Supplementary lecture notes

Algorithms and Data Structures

Gerth Stølting Brodal

November 23, 2020

Mathematics Cheat Sheet

$\mathbb{N} = \{1, 2, 3, \ldots\}$ natural numbers
$\mathbb{R}$ = real numbers
$\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$
Floor $\lfloor x \rfloor$, *ceiling* $\lceil x \rceil$

Associativity
$(a \cdot b) \cdot c = a \cdot (b \cdot c), \quad (a+b)+c = a+(b+c)$

Commutativity
$a \cdot b = b \cdot a, \quad a + b = b + a$

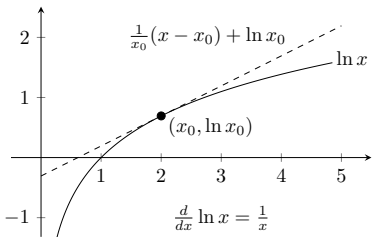
Distributivity
$a \cdot (b + c) = a \cdot b + a \cdot c$

Powers
$a^0 = 1, \quad a^1 = a, \quad a^{b+c} = a^b \cdot a^c$
$(a^b)^c = a^{b \cdot c}, \quad a^{b^c} = a^{(b^c)}, \quad a^{-b} = \frac{1}{a^b}$
$(a \cdot b)^c = a^c \cdot b^c, \quad a^{b-c} = a^b / a^c$

Roots
$\sqrt{a} = \sqrt[2]{a} = a^{1/2}, \quad \sqrt[n]{a} = a^{1/n}$
$\sqrt[n]{a \cdot b} = \sqrt[n]{a} \cdot \sqrt[n]{b}, \quad \sqrt[n]{a/b} = \sqrt[n]{a} / \sqrt[n]{b}$

Fractions
$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$
$\frac{a}{b} / \frac{c}{d} = \frac{a \cdot d}{b \cdot c}$
$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}$

Logarithms
$\log_b a = c \Leftrightarrow b^c = a, \quad b^{\log_b a} = a$
Natural logarithm
$\ln a = \log_e a, \quad e = 2.71828\ldots$
Binary logarithm $\log_2 a = \lg a$
$\log_b 1 = 0, \quad \log_b b = 1$
$\log_b(a \cdot c) = \log_b a + \log_b c$
$\log_b(a/c) = \log_b a - \log_b c$
$\log_b(a^c) = c \cdot \log_b a$
$\log_b a = \frac{\log_c a}{\log_c b}, \quad \log_b^c a = (\log_b a)^c$
$a^{\log_b c} = 2^{\log_2 a \cdot \log_2 c \cdot \frac{1}{\log_2 b}} = c^{\log_b a}$

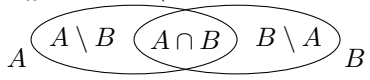


Matrices (1,4)

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \quad m \times n \text{ matrix}$$

Matrix addition ($m \times n$ og $m \times n$)
$C = A + B, \quad c_{ij} = a_{ij} + b_{ij}$
Matrix multiplication ($m \times n$ og $n \times p$)
$C = AB, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$
Constant multiplication ($m \times n$)
$C = cA, \quad c_{ij} = c \cdot a_{ij}$
$(A + B) + C = A + (B + C)$
$A + B = B + A$
$c(dA) = (cd)A$
$c(A + B) = (cA) + (cB)$
$(AB)C = A(BC)$
$A(B + C) = (AB) + (AC)$
$(A + B)C = (AC) + (BC)$

Sets
Set $A = \{a_1, a_2, \ldots, a_n\}$
Size $|A|$
Membership $x \in A$, *non-member* $x \notin A$
Empty set $\emptyset, \quad |\emptyset| = 0$
Subset $A \subseteq B$, i.e. $x \in A \Rightarrow x \in B$
Set intersection $A \cap B$
Set union $A \cup B$
Set difference $A \setminus B$ or $A - B$



$A \cup B = (A \setminus B) \cup (A \cap B) \cup (B \setminus A)$
Commutativity
$A \cap B = B \cap A, \quad A \cup B = B \cup A$
Associativity
$A \cup (B \cup C) = (A \cup B) \cup C$
$A \cap (B \cap C) = (A \cap B) \cap C$
Distributivity
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
DeMorgan's laws
$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$
Idempotence $A \cup A = A = A \cap A$
Empty set $A \cup \emptyset = A, \quad A \cap \emptyset = \emptyset$
Complement $\overline{A}$ wrt. *universe* U
$\overline{A} = U \setminus A$ where $A \subseteq U, \quad \overline{\overline{A}} = A$
$\overline{A \cup B} = \overline{A} \cap \overline{B}, \ \overline{A \cap B} = \overline{A} \cup \overline{B}$
A and B are *disjoint* $\Leftrightarrow A \cap B = \emptyset$
Cross product / Cartesian product
$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$

Sums (2)
$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$
$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$
$\sum_{i=1}^{n} (c \cdot a_i) = c \cdot \sum_{i=1}^{n} a_i$
$\sum_{i=0}^{n} 2^i = 1 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$
$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ for $a \neq 1$
$\sum_{i=0}^{\infty} a^i = \lim_{n \to \infty} \sum_{i=0}^{n} a^i = \frac{1}{1-a}, \ |a|<1$
$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
$\sum_{i=1}^{n} i \cdot a^i = \frac{a}{(1-a)^2}(1 - a^n - n \cdot a^n + n \cdot a^{n+1})$
$\sum_{i=1}^{\infty} i \cdot a^i = \frac{a}{(1-a)^2}$ for $|a| < 1$
Polynomial $P(x) = \sum_{i=0}^{k} c_i \cdot x^i$
Telescoping sum
$\sum_{i=0}^{n-1} (a_{i+1} - a_i) = a_n - a_0$
n-te *harmonic* number $H_n = \sum_{i=1}^{n} \frac{1}{i}$
$= \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$
$\ln n + \frac{1}{n} \leq H_n \leq \ln n + 1$
$\lim_{n \to \infty} (H_n - \ln n) = \gamma = 0.577215...$
$=$ Euler–Mascheroni constant

Products
$\prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdots x_n$
Factorial $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdots n$
$\ln\left(\prod_{i=1}^{n} x_i\right) = \sum_{i=1}^{n} \ln(x_i)$
$0 \leq \ln(n!) - (n \ln n - n + 1) \leq \ln n$

Probability theory (3)
Mean $\overline{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$
Binomial coefficient $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$
(number of ways to select k elements among n, indepedent on the order)
Linearity of expectation
$\mathrm{E}[\sum_{i=1}^{k} c_i \cdot X_i] = \sum_{i=1}^{k} c_i \cdot \mathrm{E}[X_i]$
Bernoulli distribution $X \sim Bern(p)$
$\Pr[X = 1] = p, \quad \Pr[X = 0] = 1 - p$
Binomial distribution $X \sim Bin(n, p)$
(sum of n Bernoulli trials)
$\Pr[X = k] = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}$
Expected value $\mu = \mathrm{E}[X] = p \cdot n$
Geometric distribution $X \sim Geom(p)$
(number of Bernoulli trials before 1)
$\Pr[X = k] = p \cdot (1 - p)^{k-1}$
$\mathrm{E}[X] = \sum_{k=1}^{\infty} k \cdot \Pr[X = k] = \frac{1}{p}$

Logical expressions
Or / disjunction $U \vee V$
And / conjunction $U \wedge V$
Not / negation $\neg U$
Implication / conditional $U \Rightarrow V$
Equivalent / biconditional $U \Leftrightarrow V$
Exists $\exists x : U(x)$
For all $\forall x : U(x)$

$\vee$	F	T
F	F	T
T	T	T

$\wedge$	F	T
F	F	F
T	F	T

	$\neg$
F	T
T	F

$\Rightarrow$	F	T
F	T	T
T	F	T

$\Leftrightarrow$	F	T
F	T	F
T	F	T

XOR	F	T
F	F	T
T	T	F

F = false, T = true

Asymptotic notation (1)
$f(x) = \mathrm{O}(g(x)) \Leftrightarrow$
$\exists c, x_0 \forall x \geq x_0 : f(x) \leq c \cdot g(x)$
$f(x) = \Omega(g(x)) \Leftrightarrow$
$\exists c > 0, x_0 \forall x \geq x_0 : f(x) \geq c \cdot g(x)$
$f(x) = \Theta(g(x)) \Leftrightarrow$
$f(x) = \mathrm{O}(g(x)) \wedge f(x) = \Omega(g(x))$

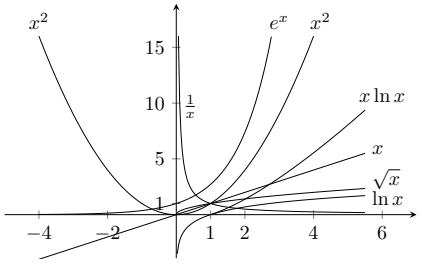
Master Theorem (1)
Constants $a, c, d > 0$, $p \geq 0$ and $b > 1$

$$T(n) = \begin{cases} c & \text{for } n \leq d \\ a \cdot T(n/b) + c \cdot n^p & \text{for } n > d \end{cases}$$

$\Downarrow$

$$T(n) = \begin{cases} \Theta(n^p) & \text{for } a < b^p \\ \Theta(n^p \cdot \log_b n) & \text{for } a = b^p \\ \Theta(n^{\log_b a}) & \text{for } a > b^p \end{cases}$$

Growth of functions



Literature
Mathematical formulas and terms (primary school), Ministry of Education, June 2017 [in Danish]
Mathematical formulas (high school, stx A), Undervisningsministeriet, May 2018 [in Danish]
Thomas H. Cormen *et al.*, Introduction to Algorithms, 3rd Edition, appendix A–C, 2009
Steve Seiden, Theoretical computer science cheat sheet, ACM SIGACT News, 27(4), 1996

Covered in (1)this course, (2)introduction to mathematics course, (3)probability course, (4)linear algebra course

Preface

These supplementary lecture notes contain material covered in the the course "Algorithms and Data Structures" at Aarhus University on the first semester of the undergraduate program in computer science. The primary text book used in the course is *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein (3^{rd} edition, 2009) [5].

The notes will be updated continuously throughout the course Fall 2020.

Gerth Stølting Brodal
Fall 2020

Contents

Chapter 1

Introduction to Algorithms

Algorithms are methods to solve problems. Often algorithms are considered in the context of computer programs, where the focus of describing an algorithm is to capture the general idea of how to solve a problem — without having to spell out all the details required by a computer program in a specific programming language. Algorithms can also address problems not necessarily executed by computers. In this chapter we discuss a few simple problems, we present algorithms to solve these problems, and we give examples of ways to reason about the algorithms and problems.

When designing algorithms the focus is to develop *correct* and *efficient* algorithms. A correct algorithm always produces the required output given a valid input, i.e. it is not allowed to give a wrong output for any valid input. To prove the correctness of an algorithm one tries to reason formally about the algorithm using mathematical tools. We measure the efficiency of an algorithm by assigning a cost measure with each execution of the algorithm. Depending on the context many different measures of efficiency can be considered: How much time does it take to the solve the problem (for some abstract notion of execution time of a an algorithm)? How many times does the algorithm compare two numbers? How much computer memory does the algorithm need to solve the problem? How many times does the algorithm access the memory of your computer? How many times does the algorithm access the hard disk? e.t.c. Usually the focus is on only one cost measure when designing an algorithm, assuming this is a bottleneck in the computation. Sometimes it can be proved formally that there is a trade-off between the cost-measures, e.g. that it is impossible to be both fast and using little memory.

1.1 Tent pole folding

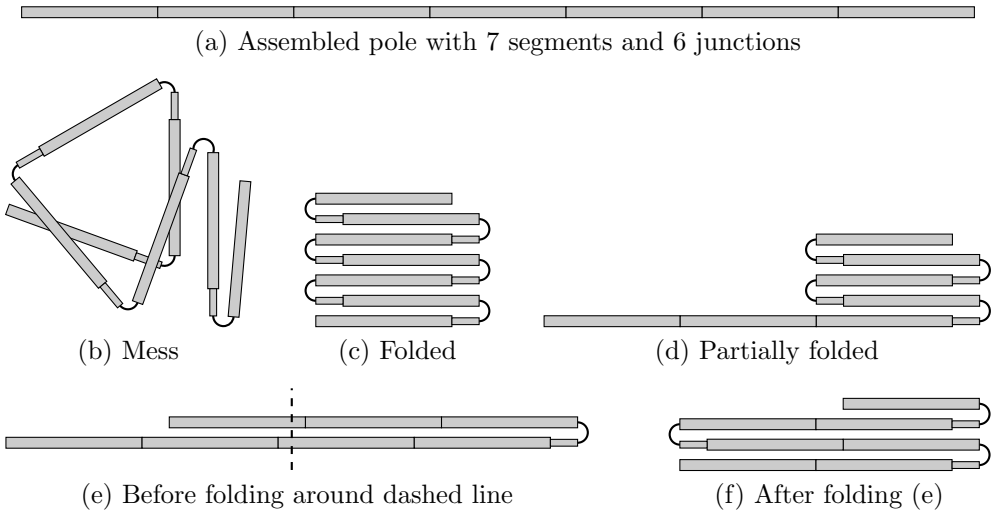
The first problem we consider is the problem of folding a *tent pole* consisting of seven segments with an elastic cord inside for easy handling. We are interested in the minimum number of folds required to completely fold an assembled pole. An assembled pole has six junctions, where it should be unplugged and folded. We want to nicely pack the segments in parallel, i.e. we want to get from the state in Figure 1.1(a) to the state in Figure 1.1(c). By just unplugging the six junctions we can end up in a mess like Figure 1.1(b).

A simple way, or algorithm, to fold a pole, is to unplug the junctions sequentially from one end to the order end, and whenever unplugging a junction to fold 180° around this point. Figure 1.1(d) shows the state after such four folds: The folded segments are nicely parallel with the last segment of the yet unfolded part.

In general, a pole of n segments has $n-1$ junctions. This is easy to see: The right ends of each of the segments are unique junctions, except for the last segment that does not have a junction at its right end. Since the above algorithm makes exactly one fold for each junction, we have the following general result.

Theorem 1 *A tent pole with n segments can be folded with $n-1$ folds.*

The question that naturally arises: Is this the best possible? Can we use fewer folds? If we are limited to only be able to unplug one junction by a fold, then the number of folds required is



(a) Assembled pole with 7 segments and 6 junctions

(b) Mess (c) Folded (d) Partially folded

(e) Before folding around dashed line (f) After folding (e)

Figure 1.1: Tent pole folding

at least the number of junctions, i.e. at least $n-1$ folds are required. In this scenario, the above algorithm achieves the minimum number of folds possible and we say it is *optimal*.

Let us consider a natural generalized fold: If all segments are parallel, like in Figure 1.1(e), then we can by a single pull unplug all junctions around a point, the dashed line in Figure 1.1(e), and fold around all these junctions by a single fold, arriving at Figure 1.1(f). This essentially allows us to consider the state in Figure 1.1(e) as a single pole with 4 segments, and the fold to result in a new pole with 2 segments.

Our second algorithm tries to reduce the length of the pole as much as possible, by always making a fold at the middle point of the pole. If a pole has length m, i.e. consists of m segments, a fold will reduce the length to $m/2$ — provided m is even. If m is odd, we cannot split exactly at the middle point (we are not allowed to break the pole). Instead we fold at the immediate junction to the left or right of the middle point. Both choices will split the pole into two pieces: One of length $\lceil m/2 \rceil$ and one of length $\lfloor m/2 \rfloor$. After folding around this point the resulting pole has length $\lceil m/2 \rceil$. We repeat folding until the resulting pole consists of one segment.

For a tent pole of length seven, see Figure 1.1(a), we perform the following three folds. The initial pole is split into pieces of length $\lceil 7/2 \rceil = 4$ and $\lfloor 7/2 \rfloor = 3$, and the result of the fold has length four and is shown Figure 1.1(e). Folding this pole at the middle results in the state in Figure 1.1(f), and finally a fold at the middle of Figure 1.1(f) brings us to the desired state Figure 1.1(c). We conclude that:

Lemma 1 *A tent pole of length 7 can be folded with 3 folds.*

Again some questions arise: Does the algorithm generalize to an arbitrary number of segments and what is the resulting number of folds expressed as a function of the initial pole length? Is this the best possible?

Quiz 1 check

How many folds does the algorithm use for folding a pole with 20 segments?

1 2 3 4 5 6 7 8 9 10

To analyze the general case, assume we start with a pole of length $n \geq 2$. The first fold will reduce the length to $\lceil n/2 \rceil$. The second fold to length $\lceil \lceil n/2 \rceil /2 \rceil$, the third fold to length $\lceil \lceil \lceil n/2 \rceil /2 \rceil /2 \rceil$, etc. until the length has been reduced to a single segment. The ceilings in the expressions unfortunately make it quite cumbersome to work with. The easy case is when n is

a power of two, say $n = 2^k$. Since each fold exactly halves the length of the pole, it follows that after i folds the length is reduced to exactly $n/2^i = 2^{k-i}$, and exactly k folds are required to reduce the length to a single segment.

We can use this to make a general analysis. Any pole length n is uniquely between two consecutive powers of two, $2^{k-1} < n \leq 2^k$. A single fold will result in a pole of length $\lceil n/2 \rceil$ and $2^{k-2} < \lceil n/2 \rceil \leq 2^{k-1}$, provided $n \geq 2$. In general after i folds the length will be n_i, where $2^{k-1-i} < n_i \leq 2^{k-i}$. It follows that after exactly k folds the length n_k is reduced to $1/2 < n_k \leq 1$. Since n_i is an integer, we have $n_k = 1$, and the algorithm performs exactly k folds. Since $2^{k-1} < n \leq 2^k$, we can apply the binary logarithm function to each side of the inequalities, and get

$$\log_2(2^{k-1}) < \log_2 n \leq \log_2(2^k) \ ,$$

i.e. $k - 1 < \log_2 n \leq k$ and we get $k = \lceil \log_2 n \rceil$. The performance of the algorithm can be summarized by the following lemma.

Lemma 2 *A tent pole with n segments can be folded with $\lceil \log_2 n \rceil$ folds.*

The question that remains: Can we do better? This question can be addressed by either trying to come up with further algorithms and analyze these, and hope they perform better, or by proving that no algorithm can perform better. We will take the second approach.

Any algorithm that makes a single fold to a pole of length n, will fold it into two pieces of length x and $n - x$, and the resulting length is $\max(x, n - x)$. Since $\max(x, n - x) \geq n/2$ for all values x, the length after the fold is *at least* $n/2$, independently of what the algorithm does. Note that $n/2$ is not necessarily an integer. It follows that any algorithm that performs i folds, can at most reduce the length to $n/2^i$. If the number of folds $i < \log_2 n$, then $n/2^i > n/2^{\log_2 n} = n/n = 1$, and the length is still strictly larger than one segment. If follows that at least $\log_2 n$ folds are required to fold a pole with n segments. Since the number of folds is an integer, it follows that the number folds required by any algorithm is at least $\lceil \log_2 n \rceil$. We can summarize our insights in the following lemma.

Lemma 3 *Any algorithm folding a tent pole with n segments requires at least $\lceil \log_2 n \rceil$ folds to reduce the length to one segment.*

We conclude that our algorithm performs the minimal number of folds possible. We say that the algorithm performs an *optimal* number of folds.

Summary: The tent pole example is an algorithmic problem that can be stated independently of computers and computer programs. Our cost measure is the number of folds performed. We analyzed the performance of a specific algorithm and gave an upper bound on the number of folds performed by this specific algorithm. Furthermore we argued generally about the behavior of any algorithm and proved that there is a lower bound for the number of folds that need to be performed by any algorithm. The binary logarithm showed up naturally when reasoning about the problem.

1.2 Swap sorting – The mathematics of solving a Puzzle

Our second algorithmic problem is to solve a puzzle by swapping pieces. Consider a rectangular puzzle consisting of n squared pieces, where all pieces have been permuted. All pieces have the correct orientation except that they may be placed at a wrong position. Your goal is to find the *minimum number* of *pairwise swaps* required to solve the puzzle, i.e. to bring all pieces to their correct positions. A single swap takes two arbitrary pieces x and y and interchanges the two pieces. Figure 1.2 shows a puzzle and a possible first swap.

A simple algorithm to solve this problem is to repeatedly identify a misplaced piece x and to swap x with the piece y occupying the correct position of x. This algorithm, that we name PUZZLE, can be described by the *pseudocode* in Figure 1.2. Pseudocode is a mixture of constructs found in common programming languages, mathematical notation, and textual descriptions. The aim with pseudocode is to be sufficiently precise so that the overall idea of how to solve a problem is conveyed — but without giving all the details. This allows one to focus on the overall ideas of

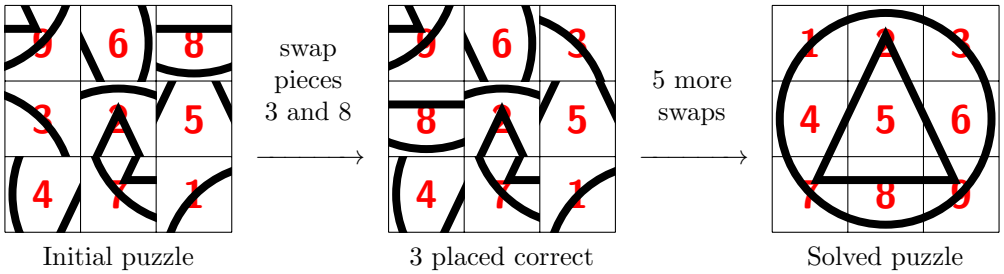


Figure 1.2: Puzzle by swapping pieces

different solutions. The underlaying assumption is that the reader of the pseudocode is able to fill in the omitted details in a meaningful manner. Depending on the detailedness of the pseudocode there may be many ways to fill in the details. An example is the algorithm in Figure 1.3 where it is unspecified in what order misplaced pieces should be considered.

Algorithm PUZZLE
1 **while** there exists a misplaced piece x **do**
2 let y be the piece at x's correct position
3 swap x and y

Figure 1.3: Algorithm to solve a puzzle by swaps

Several natural questions arise about the algorithm in Figure 1.2 and the problem in general: How many swaps are performed by the algorithm? Does the number of swaps performed depend on the order the misplaced pieces are considered? Can one come up with another algorithm that performs fewer swaps? How does the minimal number of required swaps depend on the start configuration of the pieces? What is the worst-case number of swaps necessary for a puzzle of size n, i.e. what is a worst-case permutation of the pieces? These are all algorithmic questions stated without having to talk about computer programs.

We start by making some observations about algorithm PUZZLE.

Lemma 4 *Algorithm* PUZZLE *never moves a piece placed correctly.*

Proof. Swapping two pieces x and y is triggered by x not being at its correct position. Furthermore, y is occupying x's correct position, i.e. y's correct position is not its current position. It follows that when the algorithm swaps two pieces x and y, both pieces are misplaced before the swap. □

Lemma 5 *For any puzzle with n pieces, algorithm* PUZZLE *performs at most $n-1$ swaps.*

Proof. Since each swap moves the piece x to its correct position, and neither x and y were at their correct position before the move (by Lemma 4), we know that each move decreases the number of misplaced pieces by at least one (the number of misplaced pieces is decreased by two if y happens to also be moved to its correct position by a swap). Since we start with at most n misplaced pieces, after k swaps at most $n-k$ pieces can be misplaced, and no piece is misplaced at the end, the total number of swaps is at most n. By observing that we cannot have a configuration where exactly one piece is misplaced (since the other $n-1$ pieces occupy their correct positions), it follows that after $n-1$ swaps all pieces must be at their correct position. □

The above lemma allows us to draw the conclusion that *at most* $n-1$ swaps are necessary to solve any puzzle of size n, i.e. we have proved an *upper bound* on the *worst-case complexity* of solving puzzles of size n. It is not immediate that there exist start configurations for the puzzle requiring $n-1$ swaps. In the following we will argue there indeed exist such start configurations.

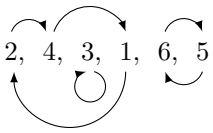
1.2.1 Tighter analysis

Obviously the number of swaps required to solve a puzzle depends on the start configuration, e.g. if all pieces happen to be placed correctly then no swaps are required. In the following we will give a more refined analysis of the number of swaps performed by the algorithm PUZZLE for a given puzzle and also give a lower bound on the number of swaps required by any algorithm for a given puzzle. The next exercise asks you to prove that there exist inputs requiring at least $\lceil n/2 \rceil$ swaps by any algorithm.

Exercise 1.1 Argue that if none of the n pieces in the start configuration are placed at their correct position, then any algorithm will require at least $\lceil n/2 \rceil$ swaps on such an input. ◁

In the following we will describe the state of a puzzle with n pieces by a list of n integers. Assume we have numbered the positions of the puzzle 1 to n, enumerating the first row left-to-right, followed by the second row, etc. The top-left position is position 1 whereas the bottom right is position n (see Figure 1.2 (right)). We identify each piece by its correct position: the ith element in the list is the correct position of the piece currently at position i. Example $(2, 4, 3, 1, 6, 5)$ is a puzzle where piece 2 is currently at position 1, piece 4 for at position 2, piece 3 is at its correct position, piece 1 at position 4 in the list, and pieces 5 and 6 are mutually interchanged. A solved puzzle corresponds to $(1, 2, 3, \ldots, n)$.

If we for each piece draw an arrow to its correct position in the list, we get the below picture. Note that a piece that is at its correct position has an arrow to itself.



Since there is exactly one outgoing and one ingoing arrow for each position in the list, the arrows give rise to a partition of the pieces into a set of *cycles*: A cycle consists of the nodes reachable by following the arrows from a start node until one gets back to the start node. In the above example there are three cycles: pieces 2, 4, and 1 form a cycle, pieces 6 and 5 form a cycle, and piece 3 is a cycle by itself.

Lemma 6 *When all n pieces are at their correct position there are exactly n cycles.*

Proof. When each of the n pieces is placed correctly, it will be a cycle by itself, and the lemma follows. □

Quiz 2 check

How many cycles are there in the puzzle configuration $(3, 4, 5, 2, 1, 6)$?

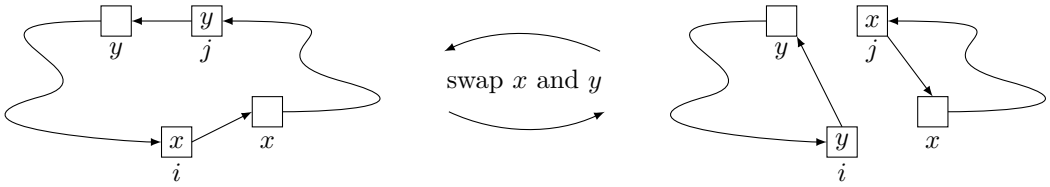
1 2 3 4 5 6

Exercise 1.2 What are the cycles in the puzzle in Figure 1.2 (left)? ◁

The hardness of a puzzle is related to the number of cycles in the start configuration. The following lemma captures the effect of swapping two pieces on the number of cycles.

