Introduction to Algorithms

1.1 Introduction

The Algorithms and Data Structures thread of Informatics 2B deals with the issues of how to store data efficiently and how to design efficient algorithms for basic problems such as sorting and searching. This thread is taught by Kyriakos Kalorkoti (KK).

There will generally be a lecture note for each lecture of this thread—these are adaptations (in various cases extensive) of earlier notes of Don Sannella, John Longley, Martin Grohe, Mary Cryan and many others from the School of Informatics, University of Edinburgh. Some exercises will be included at the end of each lecture note, attempting these (in addition to the tutorial exercises) is a good way of reinforcing your understanding of the material. Please note that there are no sample solutions to these exercises, i.e., they do not exist. It is good practice to assess your own answers but you can always seek help, e.g., at tutorials.

If you find any errors in the notes I would appreciate being informed of them.

Prerequisites. You have seen a few algorithms and data structures in first year. Amongst these are (probably) sorting algorithms (MergeSort and Selection Sort), the principle of recursion and such things as Arrays¹, lists, and trees. However these will be explained, briefly, as they are used in this thread; you might need to do a little bit of extra reading if you have not met them before.

In addition to the Computer Science prerequisites, you will need to know some mathematics. We will assume that you are familiar with: proofs by induction, series and sums, recurrences, graphs and trees. Reminders of many of these will be included in the lectures, in particular Note 2 includes a brief appendix on induction.

Textbooks. The lecture notes provide the essentials for each topic, for further material or a different slant you should consult an appropriate book. Two recommended (though not required) algorithms textbooks for this thread are:


¹-An array is by definition a sequence of successive locations in main memory (each storing items of the same type) accessible via an index; an integer usually starting at 0 or 1 and going to an upper bound determined by the length of the array. We can access any item in constant time by using its index. Strictly speaking this is for one-dimensional arrays, but the generalisation is obvious. Unfortunately some languages have introduced array like structures which are not true arrays but either call them arrays or something very similar. do not be confused by these.
is an excellent way to develop your understanding and uncover any difficulties you have. The fact that these exercise sheets do not carry any marks is irrelevant. The aim is to help you learn, that in turn will lead to much better performance in the longer term. The absence of marks for these sheets means that you have time to improve your understanding without facing a penalty.

Practicals. There will be one practical for each thread. These are marked and the marks go towards your final assessment for the course. You should make an early start on each practical, this will allow you time to explore and uncover any problem areas for you. Just as importantly you will have time to revise your efforts and present them clearly. Last minute attempts generally lead to poor results and you learn much less; the only increased outcome being more stress.

Summary. Those who choose not to engage actively with the various aspects of the course are choosing to underperform at the very least and fail at worst. The material that must be mastered builds up quite quickly both in amount and relative difficulty, consistent regular effort is the most efficient way to make good progress.

1.2 Evaluating Algorithms

Intuitively speaking, an algorithm is a step-by-step procedure (a "recipe") for performing a task. A data structure is a systematic way of organizing data and making it accessible in certain ways. The two tasks of designing algorithms, and designing data structures, are closely linked; a good algorithm needs to use data structures that are suited to the problem. You have probably already seen some examples of data structures (arrays, linked lists, and trees) and some examples of algorithms (such as Linear search, Binary search, Selection Sort, and Mergesort) in first year. Throughout this course we will cover these in more detail (a reminder of each of them will be included as well) and introduce more. How do we decide whether an algorithm is good or not, or how good it is? This is the question we are going to talk about in this introductory lecture.

The three most important criteria for an algorithm are correctness, efficiency, and simplicity. Correctness is clearly the most important criterion for evaluating an algorithm. If the algorithm is not doing what it is supposed to do, it is worthless. All the algorithms we present in this course are guaranteed to perform their tasks correctly, although we will not always cover the proofs. Some proofs of correctness are simple while other, more subtle, algorithms require complicated proofs of correctness.

