Introduction

The process of designing algorithms consists of roughly four phases:

1. Giving a formal problem description. Writing a problem description includes describing the problem setting, type of the input, and what kind of an answer we need for our question. The description does not need to be long, but it is important that there are no ambiguities.

2. Describing the algorithm. Describing the algorithm is usually done by using pseudocode, but could as well be done using natural language. Usually, it is sufficient to describe the outline of the algorithm and leave out the technical details of the implementation.

3. Proving correctness of the algorithm. Proving the correctness of the algorithm means showing that the algorithm works as intended. It must work on all inputs specified in the problem description.

4. Analysing the performance of the algorithm. The performance of the algorithm means its runtime and memory consumption. We are usually most interested in the *worst-case* runtime and memory consumption of the algorithm.

In this course, we introduce a variety of algorithm design disciplines. While many of the algorithms and techniques might be familiar from other courses, in this course we focus especially on learning to prove correctness and analyzing the runtime of the algorithms we design. The goal is to introduce the necessary tools to design our own algorithms and rigorously communicate them to others.

One way to approach the correctness of the algorithm would be to write tests for the code to pass. Unfortunately, we can never test all inputs, and without careful thinking, we might miss some important corner cases. To make sure that our code works on all inputs, we must *prove* its correctness formally. Formal proving can be done already in the design phase of the algorithm, before implementing anything. That way we can guarantee that the algorithm works as long as it is implemented according to the description.

Proving correctness is sometimes an intuitive process of simply communicating our thoughts on paper. On the other hand, very often a problem that looks simple turns out to be more complicated than first expected. Hence it is important to include a certain

level of formalism to make sure that all the steps that we take are in fact accurate. In this course, one of the goals is to learn how to precisely argue the correctness of our algorithms.

As mentioned earlier, in runtime analysis, we are mainly interested in the *worst-case runtime* of the algorithm. The worst-case runtime means that our algorithm performs at least that efficiently on *all possible inputs*. This way we can construct an upper bound for the runtime and ensure that our algorithm always reaches a certain level of efficiency. The runtime is usually measured by counting the number of primitive operations – like additions or memory accesses – the algorithm executes.

Memory consumption of the algorithm measures how much additional memory the algorithm reserves, and it is generally described as the number of primitive datatypes that need to be stored. Because the efficiency of the algorithm usually depends on the input, both runtime and memory consumption are represented as a function of the input size.

Example. Let's go through the steps above with a simple example.

1. **Problem description.** Let $A[1 \dots n]$ be an array of n integers. What is the smallest number in A?

2. **Algorithm.** We will go through the elements of A one by one and store the current smallest value. The algorithm is given as pseudocode below.

   Algorithm 1: Min(A,m)

   ```
   m ← A[1]
   for i ← 1 ... n do
   |   if A[i] < m then
   |   |   m ← A[i]
   end
   return m
   ```

3. **Correctness.** Let's use induction on the length of the array to prove that after the for-loop, variable m contains the minimum of the array A.

   Base case: When $n = 1$, there is only one element in the array, and $A[1]$ must be the minimum. We never enter the for-loop and m contains the minimum of A.

   Inductive hypothesis: Suppose that when the length of the list is k, the minimum of the array A is stored in m.

   Induction step: Let the length of the array be $k + 1$. After the first k steps, by the induction hypothesis, the variable m contains the smallest element of the subarray $A[1, \dots, k]$. After that, we enter the for-loop for one last time and compare $A[k + 1]$ to m.

   - If $A[k + 1] < m$, $A[k + 1]$ must be the smallest element of the whole array, as m was the smallest element of the subarray $A[1 \dots k]$. The algorithm sets $m \leftarrow A[k + 1]$ correctly.

- If $A[k+1] \geq m$, the variable m already contains the smallest element in the array and the algorithm works correctly by not changing m.

Now by induction, the algorithm works correctly for any $A[1 \dots n]$, $n \geq 1$.

4. **Runtime.** Let $T(n)$ be the number of basic operations for an array of size n. Before the for-loop, we have one assignment operation. Inside the for-loop, we execute at most two operations: one comparison and maybe one assignment. The for-loop runs $n-1$ times, and the number of operations is bounded by

$$T(n) \leq 1 + 2(n-1) \leq 2n.$$

Usually, we disregard the constants in runtime analysis and say that the runtime is $O(n)$.

Our algorithm only uses one integer variable, so its memory consumption is $O(1)$.

0.1 Asymptotic analysis

Asymptotic analysis is a valuable tool used in the analysis of runtime and memory consumption of algorithms. The idea is to simplify the function for the complexity as much as possible, practically getting rid of all the constant factors of the runtime. Additionally, if the runtime consists of many terms, we only focus on the most expensive one. This allows us to easily compare the performances of different algorithms designed for the same task without being restricted by the details of the implementations. Practically, all this is done by describing the behavior of algorithms when input size n "grows very large".

Throughout the section, we assume f and g to be nonnegative functions.

The *O-notation* is used to describe upper bounds for functions.

big-O-notation

Definition 0.1 (Big-O-notation)**.** Let f and g be functions. If there exists constants $c > 0$ and $n_0 \geq 0$ such that for any $n \geq n_0$ we have $f(n) \leq c \cdot g(n)$. Then $f(n) = O(g(n))$.

If $f(n)$ is a function describing the runtime of some algorithm, saying $f(n) = O(g(n))$ means that the runtime of the algorithm increases at most at the rate of the function $g(n)$ after the threshold value n_0 for the input size.

We can also define lower bounds for functions in a similar manner.

big-Ω-notation

Definition 0.2 (Big-Ω-notation)**.** If there exists constants $c > 0$, $n_0 \geq 0$ such that for all $n \geq n_0$ we have $g(n) \leq c \cdot f(n)$, then $f(n) = \Omega(g(n))$.

This means that f increases at least as quickly as g after the threshold n_0. Big-Ω-notations can also be defined using the big-O-notation:

$$f(n) = \Omega(g(n)) \iff g(n) = O(f(n)).$$

The Big-Θ-notation is used for two functions that increase at the same rate. We may formulate the definition using O- and Ω-notations:

Definition 0.3 (Big-Θ-notation)**.** *big-Θ-notation*

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Example 0.1. Here are some examples of the usage of these definitions.

1. Let $f(n) = n^2$ and $g(n) = n^3$. Then $f(n) = O(g(n))$.
2. Let $f(n) = \log n$ and $g(n) = n$. Then $f(n) = O(g(n))$.
3. Let $f(n) = 4n^2 + 5\log n$. Then $f(n) = O(n^2)$.
4. Let $f(n) = \log n$ and $g(n) = \log\log n$. Then $f(n) = \Omega(g(n))$.
5. Let $f(n) = n^n$ and $g(n) = n!$. Then $f(n) = \Omega(g(n))$.
6. Let $f(n) = n$ and $g(n) = 2n$. Then $f(n) = \Theta g(n))$.
7. Let $f(n) = n^3 + 3n$ and $g(n) = n^3 + \log n$. Then $f(n) = \Theta(g(n))$.

There exist strict correspondents for all of these concepts, defined in the sense that "function f grows strictly slower than g".

Definition 0.4 (Small-o-notation, Small-ω-notation)**.** If for *any* constant $c > 0$ there exists some n_0 such that for all $n \geq n_0$ we have $f(n) < c \cdot g(n)$, then $f = o(g(n))$. If $g(n) = o(f(n))$, we may write $f(n) = \omega(g(n))$. *small-o- and small-ω-notations*

The previous definition of o-notation is equivalent to the following limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

The o-notation is the logical opposite of the big-Ω-notation, for example

$$f(n) = o(g(n)) \iff f(n) \neq \Omega(g(n)),$$

and similarly for the small-ω- and big-O-notations.

Example 0.2. Here are some examples of o- and ω-notations.

1. Let $f(n) = \frac{1}{2}n$. We have $f(n) = \omega(\log n)$ and $f(n) = o(n^{3/2})$, but $f(n) \neq o(n)$.
2. As examples of functions that are $o(n)$, we have $\log n = o(n)$, $\sqrt{n} = o(n)$, and $\frac{n}{\log n} = o(n)$.

The following example illustrates that it is possible to use asymptotic notation using multiple variables.

Example 0.3. Consider two matrices, A of size $m \times k$ and B of size $k \times n$. Denote the element of A on the ith row and jth column by A_{ij}. The product $C = A \cdot B$ of these

matrices is defined as

$$C_{ij} = (A \cdot B)_{ij} = \sum_{\ell=1}^{k} A_{ik} \cdot B_{kj}.$$

The matrix C will have $m \times n$ entries, and for each entry, we must compute k multiplications. Hence the time complexity of this operation is $O(m \cdot n \cdot k)$.

1 Graph theory bootcamp

1 Graph theory

Graphs are versatile models used in many different fields and applications. Graphs have been widely studied and there are plenty of graph theory results that can be used for algorithm design as well as an ever-growing need for efficient algorithms related to problems concerning graphs.

A graph is a structure consisting of entities called *vertices* (or *nodes*) and connections between them called *edges*. Structures that can be modeled using graphs are for instance

- Social networks: nodes represent people and an edge between two people implies that they know each other.
- Road maps: nodes are cities and edges represent roads between them.
- Internet: nodes are websites and edges between websites represent links from one website to another.

Different settings obviously introduce a variety of different problems, all with specific requirements and definitions. The goal of this chapter is to give an introduction to the most basic graph theory concepts.

1.1 Basic definitions and terminology

Let us begin by rigorously defining a graph.

simple undirected graph

Definition 1.1 (Simple undirected graph)**.** A *simple undirected graph* is a pair $G = (V, E)$ consisting of the set V of *vertices* and sets of type $\{u, v\} \in E$, called *edges* where $u, v \in V$. The vertices u and v are called the *endpoints* of the edge $\{u, v\}$. The set of vertices is sometimes denoted $V(G)$ and similarly the set of edges by $E(G)$.

It is common to represent a graph as in Figure 1.1a, where circles represent vertices and lines between the circles represent edges. Most of the graphs on the course are *undirected*, but sometimes we need to use *directed edges*.

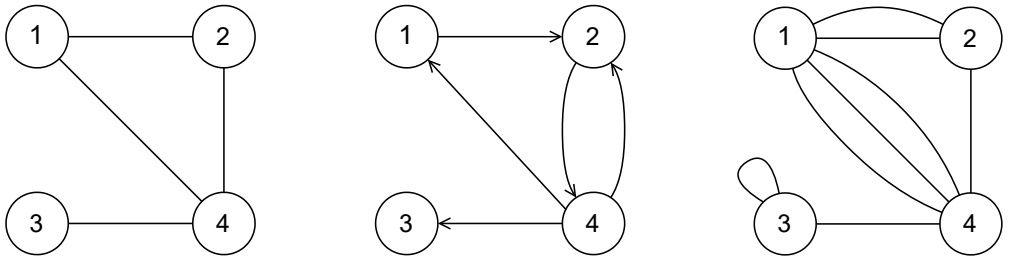
directed graph

Definition 1.2 (Directed graph)**.** A *directed graph* is a pair $G = (V, E)$ where the set

of edges E consists of ordered pairs $(u, v) \in E$, where $u, v \in V$.

The pair (u, v) is an edge *from* u *to* v, and it is commonly represented by an arrow pointing from u to v (Figure 1.1b). If it is clear from context whether a graph is directed or undirected, we can denote an edge simply by uv.

A graph that allows multiple edges between two nodes is called a *multigraph*. If the graph does not contain multiple edges between vertices it is called *simple*. In this course, almost all of the graphs we encounter are simple. *multigraph*



(a) Simple undirected graph with four nodes and four edges. Node 4 has three neighbors, and hence its degree is three.

(b) Directed graph with four nodes. Node 2 has in-degree one and out-degree two.

(c) Undirected multigraph with eight edges. Node 4 has degree five, and the degree of the node 3 is two.

Figure 1.1: Example graphs.

Definition 1.3 (Neighbor and degree)**.** Two vertices are *neighbors* if there is an edge between them. The set of all neighbors of v is denoted by $N(v)$. The number of neighbors of a vertex is called its *degree*, denoted by $\deg(v) = |N(v)|$. *neighbor* *degree*

In directed graphs, we have the analogous definitions *indegree*, the number of incoming edges, and *outdegree*, the number of outgoing edges. The *maximum degree* of the graph is $\Delta(G) = \max_{v \in V} \deg(v)$, sometimes simply denoted Δ. A graph where all vertices have the same degree d is called *d-regular*. *indegree* *maximum degree* *d-regular graph*

The following result connects the number of edges and the total degree of the graph.

Lemma 1.1 (Handshaking Lemma)**.** Let $G = (V, E)$ be a graph. Then *Handshaking lemma*

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|.$$

Proof. The degree of a node is equal to the number of edges incident to it. As an edge has two endpoints, each edge is counted twice in the sum. □

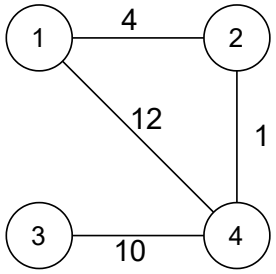
To model some problems, we sometimes need to represent a property – like capacity or length – of an edge by assigning a *weight* to it.