Lemma 7 *Swapping two pieces in the same cycle increases the number of cycles by one, and swapping two pieces in two different cycles reduces the number of cycles by one.*

Proof. Assume the two pieces to be swapped are pieces x and y currently at positions i and j, respectively. They point at their correct positions x and y, respectively. By swapping x and y, they still point at their correct positions, but x is now at position j, and y at position j. If x and y are in the same cycle, we are in the left situation below (possibly with positions $x = j$ and/or $y = i$). The swap breaks the existing cycle into two cycles, i.e. increases the number of cycles by one. If instead x and y were in two different cycles (right situation below), then the swap will join the two cycles, i.e. reduce the number of cycles of one.



The cases when one or two of the initial or resulting cycles have length one, is just a special case of the above. □

Note that the above lemma is independent of the placement of the two pieces — it holds independently of if the pieces are placed at their correct position before or after the swap.

Exercise 1.3 Can the puzzle in Figure 1.2 (left) be solved using 9 swaps? (Note that a swap requires two distinct pieces). ◁

We will first use the lemma to give a refined analysis of the algorithm Puzzle.

Lemma 8 *On a puzzle with n pieces and initially k cycles, algorithm* Puzzle *performs exactly $n - k$ swaps.*

Proof. Each swap by the algorithm swaps a piece x with the piece y at the correct position of x, i.e. x and y are in the same cycle and y is the successor of x in the cycle. By swapping x and y we create one more cycle, in particular x becomes a cycle by itself. It follows that each swap creates exactly one more cycle. Since the initial number of cycles is k and the final number of cycles is n, exactly $n - k$ swaps must be formed. □

Lemma 9 *Any algorithm that solves a puzzle with n pieces and k cycles in the start configuration requires at least $n - k$ swaps.*

Proof. Since any swap can create at most one additional cycle, at least $n - k$ swaps are required to get from k to $n - k$ cycles. □

Since algorithm Puzzle performs the provably best possible number of swaps, we say that the algorithm *optimal*. The following two exercises stretches the point that optimal swap sequences can contain many swaps that do not move any piece to a correct position, but still the swaps provably contribute towards solving the puzzle.

Exercise 1.4 Give a start configuration of a puzzle with four pieces and an optimal sequence of swaps (i.e. performs a minimal number of swap) that solves the puzzle, but where at least one of the swaps does not move any piece to its correct position. ◁

The next exercise asks you to generalize your answer from Exercise 1.4 to an arbitrary number of pieces.

Exercise 1.5 Given a number of pieces n, give a start configuration of a puzzle with n pieces and an optimal sequence of swaps (i.e. performs a minimal number of swap) that solves the puzzle, but *maximizing the number of swaps which do not move any piece to their correct position*. Argue that this is the worst-case number of swaps. *Hint.* The largest number of swaps not moving any element to a correct position is $\lfloor \frac{n}{2} \rfloor - 1$. ◁

1.2.2 Strategies for finding swaps

Algorithm Puzzle does not specify how to find the piece x to be moved to its correct position. We here give two simple strategies, see the pseudocodes in Figure 1.4.

The first algorithm, PuzzlePull, scans through the positions of the puzzle, and first fills the first position with a correct piece, then the second position, etc. by performing at most a single swap. When considering position x, there are two possibilities. If position x contains piece x, then we proceed to the next position $x + 1$ without performing a swap. Otherwise position x

```
Algorithm PuzzlePull
1  for x = 1 to n do
2      if x is not placed correctly then
3          swap x with the piece y at position x

Algorithm PuzzlePush
1  for i = 1 to n do
2      while position i does not contain piece i do
3          swap the piece x at position i with the piece y at position x
```

Figure 1.4: Two variations of algorithm PUZZLE

contains a piece y, and we find x on the board and swap x and y. It is an *invariant* of this algorithm that whenever we consider a position x, then all positions $1, \ldots, x-1$ contain the correct pieces.

Our second algorithm variation, PuzzlePush, also considers the positions of the puzzle in increasing order. But whenever considering position i, and finds here a misplaced piece x, then instead of looking for piece i on the board, it just pushes x to its correct position by swapping it with the current piece y at position x. Such a swap will not necessarily guarantee that position i is filled with its correct piece, so we just repeat the same step until position i gets the correct value. Since each swap moves one new piece to its correct position we still only have to perform $n - k$ swaps.

1.2.3 Random puzzle

We have seen that the number of cycles in the initial configuration of a puzzle determines the optimal number of swaps needed to solve the puzzle. For a random permutation of the pieces (where all permutations are equally likely with probability $\frac{1}{n!}$), the following theorem (proof in Section 1.7) captures the number of expected cycles in the permutation of the pieces. From the theorem it follows that the expected optimal number of swaps for a random puzzle is $n - \sum_{i=1}^{n} \frac{1}{i}$.

Theorem 2 *The expected number of cycles in a random permutation of n pieces is*

$$H_n = \sum_{i=1}^{n} \frac{1}{i} .$$

H_n denotes the n-th harmonic number and is approximately

$$H_n \approx \ln n + \gamma + \frac{1}{2n} - \frac{1}{12}n^{-2} + \frac{1}{120}n^{-4} - \frac{1}{252}n^{-6} + \cdots ,$$

where $\gamma = 0.577215664901\ldots$ is the Euler-Mascheroni constant.

1.3 Selection sort

There exist many different sorting algorithms. In this section we discuss the selection sorting algorithm, with the focus on reasoning about a simple algorithm. Sorting by selection is likely one of the most canonical ways of sorting. Assume you have a pile of n cards C labeled with distinct numbers, and you want to sort the cards in order of increasing labels. Algorithm SelectionSort proceeds by first finding the card with minimum label in C, and moving this to a stack S (initially empty). We now repeatedly remove the card with minimum label from C and put it on top of S. The algorithm terminates when all cards have been moved from C to S, and S is the sorted result.

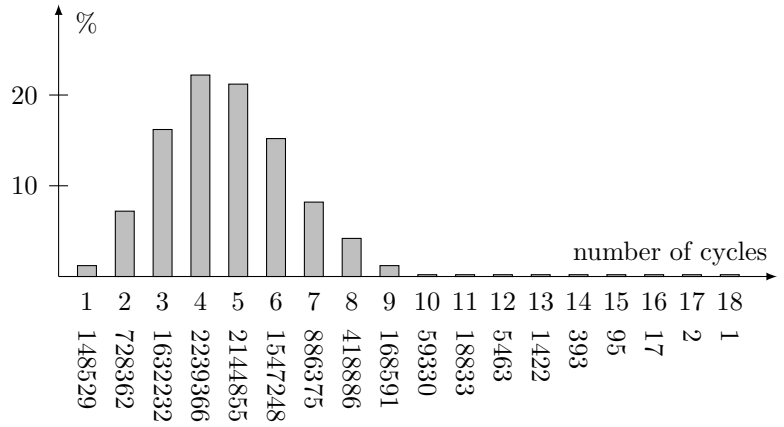
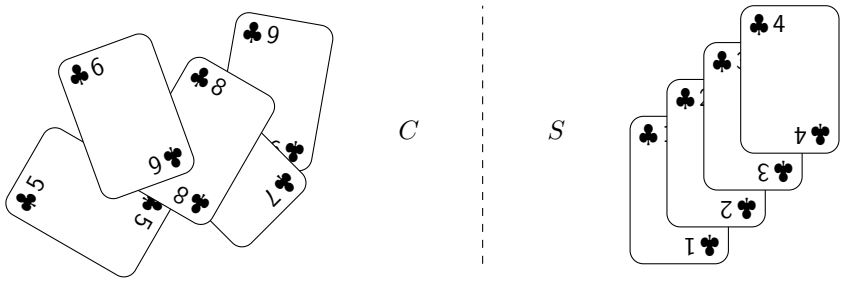


Figure 1.5: Distribution of the number of cycles in 10.000.000 random puzzles with 64 pieces. The expected number of cycles is $H_{64} \approx 4.7439$.



It might be clear to you already that this process will extract the cards in increasing label order, but let us argue a little bit more formally about the process. Throughout the execution of SelectionSort the states of C and S satisfy some properties. We will use *invariants* to capture these properties, and more importantly to argue that the output is the correct result. We will consider the following three invariants:

(a) C and S together equal the initial set of cards.

(b) S is sorted in increasing label order (lowest label at the bottom).

(c) All cards in S have smaller labels than all cards in C.

To argue that these statement are in fact invariants for SelectionSort, we need to argue that: *i*) the invariants are satisfied in the initial state, and *ii*) whenever the algorithm moves a card from C to S in a state where the invariants are satisfied, then the invariants are also satisfied after the move.

In the initial state C contains the initial set of cards whereas S is empty, implying (a)–(c) are satisfied. Whenever a card x is moved from C to S, the cards in C and S are the same cards as before the move, i.e. if (a) is satisfied before the move, then (a) is also satisfied after the move. To argue that (b) is satisfied after the move, we need (b) and (c) to be satisfied before the move. When moving a card x from C to the top of S, we by (c) know that x is larger than all cards currently on S, and by (b) that S is sorted, i.e. S remains sorted by moving x to the top of S, and (b) is satisfied after the move (note that this argument holds for any card x moved from C to S). Finally, to argue that (c) remains satisfied by moving the minimum card x from C to S, we note that (c) before the move ensures that all previous cards in S are smaller than all cards in C. Since we only add x to S, and x was the smallest card in C, then x is also smaller than all remaining cards in C, i.e. (c) remains satisfied.

To conclude that the algorithm computes the correct result, we first need to argue that the algorithm in fact *terminates*. But since $|C|$ decreases by exactly one for each move, the total

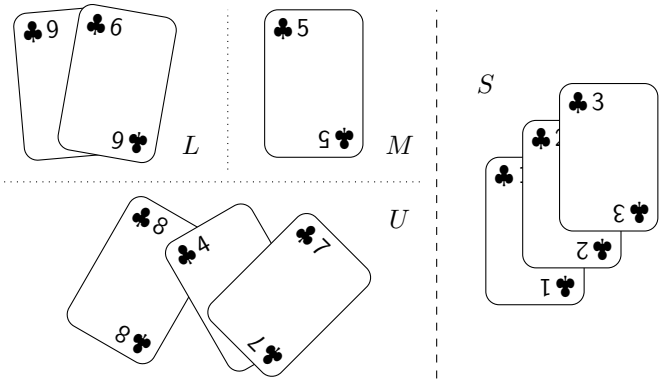


Figure 1.6: A state during the minimum search in algorithm SelectionSort

number of moves is n, i.e. the algorithm is guaranteed to terminate. To reason about the final state, we first observe that when algorithm terminates C is empty. Invariant (a) then implies that S contains the initial set of cards and (b) that these are sorted. The above argument allows us to conclude that:

Theorem 3 *Algorithm* SelectionSort *correctly sorts cards in increasing label order.*

Note that we only needed invariants (a) and (b) to conclude that S is the correct sorted output in the final state. Invariant (c) is needed to be able to reason about the intermediate states, in particular we could not reason that (b) is an invariant, without having (c) in our set of invariants.

Quiz 3 – SelectionSort check

State for each of the below statements if they are valid invariants for SelectionSort ?

	yes	no		yes	no
$1 \leq \|C\| \leq n$			$\|S\| \leq \|C\|$		
$0 \leq \|C\| \leq n$			$\min C \leq \max S$		
$\|S\| + \|C\| = n$			$\max S \leq \min C$		

We let max (min) on an empty set be $-\infty$ ($+\infty$).

1.3.1 Finding the minimum

Algorithm SelectionSort repeatedly finds the card with minimum label among the remaining cards C. Let us spell out this (easy) task. The simplest way to do this is to run through the cards in C, and to keep track of the mimimum card seen to far. To capture the progress of this search we can partition C into three sets of cards U, M and L: The set of cards not considered yet U, the current minimum M, and the set of rejected cards L with labels larger than M. The situation is illustrated in Figure 1.6. The invariants that should be maintained by the algorithm are:

(a) L, U, M and S together equal the initial set of cards.

(b) S is sorted in increasing label order (lowest label at the bottom).

(c) All cards in S have smaller labels than all cards in $L \cup U \cup M$.

(d) M contains at most one card.

(e) If L is non-empty, then M contains exactly one card, and this card has smaller label than all cards in L.

At the beginning of the minimum search all cards are in U, and L and M are empty. If M is empty (and L is empty by invariant (e)), we move an arbitrary card from U to M. While U is non-empty, pick from U an arbitrary card c and compare its label with the card in M. If c is larger, we move c to L. Otherwise the card in M is moved to L and c is movd to M. When U becomes empty, we move the card in M to the top of S, and move all cards from L to U, and start over. By a simple case analyis it can be verified that each move maintains all the above invariants.

We are now in a position where we can analyze the number of comparisons performed between the labels of two cards by algorithm SelectionSort.

Theorem 4 *Algorithm* SelectionSort *sorts n cards using $\frac{n(n-1)}{2}$ label comparisons.*

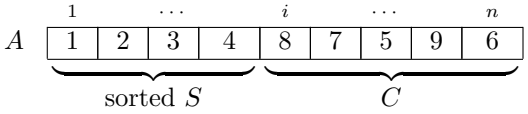
Proof. For the ith card moved to S, this is found among the $n+1-i$ remaining cards in C. Since we perform exactly one comparison per card we move out of U, except for the first card, this requires $|C|-1 = n+1-i-1 = n-i$ comparisons. It follows that the total number of comparisons is $(n-1)+(n-2)+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$. □

Exercise 1.6 Consider the card with the ith smallest label. What is the smallest number and largest number of comparisons this card can participate in? In particular what are these values for the cards with smallest label ($i = 1$) and largest label ($i = n$)? ◁

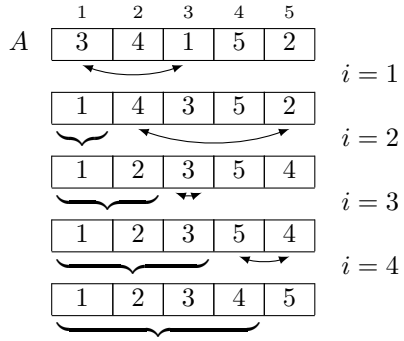
1.3.2 Pseudocode

Until now we have studied the properties of SelectionSort without refering to pseudocode. When turning to code, we assume that we have an array $A[1..n]$ of n values that should be sorted. Instead of creating two additional arrays C and S, storing the remaining values to be sorted and the already sorted values, we will work directly on A, by swapping values in A, letting S be represented by a prefix of A, and C the remaining values. Since the algorithm works directly on the input array without creating new arrays we denote SelectionSort to be an *implicit algorihtm*.

The progress of the algorithm can be captured by the below *visual invariant* together with invariants (a)–(c). Position i is the next position to be filled with the minimum from $C = A[i..n]$.



The algorithm stated as pseudocode is shown in SelectionSortAbstract in Figure 1.7. Below is illustrated the progression of the algorithm on an input of size five. The underbraced part is the sorted part corresponding to S, and arrows indicate the next swap. Note that when $A[i] = \min A[i..n]$, then $A[i]$ is just swapped with itself (below the case where $i = 3$), and that $A[n]$ is guaranteed to be the largest value after $n-1$ swaps.



```
Algorithm SelectionSortAbstract(A)
1  for i = 1 to |A| − 1 do
2      swap A[i] and minimum of A[i..|A|]
```

```
Algorithm SelectionSort(A)
1  for i = 1 to |A| − 1 do
2      k = i
3      for j = i + 1 to |A| do
4          # A[k] = min A[i..j − 1]
5          if A[j] < A[k] then
6              k = j
7      # A[k] = min A[i..|A|]
8      tmp = A[i]
9      A[i] = A[k]
10     A[k] = tmp
```

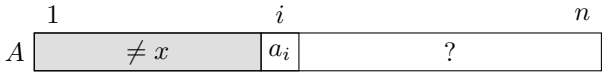
Figure 1.7: Pseudocode for SelectionSort.

Again, SelectionSortAbstract does not provide the details of how to find $\min A[i..n]$. The pseudocode SelectionSort in Figure 1.7 (right) spells out these details, in particular how to find the minimum of $A[i..n]$ by letting j run over the positions $i+1..n$ and remembering the position k with smallest value among the values considered so far, i.e. $A[k] = \min A[i..j-1]$. The last three lines swap $A[i]$ and $A[k]$. The pseudocode is very close to real code — but the basic idea behind the solution gets blurred.

1.4 Linear search

In this section we consider the most simple searching algorithm called *linear search* as a warm up for the binary search in Section 1.5. Assume we want to find an element x in an array A of length n containing elements $a_1, a_2, \ldots, a_n$, or to report that x is not contained in A. The obvious algorithm is to compare x to a_1, then x and a_2, etc. until we find the i where $a_i = x$, or we have compared x unsuccessfully to all elements in A. In the worst-case we perform exactly n comparisons, namely one between x and each of the n elements in A.

In Figure 1.8 the details are spelled out as pseudocode, in particular we represent the result "x not in A" by returning the value -1 (a somewhat arbitrary choice). We maintain an index i where a_i is the next element to be compared to i, and more importantly we have the invariant that $a_j \neq x$ for all $1 \leq j < i$. We can illustrate this by the visual invariant:



The visual invariant captures the progress of the algorithm. It clearly states what the first index and last index of the array are (not being explicit abouth this is a common source for programming errors, since sometimes algorithms assume arrays to start at 0, sometimes at 1, and sometimes even at -1 if this becomes convenient for describing the pseudocode), and that i is the next index to be considered (opposed to the last index found to be $\neq x$). One special case not illustrated well by the visual invariant, is that at the end i can have value $n+1$, when all values of A have been considered. This is special case is better captured by the following general mathematical invariant.

$$1 \leq i \leq n+1 \ \wedge \ x_j \neq x \text{ for all } 1 \leq j < i$$

By stating both the visual invariant and the mathematical invariant it should be quite clear, if not obvious, how to fill in the code. For linear search it might not appear that necessary to provide visual invariants, mathematical invariants, and pseudocode to solve the problem. But for slightly more complex algorithms they become crucial tools in the process of reasoning about the algorithm. Binary search is such an example.

```
Algorithm LinearSearch(A, x)
Input    Array A[1..n] and search key x
Output   Index i where A[i] = x; −1 if x ∉ A
1  i = 1
2  while i ≤ |A| do
3      if A[i] = x then
4          return i
5      i = i + 1
6  return −1
```

Figure 1.8: Linear search

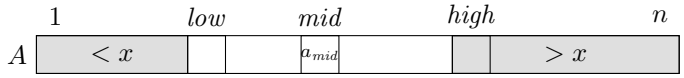
1.5 Binary search

For linear search we did not assume anything about A except that we should be able to test if two elements are equal. The price was that in the worst-case we have to do n comparisons. Binary search considers the situation where we can arrange the elements in A in sorted order with respect to some ordering of the elements. The goal by sorting the elements is to perform repeated searches with fewer comparisons. This is an example of a very simple *data structure*, where we can perform efficient queries on our data by putting a structure on the data.

That binary search can be tricky to get right is well known. Jon Bentley [2, Section 4.1] asked a group of professional programmers to implement binary search (without testing):

> *I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his* Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.*

Assume we are searching for an element x in a sorted array $A = a_1, a_2, \ldots, a_n$. The basic idea of *binary search* can be captured by the following invariant.



$$(1 \leq \mathit{low} \leq \mathit{high} \leq n+1) \ \wedge \ (a_j < x \text{ for all } 1 \leq j < \mathit{low}) \ \wedge \ (x < a_j \text{ for all } \mathit{high} \leq j \leq n)$$

The idea is to have two indexes *low* and *high* into A, where we have discarded the prefix $a_1, a_2, \ldots, a_{low-1}$ of A as being elements smaller than x, and the suffix $a_{high}, a_{high+1}, \ldots, a_n$ of A as being elements larger than x. The elements $a_{low}, \ldots, a_{high-1}$ are the remaining candidates that can be equal to n. To reduce the set of candidates we compare x with the middle element a_{mid}, where $mid = \lfloor (low + high)/2 \rfloor$ and $low \leq mid < high$. If $x = a_{mid}$ we have found x and can return *mid*. Otherwise, if $x < a_{mid}$ all elements $a_{mid}, \ldots, x_n$ are $> x$, since A is sorted, and we can decrease *high* to *mid*. Finally, if $a_{mid} < x$ all elements $a_1, \ldots, a_{mid}$ are $< x$, and we can increase *low* to $mid + 1$. If the candidate set is reduced to the empty set, we know x is not in A, and return this. Figure 1.9 gives the pseudocode.

Let us now turn towards the number of comparisons performed by BinarySearch. We assume that a single comparison between x and $A[mid]$ determines if $x < A[mid]$, $x = A[mid]$, or $x > A[mid]$. If $s = high - low$ is the number of candidates before a comparison where $a_{mid} \neq x$, then the number of candidates is reduced to $\lfloor s/2 \rfloor$ if s is odd, and $s/2 - 1$ or $s/2$ if k is even. In the worst case always $\lfloor s/2 \rfloor$ elements remain. To analyze how many comparisons BinarySearch performs in the worst case on an array of size n, we need to analyze how many times we need to divide by two and round down until we reach zero.

$$n \to \lfloor n/2 \rfloor \to \lfloor \lfloor n/2 \rfloor /2 \rfloor \to \lfloor \lfloor \lfloor n/2 \rfloor /2 \rfloor /2 \rfloor \to \cdots \to 0$$

Fortunately, if the length is a power of two minus one, i.e. $n = 2^k - 1$, then this sequence becomes manageable:

$$2^k - 1 \to 2^{k-1} - 1 \to 2^{k-2} - 1 \to 2^{k-3} - 1 \to \cdots \to 2^{k-(k-1)} - 1 \to 2^{k-k} - 1 = 0$$

```
Algorithm BinarySearch(A, x)
Input    Sorted array A[1..n] and search key x
Output   Index i where A[i] = x; −1 if x ∉ A
1  low = 1
2  high = n + 1
3  while low < high do
4      mid = ⌊(low + high)/2⌋
5      if x = A[mid] then
6          return mid
7      else if x < A[mid] then
8          high = mid
9      else if x > A[mid] then
10         low = mid + 1
11 return −1
```

Figure 1.9: Binary search

i.e. exactly k comparisons are required in the worst case. For an array of length $n \leq 2^k - 1$, the worst case size of the candidate set after i comparisons is at most the corresponding worst case size of the candidate set after i comparisons for input size $2^k - 1$. It follows that on any input of size $n \leq 2^k - 1$, at most k comparisons are performed by BinarySearch. By adding 1 on both sides, and taking the binary logarithm, we can rewrite the constraint on n to

$$\log_2(n+1) \leq \log_2(2^k) = k \; .$$

For a given n, this is satisfied for $k = \lceil \log_2(n+1) \rceil$, i.e. the algorithm performs at most $k = \lceil \log_2(n+1) \rceil$ comparisons.

Theorem 5 *Binary search on array of length n performs worst case $\lceil \log_2(n+1) \rceil$ comparisons.*

Example 1.1 Consider an array A of size 1.000.000. Here LinearSearch in the worst case performs 1.000.000 comparisons, whereas BinarySearch at most performs $\lceil \log_2 1.000.001 \rceil = 20$ comparisons. □

Quiz 4 check

How many comparisons does BinarySearch in the worst case perform on an array of size 1.000.000.000.000?

30 40 400 1600 20.000 1.000.000 20.000.000

Exercise 1.7 Can line 4 in BinarySearch be changed to $mid = \lceil (low + high)/2 \rceil$? ◁

Exercise 1.8 Describe a variant of binary search where *low* is the index of the last element of A known to be $< x$. Modify the invariant and the pseudocode to match the new invariant. ◁

1.5.1 Optimality of binary search

We have seen that in the worst case the algorithm BinarySearch requires $\lceil \log_2(n+1) \rceil$ comparisons for searching in a sorted arrays of size n. In this section we show that this is the best possible for algorithms that are only allowed to do comparisons on the elements.

Assume that an arbitrary searching algorithm is given a sorted array $A = a_1, \ldots, a_n$ and an element x — but the only information the algorithm can gain about x and the a_is is by comparing x with an a_i, and will only learn if $x < a_i$, $x = a_i$ or $x > a_i$. Our goal is to construct an *adversary* that answers the comparison queries by the algorithm in such a way that it forces

the algorithm to make as many comparisons as possible — essentially the adversary tries to postpone settling the value for x as long as possible.

The adversary fixes $a_1 = 1, a_2 = 2, \ldots, a_n = n$. It also maintains a range $]\ell, h[$ that x can be assigned values from, while being consistent with the already answered comparison queries. Initially $]\ell, h[=]0, n+1[$. Whenever the algorithm compares x with an a_i, the adversary answers $x < a_i$ if $i \leq \ell$, and answers $x > a_i$ if $i \geq h$. If $\ell < i < h$, the adversary sets $h = i$ if $|A \cap]\ell, i[\,| \geq |A \cap]i, h[\,|$, otherwise $\ell = i$. After this $i \notin]\ell, h[$ and the adversary answers the query as above. If before the query there are $s = |A \cap]\ell, h[\,|$ possible values for x, then after the query there are still at least $\lceil (s-1)/2 \rceil$ possible values for x. As long as there is at least one possible value for x in $A \cap]\ell, h[$, the algorithm cannot have determined if x is in A, since the adversary is still free to assign x a value from A or one not in A.

A lower bound for the number of comparisons required by the algorithm is how many times we need to apply $s \to \lceil (s-1)/2 \rceil$, to get from having n candidate values for x until there is zero. For $n = 2^k$ the sequence of lower bounds we have on the number of candidates becomes

$$2^k \to 2^{k-1} \to 2^{k-2} \to \cdots \to 2^{k-k} \to 0 \,,$$

i.e. at least $k+1$ comparisons are required by any algorithm to be able to answer correctly. For $n \geq 2^k$, also at least $k+1$ comparisons are required, since it is sufficient for the adversary to run the adversary strategy on the first 2^k elements of A. Letting $k = \lfloor \log_2 n \rfloor$, implies $n = 2^{\log_2 n} \geq 2^k$, i.e. at least $k+1 = \lfloor \log_2 n \rfloor + 1$ comparisons are required. Since $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$, we have the following theorem.

Theorem 6 *Any comparison based searching algorithm on a sorted array requires worst case at least* $\lceil \log_2 (n+1) \rceil$ *comparisons.*

Corollay 1 *Algorithm* BinarySearch *performs an optimal worst case number of comparisons.*

1.6 Logarithms

In this section we summarize important properties of logarithm functions typically used in the analysis algorithms. We have already seen several examples involving logarithms in the previous sections. In the analysis of the tent pole problem in Section 1.1 and the algorithm BinarySearch in Section 1.5 the binary logarithm showed up naturally, both in the analysis in the respective algorithms, and the lower bound arguments for the problems. In Section 1.2 we analyzed the number of swaps required to solve a puzzle, and related this to the number of cycles in a permutation, that according to Theorem 2 for a random puzzle was expected to be close to the natural logarithm of the number of pieces.

