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 reenforcing 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 Arrays[1], 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:

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*. McGraw-Hill, 2002 (now in its third edition, published September 2009).

[GT] Michael T. Goodrich and Roberto Tamassia, *Algorithm Design – Foundations, Analysis, and Internet Examples*. Wiley, 2002 (various editions have followed, e.g., 2014).

[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.

Both of these books cover most of the Algorithms and Data Structures material of Informatics 2B. [GT] puts a lot of emphasis on well-designed JAVA implementations of the algorithms, whereas [CLRS] is more theoretical. Of the two books, [GT] covers the course more closely (and is easier to read). However, [CLRS] is the textbook used in the third year Algorithms and Data Structures (ADS) course. Therefore if you expect to be taking this course (quite likely if you are on the CS degree) it is clearly a good idea to buy [CLRS], if you do feel the need to make a purchase at all. It is comprehensive and therefore makes an excellent reference book.

Online resources. `Wikipedia` has good coverage on many of the topics covered in the Algorithms and Data structures thread, as well as the Learning and Data thread.

Studying the notes. You should get into the habit of making a careful study of the lecture notes soon after each lecture. Skim reading them is *not* sufficient for proper study. Have pen and paper to hand so that you can work through technical details whenever appropriate. If an assertion is made without further justification then work out why it is true (unless the notes state that the assertion is beyond the scope of this course). There are some occasional reminders in the notes of the need to engage in this way (mostly by putting "[why?]" after an assertion) but you should always be asking yourself why something is true. If an assumption is made in a statement where is it used? Is it made just to simplify things or is it essential (if so how would things go wrong without it)? The material of this course goes well beyond mere parrot fashion recitation of facts.

When reading the notes for the first time it is unrealistic to expect to follow everything. At times there is a fair amount of technical detail and this can get in the way of appreciating the bigger picture. So a good strategy is likely to be to leave details for a subsequent reading and aim initially to understand the role that various things play. This does *not* mean that you can ignore the details, you do need to understand them.

The first two notes are quite long because of the need to give appropriate advice on various aspects and to establish the basics on which the rest of the course relies. The remaining notes are significantly shorter.

Lectures. You should attend all lectures. Notes are made available in order to free up lecture time for the discussion of essentials and intuitive explanations. A quick reading of the relevant note before each lecture can also be very helpful. At first the lectures will follow the notes fairly closely in order to set up the basics. Once that is done, lectures will focus on discussions that are best carried out orally. During a live presentation you should not be overly concerned with following every technical detail exactly but should be able to see the overall thrust of what is being discussed. Minute details are best studied in your own time and at your own pace. If, however, you find that you are frequently unable to follow large parts of a lecture then that is a sure sign that you are falling behind and should therefore make every effort to catch up.

Tutorials. There will be a number of tutorials (split about equally between the two threads). Appropriate exercise sheets will be issued for each tutorial. Again you should attend all of these and attempt a fair proportion of the exercises, this 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 it 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[2]. A *data structure* is a systematic way of organising 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 to discuss 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.

[2]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 $\mathcal{H}$ that takes as input any algorithm $\mathcal{A}$ and input I for $\mathcal{A}$ and returns TRUE if $\mathcal{A}$ 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 [why?].

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).

Anyhow, 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** $i \leftarrow 0$ **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

The pseudocode is fairly self explanatory, note that we use indentation to indicate scope. This avoids unnecessary clutter and works well for our purposes because our pseudocode tends to be quite short.

We now want to get an estimate of the running time of linSearch when the input consists of the array $A = \langle 2, 4, 3, 3, 5, 6, 1, 0, 9, 7 \rangle$ and the integer $k = 0$. The length of A is 10, and k occurs with index 7.

