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 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.

Prerequisites. You have seen a few algorithms and data structures in Informatics 1B. Amongst these are sorting algorithms (MergeSort and Selection Sort); you should also understand the principle of recursion. Arrays, lists, and trees were also introduced in Inf1B. If for some reason you didn't take Inf1B, then you should check the notes on these topics.

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.

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).

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. 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. 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.

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. 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. 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 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 Informatics 1B. 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.

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.

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.

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

```
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

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[1].

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 $i = 0, \dots, 7$). Line 2 is also executed 8 times. Line 3 is executed once, and Line 4 is never executed. Thus

[1]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.

```
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

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 $A = \langle 2,4,3,3,5,6,1,0,9,7 \rangle$ 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 $\{0, 1, 2, 3, \dots\}$ of natural numbers (including 0)[2].

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

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

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. 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 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

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

and if k does not appear at all it is

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

Thus the time complexity of linSearch is

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

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 1.9. Algorithm 1.10 is the pseudo-code of an algorithm that searches for an integer in a sorted array of integers using *binary search*. Note that

```
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 1.10

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[3] 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 of A as before. Then we can estimate the time complexity of binarySearch as follows:

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

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

[3]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.

at most one recursive call, either in Line 8 or in Line 10. Observing that

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

and

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

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 $T_{\textsf{binarySearch}}(\lfloor n/2\rfloor)$. Note that we have assumed that $T_{\textsf{binarySearch}}$ 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

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

for all $n\in\mathbb{N}$ with $n\ge1$ (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

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

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

For the inductive step, let $n\ge2$. We assume that we have already proved (1.2) for all $n'<n$ and now want to prove it for n. We have

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

This proves (1.2).

Remark 1.11. 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:

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

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

$$T_{\textsf{binarySearch}}(n)\le c\lg(n)+d$$

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 1.12 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 1.13 shows the graph of the two functions up to $n=10000$ (going

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

much higher tends to conceal the crossover point).

The message of Table 1.12 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*.

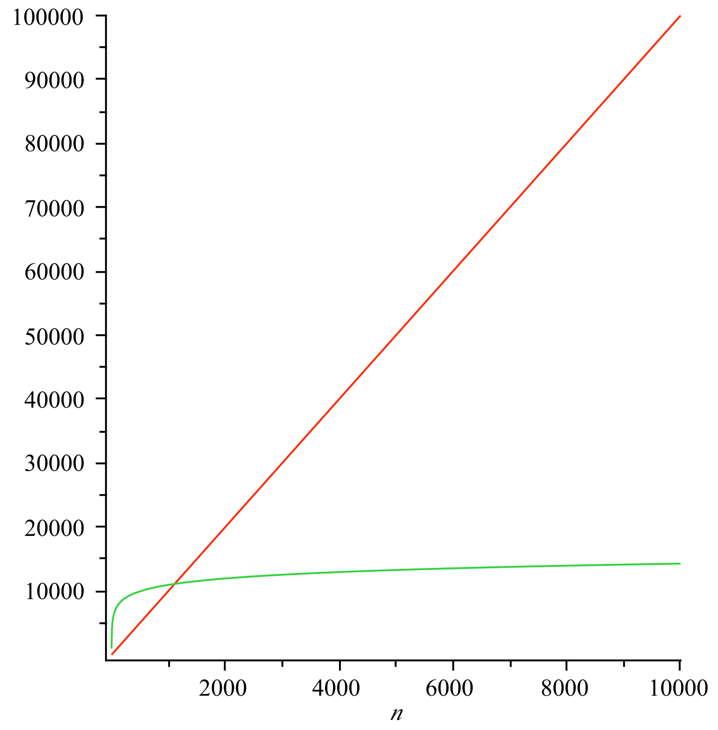

Figure 1.13. The graph of $T_{\mathsf{linSearch}}(n) = 10n + 10$ and $T_{\mathsf{binarySearch}}(n) = 1000\lg(n) + 1000$.

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 1.12 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.

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 1.14. 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[4] 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

[4]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 1.15. Time required to solve problems of size n with an algorithm of the given runtime (algorithm runtime in μs, i.e., 10^{-6} seconds)

when dismissing the current item under consideration). An intuitive understanding such as this underpins the design of new efficient algorithms.

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[5] but in many applications the size of the data is immense. Figure 1.15 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 1.16 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.

[5]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 1.16. Largest problem sizes solvable in 1 minute

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 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[6]). 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 $n!/100$ expriments, i.e.,

[6]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.

99! experiments. A direct calculation shows that

$$\begin{aligned}99! = {} & 93326215443944152681699238856266700490715968264381621468592963895\\ & 2175999932299156089414639761565182862536979208272237582511852109\\ & 168640000000000000000000000.\end{aligned}$$

Now even if the algorithm could sort 10^{50} instances per second(!) the experiment 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 1.17) 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 1.17

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 1.10 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!).

Asymptotic Growth Rates and the "Big-O" Notation

In the first lecture of this thread we defined the worst-case running time of an algorithm, and we saw how to determine this for an algorithm by analysing its (pseudo) code. We discussed the fact that if we want to abstract away from factors such as the programming language or machine used (which might change every couple of years), then we can at best expect to be able to determine the running time up to a constant factor.

In this lecture we introduce a notation which allows us to determine running time without keeping track of constant factors[1]. This is called *asymptotic notation*, and it captures how the running time of the algorithm grows with the size of the input. Asymptotic notation is a basic mathematical tool for working with functions that we only know up to a constant factor. There are three types of asymptotic notation that are commonly used in algorithmic analysis, $O(\cdot)$ ("Big-O"), $\Omega(\cdot)$ ("Omega") and $\Theta(\cdot)$ ("Theta"); the dot in the parentheses indicates that we would normally have something in that position (a function).

This note is organised as follows. We begin by introducing the most commonly used asymptotic symbol "big-O" in §2.1, in the context of functions in general rather than just runtimes. We give some examples (again just working with functions) and give some laws for working with $O(\cdot)$. In §2.2, we introduce our two other expressions $\Omega(\cdot)$ and $\Theta(\cdot)$, again in the context of functions. In §2.3 we explain how O, Ω and Θ are used to bound the worst-case running time of Algorithms and for basic Data Structure operations. Note that in Inf2B, we almost always work with worst-case running time, apart from our treatment of Dynamic Arrays. For these we perform an *amortized* analysis, where we bound the running time of a series of operations.

Before going into details for the various notations it is worth stating that like most powerful tools they take a little getting used to. However the relatively small effort required is more than amply repaid; these notations help us to avoid extremely fiddly detail that is at best tedious and at worst can prevent us from seeing the important part of the picture. The analysis of algorithms is greatly eased with the use of these tools; learning how to use them is an essential part of the course.

Finally there is a supplement at the end this note that places some matters in a general setting and should help with some misconceptions that have happened in the past (arising from a lack of separation of concerns).

2.1 The "Big-O" Notation

$\mathbb{R}$ denotes the set of real numbers.

Definition 2.1. Let $f, g : \mathbb{N} \to \mathbb{R}$ be *functions* We say that f is $O(g)$, pronounced *f is big-O of g*, if and only if there is an $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$

[1]Most of you will have seen this before, however there might be some joint degree students who have not done so. As we will be using this notation throughout the course it is essential to ensure that all have met it.

we have

$$0 \leq f(n) \leq cg(n).$$

Aside: A variant of the definition you might come across is to use absolute values for comparing the values of the functions, i.e., the required inequality is $|f(n)| \leq |g(n)|$. This has the virtue of ensuring non-negativity but forces us to work with absolute values all the time. In the end there is very little difference, the definition used in these notes is the same as the one in CLRS (see Note 1). The concern over non-negativity has a simple explanation: $-1000000 < 1$ but in terms of *size* (e.g., number of bits required to encode a number) we want to say that -1000000 is bigger than 1; comparing absolute values does the job. [Exercise for the dedicated: recall that a rational number is the ratio of two integers (a fraction), can we keep things so simple if we want to consider encoding size for rational numbers?]

Returning to the main definition, there is a slight nuisance arising from the fact that $\lg(n)$ is not defined for $n = 0$ so strictly speaking this is not a function from $\mathbb{N}$. There are various ways to deal with this, e.g., assign some arbitrary value to $\lg(0)$. We will never need to look at $\lg(0)$ so we will just live with the situation. Note also that the definition would make sense for functions $\mathbb{R} \to \mathbb{R}$ but as we will be studying runtimes measured in terms if input size the arguments to our functions are always natural numbers.

The role of n_0 can be viewed as a settling in period. Our interest is in long term behaviour (for all large enough values of n) and it is often convenient to have the freedom that n_0 allows us so that we do not need to worry about some initial atypical behaviour[2]. See Figure 2.2 for an illustration of this point.

Note that we also insist that the functions take on non-negative values from n_0 onwards. This condition is necessary to ensure that certain desirable properties hold. The fundamental issue is the following: if we have $a_1 \leq b_1$ and $a_2 \leq b_2$ then we can safely deduce that $a_1 + a_2 \leq b_1 + b_2$. Can we also deduce that $a_1 a_2 \leq b_1 b_2$? An example such as $-4 \leq 1$ and $-1 \leq 2$ but $-4 \times -1 \not\leq 1 \times 2$ shows that care must be taken. In fact the deduction we want to make is safe provided we have the extra information that $0 \leq a_1 \leq b_1$ and $0 \leq a_2 \leq b_2$ then it is indeed the case that $0 \leq a_1 a_2 \leq b_1 b_2$. From the point of view of our intended application area, runtime functions, the condition will be automatically true since runtimes are never negative. However we will be illustrating some points, and stating properties, with general mathematical functions and it is for this reason that the condition is there. Apart from a few early examples we will take it for granted and never check it for actual runtimes; indeed even in most of our general examples we have $f(n) \geq 0$ for all n so we just need to find n_0 and $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

To understand the role of c let's consider the two functions $f(n) = 10n$ and $g(n) = n$. Now these functions have exactly the same rate of growth: if we multiply n by a constant a then the values of $f(n)$ and $g(n)$ are both multiplied by a. Yet we have $f(n) > g(n)$ for all $n > 0$. However choosing $c = 10$ (or any larger value)

[2]In terms of the runtime of algorithms, we typically have some set up cost of variables and data structures which dominates things for small size inputs but is essentially negligible for large enough inputs where we see the real trend.

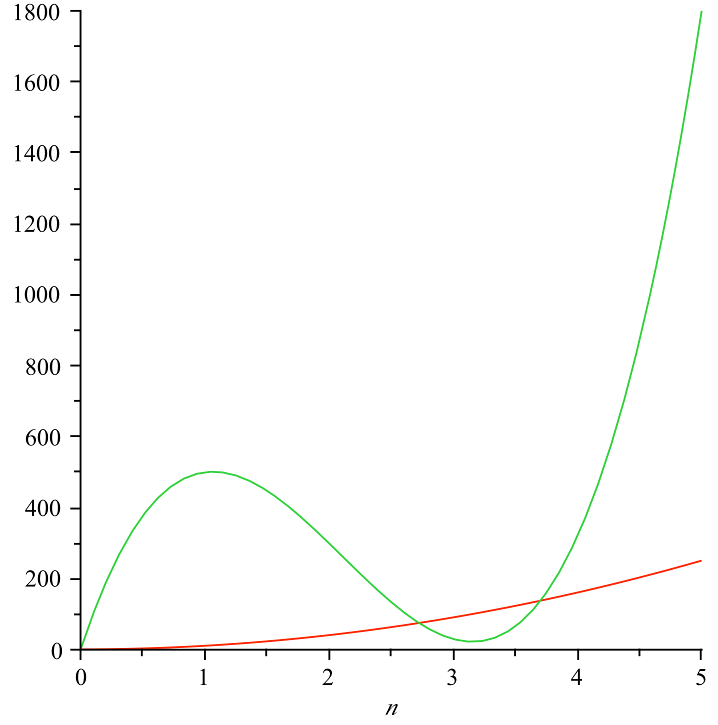

Figure 2.2. A graph of $f = 105n^3 - 665n^2 + 1060n$ and $g = 10n^2 + 1$. The graph illustrates the fact that $g(n) < f(n)$ for all $n > 4$ (indeed before that). So in verifying $g = O(f)$ we could take $n_0 = 4$, no need to make life hard by trying to find its exact value!

in the definition, we see that f is $O(g)$. Of course g is $O(f)$ as well but this is not always the case.

An informal (but useful) way of describing f is $O(g)$ is:

> *For large enough n, the rate of growth (w.r.t. n) of $f(n)$ is no greater than the rate of growth of $g(n)$.*

This latter informal description is useful, because it reminds us that if f is $O(g)$ it does *not* necessarily follow that we have $f(n) \leq g(n)$, what *does* follow is that the rate of growth of f is bounded from above by the rate of growth of g.

More formally, we define $O(g)$ to be the following set of functions:

$$O(g) = \big\{ f : \mathbb{N} \to \mathbb{R} \;\big|\; \text{there is an } n_0 \in \mathbb{N} \text{ and } c > 0 \text{ in } \mathbb{R} \text{ such that for all } n \geq n_0 \text{ we have } 0 \leq f(n) \leq cg(n). \big\}$$

Then f is $O(g)$ simply means $f \in O(g)$.

It is worth making another observation here. If we say that f is a function then f is the *name* of the function and does not denote any value of the function. To denote a value we use $f(n)$ where n denotes an appropriate value in the domain of the function[3]. Thus writing $f \leq g$ is meaningless unless we have defined some method of comparing functions as a whole rather than their values (we will not do this).

Notational convention. As we have seen, for a function g the notation $O(g)$ denotes a *set* of functions. Despite this it is standard practice to write $f = O(g)$ rather than $f \in O(g)$. This is in fact a very helpful way of denoting things, provided we bear in mind the meaning of what is being said. This convention enables us to pursue with ease chains of reasoning such as:

$$\begin{aligned} 871n^3 + 13n^2\lg^5(n) + 18n + 566 &= 871n^3 + 13n^2 O(n) + 18n + 566 \\ &= 871n^3 + O(n^3) + 18n + 566 \\ &= O(n^3) + O(n^3) + O(n^3) + O(n^3) \\ &= O(n^3). \end{aligned}$$

(Don't worry just now about the details, we will see this example later with full explanations.) To clarify, the first equality asserts that there is a function $g \in O(n)$ such that $871n^3 + 13n^2\lg^5(n) + 18n + 566 = 871n^3 + 13n^2 g(n) + 18n + 566$ (this is because it can be shown that any *fixed* power of $\lg(n)$ is $O(n)$). The second line then asserts that whatever function h we have from $O(n)$ the function $13n^2h(n)$ is $O(n^3)$. Looking now at the final equality, the assertion is that for any functions g_1, g_2, g_3, g_4 all from $O(n^3)$ we have that their sum (i.e., the function whose value at n is $g_1(n) + g_2(n) + g_3(n) + g_4(n)$) is again $O(n^3)$.

In our discussion we have also followed standard practice and suppressed the argument of a function, writing f rather than $f(n)$, whenever we do not need

[3]It is also common practice to use $f(n)$ as the name of the function when n is a variable in order to indicate the variable and that f is a univariate function. The point is that f by itself can never denote a value of the function and should never be used where a value is intended. Despite this, significant numbers of students persist in the meaningless practice and lose marks in exams as a result.

to stress it in any way. Mathematical notation is precise but is in fact closer to a natural language than non-mathematicians seem to imagine. Provided the conventions are understood a well established notation aids comprehension immensely. Judge for yourself which of the following is easier to understand and remember (they state the same thing):

- If $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$ then $f_1(n) + f_2(n) \in O\big(g_1(n) + g_2(n)\big)$.
- If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 + f_2 = O(g_1 + g_2)$.

Just one final word of warning about our usage. When we write $f = O(g)$ the equality is to be read in a directed way from left to right. Remember that what this really says is that $f \in O(g)$. This is of course at variance with the normal meaning of equality: $a = b$ means exactly the same as $b = a$. There is no possible confusion though if we bear in mind the intended meaning when a O appears, the gain is more than worth the need for interpreting equality in context.

Warning. The preceding discussion does *not* give you license to invent our own notational conventions (any more than speaking a language gives you license to invent arbitrary new words). Such conventions arise out of very long periods of usage with the useful ones being modified as necessary and finally adopted widely. A professional mathematician will introduce a new notational convention only after long thought and if it genuinely helps.

Mathematical writing. Here are some useful extracts from the notes for contributors to the London Mathematical Society:

(1) Organize your writing so that sentences read naturally even when they incorporate formulae.

(2) Formulae and symbols should never be separated merely by punctuation marks except in lists; one can almost always arrange for at least one word to come between different formulae.

(3) Draft sentences so that they do not begin with formulae or symbols.

(4) Never use symbols such as $\exists$ and $\forall$ as abbreviations in text.

In fact my view is that for the inexperienced user of Mathematics it is reasonable to say that excessive use of formal logical symbols often indicates a confusion of mind and represents the triumph of hope over understanding (based on the misguided belief that the symbols posess some kind of magic). Indeed for this course there is no need at all to use formal logical symbols such as $\exists$ and $\forall$ (with the exception of '$\Rightarrow$' and '$\Leftrightarrow$' which can be useful in a sequence of derivations—but again you must take great care to use them sensibly[4]). To put it briefly, a Mathematical argument must read fluently; the aim is to aid comprehension not

[4]I have seen, far too often, the use of these symbols where at least one side is not a statement! $P \Rightarrow Q$ and $P \Leftrightarrow Q$ are meaningless unless P and Q are both statements. It is also very common to misuse implication, getting it the wrong way round. To be precise if we have proved that $P \Rightarrow Q$ and that Q is true, we know *nothing* about P as a result of this reasoning. By contrast if we have proved that $P \Rightarrow Q$ and that P is true then we can deduce correctly that Q is also true. If, on the other hand, we know that Q is false then we can deduce that P is false.

to mask inadequate understanding. If your reasoning is faulty you are much more likely to spot this by expressing things clearly. This should come as no surprise, consider trying to understand and perhaps debug a well laid out computer program as opposed to one that is all over the place. The final presentation of your ideas should be *clear*, *concise* and *correct*.

Failure to ensure that a mathematical argument reads fluently with appropriate connecting and explanatory words accounts for a great number of common mistakes. Do not be misled by the fact that in many presentations (e.g., lectures) arguments are often sketched out. This is largely done to save time and the connecting material is usually presented orally rather than being written down. Naturally when we are first trying to find the solution to a problem we take such short cuts, the point is to work towards a final full presentation. Get into the habit of doing this if you have not already done so.

We end this digression by illustrating the need for clear precise language going hand in hand with clear precise thinking and understanding. Recall the key definition that is under discussion in this section:

> We say that f is $O(g)$ if and only if there is an $n_0 \in \mathbb{N}$ and a $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have $0 \leq f(n) \leq cg(n)$.

The wording is important. For example the phrase 'there is an $n_0 \in \mathbb{N}$ and a $c > 0$ in $\mathbb{R}$' tells us that for a given pair of functions f, g we must produce a *single choice* of n_0, and of c that do the required job (i.e., such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$). It would be wrong to start the definition with 'for all n there are $n_0 \in \mathbb{N}$ . . . ' This allows us to change n_0 (and c) with n which is certainly not the intention of the definition [would a definition that allowed us to change n_0 and c in this way but was otherwise as stated above be of any interest?]. When reading definitions (or any piece of mathematics) take care to understand such subtleties.

We now consider some examples where we just think about bounding functions in terms of other functions; this helps us to focus on understanding the notation rather than its applications. Later in this lecture note we will directly consider functions that represent the running-time of algorithms.

Examples 2.3.

(1) Let $f(n) = 3n^3$ and $g(n) = n^3$. Then $f = O(g)$.

PROOF: First of all we observe that $f(n) \geq 0$ for all n so we just need to find an n_0 and $c > 0$ such that $f(n) \leq g(n)$ for all $n \geq n_0$.

Let $n_0 = 0$ and $c = 3$. Then for all $n \geq n_0$, $f(n) = 3n^3 = cg(n)$.

Well that is clearly correct but how do we come up with appropriate values for n_0 and c? In this very simple case it is easy enough to see what values to choose. As a first illustration let's investigate what is needed for the claim to be true. We need to find $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. We have

$$3n^3 \leq cn^3 \Leftrightarrow 3 \leq c, \quad \text{provided } n > 0.$$

Here we are using the simple fact that if $ab \leq ac$ and $a > 0$ then we may divide out by a to obtain $b \leq c$. Conversely, if $a > 0$ (actually $a \geq 0$ is enough here) and $b \le c$ then we may multiply by a to obtain $ab \le ac$. (What goes wrong if we ignore the requirement that $a \ge 0$?) It follows that if we take $c = 3$ and $n_0 = 1$ the requirement for $f = O(g)$ is satisfied.

We note here that in the preceding argument we do not need the full equivalence $3n^3 \le cn^3 \Leftrightarrow 3 \le c$ it is enough to have $3 \le c \Rightarrow 3n^3 \le cn^3$. This shows that we could take $n_0 = 0$ though this is not of any importance here.

(2) $3n^3 + 8 = O(n^3)$.

PROOF: As above we have $3n^3 + 8 \ge 0$ for all n.

For this example we will give two proofs, using different constants for c, n_0. This is just to show that there are alternative constants that can be chosen (though only certain c, n_0 pairs will work).

First proof: let $c = 4$ and $n_0 = 2$. Then for all $n \ge n_0$ we have $n^3 \ge 8$ and thus $3n^3 + 8 \le 3n^3 + n^3 = 4n^3 = cn^3$.

Second proof: let $c = 11$ and $n_0 = 1$. Then for all $n \ge n_0$ we have $n^3 \ge 1$, and therefore $3n^3 + 8 \le 3n^3 + 8n^3 = 11n^3 = cn^3$.

Again let's investigate the situation to find values for c and n_0. For a constant $c > 0$ we have

$$3n^3 + 8 \le cn^3 \iff 3 + \frac{8}{n^3} \le c, \quad \text{provided } n > 0.$$

Now we note that as n increases $8/n^3$ decreases. It follows that

$$3 + \frac{8}{n^3} \le 11, \quad \text{for all } n > 0.$$

So if we take $c = 11$ and $n_0 = 1$ all the requirements are satisfied. In fact this derivation combines both of the previous proofs. If we take $n_0 = 2$ then $8/n^3 \le 1$ so that $3 + 8/n^3 \le 4$ for all $n \ge n_0$. Indeed we see that by taking n_0 sufficiently large we can use for c any value that is *strictly* bigger than 3 [can we use 3 as the value of c?].

(3) $\lg(n) = O(n)$

PROOF: Note that $\lg(n) \ge 0$ for all $n \ge 1$ (in any case we cannot have $n = 0$ as $\lg(0)$ is undefined), this tells us that our choice of n_0 must be at least 1 but we need to look at possible further requirements for the main inequality to hold.

Based on our discussion in Note 1 we would expect that in fact $\lg(n) < n$ for all $n \ge 1$. So we will try to prove this claim (in effect we are taking $n_0 = 1$ and $c = 1$). We have

$$\lg(n) < n \iff n < 2^n, \quad \text{for all } n > 0.$$

(To be formal we are using the fact that the exponentiation and logarithmic functions are strictly increasing in their argument. For the purposes of this proof all we need is that $n < 2^n \Rightarrow \lg(n) < n$.)

We prove that $2^n > n$ for all $n > 0$ by induction on n. The base case $n = 1$ is clearly true. Now for the induction step let us assume that the claim holds for n. Then

$$2^{n+1} = 2 \cdot 2^n > 2n,$$

where the inequality follows from the induction hypothesis. To complete the proof we need to show that $2n \ge n + 1$. Now

$$2n \ge n + 1 \iff n \ge 1,$$

and we have finished.

(4) $8n^2 + 10n\lg(n) + 100n + 10000 = O(n^2)$.

PROOF: As before $8n^2 + 10n\lg(n) + 100n + 10000 \ge 0$ for all n (we are lucky here because there are no negative coefficients).

We have

$$\begin{aligned} 8n^2 + 10n\lg(n) + 100n + 10000 &\le 8n^2 + 10n \cdot n + 100n + 10000, \quad \text{for all } n > 0 \\ &\le 8n^2 + 10n^2 + 100n^2 + 10000n^2 \\ &= (8 + 10 + 100 + 10000)n^2 \\ &= 10118n^2. \end{aligned}$$

Thus we can take $n_0 = 1$ and $c = 1118$.

The value for c seems rather large. This is irrelevant so far as the definition of big-O is concerned. However it is worth noting here that we could decrease the value of c at the expense of a relatively small increase in the value of n_0. We can illustrate this point with the graph in Figure 2.4. Indeed any $c > 8$ will do, the closer c is to 8 the larger n_0 has to be (see the discussion above for the second example). In the context of our intended usage of the big-O notation there is no point at all in expending more effort just to reduce some constant. In practice for runtimes the constants will depend on the implementation of the algorithm and the hardware on which it is run. We could determine the constants, if needed, by appropriate timing studies. What is independent of the factors mentioned is the growth rate and that is exactly what asymptotic notation gives us.

Warning. Graphs are very helpful in suggesting the behaviour of functions. Used carefully they can suggest the choice of n_0 and c when trying to prove an asymptotic relation. However they do *not* prove any such claim. There are at least two objections. Firstly all plotting packages can get things wrong (admittedly they are very reliable in straightforward situations). Secondly, and more seriously, a displayed graph can only show us a finite portion of a function's behaviour. We have no guarantee that if the graph is continued the suggested trend will not change (it is easy enough to devise examples of this). Like many tools they are great if used with appropriate care, dangerous otherwise.

(5) $2^{100} = O(1)$. That is, $f = O(g)$ for the functions defined by $f(n) = 2^{100}$ and $g(n) = 1$ for all $n \in \mathbb{N}$; both functions are constants.

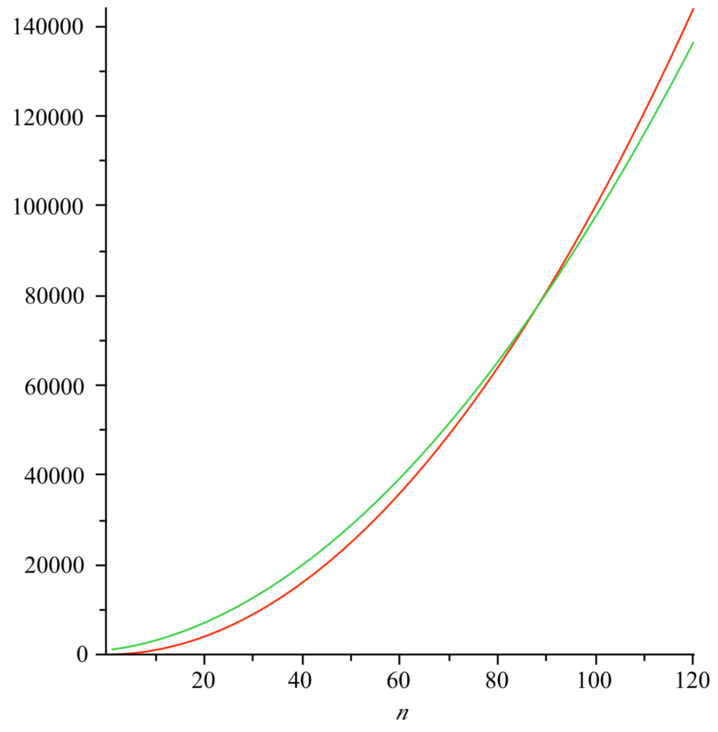

Figure 2.4. A graph of $8n^2 + 10n\lg(n) + 100n + 10000$ and $10n^2$.

PROOF: Let $n_0 = 1$ and $c = 2^{100}$.

It should be obvious that there is nothing special about 2^{100}, we could replace it by *any* non-negative constant and the claim remains true (with an obvious modification to the proof).

Note that all of the examples from (1) to (5) are proofs by first principles, meaning that we prove the "big-O" property using Definition 2.1, justifying everything directly. Theorem 2.5 lists some general laws, which can be generally used in simplifying "'big-O" expressions, and which make proofs of "big-O" shorter and easier.

Theorem 2.5. *Let $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}$ be functions, then*

(1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = O(g_1) \implies af_1 = O(g_1)$.

(2) $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 + f_2 = O(g_1 + g_2)$.

(3) $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 f_2 = O(g_1 g_2)$.

(4) $f_1 = O(g_1)$ and $g_1 = O(g_2) \implies f_1 = O(g_2)$.

(5) For any $d \in \mathbb{N}$: if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = O(n^d)$.

(6) For any constants $a > 0$ and $b > 1$ in $\mathbb{R}$: $n^a = O(b^n)$.

(7) For any constant $a > 0$ in $\mathbb{R}$: $\lg(n^a) = O(\lg(n))$.

(8) For any constants $a > 0$ and $b > 0$ in $\mathbb{R}$: $\lg^a(n) = O(n^b)$.

Note that $\lg^a(n)$ is just another way of writing $(\lg(n))^a$.

The following example shows how the facts of Theorem 2.5 can be applied:

Example 2.6. We will show that $871n^3 + 13n^2\lg^5(n) + 18n + 566 = O(n^3)$.

$$\begin{aligned}
871n^3 + 13n^2\lg^5(n) + 18n + 566 &= 871n^3 + 13n^2 O(n) + 18n + 566 && \text{by Theorem 2.5(8)}\\
&= 871n^3 + O(n^3) + 18n + 566 && \text{by Theorem 2.5(3)}\\
&= 871n^3 + 18n + 566 + O(n^3) && \\
&= O(n^3) + O(n^3) && \text{by Theorem 2.5(5)}\\
&= O(n^3) && \text{by Theorem 2.5(2)}
\end{aligned}$$

In §2.3 we will see how O is used in the analysis of the (worst-case) running time of algorithms.

We will not give the proof of every claim in Theorem 2.5. Most parts are straightforward consequences of the definition with (6) and (8) requiring extra facts. For illustration we will prove part (5). First recall that to say $f : \mathbb{N} \to \mathbb{R}$ is a polynomial function of degree d means that there are $a_0, a_1, \ldots, a_d \in \mathbb{R}$ with $a_d \neq 0$ such that

$$f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0,$$

for all $n \in \mathbb{N}$. (In this definition if we allow the possibility that $a_d = 0$ then the polynomial is of degree *at most* d; the proof is fine with this provided the actual leading coefficient is strictly positive[5].) It is very important to note that for a given polynomial d is a *constant*; it can be different for different polynomials but cannot vary once a polynomial is chosen (only n varies). Our statement assumes that $a_d > 0$, this is because if $a_d < 0$ the polynomial takes on negative values for all large enough n. So our task here is to find an n_0 and $c > 0$ such that

(1) $f(n) \geq 0$ and

(2) $f(n) \leq cn^d$

for all $n \geq 0$.

We will show the second of these properties here. The first one will follow from our discussion on p. 13; so the final choice of n_0 will be the maximum of the one found here and the one found later on. The claim is not automatically true, for example if $f(n) = n - 100$ then $f(n) < 0$ for all $n < 100$. Now

$$\begin{aligned} a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0 &\leq |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0| \\ &\leq |a_d|n^d + |a_{d-1}|n^d + \cdots + |a_1|n^d + |a_0|n^d \qquad \text{for all } n > 0 \\ &= (|a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|)n^d. \end{aligned}$$

So we can take $n_0 = 1$ and $c = |a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|$. There is a slight subtlety if we allow $a_d = 0$ since then we might have $|a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0| = 0$ but the definition of O requires that $c > 0$. No problem we just take $c = \max(1, |a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|)$ and then our proof works in all cases.

We can prove the claim in a slightly different way (essentially generalising the discussion of Example 2 on page 7). We have

$$a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0 \leq |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0|,$$

for all n. Now for all $n > 0$

$$\begin{gathered} |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0| \leq cn^d \iff \\ |a_d| + \frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d} \leq c \end{gathered}$$

The left hand side decreases as n increases. Taking $n_0 = 1$ gives us value for c we derived above (the same observation applies about the possibility that $a_d = 0$). By taking n_0 sufficiently large we can choose c to be as close to $|a_d|$ as we like (but not equal to it unless all the other coefficients are 0). You should think carefully about what can go wrong if we do not take the absolute values of the coefficients; when doing something like this simple examples can be very helpful. In this case consider a polynomial such as $n - 2$ and try the previous proofs without taking absolute values (you should find that in both cases things don't work out).

[5]By definition the leading coefficient of a polynomial is non-zero so in this context it would be enough to say that it is positive but we use 'strictly' to stress that it is non zero. For the sake of completeness it is worth pointing out that the zero polynomial (i.e., all coefficients are 0) has no degree and no leading coefficient, in contrast to all others. This is not worth worrying about for the applications of this course, no algorithm has 0 runtime!

2.2 Big-Ω and Big-Θ

"f is $O(g)$" is a concise and mathematically precise way of saying that, "up to a constant factor and for sufficiently large n, the function $f(n)$ grows at a rate no faster than $g(n)$". Sometimes, we also want to give *lower bounds*, i.e., make statements of the form "up to a constant factor, $f(n)$ grows at a rate at least as fast as $g(n)$". Big-Omega (written as Ω) is the analogue of big-O for this latter kind of statement.

Definition 2.7. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that f is $\Omega(g)$ if there is an $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have

$$f(n) \geq cg(n) \geq 0.$$

Informally, we say that f is big-Ω of g if there is some positive constant c such that for all *sufficiently large* n (corresponding to the n_0) we have $f(n) \geq cg(n) \geq 0$.

A more informal (but useful) way of describing $f = \Omega(g)$ is:

> *For large-enough n, the rate-of-growth (wrt n) of $f(n)$ is no less than the rate-of-growth of $g(n)$.*

Not surprisingly we can state laws for big-Ω that are similar to those for big-O:

Theorem 2.8. *Let $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}$ be functions, then*

(1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = \Omega(g_1) \implies af_1 = \Omega(g_1)$.

(2) $f_1 = \Omega(g_1)$ and $f_2 = \Omega(g_2) \implies f_1 + f_2 = \Omega(g_1 + g_2)$.

(3) $f_1 = \Omega(g_1)$ and $f_2 = \Omega(g_2) \implies f_1 f_2 = \Omega(g_1 g_2)$.

(4) $f_1 = \Omega(g_1)$ and $g_1 = \Omega(g_2) \implies f_1 = \Omega(g_2)$.

(5) For any $d \in \mathbb{N}$: if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = \Omega(n^d)$.

(6) For any constant $a > 0$ *in* $\mathbb{R}$*: $\lg(n^a) = \Omega(\lg(n))$.*

Compare this with Theorem 2.5 for big-O and note that two items from there do not have corresponding ones in the current theorem [why?]. Let's prove the claim of item 5, bearing in mind that in the following we are assuming $a_d > 0$.

$$a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0 \geq a_d n^d - |a_{d-1}|n^{d-1} - \cdots - |a_1|n - |a_0|,$$

for all n. (Note that we do not take the absolute value of a_d since we have assumed that $a_d > 0$.) For all $n > 0$

$$\begin{gathered} a_d n^d - |a_{d-1}|n^{d-1} - \cdots - |a_1|n - |a_0| \geq cn^d \iff \\ a_d - \left(\frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d}\right) \geq c \end{gathered}$$

Now $|a_{d-1}|/n + \cdots + |a_1|/n^{d-1} + |a_0|/n^d$ is always non-negative and is a decreasing function of n which tends to 0 as n increases. So there is some value n_0 of n such that

$$a_d > \frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d}$$

for all $n \geq n_0$. So we can take $c = a_d - (|a_{d-1}|/n_0 + \cdots + |a_1|/n_0^{d-1} + |a_0|/n_0^d)$ to satisfy the definition of big-Ω and we can use the n_0 we have already identified. Note that we do indeed have $c > 0$ as required owing to the assumption that $a_d > 0$ and the choice of n_0. Since $cn^d \geq 0$ for all n (and hence for all $n \geq n_0$ we are done. Note that we have proved here that $a_dn^d + a_{d-1}n^{d-1} + \cdots + a_1n + a_0 \geq 0$ for all $n \geq n_0$ just as promised on p. 11.

You might find the preceding proof less satisfactory than the corresponding one for big-O because there we could give the value of the constants directly in terms of the given coefficients[6]. We can do the same here: let $b = \max\{|a_{d-1}|, \ldots, |a_1|, |a_0|\}$ then

$$\begin{aligned}\frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d} &\leq b\left(\frac{1}{n} + \cdots + \frac{1}{n^{d-1}} + \frac{1}{n^d}\right)\\ &\leq \frac{bd}{n}.\end{aligned}$$

So if we ensure that $a_d > bd/n$ then we can take $c = a_d - bd/n$. Now $a_d > bd/n$ if and only if $n > bd/a_d$ (since $a_d > 0$) so we just need n to be at least as large as the next integer after bd/a_d. This is our value of n_0 and it gives us $c = a_d - bd/n_0$.

It is a good idea for you to consider why the claim is false (always) if $a_d \leq 0$. The fact that the preceding proof does not work is of course not enough to establish that the claim is false, maybe some other proof works (none does).

Finally, in this course our functions measure runtime and as a consequence the leading coefficient of any polynomial that we consider will necessarily be strictly positive. A polynomial with negative leading coefficient takes on negative values for all large enough n (indeed it goes to $-\infty$ as n goes to ∞), this gives a way to tackle the problem of the preceding paragraph).

We can combine big-O and big-Ω as follows:

Definition 2.9. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that f is $\Theta(g)$, or f *has the same asymptotic growth rate as* g, if f is both $O(g)$ and $\Omega(g)$.

Note that f is $\Theta(g)$ if and only if f is $O(g)$ and g is $O(f)$.

In the examples that follow we will just present the verification of each claim for the stated values of n_0 and c. Work out "investigation" type proofs as well.

Examples 2.10.

(1) Let $f(n) = 3n^3$ and $g(n) = n^3$. Then $f = \Omega(g)$.
(combining this with Example 2.3, (1), will give $3n^3 = \Theta(n^3)$)

PROOF: Let $n_0 = 0$ and $c = 1$. Then for all $n \geq n_0$, $f(n) = 3n^3 \geq cg(n) = g(n) \geq 0$.

[6]Note that the definition only requires that the constants exist, it does not ask for a method to find them. So as long as we prove their existence the definition is satisfied thus the first proof is perfectly OK.

(2) Let $f(n) = \lg(n)$ and $g(n) = \lg(n^2)$. Then $f = \Omega(g)$

PROOF: Let $n_0 = 1$ and $c = 1/2$. Then for every $n \geq n_0$ we have,

$$f(n) \;=\; \lg(n) \;=\; \frac{1}{2}2\lg(n) \;=\; \frac{1}{2}\lg(n^2) \;=\; \frac{1}{2}g(n).$$

The only interesting step above is the conversion of $2\lg(n)$ to $\lg(n^2)$. This follows from the well-known property of logs, $\log(a^k) = k\log(a)$.

(3) if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = \Theta(n^d)$.

PROOF: By Theorem 2.5 $f = O(n^d)$ while, by Theorem 2.8, $f = \Omega(n^d)$.

2.3 Asymptotic Notation and worst-case Running Time

Asymptotic notation seems well-suited towards the analysis of algorithms. The big-O notation, $O(\,\cdot\,)$, allows us to put an *upper bound* on the rate-of-growth of a function, and $\Omega(\,\cdot\,)$ allows us to to put a *lower bound* on the rate-of-growth of a function. Thus these concepts seem ideal for expressing facts about the running time of an algorithm. We will see that this is so, asymptotic notation is essential, but we need to be careful in how we define things. It is helpful to remind ourselves of the definition of *worst-case* running time, as this is our standard measure of the complexity of an algorithm:

Definition 2.11. 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 over all inputs of size n.

We will now also define a second concept, the *best-case* running time of an algorithm. We do *not* consider best-case to be a good measure of the quality of an algorithm, but we will need to refer to the concept in this discussion.

Definition 2.12. The *best-case running time* of an algorithm A is the function $B_{\mathsf{A}} : \mathbb{N} \to \mathbb{N}$ where $B_{\mathsf{A}}(n)$ is the minimum number of computation steps performed by A over all inputs of size n.

One way to use asymptotic notation in the analysis of an algorithm A is as follows:

- Analyse A to obtain the worst-case running time function $T_{\mathsf{A}}(n)$.
- Go on to derive upper and lower bounds on (the growth rate of) $T_{\mathsf{A}}(n)$, in terms of $O(\,\cdot\,)$, $\Omega(\,\cdot\,)$ and $\Theta(\,\cdot\,)$.

In fact such an approach would fail to capitalise on one of the most useful aspects of asymptotic notation. It gives us a way of focusing on the essential details (the overall growth rate) without being swamped by incidental things (unknown constants or sub-processes that have smaller growth rate).

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 2.13

Linear search

We now give our first example using the linSearch algorithm of Note 1. For this first example we will carry out the analysis by the two step process (since we have already carried out the first step).

Recall that we use small positive constants c_1, c_2, c_3, c_4 to represent the cost of executing line 1, line 2, line 3 and line 4 exactly once. The worst-case running time of linSearch on inputs of length n satisfies the following inequality:

$$(c_1 + c_2)n + \min\{c_3, (c_1 + c_4)\} \leq \ T_{\text{linSearch}}(n) \leq \ (c_1 + c_2)n + \max\{c_3, (c_1 + c_4)\}.$$

For linSearch, the best-case (as defined in Definition 2.12) will occur when we find the item searched for at the first index. Hence we will have $B_{\text{linSearch}}(n) = c_1+c_2+c_3$.

Now we show that $T_{\text{linSearch}}(n) = \Theta(n)$ in two steps.

$O(n)$: We know from our analysis in Note 1 that

$$T_{\text{linSearch}}(n) \leq (c_1 + c_2)n + \max\{c_3, (c_1 + c_4)\}.$$

Take $n_0 = \max\{c_3, (c_1 + c_4)\}$ and $c = c_1 + c_2 + 1$ in the definition of $O(\cdot)$. Then for every $n \geq n_0$, we have

$$T_{\text{linSearch}}(n) \leq (c_1 + c_2)n + n_0 \leq (c_1 + c_2 + 1)n = cn,$$

and we have shown that $T_{\text{linSearch}}(n) = O(n)$.

$\Omega(n)$: We know from our Note 1 analysis that

$$T_{\text{linSearch}}(n) \geq (c_1 + c_2)n + \min\{c_3, (c_1 + c_4)\}.$$

Hence $T_{\text{linSearch}}(n) \geq (c_1 + c_2)n$, since all the c_i were positive. In the definition of Ω, take $n_0 = 1$ and $c = c_1 + c_2$, and we are done.

Finally by definition of Θ (in §2.2), we have $T_{\text{linSearch}}(n) = \Theta(n)$. We then say that the *asymptotic growth rate* of linSearch is $\Theta(n)$.

The significance of the preceding analysis is that we have a guarantee that the upper bound is not larger than necessary and at the same time the lower bound is not smaller than necessary. We will discuss this further in §2.3

Finally note just how tedious is the process of keeping track of unknown constants. Not only that but readability is severely hampered. This disappears as soon as we switch to full use of asymptotic notation. Let's illustrate the point with linSearch.

Upper bound. As before we let n be the length of the input array. Line 1 is executed at most $n = O(n)$ times. It controls the loop body on lines 2 and 3. Each of these lines costs a constant, so each costs $O(1)$, and of course line 1 itself costs $O(1)$ each time it is executed. So the overall cost here is $O(n)(O(1)+O(1)+O(1)) = O(n)O(1) = O(n)$. Line 4 is executed once and costs $O(1)$ and so the total cost of the algorithm is $O(n) + O(1) = O(n)$.

Note: Strictly speaking line 1 is executed at most $n+1$ times, the extra one being to discover that the loop variable is out of bounds (i.e., $i > A.length - 1$) in those cases where the integer k does not occur in the array. This possible final time does not cause the execution of the loop body and costs a constant. Since our runtimes will always be at least a non-zero constant the cost contributed by such extra executions of loop control lines would be absorbed in the overall runtime (in terms of our analysis above the cost is $O(n) + O(1) = O(n)$). It follows that we can safely ignore this extra execution and we will do this from now on. A bit of thought shows that the same reasoning applies to nested loops as well. For the avoidance of doubt we will always count the cost of executions of a loop and its body but not the constant incurred in finding that a loop variable is out of bounds.

Lower bound. Consider any input where the integer k does not occur in the array. Then the condition in the loop is never true so we never execute the return in line 3. It follows that line 1 is executed at least $n = \Omega(n)$ times. Since each execution of the line costs $\Omega(1)$ the overall cost of this line alone is $\Omega(n)\Omega(1) = \Omega(n)$. Thus the cost of the algorithm is $\Omega(n)$.

Note: We did not bother to count the cost of lines 2 and 4 of the algorithm. This is because we already know the upper bound is $O(n)$ so once we have reached an $\Omega(n)$ lower bound we know that the rest will not increase this asymptotic cost. So why do the extra work? Let's be clear: counting the extra lines would of course have an effect on the *precise* runtime but the growth rate remains the same.

Misconceptions about O, Ω

This section is aimed at dispelling two common misconceptions, *both* of which arise from over-interpreting O's significance as an "upper bound" and Ω's significance as a "lower bound". Figure 2.14 concerning linSearch will be useful.

Misconception 1: If for some algorithm A we are able to show $T_{\mathsf{A}}(n) = O(f(n))$ for some function $f : \mathbb{N} \rightarrow \mathbb{R}$, then the running time of A is bounded by $f(n)$ for sufficiently large n.

FALSE: If $T_{\mathsf{A}}(n) = O(f(n))$, the *rate of growth* of the worst-case running time of A (with respect to the size n of the input) grows no faster than the *rate of growth* of f (wrt n). But $T_{\mathsf{A}}(n)$ may be larger than $f(n)$, *even* as n gets large—we are only

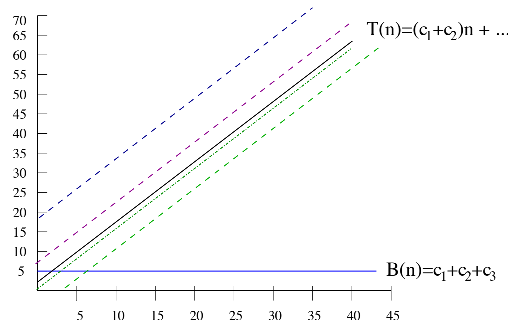

Figure 2.14. Best-case running time and worst-case running time for linSearch.

guaranteed that $T_{\mathsf{A}}(n) \le cf(n)$ for large n, where $c > 0$ is the constant used in proving that $T_{\mathsf{A}}(n)$ is $O(f(n))$.

As a concrete example of this, re-consider our analysis of the worst-case running-time of linSearch. We showed that $T_{\mathsf{linSearch}} = O(n)$ above. We never pinned down any particular values for c_1, c_2, c_3, c_4. However, whatever they are, we could have shown $T_{\mathsf{linSearch}} = O(\frac{1}{2}(c_1 + c_2)n)$ in *exactly* the same way as we showed $T_{\mathsf{linSearch}} = O(n)$ (but with a different value of c). However, it is certainly not the case that $T_{\mathsf{linSearch}}(n) \le \frac{1}{2}(c_1 + c_2)n$ for any n. In fact for any constant $\alpha > 0$, we could have shown $T_{\mathsf{linSearch}} = O(\alpha n)$. To see this graphically, look at Figure 2.3, where the various lines surrounding the "worst-case" line, both above and below the line for $T_{\mathsf{linSearch}}$ all represent potential $f(n)$ functions for $T_{\mathsf{linSearch}}(n)$ (there are infinitely many such functions).

Misconception 2: We know that the $T_{\mathsf{A}}(n) = O(f(n))$ implies that $cf(n)$ is an "upper bound" on the running time of the algorithm A on *any* input of size n, for some constant $c > 0$. This fact can sometimes mislead newcomers into believing that the result $T_{\mathsf{A}}(n) = \Omega(g(n))$ implies a lower bound on the running-time of A on all inputs of size n (with some constant being involved), which is **FALSE**. This confusion arises because there is an asymmetry in our use of O and Ω, when we work with *worst-case* running time.

O: Suppose we know that $T_{\mathsf{A}}(n) = O(f(n))$ for some function $f : \mathbb{N} \to \mathbb{R}$. Then we know that there is a constant $c > 0$ so that $T_{\mathsf{A}}(n) \le cf(n)$ for all sufficiently large n. Hence, because $T_{\mathsf{A}}(n)$ is the worst-case running time, we know that $cf(n)$ is an upper bound on the running time of A for *any input of size* n.

Ω: Suppose we know that $T_{\mathsf{A}}(n) = \Omega(g(n))$ for some function $g : \mathbb{N} \to \mathbb{R}$. Well, we know there is a constant $c' > 0$ such that $T_{\mathsf{A}}(n) \ge c'g(n)$ for all sufficiently large n. This part of the argument works fine. However, $T_{\mathsf{A}}(n)$ is the *worst-case running time*, so for any n, there may be many inputs of size n on which A takes far less time than on the worst-case input of size n. So the $\Omega(g(n))$ result does *not* allow us to determine any lower bound at all on *general* inputs of size n. It only gives us a lower bound on worst-case running time, i.e., there is at least one input of size n on which the algorithm takes time $c'g(n)$ or more.

As a concrete example of this, note that in Figure 2.3, any of the parallel lines to $T_{\mathsf{linSearch}}(n)$ are potential $\Omega(\cdot)$ functions for $T_{\mathsf{linSearch}}(n)$. These lines are far above the line for the best-case of linSearch. In fact *every* function f satisfying $T_{\mathsf{linSearch}}(n) = \Theta(f(n))$ (including those parallel lines to $T_{\mathsf{linSearch}}(n)$) is guaranteed to overshoot the line for $B_{\mathsf{linSearch}}(n)$ when n gets large enough.

Note that the asymmetry is not in the definitions of O and Ω but in the use to which we put them, i.e., producing bounds on the worst case running time of algorithms. To illustrate this point, suppose we have a group of people of various heights and the tallest person has height 1.9m. Then any number above 1.92 is an upper bound on their height but contains less information than the exact one of 1.92. If we want to be sure that 1.92 is as good as we can get then all that is necessary is to produce at least one person whose height is at least 1.92m. Note the asymmetry: for the upper bound *all* persons must satisfy it, for the lower bound *at least one* must do so.

Why bother with Ω? Is O not enough?

Why use $\Omega(\cdot)$ to bound $T_{\mathsf{A}}(n)$, when upper bounds are what we really care about? We can explain this very easily by looking at linSearch again. In §2.3 we first proved that $T_{\mathsf{linSearch}}(n) = O(n)$. However, we would have had no trouble proving $T_{\mathsf{linSearch}}(n) = O(n\lg(n))$ or $T_{\mathsf{linSearch}}(n) = O(n^2)$. Note that n, $n\lg(n)$ and n^2 differ by significantly more than a constant. So the following question arises: *How do we know which* $f(n)$ *is the "truth", at least up to a constant*? We know it cannot be $n\lg(n)$ or n^2, but how do we know it is not a sub-linear function of n? The answer is that if we can prove $T_{\mathsf{A}}(n) = O(f(n))$ and $T_{\mathsf{A}}(n) = \Omega(f(n))$ (as we did for linSearch), then we know that $f(n)$ is the true function (up to a constant) representing the rate-of-growth of $T_{\mathsf{A}}(n)$. We then say that $f(n)$ is the *asymptotic growth rate* of the (worst case) runtime of A. The true growth rate of the worst-case running time of linSearch was pretty obvious. However for other more interesting algorithms, we will only be sure we have a "tight" big-O bound on $T_{\mathsf{A}}(n)$ when we manage to prove a matching Ω bound, and hence a Θ bound.

This is perhaps the place to mention another fairly common error[7]. Suppose we have an algorithm consisting of some loops (possibly nested) from which there is no early exit and we produce an upper bound of, say, $O(n^2)$. This does *not* entitle us to claim that $\Omega(n^2)$ is also a lower bound. The 'reasoning' given for such a fallacious claim is that as there is no early exit the algorithm always does the same amount of work (true) and so the lower bound follows. This cannot possibly

[7] I am aware that quite a few such misconceptions have been pointed out in this note and there a danger of overdoing things. However each misconception discussed has occurred regularly over the years, e.g., in exam answers. The common thread is a failure to understand or apply the definitions rigorously. Naturally there will be many readers who will have grasped the essential points already; unfortunately there is no way to produce notes that modify themselves to the exact requirements of the reader!

be a correct inference: if $O(n^2)$ is an upper bound then so are $O(n^3)$, $O(n^4)$ etc. If the inference were correct it would entitle us to claim a lower bound $\Omega(n^d)$ for any $d \geq 2$! Clearly false. A genuine proof of a lower bound $\Omega(n^2)$ reassures us that in deriving the upper bound we did not over estimate things.

Note that for some algorithms, it is not possible to show matching $O(\cdot)$ and $\Omega(\cdot)$ bounds for $T_A(n)$ (at least not for a smooth function f). This is not an inherent property of the algorithms, just a consequence of our limited understanding of some very complicated situations.

Typical asymptotic growth rates

Typical asymptotic growth rates of algorithms are $\Theta(n)$ (*linear*), $\Theta\big(n\lg(n)\big)$ (*n-log-n*), $\Theta(n^2)$ (*quadratic*), $\Theta(n^3)$ (*cubic*), and $\Theta(2^n)$ (*exponential*). The reason why $\lg(n)$ appears so often in runtimes is because of the very successful divide and conquer strategy (e.g., as in binarySearch). Most textbooks contain tables illustrating how these functions grow with n (e.g., [GT] pp.19–20), see also Figures 2.15, 2.16.

2.4 The Running Time of Insertion Sort

We now analyse the *insertion sort* algorithm that you are likely to have seen elsewhere. The pseudocode for insertion sort is given below as Algorithm 2.17. The algorithm works by inserting one element into the output array at a time. After every insertion, the algorithm maintains the invariant that the $A[0\ldots j]$ subarray is sorted. Then $A[j+1]$ becomes the next element to be inserted into the (already sorted) array $A[1\ldots j]$.

Let the input size n be the length of the input array A. Suppose the inner loop in lines 4–6 has to be iterated m times. One iteration requires time $\big(O(1)+O(1)+O(1)\big) = O(1)$. Thus the total time required for the m iterations is

$$mO(1) = O(m).$$

Since m can be no larger than n, it follows that $O(m) = O(n)$ (i.e., any function that is $O(m)$ is necessarily $O(n)$, but *not* conversely—remember that we read equality only from left to right when asymptotic notation is involved). The outer loop in lines 1–7 is iterated $n-1$ times. Each iteration requires time $O(1)+O(1)+O(1)+O(n)+O(1) = O(n)$ (lines 1, 2, 3, and 7 cost $O(1)$ and the inner loop costs $O(n)$). Thus the total time needed for the execution of the outer loop and thus for insertionSort is

$$(n-1)O(n) = nO(n) = O(n^2).$$

Therefore,

$$T_{\mathsf{insertionSort}}(n) = O(n^2). \tag{2.1}$$

Note, once again, how the use of asymptotic notation has helped us to avoid irrelevant detail (such as giving names to unknown constants, or adding up several constants only to observe that this is another constant).

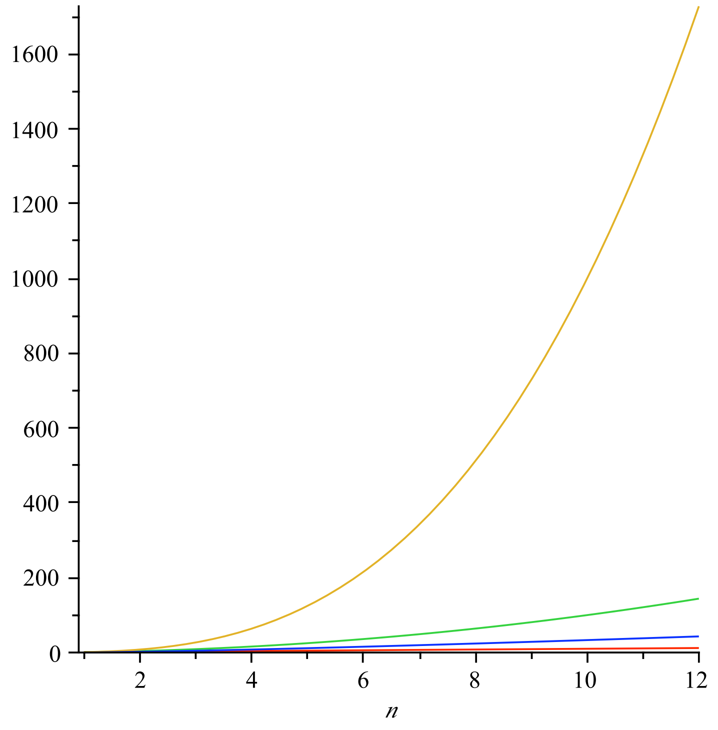

Figure 2.15. Graph of n, $n\lg(n)$, n^2 and n^3.

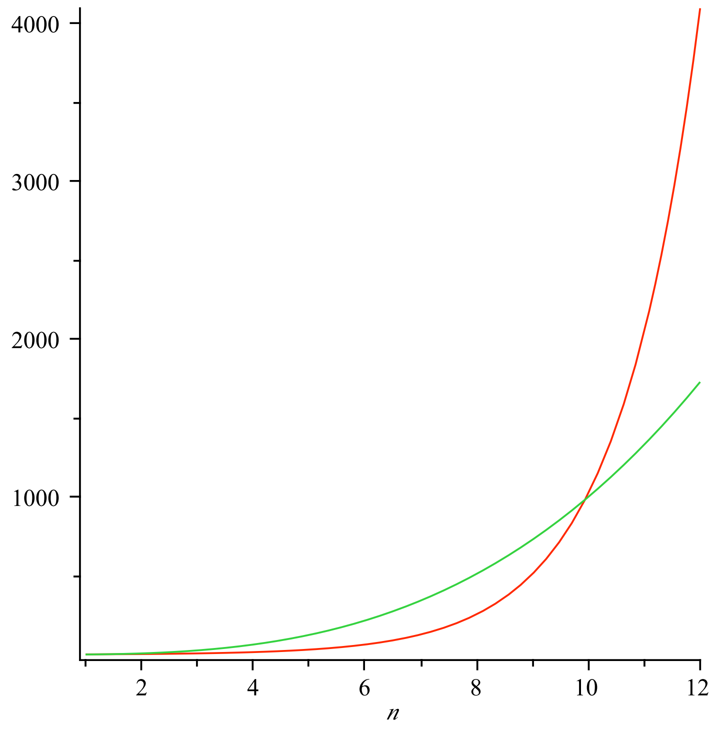

Figure 2.16. Graph of n^3 and 2^n.

Algorithm insertionSort(A)

Input: An integer array A

Output: Array A sorted in non-decreasing order

