Contents

1 Foundations **3**

1.1 A Machine Model 5

1.1.1 Higher-Level Programming Languages 6

1.1.2 Memory Management 7

1.1.3 Memory Hierarchy 8

1.2 Order of Growth 10

1.3 Simple Sums and Products 11

1.4 Recurrence Relations 11

Chapter 1

Foundations

We use computer algorithms to solve problems, e.g., to compute the maximum of a set of real numbers or to compute the product of two integers. A *problem* P consists of infinitely many *problem instances*. An instance of the maximum problem is e.g., to compute the maximum of the following five numbers $2, 7, 3, 9, 8$. An instance of the multiplication problem is, e.g., to compute the product of 257 and 123. We associate with every problem instance $p \in P$ a natural number $g(p)$, its *size*. Sometimes, the size will be a tuple of natural numbers; e.g., we measure the size of a graph by a pair consisting of the number of nodes and the number of edges. In the maximum problem we can define the size as the cardinality of the input set (5 in our example), in the multiplication problem we can define the size as the sum of the lengths of the decimal representations of the factors (6 in our example). Although the definition of size is arbitrary, there is usually a natural choice.

Execution of a program on a machine requires resources, e.g., time and space. Resource requirements depend on the input. We use $T_A(p)$ to denote the *running time* of algorithm A on problem instance p. We can determine $T_A(p)$ by experiment and measure it in milliseconds.

Global information about the resource requirements of an algorithm is in general more expressive than information about resource requirements on particular instances. Global information such as maximal running time on an input of size n cannot be determined by experiment. Two abstractions are generally used: worst case and average case behavior.

Worst case behavior is the maximal running time on any input of a particular size. We use $T_A(n)$ to denote the worst case running time (or simply running time) of algorithm A on an input of size n, i.e.,

$$T_A(n) = \sup\{T_A(p) \ ; \ p \in P \text{ and } g(p) = n\}.$$

Worst case behavior considers algorithms from a pessimistic point of view. For every n we single out the input with maximal running time.

Sometimes, we are given a probability distribution on the set of problem instances. We can then talk about *average case behavior* (or expected behavior); it is defined as the expectation of the running time for problems of a particular size, i.e.,

$$T_A^{av}(n) = E\big(\{T_A(p) \ ; \ p \in P \text{ and } g(p) = n\}\big).$$

In this book, computing expectations always amounts to computing finite sums. Of course, average case running time is never larger than worst case running time and sometimes much smaller. However, an average case analysis always poses the following question: does the actual use of the algorithm conform to the probability distribution on which our analysis is based?

We can now formulate one goal of this book. Determine $T_A(n)$ for important algorithms A. More generally, develop methods for determining $T_A(n)$. Unfortunately, this goal is beyond our reach for many algorithms at the moment. We have to confine ourselves to determine upper and lower bounds for $T_A(n)$, i.e., to asymptotic analysis. A typical claim will be: $T(n)$ is bounded above by some quadratic function. We write $T(n) = O(n^2)$ which means that $T(n) \le c \cdot n^2$ for constants $c > 0$ and n_0 and all $n \ge n_0$. Or we claim that $T(n)$ grows at least as fast as $n \log n$. We write $T(n) = \Omega(n \log n)$ which means that there are constants $c > 0$ and n_0 such that $T(n) \ge c \cdot n \log n$ for all $n \ge n_0$ (log denotes log to base two throughout this book). We come back to this notation in Section **??**.

We can also compare two algorithms A_1 and A_2 for the same problem. We say that A_1 is *faster* than A_2 if $T_{A_1}(n) \le T_{A_2}(n)$ for all n and that A_1 is *asymptotically faster* than A_2 if $\lim_{n\to\infty} T_{A_1}(n)/T_{A_2}(n) = 0$. Of course, if A_1 is asymptotically faster than A_2 then A_2 may still be more efficient than A_1 on instances of small size. This trivial observation is worth being exemplified.

