DHBW Ravensburg

Programmierung II – Algorithmen & Datenstrukturen

Prof. Dr. Andreas Schilling

www.ravensburg.dhbw.de

2. Such- & Sortieralgorithmen

Agenda

Suchalgorithmen

Elementare Strategien zum Suchen von Elementen in Listen.

Elementare Sortieralgorithmen

Elementare Strategien zum Sortieren von Listen.

Rekursive Sortier-algorithmen

Rekursive Strategien zum Sortieren von Listen.

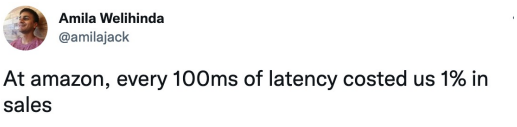
Suchalgorithmen

Motivation

- Suchabfragen sind von **grundlegender Bedeutung** in unserem täglichen Leben:
  - Auflösen einer Kunden- oder Auftragsnummer
  - Identifikation von Verkehrsverbindungen (bspw. Zug, Bahn, Flug)
  - Überführung einer URL zu einer IP-Adresse
  - Suche bei YouTube/Facebook/Instagram
- Die Dauer der Suchabfrage ist **entscheidend für das Kundenerlebnis**.
- Wichtig bei der Bewertung von Suchalgorithmen ist wie sie sich bei **wachsender Inputmenge verhalten**.

Delays in loading *web pages and videos under time pressure caused mobile* ***users' heart rates to rise an average of 38 percent***. ***Six-second delays*** *to video streaming caused stress levels to increase* ***by a third***. *To put that in context, the stress incurred is equivalent to the* ***anxiety of taking a math test or watching a horror movie*** *alone, and greater than the stress experienced by standing at the edge of a virtual cliff.*

Ericsson (2016)



Das schnelle Suchen von Elementen in großen Datensätzen ist von großer Bedeutung für unser tägliches Leben. Oft nehmen wir es als selbstverständlich an, dass Kunden- und Auftragsnummern in minimaler Zeit aufgelöst werden. Auch Anfragen, welche die schnellsten Verkehrsverbindungen identifizieren oder die Übersetzung einer URL in eine IP Adresse vornehmen, geschehen heute so schnell, das wir uns deren Ausführung oft gar nicht bewusst sind. Untersuchen von Ericsson zeigen, dass wir keine Toleranz für Wartezeiten haben, konkret verursachen bei uns bereits 6 Sekunden Wartezeit größeres Unwohlsein als am Rande einer Klippe zu stehen. Dies zeigt wie wichtig die schnelle Beantwortung von Suchabfragen ist.

Für schnelle Suchabfragen ist der verwendete Algorithmus entscheidend, insbesondere sein Grenzverhalten, da angesichts von Big-Data, sehr schnell riesige Datenmengen entstehen.

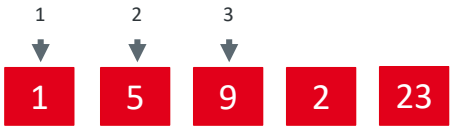
Vgl. https://www.ericsson.com/en/press-releases/2016/2/streaming-delays-mentally-taxing-for-smartphone-users-ericsson-mobility-report

Suchalgorithmen

Lineare Suche: Grundidee

- Die lineare Suche ist die **grundlegendste Form** der Suche nach einem bestimmten Element.
- Der Algorithmus **durchläuft Schritt für Schritt den Array und vergleicht jedes Element** mit dem Suchwert.
- Der Algorithmus kann sowohl auf **unsortierten als auch sortierten Zahlenfolgen** angewandt werden.
- Wenn eine sortierte Zahlenreihe vorausgesetzt werden kann, lässt sich die Laufzeit des Algorithmus optimieren.

Befindet sich die Zahl 9 im Array ?



Die grundlegendste Form die Elementsuche ist die lineare Suche. Hierbei wird jedes Feld des Arrays Schritt für Schritt beginnend mit dem ersten Element durchlaufen und mit dem Suchwert verglichen. Bei einer Übereistimmung wird die Position des Suchwerts zurückgegeben. Die lineare Suche ist sehr robust gegenüber der Zahlenanordnung im Array, das heißt sie funktioniert sowohl bei bei sortierten als auch unsortierten Zahlenfolgen. Allerdings lässt sich das Laufzeitverhalten der linearen Suche optimieren werden wenn man voraussetzen kann, dass der Array sortiert ist.

Suchalgorithmen

Lineare Suche: Implementierung

- **Inputparameter** sind der **sortierte Array** und jene Zahl deren Position bestimmt werden soll.
- Eine **for-Schleife** geht über jede Position des Arrays und **vergleicht den Feldinhalt** mit dem Suchwert
- Algorithmus **bricht die Suche ab**, wenn:
  - **die gewünschte Zahl** identifiziert wurde
  - der aktuelle **Feldinhalt größer** als die gesuchte Zahl ist (Optimierung für sortierte Zahlenfolgen)

```java
public static int linearSearch(int[] sortedArray, int searchNum){
    for(int i=0;i<sortedArray.length;i++){
        //Vergleiche jedes Element im Array mit dem Suchwert
        if (sortedArray[i] == searchNum){
            return i;
        }
        //Breche Suche ab, falls aktueller Wert größer als Suche ist
        if (sortedArray[i] > searchNum){
            return -1;
        }
    }
    //Wenn das Element nicht gefunden wurde gebe -1 zurück
    return -1;
}
```

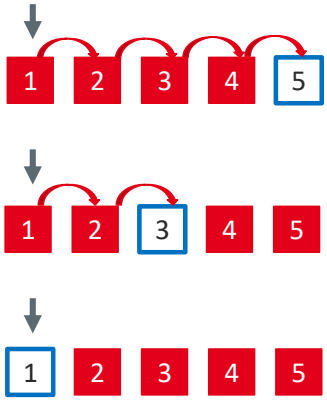
*Umsetzung der linearen Suche **mit Optimierung** für sortierte Zahlenreihe*

Befassen wir uns nun mit der Implementierung der linearen Suche. Der abgebildete Code zeigt wie die lineare Suche auf einem sortierten Array umgesetzt werden kann. Hier wird der Array Schritt für Schritt durchsucht. Bei einer Übereinstimmung mit dem gesuchten Wert wird der Feldindex zurückgegeben. Darüber hinaus bricht der Algorithmus ab, sobald der aktuelle Feldinhalt größer ist als der gesuchte Wert. Diese Abbruchbedingung ist eine Optimierung, die nur bei vorliegen eines sortierten Arrays angewendet werden kann.

Suchalgorithmen

Lineare Suche Laufzeit

- **Worst-Case**: Das gesuchte Element befindet sich am Ende des Array, sodass n-1 Elemente durchsucht werden müssen -> O(n)
- **Avg-Case**: Das gesuchte Element befindet sich in der Mitte des Array, sodass ca. $\frac{n}{2}$ Elemente durchsucht werden müsse -> O(n)
- **Best-Case**: Das gesuchte Element ist das erste Element im Array -> O(1)
- Beim Vorliegen eines sortierten Arrays, lässt sich die Suche nach **nicht vorhandenen Elementen früher abbrechen**.