```
1. for $j \leftarrow 1$ to $A.length - 1$ do
2.     $a \leftarrow A[j]$
3.     $i \leftarrow j - 1$
4.     while $i \geq 0$ and $A[i] > a$ do
5.         $A[i+1] \leftarrow A[i]$
6.         $i \leftarrow i - 1$
7.     $A[i+1] \leftarrow a$
```

Algorithm 2.17

This tells us that the *worst-case running time* of insertion sort is $O(n^2)$. But for all we know it could be better[8]. After all , it seems as though we were quite careless when we estimated the number m of iterations of the inner loop by n; in fact, m is at most j when the $A[j]$ element is being inserted.

We will prove that

$$T_{\text{insertionSort}}(n) \geq \frac{1}{2}n(n-1). \tag{2.2}$$

Since this is a statement about the *worst-case* running time, we only need to find *one* array A_n of size n for each n such that on input A_n, insertionSort does at least $\frac{1}{2}n(n-1)$ computation steps. We define this array as follows:

$$A_n = \langle n-1, n-2, \ldots, 0\rangle.$$

Let us see how often line 5 is executed on input A_n. Suppose we are in the jth iteration of the outer loop. Since $A[j] = n - j - 1$ is smaller than all the elements in $A[0 \ldots j-1]$ (these are the elements $n-1, n-2, \ldots, n-j$, in sorted order) it follows that the insertion sort algorithm will have to walk down to the very start of $A[0 \ldots j-1]$, so that line 5 is executed j times when inserting $A[j] = n - j - 1$. Thus line 5 is executed j times for element $A[j]$. Thus overall line 5 is executed

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

times. Clearly, each execution of line 5 requires at least 1 computation step. Thus (2.2) holds.

Finally $n(n-1)/2 = \Omega(n^2)$ and so our $O(n^2)$ upper bound was not an overestimate in terms of growth rate. We have shown that $T_{\text{insertionSort}}(n) = \Theta(n^2)$.

[8]There is nothing deep going on here. To return to a variant of an earlier example, if a person is 5 feet tall (approximately 1.5 metres) then it is correct to say that his/her height is no greater than $5, 6, 7, 8, \ldots$ feet. Each number gives correct but less and less accurate information.

It is worthwhile returning to the point made above about our estimation of the number m of iterations of the inner loop by n. How would we have a sense that the estimation is not so crude as to lead to too big an upper bound? Well we observed that m is at most j when the $A[j]$ element is being inserted and a little thought shows that there are inputs for which it is this bad (e.g., the input array above used for the lower bound). Now for $j \geq n/2$ we now know that $m = j = \Omega(n/2) = \Omega(n)$. As there are around $n/2$ values of j with $j \geq n/2$ we expect a runtime of around $n/2\,\Omega(n) = \Omega(n^2)$, so our estimation of j as $O(n)$ doesn't look so careless in terms of the asymptotic analysis. None of this is a precise proof but is the sort of intuition that comes naturally with enough practice and can be turned into a precise proof as shown above. Developing your intuition in this direction is important as it often helps to avoid getting bogged down in unnecessary detail.

2.5 Interpreting Asymptotic Analysis

In Lecture Note 1 we argued that determining the running time of an algorithm up to a constant factor (i.e., determining its asymptotic running time) is the best we can hope for.

Of course there is a potential problem in this approach. Suppose we have two algorithms A and B, and we find out that $T_{\mathsf{A}}(n) = \Theta(n)$ while $T_{\mathsf{B}}(n) = \Theta\big(n\lg(n)\big)$. We conclude that A is more efficient than B. However, this does not rule out that the actual running time of any implementation of A is $2^{1000}n$, whereas that of B may only be $100n\lg(n)$. This would mean that for any input that we will ever encounter in real life, B is much more efficient than A. This simply serves to underline that fact that in assessing an algorithm we need to study it along with the analyisis. This will usually give us a good idea if the analysis has swept any enormous constants under the carpet (they are not likely to be introduced by the implementation unless it is extremely inept; alas this cannot be ruled out!). Finally we can implement (carefully) any competing algorithms and experiment with them.

2.6 Further Reading

If you have [GT], you might have edition 3 or edition 4 (published August 2005). You should read all of the chapter "Analysis Tools" (especially the "Seven functions" and "Analysis of Algorithms" sections). This is as Chapter 3 in Ed. 3, Chapter 4 in Ed. 4.

If you have [CLRS], then you should read Chapter 3 on "Growth of Functions" (but ignore the material about the $o(\cdot)$ and $\omega(\cdot)$ functions).

Exercises

1. Determine which of the following statements are true or false. Justify your answer.

   (1) $n^3 = O(256n^3 + 12n^2 - n + 10000)$.

   (2) $n^2\lg(n) = \Theta(n^3)$.

   (3) $2^{\lfloor\lg(n)\rfloor} = \Theta(n)$.

2. In your *Learning and Data* lectures of Inf2B, you have seen (or soon will see) procedures for estimating the mean and variance of an unknown distribution, using samples from that distribution. What is the asymptotic running time of these mean-estimation and variance-estimation procedures?

3. Suppose we are comparing implementations of five different algorithms $A_1, \ldots, A_5$ for the same problem on the same machine. We find that on an input of size n, in the worst-case:

   - A_1 runs for $c_1\,n^2 - c_2\,n + c_3$ steps,
   - A_2 runs for $c_4\,n^{1.5} + c_5$ steps,
   - A_3 runs for $c_6\,n\lg(n^3) + c_7$ steps,
   - A_4 runs for $2^{\lfloor n/3\rfloor} + c_8$ steps.
   - A_5 runs for $n^2 - c_9\,n\lg^4(n) + c_{10}$ steps.

   Give a simple Θ-expression for the asymptotic running time of each algorithm and order the algorithms by increasing asymptotic running times. Indicate if two algorithms have the same asymptotic running time.

4. Determine the asymptotic running time of the sorting algorithm maxSort (Algorithm 2.18).

