Preface

These lecture notes are based on similar lectures I gave in 2001 at HKUST, Hong Kong, in 2000 at the University of Waterloo, Canada, together with Prof. Naomi Nishimura, in 1998 at the University of Trier, Germany, and in 1987–1999 at the University of Saarbrücken, Germany, together with Prof. Kurt Mehlhorn and Dr. Jop Sibeyn. I also use some material from David Mount’s CMSC 451 lecture notes.

Parts of the notes are rather short (and probably not completely free of minor errors) and it might be a good idea to read the corresponding chapters in the textbook CLRS as well.

I use the following abbreviations for textbooks:

- CLRS: Cormen, Leiserson, Rivest, Stein; An Introduction to Algorithms, 2nd edition.
- GKP: Graham, Knuth, Patashnik; Concrete Mathematics.

Shanghai, October 9, 2004
Prof Rudolf Fleischer
Chapter 1

Introduction

1.1 Objectives

In this course you learn

- Good algorithms for important problems (at least a few examples).
- Paradigms for good algorithms (the methods are important).
- Correctness proofs.
- Analysis of efficiency.
- Lower bounds.

Appendix A contains a short introduction of Finite Calculus, a general and powerful method to find closed formulas for sums. Appendix B defines a few elementary concepts of probability theory which are useful in this course.

All programs in these lecture notes are written in C++ (sometimes using the C++ library LEDA), or in C-similar pseudocode.

1.2 Literature

The following books are recommended for this course.

Data Structures and Algorithms


1.3 Algorithms

Definition 1.3.1. An algorithm is a set of rules (well-defined, finite) used for calculation or problem-solving (for example with a computer). A particular input to an algorithm is called a problem instance.

Computers are good in following rules, like adding up the numbers 1, 2, . . . , n, but they are not good in finding short-cuts, like computing \( \frac{n(n+1)}{2} \) instead of the sum.

The word algorithm goes back to the 9th century Persian mathematician al-Khowārizmī. And arithmetic goes back to the ancient Greek: arithmos = number.

Why is this course important? The most important courses you will ever hear are Data Structures and Efficient Algorithms. All other disciplines of computer science use data structures and algorithms, and analyze them (not always, unfortunately). It is a theoretical course, but you will learn how to tackle real world problems:

- apply known basic algorithms. In practice, there is often no time to implement sophisticated optimal algorithms, therefore sophisticated algorithm libraries (like LEDA, for example) can be very useful (and save you a lot of implementation time);
- if you do not know a solution, search the literature;
- if that fails, work on your own: try the paradigms you learned in this course;
- apply the methods you learned to analyze the algorithms you found or developed;
- are there lower bounds known?

Remember:
- “first think, then code” saves time (30-50% of a project time should be devoted to design);
- thorough analysis and lower bounds help you defend your approach to critics (boss, customer).

1.4 Course Syllabus

- Paradigms:
  - reuse of known solutions (always the easiest way to solve a problem);
  - divide-and-conquer;
  - greedy;
  - dynamic programming;
  - graph exploration;
  - branch-and-bound;
- Techniques for analysis:
  - asymptotic notation;
  - worst case/average case/amortized analysis;
1.5 Pseudocode

An algorithm is a problem solving recipe for human readers, a program is a problem solving recipe for computers.

Properties of pseudocode:

- control structures similar to Pascal or C;
- no strict syntax;
- mixed with plain English;
- suppress details;

Properties of programs:

- control structures similar to Pascal or C;
- strict syntax;
- not mixed with plain English;
- do not suppress details;

The web page (../Resources) also contains some info on pseudocode.

1.6 Running Times

What is a “good” algorithm?

- correct (the empty algorithm is very efficient, but seldom correct);
- efficient:
  - time (the main focus in this course);
  - space;
  - number of random bits;
  - 1/O-efficiency;
  - number of processors (for parallel algorithms);
  - communication bandwidth;

- etc.

- lower bounds:
  - quit trying harder;
  - no efficient algorithm is possible;

Example 1.6.1.

Problem: Find the maximum of $n$ numbers.

Algorithm: Sort the numbers, and return the largest element.

This algorithm is clearly correct, but is it efficient?

We can measure running times, but the information “it needed 10 seconds” is pretty useless. If we compare the running times of two algorithms $A_1$ and $A_2$, we still do not know which one is better (I did all measurements on a SUN Sparc Ultra 5 in my office in Hong Kong).

