LN-TT-22012-2

Tommi Tervonen
Econometric Institute
Erasmus University Rotterdam

August 2012

> "Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. " (D.E. Knuth)

1 Introduction

I have made these lecture notes to help you better understand theoretical contents of the course FEB22012, "Programmeren". The course is divided in two parts: the lectures in which we discuss topics in imperative programming, and the practical sessions in which you can get help from the student assistants for completing the home assignments. Although the course contains quite some contact teaching hours, the largest part of the course workload is individual study - you should try to do the exercises at home (except for the first ones) _before_ coming to the exercise session. Programming is a too complex discipline to be learned in a few hours in class rooms, and requires countless hours of lone suffering followed by brief moments of enlightenment.

This course is called "Programmeren", and we will indeed cover multiple topics within the scope of imperative programming paradigm that is discussed in Section 1.2. The main learning objective of the course is to understand how to develop correct and efficient programs irrespective of the implementation language. The exercises will be made in Matlab, but after this course you should be able to program also in other weakly typed languages such as R or Python.

1.1 Recall of inl. progr. course contents

A prerequisite for succesfully completing this course is understanding the material discussed in FEB21011, "Inleiding Programmeren". If you have hard time completing the assigments of this course, I recommend trying to re-read the material of FEB21011. You do not need to remember the short introduction to object oriented programming, but rest of the course yes. More specifically, you should be familiar with the following topics:

- Variables and methods
- Program flow

- Control structures
- Decisions and branching
- Binary operators
- Arithmetic operators
- Scoping

1.2 Programming paradigms

Programming paradigms refer to the philosophy behind designing a certain programming language. When you learn to program according to a certain paradigm, learning new languages of the same paradigm is easy - you only have to learn the syntax, as the manner in which problems are solved is generally the same within a single paradigm. There are three main programming paradigms (exemplary languages in parentheses):

1. Procedural / imperative paradigm (C, Pascal, Matlab, R, Fortran, Algol)
2. Object-oriented paradigm (Java, Smalltalk, C++ partially)
3. Declarative paradigm, including
   (a) Functional programming (ML, Lisp, Haskell, Erlang, Scala, Scheme)
   (b) Logic programming (Prolog)

Java which you learned in the first course falls under *object-oriented paradigm* (OOP). In OOP, the main idea is to construct classes that are instantiated to objects that can communicate (i.e. send signals, call methods) with each other. The emphasis in OOP is on creating new classes implementing behaviour required for solving the problem. One of the main advantages of OOP is that the classes can have their implementation details hidden (as `private` or `protected`). This allows to control the immense complexities inherent in programming through data hiding, which is an invaluable mechanism especially when more than one programmer are working on the same problem. For example, consider the following class:

```
public class Car {
        private char[] regNr;
        public Car(String regNr) {
                this.regNr = regNr.toCharArray();
        }
        public String getRegNr() {
                return new String(regNr);
        }
}
```

The designer of this class has decided to store the registration number as a character array, but users (other programmers) of the class see the registration number as a `String`. If another way of storing registration numbers internally becomes necessary (e.g. within a

data base), the use of the class (calling its methods) does not have to be changed and the code still keeps on working as it is.

Although is it mostly OOP, Java also includes non-object-oriented parts in the language as, for example, primitive types are not objects. These are included on one hand for the code to execute faster, and on the other hand due to historical reasons. Java is partially based on C and its object oriented extension C++, making it also adhering to the *procedural paradigm*. Procedural programming is the oldest still widely used way of creating programs, and it is also called imperative paradigm, although imperative is used often as a wider term including both procedural- and object oriented ones. In procedural programming the two main ideas are (1) to create statements that change the program states, such as

```
int x = 3; // assignment
while (x < 5) { ... } // iteration
```

and (2) to create methods that group statements in a way that minimize the scope in which the state changes apply. Imperative paradigm consists solely of (1), whereas in object-oriented paradigm (2) is replaced by focusing on objects that include encapsulation of data.

The problem with imperative programming is that the correctness of programs is in many cases extremely difficult to assess. We will come back to this in Section 4. Declarative programming languages make it feasible to prove the program correctness. Functional languages are in addition easily parallelizable, which has lead to their increased use in the past few years due to increased availability and importance of multi-core and distributed systems. Declarative languages are, however, out of our scope in this course.

1.3 Scripting languages

In addition to distinguishing between paradigms, programming languages can be classified into compiled and interpreted languages. With compiled languages the source files are compiled into binary code which can then be directly executed. With interpreted languages there is no compilation performed, but the source code itself is executed in a special environment. There is a trade-off between these two approaches: on one hand, compiled code is usually considerably faster and does not require extra software to run, making program distribution easier. On the other hand, interpreted code does not have to be compiled which makes the development faster, and it can even be input into an interactive environment one line at a time.

Java falls somewhere in between the two approaches. Its source code is compiled into byte code (not machine code) that then needs to be executed in a virtual machine. In this way Java achieves the speed ($\pm 95\%$ of equivalent fully compiled code) of compiled languages and independence of the compilation platform as there are separate machine code virtual machines for each platform.

Scripting languages are often used for less complicated programming tasks, or for those that are small enough to be implemented in a few screens of code. Matlab is a scripting language aimed mainly at scientific computing – as such, it is reasonably well suited for programming econometrical models and for running small-scale scientific experiments. A rule of thumb for using a "real" programming language such as Java instead of Matlab is

that scripting languages should not be used for making programs that will ever be used by more than a few other people.

1.4 Introduction to types

Typing systems form the core of programming languages. In the actual hardware, the computers process only sequences of bits. For example, a number type `int` is a chosen quantity (e.g. 32) of bits that has a meaning to represent a number in a certain form of binary encoding. Further abstraction provides more complex types. For example, Java's `String` represents character strings in UTF-8 encoding. The whole idea of programming in higher level languages (such as C, Java, or Matlab) is to construct abstractions and methods that compute with those. This enables a programmer to manage the underlying complexity in a layered manner.

Java is a strongly typed language. This means that every variable has a type that has to be introduced when the variable is declared. For example, the following is a valid variable declaration in Java:

```
int nrVar;
```

Now the `nrVar` can *only* be used to store integers. The typing system enables the compiler to detect errors in the following kinds of assignment statements:

```
nrVar = "my name"; // error - "my name" is of type String
nrVar = 'c'; // error - 'c' is of type char
```

Strong typing increases program safety by detecting some programming errors already at compilation phase. Strongly typed languages are suited for all kind of programs, and practically all software you are using on your computer is programmed with a strongly typed language such as Java, C or C++.

Strong typing is not the only way to implement a type system; also weak typing is possible. In weakly typed languages the variables are declared without a type, and they can hold any type of information. For example, consider the following lines of valid Matlab code:

```
nrVar = 3; % now nrVar is containing integer 3
nrVar = "my name"; % now nrVar refers to String "my name"
```

The problem with weakly typed languages is that there is (in most cases) no compilation time checking of types. So, for example, the following lines of code compile and execute correctly:

```
x = '1';
y = 1;
z = x + y; % now z = ?
```

Matlab is a weakly typed language and when executing the above code in it, `z` would end up having value 50.0: '1' would first be converted to a numerical value (49 in ASCII) and then the addition would be done with double precision numbers. The advantage in weakly typed languages is that they allow easy conversion between types, and you never

have to provide two different versions of the same method for e.g. handling `int` and `double` type input parameters. The disadvantage is, that it is a _lot_ easier to make programming mistakes. When programming in Matlab, you _really_ have to understand what is happening within the computer when the instructions are executed. After this course, you hopefully do.

2 Computing

> "The question of whether Machines Can Think... is about as relevant as the question of whether Submarines Can Swim." (E.W. Dijkstra)

The term computing refers to the abstract primitive operations that can be performed with computers. With respect to computing, Matlab and Java are equivalent and they can be used interexchangeably to solve the same problem. To understand what I mean by this, consider the traditional pocket calculator: clearly it can be used to *compute*; if you input 1+1, the result is computed with bitwise operations to come up with the output 2. Most pocket calculators also have memory for storing intermediate results with the M+ key. The more advanced ones even allow to plot functions graphically. So is there a difference between a pocket calculator and your desktop PC apart from the speed of computation and the amount of memory available?

Intuitively you might be inclined to answer yes - most pocket calculators differ substantially from desktop PCs. This is also the correct answer in a theoretical sense, with the main difference being that desktop PCs (as most modern devices classifiable as computers) are implementing a *von Neumann architecture* composed of the following three components:

1. memory for storing data processed in blocks of certain number of bits
2. instruction set including operations such as add and multiply, defined as data
3. processor that fetches instructions from the memory, and can read and write the same memory

The main innovation in a von Neumann machine is that is allows for programs stored in the same memory that is accessed by the processor. Although the pocket calculator has memory (1), instruction set (2), and processing capability, it doesn't use the same memory for storing both the *data* and the *program*. By having stored programs, we can use computers to write the programs, to compile them, and to execute them subsequently.

2.1 Numerical representation

Computers work only with binary digits (bits) that form an integer field modulo 2 (i.e. 0's and 1's). Bits are arranged in sequences of certain length that are the most primitive unit of processing for a common programmer. The fastest unit of memory in a standard PC is a register, and their size defines the processor's bit size. Most modern computers use 32 bit processing, although all the newest PC processors are 64 bit. The 32 bits can be used to represent integers within the range $[-2^{31}, 2^{31} - 1]$.

Computers use registers also for processing real numbers. When real numbers are represented with a fixed amount of bits, they can be represented only up to a certain precision.

The way real numbers are represented in modern computers is by having a possibility of the decimal point to change place, which is why they are called "floating point" numbers. The floating point numbers are represented as a pair of signed integer *exponent* e and signed *fraction* f, with a fixed *base* b for representing a number with p *digits* as:

$$(e, f) = f \times b^e$$

For example a floating decimal ($b = 10$) with 8 digits can represent Plank's constant (6.6261×10^{-27}) as

$$\begin{aligned}&(-26, +.66261000)\\&= 0.66261 \times 10^{-26}\end{aligned}$$

To make floating point computation easier, we often represent the numbers in a normalized format so that

$$\begin{aligned}|f| &< 1\\-b^p < b^p f &< b^p\end{aligned}$$

that is, the radix point appears just before the first significant digit of the number (i.e. $(-26, +.66261000)$ and not $(-27, +6.6261000)$ or $(-25, +.06626100)$. The standard way to denote floating point numbers in exponential format in programming languages is to present the fraction followed by 'E' and the exponent, e.g. the Planck's constant would be '0.66261E-26'.

Most computers use the IEEE 754 standard format for representing floating point numbers. It defines the double precision (64 bit) floating binary ($b = 2$) consisting of 1 bit for the sign, 11 bits for exponent, and 52 bits for the fraction. Figure 2.1 presents the bit layout of the format. The 53 bits (52 + 1 for the sign) can represent fractions with approximately 16 base-10 digits ($53 \log_{10} 2 \approx 15.955$). Note that all three parts (sign, exponent, and fraction) of the IEEE 754 double precision floating point number are unsigned integers. The three parts are combined into a single number with:

$$value = (-1)^{sign}(1 + \sum_{i=1}^{52} b_{-i} 2^{-i}) \times 2^{(e-1023)},$$

where e is the exponent, b bits in the fraction (indexed with $-i$, and *sign* the single sign-bit. The $1 + \sum$ part of the value is within $[1, 2)$. The exponents are offset encoded (subtracted 1023) to enable full range bit use. Exponent 0x000 is used to represent zero (with fraction = 0), and 0x7FF infinities (fraction = 0) and NaN (not a number, fraction $\neq 0$.