We can make this precise, e.g., by introducing a theoretical machine such as the Turing machine but for this course the intuitive definition will suffice. The precise definition is essential for establishing negative results, e.g., that there is no algorithm that can decide if a given algorithm terminates on a given input. To be clear we can prove that there is no algorithm $T$ that takes as input any algorithm $A$ and input $i$ for $A$ and returns TRUE if $A$ halts on input $i$ otherwise FALSE. The problem is called the Halting Problem and its algorithmic unsolvability is a fundamental result in the theory of computability. The surprising thing is that it is not hard to prove once the appropriate notions are put into place. Note that all the algorithms we study in this course can be seen quite easily to halt on all inputs, this does not contradict the unsolvability of the halting problem.

The main focus of this thread will be on efficiency. We would like our algorithms to make efficient use of the computer’s resources. In particular, we would like them to run as fast as possible using as little memory as possible (though there is often a trade off here). To keep things simple, we will concentrate on the running time of the algorithms for a large part of this thread, and not worry too much about the amount of space (i.e., amount of memory) they use. However, in our lectures on algorithms for the internet, space will be an issue since these deal with huge quantities of data.

The third important criterion of an algorithm is simplicity. We would like our algorithms to be easy to understand and implement. Unfortunately, there is often (but not always) a trade-off between efficiency and simplicity: more efficient algorithms tend to be more complicated. Which of the two criteria is more important depends on the particular task. If a program is used only once or a few times, the cost of implementing a very efficient but complicated algorithm may exceed the cost of running a simple but inefficient one. On the other hand, if a program will be used very often, the cost of running an inefficient algorithm over and over again may exceed by far the cost of implementing an efficient but complicated one. It is up to the programmer (or system designer) to decide which way to go in any particular situation. In some application areas (e.g., real time control systems) we cannot afford inefficiency even for very rare events. For example the shutting down of some critical reaction in a chemical factory should be a rare event but it must be done in good time when necessary.

1.3 Measuring the Running Time

The running time of a program depends on a number of factors such as:

1. The running time of the algorithm on which the program is based.
2. The input given to the program.
3. The quality of the implementation and the quality of the code generated by the compiler.
4. The machine used to execute the program.

Warning. These are not the only factors. An approach that uses heavy recursion can often be slow when it is coded as a program (especially in an imperative language such as JAVA—functional languages such as Haskell are better at managing recursion efficiently). Moreover, JAVA (the language we use in Inf2B) has a heavy garbage collection overhead, so when we estimate running times with JAVA’s System.currentTimeMillis() command, the cost of garbage collection often drowns out the true running time of the computation (unfortunately, for this reason, we often don’t see the effect of an efficient algorithm until the input is very large).

Anyway, in developing a theory of efficiency, we don’t want a theory that is only valid for algorithms implemented by a particular programmer using a certain programming language as well as compiler and being executed on a particular machine. Therefore, in designing algorithms and data structures, we will...
abstract away from all these factors. We do this in the following way: We describe our algorithms in a pseudo-code that resembles programming languages such as JAVA or C, but is less formal (since it is only intended to be read and understood by humans). We assume that each instruction of our pseudo-code can be executed in a constant amount of time (more precisely, in a constant number of machine cycles that does not depend on the actual values of the variables involved in the statement), no matter which “real” programming language it is implemented in and which machine it is executed on. This can best be illustrated by a very simple example, Linear search, which you are very likely to have seen before.

Example 1.1. Algorithm 1.2 is the pseudo-code description of an algorithm that searches for an integer in an array of integers. Compare this with the actual implementation as a JAVA method displayed in Figure 1.3. The easiest way to implement most of the algorithms described in this thread is as static JAVA methods. We can embed them in simple test applications such as the one displayed in Figure 1.4.

Algorithm LinSearch(A, k)
Input: An integer array A, an integer k
Output: The smallest index i with A[i] = k, if such an i exists, or -1 otherwise.
1. for i = 0 to A.length - 1 do
2. if A[i] = k then
3. return i;
4. return -1

Algorithm 1.2

public static int linSearch(int[] A, int k) {
    for (int i = 0; i < A.length; i++)
        if (A[i] == k)
            return i;
    return -1;
}

Figure 1.3. An implementation of Algorithm 1.2 in JAVA

What we would like to estimate is the number of machine cycles an execution of the algorithm requires. More abstractly, instead of machine cycles we usually speak of computation steps. If we wanted to, we could multiply the number of steps by the clock speed of a particular machine to give us an estimate of the actual running time.

