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)

   ```
   $m \leftarrow A[1]$
   for $i \leftarrow 1 \dots n$ do
       if $A[i] < m$ then
           $m \leftarrow A[i]$
   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.

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 $f(n) = O(g(n))$. ***big-O-notation***

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.

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 $f(n) = \Omega(g(n))$. ***big-Ω-notation***

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.

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 $E(G)$. *simple undirected graph*

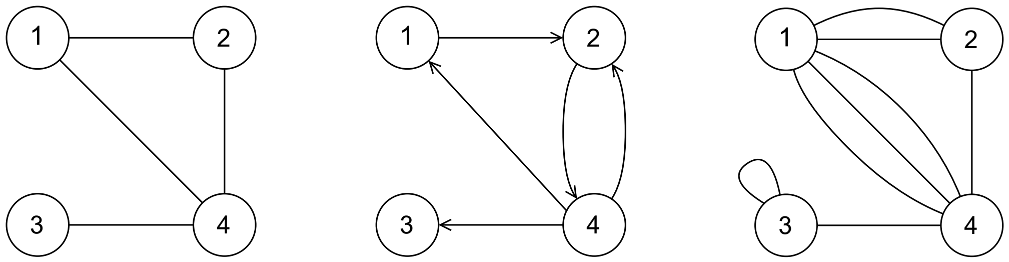
It is common to represent a graph as in Figure 1.1a, where circles represent vertices and lines between the circles represent edges. Most of the graphs on the course are *undirected*, but sometimes we need to use *directed edges*.

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

multigraph

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.



(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.

neighbor

degree

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 $\deg(v) = |N(v)|$.

indegree

maximum degree

d-regular graph

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*.

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

degree sum formula

Lemma 1.1 (Degree sum formula)**.** Let $G = (V, E)$ be a graph. Then

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

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. $\square$

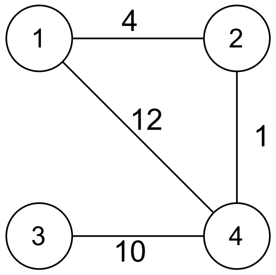
To model some problems, we sometimes need to represent a property – like capacity or length – of an edge by assigning a *weight* to it.

weighted graph

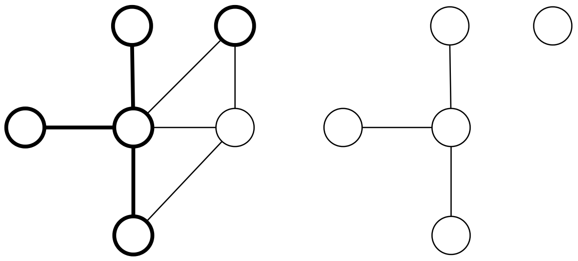
Definition 1.4 (Weighted graph)**.** A weighted graph is a graph $G = (V, E)$ together with the function $w : E \to \mathbb{R}$, where $w(e)$ is called the *weight* of the edge $e \in E$.

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*

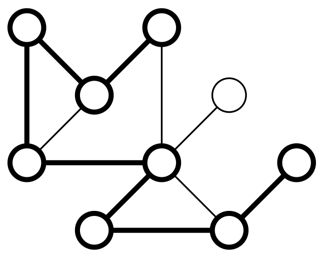
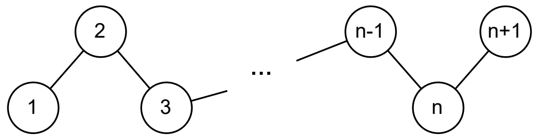



Figure 1.3: A subgraph that is a path on the left. A path of length n on the right.

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

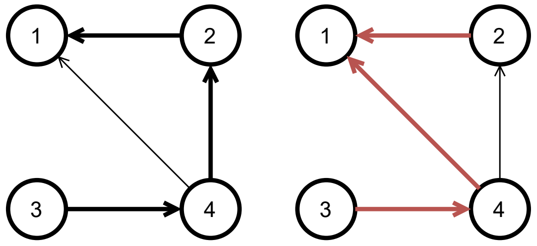


Figure 1.4: The bolded subgraph on the left is a directed path of length three but the one on the right highlighted in red is not.

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*

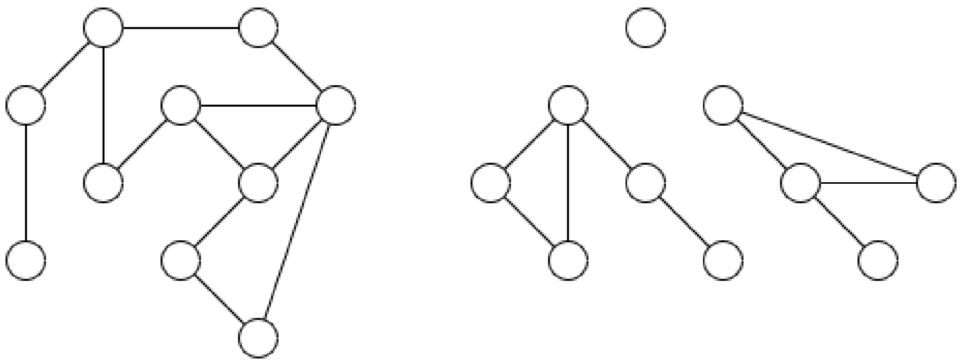


Figure 1.5: A connected graph on the left and a disconnected graph with three connected components on the right.

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*

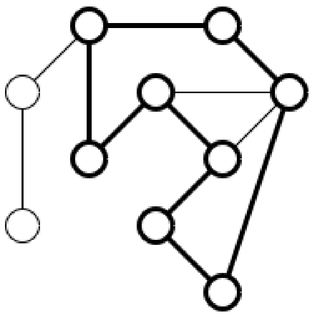


Figure 1.6: The bolded subgraph is a cycle of length eight.

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*

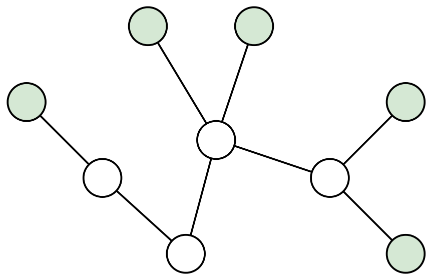


Figure 1.7: A tree with nine nodes. The leaves of the tree are colored in green.

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*

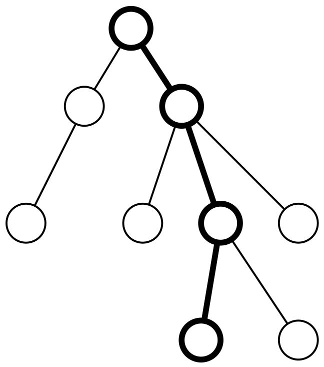
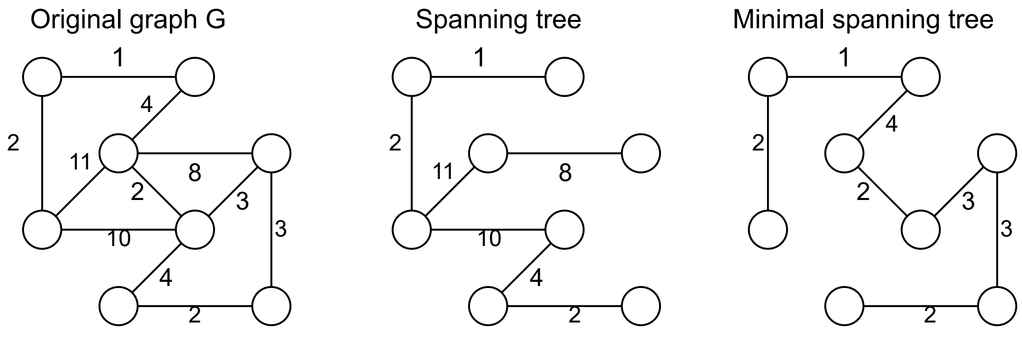


Figure 1.8: A rooted tree where the top node is the root. The depth of the tree is three. The path from the root to a leaf has been bolded.

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 $\phi^{-1}(i)$. $\phi^{-1}(i)$

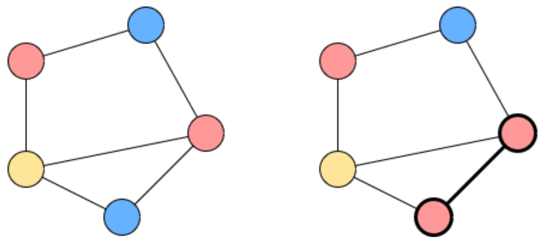


Figure 1.9: A proper vertex coloring on the left. The coloring on the right is not proper, because the bolded vertices are neighbors but both of them are red.

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.

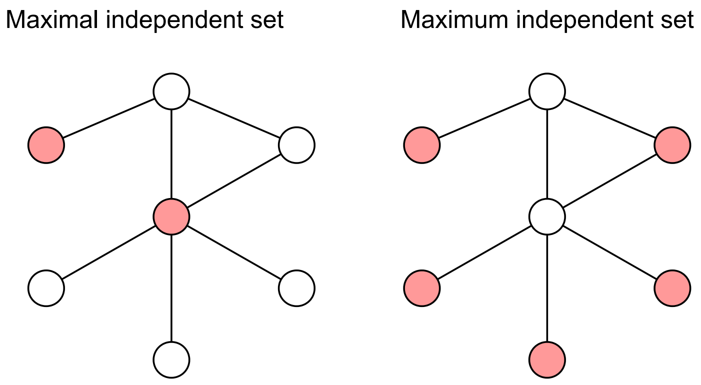


Figure 1.10: We cannot add any more vertices to the graph on the left, as all unchosen vertices already have a red-colored neighbor. There exists a larger independent set, the one on the right, which means that the independent set on the left is not maximum.

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

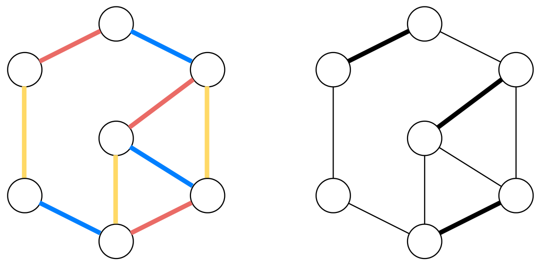


Figure 1.11: A proper edge coloring and a matching corresponding to red-colored edges. The matching happens to be maximal and maximum.

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

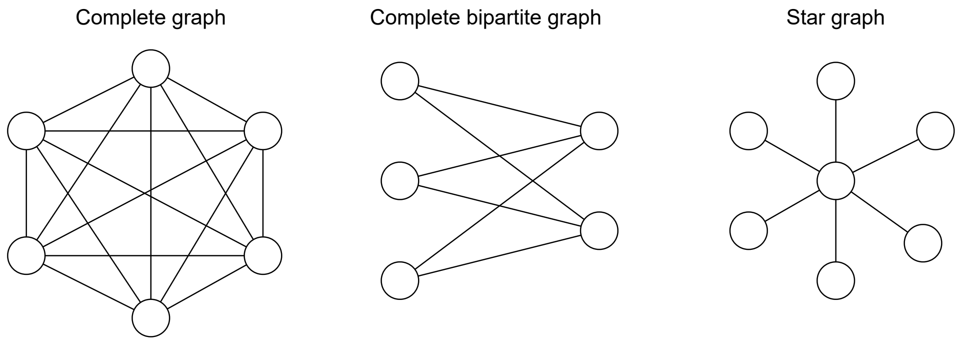
A graph $G = (V, E)$ is *bipartite*, if the set of vertices can be divided into two sets $V = A \cup B$ such that $A \cap B = \emptyset$, and all the edges in the graph are of form $uv \in E$, where $u \in A$ and $v \in B$. In other words, A and B are independent sets. Equivalently, bipartite graphs 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.

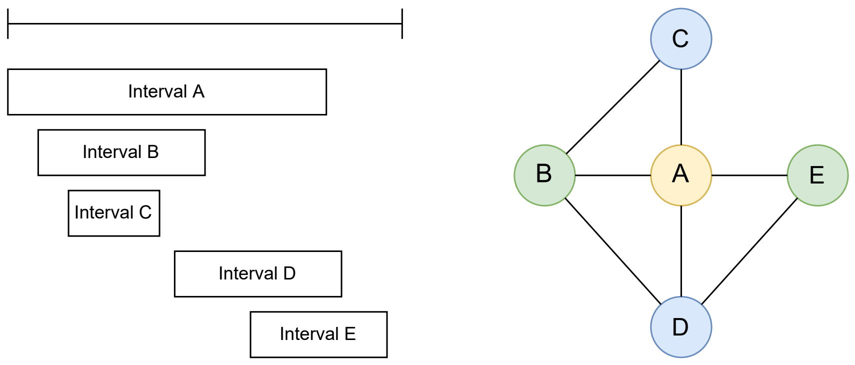


Figure 1.12: Intervals A, B, C, D, and E represented as an interval graph.

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

$$\begin{aligned} i &: \quad a > c > b, \\ j &: \quad c > a > b, \\ k &: \quad a > c > b. \end{aligned}$$

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.

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.

unstable edge

stable matching

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:

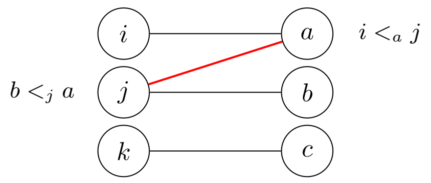
1. If there is an unstable edge $\{u, v\}$, match u and v with each other.
2. Match the previous partners of u and v with each other.

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.

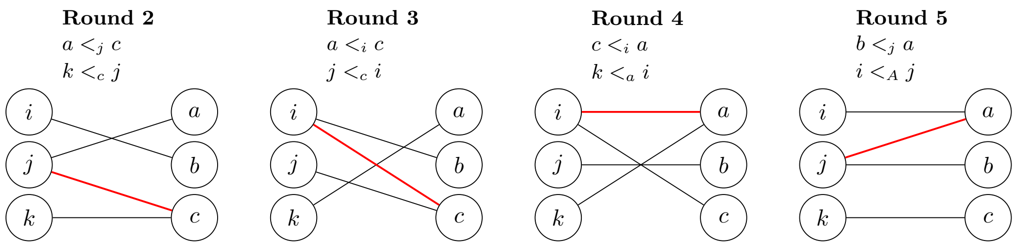
	i	j	k
1	a	c	a
2	c	a	c
3	b	b	b

	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.

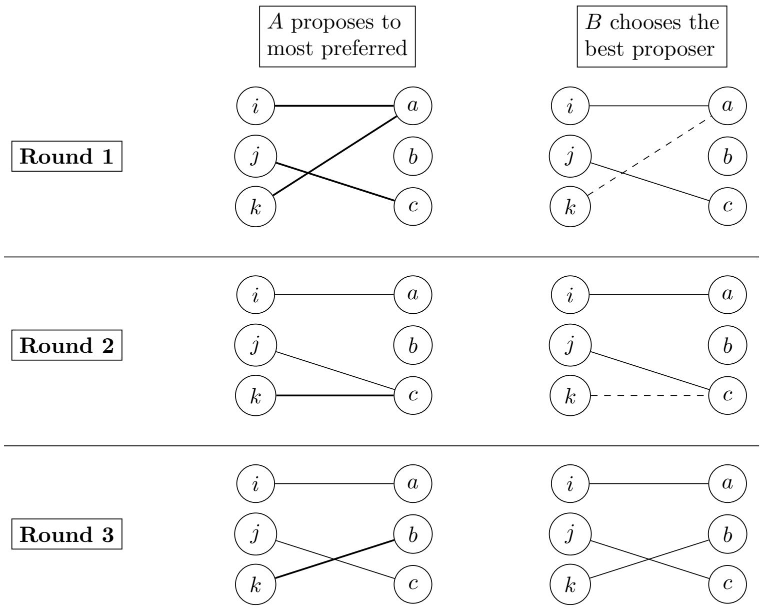
3. The statement "u has someone to propose to" in the while-loop is not really necessary, and the algorithm would be equivalent without it: it can be shown that any unmatched node must have a nonempty preference list. However, the statement makes the proof of termination a bit more straightforward.

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.

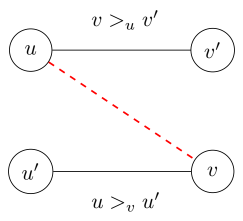
Lemma 2.2. The Gale-Shapley algorithm outputs a maximum matching.

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 $n \leq 1$ then
    return $n$
else
    return $n \cdot$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 $n - 1$.

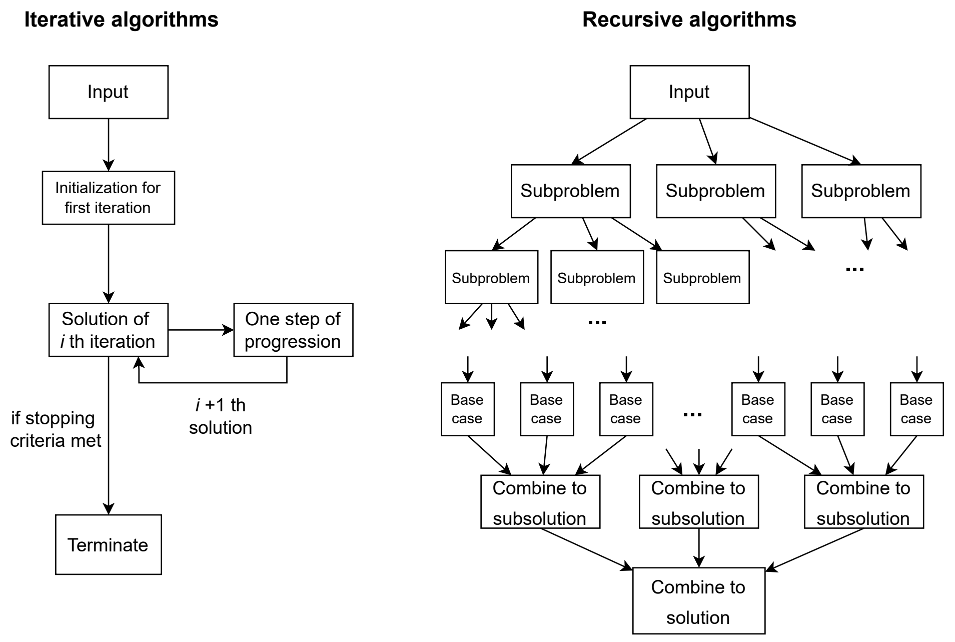
3. It must be relatively easy to combine the subproblems to construct the solution.

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 $n \leq 1$ then
    return $n$
else
    $a \leftarrow 0$; $b \leftarrow 1$
    while $i \leq n$ do
        $f \leftarrow b + a$
        $a \leftarrow b$
        $b \leftarrow f$
        $i \leftarrow i + 1$
    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 $n \leq 1$ 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.

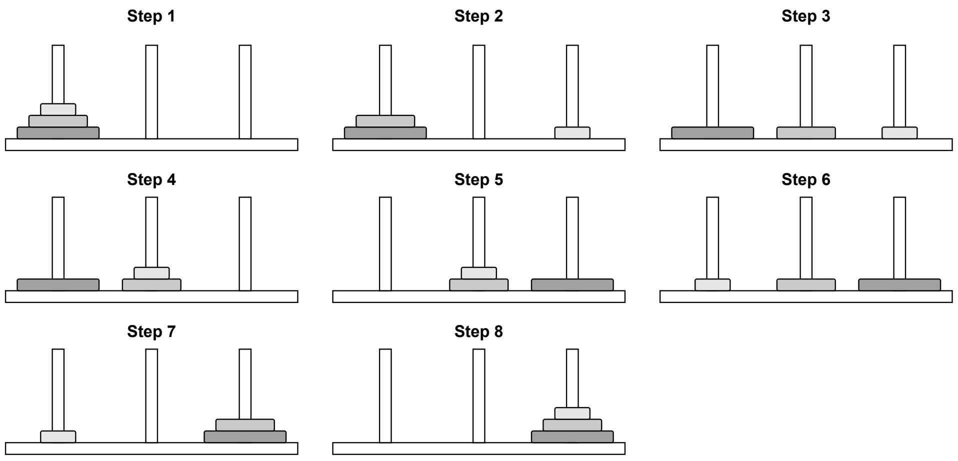


Figure 2.1: Example on how to solve the problem of Hanoi towers with three discs.

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

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

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

- From the beginning of the first subarray, $A[1, \dots, m]$ at index 1, or
- From the beginning of the second subarray, $A[m+1, \dots, n]$ 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 $B[1 \dots n]$
$A_1 \leftarrow A[1 \dots m]$
$A_2 \leftarrow A[m+1 \dots n]$
for $k \leftarrow 1 \dots n$ do
    if $A_1$ is empty then
        $B[k] \leftarrow A_2[1]$
        Remove the first element from $A_2$.
    else if $A_2$ is empty then
        $B[k] \leftarrow A_1[1]$
        Remove the first element from $A_1$.
    else if $A_1[1] \leq A_2[1]$ then
        $B[k] \leftarrow A_1[1]$
        Remove the first element from $A_1$.
    else
        $B[k] \leftarrow A_2[1]$
        Remove the first element from $A_2$.
/* Move the elements back to $A$                    */
for $k \leftarrow 1 \dots n$ do
    $A[k] \leftarrow B[k]$
```