Definition 1.4 (Weighted graph)**.** A weighted graph is a graph $G = (V, E)$ together *weighted graph*

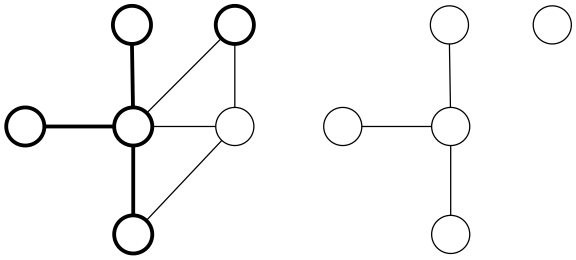
with the function $w : E \to \mathbb{R}$, where $w(e)$ is called the *weight* of the edge $e \in E$.

In figures, the weight of the edge is usually written next to the edge (Figure 1.2a). Unless stated otherwise, it is usually safe to assume that a graph is *unweighted*.

Definition 1.5 (Subgraph)**.** A *subgraph* $H = (V', E')$ of a graph $G = (V, E)$, denoted by $H \subseteq G$, satisfies $V' \subseteq V$ and $E' \subseteq E$. The graph H is called a *proper subgraph* if $G \neq H$. *subgraph*



(a) A weighted graph with four nodes and four edges. For example, the weight of the edge between nodes 3 and 4 is 10.



(b) The graph on the right is a subgraph of the graph on the left. The vertices and edges chosen to the subgraph have been bolded in the original graph.

Figure 1.2: More example graphs.

1.2 Paths and connectivity

Many graph problems involve working with the concept of *paths*. Lots of real-life structures can be modeled using paths, for instance in a road map, a path represents a route along the roads going through different cities. Paths are also used to define the concepts of *connectivity* and *distance*.

Definition 1.6 (Undirected path)**.** An *undirected path* is a graph $P = (V, E)$ with vertices $V = \{v_0, v_1, \ldots, v_n\}$ such that consecutive vertices v_i and v_{i+1} are neighbors. Formally, the set of edges is of the form *path*

$$E = \{v_0v_1, v_1v_2, \ldots, v_{n-1}v_n\}.$$

The vertices v_0 and v_n are called the *endpoints* of the path. The *length* of the path is the number of edges in it, and a path with length n is commonly denoted by P_n. *path length*

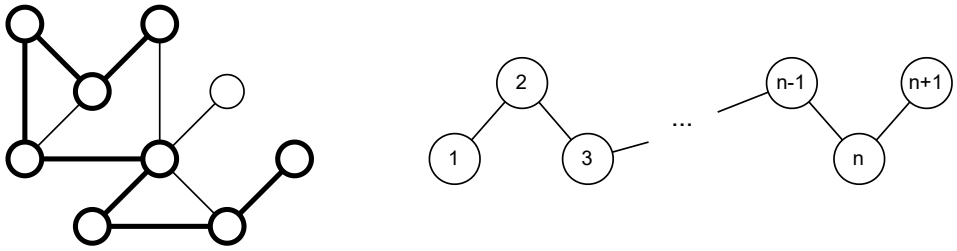


Figure 1.3: A subgraph that is a path on the left. A path of length n on the right.

For a directed graph, the definition is similar but the direction of the edges is relevant. *directed path*

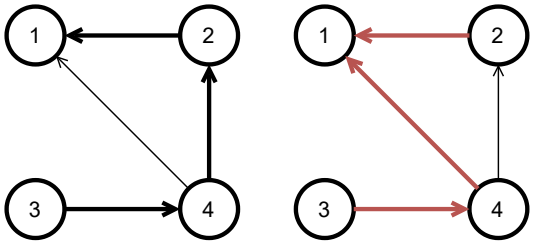


Figure 1.4: The bolded subgraph on the left is a directed path of length three but the one on the right highlighted in red is not.

Definition 1.7 (Distance)**.** The *distance* between two vertices is the length of the shortest path between them, and it is denoted by $d(u, v)$. *distance*

Example 1.1. In Figure 1.1a, the distance between nodes 1 and 4 is two, and $d(1, 3) = 2$. In the weighted graph 1.2a, the length of a path takes the weights into account. For instance, the distance between the nodes 1 and 4 is 12, and $d(1, 3) = 15$.

Definition 1.8 (i-hop neighborhood)**.** The *i-hop neighborhood* of a vertex u, denoted $N^i(u)$, is the set of vertices within distance i of the vertex: *i-hop neighborhood*

$$N^i(u) = \{v \in V : d(u, v) \leq i\}.$$

Definition 1.9 (Connected graph)**.** Two nodes are *connected* if there exists a path between them. If every pair of nodes in the graph is connected, the graph is called connected. Similarly iIn the case of directed graphs, we usually call the graph *strongly connected* if each pair of vertices has a directed path between them. *connected*

We call the connected subgraphs of a graph its *connected components*. Each vertex of a graph belongs to exactly one connected component. *connected component*

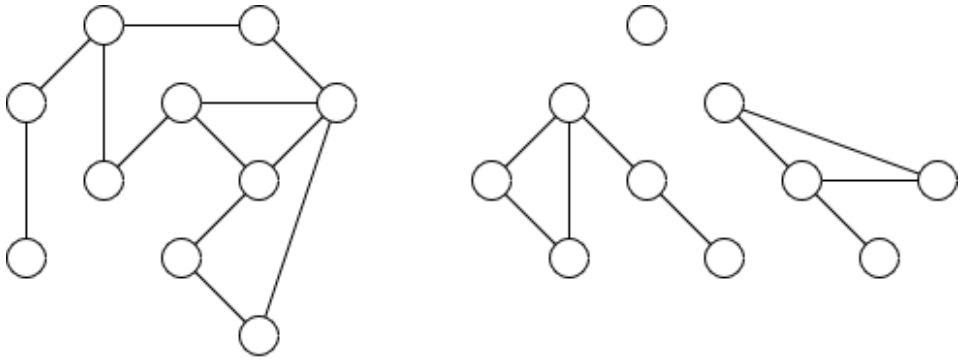


Figure 1.5: A connected graph on the left and a disconnected graph with three connected components on the right.

Definition 1.10 (Cycle)**.** A *cycle* is a connected graph where all vertices have degree two. In other words, a cycle is a graph $C = (V, E)$ where

cycle

$$V = \{v_0, \ldots, v_n\} \text{ and } E = \{v_0v_1, \ldots, v_{n-1}v_n, v_nv_0\}$$

A cycle can be thought as a path where $v_0 = v_n$. Similarly to a path, the length of a cycle is the number of edges in it.

cycle length

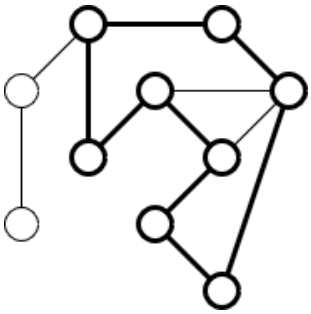


Figure 1.6: The bolded subgraph is a cycle of length eight.

1.3 Trees and forests

Trees are one of the most fundamental graph types having plenty of applications in many different fields. With trees, we can for instance

- model different hierarchical relationships between entities,
- efficiently implement many algorithms, like *binary search*,
- approximate dense graphs by removing some of the edges.

Additionally, trees and their properties have been widely studied, which gives us plenty of tools to analyze algorithms related to trees.

Definition 1.11. (Forest and tree) A *forest* is a graph that does not contain cycles. A *tree*, usually denoted by $T = (V, E)$, is a connected acyclic graph. Hence the connected components of a forest are trees. The *leaves* of the tree are the nodes of degree one.

forest
tree
leaf

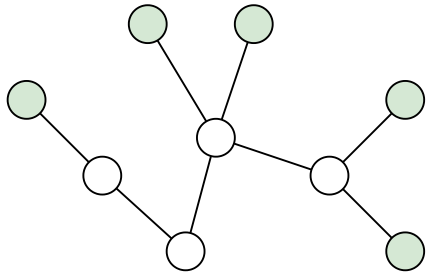


Figure 1.7: A tree with nine nodes. The leaves of the tree are colored in green.

Every tree with at least two nodes has at least one leaf. The number of edges in a tree is directly related to the number of vertices.

Lemma 1.2. Let $T = (V, E)$ be a tree. Then

$$|E| = |V| - 1.$$

Proof. We will prove this using induction on the number of vertices.

Base case: Let $|V| = 1$. Then the tree contains no edges, and the equality holds.

Induction hypothesis: The equality holds for all trees with less than k nodes.

Induction step: Let $|V| = k$. Removing any leaf from the tree removes exactly one edge, and the remaining graph is a tree with $k-1$ vertices. By the induction hypothesis, this graph has $k-2$ edges, and thus the original graph satisfies $|E| = k - 1 = |V| - 1$.

□

Sometimes we need to identify a vertex to be the *root of the tree*. A tree that has a root is called a *rooted tree*. Rooting a tree makes it sometimes easier to describe algorithms and prove claims.

root
rooted tree

Definition 1.12 (Depth)**.** The *depth* of a vertex v in a rooted tree is the distance between the root and the vertex v. The depth of the tree is the maximum depth over all the vertices in the tree.

depth

We can for instance use the depth of the vertices to iterate over the tree. We sometimes refer to the set of vertices at a certain depth as a *layer* of the tree.

layer

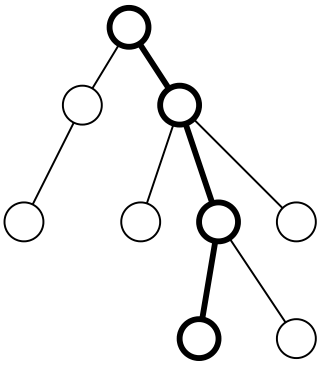
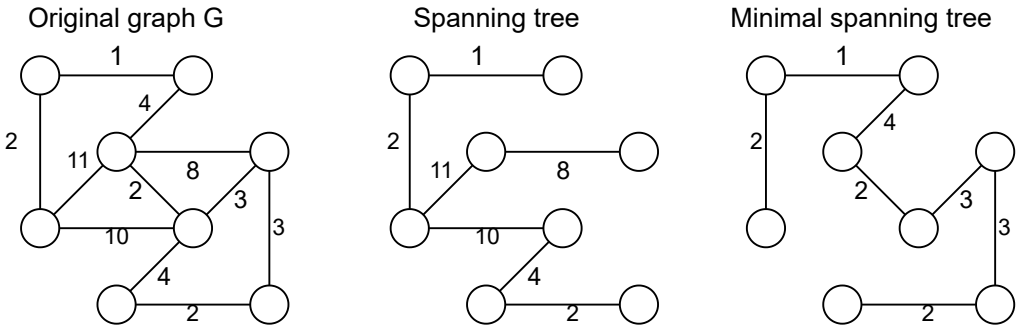


Figure 1.8: A rooted tree where the top node is the root. The depth of the tree is three. The path from the root to a leaf has been bolded.

Definition 1.13 (Spanning tree)**.** A *spanning tree* T is a subgraph of G that is a tree where $V(T) = V(G)$. *spanning tree*

Every connected graph has a spanning tree, which can be proven constructively by designing an algorithm for finding one. A graph can have multiple different spanning trees.

For weighted graphs, we are sometimes interested in finding a *minimum spanning tree*, that is a spanning tree with the minimum total weight. We will return to minimum spanning trees and algorithms related to them in the fourth week of the course. *minimum spanning tree, MST*



1.4 Graph coloring and independent sets

Graph coloring is a well-researched problem with many real-life interpretations, like scheduling and minimizing conflicts. It is known to be a difficult problem with no efficient algorithms for solving it.

When we talk about graph coloring, we mean assigning different colors to the vertices of the graph. It is common to denote the "colors" using integers.

Definition 1.14 (Proper vertex coloring)**.** A proper k-coloring is a function $\phi : V \to \{1, \ldots, k\}$ such that if $uv \in E$ then $\phi(u) \neq \phi(v)$. *k-coloring*

In other words, the goal is to assign colors for the vertices of the graph so that all neighboring vertices get a different color. Writing $\phi(v) = i$ means that the vertex v is given the color i. We denote the set of all the nodes colored with color i by $\phi^{-1}(i)$. $\phi^{-1}(i)$

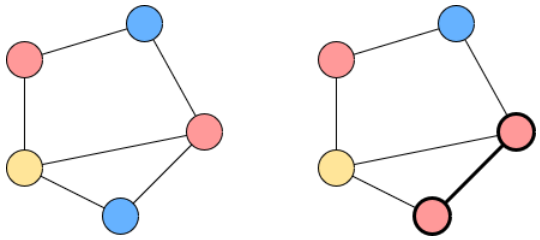


Figure 1.9: A proper vertex coloring on the left. The coloring on the right is not proper, because the bolded vertices are neighbors but both of them are red.

