Analysis of Algorithms

Peter Rossmanith

Institute for Theoretical Computer Science

RWTH Aachen

Lecture Notes and Exercises, Winter 2020

commit a38ed5be63221004ba92b9c2bb4cd9282cef83f7
Author: Peter Rossmanith <rossmani@cs.rwth-aachen.de>
Date: Mon Nov 23 09:57:51 2020 +0100

Contents

1 Analysis of Quicksort **1**

1.1 The number of partitioning phases 4

1.2 The Number of Comparisons while Partitioning 10

1.3 The number of swaps in the **do**-loop 10

1.4 The number of insertion phases 11

1.5 Number of swaps during insertion-sort 12

1.6 Conclusion 14

2 The Kirchhoff laws **21**

3 Recurrence relations **27**

3.1 Classification of recurrence relations 28

3.2 Creating a table 29

3.3 Guessing a solution andproving it by induction 29

3.4 Looking up the solution 30

3.5 Mathematica, Maple, Maxima, etc. 31

3.6 Hidden products and sums 33

3.7 Linear recurrence relations with constant coefficients 34

3.8 Summation factor 35

3.9 The Repertoire Method 37

3.10 Order Reduction 40

3.11 Extracting recurrence relations from algorithms 42
3.12 Searching an unordered array 45
3.13 Ordered arrays and binary search trees 49

4 Generating functions 59
4.1 Counting Data Structures with Generating Functions . . . 65
4.2 Bivariate Generating Functions 68
4.3 Exponential Generating Functions 68
4.4 The Symbolic Method 69
4.5 Average Stack Height 74

5 Asymptotic Estimations 79
5.1 Euler's summation formula 80
5.2 Singularity Analysis 81
5.3 Meromorphic functions 84
5.4 Algebraic Singularities 89
5.5 The Saddle Point Method 93
5.6 The Restricted Saddle Point Method 100

A Solutions to Selected Exercises 105

B MIPS Cheat Sheet 127

C Tests 129

Chapter 1

Analysis of Quicksort

We start our journey into the Analysis of Algorithms with an example. It consists of a well-known and very efficient sorting algorithm. We will see that even a very complicated algorithm can successfully be analyzed mathematically.

This first analysis of an algorithm contains almost every single important ingredient that may occur in typical situations that we may encounter when we design our own algorithms and try to analyze them. While we just take a glance on these various aspects in this introductory chapter, we will revisit them later on and learn about them in more detail:

1. Before starting to analyze the running time of some algorithm, we have to understand it completly and in every detail; otherwise a precise analysis is impossible. After having learnt about the purpose of every single instruction, we have to find an intuitive description of the *number of times this instruction is executed.* If a block of instruction is not interrupted by a branch statement, all instruction in the block can be analyzed together. Apart from this very simple rule we will later encounter several other method on how to reduce the number of instructions that have to be analyzed individually.

2. If we want to carry out an *Average case analysis*, i.e., analyse an algorithm's *expected* behavior, we need a statistical model for the inputs to model an appropriate probability distribution.

3. With the help of the—up to now—rather vague *intuivitve description*, we have to find a closed formula for the number of executions of each instruction. Here we have to take the probability distribution into account when counting the expected rather than the worst case number. Often it is impossible to find a exact closed formula or it requires too high an effort. In that case we have to be content with closed, but only approximate, formula.

4. At the end we just have to add the individual times for each instruction to get the overall expected running time in relation to the input length.

The famous Quicksort algorithm is well suited as an introductory example because it is not too trivial and well known. We will concentrate on a practical, highly optimized version rather than on a simplified one, which you will often find in beginners' textbooks.[1]

One drawback of naked quicksort is its bad performance on very small arrays, on which it is beaten by much simpler algorithms. Hence, we use a quicksort variant that partitions (and then recursively sorts) an array only if its length is bigger than a constant M. At the end we can use one run of straight-insertion sort to finish the job by cleaning up the remaining unsorted short subarrays. Another optimization addresses space consumption rathen than running time: After partitioning we sort the smaller of the two subarrays first. This well-known trick keeps the recursion depth small because the array size is at least halved in each recursive call. For efficiency reason the recursive calls are simulated by direct calls and the usage of our own stack. Figure 1.1 contains a complete program written in the language C that implements all ideas mentioned in this paragraph.

We assume that the input consists of N different numbers and want to analyze, how often each instruction in the program is executed on average, if every permutation of the given numbers occurs with the same probability. This is a standard assumption for sorting problems. Initially, the input is located in the array $a[1],\dots,a[N]$ and the sorted sequence is to be found in the same spot upon program termination.

[1] In this script we follow closely the analysis of Quicksort by Knuth [4], which is definitely not a beginner's textbook.

```
void quicksort(void)
{
    int i, j, l, r, k, t;
    l = 1;  r = N;
    if(N > M)
      while(1) {
        i = l − 1;  j = r;  k = a[j];
        do {
            do { i++; } while(a[i] < k);
            do { j−−; } while(k < a[j]);
            t = a[i];  a[i] = a[j];  a[j] = t;
        } while(i < j);
        a[j] = a[i];  a[i] = a[r];  a[r] = t;
        if(r − i ≥ i − l) {
            if(i − l > M) { push(i + 1, r);  r = i − 1; }
            else if(r − i > M) l = i + 1;
            else if(stack_is_empty) break;
            else pop(l, r);
        }
        else {
            if(r − i > M) { push(l, i − 1);  l = i + 1; }
            else if(i − l > M) r = i − 1;
            else if(stack_is_empty) break;
            else pop(l, r);
        }
      }
    for(i = 2;  i ≤ N;  i++)
        if(a[i − 1] > a[i]) {
            k = a[i];  j = i;
            do { a[j] = a[j − 1];  j−−; } while(a[j − 1] > k);
            a[j] = k;
        }
}
```

Figure 1.1: C-program for Quicksort

You can find the whole program a second time in Figure 1.2, but in a different layout that reminds of a flow chart. Instructions that are not separated by branches or target of branches are grouped into blocks. The program flow is indicated by arrows between the blocks. Next to each block you can find a symbolic name for the number of times this block is executed in the form of a variable or a short expression that may involve several variables.

Let us start by considering the variable A. This variable occurres next to several block, which implies that number of execution for those blocks are identical.

Why can we use the same variable for the two blocks $i = l-1;\ j = r;\ k = a[j]$ and $a[j] = a[r]\ldots$? The answer is quite simple: The flow into a set of blocks M must be exactly identical to the flow out of M. The flow is the program flow, i.e., the flow into a block is the number of times the block is entered and the flow out is the number of times the block is left. This situation is quite similar to solenoidal vector fields in physics (e.g., the magnetic field) or the electrical flow in a resistor network.

If the block $i = l-1;\ j = r;\ k = a[j]$ is executed A times, then it will be left A times. There is only one outgoing arrow from this block. Let us denote the set of blocks between $i = l-1;\ j = r;\ k = a[j]$ and $a[j] = a[r]\ldots$ by M. Then M will be left A times, too, which is again possible only by one arrow that leads to the block $a[j] = a[r];\ a[i] = a[j];\ a[j] = t$. We can conclude that this block is executed exactly A times, too. This line of reasoning has been relatively easy. We can discover other relationship like this in a similar way, e.g., that $I' + I'' = 1$ or $A' + A'' = A - 1$. In this way we greatly reduce the number of independent variables whose value has to be analyzed.

We will address reducing the number of variables using the flow relations in a systematic way in Chapter 2.

1.1 The number of partitioning phases

Let us return to the analysis of A. What is the intuition behind this number? The **while**-loop in Figure 1.1 is executed exactly A times. Each

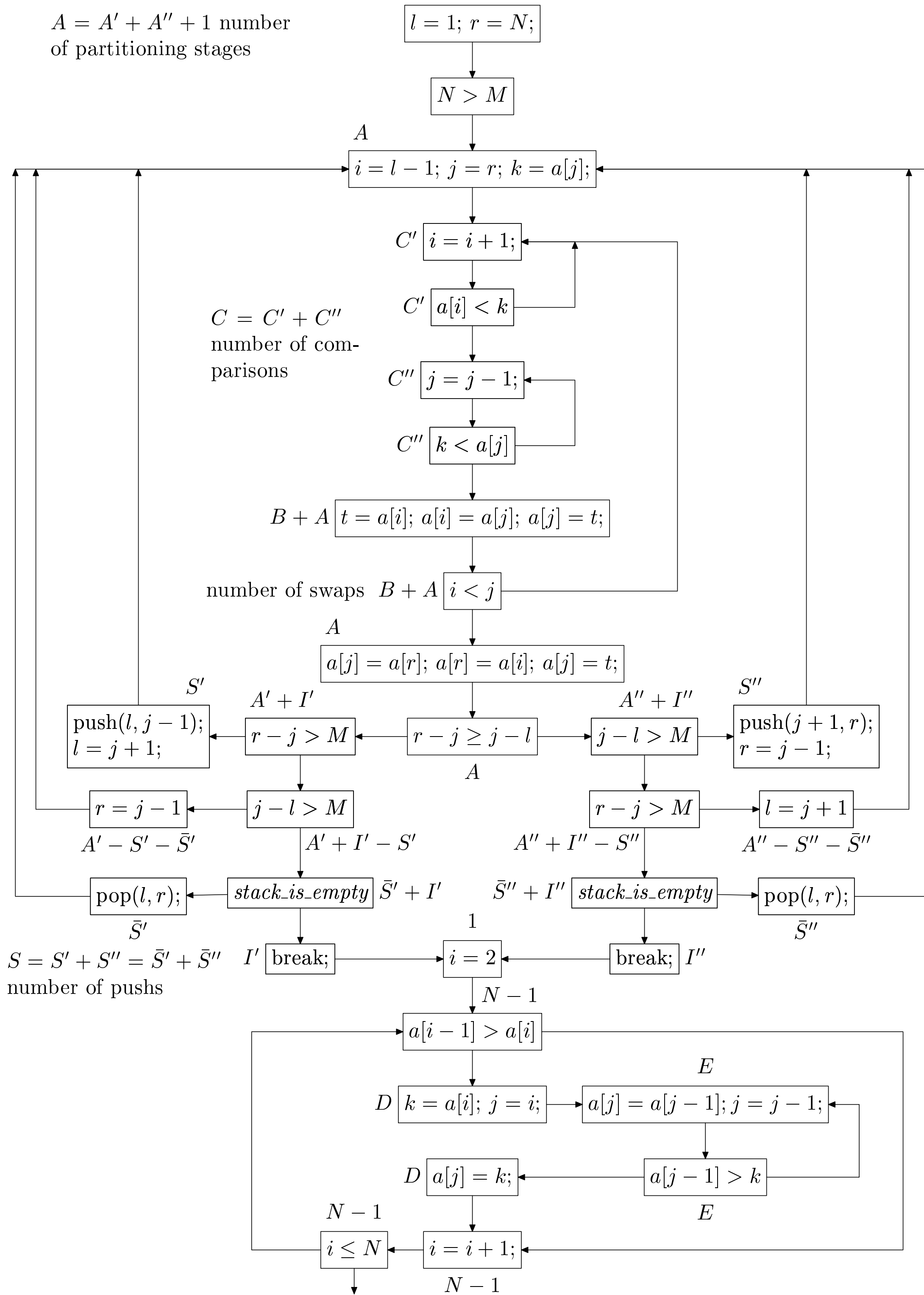

Figure 1.2: Program flow chart for the Quicksort program

execution corresponds to a partitioning of a subarray. Hence, we interpret A as the number of partitioning phases. This intuitive description is enormeously helpful. From this point on, we do not have to look at the C-program anymore, when analysing A. We just have to look at the abstract Quicksort algorithm. Even if we change the C-program the analysis of A will remain sound—it is the number of partitioning phases that is clearly independent of the concrete implementation.

Let A_N be the expected number of partitioning phases if we sort N keys by Quicksort. If $N > M$, the input is partitioned once and three subarrays are established. The middle one consists only of the pivot element and will be left untouched. The first and last subarray will be recursively sorted. This leads to additional A_k and A_{N-1-k} partitioning phases if the first and last subarray have the length k and $N-1-k$. The number k is between 0 and $N-1$. It is easy to see that the probability for each of those possibilities is exactly $1/N$: We assumed, after all, that every permutation occurs with the same probability. These ideas lead to the following relation:

$$\begin{aligned} A_N &= 1 + \frac{1}{N}\sum_{k=0}^{N-1}(A_k + A_{N-1-k}) \\ &= 1 + \frac{2}{N}\sum_{k=0}^{N-1} A_k, \quad \text{for } N > M \end{aligned}$$

If $N \leq M$, on the other hand, then clearly $A_N = 0$.

In the following we will encounter many more recurrence relations that look familiar to this one. We can write all of them as

$$X_N = \frac{2}{N}\sum_{k=0}^{N-1} X_k + f_N, \quad \text{for } N > M$$

with different functions f_k. In the case of A_N we have $f_k = 1$.