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 $n \leq 1$ then
    return
else
    $m \leftarrow \lfloor \frac{n}{2} \rfloor$
    MergeSort($A[1 \dots m]$)
    MergeSort($A[m+1 \dots n]$)
    Merge($A[1 \dots n]$, $m$)
```

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.

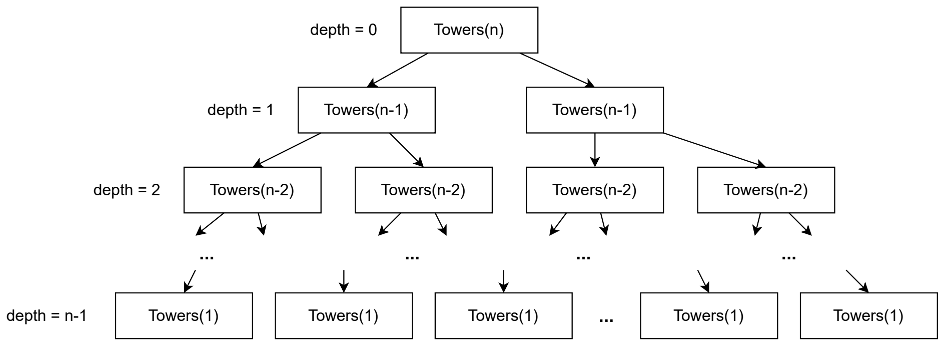
Recursion trees

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?

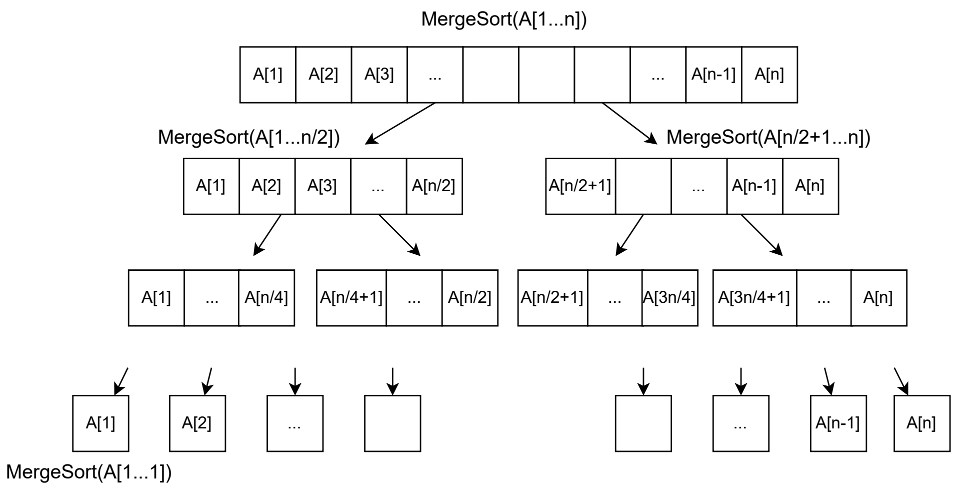
Usually, the former is relatively straightforward. For example, in the merge sort algorithm, combining subproblems, namely merging, takes $\Theta(n)$ time. To find how many recursive calls are hidden inside each subproblem, we use the concept of *recursion trees*. A recursion tree is a model, where the 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

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

$\square$

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
    $m \leftarrow \lceil \frac{n}{2} \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x$ mod $10^m$
    $c \leftarrow \lfloor y/10^m \rfloor$; $d \leftarrow y$ mod $10^m$
    $ac \leftarrow$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
    $m \leftarrow \lceil \frac{n}{2} \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x$ mod $10^m$
    $c \leftarrow \lfloor y/10^m \rfloor$; $d \leftarrow y$ mod $10^m$
    $ac \leftarrow$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.

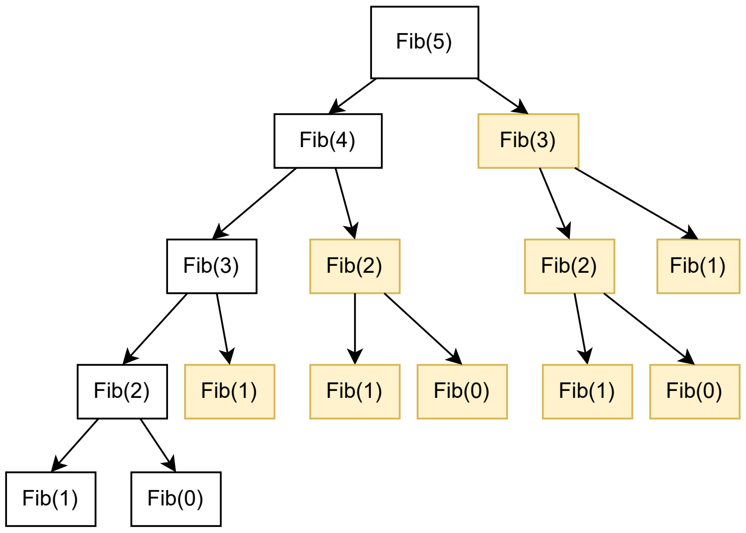
Algorithm 11: Fibonacci(n)

```
if $n \leq 1$ 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

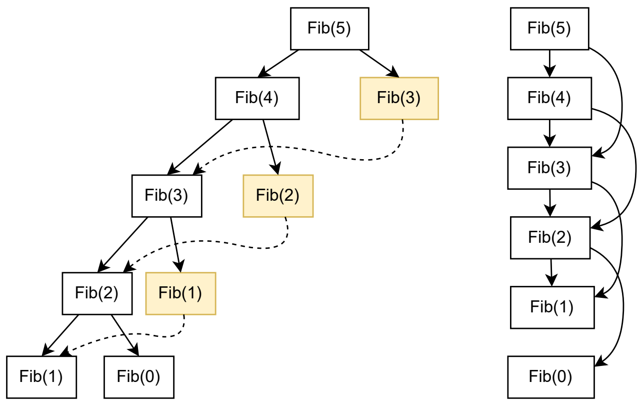
In addition to recursion trees, the dependencies of function calls can be represented as directed acyclic[1] graphs (DAGs). They are a great tool to visualize the process of removing repetitive function calls. While a recursion tree allows repetitions in the function calls, in a DAG each function call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

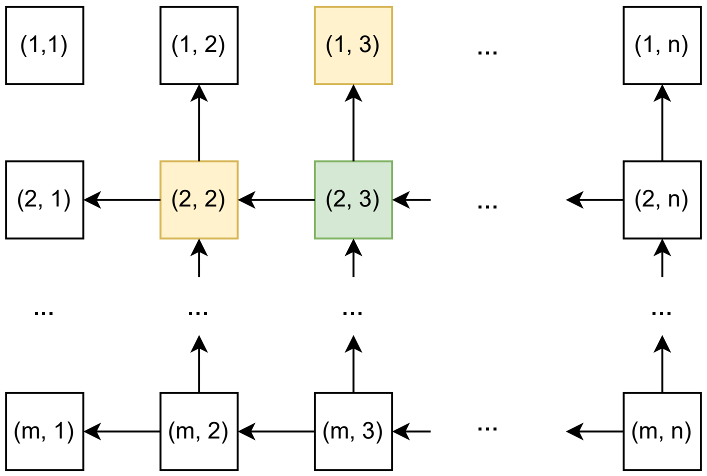


Figure 3.1: The DAG of a hypothetical algorithm. For instance, the node $(2, 3)$ depends on the nodes $(2, 2)$ and $(1, 3)$.

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

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

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.

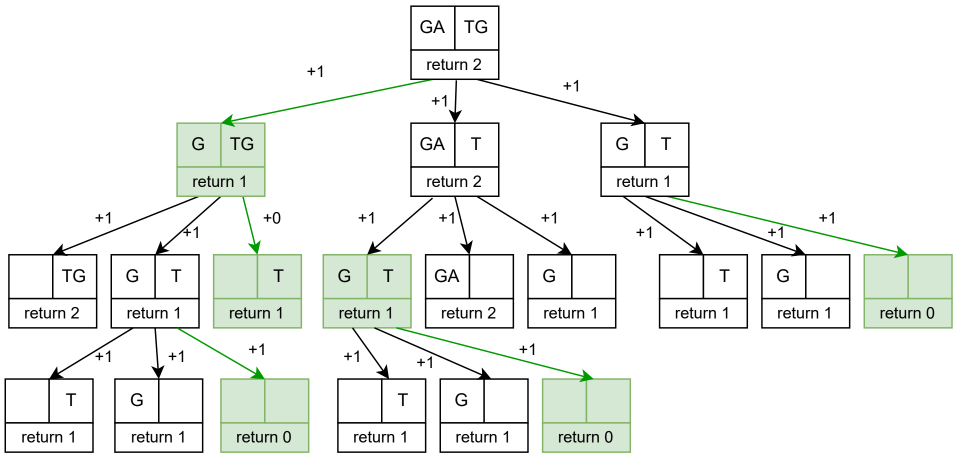
2. If b is empty, we construct b by removing all the letters of a. The cost is the length of a.