The internal representation of decimal numbers has large implications for scientific computation. First of all, operations on floating point numbers are neither associative nor distributive, that is,

$$a + (b + c) \neq (a + b) + c, \text{ for many } a, b, c$$

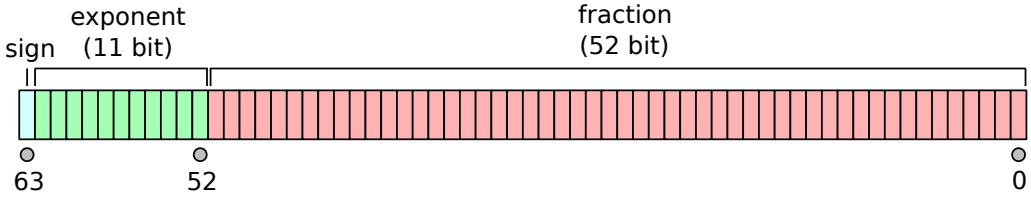

Figure 2.1: Double precision (64 bit) floating binary representation of the IEEE 754 standard. (source: Wikipedia)

$$a * (b + c) \neq (a * b) + (a * c), \text{ for many } a, b, c$$

when *a, b,* and *c* are floating point numbers. For example, consider

$$\begin{aligned} a =& 0.42 \\ b =& -0.5 \\ c =& 0.08 \end{aligned}$$

now, when computed with IEEE 754 double-precision point binaries, we get

$$\begin{aligned} (a + b) + c =& -1.3878 \times 10^{-17} \\ a + (b + c) =& 0 \end{aligned}$$

due to 0.42 being inexactly represented. Second, the limited amount of bits used to store the numbers can cause under- and overflows as results of arithmetic operations can be too small or large to be represented with the limited precision floating point numbers. For example, $1.2345678 + 1.7654321 = 3$ due to the inherent imprecision of the floating point representation. Therefore you should never compare results from operations involving floating point arithmetics with an exact value, but rather see whether they are within some threshold ϵ, e.g. $1.2345678 + 1.7654321 - 3 \leq \epsilon$. The floating point numbers are actually never precise, but represent an interval around the floating point representation. For example, 3.0 represents all the numbers within the interval $[3 - \epsilon, 3 + \epsilon]$.

Third, the accuracy of floating point operations' results depends on which operation is performed. Roughly speaking multiplication is less precise than addition, and repeated application of addition/substraction can result in arbitrarily large errors if incorrect rounding scheme is used (the IEEE 754 forces a correct rounding scheme so you do not have to worry about it). The details of algorithms for floating point arithmetics are out of our scope[1]. Although the standard way of performing floating point computations (also in Matlab) is with double precision numbers, it is also possible to represent numbers with arbitrary high precision as byte arrays. Most programming languages do not include such functionality built in, so if very high precision computations are found necessary an external library must be used (such as the CERN Colt for Java).

[1]For more information, see The Art of Computer Programming, vol. 2 (2nd ed.), sect. 4.2.2.

2.2 Computational complexity

Now that we have some idea about accuracy of results of the numerical operations, the next important question is how long will the different operations take. Addition of two double precision floating point numbers is quite complicated, but luckily most processors include a separate floating point unit (FPU) that makes the processing very fast. Then what about arbitrary precision numbers? To simplify the question, lets consider addition of two arbitrary length integers. Without loss of generality, let us consider them to be represented as bit arrays of the same length n. The sum we want to compute is then of length $n + 1$.

The standard schoolbook addition algorithm provides us with a simple way of adding numbers of an arbitrary base. For example, when $n = 4$, computing 3 (=0011b) + 5 (=0101b) proceeds from right to left by carrying the first overflowing bit to the second least significant bit, etc, and resulting in 8 (=1000b). So for each bit in the input, we have to perform 2 operations: adding together the bits of each operand, and adding to that the carry bit. So for adding together numbers with n bits, we actually have to perform $2n$ operations.

In case of integers of the size processed within the processor (n=32 or 64), the addition operation is extremely fast. But sometimes we need to perform big integer computations, and n can be 100000000 - then the overhead in performing the additions can make a noticeable difference in running times. So even the most elementary operations take time dependent on the *input size* n. In the case of integer addition, when input size grows, the amount of computational effort required by the addition algorithm grows linearly with respect to the input size. However, this is not the case in general with algorithms. For example, consider sorting an array of integers such as:

2	3	1	5	4

There are various algorithms for sorting, and we will come back to them in Section 6. For now, let us consider *insertion sort* that sorts the array by considering every element in the array in turn, and places them in the correct places according to increasing values. This resembles the way people sort their hand when playing card games - putting each card in turn in their place to have the cards arranged according to suite, number, or both. The algorithm for insertion sort is:

```
1  function a = insertionSort(a)
2          for j=2:length(a)
3                  key = a(j);
4                  i = j-1;
5                  while i > 0 && a(i) > key
6                          a(i+1) = a(i);
7                          i = i-1;
8                  end
9                  a(i+1) = key;
10         end
11 end
```

Insertion sort for the previously introduced array proceeds as follows (the currently processed element is in bold):

2	3	1	5	4	starting array
2	**3**	1	5	4	j=2
1	2	3	5	4	j=3
1	2	3	**5**	4	j=4
1	2	3	**4**	5	j=5

Let us now *analyze* the insertion sort to predict the amount of resources (memory and running time) it will require to sort an array of length n. We will assume a single-processor random-access machine (RAM) in which instructions are processed one after another - no parallelism is allowed. For memory, we can easily conclude that the algorithm uses no additional memory dependent on the input size n - only a constant amount of additional temporary variables are needed. Another way to express this is to say that insertion sort does the sorting *in place*.

The running time of algorithms is analyzed by counting the required amount of primitive operations such as assignments and arithemetic operations. The exact CPU cycles required by these operations vary by the implementation language and operating hardware used. Counting the computation *steps* instead of their exact execution times allows us to analyze the algorithms on a more abstract level. The total amount of these steps is the algorithms computational *cost*.

Let us denote by c_i the cost of executing line i, and by t_j the amount of times the while loop test on line 5 is executed. For the previously presented insertion sort algorithm the costs and the number of times each line is executed are:

Line	2	3	4	5	6	7	9
Cost	c_2	c_3	c_4	c_5	c_6	c_7	c_9
Times	n	$n-1$	$n-1$	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n}(t_j-1)$	$\sum_{j=2}^{n}(t_j-1)$	$n-1$

Now by summing the products of the costs and times, we get the total running time of

$$T(n) = c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_9(n-1)$$

Note that with the same input size n we can still get different running times depending on the t_j. For example, if the array is already completely sorted ($a(i) \leq a(j)\ \forall i < j,\ i, j \in \{1, \ldots, n\}$), then $t_j = 1\ \forall j \in \{1, \ldots, n\}$ and the *best-case running time* is

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\ &= (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9) \end{aligned}$$

By replacing the two sums of c_i's with constants $a = c_2 + c_3 + c_4 + c_5 + c_9$ and $b = c_2 + c_4 + c_5 + c_9$ we can express this as

$$T(n) = an + b$$

that is a linear function of n. Now, assume that the array is in an inverse order ($a(i) > a(j)\ \forall i < j,\ i, j \in \{1, \ldots, n\}$). The cost becomes considerably higher as lines 5, 6, and 7

are executed more times. In every iteration of the while loop the current element a(i) must be compared with each of the elements in the already sorted subarray $a(1), \ldots, a(i-1)$, so $t_j = j \; \forall j \in \{2, \ldots, n\}$. Now the *worst-case running time* is

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_9(n-1) \\ &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + (c_6 + c_7)(\frac{n(n-1)}{2}) + c_9(n-1) \\ &= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_2 + c_3 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9)n - (c_3 + c_4 + c_5 + c_9) \end{aligned}$$

that can be expressed by replacing the c_i's with constants a, b, and c, giving a quadratic running time of

$$T(n) = an^2 + bn + c$$

Most of the time when we are analyzing algorithms, we are interested in their worst case performance, because it gives an upper bound on the complexity. This allows us to give some guarantees on how bad the algorithm can perform. Fundamental algorithms are executed even in a single piece of software numerous times repetitively, so the probability that the worst case scenario happens is often high. Also, for some algorithms the input leading to worse case running time occurs often, especially when the input comes from a real process (e.g. new customers to be sorted in decreasing customer numbers).

Calculating the "exact" running time as we did it for analysis of the insertion sort is laborous. Luckily such a detailed analysis is not often required, as we are interested mostly in the order of growth of the complexity. That is, when the size of input n increases, how fast does the computational cost $T(n)$ increase? The table below shows the execution times of different T(n) assuming processing of 10^9 instructions per second. Note that algorithms with a computational cost of 2^n or $n!$ are not solvable in a reasonable time already with very modest problem sizes.

	10	100	1000	1 0000	100000	1000000
n	10^{-8}s	10^{-7}s	10^{-6}s	10^{-5}s	10^{-4}s	10^{-3}s
$n \log n$	10^{-8}s	2.4×10^{-8}s	2.0×10^{-6}s	3.5×10^{-4}s	0.1s	56s
n^2	10^{-7}s	10^{-5}s	10^{-3}s	0.1s	10s	17min
n^3	10^{-6}s	10^{-3}s	1s	17min	12d	32y
2^n	10^{-6}s	4.0×10^{13}y	3.3×10^{284}y			
$n!$	3.6×10^{-3}s	3.0×10^{141}y				

So instead of calculating the amount of times different instructions are executed when determining the complexity of an algorithm, we are usually more interested in its *asymptotic* complexity. That is, given an input size $n > n_0$, where n_0 is some constant value, how fast do the required resources (execution time or memory) grow? Furthermore, to be able to apply techniques of mathematical analysis, we would like to state the algorithms cost as a function of the input size n. The asymptotic behaviour of a function depends only on the highest order term, and not at all on the constants. For this we use the big-O notation: given a function $g(n)$, the set of functions

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 : \ 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

are asymptotically O-equivalent. For example, our previous quadratic complexity $an^2 + bn + c \in O(n^2)$, and also $3n^2 \in O(n^2)$ and $n^2 + n \log n \in O(n^2)$.

With the O-notation our analysis of algorithms and determination of their practical usefulness gets a lot easier. We can easily compare speed of algorithms, and one of $O(n^2)$ time complexity is in practice most of the cases faster than another one with a complexity of $O(n^3)$ given a sufficient input size n. Any algorithm with a polynomial complexity ($O(g(n))$, where $g(n)$ is a polynomial) is called *tractable* - it grows sufficiently slowly to be possibly of practical use. Understanding the O-notation is the foundation for using and designing efficient algorithms, and for comprehending the complexity of problems often encountered in various subfields of econometrics and management science. Unfortunately there does not exist efficient algorithms for many practically relevant problems, and the problems themselves are believed to be of non-polynomial structure[2].