If we do not constrain the number of colors that can be used, it is easy to find a proper coloring for a graph; for example, using $|V|$ colors will always give a proper coloring as we can assign each node a different color. The problem of minimizing the number of colors is generally difficult.

Definition 1.15 (Chromatic number)**.** The *chromatic number* $\chi(G)$ of the graph G is the smallest number of colors needed to color the graph. *chromatic number*

The following result allows us to determine an upper bound for the chromatic number.

Lemma 1.3. For any graph G with maximum degree Δ, we have

$$\chi(G) \leq \Delta + 1.$$

Proof. We prove this by constructing a valid $(\Delta + 1)$-coloring for the graph. We start by ordering the vertices arbitrarily by $V = \{v_1, \ldots, v_n\}$, and then color them one by one always choosing the smallest available color. When coloring the vertex v_i, it has at most Δ already colored neighbors, hence there is always at least one color available since the number of colors is $\Delta + 1$. □

This bound is still very loose, but sometimes it works as a good starting point. The proof used a greedy graph coloring algorithm. It can be shown that the greedy algorithm does not guarantee optimal coloring.

Definition 1.16 (Independent set)**.** The set $M \subseteq V$ is an *independent set* if no vertex in M has neighbors inside M. *independent set*

In other words, the graph G does not have any edges between the vertices M. For any proper k-coloring ϕ, the set $\phi^{-1}(i)$ is an independent set for any $i \in \{1, \ldots, k\}$, ie. the vertices colored with the same color form an independent set.

Definition 1.17 (Maximal independent set)**.** A *maximal independent set* M is a subset of vertices V such that adding any vertex from $V \setminus M$ to M would make M dependent. *maximal independent set*

Note that if $M \subseteq V$ is a maximal independent set, then any vertex in $V \setminus M$ has a neighbor in M. Finding a maximal independent set is easy and can be done by using a greedy approach. A graph can have many different maximal independent sets.

Definition 1.18 (Maximum independent set, Independence number)**.** An independent set with a maximum number of vertices is called a *maximum independent set*. Its size is the *independence number* of the graph, $\alpha(G)$. *maximum independent set*

Finding a maximum independent set is generally hard. It is easy to show via a counterexample that greedy algorithms can not be used to find maximum independent sets.

Remark. Generally, the word *maximal* means, that we cannot increase the current solution without breaking the definition, while *maximum* means that the solution is the largest one possible.

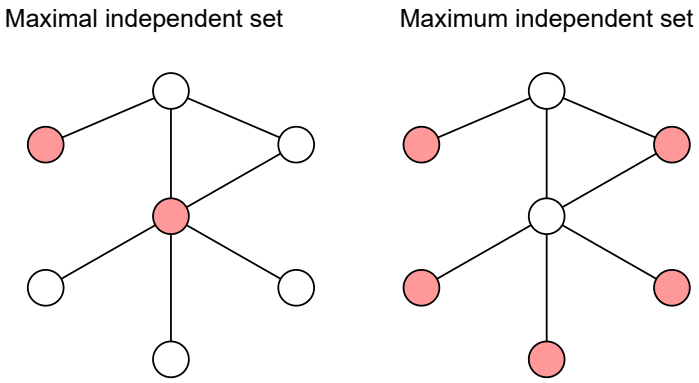


Figure 1.10: We cannot add any more vertices to the graph on the left, as all unchosen vertices already have a red-colored neighbor. There exists a larger independent set, the one on the right, which means that the independent set on the left is not maximum.

The coloring problem can also be defined for edges.

Definition 1.19. A *proper edge k-coloring* is a function $\phi : E \to \{1, \ldots, k\}$ such that $\phi(e) \neq \phi(f)$ if e and f share an endpoint. *edge coloring*

A subset of edges of the graph that do not share endpoints is called a *matching*. Similarly to the case with vertices, a proper edge coloring defines matchings, consisting of the edges colored with the same color. *matching*

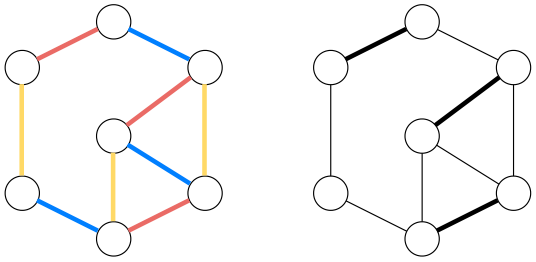


Figure 1.11: A proper edge coloring and a matching corresponding to red-colored edges. The matching happens to be maximal and maximum.

1.5 Some useful graph families

A set of graphs sharing a certain property is called a *family of graphs*. Some graph families are so widely used that we have adopted a standard notation for them. We have already introduced the graph families of n-paths P_n, n-cycles C_n, trees, and forests.

complete graph

One of the widely used graph families is the family of *complete graphs*. A complete graph of n vertices is denoted by K_n and its edge set consists of all possible edges. Complete graphs are a good source of counterexamples and can sometimes act as corner cases for graph algorithms.

bipartite graph

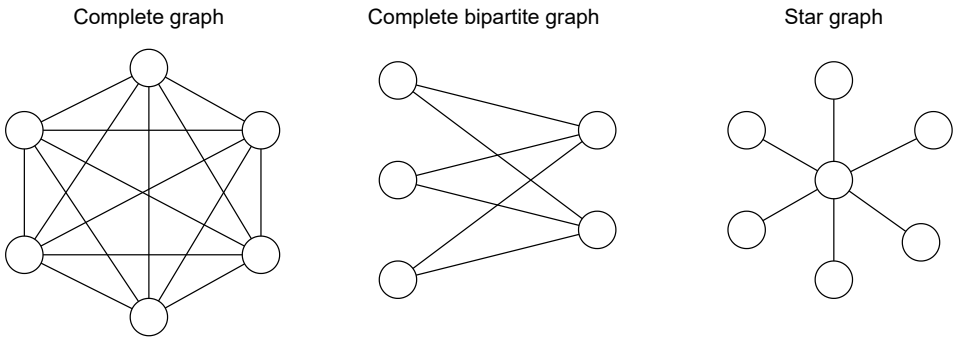
A graph $G = (V, E)$ is *bipartite*, if the set of vertices can be divided into two sets $V = A \cup B$ such that $A \cap B = \emptyset$, and all the edges in the graph are of form $uv \in E$, where $u \in A$ and $v \in B$. In other words, A and B are independent sets. Equivalently, bipartite graphs can be defined as graphs with chromatic number two or graphs that do not contain any odd-lengthed cycles. In a *complete bipartite graph* the set of edges contains all possible edges, ie.

$$E = \{uv \mid u \in A, v \in B\}.$$

A complete bipartite graph is denoted by $K_{n,m}$, where $|A| = n$ and $|B| = m$.

star graph

Star graphs $K_{1,n}$ are complete bipartite graphs where one of the sets contains only one vertex. Star graphs form a great example of the looseness of the bound introduced in Lemma 1.3: the maximum degree of a star graph is n, but its chromatic number is two.



Another important graph type is the family of *interval graphs*. They are used to represent the pairwise intersections of a given collection of intervals. Interval graphs can be used to model the scheduling problem from Example 1.2.

Definition 1.20 (Interval graph)**.** An *interval graph* is a graph defined by intervals $I_1, \ldots, I_n$ so that each vertex of the graph corresponds to one interval. The intervals are of form $I_k = [a, b]$, where $a < b$ and $a, b \in \mathbb{R}$. The edges satisfy *interval graph*

$$\{i, j\} \in E \iff I_i \cap I_j \neq \emptyset.$$

In other words, an interval graph represents how the intervals overlap with each other. The graph coloring problem of interval graphs has the following nice concrete interpretation.

Example 1.2. Let the intervals $A, \ldots, B$ in Figure 1.12 represent the schedules of lectures in a university. The minimum number of lecture halls needed for the lectures is equal to the chromatic number of the interval graph which is three.

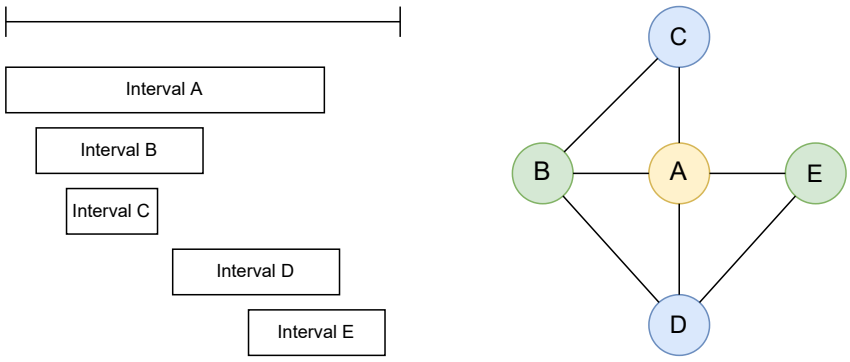


Figure 1.12: Intervals A, B, C, D, and E represented as an interval graph.

2 Stable matching problem

Now that we know the basics of graph theory, we use the terminology in practice to designing an algorithm for a graph-related problem. Mathematically, the problem we are solving is to find maximum a matching for bipartite graphs, but there exists a real-life interpretation for the problem. Practically, the nodes can represent any entities that we need to pair up, like employees and available jobs. Furthermore, each node has a preference list for the nodes of the opposite side. The goal is to find a matching where everybody is as happy as possible.

2.1 Formal definition

Let us recall the definition of a matching.

Definition 2.1 (Matching)**.** Let $G = (V, E)$ be a graph. A subset of the edges $M \subseteq E$ is a *matching* if no edge in M shares an endpoint with another edge in M. *matching*

Practically, this just means pairing up the vertices using the edges of the graph. Even though the stable matching problem can be defined for arbitrary graphs, we are particularly interested in complete bipartite graphs $G = (A \cup B, E)$, where $|A| = |B|$. Additionally, each vertex has a preference list for the vertices it can pair up with.

Example. Let $A = \{i, j, k\}$ and $B = \{a, b, c\}$. Preferences of the vertices in A can be represented as a table or as lists:

	i	j	k
1	a	c	a
2	c	a	c
3	b	b	b

i : $a > c > b$,
j : $c > a > b$,
k : $a > c > b$.

The notation $a >_i b$ means that the vertex i prefers a to b.

We may assume that all the vertices of the opposing set appear in the preference lists and that the preferences are strict.

unstable edge

stable matching

Definition 2.2 (Unstable edge and stable matching)**.** Let M be a matching for the graph $G = (V, E)$. An edge $uv \in E \setminus M$ is *unstable* if both of the vertices u and v prefer each other to their current partners. A matching M is *stable* if there are no unstable edges in the graph.

Intuitively, unstable edges are the edges that should be included in the matching to make everyone happier. In the stable matching problem, we are interested in finding the maximum stable matchings: it would be trivial to find *some* stable matching, as the empty matching $M = \emptyset$ is always stable. We will see that there always exists a maximum stable matching for complete bipartite graphs where A and B have the same size. The same does not hold for general graphs, which can be easily proven with a counterexample (exercise).

2.2 Algorithm description

We start the design process by trying out a naive approach: pick any matching and if there exists an unstable edge, fix it. Repeat the process until a maximum stable matching is reached. More precisely, we repeat the following process:

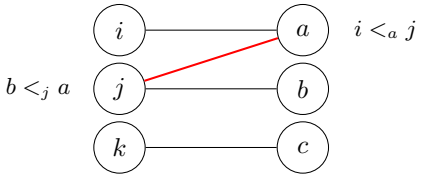
1. If there is an unstable edge $\{u, v\}$, match u and v with each other.
2. Match the previous partners of u and v with each other.

To check the correctness of the algorithm, we need to ensure that it gives the correct output and that it always terminates. Because the algorithm terminates only when we have no stable edges, the output is clearly a stable matching. Moreover, the number of edges in the matching never changes, so the matching is also maximum. Unfortunately, is easy to find a counterexample to prove that the algorithm might never terminate. Consider the following preference lists.

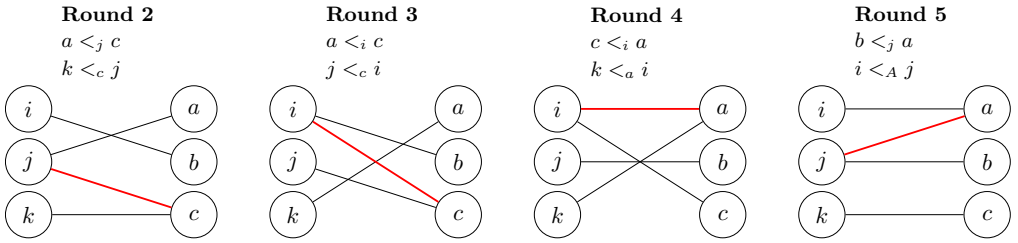
	i	j	k
1	a	c	a
2	c	a	c
3	b	b	b

	a	b	c
1	j	k	i
2	i	j	j
3	k	i	k

Since our graph is a complete bipartite graph, we only draw the edges that are in the matching and possible unstable edges in red. Suppose that we begin with the following configuration. Clearly this is not a stable matching, as the pair (j, a) is unstable.



