Sortieralgorithmen im Vergleich

Prinzipien und asymptotische Laufzeitanalyse

Vorlesung Algorithmen und Datenstrukturen

25. März 2026

Inhaltsverzeichnis

Selection Sort (Sortieren durch Auswählen)

Das Prinzip:

- Sucht in jedem Durchlauf das kleinste Element aus dem unsortierten Teil.
- Tauscht dieses Element an die nächste freie Stelle im sortierten Teil (ganz links).
- Benötigt in der Summe immer $\frac{n(n-1)}{2}$ Vergleiche.

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n^2)$ – Der Algorithmus ist „blind“ für Vorsortierungen.
- **Average Case:** $\mathcal{O}(n^2)$
- **Worst Case:** $\mathcal{O}(n^2)$

Beweis der Laufzeit: Selection Sort

- **Anzahl der Vergleiche:** Im 1. Durchlauf $n-1$, im 2. Durchlauf $n-2$, ..., im letzten Durchlauf 1.
- **Summe:** Die Gesamtzahl der Vergleiche entspricht der arithmetischen Reihe:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

- **Asymptotik:** Für sehr große n dominiert der quadratische Term n^2. Die Konstanten fallen weg.
- **Fazit:** Da der Algorithmus die innere Schleife immer komplett durchläuft (keine Abkürzung bei bereits sortierten Daten), gilt $\mathcal{O}(n^2)$ für Best, Average und Worst Case.

Bubble Sort (Sortieren durch Aufsteigen)

Das Prinzip:

- Vergleicht benachbarte Elemente und tauscht sie bei falscher Reihenfolge.
- Große Elemente „blubbern“ wie Luftblasen ans rechte Ende der Liste.
- Ein Durchlauf ohne Tausch signalisiert, dass die Liste sortiert ist.

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n)$ – Liste ist bereits sortiert (Abbruch nach 1. Durchlauf).
- **Average Case:** $\mathcal{O}(n^2)$
- **Worst Case:** $\mathcal{O}(n^2)$ – Liste ist absteigend sortiert.

Beweis der Laufzeit: Bubble Sort

- **Worst Case (absteigend sortiert):** Das kleinste Element wandert pro Durchlauf nur um eine Position nach links. Es werden $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ Vergleiche benötigt $\implies \mathcal{O}(n^2)$.
- **Best Case (aufsteigend sortiert):** Im ersten Durchlauf ($n-1$ Vergleiche) findet kein Tausch statt. Die Abbruchbedingung greift sofort $\implies \mathcal{O}(n)$.
- **Average Case:** In einer zufälligen Liste gibt es im Schnitt $\frac{n(n-1)}{4}$ Fehlstände (Inversionen). Da jeder Tausch genau einen Fehlstand behebt, bleibt das asymptotische Wachstum quadratisch $\implies \mathcal{O}(n^2)$.

Insertion Sort (Sortieren durch Einfügen)

Das Prinzip:

- Funktioniert wie das Sortieren von Spielkarten auf der Hand.
- Nimmt das nächste unsortierte Element und fügt es an der exakt richtigen Stelle im sortierten linken Teil ein.

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n)$ – Liste ist aufsteigend sortiert (nur ein Vergleich pro Element).
- **Average Case:** $\mathcal{O}(n^2)$
- **Worst Case:** $\mathcal{O}(n^2)$

Beweis der Laufzeit: Insertion Sort

- **Worst Case (absteigend sortiert):** Jedes Element muss durch den gesamten sortierten Bereich nach links verschoben werden. Aufwand: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \implies \mathcal{O}(n^2)$.
- **Best Case (aufsteigend sortiert):** Jedes der $n-1$ unsortierten Elemente wird nur ein einziges Mal mit seinem linken Nachbarn verglichen und sofort an Ort und Stelle belassen. Vergleiche: $n-1 \implies \mathcal{O}(n)$.
- **Average Case:** Im Schnitt muss ein Element nur mit der Hälfte der bereits sortierten Elemente verglichen werden: $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4} \implies \mathcal{O}(n^2)$.

Exkurs: Binary Insertion Sort

Die Idee (Vergleiche reduzieren):

- Da der linke Teil der Liste bei Insertion Sort bereits sortiert ist, können wir die **Binäre Suche (Binary Search)** nutzen.
- Anstatt linear von rechts nach links zu vergleichen, wird das Suchintervall halbiert.
- Die Anzahl der reinen Vergleiche sinkt drastisch von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \log n)$.