We do not know how many computation steps one execution of, say, Line 1 requires—this will depend on factors such as the programming language, the compiler, and the instruction set of the CPU—but clearly for every reasonable implementation on a standard machine it will only require a small constant number of steps. Here “constant” means that the number can be bounded by a number that does not depend on the values of i and A.length. Let \( c_1 \) be such a constant representing the number of computational steps to execute line 1 (for a single i).

Similarly, let us say that one execution of Lines 2, 3, 4 requires at most \( c_2, c_3, c_4 \) steps respectively.

To compute the number of computation steps executed by Algorithm 1.2 on input \( A = (2, 4, 3, 3, 5, 6, 1, 0, 9, 7) \) and \( k = 0 \), we have to find out how often each line is executed. Line 1 is executed 8 times (for the values \( i = 0, \ldots, 7 \)). Line 2 is also executed 8 times. Line 3 is executed once, and Line 4 is never executed. Thus overall, for the particular input \( A \) and \( k \), at most

\[ 8c_1 + 8c_2 + c_3 \]

computation steps are performed. A similar analysis shows that on input \( A = \)

Note that in effect we are assuming that the numbers held by the array (and the input integer \( k \)) are bounded in size. This is realistic for any actual computer. If we postulated a theoretical model in which numbers of arbitrary size can be held in array locations then we would have to take the size of the numbers into account. Even so our analysis as presented here would still be useful; to obtain an upper bound on runtime we multiply our result by an upper bound estimate on the worst case cost of any of the operation involving the numbers. Such a separation of concerns is often vital in making progress.
The definition above has an obvious counterpart for use two or more such quantities, each corresponding to the size of an input parameter (e.g., two arrays). In some cases is is more convenient to look at each of these, then the number of arrays of size \( n \) appears at the last entry, the number of computation steps performed is at most
\[
\sum_{i=1}^{k} c_i n + c_3 = (c_1 + c_2)n + c_3 + c_4.
\]
Thus the time complexity of \( \text{linSearch} \) is
\[
T_{\text{linSearch}}(n) \leq \max\{ (c_1 + c_2)n + c_3, (c_1 + c_2)n + (c_1 + c_2) \}
\[
= (c_1 + c_2)n + \max\{c_3, (c_1 + c_2) \}
\]
Since we do not know the constants \( c_1, c_2, c_3, c_4 \), we cannot tell which of the two values is larger. But the important thing we can see from our analysis is that the time complexity of \( \text{linSearch} \) is a function of the form
\[
an + b,
\]
for some constants \( a, b \). This is a linear function, therefore we say that \( \text{linSearch} \) requires linear time.

We can compare \( \text{linSearch} \) with another algorithm, which you are very likely to have seen before:

**Example 1.10.** Algorithm 1.11 is the pseudo-code of an algorithm that searches for an integer in a sorted array of integers using binary search. Note that binary search only works correctly on sorted input arrays [why?]. In this case we assume the input array is sorted in increasing order, there is an obvious modification for arrays that are sorted in decreasing order.

Let us analyse the algorithm. Again, we assume that each execution of a line of code only requires constant time\(^8\) and we let \( c_1 \) be the time required to execute Line 1 once. However, for the recursive calls in Lines 8 and 10, \( c_6 \) and \( c_8 \) only account for the number of computation steps needed to initialise the recursive call (i.e. to write the local variables on the stack etc.), but not for the number of steps needed to actually execute \( \text{binarySearch}(A, k, i, j - 1) \) and \( \text{binarySearch}(A, k, i, j + 1, i_2) \).

The size of an input \( A, k, i, j \) of binarySearch is
\[
(n - i_2) - i_1 + 1,
\]
i.e., the number of entries of the array \( A \) to be investigated. If we search the whole array, i.e., if \( i_1 = 0 \) and \( i_2 = A \text{ length} - 1 \), then \( s \) is just the number of entries
\[8\]Note that this means we are assuming that compound structures, such as arrays, are passed by reference. If local copies were made there would be a very significant effect on runtime, usually resulting in a very inefficient algorithm. In this example setting up each recursive call would cost time proportional to the size of the array being passed rather than constant time.

