, however, a number of techniques which appear again and again. The best way to learn about these is really though examples.

Algorithms consist of **sequences** of procedural steps which may themselves involve **loops** of fixed ('for...') or indeterminate ('while…', etc) length, or of **recursive function calls** (see later).

We will start with the simplest cases:

SEQUENTIAL OPERATIONS



The combination of the ith and (i+1)st steps takes a time in O(f(n)) U O(g(n)).

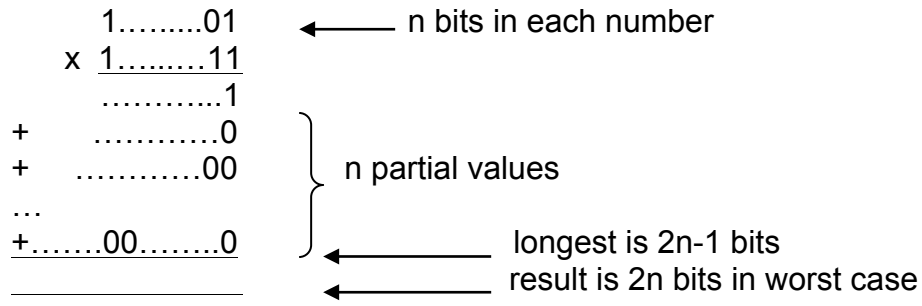
Use

$$O(f(n)) \cup O(g(n)) = O(\max(f(n), g(n)))$$

to justify neglecting all but the most time-costly step.

Example: multiplication

(i) Shift-and-add multiplication ("long multiplication")



(1) Compute n partial values, each requiring n single-bit multiplications
(2) Add the partial values (estimate as (n-1) additions of pairs of (2n-1)-bit numbers (upper bound))

Complexity (single-bit operations)
Step(1) n^2
Step(2) $(2n-1)(n-1) = 2n^2-3n+1$
Total $3n^2 - 3n+1 \in \mathbf{O(n^2)}$

n=5 example

```
          1 0 0 1 1  (19)
        × 0 1 0 1 1  (11)
0 0 0 0 1 0 0 1 1
0 0 0 1 0 0 1 1 0
0 0 0 0 0 0 0 0 0  5x5=25 single bit multiplications
0 1 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 1  having added top two rows (9 additions)
0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 1  having added top two rows (9 additions)
0 1 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 1  having added top two rows (9 additions)
0 0 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 1  having added top two rows (9 additions)

Total number of single bit operations =
25 + 9 + 9 + 9 + 9 = 61 (= 3x5^2-3x5+1)
Ans = 011010001
    = 0x256+1x128+1x64+0x32+1x16+0x8+0x4+0x2+1x1 = 209
```

(ii) A la russe method

The idea is to start with two columns (where '/' in the first column means *integer division*, ie dropping the remainder):

a	×	b
a/2		2b
a/4		4b
•		•
•		•
1		•

Create a third column, containing a copy of the number from the second column everywhere the number in the first column is odd. Add up this third column to get the result.

eg				
	19	×	11	11
	9	×	22	+22
	4	×	44	
	2	×	88	
	1	×	176	+176
				=209

There are O(n) entries in the columns, each involving work O(1), since each entry is made by either a right-shift (left column) or by adding a zero (right column). Adding the third column is $O(n^2)$. So 'à la russe' is also **$O(n^2)$** overall – but it's slightly faster than shift-and-add multiplication because it is still only O(n) before the addition stage.

Lower bound on the time-complexity of multiplication

We argued earlier that *every* algorithm for multiplying two n-bit numbers will require at least 2n single-bit operations, so has a best worst case in O(n). There is thus scope for algorithms whose performance improves on that of the simple $O(n^2)$ ones above and has a worst-case performance somewhere in between O(n) and $O(n^2)$ – we will see one such algorithm later, Strassen's algorithm, which has a worst-case performance in $O(n^{1.59...})$.

Sums of series

Before moving on to look at algorithms with loops and recursion, we need to know how to evaluate sums of series.

