INF421: Data Structures and algorithms

Leo Liberti

LIX, Ecole Polytechnique, 91128 Palaiseau, France

`liberti@lix.polytechnique.fr`

August 21, 2012

To[1] *Much*

---

[1] Not a spelling mistake.

Chers élèves,

vous avez entre vos mains la première version de mon polycopié pour le cours INF421 (*Les bases de l'algorithmique et de la programmation*) de l'Ecole Polytechnique. En tant que première version, ce polycopié contiendra sans doute beaucoup d'erreurs: merci de m'envoyer un email à `leoliberti@gmail.com` pour m'en faire part, si vous en trouvez.

Je vous rappelle que le website du cours INF421

`www.enseignement.polytechnique.fr/informatique/INF421/`

contient beaucoup de matériel didactique. Le blog

`inf421.wordpress.com`

contient du matériel didactique supplémentaire quand le cours est actif. Pour le reste de l'année, j'y publie n'importe quoi.

LEO LIBERTI, Paris, Août 2012

Contents

I Preliminaries and reminders 15

1 Computation 17

1.1 Computer hardware 17

1.1.1 Programs 17

1.2 Computer model 18

1.2.1 Models of computation 18

1.2.2 Church Hypothesis 18

1.3 Languages 18

1.3.1 Declarative languages 19

1.3.2 Decidability 19

1.3.3 Java 19

1.4 Efficiency 20

1.4.1 Structuring data 20

1.4.2 Problems 21

1.4.3 Algorithms 21

1.4.3.1 Algorithmic complexity 22

1.4.3.2 Worst-case time complexity calculations 22

1.4.3.3 Complexity orders 23

2 Java basics 25

2.1 Development of a Java program 25

2.1.1 Variables 26

2.1.1.1 References 26

- 2.1.2 Comments . . . 26
- 2.1.3 Classes . . . 27
  - 2.1.3.1 The `this` attribute . . . 27
  - 2.1.3.2 Inheritance . . . 28
  - 2.1.3.3 Interfaces . . . 28
- 2.1.4 Functions . . . 29
  - 2.1.4.1 Passing by reference or value . . . 30
  - 2.1.4.2 The `main` function . . . 30
  - 2.1.4.3 Specifiers . . . 30
- 2.1.5 Data types . . . 31
- 2.1.6 The dot operator . . . 31
- 2.1.7 The curly brackets . . . 31
- 2.1.8 The semicolon . . . 32
- 2.1.9 How code is executed . . . 32

2.2 Arrays in Java . . . 32
- 2.2.1 Dimensions . . . 32
- 2.2.2 The square bracket notation . . . 32

2.3 Example: plotting the graph of a function . . . 33
- 2.3.1 A typical output . . . 33
- 2.3.2 Comments and imports . . . 33
- 2.3.3 The class declaration . . . 34
- 2.3.4 The `main` function . . . 35
- 2.3.5 Initialization . . . 35
- 2.3.6 Function tabulation . . . 36
- 2.3.7 Converting the function table to an array . . . 36
- 2.3.8 Printing the screen . . . 37
- 2.3.9 Compilation and running . . . 38

II Data structures **39**

3 Graphs **41**

3.1 Directed graphs . . . 41

3.1.1 Directed neighbourhoods 42
3.2 Undirected graphs 42
3.2.1 Complement graphs 42
3.2.2 Neighbourhoods 43
3.2.3 Graph isomorphism 43
3.2.4 Line graph 44
3.3 Subgraphs 44
3.3.1 Stable and clique subgraphs 45
3.3.2 Some applications 45
3.4 Connectivity 46
3.4.1 Simple paths 46
3.4.2 An alternative definition of paths and connectivity 47
3.4.3 Paths: not so simple 47
3.4.4 Strong connectivity 47
3.4.5 Cycles 47
3.4.6 An alternative definition of cycles 48
3.4.7 The cycle space 48
3.4.7.1 Cycle-edge incidence vectors 49
3.5 Basic operations on graphs 49
3.5.1 Addition and removal of vertices and edges 50
3.5.2 Contraction 50

4 Linear data structures 53
4.1 Arrays 53
4.1.1 Jagged arrays 53
4.1.1.1 Adjacency lists 54
4.1.2 Array operations 54
4.1.2.1 Size in $O(1)$ 55
4.1.2.2 Moving a subsequence 55
4.1.2.3 Removal and insertion 56
4.1.3 Complexity of an incomplete loop 56
4.1.3.1 Worst-case complexity 56

4.1.3.2 Average-case complexity 57

4.1.4 Limitations of the array structure 58

4.2 Lists 58

4.2.1 Singly-linked lists 58

4.2.2 Doubly-linked lists 58

4.2.2.1 The placeholder node 59

4.2.3 Lists modelled as graphs 59

4.2.4 List operations 59

4.2.4.1 Insertion 59

4.2.4.2 Removal 60

4.2.4.3 Find 60

4.2.4.4 Access 61

4.2.4.5 Other operations 61

4.2.5 Java implementation 61

4.2.5.1 The `Node` class 62

4.2.5.2 The `DLList` class 62

4.2.5.3 The `main` function 64

4.3 Queues 65

4.3.1 Circular arrays 65

4.3.1.1 Java implementation 66

4.3.2 What are queues used for? 67

4.4 Stacks 67

4.4.1 Using stacks for validating mathematical syntax 68

4.4.1.1 Java implementation 68

4.4.1.2 The `main` method for `bracketStack` 69

4.4.1.3 Sample output 70

4.4.2 Calling functions 71

4.4.2.1 Smashing the stack for fun and profit 71

4.5 Maps 71

4.5.1 Maps as parametrized interfaces 72

4.5.2 Example of map usage in Java 73

5 Hashing **75**

5.1 Do we really need it? 75

5.1.1 The phonebook example 76

5.1.2 Formal explanation 76

5.1.3 Applications of hashing to Java 77

5.2 The last nagging doubt 77

5.3 Java implementation 78

5.3.1 A hash table without collisions 78

5.3.1.1 Keys and records 78

5.3.1.2 The main class 78

5.3.1.3 The hash function 79

5.3.1.4 Main function 79

5.3.2 A hash table allowing for collisions 80

5.3.2.1 A Java implementation of a singly-linked list 80

5.3.2.2 The main class 82

5.3.2.3 Adding elements to the hash table 83

5.3.2.4 Finding elements in the hash table 83

5.3.2.5 Main function 83

6 Trees **87**

6.1 Definitions 88

6.1.1 Roots and direction 88

6.1.2 Leafs, depth and height 88

6.1.3 Spanning tree 88

6.1.4 Vertex labels 88

6.2 Basic properties 89

6.2.1 Number of edges 89

6.2.2 Connectivity 90

6.2.3 Acyclicity 90

6.2.4 Edge swapping operation 90

6.3 The number of labelled trees 90

6.3.1 Mapping trees to sequences 91

6.3.2 Mapping sequences to trees . . . 91
6.4 Applications . . . 92
6.4.1 Finding a basis of the cycle space . . . 92
6.4.2 Chemical trees . . . 92
6.4.3 Trees and languages . . . 93
6.4.3.1 Trees and recursion . . . 93
6.4.3.2 Syntax of formal languages . . . 94
6.4.3.2.1 Construction of valid sentences . . . 94
6.4.3.2.2 Recognition of valid sentences . . . 94
6.4.3.3 Semantics of formal languages . . . 95
6.4.3.4 Syntax of natural languages . . . 95
6.4.3.5 Semantics of natural languages . . . 96
6.4.4 Trees in networks . . . 97
6.4.4.1 Commodity networks . . . 97
6.4.4.2 Distance networks . . . 97

III Algorithms 99

7 Recursive algorithms 101
7.1 Motivations . . . 101
7.1.1 Proving program properties . . . 101
7.1.2 Expressing certain procedures naturally . . . 101
7.1.2.1 Encoding the tree . . . 102
7.1.2.2 A code with limited scope . . . 102
7.1.2.3 Algorithms and problems . . . 102
7.1.2.4 Recursion saves the day . . . 103
7.1.2.5 Back to iteration . . . 103
7.2 Iteration and recursion . . . 104
7.2.1 Terminating the recursion . . . 104
7.3 Listing permutations . . . 105
7.3.1 Some background material on permutations . . . 105
7.3.1.1 Product of permutations . . . 106

7.3.1.2 Group structure 106

7.3.1.3 Cycle notation 107

7.3.2 The inductive step 107

7.3.2.1 Generalizing the example to an integer n 108

7.3.2.2 The induction starts at 1 108

7.3.3 The algorithm 108

7.3.3.1 Data structures 109

7.3.4 Java implementation 109

7.3.4.1 Class structure 109

7.3.4.2 The `main` method 110

7.3.4.3 The `printList` method 111

7.3.4.4 The `orders` method 111

7.4 The Hanoi tower 112

7.4.1 Inductive step 113

7.4.2 Base case 113

7.4.3 Java implementation 113

7.5 Recursion in logic 114

7.5.1 Definitions 114

7.5.2 Gödel's theorem 114

7.5.3 The beautiful and easy part of the proof 115

7.5.4 The other part of the proof 115

7.5.5 A natural language interpretation 115

8 Graph searching and traversal 117

8.1 Graph scanning 117

8.1.1 The Graph Scanning algorithm 117

8.1.1.1 Correctness 118

8.1.1.2 Complexity 118

8.1.1.3 Connected components 118

8.1.1.4 The exploration tree 119

8.1.1.5 Choosing $v \in Q$ 119

8.2 Breadth-first search 120

8.2.1 Paths with fewest edges 120

8.2.2 History of the BFS 121

8.2.3 Looking for a good route in public transportation 121

8.3 Depth-first search 123

8.3.1 A recursive version of DFS 124

8.3.2 History of the DFS 125

8.3.3 Easy and difficult natural languages 125

8.4 Finding a spanning tree of minimum cost 127

8.4.1 Prim's algorithm: pseudocode 128

8.4.2 Complexity of Prim's algorithm 129

9 Problems and complexity **131**

9.1 Decision problems 131

9.2 Optimization problems 132

9.2.1 Relationship between decision and optimization 132

9.3 Algorithms 132

9.4 Complexity 132

9.5 Easy and difficult problems 133

9.5.1 Reductions 133

9.5.2 The new problem is easy 134

9.5.3 The new problem is as hard as another problem 134

9.5.4 **NP**-hardness and **NP**-completeness 134

9.5.5 The most celebrated open problem in computer science 135

9.5.6 The student's pitfall 135

9.6 Exact and heuristic algorithms 135

9.6.1 A heuristic method for MAX STABLE 135

10 Sorting **137**

10.1 The searching problem 137

10.2 Searching unsorted and sorted arrays 137

10.3 The sorting problem 138

10.3.1 Considerations on the complexity of SP 138

10.3.1.1 The best algorithm for a problem 138

- 10.3.1.2 The $\Omega(\cdot)$ and $\Theta(\cdot)$ notations . . . 138
- 10.3.2 Best-case complexity of SP . . . 138
  - 10.3.2.1 The sorting tree . . . 139
  - 10.3.2.2 Formalizing the idea . . . 139
- 10.4 Sorting algorithms . . . 140
  - 10.4.1 Selection sort . . . 140
  - 10.4.2 Insertion sort . . . 140
  - 10.4.3 Merge sort . . . 141
    - 10.4.3.1 Divide and conquer . . . 141
    - 10.4.3.2 Pseudocode . . . 141
    - 10.4.3.3 Merging two sorted sequences . . . 141
    - 10.4.3.4 Worst-case complexity . . . 142
  - 10.4.4 Quick sort . . . 142
    - 10.4.4.1 Pseudocode . . . 142
    - 10.4.4.2 Partition . . . 143
    - 10.4.4.3 Worst-case complexity . . . 144
    - 10.4.4.4 Average-case complexity . . . 144
- 10.5 Exact complexity of SP . . . 146
- 10.6 Two-way partitioning . . . 147
  - 10.6.1 A paradox? . . . 147

11 Searching **149**

- 11.1 Notation . . . 149
- 11.2 Binary search trees . . . 150
  - 11.2.1 BST `min` and `max` . . . 150
  - 11.2.2 BST `find` . . . 151
  - 11.2.3 BST `insert` . . . 152
  - 11.2.4 BST `delete` . . . 152
    - 11.2.4.1 Deleting a node with both subnodes . . . 153
    - 11.2.4.2 Putting it all together . . . 154
  - 11.2.5 Complexity . . . 154
- 11.3 AVL trees . . . 154

- 11.3.1 Balance-independent methods . . . 155
- 11.3.2 Balance-dependent methods . . . 155
  - 11.3.2.1 Tree rotation properties . . . 156
  - 11.3.2.2 The remaining cases . . . 157
- 11.4 Heaps . . . 159
  - 11.4.1 Priority queues . . . 159
  - 11.4.2 Heap properties . . . 159
    - 11.4.2.1 Insertion . . . 160
    - 11.4.2.2 Maximum . . . 161
    - 11.4.2.3 Popping the maximum . . . 161
    - 11.4.2.4 Initial heap construction . . . 161

12 Shortest paths 163

- 12.1 Basic literature . . . 163
  - 12.1.1 Problem variants . . . 163
  - 12.1.2 Algorithms . . . 164
- 12.2 Weight functions . . . 164
- 12.3 The shortest path tree . . . 165
- 12.4 Dijkstra's algorithm . . . 166
  - 12.4.1 Data structures . . . 166
  - 12.4.2 Reach, settle and relax . . . 166
  - 12.4.3 A simple implementation . . . 166
    - 12.4.3.1 Complexity . . . 167
    - 12.4.3.2 Correctness . . . 167
  - 12.4.4 A more refined implementation . . . 169
    - 12.4.4.1 Pseudocode . . . 169
    - 12.4.4.2 Complexity . . . 169
  - 12.4.5 The point-to-point SPP . . . 169
- 12.5 Floyd-Warshall algorithm . . . 170
  - 12.5.1 Data structures . . . 170
  - 12.5.2 Pseudocode . . . 170
  - 12.5.3 Negative cycles . . . 171

Part I

Preliminaries and reminders

Chapter 1

Computation

ABSTRACT. What is a computer and how we represent it formally: what can be computed and what cannot. Programming languages: what can be expressed and what cannot. Touching on Java. Efficient computation: problems, data structures, and algorithms. Worst-case complexity.

This introductory chapter briefly touches on some deep theoretical topics, most of which are presented in the INF423 course, as well as in many textbooks (e.g. [17]).

1.1 Computer hardware

A computer is a piece of machinery engineered for carrying out computations electronically. Most computers consist of a *Central Processing Unit* (CPU), some banks of *Random-Access Memory* (RAM), several *Input/Output* (IO) devices, that allow the communication with the user and the external world, as well as a motherboard that wires all the components together. At any given instant, the CPU has a *state* s out of a possible set S of states. With a given frequency f, the CPU reads and executes a new *instruction*, supplied by the user, which changes its state. According to the state it is in, the CPU may perform arithmetic or logical operations on data, read data from or write it to RAM, send data to or receive it from IO devices.

1.1.1 Programs

Instructions are normally supplied by the user as an ordered list, also called a *program*. The "default behaviour" of the CPU is to execute instructions in the user-provided sequence, unless the instruction itself explicitly tells the CPU to jump to a given point in the sequence. There are instructions that tell the CPU to test a given logical condition and act differently according to the results of the test.

We remark that, since a program is defined as an ordered list of instructions, and any sublist of a list is also a list, any "subprogram" is also a program, i.e. we might use the same term "program" to refer to certain subsets of instructions of a given program.

1.2 Computer model

The computer, as described in Sect. 1.1, was first conceived by Alan Turing in 1936 [23]. Turing's mathematical model of the computer is called *Turing Machine* (TM). A TM consists of an infinite tape, divided into a countably infinite number of cells, with a device (called *head*) that can read or write symbols out of a given alphabet A on each cell of the tape. According to the state $s \in S$ the TM is in, the head either reads, or writes, or moves its position along the tape. Any TM has a set of instructions that tell it how to change its state. Turing showed that there exist TMs which can simulate the behaviour of any other TM: such TMs are called *Universal Turing Machines* (UTM).

Turing's work spawned further research, from the 1950s onwards, aimed at simplifying the description of UTMs, involving scientists of the caliber of Shannon [21] and Minsky [16]. More recently, Rogozhin [19] described UTMs with low values of $(|S|, |A|)$, e.g. $(24, 2)$, $(10, 3)$, $(7, 4)$, $(5, 5)$, $(4, 6)$, $(3, 10)$, $(2, 18)$. It appears clear that there is a trade-off between number of states and number of symbols in the alphabet.

1.2.1 Models of computation

UTMs are not the only existing models of computation — several others exist, sometimes very different from each other (see e.g. the *Game of Life*, a model for bacteria diffusion [3]). Such models are said to be Turing-complete if they can simulate a UTM. "Simulating", in this context, means using a different model C of computation in order to mimick the behaviour of a UTM. To prove this, it suffices to show that every instruction, state and symbol of the UTM can be simulated by C. If the UTM can also simulate C, then C is said to be Turing-equivalent.

1.2.2 Church Hypothesis

The *Church Hypothesis* is the statement that every Turing-complete model of computation is also Turing-equivalent. So far, no Turing-complete model of computation was found to be more "powerful" than the UTM.

1.3 Languages

A *programming language* is a set of rules for formulating instructions of a computer program. A language, by itself, does not compute. Programs written in a given language do not compute either. The relevant model of computation is a computer running a program written in a given language. It makes sense, however, to question the "expression power" of a language, e.g. is it sufficiently expressive that it can write a program that, executed on a computer, simulates a UTM? If so, then the language is called *universal*. It was shown in [5] that, in order to be universal, a given language should be able to express concatenation, tests and loops.

The *concatenation* of two instructions I_1 and I_2 is the program $I_1; I_2$. By induction, the concatenation of two programs P_1 and P_2 is the program $P_1; P_2$. A *test* instruction modifies the sequence of the instructions in a program according to whether a given logical condition is true or false. A *loop* is an instruction that tells the CPU to repeat the execution of a given program.

It is important to keep in mind the distinction between the language and its underlying computing model. A computing model with no provision for simulating a memory device (be it a tape or RAM or otherwise), may very well execute a program written in a universal language, but the resulting computing model will fail to be Turing-complete.

1.3.1 Declarative languages

So far, we made the assumption that programs consist of instructions that tell a computer (be it hardware or theoretical) to act in certain ways. The class of languages for expressing such programs are called *imperative languages*. There is another class of programming languages, called *declarative languages*, which are designed to describe sets (e.g., the set of even numbers can be described as $\{n \in \mathbb{N} \mid n \bmod 2 = 0\}$). Java, C, C++, Basic, Fortran, Pascal are all imperative languages.

Insofar as a computer acts on input data to produce output data, it can be seen as a function. Since the formal definition of a function f is a set F of pairs (ι, ω) such that $f(\iota) = \omega$, we can describe a function by means of the set F. In this sense, in order to fully describe f, it would suffice to list a set of mathematical conditions that hold for any set V if and only if $V = F$. Declarative programming languages allow such descriptions. Prolog, LISP, CAML, AMPL are all declarative languages (most also integrate some imperative instructions, too). An interesting example of a declarative universal language is given by Diophantine equations, i.e. polynomial equations with integer coefficients, were the solutions are constrained to be integers [11]: the solution set of certain Diophantine equations define functions that turn out to describe the input/output pairs of a UTM.

1.3.2 Decidability

Functions $\mathbb{N} \to \mathbb{N}$ that represent TMs are called *computable*. Sets $V \subseteq \mathbb{N}$ that are domains (or co-domains) of computable functions are called *recursively enumerable*. If V and $\mathbb{N} \smallsetminus V$ are both recursively enumerable, then they are both also called *recursive* or *decidable*. Given a set $V \subseteq \mathbb{N}$ and an integer $v \in \mathbb{N}$, one may ask whether $v \in V$ or not. This is a fundamental decision problem in number theory. It turns out that V is recursively enumerable if and only if there is a program that answers YES when $v \in V$, but may answer wrongly or even fail to terminate when $v \notin V$; moreover, V is decidable if and only if there is a program that answers YES when $v \in V$ and NO when $v \notin V$. This provides a further link between imperative and declarative languages.

It should appear clear that recursively enumerable sets are of limited value as far as proving that an answer to a problem is correct: if the algorithm answers YES, it may be a true positive or a false negative, and there is no way to tell. Moreover, how is a user supposed to know whether a program fails to terminate or is simply taking a long time? Accordingly, we try to frame problems so that the solution set is decidable.

On the other hand, several interesting sets fail to be decidable. Take, for example, Hilbert's tenth problem: *determine whether or not there exists a mechanical procedure for solving a given Diophantine equation*. Given the extent of work on Diophantine equations carried out ever since Greek civilization, this is certainly an interesting problem. It remained open until Matyasevič answered in the negative [11], by proving that there were parametric Diophantine equations whose solution set spanned the totality of recursively enumerable sets. Since decidable sets are a proper subset of recursively enumerable sets, there clearly are Diophantine equations whose solution set is undecidable.

1.3.3 Java

Our programming language of choice is Java[1]. In Java, one can:

- initialize a variable:

```
int i;
```

---

[1] Its basic syntax was provided in previous courses, such as INF311 and INF321.

- assign a constant value to a variable:

  ```
  i = 3;
  ```

- define a function:

  ```
  int f(int s) {
    s = 1;
    return s;
  }
  ```

  e.g. the above defines the trivial function $f : \mathbb{N} \to \{1\}$ that maps every integer to 1

- test a condition:

  ```
  if (i < 3) {
    f(i);
  } else {
    i = 0;
  }
  ```

- loop indefinitely over a program:

  ```
  while(true) {
    f(i);
  }
  ```

The fundamental constructs above can be mixed in a number of syntactically correct ways in order to produce increasingly complex programs.

1.4 Efficiency

Although the questions "what can you compute with a given computer?" and "what can you write with a given language?" are certainly the most fundamental — and universality gives a theoretical measure of how far computers and languages can provide an answer to these questions — there remains the important question of efficiency. How efficiently can a language express a certain program, and how fast can a computer execute a given program? The answers to these questions largely define the science (and art) of algorithmics.

1.4.1 Structuring data

In the vast majority of real computers, RAM is organized as a long but finite list of *Binary digITs* (bits). In the jargon of TMs, the underlying alphabet is $\{0, 1\}$, and each cell of the tape can hold either symbol. Although this is sufficient to guarantee universality, it is cumbersome for people to program using only two symbols.

Accordingly, a *data structure* is a segmentation of memory that carries a certain meaning to the user. For example, people noticed that most integers arising in practice are in the range `int` $= \{-2^{31}, \ldots, 2^{31} - 1\}$. Integers in this range can be described by sequences of exactly 32 bits (i.e. 4 bytes[2]: 31 bits are used to store the absolute value, and the remaining bit is used for the sign). In TM jargon, this is akin to taking 32 binary cells on the tape and declaring them to simply be "one cell" which can hold any integer in the range `int`. This, by the way, also provides a very simple example of simulation of a TM with $|A| = 32$ by a TM with $|A| = 2$.

---

[2] One byte contains 8 bits.

In real computers, we are not bound to segmenting memory using only one data structure: a part of memory can hold `int`s, another can hold bits, and yet another can hold `double`s (a data structure for storing a certain subset of rational numbers). Moreover, most programming languages allow users to declare their own data structures by combining the elementary ones in different ways. In order to make it even easier to write programs with user-defined data structures, languages usually also allow for pairing such structures with dedicated user-defined functions that handle them appropriately.

In Java, user-defined data structures are called *objects*; the formal description of an object in terms of elementary data structures is a *class*. The data items occurring in classes are called *attributes*, and the functions items are called *methods*.

1.4.2 Problems

It should be clear that programs take data as input, manipulate these data, and produce data as output. Not all possible data are valid for input (think of a program that takes a 16-bit integer as input and is fed a 32-bit one instead — something like this made the Ariane 5 rocket explode!), and the set of output data is well-defined. A program P therefore defines a set $\mathcal{I}$ of possible inputs and a set $\mathcal{O}$ of possible outputs. Do $\mathcal{I}, \mathcal{O}$ uniquely define the program that produced them? The answer is no: take for example $\mathcal{I} = \mathcal{O} = \{n \in \texttt{int} \mid n \bmod 2 = 0 \wedge n \in [0, 10]\}$. This can be obtained with either of the following (different) programs:

```
/* code 1 */
int i = 0;
while(i < 10) {
  i += 2;
  System.out.println(i);
}

/* code 2 */
int i = 0;
while(i < 10) {
  if (i % 2 == 0) then
    System.out.println(j);
  }
  i++;
}
```

We formally define a *problem* to be a set of pairs (ι, ω). A problem is *decidable* if there is at least a program (or TM) P with input set $\mathcal{I}$ and output set $\mathcal{O}$ such that $\omega = P(\iota)$ for all $(\iota, \omega) \in \mathcal{I} \times \mathcal{O}$. A decidable problem can also be seen as the class of all programs that produce the same output $\omega \in \mathcal{O}$ on a given input $\iota \in \mathcal{I}$, and reject all input not in $\mathcal{I}$.

1.4.3 Algorithms

Loosely speaking, we use "algorithm" and "program" interchangeably; a "program" usually indicates actual code that can be compiled and executed by a computer, whereas an "algorithm" is an idealized model of a program. An algorithm might contain a statements in natural (rather than formal, see Sect. 6.4.3) language, or be written formally but for a machine that does not exist in practice. In general, "algorithm" has a theoretical connotation, whereas "program" has a practical one. More formally, an *algorithm* is any program solving a decidable problem.

1.4.3.1 Algorithmic complexity

As mentioned in 1.4.2, several algorithms may solve the same problem. This immediately raises an important question: *which one should we use?* In the clearest-cut case, there might be an algorithm which takes less time and memory: it might be difficult to find it, but the question has a well-defined answer. In most cases, there is a trade-off between time and space: some algorithms may take less time but more memory, and vice versa. We refer to time measures as *time complexity* and *space complexity*. Since RAM is generally less costly than time, apart from a special cases time complexity is used more often than space complexity. In the sequel, we almost always mention complexity to mean "time complexity".

As programs consist of sequences of basic instructions, each of which takes (almost) the same time to execute on the CPU, time complexity evaluations usually count the number of instructions to be executed before the program terminates. Space complexity evaluations count the number of bytes of RAM that the program allocates whilst running. These questions are, unfortunately, ill-defined: programs may behave well with a given input and badly with another. Accordingly, we consider three possibilities: *worst-case complexity*, *best-case complexity*, and *average-case complexity*.

Best-case complexity is not often used: an algorithm may be very fast on short or trivial input, but this says nothing about how the algorithm will perform in general. Worst-case complexity is more informative (as it gives a guarantee — if an algorithm is fast in the worst case, all the better for the general case!) and usually not too difficult to evaluate: this is why it is the most studied case. Average-case complexity is very informative, as it aims at the general case, but it is often difficult to compute. And unless we can also compute the variance attached to the average, it might be misleading.

Below, we shall therefore mostly concentrate on worst-case time complexity.

1.4.3.2 Worst-case time complexity calculations

We let P be a program, and t_P be the number of instructions that are executed in P. Notice that, because of tests and loops, this is different from the number of instructions actually written in the program. The following rules-of-thumb hold.

- Basic assignments, arithmetic/logical operations and tests all have $t_P = 1$.
- If P, Q are programs and $P; Q$ is their concatenation, then

$$t_{P;Q} = t_P + t_Q. \tag{1.1}$$

- If T is a logical test and P, Q are programs, then

$$t_{\texttt{if}(T)\,P\,\texttt{else}\,Q} = t_T + \max(t_P, t_Q). \tag{1.2}$$

  Notice that the use of the max operator implements the worst-case policy; average- and best-case would yield different formulæ.

The case of the loop is more complicated, as it depends on whether the body of the loop is independent on the termination condition or not. Consider the following algorithm P, where Q is a program.

```
int i = 0;
while (i < n) {
    Q();
    i = i + 1;
  }
```

In this algorithm, `Q()` does not depend on `i`, and therefore we have $t_P = t_{\mathtt{i=0}} + n(t_Q + t_{\mathtt{i<n}} + t_{\mathtt{i+1}} + t_{\mathtt{i=\cdot}}) + t_{\mathtt{i<n}} = 2 + n(t_Q + 3)$. The last test term refers to the test $\mathtt{i} < \mathtt{n}$ when `i` and `n` have the same value, which fails and allows the loop to terminate.

1.4.3.3 Complexity orders

Suppose we are able to determine that $t_Q = \frac{1}{2}n$. Then $t_P = \frac{1}{2}n^2 + 3n + 1$. Since we would like to know the behaviour of this algorithm in the worst-case, it is interesting to look at the asymptotic behaviour of $t_P(n)$ for $n \to \infty$. It is clear that the dominating term is $\frac{1}{2}n^2$; moreover, asymptotically, the $\frac{1}{2}$ constant is not so important. We simply say that t_P asymptotically behaves like n^2, and write it as $t_P \in O(n^2)$, or "t_P is $O(n^2)$".

In general, we say that a function $f(n)$ is *order of* $g(n)$ (and write it "$f(n)$ is $O(g(n))$") if:

$$\exists c > 0 \quad \exists n_0 \in \mathbb{N} \quad \forall n > n_0 \quad (f(n) \le c\, g(n)). \tag{1.3}$$

In words: there are positive real c and integer n_0 such that $c\, g(n)$ dominates $f(n)$ for all $n > n_0$. Table 1.1 lists some complexity orders for various functions of n. Evidently, an effort should be made to find the lowest possible worst-case order, i.e. although the function $2n + 1$ is certainly $O(n^4)$, it is much more informative to say that it is $O(n)$.

Functions	*Order*
$an + b$ with a, b constants	$O(n)$
polynomial of degree d' in n	$O(n^d)$ with $d \ge d'$
$n + \log n$	$O(n)$
$n + \sqrt{n}$	$O(n)$
$\log n + \sqrt{n}$	$O(\sqrt{n})$
$n \log n^3$	$O(n \log n)$
$\frac{an+b}{cn+d}$, a, b, c, d constants	$O(1)$

Table 1.1: Complexity orders of some functions of n.

We remark that if $t_P(n)$ is a constant (i.e. n does not appear in the expression for t_P), then it is $O(1)$; e.g. looping 10^{100} times over an $O(1)$ program is itself $O(1)$.

Chapter 2

Java basics

ABSTRACT. Short introduction to Java. Variables, Objects, Classes, Interfaces, and Data types. Functions, values and references. An example: plotting the graph of a mathematical function on the screen.

Each computer brand — or even model — is different. CPU design and capabilities evolve in time, as does the set of instructions they can understand. On the other hand, programmers spend a long time learning a programming language, and once they are proficient in one, they are unwilling to spend more time learning others. Thus, languages should not change even though the underlying computer that executes them will. This is attained in essentially two ways:

- updating the compiler or interpreter;
- simulating a “software computer” that does not change.

The first paradigm is by far the most common. Since the instructions that a CPU can actually execute are very different from the constructs of human-employed programming languages, these must be translated into executable code in order to be run. *Compilers* or *interpreters* are used for this task. A compiler translates a user program at once, and writes a file containing executable code. An interpreter translates the instructions of a user program one after the other, following the program flow, and executes each one immediately. Compiled languages are: C, C++, Pascal,, Fortran, (relatively) recent Basic dialects, and many others. Interpreted languages are: perl, python, early Basic dialects, and many others. Typically, compiled languages yield code that is faster to execute than interpreted languages; on the other hand, changing an interpreted program is easier than changing a compiled one.

Java, on the other hand, rests on a piece of software that simulates a virtual computer: in Java terms, this is a *Virtual Machine* (VM). The virtual machine can evolve in time to take advantage of technical progress in computer design, but always offers the same set of primitives to the programs it can execute. This also makes it easy to port Java applications to different platforms (e.g. Windows, MacOSX, Linux): it suffices to implement an appropriate VM.

2.1 Development of a Java program

A Java program is encoded in an ASCII text file — better use the first 128 characters and steer clear of accents. Open your favourite text editor and type:

```
/*
  Name: helloWorld.java
  Purpose: a "hello world" program in Java
  Author: Leo Liberti (from an idea by B. Kernighan)
  Source: Java
  History: 17/10/2011 work started
*/

class helloWorld {
    public static void main( String [ ] args ) {
        System.out.println("hello world");
    }
}
```

Save as `helloWorld.java`. Now open a terminal window, and type:

```
javac helloWorld.java
java helloWorld
```

The first command calls the Java compiler, which translates the text above into code that can be executed on the Java VM. The second command tells the Java VM to execute this code. As an effect of the execution, the computer should print `hello world` on the same terminal as the commands were issued.

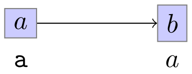
2.1.1 Variables

In Java, variable symbols can store values or addresses. Typically, variables having elementary data types such as `boolean`, `int`, `long`, `float`, `double`, `char` store values: if `int a` is a variable, then its value is stored at a certain address in memory. If `myClass` is a user-defined class, then the variable `myClass C` has a non-elementary data type: in this case, Java stores in `C` a memory address, which points to a part of memory which contains the actual value of `C`.

2.1.1.1 References

Variables containing addresses are also called *references*. Users need not concern themselves excessively with storage implementation details, aside for a few (but important) occurrences to do with copying data from a non-elementary data typed variable to another. Will Java copy the addresses or the values themselves? Copying addresses is known as *shallow copy*, whilst copying values is known as *deep copy*.

Consider a reference `a` whose value is an address a of a memory cell containing the value b. This is represented graphically as shown below.



2.1.2 Comments

A *comment* is simply text inserted in a program so as to remind human readers of the meaning of the surrounding instructions. Natural, rather than formal languages, are usually employed. Comments are supposed to make code clearer and easier to understand. A program with no comments will probably be unreadable; on the other hand, users will most likely ignore comments if every single line is commented.

In Java, comments can be single line or multi-line. A double slash ("`//`") means that the rest of the line, until its end, will contain a comment. A multi-line comment starts with "`/*`" and ends with "`*/`". An example of a multi-line comment is given in the `helloWorld` program.

Comments are ignored by the Java compiler, but some relevant information can be stored in comments, to be read by an appropriate *preprocessor* — this is software that interprets the program according to different rules, and is usually called prior to (or independently of) the compiler. This can be useful to automatically produce documentation for a given software, for example.

2.1.3 Classes

A *class* is the Java equivalent of a mathematical set defined by a property P, i.e. $\{x \mid P(x)\}$. In Java, the property P is a description of the *data type* of each piece of data x in the class, e.g. x might store an integer, a float and a string.

In Java, every program must contain at least a class. A *class* is the formal Java definition of a data structure. It usually contains a list of variable names (with associated data type) and a list of functions that determine how the values stored in the variables change. The class in the `helloWorld` program, called `helloWorld`, only contains one function called `main`.

A class is an entity which resides in the Java program. Once compiled and executed, a class, strictly speaking, does nothing. The Java VM, however, may create (either by default or because instructed to do so) *objects* of any given class. An object is therefore an instance of the class stored in memory. If we draw a parallel between Java and mathematical language, a class is to a set what an object is to an element. In other words, a class is a description of the data structure, whereas an object is an actual piece of data that is structured in memory according to the specification of its class. For example, the class

```
class IntPair {
  int first;
  int second;
}
```

defines a data structure that holds two integers between -2^{31} and $2^{31}-1$. An object `myIntPair` of this class, defined as:

```
  IntPair myIntPair = new IntPair;
```

holds a pair of integers in memory. The name `myIntPair` is arbitrary — the fact that it is similar to the class name is only supposed to help a reader identify the class directly from the object. In general, two different objects of the same class hold different data.

Class members, be they data (*attributes*) or functions (*methods*), can some specifiers, such as `public`, `static` and others. Public class members can be referenced from instructions outside the class. Static members are stored at a single address in memory, which means that all objects of the same class all share static data. By default, members are neither public nor static, which means that they can only be referenced from instructions inside the class, and each object of the class can refer to its own private copy of the data.

2.1.3.1 The `this` attribute

Suppose the class `myClass` has a method `myMethod`. Within the `myMethod` code, the Java keyword `this` is a reference to the object which will execute `myMethod`. This means that two objects of the same class have the same attribute `this`, but this will hold two different memory addresses depending on which object refers to it.

2.1.3.2 Inheritance

Just as sets can, in mathematics, be subsets of other sets, classes can be subclasses of other classes. A class `C` describing each of its objects as storing an integer, a float and a string might well be a subclass of another class `B` whose objects only store an integer and a float.

Inheritance is useful for several reasons. For example, if we need both `B` and `C` in our code and do not have inheritance, we must write

```
public class B {
  int i;
  float f;
}

public class C {
  int i;
  float f;
  String s;
}
```

Notice we are duplicating some code. If we need to change the `float` to a `double` later on, and still require `B` to be a subclass of `C`, we must remember to change the code in different places: since people forget such details, it will very likely give rise to a bug. Notice that a similar type of bug destroyed the Ariane 5 on its maiden flight. Inheritance helps avoid this issue. We define `C` as follows:

```
public class C extends B {
  String s;
}
```

In other words, we explicitly tell the compiler that there is a relationship between the two classes. This not only shortens the code, but decreases the chances of coding mistakes by delegating to the compiler the responsibility of checking that the relationship $\texttt{C} \subseteq \texttt{B}$ is maintained throughout the code.

2.1.3.3 Interfaces

An *interface* is a very special kind of class, whose purpose is that of enforcing conformance to a certain class structure. For example, in a Graphical User Interface (GUI) every window (including full program windows, dialog boxes and warning boxes) must conform to certain basic notions about windows: e.g., they possess a width, a height, a title and a frame, they have some standard buttons for minimizing, maximizing and closing, they have some standard pull-down menu, and they must remember their contents so that these can be re-drawn if another window is temporarily dragged over it. A programmer who designs a new type of window might be tempted to design a window of a different kind, say with no standard buttons. However, all other programs running on the GUI automatically assume that all windows have standard buttons, so they are free to call the associated code: in the long run, this might cause bugs and unforeseen behaviour. To avoid this, the compiler itself enforces conformance with a standard idea of window in the GUI by means of an interface: all classes defining a window must inherit from the window interface, and implement its functions as they see fit.

Remark that interfaces have no data and only contain the names, argument types and return types of the member functions. No object can be defined as member of an interface class only. By contrast, objects of different classes that both implement (i.e., inherit from) the same interface class can both be attributed the interface data type. Let us consider the example of a normal window and of a special type of window whose aspect ratio is always 3:2.

```
interface Window {
  int getWidth();
  int getHeight();
}

class MainWindow implements Window {
  int width;
  int height;
  public int getWidth() {
    return width;
  }
  public int getHeight() {
    return height;
  }
}

class FixedRatioWindow implements Window {
  int size;
  public int getWidth() {
    return 3*size;
  }
  public int getHeight() {
    return 2*size;
  }
}
```

In subsequent code, we might wish to loop over all windows. This is possible thanks to inheritance from the `Window` array.

```
  MainWindow mw = new MainWindow();
  FixedRatioWindow frw = new FixedRatioWindow();
  ArrayList<Window> a;
  // ...
  a.add(mw);
  a.add(frw);
  // now we can loop over the elements of a
```

Notice that `ArrayList<Window>` implements an array of interfaces (we shall discuss the funny angular brackets later, see Sect. 4.5.1).

2.1.4 Functions

In general, functions may take a list of arguments of given type, implement an algorithm with such arguments as input, and then optionally return a value of given type to the calling function. For example, the Java function `f` taking an integer input and returning its square is implemented as follows.

```
int f(int x) {
  return x*x;
}
```

2.1.4.1 Passing by reference or value

Consider the mathematical function $f(x) = x^2$, and the possible C++ implementation:

```
dataType f(dataType x) {
  x *= x;
  return x;
}
```

(we remark this is a valid construct in Java too, but we refer to C++ here for technical reasons). The `dataType` keyword is simply the data type of the argument `x` and of the return value of `f`. It might describe an integer or a floating-point number.

Now consider a calling function:

```
void g() {
  dataType x = 2;
  dataType y = f(x);
  cout << x << " + **2 = " << y << endl; // print string on screen
}
```

and try and imagine what the printed output will be like. Obviously, `y` will take value 4. But what value will `x` take? The definition of `f` changes the value of `x`, but will this change be remembered in the calling function?

The answer depends on whether `f` has access to the value of `x`, or to the address where `x` is stored in memory. In the first case, this value will be stored by `f` in a new memory area: the `x` in `g` and the `x` in `f` are stored in two distinct memory cells, whose values can be changed independently. in the second case, if `f` changes the value of `x`, `g` will retrieve the changed value, because `x` in `f` and `x` in `g` are stored at the same memory address. In the first case, `x` is *passed by value*, and in the second, `x` is *passed by reference*.

In Java, passing by value or by reference is not a user choice. Elementary data types (`boolean`, `int`, `long`, `float`, `double`, `char`) are passed by value, and all other data types (including user-defined classes) are passed by reference.

2.1.4.2 The `main` function

The function `main` takes as argument an array called `args`, and returns no data — this is the meaning of the `void` right before the function declaration. In the case of the `helloWorld` program, all it does is to print out to the system screen the string “hello world”.

2.1.4.3 Specifiers

Some details about how functions are translated to executable code by the Java compiler can be influenced by code *specifiers*.

The `main` function of the `helloWorld` program, for example, is specified to be `public` and `static`. The first specifier, `public`, states that this function can be called from other functions, even outside the class (normally, functions in a class can only be called by other functions in the same class). The second specifier, `static`, tells the compiler that any object of the class `helloWorld` will share a single copy of the `main` function: the function code will be stored at a precise address in memory, and this address will be stored in every object of the class `helloWorld`. Functions and variables that are declared `static` serve a purpose of sharing data between different objects of the same class.

2.1.5 Data types

A class defines the type of data it encodes: if an object is a piece of data, the class it belongs to is the *data type*. In Java, we distinguish a small set of data types, which includes `boolean`, `int`, `long`, `float`, `double` and `char`: these are called *elementary*. Elementary data types do not correspond to any defined class — rather, these types are hard-coded in the compiler. This makes a difference as far as functions are concerned. The instruction

```
void f(int a);
```

will define a function called `f` which takes as input an `int` called `a`. If the function `f` changes the value stored in `a`, this has no effect on `a` outside the function:

```
  int a = 2;
  f(a);
  System.out.println(a);
  // ...
  void f(int a) {
    a = 1;
  }
```

2.1.6 The dot operator

For Java instructions to refer to the member `first` of the object `myIntPair`, the dot operator is used, e.g.:

```
  myIntPair.first = 1;
```

With this in mind, it is easy to interpret the instruction

```
  System.out.println("hello world");
```

as a chain of embedded objects: it is a call to the function `println`, which is in the object `out`, which is itself one of the data in the object `System`.

2.1.7 The curly brackets

In Java (and also in C and C++), curly brackets, (also known as braces), are used to delimit the scope of a set of instructions. Accordingly, they mark the beginning and end of a class, function, but also of `if` blocks, and `while` and `for` loops.

2.1.8 The semicolon

Every basic instruction in Java is terminated by a semicolon. Notice that a basic instruction does not necessarily correspond to a line. Class or function definitions are not considered basic instructions, insofar as they are delimited by braces rather than ended by a semicolon.

2.1.9 How code is executed

A program can include several classes, defined in different files. However, for a program to yield executable code, there must be a *point of entry*, i.e. a first function that the Operating System (OS) knows it can call. This function can then call all the other functions in the program. In Java, this first function is called `main`, it must be defined as `static` and `void`, it takes as input the list of command line arguments passed by the user to the program via the shell prompt, and must be a `public` class member. Moreover, the name of the class containing `main` should have the same name as the Java file it is stored in (see the `helloWorld` example).

2.2 Arrays in Java

An *array* represents a list of objects of the same class. An example of an array of three `int` having value $2, 1, -1$ is shown below.

2	1	-1

The fact that the objects come one after the other is meant to suggest that they are stored in memory at contiguous addresses. This may be true or not, depending on the actual array implementation. It is usually true to a degree: an array might well be implemented as a set of distinct memory segments, each of which consists of contiguous addresses.

2.2.1 Dimensions

Arrays can be *linear* or *multidimensional.* Mathematically speaking, a linear array is similar to a vector, a two-dimensional array to a matrix, and a multidimensional array to a tensor.

2.2.2 The square bracket notation

In Java, arrays are declared using square brackets, in one of the two ways below.

```
int[] myArray;
int myArray[];
```

Declaring an array does not make it usable: first, we must allocate some memory to it.

```
myArray = new int[4]; // reserves enough memory to hold 4 int
```

It is common to combine the declaration and the memory allocation in the same instruction (both alternatives will work).

```
int[] myArray = new int[4];
int myArray[] = new int[4];
```

Multidimentional arrays get as many square bracket pairs as there are dimensions. The following defines a $(4 \times 3 \times 2)$ 3-dimensional array of `int`.

```
int[][] x = new int[4][3][2];
```

In Java, the number of components in each dimension need not be a constant.

```
int n = 2;
int m = 3;
int[][][] x = new int[4][m][n];
```

2.3 Example: plotting the graph of a function

In this section we shall present a simple worked-out Java example, consisting entirely of static data, that plots a function $y = f(x)$ of one variable on a text console screen. This program consists of a single class, called `functionPlot`, and resides in a single file called `functionPlot.java`. The class is declared static and contains a `main` function, as detailed above. The `main` function performs the following tasks:

1. initialize some data, such as the ranges $[x^L, x^U]$ where we wish to tabulate $f(x)$, the number n of function table entries, as well as the size of the "text screen" (number of rows and columns);
2. compute two vectors: $(x_1, \ldots, x_n)$ such that $x_k \in [x^L, x^U]$ for all $k \leq n$, and $(y_1, \ldots, y_n)$ such that $y_k = f(x_k)$ for all $k \leq n$;
3. fill an integer array `screen` with zeroes, apart from those entries (i, j) (set to 1) that best approximate the pairs (x_k, y_k);
4. print the array `screen` on the text console.

2.3.1 A typical output

A typical output with 20 rows, 75 columns, and $f(x) = \frac{1}{4}x + \sin(x)$ in the range $[-10, 10]$ is shown in Fig. 2.1.

2.3.2 Comments and imports

The `filePlot.java` file starts with the usual comments bearing some minimal information about the program.

```
/*
  Name: functionPlot.java
  Purpose: plotting functions in ASCII
  Author: Leo Liberti
  Source: Java
  History: 120615 work started
*/
```

```
liberti@styx$ java functionPlot
                                                                 **    **
                                                                *       **
                                                               *          *
                                                              *
                                           ****              *
                                          **  ***           *
                                        **      **         *
                                       **         **      *
                                      **            ******
                     *                *               **
                  *** ***            *
                 **     ***         *
                **        **      **
               *            **   **
              *               ***
             *
**          *
  **       *
   **    **
     *****
```

Figure 2.1: A typical output from `functionPlot`.

Next, we *import* some standard Java classes.

```
import java.io.*;
import java.util.*;
import java.lang.*;
```

The `import` command tells the compiler to read some other Java code, which might include declarations and definitions of names occurring in the program. For example, we shall make use of mathematical functions `Math.sin` and `Math.rint`, as well as the print functions `System.out.print` and `System.out.println`. We shall not explicitly declare these functions, but rather instruct the compiler to look for them in some standard libraries.