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)

```
$m \leftarrow$ length of $a$
$n \leftarrow$ 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
    $del \leftarrow$Edit$(a[1 \dots m-1], b) + 1$
    $ins \leftarrow$Edit$(a, b[1 \dots n-1]) + 1$
    if $a[m] = b[n]$ then
        $sub \leftarrow$Edit$(a[1 \dots m-1], b[1 \dots n-1])$   /* free substitution */
    else
        $sub \leftarrow$Edit$(a[1 \dots m-1], b[1 \dots n-1]) + 1$
    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 $\mathrm{Edit}(a[1\dots i], b[1\dots j])$ 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,

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

  The subsequence $a[1\dots i-1] \to b[1\dots j]$ 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,

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

  The subsequence $a[1\dots i] \to b[1\dots j-1]$ 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

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

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.

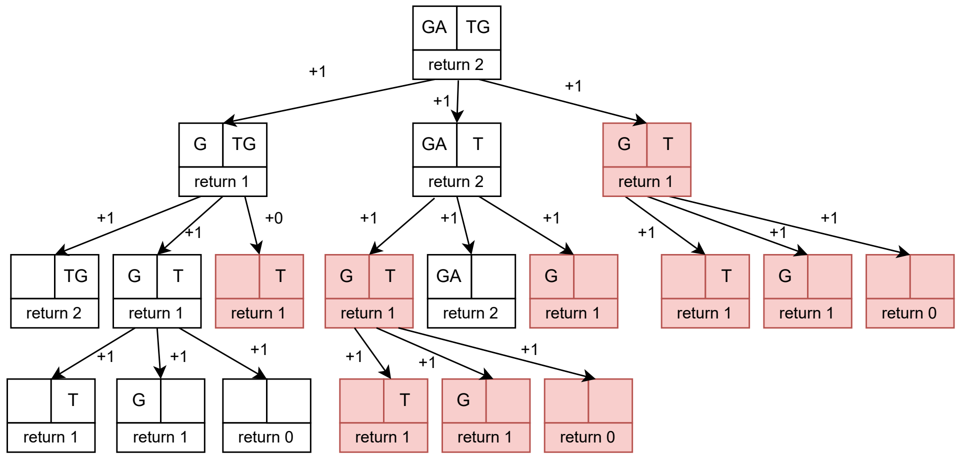
Hence the algorithm works for all b when $|a| = k$. By induction on the length of a, the algorithm works for all a and b.

□

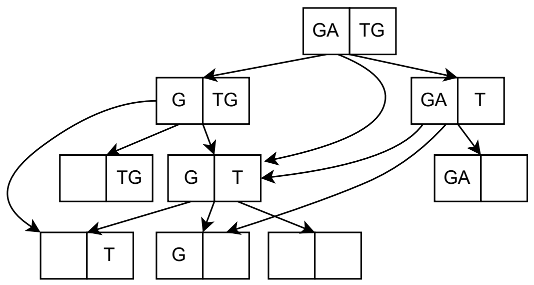
We will not prove separately that the algorithm terminates. It follows from the subproblems having strictly smaller sizes and can be proven formally by using a similar inductive argument as in Lemmas 2.1 and 2.2.

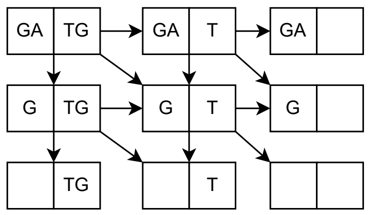
2.4 Dynamic programming improvement

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.

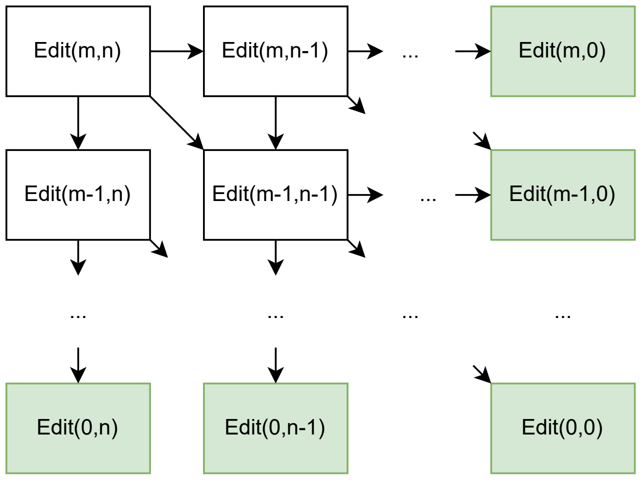


The DAG corresponding to this recursion tree is given below. The nodes can be re-ordered so that the grid-like structure of the DAG is more apparent.





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



To solve the problem, the required function calls are Edit(i, j) with $i = 0, \dots, m$ and $j = 0, \dots, n$, 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)

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

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

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

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 $g_i(x) \le 0$ for all $i = 1, \ldots, k$. An *optimum* solution x^* satisfies $f(x^*) \le f(x)$ 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

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

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 $\{1,\dots,n-1\}$ 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 $\{1,\dots,n-1\}$ and the capacity C.

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

4 Local search

In this chapter, we present a heuristic called *local search* for solving optimization problems. It has a lot of different applications, including gradient descent in deep learning, SAT solvers, and backtracking algorithms for solving the traveling salesman problem. Generally, the idea of local search is to apply small changes to a partial solution to move from one solution candidate to another. The process is then continued until an optimum is found.

We illustrate the methodology by solving the problem called Subset sum and finding minimum spanning trees for graphs.

1 Subset sum

We give two problem descriptions for the subset sum problem: subset sum as a decision problem and as an optimization problem.

Decision problem. Let $S \subset \mathbb{N}$ be a finite multi-set and $n \in \mathbb{N}$ be the *target value*. Is it possible to construct the target value n using the elements in S? In other words, does there exist a subset $I \subseteq S$ such that $\sum_{i \in I} i = n$?

Optimization problem. Let $S \subset \mathbb{N}$ be a finite multi-set[1] and $n \in \mathbb{N}$ be the target value. What is the maximum size of the subset $I \subseteq S$ such that $\sum_{i \in I} i = n$? Mathematically, the optimization problem takes the form

$$\begin{aligned} \text{maximize} \quad & |I| \\ \text{subject to} \quad & \sum_{i \in I} i = n \\ & I \subseteq S. \end{aligned} \tag{4.1}$$

We will focus mainly on the optimization problem. Notice, that any feasible solution for the optimization problem solves the decision problem, but the converse it false. Hence the optimization problem is strictly harder. In the worst case, we would need to go

[1]A multi-set is otherwise like a set, but it can contain an item multiple times. For example $\{1, 1, 3, 5, 5\}$ is a multi-set.

through all the solutions in the solution space of the decision problem to guarantee that the maximum has been found.

Example 1.1. Let $S = \{1, 1, 1, 2, 3, 5\}$ and $n = 5$. The solution space consists of all subsets that sum up to five, that is $\{1, 1, 1, 2\}$, $\{1, 1, 3\}$, and $\{5\}$. The largest one of them is $\{1, 1, 1, 2\}$ which is the optimum.

1.1 Algorithm design

To solve the problem, we will use a greedy local search heuristic. The greedy approach means that we construct the partial solutions by adding those integers that *appear* to give the subset of maximum size. To fit as many integers as possible, we want to start with the smallest integers we have. Obviously, simply choosing the smallest coins will not always end up solving the subset sum problem.

Example 1.2. With the multi-set $\{2, 2, 3, 3, 3, 4\}$ and $n = 9$, after choosing the coins $\{2, 2, 3, 3\}$ we have already exceeded the target value 9. In fact, there do not exist feasible solutions containing the coins $\{2, 2, 3\}$ nor $\{2, 2\}$. The only feasible solution that contains integer 2 is $\{2, 3, 4\}$.

The process of returning to an earlier partial solution is called *back-tracking*: we go back to the latest partial solution that might potentially work, and continue greedily skipping the value that previously caused us problems. For the subset sum algorithm, this means

1. Add integers to the subset I until the sum of the elements equals n (feasible solution found), or until we exceed n or run out of coins.
2. If the sum exceeds n, return to the most recent partial solution, ie. remove the last elements that were added. Then construct the next potential solution without using the removed elements.

Example 1.3. As we saw earlier, the partial solution $\{2, 2, 3\}$ does not lead to a feasible solution, so our next partial solution is $\{2, 2, 4\}$. This does not reach the target value 9, and we are already out of coins, so we must again back-track. The next iterations go as follows:

$$\{2,2,4\} \xrightarrow{\setminus\{2,4\}} \{2\} \to \{2,3\} \to \{2,3,3\} \to \{2,3,3,3\} \xrightarrow{\setminus\{3,3\}} \{2,3\} \to \{2,3,4\}.$$

Now we have found a solution for the decision problem: $\{2, 3, 4\}$. It happens to be an optimal solution. The other optimal solution in the solution space is $\{3, 3, 3\}$.

If there are no suitable subsets, we eventually end up iterating through all solutions. The algorithm would terminate when there are no solution candidates left.

Example 1.4. Let $S = \{1, 1, 4\}$ and $n = 3$. Then the back-tracking algorithm takes the following steps.

$$\{1\} \to \{1,1\} \to \{1,1,4\} \xrightarrow{\setminus\{1,4\}} \{1\} \to \{1,4\} \xrightarrow{\setminus\{1,4\}} \emptyset \to \{4\} \xrightarrow{\setminus\{4\}} \emptyset.$$

Unfortunately, the local search is not always able to find the optimal solution, as can be seen in the following example, where we end up finding a feasible solution that is not optimal.

Example 1.5. Let $S = \{1,3,3,3,8\}$ and $n = 9$. Then the back-tracking algorithm goes through the steps

$$\{1\} \to \cdots \to \{1,3,3,3\} \xrightarrow{\text{back-track}} \{1,3\} \to \{1,3,8\} \xrightarrow{\text{back-track}} \{1\} \to \{1,8\}.$$

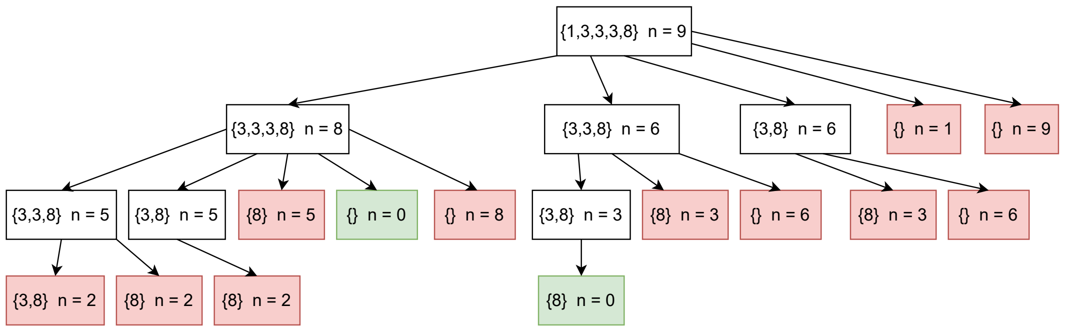
This solves the decision problem, but unfortunately the algorithm terminates before finding the optimal solution $\{3,3,3\}$.

The phenomenon shown in the example is called getting stuck in a *local optimum*. Unfortunately, the only possibility to avoid this issue is to iterate through all feasible solutions. The backtracking algorithm that goes through all of the feasible solutions is presented in Algorithm 15.

Algorithm 15: Subset sum

```
Input: Sorted multi-set of integers $S$, the target value $n$.
Output: The maximum multi-set $I$.
if $n = 0$ then
    /* Target reached                                                      */
    return $\emptyset$
else if $S$ is empty then
    /* Not a valid solution                                                */
    return null
else
    $I \leftarrow$ null /* Initialize the subset                              */
    Let $S = \{x_1, \ldots, x_m\}$
    for $x_i \in S$ do
        if $x_i \leq n$ then
            /* Try to add $x_i$ to the subset.                               */
            $S' \leftarrow \text{SubsetSum}(S \setminus \{x_1, \ldots, x_i\}, n - x_i) \cup \{x_i\}$
        else
            $S' \leftarrow$ null
        end
        if $S'$ is not null and ($I$ is null or $|I| < |S'|$) then
            /* New maximum found                                             */
            $I \leftarrow S'$
    end
    return $I$ /* If no valid solutions found, returns null                   */
```

Example 1.6. Let $S = \{1,3,3,3,8\}$, and $n = 9$. The recursion tree of the algorithm is given below. The red nodes represent function calls where there are no feasible subproblems left. The green nodes represent calls where the algorithm returns a feasible subset. The only feasible solutions are $\{1,8\}$ and $\{3,3,3\}$ and the latter is the optimum solution.



1.2 Correctness and runtime

To prove the correctness, we need to prove the following aspects.

1. The output is a feasible subset with a maximum size.
2. The algorithm terminates.

Lemma 1.1. If there exists a feasible solution, Algorithm 15 returns a feasible subset of maximum size. Otherwise, the algorithm returns "null".

Proof. The proof uses induction on the size of the multi-set S.

Base case: When $S = \emptyset$, we either have $n = 0$, when $S = \emptyset$ is the only feasible solution, hence being the optimal one, or $n > 0$ and there are no feasible solutions. In both cases, the algorithm works correctly.

Induction hypothesis: Suppose the algorithm returns an optimum subset whenever $|S| < k$.

Induction step: Suppose first that there is at least one feasible solution. For simplicity, assume the optimal solution OPT is unique. Let $x_i \in OPT$ be the first element in the optimal solution. By the induction hypothesis, the function call Subsetsum$(S \setminus \{x_1, \ldots, x_i\}, n - x_i)$ returns an optimal subset S' of the elements $S \setminus \{x_1, \ldots, x_i\}$ with target value $n - x_i$. There must exists a feasible solution for this problem, as it was assumed that x_i is the smallest element in OPT, and hence the rest of the elements $S \setminus \{x_1, \ldots, x_i\}$ can be used to construct the value $n - x_i$. Furthermore, we must have $|S'| + 1 = |OPT|$:

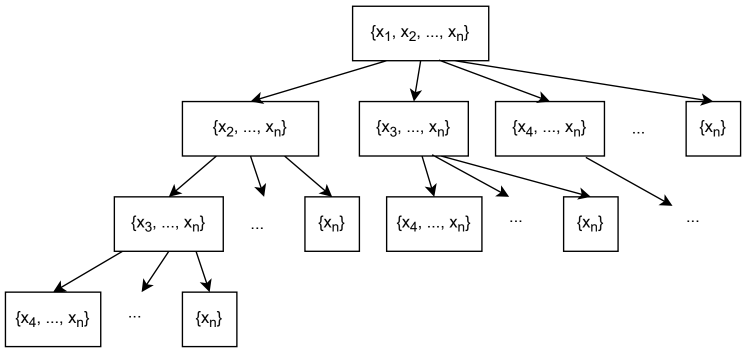
- If $|S'|+1 > |OPT|$, $S' \cup \{x_i\}$ would be a better feasible solution than OPT, resulting in contradiction.
- If $|S'|+1 < |OPT|$, $OPT \setminus \{x_i\}$ would be a better feasible solution to the subproblem $(S \setminus \{x_1, \ldots, x_i\}, n - x_i)$, contradicting the optimality of S'.

In the for-loop, the algorithm iterates over the subproblems $(S \setminus \{x_1, \ldots, x_i\}, n - x_i)$, and chooses the set $S' \cup \{x_i\}$ with maximum size. As we showed that $|S' \cup \{x_i\}| = |OPT|$, the set $S' \cup \{x_i\}$ is the subset with maximum size, and the algorithm correctly returns it.

If there are no feasible solutions, all the recursive calls inside the for-loop must return "null", and the algorithm returns "null". □

Lemma 1.2. Algorithm 15 runs in $O(2^n)$ time.

Proof. Let's take a look at the recursion tree of the algorithm. In the worst case, no branch terminates early due to constraint $x_i > n$ or $n = 0$ being satisfied, and the tree would take the following form.



Each node in the tree takes $O(1)$ time to run since apart from the recursive calls, we only have a couple of comparisons and assignment operations. Each subproblem of size m makes $m - 1$ recursive calls in the worst case. Notice that the subtree starting from node $\{x_2, \ldots, x_n\}$ has the same size as the whole recursion tree where the branch of node $\{x_2, \ldots, x_n\}$ has been removed. This gives the recurrence relation

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

and we have $T(n) = O(2^n)$. □

As expected, the algorithm we designed is not very efficient. This is not entirely devastating, as it has been shown that the subset sum problem is *NP-hard*, which practically means that it cannot be solved in polynomial time with respect to the size of the multi-set. In chapter 3.8 of *Algorithms* by Jeff Erickson, a relatively efficient dynamic algorithm is introduced for the problem. The algorithm runs in $O(mn)$ time, where n is the target value and $|S| = m$, but n can be exponential in m.

2 Minimum spanning trees

Minimum spanning trees are a special family of subgraphs, having applications for example in image segmentation, handwriting recognition, and network optimization. Finding minimum spanning trees is one of the rare problems where the greedy approach can be efficiently used to construct the optimal solution.

Throughout the section, we assume that the input graphs are connected. If the graph is not connected, the algorithm can be applied to each connected component individually.

2.1 Problem description

Let us begin with some relevant graph theory definitions.

Definition 2.1 (Total weight of a graph)**.** For a weighted graph $G = (V, E)$ with weights $w : E \to \mathbb{N}$, the *total weight* of the graph is

total weight

$$w(G) = \sum_{e \in E} w(e).$$

Definition 2.2 (Minimum spanning subgraph)**.** Let $G = (V, E)$ be a graph with weights $w : E \to \mathbb{N}$. Its *minimum spanning subgraph* G' satisfies

minimum spanning subgraph

1. (Spanning subgraph.) The subgraph $G' \subseteq G$ is connected and it *spans* the vertices of G, ie. $V(G) = V(G')$.
2. (Minimality.) The graph G' is a spanning subgraph with the smallest total weight. In other words, if G'' is any other spanning subgraph of G, then $w(G') \leq w(G'')$.

A connected graph always has a minimum spanning subgraph, because there always exists at least one feasible solution: the graph G itself. It turns out that a minimal spanning subgraph is always a tree. For this reason, minimum spanning subgraphs are generally called *minimum spanning trees*, often simply MSTs.

Lemma 2.1. A minimum spanning subgraph is always a tree.

Proof. Suppose for contradiction that the minimum spanning subgraph G' of G is not a tree. This means that there exists a cycle $C \subseteq G'$. Removing any edge $e \in C$ from G' would give a connected subgraph, which means that $G' \setminus e$ is a spanning subgraph of G with a smaller total weight. This is a contradiction, as G' was assumed to be minimum. Hence G' cannot contain cycles and it is a tree. □

The minimum spanning tree of a graph is not necessarily unique, as can be seen from Figure 4.1. However, uniqueness is guaranteed if all the edges have different weights.

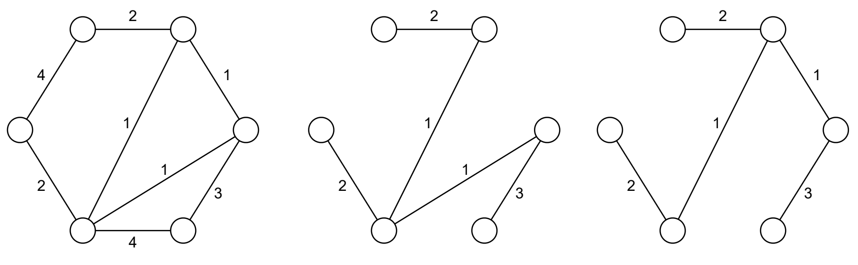
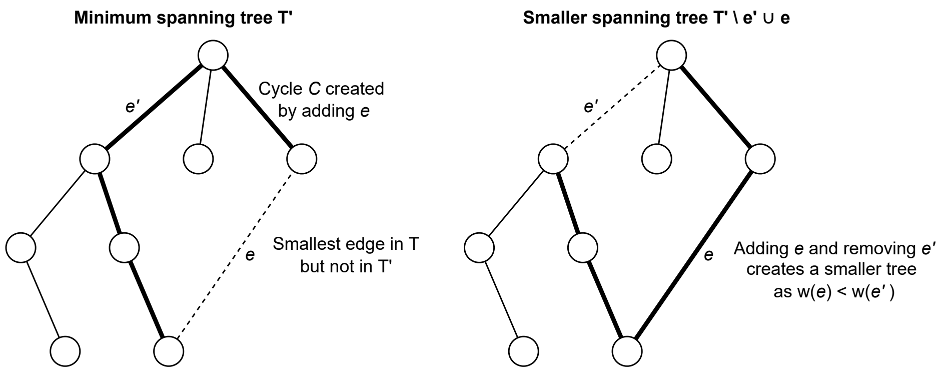


Figure 4.1: Weighted graph G on the left has two different MSTs.

Lemma 2.2. Let $G = (V, E)$ be a weighted connected graph with weights $w : E \to \mathbb{N}$ where $w(e) \neq w(e')$ for all edges $e \neq e'$. Then G has a unique MST.

Proof. Suppose G has two MSTs, T and T'. Let e be the edge with the smallest weight that is not in both of the trees. Suppose without loss of generality that $e \in T$. Because T' is a tree, adding e to T' would create a cycle $C \subseteq T' \cup e$. Removing any edge $e' \in C$ from $T' \cup e$ would give a valid spanning tree. Let $e' \in C \setminus T'$ be the edge with the smallest weight in $C \setminus T'$. Because e was the smallest edge not shared by the trees, $w(e') > w(e)$, and replacing e' with e would make T' smaller. This is a contradiction, as T' was assumed to be a minimum spanning tree.


Minimum spanning tree T'
e'
Cycle C created by adding e
e
Smallest edge in T but not in T'
Smaller spanning tree T' \ e' ∪ e
e'
e
Adding e and removing e' creates a smaller tree as w(e) < w(e')

□

2.2 Kruskal's algorithm

Kruskal's algorithm is a well-known algorithm for finding the MST using local search. The basic idea is to use the concept of *marginal gain*. Marginal gain means the benefit gained from a small change in the current state of the solution. In the MST problem, a small change means adding a specific edge to the MST. Practically, we start building the tree from an empty edge set and add edges one by one until we have a spanning tree. In each step, the goal is to choose the option that maximizes the marginal gain. Because in the MST problem we are interested in minimizing the cost of the tree, adding the edge with the smallest weight results in the greatest gain.

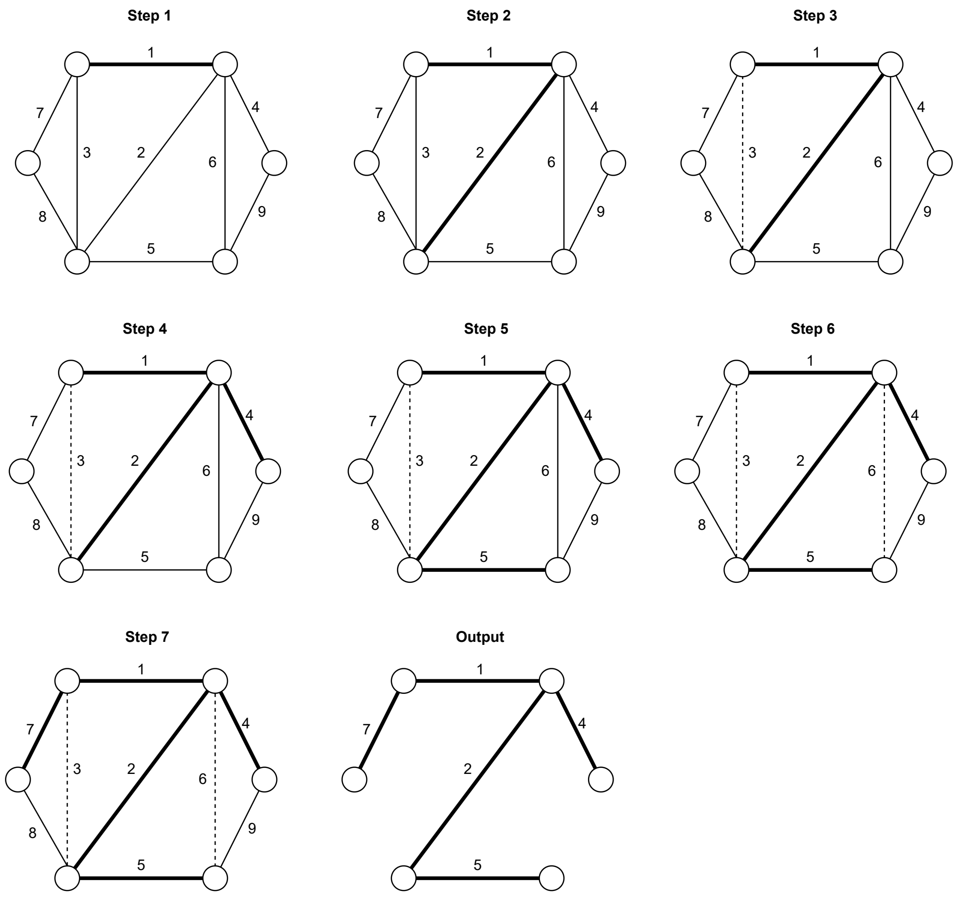
$$\max_{e \in E} \; w(T) - w(T \cup e) = \max_{e \in E} \; -w(e) = \min_{e \in E} w(e).$$

Additionally, to ensure the correctness of the algorithm we need to make sure that T is a tree. This will be done by disregarding those edges that would create a cycle. The algorithm may terminate when T has enough edges to span the graph G. Kruskal's algorithm is presented in Algorithm 16.

Algorithm 16: Kruskal's algorithm