Let us assume that we have 4 algorithms A, B, C, D for solving problem P with running times $T_A(n) = 1000n$, $T_B(n) = 200n \log n$, $T_C(n) = 10n^2$ and $T_D(n) = 2^n$ milliseconds. Then D is fastest for $1 \le n \le 9$, C is fastest for $10 \le n \le 100$ and A is fastest for $n \ge 101$. Algorithm B is never the most efficient. How large is the maximal problem instance which we can solve in one hour of computing time? The answer is 3600 (1600, 600, 21) for algorithm A (B, C, D). If the maximal solvable problem size is too small we can do either one of two things. Buy a larger machine or switch to a more efficient algorithm. Assume first that we buy a machine which is ten times as fast as the present one, or alternatively that we are willing to spend 10 hours of computing time. Then the maximal solvable problem size increases to 36000 (13500, 1900, 25) for algorithms A (B, C, D). We infer from this example that buying a faster machine hardly helps if we use a very inefficient algorithm (algorithm D) and that switching to a faster algorithm has a more drastic effect on the maximally solvable problem size. More generally, we infer from this example that asymptotic analysis is a useful concept and that special considerations are required for small instances (cf. Sections **??** and **??**).

So far, we discussed the complexity of algorithms, sometimes, we will also talk about the *complexity of problems*. An upper bound on the complexity of a problem is established by devising and analyzing an algorithm; i.e., a problem P has complexity $O(n^2)$ if there is an algorithm for P whose running time is bounded by a quadratic function. Lower bounds are more difficult to obtain. A problem P has complexity $\Omega(n^2)$ if *every* algorithm for P has running time at least $\Omega(n^2)$. Lower bound proofs require the discussion of an entire class of algorithms and are usually very difficult to obtain. Lower bounds are only available in very rare circumstances (cf. Sections **??**).

We will next define running time and storage space in precise terms. To do so we have to introduce a machine model. We want this machine model to abstract the most important features of existing computers, so as to make our analysis meaningful for every-day computing, and to make it simple enough, to make analysis possible.

1.1 A Machine Model

Before we can start to analyze the efficiency of algorithms, we need a model of the machine which is supposed to execute the implementation of the algorithms. Our default model is the *RAM* (random access machine), sometimes also called the von Neumann machine [Neu45]. The most important features of this model are that it is *sequential*, i.e., there is a single processing unit, and that it has a *uniform* memory. The memory consists of cells $S[0]$, $S[1]$, $S[2]$, $\ldots$. Each cell can hold a bit string of length w, the *word length* of the machine. Most current architectures have $w = 32$ or $w = 64$. Depending on the context, the content of a memory cell can be interpreted as an integer in the range $[0 \mathinner{..} 2^w - 1]$, or a signed integer, or a floating point number, or a sequence of characters, or $\ldots$. In addition to the main memory there is a small number of *registers* $R_1, \ldots,$ R_k.

The instruction repertoire of our RAM consists of the following types of operations.

- A *memory access* is either a *load* $R_i \leftarrow S[R_j]$ which transfers the contents of the memory cell with index R_j into a register or a *store* $S[R_j] \leftarrow R_i$ with the opposite direction of data movement.

- A register operation $R_i \leftarrow R_j \odot R_\ell$. Arithmetic operations are the usual arithmetic operations like `+`, `-`, `*`, `/`, but also the bit-wise operations `&`, `|`, `^`, `>>`, `<<`, logical operation `||`, `&&`, $\leq$, $<$, $>$, $\geq$. Also, unary operations like `!`, `~` or `-` and loading constants are supported. We may also assume that there are operations which interpret the bits stored in a register as floating point numbers, i.e., finite precision approximations of real numbers.

- A branch instruction like `JZ` j, R_i continues execution at instruction number j if register i is zero.

We will add additional instructions when needed. The traditional von Neumann model assumes that instructions are stored in the memory cells implying that code can modify itself. This "feature" has so many pitfalls however, that we assume that the instructions are stored in a separate memory which can only be modified by the operating system.

Each instruction takes a certain number of time units to execute. For example, we might define that a memory access or branch takes 2 time steps, integer division takes 7 steps and other register operations take 1 time step. The total execution time is the sum of the execution time of the instructions. For a more detailed abstract machine model we refer our readers to the recent book of Knuth [Knu99].

A RAM with word length w can address at most 2^w memory cells and hence is a finite automaton. It cannot be used to solve arbitrarily large problems. We can only do so, if we make the word length a function of the input size: a problem of size n is executed on a machine with word length $w = w(n)$. We will distinguish between logarithmic word length, our default model, and arbitrary word length.

