COMP2721: Algorithms and Data Structures II

spring 2025

Lecture Notes

Algorithms are at the heart of Computer Science, they feature in all its subdisciplines, such as Artificial Intelligence, Data Analytics, High Performance Computing, and any type of application. *Algorithmic Thinking* is perhaps the most fundamental skill that every computer scientist should have.

Imagine you are at work and you are given a computational problem, for which you are asked to design and implement an algorithm. How do you start? What approach can you take? How do you determine if an algorithm is 'good' or better than another one? In which data structure should you store the data involved?

This module will teach you basic tools and principles to answer these questions. You will learn algorithm design principles, such as Greedy algorithms, Dynamic Programming, and Divide and Conquer. You will train and learn tools to analyse the performance of algorithms. You will become familiar with different data structures, including data structures for graphs, Dictionaries and Hashing, and learn how to decide which to use for which purpose.

1 Graphs and their Representation

A graph G is a pair (V, E) consisting of a vertex set V and an edge set E. In this module we consider finite graphs only, *i.e.* V and E are finite sets. In case of simple undirected graphs the edges are two-element subsets of the vertex set. Instead of $e = \{u, v\}$ we also write $e = uv$, if $e \in E$ is the edge with endpoints $u, v \in V$. For simple directed graphs we have $E \subseteq V \times V$. Again we write $e = (u, v)$ as $e = uv$, or as $u \to v$. That is, for undirected graphs we have $uv = vu$ and for directed graphs $uv \neq vu$. In case of undirected multi-graphs we use an *incidence function* $i : E \to \binom{V}{2}$ that maps edges to the set of their endpoints. For directed graphs we use $h, t : E \to V$, where $h(e)$ is the *head* of e and $t(e)$ is its *tail*.

An algorithm that processes graphs reads a graph as string consisting of names of vertices and names of edges. For efficiency this linear representation is converted into an internal representation. We consider three of them: adjacency lists, adjacency matrices and incidence matrices.

We measure the size and the access time in $n = |V|$ (number of vertices) and $m = |E|$ (number of edges).

1.1 Adjacency List

For each vertex v in V we maintain a single-linked list of its neighbours. These lists are organised as array or single-linked list.

If there is a linear ordering on the vertices we can save some time and space (but no more than a factor of 2 in the worst case) by keeping the lists sorted, or by storing greater neighbours only.

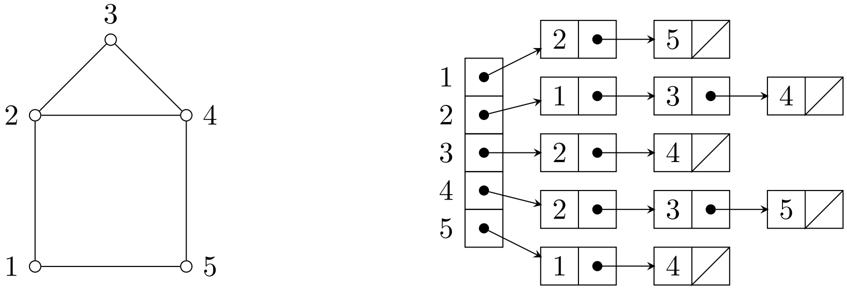


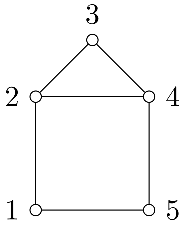
Figure 1: The house graph and its representation by adjacency lists.

Adjacency lists are the most space-efficient way to store a graph: The size of this representation is $\mathcal{O}(n+m)$. Unfortunately, $\mathcal{O}(n+m)$ the time for an adjacency check in the worst case.

Adjacency lists can be used for simple graphs only (no multiple edges are allowed, while self-loops would be possible). They can be used for directed graphs as well, where we either list out-neighbours or in-neighbours of each vertex. But we could as well create two list per vertex, one for the out-neighbours and one for the in-neighbours.

1.2 Adjacency Matrix

To represent the graph $G = (V, E)$ we create a $n \times n$ matrix A that is stored in a 2-dimensional array. If $V = \{1, 2, \dots, n\}$ then $A_{i,j} = 1$ if $ij \in E$ and $A_{i,j} = 0$ if $ij \notin E$.



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 0 & 0 & 1 & 0 \end{array}$$

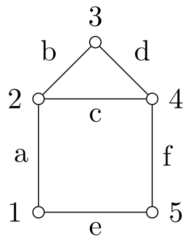
Figure 2: The house graph and its representation by the adjacency matrix.

An adjacency matrix has size $\mathcal{O}(n^2)$, and it takes $\mathcal{O}(n^2)$ time to create it. Therefore we cannot use this representation in algorithms that run in time $o(n^2)$. On the positive side, an adjacency matrix supports adjacency checks in constant time $\mathcal{O}(1)$.

Adjacency lists can be used for multi-graphs and directed graphs as well, where we store the multiplicity of each edge only.

1.3 Incidence Matrix

To represent the graph $G = (V, E)$ we create a $n \times m$ matrix B that is stored in a 2-dimensional array with entries $B_{v,e} = 1$ if $v \in V$ is and endpoint of $e \in E$ and $B_{v,e} = 0$ otherwise.



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Figure 3: The house graph and its representation by the incidence matrix.

An incidence matrix has size $\mathcal{O}(nm)$, and it takes $\mathcal{O}(nm)$ time to create it. Therefore we cannot use this representation in algorithms that run in time $o(nm)$. An incidence matrix supports adjacency checks in time $\mathcal{O}(m)$ and incidence checks in constant time $\mathcal{O}(1)$.

Incidence lists can also be used for multi-graphs and directed graphs (with entries 0, 1 and -1) because we have a column for each edge.

2 Graph Traversal

A traversal visits all vertices and edges of a graph and numbers the vertices in some order. Graph traversal forms the basis of many algorithms on graphs. In this section we discuss Depth-First Search, Breadth-First Search and Topological Sorting.

We encounter the typical steps of algorithm design:

1. Given a computional problem, come up with (or understand) an algorithm to solve it. The algorithm is typically given in pseudocode or in a suitable programming language.

2. Prove that the algorithm is correct.

3. Determine its running time.

Definition 1. An *(undirected) tree* is a connected acyclic undirected graph.

Definition 2. Let $H = (U, F)$ and $G = (U, V)$ be undirected graphs. The graph H is a *spanning subgraph* of G if H is a subgraph of G and $U = V$. A *spanning tree* is a spanning subgraph that is in fact a tree.

2.1 Breadth-First Search

Breadth-first search (or BFS for short) starts at the first vertex (or any specified vertex) of an undirected graph, and then continues to inspect its neighbours in first-in first-out (FIFO) manner. Neighbours of visited neighbours that have not been visited themselves are stored in a queue.

Lemma 3. BFS runs in linear time $\mathcal{O}(n + m)$.

Proof. Lines 10–13 run in constant time. Therefore the loop starting in line 9 requires time $\mathcal{O}(d(v))$. Since each vertex is visited only once, and lines 5, 6 and 8 run in constant time, the loop starting in line 4 requires time $\mathcal{O}(\sum_{v \in V}(1 + d(v))) = \mathcal{O}(n + m)$. Lines 2 and 3 run in time $\mathcal{O}(1)$ and $\mathcal{O}(n)$, respectively. Hence the overall running time is $\mathcal{O}(n + m)$. □

```
    input  : a graph $G = (V, E)$
    output: a BFS-forest $T = (V, F)$ of $G$ and a BFS-numbering $\sigma : V \to \mathbb{N}$
 1  begin
 2      $i \leftarrow 1$; $F \leftarrow \varnothing$; $Q \leftarrow$ emptyQueue;
 3      for $v \in V$ do mark $v$ unvisited;
 4      while there is a unvisited vertex $v \in V$ do
 5          $\sigma(v) \leftarrow i$; $i \leftarrow i + 1$;
 6          $Q \leftarrow$ enqueue$(v, Q)$; mark $v$ visited;
 7          while $Q$ is not empty do
 8              $v \leftarrow$ front$(Q)$; $Q \leftarrow$ dequeue$(Q)$;            /* remove $v$ from $Q$ */
 9              for $w \in N(v)$ do
10                  if $w$ is unvisited then
11                      $F \leftarrow F \cup \{vw\}$;
12                      $\sigma(w) \leftarrow i$; $i \leftarrow i + 1$;
13                      $Q \leftarrow$ enqueue$(w, Q)$; mark $w$ visited
```

Algorithm 1: Breadth-First Search

Lemma 4. If $G = (V, E)$ is connected then (V, F) is a spanning tree of G.

Proof.

- The outer **while**-loop (line 4) is executed only once.
- For the set U of visited vertices (U, F) is always connected.
- For the set U of visited vertices (U, F) is always acyclic.

□

Corollary. For every graph $G = (V, E)$ the graph (V, F) is a spanning forest of G having the same number of connected components as G.

2.2 Depth-First Search

Depth-first search (or DFS for short) starts at the first vertex (or any specified vertex) of an undirected graph, and then continues to inspect its neighbours in first-in last-out (FILO) manner. Neighbours of visited neighbours that have not been visited themselves are stored in a stack.

Lemma 5. DFS runs in linear time $\mathcal{O}(n + m)$.

Proof. The proof is similar to the one for BFS. Lines 9, 10 and 11 run in constant time as well as lines 6 and 7. Therefore the **for**-loop starting in line 8 runs in time $\mathcal{O}(d(v))$, and `DFS-visit` in time $\mathcal{O}(1 + d(v))$. Since every vertex is visited only once, the entire **while**-loop in line 4 requires time $\mathcal{O}(\sum_{v \in V}(1 + d(v))) = \mathcal{O}(n + m)$. Lines 2 and 3 run in time $\mathcal{O}(1)$ and $\mathcal{O}(n)$, respectively. Hence the overall running time is $\mathcal{O}(n + m)$. □

Lemma 6. If $G = (V, E)$ is connected then (V, F) is a spanning tree of G.

Proof.

- `DFS-visit` is called only once from line 4.

```
input  : a graph $G = (V, E)$
output: a DFS-forest $T = (V, F)$ of $G$ and a DFS-numbering $\sigma : V \to \mathbb{N}$
begin
    $i \leftarrow 1$; $F \leftarrow \varnothing$;
    for $v \in V$ do mark $v$ unvisited;
    while there is a unvisited vertex $v \in V$ do DFS-visit($v$);
procedure DFS-visit($v$)
    mark $v$ visited;
    $\sigma(v) \leftarrow i$; $i \leftarrow i + 1$;
    for $w \in N(v)$ do
        if $w$ is unvisited then
            $F \leftarrow F \cup \{vw\}$;
            DFS-visit($w$)
```

Algorithm 2: Depth-First Search

- For the set U of visited vertices (U, F) is always connected.
- For the set U of visited vertices (U, F) is always acyclic.

□

Corollary. For every graph $G = (V, E)$ the graph (V, F) is a spanning forest of G having the same number of connected components as G.

2.3 Topological Sort

A *topological sort* on a directed acyclic graph (or *dag* for short) $G = (V, A)$ is a numbering $\sigma : V \to \{1, 2, \dots, n\}$ of its vertices ($n = |V|$) such that, for all arcs $u \to v$ in A, $\sigma(u) < \sigma(v)$ holds. A digraph that contains a directed cycle does not admit a topological sort. Indeed, it can be proven that a directed graph admits a topological sort if, and only if, it is acyclic. We use a variant of DFS to compute a topological sort.

We consider the Algorithm 3, which uses three different labels, namely unvisited, stacked and visited, to mark the status of a vertex. For each vertex v that is marked stacked the procedure `TS-visit`(v) was called, but not terminated. When `TS-visit`(w) is called within `TS-visit`(v) then $vw \in A$. Therefore the **stop** in line 6 is executed if and only if the input graph G contains a directed cycle. That is, the algorithm can also be used to check whether a digraph is acyclic.

Lemma 7. Algorithm 3 computes a topological sort of its input graph.

Proof. Let vw be a directed edge of the digraph $G = (V, A)$. When `TS-visit`(v) is executed the vertex $w \in N^+(v)$ is considered in line 8. If w is already visited at the time then $\sigma(v) \leq i < \sigma(w)$ holds. Otherwise `TS-visit`(w) will terminate before `TS-visit`(v) terminates and therefore we have $\sigma(v) < \sigma(w)$. □

Lemma 8. Topological sort runs in linear time.

Proof. As for DFS. □

```
input  : a dag $G = (V, A)$
output: a topological sort $\sigma : V \to \mathbb{N}$ of $G$

begin
    $i \leftarrow |V|$;
    for $v \in V$ do mark $v$ unvisited;
    while there is a unvisited vertex $v \in V$ do TS-visit($v$);

procedure TS-visit ($v$)
    if $v$ is marked stacked then stop;
    if $v$ is marked unvisited then
        mark $v$ stacked;
        for $w \in N^+(v)$ do TS-visit($w$);
        mark $v$ visited;
        $\sigma(v) \leftarrow i$; $i \leftarrow i - 1$
```

Algorithm 3: Topological Sort

3 Greedy Algorithms

Greedy algorithms are algorithms that try to find an optimal solution to a problem by making the locally optimal choice at each step. In this section we discuss three important greedy algorithms, namely PRIM's algorithm, KRUSKAL's algorithm and DIJKSTRA's algorithm.

First we recall the definition and some properties of trees.

Lemma 9. Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. For $n > 0$ the following conditions are equivalent.

- G is connected and cycle-free
- G is connected and $n = m + 1$
- G is cycle-free and $n = m + 1$
- for every pair of vertices $u, v \in V$ there is exactly one path $(u, \ldots, v)$ in G

Definition 10. A *tree* is a connected acyclic graph.

Definition 11. A graph $H = (V, F)$ is a *spanning subgraph* of $G = (V, E)$ if $F \subseteq E$. A *spanning tree* is a spanning subgraph that is in fact a tree.

Definition 12. Let $G = (V, E)$ be a (connected) graph with edge-weights $\ell : E \to \mathbb{N}$. A spanning tree $T = (V, F)$ of G is called *minimum spanning tree* of G if $\sum_{e \in F} \ell(e) \leqslant \sum_{e \in F'} \ell(e)$ holds for all spanning trees (V, F') of G.

3.1 Minimum Spanning Trees: Prim's Algorithm

Variants of this algorithm were found by JARNÍK [Jar], PRIM [Prim] and DIJKSTRA [Dij]. An earlier algorithm was published by BORŮVKA, see [Bor].