```
Input: Set of weighted edges $(E, w)$.
Output: The minimum spanning tree $T$.
$T \leftarrow (V, \emptyset)$. /* Initialize with empty edge set. */
while $T$ is not connected do
    Find the edge $e \in E$ with minimal weight.
    if $T \cup e$ does not contain cycles then
        $T \leftarrow T \cup e$
    Remove $e$ from $E$.
end
return $T$
```

Example 2.1. Below we have the execution of the algorithm for an example graph. The edges added to the tree T have been bolded, and the discarded edges are dashed. The algorithm takes seven iterations to finish and returns a spanning tree of weight 19.



Correctness

To simplify the proof, we will assume that the weights of the edges are distinct. Then by Lemma 2.2 the MST of the graph is unique. With small adjustments, the proof for non-distinct weights can be done by choosing the MST to be the one with most edges common with the output T.

In the proof, we first need to ensure that the output is a spanning tree: it is connected and does not contain cycles. After that, we will prove the minimality using contradiction.

Lemma 2.3. Kruskal's algorithm returns an MST.

Proof. Let T be the output of Algorithm 16. The graph T is a tree because no edges are added that would create cycles. It must be a spanning tree because the algorithm does not terminate until T is connected. The algorithm terminates eventually because we consider each edge at most once.

Let T^* be the unique MST of the graph G, and suppose for contradiction that $T \neq T^*$. Let $e^* \in T^*$ be the edge with the smallest weight not in T. We have two cases:

1. The algorithm terminated before e^* was handled. This means that all edges in T are smaller than e^*, and hence $w(T) < w(T^*)$, which is a contradiction as T^* is the MST.
2. The algorithm discards e^* because it would create a cycle C. This means that all edges in C have been handled before e^*, and hence $w(e) < w(e^*)$ for all $e \in C$. There must be at least one edge $e' \in C$ that is not in T^* – otherwise, T^* would contain a cycle. Replacing the edge e^* with e' would make T^* smaller, which is a contradiction as T^* is the MST.

Hence, the assumption $T \neq T^*$ cannot be true, and the algorithm outputs the minimal spanning tree. □

Runtime

Let's first consider a naïve implementation of the algorithm. Let $|E| = m$ and $|V| = n$.

1. It is possible that in one iteration, we add no edges. In the worst case, we might end up needing to go through every edge before T is a tree, which means that the while loop is executed $\Omega(m)$ times. Furthermore, checking the while condition takes $\Omega(n)$ time.
2. If the edges are not ordered, finding the minimum edge will take $\Omega(m)$ time.
3. In a naive implementation, checking that $T \cup e$ does not contain cycles will take $\Omega(n)$ time.

All in all, this gives $\Omega(m \cdot \max\{m, n\})$ runtime, which is not yet very good. With careful implementation, we can reduce the runtime to $\Omega(m \log m)$.

Step 1. Unfortunately, it is not possible to reduce the number of iterations as we might need to go through all edges before T is connected. In the worst case, the number of iterations in the while-loop will inevitably be $\Omega(m)$. Furthermore, the connectedness cannot be checked efficiently, so we settle for iterating through all edges: this does not increase the worst-case runtime, as the while loop already runs $\Omega(m)$ times. The other costs will be reduced using smart preprocessing.

Step 2. To reduce the time spent on scanning the edges, we will preprocess the graph so that the edges are stored in increasing order with respect to weight. After that, we can access the minimal edge in $O(1)$ time inside the while loop. In week 2, we learned that this sorting problem can be solved in $O(m \log m)$.

Step 3. For checking the existence of cycles in $T \cup e$, we will use a data structure called *union-find*. The data structure is used to represent graphs so that the nodes in the same connected component have the same identifier. The data structure has two functions:

1. FIND(u) returns the identifier of the node u in time $O(\log n)$.
2. UNION(u,v) merges the nodes u and v in time $O(\log n)$. This corresponds to adding an edge between u and v. After the merging, the nodes u and v have the same identifier.

The details of the implementation of union-find are covered in section 3 for those who are interested.

Now we can represent the tree as a union-find structure. Initially, the tree has no edges, which corresponds to all nodes having their unique identifier. The edge $e = uv$ creates a cycle if and only if u and v are already in the same connected component. This can be checked in $O(\log n)$ time by running FIND(u) and FIND(v). If the edge does not create a cycle, we merge the connected components of u and v in $O(\log n)$ time by calling UNION(FIND(u), FIND(v)). The final algorithm is presented in Algorithm 17.

The runtime of the new algorithm is given by the following steps.

1. Sorting the edges takes $O(m \log m)$ time.
2. Union-find can be initialized in $O(n)$ time, and hence its cost is negligible compared to $O(m \log m)$.
3. While loop is run m times. Inside the while loop, we call FIND two times and UNION once, taking $O(\log n)$ time. The whole while loop runs in $O(m \log n)$.

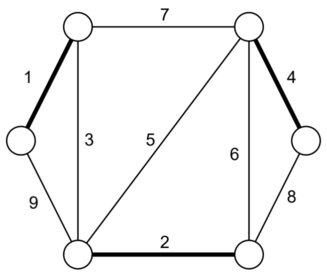
Because the input graph is connected, it must contain at least $n - 1$ edges. Hence $n = O(m)$, and the runtime is dominated by the sorting, giving the algorithm the runtime $O(m \log m)$.

Algorithm 17: Efficient Kruskal's algorithm

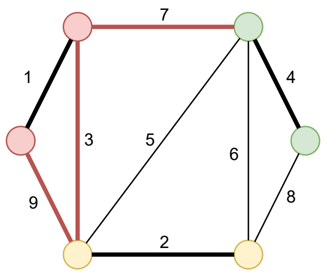
```
Input: Set of weighted edges $(E, w)$.
Output: The minimum spanning tree $T$.
Sort the edges in $E$ by weight.
Initialize union-find for vertices in $V$.
$T \leftarrow (V, \emptyset)$.
for $e \in E$ do
    Let $e = uv$.
    $id_u \leftarrow \text{FIND}(u)$
    $id_v \leftarrow \text{FIND}(v)$
    if $id_u \neq id_v$ then
        $T \leftarrow T \cup e$
        UNION($id_u$, $id_v$)
end
return $T$
```

2.3 Borůvka's algorithm

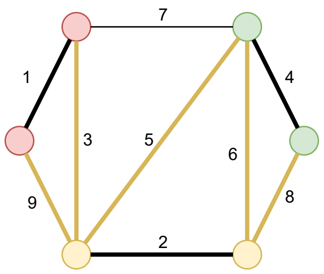
Borůvka's approach to the MST problem is very similar in nature to Kruskal's: the idea is again to add the smallest edges but now we are allowed to add multiple edges in one iteration. Practically, for each vertex $v \in V$, we consider the edges incident to it and add the one with minimal weight. Because this is not necessarily enough to connect the graph, we continue the process in the next iteration by adding the *minimum-weight edge of each connected component*. The execution of the algorithm for an example graph is given below. One can verify that this is the same MST that Kruskal's algorithm outputs.



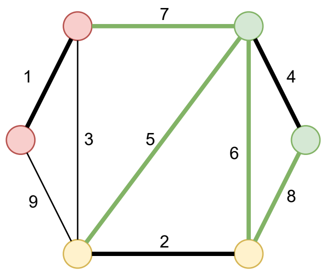
After adding the smallest edges incident to each node, we have three connected components



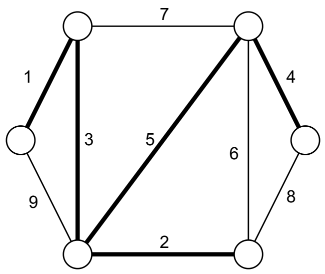
The red component has three edges, the one with weight 3 being minimum.



Minimum edge of the yellow component is the edge with weight 3.



Minimum edge of the green component is the edge with weight 5.



Graph is connected and the algorithm terminates.

Algorithm 18: Borůvka's algorithm

```
Input: Connected graph $G = (V, E)$.
Output: The minimum spanning tree $T = (V, F)$.
$F \leftarrow \emptyset$
$T \leftarrow (V, F)$
while $T$ is not connected do
    for each connected component do
        $e \leftarrow$ minimum edge incident to the component
        $F \leftarrow F \cup \{e\}$
    end
end
return $T$
```

The proof of correctness and the runtime analysis for the algorithm will be given in the tutorial session. As per usual, the proof consists of the following steps:

1. The algorithm terminates.
2. The output is a tree (connected and no cycles).
3. The output is an MST.

Lemma 2.4. The runtime of Algorithm 18 is $O(m \log n)$.

While the runtime of the algorithm is not significantly different from the runtime of

Kruskal's algorithm, it performs well with graphs with a small enough average degree. One family of such graphs is the family of *planar graphs*. The following is an informal definition of planar graphs. For a formal definition, refer to Graph theory by Diestel.

Definition 2.3 (Planar graphs)**.** The graph $G = (V, E)$ is *planar* if it can be drawn without edges crossing over each other.

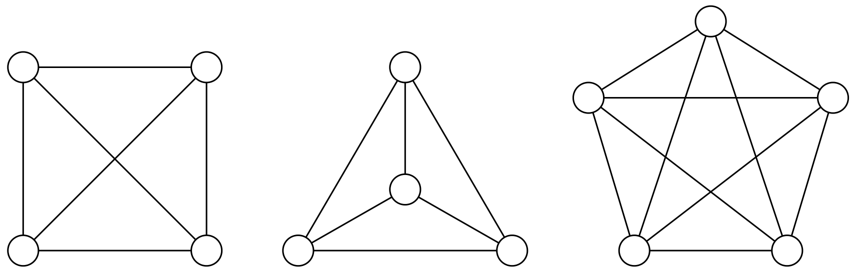


Figure 4.2: The two left-most graphs are both different drawings of K_4. Even though the left-most one has crossing edges, the graph K_4 is planar because it has a representation without crossing edges (middle). It can be shown that K_5 is not planar (right).

Planar graphs have the nice property that the number of edges is at most three times the number of vertices. Those interested in graph theory can refer to for example Diestel's *Graph Theory* for a proof.

Lemma 2.5. Let $G = (V, E)$ be a planar graph. Then $|E| \leq 3|V| - 6$. *planar graph*

Lemma 2.6. For planar graphs, the runtime of Algorithm 18 is $O(n)$.

3 Union-find

To be able to quickly combine and compare different components, we implement union-find using rooted trees, where the ID of the component is determined by the ID of the root. By balancing the tree appropriately, we ensure that the distance to the root is $O(\log n)$. In the data structure, each node has a parent and a depth: the parent allows us to iterate to the root of the tree corresponding to the component, and the depth allows us ensure that all operations work in $O(\log n)$ time.

Initialization. Initially all the nodes are in their own connected components, and hence they are all roots. The parent of the root $v \in V$ is the node v itself. The rank of all nodes in the beginning is zero. This can be implemented by calling the following function for all nodes.

Find. To find the identifier of the node, we iterate through the parents, until we reach the root. This can be done using the following while-loop. The runtime is upper-bounded by the height of the tree corresponding to the component x belongs to.

Algorithm 19: Find(x)

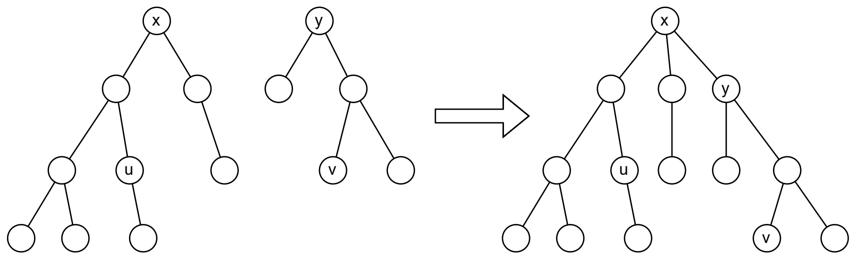
```
while $x.parent \neq x$ do
    $x \leftarrow x$.parent
return $x$
```

Union. To merge two components of u and v, we start by finding the roots of u and v. The merging is essentially implemented by making the root of one tree into parent of the other root. To avoid creating trees with depth $O(n)$, we will choose the new root to be the tree with larger depth. The new depth of the tree then either stays the same or increments by one in the case where the two trees have the same depth. Merging is implemented as follows.

Algorithm 20: Union(u, v)