It is not very hard to solve recurrences of this form. The first problem we encounter is that X_N depends on all $X_0, \dots, X_{N-1}$ instead on only a small number of different X_i's. To overcome this problem, the first step is to turn the recurrence into one of *finite order*. We can achieve that by subtracting X_{N-1} from X_N after having got rid of the interfering factors $1/N$

and $1/(N-1)$:

$$\begin{aligned} NX_N &= 2\sum_{k=0}^{N-1} X_k + Nf_N \\ (N-1)X_{N-1} &= 2\sum_{k=0}^{N-2} X_k + (N-1)f_{N-1} \end{aligned}$$

Subtraction yields

$$NX_N - (N-1)X_{N-1} = 2X_{N-1} + Nf_N - (N-1)f_{N-1}$$

or

$$NX_N = (N+1)X_{N-1} + Nf_N - (N-1)f_{N-1}, \quad \text{for } N > M+1.$$

This is a linear recurrence of first order. Such recurrences can routinely be solved by a technique called *summation factor*, as we will see later. Here this technique asks us to multiply the equations by $1/N(N+1)$:

$$\frac{X_N}{N+1} = \frac{X_{N-1}}{N} + \frac{Nf_N - (N-1)f_{N-1}}{N(N+1)}$$

Using the substitutions

$$Y_N = \frac{X_N}{N+1} \text{ and } g_N = \frac{Nf_N - (N-1)f_{N-1}}{N(N+1)}$$

yields the very simple equation

$$Y_N = Y_{N-1} + g_N, \text{ for } N > M+1.$$

We can easily solve the recurrence, but have to be careful that it holds only for $N > M+1$. It is a common mistake not to track exactly under which conditions derived equations are valid.

$$Y_N = Y_{M+1} + g_{M+2} + g_{M+3} + \cdots + g_N = Y_{M+1} + \sum_{k=M+2}^{N} g_k$$

and, after substituting back into the variable X_N, we get the solution

$$X_N = \frac{N+1}{M+2}X_{M+1} + (N+1)\sum_{k=M+2}^{N} \frac{kf_k - (k-1)f_{k-1}}{k(k+1)}.$$

Let us return to the analysis of A_N. Here $f_k = 1$ and $A_{M+1} = 1$. Replacing X_N and f_k accordingly leads to

$$\begin{aligned} A_N &= \frac{N+1}{M+2} + (N+1) \sum_{k=M+2}^{N} \frac{1}{k(k+1)} \\ &= \frac{N+1}{M+2} + (N+1) \left(\frac{N}{N+1} - \frac{M+1}{M+2} \right) = \frac{2N-M}{M+2}. \end{aligned}$$

We finally arrived at a closed formula for A_N. Do not forget that we proved this formula only for $N > M+1$. We also know that $A_N = 0$ for $N \leq M$. Finally, $A_{M+1} = 1$, which we had to establish earlier in the analysis of A_N. We can write down a closed formula for A_N that is valid for all values of N by using a case distinction:

$$A_N = \begin{cases} 0 & \text{if } N \leq M, \\ 1 & \text{if } N = M+1, \\ \dfrac{2N-M}{M+2} & \text{if } N > M+1. \end{cases}$$

Fortunately, however, $(2N - M)/(M + 2) = 1$ if $N = M + 1$ and we can merge the last two cases into one. Our final formula, which hardly can be simplified more, is

$$A_N = \begin{cases} 0 & \text{if } N \leq M, \\ \dfrac{2N-M}{M+2} & \text{if } N > M. \end{cases}$$

Let us check the validity of this formula on some special cases. What happens if $N = M + 2$? Of course, Quicksort partitions the array once. There are $M + 2$ different possibilies for choosing the pivot element and each choice bears a probability of exactly $1/(M + 2)$. There are exactly two possible choices for the pivot that force the algorithm to carry out a second partitioning. This happens only if the pivot element is either the smallest or the biggest key because then one of the subarrays has size M. Hence, with a probability of $M/(M+2)$ the algorithm partitions once and with a probability of $2/(M + 2)$ twice. The expected value is therefore

$$A_{M+2} = \frac{M}{M+2} + 2\frac{2}{M+2} = \frac{M+4}{M+2} = 1 + \frac{4}{M+2}.$$

Let us see, to what our closed formula for A_{M+2} evaluates:

$$A_{M+2} = \frac{2(M+2)-M}{M+2} = \frac{M+4}{M+2} = 1 + \frac{4}{M+2}$$

Of course, they coincide. It is advisable to test the outcome of a complicated analysis that finally yields a closed formula on some easy special cases because you can have made a mistake.

One final remark on the analysis of A_N regards the final summation we had to carry out. When solving recurrence relations—especially when we simplify them in a sequence of steps—very often we end up with a summation. For this reason, solving summations by providing an exact or approximate closed form turns out to be very important in the analysis of algorithms.

Here the summation was quite easy to solve. It is a telescopic sum because

$$\sum_{k=M+2}^{N} \frac{1}{k(k+1)} = \sum_{k=M+2}^{N} \left(\frac{1}{k} - \frac{1}{k+1}\right)$$

and consequently almost all terms cancel each other. With the advent of computer algebra systems, however, learning techniques how to solve summations become less important nowadays. This summation can be easily solved for us by a system like `maxima`:

```
Maxima 5.23.2 http://maxima.sourceforge.net
using Lisp SBCL 1.0.38-3.el6
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1) nusum(1/(k*(k+1)), k, M+2, N);
                                       N - M - 1
(%o1)                               ---------------
                                    (M + 2) (N + 1)
```

If the summation has no closed form we will see how to approximate its value with very small additional error terms.

1.2 The Number of Comparisons while Partitioning

If we partition a subarray of size N, the indexes i and j point initially to the begin and end of the subarray. When the criterion $i < j$ is no longer true, the indexes have crossed over and the partitioning phase is ended. Whenever i is increased of j decreased, exactly one comparison is carried out. In the end $i = j + 1$ holds (i.e., $j - i = -1$) and in the beginning $j - i = N - 1$. Hence, the difference between j and i decreases with each comparison from $N+1$ to -1 and the total number of comparisons it $N+1$. This is the number of comparisons in *one* partitioning phase. The total expected number of comparisons in all partitioning phases can be stated by the recurrence relation

$$C_N = N + 1 + \frac{2}{N} \sum_{k=0}^{N-1} C_k,$$

which is again of the general form with $f_k = k + 1$ and $C_{M+1} = M + 2$.

It is now easy to get a closed formula for its solution using harmonic numbers $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$.

$$\begin{aligned} C_N = N + 1 + (N+1) \sum_{k=M+2}^{N} \frac{k(k+1) - (k-1)k}{k(k+1)} &= \\ = N + 1 + 2(N+1) \sum_{k=M+2}^{N} \frac{1}{k+1} & \\ = N + 1 + 2(N+1)(H_{N+1} - H_{M+2}) & \end{aligned}$$

Very often only the number of comparisons is analyzed in textbooks and usually $M = 1$ and all comparisons occur while partitioning. If $M = 1$ this yields $2(H_{N+1} - 8/6)(N+1) = 2H_{N+1}N - \frac{8}{3}N + o(N)$.

1.3 The number of swaps in the do-loop

Let $B_N + 1$ the number of swap operations in the **do**-loop. For efficiency reasons—it saves one if-command—the algorithm performs one last swap

that is not necessary and has to be swapped back.[2] It makes sense to define B_N as the number of *real* swaps (that are not taken back) according to our philosophy that variables that we analyze have a nice intuitive meaning.

If we partition the array in two subarrays of sizes k and $N-1-k$, then the expected number of swaps that have occurred between them is $k(N-1-k)/(N-1)$. This gives us the recurrence relation

$$B_N = \frac{1}{N}\sum_{k=0}^{N-1}\left(B_k + B_{N-1-l} + \frac{k(N-1-k)}{N-1}\right) = \frac{2}{N}\sum_{k=0}^{N} B_k + \frac{N-2}{6}.$$

Here $f_k = (N-2)/6$ and $B_{M+1} = (M-1)/6$. Its solution is

$$\begin{aligned} B_N &= \frac{(N+1)(M-1)}{6(M+2)} + \frac{(N+1)}{6}\sum_{k=M+2}^{N}\frac{k(k-2)-(k-1)(k-3)}{k(k+1)} \\ &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2}. \end{aligned}$$

1.4 The number of insertion phases

Let us proceed to D_N and ponder what intuition can be found behind this variable. Whenever $a[i-1] > a[i]$, the algorithms inserts $a[i]$ into the already sorted subarray $a[1\ldots i-1]$. The variable D_N tells us exactly, how often such an insertion takes place. We should not forget that this kind of insertions happen only in the second phase of the algorithm. We get the recurrence

$$D_N = \frac{2}{N}\sum_{k=0}^{N-1} D_k$$

for $N > M$. We can look at what happens in the second phase from the perspective of the first phase. The array of length N is partitioned recursively into smaller and smaller subarrays until their sizes are at most M. Let us call these final subarrays *small*. At the end of the day D_N is the sum of all D_{k_i}'s if k_i are the length of all small subarrays.

[2] As a **do**-loop is performed at least once and the array might be already sorted and no swaps should take place, we cannot avoid at least one superfluous swap without guarding it by an if-statement.

We have a simple recurrence for the case that $N > M$, but what happens if $N \leq M$? Let us assume that i is the last index that belongs to the subarray of length N where $N \leq M$. Then the insertion that we want to count take place iff $a[i-1] > a[i]$ in the **for**-loop.

How big is the probability that $a[i-1] > a[i]$? We have to consider that $a[1], \dots, a[i-1]$ has already been sorted by insertion sort. Before that, $a[1, \dots, i]$ was in random order. At this point of time $a[i-1] > a[i]$ iff $a[i]$ is not the biggest key in $a[i-M+1], \dots, a[i]$. It is the biggest key only if it is bigger than the other $i-1$ keys. The probability for this event is $1 - 1/i$.

By a simple summation we get

$$D_N = \sum_{i=2}^{N} (1 - 1/i) = N - H_N \text{ for } N \leq M.$$

Let us look at some small values that we get from this formula: $D_0 = 0$, $D_1 = 0$, $D_2 = 1/2$. Are these correct? Yes, only if $N \geq 2$ the body of the **for**-loop is executed at all. If $N = 2$ then an insertion takes place if $a[2] > a[1]$. This happens with probability $\frac{1}{2}$.

This formula is also the key to get a grip on D_{M+1}, which we require to get a closed solution of D_N.

$$D_{M+1} = \frac{2}{M+1} \sum_{k=0}^{M} D_k = \frac{2}{M+1} \sum_{k=0}^{M} (k - H_k) = M - 2H_{M+1} + 2$$

For $N > M$ the closed formula for D_N is:

$$D_N = \frac{N+1}{M+2} D_{M+1} = \frac{N+1}{M+2}(M + 2 - 2H_{M+1}) = (N+1)\left(1 - \frac{2H_{M+1}}{M+2}\right)$$

In particular we see that $D_N = \Theta(N)$, if $M > 1$ is a constant. Hence, only a linear number of keys is moved to correct the subarrays that were left unsorted by the first phase. This behavior is not surprising.

1.5 Number of swaps during insertion-sort

The second phase of our highly optimized Quicksort algorithms is—as we have seen—basically *insertion sort*. Exactly E_N pairs of keys, which are

not in the right order, are swapped. After they have been swapped they are in the right order and stay in this relative order for all times. This is not exactly how the algorithm works because of efficiency reasons the keys are not pairwise swapped but cyclicly in larger blocks. The result is, however, the same and we can pretend they are swapped pairwise, which is much easier to imagine (and therefore analyze).

Whenever two keys are in the wrong order, the E-blocks in Figure 1.2 are executed once. That is exactly the number of *inversions* of the permutation that sorts the input. An inversion of a permutation is the number of pairs that are out of order. Formally, if $\pi\colon \{1,\ldots,n\} \to \{1,\ldots,n\}$ is a permuation, then

$$|\{\,\{i,j\} \mid 1 \le i < j \le n,\ \pi(i) > \pi(j)\,\}|$$

is the number of inversions of π. Hence we have a very nice intuitive description of E_N—it is simply the expected number of inversions of a random permutation.

A permuation of n keys has $n(n-1)/2$ pairs and each pair of keys has the wrong order with a probability of $1/2$. The probability distribution nevertheless is quite complicated because these events are clearly not independent from each other. Fortunately, we only need the expected value of the number of inversions. Because of linearity of the expected value the result simply is $n(n-1)/4$.

Let E_N be the number of inversions of the input array after the first phase of the algorithm. Then E_N is the number of times the E-blocks are executed. We get this recurrence for E_N:

$$E_N = \begin{cases} \dfrac{2}{N}\displaystyle\sum_{k=0}^{N-1} E_k & \text{für } N > M \\[2ex] \dfrac{1}{2}\dbinom{N}{2} & \text{für } N \le M \end{cases}$$

Again the form of the recurrence is in the familiar shape. Routinely, we first find out what E_{M+1} is:

$$E_{M+1} = \frac{2}{M+1}\sum_{k=0}^{M}\frac{k^2}{4} = \frac{M(M-1)}{6}$$

For $N > M$ we get, again following our routine,

$$E_N = \frac{N+1}{M+2} E_{M+1} = \frac{N+1}{M+2} \frac{M(M-1)}{6}.$$

1.6 Conclusion

You can find all results in the following table:

$$\begin{aligned}
A_N &= \frac{2N-M}{M+2} \\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2} \\
C_N &= N + 1 + 2(N+1)(H_{N+1} - H_{M+2}) \\
D_N &= (N+1)(1 - 2H_{M+1}/(M+2)) \\
E_N &= \frac{1}{6}(N+1)M(M-1)/(M+2) \\
S_N &= (N+1)/(2M+3) - 1
\end{aligned}$$

Figure 1.3 contains the C-program from figure 1.1 in the assembler language of a MIPS-processor. We choose this type of processor for the following reasons:

1. It is a typical RISC-processor and representative for processor used today and at least in the near future.
2. Among existing processors it has a relatively easy to learn instruction set. There are no special purpose register, no register windows, or other rather specialized features. You can easily learn all important instructions within a few minutes and you can read MIPS assembler programs immeadiately if you have been exposed to similiar machine languages before. This processor is also used in many embedded systems and portable computers today, which proves that it has a realistic, real world design.
3. We will also see later that it is not sufficient to analyse a program written in high level, compiled language if you are interested what effects small changes in your algorithm imply. One good example are

```
quicksort:
1           la    $7,a
            lw    $2,sp
            li    $3,1000
            li    $5,1
            move  $13,$7
            la    $4,s
$L23:
A           sll   $6,$3,2
            addu  $6,$7,$6
            sll   $10,$5,2
            lw    $14,0($6)
            addu  $10,$7,$10
            addiu $8,$5,-1
            j     $L3
            move  $9,$3

$L4:
B+A+C'-2    addiu $10,$10,4
            move  $8,$6
$L3:
C'          lw    $11,0($10)
            slt   $12,$11,$14
            bne   $12,$0,$L4
            addiu $6,$8,1

B+A         addiu $12,$9,-1
            sll   $12,$12,2
            addu  $12,$7,$12
$L5:
C''         lw    $24,0($12)
            addiu $9,$9,-1
            slt   $15,$14,$24
            bne   $15,$0,$L5
            addiu $12,$12,-4

B+A         sll   $15,$9,2
            addu  $15,$7,$15
            slt   $12,$6,$9
            sw    $24,0($10)
            bne   $12,$0,$L4
            sw    $11,0($15)

A           sll   $14,$6,2
            addu  $14,$13,$14
            lw    $9,0($14)
            sll   $12,$3,2
            sw    $9,0($15)
            addu  $12,$13,$12
            lw    $24,0($12)
```

```
            subu  $10,$3,$6
            subu  $9,$6,$5
            slt   $15,$10,$9
            sw    $24,0($14)
            bne   $15,$0,$L6
            sw    $11,0($12)

A''+I''     slt   $9,$9,4
            bnel  $9,$0,$L7

S''         slt   $10,$10,4
            addiu $9,$2,1
            sll   $10,$2,2
            sll   $9,$9,2
            addu  $10,$4,$10
            addiu $6,$6,1
            addu  $9,$4,$9
            sw    $6,0($10)
            addiu $2,$2,2
            sw    $3,0($9)
            j     $L23
            move  $3,$8

$L7:
A''+I''-S'' beq   $10,$0,$L23
            addiu $5,$6,1

            j     $L25
$L6:
A+1-S''     slt   $10,$10,4

A'+I'       bnel  $10,$0,$L10

S'          slt   $9,$9,4
            addiu $9,$2,1
            sll   $10,$2,2
            sll   $9,$9,2
            addu  $10,$4,$10
            addu  $9,$4,$9
            sw    $5,0($10)
            addiu $2,$2,2
            sw    $8,0($9)
            j     $L23
            addiu $5,$6,1

$L10:
A'+I'-S'    beq   $9,$0,$L23
            move  $3,$8

$L25:
```

```
S̄'+S̄''+1    beq   $2,$0,$L26
            addiu $3,$2,-1

S̄'+S̄''      addiu $2,$2,-2
            sll   $3,$3,2
            sll   $5,$2,2
            addu  $3,$4,$3
            addu  $5,$4,$5
            lw    $3,0($3)
            j     $L23
            lw    $5,0($5)

$L26:
1           li    $9,1073676288
            sw    $0,sp
            ori   $9,$9,0xffff
            la    $4,a+4
            li    $2,2
            la    $7,a
            li    $8,1001
$L16:
N-1         lw    $5,4($4)
            lw    $3,0($4)
            slt   $3,$5,$3
            beql  $3,$0,$L28

D           addiu $2,$2,1
            addu  $3,$2,$9
            sll   $3,$3,2
            addu  $3,$7,$3
            move  $6,$2
$L15:
E           lw    $10,-4($3)
            lw    $11,0($3)
            slt   $10,$5,$10
            sw    $11,4($3)
            addiu $6,$6,-1
            bne   $10,$0,$L15
            addiu $3,$3,-4

D           sll   $6,$6,2
            addu  $6,$7,$6
            sw    $5,0($6)
            addiu $2,$2,1
$L28:
N-1         bne   $2,$8,$L16
            addiu $4,$4,4

1           j     $31
```

Figure 1.3: Assembler listing of our C-program translated into MIPS machine code. On the left of each basic block you find the expected number of executions expressed by the variables introduced in this chapter.

> sentinel elements whose usage can improve the performance of your program, but can also slow it down—it really depends on the details.
>
> It you look at an assembler program you can estimate much better how long each instruction takes than in a high level programming language. For the older, not as sophisticated processors as today's, you could lookup in the hardware manual how many cycles each instruction takes. Today this is becoming harder and harder because the execution time depends on so many additional factors. There are, for example, one or more caches that speed up the execution of a read instruction from memory tremendously if its data value can be found in the cache. To analyze the cache behavior is not easy (although you can good data from simulations). The deep pipelining of instructions, branch prediction strategies, speculative computing, and super scalarity are further examples of moderns features that make the exact estimate of the duration of a specific maching instruction very hard. Nevertheless, the rule of thumb that one machine instruction of a RISC processor takes one cycle is still very good—that was after all one of the original design goal when RISC architectures were introduced.

Appendix B contains a short description of most MIPS instructions. You can easily find more detailed charts online.

On the other hand, most of the time we do not want to analyze an algorithm in such detail. In the rare cases that we *do* need such a precise analysis, usually the additional tedious work of looking at every machine instruction by itself takes a long time, but is still almost neglectable relative to the work that the mathematical analysis requires. After all, without a very precise mathematical analysis, counting instructions makes no sense.

Very often we do not have an implementation of an algorithm, nor do we need one for a cruder analysis. In the case of Quicksort and other sorting algorithm you will see very often only the analysis of one variable: The number of comparisons. Even this single number gives us a lot of insight. If, for example, a comparison is very expensive, then C is the dominating factor in the overall running time and we do not need the other variables. In our case—sorting numbers—this assumption does not hold.

If we count the number of executions of every single instruction in Figure 1.3 and add them together, we get the total expected number of executed machine instruction as a function of N and M:

$$I = 37A + 11B + 5C + C' + 8D + 7E + 15S + 2S'' + 7N + 14 + 2I'$$

The contribution of every variable to the running time is a constant number of machine instructions. That is not surprising since the program length itself is fixed and every instruction belongs to one (or sometimes more) of the variables. Each variable is a function of N. Only B and C grow superlinear, so only they contribute to the asymptotic running time and will dominate the other terms for large N. In practice, however, we cannot be concerned by only big N's. With our very precise analysis we can estimate the running time very precisely for *every* N.

There is also a second reason why purely asymptotic analysis are dangerous—we usually do not clearly know what *for big* N exactly means. It is the essence of asymptotic analyses that this question has to remain unanswered.

Let us turn our attention to M. This is a parameter of the algorithm and we can choose M in such a way that the running time becomes as small as possible. It is clear that the optimal choice of M also depends on N, but we expect that this dependence will be noticable only for very small N. If, however, N is very small, then Quicksort is not the right choice as a sorting algorithm and you should choose, e.g., insertion sort instead. Figure ?? shows the dependence of the running time of Quicksort for $N = 100$ in dependence of M. You can see that the primitive choice of $M = 1$ is not good at all.

Exercises

1.1 Prove that the number of executions of block $r = j - 1$ is exactly $A' - S' - \bar{S}'$.

1.2 The relationship $S' + S'' = \bar{S}' + \bar{S}''$ cannot be found by using flow relations. Nevertheless it is a sound and useful equation that helps reducing the number of independent variables. Prove that this equation indeed holds.
Hint: Consider the depth of the stack.

1.3 Complete the C-program from Figure 1.1 by adding macros *push*, *pop*, and *stack_is_empty*. The first two macros are suppossed to push two integers onto or pop them from a stack, while the third one should test whether the stack is empty

(and return 0 iff it is non-empty). Then add a main routine that calls quicksort on inputs that consist of the numbers $1, \ldots, N$ randomly permuted.

Introduce a new variable in the program that counts the number of partitioning phases. Choose a suitable value for M and establish by experiments the approximate value of A for different values of N.

1.4 Let S_N, $N \geq 1$ be a solution to the recurrence relation $S_N = \sum_{k=1}^{N} S_k/k$. All solutions form a subvector space of $\mathbf{R}^{\mathbf{N}}$, the space of real sequences. What is the dimension of this subvector space and how does a general solution look like?

1.5 Find a closed solution for $\sum_{k=M+2}^{N} \frac{k(k-2)-(k-1)(k-3)}{k(k+1)}$ by using `maxima` (or a similar system) and by doing the summation by hand.

1.6 Analyse the remaining variable S_N. First find an intuitive description behind S_N. Then construct a recurrence relation for S_N and solve it.

1.7 We have seen that $I' + I'' = 1$ and that $I', I'' \in \{0, 1\}$. We do not have to analyze their behavior in greater detail because the number of machine instruction that belong to I' and I'' is the same, so only their sum matters. If we use a highly optimizing compiler, however, it is possible that there is one machine instruction more in the I''-branch than in the I'-branch. If we strive for ludicrous precision in our analyses we cannot ignore this single instruction.

So please analyze the expected value of I'. What do think it will be? Did you guess correctly?

1.8 How many machine instructions are executed on average in Figure mips-quick1 if the program is used to sort N pairwise distinct keys in random order?

1.9 The assembler listing in Figure 1.3 contains a branch instruction in the basic block starting at label $L3. The purpose of this exercise is to analyze the penalties for wrong branch predictions on this instruction.

A commonly used branch prediction strategy is the following: The processor has two states for branch instructions, which we call YES and NO. In the state YES, the processor predicts that the branch is taken and in the state NO that it is not taken. The state is changed when two prediction in a row are wrong.

Analyze how often the branch prediction is correct. Assume that the initial state is YES. Do you expect that the prediction is good or bad for this instruction?

Do a similar analysis for the instruction `bne $12,$0,$L4` in the block after label $L5.

1.10 Extend the C-program for Quicksort with instruction that count A, B, C, D, E, and S.

Run this program once for every Permutation of the numbers $1, \ldots, 10$ and find out what $A_{10}, \ldots, S_{10}$ are. Use $M = 3$.

Compare the counted results with the predictions of our formulæ.

1.11 Write a C-program for Mergesort and analyse in the same depth as we did for Quicksort.

1.12 Consider the following algorithm to find a maximal key in an array containing natural numbers. We assume all numbers are pairwise distinct and every permutation occurs with uniform probability.

```
int maxElem(int a[], int N) {
  int i, max = -1;
  for(i = 0; i < N; i++)
    if(a[i] > max)
      max = a[i];
  return max;
}
```

How often are the instructions $a[i] > max$ and $max = a[i]$ executed on average?

1.13 The next program is presented in x86 assembler language: Again the array $ds[0] \dots ds[2*N-2]$ contains N pairwise distince natural numbers. Each permutation occurs with the same probability. How often is each instruction of this program executed on average?

`maxElem:`	`mov`	`ax, 0xFFFF`	A $ax \leftarrow -1$;
	`xor`	`dx, dx`	A $dx \leftarrow 0$;
`next:`	`cmp`	`dx, N`	B $i < N$?
	`jae`	`done`	B jump if above or equal ($i \geq N$)
	`mov`	`bx, ds:[2*dx]`	C $bx \leftarrow a[dx]$
	`cmp`	`bx, max`	C $bx > max$?
	`jna`	`skip`	C jump if not above ($bx \leq N$)
	`mov`	`ax, bx`	D $ax \leftarrow bx$
`skip:`	`add`	`dx, 0x0002`	E $ax \leftarrow ax + 1$;
	`jmp`	`next`	E jump
`done:`	`push`	`ax`	F push the maximum on the stack

1.14 Student party! DJ O*D*D is present and brought with him infinitely many songs in the three genres Rock, Gabba, and Blues. Tonight he will play n songs, so there are theoretically 3^n different combinations of genres possible. He has, howver, to obey some strange rules:

1. After a rock song, he cannot play Gabba because readjusting the equalizer takes too much time.
2. You cannot play two Gabba songs in sequence because it causes visitors to die of accelerated stupification.
3. If he plays a Blues song, he has to stick to Blues for the remaining time because everybody is feeling blue.

Set up a recurrence for the number of genre combination and solve it.

1.15 We have an array a of length N. It contains N numbers drawn independently and uniformly at random from $\{1, \ldots, N\}$. How often is each instruction of the following program executed on average?

```
count = 0;
i = 1;
while(i ≤ N)
  if(a[i]%2 == 1)
    count++;
  i++;
return count;
```

1.16 Let $w \in \{a, b\}^n$ a word that has been chosen uniformly at random. How often is the body of the **while**-loop executed on average in the following algorithm? The function *is_palindrome* tests whether a word in a palindrome, i.e., the same when read backwards.

```
i = 2;
while(i ≤ n)
  if(is_palindrome(w[1], ..., w[i]))
    return true;
  i++;
return false;
```

1.17 Two natural numbers $m \neq n$ are *friendly*, if the sum of all proper divisors of m is n—and vice versa. A son and his father wrote these two programs that compute friendly numbers. What are the running times of both programs?

Son

```
#include <iostream>
int e[150000];
int echteil(int a) {
  int n = 0;
  for(int i = 1; i + i ≤ a; i++)
    if(a%i == 0) n += i;
  e[a] = n;
  return n;
}
main() {
  for(int i = 0; i < 150000; i++) {
    int a = echteil(i);
    if(a ≥ i) continue;
    if(e[a] == i) std :: cout << i
       << " " << echteil(i) << "n n";
  }
}
```

Father

```
#include <stdio.h>
#define N 1000000
int teilersumme[N];
int main() {
  int i;
  for(i = 1; i < N; i++) {
    int p = i;
    while(p < N) {
      teilersumme[p] += i;
      p += i;
    }
  }
  for(i = 1; i < N; i++) {
    int a = teilersumme[i] − i;
    if(a < i && i == teilersumme[a] − a)
      printf("%d %dn n", a, i);
  }
  return 0;
}
```

Chapter 2

The Kirchhoff laws

When we analysed quicksort, we learned several methods who to reduce the numbers of variables that have to be analysed. A general technique to do so, which we will develop formally now, uses *Kirchhoff's laws* from Electrical Engineering. We begin by looking at a directed graph whose notes are the instructions of our program.

There is an edge between two nodes if and only if the second instruction follows directly behind the first one. In the case of a branch instruction more than one edge will emerge from a note. It is also possible that there is more than one edge that leads into a note because this note could be the goal of several branch instructions.

We also assume that there is a special note which we will call START and another one which is denoted by STOP. The program flow starts at the START note and ends at the STOP note. Let us assume, the graph has exactly n notes including START and STOP and m edges. We denote the edges by e_i by $i = 1, \dots, m$.