7

Im Folgenden untersuchen wir das asymptotische Verhalten der linearen Suche. Der Worst-Case der linearen Suche liegt vor, wenn das gesuchte Element ganz am Ende des Arrays liegt oder gar nicht im Array vorhanden ist. In einem derartigen Fall muss auf jedes Element des Arrays zugegriffen werden, sodass eine Laufzeit von O(n) entsteht, wobei n die Anzahl der Elemente im Array entspricht. Im durchschnittlichen Fall befindet sich das gesuchte Element in der Mitte des Arrays, sodass in Folge dessen auch nur ca. n/2 Vergleiche gemacht werden müssen. Da konstante Faktoren bei der Landau Notation ignoriert werden folgt hieraus O(n). Abschließend liegt der Best-Case vor, wenn sich das gesuchte Element am Anfang des Arrays befindet, sodass eine konstante Zugriffszeit entsteht O(1).

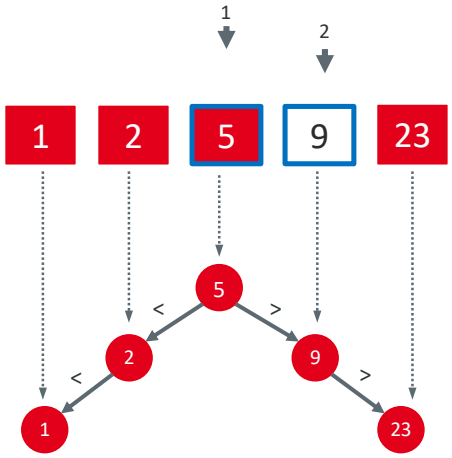
Sollte der Array sortiert sein lässt sich die Suche nach Elementen, die im Array nicht vorhanden sind, optimieren, da hier nicht mehr statisch der gesamte Array durchlaufen werden muss.

Suchalgorithmen

Binäre Suche: Grundidee

- Für die binäre Suche muss der Array in **sortierter Reihenfolge** vorliegen.
- Die binäre Suche **halbiert in jedem Schritt** den **Suchraum**.
- Die gesuchte Zahl wird mit dem Element in der **Mitte des Arrays (Pivotelement)** verglichen.
- Ist die Zahl **größer** wird mit der **rechten Hälfte** weitergearbeitet, **ansonsten** mit der **linken Hälfte**.
- Anschließend wird aus dem **reduzierten Suchraum wieder das mittlere Element** verglichen, bis das gesuchte Element gefunden wurde oder der Suchraum leer ist.

Befindet sich die Zahl 9 im Array ?



8

Die binäre Suche setzt einen sortierten Array voraus. Konkret wird in einem rekursiven Vorgehen stets das mittlere Element des Suchraums mit dem gesuchten Element verglichen und der Suchraum mit jedem Schritt halbiert. Ist das gesuchte Element kleiner als das Vergleichselement wird mit der linken Teilhälfte weitergearbeitet. Ist das gesuchte Element dagegen größer als das Vergleichselement wird mit der rechten Hälfte weitergearbeitet. Das Element das mit dem gesuchten Element verglichen wird, wird als Pivotelement bezeichnet.

Suchalgorithmen

Binäre Suche: Implementierung

- Inputparameter sind der sortierte Array und die Zahl deren Position bestimmt werden soll.
- Die Variablen **rechteGrenze** und **linkeGrenze** begrenzen den aktuellen **Suchraum.**
- Zur Bestimmung des **Pivotelements** wird die mittlere Position zwischen rechter und linker Grenze berechnet und abgerundet.
- **Abhängig vom Vergleich** des Pivotelements mit der gesuchten Zahl wird die **rechte** oder **linke Grenze angepasst**.

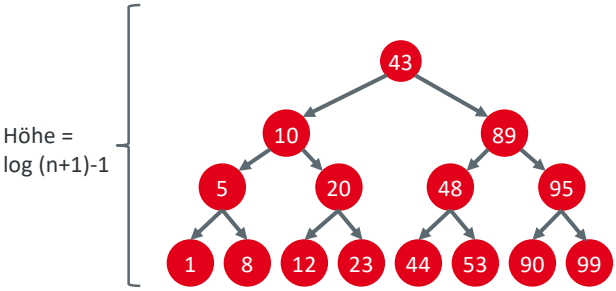
```java
public static int binarySearch(int[] sortedArray, int searchNum){
    int pivotPos = 0;
    int rechteGrenze = sortedArray.length -1;
    int linkeGrenze = 0;
    //solange die rechteGrenze nicht über die linke geht
    while (linkeGrenze <= rechteGrenze){
        pivotPos = (linkeGrenze+rechteGrenze)/2;
        //Vergleiche Pivotelement
        if(sortedArray[pivotPos] == searchNum){
            return pivotPos;
        }
        else{
            //Schaue ob das Pivotelement größer oder kleiner ist
            if (sortedArray[pivotPos] < searchNum){
                linkeGrenze = pivotPos +1 ;
            }else{
                rechteGrenze = pivotPos - 1;
            }
        }
    }
    //Wenn das Element nicht gefunden wurde gebe -1 zurück
    return -1;
}
```


Der abgebildete Code zeigt die Umsetzung der binären Suche in Java. Solange die rechte Grenze größer oder gleich der linken Grenze ist, wird die Position des Pivotelements durch die Mitte (abgerundet) zwischen linker und rechter Grenze bestimmt. Im Anschluss daran wird der Feldinhalt des Pivotelements mit dem gesuchten Element verglichen. Liegt eine Übereinstimmung vor wird die Position des Pivotelements zurückgegeben. Liegt keine Übereinstimmung vor wird abhängig davon ob das gesuchte Element größer oder kleiner als das Pivotelement ist mit dem rechten oder dem linken Teil des Array weitergearbeitet indem die entsprechende Grenze angepasst wird.

Suchalgorithmen

Binäre Suche: Laufzeit

- **Worst-Case**: Gesuchtes Element ist ein Blatt. Da die Höhe in einem balancierten Binärbaum log(n +1) entspricht folgt daraus O(log n).
- **Avg-Case**: Gesuchtes Element befindet sich innerhalb des Binärbaums. Auch dann sind maximal log n Vergleiche notwendig -> O(log n)
- **Best-Case:** Gesuchtes Element entspricht dem Wurzelelement -> O(1)



Befassen wir uns zum Abschluss mit der Laufzeiteigenschaft der binären Suche. Zum besseren Verständnis des Laufzeitverhaltens stellen wir den durchsuchten Array als Binärbaum dar. Der Worst-Case würde eintreten, wenn das gesuchte Element ein Blatt im dargestellten Suchbaum ist. Wie wir im letzten Kapitel gelernt haben entspricht die Höhe eines balancierten Binärbaums der Formel log(n+1)-1, sodass im schlimmsten Falle log(n+1)-1 Vergleiche notwendig sind und daraus O(log n) folgt. Im durchschnittlichen Fall befindet sich das gesuchte Element in der Nähe eines Blattknotens, sodass auch hier die obere Schranke O(log n) gilt. Im besten Falle entspricht das gesuchte Element dem Wurzelknoten, sodass stets mit dem ersten Vergleich das Element gefunden wird. Dies entspricht O(1).

