Inhaltsverzeichnis

1 Algorithmen und Komplexität

Oftmals steht man in der Informatik vor der Aufgabe, ein Problem (bzw. eine Klasse ähnlicher Probleme) automatisiert zu lösen. Dazu werden bspw. Programme entwickelt oder ganze Systeme entworfen, die diese Probleme lösen sollen. Will man diese *Lösungsansätze* genauer untersuchen (bspw. den benötigten Speicherplatz oder die Laufzeit für verschiedene Eingaben), so benötigt man zunächst eine präzisere Definition des Begriffs:

Definition 1.1 (Intuitive Begriffsbestimmung: Algorithmus)**.** *Ein* Algorithmus *ist eine präzise endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte. Das Verfahren ist hierbei eine wohldefinierte Rechenvorschrift, die eine Menge von Elementen als Eingabe verwendet und eine Menge von Elementen als Ausgabe erzeugt.*

Ein Algorithmus ist also eine Art Kochrezept oder eine Schritt-für-Schritt-Anleitung zur Lösung eines Problems. Die einzelnen Schritte sollten hier so einfach wie möglich sein (*elementar*) und es sollte insb. nur eine endliche Anzahl an Schritten geben. Will man Algorithmen untereinander weiter unterscheien, so kann man sie zunächst anhand der folgenden Eigenschaften klassifizieren:

Definition 1.2. *Man kann einen Algorithmus unter anderem anhand der folgenden Eigenschaft beschreiben:*

- *Abstrahierung*
  - *löst ganze Klasse von Problemen*
  - *konkretes Problem wird durch Parameter/ Daten spezifiziert*
- *Finitheit*
  - *Beschreibung hat endl. Länge (*statische Finitheit*)*
  - *Zu jedem Ausführungszeitpunkt belegt der Algo. nur endlich viel Platz (*dynamische Finitheit*)*
- *Terminierung*
  - *Hält nach endlich vielen Schritten und liefert ein Ergebnis*
  - *Kann auch sinnvoll sein, nichtterminierende Programme zu schreiben (Bspw. BS)*
- *Determinismus*
  - deterministisch: *Zu jedem Zeitpunkt gibt es genau eine Möglickeit zur Fortsetzung (sonst nichtdeterministisch)*
  - stochastisch: *Die Auswahl der Möglichkeiten (falls nichtdeterministisch) wird über Wahrscheinlichkeiten gesteuert (auch: randomisiert)*
- *Determiniertheit*

  - *Algo. heißt determiniert, falls zu gleicher Eingabe stets gleiche Ausgabe geliefert wird (dies setzt* nicht *vorraus, dass der Algo. deterministisch ist)*

- *Analysierbarkeit*
  - *Es lassen sich Laufzeit und Speicherplatzbedarf angeben*
- *Korrektheit*
  - *Sollte beweisbar sein, dass Algo. vorgegebenes Problem löst*

1.1 Algorithmische Prinzipien

Es gibt verschiedene Ansätze, um einen Algorithmus zu formulieren.

Definition 1.3 (Enumeration)**.** *Hierbei wird der Algorithmus dadruch definiert, dass zu einer Eingabe alle Lösungen aufgezählt werden. Es ist offensichtlich, dass dies in den wenigsten Fällen (zumindest in der Informatik) sinnvoll ist.*

Definition 1.4 (Divide & Conquer)**.** *Hierbei wird das Problem (in der Regel rekursiv) in Teilprobleme zerlegt, welche dann einzeln (evtl. parallel) gelöst werden können. Anschließend wird aus diesen Teillösungen eine Lösung für das Gesamtproblem (re-)konstruiert. In der Regel werden die Teilprobleme selbst wieder in kleiner Teilprobleme zerlegt bis die Lösung des einzelnen Teilproblems sehr einfach oder trivial ist.*

1.1.1 Greedy Algorithmen

Als *Greedy-Algorithmen* oder gierige Algorithmen bezeichnet man Algorithmen, die schrittweise denjenigen Folgezustand auswählen, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht. Dadurch sind Greedy-Algorithmen meist schnell, finden aber oft auch nicht die optimale Lösung eines Problems. Da es jedoch für eine Vielzahl von Problemen zu lange dauern würde oder sogar nicht innerhalb akzeptabler Zeit möglich ist, alle Lösungen zu berechnen um im Anschluss die optimale auszuwählen, nimmt man die nicht-optimale Lösung des Greedy-Verfahrens in Kauf.

1.1.2 Dynamische Programmierung

1.2 Komplexität

Um aus verschiedenen Algorithmen einen geeigneten auszuwählen, ist es oft nützlich, die Laufzeit der Algorithmen in Abhängikeit der Problemgröße zu kennen. Hierbei ist es nicht sinnvoll einfach nur die Zeit zu messen, die ein Algorithmus bei verschiedenen Eingabegrößen benötigt, da diese Zeit u. a. von der verwendeten Hardware und der genauen Implementierung oder der Programmiersprache abhängt. Stattdessen beschreibt man die Komplexität durch die Anzahl elementarer Operationen in Abhängigkeit der Eingabelänge und vernachlässigt hierbei konstante Faktoren, die nicht von der Eingabelänge abhängen.

Die *Komplexität* von Algorithmen wird vor allem durch zwei Größen beschrieben:

- **Laufzeit:** Anzahl der Schritte
- **Speicherbedarf:** Anzahl benutzter Speicherzellen

Es wäre zu mühsam die genaue Anzahl der Schritte festzustellen (es müsste dazu der verwendete Prozessor, Assembler etc. bekannt sein). Daher werden konstante Faktoren in Laufzeiten und Speicherplatz weitgehend ignoriert und man konzentriert sich auf das asymptotische Wachstum der Komplexität im Verhältnis zur Größe.