Der Haken (Das Verschieben im Speicher):

- Arrays liegen als durchgehender Block im Speicher.
- Um das Element an der gefundenen Position einzufügen, müssen alle nachfolgenden Elemente physisch um eine Position nach rechts verschoben werden (Shiften).
- Dieses „Shiften“ benötigt weiterhin $\mathcal{O}(n)$ pro Element.
- **Fazit:** Die asymptotische Gesamtlaufzeit bleibt bei $\mathcal{O}(n^2)$, auch wenn der Algorithmus in der Praxis durch weniger Vergleiche merklich schneller wird.

Merge Sort (Sortieren durch Verschmelzen)

Das Prinzip (Divide & Conquer):

- Teilt die Liste stur immer wieder in zwei Hälften auf, bis Teillisten der Länge 1 entstehen.
- Verschmilzt (merges) die Teillisten von unten nach oben wieder zu sortierten Listen.
- Rekursionsbaum hat $\log_2(n)$ Ebenen mit linearem Aufwand $\mathcal{O}(n)$ pro Ebene.

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n \log n)$
- **Average Case:** $\mathcal{O}(n \log n)$
- **Worst Case:** $\mathcal{O}(n \log n)$

Beweis der Laufzeit: Merge Sort

- **Tiefe des Rekursionsbaums:** Eine Liste der Länge n kann genau $\log_2(n)$-mal halbiert werden, bis die Teillisten die Länge 1 haben.
- **Aufwand pro Ebene:** Auf jeder Rekursionsebene müssen die Teil-Listen wieder verschmolzen werden. Das Verschmelzen zweier sortierter Listen erfordert maximal so viele Vergleiche, wie insgesamt Elemente enthalten sind. Der Gesamtaufwand pro Ebene liegt bei $\mathcal{O}(n)$.
- **Gesamtaufwand:** Arbeit pro Ebene $\times$ Anzahl der Ebenen $= n \cdot \log_2(n) \implies \mathcal{O}(n \log n)$.
- Da die Listen immer halbiert werden, ist das Verhalten unabhängig von der Vorsortierung identisch für alle Fälle.

Quicksort

Das Prinzip (Divide & Conquer):

- Wählt ein Pivot-Element aus der Liste.
- **Partitionierung:** Alle kleineren Elemente wandern nach links, alle größeren nach rechts.
- Das Pivot-Element ist danach an seiner finalen Position.
- Sehr schnell in der Praxis (In-Place, geringer Overhead).

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n \log n)$ – Pivot teilt die Liste exakt mittig.
- **Average Case:** $\mathcal{O}(n \log n)$ – Häufigster Fall in der Praxis.
- **Worst Case:** $\mathcal{O}(n^2)$ – Pivot ist extrem ungünstig.

Exkurs: Average-Case von Quicksort

Die Grundüberlegung:

- Sei $F(n)$ die durchschnittliche Anzahl (Erwartungswert) der Vergleiche für n Elemente.
- Das Pivot-Element erfordert bei der Partitionierung im ersten Schritt exakt $n-1$ Vergleiche.
- Das Pivot landet mit gleicher Wahrscheinlichkeit ($\frac{1}{n}$) an einer beliebigen Position $i \in \{1, \ldots, n\}$.
- Die verbleibenden Teillisten haben dann die Längen $i-1$ und $n-i$.

Die Rekursionsgleichung:

$$F(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (F(i-1) + F(n-i))$$

Exkurs: Average-Case von Quicksort (Umformung)

- Da $F(i-1)$ und $F(n-i)$ symmetrisch über die Summe laufen, lässt sich dies vereinfachen zu:

$$nF(n) = n(n-1) + 2\sum_{k=0}^{n-1} F(k)$$

- Um die Summe aufzulösen, betrachten wir die Gleichung für $n+1$:

$$(n+1)F(n+1) = (n+1)n + 2\sum_{k=0}^{n} F(k)$$

- Subtrahieren wir die erste von der zweiten Gleichung, entfällt die große Summe:

$$(n+1)F(n+1) - nF(n) = 2n + 2F(n)$$

- Umgestellt ergibt sich: $(n+1)F(n+1) = 2n + (n+2)F(n)$

Exkurs: Average-Case von Quicksort (Ergebnis)