Suchalgorithmen

Interpolationssuche: Grundidee

- Die Interpolationssuche setzt einen **sortierten Array** voraus.
- Im Gegensatz zur binären Suche wird **nicht immer das mittlere** Element des Arrays verglichen, sondern das Vergleichselement wird **mit linearer Interpolation** bestimmt.
- Das Vorgehen besitzt Ähnlichkeit zur Namenssuche in einem Telefonbuch, wo ebenfalls aufgrund der proportionalen Nähe zur Ausgangsseite eine Näherung durchgeführt wird.
- Formel zur Positionsbestimmung

$$P = l + \frac{s - array[l]}{array[r] - array[l]} * (r - l)$$

Befindet sich die Zahl 9 im Array ?

l = linke Grenze
r = rechte Grenze
P = Position d. Vergleichselement
s = Suchkriterium

Eine weitere Möglichkeit zur Suche nach Elementen in einem sortierten Array ist die Interpolationssuche. Die Grundidee dieser Form der Suche ist, dass mehr Mühe in die Auswahl des zu vergleichenden Elements investiert wird, statt wie bei der binären Suche immer nur das mittlere Element zu verwenden. Stattdessen wird das zu vergleichende Element via linearer Interpolation bestimmt. Diese Vorgehensweise ist ähnlich zur Namenssuche in einem Telefonbuch. Typischerweise versucht man die passende Seite durch die Nähe des aktuellen Namens mit dem gesuchten Namen zu finden. Ist der aktuelle Name lexigrafisch stark vom gesuchten Namen entfernt wird ein größerer Block an Seiten übersprungen, als wenn der aktuelle Name starke Ähnlichkeit zum gesuchten Namen aufweist.

Zur Bestimmung der Position des Vergleichselements wird die dargestellte Formel verwendet. Konkret wird hierbei das Delta zwischen dem gesuchten Element und dem aktuellen Vergleichskriterium in Proportion zur gesamten Spannbreite der im Array vertretenen Zahlen berechnet. Dieses Verhältnis wird mit der Spannbreite des aktuellen Suchraums multipliziert und auf die linke Grenze addiert.

Suchalgorithmen

Interpolationssuche: Implementierung

- Inputparameter sind der sortierte Array und jene Zahl deren Position bestimmt werden soll.
- Identischer Aufbau zur binären Suche, mit Ausnahme der **Bestimmung der Position des Pivotelements.**
- Die Position des Pivotelements wird bestimmt, gemäß Formel auf der letzten Seite
- Abhängig davon ob das gesuchte Element **größer oder kleiner als das Pivotelement** ist, wird die **linke oder rechte Grenze** angepasst.

```java
public static int interpolationSearch(int[] sortedArray, int searchNum){
    int pivotPos = 0;
    int rechteGrenze = sortedArray.length -1;
    int linkeGrenze = 0;
    //solange die rechteGrenze nicht über die linke geht
    while (linkeGrenze <= rechteGrenze){
        //Berechnung Position Pivotelement
        pivotPos = linkeGrenze +
                (searchNum - sortedArray[linkeGrenze])/
                        (sortedArray[rechteGrenze] - sortedArray[linkeGrenze])*
                (rechteGrenze - linkeGrenze);
        //Vergleiche Pivotelement
        if(sortedArray[pivotPos] == searchNum){
            return pivotPos;
        }
        else{
            //Schaue ob das Pivotelement größer oder kleiner ist
            if (sortedArray[pivotPos] < searchNum){
                linkeGrenze = pivotPos +1 ;
            }else{
                rechteGrenze = pivotPos - 1;
            }
        }
    }
    //Wenn das Element nicht gefunden wurde gebe -1 zurück
    return -1;
}
```

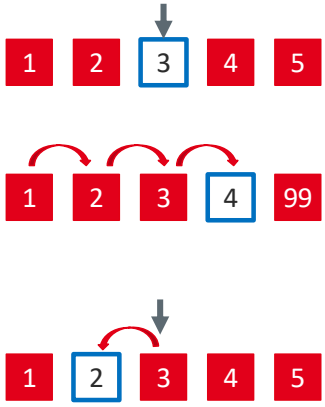

Schauen wir uns nun die Umsetzung der Interpolationssuche in Java an. Der Codeaufbau hat große Ähnlichkeit zur binären Suche. Der zentrale Unterschied liegt im Abschnitt bzgl. der Bestimmung des Pivotelements. Die Bestimmung des Pivotelements wird hierbei gemäß der Formel auf der letzten Folie bestimmt. Anschließend wird der Feldinhalt des Pivotelements mit dem gesuchten Element verglichen und falls eine Übereinstimmung vorliegt die Position des Pivotelements zurückgegeben. Liegt keine Übereinstimmung vor wird je nachdem ob der Feldinhalt des Pivotelements größer oder kleiner als das gesuchte Element ist die linke oder die rechte Grenze des Arrays angepasst.

Suchalgorithmen

Interpolationssuche: Laufzeit

- **Best-Case**: Das zu suchende Element entspricht dem ersten Pivotelements -> $\mathcal{O}(1)$
- **Worst-Case**: Die Elemente sind nicht gleichverteilt (bspw. Ausreißer). Durch die Verzerrung des Wertebereichs findet eine schrittweise Annäherung statt -> $\mathcal{O}(n)$
- **Avg-Case**: Die Elemente weisen eine Gleichverteilung auf mit geringen Ausreißern. Infolge dessen werden deutlich weniger Schritte als die binäre Suche benötigt.

  $\mathcal{O}(log\ log\ n)$



Befassen wir uns nun mit dem Laufzeitverhalten der Interpolationssuche. Im Falle des Best-Case entspricht das erste Vergleichselement dem gesuchten Element, sodass mit der ersten Abfrage bereits das gesuchte Element identifiziert wird (O(1)). Der Worst-Case tritt ein wenn die Varianz der Zahlen im Array sehr hoch ist, wie bspw. im dargestellten Fall. Bei einer derart hohen Varianz wird die Positionsbestimmung aufgrund der hohen Differenz zwischen dem gesuchten Wert und den Vergleichskriterium verzerrt und es erfolgt eine sequenzielle Näherung an das gesuchte Element. Im Extremfall werden hierfür n-1 Vergleiche notwendig, sodass O(n) folgt. Im durchschnittlichen Fall, bei einer homogenen Verteilung der Zahlen, zeigt sich, dass durch die lineare Interpolation die Performance der binären Suche übertroffen werden kann. Konkret ist es möglich den Suchraum deutlich enger einzugrenzen als bei der binären Suche, wodurch weniger Vergleiche notwendig werden, konkret O(log log n).