In Algorithm 4 the set $N(U) = \bigcup_{u \in U} N(u) \setminus U$ contains the vertices outside U that have a neighbour in U. PRIM's algorithm uses the following variables:

```
   input  : a connected graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$
   output: a minimum spanning tree $T = (V, F)$ of $G$
1  begin
2      for $v \in V$ do $\pi(v) \leftarrow \infty$;
3      choose an arbitrary vertex $u \in V$;
4      $U \leftarrow \{u\}$; $F \leftarrow \varnothing$;
5      while $N(U) \neq \varnothing$ do
6          for $v \in N(u) \setminus U$ do
7              if $\pi(v) > \ell(uv)$ then
8                  $\pi(v) \leftarrow \ell(uv)$; $p(v) \leftarrow u$
9          choose $v \in V \setminus U$ with minimal $\pi(v)$;
10         $U \leftarrow U \cup \{v\}$; $F \leftarrow F \cup \{p(v)v\}$; $u \leftarrow v$
```

Algorithm 4: Minimum Spanning Tree (Prim)

(U, F) is the spanning tree constructed so far.
$u \in U$ is last vertex added to U.
$\pi(v)$ is minimum distance from $v \in V \setminus U$ to a vertex in U known so far.
$p(v)$ is the neighbour of v in U realising $\pi(v)$.

and maintains the following invariant:

(U, F) is a minimum spanning tree of $G[U] = (U, E \cap \binom{U}{2})$.

To bound the running time we observe that the code in lines 3, 4, 7, 8 and 10 can be executed in constant time. Line 2 runs in time $\mathcal{O}(n)$. The loops starting in lines 5 and 6 are executed n times and $d(u)$ times, respectively. We keep the values of π in a min-heap and find a minimum in line 9 in time $\mathcal{O}(\log n)$ and can update $\pi(v)$ in line 7 also in time $\mathcal{O}(\log n)$. Since $\sum_{v \in V} d(v) = 2m$, the overall running time of Prim's algorithm is $\mathcal{O}(m + n \log n)$.

Theorem 13. Prim's algorithm computes a minimum spanning tree of a connected graph in time $\mathcal{O}(m + n \log n)$.

Proof. Given a connected graph $G = (V, E)$ with edge-weights $\ell : E \to \mathbb{N}$. We show that (V, F) is a spanning tree of G by induction on $|U|$. When line 3 is executed $(\{u\}, \varnothing)$ is a tree. Each time line 10 is executed we add a leaf to that tree. Therefore (U, F) is a tree when the **while**-loop terminates with $N(U) = \varnothing$ because $U = V$. Hence (V, F) is a spanning tree of G.

To prove the minimality we consider another spanning tree $R = (V, A)$ of G. We will show $\sum_{f \in F} \ell(f) \leq \sum_{e \in A} \ell(e)$.

This is obvious for $F = A$. Otherwise let uv be the first edge in $F \setminus A$ that the algorithm adds to F, and let U be the set at that time, *i.e.* $u \in U$ and $v \in V \setminus U$.

The graph $(V, A \cup \{uv\})$ contains exactly one cycle, and this cycle uses another edge $xy \neq uv$ with $x \in U$ and $y \in V \setminus U$. We have $\ell(uv) \leq \ell(xy)$ by line 9 of our algorithm. That is, for $A' = \{uv\} \cup A \setminus \{xy\}$, the graph $R' = (V, A')$ is also a spanning tree of G with $\sum_{e \in A'} \ell(e) \leq \sum_{e \in A} \ell(e)$.

We apply the same argument to $T = (V, F)$ and $R' = (V, A')$ to obtain a spanning tree $R'' = (V, A'')$ of G with $\sum_{e \in A''} \ell(e) \leq \sum_{e \in A'} \ell(e)$. We iterate this process until it terminates

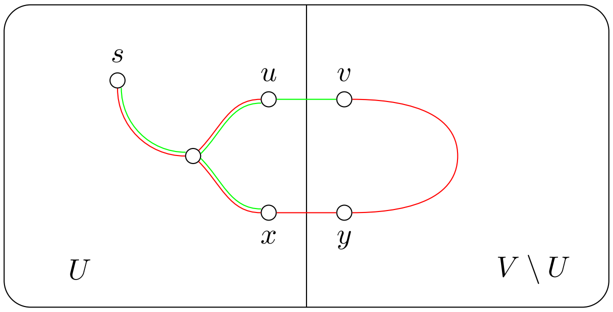


Figure 4: The exchange argument in the proof of Theorem 13.
Curves indicate paths of zero or more edges, straight lines are single edges,
F (just after uv was added) is given in green, and A (all edges of R) in red.

with a set $A'^{\cdots\prime} = F$. This will happen because G is a finite graph. We have

$$\sum_{f \in F} \ell(f) \leq \sum_{e \in A'^{\cdots\prime}} \ell(e) \leq \cdots \leq \sum_{e \in A'} \ell(e) \leq \sum_{e \in A} \ell(e)\,.$$

Hence (V, F) is a minimum spanning tree of G. □

3.2 Minimum Spanning Trees: Kruskal's Algorithm

Kruskal's algorithm [Kru] is a greedy algorithm as Prim's and Dijkstra's, because it adds the cheapest edge possible to F in each step.

```
  input  : a connected graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$
  output: a minimum spanning tree $T = (V, F)$ of $G$
1 begin
2     sort the edges $e \in E$ by their weights $\ell(e)$;
3     $F \leftarrow \varnothing$;
4     for $e \in E$ in non-decreasing order do
5         if $(V, F \cup \{e\})$ is acyclic then $F \leftarrow F \cup \{e\}$;
```

Algorithm 5: Minimum Spanning Tree (Kruskal)

Kruskal's algorithm uses the following variables:

(V, F)	is the spanning forest constructed so far.
$e \in E$	is the candidate edge to added to F.

and maintains the following invariant:

For every component $T' = (U, F')$ of (V, F), T' is a minimum spanning tree of $G[U]$.

The interesting part is the boolean condition in line 5. We can BFS or DFS on $(V, F \cup \{e\})$ to determine a cycle in that graph. However, this would lead to a running time of $\mathcal{O}(m^2)$. Faster implementations maintain a set of disjoint subsets of V that corresponds to the connected components of (V, F). The corresponding data structure should support tow operations:

find for a vertex v the subset of V containing it, because $(V, F \cup \{e\})$ is acyclic if and only if the two endpoints of e belong to different connected components of (V, F), and

union merge two disjoint sets into a new set, because these two components become one when we add e to F.

Theorem 14. KRUSKAL's algorithm computes a minimum spanning tree in time $\mathcal{O}(m \log n)$.

Proof. The time $\mathcal{O}(m \log n)$ can only be achieved by a clever union-find data structure. Anyway, line 2 requires time $O(m \log m)$ for sorting m edges. Since $m = \mathcal{O}(n^2)$ we have $\log m = \mathcal{O}(\log n)$ and therefore a bound of $\mathcal{O}(m \log n)$ for line 2. Line 3 can be done in constant time. The **for**-loop starting in line 4 is executed m times. To meet the overall bound of $\mathcal{O}(m \log n)$ we have time $\mathcal{O}(\log m)$ both for the find-operation and the union-operation, which can be achieved, see [CLRS].

KRUSKAL's algorithm computes a maximal acyclic spanning subgraph T of G. Since G is connected T is a spanning tree of G. It remains to be shown that T is a minimal spanning tree.

Let $R = (V, A)$ be any spanning tree of G. We show $\sum_{f \in F} \ell(f) \leq \sum_{e \in A} \ell(e)$. This is obvious for $A = F$. For $A \neq F$ let f be an edge in $F \setminus A$ with minimum weight. That is, all edges $e \in F$ with $\ell(e) < \ell(f)$ belong to A too and have been added to F before f. In the non-decresing order of the algorithm f is the first edge where A and F differ.

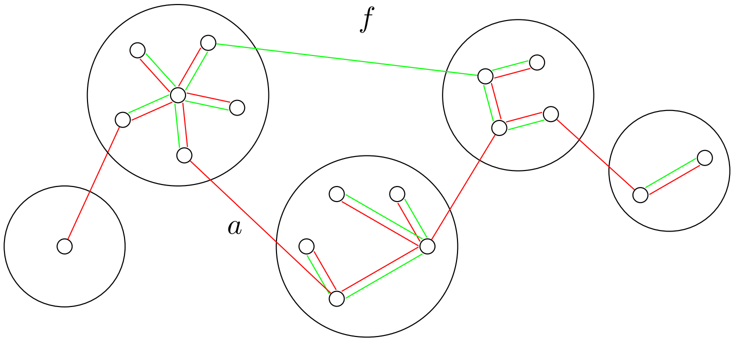


Figure 5: The exchange argument in KRUSKAL's algorithm.
The edges in F (just after f was added) are given in green, and A (all edges of R) in red.

We consider the graph $(V, A \cup \{f\})$. It has exactly one cycle, and because T is a tree this cycle contains an edge $a \in A \setminus F$. We fix any such edge a and have $\ell(f) \leq \ell(a)$.

The graph $R' = (V, A')$ with $A' = \{f\} \cup A \setminus \{a\}$ is also a spanning tree of G, and we have $\sum_{a \in A'} \ell(a) \leq \sum_{e \in A} \ell(e)$ because $\ell(f) \leq \ell(a)$.

We apply the same argument to T and R' to obtain $R'' = (V, A'')$ with $\sum_{e \in A''} \ell(e) \leq \sum_{e \in A'} \ell(e)$. We iterate this process until it terminates with a set $A'^{\cdots\prime} = F$. This will happen because G is a finite graph. We have

$$\sum_{f \in F} \ell(f) \leq \sum_{e \in A'^{\cdots\prime}} \ell(e) \leq \cdots \leq \sum_{e \in A'} \ell(e) \leq \sum_{e \in A} \ell(e) \,.$$

Hence $T = (V, F)$ is a minimum spanning tree of G. □

3.3 Single-Source Shortest Path: Dijkstra's Algorithm

Variants of the shortest path problem in undirected graphs:

	1	$+\triangleleft$	$+\ntriangleleft$	$-\oplus$	$-\ominus$
single pair shortest path					NP-c
single source shortest paths	BFS		Dijkstra		
all pairs shortest paths				F-W	

1	all edges have the same length
$+\triangleleft$	all edges have positive lengths fulfilling the triangle inequality
$+\ntriangleleft$	all edges have positive lengths
$-\oplus$	edges but no cycles of negative length may exist
$-\ominus$	cycles of negative length may exist

While Dijkstra's original algorihm found the shortest path between two nodes of a graph, today's variant of the algorithm fixes a single node as the 'source' node and finds shortest paths from the source to all other nodes in the graph, producing a shortest path tree.

Algorithm 6 was rediscovered many times, for example by Dijkstra [Dij].

```
input  : a graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$ and a source $s \in V$
output: the distances $d(s, v)$ for all $v \in V$ and
         a shortest path tree $T = (V, F)$ of $G$ with source $s$

1  begin
2      for $v \in V$ do $\pi(v) \leftarrow \infty$;
3      $d(s, s) \leftarrow 0$; $\pi(s) \leftarrow 0$; $U \leftarrow \{s\}$; $u \leftarrow s$; $F \leftarrow \varnothing$;
4      while $N(U) \neq \varnothing$ do
5          for $v \in N(u) \setminus U$ do
6              if $\pi(v) > \pi(u) + \ell(uv)$ then
7                  $\pi(v) \leftarrow \pi(u) + \ell(uv)$; $p(v) \leftarrow u$
8          choose $v \in V \setminus U$ with minimal $\pi(v)$;
9          $d(s, v) \leftarrow \pi(v)$;
10         $U \leftarrow U \cup \{v\}$; $F \leftarrow F \cup \{p(v)v\}$; $u \leftarrow v$
```

Algorithm 6: Single-Source Shortest Path (Dijkstra)

In Algorithm 6 the set $N(U) = \bigcup_{u \in U} N(u) \setminus U$ contains the vertices outside U that have a neighbour in U. Dijkstra's algorithm uses the following variables:

(U, F)	is the shortest path tree constructed so far.
$u \in U$	is last vertex added to U.
$\pi(v)$	is minimum distance from v to s known so far.
$p(v)$	is the neighbour of v in U on this path to s.

and maintains the following invariant:

(U, F) is a shortest path tree of $G[U] = (U, E \cap \binom{U}{2})$ and $d(s, v) \leq \pi(v)$ for all $v \in V$.

To bound the running time we observe that the code in lines 3, 8 and 10 can be executed in constant time. Line 2 runs in time $\mathcal{O}(n)$. The loops starting in lines 4 and 5 are executed n times and $d(u)$ times, respectively. We keep the values of π in a Fibonacci-heap and find a minimum in line 9 in time $\mathcal{O}(\log n)$ and can update $\pi(v)$ in line 7 also in time $\mathcal{O}(\log n)$. Since $\sum_{v \in V} d(v) = 2m$ the overall running time of DIJKSTRA's algorithm is $\mathcal{O}(m + n \log n)$.

Theorem 15. DIJKSTRA's algorithm solves the single-source shortest path problem in time $\mathcal{O}(m + n \log n)$.

Proof. For the run-time analysis see the above arguments, which are similar to the analysis of PRIM's algorithm. For disconnected graphs G we have $d(s,v) = \infty$ for all vertices v that do not belong to the connected component of s. In what follows we concentrate on the correctness of DIJKSTRA's algorithm on connected graphs.

Let $G = (V, E)$ be a connected graph with edge-weights $\ell : E \to \mathbb{N}$. First we show, by induction on $|U|$, that $T = (V, F)$ is a tree and that $d(s,z) = d_T(s,z)$ holds for all $z \in V$.

When line 3 is executed $(\{s\}, \varnothing)$ is a tree and $d(s,s)$ is the distance from s to itself in this tree. Each time line 10 is executed we add a leaf to that tree. The distances $d(s,w)$ and $d_T(s,w)$ do not change for the vertices $w \in U$. For the new vertex v we have $d(s,v) = d_T(s,v)$ by construction. Therefore (U, F) is a tree when the **while**-loop terminates with $N(U) = \varnothing$ because $U = V$. Hence $T = (V, F)$ is a spanning tree of G and we have $d(s,z) = d_T(s,z)$ for all $z \in V$.

It remains to show $d(s,z) = d_G(s,z)$ for all $z \in V$. We consider a shortest path tree $R = (V, A)$ such that $d_G(s,z) = d_R(s,z)$ holds for all $z \in V$. Such a tree exists because each initial part $(s, \ldots, y)$ of a shortest path $(s, \ldots, z)$ in G is a shortest path from s to y in G.

We are done if $T = R$. Otherwise let uv be the first edge in $F \setminus A$ that the algorithm adds to F, and let U be the set at that time, *i.e.* $u \in U$.

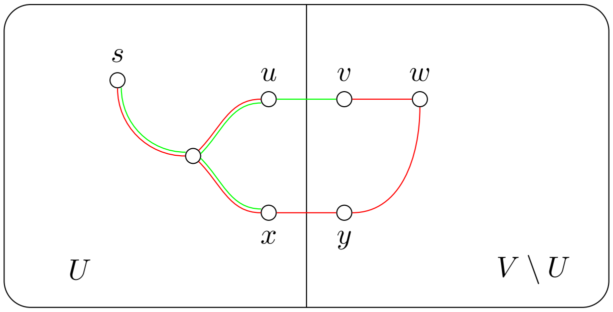


Figure 6: The exchange argument in the proof of Theorem 15.
Curves indicate paths of zero or more edges, straight lines are single edges,
F (just after uv was added) is given in green, and A (all edges of R) in red.