For symmetry reasons we add another edge called e_0 that goes from STOP to START. With E_i we denote the number of times that e_i is used in a program run. We set $E_0 = 1$ as if the program will return to its start after terminating. All together we have m different variables E_i. It will turn out that there are not all independent of each other but are subject to several equations. These equations are derived from Kirchhoff's law:

Theorem 1. ***(Kirchhoff's Law)***
Let I be the set of all i for which the edge e_i ends in some node X and let

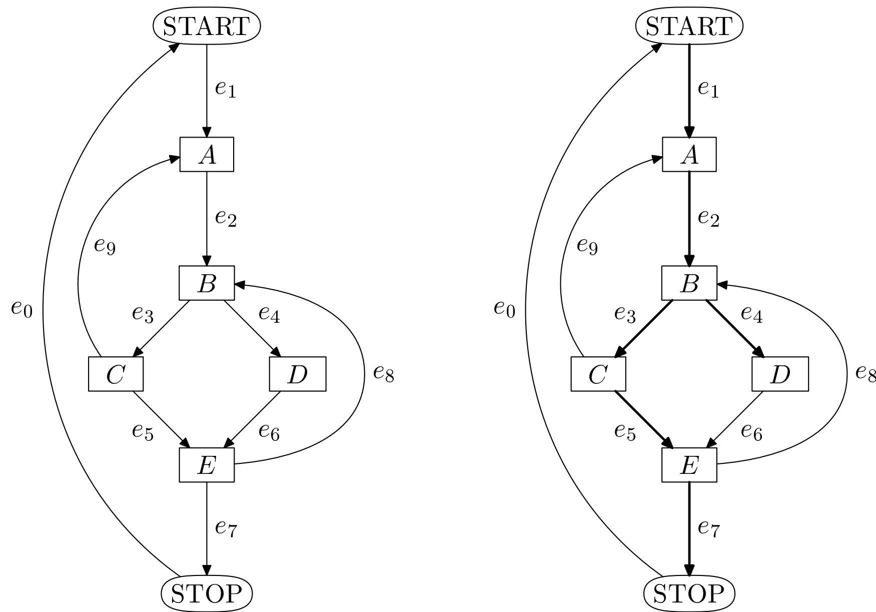

Figure 2.1: Example of a flow diagram with and without a spanning tree.

O be the set of all i for which e_i emerges from X. Then the sums

$$\sum_{i \in I} E_i = \sum_{i \in O} E_i$$

are identical and the corresponding number expresses how often the struction X is executed all together.

In the following we will develop a method which lets us choose a subset of the set of independent variables E_i such that we can derive the value of all other variables from them.

The first step is to choose a *spanning tree* for the undirected graph. In this step we ignore that edges are directed. Figure ?? contains a simple example and a spanning tree depicted by drawing its edges thicker. The spanning tree consists of the edges e_1, e_2, e_3, e_4, e_5, and e_7.

If we add any other edge to this spanning tree, then we get a unique cycle. We denote these cycles as *fundamental cycles*. In our example the edges e_0, e_6, and e_8 create such fundamental cycles. We provide each edge of a fundamental cycle with a label: "+", if the direction of this edge is the same as the direction of the unique edge in the cycle which does not belong to the spanning tree. Otherwise, we use the label "−".

In our example we have the following fundamental cycles:

$$\begin{aligned} C_0 &= e_0 + e_1 + e_2 + e_3 + e_5 + e_7 \\ C_6 &= e_6 - e_5 - e_3 + e_4 \\ C_8 &= e_8 + e_3 + e_5 \\ C_9 &= e_9 + e_2 + e_3 \end{aligned}$$

An interesting fact which is what makes this definition interesting for the analysis of algorithm, is that every fundamental cycle delivers a solution of Kirchhoff's laws: We set all $E_i = 0$ for which e_i is not part of the fundamental cycle. If on the other hand e_i belongs to the fundamental cycle, then we set $E_i = 1$ or $E_i = -1$, according to the label of e_i in the fundamental cycle.

In our example the four corresponding solutions look as follows:

1. $E_0 = 1$, $E_1 = 1$, $E_2 = 1$, $E_3 = 1$, $E_4 = 0$, $E_5 = 1$, $E_6 = 0$, $E_7 = 1$, $E_8 = 0$, $E_9 = 0$

2. $E_0 = 0$, $E_1 = 0$, $E_2 = 0$, $E_3 = -1$, $E_4 = 1$, $E_5 = -1$, $E_6 = 1$, $E_7 = 0$, $E_8 = 0$, $E_9 = 0$

3. $E_0 = 0$, $E_1 = 0$, $E_2 = 0$, $E_3 = 1$, $E_4 = 0$, $E_5 = 1$, $E_6 = 0$, $E_7 = 0$, $E_8 = 1$, $E_9 = 0$

4. $E_0 = 0$, $E_1 = 0$, $E_2 = 1$, $E_3 = 1$, $E_4 = 0$, $E_5 = 0$, $E_6 = 0$, $E_7 = 0$, $E_8 = 0$, $E_9 = 1$

So far we have four different solutions. The underlying equations are linear. Therefore, linear combinations of their solutions are again solutions. Using vector notation we can write the linear combinations of our four solutions as follows:

$$\vec{E} = \begin{pmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \\ E_8 \\ E_9 \end{pmatrix} = \lambda_1 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ -1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \lambda_4 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix} \quad (2.1)$$

For every combination of $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \in \mathbf{R}$ we get one solution of Kirchhoff's laws and *every* solution can be derived in this way.

At this point we can also notice that $E_0 = E_1 = E_7$ and $E_4 = E_6$, because the rows of the matrix are identical for them.

At this point it is easy to choose three linearly independent E_i and analyse only them. Then all other E_i can be expressed by them. In our example we choose E_0 because we already know that $E_0 = 1$. We have to choose three more. Let us assume, we choose $A = E_2$, $C = E_3$ and $D = E_4$. For this choice we get the following equation:

$$\begin{pmatrix} 1 \\ A \\ C \\ D \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix}$$

This is a linear system of equations that can be solved with the usual methods. Here we get the result $\lambda_1 = 1$, $\lambda_2 = D$, $\lambda_3 = -A + C + D$, and $\lambda_4 = A - 1$. If we insert these in (2.1) then we get the solution of all other E_i:

$$\vec{E} = \begin{pmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \\ E_8 \\ E_9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ D \\ C + D - A \\ A - 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ A \\ C \\ D \\ 1 + C - A \\ D \\ 1 \\ C + D - A \\ A - 1 \end{pmatrix}$$

All these computations can be done by computer algebra systems like Mathematica, Maple or Macsyma. In general the matrizes can become quite big.

We chose A, C, and D as the variables that we wanted to analyse. Which of the variables are chosen for this purpose depends on the concrete problem. It remains to get C and E: In this case we can express them as $B = E_3 + E_4 = C + D$ and $E = E_5 + E_6 = 1 + C + D - A$.

You can find a deeper exposition to this technique Knuth [3, Section 2.3.4.1].

Exercises

2.1 If a flow diagram consists of n nodes and m edges, how many fundamental cycles do we get?

2.2 Prove or disprove: In every flow diagram you can find a spanning tree such that all fundamental cycles contain only edges that are labeled with plus.

2.3 In dieser Aufgabe betrachten wir den Algorithmus von Prim, mit dessen Let us look at the algorithms of Prim that is used to compute minimal spanning trees in a connected weighted Graph.The input consists of an undirected graph $G = (V, E)$, and a weight function $w : E \to \mathbf{R}$, and a starting node r.

```
 1  for each u ∈ V do
 2        key[u] ← ∞
 3        π[u] ← NIL
 4  key[r] ← 0
 5  M ← V
 6  while (M ≠ ∅) do
 7        u ← min-from(M)
 8        for each v ∈ neighbors(u) do
 9              if (v ∈ M) ∧ (w(u,v) < key[v]) then
10                    π[v] ← u
11                    key[v] ← w(u,v)
```

Draw a flow diagram for this algorithms that contains all blocks. Construct a spanning tree and a corresponding fundamental cycles. Choose a minimal set of blogs whose running time can be analysed, and explain how you can derive all other variables from them.

Chapter 3

Recurrence relations

If you analyse the running time or some other parameter of an algorithm, you want to find a closed mathematical formula that describes the parameter you are analysing. More often than not you will not be able to find such a formula right away, but only some related formula that describes the parameter you are insterested in in an indirect way. When we analysed quicksort as an example we could describe an interesting parameter X_N by formulas that looked like

$$X_N = \frac{2}{N}\sum_{k=0}^{N-1} X_k + f_N.$$

An equation that contains not only the variables X_N but also variables X_k with $k < N$ are called *recurrences.*

To *solve* a recurrence relation means to find a closed formula for X_N.

In general no closed formula for the solution of a recurrence relation needs to exist—in practice, on the other hand, the analysis of algorithms very often leads to recurrence relations that indeed have a closed solution, or, whose solution can be at least very well approximated by some nice closed formula. There are also some classes of recurrence relations that can be solved by some easy fixed algorithm. Much of the material in this chapter can be found in three books [1, 2, 3], in particular in the second one by Greene and Knuth.

3.1 Classification of recurrence relations

The most general recurrence relation, which we consider, has the general form

$$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_0) \text{ for } n \geq t. \tag{3.1}$$

We consider a_n only for $n \geq 0$ and define $a_{-1} = a_{-2} = a_{-3} = \cdots = 0$.

Because (3.1) holds only for $n \geq t$, we can compute any a_n if $a_0, a_1, \ldots, a_{t-1}$ are already known. In general the solution of the recurrence relations will depend on these starting values. If the recurrence relation originates from the analysis of an algorithm, then the starting values $a_0, a_1, \ldots, a_{t-1}$ are usually fixed by the algorithm.

The recurrence relation for the number of comparisons during partitioning for the Quicksort algorithm was

$$C_N = N + 1 + \frac{2}{N} \sum_{k=0}^{N-1} C_k \text{ for } N > M$$

with the starting conditions $C_0 = C_1 = C_2 = \ldots = C_M = 0$.

We can derive these starting conditions easily from the algorithm: If $N \leq M$ then no partitioning takes place.

In general we classify recurrence relations as follows:

$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_{n-t})$	Recurrence relation of t-th order
$a_n = \sum_{k=0}^{n-1} x(k,n) a_k$	homogeneous, linear recurrence relation
$a_n = \sum_{k=0}^{n-1} x(k,n) a_k + f(n)$	linear recurrence relation
$a_n = x_1 a_{n-1} + x_2 a_{n-2} + \cdots + x_t a_{n-t}$	linear with constant coefficients

In this chapter we will look at various methods to solve typical recurrence relations that originate from the analysis of algorithms.

3.2 Creating a table

Usually a first step that we should always take is to compute some values of the solution of the recurrence relation and put them into a small table. Let us look for example at the recurrence relation

$$a_n = a_{n-1} + 2a_{n-2} \text{ for } n > 1 \text{ and } a_0 = a_1 = 1.$$

We can compute $a_2 = a_1 + 2a_0 = 3$, $a_3 = a_2 + 2a_1 = 5$, $a_4 = a_3 + 2a_2 = 11$, $a_5 = 21$, $a_6 = 43$. and get the following table:

n	0	1	2	3	4	5	6
a_n	1	1	3	5	11	21	43

By looking at the table we get a first impression how the solution looks like and we can reuse the table later to see whether our closed formula is correct. If the first values of the table coincide with the values predicted from our solution we can be reassured that we have not made any mistakes when finding the closed formula.

3.3 Guessing a solution and proving it by induction

With the help of some solutions from our short table we can try to guess a closed formula. Let us look for example at the table above for a_n. If we look at it it seems that the sequence consists of numbers that almost double in each step. It seems that there are not exactly doubling, but sometimes they are twice the predecessor plus one and sometimes minus one. If this is true, a good idea might be to look at the sum of two consequent values of a_n. We get 2, 4, 8, 16, 32, 64.

This suggests that the solution should be approximately $2^{n+1}/3$. So let us look at a table of $2^{n+1}/3$:

n	0	1	2	3	4	5	6
$\frac{1}{3}2^{n+1}$	$\frac{2}{3}$	$1\frac{1}{3}$	$2\frac{2}{3}$	$5\frac{1}{3}$	$10\frac{2}{3}$	$21\frac{1}{3}$	$42\frac{2}{3}$

Indeed it seems that the values in this table are almost the solution but they are alternatingly $\frac{1}{3}$ too small or too big. This suggest a closed formula as follows:

$$a_n = \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^n$$

Let us verify this formula on some values. If $n = 0$ we get $\frac{1}{3}2 + \frac{1}{3}(-1)^0 = 1$, for $n = 1$ we get $\frac{1}{3}4 + \frac{1}{3}(-1)^1 = 1$, and finally for $n = 5$ we get $\frac{1}{3}64 + \frac{1}{3}(-1)^5 = \frac{63}{3} = 21$.

It seems that our guess was correct but we still have to prove its correctness. Usually induction is the best method to prove such a claim. We already showed that the closed formula is correct for $n = 0$ and $n = 1$. So let us assume now that $n > 1$.

From the induction hypothesis we get

$$\begin{aligned} a_n = a_{n-1} + 2a_{n-2} &= \frac{1}{3}2^n + \frac{1}{3}(-1)^{n-1} + \frac{2}{3}2^{n-1} + \frac{2}{3}(-1)^{n-2} \\ &= \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^{n-2} \end{aligned}$$

and this coincides with our closed formula for a_n because $(-1)^{n-2} = (-1)^n$. This proves without doubt that indeed $a_n = \frac{1}{3}2^{n+1} + \frac{1}{3}(-1)^n$.

3.4 Looking up the solution

There is a very interesting book that contains most known integer sequences in lexicographical order. Of course, only the beginning of each sequence is listed together with a short description and pointers to places this series was used. You can find our series $1, 1, 3, 5, 11, 21, 43, \ldots$ in this book. There it has the name "$A(N) = A(N-1) + A(N-2)$" and there are pointers to two papers in the journal *Eureka, the Journal of the Archimedeans* (Cambridge University Mathematical Society) and *Nouvelles Correspondance Mathématique*. You can find more about this series in those two publications.