2.3.3 The class declaration

We present the class structure next. We remark that the following class contains some function *declarations* (specifiers, return type, name, argument types) without the corresponding *definition* (the function code, contained in braces); accordingly, the following code cannot be compiled "as is": all function declarations need to be followed by the corresponding definitions.

```
class functionPlot {
    //// Private data
    static int steps;    // number of function evaluations
    static double [] x;  // x coordinate values
    static double xL;    // lower bound for x coordinate values
    static double xU;    // upper bound for x coordinate values
    static double [] y;  // y coordinate values
    static double yL;    // lower bound for y coordinate values
    static double yU;    // upper bound for y coordinate values
    static int rows;     // number of text rows taken by plot
    static int columns;  // number of text columns taken by plot
    static int[][] screen;
    //// Public functions
    // this defines the function to be plotted
    public static double theFunction(double z);
    // initialize some values
    public static void initialize();
    // compute the x-y table
    public static void functionTable();
    // compute the minimum value of the y range
    public static double yMin();
    // compute the maximum value of the y range
    public static double yMax();
    // fill the screen array cells corresponding to x/y pairs with 1's
    public static void tableScreen();
    // loop over the screen array and either print a '*' (1) or a space (0)
    public static void printScreen();
    // this is the program's point of entry
    public static void main(String [] args);
}
```

All the data in the `functionPlot` class is declared `static`. This implies that all data is shared among all functions in the class. The first section of the class declares a set of (private) attributes which can only be accessed by functions within the class. There follows a section containing the (public) declarations of all class functions. Notice that the last function `main` is the point of entry of the VM when it executes the code. Every line is commented for clarity.

2.3.4 The `main` function

The `main` function simply calls four other functions in order: data initialization, function tabulation, filling of the `screen` array, and printing of the `screen` array on the console screen.

```
    public static void main(String [] args) {
        initialize();
        functionTable();
        tableScreen();
        printScreen();
    }
```

2.3.5 Initialization

The `initialize` function simply assigns some user-defined constants to some parameter variables. Users can change the behaviour of the program in certain ways by changing these parameters, re-compiling and re-executing the program.

```
    public static void initialize() {
        // user: set values for steps, x range, rows/columns for plot
        steps = 150;   // 150 entries in function table
        xL = -10;      // lower bound for x range
        xU = 10;       // upper bound for x range
        columns = 75;  // number of columns on the console screen
        rows = 20;     // number of rows on the console screen
        // user: do not change anything beyond this point
        x = new double[steps];  // the x vector
        y = new double[steps];  // the y vector
    }
```

2.3.6 Function tabulation

Tabulating a function f of one variable means computing a set of pairs (x_k, y_k) such that $y_i = f(x_k)$, for $k \leq n$ where $n \in \mathbb{N}$ is given. Accordingly, we set up two linear arrays `x`, `y` to hold `steps` values each.

```
    public static void functionTable() {
        double theStep = (xU - xL) / steps;
        x[0] = xL;
        for(int k = 1; k < steps; k++) {
            x[k] = x[k-1] + theStep;
            y[k] = theFunction(x[k]);
        }
    }
```

The function `functionTable` calls another function, `theFunction`, which is simply a user-changeable implementation of the mathematical function $f(x)$. For example, if $f(x) = \frac{1}{4}x + \sin x$, we have

```
    public static double theFunction(double z) {
        // user: change the mathematical function here at leisure
        return 0.25*z + Math.sin(z);
    }
```

Notice here that we use the symbol `z` instead of x because we already used a static variable `x` to denote the array storing the first components of the function table pairs. Since `x` is a static variable, every function in the class (including `theFunction` can read it and change it. Had we used `double x` instead of `double z` here, we would have *shadowed* the symbol `x` that denotes the array, rendering it inaccessible within this function. Also notice the library function `sin`, which is a member function of the class `Math`.

2.3.7 Converting the function table to an array

We pave the way for printing to the console screen. We model this screen by means of a two-dimensional array `screen`, indexed on rows and columns. Initially, we fill `screen` with zeroes. Later on, we loop over the function table pairs, and we change `screen`'s (i, j)-th entry to a one whenever there is a pair (x_k, y_k) such that (i, j) is the best integer approximation of (x_k, y_k) (after appropriate translation and scaling). More precisely, we let:

$$\begin{aligned} j &= \left\lfloor \frac{x_k - x^L}{x^U - x^L}\sigma + \frac{1}{2} \right\rfloor \\ i &= \left\lfloor \frac{y_k - y^L}{y^U - y^L}\rho + \frac{1}{2} \right\rfloor, \end{aligned}$$

where σ is the number of columns (`columns` in the code) and ρ the number of rows (`rows` in the code. Adding $\frac{1}{2}$ to a real number and taking its floor is the same as rounding it to the closest integer.

```
    public static void tableScreen() {
        int i;
        int j;
        screen = new int [rows][columns];
        for(i = 0; i < rows; i++) {
            for(j = 0; j < columns; j++) {
                 screen[i][j] = 0;
            }
        }
        yL = yMin();
        yU = yMax();
        for(int k = 0; k < steps; k++) {
            j = (int) Math.rint(((x[k] - xL) / (xU - xL)) * columns);
            i = (int) Math.rint(((y[k] - yL) / (yU - yL)) * rows);
            if (i < rows && j < columns) {
                 screen[rows - (i + 1)][j] = 1;
            }
        }
    }
```

The above function calls two other functions, `yMin` and `yMax`, that compute the minimum and maximum of the y range, and are defined as follows.

```
    public static double yMin() {
        double theMin = 1e30; // infinity
        for(int k = 0; k < steps; k++) {
            if (y[k] < theMin) {
                theMin = y[k];
            }
        }
        return theMin;
    }
    public static double yMax() {
        double theMax = -1e30; // -infinity
        for(int k = 0; k < steps; k++) {
            if (y[k] > theMax) {
                theMax = y[k];
            }
        }
        return theMax;
    }
```

2.3.8 Printing the screen

We now come to the function that copies the arrays `screen` onto the console screen. We interpret it so that every time `screen` contains a 1, a ‘`*`’ is printed on the screen, while all zeroes are printed as spaces.

```
    public static void printScreen() {
        for(int i = 0; i < rows; i++) {
            for(int j = 0; j < columns; j++) {
                if (screen[i][j] == 1) {
                    System.out.print("*");
                } else {
                    System.out.print(" ");
                }
            }
            System.out.print("\n");
        }
    }
```

2.3.9 Compilation and running

As for `helloWorld`, the program `functionPlot` can be compiled and executed very simply as follows.

```
javac functionPlot.java
java functionPlot
```

The output of `functionPlot` with the user-defined parameters is given in Sect. 2.3.1

2.3.1 Exercise
Extend the `functionPlot` *class so that it also prints the* x *and* y *axes.*

Part II

Data structures

Chapter 3

Graphs

ABSTRACT. Directed and undirected graphs, neighbourhoods, complements, line graphs, graph isomorphism. Subgraphs, stables and cliques. Connectivity: paths and cycles. Basic operations on graphs.

Data structures consist of pieces of data as well as relations between and among them. We are going to formalize this concept mathematically by means of graphs. Graphs are also the appropriate mathematical model for networks, be they transportation, communication, power, social or other types of networks. Learning the ropes of graph theory will therefore serve a double purpose.

3.1 Directed graphs

Formally, a *relation* on a set V is a subset of the Cartesian product $V \times V$ of all ordered pairs of elements of V. Accordingly, we define a *directed graph* (or *digraph*) as a pair $G = (V, A)$ where V is the set of *nodes* (or *vertices*) and A, a subset of $V \times V$, is the set of *arcs* of the graph. Given a digraph G, we sometimes denote its node set by $V(G)$ and arc set by $A(G)$. A reflexive relation pair (v, v) is a special type of arc called a *loop*, see e.g. the loops $(1, 1)$ and $(2, 2)$ in the leftmost graph of Fig. 3.1. A *multidigraph* associates a positive integer to each arc called its *multiplicity.* An arc with multiplicity higher than one is a *multiple arc.* A digraph is *simple* if it has no loops or multiple arcs. A digraph is *empty* if its arc set is empty (e.g. second digraph from the left in Fig. 3.1). A digraph is *bipartite* if its node set V can be partitioned in two sets U, W such that every arc $(u, w) \in A$ has $u \in U$ and $w \in W$ (e.g. third digraph from the left in Fig. 3.1). A digraph is *complete* if its arc set is $V \times V$ (e.g. last digraph from the left in Fig. 3.1). Complete digraphs are also called *directed cliques.*

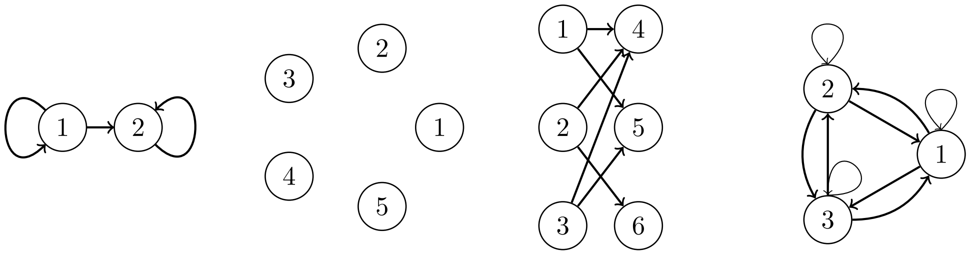


Figure 3.1: Examples of digraphs.

3.1.1 Directed neighbourhoods

Given a digraph $G = (V, A)$ and a node $u \in V$, the node set $N^+_G(u) = \{v \in V \mid (u, v) \in A\}$ is called the *outgoing neighbourhood* of u with respect to G. The node set $N^-_G(u) = \{v \in V \mid (v, u) \in A\}$ is called the *incoming neighbourhood* of u with respect to G (see Fig. 3.2). The *outdegree* of a node $v \in V$ is the cardinality of its outgoing neighbourhood, and similarly, the *indegree* of $v \in V$ is the cardinality of its incoming neighbourhood. In Fig 3.2, both the indegree and the outdegree of node 7 are equal to 3. For $u \in V$, the arc set $\delta^+_G(u) = \{(v, w) \in A \mid u = v\}$ is called *outgoing star* (or *outstar*) of u with respect to G, and the arc set $\delta^-_G(u) = \{(w, v) \in A \mid u = v\}$ is called *incoming star* (or *instar*) of u with respect to G. If there is no ambiguity we dispense with the index G.

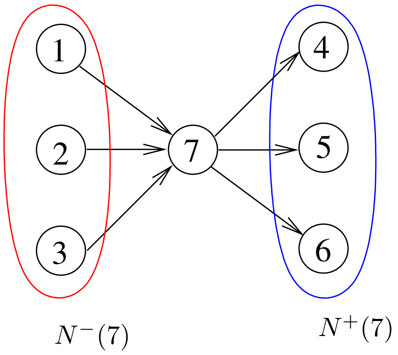


Figure 3.2: Incoming and outgoing neighbourhoods of node 7: $N^-(7) = \{1, 2, 3\}$ and $N^+(7) = \{4, 5, 6\}$.

3.2 Undirected graphs

Essentially, a graph is like a digraph where all arrows are replaced by segments. More formally, a *graph* is a pair $G = (V, E)$ where V is a set of vertices and E is a set of *edges* $\{u, v\}$ where $u, v \in V$. As for digraphs, for a graph G we sometimes denote by $V(G)$ its vertex set and by $E(G)$ its edge set. Whenever $u = v$ then $\{u, v\} = \{u\} = \{v\}$ is called a loop. In *multigraphs*, a positive integer called multiplicity is assigned to each edge; edges with multiplicity higher than one are called multiple (or parallel) edges (see Fig. 3.3). A graph without loops or multiple edges is called *simple*. Given a digraph, its *underlying graph* replaces every arc (u, v) with the corresponding edge $\{u, v\}$ (see Fig. 3.4). A graph is *empty* if its edge set is empty. A graph is *complete* (or a *clique*) if every possible edge $\{u, v\}$ is in E for all $u \neq v \in V$. Notice that complete graphs are simple. A graph $G = (V, E)$ is *bipartite* if V can be partitioned in two sets U, W such that every edge $\{u, w\} \in E$ has $u \in U$ and $w \in W$.

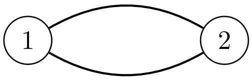


Figure 3.3: A multigraph with two parallel edges.

3.2.1 Complement graphs

Given a graph $G = (V, E)$, consider the clique $K(V)$ on V with edge set E_K consisting of all possible vertex pairs. The graph $\bar{G} = (V, \bar{E})$, where $\bar{E} = E_K \smallsetminus E$ is the *complement graph* of G (Fig. 3.5).

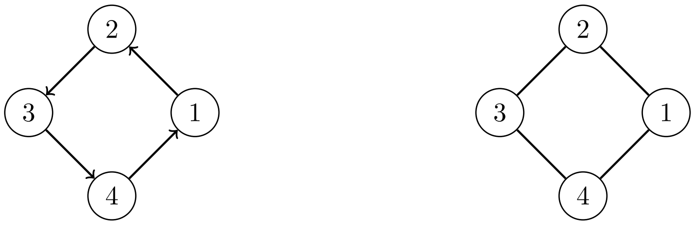


Figure 3.4: A digraph and its underlying graph.

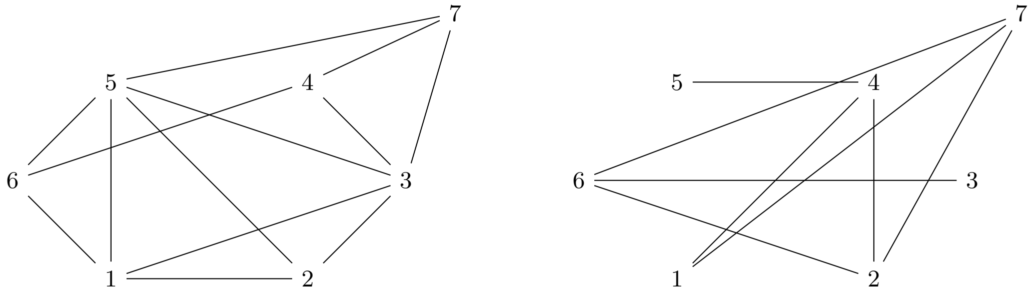


Figure 3.5: A graph and its complement.

3.2.1 Exercise
Propose an $O(n+m)$ *algorithm for constructing the complement graph of a given graph.*

3.2.2 Neighbourhoods

Given a graph $G = (V, E)$ and a vertex $v \in V$, we let $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$ be the *neighbourhood* of v and $\delta_G(v) = \{\{u, w\} \in E \mid u = v\}$ be the *star* of v in G. If there is no ambiguity, we dispense with the index G.

3.2.3 Graph isomorphism

Two graphs $G = (V, E)$ and $G' = (V, E')$ on the same vertex set V are *isomorphic* if there is a bijection $\pi : V \to V$ (also called an *automorphism* on V) such that:

$$\forall \{u, v\} \in E \ (\{\pi(u), \pi(v)\} \in E').$$

In other words, changing the vertex labels does not change the graph structure (see Fig. 3.6). Since V is taken to be finite, π is also a permutation of V (see Sect. 7.3). We denote the application of π to G by πG. If $\pi G = G$, then π is an *graph automorphism* of G.

3.2.2 Exercise
Verify that $\pi = (1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 1)$ *is not an automorphism on the graph on the left of Fig. 3.6.*

The set of all graph automorphisms of a graph G forms a *group* with respect to the composition of bijections.

3.2.3 Exercise
Verify that both $(1 \to 5)$ *is a graph automorphism of the graph on the left of Fig. 3.6. Show that this*

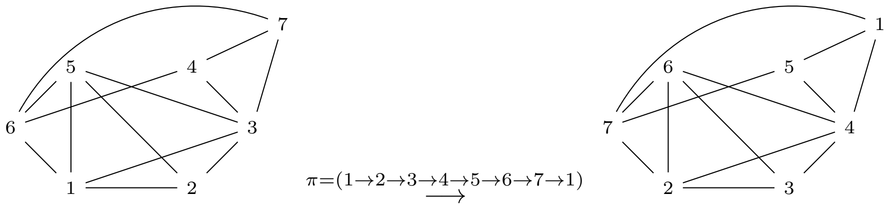


Figure 3.6: The left graph is isomorphic to the right graph.

graph has four automorphisms (including the identity, which fixes every $v \in V$), and, if you know what it means, that the structure of its group is $C_2 \times C_2$.

3.2.4 Line graph

Given a graph $G = (V, E)$ where $E = \{e_1, \ldots, e_m\}$, the *line graph* of G is the graph

$$L(G) = (E, \{\{e_i, e_j\} \mid e_i \cap e_j \neq \varnothing\})$$

where each edge of G is a vertex in $L(G)$, and edges of $L(G)$ are pairs of edges of G incident to the same vertex (see Fig. 3.7).

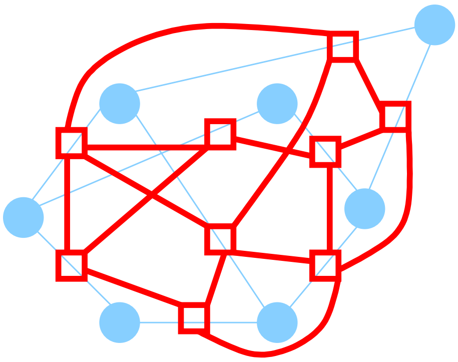


Figure 3.7: A graph G and its line graph $L(G)$ (thick lines).

3.2.4 Exercise
Prove that the degree $|N_{L(G)}(e)|$ of a vertex $e = \{u, v\}$ of $L(G)$ is $|N_G(u)| + |N_G(v)| - 2$.

3.2.5 Exercise
Give an algorithm for constructing $L(G)$ given G.

3.3 Subgraphs

Given a graph (or digraph) $G = (V, E)$, a *subgraph* is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$. A subgraph H is *induced* if, for all $u, v \in U$ such that $\{u, v\} \in E$, we have $\{u, v\} \in F$ as well (see Fig. 3.8). If $G = (V, E)$ is a given graph (or digraph) and $U \subseteq V$, we denote by $E[U]$ the subset of edges induced by U, i.e. the set of all edges in E between vertices in U. Thus, if $H = (U, F)$ is an induced subgraph of G, we can also denote F by $E[U]$. This notation is also extended to the whole subgraph: H can be denoted by $G[U]$.

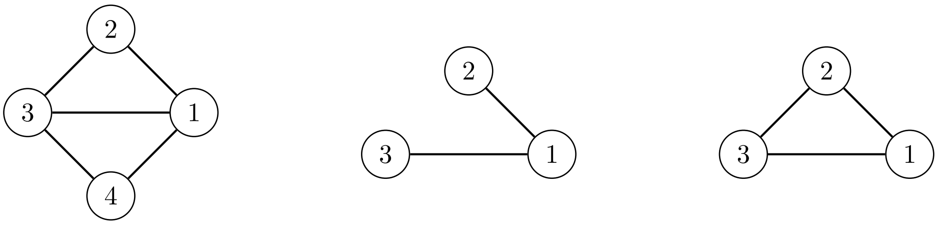


Figure 3.8: A graph, a (non-induced) subgraph and an induced subgraph.

3.3.1 Stable and clique subgraphs

An induced subgraph $H = (U, F)$ of G is a clique if it is complete, and a *stable* if it its edge set is empty (see Fig. 3.9). We also indicate stables by their vertex set only, since their edge set is empty, thus U is a stable (or *stable set*, or *independent set*) if $G[U]$ is a stable.

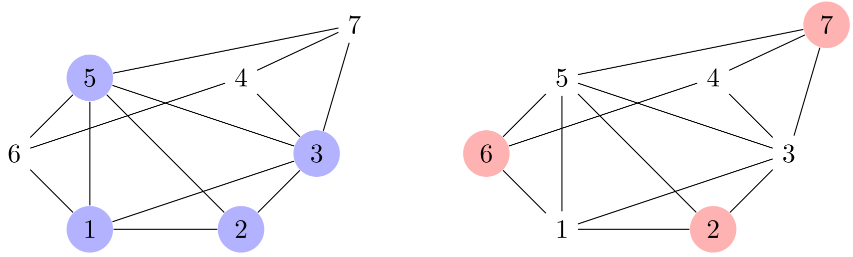


Figure 3.9: A clique (left) and a stable (right) in G.

3.3.2 Some applications

Many combinatorial problems consist of looking for a subgraph (be it induced or not) of a given graph with certain properties. The SUBGRAPH ISOMORPHISM PROBLEM (SIP), for example, is as follows. Given two graphs G, H, does G contain a subgraph which is isomorphic to H (see Fig. 3.10)? The SIP has applications in mining molecule databases, e.g. when trying to determine whether a given protein has a side-chain with a particular shape but possibly different atoms.

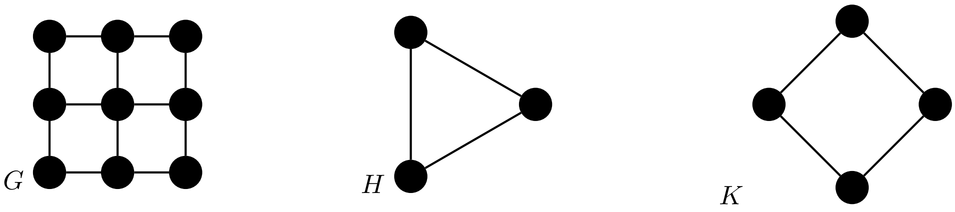


Figure 3.10: H is isomorphic to no subgraph of G, but K obviously is.

The DENSEST SUBGRAPH PROBLEM (DSP) looks for a subgraph of maximum density in a given graph (see Fig. 3.11); the density of a graph $G = (V, E)$ is defined as $\frac{|E|}{|V|}$. The DSP has applications in clustering.

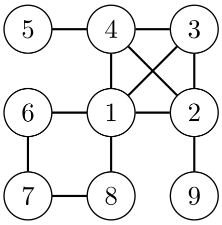


Figure 3.11: The densest subgraph of this graph is induced by $\{1, 2, 3, 4\}$.

3.4 Connectivity

We can extend the concept of neighbourhood to sets of vertices: let $G = (V, E)$ be a graph, and $U \subseteq V$. Then the neighbourhood of U in G is

$$N_G(U) = \bigcup_{u \in U} N_G(u),$$

i.e. the set of vertices adjacent to vertices in U. The *cutset* of U in G is

$$\delta_G(U) = \bigcup_{u \in U} \delta_G(u),$$

i.e. the set of edges incident to vertices in U. A cutset is *trivial* if $U = \varnothing$ or $U = V$. These definitions all extend to the case of digraph in a natural way. Cutsets allow us to explain connectivity in purely topological terms: a graph $G = (V, E)$ is *connected* if no nontrivial cutset is empty (see Fig. 3.12).

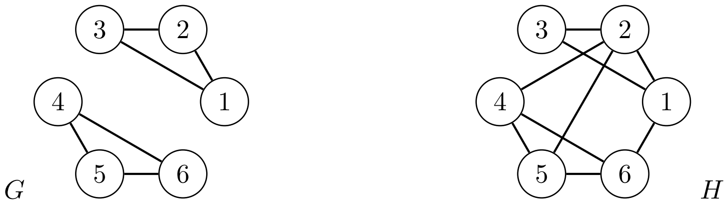


Figure 3.12: G is not connected: the cutset defined by $\{1, 2, 3\}$ is nontrivial and empty; H, on the other hand, is connected.

A digraph is connected if its underlying graph is; however, this notion of connectivity is not often used for digraphs. The appropriate connectivity notion in digraphs is called “strong connectivity” and will be discussed later.

The concept of connectivity arises in networks. In power networks, for example, blackouts may arise in large geographical areas when a part of the network is disconnected. Power networks are robust to such events when all cutsets have large cardinality.

3.4.1 Simple paths

A graph $G = (V, E)$ is a *simple path* if is it connected and each vertex has degree at most 2 (see Fig. 3.13).

Paths have multiple applications in any number of engineering, scientific and even mathematical problems. Shortest paths are often used in routing materials in transportation networks; robust paths

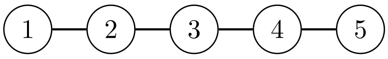


Figure 3.13: A simple path. Vertices 1,5 have unit degrees, while 2,3,4 have degree two.

are used in communication or power networks. The SHORTEST PATH PROBLEM (SPP) (see Chapter 12) often arises as a sub-problem in more complicated problems.

3.4.2 An alternative definition of paths and connectivity

Most textbooks on graph theory introduce paths before connectivity, defining the latter in terms of the former: a path with endpoints v_1, v_n is a graph $G = (V, E)$, where V is ordered as $(v_1, v_2, \ldots, v_n)$, such that for all $i < n$, $\{v_i, v_{i+1}\} \in E$. Moreover, a graph H is connected if for all $u \neq v$ in $V(H)$ there is a path in H with endpoints u, v. This, however, requires the concept of *order* to be used to introduce connectivity. Employing cutsets defines connectivity in a purely graph-theoretical way.

3.4.1 Exercise
Prove that the two definitions of graph connectivity are equivalent.

3.4.3 Paths: not so simple

In the previous sections we defined "simple paths" as well as "paths". What is the difference? A simple path is certainly a path (prove this), but the converse may not hold. Consider the path $(v_1, v_2, v_3, v_4, v_5, v_3)$ shown in Fig. 3.14. This graph is certainly a path, since it is defined by the order $v_1, v_2, v_3, v_4, v_5, v_3$, but since v_3 is repeated in the order, the path fails to be simple, as vertex 3 has degree three. In general, paths that "cross themselves" or have loops or traverse edges more than once (thus yielding a multigraph — can you see why this is so?) are not simple. Paths are also known as *walks*, or *trails* if all traversed edges are distinct.

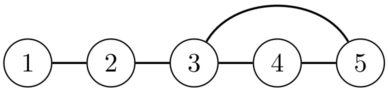


Figure 3.14: A non-simple path.

3.4.4 Strong connectivity

Walks (or trails) can be easily adapted to the case of digraphs: in this setting, a walk is an alternating sequence of nodes and arcs, beginning and ending with a node. A digraph G is *strongly connected* if for each ordered pair (u, v) of distinct nodes of $V(G)$ there is a walk in G starting at u and ending at v (see Fig. 3.15).

3.4.5 Cycles

A *simple cycle* in a graph is a path where all vertices have degree 2. A *cycle* in a graph is a subgraph where all vertices have even degree (see Fig. 3.16).

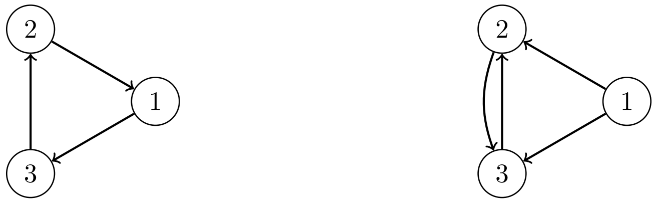


Figure 3.15: On the left, a strongly connected digraph. The graph on the right is not strongly connected, as there is no walk from vertex 3 to vertex 1.

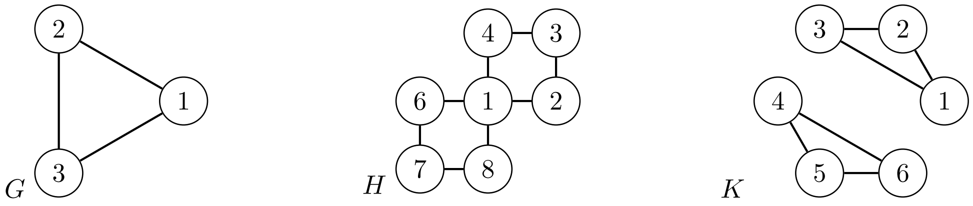


Figure 3.16: G is a simple cycle, H, K are cycles that are not simple.

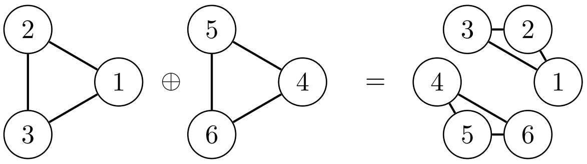
Computations concerning cycles in graphs often arise as sub-steps of certain graph-related algorithms (see e.g. Sect. 12.5.3). Cycles are often used as a proof that networks are robust to failures: if two nodes u, v of a network are involved in a cycle, then there must be two different paths from u to v: in case one fails, the other provides connectivity.

3.4.6 An alternative definition of cycles

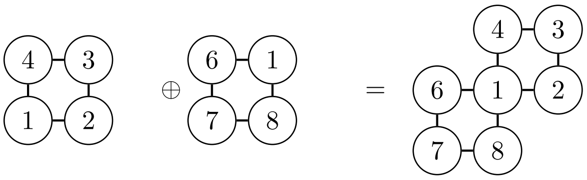
Again, most graph theory textbooks define cycles in a different way, notably as *closed walks*, which are walks whose endpoints coincide (e.g. G, H in Fig. 3.16 are closed walks). Moreover, if the walk is a trail, i.e. no edge is traversed more than once, then the corresponding closed trail is called a *tour* or *circuit.* These definitions would prevent the graph K in Fig. 3.16 to be called “cycle”, evidently, as it is not connected, whereas any walk or trail is connected by definition. And, to be fair, intuitively speaking the graph K looks like *two* cycles rather than just one. We justify our choice below.

3.4.7 The cycle space

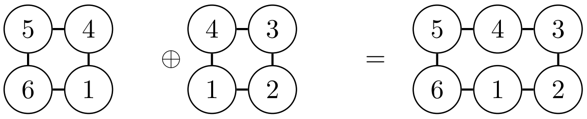
We have two reasons for defining a cycle as a subgraph of even degree. The first is its simplicity. The second is that, under this definition, the set $\mathcal{C}(G)$ of all cycles of a graph forms a vector space $(\mathcal{C}(G), \oplus)$ over the finite field $\mathbb{F}_2 = \{0, 1\}$, called the *cycle space.* The rules for adding two cycles in this vector space can be deduced from the following examples.



(3.1)



(3.2)



(3.3)

More precisely, let γ_1, γ_2 be two cycles in the graph G. If $V(\gamma_1) \cap V(\gamma_2) = \varnothing$, i.e. the two cycles have no common vertex (and hence no common edge either — why?), then the resulting cycle is simply the union of the two graphs γ_1 and γ_2, as shown in Eq. (3.1). If γ_1, γ_2 have a common vertex, again the resulting cycle is the union of γ_1, γ_2, as shown in Eq. (3.2). If the intersection of γ_1, γ_2 contains some edges, then the resulting cycle will contain edges in γ_1 or γ_2 but not both:

$$E(\gamma_1 \oplus \gamma_2) = E(\gamma_1) \triangle E(\gamma_2) = (E(\gamma_1) \cup E(\gamma_2)) \smallsetminus (E(\gamma_1) \cap E(\gamma_2)).$$

Like any vector space, the cycle space has a dimension and bases. We state the next result without proof (see e.g. [20] for a proof).

3.4.2 Theorem
The cycle space has dimension $|E| - |V| + 1$.

Bases of the cycle space allow a compact (linear) description of the (exponentially large) cycle space of a graph — this comes in handy in the classification of ring compounds in molecular chemistry, in the analysis and simulation of electrical circuits, and in the synthesis of public transportation timetables. Although extensions of these concepts to digraphs exist in the graph theory literature, they are are not sufficiently mainstream to introduce them in this course.

3.4.7.1 Cycle-edge incidence vectors

The *cycle-edge incidence vector* of a cycle γ is a binary vector with $m = |E|$ columns, that has a 1 in component $e \in E$ if and only if e is an edge in γ. For a cycle γ, let $\mathrm{inc}(\gamma)$ be the incidence vector of γ. We define a XOR operation on two binary vectors $\underline{u}, \underline{v}$ of the same length as follows: for any component index i, the i-th component of $\underline{u}\,\mathrm{XOR}\,\underline{v}$ is $\underline{u}_i\,\mathrm{XOR}\,\underline{v}_i$.

3.4.3 Exercise
For any two cycles γ, γ' *of the same graph* G, *prove that* $\mathrm{inc}(\gamma \oplus \gamma') = \mathrm{inc}(\gamma)\ \mathit{XOR}\ \mathrm{inc}(\gamma')$.

3.5 Basic operations on graphs

In later chapters, we are going to employ graphs to discuss some data structures. We are going to need mechanisms for modifying graphs as a result of the CPU modifying the data and the relations between data.

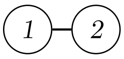
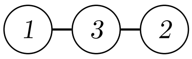
3.5.1 Addition and removal of vertices and edges

The easiest operations on a graph $G = (V, E)$ are the addition and removal of vertices and edges. We suppose that $V \subseteq \mathbb{N}$. To add a vertex to V, we pick an integer v which is not already in V and we set $V \leftarrow V \cup \{v\}$. To add an edge $\{u, v\}$ to E, we first verify whether $\{u, v\}$ is already in E or not. If it is, then we increase its multiplicity by one, obtaining a multigraph. Otherwise, we set $E \leftarrow E \cup \{\{u, v\}\}$.

To remove an edge $\{u, v\} \in E$ from E, we simply set $E \leftarrow E \smallsetminus \{\{u, v\}\}$. Removing a vertex v from V is slightly more involved, as we have to remove v and all edges incident to v:

$$\begin{array}{rcl} V & \leftarrow & V \smallsetminus \{v\} \\ E & \leftarrow & E \smallsetminus \delta(v). \end{array}$$

3.5.1 Example

To turn the graph  *into the graph* *, perform the following operations:*

1. *add vertex* 3
2. *add edges* $\{1, 3\}$, $\{2, 3\}$
3. *remove edge* $\{1, 2\}$.

Conversely, to turn the latter into the former:

1. *remove vertex* 3
2. *add edge* $\{1, 2\}$.

3.5.2 Contraction

Contracting an edge $\{u, v\} \in E$ essentially means "replace an edge with a vertex". More precisely, it consists of the following basic steps (see Fig. 3.17):

1. add a vertex z
2. for each $w \in N(u) \cup N(v)$ add the edge $\{w, z\}$
3. remove the vertices u and v.

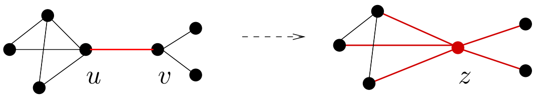


Figure 3.17: Edge contraction operation.

Contracting a whole subgraph H in a graph G means contracting each edge in $E(H)$ to the same vertex v_H (Fig. 3.18). Contracting an unlabeled subgraph H is a way to look at a complex graph "modulo H"; intuitively, it looks like zooming out of a complicated-looking graph to try and ascertain the core topological characteristics (Fig. 3.19). The graph G' obtained from G through a sequence of contractions is called a *minor* of G.

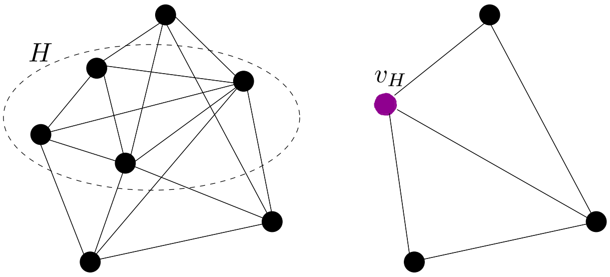


Figure 3.18: Subgraph contraction operation.

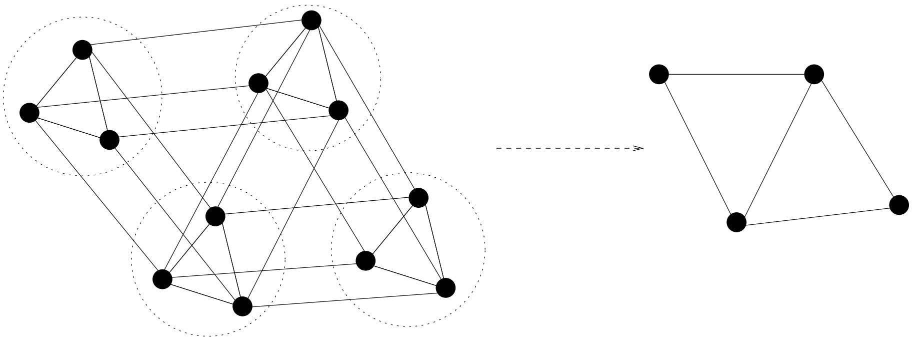


Figure 3.19: Contraction: zooming out of a complicated graph.

Chapter 4

Linear data structures

Abstract. Arrays: jagged arrays and adjacency lists, array operations, worst-case and average complexity of an incomplete loop. Lists and list operations, with a Java implementation. Queues, implemented as circular arrays. Stacks, with an example and an implementation. Maps. Use of parametrized classes.

We saw how data can be structured according to a class description, and also how arrays can contain several objects of the same class. In this context, arrays are one of many existing types of *data structures* (or *data containers*). The objects held in a data structure, such as an array, list, queue, tree or other, are called *entries*, or *elements*.

4.1 Arrays

Arrays were introduced in Sect. 2.2. Arrays implement a vector, a matrix or a tensor in mathematics. They model a contiguous, possibly multi-dimensional block of memory. Accordingly, we shall represent a linear array as a vector $x = (x_0, \ldots, x_{n-1})$, where n is the size of the array; for multi-dimensional arrays, each element x_i might itself be an array. This nesting is repeated as many times as the array has dimensions. The memory allocated to the array must be enough to hold at least n objects. Normally, the memory size does not change throughout the program, which implies that there is a limit to the number of objects we can insert into an array. Bypassing this limit is technically possible, but it involves resizing the array, which is an expensive operation in terms of CPU time.

We remark that **indexing starts from zero**[1] in Java, C and C++. The fact that the array elements are contiguous makes memory access very efficient. Reading or writing the i-th element of the array, given the memory address for $x[0]$, simply requires adding $i\times$ (length of the stored objects) bytes to it. This type of *index arithmetic* is carried out automatically by the Java compiler.

4.1.1 Jagged arrays

A *jagged array* is simply a multi-dimensional array in its most general configuration, i.e. where sub-arrays might have different sizes (see Fig. 4.1).

---

[1] In later chapters, when we pass from a code-oriented practical approach to a more theoretical setting, we shall employ arrays without necessarily assuming they are indexed starting from zero.

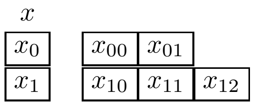


Figure 4.1: A jagged array structure.

4.1.1.1 Adjacency lists

Jagged arrays are useful to represent a graph $G = (V, E)$: rows are indexed by $v \in V$, and each row contains the elements of the star $\delta(v)$ in some order, arbitrary or otherwise. The generalization to digraphs is easy: each row only contains $\delta^+(v)$ or $\delta^-(v)$ depending on the context (we might even store both incoming and outgoing stars if need be).

4.1.1 Example
Let $V = \{1, 2, 3\}$ *and* $E = \{(1,1), (1,2), (2,3), (3,1), (3,2), (3,3)\}$*; the corresponding digraph can be represented in memory by means of a jagged array structure as follows.*

1	1	2	
2	3		
3	1	2	3

This graph representation is called *adjacency list*. Another common graph representation is by an edge array, or edge list (arc array or list in the case of digraphs). The two representations are useful in different contexts.

4.1.2 Array operations

The usual operations an array can carry out on the objects it stores are:

- read the value of the i-th component (worst-case complexity $O(1)$)
- write the value of the i-th component (worst-case complexity $O(1)$)
- return the size n of the array (worst-case complexity $O(n)$ or $O(1)$, depending on the implementation[2])
- remove the i-th element (worst-case complexity $O(n)$)
- insert an element at the i-th position (worst-case complexity $O(n)$)
- move a subsequence (contiguous set) of entries just after position i (worst-case complexity $O(n)$).

In summary, arrays are efficient when a lot of read/write accesses are needed at random positions, and the positions of the entries does not change.

4.1.2 Exercise
Show how the read and write array operations can be implemented in $O(1)$.

---

[2] With a Java array `a`, `a.length` returns its size in $O(1)$.

4.1.2.1 Size in $O(1)$

The *array size* is defined as the number of elements of the array, which is at most the amount of memory required to hold them: we call the latter quantity *array memory size*. The array memory size can be returned by asking the operating system how much memory was allocated starting to a given address.

In order to determine the array size, some array implementations employ a *delimiter*, stored as a "special character" (such as the ASCII 0 code) at the array element after the last. Finding the size can then be an $O(n)$ operation: starting with $i = 0$, increase i until x_i becomes the delimiter value, as in the pseudocode below.

```
$i \leftarrow 0$;
while $x_i \neq$ delimiter do
  $i \leftarrow i + 1$;
end while
$n \leftarrow i + 1$;
return $n$
```

This can be improved to $O(1)$ if we simply store the array size within the array itself. An array of integers could then be conceived as an object of the following class:

```
class Array {
  public int size;
  public int[] theArray;
}
```

Any array operation changing the size must then update the `size` attribute accordingly. Asking for the size then becomes a simple matter of returning `size`, which is an $O(1)$ operation.

We remark that Java's standard array implementation is already $O(1)$, without you needing to re-code a new array structure with this property. This section is simply meant to tell you that this is not a trivial array property.

4.1.2.2 Moving a subsequence

Let x be a given array of size n, $i \in \{0, \ldots, n-1\}$, and $L = (x_\ell, \ldots, x_{\ell+h})$ be the subsequence we want to move to the position right after i. This can be attained by means of h swaps. A *swap* (j, k) consists in the following instructions:

1. store the value x_j in a temporary variable t
2. copy the value x_k in the array entry indexed by j
3. copy the value of t in the array entry indexed by k.

After the swap (j, k), the effect on x is that x_j and x_k have swapped their values.

The following pseudocode moves the subsequence L to the position $i + 1$ in x, automatically re-positioning the elements that are currently stored at position $i+1$ and following (see Fig. 4.2).

```
for $j \in \{0, \ldots, h\}$ do
  swap $(\ell + j, i + j + 1)$
end for
```

This operation, which is also called *splice*, has worst-case complexity $O(n)$. The worst case occurs when the subsequence $(x_2, \ldots, x_n)$ must be moved to position 1.

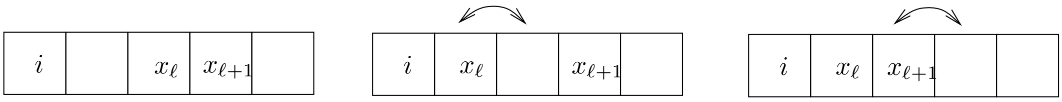


Figure 4.2: Array splicing.

4.1.2.3 Removal and insertion

Both removal and insertion operations can be carried out by splicing the array (see Sect. 4.1.2.2). To remove element x_i, we move $(x_{i+1}, \ldots, x_n)$ to position i (so that x_{i+1} overwrites x_i, and so on). To insert a new element a at position i, we move $(x_i, \ldots, x_n)$ to position $i+1$ (this is only possible if memory enough to hold $n+1$ objects was initially allocated to x), then set $x_i = a$.

4.1.3 Exercise
Show that both removal and insertion have $O(n)$ worst-case complexity.

4.1.3 Complexity of an incomplete loop

In Sect. 1.4.3.2 we analyzed the worst-case complexity of a simple loop, i.e. a loop where the termination condition is constant over the whole loop execution. Here we analyze the average-case complexity of a specific instance of a non-simple loop. Consider the following loop.

```
input $x \in \{0,1\}^n$;
int $i = 0$;
while $(i < n \wedge x_i = 1)$ do
    $x_i = 0$;
    $i = i + 1$;
end while
if $(i < n)$ then
    $x_i = 1$;
end if
output $x$;
```

The components of the input array x can only be zeroes or ones. The program asks the user to input an array x, then scans the loop while the current value is one, changing it to a zero. The loop terminates as soon as $x_i = 0$ for some i.

4.1.3.1 Worst-case complexity

Among all possible inputs of the algorithm, we have to identify the one leading to the longest execution time. From the condition $i < n \wedge x_i = 1$, either we continue until the end of the array, or we stop whenever the i-th component holds the zero value: this means that the longest loop execution occurs whenever all components of x are ones. We formalize this statement in the following proposition.

4.1.4 Proposition
The input yielding the worst complexity is $x = (1, 1 \ldots, 1)$.

Proof. Suppose the claim false, then there is a vector $x \neq (1, \ldots, 1)$ yielding a complexity $t(n) > n$. Since $x \neq (1, \ldots, 1)$, x contains at least one 0 component. Let $j < n$ be the smallest index such that

$x_j = 0$: at iteration j the loop terminates, and the complexity is $t(n) = j$, which is smaller than n: this is a contradiction. □

Notice that all other operations, aside from the loop, have $O(1)$ complexity. Therefore the overall worst-case complexity is $O(n)$.

4.1.3.2 Average-case complexity

In order to perform an average-case complexity analysis, we need to assume a probability distribution on random events. As far as the incomplete loop in Sect. 4.1.3 is concerned, the only random event is the input array x at Line 1. We assume that:

- for $b, c \in \{0, 1\}$, the event $x_i = b$ is independent from $x_j = c$ for all $i \neq j < n$;
- the events $x_i = b$ and $x_i = 1 - b$ are equally likely for each $i < n$.

From the code in Sect. 4.1.3, we derive the followith facts:

1. for any vector having first $k + 1$ components $(\underbrace{1, \ldots, 1}_{k}, 0)$, the loop is executed k times (for all $0 \leq k < n$);
2. for the vector $(\underbrace{1, \ldots, 1}_{n})$, the loop is executed n times.

By the independence of events, the event that any random vector of $k+1$ components has a specific given value $(b_0, \ldots, b_k)$ has probability $(\frac{1}{2})^{k+1}$ (which is therefore the probability of the first $k + 1$ components of k being $(\underbrace{1, \ldots, 1}_{k}, 0)$). When $k = n - 1$, we derive that the probability of x being the all-one n-vector is $(\frac{1}{2})^n$. In other words:

- the loop is executed k times with probability $(\frac{1}{2})^{k+1}$ for $k < n$;
- the loop is executed n times with probability $(\frac{1}{2})^n$.

The average number of iterations in the loop is therefore:

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n}.$$

Because $2^{-(k+1)} < 2^{-k}$, this quantity is bounded above by $\sum\limits_{k=0}^{n-1} k2^{-k} + n2^{-n}$, which is itself equal to

$$\sum_{k=0}^{n} k2^{-k}.$$

We now find the limit of this sum as n tends to infinity: this gives the asymptotic average complexity of the incomplete loop.

4.1.5 Lemma
$\lim\limits_{n\to\infty} \sum\limits_{k=0}^{n} k2^{-k} = 2$

Proof. Consider the geometric series $\sum_{k\geq 0} q^k = \frac{1}{1-q}$ for $q \in [0,1)$. Differentiate it w.r.t. q, to get $\sum_{k\geq 0} kq^{k-1} = \frac{1}{(1-q)^2}$. Multiply this by q, to get $\sum_{k\geq 0} kq^k = \frac{q}{(1-q)^2}$. For $q = \frac{1}{2}$, we get $\sum_{k\geq 0} k2^{-k} = (1/2)/(1/4) = 2$. □

It follows that the average complexity of this particular incomplete loop is $O(1)$. This marks a striking difference with the worst-case complexity of $O(n)$.