Example 1.6.2.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.36</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.71</td>
<td>0.37</td>
</tr>
<tr>
<td>4,000</td>
<td>1.48</td>
<td>1.51</td>
</tr>
<tr>
<td>8,000</td>
<td>2.93</td>
<td>6.17</td>
</tr>
</tbody>
</table>

Absolute running times are not a good measure of efficiency:

- processor dependent (clock rate);
- environment dependent (single user vs. multi-user);
- we cannot compare algorithms;

The web page (../Resources) also contains some info on pseudocode.
1.7. ASYMPTOTIC ANALYSIS

- direct/indirect memory access;
- basic operations like memory access, +, –, ·, /, etc. cost 1. This is called the unit cost measure; in the logarithmic cost measure, the cost of an operation grows linearly in the number of bits of the operands;

Remark 1.6.3.
- If each operation takes a few cycles on a real processor, then the unit cost measure describes the performance of an algorithm sufficiently well.
- The unit cost measure is independent of processor speed.
- It allows for a processor independent comparison of algorithms (but we cannot, and do not want to, predict running times).
- We can focus on certain “important” operations which dominate the running time (this makes the analysis easier):
  - Sorting: only count comparisons and data movements.
  - Chess player ranking: only count matches.
  - Recursive programs: only count recursive calls.

Space is usually measured as number of memory cells used. Both time and space are given as functions of the size of the problem instance. The problem size is the number of bits necessary to specify the instance. For RAM algorithms, size is often “number of basic variables” (if they all have constant size), like “number of items” for sorting, or “number of nodes and edges” for graph algorithms. In Example 1.6.1, we know that sorting $n$ numbers needs $n \log n$ comparisons, so we would say the algorithm has running time $n \log n$ (note that “sort” is a subroutine of cost $n \log n$, not a basic operation of cost 1).

1.7 Asymptotic Analysis

We can now compare running times in the unit cost model. Assume we have three algorithms with the following running times.

Example 1.7.1.
\[
T_1(n) = 2n \log n + 3n - 5 \\
T_2(n) = \frac{n^2}{10} - 25n + \sin n - 3 \\
T_3(n) = \begin{cases} 
3n - 4 & \text{if } n \leq 10 \\
20n^3 + 8 & \text{if } 11 \leq n \leq 55 \\
\sqrt{n} + 18 & \text{if } 56 \leq n 
\end{cases}
\]

Which algorithm is the fastest? Modern computers are fast, so running time differences for small values of $n$ are irrelevant. We are only interested in the asymptotic behaviour for large values of $n$. And since we already ignored the constant factor induced by the processor speed (all operations cost 1), we can as well ignore constant factors in the running time expressions (at least, if they are sufficiently small; a factor of $10^6$ should better not be hidden in the asymptotic notation). We are more interested whether the running time grows linearly, polynomially, or exponentially in the problem size. Formally, we use asymptotic notation to express running times.

Definition 1.7.2. Let $g : \mathbb{N} \to \mathbb{R}_+^*$ (we can cheat on that later). Then we define
\[
O(g) = \{ f : \mathbb{N} \to \mathbb{R}_+^* \mid \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}
\]
(i.e., $g(n)$ is an asymptotic upper bound on $f(n)$, up to a constant factor).
\[
o(g) = \{ f : \mathbb{N} \to \mathbb{R}_+^* \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\}
\]
(i.e., $g(n)$ is much larger than $f(n)$).

Example 1.7.3.
(a) $10n \in O(n)$ (by ‘10n’ we actually mean the function $g : \mathbb{N} \to \mathbb{R}_+^*$ with $g(n) = 10n$, but that would be too complicated to write all the time).
(b) $3n \in o(n^2)$.

Lemma 1.7.4.
\[
f \in o(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]
if the limit exists.

Proof.
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = d \iff \forall \epsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \frac{f(n)}{g(n)} \in (d - \epsilon, d + \epsilon).
\]
If $d = 0$, then $\forall \epsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) < \epsilon \cdot g(n)$. Choosing $c = \epsilon$ we see that $f \in o(g)$.
If $d > 0$, then we choose $\epsilon = c = \frac{d}{2}$. But then $\exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) > (d - \epsilon) \cdot g(n) = c \cdot g(n)$, so $f \notin o(g)$.

Example 1.7.5. $n \in o(n^2)$ because
\[
\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0.
\]
1.7. ASYMPTOTIC ANALYSIS

In the same manner we can define asymptotic lower bounds.

Definition 1.7.6. Let \( g : \mathbb{N} \to \mathbb{R}_0^+ \). Then we define

\[
\Omega(g) = \{ f : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c \cdot g(n) \leq f(n) \}
\]

(i.e., \( g(n) \) is an asymptotic lower bound on \( f(n) \), up to a constant factor),

\[
\omega(g) = \{ f : \mathbb{N} \to \mathbb{R}_0^+ \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c \cdot g(n) < f(n) \}
\]

(i.e., \( g(n) \) is much smaller than \( f(n) \)).

\[\square\]

Lemma 1.7.7. \( f \in \omega(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \)

if the limit exists.