```
Algorithm maxSort($A$)
  Input:  An integer array $A$
  Output: Array $A$ sorted in non-decreasing order

1.  for $j \leftarrow n-1$ downto $1$ do
2.      $m \leftarrow 0$
3.      for $i = 1$ to $j$ do
4.          if $A[i] > A[m]$ then $m \leftarrow i$
5.      exchange $A[m], A[j]$
```

Algorithm 2.18

Can you say anything about the "best-case" function $B_{\mathsf{maxSort}}(n)$?

5. Use two appropriate theorems to prove the following.

   (1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = \Theta(g_1) \implies af_1 = \Theta(g_1)$.

   (2) $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2) \implies f_1 + f_2 = \Theta(g_1 + g_2)$.

   (3) $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2) \implies f_1 f_2 = \Theta(g_1 g_2)$.

   (4) $f_1 = \Theta(g_1)$ and $g_1 = \Theta(g_2) \implies f_1 = \Theta(g_2)$.

Putting bounds on functions and runtimes—supplement

§**1. General setting.** This supplement discusses the process of putting upper and lower bounds from a slightly more general perspective. The material here is not new, it covers the same ground as the note but putting things this way might prove helpful to some. If you are already very clear about the issues reading this should be very quick, if this is not the case then draw your own conclusions!

Consider a function $F : \mathbb{N} \to \mathbb{R}$ defined by some means. Let's note in passing that the definition does not fix any method of computing $F(n)$ given n, it simply fixes a unique value by appropriate conditions[9]. We say that a function $U : \mathbb{N} \to \mathbb{R}$ is an *upper bound* for F if $F(n) \le U(n)$ for all n. Similarly a function $L : \mathbb{N} \to \mathbb{R}$ is a *lower bound* for F if $L(n) \le F(n)$ for all n. In many situations we don't mind if the inequalities fail to hold for some initial values as long as they hold for all large enough values of n (recall that this is the same as saying that there is an $n_0 \in \mathbb{N}$ such that the inequality holds for all $n \ge n_0$). So with this situation a lower and upper bound give us the information that

$$L(n) \le F(n) \le U(n) \qquad (\dagger)$$

for all large enough values of n. Note that the definition of a lower bound is entirely symmetrical to that of an upper bound, the only difference is the inequality.

There can be various reasons for wanting to find upper and lower bounds, for us the main one is because although we have a precise definition of F we cannot obtain an exact formula for it. If we have upper and lower bounds we are of course interested in how good they are. To illustrate the point suppose we are seeking to find information about some integer M. After some work we find that $0 \le M \le 1000000$. This certainly gives us information but of a rather imprecise nature because the lower and the upper bounds are very far apart. If we worked a bit more and found that $12 \le M \le 20$ we'd be in a better position. In terms of functions we want $L(n)$ and $U(n)$ to be as close as possible. Of course the ideal is that $L(n) = U(n)$ in which case we know $F(n)$ but this is often not possible so we must settle for something less precise[10].

There are two further refinements we can make, we will consider the first one here and the second one later on. In this course we are interested in putting

[9]A frequent error is to talk about the runtime of a mathematical function. This is meaningless until we have chosen an algorithm to compute it and even then we are referring to the runtime of the algorithm. In any case it can be proved that for most functions there is no algorithm to compute them. (This is surprisingly easy given some fundamental notions.)

[10]In fact even when we know F explicitly we might be interested in bounding it from above and below by functions that are easier to work with and are still close enough to F.

bounds on functions whose values are non-negative and what is of interest is to bound the size of their values from above and below. So, e.g., if it so happens that (unknown to us) $F(n) = n$ it is not informative to produce the lower bound $-n^{10} \le F(n)$; anything negative is a lower bound. For this reason we amend (†) to

$$0 \le L(n) \le F(n) \le U(n) \qquad (\ddagger)$$

for all large enough values of n. (An alternative is to take absolute values throughout but this is notationally heavier and is essentially equivalent to our approach.)

§**2. Putting bounds on runtimes.** Suppose we have an algorithm $\mathcal{A}$. Recall that we assume we have a method of assigning a size to the inputs of $\mathcal{A}$ such that for a given size there are only finitely many possible inputs. Now let R_n denote the set of runtimes that result by running $\mathcal{A}$ on all inputs of size n. (Of course this set depends on $\mathcal{A}$ as well but we have not indicated this in the notation just to keep it simple, no confusion can arise as we will only be considering $\mathcal{A}$ in the discussion.) Since there are only finitely many inputs to $\mathcal{A}$ for any given n it follows that R_n is a finite set and so it has a maximum and a minimum member. Recall that we made the following definitions in the course:

(1) The *worst case runtime* of $\mathcal{A}$ is the function $W : \mathbb{N} \to \mathbb{R}$ defined by

$$W(n) = \max R_n.$$

(2) The *best case runtime* of $\mathcal{A}$ is the function $B : \mathbb{N} \to \mathbb{R}$ defined by

$$B(n) = \min R_n.$$

(We are not terribly interested in the best case runtime except for illustrative purposes.) It is trivially the case that

$$0 \le B(n) \le W(n).$$

In other words $B(n)$ is a lower bound for $W(n)$ and of course $W(n)$ is an upper bound for $B(n)$. However these can be too far apart to be of much use. For example we saw that for linear search the best case runtime is a constant while the worst case is proportional to n. You should also consider the best and worst case runtimes of insertion sort, again they are far apart. Of course there are algorithms for which $B(n)$ is very close or even equal to $W(n)$ and for these cases it is a good lower bound to $W(n)$. The point is that this is not even remotely true of all of algorithms so we cannot use $B(n)$ as a good lower bound to $W(n)$ automatically.

Now pick any $r \in R_n$. By the definition of our two functions we have

$$B(n) \le r \le W(n).$$

Spelling this out, if we pick *any* input to $\mathcal{A}$ of size n and find the runtime then this is an upper bound for the best case runtime and also a lower bound for the worst case. Once again note the symmetry.

Let us now focus on $W(n)$ for some fixed n. By definition, an upper bound $U(n)$ to $W(n)$ is anything that satisfies

$$\max R_n \leq U(n).$$

Note that we just demand a bound on a *single* element of the set R_n but as this is the largest element it follows automatically that it is an upper bound to *all* elements of R_n. So when putting an upper bound we are in fact just concerned with one element of R_n but we don't know its value. So how can we put an upper bound on it? Well if we find that for *all* inputs of size n the runtime is at most $U(n)$ then of course this claim is also true of $\max R_n$, i.e., of $W(n)$. In practice we consider a general input of size n and argue from the pseudocode that the algorithm runtime will be at most a certain function of n (e.g., if we have a loop that is executed at most n times and the body costs at most a constant c then the cost of the entire loop is at most cn). This process leaves open the possibility that we have overestimated by a significant amount so to check this we look for lower bounds to $W(n)$.

Let us now consider putting a lower bound on $W(n)$. By definition a lower bound $L(n)$ to $W(n)$ is anything that satisfies

$$0 \leq L(n) \leq \max R_n.$$

Once again our interest is in putting a bound on a single element of R_n. We could do this by considering all inputs of size n but for most algorithms this would lead to a severe under estimate (recall linear search). As observed above, if we find the runtime r for any particular input of size n then we have found a lower bound to $\max R_n$, i.e., to $W(n)$. So when trying to put a lower bound we look at the structure of the algorithm and try to identify an input of size n that will make it do the *greatest* amount of work.

There is now an asymmetry in what we do. However this arises from the definition of the function which we seek to bound, *not* from the notion of upper and lower bounds.

At this point you should consider what it means to put upper and lower bounds on $B(n)$. It should be clear that in putting a lower bound we consider all inputs of size n but for an upper bound we need only consider a single appropriate input of size n. In this second case we look at the structure of the algorithm and and try to identify an input of size n that will make it do the *least* amount of work. Thus the situation is a mirror symmetry of that for $W(n)$.

§**3. Asymptotic notation.** We discussed above one refinement to the notion of bounds. For the second refinement we allow the possibility of adjusting the values of U and L by some strictly positive multiplicative constants, i.e., we focus on growth rates. This can be for various reasons, e.g., the definition of F involves unknown constants as in the case of runtimes of algorithms. So we amend the inequalities (‡) in the definition to allow the use of constants $c_1 > 0$ and $c_2 > 0$ s.t.

$$0 \leq c_1 L(n) \leq F(n) \leq c_2 U(n),$$

for all large enough values of n. Note that this says *exactly* the same as that $F = \Omega(L)$ and $F = O(U)$. Once again we ask just how good are such bounds?

The best we could hope for is that $L = U$ and if this is so then we say that $F = \Theta(U)$. Of course even if this is so we do not know the value of $F(n)$, for large enough n, because of the multiplicative constants but we do have a very good idea of the growth rate. For the final time, the definitions of O and Ω are entirely symmetrical, any asymmetry in arguments involving them comes from the nature of the functions being studied.

Sequential Data Structures

In this lecture we introduce the basic data structures for storing *sequences* of objects. These data structures are based on *arrays* and *linked lists*, which you met in first year (you were also introduced to stacks and queues). In this course, we give abstract descriptions of these data structures, and analyse the asymptotic running time of algorithms for their operations.

3.1 Abstract Data Types

The first step in designing a data structure is to develop a mathematical model for the data to be stored. Then we decide which methods we need to access and modify the data. Such a model, together with the methods to access and modify it, is an *abstract data type (ADT)*. An ADT completely determines the functionality of a data structure (*what* we want it to), but it says nothing about the implementation of the data structure and its methods (*how* the data structure is organised in memory, or which algorithms we implement for the methods). Clearly we will be very interested in which algorithms are used but not at the stage of defining the ADT. The particular algorithms/data structures that get used will influence the running time of the methods of the ADT. The definition of an ADT is something done at the beginning of a project, when we are concerned with the *specification*[1] of a system.

As an example, if we are implementing a dictionary ADT, we will need to perform operations such as look-up(w), where w is a word. We would know that this is an essential operation at the specification stage, before deciding on data structures or algorithms.

A *data structure* for *realising* (or *implementing*) an ADT is a structured set of variables for storing data. On the implementation level and in terms of JAVA, an ADT corresponds to a JAVA interface and a data structure realising the ADT corresponds to a class implementing the interface. The ADT determines the functionality of a data structure, thus an algorithm requiring a certain ADT works *correctly* with any data structure realising the ADT. Not all methods are equally efficient in the different possible implementations, and the choice of the right one can make a huge difference for the efficiency of an algorithm.

3.2 Stacks and Queues

A *Stack* is an ADT for storing a collection of elements, with the following methods:

- push(e): Insert element e (at the "top" of the stack).
- pop(): Remove the most recently inserted element (the element on "top") and return it; an error occurs if the stack is empty.
- isEmpty(): Return TRUE if the stack is empty and FALSE otherwise.

[1]You will learn about specification in Software engineering courses.

A stack obeys the LIFO (*Last-In, First-Out*) principle. The *Stack* ADT is typically implemented by building either on *arrays* (in general these need to be *dynamic arrays*, discussed in 3.4) or on *linked lists*[2]. Both types of implementation are straightforward, and efficient, taking $O(1)$ time (the (dynamic) array case is more involved, see 3.4) for any of the three operations listed above.

A *Queue* is an ADT for storing a collection of elements that retrieves elements in the opposite order to a stack. The rule for a queue is FIFO (*First-In, First-Out*). A queue supports the following methods:

- enqueue(e): Insert element e (at the "rear" of the queue).
- dequeue(): Remove the element inserted the longest time ago (the element at the "front") and return it; an error occurs if the queue is empty.
- isEmpty(): Return TRUE if the queue is empty and FALSE otherwise.

Like stacks, queues can easily be realised using (dynamic) arrays or linked lists. Again, whether we use arrays or linked lists, we can implement a queue so that all operations can be performed in $O(1)$ time.

*Stack*s and *Queue*s are very simple ADTs, with very simple methods—and this is why we can implement these ADTs so the methods all run in $O(1)$ time.

3.3 ADTs for Sequential Data

In this section, our mathematical model of the data is a linear *sequence* of elements. A sequence has well-defined *first* and *last* elements. Every element of a sequence except the first has a unique *predecessor* while every element except the last has a unique *successor*[3]. The *rank* of an element e in a sequence S is the number of elements before e in S.

The two most natural ways of storing sequences in computer memory are *arrays* and *linked lists*. We model the memory as a sequence of memory cells, each of which has a unique *address* (a 32 bit non-negative integer on a 32-bit machine). An *array* is simply a contiguous piece of memory, each cell of which stores one object of the sequence stored in the array (or rather a reference to the object). In a *singly linked list*, we allocate two successive memory cells for each object of the sequence. These two memory cells form a *node* of a sequence. The first stores the object and the second stores a reference to the next node of the list (i.e., the address of the first memory cell of the next node). In a *doubly linked list* we not only store a reference to the successor of each element, but

[2]Do not confuse the two structures. Arrays are by definition contiguous memory cells giving us efficiency both in terms of memory usage and speed of access. Linked lists do not have to consist of contiguous cells and for each cell we have to pay not only the cost of storing an item but also the location of the next cell. The disadvantage of arrays is that we cannot be sure of being able to grow them in situ whereas of course we can always grow a list (subject to memory limitations). Confusing the two things is inexcusable.

[3]A sequence can consist of a single element in which case the first and last elements are identical and of course there are no successor or predecessor elements. In some applications it also makes sense to allow the empty sequence in which case it does not of course have a first or last element.

also to its predecessor. Thus each node needs three successive memory cells. Figure 3.1 illustrates how an array, a singly linked list, and a doubly linked list storing the sequence $o1, o2, o3, o4, o5$ may be located in memory.[4] Figure 3.2 gives a more abstract view which is how we usually picture the data.

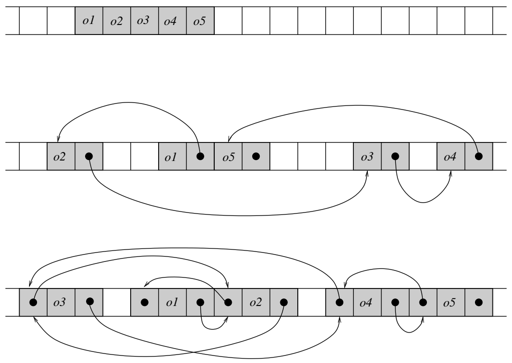

Figure 3.1. An array, a singly linked list, and a doubly linked list storing $o1, o2, o3, o4, o5$ in memory.

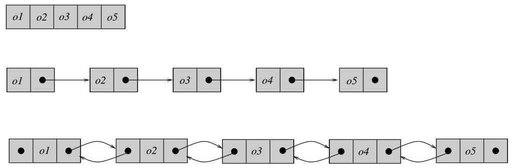

Figure 3.2. An array, a singly linked list, and a doubly linked list storing $o1, o2, o3, o4, o5$.

The advantage of storing a sequence in an array is that elements of the sequence can be accessed quickly in terms of rank. The advantage of linked lists is that they are flexible, of unbounded size (unlike arrays) and easily allow the insertion of new elements.

We will discuss two ADTs for sequences. Both can be realised using linked lists or arrays, but arrays are (maybe) better for the first, and linked lists for the second.

[4]This memory model is simplified, but it illustrates the main points.

Vectors

A *Vector* is an ADT for storing a sequence S of n elements that supports the following methods:

- elemAtRank(r): Return the element of rank r; an error occurs if $r < 0$ or $r > n - 1$.
- replaceAtRank(r, e): Replace the element of rank r with e; an error occurs if $r < 0$ or $r > n - 1$.
- insertAtRank(r, e): Insert a new element e at rank r (this increases the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n$.
- removeAtRank(r): Remove the element of rank r (this reduces the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n - 1$.
- size(): Return n, the number of elements in the sequence.

The most straightforward data structure for realising a vector stores the elements of S in an array A, with the element of rank r being stored at index r (assuming that the first element of an array has index 0). We store the length of the sequence in a variable n, which must always be smaller than or equal to $A.length$. Then the methods elemAtRank, replaceAtRank, and size have trivial algorithms[5] (cf. Algorithms 3.3–3.5) which take $O(1)$ time.

```
Algorithm elemAtRank($r$)
  1. return $A[r]$
```

Algorithm 3.3

```
Algorithm replaceAtRank($r, e$)
  1. $A[r] \leftarrow e$
```

Algorithm 3.4

```
Algorithm size()
  1. return $n$      // $n$ stores the current length of the sequence,
                        which may be different from the length of $A$.
```

Algorithm 3.5

By our general assumption that each line of code only requires a constant number of computation steps, the running time of Algorithms 3.3–3.5 is $\Theta(1)$.

[5]We don't worry about implementation issues such as error handling.

The implementation of insertAtRank and removeAtRank are much less efficient (see Algorithms 3.6 and 3.7). Also, there is a problem with insertAtRank if $n = A.length$ (we will consider this issue properly in § 3.4 on *dynamic arrays*), but for now we assume that the length of the array A is chosen to be large enough to never fill up. In the worst case the loop of insertAtRank is iterated n times and the loop of removeAtRank is iterated $n - 1$ times. Hence $T_{\mathsf{insertAtRank}}(n)$ and $T_{\mathsf{removeAtRank}}(n)$ are both $\Theta(n)$.

Algorithm insertAtRank(r, e)

```
1. for $i \leftarrow n$ downto $r + 1$ do
2.     $A[i] \leftarrow A[i - 1]$
3. $A[r] \leftarrow e$
4. $n \leftarrow n + 1$
```

Algorithm 3.6

Algorithm removeAtRank(r)

```
1. for $i \leftarrow r$ to $n - 2$ do
2.     $A[i] \leftarrow A[i + 1]$
3. $n \leftarrow n - 1$
```

Algorithm 3.7

The *Vector* ADT can also be realised by a data structure based on linked lists. Linked lists do not properly support the access of elements based on their rank. To find the element of rank r, we have to step through the list from the beginning for r steps. This makes all methods required by the *Vector* ADT quite inefficient, with running time $\Theta(n)$.

Lists

Suppose we had a sequence and wanted to remove every element satisfying some condition. This would be possible using the *Vector* ADT, but it would be inconvenient and inefficient (for the standard implementation of *Vector*). However, if we had our sequence stored as a linked list, it would be quite easy: we would just step through the list and remove nodes holding elements with the given condition. Hence we define a new ADT for sequences that abstractly reflects the properties of a linked list—a sequence of nodes that each store an element, have a successor, and (in the case of doubly linked lists) a predecessor. We call this ADT *List*. Our abstraction of a node is a *Position*, which is itself an ADT associated with *List*. The basic methods of the *List* are:

- element(p): Return the element at position p.

- first$()$: Return the position of the first element; an error occurs if the list is empty.
- isEmpty$()$: Return TRUE if the list is empty and FALSE otherwise.
- next(p): Return the position of the element following the one at position p; an error occurs if p is the last position.
- isLast(p): Return TRUE if p is the last position of the list and FALSE otherwise.
- replace(p, e): Replace the element at position p with e.
- insertFirst(e): Insert e as the first element of the list.
- insertAfter(p, e): Insert element e after position p.
- remove(p): Remove the element at position p.

List also has methods last$()$, previous(p), isFirst(p), insertLast(e), and insertBefore(p, e). These methods correspond to first$()$, next(p), isLast(p), insertFirst(e), and insertAfter(p, e) if we reverse the order of the list; their functionality should be obvious.

The natural way of realising the *List* ADT is by a data structure based on a doubly linked list. *Position*s are realised by *nodes* of the list, where each node has fields *previous*, *element*, and *next*. The list itself stores a reference to the first and last node of the list. Algorithms 3.8–3.9 show implementations of insertAfter and remove.

Algorithm insertAfter(p, e)

```
1. create a new node $q$
2. $q.element \leftarrow e$
3. $q.next \leftarrow p.next$
4. $q.previous \leftarrow p$
5. $p.next \leftarrow q$
6. $q.next.previous \leftarrow q$
```

Algorithm 3.8

Algorithm remove(p)

```
1. $p.previous.next \leftarrow p.next$
2. $p.next.previous \leftarrow p.previous$
3. delete $p$        (done automatically in JAVA by garbage collector.)
```

Algorithm 3.9

The asymptotic running time of Algorithms 3.8–3.9 is $\Theta(1)$. It is easy to see that all other methods can also be implemented by algorithms of asymptotic running time $\Theta(1)$ but only because we assume that p *is given as a direct "pointer"* (to the relevant node). An operation such as insertAtRank, or asking to insert into the element's "sorted" position, would take $\Omega(n)$ worst-case running time on a list.

Given the trade-off between using *List* and *Vector* for different operations, it makes sense to consider combining the two ADTs into one ADT *Sequence*, which will support all methods of both *Vector* and *List*. For *Sequence*, both arrays and linked lists are more efficient on some methods than on others. The data structure used in practice would depend on which methods are expected to be used most frequently in the application.

3.4 Dynamic Arrays

There is a real problem with the array-based data structures for sequences that we have not considered so far. What do we do when the array is full? We cannot simply extend it, because the part of the memory following the block where the array sits may be used for other purposes at the moment. So what we have to do is to allocate a sufficiently large block of memory (large enough to hold both the current array and the additional element we want to insert) somewhere else, and then copy the whole array there. This is not efficient, and we clearly want to avoid doing it too often. Therefore, we always choose the length of the array to be a bit larger than the number of elements it currently holds, keeping some extra space for future insertions. In this section, we will see a strategy for doing this surprisingly efficiently.

Concentrating on the essentials, we shall only implement the very basic ADT *VeryBasicSequence*. It stores a sequence of elements and supports the methods elemAtRank(r), replaceAtRank(r, e) of *Vector* and the method addLast(e) of *List*. So it is almost like a queue without the dequeue operation.

Our data structure stores the elements of the sequence in an array A. We store the current size of the sequence in a variable n and let N be the length of A. Thus we must always have $N \geq n$. The *load factor* of our array is defined to be n/N. The load factor is a number between 0 and 1 indicating how much space we are wasting. If it is close to 1, most of the array is filled by elements of the sequence and we are not wasting much space. In our implementation, we will always maintain a load factor of at least $1/2$.

The methods elemAtRank(r) and replaceAtRank(r, e) can be implemented as for *Vector* by algorithms of running time $\Theta(1)$. Consider Algorithm 3.10 for insertions. As long as there is room in the array, insertLast simply inserts the element at the end of the array. If the array is full, a new array of length twice the length of the old array plus the new element is created. Then the old array is copied to the new one and the new element is inserted at the end.

```
Algorithm insertLast($e$)
 1. if $n < N$ then
 2.     $A[n] \leftarrow e$
 3. else                    // $n = N$, i.e., the array is full
 4.     $N \leftarrow 2(N + 1)$
 5.     Create new array $A'$ of length $N$
 6.     for $i = 0$ to $n - 1$ do
 7.         $A'[i] \leftarrow A[i]$
 8.     $A'[n] \leftarrow e$
 9.     $A \leftarrow A'$
10. $n \leftarrow n + 1$
```

Algorithm 3.10

Amortised Analysis

By letting the length of the new array be at most twice the number of elements it currently holds, we guarantee that the load factor is always at least $1/2$. Unfortunately, the worst-case running time of inserting one element in a sequence of size n is $\Theta(n)$, because we need $\Omega(n)$ steps to copy the old array into the new one in lines 6–7. However, we only have to do this copying phase occasionally, and in fact we do it less frequently as the sequence grows larger. This is reflected in the following theorem, which states that if we average the worst-case performance of *a sequence of insertions* (this is called *amortised analysis*), we only need an average of $O(1)$ time for each.

Theorem 3.11. *Inserting m elements into an initially empty VeryBasicSequence using the method* insertLast *(Algorithm 3.10) takes $\Theta(m)$ time.*

If we only knew the worst-case running time of $\Theta(n)$ for a single insertion into a sequence of size n, we might conjecture a worst-case time of $\sum_{n=1}^{m} \Theta(n) = \Theta(m^2)$ for m insertions into an initially empty *VeryBasicSequence*. Our $\Theta(m)$ bound is therefore a big improvement. Analysing the worst-case running-time of a total *sequence* of operations, is called *amortised analysis*.

PROOF Let $I(1), \ldots, I(m)$ denote the m insertions. For most insertions only lines 1,2, and 10 are executed, taking $\Theta(1)$ time. These are called *cheap* insertions. Occasionally on an insertion we have to create a new array and copy the old one into it (lines 4-9). When this happens for an insertion $I(i)$ it requires time $\Theta(i)$, because all of the elements $l(1), \ldots, l(i-1)$ need to be copied into a new array (lines 6-7). These are called *expensive* insertions. Let $I(i_1), \ldots, I(i_\ell)$, where $1 \leq i_1 < i_2 < \ldots < i_\ell \leq m$, be all the expensive insertions.

Then the overall time we need for all our insertions is

$$\sum_{j=1}^{\ell} \Theta(i_j) + \sum_{\substack{1\le i\le m\\ i\ne i_1,\dots,i_\ell}} \Theta(1) \tag{3.1}$$

We now split this into two parts, one for O and one for Ω (recall that $f = \Theta(g)$ iff $f = O(g)$ and $f = \Omega(g)$). First consider O.

$$\sum_{j=1}^{\ell} O(i_j) + \sum_{\substack{1\le i\le m\\ i\ne i_1,\dots,i_\ell}} O(1) \le \sum_{j=1}^{\ell} O(i_j) + \sum_{i=1}^{m} O(1) \le O\Big(\sum_{j=1}^{\ell} i_j\Big) + O(m), \tag{3.2}$$

where at the last stage we have repeatedly applied rule (2) of Theorem 2.3 (lecture notes 2). To give an upper bound on the last term in (3.2), we have to determine the i_j. This is quite easy: We start with $n = N = 0$. Thus the first insertion is expensive, and after it we have $n = 1, N = 2$. Therefore, the second insertion is cheap. The third insertion is expensive again, and after it we have $n = 3, N = 6$. The next expensive insertion is the seventh, after which we have $n = 7, N = 14$. Thus $i_1 = 1, i_2 = 3, i_3 = 7$. The general pattern is

$$i_{j+1} = 2i_j + 1.$$

Now an easy induction shows that $2^{j-1} \le i_j < 2^j$, and this gives us

$$\sum_{j=1}^{\ell} i_j \le \sum_{j=1}^{\ell} 2^j = 2^{\ell+1} - 2 \tag{3.3}$$

(summing the *geometric series*). Since $2^{\ell-1} \le i_\ell \le m$, we have $\ell \le \lg(m) + 1$. Thus

$$2^{\ell+1} - 2 \le 2^{\lg(m)+2} - 2 = 4 \cdot 2^{\lg(m)} - 2 = 4m - 2 = O(m). \tag{3.4}$$

Then by (3.1)–(3.4), the amortised running time of $I(1), \dots, I(m)$ is $O(m)$.

Next we consider Ω. By (3.1) and by properties of Ω , we know

$$\sum_{j=1}^{\ell} \Omega(i_j) + \sum_{\substack{1\le i\le m\\ i\ne i_1,\dots,i_\ell}} \Omega(1) \ge \sum_{\substack{1\le i\le m\\ i\ne i_1,\dots,i_\ell}} \Omega(1). \tag{3.5}$$

Recall that we have shown that $i_1 = 1$ and for $j \ge 1$, $i_{j+1} = 2i_j + 1$. We claim that for any $k \in \mathbb{N}$, with $k \ge 4$, the set $\{1, 2, 3, \dots, k\}$ contains at most $k/2$ occurrences of i_j indices (This can be proven by induction, left as an exercise). Hence there at least $m/2$ elements of the set $\{1 \le i \le m \mid i \ne i_1, \dots, i_\ell\}$. Hence

$$\sum_{\substack{1\le i\le m\\ i\ne i_1,\dots,i_\ell}} \Omega(1) \ge (m/2)\Omega(1) = \Omega(m), \tag{3.6}$$

whenever $m \ge 4$. Combining (3.5) and (3.6), the running time for $I(1), \dots, I(m)$ is $\Omega(m)$. Together with our $O(m)$ result, this implies the theorem. □

Note that although the theorem makes a statement about the "average time" needed by an insertion, it is *not* a statement about average running time. The reason is that the statement of the theorem is completely independent of the input, i.e., the elements we insert. In this sense, it is a statement about the worst-case running time, whereas average running time makes statements about "average", or random, inputs. Since an analysis such as the one used for the theorem occurs quite frequently in the theory of algorithms, there is a name for it: *amortised analysis*. A way of rephrasing the theorem is saying that the *amortised (worst case) running time* of the method insertLast is $\Theta(1)$.

Finally, suppose that we want to add a method removeLast() for removing the last element of our ADT *VeryBasicSequence*. We can use a similar trick for implementing this—we only create a new array of, say, size $3/4$ of the current one, if the load factor falls below $1/2$. With this strategy, we always have a load factor of at least $1/2$, and it can be proved that the amortised running time of both insertLast and removeLast is $\Theta(1)$.

The JAVA Collections Framework contains the ArrayList implementation of List (in the class java.util.ArrayList), as well as other list implementations. ArrayList, though not implemented as a dynamic array, provides most features of a dynamic array.

3.5 Further reading

If you have [GT]: The chapters "Stacks, Queues and Recursion" and "Vectors, Lists and Sequences".

Exercises

1. Implement the method removeLast() for removing the last element of a dynamic array as described in the last paragraph of Section 3.4.

2. Prove by induction that in the proof of Theorem 3.11, the following claim (towards the end of the proof) is true:
   For any $k \in \mathbb{N}$, $k \ge 4$, the set $\{1, 2, 3, \dots, k\}$ contains at most $k/2$ occurrences of i_j indices.

3. What is the amortised running time of a sequence $P = p_1p_2 \dots p_n$ of operations if the running time of p_i is $\Theta(i)$ if i is a multiple of 3 and $\Theta(1)$ otherwise?

   What if the running time of p_i is $\Theta(i)$ if i is a square and $\Theta(1)$ otherwise?

   Hint: For the second question, use the fact that $\sum_{i=1}^{m} i^2 = \Theta(m^3)$.

4. Implement in JAVA a `Stack` class based on dynamic arrays.

Hash Tables

In this lecture we introduce the *Dictionary* ADT and give a simple data structure for it that is efficient in practice.

4.1 Dictionaries

A *Dictionary* stores key–element pairs, which are called *items*. In this and future *Dictionary*–related notes, n will denote the number of items in a dictionary. *Dictionary* allows key duplicates—several items may have the same key. A dictionary provides three basic methods:

- findElement(k): If the dictionary contains an item with key k, then return the element of such an item, otherwise return the special element NO_SUCH_KEY.
- insertItem(k, e): Insert an item with key k and element e.
- removeItem(k): If the dictionary contains an item with key k, then delete such an item and return its element, otherwise return the special element NO_SUCH_KEY.

Of course we must ensure that NO_SUCH_KEY cannot itself be an element that can be associated with any key. An alternative approach is to turn the two relevant methods into boolean functions so that success or otherwise is indicated by the value returned; the element, if any, could be returned via a parameter.

A key point to observe is that *Dictionary* allows the insertion of more than one element with the same key. In this scenario, removeItem(k) removes just one item with key k and not all of them (when there is more than one such item).

An obvious data structure for *Dictionary* can be based on the linked list of Lecture Note 3. If we implement such a *list dictionary* by inserting at the end of the list (not keeping the items in key-sorted order), the method insertItem(k, e) has running time $\Theta(1)$ and the methods findElement(k) and removeItem(k) have running time $\Theta(n)$. In this note and future notes (Lecture Notes 5–7), we will see data structures which achieve better worst-case running times for the *Dictionary* methods. In this note we present Hash Tables.

4.2 Direct Addressing and Bucket Arrays

Suppose we are in the special case where the keys of the items to be inserted into the *Dictionary* are integers in the range $0, \dots, N-1$, for some $N \in \mathbb{N}$ with $N > 0$, and suppose also that no two elements have the same key (this is a restricted case of *Dictionary*). In this case we can implement a *Dictionary* by setting up an array B of length N of the appropriate type to hold elements. The method insertItem(k, e) simply sets $B[k] \leftarrow e$. The method findElement(k) returns $B[k]$, and the method removeItem(k) returns $B[k]$ and resets $B[k]$ to $null$. The running time

of these methods is $\Theta(1)$. This data structure implementing the *Dictionary* ADT is sometimes called a *direct address table*.

The problem with direct addressing is that we have to keep an array of size N in memory, even if n, the number of items stored in the dictionary, is much smaller than N. If this is the case, we waste a great deal of memory space. Direct addressing is only acceptable if the number of items we expect to store in the dictionary is somewhat close to N. If we are in this fortunate situation, however, this simple data structure is the most efficient implementation of the *Dictionary* ADT.

Another problem with the simple direct addressing implementation is that it assumes no two items have the same key. However we can remove this restriction by changing the structure of our array of length N. We say that when we try to insert an item into a cell of the direct address table which already has an item stored, a *collision* occurs. We can handle collisions by changing the model so that we work on an array B of *Lists* of elements, and store all elements with key k in the list $B[k]$. In this context, the lists $B[i]$, for $0 \leq i \leq N-1$, are often called *buckets*, and the array B is called the *bucket array*. To insert, find, and remove elements, we use the methods insertFirst(), first(), and remove(p) (with $p = 1$) of *List*. As we saw in Lecture Note 3, the methods insertFirst(), first() and remove(p) all have a running-time of $\Theta(1)$.

Bucket arrays are very efficient if N is not much larger than the number of different keys appearing in the dictionary (for example, if we have at least cN elements in the dictionary, for $c > 0$ some constant which is not too small). If this is the case, not much space is wasted. Note that because findElement(k) and removeItem(k) are defined in terms of a key k, and we only require one element with that key to be found/removed, the first element of the list for that key may *always* be taken. Therefore in this simple case, we need not even concern ourselves with the size of the list (or bucket) for each k—the running-time is $\Theta(1)$ for all methods, regardless of how the elements are distributed.

4.3 Hash Tables

Usually, we are not in a situation where keys are small natural numbers. Hash tables extend the idea of bucket arrays to arbitrary keys. Again, we set up a bucket array B of length N, for a suitable integer N. The keys are *mapped* to natural numbers in the range $0, \dots, N-1$ using a *hash function* h that maps each key k to a number $h(k) \in \{0, \dots, N-1\}$. When an item with key k arrives to be inserted, it is inserted into the bucket $B[h(k)]$.

Suppose for a moment that we have chosen a hash function h. It is certainly possible that two elements with two different keys k_1 and k_2 end up in the same bucket $B[h(k_1)]$ if $h(k_1) = h(k_2)$. Thus we cannot just store the elements in the buckets, but must store the full items, including their keys. In effect, we let each bucket be a small dictionary. By choosing a good hash function (and by making reasonable assumptions about the distribution on the key k), we *hope* to avoid too many keys having the same hash value. This is important in this new scenario of hash values, even though it did not matter when we defined $h(k) = k$ above. The reason that it becomes important *now* is because when we implement findElement(k) or removeItem(k), we actually need to search the bucket at $h(k)$ rather than just taking any element of that bucket (not all elements in the bucket have key k). If we have a good hash function (and if our input keys are distributed well) the buckets will usually be small, and we can use simple list dictionaries for the buckets. The technique of using lists to store the elements with hash value h is sometimes called *chaining* in the literature.

A *hash table* realising the *Dictionary* ADT consists of a hash function h and an array B of length N of list dictionaries. Algorithms 4.1–4.3 implement the methods of *Dictionary*.