4.1.4 Limitations of the array structure

Although the array is perhaps the best known data container, and the easiest to understand, it has some drawbacks, which concern the evolution of the container size during the execution of the program.

Suppose we allocate enough memory to store, say, five integers, but along the execution some user input or other occurrence changes the initial assumptions, and six integers must be stored. We might allocate a new array, large enough to hold six integers, then copy the five integers from the old array to the new, then append the new integer in the last position of the new array. But this is very time-consuming (memory allocation is a relatively expensive operation, and if the old array contains many elements, the copy operation takes a time proportional to the length of the array), so it should be a last resort. Other costly copy operations may arise when a new element should be put at a position i in the middle of the array: all the elements from the $(i+1)$-st to the last should be shifted to make space for the new element. These limitations are addressed by lists, presented below.

4.2 Lists

A list is a linear data structure that models a sequence. Most of the common subsequence operations can be implemented efficiently.

4.2.1 Singly-linked lists

Lists work by abstracting contiguity: whereas arrays components are usually stored in contiguous memory cells, the entries $x_0, \ldots, x_{n-1}$ of a singly-linked list x are actually pairs $x_i = (x'_i, \nu_i)$, where x'_i is the value stored at x_i, and ν_i is a reference variable holding the memory address of the next list entry x_{i+1}. The reference ν_{n-1} of the last list element holds the `null` (invalid) address (see Fig. 4.3).

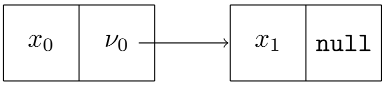


Figure 4.3: A singly-linked list representation of (x_0, x_1).

A simple Java implementation of a singly-linked list is given in Sect. 5.3.2.1.

4.2.2 Doubly-linked lists

Doubly-linked lists are similar, but each entry is a triplet (π_i, x'_i, ν_i), where x'_i, ν_i are as in the singly-linked case, and π_i is a reference to the previous list entry x_{i-1}. The reference π_0 of the first list element holds the `null` address (see Fig. 4.4).

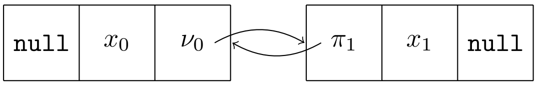


Figure 4.4: A doubly-linked list representation of (x_0, x_1).

4.2.2.1 The placeholder node

In doubly-linked lists implementations, it is common to replace the two `null` references at the beginning and end of the list with a *placeholder* entry denoted by $\perp$. The placeholder holds a selected value whose meaning is understood by the program to mean "first" or "last" according as to whether it precedes the first entry or succeeds the last entry, as shown in Fig. 4.5.

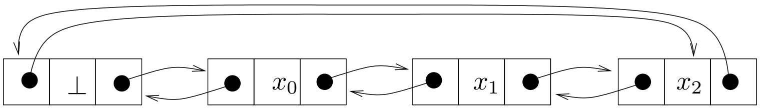


Figure 4.5: The placeholder node (leftmost) in a doubly-linked list representation of (x_0, x_1, x_2).

4.2.3 Lists modelled as graphs

It should appear clear from the pictures that we can draw a parallel between lists and graphs: list entries are nodes or vertices, and the reference relation (the arrow) is represented by an arc or edge. More precisely, a singly-linked list can be represented by a simple directed path (Fig. 4.6, left), and a doubly-linked list can be represented by a simple (undirected) path; if a placeholder is used, the doubly-linked list is represented by a simple cycle (Fig. 4.6, right).

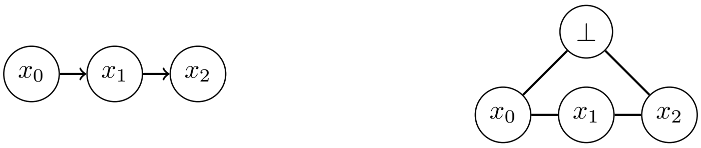


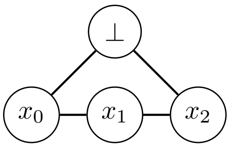
Figure 4.6: Singly-linked and doubly-linked lists for (x_0, x_1, x_2) in the graph representation.

4.2.4 List operations

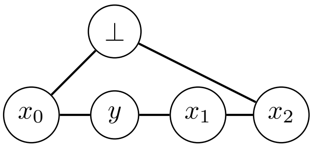
In this section we describe the basic operations of doubly-linked lists with a placeholder.

4.2.4.1 Insertion

Consider a doubly-linked list storing (x_0, x_1, x_2) and represented as the simple cycle shown below.



In order to insert an element y at position 1, we proceed as follows: we remove the edge $\{x_0, x_1\}$, we add a new vertex y, then add edges $\{x_0, y\}$ and $\{y, x_1\}$.



This can be generalized. Given a doubly-linked list $x = (x_0, \ldots, x_{n-1})$ and new element y to be inserted at position $i \in \{0, \ldots, n-1\}$, we proceed as follows:

1. remove edge $\{x_{i-1}, x_i\}$
2. add vertex y
3. add edges $\{x_{i-1}, y\}$ and $\{y, x_i\}$.

This involves 4 elementary operations on the graph, yielding an $O(1)$ method in the worst case.

4.2.4.2 Removal

Given a doubly-linked list $x = (x_0, \ldots, x_{n-1})$ and an index $i \in \{0, \ldots, n-1\}$, removing the element x_i is as follows:

1. remove vertex x_i (recall from Sect. 3.5.1 that removing a vertex also removes its associated cutset)
2. add the edge $\{x_{i-1}, x_{i+1}\}$.

4.2.1 Exercise
Show that removal is $O(1)$ in the worst case. What is the average case complexity?

4.2.4.3 Find

Given a doubly-linked list $x = (x_0, \ldots, x_{n-1})$ with placeholder $\perp$ stored both before x_0 and after x_{n-1}, an index $i \in \{0, \ldots, n-1\}$, and a value b of the same data type as the list, the find operation is as follows: if $b \in x$ then return x_j such that $x_j = b$, else return $\perp$. This can be implemented by travelling along the cycle, starting at vertex x_i, traversing vertices $x_{i+1}, x_{i+2}, \ldots$ until either $x_j = b$ for some j, or $x_j = \perp$. We then return x_j.

4.2.2 Exercise
Show that find is an $O(n)$ method in the worst case. Compute the average case complexity assuming x is an array holding binary values.

4.2.4.4 Access

Accessing list element i means reading it or changing its value. Since list entries are non-contiguous, index arithmetic is meaningless. Accordingly, we must start at the vertex labelled x_0 and travel along the cycle in the direction of increasing indices, counting the vertices and stopping at the i-th one, then perform the read or write operation. The worst-case is whenever $i = n - 1$, which yields $O(n)$ methods for both reading and writing.

4.2.3 Exercise
Show that a simple implementation of a method for computing the list size has $O(n)$ worst case complexity; point out how to implement an $O(1)$ size operation.

4.2.4.5 Other operations

The following table details list operations with the corresponding worst-case complexity.

Operation	*Complexity*
Access i-th entry	$O(n)$
Find next node having given value	$O(n)$
Size	$O(n)$ or $O(1)$
Is the list empty?	$O(1)$
Access first/last node	$O(1)$
Remove element at given position	$O(1)$
Insert element at given position	$O(1)$
Move subsequence to given position	$O(1)$
Pop from front/back	$O(1)$
Push to front/back	$O(1)$
Concatenate	$O(1)$

Three operations, *push*, *pop* and *concatenate*, have never been introduced before. Pushing and popping are terms connected with stacks, which we shall discuss in Sect. 4.4. In the context of lists, pushing an element at the front or back of a list means to add an element at the first or last position; popping means removing. Concatenating a list $x = (x_0, \ldots, x_{n-1})$ and a list $y = (y_0, \ldots, y_{m-1})$ yields the list $(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})$.

4.2.5 Java implementation

We propose a very simple doubly-linked list implementation, stored in the text file `DLList.java`, structured in two classes: `Node` and `DLList`. The latter, since it has the same name as the file that stores it, also has the point of entry `main`. We remark that neither class is declared `static`; consequently, objects of these classes are not automatically allocated in memory: they must be allocated manually using the `new` operator.

In our implementation, every `DLList` element is a `Node` object. Each `Node` has references to previous and next nodes in the list, and has methods for inserting other nodes after itself, removing itself from the list, as well as printing its own data contents. The `DLList` stores a reference to the $\perp$ node, and implements all other list methods. We emphasize that leaner implementations of a doubly-linked list are possible: specifically, it is possible (and also desirable[3]) to implement a list using only one class, such as the singly-linked list implementation given in Sect. 5.3.2.1. We hope our implementation will also give readers an idea of the interplay between two different classes in the same program.

---

[3] If you, as a student, are required in an exam to propose a linked list implementation, it is safe to assume that you should target a one-class implementation.

The `main` method is not strictly necessary: data structure implementations usually provide *libraries*, employed by users within other programs.

4.2.5.1 The `Node` class

As usual, we begin the file `DLList.java` by writing appropriate header comments and declaring useful imports.

```
class Node {
    public Node prev;
    public Node next;
    public int datum;

    public void remove() {
        this.prev.next = this.next;
        this.next.prev = this.prev;
    }

    public void insert(Node M) {
        this.prev = M;
        this.next = M.next;
        M.next = this;
        this.next.prev = this;
    }

    public void print() {
        System.out.print(this.datum + " ");
    }
}
```

As previously mentioned, a `Node` stores a reference to previous and next elements in the list, as well as a piece of data (the `datum`, which in this case has `int` type). The `this` keyword may help disambiguate equally named attributes of different objects of the same class. The piece of code that executes `this` belongs to a specific object in memory: `this` contains the address where this object is stored.

Removal of a node works by setting its previous node's `next` attribute to its next node's `previous` attribute, and vice-versa, as shown in Fig. 4.7.

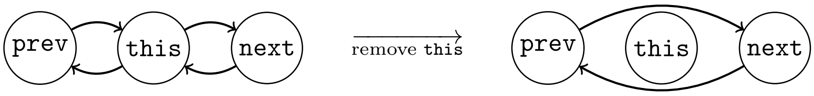


Figure 4.7: Implementing a `Node`'s removal.

Insertion is similar to removal, but the point of view of the code changes: `this` becomes the previous node of the new node being inserted (called `M` in the code above).

4.2.5.2 The `DLList` class

As for the `functionPlot` example, we give the `DLList` class structure first: this code will not be functional until all the class method implementations have been filled in.

```
class DLList {
    public Node bot; // the beginning/end-of-list placeholder
    public DLList(); // class constructor
    public Node first(); // return the first element
    public Node last();  // return the last element
    public void pushBack(int t); // insert a new element t after the last one
    public int popBack(); // return last element's datum and remove from list
    public int size();    // return the list size
    public void print();  // print the whole list
    public Node find(int b); // return the first node containing datum b
    public Node findNext(Node x, int b); // start looking from Node x
    public static void main(String[] args); // the point-of-entry
}
```

The only attribute of the `DLList` class is the `bot` node, which implements the $\perp$ placeholder. The other class members are all methods. All members are `public`. (this decision is somewhat arbitrary, and dictated by simplicity: usually, attributes are private and methods are public).

The first method, which has the same name as the class and has no return type, is known as the *constructor* of the class. It is called whenever a program issues the instruction

```
  DLList myList = new DLList();
```

in order to manually allocate some memory to the reference `myList`. Constructors are mostly used to allocate memory and/or initialize some variables with default values

```
    public DLList() {
        this.bot = new Node();
        this.bot.prev = bot;
        this.bot.next = bot;
        this.bot.datum = 0;
    }
```

The `first` and `last` methods return the first and last elements of the list respectively.

```
    public Node first() {
        return bot.next;
    }
    public Node last() {
        return bot.prev;
    }
```

The `pushBack` method inserts a new list element after the last, and `popBack` returns the last element, then removes it.

```
    public void pushBack(int t) {
        Node N = new Node();
        N.datum = t;
        N.insert(this.last());
    }
    public int popBack() {
        int t = this.last().datum;
        this.last().remove();
        return t;
    }
```

Our implementation of the `size` method is not efficient, as it takes $O(n)$ in the worst case.

```
public int size() {
    int t = 0;
    Node x = first();
    while(x != bot) {
        t++;
        x = x.next;
    }
    return t;
}
```

The `print` method prints the whole list

```
public void print() {
    Node x = first();
    while(x != bot) {
        x.print();
        x = x.next;
    }
    System.out.println();
}
```

The `find` method looks for a node containing the given argument in its `datum` attribute. The `findNext` method does the same but starts looking from a given node `x`.

```
public Node find(int b) {
    return findNext(bot.next, b);
}
public Node findNext(Node x, int b) {
    bot.datum = b;
    while(x.datum != b) {
        x = x.next;
    }
    return x;
}
```

4.2.5.3 The `main` function

We give below a possible `main` function, just for validation purposes.

```
public static void main(String[] args)  {
    // in order to call those attributes/methods which are NOT
    // static, you need to first instantiate a dynamic object
    // of the class
    DLList dl = new DLList();

    System.out.println("size=" + dl.size());
    dl.pushBack(4);
    dl.pushBack(3);
    dl.pushBack(7);
    System.out.print("list is now ");
    dl.print();
    System.out.println("popping last element: " + dl.popBack());
    System.out.print("list is now ");
    dl.print();
    dl.pushBack(5);
    dl.pushBack(9);
```

```
        dl.pushBack(4);
        System.out.print("list is now ");
        dl.print();
        System.out.println("finding node containing 9");
        Node N = dl.find(9);
        if (N != dl.bot) {
            System.out.println("deleting node containing 9");
            N.remove();
        }
        System.out.print("list is now ");
        dl.print();
    }
```

4.3 Queues

A *queue* is a linear data structure with four important operations:

- insert a new element at the end of the queue
- retrieve and delete the element at the beginning of the queue
- test whether the queue is empty or not
- find the size of the queue.

These operations need to be performed fast: ideally, they should be worst-case $O(1)$. Although queues can be simulated using arrays or lists, we are going to describe a queue implementation using *circular arrays*. In queues, the first element to go in is also the first to come out. This is known as "first in, first out" (FIFO).

The need for a specific implementation for queues arises because queues grow at the end and shrink at the beginning. Imagine storing a queue in an array: after any given number of insert operations followed by a corresponding retrieve/delete, the queue has the same length but has "moved" somewhere else in memory. Eventually the array will exhaust its allocated space, even though the number of stored elements may be constant (see Fig. 4.8).

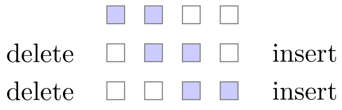


Figure 4.8: A queue worms its way along the array, exhausting and wasting space.

4.3.1 Circular arrays

The appropriate graph model for a circular array of size n is a cycle with $n+1$ vertices (see Fig. 4.9).

Let $C = (V, E)$ be a cycle: we define $V = \{q_0, \ldots, q_n\}$ and $E = \{\{q_i, q_{i+1}\} \mid i \in \{0, \ldots, n-1\}\}$. At each vertex of V we store the address of an object of the same class, or a `null` address denoted by $\perp$. We want to store a queue $d = (d_0, \ldots, d_t)$ in C, in such a way that whenever the queue "advances" in memory, by means of repeated insertions at one end and removals at the other end (but keeping its size constant), the storage will "cycle" through C, re-occupying the same memory cells whenever these are

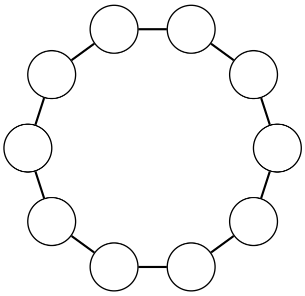


Figure 4.9: The graph representation of a circular array: a cycle.

empty (see Fig. 4.10). Obviously, we have to make the assumption that the size t of the queue will never exceed the size n of the circular array. If vertex q_k contains the last queue element d_t at any time, then we store an end-of-queue placeholder $\perp$ at vertex q_{k+1}. In order for vertex indices never to exceed n, we employ arithmetic modulo n on indices (so that, e.g., q_{k+1} is in fact $q_{k+1 \pmod n}$).

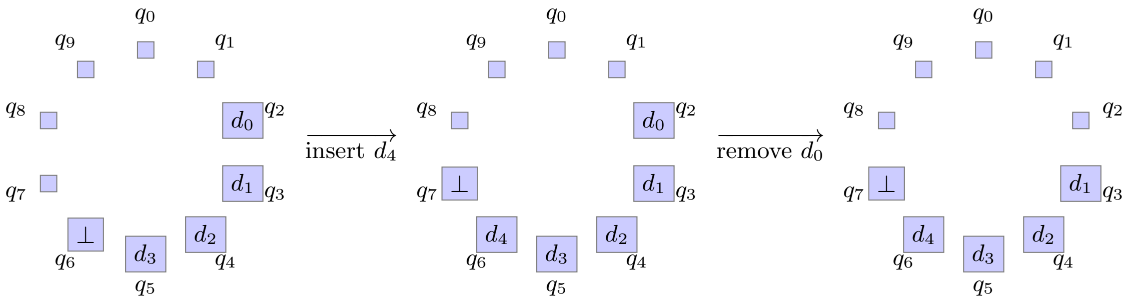


Figure 4.10: The queue "advances" in the memory cells of the circular array.

4.3.1.1 Java implementation

The implementation of a circular array is very simple; it is based on a linear array of fixed length. All index arithmetic is carried out modulo that length.

```
public class circularArray {
    // attributes (private)
    int[] q;// the array
    int n;  // array size
    int h;  // index of head element (first data element stored at q[h])
    int t;  // index of tail element (last data element stored at q[t-1])

    // methods (public)
    public circularArray(int qlen) {  // class constructor
        n = qlen;
        q = new int[n];
        h = 0;
        t = 0;
    }
    public boolean isEmpty() {  // test whether array is empty
        if (h == t) {
            return true;
        } else {
            return false;
        }
    }
    public int first() {  // return first element
        assert(!isEmpty()) : "error: queue is empty, cannot read first element";
        return q[h];
    }
    public int popFront() { // read and remove the first element
        int p = this.first();
        h++;
        return p;
    }
    public int size() { // return the
        int theSize = (t - h + n) % n;
        return theSize;
    }
    public void pushBack(int d) { // insert new element at array end
        assert(this.size() < n) :
        "error: queue is full: cannot push elements on its back";
        q[t] = d;
        t = (t+1) % n;
    }
}
```

Notice that, even though any array element can be accessed in $O(1)$, the `circularArray` class offers a limited interface, functional to a queue: it is only possible to push new elements on the back of the array and pop elements from the front.

Notice also that the class constructor takes an argument as input, i.e. the array length. Constructors can take arguments as any other function, but their return type is not declared, since it is defined to be the same class as they belong to.

The `assert` is used for error detection: if the test passed to `assert` as an argument is false, execution stops with the given error message.

4.3.1 Exercise
Show that every method of the `circularArray` *class is* $O(1)$.

4.3.2 What are queues used for?

Queues are handy when simulating the behaviour of queues in the physical world, be they human, car, packet or other type of queues. Secondly, they are essential in the implementation of a well-known algorithm for graph exploration, namely breadth-first search (BFS). We shall discuss BFS in Sect. 8.2. Lastly, a queue variant called *priority queue* (see Sect. 11.4.1) is essential to the implementation of most algorithms for computing shortest paths in graphs (see Chapter 12).

4.4 Stacks

A stack is similar to a queue, but instead of pushing at the back and popping from the front, we push and pop from the back. A stack data structure is appropriate in situations where the last element to be stored is the first to be retrieved. This is known as “last in, first out” (LIFO).

4.4.1 Using stacks for validating mathematical syntax

At a glance, are you able to state whether the following mathematical expression has a balanced number of brackets?

$$\left((((x+y)z-2\sin(x))+\frac{1}{2}\left(\sum_{i\leq 3}p_i^2\right))/\log((p_1(p_2-p_3^{(\cos(\tan(x)))})))\right)$$

One way to go about this task is to keep a counter, initially set at zero: scanning the expression, increase the counter by one unit every time you read an open bracket, and decrease it by one unit every time you read a closed bracket. If the counter is always ≥ 0 and gets to zero by the time the scanning is over, then the brackets are balanced.

What if you had square brackets as well, however? A situation like $([1+2)]$ would result in a correct balancing, but would miss the incorrect embedding. To overcome this problem, we can use a stack: every time an open bracket is read, push the closing corresponding bracket on the stack. Every time a closing bracket is read, pop the last element off the stack, and verify it is the same. If we get to the end of the stack at the same time the scanning ends, then the brackets are well-balanced, otherwise the expression is wrong.

4.4.1 Exercise
Prove the last assertion formally.

4.4.1.1 Java implementation

We implement the above idea, and the associated stack, in a class called `bracketStack`. The implementation of the `main` function is given later.

```
class bracketStack {
    // attributes (private)
    int [] theStack; // the stack is implemented as an array
    int pos;         // the current stack size is stored in pos

    // methods (public)
    public bracketStack(int maxSize) { // class constructor
        theStack = new int[maxSize];
        pos = 0;
    }
    public Boolean isEmpty() { // is the stack empty?
        Boolean t = Boolean.TRUE;
        if (pos > 0) {
            t = Boolean.FALSE;
        }
        return t;
    }
    public void push(int element) { // push an element on the stack
        assert(pos < theStack.length) : "no more memory for growing stack";
        theStack[pos] = element;
        pos++;
    }
    public int pop() {  // pop an element from the stack
        assert(!isEmpty());
        pos--;
        return theStack[pos];
    }

    // the main method
    public static void main(String[] args);
}
```

4.4.2 Exercise
Show that every method in this stack implementation is $O(1)$

4.4.1.2 The `main` method for `bracketStack`

Find below the implementation of bracket-checking algorithm discussed above.

Notice that this code reads the mathematical expression from the command line: it is an example of how to use the `String[] args` array. `String` is a standard library class for storing strings. When a shell command such as:

```
java bracketStack "((([])))"
```

is given, the Java compiler makes all arguments after the name of the program (in this case, `bracketStack`) available to the program itself within the array of strings `args`. More precisely, `args[0]` contains the first command, `args[1]` contains the second, and so on. Since `args` is declared as `String[]`, every element is a string; we can therefore `args[0].length()` to find the length of the string stored in `args[0]`.

```
public static void main(String[] args) {

    // read bracketed sentence from cmd line
    int argSize = args[0].length();
    String theArg = args[0];

    // initialize dynamic object of this class
    bracketStack brackStack = new bracketStack(argSize);

    // counter is the current character index in the input sentence
    int counter = 0;

    // character popped off the stack
    int elt;

    // error status:
    //   0 is "valid sentence"
    //   1 is "wrong closing bracket"
    //   2 is "too many closing brackets"
    //   3 is "not enough closing brackets"
    int err = 0;

    // look at every character of the input sentence
    while(counter < argSize) {

        if (theArg.charAt(counter) == '(') {
            // whenever an open round bracket is found,
            // push the closing one on the stack
            brackStack.push(')');

        } else if (theArg.charAt(counter) == '[') {
            // whenever an open square bracket is found,
            // push the closing one on the stack
            brackStack.push(']');
        } else if (theArg.charAt(counter) == ')' ||
                   theArg.charAt(counter) == ']') {
            // if any type of closing bracket is found
            if (brackStack.isEmpty()) {
                // ...and the stack is empty, it means there are more
                // closing brackets than open ones
                err = 2;
                break;

            } else {
                // ...pop an element off the stack
                elt = brackStack.pop();
                if (elt != theArg.charAt(counter)) {
                    // if this is different from the closing bracket,
                    // it means that either an open round bracket was
                    // closed by a square one, or vice versa
                    err = 1;
                    break;
                }
            }
        }
        // increase the character counter
        counter++;
    }
    if (err == 0 && !brackStack.isEmpty()) {
        // if no error of type 1 or 2 were reported, but the stack is not
        // empty yet, it means that there were more open than closed
        // brackets
        err = 3;
    }

    // report status message
    if (err == 1) {
        System.out.println("wrong closing bracket at pos " + (counter+1));
    } else if (err == 2) {
        System.out.println("too many closing brackets");
    } else if (err == 3) {
        System.out.println("not enough closing brackets");
    } else {
        System.out.println("valid sentence");
    }
}
```

4.4.1.3 Sample output

Here's some sample output from the `bracketStack` program, when called from the command line.

```
$ java bracketStack "(([]]"
wrong closing bracket at pos 3
$ java bracketStack "(())"
valid sentence
$ java bracketStack "(()))]"
too many closing brackets
$ java bracketStack "[(())]"
valid sentence
$ java bracketStack "([1+2)]"
wrong closing bracket at pos 6
```

```
$ java bracketStack "[(1+2)]"
valid sentence
```

4.4.2 Calling functions

The main use of stacks within an operating system (OS) is to simulate the function call mechanism. In fact, classic CPUs have native instructions for transferring control to the code instruction stored at a given memory address (the so-called "GOTO" statement, also called `jmp` in assembly language), but not necessarily a `return` statement. This, however, is easily implemented by simply storing the calling address before `jmp`ing to the new one, and then re-`jmp`ing to the stored calling address whenever a `return` is issued.

Recall that a `return` statement issued from a function h can only return control to the function g that directly called h, but not to the function f that called g (see Fig. 4.11). Therefore, when nested calling occurs, the addresses of the calling functions can be pushed on a stack, and popped at each successive `return` statement (see Fig. 4.12).

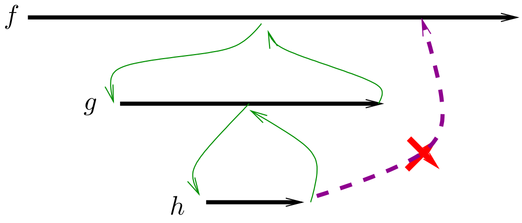


Figure 4.11: The function h cannot return control to f: it must return it to its direct calling function g.

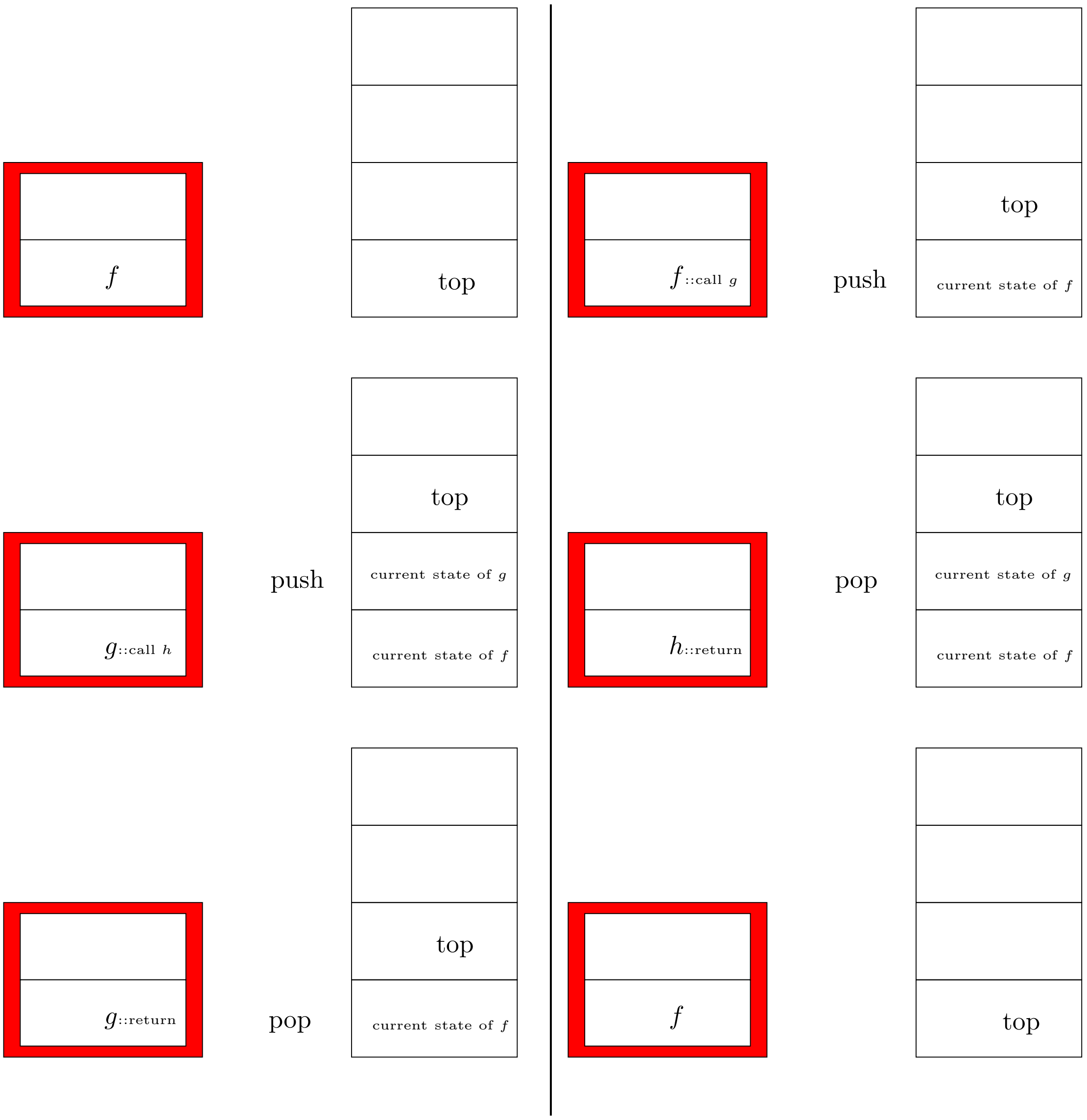
4.4.2.1 Smashing the stack for fun and profit

The first serious breaches into interconnected computer systems was carried out by a technique called "smashing the stack". It was based on a poor implementation of some string input functions, where a string was simply a fixed-length array of characters. These input functions did not check that the length of the user input was actually smaller than the allocated array memory, and hence a long input would overwrite memory past the array bounds. If the array was allocated on the same stack used for the function calls, then a hacker would translate some particularly nasty code into ASCII and pass it to the string input function. On returning from the string input function, the return address was overwritten with the nasty code, which got executed and spawned all sorts of actions, such as opening a root shell on the hacker's terminal (see Fig. 4.13).

4.5 Maps

Maps are representations of functions. In mathematics, a function f from a set X to a set Y as is defined as a set F of pairs (x, y), with $x \in X$ and $y \in Y$, and such that if $y = f(x)$ and $y' = f(x)$ then $y = y'$. This property is known as *well-definedness* of the function. Correspondingly, a *map* is a data structure defined over two non-elementary data types, that play the role of X and Y, for storing well-defined pairs (x, y). In Java, a map where X, Y are both integers is defined as:

```
  Map<Integer,Integer> theMap;
```



f
top
f::call g
push
top
current state of f
g::call h
push
top
current state of g
current state of f
h::return
pop
top
current state of g
current state of f
g::return
pop
top
current state of f
f
top

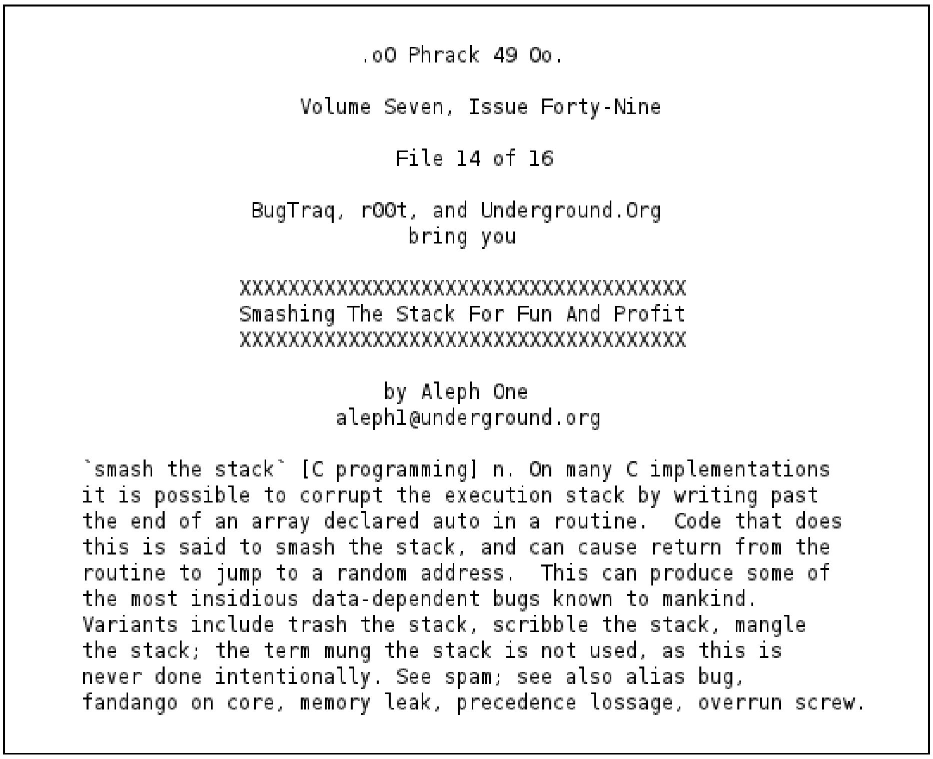
Figure 4.12: What happens to the OS stack as f calls g which calls h.

4.5.1 Maps as parametrized interfaces

Two remarks are in order. First, a `Map` is an interface (see Sect. 2.1.3.3) rather than a class — Java offers different implementations of a map.

Second, a Java `Map` is *parametrized* over two other classes. This is common in *generic containers*: these are data containers, such as arrays, lists, maps and so on, whose code is object-independent: in other words, objects of any class can be stored in such containers, it suffices that the "parameter class" be declared in the program by using the angle bracket notation `<>`. A requirement for the parameter class is that it should be a non-elementary data type. In our case, since `int` is an elementary data type, we use the non-elementary equivalent `Integer`.

We remark that classes can also be parametrized (as well as interfaces), and that a common name in the literature for parametrizing a type with another type is *template*.



```
                        .oO Phrack 49 Oo.

                    Volume Seven, Issue Forty-Nine

                            File 14 of 16

                BugTraq, r00t, and Underground.Org
                            bring you

               XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
               Smashing The Stack For Fun And Profit
               XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                          by Aleph One
                      aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine.  Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address.  This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.
```

Figure 4.13: The *Phrack* issue that made stack smashing famous.

4.5.2 Example of map usage in Java

See the following example of the use of a Java map, implemented as a `HashMap` (see Ch. 5 for details about hashing).

```
Map<String,Integer> phonebook = new HashMap<String,Integer>();
phonebook.put("Leo", 169334138);
phonebook.put("Tom", 169334425);
System.out.println("Leo’s phone number is 0" + phonebook.get("Leo"));
```

Chapter 5

Hashing

ABSTRACT. Hashing: efficiently finding items in a large-sized array. Motivation, example, Java implementations.

A *hash table* is a data structure for storing a function $\tau : K \to U$, where K is a set of *keys* and U is a set of *records*, so that $\tau(k)$ can be retrieved efficiently from memory.

5.1 Do we really need it?

The definition of a hash table might appear puzzling at first sight: why not store functions by means of lists or arrays? We saw in Sect. 4.2.4.3 that retrieving a specific element of a list takes time proportional to the list length. Hence, storing the pairs (key, record) in a list will not yield a retrieval-efficient data structure. What about an array, however? After all, if `a` is an array of records, and `i` is an `int`, then `a[i]` is retrieved in constant time by simply looking up the value stored at the memory address indexed by the base address of `a` plus `i` times the number of bytes taken to store an `int`. If, however, the function `a` were to map integers from the set $K = \{1, 16, 1643, 1094382\}$, using an array would require allocating space for 1094382 elements, although we would only use four of them: definitely too wasteful.

What if our keys are names, as might happen in a telephone directory? We could address this issue by listing all possible names in an order, and then remember the order rank of each name. This has drawbacks. Storing the assignment (name, rank) would take a huge amount of memory if we wanted to do this for *every possible name in existence*: and besides, we would have simply shifted the problem to efficiently scan this assignment between names and ranks. Suppose now we choose an alphabetical order, and only limit the assignment to a given subset of names (for example, those names corresponding to people we know). As we get to meet new people, we need to store their names in the subset. Thus, we have to insert new elements in the assignment: if this is stored as an array, we know from Sect. 4.1.2 that insertion takes time proportional to the array length, which may turn out to be too slow if the array grows considerably.

In summary, neither lists nor arrays provide the efficient data structure we are looking for: and this motivates the definition of a hash table.

5.1.1 The phonebook example

The solution is given by a phonebook. This consists of a notebook whose pages are indexed with letters: A, B, . . . , Z. One writes a name and the corresponding telephone number (say “Leo Liberti, 0169334138”) into the page indexed by the initial letter of the surname (‘L’ in this case). Every page has a finite length, but we assume the mechanism is unlikely to fail because most people have fewer acquaintances whose names begin with a specific letter than can be written in a page. Thus, in order to find the key “Leo Liberti” in the phonebook and retrieve the record “0169334138”, it suffices to identify the correct page, and then to scan the whole page for the correct key. Identifying the correct page is an operation that requires constant time, since the page position from the beginning is proportional to the position of the corresponding letter with respect to the first alphabet letter ‘A’, and the size of the alphabet is a constant. The time taken by scanning each page is at worst proportional to the page length, which is constant over the whole notebook — hence this time is also constant. Overall, then, a phonebook allows you to find a pair (key, record) in constant time, independently of the actual number of phonebook entries, as long as they fit in the phonebook.

Here is some good news: if you understood this phonebook example, you understood the idea of hashing.

5.1.2 Formal explanation

Now keep the phonebook of Sect. 5.1.1 in mind and let K be a set of keys, U be a set of records, and τ be a table mapping a subset of K to U. We denote by $\operatorname{dom}\tau$ the domain of τ, and assume that $\operatorname{dom}\tau$ is “small” with respect to K. Using the phonebook example metaphor: there are many millions, perhaps billions of names in the world, but the set of your acquaintances counts dozens, hundreds or in the most extreme case thousands, of names.

Consider a set I of indices, which we shall assume for simplicity to be $\{0, 1, \ldots, p-1\}$, of cardinality “more or less” like that of U. By “more or less” we mean that $|I|$ is $O(|U|)$. Also consider a *hash function* $h : K \to I$ that maps keys to indices. Finally, consider an array σ, indexed by I, whose elements are linear data structures of some type (e.g. arrays or lists), all having the same fixed size α. For any $k \in \operatorname{dom}\tau$, we store $\tau(k)$ in the hash table σ, within the fixed size linear data structure $\sigma(h(k))$ (see Fig. 5.1). Should two distinct keys k, k' be mapped by h to the same index i, then the corresponding records $\tau(k), \tau(k')$

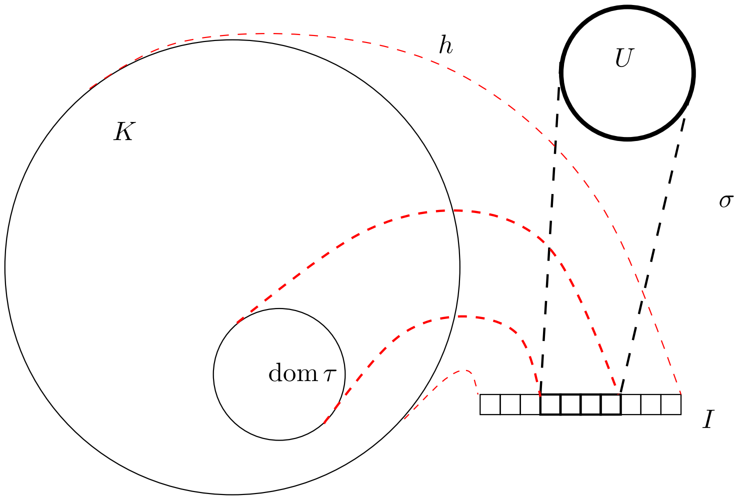


Figure 5.1: Hashing.

would both be stored in the fixed size linear data structure $\sigma(i)$. This assumes that no more than α keys are mapped by h to the same index.

In the phonebook metaphor, h maps names to their initials, we assume that no phonebook user has more similarly-initialled acquaintances than the α lines in each page of the phonebook τ, and if we have a name k with initial $h(k)$, we write the corresponding record on the page indexed by $h(k)$.

5.1.3 Applications of hashing to Java

Aside from the phonebook, hash tables are often used for storing maps (see the `HashMap` class used in Sect. 4.5).

Hash functions (the function h mapping keys to indices) also have very useful applications of their own. In Java programming, for example, objects often occupy sizable chunks of memory; if we need to know whether two separate objects a, b of the same class C, stored at different addresses, contain the same data, we have to run a byte comparison on the memory occupied by a and b. This will run in time $O(\max(|a|, |b|))$, where $|a|, |b|$ are the memory sizes of a, b respectively. The way this can be done in $O(1)$ is by devising an injective hash function $h : C \to$ `int` that computes an integer code for unique to each object in the class C. This way, $a = b$ if and only if $h(a) = h(b)$ (the latter test can be done in constant time).

Although it is very difficult to construct a hash function that is guaranteed to be injective, it is not too hard to only require h to be injective with high probability (i.e. minimizing the occurrences that $a \neq b$ even though $h(a) = h(b)$). Java offers a default hash function, called `hashCode()`, that applies to every class:

```
public class C { ... };
// ...
C a = new C();
C b = new C();
// ...
if (a.hashCode() == b.hashCode()) {
  System.out.println("a = b");
}
```

Since the Java developers recognize that more efficient hash functions can be developed by the programmers in certain instances, the `hashCode` function can be *overloaded* (i.e., replaced by a user-defined function with the same name and taking the same number and type of arguments).

5.2 The last nagging doubt

Perhaps some readers are still doubtful of hash functions. After all, they will argue, we need to store the map $h : K \to I$ somewhere, and we are just shifting the problem again. We cannot use a list because searching is inefficient, and we cannot use an array because it would either take too much memory to store, or too long to insert new pairs (key, index), just as it happened in Sect. 5.1

Here is where the magic lies: *we do not need to store h explicitly.* The hash function h is computed directly by the data representation of k. If k is a name, for example, it is encoded as a string of characters; each character corresponds to an integer between 0 and 255 called its ASCII code: hence a name of n letters is simply a sequence in $\{0, \ldots, 255\}^n$. Any algorithm that takes such a sequence k as input and outputs a single integer $i \in I$ defines a hash function $h : K \to I$. In order for the whole scheme to work, however, we need h to be computed in constant time. This is obtained by: coding efficiently and carefully, and setting a fixed limit to the key lengths (or only considering a fixed amount of information in each key) when computing h.

For example, a very simple (certainly not injective) hash function between names and integers would associate to a character string the sum of the ASCII character codes, modulo the largest possible representable integer. The name "Leo" would be hashed to $76 + 101 + 111 = 288$, because the ASCII codes of 'L', 'e', 'o' are $76, 101, 111$. Thus we could store the record for the key "Leo" in a linear data structure stored at $\sigma(288)$.

5.3 Java implementation

We are going to present two different implementations of a hash table σ. The first one is simpler, and assumes the hash function h is injective. As a consequence, each linear data structure in $\sigma(h(k))$ only ever holds one record (why?): in other words, σ need not be an array of linear data structures that contain records, but only an array that directly contains records. The second implementation does not assume h to be injective, allows *collisions* (i.e. more keys mapped to the same index), and uses singly-linked lists as linear data structures to be stored in the array σ.

5.3.1 A hash table without collisions

We store the Java program in this section within the file `hashingSimple.java`. It contains two classes: the definition of key and record pairs, and the main class. Within the main class `hashingSimple`, we only need two methods: one called `hash()` that defines the hash function h, and `main` function.

5.3.1.1 Keys and records

After the usual initial comment header and the import declaration, we define a pair (key, record), of type `String`^{2} and named `stringPair`, by means of the following class.

```
class stringPair {
    public String key;
    public String record;
    stringPair(String k, String r) {
        key = k;
        record = r;
    }
}
```

5.3.1.2 The main class

The class description is as follows.

```
public class hashingSimple {
   public static int hash(stringPair s, int p);
   public static void main(String[] args);
}
```

Since all methods are static, there is a global object called `hashingSimple`.

5.3.1.3 The hash function

The `hash()` function takes a stringPair `s` and an integer prime `p` and computes the hash function by means of summing the ASCII codes of all characters of `s`, then reducing the total modulo `p`.

```
public static int hash(stringPair s, int p) {
    int h = 0;
    for(int i = 0; i < s.key.length(); i++) {
        h += (int) s.key.charAt(i); // the i-th character of s.key
    }
    return h % p;
}
```

This hash function is a special case of the hash function family in Eq. (5.1). Let the key set K be a large set of integer sequences $(k_1, \ldots, k_\ell)$ having the same length ℓ (pad shorter sequences with zeroes otherwise). Let p be a prime with $p > |U|$. The index set I can be defined as $\{0, \ldots, p-1\}$. It turns out that, for each integer sequence $(a_1, \ldots, a_\ell) \in I^\ell$,

$$h_a(k) = \sum_{j \le \ell} a_j k_j \pmod{p} \tag{5.1}$$

is a valid hash function. Notice that computing $h_a(k)$ is $O(\ell)$, in the worst case, with ℓ a fixed constant. In practice, computing $h_a(k)$ is very fast. The `hash()` function above is Eq. (5.1) with $a = (1, \ldots, 1)$.

5.3.1.4 Main function

The `main` function defines `sigma` then “plays around” with the hash table and function. First, we define a linked list (the pre-confectioned Java parametrizable class `LinkedList`) containing a set of pairs (key, record).

```
public static void main(String[] args) {
   LinkedList<stringPair> KU = new LinkedList<stringPair>();
   KU.add(new stringPair("Leo", "PCC"));
   KU.add(new stringPair("Pierre", "CR2 CNRS"));
   KU.add(new stringPair("Annick", "Prof (Belgium)"));
   KU.add(new stringPair("Andy", "Prof 2eme classe"));
   KU.add(new stringPair("David", "IR2 CNRS"));
   KU.add(new stringPair("Vincent", "CR2 CNRS"));
   KU.add(new stringPair("Nora", "postdoc"));
   KU.add(new stringPair("Hassan", "postdoc"));
   KU.add(new stringPair("Olivier", "Prof"));
   KU.add(new stringPair("Benjamin", "PA"));
```

Next, we initialize an appropriate prime for the hash function.

```
   int p = 13; // a prime >= KU.length
```

Next, we allocate enough memory to the hash table (defined as the array `sigma`).

```
   // hash table: map I->U
   stringPair[] sigma = new stringPair[p];
```

Observe the following Java shorthand syntax for looping over all members of the `LinkedList`. This loop fills the hash table, storing each element `hp` in $\sigma(h_1(\texttt{hp}))$. Since we are assuming h_1 to be injective, we need not concern ourselves with the case where $\sigma(h_1(\texttt{hp})) = \sigma(h_1(\texttt{hp}'))$ for $\texttt{hp}' \neq \texttt{hp}$.

```
    for(stringPair hp : KU) {
        sigma[hash(hp,p)] = hp;
    }
```

Finally, we test the hash table by querying it in several ways.

```
    stringPair inhp = new stringPair("Annick", "");
    stringPair outhp = null;
    inhp.key = "Pierre";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);
    inhp.key = "Andy";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);
    // Leo collides with Olivier
    inhp.key = "Leo";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);
    inhp.key = "Olivier";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);
    // Annick collides with Benjamin
    inhp.key = "Annick";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);
    inhp.key = "Benjamin";
    outhp = sigma[hash(inhp,p)];
    System.out.println("find " + inhp.key + ": " +
        outhp.key + " is a " + outhp.record);

}
```

5.3.2 A hash table allowing for collisions

We store the Java program in this section within the file `hashingChaining.java`. It contains three classes: the definition of key and record pairs (the same given in Sect. 5.3.1.1), a class implementing a singly-linked list, and the main class.