Meanwhile in the mordern world of the WWW there is an alternative that you can find under the URL

`https://oeis.org`

At this webpage you can enter the beginning of your series and the answer to the input 1, 1, 3, 5, 11, 21, 43 is depicted in Figure 3.1. The web page also reveals a name of our series: *Jacobstahl sequence*. Moreover you find more pointers to literature and also a closed formula for the term a_n.

3.5 Mathematica, Maple, Maxima, etc.

There are some computer algebra systems that are able to solve simple recurrence relations directly. For the system Mathematica the corresponding function is named RSolve. We can use Mathematica to solve our example problem:

```
RSolve[{a[n] == a[n - 1] + 2 a[n - 2],
  a[0] == a[1] == 1}, a[n],
 n]
```

$$\left\{\left\{a(n) \to \frac{(-1)^n}{3} + \frac{2^{n+1}}{3}\right\}\right\}$$

Mathematica finds the same solution as we did. The other well-known algebra system Maple can solve the recurrence, too:

```
> rsolve({a(n) = a(n-1)+2*a(n-2), a(0..1)=1}, a(n));
                                          n        n
                                 1/3 (-1)  + 2/3 2
>
```

The free computer algebra system *maxima* is also able to solve such a simple recurrence:

```
Maxima 5.23.2 http://maxima.sourceforge.net
using Lisp SBCL 1.0.38-3.el6
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
```

login

This site is supported by donations to The OEIS Foundation.

THE ON–LINE ENCYCLOPEDIA OF INTEGER SEQUENCES®

founded in 1964 by N. J. A. Sloane

Annual appeal: Please make a donation to keep the OEIS running! Over 6000 articles have referenced us, often saying "we discovered this result with the help of the OEIS".

Other ways to donate

Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

A001045 Jacobsthal sequence (or Jacobsthal numbers): a(n) = a(n-1) + 2*a(n-2), with a(0) = 0, a(1) = 1. 575
(Formerly M2482 N0983)

0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683, 1365, 2731, 5461, 10923, 21845, 43691, 87381, 174763, 349525, 699051, 1398101, 2796203, 5592405, 11184811, 22369621, 44739243, 89478485, 178956971, 357913941, 715827883, 1431655765, 2863311531, 5726623061 (list; table; graph; refs; listen; history; text; internal format)

OFFSET 0,4

COMMENTS

```
Number of ways to tile a 3 X (n-1) rectangle with 1 X 1 and 2 X 2 square tiles.
Also, number of ways to tile a 2 X (n-1) rectangle with 1 X 2 dominoes and 2 X 2
  squares. - Toby Gottfried, Nov 02 2008
Also a(n) counts each of the following four things: n-ary quasigroups of order 3
  with automorphism group of order 3, n-ary quasigroups of order 3 with
  automorphism group of order 6, (n-1)-ary quasigroups of order 3 with
  automorphism group of order 2 and (n-2)-ary quasigroups of order 3. See the
  McKay-Wanless (2008) paper. - Ian Wanless, Apr 28 2008
Also the number of ways to tie a necktie using n + 2 turns. So three turns make an
  "oriental", four make a "four in hand" and for 5 turns there are 3 methods:
  "Kelvin", "Nicky" and "Pratt". The formula also arises from a special random
  walk on a triangular grid with side conditions (see Fink and Mao, 1999). -
  arne.ring(AT)epost.de, Mar 18 2001
Also the number of compositions of n + 1 ending with an odd part (a(2) = 3 because
  3, 21, 111 are the only compositions of 3 ending with an odd part). Also the
  number of compositions of n + 2 ending with an even part (a(2) = 3 because 4,
  22, 112 are the only compositions of 4 ending with an even part). - Emeric
  Deutsch, May 08 2001
Arises in study of sorting by merge insertions and in analysis of a method for
  computing GCDs - see Knuth reference.
Number of perfect matchings of a 2 X n grid upon replacing unit squares with
  tetrahedra (C_4 to K_4):
o----o----o----o...
| \/ | \/ | \/ |
| /\ | /\ | /\ |
o----o----o----o... - Roberto E. Martinez II, Jan 07 2002
Also the numerators of the reduced fractions in the alternating sum 1/2 - 1/4 + 1/8
  - 1/16 + 1/32 - 1/64 + ... - Joshua Zucker, Feb 07 2002
Also, if A(n), B(n), C(n) are the angles of the n-orthic triangle of ABC then A(1)
  = Pi - 2A, A(n) = s(n)*Pi + (-2)^n*A where s(n) = (-1)^(n-1) * a(n) [1-orthic
  triangle = the orthic triangle of ABC, n-orthic triangle = the orthic triangle
  of the (n-1)-orthic triangle]. - Antreas P. Hatzipolakis
  (xpolakis(AT)otenet.gr), Jun 05 2002
Also the number of words of length n+1 in the two letters s and t that reduce to
  the identity 1 by using the relations sss = 1, tt = 1 and stst = 1. The
  generators s and t and the three stated relations generate the group S3. - John
```


Figure 3.1: Erste Seite der Antwort der *On-Line Encyclopedia of Integer Sequences* auf die Eingabe 1,1,3,5,11,21,43.

```
(%i1) load("solve_rec");
(%o1)   /usr/share/maxima/5.23.2/share/contrib/solve_rec/solve_rec.mac
(%i2) solve_rec(a[n]-a[n-1]-2*a[n-2], a[n], a[0]=1, a[1]=1);
                                                n + 1        n
                                               2        (- 1)
(%o2)                                    a  = ------ + ------
                                          n     3        3
```

3.6 Hidden products and sums

The most simple recurrence relations are of the form

$$a_n = x_n a_{n-1} \text{ and } b_n = b_{n-1} + y_n.$$

Both forms are related to each other. If you substitute $\bar{a}_n = \log(a_n)$ then the left recurrence relation turns into a recurrence relation of the right hand type. We will call these two types of recurrence relations *hidden products* and *hidden sums*.

The recurrence relation on the left hand side can be solved by repeatedly inserting the right hand side. The procedure leads to a product:

$$a_n = x_n a_{n-1} = x_n x_{n-1} a_{n-2} = \cdots = x_n x_{n-1} x_{n-2} x_{n-3} \cdots x_2 x_1 a_0 = a_0 \prod_{k=1}^{n} x_n.$$

In the same way iteratively inserting leads to a sum for the recurrence relation on the right hand side.

Theorem 2. The solutions of the recurrence relations

$$a_n = x_n a_{n-1} \qquad \text{and} \qquad b_n = b_{n-1} + y_n$$

are

$$a_n = a_0 \prod_{k=1}^{n} x_n \qquad \text{and} \qquad b_n = b_0 + \sum_{k=1}^{n} y_n.$$

3.7 Linear recurrence relations with constant coefficients

A very simple recurrence form are homogeneous linear recurrence relations with constant coefficients. In the most general form they look like

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \text{ for } n \geq t \tag{3.2}$$

Here we have a recurrence relation of t-th order. The coefficients $c_i \in \mathbf{R}$ are the coefficients of the recurrence relations and do not depend on n (therefore *constant* coefficients.)

Linear recurrence relations with constant coefficients can always be solved and additionally they can be solved with a fixed algorithm. In the following we will develop such a general algorithm that solves these kind of recurrences.

Let us first assume that there exists a solution of the form $a_n = \alpha^n$ where $\alpha \in \mathbf{C}$. If we insert this solution into the recurrence and set $n = t$ then we get

$$\alpha^t = c_1\alpha^{t-1} + c_2\alpha^{t-2} + \cdots + c_{t-1}\alpha + c_t.$$

Such a solution implies that α is a root *characteristic polynomial.*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1}z - c_t.$$

On the other hand it is also clear that $a_n = \alpha^n$ is indeed a solution to (3.2) if α is a root of the characteristic polynomial.

If α happens to be a root of χ with multiplicity k then additionally $a_n = n^j\alpha^n$ for $0 \leq j < k$ are solutions to the recurrence relations. We can check this fact by inserting the solution into the recurrence:

$$n^j\alpha^n = \sum_{r=1}^{t} c_r(n-r)^j\alpha^{n-r},$$

which is equivalent to

$$n^j\alpha^t - \sum_{r=1}^{t} c_r(n-r)^j\alpha^{t-r} = 0.$$

The left hand side of the above equation is a linear combination of $\chi(\alpha)$, $\chi'(\alpha)$, $\chi''(\alpha)$, $\ldots$, $\chi^{(j)}(\alpha)$. The first k derivatives of χ are 0 at α because α is a root of χ with multiplicity k.

Theorem 3. The homogeneous linear recurrence relation with constant coefficients

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \text{ for } n \geq t$$

has the solutions $a_n = n^j \alpha^n$ for all roots α of the characteristic polynomial

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

and for all $j = 0, 1, \ldots, k-1$ where k is the multiplicity of the root α. All these solutions are linearly independent. They form a basis of the vector space of all solutions.

Because the recurrence relation is linear and homogeneous multitudes of a solution and sums of solutions are again solutions of the recurrence. In that way we have constructed exactly t linearly independent solutions and we noted that they are a basis of the vector space of all solutions.

If we have t initial conditions, for example the values of a_0, $a_1, \ldots$, a_{t-1}, then we get exactly one solution by a linear combination of the solutions in our basis. To find the right linear combination we just have to solve a linear system of equations.

Let us look at

$$a_n = a_{n-1} + 2a_{n-2} \text{ für } n > 1 \text{ und } a_0 = a_1 = 1.$$

The characteristic polynomial is $q(z) = z^2 - z - 2$. We can see immediately that -1 is a root. Using polynomial division of $q(z)$ by $z+1$ we get the result $z-2$ and a second root is 2. All solutions are therefore of the form

$$a_n = \lambda 2^n + \mu(-1)^n.$$

To establish the values of the constants λ and μ we have to use the initial conditions. If we insert the initial conditions into the recurrence relation we get $1 = \lambda + \mu$ and $1 = 2\lambda - \mu$. Solving this system of equations yields $\lambda = \frac{1}{3}$ and $\mu = \frac{2}{3}$.

3.8 Summation factor

We can always convert a linear recurrence relation of first order into a sum. We used this technique already when solving the recurrence relations for

Quicksort. After we turned them into a first order recurrence relation they took the following form:

$$NX_N = (N+1)X_{N-1} + Nf_N - (N-1)f_{N-1}, \text{ for } N > M+1$$

We multiplied this equation with $1/N(N+1)$ and finally got after a substitution the very simple equation of the form

$$Y_N = Y_{N-1} + g_N.$$

In the following we will develop a technique that allows us to do a similar transformation with all linear recurrence relations of first order.

Theorem 4. The linear recurrence relation of first order

$$a_n = x_n a_{n-1} + y_n \text{ for } n > 0$$

and $a_0 = 0$ has the solution

$$a_n = y_n + \sum_{j=1}^{n-1} y_j x_{j+1} x_{j+2} \dots x_n.$$

We prove this theorem by dividing the recurrence relation by $x_n x_{n-1} x_{n-2} \dots x_1$, which gives us

$$\frac{a_n}{x_n x_{n-1} x_{n-2} \dots x_1} = \frac{a_{n-1}}{x_{n-1} x_{n-2} x_{n-3} \dots x_1} + \frac{y_n}{x_n x_{n-1} x_{n-2} \dots x_1}.$$

If we substitute $b_n = a_n/(x_n x_{n-1} x_{n-2} \dots x_1)$ we get the simpler recurrence relation

$$b_n = b_{n-1} + \frac{y_n}{x_n x_{n-1} x_{n-2} \dots x_1}$$

that we can easily solve by a summation. In this method we call the product $1/x_n x_{n-1} x_{n-2} \dots x_1$ a *summation factor.* It gives this method its name. Very often the summation factor is quite simple because a lot of cancellation goes on.

Let us try to apply the technique of summation factors on the reccurence relation

$$a_n = 2a_{n-1} + n \text{ for } n > 0$$

and $a_0 = 0$. In this case $x_n = 2$ and $y_n = n$. Therefore the solution is

$$a_n = n + \sum_{j=1}^{n-1} j \cdot 2^{n-j} = 2^{n+1} - 2 - n.$$

Indeed Mathematica can solve this recurrence relation, too:

```
In[4]:= RSolve[{a[0]==0, a[n]==2a[n-1]+n}, a[n], n]
                                1 + n
Out[4]= {{a[n] -> -2 + 2             - n}}
```

3.9 The Repertoire Method

We can use the repertoire method mainly for linear recurrence relations. This method shows that solving recurrence relations is more art than science. To master the repertoire method we need a lot of intuition. When analysing algorithms, usually we know how the solution will roughly look like. In general when applying the repertoire method we start from a recurrence relation of the form

$$a_n = x_{1,n} a_{n-1} + x_{2,n} a_{n-2} + x_{3,n} a_{n-3} + \ldots + x_{t,n} a_{n-t} + f_n.$$

At this point we imagine some solution of the equation and find out for which f_n we get this solution. We do the same for many different potential solutions and each time we get a different f_n. Because linear combinations of solutions are again solutions of a recurrence relation we can get the solution of the original recurrence relation by forming a linear combination of our potential solutions such that the corresponding linear combination of the different f_n's yields the original f_n of the orginial recurrence relation.
We demonstrate the repertoire method on Quicksort (with $M = 0$):

$$a_n = f_n + \frac{2}{n} \sum_{k=0}^{n-1} a_k$$

We start with a potential solution $a_n = 1$ and get

$$f_n = a_n - \frac{2}{n} \sum_{k=0}^{n-1} a_k = 1 - \frac{2}{n} \sum_{k=0}^{n-1} 1 = -1.$$

This means that our guessed solution $a_n = 1$ is indeed correct if $f_n = -1$. But the real f_n is different. We proceed by trying other potential solutions, computing the corresponding f_n and see what linear combinations of these f_n's we can get.