- Dividiert man die umgestellte Gleichung durch $(n+1)(n+2)$, entsteht eine Form, die zu einer Teleskopsumme führt.
- Durch fortlaufendes Aufsummieren (für $k = 1, 2, \ldots, n-1$) stößt man auf die harmonische Reihe H_n:
$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$
- Aus der Analysis ist bekannt, dass $H_n \sim \ln(n)$ für große n.
- Das exakte Resultat nähert sich asymptotisch an:
$$F(n) \sim 2n \ln(n)$$
- **Fazit:** Da Konstanten ignoriert werden, ist der durchschnittliche Aufwand $\mathcal{O}(n \log n)$. Paradoxerweise tritt der Worst-Case von $\frac{n(n-1)}{2}$ Vergleichen (also $\mathcal{O}(n^2)$) genau dann auf, wenn die Liste bereits von Anfang an perfekt sortiert ist!

Heapsort

Das Prinzip:

- Interpretiert die Liste als Binärbaum und baut einen **Max-Heap** auf (größtes Element ist die Wurzel).
- Tauscht die Wurzel ans Ende der Liste (Element ist nun sortiert).
- Lässt das neue, kleine Wurzelelement nach unten sinken, um den Heap zu reparieren.
- Vereint die Geschwindigkeit von Merge Sort mit der Speicher-Effizienz (In-Place) von Selection Sort.

Asymptotische Laufzeit:

- **Best Case:** $\mathcal{O}(n \log n)$ (Sonderfälle mit $\mathcal{O}(n)$ ausgenommen)
- **Average Case:** $\mathcal{O}(n \log n)$
- **Worst Case:** $\mathcal{O}(n \log n)$

Beweis der Laufzeit: Heapsort

- **Phase 1 (Build-Heap):** Ein Max-Heap wird aufgebaut. Die meisten Knoten sind Blätter und können nicht tief sinken. Der Aufbau kostet erstaunlicherweise nur $\mathcal{O}(n)$ (Beweis auf nächster Folie).
- **Phase 2 (Sortdown):** Das Maximum (Wurzel) wird entnommen und an das Array-Ende getauscht. Das neue Element sinkt nach unten (Heapify).
- **Aufwand der Phase 2:** Dies geschieht für $(n-1)$ Elemente. Der Baum ist balanciert und hat die Tiefe $\log_2(n)$. Das Sinkenlassen kostet maximal $\mathcal{O}(\log n)$.
- **Gesamtaufwand:** $\mathcal{O}(n) + \mathcal{O}(n \log n) \implies \mathcal{O}(n \log n)$.

Exkurs: Warum Build-Heap in $\mathcal{O}(n)$ läuft

Intuition vs. Realität:

- Intuitiv denkt man: n Elemente einfügen, jedes sinkt bis zu $\log_2(n)$ tief $\implies \mathcal{O}(n \log n)$. **Das ist ein Denkfehler!**
- Wir bauen den Heap „Bottom-Up“ auf. Die allermeisten Knoten sind Blätter (Ebene $h = 0$) und sinken exakt **0 Schritte**.
- Auf der Ebene h (von unten gezählt) gibt es im Binärbaum maximal $\frac{n}{2^{h+1}}$ Knoten. Diese können höchstens h Schritte sinken.

Die mathematische Summe:

$$\text{Gesamtarbeit } W \leq \sum_{h=0}^{\log_2(n)} \frac{n}{2^{h+1}} \cdot h = \frac{n}{2} \sum_{h=0}^{\log_2(n)} \frac{h}{2^h}$$

Exkurs: Warum Build-Heap in $\mathcal{O}(n)$ läuft (Teil 2)

- Betrachten wir den Grenzwert dieser Summe als unendliche Reihe, so gilt aus der Analysis:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

- Da unser Baum endlich ist (er hat nur $\log_2(n)$ Ebenen), ist unsere Summe strikt kleiner oder gleich dem Grenzwert des Unendlichen:

$$W \leq \frac{n}{2} \cdot 2$$

- Die Konstante 2 kürzt sich wunderschön weg, es bleibt übrig:

$$W \leq n$$

- **Fazit:** Die Gesamtzahl der Vergleiche zum Aufbauen eines Max-Heaps ist nach oben strikt durch n begrenzt. Das asymptotische Wachstum ist damit bewiesenermaßen **linear**, also $\mathcal{O}(n)$.