```
$x \leftarrow$ FIND($u$)
$y \leftarrow$ FIND($v$)
if $x.rank > y.rank$ then
    $y$.parent $\leftarrow x$
else if $x.rank < y.rank$ then
    $x$.parent $\leftarrow y$
else
    $x$.parent $\leftarrow y$
    $x$.depth $\leftarrow x$.depth +1
```

Below is an example of a situation where the tree rooted with x has larger depth than the tree of y, and after merging the depth of the tree stays the same.



5 Greedy algorithms

1 Greedy algorithms

Greedy algorithms are algorithms that construct the solution by extending it with small "seemingly optimal" steps. The name "greedy" comes from the heuristic of choosing the best available next step. For instance with minimum spanning trees, we add the smallest edge to the tree, and with greedy graph coloring we use the smallest available color.

Even though the greedy algorithm for MSTs was able to output an optimum tree, in general, the greedy heuristic is not able to solve a problem exactly. They can still sometimes be used to construct "good enough" solutions: for instance, the greedy graph coloring outputs $\Delta + 1$ coloring, which is not always optimal, but in some cases could be enough. For some problems – like graph coloring – it can be proven that there are no efficient algorithms, and solving the problems approximately is the only way to reach a reasonable runtime. These types of algorithms that trade precision of the solution with runtime are called *approximation algorithms*.

This week we revisit the Knapsack algorithm that we solved dynamically in week 3. Instead of finding the optimal solution, we will find an approximation for the optimal value using a fast greedy algorithm. After that, we introduce the set cover problem, which has no efficient algorithm that finds the optimal solution. We design a greedy algorithm that runs in polynomial time and show that the output gives an approximation that is at most $\Theta(\log n)$ times worse than the optimal.

1.1 Approximation algorithms

Approximation algorithms allow us to trade the precision of the solution for runtime, memory consumption, or complexity of the algorithm. To be able to rely on the approximation algorithm, we must have tools to quantify the accuracy of the approximation. The output of the algorithm should always be within certain bounds from the optimum. The following definition gives us a framework for comparing approximations.

Definition 1.1 (α-approximation)**.** Let OPT be the optimal value of a minimization problem. The feasible solution X is an α-approximation for the optimization problem if

$$OPT \leq \text{cost}(X) \leq \alpha \cdot OPT.$$

In other words, the solution X is at most α times worse than an optimal solution. These types of approximations are called *multiplicative approximations*. For maximization problems the definition is similar, but the α-approximation satisfies a lower bound

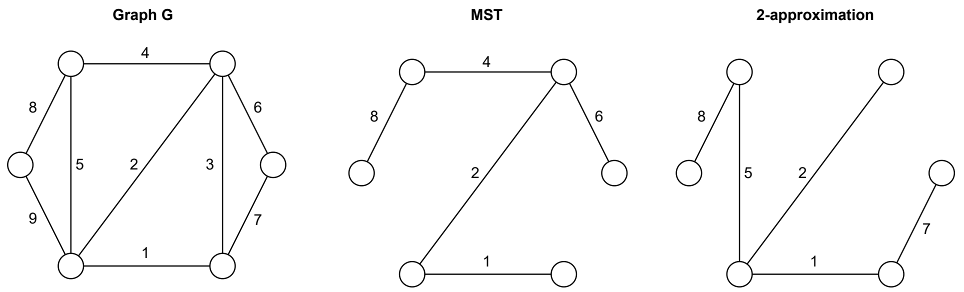
$$(1/\alpha) \cdot OPT \leq \text{cost}(X) \leq OPT,$$

or equivalently

$$\text{cost}(X) \leq OPT \leq \alpha \cdot \text{cost}(X).$$

Remark. The approximate solution X must still be feasible. For instance, in the graph coloring problem, an approximation can use more colors than an optimal solution, but the coloring must still be a proper coloring.

Example 1.1. Consider the graph G below with the MST given in the middle. The tree T on the right is a 2-approximation, as it is a feasible spanning tree for G and $w(T) = 23 \leq 2 \cdot w(\text{MST}) = 2 \cdot 21 = 42$.



Definition 1.2 (α-appriximation algorithm)**.** An α-approximation algorithm is an algorithm whose output is an α-approximation for the given problem.

Example 1.2. A 2-approximation algorithm for the MST problem always outputs a spanning tree that is at most twice as heavy as the minimum spanning tree.

2 Knapsack problem revisited

In Graded Exercise 3, we designed a dynamic algorithm to solve the knapsack problem in $O(mn)$ time. This week, our goal is to design a fast greedy 2-approximation algorithm. This means that the algorithm must output a feasible solution that has a value of at least half of the optimum. The problem description is slightly altered to simplify the analysis: this time we can pick any item as many times as we wish.

2.1 Problem description

Let's recall the knapsack problem from Week 3. We have a set of items I, each item $i \in I$ having a value v_i and a weight w_i. We want to fill in the knapsack with a weight capacity C so that the value of the items inside the knapsack is maximized. The items we pack must fit into the knapsack, and hence all feasible solutions S must satisfy the constraint

$$\sum_{i \in S} w_i \leq C.$$

Additionally, we are allowed to pack a single item $i \in I$ multiple times, ie. S can be a multi-set. We represent the multi-set S by defining a *quantity* x_i for each item $i \in I$. For example, if the item i appears four times in the solution, we set $x_i = 4$. The optimization problem takes the form

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

Example 2.1. Let the capacity of the knapsack be 10. We have three items to choose from: apples, gold bars, and peanuts. The values and the weights are given in the table below.

Item	value	weight
apples	5	1
gold bar	100	8.5
peanuts	0.5	0.3

The optimal solution is to pack one of each of the items, giving a total value of 105.5. Some other feasible solutions are

- Packing ten apples giving value 50. This is a 3-approximation.
- Packing 5 apples and 16 peanuts with the value of 33. This is a 4-approximation.
- Packing one gold bar and 5 peanuts with the value of 102.5. This is a 2-approximation.

2.2 Algorithm design

We base our greedy algorithm on the concept of marginal gain, that is constructing the solution piece by piece using the *seemingly best* option at each step. In the knapsack problem, it is not enough to choose the most valuable item: it is easy to find counter-examples where the optimal solution does not contain the most valuable item. Instead, we are interested in maximizing the *relative cost* of the item, ie. the ratio between the value and the weight of the item.

From this observation, the algorithm is relatively simple: first, we discard all items $i \in I$ satisfying $w_i > C$, ie. items that are too heavy to fit in the knapsack. After that, we fill in the knapsack with the item that has the best relative cost.

It is easy to see that the solution returned by this algorithm might not be optimal and that the algorithm could be easily improved by continuing the process: packing the second most valuable option that fits. For our analysis, this is not needed, as it would increase the runtime and we only aim to find a 2-approximation. The algorithm is described in Algorithm 21.

Algorithm 21: Greedy Knapsack 2-approximation

Input: Set of items I with weight and value and capacity C.
Output: The item max that will be packed and its quantity.

```
$max \leftarrow 0$/* Store the item with maximal relative value */
for $i \in I$ do
    if $w_i \leq C$ then
        If item $i$ has higher relative cost than $max$, set $max \leftarrow i$.
end
$n \leftarrow \lfloor \frac{C}{w_{max}} \rfloor$ /* How many items fit into the knapsack */
return $(max, n)$ /* Returns the optimal item and its quantity */
```

2.3 Correctness

For the correctness of the algorithm, we need to prove that

1. The algorithm terminates.
2. The output is feasible.
3. The output has a value of at least half of the optimum value.

The first two points are a direct result of the structure of the algorithm. The algorithm always terminates after iterating through the items and computing the two arithmetic operations in the end. Feasibility is guaranteed too, as we only pack $n = \lfloor \frac{C}{w_{max}} \rfloor$ of the best item and

$$\sum_{i \in I} w_i x_i = n \cdot w_{max} \leq \frac{C}{w_{max}} \cdot w_{max} = C.$$

The only non-trivial part is to prove that the output is a 2-approximation. In the proof, we denote the total value of the subset S by $V(S)$.

Lemma 2.1. Algorithm 21 returns a 2-approximation for the Knapsack problem.

Proof. Let the multiset S^* be the optimal solution, and let S be the multiset corresponding to the output of Algorithm 21. Notice that

$$V(S^*) = \sum_{i \in S^*} v_i = \sum_{i \in S^*} v_i \cdot \frac{w_i}{w_i} = \sum_{i \in S^*} w_i \cdot \frac{v_i}{w_i} \leq \sum_{i \in S^*} w_i \cdot \frac{v_{max}}{w_{max}} \leq C \cdot \frac{v_{max}}{w_{max}}.$$

Because we discard the items that cannot fit in the knapsack, the set S contains the most valuable item at least once, and $V(S) \geq v_{max}$. Let $n = \lfloor \frac{C}{w_{max}} \rfloor$ and recall that $V(S) = n \cdot v_{max}$. Then

$$(n + 1) \cdot v_{max} = V(S) + v_{max} \leq V(S) + V(S) = 2 \cdot V(S).$$

Finally, because $n = \lfloor \frac{C}{w_{max}} \rfloor$ implies that $(n+1) > \frac{C}{w_{max}}$, we have

$$V(S^*) \leq C \cdot \frac{v_{max}}{w_{max}} < \frac{w_{max} \cdot (n+1) \cdot v_{max}}{w_{max}} = (n+1) \cdot v_{max} \leq 2 \cdot V(S).$$

□

2.4 Runtime

The runtime of the algorithm is $O(n)$, where $|I| = n$. Indeed, the algorithm

1. Iterates through the n items in the for-loop. Inside the for-loop, the algorithm makes two comparisons which is done in $O(1)$ time. Hence the whole for-loop takes $O(n)$ time.
2. After the for loop, the algorithm computes two arithmetic operations in $O(1)$ time.

Recall that the algorithm we designed in Week 3 had runtime $O(nC)$, so the runtime has been significantly improved. The memory consumption has been reduced too: Algorithm 21 only requires $O(1)$ additional memory, while the memory consumption of the algorithm in Week 3 was $O(C)$.

3 Set cover

The formal definition of the set cover problem is quite abstract and mathematical but it comes with many real-life interpretations such as virus detection and code testing. Many graph problems are variations of the set cover problem. It is also an *NP-complete*[1] problem, which means that any NP-hard problem can be represented as a variation of the set cover problem.

There are no sub-polynomial algorithms solving the set cover problem exactly. In this section, we design an approximation algorithm for the problem running in $O(n \log n)$ time. This is a significant improvement compared to any exact algorithm solving the problem. The approximation ratio of the algorithm is $\Omega(\log n)$, and it is known that no greedy algorithm can construct better approximations than this.

3.1 Problem description

Let $U = \{1, \ldots, n\}$ be a set of items, and $\{S_1, \ldots, S_m\}$ be a set of subsets of U. In the set cover problem, the goal is to find the minimal number of sets S_i such that their union covers all elements of U. In other words, find a minimum size $I \subseteq \{1, \ldots, m\}$ such that

$$\bigcup_{i \in I} S_i = U.$$

To guarantee that a solution exists, all $u \in U$ must belong to at least one subset S_i.

[1]We return to the subject of NP-completeness in Week 10.

This problem can be equivalently described using bipartite graphs: let G be a bipartite graph with the vertex set $V = S \cup U$. Find the smallest number of vertices $I \subseteq S$ such that all vertices $u \in U$ have a neighbor in I.

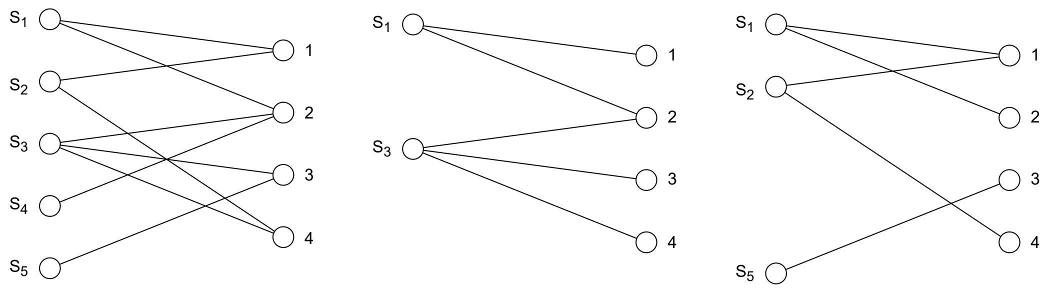
To turn a set cover problem into the bipartite graph representation, we construct the following graph.

1. Each element in U gets assigned a vertex.
2. Each subset in $S = \{S_1, \dots, S_m\}$ gets assigned a vertex.
3. There is an edge between the vertex $u \in U$ and the vertex S_i if and only if $u \in S_i$.

Example 3.1. Let $U = \{1, 2, 3, 4\}$ and the subsets be the following:

- $S_1 = \{1, 2\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 3, 4\}$
- $S_4 = \{2\}$
- $S_5 = \{3\}$.

This setting can be represented as a bipartite graph below. Because no set S_i contains all of the items, the optimal cover must have size of at least two. One feasible solution is $\{S_1, S_3\}$, and hence the optimum is two. An example of a feasible, non-optimal solution is $\{S_1, S_2, S_5\}$ which is a 1.5-approximation.

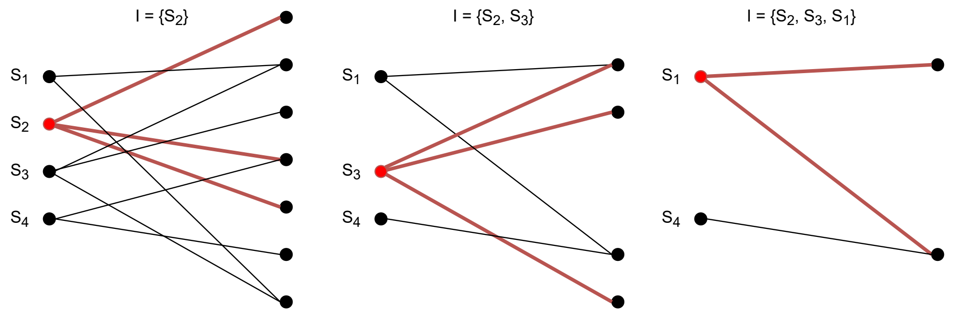


3.2 Algorithm design

As we want to minimize the number of subsets in the cover, the natural approach to maximize the marginal profit is to choose as large subsets as possible. In the bipartite graph representation, this means choosing the nodes of S with the highest degree. We base our greedy algorithm on this idea: in each iteration we add the vertex of S with the highest degree. After adding a vertex to the cover, we need to remove all its neighbors from the graph, because at each step we want to maximize the number of still uncovered nodes. We continue adding vertices of S until there are no more uncovered vertices in U. The algorithm is described as pseudocode in Algorithm 22.