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

    public static void main(String[] args) {
        int[] A = {2,4,3,3,5,6,1,0,9,7};
        System.out.println(linSearch(A,0));
        System.out.println(linSearch(A,10));
    }
}
```

Figure 1.4. A simple (but complete) JAVA application involving Algorithm 1.2

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[3].

To compute the number of computation steps executed by Algorithm 1.2 on input $A = \langle 2, 4, 3, 3, 5, 6, 1, 0, 9, 7\rangle$ 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 $A =$

[3]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.

$\langle 2, 4, 3, 3, 5, 6, 1, 0, 9, 7 \rangle$ and $k = 10$ at most

$$11c_1 + 10c_2 + c_4$$

computation steps are performed. (Line 1 is performed 11 times, the last time to check that $11 \geq 10 = A.length$.)

So far, this looks like a tedious exercise. All we have obtained is a dubious-looking estimate, involving unknown constants, on the number of computation steps performed by our algorithm on two particular inputs. What we would like to have is an estimate on the number of computation steps performed on *arbitrary* inputs. But clearly, we cannot just disregard the input: an execution of our algorithm can be expected to perform more computation steps if the input array has length 100,000 than if the input array has length 10. Therefore, we define the *running time* of the algorithm as a function of the *size* of the input, which in Example 1.1 is the length of A (plus 1, for the value of k). But even for input arrays of the same length the execution time may vary greatly, depending on where the key k actually appears in the array. We resolve this by giving an estimate on the number of computation steps needed in the *worst case*.

In these notes the symbol $\mathbb{N}$ denotes the set $\{0, 1, 2, 3, \ldots\}$ of natural numbers (including 0)[4].

Definition 1.5. The *(worst-case) running time* of an algorithm A is the function $T_{\mathsf{A}} : \mathbb{N} \to \mathbb{N}$ where $T_{\mathsf{A}}(n)$ is the maximum number of computation steps performed by A on an input of size n.

Remark 1.6. Note that for a given size there are only finitely many possible inputs of that size. For example if we are considering arrays of size n each location can only hold integers up to some maximum size. Let's say there are B of these, then the number of arrays of size n is B^n [why?].

The point of this observation is that T_{A} is always well defined: for a given n we just have a finite number of runtimes to consider so there will indeed be a maximum. If there were infinitely many possible runtimes there might not be a maximum since the runtime numbers could keep growing.

Remark 1.7. For most of our algorithms we will represent the size of the input by one quantity (usually denoted n). In some cases is is more convenient to use two or more such quantities, each corresponding to the size of an input parameter (e.g., two arrays). The definition above has an obvious counterpart for this situation.

Remark 1.8. Note that the function $T_{\mathsf{A}} : \mathbb{N} \to \mathbb{N}$ is a perfectly normal mathematical function, it is defined via a property of the algorithm A but it is *not itself* an algorithm. It therefore makes no sense to speak of the runtime of T_{A} any more than it does to speak of the runtime of $\sin$, $\cos$ or $\ln$.

Remark 1.9. There is also a notion of *average time complexity* (it is not central to Inf2B). Average-case complexity is measured by considering all possible sets of inputs for a given problem, and taking the average of the running-times for all those inputs. [GT] and [CLRS] have more details of average case complexity.

Example 1.1 (cont'd). Let us analyse the worst-case time complexity of algorithm linSearch. It is obvious that the worst-case inputs are those where either k

[4]Some authors exclude 0 from the natural numbers (which I find rather unnatural)–KK.

only appears as the last entry of A or where k does not appear at all (this is obvious because in these cases, we have to execute the loop $A.length$ times). For more complicated situations the worst case is not necessarily so obvious.