For our Quicksort equation we will choose the following repertoire:

a_n	$f_n = a_n - \frac{2}{n}\sum_{k=0}^{n-1} a_k$	a_0
1	-1	1
H_n	$2 - H_n$	0
nH_n	$\frac{1}{2}(n-1) + H_n$	0

Let us assume we want to analyse the number of comparisons. In that case $f_n = n + 1$. Not suprisingly we don't have a solution for this specific f_n in our repertoire. On the other hand, we can get $n + 1$ as a linear combation of the three f_n's which are contained in our repertoire:

$$n + 1 = 2\left(\frac{1}{2}(n-1) + H_n\right) + 2\,(2 - H_n) + 2\,(-1)$$

Consequently, we can get a solution for the recurrence relation with $f_n = n + 1$ as

$$a_n = 2\,(nH_n) + 2\,(H_n) + 2\,(1) = 2(n+1)H_n + 2.$$

While we have a solution now, unfortunately, the starting condition $a_0 = 0$ is not fulfilled. Instead we get $a_0 = 2$. To remedy this situation we need a bigger repertoire so that the linear combinations do not yield only the correct f_n but also the correct starting condition. For this end we add another function to our repertoire:

a_n	$f_n = a_n - \frac{2}{n}\sum_{k=0}^{n-1} a_k$	a_0
1	-1	1
H_n	$2 - H_n$	0
nH_n	$\frac{1}{2}(n-1) + H_n$	0
n	1	0

Now we can express $n + 1$ as a linear combination of $2 - H_n$, $\frac{1}{2}(n-1) + H_n$ and 1 and get a solution with the correct starting condition because in all

these cases $a_0 = 0$ holds:

$$n + 1 = 2\left(\frac{1}{2}(n-1) + H_n\right) + 2\,(2 - H_n) - 2\,(1)$$

and the solution is

$$2(nH_n) + 2(H_n) - 2(n) = 2nH_n + 2H_n - 2n.$$

The general procedure when using the repertoire method is as follows: we start with a recurrence relation of the form

$$a_n = \sum_{i=1}^{t} x_{i,n} a_{n-t} + f(n).$$

The coefficients $x_{i,n}$ may depend on n.

Step 1: We choose a repertoire b_n, c_n, d_n,... of different series and compute for each of them $f_b(n) = b_n - \sum_{i=1}^{t} x_{i,n} b_{n-t}$. In this way b_n is a closed solution of the recurrence relation

$$b_n = \sum_{i=1}^{t} x_{i,n} b_{n-1} + f_b(n) \text{ for } n \geq t$$

Step 2: If we can express $f(n)$ as a linear combination of $f_b(n)$, $f_c(n)$,..., let us say as

$$f(n) = \beta f_b(n) + \gamma f_c(n) + \delta f_d(n) + \cdots$$

then we get

$$a_n = \beta b_n + \gamma c_n + \delta d_n + \cdots$$

and he have a solution of the recurrence relation with some specific starting conditions.

Step 3: If we want to find a solution for a_n for different starting conditions, which is usually the case, then we have to use a different linear combination. For this end the repertoire must be big enough in order to have as many linearly independent solutions for a_n such that we can enforce the correct starting conditions by some linear combination of the solutions.

3.10 Order Reduction

Sometimes we can reduce a recurrence relation of higher order to several recurrence relations of smaller order. For this end we define the so-called *shift operator* E, which maps sequences to sequences. This operator is defined via

$$Ef_n = f_{n+1},$$

which means that this operator shifts all elements in a sequence by one position to the beginning. We can interpret the expression Ef_n in two different ways: If we interpret f_n as a sequence, then $Ef_n = f_{n+1}$ is simply the shifted sequence. A second possibility is to interpret f_n as an operator, too, which gives us $f_n g_n$ if applied to the sequence g_n. Then $Ef_n = f_{n+1}E$.

To work with linear operators can be counterintuitive in the beginning. While the associative law is still valid (for example $E(f_n g_n) = (Ef_n)g_n$), which means that we don't have to care about the setting of parenthesis, the commutative law certainly is invalid. For example $En = (n+1)E$ (n is interpreted here as a series whose n's element is n). Similarly $En^2 = (n^2 - 2n + 1)E$.

It is always possible to write a linear recurrence relation of t's order as follows:

$$p(E)a_n = f(n),$$

where p is a polynomial of degree t whose coefficients are themselves sequences because

$$a_n = x_{1,n}a_{n-1} + x_{2,n}a_{n-2} + \cdots + x_{t,n}a_{n-t} + f(n),$$

which is equivalent to

$$(E^t - x_{1,n}E^{t-1} - x_{2,n}E^{t-2} - x_{3,n}E^{t-3} - \cdots - x_{t,n}E^0)a_{n-t} = f(n).$$

While it is always possible to factor polynomials whose coefficients are complex numbers, for polynomials whose coefficients are sequences, this is not always the case. If we are lucky, however, we can write $p(E) = q(E)r(E)$. In that way, it is sometimes possible to factor a polynomial $p(E)$. If we indeed succeed in factoring the polynomial, we still have to solve $q(E)r(E)a_n = f(n)$. We start with the substitution $b_n = r(E)a_n$ and first solve the recurrence relation

$$q(E)b_n = f(n),$$

which is a recurrence relation of lower order than the original one. In that way, we can get a closed formular for b_n. In the next step we solve the recurrence

$$r(E)a_n = b_n,$$

which is again a recurrence relation of lower order and after solving it we get the solution for a_n.

In general we use the following recipe:

First you write a linear recurrence relation in operator form as

$$p(E)a_n = f(n)$$

and try to factor it as $p(E) = q(E)r(E)$. In the next step you solve

$$q(E)b_n = f(n) \text{ and } r(E)a_n = b_n.$$

The recurrence relation is in that way reduced to two recurrence relations of smaller order.

As an example, let us again look at the recurrence relation

$$a_{n+2} - (n+2)a_{n+1} + na_n = n. \tag{3.3}$$

In operator notation this recurrence relation looks as follows:

$$(E^2 - (n+2)E + n)a_n = n$$

We can indeed factor this polynomial because $(E-1)(E-n) = E^2 - (n+2)E + n$. Please note that $En = (n+1)E$. The recurrence relation has now the form

$$(E-1)(E-n)a_n = n$$

and we start by solving $(E-1)b_n = n$ or, equivalently $b_{n+1} = b_n + n$. This is a hidden sum and we get the solution with the help of Theorem 2 resulting in

$$b_n = \sum_{k=0}^{n}(n-1) = \frac{n(n-1)}{2} + b_0 = \frac{n(n-1)}{2} + a_1.$$

In the final step we have to solve $(E-n)a_n = b_n = n(n-1)/2 - a_1$, which is a recurrence relation that can also be written as

$$a_{n+1} = na_n + \frac{n(n-1)}{2} + a_1. \tag{3.4}$$

This is a linear recurrence of first order. Solving it yields

$$a_n = \frac{(n-1)!}{2}\left(\sum_{k=1}^{n-1}\frac{k^2-k+2a_1}{k!}\right) + a_1(n-1)!. \tag{3.5}$$

As usual the result is in the form of a summation. Let us take a closer look at the interesting part inside the big parentheses:

$$\begin{aligned}
\sum_{k=1}^{n-1}\frac{k(k-1)+2a_1}{k!} &= \sum_{k=2}^{n-1}\frac{1}{(k-2)!} + \sum_{k=1}^{\infty}\frac{2a_1}{k!} = \sum_{k=0}^{n-3}\frac{1}{k!} + 2a_1\sum_{k=1}^{n-1}\frac{1}{k!} \\
&= \sum_{k=0}^{n-3}\frac{1}{k!} + 2a_1\Big(\sum_{k=0}^{n-1}\frac{1}{k!} - 1\Big) = e - \sum_{k=n-2}^{\infty}\frac{1}{k!} + 2a_1\Big(e - \sum_{k=n}^{\infty}\frac{1}{k!} - 1\Big) \\
&= e - O(1/(n-2)!) + 2a_1(e - O(1/n!) - 1) = 2a_1(e-1) + e + O(1/(n-2)!)
\end{aligned}$$

Inserting the result into (3.5) gives us an asymptotic estimate of a_n:

$$\begin{aligned}
a_n &= \frac{(n-1)!}{2}\Big(2a_1(e-1) + e + O(1/(n-2)!)\Big) + a_1(n-1)! \\
&= (n-1)!(a_1 e + e/2) + O(n)
\end{aligned}$$

Let us choose a_1 as the starting condition. Then $a_{10} = 1479610$ and the estimate is $1479615.164\ldots$, which is about 5 too high, but still very close.

3.11 Extracting recurrence relations from algorithms

Let us look at the following **while**-loop:

while $i \le j$ **do**
  $i \leftarrow i + 1$; $j \leftarrow j - i$
od

We can ask the question: How often will the body of this loop be executed? Obviously, the answer to this question depends on the values the variables i and j contain at the beginning. Let us call these values i_0, j_0 and furthermore, denote by i_n, j_n the values of the variables i and j *after the n's iteration* of the loop.

Let us start by constructing a small table for a small example. For this purpose, let us choose $i_0 = -3$ and $j_0 = 10$.

n	0	1	2	3	4	5	6	7
i_n	−3	−2	−1	0	1	2	3	4
j_n	10	12	13	13	12	10	7	3

Because $i_7 > j_7$, the loop will not be executed for the 8's time.

It is now easy to write down a recurrence relation for i_n and j_n:

$$i_n = i_{n-1} + 1 \text{ and } j_n = j_{n-1} - i_{n-1} - 1$$

These two recurrence relations are interleaved but only the second with the first. We can solve the recurrence relation for i_n immediately and the result is simply $i_n = i_0 + n$. This closed form can be inserted into the second recurrence relation

$$j_n = j_{n-1} - (i_0 + n - 1) - 1 = j_{n-1} - i_0 - n.$$

Again, this is a hidden sum and the solution is

$$j_n = j_0 - \sum_{k=1}^{n} (i_0 + k) = j_0 - n i_0 - \frac{n(n+1)}{2}.$$

The body of the loop is executed as long as $i \leq j$ holds. The $(n+1)$st execution takes place if and only if the nth execution took place and additionally $j_n - i_n \geq 0$. If $i_0 > j_0$, then the loop will not be executed at all.

Let us take a closer look at the condition $j_n - i_n \geq 0$:

$$j_n - i_n = -\frac{n(n+1)}{2} - n i_0 + j_0 - i_0 - n = -\frac{1}{2}n^2 - \frac{1}{2}(3 + 2i_0)n + j_0 - i_0 \geq 0$$

If we multiply this inequality by -2, we get the following one, which looks a little bit nicer:

$$n^2 + (3 + 2i_0)n - 2(j_0 - i_0) \leq 0. \qquad (3.6)$$

The equation $x^2 + (3 + 2i_0)x - 2(j_0 - i_0) = y$ describes a parabola. We can assume that $n = 0$ is a solution of (3.6) because otherwise the loop is not

executed at all. Because of this, the parabola will have real roots. The inequality (3.6) will be fulfilled for all ns starting from 0 up to the right root. This second root is

$$\zeta = -i_0 - \frac{3}{2} + \frac{1}{2}\sqrt{4i_0(i_0+1) + 8j_0 + 9}.$$

With other words (3.6) holds for $0 \leq n \leq \zeta$, which is equivalent to $0 \leq n \leq \lfloor \zeta \rfloor$. All together the body of the loop will be executed $\lfloor \zeta \rfloor + 1$ times. Explicitly written this number is

$$\left\lfloor \frac{1}{2}\sqrt{4i_0(i_0+1) + 8j_0 + 9} - i_0 - \frac{1}{2} \right\rfloor. \tag{3.7}$$

At this point, it might be a good idea to test this explicit formula for the number of executions on an example. Let us assume again that $i_0 = -3$ und $j_0 = 10$. If we plug in these values into (3.7) we get

$$\left\lfloor \frac{1}{2}\sqrt{4(-3)(-2) + 8 \cdot 10 + 9} + 3 - \frac{1}{2} \right\rfloor = \left\lfloor \frac{1}{2}\sqrt{117} + \frac{5}{2} \right\rfloor = \lfloor 7.91 \rfloor = 7.$$

This result is correct.

Let us try a more complicated problem:

```
for k = 1 to m
  i ← k; j ← k²;
  while i ≤ j do
    i ← i + 1; j ← j − i
  od
od
```

How often is the body of the inner loop executed?

We denote by $W(i_0, j_0)$ the number of executions of the body of the while loop if at its beginning the variables have the values $i = i_0$ and $j = j_0$. Above we already established a closed formula for $W(i_0, j_0)$: $W(i_0, j_0) = \lfloor \frac{1}{2}\sqrt{4i_0(i_0+1) + 8j_0 + 9} - i_0 - \frac{1}{2} \rfloor$.

$$\sum_{k=1}^{n}\lfloor\sqrt{k}\rfloor = \sum_{k=1}^{n}\sum_{i=1}^{\lfloor\sqrt{k}\rfloor} 1 = \sum_{k}\sum_{i}(1 \le k \le n \wedge 1 \le i \le \lfloor\sqrt{k}\rfloor) =$$
$$= \sum_{k}\sum_{i}(1 \le k \le n \wedge 1 \le i^2 \le k) = \sum_{n}\sum_{i}(1 \le i^2 \le k \le n) =$$
$$= \sum_{i=1}^{\lfloor\sqrt{n}\rfloor}\sum_{k=i^2}^{n} 1 = \sum_{i=1}^{\lfloor\sqrt{n}\rfloor}(n - i^2 + 1) = \lfloor\sqrt{n}\rfloor(n+1) - \frac{i(i+\frac{1}{2})(i+1)}{3}\Bigg|_{i=0}^{\lfloor\sqrt{n}\rfloor} =$$
$$= \lfloor\sqrt{n}\rfloor(n+1) - \frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+\frac{1}{2})(\lfloor\sqrt{n}\rfloor+1)}{3}$$

Figure 3.2: How to compute the sum of $\lfloor\sqrt{k}\rfloor$.