The graph $(V, A \cup \{uv\})$ contains exactly one cycle, and this cycle uses another edge $xy \neq uv$ with $x \in U$ and $y \in V \setminus U$. The way we choose v in line 8 ensures $\pi(v) \leq \pi(y)$ at this stage, and therefore $\pi(v) \leq d(s,z)$ for all $z \in V \setminus U$.

Note that $u = x$ or $v = y$ is possible, but not both. The graph $(V, A \cup \{uv\})$ contains exactly one cycle, which splits into a path $P = (u, \ldots, x)$ in $G[U]$ and a path $Q = (v, \ldots, y)$ in $G - U$. Let w be the second vertex on Q, or $w = x$ if $v = y$. Anyway, we have $uv \in F \setminus A$.

That is, for $A' = \{uv\} \cup A \setminus \{vw\}$, the graph $R' = (V, A')$ is also a spanning tree of G. We have $d_{R'}(s,v) \leq d_R(s,v)$ since $\pi(v) \leq \pi(y)$ and all edges on the path Q have non-negative weights. For all other vertices $z \neq v$ we have $d_{R'}(s,z) = d_R(s,z)$. Hence $d_{R'}(s,z) \leq d_R(s,z)$ holds for all $z \in V$.

We apply the same argument to T and R' to obtain a spanning tree R'' of G with $d_{R''}(s,z) \leq d_{R'}(s,z)$ for all $z \in V$. We iterate this process until it terminates with a tree $R'^{\cdots\prime} = T$. This will happen because G is a finite graph. For all $z \in V$ we have

$$d_T(s,z) = d_{R'^{\cdots\prime}}(s,z) \leq \cdots \leq d_{R'}(s,z) \leq d_R(s,z) = d(s,z)\,.$$

Hence T is a shortest path tree of G, *i.e.* $d_T(s,z) = d_G(s,z)$ holds for all $z \in V$. □

3.4 Theoretical Foundations for Greedy Algorithms

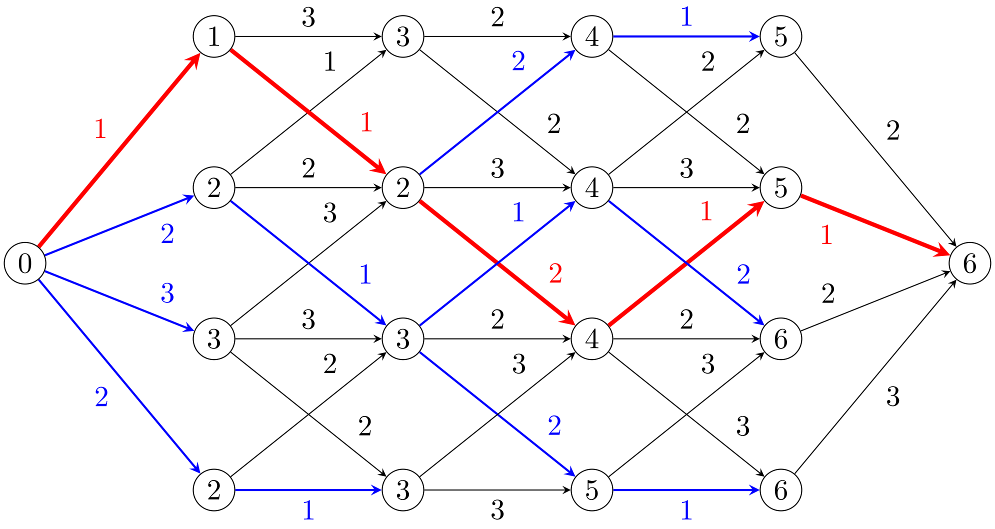
There is a mathematical theory of greedy algorithms, see Section 16.4 on matroids in [CLRS].

4 Dynamic Programming

Some computational problems can be solved by splitting them up into smaller, simpler problems, called *sub-problems*, in such a way that combining the solutions for the sub-problems allows solving the original problem. This can be done recursively or iteratively. Algorithms that proceed in this way are called *dynamic programming algorithms*. Dynamic programming is used widely for designing efficient algorithms both in practical applications and in research. All dynamic programming algorithms are based on the fact that every initial part of an optimal solution is optimal for the corresponding sub-problem. In general not all optimal sub-solutions extend to a globally optimal solution.

In this section we will see how dynamic programming can be used to improve the running time of algorithms, and we will discuss some prominent examples of dynamic programming algorithms.

4.1 Shortest Path in Layered Digraphs



4.2 Matrix Chain-Product

4.2.1 The Problem

We wish to compute the product

$$A = A_1 \cdot A_2 \cdot A_3 \cdot \cdots \cdot A_{n-1} \cdot A_n$$

where A_i is a $d_{i-1} \times d_i$ matrix. Because matrix multiplication is associative we can clever parenthesise the product to minimise the number of scalar multiplications.

Example. Let $n = 3$, $d_0 = 5$, $d_1 = 50$, $d_2 = 20$ and $d_3 = 10$.

$(A_1 \cdot A_2) \cdot A_3$. We first compute the 5×20 matrix $A' = A_1 \cdot A_2$. This requires $5 \cdot 50 \cdot 20 = 5000$ scalar multiplications. Next we compute the 5×10 matrix $A = A' \cdot A_3$. This requires $5 \cdot 20 \cdot 10 = 1000$ scalar multiplications, 6000 in total.

$A_1 \cdot (A_2 \cdot A_3)$. This time we start computing the 50×10 matrix $A'' = A_2 \cdot A_3$. This requires $50 \cdot 20 \cdot 10 = 10000$ scalar multiplications. Now we obtain the 5×10 matrix A as $A_1 \cdot A''$. This requires $5 \cdot 50 \cdot 10 = 2500$ scalar multiplications, 12500 in total.

4.3 Recurrence Equation

For $1 \leqslant i \leqslant k \leqslant n$ let $N[i, k]$ denote minimum number of scalar multiplications to compute $A_i \cdot \cdots \cdot A_k$. We observe

$$\begin{array}{ll} N[i,i] = 0 & \text{for all } i \text{ with } 1 \leqslant i \leqslant n \\ N[i,k] = \min\{N[i,j] + d_{i-1}d_jd_k + N[j+1,k] \mid i \leqslant j < k\} & \text{for all } i \text{ and } k \text{ with } 1 \leqslant i < k \leqslant n \end{array}$$

If the minimum in the expression for $N[i, k]$ is realised for j then the last matrix multiplication in the product $A_i \cdots A_k$ should be $(A_i \cdots A_j) \cdot (A_{j+1} \cdots A_k)$.

4.3.1 Pseudo Code

```
input : an integer $n > 0$ (number of matrices $A_i$ to multiply) and an array $d[0..n]$
        (sizes, $A_i$ is a $d[i-1] \times d[i]$ matrix)
output: $N[1, n]$ the minimum number of scalar multiplications
begin
    for $i \leftarrow 1$ to $n$ do $N[i, i] \leftarrow 0$;
    for $b \leftarrow 1$ to $n - 1$ do
        for $i \leftarrow 1$ to $n - b$ do
            $k \leftarrow i + b$; $m \leftarrow d[i-1] * d[k]$; $N[i, k] \leftarrow \infty$;
            for $j \leftarrow i$ to $k - 1$ do
                $N[i, k] \leftarrow \min(N[i, k], N[i, j] + m * d[j] + N[j+1, k])$
```

Algorithm 7: Matrix Chain-Product

4.3.2 Correctness and Running Time

The correctness is obvious.

We compute the values $N[i,i]$ in time $\mathcal{O}(n)$. The three nested for-loops contribute a factor n each. The arithmetic operations and assignments require constant time. Therefore the algorithm runs in $\mathcal{O}(n^3)$.

4.4 All pair shortest paths

To compute the distance between every pair of vertices in a edge-weighted graph we can run a SSSP algorithm for every start vertex. However, there are better ways. We can compute the powers A^2, A^4, A^8 of the adjacency matrix, where the entry for u and v is 0 if $u = v$, $\ell(u,v)$ if u and v are adjacent, and ∞ otherwise. Instead of the usual $=$ and $*$ we use max and $+$. For a graph with n vertices we need $\log n$ matrix multiplications.

Here we consider the approach by Floyd [Flo] and Warshall [War] that leads to a dynamic programming algorithm. This algorithm also works for directed graphs, and only for directed graphs in case of edges with negative weight. The algorithm fails on graphs containg a cycle of negative weight.

4.4.1 Recurrence Equation

We fix a linear order on the vertices of the input graph $G = (V,E)$, wlog $V = \{1,2,3,\ldots,n\}$ and compute values $d_k(u,v)$ which are the distances between verticres u and v in the subgraph of G induced by u, v and the vertices $1,2,\ldots,k$. The distances in G are d_n.

$$\begin{aligned}
&d_0(v,v) = 0 && \text{for all } v \in V\\
&d_0(u,v) = \ell(u,v) && \text{for all } uv \in E\\
&d_0(u,v) = \infty && \text{for all } uv \notin E, u \neq v\\
&d_k(u,v) = \min(d_{k-1}(u,v), d_{k-1}(u,k) + d_{k-1}(k,v)) && \text{for all } u,k,v \in V
\end{aligned}$$

4.4.2 Pseudo Code

4.4.3 Correctness

Since we allow negative lengths, we have to ensure that the concatenation of the paths $(u,\ldots,k)$ and $(k,\ldots,v)$ is a path $(u,\ldots,v)$. We apply induction on k. For $k = 0$ all our shortest paths have at most one edge. For $k > 0$ assume, for the sake of a contradiction that $(u,\ldots,k,\ldots,v)$ contains a vertex w twice. By induction hypothesis $(u,\ldots,k)$ and $(k,\ldots,v)$ are paths, and both contain w. If the walk $(u,\ldots,w,\ldots,k,\ldots,w,\ldots,v)$ is shorter than the path $(u,\ldots,w,\ldots,v)$ then the cycle $(w,\ldots,k,\ldots,w)$ has negative length. Since such cycles do not exist in G the concatenation of $(u,\ldots,k)$ and $(k,\ldots,v)$ is a path $(u,\ldots,v)$.

4.4.4 Running Time

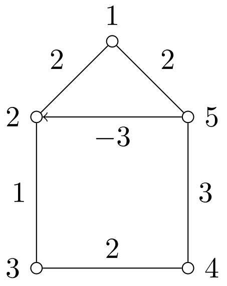
The initialisation for $k = 0$ takes time $\mathcal{O}(n^2)$, the rest $\mathcal{O}(n^3)$, which dominates the running time.

As stated, the algorithm uses $\mathcal{O}(n^3)$ space as well, but this can be reduced to $\mathcal{O}(n^2)$.

```
input  : a directed graph $G = (V, E)$ with weights $\ell : E \to \mathbb{Z}$ such that $G$ has no
         cycle of negative length, w.l.o.g. $V = \{1, 2, \dots, n\}$
output: the distances $d_n(u, v)$ for all $u, v \in V$
begin
    for $u \in V$ do
        for $v \in V$ do
            if $u = v$ then
                $d_0(u, v) \leftarrow 0$
            else
                if $uv \in E$ then
                    $d_0(u, v) \leftarrow \ell(u, v)$
                else
                    $d_0(u, v) \leftarrow \infty$
    for $k \leftarrow 1$ to $n$ do
        for $u \in V$ do
            for $v \in V$ do
                $d_k(u, v) \leftarrow \min(d_{k-1}(u, v), d_{k-1}(u, k) + d_{k-1}(k, v))$
```

Algorithm 8: All Pair Shortest Path (Floyd/Warshall)

4.4.5 Example



$$\begin{array}{c}
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0_0 & 2_0 & \infty & \infty & 2_0 \\
2 & 2_0 & 0_0 & 1_0 & \infty & \infty \\
3 & \infty & 1_0 & 0_0 & 2_0 & \infty \\
4 & \infty & \infty & 2_0 & 0_0 & 3_0 \\
5 & 2_0 & -3_0 & \infty & 3_0 & 0_0
\end{array} \\
d_0(u, v)
\end{array}
\quad
\begin{array}{c}
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0_0 & 2_0 & \infty & \infty & 2_0 \\
2 & 2_0 & 0_0 & 1_0 & \infty & 4_1 \\
3 & \infty & 1_0 & 0_0 & 2_0 & \infty \\
4 & \infty & \infty & 2_0 & 0_0 & 3_0 \\
5 & 2_0 & -3_0 & \infty & 3_0 & 0_0
\end{array} \\
d_1(u, v)
\end{array}
\quad
\begin{array}{c}
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0_0 & 2_0 & 3_2 & \infty & 2_0 \\
2 & 2_0 & 0_0 & 1_0 & \infty & 4_1 \\
3 & 3_2 & 1_0 & 0_0 & 2_0 & 5_2 \\
4 & \infty & \infty & 2_0 & 0_0 & 3_0 \\
5 & -1_2 & -3_0 & -2_2 & 3_0 & 0_0
\end{array} \\
d_2(u, v)
\end{array}$$

$$\underset{d_3(u,v)}{\begin{array}{cc} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0_0 & 2_0 & 3_2 & 5_3 & 2_0 \\ 2_0 & 0_0 & 1_0 & 3_3 & 4_1 \\ 3_2 & 1_0 & 0_0 & 2_0 & 5_2 \\ 5_3 & 3_3 & 2_0 & 0_0 & 3_0 \\ -1_2 & -3_0 & -2_2 & 0_3 & 0_0 \end{bmatrix} \end{array}} \quad \underset{d_4(u,v)}{\begin{array}{cc} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0_0 & 2_0 & 3_2 & 5_3 & 2_0 \\ 2_0 & 0_0 & 1_0 & 3_3 & 4_1 \\ 3_2 & 1_0 & 0_0 & 2_0 & 5_2 \\ 5_3 & 3_3 & 2_0 & 0_0 & 3_0 \\ -1_2 & -3_0 & -2_2 & 0_3 & 0_0 \end{bmatrix} \end{array}} \quad \underset{d_5(u,v)}{\begin{array}{cc} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0_0 & -1_5 & 0_5 & 2_5 & 2_0 \\ 2_0 & 0_0 & 1_0 & 3_3 & 4_1 \\ 3_2 & 1_0 & 0_0 & 2_0 & 5_2 \\ 2_5 & 0_5 & 1_5 & 0_0 & 3_0 \\ -1_2 & -3_0 & -2_2 & 0_3 & 0_0 \end{bmatrix} \end{array}}$$

4.5 CYK algorithm

The Cocke-Younger-Kasami (CYK) algorithm is a parsing algorithm for context-free grammars in Chomsky normal form. It was named after its inventors, John Cocke [Co], Daniel Younger [You] and Tadao Kasami [Kas]. Let $G = \langle T, V, P, S\rangle$ be a CFG in CNF, that is

- T is a finite set of *terminals*,
- V is a finite set of *variables*, $V \cap T = \varnothing$
- $P \subseteq V \times ((V \times V) \cup T)$ is the set of *productions*
- $S \in V$ is the *start symbol*

Since G is in CNF, the productions are of the form

$$A \to BC \quad \text{or} \quad A \to a \quad \text{or} \quad S \to \varepsilon$$

for $A, B, C \in V$ and $a \in T$, and where ε denotes the empty string. If the production $S \to \varepsilon$ belongs to the grammar then S cannot occur on the right hand side of a production. For simplicity we assume all productions are written as $A \to BC$ for $A, B, C \in V$ or $A \to a$ for $A \in V$ and $a \in T$.