Arithmetic series

The notation

$$\sum_{i=a}^{b} f(i)$$

means

f(a)+f(a+1)+……+f(b).

Note we can use the notation even when f is independent of i:

$\sum_{i=a}^{b} c$ means $(c+c+\ldots+c) = c(b-a+1)$

!!! ➡ **b-a+1** times

The simplest – and most useful – case is when f(i) = i. In this case it is easy to derive a formula (a 'closed form' which does not have the summation symbol) for $\sum_{i=a}^{b} f(i)$**:**

$$2 \times \sum_{i=1}^{n} i = \quad 1 \; + \; 2 \; + \ldots + (n-1) + n$$

$$+ \; n \; + (n-1) + \ldots + \; 2 \; + 1$$

$$= (n+1) + (n+1) + \ldots + (n+1) + (n+1)$$

(n copies)

$$= n(n+1)$$

$$\rightarrow \quad \sum_{i=1}^{n} i = \frac{n}{2}(n+1)$$

It's sometimes the case that the sum to be evaluated doesn't start with i=1:

$$\sum_{i=j}^{n} i = \sum_{i=1}^{n} i - \sum_{i=1}^{j-1} i$$
$$= \frac{n}{2}(n+1) - \frac{(j-1)}{2} j$$

Geometric series

This is the other type of simple series summation which is very useful in algorithmics.

Let $S(n) = a + a^2 + a^3 + \ldots + a^n = \sum_{i=1}^{n} a^i$

$\rightarrow$ $a.S(n) = a^2 + a^3 + a^4 + \ldots + a^{n+1}$
$= S(n) - a + a^{n+1}$

So

$$\sum_{i=1}^{n} a^i = \frac{a(1-a^n)}{1-a} \quad \text{for } a \neq 1$$

Note that the formula works *only* for $a \neq 1$ - if a=1 get sum 0/0, which gives an undefined value. The calculation has to be done differently in this case:

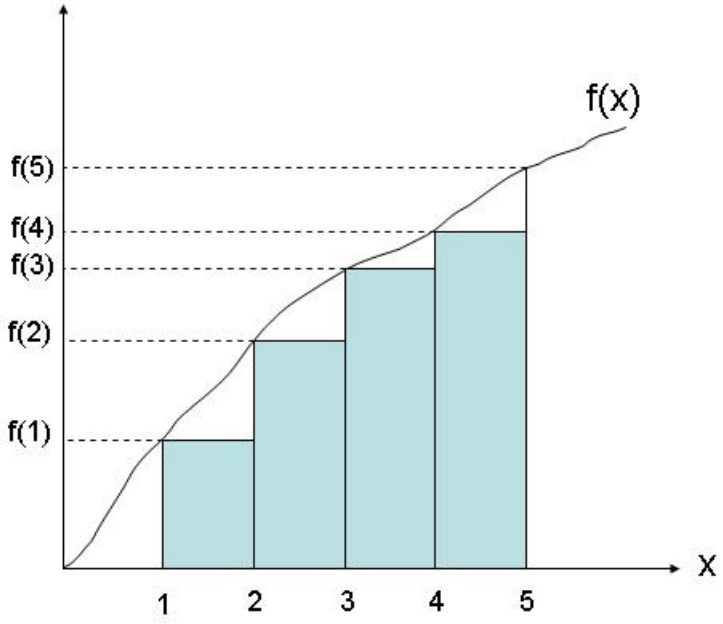
If a = 1:

$$\sum_{i=1}^{n} a^i = \sum_{i=1}^{n} 1 = (1 + 1 + \ldots + 1) = n$$
(n terms)

Estimating a sum by an integral

In real-life algorithmic analyses it's often the case that the series to be summed is not of one of the above simple forms, or any other for which a sum formula can be easily found. (The average-case analysis of Quicksort – considered later – is an example of this sort.) It's then necessary to **estimate** the sum.

Assume that where a sum from a..b is required that the function f(x) is non-decreasing between x=a and x=b+1.



The area in the boxes (where in the example a=1, b=4) is the desired sum $\sum_{i=1}^{4} f(i)$.