```
Algorithm 22: Greedy Set Cover
Input: Bipartite graph $G = (S \cup U, E)$.
Output: Subset $I \subseteq S$ such that all $u \in U$ have a neighbor in $I$.
$I \leftarrow \emptyset$
while $\exists u \in U$ do
    Let $v \in S$ be the node with the largest degree.
    $I \leftarrow I \cup v$
    Remove $v$ and $N(v)$ from $G$.
end
return $I$
```

Example 3.2. Let's run the algorithm for an example graph. In case of ties, we choose one of the sets.



3.3 Correctness

Again, we need to show the following three things.

1. The algorithm terminates.
2. The output is a set cover.
3. The cover is at most $O(\log n)$ times bigger than the optimum cover.

Lemma 3.1. Algorithm 22 terminates.

Proof. We may assume, that $\forall u \in U \; \exists v \in S$ such that $u \in N(v)$: if this does not holds, a set cover does not exist. In each iteration, we add at least one new node to the cover. Because we remove the nodes that are neighbors to the added node, we end up removing all the nodes of U latest after adding all the vertices of S to our cover. Then the condition of the while-loop is false and the algorithm terminates. □

Lemma 3.2. The output of Algorithm 22 is a set cover.

Proof. The algorithm terminates when there are no vertices in U. Because vertices are removed only if they have a neighbor in I, in the end, all the vertices in U must be covered by I. □

To prove that the algorithm returns a $O(\log n)$-approximation, we give an alternative way to count the size of the output by iterating over the vertices of U. Then using algebraic manipulations, we end up with the desired bound.

Lemma 3.3. The output of Algorithm 22 is an $O(\log n)$-approximation.

Proof. Let I be the set of vertices in the output of Algorithm 22. Consider a node $v \in I$. The "cost" of adding the node v to the cover is one, as it increases the size of the cover by one. In the iteration v was added, we can count the cost of adding the node v by giving the remaining neighbors of v the cost $\frac{1}{\deg(v)}$. After that, the vertices $N(v)$ will be removed from the graph, and no other node $v' \in I$ can give them additional cost. Let $c(u)$ denote the cost assigned to the node $u \in U$. We can now count the vertices in I by

$$|I| = \sum_{u \in U} c(u).$$

Now, let I^* be the optimal solution. Because I^* covers all vertices, we have

$$U = \bigcup_{v \in I^*} N(v),$$

and we can iterate over the vertices of U by iterating over the neighbors of I^*. We get

$$|I| = \sum_{u \in U} c(u) \leq \sum_{v \in I^*} \sum_{u \in N(v)} c(u).$$

There is an inequality, as a node $u \in U$ can have multiple neighbors in I^*. Now

$$\sum_{v \in I^*} \sum_{u \in N(v)} c(u) \leq \sum_{v \in I^*} \max_v \left\{ \sum_{u \in N(v)} c(u) \right\} = |I^*| \cdot \max_v \left\{ \sum_{u \in N(v)} c(u) \right\}.$$

Now we just need to show that $\max_v \left\{ \sum_{u \in N(v)} c(u) \right\} = O(\log n)$. Suppose that a node v has d uncovered neighbors at this iteration. Because the algorithm always chooses the node with the most neighbors left, we have two options:

1. The node v is chosen to I and all of its neighbors get cost $1/d$.
2. Some other node v' with $\deg(v') \geq d$ is chosen and some neighbors of v may get cost $1/\deg(v') \leq 1/d$.

This means, that the neighbors of v can have a cost larger than $1/d$ only if v has less than d neighbors left. In the worst case, the neighbors of v will be covered one by one each with cost $1/i$ for $i = 1, \ldots, \deg(v)$. We get the bound[2]

$$\sum_{u \in N(v)} c(u) \leq \sum_{i=1}^{\deg(v)} \frac{1}{i} = O(\log \deg(v)),$$

[2]To prove that the harmonic series $\sum_{i=1}^{k} \frac{1}{i}$ is $O(\log k)$, one can for instance approximate it using the integral $\int \frac{1}{x} dx = \log x$.

where $\deg(v)$ denotes the original degree of v. Because $\deg(v) \leq |U| = n$, we have $O(\log\deg(v)) = O(\log n)$. □

Notice that now we have proven that the output is at most $O(\log n)$ times worse than the optimal cover. Next, we aim to prove that this is in fact tight, ie. there exist cases when the output is $\Omega(\log n)$ times the optimum In the following proof, we construct a family of input graphs that gives a $\Omega(\log n)$-approximation.

Lemma 3.4. In the worst case, Algorithm 22 returns $\Theta(\log n)$-approximation.

Proof. Let $G = (S \cup U, E)$ be a bipartite graph with $U = \{u_1, \ldots, u_{2^k-1}\}$ and $S = \{S_0, \ldots, S_k, S_b, S_r\}$. The sets S_b and S_r are special sets defined as

$$S_b = \{u_i \in U : i \text{ is even}\} \text{ and } S_r = \{u_i \in U : i \text{ is odd}\}.$$

The rest of the sets are defined by

$$S_j = \{u_i \in U : 2^{j-1} \leq i \leq 2^j - 1\}.$$

This setting is represented in Figure 5.1. The choice $\{S_b, S_r\}$ is the optimal solution to the problem.

Notice that the sets S_k, S_b, and S_r all have 2^{k-1} neighbors, and hence are equally good options for the first iteration of the greedy algorithm. Suppose that the greedy algorithm happens to choose S_k – we are allowed to assume this, as we are looking for a worst-case scenario. The degree of S_b and S_r drops to 2^{k-2}, and for the next iteration, S_b, S_r, and S_{k-1} are equally good options. Suppose that in each iteration, the algorithm continues choosing the set S_i over S_b and S_r, and in the end, it returns a cover with $\log n$ sets, since $|\{S_0, \ldots, S_k\}| = \log n$. The optimal cover has size two, and hence the output of the algorithm is a $\Theta(\log n)$-approximation.

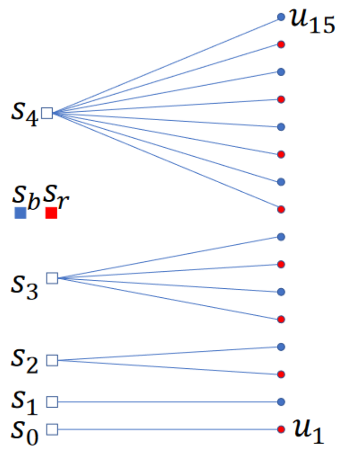


Figure 5.1: The sets $S_0, \ldots, S_4$ are represented using edges, the vertices in the sets S_b and S_r are color-coded.

□

3.4 Runtime

The following proof does not aim to give the tightest bound for the runtime, the idea is just to demonstrate that the runtime is polynomial. This is an improvement to any exact algorithm, as the problem is NP-hard.

Lemma 3.5. The runtime of Algorithm 22 is $O(m^2n)$.

Proof. Let $|S| = m$ and $|U| = n$. First of all, the while-loop runs m times in the worst case, as we add one node to the cover each iteration, and might need to add all of the vertices to the cover. The condition of the while-loop can be checked in $O(1)$ time. Inside the while-loop, we need to

1. Find the node with highest degree. We can compute the degree of the nodes in S by iterating over the edges. The number of edges is bounded by $O(mn)$.

2. Remove one node from S and its neighbors from U. We may represent the graph as a $m \times n$ adjacency matrix, where the nodes of U correspond to columns and nodes of S to rows. Using an appropriate data structure, we may remove a node from U in $O(1)$ time[3]. The number of nodes to be removed is bounded by n. To remove one node from S, we need to remove a row from the adjacency matrix, which takes $O(n)$ time. Hence the runtime of the removals is $O(n)$.

Putting this all together gives the runtime $O(m) \cdot (O(mn) + O(n)) = O(m^2n)$.

□

[3]For example using linked lists.

6 Randomized algorithms

Consider the following algorithm for finding a given element k from an array.

Algorithm 23: Search($A[1, \dots, n], k$)