We then add the edge (j, a) to our matching, and pair the newly unmatched nodes b and i. Continuing the algorithm gives



It turns out that only after four rounds we are back to the initial configuration. This means that the algorithm is going to end up in an infinite loop, repeating the steps above. Our naive algorithm is incorrect, since on some inputs it never terminates.

Another, more elaborate approach starts building the matching in increments by letting all vertices on one side propose their favourite vertices at once, and let the other side choose their favourites among the proposers. This algorithm is known as the Gale-Shapley algorithm.

Algorithm 2: Gale-Shapley Algorithm

Input: Bipartite graph $G = (A \cup B, E)$ and preference lists for the nodes
while $\exists$ *unmatched* $u \in A$ *that has someone to propose to* **do**
1. Unmatched nodes in A propose to their preferred node in B.
2. Nodes in B match with their favourite proposer, possibly leaving their current match.

end

In Algorithm 2, some of the details have been omitted:

1. After $u \in A$ proposes to node $v \in B$, node v gets removed from u's preference list. If u ever proposes again, it will not propose to v again but proposes the its next favorite node.

2. Even though in the pseudocode all of the nodes propose simultaneously, in the practical implementation only one node proposes at a time. This does not change the behaviour of the algorithm.

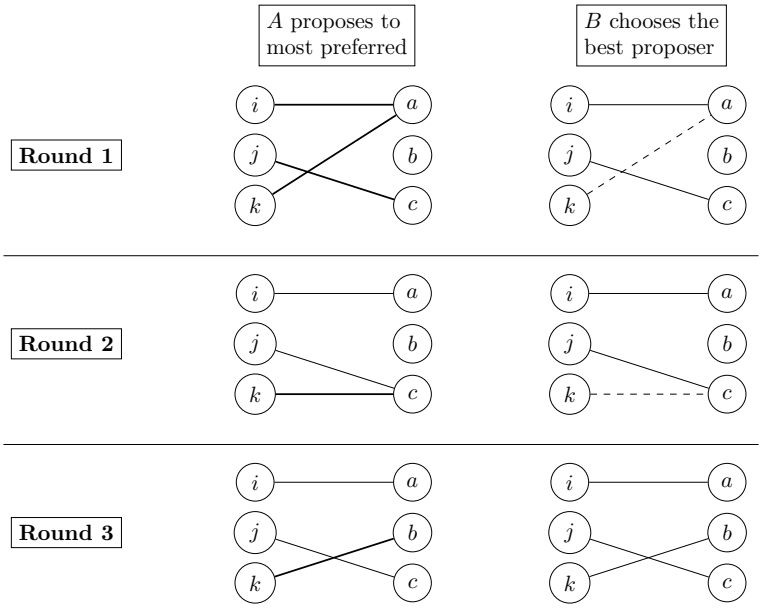
3. The statement "u has someone to propose to" in the while-loop is not really necessary, and the algorithm would be equivalent without it: it can be shown that any unmatched node must have a nonempty preference list. However, the statement makes the proof of termination a bit more straightforward.

Example 2.1. Consider the following preference lists

	i	j	k
1	a	c	a
2	c	b	c
3	b	a	b

	a	b	c
1	j	k	i
2	i	j	j
3	k	i	k

and let $A = \{i, j, k\}$ be the proposing vertices and $B = \{a, b, c\}$ the proposees. The Gale-Shapley algorithm finishes in three rounds:



2.3 Correctness and runtime

To show the correctness of the algorithm, we need to prove the following aspects

1. The algorithm terminates on all inputs.

2. The algorithm outputs a maximum matching.

3. There are no unstable edges in the output, ie. the matching is stable.

Together these claims guarantee that the algorithm always outputs a maximum stable matching. We begin with the first statement.

Lemma 2.1. The Gale-Shapley algorithm always terminates.

Proof. Let $|A| = |B| = n$. Denote the number of nodes the vertex $u \in A$ has not yet proposed to by $\phi(u)$. In the beginning of the algorithm we have $\phi(u) = n$ for all $u \in A$, because no one has yet proposed. Denote the total number of nodes left in the preference lists by $\Phi = \sum_{u \in A} \phi(u) = n^2$. Whenever a node proposes, a vertex will be removed from its preference list, and Φ reduces by one. Each round at least one node must propose: if there are no proposers, all the vertices are already matched and the algorithm terminates. This means that each round Φ reduces by at least one, and after at most n^2 rounds it reaches zero. This means that all the preference lists are empty, and the algorithm terminates. □

This analysis also gives an upper bound for the number of rounds in the algorithm, $O(n^2)$. It can be shown that for some configurations only one vertex will propose each round and hence the number of rounds is actually $\Omega(n^2)$. For analysing the runtime, we need to estimate the runtime of one round. Each round we need to

- Find an unmatched node $u \in A$ and let it propose to $v \in B$.
- Either match u with v or leave u unmatched.

While the second step runs in $O(1)$ time, the runtime of the first step depends on the implementation. In a naïve implementation it takes linear time to determine which nodes are unmatched, and the runtime would be $\Theta(n^3)$. If the algorithm is implemented so that the unmatched nodes are easily accessible, we may reach the runtime of $\Theta(n^2)$.

Now that we know that the algorithm terminates, we can move on to verify that the output really is a maximum matching.

Lemma 2.2. The Gale-Shapley algorithm outputs a maximum matching.

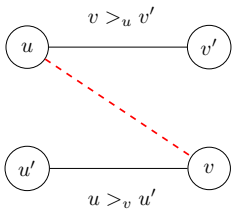
Proof. We need to show that each node is matched to *exactly* one node. After each iteration of the algorithm, each vertex is matched to at most one vertex: each vertex in A proposes at most one vertex, and each vertex in B accepts only one proposal at a time. Suppose that in the end of the algorithm, there exists an unmatched node $u \in A$. Because $|A| = |B|$, this means that there must be an unmatched $v \in B$. Because v prefers anyone to being alone, it must never have been proposed to. This is a contradiction, because the algorithm has ended and u must have exhausted its preference list, and proposed to v at some point. □

Now we are ready to finish the proof of correctness by showing that the matching is

also stable.

Lemma 2.3. The Gale-Shapley algorithm outputs a stable matching.

Proof. Suppose for contradiction that we have an unstable edge $uv \in E$ that is not in the matching. By definition, this means that v prefers u to its current partner. However, if u had proposed to v at any point of the algorithm, v would have left its current partner for u. This means that u never proposed v. This is a contradiction, because u prefers v to its current partner, and u is supposed to propose the nodes in the order of preference.



□

2 Recursion

1 Recursive algorithms

Consider the following definition of the factorial of an integer n.

$$n! = \begin{cases} 1 & \text{when } n = 1, \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

Instead of explicitly defining the value of $n!$, the definition relies on a factorial of a smaller integer. For instance, to solve the value $5!$, we would first need to solve the value $4!$, but this requires $3!$, which requires $2!$. We then compute $2!$ using the base case $2! = 2 \cdot 1! = 2$, and the chain of function calls terminates returning $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

This approach, where the definition refers to itself, is called *recursion*. Algorithmically, the idea is to break a problem into subproblems, solve them recursively, and then combine them to solve the original problem. The subproblems must have the same problem description as the original problem – this allows us to solve the subproblems by letting the algorithm call itself. For the factorial, a recursive algorithm would take the following form.

Algorithm 3: Factorial(n)

```
if n ≤ 1 then
    return n
else
    return n·Factorial(n − 1)
```

Generally, to use the recursive approach, we need to ensure the following things.

1. There must be an explicitly computable base case that terminates the recursive chain of function calls. For example, the base case for the factorial problem is $n = 1$.

2. Each subproblem must be strictly smaller than the original problem. This ensures that we eventually reach the base case. For example, to solve the factorial of n, we need to solve the factorial of the smaller integer $n - 1$.

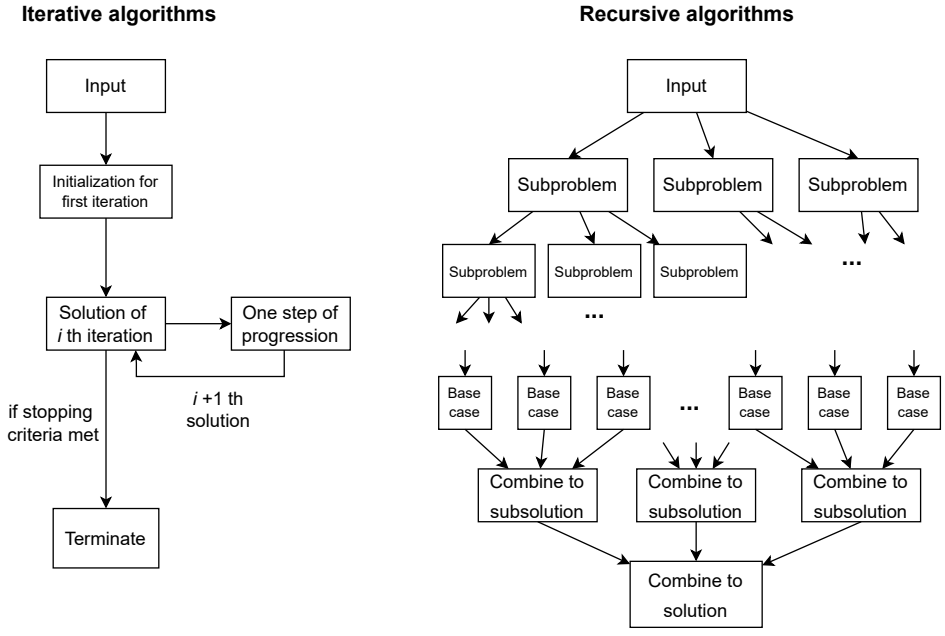
3. It must be relatively easy to combine the subproblems to construct the solution.

Otherwise, the time consumption quickly blows up and using the recursive approach might be counter-productive. For example, in the factorial problem, the subproblems can be combined by computing only one multiplication[1].

To get familiar with recursion, we start by comparing it to the iterative approach. Then we learn about proving the correctness and analyzing runtime of recursive algorithms using two classical examples: towers of Hanoi and merge sort. Finally, we use everything we learned to design an efficient recursive algorithm for multiplying large integers.

1.1 Recursion vs. iteration

Recursive and iterative algorithms both attempt to solve the original problem in small steps. The main difference between the approaches is the way they handle the subproblems. An iterative algorithm starts from the base case and step by step increments the solution until we have constructed the solution of the original problem. The recursive approach takes a bit different perspective. We still have a base case to ensure the termination of the algorithm, but now we simply *assume*, that the smaller cases will be solved correctly. Theoretically, we can consider solving the subproblems as a *black box* that we do not really need to implement. Generally, the idea of recursive algorithms is identical to that of inductive proofs: after showing that the base case works, it is enough to prove that we can move from the correctly solved subsolution (induction hypothesis) to the original solution.



[1]Actually, multiplication of arbitrarily large numbers cannot be done in $O(1)$ time. It is still okay to count the number of multiplications when comparing the algorithms to each other. The true runtime of the Fibonacci algorithm is hence higher than $O(n)$. The runtime with the realistic runtime for multiplication is analyzed in the book *Algorihtms* by Jeff Erickson.

Example 1.1. The *Fibonacci sequence* is defined recursively by

$$F(n) = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ F(n-1) + F(n-2), & \text{otherwise.} \end{cases} \tag{2.1}$$

An iterative algorithm for finding the nth Fibonacci number starts from the base cases $n = 0$ and $n = 1$, and then step by step computes the ith number by

$$F(i) = F(i-1) + F(i-2),$$

until we reach the desired number $F(n)$.

Algorithm 4: Fib(n)

```
if n ≤ 1 then
    return n
else
    a ← 0; b ← 1
    while i ≤ n do
        f ← b + a
        a ← b
        b ← f
        i ← i + 1
    return b
```

In the recursive approach, we can follow precisely the definition given in (2.1) by letting the function call itself.

Algorithm 5: Fib(n)

```
if n ≤ 1 then
    return n
else
    return Fib(n − 1) + Fib(n − 2)
```

2 Correctness of recursive algorithms

2.1 Towers of Hanoi

In the puzzle of Hanoi Towers, we have n different-sized discs and three needles on which the discs can be placed (Figure 2.1). In the beginning, all the discs are on the first needle, ordered so that the heaviest is at the bottom and the lightest is at the top. To goal is to move all the discs to the third needle but

- Only one disc can be moved at a time and
- A disc can never be placed on top of a smaller disc.

As an example, the solution for the case where we have three discs is given below. To

gain more intuition on how to solve the problem, we suggest trying to solve the case with four discs.