Suppose A has length n. If k appears at the last entry, the number of computation steps performed is at most

$$c_1n + c_2n + c_3 = (c_1 + c_2)n + c_3,$$

and if k does not appear at all it is

$$c_1(n+1) + c_2n + c_4 = (c_1 + c_2)n + (c_1 + c_4).$$

Thus the time complexity of linSearch is

$$\begin{aligned} T_{\text{linSearch}}(n) &\leq \max\big\{(c_1+c_2)n + c_3, (c_1+c_2)n + (c_1+c_4)\big\} \\ &= (c_1+c_2)n + \max\big\{c_3, (c_1+c_4)\big\} \end{aligned}$$

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 linSearch is a function of the form

$$an + b, \tag{1.1}$$

for some constants a, b. This is a linear function, therefore we say that linSearch requires *linear time*.

We can compare 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[5] and we let c_i be the time required to execute Line i once. However, for the recursive calls in Lines 8 and 10, c_8 and c_{10} 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 binarySearch$(A, k, i_1, j-1)$ and binarySearch$(A, k, j+1, i_2)$.

The size of an input A, k, i_1, i_2 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 $i_2 = A.length - 1$ then n is just the number of entries

[5]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_1, i_2)

Input: An integer array A sorted in increasing order, integers i_1, i_2 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**
2.   **return** -1
3. **else**
4. $j \leftarrow \lfloor \frac{i_1+i_2}{2} \rfloor$
5.   **if** $k = A[j]$ **then**
6.   **return** j
7.   **else if** $k < A[j]$ **then**
8.   **return** binarySearch$(A, k, i_1, j-1)$
9.   **else**
10.   **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:

$$T_{\text{binarySearch}}(n) \leq \sum_{i=1}^{10} c_i + T_{\text{binarySearch}}(\lfloor n/2 \rfloor).$$

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. First we note the following facts: for all integers x and y

(1) $\lfloor y/2 \rfloor - x = \lfloor y/2 - x \rfloor$.

(2) $x - \lfloor (x+y)/2 \rfloor \leq \lfloor (y-x+1)/2 \rfloor$ provided $y \geq x$.

The first one is very easy. To prove the second one we observe that for any integer $n \geq 0$ we have $\lfloor n/2 \rfloor$ is equal $n/2$ if n is even otherwise it is equal to $n/2 - 1/2$. Moreover $x + y$ is even if and only if $y - x$ is even. Now if $x + y$ is even then $x - \lfloor (x+y)/2 \rfloor = x - (x+y)/2 = (y-x)/2$ while $\lfloor (y-x+1)/2 \rfloor = (y-x+1)/2 - 1/2 = (y-x)/2$, since $y-x+1$ is odd. If, on the other hand, $x+y$ is odd then $x - \lfloor (x+y)/2 \rfloor = x - (x+y)/2 - 1/2 = (y-x-1)/2$ while $\lfloor (y-x+1)/2 \rfloor = (y-x+1)/2$, since $y - x + 1$ is even.

Now

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

and

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

$$i_2 - (j+1) + 1 = i_2 - \left\lfloor \frac{i_1 + i_2}{2} \right\rfloor = \left\lfloor \frac{i_2 - i_1}{2} \right\rfloor = \left\lfloor \frac{n-1}{2} \right\rfloor \le \lfloor n/2 \rfloor,$$

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 cost 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 so here in order not to disrupt the key point). We let $c = \sum_{i=1}^{10} c_i$ and claim that

$$T_{\text{binarySearch}}(n) \le c(\lg(n) + 2) \tag{1.2}$$

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

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

$$T_{\text{binarySearch}}(1) \le c + T_{\text{binarySearch}}(0) \underset{(\star)}{\le} 2c = c(\lg(1) + 2).$$

The inequality $(\star)$ holds because $T_{\text{binarySearch}}(0) \le c_1 + c_2 \le c$.

For the inductive step, let $n \ge 2$. We assume that we have already proved (1.2) for all $n' < n$ and now want to prove it for n. We have

$$\begin{aligned} T_{\text{binarySearch}}(n) &\le c + T_{\text{binarySearch}}(\lfloor n/2 \rfloor) \\ &\le c + c(\lg(\lfloor n/2 \rfloor) + 2) \\ &\le c + c(\lg(n) - 1 + 2) \qquad \big(\text{since } \lg(\lfloor n/2 \rfloor) \le \lg(n/2) = \lg(n) - 1\big) \\ &= c(\lg(n) + 2). \end{aligned}$$

This proves (1.2).