Definition 1.5 (Asymptotisches Wachstum)**.** *Zu einer Funktion $f : \mathbb{N}_0 \to \mathbb{R}$ wird definiert:*

(i) Die Menge

$$\mathcal{O}(f(n)) = \{\, g : \mathbb{N}_0 \to \mathbb{R} \;:\; \exists c, n_0 > 0 :\; \forall n > n_0 :\; |g(n)| \leq c \cdot |f(n)| \,\}$$

*der Funktionen, die **höchstens so schnell wachsen** wie f.*

(ii) Die Menge

$$\Omega(f(n)) = \{\, g : \mathbb{N}_0 \to \mathbb{R} \;:\; \exists c, n_0 > 0 :\; \forall n > n_0 :\; |g(n)| \geq c \cdot |f(n)| \,\}$$

*der Funktionen, die **mindestens so schnell wachsen** wie f.*

(iii) Die Menge

$$\Theta(f(n)) = \{\, g : \mathbb{N}_0 \to \mathbb{R} \;:\; \exists c_1, c_2, n_0 > 0 :\; \forall n > n_0 :\; c_1 \leq \frac{|g(n)|}{|f(n)|} \leq c_2 \,\}$$

*der Funktionen, **die so schnell wachsen** wie f.*

(iv) Die Menge

$$o(f(n)) = \{\, g : \mathbb{N}_0 \to \mathbb{R} \;:\; \forall c > 0 :\; \exists n_0 > 0 :\; \forall n > n_0 :\; c \cdot |g(n)| \leq |f(n)| \}$$

*der Funktionen, **die gegenüber** f verschwinden.*

(v) Die Menge

$$\omega(f(n)) = \{\, g : \mathbb{N}_0 \to \mathbb{R} \;:\; \forall c > 0 :\; \exists n_0 > 0 :\; \forall n > n_0 : |g(n)| \geq c \cdot |f(n)| \}$$

*der Funktionen, **denen gegenüber** f verschwinden.*

Anmerkung 1.6. *Offensichtlich gilt für $f : \mathbb{N} \to \mathbb{R}$*

$$\mathcal{O}(g) \cap \Omega(g) = \Theta(g)$$

und damit für $f, g : \mathbb{N}_0 \to \mathbb{R}$

$$f \in \mathcal{O}(g(n)) \wedge f \in \Omega(g(n)) \Leftrightarrow f \in \Theta(g(n)).$$

Beispiel 1.7. *Sei $p : \mathbb{N}_0 \to \mathbb{R}$ ein Polynom der Form $p(x) = \sum_{i=0}^{d} a_i \cdot x^i$ mit $a_i \in \mathbb{R}$ für $i \in \{0, \ldots, d\}$ und $a_d \neq 0$. Insb. ist also $deg(f) = d$. Dann gilt für alle $n \geq 1$:*

$$p(n) = \sum_{i=0}^{d} a_i \cdot n^i = \left(a_d + a_{d-1} \cdot \frac{1}{n} + \cdots + a_0 \cdot \frac{1}{n^d} \right) \cdot n^d$$

und damit

$$|p(n)| \leq \left(\sum_{i=0}^{d} |a_i| \right) \cdot n^d \text{ für alle } n \geq 1,$$

also $p(n) \in \mathcal{O}(n^d)$. Das Polynom lässt sich weiter umschreiben zu

$$\begin{aligned} p(n) &= a_d \cdot \left(1 + \frac{a_{d-1}}{a_d} \cdot \frac{1}{n^1} + \cdots + \frac{a_0}{a_d} \cdot \frac{1}{n^d} \right) \cdot n^d \\ &= a_d \cdot \left[1 + \frac{1}{n} \cdot \left(\frac{a_{d-1}}{a_d} + \frac{a_{d-2}}{a_d} \cdot \frac{1}{n} + \cdots + \frac{a_0}{a_d} \cdot \frac{1}{n^{d-1}} \right) \right] \cdot n^d \end{aligned}$$

sodass

$$|p(n)| \geq |a_d| \cdot n^d \text{ für alle } n > \sum_{i=0}^{d-1} \left| \frac{a_i}{a_d} \right|$$

also $p(n) \in \Sigma(n^d)$ und damit insgesamt

$$p(n) \in \Theta(n^d).$$

bzw. allgemeiner für ein beliebiges Polynom $q : \mathbb{N} \to \mathbb{R}$:

$$q(n) \in \Theta(n^{\deg(q)})$$

Das Wachstum einer durch ein Polynom beschriebenen Zahlenfolge hängt also nur vom Grad des Polynoms ab.

Wir wollen nun einige Aussagen betrachten, die sich direkt aus der obigen Definition ergeben:

Lemma 1.8. *Seien $f, g : \mathbb{N}_0 \to \mathbb{R}$ gegeben. Dann gilt:*

(i) $g \in \mathcal{O}(f) \Leftrightarrow f \in \Omega(g)$
$g \in \Theta(f) \Leftrightarrow f \in \Theta(g)$

(ii) Die Basis eines Logarithmus spielt für das Wachstum keine Rolle:
$\log_b(n) \in \Theta(\log_2(n))$ *für alle* $b > 1$.

(iii) Logarithmen wachsen langsamer als alle Polynome:
$(\log_2(n))^d \in o(n^\epsilon)$ *für alle* $d \in \mathbb{N}_0$ *und jedes* $\epsilon > 0$.

(iv) Exponetielles Wachstum ist immer schneller als polynomielles:
$n^d \in o((1+\epsilon)^n)$ *für alle* $d \in \mathbb{N}_0$ *und jedes* $\epsilon > 0$.

Proposition 1.9. *Seien* $f, g : \mathbb{N}_0 \to \mathbb{R}$ *monoton wachsende Funktionen. Dann gilt*

$$\begin{aligned}
f(n) \in \mathcal{O}(g(n)) &\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
f(n) \in \Omega(g(n)) &\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \\
f(n) \in \Theta(g(n)) &\Leftrightarrow 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
f(n) \in o(g(n)) &\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \\
f(n) \in \omega(g(n)) &\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty
\end{aligned}$$

1.2.1 Master-Theorem

Das *Master-Theorem* bietet eine schnelle Lösung für die Frage, in welcher Laufzeitklasse eine gegebene rekursiv definierte Funktion liegt.