Definition 1.1 For a real value $x > 0$ and real *base* $b > 1$, the logarithm of x with base b, denoted $\log_b x$, is defined as the inversion of the exponential function

$$y = \log_b x \;\Leftrightarrow\; b^y = x$$

□

Figure 1.10 illustrates three often used logarithmic functions: the binary logarithm $\log_2 x$, the decimal logarithm $\log_{10} x$, and the natural logarithm $\ln x = \log_e x$. All logarithms have $\log_b 1 = 0$. Some very useful identities involving logarithm functions are the following, where $x > 0$, $y > 0$, and p can be any real value.

$$\log_b(x \cdot y) = \log_b x + \log_b y \tag{1.1}$$

$$\log_b(x/y) = \log_b x - \log_b y \tag{1.2}$$

$$\log_b(x^p) = p \cdot \log_b x \tag{1.3}$$

$$\log_b x = \frac{\log_a x}{\log_a b} \tag{1.4}$$

$$\log_b(b^p) = p \tag{1.5}$$

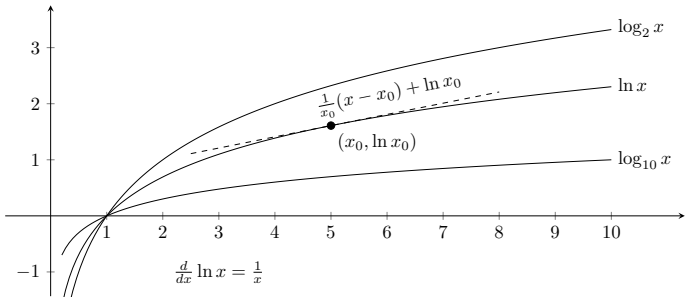


Figure 1.10: Logarithm functions $\ln x$, $\log_2 x$ and $\log_{10} x$

To prove (1.1) and (1.2) we use $a^{b+c} = a^b \cdot a^c$ and $a^{b-c} = a^b / a^c$ to get the identites

$$b^{\log_b x + \log_b y} = b^{\log_b x} \cdot b^{\log_b y} \stackrel{\text{def}}{=} x \cdot y \stackrel{\text{def}}{=} b^{\log_b (x \cdot y)}$$

$$b^{\log_b x - \log_b y} = b^{\log_b x} / b^{\log_b y} \stackrel{\text{def}}{=} x / y \stackrel{\text{def}}{=} b^{\log_b (x/y)} ,$$

from which it follows that $\log_b x + \log_b y = \log_b (x \cdot y)$ and $\log_b x - \log_b y = \log_b (x/y)$, since $f(x) = b^x$ is a strictly increasing function for $b > 1$. Similarly for (1.3) we use $(a^b)^c = a^{b \cdot c}$ and have

$$b^{\log_b (x^p)} \stackrel{\text{def}}{=} x^p \stackrel{\text{def}}{=} (b^{\log_b x})^p = b^{(\log_b x) \cdot p} = b^{p \cdot \log_b x} ,$$

i.e. $\log_b (x^p) = p \cdot \log_b x$. Equation 1.4 follows from

$$a^{\log_a x} \stackrel{\text{def}}{=} x \stackrel{\text{def}}{=} b^{\log_b x} \stackrel{\text{def}}{=} (a^{\log_a b})^{\log_b x} = a^{(\log_a b) \cdot (\log_b x)} ,$$

i.e. $\log_a x = (\log_a b) \cdot (\log_b x)$. Finally, (1.5) follows from (1.3) by

$$\log_b (b^p) \stackrel{(1.3)}{=} p \cdot \log_b b \stackrel{\text{def}}{=} p \cdot 1 = p .$$

The relationship between two logarithm functions with different bases a and b is captured by (1.4), stating that the difference is a constant factor $\log_a b$. In subsequent chapters we often ignore constant factors when using the so called O-notation, and (1.4) therefor in many cases allows us to omit the base on the logarithm.

The natural logarithm $\ln x$ is the logarithm with base $e = 2.7182818\ldots$, i.e. $\ln x = \log_e x$. The natural logarithm is characterized by having derivative equal to one at $x = 1$. Some essential properties of the natural logarithm are

$$\frac{d}{dx} \ln x = \frac{1}{x} \tag{1.6}$$

$$\ln y = \int_1^y \frac{1}{x} dx \tag{1.7}$$

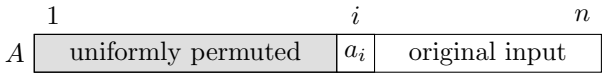
$$H_n - \ln n \to \gamma \quad \text{for} \quad n \to \infty , \tag{1.8}$$

where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the nth harmonic number and $\gamma = 0.577215664901\ldots$ is the Euler-Mascheroni constant.

1.7 The number of cycles in a random permutation

In this section we will sketch the proof of Theorem 2, i.e. that a random permutation has expected H_n cycles. The proof illustrates how we can use the analysis of an algorithm to obtain mathematical insights in random permutations.

The first question is how to generate a random permutation. One solution is the algorithm SHUFFLE in Figure 1.11 (left). The algorithm considers the elements $a_1, a_2, \ldots, a_n$ one by one, maintaining the invariant that when the algorithm considers a_i, $A[1..i-1]$ contains a random permutation of the $i-1$ elements $a_1, \ldots, a_{i-1}$, and all $(i-1)!$ permutations have equal probability $1/(i-1)!$.



By swapping a_i with any element from $A[1..i]$, the algorithm maintains this invariant since this random choice is (intuitively and mathematically) independent from previous choices: a_i remains at position i with probability $1/i$, i.e. any permutation with a_i as the last element has probability $1/i \cdot 1/(i-1)! = 1/i!$, since $A[1..i-1]$ had this permutation with probability $1/(i-1)!$. If a_i is swapped with $A[j]$, for a specific j, where $1 \leq j < i$, then this happens with probability $1/i$. Any permutation of $a_1, \ldots, a_i$ with $A[j] = a_i$, has been created by swapping a_i with the previous $A[j]$. Since the permutation before this swap had probability $1/(i-1)!$, the resulting permutation of $a_1, \ldots, a_i$ has probability $1/i \cdot 1/(i-1)!) = 1/i!$. It follows that after having processed all $a_1, \ldots, a_n$ all permutations have equal probability $1/n!$.

Algorithm SHUFFLE(A)
Input Array $A[1..n]$ to be randomly shuffled
1 **for** $i = 1$ **to** n **do**
2 j = random integer in $\{1, 2, \ldots, i\}$
3 swap $A[i]$ and $A[j]$

Algorithm BADSHUFFLE(A)
Input Array $A[1..n]$ non-uniformly shuffled
1 **for** $i = 1$ **to** n **do**
2 j = random integer in $\{1, 2, \ldots, n\}$
3 swap $A[i]$ and $A[j]$

Figure 1.11: Permute A randomly

It should be emphasized that the random integer in line 2 in algorithm SHUFFLE is taken from the range $1, \ldots, i$. In algorithm BADSUFFLE(A), Figure 1.11 (right), i has been replaced by n in line 2. This innocent change unfortunately makes the algorithm generate permutations with non-uniform probabilities, e.g. if $A = [1, 2, 3]$, then it can be shown that $[1, 2, 3]$, $[3, 1, 2]$, $[3, 2, 1]$ are generated with probability $4/27$, whereas $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$ are generated with probability $5/27$.

Finally let us turn to the expected number of cycles in the generated random permutations. When algorithm SHUFFLE adds a_i to the permuted part, there are two possible actions: Either a_i stays at position i, in which case a_i becomes a new cycle of length one. Otherwise, the swap inserts a_i into an existing cycle and the number of cycles does not change. It follows that when considering a_i, with probability $1/i$ one new cycle is introduced, and with probability $1 - 1/i$ the number of cycles remains unchanged. It follows that expected $1/i$ cycles are introduced when including a_i in the permutation. Summing over all i, the total number of cycles introduced is expected

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i} = H_n \overset{(1.8)}{\approx} \ln n \, .$$

1.8 Algorithms on integers

Computers are number crunchers. The basic unit stored in a computer is a *bit*, which is a number that either equals zero or one. The power of computers is achieved by how we interpret and work with bits. The physical RAM (random access memory) of a modern computer stores on the order 100.000.000.000 bits. In this section we will consider basic algorithms for working with integers presented by a sequence of bits.

1.8.1 Number representations

Let us start with a recap of the basic definition of decimal numbers. For small numbers zero, one, two, ..., nine we have special symbols to represent these values, denoted *digits*.

Symbol/digit	0	1	2	3	4	5	6	7	8	9
Unary		|	||	|||	||||	|||||	||||||	|||||||	||||||||	|||||||||

For numbers larger than nine, we use a *positional number system* where we write several digits after each other. In particular ten is written 10. The value of an n digit decimal number

$d_{n-1}d_{n-2}\cdots d_1d_0$ equals

$$\sum_{i=0}^{n-1} d_i \cdot 10^i = d_{n-1} \cdot 10^{d_{n-1}} + d_{n-2} \cdot 10^{d_{n-2}} + \cdots + d_1 \cdot 10^1 + d_0 \cdot 10^0 \ .$$

Here we assume the reader to familiar with basic arithmetic and that $10^0 = 1$. An example could be the value

$$1849_{10} = 1 \cdot 10^3 + 8 \cdot 10^2 + 4 \cdot 10^1 + 9 \cdot 10^0 \ .$$

Note that we subscript the value by the *base* of the number system. If the base is obvious from the context the base will in most cases by omitted. Otherwise the default base is 10.

Other typical positional digit systems are binary numbers (base 2), octal numbers (base 8), and hexadecimal numbers (base 16). A base b system uses b different symbols for the values zero to $b-1$. For hexadecimal numbers we are in the special case that we can only reuse the digits 0 – 9 for the values zero to nine, but for the values ten to fifteen new symbols are needed. Here the convention is to use the six letters A – F.

Decimal value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary symbol	0	1														
Octal symbol	0	1	2	3	4	5	6	7								
Hexadecimal symbol	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

The value of an n digit number $d_{n-1}d_{n-2}\cdots d_1d_0$ in base b equals $\sum_{i=0}^{n-1} d_i \cdot b^i$, e.g. $\text{ACDC}_{16} = 10 \cdot 16^3 + 12 \cdot 16^2 + 13 \cdot 16^1 + 12 \cdot 16^0 = 44252$.

$$1849_{10} = 11100111001_2 = 3471_8 = 739_{16}$$

The popularity of octal and hexadecimal numbers is due to the fact that 8 and 16 are both powers of two, respectively 2^3 and 2^4. This makes it easy to convert a binary number to the corresponding octal and hexadecimal numbers. For octal numbers groups of three bits make up one octal digit, whereas for hexadecimal numbers one considers blocks of four bits.

$$\underbrace{011}_{3}\underbrace{100}_{4}\underbrace{111}_{7}\underbrace{001}_{1}{}_2 = 3471_8 \qquad \underbrace{0111}_{7}\underbrace{0011}_{3}\underbrace{1001}_{9}{}_2 = 739_{16}$$

In the above we prepend the binary number with the digit 0 to make the length divisible by three and four, respectively.

1.8.2 Addition

Consider computing the decimal sum $843 + 572 = 1415$. The classic addition algorithm (in the following denoted the school method) so is to write the numbers below each other, and proceed right-to-left adding the two digits above each other and possibly a carry, write down the least significant digit below the numbers and the possible most significant digit as a carry over the next position to the left.

$$\begin{array}{rcccc} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & & 8 & 4 & 3 \\ + & & 5 & 7 & 2 \\ \hline & 1 & 4 & 1 & 5 \end{array}$$

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

+	0	1
0	0	1
1	1	10

Figure 1.12: Decimal (left) and binary (right) addition tables

This essentially states the following rewriting in a very compact form and using the addition table in Figure 1.12 (left):

$$\begin{aligned}
843 + 572 &= (\mathbf{8} \cdot 10^2 + \mathbf{4} \cdot 10^1 + \mathbf{3} \cdot 10^0) + (\mathbf{5} \cdot 10^2 + \mathbf{7} \cdot 10^1 + \mathbf{2} \cdot 10^0) \\
&= (\mathbf{8} + \mathbf{5}) \cdot 10^2 + (\mathbf{4} + \mathbf{7}) \cdot 10^1 + (\mathbf{3} + \mathbf{2}) \cdot 10^0 \\
&= (8 + 5) \cdot 10^2 + (4 + 7) \cdot 10^1 + \mathbf{5} \cdot 10^0 \\
&= (8 + 5) \cdot 10^2 + \mathbf{11} \cdot 10^1 + 5 \cdot 10^0 \\
&= (8 + 5) \cdot 10^2 + (\mathbf{1} \cdot 10 + \mathbf{1}) \cdot 10^1 + 5 \cdot 10^0 \\
&= (8 + 5) \cdot 10^2 + (\mathbf{1} \cdot 10^2 + \mathbf{1} \cdot 10^1) + 5 \cdot 10^0 \\
&= ((8 + 5) + \mathbf{1}) \cdot 10^2 + \mathbf{1} \cdot 10^1 + 5 \cdot 10^0 \\
&= (\mathbf{13} + 1) \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 \\
&= \mathbf{14} \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 \\
&= (\mathbf{1} \cdot 10 + \mathbf{4}) \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 \\
&= \mathbf{1} \cdot 10^3 + \mathbf{4} \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 \\
&= 1415
\end{aligned}$$

Binary numbers are added in exactly the same way, except that we use the binary addition table Figure 1.12 (right). In particular note that $1_2 + 1_2 = 10_2$.

$$\begin{array}{rr}
 & {\scriptstyle 1\ 1\ 1} \\
 & 1\ 1\ 1\ 0\ {}_2 \\
+ & 1\ 0\ 1\ 1\ {}_2 \\
\hline
 & 1\ 1\ 0\ 0\ 1\ {}_2
\end{array}$$

The above binary addition, corresponds to the following computation. Note that we allow us to write ${10_2}^3$ where the base is in binary representation and the exponent is in decimal notation,

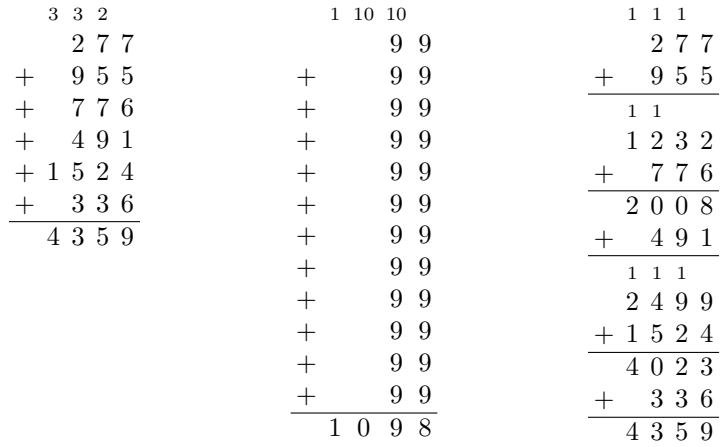


Figure 1.13: School method for adding multiple decimal numbers

e.g. ${10_2}^3 = 1000_2$.

$$
\begin{aligned}
1110_2 + 1011_2 &= (\mathbf{1} \cdot {10_2}^3 + \mathbf{1} \cdot {10_2}^2 + \mathbf{1} \cdot {10_2}^1 + \mathbf{0} \cdot {10_2}^0) \\
&+ (\mathbf{1} \cdot {10_2}^3 + \mathbf{0} \cdot {10_2}^2 + \mathbf{1} \cdot {10_2}^1 + \mathbf{1} \cdot {10_2}^0) \\
&= (\mathbf{1}+\mathbf{1}) \cdot {10_2}^3 + (\mathbf{1}+\mathbf{0}) \cdot {10_2}^2 + (\mathbf{1}+\mathbf{1}) \cdot {10_2}^1 + (\mathbf{0}+\mathbf{1}) \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + (1+0) \cdot {10_2}^2 + (1+1) \cdot {10_2}^1 + \mathbf{1} \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + (1+0) \cdot {10_2}^2 + (\mathbf{10}_2) \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + (1+0) \cdot {10_2}^2 + (\mathbf{10}_2+\mathbf{0}) \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + ((1+0)+\mathbf{1}) \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + (\mathbf{1}+1) \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + \mathbf{10}_2 \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (1+1) \cdot {10_2}^3 + (\mathbf{10}_2+\mathbf{0}) \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= ((1+1)+\mathbf{1}) \cdot {10_2}^3 + 0 \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= (\mathbf{10}_2+1) \cdot {10_2}^3 + 0 \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= \mathbf{1} \cdot {10_2}^4 + 1 \cdot {10_2}^3 + 0 \cdot {10_2}^2 + 0 \cdot {10_2}^1 + 1 \cdot {10_2}^0 \\
&= 11001_2
\end{aligned}
$$

1.8.3 Adding multiple numbers

The school method for the addition of two decimal numbers easy generalizes to adding multiple decimal numbers, see Figure 1.13. Instead of only adding two digits, you add all the digits above each other. This method has the drawback that you need to add more than two digits, where the addition table, Figure 1.12, only supports adding two digits. To solve this, you often end up doing an intermediate sequence of additions of a multiple digit number with a single digit number, i.e. an easier form of addition. E.g. in Figure 1.13 (left), for the first column you end up performing the following sequence of additions $7+5=12$, $12+6=18$, $18+1=19$, $19+4=23$, $23+6=29$. Another complication is that the carry is now not limited to the value 0 or 1, as is the case for adding only two numbers (in any base, actually). In Figure 1.13 (middle) is an example where the carry even has more decimal digits.

Exercise 1.9 Adding n decimal numbers (with an arbitrary number of digits), how big can the carry become? How many digits are there in the carry (when written in decimal)? ◁

To avoid having to deal with carries larger than one, the trick is to use repeated addition of two numbers, using the identity

$$x_1 + x_2 + \cdots + x_n = ((\cdots((x_1 + x_2) + x_3) + \cdots) + x_{n-1}) + x_n \ .$$

+	0	1	2	3	4	5	6	7	8	9
0	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
1	1	0	-1	-2	-3	-4	-5	-6	-7	-8
2	2	1	0	-1	-2	-3	-4	-5	-6	-7
3	3	2	1	0	-1	-2	-3	-4	-5	-6
4	4	3	2	1	0	-1	-2	-3	-4	-5
5	5	4	3	2	1	0	-1	-2	-3	-4
6	6	5	4	3	2	1	0	-1	-2	-3
7	7	6	5	4	3	2	1	0	-1	-2
8	8	7	6	5	4	3	2	1	0	-1
9	9	8	7	6	5	4	3	2	1	0

+	0	1
0	0	-1
1	1	0

Figure 1.14: Decimal (left) and binary (right) subtraction tables

See Figure 1.13 (right) for a resulting computation. For any number system this will guarantee that the carries are always either zero or one.

1.8.4 Negative numbers and subtraction

We will not go into the low level representation of negative numbers on computers, but just assume a negative number is represented as a number in some positional number system with a prefix "−", e.g. -42_{10}.

When subtracting two non-negative numbers $x - y$, we assume $x \geq y$, since otherwise we can just compute $-(y - x)$ instead, i.e. subtract x and y and negate the result. Similarly to addition, we can write the digits above each other, and subtract for each position, starting at the lowest position, possible with a *negative carry*.

$$\begin{array}{rcccc} & \scriptstyle -1 & \scriptstyle -1 & & \\ & 5 & 3 & 4 & 7 \\ - & 1 & 8 & 6 & 3 \\ \hline & 3 & 4 & 8 & 4 \end{array}$$

In the above example $7 - 3 = 4$ at position 0, but $4 - 6 = -2$ at position 1. Here we exploit $-2 = -10 + 8$, i.e. we get a carry of -1. For position 2, we have $3 - 8 - 1 = (3 - 8) - 1 = -5 - 1 = -(5 + 1) = -6 = -10 + 4$, i.e. again a carry of -1, and finally for position 3 we have $5 - 1 - 1 = (5 - 1) - 1 = 4 - 1 = 3$.

For binary represented numbers the calculation becomes similar.

$$\begin{array}{rcccccc} & \scriptstyle -1 & \scriptstyle -1 & & \scriptstyle -1 & & \\ & 1 & 0 & 0 & 1 & 0 & 1_2 \\ - & & 1 & 1 & 0 & 1 & 0_2 \\ \hline & 0 & 0 & 1 & 0 & 1 & 1_2 \end{array}$$

For position 0, we have $1 - 0 = 0$. For position 1, we have $0 - 1 = -1 = -10_2 + 1$, i.e. a -1 carry. Position 2, $1 - 0 - 1 = 0$. Position 3, $0 - 1 = -1 = -10_2 + 1$, and a -1 carry. Position 4, $0 - 1 - 1 = -10_2 + 0$, ie. a -1 carry. Finally position 5, $1 - 1 = 0$.

To subtract many numbers, we just do one subtraction and many additions using the identity

$$x_1 - x_2 - x_3 - \cdots - x_n = (\cdots((x_1 - x_2) - x_3) - \cdots) - x_n = x_1 - (x_2 + x_3 + \cdots + x_n) \, .$$

1.8.5 Multiplication

Addition and subtraction are easy operations, in the sense we can consider each position independently, possibly taking a carry into account from the previous position. The total number of bit operations, e.g. table lookups, required to add or subtract two binary numbers with n digits is on the order n. Multiplication is a harder problem.

The first simple observation is that multiplying a decimal number by 10, results in the same decimal number with a zero added to the right, e.g. $10 \cdot 42 = 420$. Furthermore, multiplying by the kth power of 10, corresponds to adding k zeors to the right, e.g. $10^3 \cdot 42 = 42000$. The same

·	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

·	0	1
0	0	0
1	0	1

Figure 1.15: Decimal (left) and binary (right) multiplication tables

applys for any base: Multiplying a base b number by $b^k = 10_b{}^k$ is the same b-ary number with k zeros added to the right.

Let us consider the computation of the product $365 \cdot 427$:

$$\begin{aligned} 365 \cdot 427 &= (3 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0) \cdot 427 \\ &= 3 \cdot 427 \cdot 10^2 + 6 \cdot 427 \cdot 10^1 + 5 \cdot 427 \cdot 10^0 \\ &= 1281 \cdot 10^2 + 2562 \cdot 10^1 + 2135 \cdot 10^0 \\ &= 128100 + 25620 + 2135 \\ &= 155855 \end{aligned}$$

A common way to write this is something like the below.

$$\begin{array}{r} 3\ 6\ 5\ \cdot\ 4\ 2\ 7 \\ \hline 1\ 2\ 8\ 1 \\ 2\ 5\ 6\ 2 \\ 2\ 1\ 3\ 5 \\ \hline 1\ 5\ 5\ 8\ 5\ 5 \end{array}$$

Addition of multiple numbers we know how to do from previous sections, but the basic step of multiplying a number with a digit has not been described yet, e.g. $6 \cdot 427$. Assuming we can multiply two digits, e.g. by a table lookup to Figure 1.15, we can do the following computation.

$$\begin{aligned} 6 \cdot 427 &= 6 \cdot \left(4 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0\right) \\ &= 6 \cdot 4 \cdot 10^2 + 6 \cdot 2 \cdot 10^1 + 6 \cdot 7 \cdot 10^0 \\ &= 24 \cdot 10^2 + 12 \cdot 10^1 + 42 \cdot 10^0 \\ &= 24 \cdot 10^2 + (12 + 4) \cdot 10^1 + 2 \cdot 10^0 \\ &= 24 \cdot 10^2 + 16 \cdot 10^1 + 2 \cdot 10^0 \\ &= (24 + 1) \cdot 10^2 + 6 \cdot 10^1 + 2 \cdot 10^0 \\ &= 25 \cdot 10^2 + 6 \cdot 10^1 + 2 \cdot 10^0 \\ &= 2 \cdot 10^3 + 5 \cdot 10^2 + 6 \cdot 10^1 + 2 \cdot 10^0 \\ &= 2562 \end{aligned}$$

The above calculations can be compactly written by something like the below.

$$\begin{array}{r} {\scriptstyle 2\ 1\ 4} \\ 6\ \cdot\ 4\ 2\ 7 \\ \hline 2\ 5\ 6\ 2 \end{array}$$

When turning to multiplying numbers in the binary representation calculations becomes simpler, since multiplying a number x with a digit d is either x if $d = 1$ or 0 if $d = 0$. Multiplying two numbers x and y with n bits each now reduces to adding n numbers, each being either 0 or the binary representation of x shifted some positions to the left (i.e. with between 0 and $n - 1$ zeros added as the least significant digits).

$$\begin{array}{r} 1\,0\,1\,1\,1_2 \cdot 1\,0\,1\,0\,1_2 \\ \hline 1\,0\,1\,0\,1_2 \\ 0\,0\,0\,0\,0_2 \\ 1\,0\,1\,0\,1_2 \\ 1\,0\,1\,0\,1_2 \\ 1\,0\,1\,0\,1_2 \\ \hline 1\,1\,1\,1\,0\,0\,0\,1\,1_2 \end{array}$$

In total this implies that the total number of bit operations is on the order n^2, since we are adding n numbers each with at most $2n$ bits.

Exercise 1.10 Let x be the binary number $111\cdots111_2$ consisting of n digits all equal to 1. What is the binary representation of x^2? ◁

In practice the following observation is used to speed up multiplications. Assume $x = \overbrace{11111}^{j}\overbrace{00000}^{i}{}_2$. Then we have $x = 1\overbrace{0000000000}^{i+j}{}_2 - 1\overbrace{00000}^{i}{}_2$. To compute $x \cdot y$, for some y, it is sufficient to compute a single subtraction $y \cdot 2^{i+j} - y \cdot 2^i$, instead of adding j numbers $y\cdot 2^i + y\cdot 2^{i+1} + \cdots + y\cdot 2^{i+j-1}$. It follows when we multiply x and y, we can replace a consecutive block of 1s in the binary representation of x by a single addition and a single subtraction. If there are few (say k) consecutive blocks of 1s in the binary representation of x, we only need to perform few additions and subtractions (k additions and at most k subtractions; we only have subtractions for blocks containing at least two 1s).

$$\begin{array}{r} 1\;0\;\overbrace{1\;1\;1}_2 \cdot 1\,0\,1\,0\,1_2 \\ \hline 1\,0\,1\,0\,1_2 \\ \begin{cases} + & 1\,0\,1\,0\,1_2 \\ - & 1\,0\,1\,0\,1_2 \end{cases} \\ \hline 1\,1\,1\,1\,0\,0\,0\,1\,1_2 \end{array}$$

Exercise 1.11 What is the order of bit operations done to multiply two binary numbers x and y, each with n digits, and where x has k blocks of 1s? ◁

Karatsuba in 1960 [13] presented a, by now classic, simple divide-and-conquer algorithm to multiply two n bit binary numbers using on the order $n^{\log_2 3} = n^{1.58\cdots}$ bit operations, significantly improving the n^2 bound achieved by the school method (that at this point of time was conjectured to be the best possible). In 1971, Schönhage and Strassen improved the bound further to $n \cdot \log n \cdot \log\log n$. Karatsuba in 1995 wrote an article on the history on multiplication [12]. Only recently, Harvey and van der Hoeven in 2019 [11] announced an algorithm for multiplying binary numbers where the number of bit operations is on the order $n \log n$ (with a gigantic constant).