Proof. Analogously to the proof of Lemma 1.7.4. \(\square\)

Example 1.7.8. \( n \in \omega(\sqrt{n}) \) because

\[
\lim_{n \to \infty} \frac{n}{\sqrt{n}} = \lim_{n \to \infty} \sqrt{n} = \infty.
\]

\[\square\]

We finally need a notation for the case that a function \( g \) is an exact asymptotic bound for the function \( f \).

Definition 1.7.9. Let \( g : \mathbb{N} \to \mathbb{R}_0^+ \). Then we define

\[
\Theta(g) = \Omega(g) \cap \Omega(g)
\]

\[
= \{ f : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c_1, c_2 > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}
\]

\[\square\]

Lemma 1.7.10. \( f(n) \in \Theta(g(n)) \iff 0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \)

if the limit exists.

Proof. Follows from Lemmas 1.7.4 and 1.7.7. \(\square\)

Example 1.7.11. \( \lim_{n \to \infty} \frac{\frac{n^2 - 3n + 5}{n}}{n} = \frac{1}{2} \), thus \( \frac{1}{2} n^2 - 3n + 5 \in \Theta(n^2) \).

\[\square\]

Example 1.7.12 (l'Hôpital’s Rule).

We want to compare \( \log n \) and \( \sqrt{n} \). But what is \( \lim_{n \to \infty} \frac{\log n}{\sqrt{n}} \)?

We can use l'Hôpital’s Rule: If \( f \) and \( g \) are differentiable, \( \lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty \), and \( \lim_{n \to \infty} \frac{f'(n)}{g'(n)} \) exists, then \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)} \).

In our case, \( f(n) = \log n \) and \( g(n) = \sqrt{n} \), so we have

\[
\lim_{n \to \infty} \frac{\log n}{\sqrt{n}} = \lim_{n \to \infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n \to \infty} 2 \cdot \frac{\log n}{\sqrt{n}} = 2 \cdot \lim_{n \to \infty} \frac{1}{\sqrt{n}} = 0.
\]

Thus, \( \log n \in o(\sqrt{n}) \) by Lemma 1.7.4. \(\square\)

Example 1.7.13. We want to compare \( n(1 + \sin n) \) and \( n \).

But \( \lim_{n \to \infty} \frac{n(1 + \sin n)}{n} = \lim_{n \to \infty} (1 + \sin n) \) does not exist. However, we know that \( n(1 + \sin n) \leq 2n \) for all \( n \), so we can conclude that \( n(1 + \sin n) \in O(n) \).

Since there are arbitrarily large \( n \) with \( \sin n = 1 \), \( n(1 + \sin n) \notin o(n) \). And since there are arbitrarily large \( n \) with \( \sin n = -1 \), \( n(1 + \sin n) \notin \Theta(n) \). \(\square\)

We usually write \( \log(n(1 + \sin n)) = \log(n) + \log(n + 1) \) instead of \( O(n) \), but this equality has really to be read from left to right; \( \log(n(1 + \sin n)) = \log(n) \) does not make any sense. This abuse of notation allows us to write formulas like \( n^2 + 3n - 5 = n^2 + O(n) \), emphasizing that the constant factor of the dominating term \( n^2 \) is 1; this is a stronger statement than just writing \( n^2 + 3n - 5 = O(n^2) \).

We can now continue Example 1.7.1 and compare the running times.

\[
T_1(n) = \Theta(n \log n),
\]

\[
T_2(n) = \Theta(n^2),
\]

\[
T_3(n) = \Theta(\sqrt{n}).
\]

Thus, the third algorithm is asymptotically fastest, and the second is asymptotically slowest.

Remark 1.7.14. Always express running times in the simplest possible form: write \( O(n^2) \), not \( O(10n^2 + 3n) \).

However, “simplest possible” can depend on the context. Assume we want to compare two running times \( T_1 \) and \( T_2 \). If \( T_1 \) and \( T_2 \) are sufficiently different, it is enough to determine \( T_1 \) and \( T_2 \) roughly: \( \Theta(n^2) \) is slower than \( \Theta(n) \). But if \( T_1 \) and \( T_2 \) are similar then constant factors and lower order terms matter: \( 100n^2 + O(n) \) is slower than \( 10n^2 + O(n) \), but faster than \( 100n^2 + \Theta(n^2) \). \(\square\)

Since asymptotic notation ignores constants running times become independent of real implementations of pseudocode. Proofs of the following basic rules are easy and are left as an exercise.
Theorem 1.7.15.

1. Transitivity: For \( op \in \{o, O, \omega, \Omega, \Theta\} \) we have
   
   If \( f = op(g) \) and \( g = op(h) \), then \( f = op(h) \).

2. Reflexivity: For \( op \in \{O, \Omega, \Theta\} \) we have
   
   \[ f = op(f) \].