Logarithmic Word Length: Our default assumption is that the word length is logarithmic in the length of the input, i.e., $w = c \cdot \log n$ for some constant c. Almost all results in this book do not depend on the exact value of c as long as c is reasonably large, say at least 2, and hence we will leave the exact value of c unspecified. A machine with logarithmic word length can address $2^w = n^c$ memory cells and the contents of each cell can be interpreted as an integer in the range $[0 .. n^c - 1]$. We analyze our algorithms with respect to the default word length unless we say explicitly otherwise.

Arbitrary Word Length: Sometimes, it is appropriate to treat word length as an independent parameter, e.g., when one wants to study the effect of the parallelism on the word level. In the arbitrary word length model running times are stated as functions of problem size n and word length w.

It is important to remember that the above RAM model is an abstract model. Do not confuse it with physically existing machines. The major differences are:

- The execution times of instructions cannot simply be summed in reality. Modern microprocessors execute many instruction concurrently using pipelining and instruction parallelism. For a more detailed introduction to processor architecture refer to the textbooks of Hennessey and Patterson [HP96] and Müller and Paul [MP95].

- A physical machine has finite memory. Therefore, it is an important task of algorithm analysis to determine the space complexity of algorithms It will tell us how large a problem can be solved on a particular machine (if time plays no role). Please observe that a machine that can access 10^8 distinct memory cells in a second (which vintage 2000 processors can) can read a Gigabyte of memory in a few seconds. Thus linear time and space algorithms can run out of space within a few seconds and hence space may be more constraining than time.

- Memory is not really uniform. Modern computers have a hierarchy of several memory levels. With each new level both capacity and access delay increase. We explain this issue in more detail in Section 1.1.3.

Despite the differences between our abstract model and physically existing machines, the analysis of algorithms on our abstract model tells us a lot (but not everything) about their behavior on real machines.

1.1.1 Higher-Level Programming Languages

We will never specify a program on the level of machine instructions. Algorithm design would have never reached its current level of sophistication, if algorithms had to specified at the machine level. We will use higher level imperative programming languages like C++ or Java or pseudo-code. We will not use functional (ML, Haskill), or constraint (XXX, Oz), or logic programming (Prolog) because their mapping to the machine level is usually too indirect for complexity analysis. It is a major open problem to develop a satisfactory theory of efficiency for the non-imperative programming paradigms.

What do we loose or gain by using higher-level programming languages?

We certainly gain in convenience.

Imperative programming languages are mapped to the machine level by compilers. This mapping is fairly direct. Expression evaluation and assignment map into a sequence of loads, stores, and arithmetic operations, loop control maps into a small number of loads, stores, arithmetic operations, and branch instructions, and the same is true for other higher level programming constructs like procedure calls or method invocation. We do not expect our readers to be compiler experts, but you should have some feeling of how important higher-level programming constructs, like expressions, looks, and procedure calls are implemented on the machine level.

Compilers introduce additional fuzziness into algorithm analysis. We may know in principle, how a particular high level construct is treated by a typical compiler, but we do not know, how the construct is handled by a specific compiler without tediously inspecting the generated assembler code. This adds to the fuzziness of execution times of single instructions due to pipelining, instruction parallelism and memory access times. The best we can hope for is therefore that theoretical analysis predicts running time up to constant factors. This is another rationale for asymptotic analysis.

There is also a positive side of the coin. The fuzziness simplifies the task of analyzing an algorithm. It suffices to count "representative operations", like the elementary arithmetic operations in Section **??** or comparison operations in sorting algorithms in Section **??**, as long as the count $c(n)$ of these operations has the property that the total number of operations executed is of the order $O(c(n))$.

What we said about higher level programming languages also applies to pseudo-code. As long as it is clear, how a construct is to be realized on the machine level, we may use it in the formulation of our programs.

1.1.2 Memory Management, A First Glimpse

Higher-level programming languages divide memory into two parts, called the *stack* and the *heap*[1], respectively, see Figure **??**. All structures whose size and number is known at compile time are stored on the stack and all structures where either the size or their number becomes only known during execution are stored on the heap.