Satz 1.10 (Master-Theorem)**.** *Seien* $a \geq 1$ *und* $b > 1$ *Konstanten. Sei* $f : \mathbb{N} \to \mathbb{R}^+$ *eine Funktion, und sei* $T(n)$ *durch die folgende Rekursion definiert:*

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

wobei wir $\frac{n}{b}$ *wahlweise als* $\left\lceil \frac{n}{b} \right\rceil$ *oder* $\left\lfloor \frac{n}{b} \right\rfloor$ *interpretieren können. Dann gilt:*

1. $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ *für ein* $\epsilon > 0 \Rightarrow T(n) \in \Theta(n^{\log_b a})$
2. $f(n) \in \Theta(n^{\log_b a}) \Rightarrow T(n) \in \Theta(n^{\log_b a} \log n)$
3. $f(n) \in \Omega(n^{\log_b a + \epsilon})$ *für ein* $\epsilon > 0$ *und für ein* c *mit* $0 < c < 1$ *gilt*
   $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) \Rightarrow T(n) \in \Theta(f(n))$

Anmerkung 1.11. *Trifft keiner der drei Fälle zu, so liefert das Master-Theorem keine Aussage und es muss eine andere Methode gewählt werden, um die Laufzeitklasse zu bestimmen.*

Beispiel 1.12. *Es sei die rekursive Funktion* $T : \mathbb{N} \to \mathbb{R}$ *gegeben mit*

$$T(n) = 9T\left(\frac{n}{3}\right) + n.$$

Es ist also $a = 9, b = 3$ *und* $f(n) = n$. *Damit ergibt sich* $\log_3 9 = 2$ *und somit* $n^{\log_3 9} = n^2$. *Offensichtlich ist* $f(n) = n \in \mathcal{O}(n^{\log_3 9 - \epsilon}) = \mathcal{O}(n^{2-\epsilon})$ *für* $\epsilon = 1$. *Es trifft also die Voraussetzung des ersten Falls zu und damit erhält man*

$$T(n) \in \Theta(n^2).$$

Beispiel 1.13. *Es sei nun*

$$T(n) = T\Big(\frac{2}{3}n\Big) + 1$$

gegeben. Es ist also $a = 1, b = \frac{3}{2}$ *und* $f(n) = 1$. *Damit ist* $n^{\log_b a} = n^{\log_{\frac{2}{3}} 1} = n^0 = 1$. *Wir befinden uns also im Fall 2 des Master-Theorems, da* $f(n) \in \Theta(1) = \Theta(n^{\log_b a})$. *Damit erhalten wir*

$$T(n) \in \Theta(\log n)$$

1.2.2 Grenzen der Komplexität

Die oben eingeführten Klassen $\mathcal{O}, \Omega$ und Θ vereinfachen das Vergleichen von Algorithmen ungemein. Man kann relativ schnell erkennen, welcher Algorithmus für ein bestimmtes Problem schneller ist. Es ist jedoch zu beachten, dass insbesondere Vergleiche von Algorithmen, die in der selben Klasse liegen, teilweise nicht besonders aussagekräftig sein können, wie das folgende Beispiel verdeutlicht:

Wie wir im Kapitel 3 noch sehen werden, liegen sowohl QUICKSORT als auch MERGESORT in besten Fall in $\Theta(n \log n)$. QUICKSORT liegt im schlechtesten Fall in $\Theta(n^2)$ – anscheinend weitaus schlechter als MERGESORT, welcher auch im schlechtesten Fall in $\Theta(n \log n)$ liegt.

In der Praxis kommt QUICKSORT jedoch weitaus häufiger zum Einsatz als MERGESORT. Das liegt vor allem daran, dass die in der Definition verwendeten Konstanten c bei letzterem deutlich größer sind als bei QUICKSORT. Zudem ist die Wahrscheinlichkeit, dass der schlechteste Fall Eintritt, zum Einen sehr gering und kann zum Anderen durch geschickte Implementierung sogar noch verringert werden.

2 Grundlegende Datenstrukturen und deren Algorithmen

Algorithmen manipulieren dynamische Mengen von Elementen (Einabe → Ausgabe). Diese Mengen werden durch verschiedene abstrakte Datenstrukturen realisiert. Diese Datenstrukturen unterscheiden sich insbesondere durch Effizienz in Bezug auf Manipulationen (Einfügen, Löschen etc.). Die Wahl der richtigen Datenstruktur hängt also davon ab, welche Operationen der zugrundeliegende Algorithmus verwendet und wie effizient die Datenstruktur diese Operationen umsetzt. Im Folgenden sollen einige grundlegenden Datenstrukturen eingeführt und deren Vor- und Nachteile diskutiert werden.

Die Dantestrukturen sollten also insbesondere nicht für genau einen Algorithmus entworfen werden, sondern wiederverwendbar gestaltet sein und von unnötigen Details abstrahieren. Darüberhinaus sollten sie unabhängig ihrer späteren Implementierung in einer konkreten Programmiersprache spezifiziert werden.

Wir wollen nun zunächst Graphen und Bäume eingeführen, welche die Grundlage für viele der nachfolgend eingeführten Datenstrukturen bilden.

2.1 Graph

Definition 2.1. *Ein Graph G ist ein geordnetes Paar (V, W), wobei V eine Menge von Knoten und $E \subseteq V \times V$ eine Menge von Kanten bezeichnet. Dabei sind Kanten paarweise Verbindungen zwischen Knoten. Diese Kanten können mit sog. Gewichten versehen werden. Grundsätzlich unterscheidet man* gerichtete *und* ungerichtete *Kanten. Sind alle Paare $(v_1, v_2) \in E$ gerichtet, d. h. $(v_1, v_2) \neq (v_2, v_1)$, dann spricht man von einem gerichteten Graphen (*Digraph*). Ein ungerichteter Graph, bei dem jeder Knoten mit allen anderen Knoten verbunden ist, heißt* vollständig.

Beispiel 2.2. *Abbildung 1 zeigt ein Beispiel für einen Digraphen mit $V = \{1, 2, 3, 4, 5, 6\}$ und den Kanten*

$$E = \{(1,4), (2,1), (2,3), (3,6), (4,2), (4,3), (4,5), (5,1), (5,3), (5,6), (6,4)\}.$$

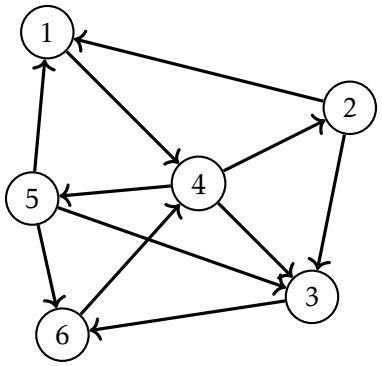

Abbildung 1: Beispiel für einen Digraphen

Proposition 2.3. *Sei $G = (E, V)$ ein Graph. Dann gilt:*

(i) Für die Anzahl von Kanten in einem Digraphen ohne Schleifen und Mehrfachkanten gilt

$$|E| \leq |V|(|V| - 1).$$

(ii) Für die Anzahl von Kanten in einem ungerichteten Graphen ohne Schleifen und Mehrfachkanten gilt

$$|E| \leq \frac{1}{2}|V|(|V| - 1).$$

In vollständigen Graphen gilt jeweils Gleichheit.