1.8.6 Division

In this section we consider how to perform binary integer division, i.e. given two integers $x \geq 0$ and $y \geq 1$ in binary representation, to compute the binary representation of $i = \lfloor x/y \rfloor$.

Let us first recall how to do decimal division $\lfloor x/y \rfloor$. Assume x has n digits and y has m digits, and the most significant digit in y is non-zero, i.e. $y_{m-1} \neq 0$. Since $x < 10^n$ and $y \geq 10^{m-1}$, we have $\lfloor x/y \rfloor \leq x/y < 10^n/10^{m-1} = 10^{n-m+1}$, i.e. $\lfloor x/y \rfloor < 10^{n-m+1}$ and the result has at most $n - m + 1$ digits $d_{n-m}, \ldots, d_1, d_0$. We have

$$x = \left(d_{n-m} \cdot 10^{n-m} + \cdots + d_1 \cdot 10^1 + d_0 \cdot 10^0\right) \cdot y + r \;,$$

where r is the remainder satisfying $0 \leq r < y$. We find the digits of the result in decreasing index order, i.e. first we find d_{n-m}, then d_{n-m-1}, e.t.c. The remainder starts with $r = x$. We first find d_{n-m} to be the largest digit satisfying $d_{n-m} \cdot 10^{n-m} \cdot y \leq r$ and update r to $r - d_{n-m} \cdot 10^{n-m} \cdot y$, and then repeat for the remaining digits. To find d_i we can e.g. repeatedly apply addition by $y \cdot 10^i$, until the result becomes larger than r, or perform a binary search over the digits 0..9. Figure 1.16 gives an example of a decimal division.

$$
\begin{array}{rl}
\overbrace{2\ 7\ 0\ 3\ 9\ 4\ 0}^{x} & \overbrace{3\ 6\ 5}^{y} = \underset{d_4\ d_3\ d_2\ d_1\ d_0}{0\ 7\ 4\ 0\ 8} \\
\hline
2\ 7\ 0\ 3\ 9\ 4\ 0 & \\
-\ 0\ 0\ 0 & = 0 \cdot 10^4 \cdot 365 \\
\hline
2\ 7\ 0\ 3\ 9\ 4\ 0 & \\
-\ 2\ 5\ 5\ 5 & = 7 \cdot 10^3 \cdot 365 \\
\hline
\quad 1\ 4\ 8\ 9\ 4\ 0 & \\
-\ \quad 1\ 4\ 6\ 0 & = 4 \cdot 10^2 \cdot 365 \\
\hline
\qquad\quad 2\ 9\ 4\ 0 & \\
-\ \qquad\quad 0\ 0\ 0 & = 0 \cdot 10^1 \cdot 365 \\
\hline
\qquad\quad 2\ 9\ 4\ 0 & \\
-\ \qquad\quad 2\ 9\ 2\ 0 & = 8 \cdot 10^0 \cdot 365 \\
\hline
\qquad\qquad\quad 2\ 0 & = \text{remainder } r
\end{array}
$$

Figure 1.16: Decimal division $\lfloor 2703940/365 \rfloor = 7408$

To perform a binary division the algorithm is similar, except that it is simpler since each digit d_i is either zero or one. The idea is again to identify the digits at positions p in the binary representation of the result i in decreasing position order, starting with the maximum position p, where $y \cdot 2^p \leq x$. We start with the remainder $r = x$. Whenever $y \cdot 2^p \leq r$, we add 1 to the binary representation of i at position p, and subtract $y \cdot 2^p$ from r. This process maintains the invariant that $i \cdot y + r = x$ and that the remainder is bounded $r < y \cdot 2^{p+1}$, i.e. $\lfloor r/y \rfloor$ can be written with digits $d_p d_{p-1} \ldots d_1 d_0$. The pseudocode for the binary integer division is given in Figure 1.17 and an example is shown in Figure 1.18.

```
Algorithm IntegerDivision(x, y)
Input  Integers x ≥ 0 and y ≥ 1
Output Integer i = ⌊x/y⌋, i.e. 0 ≤ x − i · y < y
1  p = 0
2  while y · 2^(p+1) ≤ x do
3      p = p + 1
4  i = 0
5  r = x
6  # Invariant: x = i · y + r and r < y · 2^(p+1)
7  while y ≤ r do
8      if y · 2^p ≤ r then
9          i = i + 2^p    # set position p in i = 1
10         r = r − y · 2^p
11     p = p − 1
12 return i
```

Figure 1.17: Binary integer division

Let us consider the number of bit operations in a division. Assume x and y have at most n and m digits in their binary representation, respectively. The result i has at most $n - m + 1$ digits, i.e. at most $n - m + 1$ comparisons and subtractions of numbers are performed, each of length at most n. Observing that for each iteration we need to consider only the $m + 1$ most significant bits of the remainder in the subtraction, the total number of bit operations is at most on the order $(n - m + 1)(m + 1)$.

Note. As mentioned, multiplication can be performed significantly faster than order n^2. In fact division can be performed as fast as multiplication by using the Newton-Raphson method for finding roots of a function, and applying multiplication with increasing number of digits.

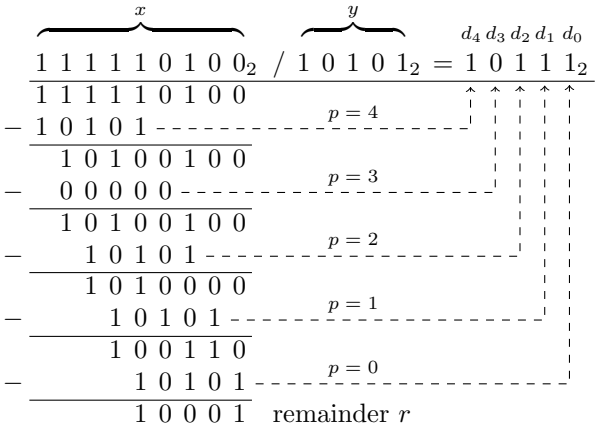


Figure 1.18: Binary division $\lfloor x/y \rfloor = \lfloor 500/21 \rfloor = \lfloor 111110100_2/10101_2 \rfloor = 10111_2 = 23$

Algorithm BaseRepresentation(x, b)
Input Integers $x \geq 0$ and base $b \geq 2$
Output Digits $d_0, d_1, \ldots$ of the b-ary representation of x

```
1  p = 0
2  while x > 0 do
3      i = ⌊x/b⌋    # IntegerDivision(x, b)
4      d_p = x − i · b
5      x = i
6      p = p + 1
```

Figure 1.19: Converting an integer to base b representation

1.8.7 Converting binary to decimal representation

Computers operate internally using the binary representation of numbers, but humans prefer numbers to be presented as decimal numbers. We can find the decimal representation of a number x by first computing the least significant digit $d_0 = x - 10 \cdot \lfloor x/10 \rfloor$, i.e. the remainder of dividing x by 10, and then repeatedly finding the binary representation of $\lfloor x/10 \rfloor$. Figure 1.19 shows pseudocode for converting a number to an arbitrary base b number representation. Note that line 3 uses the algorithm IntegerDivision. Since in each iteration we essentially make one division (algorithm IntegerDivision also finds the remainder, it is stored in the variable r). If n is the number of bits in the binary representation of x, then the number of digits in the b-ary representation is $x \approx \log_b x = (\log_2 x)/(\log_2 b) \approx n/\log_2 b$. Since a division takes order n^2 bit operations, the total number of bit operations is order $n^3/\log_2 b$.

Exercise 1.12 Assume you have an n bit binary number x, and you want to know the k most significant digits of the decimal representation of x. We want to avoid generating all the digits in the decimal representation of x, since n can be much larger than k. Describe an algorithm and state the order of bit operations performed as a function of k and n. ◁

1.9 Induction

A key concept in computer science is proofs by mathematical induction. Induction is used intensively in the analysis of algorithms, both to prove the correctness of algorithms and to analyze their resource requirements. But induction is also inherent to the design process of algorithms — but often just implicit. Before giving a formal definition of mathematical induction and a sequence of examples of proofs by mathematical induction, let us consider a very simple example of designing an algorithm.

Consider the following problem:

Compute the sum of n numbers $x_1, \ldots, x_n$.

If we let $S_i = x_1 + \cdots + x_i$ denote the sum of the first i numbers, the goal is to compute S_n. A simple observation is that S_n can be expressed in terms of S_{n-1} by the equation

$$S_n = \underbrace{x_1 + x_2 + \cdots + x_{n-1}}_{S_{n-1}} + x_n = S_{n-1} + x_n \ . \tag{1.9}$$

This observation is implicitly used in the standard way you would compute the sum of n numbers, expressed by the below expression, that also can be derived by unfolding the above observation until only x_1 remains.

$$S_n = \underbrace{((\cdots(\underbrace{(((x_1 + x_2) + x_3) + x_4)}_{S_{i-1}} + x_5)\cdots) + x_{n-1}) + x_n}_{S_i}$$

The above equality naturally gives raise to the following algorithm to compute $S_1, \ldots, S_n$. Note how (1.9) is explicit in line 3 of the code.

```
1  S_1 = x_1
2  for i = 2 to n do
3      S_i = S_{i-1} + x_i
```

Since we only need the final value S_n, there is no need to remember all $S_1, \ldots S_n$, except for the most recently computed. This will result in the following simplified code only maintaining a single sum S.

```
1  S = x_1
2  for i = 2 to n do
3      S = S + x_i
```

To argue about the correctness of this algorithm, one would go along stating that before an iteration of the loop we will have $S = S_{i-1}$ and after the iteration $S = S_i$ making essential use of (1.9). To stick the pieces together to a formal proof one applies mathematical induction.

1.9.1 Induction principle

Induction principle
Assume you want to prove a finite or infinite sequence of statements

$$P_1, P_2, P_3, \ldots, P_n, P_{n+1}, \ldots$$

The induction principle states that to prove all these statements, it is sufficient to prove

Base case (basis): P_1 is true.

Induction step: $P_n \Rightarrow P_{n+1}$ for all $n \geq 1$. We denote P_n to be the *induction hypothesis* (i.h.).

In the induction principle we have indexed the statements beginning with P_1. This is not important. Depending of the context it might be more convenient to have the first statement indexed P_0, P_2, P_{42} or something completely different. Also the induction principle is stated parameterized by the variable n. You can of course also use other variable names to do the induction on.

Sometimes it comes convenient to assume all $P_1, \ldots, P_n$ are true when proving P_{n+1}, instead of only using the induction hypothesis P_n. This corresponds to proving the following sequence of statements $Q_1, Q_2, \ldots, Q_n, Q_{n+1} \ldots$ by induction, where

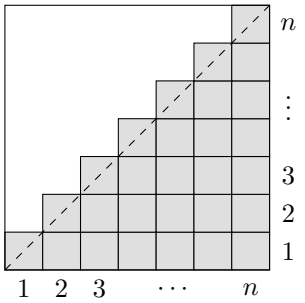
$$Q_n = P_1 \wedge P_2 \wedge \cdots \wedge P_n \ .$$

In the proof below for the Fibonacci numbers we apply this idea.

1.9.2 Finite sums

Our first application of mathematical induction is to prove some statements about finite and infinite sums repeatedly appearing in the analysis of algorithms. The first is $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n$, as e.g. appearing in the analysis of selection sort. That this sum equals $n^2/2 + n/2$ has a simple visual argument, see below right: The value i corresponds to i unit squares stacked on top of each other. The sum $1 + 2 + \cdots + n$ corresponds to the area of the gray squares. These clearly constitute half of the big $n \times n$ square plus n half squares (above the dashed diagonal). It follows that the sum has value $n^2/2 + n/2 = n(n+1)/2$.

$$
\begin{aligned}
P_1 &: & 1 &= \frac{1(1+1)}{2} = 1 \\
P_2 &: & 1+2 &= \frac{2(2+1)}{2} = 3 \\
P_3 &: & 1+2+3 &= \frac{3(3+1)}{2} = 6 \\
& & &\vdots \\
P_n &: & 1+2+\cdots+n = \sum_{i=1}^{n} i &= \frac{n(n+1)}{2}
\end{aligned}
$$



To prove the statement by induction, we define the infinite sequence of statements $P_1, P_2, \ldots$ where P_n is the statement $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Theorem 7 $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$ *for all* $n \geq 1$.

Proof. Proof by induction in n. The base case is when $n = 1$. Here $\sum_{i=1}^{n} i = 1 = \frac{1(1+1)}{2}$, i.e. the statement is true for the base case $n = 1$. For the induction step $n \geq 1$, we make the induction hypothesis that the statement is true for n, i.e. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. We should prove that the statement is true for $n+1$, i.e. $\sum_{i=1}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$. We have

$$
\sum_{i=1}^{n+1} i \stackrel{\text{def}}{=} (n+1) + \sum_{i=1}^{n} i \stackrel{\text{i.h.}}{=} (n+1) + \frac{n(n+1)}{2} = \frac{2(n+1) + n(n+1)}{2} = \frac{(n+1)((n+1)+1)}{2} ,
$$

i.e. the statement is true for $n+1$. By the induction principle $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \geq 1$. □

Our next sum is $\alpha^0 + \alpha^1 + \cdots + \alpha^n$. For $\alpha > 1$ this is the sum of exponentially increasing terms, but the following statement actually holds for all real α, except and $\alpha = 1$. For $\alpha = 1$ the sum $\sum_{i=0}^{n} 1^i$ is trivially n. Note that in the below proof the base case is when $n = 0$.

Theorem 8 $\sum_{i=0}^{n} \alpha^i = \alpha^0 + \alpha^1 + \alpha^2 + \cdots + \alpha^{n-1} + \alpha^n = \frac{\alpha^{n+1}-1}{\alpha-1}$, *for* $\alpha \neq 1$.

Proof. Proof by induction in n. The base case is when $n = 0$. Here $\sum_{i=0}^{0} \alpha^i = \alpha^0 = 1 = \frac{\alpha^1-1}{\alpha-1}$, since $\alpha \neq 1$, i.e. the statement is true for the base case $n = 0$. For the induction step when $n \geq 0$, we make the induction hypothesis that the statement is true for n, i.e. $\sum_{i=0}^{n} \alpha^i = \frac{\alpha^{n+1}-1}{\alpha-1}$. We have to prove the statement for $n+1$, i.e. $\sum_{i=0}^{n+1} \alpha^i = \frac{\alpha^{(n+1)+1}-1}{\alpha-1}$. We have

$$
\sum_{i=0}^{n+1} \alpha^i \stackrel{\text{def}}{=} \alpha^{n+1} + \sum_{i=0}^{n} \alpha^i \stackrel{\text{i.h.}}{=} \alpha^{n+1} + \frac{\alpha^{n+1}-1}{\alpha-1} = \frac{\alpha^{n+1}(\alpha-1) + \alpha^{n+1} - 1}{\alpha-1} = \frac{\alpha^{(n+1)+1} - 1}{\alpha-1} ,
$$

i.e. the statement is true for $n+1$. The theorem follows by the induction principle. □

Note that for $|\alpha| < 1$, we have $\lim_{n\to\infty} \frac{\alpha^{n+1}-1}{\alpha-1} = \frac{0-1}{\alpha-1}$.

Corollay 2 $\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$ *for* $|\alpha| \leq 1$.

For the special case $\alpha = 2$ we have.

Corollay 3 $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$.

For n a power of two, i.e. $n = 2^k$, we have $\sum_{i=0}^{k} 2^i = 1+2+4+\cdots+n/2+n = 2^{k+1}-1 = 2n-1$.

Corollay 4 $n + n/2 + n/4 + \cdots + 4 + 2 + 1 = 2n - 1$ *for* n *a power of two.*

1.9.3 Fibonacci numbers

The *Fibonacci numbers* are an infinite sequence of numbers $F_0, F_1, F_2, \ldots$ defined by $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. We say that the Fibonacci numbers are *recursively defined*. The Fibonacci numbers $F_0 \ldots F_{10}$ are:

n	0	1	2	3	4	5	6	7	8	9	10
F_n	0	1	1	2	3	5	8	13	21	34	55

The Fibonacci numbers grow rapidly, e.g. $F_{100} = 354224848179261915075$, in fact exponentially. Theorem 9 below states that $F_n \approx \varphi^n$, where $\varphi = \frac{1+\sqrt{5}}{2} = 1.6180\ldots$ is the *golden ratio*. The properties of the Fibonacci sequence have been intensely studied in mathematics, and in computer science they e.g. appear naturally in the analysis of the data structure called Fibonacci heaps [9]. The purpose of this section is to apply induction to analyze the properties of a recursively defined sequence of numbers.

Exercise 1.13 Compute F_{20}. ◁

Theorem 9 *For all $n \geq 1$, $\varphi^{n-2} \leq F_n \leq \varphi^{n-1}$ where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Proof. To prove an exponential upper bound on F_n, we will try to prove $F_n \leq c \cdot \alpha^n$ for some constants $c > 0$ and $\alpha \geq 1$. We will do this by a proof by induction. The statement P_n that we will prove by induction is the following for $n \geq 2$.

$$P_n : F_i \leq c \cdot \alpha^i \quad \text{for all} \quad 1 \leq i \leq n \ .$$

During the proof some restrictions will be realized on the possible values for c and α, for having a valid proof by induction.

We have $F_1 = 1$, i.e. $F_1 \leq c \cdot \alpha^1$ provided $c \geq 1/\alpha$, and $F_2 = 1 \leq c \cdot \alpha^2$ provided $c \geq 1/\alpha^2$. It follows that the base case $n = 2$ is true, provided $c \geq \max(1/\alpha, 1/\alpha^2)$. For the induction step $n \geq 2$, we consider the induction hypothesis that $F_i \leq c \cdot \alpha^i$ for all $1 \leq i \leq n$, and prove that this implies that $F_i \leq c \cdot \alpha^i$ for all all $1 \leq i \leq n+1$. To prove this, it is sufficient to prove $F_{n+1} \leq c \cdot \alpha^{n+1}$ under the induction hypothesis.

$$F_{n+1} \stackrel{\text{def}}{=} F_n + F_{n-1} \stackrel{\text{i.h.}}{\leq} c \cdot \alpha^n + c \cdot \alpha^{n-1} = c \cdot \alpha^{n+1} \cdot (\alpha^{-1} + \alpha^{-2}) \leq c \cdot \alpha^{n+1} \ ,$$

provided $\alpha^{-1} + \alpha^{-2} \leq 1$. We conclude that $F_n \leq c \cdot \alpha^n$, provided $c \geq \max\{1/\alpha, 1/\alpha^2\}$ and $\alpha^{-1} + \alpha^{-2} \leq 1$. The conditions reduce to $c \geq 1/\alpha$ (since $\alpha \geq 1$) and $0 \leq \alpha^2 - \alpha - 1$ (multiply by α^2 and rearrange). The later is satisfied for $\alpha \geq \frac{-(-1)+\sqrt{(-1)^2-4\cdot(-1)\cdot 1}}{2\cdot 1} = \varphi$. It follows that by setting $\alpha = \varphi$ and $c = 1/\varphi$, we have a valid induction argument and $F_n \leq c \cdot \alpha^n = \varphi^{-1} \cdot \varphi^n = \varphi^{n-1}$.

For the lower bound, we similarly assume $F_n \geq c \cdot \alpha^n$ for some constants $c > 0$ and $\alpha \geq 1$. Doing the same calculations but replacing "$\leq$" by "$\geq$", we arrive at the the conditions $c \leq \min\{1/\alpha, 1/\alpha^2\}$ and $\alpha \leq \varphi$. Setting $\alpha = \varphi$ and $c = \alpha^{-2}$ yields the lower bound $F_n \geq \varphi^{n-2}$. □

In the above proof we could have used $F_n \leq \varphi^{n-1}$ directly as the induction hypothesis without introducing c and α. This would simplify for the proof and the following exercise asks you to do this.

Exercise 1.14 Prove $F_n \leq \varphi^{n-1}$, for all $n \geq 1$, directly using this as an induction hypothesis. ◁

In fact the Fibonnacci numbers have a closed formula (Binet's formula), that you can prove by a simple induction argument using $1 + \varphi = \varphi^2$ and $2 - \varphi = (1 - \varphi)^2$,

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \quad \text{where} \quad \varphi = \frac{1+\sqrt{5}}{2} \ .$$

1.9.4 Bounding recurrence inequalities

Assume you have an infinite sequence of positive numbers $T_0, T_1, T_2, T_3, \ldots$. Furthermore assume that the numbers satisfy the following *recurrence inequality* upper bounding each of the T_n values by a function of the previous values.

$$T_n \leq \begin{cases} 1 & \text{if } 0 \leq n \leq 3 \\ 1 + T_{n-1} + T_{n-4} & \text{if } n > 3 \ . \end{cases}$$

Exercise 1.15 Use the recurrence inequality to give upper bounds for $T_0, \ldots, T_{20}$. ◁

This recurrence inequality pops up in the analysis of an algorithm for solving the independent set problem on graphs (Problem 1.3 and [7]). We will use induction to prove the following upper bound on T_n.

Theorem 10 $T_n \leq 2 \cdot 1.38028^n - 1$.

Proof. We prove by induction in n that $T_n \leq c \cdot \alpha^n - 1$ for constants c and α. The statement we prove by induction is the following for all $n \geq 3$:

$$P_n : T_i \leq c \cdot \alpha^i - 1 \quad \text{for all} \quad 0 \leq i \leq n \ .$$

The base case is P_3, where we have $T_i = 1 \leq c - 1 \leq c \cdot \alpha^i - 1$, for $0 \leq i \leq 3$, provided $c \geq 2$ and $\alpha \geq 1$. For the induction step we assume the induction hypothesis $T_i \leq c \cdot \alpha^i - 1$ to be true for all $0 \leq i \leq n$, and only need to prove $T_{n+1} \leq c \cdot \alpha^{n+1} - 1$. We have $T_{n+1} \overset{\text{def}}{\leq} 1 + T_n + T_{n-3} \overset{\text{i.h.}}{\leq} 1 + (c \cdot \alpha^n - 1) + (c \cdot \alpha^{n-3} - 1) = c \cdot \alpha^{n+1}(\alpha^{-1} + \alpha^{-4}) - 1 \leq c \cdot \alpha^{n+1} - 1$, provided $\alpha^{-1} + \alpha^{-4} \leq 1$, or equivalently $\alpha^4 - \alpha^3 - 1 \geq 0$, which is satisfied for $\alpha \geq 1.38028$. It follows that the statement is true for $c \geq 2$ and $\alpha \geq 1.38028$. □

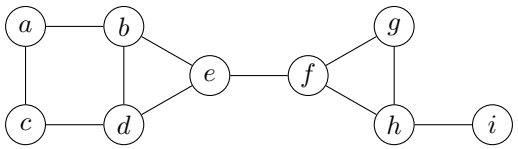
Exercise 1.16 Prove by induction that the following recurrence equality for $T_1, T_2, T_3, \ldots$

$$T_n = \begin{cases} 1 & \text{if } n = 1 \\ n + 2 \cdot T_{n-1} & \text{if } n > 1 \ , \end{cases}$$

has solution $T_n = 2^{n+1} - n - 2$. (This recurrence occurs e.g. in the analysis of building the binary heap data structure). ◁

1.9.5 Euler's formula for planar graphs

Next we will use an induction proof to prove an essential statement about planar graphs known as *Euler's formula*. Below is an example of a planar graph.



The graph consists of a set of *vertices* V (also called vertices) shown as circles, and a set of *edges* E connecting pairs of vertices show as strait lines between vertices (we assume that there is at most one edge between any pair of vertices, i.e. there are no parallel edges, and an edge does not connect a vertex with itself, i.e. there are no self-loops).

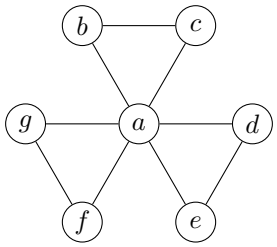
$$\begin{aligned} V &= \{a, b, c, d, e, f, g, h, i\} \\ E &= \{(a, b), (a, c), (b, d), (b, e), (c, d), (d, e), (e, f), (f, g), (f, h), (g, h), (h, i)\} \end{aligned}$$

A graph is *planar* if it can be drawn in the plane without crossing edges, we call such a drawing of the graph a *planar embedding*. A graph is *connected* if for all pairs of vertices u and v there

is a sequence of edges (a path) leading from u to v (you are allowed to walk in both directions along an edge). For a planar embedding we denote a region of the plane bounded by edges a *face*. In the above we have the faces bounded by the three cycles (a, b, c, d), (b, e, d) and (f, g, h), and the unbounded outer face, i.e. in total 4 faces.

Quiz 5 – Faces in a planar graph check

How many faces are there in the above planar graph?



1 2 3 4 5 6 7

Euler's formula relates the number of vertices V and edges E in a planar graph with the number of faces F in any embedding of the graph.

Theorem 11 (Euler's formula) *Any embedding of a planar connected graph with $V \geq 1$ vertices, E edges, and F faces satisfies $F + V - E = 2$.*

Proof. We will prove this by induction in the number of edges E in the graph. We first observe that any connected planar graph can be drawn by the following process: Draw a start vertex v, and repeatedly add edges by either *i*) connect two already drawn vertices by an edge, *ii*) connect a new vertex to the existing graph with an edge. Figure 1.20 shows an example. In each of the twelve steps the new edges and vertices are shown with bold edges. Steps 6, 7 and 11 are examples of *i*, whereas steps 2, 3, 4, 5, 8, 9, 10 and 12 are examples of *ii*.

The statements $P_0, P_1, \ldots$ we prove by induction is:

> P_n: Any planar connected graph with $E \leq n$ edges, $V \geq 1$ vertices, and F faces satisfies $F + V - E = 2$.

For the base case P_0, there is only one possible connected graph: A graph with single vertex and no edges. The unbounded face is the only face. We have $V = 1$, $E = 0$, and $F = 1$, i.e. $F + V - E = 1 + 1 - 0 = 2$ and the base case is true.