\[8\]Note that this means we are assuming that compound structures, such as arrays, are passed by reference. If local copies were made there would be a very significant effect on runtime, usually resulting in a very inefficient algorithm. In this example setting up each recursive call would cost time proportional to the size of the array being passed rather than constant time.
Algorithm binarySearch($A, k, i, j$)

**Input:** An integer array $A$ sorted in increasing order, integers $i, j$ and $k$

**Output:** An index $i$ with $i_1 \leq i \leq i_2$ and $A[i] = k$, if such an $i$ exists, or $-1$ otherwise.

1. if $i_2 < i_1$ then return $-1$
2. else return $j$
3. if $k = A[j]$ then return $j$
4. else if $k < A[j]$ then return binarySearch($A, k, i_2, j - 1$)
5. else return binarySearch($A, k, j + 1, i_2$)

Algorithm 1.11

of $A$ as before. Then we can estimate the time complexity of binarySearch as follows:

$$T_{\text{binarySearch}}(n) \leq \sum_{i=1}^{\lceil \log_2 n \rceil} c_i + T_{\text{binarySearch}}(\lfloor n/2 \rfloor).$$

Here we add up the number of computation steps required to execute all lines of code and the time required by the recursive calls. Note that the algorithm makes at most one recursive call, either in Line 8 or in Line 10. Observing that

$$j - (j - 1) = \frac{i_1 + i_2}{2} - i_1 = \frac{i_2 - i_1}{2} = \frac{n - 1}{2} \leq \lfloor n/2 \rfloor$$

and

$$j_2 - (j_2 - 1) = \frac{i_2 + i_2}{2} - \frac{i_2 - i_2}{2} = \frac{n - 1}{2} \leq \lfloor n/2 \rfloor,$$

we see that this recursive call is made on an input of size at most $\lfloor n/2 \rfloor$. Thus the cost of the recursive call is at most $T_{\text{binarySearch}}(\lfloor n/2 \rfloor)$. Note that we have assumed that $T_{\text{binarySearch}}$ is a non-decreasing function of its argument. This is clearly a very plausible assumption and is easy to establish (we will not do here in order not to disrupt the key point). We let $c = \sum_{i=1}^{\lceil \log_2 n \rceil} c_i$ and claim that

$$T_{\text{binarySearch}}(n) \leq c(\log_2 n) + 2$$

for all $n \in \mathbb{N}$ with $n \geq 1$ (recall that $\log_2 n = \log_2(n)$, i.e., ‘log to the base 2’).

We prove (1.2) by induction on $n$. For the induction base $n = 1$ we obtain

$$T_{\text{binarySearch}}(1) \leq c + T_{\text{binarySearch}}(0) \leq 2c = c(\log_2 1) + 2.$$
Inf2B Algorithms and Data Structures Note 1 Informatics 2B

<table>
<thead>
<tr>
<th>( n )</th>
<th>( T_{\text{linSearch}}(n) )</th>
<th>( T_{\text{binarySearch}}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>422</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>7644</td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
<td>10966</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000</td>
<td>14288</td>
</tr>
<tr>
<td>100,000</td>
<td>1,000,000</td>
<td>17,610</td>
</tr>
<tr>
<td>1,000,000</td>
<td>10,000,000</td>
<td>20532</td>
</tr>
</tbody>
</table>

**Table 1.13.** The time complexity of linSearch and binarySearch under the assumption that \( T_{\text{linSearch}}(n) = 10n + 10 \) and \( T_{\text{binarySearch}}(n) = 1000\log(n) + 1000 \).

The message of Table 1.13 and the graph is that the constant factors \( a, b, c, d \) are dominated by the relationship between the growth rate of \( n \) versus \( \log(n) \). So what will be most important in analysing the efficiency of an algorithm will be working out its underlying growth rate, forgetting constants.

It is easy to understand the behaviour of the two functions. By definition \( n = 2^{\log(n)} \).

Thus in order to increase the value of \( \log(n) \) by 1 we must double the value of \( n \). So to increase \( \log(n) \) by 10 we must multiply the value of \( n \) by \( 2^{10} = 1024 \) and to increase the value of \( \log(n) \) by 20 we must multiply the value of \( n \) by \( 2^{20} = 1,048,576 \).