5.3.2.1 A Java implementation of a singly-linked list

For a theoretical description of a singly-linked list, see Sect. 4.2.1. We first give the class description.

```
class singlyLinkedList {
    // each element of the list is a pair of strings
    public stringPair datum;
    // pointer to the next element of the list
    public singlyLinkedList next;
    // default constructor
    public singlyLinkedList();
    // constructor that also initializes the first element
    public singlyLinkedList(stringPair sp);
    // add an element
    public void add(stringPair sp);
    // find the first element whose first string is equal to key
    //   (assumes keys are not duplicated, so at most one element of the list
    //    has a given key)
    public stringPair find(String key);
    // print the list out
    public void print();
}
```

Observe that the class has two constructors, one of which also initializes the first element (this is an example of function overloading). As mentioned above, `stringPair` is defined in Sect. 5.3.1.1.

The default constructor simply sets both attributes `datum` and `next` to `null`

```
    public singlyLinkedList() {
        datum = null;
        next = null;
    }
```

The initializing constructor stores the address of the argument `sp` to `datum`.

```
    public singlyLinkedList(stringPair sp) {
        datum = sp;
        next = null;
    }
```

The following recursive algorithm adds an element to the list. Recursion is used to loop over the list elements to get to the end before adding the new element.

```
    public void add(stringPair sp) {
        if (next == null) {
            // this is the last list element, add a new one
            next = new singlyLinkedList(sp);
        } else {
            // there's a next one, add to that
            next.add(sp);
        }
    }
```

Here follows the method for finding an element in the list.

```
    public stringPair find(String key) {
        stringPair ret = null;
        if (datum != null && datum.key.equals(key)) {
            // we found the correct list element
            ret = datum;
        } else if (next != null) {
            // up to here the key wasn't found, try the next list element
            ret = next.find(key);
        }
        return ret;
    }
```

5.3.1 Exercise
Propose an implementation for the `print` *method in the* `singlyLinkedList` *class.*

5.3.2.2 The main class

The class description is as follows. With respect to `hashingSimple`, notice not every class member is static. We therefore also include a constructor.

```
public class hashingChaining {
   // data attributes
   public int thePrime;           // prime used for hash function
   singlyLinkedList[] sigma;      // the hash table

   // constructors
   public hashingChaining();      // class constructor #1
   public hashingChaining(int p); // class constructor #2

   // hash function
   public static int hash(stringPair s, int p);

   // data structure methods
   public void add(stringPair kr);     // add method
   public stringPair find(String key); // find method

   // main function
   public static void main(String[] args);
}
```

We remark that `hashingChaining` includes `add` and `find` methods, which were not there in `hashingSimple`. This is because, if the hash function is injective, then the hash table is a simple array. Accordingly, add and find are simply the array's own add and find methods. If the hash table `sigma` is an array of singly linked lists, one must first check whether `sigma[i]` is `null` or allocated, and act accordingly.

The constructors are as follows.

```
    public hashingChaining() {
        // a default, largish prime
        // (catastrophe if the table has more elements!)
        thePrime = 5519; // this is a largish prime
    }
    public hashingChaining(int p) {
        thePrime = p;
    }
```

The hash function is the same as in Sect. 5.3.1.3, aside from the implementation detail that the prime is stored as the class attribute `thePrime` instead of being passed to the `hash()` function.

5.3.2 Exercise
Propose a `hash()` *function updated as explained above.*

5.3.2.3 Adding elements to the hash table

Adding an element to the hash table first requires checking whether $\sigma(h(k))$ is allocated or not. If not, a new singly-linked list is stored at $\sigma(h(k))$. Then the record corresponding to k is stored in the list at $\sigma(h(k))$.

```
public void add(stringPair kr) {
    int hval = hash(kr.key);
    if (sigma[hval] == null) {
        // hash table's corresponding entry is empty
        // initialize it with a new one-element list
        sigma[hval] = new singlyLinkedList(kr);
    } else {
        // hash table's corresp. entry already has a list, add to it
        sigma[hval].add(kr);
    }
}
```

5.3.2.4 Finding elements in the hash table

Finding an element k in the hash table σ simply consists in calling the `find` method of the singly-linked list $\sigma(h(k))$.

```
public stringPair find(String k) {
    singlyLinkedList hashList = sigma[hash(k)];
    return hashList.find(k);
}
```

5.3.2.5 Main function

The main function of the `hashingChaining` class has a similar structure to the one given for `hashingSimple` (Sect. 5.3.1.4), but the technical details are different. We use our own `singlyLinkedList` to store the `stringPair`s initially, we dynamically create an object of the `hashingChaining` class, we cannot use the shorthand method to loop over the `singlyLinkedList` object in order to fill the hash table, but have to resort to a longer construct.

```
public static void main(String[] args) {
  // create a new list with some names and qualifications
  singlyLinkedList KU=new singlyLinkedList(new stringPair("Leo", "PCC"));
  KU.add(new stringPair("Pierre", "CR2 CNRS"));
  KU.add(new stringPair("Annick", "Prof"));
  KU.add(new stringPair("Andy", "Prof 2eme classe"));
  KU.add(new stringPair("David", "IR2 CNRS"));
  KU.add(new stringPair("Vincent", "CR2 CNRS"));
  KU.add(new stringPair("Nora", "postdoc"));
KU.add(new stringPair("Hassan", "postdoc"));
KU.add(new stringPair("Olivier", "Prof"));
KU.add(new stringPair("Benjamin", "PA"));

System.out.println("The original data list:");
KU.print();

// the prime should be larger than this list’s length
int p = 13;
hashingChaining h = new hashingChaining(p);

// initialize and fill the hash table
System.out.println("Scanning the list and filling hash table...");
h.sigma = new singlyLinkedList[p];
singlyLinkedList current = KU;
int hval = 0;
while(current != null) {
    // scan the list
    h.add(current.datum);
    current = current.next;
}

// verification of the hash table contents
System.out.println("-----------------------------");
System.out.println("Verifying hash table:");
for(int i = 0; i < p; i++) {
    if (h.sigma[i] == null) {
        System.out.println("hashTable[" + i + "] = _|_");
    } else {
        System.out.print("hashTable[" + i + "] = ");
        h.sigma[i].print();
    }
}
System.out.println("-----------------------------");

// query the hash table
System.out.println("Querying hash table:");

// Pierre has no collisions
String theKey = "Pierre";
stringPair theElement = h.find(theKey);
System.out.println("find " + theKey + ": " +
   theElement.key + " is a " + theElement.record);

// Andy has no collisions either
theKey = "Andy";
theElement = h.find(theKey);
System.out.println("find " + theKey + ": " +
   theElement.key + " is a " + theElement.record);
// Leo collides with Olivier
theKey = "Leo";
theElement = h.find(theKey);
System.out.println("find " + theKey + ": " +
   theElement.key + " is a " + theElement.record);
theKey = "Olivier";
theElement = h.find(theKey);
System.out.println("find " + theKey + ": " +
   theElement.key + " is a " + theElement.record);
```

```
    // Annick collides with Benjamin
    theKey = "Annick";
    theElement = h.find(theKey);
    System.out.println("find " + theKey + ": " +
       theElement.key + " is a " + theElement.record);

    theKey = "Benjamin";
    theElement = h.find(theKey);
    System.out.println("find " + theKey + ": " +
       theElement.key + " is a " + theElement.record);
  }
```

Chapter 6

Trees

Abstract. Root, leaf, direction, depth. Spanning trees. Some mathematical properties of trees. There are 2^{n-2} labelled trees on n vertices. Applications to algebraic graph theory, chemistry, languages and networks.

Mathematically speaking, a *tree* is a connected graph without cycles (see Fig. 6.1).

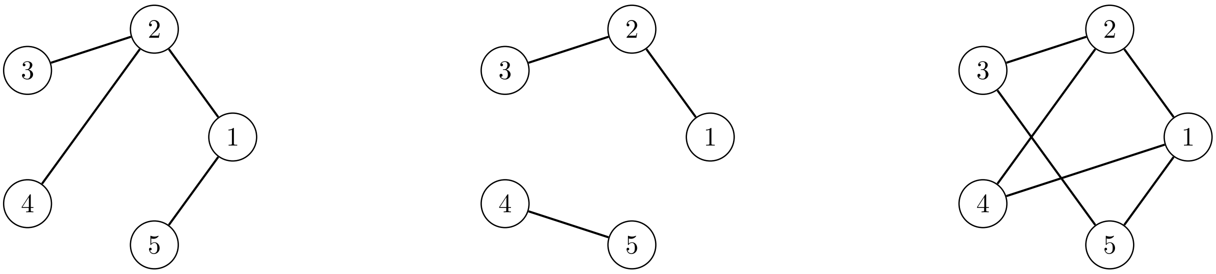


Figure 6.1: A tree (left), a graph which fails to be a tree because disconnected (middle), and a graph which fails to be a tree because of the presence of cycles (right).

6.0.3 Exercise
Compute the dimension of the cycle spaces of the three graphs in Fig. 6.1.

Trees are useful in computer science because they model *tree data structures.* Whenever we refer to a tree data structure, we call vertices *nodes.* Among other things, tree structures are used to implement efficient general-purpose algorithms for searching and sorting sets. The fundamental reason why tree structures may give rise to more efficient algorithms with respect to linear structures is that they encode more relationship information. Whereas a linear data structure only encodes previous and next elements, a tree node might be adjacent to several other nodes. This wealth of information can be used to look for different node iteration orders, giving rise to more efficient ways to structure computing operations.

6.1 Definitions

6.1.1 Roots and direction

A *rooted tree* is a tree with a distinguished vertex, called the *root*. A rooted tree is *directed* if it is a rooted tree that is also a directed graph, and such that arcs are either all directed towards the root, or all directed away from it (see Fig. 6.2).

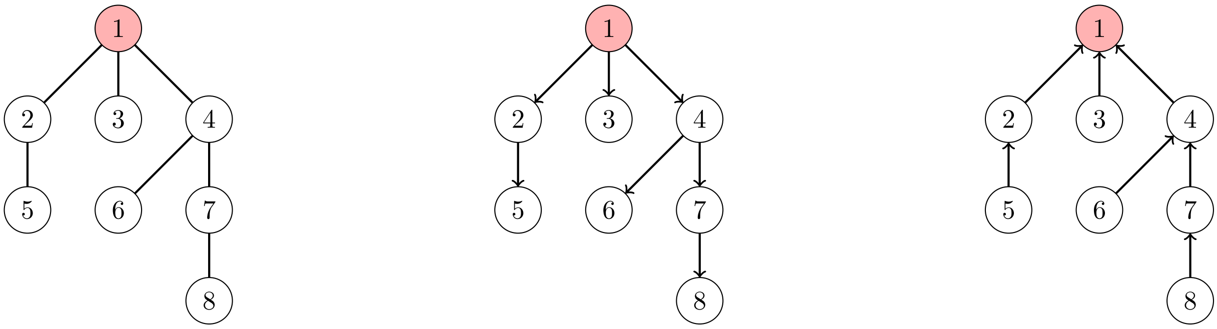


Figure 6.2: A rooted tree (left) and two directed rooted trees (middle, right).

6.1.2 Leafs, depth and height

In a tree, any non-root vertex with degree 1 is called a *leaf*. The *height* or *depth* of a rooted tree is the number of edges in the longest path from a leaf to the root (or from the root to a leaf if the arcs are directed away from the root). For example, the height of the trees in Fig. 6.2 is 3. The *level* of a vertex in a rooted tree is the length of the path from that vertex to the root, plus 1. E.g. vertex 7 in Fig. 6.2 is at level 3.

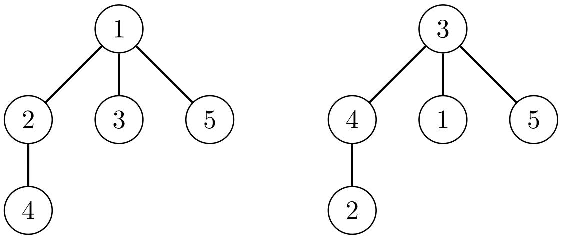
For a tree T, we denote by $L(T)$ the set of its leaf vertices.

6.1.3 Spanning tree

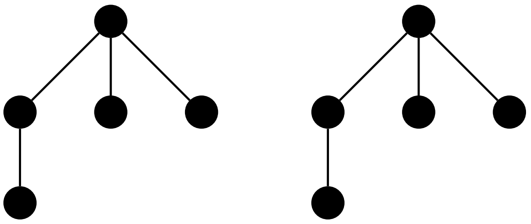
If $G = (V, E)$ is a graph and $H = (V, F)$ is a connected subgraph of G with edges adjacent to each $v \in V$, H is a *spanning subgraph* of G. If $T = (V, F)$ is a tree on the same vertices as V, T is called a *spanning tree* of G. We shall see in Sect. 6.2 below that a spanning tree of G is a minimally connected spanning subgraph of G.

6.1.4 Vertex labels

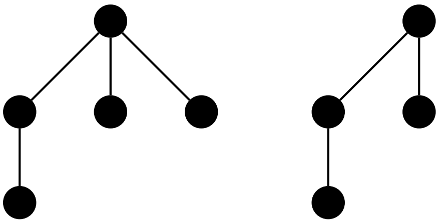
Since a tree is just a graph of special type, it has a set of vertices and one of edges, just like all graphs do. Consider the set V of vertices: each vertex has a name, also called *label* in this context. Thus, for example, we might write $V = \{1, 2, \ldots, n\}$ or $V = \{v_1, v_2, \ldots, v_n\}$ or even $V = \{u, v, w\}$ if V only has three vertices. The point is that every vertex can be distinguished by any other. This means, for example, that the two trees below are different mathematical entities.



In the three on the left, for example, vertex 1 is adjacent to vertex 4, whereas in the tree of the right this is false. However, if we ignore the vertex labels, we obtain two trees that look exactly the same.



This is not to say that every two trees only differ because of the vertex labels. The two trees below are decidedly different, although they have no labels.



In the tree on the left, there is one vertex with degree 3. In the tree on the right, all vertices have degree 2.

What should be clear from these examples is that differences may have to do with the graph structural properties, such as the vertex degree, or simply because the vertex labels are assigned differently. In the first case, we speak of *unlabelled trees*, in the second of *labelled trees.* Mathematically speaking, an unlabelled tree can be defined as a chosen representative of an equivalence class containing all trees that are “structurally the same”.

6.2 Basic properties

6.2.1 Number of edges

6.2.1 Proposition
A tree on n vertices has $n-1$ edges.

Proof. By induction: the case with 1 vertex is trivial (there can be no edge). Now take a tree with $n-1$ vertices and assume it has $n-2$ edges. Add an n-th disconnected vertex v to the graph, and consider how it can be connected to the tree. Either it is connected to only one vertex u in the tree, or to more than one. In the former case the new edge does not belong to any new cycle, since the degree of u is one.

Hence the new graph is connected and without cycles: in short, it is a tree with n vertices and $n-1$ edges. In the latter case, u must be connected to at least two vertices $u \neq w$ in the tree. Since the tree is connected, there is a path p with source u and destination w. But then the two edges $\{u, v\}$ and $\{v, w\}$ and p form a cycle. So the new graph cannot be a tree. □

6.2.2 Connectivity

6.2.2 Corollary
Removing an edge from a tree disconnects the graph.

The proof should be obvious.

6.2.3 Acyclicity

6.2.3 Corollary
Adding an edge to a spanning tree of a graph G determines a unique cycle in G.

6.2.4 Exercise
Prove Cor. 6.2.3.

6.2.4 Edge swapping operation

6.2.5 Corollary
Given a graph G and a spanning tree T, if we add an edge e not in the tree to T then remove an edge f in the unique cycle determined by e, we obtain a different spanning tree of G.

This corollary is the basis for many algorithms that search within the set of all spanning trees of a graph: pick an edge e of the graph which is not already in the tree, add it to the tree, identify the unique cycle, pick an edge $f \neq e$ in the cycle, remove it from the tree, and you get a different spanning tree of the graph.

6.2.6 Exercise
Prove that all spanning trees of G can be constructed by means of "edge swaps" as described above, starting from any spanning tree.

6.3 The number of labelled trees

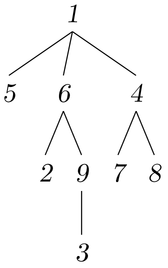
A very famous result of A. Cayley states that there are n^{n-2} possible different labelled trees having n vertices. We are going to prove this result using a method first proposed by Prüfer: we shall consider the set of all trees on a set V of n vertices and construct a bijection on the set of all sequences of $n-2$ elements of V. Since the latter has n^{n-2} elements (why?), the result will be proved. The bijection is constructed explicitly: we provide two algorithms, one for mapping trees into $(n-2)$-sequences, and one for mapping $(n-2)$-sequences into trees. These sequences are called *Prüfer sequences.*

6.3.1 Mapping trees to sequences

The first algorithm works by progressively removing leaf vertices of the given tree T and storing their parents; naturally, when a "layer" of leaves attached to a vertex v is removed, then v becomes a leaf itself, and so it will be dealt with successively. The leaf vertices are removed in a certain order (i.e. the minimum leaf vertex label is chosen for removal): this induces an order on the sequence of stored parents. At all times, we denote by $L(T)$ the set of leaf vertices of T (of course, as T is updated, $L(T)$ changes).

```
for $k \in \{1, \ldots, |V| - 2\}$ do
    $v = \min L(T)$;
    let $e$ be the only edge incident to $v$;
    let $t_k \neq v$ be the other node incident to $e$;
    $T \leftarrow T \smallsetminus \{v\}$;
end for
return $t = (t_1, \ldots, t_{|V|-2})$
```

6.3.1 Example
For the tree below:



the Prüfer sequence is $(6, 9, 1, 4, 4, 1, 6)$.

6.3.2 Mapping sequences to trees

In order to map a Prüfer sequence $t = (t_1, \ldots, t_{n-2})$ back to its corresponding tree, we scan $V \smallsetminus t$ from smallest to largest vertex label ℓ, we pair vertex ℓ with the first component t_1 of t, add $\{t_1, \ell\}$ as an edge to the tree, and we remove t_1 from t and ℓ from V. Two remarks: as we remove t_1 from t, the next component will be called t_1 at the successive iteration; and $V \smallsetminus t$ might also acquire further elements during the algorithm, namely those that no longer appear in t. When t is the empty sequence, it can be proved that $V \smallsetminus t$ is the last edge to be added to the tree.

```
let $T = \varnothing$ be the empty tree;
while $t \neq \varnothing$ do
    let $\ell = \min V \smallsetminus t$;
    add the edge $\{\ell, t_1\}$ to $T$;
    remove $t_1$ from $t$, and renumber the remaining elements so that the first is still called $t_1$;
    remove $\ell$ from $V$;
end while
at this point, $V \smallsetminus t = \{u, v\}$: add it as an edge to $T$.
```

By the two algorithms above, there are as many trees as there are $(n-2)$-sequences with entries in $\{1, \ldots, n\}$. Since the number of such sequences is n^{n-2}, we have the following result.

6.3.2 Theorem
There are n^{n-2} *different labelled trees with* n *vertices.*

6.3.3 Exercise
Implement the two algorithms in this section using Java.

6.4 Applications

Trees have several applications as data structures in computer science: we shall see the main ones in Part III. Here we list some direct applications of trees to graph theory, chemistry, linguistics and networks.

6.4.1 Finding a basis of the cycle space

In Sect. 3.4.7 we introduced the concept of the cycle space, i.e. the vector space of all the cycles of a graph over the finite field $\mathbb{F}_2$. The sum of two cycles is defined as the XOR operation over the cycle-edge incidence vectors corresponding to the two cycles, and denoted by the binary operator $\oplus$.

Here we show that any spanning tree of a graph induces a basis for the cycle space. Let $T = (R, F)$ be a spanning tree of a connected graph $G = (V, E)$. This partitions E into tree edges F and non-tree edges $E \smallsetminus F$. The latter are also called *chords*. By Cor. 6.2.3, every chord determines a unique cycle of G, namely the chord $\{u, v\}$ union the unique path on T from u to v. Incidentally, since there are $m = |E|$ edges, $n-1$ tree edges (by Prop. 6.2.1), every spanning tree induces exactly $m-n+1$ cycles.

6.4.1 Lemma
Let $\Gamma = \{\gamma_1, \ldots, \gamma_{m-n+1}\}$ be the cycles induces by a spanning tree T. Then Γ is linearly independent over $\mathbb{F}_2$.

Proof. Let $a_1, \ldots, a_{m-n+1} \in \{0, 1\}$ such that:

$$\bigoplus_{i=1}^{m-n+1} a_i \gamma_i = 0, \qquad (6.1)$$

where 0 stands for the empty cycle. If any $a_i = 1$, then the incidence vector of the cycle γ_i has a 1 in the edge column corresponding to the chord that determines γ_i. In order for this 1 to become a 0 in Eq. (6.1), we need at least another cycle spanned by Γ to have a 1 in that edge column, for $b = 1$ is the only element of $\mathbb{F}_2$ that satisfies $1 \oplus b = 0$. However, γ_i is by definition the only cycle in Γ that has a 1 in the edge column corresponding to that chord. Hence $a_i = 0$ for every $i \leq m-n+1$, which implies that Γ is a linearly independent set. □

By Lemma 6.4.1 and Thm. 3.4.2, we conclude that the set of cycles determined by the chords of any spanning tree form a basis of the cycle space. Cycle bases corresponding to a spanning tree are called *fundamental*.

6.4.2 Chemical trees

Until the mid-19th century, scientists thought molecules were completely defined by their atomic formula, e.g. paraffins would be $\mathrm{C}_k\mathrm{H}_{2k+2}$. It was then noticed that different bond relations between atoms gave rise to substances having different properties. Different substances with the same atomic formulæ are called isomers (see Fig. 6.3). We remark that different atoms have different properties: carbons, for example, can be incident to exactly 4 edges, whilst hydrogens, can only be incident to exactly 1 edge (we also say that carbons have *valence* 4 and hydrogens have valence 1).

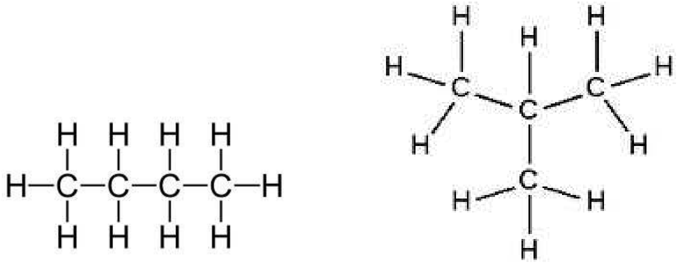


Figure 6.3: Butane (on the left) and isobutane (on the right).

Some graph structures of specific properties are also known, e.g. paraffins are known to have tree-like bond relations. Thus, a natural question arose: how many different isomers with certain graph properties shared a same chemical formula? From a purely formal (rather than chemical) point of view, listing all paraffins is equivalent to listing all trees with a given number n of vertices. This can be done by listing all Prüfer sequences and transforming them to the corresponding trees (see Sect. 6.3.2).

6.4.2 Exercise
Devise and implement an algorithm for listing all trees of n carbons and hydrogens. Make sure the corresponding valences are respected. List all trees of carbons and hydrogens have 4,5 and 6 vertices.

6.4.3 Trees and languages

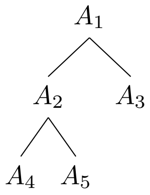
A language can be either formal or natural. Formal languages follow a well-defined set of precise syntactical rules for producing valid sentences. Programming languages, for example, are formal languages (see Sect. 1.3). Moreover, in a formal language each valid sentence can have at most one meaning.

Languages that are not formal are called natural, e.g. English and French are natural languages. The validity of a sentence in a natural language is a fuzzy notion. It is clear that certain sentences are invalid (e.g. “me you drink I dog”), and it is also clear that certain sentences are valid (e.g. “my name is Leo”), but there are sentences that may be valid or invalid depending on a context which goes beyond the language itself (e.g. “would you please… no, leave it” has an auxiliary verb — “would” — that lacks a main verb; and yet, in a context of hesitation, it makes sense), or other sentences that are syntactically ambiguous (e.g. “Ibis redibis non morieris in bellum”, which can be translated as “you will go and come back, you won’t die in war” as well as “you will go but not come back, and die in war”).

6.4.3.1 Trees and recursion

Trees are the best type of structure to model a recursive behaviour. Each recursive action will refer to itself (or variations of itself) one or more times during its existence. This creates a hierarchy of actions, related by reference, which is best represented as a tree.

6.4.3 Example
In the example below, an action A is declined into five variants $A_1, \ldots, A_5$: A_1 refers to A_3 and A_2, which itself refers to A_4 and A_5.



6.4.3.2 Syntax of formal languages

A sentence of a formal language is defined to be valid in a constructive way. Consider for example the formal language of all mathematical expressions consisting of constants $c \in \mathbb{R}$, variable symbols x_i for any $i \in \mathbb{N}$, the binary operators sum, difference, multiplication, division, power, and the unary operators unary minus, logarithm and exponential. Valid sentences in this language are called expressions.

6.4.3.2.1 Construction of valid sentences We define expressions as follows:

- constants and variable symbols are expressions,
- for any two expressions f and g, $f+g, f-g, fg, f/g, f^g$ are expressions,
- for any expression f, $-f, \log(f), \exp(f)$ are expressions,
- no other sentence, aside from those obtained by the rules above, is an expression.

The concept of expression is used recursively to construct ever more complicated expressions. The last rule bars any other sentence from being valid.

6.4.3.2.2 Recognition of valid sentences Now suppose we are given a sentence, and we have to decide whether it is an expression or not. If we are able to "break it down" into sub-expressions so as to obtain the recursive process that led to its construction, then it is valid, otherwise, because formal languages are not ambiguous, it is not. Accordingly, we write the rules above in a slightly different form.

- an expression is either a term, or a sum of an expression with a term, or a difference of terms
- a term is either a power, or a multiplication of a term by a power, or a division of powers
- a power is either a function, or a function to the power of a function
- a function is either a leaf, or the negative of an expression, or the logarithm or exponential of an expression, or simply an expression within brackets
- a leaf is either a constant or a variable symbol.

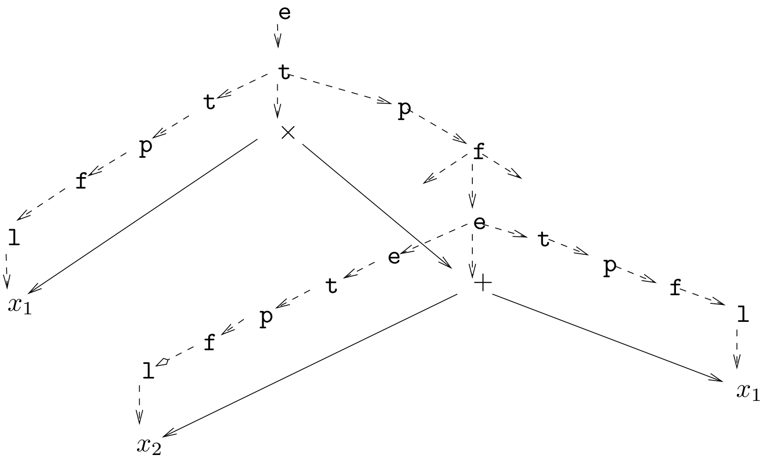
Here, the names *term*, *power*, *function*, *leaf* denote a hierarchy level in the recursive application of our validity verification method. Given a character string containing a sentence, we try and match the rules above, starting from operators of lowest precedence (sums and differences), to those of highest precedence (unary operators). The set of rules above are the *grammar* of the formal language of mathematical expressions. It is written succinctly as follows:

$$\left.\begin{array}{lclclclclcl}
\mathtt{e} & : & \mathtt{t} & | & \mathtt{e}+\mathtt{t} & | & \mathtt{t}-\mathtt{t} & & & & \\
\mathtt{t} & : & \mathtt{p} & | & \mathtt{t}\times\mathtt{p} & | & \mathtt{p}\div\mathtt{p} & & & & \\
\mathtt{p} & : & \mathtt{f} & | & \mathtt{f}\uparrow\mathtt{f} & & & & & & \\
\mathtt{f} & : & \mathtt{l} & | & -(\mathtt{e}) & | & \log(\mathtt{e}) & | & \exp(\mathtt{e}) & | & (\mathtt{e}) \\
\mathtt{l} & : & \multicolumn{9}{l}{x_i \quad (1 \le i \le n) \quad | \quad c \quad (c \in \mathbb{R}).}
\end{array}\right\} \tag{6.2}$$

where the *grammar labels* `e`, `t`, `p`, `f`, `l` obviously denote expression, term, power, function and leaf.

The process of recognizing the validity of a given sentence, also called *parsing*, naturally yields a tree (called the *derivation tree* or *parse tree*), whose vertices are labelled by the grammar labels. The derivation tree subsumes the so-called *expression tree*, which relates operators and operands, as shown in Example 6.4.4.

6.4.4 Example
Consider the expression $x_1(x_1 + x_2)$. The whole expression is initially parsed as a term, which is parsed as a multiplication of two terms: the first, x_1, is parsed as a term, then as a power, then as a function, and finally as a leaf. The second, $(x_1 + x_2)$, is parsed as a power, then as a function, then, shedding its brackets, as an expression, which is itself parsed as a sum of an expression and a term, and so on. The derivation tree is shown below as a directed tree with dashed arcs. The subsumed expression tree is shown as a directed tree with whole arcs.



We remark that the name "leaf" assigned to the grammar label `l` now makes sense, insofar as leaves label the leaf vertices of the derivation tree.

6.4.3.3 Semantics of formal languages

Although this fails to even begin to skim the surface of a very complicated story, the semantics (read: "meaning") of a formal (valid) sentence is an assignment of certain entities to the variable symbols. The entities are usually sets, or numbers, or other mathematical abstractions. But this need not be so. Rudolf Carnap defined formal languages in several scientific fields, including physics and chemistry. In fact, alternative semantics allow a formal system to describe a reality other than mathematical.

6.4.3.4 Syntax of natural languages

Noam Chomsky, in [7], attempted to extend the use of derivation trees to parse natural language sentences, as shown in Fig. 6.4.

6.4.5 Example
The sentence (S) "sincerity may frighten the boy" in Fig. 6.4 is parsed in the noun phrase (NP), the auxiliary (Aux) and the verb phrase (VP). Recursively, NP is simply parsed into a noun (N); Aux into a modal (M), and VP into a verb (V) and a new noun phrase (VP), which is itself parsed into a determiative article (Det) and a noun (N).

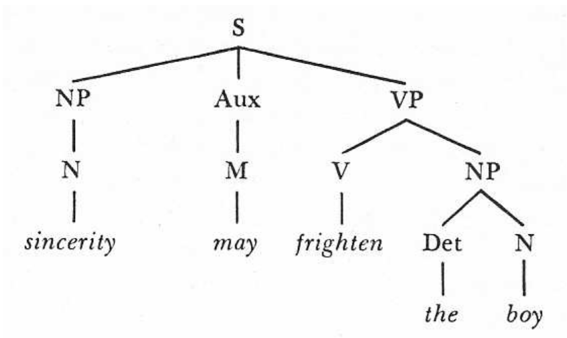


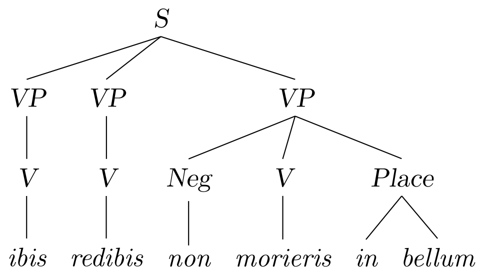
Figure 6.4: A derivation tree proposed by Chomsky [7, p. 65] for the sentence "sincerity may frighten the boy".

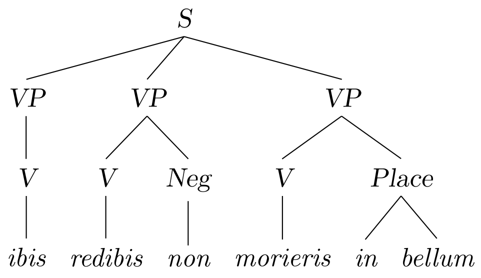
6.4.6 Exercise
Try writing a natural language grammar for a subset of a natural language of your choice.

6.4.3.5 Semantics of natural languages

Richard Montague, a student of Alfred Tarski's, borrowed the tools of axiomatic set theory, especially recursion, to address the problem of ambiguity in natural languages [18]. An ambiguous sentence is one that has multiple derivation trees, see Example 6.4.7.

6.4.7 Example
Consider the latin sentence ibis redibis non morieris in bellum cited at the beginning of Sect. 6.4.3. It has two different parse trees:





By assigning the common latin meanings to the leaves ibis, redibis, non, morieris, in, bellum, there follow two different interpretations of these two trees. We remark that the Latin syntax makes the tu (you) noun phrase implicit in the verb phrases; and each VP in a single sentence is linked by the other by an implicit et (and).

From a syntactical point of view, Montague devised ways to modify natural grammars so that ambiguities would be restricted to leaves. He obtained this by allowing leaves to stand for whole sets of (ambiguous) subtrees. For a given ambiguous sentence, this somehow yielded unique grammar trees "modulo ambiguity".

Some ambiguous sentences resisted this approach. To deal with them, Montague postulated the existence of appropriate semantics, so that the meaning assigned to the leaves would make at most one derivation tree true, and all the other ambiguous ones false. This is similar to the Sybilla telling the soldier "ibis redibis non morieris in bello" *after* the soldier was safely back from the war, rather than before: by its very presence, the soldier made one of the two ambiguous trees yield a false sentence, and the other a true one. In very poor words, this amounts to the mathematical formalization of the concept of understanding a potentially ambiguous sentence by a given context.

6.4.4 Trees in networks

A *network* is simply another name for "graph", used in certain engineering and scientific communities.

6.4.4.1 Commodity networks

Consider a set of geographical sites that need to be connected in order to exchange information, or electrical power, or any other commodity. Any site can serve as proxy to route the commodity between any site pair. Laying a connection infrastructure, be it eletrical or optical wires, or pipes, incurs a cost: this could be unitary (each communication link has a fixed cost), or proportional to the length of each pairwise segment. We initially consider the set of potential links between pairs of sites that can be geographically linked, and model this as a graph $G = (V, E)$, where V is the set of sites and E the set of potential links (such a graph is also known as a *network*). We want to find a subgraph of G that offers point-to-point connectivity at minimum cost, while serving all sites. Since all costs are nonnegative, this is a minimally connected subgraph of G, which, by Sections 6.1.3 and 6.2, corresponds to a spanning tree of G of minimum cost (see Fig. 6.5).

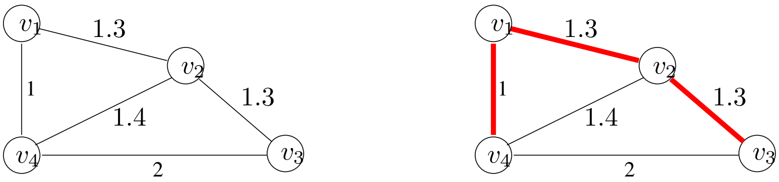


Figure 6.5: A graph with its minimum spanning tree.

6.4.4.2 Distance networks

Spanning trees are also useful for storing compact encodings of massive data sets. Consider for example a long list $V = (v_1, \ldots, v_n)$ of long binary sequences $v_i = (v_{i1}, \ldots, v_{im})$ of the same length m.

First, we compute all *Hamming distances* h_{ij} between every pair $\{v_i, v_j\}$, which we use to label the edges of a complete graph on V, also known as *distance network*. We remark that the Hamming distance between two bit sequences of the same length is the number of flips necessary to transform a sequence into the other, e.g. 01100 and 00110 have Hamming distance 2, since we must flip the second and fourth bits to obtain a sequence from the other.

Netx, we find the minimum cost spanning tree T in the distance network, and enrich each edge $\{v_i, v_j\}$ in T with the sequence $K_{ij} = (k_1, \ldots, k_{h_{ij}})$ of element indices such that flipping all v_{ik_ℓ}, for $\ell \in \{1, \ldots, h_{ij}\}$, yields v_j (and vice versa: why?). This means that, if we store v_i, we only need to store the sequence K_{ij} in order to retrieve v_j. Since K_{ij} is shorter than v_j, we gain in storage space.

Finally, we store the following information:

- any $v_0 \in V$;
- the tree T with the edge information (h_{ij}, K_{ij}) for each edge $\{v_i, v_j\}$.

This guarantees that every $v \in V$ can be reconstructed using v_0 and the information in T. Since T has minimum cost, $\sum_{i,j} |K_{ij}|$ is minimum, which means that this is the most compact possible encoding of V.

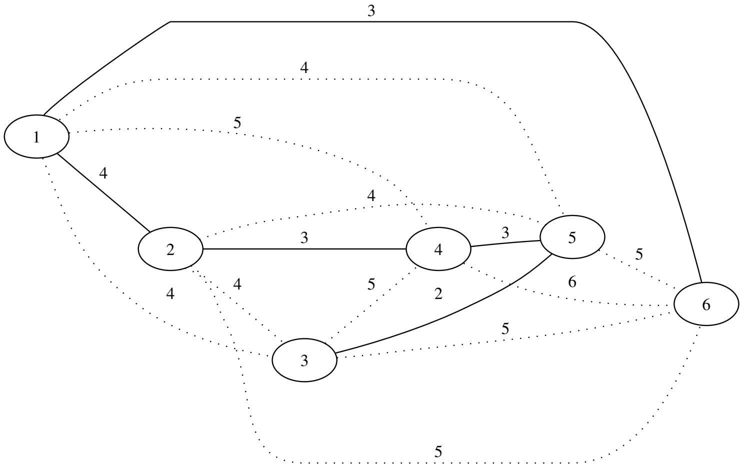
6.4.8 Example
Consider the following set V of bit sequences:

1. `011100011101`
2. `101101011001`
3. `110100111001`
4. `101001111101`
5. `100100111101`
6. `010101011100`,

with pairwise Hamming distance given by the following matrix (we only report the upper right triangle, since the lower left is symmetric (why?)

	1	2	3	4	5	6
1	0	4	4	5	4	3
2	-	0	4	3	4	5
3	-	-	0	5	2	5
4	-	-	-	0	3	6
5	-	-	-	-	0	5
6	-	-	-	-	-	0.

The minimum spanning tree has cost 5:



Part III

Algorithms

Chapter 7

Recursive algorithms

ABSTRACT. Motivations for using recursion. Recursion as a loop. Examples of recursive algorithms (with some Java implementations): enumerating permutations and the Hanoi tower. Recursion in logic: Gödel's incompleteness theorem.

A recursive algorithm is one that includes one or more recursive procedures. A procedure is recursive if one of its instructions is a call to itself. This may not sound so convincing, except that procedures are implemented as functions in a programming language, and hence take input arguments: a recursive function call will almost certainly take a different set of argument values than the calling function took.

7.1 Motivations

7.1.1 Proving program properties

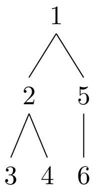
One of the main points in favour of using recursion is that it usually makes it easy to prove that a recursive function actually does what it is supposed to do. This is related to mathematical induction: it suffices to check that an induction start and an induction step hold, and the property is proved for all values of a certain discretely changing argument.

7.1.1 Exercise
Using mathematical induction, prove that the following recursive program computes $n!$.

```
function $f(n)$ {
  if $(n = 0)$ then
    return 1
  end if
  return $n \times f(n-1)$
}
```

7.1.2 Expressing certain procedures naturally

Another strong point for recursion is that it allows to write certain programs more "naturally". Try programming a computer to explore the tree below so that it follows the vertices in the natural order $1, 2, 3, 4, 5, 6$.



Notice that in this case the natural order arises from exploring the tree depth-first, starting from vertex 1. This is called *depth-first search* (DFS) and will be discussed in later chapters.

7.1.2.1 Encoding the tree

Since a tree is a graph, we encode it using an adjacency list (see Sect. 4.1.1.1) A, defined as follows:

A_1: $A_{11} = 2$, $A_{12} = 5$
A_2: $A_{21} = 3$, $A_{22} = 4$
A_3: $\varnothing$
A_4: $\varnothing$
A_5: $A_{51} = 6$
A_6: $\varnothing$,

so that A_{ij} is the vertex label of the j-th child of vertex i of the tree.

7.1.2.2 A code with limited scope

A naïf student might initially code something like:

```
int $a = 1$;
print $a$;
for (int $z = 1$ to $|A_a|$) do
  int $b = A_{az}$;
  print $b$;
  for (int $y = 1$ to $|A_b|$) do
    int $c = A_{by}$;
    print $c$;
    ...
  end for
end for
```

For the given tree, this might work. But change the tree, and the code stops working: the number of loops depends on the number of vertices, and it is "hard-coded" in the pseudocode. Ideally, our codes should work for all similarly structured inputs, in this case all trees.

7.1.2.3 Algorithms and problems

More formally, we recall that a problem is a set of inputs with relative answers (see Sect. 1.4.2); here, the relevant decision problem is "given a tree on n vertices numbered 1 to n, does DFS exploration yield the order $1, \ldots, n$?" The possible inputs (also called *instances* of the problem) are the pairs (n, T) where $n \in \mathbb{N}$ and T is a tree on n vertices. As was said in Sect. 1.4.3, an algorithm is supposed to be able to solve a problem (i.e. a whole infinite set of instances) rather than a single instance, or a handful thereof, taking each individual instance as input. So the code in Sect. 7.1.2.2 is no good.

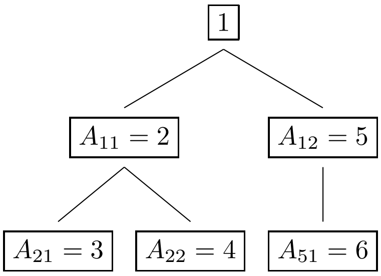
7.1.2.4 Recursion saves the day

We can very naturally model the depth-first nature of the tree exploration using recursion, by calling $f(1)$ where f is the function below.

```
function $f$(int $\ell$) {
  print $\ell$;
  for (int $i = 1$ to $|A_\ell|$) do
    $f(A_{\ell i})$;
  end for

}
```

If we trace the input argument ℓ to the function f below, we obtain the following function call tree:



which, we remark in order to emphasize how naturally recursion can model a DFS, is the same as the original input tree.

7.1.2.5 Back to iteration

To those students who might think that recursion is necessary to express certain types of programs, we explicitly say *this is not the case*! It suffices to introduce a stack and change recursive calls back to their internal form (pushing addresses on the stack, and then popping them in order to jump back to the calling functions, see Sect. 4.4.2). Here are the actions that the recursive algorithm of Sect. 7.1.2.4 performs on the given tree.

1. $\ell = 1$; print 1
2. $|A_1| = 2$; $i = 1$
3. call $f(A_{11} = 2)$ [push $\ell = 1$]
4. $\ell = 2$; print 2
5. $|A_2| = 2$; $i = 1$
6. call $f(A_{21} = 3)$ [push $\ell = 2$]
7. $\ell = 3$; print 3
8. $A_3 = \varnothing$
9. return [pop $\ell = 2$]
10. $|A_2| = 2$; $i = 2$
11. call $f(A_{22} = 4)$ [push $\ell = 2$]
12. $\ell = 4$; print 4
13. $A_4 = \varnothing$
14. return [pop $\ell = 2$]
15. return [pop $\ell = 1$]
16. $|A_1| = 2$; $i = 2$
17. call $f(A_{12} = 5)$ [push $\ell = 1$]

18. $\ell = 5$; print 5
19. $|A_5| = 1$; $i = 1$
20. call $f(A_{51} = 6)$ [push $\ell = 5$]
21. $\ell = 6$; print 6
22. $A_6 = \varnothing$
23. return [pop $\ell = 5$]
24. return [pop $\ell = 1$]
25. return; end

7.1.2 Exercise
Write an iterative version of the recursive algorithm in Sect. 7.1.2.4. Make sure it works on every possible input.

7.2 Iteration and recursion

Compare the following two codes:

```
while (true) do
  print "Leo";
end while
```

```
function f() {
  print "Leo";
  f();
}
f();
```

Both programs yield the same infinite loop that prints “Leo” on the screen without ever ending. The important point is that recursion is a form of loop. Consider the following iterative code for computing factorials:

```
input $n$;
$r = 1$
for ($i = 1$ to $n$) do
  $r = r \times i$
end for
output $r$
```

and compare it with the recursive code in Exercise 7.1.1. Whereas the iterative code uses a loop and assignments, the recursive version only uses recursion and nothing else. It might appear strange at first sight: assignments are the way computers have to write to memory — can recursion stand in for memory? Not quite: the subtle point is that recursion needs the OS to implement function calls, and that these, in turn, need a stack to work (see Sect. 4.4.2). Recursion is implicitly making use of the stack memory during the `return` call: the returned values are stored on the stack, where the calling function can access them. Insofar as universality is concerned, machines with two stacks and an alphabet of only one symbol are known to be UTMs.

7.2.1 Terminating the recursion

The recursive procedure in the previous section did not terminate, as the recursive function $f()$ called itself without arguments and returned no value: all things being equal, there was no reason why the next call should be any different from the previous.

7.2.1 Exercise
Prove formally, using mathematical induction, that a function $f()$ without input arguments and return values, that just calls itself without doing anything else, generates an infinite loop.

Now consider a function $f(n)$, where $n \in \mathbb{N}$, and suppose that the implementation of f calls itself over a different argument value, say $n-1$. Suppose also that, before calling itself, f implements a test to ascertain that $n > 0$, whereas f terminates without recursion if $n = 0$. Then we can conclude that f does not yield an infinite loop.

7.2.2 Exercise
Prove the last assertion formally.

Typically, a general schema for recursive function implementations is the following.

```
if $n$ is a "base case" then
  compute $f(n)$ directly, do not recurse
else
  recurse on $f(i)$ with some $i < n$
end if
```

If we plot the values taken by n against the level of recursion (this can be seen as stack size), in order for f to terminate we need a graph like the one in Fig. 7.1 (left). Actually, it can be much crazier, it just suffices that it hits the "base case" in finite time (Fig. 7.1, right).