```
$I \leftarrow \{1, \dots, n\}$
$i \leftarrow$ random element from $I$
while $A[i] \neq k$ do
    $I \leftarrow I \setminus \{i\}$
    $i \leftarrow$ random element from $I$
end
return $i$
```

Instead of going through the array from the beginning to the end, in each iteration, we check a new random index from the array A. In the worst case, we end up going through all the indices of A before finding k, which gives us the worst-case runtime of $O(n)$. The deterministic algorithm that systematically iterates over the array A has exactly the same runtime, so the designs can be considered equally good. On the other hand, in some cases, the randomized algorithm could be considered better: if the array A and the element k in our application behave so that k is almost always at the end of the array, the deterministic algorithm always needs to iterate through the whole array, while the randomized version needs $n/2$ array accesses *on average*.

Algorithms that use randomness in some part of their implementation are called *randomized algorithms*. Randomness can appear in many different forms in the algorithm. For instance, the runtime of the algorithm might depend on randomness or the algorithm might fail to solve the problem with a certain probability. To be able to analyze randomized algorithms, we need to be able to model probabilistic phenomena and compute the probabilities related to the algorithm. We also need to define what it means that the algorithm does something *on average* and how far can we *expect* to be from the average. To do all this, we introduce some basic probabilistic tools.

The purpose of this week is not to dive deep into probability theory. Instead, the goal is to give a reminder of some basic probability theory results. These basic results form a powerful tool to analyze and design algorithms using randomness.

1 Probability basics

Because we are interested in using probability theory only in algorithm design, we mostly focus on the discrete versions of different definitions. This roughly means that most of the structures we inspect are either finite or countable.

1.1 General definitions

Definition 1.1 (Probability space). A probability space is a triplet (Ω, F, P), where

probability space

1. The set Ω is called *sample space*. The elements in Ω are called *outcomes*. For example, for a single coin toss, the possible outcomes are "heads" and "tails", so we can write $\Omega = \{H, T\}$.

sample space
outcome

2. The set F is the *event space* and it consists of different subsets of Ω called *events*. Formally F has to satisfy very specific properties[1], but on this course, it is enough to assume that F contains all subsets of Ω including $\emptyset$ and Ω itself. As an example for the coin toss, some of the events are

event space
event

   - $\emptyset =$ "the coins is neither heads nor tails"
   - $\{H\} =$ "the coin is heads"
   - $\{H, T\} =$ "the coin is heads or tails"

3. The function $P : F \to [0, 1]$ is the *probability function*[2] and it outputs the probability of a given event. It gives values between zero and one, zero meaning an impossible event and one meaning a sure event. For example, for a fair coin toss, we have

probability function

   - $P(\emptyset) = 0$. The coin must be either heads or tails, so this event is impossible. For similar reasons, $P(\{H, T\}) = 1$, which makes $\{H, T\}$ a *sure event*.
   - $P(\{H\}) = \frac{1}{2}$. The coin toss is fair, so heads and tails are equally likely events.

sure event

We say that the event A happens if it contains an outcome that happens. Because events are sets, we can construct new events using set operations. Let $A, B \in F$ be events.

1. Union $A \cup B$ means "Event A happens or event B happens".
2. Intersection $A \cap B$ means "Event A happens and event B happens".
3. Complement of A, denoted by A^c, means "Event A does not happen".
4. Let $A_1, \ldots, A_n \in F$. Then $\bigcup_{i=1}^{n} A_i = A_1 \cup \cdots \cup A_n$ means "at least one of the events A_i happens".

union of events
intersection of events
complement event

Next, we introduce basic rules the probability function must follow. We will not prove

[1]Mathematically, F is a *sigma-algebra*, but there is no need to go in-depth into that. Those who are interested are free to research the concept.

[2]The probability function is mathematically a *measure*.

these results, as they all follow directly from the mathematical definitions.

Corollary 1.1. Let (Ω, F, P) be a probability space. Let $A, B \in F$. Then

- $\mathrm{P}(\Omega) = 1$ and $\mathrm{P}(\emptyset) = 0$.
- If $A \cap B = \emptyset$, ie. A and B are disjoint events, then $\mathrm{P}(A \cup B) = \mathrm{P}(A) + \mathrm{P}(B)$. This result holds for arbitrarily many pairwise disjoint sets.
- The complement of the event A is defined as $A^c = \Omega \setminus A$. The probability of the complement event is $\mathrm{P}(A^c) = 1 - \mathrm{P}(A)$.
- If $A \subseteq B$, then $\mathrm{P}(A) \leq \mathrm{P}(B)$.

Example 1.1 (Two even coin tosses)**.** Suppose that we throw two fair coins. The sample space is $\{HH, HT, TH, TT\}$, and each outcome in the sample space happens with the same probability $1/|\Omega|$. Now consider the event "at least one of the coins is heads". In set notation, it is

$$A = \{HH, HT, TH\}.$$

The set A is the union of the three disjoint events $\{HH\}$, $\{HT\}$, and $\{TH\}$, and we get the probability

$$\mathrm{P}(A) = \mathrm{P}(\{HH\} \cup \{HT\} \cup \{TH\}) = \mathrm{P}(\{HH\}) + \mathrm{P}(\{HT\}) + \mathrm{P}(\{TH\}) = \frac{3}{4}.$$

Now we are ready to present our first probabilistic inequality. As we saw earlier, the probability of the union of disjoint events is easy to compute. Unfortunately, we encounter non-disjoint events all the time (eg. events $\{HT, HH\}$ and $\{TH, HH\}$ are not disjoint, as they both contain the outcome HH), and the exact probabilities might be difficult to compute. Luckily, we have the following inequality called *union-bound* which allows us to estimate the probability of the union when the events are not necessarily disjoint. It is a powerful tool in algorithm design, as unions of events frequently appear in the runtime analysis of probabilistic algorithms. Figure 6.1 shows a situation where the inequality is not tight.

Lemma 1.1 (Union-bound)**.** Let $A_1, \ldots, A_n$ be events. Then *union bound*

$$\mathrm{P}\left(\bigcup_{i=1}^{n} A_i\right) \leq \sum_{i=1}^{n} \mathrm{P}(A_i).$$

Recall that for pairwise disjoint events, the inequality is tight. Sometimes the union bound is not powerful enough, and the right-hand side of the equation may well end up being much larger than 1, hence giving a trivial bound. Generally, the individual probabilities of the events must be small enough to construct a non-trivial bound.

Example 1.2. Let A_i be the event that the ith day of the week is rainy. Suppose that based on data from previous years, we can approximate the probability of rain as $\mathrm{P}(A_i) = 0.05$ for all $i = 1, \ldots, 7$. Then, the probability that at least one day is rainy is at most

$$\mathbf{P}\left(\bigcup_{i=1}^{7} A_i\right) \leq \sum_{i=1}^{7} \mathbf{P}(A_i) = 7 \cdot 0.05 = 0.35.$$

This means that the probability of the whole week being free of rain is at least 65%:

$$1 - \mathbf{P}\left(\bigcup_{i=1}^{7} A_i\right) \geq 0.65.$$

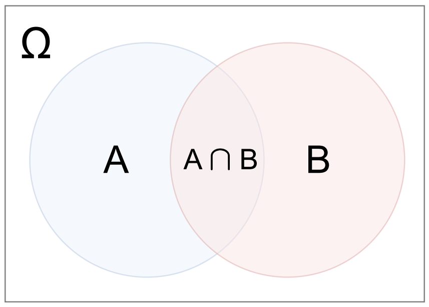


Figure 6.1: The square represents the sample space Ω, the blue circle is the event A, and the red circle is the event B. The probabilities are represented as areas of the circles. Because $A \cap B \neq \emptyset$, we have $\mathbf{P}(A \cup B) = \mathbf{P}(A) + \mathbf{P}(B) - \mathbf{P}(A \cap B) < \mathbf{P}(A) + \mathbf{P}(B)$.

1.2 Independence and conditional probability

Earlier we talked about independent events as events that do not contain same outcomes. Generally, independence means that the events do not affect each other. For example, if we throw a coin twice and we know that the first throw was tails, we can determine that the event of throwing two heads is now impossible: the knowledge of one event happening has affected the probability of some other event.

The following definition allows us to determine independence based on the probability function. From another point of view, it can be used to easily compute the probabilities of intersections when the events in question are known to be independent of each other.

Definition 1.2 (Independent events)**.** The events $A_1, \ldots, A_n$ are called *mutually independent* if and only if for all subsets $I \subseteq \{1, \ldots, n\}$ we have

independence

$$\mathbf{P}\left(\bigcap_{i \in I} A_i\right) = \prod_{i \in I} \mathbf{P}(A_i).$$

Remark. Independency is not the same as events being disjoint. In fact, non-trivial disjoint events are never independent: let $A \cap B = \emptyset$, $\mathbf{P}(A) \neq 0$, and $\mathbf{P}(B) \neq 0$. Then

$$\mathbf{P}(A \cap B) = \mathbf{P}(\emptyset) = 0 \neq \mathbf{P}(A)\mathbf{P}(B).$$

Example 1.3 (Independent coin tosses)**.** Suppose we throw an even coin n times and the throws are independent of each other (the result of a coin toss cannot affect the result of future tosses). Let $H_i =$ "on ith throw, the coin lands on heads". The probability, that all of the throws land on heads is

$$P\left(\bigcap_{i=1}^{n} H_i\right) = \prod_{i=1}^{n} P(H_i) = \prod_{i=1}^{n} \frac{1}{2} = 2^{-n}.$$

In real life, some events happen in steps. In those cases, we are able to analyze the probabilities of the later events knowing that some events have already happened. For instance, consider a coin that may not be fair. After throwing ten tails with the coin – which is relatively unlikely for a fair coin – we might suspect that the coin is favorable for tails. This phenomenon is called *conditional probability*.

conditional probability

Definition 1.3 (Conditional probability)**.** Let $A, B \in F$ and $P(B) > 0$. The conditional probability $P(A \mid B)$ is the probability of event A happening knowing that event B already happened. It is defined by

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

Notice, that when A and B are independent, we have

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A).$$

Indeed, if the events A and B are independent, knowing that event B has happened does not affect event A.

Example 1.4. You play a game where you win if you throw three heads in a row. The coin is fair and the throws are independent of each other. Let $H_i =$ "ith throw is heads" and $T_i =$ "ith throw is tails". The probability of winning the game is P("you win") $= \frac{1}{2^3} = 1/8$. Suppose that we are lucky and throw heads on the first throw. Then the probability of winning is

$$P(\text{"you win"} \mid H_1) = \frac{P(\text{"you win"} \cap H_1)}{P(H_1)} = \frac{P(H_1 \cap H_2 \cap H_3)}{P(H_1)} = \frac{1/2^3}{1/2} = \frac{1}{4}.$$

Unfortunately, the second throw is tails. Then, as expected, the probability of winning is

$$\begin{aligned} P(\text{"you win"} \mid H_1 \cap T_2) &= \frac{P(\text{"you win"} \cap (H_1 \cap T_2))}{P(H_1 \cap T_2)} \\ &= \frac{P\left((H_1 \cap H_2 \cap H_3) \cap (H_1 \cap T_2)\right)}{P(H_1 \cap T_2)} = 0, \end{aligned}$$

because $P(H_2 \cap T_2) = 0$ due to H_2 and T_2 being disjoint.

Corollary 1.2. Let $B \in F$ and $A_1, \ldots, A_n \in F$ be a *partition* for the sample space Ω, ie. the events are pairwise disjoint and cover the whole sample space. Then

$$P(B) = \sum_{i=1}^{n} P(B \cap A_i) = \sum_{i=1}^{n} P(B \mid A_i) \cdot P(A_i).$$

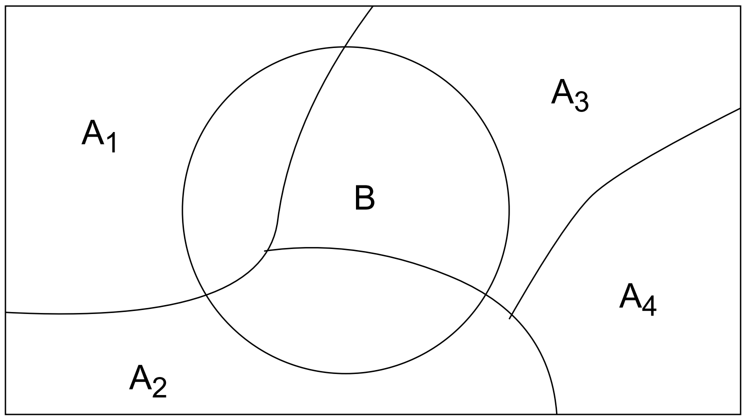


Figure 6.2: Suppose the sample space can be divided into sets A_1, A_2, A_3, and A_4 as in the figure. We can represent the probability of the set B in the middle as a sum of the parts in each A_i, ie. $P(B) = P(A_1 \cap B) + P(A_2 \cap B) + P(A_3 \cap B) + P(A_4 \cap B)$.

Example 1.5. We have two coins: one that favors tails and one that favors heads. The coin that favors tails gives tails with the probability of 0.9 and heads with 0.1. The coin that favors heads works similarly. We choose one of the coins at random. Let A_h = "the coin favors heads" and A_t = "the coins favors tails". Let H_3 = "we throw three heads in a row". We have

$$\mathrm{P}(H_3) = \mathrm{P}(H_3 \mid A_h)\mathrm{P}(A_h) + \mathrm{P}(H_3 \mid A_t)\mathrm{P}(A_t) = \left(\frac{9}{10}\right)^3 \cdot \frac{1}{2} + \left(\frac{1}{10}\right)^3 \cdot \frac{1}{2} = 0.365.$$

For comparison, the probability of throwing three heads in a row with a fair coin is $2^{-3} = 0.125$.

1.3 Random variables

A random variable is a variable whose value depends on the randomness of the probability space. A random variable can generally take any kind of value, but the following definition concerns real-valued random variables.

Definition 1.4 (Random variable)**.** A real-valued random variable is a function $X : \Omega \to \mathbb{R}$. The probability of the random variable getting the value $a \in \mathbb{R}$ is determined by the probability function as *random variable*

$$\mathrm{P}(X = a) = \mathrm{P}(\{\omega \in \Omega \mid X(\omega) = a\}).$$

Example 1.6 (Indicator random variable)**.** Let $A \in F$ be an event. Then $\mathbf{1}_A$ is called the *indicator random variable* of A defined by *indicator variable*

$$\mathbf{1}_A(\omega) = \begin{cases} 1 & \text{when } \omega \in A \\ 0 & \text{when } \omega \notin A. \end{cases}$$

Example 1.7 (Tasks and servers)**.** Consider n identical servers and k computing tasks. Each server has an equal probability of being assigned a task and each task will be assigned independently. Let the random variables $X_1, \ldots, X_n$ denote the number of tasks each computer has after the tasks have been distributed. The probability that the server i has no tasks after all tasks have been distributed is given by

$$\begin{aligned} \mathrm{P}(X_i = 0) &= \mathrm{P}\left(\bigcap_{j=1}^{k} \text{“task } j \text{ is not assigned for server } i\text{”}\right) \\ &= \prod_{j=1}^{k} 1 - \overbrace{\mathrm{P}(\text{“task } j \text{ is assigned for server } i\text{”})}^{=\frac{1}{n} \text{ because server chosen uniformly}} \\ &= \left(\frac{n-1}{n}\right)^k. \end{aligned}$$

We can also construct random variables using arithmetic operations. Let X and Y be real-valued random variables. Then

- $X + Y$ and $X \cdot Y$ are random variables.
- $\max\{X, Y\}$ and $\min\{X, Y\}$ are random variables.

probability distribution

Definition 1.5 (Distribution)**.** The distribution of a random variable is the function $f_X : \mathbb{R} \to [0, 1]$ defined by

$$f_X(a) = \mathrm{P}(X = a).$$

Random variables describing real-life experiments often follow specific patterns, and the distributions modeling these phenomena have been studied a lot. Some of these well-known distributions are introduced in the example below.

Example 1.8. Some well-known random variables and their distributions are

uniform distribution

1. Uniform distribution. When X gets values from a set $A = \{a_1, \ldots, a_n\}$ the distribution is defined for all $a \in A$ as

$$f_X(a) = \frac{1}{n}.$$

bernoulli distribution

2. Bernoulli distibution. Random variable X is an indicator random variable. This means that $\mathrm{P}(X = 1) = p$ and $\mathrm{P}(X = 0) = 1 - p$. The distribution is hence

$$f_X(a) = \begin{cases} p, & \text{when } a = 1, \\ 1 - p, & \text{when } a = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Example: We may represent the event “it rains today” as an indicator random variable X such that X gets value 1 if it rains and 0 otherwise. Then X is a Bernoulli variable, where p is the probability of rain.

3. Binomial distribution. The random variable X is a sum of identically distributed indicator (or Bernoulli) random variables $X_1, \ldots, X_n$. The random varialbe X gets values from $\{1, \ldots, n\}$. Let $\mathrm{P}(X_i) = p$ for all $i \in \{1, \ldots, n\}$. Then the distribution of X takes the form *binomial distribution*
$$f_X(k) = \binom{n}{k} p^k (1-p)^{n-k},$$
where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

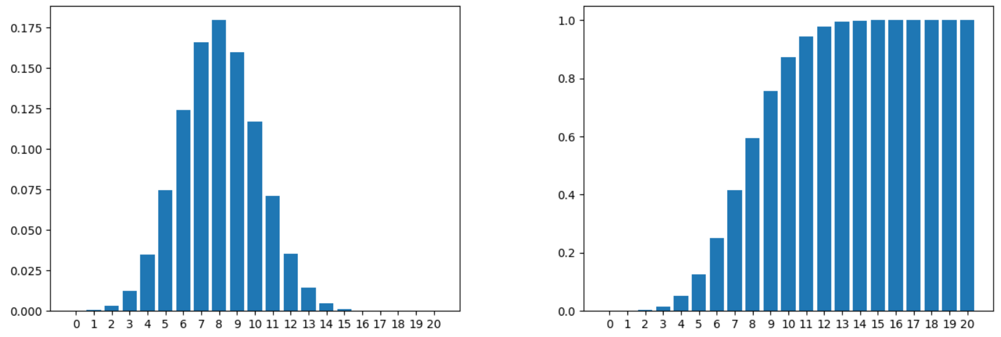
   Example: Consider the number of rainy days in a month, given that the probability of rain each day is p. If X_i is the indicator variable of it raining on ith day, then $X = \sum_i X_i$ is the number of rainy days, and it is distributed binomially.

4. Poisson distribution with parameter $\lambda > 0$. The random variable X is a non-negative integer with the distribution *poisson distribution*
$$f_X(k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$
The probability is at its highest when $k = \lambda$. After that, the probabilities quickly decrease as k increases.

   Example: Poisson distribution is used when modeling phenomena that can happen arbitrarily many times, but the probability of a relatively large number of occurrences is low. Examples of phenomena that can be modeled with Poisson distribution are the number of goals in a football game, lightning strikes on the same person, or the number of people that visit a certain museum in a year.

Definition 1.6 (Cumulative distribution function)**.** The cumulative distribution function (CDF) $F_X : \mathbb{R} \to [0, 1]$ for the random variable X is defined as *cumulative distribution function*
$$F_X(a) = \mathrm{P}(X \le a).$$

Example 1.9. In the figure below, on the left we can see the binomial distribution when $n = 20$ and $p = 0.4$. Each bar in the plot represents the value $f(i) = \binom{n}{i} p^i (1-p)^{n-i}$. The corresponding cumulative distribution function can be seen on the right. In the discrete case, the values for the cumulative distribution function can be computed from
$$F(a) = \mathrm{P}(X \le a) = \sum_{i=0}^{a} \binom{n}{a} p^a (1-p)^{n-a},$$
where $a \in \{0, \ldots, n\}$.



1.4 Expectation and variance

To be able to estimate the behavior of a random variable, we do not necessarily need to know its entire distribution. In some cases, it is enough to inspect the average behavior of the random variable. This is called the expectation of the random variable. It is practically a weighted average over the possible values the random variable can take.

expectation

Definition 1.7 (Expectation of a discrete random variable)**.** Let A be the set of different values the random variable X can take. Then the *expectation* or *expected value* of the random variable is defined by

$$\mathrm{E}[X] = \sum_{a \in A} a \cdot \mathrm{P}(X = a).$$

The expectation is sometimes denoted by μ.

Example 1.10. You play a game where you win if you can throw three heads in a row with a fair coin. To play the game, you need to buy a 1-euro ticket, and if you win, you get 10 euros. The expected profit you make by playing the game is

$$\mathrm{E}[\text{“profit”}] = \mathrm{P}(\text{“you win”}) \cdot (10 - 1) + \mathrm{P}(\text{“you lose”}) \cdot (-1) = \frac{1}{8} \cdot 9 - \frac{7}{8} = \frac{1}{4},$$

which roughly means that you are expected to gain 25 cents from playing the game.

As we can see from Example 1.10, the expectation does not tell anything about the actual distribution of the random variable: we cannot win 0.25 euros by playing the game once, we always either lose 1 euro or profit 9 euros. Even though the expectation is positive, the probability of losing money when playing the game once is still much higher than profiting from it.

The following property of expectation makes it easy to compute the expectation for sums of random variables.

Lemma 1.2 (Linearity of expectation)**.** Let $X_1, \ldots, X_n$ be random variables, and $a_1, \ldots, a_n$

be real numbers. Then

$$\mathrm{E}\left(\sum_{i=1}^{n} a_i \cdot X_i\right) = \sum_{i=1}^{n} a_i \cdot \mathrm{E}[X_i].$$

Example 1.11. Suppose you decide to play the game from Example 1.10 n times. Let the profit of the ith game be X_i. Then the expected profit of n games is

$$\mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \frac{n}{4}.$$

This means that the more you play the game, the more you are expected to win. For instance, after eight games, the expected profit is two euros.

Example 1.12 (Tasks and servers)**.** Suppose we have m computational tasks and n servers for computing the tasks. Each task gets assigned uniformly at random to a server. Let X_i denote the number of tasks the server i gets. The probability of the server i getting a certain task j is $1/n$, and hence the expected total number of tasks is

$$\mathrm{E}[X_i] = \sum_{j=1}^{m} \mathrm{E}[\text{“server } i \text{ gets } j\text{th task”}] = \sum_{j=1}^{m} \frac{1}{n} = \frac{m}{n}.$$

While expectation might not tell much about a single random event, the following result implies that for many repetitions the average value of identically distributed random variables tends towards the expectation.

Theorem 1.1 (The law of large numbers)**.** Let $X_1, \ldots, X_n$ be identically distributed random variables with $X_i = \mu$ for all $i \in \{1, \ldots, n\}$. Then

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} X_i = \mu.$$

To get more information about the distribution of the random variable, we might want to know how far away from the expectation the actual values of the random variable usually are. The variance and standard deviation of a random variable are used to describe this.

Definition 1.8 (Variance and standard deviation)**.** The *variance* of a random variable X is defined by *variance*

$$\mathrm{Var}(X) = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - E[X]^2.$$

The *standard deviation* is the square root of the variance, $\sigma(X) = \sqrt{\mathrm{Var}(X)}$. The standard deviation is sometimes denoted just by σ. *standard deviation*

Lemma 1.3 (Variance of independent random variables)**.** Let $X_1, \ldots, X_n$ be independent random variables. Then

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\left(\sum_{i=1}^{n} \mathrm{Var}(X_i)\right).$$

Example 1.13. Let's compute the variance of n repetitions of the game in Example 1.10. The expectation of X_i^2 is $\mathbf{E}[X_i^2] = \frac{1}{8} \cdot 9^2 + \frac{7}{8} \cdot (-1)^2 = \frac{81+7}{8} = 11$, and hence the variance is given by

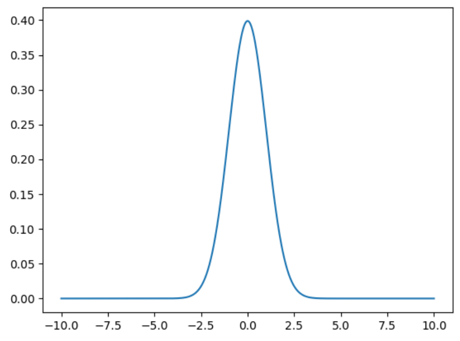
$$\text{Var}(X_i) = \mathbf{E}[X_i^2] - \mathbf{E}[X_i]^2 = 11 - \frac{1}{4} = \frac{43}{4}.$$

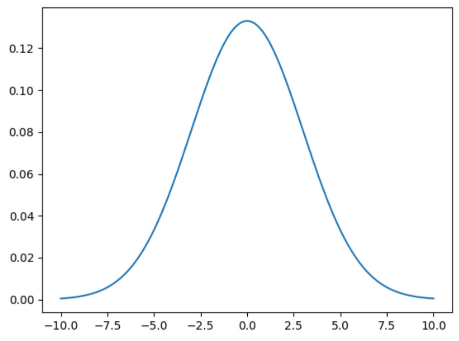
As the games are independent, we may use Lemma 1.3 to compute the variance of the total profits. We have

$$\text{Var}\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2} \cdot n \cdot \frac{43}{4} = \frac{43}{4n}.$$

We can see that when n increases, the variance tends to zero. This has a nice interpretation: the more we play the game, more likely it is that we are close to the expected value.

Example 1.14 (Normal distribution)**.** In the figure below, one can see two plots of distribution called *normal distribution*. Both of the distributions have the expected value 0 but the distribution on the left has smaller variance than the one on the right. This affects the curve so that the values are more likely to be closer to the expectation when the variance is lower.





2 Probability in algorithm design

Generally, there are two types of randomized algorithms:

1. *Monte Carlo algorithms*, where the correctness of the algorithm is probabilistic. For example, the Solovay-Strassen primality test picks a random integer and checks a certain condition to verify that the number is maybe a prime. This check is repeated multiple times to increase the probability that the number is prime. Without testing all integers, we cannot guarantee that the number is really prime, but not checking every integer allows us to improve the runtime. We may choose to check *large enough* number of integers, so that the probability that the number is prime might be good enough for our application, but the runtime remains reasonable.

Monte Carlo algorithm

2. *Las Vegas algorithms*, where the runtime of the algorithm is probabilistic. For example, Algorithm 23 from the introduction is a Las Vegas algorithm.

Las Vegas algorithm

For Las Vegas algorithms, to get an estimate of the efficiency of an algorithm, it is reasonable to analyze its expected runtime. By Theorem 1.1, this means that if the algorithm is run multiple times, on average it will run in the expected time. Sometimes we are also interested in finding *almost guaranteed* probabilistic runtimes. Usually, the probabilities also depend on the size of the input. The following definition gives kind of a probabilistic equivalent for the asymptotic notion.

Definition 2.1 (With high probability)**.** Let A be an event whose probability depends on the parameter $n \in \mathbb{N}$. We say that A happens *with high probability*, if for any constant $c > 0$

with high probability

$$\mathrm{P}(A) \geq 1 - \frac{1}{n^c}.$$

This means that if an event happens with high probability, we can essentially assume that it happens almost surely as long as the input size n is large enough.

Example 2.1 (Random graphs)**.** Suppose that a vertex $v \in V$ has at least one neighbor with high probability, ie. for any $c' > 0$ we have

$$\mathrm{P}(\text{“}v \text{ has at least one neighbor”}) \geq 1 - \frac{1}{n^{c'}},$$

where $n = |V|$. Let X_i be the event “vertex i has no neighbors”. Let $c > 0$ be any constant. The probability that there exists at least one vertex with no neighbors is

$$\mathrm{P}\left(\bigcup_{i=1}^{n} X_i\right) \leq \sum_{i=1}^{n} \mathrm{P}(X_i) \leq \sum_{i=1}^{n} \frac{1}{n^{c'}} = \sum_{i=1}^{n} \frac{1}{n^{c+1}} = \frac{1}{n^c}.$$

In the third step, we use the definition of “with high probability” for constant $c' = c+1$.

This means that

$$\mathrm{P}(\text{“every vertex has at least one neighbor”}) = 1 - \mathrm{P}\left(\bigcup_{i=1}^{n} X_i\right) \geq 1 - \frac{1}{n^{c'}},$$

and every vertex has at least one neighbor with high probability.

Example 2.2. Consider a hypothetical algorithm that runs in $O(1)$ time with high probability, ie. with probability $1 - \frac{1}{n^c}$ for any $c > 0$. Let the probability that the algorithm runs in $\Theta(n^{c+1})$ time be $\frac{1}{n^c}$. Then the expected runtime of the algorithm would be

$$O(1) \cdot \left(1 - \frac{1}{n^c}\right) + \Theta(n^{c+1}) \cdot \frac{1}{n^c} = O(1) - O\left(\frac{1}{n^c}\right) + \Theta\left(\frac{n^{c+1}}{n^c}\right) = \Theta(n).$$

Hence a good runtime with high probability does not guarantee a good runtime in expectation.

3 Application: Quicksort

Quicksort is a recursive sorting algorithm with probabilistic runtime, ie. a Las Vegas algorithm. In this section, we start by introducing the deterministic version of Quicksort that in the worst case has $O(n^2)$ runtime, but on average finishes in time $O(n \log n)$. Then we make small improvements to the algorithm and show that it reaches the runtime of $O(n \log n)$ in expectation and $O(n \log n)$ with high probability.

On the other hand, the merge sort has a runtime of $O(n \log n)$ deterministically. Even though the asymptotic runtime is the same and the merge sort could be considered to be a better option, Quicksort tends to be faster than the merge sort in practice: unlike the merge sort, it does not need additional memory to work, and it behaves well with caching.

3.1 Deterministic algorithm

For simplicity, we assume that all the elements in the array are distinct. The deterministic algorithm consists of three steps.

1. Choose any element in the array and call it a *pivot*. The pivot is chosen deterministically for instance from the end of the array.
2. Move elements smaller than the pivot to the left side of the pivot, and the elements larger than pivot to the right side of the pivot. This is called *partitioning*.
3. Recursively sort the elements on the left and right sides of the pivot.

Notive that the partition step can be done by modifying the original array, and we do not need an auxiliary array as with the merge sort.

The base case happens when the array has at most one element and hence is already sorted. Because the pivot is excluded from the subproblems, they are guaranteed to be strictly smaller than the original array. The details of the algorithm can be seen in Algorithm 25.

Algorithm 24: Partition(A, p)