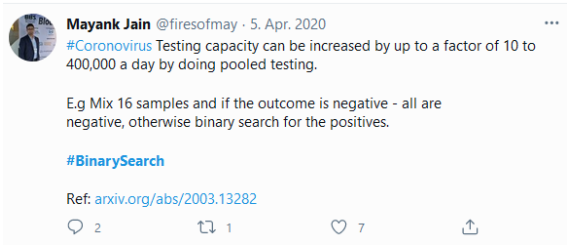
Suchalgorithmen

Überblick

- Im **Best-Case** haben **alle** Suchalgorithmen **eine konstante Suchdauer.**
- Im **Worst-Case** schneidet die **Binärsuche** mit einer Laufzeit von $\mathcal{O}(\log n)$ Operationen am besten ab.
- Im **Avg.-Case** schneidet die **Interpolationssuche** am besten ab $\mathcal{O}(\log \log n)$.
- Nur die **lineare Suche** unterstützt **nicht sortierte** Arrays.
- **Grobes Wissen** zur Struktur der Daten ist **zentral** um abzuschätzen ob die Interpolationssuche verwendet werden sollte.

Algorithmus	Best-Case	Avg.-Case	Worst-Case
Lineare Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binäre Suche	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Interpolation	$\mathcal{O}(1)$	$\mathcal{O}(\log \log n)$	$\mathcal{O}(n)$

Binäre Suche in der Virologie:



Stellen wir die drei beleuchteten Suchalgorithmen einander gegenüber fällt auf, dass alle Algorithmen im Best-Case eine konstante Suchdauer aufweisen. Im Falle des Worst-Case benötigt die binäre Suche am wenigsten Vergleiche. Im durchschnittlichen Fall, schneidet die Interpolationssuche am besten ab, da hier die Position des Pivotelements dynamisch bestimmt wird, was im Falle von homogen verteilten Zahlenfolgen deutlich schneller zum Ziel führt als bei allen anderen Verfahren. Die lineare Suche schneidet im Vergleich zur binären Suche und zur Interpolationssuche schlechter ab, es darf aber nicht vergessen werden, dass die lineare Suche im Gegensatz zu den anderen Verfahren auch mit nicht sortierten Zahlenreihen erfolgreich funktioniert.

Der Vergleich der Suchalgorithmen zeigt, dass ein grobes Wissen zur Struktur der vorliegenden Elemente notwendig ist um den passendsten Suchalgorithmus zu bestimmen.

Aufgabe 5

Suchalgorithmen

a) Finden Sie die Zahl 52 in den nebenstehenden Zahlenfolgen anhand der **linearen Suche (ohne Optimierung)**, der **binären Suche** und der **Interpolationssuche**. Notieren Sie dabei alle Elementvergleiche.

b) Welchen Suchalgorithmus würden Sie bei einer verketteten Liste verwenden und warum?

I. 1 5 8 36 52 86

II. 1 5 52 65 72 98

III. 86 8 5 36 52 1

Aufgabe 5

Suchalgorithmen

c) Implementieren Sie die vorgestellte binäre Suche als rekursiven Algorithmus. Testen Sie Ihr Vorgehen anschließend mit der ersten Zahlenreihe von Folie 8

Agenda

Suchalgorithmen

Elementare Strategien zum Suchen von Elementen in Listen.

Elementare Sortieralgorithmen

Elementare Strategien zum Sortieren von Listen.

Rekursive Sortier-algorithmen

Rekursive Strategien zum Sortieren von Listen.

Sortieralgorithmen

Grundlagen

- Unter Sortieren versteht man den Vorgang eine Liste von Daten in eine **geordnete Reihenfolge** zu bringen.
- Sortierte Datensätze sind **zentral für unser Leben**:
  - Verbesserte Suchleistungen (bspw. Telefonbuch, Bibliothek)
  - Für den Benutzer sind sortierte Daten deutlich einfacher zu interpretieren (bspw. Zeitreihen)
  - Berechnungen von Statistiken (bspw. Median)
- Etwa ein **Viertel der kommerziellen Rechenleistung** wird für Sortieralgorithmen verwendet
  (Ottmann & Widmayer 2012)

Neben der Elementsuche ist das Sortieren von Zahlenfolgen eine zentrale Grundoperation der Informatik. Unter sortieren versteht man, dass die Elemente einer Liste in eine geordnete Reihenfolge gebracht werden. Das Sortieren von Elementen begleitet uns so allgegenwärtig in unserem täglichen Leben, dass wir uns dessen oft gar nicht bewusst sind. In Büchereien sind beispielsweise die Bücher in einer Reihenfolge angeordnet. Das gleiche gilt für die Ablage in Schränken (bspw. Apothekerschrank). Neben dem schnelleren Zugriff dient das Sortieren aber auch der erleichterten menschlichen Interpretation von Daten, bspw. wenn Sie ein Phänomen über längere Zeitpunkte hinweg betrachten. Auch Statistiken wie bspw. die Medianberechnung benötigen zur Berechnung eine sortierte Liste.

Aufgrund dieser zentralen Bedeutung für den Benutzer aber auch für das Funktionieren von Anwendungen ist es nicht verwunderlich, dass ca. ein Viertel der kommerziellen Rechenleistung nur für Sortieralgorithmen verwendet wird.

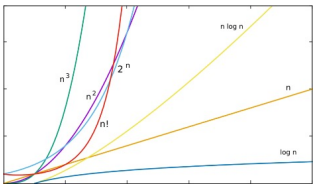
Sortieralgorithmen

Grundlagen

Es gibt **nicht** den **universell besten Sortieralgorithmus**, sondern es müssen **stets** folgende Eigenschaften **abgewogen werden:**

- **Laufzeit:** Wie verhält sich die Laufzeit des Algorithmus (in Operationen) bei steigendem Input? Wie lautet die asymptotische Ordnung?
- **Stabilität**: Bleibt die Ordnung gleichwertiger Elemente bestehen?
- **Speicherplatz:** Wieviel Speicherplatz wird vom Algorithmus zur Sortierung benötigt?

Laufzeitkomplexität



Geburtsjahr	Nachname
1989	Müller
1990	Schmidt
1992	Maurer
1992	Schulze

Bei stabilen Sortierungen bleibt die ursprüngliche Reihenfolge gleichartiger Elemente erhalten

Bevor wir uns mit konkreten Algorithmen zur Sortierung beschäftigen, betrachten wir im Folgenden die Eigenschaften von Sortieralgorithmen. Dies ist notwendig, da es wie bei den Suchalgorithmen nicht einen universtell besten Sortieralgorithmus gibt, sondern es müssen bei der Auswahl stets kontextspezifische Eigenschaften berücksichtigt werden.

Eine der wichtigsten Eigenschaften ist das Laufzeitverhalten des Sortieralgorithmus bei steigender Inputmenge. Wie bei der Elementsuche identifizieren wir hier die Komplexitätsklasse des jeweiligen Algorithmus.

