

COMP202 Complexity of Algorithms Introductory Lectures

- ▶ Lecturer: Russell Martin, Room 319 Ashton Building, email: `Russell.Martin (at) liverpool.ac.uk`
- ▶ Lectures: Monday (2pm), Thursday (11am) and Friday (3pm)
- ▶ Assessment: 2 class tests (20%) and final exam (80%)
- ▶ Course web page:
<http://www.csc.liv.ac.uk/~martin/teaching/comp202/>

Module Outcomes

The overall goal of this module is to train you to *think algorithmically*, to be able to analyze the correctness of algorithms and their performance in terms of the use of time and space, and to be able to think critically about algorithmics in general.

Module Outcomes (cont.)

More specifically, by the end of this module a student should

1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions;
2. have fluency with the structure and utilization of abstract data types such as stacks, queues, lists, and trees in conjunction with classical algorithmic problems such as searching, sorting, graph algorithms, etc;
3. have a comfortable knowledge of graphs, their role in representing various relationships, and be familiar with several of the algorithms for solving common problems such as finding minimum spanning trees, maximum flows, shortest paths, and matchings;

Module Outcomes (cont.)

4. understand the basic concepts from number theory that are used in “public-key” cryptography, and be able to explain the concepts that underlie the RSA encryption/decryption scheme;
5. be familiar with formal theories providing evidence that many important computational problems are inherently intractable, e.g., NP-completeness.

What is an algorithm?

- ▶ An *algorithm* is a sequence of steps for performing a task in a finite amount of time.
- ▶ What algorithms do you already know?

Course Texts

Primary: *Algorithm Design*, M.T. Goodrich and R. Tamassia (referred to as [GT] in these notes).

Secondary (reference): *Introduction to Algorithms*, T.H. Cormen, C. E. Leiserson and R.L. Rivest.

What is an algorithm? (cont.)

- ▶ For computational algorithms, we use *data structures* to store, access, and manipulate data.
- ▶ What data structures have you encountered in the past?

Our fundamental interest is to design “good” *algorithms* which utilize efficient *data structures*. In this context “good” (typically) means *fast* in a way to be clarified during this course.

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

Relevant questions

- ▶ How is the list stored?
- ▶ Is it sorted or not?
If the list is sorted, it's (often) easier to perform the search.
- ▶ If it's not sorted, should we sort it first?
- ▶ Are we going to be doing this search operation many times? (Then we might want to sort the list first, if necessary.)
- ▶ Are we going to be updating (i.e. adding/deleting items from) the list?

Experimental Analysis of Algorithms

- ▶ We are interested in *dependency* of the running time on *size* of the input.

The *size* could mean the number of vertices and edges if we are operating on a graph, the length of a message we're encoding/decoding, and/or the actual length of numbers we're processing in terms of the bits needed to store them in memory.

To analyze algorithms we might perform *experiments* to empirically observe the running time.

Doing so requires a good choice of sample inputs, and an appropriate number of tests (so that we can have *statistical certainty* about our analysis).

Algorithm Analysis

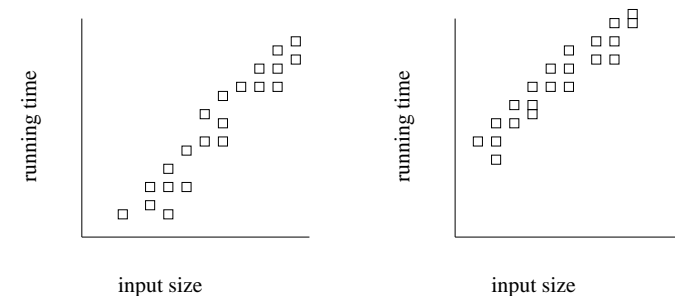
- ▶ **Primary interest:** Running time (*time-complexity*) of the algorithm and the operations on data structures.
- ▶ **Secondary interest:** Space (or "memory") usage (*space-complexity*).

This course focuses on the *mathematical* design and analysis of algorithms and their associated data structures, utilizing relevant mathematical tools to do so.

Let's first discuss the difference between *experimental analysis* and *theoretical analysis* of algorithms.

Experimental Analysis (cont.)

Also, the running time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.



Drawbacks of Experimental Analysis

1. Experiments are performed on a *limited set* of test inputs.
2. Requires all tests to be performed using *same hardware and software*.
3. Requires implementation and execution of algorithm.

Theoretical Analysis (cont.)