For a fixed CFG in CNF, the membership problem is, given a string $s = x_1 x_2 \cdots x_n$ consisting of n terminals $x_i \in T$, is there a derivation $S \Rightarrow^* s$? Our goal is to solve the membership problem.

4.5.1 Recurrence equation

For indices i and k with $1 \le i \le k \le n$ we consider the set $V(i,k) \subseteq V$ defined by

$$V(i,k) = \{A \in V \mid A \Rightarrow^* x_i x_{i+1} \cdots x_k\}$$

We have

$$\begin{aligned} V(i,i) &= \{A \in V \mid (A \to x_i) \in P\} && \text{for } 1 \le i \le n \\ V(i,k) &= \{A \in V \mid \exists (A \to BC) \in P\ \exists j\ \ (i \le j < k \wedge B \in V(i,j) \wedge C \in V(j+1,k))\} \\ & && \text{for } 1 \le i < k \le n \end{aligned}$$

The string s can be derived if $S \in V(1,n)$.

4.5.2 Examples

We consider the CFG $\langle T, V, P, S\rangle$ with $T = \{a, b\}$, $V = \{S, A, B, C\}$ and the productions

$$S \to AB \mid BC \qquad A \to BA \mid a \qquad B \to CC \mid b \qquad C \to AB \mid a$$

For $s = baaba$ we compute the following values $V(i, k)$:

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$i=1$	$\{B\}$	$\{S,A\}$	$\varnothing$	$\varnothing$	$\{S,A,C\}$
$i=2$		$\{A,C\}$	$\{B\}$	$\{B\}$	$\{S,A,C\}$
$i=3$			$\{A,C\}$	$\{S,C\}$	$\{B\}$
$i=4$				$\{B\}$	$\{S,A\}$
$i=5$					$\{A,C\}$

For $s = bbabaa$ we compute the following values $V(i, k)$:

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
$i=1$	$\{B\}$	$\varnothing$	$\{A\}$	$\{S,C\}$	$\{B\}$	$\{S,A\}$
$i=2$		$\{B\}$	$\{S,A\}$	$\{S,C\}$	$\{B\}$	$\{S,A\}$
$i=3$			$\{A,C\}$	$\{S,C\}$	$\{B\}$	$\{S,A\}$
$i=4$				$\{B\}$	$\{S,A\}$	$\varnothing$
$i=5$					$\{A,C\}$	$\{B\}$
$i=6$						$\{A,C\}$

4.5.3 Pseudo code

```
fixed  : A CFG $G = \langle T, V, P, S\rangle$ in CNF is not part of the input.
input  : a string $x_1 x_2 \ldots x_n \in T^*$
output: a boolean value indicating whether $x_1 x_2 \ldots x_n$ can be generated by $G$
begin
    for $i \leftarrow 1$ to $n$ do
        $V(i, i) \leftarrow \{A \in V \mid (A \to x_i) \in P\}$
    for $b \leftarrow 1$ to $n - 1$ do
        for $i \leftarrow 1$ to $n - b$ do
            $k \leftarrow i + b$; $V(i, k) \leftarrow \varnothing$;
            for $j \leftarrow i$ to $k - 1$ do
                for $(A \to BC) \in P$ do
                    if $B \in V(i, j)$ and $C \in V(j + 1, k)$ then  $V(i, k) \leftarrow V(i, k) \cup \{A\}$ ;
    if $S \in V(1, n)$ then accept $x_1 x_2 \ldots x_n$ else reject $x_1 x_2 \ldots x_n$;
```

Algorithm 9: CYK parsing of context-free languages (Cocke/Younger/Kasami)

4.5.4 Correctness and Running Time

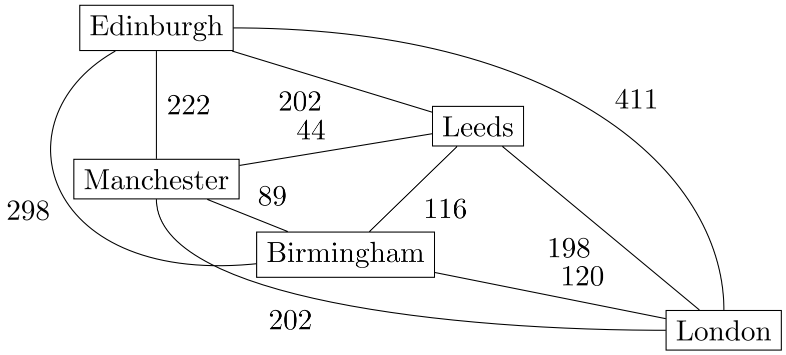
The correctness follows from the recurrence equation.

The CYK-algorithm runs in time $O(n^3)$ since the grammar is fixed. A factor linear in $|P|$ is hidden. For use in a compiler this is to slow; we want linear time. Therefore the grammars of programming languages are restricted context-free grammars.

4.6 Travelling Sales Person

The Travelling Sales Person Problem (TSP) is probably one of the most intensively studied algorithmic problems. Given a number of cities and the distances between any pair of cities, the task is to find a shortest round tour, that starts in one city, visits each other city once and brings the sales person back to the start city.

This problem can be modelled as a graph problem. We are given a complete undirected graph $G = (V, E)$ with edge weights $c\colon E \to \mathbb{N}$. (Recall that an undirected graph is *complete*, if every two distinct vertices are connected by an edge.) The task is to find a minimum length round tour from one vertex through all other vertices and back to the start vertex.



4.7 Recurrence Equation

Let $C = \{1, 2, \ldots, n\}$ be the set of cities to be visited. Without loss of generality the tour starts in city 1. For a set $S \subseteq C$ with $1 \in S$ and a city $c \in S \setminus \{1\}$ let $\ell(S, c)$ be the length of a shortest trip from 1 to c visiting all cities in S. We observe:

$$\begin{aligned} \ell(\{1,c\},c) &= d[1,c] && \text{for all } c \in C \setminus \{1\} \\ \ell(S,c) &= \min\{\ell(S \setminus \{c\}, a) + d[a,c] \mid a \in S \setminus \{1,c\}\} && \text{for all } S \text{ and } c \text{ with } \{1,c\} \subset S \subseteq C \end{aligned}$$

A minimum tour through all cities in $\{1, 2, \ldots, n\}$ has length $\min\{\ell(C, c) + d[c, 1] \mid c \in C \setminus \{1\}\}$.

4.7.1 Pseudo Code

The pseudo-code [Bel, HeKa] is Algorithm 10 on page 19.

4.7.2 Correctness and Running Time

The correctness is obvious. Since $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ the overall running time is $\mathcal{O}(2^n n^2)$.

```
input  : an integer $n > 2$ (number of cities) and a 2-dimensional array $d[1..n, 1..n]$
         of pairwise distances
output: a permutation $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$ minimising
         $d[\pi(n), \pi(1)] + \sum_{i=2}^{n} d[\pi(i-1), \pi(i)]$
begin
    for $c \in \{2, \dots, n\}$ do  $\ell(\{1, c\}, c) \leftarrow d[1, c]$; $p(\{1, c\}, c) \leftarrow 1$ ;
    for $k \leftarrow 2$ to $n - 1$ do
        for $S' \in \binom{\{2,\dots,n\}}{k}$ do
            for $c \in S'$ do
                $S \leftarrow S' \cup \{1\}$; $\ell(S, c) \leftarrow \infty$;
                for $a \in S' \setminus \{c\}$ do
                    $\ell' \leftarrow \ell(S \setminus \{c\}, a) + d[a, c]$;
                    if $\ell' < \ell(S, c)$ then  $\ell(S, c) \leftarrow \ell'$; $p(S, c) \leftarrow a$ ;
    $\ell'' \leftarrow \infty$;
    for $a \in \{2, \dots, n\}$ do
        $\ell' \leftarrow \ell(\{1, \dots, n\}, a) + d[a, 1]$;
        if $\ell' < \ell''$ then $\ell'' \leftarrow \ell'$; $\pi(n) \leftarrow a$;
    $S \leftarrow \{1, \dots, n\}$;
    for $i \leftarrow n$ downto 2 do
        $\pi(i-1) \leftarrow p(S, \pi(i))$; $S \leftarrow S \setminus \{\pi(i)\}$
```

Algorithm 10: Travelling Sales Person (Bellman / Held & Karp)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 5 & 2 & 2 & 5 \\ 2 & 5 & 0 & 2 & 4 & 6 \\ 3 & 2 & 2 & 0 & 1 & 4 \\ 4 & 2 & 4 & 1 & 0 & 2 \\ 5 & 5 & 6 & 4 & 2 & 0 \end{array}$$

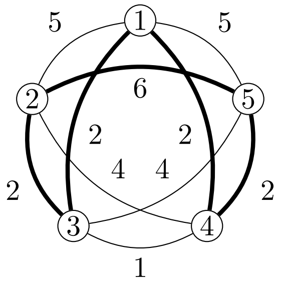


Figure 7: An instance of TSP. The unique optimal tour (1,3,2,5,4,1) does not use the edge $\{3, 4\}$, which has minimum weight, but uses the edge $\{2, 5\}$, which has maximum weight.

S	c	$\ell(S,c)$	$p(S,c)$
$\{1,2\}$	2	5	1
$\{1,4\}$	4	2	1
$\{1,2,3\}$	2	4	3
$\{1,2,4\}$	2	6	4
$\{1,2,5\}$	2	11	5
$\{1,3,4\}$	3	3	4
$\{1,3,5\}$	3	9	5
$\{1,4,5\}$	4	7	5
$\{1,2,3,4\}$	2	5	3
$\{1,2,3,4\}$	4	8	3
$\{1,2,3,5\}$	3	13	2
$\{1,2,4,5\}$	2	10	5
$\{1,2,4,5\}$	5	11	4
$\{1,3,4,5\}$	4	8	5
$\{1,2,3,4,5\}$	2	10	3
$\{1,2,3,4,5\}$	4	12	5
$\{1,3\}$	3	2	1
$\{1,5\}$	5	5	1
$\{1,2,3\}$	3	7	2
$\{1,2,4\}$	4	9	2
$\{1,2,5\}$	5	11	2
$\{1,3,4\}$	4	3	3
$\{1,3,5\}$	5	6	3
$\{1,4,5\}$	5	4	4
$\{1,2,3,4\}$	3	8	2
$\{1,2,3,5\}$	2	11	3
$\{1,2,3,5\}$	5	10	2
$\{1,2,4,5\}$	4	13	5
$\{1,3,4,5\}$	3	8	5
$\{1,3,4,5\}$	5	5	4
$\{1,2,3,4,5\}$	3	12	2
$\{1,2,3,4,5\}$	5	10	4

Table 1: values of $\ell(S,c)$ and $p(S,c)$ for the instance in Figure 7

4.8 Example

For the example shown in Figure 7, the values of $\ell(S,c)$ and $p(S,c)$ are given in Table 1.

We use the values for $S = \{1,2,3,4,5\}$ to compute the length of a tour for the four possibilities of the last city visited before returning to city 1. These are $a = 2 :\ 10+5$, $a = 3 :\ 12+2$, $a = 4 :\ 12+2$ and $a = 5 :\ 10+5$. The minimal length of a tour is therefore 14 for $a = 3$ or $a = 4$. Following the previous cities stored in p we obtain the tours $(1,4,5,2,3,1)$ and $(1,3,2,5,4,1)$, which is the reverse of the former.

5 Divide-and-Conquer

5.1 Introduction

The *divide-and-conquer* technique leads to recursive algorithms. This paradigm involves three steps at each level of the recursion:

Divide the current instance into a number of subinstances to the same problem, but smaller in size.

Conquer the subinstances by solving them recursively.
If an instance is small enough solve it directly.

Combine the solutions to the subinstances into a solution for the original instance.

Example. binary search, merge sort, quick sort

5.2 Recurrences

The running time of a divide-and-conquer algorithm can be bound by a function $T : \mathbb{N} \to \mathbb{N}$ given by a recurrence.

Example.

Fibonacci numbers

$$\begin{aligned} &F(0) = 0 \\ &F(1) = 1 \\ &F(n) = F(n-1) + F(n-2) && \text{for all } n > 1 \end{aligned}$$

Towers of Hanoi

$$\begin{aligned} &T(0) = 0 \\ &T(n) = 2 \cdot T(n-1) + 1 && \text{for all } n > 0 \end{aligned}$$

quick sort

$$\begin{aligned} &T(1) = 0 \\ &T(n) = n + T(n-1) && \text{for all } n > 0 \end{aligned}$$

Each recurrence consists of *initial conditions* $T(n) = c_n$ for small n and a *recursive equation* that describes $T(n)$ in terms of of n and $T(m)$ for some $m < n$.

5.2.1 Direct Method

We solve the quick-sort recurrence by substitutions.

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + T(n-3) \\ &= n + (n-1) + (n-2) + \cdots + 2 + T(1) \\ &= n + (n-1) + (n-2) + \cdots + 2 + 1 + T(0) \\ &= n + (n-1) + (n-2) + \cdots + 2 + 1 + 0 \\ &= \tfrac{1}{2}n(n+1) \\ &= \mathcal{O}(n^2) \end{aligned}$$

5.2.2 Guessing a Solution

We guess $F(n) = c^n$ is (for a suitable constant c) the solution of the Fibonacci recurrence.

$$\begin{aligned} c^n &= c^{n-1} + c^{n-2} \\ c^2 &= c + 1 \\ c^2 - c - 1 &= 0 \\ c_{1,2} &= \tfrac{1}{2} \pm \sqrt{\left(\tfrac{1}{2}\right)^2 + 1} \\ c_1 &= \tfrac{1}{2}(1 + \sqrt{5}) \\ c_2 &= \tfrac{1}{2}(1 - \sqrt{5}) \end{aligned}$$

We now guess $F(n) = a \cdot c_1^n + b \cdot c_2^n$ is the solution for suitable constants a and b and use the initial conditions to determine a and b.

$$\begin{aligned} 0 &= a + b && n = 0 \\ 1 &= \frac{a}{2}\big(1+\sqrt{5}\big) + \frac{b}{2}\big(1-\sqrt{5}\big) && n = 1 \end{aligned}$$

which implies

$$\begin{aligned} a &= +\tfrac{1}{5}\sqrt{5} \\ b &= -\tfrac{1}{5}\sqrt{5} \end{aligned}$$

Let $G(n) = a \cdot c_1^n + b \cdot c_2^n$. We already know $F(0) = G(0)$ and $F(1) = G(1)$. Let us try to prove $F(n) = G(n)$ for all $n \geqslant 2$. We use mathematical induction.