The stack memory is managed automatically. The stack memory is divided into stack frames (also called activation records), one for main program and one for each active invocation of a procedure or block. When a procedure or block

[1] Some authors prefer the name *dynamic memory* or *free store* for the heap

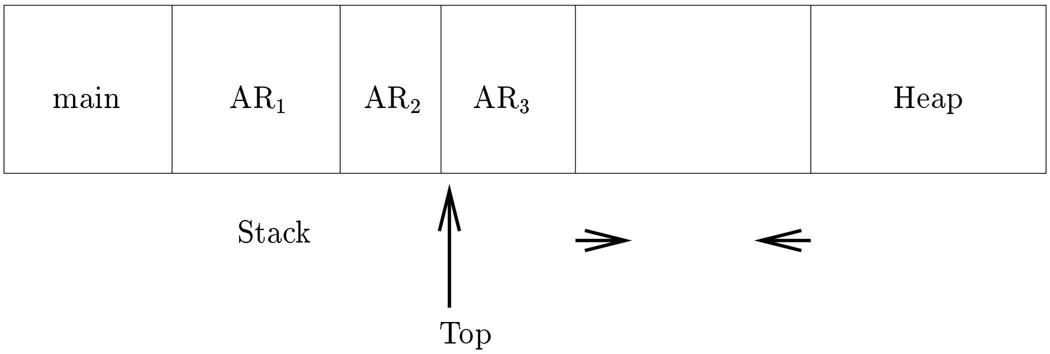

Figure 1.1: The memory is divided into a stack (left side) and a heap (right side). The two parts grow towards each other. The stack consists of a sequence of stack frames. The register *top* contains the base address of the topmost (= rightmost) stack frame.

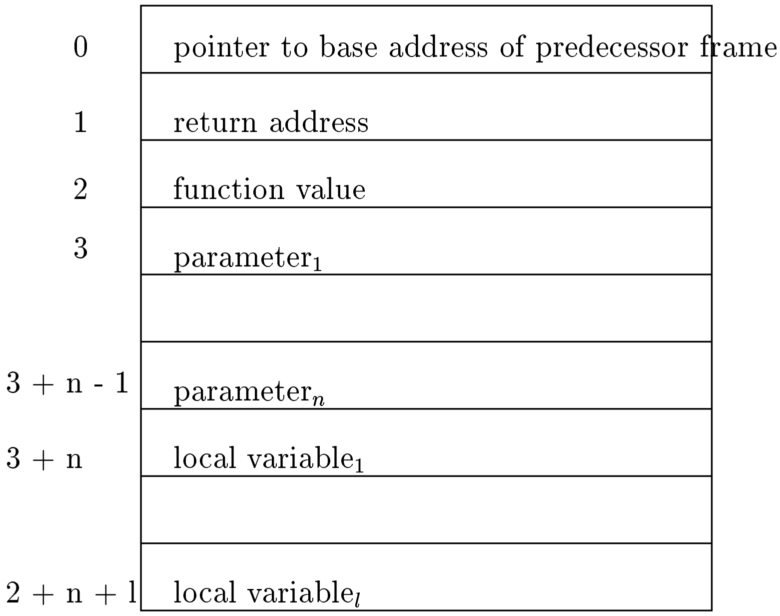

Figure 1.2: A stack frame for a procedure with p parameters and l local variables consists of $p+l+3$ cells. The first cell stores the bases address of the predecessor frame, the second cell stores the return address for the procedure invocation, the third address stores the return value of the procedure, the next p cells store the parameters, and the last l cells are for the local variables.

---

is entered a new stack frame is allocated; the stack frame is freed again upon exit from the procedure. Since active procedure calls are nested, it is always the most recently allocated frame that is deleted (hence the word stack memory).

Figure **??** shows the details of a stack frame. A stack frame for a procedure with n parameters and l local variables consists of $n+l+3$ cells. This number is known to the compiler. We assume in our presentation that all global variables are declared in the main program. An additional entry in a stack frame is needed otherwise. We refer the reader to any textbook on compilation ([**?**]).

What code needs to be generated for procedure calls and procedure return? For a function call $f(x_1, \ldots, x_n)$ within a procedure p the following code is generated. We use s for the size of the activation records for p.

```
(1) store the parameters x1, ...,xn in the new stack frame, i.e.,
    at positions top + s + 3, ..., top + s + 3 + n-1
(2) store return address at position top + s + 2
(3) store current value of top at position top + s
(4) change value of top to top + s
(5) branch to body of f
```

We come to the code for a function return. The terminating procedure simply jumps to the return address.

```
branch to return address (stored at position top + 1)
```

The calling procedure deletes the stack frame of the called procedure and copies the return value into its local memory.

```
(1) delete stack frame, i.e., change top to the value stored at top
(2) copy the return value (stored at position top + s + 2) into the
    local memory
```