Beweis (nur für (i)). Jeder Knoten hat höchstens alle Knoten außer sich selbst als Nachbar, er hat also $|V|-1$ Nachbarn. Da dies für jeden Knoten des Graphen möglich ist folgt die Behauptung. □

Eine weitere Möglichkeit Graphen anzugeben sind sog. *Adjazenzlisten* oder *Adjazenzmatrizen*. Hierbei listet man für die jeweiligen Knoten alle Nachbarn auf (Adjazenzliste) bzw. setzt in einer Matrix den Eintrag a_{ij} auf 1, falls es eine Kante vom Knoten v_1 nach v_2 gibt, bzw. allgemein:

Definition 2.4 (Adjazenzmatrix)**.** *Eine Adjazenzmatrix eines Graphen $G = (V, E)$ ist eine $|V| \times |V|$-Matrix $A = (a_{ij})$ mit*

$$a_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{sonst.} \end{cases}$$

Anmerkung 2.5. *Offensichtlich sind Adjazenzmatrizen von ungerichteten Graphen symmetrisch.*

Für den im Beispiel 2.2 definierten Graph sieht die Adjazenzmatrix folgendermaßen aus:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Für gewichtete Graphen eignet sich die Darstellung in Form von Adjazenzmatrizen auch: hierbei kann anstatt des Wertes 1 in der Adjazenzmatrix einfach das Gewicht der jeweiligen Kante eingetragen werden.

Für gerichtete Graphen wollen wir nun weitere Begriffe einführen:

Definition 2.6. *Sei $G = (V, E)$ ein gerichteter Graph. Dann definieren wir*

- *die eingehende Nachbarmenge von $v \in V$:*

$$N^+(v) = \{u \in V \mid (u,v) \in E\}$$

- *die ausgehende Nachbarmenge von $v \in V$:*

$$N^-(v) = \{w \in V \mid (v,w) \in E\}$$

- *den Eingangsgrad von $v \in V$:*

$$deg_{in}(v) = \left|N^+(v)\right|$$

- *den Ausgangsgrad von $v \in V$:*

$$deg_{out}(v) = \left|N^{-}(v)\right|$$

Setzt man Graphen als Datenstruktur ein, so ist es häufig das Ziel, den (kürzesten) Weg zwischen zwei Knoten zu finden. Dies führt uns zur nächsten

Definition 2.7 (Pfade)**.** *Es sei $G = (V, E)$ ein Graph. Ein Pfad ist eine Folge von paarweise verschiedenen Knoten $u_1, \ldots, u_k \in V$, wobei $(u_i, u_{i+1}) \in E$ für alle $i \in \{1, \ldots, k-1\}$.*

Die Länge eines Pfades ist die Anzahl der Kanten (falls die Kanten von G nicht gewichtet sind) bzw. die Summe der Kantengewichte. Dabei sei $d(u,v)$ die Länge des kürzesten Pfades von u nach v. Der Durchmesser D sei definiert durch $D = \max_{u,v \in V} d(u,v)$.

Ist $v_1 = v_k$, so heißt der Pfad $(v_1, \ldots, v_k)$ Zyklus [1]. Gilt zusätzlich $v_i \neq v_j$ für $i, j \in \{1, \ldots, k-1\}$ mit $i \neq j$, dann heißt der Pfad Kreis.

Definition 2.8. *Es sei $G = (V, E)$ ein Graph mit $|V| \geq 1$. G heißt* zusammenhängend, *falls für jedes Knotenpaar (u, v) mit $u, v \in E$ ein Weg von u nach v in G exisitert.*

2.1.1 Breitensuche (Breadth-First-Search)

Bei der Breitensuche handelt es sich um ein Verfahren zum Durchsuchen (oder nur Durchlaufen) der Knoten eines Graphen. Hierbei werden zunächst alle Knoten beschritten, die vom Ausgangsknoten direkt erreichbar sind. Erst danach werden Folgeknoten beschritten. Dazu verwendet man in der Regel eine Warteschlange (vgl. Abschnitt *Queue*).

Informell lässt sich die Breitensuche folgendermaßen beschreiben:

1. Bestimme den Knoten, an dem die Suche beginnen soll, markiere ihn als besucht und füge ihn der Warteschlange hinzu
2. Entnimm einen Knoten vom Beginn der Warteschlange
   - Falls der entnommene Knoten der gesuchte Knoten ist, brich die Suche ab (und gebe den Knoten oder "gefunden" zurück)
   - Andernfalls hänge alle unmarkierten Nachfolge diese Knoten ans Ende der Warteschlange an und markiere sie als besucht
3. Prüfe ob die Warteschlange leer ist. Falls ja, beende die Suche (gesuchtes Element wurde nicht gefunden)
4. Wiederhole Schritt 2.

[1] Hierbei wird allerding i. d. R. nicht verlangt, dass $(v_1, \ldots, v_k)$ ein Pfad ist – es genügt dass $(v_1, \ldots, v_k)$ ein Weg ist (die Knoten müssen also nicht paarweise verschieden sein).

Als Pseudocode könnte die Breitensuche folgendermaßen aussehen:

Algorithmus 1 Breitensuche mit Startknoten *start* und gesuchtem Knoten *goal*

```
procedure BFS(start, goal)
    for all node i do                          ▷ Anfangs sind keine Knoten besucht
        visited[i] ← false
    end for
    queue.PUSH(start)
    visited[start] ← true
    while not queue.EMPTY do
        node ← queue.POP( )
        if node is goal then
            return true
        end if
        for child in neighbors[node] do
            if not visited[child] then
                queue.PUSH(child)
                visited[child] ← true
            end if
        end for
    end while
    return false
end procedure
```

Proposition 2.9 (Laufzeit)**.** *Im schlechtesten Fall müssen alle möglichen Pfade zu allen möglichen Knoten betrachtet werden, daher gilt für die Laufzeit der Breitensuche*

- *Falls Adjazenzlisten benutzt werden:* $\mathcal{O}(|V| + |E|)$
- *Falls Adjazenzmatrizen benutzt werden:* $\mathcal{O}(|V|^2)$.

