Analysis of Algorithms

Peter Rossmanith
Institute for Theoretical Computer Science
RWTH Aachen

Lecture Notes and Exercises, Summer 2012

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 **23**

3 Recurrence relations **29**

3.1 Classification of recurrences 30

3.2 Creating a table 31

3.3 Guessing a solution and proving by induction 31

3.4 Looking up the solution 32

3.5 Mathematica, Maple & Co 33

3.6 Hidden products and sums 33

3.7 Linear recurrence relation with constant factors 35

3.8 Summation factor 37

3.9 The Repertoire method 38

3.10 Order reduction 41

3.11 Extracting recurrence relations from algorithms 43
3.12 Searching an unordered array 46
3.13 Geordnete Felder und binäre Suchbäume 50

4 Generating functions **61**
4.1 Zählen mit erzeugenden Funktionen 68
4.2 Bivariate erzeugende Funktionen 70
4.3 Exponentielle erzeugende Funktionen 71
4.4 Die symbolische Methode 72
4.5 Mittlere Stackgröße 77

5 Asymptotische Abschätzungen **83**
5.1 Die Eulersche Summationsformel 84
5.2 Singularitätenanalyse 86
5.3 Meromorphe Funktionen 89
5.4 Algebraische Singularitäten 95
5.5 Die Sattelpunktmethode 98

A Solutions to Selected Exercises **109**

B MIPS Cheat Sheet **131**

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 $a[1],\ldots,a[N]$ 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 [?], 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

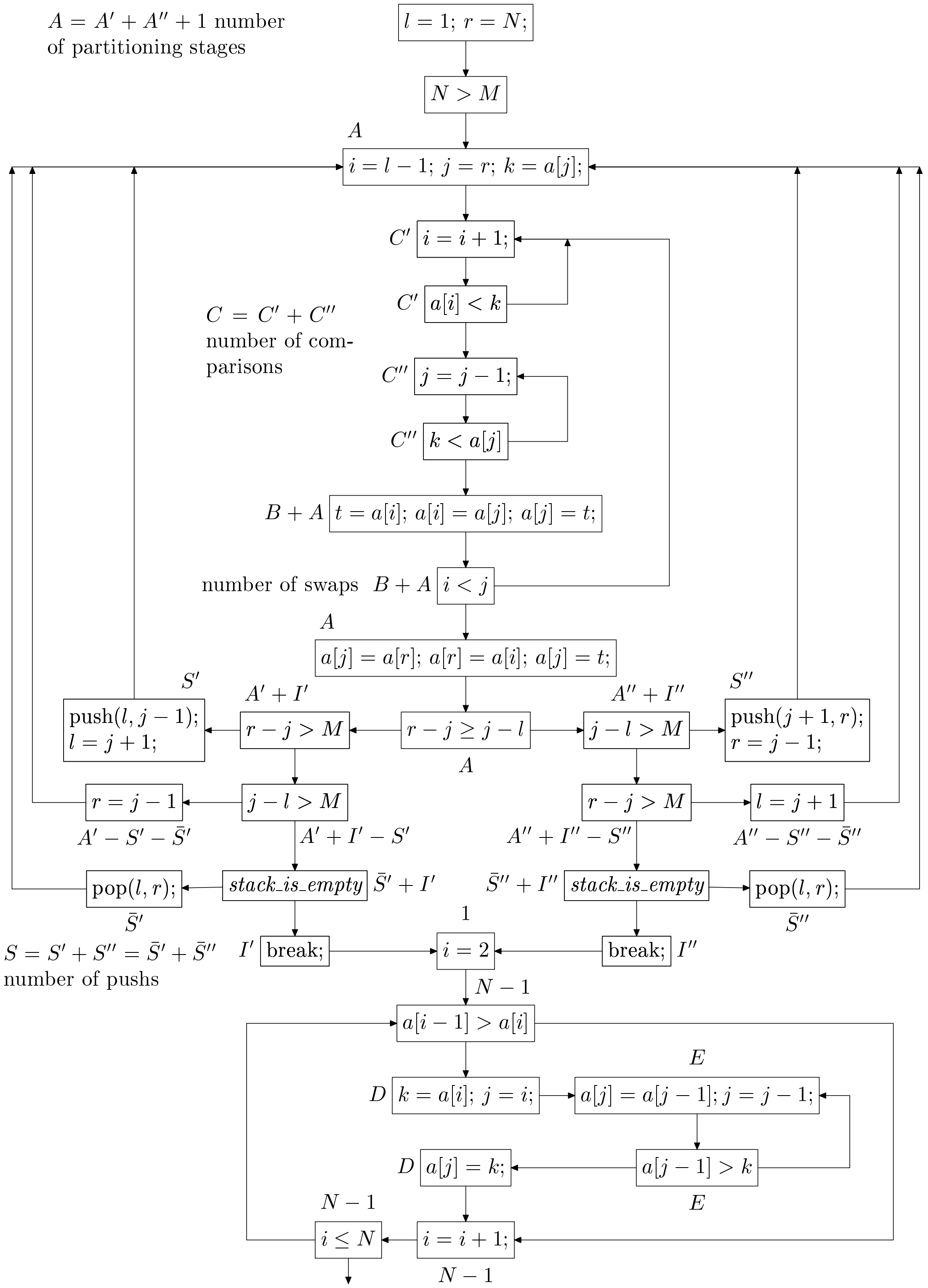


Figure 1.2: Program flow chart for the Quicksort program

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

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

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 $X_0, \ldots, X_{N-1}$ 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 $N \le M$. 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:

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

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

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

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 a recurrence relation

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

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 $a[1], \ldots, a[i-1]$ has already been sorted by insertion sort. Before that, $a[1, \ldots, i]$ was in random order. At this point of time $a[i-1] > a[i]$ iff $a[i]$ is not the biggest key in $a[i-M+1], \ldots, a[i]$. 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.

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

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

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

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 $\pi\colon \{1,\dots,n\} \to \{1,\dots,n\}$ 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:

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

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 $1, \dots, N$ 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 $1, \dots, 10$ and find out what $A_{10}, \dots, S_{10}$ 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 $ds[0] \ldots ds[2 * N - 2]$ 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 ← −1;
           xor   dx, dx            A dx ← 0;
next:      cmp   dx, N             B i < N ?
           jae   done              B jump if above or equal (i ≥ N)
           mov   bx, ds:[2*dx]     C bx ← a[dx]
           cmp   bx, max           C bx > max ?
           jna   skip              C jump if not above (bx ≤ N)
           mov   ax, bx            D ax ← bx
skip:      add   dx, 0x0002        E ax ← ax + 1;
           jmp   next              E jump
done:      push  ax                F push the maximum on the stack
```

1.14 Studentenparty! DJ O*D*D ist im Haus und hat unendlich viele Lieder aus den drei Genres Rock, Gabba und Blues zur Verfügung. In der Nacht wird er n Lieder spielen, so dass es theoretisch 3^n mögliche Kombinationen der Genres gibt. Allerdings gelten einige Einschränkungen:

1. Unmittelbar nach einem Rocksong kann kein Gabbatitel gespielt werden, da die dafür nötige Anpassung der Equalizer zu lange dauerte.

2. Unmittelbar nach einem Gabbatitel kann kein weiterer Gabbatitel aufgelegt werden, da die Besucher sonst an Expressverblödung stürben.

3. Sobald einmal Blues gespielt wurde, kann nur noch Blues gespielt werden, weil alle traurig sind.

Stellen Sie Rekursionsgleichungen für die Anzahl der Kombinationen auf und bestimmen Sie die Lösungen!

1.15 Gegeben ist ein Array a der Länge N, in dessen Zellen jeweils zufällig und gleichverteilt gewählte Zahlen aus $\{1, \ldots, N\}$ stehen. Wie oft werden die verschiedenen Anweisungen des folgenden Algorithmus im Erwartungswert ausgeführt?

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

1.16 Gegeben ist ein Wort w aus $\{a, b\}^n$, das zufällig und gleichverteilt gewählt wird. Wie oft wird der Rumpf der `while`-Schleife im folgenden Algorithmus im Erwartungswert ausgeführt? Die Funktion `is_palindrome` prüft naheliegenderweise, ob das ihr übergebene Wort ein Palindrom ist, also rückwärts gelesen sich selbst ergibt.

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

1.17 Zwei natürliche Zahlen $m \neq n$ sind *befreundet*, wenn die Summe der echten Teiler von m genau n ergibt – und umgekehrt. Ein Sohn und sein Vater schrieben diese zwei Programme, welche befreundete Zahlen berechnen. Was sind die Laufzeiten der beiden Programme?

Sohn

```
#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";
  }
}
```

Vater

```
#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␣%d\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 $i = 1, \ldots, m$.

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. ***(Kirchhoffsches Gesetz)***
Let I be the set of all i for which the edge e_i ends in some node X and let

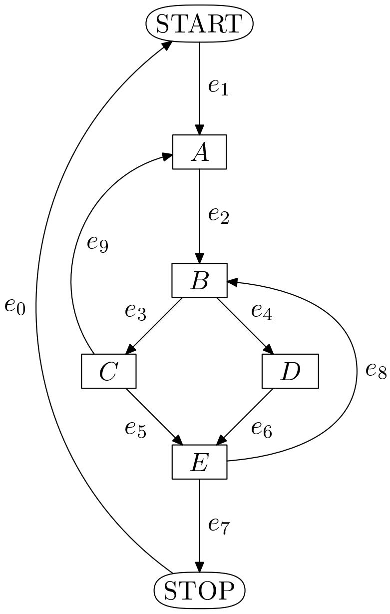


Figure 2.1: Beispiel für einen Programmflußgraphen mit und ohne Spannbaum.

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:

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

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

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 [?, 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.

```
for each u ∈ V do
    key[u] ← ∞
    π[u] ← NIL
key[r] ← 0
M ← V
while (M ≠ ∅) do
    u ← min-from(M)
    for each v ∈ neighbors(u) do
        if (v ∈ M) ∧ (w(u, v) < key[v]) then
            π[v] ← u
            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 a running time or some other parameter of an algorithm, you want to find a close mathematical formula that describes the parameter you are analysing. More often than not you are not able to find such a formula writer way, 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 the 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 close formula for X_N.

In general no closed formula for the solution of a recurrence relation does exist—in practice, on the other hand, 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.

3.1 Classification of recurrences

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

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

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 $a_0, a_1, \dots, a_{t-1}$ are already fixed. In general the solution of the recurrence relations will depend on these starting values. If the recurrence relations originates from the analysis of an algorithm, then the starting values of $a_0, a_1, \dots, a_{t-1}$ are usually fixed by the algorithm.

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

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

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:

$$
\begin{aligned}
a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-t}) &\text{ Recurrence relation of t-th order} \\
a_n = \sum_{k=0}^{n-1} x(k,n) &\text{ homogenious, linear recurrence relation} \\
a_n = \sum_{k=0}^{n-1} x(k,n) + f(n) &\text{ linear recurrence relation} \\
a_n = x_1 a_{n-1} + x_2 a_{n-2} + \cdots + x_t a_{n-t} &\text{ linear with constant coefficients}
\end{aligned}
$$

In this chapter we will look 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 to 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 $a_2 = a_1 + 2a_0 = 3$, $a_3 = a_2 + 2a_1 = 5$, $a_4 = a_3 + 2a_2 = 11$, $a_5 = 21$, $a_6 = 43$. Then we 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 haven't made any mistake when finding the closed formula.

3.3 Guessing a solution and proving 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 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 +1 and sometimes they are the predecessor -1. 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 solutions but they are alternatingly $\frac{1}{3}$ too small or too big. This suggest as a closed formula as follows:

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

Let us varify 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 lexikographical order. Of course, only the beginning of each sequence is listed together with the 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 these series in these two publications.

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

`http://www.research.att.com/njas/sequences/index.html`

At this page 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 three more pointers to literature and also a closed formula for the term a_n.

3.5 Mathematica, Maple & Co

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

>
```

3.6 Hidden products and sums

The most simple recurrence relations are of the form

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

$\bar{a}_n = \log(a_n)$ Both forms are related to each other. If you substitute then the left recurrence relation turns into a recurrence relation of the right hand

AT&T Integer Sequences RESEARCH

Greetings from the On-Line Encyclopedia of Integer Sequences!

Abbreviations used | To identify a reference | To identify a sequence from its A number | Return to previous page
Please wait a moment while I search the data-base.

```
Matches (up to a limit of 10) found for  1 1 3 5 11 21 43  :

----------------------
%I A001045 M2482 N0983
%S A001045 0,1,1,3,5,11,21,43,85,171,341,683,1365,2731,5461,10923,21845,43691,
%T A001045 87381,174763,349525,699051,1398101,2796203,5592405,11184811,22369621
%N A001045 Jacobstahl sequence: a(n) = a(n-1) + 2a(n-2).
%R A001045 FQ 10 499 72; 26 306 88. JCT A26 149 79. AMM 102 705 95. TYCM 28 76 9
%O A001045 0,4
%A A001045 njas
%K A001045 nonn
%F A001045 a(n) = 2^n - a(n-1). a(n) = 2*a(n-1) - (-1)^n = {2^n - (-1)^n}/3.
```

Abbreviations used | To identify a reference | To identify a sequence from its A number | Return to previous page

See also: My home page | Integer Sequences home page | The book: The Encyclopedia of Integer Sequences | The full bibliography

Home | People | Library | Demos | History | News | TOC

[Home | People | Library | Demos | History | News | TOC]

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

type. We will call these two types of recurrence relations *hidden products* and *hidden sum.*

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

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

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 relation with constant factors

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

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

Here we have a recurrence relations 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 k-fold root of χ 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 side of aboves 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 k-fold root of χ.

Theorem 3. The homogenious linear recurrence relation with constant coefficients

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

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 multitude of the root α. All these solutions are linearly independent. The form a base of the vector base of all solutions.

Because the recurrence relation is linear and homogenious 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 base of the vector base of all solutions.

If we have t initial conditions, for example the value a_0, $a_1, \ldots$, a_{t-1} then we get exactly one solution by a linear combination of the solutions in our base. 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 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 it had 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 proove 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 $x_n x_{n-1} x_{n-2} \dots x_1$ 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 really f_n is different. We proceed now to try 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 comparisms. 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:

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

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

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

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 ant 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:

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

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 series and compute for each of them wobei where $f_b(n) = b_n - \sum_{i=1}^{t} x_{i,n} b_{n-t}$. In this way b_n is a closed solution of the recurrence relation

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

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 ant 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 ant we define the so-called *shift operator* E, which maps sequences to sequences. This operator is defined wire

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

which means that this operator shifts all members of a sequence by 1 to the top. 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 operator can be contraintuitive 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 parenthisis, 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:

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

here 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 possible. But if we are lucky, 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 polynomial, we then 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. 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 you try to factor it $p(E) = q(E)r(E)$. Next we consequently 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 shape

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

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 this the help of theorem 2 which is

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

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

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

3.11 Extracting recurrence relations from algorithms

Let us look at the following while:

while $i \leq j$ **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, we 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 to the first. We can solve the recurrence relation for i_n immediately and the result is simply $i_n = i_0 + n$. This result 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 $i \le j$ holds. The $(n+1)$'s execution takes place if and only if the n's execution took place and additionally $j_n - i_n \ge 0$. If $i_0 > j_0$, then the loop will not be executed at all.

Let us look at the condition $j_n - i_n \ge 0$ closer:

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

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

$$n^2 + (3 + 2i_0)n - 2(j_0 - i_0) \le 0. \tag{3.5}$$

The equation $x^2 + (3 + 2i_0)x - 2(j_0 - i_0) = y$ describes a parabola. We can assume that $n =$ is a solution of (3.5) because otherwise the loop is not executed at all. Because of this, the parabola will have real roots. The inequality (3.5) 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 (3.5) holds for $0 \le n \le \zeta$, which is equivalent to $0 \le n \le \lfloor \zeta \rfloor$. All together the body of the loop will be executed $\lfloor \zeta \rfloor + 1$ times. Explicitly written this number is

$$\left\lfloor \frac{1}{2}\sqrt{4i_0(i_0+1) + 8j_0 + 9} - i_0 - \frac{1}{2} \right\rfloor. \tag{3.6}$$

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 in (3.6) 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
  $i \leftarrow k;\ j \leftarrow k^2;$
  while $i \le j$ **do**
    $i \leftarrow i + 1;\ j \leftarrow j - i$
  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

$$\sum_{k=1}^{m} W(k,k^2) = \sum_{k=1}^{m}\left\lfloor\frac{1}{2}\sqrt{4k(k+1)+8k^2+9} - k - \frac{1}{2}\right\rfloor$$
$$= \sum_{k=1}^{m}\left\lfloor\frac{1}{2}\sqrt{12k^2\left(1+\frac{1}{3k}+\frac{4}{4k^2}\right)} - k - \frac{1}{2}\right\rfloor$$
$$= \sum_{k=1}^{m}\left\lfloor\sqrt{3}\cdot k\left(1+\frac{1}{6k}+O\left(\frac{1}{k^2}\right)\right) - k - \frac{1}{2}\right\rfloor$$
$$= \sum_{k=1}^{m}\left\lfloor k(\sqrt{3}-1)+\frac{\sqrt{3}}{6}-\frac{1}{2}+O(k^{-1})\right\rfloor$$
$$= \sum_{k=1}^{m}\Big(k(\sqrt{3}-1)+O(1)\Big) = \frac{\sqrt{3}-1}{2}m^2 + O(m) \approx 0.366\cdot m^2 \quad (3.7)$$

Instead of establishing an exact formula for this summation, we content ourselves with the rough estimation. For this ant we use Taylor's theory:

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

If the use the value $m = 1000$ in our approximatively correct formula, we get 366025. The exact values is 365682.

3.12 Searching an unordered array

We will start with an exmaple. Let us assume we have an array $a[1]\ldots a[n]$ with parewise distinct numbers. We want write a program that can find out whether a given number is contained in the array. An obvious solution in the programming language C might look as follows:

```
int search1(int v)
{
    int i;
    for(i=1; i ≤ n; i++)
        if(a[i] ≡ v) return 1;
    return 0;
}
```

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 a *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 assembler 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 result $24 + 4(n+1) = 4n + 28$ machine instructions on average, if the size of the array is n.

```
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: Suchprogramm

To solve this program we need not look at a recurrence relation as it seems. This isn't completely correct. The situation is so simple that you can see the solution at once. Just for fun it is also possible to solve it systematically with recurrence relations. 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 did exactly one comparism. 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 letter endever needs another Z_{n-1} comparisms 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 replace the access to the array by an index by a pointer access. Moreover, let us count the array elements back-

```
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: Noch ein Suchprogramm

wards in order to have a more efficient comparism to zero. The resulting program is

```
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.

In that case 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’s not easy to improve this program significantly. There is, however, one trick left that helps a lot: we will not count. 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=0, *p;
  a[n+1]=v;
  p=a;
  do { p++; } while(*p ≠ v);
  if(p ≡ a+n+1) return 0;
  else return 1;
}
```

```
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: Schlauere Suche. Der Sprung zum Label L6 wird bei einer erfolgreichen Suche nie durchgeführt.

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 $n \geq 5$.

3.13 Searching an ordered array and binary search trees

Let us first consider the problem of searching in an ordered array. We assume as usual that the array $a[1], \ldots, a[n]$ contains n different numbers but this time they are in order. 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 comparisms, 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.

The first algorithm we consider again 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 abord 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

$$\Pr(K = k) = \frac{1}{n+1} \text{ för } 1 \leq k \leq n+1$$

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 $1 \leq K \leq n$. In that case the probability is exactly $\Pr(K = k) = 1/n$. Consequently we get

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

```
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: Lineare Suche in geordnetem Feld.

We can expect that the successful and unsuccessful case are not very different in regard of the running time.

Let us now count the number of executed machine instructions of the corresponding assembler 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.

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

Not suprisingly a binary search tree is a binary tree. It consists either of only one node which we call the root or a root that has two children that are themselves binary trees. We will distinguish between an *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 comparisms 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 comparism 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 comparism 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 comparism is equal then the algorithm will stop the search because the desired element has been found. With other words we assume that every comparism is against the key we are searching for.

As usual we draw binary search trees as a binary tree but we will draw

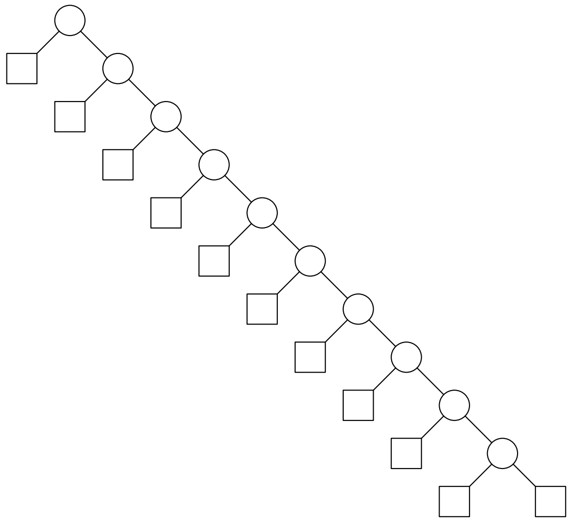


Figure 3.7: Ein Suchbaum für die lineare Suche in einem geordnetem Feld der Länge 10.

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 emphasized size $|T|$, which is exactly the number of internal nodes, the *internal path lengths* $\pi(T)$ and the *external path lengths* $\xi(t)$.

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 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 comparisms that an algorithm performs.

Theorem 5. Let T be the comparism 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 number of comparisms 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 on the 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 comparisms 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 comparisms 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 did we make a 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 comparism is no longer necessary. Still the program carries out another comparism with $a[n+1]$. This is not a comparism according to our definition because *there is only one possible answer and the comparism is redundant.* The theory of comparism trees works only if the probability of reaching every external node in an unsuccessful search is the same. In this case this is not true if there are redundant comparisms. The last comparism cannot be spotted in the search tree we drew: The search tree contains only n internal nodes labeled with the comparisms $a[1] : v, \ldots, a[n] : v$. If the algorithm visits the last external node in the tree it performs another redundant comparism and that happens with the probability of $1/(n+1)$ in an unsuccessful search. The real number of comparisms 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 comparism 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) 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 comparisms are only correct if all preconditions are met. We have to check them carefully and have to take any exceptions into consideration.

Binary search

Should the array be already 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 left and r is the right most element of this subarray. The algorithm compares with the key in the middle. If it is the correct one, the algorithm immediately terminates. Otherwise the right or left order 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 comparisms $v : a[i]$. 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 this definitions in mind $C_1 = 1$ because $N = 1$ implies that $r = l$ and after computing `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 $r := \lceil (r + l)/2\rceil - 1$ or $l := \lceil (r + l)/2\rceil + 1$ will be executed. In both cases this implies $N := \lfloor N/2\rfloor$; as we expected the size of the active subarray is

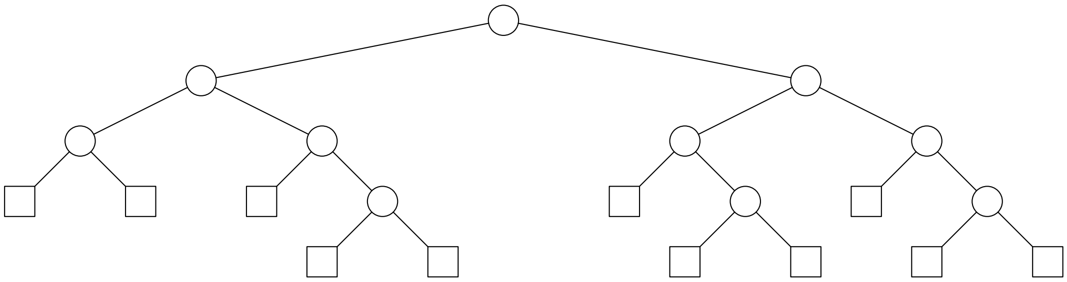


Figure 3.8: Ein Suchbaum für die binäre Suche in einem geordnetem Feld der Länge 10.

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 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 with 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's 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 resultis can be easily checked with the help of the binary search tree in figure 3.8.

Exercises

3.1 Lösen Sie die folgende Rekursionsgleichung und finden Sie eine besonders schöne Art, die ermittelte Lösung aufzuschreiben.