For the induction step assume the induction hypothesis P_n is true, i.e. all graphs with at most n edges satisfies the formula. We should prove $P_n + 1$, i.e. we should prove that any graph $n + 1$ edges satisfies the inequality. Consider any connected planar graph with $n + 1$ edges. This graph can be drawn using i and ii above. Before the last step, the graph has exactly n edges, and say V vertices and F faces. By construction this graph is connected, i.e. we can use the induction hypothesis to conclude $F + V - E = 2$. If the last step is of type i, two existing vertices are connected by an edge. This new edge is drawn through an existing face (either a bounded or the unbounded face). Steps 6 and 11 in Figure 1.20 split the unbounded region, whereas step 7 splits a bounded region. Since the two vertices were connected before inserting the edge, there must exist a path along the boundary of the split face between these two nodes, i.e. the new edge splits an existing face into two faces, where at least one of the faces is bounded. It follows that the number of vertices remains unchanged, whereas the number of edges and faces both increase by one, and we have $(F + 1) + V - (E + 1) = F + V - E \overset{\text{i.h.}}{=} 2$. On the other hand, if the last step is of type ii, then both the number of vertices and edges increase by one, but the number faces remains unchanged, and we have $F + (V + 1) - (E + 1)) = F + V - E \overset{\text{i.h.}}{=} 2$. We conclude P_{n+1} is true. □

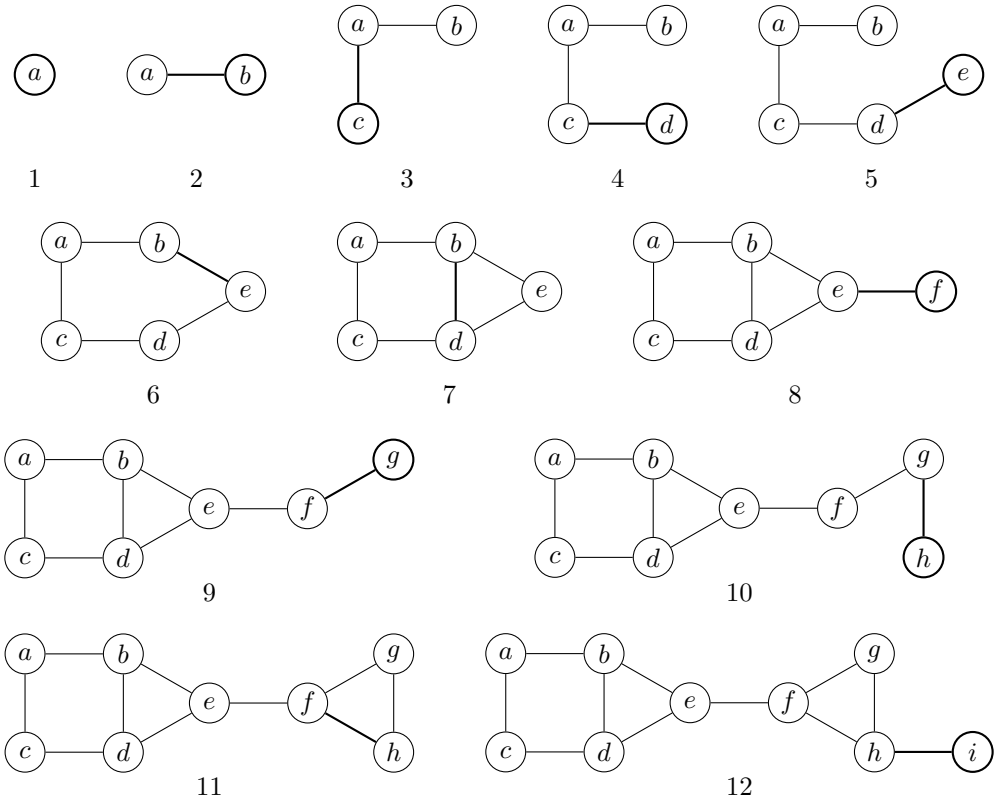


Figure 1.20: Drawing a connected planar graph

If a planar graph is not connected, we can make it connected by adding edges between disconnected parts until the graph is connected, without introducing new vertices and faces (the new edges will have the same face on both sides). Since the resulting graph satisfies Euler's formula, and we only added edges, we have:

Corollay 5 *Any planar graph with $V \geq 1$ vertices, E edges, and F faces satisfies $F+V-E \geq 2$.*

Note the above statements are true even for the case where there are parallel edges and self-loops. Furthermore, a consequence is all embeddings of a planar given graph will have the same number of faces.

Exercise 1.17 In this exercise we show that in any planar graphs with no parallel edges and self-loops, the number of edges is at most linear in the number of vertices.

(a) Prove that any connected planar graph with at least three vertices and without parallel edges and self-loops satisfies $E \leq 3V - 6$. *Hint.* The boundary of each face consists of at least three edges, and each edge is adjacent to at most two faces.

(b) Prove that any planar graph without parallel edges and self-loops satisfies $E \leq 3V$. *Hint.* Consider each connected component of the graph independently.

◁

1.10 Invariants

Invariants are statements that are true throughout a computation. In the following we try to give a general definition, although we will primarily apply invariants in the context of loop invariants. Let $S_0 \to S_1 \to S_2 \to \cdots$ be *states* during a finite or infinite computation. An example of a state could be the content of the variables of a program and "$\to$" the execution of an iteration

of a **while**-loop, transforming the state of the variables before the execution of the loop, to the state after the execution of the loop. Our goal is to prove that all states in the sequence of states satisfy some invariant I, i.e. $I(S_i)$ is true for all $i \geq 0$. Here I maps each possible state to true/false values.

Invariance principle
Consider a finite or infinite sequence of states

$$S_0 \to S_1 \to S_2 \to \cdots \to S_n \to S_{n+1} \to \cdots$$

The invariance principle states that to prove all states in the sequence to satisfy invariant I, it is sufficient to prove

Base case: $I(S_0)$ is true.

Induction step: If $S \to S'$ then $I(S) \Rightarrow I(S')$. Here $I(S)$ is the induction hypothesis.

Consider the below algorithm that 10 times increments a variable s with a random value in the range 2 to 5. It might be obvious that at the end the variable s has a value between 20 and 50, but let us try to reason about it using the invariance principle. When applying the invariance principle to a loop, we call it a *loop invariant*.

Algorithm RandomIncrements
```
1  i = 0
2  s = 0
3  # Invariant: 0 ≤ i ≤ 10 and 2i ≤ s ≤ 5i
4  while i < 10 do
5      r = random integer from {2, 3, 4, 5}
6      s = s + r
7      i = i + 1
```

For a state we only consider the variables i and s. Let S_0 be the state when we reach the **while**-loop from the preceding code and S_i the state after the ith iteration of the while loop. The first simple invariant is $I(S) : i \geq 0$. To see this we have the base case when we reach the loop the first time, where $i = 0$, i.e. the invariant is satisfied for S_0. For the induction step we assume $S \to S'$, i.e. the loop is executed, and $I(S)$ is true before the execution of the loop (induction hypothesis). Let i and s denote the values before the execution of the loop, and i' and s' the values after the execution of the loop. Since $i' = i + 1$ we have $i' \stackrel{\text{def}}{=} i + 1 \geq i \stackrel{\text{i.h.}}{\geq} 0$, i.e. $I(S')$ is true and therefor the induction step is true. It follows that in any execution of the program, $i \geq 0$ is an invariant for the **while**-loop.

Typically we use invariants to express relationships between the different components of a state. In the previous example we can prove that $2i \leq s \leq 5i$ will be a loop invariant for any execution of RandomIncrement: In S_0 we have $s = 0$ and $i = 0$, i.e. $2i \leq s \leq 5i$. For the induction step we have $2i \leq s \leq 5i$, and an execution of the loop results in $i' = i+1$ and $s' = s+r$ for some random r, where $2 \leq r \leq 5$. It follows that $s' \stackrel{\text{def}}{=} s+r \stackrel{\text{def}}{\leq} s+5 \stackrel{\text{i.h.}}{\leq} 5i+5 = 5(i+1) \stackrel{\text{def}}{=} 5i'$. Similarly we can prove $s' \geq 2i'$. It follows that $2i \leq s \leq 5i$ is an invariant for any execution of the loop.

A tricky detail is that the invariant should hold after each execution of the **while**-loop, in particular after the last iteration. The statement $i \leq 10$ is a valid invariant, if we assume i can only take integer values: For S_0 we have $i = 0$, i.e. $i \leq 10$ is satisfied. Assume the loop is executed in a state $S \to S'$. Since we execute the loop, we know the condition of the loop is true, i.e. $i < 10$. Since i is an integer, this implies $i \leq 9$. We have $i' = i+1 \leq 9+1 = 10$, i.e. invariant holds for the induction step. On the other hand, the statement $i < 10$ is not an invariant, since it is not satisfied after the last iteration of the loop.

To be able to conclude that $20 \leq s \leq 50$ when the loop terminates, we can argue the following

is an invariant for the loop:

$$i \text{ is an integer } \wedge\ 0 \leq i \leq 10\ \wedge\ 2i \leq s \leq 5i\ .$$

Having done so, we can conclude that when the loop terminates, the condition $i < 10$ is false, i.e. $i \geq 10$. Together with the invariant $i \leq 10$, we conclude $i = 10$. Plugging $i = 10$ into the invariant $2i \leq s \leq 5i$, gives us the bound $20 \leq s \leq 50$ for the final state.

Quiz 6 check

State for each statements if it is a valid invariant for RandomIncrements.

	Yes	No
$i \geq -1$		
$i \leq 11$		
$i \geq 7$		
$s \leq 10i$		
$s > i$		

Exercise 1.18 Prove that $s = i^2$ is a valid loop invariant for algorithm Square. Prove that at the end of the execution of the algorithm $s = n^2$.

Algorithm Square(n)
Input Integer $n \geq 1$
Output $s = n^2$

```
1  i = 1
2  s = 1
3  # Invariant: s = i^2
4  while i < n do
5      s = s + 2 · i + 1
6      i = i + 1
```

◁

Exercise 1.19 Prove that the algorithm Squared(n) computes $s = n^2$. Find an appropriate loop invariant I, and use this to conclude that $s = n^2$ when the algorithm terminates.

Algorithm Squared(n)
Input Integer $n \geq 0$
Output $s = n^2$

```
1  i = 0
2  s = 0
3  # Invariant: I
4  while i < n do
5      i = i + 1
6      s = s + i
7  s = s + s − n
```

◁

1.11 Fast integer division*

In this section we show how to perform the integer division $\lfloor N/D \rfloor$, where N is the *numerator* and D is the *denominator*. The goal is to achieve that the number of bit operations used is on the same order as for a single multiplication. This correlation between multiplication and division was first observed by Cook in 1966 [4] (see textbook by Aho, Hopcroft and Ullman *et al.* [1, Chapter 8.2]). A fast integer multiplication algorithm will imply an equally fast integer division

algorithm (up to a constant factor), e.g. when using Karatsuba's multiplication algorithm using order $n^{\log_2 3} = n^{1.58\cdots}$ bit operations. This is important when performing calculations on numbers with many digits, say millions of digits. The basic idea is to apply the Newton-Raphson root finding algorithm to an appropriate function to compute the fraction $1/D$ as a positional binary number, and then to multiply this result with N. Here $1/D$ is not an integer, so along the way we need to consider how to represent the approximation of fractional numbers using binary positional numbers.

1.11.1 Fractions as binary numbers

For positional decimal numbers the ith digit after the point represents $1/10^i$, i.e. $0.1 = 10^{-1}$, $0.01 = 10^{-2}$, $0.001 = 10^{-3}$, ... An example is

$$123.\underline{456}_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + \underline{4} \cdot 10^{-1} + \underline{5} \cdot 10^{-2} + \underline{6} \cdot 10^{-3}$$

For positional binary numbers with a point we have the same interpretation, except that the base is 2.

$$100.\underline{101}_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + \underline{1} \cdot 2^{-1} + \underline{0} \cdot 2^{-2} + \underline{1} \cdot 2^{-3} = 4 + 0.5 + 0.125 = 4.625$$

The general value of a binary number with fractions is

$$b_n \cdots b_1 b_0 \,.\, b_{-1} b_{-2} \cdots b_{-m} = \sum_{i=-m}^{n} b_i \cdot 2^i \ .$$

As with decimal numbers, where we require an infinite number of digits to represent some values, e.g. $1/3 = 0.3333\ldots$, the same applies to binary numbers. In particular we cannot represent $1/10$ exactly as a positional binary number with a finite number of digits

$$\frac{1}{10} = 0.1_{10} = 0.0001100110011001100110011001100110011\ldots_2$$

In practice for numbers stored as "float" or "double" on a computer only the most significant bits are stored, i.e. only approximations of numbers are stored (typically only the 24 or 53 most significant bits are stored). We end up with strange results as

$$0.3 - 0.2 - 0.1 = -2.7755575615628914 \cdot 10^{-17} = -2^{-55}$$

when using 64 bit IEEE 754 floats.

1.11.2 Computing $1/D$ using Newton-Raphson

We start by considering the Newton-Raphson algorithm to compute $1/D$, for $D \in \left[\frac{1}{2}, 1\right[$, with an arbitrary precision (using real numbers, or "floats" when executed on a computer). The magic is the following: Computing $1/D$ is equivalent to computing the root of the function

$$f(x) = D - 1/x \ .$$

Clearly, $f(1/D) = D - 1/(1/D) = D - D = 0$, i.e. $1/D$ is a root of f (and the only root). Furthermore, for $D \in \left[\frac{1}{2}, 1\right[$ we have $1/D \in \,]1, 2]$.

The Newton-Raphson algorithm repeatedly finds better-and-better approximations $x_0 < x_1 < x_2 < \cdots$ to $1/D$, where each $x_i < 1/D$. We will just set $x_0 = 1$ (other initial approximations are discussed below). Given an x_i, the next x_{i+1} is defined by the intersection of the x-axis with the tangent through $(x_i, f(x_i))$. See Figure 1.21. Since the slope of the tangent at $(x, f(x))$ is $f'(x) = 1/x^2$, we have

$$x_{i+1} = x_i - f(x_i)/f'(x_i) = x_i - (D - 1/x_i) \cdot x_i^2 = 2 \cdot x_i - D \cdot x_i^2 = (2 - D \cdot x_i) \cdot x_i \ .$$

The beauty is that x_{i+1} can be computed from x_i only using two multiplications and one subtraction (of real numbers). In Figure 1.22 the pseudocode is given.

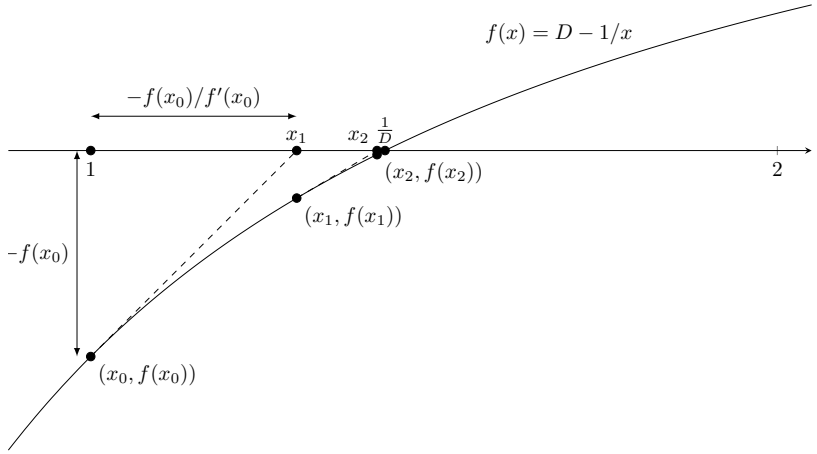


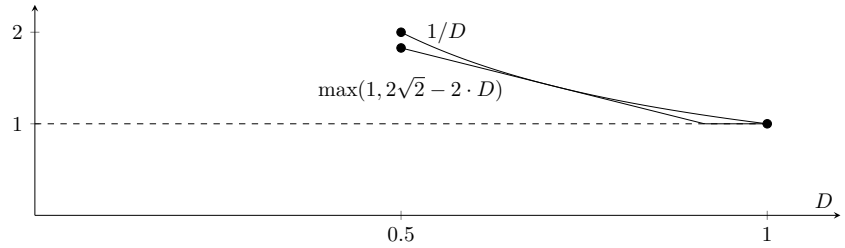
Figure 1.21: Newton-Raphson root finding for $f(x) = D - 1/x$ when $D \in [\frac{1}{2}, 1[$; above $D = 0.7$

Algorithm NEWTON-RAPHSON-RECIPROCAL(D)
Input Real value $D \in [\frac{1}{2}, 1[$
Output Real value $x \approx 1/D$

```
1  last = 0
2  x = 1
3  # Loop only terminates if fractional values have limited precision
4  while last < x do
5      last = x
6      x = (2 − D · x) · x
```

Figure 1.22: Computation of $1/D$

Although, mathematically the x_is form an infinite strictly increasing sequence of numbers $< 1/D$ (since f is a concave function with $f''(x) = \frac{-2}{x^3} < 0$ for $x \geq 1$), on a computer the limited precision caused by the number of digits represented implies that eventually $x_{i+1} = x_i$, and the computation terminates. For 64-bit floats (IEEE 754), this algorithm terminates amazingly fast. Table 1.1 shows that 5–6 iterations are sufficient for $D \in \{0.5, 0.7, 0.9\}$. The number of iterations can be reduced if we choose the initial approximation x_0 closer to $1/D$. E.g. Table 1.2 shows the reduced number of iterations if $x_0 = \max(1, 2\sqrt{2} - 2 \cdot D) \leq 1/D$ (note that $2\sqrt{2} \approx 2.8284$ is just a constant).



In fact, the Newton-Raphson algorithm also works if the initial approximation $x_0 > 1/D$. Since f is concave, the next approximation $x_1 < 1/D$, and subsequently $x_1 < x_2 < x_3 < \cdots < 1/D$. This allows for an even smaller initial error. In the following, we stick to $x_0 = 1$, ensuring that $x_0 < x_1 < x_2 < \cdots$ is an increasing sequence of lower bound approximations for $1/D$.

Let us consider the *error* during the computation. The error ε_i for the approximation x_i is defined as $\varepsilon_i = 1/D - x_i$, i.e. how far x_i is from the correct result $1/D$. For the initial $x_0 = 1$ we have $\varepsilon_0 = 1/D - 1$. For the subsequent errors we have

$$\varepsilon_{i+1} \stackrel{\text{def}}{=} \frac{1}{D} - x_{i+1} \stackrel{\text{def}}{=} \frac{1}{D} - (2 \cdot x_i - D \cdot x_i^2) = D\left(\frac{1}{D^2} - \frac{2 \cdot x_i}{D} + x_i^2\right) = D\left(\frac{1}{D} - x_i\right)^2 \stackrel{\text{def}}{=} D \cdot \varepsilon_i^2 \,.$$

D	0.5	0.7	0.9
x_0	1.0	1.0	1.0
x_1	1.5	1.3	1.1
x_2	1.875	1.4170000000000003	1.111
x_3	1.9921875	1.4284777000000002	1.1111111
x_4	1.999969482421875	1.428571422421897	1.111111111111111
x_5	1.9999999995343387	1.4285714285714286	1.1111111111111112
x_6	2.0		

Table 1.1: Convergence of $1/D$ computations (using 64 bit floats), $x_0 = 1.0$

D	0.5	0.7	0.9
x_0	1.8283999999999998	1.4284	1.0283999999999998
x_1	1.9852767199999999	1.428571408	1.1049540959999997
x_2	1.9998916125130208	1.4285714285714284	1.1110769931595406
x_3	1.9999999941260767	1.4285714285714286	1.11111111006348
x_4	2.0		1.1111111111111112

Table 1.2: Convergence of $1/D$ computations (using 64 bit floats), $x_0 = \max(1, 2\sqrt{2} - 2 \cdot D)$

Since $D < 1$, it follows that if $\varepsilon_i \leq 10^{-k}$ then $\varepsilon_{i+1} < \varepsilon_i^2 \leq 10^{-2k}$, i.e. the number of correct digits after the point doubles by each iteration. This explains the fast termination. Specifically, we have

$$\varepsilon_1 = D \cdot \varepsilon_0^2 \quad \varepsilon_2 = D \cdot \varepsilon_1^2 = D(D \cdot \varepsilon_0^2)^2 = D^3 \cdot \varepsilon_0^4 \quad \varepsilon_3 = D \cdot \varepsilon_2^2 = D(D^3 \cdot \varepsilon_0^4)^2 = D^7 \cdot \varepsilon_0^8 \quad \text{e.t.c.}$$

The next exercise asks you to prove that $\varepsilon_i = D^{2^i-1} \cdot \varepsilon_0^{2^i}$.

Exercise 1.20 Prove by induction that $\varepsilon_i = D^{2^i-1} \cdot \varepsilon_0^{2^i}$. ◁

Since $\varepsilon_0 = 1/D - x_0 = 1/D - 1$, we have

$$\varepsilon_i = D^{2^i-1} \cdot \left(\frac{1}{D} - 1\right)^{2^i} = \frac{\left(D\left(\frac{1}{D} - 1\right)\right)^{2^i}}{D} = \frac{(1-D)^{2^i}}{D} ,$$

and using $D \in \left[\frac{1}{2}, 1\right[$, we get the following upper bound

$$\varepsilon_i \leq \frac{\left(1 - \frac{1}{2}\right)^{2^i}}{\frac{1}{2}} = 2 \cdot \left(\frac{1}{2}\right)^{2^i} = 2^{1-2^i} .$$

Example: For 6 iterations, the error $\varepsilon_6 \leq 2^{-63} \approx 1.08 \cdot 10^{-19}$. This is consistent with the picture we see in Table 1.1 that six iterations are sufficient.

1.11.3 Integer division

We now turn to the integer division $\lfloor N/D \rfloor$, where $N \geq 0$ and $D \geq 1$ are integers. The task is to compute two unique integers, the *quotient* $q = \lfloor N/D \rfloor$ and the *remainder* r, satisfying $N = q \cdot D + r$ and $0 \leq r < D$.

Before describing the algorithm we introduce some simple operations on binary numbers. For a non-negative integer w, we let $\text{bits}(w)$ denote the number of bits in the binary representation of w, i.e. $2^{\text{bits}(w)-1} \leq w < 2^{\text{bits}(w)}$ for $w > 0$ and $\text{bits}(0) = 0$. For a positive integer w, we let $\text{lsb}(w)$ denote the position of the least significant bit equal to 1 in the binary representation of w. Example: $\text{bits}(\underline{110110}_2) = 6$ and $\text{lsb}(10110\underline{1}000_2) = 3$. Two further useful operations are *left shift* and *right shift*. The left shift operator $w \texttt{<<} i$ appends i zeros to the right of the binary representation of w, whereas the right shift operator $w \texttt{>>} i$ drops the i rightmost bits of the binary representation of w. Example $\underline{11010}_2 \texttt{<<} 3 = \underline{11010}000_2$ and $\underline{11010}_2 \texttt{>>} 2 = \underline{110}_2$. When w and i are non-negative integers, the result of the two shift operators are $w \texttt{<<} i = w \cdot 2^i$ and $w \texttt{>>} i = \lfloor w/2^i \rfloor$. Many programming languages support the operators << and >> natively. Furthermore for a real

value x and a nonnegative integer d, we will use the notation $\lfloor x \rfloor_d = \lfloor x \cdot 2^d \rfloor / 2^d$, i.e. x rounded down to the first d digits after the point. In particular $\lfloor x \rfloor_0 = \lfloor x \rfloor$. E.g. $\lfloor 10.\underline{101}0011_2 \rfloor_3 = 10.\underline{101}_2$. Similarly $\lceil x \rceil_d = \lceil x \cdot 2^d \rceil / 2^d$, e.g. $\lceil 1.\underline{1101}00101_2 \rceil_4 = 1.\underline{1110}_2$. Note that if there is a digit=1 somewhere to the right of the d first digits, then $\lceil x \rceil_d = \lfloor x \rfloor_d + \frac{1}{2^d}$.

Quiz 7 check

The value of 13 << 2.

0 13 26 39 52 1300

The value of 13 >> 2.

0 1 2 3 3.25 4 6 6.5

In the following we let $n = \text{bits}(N)$ and $m = \text{bits}(D)$ denote the number of digits in the binary representations of N and D, in particular $2^{m-1} \le D < 2^m$. We let $\tilde{D} = D/2^m$, i.e. $\tilde{D} \in \left[\frac{1}{2}, 1\right[$. The basic idea is to apply Newton-Raphson to compute an approximation x to $1/\tilde{D}$, where $1/\tilde{D} = x + \varepsilon$ for a sufficiently small $\varepsilon \ge 0$. Here x will be a binary positional number with a limited, but sufficient, number of digits. To avoid computing on unessential digits, during the iterations of the Newton-Raphson algorithm, we will increase exponentially the number of digits in the involved computations. The thereby introduced increase in the errors is marginal. The multiplications in the last iteration will dominate the running time of all iterations.