3 Memory organization

> "Computers have lots of memory but no imagination." (Unknown)

In the complexity analysis discussed in the previous section we assumed unlimited memory. In practice the memory is always limited although large. The memory requirements of an algorithm can depend on the input size, and they can be analyzed in a similar way as the running time, stated as a function of the input size with the O-notation (e.g. O(n)).

The way we represent information has implications on how much memory is required. For example, modern computers use 64 bit registers, meaning that they can operate efficiently with 64 bit integers represented as binary numbers. The last bits of number 3 are 0011, and of number 4 they are 0100. We could also consider representing integers by having the index of the bit corresponding to the number to be non-zero and the other bits zero, i.e. 3 would be represented with 0100 and 4 with 1000. This representation, however, is not minimal: most of the bits would stay unused as e.g. 1010 would not represent any number (or alternatively number 6 would have 2 different representations).

It is easy to see that the standard technique for representing positive integers is optimal, although for negative ones this is not the case (they are most often represented as 2's complements). For higher order *data structures* the memory representation can be more complex to understand, and it can make a large difference for the amount of memory required by an algorithm operating on the data structure and affect its execution time. Memory of computers is linear and *addressed* with an integer index indicating a slot in the memory. In high level programming languages such as Matlab or Java the memory location of a variable is invisible to the programmer unless you e.g. print out the object reference variable in Java. The actual memory address is in almost all cases irrelevant for the programmer.

[2]This relates to the largest still unsolved problem in mathematics: if the solution of a problem can be verified by a computer in polynomial time, can it then also be solved in polynomial time (P=NP)? Most scientists believe that the answer is no ($P \neq NP$), but no proof has yet been given. For more discussion, see e.g. `http://en.wikipedia.org/wiki/P_versus_NP_problem`.

Standard PC architecture is composed hierarchically of memories of different speed and size. The arithmetic unit is the core of the processor and can perform very fast operations on the processor *registers* - these are the fastest memory available and often serve a specific purpose (e.g. for adding numbers together or pointing to a certain memory location). There are, however, a very limited amount of registers available (usually between 5 and 50). The next fastest memory is what is commonly called the main memory of the computer - its random access memory. For operating on it, the contents have to be first be loaded into the processors register(s), and after performing the desired operations the result has to be stored back in the main memory. The amount of memory in standard PC's is large (e.g. 3GB) but not unlimited. In case the main memory runs out, the operating system can *swap* infrequently used memory pages into the hard disk, thus expanding the main memory in a way that is very slow to access.

3.1 Matrix representations

Many data structures can be represented as *matrices* and a lot of commonly used algorithms (e.g. simplex) can be formulated to operate on matrices. Consequently Matlab has been designed to be matrix-oriented. In type conversions of Matlab, matrices have priority: if there are two ways to convert a type, the conversion to matrix has priority. For example,

```
a = [3, 4];
b = '1';
c = a*b;
```

leads to c having value [147, 196] as b is first converted into a matrix of integers, and afterwards a matrix multiplication is performed.

Matrices can be represented in different ways. Let us consider first the naive approach: storing matrices as 2-dimensional arrays. In this way, the first dimension contains references to the second dimension, that in turn contains the actual value. This is illustrated in Figure 3.1. Now in order to access the element in location [a, b], we have to do 2 memory lookups: first to fetch the location of the column with a lookup in the first dimension [a,], and then to fetch the actual value by the second dimension [a,b].

We can represent matrices also in a more efficient way. Remember that computer memory is organized in a linear manner: there is essentially only a single dimension. Now let us exploit the fact that matrices are always rectangular: all rows are of the same length, and all the columns as well. This allows us to represent matrices as a one-dimensional array of length a*b, where element at [a, b] can be found at index [(a-1)*n+b]. This is illustrated in Figure 3.2 showing a *row-major* matrix representation: the elements are ordered according to sequences of rows. Another way of flattening a matrix to a linear sequence is to order the elements by columns in a *column-major order*. Then the element [a, b] would be found at index [(b-1)*m+a]. For example, a matrix of

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

would be represented as [1 2 3 4 5 6] in the row-major order, and [1 4 2 5 3 6] in the column-major order. The row-major order is more efficient when the elements are accessed

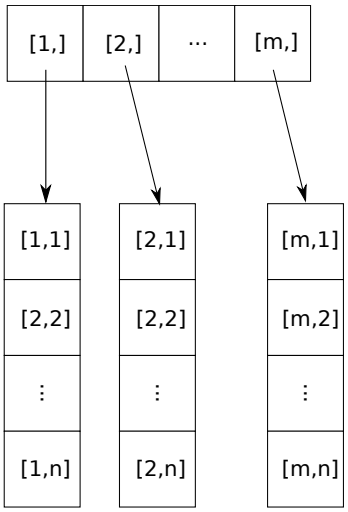

Figure 3.1: Naive representation of an $m \times n$ matrix.

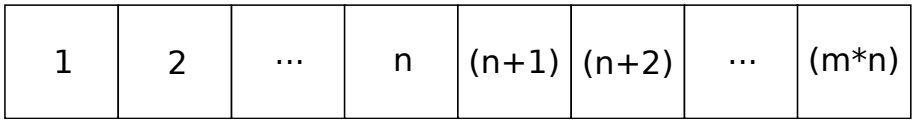

Figure 3.2: Row-major representation of an $m \times n$ matrix.

sequentially. The column-major one enables more efficient matrix multiplication. Treating a row-major array as a column-major array is the same as transposing it. The choice or row- or colum-majority can affect non-asymptotically the running times of algorithms operating on the matrices as elements that are stored and accessed contiguously can be cached. Matlab stores matrices in the column-major order.

The row- and column-major representations are efficient when the matrices are *dense*, that is, they contain a reasonably small amount of zero elements. When matrices are *sparse*, that is, they contain a lot of zeroes, another representation is more efficient. For example, consider the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Representing it directly as a flattened matrix would require 25 integers of memory. Sparse matrices can be represented more efficiently by simply listing the non-zero elements. The previous matrix would then have representation of ([3, 4, 2], [5, 1, 1]) that requires only 6 integers of memory. There are also matrices with special structures that can be represented more effficiently, e.g. the diagonal matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

that can be represented as a vector listing elements on the diagonal [1, 3, 2, 7, 4]. A special case of diagonal matrices is the identity matrix I_n, e.g. I_5:

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

that can be represented with a single integer n.

3.2 Matrix multiplication

Multiplication of matrices ($C = AB$) is one one of the most important algorithms for matrix-oriented languages such as Matlab. We can imitate the schoolbook way of multiplying two matrices to produce the naive multiplication algorithm:

```
function C = multiply(A, B)
        C = zeros(rows(A), columns(B));
        for (i=1:rows(A))
                for (j=1:columns(B))
                        sum = 0;
                        for (k=1:columns(A))
                                sum = sum + A(i, k) * B(k, j);
                        end
                        C(i, j) = sum;
                end
        end
end
```

Now, without a loss of generality, assume that A and B are square and of the same size $n \times n$. Then the running time of `multiply` is $O(n^3)$ - quite a high complexity for an algorithm that is often applied. There have been devised other, more efficient algorithms, of which the most well known one is the Strassen algorithm. Let us assume that n is a power of 2 (if it is not, empty rows and columns can be added), and divide the multiplication into submatrices as

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix}$$

Now, the matrix multiplication breaks down to computing

$$
\begin{aligned}
r &= ae + bf \\
s &= ag + bh \\
t &= ce + df \\
u &= cg + dh
\end{aligned}
$$

of which each one is composed of two multiplications of $n/2 \times n/2$ matrices and addition of their $n/2 \times n/2$ products. The multiplication of $n/2 \times n/2$ matrices can be done in the same manner, leading into a recursive algorithm with a running time of

$$T(n) = 8T(n/2) + O(n^2)$$

This equation is not yet completely solved, but we can write it open as

$$
\begin{aligned}
T(n) &= 8T(n/2) + n^2 \\
&= n^2 + 8((n/2)^2 + 8T(n/4)) \\
&= n^2 + 8((n/2)^2 + 8((n/4)^2 + 8T(n/16))) \\
&= n^2 + 2n^2 + 4n^2 + 8T(n/16)))
\end{aligned}
$$

and as the i^{th} term in this series is $2^{i-1}n^2$,

$$
\begin{aligned}
T(n) &= n^2 + 2n^2 + 4n^2 + \cdots + 2^{\log_2 n} O(1) \\
&= n^2 \sum_{i=0}^{\log_2 n} 2^i + O(n^{\log_2 2}) \\
&= n^2 \frac{2^{\log_2(n+1)} - 1}{2 - 1} + O(n) \\
&\leq n^2 O(2^{\log_2 n}) + O(n) = n^2 O(n) + O(n) \\
&= O(n^3)
\end{aligned}
$$

which is the complexity of the naive matrix multiplication algorithm. But Strassen found out that not all 8 matrix multiplications actually have to be made, as it is possible to compute C with 7 multiplications and 18 additions as

$$
\begin{aligned}
P_1 &= ag - ah \\
P_2 &= ah + bh \\
P_3 &= ce + de \\
P_4 &= df - de \\
P_5 &= ae + ah + de + dh \\
P_6 &= bf + bh - df - dh \\
P_7 &= ae + ag - ce - cg \\
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

leading to an overall computational cost of

$$\begin{aligned} T(n) &= 7T(n/2) + O(n^2) \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81}) \end{aligned}$$

which is asymptotically significantly faster than the naive addition. Strassen's algorithm is, however, slower with small n, and often the matrix multiplication algorithm to be used is chosen on-the-fly based on the dimensionality of the matrices to be multiplied. There are even more efficient matrix multiplication algorithms with complexity $O(n^{2.38})$, but they are extremely complex and faster only with reasonably large values of n.

The Strassen algorithm shows a *divide-and-conquer* approach to problem solving: the original problem is divided into subproblems that can be individually solved. The divided subproblems can then be again divided until a problem that is trivial to solve is encountered (e.g. multiplying matrices of size 2×2). Then the solutions to subproblems are composed one level at a time to form a solution to the original problem. Divide-and-conquer algorithms have been found to be very effective in many problems, such as sorting that I will cover later on.

4 Program correctness

> "There are two ways to write error-free programs; only the third one works." (A. J. Perlis)
>
> "There are two ways to write error-free programs; only one third works." (V. Milea)

Programming by the imperative paradigm can be very efficient as the statements are often directly executable. However, imperative paradigm does have its disadvantages, the most severe of them being *side effects.* For example, consider the following method for counting the sum of integers in an array:

```
public static int countSum(int[] array) {
        for (int i=1;i<array.length;i++) {
                array[i] = array[i] + array[i-1];
        }
        return array[array.length-1];
}
```

It is easy to see that the method works and returns sum of the numbers in the array. However, it also does something else as well: as a *side effect* it modifies the array. This is unnecessary for computing the sum as we could use a local variable to hold the temporary sum instead, and the user of this method probably does not expect such a side effect to occur. Undesired side effects make the code difficult to understand and cause hard to find bugs. But not all side effects are undesired; for example, consider a method for sorting the contents of an array in an ascending order. In that case the side effect *is* the desired functionality.

We distinguish between two types of methods: functions and procedures. In object oriented paradigm these relate to the accessor- and mutator methods. Functions are methods that return a value, but do not alter the method parameters in any way. Procedures are ones that alter some of the method parameters, and often do not return a value.