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

3.2 Drill Sergeant Even hat schlechte Laune und neue Rekruten, die er in Zweierreihen über den Kasernenhof marschieren lassen will. Deshalb flippt Even immer aus, wenn die Anzahl der Rekruten ungerade ist, und scheucht die Soldaten über die **DEATH LANE**. Dabei erkrankt jeder Rekrut mit Wahrscheinlichkeit $\frac{1}{2}$ und fällt somit für den Rest des Drills aus. Dieses unwürdige Spektakel wiederholt sich solange, bis schließlich eine gerade Anzahl von gesunden Soldaten übrig ist.
Wieviele Scheuch-Iterationen werden im Durchschnitt durchgeführt?

3.3 Lösen Sie die folgende Rekursionsgleichung! Es sei $a_0 = 0$, $a_1 = 3$ und $a_n = 4a_{n-1} - 4a_{n-2}$ für $n > 1$.

3.4 Lösen Sie die folgende Rekursionsgleichung! Es sei $b_1 = b_2 = b_3 = 1$ und $b_n = 3b_{n-1} - 4b_{n-2} + 12b_{n-3}$ für $n > 3$.

3.5 Vergleichen Sie die Lösung $2(nH_n) + 2(H_n) - 2(n) = 2nH_n + 2H_n - 2n$ von Seite 40 mit der allgemeinen Lösung aus dem ersten Kapitel, indem Sie dort $M = 0$ setzen.

3.6 Ein Algorithmus betrachtet alle Indexpaare $i < j$ eines Arrays mit n Zellen und ruft sich dann selbst auf allen echten Präfixen des Arrays (rekursiv) auf. Wieviele Paare werden insgesamt betrachtet, Wiederholungen mit eingerechnet? Verwenden Sie die Repertoire-Methode.

3.7 Wenden Sie einen Summationsfaktor auf (3.4) an und finden Sie schließlich die Lösung der ursprünglichen Rekursionsgleichung (3.3) zwar nicht in geschlossenerer Form, aber als eine Summe.

3.8 Lösen Sie die Rekursion

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

durch Reduktion der Ordnung.

3.9 Bestimmen Sie die Anzahl der Schleifendurchläufe für das folgende Programmstück, wenn zu Beginn $0 < i$ gilt:

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

3.10 Bearbeiten Sie den Fall $i \leq 0$ für die letzte Aufgabe.

3.11 Analysieren Sie die Laufzeit der erfolgreichen Suche, wenn jedes Element genau *zweimal* im Array vorkommt und wieder jede Permutation gleich wahrscheinlich ist.

3.12 Vergleichen Sie alle drei Suchalgorithmen bezüglich der erfolglosen Suche.

3.13 Wir betrachten folgenden Algorithmus zur Suche von Element x in einem sortierten Feld a der Länge $n = km + 1$:

```
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) Zeichnen Sie den Suchbaum und berechnen Sie die interne und externe Pfadlänge für $n = 10; m = 3$.

b) Bestimmen Sie C^+ und C^- für beliebige m, k.

c) Wie sollte m für ein gegebenes n gewählt werden, um die Laufzeit zu optimieren?

3.14 Verifizieren Sie, daß die Behauptung $N := \lfloor N/2 \rfloor$ auf Seite 57 wirklich stimmt.

3.15 Wir möchten diese beiden Programme für die Suche in einem geordneten Feld vergleichen:

We want to compare these two programs for the search in a sorted field:

```
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;
}
```

Analysieren Sie, wie viele **if**-Anweisungen in den Programmen bei einer erfolgreichen beziehungsweise erfolglosen Suche nach einem Element v ausgeführt werden. Bei *binsearch* ist eine exakte Lösung, bei *binsearch2* lediglich eine Näherung der Form $f(n) + O(1)$ gesucht.

Analyse how many **if**-instructions are exported by the programs in case of a successful or unsuccessful search for v.

Annahmen:

Assumptions:

- *Das Feld enthält* n unterschiedliche *Elemente.*
- The field contains n *differential* elements.
- *Bei der erfolgreichen Suche wird jedes Feldelement mit gleicher Wahrscheinlichkeit gesucht.*
- In case of the successful search every field element with equal probability is quested.
- *Bei der erfolglosen Suche wird* v *ebenfalls zufällig gewählt und fällt jeweils mit Wahrscheinlichkeit* $\frac{1}{n+1}$ *in eine der* $n+1$ *Lücken.*
- In case of the unsuccessful search n is also selected accidentally and each one falls into one of the gaps with the probability $\frac{1}{n+1}$.

Chapter 4

Generating functions

Wir haben uns oft mit Folgen $(g_0, g_1, g_2, \ldots)$ beschäftigt, wobei die einzelnen Glieder g_n den Verbrauch einer Ressource oder eine andere kombinatorische Größe angeben. Oft ist die Folge implizit, z.B. durch eine Rekursionsgleichung gegeben. Wir nennen

$$G(z) = \sum_{n=0}^{\infty} g_n z^n$$

die *Erzeugende Funktion (Generating Funktion, GF)* der Folge $(g_n)_{n=0}^{\infty}$. Die fundamentale Aufgabe in der Analyse von Algorithmen ist es, einen expliziten Ausdruck für g_n oder eine gute Näherung dafür zu finden, wenn die Folge nur implizit gegeben ist. Erzeugende Funktionen sind das wichtigste Werkzeug hierfür, welches wir kennenlernen werden. We were often occupying ourselves with the sets $(g_0, g_1, g_2, \ldots)$, where the single coefficients g_n indicate the usage of a ressource or another combinatorial size. This set is often implicit, e.g. given by a recursion equation. We name

$$G(z) = \sum_{n=0}^{\infty} g_n z^n$$

the *Generating Funktion, GF* of the set $(g_n)_{n=0}^{\infty}$. The fundamentale task in the analysis of algorithms is to find an explicit expression for g_n or a good approximation if the set is only given explicitly. Generating functions are the most important tool for this purpose, which we will get to know.

Ein wichtiger Teilschritt in vielen Analysen ist es, aus $G(z)$ das $n$te Folgenglied zu extrahieren. Theoretisch kann man $G(z)$ in eine Taylorreihe

Folge	a_n	OGF
$1, 0, 0, 0, 0, 0, 0, \ldots$	$(n = 0)$	1
$0, 1, 0, 0, 0, 0, 0, \ldots$	$(n = 1)$	z
$1, 1, 1, 1, 1, 1, 1, \ldots$	1	$\frac{1}{1-z}$
$0, a, a^2, a^3, a^4, a^5, \ldots$	a^n	$\frac{1}{1-az}$
$1, 2, 3, 4, 5, 6, 7, 8, \ldots$	$n+1$	$\frac{1}{(1-z)^2}$
$\binom{r}{0}, \binom{r}{1}, \binom{r}{2}, \binom{r}{3}, \ldots$	$\binom{r}{n}$	$(1+z)^r$
$\binom{k}{k}, \binom{k+1}{k}, \binom{k+2}{k}, \binom{k+3}{k}, \ldots$	$\binom{k+n}{k}$	$\frac{1}{(1-z)^{k+1}}$
$1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \ldots$	$\frac{1}{n!}$	e^z

Table 4.1: Important generating functions and their sets.

entwickeln,

$$G(z) = G(0) + zG'(0) + \frac{z^2}{2}G''(0) + \frac{z^3}{3!}G'''(0) + \ldots,$$

woraus sich

$$[z^n]G(z) = \frac{d^n}{dz^n}\frac{G(z)}{n!}\bigg|_{z=0}$$

ablesen läßt. Mit $[z^n]G(z)$ bezeichnen wir den Koeffizienten vor z^n in der Potenzreihe $G(z)$. Dieser Weg ist gewöhnlich zu steinig und es gibt in der Regel bessere Methoden den $n$ten Koeffizienten zu extrahieren. Eine Tabelle mit wichtigen, bekannten erzeugenden Funktionen ist dabei sehr nützlich. Tabelle 4.1 enthält die wichtigsten. An important sub-step in many parts of analysis is to extract the n-th coefficient out of $G(z)$. Theoretically, one can develop $G(z)$ in a Taylor series,

$$G(z) = G(0) + zG'(0) + \frac{z^2}{2}G''(0) + \frac{z^3}{3!}G'''(0) + \ldots,$$

where

$$[z^n]G(z) = \frac{d^n}{dz^n}\frac{G(z)}{n!}\bigg|_{z=0}$$

can be read off from. By $[z^n]G(z)$ we indicate the coefficient before z^n in the power series $G(z)$. This way is usually too stony and normally there are better methods to extract the n-th coefficient. A table with important known generation functinos is very useful thereby. Table 4.1 contains the most important ones.

Mit Hilfe von Tabelle 4.1 und 4.2 und anderen Formelsammlungen lassen sich viele Funktionen in Reihen entwickeln. Man muß die erzeugenden Funktionen nur erst entsprechend umformen, damit sie zu einem Eintrag in den Tabellen passen.

Wenden wir unsere bisherigen Kenntnisse an einer einfachen Rekursionsgleichung an, der Rekursionsgleichung für Fibonacci-Zahlen:

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \text{ für } n > 1$$

Der erste Schritt, um eine solche Rekursionsgleichung anzugehen, ist es eine einzige Formel zu finden, welche F_n für alle n richtig definiert. Wir

$$A(z) = \sum_{n=0}^{\infty} a_n z^n \qquad B(z) = \sum_{n=0}^{\infty} b_n z^n$$

$zA(z) =$	$\displaystyle\sum_{n=0}^{\infty} a_{n-1} z^n$	(Rechts shift)
$\dfrac{A(z) - a_0}{z} =$	$\displaystyle\sum_{n=0}^{\infty} a_{n+1} z^n$	(Links shift)
$A'(z) =$	$\displaystyle\sum_{n=0}^{\infty} (n+1) a_{n+1} z^n$	(Differenzieren)
$\displaystyle\int_0^z A(t)\,dt =$	$\displaystyle\sum_{n=1}^{\infty} \frac{a_{n-1}}{n} z^n$	(Integrieren)
$A(\lambda z) =$	$\displaystyle\sum_{n=0}^{\infty} \lambda^n a_n z^n$	(Skalieren)
$A(z) + B(z) =$	$\displaystyle\sum_{n=0}^{\infty} (a_n + b_n) z^n$	(Addition)
$(1 - z)A(z) =$	$\displaystyle a_0 + \sum_{n=0}^{\infty} (a_n - a_{n-1}) z^n$	(Differenz)
$A(z)B(z) =$	$\displaystyle\sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k b_{n-k} \right) z^n$	(Faltung)
$\dfrac{A(z)}{1-z} =$	$\displaystyle\sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k \right) z^n$	(Partialsumme)

Table 4.2: Einige Operationen mit erzeugenden Funktionen. Wir definieren $a_n = b_n = 0$ für $n < 0$.

wollen also die Fallunterscheidung loswerden. Wir verwenden dazu folgende Konvention:

$$(\textit{Bedingung}) = \begin{cases} 1 & \text{falls } \textit{Bedingung} \text{ wahr} \\ 0 & \text{sonst} \end{cases}$$

Die Rekursionsgleichung $F_n = F_{n-1} + F_{n-2}$ ist nur für $n > 1$ gültig, für $n = 1$ ist sie aber falsch, denn $F_1 = 1$, aber $F_0 + F_{-1} = 0$ (wir nehmen $F_n = 0$ an, für $n < 0$). Zufällig stimmt aber $F_n = F_{n-1} + F_{n-2}$ auch für $n = 0$. Folgende Formel ist daher für alle n richtig:

$$F_n = F_{n-1} + F_{n-2} + (n = 1) \tag{4.1}$$

Wir interessieren uns nun für die erzeugende Funktion

$$G(z) = \sum_{n=0}^{\infty} F_n z^n.$$

Um $G(z)$ zu erhalten, multiplizieren wir beide Seiten von (4.1) mit z^n und summieren über n von 0 bis ∞:

$$\sum_{n=0}^{\infty} F_n z^n = \sum_{n=0}^{\infty} F_{n-1} z^n + \sum_{n=0}^{\infty} F_{n-2} z^n + \sum_{n=0}^{\infty} (n = 1) z^n \tag{4.2}$$

Die letzte Summe ist einfach z und die anderen Summen lassen sich leicht umformen und wir erhalten

$$G(z) = zG(z) + z^2 G(z) + z. \tag{4.3}$$

Wir erhalten diese Gleichung auch schneller direkt aus (4.1) indem wir die Umformungsregeln aus Tabelle 4.2 verwenden. Die ersten beiden Terme der rechten Seite F_{n-1} und F_{n-2} sind die Folge F_n um 1 bzw. 2 nach rechts verschoben und daher sind ihre erzeugenden Funktionen einfach $zG(z)$ und $z^2G(z)$. Die erzeugende Funktion von $(0, 1, 0, 0, 0, 0, \ldots)$ ist z, denn diese Folge ist die Folge $(1, 0, 0, 0, 0, 0, \ldots)$ um 1 nach rechts verschoben. Die letzte Folge findet sich wieder in Tabelle 4.1. Die erzeugende Funktion der rechten Seite ist jetzt einfach die Summe der drei Funktionen (Additionsregel). Auf diese Weise erhalten wir eine algebraische Gleichung für $G(z)$. Unter Umständen können wir auch eine Differentialgleichung oder auch eine Integral- oder Integro-Differentialgleichung.

Lösen wir (4.3) nach $G(z)$ auf, dann erhalten wir die Lösung der algebraischen Gleichung und damit eine explizite Formel für $G(z)$:

$$G(z) = \frac{z}{1 - z - z^2}$$

Jetzt müssen wir $G(z)$ in eine Potenzreihe entwickeln. Dazu formen wir $G(z)$ so um, daß wir es in Tabelle 4.1 finden. Wir machen eine Partialbruchzerlegung der rationalen Funktion $1/(1 - z - z^2)$. Hierzu brauchen wir die Nullstellen von $z^2 + z - 1$, welche

$$\frac{1}{\phi} = \frac{\sqrt{5} - 1}{2} \text{ und } \frac{1}{\hat{\phi}} = \frac{-\sqrt{5} - 1}{2}$$

sind. Wir können also

$$\frac{1}{1 - z - z^2} = \frac{A}{1 - \phi z} + \frac{B}{1 - \hat{\phi} z}$$

schreiben, wobei wir die Parameter A und B noch bestimmen müssen.

Setzen wir $z = 0$ erhalten wir $1 = A + B$, also ist $B = 1 - A$. Setzen wir $z = 1$ erhalten wir $-1 = A/(1-\phi) + (1-A)/(1-\hat{\phi})$. Nun vereinfachen wir. Aus $1/(1-\phi) = -\phi$ und $1/(1-\hat{\phi}) = -\hat{\phi}$ erhalten wir $1 = \phi A + \hat{\phi} - \hat{\phi} A = \hat{\phi} + \sqrt{5}A$, denn $\phi - \hat{\phi} = \sqrt{5}$. Schließlich erhalten wir $A = \phi/\sqrt{5}$.

Aus Tabelle 4.1 lesen wir ab, daß

$$\frac{1}{1 - \phi z} = \sum_{n=0}^{\infty} \phi^n z^n \text{ und } \frac{1}{1 - \hat{\phi} z} = \sum_{n=0}^{\infty} \hat{\phi}^n z^n$$

ist. Insgesamt erhalten wir

$$G(z) = \frac{z}{1 - z - z^2} = Az \sum_{n=0}^{\infty} \phi^n z^n + Bz \sum_{n=0}^{\infty} \hat{\phi}^n z^n,$$

woraus sich sofort der $n$te Koeffizient ablesen läßt:

$$[z^n]G(z) = A\phi^{n-1} + B\hat{\phi}^{n-1} = \frac{1}{\sqrt{5}} \left(\phi^n + \hat{\phi}^n \right)$$

Dieses Ergebnis hätten wir natürlich auch leicht durch konventionelle Methoden gewinnen können, denn die Rekursionsgleichung war linear mit konstanten Koeffizienten. Dieses Beispiel diente aber nur zur Einführung der erzeugenden Funktionen.

Wir werden die Formeln in den Tabellen 4.1 und 4.2 nicht alle beweisen. Begnügen wir uns in Tabelle 4.1 mit einem Beweis der letzten beiden Formeln.

Zunächst haben wir dort die Folge

$$0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots$$

Die GF dieser Folge ist dann nach Definition und der Regel *Integrieren* in Tabelle 4.2

$$\sum_{n=1}^{\infty} \frac{1}{n} z^n = \int_0^z \frac{1}{1-t}\, dt,$$

denn $1/(1-z)$ ist die GF der Folge $(1, 1, 1, 1, \ldots)$. Wir können dieses Integral mit der Formel

$$\int \frac{f'(t)}{f(t)} dt = \ln(f(t)) + C$$

lösen und erhalten als GF

$$\int_0^z \frac{1}{1-t}\, dt = \ln \frac{1}{1-z}.$$

Als nächstes nehmen wir uns die Folge

$$H_0, H_1, H_2, H_3, H_4, \ldots$$

vor. Schreiben wir die GF aus, mit der Definition von H_n eingesetzt erhalten wir

$$\sum_{n=0}^{\infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) z^n.$$

Dieser Ausdruck ist ein Spezialfall der Regel *Partialsumme* mit $a_k = 1/k$ für $k > 0$ und $a_0 = 0$. Die Folge a_k haben wir eben behandelt und ihre GF ist $\ln(1/(1-z))$. Daher erhalten wir wie erwartet

$$\sum_{n=0}^{\infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) z^n = \frac{1}{1-z} \ln \frac{1}{1-z}.$$

4.1 Zählen mit erzeugenden Funktionen

Mit erzeugenden Funktionen können wir die Anzahl von Gegenständen einer bestimmten Größe in einer Menge zählen. Wir wenden dies am Beispiel binärer Bäume an.

Ein binärer Baum besteht entweder aus nur einem externen Knoten oder aus einem internen Knoten, welcher selbst zwei binäre Bäume als Kinder besitzt. Wir sind in der Anzahl der binären Bäume mit einer bestimmten Anzahl interner Knoten interessiert. Wir definieren also die erzeugende Funktion

$$T(z) = \sum_{n=0}^{\infty} t_n,$$

wobei t_n die Anzahl binärer Bäume mit n interner Knoten sind.

Wir erhalten für t_n die Rekursionsgleichung

$$t_n = \sum_{k=0}^{n-1} t_k t_{n-1-k} + (n = 0).$$

Durch Multiplikation beider Seiten mit z^n und Summieren über n ergibt sich daraus

$$\begin{aligned} T(z) &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n-1} t_k t_{n-1-k} \right) z^n + 1 \\ &= z \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} t_k t_{n-k} \right) z^n + 1 \\ &= zT(z)^2 + 1 \end{aligned}$$

Wir kommen aber viel einfacher auf diese Beziehung: Wir denken uns T als die Menge aller binären Bäume. Dann ist $T = E + ITT$. Wir abstrahieren aber und interessieren uns nur für die Anzahl interner Knoten. Das heißt wir ersetzen E durch z^0 und I durch z^1. Statt T schreiben wir $T(z)$. Das ergibt:

$$T(z) = zT(z)^2 + 1$$

Wir müssen nun $T(z)$ in eine Reihe entwickeln, um $t_n = [z^n]T(z)$ ablesen zu können.

Zuerst sollten wir die Gleichung nach $T(z)$ auflösen, was nicht schwer ist, da es eine quadratische Gleichung ist. Wir erhalten

$$T(z) = \frac{1}{2z} \pm \frac{1}{2z}\sqrt{1-4z}.$$

Es gibt natürlich zwei Lösungen, denn es war ja eine quadratische Gleichung, aber unser Problem hat nur eine Lösung. Wie ist das möglich? Ganz einfach: Wir lösen eine funktionale Gleichung für $T(z)$. Die eine Lösung der Gleichung ist eine im Ursprung analytische Funktion, welche sich in eine Potenzreihe entwickeln läßt, während die andere Lösung im Ursprung nicht analytisch ist. Wir sehen schnell, welche Lösung die richtige ist. Entwickeln wir zunächst $\sqrt{1-4z}$ in eine Potenzreihe, was direkt mit der Newtonschen Formel geht.

$$T(z) = \frac{1}{2z} \pm \frac{1}{2z}\sqrt{1-4z} = \frac{1}{2z} \pm \frac{1}{2z}\sum_{n=0}^{\infty}\binom{1/2}{n}(-4)^n z^n$$

Jetzt sehen wir, das wir die Lösung mit dem Minuszeichen wählen müssen, denn dann verschwindet der Pol bei $z = 0$.

$$T(z) = -\frac{1}{2z}\sum_{n=1}^{\infty}\binom{1/2}{n}(-4)^n z^n = -\frac{1}{2}\sum_{n=0}^{\infty}\binom{1/2}{n+1}(-4)^{n+1} z^n$$

Wir lesen das Ergebnis ab:

$$t_n = [z^n]T(z) = -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1}$$

Es kann nichts schaden, diese Formel an einem kleinen Beispiel zu überprüfen, zum Beispiel $n = 3$.

$$t_3 = -\frac{1}{2}\binom{1/2}{n+4}(-4)^4 = 5$$

Das Ergebnis läßt sich noch vereinfachen, aber statt dessen gehen wir noch einen anderen Weg, um zu diesem Ergebnis zu kommen.

Unangenehm an der Herleitung war die quadratische Gleichung. Wir lernen nun einen wichtigen Satz kennen, welcher die Koeffizienten in einer Potenzreihe in Beziehung zu den Koeffizienten in der Potenzreihe der inversen Funktion setzt.

Theorem 6. ***(Lagrange-Inversion)***
Sei $G(z)$ eine GF, für welche $z = f(G(z))$ gilt mit $f(0) = 0$ und $f'(0) \neq 0$. Dann gilt

$$[z^n]G(z) = \frac{1}{n}[u^{n-1}]\left(\frac{u}{f(u)}\right)^n.$$

Direkt auf $T(z)$ können wir die Lagrange-Inversion nicht anwenden, da das resultierende f nicht die nötigen Eigenschaften besitzt. Setzen wir $H(z) = zT(z)$ geht es aber sehr gut. Die entsprechende funktionale Gleichung ist dann

$$H(z) = z + H(z)^2,$$

welche sich sehr leicht nach z auflösen läßt, aber wieder nicht so leicht nach $H(z)$. Wir haben $z = f(H(z))$ mit $f(t) = t - t^2$. Mit $f(0) = 0$ und $f(0) = 1$ sind die Bedingungen für die Lagrange-Inversion erfüllt:

$$[z^n]H(z) = \frac{1}{n}[u^{n-1}]\left(\frac{1}{1-u}\right)^n$$

Eine Formel aus Tabelle 4.1 paßt zur rechten Seite,

$$\frac{1}{(1-u)^n} = \sum_{k=0}^{\infty}\binom{k+n-1}{k}z^k$$

so daß sich

$$[z^n]H(z) = \frac{1}{n}\binom{2n-2}{n-1}$$

ergibt. Für $T(z)$ heißt das

$$t_n = \frac{1}{n+1}\binom{2n}{n}.$$

Überprüfen wir diese Formel wieder für $n = 3$:

$$t_3 = \frac{1}{4}\binom{6}{3} = 5$$

4.2 Bivariate erzeugende Funktionen

Bisher betrachteten wir GFs mit einer Variable z, welche mit einer Folge korrespondiert. Wir können dies selbstverständlich auf GFs mit mehreren

Variablen verallgemeinern. Diese entsprechen dann mehrdimensionalen Folgen. Im Falle zweier Variablen nennen wir sie *bivariate erzeugende Funktionen (Bivariate Generating Functions, BGFs)*.

Rechnen wir ein einfaches Beispiel mit BGFs, für welches wir das Ergebnis bereits kennen. Wieviele Strings gibt es über dem Alphabet $\{0,1\}$, welche genau m 1en enthalten und n Zeichen lang sind?

Wir können die Menge aller Bitstrings rekursiv folgendermaßen definieren:

$$B = \{\epsilon\} \cup 0B \cup 1B$$

Definieren wir die BGF

$$B(u,z) = \sum_{n,m\geq 0} b_{mn} u^m z^n,$$

wobei b_{mn} die Anzahl der Bitstrings der Länge n mit genau m 1en bezeichnet, dann erhalten wir

$$B(u,z) = 1 + zB(u,z) + uzB(u,z).$$

Wir lösen nach $B(u,z)$ auf und entwickeln das Ergebnis in eine Reihe:

$$B(u,z) = \frac{1}{1 - z(1+u)} = \sum_{n=0}^{\infty} z^n (1+u)^n = \sum_{n=0}^{\infty} \sum_{k=0}^{n} \binom{n}{k} z^n u^k$$

Nun können wir $b_{mn} = [u^m z^n]B(u,z) = \binom{n}{m}$ ablesen.

4.3 Exponentielle erzeugende Funktionen

Wir definieren zu einer Folge $g_0, g_1, g_2, g_3, \ldots$ die *exponentielle erzeugende Funktion (Exponential Generating Function, EGF)*

$$G(z) = g_0 \frac{1}{1!} + g_1 \frac{z}{2!} + g_2 \frac{z^2}{3!} + g_2 \frac{z^3}{3!} + \cdots = \sum_{n=0}^{\infty} g_n \frac{z^n}{n!}.$$

Der $n$te Koeffizient in der EGF $G(z)$ von $(g_n)_{n=0}^{\infty}$ ist dann natürlich

$$g_n = n![z^n]G(z).$$

In manchen Fällen kommt man mit einer EGF leichter zum Ziel als mit einer GF und manchmal ist es umgekehrt. Der Unterschied besteht darin, daß es für verschiedene Folgen verschieden einfache GFs und EGFs gibt und daß sich EGFs verschiedenen transformieren, wenn man die Folge verändert. Die Tabelle 4.3 enthält EGFs für mehrere oft vorkommende Folgen und Tabelle 4.4 enthält Transformationsregeln für EGFs. Diese beiden Tabellen entsprechen den Tabellen 4.1 und 4.2 für GFs.

4.4 Die symbolische Methode

In diesem Unterkapitel lernen wir etwas systematischer, wie man Objekte mit erzeugenden Funktionen zählen kann, ohne den Umweg über eine Rekursionsgleichung zu machen. Wir konzentrieren uns dabei auf *rekursive* Objekte.

Für die Konstruktion von Mengen von Objekten lassen wir folgende Möglichkeiten zu, wobei $\mathcal{M}$ die ganze Menge bezeichnet und $\mathcal{M}_1$ und $\mathcal{M}_2$ bereits definierte Mengen sind.

1. atomares Objekt: $\mathcal{M} = \{x\}$
2. Paarbildung: $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$
3. Vereinigung: $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$
4. Folgen von Objekten: $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_1 \times \mathcal{M}_1 \cup \mathcal{M}_1 \times \mathcal{M}_1 \times \mathcal{M}_1 \cup \cdots$

Sagen wir, die Größe eines Objekts ist die Anzahl der Atome, die es enthält. Eine genaue Definition der Größe können wir durch den rekursiven Aufbau der Objekte angeben:

$$\begin{aligned} |x| &= f(x) && \text{(atomares Objekt)} \\ |(x, y)| &= |x| + |y| && \text{(Paarbildung)} \\ |(x_1, x_2, \ldots, x_m)| &= |x_1| + \cdots + |x_m| && \text{(Folgen)} \end{aligned}$$

Die Größe eines atomaren Objekts können wir dabei beliebig festlegen. Wir sind nun aber interessiert in der Anzahl von Objekten einer bestimmten Größe in einer gegebenen Menge von Objekten.

Folge	a_n	OGF
$1, 0, 0, 0, 0, 0, 0, \ldots$	$(n = 0)$	1
$0, 1, 0, 0, 0, 0, 0, \ldots$	$(n = 1)$	z
$1, 1, 1, 1, 1, 1, 1, \ldots$	1	e^z
$1, c, c^2, c^3, c^4, c^5, c^6, \ldots$	c^n	e^{cz}
$0, 1, 2, 3, 4, 5, 6, \ldots$	n	ze^z
$\binom{0}{m}, \binom{1}{m}, \binom{2}{m}, \ldots$	$\binom{n}{m}$	$\frac{z^m}{m!}e^z$
$1, 0, 1, 0, 1, 0, 1, \ldots$	$1 + (-1)^n$	$\cosh(z) = \frac{1}{2}(e^z + e^{-z})$
$0, 1, 0, 1, 0, 1, 0, \ldots$	$1 - (-1)^n$	$\sinh(z) = \frac{1}{2}(e^z - e^{-z})$
$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \ldots$	$\frac{1}{n+1}$	$\frac{e^z - 1}{z}$
$0!, 1!, 2!, 3!, 4!, 5!, 6!, \ldots$	$n!$	$\frac{1}{1-z}$

Table 4.3: Wichtige EGFs und ihre Folgen.

$$A(z) = \sum_{n=0}^{\infty} a_n \frac{z^n}{n!} \qquad B(z) = \sum_{n=0}^{\infty} b_n \frac{z^n}{n!}$$

$$\int_0^z A(t)\,dt = \sum_{n=0}^{\infty} a_{n-1} \frac{z^n}{n!} \qquad \text{(Rechts shift)}$$

$$A'(z) = \sum_{n=0}^{\infty} a_{n+1} \frac{z^n}{n!} \qquad \text{(Links shift)}$$

$$zA(z) = \sum_{n=0}^{\infty} n a_{n-1} \frac{z^n}{n!} \qquad \text{(Indexmultiplikation)}$$

$$\frac{A(z) - A(0)}{z} = \sum_{n=1}^{\infty} \frac{a_{n+1}}{n+1} \frac{z^n}{n!} \qquad \text{(Indexdivision)}$$

$$A(\lambda z) = \sum_{n=0}^{\infty} \lambda^n a_n \frac{z^n}{n!} \qquad \text{(Skalieren)}$$

$$A(z) + B(z) = \sum_{n=0}^{\infty} (a_n + b_n) \frac{z^n}{n!} \qquad \text{(Addition)}$$

$$A'(z) - A(z) = \sum_{n=0}^{\infty} (a_{n+1} - a_n) \frac{z^n}{n!} \qquad \text{(Differenz)}$$

$$A(z)B(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} \right) \frac{z^n}{n!} \qquad \text{(Binomialfaltung)}$$

$$e^z A(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} \binom{n}{k} a_k \right) \frac{z^n}{n!} \qquad \text{(Binomialsumme)}$$

Table 4.4: Einige Operationen mit EGFs. Wir definieren wieder $a_n = b_n = 0$ für $n < 0$.

Sei $M(z)$ die erzeugende Funktion der Folge, deren $n$tes Glied die Anzahl der Objekte der Größe n in der Menge $\mathcal{M}$ angibt und $M_1(z)$ und $M_2(z)$ die entsprechenden erzeugenden Funktionen für die Mengen $\mathcal{M}_1$ und $\mathcal{M}_2$. Dann gilt:

$$\begin{aligned}
\mathcal{M} = \emptyset &\Longrightarrow M(z) = 0 \\
\mathcal{M} = \{x\} &\Longrightarrow M(z) = z^{|x|} \\
\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2 &\Longrightarrow \mathcal{M}(z) = \mathcal{M}_1(z) + \mathcal{M}_2(z) \text{ falls } \mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset \\
\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 &\Longrightarrow \mathcal{M}(z) = \mathcal{M}_1(z)\mathcal{M}_2(z) \\
\mathcal{M} = \bigcup_{k=1}^{\infty} \mathcal{M}_1^k &\Longrightarrow \mathcal{M}(z) = \frac{\mathcal{M}_1(z)}{1 - \mathcal{M}_1(z)}
\end{aligned}$$

Beispiel: Binärbäume lassen sich definieren als

$$\mathcal{B} = \square \cup \bigcirc \times \mathcal{B} \times \mathcal{B},$$

mit $|\square| = 0$ und $|\bigcirc| = 1$. Daraus erhalten wir folgende Funktionalgleichung für die erzeugende Funktion $B(z)$:

$$B(z) = 1 + zB(z)^2$$

Wir haben diese Gleichung schon früher gelöst.

Beispiel: Geordnete Bäume mit beliebigem Grad können wir definieren als

$$\mathcal{T} = \square \cup \bigcirc \times \bigcup_{k=1}^{\infty} \mathcal{T}^k,$$

woraus sich sofort folgende Gleichung ergibt:

$$T(z) = 1 + \frac{zT(z)}{1 - T(z)}$$

Lösen wir die Gleichung nach $T(z)$ auf erhalten wir

$$T(z) = 1 - \frac{z}{2} \pm \frac{1}{2}\sqrt{z(z-4)}.$$

Leider ist diese Funktion im Ursprung nicht analytisch und besitzt keine Entwicklung in eine Potenzreihe. Wie ist das möglich?

Zählen wir einfach einmal selbst nach. Es gibt nach unserer Definition genau einen Baum der Größe 0, nämlich den Baum, der nur aus einem externen Knoten besteht. Wieviele Bäume der Größe 1 gibt es? Aha, offensichtlich gibt es hier unendlich viele, nämlich die Wurzel mit einem, zwei, drei, usw. externen Knoten als Kinder. Mit erzeugenden Funktionen können wir nur Folgen darstellen, deren Glieder alle endlich sind.

Formulieren wir also die Frage um: Wieviele Bäume gibt es mit n Knoten, wobei wir interne und externe Knoten zählen?

Jetzt erhalten wir die Funktionalgleichung

$$T(z) = z + \frac{zT(z)}{1 - T(z)},$$

welche so aussieht, wenn wir sie nach $T(z)$ auflösen:

$$T(z) = \frac{1}{2} \pm \frac{1}{2}\sqrt{1 - 4z}$$

In eine Potenzreihe entwickelt:

$$T(z) = \frac{1}{2} + \frac{1}{2}\sum_{n=0}^{\infty} \binom{1/2}{n} (-4)^n z^n$$

Für $n > 0$ erhalten wir also

$$[z^n]T(z) = -\frac{1}{2}\binom{1/2}{n}(-4)^n =$$

Wir haben bisher angenommen, daß sich alle Atome in einem zusammengesetzten Objekt sich nicht unterscheiden. Zum Beispiel erhalten wir je ein Objekt der Größe n für folgende rekursive Definition:

$$\mathcal{A} = \{x\} \cup \{x\} \times \mathcal{A}$$

Die Menge $\mathcal{A}$ besteht hier einfach aus allen n-Tupeln $(x, x, x, \ldots, x)$ und für jede Größe gibt es genau eines.

Würden sich aber die Atome unterscheiden, hätten wir $n!$ verschiedene n-Tupel.

Wir nennen die bisher untersuchten Objekte *unmarkierte Objekte* und Objekte, in welchen sich die Atome unterscheiden *markierte Objekte*.

Für diese *markierten Objekte* bieten sich EGFs an, denn es gibt um einen Faktor $n!$ mehr markierte Objekte als entsprechende unmarkierte Objekte (wenn das Objekt insgesamt n Atome enthält).

Wir können zur Anschauung beispielsweise annehmen, daß die Atome in einem Objekt gerade $x_1, x_2, \ldots, x_n$ sind. Wenn wir diese Vorstellung haben, müssen beim Zusammensetzen von Objekten diese Bezeichnungen natürlich entsprechend geändert werden.

Sei nun $\hat{M}(z)$ die EGF

$$\hat{M}(z) = \sum_{n=0}^{\infty} m_n \frac{z^n}{n!},$$

wobei m_n die Anzahl verschiedener markierter Objekte in der Menge $\mathcal{M}$ ist.

$$\begin{aligned}
\mathcal{M} = \emptyset &\Longrightarrow \hat{M}(z) = 0 \\
\mathcal{M} = \{x\} &\Longrightarrow \hat{M}(z) = z^{|x|} \\
\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2 &\Longrightarrow \hat{M}(z) = \hat{M}_1(z) + \hat{M}_2(z) \text{ falls } \mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset \\
\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 &\Longrightarrow \hat{M}(z) = \hat{\mathcal{M}}_1(z)\hat{M}_2(z) \\
\mathcal{M} = \bigcup_{k=1}^{\infty} M_1^k &\Longrightarrow \hat{M}(z) = \frac{\hat{M}_1(z)}{1 - \hat{M}_1(z)}
\end{aligned}$$

Die EGFs verhalten sich also bei markierten Objekten gerade so, wie sich OGFs bei unmarkierten Objekten verhalten.

4.5 Mittlere Stackgröße