```
Move $p$ to the first index $A[0]$.
$i \leftarrow 1; j \leftarrow n-1$ /* Keep track of current index and end of the list */
while $i \leq j$ do
    if $A[i] > p$ then
        Swap $A[i]$ and $A[j]$.
        $j \leftarrow j-1$.
    else
        $i \leftarrow i+1$.
    end
end
Swap $A[0]$ and $A[i-1]$.
return $i-1$ /* Return the location of pivot */
```

Algorithm 25: Quicksort(A)

```
$n \leftarrow |A|$
if $n \leq 1$ then
    /* List already sorted */
    return
else
    Set the pivot $p \leftarrow A[i_p]$. /* $i_p$ is deterministic, eg. 1 or $|A|$ */
    $i \leftarrow$Partition$(A, p)$ /* $i$ is the new location of the pivot */
    Quicksort($A[0, \ldots, i-1]$)
    Quicksort($A[i+1, \ldots, n-1]$)
```

Lemma 3.1. Algorithm 24 partitions the array A correctly.

Proof. It is enough to show that throughout the algorithm we have $A[k] < p$ for all $1 < k < i$ and $A[k] > p$ for all $k > j$. Recall that we assumed that all the elements in the array are distinct, and hence no element can be equal to the pivot.

Base case: At the beginning of the algorithm $i = 1$ and $j = n-1$, so the claim holds trivially.

Induction hypothesis: Suppose that in some iteration, $A[k] < p$ for all $1 < k < i$ and $A[k] > p$ for all $k > j$ holds before going inside the while-loop.

Induction step: We show that the induction hypothesis holds after executing the inside of the while-loop. Inside the while-loop, we have two cases:

1. $A[i] > p$ and we enter the if-clause. We swap the elements $A[i]$ and $A[j]$. $A[k] < p$ still holds for all $1 < k < i$ because i was not changed and elements in $A[1, \ldots, i-1]$ stayed intact. Also $A[k] > p$ holds for all $k > j$ by the induction hypothesis. After swapping the elements at i and j, we have $A[j] > p$, and we can safely decrement j by one.

2. The if-statement is false and $A[i] < p$. We can set $i \leftarrow i+1$ and $A[1, \ldots, i-1] < p$ still holds.

We exit the while loop when $i > j$, which happens when $i = j + 1$. This means that $A[1, \ldots, i-1] < p$ and $A[i, \ldots, n-1] > p$. After swapping the elements $A[1]$ and $A[i-1]$, the pivot is at index $i - 1$, everything on its left side is smaller than p, and everything on its right side is larger than p. Hence partitioning is correct. $\square$

As the following proof of correctness is very similar to the one for the merge sort, it is suggested to prove the correctness as an exercise before taking a look at the proof below.

Lemma 3.2. Algorithm 25 is correct.

Proof. The algorithm terminates by induction, as the recursive calls get strictly shorter arrays as input and hence eventually reach the base case $|A| \leq 1$. To show that the algorithm sorts the array A, we use induction on the length of the array A.

Base case: Let $|A| \leq 1$. Then the array is already sorted and the algorithm is correct by doing nothing.

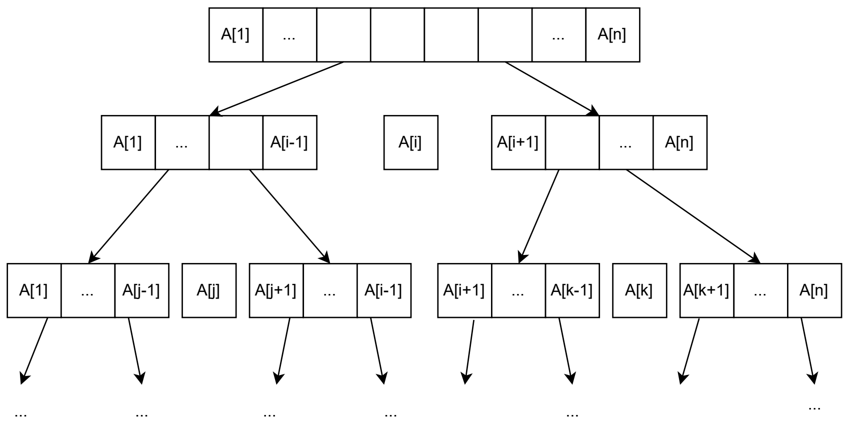
Induction hypothesis: Suppose the algorithm works correctly for all arrays with $|A| < k$.

Induction step: Let $|A| = k$. After calling partition, all elements in $A[0, \ldots, i-1]$ are smaller than p, and all elements in $A[i+1, \ldots, k-1]$ are larger than p. Because the arrays $A[0, \ldots, i-1]$ and $A[i+1, \ldots, k-1]$ are shorter than k, the algorithm sorts them correctly by the induction hypothesis. Because all elements in $A[0, \ldots, i-1]$ are smaller than p, the array $A[0, \ldots, i]$ is also sorted. Similarly, because $A[i+1, \ldots, k-1]$ are larger than p, the whole array $A[0, \ldots, i, i+1, \ldots, k-1]$ is sorted. $\square$

Lemma 3.3. The runtime of Algorithm 25 is $O(n \log n)$ in the best case and $O(n^2)$ in the worst case.

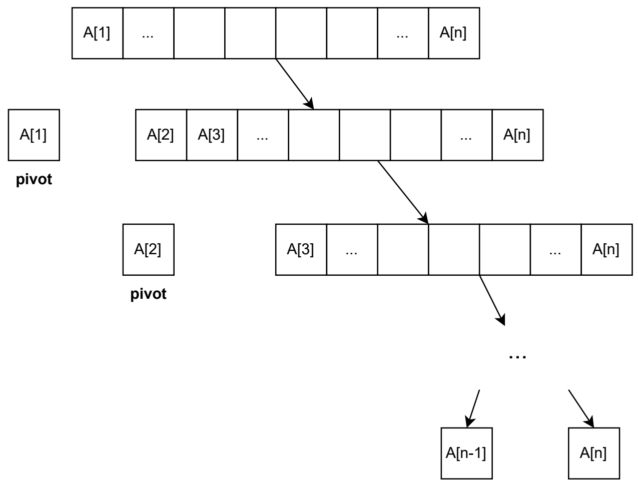
Proof. Let's first analyze the runtime of the partition. In each iteration, we either decrease the index j or increase the index i and hence we handle exactly one element until all elements have been handled. This means that an array of size k takes $O(k)$ time to partition.

Next, let's take a look at the recursion tree of the algorithm. In each iteration, we split the input into two parts, one with size $|A[0, \ldots, i-1]|$ and other with size $|A[i+1, \ldots, n-1]|$, until the array has been split into arrays of one element.



Similarly to the merge sort, each layer of the tree has at most n elements, and hence partitioning one layer takes $O(n)$ time. The total runtime is bounded by $O(nd)$ where d is the depth of the tree. To minimize the depth of the tree, in each iteration the partition has to divide the array in half. This would result in the same runtime as the merge sort.

Unfortunately, we reach the runtime of $O(n \log n)$ only when *in each recursive call* approximately half of the elements are smaller than the pivot and half of them are larger. In the worst case, the pivot is the smallest or largest element of the array in each iteration, and we end up with the following recursion tree.



The depth of this worst-case tree is $O(n)$, and as partition takes $O(n)$ time on each layer, we end up with worst-case runtime of $O(n^2)$. □

As we saw the sizes of the partitions are entirely determined by the pivot, and it is not far-fetched to assume that deteministically chosen pivot could always be the worst one: the input data might naturally be in such an order that the largest or smallest elements are located in a specific part of the array. To guarantee that the runtime is almost always $O(n \log n)$, we make a small modification to the algorithm concerning the choice of the pivot.

3.2 Paranoid Quicksort

For the randomized version of Quicksort we give the definition of a *good pivot*.

Definition 3.1. We call pivot p *good* if it is larger and smaller than 10% of the elements in the array. In other words, if the pivot is the ith smallest element in the array, p is good if $|A|/10 < i < 9 \cdot |A|/10$. If the size of the array is less than ten, the pivot is guaranteed to be good.

In the randomized algorithm we simply rechoose the pivot until it is good. The pseudocode for the algorithm is represented in Algorithm 26.

Algorithm 26: Quicksort(A)

```
$n \leftarrow |A|$
if $n \leq 1$ then
    /* List already sorted                                   */
    return
else
    Randomly choose a pivot $i_p \in \{0, \dots, n-1\}$.
    $p \leftarrow A[i_p]$
    $i \leftarrow$Partition$(A, p)$/* $i$ is the new location of the pivot   */
    while $i < n/10$ or $i > n/10$ do
        /* If the pivot is bad, rechoose it until it's good.  */
        Rechoose the pivot $i_p \in \{0, \dots, n-1\}$.
        $p \leftarrow A[i_p]$
        $i \leftarrow$Partition$(A, p)$
    Quicksort($A[0, \dots, i-1]$)
    Quicksort($A[i+1, \dots, n-1]$)
```

To figure out whether a pivot is good or not, we must partition the array. This means that rechoosing the pivot takes $O(n)$ time. In the runtime analysis, we see that this sacrifice ends up being beneficial as it guarantees logarithmic depth of the recursion tree.

[Maybe a picture of the worst case recursion tree in the new algorithm]

Lemma 3.4. The expected runtime of Algorithm 26 is $O(n \log n)$.

Proof. The recursion tree for the algorithm is otherwise the same as earlier, but now

- The largest of the two arrays after the partition has size at most $\frac{9n}{10}$.
- We might need to choose the pivot multiple times, which means that partition needs to be done repeatedly.

Let's first analyze the depth of the recursion tree. In the worst case, in each iteration the partition is as uneven as possible, that is each iteration one of the arrays is $9/10$th of the previous array. Hence $(\frac{10}{9})^d = n \implies d = O(\log n)$, where the O-notation hides the constant created by the base-change of the logarithm.

Partition still takes $O(n)$ time for each layer of the recursion tree. Let's inspect the expected number of partitions we need. Let $R(n)$ be the random variable representing the time spent on choosing the pivot and partitioning on one layer of the recursion tree. Let $X =$"number of times a pivot chosen" and notice that $\mathrm{E}[R(n)] = \mathrm{E}[X] \cdot O(n)$. We have P("pivot is good") $\geq 4/5$ because at least 80% of the elements are good choices for the pivot. For $X = i$ we need to first choose $i - 1$ bad pivots, and hence

$$\mathrm{P}(X = i) \leq \frac{1}{5^{i-1}} \cdot \frac{4}{5} = \frac{4}{5^i}.$$

As a rough approximation, we have

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \mathrm{P}(X = i) \cdot i \leq 4 \cdot \sum_{i=1}^{\infty} \frac{i}{5^i} \leq 4 \cdot \frac{1}{2} = O(1).$$

Hence $\mathrm{E}[R(n)] = O(n)$.

Putting all this together gives the expected runtime of $O(n \log n)$. □

Lemma 3.5. Algorithm 26 runs in $O(n \log n)$ time with high probability.

The proof of the high-probability runtime for Quicksort is relatively complex, and hence it will not be proven on the course. For educational purposes, we prove in the tutorial session that Algorithm 26 has $O(n \log^2 n)$ runtime with high probability.