When a method is called in Java, the object parameters are passed as references: if the object contents are altered from within the method, the changes are visible to the caller (and the method is a procedure). This is called *passing* parameters *by reference*. Another option is to *pass by value*; this is how primitive data types are passed in Java - a local copy of the parameter is created and any changes to it are visible only within the method scope. Also more complex data types (e.g. arrays) can be passed by value, meaning that in the method call a local copy has to be created. This can be extremely inefficient - for example, consider a method that finds a smallest element in the array. As there is no side effects, making a local copy of the parameter in unnecessary, but if *pass by value* scheme is used on the language level, such is always created, making it impossible to implement the algorithm efficiently.

Matlab consistently passes parameters by value and as a caller you never have to worry about side effects. In case you want to modify one of the parameters, you have to return the local copy and the caller can then store it as the new value. So all methods are functions in Matlab, though passing object handles allows to pass objects by reference (but remember: matrices are _not_ objects in Matlab). In addition, some optimization is made by the Matlab compiler in passing a reference instead of a value if the parameter is not modified within the function.

4.1 Pre- and post-conditions

When you write a method there is always a *contract* between the method designer (the supplier, you) and the caller (consumer, possibly someone else). The method signature defines the contract only partially and its main function is to enable compilation time checking of programming errors preventable through design of the language. However, there are often conditions in the contract that cannot be stated in the signature. For example, consider a method with the following signature for sorting an array from a certain index upwards:

```
function array = sortArrayFromIndex(array, index)
```

Now the contract between method supplier and consumer could consist of the following parts:

1. The `index` has to be in the range `[1, length(array)]` (responsibility of the consumer)
2. If consumer calls the method adhering to (1), then after the method call the following holds:
   `array[index] < array[index+1] < ... < array[length(array)]` (responsibility of the supplier)

These are the method pre- (1) and post-conditions (2). The contract allows us to abstract *what* is computed from *how* it is done. Most programming languages do not provide reserved words for stating pre- and post-conditions, as they anyway cannot be checked

by the compiler. The general convention is to document the conditions in the method documentation, and e.g. in the Java API you will find pre- and post-conditions stated widely. When writing down the conditions, you should be as exact as possible, and it is generally a good idea to state the contract in an algorithmic format if possible. For example, our sorting algorithm's contract could be documented as:

```
% Sorts the array in ascending order starting from index
%
% PRECOND: 0 < index <= length(array)
% POSTCOND: array(index) < ... < array(length(array))
function array = sortArrayFromIndex(array, index)
```

Sometimes, especially in pseudo-code, pre-conditions are documented as "Requires" and post-conditions as "Ensures".

When designing a method where you state the pre-conditions, if the caller does not adhere to the contract, you are not obliged to ensure correct computation from that point onwards - there is a clear mistake in the program logic. However, it is recommended to terminate program execution at this point as finding bugs due to breaching of pre-conditions is often extremely hard. Most programming languages provide a special method `assert(x)` for this purpose. If `x` evaluates to false, the program execution is terminated with a message explaining the assertion. So for example our sorting method's pre-condition could be assured with:

```
function array = sortFromIndex(array, index)
        assert(index > 0 && index <= length(array));
        ... % do the actual sorting
end
```

Often the assertions are enabled only during the program development phase, and when the software is deployed, the assertions are disabled with compiler or virtual machine settings. In Java the assertions are not enabled by default (but need to be enabled in the virtual machine specifically), whereas in Matlab they are always enabled.

4.2 Halting problem

The main difference between mathematical formulas and algorithms is that whereas in mathematics the relations are defined to be invariant by time (e.g. $x = 2$ means that x is 2 now and forever), in algorithms x would be a variable and its value can change over time. This makes proving program correctness extremely hard. So how can we assure that a program eventually ends its execution? Consider the following three algorithms:

```
for(i=1:10)
        printf("%d th integer\n", i);
end
```

```
nr = input("How many integers you want to be printed?");
for(i=1:nr)
        printf("%d th integer\n", i);
        sleep(10*i);
end
```

```
green = true;
while(green)
        green = false;
        sleep(10);
        green = true;
end
```

The first program will iterate 10 times. The second iterates an amount input by the user (maximum of the largest possible integer), and the third one will iterate forever. This brings us to the question that, given a program and an input to the program, can we algorithmically determine whether the program will eventually stop when it is given that input? We could try with the input and wait for a certain amount of time. If the program stops within this time, we know it stops. But if it does not stop, the only thing we can conclude is that it does not stop within the time we have waited. For example, this way we would never find out whether an algorithm with complexity *O(n!)* stops with a worst case input of size 100.

This is known as the *Halting problem:* given a program, decide algorithmically (i.e. with another program) whether the program finishes running or continues to run forever. This problem is equivalent to the previous one that takes the input into account. Let us be optimistic and assume that there exists a solution (a program) H to this problem that takes two inputs:

1. a program P to analyze
2. an input I

H would then output "halt" if H determines that P stops on input I or "loop" otherwise. Remember that with stored program computers the programs are considered data as well: we can now call H with P as both the program to analyze and as the input: *H(P, P).*

Let us now construct another program K that takes the output of H as its input and outputs "halt" if output of H is "loop", and "halt" otherwise. That is, the output of K is the inverse of the output of H given as its input. Given that K is a program, we can use it as the input to K. Now:

- if H says that K halts, then K itself would loop
- if H says that K loops, then K will halt

So given *any* solution H to the halting problem, we can construct an input that always causes the solution to fail. Therefore the halting problem is *undecidable.*

4.3 Loop termination & invariants

So programs cannot be used to solve the halting problem. In procedural programming paradigm the main structure causing infinite computation is iteration (for and while loops). Remember that recursion can always be converted to a computationally equivalent iteration. Although we cannot have algorithms that determine whether a loop will terminate, we can ensure a termination condition ourselves. And we have another technique for assuring correctness of the loop: the *loop invariant*. Whereas pre- and post-conditions allow us to document the contract between designer and user of a method and to control program correctness on a level of abstraction of the method, the loop termination condition and invariant allow to ensure correct internal logic within the method.

Loop invariants state a condition that holds in *the beginning* of the loop and *in the end* of each iteration. To illustrate what I mean with this, let us consider an algorithm for finding the maximum value in an array. The algorithm and its corresponding loop invariant is

```
int i=1;
max = array(1);
while(i < length(array)) % invariant: max = largest of array[1..i]
        if (max < array(i+1))
                max = array(i+1);
        end
        i = i+1;
end
```

Before entering the loop `max` is largest of the `array[1..1]`, so the invariant holds (`max` is `array[1]`). At the end of each iteration, max is has either its previous value, or the next value to check if that is larger than `max.` So the invariant holds also at the end of the loop. By induction on `i` we can prove that the algorithm finds the largest element of the array. The termination condition is simple to assess: on each iteration, we increase the loop counter by 1, and as the termination condition compares the counter to `length(array)` that does not change during the iterations, we can conclude that the algorithm always terminates.

For a more complex example, consider our previously introduced signature of the sorting algorithm with an added implementation of insertion sort with a corresponding loop invariant for the outer for-loop:

```
function array = insertionSortFromIndex(array, index)
        assert(index > 0 && index <= length(array));
        for j=index:length(a)
        % loop invariant: array[index..j] is sorted
                key = array(j);
                i = j-1;
                while i > index && array(i) > key
                        array(i+1) = array(i);
                        i = i-1;
                end
                array(i+1) = key
        end
end
```

Now notice that the loop invariant does not hold *within* the loop: it holds before entering the loop, and at the end of each iteration, but not necessarily within it.

5 Data structures

> "A list is only as strong as its weakest link." (D.E. Knuth)

Programming is about data and operations. Until now I have concentrated mostly on the computational operations. As we already learned when discussing matrices, the way the elements are stored has implications on how much memory is required but also on how fast elementary operations on a *set* of elements takes. For example, matrices are a static data structure and if a 2×2 matrix is resized to a 3×3 one, memory needs to be allocated for the new matrix and then the data copied from the old matrix to the new one. This requires O(n) operations, which is reasonably expensive if only 1 additional element needs to be stored. Other data structures allow dynamic allocation with the cost of O(1), but are slower for accessing an arbitrary element of the set.

Data structures provide means to store *elements* with *keys*. For example, when storing integers, the keys are the actual integer values themselves. The storage structure itself is dynamic: new elements can be added and removed, and an element with a certain key can be seached for. If the structure allows duplicates of elements with keys, it is a *bag* or a multiset. For example, arrays are bag structures as they allow to store twice the element with the same key. If duplicates are not allowed, we talk of a *set* that has semantics of a mathematical set. In addition, the set of elements can be ordered, partially ordered, or unordered.

Knowing data structures is a prerequisite for programming efficiently as there always exists a complexity trade-off between the different operations involved in searching, storing and retrieving values with the chosen data structure. Consider the most elementary data structure: an array. Now when you initialize an array, a certain amount of memory is allocated in the linear memory, exactly the size of the array. At some point you might need to add new elements into the array, so new memory has to be allocated and the elements copied. You could imagine allocating a larger block of memory in the first place and using a variable to store the amount of free places. But at some point you might need

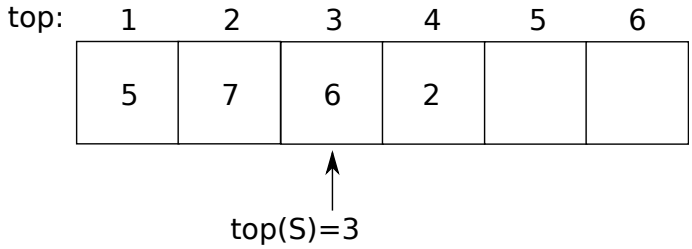

Figure 5.1: Stack with an array of size 6 after 4 pushes (5, 7, 6, 2) and one pop (removing 2).

more space than is contiguously available. So there is no way to ensure O(1) insertion operation with standard arrays - the operation is inherently of complexity O(n). For constant time insertion, we have to reconsider how to store and access the elements, and to use more advanced data structures. The table below shows the complexity of standard operations and data structures I will discuss in this section.

Structure	Random access	Insertion	Deletion	Search	Min/Max
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Linked list	O(n)	O(1)	O(1)	O(n)	O(n)
(balanced) Tree	N/A	O(log n)	O(log n)	O(log n)	O(log n)
Heap	N/A	O(log n)	O(log n)	O(n)	O(1)

5.1 Stacks and Queues

Before we discuss the other data structures, let us start with two elementary ones, stack and queue, that simply restrict the use of a standard array. Stacks are arrays that allow elements to be *pushed* to the top, and *popped* from the top as well. Stacks implement the so-called last-in first-out (LIFO) semantics: the last element that was pushed is the first one to get popped. Queues implement first-in first-out (FIFO) behaviour: the first element that was *queued* is the first one to be *dequeued*. Stacks and queues differ from arrays in that random access is not allowed: the elements can only be added to the end and accessed from the end (stack) or from the beginning (queue) of the structure.

Stacks can be implemented as an array augmented with an additional variable indicating the top index of the stack. This is illustrated in Figure 5.1: the stack is represented with an array of size 6. Initially the stack is empty and top(S) = 0. Then, when push(S, 5) is executed, top(S)=1. Three more elements are pushed (7, 6 and 2), after which top(S) = 4. Finally the last element is popped and top(S) becomes 3. Note that there is no need to remove the element from the array unless it is an object reference (otherwise garbage collection would not free the memory used by the object).