Beispiel 2.10. *Interpretiert man einen Graphen als Baum und definiert die Reihenfolge der Nachfolger eines Knotens von links nach rechts, so würde man den Baum in Abbildung 2 Ebene für Ebene und innerhalb der Ebenen von links nach rechts durchlaufen und erhielte dann die Reihenfolge*

$$17 \to 4 \to 11 \to 18 \to 3 \to 9 \to 13.$$

2.1.2 Tiefensuche (Depth-First-Search)

Wie auch die Breitensuche ist die *Tiefensuche* ein Verfahren zum Suchen von Knoten in einem Graphen. Im Gegensatz zur Breitensuche wird ein Pfad zunächst vollständig in die Tiefe beschritten, bevor abzweigende Pfade beschritten werden.
Informell lässt sich die Tiefensuche folgendermaßen beschreiben:

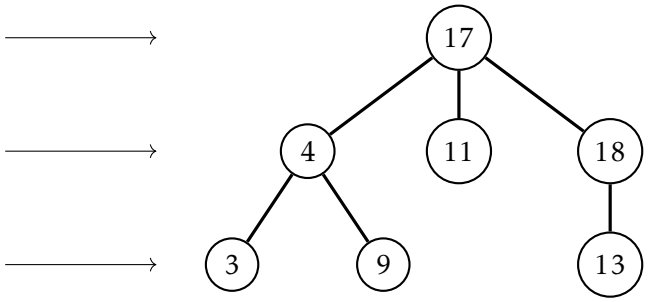

Abbildung 2: Breitensuche in einem Baum

1. Bestimme den Startknoten
2. Bestimme alle Nachfolger bzw. Nachbarn des Knotens und speichere alle noch nicht erschlossenen Nachfolger in einem Stack
3. Rufe rekursiv für jeden Knotne in dem Stack die Tiefensuche auf
   - Falls das gesuchte Element gefunden wurde breche ab
   - Falls es keine nicht erschlossenen Nachfolger mehr gibt, lösche den obersten Knoten aus dem Stack und rufe für den jetzt oberen Knoten im Stack die Tiefensuche auf

In Pseudocode könnte die Tiefensuche folgendermaßen aussehen:

Algorithmus 2 Tiefensuche mit Startknoten *start* und gesuchtem Knoten *goal*

```
procedure DFS(start, goal)
    if start is goal then
        return start
    end if
    stack ← NEIGHBORS(start)
    while stack is not empty do
        node ← POP(stack)
        DFS(node, goal)
    end while
end procedure
```

Beispiel 2.11. *Mit dem selben Baum wie in Abbildung 2 erhält man nun mit der Tiefensuche (siehe Abb. 3) die Reihenfolge*

$$17 \to 4 \to 3 \to 9 \to 11 \to 18 \to 13$$

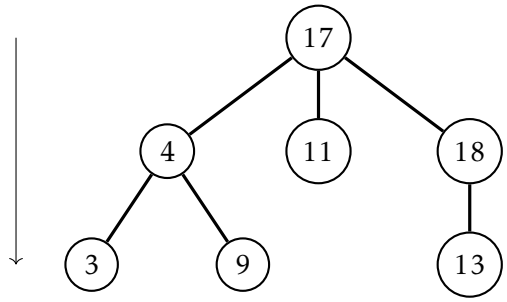

Abbildung 3: Tiefensuche in einem Baum

2.2 Bäume

Ein Baum ist ein spezieller Typ von Graph, der zusammenhängend ist und keine geschlossenen Pfade (Kreise) enthält. Die Knoten mit Grad 1 heißen Blätter (sie haben insbesondere Ausgangsgrad 0). Beispiele für Bäume haben wir bereits in Abbildung 2 und 3 gesehen. In der Regel betrachten wir nur gerichtete Bäume, die einen ausgezeichneten Startknoten besitzen, die sog. Wurzel. Die Wurzel zeichnet sich dadurch aus, dass sie der einzige Knoten mit Eingangsgrad 0 ist.

Man erkennt leicht, dass es zwischen zwei Knoten eines Baums nur genau einen Pfad geben kann. Außerdem gilt für die Anzahl der Kanten eines Baums $G = (E, V)$:

$$|E| = |V| - 1,$$

da es zu jedem Knoten eine Kante gibt, außer zur Wurzel.

Im Folgenden betrachten wir hauptsächlich *Binärnäume*, welche sich dadurch auszeichnen, dass jeder Knoten eines Binärbaums nur höchstens zwei direkte Nachkommen (*Kinder*) haben kann. Als *Tiefe* eines Knotens bezeichnen wir die Anzahl der Kanten bis zur Wurzel. Dabei ist die Höhe des Baumes die maximal auftretende Tiefe.[2] Haben alle Blätter die gleiche Tiefe, so heißt der Baum vollständig.

2.2.1 Suche in Bäumen

Wir unterscheiden zunächst die folgenden drei Varianten, um einen Baum zu durchlaufen:

Definition 2.12. *Es sei T ein geordneter Baum mit Wurzel r und Teilbäumen $T_1, \ldots, T_m$.*

(a) Wenn T in Postorder *durchlaufen wird, dann werden (rekursiv) die Teilbäume $T_1, \ldots, T_m$ nacheinander durchlaufen und danach wird die Wurzel r besucht.*

(b) Wenn T in Präorder *durchlaufen wird, dann wird zuerst r besucht und dann werden die Teilbäume $T_1, \ldots, T_m$ (rekursiv) durchlaufen.*

[2] Oft definiert man jedoch die Höhe des leerens Baums als 0, muss also zur Anzahl der Kanten noch 1 addieren.

(c) Wenn T in Inorder *durchlaufen wird, wird zuerst T_1 (rekursiv) durchlaufen, sodann wird die Wurzel r besucht und letzlich werden die Teilbäume $T_2, \ldots, T_m$ (rekursiv) durchlaufen.*

2.2.2 Binärer Suchbaum

Ein *binärer Suchbaum* ist ein binärer Baum, bei dem den Knoten Schlüssel zugewiesen werden und die Schlüssel des linken Teilbaums eines Knotens nur kleiner (oder gleich) und die des rechten nur größer (oder gleich) als der Schlüssel des Knotens selbst sind. Sie lösen (wie auch andere Suchbäume) das sog. *Wörterbuchproblem*. Angenommen ist eine große Anzahl von Schlüsseln, denen jeweils ein Wert beigegeben ist. In einem Wörterbuch deutsch–englisch ist das deutsche Wort der Schlüssel und englische Wörter sind der gesuchte Wert. Ähnlich verhält es sich bei einem Telefonbuch mit Namen und Adresse als Schlüssel und der Telefonnummer als dem gesuchten Wert.