Eine weitere Eigenschaft der Sortieralgorithmen betrifft die Stabilität einer vorhandenen Reihenfolge. Zur Verdeutlichung nehmen wir das Beispiel der dargestellten Tabelle, in der die Namen nach deren Geburtsjahr geordnet sind. Ein stabiler Algorithmus für das Geburtsjahr würde nun die bereits bestehende Sortierung des Nachnamens berücksichtigen, sodass auch nach der Sortierung nach dem Geburtsjahr die Person Maurer vor der Person Schulze steht.

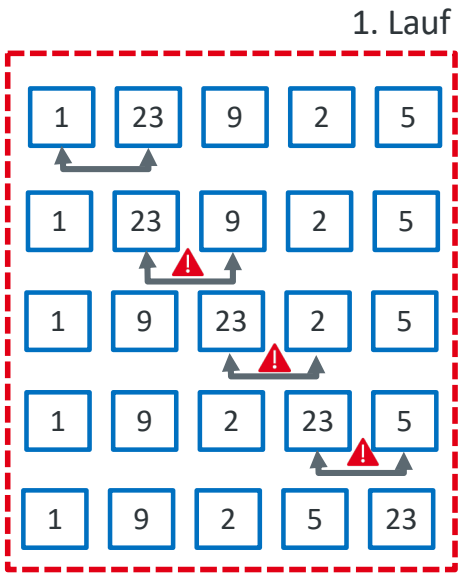
Abschließend ist ein sehr wichtiges Kriterium zur Beurteilung der Sortieralgorithmen die Größe des benötigten Speicherplatzes und die Frage ob der Speicherplatz mit der Inputmenge wächst.

Grafik https://www.osa.fu-berlin.de/informatik_msc/beispielaufgaben/laufzeiten/index.html

Sortieralgorithmen

Bubblesort: Grundidee

- Bei **Bubblesort** durchläuft der Algorithmus pro Lauf sequenziell ein Array und **vergleicht jedes Element mit seinem Nachtbar**. Befinden sich die Elemente nicht in der richtigen Ordnung zueinander werden sie miteinander getauscht.
- **Nach dem ersten Durchlauf** befindet sich das **größte Element am Ende des Arrays**. Nach dem zweiten Durchlauf befindet sich das zweitgrößte Element an zweitletzter Position.
- Insgesamt sind **n-1 Durchläufe notwendig**, bis Bubblesort eine sortierte Liste zurückgibt.
- Das Verfahren hat seinen Namen durch die Parallele mit Luftblasen wo **größere Luftblasen schneller als kleinere aufsteigen**.



Bubblesort stellt die grundlegendste Form dar einen Array zu sortieren. Hierbei wird jedes Element mit seinem Nachbar verglichen und das größere Element der Beiden wird nach rechts gesetzt. Nach dem ersten Durchlauf, d. h. dem schrittweisen paarweisen Vergleich mit allen Elementen (vgl. Grafik), befindet sich das größte Element am Ende des Arrays. Insgesamt sind n-1 Durchläufe notwendig um einen sortierten Array zu erzeugen. Der Algorithmus trägt seinen Namen, aufgrund seiner Ähnlichkeit zum Aufsteigen von Luftblasen unter Wasser. Auch hier gibt es das Phänomen, dass große Luftblase schneller an die Oberfläche steigen als kleine Luftblasen.

Sortieralgorithmen

Bubblesort: Implementierung

- Die **äußere Schleife** repräsentiert die **einzelnen Durchläufe.**
- Die **innere Schleife** setzt die **paarweisen Vergleiche** um.
- Der **Algorithmus** benötigt nur **ein Feld (temp) als Zwischenspeicher** (konstanter Speicherverbrauch).
- Bubblesort ist ein **stabiler Sortieralgorithmus**, der die ursprüngliche Ordnung des Array respektiert.

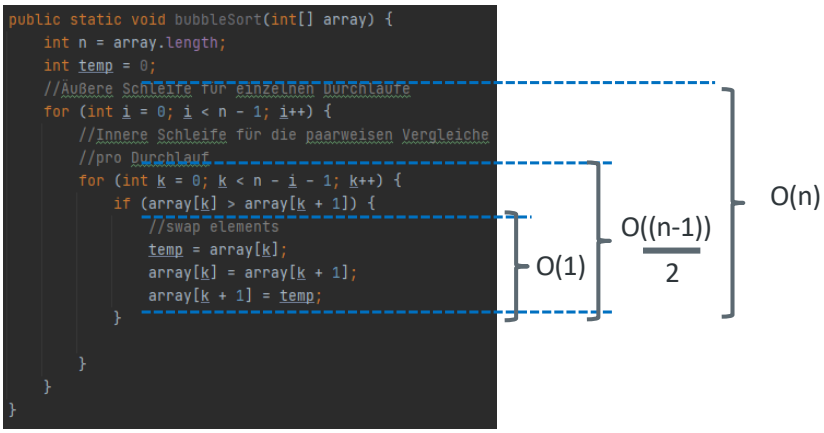
```java
public static void bubbleSort(int[] array) {
    int n = array.length;
    int temp = 0;
    //Äußere Schleife für einzelnen Durchläufe
    for (int i = 0; i < n - 1; i++) {
        //Innere Schleife für die paarweisen Vergleiche
        //pro Durchlauf
        for (int k = 0; k < n - i - 1; k++) {
            if (array[k] > array[k + 1]) {
                //swap elements
                temp = array[k];
                array[k] = array[k + 1];
                array[k + 1] = temp;
            }

        }
    }
}
```


Schauen wir uns nun an wie wir Bubblesort in Java implementieren können. Hierbei repräsentiert die äußere Schleife, die einzelnen Durchläufe durch das Array. Die innere Schleife setzt die paarweisen Vergleiche um. Hier wird jedes Element mit seinem Nachbarelement verglichen wobei das größere Element der Beiden auf die rechte Position geschrieben wird. Bei einem Wechsel wird die Variable temp als Zwischenspeicher genutzt. Dadurch dass der Algorithmus einen konstanten Zwischenspeicher benötigt gilt Bubblesort als sehr speichereffizient. Abschließend ist festzuhalten, dass Bubblesort ein stabiler Sortieralgorithmus ist, der nicht die Reihenfolge von gleichgroßen Elementen verändert.