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) && \text{recursive equation} \\ &= G(n-1) + G(n-2) && \text{induction hypothesis} \\ &= ac_1^{n-1} + bc_2^{n-1} + ac_1^{n-2} + bc_2^{n-2} && \text{definition of } G \\ &= ac_1^{n-2}(c_1+1) + bc_2^{n-1}(c_2+1) && \\ &= ac_1^n + bc_2^n && c_1^2 = c_1 + 1;\ c_2^2 = c_2 + 1 \\ &= G(n) && \text{definition of } G \end{aligned}$$

We proved

$$F(n) = \frac{1}{5}\sqrt{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right],$$

which means $F(n) = \mathcal{O}(c_1^n)$.

"Guessing asolution" applies to recursive equations of the type $T(n) = f(n) + \sum_{i=1}^k a_i T(n-i)$, where $k > 0$ is an integer, and so are the coefficients $a_i \geqslant 0$ for $i = 1, 2, \ldots, k$. If $\sum_{i=1}^k a_i = 1$ then the direct method applies and leads to $T(n) = \mathcal{O}(n \cdot f(n))$ for monotonously non-decreasing f. The general approach for $\sum_{i=1}^k a_i > 1$ is:

- Consider a function T' that fulfills the recursive equation $T'(n) = \sum_{i=1}^k a_i T'(n-i)$.
- Guess $T'(n) = c^n$ and substitute this into the recursive equation for T'.
- Compute the largest root c_1 of the resulting polynomial equation, or approximate it, for instance by Newton's method.
- Prove $T(n) = \mathcal{O}(c_1^n \cdot f(n))$ by induction.

  In case you use an approximation of c_1 and $f(n)$ is bounded from above by a polynomial in n, note that $c_1^n \cdot f(n) = \mathcal{O}((c_1 + \varepsilon)^n)$ holds for all $\varepsilon > 0$. That is, you prove $T(n) = \mathcal{O}((c_1 + \varepsilon)^n)$ by induction.

5.2.3 The Master Method

The master method applies to recurrences of the form

$$T(n) = a \cdot T(\tfrac{n}{b}) + f(n)$$

where $a \geqslant 1$ and $b > 1$ are constants and $f(n) > 0$ for almost all n.

Theorem 16 (Master theorem)**.** Let a, b, f, and T as above. Then $T(n)$ can be bounded asymptotically as follows:

- If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \mathcal{O}(n^{\log_b a})$.
- If $f(n) = \mathcal{O}(n^{\log_b a})$, then $T(n) = \mathcal{O}(n^{\log_b a} \log n)$.
- If $n^{\log_b a + \epsilon} = \mathcal{O}(f(n))$ for some constant $\epsilon > 0$, and if $af(\frac{n}{b}) \leqslant cf(n)$ for some constant $c < 1$ and almost all n, then $T(n) = \mathcal{O}(f(n))$.

Remark.

- For a proof see [CLRS].
- "For almost all n" means for all but finitely many n.
- The asymptotic behaviour of T does not depend on the initial condition for $T(0)$.
- In the recursive equation, $T(n/b)$ can be replaced by $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$.
- The asymptotic behaviour of T is determined by the larger of the two functions $f(n)$ and $g(n) = n^{\log_b a}$. Here "larger" means larger by a polynomial factor n^{ϵ} for some $\epsilon > 0$. Roughly writing, the three cases are
  - $f \in \mathcal{O}(g)$ and $g \notin \mathcal{O}(f)$ imply $T \in \mathcal{O}(g)$
  - $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$ imply $T \in \mathcal{O}(g \cdot \log)$
  - $f \notin \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$ imply $T \in \mathcal{O}(f)$
- The condition $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ in the first case implies $f(n) = \mathcal{O}(n^{\log_b a})$ in the middle case. We use the better bound of the first case whenever it applies.
- Note the extra *regularity condition* $\exists c < 1 \big(af(n/b) \leqslant cf(n)\big)$ in the third case.
- The three cases do not cover all possibilities.

Example. We analyse binary search. The recursive equation is

$$T(n) = T(n/2) + 1\,.$$

Now $n^{\log_2 1} = 1$ implies

$$T(n) = \mathcal{O}(\log n)\,.$$

In case of merge sort the recursive equation is

$$T(n) = 2T(n/2) + n\,.$$

Since $n^{\log_2 2} = n$ the master theorem gives

$$T(n) = \mathcal{O}(n \log n)\,.$$

Theorem 17 (New Master Theorem [AB98]). Let $T : \mathbb{R}_{\geq 1} \to \mathbb{R}_{>0}$ be a function such that

$$T(x) = \begin{cases} h(x) & \text{for } 1 \leq x \leq x_0 \\ a \cdot T(x/b) + f(x) & \text{for } x > x_0 \end{cases}$$

where

1. $a > 0$, $b > 1$, and $x_0 \geq b$ are constants,
2. $d_1 \leq h(x) \leq d_2$ for some positive constants d_1 and d_2 and all x with $1 \leq x \leq x_0$,
3. f is a nonnegative function satisfying the *polynomial growth condition*, i.e., there are positive constants c_1 and c_2 such that $c_1 f(x) \leq f(u) \leq c_2 f(x)$ for all $x > x_0$ and u with $x/b \leq u \leq x$, and
4. (technical condition) $I(x_0) < \infty$ where $I(x) = \int_1^x \frac{f(u)}{u^{p+1}}\,\mathrm{d}u$ and $p = \log_b a$.

Then $T(x) = \Theta(x^p(1 + I(x)))$.

Remark.

- For a proof see [AB98], for further information see [Lei].
- For $f(u) = u^p$ we have $\int_1^x \frac{1}{u}\,\mathrm{d}u = \ln x$.
- We have $\int_1^x u^e\,\mathrm{d}u = (x^{e+1} - 1)/(e + 1)$ for all real exponents $e \neq -1$.

Example.

- If $T(x) = 4T(x/2) + x$ then $p = 2$ and $T(x) = \Theta(x^2(1 + \int_1^x \frac{u}{u^3}\,\mathrm{d}u)) = \Theta(x^2)$.
- If $T(x) = 2T(x/2) + x$ then $p = 1$ and $T(x) = \Theta(x(1 + \int_1^x \frac{u}{u^2}\,\mathrm{d}u)) = \Theta(x \log x)$.
- If $T(x) = 3T(x/2) + x^2$ then $p = \log_2 3$ and $T(x) = \Theta(x^{\log 3}(1 + \int_1^x \frac{u^2}{u^{1+\log 3}}\,\mathrm{d}u)) = \Theta(x^{\log 3}(1 + x^{2-\log 3})) = \Theta(x^2)$.
- If $T(x) = T(x/2) + \log x$ then $p = 0$ and $T(x) = \Theta(x^0(1 + \int_1^x \frac{\log u}{u}\,\mathrm{d}u)) = \Theta(\log^2 x)$.

5.3 Maximum Independent Set

5.3.1 Graph-Theoretic Concept

Let $G = (V, E)$ be a finite, simple, undirected graph. A set $U \subseteq V$ is *independent* if $\{u, v\} \notin E$ for all $u, v \in U$. U is a *clique* if $\{u, v\} \in E$ for all $u \in U$ and all $v \in U \setminus \{u\}$. A vertex of degree 0 is *isolated* and a vertex of degree 1 is *pendent*. A vertex $s \in V$ is *simplicial* in G if $N(s)$ is a clique of G. Isolated and pendent vertices are simplicial. By $\alpha(G)$ we denote the maximum size of an independent set of G.

Lemma 18. Let $G = (V, E)$ be a graph as above.

1. $\alpha(G) = 0$ if and only if $V = \varnothing$.
2. $\alpha(G) = 1 + \alpha(G - N[s])$ holds for every simplicial vertex s of G.

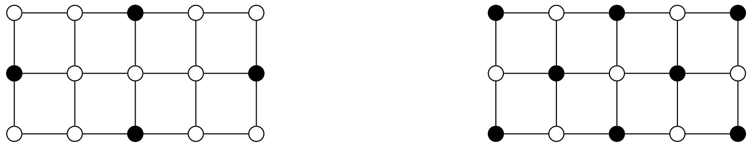


Figure 8: The black vertices form independent sets in this grid graph.
The independent set on the left is maximal, the one in the right is maximum.

3. Let v and w be adjacent non-simplicial vertices of degree 2 in G. Then $\alpha(G) = 1+\alpha(G')$, where G' is the graph obtained from $G-\{v,w\}$ by adding the edge $\{u,x\}$ and (u,v,w,x) is a path in G.

4. Let v be a vertex of degree 2 with non-adjacent neighbours u and w. Then $\alpha(G) = \max\{1+\alpha(G-N[v]), 2+\alpha(G-N[u]-N[w])\}$.

5. $\alpha(G) = \max\{\alpha(G-v), 1+\alpha(G-N[v])\}$ holds for every vertex v of G.

Proof. The set $\{v\}$ is independent for every vertex $v \in V$. This implies 1.

Let s be a simplicial vertex of G and let U be a maximum independent set of G. If $N[s] \cap U = \varnothing$ then $\{s\} \cup U$ is also independent contradicting the maximality of U. Otherwise $N[s] \cap U$ contains exactly one vertex, since $N[s]$ is a clique and U is independent. Since $\{s\} \cup (U \setminus N[s])$ is independent too and 2 follows.

In case 3 we have $\alpha(G) \leqslant 1+\alpha(G')$ since v and w are adjacent in G. For the sake of a contradiction we assume an independent set U' of G' such tat neither $U' \cup \{v\}$ nor $U' \cup \{w\}$ is independent in G. Then both $u \in U'$ and $x \in U'$, contradicting the fact that U' is independent and $\{u,x\}$ is an edge of G'. This implies 3

In case 4 we show that for every independent set U of G there is an independent set U' with $|U| \leqslant |U'|$ such that $v \in U'$ or $\{u,w\} \subseteq U'$. If $\{u,w\} \subseteq U$ then let $U' = U$, otherwise $U' = \{v\} \cup U \setminus \{u,w\}$.

Finally let v be any vertex of G and let U be a maximum independent set. Then $v \in U$ implies $\alpha(G) = 1+\alpha(G-N[v])$ and $v \notin U$ implies $\alpha(G) = \alpha(G-v)$. Together this is 5. □

5.3.2 Algorithms

We can turn the Lemma into a divide and conquer algorithm computing $\alpha(G)$ for the input graph G. If we use only properties 1 and 5 we obtain algorithm 11.

```
function mis ($G$)
    case do
        $V = \varnothing$ :   return 0;
        true :   choose a vertex $v \in V$; return max(mis($G-v$), $1+$mis($G-N[v]$));
```

Algorithm 11: Maximum idependent set in time $\mathcal{O}(2^n)$.

In the worst case the algorithm always chooses an isolated vertex v. Hence its running time $T(n)$ can be described by the following recurrence:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 2 \cdot T(n-1) && \text{for } n > 0, \end{aligned}$$

which means $T(n) = 2^n$. This might be faster than checking all 2^n subsets of V for independence, but only by a polynomial factor. To improve the running time we should ensure $|N[v]| > 1$ in the default case.

```
function mis (G)
    case do
        $V = \varnothing$ :   return 0;
        $s \in V$ is isolated in $(V, E)$ :   return $1 + \texttt{mis}(G - s)$;
        true :   choose a vertex $v \in V$; return $\max(\texttt{mis}(G - v), 1 + \texttt{mis}(G - N[v]))$;
```

Algorithm 12: Maximum idependent set in time $\mathcal{O}(1.618^n)$.

We could do so by considering isolated vertices separately. This would give algorithm 12. The improved run time is described by

$$\begin{aligned} T(0) &= 1 \\ T(n) &= \max\{T(n-1), T(n-1) + T(n-2)\} && \text{for } n > 0, \end{aligned}$$

which is the FIBONACCI-sequence. Therefore we have $T(n) = \mathcal{O}(1.618^n)$.

```
function mis (G)
    case do
        $V = \varnothing$ :   return 0;
        $s \in V$ is isolated or pendant in $(V, E)$ :   return $1 + \texttt{mis}(G - s)$;
        true :   choose a vertex $v \in V$; return $\max(\texttt{mis}(G - v), 1 + \texttt{mis}(G - N[v]))$;
```

Algorithm 13: Maximum idependent set in time $\mathcal{O}(1.4656^n)$.

If we consider isolated and pendant vertices separately, the recurrence would be

$$\begin{aligned} T(0) &= 1 \\ T(n) &= \max\{T(n-1), T(n-1) + T(n-3)\} && \text{for } n > 0, \end{aligned}$$

which leads to $T(n) = \mathcal{O}(1.4656^n)$.

If we additionally use 3 from the lemma we obtain

$$\begin{aligned} T(0) &= 1 \\ T(n) &= \max\{T(n-1), T(n-2), T(n-3) + T(n-5), T(n-1) + T(n-4)\} && \text{for } n > 0, \end{aligned}$$

where the recursive case corresponds with 2, 3, 4 and 5 from the lemma. The maximum real root of $c^5 - c^2 - 1 = 0$ is $c_{\max} = 1.1939$, and the maximum real root of $d^4 - d^3 - 1 = 0$ is $d_{\max} = 1.3803$. Hence this algorithm runs in time $\mathcal{O}(1.3803^n)$.

5.3.3 Further algorithms

In 1977 Tajan and Trojanowski [TaTr] developed an $\mathcal{O}(2^{n/3})$-time algorithm for the maximum independent set problem, where $2^{1/3} \approx 1.2599$. In 2001 Robson [Rob] claimed an $\mathcal{O}(2^{n/4})$-time algorithm; $2^{1/4} \approx 1.1892$.

5.4 Integer Multiplication

5.4.1 History

Given two n-bit integers, what is the best asymptotic running time, in terms of n, to multiply them? In 1960, A. N. Kolmogorov conjectured an $\Omega(n^2)$ lower bound. He organised a seminar at the Moscow State University during which he stated this conjecture. One week later, a bright young student named A. A. Karatsuba had disproved his conjecture by providing an algorithm with running time $O(n^{\log_2 3})$ [Kar].

5.4.2 Karatsuba's algorithm

Let x and y be two n-bit integers. Let $x = x_{\mathrm{h}} b^n + x_{\mathrm{l}}$ and $y = y_{\mathrm{h}} b^n + y_{\mathrm{l}}$. Then $xy = x_{\mathrm{h}} y_{\mathrm{h}} b^{2n} + (x_{\mathrm{h}} y_{\mathrm{l}} + x_{\mathrm{l}} y_{\mathrm{h}}) b^n + x_{\mathrm{l}} y_{\mathrm{l}}$. On the other hand, $xy = p_1 b^{2n} + (p_3 - p_1 - p_2) b^n + p_2$ where $p_1 = x_{\mathrm{h}} y_{\mathrm{h}}$, $p_2 = x_{\mathrm{l}} y_{\mathrm{l}}$ and $p_3 = (x_{\mathrm{h}} + x_{\mathrm{l}})(y_{\mathrm{h}} + y_{\mathrm{l}})$. The latter identity can by turned into a recursive algorithm running in time $T(n) = 3T(n/2) + n = \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.585})$.