In beiden Beispielen sind die Schlüssel üblicherweise sortiert, was es ermöglicht das Suchen nach Schlüsseln deutlich zu beschleunigen: Man schlägt das Buch zunächst in der Mitta auf und überprüft, ob der Schlüssel gefunden wurde. Falls er nicht gefunden wurde, der gesuchte Schlüssel aber kleiner ist, als der betrachtete, kann man nun die hintere Hälfte des Buches schon ausschließen. So kann man nun rekursiv mit der entsprechenden anderen Hälfte fortfahren und gelangt so bei n Schlüsseln mit maximal $\lceil \log_2(n+1) \rceil$ Vergleichen zum Ziel.[3]

Einen Binärbaum würde man so aufbauen, dass zuerst ein Element als Wurzel eingefügt wird. Wird ein neues Element eingefügt, dessen Schlüssel kleiner ist als der der Wurzel, wird es in den linken Teilbaum eingefügt (rekursiv mit dem linken Kind als neue Wurzel, solange bis man es als Blatt einfügen kann). Analog verfährt man für Elemente mit größerem Schlüssel.

Hierbei kann es natürlich vorkommen, dass der Baum nicht balanciert ist oder im Extremfall sogar zur lineare Liste wird (wenn man immer nur größere bzw. immer nur kleinere Elemente einfügt). Aus dieser Problematik heraus sind einige Varianten des Binären Suchbaums entstanden, die die Baumstruktur beim Einfügen oder Löschen so verändern, dass der Baum balanciert wird oder die Struktur zumindest praktischer wird.

2.3 Hashing und Hashtabellen

Hashtabellen sind eine weitere Datenstruktur, die wie jede andere Datenstruktur (Arrays, Heaps, Bäume) auch Stärken und Schwächen hat. Hashtabellen eignen sich besonders, so lange wir nur eine Menge von Daten benötigen, in die wir schnell einfügen, löschen und in der wir suchen können, sind Hashtabellen meist sehr effizient.[4]

Die zugrundeliegende Idee ist es, aus den zu speichernden Elementen selbst deren Position in einer Tabelle zu schließen. Dann müsste man beim Suchen einen Elements

[3] Dieses Verfahren nennt man *Binäre Suche*

[4] Dabei hängt die Effizienz der Hashtabelle maßgeblich von der Qualität der verwendeten Hashfunktion ab, die für die Art der verwendeten Daten geeignet sein muss.

nur dessen Position berechnen und könnte direkt an die richige Stelle in der Tabelle springen.

Definition 2.13. *Eine* Hashtabelle *ist eine Datenstruktur, welche mit Hilfe einer Funktion* $h : U \to D$ *– die sog.* Hashfunktion *– die Position berechnet, an der der Schlüssel* $u \in U$ *im Elementebereich (also in der Hashtabelle)* $D \subset \mathbb{N}$ *gespeichert werden soll. Der Elementbereich* D *ist dabei eine natürliche Zahl in* $\{0, 1, \ldots, m-1\}$*, wobei* m *die Länge der Hashtabelle ist.*

Die Hashfunktion berechnet also für ein gegebenes Element dessen Position in einer Tabelle. Sucht man nun ein Element, oder will ein neues Element einfügen, kann man mit Hilfe der Hashfunktion die Position des Elements berechnen und kann dann in konstanter Zeit ($\mathcal{O}(1)$) direkt an die entsprechende Stelle in der Tabelle springen. Hashtabellen sind also insbesondere nur geeignet, wenn aus den zu speichernden Elementen einfach eindeutige Schlüssel generiert werden können.
Gute Hashfunktion sollten

- surjektiv sein, d.h. die ganze Tabelle ausnutzen
- die Schlüssel möglichst gleichmäßig verteilen

Anmerkung 2.14. *In der Regel gilt* $|U| > |D|$*, es gibt also mehr Schlüssel als Plätze in der Hashtabelle.*[5] *Da die Hashfunktion i. A. auch nicht injektiv ist, kann es sein, dass verschiedene Elemente (bzw. verschiedene Schlüssel) auf denselben Index (Hashwert) abgebildet werden. Dies bezeichnet man als* ***Kollision****. Es müssen also Strategien gefunden werden, die solche Kollisionen erkennen und behandeln.*

2.3.1 Hashverfahren mit Verkettung

Wir wollen in den Folgenden Abschnitten einige Möglichkeiten diskutieren, Kollisionen zu behandeln. Eine intuitive Möglichkeit ist es, bei einer Kollision einfach mehrere Elemente an der selben Stelle in der Hashtabelle zu speichern, indem man für jeden Eintrag der Hashtabelle eine Liste speichert, der man dann die kollidierenden Elemente hinzufügt.

2.3.2 Hashtabelle mit Sondierung

ToDo

2.4 Heap

Ein *Heap* (deutsch: Haufen) ist eine auf Bäumen basierende Datenstruktur.

[5] Die Schlüsselmenge zu verkleinern ist ja gerade einer der Gründe, Hashtabellen zu benutzen.

3 Sortieralgorithmen

Bevor verschiedene Sortieralgorithmen diskutiert werden, wollen wir zunächst einige Begriffe einführen, die Sortieralgorithmen charakterisieren und eine Möglichkeit eingeführen, die benutzt wird , um die Korrektheit eines (Sortier-)Algorithmus zu beweisen:

Definition 3.1. *Ein Sortierverfahren heißt* stabil, *falls es die relative relative Reihenfolge gleicher Elemente durch das Sortieren nicht verändert.*

Definition 3.2. *Ein (Sortier-)Algorithmus arbeitet* in-place *(manchmal auch:* in-situ*), wenn er außer dem von der Eingabe benötigten Speicherplatz nur eine konstante, von der Eingabelänge unabhängige Menge von Speicher benötigt. Arbeitet ein Algorithmus nicht in-place, dann arbeitet er* out-of-place.

3.1 Schleifeninvarianten

Eine Invariante ist eine Aussage, die über die Ausführung bestimmter Programmbefehle hinweg gilt. Für eine **Schleifeninvariante** gilt hierbei, dass sie

1. *unittelbar vor Eintritt* in eine Schleife und
2. *am Ende* des Schleifenrumpfes wieder

gilt. Die Schleifenvariante muss außerdem eine *sinnvolle* Eigenschaft der Schleife wiedergeben. Typischerweise beschreiben Schleifeninvarianten Wertebereiche von Variablen oder Beziehungen der Variablen untereinander.