The total number is simply

$$\begin{aligned}
\sum_{k=1}^{m} W(k,k^2) &= \sum_{k=1}^{m}\left\lfloor\frac{1}{2}\sqrt{4k(k+1)+8k^2+9} - k - \frac{1}{2}\right\rfloor \\
&= \sum_{k=1}^{m}\left\lfloor\frac{1}{2}\sqrt{12k^2\left(1+\frac{1}{3k}+\frac{4}{4k^2}\right)} - k - \frac{1}{2}\right\rfloor \\
&= \sum_{k=1}^{m}\left\lfloor\sqrt{3}\cdot k\left(1+\frac{1}{6k}+O\left(\frac{1}{k^2}\right)\right) - k - \frac{1}{2}\right\rfloor \\
&= \sum_{k=1}^{m}\left\lfloor k(\sqrt{3}-1)+\frac{\sqrt{3}}{6}-\frac{1}{2}+O(k^{-1})\right\rfloor
\end{aligned}$$
$$= \sum_{k=1}^{m}\Big(k(\sqrt{3}-1)+O(1)\Big) = \frac{\sqrt{3}-1}{2}m^2 + O(m) \approx 0.366\cdot m^2 \quad (3.8)$$

Instead of establishing an exact formula for this summation, we just compute an estimate. For this end we use Taylor's theorem:

$$\sqrt{1+x} = 1 + \frac{1}{2}x + O(x^2)$$

If use the value $m = 1000$ in our approximatively correct formula, we get 366025. The exact value is 365687.

3.12 Searching an unordered array

We start with an example. Let us assume we have an array $a[1]\ldots a[n]$ with pairwise distinct numbers. We want to write a program that finds out

whether a given number is contained in the array. An obvious solution in the programming language C might look as follows:

```
int n;
int a[100];
```

What is the running time of this program?

It is quite obvious that in this case the answer depends on various factors. One of them is whether v is contained in the array or not.

Let us first consider an *unsuccessful* search: The **for**-loop will be executed n times and after that 0 is returned.

In the case of a successful search, on the other hand, there is some i with $a[i] = v$. To be able to analyse this case, we need to know something about which i happens to have this property. In general, we can try to make a statistical assumption about the input. In the following we will assume that all elements in the array are in a random order. Then for each i between 1 and n the probability that $a_i = v$ is exactly $1/n$.

Let us denote the running time of the program by $L(i)$, if $a[i] = v$. The average running time is then

$$\frac{1}{n}\sum_{i=1}^{n} L(i).$$

All that is left is to find a closed formula for $L(i)$. The **for**-loop and the **if**-statement are executed exactly i times. If $L(i)$ is the number of machine instructions, we need to look at the machine programe (Figure 3.3). Let Z be the average number of executions of the **for**-loop. We get

$$Z = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2}.$$

The running time of the program happens to be $24 + 8Z$ machine instructions according to figure 3.3. This results in $24+4(n+1) = 4n+28$ machine instructions on average, if the size of the array is n.

Is it necessary to look at a recurrence relation to solve this problem? At first glance no, but this assumption is not completely correct.

The situation is just so simple that you can see the solution at once. Just for fun it is also possible to solve it systematically with recurrence relations.

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#24
1 sw 0(r29),r2
1 sw 4(r29),r3
1 sw 8(r29),r4
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r3,(r1)
1 sgti r1,r3,#0
1 beqz r1,L3
1 lw r4,(r30)
1 lhi r1,((_a)>>16)&0xffff
1 addui r1,r1,(_a)&0xffff
1 lw r2,(r1)
1 addi r31,r2,#4
1 slli r1,r3,#0x2
1 add r2,r1,r2
L5:
  Z lw r1,(r31)
  Z seq r1,r1,r4
  Z bnez r1,L8
  Z addi r1,r0,#1
Z-1 addi r31,r31,#4
Z-1 sle r1,r31,r2
Z-1 bnez r1,L5
Z-1 nop
L3:
  1 addi r1,r0,#0
L8:
 1 lw r2,0(r29)
 1 lw r3,4(r29)
 1 lw r4,8(r29)
 1 lw r31,-8(r30)
 1 add r29,r0,r30
 1 jr r31
 1 lw r30,-4(r30)
```

Figure 3.3: Search program

In order to do so, we have to reduce the case of n elements to the case of $n-1$ elements. This is not complicated: $Z_1 = 1$, because if we search for just one key and you find it then you use exactly one comparison. If $n > 1$, we get $Z_n = 1 \cdot \frac{1}{n} + (1 - \frac{1}{n})(1 + Z_{n-1})$ because with a probability of $\frac{1}{n}$ we can find v in the first place of the array and with a probability of $1 - \frac{1}{n}$ we have to search it in the remaining $n-1$ places. The latter task needs another Z_{n-1} comparisons on average. The recurrence looks as follows:

$$Z_n = 1 + (1 - \frac{1}{n})Z_{n-1}$$

where $Z_1 = 1$. This is a linear recurrence relation of first order that can be solved with a summation factor. On page ?? we can find the solution as a formula. In this case we get

$$\begin{aligned} Z_n &= 1 + \sum_{j=1}^{n-1} \left(1 - \frac{1}{j+1}\right)\left(1 - \frac{1}{j+2}\right)\left(1 - \frac{1}{j+3}\right)\cdots\left(1 - \frac{1}{n}\right) \\ &= 1 + \sum_{j=1}^{n-1} \frac{j}{j+1}\frac{j+1}{j+2}\frac{j+2}{j+3}\cdots\frac{n-1}{n} = 1 + \sum_{j=1}^{n-1}\frac{j}{n} = 1 + \frac{n-1}{2} = \frac{n+1}{2}. \end{aligned}$$

The overall result is correct. Were we allowed to use this formula at all? Yes, because all pre-conditions are valid in particular $Z_0 = 0$.

Let us now improve the search algorithm in order to see what impact our improvements have. In a first step we access the array by a pointer instead of an index. Moreover, let us count the array elements backwards in order to have a more efficient comparisons with zero. The resulting program is

```
1 sw -4(r29),r30                 1 lhi r1,((_n)>>16)&0xffff      Z seq r1,r1,r3          1 addi r1,r0,#0
1 add r30,r0,r29                 1 addui r1,r1,(_n)&0xffff       Z bnez r1,L8           L8:
1 sw -8(r29),r31                 1 lw r31,(r1)                   Z addi r1,r0,#1         1 lw r2,0(r29)
1 subui r29,r29,#16              1 sgti r1,r31,#0              Z-1 addi r31,r31,#1       1 lw r3,4(r29)
1 sw 0(r29),r2                   1 beqz r1,L3                  Z-1 sgti r1,r31,#0        1 lw r31,-8(r30)
1 sw 4(r29),r3                   1 lw r3,(r30)                 Z-1 bnez r1,L9            1 add r29,r0,r30
1 lhi r1,((_a)>>16)&0xffff       1 addi r2,r2,#4               Z-1 addi r2,r2,#4         1 jr r31
1 addui r1,r1,(_a)&0xffff       L9:                              1 addi r2,r2,#-4        1 lw r30,-4(r30)
1 lw r2,(r1)                     Z lw r1,(r2)                   L3:
```

Figure 3.4: Another search program

```
int search2(int v) {
  int i;
  int * p = &a[0];
  for(i = n; i > 0; i--) {
    if(*++p == v) return 1;
  }
  return 0;
}
```

The corresponding DLX assembler program can be found in figure 3.4.

This time we get $24+8Z$ machine instructions, which is $4n+24$ on average. This “improvement” is not very good because we save only 4 instructions.

It seems that it is not easy to improve this program significantly. There is, however, one trick left that helps a lot: we avoid counting. In order to do so, we store v at the end of the array and consequently we don't have to check anymore whether we reached the end of the array:

```
int search3(int v) {
  int i;
  int * p = &a[0];
  a[n + 1] = v;
  while(*p ≠ v) p++;
  if(p == &a[n + 1]) return 0;
  return 1;
}
```

This time it turns out that $35 + 4Z$ machine instructions are executed. On average this makes $2n + 37$ instructions. For big n this is much faster as the previous solutions, but for small n it might be slower. If you are only interested in successful searches, then this more clever search will be superior for $n \geq 5$.

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#24
1 sw 0(r29),r2
1 sw 4(r29),r3
1 sw 8(r29),r4
1 lw r3,(r30)
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r1,(r1)
1 lhi r2,((_a+4)>>16)&0xffff
1 addui r2,r2,(_a+4)&0xffff
1 slli r1,r1,#0x2
1 add r1,r1,r2
1 lhi r4,#1
1 addui r4,r4,#34464
1 sw (r1),r4
1 addi r31,r2,#-4
1 addi r31,r31,#4
  L9:
P-1 lw r2,(r31)
P-1 sgt r1,r2,r3
P-1 bnez r1,L9
P-1 addi r31,r31,#4
1 addi r31,r31,#-4
1 seq r2,r2,r3
1 bnez r2,L6
1 addi r1,r0,#0
  L6:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r4,8(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 jr r31
1 lw r30,-4(r30)
```

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#16
1 sw 0(r29),r2
1 sw 4(r29),r3
1 lw r31,(r30)
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r1,(r1)
1 lhi r2,((_a+4)>>16)&0xffff
1 addui r2,r2,(_a+4)&0xffff
1 slli r1,r1,#0x2
1 add r1,r1,r2
1 sw (r1),r31
1 addi r2,r2,#-4
1 addi r2,r2,#4
L9:
1 lw r1,(r2)
1 sne r1,r1,r31
1 bnez r1,L9
1 addi r2,r2,#4
1 addi r2,r2,#-4
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r1,(r1)
1 slli r1,r1,#0x2
1 lhi r3,((_a+4)>>16)&0xffff
1 addui r3,r3,(_a+4)&0xffff
1 add r1,r1,r3
1 seq r2,r2,r1
1 bnez r2,L6
1 addi r1,r0,#0
1 addi r1,r0,#1
L6:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 lw r30,-4(r30)
1 jr r31
1 nop
```

Figure 3.5: Intelligent search. The branch to label L6 is never taken in a successful search.

3.13 Searching an ordered array and binary search trees

Let us consider the problem of searching an ordered array. We assume as usual that the array $a[1], \ldots, a[n]$ contains n different numbers, but this time they are ordered. How long does it take on average to find one of these numbers if we search for each of them with the same probability? (Again this is a *successful* search.) We can also ask ourselves the question how long it takes to find out that some number is *not* contained in the array. Which probability distribution is the right one for this unsuccessful search? If the algorithm is based only on comparisons, then its running time depends on the place where the number that we are searching for belongs to. In principal there are $n+1$ places, i.e., before the first array element, behind the last array element, or in one of the $n-1$ gaps in between.

Again, the first algorithm we consider searches the array from left to right. As soon as we see an array element that is bigger than the key we are searching, we can abort the program. Let us assume we can put a pseudo number H behind the end of the array. This H should be bigger than all

numbers that occur in the array.

```
int search4(int v)
{
    int * p;
    a[n + 1] = H;
    p = a;
    do { p++; } while(*p < v);
    if(*p == v) return 1;
    else return 0;
}
```

The running time of this program depends on how often the instruction $p{+}{+}$ is carried out. If the search in unsuccessful we increase p until it points to $a[K]$ where K is the smallest index with $a[K] \geq v$. In the beginning p points to $a[0]$. In the case of an unsuccessful search the pointer is increased exactly K times. K is a random variable with the distribution

$$\Pr[K = k] = \frac{1}{n+1} \text{ for } 1 \leq k \leq n+1.$$

Let us call the average number of times the instruction p++ is carried out P_n if the array has n elements. With other words P_n is simply the expected value of K:

$$P_n = E[K] = \sum_{k=1}^{n+1} k \cdot \Pr(K = k) = \frac{(n+2)(n+1)}{2(n+1)} = 1 + \frac{n}{2}$$

How big is P_n in the case of an unsuccessful search? If the search is successful then $a[K] = v$ for exactly one $1 \leq K \leq n$. In that case $\Pr[K = k] = 1/n$. Consequently, we get

$$P_n = E[K] = \sum_{k=1}^{n+1} k \cdot \Pr[K = k] = \frac{(n+1)n}{2n} = \frac{1}{2} + \frac{n}{2}.$$

We can expect that the successful and unsuccessful case are similar with regard to the running time.

Let us now count the number of executed machine instructions of the corresponding machine program in Figure 3.6.

The running time is $27+4P_n$ on average, i.e., $2n+31$ for a successful search and $2n + 29$ for an unsuccessful search.