```
Algorithm findElement($k$)
  1. Compute $h(k)$
  2. return $B[h(k)]$.findElement($k$)
```

Algorithm 4.1

```
Algorithm InsertItem($k, e$)
  1. Compute $h(k)$
  2. $B[h(k)]$.insertItem($k, e$)
```

Algorithm 4.2

```
Algorithm removeItem($k$)
  1. Compute $h(k)$
  2. return $B[h(k)]$.removeItem($k$)
```

Algorithm 4.3

The running time of the methods obviously depends on the time needed to compute $h(k)$ and on the running time of the corresponding methods of the list dictionaries. Let us assume that computing $h(k)$ requires at most T_h computation steps. (Of course T_h may depend on the key, so it would be more precise to write $T_h(n_{\text{key}})$, where n_{key} denotes the size of the key. On the other hand, keys often have constant size.)

Recall that the method insertItem of list dictionary has running time $\Theta(1)$ and the methods findElement and removeItem have a worst-case running time of $\Theta(m)$, when m is the number of items in the dictionary. Then the worst-case running time of InsertItem of the hash table dictionary is $T_h + \Theta(1)$. The worst-case running time of both findElement and removeItem is $T_h + \Theta(m)$, where m denotes the size of the bucket $B[h(k)]$ associated with the key k. Since it could happen that all items end up in the same bucket, in the worst case we have

$m = n$. Thus in the worst case, a hash table dictionary is no more efficient then a list dictionary. However, by choosing an appropriate hash function h, *and* by assuming that the input keys are *uniformly distributed*, we can avoid ending up in the worst case in all practically relevant situations. As a matter of fact, we can usually *expect* the size m of all buckets to be $O(1)$, giving us an *expected* running time of $T_h + \Theta(1)$ for all methods.

Remark 4.4. Before discussing the choice of good hash functions, note two facts relating to the basic implementation of hash tables above. First note that instead of using list dictionaries as buckets, we could use any other implementation of *Dictionary*. List dictionaries are the most efficient for small dictionaries, however, and we expect our buckets to be small. Second, note that there are versions of hash tables that do not use external buckets, but store all items directly in the array. While this is a bit more space efficient, it requires sophisticated schemes for handling collisions (usually referred to as *open addressing schemes*). Details can be found in most algorithms textbooks (e.g., Chapter 11 of [CLRS], Section 2.5 of [GT], Chapter 14.3 of Sedgewick).

Another reference is the classic three volume series *The Art of Computer Programming* by D.E. Knuth (Addison Wesley) which contains extensive discussions of many of the algorithms and data structures we will see in this thread. They are seriously challenging books. However, if you really want to cover a topic in great detail, this is often the place to look. Section 6.4 of the third volume (entitled *Sorting and Searching*) is on hashing.

4.4 Hash Functions

We want to find a hash function h that maps keys to natural numbers in a fixed range $0, \ldots, N-1$ in such a way that

(H1) h evenly distributes the keys over the whole range. That is, the total number of keys mapped to the number j is roughly the same for all $j \in 0, \ldots, N$.

(H2) h is easy to compute.

A common strategy is to first map arbitrary objects to integers by a function called the *hash code*, and then map arbitrary integers to integers in the range $0, \ldots, N-1$ by a function called the *compression map*:

$$\begin{tikzcd}
\boxed{\text{Arbitrary Objects}} \arrow[r, "\text{hash code}"] & \boxed{\text{Integers}} \arrow[r, "\text{compression}", "\text{map}"'] & \boxed{\{0,\ldots,N-1\}}
\end{tikzcd}$$

Hash Codes

Since every object is represented as a sequence of bits in memory anyway, in principle there is no problem: we can just consider the sequence of bits representing a key as an integer and let this integer be the hash code of the key.

In practice, though, there might be a problem, because integers on a real computer cannot be arbitrarily large. Suppose we are working with a machine where integers can be at most 32 bits long. The easiest way to assign hash codes in this situation is to only take the leftmost (or rightmost) 32 bits of the bit representation of the key to be the hash code. In general, this may be a dangerous strategy, because all keys whose bit representation starts with the same 32 bits will get the same hash code and will end up in the same bucket. For example, if keys are strings, then all strings starting with the same few letters will get the same hash code. Since the distribution of strings arising in practical applications is hardly ever random, the resulting hash map will not usually satisfy condition (H1) in practical applications.

Summation hash code

If we want to take all bits of the representation of the key into account, we could proceed as follows: instead of considering the bit representation of the key as one integer, we consider it as a sequence of integers, $a_0, \ldots, a_{\ell-1}$. One way of mapping this sequence to a single integer is to sum them all up. Thus if we have a key k whose bit representation corresponds to integers $a_0, \ldots, a_{\ell-1}$, we could let the hash code of k be

$$a_0 + a_1 + \cdots + a_{\ell-1}.$$

We simply disregard overflows in the summation, that is, we actually let the hash code be $\sum_{i=0}^{\ell-1} a_i \bmod 2^{32}$.

While this summation hash code is a quite reasonable choice, there are still some problems with condition (H1). For example, we can recognise some "patterns" in the way sequences get mapped to buckets—keys corresponding to sequences (a_0, a_1, a_2), (a_1, a_2, a_0), and (a_1, a_0, a_2) will get the same hash code, which may be problematic in some applications. More generally any permutation of a sequence is given the same code.

Polynomial hash code

For the *polynomial hash code*, we choose a fixed integer x and let the hash code of a key k corresponding to the sequence $a_0, \ldots, a_{\ell-1}$ be

$$a_0 + a_1x + a_2x^2 + \cdots a_{\ell-1}x^{\ell-1}$$

(again disregarding overflows). This polynomial hash code can be especially well-suited to keys of type `String`. Suppose our `String` $s = s_0 \ldots s_{\ell-1}$ is a sequence of 7-bit characters (so each $s_i \in \{0, 1 \ldots, 127\}$), and suppose we want to map to a hash value in the set $\{0, 1, \ldots, N-1\}$. Then we can perform the mapping by choosing an appropriate integer x and evaluating $(\sum_{i=0}^{\ell-1} s_i x^i) \bmod N$. *It is very important, in choosing the value of x, to make sure that x and N are coprime* (i.e., they do not share common factors, in other words the only natural number that divides both of them is 1). If x and N are not coprime, then small strings will not get well distributed among the buckets of the Hash Table. As an example, Java's `HashMap` uses a polynomial hash function with $x = 31$, which is prime, to hash keys of type `String` (for `HashMap`, usually $N = 2^k$ for some integer k). The observations here are based on practical experience rather than any theoretical analysis.

Polynomial hash codes seem to satisfy (H1) (at least on an heuristic level, we have given no proof), it it not clear that they satisfy (H2). However, polynomials can be evaluated quite efficiently, as the following remarks show.

Aside: Evaluating Polynomials

Suppose we are given integers $a_0, \ldots, a_{\ell-1}, x$ and want to compute

$$a_0 + a_1x + a_2x^2 + \cdots a_{\ell-1}x^{\ell-1}. \tag{4.1}$$

A naïve way of doing this requires $\Theta(\ell^2)$ arithmetic operations (additions and multiplications). However, there is a clever way of rewriting the polynomial in such a way that it can be evaluated with only $\Theta(\ell)$ arithmetic operations. *Horner's rule* says that the polynomial (4.1) is equal to

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{\ell-2} + x \cdot a_{\ell-1}) \cdots)).$$

This expression clearly can be evaluated with only $\Theta(\ell)$ arithmetic operations.

It is interesting to note that Horner's rule has been proved to be optimal. Any algorithm that can evaluate an arbitrary polynomial of degree l for arbitrary x must use at least as many additions/subtractions and at least as many multiplications as Horner's rule (we allow divisions but they do not help). This is a rare example where we know the exact *complexity* of a non-trivial computational problem.

Compression Maps

Many Hash Functions only consist of a "hash code", and do not involve a compression phase. However, depending on the domain, it may be helpful to first perform a hash to a large key space, and then "compress" these large integers to integers in the range $0, \ldots, N-1$. Like the hash code, the compression map should satisfy conditions (H1) and (H2). It turns out that compression functions of the following form do this fairly well: an integer k is mapped to

$$|ak + b| \bmod N,$$

where a, b are just randomly chosen integers. We choose a, b at running time whenever we create a new hash table. This method works particularly well if a and N are coprime.

If a and N are not coprime then the map is definitely bad. Let's assume that a, b and k are non-negative to simplify the discussion. If we are to have a good hash map then, at the very least, for each location r of the array there should be a key that hashes into it, i.e., the equation

$$ak + b \equiv r \bmod N$$

should have a solution (for an integer k, if k is negative we can get a positive value by adding enough multiples of N to it). Note that we may assume $0 \le b \le N-1$ because we take the remainder after division by N so any quotient would be lost anyway. Hence 0 hashes to b since $a \cdot 0 + b = b$ and this stays as b after taking the remainder when divided by N. The preceding equation is the same as requiring that the equation

$$ak + b = qN + r$$

has a solution for integers q, k. Rearranging we have

$$ak - qN = r - b.$$

So if $d = \gcd(a, b)$ divides both a and N then it divides $r - b$, i.e., $r - b = sd$ for some integer s. Hence the only possible values of r are $b + sd$ for integers s. We know that 0 hashes to b but then the next location to which any key might hash is $b + d$. So if $d > 1$ then nothing hashes to $b + 1, b + 2, \ldots, b + d - 1$. In general, our argument shows that the only locations to which anything can hash are all those of the form $b + sd$ where $0 \le b + sd \le N - 1$ (remember that s can be negative as well as positive or 0). In fact we can prove (using a simple fact deduced from the Euclidean algorithm for finding greatest common divisors[1]) that all such locations do have a key hashing to them. So, unless $\gcd(a, N) = 1$ the compression map is definitely bad. If $\gcd(a, N) = 1$ then the map still might be bad because the keys do not spread out evenly but this is a property of the distribution of keys which varies with the application domain.

4.5 Load Factors and Re-hashing

The choice of a good size for N obviously depends on the number n of items we intend to store in our hash table. We call the fraction n/N the *load factor* of the hash table. We do not want the load factor to be too high, because that would lead to many collisions. We don't want it to be too low, because that would be a waste of memory space. A common load factor in practice (e.g., with the JVC `HashMap`) is $3/4$.

If we know n, we can choose N to be a prime number of size roughly $(4/3)n$. As N is prime it is automatically coprime to all $a < 2N$ provided $N \neq 2$, which is the case in any sensible application of hashing. If we do not know n in advance, or if n changes over time because items are inserted and removed, we must occasionally choose a new N, create a new hash table, and move all items from the old table to the new one. This is called *re-hashing*. We adopt a similar strategy as the one we used for dynamic arrays in Lecture Note 3. Whenever the load factor gets too far above $3/4$ we choose a new prime number N close to $(4/3)n$ and create a new bucket array of size N. We choose a new compression function and thus obtain a new hash function that maps our keys to the new range $0, \ldots, N - 1$. Then we move all items stored in the old table to the new table. We proceed similarly if the load factor falls too far below $3/4$. To do this we need to be able to find prime numbers of a given size. There are efficient ways of doing this, but beyond the scope of this course. A common trick in practice is to store a table containing for $7 \le k \le 31$, the closest prime number to 2^k, which would easily cover all practical applications (though with 2 rather than $4/3$).

[1]This states that the equation $au + bv = c$ where a, b, c are integers has a solution for integers u, v if and only if $\gcd(a, b)$ divides c.

4.6 Hashing in Java

The *Dictionary* ADT is similar (though not the same) to the `Map` ADT of the Java Collections framework. The difference is that `Map` of JVC requires that the keys should all be distinct (does not allow different elements with the same key). One of the implementations of *Map* that is provided in the JVC is the `HashMap` implementation, which is a Hash Table where the number of buckets N may vary (but the default is 16). `HashMap` has parameters which allow the user to specify the (initial) table size N, and the desired "load factor" (the average number of items per bucket). This Java implementation "re-sizes" the table as the number of items n increases (this aspect of the implementation is similar to that of the *dynamic array* that we met in Lecture Note 3). Different hash functions are used, depending on the domains of the `key` which is being hashed. Java also offers a `Hashtable` implementation which is almost identical to `Hashmap`.

4.7 Further Reading

In [GT], the "Maps and Dictionaries" chapter (chapter 8 in edition 3, Chapter 9 in edition 4) has a lot of material on this topic. Sections 8.1, 8.2 and 8.3 are directly relevant to this lecture.

In [CLRS], there is an entire chapter on "Hash Tables".

Chapter 14 of "Algorithms in Java" (3rd Ed), by Robert Sedgewick, has a *very* nice presentation of Hashing.

For information on the *HashMap* interface of JVC:
`http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html`

Exercises

1. For an ASCII character c, let $A(c)$ denote the ASCII code of c (an integer between 0 and 127). Let f, g be hash codes for ASCII strings defined by

$$\begin{aligned} f(c_0 \ldots, c_{n-1}) &= A(c_0) + \ldots + A(c_{n-1}), \\ g(c_0 \ldots, c_{n-1}) &= A(c_0) + A(c_1) \cdot 3 + \ldots + A(c_{n-1}) \cdot 3^{n-1}. \end{aligned}$$

   Compute f and g for a few example strings and write JAVA methods computing $f(s)$ and $g(s)$, respectively, for a given string.

2. Draw the 11–item hash table resulting from hashing keys with hash codes 12, 44, 12, 88, 23, 94, 11, 39, 20, 16, and 5 using the compression map $g(i) = (2i + 5) \bmod 11$.

AVL Trees

We have already seen one implementation of the *Dictionary* ADT that works well in practice—hash tables. You have very probably already met binary search trees (they are discussed below). Both of these data structures have a worst case running time of $\Theta(n)$ for findElement, insertItem and removeItem. In this lecture note, we study a *balanced* binary search tree that implements the methods findElement, insertItem and removeItem of *Dictionary* with a worst-case running time of $\Theta(\lg(n))$ for the methods. The balanced binary trees studied here were invented in 1962 by G. Adelson-Velskii and E.M. Landis and are called the AVL Trees.

5.1 Binary Search Trees

A tree is *binary* if each of its vertices either has two children, in which case it is called an *internal vertex*, or no children at all, in which case it is called a *leaf* or an *external vertex*. We refer to the two children of an internal vertex v as the *left* child and *right* child of v. The vertices of a search tree store *items* consisting of *keys* and *elements*. It is convenient to set up the tree in such a way that only the internal vertices store items[1]. The keys are taken from an ordered set (in Java, keys must be of some `Comparable` type). A binary tree storing items in its internal vertices is a *binary search tree* if for every vertex v, all keys stored in the left subtree of v (rooted at the left child of v) are less than or equal to the key of v and all keys stored in the right subtree are greater than or equal to the key of v. (If we do not allow duplicate keys then the inequalities are strict.)

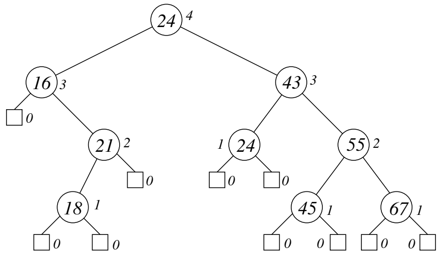

Figure 5.1. A binary search tree. The keys are shown inside the vertices, and the heights are shown next to the vertices.

Figure 5.1 shows an example of a binary search tree. In all our examples, the keys are natural numbers, and we show only the keys of the vertices.

[1]This may seem like a waste of space, but when implementing a search tree we can just represent the leaves as null references.

We now present the implementation of findElement() on any binary search tree.

```
Algorithm findElement($k$)
1. if isEmpty($T$) then return NO_SUCH_KEY
2. else
3.     $u \leftarrow root$
4.     while (($u$ is not null) and $u.key \neq k$) do
5.         if ($k < u.key$) then $u \leftarrow u.left$
6.         else $u \leftarrow u.right$
7.     od
8.     if ($u$ is not null) and $u.key = k$ then return $u.elt$
9.     else return NO_SUCH_KEY
```

Algorithm 5.2

Recall that the *height* of a vertex v of a tree is the maximum length over all paths from v to a leaf of the tree, obtained by following left or right child vertices (we count the number of vertices on a path but not the leaf). The *height* $h = h(T)$ of a tree T is the height of the root. Figure 5.1 shows the height of each vertex.

Examining Algorithm 5.1, it should be clear that, since we walk one step further away from the root with every iteration of the loop at lines 4–7, the worst-case running time of findElement is $O(h)$, where h is the height of the tree. Hence we have:

Theorem 5.3. *For any binary search tree implementation of the Dictionary ADT, findElement has asymptotic worst-case running time $O(h)$, where h is the height of the tree.*

In fact the basic insertItem and removeItem implementations on a (not necessarily balanced) binary search tree also achieve worst-case running time $O(h)$. The problem is that binary search trees can be completely *unbalanced*. In the degenerate case, a tree storing n items can have height n, in which case it essentially looks like a linked list.

In §§5.3 and 5.4 we will describe special implementations of insertItem and removeItem on an AVL tree, which have the affect of keeping the height of the AVL tree bounded by $O(\lg(n))$, and hence guarantee that findElement, insertItem and removeItem all run in $O(\lg(n))$ time.

5.2 AVL trees

We now define the special balance property we maintain for an AVL tree.

Definition 5.4.

(1) A vertex of a tree is *balanced* if the heights of its children differ by one at most.

(2) An *AVL tree* is a binary search tree in which *all* vertices are balanced.

The binary search tree in Figure 5.1 is not an AVL tree, because the vertex with key 16 is not balanced. The tree in Figure 5.5 is an AVL tree.

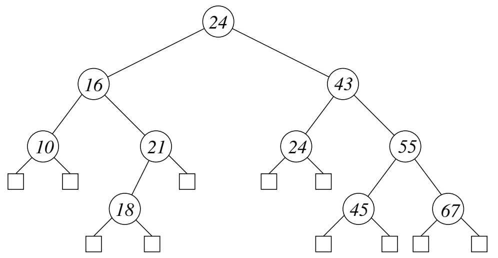

Figure 5.5. An AVL tree.

We now come to our key result.
Theorem 5.6. *The height of an AVL tree storing n items is $O(\lg(n))$.*

PROOF For $h \in \mathbb{N}$, let $n(h)$ be the minimum number of items stored in an AVL-tree of height h. Observe that $n(1) = 1$, $n(2) = 2$, and $n(3) = 4$.

Our first step is to prove, by induction on h, that for all $h \geq 1$ we have

$$n(h) > 2^{h/2} - 1. \tag{5.1}$$

As the induction bases, we note that (5.1) holds for $h = 1$ and $h = 2$ because $n(1) = 1 > \sqrt{2} - 1$ and $n(2) = 2 > 2^1 - 1$.

For the induction step, we suppose that $h \geq 3$ and that (5.1) holds for $h - 1$ and $h - 2$. We observe that by Defn 5.4, we have

$$n(h) \geq 1 + n(h-1) + n(h-2),$$

since one of the two subtrees must have height at least $h-1$ and the other height at least $h - 2$. (We are also using the fact that if $h_1 \geq h_2$ then $n(h_1) \geq n(h_2)$, this is very easy to prove and you should do this.) By the inductive hypothesis, this yields

$$\begin{aligned} n(h) &> 1 + 2^{\frac{h-1}{2}} - 1 + 2^{\frac{h-2}{2}} - 1 \\ &= (2^{-\frac{1}{2}} + 2^{-1})\, 2^{\frac{h}{2}} - 1 \\ &> 2^{\frac{h}{2}} - 1. \end{aligned}$$

The last line follows since $2^{-\frac{1}{2}} + 2^{-1} > 1$. This can be seen without explicit calculations as it is equivalent to $2^{-\frac{1}{2}} > 1/2$ which is equivalent to $2^{\frac{1}{2}} < 2$ and this now follows by squaring. This completes the proof of (5.1).

Therefore for every tree of height h storing n items we have $n \geq n(h) > 2^{h/2} - 1$. Thus $h/2 < \lg(n+1)$ and so $h < 2\lg(n+1) = O(\lg(n))$. This completes the proof of the Theorem. □

It follows from our discussion of findElement that we can find a key in an AVL tree storing n items in time $O(\lg(n))$ in the worst case. However, we cannot just insert items into (or remove items from) an AVL tree in a naïve fashion, as the resulting tree may not be an AVL tree. So we need to devise appropriate insertion and removal methods that will maintain the balance properties set out in Definition 5.4.

5.3 Insertions

Suppose we want to insert an item (k, e) (a key-element pair) into an AVL tree. We start with the usual insertion method for binary search trees. This method is to search for the key k in the existing tree using the procedure in lines 1–7 of findElement(k), and use the result of this search to find the "right" leaf location for an item with key k. If we find k in an internal vertex, we must walk down to the largest near-leaf in key value which is no greater than k, and then use the appropriate neighbour leaf. In effect we ignore the fact that an occurrence of k has been found and carry on with the search till we get to a vertex v after which the search takes us to a leaf l (see Algorithm 5.1). The vertex v is the "largest near-leaf." We make a new internal vertex u to replace the leaf l and store the item there (setting $u.key = k$, and $u.elt = e$).

The updating of u from a leaf vertex to an internal vertex (which will have two empty child vertices, as usual) will sometimes cause the tree to become unbalanced (no longer satisfying Definition 5.4), and in this case we will have to repair it.

Clearly, any *newly unbalanced* vertex that has arisen as a result of inserting into u must be on the path from the new vertex to the root of the tree. Let z be the unbalanced vertex of minimum height. Then the heights of the two children of z must differ by 2. Let y be the child of z of greater height and let x be the child of y of greater height. If both children of y had the same height, then z would already have been unbalanced before the insertion, which is impossible, because before the insertion the tree was an AVL tree. Note that the newly added vertex might be x itself, or it might be located in the subtree rooted at x. Let V and W be the two subtrees rooted at the children of x. Let X be the subtree rooted at the sibling of x and let Y be the subtree rooted at the sibling of y. Thus we are in one of the situations displayed in Figures 5.7–5.10(a). Now we apply the operation that leads to part (b) of the respective figure. These operations are called *rotations*; consideration of the figures shows why.

By applying the rotation we balance vertex z. Its descendants (i.e., the vertices below z in the tree before rotation) remain balanced. To see this, we make the following observations about the heights of the subtrees V, W, X, Y:

- height$(V) - 1 \leq$ height$(W) \leq$ height$(V) + 1$ (because x is balanced). In the Figures, we have always assumed that W is the higher tree, but it does not make a difference.
- max{height(V), height(W)} = height(X) (as y is balanced and height$(x) >$ height(X)).

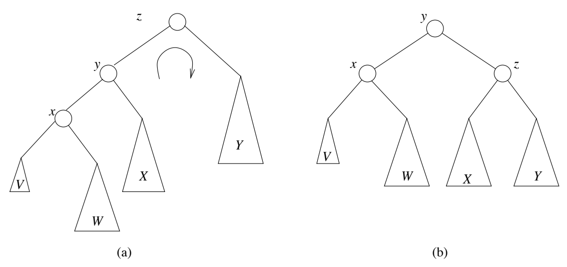

Figure 5.7. A clockwise single rotation

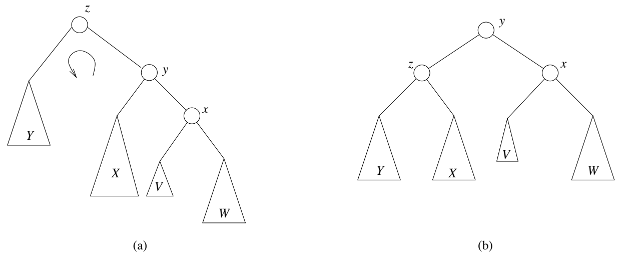

Figure 5.8. An anti-clockwise single rotation

- $\max\{\text{height}(V), \text{height}(W)\} = \text{height}(Y)$ (because $\text{height}(Y) = \text{height}(y) - 2$).

The rotation reduces the height of z by 1, which means that it is the same as it was before the insertion. Thus all other vertices of the tree are also balanced.

Algorithm 5.11 overleaf summarises insertItem(k, e) for an AVL-tree. In order to implement this algorithm efficiently, each vertex must not only store references to its children, but also to its parent. In addition, each vertex stores the height of its left subtree minus the height of its right subtree (this may be -1, 0, or 1).

We now discuss $T_{\text{insertItem}}(n)$, the worst-case running time of insertItem on a AVL tree of size n. Let h denote the height of the tree. Line 1 of Algorithm 5.11 requires time $O(h)$, and line 2 just $O(1)$ time. Line 3 also requires time $O(h)$, because in the worst case one has to traverse a path from a leaf to the root. Lines 4.-6. only require constant time, because all that needs to be done is redirect a few references to subtrees. By Theorem 5.3, we have $h \in O(\lg(n))$. Thus the overall asymptotic running time of Algorithm 5.11 is $O(\lg(n))$.

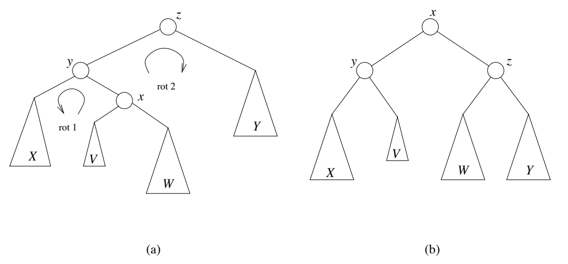

Figure 5.9. An anti-clockwise clockwise double rotation

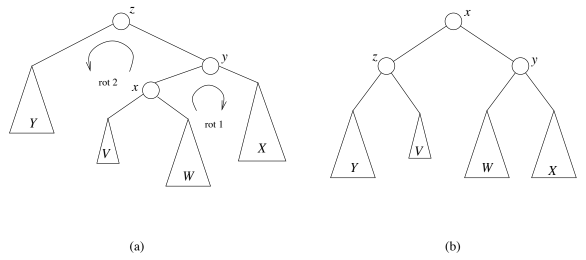

Figure 5.10. A clockwise anti-clockwise double rotation

5.4 Removals

Removals are handled similarly to insertions. Suppose we want to remove an item with key k from an AVL-tree. We start by using the basic removeItem method for binary search trees. This means that we start by performing steps 1–7 of findElement(k) hoping to arrive at some vertex t such that $t.key = k$. If we achieve this, then we will delete t (and **return** $t.elt$), and replace t with the closest (in key-value) vertex u (u is a "near-leaf" vertex as before—note that this does not imply *both* u's children are leaves, but one will be). We can find u by "walking down" from t to the near-leaf with largest key value no greater than k. Then t gets u's item and u's height drops by 1.

After this deletion and re-arrangement, any *newly unbalanced* vertex must be on the path from u to the root of the tree. In fact, there can be at most one unbalanced vertex (why?). Let z be this unbalanced vertex. Then the heights of the two children of z must differ by 2. Let y be the child of z of greater height and x the child of y of greater height. If both children of y have the same height, let x be the left child of y if y is the left child of z, and let x be the right child of y if y is the right child of z. Now we apply the appropriate rotation and obtain a tree where z is balanced and where x, y remain balanced. However, we may have

```
Algorithm insertItem($k, e$)
 1. Perform lines 1.-7. of findElement()($k, e$) on the tree to find the "right"
    place for an item with key $k$ (if it finds $k$ high in the tree, walk down to
    the "near-leaf" with largest key no greater than $k$).
 2. Neighbouring leaf vertex $u$ becomes internal vertex, $u.key \leftarrow k$, $u.elt \leftarrow e$.
 3. Find the lowest unbalanced vertex $z$ on the path from $u$ to the root.
 4.     if no such vertex exists, return (tree is still balanced).
 5.     else
 6.         Let $y$ and $x$ be child and grandchild of $z$ on $z \to u$ path.
 7.         Apply the appropriate rotation to $x, y, z$.
```

Algorithm 5.11

reduced the height of the subtree originally located at z by one, and this may cause the parent of z to become unbalanced. If this happens, we have to apply a rotation to the parent and, if necessary, rotate our way up to the root. Algorithm 5.12 on the following page gives pseudocode for removeItem(k) operating on an AVL-tree.