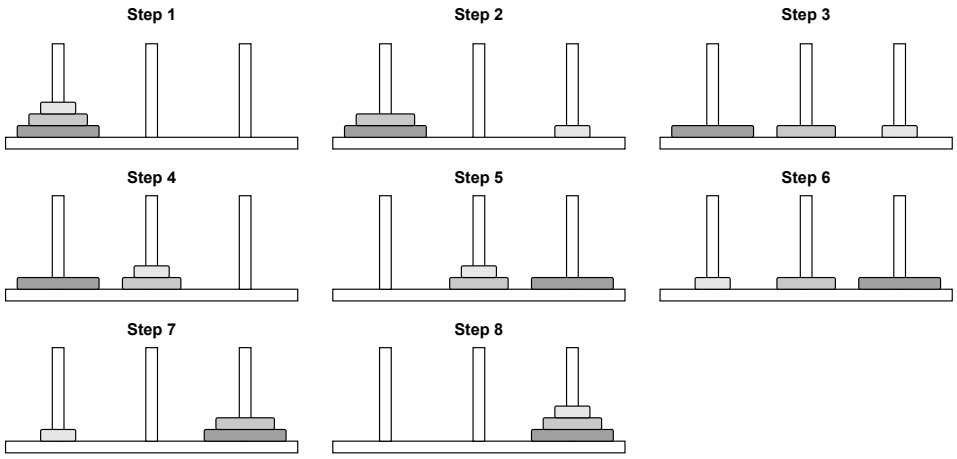


Figure 2.1: Example on how to solve the problem of Hanoi towers with three discs.

Now, let's design a recursive algorithm for solving the problem. We start by finding the base case: when we only have one disc, we can simply move it to the third needle. For n discs we must begin by somehow moving the $n-1$ lighter discs out of the way, and then move the largest disc to the third needle. After that, we want to move the $n-1$ discs to the destination. But how do we move the smallest $n-1$ discs? We do not really need to know, as we already have all the components of our recursive approach: the base case and the strictly smaller subproblem ($n-1$ discs).

The algorithm is given as pseudocode in Algorithm 6. Our function takes four arguments: the number of discs, the location of the initial needle, the goal needle, and the free-to-use extra needle. As we can see, instead of explicitly describing the moves we need to make for $n-1$ discs, we simply *delegate* the task for the recursive call assuming it completes it correctly.

In Algorithm 6, the initial needle is called src, the destination needle $dest$, and the extra needle tmp. Notice that in the recursive call where we want to move the top $n-1$ discs to the extra needle, the needle tmp is our destination, and hence given as the third parameter for the algorithm.

Algorithm 6: Towers(n, src, $dest$, tmp)

```
if n = 1 then
    Move the disc from src to dest.
else
    Towers(n − 1, src, tmp, dest) /* Move n − 1 discs to tmp      */
    Move the largest disc from src to dest.
    Towers(n − 1, tmp, dest, src)/* Move the discs from tmp to dest */
```

To prove the correctness of the algorithm, we need to show that it terminates and that it completes the task without violating the rules in the problem description. All of these claims can be proven with a basic inductive argument on the number of discs.

Lemma 2.1. Algorithm 6 is correct.

Proof. Base case: When $n = 1$, we can simply move the disc from src to $dest$. This does not violate the rules and it clearly terminates.

Induction hypothesis: Suppose the algorithm works as it's supposed to when we have at most k discs.

Induction step: When we have $k + 1$ discs, we enter the else-clause where we call the algorithm for k discs. By the induction hypothesis, the function call terminates and we can move the k discs without violating the rules. The k smallest discs are now in the needle tmp. Moving the largest disc to $dest$ can be now done without violating rules. After that, we can move the k smallest discs to scr by the induction hypothesis. □

2.2 Merge sort

In a sorting problem, we are given an array of comparable objects and the task is to reorder the list in ascending order. Formally, if the input array A is indexed from one to n, we want the output to satisfy

$$A[i] \leq A[j] \iff i \leq j \quad \forall i, j \in \{1, \ldots, n\}.$$

Merge sort is an algorithm that tackles this problem using recursion. It is a classic example of a technique called *divide and conquer*, which practically means simply dividing the original problem into multiple subproblems of roughly even size. When using a recursive approach, the subproblems are then solved recursively. To use this technique, we must design a subroutine for efficiently combining already sorted arrays. This subroutine is called *merging*.

Suppose we have sorted two subarrays, $A[1, \ldots, m]$ and $A[m + 1, \ldots, n]$, and we want to combine them into the fully sorted array $A[1, \ldots, n]$. Because the subarrays are sorted, the smallest element can be found either

- From the beginning of the first subarray, $A[1, \ldots, m]$ at index 1, or
- From the beginning of the second subarray, $A[m + 1, \ldots, n]$ at index $m + 1$.

We move the smallest element to an auxiliary array B and remove it from A. We then continue inspecting the remaining elements of A. The second smallest element is again found from the beginning of the first or the second subarray. We continue this process until all the n elements have been moved to B. After that, we move the elements back to the array A, which now contains all the elements in an increasing order. The merging algorithm is described in Algorithm 7.

Algorithm 7: Merge($A[1 \dots n]$, m)

```
Initialize B[1...n]
A_1 ← A[1...m]
A_2 ← A[m+1...n]
for k ← 1...n do
    if A_1 is empty then
        B[k] ← A_2[1]
        Remove the first element from A_2.
    else if A_2 is empty then
        B[k] ← A_1[1]
        Remove the first element from A_1.
    else if A_1[1] ≤ A_2[1] then
        B[k] ← A_1[1]
        Remove the first element from A_1.
    else
        B[k] ← A_2[1]
        Remove the first element from A_2.
/* Move the elements back to A                    */
for k ← 1...n do
    A[k] ← B[k]
```

Lemma 2.2. Algorithm 7 merges the sorted subarrays $A[1, \dots, m]$ and $A[m+1, \dots, n]$ correctly.

Proof. We will prove this using induction on the number of steps taken in the for-loop. Our induction hypothesis is, that after k steps

1. The first k items in $B[1, \dots, k]$ will be sorted.
2. The first k elements $B[1, \dots, k]$ are smaller or equal to the remaining elements in A_1 and A_2.

If this hypothesis holds, then after n steps, the array B will be fully sorted and then moved back to A.

Base case: When $k = 1$, the array B only contains one element, and hence it is sorted. The element we added is $\min(A_1[1], A_2[1])$. Because $A_1[1] = \min(A_1)$ and $A_2[1] = \min(A_2)$, we have

$$B[1] = \min(A_1, A_2).$$

Inductive step: Consider the $(k+1)$th step of the algorithm. Adding *any* element from A_1 or A_2 will keep $B[1, \dots, k+1]$ sorted due to the second requirement of the induction hypothesis. Adding the smallest remaining element, $\min(A_1, A_2)$, ensures that the second requirement in the hypothesis holds after the $(k+1)$th step. If one of the arrays is already empty, the smallest element can be found from the first index of the nonempty array. Otherwise the smallest element is $\min(A_1[1], A_2[1])$, as the subarrays A_1 and A_2 are sorted.

□

Now that we are able to merge sorted arrays, we can introduce the recursive algorithm. Our base case is when $n = 1$: then the array is already sorted and ready to be returned. Otherwise, we divide the array into two subarrays of equal length and recursively ask our algorithm to sort them. The sorted subarrays will then be combined using Algorithm 7.

Algorithm 8: MergeSort($A[1 \dots n]$)

```
if n ≤ 1 then
    return
else
    m ← ⌊n/2⌋
    MergeSort(A[1 ... m])
    MergeSort(A[m + 1 ... n])
    Merge(A[1 ... n], m)
```

Lemma 2.3. Algorithm 8 is correct.

Proof. The correctness here means that the array A is sorted after the execution of the algorithm and that the algorithm always terminates. Both of these can be handled simultaneously using induction. We will use induction on the size of the array.

Base case: When $n = 1$, the array is already sorted and the algorithm works correctly by doing nothing.

Induction hypothesis: Suppose the algorithm sorts A and terminates for all $1 \leq n \leq k$.

Induction step: Let $n = k + 1$. The algorithm divides the $k + 1$ elements in two parts, one with size $\lfloor \frac{k+1}{2} \rfloor$ and other of size $\lceil \frac{k+1}{2} \rceil$. We have an upper bound

$$\left\lceil \frac{k+1}{2} \right\rceil \leq \frac{k+2}{2} = \frac{k}{2} + 1 \leq \frac{k}{2} + \frac{k}{2} = k,$$

whenever $k \geq 2$, and when $k = 1$, we have $\frac{k+1}{2} = 1$. Hence both of the recursive calls have a size at most k, and by our induction hypothesis, will be sorted correctly. Then, the two sorted subarrays will be merged to the sorted array $A[1, \dots, k + 1]$. As we already proved that the merge algorithm is correct, we are done. □

2.3 Runtime analysis of recursive algorithms

There are many different ways to analyze the runtime of recursive algorithms. Here, we present two of the main tools for this purpose.

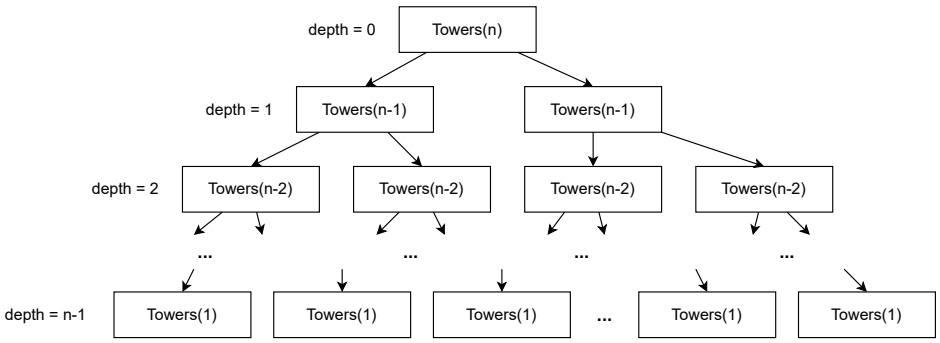
Recursion trees

Since our algorithm calls itself, to analyze the runtime we need to know

1. What is the time consumption between the function calls?
2. How many times the algorithm calls itself?

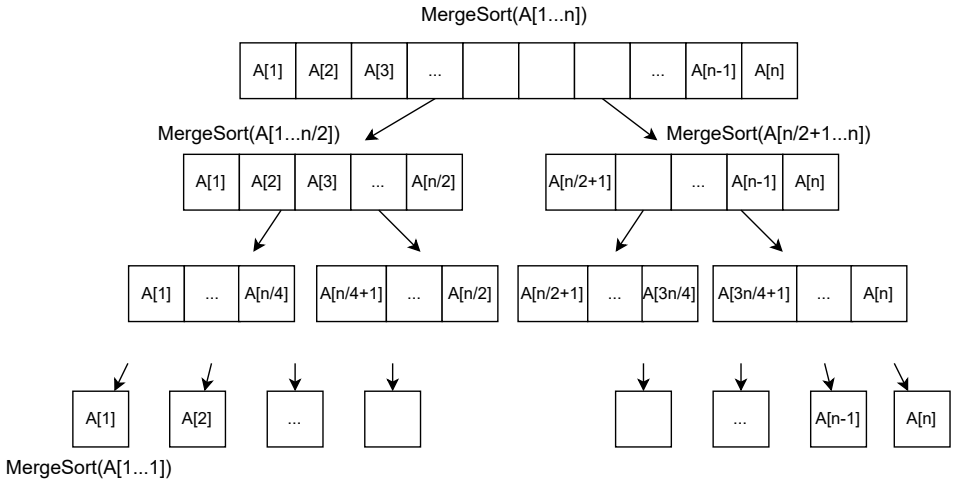
Usually, the former is relatively straightforward. For example, in the merge sort algorithm, combining subproblems, namely merging, takes $\Theta(n)$ time. To find how many recursive calls are hidden inside each subproblem, we use the concept of *recursion trees*. A recursion tree is a model, where the nodes represent function calls and arrows represent dependencies between them, hence allowing us to estimate the total number of function calls.

Example 2.1 (Hanoi Towers)**.** Let's draw the recursion tree for the problem of Hanoi towers. Each function `Tower(n)` will call the function `Tower(n-1)` twice. Recursively, both of the calls will then call `Tower(n-2)` twice and so on, until we reach the base case. This gives us the following recursion tree:



This is a complete binary tree with depth $n - 1$, so the total number of nodes is $1 + 2 + \cdots + 2^{n-1} = 2^n - 1$. As each node only performs one move, the number of moves is $O(2^n)$.

Example 2.2 (Merge sort)**.** We assume that the size of the array is a power of two so that we do not need to worry about rounding. In each call, the algorithm divides the array in two until the array reaches size one. This process can be seen below.



Each of the nodes in this figure represents one function call of MergeSort, and inside each of them, the merge function is called once. The time complexity of merging a list of size k is $\Theta(k)$ (left as an exercise). Notice, that on each layer in the recursion tree, the length of the arrays combined is n, and hence each layer takes $\Theta(n)$ time. The depth d of the tree satisfies $2^d = n$ because the bottom is reached after dividing n by two d times. Hence $d = \log n$ and the total runtime of the algorithm is $O(n \log n)$.