We next give an example to make the discussion more concrete. Consider the following procedure for computing Fibonacci numbers. We admit that the procedure is inefficient, but our current goal is to demonstrate the connection between higher level programming languages and RAMs.

```
int Fib(int n) // returns the n-th Fibonacci number
{ if ( n <= 1) return 1;
  int a = Fib(n - 1);
  int b = Fib(n - 2);
  return a + b;
}
```

The procedure has one parameter and two local variables and hence its stack frames consists of 6 cells. The following piece of RAM-code realizes this procedure. In this code we have taken the liberty of extending the instruction repertoire of our RAM, e.g., instead of XXX we would have to write XXX.

```
Fib:  if ( M[top + 3] > 1 ) goto Fib1;    // n <= 1
      M[top + 2] = 1;                     // return value = 1
      goto M[top + 1];                    // branch to return address
Fib1: M[top + 6 + 3] = M[top + 3] - 1;    // store n - 1 at position
                                          // of first parameter
      M[top + 6 + 1] = address of label Fib2;
      M[top + 6]     = top;               // store predecessor
      top            = top + 6;           // push stack frame
      goto Fib;
Fib2: top            = M[top + 6];        // pop stack frame
      M[top + 4]     = M[top + 6 + 2];    // a = Fib(n - 1)
      M[top + 6 + 3] = M[top + 3] - 2;    // store n - 2 at position
```

```
                                              // of first parameter
      M[top + 6 + 1] = address of label Fib3;
      M[top + 6]     = top;                   // store predecessor
      top            = top + 6;               // push stack frame
      goto Fib;
Fib3: top            = M[top + 6];            // pop stack frame
      M[top + 5]     = M[top + 6 + 2];        // b = Fib(n - 1)
      M[top + 2]     = M[top + 4] + M[top + 5];  // return value = a + b
      goto M[top + 1];
```

This completes our discussion of stack memory.

Dynamic memory must be handled by the programmer. Dynamic memory is allocated by a call of the `new` operator and freed again by the call of the `delete` operator. More precisely,

```
T* p = new T;
T* q = new T[n];
```

allocate room for an object (for n objects, respectively) of type T on the heap and returns a pointer to the object (the zeroth object in the array).. We store the pointer in a pointer variable. The space is returned to the dynamic memory by

```
delete   p;
delete[] q;
```

respectively. Memory management requires fairly sophisticated algorithms. They will be the subject of Chapter **??**. For the moment, we are content to introduce the name *memory allocation time* for the cost of a call to `new`. More precisely, we will use $T_{alloc}(n)$ for the cost of allocating a chunk of n words on the heap. We assume that every call to `new` already pays for the corresponding call to `delete`; i.e., invocations of `delete` are free.

Lemma 1 *There is a memory management scheme with* $T_{alloc}(n) = O(1)$.

Proof: See Chapter **??**. ∎

1.1.3 Memory Hierarchy

The memory subsystem of physical machines is quite complicated. There are usually several levels of cache memory, there is main memory, there are disks, and there are sophisticated address translation mechanism. In order to cope with this overwhelming details, we again resort to an abstract model.

The *external memory* machine of Vitter and Shriver [VS94] consists of a RAM with a limited number of *internal* memory cells $S[0], \ldots, S[M-1]$ plus a potentially unlimited supply of *external memory blocks* $E[0], E[1], \ldots$. Each block can store the content of B internal memory cells. An *external operation* $S[i], \ldots, S[i+B-1] \leftarrow E[j]$ or $E[j] \leftarrow S[i], \ldots, S[i+B-1]$ transfers an entire block between internal and external memory.

Why are we restricting ourselves to this special way of data transfer? The main reason is that transferring a single internal memory cell would not be much faster than transferring an entire block (we choose B so that this is the case). For example, a modern hard disk needs about 5ms to bring its head into a position before any data can be read. Within another 5ms it can then transfer about 64 Kbytes of data. So it makes sense to set the block size to this value yielding 10ms for an external operation. If we need less data, the additionally transferred data costs us at most a factor of two in the execution time. If we need more data we use several block transfers.

The external memory model is not only useful for modeling data exchange with hard disks. It can also be used to model the relation between a data cache in the processor and the main memory. The model fits even better here since the hardware actually enforces that data is exchanged in fixed size *cache blocks*. A problem is that we do not have direct control over which blocks are transferred. This complicates algorithm analysis considerably.