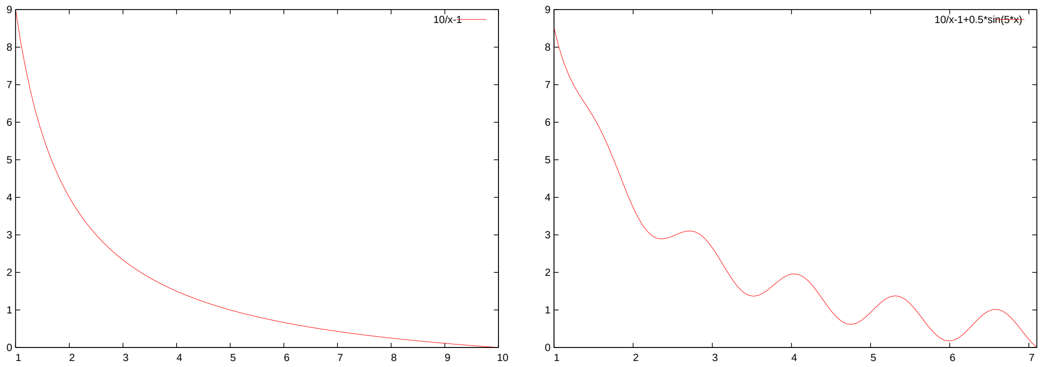


Figure 7.1: Input argument n of a recursive procedure in function of recursion level.

7.3 Listing permutations

A *permutation* of n elements is a function that maps an n-element sequence to another n-element sequence having the same elements as the first, but in a different order. More formally, a permutation is a bijection from a finite set V to itself, i.e. an automorphism on V.

In this section we show how recursion can help us list all possible permutations of n elements (for any given n).

7.3.1 Some background material on permutations

We denote a permutation π on the set $[n] = \{1, \ldots, n\}$ by listing the action of the permutation on each element on $[n]$, for example:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

sends $1 \to 2$, $2 \to 3$, $3 \to 4$ and $4 \to 1$.

Sometimes the first line of the permutation representation above is skipped, and we only denote π by the second line.

7.3.1.1 Product of permutations

The product of π by the permutation $\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{smallmatrix}\right)$, defined by applying σ first and π later, and denoted as $\pi\sigma$, has the following effect:

$$\begin{array}{ccc} 1 \xrightarrow{\sigma} & 4 & \xrightarrow{\pi} 1 \\ 2 \xrightarrow{\sigma} & 3 & \xrightarrow{\pi} 4 \\ 3 \xrightarrow{\sigma} & 2 & \xrightarrow{\pi} 3 \\ 4 \xrightarrow{\sigma} & 1 & \xrightarrow{\pi} 2, \end{array}$$

i.e. it is the permutation:

$$\pi\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 \end{pmatrix}.$$

We remark that the product of permutations is a composition of bijections. Since the composition of two bijections on the same set is another bijection on that set, the product of two permutations is still a permutation.

7.3.1 Exercise
Prove that the compositions of two bijections on V is another bijection on V.

7.3.1.2 Group structure

We now take a more abstract look at the permutation product. This product is a binary operator between permutations: mathematically speaking, it maps pairs of permutations into another permutation. Whenever a k-ary operator maps V^k to a set $U \subseteq V$, we say that the operator is *closed* or that the set V is closed with respect to that operator.

7.3.2 Exercise
Proving that the product of permutations is associative is easy but long (and, for me, also tedious, since I already did this when I was a student): amuse yourself and do it as an exercise.

The identity of the permutation product is the permutation $e = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{smallmatrix}\right)$, and the inverse of each permutation is obtained by simply "reversing the arrows": if a permutation π sends i to j, then π^{-1} sends j to i. In other words, this means that π^{-1} sends $\pi(i)$ to i, and therefore that $\pi^{-1}(\pi(i)) = i$ for all $i \in [n]$, which implies that $(\pi^{-1}\pi)(i) = i$, i.e. that $\pi^{-1}\pi = e$. Thus, the set of permutations of n elements form a group under the permutation product. This group, called the *symmetric group of order n*, is denoted by S_n.

7.3.3 Exercise
Prove that $|S_n| = n!$.

We remark that, by a theorem of Cayley's, any finite group is isomorphic to a subgroup of S_n for some n.

7.3.1.3 Cycle notation

A *cycle permutation* (or simply a *cycle*) is a permutation $\pi \in S_n$ with a sequence $(v_1, \ldots, v_\ell)$ such that $\pi(v_i) = v_{i+1}$ for all $i < ell$ and $\pi(v_\ell) = v_1$, and $\pi(v) = v$ for all other elements $v \in V \smallsetminus \{v_1, \ldots, v_\ell\}$. Informally, the action of π on V is described graphically in Fig. 7.2 for a case where $\ell = 6$.

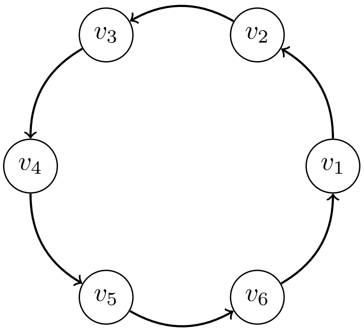


Figure 7.2: The action of a cycle permutation.

Cycles allow a more compact way of writing permutations. The permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix},$$

for example, only swaps 1 and 2 but still takes 9 pairs of integers to write down: this is wasteful. But we can easily recognize that π is the cycle of length 2 sending $1 \to 2$ and $2 \to 1$ and fixing all the other integers. We therefore write π more simply as $(1, 2)$. In general, a cycle permutation sending $\pi(v_i)$ to v_{i+1} for all $i < \ell$ and $\pi(v_\ell)$ to v_1 is denoted by its defining sequence $(v_1, \ldots, v_\ell)$.

Let $\pi = (v_1, \ldots, v_h)$ and $\sigma = (u_1, \ldots, u_k)$ be two cycles. If these two cycles have no common elements, then their product $\pi\sigma$ simply sends $v_i \to v_{i+1}$ for $i < h$, $u_i \to u_{i+1}$ for $i < k$, $v_h \to v_1$ and $u_k \to u_1$. In other words, the actions of π and σ are *disjoint*.

7.3.4 Exercise
Prove that, if π, σ are two disjoint cycles, then $\pi\sigma = \sigma\pi$. Show that this property is lost in general if the two cycles are not disjoint.

We write the product of two disjoint cycles by simply juxtaposing the two cycles, namely:

$$(v_1, \ldots, v_h)(u_1 \ldots, u_k).$$

If the cycles π, σ have some common elements, this analysis no longer holds. For example, if $\pi = (1, 2, 3)$ and $\sigma = (1, 2)$, $\pi\sigma$ has the following effect (we apply σ first and π later): $1 \to 2 \to 3$, $2 \to 1 \to 2$, $3 \to 3 \to 1$, which we can write as $(1, 3)$. What is true, however, is that any product of non-disjoint cycles can be written as a product of (possibly different) disjoint cycles, and moreover that any permutation can be written as a product of disjoint cycles in a unique way apart from the order of the factors (see [8], p. 59).

7.3.2 The inductive step

We represent permutations by means of the second row of the $2 \times n$ matrix notation introduced in Sect. 7.3.1.

The inductive step of our reasoning can be explained by means of an example. Suppose $n = 4$ and we are able to produce a complete list of all permutations of 3 elements:

$$(1, 2, 3), (1, 3, 2), (3, 1, 2), (3, 2, 1), (2, 3, 1), (2, 1, 3).$$

We write each of these permutations four times, and write the number 4 in every possible position, as follows.

$$\begin{array}{cccccccccc}
1 & 2 & 3 & \mathbf{4} & \quad & 3 & 2 & 1 & \mathbf{4} \\
1 & 2 & \mathbf{4} & 3 & & 3 & 2 & \mathbf{4} & 1 \\
1 & \mathbf{4} & 2 & 3 & & 3 & \mathbf{4} & 2 & 1 \\
\mathbf{4} & 1 & 2 & 3 & & \mathbf{4} & 3 & 2 & 1
\end{array}$$

$$\begin{array}{cccccccccc}
1 & 3 & 2 & \mathbf{4} & \quad & 2 & 3 & 1 & \mathbf{4} \\
1 & 3 & \mathbf{4} & 2 & & 2 & 3 & \mathbf{4} & 1 \\
1 & \mathbf{4} & 3 & 2 & & 2 & \mathbf{4} & 3 & 1 \\
\mathbf{4} & 1 & 3 & 2 & & \mathbf{4} & 2 & 3 & 1
\end{array}$$

$$\begin{array}{cccccccccc}
3 & 1 & 2 & \mathbf{4} & \quad & 2 & 1 & 3 & \mathbf{4} \\
3 & 1 & \mathbf{4} & 2 & & 2 & 1 & \mathbf{4} & 3 \\
3 & \mathbf{4} & 1 & 2 & & 2 & \mathbf{4} & 1 & 3 \\
\mathbf{4} & 3 & 1 & 2 & & \mathbf{4} & 2 & 1 & 3
\end{array}$$

We just obtained a complete list of all permutations of 4 elements.

7.3.2.1 Generalizing the example to an integer n

For a general n, if we have a complete list of permutations of $n - 1$ elements, we have to write each of them n times, then insert the number n at ever possible position (there are n such positions) in each block of n equal permutations of $n - 1$ elements. This exhausts all the possibilities.

7.3.5 Exercise
Reasoning by contradiction, prove the last statement formally.

7.3.2.2 The induction starts at 1

Can we list all permutations of one element? Sure, here it goes: (1). This will be the base case of our recursive method.

7.3.3 The algorithm

The recursive function `permutations`, given below, takes an integer n as input, and returns a set L of all permutations of n elements. It makes use of a temporary set L' containing all permutations of $n - 1$ elements, which is obtained by a recursive call to `permutations` with the argument set at $n - 1$. The permutation π is represented, as in Sect. 7.3.1, as a sequence of integers $(\pi_1, \ldots, \pi_{n-1})$.

```
function permutations($n$) {
if ($n = 1$) then
   $L = \{(1)\}$;
else
   $L' =$ permutations($n - 1$);
   $L = \varnothing$;
   for ($(\pi_1, \ldots, \pi_{n-1}) \in L'$) do
      for ($i \in \{1, \ldots, n\}$) do
         $L \leftarrow L \cup \{(\pi_1, \ldots, \pi_{i-1}, n, \pi_i, \ldots, \pi_{n-1})\}$;
      end for
   end for
end if
return $L$;
}
```

7.3.3.1 Data structures

Since the number n needs to be inserted at every possible position in the sequence $\pi = (\pi_1, \ldots, \pi_{n-1})$, we choose a list for storing π. As for the sets L, L', since we know their sizes *a priori* (they have $n!$ and, respectively, $(n-1)!$ elements, by Exercise 7.3.3), arrays will suffice. For π, on the other hand, since we need to add the element n at every possible position, we need linked lists.

7.3.4 Java implementation

Since permutations are in fact orders on sets with elements in $\{1, \ldots, n\}$, the Java implementation of the permutation listing algorithm is stored in the file `Orders.java`, containing only the class `Orders`, which also contains the `main` method.

The file starts as usual, with comments and imports.

```
/*
  Name: Orders.java
  Purpose: list all permutations of n symbols
  Author: Leo Liberti
  Source: Java
  History: 28/8/11  work started
*/

import java.io.*;
import java.util.*;
import java.lang.*;
```

7.3.4.1 Class structure

Here is the structure of the Java class: it consists of just three static methods. As mentioned above, we represent permutations as linked lists and sets of permutations as arrays.

```
class Orders  {
    // used to print a list of integers in permutation format
    public static void printList(LinkedList<Integer> tl);
    // find all permutations (or orders) on n symbols
    public static ArrayList<LinkedList<Integer> > orders(int n);

    // the main method
    public static void main(String[] args);
}
```

The `printList` method simply prints out a linked list of integers: this is the representation of a permutation, so this method actually prints a permutation to the screen.

The `orders` method implements the recursive algorithm, and returns a set L of all permutations of n elements. We need to spend a paragraph about the difficult-looking type

`ArrayList<LinkedList<Integer> >`.

The `ArrayList` type is a parametrizable class that implements an array whose elements contain whatever parameter type is given between angular brackets. In this case, the parameter type is `LinkedList<Integer>`, which is the same type taken by `printList` to print out permutations. So, all in all,

`ArrayList<LinkedList<Integer> >`

is the type of L and L' in the algorithm.

By the way, we are using `Integer`s rather than `int`s because the former are passed by reference whilst the latter are passed by value, and you should always use "reference classes" (rather than elementary data types) as parameter types to parametrizable classes.

7.3.4.2 The `main` method

The point of entry reads an integer n from the first argument of the command line, then calls the recursive function `orders(`n`)`, storing the resulting set in the `ArrayList L` of `LinkedList`s of `Integer`s, then declares an `Iterator` object to `L`.

`Iterator`s behave like pointers, insofar as their `next()` method will return the next object stored in the linear data structure. Their `hasNext()` method allows testing whether the current reference being held in the `Iterator` points to the dummy object signaling the end of data structure. If this is not the case, then the next reference is printed on the screen.

```
    public static void main(String[] args) {

        // read n from cmd line
        int n = Integer.parseInt(args[0]);

        // first call to recursive function
        ArrayList<LinkedList<Integer> > L = orders(n);

        // loop on permutation list and print them all out
        Iterator<LinkedList<Integer> > ait = L.iterator();
        while(ait.hasNext()) {
            printList(ait.next());
            System.out.println();
        }
    }
```

7.3.4.3 The `printList` method

The `printList` method also makes use of `Iterator`s to print on the screen the list representation of the permutations.

```
public static void printList(LinkedList<Integer> tl) {
    Iterator<Integer> lit = tl.iterator();
    System.out.print("( ");
    while(lit.hasNext()) {
        System.out.print(lit.next() + " ");
    }
    System.out.print(")");
}
```

7.3.4.4 The `orders` method

This is the implementation of the main recursive function. The comments are in the code.

```
public static ArrayList<LinkedList<Integer> > orders(int n) {

    // an array whose components are lists of integers
    // the i-th component of the array contains the i-th permutation
    ArrayList<LinkedList<Integer> > L =
        new ArrayList<LinkedList<Integer> >();

    if (n == 1) {
        // base case for recursion

        // when n=1, there is only one permutation: the identity (1)
        LinkedList<Integer> identity = new LinkedList<Integer>();
        identity.add(1);
        L.add(identity);

    } else {
        // recursive step

        // L' is the list of permutations on n-1 symbols
        ArrayList<LinkedList<Integer> > Lprime = orders(n-1);

        // loop on permutations of n-1 symbols
        Iterator<LinkedList<Integer> > ait = Lprime.iterator();
        while(ait.hasNext()) {

            // smallOrder is the next permutation in L'
            LinkedList<Integer> smallOrder = ait.next();

            // loop on permutation vector indices from last to first
            for(int i = n-1; i >= 0; i--) {

                // insert symbol n at the i-th position in the smaller perm
                smallOrder.add(i, (Integer) n);

                // produce a new permutation as a copy of the old one
```

```
                    LinkedList<Integer> newOrder =
                        new LinkedList<Integer>(smallOrder);

                    // add this new permutation to the list L
                    L.add(newOrder);

                    // remove symbol n from smaller permutation
                    smallOrder.remove(i);
                }
            }

        }
        return L;
    }
```

7.4 The Hanoi tower

This is a great classic in the didactics of recursion; you'll find hundreds, if not thousands, of different treatments either in print or online, so we'll skim over this quickly.

The Hanoi towers is a game which consists of three poles, one of which holds a stack of k concentric flat cylinders with increasing radii (disks at the bottom of the stack have higher radius). The (unique) player is challenged to move the stack, one cylinder at a time, from one pole to another without ever changing the radius order (larger cylinders below smaller ones) — see Fig. 7.3.

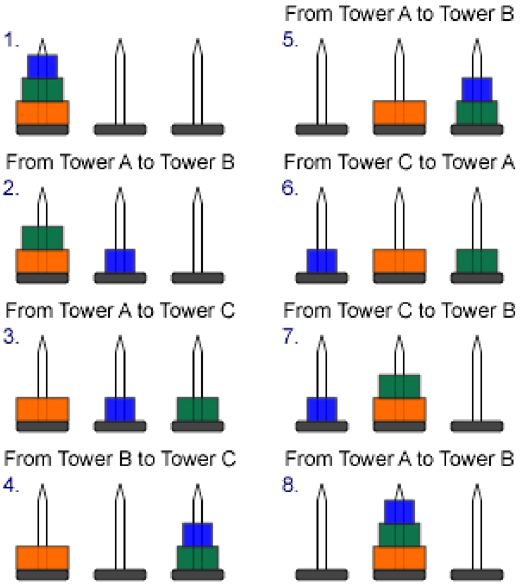

1.
From Tower A to Tower B
2.
From Tower A to Tower C
3.
From Tower B to Tower C
4.
From Tower A to Tower B
5.
From Tower C to Tower A
6.
From Tower C to Tower B
7.
From Tower A to Tower B
8.

Figure 7.3: The moves of the Hanoi towers game (picture taken online some time ago on Wikipedia — I think).

7.4.1 Inductive step

We shall number the poles from 1 to 3. In order to move k cylinders from pole 1 to pole 3, this is what we do:

1. move the topmost $k - 1$ cylinders from pole 1 to pole 2
2. move the largest (bottom) cylinder from pole 1 to pole 3
3. move the $k - 1$ discs from pole 2 to pole 3.

Notice that in order to move k cylinders, we move $k - 1$ cylinders twice, and move just one (the largest) cylinder from a pole to another (free) pole. This solution seems to be good, as long as we know how to move $k - 1$ cylinders.

We also remark that the steps involving the movement of $k - 1$ cylinders present no fundamental difference, aside from size, with moving k cylinders. Since we are leaving the largest cylinder fixed, it is never problematic: it is the largest, and it is at the bottom.

7.4.2 Base case

Our recursion base case is to deal with the case where $k = 1$, i.e. the stack only consists of one cylinder. This is easy: simply move it from pole 1 to pole 3, since there are no other complicating cylinders.

7.4.1 Exercise
Is the Hanoi tower game harder or easier if we add poles?

7.4.3 Java implementation

The Java implementation of the Hanoi tower game is really simple. We hold a single `Hanoi` class in the file `Hanoi.java`, which starts with the usual comments and imports. The `Hanoi` class only consists of two static methods: `move(int from, int to, int k)`, which moves k cylinders from pole `from` to pole `to`; and `main`.

```
// move "cylinders" cylinders from stack "from" to stack "to"
public static void move(int from, int to, int k) {
    if (k == 1) {
        // base case for recursion
       System.out.println("move upper cyl. on stack "+from+" to stack "+to);
    } else {
        // recursive step

        // computing the index of the other stack
        // (which is neither "from" nor "to")
        int other = 6 - (from + to);

        // first move k-1 cylinders from "from" to "other"
        move(from, other, k-1);

        // now move one cylinder from "from" to "to"
        move(from, to, 1);
```

```
            // finally move k-1 cylinders from "other" to "to"
            move(other, to, k-1);
        }
    }
```

7.4.2 Exercise
Prove that if $i \neq j \in \{1,2,3\}$*, then* $h = 6 - (i + j)$ *is such that* $\{i, j, h\} = \{1, 2, 3\}$.

The `main` method reads the number of cylinders in the initial stack from the command line.

```
    public static void main(String[] args) {
        // read number of cylinders to move from cmd line
        String theArg = args[0];
        int cylinders = Integer.parseInt(theArg);

        // move these cylinders from stack 1 to stack 3
        move(1,3,cylinders);
    }
```

7.5 Recursion in logic

Most of axiomatic set theory is built on recursive principles. One starts with very elementary notions, such as the empty set, and recursively applies simple operators and modifiers, to build ever more complex structures. Infinities of different ordinality and cardinality are treated using transfinite recursion [14].

In this section, I shall try to give you a very simplified, schematic and partial view of what is possibly the most famous theorem in mathematics: Gödel's incompleteness theorem.

7.5.1 Definitions

Axioms are given sentences (of a formal language) that define certain abstract entities (in our case, we require axioms to describe at least the nonnegative integers). Axioms are true by definition. The notation $\Phi \vdash \psi$ indicates that sentence ψ is a logical consequence of sentences in set Φ. The logical rules implicated in $\Phi \vdash \psi$ are required to be carried out by a computer. Let A be a set of axioms sufficient to define integer arithmetic (e.g. Peano's Axioms). A *theory* is a set $T \supseteq A$ of sentences such that $A \vdash \phi$ for each $\phi \in T$.

- A theory is *consistent* when it does not contain pairs of contradictory sentences $\phi, \neg\phi$.
- A theory is *complete* when every true statement expressible in the language is also in the theory.

Notice that there is a distinction between truth and logical consequence: a sentence might be true although we may not be able to prove it. This was the situation with Fermat's last theorem until A. Wiles found a proof. Completeness essentially asks a theory that it should prove every true statement.

7.5.2 Gödel's theorem

It was Hilbert's dream to prove that a set of axioms powerful enough to do integer arithmetic in would be both consistent and complete, i.e. all proofs could be derived computationally for all and only for true sentences. Gödel shattered Hilbert's dream, showing that no axiomatic system for integer arithmetic can ever be both consistent and complete.

7.5.3 The beautiful and easy part of the proof

If T is inconsistent, then every valid sentence of the language, be it true or false, can be proved (can you show this to be the case?). So we assume T is consistent and aim to show that there exists a true sentence not in T. Consider the sentence γ, defined recursively as $T\neg \vdash \gamma$. In natural language, γ states "I cannot be proved in T".

By the law of the excluded middle, exactly one sentence in the set $\{\gamma, \neg\gamma\}$ is true, and the other is false. We aim to prove that neither is in T: this way it does not matter which is true and which is false, at least one true sentence of the language will fail to be in T.

We ask the following questions.

- Is $\gamma \in T$? If so, then $T \vdash \gamma$, which means that $T \vdash (T \nvdash \gamma)$, i.e. $T \nvdash \gamma$, i.e. $\gamma \notin T$. This is a contradiction.
- Is $(\neg\gamma) \in T$? If so, then $T \vdash \neg\gamma$, i.e. $T \vdash \neg(T \nvdash \gamma)$, that is $T \vdash (T \vdash \gamma)$, thus $T \vdash \gamma$. In other words, assuming $T \vdash \neg\gamma$ leads to $T \vdash \gamma$, which implies that T is inconsistent. This is a contradiction, as we had assumed T to be consistent.

We can only conclude that T must be incomplete.

7.5.4 The other part of the proof

It is not immediately evident that the recursive definition $T\neg \vdash \gamma$ has a "base case". The most difficult part of Gödel's proof is to encode all the logic he needed for his argument within nonnegative integers. In particular, he was able to provide a "finiteness proof" for his recursive definition. This is very technical, and we shall cowardly eschew it here. However, if you feel up to a *really hard task*,

7.5.1 Exercise
Provide a formal proof of Gödel's theorem.

This has something to do with mapping all valid sentences to nonnegative integers bijectively, and showing that there exists an integer that maps back to γ. Of course you are free to read Gödel's original proof (bordering on the incomprehensible) as well as any of the dozens of books (technical and otherwise) published about this matter.

7.5.5 A natural language interpretation

Let us go back to γ, which was translated above to the natural language sentence "I cannot be proved in T". The easy part of Gödel's proof is as follows. Prove "I cannot be proved": if you can, what cannot be proved is proved, which is a contradiction. So prove "it is not true that I cannot be proved". If you can, then "I" can be replaced by what it stands for, i.e. "I cannot be proved": you then just proved that "it is not true that 'I cannot be proved' cannot be proved", which is the same as "it is true that 'I cannot be proved' can be proved" (we removed the double negation), i.e. 'I cannot be proved' can be proved, which is again a contradiction. This is why neither γ nor its converse can be proved in T, which is the reason why T must be incomplete.

Chapter 8

Graph searching and traversal

Abstract. How to efficiently visit all vertices in a graph: breadth-first search, depth-first search, and Prim's algorithm (with applications).

In this chapter we shall present and discuss some basic (and efficient) algorithms that explore, search and examine vertices and edges in graphs.

8.1 Graph scanning

A common task when solving problems on graphs is to visit all the vertices: this may be a precondition for verifying some claim over the graph vertices, or spawn actions on every vertex. The technical term is *scanning* the vertices of a graph (or the nodes of a digraph), starting from a given vertex (or node) s.

8.1.1 The Graph Scanning algorithm

Consider the following Graph Scanning algorithm. It scans vertices (then puts them in a set R), and examines vertices from a set Q.

```
Require: $G = (V, E)$, $s \in V$, $R = \{s\}$, $Q = \{s\}$
while $Q \neq \varnothing$ do
    choose $v \in Q$ // $v$ is scanned
    $Q \leftarrow Q \smallsetminus \{v\}$
    for $w \in N(v) \smallsetminus R$ do
        $R \leftarrow R \cup \{w\}$
        $Q \leftarrow Q \cup \{w\}$
    end for
end while
```

While there remain vertices in Q, we pick one, say v, and explore the vertices of its star $N(v)$ (or outgoing star $N^+(v)$ in case of a digraph) that are still unscanned. We mark these as "scanned" by putting them in R, and also put them in Q to be examined later. The algorithm ends when there are no more vertices in Q.

8.1.1.1 Correctness

The GRAPH SCANNING algorithm scans all vertices connected to s, by the following theorem.

8.1.1 Theorem
If there is a path P from s to $z \in V$, then GRAPH SCANNING *scans z.*

Proof. Suppose the claim is false, then there must exist an edge $\{x, y\}$ in P such that, after the termination of GRAPH SCANNING, $x \in R$ and $y \notin R$, since R is the set of scanned vertices. The previous statement holds by induction: it if were false, then by induction on the path length you would immediately conclude that $z \in R$, but we are supposing the claim false now. By Steps 5-6, x is added to Q at the same time it is added to R. Also, the algorithm will not stop until Q is empty, so at a certain point x is chosen from Q at Step 2. Immediately afterwards, all vertices in $N(v) \smallsetminus R$ are examined and scanned — in particular, since $\{x, y\}$ is in P and P is a path in G, then $y \in N(v)$ and $y \notin R$ means that y is added to R, which is a contradiction — hence the claim must be true. □

8.1.2 Exercise
Adapt Thm. 8.1.1 to digraphs.

8.1.1.2 Complexity

The GRAPH SCANNING algorithm takes time $O(n + m)$, where $n = |V|$ and $m = |E|$.

8.1.3 Theorem
If the graph is encoded as adjacency lists, GRAPH SCANNING *takes $O(n + m)$ in the worst case.*

Proof. By Thm. 8.1.1, each vertex enters R at least once. By Step. 4, it enters R only once. Moreover, no vertex enters Q without entering R (Steps 5-6), so each vertex enters Q at most once. This accounts for the $O(n)$ part. Now we claim that each edge $\{x, y\}$ is only considered twice. When $x = v$ in Step 2, then $y \in N(x)$, so either $y = w$ in Step 5-6 or $y \in R$ in the test at Step 4. In both cases, the pair $\{x, y\}$ is considered once. It is considered a second time when $y = v$, since $x \in N(y)$. This accounts for the $O(m)$ part. The choice of v at Step 2, the removal at Step 3, and the insertion of w at Step 6 can be done in constant time if Q is a linked list. The verification that $w \notin R$ at Step 4 and the insertion at Step 5 can be done in constant time if R is a binary array of n bits, with the v-th bit set to 1 if $v \in R$ and to 0 otherwise. Thus the total running time in the worst case is $O(n + m)$. □

8.1.4 Exercise
Adapt Thm. 8.1.3 to digraphs.

8.1.5 Exercise
Implement the GRAPH SCANNING *algorithm in Java.*

8.1.1.3 Connected components

Notice that GRAPH SCANNING identifies a *connected component* of the graph G containing the source vertex s. This follows by Thm. 8.1.1: if there is a path from s to z, then the algorithm scans z.

8.1.6 Exercise
Prove that if there is no path from s to z, then GRAPH SCANNING *starting from s does not scan z.*

This can be used to identify all connected components of a graph: let a be an integer array indexed by V, which associates to $v \in V$ the index of the connected component it belongs to. Initially, $a(v) = 0$ for all $v \in V$. Now let $i = 1$, pick any $v \in V$ and run Graph Scanning, setting $a(w) = i$ whenever w is scanned. When the algorithm terminates, increase i, pick any other $v \in V$ with $a(v) = 0$, and repeat, until no more v have $a(v) = 0$.

8.1.7 Exercise
Implement a Java algorithm for identifying all connected components in a graph.

8.1.1.4 The exploration tree

Consider the following modification of Graph Scanning algorithm: we initialize an edge set $F = \varnothing$ at the beginning, and then insert the instruction $F \leftarrow F \cup \{v, w\}$ between Step 4 and 5.

8.1.8 Theorem
At the end of the Graph Scanning *algorithm, the graph* $T = (R, F)$ *is a tree.*

Proof. That T is connected follows from Thm. 8.1.1. Suppose now, to arrive at a contradiction, that T has a cycle C. Let x, y be two vertices in C. Since C is a cycle, there are two paths P_1, P_2 with $P_1 \cup P_2 = C$, both from x to y, that have no vertex in common aside from x and y (see Fig. 8.1). Assume, without loss of generality, that x is scanned before y. Since we assumed C to be a subset of T, two edges, $\{v_2, y\}$ in P_1 and $\{v_4, y\}$ in P_2 will be added to F. Assume that $\{v_2, y\}$ is added first; then y becomes scanned and enters R. But then when v_4 is chosen from Q at a later iteration, because $y \in R$, $\{v_4, y\}$ cannot be added to F, hence nor to P_2 and nor to C, which provides us with a contradiction. Hence T cannot contain cycles, which concludes the argument. □

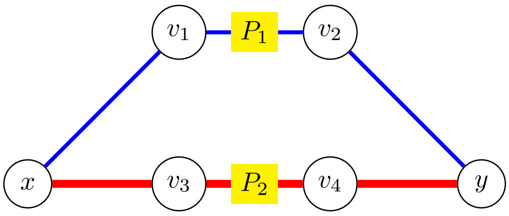


Figure 8.1: A cycle C can be partitioned in two paths P_1, P_2 from x to y.

Thus, Graph Scanning identifies a tree T in G.

8.1.9 Exercise
Prove that, if G is connected, then the tree T is spanning.

8.1.1.5 Choosing $v \in Q$

The choice of $v \in Q$ at Step 2 determines the order in which the nodes are scanned. This order can be changed using different data structures to implement the set Q. The two most commonly used ones are stacks and queues.

8.2 Breadth-first search

Let Q be a queue in the GRAPH SCANNING algorithm. We can only remove elements from one end using the `popFront()` method, which removes the first element of the queue and returns. And we can only insert elements at the other end using the `pushBack()` method, which simply inserts a new element as the last in the queue.

The corresponding implementation of the GRAPH SCANNING algorithm has a remarkable property: it ranks vertices according to how far they are from the given source vertex s (the distance of v from s being the number of edges on the shortest path from s to v). We enrich the GRAPH SCANNING algorithm with a vertex ranking function α such that $\alpha(s) = 0$ at the outset, and $\alpha(w) = \alpha(v) + 1$ if $w \in N(v)$ at Steps 5-6.

Here is the BREADTH-FIRST SEARCH (BFS) algorithm.

```
Require: $G = (V, E)$, $s \in V$, $R = \{s\}$, $Q = \{s\}$ is a queue
1: $\alpha(s) = 0$
2: while $Q \neq \varnothing$ do
3:   pop $v$ from the front of the queue $Q$
4:   for $w \in N(v) \smallsetminus R$ do
5:     $\alpha(w) = \alpha(v) + 1$
6:     $R \leftarrow R \cup \{w\}$
7:     push $w$ on the back of the queue $Q$
8:   end for
9: end while
```

8.2.1 Paths with fewest edges

What kind of paths does BFS determine? Recall by Sect. 8.1.1.4 that GRAPH SCANNING (and hence BFS) identifies a tree T within the given graph $G = (V, E)$; if the graph is connected, the tree is spanning.

8.2.1 Exercise
Prove that, given any graph G, any spanning tree T of G, and any pair of distinct vertices x, y of G, there is a unique path from x to y using edges of T.

The spanning tree identified by the BFS is also called the *BFS tree*. We are now going to show that the BFS tree is also a shortest path tree for G, as long as the length of a path is counted as the number of its edges.

8.2.2 Lemma
Let $(s, v_1, \ldots, v_k)$ be any path in the BFS tree. Then $\alpha(v_k) = k$.

Proof. By induction on k. When $k = 1$ this holds because at Step 5 α is set to 1 for all vertices in $N(s) \smallsetminus R$, which includes v_1. Assume the result holds for $k - 1$. Consider the iteration of the BFS when v_{k-1} is extracted from Q at Step 3: by the induction hypothesis, $\alpha(v_{k-1}) = k - 1$. Since $\{v_{k-1}, v_k\}$ is in the path, which is itself in the BFS tree, v_k is not yet in R when v_{k-1} is extracted from Q: so BFS performs step 5 with $w = v_k$, which implies that $\alpha(v_k) = k$, as claimed. □

The *BFS rank* of a vertex $v \in V$ is the number of edges in the unique path from s to v in the BFS tree.

8.2.3 Lemma
For any k, all vertices of BFS rank k are adjacent in Q.

Proof. By induction on k. When $k = 0$, this is obvious as there is only one vertex with this BFS rank, namely s. Assume the property holds for $k-1$, then all the vertices u with $\alpha(u) = k-1$ enter the queue Q one after the other. Now because of Step 5, all vertices v with $\alpha(v) = k$ enter the queue because they are adjacent to a vertex u with $\alpha(u) = k-1$. Since all such vertices are adjacent in Q, and vertices are extracted from Q consecutively, result follows. □

8.2.4 Corollary
If $\alpha(u) < \alpha(v)$, u enters Q before v does.

8.2.5 Exercise
Prove Cor. 8.2.4.

8.2.6 Exercise
Prove that α is a well-defined function, i.e. no vertex v is assigned two different values of α.

8.2.7 Theorem
Given a graph $G = (V, E)$ with unit edge costs, a BFS on G from a vertex $s \in V$ determines a shortest path tree rooted at s.

Proof. Let $t \neq s$ be a vertex in V, and consider a path $R = (s, v_1, \ldots, v_k = t)$ on the BFS tree T. Suppose, to get a contradiction, that the path R is not shortest from s to t. Since subpaths of a shortest path are also shortest, there must there must exist a sub-path of $R' = (s, v_1, \ldots, v_h)$ of R, with $h < k$, such that R' is not shortest from s to v_h. By Lem. 8.2.2, $\alpha(v_h) = h$. Since R' is not shortest, there must be a different path $P = (s, u_1, \ldots, u_\ell, v_h)$ which is shortest from s to v_h: necessarily we have $\ell + 1 < h$. Again by Lem. 8.2.2 we have $\alpha(u_\ell) = \ell$. Moreover, by Cor. 8.2.4, u_ℓ enters Q before v_h, so the BFS finds P before R'. Because u_ℓ, v_h are consecutive vertices on the path P, there must be an edge $\{u_\ell, v_h\} \in E$, and since P is found before R', we have $\alpha(v_h) = \ell + 1$ (by Step 5); and since $l + 1 < h = \alpha(v_h)$, we obtain $\alpha(v_h) < \alpha(v_h)$, a contradiction. □

Algorithmically, we can get rid of R and only use the storage for α, as follows. We allocate an integer array α indexed on V, and initialize it at $\alpha(v) = |V| + 1$ for all $v \in V$. We replace the test $w \in N(v) \smallsetminus R$ at Step 4 with

$$w \in \{N(v) \mid \alpha(v) = |V| + 1\},$$

and eliminate Step 6 and all other references to R. This works because $\alpha(v) = |V| + 1$ if and only if v is unscanned: whenever a vertex is scanned, its α value is assigned by Step 5 to a value $< |V|$.

8.2.2 History of the BFS

BFS was independently discovered by several people, and no-one quite knows who was first. The first publication I could find is Claude Berge's book [2], published in 1958. The description is shown in Fig. 8.2.

8.2.3 Looking for a good route in public transportation

Consider a bus network with the following timetables:

A	
1	h:00
2	h:10
3	h:30

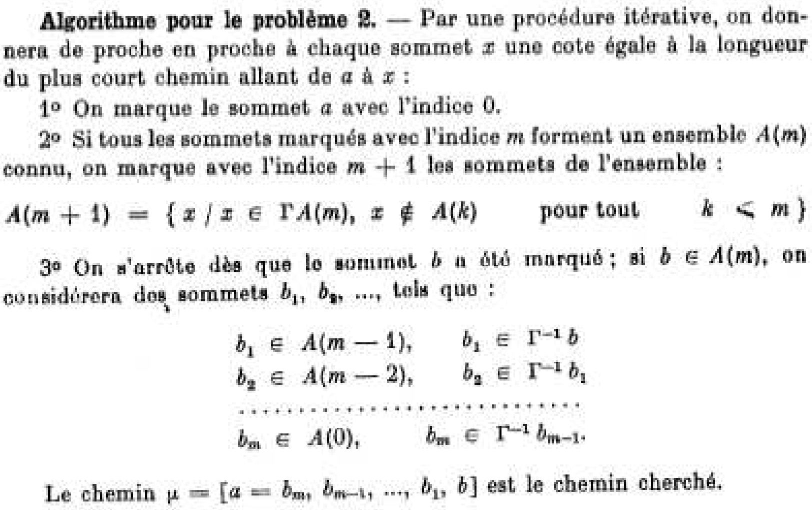
B	
1	h:00
4	h:20
5	h:40

C	
2	h:10
3	h:20
5	h:30

D	
4	h:20
5	h:40
6	h:50

E	
2	h:05
5	h:10
6	h:30

F	
3	h:25
4	h:30
6	h:40



Algorithme pour le problème 2. — Par une procédure itérative, on donnera de proche en proche à chaque sommet x une cote égale à la longueur du plus court chemin allant de a à x :

1° On marque le sommet a avec l'indice 0.

2° Si tous les sommets marqués avec l'indice m forment un ensemble $A(m)$ connu, on marque avec l'indice $m+1$ les sommets de l'ensemble :

$$A(m+1) = \{ x \,/\, x \in \Gamma A(m),\ x \notin A(k) \quad \text{pour tout} \quad k \leqslant m \}$$

3° On s'arrête dès que le sommet b a été marqué ; si $b \in A(m)$, on considérera des sommets $b_1, b_2, \ldots$, tels que :

$$\begin{array}{ll} b_1 \in A(m-1), & b_1 \in \Gamma^{-1} b \\ b_2 \in A(m-2), & b_2 \in \Gamma^{-1} b_1 \\ \ldots\ldots\ldots\ldots\ldots & \\ b_m \in A(0), & b_m \in \Gamma^{-1} b_{m-1}. \end{array}$$

Le chemin $\mu = [a = b_m, b_{m-1}, \ldots, b_1, b]$ est le chemin cherché.

Figure 8.2: The original description of the BFS algorithm in [2].

How do we find a convenient itinerary from bus stop 1 to bus stop 6, leaving at h:00?

We model the problem by means of an *event network*: the nodes are labelled by pairs (bus stop, minutes after the hour), and an arc $((b_1, m_1), (b_2, m_2))$ denotes: (i) a bus going from stop b_1 at h:m_1 to stop b_2 at h:m_2, if $b_1 \neq b_2$; (ii) waiting at stop b_1 from h:m_1 to h:m_2 if $b_1 = b_2$. Each arc $((b_1, m_1), (b_2, m_2))$ is labelled by the length of time that separates the events (b_1, m_1) and (b_2, m_2). For the timetables above, the event network is given in Fig. 8.3.

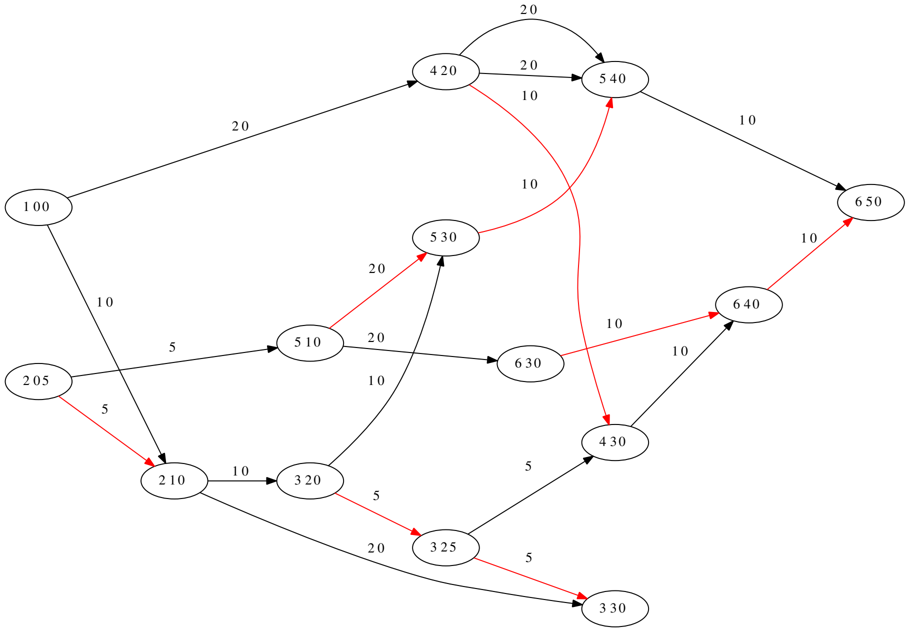


Figure 8.3: An event network.

We apply BFS to the event network of Fig. 8.3, with $s = (1, 00)$. Here is the evolution of the queue Q:

1. $Q = \{(1, 00)\}$ (initialization)
2. $Q = \varnothing$ (Step 3)
3. $Q = \{(2, 10), (4, 20)\}$ (Step 7)
4. $Q = \{(4, 20)\}$ (Step 3)
5. $Q = \{(4, 20), (3, 20), (3, 30)\}$ (Step 7)
6. $Q = \{(3, 20), (3, 30)\}$ (Step 3)
7. $Q = \{(3, 20), (3, 30), (4, 30), (5, 40)\}$ (Step 7)
8. $Q = \{(3, 30), (4, 30), (5, 40)\}$ (Step 3)
9. $Q = \{(3, 30), (4, 30), (5, 40), (3, 25), (5, 30)\}$ (Step 7)
10. $Q = \{(4, 30), (5, 40), (3, 25), (5, 30)\}$ (Step 3)
11. $Q = \{(5, 40), (3, 25), (5, 30)\}$ (Step 3)
12. $Q = \{(5, 40), (3, 25), (5, 30), (6, 40)\}$ (Step 7)
    At this point, we know we can get to bus stop 6 at h:40, but since there are other nodes labelled with bus stop 6 (one of which has a time label h:30), we have to continue and see if there are some other paths leading to a better time.
13. $Q = \{(3, 25), (5, 30), (6, 40)\}$ (Step 3)
14. $Q = \{(3, 25), (5, 30), (6, 40), (6, 50)\}$ (Step 7)
    So we can also get to bus stop 6 at h:50.
15. $Q = \{(5, 30), (6, 40), (6, 50)\}$ (Step 3)
16. $Q = \{(6, 40), (6, 50)\}$ (Step 3)
17. $Q = \{(6, 50)\}$ (Step 3)
18. $Q = \varnothing$ (Step 3).

Thus we conclude that the most convenient itinerary from stop 1 to stop 6 arrives at 6 at h:40. On the other hand, we never specified what "convenient" really meant. We automatically assumed it meant "fast", but, as we said before, the BFS finds shortest paths to all nodes only in terms of number of edges. Since each edge in the event network represents a change of mode of transportation (there are seven such modes: waiting, bus A, . . . , bus F), the BFS identifies paths that minimize the number of changes. Since $(6, 40)$ is the first node for bus stop 6 that entered the queue, in view of Cor. 8.2.4 the (directed) path $((1, 00), (4, 20), (4, 30), (6, 40))$ is the shortest in terms of number of changes. It suffices to change three times: bus B from stop 1 to stop 4, then wait for 10 minutes at stop 4, then bus F from stop 4 to stop 6.

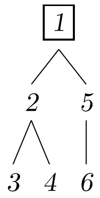
8.3 Depth-first search

Let Q be a stack in the Graph Scanning algorithm. We insert and remove elements from one end of the linear data structure only, using the methods `push()` and `pop()`. The resulting algorithm is called Depth-First Search (DFS).

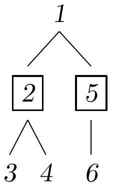
```
Require: $G = (V, E)$, $s \in V$, $R = \{s\}$, $Q = \{s\}$ is a stack
while $Q \neq \varnothing$ do
    pop $v$ from the stack $Q$
    for $w \in N(v) \smallsetminus R$ do
        $R \leftarrow R \cup \{w\}$
        push $w$ on the stack $Q$
    end for
end while
```

Understanding DFS may be hard at first. Let us first see the effect on DFS on a tree.

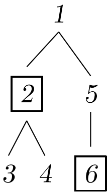
8.3.1 Example
Consider the following tree, setting $s = 1$. Boxed elements are on the stack.



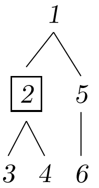
1 *is pushed*



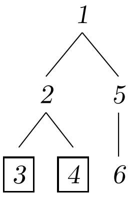
1 *is popped,* $2, 5$ *are pushed (in this order)*



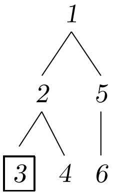
5 *is popped,* 6 *is pushed*



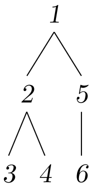
6 *is popped*



2 *is popped,* $3, 4$ *are pushed (in this order)*



4 *is popped*



3 *is popped*

So the order of the visit is $1, 5, 6, 2, 4, 3$.

In other words, every path from the source to a leaf is explored by depth first, backtracking from every reached leaf to the closest ancestor (i.e. previously visited vertex on the same path) with at least three neighbours, or the source, and continuing until each edge has been traversed exactly twice (once in each direction). On general graphs, just imagine the effect of the test $v \notin R$ at Step 3: if v was just popped off the stack a node has already been scanned previously, the edge is ignored.

DFS has been used for several different tasks in graph theory, see e.g. the WikiPedia entry `http://en.wikipedia.org/wiki/Depth-first_search`.

8.3.1 A recursive version of DFS

DFS can also be written as a recursive algorithm. This should not be so surprising, in view of the relationship between stacks and recursion discussed in Chapter 7. Given a connected graph $G = (V, E)$ and a source vertex $s \in V$, consider the following recursive function `dfs(`G, s, R, v`)`:

```
for $w \in N(v) \smallsetminus R$ do
    $R \leftarrow R \cup \{w\}$
    dfs($G, s, R, w$)
end for
```

The DFS search of G from s is then simply `dfs(`$G, s, \{s\}, s$`)`.

8.3.2 Exercise
Rewrite the recursive DFS algorithm iteratively by means of a stack, and verify that it is the same as the DFS algorithm given at the beginning of Sect. 8.3.