Rechnen wir nun ein kleines Beispiel. Wir nehmen an, ein Algorithmus löst ein Problem der Größe n auf folgende Weise: Wenn $n = 1$, dann kann er das Problem direkt lösen. Wenn $n > 1$, dann teilt es das Problem in zwei Teile mit den Größen m und $n - m$ mit $1 \leq m \leq n - 1$ und löst rekursiv zuerst das Problem der Größe m und dann das zweite Problem der Größe $n - m$. Während der Algorithmus das erste Problem löst, merkt er sich auf dem Keller, welches zweite Problem er noch lösen muß. Während er das zweite Problem löst, merkt er sich die Lösung des ersten Problems auf dem Keller. Sind dann beide Probleme gelöst kann der Algorithmus aus beiden Teillösungen direkt die Lösung des ursprünglichen Problems bilden.

Wir interessieren uns dafür, wieviel Speicher für den Keller benötigt wird. Genauer gesagt, wir interessieren uns für den durchschnittlichen Speicherverbrauch im Stack gemittelt über die gesamte Ausführungszeit des Algorithmus. Zusätzlich nehmen wir an, daß jede mögliche Rekursionsstruktur mit gleicher Wahrscheinlichkeit vorkommt (was nicht damit gleichbedeutend ist, daß wenn ein Problem in Teilprobleme zerteilt wird, stets alle Teilungsmöglichkeiten gleich wahrscheinlich sind).

Den Ablauf des Algorithmus können wir uns als Binärbaum vorstellen: Jeder innere Knoten entspricht der Lösung eines Teilproblems, dessen Größe mindestens 2 ist und die externen Knoten Teilproblemen der Größe 1. Der Baum hat also $n-1$ innere Knoten und n externe Knoten. Die Belegung des Kellers während eines Schrittes ist nun proportional zur Tiefe des Knotens dieses Schrittes im Baum. Die mittlere Belegung des Kellers ist daher

$$\frac{\xi(t)+\pi(t)}{2n-1} = \frac{2\pi(t)+2n}{2n-1}$$

für einen Baum t mit n internen Knoten.

Es genügt daher folgende Frage zu beantworten: Wir groß ist $\pi(t)$ durchschnittlich für alle Binärbäume t mit n internen Knoten?

Hierzu berechnen wir einfach

$$p_n = \sum_{t\in\mathcal{T}} \pi(t)(|t| = n),$$

wobei $\mathcal{T}$ die Menge aller Binärbäume ist. Um die durchschnittliche externe Pfadlänge aller Binärbäume mit n externen Knoten zu erhalten müssen wir dann nur noch p_n durch die Anzahl solcher Bäume teilen. Diese Anzahl haben wir bereits berechnet — sie ist $b_n = \frac{1}{n+1}\binom{2n}{n}$.

Wir bestimmen nun p_n. Hierzu sei $P(z)$ die zugehörige OGF,

$$P(z) = \sum_{n=0}^{\infty} p_n z^n = \sum_{t\in\mathcal{T}} \pi(t) z^{|t|}.$$

Ferner werden wir die Anzahl b_n von Binärbäumen der Größe n benötigen und die zugehörige OGF $B(z)$,

$$B(z) = \frac{1-\sqrt{1-4z}}{2z}.$$

Wenn $t \in t_1 \times t_2$, dann $\pi(t) = \pi(t_1) + \pi(t_2) + |t| - 1$. Wir zerlegen $P(z)$ entsprechend:

$$P(z) = \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_1)z^{|t|} + \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_2)z^{|t|} + \sum_{t=t_1\times t_2\in\mathcal{T}} |t|z^{|t|} - \sum_{t=t_1\times t_2\in\mathcal{T}} z^{|t|} = \\ = A + B + C - D,$$

wobei

$$A = \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_1)z^{|t|} \qquad B = \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_2)z^{|t|}$$
$$C = \sum_{t=t_1\times t_2\in\mathcal{T}} |t|z^{|t|} \qquad D = \sum_{t=t_1\times t_2\in\mathcal{T}} z^{|t|}$$

Sehen wir uns nun nacheinander A, B, C und D genauer an. Am einfachsten ist allerdings D, welches sich auf den ersten Blick als das altbekannte $B(z)-1$ herausstellt.

$$A = \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_1)z^{|t|} = \sum_{t=t_1\times t_2\in\mathcal{T}} \pi(t_1)z^{|t_1|+|t_2|+1} = \\ = z\sum_{t_1\in\mathcal{T}} \pi(t_1)z^{|t_1|} \sum_{t_2\in\mathcal{T}} z^{|t_2|} = z\sum_{t\in\mathcal{T}} \pi(t)z^{|t|} \sum_{t\in\mathcal{T}} z^{|t|} = zP(z)B(z)$$

Wir können bei B genauso verfahren und erhalten wegen der Symmetrie zwischen t_1 und t_2 dann $A = B$. Die Potenzreihe C sieht nach einer differenzierten Reihe aus. Wir müssen sie nur ein wenig umformen.

$$C = \sum_{t=t_1\times t_2\in\mathcal{T}} |t|z^{|t|} = z\sum_{t=t_1\times t_2\in\mathcal{T}} |t|z^{|t|-1} = z\Big(\sum_{t=t_1\times t_2\in\mathcal{T}} z^{|t|}\Big)' = zB(z)'.$$

Insgesamt erhalten wir

$$P(z) = 1 + 2zP(z)B(z) + zB(z)' - B(z)$$

oder, aufgelöst nach $P(z)$,

$$\begin{aligned}
P(z) &= \frac{1-1/z}{\sqrt{1-4z}} + \frac{1/z-3}{1-4z} \\
&= \left(1-\frac{1}{z}\right)\sum_{n=0}^{\infty}\binom{-1/2}{n}(-4)^n z^n + \left(\frac{1}{z}-3\right)\sum_{n=0}^{\infty}4^n z^n \\
&= \left(1-\frac{1}{z}\right)\sum_{n=0}^{\infty}\binom{2n}{n}z^n + \left(\frac{1}{z}-3\right)\sum_{n=0}^{\infty}4^n z^n \\
&= \sum_{n=0}^{\infty}\binom{2n}{n}z^n - \sum_{n=0}^{\infty}\binom{2(n+1)}{n+1}z^n - 3\sum_{n=0}^{\infty}4^n z^n + \sum_{n=0}^{\infty}4^{n+1}z^n \\
&= \sum_{n=0}^{\infty}4^n z^n - \sum_{n=0}^{\infty}\frac{3n+1}{n+1}\binom{2n}{n}z^n
\end{aligned}$$

Exercises

4.1 Geben Sie die erzeugenden Funktionen der folgenden Folgen an:

1.	$a_n = 2^n + 3^n$	2.	$b_n = (n+1)2^{n+1}$	3.	$c_n = \alpha^n \binom{k}{n}$
4.	$d_n = (n-1)$	5.	$e_n = (n+1)^2$		

4.2 Finden Sie die erzeugende Funktion für die Folge, welche durch die folgende Rekursionsgleichung gegeben ist: $f_n = f_{n-1} + 2f_{n-2} + 3f_{n-3} + \cdots + nf_0$ für $n > 0$ und $f_0 = 1$.

4.3 Geben Sie die Folgen an, die zu diesen erzeugenden Funktionen gehören:

1. $A(z) = 3^z$
2. $B(z) = 1/\sqrt{1-z/2}$
3. $C(z) = (1+z)/(1-z)$

4.4 Stellen Sie $\binom{1/2}{n}$ als einen Ausdruck dar, der nur ganzzahlige Binomialkoeffizienten enthält.

4.5 Beantworten Sie die folgende Frage mit Hilfe der Lagrange-Inversion: Wie viele verschiedene Ausdrücke lassen sich mit der unten aufgeführten Grammatik (**if**, **then**, **else**, **fi**, **true** und **false** sind ihre Terminalsymbole) in genau n Schritten erzeugen?

$$\begin{aligned}
S \;\rightarrow\; & \textbf{if } S \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \\
& \textbf{if } S \textbf{ then } S \textbf{ fi} \mid \\
& \textbf{true} \mid \\
& \textbf{false}
\end{aligned}$$

4.6 Wir nennen eine Sequenz von **push**- und **pop**-Operationen, kurz $\uparrow$ und $\downarrow$, *gültig*, wenn sie gleich viele $\uparrow$ und $\downarrow$ enthält und kein Präfix der Sequenz weniger $\uparrow$ als $\downarrow$ enthält. Beispielsweise ist $(\uparrow,\uparrow,\downarrow,\downarrow,\uparrow,\downarrow)$ gültig, $(\uparrow,\downarrow,\uparrow,\uparrow)$ und $(\uparrow,\downarrow,\downarrow,\uparrow)$ hingegen sind es nicht. Die Anzahl der $\uparrow$ in einer gültigen Sequenz bezeichnen wir als *Länge* dieser Sequenz.

Wieviele gültige Sequenzen der Länge n gibt es?

4.7 Eine *Spitze* in einer gültigen Sequenz (siehe vorige Aufgabe) ist ein Paar benachbarter Elemente $(\uparrow,\downarrow)$.

Stellen Sie die bivariate erzeugende Funktion für die Anzahl der gültigen Sequenzen der Länge n mit genau m Spitzen auf. Verwenden Sie hierzu zunächst die abstrakte Mengenschreibweise (vergleiche Vorlesung, Tutoraufgabe 2 oder Hausaufgabe 1).

Hinweis: Es könnte hilfreich sein, die beiden Fälle *genau eine Spitze* und *mindestens zwei Spitzen* zunächst getrennt zu betrachten.

4.8 Wie lautet die erzeugende Funktion für die Anzahl der Bäume mit genau n internen und m externen Knoten? Lässt sich eine Symmetrie zur Anzahl der Bäume mit m internen und n externen Knoten nachweisen?

4.9 Berechnen Sie aus der geschlossenen Form von p_n nun die durchschnittliche Stackhöhe. Verwenden Sie dabei die Stirlingsche Formel, um eine asymptotische Näherung von $\binom{2n}{n}$ zu erhalten. Geben Sie das Ergebnis möglichst genau an.

Chapter 5

Asymptotische Abschätzungen

Oft genügt uns eine Näherungslösung, welche nur asymptotisch gültig ist, deren Abweichung von der tatsächlichen Lösung aber auch für kleine Zahlen nur wenig abweicht. Oft ist es gar nicht möglich, die genaue Lösung in geschlossener Form zu erhalten. Manchmal ist dieses wohl möglich, aber die resultierende Formel ist sehr kompliziert. Eine nur asymptotisch gültige, aber einfacher zu durchschauende Formel kann dann nützlicher sein. Ein Beispiel für diesen Fall war die mittlere interne Pfadlänge eines Binärbaums mit n internen Knoten. Die exakte und asymptotische Lösung, welche wir früher erarbeiteten, war

$$4^n - \frac{3n+1}{n+1}\binom{2n}{n} = n\sqrt{\pi n} - 3n + O(\sqrt{n}).$$

Schließlich gibt es noch eine dritte Möglichkeit: Manchmal kann man mit wenig Aufwand eine gute Näherung berechnen, kann die exakte Lösung aber nur mit sehr großem Aufwand herausfinden. Wenn die Näherungslösung genügt, dann lohnt es sich nicht, diesen Aufwand zu betreiben.

Neben der O-Notation verwenden wir noch die Symbole $\sim$ und $\asymp$. Die Definition von $\asymp$ folgt später, und $\sim$ wird auf zwei Arten verwendet.

Wir schreiben

$$f(n) \sim g(n) \iff f(n) = g(n) + o(g(n)).$$

Zum Beispiel ist $\ln(n + 1/n) \sim \ln(n)$ für $n \to \infty$, wobei wir letzteres weglassen, wenn es aus dem Zusammenhang erkennbar ist. Ein weiteres Beispiel ist $e^x \sim 1 + x$ für $x \to 0$.

Figure 5.1: Asymptote-Test

Figure 5.2: Asymptote-Test 2

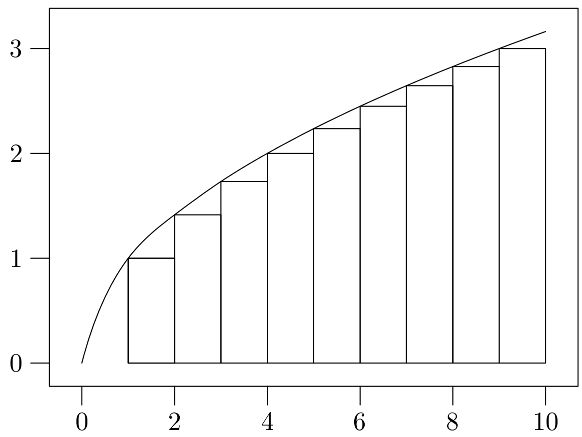
Die zweite Möglichkeit um $\sim$ zu verwenden sind *asymptotische Expansionen*. In diesem Fall ist die rechte Seite der Relation eine Reihe.

Wir schreiben

$$f(n) \sim \sum_{k=0}^{\infty} g_k(n) \iff f(n) \sim \sum_{k=0}^{m} g_l(n) \text{ fr alle } m \in \mathbf{N}. \tag{5.1}$$

Eine asymptotische Expansion bietet also eine ganze Folge von immer genauer werdenden Näherungen an $f(n)$ an.

Die einfachste Weise, zu asymptotischen Expansionen zu kommen, ist die Darstellung einer Funktion als Potenzreihe. In diesen Fällen können wir statt $\sim$ sogar das Symbol $=$ verwenden. Im allgemeinen können wir in einer asymptotischen Expansion $\sim$ aber *nicht* durch $=$ ersetzen. Die Reihe in (??) kann divergent sein und ist es oft auch.



5.1 Die Eulersche Summationsformel

Die Eulersche Summationsformel beruht darauf, eine Summe durch ein Integral zu ersetzen.

Theorem 7. ***(Eulersche Summationsformel)***
Wenn $\int_1^\infty |f^{(i)}(x)|\,dx$ für $1 \leq i \leq 2m$ existiert, dann gilt

$$\sum_{k=1}^{n} f(k) = \int_1^n f(x)\,dx + \frac{1}{2}f(n) + C + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)!} f^{(2k-1)}(n) + R_m,$$

wobei $R_m = O\left(\int_n^\infty |f^{(2m)}(x)|\,dx\right)$ und $B_k = n![z^n]z/(e^z - 1)$ die Bernoulli-Zahlen sind:

n	0	1	2	3	4	5	6
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$

Theorem 8.

$$\begin{array}{cccccccc} n & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ B_n & 1 & -\frac{1}{2} & \frac{1}{6} & 0 & -\frac{1}{30} & 0 & \frac{1}{42} \end{array} \tag{5.2}$$

Wenn stets $f^{(i)} = o(f^{(i+1)})$, dann liefert die Eulersche Summationsformel eine asymptotische Expansion. Wir rechnen zwei Beispiele.

Die harmonischen Zahlen sind durch die Summe

$$H_n = \sum_{k=1} \frac{1}{k}$$

definiert. Wir setzen also $f(x) = 1/x$ und müssen nun die ite Ableitung von $f(x)$ berechnen. Dies ist hier sehr leicht und wir erhalten

$$f^{(i)}(x) = i!(-1)^i x^{-1-i}.$$

Die Bedingungen der Eulerschen Summationsformel sind erfüllt und wir erhalten durch Einsetzen

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &\sim \int_1^n \frac{dx}{x} + \frac{1}{2n} + C + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)!} \frac{-(2k-1)!}{n^{-2k}} \\ &\sim \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \cdots \end{aligned}$$

Die unbekannte Konstante wird γ genannt und läßt sich nicht auf einfache Weise durch andere bekannte mathematische Konstanten darstellen.

$$\begin{aligned}
n! &= \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+\frac{1}{288n^2}+O(n^{-3})\right) \quad \text{(Stirlingsche Formel)} \\
H_n &= \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(n^{-4}) \quad \text{(Harmonische Zahlen),} \\
&\qquad \text{mit } \gamma \approx 0.57721566490153286060651209
\end{aligned}$$

Table 5.1: Einige wichtige asymptotische Näherungen

Als nächstes Beispiel wählen wir $n!$. Da $n!$ durch ein Produkt statt durch eine Summe definiert ist, wenden wir die Eulersche Summationsformel statt dessen auf $\ln(n!)$ an.

$$\begin{aligned}
\ln(n!) = \sum_{k=1}^{n} \ln(k) &\sim \int_1^n \ln(x)\,dx + \frac{1}{2}\ln n + \ln\sigma + \sum_{k=!}^{\infty} \frac{B_{2k}}{(2k)!}\frac{(2k-2)!}{n^{2k-1}} \\
&\sim (n+\frac{1}{2})\ln n - n + \ln\sigma + \frac{1}{12n} - \frac{1}{360n^3} + \cdots
\end{aligned}$$

Daraus erhalten wir die Näherung für $n!$. Es stellt sich heraus, daß $\sigma = \sqrt{2\pi}$.

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+\frac{1}{288n^2}-\frac{139}{51840n^3}+\cdots\right)$$

5.2 Singularitätenanalyse

Die Eulersche Summationsformel war eine Anwendung reeller Analysis. Die Methoden der reellen Analysis sind jedoch zu beschränkt. Wir wenden uns nun der komplexen Analysis zu. Im folgenden interessieren wir uns in erster Linie für Näherungen der Koeffizienten einer OGF oder EGF.

Unser erstes Theorem wird am einfachsten anzuwenden sein, aber auch die ungenauesten Näherungen liefern. Wir beschäftigen uns zunächst nur mit dem *exponentiellen Wachstum* dieser Koeffizienten.

Wir definieren hierzu für eine Folge a_n und eine positive reelle Zahl

$$a_n \asymp K^n \iff \limsup_{n\to\infty} |a_n|^{1/n} = K.$$

Wir können $a_n \asymp K^n$ auch so ausdrücken: Für jedes $\epsilon > 0$, sei es noch so klein, gilt $|a_n| > (K - \epsilon)^n$ für unendlich viele n, und $|a_n| < (K + \epsilon)^n$ gilt für alle n bis auf endlich viele Ausnahmen.

Wir können auch sagen, daß $a_n = \theta(n)K^n$ für alle n, wobei $\theta(n)$ eine subexponentielle Funktion ist, also langsamer als jede Exponentialfunktion wächst.

Eine Funktion ist *analytisch* im Punkt z_0 wenn sich durch eine Potenzreihe $\sum_{n=0}^{\infty} f_n z^n$ in einer Umgebung um z_0 darstellen läßt. Hierbei ist z_0 eine komplexe Zahl und die Umgebung liegt in der komplexen Zahlenebene.

Die Funktion $z \mapsto 1/z$ ist in der ganzen Ebene außer im Ursprung analytisch. Ein Punkt, in welchem eine Funktion aufhört analytisch zu sein, heißt *Singularität* dieser Funktion. Die Funktion $z \mapsto 1/z$ hat also eine Singularität und zwar 0.

Wir nennen eine Singularität *dominant*, wenn sie eine Singularität mit minimalem Betrag ist. Hier hilft meist ein Satz von *Pringsheim*, welcher besagt, daß Potenzreihen mit nicht-negativen Koeffizienten stets eine positive, reelle dominante Singularität besitzen. Wir können in diesem Fall also auf der reellen Achse bleiben, wenn wir nach dem Betrag der dominanten Singularitäten suchen.

Theorem 9. Die dominanten Singularitäten einer GF $f(z)$ bestimmen ihr exponentielles Wachstum: Es sei $f(z)$ eine GF und z_0 eine dominante Singularität mit $R = |z_0|$. Dann gilt

$$[z^n]f(z) \asymp \left(\frac{1}{R}\right)^n .$$

Betrachten wir als Beispiele die EGF $S(z)$ und die OGF $U(z)$, in geschlossener Form dargestellt als

$$S(z) = \frac{1}{2 - e^z} \text{ und } U(z) = \frac{1 - z - \sqrt{(1 - 3z)(1 + z)}}{2z}.$$

Die Funktion $S(z)$ hat Singularitäten bei allen z, für welche $e^z = 2$ ist, d.h., $z = \ln(2) + 2k\pi i$ für alle $k \in \mathbf{Z}$. Die dominante Singularität ist $\ln 2$ und wir erhalten

$$n![z^n]S(z) \asymp n! \left(\frac{1}{\ln 2}\right)^n .$$

Theorem

Es sei $f(z) := \sum_{n=0}^{\infty} f_n z^n$ mit $f_n \in \mathbb{C}$.

Dann gilt es ein $R \in \mathbb{R}_0^+ \cup \{\infty\}$, so daß

1.) $f(z)$ absolut konvergiert, falls $|z| < R$

2.) $f(z)$ divergiert, falls $|z| > R$.

Beweis

Wir zeigen zunächst, daß aus der Konvergenz von $f(z_0)$, die absolute Konvergenz von $f(z_1)$ für $|z_1| < |z_0|$ folgt:

Da $f(z_0)$ konvergiert, gilt $\lim_{n\to\infty} f_n z_0^n = 0$. Insbesondere ist $|f_n z_0^n| < C$ für ein $C \in \mathbb{R}_0^+$ bis auf endlich viele Ausnahmen.

Für ein genügend kleines $\varepsilon > 0$ gilt außerdem $|z_1| < (1-\varepsilon)|z_0|$.

Damit ist $|f_n z_1^n| < (1-\varepsilon)^n C$ und

$$\sum_{n=0}^{\infty} |f_n z_1^n| < C \sum_{n=0}^{\infty} (1-\varepsilon)^n = \frac{C}{\varepsilon}.$$

Als nächstes zeigen wir, daß wenn $f(z_0)$ divergiert, dann divergiert auch $f(z_1)$, falls $|z_1| > |z_0|$:

Wie wir eben sahen, folgte aus der Konvergenz von $f(z_1)$, die Konvergenz von $f(z_0)$, was ein Widerspruch wäre.

Sei nun R das Infimum aller $|z|$ für die $f(z)$ divergiert oder $R = \infty$ falls $f(z)$ in ganz $\mathbb{C}$ konvergiert.

Nach obigen Überlegungen ist nun klar, daß $f(z)$ divergiert, falls $|z| > R$. Andererseits konvergiert $f(z)$ für $|z| < R$ nach der Definition von R. □

Anmerkung: Es ist nicht klar, wie sich $f(z)$ für $|z| = R$ verhält, außer daß es mindestens ein z mit $|z| = R$ geben muß, so daß $f(z)$ divergiert.

Betrachten wir als Beispiel

$$\sum_{n=0}^{\infty} z^n.$$

Hier ist $R = 1$ und die Reihe divergiert für <u>alle</u> z mit $|z| = 1$. Die Reihe

$$\sum_{n=0}^{\infty} \frac{1}{n} \cdot z^n$$

verhält sich ganz anders. Hier ist $R = 1$ und sie divergiert ebenfalls für $z = 1$. Für $z = -1$ dagegen konvergiert sie bedingt, da sie eine Leibnizreihe bildet. Eine Reihe mit sehr schnell wachsenden Koeffizienten, wie zum Beispiel

$$\sum_{n=0}^{\infty} n!\, z^n$$

kann einen Konvergenzradius von $R = 0$ besitzen und wenn die Koeffizienten sehr schnell fallen, wie zum Beispiel in

$$\sum_{n=0}^{\infty} \frac{1}{n!} z^n$$

dann ist auch $R = \infty$ möglich.

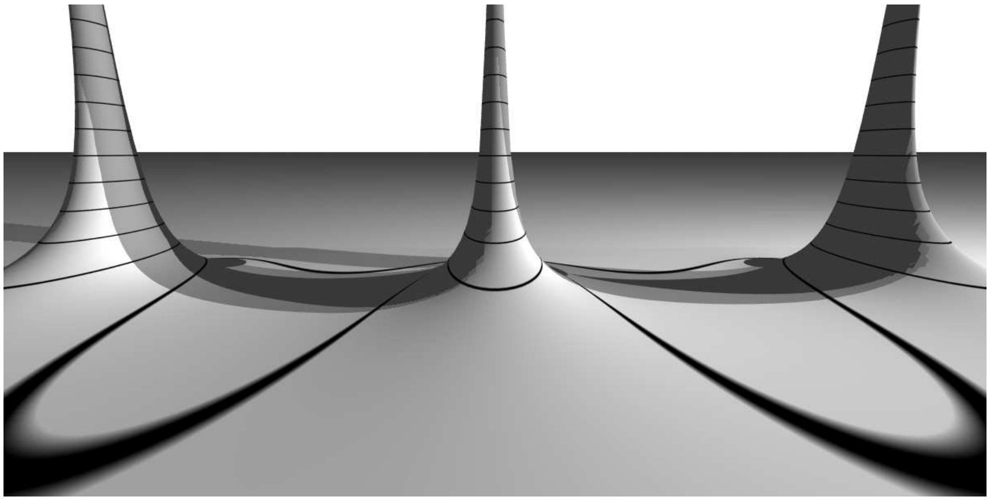


Figure 5.3: Der Betrag der Funktion $S(z) = 1/(2 - e^z)$. Die dominante Singularität bei $\ln 2$ ist in der Mitte zu sehen. Daneben liegen die beiden nächsten Singularitäten bei $\ln 2 \pm 2\pi i$.

Bei $U(z)$ finden wir die Singularitäten bei jenen z, für welche der Ausdruck unter der Wurzel 0 wird; dieses sind $\frac{1}{3}$ und -1. Bei 0 hat $U(z)$ keine Singularität, denn $\lim_{z\to 0} U(z) = \frac{1}{2}$. Die dominante Singularität ist $\frac{1}{3}$ und daher

$$[z^n]U(z) \asymp 3^n.$$

5.3 Meromorphe Funktionen

Das exponentielle Wachstum läßt sich leicht für beliebige GFs bestimmen. Die nächste Formel liefert genauere Näherungen, ist aber nur für eine Unterklasse aller Funktionen gültig — für die meromorphen Funktionen, welche wir jetzt definieren.

Eine Funktion $f(z)$ heißt in z_0 meromorph, wenn es Funktionen $g(z)$ und $h(z)$ gibt, so daß in einer Umgebung von z_0

$$f(z) = \frac{g(z)}{h(z)}$$

gilt und $g(z)$ und $h(z)$ in z_0 analytisch sind. Außerdem darf natürlich nicht $h = 0$ sein.

Es ist leicht einzusehen, daß $f(z)$ in z_0 genau dann meromorph ist, wenn es eine Entwicklung

$$\begin{aligned} f(z) = \sum_{n=-k}^{\infty} a_n(z-z_0)^n = \\ = \frac{a_{-k}}{(z-z_0)^k} + \frac{a_{-(k-1)}}{(z-z_0)^{k-1}} + \cdots + \frac{a_{-1}}{z-z_0} + \sum_{n=0}^{\infty} a_n(z-z_0)^n = \\ \frac{P(z)}{(z-z_0)^k} + \sum_{n=0}^{\infty} a_n(z-z_0)^n = \end{aligned}$$

gibt, die in einer Umgebung von z_0 gültig ist. Dabei ist k eine beliebige natürliche Zahl und wir können annehmen, daß $a_{-k} \neq 0$ ist. Das Polynom $P(z)$ ist dabei

$$P(z) = \sum_{n=0}^{k-1} a_{n-k}(z-z_0)^n.$$

Wir nennen dies die *Laurententwicklung* der Funktion $f(z)$ im Punkt z_0.

Wir sagen, daß die Funktion $f(z)$ in z_0 einen Pol $r$ter Ordnung hat, wenn $f(z)(z_0-z)^r$ in z_0 analytisch ist, aber $f(z)$ selbst nicht. Dann hat $f(z)$ hier einen Pol $k$ter Ordnung. Erwartungsgemäß sagen wir, daß eine Funktion in einem Gebiet G meromorph ist, wenn sie in jedem Punkt von G meromorph ist. Es ist jetzt aber klar, daß

$$f(z) - \sum_{n=-k}^{-1} a_n(z-z_0)^n = f(z) - \frac{P(z)}{(z-z_0)^k}$$

in z_0 keinen Pol mehr hat. Wenn f in einem Gebiet G meromorph ist, können wir also im Prinzip durch Addition einfacher Funktionen die Pole "'verschwinden"' lassen. Es bleibt dann eine analytische Funktion übrig. Das Verhalten der Koeffizienten $[z^n]f(z)$ hängt aber dann im wesentlichen nur von den wegaddierten Polen ab, denn die übrigbleibende analytische Funktion hat dann asymptotisch langsamer wachsende Koeffizienten. Wir können diese Erkenntnis im folgenden Theorem zusammenfassen.

Theorem 10. Es sei $f(z)$ meromorph für $|z| \leq R$. Innerhalb dieses Kreises habe $f(z)$ die Pole $\alpha_1, \alpha_2, \dots, \alpha_m$. Außerdem sei $f(z)$ im Ursprung und für $|z| = R$ analytisch. Dann gibt es Polynome $P_1(z), \dots, P_m(z)$, so daß

$$[z^n]f(z) = \sum_{j=1}^{m} P_j(n)\alpha_j^{-n} + O(R^{-n}).$$

Der Grad von $P_j(z)$ ist um eins kleiner als die Ordnung des Pols α_j.

Die Polynome lassen sich leicht bestimmen. Wir nehmen wieder $S(z) = 1/(2-e^z)$ als Beispiel. Wählen wir $R = 6$, dann gibt es nur die eine Singularität $\ln 2$ innerhalb des Kreises $|z| = R$. Der Pol $\ln 2$ hat die Ordnung 1. Wir sehen uns nun an, wie sich $S(z)$ für $z \to \ln 2$ verhält.

Es gilt $e^z \sim 2 - 2\ln(2) + 2z$ für $z \to \ln 2$ und daher

$$\frac{1}{2-e^z} \sim \frac{1}{2}\frac{1}{\ln 2 - z} = \frac{1}{2\ln 2}\frac{1}{1-(1/\ln 2)z} = \frac{1}{2\ln 2}\sum_{n=0}^{\infty}\left(\frac{1}{\ln 2}\right)^n z^n.$$

Das gesuchte Polynom $P_1(z)$ ist hier einfach $1/(2\ln 2)$. Im allgemeinen lassen sich alle $P_i(z)$ so bestimmen.

Die Lösung unseres Beispielproblems ist hiermit

$$[z^n]S(z) = \frac{1}{2}\left(\frac{1}{\ln 2}\right)^{n+1} + O(6^{-n}).$$

Betrachten wir ein weiteres Beispiel, welches belegt, daß man mit asymptotischen Methoden sehr leicht eine sehr gute Näherung berechnen kann, obwohl eine exakte Lösung unerreichbar scheint.

Ein bekanntes Beispiel aus der Wahrscheinlichkeitsrechnung sind die Matrosen, die ihre Kojen zufällig wählen. In dieser Geschichte gibt es n Matrosen, die sich auf einem Landurlaub betrinken und dann, auf das Schiff zurückgekehrt, betrunken eine beliebige Koje wählen. Es läßt sich nun leicht berechnen, daß die Wahrscheinlichkeit, daß ein bestimmter Matrose in seiner eigenen Koje schläft, gerade $1/n$ ist und damit im Mittel ein Matrose in das eigene Bett findet. Wie groß ist aber die Wahrscheinlichkeit, daß *kein* Matrose in seinem eigenen Bett schläft?

Eine Permutation, die keinen Fixpunkt enthält, wird *derangement* genannt. Die Matrosenfrage ist also verwandt mit dem Problem, die Anzahl D_n der *derangements* einer n-elementigen Menge zu bestimmen. Da ein *derangements* zwischen einem und n Fixpunkte entält und es genau $\binom{n}{k}D_{n-k}$ Permutationen mit *genau* k Fixpunkten gibt, ist es sehr leicht eine Rekursionsgleichung für D_n anzugeben:

$$D_n = n! - \sum_{k=1}^{n}\binom{n}{k}D_{n-k}$$

Leicht umgeschrieben bekommt sie diese einfachere Form:

$$n! = \sum_{k=1}^{n} \binom{n}{k} D_k$$

Da es sich um markierte Objekte handelt, versuchen wir die Gleichung mit einer EGF zu lösen. Die rechte Seite ist eine Binomialsumme und so erhalten wir direkt

$$\frac{1}{1-z} = e^z D(z)$$

und damit die EGF für D_n in geschlossenere Form:

$$D(z) = \frac{e^{-z}}{1-z}$$

Wir können jetzt sofort interessante Eigenschaften von D_n direkt ablesen. Zunächst fällt auf das es nur eine Singularität bei 1 gibt. Das heißt also $n![z^n]D(z) \asymp n!$ und damit $D_n \asymp n!$ oder, noch einmal anders ausgedrückt, daß das Verhältnis von D_n und $n!$ subexponentionell ist. Das ist schon einmal überraschend, denn es bedeutet daß es relativ viele Permutationen *derangements* sind. Die Wahrscheinlichkeit, daß alle Sehmänner in fremden Kojen liegen, muß also relativ hoch sein.

Wie hoch ist sie aber genau? Gehen wir in der Analyse weiter e^{-z} ist in der ganzen Gaußschen Zahlenebene analytisch und hat nur bei ∞ eine Singularität. Daher ist $D(z)$ meromorph und hat nur bei $z = 1$ einen Pol erster Ordnung. Also können wir D_n darstellen als

$$n![z^n]D(z) = P_1(n)\alpha_1^n + O(R^{-n}) = C \cdot 1^n + O(R^{-n})$$

wobei wir R beliebig groß wählen können. Die Konstante C ist das Polynom erster Ordnung und wir können C bestimmen, indem wir $D(z)$ für $z \to 1$ betrachten:

$$\frac{e^{-z}}{1-z} \sim \frac{1}{e} \sum_{n=0}^{\infty} z^n \text{ für } z \to 1$$

Unser Ergebnis ist jetzt schließlich

$$D_n = \frac{n!}{e} + O(n!\,\epsilon^n)$$

für jedes $\epsilon > 0$. Da der Fehlerterm sehr schnell fällt, können wir erwarten, daß die Näherung auch für kleine n sehr gut ist. Dafür gibt es aber keine

n	D_n	$n!$	$n!/D_n$
2	1	2	2.000000000000000
3	2	6	3.000000000000000
4	9	24	2.666666666666667
5	44	120	2.727272727272727
6	265	720	2.716981132075472
7	1 854	5 040	2.718446601941748
8	14 833	40 320	2.718263331760264
9	133 496	362 880	2.718283693893450
10	1 334 961	3 628 800	2.718281657666404
11	14 684 570	39 916 800	2.718281842777827
12	176 214 841	479 001 600	2.718281827351874
13	2 290 792 932	6 227 020 800	2.718281828538486
14	32 071 101 049	87 178 291 200	2.718281828453728
15	481 066 515 734	1 307 674 368 000	2.718281828459379
16	7 697 064 251 745	20 922 789 888 000	2.718281828459026
17	130 850 092 279 664	355 687 428 096 000	2.718281828459046
18	2 355 301 661 033 953	6 402 373 705 728 000	2.718281828459045
19	44 750 731 559 645 106	121 645 100 408 832 000	2.718281828459045

Table 5.2: Eine sehr gute Näherung für D_n ist $n!/e$. Der Wert von e auf 15 Dezimalstellen ist 2.718281828459045.

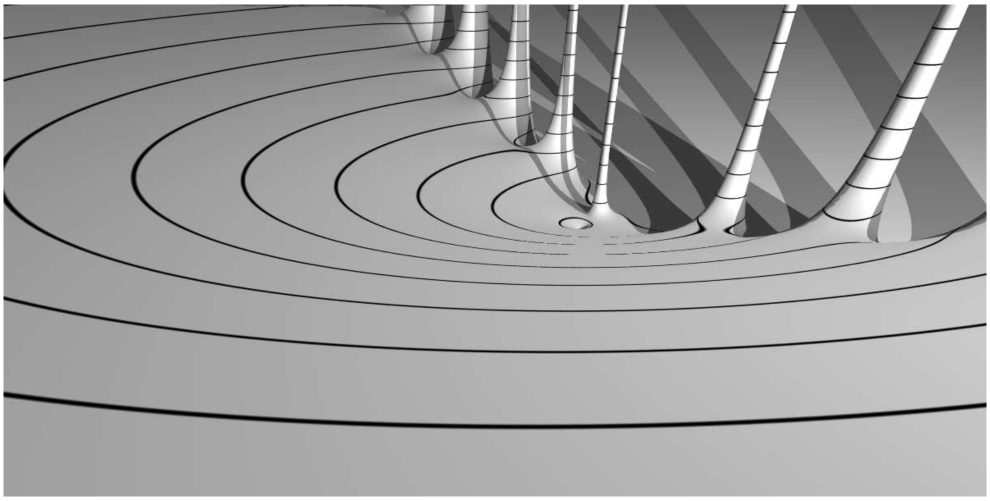


Figure 5.4: Der Betrag der Funktion $z(1-z)/(e^{z-1}-z)$. In der Mitte ist die einzige Nullstelle bei $z=0$ zu sehen. Daneben ist die dominante Singularität bei $z=1$, umgeben von einer Schar konjugiert komplexer weiterer Singularitäten.

Garantie, denn die versteckte Konstante im grossen O könnte ja gigantisch sein (und sie hängt natürlich von ϵ ab).

Tabelle 5.2 zeigt, daß die Näherung in der Tat exzellent ist.

Betrachten wir als weiteres Beispiel die erzeugende Funktion

$$L(z) = \sum_{n=0}^{\infty} L_n z^n = \frac{z(1-z)}{e^{z-1}-z},$$