It can be seen from the illustration that this area is *not greater than* the area under the curve from x=1 to x=5.

In general using this graphical argument

$$\sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)\ dx$$

It can also be shown that if the function f(x) is non-increasing between x=a-1 and x=b that

$$\sum_{i=a}^{b} f(i) \;\leq\; \int_{a-1}^{b} f(x)\, dx$$

Draw a similar picture and think about it.

(If you think this variant of the approximation is less relevant because algorithmic work functions are not expected to be uniformly decreasing, consider that *components* of them might still behave this way -- see Gaussian elimination example later.)

Example: $\sum_{i=1}^{n} i^2 = ?$

Use $\sum_{i=1}^{n} i^2 \;\leq\; \int_{1}^{n+1} x^2 dx$

$$= \left[\frac{x^3}{3}\right]_1^{n+1} = \tfrac{1}{3}\left[(n+1)^3 - 1\right], \text{ and hence } \sum_{i=1}^{n} i^2 \in O(n^3)$$

(A more general argument along the same lines can be used to show that $\sum_{i=1}^{n} i^k \in O(n^{k+1})$)

ALGORITHMS WITH LOOPS

In the simplest cases where the loop is executed a fixed number of times (a ‘for’ loop) the complexity is just the cost of one pass through the loop multiplied by the number of iterations.

```
ALGORITHM Sum( A[0..n-1] )
// Outputs the sum of the elements in A[0..n-1]
sum <− 0
for i <− 0 to n-1 do
        sum <− sum + A[i]
return sum
```

There is only one choice here for the elementary operation, addition. There are **n** additions to 'sum' and hence this takes time (measured by the number of additions performed) in **O(n)**.

Compare with this example from Levitin, p.61:

```
ALGORITHM MaxElement( A[0..n-1] )
// Outputs the value of the largest element in A[0..n-1]
maxval <− A[0]
for i <− 1 to n-1 do
        if A[i] > maxval
                maxval <− A[i]
return maxval
```

In this case there are *two* operations in the 'for' loop, comparison and assignment, that might be candidates for the role of the elementary operation. However note the assignment is only done if the comparison returns true and hence it is the comparison that gives the best measure of the worst case cost of the algorithm.

As with the first example it's here easy to see the cost, in terms of the number of comparisons, must be **n-1** and hence this algorithms too is in **O(n)**.

More formally--

Let C(n) be the cost of executing MaxElement for an n-element array. Counting comparisons at unit cost

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \quad \in O(n)$$

* * *

If there are several nested loops of this type the complexity is the cost of one pass through the **innermost** loop multiplied by the **total** number of iterations. However for any given loop the amount work done may depend on the outer loop it is embedded in:

```
for i <− 1 to n do
        for j <− 1 to i do
                // something at unit cost
```

The cost of the work here is

$$C(n) = \sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} 1+1+1...+1$$

(1 added to itself i times)

$$= \sum_{i=1}^{n} i = \frac{n}{2}(n+1) \quad \text{(using familiar formula for sum-of-i)}$$

If you are asked to "simplify your answer using O-notation" you are being invited to use the three rules on pp.10-11. You can use them informally, you don't need to quote the rules but you should bear them in mind and make sure of your reasoning.

In this case since $O\left(\frac{n}{2}(n+1)\right)$ = O(max($n^2/2$, n/2))

= $O(n^2/2)$

we just keep the leading term, and since O(kf(n)) = O(f(n)) the 1/2 can be dropped to give the work again as **$O(n^2)$**.

(In this case of questions asking you to "prove that f(n) is in the order of g(n)" the word "prove" implies you need to use the formal definition of 'O' on p.9 and only this -- or another equally formally structured mathematical argument -- constitutes a full answer.)