Sortieralgorithmen

Bubblesort: Laufzeit



- **Worst-Case**: Äußere und innere Schleife werden durchlaufen und jedes Element wird getauscht:
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$
- **Avg-Case**: Äußere und innere Schleife wird durchlaufen aber nur die Hälfte der Elemente werden getauscht
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$
- **Best-Case**: Äußere und innere Schleife wird durchlaufen, aber es wird kein Element getauscht.
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$

22

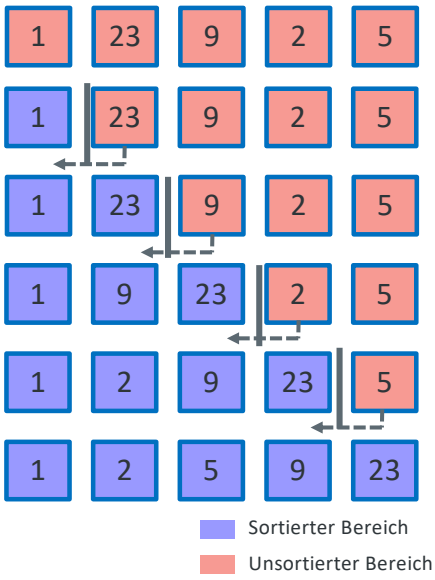
Widmen wir uns nun der Codeanalyse um das Laufzeitverhalten von Bubblesort zu verstehen. Die äußere Schleife (Anzahl der Durchläufe) geht über den Array n mal. Die innere Schleife benötigt dagegen insgesamt (n-1)/2 Schleifendurchläufe. Die Herleitung dieser Approximation, kann man sich am Besten visuell als die Hälfte eines regulären Schleifendurchlaufs vorstellen, da die Länge des Durchlaufs mit der Position der übergeordneten Schleife abnimmt. Der Elementwechsel hat einen konstanten Aufwand.

Der Worst-Case für Bubblesort liegt vor, wenn die Elemente des Arrays invers zur Zielreihenfolge vorliegen. In diesem Fall muss die äußere und innere Schleife durchlaufen werden und jedes Element getauscht werden. Infolge dessen sind $O(n^2)$ Vergleiche notwendig. Im Average-Case wo der Array „halb-sortiert“ vorliegt, müssen auch wieder die äußere und die innere Schleife komplett durchlaufen werden, aber nur die Hälfte der Elemente getauscht werden -> $O(n^2)$. Abschließend muss auch im Best-Case, wenn der Array bereits sortiert vorliegt und kein Element getauscht wird, die äußere und innere Schleife durchlaufen werden -> $O(n^2)$

Sortieralgorithmen

Insertionsort: Grundidee

- Der **Insertionsort** Algorithmus untergliedert den Array in einen **sortierten** und einen **unsortierten Bereich.**
- **Zu Beginn** ist **nur das erste Element „geordnet“**.
- **Schritt für Schritt** werden **Elemente** aus dem **unsortierten Bereich** an die **richtige Stelle im sortierten Bereich** abgelegt.
- **Mit jedem Durchlauf wächst** der **sortierte Bereich** und der unsortierte Bereich schrumpft.
- Insertionsort hat große Ähnlichkeit zum **„intuitiven“ Sortieren der Karten in einem Kartenspiel.**



Insertionsort ist ein weiterer elementarer Sortieralgorithmus. Dieser Algorithmus unterteilt den zu sortierenden Array in einen sortierten und einen unsortierten Bereich, wobei der sortierte Bereich zu Beginn nur aus dem ersten Element besteht. Sukzessiv werden nun Elemente aus dem unsortierten Bereich entnommen und an die passende Stelle im sortierten Bereich abgelegt. Durch dieses Vorgehen verkleinert sich der unsortierte Bereich des Arrays Schritt für Schritt wohingegen der sortierte Bereich des Arrays wächst. Das Vorgehen des Algorithmus hat große Ähnlichkeit zum intuitiven Sortieren von Spielkarten in einem Kartenspiel, bei dem man ebenfalls den sortierten Bereich der Karten schrittweise ausbaut.

Sortieralgorithmen

Insertionsort: Implementierung

- Die **äußere Schleife** wählt das **nächste zu vergleichende Element** aus.
- Die **innere Schleife** identifiziert die **korrekte Position** für das ausgewählte Element **im sortierten Bereich**.
- Der Algorithmus benötigt nur **ein Feld (temp)** für die Sortierung **(konstanter Speicherverbrauch)**.
- **Die Sortierung ist stabil**, d.h. sie behält die ursprüngliche Ordnung gleichartiger Elemente bei.

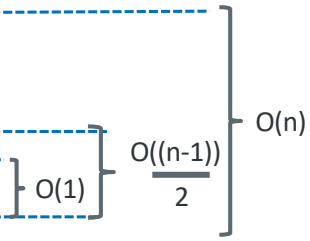
```java
public static int[] insertionSort(int[] unsortedArray) {
    int temp = 0;
    //Äußere Schleife wählt das nächste zu vergleichende Element aus
    for (int i = 0; i < unsortedArray.length; i++) {
        temp = unsortedArray[i];
        int k = i;
        //Innere Schleife identifiziert die korrekte Position im
        // sortierten Bereich
        while(k>0  &&  unsortedArray[k-1]>temp){
            unsortedArray[k] = unsortedArray[k-1];
            k--;
        }
        unsortedArray[k] = temp;
    }
    return unsortedArray;
}
```


Der folgende Codeauszug beschreibt die Implementierung von Insertionsort in Java. Hierbei repräsentiert die äußere Schleife die Auswahl des nächsten Elements aus dem unsortierten Bereich und die innere Schleife beschreibt die Einordnung des ausgewählten Elements im sortierten Teil des Array. Der Laufindex k wird verwendet um jene Stelle im sortierten Bereich des Arrays zu identifizieren, bei der zum ersten Mal der Vorgänger (k-1) kleiner oder gleich groß dem ausgewählten Element ist. Wie auch Bubblesort benötigt Insertionsort nur ein weiteres Feld als Zwischenspeicher, sodass ein konstanter Speicherverbrauch vorliegt. Ferner ist die Sortierung stabil, sodass die ursprüngliche Reihenfolge gleich großer Elemente beibehalten wird.

Sortieralgorithmen

Insertionsort: Laufzeit

```
public static int[] insertionSort(int[] unsortedArray) {
    int temp = 0;
    //Äußere Schleife wählt das nächste zu vergleichende
    for (int i = 0; i < unsortedArray.length; i++) {
        temp = unsortedArray[i];
        int k = i;
        //Innere Schleife identifiziert die korrekte Posi
        // sortierten Bereich
        while(k>0  &&  unsortedArray[k-1]>temp){
            unsortedArray[k] = unsortedArray[k-1];
            k--;
        }
        unsortedArray[k] = temp;
    }
    return unsortedArray;
}
```



- **Worst-Case**: Äußere und innere Schleife wird durchlaufen und jedes Element wird getauscht:
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$
- **Avg-Case**: Äußere Schleife wird ganz aber innere Schleife nur für die Hälfte der Elemente durchschritten
$$\mathcal{O}(n * \frac{n-1}{4}) = \mathcal{O}(n^2)$$
- **Best-Case**: Äußere Schleife wird ganz und aber innere Schleife nie durchschritten
$$\mathcal{O}(n-1) = \mathcal{O}(n)$$

Befassen wir uns zum Abschluss mit der Laufzeitanalyse von Insertionsort. Die äußere Schleife durchschreitet einmal komplett den Array um das nächste zu vergleichende Element auszuwählen. Die Vergleiche für die innere Schleife hängen von der Ausgangssortierung des Arrays ab. Im Worst-Case liegen die Elemente in inverser Reihenfolge zur Zielreihenfolge vor, sodass jedes Element getauscht werden muss. In Folge dessen muss die innere Schleife stets komplett durchlaufen werden sodass insgesamt n*(n-1)/2 -> O(n2) Vergleichsoperationen notwendig sind. Im durchschnittlichen Fall dagegen, wenn die Elemente etwas sortierter vorliegen muss im Durchschnitt nur die Hälfte der Vergleiche im sortierten Teil vorgenommen werden sodass n*(n-1)/4 -> O(n2). Im besten Falle liegt der Array bereits sortiert vor, sodass die innere Schleife nie durchlaufen wird und in Folge dessen nur n-1 -> O(n) Vergleiche notwendig sind.