Master Theorem

Sometimes drawing the recursion tree is unnecessarily complex. Instead, we can try to describe the runtime by writing a *recurrence relation* for it. As an example, let's write the recurrence relation for the Hanoi towers algorithm. Let $T(n)$ denote the number of moves needed for n discs. In one iteration, the algorithm

- Calls itself with value $n - 1$ (making $T(n - 1)$ moves).
- Moves one disc.
- Calls itself again with value $n - 1$.

Now we can deduce that the function for the runtime is of the form

$$T(n) = T(n - 1) + 1 + T(n - 1) = 2 \cdot T(n - 1) + 1.$$

The function $T(n)$ is still unknown but using the equation, we can attempt to solve the recurrence using induction. To get an idea of where to start, let's compute some values of $T(n)$. When $n = 1$, the algorithm only performs one move. The rest of the values can be computed iteratively.

n	1	2	3	4	$\cdots$	n?
$T(n)$	1	3	7	15	$\cdots$	$2^n - 1$?

In general, when solving recurrence relations, it might be beneficial to compute some of the values. Now that we have an idea of what the recurrence might look like, we can try to prove it using induction.

Lemma 2.4. Let $T(1) = 1$ and $T(n) = 2 \cdot T(n - 1) + 1$. Then $T(n) = 2^n - 1$ for all $n \geq 1$.

Proof. The base case $n = 1$ satisfies $T(1) = 2^1 - 1 = 1$. Now suppose the claim holds for all $n \leq k$. Then

$$T(k + 1) = 2 \cdot T(k) + 1 = 2 \cdot (2^k - 1) + 1 = 2^{k+1} - 1.$$

□

Unfortunately, it is not always easy to solve the recurrence relation using a basic inductive argument. The following theorem is useful whenever the recurrence relation takes a specific form.

Theorem 2.1 (Master Theorem)**.** Consider the following recurrence relation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where $a \geq 1$ and $b > 1$.

- If $f(n) = O(n^c)$ for some $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- If $f(n) = \Omega(n^c)$ for some $c > \log_b a$, then $T(n) = \Theta(f(n))$.

Remark. Remember that $\log_b a = \frac{\ln a}{\ln b}$.

Usually, the recurrence relations of divide and conquer algorithms take the appropriate form so that we can use the theorem. Practically, the constant a represents how many times the function is called in one step, the constant b tells the relative size of the subproblem, and $f(n)$ is the time consumption of everything else inside one function call. To get familiar with this theorem, let's use it to solve the recurrence relation for the runtime of merge sort.

Example 2.3 (Merge sort)**.** Let's find the recurrence relation for merge sort. To simplify the analysis, we assume that n is a power of two. For an instance of size n, merge sort will

1. Call itself for two instances of size $\frac{n}{2}$ and $\frac{n}{2}$.
2. Call merge function to construct a sorted list of size n.

The for-loop in merge sort iterates over $i = 1, \ldots, n$, and hence runs in $\Theta(n)$ time. Hence,

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n).$$

Now we can use the Master Theorem with parameters $a = b = 2$. We have $f(n) = \Theta(n^{\log_2 2}) = \Theta(n)$, which gives $T(n) = \Theta(n \log n)$.

3 Application: Integer multiplication problem

In this section, we design a fast algorithm for finding the product of two large integers, one with m digits and the other with n digits. We start with a naive implementation to get more familiar with the time consumption of this task, and then use recursion to design an efficient algorithm.

3.1 Naive implementation

In elementary school, we learned to multiply large numbers digit by digit, each step "adding appropriately many zeros", and then by summing the subresults we get the product. This allows us to reduce the difficult problem of multiplying arbitrarily large numbers into many multiplications of small numbers. For instance, the multiplication

$724 \cdot 29$ would be broken into the following small multiplications and summations:

$$\begin{aligned} 42 \cdot 29 &= (7 \cdot 10^2 + 2 \cdot 10^1 + 4)(2 \cdot 10 + 9) \\ &= 9 \cdot 4 + 9 \cdot 2 \cdot 10 + 9 \cdot 7 \cdot 10^2 + 2 \cdot 4 \cdot 10 + 2 \cdot 2 \cdot 10^2 + 2 \cdot 7 \cdot 10^3 \\ &= 36 + 180 + 6300 + 80 + 400 + 14000 \\ &= 20996 \end{aligned}$$

It is common to represent the computations so that the numbers lay on top of each other:

$$\begin{array}{r} 724 \\ * \quad 29 \\ \hline 36 \\ 180 \\ 6300 \\ 80 \\ 400 \\ +\ 14000 \\ \hline 20996 \end{array}$$

Let $x = (x_1, \ldots, x_m)$ and $y = (y_1, \ldots, y_n)$ be the digit representations of the numbers we want to multiply. Mathematically our simplification ends up taking the form

$$x \cdot y = \left(\sum_{i=0}^{m-1} 10^i \cdot x_i \right) \cdot \left(\sum_{j=0}^{n-1} 10^j \cdot y_j \right) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} 10^{i+j} \cdot x_i \cdot y_j.$$

We have $m \cdot n$ multiplications to compute, and in the end, we sum these subproblems together (also digit by digit). The multiplications end up dominating the runtime and the algorithm will run in $O(mn)$ time.

This naïve implementation is iterative in nature as we iterate over the digits to compute the product. Could a recursive approach give a better result?

3.2 Recursive approach

Our earlier algorithm relied on the fact that multiplying by powers of ten is relatively easy: we can simply "add zeros" to the end of the number. Consider a number with n digits represented in the array form. Multiplying it by 10^k means moving each of the digits k places further in the array, which takes $O(n)$ time. Now suppose we have two integers, x and y, both having n digits. Assuming n is even, we may represent their product by

$$x \cdot y = (10^{n/2} \cdot a + b) \cdot (10^{n/2} \cdot c + d) = 10^n \cdot ac + 10^{n/2} \cdot (ad + bc) + b \cdot d, \tag{2.2}$$

where a represents the $n/2$ most significant digits of x and b consists of the $n/2$ least significant digits. More generally for any $m \geq 0$, we can represent any number in the form

$$x = a \cdot 10^m + b,$$

where $a = \lfloor x/10^m \rfloor$ and $b = x \bmod 10^m$, or in other words b being the m least significant digits and a representing the most significant ones.

Consider x and y with n digits. Then by the formula (2.2), the numbers a, b, c, and d have $n/2$ digits, and we have our subproblems with strictly smaller sizes. We just need to recursively compute $a \cdot c$, $b \cdot c$, $a \cdot d$, and $b \cdot d$, and perform the appropriate shifts and summations.

We can choose the base case to be practically anything with "few enough" digits, and in Algorithm 10 we have chosen one-digit numbers to be small enough.

Algorithm 9: Multiply(x, y, n)

if $n = 1$ **then**
  return $x \cdot y$
else
  $m \leftarrow \lceil \frac{n}{2} \rceil$
  $a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x \bmod 10^m$
  $c \leftarrow \lfloor y/10^m \rfloor$; $d \leftarrow y \bmod 10^m$
  $ac \leftarrow$Multiply(a, c, m)
  $ad \leftarrow$Multiply(a, d, m)
  $bc \leftarrow$Multiply(b, c, m)
  $bd \leftarrow$Multiply(b, d, m)
  return $10^{2m} \cdot ac + 10^m \cdot (ad + bc) + bd$

The correctness of the algorithm follows simply from the earlier algebraic observations. Let's analyze the runtime using recurrence relations. For n digits we have

- Digit manipulations of complexity $O(n)$ for $a, b, c,$ and d.
- Four recursive calls for $n/2$ digits.
- Four additions and two-digit shifts, all within complexity of $O(n)$.

All in all, the recurrence relation takes the following form:

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n).$$

This can be solved using Master Theorem with parameters $a = 4$ and $b = 2$. We have $\log_2 4 = 2$, and $f(n) = O(n)$, and hence $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$. Unfortunately, this is not an improvement compared to earlier. If we want to outperform this algorithm using the divide and conquer method, we should aim to decrease the value of $\log_b a$.

3.3 Karatsuba algorithm

To improve our algorithm, the goal is to reduce the number of recursive calls required. It turns out, that the multiplication can be done using only three recursive calls. Consider the following algebraic identity:

$$ac + bd - (a - b) \cdot (c - d) = ad + bc.$$

To compute $10^{2m} \cdot ac + 10^m \cdot (ad + bc) + bd$, we still need to compute ac and bd as before, but we can now skip computing the two products ad and bc and instead compute the product $(a - b)(c - d)$. Our algorithm ends up taking the following form.

Algorithm 10: Multiply(x, y, n)

if $n = 1$ **then**
  return $x \cdot y$
else
  $m \leftarrow \lceil \frac{n}{2} \rceil$
  $a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x \bmod 10^m$
  $c \leftarrow \lfloor y/10^m \rfloor$; $d \leftarrow y \bmod 10^m$
  $ac \leftarrow$**Multiply(**a, c, m**)**
  $bd \leftarrow$**Multiply(**b, d, m**)**
  $ad + bc \leftarrow ac + bd -$**Multiply(**$a - b, c - d, m$**)**
  return $10^{2m} \cdot ac + 10^m \cdot (ad + bc) + bd$

Now our algorithm performs

- Four digit manipulations in $O(n)$ time, as before.
- Three recursive multiplications with $n/2$ digits.
- Summation and digit shifting, again in $O(n)$ time.

This gives us the recurrence relation

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n),$$

and by Master Theorem we get $T(n) = \Theta(n^{log_2 3}) \approx \Theta(n^{1.585})$, which is an improvement.

3 Dynamic programming

1 Dynamic programming

So far we have mainly focused on the time complexity of the algorithms and given the memory consumption of the algorithms only a little thought. This week, we learn tools to reduce the runtime of recursive algorithms by using some extra memory in a clever way.

When running recursive algorithms, we sometimes end up processing the same function call multiple times. The basic idea of dynamic programming is to improve the algorithm by avoiding repetitive function calls. This is done by storing already solved subproblems in memory. Then the memory consumption can be possibly reduced by carefully inspecting the dependencies of the function calls in the recursive algorithm.

We will start by revisiting the recursive Fibonacci algorithm from Week 2 and identify the problems in the design. We will then solve the edit distance problem step by step using the dynamic approach.

1.1 Repetitive computations and memoization

In the second week, we introduced the following recursive algorithm for computing Fibonacci numbers.

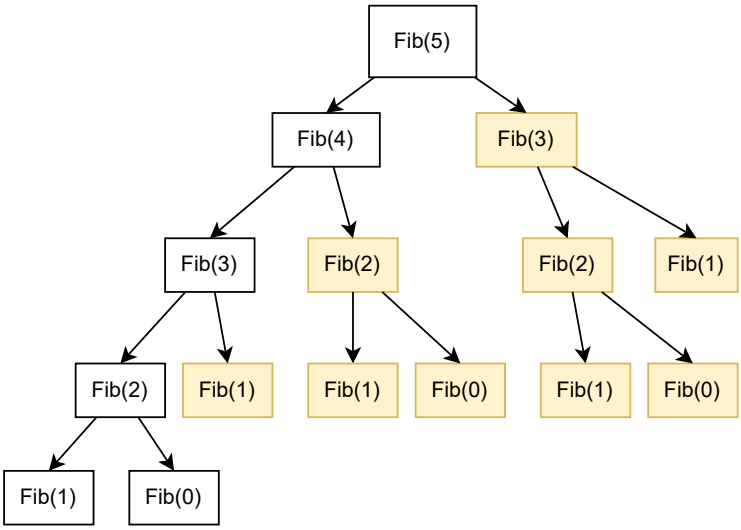
Algorithm 11: Fibonacci(n)

```
if n ≤ 1 then
    return n
else
    return Fibonacci(n − 1) + Fibonacci(n − 2)
end
```

The recurrence relation correspoding the algorithm is

$$T(n) = T(n-1) + T(n-2) \geq 2 \cdot T(n-2),$$

which for the base cases $T(0) = T(1) = 1$ gives $T(n) \geq O(2^{n/2})$. This is significantly worse than the runtime of the iterative version which runs in $O(n)$ time. To see why this happens, we can take a peek at the recursion tree of the algorithm.



As highlighted in the recursion tree, some of the values are computed multiple times. We only need to compute $n + 1$ nodes of the tree to solve the original problem. If we save these values in an array, each of the repetitive branches can be handled in $O(1)$ time, and the time consumption drops to $O(n)$ – which is the same as the runtime of the iterative approach. The memory consumption of the modified algorithm increased from $O(1)$ to $O(n)$, but the runtime has decreased from exponential to linear. The technique where we reduce the number of repetitive function calls by reserving memory is called *memoization*.

The modified algorithm is illustrated in Algorithm 12. The recursion tree of this algorithm can be constructed by removing the yellow nodes from the old recursion tree.