The goal is to achieve $\varepsilon \le 2^{m-n}$. Then, since $N < 2^n$, we have $x \cdot N/2^m = (1/\tilde{D} - \varepsilon) \cdot N/2^m = (1/(D/2^m) - \varepsilon) \cdot N/2^m = N/D - \varepsilon \cdot N/2^m > N/D - \varepsilon \cdot 2^n/2^m \ge N/D - 1$, i.e. $N/D - 1 \le x \cdot N/2^m \le N/D$ and $\lfloor N/D \rfloor - 1 \le \lfloor x \cdot N/2^m \rfloor \le \lfloor N/D \rfloor$. Since $q = \lfloor N/D \rfloor$, it follows that

$$q = \lfloor x \cdot N/2^m \rfloor \quad \text{or} \quad q = \lfloor x \cdot N/2^m \rfloor + 1 \; . \tag{1.10}$$

We have seen that the core of the Newton-Raphson algorithm is the update step

$$x_{i+1} = (2 - \tilde{D} \cdot x_i) \cdot x_i \; . \tag{1.11}$$

We will only compute an approximation of each x_i with d_i digits after the point, where $0 = d_0 \le d_1 \le d_2 \le \cdots$ is a non-decreasing sequence. Instead of using all digits of $\tilde{D}$ in the computation of x_{i+1}, we will only use the d_{i+1} most significant digits $D_{i+1} = \lceil \tilde{D} \rceil_{d_{i+1}}$. The ceiling ensures $1 \ge D_1 \ge D_2 \ge \cdots \ge \tilde{D}$, i.e. $1 \le \frac{1}{D_1} \le \frac{1}{D_2} \le \cdots \le \frac{1}{\tilde{D}}$. More precisely instead of computing (1.11) we will compute

$$x_{i+1} = \lfloor (2 - D_{i+1} \cdot x_i) \cdot x_i \rfloor_{d_{i+1}} \; . \tag{1.12}$$

To avoid working with fractional binary numbers, in the pseudocode in Figure 1.23, instead of computing x_i, we compute integers $y_i = x_i \cdot 2^{d_i}$. The update (1.12) becomes

$$y_{i+1} = \left(\left((2 \mathbin{<<} (d_i + d_{i+1})) - \left(D_{i+1} \cdot 2^{d_{i+1}} \right) \cdot y_i \right) \cdot y_i \right) \mathbin{>>} 2d_i \; , \tag{1.13}$$

where

$$D_{i+1} \cdot 2^{d_{i+1}} = ((D \mathbin{<<} d_{i+1}) \mathbin{>>} m) + \begin{cases} 0 & \text{if } \text{lsb}(D) \ge m - d_{i+1} \\ 1 & \text{if } \text{lsb}(D) < m - d_{i+1} \; . \end{cases}$$

To bound the errors, let $\delta_i = 2^{-d_i}$, i.e. $D_i - \tilde{D} < \delta_i$. Since $x_0 = 1$, we have $\varepsilon_0 = \frac{1}{\tilde{D}} - 1$. For

Algorithm INTEGER-DIVISION(N/D)
Input Integers $N \geq 0$ and $D \geq 1$
Output Integers q and r, where $N = D \cdot q + r$ and $0 \leq r < D$

```
1  y = 1    # x_0 = 1
2  d = 0    # x_0 has no digits after the point
3  k = 0    # Error ε ≤ 2^{-k}
4  while k < bits(N) − bits(D) do
5      d_last = d    # d_i
6      if d = 0 then
7          d = 6    # d_1 = 6, k_1 = 0
8      else if k = 0 then
9          k = 2    # d_2 = 6, k_2 = 2
10     else
11         k = k + k − 1    # k_{i+1} = 2 · k_i − 1
12         d = k + 4        # d_{i+1} = k_{i+1} + 4
13     D̄ = (D << d) >> bits(D)    # D_{i+1} = d_{i+1} most significant bits of D
14     if d < bits(D) − lsb(D) then
15         D̄ = D̄ + 1    # rounded up if a pruned digit equals 1
16     y = (((2 << (d + d_last)) − D̄ · y) · y) >> 2d_last    # y_{i+1} = x_{i+1} · 2^{d_{i+1}}
17 q = (N · y) >> (d + bits(D))    # q = ⌊N · x/2^{bits(D)}⌋
18 r = N − D · q
19 if r ≥ D then
20     q = q + 1    # q = ⌊N · x/2^{bits(D)}⌋ + 1
21     r = r − D
22 return (q, r)
```

Figure 1.23: Integer division using Newton-Raphson iteration

N	=	347682327897	=	$101000011110011011110110100110101011001_2$
D	=	234707	=	111001010011010011_2
q	=	1481346	=	101101001101010000010_2
r	=	52275	=	1100110000110011_2

i	k_i	d_i	$y_i = x_i \cdot 2^{d_i}$	$D_i \cdot 2^{d_i}$
0	0	0	1_2	$D = 111001010011010011_2$
1	0	6	1000110_2	111010_2
2	2	6	1000110_2	111010_2
3	3	7	10001110_2	1110011_2
4	5	9	1000111011_2	111001011_2
5	9	13	10001110111101_2	1110010100111_2
6	17	21	1000111011110110100010_2	1110010100110100110000_2
7	33	37	$100011101111011010001011011001101001 10_2$	$11100101001101001100000000000000000000_2$

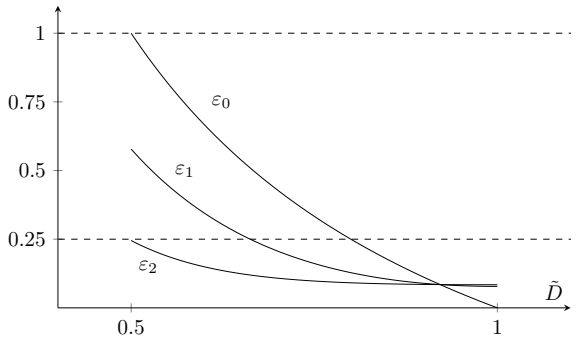
Figure 1.24: Computing N/D

the subsequent iterations we have

$$\begin{aligned}
\varepsilon_{i+1} &\stackrel{\text{def}}{=} \frac{1}{\tilde{D}} - x_{i+1} \\
&\stackrel{\text{def}}{=} \frac{1}{\tilde{D}} - \lfloor (2 - D_{i+1} \cdot x_i) \cdot x_i \rfloor_{d_{i+1}} \\
&< \frac{1}{\tilde{D}} - (2 - D_{i+1} \cdot x_i) \cdot x_i + \delta_{i+1} \\
&< \frac{1}{\tilde{D}} - 2 \cdot x_i + \tilde{D} \cdot x_i \cdot x_i + 4\delta_{i+1} + \delta_{i+1} && \text{since } x_i \leq 2 \text{ and } D_{i+1} - \tilde{D} < \delta_{i+1} \\
&= \tilde{D} \left(\left(\frac{1}{\tilde{D}} \right)^2 - \frac{2 \cdot x_i}{\tilde{D}} + x_i^2 \right) + 5\delta_{i+1} \\
&= \tilde{D} \left(\frac{1}{\tilde{D}} - x_i \right)^2 + 5\delta_{i+1} \\
&\stackrel{\text{def}}{=} \tilde{D} \cdot \varepsilon_i^2 + 5\delta_{i+1}
\end{aligned}$$

If we let $d_1 = d_2 = 6$, i.e. $\delta_1 = \delta = 2^{-6} = \frac{1}{64}$, then for all $\tilde{D} \in \left[\frac{1}{2}, 1\right[$ we get the error bounds

$$\begin{aligned}
\varepsilon_0 &= \frac{1}{\tilde{D}} - 1 \leq 1 && \text{since } x_0 = 1 \\
\varepsilon_1 &< \tilde{D} \left(\frac{1}{\tilde{D}} - 1 \right)^2 + 5\delta_1 < 0.58 \\
\varepsilon_2 &< \tilde{D} \left(\tilde{D} \left(\frac{1}{\tilde{D}} - 1 \right)^2 + 5\delta_1 \right)^2 + 5\delta_2 < 0.25
\end{aligned}$$



For $i \geq 2$ we will aim at an error bound $\varepsilon_i \leq 2^{-k_i}$, where $k_i = 2^{i-2} + 1$. We achieve this by setting $d_i = k_i + 4$. We first observe that for $i = 2$, then $k_i = 2^{2-2} + 1 = 2^0 + 1 = 1 + 1 = 2$ and $d_2 = k_2 + 4 = 2 + 4 = 6$ and $2^{-k_i} = \frac{1}{4} > \varepsilon_2$ is consistent with the above. To compute the k_i values we observe

$$k_{i+1} = 2^{(i+1)-2} + 1 = 2 \cdot 2^{i-2} + 1 = 2 \cdot (2^{i-2} + 1) - 1 = 2k_i - 1 \,.$$

Finally, for $i \geq 2$, we get the error bound $\varepsilon_i < 2^{-k_i}$ by induction

$$\begin{aligned}
\varepsilon_{i+1} &< \tilde{D} \cdot \varepsilon_i^2 + 5\delta_{i+1} \\
&< \varepsilon_i^2 + 8\delta_{i+1} && \text{since } \tilde{D} \leq 1 \\
&\stackrel{\text{i.h.}}{\leq} \left(2^{-k_i}\right)^2 + 2^{-(k_{i+1}+4)+3} && \text{since } \delta_{i+1} = 2^{-d_{i+1}} = 2^{-(k_{i+1}+4)} \\
&= 2^{-2k_i} + 2^{-(2k_i-1+4)+3} && \text{since } k_{i+1} = 2k_i - 1 \\
&= 2 \cdot 2^{-2k_i} \\
&= 2^{-(2k_i-1)} \\
&= 2^{-k_{i+1}} \,.
\end{aligned}$$

In the beginning we observed that we can terminate when $\varepsilon_i \leq 2^{m-n}$. Since $\varepsilon_i \leq 2^{-k_i}$, we can terminate when $-k_i \leq m-n$, i.e. when $k_i = 2^{i-2}+1 \geq n-m$, or equivalently after $i \geq 2+\log_2(n-m-1)$ iterations.

To bound the total number of bit operations, we observe that in the ith iteration we work on integers with at most $1+d_i$ digits. Since all operations (additions, subtractions, shiftings) can be performed with order d_i bit operations, the bottleneck becomes the two multiplications in (1.13). Assuming these can be performed in time $T_{\text{mult}}(d_i)$, the total time becomes order

$$\begin{aligned}
\sum_i T_{\text{mult}}(d_i) &= \sum_i T_{\text{mult}}\left(2^{i-2}+1+4\right) \\
&\approx \sum_i T_{\text{mult}}\left(2^i\right) \\
&\approx \sum_i T_{\text{mult}}\left(\frac{n-m}{2^i}\right) && \text{last } d_i \approx n-m \\
&\leq \sum_i \frac{1}{2^i} \cdot T_{\text{mult}}\left(n-m\right) && (*) \\
&\approx T_{\text{mult}}\left(n-m\right) \cdot \sum_i \frac{1}{2^i} \\
&\approx T_{\text{mult}}\left(n-m\right)\ , && \text{since } \sum_{i=0}^{\infty} \frac{1}{2^i} = 2
\end{aligned}$$

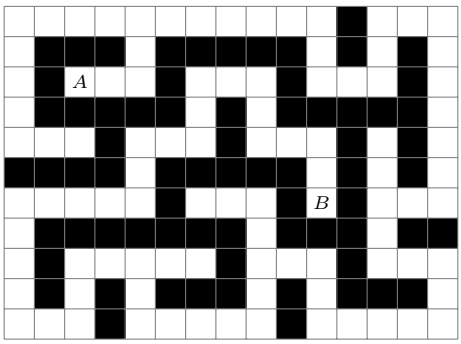
where we in (*) use the assumption $T_{\text{mult}}(c \cdot n) \geq c \cdot T_{\text{mult}}(n)$, for $c \geq 1$, i.e. multiplying twice as long numbers takes at least twice as long time.

In the final steps of the computation of q and r, we perform two additional multiplications. We conclude that if two integers with n bits can be multiplied using $T_{\text{mult}}(n)$ bit operations, then we can also divide two integers with n bits using order $T_{\text{mult}}(n)$ bit operations. The pseudocode in Figure 1.23 summarizes the previous discussion and Figure 1.24 gives an example of the computation in the loop.

1.12 Problems

The following Problems 1.1–1.7 illustrate questions that can be solved by general algorithmic techniques covered later in the course. For each problem the relevant technique is stated, but you should try to solve the problems without looking into the relevant material.

Problem 1.1 What is the *shortest path* from A to B in the below maze? Black cells are blocked and the length of a path is the number of times you move from a cell to an adjacent unblocked cell. Describe the steps of your algorithm.



(The problem can be solved using the breadth first search algorithm, BFS). ◁

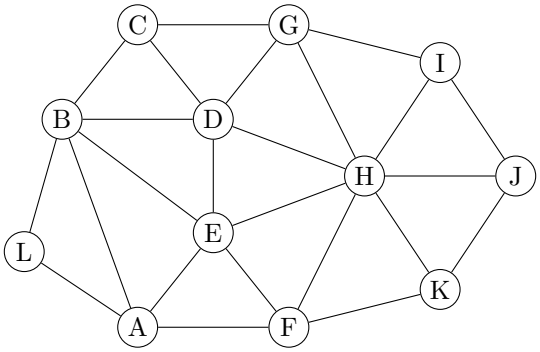
Problem 1.2 Find a *longest increasing subsequence* in the following sequence of distinct numbers:

30 83 73 80 59 63 41 78 68 82 53 31 22 74 6 36 99 57 43 60

Your task is to remove as few numbers as possible such that remaining numbers in the sequence appear in increasing order. Can you formulate your solution as a general algorithm? *Hint:* For each value, find the length of the longest increasing subsequences ending with this value. (The problem can be solved efficiently using the technique of dynamic programming).

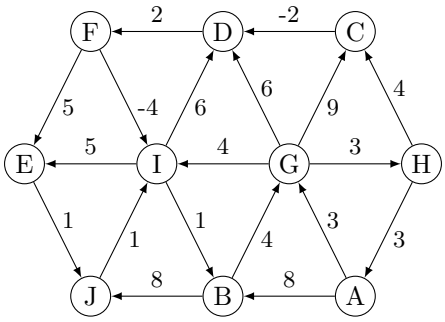
Can you use the insights in the solution to prove the following classic result by Paul Erdős and George Szekeres from 1935? *Any sequence of n distinct numbers contains an increasing or decreasing subsequence of length at least $\lceil \sqrt{n} \rceil$.* ◁

Problem 1.3 Find a *maximum independent set* of nodes in the below graph.



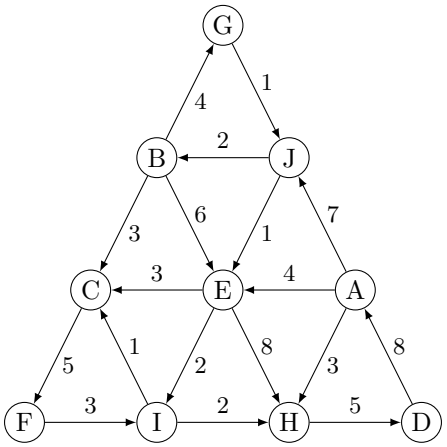
A *graph* consists of a set of *nodes*, here $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots, \mathrm{K}\}$, and a set of *edges*, where each edge connects a pair of nodes. Two nodes connected by an edge are said to be *adjacent*. An independent set is a subset of the nodes where no pair of nodes are adjacent The goal is to find a set of nodes of maximum cardinality that is an independent set. The set $\{\mathrm{A}, \mathrm{D}, \mathrm{J}\}$ is a set of independent nodes, since none of the possible pairs (A, D), (A, J) and (D, J) are connected by an edge in the graph. In fact this is a *maximal* solution, since you cannot add any of the remaining nodes, without introducing a pair of adjacent nodes. But there exist larger independent sets of nodes — your task is to find a largest. (This problem is known to be "NP-complete", i.e. "complete" for the complexity class **N**ondeterministic **P**olynomial time; in practice this means that the most efficient algorithms essentially try all possible subsets of nodes to solve the problem; see e.g. the text book by Fomin and Kratsch [7]). ◁

Problem 1.4 In the below *directed weighted graph* find a *shortest path* from A to J.



In a directed graph each edge has an orientation, e.g. the edge between A and G is oriented from A to G. If we consider edges indicating where we can walk, then we can go from A to G along the edge from A to G, but we cannot go in the opposite direction along the edge. A *path* from a node u to another node v is a sequence of directed edges that allows you to walk from u to v. In this problem each edge has an associated *weight* denoting the price to walk along the edge. The *length* of a path is the sum of the weights of the edges along the path. E.g. the path G→D→F→I has length $6 + 2 - 4 = 4$. Your task is to find a path from A to J with *minimum length*. Note that in the example some edges have *negative* weights. (This problem can be solved using Bellman-Ford's single source shortest path algorithm). ◁

Problem 1.5 Find two nodes u and v in the below graph, such that the distance from u to v is as large as possible. The distance from a node to another node is the length of the shortest path between the nodes (see Problem 1.4). If no path exists the distance is $+\infty$.



(This problem can be solved using Floyd-Warshall's or Dijkstra's shortest paths algorithms). ◁

Problem 1.6 You are given the below spreadsheet. Describe an order to compute the equations in the spreadsheet, i.e. the cells starting with "=". To compute an equation, the content of the cells that it depends on must already have been computed or be simple values.

	A	**B**	**C**	**D**	**E**	**F**	**G**
1	2	10	=B1	=A1∗B1	=D1	=B1/C4	=F1∗G4
2	4	12	=B2+C1	=A2∗B2	=D2+E1	=B2/C4	=F2∗G4
3	3	15	=B3+C2	=A3∗B3	=D3+E2	=B3/C4	=F3∗G4
4			=C3		=E3	80	=E4−F4

(This problem can be solved by topological sorting the directed graph of the dependencies). ◁

Problem 1.7 For the US presidential election in 2020 each state has the following electoral votes. The candidate with the most votes in a state receives all the electoral votes of the state. In total there are 538 electoral votes. Determine if it is possible that the election between two candidates ends in a tie, i.e. both candidates receives an equal number of electoral votes.

Alabama	9	Kentucky	8	North Dakota	3
Alaska	3	Louisiana	8	Ohio	18
Arizona	11	Maine	4	Oklahoma	7
Arkansas	6	Maryland	10	Oregon	7
California	55	Massachusetts	11	Pennsylvania	20
Colorado	9	Michigan	16	Rhode Island	4
Connecticut	7	Minnesota	10	South Carolina	9
District of Columbia	3	Mississippi	6	South Dakota	3
Delaware	3	Missouri	10	Tennessee	11
Florida	29	Montana	3	Texas	38
Georgia	16	Nebraska	5	Utah	6
Hawaii	4	Nevada	6	Vermont	3
Idaho	4	New Hampshire	4	Virginia	13
Illinois	20	New Jersey	14	Washington	12
Indiana	11	New Mexico	5	West Virginia	5
Iowa	6	New York	29	Wisconsin	10
Kansas	6	North Carolina	15	Wyoming	3

(This is an example of a partitioning problem that can be solved efficiently using the general technique of dynamic programming for integer values). ◁

Problem 1.8 (Searching an infinite sorted list *) Assume we have an infinite long increasing sequence of real values $x_1 < x_2 < x_3 < \cdots$ and we want to find the position of a real value $y > x_1$ in this list. More specifically, we want to find the index d where $x_{d-1} < y \leq x_d$,

i.e. $x_d = y$ if y if y is in the list, and otherwise x_d is the successor of y in the list. E.g. for the sequence 3 5 7 9 11 17 19 23 31 33 37 ... and $y = 11$ we have $d = 5$, since $y = d_5 = 11$, whereas for $y = 24$ we have $d = 9$, since $23 = d_8 < y \leq d_9 = 31$. We assume the sequence to be divergent, i.e. there always exists such a d.

Describe an algorithm to find the index d, where the number of comparisons performed between y and the x_is is a function of the final value of d. What is the most efficient algorithm (fewest number of comparisons performed) you can find? Argue that your algorithm finds the correct index d, such that either $y = x_d$ or $x_{d-1} < y < x_d$. Analyze your algorithm — how many comparisons does your algorithm perform as a function of d?

(This problem is intentionally open ended; it was studied by Bentley and Yao in 1976 [3] if you are interested in the research on the problem). ◁

Problem 1.9 (Rank lookup in two sorted lists*) Assume you are given two sorted lists $X = x_1, \ldots, x_n$ and $Y = y_1, \ldots, y_m$. Assume all elements are distinct. Given an index r, how fast (number of comparisons) can you find the rth smallest element in $X \cup Y$? We denote this the element of *rank* r in $X \cup Y$. State the worst-case number of comparisons as a function of n, m and/or r. *Example*: If $X = 3, 7, 12, 13, 27, 31, 42$, $Y = 4, 11, 17, 33, 37, 39, 51$, and $r = 8$, then 27 has rank 8 in $X \cup Y$. (Frederickson and Johnson [8] considered the problem for an arbitrary number of sorted lists). ◁

Problem 1.10 (Interpolation search) Assume you are going to search in a sorted array with n real numbers $x_1, \ldots, x_n$, where each number has been chosen uniformly at random in the interval $[0, 1]$, i.e. you can expect the numbers to be somehow evenly distributed in the interval. Assume you are going to search for a value $y \in [0, 1]$. Can you come up with an algorithm that exploits that the input is somehow evenly distributed? (You do not need to give a formal analysis of your algorithm; Yao and Yao [15] proved that a search can be performed in order expected $\log(\log n)$ comparisons). ◁

Chapter 2

Data Structures

2.1 Red-black trees

Red-black trees were introduced by Guibas and Sedgewick in 1978 [10], as a class of balanced binary search trees with logarithmic time operations. A *red-black tree* is a binary search tree, where each node is colored either *red* or *black*. A valid red-black tree must satisfy the following two invariants:

- Each red node must have a black parent, in particular the root must be black.
- The number of black nodes on all root-to-leaf paths is the same.

Here we assume a leaf is an external node not storing an element, and all internal nodes store an element. This ensures all internal nodes have two children. In an actual implementation a leaf would just be a **null** pointer. Figure 2.1 shows a valid red-black tree, where red nodes are shown as double circles and leaves as squares. All root-to-leaf paths have two black nodes.

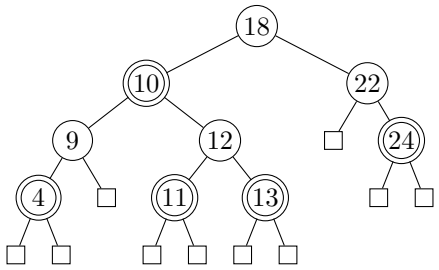
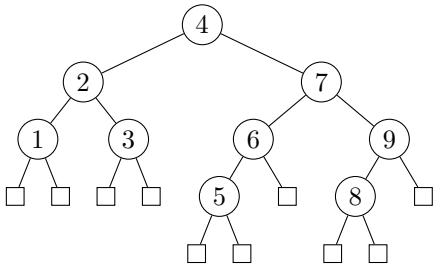


Figure 2.1: A red-black tree

Quiz 8 – Red-black tree invariants check

For each of the sets of nodes, state if the binary search tree is a valid red-black tree if exactly these nodes are colored red.



	Yes	No
2, 5, 6, 8, 9		
1, 3, 5, 7, 8		
4, 5, 8		
2, 5, 7, 8		
5, 8		

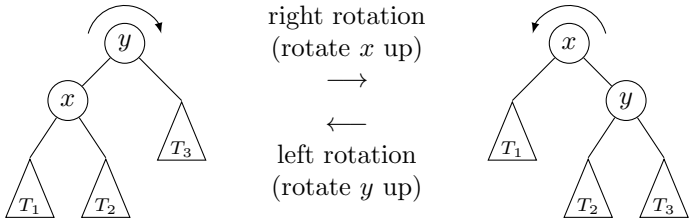


Figure 2.2: Rotations in a search tree

Not all binary search trees can be colored into a valid red-black tree. The following theorem captures the essential consequence of the red-black tree invariants.

Theorem 12 *A red-black tree with n elements has height* $\mathrm{O}(\log n)$.

Proof. Assume a highest leaf v in a red-black tree has depth d (where the root has depth zero). The root-to-leaf path to v contains at most d black internal nodes. By assumption, the topmost d levels of the tree do not contain a leaf, i.e. they are filled with $\sum_{i=0}^{d-1} 2^i = 2^d - 1$ internal nodes. We have $2^d - 1 \leq n$, i.e. $d \leq \log_2(n+1)$. For a deepest leaf u in the tree, the root-to-leaf path to u also contains at most d black nodes. Since each red node must have black parent, the path also contains at most d red nodes. It follows that the path to u contains at most $2d \leq 2\log_2(n+1)$ internal nodes. In Figure 2.1, v could be the left child of 22 with depth $d = 2$, and u any child of 4, 11 or 13. □

2.1.1 Insertions

Insertions into red-black trees is conceptually quite simple, although an implementation requires several symmetric cases to be handled explicitly, making an implementation less trivial. Here we will focus presenting the overall idea. The pseudocode for the insertion is shown in Figure 2.3, and Figure 2.4 visualizes the three types of transformations used to reestablish the red-black tree invariants. The result of inserting 15 into the red-black tree in Figure 2.1 is shown in Figure 2.5.

To insert an element e we first perform a standard top-down search tree search for the leaf x where e should be inserted. The leaf x is replaced by a red internal node storing e and with two leaves below. The new leaves have depth one larger than the depth of x, but by coloring x red the root-to-leaf paths to these leaves contain the same number of black nodes as before. The only problem with respect to the red-black invariants is that x might not have a black parent. The method FixInsert(x) solves this problem, by repeatedly applying the three transformations below until the problem is gone.

Case I is when x is the root, but x is red. Then we color x black. This increases by one the number of black nodes on all root-to-leaf paths.

The cases II and II apply when x is red, and its parent p is red. Then the parent of p'' of p must exist and be black (since x is the only red node that might not have a black parent).

Case II is when the sibling p' of p is a red node. In this case we color p' and p black and p'' red, and let $x = p''$ be the new red node that potentially has no black parent. This recoloring ensures all leaves below p'' still have the same number of black nodes on their root-to-leaf paths. Figure 2.5 from a) to b) illustrates this case.

Case III is when the sibling p' of p is a black node or a leaf. Then we restructure the three nodes x, p' and p'' into a perfect balanced three with three nodes left-to-right a, b, c, where a and c are red and b is black. Let T_1, T_2, T_3, T_4 be the four trees, left-to-right, that are rooted at the two children of x, the sibling of x, and the sibling p' of p. T_1 and T_2 become the children of a, and T_3 and T_4 become the children of c. The number of black nodes on root-to-leaf paths to leaves in T_1, T_2, T_3, T_4 remains unchanged (of x, p, p'' only p'' contributed a black node before the restructuring, and after only b does, since a and c are red). Figure 2.5 from b) to c) illustrates this case.

Cases I and III eliminate the invariant violations, whereas case II eliminates the invariant violation at node x but potentially creates a new invariant violation at p''. But the distance

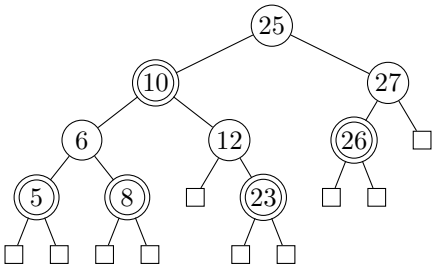
Algorithm RedBlackInsert(e)
1 perform a top-down search to identify the leaf x where e should be inserted
2 let x become a *red* internal node storing e with two leaves below
3 FixInsert(x)

Algorithm FixInsert(x)
Input x is a red node, potentially with a red parent
1 **if** x is the root **then**
2 color x *black* # *case I*
3 **else if** parent p of x is *red* **then**
4 **if** the sibling p' of p is a *red* node **then**
5 color p and p' *black*, and parent p'' of p *red* # *case II*
6 FixInsert(p'')
7 **else**
8 let a, b, c be x, p and the parent p'' of p in sorted order # *case III*
9 let T_1, T_2, T_3, T_4 be the subtrees left-to-right rooted at children of x, p and p'' (but not containing x)
10 let b be *black* and replacing p''
11 let a be *red* and left child of b, with left child T_1 and right child T_2
12 let c be *red* and right child of b, with left child T_3 and right child T_4

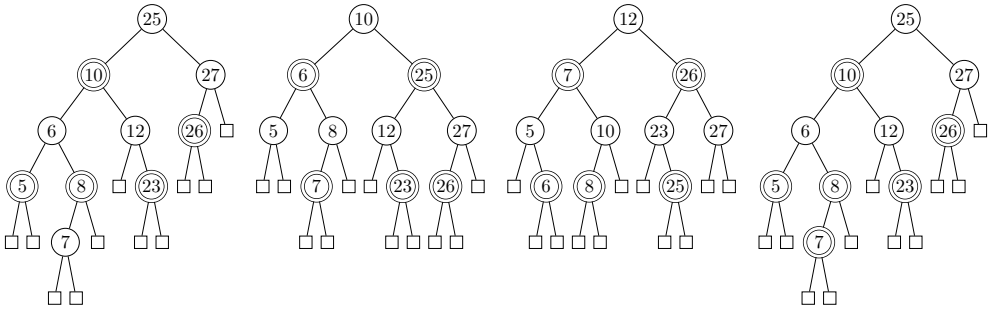
Figure 2.3: Red-black tree insertion

from the problematic node to the root decreases by two, i.e. case II needs to be applied at most $O(\log n)$ times by Theorem 12. It follows that insertions take $O(\log n)$ time. In an actual implementation, case III will restructure the tree by one or two *rotations*, followed by recoloring a few nodes. The rotations are drawn as circular arcs above the nodes in Figure 2.4. The two types of rotations, left and right, are shown in Figure 2.2.

Quiz 9 check



What is the resulting red-black tree after inserting 7 in the above red-black tree?