```
Algorithm removeItem($k$)
 1. Perform lines 1–7 of findElement($k$) on the tree to get to vertex $t$.
 2. if we find $t$ with $t.key = k$,
 3.     then remove the item at $t$, set $e = t.elt$.
 4.     Let $u$ be "near-leaf" closest to $k$. Move $u$'s item to $t$.
 5.     while $u$ is not the root do
 6.         let $z$ be the parent of $u$
 7.         if $z$ is unbalanced then
 8.             do the appropriate rotation at $z$
 9.         Reset $u$ to be the (possibly new) parent of $u$
10.     return $e$
11. else return NO_SUCH_KEY
```

Algorithm 5.12

We now discuss the worst-case running-time $T_{\text{removeItem}}(n)$ of our AVL implementation of removeItem. Again letting h denote the height of the tree, we recall that line 1 requires time $O(h) = O(\lg(n))$. Lines 2. and 3. will take $O(1)$ time, while line 4. will take $O(h)$ time again. The loop in lines 5–9 is iterated at most h times. Each iteration requires time $O(1)$. Thus the execution of the whole loop requires

time $O(h)$. Altogether, the asymptotic worst-case running time of removeItem is $O(h) = O(\lg(n))$.

5.5 Ordered Dictionaries

AVL trees are the first data structures for *Dictionaries* we have seen that implement all methods with a worst case running time of $O(\lg(n))$. There are a few similar tree-based dictionary implementations with the same running time. Still, hash tables are more efficient in practice. The main advantage of AVL trees and similar tree-based *Dictionary* implementations is that they support the order of the keys quite efficiently. The *OrderedDictionary* ADT is an extension of the *Dictionary* ADT by methods such as closestKeyBefore(k), which finds the closest key less than or equal to k that is stored in the dictionary. Such methods can easily be implemented on AVL trees with a running time of $O(\lg(n))$, but there is no efficient way of supporting them on hash tables. The same holds for so-called *range queries* such as findAllElementsBetween(k_1, k_2), i.e., find all elements whose key is between k_1 and k_2. Range queries are very important in database applications.

5.6 Reading

[GT] covers AVL trees, but [CLRS] only has details of a different balanced tree called the *Red-Black Tree*.

Note there are detailed worked examples for insertItem and removeItem in the slides to accompany this lecture note.

Exercises

1. Perform the following sequence of operations on an initially empty AVL tree: insertItem(44), insertItem(17), insertItem(32), insertItem(78), insertItem(50), insertItem(88), insertItem(48), insertItem(62), removeItem(32), removeItem(88).

2. Describe a family of AVL-trees in which a single removeItem operation may require $\Theta(\lg(n))$ rotations.

3. Suppose we wanted to implement an extra method findByRank(r) (as described in LN 3 for the *Vector* class) on the AVL tree data structure, and we want to get $O(\lg(n))$ time for this method also. Sketch a plan of how we can accomplish this (while guaranteeing $O(\lg(n))$ running time for our three existing methods)?

4. Give a pseudo-code implementation of the range query

   $$\text{findAllElementsBetween}(k_1, k_2)$$

   on AVL-trees. What is the running time of your method? Is it possible to implement findAllElementsBetween(k_1, k_2) with a running time of $O(\lg(n))$?

Hint: The number of elements returned by the method has an effect on the running time.

Priority Queues and Heaps

In this lecture, we will discuss another important ADT called *PriorityQueue*. Like stacks and queues, priority queues store arbitrary collections of elements. Recall that stacks have a Last-In First-Out (LIFO) access policy and queues have a First-In First-Out (FIFO) policy. Priority queues have a more complicated policy: each element stored in a priority queue has a *priority*, and the next element to be removed is the one with highest priority.

Priority queues have numerous applications. For example, they can be used to schedule jobs on a shared computer. Some jobs will be more urgent than others and thus get higher priorities. A priority queue keeps track of the jobs to be performed and their priorities. If a job is finished or interrupted, then one with highest priority will be performed next.

6.1 The *PriorityQueue* ADT

A *PriorityQueue* stores a collection of *elements*. Associated with each element is a *key*, which is taken from some linearly ordered set, such as the integers. Keys are just a way to represent the priorities of elements; larger keys mean higher priorities. In its most basic form, the ADT supports the following operations:

- insertItem(k, e): Insert element e with key k.
- maxElement(): Return an element with maximum key; an error occurs if the priority queue is empty.
- removeMax(): Return and remove an element with maximum key; an error occurs if the priority queue is empty.
- isEmpty(): Return TRUE if the priority queue is empty and FALSE otherwise.

Note that the keys have quite a different function here than they have in *Dictionary*. Keys in priority queues are used "internally" to determine the position of an element in the queue (they don't really mean anything when they stand-alone), whereas in dictionaries keys are the main means of accessing elements. Observe that in the *PriorityQueue* ADT, we cannot access elements directly using their keys, we can only take an element that currently has a maximum key (which is something that depends on relative order).

In many textbooks, you'll find priority queues in which a *smaller* key means higher priority and which consequently support methods minElement() and removeMin() instead of maxElement() and removeMax(). This is a superficial distinction: by reversing all comparisons, it is easy to turn one version into the other. More formally, if $\leq$ is the order on our keys then we define the new order $\leq_{\text{rev}}$ by

$$k_1 \leq_{\text{rev}} k_2 \Leftrightarrow k_2 \leq k1.$$

It is a simple exercise to show that $\leq_{\text{rev}}$ is a total order if and only if $\leq$ is.

6.2 The search tree implementation

A straightforward implementation of the *PriorityQueue* ADT is based on binary search trees. If we store the elements according to the value of their keys, with the largest key stored in the rightmost internal vertex of the tree (remember that we have the habit of keeping all the leaf vertices empty), we can easily implement all methods of *PriorityQueue* such that their running time is $\Theta(h)$, where h is the height of the tree (except isEmpty(), which only requires time $\Theta(1)$). If we use AVL trees, this amounts to a running time of $\Theta(\lg(n))$ for insertItem(k, e), maxElement() and removeMax().

6.3 Abstract Heaps

A *heap* is a data structure that is particularly well-suited for implementing the *PriorityQueue* ADT. Although also based on binary trees, heaps are quite different from binary search trees, and they are usually implemented using arrays, which makes them quite efficient in practice. Before we explain the heap data structure, we introduce an abstract model of a heap that will be quite helpful for understanding the priority queue algorithms that we develop here.

We need some more terminology for binary trees: for $i \geq 0$, the ith *level* of a tree consists of all vertices that have distance i from the root. Thus the 0th level consists just of the root, the first level consists of the children of the root, the second level of the grandchildren of the root, etc. Recall that a *complete* Binary Search Tree of height h has $2^{h+1} - 1$ vertices in total (counting both the internal vertices and the leaves). We say that a binary tree of height h (where we have the usual notion of *left* and *right* children) is *almost complete* if levels $0, \ldots, h-1$ have the maximum possible number of vertices (i.e., level i has 2^i vertices), and on level h we have *some of* the possible vertices for that level, such that the vertices (leaves) of level h that *do* belong to the tree at level h appear in a contiguous group to the left of the vertices-positions that *do not* belong to the tree. Figure 6.1 shows an almost complete tree and two trees that are not almost complete.

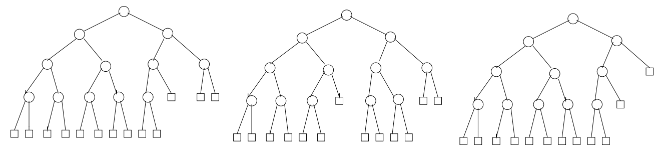

Figure 6.1. The first tree is almost complete; the second and third are not.

Now we have a theorem (similar to proof that AVL trees have $O(\lg n)$ height):
Theorem 6.2. *An almost complete binary tree with n internal vertices has height* $\lfloor \lg(n) \rfloor + 1$.

PROOF We first recall that a complete binary tree (a binary tree where all levels have the maximum possible number of vertices) of height h has $2^{h+1} - 1$ vertices in total (2^h leaves and $2^h - 1$ internal vertices). This can be proved by induction on h.

The number of internal vertices of an almost complete tree of height h is greater than the number of internal vertices of a complete tree of height $h-1$. Moreover it is at most the number of internal vertices of a complete tree of height h. Thus for the number n of vertices of an almost complete tree of height h we have

$$2^{h-1} \leq n \leq 2^h - 1.$$

For all n with $2^{h-1} \leq n \leq 2^h - 1$ we have $\lfloor \lg(n) \rfloor = h-1$; this implies the theorem. $\square$

In our abstract model, a heap is an almost complete binary tree whose internal vertices store items such that the following *heap condition* is satisfied:

(H) For every vertex v except the root, the key stored at v is smaller than or equal to the key stored at the parent of v.

The *last vertex* of a heap of height h is the rightmost internal vertex in the hth level.

Insertion

To insert an item into the heap, we create a new last vertex and insert the item there. It may happen that the key of the new item is larger than the key of its parent, its grandparent, etc. To repair this, we bubble the new item up the heap until it has found its position.

Algorithm insertItem(k, e)

```
1. Create new last vertex $v$.
2. while $v$ is not the root and $k > v.parent.key$ do
3.     store the item stored at $v.parent$ at $v$
4.     $v \leftarrow v.parent$
5. store $(k, e)$ at $v$
```

Algorithm 6.3

Consider the insertion algorithm 6.3. The correctness of the algorithm should be obvious (we could prove it by induction). To analyse the running time, let us assume that each line of code can be executed in time $\Theta(1)$. This needs to be justified, as it depends on the concrete way we store the data. We will do this in the next section. The loop in lines 2–4 is iterated at most h times, where h is the height of the tree. By Theorem 6.2 we have $h = \Theta(\lg(n))$. Thus the running time of insertItem is $\Theta(\lg(n))$.

Finding the Maximum

Property (H) implies that the maximum key is always stored at the root of a heap. Thus the method maxElement just has to return the element stored at the root of the heap, which can be done in time $\Theta(1)$ (with any reasonable implementation of a heap).

Removing the Maximum

The item with the maximum key is stored at the root, but we cannot easily remove the root of a tree. So what we do is replace the item at the root by the item stored at the last vertex, remove the last vertex, and then repair the heap. The repairing is done in a separate procedure called heapify (Algorithm 6.4). Applied to a vertex v of a tree such that the subtrees rooted at the children of v already satisfy the heap property (H), it turns the subtree rooted at v into a tree satisfying (H), without changing the shape of the tree.

Algorithm heapify(v)

```
1. if $v.left$ is an internal vertex and $v.left.key > v.key$ then
2.     $s \leftarrow v.left$
3. else
4.     $s \leftarrow v$
5. if $v.right$ is an internal vertex and $v.right.key > s.key$ then
6.     $s \leftarrow v.right$
7. if $s \neq v$ then
8.     exchange the items of $v$ and $s$
9.     heapify$(s)$
```

Algorithm 6.4

The algorithm for heapify works as follows: In lines 1–6, it defines s to be the largest of the keys of v and its children. Since the children of v may be leaves not storing items, some care is required. If $s = v$, the key of v is larger than or equal to the keys of its children. Thus the heap property (H) is satisfied at v. Since the subtrees rooted at the children are already heaps, (H) is satisfied at every vertex in these subtrees, so it is satisfied in the subtree rooted at v, and no further action is required. If $s \neq v$, then s is the child of v with the larger key. In this case, the items of s and v are exchanged, which means that now (H) is satisfied at v. (H) is still satisfied by the subtree of v that is not rooted by s, because this subtree has not changed. It may be, however, that (H) is no longer satisfied at s, because s has a smaller key now. Therefore, we have to apply heapify recursively to s. Eventually, we will reach a point where we call heapify(v) for a v that has no internal vertices as children, and the algorithm terminates.

The running time of heapify can be easily expressed in terms of the height h of v. Again assuming that each line of code requires time $\Theta(1)$ and observing that the recursive call is made to a vertex of height $h-1$, we obtain

$$T_{\text{heapify}}(h) = \Theta(1) + T_{\text{heapify}}(h-1).$$

Moreover, the recursive call is only made if the height of v is greater than 1, thus heapify applied to a vertex of height 1 only requires constant time. To solve the recurrence, we use the following lemma:

Lemma 6.5. *Let $f : \mathbb{N} \to \mathbb{R}$ such that for all $n \geq 1$*

$$f(n) = \Theta(1) + f(n-1).$$

Then $f(n)$ *is* $\Theta(n)$.

PROOF We first prove that $f(n)$ is $O(n)$. Let $n_0 \in \mathbb{N}$ and $c > 0$ such that $f(n) \leq c + f(n-1)$ for all $n \geq n_0$. Let $d = f(n_0)$. We claim that

$$f(n) \leq d + c\,(n - n_0) \tag{6.1}$$

for all $n \geq n_0$.

We prove (6.1) by induction on $n \geq n_0$. As the induction basis, we note that (6.1) holds for $n = n_0$.

For the induction step, suppose that $n > n_0$ and that (6.1) holds for $n-1$. Then

$$f(n) \leq c + f(n-1) \leq c + d + c\,(n-1-n_0) = d + c\,(n-n_0).$$

This proves (6.1).

To complete the proof that $f(n)$ is $O(n)$, we just note that

$$d + c\,(n - n_0) \leq d + c\,n = O(n).$$

The proof that $f(n)$ is $\Omega(n)$ can be done completely analogously and is left as an exercise. □

The Lemma should be intuitively obvious: we have $f(n) = \Theta(1) + f(n-1)$. Remember that $\Theta(1)$ means essentially a non-zero constant[1]. So, unwinding the recurrence, we have

$$f(n) = \underbrace{\Theta(1) + \Theta(1) + \cdots + \Theta(1)}_{n \text{ times}} + f(0).$$

Of course $f(0)$ is just a constant so it is clear that the expression on the r.h.s. is $\Theta(n)$. An experienced theoretician would not bother with the proof by induction, not because of laziness but because of genuine understanding. (The acid test for this is to be able to supply the proof if necessary.)

Applying Lemma 6.5 to the function $T_{\mathsf{heapify}}(h)$ yields

$$T_{\mathsf{heapify}}(h) = \Theta(h). \tag{6.2}$$

Finally, Algorithm 6.3 shows how to use heapify for removing the maximum.

Since the height of the root of the heap, which is just the height of the heap, is $\Theta(\lg(n))$ by Theorem 6.2, the running time of removeMax is $\Theta(\lg(n))$.

6.4 An array based implementation

The last section gives algorithms for a tree-based heap data structure. However, one of the reasons that heaps are so efficient is that they can be stored in arrays in a straightforward way. The items of the heap are stored in an array H, and the internal vertices of the tree are associated with the indices of the array by numbering them level-wise from the left to the right. Thus the root is associated with 0, the left child of the root with 1, the right child of the root with 2, the leftmost grandchild of the root with 3, etc. (see Figure 6.7). For every vertex v the left child

[1]Strictly speaking a function such as $1 + 1/(n+1)$ is $\Theta(1)$ but of course is not constant. However this function is bounded form below by 1 and from above by 2 and it does no harm to think of it as a constant in our context. The same applies to any $\Theta(1)$ function.

```
Algorithm removeMax()
  1. $e \leftarrow root.element$
  2. $root.item \leftarrow last.item$
  3. delete $last$
  4. heapify($root$)
  5. return e;
```

Algorithm 6.6

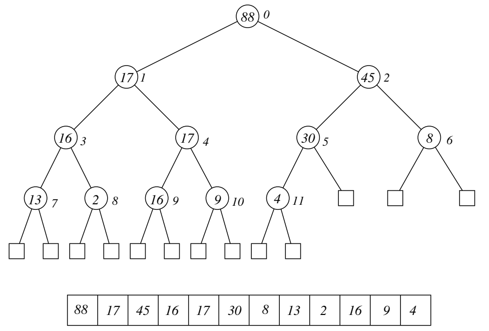

Figure 6.7. Storing a heap in an array. The small numbers next to the vertices are the indices associated with them.

of v is $2v+1$ and the right child of v is $2v+2$ (if they are internal vertices). The root is 0, and for every vertex $v > 0$ the parent of v is $\lfloor \frac{v-1}{2} \rfloor$. We store the number of items the heap contains in a variable $size$. Thus the last vertex is $size - 1$.

Observe that we only have to insert or remove the last element of the heap. Using dynamic arrays, this can be done in amortised time $\Theta(1)$. Therefore, using the algorithms described in the previous section, the methods of *PriorityQueue* can be implemented with the running times shown in Table 6.8. In many important priority queue applications we know the size of the queue in advance, so we can fix the array size when we set up the priority queue (no need for a dynamic array). In this situation, the running times $\Theta(\lg(n))$ for insertItem and removeMax are not amortised, but simple worst case running times.

Algorithm 6.9 gives the pseudocode to take an array H storing items and turn it into a heap. This method is very useful in applications:

To understand buildHeap, observe that $\lfloor \frac{n-2}{2} \rfloor$ is the maximum v such that $2v + 1 \leq n - 1$, i.e., the last vertex of the heap that has at least one child. Recall that if heapifyv is applied to a vertex v of a tree such that the subtrees rooted at the children of v already satisfy the heap property (H), this turns the subtree rooted at v into a tree satisfying (H). So the algorithm turns all subtrees of the tree into a heap in a bottom up fashion. Since subtrees of height 1 are automatically heaps,

method	running time
insertItem	$\Theta(\lg(n))$ amortised
maxElement	$\Theta(1)$
removeMax	$\Theta(\lg(n))$ amortised
isEmpty	$4\Theta(1)$

Table 6.8.

Algorithm buildHeap(H)

```
1. $n \leftarrow H.length$
2. **for** $v \leftarrow \lfloor \frac{n-2}{2} \rfloor$ **downto** $0$ **do**
3.     heapify($v$)
```

Algorithm 6.9

the first vertex where something needs to be done is $\lfloor \frac{n-2}{2} \rfloor$, the last vertex of height at least 2.

Straightforward reasoning shows that the running time of heapify() is

$$O(n \lg(n)),$$

because each call of heapify requires time $O(\lg(n))$. Surprisingly, a more refined argument shows that we can do better:

Theorem 6.10. *The running time of buildHeap is $\Theta(n)$, where n is the length of the array H.*

PROOF Let $h(v)$ denote the height of a vertex v and $m = \lfloor \frac{n-2}{2} \rfloor$. By 6.2, we obtain

$$\begin{aligned} T_{\text{buildHeap}}(n) &= \sum_{v=0}^{m} \Theta(1) + T_{\text{heapify}}(h(v)) \\ &= \Theta(m) + \sum_{v=0}^{m} \Theta\big(h(v)\big) \\ &= \Theta(m) + \Theta\Big(\sum_{v=0}^{m} h(v)\Big). \end{aligned}$$

Since $\Theta(m) = \Theta(n)$, all that remains to prove is that $\sum_{v=0}^{m} h(v) = O(n)$. Observe that

$$\begin{aligned} \sum_{v=0}^{m} h(v) &= \sum_{i=1}^{h} i \cdot \big(\text{number of vertices of height } i\big) \\ &= \sum_{i=0}^{h-1} (h-i) \cdot \big(\text{number of vertices on level } i\big) \end{aligned}$$

$$\le \sum_{i=0}^{h-1} (h-i) \cdot 2^i,$$

where $h = \lfloor \lg(n) \rfloor + 1$. The rest is just calculation. We use the fact that

$$\sum_{i=0}^{\infty} \frac{i+1}{2^i} = 4. \tag{6.3}$$

Then

$$\begin{aligned} \sum_{i=0}^{h-1} (h-i)\, 2^i &= n \frac{\sum_{i=0}^{h-1} (h-i)\, 2^i}{n} = n \sum_{i=0}^{h-1} (h-i) \frac{2^i}{n} \\ &\le n \sum_{i=0}^{h-1} (h-i)\, 2^{i-(h-1)} \qquad \text{(since } n \ge 2^{h-1}\text{)} \\ &= n \sum_{i=0}^{h-1} (i+1)\, 2^{-i} \\ &\le n \sum_{i=0}^{\infty} \frac{i+1}{2^i} \\ &= 4n. \end{aligned}$$

Thus $\sum_{v=0}^{m} h(v) \le 4n = O(n)$, which completes our proof. □

6.5 Further reading

If you have [GT], there is an entire chapter on "Priority Queues" (with details about Heaps).

If you have [CLRS], look at the chapter titled "Heapsort" (ignore the sorting for now).

Exercises

1. Where may an item with minimum key be stored in a heap?
2. Suppose the following operations are performed on an initially empty heap:
   - Insertion of elements with keys $20, 21, 3, 12, 18, 5, 24, 30$.
   - Two removeMax operations.

   Show how the data structure evolves, both as an abstract tree and as an array.
3. Show that for any n there is a sequence of n insertions into an initially empty heap that requires time $\Omega(n \lg n)$ to process.

   Remark: Compare this to the method buildHeap with its running time of $\Theta(n)$.
4. Let $S = \sum_{i=0}^{\infty} \frac{i+1}{2^i}$, we assume without proof that the series does indeed converge (this is easy to show). Prove that $S = 1 + \frac{1}{2}(S + \sum_{i=0}^{\infty} \frac{1}{2^i}) = 2 + \frac{1}{2}S$. Deduce that $S = 4$.

Sorting, Merge Sort and the Divide-and-Conquer Technique

This and a subsequent next lecture will mainly be concerned with *sorting algorithms*. Sorting is an extremely important algorithmic problem that often appears as part of a more complicated task. Quite frequently, huge amounts of data have to be sorted. It is therefore worthwhile to put some effort into designing efficient sorting algorithms.

In this lecture, we will mainly consider the mergeSort algorithm. mergeSort is based on the principle of *divide-and-conquer*. Therefore in this note we will also discuss divide-and-conquer algorithms and their analysis in general.

7.1 The Sorting Problem

The problem we are considering here is that of sorting a collection of *items* by their *keys*, which are assumed to be comparable. We assume that the items are stored in an array that we will always denote by A. The number of items to be sorted is always denoted by n. Of course it is also conceivable that the items are stored in a linked list or some other data structure, but we focus on arrays. Most of our algorithms can be adapted for sorting linked lists, although this may go along with a loss in efficiency. By assuming we work with arrays, we assume we may keep all the items in memory at the same time. This is not the case for large-scale sorting, as we will see in a later lecture.

We already saw the insertionSort sorting algorithm in Lecture 2 of this thread. It is displayed again here as Algorithm 1. Recall that the asymptotic worst-case

```
Algorithm insertionSort($A$)
1. for $j \leftarrow 1$ to $A.length - 1$ do
2.     $a \leftarrow A[j]$
3.     $i \leftarrow j - 1$
4.     while $i \geq 0$ and $A[i].key > a.key$ do
5.         $A[i+1] \leftarrow A[i]$
6.         $i \leftarrow i - 1$
7.     $A[i+1] \leftarrow a$
```

Algorithm 1

running time of insertionSort is $\Theta(n^2)$. For a sorting algorithm, this is quite poor. We will see a couple of algorithms that do much better.

7.2 Important characteristics of Sorting Algorithms

The main characteristic we will be interested in for the various sorting algorithms we study is worst-case running-time (we will also refer to best-case and average-case from time to time). However, we will also be interested in whether our sorting algorithm is *in-place* and whether it is *stable*.

Definition 2 *An sorting algorithm is said to be an* in-place *algorithm if it can be (simply) implemented on the input array, with only $O(1)$ extra space (extra variables).*

Clearly it is useful to have an in-place algorithm if we want to minimize the amount of space used by our algorithm. The issue of minimizing space only really becomes important when we have a large amount of data, for applications such as web-indexing.

Definition 3 *A sorting algorithm is said to be* stable *if for every i, j such that $i < j$ and $A[i].key = A[j].key$, we are guaranteed that $A[i]$ comes before $A[j]$ in the output array.*

Stability is a "desirable" property for a sorting algorithm. We won't really see why it is useful in Inf2B (if you take the 3rd-year *Algorithms and Data Structures* course, you'll see that it can lead to a powerful reduction in running-time for some cases of sorting[1]).

If we refer back to insertionSort, we can see that it is clearly an *in-place* sorting algorithm, as it operates directly on the input array. Also, notice that the test $A[i].key > a.key$ in line 4. is a strict inequality and so that when $A[i]$ is inserted into the (already sorted) array $A[1 \ldots i-1]$, it will be inserted *after* any element with the same key within $A[1 \ldots i-1]$. Hence insertionSort is stable.

7.3 Merge Sort

The idea of merge sort is very simple: split the array A to be sorted into halves, sort both halves recursively, and then *merge* the two sorted subarrays together to one sorted array. Algorithm 4 implements this idea in a straightforward manner. To make the recursive implementation possible, we introduce two additional parameters i and j marking the boundaries of the subarray to be sorted; mergeSort(A, i, j) sorts the subarray $A[i \ldots j] = \langle A[i], A[i+1], \ldots, A[j] \rangle$. To sort the whole array, we obviously have to call mergeSort($A, 0, n-1$).

The key to MergeSort is the merge() method, called as merge(A, i, mid, j) in Algorithm 4. Assuming that $i \leq mid < j$ and that the subarrays $A[i \ldots mid]$ and $A[mid{+}1 \ldots j]$ are both sorted, merge(A, i, mid, j) sorts the whole subarray $A[i \ldots j]$. The merging is achieved by first defining a new array B to hold the sorted data. Then we initialise an index k for the $A[i \ldots mid]$ subarray to i, and index ℓ for the $A[mid+1 \ldots j]$ subarray to $mid + 1$. We walk these indices up the array, at each step storing the minimum of $A[k]$ and $A[\ell]$ in B, and then incrementing that index and the "current index" of B.

[1]This is the Radix-sort algorithm, if you want to look this up.

```
Algorithm mergeSort$(A, i, j)$

 1. if $i < j$ then
 2.     $mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$
 3.     mergeSort$(A, i, mid)$
 4.     mergeSort$(A, mid+1, j)$
 5.     merge$(A, i, mid, j)$
```

Algorithm 4

Algorithm 5 shows an implementation of merge(). Note that at the end of the initial merge (loop on line 3) there might be some entries left in the lower half of A or the upper half but not both; so at most one of the loops on lines 11 and 15 would be executed (these just append any left over entries to the end of B). Figure 6 shows the first few steps of merge() on an example array.

Let us analyse the running time of merge. Let $n = j-i+1$, i.e., n is the number of entries in the subarray $A[i \dots j]$. There are no nested loops, and clearly each of the loops of merge is iterated at most n times. Thus the running time of merge is $O(n)$. Since the last loop is certainly executed n times, the running time is also $\Omega(n)$ and thus $\Theta(n)$. Actually, the running time of merge is $\Theta(n)$ no matter what the input array A is (i.e., even in the best case).

Now let us look at mergeSort. Again we let $n = j - i + 1$. We get the following expression for the running time $T_{\text{mergeSort}}(n)$:

$$\begin{aligned} T_{\text{mergeSort}}(1) &= \Theta(1), \\ T_{\text{mergeSort}}(n) &= \Theta(1) + T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + T_{\text{merge}}(n). \end{aligned}$$

Since we already know that the running time of merge is $\Theta(n)$, we can simplify this to

$$T_{\text{mergeSort}}(n) = T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n).$$

We will see in §7.5 that solving this recurrence yields

$$T_{\text{mergeSort}}(n) = \Theta(n \lg(n)).$$

Our analysis will actually show that the running time of mergeSort is $\Theta(n \lg(n))$ no matter what the input is, i.e., even in the best case.

The runtime of $\Theta(n \lg(n))$ is not surprising. The recurrence shows that for an input of size n the cost is that of the two recursive calls plus a cost proportional to n. Now each recursive call is to inputs of size essentially $n/2$. Each of these makes two recursive calls for size $n/4$ and we also charge a cost proportional to $n/2$. So the cost of the next level is a one off cost proportional to n (taking account of the two $n/2$ one off costs) and 4 calls to size $n/4$. So we see that at each recursive call we pay a one off cost that is proportional to n and then make 2^r calls to inputs of size $n/2^r$. The base case is reached when $n/2^r = 1$, i.e., when $r = \lg(n)$. So we have $\lg(n)$ levels each of which has a cost proportional to n.