When more than 6 elements are added to the stack, a new array needs to be allocated and the current elements copied there. So the complexity of operations of a stack is at most the complexity of operations of an array without the possibility of random access. Although it offers "just" restrictions with no gains in operation complexity, stack is widely used due to its close link with recursion. For example, when a program is loaded into execution, a certain part of the main memory is reserved for its local stack, and when a method call is made, the local variables and the return address from the method are

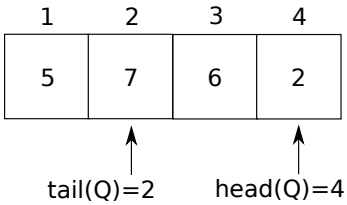

Figure 5.2: Queue with an array of size 4 after 4 queues (3, 7, 6, 2), 3 dequeues (remove 3, 7 and 6), and 1 enqueue (5): contents of Q are [2, 5].

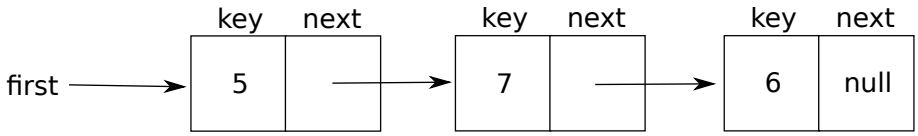

Figure 5.3: Single-linked list with three elements representing linear order [5, 7, 6].

pushed to the stack. When the method returns, these are popped. This is an efficient and simple technique allowing a virtually unlimited depth of recursive method calls (until the memory runs out).

Queues are composed of a *head* and a *tail*. When new elements are inserted into the queue (i.e. queued), they are placed at the tail of the queue. When an element is removed (i.e. dequeued), it is taken from the head of the queue. The most efficient way to implement queue in an array is to use a circular structure: that is, if the tail goes past the end of the array, it is moved to the beginning. This is shown in Figure 5.2. Note that the enqueue operation can cause an overflow if the array is not large enough to hold all the elements and the operation can be implemented so that it allocates new memory when this happens. Dequeue and pop are not applicable when the corresponding data structure is empty; this should be documented with a pre-condition.

5.2 Linked list

Linked list is a data structure in which each element is allocated its own *node* that points to the following node. Like the array, it is a linear order, but without the possibility of referring in O(1) time into a random element. Each node has a key, that is, the data contents of the node, and a reference to the next node. A sample list structure is presented in Figure 5.3. A linked list can be referred to with its first element.

When a program is loaded into execution, it is organized into three segments: text, stack, and heap. The text segment contains the actual executable machine code. As mentioned in connection to stacks, the stack segment is used for storing method return addresses (to the text segment) and temporary variables of the method calls. The third segment, heap, is the memory segment used for dynamic memory allocation (e.g. with Java's `new` operator). Heap allocation in Matlab is very limited and most of data types can only be referred to by value: only objects can be passed around by references that are called handles in Matlab. For implementing linked list nodes, we can use a class with no methods and public access to its member variables:

```
classdef node < handle
        properties
                key
                next
        end
end
```

We need also another class for holding the first node:

```
classdef linkedlist < handle
        properties
                first
        end
end
```

Note that there are no null pointers in Matlab, but we can use [] (empty matrix) to simulate a null. A list can be now initialized by constructing a first node and referring to it in the list structure:

```
function L = initLinkedList(value)
        L = linkedList();
        fNode = node();
        fNode.key = value;
        fNode.next = [];
        L.first = fNode;
end
```

Searching for a node with a certain key is O(n) operation like with arrays, and can be done as follows:

```
function node = findKey(L, key)
        curNode = L.first;
        while (curNode != [])
                if (curNode.key == key)
                        node = curNode;
                        break;
                end
                curNode = curNode.next;
        end
end
```

Inserting nodes to the list can be made in the beginning or after a certain node. When a node is inserted to the beginning, the firstNode reference has to be changed. When a node is inserted after another node, re-routing of the references has to be done. Figure 5.4 illustrates these situations. The insertion algorithms are:

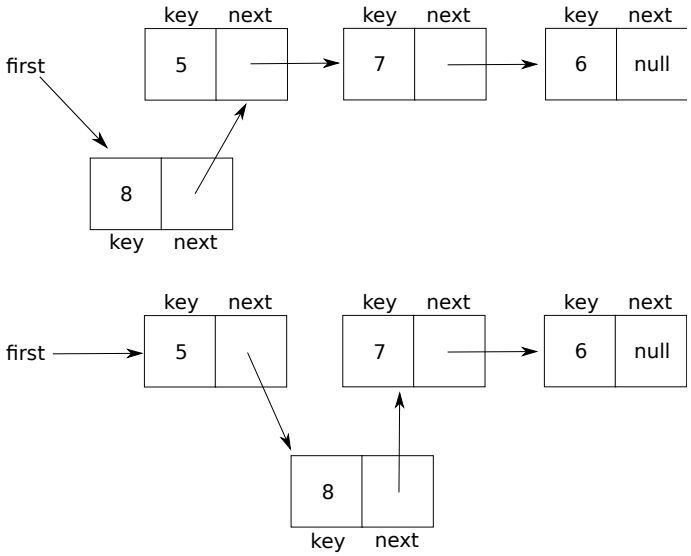

Figure 5.4: Insertion a node with key 8 to the beginning (top) and to after the node with key 5 (bottom).

```
function insertIntoBeginning(L, value)
        newNode = node();
        newNode.key = value;
        newNode.next = L.first;
        L.first = newNode;
end

function insertAfterNode(node, value)
        newNode = node();
        newNode.key = value;
        newNode.next = node.next;
        node.next = newNode;
end
```

When a node is deleted, it can simple be spliced out from the list, and the links pointing to and from it rerouted. This is shown in Figure 5.5. Note that for deleting a node, we need to know which node points to it in order to do the re-routing. Deleting the first node is easier: we just need to set the list's first to point to the second node. The algorithms for deletion are:

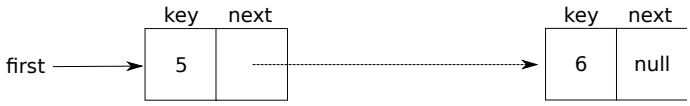

Figure 5.5: Deleting the second node from the linked list of Figure 5.3.

```
% PRECOND: node.next != []
function deleteNodeAfter(node)
        node.next = node.next.next;
end

% PRECOND: L.first != []
function deleteFirstNode(L)
        L.first = L.first.next;
end
```

Linked lists can be used for representing stacks, and provide O(1) push and pop operations, although the memory required is larger as for each element in the stack also the reference to the next node needs to be stored. Linked lists can also be used as circular structures, and in this case the `next` of the last node in the list will point to the first node, as illustrated in Figure 5.6.

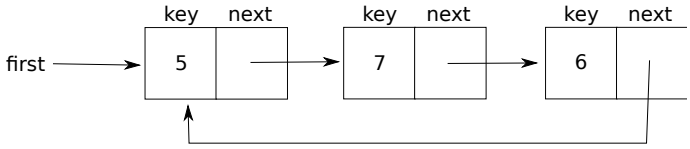

Figure 5.6: A circular linked list.

Linked lists can also be double-linked: this allows to navigate the list both forward and backward. Instead of only a reference to the next node, the double linked list nodes have references to both the previous and the next node. Additionally the list structure has references to both first and last nodes of the list instead of just to the first one.

5.3 Trees

Trees are the most important nonlinear structure. They are acyclic undirected graphs with one node designated as the root of the tree. Trees are defined recursively as:

1. empty T is a tree
2. if T is not empty, a T has exactly one node designated as the root(T)
3. the remaining nodes ($T - root(T)$) of a tree are partitioned into m disjoint sets $T_1, \ldots, T_m$. Each of these are in turn a tree, and are called subtrees of T.

Tree nodes can be implemented in Matlab similarly to how we did the nodes of linked lists:

```
classdef treeNode < handle
        properties
                key
                left
                right
        end
end
```

The tree can then be referred to simply by referring to the root node. Nodes in a tree relate to each other similarly as people do in a conventional family tree: Figure 5.7 presents an example tree with these relations. Trees have always a single root (here 6). Here the root has three children (9, 3 and 2). Each of them has the other two as siblings. The root is a parent of 9, 3 and 2. 9 is an ancestor of 5, and 5 is a descendant of 9. This tree has 4 levels: they are numbered starting from 0 (the root). Depth of the tree is its maximum level, in this case 3, which is the maximum amount of edges from the root to any node. Nodes without any children are called leafs.

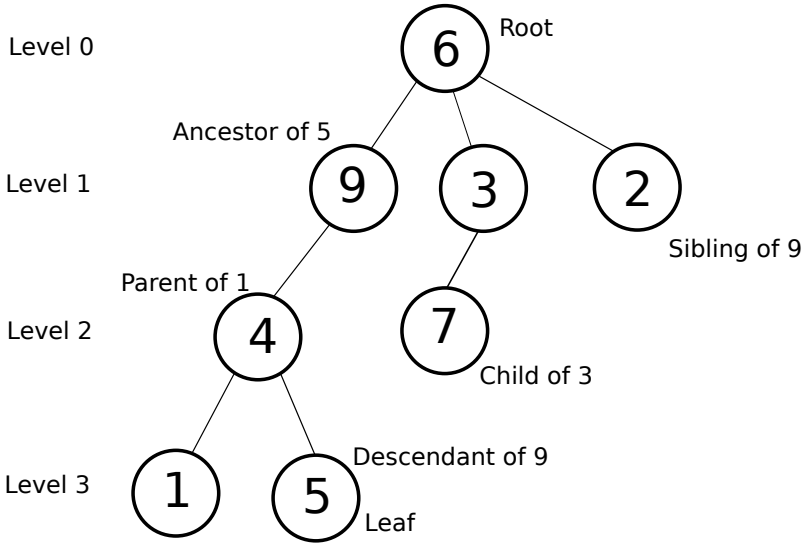

Figure 5.7: An example tree presenting different relations between its nodes.

Usually trees are used with a maximum amount of subtrees m. The simplest trees are binary trees with $m = 2$ ($m = 1$ would be a linked list). Binary trees are especially handy for presenting ordered data. For example, arithmetic expressions can be represented with tree structures by taking into account standard rules of operator precedence (* and / before + and -). The following arithmetic expression

$$a - b * (c/d + e/f)$$

can be represented with the binary tree in Figure 5.8. The order of traversing the tree is important here: to "read" the original arithmetic expression we have to process the tree with an inorder traversal scheme:

1. Traverse the left subtree

2. Visit the root

3. Traverse the right subtree

There are two other ways to traverse the tree as well: preorder traversal (root, left subtree, right subtree) and postorder traversal (left subtree, right subtree, root). Note that traversing the tree representing the arithmetic expression with the preorder traversal results in polish notation (-a*b+/cd/ef), and the postorder traversal in reverse polish notation (abcd/ef/+*), both eliminating the need for parentheses.