```
1 sw -4(r29),r30
1 add r30,r0,r29
1 sw -8(r29),r31
1 subui r29,r29,#24
1 sw 0(r29),r2
1 sw 4(r29),r3
1 sw 8(r29),r4
1 lw r3,(r30)
1 lhi r1,((_n)>>16)&0xffff
1 addui r1,r1,(_n)&0xffff
1 lw r1,(r1)
1 lhi r2,((_a+4)>>16)&0xffff
1 addui r2,r2,(_a+4)&0xffff
1 slli r1,r1,#0x2
1 add r1,r1,r2
1 lhi r4,#1
1 addui r4,r4,#34464
1 sw (r1),r4
1 addi r31,r2,#-4
1 addi r31,r31,#4
    L9:
P-1 lw r2,(r31)
P-1 sgt r1,r2,r3
P-1 bnez r1,L9
P-1 addi r31,r31,#4
  1 addi r31,r31,#-4
  1 seq r2,r2,r3
  1 bnez r2,L6
  1 addi r1,r0,#0
    L6:
1 lw r2,0(r29)
1 lw r3,4(r29)
1 lw r4,8(r29)
1 lw r31,-8(r30)
1 add r29,r0,r30
1 jr r31
1 lw r30,-4(r30)
```

Figure 3.6: Linear search in an ordered array.

In the following we will analyse algorithms that are based on comparisons with the help of the theory of *binary search trees*. This theory helps us to analyse the average number of comparisons that some class of algorithms execute.

Not suprisingly a binary search tree is a binary tree. It consists either of only one node that we call the root or a root that has two children that are themselves binary trees. We will distinguish between *internal* and *external* nodes: An internal node is a node that has itself two children, an external node in contrary has no children. External nodes are usually called leaves.

The comparisons performed by an algorithm lead in a natural way to a binary search tree: The root of a tree will be labeled with the first comparison the algorithm makes. The left child of the root will be the binary search tree of the following part of the algorithm that is executed if the result of the first comparisons was *smaller*. Similarly the right child of the root is the binary search tree for the result *bigger*. Let us assume for the moment that we are analysing a search algorithm and if the outcome of a comparison is equal then the algorithm will stop the search because the desired element has been found. With other words we assume that every comparison is against the key we are searching for.

As usual we draw binary search trees as a binary tree but we will draw internal nodes as circles and external nodes as squares. You can find the depiction of a search tree for an algorithm for linear search in an ordered array in Figure 3.7. The size of the array is in this case 10.

In the following we will recursively define some important parameters of a binary search tree T: The *size* $|T|$, which is exactly the number of internal

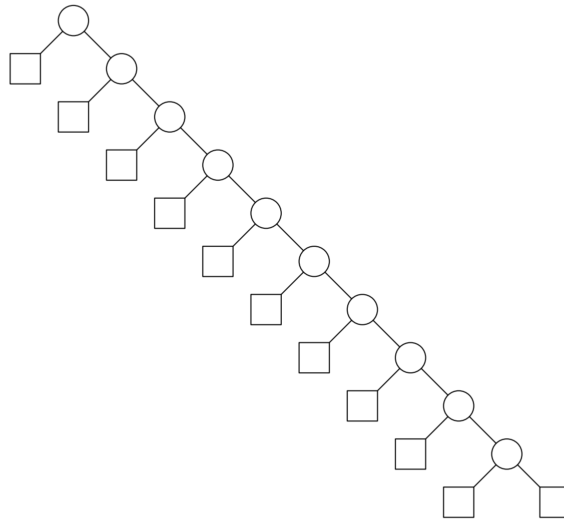

Figure 3.7: A search tree for a linear search in an ordered array of length 10.

nodes, the *internal path length* $\pi(T)$ and the *external path length* $\xi(t)$.

If T consists only of a root we define $|T| = 0$, $\pi(T) = 0$, $\xi(T) = 0$. In the case that T consists of a root that has the binary search trees T_1 and T_2 as its children, then we will define $|T| = |T_1|+|T_2|+1$, $\pi(T) = \pi(T_1)+\pi(T_2)+|T|-1$ and $\xi(T) = \xi(T_1) + \xi(T_2) + |T| + 1$.

Informally, the internal path length is the sum of all *levels* of all internal nodes and the external path length is the sum of all levels of all external nodes. The level of a node is its distance to the root, where the root itself is on Level 0.

One beautiful fact about these definitions is that if we know $\pi(T)$ and $\xi(T)$ we can easily compute the average number of comparisons that an algorithm performs.

Theorem 5. Let T be the comparison tree of an algorithm and let every element be chosen with uniform probability in the case of a successful search or each position between elements including the outer left and outer right position with uniform probabilty in the case of an unsuccessful search, then the average number of comparisons is

$$\begin{aligned} C^+ &= \frac{\pi(T)}{|T|} + 1 \text{ in the successful case,} \\ C^- &= \frac{\xi(T)}{|T|+1} \text{ in the unsuccessful case.} \end{aligned}$$

Moreover the following correspondance holds between the internal and external path length:

$$\xi(T) = \pi(T) + 2|T|$$

There is also the following correspondance between C^+ and C^-:

$$C^- = (C^+ + 1)\left(1 - \frac{1}{|T| + 1})\right)$$

The number of external nodes is always $|T| + 1$.

Let us use these formulas for a linear search in an array of n elements. As search trees we get caterpillars L_n as you can see in figure 3.7. The internal path length is

$$\pi(L_n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2},$$

because on each level between 0 and $n-1$ there is exactly one internal node. The external path length is then

$$\xi(L_n) = \pi(L_n) + 2n = \frac{n(n+3)}{2}.$$

The average number of comparisons in the case of an unsuccessful search is

$$\frac{\pi(L_n)}{|L_n|} + 1 = \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

and the number of comparisons in the case of an unsuccessful search should be, according to our formula,

$$\frac{\xi(L_n)}{|L_n| + 1} = \frac{n(n+3)}{2(n+1)} = \frac{n}{2} + \frac{n}{n+1}.$$

Obviously this is not correct.

Where is the mistake? The error lies in the way the program proceeds if the key that we search is bigger than $a[n]$ i.e. the last element in the array. After the program verified that $a[n] < v$ a consequent comparison is no longer necessary. Still the program carries out another comparison with $a[n+1]$. This is not a comparison according to our definition because *there is only one possible answer and the comparison is redundant.* The theory of comparison trees works only if the probability of reaching

every external node in an unsuccessful search is the same. This might not be true if there are redundant comparisons.

The last comparison cannot be spotted in the search tree we drew: The search tree contains only n internal nodes labeled with the comparisms $a[1] : v, \dots, a[n] : v$. If the algorithm visits the last external node in the tree it performs another redundant comparison and that happens with a probability of $1/(n+1)$ in an unsuccessful search. The actual number of comparisons taken on average in an unsuccessful search is consequently

$$\frac{\xi(L_n)}{|L_n|+1} + \frac{1}{n+1} = \frac{n}{2} + 1,$$

and this coincides with the result we got when we analysed this program traditionally without the help of comparison trees.

No problem would have occured if the program were written in the following form

```
int search5(int v)
{
  int i = 0;
  do { i++; } while(i ≤ n && a[i] > v);
  if(i == n + 1) return 1;
  else return 0;
}
```

Here indeed only $n/2+n/(n+1)$ comparisms are done on average in an unsuccessful search. This program however is much slower. What we should learn from this: The formulas for the average number of comparisons are only correct if all preconditions are met. We have to check them carefully and have to take any exceptions into consideration.

Binary search

If the array is ordered, *binary* search will be the method of choice: We will compare v with the key that is approximately in the middle of the array. Doing so reduces the problem to searching the key in an array of only half the size. The following algorithm does exactly that:

```
int binsearch(int v)
{
    int l, r, m;
    l = 1;  r = n;
    while(l ≤ r) {
        m = (r + l)/2;
        if(v == a[m]) return 1;
        if(v < a[m]) r = m − 1; else l = m + 1;
    }
    return 0;
}
```

This algorithm works as follows. It uses two variables l and r to remember the subarray in which we still have to search. Here l is the leftmost and r the rightmost element of this subarray. The algorithm compares the key to the key in the middle. If it is the correct one, the algorithm immediately terminates. Otherwise the right or left border of the subarray that still might contain v will be adjusted and we continue the search.

If we designate by B_n how often the instruction $m = (r+l)/2$ is performed then B_n is exactly the number of comparisons $v : a[i]$. Let us first consider the unsuccessful search because it is easier to analyse and also let us start with the worst case.

Let $N = r - l + 1$ be the size of the active subarray. Let C_N be the number of times the instruction $m = (r+l)/2$ is still executed if now $r - l + 1 = N$. With these definitions in mind we get $C_1 = 1$ because $N = 1$ implies that $r = l$ and after executing $m = (r+l)/2$ the algorithm either terminates, or r is increased, or l is decreased. After that the **while**-loop immediately terminates.

If $N > 1$ then $m = (r+l)/2$ is executed at least once and after that either $r := \lfloor (r+l)/2 \rfloor - 1$ or $l := \lfloor (r+l)/2 \rfloor + 1$ will be executed. In both cases this implies $N := \lfloor N/2 \rfloor$; as we expected the size of the active subarray is cut roughly in half with each iteration.

How does the binary search tree for this algorithm look like? Figure 3.8 shows the search tree for $n = 10$. In general it will be an almost complete binary tree in which only nodes on the last level might be missing.

If the search tree has exactly 2^k external nodes then all of them are located on level k and the external path length is exactly $k2^k$. Let us now look at

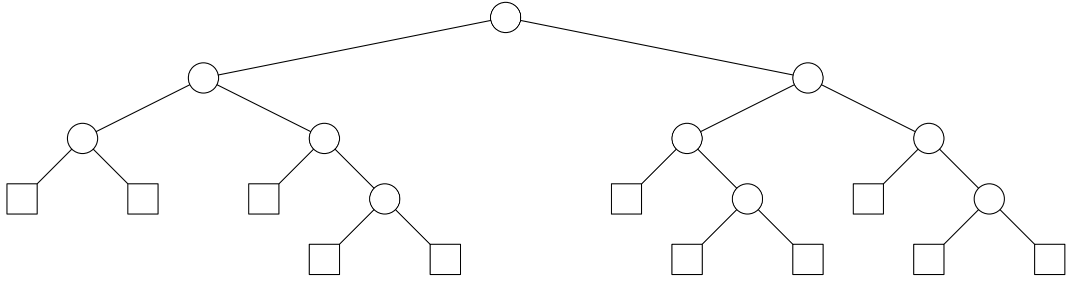

Figure 3.8: A search tree for binary search in an ordered array of size 10.

the general case. If we have n internal nodes we have exactly $n+1$ external nodes and $2(n+1-2^{\lfloor \log(n+1) \rfloor})$ of them are located on level $\lfloor \log(n+1) \rfloor + 1$. The remaining nodes i.e. exactly $2^{\lfloor \log(n+1) \rfloor + 1} - n - 1$ of them are located on level $\lfloor \log(n+1) \rfloor$. If we denote the search tree for binary search in an array with n elements by B_n we consequently get

$$\xi(B_n) = (n+1)(\lfloor \log(n+1) \rfloor + 2) - 2^{\lfloor \log(n+1) \rfloor + 1}.$$

From this it is easy to compute the internal path length:

$$\pi(B_n) = \xi(B_n) - 2n = (n+1)\lfloor \log(n+1) \rfloor - 2^{\lfloor \log(n+1) \rfloor + 1} + 2.$$

As always it is a good idea to test this formula on a small example. Let us again choose $n = 10$ because we already have the corresponding search tree in figure 3.8. We get $\xi(B_{10}) = 11 \cdot 5 - 16 = 39$ and $\pi(B_{10}) = 11 \cdot 3 - 16 + 2 = 19$. Both results can be easily checked with the help of the binary search tree in figure 3.8.

Exercises

3.1 Solve the following recurrence relation and find a nice way to write down the solution.

$$\begin{aligned} c_0 &= 2 \\ c_1 &= 4 \\ c_n &= c_{n-2}^{\log c_{n-1}} \end{aligned}$$

3.2 Drill Sergeant Even is in a bad mood and lets his new recruits march in a row of two along the yard. He flips his lid whenever the number of recruits is odd and then drives them along DEATH LANE. When this happens to a recruit he gets

sick with a probability of 1/2 and cannot partake in the exercise anymore. This spectacle is repeated until the number of recruits becomes even.
How many runs through DEATH LANE take place on average?

3.3 Solve the following recurrence! Let $a_0 = 0$, $a_1 = 3$, and $a_n = 4a_{n-1} - 4a_{n-2}$ for $n > 1$.

3.4 Solve the following recurrence relation. Let Es sei $b_1 = b_2 = b_3 = 1$ and $b_n = 3b_{n-1} - 4b_{n-2} + 12b_{n-3}$ for $n > 3$.

3.5 Compare the solution $2(nH_n)+2(H_n)-2(n) = 2nH_n+2H_n-2n$ form page 39 to the general solution from the first chapter by setting $M = 0$.

3.6 Given an array a of length n, an algorithm compares all pairs $(a[i], a[j])$ for all $i < j \leq n$, and then calls itself recursively on all proper prefixes of a.
How often does the algorithm compare two pairs? Use the repertoire method!

3.7 Improve the estimate of (??). The goal is to get an additive error term of $O(1/n)$ or better. How far away is your new estimate for a_{10} from the true value?

3.8 Use a summation factor on (3.4) and find the solution of the recurrence (3.3) not in closed form, but as a summation.

3.9 Solve the recurrence

$$\begin{aligned} a_0 &= 8000 \\ a_1 &= 1/2 \\ a_{n+2} + a_{n+1} - n^2 a_n &= n! \end{aligned}$$

by order reduction.

3.10 Compute the number of times the body of the **while**-loop is performed, if initially $0 < i$ holds.

```
while i <= j
   i := i+j ;
   if i > j then j:=j+10 ;
```

3.11 Solve the last exercise with the assumption that $i \leq 0$.

3.12 Analyse the running time of a successful search for the program in Figure 3.3 if every element in the array occurs twice and again every permutation has the same probability.

3.13 Compare all three search algorithms according to successful searches.

3.14 Consider the following algorithm that searches an element x in a sorted array a of length $n = km + 1$:

```
i:= 1 ;
while a[i]<=x
```

```
  if a[i]=x then return i ;
  i:=i+m ;
  if i>n return 0 ;
for j=i-1 downto max(1,i-(m-1))
  if a[j]=x then return j ;
return 0 ;
```

a) Draw the search tree and compute the internal and external path length for $n = 10$ and $m = 3$.

b) Determine C^+ and C^- for arbitrary m, k.

c) What is, for given n, the best choice for m w.r.t. the running time?

3.15 Verify that the claim $N := \lfloor N/2 \rfloor$ on page 55 is correct.

3.16 We want to compare the following two programs for a search in a sorted array:

```
int binsearch(double v)
{
  int l,r,m;
  l=1; r=N;
  while (l ≤ r) {
    m=(r+l)/2;
    if (v ≡ a[m]) return 1;
    if (v<a[m]) r=m−1; else l=m+1;
  }
  return 0;
}
```

```
int binsearch2(double v)
{
  int l,r,m;
  l=1; r=N;
  while (r−l>1) {
    m=(r+l)/2;
    if (v<a[m]) r=m−1; else l=m;
  }
  if (a[l] ≡ v) return 1;
  if (a[r] ≡ v) return 1;
  return 0;
}
```

Analyse how many **if**-instructions are executed by the programs in case of a successful or unsuccessful search for v. Find an exact solution for the first program and an estimate of the form $f(n) + O(1)$ for the second one. Make the usual assumptions about v.