5.4.3 Further algorithms

In 1971 Schönhage and Strassen [ScSt] developed an algorithms that multiplies two n-bit integers in time $\mathcal{O}(n \cdot \log n \cdot \log\log n)$. This algorithm uses fast Fourier transforms. In 2007 Fürer improved this to $\mathcal{O}(n \cdot \log n \cdot 2^{\log^* n})$. In 2019 Harvey and van der Hoeven proposed an $\mathcal{O}(n \log n)$ algorithm [HvdH].

5.5 Matrix Multiplication

5.5.1 Strassen's Algorithm—Idea

Strassen [Str] applied the divide-and-conquer method to matrix multiplication. To multiply two $2n \times 2n$ matrices he divides each of them into four $n \times n$ sub-matrices.

$$\begin{array}{ccccc} C & = & A & \cdot & B \\ \begin{pmatrix} r & s \\ t & u \end{pmatrix} & = & \begin{pmatrix} a & b \\ c & d \end{pmatrix} & \cdot & \begin{pmatrix} e & f \\ g & h \end{pmatrix} \end{array}$$

$$r = ae + bg \qquad\qquad s = af + bh$$
$$t = ce + dg \qquad\qquad u = cf + dh$$

If we evaluate the expressions for r, s, t and u in the standard way, and multiply the submatrices by the same method, we obtain a running time of $T(n) = 8T(n/2) + \mathcal{O}(n^2) = \mathcal{O}(n^3)$, which matches the usual bound. If we can save one multiplication, the bound would be $T(n) = 7T(n/2) + \mathcal{O}(n^2) = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.81})$.

5.5.2 Strassen's Algorithm—Overview

1. Divide A into sub-matrices a, b, c, d and B into e, f, g, h. A and B are $2n \times 2n$ matrices, all sub-matrices are $n \times n$.
2. Using $\mathcal{O}(n^2)$ scalar additions and subtractions, compute 14 matrices A_1, B_1, A_2, B_2, $\ldots$, A_7, B_7, each of which is $n \times n$.
3. Recursively compute the seven matrix products $C_i = A_i B_i$ for $i = 1, 2, \ldots, 7$.
4. Compute the desired sub-matrices r, s, t, u of the result matrix C by adding or subtracting various combinations of the C_i matrices, using only $\mathcal{O}(n^2)$ scalar additions and subtractions.

5.5.3 Strassen's Algorithm—Details

$$\begin{array}{llll}
A_1 = a & B_1 = f - h & A_2 = a + b & B_2 = h \\
A_3 = c + d & B_3 = e & A_4 = d & B_4 = g - e \\
A_5 = a + d & B_5 = e + h & A_6 = b - d & B_6 = g + h \\
A_7 = a - c & B_7 = e + f & &
\end{array}$$

$$\begin{array}{ll}
r = C_5 + C_4 - C_2 + C_6 & s = C_1 + C_2 \\
t = C_3 + C_4 & u = C_5 + C_1 - C_3 - C_7
\end{array}$$

5.5.4 Strassen's Algorithm—Discussion

- The constant factor hidden in the running time of STRASSEN's algorithm is larger than the constant factor in the naive $\mathcal{O}(n^3)$ method.
- When the matrices are sparse, methods tailored for sparse matrices are faster
- STRASSEN's algorithm is not quite as numerically stable as the naive method.
- The sub-matrices formed at levels of recursion consume space.

5.5.5 Further Improvement

COPPERSMITH and WINOGRAD developed an algorithm in 1990 that multiplies two $n \times n$ matrices in $\mathcal{O}(n^{2.376})$ time [CoWi]. In 2014 WILLIAMS proposed an algorithm that runs in time $\mathcal{O}(n^{2.373})$ [Will].

6 Abstract Data Types

An *abstract data type* (*ADT*) is a collection of types, constants and operations that can be performed on objects of these types, which are specified by characteristic identities.

The *stack*, the *queue*, the *priority queue*, and the *dictionary* are important examples of abstract data types.

Modern programming languages support the implementation of ADTs by means of classes.

Example. ADT Stack

types

E	type of elements
$S = S(E)$	type of stacks (parametrised)

constants and functions

`empty`	:	S
`isEmpty`	:	$S \to \mathbf{bool}$
`push`	:	$S \times E \to S$
`peek`	:	$S \to E$
`pop`	:	$S \to S$

specification

$$\begin{aligned}
\texttt{isEmpty}(\texttt{empty}) &= \mathsf{true} \\
\texttt{isEmpty}(\texttt{push}(s,x)) &= \mathsf{false} && \text{for all } s \in S, x \in E \\
\texttt{peek}(\texttt{empty}) &= \bot \\
\texttt{peek}(\texttt{push}(s,x)) &= x && \text{for all } s \in S, x \in E \\
\texttt{pop}(\texttt{empty}) &= \bot \\
\texttt{pop}(\texttt{push}(s,x)) &= s && \text{for all } s \in S, x \in E
\end{aligned}$$

7 ADT priority queue

7.1 Definition

types

E	linearly ordered type of elements
$P = P(E)$	type of priority queues (parametrised)

constants and functions

`empty`	:	P
`isEmpty`	:	$P \to \mathbf{bool}$
`insert`	:	$P \times E \to P$
`minimum`	:	$P \to E$
`deleteMin`	:	$P \to P$

specification

Let $\texttt{set}(x_1, x_2, x_3, \ldots, x_n) = \texttt{insert}(\texttt{insert}(\texttt{insert}(\ldots\texttt{insert}(\texttt{empty}, x_n), \ldots, x_3), x_2), x_1)$.

$$\begin{aligned}
\texttt{isEmpty}(\texttt{empty}) &= \mathsf{true} && \\
\texttt{isEmpty}(\texttt{insert}(p, x)) &= \mathsf{false} && \text{for all } p \in P, x \in E \\
\texttt{insert}(\texttt{insert}(p, x), y) &= \texttt{insert}(\texttt{insert}(p, y), x) && \text{for all } p \in P, x, y \in E \\
\texttt{minimum}(\texttt{empty}) &= \bot && \\
\texttt{minimum}(\texttt{set}(x_1, \ldots, x_n)) &= \min\{x_1, \ldots, x_n\} && \text{for all } x_1, \ldots, x_n \in E \\
\texttt{deleteMin}(\texttt{empty}) &= \bot && \\
\texttt{deleteMin}(\texttt{set}(x_1, \ldots, x_n)) &= \texttt{set}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) && \text{where } x_i = \min\{x_1, \ldots, x_n\} \\
& && \text{for all } x_1, \ldots, x_n \in E
\end{aligned}$$

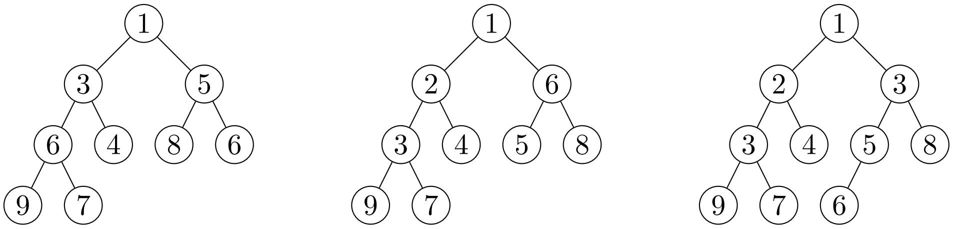
7.2 Binary Heaps

A *binary heap* is a rooted binary ordered tree with nodes from a totally ordered universe characterised by its

heap order: every parent node is less than all its children, and its

heap shape: all layers are complete except the last one, which is filled from the left.

Example. A binary heap, a binary tree not in heap order, and a binary tree not in heap shape.



We will use binary heaps to implement the ADT priority queue.

7.2.1 Insert

We insert the new element in the last layer such that the heap shape is maintained. Then we "toggle up" to re-establish the heap order.

7.2.2 Delete Minimum

We remove the root (which contains the minimum) and replace it by the last element. This way we maintain the heap shape. Now we "toggle down" do re-establish the heap order.

7.2.3 Time Analysis

A heap of size $2^h - 1$ is a complete binary tree with h levels. Hence a heap of size n has $\lceil \log n \rceil$ levels. Running `insert` or `deleteMin` we toggle at most $\lceil \log n \rceil - 1$ times. Consequently both functions will run in time $\mathcal{O}(\log n)$.

7.3 Binary Heap as Array

Binary heaps can be implemented by pointers. Alternatively, a heap of size n can be stored in an array $h[1..n]$. The root node goes to $h[1]$ and the children of the node in $h[i]$ can be found in $h[2i]$ and $h[2i+1]$.

```
function empty
    allocate memory for an array $h[0..\texttt{maxHeapSize}]$;
    return (0,h)

function isEmpty $(n, h)$
    if $n = 0$ then
        return true
    else
        return false

function minimum $(n, h)$
    return $h[1]$

function swap $((n, h), i, j)$
    $t \leftarrow h[i]$; $h[i] \leftarrow h[j]$; $h[j] \leftarrow t$;
    return $(n, h)$

function toggleUp $((n, h), i)$
    $j \leftarrow \lfloor i/2 \rfloor$;
    if $i > 1$ and $h[j] > h[i]$ then
        return $tgu(\texttt{swap}((n, h), i, j), j)$
    else
        return $(n, h)$

function toggleDown $((n, h), i)$
    $j \leftarrow 2 * i$;
    if $j + 1 \leqslant n$ and $h[j + 1] < h[j]$ then $j \leftarrow j + 1$;
    if $j \leqslant n$ and $h[j] < h[i]$ then
        return toggleDown(swap$((n, h), i, j), j)$
    else
        return $(n, h)$

function insert $((n, h), t)$
    $h[n] \leftarrow t$; $n \leftarrow n + 1$;
    return $tgu((h, n), n)$

function deleteMin $((n, h))$
    $h[1] \leftarrow h[n]$; $n \leftarrow n - 1$;
    return toggleDown$((h, n), 1)$
```

Algorithm 14: binary heap

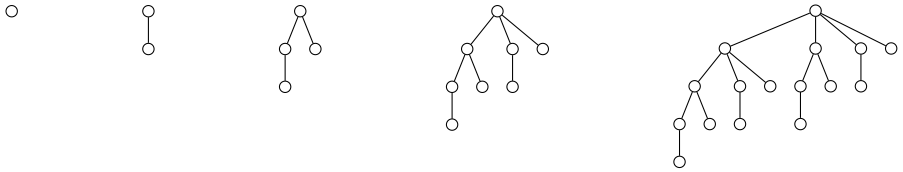
7.4 Binomial Heaps

A binomial heap is a priority queue data structure similar to the binary heap, but more versatile. It supports quicker merging of two heaps, is a building block in other data structures (*e.g.* FIBONACCI heaps), and has better amortized running time.

7.4.1 Binomial Trees

A *binomial tree* B_k is a rooted ordered tree defined recursively. B_0 consists of a single node. B_k consists of two binomial trees B_{k-1} that are linked together: the root of one is the leftmost child of the root of the other.

Example.



Lemma 19. For all $k \geqslant 0$, the binomial tree B_k,

- contains 2^k nodes,
- has height k,
- contains exactly $\binom{k}{i}$ nodes of depth i for $i = 0, 1, \ldots, k$, and
- the root of B_k has degree k and its children are roots of $B_{k-1}, B_{k-2}, B_{k-3}, \ldots, B_0$.

7.4.2 Binomial Heaps

A *binomial heap* is a collection of binomial trees with the following properties:

- each binomial tree is in heap order,
- for each k there is at most one B_k in the collection.

Let $n = (b_\ell b_{\ell-1} \ldots b_1 b_0)_2$, that is $n = \sum_{k=0}^{\ell} b_k 2^k$ for $b_k \in \{0, 1\}$. Then a binomial heap of size n contains a binomial tree B_k if and only if $b_k = 1$ for $0 \leqslant k \leqslant \log n$.

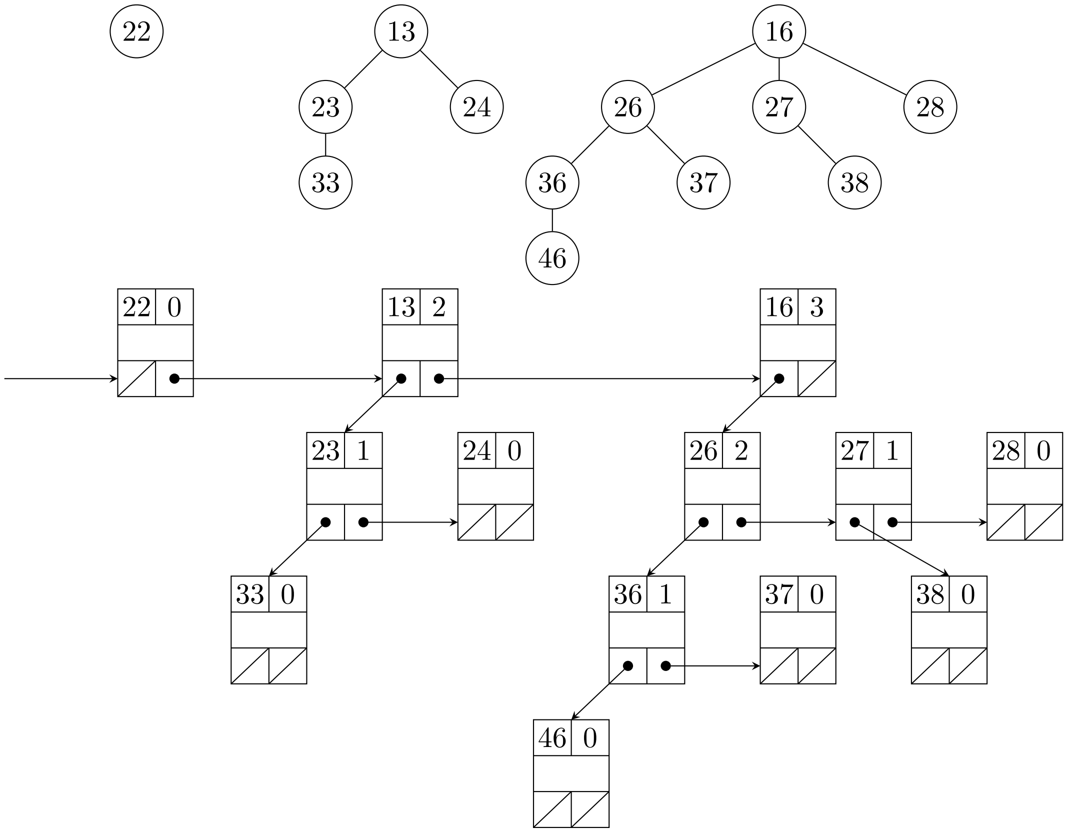
7.4.3 Left-Child, Right-Sibling Representation

For each node of a binomial tree we store

- its degree (degree)
- the key (key)
- all satellite information (info)
- a pointer to its leftmost child (child)
- a pointer to its right sibling (sibling)

<table>
<tr><td>key</td><td>degree</td></tr>
<tr><td colspan="2">info</td></tr>
<tr><td>child</td><td>sibling</td></tr>
</table>

Example.