wobei L_n die Länge des n-ten Laufs einer zufälligen Folge von Zahlen ist. Es ist leicht zu sehen, daß $L_1 = 1 + \frac{1}{2} + \frac{1}{3!} + \frac{1}{4!} + \cdots = e - 1$ ist.

Was sind nun die Singularitäten von $L(z)$? Wir finden die dominante Singularität $z_0 = 1$ auf der positiven reellen Achse und die nächsten beiden konjugiert komplexen Singularitäten z_1 und $\bar{z}_1$ im rechten oberen bzw. unteren Quadranten. Da wir z mit $e^{z-1} = z$ suchen müssen simultan die beiden Gleichungen $e^{x-1}\cos y = x$ und $e^{x-1}\sin y = y$ für den Real- und Imaginärteil erfüllt sein. Abbildung 5.4 zeigt die Schar der Singularitäten. Wir bestimmen nun die Ordnung des Pols $z_0 = 1$: Es stellt sich heraus, daß

$$\lim_{z\to 1} \frac{1-z}{e^{z-1}-z} = \lim_{z\to 1} \frac{-1}{e^{z-1}-1}$$

Figure 5.5: Die Gamma-Funktion

nicht existiert, aber

$$\lim_{z\to 1}\frac{1-z}{e^{z-1}-z}(z-1)=\lim_{z\to 1}\frac{z^2-2z+1}{z-e^{z-1}}=\lim_{z\to 1}\frac{2z-2}{1-e^{z-1}}=\lim_{z\to 1}\frac{2}{-e^{z-1}}=-2$$

sehr wohl existiert. Daher ist die $z_0 = 1$ ein Pol erster Ordnung.

Außerdem hat $L(z) + 2/(z-1)$ jetzt keine Singularität bei 1 mehr. Die nächsten Singularitäten sind weit weg: Es ist $|z_1| = |\bar{z}_1| = 8.07556\ldots > 8$ und daher ist

$$[z^n]L(z) = L_n = [z^n]\frac{-2}{z-1} + O(8^{-n}) = 2 + O(8^{-n}).$$

Die Folge der L_n konvergiert also recht schnell gegen 2.

5.4 Algebraische Singularitäten

Eine Funktion $f(z)$ hat eine *algebraische Singularität* im Punkt z_0, wenn wir $f(z)$ schreiben können als

$$f(z) = h(z) + \sum_{j=1}^{m}\left(1-\frac{z}{\alpha}\right)^{c_j} g_j(z),$$

wobei $h(z)$ und $g_j(z)$ analytisch in z_0 sind und $c_j \neq \{0, 1, 2, \ldots\}$.

Pole sind spezielle algebraische Singularitäten.

Leider können wir bei algebraischen Singularitäten nicht so gute Abschätzungen finden wie bei meromorphen Funktionen. Wir begnügen uns mit dem folgenden Satz von Darboux, welcher es erlaubt, mit wenig Aufwand eine asymptotische Abschätzung zu finden.

Theorem 11. Es sei $f(z)$ analytisch für $|z| < R$ und habe auf $|z| = R$ nur algebraische Singularitäten. Es gelte

$$f(z) = h(z) + \sum_{j=1}^{k} (1 - z/\alpha_j)^{c_j} g_j(z),$$

wobei $h(z)$ und alle $g_j(z)$ analytisch in $|z| \leq R$ sind und α_j die Singularitäten auf $|z| = R$. Es sei $c_j \notin \{0, 1, 2, \ldots\}$. Wir definieren $a = \min\{\Re(\alpha_j) \mid 1 \leq j \leq k\}$. Dann gilt

$$[z^n]f(z) = -\sum_j \frac{g_j(\alpha_j) n^{-c_j-1}}{\Gamma(-c_j)\alpha_j^n} + o(R^{-n} n^{-a-1}),$$

wobei die Summe nur über alle j mit $\Re(c_j) = a$ geht.

Die Idee hinter der Methode von Darboux ist dieselbe wie hinter der Behandlung von Polen: Falls $G(z)$ bei α eine algebraische Singularität hat, dann finden wir eine Funktion $H(z)$, die einerseits leicht in eine Potenzreihe entwickelt werden kann und andererseits ebenfalls bei α eine algebraische Singularität besitzt. Wenn wir das geschickt anstellen, dann wird $[z^n]G(z) \sim [z^n]H(z)$ gelten, weil $[z^n](G(z) - H(z))$ asymptotisch langsamer wächst. Leider können wir die algebraische Singularität auf diese Weise im allgemeinen nicht ganz verschwinden lassen, wie es bei Polen möglich war.

Nehmen wir uns zur Illustration jetzt die OGF

$$G(z) = \sqrt{(1-z)(1-\alpha z)}$$

vor, wobei $\alpha < 1$ gelten soll. Jetzt hat $G(z)$ die Singularitäten 1 und $1/\alpha$, wobei 1 die dominante ist. Die Singularität 1 ist algebraisch, da ja $G(z) = (1-z)^{1/2}R(z)$ ist, wobei dann $R(z)$ in 1 analytisch ist. Wir wollen jetzt eine Vergleichsfunktion finden, so daß $[z^n](G(z) - H(z))$ klein ist, so daß $H(z)$ eine gute Näherung ist. Hierfür sollte $H(z)$ eine algebraische Singularität bei 1 haben und sich in der Nähe von 1 wie $G(z)$ verhalten. Durch eine Taylorentwicklung an der Stelle 1 von $G(z)$ können wir eine schöne Funktion finden, die in der Nähe von 1 sich wie $G(z)$ verhält:

$$G(z) \sim (z-1)^{1/2}\sqrt{\alpha-1} + (z-1)^{3/2}\frac{\sqrt{\alpha(\alpha-1)}}{2(\alpha-1)} - (z-1)^{5/2}\frac{\sqrt{\alpha^2(\alpha-1)}}{8(\alpha-1)^2}$$

Wählen wir also zunächst einmal $H(z) = (z-1)^{1/2}\sqrt{\alpha-1}$ und betrachten wir $G(z) - H(z) = (1-z)^{1/2}(\sqrt{1-\alpha z} - \sqrt{1-\alpha})$. Der zweite Faktor $\sqrt{1-\alpha z} - \sqrt{1-\alpha}$ hat jetzt bei 1 keine Singularität mehr. Ganz im Gegenteil: Es gilt $\sqrt{1-\alpha z} - \sqrt{1-\alpha} \sim 0$ für $z \to 1$. Das haben wir auch erwartet und sogar

$$\frac{\sqrt{1-\alpha z} - \sqrt{1-\alpha}}{1-z}$$

hat keine Singularität bei 1. Daher können wir $G(z) - H(z)$ als

$$G(z) - H(z) = (1 - z)^{3/2} \frac{\sqrt{1 - \alpha z} - \sqrt{1 - \alpha}}{1 - z}$$

schreiben. Ebenso wie $G(z)$ hat $G(z) - H(z)$ eine algebraische Singularität bei 1, aber die Ordnung ist $3/2$ statt $1/2$. Wir werden nun sehen, daß die Koeffizienten von $G(z) - H(z)$ asymptotisch wesentlich kleiner sind als die Koeffizienten von $G(z)$. Dadurch wird $[z^n]H(z)$ eine gute Näherung für $[z^n]G(z)$.

Um $[z^n](G(z) - H(z)$ abzuschätzen, schreiben wir $G(z) - H(z) = A(z)B(z)$ mit $A(z) = (1-z)^{3/2}$ und $B(z) = (\sqrt{1 - \alpha z} - \sqrt{1 - \alpha})/(1-z)$. Die dominante Singularität für $B(z)$ ist $1/\alpha$ und daher gilt $[z^n]B(z) \asymp \alpha^{-n}$ oder $[z^n] = O(r^{-n})$ für ein $r > 1$ (und $r < 1/\alpha$).

Den Koeffizienten für $A(z)B(z)$ erhalten wir als Faltung:

$$[z^n]A(z)B(z) = \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k}$$

Am Anfang der Summe dominieren die a_k und am Ende die b_{n-k}, was eine Zerlegung der Summe in wenigstens zwei Teile empfiehlt. Für b_n haben wir bereits die sehr starke Abschätzung $b_n = O(r^{-n})$ und wir erhalten eine Abschätzung für a_n, indem wir $A(z)$ nach der Newton-Formel entwickeln:

$$a_n = \binom{3/2}{n}(-1)^n = \binom{n - 5/2}{n} = O(n^{-5/2})$$

Jetzt können wir die Teilsummen großzügig abschätzen:

$$\sum_{k=0}^{\lceil n/2 \rceil} a_k b_{n-k} = O(r^{-n/2})$$
$$\sum_{\lfloor n/2 \rfloor}^{n} a_k b_{n-k} = O(n^{-5/2})$$

Insgesamt haben wir $[z^n](G(z) - H(z)) = O(r^{-n/2}) + O(n^{-5/2}) = O(n^{-5/2})$ und damit $[z^n]G(z) = [z^n]H(z) + O(n^{-5/2})$. Da $H(z)$ sich direkt expandieren läßt, erhalten wir folgendes Ergebnis:

$$[z^n]G(z) = \sqrt{\alpha - 1}\binom{1/2}{n}(-1)^n + O(n^{-5/2}) = \sqrt{\alpha - 1}\frac{n^{-3/2}}{\Gamma(1/2)} + O(n^{-5/2})$$

Betrachten wir $n = 50$ und $\alpha = 1/2$. Der genaue Wert von $[z^n]G(z)$ ist dann

$$-\frac{99827864011764212779295458819104396304431}{178405961588244985132285746181186892047843328} = -5.595545\ldots\times 10^{-4}$$

und

$$\sqrt{1-1/2}(-1)^{50}\binom{1/2}{50} = -5.684655\ldots \times 10^{-4}.$$

5.5 Die Sattelpunktmethode

Im letzten Abschnitt befassen wir uns mit erzeugenden Funktionen, die keine Singularitäten besitzen. Für solche Funktionen haben wir noch kein allgemeines Verfahren, welches die Koeffizienten bestimmt.

Wir behelfen uns hier mit dem Residuumtheorem, einem Satz der Funktionentheorie, aus welchem folgt:

$$g_n = \frac{1}{2\pi i}\int_\Gamma \frac{G(z)}{z^{n+1}}dz,$$

wobei Γ eine geschlossene Kurve ist, die einmal den Ursprung im mathematisch positivem Sinn umläuft.

Warum gilt dieser Satz? Betrachten wir einmal einen Kreis um den Urspung:

$$\int_C \frac{1}{z^{n+1}} = \int_0^{2\pi} e^{-(n+1)\phi i} de^{\phi i} = \int_0^{2\pi} e^{-(n+1)\phi i} i e^{\phi i} d\phi =$$
$$i\int_0^{2\pi} e^{n\phi i} d\phi = \begin{cases} 2\pi i & \text{falls } n = 0 \\ \frac{1}{n}e^{n\phi i}\,\Big|_{\phi=0}^{2\pi} = 0 & \text{sonst} \end{cases}$$

Es stellt sich also heraus, daß nur für $n = 0$ das Integral nicht verschwindet.

Wegen der Linearität des Integrals erhalten wir daraus direkt das Residuumtheorem.

Gewöhnlich wird dieser Satz verwendet, um Integrale berechnen zu können, aber wir verwenden ihn hier genau umgekehrt: Aus einer Näherung des Integrals erhalten wir den Koeffizienten einer Erzeugendenfunktion.

Um so ein Integral näherungsweise berechnen zu können, verwenden wir im allgemeinen einen Trick, der der Methode den Namen gibt. Wir haben ja die Freiheit, die Kurve um den Ursprung frei zu w"ahlen und es ist wohl nicht überraschend, daß manche Kurven zu einfacher zu berechnenden Integralen führen als andere. Am besten scheint es, wenn der Wert des Integranden fast auf der ganzen Kurve sehr klein ist und daher der Wert des Integrals hauptsächlich von einem sehr kurzen Kurvenstück stammt.

Ist das der Fall, können wir die Kurve in zwei Teile spalten. Wir müssen abschätzen, daß das erste Integral sehr klein ist. Dann müssen wir den Wert des zweiten Integrals näherungsweise bestimmen. Da aber das Kurvenstück sehr kurz ist, kann das relativ einfach sein. Beispielsweise ist eine Taylorapproximation ja recht genau, wenn man sich nicht zu weit vom Entwicklungspunkt entfernt.

Es gibt eine gute Heuristik, so eine Kurve zu finden: Da wir eine ganze Funktion (ohne Singularitäten) durch z^{n+1} teilen, hat die resultierende Funktion eine Singularität im Ursprung und eine im unendlich fernen Punkt. Der Betrag einer in einem Gebiet G analytischen Funktion hat in G keine Minima oder Maxima. Der Betrag geht aber in der Nähe des Ursprungs gegen unendlich. Der Betrag muß also einen Sattelpunkt zwischen den beiden Singularitäten besitzen (zwischen 0 und ∞). Die Umgebung eines Sattelpunkts ist quer zum Sattel in der Regel stark abfallend. Also sollten wir den Weg des Integrals durch (oder nahe an) den Sattelpunkt führen, in der Hoffnung daß das Integral nur dort einen großen Beitrag erhält. Der Rest des Weges muß dann noch, falls nötig, geschickt zu einer geschlossenen Kurve um den Ursprung geschlossen werden. Das soll in einer Weise geschehen, daß dieser Teil der Kurve möglichst "'im Flachen"' verläuft.

Wir werden als Beispiel die erzeugende Funktion $G(z) = e^z$ verwenden, da sie relativ einfach ist und wir das Ergebnis $g_n = 1/n!$ schon kennen. Dadurch können wir gut sehen, wie groß der Fehler tatsächlich ist.

Wenn wir die Funktion e^z/z^{n+1} betrachten, dann finden wir einen Sattenpunkt bei $n + 1$. Es genügt ja, die Funktion abzuleiten und Null zu setzen.

Abbildung 5.6 zeigt den Betrag dieser Funktion in der Gaußschen Ebene. Der Sattel ist dabei deutlich zu erkennen.

Rechnerisch ist es hier einfacher durch den Punkt n zu gehen, was aber auch sehr nahe am Sattelpunkt liegt. Wir verwenden einfach einen Kreis,

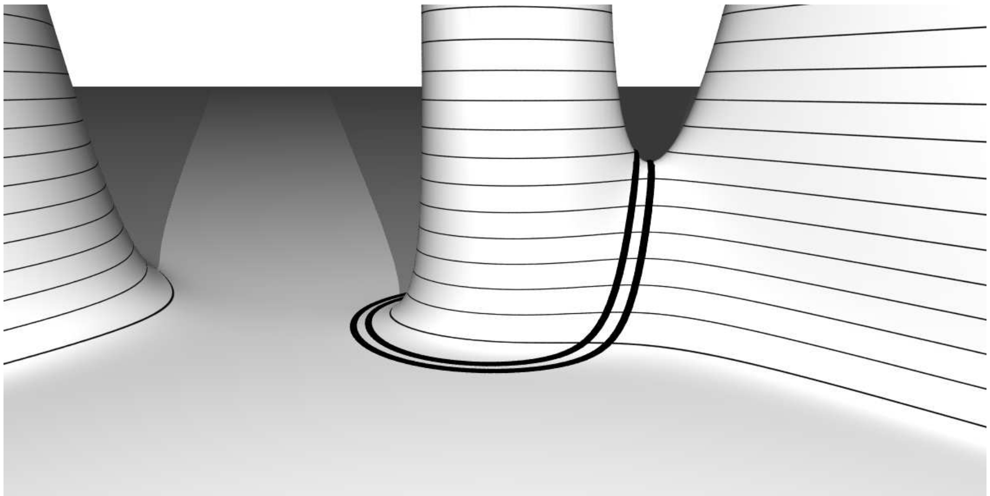


Figure 5.6: Der Betrag der Funktion e^z/z^{n+1} für $n = 5$. Es sind zwei mögliche Integrationswege eingezeichnet. Der äußere Kreis geht genau durch den Sattelpunkt (mit $R(R+1) = n+1$) und der innere Kreis erfüllt die für die Rechnung günstige Identität $R(R+1) = n$. Man sieht deutlich die Singularität im Ursprung. Entlang der reellen Achse steigt die Funktion stark an, wenn man sich vom Ursprung entfernt.

der mit Radius n um den Ursprung geht.

$$\frac{1}{2\pi i}\int_C \frac{e^z}{z^{n+1}}dz = \frac{1}{2\pi i}\int_0^{2\pi} \frac{e^{ne^{i\phi}}}{n^{n+1}e^{(n+1)i\phi}}dne^{i\phi} =$$
$$= \frac{1}{2\pi}\int_0^{2\pi} \frac{e^{ne^{i\phi}}}{n^{n+1}e^{(n+1)i\phi}}ne^{i\phi}d\phi = \frac{1}{2\pi}\int_0^{2\pi} \frac{e^{ne^{i\phi}}}{n^n e^{ni\phi}}d\phi = \frac{1}{2\pi}\frac{A+B}{n^n}$$

mit

$$A = \int_{-\delta}^{\delta} \frac{e^{ne^{i\phi}}}{e^{ni\phi}}d\phi \qquad \text{und} \qquad B = \int_{\delta}^{2\pi-\delta} \frac{e^{ne^{i\phi}}}{e^{ni\phi}}d\phi$$

Wenn wir δ entsprechend wählen, sollte A den Großteil des Integrals enthalten und B asymptotisch zu vernachlässigen, d.h. $B = o(A)$ und damit $A + B \sim A$. Wir müssen A und B genau analysieren und beginnen mit A:

$$A = \int_{-\delta}^{\delta} e^{ne^{i\phi}-ni\phi}d\phi$$

Für den Exponenten gilt:

$$ne^{i\phi} - ni\phi \sim n(1 - \frac{\phi^2}{2} + O(\phi^3))$$

und damit

$$A = \int_{-\delta}^{\delta} e^{n(1-\delta^2/2+O(\phi^3))} d\phi = e^n \int_{-\delta}^{\delta} e^{-n\phi^2/2}(1 + O(n\phi^3)) d\phi =$$
$$= e^n(1 + O(n\delta^3)) \int_{-\delta}^{\delta} e^{-n\phi^2/2} d\phi$$

Für eine gute Abschätzung muß also $n\phi^3 = o(1)$ gelten, also $\delta = o(n^{-1/3})$. Um das Integral zu lösen, substituieren wir $\phi = t\sqrt{2/n}$ und erhalten:

$$\int_{-\delta}^{\delta} e^{-n\phi^2/2} d\phi = \int_{-\delta\sqrt{n/2}}^{\delta\sqrt{n/2}} e^{-t^2} dt \sqrt{2/n} = \sqrt{\frac{2}{n}} \int_{-\delta\sqrt{n/2}}^{\delta\sqrt{n/2}} e^{-t^2} dt$$

Für dieses Integral gibt es keine geschlossene Lösung, aber das uneigentliche Integral $\int_{-\infty}^{\infty} e^{-t^2} dt = \sqrt{\pi}$ ist wohlbekannt.

$$\sqrt{\frac{2}{n}} \int_{-\delta\sqrt{n/2}}^{\delta\sqrt{n/2}} e^{-t^2} dt = \sqrt{\frac{2}{n}} \int_{-\infty}^{\infty} e^{-t^2} dt + 2\sqrt{\frac{2}{n}} \int_{\delta\sqrt{n/2}}^{\infty} e^{-t^2} dt =$$
$$= \sqrt{\frac{2\pi}{n}} + O\left(n^{-1/2} \int_{\delta\sqrt{n/2}}^{\infty} e^{-t} dt\right) = \sqrt{\frac{2\pi}{n}} + O(n^{-1/2} e^{-\delta\sqrt{n/2}}) \sim \sqrt{\frac{2\pi}{n}}$$

Diese Abschätzung funktioniert, falls $\delta\sqrt{n/2} = \omega(1)$, da wir nur dann e^{-t^2} durch e^{-t} ersetzen können. Wir müssen also $\delta = \omega(n^{-1/2})$ wählen und erhalten $A \sim \sqrt{2\pi/n}e^n$.

Schließlich können wir uns B zuwenden:

$$B = \int_{\delta}^{2\pi-\delta} e^{ne^{i\phi}-ni\phi} d\phi \le \int_{\delta}^{2\pi-\delta} \left| e^{ne^{i\phi}-ni\phi} \right| d\phi =$$
$$= \int_{\delta}^{2\pi-\delta} e^{n\cos\phi} d\phi \le 2\pi e^{n\cos\delta} \le 2\pi e^n \cdot e^{-n\delta^2/2} = e^{o(n)}$$

falls $n\delta^2 = \omega(1)$, also $\delta = \omega(n^{-1/2})$. Wir können $\delta = n^{-5/12}$ wählen, um alle Bedingungen an δ zu erfüllen. Insgesamt ergibt sich: $A + B \sim \sqrt{\frac{2\pi}{n}}e^n$ und damit $\frac{1}{2\pi i}\int_C \frac{e^z}{z^{n+1}} dz = \frac{1}{2\pi}\frac{A+B}{n^n} \sim \frac{1}{2\pi n}\frac{e^n}{n^n}$.

Damit ergibt sich $n! \sim \sqrt{2\pi n}\frac{n^n}{e^n}$, eine Abschätzung für die Fakultätsfunktion. Die Gaußsche Summationsformel ist zwar genauer, konnte aber die Stirlingkonstante $\sigma = \sqrt{2\pi}$ nicht liefern.

$I(z) = e^{z+z^2/2}$

$$\left(\frac{I(z)}{z^{n+1}}\right)' = \frac{(z+1)e^{z+z^2/2}}{z^{n+1}} - (n+1)\frac{e^{z+z^2/2}}{z^{n+2}}$$

Null, wenn $z(z+1) = n+1$. Der Sattelpunkt liegt eher ungefähr bei $\sqrt{n}$. Wir wählen als Intergrationsweg einen Kreis, lassen aber den Radius R noch offen.

$$[z^n]I(z) = \frac{1}{2\pi i}\int_c \frac{e^{z+z^2/2}}{z^{n+1}}dz = \frac{1}{2\pi i}\int_0^{2\pi} \frac{e^{Re^{i\phi}+R^2e^{2i\phi}/2}}{R^{n+1}e^{i(n+1)\phi}}dRe^{i\phi} =$$
$$= \frac{1}{2\pi}\int_0^{2\pi} e^{Re^{i\phi}+R^2e^{2i\phi}/2-in\phi}d\phi = \frac{1}{2\pi R^n}(A+B)$$

wobei

$$A = \int_{-\delta}^{\delta} e^{Re^{i\phi}+R^2e^{2i\phi}/2-in\phi}d\phi$$

und

$$B = \int_{\delta}^{2\pi-\delta} e^{Re^{i\phi}+R^2e^{2i\phi}/2-in\phi}d\phi$$

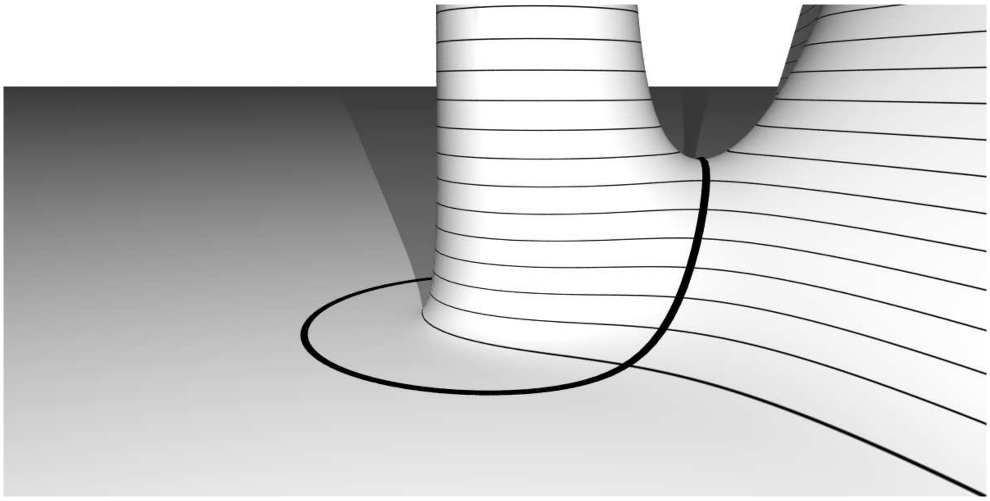
Sehen wir uns zunächst A an:

$$e^{Re^{i\phi}+R^2e^{2i\phi}/2-in\phi} = e^{R(1+i\phi-\phi^2/2+O(\phi^3))+R^2(\frac{1}{2}+i\phi-\phi^2+O(\phi^3))-in\phi} =$$
$$= e^{R+R^2/2}\cdot e^{-\phi^2(R/2+R^2)+i\phi(R+R^2-n)+(R+R^2)O(\phi^3)}$$

Wir wählen nun $R^2+R=n$, so daß der Faktor vor $i\phi$ verschwindet. Dann ist $R = \frac{1}{2}\sqrt{4n+1}-\frac{1}{2}$ und wir erhalten

$$A = e^{R+R^2/2}\int_{-\delta}^{\delta} e^{i\phi^2(n+O(\sqrt{n}))+O(\phi^3)}d\phi =$$
$$= e^{R+R^2/2}\int_{-\delta}^{\delta} e^{i\phi^2 n}(1+O(\phi^2\sqrt{n}))(1+O(n\phi^3))d\phi =$$
$$= e^{R+R^2/2}(1+O(\delta^2\sqrt{n})+O(\delta^3 n))\int_{-\delta}^{\delta} e^{-\phi^2 n}d\phi$$

Wir brauchen die Eigenschaften $\delta^2\sqrt{n} = o(1)$ und $\delta^3 n = o(1)$, damit dies eine gute Näherung für A ist. Das bedeutet $\delta = o(n^{-1/3})$ und $\delta = o(n^{-1/4})$, wobei die erste Vorderung natürlich die stärkere ist. Wenden wir uns



schließlich den verbleibenden Intergral zu. Wir substituieren zunächst $t = \phi/\sqrt{n}$

$$\int_{-\delta}^{\delta} e^{-\phi^2 n} d\phi = \int_{-\delta\sqrt{n}}^{\delta\sqrt{n}} e^{-t^2} d\frac{t}{\sqrt{n}} = \frac{1}{\sqrt{n}} \left(\int_{\infty}^{-\infty} e^{-t^2} dt - 2 \int_{\delta\sqrt{n}}^{\infty} e^{-t^2} \right) =$$

$$= \frac{1}{\sqrt{n}} \left(\sqrt{\pi} + o(1) \right)$$

Wir erhalten $\int_{\delta\sqrt{n}}^{\infty} dt = o(1)$, falls nur $\delta\sqrt{n} = w(1)$, also wenn $\delta = w(n^{-1/2})$ ist. Da wir gleichzeitig $\delta = o(n^{-1/3})$ und $\delta = \omega(n^{-1/2})$ brauchen, wählen wir nun $\delta = n^{-5/12}$. Insgesamt ergibt sich

$$A = e^{R+R^2/2} \sqrt{\frac{\pi}{n}} (1 + o(1))$$

und wir können uns B zuwenden:

$$B = \int_{\delta}^{2\pi-\delta} e^{Re^{i\phi} + R^2 e^{2i\phi}/2 - in\phi} d\phi$$

Wir möchten zeigen, daß B klein ist. Hierzu genügt es, den Betrag des Integranden zu betrachten:

$$e^{R\cos\phi + R^2\cos(2\phi)/2}$$

Damit $|e^{Re^{i\phi} + R^2 e^{2i\phi}/2 - in\phi}| = o(e^{R+R^2/2})$ gilt, genügt es, wenn eine der folgenden Bedingungen erfüllt wird:

1. $1 - \cos\phi = \omega(1/\sqrt{n})$

2. $1 - \cos 2\phi = \omega(1/n)$

Wir müssen nun nachweisen, daß für jedes ϕ mit $\delta \leq \phi \leq 2\pi - \delta$, wenigstens eine der zwei Bedingungen erfüllt sind. Am einfachsten ist es vielleicht, sich anzusehen, für welche δ die Bedingungen jeweils verletzt werden:

1. $\phi = O\left(1/\sqrt{n}\right)$ oder $\phi = 2\pi + O\left(1/\sqrt{n}\right)$

2. $\phi = O\left(1/n\right)$ oder $\phi = \pi + O\left(1/n\right)$ oder $\phi = 2\pi + O\left(1/n\right)$

Die beiden Bedingungen zusammen lassen nur die Möglichkeiten $\phi = O\left(1/n\right)$ und $\phi = 2\pi + O\left(1/n\right)$ übrig. Da aber $n^{\frac{-5}{12}} \leq \phi \leq 2\pi - n^{\frac{-5}{12}}$, fallen auch diese weg. Zusammenfassend haben wir nun

$$A + B \sim \sqrt{\frac{\pi}{n}} e^{R+R^2/2},$$

was sich noch vereinfachen läßt:

$$\begin{aligned} R + \frac{R^2}{2} &= n - \frac{R^2}{2} = n - \frac{1}{8}\left(\sqrt{4n+1} - 1\right)^2 = \\ &= n - \frac{1}{8}\left(4n + 1 - 2\sqrt{4n+1} + 1\right) = \frac{n}{2} - \frac{1}{4} + \frac{\sqrt{n}}{2} + O(1/n) \end{aligned}$$

und damit

$$A + B \sim \sqrt{\frac{\pi}{n}} e^{n/2 + \sqrt{n}/2 - 1/4}$$

Wir erhalten

$$[z^n]I(z) = \frac{1}{2\pi R^n}(A + B) \sim \frac{e^{n/2+\sqrt{n}/2-1/4}}{2\sqrt{\pi n} R^n}$$

Zuletzt vereinfachen wir noch R^n:

$$\begin{aligned} R^n &= e^{n \ln(\frac{1}{2}\sqrt{4n+1} - \frac{1}{2})} = e^{n(\ln\frac{1}{2} + \ln(\sqrt{4n}(1 - 1/\sqrt{4n} + O(1/n))))} = \\ &= 2^{-n}\sqrt{4n}^n e^{n(-1/\sqrt{4n} + O(1/n))} \sim n^{n/2} e^{-sqrt n/2} \end{aligned}$$

Als Endergebnis erhalten wir

$$[z^n]I(z) \sim \frac{e^{n/2+\sqrt{n}/2-1/4}}{2\sqrt{\pi n} R^n} \sim \frac{e^{n/2+\sqrt{n}/2-1/4}}{2\sqrt{\pi n} n^{n/2} e^{-\sqrt{n}/2}} = \frac{e^{n/2+\sqrt{n}-1/4}}{2\sqrt{\pi n} \cdot n^{n/2}}$$

und

$$I_n = n![z^n]I(z) \sim \sqrt{2\pi n}\frac{n^n}{e^n}\frac{e^{n/2+\sqrt{n}-1/4}}{2\sqrt{\pi n}\cdot n^{n/2}} = \frac{n^{n/2}e^{\sqrt{n}-1/4}}{\sqrt{2}\,e^{n/2}}$$

Wenn wir $n = 50$ setzen, dann ist $I_n/n!$ etwa 9.17×10^{-31} und unsere Abschätzung ist etwa 8.82×10^{-31}.

Die eingeschränkte Sattelpunktmethode

Die Sattelpunktmethode vollständig anzuwenden ist im allgemeinen problemlos möglich, kann aber zu längeren Rechnungen führen. Mit wesentlich weniger Aufwand können wir eine obere Schranke erhalten, die zwar im allgemeinen wesentlich schlechter ist, als die Abschätzung durch die vollständige Sattelpunktmethode, aber oft ausreicht. Die Idee, wie man viel Rechnerei sparen kann, ist sehr einfach: Wir ersetzen ein kompliziertes Integral, welches entlang eines Kreises insgesamt recht flach und dann steil über den Sattelpunkt führt durch ein Integral einer *konstanten* Funktion über den selben Weg. Wählen wir die Konstante wenigstens so groß, wie der größte ursprüngliche Funktionswert entlang des Integrationsweges, dann ist klar, daß wir den tatsächlichen Wert von oben abschätzen. Abbildung 5.7 illustriert diese Idee. Der Idealfall ist natürlich, daß das Maximum (des Betrags) der Funktion im Sattelpunkt angenommen wird. Der zweite Teil des folgenden Theorems beruht auf hinreichenden Bedingungen die genau dies garantieren, während der erste Teil immer gilt.

Theorem 12. Es sei $f(z)$ im Ursprung analytisch, die Koeffizienten $[z^n]f(z)$ seien nicht-negativ und der Konvergenzradius for f sei R. Weiter sei $f(0) \neq 0$ und es gebe unendlich viele $[z^n]f(z) \neq 0$.

1. Es gilt $[z^n]f(z) \leq \inf_{0<r<R} f(r)/r^n$.

2. Falls $\lim_{r\to R-} f(r) = +\infty$, dann hat die Gleichung $\zeta f'(\zeta) = nf(\zeta)$ für große n eine eindeutige Lösung $\zeta(n)$ in $(0, R)$ und $[z^n] \leq f(\zeta(n))\,\zeta(n)^{-n}$.

Proof. Der erste Teil des Satzes sagt ja im wesentlichen, daß wenn wir einen Kreis mit Radius r wählen, dann finden wir das Maximum der Funktion $|f(z)/z^n|$ auf diesem Kreis genau auf der reellen Achse. Da alle Koeffizienten nicht-negativ sind, gilt

$$\max_{|z|=r} |f(z)| \leq \max_{|z|=r} \sum_{n=0}^{\infty} f_n |z^n| = f(r).$$

Wir können jetzt folgendermaßen rechnen:

$$f_n \leq \left| \frac{1}{2\pi i} \int_{|z|=r} \frac{f(z)}{z^{n+1}} \, dz \right| \leq \frac{1}{r^n} \max_{|z|=r} |f(z)| = \frac{f(r)}{r^n}$$

Für den zweiten Teil müssen wir jetzt zeigen, daß $f(r)/r^n$ bei $\zeta(n)$ ein Minimum hat und daß $\zeta(n)$ eindeutig ist. Falls $r \to 0+$ oder $r \to R-$, dann $f(r)/r^n \to +\infty$. Es gibt also ein Minimum in $(0, R)$. Dieses Minimum ist eindeutig, falls die Funktion im ganzen Intervall linksgekrümmt ist. Betrachten wir die zweite Ableitung

$$\frac{r^2 f''(r) - 2nrf'(r) + n(n+1)f(r)}{r^{n+2}} = \frac{1}{r^{n+2}} \sum_{m=0}^{\infty} (n+1-m)(n-m) f_m r^m,$$

dann sieht man leicht, daß sie für $r \in (0, R)$ positiv ist: Sowohl f_m als auch $(n+1-m)(n-m)$ sind nicht-negativ und für unendlich viele m sogar positiv.

Zuletzt ist klar, daß dieses eindeutige Minimum bei $\zeta(n)$ liegen muß, da dieses der Punkt ist, an dem die erste Ableitung von $f(r)/r^n$ verschwindet.

□

Exercises

5.1 Ordnen Sie die drei Folgen mit diesen EGFs nach ihrem asymptotischen Wachstum: $A(z) = 1/\sqrt{1 - z/2}$, $B(z) = (1+z)/(1-z)$ und $C(z) = \frac{1}{1-e^{-z-1/3}}$.

5.2 Was ist das exponentielle Wachstum der Folge mit dieser erzeugenden Funktion:

$$\frac{z^3 - 11z^2 + 39z - 45}{x^5 - 4\,x^4 - 24\,x^3 + 160\,x^2 - 304\,x + 192}$$

5.3 Sie sind ein begeisterter Gärtner und immer auf der Suche nach Schnäppchen. Wie sie wissen, setzt sich der Normalpreis für geordnete gewurzelte Bäume zusammen aus 3 Euro pro internem Knoten und 1 Euro pro Blatt. Nun macht Ihnen die lokale Baumschule folgendes Angebot: Alle Bäume, die normalerweise n Euro pro Stück kosten, für zusammen nur 2^n Euro! Als Sammler interessieren Sie sich

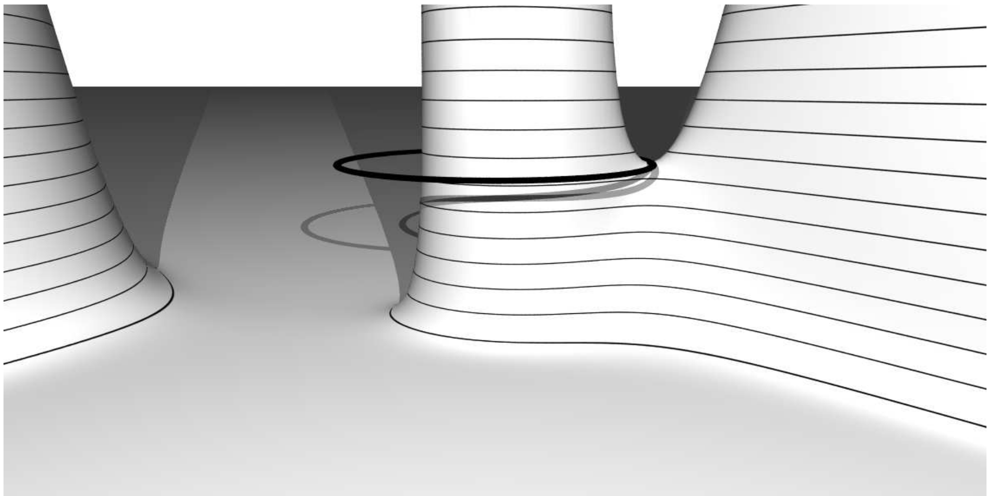


Figure 5.7: Das Integral über eine konstante Funktion, ist eine obere Schranke für das tatsächliche Kurvenintegral.

zunächst nur für Exklusives und wollen daher wissen, ob sich dieses Angebot für möglichst teuere Bäume rechnet. Glücklicherweise erinnern Sie sich an die Erzeugendenfunktion der Anzahl der Bäume mit m internen und n externen Knoten:

$$T(u, z) = -\frac{\sqrt{z^2 + (-2u - 2)\,z + u^2 - 2u + 1} + z - u - 1}{2}$$

5.4 Bestimmen Sie $[z^n]\frac{1}{2-e^z}$ bis auf $O(12^{-n})$ genau.

5.5 Wie teuer müssen die Einzelbäume mindestens sein, damit sich das Angebot aus der letzten Aufgabe lohnt? Verwenden Sie Maxima.

5.6 Wie läßt sich D_n mithilfe von Stirlingzahlen darstellen?

5.7 Finden Sie eine bessere Näherung für L_n als $2 + O(8^{-n})$ indem sie die nächsten beiden Singularitäten berücksichtigen.

5.8 Berechnen Sie eine Näherung für $1/n!$ indem Sie die Sattelpunktmethode auf e^z anwenden und einen rechteckigen Integrationsweg wählen.

Appendix A

Solutions to Selected Exercises

1.1 quite easy...

1.6 Die Zahl S_N birgt lediglich die Fragestellung, wie oft das Quicksort-Programm etwas auf den Stack legt. Wenn das zu sortierende Array weniger als $2M+3$ Zellen hat, wird höchstens auf einem der beiden Unterarrays, die durch die Partitionierung entstanden sind, weitergerechnet — also überhaupt nichts auf den Stack gelegt. Wenn das Array hingegen exakt $2M+3$ Zellen hat, kann es passieren, dass die Partitionierung genau in der Mitte stattfindet. Dann haben beide Unterarrays die Größe $M+1$, so dass ein entsprechendes Intervall auf den Stack gepusht wird. Die Wahrscheinlichkeit dafür, dass die Teilung genau in der Mitte passiert, beträgt natürlich $\frac{1}{2M+3}$.

Im Allgemeinen gilt, dass genau dann etwas auf den Stack gelegt werden muss, wenn beide Unterarrays mindestens die Größe $M+1$ haben. Es gibt $N-2M-2$ natürliche Zahlen k mit $M+1 < k < N-M$, so dass bei zufälliger Wahl eines Indizes mit Wahrscheinlichkeit $\frac{N-2M-2}{N}$ etwas auf den Stack gepusht wird. Für $N \geq 2M+3$ gilt somit

$$S_N = \frac{2}{N}\sum_{k=0}^{N-1} S_k + \frac{N-2M-2}{N}.$$

Ab jetzt verwenden wir wieder die üblichen Tricks und erhalten für $N \geq 2M+4$:

$$
\begin{aligned}
NS_N - (N-1)S_{N-1} &= 2S_{N-1} + 1 \\
NS_N &= (N+1)S_{N-1} + 1 \\
\sigma_N &:= \frac{S_N}{N+1} = \sigma_{N-1} + \frac{1}{N(N+1)} \\
\sigma_N &= \sum_{k=2M+4}^{N} \frac{1}{k(k+1)} + \sigma_{2M+3} = \frac{1}{2M+4} - \frac{1}{N+1} + \sigma_{2M+3} \\
S_N &= \frac{N+1}{2M+4} - 1 + \frac{(N+1)}{(2M+3)(2M+4)} = \frac{N+1}{2M+3} - 1
\end{aligned}
$$

1.10 A program with counters added can be found in Figure A.1. The output of the program is:

```
N=10 M=3
A=12337920
B=10284000
C=98717760
D=6652800
E=7983360
S=806400
```

1.12 Anweisung $a[i] > max$ wird sowieso N-mal durchlaufen. Da *max* negativ initialisiert wurde, muß es selbst im *best case* einmal überschrieben werden, was in Anweisung $max = a[i]$ passiert. Wenn das Array direkt mit dem Maximum beginnt, reicht demnach ein einziges Update von *max* im Schritt $i = 0$. Im schlimmsten Fall hingegen fällt der Vergleich immer positiv aus. Dann werden die beiden Anweisungen jeweils N-mal durchlaufen. Voraussetzung ist natürlich, daß das Array aufsteigend sortiert vorliegt.

Der *best case* tritt auf, wenn schon das erste Element das Maximum ist. Die Wahrscheinlichkeit hierfür ist natürlich $1/N$. Der *worst case* hingegen tritt nur mit Wahrscheinlichkeit $1/N!$ auf, da von den $N!$ Permutationen unter N verschiedenen Zahlen nur eine einzige komplett aufsteigend sortiert ist.

Der Vergleich wird wie gehabt N-mal durchgeführt; für die Anweisung $max = a[i]$ sieht es etwas komplizierter aus. Insbesondere sind die Wahrscheinlichkeiten, daß für zwei verschiedene Belegungen von i jeweils die Zuweisung durchgeführt wird, nicht unabhängig. Dieses Problem können wir aber mit der Linearität des Erwartungswertes erschlagen.

Die Wahrscheinlichkeit, daß das k-te Element (bei Zählung von 1 bis N) größer ist als seine $k-1$ Vorgänger, beträgt $1/k$. Dies kann man sich auch so veranschaulichen: unter den $k!$ Permutationen der ersten k Elemente gibt es genau