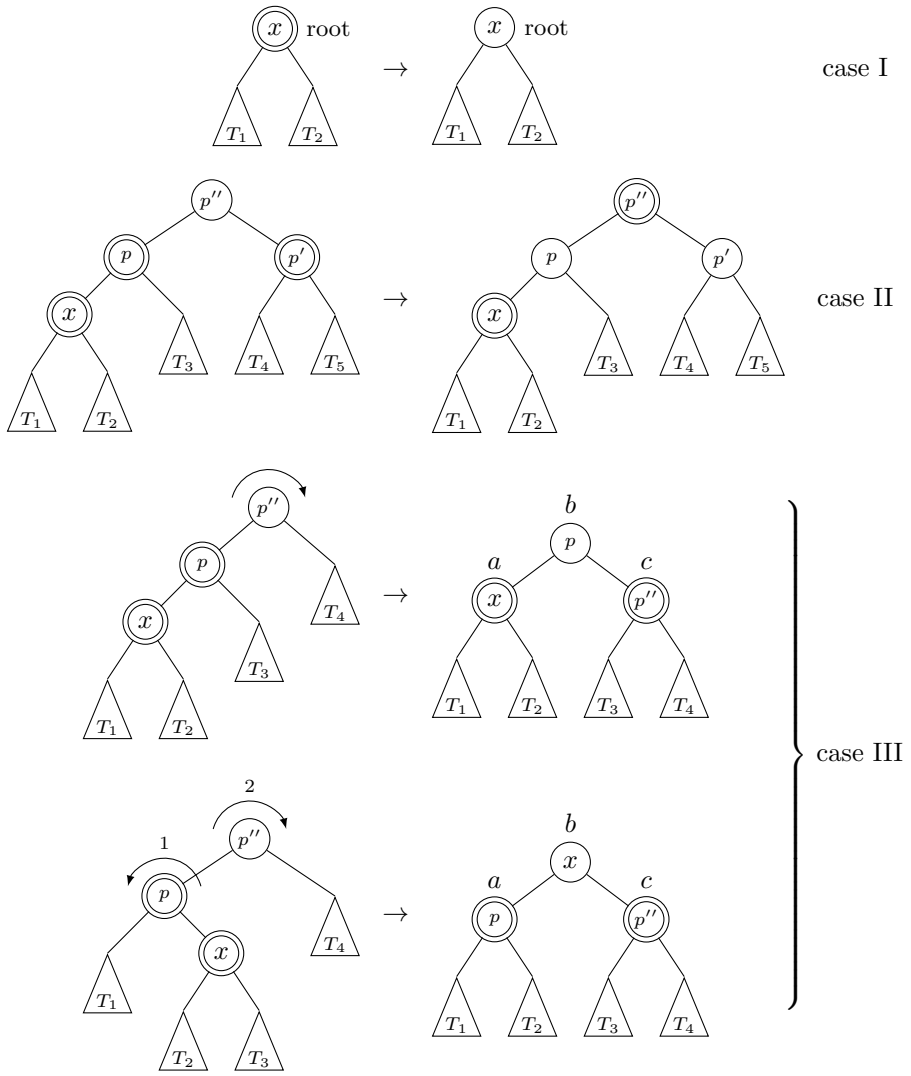


Figure 2.4: FixInsert(x) transformations (each subtree T_i has either a black root or is a leaf; case II shows one out of four symmetric cases, and case III shows two out of four symmetric cases)

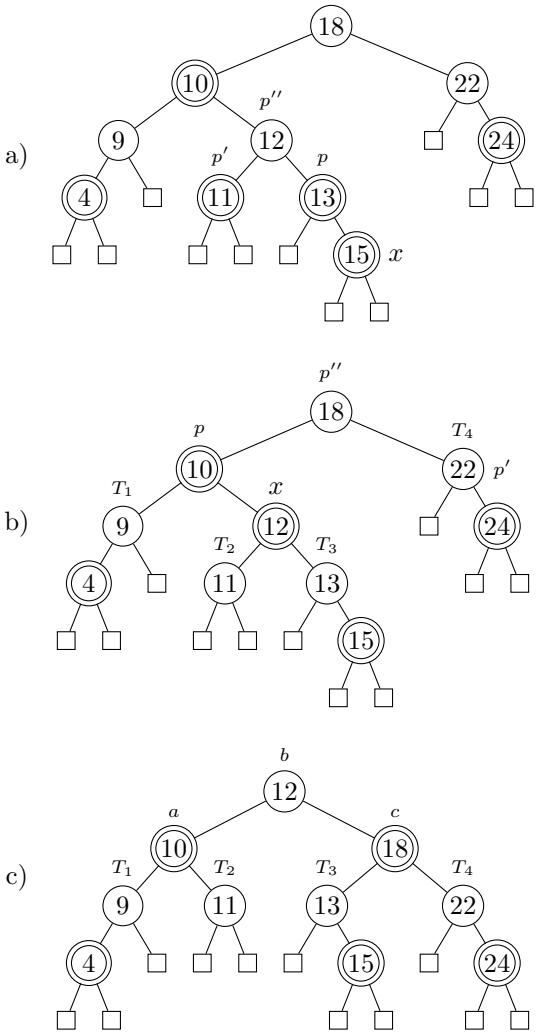


Figure 2.5: Red-black tree INSERT(15)

2.1.2 Deletions

To delete an element from a red-black tree, we first perform a standard top-down search tree search for the node x containing the element to be deleted. Similarly to the unbalanced tree case, if x has two children, first we swap x with its successor y to ensure that the node to be removed has at most one child. Since x is crucial to the red-black tree invariants if it is black, we cannot just delete it. If x is red, it is safe to remove x, but otherwise we need to restructure the tree before removing x. The nine cases (with symmetric cases omitted) are shown in Figure 2.6, where x is the node to be deleted. The pseudocode for the deletion algorithm can be found in Figure 2.7.

As already stated, if the node x is red, we can just delete it (case 1). If it is black, but its child is red, we can color the child red, and remove x (case 2). This will guarantee that the number of black nodes on all root-to-leaf paths to leaves below x remains unchanged. If x is still in the tree, it is now guaranteed that it is black and its only child is black or a leaf. We then repeatedly apply transformations 3–8 in Figure 2.6 until x has been deleted. In cases 3 and 6–8 we immediately get rid of the problematic node, case 4 makes the parent of x red, where one of the cases 6–8 in the next step will remove x. Only case 5 does not remove the problematic node, but here x is moved one level closer to the root, i.e. case 5 can at most be applied $O(\log n)$ times, and the total time for a deletion becomes $O(\log n)$. In the illustration of the cases, the symmetric cases are omitted, where x is the right child.

In case 3, x has been moved all the way to the root. In this case we we can just remove x.

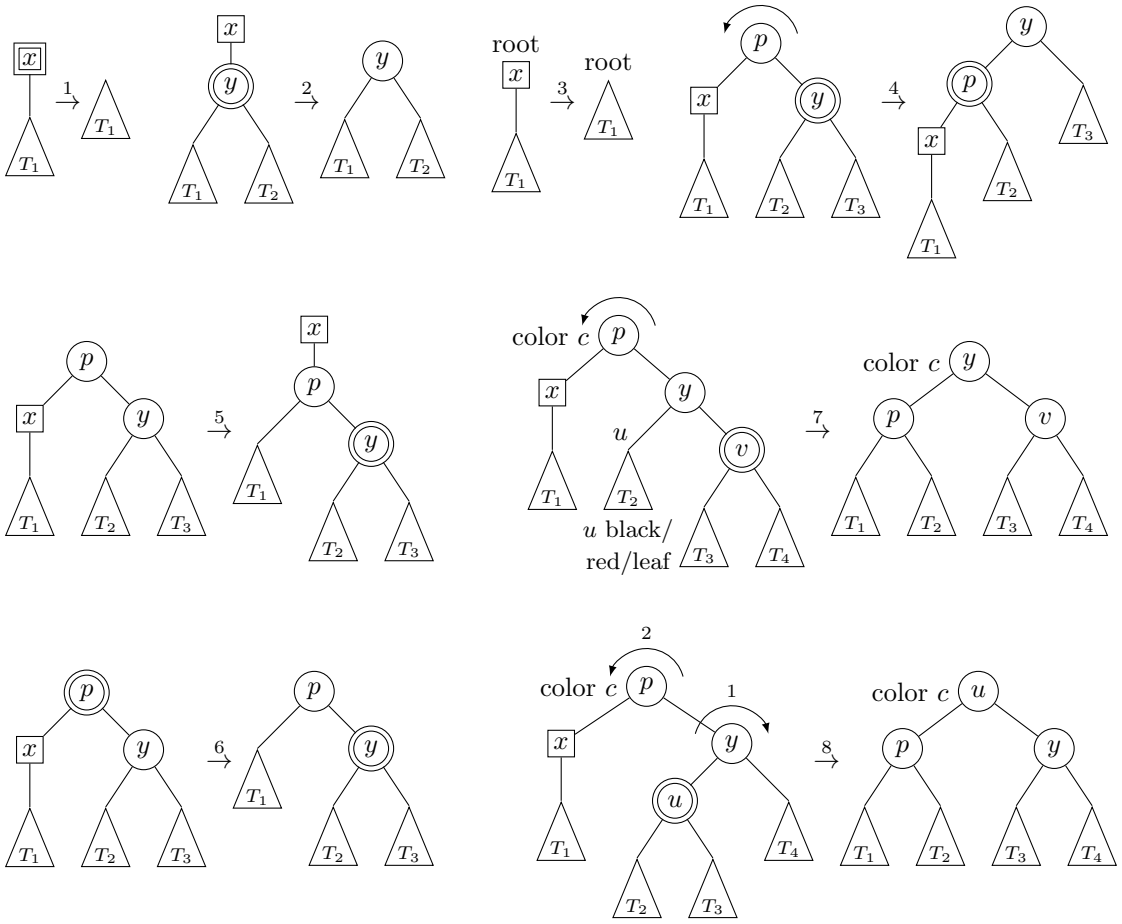


Figure 2.6: Deletion transformations (each subtree T_i has either a black root or is a leaf)

Recall that we are guaranteed that the child of x is black. This will reduce by one the number of black nodes on all root-to-leaf paths. Otherwise, x is not the root, and it must have a parent p. Furthermore, x must have a sibling y, since otherwise the number of black nodes would not be the same on all root-to-leaf paths. Case 4 is when y is red. Then p must be black and the children of y cannot be red. In this case we can rotate y up, color p red, and y black; see Figure 2.6. This guarantees that the red-black tree invariants remains satisfied, and the parent of x is now red and the sibling of x is black. The remaining cases only need to consider when y is black.

If y is black and has no red child, we are in cases 5 and 6. If p is also black, we can move x above p by letting x become the child of the parent of p, and letting the child of x replace x as a child of p. By coloring y red, the invariants remain satisfied. This is case 5 where x is moved one level closer to the root. If p instead is red, we can just remove x and swap the colors of x and y. This is case 6. In the remaining cases 7–8, y is black and has at least one red child (p can be either red or black). Let u and v be two children of y, such that v is furthest away from x in the tree (and sorted order). If v is red, then we are in cases 7, where we rotate y up, color v black, swap the colors of y and p, and remove x. This will guarantee that the number of black nodes to a leaf remains unchanged. The final case 8, is when u is red and v is black. In this case we rotate u up two levels, let u have the color of p, color p black, and remove x.

In an actual implementation one would not let x be explicitly represented in the tree while performing the transformations. Instead one would remove x at the beginning of the deletion, and implicitly represent x by keeping track on which tree edge the node x is on. This will in particular simplify the updates to the tree caused by case 5 (that can be repeated $O(\log n)$ times during a deletion): Only y needs to be colored red.

```
Algorithm RedBlackDelete(x)
Input  Delete node x from a red-black tree
1  if x has two children then
2      y = min(x.right)   # successor of x
3      swap elements at x and y
4      x = y
5  # x has at most one non-leaf child; if both are leaves we say x has a single leaf child
6  if x is red then
7      remove x (replace x by its child in the tree)   # case 1
8  else if x has a red child then
9      color the child black and remove x   # case 2
10 else
11     FixDelete(x)

Algorithm FixDelete(x)
Input  x is a black node to be deleted and its single child is black or a leaf
1  if x is the root then
2      remove x   # case 3
3  else if sibling y of x is red then
4      rotate y up   # case 4
5      FixDelete(x)   # only cases 6–8 will apply
6  else if y has no red child then
7      color y red
8      if parent p of x is black then
9          the child of x replaces x as a child of p, make x the parent of p   # case 5
10         FixDelete(x)   # distance to root reduced by one
11     else
12         color p black, remove x   # case 6
13 else
14     let u and v be children of y, such that u is closest to x
15     if v is a red node then
16         swap colors of p and y, rotate y up, remove x   # case 7
17     else
18         rotate u up twice, color u with the color of p, color p black   # case 8
```

Figure 2.7: Red-black tree deletion

2.1.3 Join

Assume we have two red-black trees T_1 and T_2, with n_1 and n_2 elements, respectively, and where $x_1 \leq x_2$ for all $x_1 \in T_1$ and $x_2 \in T_2$. The operation JOIN(T_1, T_2) should create a red-black tree containing all elements of both T_1 and T_2 (and T_1 and T_2 cease to exist). To implement this operation we first delete the minimum element x from T_2, such that $x_1 \leq x$ for all $x_1 \in T_1$ and $x \leq x_2$ for all $x_2 \in T_2$, and instead consider the operation JOIN(T_1, x, T_2). Let the black height of a node v be the number of internal black nodes on a path from v to a leaf.

Assume the black height of the root of T_1 is at least the black height of the root of T_2 (the case where T_2 has higher black height is symmetric). We can implement JOIN(T_1, x, T_2) by first traversing the rightmost path of T_1 to find the black node t_1 with black height equal to that of the root t_2 of T_2. Let p be the parent of t_1. By letting x be a red node, with left child t_1 and right child t_2, and parent p, we have spliced T_2 into T_1 such that it is a search tree where all root-to-leaf paths still have the same number of black nodes, and all red nodes have a black parent, except for possibly x. To fix the potential color problem at x, we call FIXINSERT(x). The time for JOIN will be $O(\log(n_1 + n_2))$. Figure 2.8 illustrates the steps of JOIN$(T_1, 37, T_2)$, where c) and d) are the steps performed when calling FIXINSERT(x).

2.2 Fenwick trees

A common task is to keep track of *prefix sums* of an array $A[0..n]$, while entries of A change. More specifically we consider data structures supporting

- INC(i, d) update $A[i] = A[i] + d$, where d can be both positive and negative,
- SUM(i) return $\sum_{j=0}^{i} A[j]$.

If we only maintain A, we can support INC in time $O(1)$ and SUM in time $O(i)$, i.e. worst-case time $O(n)$. The *Fenwick tree* [6] is a very simple data structure supporting both operations in time $O(\log n)$. It stores a single array F, of the same size as A, and without storing A.

Conceptually, the basic idea is to maintain a perfectly balanced tree with A as the leaves (assuming A contains a power of two elements), and each internal node stores the sum of all leaves in the subtree. An INC(i, d) operation adds d to leaf i and all its ancestors, and SUM(i) adds the roots of maximal complete subtrees with span in $A[0..i]$ (the gray nodes in Figure 2.9). Both operations take time proportional to the height of the tree, i.e. $O(\log n)$. The Fenwick tree stores an array F, where $F[i]$ stores the sum at the lowest ancestor of the ith leaf, that is a left child of its parent. Note that $F[i] = A[i]$ for all even i, and only a subset of the tree nodes is stored.

The pseudocodes for the operations on a Fenwick tree are given in Figure 2.10, where "|" and "&" denote bitwise *or* and *and* on binary numbers. It is amazingly short. The INIT operation assumes all A to be zeros, i.e. all subtree sums are also zero and F should contain zeros only.

For SUM(i), we observe that $F[i]$ stores the sum of the leaves in the largest subtree, where leaf i is the rightmost leaf. If j is leftmost leaf in this subtree, then SUM$(i) = F[i] +$ SUM$(j - 1)$. The expression $i \,\&\, (i + 1)$ computes j, by setting all the rightmost ones in the binary representation of i to zero (the binary representation of i corresponds to the left/right branching on the root-to-leaf path in the tree). In Figure 2.9 SUM$(13) = F[13] + F[11] + F[7]$, where $13 \,\&\, (13 + 1) = 12$, $11 \,\&\, (11 + 1) = 8$, and $7 \,\&\, (7 + 1) = 0$.

$$\begin{aligned} i &= 1011_2 = 11_{10} \\ i+1 &= 1100_2 = 12_{10} \\ i \,\&\, (i+1) &= 1000_2 = 8_{10} \end{aligned}$$

To perform INC(i, d) all nodes stored in F representing sums spanning $A[i]$ need to be incremented. These are store at indexes $\geq i$ in F and are exactly the ancestors that are a left child of their parent. These can be computed by repeatedly setting in i the least significant bit equal to zero to one. For an increment to $A[9]$, we in F need to increment positions $9 = 1001_2$,

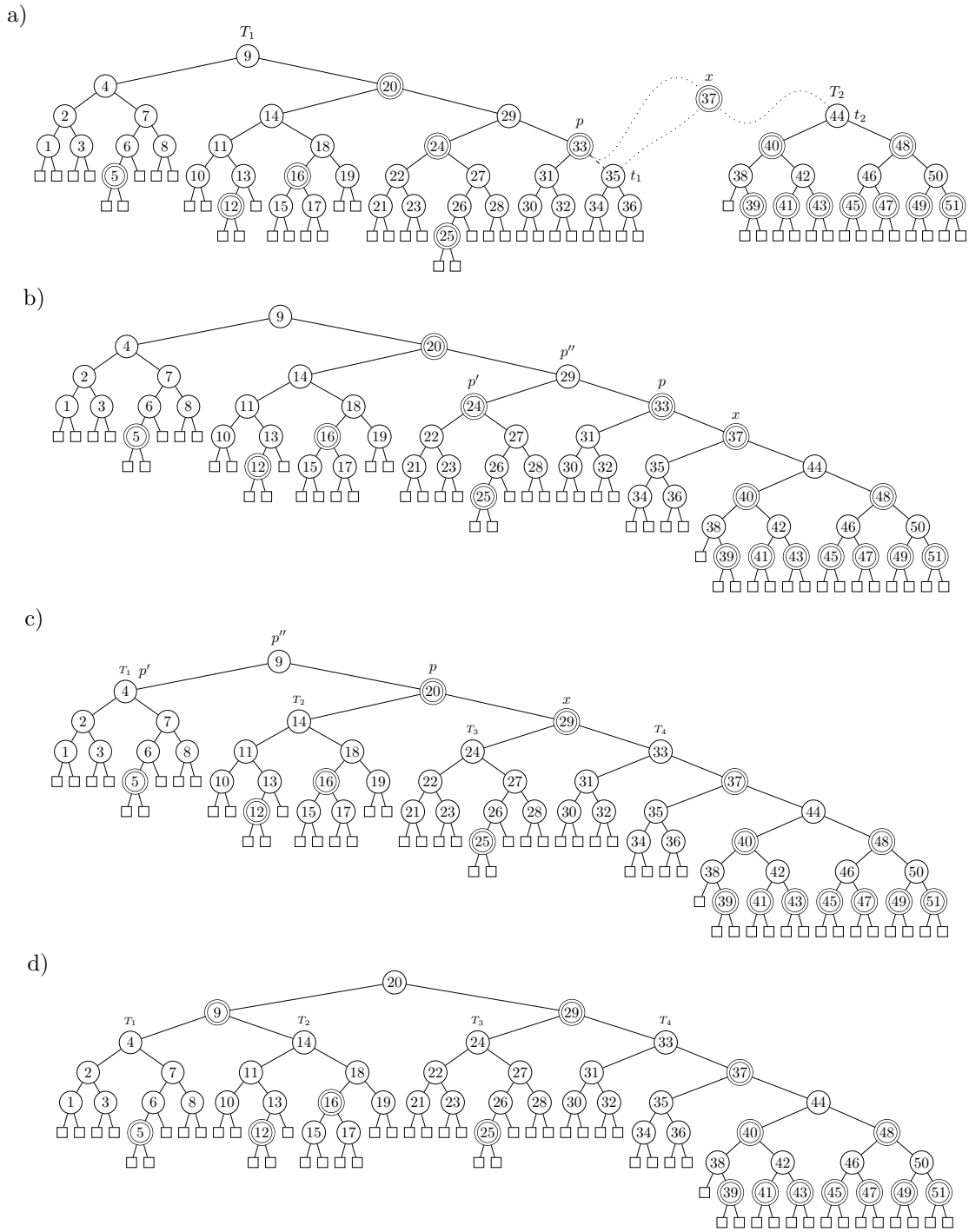


Figure 2.8: JOIN(T_1, x, T_2)

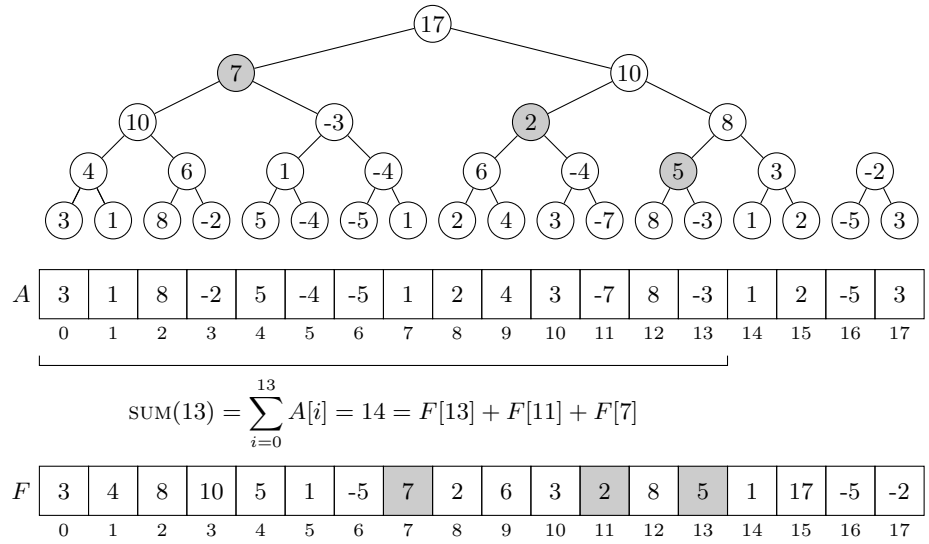


Figure 2.9: The Fenwick tree F for an array A

Algorithm INIT(n)
1 $F[0..n]$ = array with zeros

Algorithm SUM(i)
1 $s = 0$
2 **while** $i \geq 0$ **do**
3 $\quad s = s + F[i]$
4 $\quad i = (i \,\&\, (i+1)) - 1$
5 **return** s

Algorithm INC(i, d)
1 **while** $i \leq n$ **do**
2 $\quad F[i] = F[i] + d$
3 $\quad i = i \mid (i+1)$

Figure 2.10: Fenwick tree algorithms

$11 = 10\underline{1}1_2$, and $15 = 1\underline{1}11_2$. The expression $i \mid (i+1)$ sets the lowest bit equal to zero to one.

$$\begin{array}{rcccc} i & = & 1\underline{0}11_2 & = & 11_{10} \\ i+1 & = & 1\underline{1}00_2 & = & 12_{10} \\ i \mid (i+1) & = & 1\underline{1}11_2 & = & 15_{10} \end{array}$$

Exercise 2.1 Given the Fenwick tree F for an array A, describe how to support GET(i) that returns $A[i]$ when only F is available. What is the running time? ◁

Exercise 2.2 We can construct the Fenwick tree F for an array $A[0..n]$ by calling INC($i, A[i]$), for $i = 0, \ldots, n$, in time $O(n \log n)$. Describe how to construct F from A in time $O(n)$. ◁

Chapter 3

Amortized Analysis

Quiz 10 – Linear probing in dynamic arrays check

Assume we store a set of n numbers in an array of size N using hashing and linear probing, such that the load factor is in the interval $\left[\frac{1}{4}, \frac{3}{4}\right]$, i.e. $\frac{1}{4}N \leq n \leq \frac{3}{4}N$. Whenever $n < \frac{1}{4}N$ or $n > \frac{3}{4}N$, the n elements are *reinserted* into a new array of size $N = 2n$ (using a new hash function), i.e. for the new array we have $n = \frac{1}{2}N$. State the potential function Φ that can be used to prove that the total number of *reinsertions* into the hash table is amortized O(1) per insertion in and deletion from the set.

$$N - n \quad n \quad \tfrac{3}{4}N - n \quad n - \tfrac{1}{4}N \quad 2 \cdot |2n - N| \quad \left(n - \tfrac{1}{4}N\right)\left(\tfrac{3}{4}N - n\right)$$

Note: In this question we use a potential function to argue about the number of *reinsertions* instead of the running time.

Quiz 11 – Red-black trees with deletions in amortized constant time check

Red-black trees support INSERT and DELETE on a tree with n elements in worst-case time $O(\log n)$. State the potential function Φ that can be used to argue that INSERT takes amortized time $O(\log n)$ and DELETE amortized time O(1).

$$\textstyle\sum_{i=1}^{n} i \quad \sum_{i=1}^{n} \log i \quad \log n \quad n \quad n + \log n$$

Note: In this exercise the cost of deleting elements is charged to the insertion of the elements.

Quiz 12 – Deletions in binary heaps check

A binary max-heap supports INSERT og EXTRACTMAX in worst-case $O(\log n)$ tid. State the potential function Φ that can be used to argue that INSERT takes amortized time $O(\log n)$ and EXTRACTMAX amortized time O(1).

$$\log n \quad n \quad n + \log n \quad n - \log n \quad n \log n$$

Note: In this exercise the cost of deleting elements is charged to the insertion of the elements.

Quiz 13 – Binary heaps in dynamic arrays check

Consider a binary max-heap containing n elements stored in an array of size N. The operations Insert and ExtractMax are implemented as in a standard binary max-heap, except when Insert is performed when $n = N$, where the heap is copied to a new array of size $2n$ before a standard heap insertion is performed, i.e. N is doubled. State the potential function Φ that can be used to argue that Insert takes amortized time $O(\log n)$ and ExtractMax amortized time $O(1)$.

$$N - n + \log n \quad n \log n \quad |2n - N| + \sum_{i=1}^{n} \log i \quad N + n \log n \quad n + \log n$$

Note: In this exercise both the cost of deleting elements and resizing the array are charged to the insertions.

Quiz 14 – Binary counter in a dynamic array check

Consider an array $A[0..N-1]$ of length N, where each $A[i]$ either equals 0 or 1. We consider A to represent the binary number $n = \sum_{i=0}^{N-1} 2^i \cdot A[i]$. The operation Inc$(A)$ increases n by 1 by flipping $A[0], A[1], ...$ until the first $A[i]$ flips from 0 to 1. If all entries of A were 1 (i.e. $n = 2^N - 1$ before Inc(A) is performed), the array A is replaced with a new array of double size $2N$, where all $A[i] = 0$ except $A[N] = 1$. State the potential function Φ that can be used to argue that Inc takes amortized $O(1)$ time. Below k is the number of $A[i] = 1$, i.e. $k = |\{i \mid A[i] = 1\}|$.

$$k \qquad n \qquad N - k \quad \log n - \tfrac{1}{2}N \quad |2k - N|$$

Note: In this exercise the potential should both cover flipping a sequence of 1s to 0s, and for the doubling of the array

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Jon Louis Bentley. *Programming pearls, Second edition*. Addison-Wesley Professional, 1999.