This behaviour is an example of exponential growth.

We have encoded both of the linSearch and binarySearch algorithms in JAVA, and have run tests on arrays of various sizes (performed on a DICE machine in January 2008), in order to see how the details of Table 1.13 influence the performance of the algorithms in practice. A line of the table is to be read as follows. Column 1 is the input size. Each of the two algorithms was run on 200 randomly generated arrays of this size. Column 2 is the worst case time (over the 200 runs) of linSearch, Column 3 the average time. Similarly, Column 4 is the worst-case time of binarySearch and Column 5 the average time.

The complete test application we used is Search.java, and can be found on the Inf2B webpages. The way the running time is measured here is quite primitive: The current machine time is before starting the algorithm is subtracted from the time after starting the algorithm. This is not a very exact way of measuring the running time in a multi-tasking operating system. However, in current JAVA compilers, there is unfortunately no way of turning off the garbage collection facility, so a rough estimate is the best we can hope for.

Before we leave this comparison it is well worth thinking about the reason for the far superior efficiency of binarySearch over linSearch. Compare the following observations on what happens after one comparison:

linSearch: we either succeed or have dismissed just one item of current data and the rest of the data to be examined.

binarySearch: the changes in the search interval are such that the midpoint is always close to the target--in most cases, at the target.

---

*Given by java.lang.System.currentTimeMillis().*
Table 1.15. Running Times of \texttt{linSearch} and \texttt{binarySearch}

<table>
<thead>
<tr>
<th>size</th>
<th>\texttt{wc linS}</th>
<th>\texttt{ave linS}</th>
<th>\texttt{wc binS}</th>
<th>\texttt{ave binS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>100</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>1000</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>10000</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>200000</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>400000</td>
<td>3 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>600000</td>
<td>3 ms</td>
<td>1.5 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>800000</td>
<td>3 ms</td>
<td>1.5 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>1000000</td>
<td>5 ms</td>
<td>2.1 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>2000000</td>
<td>7 ms</td>
<td>3.7 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>4000000</td>
<td>12 ms</td>
<td>6.9 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>6000000</td>
<td>24 ms</td>
<td>11.6 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
<tr>
<td>8000000</td>
<td>24 ms</td>
<td>15.6 ms</td>
<td>\leq 1 ms</td>
<td>\leq 1 ms</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{Time} & \textbf{$n = 10$} & \textbf{$n = 20$} & \textbf{$n = 50$} & \textbf{$n = 100$} & \textbf{$n = 1000$} \\
\hline
\texttt{nlgn} & 33 \mu s & 86 \mu s & 282 \mu s & 664 \mu s & 10 ms \\
\hline
\texttt{n^2} & 100 \mu s & 400 \mu s & 3 ms & 10 ms & 1 s \\
\hline
\texttt{n^5} & 100 ms & 3 s & 5 mins & 3 hrs & 32 yrs \\
\hline
$2^n$ & 1 ms & 1 s & 36 yrs & $4 \times 10^{18}$ yrs \\
\hline
n! & 4 s & 774 cents \\
\hline
\end{tabular}
\caption{Time required to solve problems of size $n$ with an algorithm of the given runtime (algorithm runtime in \mu s, i.e., $10^{-6}$ seconds)}
\end{table}

\subsection{1.4 The need for efficient algorithms}

Efficiency is not usually a major issue for small problem sizes\footnotemark[7] but in many applications the size of the data is immense. Figure 1.16 illustrates the problem very clearly.

The preceding discussion shows that the replacement of the factor $n$ in the worst case runtime of \texttt{linSearch} with $\lg n$ in the worst case runtime of \texttt{binarySearch} is a huge improvement. We will see similar gains, e.g., when we consider sorting where various simple algorithms have worst case runtime proportional to $n^2$ whereas more sophisticated ones have worst case runtime proportional to $n \lg n$. No improvement of hardware speed can match this, since such an improvement is by some constant factor.