selectiomsort mit 2 1er

Sortieralgorithmen

Selectionsort: Grundidee

- Auch der **Selectionsort** Algorithmus unterscheidet zwischen einem **sortierten** und einem **unsortierten** Bereich des Arrays.
- In jedem Durchlauf wird das **kleinste Element des unsortierten Bereichs identifiziert** und an die letzte Stelle des sortierten Bereichs gesetzt.
- **Mit jedem Durchlauf wächst** der **sortierte Bereich** und der unsortierte Bereich schrumpft.
- Im Vergleich zu den bisherigen Sortierverfahren wird hier **mehr Aufwand in die Auswahl des zu vergleichenden Elements** gesteckt.

Der letzte elementare Sortieralgorithmus mit dem wir uns befassen ist Selectionsort. Ähnlich zum Insertionsort Algorithmus unterteilt Selectionsort den vorliegenden Array in einen sortierten und einen unsortierten Abschnitt. Zu Beginn ist der unsortierte Bereich leer. In jedem Durchlauf wird nun da kleinste Element des unsortierten Bereichs identifiziert und auf die aktuelle Stelle im sortierten Abschnitt gesetzt. In Folge dessen wächst mit jedem Durchlauf der sortierte Teil des Arrays und der unsortierte Teil wird kleiner. Vergleicht man dieses Vorgehen mit der Funktionsweise von Insertionsort fällt auf, dass bei Selectionsort mehr Aufwand in die Identifikation des kleinsten Elements im unsortierten Bereich gelegt wird wodurch kein Aufwand für die Positionsbestimmung im sortierten Abschnitt entsteht. Im Vergleich hierzu, hat Insertionsort kein Aufwand für die Identifikation des nächsten Elements dafür aber für die Einordnung des entsprechenden Elements in den sortierten Bereich des Arrays.

Sortieralgorithmen

Selectionsort: Implementierung

- Die **äußere Schleife** bestimmt die **aktuelle Position im sortierten** Bereich.
- Die **innere Schleife** identifiziert das **kleinste Element** im unsortierten Bereich.
- Der **Algorithmus** benötigt nur **ein Feld (temp) als Zwischenspeicher** (konstanter Speicherverbrauch).
- **Die Sortierung ist nicht stabil**, d.h. sie behält nicht die ursprüngliche Ordnung gleichartiger Elemente bei.

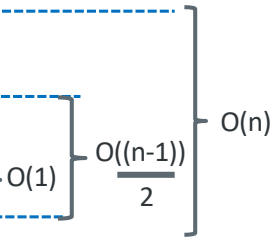
```java
public static int[] selectionSort(int[] unsortedArray){
    int temp = 0;
    //Äußere Schleife identifiziert die aktuelle Position im
    //sortierten Bereich
    for (int i=0;i<unsortedArray.length-1;i++){
        //Innere Schleife itentifiziert das kleinste Element im
        //unsortierten Bereich
        for(int k=i+1;k<unsortedArray.length;k++){
            if(unsortedArray[i] > unsortedArray[k]){
                temp = unsortedArray[i];
                unsortedArray[i] = unsortedArray[k];
                unsortedArray[k] = temp;
            }
        }
    }
    return unsortedArray;
}
```


Schauen wir uns nun die Implementierung von Selectionsort an. Auch hier haben wir es mit zwei Schleifen zu tun. Die äußere Schleife durchläuft den Array um die aktuell zu besetzende Position im sortierten Teil des Arrays zu bestimmen. Die innere Schleife durchläuft den unsortierten Teil des Array und identifiziert das kleinste Element in diesem Bereich. Wird ein Element identifiziert das kleiner ist als der Wert auf der aktuellen Position, wird ein Tausch mithilfe des Zwischenspeichers (temp) durchgeführt. Die Sortierung von Selectionsort ist nicht stabil, da die ursprüngliche Ordnung gleichartiger Elemente nicht beibehalten wird.

Sortieralgorithmen

Selectionsort: Laufzeit

```
public static int[] selectionSort(int[] unsortedArray){
    int temp = 0;
    //Äußere Schleife identifiziert die aktuelle Positio
    //sortierten Bereich
    for (int i=0;i<unsortedArray.length-1;i++){
        //Innere Schleife itentifiziert das kleinste Ele
        //unsortierten Bereich
        for(int k=i+1;k<unsortedArray.length;k++){
            if(unsortedArray[i] > unsortedArray[k]){
                temp = unsortedArray[i];
                unsortedArray[i] = unsortedArray[k];
                unsortedArray[k] = temp;
            }
        }
    }
    return unsortedArray;
}
```



- **Worst-Case**: Äußere und innere Schleife werden stets ganz durchlaufen
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$
- **Avg-Case**: Äußere und innere Schleife wird ganz durchschritten, aber nur die Hälfte der Elemente getauscht
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$
- **Best-Case**: Äußere und innere Schleife wird ganz durchschritten, aber kein Element getauscht
$$\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$$

28

Befassen wir uns zum Abschluss mit der Laufzeitanalyse von Selectionsort. Die äußere Schleife durchläuft den Array zur Positionsbestimmung komplett. Die innere Schleife vergleicht alle Elemente des unsortierten Teils miteinander um das kleinste Element zu identifizieren. Abschließend wird, sofern nötig, der Elementtausch vorgenommen.

Im Worst-Case liegt als Ausgangssituation ein invers sortierter Array vor, bei dem für jedes Element ein Tausch vorgenommen werden muss. Für die Anzahl der notwendigen Vergleiche ist die Ausgangssituation jedoch irrelevant, da diese fix n*(n-1)/2 -> $O(n^2)$ betragen. Auch im durchschnittlichen Fall, wo Teile des Arrays bereits sortiert vorliegen müssen n*(n-1)/2 -> $O(n^2)$ Vergleiche gemacht werden, da für die Bestimmung des kleinsten Elements im unsortierten Teil des Arrays die vorliegende Sortierung keine Rolle spielt. In Folge dessen sind auch n*(n-1)/2 -> $O(n^2)$ Vergleiche für den Best-Case notwendig, da der Selectionsort Algorithmus die vorliegende Sortierung der Elemente ignoriert.

Sortieralgorithmen

Überblick: Lineare Sortierverfahren

- Die vorgestellten Algorithmen unterscheiden sich in Ihrer Vorgehensweise zur Sortierung aber haben alle ein **ähnliches Laufzeitverhalten**.
- **Insertionsort** hat im Falle des **Best-Case** das **beste Laufzeitverhalten**
- Alle vorgestellten Sortieralgorithmen haben einen **konstanten Speicherbedarf.**
- **Im Gegensatz zu Bubblesort** und **Insertionsort** führt **Selectionsort keine stabile** Sortierung durch.