Algorithm 12: Fibonacci(n)

Initialize a global array $A[0 \dots n]$ for the subproblems.
$A[0] \leftarrow 0$; $A[1] \leftarrow 1$
def Fib(k)**:**
  if k *is already computed* **then**
    return $A[k]$
  else
    $A[k] \leftarrow$ Fib$(k-1)$ + Fib$(k-2)$
    return $A[k]$
  end
return Fib(n)

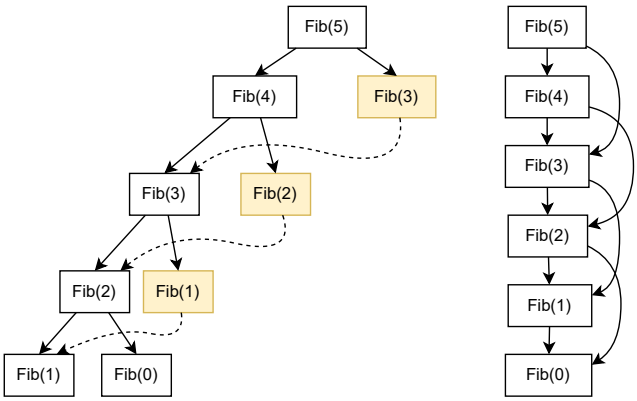
1.2 Directed acyclic graphs

In addition to recursion trees, the dependencies of function calls can be represented as directed acyclic[1] graphs (DAGs). They are a great tool to visualize the process of removing repetitive function calls. While a recursion tree allows repetitions in the function calls, in a DAG each function call with specific parameters only appears once.

Constructing a DAG based on a recursion tree can be done in two steps:

1. Remove the branches that are below repetitive function calls.
2. If a function call appears more than once, combine it into a single node. Add arrows from all the nodes depending on this function call.

In the figure below one can see how the dependencies of the recursive Fibonacci algorithm are represented as a DAG.



1.3 Reducing memory consumption

In the DAG of the Fibonacci algorithm, we can see that the call Fib(5) only depends on the nodes Fib(4) and Fib(3). This means that if we compute the values Fib(i) for $i = 1, \ldots, n$ starting from the lowest, we do not need to save all of the values in the memory: only storing the two highest ones suffices, which gives the memory consumption $O(1)$. Notice that this change makes our dynamic algorithm identical to the iterative one.

When trying to improve the memory consumption of the algorithm, inspecting the DAG of the algorithm is usually beneficial. It also allows us to determine the optimal order to compute the subproblems. For instance, consider the following grid-like DAG for a problem where the goal is to compute the node (m, n).

[1]The function dependencies cannot contain cycles, because otherwise, the recursive calls might produce an infinite loop.

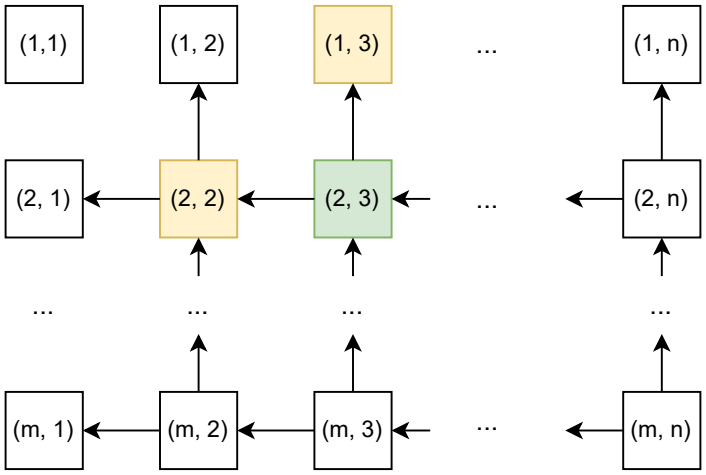


Figure 3.1: The DAG of a hypothetical algorithm. For instance, the node $(2, 3)$ depends on the nodes $(2, 2)$ and $(1, 3)$.

The nodes on the top row and on the leftmost column are the base cases as they do not depend on other nodes. They can be instantly filled into the array containing the intermediate values. Otherwise, the node (i, j) depends on nodes $(i-1, j)$ and $(i, j-1)$.

Because the nodes do not depend on the nodes above them or on the right side of them, we may compute the values in two ways:

1. Column by column from top to bottom and left to right.
2. Row by row from left to right and top to bottom.

In a naive implementation, we can store all the values in a $m \times n$ array and the memory consumption would be $O(mn)$. However, when we choose to compute the values column by column, we only need the latest column to compute the new column, and the memory consumption can be reduced to $O(m)$. Similarly, if we choose the second option, we only need to store the ith row to compute the $(i+1)$th row, and the memory consumption is $O(n)$.

2 Application: Edit distance

2.1 Problem description

Th edit distance problem originates from biology, from the need to efficiently compare long DNA sequences. If two species are evolutionarily close to each other, the number of mutations in the sequence should be relatively low. DNA sequences are strings consisting of letters "A", "C", "G", and "T", and mutations are modeled as different modifications on the string. We consider three types of modifications:

1. One letter disappears from the string, eg. "$ACCT$" becomes "ACT".

2. One letter changes, eg. "$ACCT$" becomes "$ACGT$".

3. One letter appears in the string, eg. "$ACCT$" becomes "$ACCTA$".

The distance between two strings a and b would then be the minimum number of modifications, or *edits*, needed to change a into b. For example, the minimum distance between ACG and $ATGG$ is two, as we need at least two edits:

$$ACG \overset{C \to T}{\longrightarrow} ATG \overset{\text{add } G}{\longrightarrow} ATGG.$$

Notice that the edit sequences are not unique: for the words ACG and $ATGG$ there exists alternative edit sequence

$$ACG \overset{\text{remove } C}{\longrightarrow} AG \overset{\text{add } T}{\longrightarrow} ATG \overset{\text{add } G}{\longrightarrow} ATGG.$$

There can be even multiple different minimal sequences. To ensure that the minimum has been found, we could test all possible edit sequences, but this is not a realistic approach as we could generally generate arbitrarily long edit sequences. Furthermore, in the applications of edit distance, we are usually dealing with enormous words, which makes efficiency even more crucial. Luckily, there is a way to reduce the number of possible choices.

2.2 Recursive algorithm

We approach the problem by considering the solution as a sequence of edits. We edit the string a letter by letter so that the rightmost edit is performed last. Let's take a look at possible edit sequences. The last edit must be either deletion, substitution, or insertion. As an example, consider the strings "$ACCTG$" and "$AAGT$". We have three possible edit sequences

$$\begin{aligned}
&\text{delete :} && ACCTG \overset{?}{\to} \ldots \overset{?}{\to} AAGT\underline{G} \overset{\text{remove } G}{\longrightarrow} AAGT \\
&\text{substitute :} && ACCTG \overset{?}{\to} \ldots \overset{?}{\to} AAG\underline{G} \overset{G \to T}{\longrightarrow} AAGT \\
&\text{insert :} && ACCTG \overset{?}{\to} \ldots \overset{?}{\to} AAG_ \overset{\text{insert } T}{\longrightarrow} AAGT
\end{aligned}$$

Notice that the type of the last edit allows us the deduce how the string looks before the last edit: for instance, in the case of insertion, the string is of form $b[1 \ldots n-1]$ before the last edit and the edits marked with question marks should solve the problem $a[1 \ldots m] \to b[1 \ldots n-1]$.

In the recursive algorithm, we solve the unknown edits by calling the function itself. For the example above, this means solving the following subproblems.

1. Deletion: what is the distance between $ACCT$ and $AAGT$?

2. Substitution: what is the distance between $ACCT$ and AAG?

3. Insertion: what is the distance between $ACCTG$ and AAG?

The optimum must be one of these three options, namely the sequence with the least edits. The strings in the subproblems are strictly smaller than in the original problem,

so using recursion is reasonable. The base case is reached when either of the strings is empty:

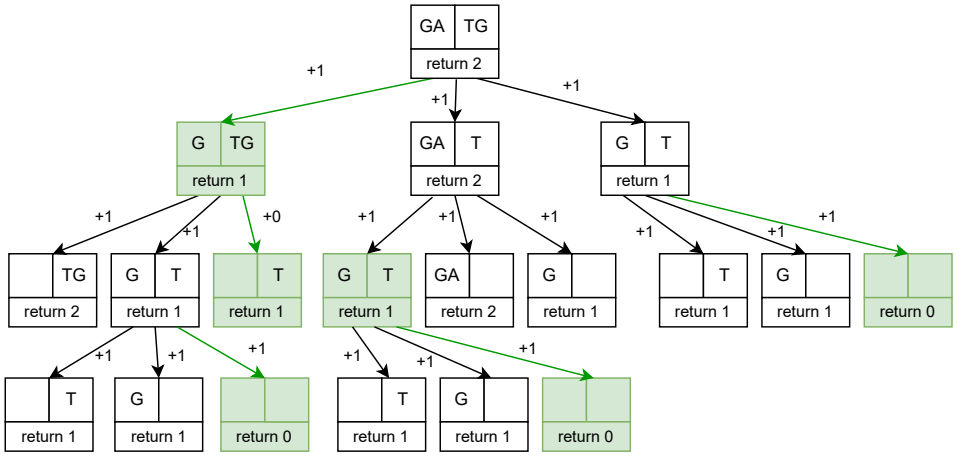
1. If a is empty, the optimal way to construct b is by inserting all letters of b. The cost is the length of b.
2. If b is empty, we construct b by removing all the letters of a. The cost is the length of a.

This recursive approach is represented in Algorithm 13. Note that when substituting the last letter of a with the last letter of b, it can happen that $a[m] = b[n]$, and the substitution is not needed. In those cases, the edit distance is simply the edit distance between the strings $a[1 \dots m-1]$ and $b[1 \dots n-1]$.

Algorithm 13: Edit(a, b)

```
m ← length of a
n ← length of b
if m = 0 then
    return n                    /* insert all letters of b          */
else if n = 0 then
    return m                    /* remove all letters of a          */
else
    del ←Edit(a[1 ... m − 1], b) + 1
    ins ←Edit(a, b[1 ... n − 1]) + 1
    if a[m] = b[n] then
        sub ←Edit(a[1 ... m − 1], b[1 ... n − 1])/* free substitution */
    else
        sub ←Edit(a[1 ... m − 1], b[1 ... n − 1]) + 1
    return min(del, ins, sub)
```

Example 2.1. The recursion tree of the algorithm with inputs GA and TG is given below. Each node represents a function call and the output of the function is given below each node. The minimal function call in each branch is highlighted in green. One of the optimal paths is $GA \to TGA \to TG$, taking two edits.



2.3 Correctness of the recursive algorithm

As usual, we prove the correctness of the recursive algorithm using induction. This time, have two parameters to take care of in our inductive proof: the length of a and the length of b. To make the proof as simple as possible, we split it into two lemmas.

In the first part of the proof, we show that if the recursive calls return the correct output, the original problem will be solved correctly. Then, we prove the correctness using nested induction.

To simplify the notation, we will from now on denote $\mathrm{Edit}(a[1 \ldots i], b[1 \ldots j])$ by $\mathrm{Edit}(i, j)$.

Lemma 2.1. Suppose the algorithm computes correctly the subproblems $\mathrm{Edit}(i-1, j)$, $\mathrm{Edit}(i, j-1)$ and $\mathrm{Edit}(i-1, j-1)$. Then $\mathrm{Edit}(i, j)$ is computed correctly.

Proof. We begin by showing that the algorithm computes correctly the values del, ins, and rep.

- Deletion. Consider the optimal edit sequence ending in deletion,

  $$b[1 \ldots j]a[i] \to b[1 \ldots j].$$

  The subsequence $a[1 \ldots i-1] \to b[1 \ldots j]$ must be optimal, too. By the induction hypothesis, its lenght is $\mathrm{Edit}(i-1, j)$, and hence the length of the whole sequence is $\mathrm{Edit}(i-1, j) + 1$.

- Insertion. Consider the optimal edit sequence ending in insertion,

  $$b[1 \ldots j-1] \to b[1 \ldots j].$$

  The subsequence $a[1 \ldots i] \to b[1 \ldots j-1]$ must be optimal and has length $\mathrm{Edit}(i, j-1)$. Hence the length of the whole sequence is $\mathrm{Edit}(i, j-1) + 1$.

- Substitution. The optimal edit sequence ending in substitution takes the form

  $$b[1 \ldots j-1]a[i] \to b[1 \ldots j].$$

The first edits transform $a[1 \dots i-1]$ into $b[1 \dots j-1]$, which has length $\mathrm{Edit}(i-1, j-1)$. The last step does not need a substitution if the letters $a[i]$ and $b[j]$ are the same, in which case the length of the whole sequence is $\mathrm{Edit}(i-1, j-1)$. If $a[i] \neq b[j]$, the sequence has length $\mathrm{Edit}(i-1, j-1)+1$.

The edit distance between a and b is the length of the shortest sequence of the three options. This value is $\min\{del, ins, sub\}$ which is what the algorithm outputs.