```
Algorithm merge$(A, i, mid, j)$

 1. initialise new array $B$ of length $j - i + 1$
 2. $k \leftarrow i$; $\ell \leftarrow mid + 1$; $m \leftarrow 0$
 3. while $k \le mid$ and $\ell \le j$ do (merge halves of $A$ into $B$)
 4.     if $A[k].key \le A[\ell].key$ then
 5.         $B[m] \leftarrow A[k]$
 6.         $k \leftarrow k + 1$
 7.     else
 8.         $B[m] \leftarrow A[\ell]$
 9.         $\ell \leftarrow \ell + 1$
10.     $m \leftarrow m + 1$
11. while $k \le mid$ do (copy anything left in lower half of $A$ to $B$)
12.     $B[m] \leftarrow A[k]$
13.     $k \leftarrow k + 1$
14.     $m \leftarrow m + 1$
15. while $\ell \le j$ do (copy anything left in upper half of $A$ to $B$)
16.     $B[m] \leftarrow A[\ell]$
17.     $\ell \leftarrow \ell + 1$
18.     $m \leftarrow m + 1$
19. for $m = 0$ to $j - i$ do (copy $B$ back to $A$)
20.     $A[m + i] \leftarrow B[m]$
```

Algorithm 5

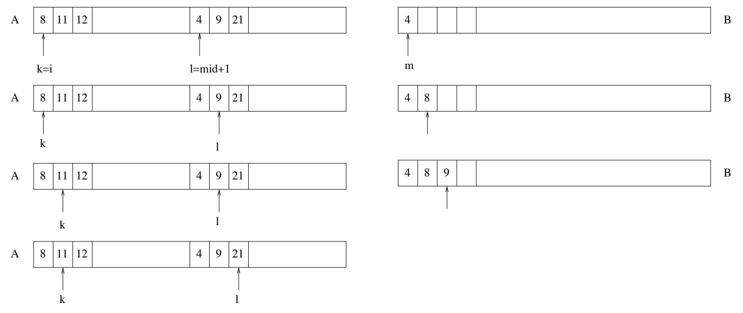

Figure 6. The first few steps of merge() on two component subarrays.

Thus the total cost is proportional to $n\lg(n)$. We will prove the correctness of this sketch argument below.

Since $n\lg(n)$ grows much slower than n^2, mergeSort is much more efficient than insertionSort. As simple experiments show, this can also be observed in practice. So usually mergeSort is preferable to insertionSort. However, there are situations where this is not so clear. Suppose we want to sort an array that is almost sorted, with only a few items that are slightly out of place. For such "almost sorted" arrays, insertionSort works in "almost linear time", because very few items have to be moved. mergeSort, on the other hand, uses time $\Theta(n\lg(n))$ *no matter what the input array looks like*. insertionSort also tends to be faster on very small input arrays, because there is a certain overhead caused by the recursive calls in mergeSort.

Maybe most importantly, mergeSort wastes a lot of space, because merge copies the whole array to an intermediate array. Clearly mergeSort is *not* an in-place algorithm in the sense of Definition 2. As to the question of stability for mergeSort, line 4 of merge achieves stability for mergeSort *only because* it has a *non-strict* inequality for testing when we should put the item $A[k]$ (from the left subarray) down before the item $A[\ell]$ (from the right subarray). In some places you may see non-stable versions of mergeSort, where a strict inequality is used in line 4.

7.4 Divide-and-Conquer Algorithms

The *divide-and-conquer* technique involves solving a problem in the following way:

(1) Divide the input instance into several instances of the same problem of smaller size.

(2) Recursively solve the problem on these smaller instances.

(3) Combine the solutions for the smaller instances to a solution for the original instance (so after the recursive calls, we do some "extra work" to find the solution for our original instance).

Obviously, mergeSort is an example of a divide-and-conquer algorithm.

Suppose now that we want to analyse the running time of a divide-and-conquer algorithm. Denote the size of the input by n. Furthermore, suppose that the input instance is split into a instances of sizes $n_1, \dots, n_a$, for some constant $a \geq 1$. Then we get the following recurrence for a running time $T(n)$:

$$T(n) = T(n_1) + \ldots + T(n_a) + f(n),$$

where $f(n)$ is the time required by steps (1) (for setting up the recursions) and (3) (for the "extra work"). Typically, $n_1, \dots, n_k$ are all of the form $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ for some $b > 1$. Disregarding floors and ceilings for a moment, we get a recurrence of the form:

$$T(n) = a\,T(n/b) + f(n).$$

If we want to take floors and ceilings into account, we have to replace this by $T(n) = a_1T(\lfloor n/b \rfloor) + a_2 \cdot T(\lceil n/b \rceil) + f(n)$. It is usually the case that the asymptotic growth rate of the solution to the recurrence does not depend on floors and ceilings. Throughout this lecture note we will usually disregard the floors and ceilings (this can be justified formally).

To specify the recurrence fully, we need also to say what happens for small instances that can no longer be divided, and we must specify the function f. We can always assume that for input instances of size below some constant n_0 (typically, $n_0 = 2$), the algorithm requires $\Theta(1)$ steps. A typical function f occurring in the analysis of divide-and-conquer algorithms has an asymptotic growth rate of $\Theta(n^k)$, for some constant k.

To conclude: The analysis of divide-and-conquer algorithms usually yields recurrences of the form

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ a\,T(n/b) + \Theta(n^k) & \text{if } n \geq n_0, \end{cases} \tag{7.1}$$

where $n_0, a \in \mathbb{N}$, $k \in \mathbb{N}_0$, and $b \in \mathbb{R}$ with $b > 1$ are constants. (We normally assume that n is a power of b so that n/b is an integer as the recurrence "unwinds." Without this assumption we must use floor or ceiling functions as appropriate. The fact is that the asymptotic growth rate for the type of recurrence shown; see CLRS for a discussion.)

7.5 Solving Recurrences

Example 7. Recall that for the running time of merge sort we got the following recurrence:

$$T_{\text{mergeSort}}(n) = \begin{cases} \Theta(1) & \text{if } n < 2, \\ T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

We first assume that n is a power of 2, say, $n = 2^\ell$; in this case we can omit floors and ceilings. Then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2\big(2T(n/4) + \Theta(n/2)\big) + \Theta(n) \\ &= 4T(n/4) + 2\Theta(n/2) + \Theta(n) \\ &= 4\big(2T(n/8) + \Theta(n/4)\big) + 2\Theta(n/2) + \Theta(n) \\ &= 8T(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \\ &\vdots \\ &= 2^kT(n/2^k) + \sum_{i=0}^{k-1} 2^i\Theta(n/2^i) \\ &\vdots \\ &= 2^\ell T(n/2^\ell) + \sum_{i=0}^{\ell-1} 2^i\Theta(n/2^i) \end{aligned}$$

$$\begin{aligned}
&= nT(1) + \sum_{i=0}^{\ell-1} 2^i \Theta(n/2^i) \\
&= n\Theta(1) + \sum_{i=0}^{\lg(n)-1} 2^i \Theta(n/2^i) \\
&= \Theta(n) + \Theta(n\lg(n)) \\
&= \Theta(n\lg(n)).
\end{aligned}$$

Strictly speaking the argument just given, usually referred to as unwinding the recurrence, is incomplete (those dots in the middle). We have taken it on trust that the last lines are the correct outcome. We can easily fill in this "gap" using induction (e.g. by proving that the claimed solution is indeed correct). However as long as the unwinding process is carried out carefully the induction step is a matter of routine and is usually omitted. You should however be able to supply it if asked.

Although we have only treated the special case when n is a power of 2, we can now use a trick to prove that we have $T(n) = \Theta(n \lg n)$ for all n. We make the reasonable assumption that $T(n) \le T(n+1)$ (this can be proved from the recurrence for $T(n)$ by induction). Now notice that for every n, there is some exponent k such that $n \le 2^k < 2n$. This is the fact we need. We observe that $T(n) \le T(2^k) = O(2^k \lg(2^k)) = O(2n\lg(2n)) = O(n \lg n)$. Likewise for every $n \ge 1$ there is some k' such that $n/2 < 2^{k'} \le n$. Hence $T(n) = \Omega((2^{k'})\lg(2^{k'})) = \Omega((n/2)\lg(n/2)) = \Omega(n \lg n)$. Putting these two together we have $T(n) = \Theta(n \lg n)$.

An alternative approach is to use the full recurrence directly though this is rather tedious. Note though that once we have a good guess at what the solution is we can prove its correctness by using induction and the full recurrence, we do need to unwind the complicated version.

Before moving on it is worth explaining one possible error that is sometimes made when unwinding recurrences. We have

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= 2\big(2T(n/4) + \Theta(n/2)\big) + \Theta(n) \\
&= 4T(n/4) + 2\Theta(n/2) + \Theta(n)
\end{aligned}$$

It is very tempting to simplify the last line to $4T(n/4) + \Theta(n)$ on the basis that $2\Theta(n/2) + \Theta(n) = \Theta(n)$ and do likewise at each stage. It is certainly true that $2\Theta(n/2) + \Theta(n) = \Theta(n)$. However such replacements are only justified if we have a *constant* number of $\Theta(\cdot)$ terms. In our case we unwind the recurrence to obtain $\lg(n)$ terms and this is not a constant. You are strongly encouraged to unwind the recurrence with the $\Theta(n)$ replaced by cn where $c > 0$ is some constant. Observe that at each stage we obtain a multiple of cn as an additive term (cn, $2cn$, $3cn$ etc.). Of course after any fixed number of stages we just have a constant multiple of n. But after $\log(n)$ stages this is not the case.

By further exploiting the idea used in this example, one can prove the following general theorem that gives solutions to recurrences of the form (7.1). Recall that $\log_b(a)$ denotes the logarithm of a with base b, i.e., we have

$$c = \log_b(a) \iff b^c = a.$$

For example, $\log_2(8) = 3$, $\log_3(9) = 2$, $\log_4(8) = 1.5$.

The following theorem is called the *Master Theorem* (for solving recurrences). It makes life easy by allowing us to "read-off" the $\Theta(\cdot)$ expression of a recurrence without going through a proof as we did for mergeSort:

Theorem 8. *Let $n_0 \in \mathbb{N}$, $k \in \mathbb{N}_0$ and $a, b \in \mathbb{R}$ with $a > 0$ and $b > 1$, and let $T : \mathbb{N} \to \mathbb{N}$ satisfy the following recurrence:*

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < n_0; \\ aT(n/b) + \Theta(n^k), & \text{if } n \ge n_0. \end{cases}$$

Let $e = \log_b(a)$; we call e the critical exponent. *Then*

$$T(n) = \begin{cases} \Theta(n^e), & \text{if } k < e \quad \text{(I)}; \\ \Theta(n^e \lg(n)), & \text{if } k = e \quad \text{(II)}; \\ \Theta(n^k), & \text{if } k > e \quad \text{(III)}. \end{cases}$$

The theorem remains true if we replace $aT(n/b)$ in the recurrence by $a_1T(\lfloor n/b \rfloor) + a_2T(\lceil n/b \rceil)$ for $a_1, a_2 \ge 0$ with $a_1 + a_2 = a$.

In some situations we do not have a $\Theta(n^k)$ expression for the extra cost part of the recurrence but do have a $O(n^k)$ expression. This is fine, the Mater Theorem still applies but gives us just an upper bound for the runtime, i.e., in the three cases for the value of $T(n)$ replace Θ by O.

Examples 9.

(1) Now reconsider the recurrence for the running time of mergeSort:

$$T_{\text{mergeSort}}(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n), & \text{if } n \ge 2. \end{cases}$$

In the setting of the Master Theorem, we have $n_0 = 2$, $k = 1$, $a = 2$, $b = 2$. Thus $e = \log_b(a) = \log_2(2) = \lg(2) = 1$. Hence $T_{\text{mergeSort}}(n) = \Theta(n\lg(n))$ by case (II).

(2) Recall the recurrence for the running time of binary search:

$$T_{\text{binarySearch}}(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ T_{\text{binarySearch}}(\lfloor n/2 \rfloor) + \Theta(1), & \text{if } n \ge 2. \end{cases}$$

Here we take $n_0 = 2$, $k = 0$, $a = 1$, $b = 2$. Thus $e = \log_b(a) = \log_2(1) = 0$ and therefore $T_{\text{binarySearch}}(n) \in \Theta(\lg(n))$ by case (II) of the Master Theorem.

(3) Let T be a function satisfying

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ 3T(n/2) + \Theta(n), & \text{if } n \ge n_0. \end{cases}$$

Then $e = \log_b(a) = \log_2(3) = \lg 3$. Hence $T(n) \in \Theta(n^{\lg(3)})$ by case (I).

(4) Let T be a function satisfying

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ 7T(n/2) + \Theta(n^4), & \text{if } n \ge n_0. \end{cases}$$

Then $e = \log_b(a) = \log_2(7) < 3$. Hence $T(n) \in \Theta(n^4)$ by case (III) of the Master Theorem.

Exercises

1. Solve the following recurrences using the Master Theorem:

   (a) $T(n) = 3T(n/2) + n^5$

   (b) $T(n) = 4T(n/3) + n$

   (c) $T(n) = 8T(n/3) + n^2$

   (d) $T(n) = 8T(n/3) + n$

   (e) $T(n) = 8T(n/3) + n\lg(n)$
   Remark: The Master Theorem is not directly applicable in this case. Find a way to use it anyway. In each case $T(n)$ is assumed to be $\Theta(1)$ for some appropriate base case.

2. Let $T(n)$ be the function satisfying the following recurrence:

$$T(n) = \begin{cases} 3, & \text{if } n = 1; \\ T(n-1) + 2n, & \text{if } n \geq 2. \end{cases}$$

   Solve the recurrence and give an explicit formula for T.

   Try to derive a general formula for the asymptotic growth rate of functions satisfying recurrences of the form

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq n_0; \\ T(n-k) + f(n), & \text{if } n > n_0, \end{cases}$$

   where $k, n_0 \in \mathbb{N}$ such that $1 \leq k \leq n_0$ and $f : \mathbb{N} \to \mathbb{N}$ is a function.

3. Prove that $\sum_{i=0}^{\lg(n)-1} 2^i \Theta(n/2^i) = \Theta(n \lg(n))$, this was used in solving the recurrence for $T_{\mathsf{mergeSort}}(n)$. This is much easier than it looks, remember that to say $f = \Theta(g)$ means there is an $n_0 \in \mathbb{N}$ and constants $c_1, c_2 \in \mathbb{R}$ both strictly bigger than 0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Recall also the formula for summing geometric series: $\sum_{i=0}^{s} r^i = (1 - r^{i+1})/(1 - r)$.

Heapsort and Quicksort

We will see two more sorting algorithms in this lecture. The first, heapSort, is very interesting theoretically. It sorts an array with n items in time $\Theta(n \lg n)$ *in place* (i.e., it doesn't need much extra memory). The second algorithm, quickSort, is known for being the most efficient sorting algorithm in practice, although it has a rather high worst-case running time of $\Theta(n^2)$.

8.1 Heapsort

To understand the basic principle behind heapSort, it is best to recall maxSort (Algorithm 8.1), a simple but slow sorting algorithm (given as Exercise 4 of Lecture Note 2). The algorithm maxSort repeatedly picks the maximum key in the subarray it currently considers and puts it in the last place of this subarray, where it belongs. Then it continues with the subarray containing all but the last item.

```
Algorithm maxSort($A$)
 1. for $j \leftarrow A.length - 1$ downto $1$ do
 2.     $m \leftarrow 0$
 3.     for $i = 1$ to $j$ do
 4.         if $A[i].key > A[m].key$ then $m \leftarrow i$
 5.     exchange $A[m], A[j]$
```

Algorithm 8.1

The algorithm heapSort follows the same principle, but uses a heap to find efficiently the maximum in each step. We will define heapSort by building on the methods of the Lecture Note on Priority Queues and Heaps. However, for heapSort, we only need the following two methods:

- removeMax(): Return and remove an item with maximum key.
- buildHeap(A) (and by implication, heapify()): Turn array A into a heap.

Notice that because we provide all the items at the beginning of the algorithm (our use of the heap is *static* rather than *dynamic*), we do not need the method insertItem() as an individual method. With the implementations explained in the Lecture Note on Priority Queues and Heaps, removeMax() has a running time of $\Theta(\lg n)$ and buildHeap has a running time of $\Theta(n)$.

With these methods available, the implementation of heapSort is very simple. Consider Algorithm 8.2. To see that it works correctly, observe that an array A of size n is in sorted (increasing) order *if and only if* for every j with $1 \le j \le A.length - 1 = n-1$, the entry $A[j]$ contains a maximum element of the subarray $A[0 \ldots j]$.

Line 3 of heapSort() ensures that at each index from $A.length - 1$ down to 0, we always insert a maximum element of the remaining collection of elements into $A[j]$ (and remove it from the heap, reducing the collection of elements there). Hence it sorts correctly.

```
Algorithm heapSort($A$)
 1. buildHeap($A$)
 2. for $j \leftarrow A.length - 1$ downto $0$ do
 3.     $A[j] \leftarrow$ removeMax()
```

Algorithm 8.2

The analysis is easy: Line 1 requires time $\Theta(n)$, as shown at the end of the Lecture Note on Priority Queues and Heaps. An iteration of the loop in lines 2–3 for a particular index j requires time $\Theta(\lg j)$ (because we work from the top of the array down, therefore when we consider index j, the heap contains only $j+1$ elements). Thus we get

$$T_{\text{heapSort}}(n) = \Theta(n) + \sum_{j=2}^{n} \Theta(\lg(j)) = \Theta(n) + \Theta\Big(\sum_{j=2}^{n} \lg(j)\Big).$$

Since for $j \le n$ we have $\lg(j) \le \lg(n)$,

$$\sum_{j=2}^{n} \lg(j) \le (n-1)\lg(n) = O(n \lg n).$$

Since for $j \ge n/2$ we have $\lg(j) \ge \lg(n) - 1$,

$$\sum_{j=2}^{n} \lg(j) \ge \sum_{j=\lceil n/2 \rceil}^{n} (\lg(n) - 1) = \lfloor n/2 \rfloor (\lg(n) - 1) = \Omega(n \lg(n)).$$

Therefore $\sum_{j=2}^{n} \lg(j) = \Theta(n \lg n)$ and

$$T_{\text{heapSort}}(n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n).$$

In the case where all the keys are *distinct*, we are guaranteed that for a constant fraction of the removeMax calls, the call takes $\Omega(\lg \widehat{n})$ time, where $\widehat{n}$ is the number of items at the time of the call. This is not obvious, but it is true—it is because we copy the "last" cell over onto the root before calling heapify. So in the case of distinct keys, the "best case" for heapSort is also $\Omega(n \lg n)$ [1].

By recalling the algorithms for buildHeap and (the array-based version of) removeMax it is easy to verify that heapSort sorts in place. After some thought it can be seen that heapSort is *not* stable.

[1] This is difficult to prove. The result is due to B. Bollobas, T.I. Fenner and A.M. Frieze, "On the Best Case of Heapsort", *Journal of Algorithms*, **20**(2), 1996.

8.2 Quicksort

The sorting algorithm quickSort, like mergeSort, is based on the *divide-and-conquer* paradigm. As opposed to mergeSort and heapSort, quickSort has a relatively bad worst case running time of $\Theta(n^2)$. However, quickSort is very fast in practice, hence the name. Theoretical evidence for this behaviour can be provided by an *average case analysis*. The average-case analysis of quickSort is too technical for *Informatics 2B*, so we will only consider worst-case and best-case here. If you take the 3rd-year *Algorithms and Data Structures (ADS)* course, you will see the average-case analysis there.

The algorithm quickSort works as follows:

(1) If the input array has less than two elements, there is nothing to do.

Otherwise, *partition* the array as follows: Pick a particular key called the *pivot* and divide the array into two subarrays, of which the first only contains items whose key is smaller than or equal to the pivot and the second only items whose key is greater than or equal to the pivot.

(2) Sort the two subarrays recursively.

Note that quickSort does most work in the "divide" step (i.e., in the partitioning routine), whereas in mergeSort the dividing is trivial, but the "conquer" step must reassemble the recursively sorted subarrays using the merge method. This is not necessary in quickSort, because after the first step all elements in the first subarray are smaller than those in the second subarray. A problem, which is responsible for the bad worst-case running time of quickSort, is that the partitioning step is not guaranteed to divide the array into two subarrays of the same size (if we could enforce this somehow, we would have a $\Theta(n \lg n)$ algorithm). If we implement partitioning in an obvious way, all items can end up in one of the two subarrays, and we only reduce the size of our problem by 1.

Algorithm 8.3 is a pseudo-code implementation of the main algorithm.

```
Algorithm quickSort($A, i, j$)

1. if $i < j$ then
2.         $split \leftarrow$ partition($A, i, j$)
3.         quickSort($A, i, split$)
4.         quickSort($A, split + 1, j$)
```

Algorithm 8.3

All the work is done in the partitioning routine. First, the routine must pick a pivot. The simplest choice is to pick the key of the first element. We want to avoid setting up a new, temporary array, because that would be a waste of memory space and also time (e.g., for initialising the new array and copying elements back to the old one in the end). Algorithm 8.4 is an in-place partitioning routine. Figure 8.5 illustrates how partition works.

```
Algorithm partition($A, i, j$)

1. $pivot \leftarrow A[i].key$
2. $p \leftarrow i - 1$
3. $q \leftarrow j + 1$
4. while TRUE do
5.         do $q \leftarrow q - 1$ while $A[q].key > pivot$
6.         do $p \leftarrow p + 1$ while $A[p].key < pivot$
7.         if $p < q$ then
8.                 exchange $A[p]$, $A[q]$
9.         else return $q$
```

Algorithm 8.4

To see that partition works correctly, we observe that after each iteration of the main loop in lines 4–8, for all indices r such that $q < r \leq j$ we have $A[r].key \geq pivot$, and for all r such that $i \leq r < p$ we have $A[r].key \leq pivot$. Actually, after all iterations except maybe the last we also have $A[p].key \leq pivot$. Formally, we can establish this by an induction on the number of iterations of the loop. In addition, we must verify that our return value q is greater than or equal to i and smaller than $j - 1$. It is greater than or equal to i because at any time during the computation we have $A[i].key \leq pivot$. The q returned is smaller than j, because after the first iteration of the main loop we have $q \leq j$ and $p = 1$. If q is already smaller than j, it remains smaller. If not, we still have $p < q$. So there will be a second iteration, after which $q \leq j - 1 < j$.

We can now use the properties derived to show that Algorithm 8.3 is correct. Firstly recall that we set *split* to be the value of q returned. The fact that $i \leq q \leq j - 1 < j$ means that $A[i..q]$ and $A[q + 1..j]$ are non-empty. Thus each call is on a portion of the array that is smaller than the input one and so the algorithm halts; otherwise there is the danger of an infinite recursion which would result if one portion was always empty. We must also show that the keys in the two portions are appropriate, i.e., at the time of call $A[r] \leq pivot$ for $i \leq r \leq q$ and $A[r] \geq pivot$ for $q + 1 \leq r \leq j$. The second inequality follows immediately from above (since we have $A[r].key \geq pivot$ for $q < r \leq j$). To derive the first one observe first that when partition(A, i, j) halts we have either $p = q$ or $p = q + 1$. From the preceding paragraph we know that $A[r].key \leq pivot$ for $i \leq r < p$. So if $p = q + 1$ we are done. Otherwise $p = q$ and we just need to verify that $A[p] \leq pivot$. For this we note that the loop on line 5 terminates when the test $A[q].key > pivot$ fails (for the current value of q). So the last time it was executed we had $A[q].key \not> pivot$, i.e., $A[q].key \leq pivot$. Since this happens for the final value of q and $p = q$ we are done.

Let us analyse the running time. Let $n = j - i + 1$. We observe that during the execution of partition, we always have $q \geq p - 1$, because all keys below p are always smaller than or equal to $pivot$ and all keys above q are always larger than or equal to $pivot$. This implies that the running time (best-case, average-case and

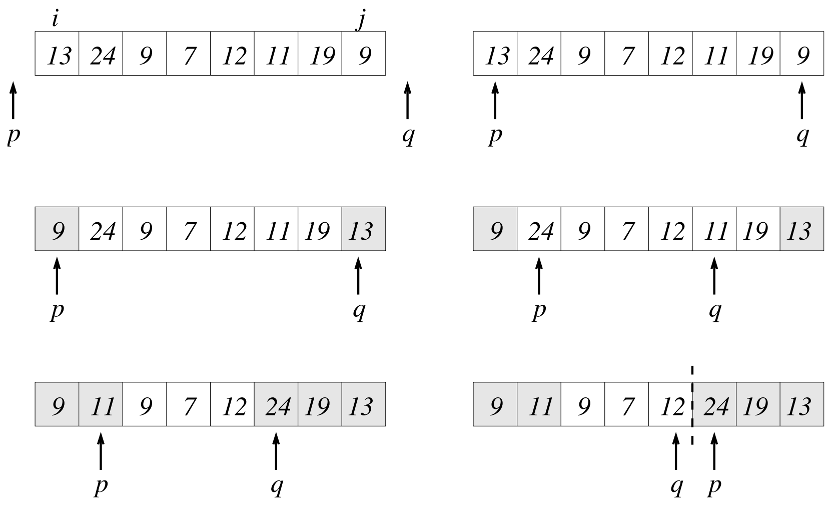

Figure 8.5. The operation of partition

worst-case) of partition is $\Theta(n)$.

Unfortunately, this does not give us a simple recurrence for the running time of quickSort, because we do not know where the array will be split. All we get is the following:

$$T_{\text{quickSort}}(n) = \max_{1\le s\le n-1}\big(T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n-s)\big) + \Theta(n).$$

It can be shown that this implies that the worst-case running-time $T_{\text{quickSort}}(n)$ of quickSort satisfies $T_{\text{quickSort}}(n) = \Theta(n^2)$.

We discuss the intuition behind this. The worst-case (intuitively speaking) for quickSort is that the array is always partitioned into subarrays of sizes 1 and $n-1$, because in this case we get a recurrence

$$T(n) = T(n-1) + T(1) + \Theta(n) = T(n-1) + \Theta(n),$$

which implies $T(n) = \Theta(n^2)$. Of course we need to also justify our assumption that this combination of partitions could happen—in fact, one example of a case which causes this bad behaviour is the case when the input array A is initially sorted, which is not uncommon for some applications.

The best case arises when the array is always split in the middle. Then we get the same recurrence as for mergeSort,

$$T(n) = T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor) + \Theta(n),$$

which implies $T(n) = \Theta(n\lg(n))$. Fortunately, the typical or "average" case is much closer to the best case than to the worst case. A mathematically complicated analysis shows that for random arrays with each permutation of the keys being equally likely, the average running time of quickSort is $\Theta(n\lg n)$. Essentially good splits are much more likely than bad ones so on average the runtime is dominated by them.

Nevertheless, the poor running time on sorted arrays (and similarly on nearly sorted arrays) is a problem. It can be avoided, though, by choosing the pivot differently. For example, if we take the key of the middle item $A[\lfloor(i+j)/2\rfloor]$ as the pivot, the running time on sorted arrays becomes $\Theta(n\lg n)$, because they will always be partitioned evenly. But there are other worst case arrays with a $\Theta(n^2)$ running time for this choice of the pivot. A strategy that avoids this worst-case behaviour altogether, with high probability, is to choose the pivot randomly (this is a topic for a more advanced course). This strategy is an example of a *randomised algorithm*, an approach that has gained increasing importance. Note that the runtime of this algorithm can vary between different runs on the same input. This approach for quicksort guarantees that if we run the algorithm over a large number of times on inputs of size n then the runtime will be $O(n\lg n)$ no matter what inputs are supplied. The proof of this claim is beyond the scope of this course, it can be found in [CLRS].

We conclude with an observation that quickSort is an in-place sorting algorithm. However, it is not too difficult to come up with examples to show it is not stable.

8.3 Further Reading

If you have [GT], look for heapSort in the "Priority Queues" chapter, and quicksort in the "Sorting, Sets and Selection" chapter. [CLRS] has an entire chapter titled "Heapsort" and another chapter on "Quicksort" (ignore the advanced material).

Exercises

1. Suppose that the array

$$A = \langle 5, 0, 3, 11, 9, 8, 4, 6\rangle.$$

   is sorted by heapSort. Show the status of the heap after buildHeap(A) and after each iteration of the loop. (Represent the heap as a tree.)

2. Consider the enhanced version printQuickSort of quickSort displayed as Algorithm 8.6. Line 1 simply prints the keys of $A[i],\ldots,A[j]$ on a separate line of the standard output. Let $A = \langle 5, 0, 3, 11, 9, 8, 4, 6\rangle$.

   What does printQuickSort($A, 0, 7$) print?

3. Give an example to show that quickSort is not stable.