- ▶ When we analyze an algorithm in this fashion we aim to associate a function $f(n)$ to each algorithm, where $f(n)$ characterizes the running-time in terms of some measure of the *input-size*, n .
(For example, n could be the number of items to be sorted in a list.)

Typical functions include: $n, \log n, n^2, n \log n, n^5, 2^n, \dots$

Theoretical Analysis

Benefits over experimental analysis:

1. Can take all possible inputs into account.
2. Can compare efficiency of two (or more) algorithms, independent of hardware/software environment.
3. Involves studying high-level descriptions of algorithms (e.g. *pseudo-code*).

Theoretical Analysis (requirements)

For formal theoretical analysis, we need:

1. A *language* to describe algorithms.
2. *Computational model* in which algorithms are executed.
3. *Metric* for measuring running-time.
4. A way of characterizing running-time.

Pseudo-code

- ▶ Pseudo-code is a high-level description language for algorithms.

Pseudo-code provides more structured description of algorithms.

Allows high-level analysis of algorithms to determine their running time (and space requirements).

Pseudo-code (cont.)

- ▶ Pseudo-code is a mixture of natural language and high-level programming language (e.g., Java, C, etc.).

Describes a generic implementation of data structures or algorithms.

- ▶ Pseudo-code includes: *expressions, declarations, variable initialization and assignments, conditionals, loops, arrays, method calls*, etc.

I won't formally define a strict method for giving pseudo-code, but when using it we aim to describe an algorithm in a manner that would allow a competent programmer to translate the pseudo-code into program code *without misinterpretation* of the algorithm.

Pseudo-code example

Finding the *minimum element* of a set of numbers, stored in an array *A*.

Minimum-Element(*A*)

```
    /** Find the minimum of elements in the array A. */
1  min ← A[1]
2  for j ← 2 to length[A]
3      do
4          ▷ Compare element A[j] to current min
5          ▷ and store it if it's smaller.
6          if A[j] < min
7              then min ← A[j]
8  return min
```

Computational Model

Our computational model is like that typically found on modern-day computers.

- ▶ CPU connected to a bank of *memory cells*.
- ▶ Each memory cell can store a number, character, or address.

We use a set of high-level *primitive operations*: *assignment, method invocation, arithmetic operations, comparison of numbers, array indexing, returning from a method*, etc.

Computational Model (cont.)

Assumption: primitive operations (like a single addition, multiplication, or comparison) require *constant time* to perform. (Not necessarily true, e.g. multiplication takes about three to four times as long to perform on a computer than addition.)

- ▶ When we analyze the *running time* of an algorithm we *count* the number of operations executed during the course of the algorithm.

Counting Primitive Operations

Minimum-Element(A)

```
1  $min \leftarrow A[1]$ 
2 for  $j \leftarrow 2$  to  $length[A]$ 
3   do
4     ▷ Compare element  $A[j]$  to current  $min$ 
5     ▷ and store it if it's smaller.
6     if  $A[j] < min$ 
7       then  $min \leftarrow A[j]$ 
8 return  $min$ 
```

How many primitive operations?

1. One assignment in line 1;
 2. $length[A] - 1$ more assignments (to j) in line 2;
 3. Line 6 gives $length[A] - 1$ comparisons;
 4. A possible $length[A] - 1$ more assignments from line 7;
 5. Finally there's the last statement to execute once;
- ⇒ A maximum of $3(length[A] - 1) + 2$ primitive operations.

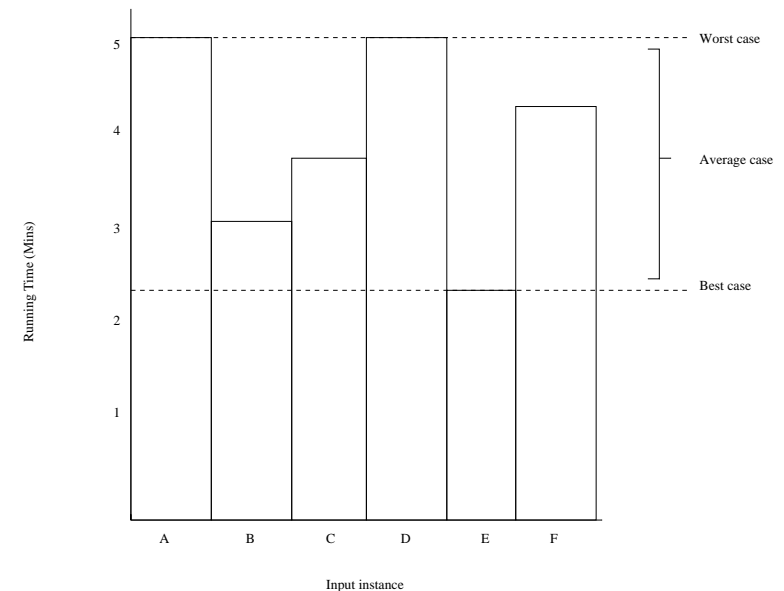
Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.