8.3.2 History of the DFS

The DFS is one of the oldest methods in graph theory; its first application was finding the way out of mazes. Recursion is easily explained in this setting: a person *inside* a maze can barely be tele-transported to a different location without actually travelling through the maze; so that from any visited node, one can only go on visiting the adjacent nodes, and so on recursively.

U. Eco, in *The Name of the Rose*, has William of Baskerville read the following text from an ancient manuscript, which must have dated before 1300:

> *To find the way out of a labyrinth, there is only one means. At every new junction, never seen before, the path we have taken will be marked with three signs. If, because of previous signs on some of the paths of the junction, you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps. But if one or two apertures of the junction are still without signs, you will choose any one, making two signs on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three. All the parts of the labyrinth must have been visited if, arriving at a junction, you never take a passage with three signs, unless none of the other passages is now without signs.*

Of course, Eco's book is a novel, not history. Furthermore, the algorithm looks much more complicated than DFS. Yet there is both backtracking ("retrace your steps") and the statement of a theorem about termination. It is also clear that this algorithm, unlike DFS, allows a searching agent to re-visit some junctions.

König, in the chapter of his book [13] dedicated to the labyrinth problem, reports the algorithms of Wiener's, published in 1873, and Trémaux, published in 1882. The latter is now established [4] with being the official creator of the DFS. WikiPedia reports that Trémaux was enrolled at Ecole Polytechnique (X1876).

R. Tarjan published a series of works that established DFS as a very versatile algorithm, which can be used for many important fundamental tasks in graph theory. For example, DFS can be used to identify the blocks of a graph and the cut vertices in linear time. A *cut vertex* is a vertex which, if removed, disconnects the graph. A *block* is a subgraph connected to the rest of the graph by means of a single cut vertex.

DFS can also be used to determine whether a graph is planar (i.e. whether it can be drawn in the plane without edge intersections). On digraphs, DFS can be used to determine a *topological order* on the vertices of a Directed Acyclic Graph (DAG). The vertices of a DAG are in topological order if they are labelled so that, for any arc (u, v), we have $u < v$.

8.3.3 Easy and difficult natural languages

In Sect. 6.4.3.4, we explained that understanding the grammatical structure of sentences in natural languages requires them to be parsed into a derivation tree. When you hear a sentence, however, you simply hear a sequence of words. How is a sequence (a linear structure) turned into a tree? We may assume that our brain is doing the parsing, and that the leaves of the tree are some fundamental syntactical units to which a basic meaning is attached. The complex meaning of the whole sentence is then put together by the brain by combining the basic meanings of the leaves, retracing each path from the leaves to the root.

Under this hypothesis, the brain performs a DFS on the derivation tree. The memory effort made by the brain is proportional to the tree depth: one needs to remember all the intermediate vertices along a path from a leaf to the root in order to build the meaning relative to that path. The derivation tree of the sentence *the soft furry cat purrs* (see Fig. 8.4) has depth 6; the brain must remember at most a sequence of 6 vertices to retrace to the root vertex. Miller, in 1956, proposed that on average, the human memory can recall seven random words without efforts. In our setting, we can take this to mean that the maximum "stack size" for the brain is 7. Sentenes are "simple" if the depth is 7 or less, and "complicated" if more.

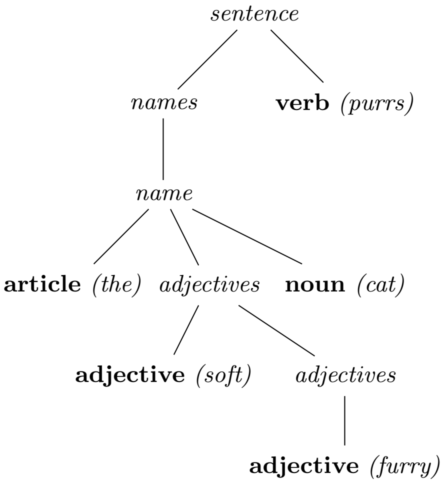


Figure 8.4: The derivation tree of *the soft furry cat purrs.*

The brain, however, also follows a precise order when processing vertices at Step 3 of the (iterative) DFS algorithm. This order is that of the temporal arrival of the words to the ear. What is said before is processed before what is said later. We identify two "tree shapes" that are at the extrema of all possible derivation trees: the *regressive trees*, slanted towards the left, and the *progressive trees*, slanted towards the right (see Fig. 8.5). A regressive derivation tree corresponds to a sentence whose basic meanings comes later in time: in "the soft, furry cat purrs", for example, the most basic meanings are assigned to the leaves *cat* and *purrs*, the last words to be heard by the ear. A progressive derivation tree corresponds to a sentence whose basic meanings come early in time: in "l'éleve retardataire n'apprend que la moitié des choses qu'on lui enseigne", the most basic meanings are assigned to the leaves *éleve*, *retardataire* and *apprend*, the first words to be heard.

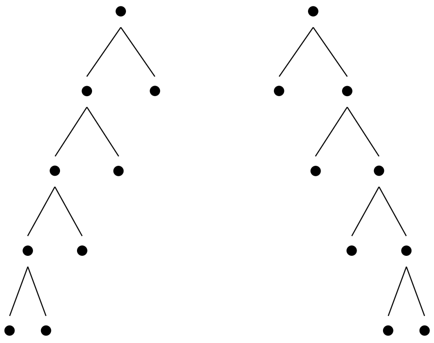


Figure 8.5: Regressive (left) and progressive (right) tree shapes.

When DFS is applied to a regressive tree, it needs as much stack size as the tree depth. However, when DFS is applied to a progressive tree, something different happens: once the DFS scans the deepest (and rightmost) leaf, it no longer needs to backtrack to the root vertex, because the sentence is finished — there are no more words being heard! In other words, your brain makes much less effort with progressive than with regressive trees. In algorithmic terms, this means that you no longer need to push vertices on the stack at Step 5 if you process vertices on a left-to-right order, and you are exploring the rightmost path [24].

We can draw some linguistic conclusions from all this. Anglosaxon languages, for example, prefix adjectives to nouns, and are therefore more regressive: when saying "soft, furry cat", your ear hears *soft* and *furry* before *cat*, which has the basic meaning: therefore, the adjectives must be remembered (pushed on the stack) until the noun occurs. Latin languages decrease this tendency, and are more progressive: in "le lion imposant et sauvage", you do not need any stack: the basic meaning *lion* comes before *imposant* and *sauvage*. Somehow, it seems that latin languages are easier from the brain to handle, as far as the memory effort is concerned. Classical Latin, on the other hand, is different. Consider the famous *incipit* of the Aeneid's second canto, "Inde toro pater Æneas sic orsus ab alto". Notice how the order looks random: a literal English translation is "Thereafter seat father Eneas thus standing from a high" (a more successful translation would be "Thereafter father Eneas, thus standing from a hight seat"). By messing up the order of the words in the sentence, it produced derivation trees requiring a non-temporal order on the $N(v)$ loop in the DFS Step 3 in order to reconstitute the sentence's meaning. Perhaps this is why Latin is a dead language.

8.4 Finding a spanning tree of minimum cost

A tree $T = (U, F)$ in a graph $G = (V, E)$ is spanning if $U = V$, as explained in Sect. 6.1.3. Two applications of minimum cost spanning trees to networks, whose edges are assigned costs, were discussed in Sect. 6.4.4.

8.4.1 Theorem
Let $G = (V, E)$ be a graph and $c : E \to \mathbb{R}$ be an edge cost function. Let $T = (R, F)$ be a spanning tree of G. T is of minimum cost if and only if for each proper subset U of V, the unique edge of the cutset $\delta(U)$ in T has minimum cost in $\delta(U)$.

Proof. ($\Rightarrow$) Proceed by contradiction, and assume T is of minimum cost, but there is a proper subset $U \subsetneq V$ such that the unique edge e in $F \cap \delta(U)$ is not of minimum cost. Because of the minimal connectedness of T, removing any edge disconnects T into two trees T_1 and T_2, with T_1 spanning U, and T_2 spanning $V \smallsetminus U$. By definition of a cutset, every edge in $\delta(U)$ has one incident vertex in U and the other in $V \smallsetminus U$, so every such edge can be used to re-connect T_1 and T_2 without creating any cycles. Let f be the minimum cost edge in $\delta(U)$: then $T' = T_1 \cup \{f\} \cup T_2$ is connected, spans all V, and has no cycles: it is therefore a spanning tree of G. Moreover, the cost of T' is $c(T') = c(T_1) + c(f) + c(T_2) < c(T_1) + c(e) + c(T_2) = c(T)$, so T could not be a minimum cost spanning tree, which goes against our assumption.

($\Leftarrow$) Let $T = (R, F)$ be a spanning tree for G such that $T \cap \delta(U) = \{e\}$ be an edge of minimum cost in $\delta(U)$ for each proper subset $U \subsetneq V$. Let $T' = (U, F')$ be a minimum cost spanning tree with $E \cap E'$ as large as possible. Let $f \in E \smallsetminus E'$. Because of the minimal connectedness of T, the removal of f from T disconnects T into two disconnected trees T_1, T_2 spanning respectively a proper subset $U \subsetneq V$ and $V \smallsetminus U$. Consider the unique edge g in $\delta(U) \cap T'$: if $c(g) = c(f)$ then, since $E \cap E'$ is as large as possible, $g = f$; but this is impossible since f was chosen to be outside E'. Also, by the hypothesis on T, f must have minimum cost within $\delta(U)$. But then $c(g) > c(f)$. So the tree $T'[U] \cup \{f\} \cup T'[V \smallsetminus U]$ is different from T' and has lower cost than T', which is impossible as T' was assumed to have minimum cost. Thus $E \smallsetminus E'$ must be empty, which implies $E = E'$, which means that T has minimum cost. □

8.4.2 Exercise
Prove that for any proper subset $U \subsetneq V$, $|F \cap \delta(U)| = 1$, and that for any tree T and edge $e \in \delta(U)$, $T[U] \cup \{e\} \cup T[V \smallsetminus U]$ is a tree.

Here we give an algorithm to determine a Minimum-cost Spanning Tree (MST) based on the ($\Leftarrow$) direction of Thm. 8.4.1. This algorithm, due to R. Prim and published in 1957, “grows” a spanning tree $T = (R, F)$, initially set to $(\{s\}, \varnothing)$, starting from a source vertex $s \in V$. At the outset, it selects from the cutset $\delta(s)$ the cheapest edge $\{s, w\}$ and adds it to F, also adding w to R. The general iteration is as follows. The cheapest edge $\{v, w\}$ is selected from the cutset $\delta(R)$; by definition, exactly one between v and w is in R, assume this to be v without loss of generality. Then $\{v, w\}$ is added to F and w to R.

8.4.1 Prim’s algorithm: pseudocode

Here follows a more detailed pseudocode for Prim’s Algorithm. Given a weighted graph $G = (V, E, c)$ where $c : E \to \mathbb{R}$ is an edge cost function, and a source vertex $s \in V$, it outputs a spanning tree $T = (R, F)$ of G of minimum cost. We denote the edge weight $c(\{u, v\})$ by c_{uv} for any edge $\{u, v\} \in E$. Prim’s algorithm employs the following data structures:

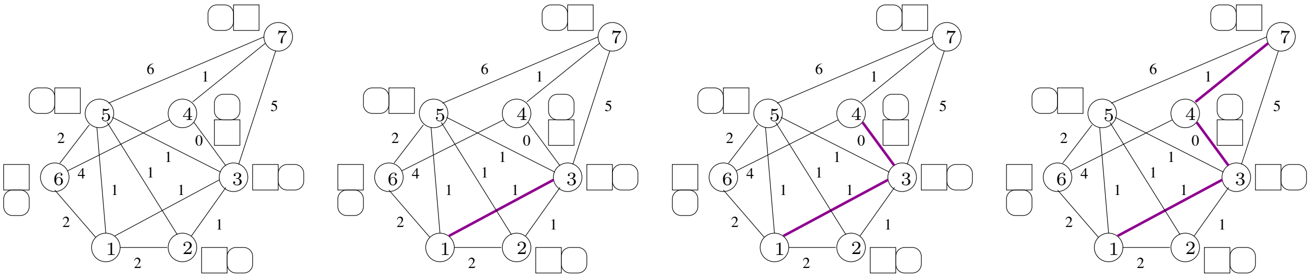
- R: set of reached vertices (vertices in the tree)
- F: set of edges in the tree
- u: best next vertex
- $\zeta : V \smallsetminus R \to \mathbb{R}$: cost of reaching from R a vertex outside R
- $\pi : V \smallsetminus R \to R$: immediate predecessor in T to a vertex outside R.

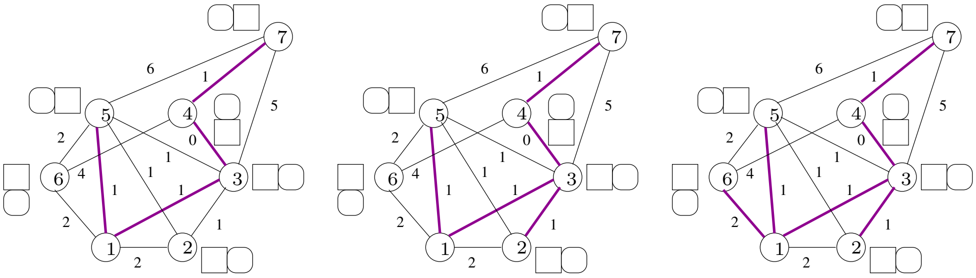
It works by iteratively choosing the best edge $\{u, v\}$ in the current cutset (containing u but not v) and adding v to the cutset. It terminates when the cutset contains every vertex.

```
$R = \{s\}$, $F = \varnothing$, $\forall v \in V$ set $\zeta(v) = \infty$, $pi(v) = s$
for $w \in N(s)$ do
    $\zeta(w) = c_{sw}$
end for
while $R \neq V$ do
    let $u \in V \smallsetminus R$ such that $\zeta(u)$ is minimum
    mark $u$ as reached by adding it to $R$
    add the edge $\{\pi(u), u\}$ to $T$
    update $\zeta, \pi$: $\forall v \in N(u)$ s.t. $\zeta(v) > c_{uv}$, let $\zeta(v) = c_{uv}$ and $\pi(v) = u$.
end while
```

8.4.3 Example

Here is the effect of Prim’s algorithm on the following weighted graph.





8.4.4 Exercise
Number the vertices in the graph of Example 8.4.3, then fill the empty boxes with the values for ζ, π at every vertex.

8.4.2 Complexity of Prim's algorithm

The initialization takes $O(n)$ (Step 2). The main loop (Step 5) also takes $O(n)$. The choice of u (Step 6) takes $O(n)$, the updates of R, T (Steps 7-8) take $O(1)$, and the updates of ζ, π (Step 9) takes $O(n)$. Altogether, this is:

$$\begin{aligned} O(n + n(n+1+n)) &= O(n + n^2 + n + n^2) \\ &= O(2(n + n^2)) \\ &= O(n + n^2) = O(n^2). \end{aligned}$$

Chapter 9

Problems and complexity

ABSTRACT. A theoretical excursion in the theory of complexity. Problems and complexity classes: **P** and **NP**. **NP**-hard and **NP**-complete problems. Exact and heuristic algorithms.

9.1 Decision problems

In Sect. 1.4.2 and 7.1.2.3, we defined a problem to be a set of pairs (input,output). In fact, we understand a *decision problem* to be an infinite set of *instances*, which are data objects of the same type, together with a formal question that can only be answered by YES or NO. For example, the CONNECTED GRAPH PROBLEM (CGP) takes as input an undirected graph $G = (V, E)$ and asks to determine whether the graph is connected or not. We remark that in order to qualify as a formal decision problem, the instance set *must* be infinite.

In a decision problem, it is not sufficient to guess the answer: we require a proof. Accordingly, the answer to a given problem instance must be YES or NO with a *certificate* that anyone can check computationally to establish the truth of the answer. A certificate may be a combinatorial structure which, by its very presence, testifies as to the truth of the answer; or a theorem with its formal proof; or even the printout of all the computation carried out to obtain the YES or the NO. In the case of the CGP, for example, we might exhibit a spanning tree for a YES, or an empty nontrivial cutset for a NO: a graph has a spanning tree if and only if it is connected, and has a nonempty nontrivial cutset if and only if it is not. Notice that an instance might have more than one certificate: for the CGP example, any spanning tree is an acceptable YES certificate, and any nonempty nontrivial cutset is an acceptable NO certificate.

An infinite subset of instances out of a given problem is sometimes called a *problem case* or *subproblem*. For example, the CGP restricted to cliques is an infinite class of instances (there is a clique for every integer n). Because it is easy to show that every clique is connected, the answer to each instance of this CGP subproblem is always YES, and the proof, valid for every instance, provides an instance-independent certificate.

9.2 Optimization problems

In an *optimization problem* we assign scalar values to all possible YES certificates, and look for certificates having smallest (*minimization*) or greatest (*maximization*) value. If an instance of an optimization problem is a NO instance, we simply require the proof for the NO (no scalar values are assigned to NO instances).

The MINIMUM SPANNING TREE (MST) problem is an example of an optimization problem. Disconnected graph instances are NO instances, whilst all connected weighted graphs provide YES instances, and among the set of all spanning trees we require one with minimum cost (more than one spanning tree might have minimum cost: consider the case where all edges have unit weight).

9.2.1 Relationship between decision and optimization

The abstract decision problem is as follows. Given a set U and a subset V of U, determine whether V is empty or not. The abstract optimization problem would add a scalar-valued function $\mu : U \to \mathbb{R}$ and ask to determine whether V is empty, and, if not, find $v \in V$ with maximum or minimum μ value.

Let $M = \{\mu(u) \mid u \in U\}$. A minimization problem (U, V, μ) on a finite set U can be solved by solving $O(\log |M|)$ decision problems (U, V_α) defined by $V_\alpha = \{v \in V \mid \mu(v) \leq \alpha\}$. We proceed by bisection (see Sect. 10.2) on the scalar set M. We start with α set to the median of M, and solve (U, V_α). If $V_\alpha = \varnothing$ we replace M by $\{\beta \in M \mid \beta > \alpha\}$, otherwise by $\{\beta \in M \mid \beta \leq \alpha\}$, and we repeat. By the familiar bisection argument,[1] we need at most $O(\log |M|)$ iterations before a $v \in V$ minimizing μ is found.

9.2.1 Exercise
Adapt the above algorithm to the maximization case.

9.2.2 Exercise
Why does U need to be finite, in the above algorithm?

For example, the MST problem is equivalent to solving a set of decision problems of the form, "given a weighted graph, and a scalar k, does it have a spanning tree with cost less than k?".

9.3 Algorithms

In this context, we require an algorithm to solve a problem, rather than a finite set of instances. Procedures that only work with a certain graph, or a finite set of graphs are not considered algorithms. Instead, an algorithm is the description of a computational procedure, which takes *any* instance as an input, and provides an answer with its proof (it might even fail to terminate if the problem is undecidable).

9.4 Complexity

The interest in grouping infinite sets of instances into problems, and to only consider algorithms that can potentially solve all instances of a problem, is to provide asymptotic answers with respect to algorithmic complexity. We are mainly interested in three points of view. First, what is the complexity of an algorithm for a given problem? This is discussed in Sect. 1.4.3.1-1.4.3.3, and, via worst-case complexity, aims to establish an upper bound for the asymptotic complexity of an algorithm as the instance size increases.

---

[1] See INF311.

Secondly: what is the algorithm that performs most efficiently on a given problem? This point of view goes under the name of *problem complexity*, and aims to establish a lower bound for the asymptotic worst-case complexity of the best algorithm for solving the problem as the instance size increases.

Typically, in the first case, we want to establish results like "the complexity of Prim's algorithm is $O(n^2)$." In the second case, we want to establish results like "the MST problem has polynomial complexity". Notice that, in the second case, there might be other algorithms that take an exponential time to solve the MST in the worst-case (such as for example listing all spanning trees), but we only focus on the best. The second case is an abstraction of the first: whereas we looked at single algorithms in the first case, we look at the best over *all* algorithms for a given problem in the second case.

9.5 Easy and difficult problems

The third point of view abstracts problems too. We group problems into *problem classes* and classify them by algorithmic efficiency: we have a problem class **P** of all decision problems that can be solved in worst-case asymptotic time bounded by a polynomial in the instance size, and a problem class **NP** of all decision problems whose YES certificates can be verified in worst-case asymptotic time bounded by a polynomial in the instance size (notice we do not require any condition on NO certificates), as well as many other problem classes. Informally, **P** is the class of problems that are "easy" to solve, whereas **NP** is the class of problems whose solutions can be verified efficiently.

So we have a formal way to say a problem is easy (it can be solved in polynomial time), and a formal way to say the solutions of a problem are easy to check. How about a formal way to say that a problem is difficult? In [10], Garey and Johnson argue that a convenient way to do so would be to say that a problem is difficult if no-one could solve it efficiently to date. The formalization of this concept is **NP**-hardness.

9.5.1 Reductions

Suppose we are given a new problem P for which we have to conceive and implement a solution algorithm. We might notice and exploit a certain similarity between P and a problem we already know how to solve, say Q. For example, suppose P is the problem of finding a stable (see Sect. 3.3.1) with at least k vertices in a graph G, for given k and G (this problem is known as k-Stable). The similarity is usually derived theoretically; consider for example the following result.

9.5.1 Lemma
Given a graph $G = (V, E)$ and $U \subseteq V$, U is a stable set of G if and only if $\bar{G}[U]$ is a clique in $\bar{G}$.

Proof. ($\Rightarrow$) Since U is a stable in G, the edge set of $G[U]$ is empty by definition. Thus, in the complement graph, the edge set of $\bar{G}[U]$ is complete, i.e. $\bar{G}[U]$ is a clique in G. The ($\Leftarrow$) direction is symmetric. □

According to Lemma 9.5.1, we can take Q to be the problem of finding a clique in G with at least k vertices (this problem is known as k-Clique). Now an algorithm for solving P, given a graph G and the integer k, is as follows:

1. construct the complement graph $\bar{G}$
2. solve Q on $\bar{G}$ to obtain a clique $C = (U, F)$ in $\bar{G}$
3. return U as a stable set with k vertices in G

Notice the structure of this algorithm: first we transform the input of P to the input of Q, then we solve Q, then we transform the output of Q back to the output of P. In our example, by Exercise 3.2.1 the complement graph can be constructed in polynomial time. If both input and output transformations can be carried out in polynomial time, we obtain a *polynomial reduction* of P to Q.

9.5.2 The new problem is easy

If we were able to carry out both transformations in polynomial time and Q were in **P**, we would have found a polynomial algorithm for P, and we would have thus shown that $P \in \mathbf{P}$ too.

We remark that, so far, no-one was able to show that the problem of finding a k-clique subgraph of G is in **P**. So our example with cliques and stables does not fall in this category.

9.5.3 The new problem is as hard as another problem

If we intuitively believe that P is a difficult problem, as seems to be the case with k-STABLE, it makes little sense to try and reduce it to an easy one. As mentioned above, one way to show P is difficult is to show that every other "intuitively difficult" problem polynomially reduces to P. In other words, if P could be solved efficiently, every "intuitively difficult" problem could also be solved efficiently by transforming an instance to an instance of P, then solve P, then transform the output back to the difficult problem (notice this exchanges the roles of P and Q in Sect. 9.5.1). In this setting, P being easy seems unlikely, for it would mean that everyone had a wrong intuition about the difficulty of all other "intuitively difficult" problems.

9.5.4 NP-hardness and NP-completeness

In order to formalize the discussion in the previous section, we define the problem P to be **NP**-*hard* if every problem in **NP** can be polynomially reduced to P. We define P to be **NP**-*complete* if it is **NP**-hard and also belongs to **NP**.

9.5.2 Example
*It turns out that k-CLIQUE is **NP**-complete. The problem* MAX CLIQUE, *which asks to find the greatest complete subgraph in a given graph, is **NP**-hard: since it is an optimization problem, and **NP** is a subclass of all decision problems, by definition* MAX CLIQUE *cannot belong to **NP**. We emphasize, however, that k-CLIQUE is the "decision version" of the* MAX CLIQUE *optimization problem, in the sense given in Sect. 9.2.1.*

This leaves us with just one basic question: how can we possibly hope to reduce *every problem* in **NP** to a specific problem P? S. Cook first provided a proof of **NP**-completeness: he encoded a Turing Machine with certain properties into a propositional formula. With Cook's approach, "every problem in **NP**" was replaced by the TM used for its solution. Once a single problem is shown to be **NP**-hard, it can also be used as a representative of "every problem in **NP**". Cook's theorem thus paved the way for decades of polynomial reduction based **NP**-hardness proofs.

Back to our stables and cliques, since by Example 9.5.2 we know that k-CLIQUE is **NP**-complete, reducing it to k-STABLE proves that the latter is also **NP**-complete. But this is easily done, since Lemma 9.5.1 is an "if and only if" result, and the polynomial reduction works both ways. So we conclude that k-STABLE is at least as hard to solve as the most difficult problem in **NP**.

9.5.5 The most celebrated open problem in computer science

It would be very nice if we were able to turn the definition of a difficult problem from “no-one else can solve it efficiently” to “it is impossible to solve it efficiently”. Translated in the formal terminology about **P** and **NP**, the latter corresponds to proving that $\mathbf{P} \neq \mathbf{NP}$, i.e. there is at least one problem in **NP** that cannot be solved in polynomial time *for sure*. $\mathbf{P} \neq \mathbf{NP}$ is the most celebrated conjecture in computer science. The first person who is able to establish whether $\mathbf{P} \neq \mathbf{NP}$ or $\mathbf{P} = \mathbf{NP}$ stands to gain hefty monetary rewards, too. Unfortunately, four decades of work in this sense yielded no definitive result yet. Since we value our intuition, and we would not like to think that someone in the future will be able to show that all the problems we thought difficult are really easy, people say it is unlikely that $\mathbf{P} = \mathbf{NP}$. The method for proving $\mathbf{P} = \mathbf{NP}$ would seem straightforward: simply find a polynomial algorithm for any **NP**-hard problem. On the other hand, no convincing proof methodology for even attempting to prove the converse was found to date.

J. Edmonds, who first proposed that efficient algorithms are those that run in time bounded by a polynomial in the instance size, once said during a seminar at the Institut Poincaré in Paris, that Gödel’s incompleteness theorem (see Sect. 7.5.2) is at play here, and that $\mathbf{P} = \mathbf{NP}$ is true but cannot be proved from the standard axioms.

9.5.6 The student’s pitfall

When asked to show the **NP**-hardness of problem P, every student seems to step in the following pitfall: he or she reduces P to another **NP**-hard problem Q, then claims the work is done. This is wrong! Remember:

1. If Q is in **P**, then a polynomial reduction $P \to Q$ proves P is also in **P**.
2. If Q is **NP**-hard, then a polynomial reduction $Q \to P$ proves P is also **NP**-hard.

By contrast, polynomially reducing Q to P when Q is in **P**, or P to Q when Q is **NP**-hard proves absolutely nothing.

9.6 Exact and heuristic algorithms

So what do we do after we decide whether a new problem P is easy or difficult? In the first case, the polynomial reduction automatically gives an exact algorithm for finding guaranteed solutions, as we saw in Sect. 9.5.1. We might perhaps wish to improve or fine-tune that algorithm, but essentially we are able to find exact solutions of P.

If P turns out to be difficult, we can either use a non-polynomial solution algorithm to find exact and guaranteed solutions, or use *heuristic algorithms*: these are methods based on intuitive common sense, which do their best to try and find solutions in a limited amount of time (or resources). If they do, these solutions are valid; but if they do not, this does not imply that the input instance is NO.

Heuristic algorithms for optimization problems provide solutions which may not be optimal, but are supposed to be “good enough”.

9.6.1 A heuristic method for MAX STABLE

Since MAX STABLE is **NP**-hard, and it seems unlikely that **P**=**NP**, we provide a greedy heuristic for solving this problem. Our heuristic is based on the idea that any maximum stable is also maximal. We recall that a stable is maximum if it has largest cardinality among all stable sets of the graph; and a stable is maximal if there is no larger stable containing it as a subset. In other words, the "maximum" property needs to be checked globally (which is time-consuming), whereas the "maximal" property only needs to be checked locally (which is much more efficient).

9.6.1 Exercise
Prove formally that any maximum stable is also maximal. Show that the converse is not necessarily true.

Although maximal stables are not necessarily maximum, they might be: after all, maximality is a necessary, if not sufficient, condition to be maximum. Moreover, maximal sets can be "grown" efficiently by simply adding all elements until it is possible to do so. We start from an empty stable U, then scan all vertices of V, adding them to U as long as no pair of vertices in U is adjacent to an edge. We can actually improve on this algorithm by ordering the vertices first.

```
1: $U = \varnothing$;
2: order $V$ by increasing values of $|N(v)|$;
3: while $V \neq \varnothing$ do
4:     $v = \min V$;
5:     $U \leftarrow U \cup \{v\}$;
6:     $V \leftarrow V \smallsetminus (\{v\} \cup N(v))$
7: end while
```

9.6.2 Exercise
Explain the presence of Step 6: does it improve the algorithm? If so, why? If not, why not?

Chapter 10

Sorting

Abstract. The searching problem and the sorting problem: complexity in the best case. Sorting algorithms: selection, insertion, merge and quick sort. Two-way partitioning.

Let $V = (v_1, \ldots, v_n)$ be a sequence with a natural order $<$ defined on its elements. The sequence V is *sorted* if:

$$\forall i < j \leq n \quad v_i \leq v_j, \tag{10.1}$$

and *unsorted* otherwise.

10.1 The searching problem

A fundamental question that often occurs in algorithms is whether a given set V contains a given element u. This is a decision problem with input V, u, called the Searching Problem. If V is unsorted, searching takes longer than if V is sorted, as shown below. In this chapter we discuss methods for sorting linear data structures.

10.2 Searching unsorted and sorted arrays

If $V = (v_1, \ldots, v_n)$ is stored as a linear data structure, say an array, and this array is unsorted, the only possible approach is brute force: verify each element of V in turn, and stop with YES when v is found, or terminate with NO at the end of the array. This method is obviously $O(n)$.

If the array were sorted, on the other hand, we could proceed using bisection. Here is the recursive pseudocode for `bisection(`V, w`)`.

```
Require: $V = \{v_1, \ldots, v_n\}$ is sorted
if $V = \varnothing$ then
    return NO
end if
let $i = \lceil \frac{n}{2} \rceil$
if $u = v_i$ then
    return YES
else if $u < v_i$ then
```

```
    let $V = \{v_1, \ldots, v_{i-1}\}$
else if $u > v_i$ then
    let $V = \{v_{i+1}, \ldots, v_n\}$
end if
return bisection($V, u$)
```

It is well known[1] that the worst-case complexity of `bisection` is $O(\log n)$.

So, if an algorithm needs to repeatedly test for membership in V, it makes sense to invest some CPU time to sort V first, prior to calling the membership test procedure.

10.3 The sorting problem

The Sorting Problem (SP) is as follows. Given a sequence $s = (s_1, \ldots, s_n)$ of elements of a set S endowed with a total order $<$, find a permutation (see Sect. 7.3) π of n symbols such that $\pi s = (s_{\pi(1)}, \ldots, s_{\pi(n)})$ satisfies Eq. (10.1). In other words, we want to order s according to $<$.

10.3.1 Considerations on the complexity of SP

In Sect. 9.4, we discussed two types of complexity: the complexity of an algorithm (how long it takes to execute), and the complexity of a problem, i.e. how long does the *best* algorithm for the problem takes to execute. In Sect. 9.5 we went on to classify problems according to whether they are known or unknown to have polynomial complexity.

10.3.1.1 The best algorithm for a problem

In fact, however, we do not need to know the best possible algorithm that solves a given problem to state that the problem belongs to the class **P**: it suffices for this purpose to find at least *one* polynomial algorithm, even if it is not the best possible. This is rather lucky, because proving that a certain algorithm is best for a given problem seems to require looking at the infinite class of all algorithms solving the problem. This task is so difficult that, if one could prove that the best algorithm for solving an **NP**-complete problem (such as e.g. k-Stable) is exponential, one would have settled the $\mathbf{P} \neq \mathbf{NP}$ conjecture.

10.3.1.2 The $\Omega(\cdot)$ and $\Theta(\cdot)$ notations

We introduced in Sect. 1.4.3.3 the $O()$ notation to express asymptotic worst-case complexity: a function $t(n)$ is $O(p(n))$ if there is an $N \in \mathbb{N}$ such that, for all $n > N$, $t(n) \leq p(n)$. There is also a notation for the asymptotic *best-case complexity*: a function $t(n)$ is $\Omega(p(n))$ if there is an $N \in \mathbb{N}$ such that, for all $n > N$, $t(n) \geq p(n)$ — incidentally, if a function $t(n)$ is both $O(n)$ and $\Omega(n)$, we say it is $\Theta(n)$. Best-case complexity is the correct notation for the difficult task we mentioned above: can we say that SP is $\Omega(p(n))$ for some function p?

10.3.2 Best-case complexity of SP

SP is one of those few cases where we can say something about the best-case complexity of a problem. We must, however, assume no prior knowledge of the type of data stored in S.

---

[1] See INF311.

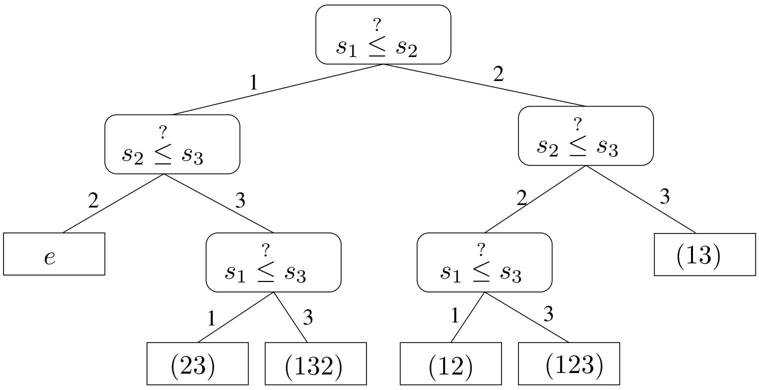
More precisely, we assume that the only way to conclude that $u < v$ in $O(1)$ is to call the machine language instruction for comparing bytes a constant number of times. This may not always be the case: for example, if we knew a priori that $S = \{0, 1\}$, we could conclude that $u < v$ in $O(1)$ by testing whether $u \neq v$ and $u = 0$.

Any sorting algorithm capable of dealing with any input set S must be a set of instructions (tests, loops, etc.) containing a sufficiently large number of *comparisons*: these are tests establishing whether $u < v$ or not. Informally, our requirement aims at generality: if our algorithm must be able to cater for all data types, then it must make use of comparisons. This endows SP with sufficient structure to make us able to reason on its best-case complexity.

10.3.2.1 The sorting tree

Accordingly, we can describe any comparison-based sorting algorithm via a *sorting tree*, which represents the logical flow of the sorting algorithm based on the comparisons only.

10.3.1 Example
Here is a sorting tree for sorting $s = (s_1, s_2, s_3)$.



$s_1 \overset{?}{\leq} s_2$
1
2
$s_2 \overset{?}{\leq} s_3$
$s_2 \overset{?}{\leq} s_3$
2
3
2
3
e
$s_1 \overset{?}{\leq} s_3$
$s_1 \overset{?}{\leq} s_3$
(13)
1
3
1
3
(23)
(132)
(12)
(123)

In the picture above, e stands for the identity permutation; the other permutations are expressed in cycle notation (see Sect. 7.3.1.3).

Sorting trees represent the possible ways to chain comparisons as to sort all possible input sequences of a given size; moreover, as mentioned earlier, any comparison-based sorting algorithm running over input of given size corresponds to a particular sorting tree. In other words, the set of all execution traces out of any possible comparison-based sorting algorithm is a subset of all sorting trees. Hence, the best-case complexity can be defined in terms of the best sorting trees.

10.3.2.2 Formalizing the idea

Let $\mathbb{T}_n$ be the set of all sorting trees for sequences of length n. Different inputs lead to different permutations in the leaf nodes of each sorting tree. For a sorting tree $T \in \mathbb{T}_n$ and a permutation π, we denote by $\ell(T, \pi)$ the length of the path in T from the root to the leaf containing π. For each $n \geq 0$ we can express the best-case complexity for SP as:

$$B_n = \min_{T \in \mathbb{T}_n} \max_{\pi \in S_n} \ell(T, \pi).$$

We remark that sorting trees are binary trees, since there are only two answers (YES or NO) to each comparison. Notice that a binary tree with depth bounded by k has at most 2^k nodes. Let T^* be the sorting tree of the best sorting algorithm: its number t of nodes must then be at most 2^{B_n}. Moreover, since any sorting tree lists all $n!$ possible permutations in its leaves, it must have at least $n!$ nodes. Hence $n! \le t \le 2^{B_n}$, whence $n! \le 2^{B_n}$, which implies

$$B_n \ge \lceil \log n! \rceil .$$

By Stirling's approximation formula [6], $\log n! = n \log n - \frac{1}{\ln 2} n + O(\log n)$, so we can conclude that B_n is bounded below by a function proportional to $n \log n$, i.e. B_n is $\Omega(n \log n)$.

10.4 Sorting algorithms

There are scores of general-purpose algorithms for sorting a sequence $s = (s_1, \ldots, s_n)$ of elements from a totally ordered set S. Many work well with some sequences but not with others. Here we only discuss a couple of the simplest algorithms (selection and insertion sort), together with two of the best (merge and quick sort).

10.4.1 Selection sort

In SELECTION SORT we start from $s = (s_1, \ldots, s_n)$ and end with a sorted sequence t, initially empty. We iteratively select the minimum element of s, move it to the leftmost free slot in t, and remove it from s. Since selecting the minimum from an unsorted array requires scanning the whole array, this takes $O(n^2)$ in the worst case.

10.4.1 Example
Let $s = (3, 1, 4, 2)$ *and* $t = \varnothing$. SELECTION SORT *performs the following sequence of steps:*

$$\begin{aligned} (3, \boxed{1}, 4, 2), \varnothing &\rightarrow (3, 4, \boxed{2}), (1) \\ &\rightarrow (\boxed{3}, 4), (1, 2) \\ &\rightarrow (\boxed{4}), (1, 2, 3) \\ &\rightarrow \varnothing, (1, 2, 3, 4). \end{aligned}$$

10.4.2 Insertion sort

INSERTION SORT is somehow dual to SELECTION SORT: instead of choosing the minimum from s, we iteratively choose the leftmost element of s, insert it in t at the correct position, and remove it from s. Since finding the correct position in t requires scanning t, this takes $O(n^2)$ in the worst case. Empirically, INSERTION SORT is known to be fast for small values of n.

10.4.2 Example
Let $s = (3, 1, 4, 2)$ *and* $t = \varnothing$. INSERTION SORT *performs the following sequence of steps:*

$$\begin{aligned} (\boxed{3}, 1, 4, 2), \varnothing &\rightarrow (\boxed{1}, 4, 2), (3) \\ &\rightarrow (\boxed{4}, 2), (1, 3) \\ &\rightarrow (\boxed{2}), (1, 3, 4) \\ &\rightarrow (1, 2, 3, 4). \end{aligned}$$

10.4.3 Exercise
Implement INSERTION SORT *in Java. What data structure did you employ for t?*

10.4.3 Merge sort

MergeSort is a recursive algorithm of the Divide-and-Conquer class.

10.4.3.1 Divide and conquer

Divide-and-Conquer names a family of algorithms that split a complex problem into two or more subproblems having the same structure, recursively solve each of them (the recursion is on the problem size), then recombine the partial solutions from both subproblems to construct a solution of the original problem. Divide-and-Conquer is essentially recursion (see Chapter 7) combined with bisection (see Sect. 9.2.1 and 10.2).

10.4.3.2 Pseudocode

The idea of MergeSort is to partition s mid-way and create two smaller unsorted subsequences s', s'', sort each of them recursively, and recombine the sorted subsequences so that the sorting of the resulting sequence is maintained. Here is `mergeSort(`s`)`.

```
if $|s| \le 1$ then
    return $s$; // base case
else
    $m = \lfloor \frac{|s|}{2} \rfloor$;
    $s' =$ mergeSort$((s_1, \ldots, s_m))$;
    $s'' =$ mergeSort$((s_{m+1}, \ldots, s_n))$;
    return merge$(s', s'')$;
end if
```

10.4.4 Example
If $s = (5, 3, 6, 2, 1, 9, 4, 3)$, *we first split* s *midway to obtain* $(5, 3, 6, 2)$ *and* $(1, 9, 4, 3)$. *These subsequences are sorted recursively, to yield* $s' = (2, 3, 5, 6)$ *and* $s'' = (1, 3, 4, 9)$*;* s', s'' *are then merged (see Example 10.4.6) to* $s = (1, 2, 3, 3, 4, 5, 6, 9)$.

10.4.5 Exercise
Show, by exhibiting a few examples, that splitting s *mid-way intuitively yields a more balanced recursion tree.*

10.4.3.3 Merging two sorted sequences

We still need to specify how to efficiently merge two sorted subsequences $r = (r_1, \ldots, r_h)$ and $t = (t_1, \ldots, t_k)$ into a single sorted sequence s. Here is $s =$`merge`(r, t).

```
$r_{h+1} = \infty$, $t_{k+1} = \infty$
$i = 1, j = 1, \ell = 1$
while $i \le h \vee j \le k$ do
    if $r_i \le t_j$ then
        $s_\ell = r_i$
        $i \leftarrow i + 1$
    else
        $s_\ell = t_j$
        $j \leftarrow j + 1$
    end if
    $\ell \leftarrow \ell + 1$
    return $s$
```

```
end while
```

Since all elements of r, t are scanned, `merge`(r, t) runs in $O(h + k)$ in the worst case.

10.4.6 Example
Let $r = (2, 3, 5, 6)$ *and* $t = (1, 3, 4, 9)$. *This is how* `merge` *works.*

$$\begin{array}{rcl}
\begin{matrix}(2,3,5,6)\\(\boxed{1},3,4,9)\end{matrix} & \to & \varnothing \\
\begin{matrix}(\boxed{2},3,5,6)\\({\color{gray}1},3,4,9)\end{matrix} & \to & (1) \\
\begin{matrix}({\color{gray}2},\boxed{3},5,6)\\({\color{gray}1},3,4,9)\end{matrix} & \to & (1,2) \\
\begin{matrix}({\color{gray}2,3},5,6)\\({\color{gray}1},\boxed{3},4,9)\end{matrix} & \to & (1,2,3) \\
\begin{matrix}({\color{gray}2,3},5,6)\\({\color{gray}1,3},\boxed{4},9)\end{matrix} & \to & (1,2,3,3) \\
\begin{matrix}({\color{gray}2,3},\boxed{5},6)\\({\color{gray}1,3,4},9)\end{matrix} & \to & (1,2,3,3,4) \\
\begin{matrix}({\color{gray}2,3,5},\boxed{6})\\({\color{gray}1,3,4},9)\end{matrix} & \to & (1,2,3,3,4,5) \\
\begin{matrix}({\color{gray}2,3,5,6})\\({\color{gray}1,3,4},\boxed{9})\end{matrix} & \to & (1,2,3,3,4,5,6) \\
\begin{matrix}({\color{gray}2,3,5,6})\\({\color{gray}1,3,4,9})\end{matrix} & \to & (1,2,3,3,4,5,6,9) = s.
\end{array}$$

10.4.7 Exercise
What is the purpose of setting r_{h+1} *and* t_{k+1} *to* ∞*? How would you implement this in Java, since* r *only has* h *elements and* t *only* k*?*

10.4.3.4 Worst-case complexity

Each call to `merge` has complexity $O(n)$, and, by bisection, the complexity of the DIVIDE-AND-CONQUER recursion is $O(\log n)$. This yields an overall worst-case complexity of $O(n \log n)$. This result is used in Sect. 10.5 below.

10.4.4 Quick sort

QUICKSORT is another DIVIDE-AND-CONQUER algorithm, somehow dual to MERGESORT. Whereas in MERGESORT we recurse first and work on subsequences later, in QUICKSORT we work on the sequence first and recurse later.

10.4.4.1 Pseudocode

The idea is the following: we choose a *pivot value* p (any element s_i of the unsorted sequence s will do, so we arbitrarily pick[2] $p = s_1$), then partition $s \smallsetminus \{s_1\}$ into two subsequences s', s'' of s such that s' contains all $s_i < s_1$, and s'' all $s_i \geq s_1$. Now we recursively sort s', s'', and let t be composed by the elements of s', followed by the pivot s_1, followed by the elements of s'' (we denote this operation by (s', p, s'')). The sequence t is obviously sorted, since s', s'' are sorted, and $s'_i < s_1 \leq s''_j$ for all i, j. Here is the pseudocode for `quickSort(`s`)`.

---

[2] For real-world implementations, this choice is suboptimal, see `http://en.wikipedia.org/wiki/Quicksort#Choice_of_pivot`.

```
if $|s| \leq 1$ then
    return $\varnothing$; // base case
else
    $p = s_1$ // the pivot
    $(s', s'') =$ partition$(s, p)$;
    $s' =$ quickSort$(s')$;
    $s'' =$ quickSort$(s'')$;
    $s = \leftarrow (s', p, s'')$;
end if
```

10.4.4.2 Partition

To complete the description of QuickSort, we have to exhibit a pseudocode for `partition`. This is easy: we scan $s \setminus \{s_1\}$: if $s_i < s_1$ we put it in s', if $s_i \geq s_1$ we put it in s''. This has worst-case complexity $O(n)$.

10.4.8 Exercise
Since we already disposed of s_1, why can't we simply say test $s_i > s_1$ above, instead of $s_i \geq s_1$?

10.4.9 Example
Here is the effect of `partition` *on* $(5, 3, 6, 2, 1, 9, 4, 3)$ *with pivot* $p = s_1 = 5$.

$$\begin{aligned}
(\mathbf{5}, \boxed{3}, 6, 2, 1, 9, 4, 3) &\rightarrow \varnothing, \varnothing \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3), \varnothing \\
(\mathbf{5}, 3, \boxed{6}, 2, 1, 9, 4, 3) &\rightarrow (3), \varnothing \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3), (6) \\
(\mathbf{5}, 3, 6, \boxed{2}, 1, 9, 4, 3) &\rightarrow (3), (6) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3, 2), (6) \\
(\mathbf{5}, 3, 6, 2, \boxed{1}, 9, 4, 3) &\rightarrow (3, 2), (6) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3, 2, 1), (6) \\
(\mathbf{5}, 3, 6, 2, 1, \boxed{9}, 4, 3) &\rightarrow (3, 2, 1), (6) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3, 2, 1), (6, 9) \\
(\mathbf{5}, 3, 6, 2, 1, 9, \boxed{4}, 3) &\rightarrow (3, 2, 1), (6, 9) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3, 2, 1, 4), (6, 9) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, \boxed{3}) &\rightarrow (3, 2, 1, 4), (6, 9) \\
(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) &\rightarrow (3, 2, 1, 4, 3), (6, 9).
\end{aligned}$$

10.4.10 Exercise
Implement `partition` *in Java so that, instead of producing two new sequences s', s'', it updates the input s so that it has all elements of s' first, then s_1, then all elements of s''. Make sure you allocate no new memory: the update must be "in place". [Hint: consider swapping pairs of elements in s].*

10.4.4.3 Worst-case complexity

Differently from (and worse than) MergeSort, the worst-case complexity of QuickSort is $O(n^2)$. This is because we can make the MergeSort recursion tree balanced by splitting s mid-way, but we cannot obtain the same on the QuickSort tree balanced. So: `partition` is $O(n)$ and the tree depth is also $O(n)$, which makes $O(n^2)$.