Remark 1.12. The preceding proof by induction gives us an unimpeachable demonstration that (1.2) is indeed correct. However it provides no intuitive insight as to why. Let us reconsider the problem. What we know is that when `binarySearch` is called on an array of size n it does a constant amount of work (which we have denoted by c) and then recurses on an array of size around $n/2$ (unless it has finished). Well at this level it also does a constant amount of work (i.e., c) and then recurses on an array of size around $n/2^2$ (unless it has finished). So in the worst case it keeps recursing having done c amount of work each time. When does this stop? It does so when we get to an array of size 1 and then it makes one more call (we could avoid this by changing the code slightly). After r recursive calls the size of the array is around $n/2^r$ so we get to a size 1 aray when $2^r = n$, i.e., when $r = \lg(n)$ (strictly we should adjust this as r must be an integer, but we are just trying to get a rough idea of the situation). Thus the number of recursive calls is at most $\lg(n) + 1$ and each of them does c amount of work but we also do c amount of work before the first recursive call. This gives us $c(\lg(n) + 2)$ as an upper bound for the total amount of work (or time). Having obtained this we can then proceed to prove its correctness by induction.

Hence we get the following bound on the running time of linSearch:

$$T_{\text{linSearch}}(n) \le an + b$$

for suitable constants a, b, and for the running time of binarySearch we have

$$T_{\text{binarySearch}}(n) \leq c \lg(n) + d$$

for suitable constants c, d. Now what does this mean? After all, we don't know the constants a, b, c, d. Well, even if we don't know the exact constants, it tells us that binary search is much faster than linear search. Remember where the constants came from: they were upper bounds on the number of machine cycles needed to execute a few very basic instructions. So they won't be very large. To support our claim that binary search is faster than linear search, let us make an assumption on the constants that strongly favours sequential search: Assume that $a = b = 10$ and $c = d = 1000$. Table 1.13 shows the number of computation steps performed by the two algorithms for increasing input sizes. Of course for small input arrays, the large constants we assumed for binarySearch have a strong impact. But already for arrays of size 10,000 binarySearch is more than 6 times faster, and for arrays of size 1,000,000, binarySearch is about 50 times faster. Figure 1.14 shows the graph of the two functions up to $n = 10000$ (going

n	$T_{\text{linSearch}}(n)$	$T_{\text{binarySearch}}(n)$
10	110	$4,322$
100	$1,010$	$7,644$
$1,000$	$10,010$	$10,966$
$10,000$	$100,010$	$14,288$
$100,000$	$1,000,010$	$17,610$
$1,000,000$	$10,000,010$	$20,932$

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

much higher tends to conceal the crossover point).

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 $\lg(n)$. 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

$$n = 2^{\lg(n)}.$$

Thus in order to increase the value of $\lg(n)$ by 1 we must *double* the value of n. So to increase $\lg(n)$ by 10 we must multiply the value of n by $2^{10} = 1024$ and to increase the value of $\lg(n)$ 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

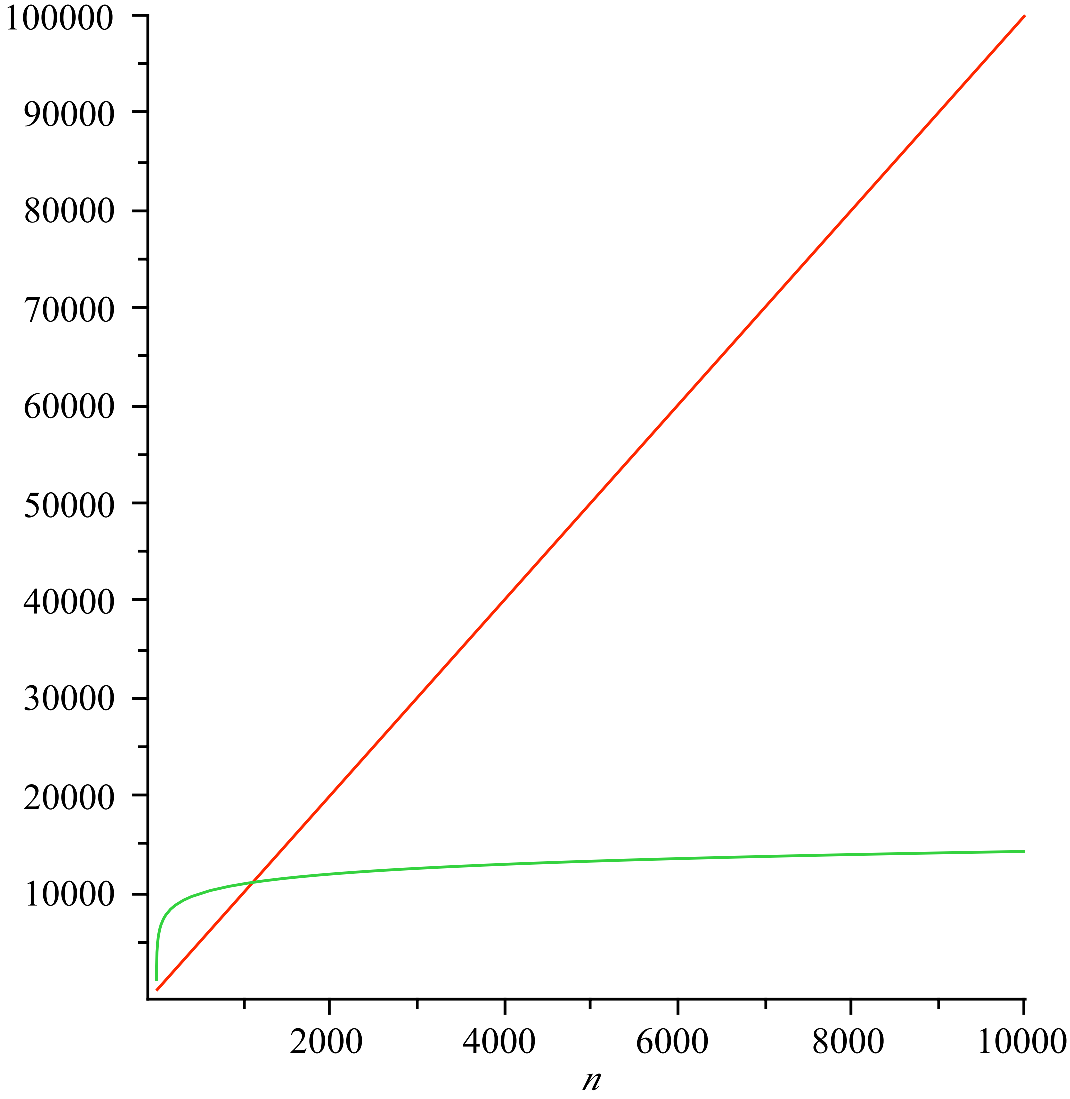

Figure 1.14. The graph of $T_{\mathsf{linSearch}}(n) = 10n + 10$ and $T_{\mathsf{binarySearch}}(n) = 1000\lg(n) + 1000$.

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.

size	wc linS	avc linS	wc binS	avc binS
10	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
100	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
1000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
10000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
100000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
200000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
400000	3 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
600000	3 ms	1.3 ms	≤ 1 ms	≤ 1 ms
800000	3 ms	1.5 ms	≤ 1 ms	≤ 1 ms
1000000	5 ms	2.1 ms	≤ 1 ms	≤ 1 ms
2000000	7 ms	3.7 ms	≤ 1 ms	≤ 1 ms
4000000	12 ms	6.9 ms	≤ 1 ms	≤ 1 ms