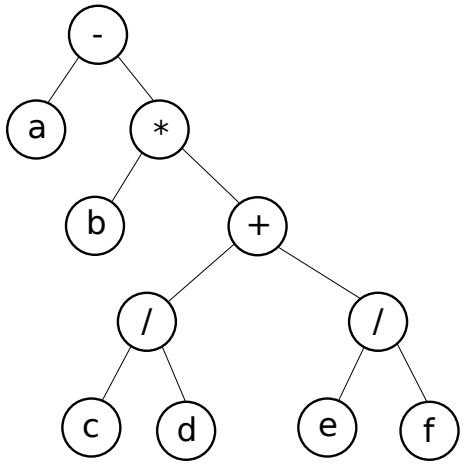

Figure 5.8: Binary tree representation of $a - b * (c/d + e/f)$.

A binary search tree is a binary tree with the extra property that key of the root of the left subtree < root < key of the root of right subtree. When data is structured as a binary search tree, finding an element with a certain key can be fast. The algorithm for finding a node with a certain key is:

```
% Returns a node with the given key, or [] if such does not exist
function n = findNode(T, key)
        if (T.key == key)
                n = T;
        elseif (key < T.key)
                if (T.left == [])
                        n = [];
                else
                        n = findNode(T.left, key);
                end
        else % key > T.key
                if (T.right == [])
                        n = [];
                else
                        n = findNode(T.right, key);
                end
        end
end
```

For example, consider the binary search tree in Figure 5.9. For finding the node with key 3 the search proceeds from the root to its left child (4) and from there to the left child (1), where the search terminates returning [] as the key is not found. Trees that have a small difference between the lowest and highest levels of leafs are said to be *balanced*, and nodes with certain keys can be found quickly in such trees. For example, finding any node in the tree in Figure 5.9 takes maximum 3 comparison operations - a lot less than searching in a linear list (7). For balanced trees, the complexity to find a certain node is O(log n).

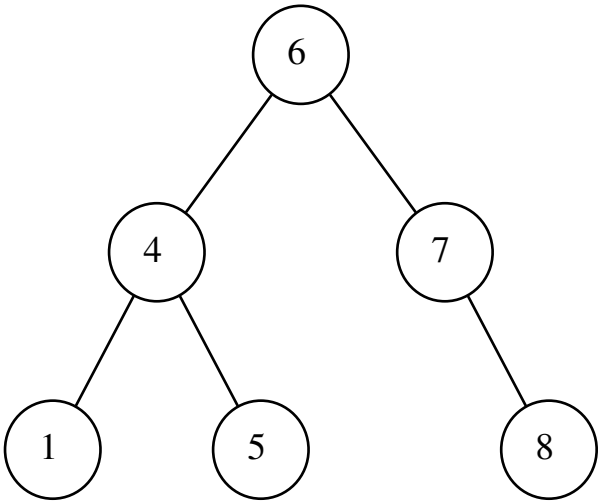

Figure 5.9: A 3-level binary search tree.

5.4 Heap

Heap is a special type of binary tree that is *complete* - that is, each level of height x except the last one has exactly 2^x nodes (root has 1, first level 2, second 4, ...). In addition heaps fulfill two additional properties: (1) the last level of the tree is filled from left to right, i.e., there can be empty slots only on the right side of the last node in the last level, and (2) the key of each node in the heap is maximum the key of the parent (i.e. 11 cannot be the child of 10). The second condition is called the *heap property*, and it distinguishes heaps from other types of trees. An example heap is shown in Figure 5.10. Note that heaps are not binary search trees, as the left child of a node can have larger value than the right child.

Heaps support the operation of retrieving the *largest* (or alternatively the smallest, these are called max- and min-heaps, respectively) value in O(1) time, as it is always the root node. As heaps are complete binary trees, they can be stored in an array so that the i^{th} element of j^{th} level is located in the index

$$2^j + (i - 1)$$

For example, the heap of Figure 5.10 can be stored in an array as

9	7	5	6	4	1

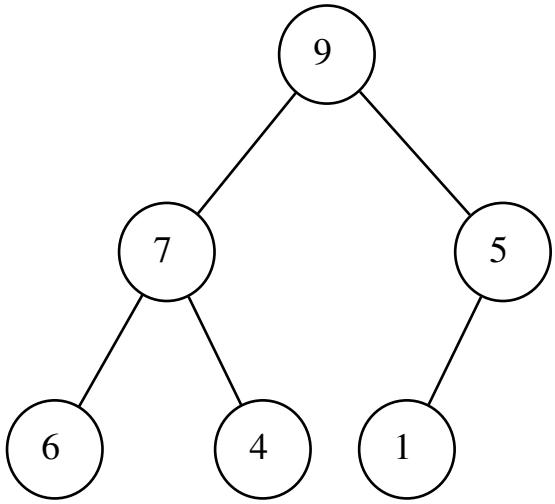

Figure 5.10: A heap.

The array representation is useful as it allows us to also find the parent of a node in O(1) time without explicitly storing in each node a reference to its parent. The parent of node n is in index $\lfloor n \rfloor$, the left child is in index $2n$, and the right one in index $2n+1$. Knowing this we can construct shortcut functions for obtaining the correct indices:

```
function l = left(n)
        l = 2*n;
end

function r = right(n)
        r = 2*n + 1;
end

function p = parent(n)
        p = floor(n / 2);
end
```

When a node is added to the heap, it always becomes the rightmost node of the last layer (in case it still has space), or the first node of a new layer (in case the last layer is full). This assures that the heap continues being a complete tree. For example, adding a node 10 to the heap of Figure 5.10 results in what is seen in Figure 5.11 (a). Now the heap property is violated, however, and the heap needs to be corrected by moving the largest value up through the heap. The function for this is:

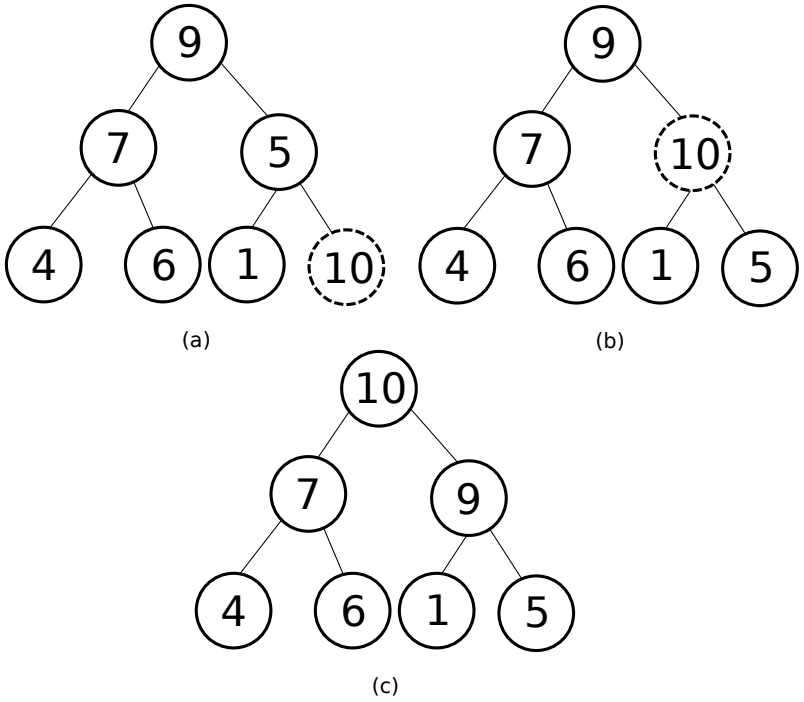

Figure 5.11: `heapInsert` adding node with key 10. First the node is added, and then it is iteratively "bubbled" to its correct position.

```
function H = heapInsert(H, key)
        i = length(H) + 1;
        while (i > 1 & H(parent(i)) < key)
                H(i) = H(parent(i));
                i = parent(i);
        end
        H(i) = key;
end
```

The insertion of 10 starts with `heapInsert(H, 10)`: 10 is added as the last node in the heap (Fig. 5.11 (a)). In the first iteration of `while` loop, as 10 is larger than 5, 10 is replaced with 5 (Fig. 5.11 (b)). In the second iteration 10 is larger than 9 and 9 is replaced to the location `i` is pointing at (right child of the root). The while loop ends, and 10 is placed at its correct location at the root. Note that there is no need to examine left children of the nodes processed during the iterations as the heap property does not say anything about the relation between the children, and as if the added node is larger than its parent, it is then surely also larger than its sibling. For this reason the `heapInsert` is of complexity O(log n).

Deletion of a node from the heap is always made by removing the top node – in this way the heap implements *priority queue* semantics: nodes may be added in arbitrary order, but when they are removed, the order is based on the key values; the largest node in the heap comes out first. When a node is removed from the top, it is replaced with the *last* node of the heap. This naturally leads to the heap property being violated. Let us remove the root node from the heap presented in Figure 5.11 (c) and replace it with the last node. This leads to situation depicted in in Figure 5.12 (a). Now to correct the heap, we take the

largest of its children (9) and swap the root (5) with it. This leads to the heap property being restored from root to the left subtree. Then heapify is recursively performed on the new location of 5 (Figure 5.12). As it is larger then both of its children, it also fulfills the heap property and thus the whole heap H does and the algorithm can terminate. This `heapify` function is:

```
function H = heapify(H, n, endIndex)
        largest = 0;
        l = left(n)
        r = right(n)
        if (l <= endIndex && H(l) > H(n))
                largest = l;
        else
                largest = n;
        end
        if (r <= endIndex && H(r) > H(largest))
                largest = r;
        end
        if (largest != n)
                temp = H(largest);
                H(largest) = H(n);
                H(n) = temp;
                H = heapify(H, largest, endIndex);
        end
end
```

and the full function for removing the root node from the heap:

```
function H = heapRemove(H)
        max = H(length(H));
        H(1) = H(length(H));
        H(length(H)) = [];
        heapify(H, 1, length(H));
end
```

The `heapify` function takes O(log n) time as it performs maximum O(1) operations for each level of the heap. `heapRemove` is of the same complexity as it performs only constant amount of work in addition to calling `heapify`.

All the heap operations I presented run in time O(log n) - so relatively fast even for large sizes of n. Heaps are very useful data structures as you often need to store elements in a way that they are queued in order of the keys. For example, remember Dijkstra's algorithm from the first programming course? Its complexity can be improved by using Fibonacci heaps that are a heap structure[3].

[3]They are out of our scope, but if you are interested, see `http://en.wikipedia.org/wiki/Fibonacci_heap`

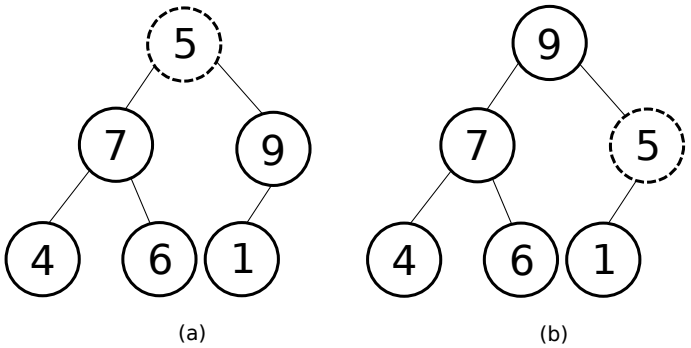

Figure 5.12: Heapifying after replacing the root node with the last node.