10.4.11 Exercise
Show that picking the median value in s does not necessarily make the QuickSort *recursive tree balanced.*

10.4.12 Proposition
The worst-case complexity of QuickSort *is $O(n^2)$.*

Proof. It suffices to exhibit an instance where QuickSort takes time proportional to n^2. Consider the input $(n, n-1, \ldots, 1)$ with pivot $p = s_1$. At recursion level 1, $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \varnothing$; at recursion level 2, $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \varnothing$; and so on, down to $p = 1$ (base case). Each call to `partition` takes $O(n)$, hence the result. □

10.4.4.4 Average-case complexity

The reason why QuickSort is so popular is that in practice, and for general input, it is the fastest sorting algorithm. Some theoretical support for this statement can be found by performing an average case complexity analysis, which yields $O(n \log n)$. We report a proof by P. Cameron [6] in this section. The proof is long, most of the steps are algebraic manipulations of equations, power series and differential equations, but I think that the overall proof structure is clear.

In order to understand Cameron's proof we have to introduce *recurrence relations*. These are relations over different elements of a sequence, expressed as a function of their positional index n in the sequence e.g. $q_1 = 0, q_n = q_{n-1} + 1$ is a recurrence relation satisfied by all integers in $\mathbb{N}$; $q_0 = 0, q_1 = 1, q_n = q_{n-1} + q_{n-2}$ is satisfied by the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 11, \ldots$. Sometimes a recurrence relation has a closed-form solution, e.g. $q_0 = 1, q_n = 3q_{n-1}$ has solution $q_n = 3^n$. In our setting, n is the length of the unsorted sequence s and q_n is the average number of comparisons taken by QuickSort.

10.4.13 Theorem
On average over all possible inputs, the complexity of QuickSort *is $O(n \log n)$.*

Proof. First, notice that `partition`(s) involves $n-1$ comparisons. Assume that the pivot $p = s_1$ is the k-th smallest element of s. It is then easy to show that the resulting sorting tree has a left subtree with q_{k-1} nodes and a right subtree with q_{n-k} nodes on average (since p is k-smallest in s, picture an unbalanced tree with the given proportions), so $q_n = q_{k-1} + q_{n-k}$. We average this over the n values that k can take, and obtain:

$$q_n = n - 1 + \frac{1}{n} \sum_{k=1}^{n} (q_{k-1} + q_{n-k})$$

Notice that in the sum $\sum_{k=1}^{n} (q_{k-1} + q_{n-k})$, each q_k occurs twice (to see this, consider the table below).

k	q_{k-1}	q_{n-k}
1	q_0	q_{n-1}
2	q_1	q_{n-2}
$\vdots$	$\vdots$	$\vdots$
$n-1$	q_{n-2}	q_1
n	q_{n-1}	q_0

Hence we can write:

$$q_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} q_k \tag{10.2}$$

We now consider the *formal power series*

$$Q(t) = \sum_{n\geq 0} q_n t^n. \tag{10.3}$$

If $Q(t)$ is known, then the value for each q_n can also be obtained as follows: differentiate $Q(t)$ n times with respect to t, set $t = 0$, and divide the result by n (convince yourself this works). We multiply each side of the recurrence relation (10.2) by nt^n and sum over all $n \geq 0$, to get:

$$\sum_{n\geq 0} nq_n t^n = \sum_{n\geq 0} n(n-1)t^n + 2\sum_{n\geq 0}\left(\sum_{k=0}^{n-1} q_k\right)t^n$$

We now replace each of these three terms: this will yield a neater expression for $Q(t)$.

1. Differentiate $Q(t)$ with respect to t and multiply by t to get an expression for the first term:

   $$t\frac{dQ(t)}{dt} = t\sum_{n\geq 0} nq_n t^{n-1} = \sum_{n\geq 0} nq_n t^n.$$

2. It is well known that:

   $$\sum_{n\geq 0} t^n = \frac{1}{1-t} \tag{10.4}$$

   for all $0 \leq t < 1$. Since $Q(t)$ is a formal power series, the values that t takes are not important, all else being equal — so we can accept a constraint $0 \leq t < 1$; in any case what is important to us are the coefficients $q_0, q_1, \ldots$. Differentiate Eq. (10.4) twice with respect to t, to get:

   $$\sum_{n\geq 0} n(n-1)t^{n-2} = \frac{2}{(1-t)^3}$$

   Multiply both members by t^2 to get an expression for the second term:

   $$\sum_{n\geq 0} n(n-1)t^n = \frac{2t^2}{(1-t)^3}. \tag{10.5}$$

3. Now for the third: the n-th term of the sum $\sum_{n\geq 0}(\sum_{k=0}^{n-1} q_k)t^n$ can be written as

   $$\sum_{k=0}^{n-1} t^{n-k}(q_k t^k)$$

   Hence, the whole sum over n can be written as the following product (convince yourself that this is true by testing a few finite examples by hand):

   $$(t + t^2 + t^3 + \ldots)(q_0 + q_1 t + q_2 t^2 + q_3 t^3 + \ldots)$$

   The first factor is $\sum_{n\geq 0} t^n = \frac{1}{1-t}$, and the second is simply the expression for $Q(t)$, hence the third term is $\frac{2tQ(t)}{1-t}$.

Putting all this together, we obtain a first-order differential equation for $Q(t)$:

$$tQ'(t) = \frac{2t^2}{(1-t)^3} + \frac{2t}{1-t}Q(t) \tag{10.6}$$

We remark that if we differentiate the expression $(1-t)^2Q(t)$ w.r.t. t, we get:

$$\frac{d}{dt}((1-t)^2Q(t)) = (1-t)^2Q'(t) - 2(1-t)Q(t). \tag{10.7}$$

We rearrange the terms of Eq. (10.6) to get:

$$tQ'(t) - \frac{2t}{1-t}Q(t) = \frac{2t^2}{(1-t)^3}. \tag{10.8}$$

We multiply Eq. (10.8) through by $\frac{(1-t)^2}{t}$ and get:

$$(1-t)^2Q'(t) - 2(1-t)Q(t) = \frac{2t}{1-t}. \tag{10.9}$$

The right hand side of Eq. (10.7) is the same as the left hand side of Eq. (10.9), hence we can rewrite Eq. (10.7) as:

$$\frac{d}{dt}((1-t)^2Q(t)) = \frac{2t}{1-t}. \tag{10.10}$$

Now, straightforward integration w.r.t. t yields:

$$Q(t) = \frac{-2(t + \log(1-t))}{(1-t)^2}. \tag{10.11}$$

The next step consists in writing the power series for log and $1/(1-t)^2$, rearrange them in a product, and read off the coefficient q_n of the term in t^n. Without going into details, this yields:

$$q_n = 2(n+1)\sum_{k=1}^{n} \frac{1}{k} - 4n \tag{10.12}$$

for all $n \geq 0$. For all $n \geq 0$, the term $\sum_{k=1}^{n} \frac{1}{k}$ is an approximation of:

$$\int_1^n \frac{1}{x}dx = \log(n) + O(1). \tag{10.13}$$

Thus, we finally get an asymptotic expression for q_n:

$$\forall n \geq 0 \quad q_n = 2n\log(n) + O(n) \tag{10.14}$$

This shows that the average number of comparisons taken by QuickSort is $O(n \log n)$. □

10.5 Exact complexity of SP

By Sect. 10.3.2.2, the best-case complexity of SP is $\Omega(n \log n)$. Notice we did not actually prove that an algorithm is best; instead, we found a smart way to represent all possible algorithmic outputs (or rather, traces of algorithmic executions where we only listed the comparisons with their respective mutual flows) as combinatorial structures (trees) and exhibited a way to minimize over them by means of lower and upper bounds in terms of the number of tree nodes.

Also, by Sect. 10.4.3.4, the worst-case complexity of an actual sorting algorithm is $O(n \log n)$. We can therefore conclude that the exact complexity of the sorting algorithm is $\Theta(n \log n)$.

10.6 Two-way partitioning

This is a sorting algorithm for ordered sets S with only two elements, say $0, 1$ with $0 < 1$. The input is a binary sequence $s = (s_1, \ldots, s_n)$, and the output is a reordering of s such that all zeroes come before all ones. The method is as follows: we keep two indices, i, j, initially set at $i = 1$ and $j = n$. If s_i, s_j are out of place, we swap them, otherwise we leave them fixed. We then increase i and decrease j while $i \leq j$.

Here is the pseudocode for `partition2way(`s`)`:

```
$i = 1$; $j = n$;
while $i \leq j$ do
  if $s_i = 0$ then
    $i \leftarrow i + 1$;
  else if $s_j = 1$ then
    $j \leftarrow j - 1$;
  else
    swap($s, i, j$);
    $i \leftarrow i + 1$;
    $j \leftarrow j - 1$;
  end if
end while
```

Its worst-case complexity is evidently $O(n)$.

10.6.1 Example
Here is how two-way partitioning sorts $(1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$*:*

1. *swap* $(1, 8)$, *get* $(\mathbf{0}, 0, 0, 1, 1, 0, 0, \mathbf{1}, 1, 1)$
2. *swap* $(4, 7)$, *get* $(0, 0, 0, \mathbf{0}, 1, 0, \mathbf{1}, 1, 1, 1)$
3. *swap* $(5, 6)$, *get* $(0, 0, 0, 0, \mathbf{0}, \mathbf{1}, 1, 1, 1, 1)$.

10.6.1 A paradox?

We proved in Sect. 10.5 that the exact asymptotic complexity of SORTING PROBLEM is $\Theta(n \log n)$, and here we go exhibiting an $O(n)$ sorting algorithm. This only looks like a paradox. We had warned that our reasoning in terms of sorting trees only held if no prior knowledge of the data type of S was available. In our case, we know that $S = \{0, 1\}$ aprioristically. In fact, S is so small that no comparison is necessary to determine whether $s_i < s_j$: since we expect all zeros to come before all ones, if $i < j$ it suffices that $s_i = 1$ or $s_j = 0$ to establish that s_i, s_j are out of place and have to be swapped.

Chapter 11

Searching

ABSTRACT. Data structures for searching efficiently. Binary search trees, balanced trees, heaps.

As mentioned in Sect. 10.1, searching a set V for a given element is a fundamental tool in algorithms. Chapter 10 dealt with this problem by sorting the linear data structure storing V as a pre-processing step to repeatedly searching V. This implies that V does not change between one search query to the next: V is a static data container. It often happens, however, that data containers might be dynamic, in the sense that V evolves between search queries. Since it would be inefficient to re-sort V as elements are added or deleted to it, in this chapter we discuss techniques for keeping V sorted, i.e. to modify insertion and deletion procedures so that if V is sorted before the changes, then V is also sorted after the changes.

We look at both decision and optimization problems: we are given a finite set V and an element v of the same data type as the elements in V, and ask the question, "does v belong to V"? Also, given a finite set V and a scalar function $\mu : V \to \mathbb{R}$, we might want to find the element of v with maximum or minimum μ value (see Sect. 9.2.1). The typical methods for searchable data structures are `find`, `insert`, `delete`, `min` and `max` — the meaning of each being evident from the name.

These methods provide a fundamental algorithmic toolbox, are called many times, and must therefore be very efficient. Efficiency is provided by storing V by means of appropriate data structures. Most often, these data structures are trees, with elements stored in such a way as to be found by exploring only one path from the root to a leaf. Since these paths are on average $O(\log |V|)$ long, searching usually only takes $O(\log |V|)$ or less, rather than the $O(|V|)$ necessary with linked lists.

Because we use tree data structures, method implementations will be recursive.

11.1 Notation

We usually think of such trees T as rooted and directed from their root $r(T)$ to their leaves. We refer to nodes instead of vertices, specifically to *parent*, *ancestor* and *children* nodes, also called *subnodes*. A tree is *k-ary* if every node has at most k subnodes, and *binary* if every node has either zero or 2 subnodes (a node has no subnodes if and only if it is a leaf). For a non-root node v of T, $P(v)$ denotes the *parent* of v. For a non-leaf node v of T, we distinguish the *left subnode* $L(v)$ and the *right subnode* subnodes $R(v)$. We also denote by $D(v)$ the depth of the tree rooted at v.

The set of nodes reachable from $L(v)$ is the *left subtree* rooted at $L(v)$, and the set of nodes reachable from $R(v)$ is the *right subtree* rooted at $R(v)$ (see Fig. 11.1).

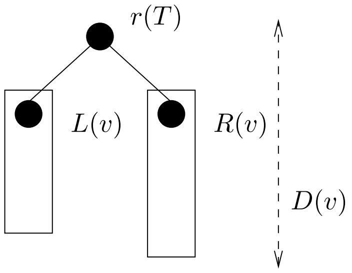


Figure 11.1: The notation used on search trees, where $v = r(T)$.

11.2 Binary search trees

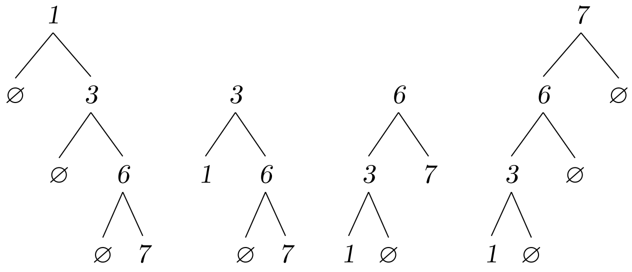
Binary Search Trees (BST) make it easy to store sorted sequences, which in turn make it easy to answer the question "does the sequence contain a certain element?" So we assume V is totally ordered by the relation $<$.

The principle underlying BSTs is that every non-leaf node $v \in V$ is such that

$$L(v) \leq v < R(v). \tag{11.1}$$

The order relation $<$ is the order on V mentioned above: this does NOT imply that the nodes all store scalars, although our examples will involve scalars for simplicity. All methods (`find`, `insert`, `delete`, `min`, `max`) are $O(\log n)$ on average and $O(n)$ in the worst case, where $n = |V|$.

11.2.1 Example
The following are all valid BSTs storing $V = \{1, 3, 6, 7\}$.



Naturally, some (those with smaller depth), yield more efficient searches than others.

All recursive BST methods are designed so that the base case is on the leaf nodes. In particular, they do nothing on empty nodes, which are simply implemented as `null` reference.

11.2.1 BST `min` and `max`

Here follows the recursive function `min(`v`)` for finding the minimum element in the subtree rooted at v.

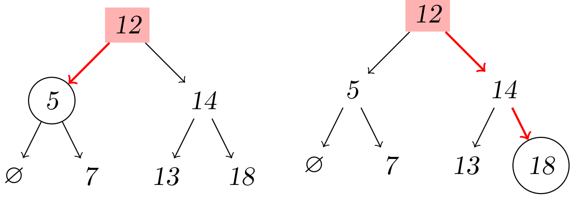
```
if $L(v) = \varnothing$ then
    return $v$;
else
    return $\min(L(v))$;
end if
```

Finding the minimum element in V is obtained by calling `min(`r`)`, where r is the root of the tree. This code simply follows the leftmost path as long as it is possible. The returned element is minimum by Eq. (11.1).

The method for finding maximum is similar.

```
if $R(v) = \varnothing$ then
    return $v$;
else
    return $\max(R(v))$;
end if
```

11.2.2 Example
Finding the minimum and maximum of $V = \{12, 5, 14, 7, 13, 18\}$.



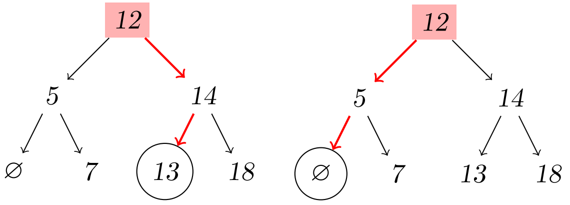
11.2.2 BST `find`

Here is the recursive function `find(`v`)`. The special marker **not_found** might be implemented as a `null` value or as raising a Java exception.

```
ret = not_found;
if $v = k$ then
    ret = $v$;
else if $k < v$ then
    ret = find($k$, L($v$));
else
    ret = find($k$, R($v$));
end if
return ret;
```

Finding v in V is obtained by calling `insert(`r`)`.

11.2.3 Example
Successfully finding 13 *in* $V = \{12, 5, 14, 7, 13, 18\}$ *(left) and unsuccessfully searching for* 1 *(right).*



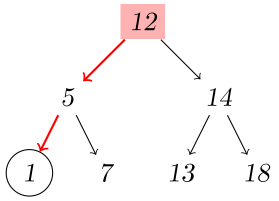
11.2.3 BST `insert`

Here is the recursive function `insert(`w`,`v`)`. It takes as input the element w to insert into the subtree rooted at v. The marker **already_in_set** can be implemented as raising a Java exception, or simply doing nothing.

```
if $w = v$ then
  return already_in_set;
else if $w < v$ then
  if $L(v) = \varnothing$ then
    $L(v) = w$;
  else
    insert($w, L(v)$);
  end if
else
  if $R(v) = \varnothing$ then
    $R(v) = w$;
  else
    insert($w, R(v)$);
  end if
end if
```

To insert w into V, simply call `insert(`w`,`r`)`.

11.2.4 Example
Inserting a 1 *into* V.



11.2.4 BST `delete`

Deletion is possibly the only nontrivial operation of a BST. Deleting a leaf node w is easy: it can simply be removed together with its incoming arc $(P(w), w)$ (Fig. 11.2, left). If $R(w) = \varnothing$ and $L(w) \neq \varnothing$, replace w with $L(w)$ (Fig. 11.2, middle).

11.2.5 Exercise
Prove that this case of deletion (Fig. 11.2, middle) satisfies Eq. (11.1).

Similarly, if $L(w) = \varnothing$ and $R(w) \neq \varnothing$, replace w with $R(w)$ (Fig. 11.2, right). Of course "replacing" here has a precise meaning: we define first the utility function `unlink(`w`)`: applied to node w, this function completely disconnects w from the tree.

```
let $P(w)$ =null
let $L(w)$ =null
let $R(w)$ =null
```

Replacing node w with u means to connect u to the same parent and subnodes of w. Here is `replace(`w, u`)` (see Fig. 11.3).

```
if $R(P(w)) = v$ then
```

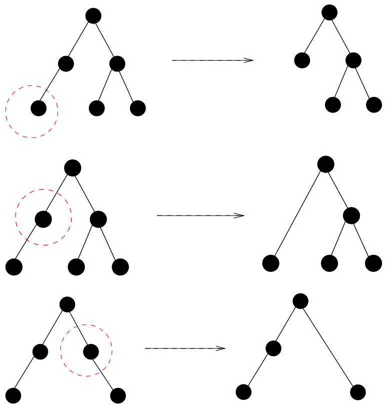


Figure 11.2: Deletion of BST nodes: easy cases.

```
    $R(P(w)) \leftarrow u$ // $u$ is a right subnode
else
    $L(P(w)) \leftarrow u$ // $u$ is a left subnode
end if
if $u \neq \varnothing$ then
    $P(u) \leftarrow P(w)$
end if
unlink($w$)
```

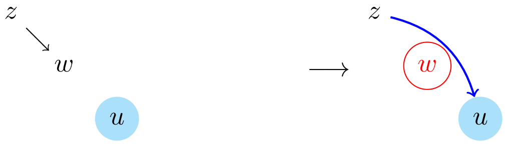


Figure 11.3: Replacing a node w with a node u.

11.2.4.1 Deleting a node with both subnodes

If w has 2 non-null subnodes, deletion becomes slightly more complicated. We swap the value of w and of the minimum element u of the right subtree of w and then delete u, which ends up being one of the easy deletion cases above, since it has a `null` left subnode. Here is `replaceValueMinRight(`w`)`, which returns the node u to be subsequently deleted.

```
$T'$ =right subtree rooted at $w$
$u = \min T'$
swap the values of $w$ and $u$
return $u$
```

To show that this works, we have to show that Eq. (11.1) holds in the resulting BST. By Eq. (11.1), the minimum element of a BST is always the leftmost node without a left subtree.

11.2.6 Exercise
Prove the previous statement.

Since u is in the right subtree of w, by Eq. (11.1) $u > w$ and also u is greater than all the elements in the left subtree of w. Also, since u is minimum in the right subtree, u is smaller than all other elements of the right subtree. So replacing w with u yields a BST where the new root is greater than (or equal to) all nodes in its left subtree, and smaller than all nodes in its right subtree. Thus Eq. (11.1) holds in the new BST.

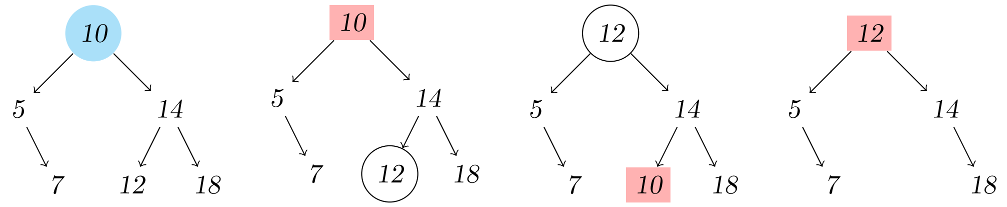
11.2.7 Exercise
Are there any other possibilities for deleting v from a BST in such a way that Eq. (11.1) *holds?*

11.2.4.2 Putting it all together

Here is the code for `delete`(w, v), which deletes element w from the tree rooted at v.

```
if $w < v$ then
  delete$(w, L(v))$;
else if $w > v$ then
  delete$(w, R(v))$;
else
  // base case of recursion, $w = v$
  if $L(v) = \varnothing \wedge R(v) = \varnothing$ then
    unlink($v$)
  else if $L(v) = \varnothing \wedge R(v) \neq \varnothing$ then
    replace($v, R(v)$)
  else if $L(v) \neq \varnothing \wedge R(v) = \varnothing$ then
    replace($v, L(v)$)
  else
    $u$ =replaceValueMinRight($v$)
    delete($u, v$) // an easy case
  end if
end if
```

11.2.8 Example
In order to delete the element 10 *from the tree rooted at* r*, we call* `delete`$(10, r)$ *on the following tree.*



The tree *Minimum elt.* *Swap values* *Delete u*

11.2.5 Complexity

All the recursive methods for BSTs have worst-case complexity proportional to the length of the longest path from the root to a leaf. If the tree is balanced, this is $O(\log n)$, otherwise it is $O(n)$. Notice we supplied no method for balancing a BST yet. Inserting $1, 3, 6, 7$ in an empty BST in this order yields the unbalanced BST in Fig. 11.4.

11.3 AVL trees

Adelson-Velskii-Landis (AVL) trees are BSTs with a mechanism for balancing the tree. For a BST T rooted at v, we let $B_T(v)$ be the depth difference between left and right subtrees of T rooted at v (we

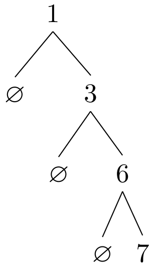


Figure 11.4: An unbalanced BST.

drop the subscript T when it is clear from the context):

$$B_T(v) = D(L(v)) - D(R(v)).$$

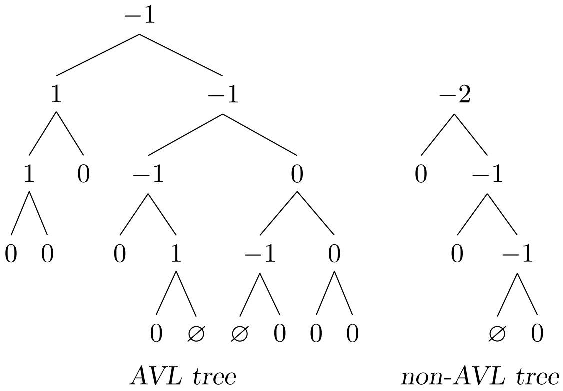
An AVL tree T always has the following property:

$$\forall v \in T \; B_T(v) \in \{-1, 0, 1\}. \tag{11.2}$$

This means that $B(r(T))$ is $O(1)$ (asymptotics on the number n of nodes in the BST), which implies that all recursive BST methods are $O(\log n)$.

11.3.1 Exercise
Prove formally that in a BST with n *nodes, Eq.* (11.2) *yields a maximum root→leaf path length of* $\log n$.

11.3.2 Example
Here are examples of AVL and non-AVL BSTs. The nodes are labelled with $B(v)$.



11.3.1 Balance-independent methods

The methods `min`, `max`, `find` do not change the BST, so if Eq. (11.2) holds before the call, it also holds after the call.

11.3.2 Balance-dependent methods

The methods `insert` and `delete` either add or remove a node from the BST. This means that if Eq. (11.2) holds before the call, after the call we might have $B(v) \in \{-2, -1, 0, 1, 2\}$. In this section we introduce methods for rebalancing based on tree rotation.

We consider a BST rooted at u, with a left subtree called α, and right subtree rooted at $v = R(u)$, which itself has a left subtree called β and a right subtree called γ. A *left rotation* rearranges the arcs so that the BST is rooted at v, its right subtree is γ, and its left subtree is rooted at u, with has left subtree α and right subtree β, as shown in Fig. 11.5. The *right rotation* is the inverse transformation. The operation of rotating a BST is such that:

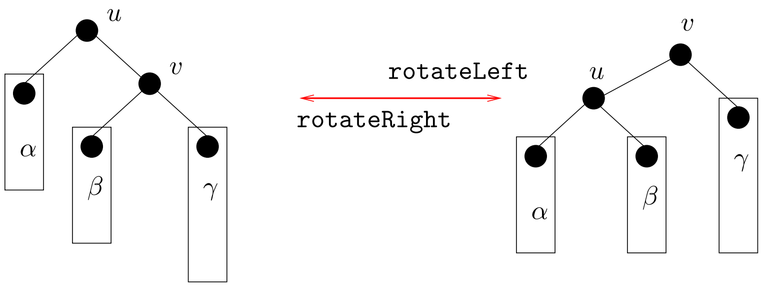


Figure 11.5: `rotateLeft` transforms the left BST T into the right one T'. `rotateRight` transforms the right BST T' into the left one T.

1. it is invariant with respect to the BST property (Eq. (11.1));
2. If $B_T(u) = -2$, then $B_{T'}(v) = 0$; if $B_{T'}(v) = 2$, then $B_T(u) = 0$.

Thus, it can be used for rebalancing BSTs having a shape like T or T'.

11.3.2.1 Tree rotation properties

In order to prove the properties above, we introduce an algebraic notation for BSTs: we let a BST rooted at r be denoted by $\langle L(r), r, R(r)\rangle$. For example, the left BST in Fig. 11.5 is $T = \langle \alpha, u, \langle \beta, v, \gamma\rangle\rangle$, and the right one is $T' = \langle\langle \alpha, u, \beta\rangle, v, \gamma\rangle$. Thus, we have

- `rotateLeft`$(T) = T'$
- `rotateRight`$(T') = T$

Directly by definition, we infer that `rotateRight`(`rotateLeft`(T)) = `rotateLeft`(`rotateRight`(T)) = T.

11.3.3 Proposition
If T is a BST, `rotateLeft`(T), `rotateRight`(T') *are BSTs, i.e. they satisfy Eq. (11.1).*

Proof. The node order in the tree only changes for u and v. In T, $v = R(u)$, which by Eq. (11.1) implies $u < v$. In `rotateLeft`(T), $u = L(v)$, which also implies $u < v$. The proof for T' is similar. □

Now suppose that the depths of α, β is h, and the depth of γ is $h+1$. Then the depth difference $B(u)$ of the tree $\langle \alpha, u, \langle \beta, v, \gamma\rangle\rangle$ is -2, and, similarly, $B(v) = 2$.

11.3.4 Proposition
$B(r($`rotateLeft`$(T))) = B(r($`rotateRight`$(T'))) = 0$.

11.3.5 Exercise
A proof sketch for Prop. 11.3.4 is that since the subtrees α, γ are swapped, and those subtrees are the cause of the unbalance, the rotated tree is balanced. Formalize this proof sketch.

11.3.2.2 The remaining cases

Consider the case of a tree $S = \langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle$ where $D(r(\alpha)) = D(r(\beta)) = h$, and $D(r(\gamma) = h + 1$. This does not fall into either of the cases T, T' above. Because γ is the left subtree of v, it has depth $h + 3$, while the depth of α is $h + 1$, so $B_S(u) = -2$ and the tree is unbalanced (see Fig. 11.6). Rotating S, however, only exchanges the roles of α, β, fixing γ; this results in another unbalanced tree. To deal with tree shape S, we "break up" the subtree γ, writing it as $\langle \lambda, r(\gamma), \mu \rangle$. Assume first that λ has depth h and μ has depth $h - 1$, as shown in Fig. 11.7, so that $B(r(\gamma)) = 1$, and call this tree shape S'. We now recognize the BST rooted at v as a tree of shape T'. Although it is not unbalanced itself, as $B(v) = 1$, we rotate it right: the effect of this operation will be to "shift" the unbalance of the tree shape S' from the subtree γ to the subtree β. Thereafter, the resulting BST will be still unbalanced but will have shape S, and a left rotation will balance it (see Fig. 11.8).

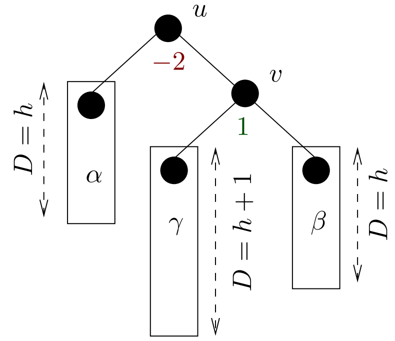


Figure 11.6: The unbalanced tree shape S.

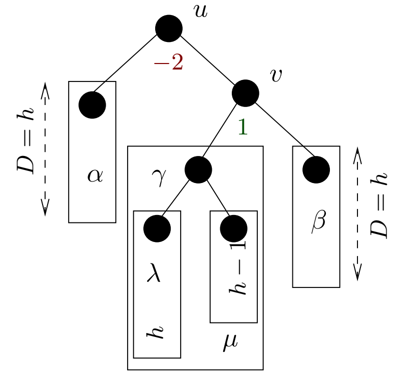


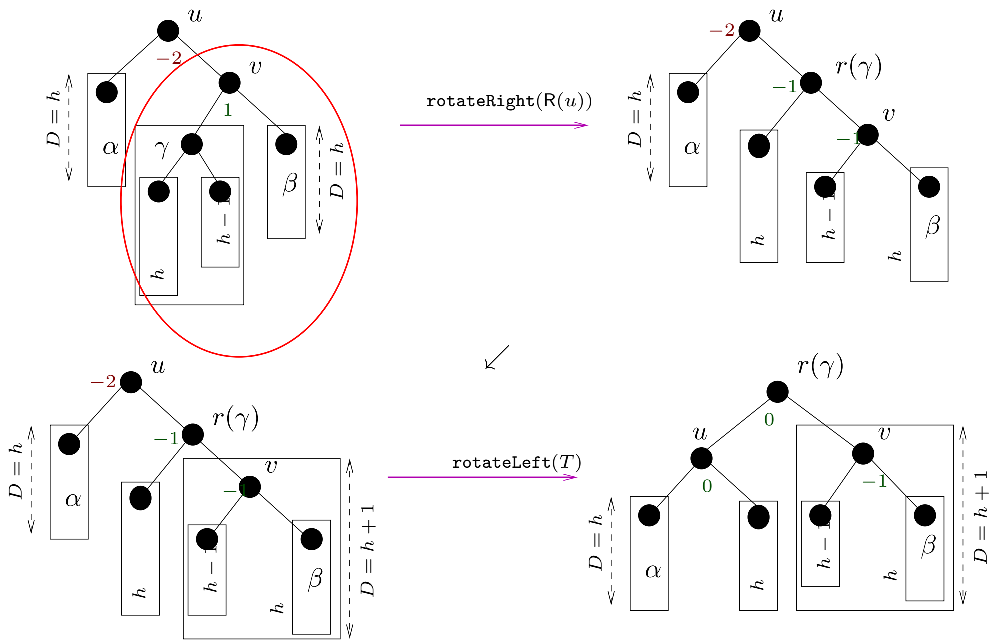
Figure 11.7: The unbalanced tree shape S'.

Another tree shape S'', which has λ with depth $h - 1$ and μ with depth h, is symmetric with respect to S' and can be handled in exactly the same way, i.e. right rotation of the right subtree of the root, followed by a left rotation of the resulting BST (Fig. 11.9).

A last unbalanced tree shape $\bar{S}$ rooted at u needs handling. This is symmetric with S, having $B(u) = 2$ (see Fig. 11.10). Following the same pattern as above, we can distinguish two further symmetric unbalanced tree shapes $\bar{S}', \bar{S}''$, both subsumed by $\bar{S}$. Both can be rebalanced in the same way, by rotating $L(u)$ left first, and then rotating the resulting BST right.

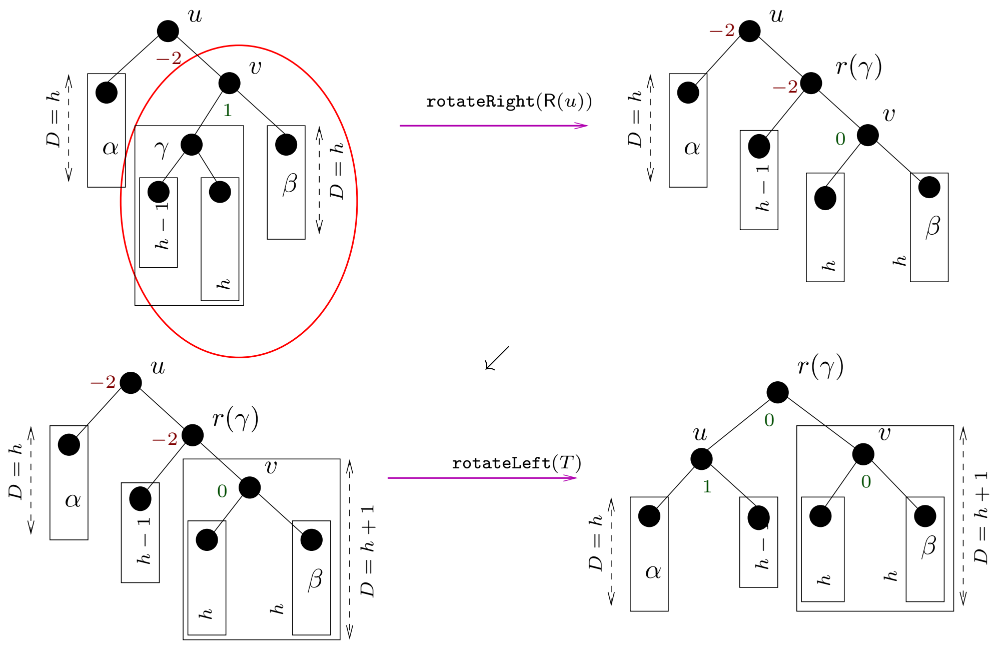
11.3.6 Exercise

Convince yourself that $T, T', S, \bar{S}$ exhaust the possible tree shapes that can occur after an `insert` *or*



u
v
−2
1
γ
α
β
$D = h$
h
$h-1$
rotateRight(R(u))
$r(\gamma)$
−1
rotateLeft(T)
0
$D = h+1$

Figure 11.8: Dealing with tree shape S': a right rotation of the right subtree followed by a left rotation of the result rebalances the BST.



u
v
−2
1
γ
α
β
$D = h$
h
$h-1$
rotateRight(R(u))
$r(\gamma)$
0
rotateLeft(T)
$D = h+1$

Figure 11.9: Tree shape S'', symmetric with S' and handled in the same way.

`delete` *operation in a BST.*

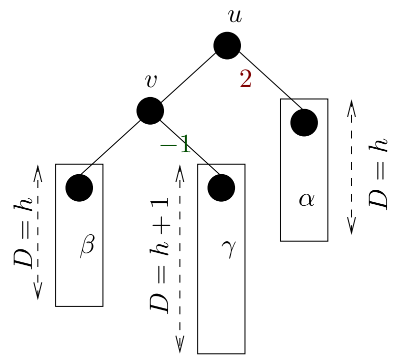


Figure 11.10: The unbalanced tree shape $\bar{S}$.

11.4 Heaps

A *heap* is a basic tree-like data structure that is specially conceived to implement the concept of a *priority queue* efficiently.

11.4.1 Priority queues

A priority queue is simply a queue (see Sect. 4.3) with an additional node extraction mechanism. Specifically, we associate a scalar $p(v)$ (called *priority*) to each element v of the queue V, and want to be able to efficiently extract an element from highest priority. Accordingly, we introduce two new methods to the queue's standard set of methods: `max` and `popMax`. The former returns the priority of the element with highest priority, and the latter returns the element of highest priority and removes it from V. We also modify `insert`$(v, p(v))$ to also take the priority of v as input.

11.4.1 Exercise
Show that, if $p(v)$ is an integer specifying the order of entrance in the queue, `popMax` *has the same effect as the* `popFront` *method in standard queues.*

11.4.2 Heap properties

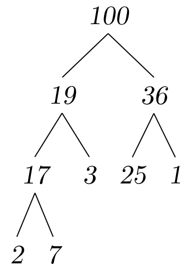
Having motivated heaps, we now look at them in more depth. A heap is a binary tree with the following properties.

1. All tree levels except perhaps the last one are fully filled; the last one is filled left-to-right (*shape property*).
2. Every node stores an element of higher property than its subnodes (*heap property*).

We remark that a heap is not a BST, as Eq. (11.1) is not necessarily satisfied.

Intuitively, the shape property ensures that the tree is balanced, and hence most depth-dependent methods are $O(\log n)$ instead of $O(n)$ (where $n = |V|$). The heap property induces the element of V having highest priority to be stored as the root node.

11.4.2 Example
An example of heap where $V \subseteq \mathbb{N}$ and the priority order is simply the usual integer order.



11.4.3 Proposition
If V is a heap, $\forall v \in V\ B_V(v) \in \{0,1\}$.

Proof. This follows trivially from the shape property. Since all levels are filled completely apart perhaps from the last, $B(Q) \in \{-1,0,1\}$. Since the last is filled left-to-right, $B(Q) \neq -1$. □

Thus, a heap is a balanced binary tree. Since a heap is not a BST (and thus not an AVL tree), we cannot use the same balanced insertion and deletion methods as for AVL trees.

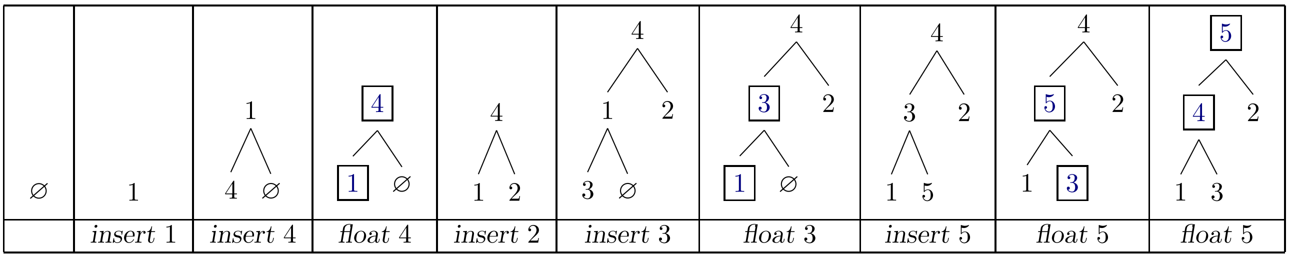
Notice that the shape property induces a linear order $\prec$ on the heap nodes, based on levels: for two nodes u, v in a heap, we let $u \prec v$ if and only if either the depth of u is smaller than the depth of v, or, if u, v are on the same level, if u is on the left of v. This order defines a notion of $\prec$-successor, which can also be extended to the $\prec$-last element in the heap: the $\prec$-successor position after the $\prec$-last element is either the leftmost "free slot" in the last level, if this is not completely filled, or the leftmost slot in a new level otherwise. This position is also called the *bottom of the heap.*

11.4.2.1 Insertion

In order to add a new element v with priority $p(v)$ to the heap V, we insert it at the bottom of the heap, then "float it up" the path to the root while its priority is higher than that of its parent. Here is the code for `floatUp(`v`)`, which floats a node v up a heap V rooted at r, until v's position satisfies the heap property.

```
while $v \neq r \land p(v) > p(P(v))$ do
    swap $v$ with $P(v)$
end while
```

11.4.4 Example
As an example, we insert $1, 4, 2, 3, 5$ *in an empty heap.*



11.4.5 Exercise
Write `floatUp` *as a recursive algorithm.*

11.4.6 Exercise
Prove that heap insertion maintains the shape and heap properties.

Because a heap is a balanced tree, insertion takes $O(\log n)$.

11.4.2.2 Maximum

Returning the highest prority over all heap elements is equivalent to simply returning the priority of the root element, because of the `floatUp` operation. Obviously, the complexity of the `max` method is $O(1)$.

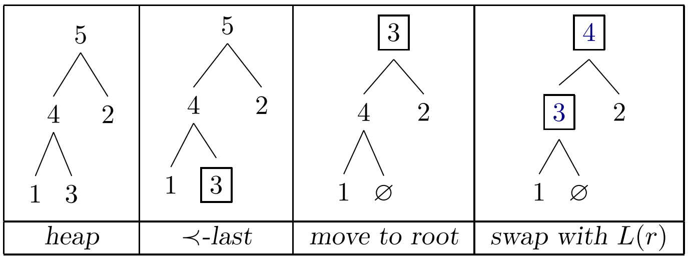
11.4.2.3 Popping the maximum

Since the highest priority element is at the root of the heap, we save the root for returning it later. We then overwrite the root with the $\prec$-last heap element, i.e. the rightmost "filled slot" on the last heap level, and unlink the latter (the `unlink` method can be borrowed from BSTs). Finally, we float the updated root down the heap while one of its subnodes have higher priority. Here is the `floatDown(`v`)` method.

```
while $v < \max(L(v), R(v))$ do
    swap $v$ with $\max(L(v), R(v))$
end while
```

In `floatDown`, we assume that if v has no left subnode, then $L(v), R(v)$ return $\varnothing$, and that $\varnothing$ has priority $-\infty$.

11.4.7 Example
Here is an example of popping the maximum element from the heap.



The `floatDown` operation takes $O(\log n)$ in the worst case, so `popMax` also takes $O(\log n)$.

11.4.8 Exercise
Prove that `floatDown` *preserves the shape property.*

11.4.2.4 Initial heap construction

In Example 11.4.4, we constructed a new heap by inserting all the element in an empty heap, one after the other. This has complexity $O(n \log n)$, since we must insert n element, and each insertion takes $O(\log n)$. This is suboptimal; consider the following procedure instead.

1. Insert elements in a binary tree V in their natural order, respecting the shape property but not the heap property.
2. For each $v \in V$, call `floatDown(`v`)`.

It should be obvious that the net effect of the above procedure is the same as inserting elements in a heap one by one, since the first step ensures the shape property is satisfied, whilst the second ensures the heap property is satisfied (by Exercise 11.4.8, the shape property is preserved).

A superficial worst-case analysis gives the above procedure at $O(n \log n)$: there are n elements, and each `floatDown` costs $O(\log n)$ by Sect. 11.4.2.3. There is a more refined analysis, however. The `floatUp(`v`)` and `floatDown(`v`)` methods take a CPU time proportional to the level ℓ of the node v. There are at most $\lceil \frac{n}{2^{\ell+1}} \rceil$ nodes at level ℓ, and $O(\log n)$ possible levels. Thus, the overall worst-case complexity is:

$$\begin{aligned}
\sum_{\ell=0}^{\lceil \log n \rceil} \frac{n}{2^{\ell+1}} O(\ell) &= O\left(n \sum_{\ell=0}^{\lceil \log n \rceil} \frac{1}{2^\ell} \right) \\
&\leq O\left(n \sum_{\ell=0}^{\infty} \frac{1}{2^\ell} \right) \\
&= O(2n) \\
&= O(n).
\end{aligned}$$

11.4.9 Exercise
Implement a Java heap with the following methods: `insert(`v`)`, `max()`, `popMax()`. *Implement the two versions of* `initialize(`V`)` *given in this section, and compare the CPU time they take over sets of 10, 100, 1000, 10000, 100000 elements. Do your empirical observations fit the theory?*

Chapter 12

Shortest paths

ABSTRACT. Shortest path problems and variants. Negative weights and negative cycles. Dijkstra's algorithm: simple and more refined pseudocodes, with complexity. Floyd-Warshall's algorithm.

Given an edge-weighted directed or undirected graph, the problem of finding shortest paths in the graph, in terms of the sum of the weights of the path edges, has dozens of applications, in logistics, communication networks, power networks, engineering,, computer science itself (shortest path computations are often sub-steps of more complex algorithms), and other fields.

Although in digraphs paths are technically known as walks, in this chapter we shall nonetheless call them paths for historical reasons. Similarly, we use the term cycle to possibly mean a circuit in a digraph. Moreover, undirected graphs occurring in path problems can be replaced by digraphs with pairs of antiparallel arcs (u, v) and (v, u) for every edge $\{u, v\}$ in the original graph G.

In the rest of this chapter, we assume that all graphs are connected, and that digraphs are strongly connected.

12.1 Basic literature

12.1.1 Problem variants

Although the concept of a shortest path is easy to grasp, there are several different formal variants of the shortest path problem. Here are some of these.

- The SHORTEST PATH PROBLEM (SPP). Given a directed or undirected graph $G = (V, A)$, a node or vertex $s \in V$, and a non-negative arc or edge weight function $c : A \to \mathbb{R}_+$, find c-shortest paths from s to all other vertices of V. This problem is in **P**.
- The SPP with unit weights is the SPP with $c : A \to \{1\}$, i.e. all arcs/edges have the same (unit) weight. This problem is in **P**.
- The NEGATIVE CYCLE PROBLEM (NCP) asks to determine whether G has a cycle of negative weight (the weigth of a cycle is the sum of the weights of the cycle edges). This problem is in **P**.

- The SPP with negative weights is the SPP with $c : A \to \mathbb{R}$, i.e. c can also take negative values. This problem is in **P**.
- The POINT-TO-POINT SHORTEST PATH PROBLEM (P2PSPP). Given G, c, and two distinct nodes or vertices $s, t \in V$, find a c-shortest path from s to t. This problem is in **P**.
- The ALL SHORTEST PATHS (ASP) problem asks to determine all shortest paths from u to v for any couple (u, v) of nodes or vertices in V. This problem is in **P**.
- The SHORTEST SIMPLE PATH (SSP) problem consists in finding the shortest simple path from s to t in G. This problem is **NP**-hard.
- The LONGEST PATH PROBLEM (LPP) asks to find the longest simple path from s to t in G. This problem is **NP**-hard.
- The UNDIRECTED SPP (USPP) with weights in N requires G to be an undirected graph and can be solved in linear time.