Another example from Levitin (p.63):

```
ALGORITHM UniqueElements( A[0..n-1] )
// Returns true if all elements in A are distinct, false otherwise
for i <− 0 to n-2 do
        for j <− i+1 to n-1 do
                if A[i]=A[j] return false
return true
```

There is only one candidate for elementary operation, the test 'A[i]=A[j]?' There are two worst case situations: where the array contains distinct elements (all passes through the inner loop are executed, with the conditional evaluating true every time); and where only the last two elements A[n-2], A[n-1] are the same (as above but returns false on the very last test). In either of these situations the cost of the work in terms of the number of array element comparisons is

$$C(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1-(i+1)+1) = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1)\sum_{i=0}^{n-2} 1 \quad - \quad \sum_{i=1}^{n-2} i$$

(n-1) is a constant w.r.t. the sum

the contribution from i=0 is zero, so sum from i=1 for convenience

$$= (n-1)^2 \quad - \quad \frac{1}{2}(n-2)(n-1)$$

use the sum-over-i formula with the upper limit adjusted

$$= \frac{n}{2}(n-1) \qquad \rightarrow \quad C(n) \in O(n^2)$$

note this is the same as the number of distinct pairs should all n array elements be different

A last 'for' loop example, Gaussian elimination:

The recursive algorithm for calculating the determinant of a matrix is as discussed earlier unusable for all but very small instances, taking a time in O(n!).

However this quantity can be calculated very efficiently by first reducing the matrix to an *upper-triangular* form (all elements below the main diagonal are zero).

Once a matrix is in upper-triangular form its determinant is just the product of elements on the diagonal, and so can be evaluated with (n-1) multiplications, ie **O(n)** arithmetic operations.

The general form of the reduction algorithm for an n × n matrix is

```
ALGORITHM GaussianElimination
// Replaces matrix A[1..n,1..n] by an equivalent
// matrix in upper-triangular from
for i <− 2 to n
        for j <− 1 to i-1
                for k <− j to n
                        // subtract appropriately weighted row elements
```

In this i is the row, j the row above, k the column in which the next element is to be set to zero -- but don't worry about the details, this is just providing a further example for analysis.

Let the mathematical operations in the innermost loop cost one unit of time. Then the overall cost is given by

$$C(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} \sum_{k=j}^{n} 1$$

$$= \sum_{i=2}^{n} \sum_{j=1}^{i-1} (n + 1 - j)$$

split the sum into two parts, depending/not depending on j...

$$= \sum_{i=2}^{n}(n+1)(i-1) \quad - \quad \sum_{i=2}^{n}\frac{(i-1)}{2}i$$

first sum was (n+1) added to itself (i-1) times — *using summation formula with index i -> j and upper limit adjusted*

$$= \sum_{i=1}^{n}\left((n+1)(i-1) - \frac{i}{2}(i-1)\right)$$

contribution from i=1 is 0, so summing from i=1 makes no difference here

$$= \sum_{i=1}^{n}\left(-\frac{i^2}{2} + ni + \frac{3}{2}i - (n+1)\right)$$

The first two terms in the sum are expected to give contributions proportional to n^3. So can't we just say $C(n) \in O(n^3)$?

The two contributions are oppositely signed. In situations like this it is possible (though maybe not likely) they would cancel to give an overall rate of growth of lower order.

In any case it is an opportunity to practise using the summation approximation formulae on pp.19-20.

Since $-i^2$ is a non-increasing function for i = 0..n

$$\sum_{i=1}^{n} -i^2 \;\le\; \int_{0}^{n} -x^2\,dx \;=\; \left[\frac{-x^3}{3}\right]_0^n = \frac{-n^3}{3}$$

and hence

$$C(n) \;\le\; \frac{-n^3}{6} + n.\frac{n}{2}(n+1) + \sum_{i=1}^{n}\left(\frac{3}{2}i - (n+1)\right) \;=\; \frac{n^3}{3} + O(n^2)$$

contributions that are constants or that are linear or quadratic in n

Hence it is true that $C(n) \in O(n^3)$.

Aside:

There is in fact also a formula for sum-of-i^2 --

$$\sum_{i=1}^{n} i^2 = \frac{n}{6}(n+1)(2n+1)$$

-- though you don't need to memorise it -- and using this it can be shown there are exactly

$$\frac{n}{3}(n^2-1)$$

passes through the innermost loop, so again = $\frac{n^3}{3} + O(n^2) \in O(n^3)$