- ▶ *Average-case* complexity refers to running time as an *average taken over all inputs* of the same size.
- ▶ *Worst-case* complexity refers to running time as the *maximum taken over all inputs* of the same size.

Usually, we're most interested in worst-case complexity.

Average- vs. Worst-Case Complexity (cont.)



Asymptotic notation

- ▶ *Asymptotic notation* allows characterization of the *main factors* affecting running time.

Used in a *simplified analysis* that estimates the number of primitive operations executed *up to a constant factor*.

Such notation lets us compare the running times of two algorithms.

You first encountered this type of notation in COMP108.

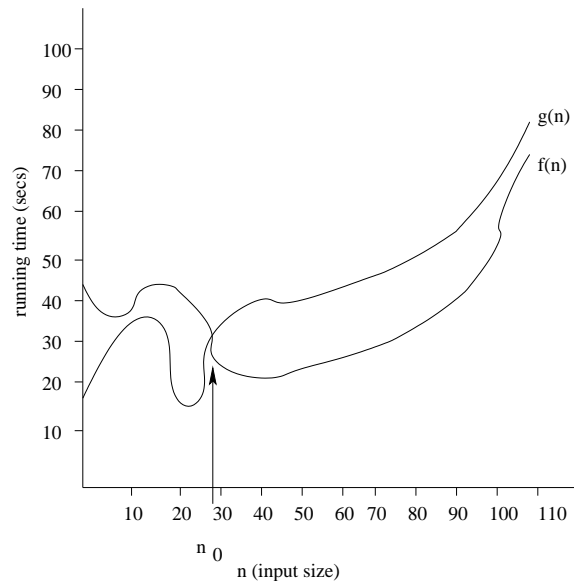
“Big-Oh” Notation

“Big-Oh” notation is probably the most commonly used form of asymptotic notation.

- ▶ Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say $f(n)$ is $O(g(n))$, written $f(n) \in O(g(n))$, if there are constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

“Big-Oh” Notation (cont.)



Example: $7n - 4 \in O(n)$

- ▶ Need to find constants c and n_0 such that: $7n - 4 \leq cn$, for all $n \geq n_0$.
- ▶ One possible choice is $c = 7$ and $n_0 = 1$.
- ▶ In fact, this is just one of infinitely many choices, because any real number $c \geq 7$ and any integer $n_0 \geq 1$ would be OK.

Further examples of Big-Oh

1. $3n^2 + n + 16$ is $O(n^2)$. Why?

We know that $n \leq n^2$ for all $n \geq 1$. Also, $16 \leq n^2$ for $n \geq 4$.
So

$$3n^2 + n + 16 \leq 3n^2 + n^2 + n^2 = 5n^2 \quad \text{for all } n \geq 4.$$

2. $13n^3 + 7n \log n + 3$ is $O(n^3)$. Why?

Because $\log n \leq n \leq n^2$ for all $n \geq 1$, and for similar reasons as above we may conclude that

$$13n^3 + 7n \log n + 3 \leq 21n^3 \quad \text{for all "large enough" } n.$$

3. Any polynomial $a_k n^k + \dots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is $O(n^k)$.

Further examples of Big-Oh (cont.)

4. Also, any polynomial $a_k n^k + \dots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is also $O(n^j)$ for all $j \geq k$.
5. Therefore, $45n^5 - 10n^2 + 6n - 12$ is $O(n^5)$ (and is also $O(n^8)$, or $O(n^9)$, or, indeed, $O(n^k)$ for any $n \geq 5$).
6. \sqrt{n} is $O(n)$.
7. $3 \log n + \log \log n$ is $O(\log n)$.
8. $\log n$ is $O(n)$, and $\log n$ is also $O(\sqrt{n})$.
9. 2^{70} is $O(1)$.
10. $5/n$ is $O(1/n)$, and $5/n$ is also $O(1/\sqrt{n})$. (Why?)