7.5 Summary of Running Times (**w**orst case/**a**mortised)

	binary heap (w)	binomial heap (w)	FIBONACCI heap (a)
`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`minimum`	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`deleteMin`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`merge`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`decreaseKey`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`delete`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

```
function empty
    return nil

function isEmpty($h$)
    if $h$ = nil then
        return true
    else
        return false
```

Algorithm 15: binomial heap

```
function minimum ($h$)
    $m \leftarrow \infty$;
    while $h \neq$ nil do
        if $h$.key $< m$ then $m \leftarrow h$.key; $i \leftarrow h$.info;
        $h \leftarrow h$.sibling
    return $(m, i)$
function link ($s, t$)
    if $s$.key $< t$.key then
        $t$.sibling $\leftarrow s$.child;
        $s$.child $\leftarrow t$;
        $s$.degree $\leftarrow s$.degree $+ 1$;
        return $s$
    else
        $s$.sibling $\leftarrow t$.child;
        $t$.child $\leftarrow s$;
        $t$.degree $\leftarrow t$.degree $+ 1$;
        return $t$

function merge ($g, h$)
    if isEmpty($g$) then return $h$;
    if isEmpty($h$) then return $g$;
    if $g$.degree $< h$.degree then
        $g$.sibling $\leftarrow$ merge($g$.sibling, $h$);
        return $g$
    if $g$.degree $> h$.degree then
        $h$.sibling $\leftarrow$ merge($g$, $h$.sibling);
        return $h$
    $x \leftarrow$ merge($g$.sibling, $h$.sibling);
    $y \leftarrow$ link($g, h$); $y$.sibling $\leftarrow$ nil;
    return merge($x, y$)
function insert ($(k, i), h$)
    allocate a new cell at $t$;
    $t$.degree $\leftarrow 0$;
    $t$.key $\leftarrow k$;
    $t$.info $\leftarrow i$;
    $t$.child $\leftarrow$ nil;
    $t$.sibling $\leftarrow$ nil;
    return merge($t, h$)
```

Algorithm 16: binomial heap—continued

```
function deleteMin($h$)
    $(m, i) \leftarrow$ minimum($h$);
    if $h$.key $= m$ then
        $x \leftarrow h$.sibling;
        $t \leftarrow h$;
        $t$.sibling $\leftarrow$ nil
    else
        $x \leftarrow h$; $z \leftarrow h$;
        while $z$.sibling.key $\neq m$ do $z \leftarrow z$.sibling;
        $t \leftarrow z$.sibling;
        $z$.sibling $\leftarrow z$.sibling.sibling
    $s \leftarrow t$.child;
    $y \leftarrow$ nil;
    while $s$.sibling $\neq$ nil do
        $z \leftarrow s$;
        $z$.sibling $\leftarrow y$;
        $y \leftarrow z$;
        $s \leftarrow s$.sibling
    deallocate the cell at $t$;
    return merge($x, y$)
```

Algorithm 17: binomial heap—further continued

8 ADT dictionary

8.1 Definition

types	K	totally ordered type of keys
	I	type of information
	$D = D(K, I)$	type of dictionaries (parametrised)

constants and functions

`empty`	:	D
`isEmpty`	:	$D \to$ **bool**
`member`	:	$D \times K \to$ **bool**
`lookUp`	:	$D \times K \to I$
`insert`	:	$D \times K \times I \to D$
`delete`	:	$D \times K \to D$

specification

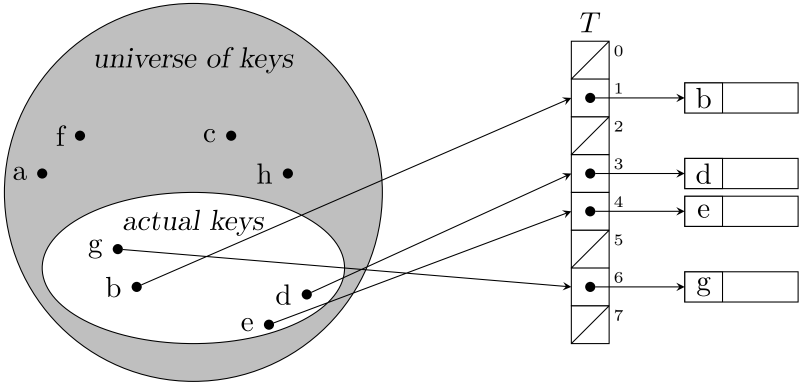
$$
\begin{array}{lcll}
\texttt{isEmpty}(\texttt{empty}) & = & \textsf{true} & \\
\texttt{isEmpty}(\texttt{insert}(d,k,i)) & = & \textsf{false} & \text{for all } d \in D, k \in K, i \in I \\
\texttt{member}(\texttt{empty}, l) & = & \textsf{false} & \text{for all } l \in K \\
\texttt{member}(\texttt{insert}(d,k,i), k) & = & \textsf{true} & \text{for all } d \in D, k \in K, i \in I \\
\texttt{member}(\texttt{insert}(d,k,i), l) & = & \texttt{member}(d,l) & \text{for all } d \in D, k, l \in K, i \in I, k \neq l \\
\texttt{lookUp}(\texttt{empty}, l) & = & \bot & \text{for all } l \in K \\
\texttt{lookUp}(\texttt{insert}(d,k,i), k) & = & i & \text{for all } d \in D, k \in K, i \in I \\
\texttt{lookUp}(\texttt{insert}(d,k,i), l) & = & \texttt{lookUp}(d,l) & \text{for all } d \in D, k, l \in K, i \in I, k \neq l \\
\texttt{insert}(\texttt{insert}(d,k,i), k, j) & = & \bot & \text{for all } d \in D, k \in K, i, j \in I \\
\texttt{delete}(\texttt{empty}, l) & = & \bot & \text{for all } l \in K \\
\texttt{delete}(\texttt{insert}(d,k,i), k) & = & d & \text{for all } d \in D, k \in K, i \in I \\
\texttt{delete}(\texttt{insert}(d,k,i), l) & = & \texttt{insert}(\texttt{delete}(d,l), k, i) & \\
 & & & \text{for all } d \in D, k, l \in K, i \in I, k \neq l
\end{array}
$$

8.2 Hashing

Hashing is a method to implement the ADT dictionary. A *hash table* is an array $T[0..m-1]$ of pointers to cells. Each cell contains one key and the corresponding satellite information. A *hash function* is a mapping $h : K \to \{0, 1, 2, \ldots, m-1\}$, where K is the set of all possible keys (sometimes called U, the universe of keys) and $\{0, 1, 2, \ldots, m-1\}$ is the set of *slots* in the hash table. If $|K| > m$ then there are different keys k_1 and k_2 such that $h(k_1) = h(k_2)$. We call this situation a *collision*.

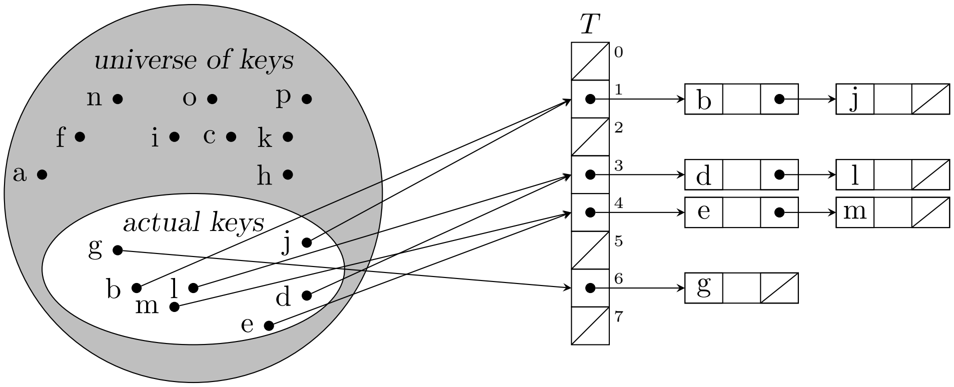
8.2.1 Direct-Address Tables

In this case we have $|K| = m$ and h is a one-to-one function.



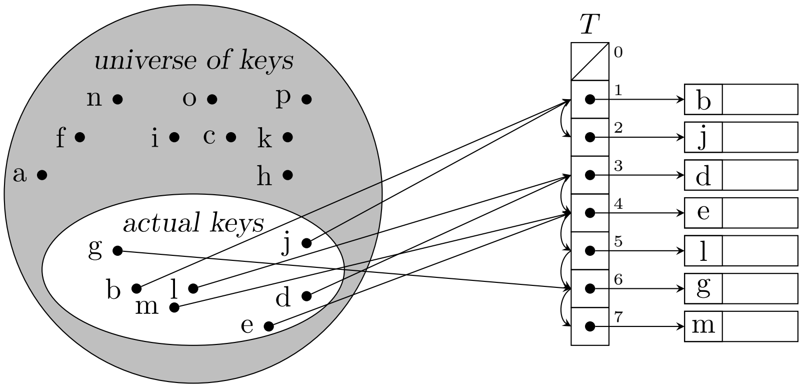
$K = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{h}\}$ $\qquad$ $m = 8$ $\qquad$ $h(x) = x - \mathrm{a}$

8.2.2 Closed-Address Tables—Chaining



$K = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{p}\}$ $\qquad m = 8$ $\qquad h(x) = (x - \mathrm{a}) \bmod m$

8.2.3 Open-Address Tables—Probing



$K = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{p}\}$ $\qquad m = 8$ $\qquad h(x, i) = (x - \mathrm{a} + i) \bmod m$

8.2.4 Analysis of Closed-Address Tables

The *load factor* of a hash table T with m slots that stores n elements is $\alpha = \frac{n}{m}$.

If all elements hash to the same slot we create a list of length n. Hence in the worst case the operations `member`, `lookUp` and `delete` require $\mathcal{O}(n)$ time. In the best case, all elements hash to different slots and hence the running time is constant.

The average performance depends on how well the keys are distributed to slots. Here we assume that any element is equally likely to hash into any of the slots, independent of where any other element has hashed to. This is the assumption of *simple uniform hashing*.

For $j = 0, 1, 2, \ldots, m-1$, let n_j denote the length of the list $T[j]$, so that

$$n = n_0 + n_1 + \cdots + n_{m-1}\,.$$

The expected value of n_j is $\mathrm{E}[n_j] = \alpha$ where $\alpha = n/m$.

Theorem 20. In a hash table where collisions are resolved by chaining, an unsuccessful search takes expected time $\mathcal{O}(1+\alpha)$, under the assumption of simple uniform hashing.

Successful Search (*for information only*)

Let $x_1, x_2, \dots, x_n$ be the elements in the table and $k_1, k_2, \dots, k_n$ their keys. We define a random variable $X_{i,j} = 1$ if $h(k_i) = h(k_j)$ and $X_{i,j} = 0$ otherwise.

Under the assumption of simple uniform hashing we have $\Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$ and hence $\mathrm{E}[X_{i,j}] = \frac{1}{m}$.

The expected number of elements examined in a successful search is

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{i,j}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}\left(n^2 - \frac{1}{2}n(n+1)\right) \\
&= 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
\end{aligned}$$

Theorem 21. In a hash table where collisions are resolved by chaining, a successful search requires $\mathcal{O}(1+\alpha)$ expected time under the assumption of simple uniform hashing.

We give an overvew of the running times of the operations. `insert` can be done in constant time by adding new elements at the *beginning* of the lists.

operations	expected time
`empty`, `isEmpty`	$\mathcal{O}(m)$
`member`, `lookUp`, `delete`	$\mathcal{O}(1+\alpha)$
`insert`	$\mathcal{O}(1)$

What Makes a Good Hash Function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing. In practise we do not know the distribution of keys, and they may not be drawn independently. If the keys are known to be random real numbers k independently and uniform distributed in the range $0 \leqslant k < 1$, the hash function

$$h : \quad [0,1) \to \{0, 1, \dots, m-1\} \qquad k \mapsto \lfloor km \rfloor$$

satisfies the condition of simple uniform hashing.

Example. division method $h(k) = k \bmod m$

multiplication method Let $0 < A < 1$ (e.g. $A = (\sqrt{5}-1)/2$) and $h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$ where $kA \bmod 1 = kA - \lfloor kA \rfloor$.

Evaluating Polynomials

We have

$$\sum_{i=0}^{n} a_i x^i = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + \cdots + x \cdot (a_{n-1} + x \cdot a_n) \cdots)))$$

Hence $h(a_0 a_1 \dots a_n) = (\sum_{i=0}^{n} a_i x^i) \bmod m$ can be computed as follows:

```
begin
    $h \leftarrow a_n \bmod m$;
    for $i \leftarrow n-1$ downto $0$ do
        $h \leftarrow (a_i + x \cdot h) \bmod m$
```

8.2.5 Analysis of Open-Address Tables

In open addressing, all elements are stored in the hash table itself. The load factor α cannot exceed 1. To `insert` an element we successively examine, or *probe*, the hash table until we find an empty slot. The hash function becomes

$$h : K \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

where $h(k, \cdot)$ is a permutation for each $k \in K$.

linear probing $h(k, i) = (h'(k) + i) \bmod m$

quadratic probing $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

double hashing $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

To analyse open-address tables we assume *universal hashing*. That is, we choose a hash function at random independent from the set of keys to be stored.

Example. Let $p > m$ be a prime number.

$$\begin{aligned} h_{a,b}(k) &= ((ak + b) \bmod p) \bmod m \\ \mathcal{H}_{p,m} &= \{h_{a,b} : 1 \leqslant a < p, 0 \leqslant b < p\} \end{aligned}$$

The expected number of probes in an unsuccessful search is at most

$$\frac{1}{1-\alpha},$$

assuming universal hashing and $\alpha < 1$.

The expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

if each key in the table is equally likely to be searched for, assuming universal hashing and $\alpha < 1$.

8.3 Binary Search Trees

Binary search trees are used to implement the ADT dictionary. A binary search tree is a rooted ordered binary tree. Each node contains a key and the corresponding satellite information. The keys satisfy the *binary-search-tree property*:

> Let node x be an ancestor of node y, and let k and l be the keys stored in x and y, respectively. Then $l < k$ if y appears in the subtree rooted at the left child of x, and $k < l$ if y appears in the right subtree.

Time Analysis

operations	time
`empty`, `isEmpty`	$\mathcal{O}(1)$
`member`, `lookUp`	$\mathcal{O}(n)$
`insert`, `delete`	$\mathcal{O}(n)$

In a binary search tree storing n elements the distance from a leaf to the root is $n-1$ if the tree is in fact a path. This distance determines the running time of the operations `member`, `lookUp`, `insert` and `delete` in the worst case.

To speed-up these operations to run in $\mathcal{O}(\log n)$ time we *balance* the binary search tree.