We will discuss the role of the memory hierarchy in algorithm design in Chapter **??**.

Exercises:

1. What word length did we assume for the multiplication algorithms in the introduction? (Hint: You *can* get that wrong.)

2. Name at least two ways which make registers different from memory cells.

3. Other definitions of the RAM model assume that there is only one register or none at all. Think for 25 second how this influences the instruction set of the machine and why we choose to use a somewhat more complex model.

4. Is it possible to work with a single register in our model? Why? How about two registers?

5. Find out at least two reasons why self-modifying code is undesirable. Explain why it can make sense to have self-generated code. What is the difference?

6. Explain why it can make sense to count the number of operations of a certain type $c(n)$ more exactly than just asymptotically even if we do not know how exactly this count translates into execution times.

7. Consider the following program. In this program we assume that *int*s can store arbitrarily large integers.

```
int pow1(int n)
{ if ( n == 0 ) return 1;
  else return 2 * pow1(n - 1);
}
int pow2(int n)
{ if ( n == 0 ) return 1;
  else return pow2(n - 1) + pow2(n - 1);
}
main(){
int n; cin >> n;   // reads n from standard input
```

```
cout << pow1(n);    // writes pow1(n) to standard output
cout << pow2(n);
}
```

- Determine the running times of *pow1* and *pow2* as functions of n. It suffices to determine the number of arithmetic operations (additions and subtractions).
- Translate the program above into the machine language of our RAM. Invent instructions for input and output.

1.2 Order of Growth

In most cases we will not be able to derive exact expressions for the running time of algorithms; rather we have to be content with order of growth analysis. We use the following notation.

Definition 1 *Let $f : \mathbb{N}_0 \mapsto \mathbb{N}_0$ be a function. Then $O(f)$, $\Omega(f)$ and $\Theta(f)$ denote the following sets of functions:*

$$
\begin{aligned}
O(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0 \ ; \exists c > 0 \ \exists n_0 : \ g(n) \le c \cdot f(n) \textit{ for all } n \ge n_0\} \\
\Omega(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0 \ ; \exists c > 0 \ \exists n_0 : \ g(n) \ge c \cdot f(n) \textit{ for all } n \ge n_0\} \\
\Theta(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0 \ ; \exists c > 0 \ \exists n_0 : \ (1/c) \cdot f(n) \le g(n) \le c \cdot f(n) \textit{ for all } n \ge n_0\} \\
o(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0 \ ; \forall c > 0 \ \exists n_0 : \ g(n) \le c \cdot f(n) \textit{ for all } n \ge n_0\}
\end{aligned}
$$

We use the O-notation in proofs of upper bounds, the Ω-notation in proofs of lower bounds and the Θ-notation, whenever we can determine the order of growth exactly.

It is customary to use the notations above together with the equality sign instead of the symbols $\in$, $\subseteq$ for the relations 'element of' and 'subset of', i.e., we write

$$n^2 + 5n = n^2 + O(n) = O(n^2) \quad \text{instead of} \quad n^2 + 5n \in n^2 + O(n) \subseteq O(n^2) \ .$$

More precisely, if $\circ$ is an n-ary operation on functions and $A_1, \ldots, A_n$ are sets of functions, then $\circ(A_1, \ldots, A_n)$ denotes the natural extension to sets of functions, e.g., $A_1 + A_2 = \{a_1 + a_2 \ ; a_1 \in A_1 \text{ and } a_2 \in A_2\}$. Singleton sets are denoted by their single member. Then expressions α and β which contain O-expressions denote sets of functions and $\alpha = \beta$ stands for $\alpha \subseteq \beta$. Equalities containing O-expressions can *only be read from left to right.* The terms of sequence $A_1 = A_2 = A_3 = \cdots = A_k$ represent larger and larger sets of functions; the bounds become coarser and coarser from left to right.

Exercises:

1. Is $(n+4)^2 = O(n^2)$? Is $O(n \log n) = O(n^2)$?
2. Let a and b be arbitrary real constants with $b > 0$. Show $(n+a)^b = \Theta(n^b)$.