6000000	24 ms	11.6 ms	≤ 1 ms	≤ 1 ms
8000000	24 ms	15.6 ms	≤ 1 ms	≤ 1 ms

Table 1.15. Running Times of linSearch and binarySearch

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[6] 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: we either succeed or have dismissed around half of the current data with the other half remaining to be examined

Clearly for both algorithms the worst case is that we do not succeed early. Now linSearch plods its way dismissing one item at a time while binarySearch dismisses half the data (and does so *without doing any more work* than linSearch when dismissing the current item under consideration). An intuitive understanding such as this underpins the design of new efficient algorithms.

[6]Given by `java.lang.System.currentTimeMillis()`.

Time	$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 1000$
$n \lg n$	33 μs	86 μs	282 μs	664 μs	10 ms
n^2	100 μs	400 μs	3 ms	10 ms	1 s
n^5	100 ms	3 s	5 mins	3 hrs	32 yrs
2^n	1 ms	1 s	36 yrs	4×10^{16} yrs	
$n!$	4 s	774 cents			

Figure 1.16. Time required to solve problems of size n with an algorithm of the given runtime (algorithm runtime in μs, i.e., 10^{-6} seconds)

The preceding discussion shows that the replacement of the factor n in the worst case runtime of linSearch with $\lg n$ in the worst case runtime of 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 $n \log n$. No improvement of hardware speed can match this, since such an improvement is by some constant factor.

1.4 The need for efficient algorithms

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

With the impressive advances in hardware it might be tempting to assume that this will resolve the issue of efficiency. Figure 1.17 shows that this approach cannot work; the 1990 date has been left because it underlines the fact that the point being made is independent of the dates involved. Note, in particular the 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

[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.

	Fastest comp. in 1990	Comp. 1000 times faster
$n \lg n$	1.5 trillion	1000 trillion
n^2	8 million	260 million
n^5	570	2300
2^n	46	56
$n!$	16	18

Figure 1.17. Largest problem sizes solvable in 1 minute

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 have $n = 100$ or indeed much more (i.e., the algorithm could be expected to sort 100 objects). Let's consider trying to obtain a reliable estimate for the runtime of an algorithm just on this size of input (to keep things simple we will assume that each input consists of the integers $1, 2, \ldots, 100$ in some order). We will be generous and accept the timings based on just 1% of all possible inputs (of course this is very unreliable for worst case runtimes[8]). How many experiments does this imply? There are $n!$ different ways of ordering n distinct objects so this is the number of possible inputs. Thus we must carry out $n!/100$ experiments, i.e., $99!$ experiments. A direct calculation shows that

$$\begin{aligned} 99! = {} & 9332621544394415268169923885626670049071596826438162146859296389 \\ & 5217599993229915608941463976156518286253697920827223758251185210 \\ & 91686400000000000000000000000. \end{aligned}$$

Now even if the algorithm could sort 10^{50} instances per second(!) the experiment

[8]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.

would take at least

$$\frac{99!}{60 \times 60 \times 24 \times 366 \times 10^{50}} \approx 2.951269209 \times 10^{98}$$

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. However it is always essential to have a clear idea about the significance of any conclusions reached. When coupled with a good analysis experiments are an invaluable addition to our knowledge.

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.

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

Algorithm 1.18

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

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

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

$$T_{\mathsf{A}}(n) \geq n.$$

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 programing language, no rigid syntax for a start!).