□

The following proof uses *nested induction* which here means that we first use induction on the length of a and then, in the induction step, we use induction on the length of b. The complexity of the proof comes from the fact, that the recursion in the algorithm depends on both the length of a and b.

Lemma 2.2. Algorithm 13 outputs the edit distance between a and b.

Proof. Base case: Let $|a| = 0$. Then we need at least $|b|$ insertions, as otherwise, we cannot have $|b|$ letters in the end result. Hence the edit distance is $|b|$.

Induction hypothesis: Suppose that for all $|a| < k$ and all b the algorithm works.

Induction step: Let $|a| = k$. We now use induction on the length of b.

- Base case: When $|b| = 0$, we must delete all letters, so the edit distance is $|a|$.
- Ind. hypothesis: Suppose the algorithm works when $|b| < \ell$, ie. $\mathrm{Edit}(k, \ell-1)$ is correct.
- Ind. step: By the induction hypothesis, $\mathrm{Edit}(k-1, \ell)$ and $\mathrm{Edit}(k-1, \ell-1)$ are correct, and hence by Lemma 2.1, $\mathrm{Edit}(k, \ell)$ is correct.

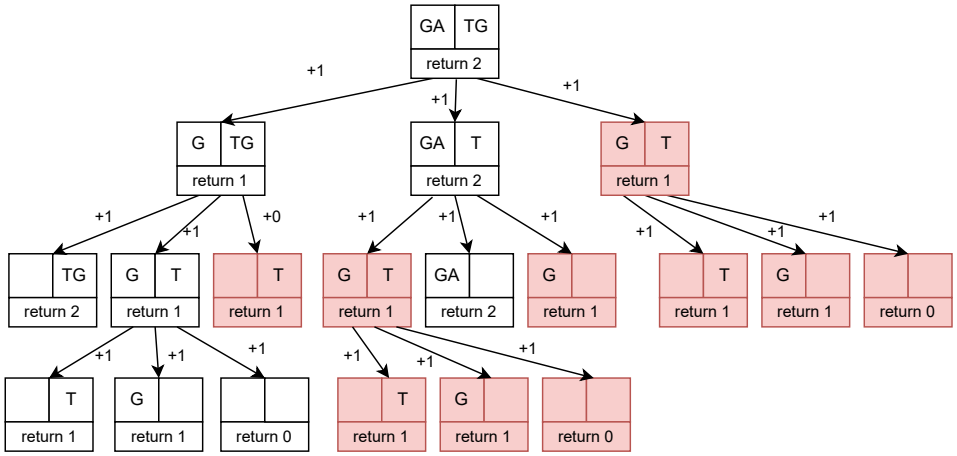
Hence the algorithm works for all b when $|a| = k$. By induction on the length of a, the algorithm works for all a and b.

□

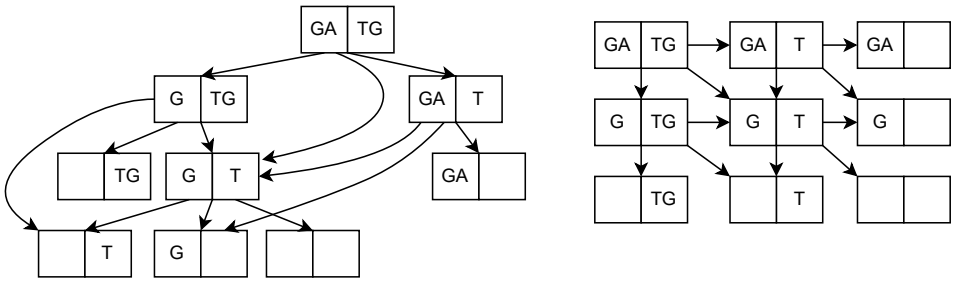
We will not prove separately that the algorithm terminates. It follows from the subproblems having strictly smaller sizes and can be proven formally by using a similar inductive argument as in Lemmas 2.1 and 2.2.

2.4 Dynamic programming improvement

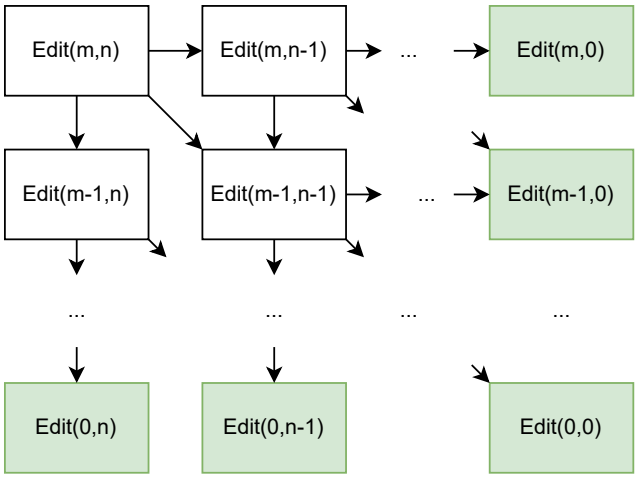
As we saw in Example 2.1, already with strings of length two we have lots of repetitive function calls. The repetitions are highlighted below.



The DAG corresponding to this recursion tree is given below. The nodes can be re-ordered so that the grid-like structure of the DAG is more apparent.



For general a and b the DAG takes the following form.



To solve the problem, the required function calls are Edit(i, j) with $i = 0, \ldots, m$ and $j = 0, \ldots, n$, which means that we need to fill in a $(m + 1) \cdot (n + 1)$ array to store the intermediate results. The highlighted nodes in the DAG are the base cases, as they do not depend on any function calls. Furthermore, the nodes never depend on nodes below them or on their right side so we can fill in the values either column by column or row by row starting from the base cases.

The dynamic algorithm is represented as pseudocode in Algorithm 14. In the algorithm, we chose to fill in the values row by row.

Algorithm 14: Edit(a, b)

```
n ← length of a; m ← length of b
Initialize array A with (m + 1) · (n + 1) entries.
A[i, 1] ← i for i = 1 ... m + 1.
A[1, j] ← j for j = 1 ... n + 1.
for i ← 2 ... m + 1 do
    for j ← 2 ... n + 1 do
        del ← A[i − 1, j] + 1
        ins ← A[i, j − 1] + 1
        if a[i] = b[j] then
            rep ← A[i − 1, j − 1]
        else
            rep ← A[i − 1, j − 1] + 1
        A[i, j] = min(del, ins, rep)
    end
end
return A[m + 1, n + 1]
```

The correctness of the algorithm follows from equivalence to the recursive algorithm: we changed nothing in the algorithm except the order in which we solve the subproblems.

Let's analyze the runtime and memory consumption of this algorithm. In the algorithm, we

1. Fill in the base cases $A[i, 1]$ and $A[1, j]$ in $O(m + n)$ time.
2. Inside the nested for-loop compute the value $A[i, j]$. This will be done in $O(1)$ time, as we need a constant number of array accesses, arithmetic operations, and comparisons. Hence the whole for-loop takes $O(mn)$ time.

Since the second step dominates the runtime, the algorithm runs in $O(mn)$ time. For the memory consumption, we need space for mn integers in the array A and $O(1)$ auxiliary variables. Hence, the memory consumption of the algorithm is $O(mn)$. On the other hand, we can see from the DAG that to compute a new row, we only need to know the previous row. This allows us to reduce the memory consumption to $O(m)$.

Problem types

In this chapter, we talk about the difference between two types of problem settings: decision problems and optimization problems. As an example, consider the graph coloring problem, where we can set different goals when trying to find a proper coloring for the input graph. All of the following questions can be considered to be graph-coloring problems, but they vary in difficulty:

1. Is there a proper coloring for the given graph?
2. Can the given graph be colored with k colors?
3. What is the minimum number of colors needed to properly color the graph?

For the first question, the answer is always yes: we can assign each vertex a different color to yield a proper $|V|$-coloring. The second question is already non-trivial: we either need to find a proper k-coloring or iterate over all possible k-colorings to prove that a proper one does not exist. The third problem is the hardest one: to answer the question, we need to find the smallest k for which the second problem has a solution. From another point of view, if we know how to solve the third problem, we can immediately answer the second one.

The first two questions in the example concern the existence of a solution. These types of problems, where the task is to answer a yes-or-no question, are called *decision problems*. Examples of decision problems are for instance

- Does this graph have a complete subgraph of size k?
- Does this sudoku have a solution?
- Does this graph have a stable matching?

In the third question concerning the graph coloring, the goal was to find the *best* possible solution. These types of problems are called *optimization problems*. We have already encountered many optimization problems on the course like finding the chromatic number of a graph, computing the edit distance, and constructing a minimum spanning tree for a given graph. So far we have defined these problems using natural language, but in general, there exists a standard way to describe optimization problems mathematically which we will explore next.

Optimization problems

All optimization problems consist of the following three basic elements.

1. *Parameters*. Parameters are used to describe possible solutions to the problem. For example in graph coloring, the parameters are the colors assigned for the vertices. More specifically, each vertex would have a parameter (or variable) representing its color.

2. *Constraints*. Constraints are used to describe the limitations set on the parameters. For example in graph coloring, the constraint is that two neighboring nodes cannot have the same color. A solution is called *feasible* if it satisfies the constraints, but is not necessarily optimal.

3. *Objective*. This means the goal of the problem. It is usually given as a function depending on the parameters. The goal of the optimization problem is to minimize or maximize the value of the objective function. For example when finding an optimum graph coloring, the objective function is the total number of colors used, and we aim at minimizing it.

Mathematically, an optimization problem typically takes the form

$$\begin{aligned} &\text{minimize} && f(x_1, \ldots, x_n) \\ &\text{subject to} && g_i(x_1, \ldots, x_n) \leq 0 \; \forall \; i = 1, \ldots, k. \end{aligned}$$

The function f is the *objective function*, and its value depends on the parameters $x = (x_1, \ldots, x_n)$. We want to find the minimum value for $f(x)$ such that the parameters x satisfy the constraints $g_i(x) \leq 0$ for all $i = 1, \ldots, k$. An *optimum* solution x^* satisfies $f(x^*) \leq f(x)$ for all feasible parameters x, ie. it is the best solution among all feasible solutions. If there are no constraints g_i, the optimization problem is *unconstrained*.

Generally, the constraints are not necessarily inequalities: they can be any statements depending on the parameters, as we can see from the next example.

Example (Optimization problem: graph coloring). Let's formulate the graph coloring problem as a mathematical optimization problem. Our objective is to minimize the number of colors needed to color $G = (V, E)$ while making sure that the coloring is proper. We will assign parameters $x_1, \ldots, x_n$ for the nodes in V, and the parameter x_i represents the color of $v_i \in V$. We want the parameters x_i to only take positive integer values, which gives us the constraint

$$x_i \in \mathbb{N}_{>0}, \quad \forall i = 1, \ldots, n.$$

To ensure that the assigned colors give a proper coloring, we introduce the constraints

$$x_i \neq x_j, \quad \forall i, j \text{ st. } \{v_i, v_j\} \in E.$$

Now, the objective value is the number of colors needed. If the graph is colored optimally, the colors are chosen from the integers $\{1, \ldots, \chi(G)\}$, and the number of colors is the largest value in $x_1, \ldots, x_n$. The optimization problem takes the form

$$\begin{aligned} \text{minimize} \quad & \max\{x_1, \ldots, x_n\} \\ \text{subject to} \quad & x_i \neq x_j \quad \forall i, j \text{ st. } \{v_i, v_j\} \in E, \\ & x_i \in \mathbb{N}_{>0}, \quad \forall i = 1, \ldots, n. \end{aligned}$$

Example (Optimization problem: Knapsack). In the Knapsack problem, the goal is to pack a knapsack with the most valuable items so that the weight capacity C of the knapsack is not exceeded. We have a set of items I and each item $i \in I$ has value $v_i \geq 0$ and weight $w_i \geq 0$. The objective is to choose a subset $S \subseteq I$ that maximizes the total value of the packed items:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq C, \\ & S \subseteq I. \end{aligned}$$

All sets $S \subseteq I$ with $\sum_{i \in S} w_i \leq C$ are feasible, and hence give us a lower bound for the optimal value. To find the optimum set S^*, we must ensure that no other feasible solution has a higher total value. The dynamic approach is especially convenient for this problem as it can be solved recursively by inspecting the last element in I. We have two cases

1. We pack the item n and solve the rest of the problem recursively with the items $\{1, \ldots, n-1\}$ and the capacity $C - w_n$.
2. We do not pack the item n, and solve the rest of the problem recursively with the items $\{1, \ldots, n-1\}$ and the capacity C.

To solve the problem dynamically using memoization, we define the subproblems (I_k, w) for all $k = \{0, 1, \ldots, n\}$ and $w = \{0, 1, \ldots, C\}$ by

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq w, \\ & S \subseteq I_k. \end{aligned}$$

These subproblems are then combined appropriately to solve the original problem which is the subproblem (I_n, C). We solve the problem dynamically in Graded Exercise 3.