6 Sorting & searching

"2 or more, use a for" (E.W. Dijkstra)

The heap data structure presented in the previous section has one important use I did not yet discuss: sorting. For heap to be used for sorting, we have to first consider building a heap from an array A of arbitrary integers. This can be achieved iteratively with:

```
function A = buildHeap(A)
        s = floor(length(A)/2);
        while (s > 0)
                heapify(A, s, length(A));
                s = s - 1;
        end
end
```

To see how buildHeap proceeds, consider building one from the array [2 3 1 7 4 6] (Figure 6.1).

A good asymptotic bound for `buildHeap` is not so easy to find. The `while` loop goes from length(A) / 2 to 1, so $n/2$ iterations. However, the time taken by heapify at each iteration varies according to the height of the currently considered tree. In an n-node heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h. The complexity of `heapify` when called with a node of height h is O(h) (i.e. O(log n) for full heap, where $h = \log_2 n$). Then the total cost of `buildHeap` is

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right)$$

The summation is equal to 2, so the running time is O(n). Now given that we have constructed a heap from an unsorted array in linear time, let us repeatedly swap the root (=the largest node) with the last one and fix the heap of one length shorter. In this way the last element in the array will be the largest, the second last the second largest, etc, so the array will be sorted in an increasing order. The code for this is:

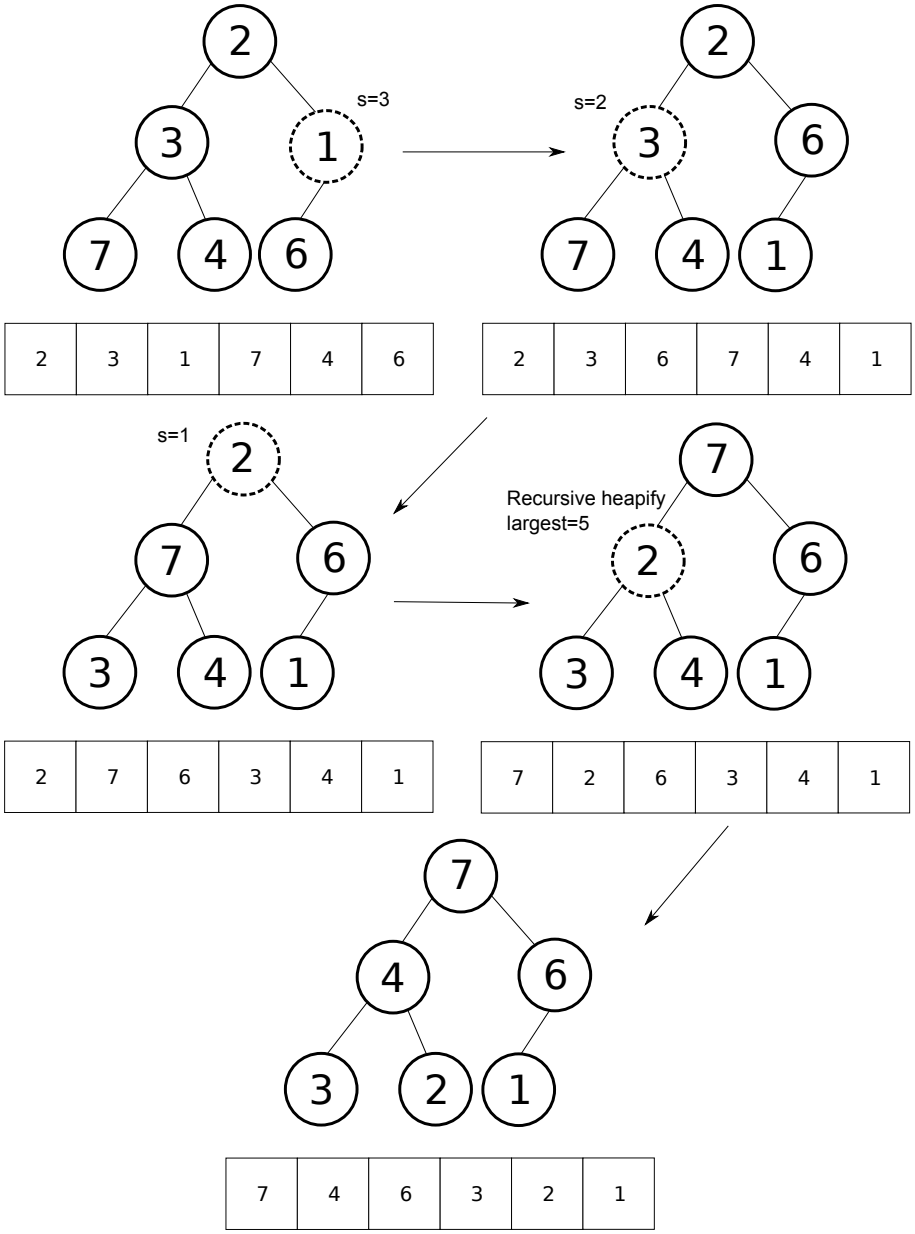

Figure 6.1: Iterations of buildHeap when executed with [2 3 1 7 4 6].

```
function A = heapSort(A)
        s = length(A);
        while (s > 1) % until s == 2, but this is a safer condition
                % swap A(1) and A(s)
                tmp = A(1);
                A(1) = A(s);
                A(s) = tmp;
                A = heapify(A, 1, s);
                s = s - 1;
        end
end
```

Execution of heapSort on the heap of Figure 6.1 is shown in Figure 6.2. Heapsort takes O(n) time to build the heap, and `heapSort` does n-1 iterations in which each the function `heapify` is executed that in turn takes O(log n) time. So the total complexity is O(n) + O(n log n) = O(n log n). Note that heapsort, like insertion sort, does the sorting in place and requires only a constant amount of additional memory.

6.1 Design of algorithms (incremental & divide-and-conquer)

The sorting algorithms I have presented until now, insertion sort and heapsort, implement an incremental approach to sorting: the array is sorted iteratively so that the problem size decreases by each iteration. The Strassen's matrix multiplication algorithm applied another kind of approach: dividing the problem into multiple subproblems and applying the same division to these until a problem that is trivial to solve was reached. Then the solution to the original problem was composed of these subproblem solutions. This is called the divide-and-conquer approach, and algorithms applying it can sometimes be surprisingly fast, as we will see in the following subsections. The term divide and conquer comes from the way Caesar administered the roman empire by splitting the conquered areas in smaller regions that could then be easier ruled than larger ones which might unite and challenge the Romans.

6.2 Mergesort

Let us consider a divide-and-conquer approach for sorting an array. For this we first need to recognize the trivial case: an array of length 1, as it is always necessarily sorted. Any larger arrays we can divide in two subarrays of approximately the same size. By doing this recursively we divide the original array to pieces of size 1 as is demonstrated by the first 4 levels of Figure 6.3 (for sorting the array [38 27 43 3 9 82 10]). This is not yet too innovative as we did not actually sort anything yet – we just divided the array in a layered manner in atomic parts. The second part, that is the last three layers of Figure 6.3, does the magic: after we have arrived at the trivial arrays of size 1, we combine them one layer at a time so that each time the combined array is sorted, leading into having the original array sorted in the end.

The procedure outlined above is called mergesort due to the second phase where the subarrays are merged together. The full procedure for mergesort is:

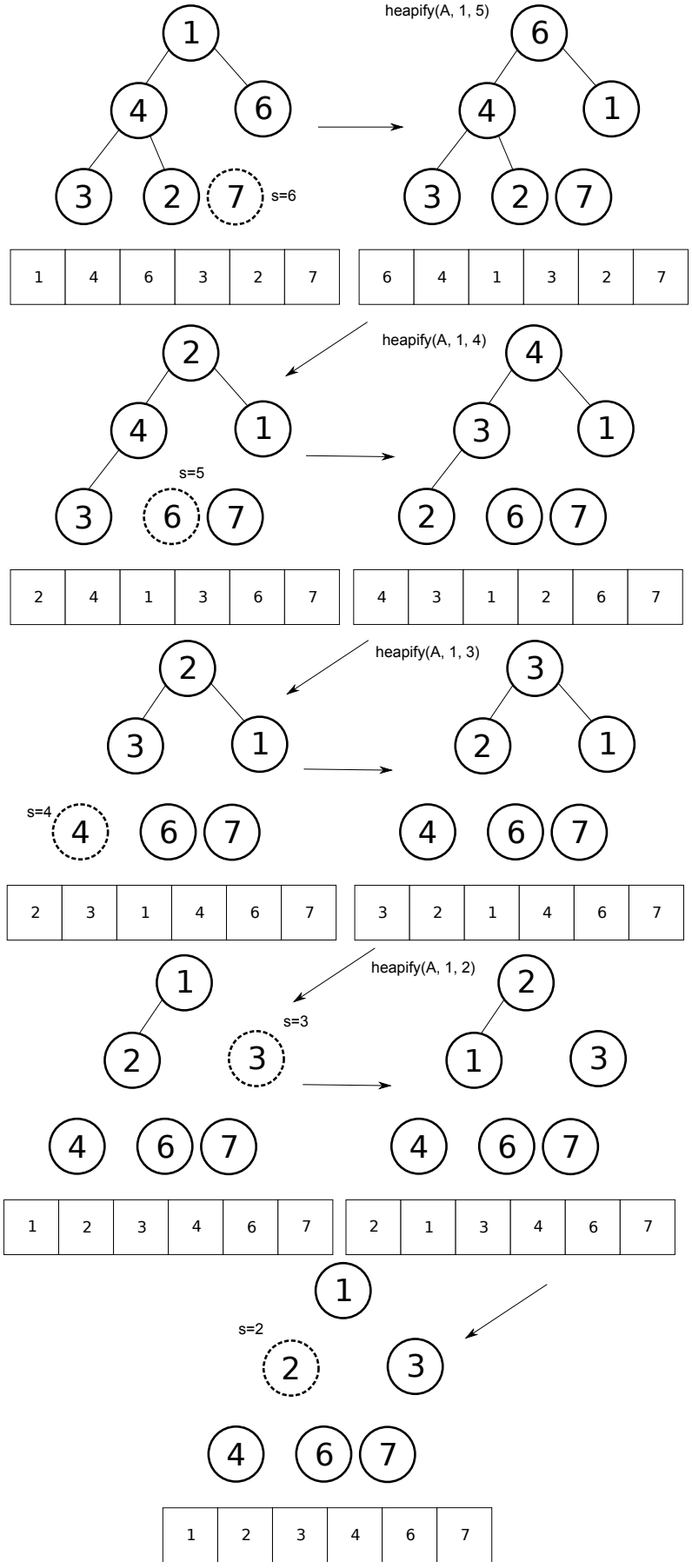

Figure 6.2: Iterations of heapSort on heap [7 4 6 3 2 1].