Beispiel 3.3. *Der folgende Algorithmus multipliziert die beiden Variablen a und b miteinander:*

Algorithmus 3 Beispiel für Algorithmus mit Schleifeninvariante $(x \cdot y) + p = a \cdot b$

```
 1: procedure MULTIPLY(a, b)
 2:     x ← a
 3:     y ← b
 4:     p ← 0                          ▷ Die Invariante muss vor der Schleife gelten
 5:     while x > 0 do
 6:                                    ▷ Die Invariante muss am Anfang jedes Durchlaufs gelten
 7:         p ← p + y
 8:         x ← x − 1
 9:                                    ▷ Die Invariante muss am Ende jedes Durchlaufs gelten
10:     end while
11:                                    ▷ Die Invariante muss auch direkt nach der Schleife gelten
12:     return p
13: end procedure
```

Anmerkung: *Zwischen Zeile 7 und 8 muss die Schleifeninvariante nicht gelten.*

3.2 Vergleichsbasierte Sortieralgorithmen

3.2.1 Insertionsort

Dieser Abschnitt behandelt das Sortierverfahren Insertionsort (Sortieren durch Einfügen). Die Idee des Algorithmus ist, die typische *menschliche* Vorgehensweise – bspw. beim Sortieren eines Stapels von Karten – umzusetzen. D. h. es wird mit der ersten Karte ein neuer Stapel gestartet. Anschließend nimmt man jeweils die nächste Karte des Originalstapels und fügt diese an der richtigen Stelle im neuen Stapel ein.

Anmerkung 3.4. *Insertionsort arbeitet in-place und ist stabil.*

Algorithmus 4 Insertionsort

```
for i ← 1 to len(A) do
    key ← A[i]
    j ← i
    while j > 0 & A[j − 1] > key do
        A[j] ← A[j − 1]
        j ← j − 1
    end while
    A[j] ← key
end for
```

Proposition 3.5 (Laufzeit)**.** *Insertionsort liegt im besten Fall in* $\Theta(n)$ *und im durchschnittlichen und schlechtesten Fall* $\mathcal{O}(n^2)$*.*

Beweis. Im **besten Fall** ist die Liste schon sortiert. Die Anzahl der Vergleiche ist gleich der Anzahl der Schleifendurchläufe: $n-1$. Bei jedem Rückweg zur Einfügeposition nimmt man den Faktor 1. Somit beträgt die Gesamtzahl der Vergleiche: $(n-1)\cdot 1 = n-1$. Für große Listen lässt ich abschätzen: $n-1 \approx n$, also haben wir linearen Aufwand.

Im **mittleren Fall** ist die Liste unsortiert. Die Einfügeposition befindet sich wahrscheinlich auf der Hälfte des Rückwegs. Bei jedem der $n-1$ Rückwege muss ein $\frac{i-1}{2}$ Vergleich addiert werden. Die Gesamtzahl der Vergleiche beträgt dann:

$$
\begin{aligned}
&\frac{n-1}{2} + \frac{n-2}{2} + \cdots + \frac{2}{2} + \frac{1}{2} \\
&= \frac{(n-1) + (n-2) + \cdots + 2 + 1}{2} \\
&= \frac{1}{2} \cdot \frac{n \cdot (n-1)}{2} \\
&= \frac{n \cdot (n-1)}{2} \\
&\approx \frac{n^2}{4}
\end{aligned}
$$

Daraus ergibt sich ein quadratischer Aufwand.

Im **schlechtesten Fall** ist die Liste absteigend sortiert. Bei jedem der $n-1$ Rückwege müssen $i-1$ Elemente verglichen werden. Analog zu vorhergehenden Überlegungen, gibt es hier aber doppelte Rückwegelänge. Daraus ergibt sich die Gesamtzahl der Vergleiche

$$(n-1)+(n-2)+\cdots+2+1 = \frac{n\cdot(n-1)}{2} \approx \frac{n^2}{2}.$$

Daraus ergibt sich wiederum quadratischer Aufwand. □

3.2.2 Bubblesort

Bubblesort ist ein weitere vergleichsbasierter Sortieralgorithmus. Hierbei wird die zu sortierende Liste durchlaufen bis zwei Werte falsch sortiert sind. Dies wird korrigiert und das Ganze wird wiederholt bis das Feld vollständig sortiert ist.

Hierbei wird das Feld von hinten nach vorne sortiert (zuerst steht das größte Element ganze hinten, dann das zweitgrößte an der vorletzten Stelle etc.).

Algorithmus 5 Bubblesort

```
for i ← len(A) − 1 to 0 do
    for j ← 0 to i do
        if A[j] > A[j + 1] then
            swap(A[j], A[j + 1])
        end if
    end for
end for
```

Proposition 3.6 (Laufzeit)**.** *Im schlechtesten und im durchschnittlichen Fall gilt für die Laufzeit (mit $n = len(A)$):*

$$T(n) \in \sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}\Big(\sum_{i=1}^{n} i\Big) = \mathcal{O}\Big(\frac{n(n+1)}{2}\Big) = \mathcal{O}(n^2).$$

Im besten Fall ist die Liste bereits sortiert und es muss nichts getauscht werden. Dann ist Bubblesort in $\mathcal{O}(n)$.

3.2.3 Quicksort

Quicksort ist ein schneller, rekursiver Sortieralgorithmus, der nach dem Prinzip *divide and conquer* arbeitet. Hierbei wird zunächst die sortierende Liste in zwei Teillisten getrennt. Dazu wird ein sog. Pivotelement aus der Liste ausgewählt. Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Teilliste und alle Elemente, die größer sind, in die rechte Teilliste. Insbesondere sind also nach der Aufteilung die Elemente der linken Liste kleiner oder gleich den Elementen der rechten Liste.

Anschließend muss noch jede Teilliste in sich sortiert werden, um die Sortierung zu beenden. Dazu wird der Quicksort-Algorithmus rekursiv auf der linken und rechten Teilliste ausgeführt. Die Rekursion bricht ab, falls die Länge der Teilliste kleiner oder gleich 1 ist.

Algorithmus 6 Quicksort mit Hilfsfunktion PARTITION

```
procedure QUICKSORT(A, low, high)
    if low < high then
        piv ← PARTITION(A, low, high)        ▷ piv ist der Index des Pivotelements,
                                             ▷ d. h. A[piv] steht jetzt an der richtigen Stelle
        QUICKSORT(A, low, piv − 1)           ▷ Linke Teilliste sortieren
        QUICKSORT(A, piv + 1, high)          ▷ Rechte Teilliste sortieren
    end if
end procedure
```

Die Hilfsfunktion PARTITION kann folgendermaßen implementiert werden: Man nimmt das letzte Element der Liste als Pivotelement und verschiebt es innerhalb der Liste an die richtige Stelle. Alle Elemente, die kleiner als das Pivotelement sind, werden links davon platziert, alle Elemente die größer sind, rechts davon.