[3] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.

[4] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Department of Mathematics, Harvard University, Cambridge, Mass., 1966.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: https://mitpress.mit.edu/books/introduction-algorithms-third-edition.

[6] Peter M. Fenwick. A new data structure for cumulative probability tables: An improved frequency-to-symbol algorithm. *Software - Practice and Experience*, 26(4):489–490, 1996. doi:10.1002/(SICI)1097-024X(199604)26:4<489::AID-SPE22>3.0.CO;2-S.

[7] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science (TTCS). An EATCS Series. Springer, 2010. doi:10.1007/978-3-642-16533-7.

[8] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208, 1982. doi:10.1016/0022-0000(82)90048-4.

[9] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

[10] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

[11] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. Technical Report hal-02070778, HAL, March 2019. URL: https://hal.archives-ouvertes.fr/hal-02070778.

[12] Anatolii A. Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics*, 211:169–183, 1995.

[13] Anatolii A. Karatsuba and Yuri P. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145(2):293–294, 1962. Translation in the academic journal Physics-Doklady, 7:595–596, 1963. URL: http://mi.mathnet.ru/eng/dan26729.

[14] János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in $O(n \log n)$ time. *SIAM Journal of Discrete Mathematics*, 11(3):347–372, 1998. doi:10.1137/S0895480196304982.

[15] Andrew Chi-Chih Yao and F. Frances Yao. The complexity of searching an ordered random table (extended abstract). In *17th Annual Symposium on Foundations of Computer Science*, pages 173–177. IEEE Computer Society, 1976. doi:10.1109/SFCS.1976.32.

Appendix A

Examples of Exam Questions

Quiz 15 – Asymptotic notation check

Below $\log n$ denotes the binary logarithm of n.

	Yes	No
$n \cdot \log n$ is $\mathrm{O}(\log n)$		
$n^{0.001}$ is $\mathrm{O}(n^{0.01})$		
$(\log n)/5$ is $\mathrm{O}(\log(n!))$		
$4^{\log n}$ is $\mathrm{O}(n^3)$		
$6 \log n^2 + (\log n)^6$ is $\mathrm{O}(n \cdot \log n)$		
$n \cdot (\log n)/2 + n \cdot (\log n)/2$ is $\mathrm{O}(n^{1/3})$		
$n^{0.1}/4$ is $\mathrm{O}(n^{0.01})$		
$2n^{1/3}$ is $\mathrm{O}(n^3)$		
$2n^{0.001}$ is $\mathrm{O}(2^{3 \log n})$		
$(\log n)^2$ is $\mathrm{O}(n \cdot \log n)$		
$\log n$ is $\mathrm{O}(\sqrt{n})$		
n^n is $\mathrm{O}(\sqrt{n})$		
$8^{\log n}$ is $\mathrm{O}(n)$		
$2n^{2/3}$ is $\mathrm{O}((\log n)^3)$		
n^2 is $\mathrm{O}(2^n)$		
$n \cdot \log n$ is $\mathrm{O}(1)$		
$\sqrt{n}/5$ is $\mathrm{O}(n)$		
n^2 is $\mathrm{O}(2^{3 \log n})$		
$n^n + n\sqrt{n}$ is $\Omega(n \cdot \log n)$		
$5 \cdot 8^{\log n}$ is $\Theta(2^{3 \log n})$		
$n^{0.001}$ is $\Theta(3^n)$		
n^2 is $\Omega(2^n)$		
$3\sqrt{n}$ is $\Theta(n \cdot \log n)$		
$2^{2 \log n}$ is $\Omega(n^{0.01})$		

Quiz 16 – Analysis of running times of loops check

State for each of the below algorithms the running time as a function of n in Θ-notation.

Algorithm LOOP1(n)
```
1  i = n
2  while i > 0 do
3      i = i − 1
```

Algorithm LOOP2(n)
```
1  i = 1
2  while i ≤ n do
3      i = i + i
```

Algorithm LOOP3(n)
```
1  s = 0
2  i = 1
3  while s ≤ n do
4      s = s + i
5      i = i + 1
```

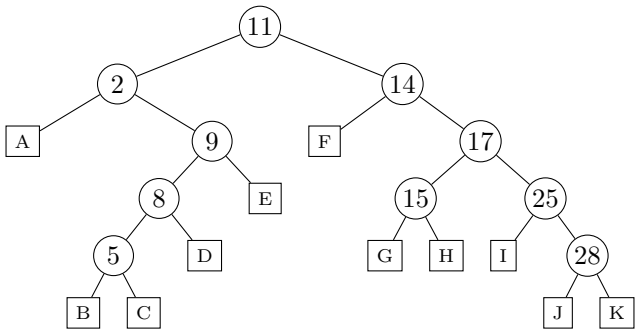
Algorithm LOOP4(n)
```
1  s = 1
2  i = 1
3  while i ≤ n do
4      for j = 1 to i do
5          s = s + 1
6      i = 2 ∗ i
```

Algorithm LOOP5(n)
```
1  i = 1
2  while i ≤ n do
3      j = 1
4      while j ≤ n do
5          j = 2 ∗ j
6      i = 2 ∗ i
```

Algorithm LOOP6(n)
```
1  s = 1
2  for i = 1 to n do
3      j = s
4      while j > 0 do
5          s = s + 1
6          j = j − 1
```

	$\Theta(2^n)$	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n^2)$	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta((\log n)^2)$	$\Theta(n^3)$
loop1								
loop2								
loop3								
loop4								
loop5								
loop6								

Quiz 17 – Insertions into an unbalanced search tree check



State at which leaf A–K each of the elements 27, 16, 7, 23 and 13 should be inserted (for each insertion it is assumed that the tree only contains the above ten elements).

	A	B	C	D	E	F	G	H	I	J	K
INSERT(27)											
INSERT(16)											
INSERT(7)											
INSERT(23)											
INSERT(13)											

Quiz 18 – Insertions into a max-heap check

The resulting binary max-heap after insertion the elements 2, 9, 6, 3, 12, 8 and 5 in the given order with MAX-HEAP-INSERT, starting with an initial empty heap.

1	2	3	4	5	6	7
12	9	8	2	3	6	5

1	2	3	4	5	6	7
12	9	8	6	5	3	2

1	2	3	4	5	6	7
12	9	8	3	2	6	5

1	2	3	4	5	6	7
2	9	6	3	12	8	5

1	2	3	4	5	6	7
9	12	8	3	2	6	5

Quiz 19 – Build-Max-Heap check

1	2	3	4	5	6	7	8	9
7	1	6	5	2	8	9	4	3

The result of applying BUILD-MAX-HEAP to the above array.

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9
7	5	9	4	2	8	6	1	3

1	2	3	4	5	6	7	8	9
9	5	8	4	2	6	7	1	3

1	2	3	4	5	6	7	8	9
9	8	7	6	5	4	3	2	1

1	2	3	4	5	6	7	8	9
9	5	8	4	2	7	6	1	3

Quiz 20 – Deletion from a max-heap check

1	2	3	4	5	6	7	8	9	10	11	12	13
23	22	20	16	21	18	17	1	11	3	2	7	4

The result of applying HEAP-EXTRACT-MAX to the above max-heap.

1	2	3	4	5	6	7	8	9	10	11	12
22	21	20	16	4	18	17	1	11	3	2	7

1	2	3	4	5	6	7	8	9	10	11	12
22	21	17	20	18	16	1	11	3	2	7	4

1	2	3	4	5	6	7	8	9	10	11	12
22	21	20	16	3	18	17	1	11	4	2	7

1	2	3	4	5	6	7	8	9	10	11	12
22	21	20	16	3	18	17	1	11	2	7	4

1	2	3	4	5	6	7	8	9	10	11	12	13
22	21	20	16	3	18	17	1	11		2	7	4

Quiz 21 – Partition check

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	26	11	14	4	24	20	23	3	10	25	7	18	6	8

The result of applying PARTITION$(A, 3, 14)$ to the above array A.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	4	6	7	8	10	11	14	18	20	23	24	25	26	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	26	3	4	6	7	10	11	14	18	20	23	24	25	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	26	4	3	6	24	20	23	14	10	25	7	18	11	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	26	4	3	6	11	14	24	20	23	10	25	7	18	8

Quiz 22 – Radix-sort check

4011 0013 4113 4403 3113 0003

Consider RADIX-SORT applied to the above list of numbers ($d = 4$, $k = 5$). State the partially sorted list after RADIX-SORT has sorted the numbers after the *two* least significant digits.

4011	4403	0003	0013	4113	3113
0003	0013	3113	4011	4113	4403
4403	0003	4011	0013	4113	3113
0013	0003	3113	4011	4113	4403
0003	4403	4011	0013	3113	4113

Quiz 23 – Linear probing check

0	1	2	3	4	5	6	7	8	9	10
				14	3			17	4	2

The above hash table of size 11 has been constructed using *linear probing* using the hash function $h(k) = 5k \bmod 11$.

State for each of the elements 0, 1, 6, 8 and 9 where it will be inserted in the hashtable (for each insertion the hash table only contains the above elements 2, 3, 4, 14 and 17).

	0	1	2	3	4	5	6	7	8	9	10
INSERT(0)											
INSERT(1)											
INSERT(6)											
INSERT(8)											
INSERT(9)											

Quiz 24 – Quadratic probing check

0	1	2	3	4	5	6	7	8	9	10
			5		12	10	17	6		

The above hash table of size 11 has been constructed using *quadratic probing* using the hash functions $h'(k) = 5k \bmod 11$ and $h(k,i) = (h'(k) + 5i + 5i^2) \bmod 11$.

State for each of the elements 1, 2, 3, 7 and 8 where it will be inserted in the hashtable (for each insertion the hash table only contains the above elements 5, 6, 10, 12 og 17).

	0	1	2	3	4	5	6	7	8	9	10
INSERT(1)											
INSERT(2)											
INSERT(3)											
INSERT(7)											
INSERT(8)											

Quiz 25 – Double hashing check

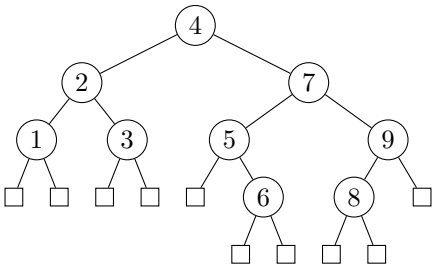
0	1	2	3	4	5	6	7	8	9	10
0		19			8			15		5

The above hash table of size 11 has been constructed using *double hashing* using the hash functions $h_1(k) = 2k \bmod 11$ and $h_2(k) = 1 + (3k \bmod 10)$.

State for each of the elements 1, 3, 4, 6 and 10 where it will be inserted in the hashtable (for each insertion the hash table only contains the above elements 0, 5, 8, 15 and 19).

	0	1	2	3	4	5	6	7	8	9	10
INSERT(1)											
INSERT(3)											
INSERT(4)											
INSERT(6)											
INSERT(10)											

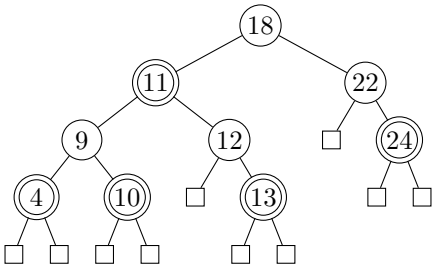
Quiz 26 – Valid red-black trees check



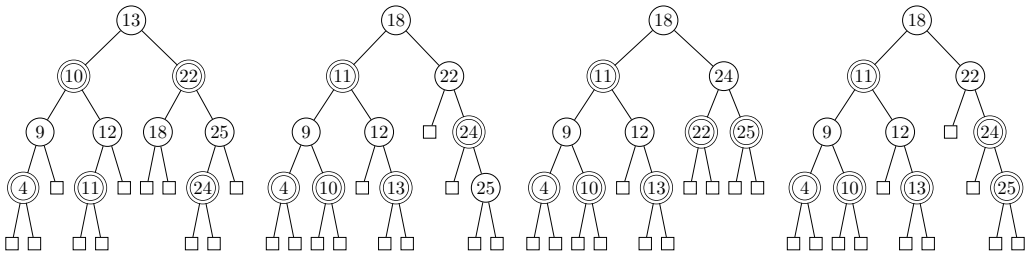
For each of the below sets, state if the above binary tree is a valid red-black if these nodes are the red nodes.

	Yes	No
4, 6, 8		
6, 8		
2, 6, 7, 8		
2, 5, 6, 8, 9		
1, 3, 6, 7, 8		

Quiz 27 – Insertions into red-black trees check



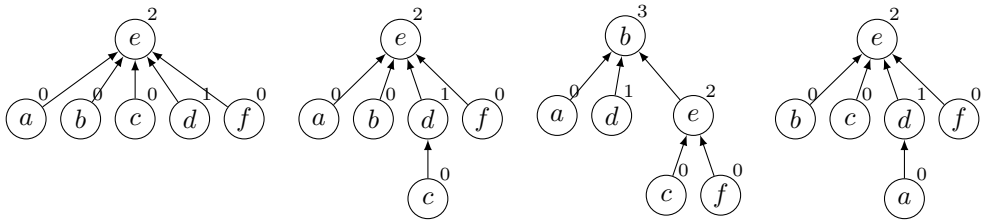
State the resulting red-black tree by insertion 25 in the above red-black tree (double circles are red nodes).



Quiz 28 – Union find operations check

MAKESET(a)
MAKESET(b)
MAKESET(c)
MAKESET(d)
MAKESET(e)
MAKESET(f)
UNION(a, d)
UNION(c, a)
UNION(f, e)
UNION(c, f)
UNION(c, b)
FIND-SET(a)

State the resulting union-find data structure after the above sequence of operations, when using union-by-rank and path compression.



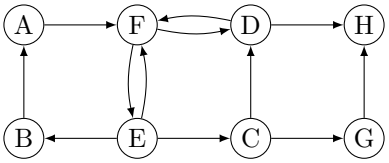
Quiz 29 – Recurrence equations check

State the solution to each of the below recurrence equations, where $T(n) = 1$ for $n \leq 1$.

$\Theta(\log n)$ $\Theta(\sqrt{n})$ $\Theta(n)$ $\Theta(n \log n)$ $\Theta(n^2)$ $\Theta(n^2 \log n)$ $\Theta(n^3)$

$T(n) = 9 \cdot T(n/3) + 2$
$T(n) = T(n-1) + 2$
$T(n) = 9 \cdot T(n/3) + n^2$
$T(n) = 5 \cdot T(n/5) + n$
$T(n) = 2 \cdot T(n/3) + n^3$
$T(n) = 8 \cdot T(n/2) + 2$
$T(n) = T(n-1) + n^2$
$T(n) = 3 \cdot T(n/5) + n$
$T(n) = 4 \cdot T(n/2) + 2$
$T(n) = 2 \cdot T(n/3) + n^2$

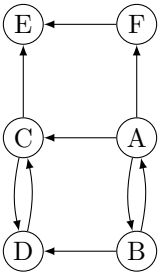
Quiz 30 – Breadth first seach (BFS) check



Consider a breadth first search of the above graph **starting at node A**. State the order the nodes are removed from the queue Q by the BFS algorithm. It is assumed that the graph is given by alphabetically sorted adjacency lists.

A F D H E B C G A F D E H B C G A F D E H C B G A F E D C B H G

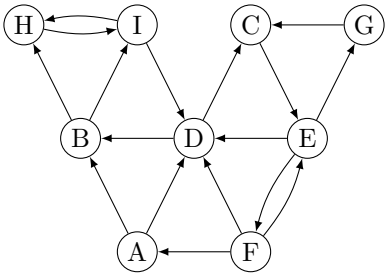
Quiz 31 – Valid BFS trees check



State for each of the below sets of it is a valid BFS tree for a breadth first search of the above graph **starting at the node A** and for an arbitrary ordering of the adjacency lists.

	Yes	No
(A,B) (A,F) (B,D) (C,E) (D,C)		
(A,B) (A,F) (B,D) (D,C) (F,E)		
(A,B) (A,C) (A,F) (B,D) (F,E)		
(A,B) (A,C) (A,F) (C,D) (F,E)		
(A,B) (A,C) (A,F) (B,D) (C,E)		

Quiz 32 – Depth first search (DFS) check



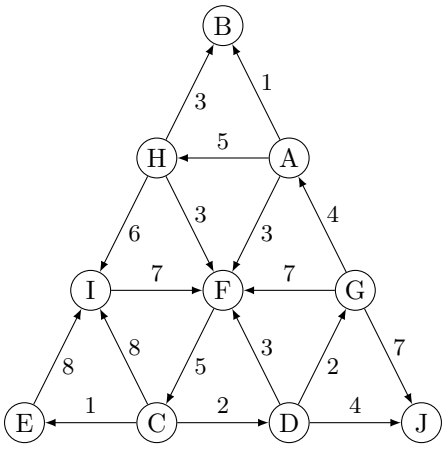
Consider a depth first search of the above graph stating in **node A**, and the outgoing edges of a node are visted in alphabetical order. State the order the nodes are assigned **discovery time**.

A B H I D C E G F A D B H I C E G F A B D H I C E F G A B H I D C E F G

State for each of the below edges the type it gets by running the DFS algorithm.

	Tree edge	Back edge	Cross edge	Forward edge
(C, E)				
(F, D)				
(A, D)				
(G, C)				

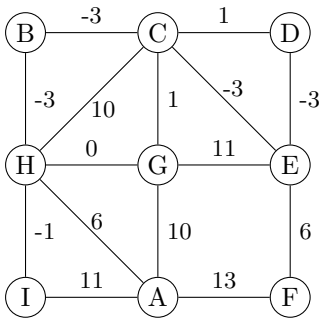
Quiz 33 – Dijkstra's algorithm check



Consider the application of Dijkstra's algorithm to compute the shortest distance from **node A** to all the nodes in the above graph. State the order the nodes are removed from the priority queue in Dijkstra's algorithm.

A B F C D G J E I H
A B F H C E D I G J
A B F H C E D G J I
A B F H C I D E G J

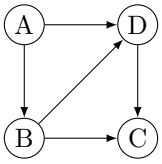
Quiz 34 – Prim's algorithm check



Consider finding a minimum spanning tree in the above graph using Prim's algorithm starting at **node A**. State the order the nodes are included in the minimum spanning tree (removed from the priority queue by Prim's algorithm).

A H B C E D G F I A H B C E D F G I A H B C E D F I G A H B C E D I G F

Quiz 35 – Topological sorting check

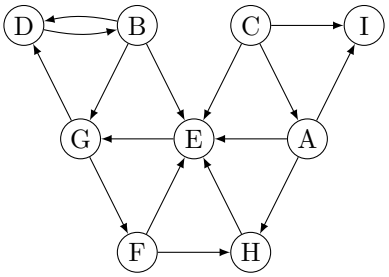


For each of the below orders state if it is a valid topological ordering of the nodes of the above graph.

	Yes	No
A B C D		
C B D A		
A C D B		
D B A C		
A B D C		

Quiz 36 – Strongly connected components

check



The number of strongly connected components in the above graph.

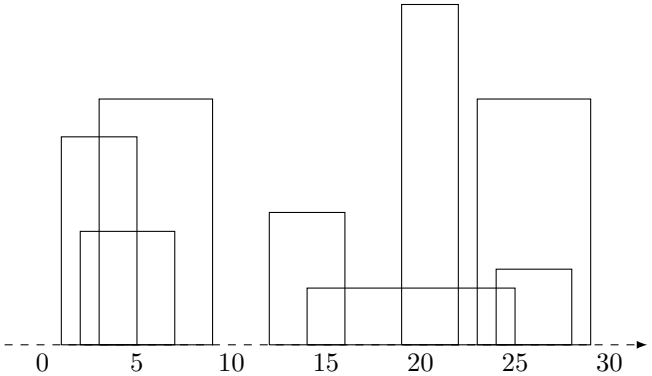
1 2 3 4 5 6 7 8 9

Appendix B

Problems

Problem B.1 (Nuts and bolts) You are given n nuts and n bolts, which fit pairwise, and all bolts and all nuts have distinct sizes. Given a nut and a bolt, you can decide if they fit, or the nut is too small or too large for the bolt. It is not possible to directly decide the relative sizes of two bolts, or the relative size of two nuts. The only way you can gain information is to compare a nut and a bolt. Describe an algorithm to pair the nuts and the bolts. What is the running time of the algorithm? (This problem has a simple randomized solution with expected running time $\mathrm{O}(n \log n)$; to find a deterministic algorithm with running time $\mathrm{O}(n \log n)$ turned to be a much harder problem, i.e. to find an algorithm that does not use randomization [14]). $\triangleleft$

Problem B.2 (Skyline) The outlines of buildings in a city can be represented by a set of rectangles. A building with height h is a represented by the triple (l, h, r), where the lower-left corner of the rectangle is $(l, 0)$ and the upper-right is (r, h).



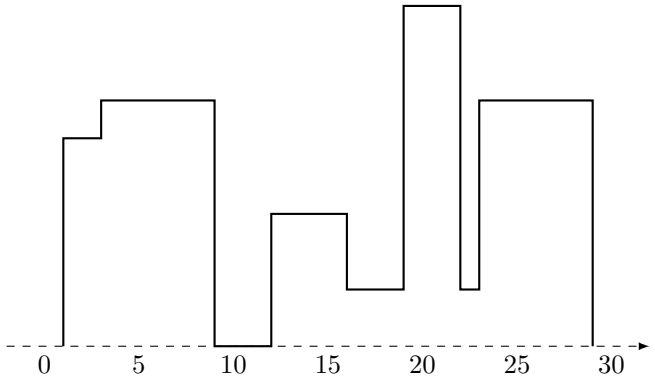
(1,11,5) (2,6,7) (3,13,9) (12,7,16) (14,3,25) (19,18,22) (23,13,29) (24,4,28)

The *skyline* of a city with n buildings can be represented by a polygonal line, that can be described by the sequence

$$(x_0, h_1, x_1, h_2, x_2, \ldots, x_{k-1}, h_k, x_k) \ ,$$

where $x_0 < x_1 < x_2 < \cdots < x_k$ and h_i is the height between x_{i-1} and x_i. Note that the skyline of a single building (l, h, r) is also (l, h, r).

The skyline for the above set of rectangles is the below.



(1,11,3,13,9,0,12,7,16,3,19,18,22,3,23,13,29)

(a) Describe an algorithm that can add a new building (l, h, r) to an existing skyline $(x_0, h_1, x_1, \ldots, x_{n-1}, h_n, x_n)$ in time $O(n)$.

(b) Generalize the solution from (a) to an algorithm to combine two skylines for n and m buildings, respectively, to a single skyline in time $O(n + m)$.

(c) Describe a divide-and-conquer algorithm for computing the skyline of n buildings, that uses (b) as a subroutine. What is the running time of your algorithm?

◁

Problem B.3 (Merging words)

Let $x = x_1x_2 \ldots x_n$, $y = y_1y_2 \ldots y_m$ and $z = z_1z_2 \ldots z_{n+m}$ be three strings of lengths n, m, and $n + m$, respectively. We say that z is a *merge* of x and y, if x and y can be found as two disjoint subsequences in z. Example $\underline{\text{T}}\ \overline{\text{H}}\ \underline{\text{R}}\ \underline{\text{E}}\ \overline{\text{E}}\ \overline{\text{N}}\ \underline{\text{E}}\ \underline{\text{S}}\ \overline{\text{S}}$ is a merge of $\underline{\text{T}}\ \underline{\text{R}}\ \underline{\text{E}}\ \underline{\text{E}}\ \underline{\text{S}}$ and $\overline{\text{H}}\ \overline{\text{E}}\ \overline{\text{N}}\ \overline{\text{S}}$.

For $0 \leq i \leq n$ and $0 \leq j \leq m$, we let $M[i, j]$ denote the boolean value that is true if and only if $z_1z_2 \ldots z_{i+j}$ is a merge of $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$. Note that for $i = 0$ we let $x_1x_2 \ldots x_i$ denote the empty string. $M[i, j]$ can be described by the following recurrence:

$$M[i,j] = \begin{cases} \text{true} & \text{if } i = 0 \ \wedge \ j = 0 \\ z_j = y_j \ \wedge \ M[0, j-1] & \text{if } i = 0 \ \wedge \ j > 0 \\ z_i = x_i \ \wedge \ M[i-1, 0] & \text{if } i > 0 \ \wedge \ j = 0 \\ (z_{i+j} = y_j \ \wedge \ M[i, j-1]) \ \vee \ (z_{i+j} = x_i \ \wedge \ M[i-1, j]) & \text{if } i > 0 \ \wedge \ j > 0\,. \end{cases}$$

(a) Fill out the below table for M when $x = \text{SHOE}$, $y = \text{EARS}$ and $z = \text{SEAHORSE}$.

$M[i,j]$		E	A	R	S
	0	1	2	3	4
	0				
S	1				
H	2				
O	3				
E	4				

(b) Give an algorithm based on dynamic programming that decides if z is a merge of x and y. Describe the algorithm using pseudocode. What is the running time of the algorithm?

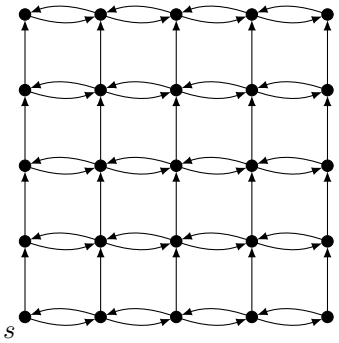
(c) Extend the algorithm to also report the indexes into z, that constitute the subsequence equal to x.

◁

Problem B.4 (Grid graphs) A *grid graph* is a graph with k^2 organized in a grid of k rows and k columns, with node $v_{i,j}$ being the jth node in row i, where row one is the bottom row. Let $s = v_{1,1}$. The nodes and edges of a grid graph are:

$$\begin{aligned} V &= \{v_{i,j} \mid 1 \leq i \leq k \ \wedge \ 1 \leq j \leq k\} \\ E &= \{(v_{i,j}, v_{i,j+1}) \mid 1 \leq i \leq k \ \wedge \ 1 \leq j < k\} \ \cup \\ &\quad \{(v_{i,j}, v_{i,j-1}) \mid 1 \leq i \leq k \ \wedge \ 1 < j \leq k\} \ \cup \\ &\quad \{(v_{i,j}, v_{i+1,j}) \mid 1 \leq i < k \ \wedge \ 1 \leq j \leq k\} \end{aligned}$$

The below is the grid graph for $k = 5$.



In the following we assume that the edges all have assigned a positive weight.

(a) State the number of nodes n and edges m in a grid graph as a function of k.

(b) What is the running time of Dijkstra's algorithm when used to find the shortest distance from s to all the nodes of the graph? State the running time as a function of k.

(c) Describe an algorithm that finds the shortest distance from s to all nodes in a grid graph in time $\mathrm{O}(m)$. Argue for the running time and the correctness of the algorithm.

$\triangleleft$