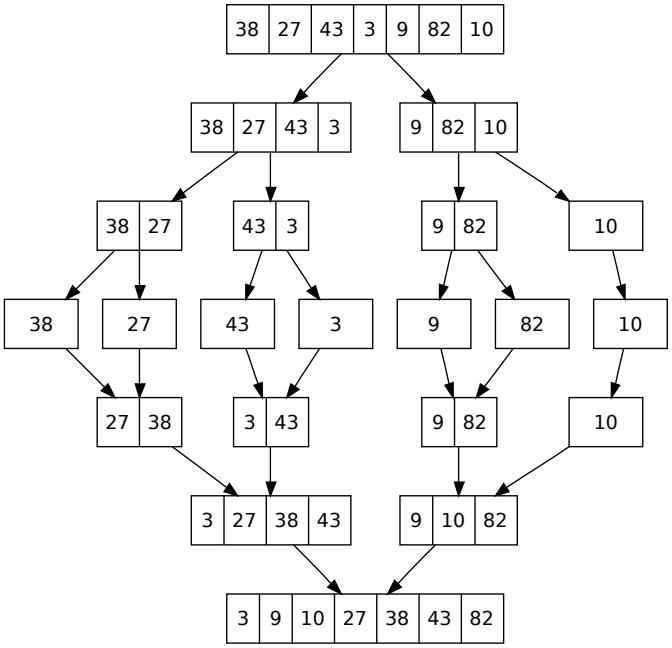

Figure 6.3: Mergesort on array [38 27 43 3 9 82 10]. (source: wikipedia)

```
function A = mergeSort(A)
        if (length(A) > 1) % if not, it's our trivial case
                middle = floor(length(A) / 2);
                leftList = A(1:middle);
                rightList = A((middle+1):length(A));
                leftList = mergeSort(leftList);
                rightList = mergeSort(rightList);
                A = merge(leftList, rightList);
        end
end
```

and for this we need also a method for merging the two arrays:

```
function c = merge(a, b)
        lena = length(a);
        lenb = length(b);
        c=zeros(1,lena+lenb);
        inda = 1; % index to move along vector 'a'
        indb = 1; % index to move along vector 'b'
        indc = 1; % index to move along vector 'c'

        while ((inda <= lena) && (indb <= lenb))
                if a(inda) < b(indb)
                        c(indc) = a(inda);
                        inda = inda + 1;
                else
                        c(indc) = b(indb);
                        indb = indb + 1;
                end
                indc = indc + 1;
        end

        % copy any remaining elements of the 'a' into 'c'
        while (inda <= lena)
                c(indc) = a(inda);
                indc = indc + 1;
                inda = inda + 1;
        end
        % copy any remaining elements of the 'b' into 'c'
        while (indb <= lenb)
                c(indc) = b(indb);
                indc = indc + 1;
                indb = indb + 1;
        end
end
```

The complexity of the full mergesort is easy to analyze. In each step, we make two recursion steps each with an input half the size of the current one, so their total cost is 2T(n/2). In addition we have to merge the two lists. Now note that as we assume that the lists we obtain from recursion are already ordered, merging them is easy and takes only n operations. So the total cost is T(n) = 2T(n/2) + n, that evaluates as O(n log n). An intuitive way of arriving to this upper bound is by looking at Figure 6.3: the amount of levels for reaching the trivial sorting (arrays of length 1) is $\log_2 n$, and this is the same amount that is taken for coming "backwards" from the recursion when in each level we have to make O(n) operations for merging the arrays together. This gives the correct complexity log n * O(n) = O(n log n).

Note that mergesort does not do the sorting in place, but instead requires extra memory for holding the temporary lists: n for the first level, then n/2, n/4, ..., so the total memory complexity for arriving until the end of the recursion is 2n = O(n). When coming backwards from the recursion the parts already "used" do not have to be maintained in a space-efficient implementation, so the overall memory complexity is O(n).

6.3 Quicksort

Mergesort is an O(n log n) complexity algorithm for sorting, and although it is in practice faster than heapsort due to smaller coefficients hidden in the O-notation, it does not sort in place. So heap and mergesort already provide us two sorting algorithms with low asymptotic complexity and a trade-off between computational- and memory complexity. However, the most used sorting algorithm is neither merge- nor heapsort, but quicksort. The basic idea of quicksort is similar to mergesort: in each step, the array is divided in two parts that are sorted recursively until a trivial case (an array of size 1) is reached.

The division phase in quicksort is different from that of mergesort: instead of splitting the array in two approximately equal sized subarrays, the first element of the array is chosen as the *pivot*, and the array is *partitioned* so that the elements left from the pivot will be smaller or equal to the pivot, and all the elements to its right larger than the pivot. The partitioning is done iteratively by growing these two partitions starting from the first and the last indices, respectively.

The complete quicksort is:

```
% Sort array A from index p to index r
function A = quickSort(A, p, r)
        if (p < r)
                [A, q] = partition(A, p, r);
                A = quickSort(A, p, q);
                A = quickSort(A, q+1, r);
end
```

and the required partition method:

```
% Partition array A from indices p to r (inclusive) so,
% that each A[p...q] is < A[q+1...r]
function [A, q] = partition(A, p, r)
        x = A[p];
        i = p-1;
        j = r+1;
        while (i < j)
                done = false;
                while (!done)
                        j = j-1;
                        if (A(j) > x)
                                done = true;
                        end
                end
                done = false;
                while (!done)
                        i = i+1;
                        if (A(i) < x)
                                done = true;
                        end
                end
                if (i < j)
                        % swap A(i) and A(j)
                        temp = A(i);
                        A(i) = A(j);
                        A(j) = temp;
                end
        end
        q = j;
end
```

As an example, consider partitioning the array [3 7 8 5 2 1] as shown in Figure 6.4. The sorting starts by choosing the first element (3) as the pivot. Then the partitions on the left and right ends of the array are grown until an element is found from the left side that is ≥ 3, and from the right size that is ≤ 3. These are 3 (the pivot) and 1, respectively, and they are swapped. Then the indices `i` and `j` move again to grow the partitions, and stop after 1 step - and these are then swapped. In third iteration `j` passes over `i`, so now the array is partitioned: elements [1:2] are smaller than the pivot (3), and elements [3:6] are equal or larger than the pivot. Now we can recursively sort again these subarrays.

The complexity of quicksort depends heavily on the partitioning: the worst case happens when in each recursion step the largest element is chosen as the pivot, as then the partitioning becomes extremely unbalanced: left side will have n-1 elements and the right side only one (the pivot). Now if in the next recursion step the largest element is chosen again as the pivot, there are n-2 alements on the left side and one (the pivot) on the right side. This continues until the trivial case (1 element) is reached, so the complete cost is

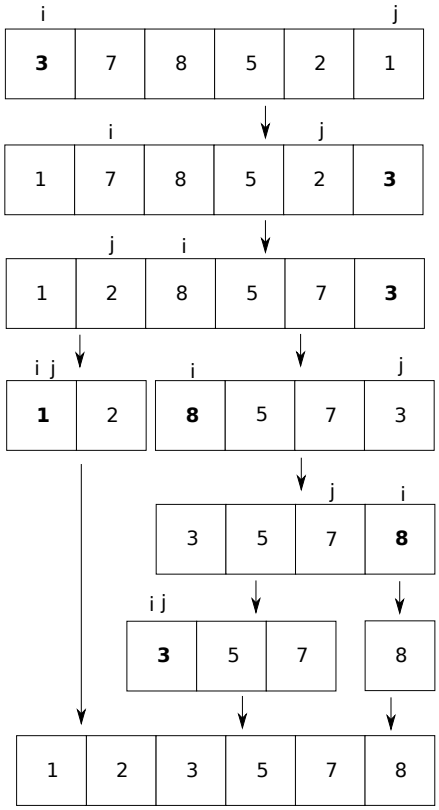

Figure 6.4: All steps of quicksort on [3 7 8 5 2 1].

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= \sum_{k=1}^{n} O(k) \\ &= O(\sum_{k=1}^{n} k) \\ &= O(n^2) \end{aligned}$$

which is not better than insertion sort, and is worse then heap- and mergesort. As it has $O(n^2)$ worse case complexity, why has quicksort been for decades the "de facto" sorting algorithm? As the worst case performance is quite bad, what about the best case one? In best case the partitioning in each recursion step results in the median element being chosen as the pivot, which leads to the left and right partitions becoming approximately equally large. In this way the complexity becomes:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log_2 n) \end{aligned}$$

that is the same as for quick- and mergesort, though the underlying constants are smaller and therefore quicksort is in practice often faster than merge- or heapsort. Still, the worse-case complexity of $O(n^2)$ is not making quicksort a too attractive sorting algorithm for

us. The worse case happens with already sorted lists which can be quite often if the input comes from a real-life process (e.g. sorting customers according to their ID numbers). But what if we can guarantee some semi-balanced partitioning? That is, if for example in each partitioning step the array gets split 9-to-1? Then the complexity becomes

$$\begin{aligned} T(n) &= T(9n/10) + T(n/10) + n \\ &= O(n \log_{10/9} n) \\ &= O(n \log n) \end{aligned}$$

as the base of the logarithm does not affect the asymptotic running time. Similarly, a 99-to-1 split yields O(n log n) complexity, so on *average* quicksort performs as good as merge- and heapsort. The problem is still that sometimes the worse case scenario happens. A way out from this is by using a non-deterministic modification of the algorithm and in each recursion step before the partitioning to swap a random element of the array with the first one, and use the new first element as the pivot. In this way we get an *expected* running time of O(n log n) with a very high probability for any input. Analysis of the randomized version is more complicated and out of our scope, but in practice most quicksort implementations use its randomized version with the following modified partitioning algorithm:

```
function A = randomizedPartition(A, p, r)
        i = p + (round(rand(1) * (r-p)));
        temp = A(p);
        A(p) = A(i);
        A(i) = temp;
        A = partition(A, p, r);
end
```

6.4 Binary search

Now that we know multiple ways to sort an array, let us consider finding elements from the array. In Section 5.3 I introduced binary search trees, and when we discussed heaps, we saw that they were representable with arrays. This is true also of trees, although representing them with dynamic structures is most of the time more suitable than an array representation. But let us now consider how to represent as a binary search tree a sorted array such as:

1	4	5	6	7	8

There are multiple ways to represent it, but recall that for minimal search cost the tree should be balanced: the difference with the lowest and highest levels of leaf nodes should be minimized. We can achieve this with the given array by taking the medium node as the root, elements left of it as its left subtree and elements to the right as its right subtree. We repeat the same procedure recursively for the left- and right subtrees, and end up with the tree in Figure 5.9. Now that the array is represented as a binary search tree, writing a search procedure is trivial, and any element can be found in O(log n) time.

This shows us that data structures are often only different ways of looking at a problem, but this different view can lead to new ways at approaching the problem, which can enable invention of clearer and/or more efficient algorithms. In the end, the main goals in programming are efficiency of the computation and simplicity of the solution. Most of the time these two goals are aligned. As T. Hoare put it:

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Acknowledgements and further reading

Although most of these lecture notes are written roughly from my own memory, I have complemented widely with existing literature. If you want to learn more about complexity analysis and algorithm design, I recommend Leicerson, Cormen, and Rivest: Introduction to Algorithms (MIT Press). For a comprehensive, entertaining and practical introduction to everything you need to know about programming, see Knuth: The Art Of Computer Programming (vols 1-4A, Addison-Wesley). Some inspiration on matrices I got from Goulb and van Loan: Matrix Computations (The Johns Hopkins University Press). I also acknowledge the countless lecture notes, slides, and wiki entries I encountered in the internet while writing this - my gratitude goes to their authors for making them freely available.

Although being modest in size, I'm sure these lecture notes contain countless errors. As Linus Torvalds said, "given enough eyeballs, all bugs are shallow". I would be extremely grateful for the readers to report any errors found, whether they are on contents, writing style or spelling, by email to `ln-bugs@smaa.fi`.