3. Show $2^{n+1} = O(2^n)$. Is $4^n = O(2^n)$? Is $2^{2n} = O(2^n)$?

4. Which of the following statements are true?
   - $f(n) + g(n) = O(\max(f(n), g(n)))$.
   - $f(n) + g(n) = O(\min(f(n), g(n)))$.
   - If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$.
   - If $f(n) = O(g(n))$ then $\log f(n) = \log g(n)$.
   - $f(n) + o(f(n)) = O(f(n))$.
   - $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.

1.3 Simple Sums and Products

1.4 Recurrence Relations

We collect some facts about recurrence relations.

Theorem 1 *Let a and b be positive integers with $b \geq 2$ and let g be a function defined on the positive reals. Let $T(x)$ be defined by*

$$T(x) = \begin{cases} g(1) & \text{if } x = 1, \\ a \cdot T(x/b) + g(x) & \text{if } x > 1. \end{cases}$$

Then

$$T(n) = \sum_{j=0}^{k} a^j g(\frac{n}{b^j}) \quad \text{for } n = b^k.$$

Proof: We use induction on n. Let $n = b^k$. For $n = 1 = b^0$ we have

$$T(n) = g(1) = \sum_{j=0}^{0} a^j g(\frac{n}{b^j}) \ .$$

So assume $n > 1$. Then $n/b = b^{k-1}$ and hence

$$T(n/b) = \sum_{j=0}^{k-1} a^j g(\frac{n/b}{b^j}) \ .$$

Substituting this expression into the recurrence yields

$$\begin{aligned} T(n) &= a \cdot T(n/b) + g(n) \\ &= a \cdot \sum_{j=0}^{k-1} a^j g(\frac{n/b}{b^j}) + g(n) \\ &= \sum_{j=0}^{k-1} a^{j+1} g(\frac{n}{b^{j+1}}) + g(n) \\ &= \sum_{j=1}^{k} a^j g(\frac{n}{b^j}) + a^0 g(n/b^0) \\ &= \sum_{j=0}^{k} a^j g(\frac{n}{b^j}) \ . \end{aligned}$$

∎

In the next theorem we specialize the theorem above for specific functions g. In this form the theorem is usually called the master theorem for recurrences; it covers most recurrences that appear in this book.

Theorem 2 (Master Theorem for Recurrences) *Let a and b be positive integers with $b \geq 2$, let α and β be non-negative reals, and let $g(x) = O(x^\alpha \log_b^\beta x)$ be a function defined on the positive reals. Let $T(x)$ be defined by*

$$T(x) = \begin{cases} g(1) & \text{if } x = 1, \\ a \cdot T(x/b) + g(x) & \text{if } x > 1. \end{cases}$$

Then

$$T(n) = \begin{cases} O(n^{\log_b a} \log_b^\beta n) & \text{if } a > b^\alpha, \\ O(n^\alpha \log_b^{\beta+1} n) & \text{if } a = b^\alpha, \\ O(n^\alpha \log_b^\beta n) & \text{if } a < b^\alpha \text{ and } \alpha > 0 \text{ or } \beta \geq 1, \\ O(\log_b n) & \text{if } a < 1 \text{ and } \alpha = 0 \text{ and } \beta < 1. \end{cases}$$

for n a power of b.

Proof: Let $c \in I\!R$ be such that $g(x) \leq x(1 + x^\alpha \log_b^\beta x)$ for all x. Then (by Theorem 1)

$$T(n) = \sum_{j=0}^{k} a^j g(\frac{n}{b^j}) = c(k + 1 + n^\alpha \sum_{j=0}^{k} \frac{a^j}{b^{j\alpha}} \log_b \frac{n}{b^j}) \ .$$

Observe next that $n/b^j \leq n$ and hence $\log_b(n/b^j) \leq \log_b n$ for all j. Thus

$$T(n) \leq c(k + 1 + n^\alpha \log_b^\beta n \sum_{j=0}^{k} (\frac{a}{b^\alpha})^j) \ .$$

We now distinguish cases according to the value of a/b^α.

If $a > b^\alpha$,

$$\sum_{j=0}^{k} (\frac{a}{b^\alpha})^j = O((\frac{a}{b^\alpha})^{k+1}) = O((\frac{a}{b^\alpha})^k) = O(\frac{n^{\log_b a}}{n^\alpha}) \ ,$$

since $b^k = n$ and $a^k = b^{k \log_b a} = n^{\log_b a}$. Hence

$$T(n) \leq c(k + 1 + O(n^{\log_b a} \log_b^\beta n)) = O(n^{\log_b a} \log_b^\beta n) \ ,$$

since $k = \log_b n = O(n^{\log_b a})$. Observe that $\log_b a > \alpha \geq 0$.