\footnotetext[7]{Note however that an inefficient algorithm that is run a huge number of times is a problem even if each instance is small. Furthermore, if we are dealing with safety critical systems then even the slightest delay can be a real problem.}

Figure 1.16. Largest problem sizes solvable in 1 minute

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\multicolumn{5}{|c|}{Fastest comp. in 1990} \\
\hline
\texttt{nlgn} & 1.5 trillion & 1000 trillion \\
\hline
\texttt{n^2} & 8 million & 260 million \\
\hline
\texttt{n^5} & 570 & 2300 \\
\hline
$2^n$ & 46 & 56 \\
\hline
n! & 16 & 18 \\
\hline
\end{tabular}
\caption{Largest problem sizes solvable in 1 minute (algorithm runtime in \mu s, i.e., $10^{-6}$ seconds)}
\end{table}
drastic effect that exponential runtimes have (in our table these are $2^n$ and $n!$). Of course we aim for the best of both worlds: efficient algorithms and fast hardware.

### 1.5 The need for theoretical analysis

Finally in this note we discuss why a theoretical analysis of algorithms is essential. It might at first seem to be sufficient to implement algorithms of interest to us and run some timing experiments. There are many problems with taking this approach exclusively; we will focus on the two most critical ones.

First of all our experiments can only go up to some upper bound on the size $n$ of the input. Thus they cannot show us the trend as $n$ increases. However we could just note here that in any realistic context there will indeed be an upper bound to $n$ albeit we might be hard put to it to fix it precisely. Let’s concede this point, even though it is not as clear as it might seem.

The real problem is as follows. Consider one of the most fundamental computational problems: sorting objects on which we have a known linear order, let’s say integers. It is surely reasonable to expect that in many applications we will want the fastest runtime in practice. Produce a version of the algorithm that uses only a while loop. This is a simple exercise and gives you an opportunity to get used to expressing things with pseudocode (much easier than any actual programming language, no rigid syntax for a start!).

Now even if the algorithm could sort $10^{28}$ instances per second(!) the experiment would take at least

$$99! \approx 9.33262154444311586286467099071558628418793953679065981734739510217361981495361222180421763297800526934966836189951709107133982557649152454057072218523552866341996843895217750993229515608941463976156514286256307932082722373257852118521$$

Now even if the algorithm could sort $10^{28}$ instances per second(!) the experiment would take at least

$$10^{28} \times 60 \times 60 \times 24 \times 365 \times 10^{18} \approx 2.951269209 \times 10^{48}$$

years to complete (we have been generous again and taken each year to be a leap year). Time enough for a cup of tea then!

This section must not be taken to imply that experiments have no value. Such a sample size might well provide a reliable estimate but we cannot know this without further reasoning about our algorithm. Here we are addressing the shortcomings of an approach based solely on experiments.

### 1.6 Reading Material

[CLRS], Chapter 1 and Sections 2.1-2.2.

#### Exercises

1. Consider the algorithm findMax (Algorithm 1.18) that finds the maximum entry in an integer array.

   **Algorithm findMax$(A)$**

   **Input:** An integer array $A$

   **Output:** The maximum entry of $A$.  

   1. $m \leftarrow A[0]$
   2. for $i \leftarrow 1$ to $A.length - 1$ do
   3. if $A[i] > m$ then
   4. $m \leftarrow A[i]$
   5. return $m$

   **Algorithm 1.18**

   Show that there are constants $c_1, c_2, d_1, d_2 \in \mathbb{N}$ such that

   $$c_1 n + d_1 \leq T_{findMax}(n) \leq c_2 n + d_2$$

   for all $n$, where $n$ is the length of the input array. Argue that for every algorithm $A$ for finding the maximum entry of an integer array of length $n$ it holds that

   $$T_A(n) \geq n$$

   2. You have 70 coins that are all supposed to be gold coins of the same weight, but you know that 1 coin is fake and weighs less than the others. You have a balance scale; you can put any number of coins on each side of the scale at one time, and it will tell you if the two sides weigh the same, or which side is lighter if they don’t weigh the same. Outline an algorithm for finding the fake coin. How many weighings will you do? What does this puzzle have to do with the contents of this lecture note?

   3. Algorithm 1.11 makes use of recursion which is an unnecessary overhead if we want the fastest runtime in practice. Produce a version of the algorithm that uses only a while loop. This is a simple exercise and gives you an opportunity to get used to expressing things with pseudocode (much easier than any actual programming language, no rigid syntax for a start!).