Common functions

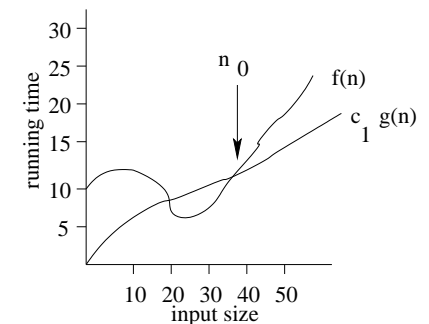
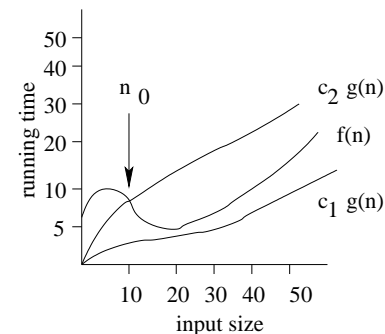
- ▶ Logarithmic $O(\log n)$
- ▶ Linear $O(n)$
- ▶ Quadratic $O(n^2)$
- ▶ Polynomial $O(n^k)$ for a positive integer k
- ▶ Exponential $O(a^n)$, $a > 1$

$\Omega(n)$ and $\Theta(n)$ notation

- ▶ We say that $f(n)$ is $\Omega(g(n))$ (*big-Omega*) if there are real constants c and n_0 such that:

$$f(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

- ▶ We say that $f(n)$ is $\Theta(g(n))$ (*Theta*) if $f(n)$ is $\Omega(g(n))$ and $f(n)$ is also $O(g(n))$. (Equivalently, $g(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$.)



Examples

1. $3 \log n + \log \log n$ is $\Omega(\log n)$.
2. $\frac{2}{3}n^2 - n \in \Omega(n^2)$.
3. $3 \log n + \log \log n \in \Theta(\log n)$.
4. $2n^3 - n + 7 \in \Theta(n^3)$.
5. A polynomial $a_k n^k + \dots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$ is $\Theta(n^k)$.

More information can be found on the COMP108 web page, and in Section 1.2 in [GT].

Importance of asymptotics

Maximum size allowed for an input instance for various running times to be solved in 1 second, 1 minute and 1 hour, assuming a 1MHz machine:

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n \log n$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

Back to “finding the minimum”

We previously said that if an array A has n items, then the running time of our “minimum finding algorithm” takes $3n - 1$ primitive operations.

Thus, this is an algorithm that is $O(n)$.

However, this is best possible, since we have to examine each item in the list to really determine the minimum. (So it's a $\Omega(n)$ algorithm.)

Putting these observations together, our minimum finding algorithm on a list of n elements is a $\Theta(n)$ algorithm.

Growth rates (running time)

Functions ordered by growth rate:

$\log_2 n$	\sqrt{n}	n	$n \log_2 n$	n^2	n^3	2^n
1	1.4	2	2	4	8	4
2	2	4	8	16	64	16
3	2.8	8	24	64	512	256
4	4	16	64	256	4096	65536
5	5.7	32	160	1024	32768	4294967296
6	8	64	384	4096	262144	1.84^{19}
7	11	128	896	16384	2097152	2.40×10^{38}
8	16	256	2048	65536	16777216	1.15^{77}
9	23	512	4608	262144	134217728	1.34×10^{154}
10	32	1024	10240	1048576	1073741824	1.79^{308}

\mathcal{NP} -completeness (a brief mention for now)

- ▶ There are certain problems whose exact solutions seem hard to find, i.e. solving large instances of them could take decades or centuries of computing time since it appears we have to try (nearly) all of the exponentially many possible solutions.
- ▶ Many of these are, however, important problems that business and industry (airlines, manufacturers, telecommunications companies, etc.) are interested in solving, such as some scheduling and optimization problems. This “hardness” can also be described in terms of the game Minesweeper, the very one included with the Windows OS.

In the absence of *exact* solutions, good *approximations* are used, but we won't generally study these *approximation algorithms*.

\mathcal{NP} -completeness (cont.)

This is such an important and fundamental problem that it's been named as one of the seven “Millennium Problems” by the Clay Institute and can earn you \$1 million dollars for its solution (and a place in mathematical and computer science history).

Go to the website

www.claymath.org/millennium/P_vs_NP
to read more.

\mathcal{NP} -completeness (cont.)

Certain problems, the so-called class of \mathcal{NP} -complete problems, are the “hardest” ones.

Somewhat surprisingly, if *any one* of these special problems can be solved efficiently, then *all* of them can be solved efficiently. So these problems are equivalent in a formal computational sense.

We will examine \mathcal{NP} -completeness in closer detail later on in the course.