Übersicht der Laufzeiten

Algorithmus	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Tabelle: Laufzeitkomplexitäten der Sortieralgorithmen

Sequenzielle Suche (Prinzip)

- Der Algorithmus `SequSearch` sucht ein Element x in einem Array $a[0..n]$ unter den ersten n Elementen.
- Dabei inspizieren wir die Elemente von $a[0..n-1]$ nacheinander, bis x gefunden oder das Ende von a erreicht ist.
- Der Algorithmus benötigt die Variable $a[n]$ als Wächterelement.

Sequenzielle Suche (Algorithmus)

Ablauf:

1. Initialisierung von Index i.
2. Setze das Wächterelement $a[n] \leftarrow x$ und starte bei $i \leftarrow 0$.
3. Führe eine `while`-Schleife aus, solange $a[i] \neq x$ gilt.
4. Erhöhe in jedem Schritt den Index: $i \leftarrow i + 1$.
5. Gib den gefundenen Index i zurück.

Binäre Suche (Prinzip)

- Der Algorithmus `BinSearch` sucht ein Element x in einem bereits sortierten Array a mit n Elementen.
- Er folgt der Divide-and-Conquer-Strategie.
- Das zu suchende Element x wird gezielt mit dem mittleren Element $a[i]$ verglichen.
- Falls der Vergleich $a[i] = x$ ergibt, ist das gesuchte Element gefunden.
- Falls x kleiner als $a[i]$ ist, befindet sich x links von $a[i]$, und falls x größer als $a[i]$ ist, befindet es sich rechts von $a[i]$.
- Das Teilarray für die weitere Suche ist jeweils etwa halb so groß wie das ursprüngliche Array, wodurch die Lösung des Problems auf die Lösung des Teilproblems reduziert wird (Lösungen müssen nicht zusammengesetzt werden).

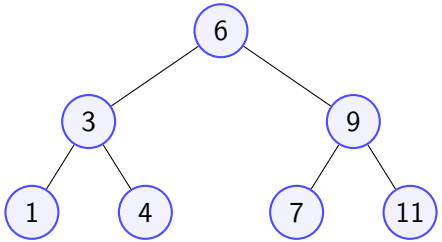
Binäre Suche (Algorithmus)

Ablauf:

1. Initialisiere die Suchgrenzen: $l \leftarrow 0$ und $r \leftarrow n - 1$.
2. Starte eine `repeat-until`-Schleife.
3. Berechne die Mitte: $i \leftarrow (l + r)$ div 2.
4. Wenn $a[i] < x$ gilt, verschiebe die linke Grenze ($l \leftarrow i + 1$), andernfalls die rechte Grenze ($r \leftarrow i - 1$).
5. Wiederhole dies, bis $a[i] = x$ oder $l > r$.
6. Wenn $a[i] = x$, gib i zurück, andernfalls -1.

Binäre Suche (Zugriffspfade)

- Die Zugriffspfade, die beim Suchen der Elemente entstehen, lassen sich strukturiert darstellen.
- Ein binärer Suchbaum dient der Navigation bei der Suche in einem Array.



Binäre Suche (Laufzeitanalyse - Satz)

Iterationen im schlechtesten Fall:

- Für die Anzahl $I(n)$ der Iterationen der `repeat-until`-Schleife gilt im schlechtesten Fall: $I(n) \leq \lfloor \log_2(n) \rfloor + 1$.
- Da pro Iteration Vergleiche stattfinden, beträgt die Anzahl der Vergleiche $2I(n)$.
- Die maximale Anzahl der Vergleiche ist somit $\leq 2(\lfloor \log_2(n) \rfloor + 1)$.

Binäre Suche: Beweis

Beweisführung:

- In jeder Iteration teilen wir $n-1$ Elemente auf in Teilarrays mit etwa $\lfloor \frac{n}{2} \rfloor$ vielen Elementen.
- Im schlechtesten Fall terminiert die `repeat-until`-Schleife mit $l > r$.
- $I(n)$ genügt daher der folgenden Rekursion: $I(n) \leq I(\lfloor \frac{n}{2} \rfloor) + 1$.
- Der Basisfall dieser Rekursion ist $I(1) = 1$.
- Hieraus folgt die Behauptung (mit Verweis auf Satz 1.28).