If $a = b^\alpha$, $\sum_{j=0}^{k} (a/b^\alpha)^j = k + 1$ and hence

$$T(n) \leq c(k + 1 + (k+1) n^{\log_b a} \log_b^\beta n) = O(n^{\log_b a} \log_b^{\beta+1} n) \ ,$$

since $\beta \geq 0$.

If $a < b^\alpha$, $\sum_{j=0}^{k}(a/b^\alpha)^j = O(1)$, since the series is a converging and hence

$$T(n) \leq c(k + 1 + O(n^{\log_b a} \log_b^\beta n)) \ .$$

This is $O(n^{\log_b a} \log_b^\beta n)$ if either $a > 0$ or $\beta \geq 1$ and is $O(k) = O(\log_b n)$ otherwise. ∎

Theorem 3 *Let f, g, and r be function defined for integer arguments, let n_0 be an integer, and let U be defined by*

$$U(n) = \begin{cases} g(n) & \text{if } n \leq n_0, \\ f(n) \cdot U(r(n)) + g(n) & \text{if } n > n_0. \end{cases}$$

For an integer n let $k(n)$ be the minimal k such that[2] *$r^{(k)}(n) \leq n_0$. Then*[3]

$$U(n) = \sum_{j=0}^{k(n)} g(r^{(j)}(n)) \prod_{i=0}^{j-1} f(r^{(i)}(n)) \ .$$

Proof: We use induction on n. For $n \leq n_0$ we have $k(n) = 0$ and hence $U(n) = g(n) = \sum j = 0^0 g(r^{(j)}(n)) \prod_{i=1}^{j} f(r^{(i)}(n))$

So assume $n > n_0$. Then $k(r(n)) = k(n) - 1$ and

$$\begin{aligned}
U(n) &= f(n) \cdot U(r(n)) + g(n) \\
&= g(n) + f(n) \cdot \sum_{j=0}^{k(r(n))} g(r^{(j)}(r(n))) \prod_{i=0}^{j-1} f(r^{(i)}(r(n))) \\
&= g(n) + f(n) \cdot \sum_{j=0}^{k(r(n))} g(r^{(j+1)}(n)) \prod_{i=0}^{j-1} f(r^{(i+1)}(n)) \\
&= g(n) + \sum_{j=0}^{k(n)-1} g(r^{(j+1)}(n)) f(n) \cdot \prod_{i=1}^{j} f(r^{(i)}(n)) \\
&= g(n) + \sum_{j=0}^{k(n)-1} g(r^{(j+1)}(n)) \prod_{i=0}^{j} f(r^{(i)}(n)) \\
&= g(n) + \sum_{j=1}^{k(n)} g(r^{(j)}(n)) \prod_{i=0}^{j-1} f(r^{(i)}(n)) \\
&= \sum_{j=0}^{k(n)} g(r^{(j)}(n)) \prod_{i=0}^{j-1} f(r^{(i)}(n)) \ ,
\end{aligned}$$

where the first equality is the definition of $U(n)$, the second equality follows from the induction hypothesis applied to $r(n)$, the third equality follows from

[2] The iterates $r^{(k)}$ of a function r are defined by $r^{(0)}(n) = n$ for all n and $r^{(k+1)}(n) = r(r^{(k)}(n))$.

[3] A product $\prod_{i=0}^{-1}$ is by definition equal to one.

$r^{(k)}(r(n)) = r^{(k+1)}$ and $k(r(n)) = k(n) - 1$, the fourth equality follows from writing i instead of $i+1$ in the product, the fifth equality follows from pushing $f(n)$ into the product, the sixth equality follows from replacing $j+1$ by j, and the final equality follows from pulling $g(n)$ into the sum. ∎

Hannah's theorem

Bibliography

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann, 1996.

[Knu99] Donald E. Knuth. *MMIXware: A RISC Computer for the Third Millennium.* Number 1750 in LNCS. Springer, 1999.

[MP95] S. Müller and W. Paul. *The Complexity of Simple Computer Architectures.* LNCS Tutorial. Springer Verlag, 1995.

[Neu45] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[VS94] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.