```c
#include"rand.c"
#include <stdio.h>
#define push(a, b) {s1[sp] = a; s2[sp++] = b;}
#define pop(a, b) {a = s1[--sp]; b = s2[sp];}
#define stack_is_empty (sp == 0)


int sp = 0, s1[50], s2[50];
int a[10001];
int N, M, A, B, C, D, E, S;

void quicksort(int N)
{
    int i, j, l, r, k, t;
    l = 1; r = N;
    if(N > M)
      while(1) {
        i = l - 1; j = r; k = a[j];
        A++; B--;
        do {
            B++;
            do {C++; i++;} while(a[i] < k);
            do {C++; 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) {S++; 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) {S++; 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]) {
            D++; k = a[i]; j = i;
            do {E++; a[j] = a[j - 1]; j--;} while(a[j - 1] > k);
            a[j] = k;
        }
}
```

$(k-1)!$ viele, bei denen das größte Element an der k-ten Stelle steht. Die erwartete Anzahl der ausgeführten Zuweisungen beträgt demnach

$$\sum_{i=0}^{N-1} \frac{1}{i+1} = \sum_{k=1}^{N} \frac{1}{k} = H_N = \ln N + \gamma + o(1),$$

wobei H_N die N-te harmonische Zahl und γ etwa 0.78 ist.

1.13 Die Blöcke `A` und `F` werden offensichtlich jeweils einmal durchlaufen. Der Block `B` wird für jedes i von 0 bis N ausgeführt, also $N+1$-mal. Da der Sprung nach done nur für $i = N$ erfolgt, wird der Block `C` genau N-mal betreten. Die Anzahl der Durchläufe von `D` ergibt sich gemäß des letzten Teils der ersten Tutoraufgabe als H_N. Der Block `E` schließlich wird genauso oft betreten wie Block `C` (wobei `D` entweder durchlaufen oder übersprungen wird).

Block	Durchläufe	Kosten	Blockkosten	kum. Gesamtkosten
`A`	1	2	2	2
`B`	$N+1$	2	$2N+2$	$2N+4$
`C`	N	3	$3N$	$5N+4$
`D`	H_N	1	H_N	$5N+H_N+4$
`E`	N	2	$2N$	$7N+H_N+4$
`F`	1	1	1	$7N+H_N+5$

Im *average case* werden also $7N + H_N + 5$ Instruktionen ausgeführt; das sind $7N + \ln N + 5.78 + o(1)$ viele. Zur Einschätzung: der natürliche Logarithmus von einer Milliarde liegt unter 21.

1.14 Es bezeichne K_n die Anzahl der gültigen Kombinationen. Ferner bezeichne R_n, G_n bzw. B_n die Anzahl der Kombinationen, wenn das erste Lied aus dem Genre Rock, Gabba bzw. Blues stammt. Dann ergeben sich die folgenden Gleichungen:

- $K_n = R_n + G_n + B_n$
- $R_n = R_{n-1} + B_{n-1}$
- $G_n = R_{n-1} + B_{n-1}$
- $B_n = 1$

Wiederholtes Einsetzen liefert sehr schnell die Lösung $R_n = G_n = n$ und somit $K_n = 2n + 1$.

1.15 Die Zeilen 1, 2 und 7 werden natürlich jeweils einmal durchlaufen, die Zeile 3 genau $(N+1)$-mal. Die Zeilen 4 und 6 werden deshalb N-mal ausgeführt. Für jeden Durchlauf der Zeile 4 wird mit Wahrscheinlichkeit $\lceil N/2 \rceil / N$ Zeile 5 ausgeführt.

Für gerade N ist dies $\frac{1}{2}$, für ungerade N hingegen $\frac{1}{2} + 1/2N$. Der Erwartungswert ist hier also $\lceil N/2 \rceil$.

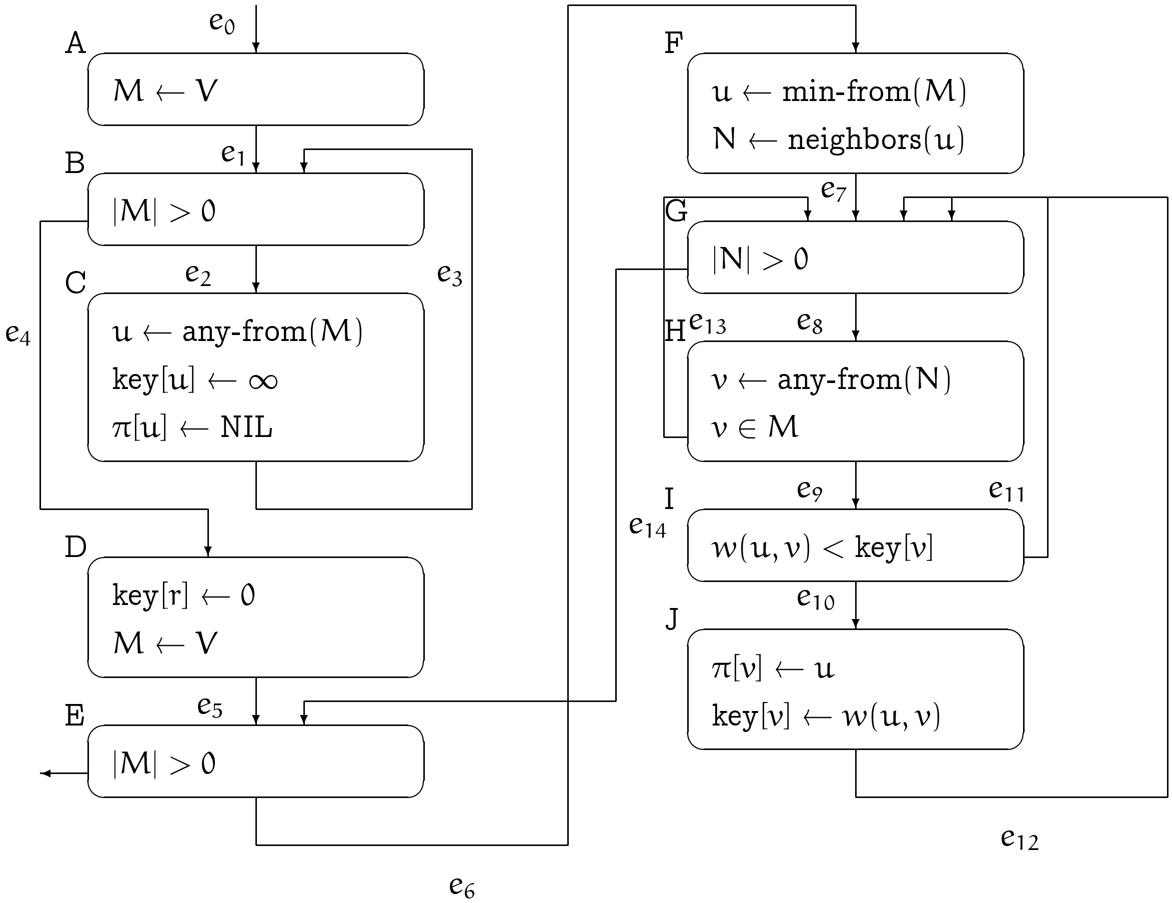
1.16 Ohne Beschränkung der Allgemeinheit beginne w mit a. Wenn auch w_2 ein a ist, was mit Wahrscheinlichkeit $\frac{1}{2}$ passiert, haben wir bereits mit einer Iteration ein Palindrom gefunden. Andernfalls haben wir den Präfix ab. In der folgenden Iteration finden wir genau dann ein Palindrom, wenn ein a auftaucht; ansonsten haben wir den Präfix abb. Es ist leicht einzusehen, dass wir im Folgenden genau in dem Moment ein Palindrom vorliegen haben, in dem ein a als letztes betrachtetes Zeichen vorkommt. Die erwartete Anzahl der Iterationen ist also

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

wobei der letzte Summand daher rührt, dass auch für $w = ab^{n-1}$ genau $n-1$ Durchläufe der `while`-Schleife stattfinden. Mit Formel ?? ergibt sich also

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

2.3 Das Flussdiagramm sieht wie folgt aus. Die **for**-Schleifen sind jeweils etwas umgeschrieben worden, weil Initialisierung und Iterationsbedingung getrennt werden müssen.



Wir wählen den Spannbaum $e_1, e_2, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}$. Somit ergeben sich die folgenden fundamentalen Zyklen:

$$
\begin{aligned}
C_0 &= e_0 + e_1 + e_4 + e_5 \\
C_3 &= e_3 + e_2 \\
C_{11} &= e_{11} + e_8 + e_9 \\
C_{12} &= e_{12} + e_8 + e_9 + e_{10} \\
C_{13} &= e_{13} + e_8 \\
C_{14} &= e_{14} + e_6 + e_7
\end{aligned}
$$

Der Lösungsraum enhält alle Linearkombinationen dieser sechs Vektoren:

E_0	1	0	0	0	0	0
E_1	1	0	0	0	0	0
E_2	0	1	0	0	0	0
E_3	0	1	0	0	0	0
E_4	1	0	0	0	0	0
E_5	1	0	0	0	0	0
E_6	0	0	0	0	0	1
E_7	0	0	0	0	0	1
E_8	0	0	1	1	1	0
E_9	0	0	1	1	0	0
E_{10}	0	0	0	1	0	0
E_{11}	0	0	1	0	0	0
E_{12}	0	0	0	1	0	0
E_{13}	0	0	0	0	1	0
E_{14}	0	0	0	0	0	1

Wir wählen sechs unabhängige Zeilen und erhalten das Gleichungssystem

$$\begin{pmatrix} E_0 \\ E_2 \\ E_{11} \\ E_{12} \\ E_{13} \\ E_{14} \end{pmatrix} = \begin{pmatrix} 1 \\ C \\ I - J \\ J \\ H - I \\ G - H \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix}.$$

Die Lösung ist offensichtlich. Wir sehen, dass $E_0 = E_1 = E_4 = E_5$, $E_2 = E_3$, $E_6 = E_7 = E_{14}$, $E_{10} = E_{12}$. Damit ist $A = 1$, $B = E_1 + E_3 = C + 1$, $D = E_4 = 1$, $E = E_5 + E_{14} = 1 + G - H$, $F = E_6 = G - H$, und wir müssen für eine vollständige Analyse nur die Werte C, G, H, I, J ausrechnen.

3.1 Wir nehmen erstmal den Logarithmus und erhalten

$$\log c_n = \log c_{n-1} \cdot \log c_{n-2}.$$

Jetzt nehmen wir nochmal den Logarithmus, um statt des Produktes eine Summe zu haben, und bekommen

$$\log\log c_n = \log\log c_{n-1} + \log\log c_{n-2}.$$

Eine Substitution mit $d_n = \log\log c_n$ liefert die folgenden Gleichungen.

$$\begin{aligned} d_0 &= 0 \\ d_1 &= 1 \\ d_n &= d_{n-1} + d_{n-2} \end{aligned}$$

Jetzt ist große Freude in unseren Herzen, denn wir erkennen die Gleichheit $c_n = 2^{2^{F_n}}$, wobei F_n für die n-te Fibonacci-Zahl stehen soll.

3.2 Für eine gerade Anzahl von Rekruten werden natürlich überhaupt keine Iterationen durchgeführt, ansonsten ergibt sich wieder eine Rekursionsgleichung:

$$I_n = \begin{cases} 0 & \text{falls } n \text{ gerade} \\ 1 + \sum_{k=0}^{n} \binom{n}{k} 2^{-n} I_k & \text{sonst} \end{cases}$$

Allerdings ist die Wahrscheinlichkeit, dass bei einer Scheuch-Iteration eine ungerade Anzahl von Soldaten erkrankt, also eine gerade Anzahl von gesunden Rekruten übrig bleibt, schlichtweg $\frac{1}{2}$. Daher ist die genaue Zahl der Soldaten gar nicht entscheidend, und für ungerade n gilt $S_n = S_1 = 1 + \frac{1}{2}S_1$, also ist $S_n = 2$ für jedes ungerade n.

3.3 Das charakteristische Polynom ist hier $x^2-4x+4 = (x-2)^2$ mit der doppelten Nullstelle $x_0 = 2$. Daher hat die Lösung die Form $a_n = \lambda 2^n + \mu n 2^n$. Mit den Nebenbedingungen ergibt sich $a_0 = 0 = \lambda$ und $\mu = \frac{a_1}{2} = \frac{3}{2}$.

3.4 Das charakteristische Polynom ist hier $x^3 - 3x^2 + 4x - 12$. Wir erraten die erste Nullstelle $x_0 = 3$. Mittels Polynomdivision durch $(x-3)$ ergibt sich $x^2 + 4$. Die beiden anderen Lösungen sind natürlich $x_1 = 2i$ und $x_2 = -2i$. Noch weniger Arbeit haben wir, falls wir Maxima bemühen:

```
(C1) solve(z^3-3*z^2+4*z-12=0,z);
(D1)                    [z = - 2 %I, z = 2 %I, z = 3]
```

Folglich erhalten wir die Form $b_n = \lambda 3^n + \mu_1(2i)^n + \mu_2(-2i)^n$. Da wir keine nichtreellen Lösungen erwarten (es geht um Laufzeiten), fordern wir $\mu_1 = \mu_2 =: \mu \in \mathbf{R}$ oder $\mu_1 = -\mu_2 = i\rho$ mit $\rho \in \mathbf{R}$ und schreiben $b_n = \lambda 3^n + \mu 2^n(i^n + (-i)^n) - \rho 2^n i(i^n - (-i)^n)$. Mit der bekannten Formel $e^{i\phi} = \cos\phi + i\sin\phi$, bzw. den Spezialfällen $i = e^{\frac{\pi}{2}i}$ und $-i = e^{-\frac{\pi}{2}i}$ erhalten wir

$$\begin{aligned} i^n + (-i)^n &= e^{ni\pi/2} + e^{-ni\pi/2} \\ &= \cos(n\pi/2) + i\sin(n/2) + \cos(-n/2) + i\sin(-n/2) \\ &= 2\cos(n/2) \end{aligned}$$

und genauso

$$\begin{aligned} i^n - (-i)^n &= e^{ni\pi/2} - e^{-ni\pi/2} \\ &= \cos(n\pi/2) + i\sin(n\pi/2) - \cos(-n\pi/2) - i\sin(-n\pi/2) \\ &= 2i\sin(n\pi/2) \end{aligned}$$

und damit insgesamt $b_n = \lambda 3^n + \mu 2^n \cos(n/2) + \rho 2^n \sin(n/2)$.

Die Nebenbedingungen ergeben $a_0 = \lambda + \mu = 1$, $a_1 = 3\lambda + 2\rho = 1$ sowie $a_2 = 9\lambda - 4\mu = 1$ und daher $\lambda = \frac{5}{13}, \mu = \frac{8}{13}, \rho = -\frac{1}{13}$.

3.6 Zuerst die Gleichung. Sie lautet $R_0 = 0$ und, für $n \geq 1$,

$$R_n = \binom{n}{2} + \sum_{k=0}^{n-1} R_k.$$

Wir können beispielsweise das folgende, nichtminimale Repertoire benutzen:

R_0	R_n	$f_n = R_n - \sum_{k=0}^{n-1} R_k$
1	a^n	$\frac{2-a}{1-a} a^n - \frac{1}{1-a}$
1	2^n	1
1	1	$1-n$
0	n	$n - \binom{n}{2}$

Um $\binom{n}{2}$ zu erhalten, greifen wir auf die letzte Zeile zurück. Zum Ausgleich verwenden wir dann noch die dritte und abschließend die zweite Zeile. Wir erhalten somit:

$$-\left(n - \binom{n}{2}\right) - (1-n) + 1 = \binom{n}{2}$$

Weil der Ausdruck aus der dritten Zeile subtrahiert, der Ausdruck aus der zweiten Zeile jedoch addiert wurde, ist die Anfangsbedingung $R_0 = 0$ erfüllt. Als Ergebnis erhalten wir

$$R_n = 2^n - n - 1.$$

Der Test für die ersten Indizes verläuft dann positiv.

3.8 Es gilt

$$\begin{aligned}
a_{n+2} + a_{n+1} - n^2 a_n &= (E-n)\underbrace{(E+n)a_n}_{b_n} \\
(E-n)b_n &= n! \\
b_{n+1} - n b_n &= n! \\
b_n &= n! \\
(E+n)a_n &= n! \\
a_{n+1} + n a_n &= n! \\
a_n &= \frac{(n-1)!}{2}
\end{aligned}$$

für $n > 0$ und $a_0 = 8000$.

3.9 Wir bezeichnen die Zählerstände nach dem $n$ten Durchlauf mit i_n bzw. j_n. Für $i_0 > j_0$ wird das Schleifeninnere nie ausgeführt. Sei im Folgenden $0 < i_0 \leq j_0$. Dann erhalten wir die Rekursion

$$\begin{aligned} i_n &= i_{n-1} + j_{n-1} \\ j_n &= j_{n-1} + 10 \end{aligned}$$

und durch Einsetzen ergibt sich die Lösung

$$\begin{aligned} j_n &= j_0 + 10n \\ i_n &= i_{n-1} + 10(n-1) + j_0 \\ &= i_0 + \sum_{k=1}^{n} (10(k-1) + j_0) \\ &= i_0 + 5n(n-1) + nj_0. \end{aligned}$$

Die Schleife wird ausgeführt, solange $i_n - j_n \leq 0$, das heißt für

$$5n^2 + (j_0 - 15)n + i_0 - j_0 \leq 0.$$

Für positive n gilt das, solange

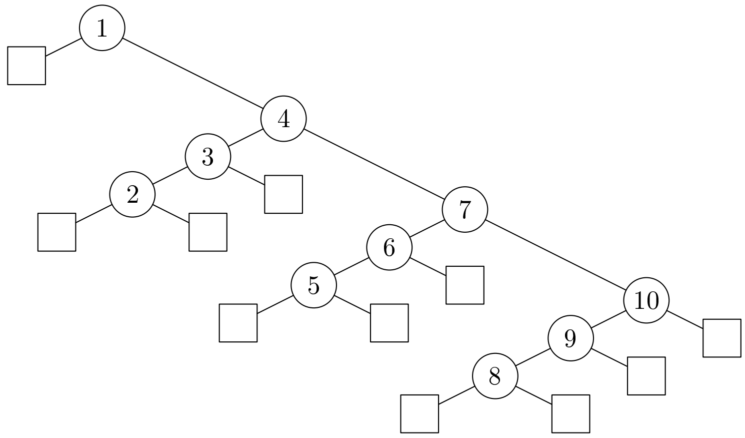
$$n \leq \frac{15 - j_0 + \sqrt{(j_0 - 15)^2 - 20(i_0 - j_0)}}{10} =: a(i_0, j_0).$$

Die Schleife wird in diesem Fall also $\lfloor a(i_0, j_0) \rfloor + 1$ mal ausgeführt.

3.10 Für $i \leq j \leq 0$ terminiert die Schleife nicht. Es ist also nur der Fall interessant, daß $i \leq 0 < j$. Dann ist $0 < i_{n_1} = i_0 + \lfloor -i_0/j_0 + 1 \rfloor j_0 \leq j_{n_1} = j_0$ nach $n_1 = \lfloor -i_0/j_0 \rfloor + 1$ Schritten, und es tritt der erste Fall aus der Tutorübung ein. Das heißt, es gibt jetzt noch $n_2 = \lfloor a(i_{n_1}, j_{n_1}) \rfloor + 1$ zusätzliche Durchläufe und insgesamt wird die Schleife $n_1 + n_2$ mal durchlaufen.

3.13

a) Die Pfadlängen sind $\pi(T) = 0 + 1 + 2 + 2 + 3 + 3 + 3 + 4 + 4 + 5 = 27$ und $\xi(T) = 1 + 3 + 4 + 4 + 4 + 4 + 5 + 5 + 5 + 6 + 6 = 47$:



b) Es genügt, π zu ermitteln. Dann ist

$$C^+ = \frac{\pi(T)}{n} + 1 \text{ und } C^- = \frac{\pi(T)+2n}{n+1}.$$

Es gilt

$$\begin{aligned}
\pi(T) &= \sum_{i=1}^{k} \sum_{j=i}^{i+m-1} j \\
&= \sum_{i=1}^{k} \frac{(i+m)(i+m-1)}{2} - \frac{i(i-1)}{2} \\
&= \frac{1}{2} \sum_{i=1}^{k} (2mi + m(m-1)) \\
&= \frac{mk(k+1) + km(m-1)}{2} \\
&= \frac{km^2 + mk^2}{2}.
\end{aligned}$$

Test am Beispiel aus a): $\pi(T) = (3*9+3*9)/2 = 27$.

c) Die Suchtiefe (im average case) hängt in beiden Fällen linear von $\pi(T)$ ab. Daher gilt es, diese Größe zu minimieren. Wir drücken π mit Hilfe von $n' := n-1 = km$ bis auf die Konstante aus als $kn' + n'^2/k$ und leiten nach k ab, um die Gleichung $n' - n'^2/k^2 = 0$ zu erhalten. Auflösen nach k liefert $k = \sqrt{n}'$. Im Fall $n = k^2 + 1$ ist das genau das Optimum. Ansonsten wird man aus Symmetriegründen die nächste ganzzahlige Lösung wählen.

3.15 Das erste Programm wurde schon behandelt, und wir kennen daher die Größen

The first program was handled already and so we know the sizes $C^- = \lfloor \log(n+1) \rfloor + 2 - 2^{1-\{\log(n+1)\}}$ and $C^+ = \ldots$. Bei einer erfolglosen Suche werden $2C^-$ viele **if**-Anweisungen ausgeführt, da es zu C^- Durchläufen kommt, wobei jeder Durchlauf zwei **if**-Anweisungen umfasst. Bei einer erfolgreichen Suche hingegen sind es $2C^+ - 1$ viele **if**-Anweisungen, da die zweite **if**-Anweisung im letzten Durchlauf nicht mehr erreicht wird.

At the unsuccessful search $2C^-$ **if**-instructions are exported because it comes to C^- flows where each flow contains two **if**-instructions. At the successful search, however, we have $2C^+ - 1$ **if**-instructions because the second **if**-instructions in the last flow is not achieved anymore.

Sehen wir uns jetzt *binsearch2* an. Es bezeichne B_n die Anzahl der ausgeführten **if**-Anweisungen, wenn der betrachtete Teil des Arrays noch $n = r - l + 1$ Elemente

umfasst. Wenn weniger als drei Elemente betrachtet werden, wird die **while**-Schleife sofort übersprungen. Also gilt im Falle der erfolglosen Suche $B_1 = B_2 = 2$.
Now we look at the *binsearch2*. Let B_n indicate the amount of exported **if**-instructed when the observed part of the array still contains $n = r - l + 1$ elements. If less than three elements are observed, the **while**-loop will be skipped immediately. So in case of the unsuccessful search $B_1 = B_2 = 2$ holds.
Für größere n wird der Rumpf der **while**-Schleife betreten und eine **if**-Anweisung ausgeführt. Falls $a[m] > v$ gilt, müssen wir links weitersuchen, sonst rechts. Danach wird das zu untersuchende Teilarray kleiner, entweder $\lfloor n/2 \rfloor$ oder $\lceil n/2 \rceil$. Wir stellen die beiden Rekursionsgleichungen

$$\overline{B}_n = \overline{B}_{\lceil n/2 \rceil} + 1$$
$$\underline{B}_n = \underline{B}_{\lfloor n/2 \rfloor} + 1$$

für $n > 2$ und $\overline{B}_n = \underline{B}_n = 2$ für $n \le 2$ auf. Dann gilt sicherlich

$$\underline{B}_n \le B_n \le \overline{B}_n.$$

Wir können leicht geschlossene Formen für $\underline{B}$ und $\overline{B}$ finden. Es gilt ja sicher $\underline{B}_{2^k} = k+1$ für $k > 0$ und allgemein $\underline{B}_n = \lfloor \log(n) \rfloor + 1$ für $n > 2$, $\underline{B}_{2^k+r} = \underline{B}_{2^k}$ für $r < 2^k$.
For bigger n the body of the **while**-loop is entered and an **if**-instructions is exported. If $a[m] > v$ holds, we have to keep searching on the left, otherwise on the right. Then the partial array to be examined becomes smaller, either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$. We establish the following two recursion equations

$$\overline{B}_n = \overline{B}_{\lceil n/2 \rceil} + 1$$
$$\underline{B}_n = \underline{B}_{\lfloor n/2 \rfloor} + 1$$

for $n > 2$ and $\overline{B}_n = \underline{B}_n = 2$ für $n \le 2$. Then

$$\underline{B}_n \le B_n \le \overline{B}_n$$

most certainly holds. We can easily find compact forms for $\underline{B}$ and $\overline{B}$. For $k > 0$ $\underline{B}_{2^k} = k+1$ definitely holds and generally $\underline{B}_n = \lfloor \log(n) \rfloor + 1$ for $n > 2$, $\underline{B}_{2^k+r} = \underline{B}_{2^k}$ for $r < 2^k$.
Genauso gilt $\overline{B}_{2^k} = k + 1$ für $k > 0$. Hier ist aber $\overline{B}_{2^k+r} = \overline{B}_{2^k} + 1$ für $0 < r < 2^k$. Daher ist $\overline{B}_n = \lceil \log(n) \rceil + 1$ für $n > 2$.
Just as $\overline{B}_{2^k} = k+1$ holds for $k > 0$. Here, however, $\overline{B}_{2^k+r} = \overline{B}_{2^k} + 1$ for $0 < r < 2^k$. Therefore $\overline{B}_n = \lceil \log(n) \rceil + 1$ is for $n > 2$.

Insgesamt erhalten wir also $B_n = \log(n) + O(1)$.

Altogether, we get $B_n = \log(n) + O(1)$.

4.1

1. Die erzeugende Funktion der Folge (α^n) ist $\sum_{n\geq 0} \alpha^n z^n$, in geschlossener Form ist dies einfach $\frac{1}{1-\alpha z}$. Also ist die erzeugende Funktion von $a_n = 2^n + 3^n$ einfach $\frac{1}{1-2z} + \frac{1}{1-3z}$.

2. Wir beginnen mit (2^n) und $\frac{1}{1-2z}$. Differenzieren liefert $b_n = (n+1)2^{n+1}$ mit erzeugender Funktion $\frac{2}{(1-2z)^2}$.

3. Die Folge der $\binom{k}{n}$ hat die erzeugende Funktion $(1+z)^k$. Skalieren mit α liefert direkt die gesuchte Folge $c_n = \alpha^n \binom{k}{n}$ mit erzeugender Funktion $(1+\alpha z)^k$.

4. Wir wissen, dass die Folge $(n+1) = 1, 2, 3, 4, \ldots$ die erzeugende Funktion $\frac{1}{(1-z)^2}$ hat. Um von hier zu $d_n = -1, 0, 1, 2, 3, \ldots$ zu gelangen, verschieben wir zunächst zweimal nach rechts. Dies ergibt die Folge $0, 0, 1, 2, 3, 4, \ldots$ mit erzeugender Funktion $\frac{z^2}{(1-z)^2}$. Nun subtrahieren wir noch die Folge $1, 0, 0, \ldots$ und erhalten so d_n mit erzeugender Funktion $\frac{z^2}{(1-z)^2} - 1$.

5. Wir wissen immer noch, dass die Folge $(n+1) = 1, 2, 3, 4, \ldots$ die erzeugende Funktion $\frac{1}{(1-z)^2}$ hat. Wir verschieben nach rechts und erhalten (n) samt erzeugender Funktion $\frac{z}{(1-z)^2}$. Differenzieren ergibt die gesuchte Folge $e_n = (n+1)^2$ mit erzeugender Funktion $\frac{z+1}{(1-z)^3}$.

4.2 Wir müssen zunächst eine Form der Rekursionsgleichung finden, die für alle n (mit der Annahme, daß $f_n = 0$ für $n < 0$) gilt. Dafür muß im Fall $n = 0$ eine 1 addiert werden, was sich leicht mit dem Kroneckerdelta erreichen läßt. Die Gleichung lautet jetzt also

$$f_n = \delta_{n0} + \sum_{i=1}^{n} i f_{n-i}.$$

Dies können wir für alle n mit z^n multiplizieren und aufaddieren, so daß wir insgesamt

$$\sum_{0\leq n} f_n z^n = \sum_{0\leq n} (\delta_{n0} + \sum_{i=1} i f_{n-i}) z^n = 1 + \sum_{0\leq n} z^n \sum_{i=0} i f_{n-i}$$

erhalten. Der rechteste Term erinnert uns natürlich sofort an die Faltungsregel, und wir schreiben das Ganze als

$$F(z) = F(\frac{z}{(1-z)^2}) + 1$$

mit der Reihe aus Tutoraufgabe 2. Auflösen nach F liefert $F(z) = \frac{(1-z)^2}{z^2-3z+1}$.

4.3 $A(z) = 3^z = e^{z \ln 3} = \sum_{0 \le n} \frac{(\ln 3)^n z^n}{n!}$. Dazu gehört die Folge $1, \ln 3, \frac{(\ln 3)^2}{2}, \frac{(\ln 3)^3}{3!} \ldots$.
Mit der Newtonschen Formel inklusive Skalierung mit $-1/2$ ergibt sich die Folge aller $(-\frac{1}{2})^n \binom{-\frac{1}{2}}{n}$. Dies kann man auch weiter auflösen:

$$\begin{aligned}
(-\frac{1}{2})^n \binom{-\frac{1}{2}}{n} = (-\frac{1}{2})^n (-\frac{1}{2})^{\underline{n}} &= \frac{1}{n!4^n} 1 \cdot 3 \cdot 5 \cdots (2n-1) \\
= \frac{(2n)!}{n!4^n} \frac{1}{2^n n!} &= \binom{2n}{n} \frac{1}{8^n}
\end{aligned}$$

Wir addieren die Folge $1, 1, 1, \ldots$ und die gleiche um eins nach rechts verschobene Folge $0, 1, 1, 1, \ldots$. Das Ergebnis lautet $1, 2, 2, 2, \ldots$.

4.4

$$\begin{aligned}
\binom{1/2}{n} = \frac{(\frac{1}{2})^{\underline{n}}}{n!} &= \frac{1}{n!}(\frac{1}{2})(\frac{-1}{2})(\frac{-3}{2}) \cdots (\frac{-2n+3}{2}) \\
&= \frac{2^{-n}}{n!}(-1)(-3)(-5) \cdots (-2n+3) \\
&= (-1)^{n-1} \frac{2^{-n}}{n!} \underbrace{1 \cdot 3 \cdot 5 \cdots (2n-3)}_{n-1 \text{ Terme}} \\
&= (-1)^{n-1} \frac{2^{-n}}{n!} \frac{(2n-3)!}{\underbrace{2 \cdot 4 \cdot 6 \cdots (2n-4)}_{n-2 \text{ Terme}}} \\
&= (-1)^{n-1} \frac{2^{-n}}{n!} \frac{(2n-3)!}{2^{n-2}(n-2)!} \\
&= (-1)^{n-1} \frac{1}{4^{n-1}} \frac{(2n)! n(n-1)}{n!(2n)(2n-1)(2n-2)n!} \\
&= (-1)^{n-1} \frac{1}{4^{n-1}} \frac{1}{4} \frac{n(n-1)}{n(2n-1)(n-1)} \binom{2n}{n} \\
&= (-1)^{n-1} \frac{1}{4^n} \frac{1}{2n-1} \binom{2n}{n}
\end{aligned}$$

4.5 Die Grammatik ist eindeutig, so dass die gesuchte Anzahl der erzeugbaren Ausdrücke genau die Anzahl der möglichen Ableitungen mit n Schritten ist. Wenn nun ein S expandiert werden soll, stehen also die vier oben aufgeführten Möglichkeiten zur Verfügung, die drei, zwei, null beziehungsweise null S auf der rechten Seite tragen. Somit erhalten wir direkt die erzeugende Funktion

$$I(z) = zI(z)^3 + zI(z)^2 + 2z.$$

Ein Auflösen nach z ist hier offensichtlich einfacher als ein Auflösen nach $I(z)$. Es liefert

$$z = \frac{I(z)}{I(z)^3 + I(z)^2 + 2},$$

wir werden für die Lagrange-Inversion also die Funktion

$$f(t) = \frac{t}{t^3 + t^2 + 2}$$

verwenden. Das Ziel ist natürlich wieder der n-te Koeffizient:

$$\begin{aligned}
[z^n]I(z) &= \frac{1}{n}[u^{n-1}](\frac{u(u^3+u^2+2)}{u})^n \\
&= \frac{1}{n}[u^{n-1}](u^3+u^2+2)^n \\
&= \frac{1}{n}[u^{n-1}]\sum_{k=0}^{n}\binom{n}{k}\underbrace{(u^3+u^2)^k}_{(1+u)u^2}2^{n-k} \\
&= \frac{1}{n}[u^{n-1}]\sum_{k=0}^{n}\binom{n}{k}(1+u)^k u^{2k}2^{n-k} \\
&= \frac{1}{n}[u^{n-1}]\sum_{k=0}^{n}\binom{n}{k}2^{n-k}\sum_{m=0}^{k}\binom{k}{m}u^{m+2k}
\end{aligned}$$

Um den $(n-1)$-ten Koeffizienten bestimmen zu können, müssen wir uns hier überlegen, wann die Exponenten von u, also die $m+2k$, den Wert $n-1$ annehmen. Gleichsetzen liefert offenbar $m = n-1-2k$. Wir schreiben den Integer-

Wahrheitswert eines Ausdrucks A im Folgenden wieder als (A):

$$
\begin{aligned}
\ldots &= \frac{1}{n}[u^{n-1}]\sum_{k=0}^{n}\binom{n}{k}2^{n-k}\sum_{m=0}^{k}\binom{k}{m}u^{m+2k} \\
&= \frac{1}{n}\sum_{k=0}^{n}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}(0 \le n-1-2k \le k) \\
&= \frac{1}{n}\sum_{k=0}^{n}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}(2k \le n-1 \le 3k) \\
&= \frac{1}{n}\sum_{k}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}(2k \le n-1 \le 3k)(0 \le k \le n) \\
&= \frac{1}{n}\sum_{k}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}(0 \le 2k \le n-1 \le 3k \le 3n) \\
&= \frac{1}{n}\sum_{k}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}(2k \le n-1 \le 3k) \\
&= \frac{1}{n}\sum_{k}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}\left(k \ge \left\lceil\frac{n-1}{3}\right\rceil \wedge k \le \left\lfloor\frac{n-1}{2}\right\rfloor\right) \\
&= \frac{1}{n}\sum_{k=\lceil\frac{n-1}{3}\rceil}^{\lfloor\frac{n-1}{2}\rfloor}\binom{n}{k}2^{n-k}\binom{k}{n-1-2k}
\end{aligned}
$$

An dieser Stelle sollte man einige Werte für n einsetzen, um das Ergebnis zu überprüfen. Aus der Grammatik ergeben sich direkt die folgenden Anfangswerte:

n	0	1	2	3	4
i_n	0	2	0	4	8

4.6 Im Folgenden bezeichne p_n die Anzahl der gültigen Sequenzen der Länge n. In der Mengenschreibweise aus der Vorlesung gilt

$$P = (\uparrow P \downarrow)^*,$$

wie man nach kurzer Überlegung einsieht. Die erzeugende Funktion lautet demnach einfach

$$P(z) = \frac{1}{1 - zP(z)}.$$

Dies lässt sich direkt zu $1 - P(z) + zP(z)^2 = 0$ umformen, und das Auflösen dieser quadratischen Gleichung liefert (neben einer für uns uninteressanten) auch die folgende Lösung.

$$\begin{aligned}
P(z) &= -\frac{\sqrt{1-4z}-1}{2z} \\
&= -\frac{1}{2z}(\sum_{n\geq 0}\binom{\frac{1}{2}}{n}(-4)^n z^n - 1) \\
&= -\frac{1}{2}(\sum_{n\geq 0}\binom{\frac{1}{2}}{n}(-4)^n z^{n-1} - \frac{1}{z}) \\
&= -\frac{1}{2}(\sum_{n\geq -1}\binom{\frac{1}{2}}{n+1}(-4)^{n+1} z^{n} - \frac{1}{z})
\end{aligned}$$

Nun können wir den n-ten Koeffizienten bestimmen. Hierzu verwenden wir noch das Ergebnis aus Tutoraufgabe 1.

$$\begin{aligned}
p_n = [z^n]P(z) &= -\frac{1}{2}\binom{\frac{1}{2}}{n+1}(-4)^{n+1} \\
&= -\frac{1}{2}(-1)^n \frac{1}{4^{n+1}}\frac{1}{2n+1}\binom{2n+2}{n+1} \\
&= \frac{1}{4n+2}\binom{2n+2}{n+1}
\end{aligned}$$

4.7 Für den Fall *genau eine Spitze* haben wir natürlich $G =\uparrow G \downarrow \cup \uparrow\downarrow$. Für den Fall *mindestens zwei Spitzen* gilt demnach $M =\uparrow M \downarrow \cup \uparrow G^{\geq 2} \downarrow$. Die beiden Fälle sind disjunkt, so dass es naheliegt, $S = (G \cup M)^*$ zu schreiben. Dies gilt sogar tatsächlich, da die implizierte Grammatik trotz des Kleeneschen Abschlusses eindeutig ist (dies gälte nicht, wenn wir zum Beispiel $M =\uparrow M \downarrow \cup G^{\geq 2}$ vorliegen hätten).
Jetzt können wir die erzeugenden Funktionen direkt hinschreiben:

$$\begin{aligned}
G(u,z) &= zG(u,z) + uz \\
M(u,z) &= zM(u,z) + \frac{G(u,z)^2}{1-G(u,z)}z^2 \\
S(u,z) &= \sum_{k\geq 0}(G(u,z)+M(u,z))^k = \frac{1}{1-G(u,z)-M(u,z)}
\end{aligned}$$

4.8 Wir gehen von einer induktiven Definition von Bäumen aus: Ein Baum ist entweder ein externer Knoten oder aber ein interner Knoten mit mindestens einem, aber beliebig vielen Unterbäumen. Dies ergibt die Mengendarstellung

$$T = \square \cup \bigcirc \times T^+$$

und somit die erzeugende Funktion

$$T(u,z) = u + z\frac{T(u,z)}{1 - T(u,z)}.$$

Diese Gleichung müssen wir nun nach $T(u,z)$ auflösen. Diese Aufgabe dirigieren wir an `maxima` weiter: ein `solve(T=u+z*T/(1-T),T)` liefert uns die beiden Lösungen

$$-\frac{\sqrt{z^2 + (-2*u-2)*z + u^2 - 2*u + 1} + z - u - 1}{2}$$

und

$$+\frac{\sqrt{z^2 + (-2*u-2)*z + u^2 - 2*u + 1} - z + u + 1}{2}.$$

Wir wissen, dass für $u = z = 0$ eine 0 herauskommen muss. Folglich ist die erste der beiden Lösungen die weiterführende.

Um an die Koeffizienten zu gelangen, wollen wir die Taylor-Entwicklung von $T(u,z)$ betrachten. Deshalb geben wir `T:-((sqrt(z^ 2+(-2*u-2)*z+u^ 2-2*u+1)+z-u-1)/2);` und dann `taylor(T,[z,u],0,5);` ein. Wir lesen die Koeffizienten ab:

$n+m$	Term	Anzahlen
1	u	$1 \times (0\triangle 1)$ (innere $\triangle$ externe)
2	uz	$1 \times (1\triangle 1)$
3	$uz^2 + u^2z$	$1 \times (2\triangle 1)$, $1 \times (1\triangle 2)$
4	$uz^3 + 3u^2z^2 + u^3z$	$1 \times (3\triangle 1)$, $3 \times (2\triangle 2)$, $1 \times (1\triangle 3)$
5	$uz^4 + 6u^2z^3 + 6u^3z^2 + u^4z$	$1 \times (4\triangle 1)$, $6 \times (3\triangle 2)$, $6 \times (2\triangle 3)$, $1 \times (1\triangle 4)$

Zur Prüfung auf Symmetrie betrachten wir $T(u,z) - T(z,u)$. Wenn diese Differenz genau 0 wäre, hätten wir die Symmetrie. In `maxima` geben wir ein:

```
TT(u,z):=-((sqrt(z\^ 2+(-2*u-2)*z+u\^ 2-2*u+1)+z-u-1)/2);
ev(TT(u,z)-TT(z,u),expand);
```

Es ergibt sich aber nicht etwa 0, sondern $u - z$. Dies bedeutet offenbar, dass die Symmetrie eigentlich vorliegt, aber für Bäume, die aus genau einem Knoten bestehen, nicht gilt. Natürlich gibt es einen Baum, der nur aus einem Blatt besteht, aber es gibt keinen Baum, der nur aus einem inneren Knoten besteht. Daher sind wir mit diesem Ergebnis vollkommen einverstanden.

5.1 Wir müssen nur absteigend nach dem Betrag der dominanten Singularität ordnen. $A_n \asymp 1/2^n$, $B_n \asymp 1$, $C_n \asymp 3^n$.

5.2 Der Nenner ist $(x-4)(x-2)^3(x+6)$ und die dominante Singularität demnach 2. Der $n$te Koeffizient der Folge ist daher $\asymp (1/2)^n$.

5.3 Weil wir interne Knoten dreifach berücksichtigen, lautet die Erzeugendenfunktion der Anzahl der Bäume, die n Euro kosten,

$$TT(z) := T(z^3, z) = -\frac{\sqrt{z^6 + (-2z-2)\,z^3 + z^2 - 2z + 1} + z^3 - z - 1}{2}.$$

Um die Rentabilität abzuschätzen, benötigen wir lediglich die Größe des exponentiellen Wachstums der Koeffizienten. Das ist aber genau der Kehrwert der dominanten Singularität, die sich nach der Vorlesung (Satz von Pringsheim) auf der reellen Achse befinden muß, weil wir nur nichtnegative Koeffizienten erwarten. Wir suchen also die kleinste reelle Nullstelle der Funktion $f(z) = z^6 + (-2z-2)\,z^3 + z^2 - 2z + 1$ unter der Wurzel. Diese können wir etwa mit dem Newton-Verfahren annähern. Wir vermuten, daß die Anzahl sich superpolynomiell entwickeln muß und fangen daher mit dem Startwert $z_0 = 1$ an. Iteriert wird nach der Formel $z_{i+1} := z_i - \frac{f(z_i)}{f'(z_i)}$. Unter die Grenze $\frac{1}{2}$ kommen wir schon nach wenigen Iterationen[1]. Also ist das exponentielle Wachstum größer als 2^n. Damit sind alle n-Euro-Bäume zusammen für ausreichend große n mehr als $n2^n$ Euro wert. Der Deal lohnt sich also zumindest für sehr teure Bäume.

5.4 In der Vorlesung wurde gezeigt, daß der erste Summand der Koeffizienten — aus der dominanten Singularität — $\frac{1}{2}(\frac{1}{\ln 2})^{n+1}$ beträgt. Wir untersuchen also die Singularitäten mit dem nächstgrößeren Betrag, $\ln 2 \pm 2\pi i$. Nach L'Hôpital gilt

$$\lim_{z \to \ln 2 \pm 2\pi i} \frac{z - \ln 2 \mp 2\pi i}{2 - e^z} = -\frac{1}{2}$$

und deshalb

$$S(z) \sim -\frac{1}{2}\frac{1}{z - \ln 2 \mp 2\pi i} = \frac{1}{2}\frac{1}{\ln 2 \mp 2\pi i}\frac{1}{1 - \frac{z}{\ln 2 \mp 2\pi i}}.$$

Nun können wir mit Theorem ?? die Koeffizienten ablesen:

$$\begin{aligned}
[z^n]S(z) &= \frac{1}{2}\left((\frac{1}{\ln 2})^{n+1} + (\frac{1}{\ln 2 + 2\pi i})^{n+1} + (\frac{1}{\ln 2 - 2\pi i})^{n+1}\right) + O(r)^{-n} \\
&= \frac{1}{2}\left((\frac{1}{\ln 2})^{n+1} + r^{n+1}(e^{i\phi(n+1)} + e^{-i\phi(n+1)})\right) + O(r)^{-n} \\
&= \frac{1}{2}\left((\frac{1}{\ln 2})^{n+1} + r^{n+1}\cos(\phi(n+1))\right) + O(r)^{-n}
\end{aligned}$$

[1] Maxima verrät uns den genauen Wert

$$z = \left(\frac{\sqrt{31}}{6\sqrt{3}} + \frac{29}{54}\right)^{\frac{1}{3}} + \frac{1}{9\left(\frac{\sqrt{31}}{6\sqrt{3}} + \frac{29}{54}\right)^{\frac{1}{3}}} - \frac{2}{3} \approx 0.46557123187677$$

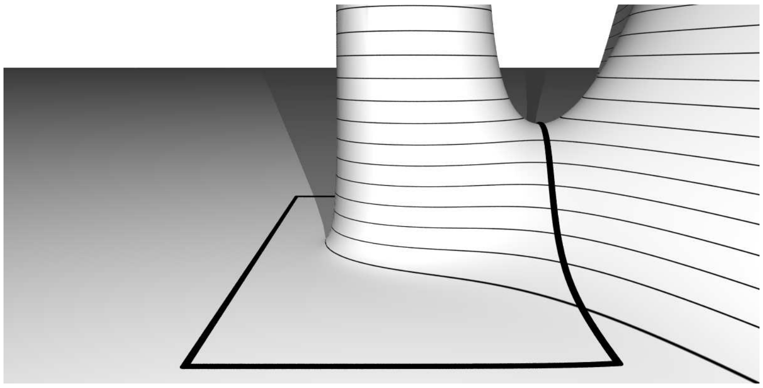


Figure A.2: Rechteckiger Integrationsweg.

mit $r = 1/\sqrt{\ln^2 2 + 4\pi^2} \approx 12.58547409739904$, $\phi = \arctan(\frac{2\pi}{\ln 2})$.

5.5 Wir wollen herausfinden, ab welchem n die Beziehung $|n\text{-Euro-Trees}| * n > 2^n$ gilt. Wir kennen schon die GF der Anzahl der n-Euro-Bäume. Wenn wir davon die GF von $\frac{2^n}{n}$ abziehen, dann müssen wir nur noch prüfen, wo die Koeffizienten positiv werden. Diese Reihe erhalten wir nach der Integrationsregel aus $\frac{1}{1-2z}$ als $-\ln(1-2z)$ — additive Konstanten sind uns dabei egal, sie haben ja nur auf Koeffizient null einen Einfluß. Schließlich liefert uns Maxima die Taylorreihe der Differenz wie folgt:

```
taylor(TT(z^3,z)+LOG(1-2*z),z,0,59);
```

$$
\begin{aligned}
&-2z - 2z^2 - \frac{5z^3}{3} - 3z^4 - \frac{27z^5}{5} - \frac{29z^6}{3} - \frac{114z^7}{7} - 28z^8 - \frac{449z^9}{9} - \frac{452z^{10}}{5} - \frac{1806z^{11}}{11} - \\
&\frac{898z^{12}}{3} - \frac{7152z^{13}}{13} - \frac{7128z^{14}}{7} - \frac{28388z^{15}}{15} - 3528z^{16} - \frac{112168z^{17}}{17} - \frac{111407z^{18}}{9} - \frac{442341z^{19}}{19} - \\
&\frac{219359z^{20}}{5} - \frac{1739102z^{21}}{21} - \frac{1722173z^{22}}{11} - \frac{6816098z^{23}}{23} - \frac{1684526z^{24}}{3} - \frac{26617557z^{25}}{25} - \frac{26259027z^{26}}{13} - \\
&\frac{103502663z^{27}}{27} - \frac{50931766z^{28}}{7} - \frac{400448097z^{29}}{29} - \frac{392965982z^{30}}{15} - \frac{1539917128z^{31}}{31} - 94114093z^{32} - \\
&\frac{5877808043z^{33}}{33} - \frac{5722604211z^{34}}{17} - \frac{22228275248z^{35}}{35} - \frac{10761068158z^{36}}{9} - \frac{83076532495z^{37}}{37} - \frac{79864793460z^{38}}{19} - \\
&\frac{305760847556z^{39}}{39} - \frac{72788756602z^{40}}{5} - \frac{1102326646900z^{41}}{41} - \frac{1035906138782z^{42}}{21} - \frac{3860335812782z^{43}}{43} - \\
&\frac{1779132981971z^{44}}{11} - \frac{12944637879497z^{45}}{45} - \frac{11570696492859z^{46}}{23} - \frac{40422002625916z^{47}}{47} - \frac{4271457879145z^{48}}{3} - \\
&\frac{110022149705736z^{49}}{49} - \frac{81582352665087z^{50}}{25} - \frac{204989522667554z^{51}}{51} - \frac{37778180087093z^{52}}{13} + \frac{250071962598454z^{53}}{53} + \\
&\frac{839276917951567z^{54}}{27} + \frac{5871575358163582z^{55}}{55} + \frac{2138599963933562z^{56}}{7} + \frac{45668117519922967z^{57}}{57} + \\
&\frac{57886811064368086z^{58}}{29} + \frac{283704070778308888z^{59}}{59} + \ldots
\end{aligned}
$$

Man sieht, daß sich das Geschäft ab 53 Euro pro Baum lohnt (also ab Anlagebeträgen von 2^{53} Euro).

5.8 Wir wählen ein Rechteck durch die Punkte $\pm iD$, $-S$ und R. Während wir D und S erst einmal offen lassen, setzen wir $R = n+1$, damit der Integrationspfad durch den Sattelpunkt läuft. Das gesuchte Integral setzt sich nun aus Teilintegralen aus den vier Strecken zusammen, deren Werte wir $I_1, \mathtt{ldots}, I_4$ nennen können.

Es bietet sich an, den Grenzwert $D \to \infty$ zu betrachten, denn verschwinden sofort die Teilintegrale

$$I_2 = \int_{R+iD}^{-S+iD} \frac{e^z}{z^{n+1}} dz \text{ und } I_4 = \int_{-S-iD}^{R-iD} \frac{e^z}{z^{n+1}} dz.$$

Es ist auch klar, daß

$$I_3 = \int_{-S-iD}^{-S+iD} \frac{e^z}{z^{n+1}} dz = O(e^{-S}$$

ist, so daß wir I_3 vernachlässigen können, wenn wir $S = n$ setzen.
Es bleibt das interessante Integral

$$I_1 = \int_{R-iD}^{R+iD} \frac{e^z}{z^{n+1}} dz,$$

welches wir wieder in zwei Teile zerlegen:

$$A = \int_{-\delta}^{\delta} \frac{e^{R+it}}{(R+it)^{n+1}} i\, dt$$

und

$$B = \int_{-\infty}^{\delta} \frac{e^{R+it}}{(R+it)^{n+1}} i\, dt + \int_{\delta}^{\infty} \frac{e^{R+it}}{(R+it)^{n+1}} i\, dt.$$

Schätzen wir zunächst einmal A ab:

$$iA = \int_{-\delta}^{\delta} \frac{e^{R+it}}{(R+it)^{n+1}}\, dt = \frac{e^R}{R^{n+1}} \int_{-\delta}^{\delta} e^{it-(n+1)\ln(1+it/R)} dt =$$

$$= \frac{e^R}{R^{n+1}} \int_{-\delta}^{\delta} e^{t^2/2(n+1)} \big(1 + O(t^3/n^2)\big)\, dt = \frac{e^R}{R^{n+1}} \big(1 + O(\delta^3/n^2)\big) \int_{-\delta}^{\delta} e^{t^2/2(n+1)}\, dt$$

Wir wählen δ so, daß $\delta^3/n^2 = o(1)$ und erhalten so

$$iA = \frac{e^R}{R^{n+1}} \sqrt{2(n+1)\pi}$$

durch eine einfache Substitution und dem üblichen Anhängen neuer Schwänze um das Integral von $-\infty$ bis ∞ nehmen zu können.
Wir erhalten nun insgesamt

$$1/n! = [z^n]e^z = \frac{1}{2\pi i} \oint \frac{e^z}{z^{n+1}} dz \sim \frac{1}{2\pi} A = \frac{1}{2\pi} \frac{e^R}{R^{n+1}} \sqrt{2(n+1)\pi}$$

Appendix B

MIPS Cheat Sheet

Add	`add`
Add Immediate	`addi`
Add Imm. Unsigned	`addiu`
Add Unsigned	`addu`
Subtract	`sub`
Subtract Unsigned	`subu`
And	`and`
And Immediate	`andi`
Nor	`nor`
Or	`or`
Or Immediate	`ori`
Xor	`xor`
Xor Immediate	`xori`
Shift Left Logical	`sll`
Shift Right Logical	`srl`
Shift Right Arithmetic	`sra`
Shift Left Logical Var.	`sllv`
Shift Right Logical Var.	`srlv`
Shift Right Arithmetic Var.	`srav`
Set Less Than	`slt`
Set Less Than Imm.	`slti`
Set Less Than Imm. Unsign.	`sltiu`
Set Less Than Unsigned	`sltu`

Branch On Equal	`beq`
Branch On Not Equal	`bne`
Branch Less Than	`blt`
Branch Greater Than	`bgt`
Branch Less Than Or Equal	`ble`
Branch Greater Than Or Equal	`bge`
Jump	`j`
Jump And Link	`jal`
Jump Register	`jr`
Jump And Link Register	`jalr`
Move	`move`
Load Byte	`l`
Load Byte Unsigned	`lbu`
Load Halfword	`lh`
Load Halfword Unsigned	`lhu`
Load Upper Imm.	`lui`
Load Word	`lw`
Load Immediate	`li`
Load Address	`la`
Store Byte	`sb`
Store Halfword	`sh`
Store Word	`sw`