Algorithmus 7 Hilfsfunktion PARTITION

```
procedure PARTITION(A, low, high)
    pivot ← A[high]
    i ← low − 1
    for j ← low to high do
        if A[j] ≤ pivot then                 ▷ Falls aktuelles Element ≤ Pivot
            i ← i + 1                        ▷ Index des kleineren Elements erhöhen
            SWAP(A[i], A[j])
        end if
    end for
    SWAP(A[i + 1], A[high])
    return i + 1
end procedure
```

Proposition 3.7 (Laufzeit)**.** *Im schlechtesten Fall liegt QuickSort in $\mathcal{O}(n^2)$. Im besten Fall und im durchschnittlichen Fall gilt für die asymptotische Laufzeit $\mathcal{O}(n \log n)$.*

Beweis. Im Worst Case das Pivotelement stets so gewählt, dass es das größte oder kleinste Element der Liste ist. Die zu untersuchende Liste wird dann in jedem Rekursionsschritt nur um eins kleiner.

Wird das Pivot-Element ideal gewählt, so wird die Liste in jedem Schritt in zwei Teillisten halbiert. Diese Listen werden wiederum halbiert etc. Es ergeben sich also im 2. Schritt zwei Listen der Länge $\frac{n}{2}$, im 3. Schritt vier Listen der Länge $\frac{n}{4}$ und damit im

$\log n$-ten Schritt 1-elementige Listen. Für das Vertauschen der Schlüsselelemente sind in jedem Schritt n Vergleiche notwendig, so dass sich insgesamt

$$T(n) = n \cdot \log n$$

Vergleiche ergeben. □

3.2.4 Mergesort

Wie Quicksort funktioniert auch Mergesort nach dem Prinzip *Divide and Conquer*.

Anmerkung 3.8. *Mergesort ist stabil, aber in der Regel nicht in-place.*

Algorithmus 8 Mergesort unter Verwendung der Hilfsfunktion MERGE

```
procedure MERGESORT(A)
    if |A| ≤ 1 then
        return A
    end if
    middle ← ⌊|A|/2⌋

    for x in A up to middle do          ▷ Links der Mitte (inkl. Mitte) in linke Teilliste
        left.APPEND(x)
    end for
    for x in A after middle do          ▷ Rechts der Mitte (ohne Mitte) in rechte Teilliste
        right.APPEND(x)
    end for

    left ← MERGESORT(left)
    right ← MERGESORT(right)

    return MERGE(left, right)
end procedure
```

Die Funktion MERGE kann hierbei auf verschiedene Arten implementiert werden, eine einfache Varaiante könnte hierbei folgendermaßen aussehen:

Algorithmus 9 Hilfsfunktion MERGE

```
procedure MERGE(left, right)
    result ← []
    while |left| > 0 and |right| > 0 do
        if HEAD(left) ≤ HEAD(right) then
            result.APPEND( HEAD(left) )
            left ← TAIL(left)
        else
            result.APPEND( HEAD(right) )
            right ← TAIL(right)
        end if
        if |left| > 0 then
            result.APPEND(left)
        end if
        if |right| > 0 then
            result.APPEND(right)
        end if
    end while
end procedure
```

3.3 Nicht-Vergleichsbasierte Sortieralgorithmen

3.3.1 Countingsort

Countingsort ist das erste Beispiel für einen nicht-vergleichsbasierten Sortieralgorithmus. Allerdings ist Countingsort nur einsetzbar, falls die zu sortierenden Daten natürliche Zahlen innerhalb eines bekannten Intervalls sind (bzw. die Daten in ein solches Intervall kodiert werden können).

Countingsort benötigt jedoch ein Hilfsarray, das so groß ist wie die Spannweite des Intervalls (für ein Intervall $[n, m] \subsetneq \mathbb{N}$ benötigt man ein Hilfsarray der Länge $m - n + 1$). Dieser Algorithmus ist also nur für relativ kleine Intervallgrößen geeignet (bspw. das Alter von Einwohnern eines Landes: $[0, 150]$).

Der Algorithmus zählt, wie oft jeder der Eingabewerte vorkommt und speichert die jeweilige Anzahl in einem Hilfsarray. Mit Hilfe dieses Arrays wird für jedes Element der Eingabe die Zielposition in der Ausgabe berechnet.

Proposition 3.9 (Laufzeit)**.** *Die Laufzeit von Countingsort ist in jedem Fall $\mathcal{O}(n \cdot k)$ für n die Länge der Eingabe und k die Größe des verwendeten Intervalls.*

3.3.2 Radixsort

Radixsort ein weiterer nicht-vergleichsbasierter Sortieralgorithmus. Er basiert auf Countingsort, bzw. verwendet Countingsort in den einzelnen Schritten. Wie Countingsort ist auch Radixsort nur für das Sortieren Zahlen geeignet (oder geeignete Kodierungen).

Algorithmus 10 Countingsort für ein Intervall $[0, k]$

```
procedure COUNTINGSORT(A, k)
    C ← ARRAY(0, k)                          ▷ C sollte mit Nullen gefüllt werden
    B ← ARRAY(1, |A|)                        ▷ Sortierte Liste
    for j ← 1 to |A| do
        C[A[j]] ← C[A[j]] + 1                ▷ Schreibe in C[m] wie häufig m in A vorkommt
    end for
    for m ← 1 to k do
        C[m] ← C[m] + C[m − 1]               ▷ Adressrechnung
    end for
    for j ← |A| to 1 do
        B[C[A[j]]] ← A[j]                    ▷ Kopiere auf jeweilige Zieladresse in B
        C[A[j]] ← C[A[j]] − 1                ▷ Dekrementiere Zieladresse (falls gleiche Werte in A)
    end for
    return B
end procedure
```

Zunächst werden die Elemente aufsteigend nach ihrer letzten Stelle[6] sortiert (mittels Countingsort). Im Anschluss wird aufsteigend nach der zweitletzten Stelle sortiert (wieder mit Countingsort). Dieses Vorgehen wird für alle Stellen wiederholt. Da Countingsort stabil ist, werden bereits sortierte Zahlen nicht wieder vertauscht, falls sie in einer Stelle übereinstimmen.

[6] Also der niederwertigsten Stelle, bspw. für 234 die Stelle 4