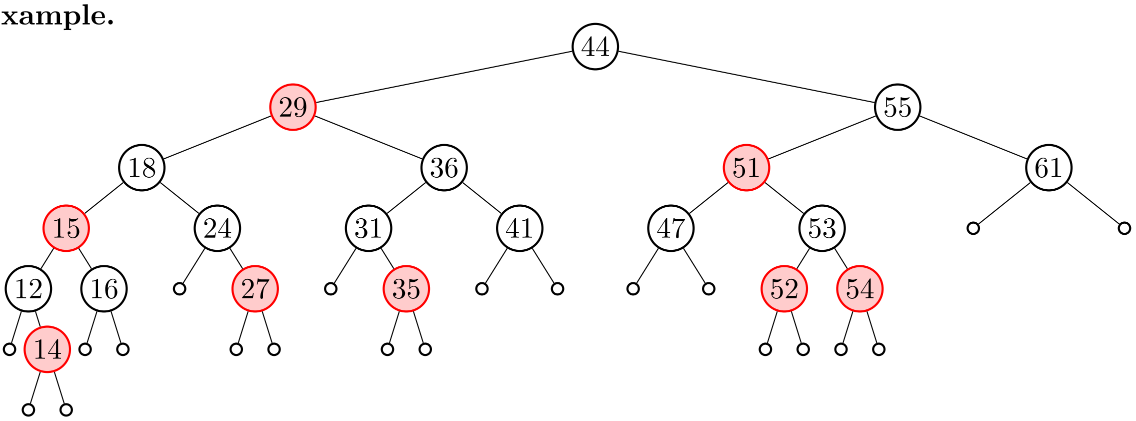
8.4 Red-Black Trees

A *red-black tree* [GuSe] is a binary search tree where the nodes are coloured red or black, augmented by void black leaves, such that

- every node is either red or black,
- the root is black,
- no red node has a red child, and
- every path from the root to a void leaf contains the same number of black nodes.

Red-black trees are balanced in the following way: The distances of two leaves from the root differ at most by a factor of two.

Example.



8.4.1 Black-Height

The number of black nodes on any path from, but not including, a node x down to a leaf is the *black-height* of the node, denoted by $\mathrm{bh}(x)$.

Lemma 22. A red-black tree with n internal nodes has height at most $2\log(n+1)$.

Proof.

- The subtree rooted at x contains at least $2^{\mathrm{bh}(x)}-1$ internal nodes.

  base step x has height 0.

  x is a leaf. The subtree rooted at x contains $2^{\mathrm{bh}(x)}-1 = 2^0-1 = 0$ internal nodes.

  inductive step x has positive height.

  The children y and z of x fulfil $\mathrm{bh}(x)-1 \leqslant \mathrm{bh}(y) \leqslant \mathrm{bh}(x)$ and $\mathrm{bh}(x)-1 \leqslant \mathrm{bh}(z) \leqslant \mathrm{bh}(x)$.

  By induction hypothesis, the subtree rooted at x contains at least $(2^{\mathrm{bh}(y)}-1)+(2^{\mathrm{bh}(z)}-1)+1 \geqslant 2^{\mathrm{bh}(x)}-1$ internal nodes.

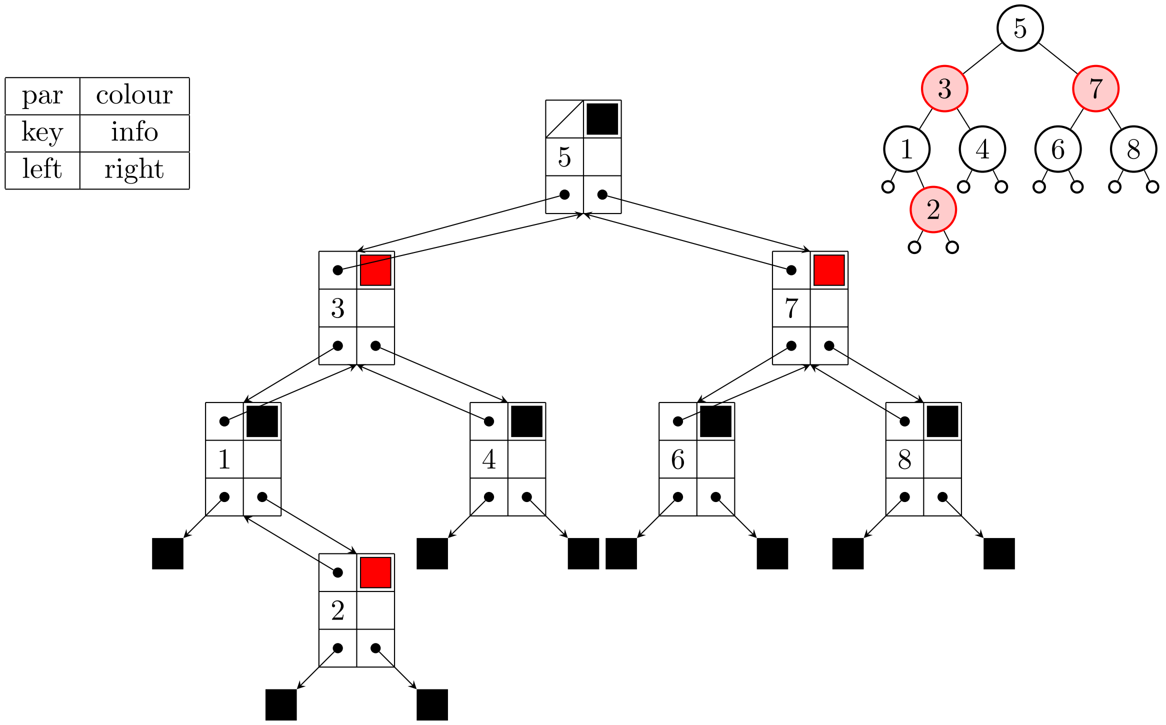
- Let h be the height of the tree. Then $n \geqslant 2^{h/2}-1$.

  Half of the nodes on any path from the root to a leaf, not including the root, must be black. The root has black-height at least $h/2$.

- $h \leqslant 2\log(n+1)$.

  Follows from $\log(n+1) \geqslant h/2$. □

8.4.2 Pointer Implementation



par	colour
key	info
left	right

8.4.3 Operations for Red-Black Trees

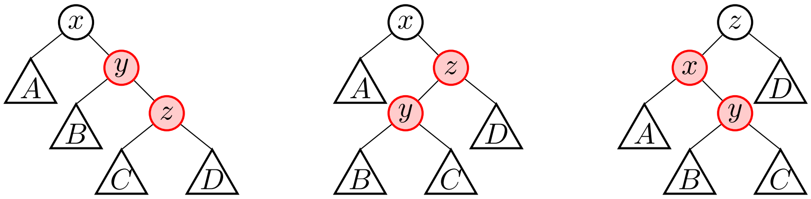
The operations `empty`, `isEmpty`, `member` and `lookUp` do not modify the tree and can be realised as for ordinary binary search trees.

The operation `insert` for binary search trees always adds a leaf. In red-black trees we colour the new leaf red. This way we maintain all properties of a red-black tree, but the new leaf might have a red parent node. We handle this case by *rotations*.

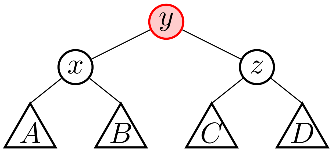
Without loss of generality we may restrict the operation `delete` to nodes adjacent to a void leaf. If such a node is red, it is safe to delete it. Otherwise we use rotations again.

Eliminating red nodes with red parents

Each red node with red parent matches one of the templates:



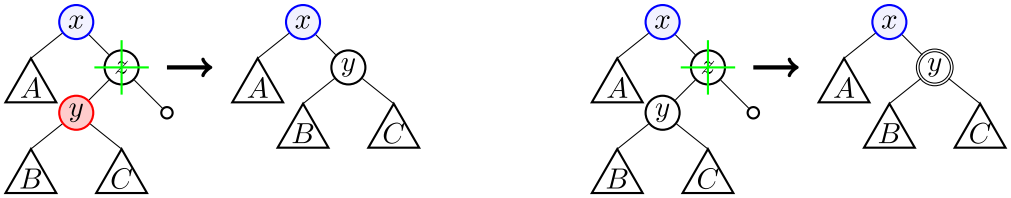
Each template rotates to:



Deleting a black node adjacent to a void

We want to remove a black node y from a red-black tree. The children of y are z and a void leaf. The idea is to move y's blackness to z.

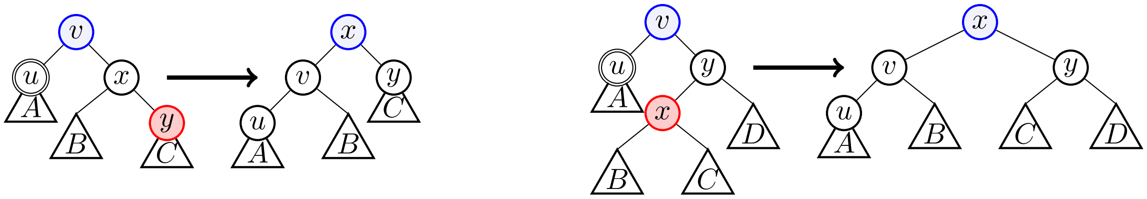
If z is red we simply change its colour to black. Otherwise we add an extra black to z, that is, z becomes double black. Node x keeps its colour, which is either red or black.



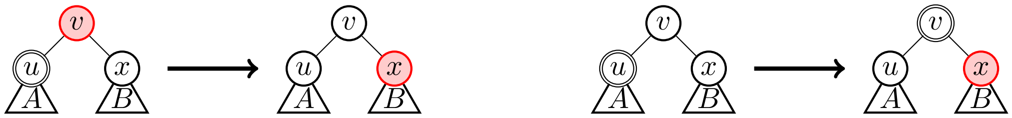
To get rid of double black nodes we rotate or push them towards the root.

Black Sibling With a Red Child

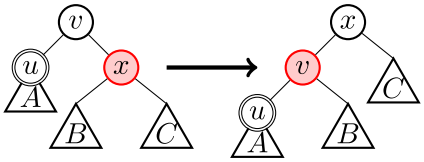
We apply a single or double rotation. Again, blue is a colour variable, either red or black.



Black Sibling With Black Children
We push one unit of blackness towards the root.



Red Sibling
We rotate such that one of the previous cases applies.



8.4.4 AVL Trees

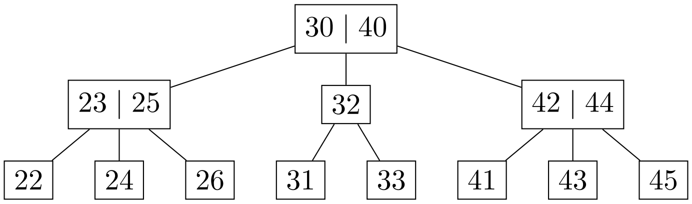
AVL trees (due to Adel'son-Vel'skiĭ and Landis [AVL]) are another type of balanced binary search trees. For each node in an AVL tree, the height of the left and right subtrees differ by at most 1. We store this difference in each node.

The operations `empty`, `isEmpty`, `member` and `lookUp` work as for ordinary binary search trees. For `insert` and `delete` we use rotations to keep the balance. Since an AVL tree with n nodes has height $\mathcal{O}(\log n)$, all dictionary operations run in time $\mathcal{O}(\log n)$.

8.4.5 2-3 Trees

A *2-3 tree* [Hop] is a rooted ordered search tree such that

- each internal node has degree 2 or 3, and
- all leaves have the same distance from the root.



2-3 trees can be used to implement the ADT dictionary. The operations `empty` and `isEmpty` will run in constant time, the remaining ones run in $\mathcal{O}(\log n)$ time.

References

[AHU] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: Data structures and algorithms. Addison-Wesley 1987.

[CLRS] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein: Introduction to Algorithms. MIT Press 2009.

[DPV] Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani: Algorithms. McGraw-Hill 2008.

[KT] Jon Kleinberg, Éva Tardos: Algorithm design. Addison Wesley 2005.

[Lev] Anany Levitin: Design and analysis of algorithms Pearson/Addison-Wesley 2007.

[AB98] Mohamad Akra and Louay Bazzi: On the solution of linear recurrence equations. *Computational Optimization and Applications* **10** (1998) 195–210.

[AVL] Georgy Adel'son-Vel'skiĭ and Evgenii Landis: An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences* **146** (1962) 263–266.

[Bel] Richard Bellman: Dynamic programming treatment of the traveling salesman problem. *Journal of the ACM* **9** (1962) 61–63.

[Bor] Jaroslav Nešetřil, Eva Milková and Helena Nešetřilová: Otakar Borůvka on minimum spanning tree problem. Translation of both the 1926 papers, comments, history. *Discrete Mathematics* **233** (2001) 3–36.

[Co] John Cocke and Jacob T. Schwartz: Programming languages and their compilers: Preliminary notes. Technical report CIMS, NYU, April 1970.

[CoWi] Don Coppersmith and Shmuel Winograd: Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* **9** (1990) 251–.280

[Dij] Edsger W. Dijkstra: A note on two problems in connexion with graphs. *Numerische Mathematik* **1** (1959) 269–271.

[Flo] Robert W. Floyd: Algorithm 97: shortest path. *Communications of the ACM* **5** (1962) 345.

[GuSe] Leonidas J. Guibas and Robert Sedgewick: A dichromatic framework for balanced trees. *Proceedings 19th FoCS* (1978) 8–21. http://doi.org/10.1109/SFCS.1978.3

[HeKa] Michael Held and Richard M. Karp: A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics* 10 (1962) 196–210.

[HvdH] David Harvey and Joris van der Hoeven: Integer multiplication in time $O(n \log n)$. 2019. hal-0207077 http://hal.archives-ouvertes.fr/hal-02070778

[Hop] John E. Hopcroft: 2-3 trees. Unpublished manuscript (1970) according to [CLRS].

[Jar] Vojtěch Jarník: O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti* **6** (1930) 57–63.

[Kar] Anatoly A. Karatsuba and Yuri P. Ofman: Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences* **145** (1962) 293–294.

[Kas] Tadao Kasami: An efficient recognition and syntax-analysis algorithm for context-free languages. University of Hawaii / Air Force Cambridge Lab AFCRL-65-758, (1965).

[Kru] Joseph B. Kruskal: On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* **7** (1956) 48–50.

[Lei] Tom Leighton: Notes on better master theorems for divide and conquer recurrences. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.1636

[Prim] Robert C. Prim: Shortest connection networks and some generalizations. *Bell System Technical Journal* **36** (1957) 1389–1401.

[Rob] John M. Robson: Algorithms for maximum independent sets. *Journal of Algorithms* **7** (1986) 425–440.

[ScSt] Arnold Schönhage and Volker Strassen: Schnelle Multiplikation großer Zahlen. *Computing* **7** (1971) 281–292.

[Str] Volker Strassen: Gaussian elimination is not optimal. *Numerische Mathematik* **13** (1969) 354–356.

[TaTr] Robert E. Tarjan and Anthony E. Trojanowski: Finding a maximum independent set. *SIAM Journal on Computing* **6** (1977) 537–546.

[War] Stephen Warshall: A theorem on boolean matrices. *Journal of the ACM* **9** (1962) 11–12.

[Will] Virginia Vassilevska Williams (Stanford University): Multiplying matrices in $O(n^{2.373})$ time. 2014. http://people.csail.mit.edu/virgi/matrixmult-f.pdf

[You] Daniel H. Younger: Recognition and parsing of context-free languages in time n^3. *Information and Control* **10** (1967) 189–208.