3. Symmetry:
   
   \[ f = \Theta(g) \iff g = \Theta(f) \].

4. Transpose Symmetry:
   
   \[ f = O(g) \iff g = \Omega(f) \],
   \[ f = o(g) \iff g = \omega(f) \].

Example 1.7.16. For \( 0 < \alpha < \beta, 0 < a < b, \) and \( 0 < A < B \) we have the following hierarchy of faster growing functions:

\[ \log^* n < \log \log n < \log a n < \log^3 n < n < n^a \log^a n < n^a \log^3 n < n^b \]

\[ < A^n < A^n n^a < A^n n^b < B^n. \]

Example 1.7.17. Let’s remember the running times we measured in Example 1.6.2. What is the asymptotic growth rate of the two algorithms?

The running time of \( A_1 \) approximately doubles when we double the input size, so we have \( T_1(2n) = 2T_1(n) \), which implies \( T_1(n) = \Theta(n) \).

The running time of \( A_2 \) approximately quadruples when we double the input size, so we have \( T_2(2n) = 4T_2(n) \), which implies \( T_2(n) = \Theta(n^2) \).

We note that reading asymptotic growth rates off a table of running times can be misleading. Lower order terms like \( \log n, \log \log n, \) or \( \log^* n \) can often not be clearly recognized.

1.8 Analysis of Pseudocode

Consider the underlying model and count basic operations (in the RAM model each operation costs 1, for sorting algorithms we only count comparisons, etc.). Counting operations is straightforward for sequential tasks like loops: just add up and simplify.

Example 1.8.1. Consider the following generic loop

\[ \text{(1) for } i = 1 \text{ to } n \text{ do } A(i) ; \]

The running time \( T(n) \) of the loop depends on the time of the loop body \( A(i) \).

1. \( T(A(i)) = \Theta(1) \)

As a rule of thumb, always work from innermost to outermost. If the loop body cost is independent of the iteration (as in this case), multiply by the number of iterations:

\[ T(n) = n \cdot \Theta(1) = \Theta(n) \].

2. \( T(A(i)) = \Theta(i) \)

Otherwise, we must sum over iterations of the body cost:

\[ T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta \left( \sum_{i=1}^{n} i \right) \leq \frac{n \cdot (n + 1)}{2} = \Theta(n^2) \]

In Appendix A you find a powerful method for finding closed formulas of sums (but that is not the topic of this course).

3. \( T(A(i)) = \Theta(\frac{n^2}{n}) \)

If it becomes difficult, a rough count can at least give an upper bound:

\[ T(A(i)) \leq \Theta(n) \Rightarrow T(n) \leq n \cdot \Theta(n) = \Theta(n^2) \].

Note that we write \( T(n) \leq \Theta(n^2) \) which means \( \Theta(n^2) \) is not the exact asymptotic growth rate of \( T(n) \), just an upper bound, i.e., we only know \( T(n) = O(n^2) \). The exact growth rate is in this case:

\[ T(n) = \Theta \left( \sum_{i=1}^{n} \frac{n}{2^i} \right) = \Theta \left( n \cdot \left( 1 - \frac{1}{2^n} \right) \right) = \Theta(n). \]

4. \( T(A(i)) = \Theta(\log i) \)

Since \( \log i < \log n \), we easily see \( T(n) = O(n \log n) \). On the other hand,

\[ T(n) = \log 1 + \cdots + \log \frac{n}{2} + \cdots + \log n \geq \frac{n}{2} \cdot \log \frac{n}{2} = \frac{n}{2} \left( \log n - 1 \right) = \Omega(n \log n). \]

Thus \( T(n) = \Theta(n \log n) \).
Determining the running time becomes more difficult for non-sequential tasks.

(1) \[\textbf{if } x \leq y \textbf{ then } A \textbf{ else } B \textbf{ fi ;}\]

Just adding the running times of $A$ and $B$ would give a correct upper bound, but probably not a tight one. Even taking the maximum may overestimate the running time (if the maximal running time in $A$ only happens for some $x > y$, for example), but that is the best we can do usually. We then get an upper bound on the \textit{worst case running time} which is defined as the maximum of all running times over all inputs of the same size (if we sort we consider all possible inputs of $n$ numbers and then take the maximum running time). Analogously, the \textit{best case running time} is the minimal running time over all inputs of the same size. If we know a probability distribution on the inputs, we can also determine the \textit{average running time} which is the weighted median of all running times for all inputs of the same size. If the algorithm uses randomization, we usually determine \textit{expected running time} (or more correct: the worst case average running time) which is the weighted median of the running times of all possible computations for any particular instance. Finally, we sometimes determine the \textit{amortised running time} which is the average running time of a single operation in a long sequence of operations. This will be the topic of the next chapter.