```
**Algorithm** printQuickSort$(A, i, j)$

1. **print** $A[i \dots j]$
2. **if** $i < j$ **then**
3.         $split \leftarrow$ partition$(A, i, j)$
4.         quickSort$(A, i, split)$
5.         quickSort$(A, split + 1, j)$
```

Algorithm 8.6

Graphs and BFS

We will devote two lectures of the Algorithms and Data Structures thread to an introduction to graph algorithms. As with many other topics we could spend the entire course on this area.

9.1 Directed and Undirected Graphs

A *graph* is a mathematical structure consisting of a set of *vertices* and a set of *edges* connecting the vertices. Formally, we view the edges as pairs of vertices; an edge (v, w) connects vertex v with vertex w. Formally:

- V is a set, and
- $E \subseteq V \times V$.

We write $G = (V, E)$ to denote that G is a graph with vertex set V and edge set E. A graph $G = (V, E)$ is *undirected* if for all vertices $v, w \in V$, we have $(v, w) \in E$ if and only if $(w, v) \in E$, that is, if all edges go both ways. This definition makes it clear that undirected graphs are just special directed graphs. When studying undirected graphs we often represent a complementary pair of directed edges (u, v) and (v, u) as just one 'undirected' edge using some special notation. In this introduction we will not go to this extent but state in each case if a graph is directed or undirected. A useful convention is that in drawing diagrams of directed graphs we indicate the direction of an edge with an arrow. In the case of undirected graphs we do not draw pairs of directed edges (one from u to v and one from v to u) but just one edge without an arrow[1]. If we want to emphasise that the edges have a direction, we say that a graph is *directed*.

Note that in principle V and hence E can be infinite sets. Such graphs are very useful in many areas (including computing) but for this introduction we will assume always that V is finite. Since $E \subseteq V \times V$ is follows that E is also finite.

Example 9.1. Figure 9.2 shows a drawing of the (directed) graph $G = (V, E)$ with vertex and edge sets given by:

$$V = \{0, 1, 2, 3, 4, 5, 6\}$$
$$E = \{(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)\}.$$

We state here a definition that will be needed subsequently.

Definition 9.3. Let $v \in V$ be a vertex in a directed graph $G = (V, E)$.

- The *in-degree* $\text{in}(v)$ of v is the number of incoming edges to v, i.e., the number of edges of form (u, v). The set of in-edges to u is written as $\text{In}(v)$.
- The *out-degree* $\text{out}(v)$ of v is the number of outgoing edges from v, i.e., the number of edges of form (v, u). The set of out-edges from v is written as $\text{Out}(v)$.

[1]Do not confuse a diagram representing a graph with the graph itself. The graph is the abstract mathematical structure and can be represented by many different diagrams.

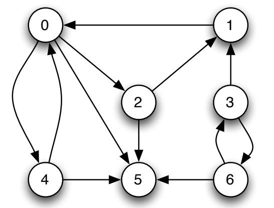

Figure 9.2. A directed graph

In an undirected graph, the *degree* of v is the number of edges $e \in E$ for which one endpoint is v. Two vertices are *adjacent* if they are joined by an edge. You should also know definitions of graph theory concepts such as *paths*, *cycles*, *connectedness* and *connected components* from your *Maths-for-Informatics* courses. You will find them in any textbook on graph theory or discrete mathematics and also in most books on algorithms. Graphs are a useful mathematical model for numerous problems and structures. We give just a few examples.

Example 9.4. *Airline route maps.*
Vertices represent airports, and there is an edge from vertex A to vertex B if there is a direct flight from the airport represented by A to the airport represented by B.

Example 9.5. *Electrical Circuits.*
Vertices represent diodes, transistors, capacitors, switches, etc., and edges represent wires connecting them.

Example 9.6. *Computer Networks.*
Vertices represent computers and edges represent network connections (e.g., cables) between them.

Example 9.7. *The World Wide Web.*
Vertices represent webpages, and edges represent hyperlinks.

Example 9.8. *Flowcharts.*
A flowchart illustrates the flow of control in a procedure. Essentially, a flowchart consists of boxes (vertices) containing statements of the procedure and arrows (directed edges) connecting the boxes to describe the flow of control.

Example 9.9. *Molecules.*
Vertices are atoms, edges are bonds between them.

The graphs in Examples 9.4, 9.7 and 9.8 are directed. The graphs in Examples 9.5, 9.6 and 9.9 are undirected.

9.2 Data structures for graphs

Let $G = (V, E)$ be a graph with n vertices. We assume that the vertices of G are numbered $0, \dots, n - 1$ in some arbitrary manner.

The adjacency matrix data structure

The *adjacency matrix* of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge from vertex number } i \\ & \text{to vertex number } j; \\ 0, & \text{otherwise.} \end{cases}$$

For example, the adjacency matrix for the graph in Figure 9.2 is

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Note that the adjacency matrix depends on the particular numbering of the vertices.

The adjacency matrix data structure stores the adjacency matrix of the graph as a 2-dimensional Boolean array, where TRUE represents 1 (i.e., there is an edge) and FALSE represents 0 (i.e., there is no edge). It is worth noting here that for some applications it is preferable to use actual numbers rather than Boolean values. The main advantage of the adjacency matrix representation is that for all vertices v and w we can check in constant time whether or not there is an edge from vertex v to vertex w.

However we pay a price for this fast access. Let m be the number of edges of the graph. Note that m can be at most n^2. If m is close to n^2, we call the graph *dense*, and if m is much smaller than n^2 we call it *sparse*. Storing a graph with n vertices and m edges will usually require space at least $\Omega(n+m)$. However, the adjacency matrix uses space $\Theta(n^2)$, and this is much more than $\Theta(n+m)$ for sparse graphs (when a large fraction of entries in the adjacency matrix consists of zero). Moreover, many algorithms have to inspect all edges of the graph at least once, and to do this for a graph given in adjacency matrix representation, such an algorithm will have to inspect every matrix entry at least once to make sure that it has seen all edges. Thus it will require time $\Omega(n^2)$; we will see some important algorithms that run in time $\Theta(n+m)$ with an appropriate data structure.

The adjacency list data structure

The *adjacency list* representation of a graph G with n vertices consists of an array *vertices* with n entries, one for each vertex. The entry for vertex v is a list of all vertices w such there is an edge from v to w. We make no assumptions on the order in which the vertices adjacent to a vertex v appear in the adjacency list, and our algorithms should work for any order.

Figure 9.10 shows an adjacency list representation of the graph in Figure 9.2.

For sparse graphs an adjacency list is more space efficient than an adjacency matrix. For a graph with n vertices and m edges it requires space $\Theta(n+m)$, which might be much less than $\Theta(n^2)$. Moreover, if a graph is given in adjacency list

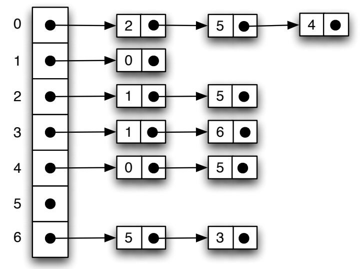

Figure 9.10. Adjacency list representation of the graph in Figure 9.2

representation one can visit efficiently all neighbours of a vertex v; this just requires time

$$\Theta\big(1 + \text{out}(v)\big)$$

and not time $\Theta(n)$ as for the adjacency matrix representation[2]. Therefore, visiting all edges of the graph only requires time $\Theta(n+m)$ and not time $\Theta(n^2)$ as for the adjacency matrix representation. On the other hand, finding out whether there is an edge from vertex v to vertex w requires (in the worst case) stepping through the whole adjacency list of v, which could have up to n entries. Thus a *simple adjacency test* takes time $\Theta(n)$ in the worst case, compared to $\Theta(1)$ for adjacency matrices.

Extensions

We have only described the basic data structures representing graphs. Vertices are just represented by the numbers they get in some numbering, and edges by the numbers of their endpoints. Often, we want to store additional information. For example, in Example 9.4 we might want to store the names of the airports represented by the vertices, or in Example 9.7 the URLs of the webpages. To do this, we create separate vertex objects that store the number of a vertex and an object that contains the additional data we want to store at the vertex. In the adjacency list representation, we include the adjacency list of a vertex in the vertex object. Then the graph is represented by an array (possibly a dynamic array) of vertex objects.

Similarly, we might want to store additional information on the edges of a graph; in Example 9.4 we may want to store flight numbers. We can do this by setting up separate edge objects which will store references to the two endpoints of an edge and the additional information. Then in the adjacency list representations, the lists would be lists of such edge objects.

A frequent situation is that edges of a graph carry *weights*, which are real numbers providing information such as the cost of a flight in Example 9.4 or the capacity of a wire or network connection in Examples 9.5 and 9.6. Graphs whose edges carry weights are called *weighted graphs*.

[2]It might be tempting to replace $\Theta\big(1 + \text{out}(v)\big)$ with $\Theta\big(\text{out}(v)\big)$ but this is wrong if $\text{out}(v) = 0$.

9.3 Traversing Graphs

Most algorithms for solving problems on graphs examine or process each vertex and each edge of the graph in some particular order. The skeleton of such an algorithm will be a *traversal* of the graph, that is, a strategy for visiting the vertices and edges in a suitable order.

Breadth-first search (BFS) and depth-first search (DFS) are two traversals that are particularly useful. Both start at some vertex v and then visit all vertices reachable from v (that is, all vertices w such that there is a path from v to w). If there are vertices that remain unvisited, that is, if there are vertices that are not reachable from v, then the only way they can be listed is if the search chooses a new unvisited vertex v' and visits all vertices reachable from v'. This process would have to be repeated until all vertices are visited.

We can present the general graph searching strategy as algorithms 9.11 and 9.12. In these algorithms we assume that we start with every vertex unmarked and we

```
Algorithm search($G$)

1. initialise schedule $S$
2. for all $v \in V$ do
3.     if $v$ is not marked then
4.         searchFromVertex($G, v$)
```

Algorithm 9.11

```
Algorithm searchFromVertex($G, v$)

1. mark $v$
2. put $v$ onto schedule $S$
3. while schedule $S$ is not empty do
4.     remove a vertex $v$ from $S$
5.     for all $w$ adjacent to $v$ do
6.         if $w$ is not marked then
7.             mark $w$
8.             put $w$ onto schedule $S$
```

Algorithm 9.12

have some efficient way of marking it. Note that once a vertex is marked it stays as such. The schedule S is some (efficient) data structure such that we can put vertices on it and take them off one at a time. For BFS we use a *Queue* while for DFS we use a *Stack* as our schedule S. As we will see, because recursive procedures use an inherent stack we will be able to re-express DFS rather neatly without an explicit schedule S (all the same it is there but given to us by the underlying system for handling recursion).

Breadth-First Search

A BFS starting at a vertex v first visits v, then it visits all neighbours of v (i.e., all vertices w such that there is an edge from v to w), then all neighbours of the neighbours that have not been visited before, then all neighbours of the neighbours of the neighbours that have not been visited before, etc. For example, one BFS of the graph in Figure 9.2 starting at vertex 0 would visit the vertices in the following order:

$$0, 2, 5, 4, 1$$

It first visits 0, then the neighbours $2, 5, 4$ of 0. Next are the neighbours of 2, which are 1 and 5. Since 5 has been visited before, only 1 is added to the list. All neighbours of 5, 4, and 1 have already been visited, so we have found all vertices that are reachable from 0. Note that there are other orders in which a BFS starting at 0 could visit the vertices of the graph, because the neighbours of 0 might be visited in a different order. If the neighbour vertices are visited in numerical order, then the BFS from 0 would be $0, 2, 4, 5, 1$. Vertices 3 and 6 are not reachable from 0, so to visit them we must to start another BFS, say at 3.

The traversal heavily depends on the vertex we start at. If we start a BFS at vertex 6, for example, all vertices are reachable, and the vertices are visited in one sweep, for example:

$$6, 5, 3, 1, 0, 2, 4.$$

Other possible orders are $6, 3, 5, 1, 0, 2, 4$ and $6, 5, 3, 1, 0, 4, 2$ and $6, 3, 5, 1, 0, 4, 2$.

In an *undirected graph*, however, the number of different BFS searches (or DFS searches) that need to be made to visit all vertices is independent of the choice of start vertices.

During a BFS we have to store vertices that have been visited so far and also the vertices that have been completely processed (all their neighbours have been visited). We maintain a Boolean array $visited$ with one entry for each vertex, which is set to TRUE when the vertex is visited. The vertices that have been visited, but have not been completely processed, are stored in a *Queue*. This guarantees that vertices are visited in the right order—vertices that are discovered first will be processed first. Algorithms 9.13 and 9.14 show a BFS implementation in pseudocode. The main algorithm bfs first initialises the $visited$ array and the queue and then loops through all vertices, starting a bfsFromVertex for all vertices that have not been marked 'visited' in previous invocations of bfsFromVertex. The subroutine bfsFromVertex visits all vertices reachable from the start vertex in the way described above. The inner loop in lines 5–8 can be implemented using an iterator over the adjacency list of the vertex v.

To see the progress of bfs(G) we can put a **print** v statement after each $visited[v] =$ TRUE. In practice, BFS (or DFS) is often applied to applications such as all-pairs shortest paths for a graph. For applications like this, the "current vertex" obtained by BFS is used in the top-level algorithm for the particular application.

Algorithm bfs(G)

```
1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. Initialise Queue $Q$
3. for all $v \in V$ do
4.     if $visited[v] = $ FALSE then
5.         bfsFromVertex($G, v$)
```

Algorithm 9.13

Algorithm bfsFromVertex(G, v)

```
1. $visited[v] = $ TRUE
2. $Q$.enqueue($v$)
3. while not $Q$.isEmpty() do
4.     $v \leftarrow Q$.dequeue()
5.     for all $w$ adjacent to $v$ do
6.         if $visited[w] = $ FALSE then
7.             $visited[w] = $ TRUE
8.             $Q$.enqueue($w$)
```

Algorithm 9.14

Exercises

1. Give an adjacency matrix and an adjacency list representation for the graph displayed in Figure 9.15. Give orders in which a BFS starting at vertex n may

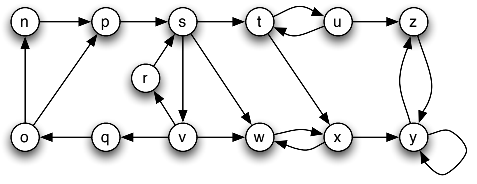

Figure 9.15.

traverse the graph.

Graphs and DFS

In this note, we turn to an alternative to BFS, which is Depth-First Search (DFS).

10.1 Depth-first search

A DFS starting at a vertex v first visits v, then some neighbour w of v, then some neighbour x of w that has not been visited before, etc. When it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before. It continues with this neighbour until it has to backtrack again. Eventually, it will visit all vertices reachable from v. Then a new DFS is started at some vertex that is not reachable from v, until all vertices have been visited.

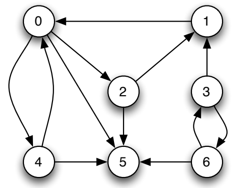

Figure 10.1. A directed graph

Reconsider the graph of Lecture Note 13, shown above in Figure 10.1. A DFS on the graph of Figure 10.1 starting at 0 might visit the vertices in the order

$$0, 2, 1, 5, 4.$$

After it has visited $0, 2, 1$ the DFS backtracks to 2, visits 5, then backtracks to 0, and visits 4. A DFS starting at 0 might also visit the vertices in the order $0, 4, 5, 2, 1$ or $0, 5, 4, 2, 1$ or $0, 2, 5, 1, 4$. As for BFS, this depends on the order in which the neighbours of a vertex are processed.

DFS can be implemented in a similar way as BFS using a *Stack* instead of a *Queue*, see Algorithms 10.2 and 10.3.

10.2 A recursive implementation of DFS

A second way to implementing DFS is via a recursive algorithm (essentially we rely on the stack used to implement recursion rather than maintaining our own). To visit all vertices reachable from the start vertex v, we visit v, then the first neighbour of v and all vertices reachable from this first neighbour—remember that the search is *depth-first*. Then it visits the next neighbour and all new

```
Algorithm dfs($G$)
1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. Initialise Stack $S$
3. for all $v \in V$ do
4.     if not $visited[v]$ then
5.         dfsFromVertex($G, v$)
```

Algorithm 10.2

```
Algorithm dfsFromVertex($G, v$)
1. $S$.push($v$)
2. while not $S$.isEmpty() do
3.     $v \leftarrow S$.pop()
4.     if not $visited[v]$ then
5.         $visited[v] = $ TRUE
6.         for all $w$ adjacent to $v$ do
7.             $S$.push($w$)
```

Algorithm 10.3

```
Algorithm dfs($G$)
1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. for all $v \in V$ do
3.     if not $visited[v]$ then
4.         dfsFromVertex($G, v$)
```

Algorithm 10.4

```
Algorithm dfsFromVertex($G, v$)
1. $visited[v] \leftarrow$ TRUE
2. for all $w$ adjacent to $v$ do
3.     if not $visited[w]$ then
4.         dfsFromVertex($G, w$)
```

Algorithm 10.5

vertices reachable from it, etc. Algorithms 10.4 and 10.5 give pseudocode for a recursive implementation of DFS.

The analysis of algorithm dfs might seem complicated because of the recursive calls inside the loops. Rather than writing down a recurrence for dfs, let us analyze the running-time directly. Let n be the number of vertices of the input graph G and let $m = |E|$. Then dfs(G) requires time $\Theta(n)$ for initialisation. Moreover, dfsFromVertex(G, v) requires time $\Theta(1 + \text{out-degree}(v))$, because the loop is iterated out-degree(v) times (as usual we add 1 in case out-degree$(v) = 0$)

Now the crucial observation is that dfsFromVertex(G, v) is invoked exactly once for every vertex v. To see that it is invoked *at least* once, note that $visited[v]$ is set to TRUE only if dfsFromVertex(G, v) is invoked. So if the method was never invoked, then $visited[v]$ would remain FALSE. But this cannot happen, because for all v with $visited[v] = \text{FALSE}$, dfsFromVertex$(G, v)$ is invoked in Line 4 of dfs(G) (in the v-execution of the loop). To see that dfsFromVertex(G, v) is invoked *at most* once, note that it is only invoked if $visited[v] = \text{FALSE}$. However, after its first execution $visited[v]$ is TRUE and can never become FALSE again. So dfsFromVertex(G, v) is invoked exactly once for every vertex v.

Therefore, we get the following expression for the running time of dfs(G):

$$\begin{aligned}\Theta(n) + \sum_{v\in V}\Theta(1 + \text{out-degree}(v)) &= \Theta\Big(n + \sum_{v\in V}(1 + \text{out-degree}(v))\Big)\\ &= \Theta\Big(n + n + \sum_{v\in V}\text{out-degree}(v)\Big)\\ &= \Theta\Big(n + \sum_{v\in V}\text{out-degree}(v)\Big)\end{aligned}$$

Let m be the number of edges of G. Then $\sum_{v\in V}\text{out-degree}(v) = m$, and we get

$$T_{\mathsf{dfs}}(n, m) = \Theta(n + m).$$

We count an undirected edge as two edges, one in each direction. We then get $\sum_{v\in V}\text{degree}(v) = 2m$ for undirected graphs, but since $2m = \Theta(m)$, this makes no difference.

10.3 DFS Forests

A DFS starting at some vertex v explores the graph by building up a *tree* that contains all vertices that are reachable from v and all edges that are used to reach these vertices. We call this tree a *DFS tree*. A complete DFS exploring the full graph (and not only the part reachable from a given vertex v) builds up a collection of trees, or forest, called a *DFS forest*.

Suppose that we explore the graph in Figure 10.1 by a DFS starting at vertex 0 that visits the vertices in the following order: $0, 2, 1, 5, 4, 3, 6$. The corresponding DFS forest is shown in Figure 10.6.

Note that a DFS forest is not unique. Figure 10.7 shows another DFS forest for the graph in Figure 10.1.

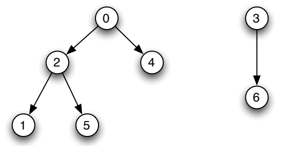

Figure 10.6.

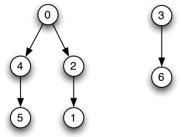

Figure 10.7.

10.4 Connected components

As a first application of DFS, we compute the connected components of an undirected graph $G = (V, E)$. Recall that we say that a subset C of V is *connected* if for all $v, w \in C$ there is a path from v to w (if $v = w$ there is always a "path" of length 0 from v to w.) A *connected component* of an undirected graph G is a *maximal* connected subset C of V. Here "maximal connected subset" means that there is no connected subset C' of V that strictly contains C. An undirected graph is *connected* if it has just one connected component (for all vertices v, w there is a path from v to w).

Our algorithm is based on the following two simple observations, which do not necessarily hold for directed graphs:

(1) Each vertex of an undirected graph is contained in exactly one connected component.

(2) For each vertex v of an undirected graph, the connected component that contains v is precisely the set of all vertices that are reachable from v.

Hence dfsFromVertex(G, v) visits the set of vertices in the connected component of v.

We modify dfs as follows: We add a statement **print** v after line 1 of dfsFromVertex(G, v). After this modification, dfsFromVertex(G, v) prints the vertices in the connected component of v. Then we add a statement **print** "New Component" before each call of dfsFromVertex(G, v) in line 4 of dfs. If the set of connected components is needed as input for another algorithm, we can instead get dfs to return the connected components in some convenient format.

The asymptotic running time of this algorithm for computing the connected components is clearly the same as the running time of dfs.

Components of directed graphs

It is not so clear what connectivity means in directed graphs. For example, is the graph in Figure 10.1 connected? It looks "joined up", but then vertex 6 is not reachable from vertex 0. We say that a directed graph G is *weakly connected* if the undirected graph we obtain from G by disregarding the direction of the edges is connected. A directed graph is *strongly connected* if for all vertices v, w there is a path from v to w and a path from w to v. Strong connectivity is a more important notion. The graph in Figure 10.1 is weakly connected, but not strongly connected; for example, there is no path from 0 to 6.

Derived from the notions of weak and strong connectivity, we have *weakly connected components* and *strongly connected components*. For example, the digraph (i.e., directed graph) in Figure 10.1 only has one weakly connected component (containing all vertices), and it has three strongly connected components:

$$0, 1, 2, 4$$
$$3, 6$$
$$5.$$

Computing the weakly connected components of a directed graph is easy: writing an algorithm that does this is a good exercise. Computing the strongly connected components is much harder. It can also be done by an algorithm based on DFS, but this is outside the scope of this course.

10.5 Classifying vertices during a DFS

Let G be a graph. Recall that during an execution of dfs(G), the subroutine dfsFromVertex(G, v) is invoked exactly once for each vertex v. Let us call vertex v *finished* after dfsFromVertex(G, v) is completed. During the execution of dfs(G), a vertex can be in three states:

- not yet visited (let us call a vertex in this state *white*),
- visited, but not yet finished (*grey*).
- finished (*black*).

We can modify our recursive DFS algorithm so that it keeps track of the states of the vertices, see Algorithms 10.8 and 10.9.

The following property will be important in the next section:

Lemma 10.10. *Let $G = (V, E)$ be a graph, $v \in V$. Consider the time during execution of dfs(G) when* dfsFromVertex(G, v) *is called. Then for all vertices w we have:*

(1) If w is white and reachable from v, then w will be black before v.

(2) If w is grey, then v is reachable from w.

We will not give a formal proof here. Intuitively, (1) follows from the fact that

```
Algorithm dfs($G$)
1.  Initialise array $state$ by setting all entries to $white$.
2.  for all $v \in V$ do
3.      if $state[v] = white$ then
4.          dfsFromVertex($G, v$)
```

Algorithm 10.8

```
Algorithm dfsFromVertex($G, v$)
1.  $state[v] \leftarrow grey$
2.  for all $w$ adjacent to $v$ do
3.      if $state[w] = white$ then
4.          dfsFromVertex($G, w$)
5.  $state[v] \leftarrow black$
```

Algorithm 10.9

all vertices w that are reachable from v are either black before dfsFromVertex(G, v) is started (and thus black before v) or they will be visited during the execution of dfsFromVertex(G, v), because a DFS starting at v visits all vertices reachable from v that have not been visited earlier. Thus they will become black while v is still grey. (2) follows from the fact if w is still grey, dfsFromVertex(G, w) is not yet completed. However, a DFS starting at w only visits vertices reachable from w. Thus v must be reachable from w.

10.6 Topological Sorting

Suppose we have a list of tasks to do, some of which depend on others to be completed first. For example, a practical exercise may involve 10 tasks, numbered 0–9.

- Task 0 must be completed before Task 1 is started.
- Task 1 and Task 2 must be completed before Task 3 is started.
- Task 4 must be completed before Task 0 or Task 2 are started.
- Task 5 must be completed before Task 0 or Task 4 are started.
- Task 6 must be completed before Task 4, Task 5 or Task 7 are started.
- Task 7 must be completed before Task 0 or Task 9 are started.
- Task 8 must be completed before Task 7 or Task 9 are started.

- Task 9 must be completed before Task 2 or Task 3 are started.

We can arrange this information in a more digestible form as a *dependency graph*. The vertices of this directed graph are the tasks to be performed, and there is an edge from task v to task w if v must be completed before w can be started. Figure 10.11 shows the dependency graph of our example.

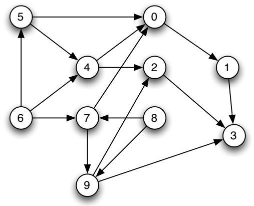

Figure 10.11.

Before carrying out the tasks we must arrange them in an order that respects all dependencies. This is called a *topological order* of the dependency graph.

Definition 10.12. Let $G = (V, E)$ be a directed graph. A *topological order* of G is a total order $\prec$ of the vertex set V such that for all edges $(v, w) \in E$ we have $v \prec w$.

For example, one topological order of the digraph in Figure 10.11 is

$$8 \prec 6 \prec 7 \prec 9 \prec 5 \prec 4 \prec 2 \prec 0 \prec 1 \prec 3.$$

It is not obvious how to find a topological order of a digraph efficiently (assuming it has one). In fact, there are many digraphs that do not have a topological order: If there are vertices v, w such that there is both an edge from v to w and an edge from w to v, then no topological order of the graph exists (we cannot take either $v \prec w$ nor $w \prec v$, so any ordering is not *total*). In general, if the graph has a directed cycle then there is no topological order.

A directed graph that does not have a cycle is called a *directed acyclic graph* (*DAG*). The following theorem holds for DAGs.

Theorem 10.13. *A directed graph has a topological order if, and only if, it is a DAG.*

Our algorithm for finding the topological order of a DAG is based on dfs. Consider the execution of dfs(G) on a directed graph G. We define the order $\prec$ on V by setting $v \prec w$ if w becomes black before v (later finishers are smaller).

We now show that if G is a DAG, then $\prec$ is a topological order of G. Assume that $G = (V, E)$ is a DAG. Let $(v, w) \in E$. We have to prove that $v \prec w$, i.e., that w becomes black before v. Consider the moment in the DFS when dfsFromVertex(G, v) is called.

- If w is already black at this moment, there is nothing to prove.
- If w is white, then by Lemma 10.10(1), w will be black before v.

- If w is grey, then by Lemma 10.10(2) v is reachable from w. Thus there is a path from w to v, and together with the edge (v, w), this path forms a cycle. But we assumed that G is acyclic, so this cannot happen.

Note that our argument gives us some additional information: If we find, during the execution of dfsFromVertex(G, v) for some vertex v, an edge from v to a grey vertex w, then we know that G contains a cycle.

We can modify our basic DFS algorithm in order to get an algorithm for computing the order $\prec$ and printing the vertices in this order. If G is not a DAG, our algorithm will simply print "G has a cycle". Otherwise the algorithm adds all vertices to the front of a linked list when they become black. Vertices becoming black earlier appear later in the list, which means that the list is in order $\prec$. If during the execution of sortFromVertex(G, v) for some vertex v, an edge from v to a grey vertex w is found, then there must be a cycle, and the algorithm stops.

Algorithm topSort(G)

```
1. Initialise array $state$ by setting all entries to $white$.
2. Initialise linked list $L$
3. for all $v \in V$ do
4.     if $state[v] = white$ then
5.         sortFromVertex$(G, v)$
6. print all vertices in $L$ in the order in which they appear
```

Algorithm 10.14

Algorithm sortFromVertex(G, v)

```
1. $state[v] \leftarrow grey$
2. for all $w$ adjacent to $v$ do
3.     if $state[w] = white$ then
4.         sortFromVertex$(G, w)$
5.     else if $state[w] = grey$ then
6.         print "$G$ has a cycle"
7.         halt
8. $state[v] \leftarrow black$
9. $L$.insertFirst$(v)$
```

Algorithm 10.15

The running time of topSort is the same as that of dfs, $\Theta(n + m)$.

Exercises

1. Give orders in which a BFS and DFS starting at vertex n may traverse the graph displayed in Figure 10.16.

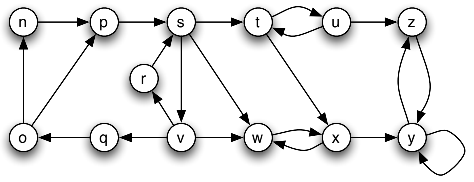

Figure 10.16.

2. Give 3 different DFS forests for the graph in Figure 10.16.
3. Let G be a graph with n vertices and m edges. Explain why $O(\lg m)$ is $O(\lg n)$.
4. The *reflexive transitive closure* of a directed graph $G = (V, E)$ is the graph G^* with the same vertex set V as G and an edge from vertex v to vertex w if there is a path (possibly of length 0) from v to w.

   Describe an algorithm that computes the reflexive transitive closure G^* of a graph G in time $O(n(n + m))$, where n is the number of vertices and m the number of edges of G. Represent the output G^* in adjacency matrix representation.

Indexing and Sorting for the WWW

We will devote a couple of lectures of the ADS thread to Algorithms for the WWW. The material is more applied than most other topics though all are concerned with appropriate efficient algorithms to various problems.

11.1 Indexing

An *inverted index* to a set of documents (or webpages on the web) is conceptually similar to the index of an ordinary textbook. The idea is that we have a set of *terms* that appear in the text of the documents. Sometimes every individual word appearing in the documents is a term; possibly with some pruning to account for upper case versus lower case letters (*case folding*) and linguistic features[1]. At other times we have a restricted set of given terms. There are other types of index, but this is the most common and is therefore the one we concentrate on here.

An index can be in many forms, but the standard form is that for every term t, there is an associated list of document identifiers $d_1, \ldots, d_{n_t}$, where n_t is the number of documents in which t appears. Therefore once we have the index we can search our set of documents very quickly by finding the term t in our index (probably using binary search). In Figure 11.1, we show a "set of documents" each comprising a single line of a particular nursery rhyme. In Figure 11.2 we show two inverted indexes for this "set of documents." The figures are taken from Chapter 3 of the *Managing Gigabytes* book mentioned at the end of these notes.

Document	Text
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

Figure 11.1. A children's rhyme, each line being treated as a document .

Each of the two indexes of Figure 11.2 consist of a *lexicon* (the set of *terms*), and for every term in the lexicon, an inverted file entry for that term. An inverted file entry is the frequency of a term followed by a list of locations where that term appears in the document set; for example, in the left index, the inverted file for the term 'hot' is $\langle 2; 1, 4\rangle$. Note that both of the Indexes has a frequency of 2 (appears in two documents) for every term in the lexicon—this is just a coincidence for this particular example.

[1]If we are indexing documents in English it makes sense to regard *words* and *wording* as occurrences of the same stem: *word*. This is referred to as *stemming*.

Number	Term	Documents
1	cold	$\langle 2; 1, 4\rangle$
2	days	$\langle 2; 3, 6\rangle$
3	hot	$\langle 2; 1, 4\rangle$
4	in	$\langle 2; 2, 5\rangle$
5	it	$\langle 2; 4, 5\rangle$
6	like	$\langle 2; 4, 5\rangle$
7	nine	$\langle 2; 3, 6\rangle$
8	old	$\langle 2; 3, 6\rangle$
9	pease	$\langle 2; 1, 2\rangle$
10	porridge	$\langle 2; 1, 2\rangle$
11	pot	$\langle 2; 2, 5\rangle$
12	some	$\langle 2; 4, 5\rangle$
13	the	$\langle 2; 2, 5\rangle$

Number	Term	Documents;Words
1	cold	$\langle 2; (1; 6), (4; 8)\rangle$
2	days	$\langle 2; (3; 2), (6; 2)\rangle$
3	hot	$\langle 2; (1; 3), (4; 4)\rangle$
4	in	$\langle 2; (2; 3), (5; 4)\rangle$
5	it	$\langle 2; (4; 3, 7), (5; 3)\rangle$
6	like	$\langle 2; (4; 2, 6), (5; 2)\rangle$
7	nine	$\langle 2; (3; 1), (6; 1)\rangle$
8	old	$\langle 2; (3; 3), (6; 3)\rangle$
9	pease	$\langle 2; (1; 1, 4), (2; 1)\rangle$
10	porridge	$\langle 2; (1; 2, 5), (2; 2)\rangle$
11	pot	$\langle 2; (2; 5), (5; 6)\rangle$
12	some	$\langle 2; (4; 1, 5), (5; 1)\rangle$
13	the	$\langle 2; (2; 4), (5; 5)\rangle$

Figure 11.2. Two indexes for the "set of documents" of Figure 11.1.
The left index gives frequency and list of document numbers.
The right index gives frequency, document numbers and occurrence within the relevant document.

The *granularity* of an *inverted index* is the detail to which we record the location of a term in the document. In the index on the left of Figure 11.2, the granularity is document-level. In the index on the right of that Figure, the granularity is word-level.

The terms of a lexicon have an associated ordering, given by the ordering of their term-numbers. For the indexes in Figure 11.2 the ordering is alphabetical, however this is not generally the case (and the mapping of terms to term numbers will need to be stored).

11.2 Querying using an Index

An Inverted Index enables fast search on a set of documents, when queries are formulated using the terms of the lexicon. The quality of the query replies will partly depend on the granularity of the *Inverted Index*: for example, if we perform the search "pease" on the left Index of Figure 11.2, then we are informed of the identifiers of the two documents which contain "pease", but we do not know where the term is located in the document (obviously not a big problem with the 'documents' of Figure 11.1). We would need to check the document itself to look for the position of that term. However, with the finer right-hand Index in Figure 11.2, we will directly recover the positions of the terms in the documents.

The queries we might need to make are usually more complicated than single-term queries. However, assuming that our Inverted Index stores the inverted file entry for every term t in order of document index, then we could use a *merging* technique to answer simple boolean queries such as "pease" AND "hot". Solutions to boolean queries can be generated in time linear in the length of lists for the two terms, by performing a synchronised scan of the two lists in a manner similar to the merge routine of mergeSort. The operation performed by this scan will be

$\cap$ (intersection) in the case of AND and $\cup$ (union) in the case of OR, and can be applied inductively, still in linear time.

In practice, when our documents are webpages, we will want to rank the pages, rather than treat them all as equal. We will discuss this issue (in the context of the PageRank algorithm of Google) in a subsequent lecture.

11.3 Constructing a Large-Scale Index

Indexing is the act of preprocessing a set of documents to construct an inverted index for those documents. It is this process that we are concerned with in this lecture. We are concerned with "indexing in the large" where the set of documents to be indexed might take up many gigabytes of memory. Therefore in addition to considering asymptotic running times of our algorithms, the hidden constant inside these terms will be of interest to us. Also, we will be concerned with the amount of memory used by our algorithms, as well as the amount of disk space used.

If we set aside some concerns there are various efficient algorithms that could be used to construct an inverted index. For example, if we are not concerned about the amount of memory available (which up to now has always been the case), we could just extract all $\langle t, d\rangle$ pairs from all documents and then sort these terms in memory. In Section 11.4 we show how to do something like this in a bit more detail. However, the scenario that we consider in this Lecture Note (and which arises very frequently in indexing for the web) is one where the computer performing the indexing is being stretched to its very limit. Hence the goal is to get the computer to handle as many documents as possible while leaving enough space to perform computations on those documents. We are happy to work partly from the disk (rather than in memory) in doing the indexing if this will allow more documents to be handled. On the other hand disk-accesses must be rare (since moving to a new location on disk is very expensive compared to main memory).

11.4 Memory-Based Inversion

If we do not need to worry about the amount of memory available, then we have a simple algorithm. We can take each document in turn, parse it, and do lexical analysis on that document (removing stop words like "the"), to extract the terms for the document in the form of $\langle t, d, f_{d,t}\rangle$ triples (where d is the document, t a term, and $f_{t,d}$ the frequency of that term in the document). Then we only need to take the union of the document lists and sort them. One way we can choose to implement this is as a *Dictionary* implementation where the *items* we store in the dictionary are *(key, list)* pairs, where each list is a list of $\langle d, f_{d,t}\rangle$ entries (for its associated term t). On recovering a set of terms from a new document d, we only need to search the dictionary and append relevant $\langle d, f_{d,t}\rangle$ pairs to the list attached to key t (inserting a new item if necessary). This is the approach we take in Algorithm 11.3.

The *Dictionary* data structure can be any *Dictionary* structure, but for efficiency it is better if it is a structure that keeps the elements somehow in sorted

order. AVL trees give $\Theta(\lg n)$ performance for inserting, searching and deleting; however hash tables (which don't have smaller overheads) can be more efficient in practice.