12.1.2 Algorithms

To every problem variant, there corresponds a specific algorithm.

- The SPP can be solved by Dijkstra's algorithm (see Sect. 12.4 below), which bears some resemblance to the GRAPH SCANNING algorithm (see Sect. 8.1) with Q implemented as a priority queue (see Sect. 11.4.1), and no restrictions about scanning a vertex more than once. Dijkstra's algorithm runs in polynomial time (simple implementation: $O(n^2)$).
- The SPP with unit weights is solved using BREADTH-FIRST SEARCH as explained in Sect. 8.2. BFS runs in linear time $O(n + m)$.
- The NCP and the SPP with negative weights are related. In fact, the issue with having negative weights in the SPP is that the path weight might be unbounded. More precisely, if there is a cycle of negative weight, any path from s to t can travel along the cycle as many times as desired (notice that the path is not required to be simple) to reduce its weight as much as desired. The Bellman-Ford algorithm, which runs in polynomial time $O(nm)$, scans all arcs repeatedly to identify either a shortest path tree or identify a negative weight cycle, and therefore solves both the NCP and the SPP with negative weights.
- The ASP can be solved using the Floyd-Warshall algorithm, which runs in polynomial time $O(n^3)$ and, interestingly, also solves the NCP.
- The algorithm for solving the USPP with nonnegative integer weights is given in [22].

The SSP and LSP can be solved either using brute force, or using an implicit enumeration method, such as the BRANCH-AND-BOUND (BB) algorithm, or using heuristics (see Sect. 9.6).

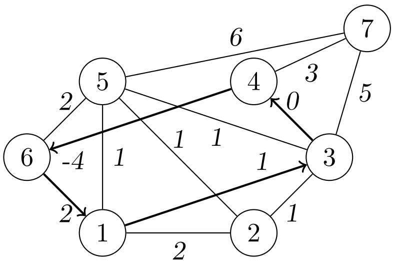
12.2 Weight functions

If F is a set of c-weighted arcs in the digraph $G = (V, A)$, the cost function $c : A \to \mathbb{R}$ can be extended to sets of arcs by setting

$$c(F) = \sum_{(u,v)\in F} c_{uv}.$$

12.2.1 Exercise
Prove formally that, if there exist a negative weighted circuit in the digraph $G = (V, A)$, *no walk* $P = (U, F)$ *attains the minimum on the function* c *extended to sets of arcs.*

12.2.2 Example
The cycle emphasized in the graph below has negative weight $1 + 0 - 4 + 2 = -1 < 0$.



If an arc weight function $c : A \to \mathbb{R}$ yields no negative cycles on a digraph $G = (V, A)$, it is called *conservative*.

12.3 The shortest path tree

Given a graph $G = (V, E)$, a conservative weight function $c : E \to \mathbb{R}$ and a source vertex $s \in V$, it is not immediately evident that the union of all shortest paths from s to all other vertices forms a spanning tree of G. After all, two paths from s to two vertices v, w might intersect at a single vertex u, and thus form a cycle including s and u. This is indeed possible; however, shortest paths need not be unique, and if two shortest paths from s to v, w form a cycle, then there must exist another shortest path from s to w which follows the shortest path from s to v until u, and forks towards w later (and similarly for the case where v, w are inverted).

12.3.1 Theorem
If c *is conservative, every initial subpath of a shortest path is a shortest path.*

Proof. Let P be a shortest path from s to v, and let u be a vertex in P different from u, v. Suppose, to get a contradiction, that the initial subpath P_0 of P from s to u is not a shortest path. Since G is connected, there is a shortest path from s to every other vertex in G, so let Q be a shortest path from s to u. Since Q is shortest and P_0 is not, we have $c(Q) < c(P_0)$. Now consider the path Q' from s to v consisting of Q followed by the path $P \smallsetminus P_0$ from u to v: the cost of Q' is $c(Q) + c(P \smallsetminus P_0) < c(P_0) + c(P \smallsetminus P_0) = c(P)$, which means that P is not shortest, against the assumption. So P_0 is a shortest path from s to u, as claimed. □

The solutions of all SPP variants are Shortest Path Trees (SPT), often encoded by means of two maps π, d: $\pi(v)$ stores the parent of vertex v in the tree rooted at s, and $d(v)$ is the weight of the shortest path from s.

12.3.2 Exercise
Adapt Thm. 12.3.1 to digraphs. The (directed) SPT should be oriented out of the source.

12.4 Dijkstra's algorithm

Dijkstra's algorithm solves the SPP: given a nonnegatively weighted digraph $G = (V, A)$ and a source node $s \in V$, find a SPT from the source to all other nodes.

12.4.1 Exercise
Prove that a nonnegative edge function is conservative.

12.4.1 Data structures

We label the nodes of the digraph $V = \{1, \ldots, n\}$ and maintain two integer arrays[1] π_v and d_v: for all $v \in V$, π_v is the parent node of v in the SPT rooted at s, and d_v is the weight of shortest path from s to v. Initially, $\pi_v = s$ for all v, $d_s = 0$ and $d_v = \infty$ for all $v \neq s$.

12.4.2 Exercise
Consider the star digraph on V with arc set $\{(s, v) \mid v \in V\}$, each arc with weight ∞. Show that there is only one possible SPT from s, that the predecessor of each $v \neq s$ is s, and that the weight of a shortest path from s to v is ∞ for each v.

12.4.2 Reach, settle and relax

As mentioned above, Dijkstra's Algorithm is similar to Graph Scanning with Q implemented as a priority queue (see Sect. 11.4.1), an update of the arrays π_v, d_v as the algorithm progresses, and no restriction about scanning a node more than once.

We are going to introduce a slightly different terminology in order to align with the current shortest path literature. A node $v \neq s$ such that $d_v \neq \infty$ is *reached*. A node $v \in V$ is *settled* when π_v, d_v no longer change during the rest of the algorithm's execution. After a node u is settled, each node v in its star is checked: if shortest paths through u should take the arc (u, v), then we update $p_v = u$ (the parent of v becomes u) and $d_v = d_u + c_{uv}$ (the weight of the shortest path to v is the weight to the shortest path to u plus the weight of the arc (u, v)). This update is also called *relaxing* the arc (u, v) (see Fig. 12.1). The code for `relax`(u, v), which also includes the check, is as follows.

```
if $d_u + c_{uv} < d_v$ then
    $d_v \leftarrow d_u + c_{uv}$;
    $p_v \leftarrow u$;
end if
```

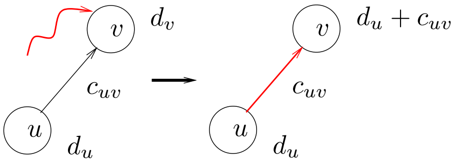


Figure 12.1: Relaxing the arc (u, v).

12.4.3 A simple implementation

With the terminology in place, the pseudocode is as follows.

---

[1] For clarity of exposition, we index these arrays starting from 1. In Java/C/C++, indexing would start from 0.

```
1: while $\exists$ unsettled nodes do
2:     Let $u$ be an unsettled node with minimum $d_u$;
3:     Settle $u$;
4:     for $(u,v) \in A$ do
5:         Relax $(u,v)$;
6:     end for
7: end while
```

12.4.3 Exercise
Prove that if $d_v = \infty$ *at Step 4, relaxing* (u,v) *will necessarily make* d_v *reached.*

12.4.4 Exercise
Copy the GRAPH SCANNING *algorithm and adapt it so it becomes an implementation of* DIJKSTRA'S ALGORITHM. *Prove that the two algorithms yield the same solutions for any SPP instance.*

12.4.5 Exercise
Convince yourself that each node is settled at exactly once.

12.4.6 Example
Fig. 12.2 shows a worked-out example of Dijkstra's algorithm finding the SPT on a digraph.

12.4.7 Exercise
Write down the data structures π, d *for each step of Example 12.4.6.*

12.4.3.1 Complexity

The worst-case complexity of a simple implementation for Dijkstra's algorithm is $O(n^2)$: each node is settled exactly once, so the outer loop at Step 1 is executed $O(n)$ times; finding the minimum element of V takes no longer than $O(n)$ independently of the data structure (less with a priority queue); and the inner loop at Step 4 is executed at worst $O(n)$ times, since any vertex can be adjacent to at most $n-1$ other vertices. Simple implementations are competitive in dense graphs, but graphs are often sparse in practice.

12.4.3.2 Correctness

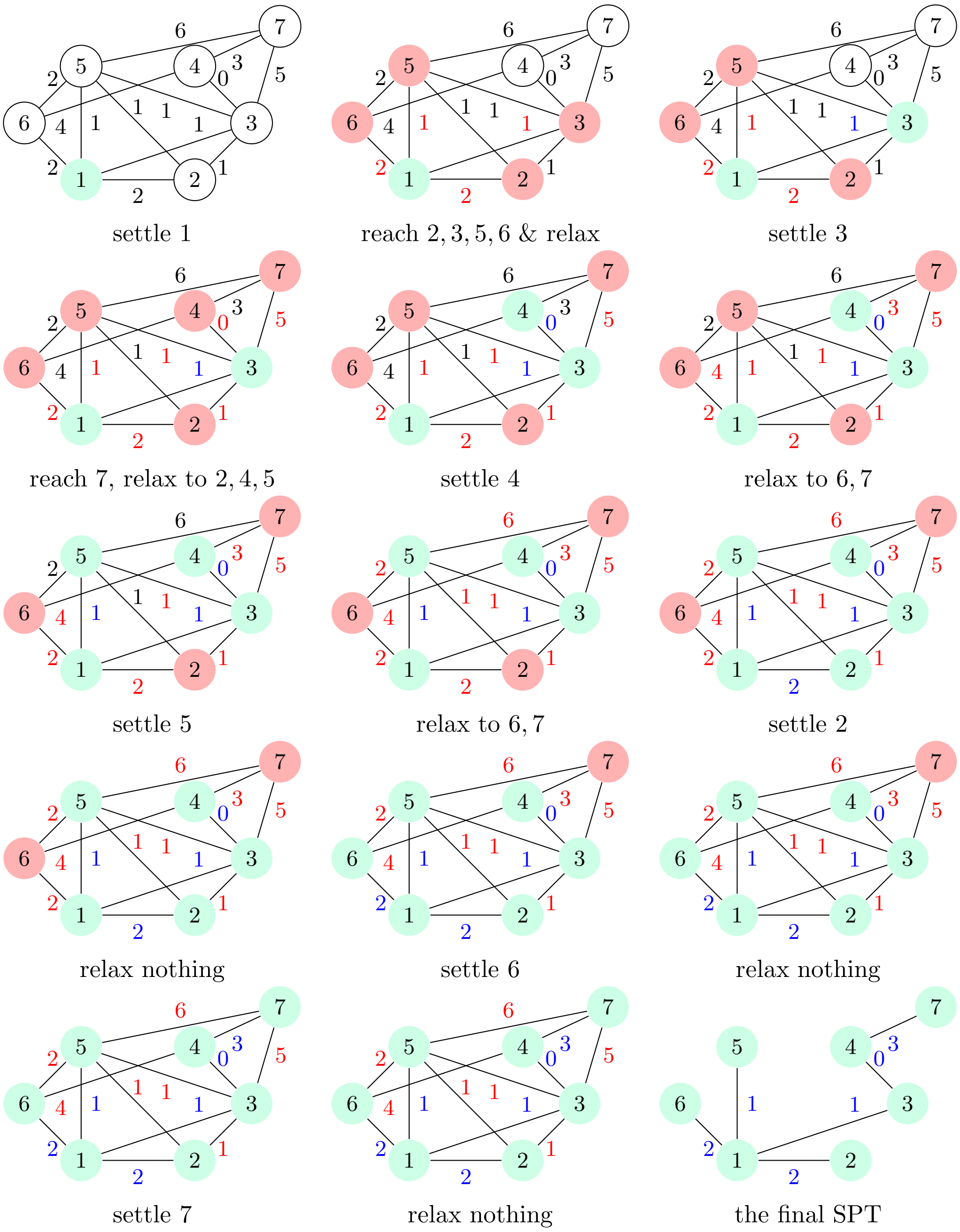
It should be evident that a node must be reached before it can be settled.

12.4.8 Proposition
All nodes are settled by Dijkstra's algorithm.

Proof. By Thm. 8.1.1, and because Dijkstra's algorithm is essentially like GRAPH SCANNING with one fewer check in the inner loop, all nodes are reached. Since the outer loop continues until all nodes are settled, the algorithm terminates with this condition holding. □

12.4.9 Theorem
At any iteration of Dijkstra's algorithm and for each $v \in V$, d_v *is the weight of a shortest path from* s *to* v *where all predecessors of* v *in the path are settled.*

Proof. By induction on the iteration index k. Let S be the set of settled nodes at iteration $k-1$, let u be chosen at Step 2 of iteration k, and P^* be the path from s to v determined by Dijkstra's algorithm. Suppose there is another path P from s to v with weight $c(P)$. Since $v \notin S$, there must be $(w,z) \in A$ with $w \in S$ and $z \notin S$ s.t. $P = P_1 \cup \{(w,z)\} \cup P_2$, where $V(P_1) \subseteq S$. Then $c(P) =$


settle 1 reach $2, 3, 5, 6$ & relax settle 3

reach 7, relax to $2, 4, 5$ settle 4 relax to $6, 7$

settle 5 relax to $6, 7$ settle 2

relax nothing settle 6 relax nothing

settle 7 relax nothing the final SPT

Figure 12.2: Dijkstra's algorithm running over a small graph (see Example 12.4.6).

$c(P_1) + c_{wz} + c(P_2) \geq c(P_1) + c_{wz}$ because we subtracted $c(P_2)$, $c(P_1) + c_{wz} = d_w + c_{wz}$ by the induction hypothesis, $d_w + c_{wz} = d_z \geq d_v$ because otherwise d_v would not be minimum, contradicting the choice of v at Step 2, and finally $d_z \geq d_v = c(P^*)$, so that P^* is a shortest path from s to v, as claimed. $\square$

12.4.4 A more refined implementation

In the complexity analysis of the simple implementation of Dijkstra's algorithm, we did not exploit the fact that V is implemented as a priority queu (see Sect. 11.4.1). Moreover, because only reached nodes can be settled, V need only contain unsettled, reached nodes at any time. Because Step 2 requires the node v with minimum d_v, we use d_v as priorities. This, however, raises an issue: since d_v is updated every time an arc is relaxed, priorities might need to be changed for nodes that are already in the queue. One way to implement the update of the priority d_v for the node v in a heap (the tree structure that implements a priority queue, see Sect. 11.4) is to delete v ($O(\log n)$) and re-insert it with the updated priority ($O(\log n)$).

12.4.4.1 Pseudocode

In the pseudocode, we distinguish between V, the set of nodes, and Q, the priority queue of reached unsettled nodes.

```
 1: $\forall v \in V\ d_v = \infty$, $d_s = 0$;
 2: $\forall v \in V\ \mathtt{p}_v = s$;
 3: $Q$.insert($s, d_s$);
 4: while $Q \neq \varnothing$ do
 5:   Let $u = Q$.popMin();
 6:   for $(u,v) \in \delta^+(u)$ do
 7:     Let $\Delta = d_u + c_{uv}$;
 8:     if $\Delta < d_v$ then
 9:       Let $d_v = \Delta$;
10:       Let $\mathtt{p}_v = u$;
11:       $Q$.delete($v$); // no effect if $v \notin Q$
12:       $Q$.insert($v, d_v$);
13:     end if
14:   end for
15: end while
```

12.4.4.2 Complexity

By Exercise 12.4.5, nodes are settled exactly once. This implies that Step 5 is executed $O(n)$ times, each taking $O(\log n)$; and also that each arc is relaxed exactly once, since its head vertex must be settled in order for an arc to be relaxed inside the loop at Step 6; since priority update takes $O(\log n)$, as remarked above, this implementation is $O((n+m)\log n)$ overall. This is worse than $O(n^2)$ if the digraph is dense but better if it is sufficiently sparse, which is most usually the case.

There exists also an $O(m + n\log n)$ Dijkstra algorithm implementation using more refined data structures.

12.4.5 The point-to-point SPP

Solving the point-to-point SPP from s to t can be seen as a "part" of the SPP: it suffices to stop the search as soon as t is settled, since, by definition, a settled node will keep its priority until the end of the algorithm.

Accordingly, we can use a Dijkstra's algorithm modified by inserting the code below between Step 5 and 6:

```
if $u = t$ then
```

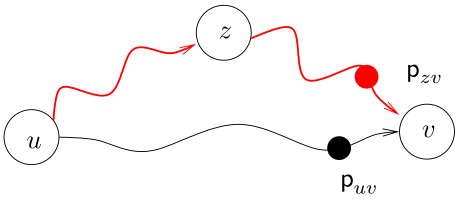
```
    exit;
  end if
```

12.5 Floyd-Warshall algorithm

The Floyd-Warshall algorithm finds all shortest paths (i.e. it solves the ASP) on a digraph $G = (V, A)$ with any arc weight function $c : A \to \mathbb{R}$.

12.5.1 Data structures

As data structures, we use two $n \times n$ matrices p, d: p_{uv} is the predecessor of v in the shortest path from u; d_{uv} is the cost of the shortest path from u to v. The principle of the algorithm is the following: for each node z and node pair u, v, we check whether the current shortest path from u to v can be improved by passing through z.



If so, we update d_{uv} to $d_{uz} + d_{zv}$ and p_{uv} to p_{zv}, otherwise we consider the next triplet z, u, v.

12.5.2 Pseudocode

I find this algorithm is the very simplest to remember!

```
$\forall u, v \in V\ d_{uv} = \begin{cases} c_{uv} & \text{if } (u,v) \in A \\ \infty & \text{otherwise} \end{cases}$
$\forall u, v \in V\ \mathsf{p}_{uv} = u$
for $z \in V$ do
  for $u \in V$ do
    for $v \in V$ do
      $\Delta = d_{uz} + d_{zv}$;
      if $\Delta < d_{uv}$ then
        $d_{uv} = \Delta$;
        $\mathsf{p}_{uv} = \mathsf{p}_{zv}$;
      end if
    end for
  end for
end for
```

The complexity of the Floyd-Warshall algorithm is clearly $O(n^3)$.

12.5.1 Exercise
Prove that the algorithm is correct. Try to express the following concept formally: at termination, every possible triplet was checked.

12.5.3 Negative cycles

It is interesting to remark that the Floyd-Warshall algorithm can also solve the NCP. Assume there is a negative cycle through u; when $u = v$, the “triangulations” through other nodes z will eventually yield $d_{uu} < 0$, since a path from u to itself via the negative cycle will have negative weight. Whenever that happens, terminate the algorithm: a negative cycle was found.

The modification to the pseudocode above is trivial: after Step 6, insert this code:

```
if $\Delta < 0$ then
  exit;
end if
```

Bibliography

[1] Ph. Baptiste and L. Maranget. *Programmation et Algorithmique (in French).* Polycopié de l'Ecole Polytechnique, Palaiseau, 2009.

[2] C. Berge. *Théorie des graphes et ses applications.* Dunod, Paris, 1958.

[3] E. Berlekamp, J. Conway, and R. Guy. *Winning ways for your mathematical plays, vol. 2.* Academic Press, 1982.

[4] N. Biggs, E. Lloyd, and R. Wilson. *Graph Theory 1736-1936.* Oxford University Press, Oxford, 1976.

[5] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[6] P. Cameron. *Combinatorics: Topics, Techniques, Algorithms.* Cambridge University Press, Cambridge, 1994.

[7] N. Chomsky. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, MA, 1965.

[8] A. Clark. *Elements of Abstract Algebra.* Dover, New York, 1984.

[9] G. Dowek. *Les principes des langages de programmation (in French).* Editions de l'Ecole Polytechnique, Palaiseau, 2008.

[10] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of* **NP**-*Completeness.* Freeman and Company, New York, 1979.

[11] J. Jones. Universal diophantine equation. *Journal of Symbolic Logic*, 47(3):549–571, 1982.

[12] D.E. Knuth. *The Art of Computer Programming, Part I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 2nd edition, 1997.

[13] D. König. *Theory of Finite and Infinite Graphs.* Birkhäuser, Boston, 1990.

[14] K. Kunen. *Set Theory. An Introduction to Independence Proofs.* North Holland, Amsterdam, 1980.

[15] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures.* Springer, Berlin, 2008.

[16] M. Minsky. Size and structure of universal turing machines using tag systems. In *Recursive Function Theory*, volume 5 of *Symposia in Pure Mathematics*, pages 229–238. AMS, Providence, 1962.

[17] M. Minsky. *Computation: Finite and infinite machines.* Prentice-Hall, London, 1967.

[18] R. Montague. *Formal Philosophy.* Yale University Press, London, 1974.

[19] Y. Roghozin. Small universal Turing machines. *Theoretical Computer Science*, 168:215–240, 1996.

[20] S. Seshu and M.B. Reed. *Linear Graphs and Electrical Networks.* Addison-Wesley, Reading, MA, 1961.

[21] C. Shannon. A universal Turing machine with two internal states. In C. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 157–165, Princeton, 1956. Princeton University Press.

[22] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[23] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937.

[24] V. Yngve. The depth hypothesis. In *Structure of Language and its Mathematical Aspects*, volume XII of *Proceedings of Symposia in Applied Mathematics.* American Mathematical Society, Providence, 1961.

Index

$O(\cdot)$, 23, 138
$[n]$, 105
$\Omega(\cdot)$, 138
$\Theta(\cdot)$, 138
$\perp$, 61, 63
k-CLIQUE, 133, 134
k-STABLE, 133, 134, 138
;, 32
`<>`, 72
{}, 31

abstraction, 133
action, 105
address, 30, 103
    return, 71
Adelson, 154
algorihm, 109
algorithm, 22, 29, 102, 132, 133
    Bellman-Ford, 164
    best, 138
    Dijkstra, 164, 166, 167, 169
    efficient, 87
    exact, 135
    execution, 146
    exponential, 138
    Floyd-Warshall, 164, 170
    heuristic, 135
    input, 56
    non-polynomial, 135
    polynomial, 135, 138
    Prim, 128
    Prim's, 133
    recursive, 124, 141, 160
    sorting, 144, 147
        best, 140
        comparison-based, 139
        general-purpose, 140
    sub-step, 163
alphabet, 18, 20, 104
ambiguous, 94
AMPL, 19
ancestor, 124
    node, 149
approximation, 140
arc, 41, 59, 122
    multiple, 41
    multiplicity, 41
    relaxed, 169
`args`, 30
Ariane 5, 28
array, 30, 32, 33, 36, 37, 53, 75, 77, 109, 137
    access, 67
    binary, 118
    bound, 71
    circular, 65
        implementation, 66
    complexity, 54
    component, 58
    delimiter, 55
    efficiency, 54
    entry, 55
    insert, 54, 75
    insertion, 56
    integer, 121, 166
    jagged, 53
    linear, 32, 36, 53
    memory, 71
    memory size, 55
    move, 54
    multidimensional, 32, 53
    operation
        copy, 58
    operations, 54
    read, 54
    removal, 56
    remove, 54
    scanning, 140
    size, 54, 55
    sorted, 137
    splice, 55
    two-dimensional, 32, 36
    unsorted, 137, 140
    write, 54
`ArrayList`, 29, 110

ASCII, 25, 71, 77, 79
ASP, 164, 170
assembly, 71
`assert`, 67
assignment, 22, 75, 104
atom, 45, 92
atomic formula, 92
attribute, 21, 27, 35, 81
automorphism, 43, 105
  graph, 43
AVL, 154
axiom, 114

backtrack, 124, 125
base address, 75
base case, 150
Basic, 19, 25
BB, 164
Bellman, R., 164
BFS, 67, 120, 123, 164
  rank, 120
bijection, 43, 106
bipartite, 41, 42
bisection, 132, 137, 141, 142
bit, 20, 118
block
  for, 31
  if, 31
  while, 31
bond, 92
`boolean`, 31
`bot`, 63
bound, 146
brace, 31, 34
bracket
  curly, 31
  square, 32
BRANCH-AND-BOUND, 164
breadth-first search, 67
BREADTH-FIRST SEARCH, 120, 164
brute force, 137
BST, 150, 154, 158–160
  balancing, 154
  delete, 154, 158
  deletion, 152
    two subnodes, 153
  `find`, 151
  insert, 157
  `insert`, 152
  left, 156
  `max`, 151
  `min`, 150
  minimum, 153
  property, 156
  rebalancing, 156
  recursive
    methods, 150
  right, 156
  root, 154
  rooted, 156, 157
  rotating, 156
bug, 28
byte
  comparison, 139

C, 19, 25, 31, 53
C++, 19, 25, 30, 31, 53
call
  recursive, 108
Cameron, P., 144
CAML, 19
carbon, 92
cardinality
  largest, 136
Carnap, R., 95
Cartesian product, 41
Cayley, 90
cell, 18
certificate, 131
  NO, 133
  YES, 133
`char`, 26, 31
chemical formula, 93
Chomsky, N., 95
chord, 92
Church Hypothesis, 18
Church, A., 18
circuit, 48, 163
  weighted, 165
circular array, 65
`circularArray`, 67
class, 21, 27, 28, 33, 53, 80
  constructor, 63
  function, 35
  implementation, 62
  infinite, 138
  inheritance, 28
  interface, 28
  member, 27, 32, 82
  parameter, 72
  parametrizable, 110
  parametrized, 72
  reference, 110
  structure, 28, 34
  user-defined, 26
clique, 42, 45, 131, 133
  directed, 41
co-domain, 19

code
  duplication, 28
  executable, 32
collision, 78
column, 33, 36
command line, 70
comment, 26, 33, 62, 109, 113
commodity, 97
comparison, 139, 144, 146, 147
  byte, 77
compiler, 25, 27, 28, 30, 31, 53
complete, 41, 45, 133
complete graph, 42
complexity
  algorithm, 133, 138
  array, 54
  asymptotic, 57, 138, 147
  average case, 57, 60
  average-case, 22, 56, 144
  best-case, 22, 138
  exact, 147
  lower bound, 133
  order, 23
  polynomial, 138
  problem, 133, 138
  space, 22
  time, 22
  upper bound, 132
  worst-case, 22, 23, 132, 138, 147
component, 33
components, 36
composition, 106
computable, 19
computer science, 163
concatenate, 61
concatenation, 18, 22
conformance, 28
conjecture, 138
connected, 46
  component, 118
connectedness
  minimal, 127
connectivity, 97
conservative, 165, 166
console, 33, 35–37
constant, 20, 23, 94, 139
constant time, 75
constructor, 63, 82
contraction, 50
contradiction, 108, 119
control, 71
Cook, S., 134
copy
  deep, 26
  shallow, 26
cost, 97
  function, 128
  minimum, 127, 132
  nonnegative, 97
  unitary, 97
couple, 164
CPU, 17, 25
cutset, 46, 60, 127, 128
  empty
    nontrivial, 131
  trivial, 46
cycle, 60, 65, 87, 89, 119, 127, 163, 165
  basis, 92
    fundamental, 92
  disjoint, 107
  length, 107
  negative, 163–165
  set, 92
    linearly independent, 92
  simple, 47, 59
  space, 48, 87
    basis, 92
  sum, 92
  unique, 90, 92
cylinder, 112
  radius, 112

DAG, 125
data
  container, 53, 149
    dynamic, 149
    size, 58
    static, 149
  object, 131
  shared, 27, 35
  structure, 20, 21, 27, 41, 53, 71, 110, 119, 128, 149
    element, 53
    elementary, 21
    entry, 53
    fixed size, 76
    linear, 58, 87, 110, 137, 149
    searchable, 149
    tree, 87, 149
    tree-like, 159
    user-defined, 21
  type, 26, 27, 31, 60, 131, 138, 149
    elementary, 26, 30, 31
    non-elementary, 71, 72
data container, 58
decidable, 19, 21
decision, 19, 131
declaration, 34

declarative, 19
definition, 34
degree, 46, 89
delete, 149, 155
delimiter, 55
depth, 88, 124, 156
  difference, 156
depth-first, 102
DEPTH-FIRST SEARCH, 123
DFS, 102, 123, 125
  scan, 126
  search, 124
`dfs`, 124
digraph, 41, 54, 118, 163, 165, 166, 170
  acyclic, 125
  arc list, 54
  bipartite, 41
  complete, 41
  dense, 169
  empty, 41
  nonnegatively weighted, 166
  simple, 41
  star, 166
  strongly connected, 47, 163
Dijkstra, E., 164
dimension, 33, 87
Diophantine equation, 19
disconnected, 87
distance
  Hamming, 97
DIVIDE-AND-CONQUER, 141, 142
`DLList`, 61, 62
domain, 19, 76
dot operator, 31
`double`, 21, 26, 31

Eco, U., 125
edge, 42, 59, 89, 92, 97, 118–120, 124
  addition, 50, 60
  cheapest, 128
  column, 92
  contraction, 50
  cost function, 127
  intersection, 125
  number, 123
  removal, 50, 60
  set, 42, 45, 133
  swap, 90
  tree, 92
  weighted, 163
Edmonds, J., 135
efficiency
  algorithmic, 133
efficient, 20
element
  leftmost, 140
  minimum, 140, 150
empty, 45, 132
engineering, 163
equation, 144
  differential, 144
    first-order, 146
error
  detection, 67
event
  independence, 57
  independent, 57
expression, 94
  mathematical, 94
  tree, 95

factor, 145
factorial
  iterative, 104
  recursive, 101
Fermat, 114
Fibonacci (L. Pisano), 144
field
  finite, 92
FIFO, 65
file, 32
find, 149, 155
flip, 97
`float`, 26, 31
`floatDown`, 161, 162
floating-point, 30
`floatUp`, 161, 162
floor, 36
Floyd, 164
Ford, L., 164
formula
  propositional, 134
  Stirling, 140
Fortran, 19, 25
function, 19, 20, 23, 71
  arc
    weigth, 170
  arc weight, 165
  argument, 29
  call, 71, 104
    nested, 71
    recursive, 101
    tree, 103
  computable, 19
  edge weight, 163
  input, 71
  overloading, 77
  plot, 33

- recursive, 124, 150
- scalar-valued, 132
- user-defined, 21
- weight
  - conservative, 165
- well-defined, 71

`functionPlot`, 33

Gödel
- incompleteness theorem, 114

Gödel, K., 114
Game of Life, 18
Garey, M., 133
generic container, 72
GOTO, 71
grammar
- formal, 94
- label, 95

graph, 41, 42, 54, 59, 102, 120, 124, 132, 163
- adjacency list, 54
- automorphism, 43
- bipartite, 42
- block, 125
- complement, 42, 133
- complete, 42, 97
- connected, 46, 87, 120, 124, 131, 163
- dense, 167
- directed, 41, 88, 163
- disconnected, 132
- edge list, 54
- empty, 42
- exploration, 67
- isomorphism, 43
- line, 44
- minor, 50
- operation, 50, 60
- planar, 125
- problem, 117
- representation, 54
- simple, 42
- sparse, 167
- strongly connected, 46
- structure, 43
- theory, 125
- underlying, 42
- undirected, 163, 164
- weighted, 128

GRAPH SCANNING, 117, 119, 120, 123, 164, 166, 167
greedy, 135
group, 43, 106
- structure, 44
- symmetric, 106

GUI, 28
Hamming, 97
Hanoi towers, 112
- Java, 113

hash
- table, 80

hash function, 76, 78, 79, 83
- family, 79
- injective, 77, 78

hash table, 75, 78, 79
- find, 83

`hashCode`, 77
head, 18
heap, 159, 160, 169
- bottom, 160
- empty, 161
- float, 161
- `floatUp`, 160
- insertion, 160
- last, 160
- level, 160, 161
- new, 161
- node, 160
- `popMax`, 161
- property, 159–161
- root, 161
- shape properties, 160
- shape property, 159, 160
- unlink, 161

height, 88
`helloWorld`, 27
heuristic, 135, 164
- greedy, 135

Hilbert, D., 19, 114
hydrogen, 92

identity, 44
imperative, 19
implementation
- recursive, 149

import, 34, 62, 109, 113
indegree, 42
independent
- set, 45

index, 53, 76, 79, 147
- arithmetic, 53, 61
- iteration, 167
- positional, 144

induced, 44
induction
- mathematical, 101, 104

infinite, 131
information
- exchange, 97

inheritance, 28

injective, 80
input, 71, 102, 131–133, 137, 144, 152
  argument, 104, 105
  general, 144
insert, 149, 155
  array, 54
insertion, 161
INSERTION SORT, 140
instance, 102, 131, 132, 144
  NO, 132
  size, 133
  YES, 132
instar, 42
instruction, 17, 25, 27, 71
  basic, 32
`int`, 20, 26, 31, 110
`Integer`, 72, 110
integer, 30, 33, 58, 133, 144
integer rounding, 34, 36
integration, 146
interface, 28, 67
  implementation, 29
interpreter, 25
inverse, 156
IO, 17
isomer, 92
iteration, 119, 167
  successive, 91
`Iterator`, 110, 111
  `.hasNext()`, 110
  `.next()`, 110
itinerary, 122, 123

Java, 19, 25, 27, 53, 109
  compiler, 30
  exception, 151, 152
  function, 29
  programming, 77
`java`, 26
`javac`, 26
`jmp`, 71
Johnson, D., 133
junction, 125

key, 75, 76, 78–80

label, 88
Landis, 154
language, 18, 93
  assembly, 71
  declarative, 19
  formal, 26, 93
    semantics, 95
    syntax, 94
  imperative, 19
  machine, 139
  meaning, 93
  natural, 26, 93, 125
    semantics, 96
    syntax, 95
  programming, 25, 93
  universal, 19
leaf, 88, 94, 95, 124, 126, 139, 149, 154
  deepest, 127
  rightmost, 127
left, 160
level, 88, 160
library, 36, 62
  standard, 34, 69
LIFO, 67
line graph, 44
linear
  independence, 92
  time, 164
link, 97
`LinkedList`, 79, 80, 110
Linux, 25
LISP, 19
list, 58, 75, 77
  access, 61
  adjacency, 54, 102, 118
  arc, 54
  concatenate, 61
  doubly-linked, 58
  edge, 54
  find, 64
  find next, 64
  first, 63
  implementation, 61
  insertion, 59
  Java, 61
  last, 63
  linked, 109, 110, 118, 149
  next, 58, 61
  operation, 59
  placeholder, 59
  pop, 61, 63
  previous, 58, 61
  print, 64
  push, 61, 63
  read, 61
  removal, 60
  singly-linked, 58, 61, 78, 83
    find, 83
    implementation, 80
  size, 63
  write, 61
logistics, 163
`long`, 26, 31

Longest Path Problem, 164
loop, 18, 20, 22, 31, 36, 41, 56, 80, 102, 139
  incomplete
    complexity, 57
  infinite, 104, 105
  inner, 167
  internal, 167
  outer, 167
  simple, 56
  termination, 56
LPP, 164
LSP, 164

MacOSX, 25
`main`, 27, 30–33, 61, 78
`Map`, 72
map, 71, 77
`Math`, 36
matrix, 32, 53
Matyasevič, 19
max, 149, 155, 159, 161
Max Clique, 134
Max Stable, 135
maximization, 132
maximum, 37
maze, 125
median, 132, 144
member, 31
memory, 18, 20, 30, 104
  access, 53
  address, 26, 27, 31, 58, 71, 75
  allocation, 32, 63
    automatic, 61
    manual, 61
  block, 53
  cell, 58, 65
  object, 62
  size, 77
  swap, 55
merge, 141
  efficiently, 141
MergeSort, 141, 142, 144
method, 21, 27
  depth-dependent, 159
  static, 109
min, 149, 155
minimization, 132
minimum, 37, 140
Minimum Spanning Tree, 132
minor, 50
Minsky, 18
model of computation, 18
molecule, 45, 92
Montague, R., 96
move
  array, 54
MST, 128, 132
multidigraph, 41
multigraph, 42

NCP, 164, 171
neighbour, 124
neighbourhood, 43
  outgoing, 42
networj
  event, 122
network, 97
  communication, 163
  distance, 97
  event, 123
  power, 163
`new`, 61
node, 41, 59, 123, 139, 144, 149
  child, 149
  degree, 88
  depth, 160
  extraction, 159
  labelled, 122
  leaf, 139, 150, 152
  leftmost, 153
  level, 162
  non-leaf, 149, 150
  non-root, 149
  order, 156
  pair, 170
  parent, 149, 166
  reached, 166, 169
  root, 159
  scanning, 117
  settled, 166, 167
  source, 166
  unsettled, 169
NP, 133, 134
NP-complete, 134
NP-hard, 134, 135
`null`, 58, 65, 81, 150

object, 21, 27, 28, 31, 32, 53, 62, 110
  address, 65
  equal, 77
  interface, 29
  member, 31
  type, 131
operand, 95
operating system, 32, 55, 71
operator, 95
  k-ary, 106
  binary, 94
  closed, 106

  dot, 31
  precedence, 94
  unary, 94
order, 47, 105
  alphabetical, 75
  linear, 160
  natural, 137, 161
  rank, 75
  relation, 150
  topological, 125
  total, 138
ordered
  totally, 150
orders, 109
oriented, 165
OS, 32, 71, 104
outdegree, 42
output, 131, 133
  algorithmic, 146
outstar, 42
overloading, 81

P, 133
P2PSPP, 164, 169
pair, 71, 122
paradox, 147
paraffin, 93
parent, 160
parsing, 95, 125
partition, 142
Pascal, 19, 25
passing
  by reference, 30, 110
  by value, 30, 110
path, 47, 88, 119, 120, 123–125, 139, 149, 163
  all, 164
  directed, 59, 123
  edge, 163
  endpoint, 47
  length, 118, 120
  longest, 88, 154, 164
  not unique, 165
  shortest, 123, 163, 165, 170
    weight, 166, 167
  simple, 46, 47, 59, 164
  unique, 92, 120
Peano
  axioms, 114
Peano, G., 114
perl, 25
permutation, 105, 108, 138, 140
  action, 105
  cycle, 107, 139
  identity, 139
  inverse, 106
  list, 108
  product, 106
pivot, 142–144
placeholder, 59, 63
point of entry, 32, 35, 61, 110
POINT-TO-POINT SPP, 164
pointer, 110
polynomial, 19, 133
polynomial time, 134
pop, 61
`popFront()`, 120
`popFront`, 159
`popMax`, 159
position
  correct, 140
power, 97
  series
    log, 146
Prüfer, 90
predecessor, 167, 170
preprocessor, 27
PRIM'S ALGORITHM, 128
Prim, R., 128
prime, 79
print, 34, 36
priority, 159, 169
  higher, 160, 161
  highest, 159, 161
priority queue, 67
private, 27, 35
probability, 57
probability distribution, 57
problem, 21, 22
  case, 131
  class, 133
  complex, 141
  complexity, 133
  decidable, 21, 22
  decision, 19, 131, 132, 137
    abstract, 132
  instance, 102, 131
  minimization, 132
  MST, 132
    complexity, 133
  NP-complete, 138
  NP-hard, 135
  optimization, 132, 135
    abstract, 132
  size, 141
  undecidable, 132
procedure
  recursive, 101
product, 145, 146

    associative, 106
program, 17, 18
    execution, 58
programming language, 18
Prolog, 19
protein, 45
pseudocode, 102, 128
    recursive, 137
public, 27, 30, 32, 35
`public`, 63
push, 61
`pushBack()`, 120
python, 25

query
    search, 149
queue, 65, 67, 119, 120, 123, 159
    applications, 67
    beginning, 65
    delete, 65
    empty, 65
    end, 65
    insert, 65
    insertion, 65
    placeholder, 66
    priority, 67, 159, 164, 166, 169
    removal, 65
    retrieve, 65
    size, 65
QUICKSORT, 142, 144

RAM, 17, 18, 20
random
    event, 57
random access, 54
read
    array, 54
record, 75, 76, 78, 80
recursion, 94, 124, 125, 141, 142
    level, 105, 144
recursive, 19
recursively enumerable, 19
reduction
    polynomial, 134
reference, 26, 58, 61–63, 110
    next, 110
    `null`, 150
    passing, 110
    passing by, 30
relation, 41, 150
    recurrence, 144
remove
    array, 54
replace, 152
resource
    limited, 135
return
    type, 63
    value, 104
`return`, 71
return value, 30
Roghozin, 18
root, 88, 125, 139, 149, 154
    path to, 160
    priority, 161
    shell, 71
rotation
    left, 156, 157
    right, 156–158
rotationa
    left, 158
route, 97
row, 33, 36, 54
rule
    syntactical, 93

scalar, 159
scanning, 117
scope, 31
search
    depth-first, 102
    query, 149
searching, 87, 137, 149
SEARCHING PROBLEM, 137
see stack, pop, 61
see stack, push, 61
SELECTION SORT, 140
semantics, 95, 96
semicolon, 32
sentence
    ambiguous, 93
    meaning, 95
    structure, 125
    valid, 93
sequence, 58, 79, 105, 125, 137–140, 142
    binary, 147
    elements, 144
    empty, 91
    Fibonacci, 144
    Prüfer, 90, 91
    sorted, 137, 141, 143, 150
    unsorted, 137, 142, 144
series
    power, 144
        formal, 145
set, 27, 71, 95
    closed, 106
    decidable, 19
    input, 139

  - property, 27
  - recursive, 19
  - scalar, 132
  - stable, 133
  - totally ordered, 140
- settled, 166, 167, 169
- shadowing, 36
- Shannon, 18
- shape
  - property, 161
- shell, 32
- SHORTEST PATH PROBLEM, 47, 163
- SHORTEST SIMPLE PATH, 164
- side-chain, 45
- simulation, 18, 21
- sine, 34
- `singlyLinkedList`, 83
- size
  - array, 54
- slot
  - free
    - leftmost, 140
- solution, 141
  - closed-form, 144
  - optimal, 135
  - partial, 141
- sort
  - insertion, 140
  - merge, 140, 141
  - quick, 140
  - recursively, 143
  - selection, 140
- sorted, 137, 149
- sorting, 87, 149
  - tree, 139
- SORTING PROBLEM, 138, 147
- source, 124
- SP, 138
- spanning, 127
- sparse, 169
- specifier, 30
- SPP, 47, 163, 164, 166
  - instance, 167
  - negative weights, 164
- SPT, 165, 166
  - directed, 165
- SSP, 164
- stable, 45, 133
  - maximal, 135
  - maximum, 135
  - set, 45, 133
- stack, 61, 67, 71, 103, 104, 119, 123, 124
  - implementation, 68
  - pop, 67, 103, 123
  - push, 67, 103, 123
  - size, 105, 126
  - smashing, 71
- star, 43, 54
  - incoming, 42, 54
  - neighbourhood, 42
  - outgoing, 42, 54
- state, 17
- static, 27, 31–33, 82
  - method, 78
- `static`, 61
- Stirling, 140
- storage, 121
- `stringPair`, 83
- strongly connected, 47
- structure
  - combinatorial, 146
- subgraph, 44, 97
  - k-clique, 134
  - complete, 134
  - contraction, 50
  - densest, 45
  - induced, 44, 45
  - isomorphism, 45
  - minimally connected, 97
  - spanning, 88
- subnode, 149
  - left, 149
  - right, 149
- subpath
  - initial, 165
- subproblem, 131, 141
- subsequence, 55, 58, 142
  - sorted, 141
- subset, 132
  - proper, 127
- subtree, 156, 157
  - left, 144, 149, 150, 153, 154, 157
  - right, 144, 153, 154, 158
  - rooted, 150
- successor, 160
- sum, 145
- swap, 55, 107
- symbol, 18, 20
- syntax, 95
- `System`, 31
- `System.out`, 31, 34

- tape, 18, 21
- Tarjan, R., 125
- Tarski, A., 96
- template, 72
- tensor, 32, 53
- termination, 22

    failure, 19, 132
test, 18, 20, 22, 139
    termination, 105
theory, 114
    complete, 114
    consistent, 114
    inconsistent, 115
`this`, 27, 62
time
    limited, 135
TM, 18, 134
tour, 48
trace, 146
trade-off, 18, 22
trail, 47, 48
transportation
    mode, 123
tree, 87, 89, 101, 119, 120, 124, 125, 127, 146, 149, 157
    k-ary, 149
    AVL, 154, 160
    balanced, 144, 159
    balancing, 154
    BFS, 120
    binary, 139, 149, 159, 161
        balanced, 160
    connected, 90
    data structure, 87
    depth, 88, 126, 139
    derivation, 95, 125
    directed, 88, 149
    edge
        removal, 127
    expression, 95
    function call, 103
    height, 88
    labelled, 89, 90
    leaf, 88
    level, 88, 159, 162
        filled, 160
    listing, 93
    MERGESORT, 144
    node, 87, 146
    parse, 95
    progressive, 126
    QUICKSORT, 144
    recursion, 144
        balanced, 141
    regressive, 126
    root, 151
    rooted, 88
        directed, 88
    rotation, 155
    search
        binary, 150
    shape, 156–158
        symmetric, 157
    shortest path, 120, 164, 165
    sorting, 139, 140, 144, 147
    spanning, 88, 90, 92, 97, 119, 120, 127, 131, 165
        minimum cost, 97, 128
    unbalanced, 144, 157
    unlabelled, 89
Turing
    Machine, 134
    machine, 18
        universal, 18, 104
Turing, A., 18
Turing-complete, 18
Turing-equivalent, 18
type
    elementary, 110
    parameter, 110
    parametrized, 72

unbalance, 156, 157
unbounded
    path weight, 164
undecidable, 19
universal, 20
    language, 18
    Turing machine, 18
universality, 104
unlink, 152, 161
unsorted, 137
user, 35, 36, 38
USPP, 164
UTM, 18, 19, 104

valence, 92
value, 132, 144
    maximum, 132
    minimum, 132
    passing, 110
    passing by, 30
variable, 19, 94
    address, 26
    name, 27
    static, 36
    symbol, 26, 94, 95
    temporary, 55
    value, 26
        default, 63
vector, 32, 33, 53, 57
    binary, 49
    incidence, 49
        cycle-edge, 49, 92
    space, 92

Velksii, 154
vertex, 41, 59, 65, 89, 119, 133, 149, 165
    addition, 50, 60
    adjacent, 167
    BFS rank, 120
    cut, 125
    degree, 89
    disconnected, 89
    distinct, 120
    incident, 127
    label, 43, 88, 91
    leaf, 91
    level, 88
    non-root, 88
    order, 91
    pair, 42
    parent, 91, 165
    rank, 120
    removal, 50, 60, 91
    root, 88, 126, 127
    scanning, 117
    sequence, 126
    set, 43, 45
    source, 118, 124, 128, 165
virtual machine, 25, 27, 35
void, 32
`void`, 30

walk, 47, 163, 165
    closed, 48
Warshall, 164
weight
    negative, 164
    sum, 163
    unit, 163
well-defined, 71
Wiles, A., 114
Windows, 25
word, 125
write
    array, 54

XOR, 49, 92