```
Algorithm memoryBasedInversion($D$)
1.  Create a Dictionary data structure.
2.  for $i \leftarrow 1$ downto $|D|$ do
3.      Take document $d_i \in D$ and parse it into index terms.
4.      for each index term $t$ in $D_i$ do
5.          Let $f_{d_i,t}$ be the frequency of $t$ in $d_i$.
6.          If $t$ is not in $S$, insert it.
7.          Append $\langle d_i, f_{d_i,t}\rangle$ to $t$'s list in $S$.
8.  for each term $1 \le t \le T$ do
9.      Make a new entry in the inverted file.
10.     for each $\langle d, f_{d,t}\rangle$ in $t$'s list in $S$ do
11.         Append $\langle d, f_{d,t}\rangle$ to $t$'s inverted file entry.
12.     Append $t$'s entry to the inverted file.
```

Algorithm 11.3

Sometimes we might compress t's inverted file entry before we save it to the inverted file in line 12.

Observe that with this Algorithm, the *term numbers* that label the terms can be considered to be in the same order as the terms themselves (as in Figure 11.2). We do not mention this explicitly, but it is implicit in our storing of data in S according to the key t.

The inversion time $T_I(D)$ required by this algorithm consists of

- The time $T_p(D)$ to read, parse and lexically analyse all the documents (reading and parsing takes linear time with a small constant, and if we are lucky, the same will be true for lexical analysis).

- The time $T_q(D)$ to query S and append an entry to an item of S for every $\langle t, d\rangle$ pair in D (this is $O(n\lg(n))$, where n is the number of $\langle t, d\rangle$ terms in D) .

- The time $T_w(D)$ to implement the loop in lines 8-12 to write the *inverted file*, linear in the size of the Inverted Index (the number of $\langle t, d\rangle$ terms in D).

This is a simple and reasonably-efficient algorithm. Due to the removal of stop words and recurrences of the same term in a given document, it is very likely that $O(n\lg(n))$ is no larger than a small multiple of D (and in any case $O(n\lg(n))$ is only an upper bound). This indexing algorithm probably takes linear time in the size of the input.

However, this is not our main concern. The efficiency of an Indexing algorithm is measured in hours and not in asymptotic notation. In *Managing Gigabytes* some actual figures are given for this algorithm. Another issue is the limit that exists on the size of the set of documents, due to the limited size of the memory. This raises the question of whether we need to work 'in memory' all the time. Could we store some of the Documents (or the partially constructed index) on disk throughout the algorithm (we might want to store the inverted file to disk after the algorithm has finished)? The issue for disk access is moving a head that reads from the disk; however, sequential access of a large block of the disk will not take much more time than the original access. It is true that we could implement lines 1-7 on disk without loss of performance. That is because the entries are added into the data structure in a dynamic fashion; when allocating a new vertex (if we are using an AVL implementation of *Dictionary*), it is allocated sequentially in memory (even if we think of it in pointer form). However, for lines 8-12, it is not possible to work with part of S stored in memory. When we examine the linked list of $\langle d, f_{d,t}\rangle$ entries for term t (in order to append them together onto the *inverted file*), we cannot expect these to be close to each other in memory (or on disk, should we take this approach). This phase of the algorithm would be incredibly slow on disk (again, the numbers involved are discussed in *Managing Gigabytes*).

11.5 Sort-Based Inversion

We now present an alternative algorithm for constructing an inverted index that does allow the disk to be used intelligently throughout the algorithm. The *temp file* used in the algorithm refers to a file on disk. This algorithm uses mergeSort to sort files that lie on disk; mergeSort can be used as an *external sorting* algorithm, because it processes its input in a sequential fashion).

External Merge Sort

In the sorting context, External MergeSort is used when we want to sort a set of n items, and where $n \gg K$) (read $\gg$ as "much bigger than"), where K is the number of items that we can fit into the computer's memory. The n items to be sorted are stored on disk-A (analagous to array A in standard mergeSort) and will be sorted into disk-B (analagous to the scratch array B in standard mergeSort). We initially break the data on disk-A into n/K sequentially arranged "blocks", each of size K, and individually sort each of these blocks in memory[2]. Then External MergeSort performs a "bottom-up" version of mergeSort (note that disk-A and disk-B swap roles of input/output disk as j is incremented during externalMergeSort).

Note that during lines 5-12, externalMergeSort needs to access consecutive blocks of the current input disk. Once a the start of a block is found all subsequent accesses are sequential. Moreover once we have finished with a pair of blocks the pointer to the second block is now at the start of the first block of the next pair. Hence the number of non-sequential disk accesses is approximately

[2]For the sake of simplicity we assume that in expressions such as n/K we obtain an integer. Taking care of cases when this is not so would just obscure the idea.

```
Algorithm externalMergeSort($A$)
1.  for $i = 1$ to $n/K$ do
2.      read block-$i$ of disk-A (containing $K$ items) into memory;
3.      sort block-$i$ in memory using any 'in-place' algorithm (eg quicksort);
4.      write the sort of block $i$ out to disk-B.
5.      /* disk-B now becomes current input-disk */
6.  for $j = 1$ to $\lceil \lg(n/K) \rceil$ do
7.      for $i = 1$ to $(n/2^{j+1}K)$ do
8.          buffer the first $K/3$ entries of block-$i$ and block-$i+1$ from
            current input-disk into memory ;
9.          initialize the output buffer $b$ (of size $K/3$);
10.         while there are items left to sort do
11.             perform externalMerge on the small blocks in-memory
12.             /* outputting buffer $b$ when it is full, and inputting
13.               more of block-$i$/block-$i+1$ when needed */
14.         od
15.     swap role of current input-disk between A and B.
```

Algorithm 11.4

$\sum_{j=1}^{\lceil \lg(n/K) \rceil}(n/(2^j K)+1)$, which is $\Theta(n/K)$. Aside from this there is a pointer to the current output-disk.

Finally note that there are variations on this algorithm but the differences are in the details (e.g., what buffering is used) rather than the overall idea.

Sort-based Inversion

In contrast to Algorithm 11.3, Algorithm 11.5 does not output the terms in alphabetical order. Instead they are output in the order in which they have their "first appearance" in the set of documents. This is a result of the fact that we will perform the processing of the documents hand-in-hand with the construction of the index; hence we need to construct parts of the index before we know all the terms that will belong to the final index. The ordering is memoized by associating with every term t a *term number* τ, which marks its position in the ordering.

Throughout the presentation of this algorithm, K denotes the number of inverted file entries $\langle t, d, f_{d,t} \rangle$ that can be held in memory (clearly this will not be a tight fit, rather K will be an upper bound set to ensure that the algorithm has enough working memory under these circumstances).

```
Algorithm sortBasedInversion($D$)
 1. Create a Dictionary data structure.
 2. Create an empty temp file on disk.
 3. for $i \leftarrow 1$ downto $|D|$ do
 4.     Take document $d_i \in D$ and parse it into index terms (etc).
 5.     for each index term $t$ in $D_i$ do
 6.         Let $f_{d_i,t}$ be the frequency of $t$ in $d_i$.
 7.         Check whether $t \in S$ (and check term number $\tau$).
 8.         If $t \notin S$, insert it (with the next free term number $\tau$).
 9.         Write $\langle \tau, d_i, f_{d_i,\tau} \rangle$ to temp file ($\tau$ is $t$'s term number).
10. Call externalMergeSort on temp file, to sort in order of $\langle \tau, d \rangle$ (with memory
    size $K$);
11. /* temp file now sorted. Output inverted file. */
12. for $1 \leq \tau \leq T$ do
13.     Start a new inverted file entry for $t$ (term number $\tau$).
14.     Read the triples $\langle \tau, d, f_{d,\tau} \rangle$ from temp file into $t$'s entry.
15.     Append $t$'s entry to the inverted file.
```

Algorithm 11.5

The main difference between our new algorithm and Algorithm 11.3 is the way we perform sorting. In our memory-based algorithm we never *directly* performed any sorting. However by working with a (say) AVL tree implementation

of *Dictionary*, and inserting new terms into that, and by appending new $\langle d, f_{d,t} \rangle$ entries to the end of the list for t, the effect was to perform insertionSort on the $\langle t, d \rangle$ pairs[3] in our set of documents. However, as we discussed in Section 11.4, Algorithm 11.3 cannot be adapted to use disk space. In Algorithm 11.5, we make use of externalMerge.

Our Sort-Based algorithm is organised in a few phases. At the beginning of the algorithm (lines 1-2), we set up a *Dictionary* (which will just contain (*term*, *term number*) pairs) and we open a disk file called *temp file*. In the first phase of the algorithm (lines 3-9), we examine the documents in sequential order, and append all $\langle \tau, d, f_{d,\tau} \rangle$ triples (in sequential order) into our disk file (updating the lexicon whenever we find a new term). Next in line 10 we call externalMergeSort to sort the entire *temp file* in order of $\langle \tau, d \rangle$. Recall from our discussion about externalMergeSort that this uses $\Theta(n/K)$ (specifically, about $3(n/K)$) disk accesses; since these are very expensive, we need a computer with a sufficiently large value of K (so we don't need too many runs). Finally, in lines 11-15, we use the sorted *temp file* to construct the Inverted Index on disk.

There are some issues for Algorithm 11.5 that were not relevant for Algorithm 11.3 but the most important one is disk accessing. For Sort-Based Indexing, we will require K to be smaller than n by some constant factor c (otherwise we are not saving any space in using the disk). However, if c is too large, the disk accesses begin to downgrade the running-time of the algorithm. Hence we need to keep a sensible balance (see *Managing Gigabytes* for numbers).

As with Algorithm 11.3, we do have the option of compressing an inverted file entry before appending it into the inverted file index. We should also note that it is even possible to work with a compressed format of the temporary file throughout the execution of Algorithm 11.5, leading to optimized performance. The compression has the effect of shrinking the size of the files stored on disk (or in memory), and therefore reduces the number of disk accesses that need to be performed by the algorithm. The merge operation can still be performed on the runs stored on disk, as long as care is taken. Hence with compression we can obtain a further speeded version of our Sort-based algorithm.

11.6 Further Reading

Neither [GT] nor [CLRS] present any material on Algorithms for the WWW.

A good textbook is *Managing Gigabytes*, by Ian. H. Witten, Alistair Moffat, and Timothy. C. Bell. The relevant chapter for this lecture is Chapter 5 (and some parts of Chapter 3). This book gives numbers for the time and space taken by various Index-constructing algorithms (in terms of hours, gigabytes etc).

There is also a lot of information available online. For example, here are two papers (both are rather application-specific):

- Building a distributed Full-test Index for the Web, by S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. *ACM Transactions on Information Sys-*

[3]In fact, it is a good time to mention that in general, using a balanced tree (either an AVL tree or a red-black tree) is an immediate way of modifying insertionSort so that it runs in $\Theta(n \lg n)$ time in the worst-case; recall that insertionSort has worst-case time $\Omega(n^2)$ in its standard linked list form.

tems (TOIS), **19**(3). Online at:
http://www10.org/cdrom/papers/275/

- Very Large Scale Information Retrieval, by David Hawking. In *Text and Speech Triggered Information Access*, Eds. Gregory Grefenstette and Steve Renals, 2003. Online at:
http://www.inf.ed.ac.uk/teaching/courses/tts/papers/hawking.pdf

Exercise

Simulate the externalMergeSort algorithm of this note. We assume that $K = 6$ and the input data (held on Disk A) consists of:

$$25, 2, 48, 48, 2, 36, 36, 9, 25, 9, 26, 33, 7, 46, 1, 8, 20, 40, 38, 3, 43, 12, 18, 9$$

After the initial sort of blocks of size K Disk B now has the data in the form:

$$[2, 2, 25, 36, 48, 48], [9, 9, 25, 26, 33, 36], [1, 7, 8, 20, 40, 46], [3, 9, 12, 18, 38, 43]$$

where we have delineated each sorted block by putting it in square brackets (this is just for convenience). Proceed to simulate the algorithm, a good level of granularity to use is to show what happens at each read or write to disk of data blocks of size $K/3$. As each data item is processed you could cross it out or underline it. Here is a possible diagram:

Disk A :

Disk B : [$\underline{2}$,$\underline{2}$,25,36,48,48],[$\underline{9}$,$\underline{9}$,25,26,33,36],[1,7,8,20,40,46],[3,9,12,18,38,43]

RAM:

2			...
2			...
9			...
9			...
			...
			...

The table for the RAM section shows available memory in a column. Each column is used to keep track of the state of the RAM as blocks are read in or written out (it just saves a lot of rubbing out). Each block of size $K/3$ of the RAM is delineated with a double line (of course there is just to help keep track of the simulation). the diagram shows the situation just after sub-blocks of size $K/3$ have been read into the RAM. The next phase is to start merging these.

A complete simulation would be rather long. A good compromise is to simulate to completion the processing of the first two sorted blocks of Disk B as shown in the diagram. Then start the process for the next two blocks but jump to its completion. After that Disk A will have two sorted blocks [what size will they be?]. Do a few steps of the algorithm on those till you are happy that you have understood the process.

Ranking Queries on the WWW

In the previous lecture we discussed the task of building an index for a large set of documents. This process is often called *Inversion* in the literature, because of the fact that we take a set of documents (this set can be thought of as a function from Documents to sets of Index terms), and construct an Index (which is a function from Index terms to sets of Documents). This is more-or-less inverting the original "function" corresponding to the set of documents.

The setting for the previous lecture was the large-scale environment, where there are very many documents to process. Therefore this setting includes the problem of Indexing the WWW (where documents are webpages), where the "scale" is billions of documents. The two algorithms that we presented for Indexing are not directly applicable to indexing the web, because they are sequential stand-alone algorithms that are executed on a single computer (even if we do use the hard disk for the second algorithm). Given the scale of the WWW, search engines such as Google$^{\text{TM}}$ must maintain large clusters of connected machines which individually perform indexing on their own set of webpages. The Indexing task must be synchronized across all the machines in the cluster, so that we have a (distributed) index for the entire web. This issue of how the indexing task is synchronized across the Google$^{\text{TM}}$ servers involves many concerns, some of which are hardware-related, some algorithmic, some IR-related (IR means Information Retrieval) etc. These cannot be discussed in any significant way in the time available so we leave them. If you are interested in learning something about how a search engine system fits together, have a look at the Brin and Page paper, "An Anatomy ..." (referenced in Further Reading).

In this lecture we turn our attention to a topic related to Indexing: the topic of *ranking*. Suppose we have constructed an Index for a set of webpages. Now suppose we do a search for "Pease porridge". The index will contain very many webpages that are relevant for this term. How do we rank them in order of importance? Certainly we don't want all the pages. For example, a search using Google$^{\text{TM}}$ with the term "Pease porridge", keeping the quotes to indicate that the words must be adjacent in webpages, the estimate on the first page is 42900 pages (search carried out on 30 December 2008).

One approach to ranking webpages for a particular query might be to count the number of occurrences of the query terms in the webpages (our algorithms for Indexing from Lecture Note 11 actually compute this information and store it in the Index). In practice, this is a bad approach as very informative webpages may contain very few or even no occurrences of the query term(s). For example, if we search for "University of Edinburgh", there are many webpages that have far more more occurrences of this phrase than the university webpage at `http://www.ed.ac.uk`. All the same, the university webpage is first in the ranking (as it should be).

Instead, most ranking systems take a very different approach to the ranking of webpages, by using the *link structure* of the WWW to determine a ranking. The idea is that by linking to another page, the source page confers authority on

the destination page. This is the idea behind PageRank$^{\text{TM}}$ and other models of ranking such as Kleinberg's Hub-Authority model (see Further Reading).

12.1 Basic PageRank$^{\text{TM}}$

Regardless of the model of ranking we use, the ranking of webpages will vary greatly depending with the query. Therefore for this lecture we mostly assume that we are working with a set of webpages relating to one particular query. We could think of this as being the list of pages associated with the query term in the Inverted Index for the query. In practice there will be extra pages in our set of documents, since some relevant webpages may not contain any occurrences of the query term. We sometimes see evidence of this in Google$^{\text{TM}}$ when we see the following message in a cached copy of a page:

```
These terms only appear in links pointing to this page:  ...
```

Keeping in mind the principle that a link to a webpage confers authority on that webpage, we have the following model of a ranking system. We consider the web (or the part of it pertaining to this particular query) as a *directed graph*, where the set of *vertices* $V = [N]$ (where $[N]$ is shorthand for $\{1, 2, \dots, N\}$) is the set of webpages (pertaining to that query) and the *directed edges* $E \subseteq [N] \times [N]$ of the graph are the links of the webgraph. Let $G = (V, E)$ denote our webgraph. Let $M = |E|$, the number of edges. Recall the following definitions (which apply to any graph):

Definition 12.1. Let $u \in V$ be a vertex in the webgraph.

- The *in-degree* $\text{in}(u)$ of u is the number of incoming edges to u (number of links from other pages that point to u). The set of in-edges to u is written as $\text{In}(u)$.
- The *out-degree* $\text{out}(u)$ of u is the number of outgoing edges from u (links that u has to other pages). The set of out-edges from u is written as $\text{Out}(u)$.

Definition 12.2. The *adjacency matrix* of G is the $N \times N$ matrix $A = (a_{ij})_{0 \leq i,j \leq N-1}$ with

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge from vertex number } i \\ & \text{to vertex number } j; \\ 0, & \text{otherwise.} \end{cases}$$

We are now ready to explain the PageRank$^{\text{TM}}$ model. The idea is, as mentioned above, that a link into a vertex u confers some authority on u. Therefore one possible ranking could be to assign a rank that is directly proportional to the *number of webpages* pointing into u (i.e., proportional to $\text{in}(u)$). That is a reasonable idea except for the fact that we do not consider all links into u to confer the same amount of authority. Instead, the webpages that are themselves ranked highly should be able to confer more authority than lower-ranked webpages. Also, a link from a page with few links on it should be regarded as more significant than a link from a page that has a huge number of links.

In this basic treatment we will make certain assumptions for technical reasons. We assume that all web pages in our system have some outgoing links. This is not an entirely natural assumption (especially as we really have a sub-webgraph obtained by looking at the links between the webpages relevant to our query) but it is crucial to our argument so we make it (for now). We assume that our webgraph is strongly connected—in other words, that for any two pages u and v, there is a sequence of links leading from u to v. With these assumptions, here is PageRank™: Let R_v denote the *rank of* v for any webpage $v \in [N]$. For every webpage u in our collection, the following equality must hold:

$$R_u \;=\; \sum_{v \in \mathrm{In}(u)} R_v / |\mathrm{Out}(v)|$$

So the rank of u is the total amount of Rank given from the incoming links to u. Considering the entire system of ranks of the webgraph, we can actually write this condition using a weighted form of the adjacency matrix of the webgraph:

$$(R_1, R_2, \ldots, R_N) \;=\; (R_1, R_2, \ldots, R_N) \begin{pmatrix} p_{11} & p_{12} & \ldots & p_{1N} \\ p_{21} & p_{22} & \ldots & p_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ p_{N1} & p_{N2} & \ldots & p_{NN} \end{pmatrix} \tag{12.1}$$

where

$$p_{uv} \;=\; \begin{cases} 1/\mathrm{out}(u) & \text{if } v \in \mathrm{Out}(u) \\ 0 & \text{otherwise.} \end{cases}$$

Note the following relationship between P and the adjacency matrix A:

$$p_{uv} \;=\; a_{uv}/\mathrm{out}(u),$$

for all $u, v \in V$.

We could also write (12.1) in shorthand as

$$R^T = R^T P, \tag{12.2}$$

where $P = [p_{uv}]_{u,v \in [N]}$ and R is the column vector of ranks for $[N]$ (as usual R^T denotes the transpose of a vector or matrix). Examining (12.2), notice that it is exactly the same as asking for a fixed point of the following type

$$R = P^T R, \tag{12.3}$$

Furthermore, note that $R = P^T R$ is almost the condition for R to be an eigenvector of P^T. The only difference is that the related eigenvalue λ does not seem to appear in the equation (we would expect to have $P^T x = \lambda x$). This is because the ranking R is the eigenvector of P^T *corresponding to the eigenvalue* 1.

This raises quite a few questions, the main questions being:

- How do we know that 1 is an eigenvalue of the matrix P^T?
- If 1 *is* an eigenvalue of P^T, then how do we know that it is a *simple* eigenvalue (i.e., that any two ranking vectors R that satisfy $R^T = R^T P$ are *linearly dependent*—one is a non-zero multiptle of the other)?

These questions are important if the PageRank™ model is to mean anything in the context of our webgraphs.

We start by considering the first question: how do we know that the matrix has an eigenvector of value 1? Have a look at the system of equations (12.1). Consider the particular row of matrix P corresponding to the webpage $u \in [N]$. Then the sum of the values in that column is the sum of fractional rankings that u passes to each of its links:

$$\sum_{v=1}^{N} p_{uv} \;=\; \Big(\sum_{v \in \mathrm{Out}(u)} 1/\mathrm{out}(u) \Big) + \big(\textstyle\sum_{v \notin \mathrm{Out}(u)} 0 \big) \;=\; |\mathrm{Out}(u)|/\mathrm{out}(u) + 0 \;=1.$$

So for every row of the matrix P, the entries of that row sum to 1. There is a special name for this sort of matrix in the literature—it is called a *stochastic matrix*, and a stochastic matrix is guaranteed to have the eigenvalue 1 (we omit the proof). So first question answered.

How can we be sure that the matrix cannot have two linearly independent eigenvectors for 1? This depends on our assumptions about the structure of the web graph. We assumed that for every pair of webpages u and v, there was a sequence of links leading from u to v through our webgraph—a graph with this property is known as a *connected graph*. With this assumption (and another tiny assumption, that the series of links from u to v are *aperiodic*[1]), then we can be assured of the uniqueness of the vector R. Again we omit the proof.

12.2 Finding Eigenvectors for eigenvalue 1

We now make an observation about how we can find a ranking R in the PageRank™ model. It is not as difficult as it might seem, particularly because we know that R corresponds to the eigenvector 1. We just have to observe that our conditions for R given in (12.1) simply correspond to a linear system of equations. You will have learnt various methods for solving these in your mathematics classes, and moreover, efficient algorithms for applying these methods do exist (though that's not the approach taken with respect to web ranking—see later). As a very simple example, consider the webgraph of Figure 12.3, perhaps dating back to the early 1990s when there were hardly any webpages on the web.

This mini-graph satisfies all of the conditions that were mentioned in this section, even the condition of aperiodicity (this is obvious because the graph has a cycle of length 2 and a cycle of length 3; so no need to look at any more). Therefore there is a unique (up to constant multiples) eigenvector R satisfying $R^T = R^T P$. To find this particular eigenvector, we write down the matrix P, assuming the webpages are in order u, v, w, z:

$$(R_u, R_v, R_w, R_z) \;=\; (R_u, R_v, R_w, R_z) \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix}. \tag{12.4}$$

[1]This condition requires that if we look at the lengths of all the cycles of the graph then the largest integer that divides *all* of them is 1. Note that if we find a cycle of length 5 (say) and one of length 8 (say) in the graph we know it is aperiodic [why?], an important saving in time.

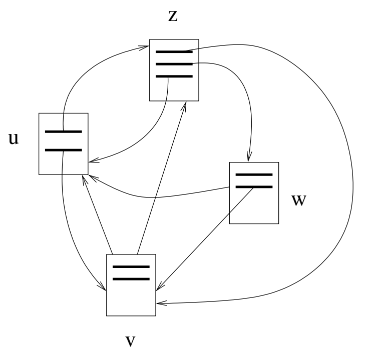

Figure 12.3. An example webgraph returned by a rare query in ancient times.

We can straightaway read off the equality $R_w = R_z/3$. Then we can eliminate R_w by first expressing R_w in terms of R_z on the right-hand side of the equations.

$$(R_u, R_v, R_w, R_z) \quad = \quad (R_u, R_v, R_w, R_z) \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ \frac{1}{3}+\frac{1}{6} & \frac{1}{3}+\frac{1}{6} & \frac{1}{3} & 0 \end{pmatrix}. \tag{12.5}$$

Now we can just write $R_w = R_z/3$ to one side and continue with a smaller matrix:

$$(R_u, R_v, R_z) \quad = \quad (R_u, R_v, R_z) \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}. \tag{12.6}$$

This is equivalent to:

$$(R_u, R_v - R_z, R_z) \quad = \quad (R_u, R_v, R_z) \begin{pmatrix} 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}. \tag{12.7}$$

Observe that the middle equation reads $R_v - R_z = (R_z - R_v)/2$. This means that $R_v = R_z$. Observe that the final equation states that $R_z = (R_u + R_v)/2$. Then with our new knowledge that $R_z = R_v$, this tells us that $R_u = R_z$ also.

An alternative, but equivalent approach, is to multiply out the right hand side of (12.4) to obtain a set of linear equations:

$$\begin{aligned} R_u &= \frac{1}{2}R_v + \frac{1}{2}R_w + \frac{1}{3}R_z \\ R_v &= \frac{1}{2}R_u + \frac{1}{2}R_w + \frac{1}{3}R_z \\ R_w &= \frac{1}{3}R_z \\ R_z &= \frac{1}{2}R_u + \frac{1}{2}R_v. \end{aligned}$$

It is clear that subtracting the second equation from the first would give us useful information, we obtain

$$R_u - R_v = \frac{1}{2}R_v - \frac{1}{2}R_u$$

from which it follows that $R_v = R_u$. Substituting into the fourth equation we obtain $R_z = R_u$. This method is probably preferable for such small examples.

Hence we have a solution $R_u = R_v = R_z$, $R_w = R_z/3$. So if we take any value r and set $R = (r, r, r/3, r)$, we get a solution to (12.4). Note all these solutions are essentially one eigenvector, they are all constant multiples of $(1, 1, 1/3, 1)$. Of course all that a constant multiple does is to change the scale of ranking but it does not alter relative positions. It is a good idea to check that this solution works in (12.4) as an exercise. Note that this solution is not the same as the answer we would get by allocating rank directly according to in-degree.

12.3 PageRank™ in general

In general of course, the web graph will not satisfy the extremely nice conditions that we laid out in the previous subsection (where we want a sequence of links from any page u to any other page v). There is a modified definition of PageRank™ which works in the general case. The intuition behind this ranking system for general web graphs is not quite as elegant as before, however, it allows us always to come up with a ranking regardless of the structure of the webgraph.

We observe first that when there are webpages with no outgoing links, they in some sense "leak" some rank value from the entire web graph. That's because they "take in" rank, but never send any out (corresponding to an all-0's column vector for that page). Hence the eigenvalue that we should consider may be slightly smaller than 1. PageRank™ takes care of this by using the symbol $1/c$ for this eigenvalue, and specifying that it should be as large as possible. The other condition that was required by our basic version of PageRank™ was that there should be a sequence of links from u to v for any pair of pages u and v. We accomplish this by assuming that for every page u, there is a small amount of rank assigned to every page v in the entire system (basically this models the chance that the user might hop to another page at random, without using the links). This means that there is an artificial sequence of links from any page u to any other page v (by co-incidence, this also ensures the overall matrix is aperiodic).

In this more general setting, the PageRank™ model requires that for the minimum value of $c > 1$, every webpage u in our collection, the following equality should be satisfied for every page u:

$$R'(u) \;=\; c^{-1}(1-p) \sum_{v \in \mathrm{In}(u)} R'(v)/|\mathbf{Out}(v)| + c^{-1}(p/N)\mathbf{1}$$

Here p is the (small) fraction of ranking that is leaked uniformly to all pages in the webgraph, $1/c$ is the maximum eigenvalue (for this new system), and **1** is the vector consisting of 1 in every position, i.e., $(1, 1, \ldots, 1)$, the length being deduced from the context.

In this setting we know that, apart from linearly dependent solutions, we have a ranking (for the maximum eigenvalue) for any web graph we consider.

Techniques for solving a linear system apply to this case, though they are a bit more complicated than for the basic (unrealistic) case. In practice, a web crawl (a bit like a *random walk* on the webgraph) is used in constructing the ranking.

12.4 Further Reading

Neither [GT] nor [CLRS] present any material on Algorithms for the WWW. There is also a lot of information online, for example:

- An Anatomy of a Large-Scale Hypertextual Web Search Engine, by Sergey Brin and Lawrence Page, 1998. Online at:

  `http://www-db.stanford.edu/ backrub/google.html`

- The PageRank Citation Ranking: Bringing Order to the Web, by Page, Brin, Motwani and Winograd, 1998. Available online from:

  `http://dbpubs.stanford.edu:8090/pub/1999-66`

- Authoritative Sources in a Hyperlinked Environment, by Jon Kleinberg. Available Online from Jon Kleinberg's webpage:

  `http://www.cs.cornell.edu/home/kleinber/`