Algorithmus	Best-Case	Avg.-Case	Worst-Case
Bubblesort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Rückblickend lässt sich zusammenfassen, das obwohl die Vorgehensweise der drei vorstellten Sortieralgorithmen sich grundlegend unterscheidet, alle ein ähnliches Laufzeitverhalten aufweisen. Einzig der Insertionsort Algorithmus benötigt im Best-Case nicht ein Laufzeitverhalten von $O(n^2)$ sondern von $O(n)$. Eine weitere Gemeinsamkeit der vorgestellten Algorithmen betrifft den benötigten Speicherbedarf. Alle Algorithmen benötigen einen konstanten Speicherbedarf. Abschließend fällt auf, dass es sich bei Bubble- und Insertionsort um stabile Sortieralgorithmen handelt während Selectionsort keine stabile Sortierung durchführt.

Zur Verdeutlichung warum Selectionsort nicht stabil ist nehmen Sie die Zahlenfolge: 5 3 5‘ 1 als Beispiel. Bereits im ersten Schritt wird hier 5 und 1 miteinander getauscht, wodurch fortan 5‘ vor 5 steht und so die ursprüngliche Ordnung verletzt wird.

Agenda

Suchalgorithmen

Elementare Strategien zum Suchen von Elementen in Listen.

Elementare Sortieralgorithmen

Elementare Strategien zum Sortieren von Listen.

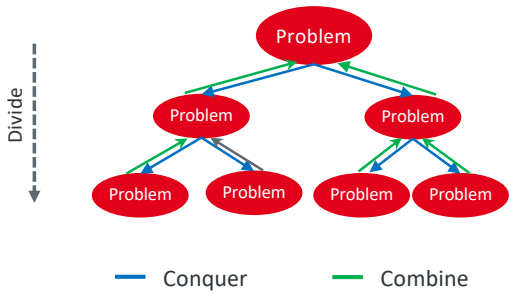
Rekursive Sortier-algorithmen

Rekursive Strategien zum Sortieren von Listen.

Rekursive Sortieralgorithmen

Grundlagen

- Aufgrund unnötiger Vergleiche werden bei den elementaren Sortieralgorithmen im **Avg.- und Worst Case O(n^2) Vergleiche benötigt**.
- **Rekursive Sortieralgorithmen** nutzen das **Divide & Conquer** Prinzip:
  - **Divide**: Zerlege das Gesamtproblem in kleinere Probleme.
  - **Conquer**: Bearbeite die Aufgabe durch rekursiven Selbstaufruf.
  - **Combine**: Kombiniere die Lösungen der kleinsten Ebenen um eine Lösung für das Gesamtproblem zu ermitteln.



Bei rekursiven Sortieralgorithmen wird der gesamte Array solange zerlegt und bearbeitet (**divide & conquer**) bis nur noch einzelne Element vorliegen, die anschließend kombiniert werden (**combine**).

Wie wir im letzten Abschnitt gesehen haben führen die unnötigen Vergleiche dazu, dass die elementaren Sortieralgorithmen im Worst- und Avg. Case Vergleiche in der Größenordnung von O(n^2) benötigen. Um dieses Laufzeitverhalten zu schlagen setzen die rekursiven Sortieralgorithmen auf das sogenannte Divide&Conquer Prinzip. Dieses Prinzip besteht aus drei grundlegenden Schritten.

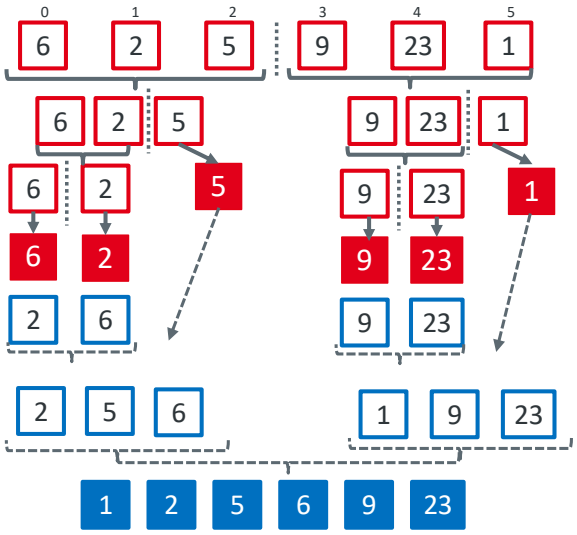
- Divide -> Solange wie möglich wird das zu bearbeitende Problem in kleinere, voneinander getrennt bearbeitbare, Teilprobleme unterteilt.
- Conquer -> Bearbeitung der Teilprobleme durch rekursiven Selbstaufruf
- Combine -> Nachdem das Problem auf elementarster Ebene (dem einzelnen Element) gelöst wurde, werden die Teillösungen zur Gesamtlösung kombiniert

Gute Literatur zu rekursiven Sortieralgorithmen:
https://medium.com/basecs/tagged/sorting-algorithms

Rekursive Sortieralgorithmen

Mergesort: Grundidee

- **Mergesort** zerteilt die Ausgangsliste in der Divide & Conquer Phase solange rekursiv bis nur noch einzelne Elemente vorliegen.
- Hierzu wird die Ausgangsliste an der **mittleren Position** (Ganzzahl) in zwei Teillisten geteilt.
- In der **Combine-Phase** werden die Elemente der zwei Teillisten durch paarweisen Vergleich miteinander zu einer sortierten Teilliste zusammengesetzt.



Der erste rekursive Sortieralgorithmus, den wir uns genauer anschauen ist Mergesort. Gemäß dem Divide & Conquer Prinzip teilt Mergesort die Ausgangsliste solange in Teillisten auf, bis sie nur noch aus einem einzelnen Element bestehen. Hierzu wird die Ausgangsliste an der mittleren Position (Ganzzahl von (linke Grenze + rechte Grenze) / 2) in zwei voneinander getrennte Teillisten aufgeteilt. Dieser Schritt wird solange wiederholt, bis alle Listen nur noch aus einem Element bestehen. In der Combine-Phase werden die einzelnen Listen anschließend durch paarweisen Vergleich der Elemente zu einer sortieren Liste zusammengesetzt.

Rekursive Sortieralgorithmen

Mergesort: Implementierung

- Solange die **rechte Grenze größer als die linke Grenze ist**, erfolgt ein **Rekursionsaufruf**.
- Für jeden Rekursionsschritt wird ein Array mit der gleichen Länge benötigt (vgl. tmp)
- Nachdem nur noch ein einzelnes Element vorliegt erfolgt rückwirkend durch **paarweisen Vergleich** die Zusammensetzung der Teillisten.
- Der **temporäre Speicher** (tmp[]) wird benötigt, damit die geordnete Reihenfolge in das original Array geschrieben werden kann.

```java
public static void mergeSort(int[] unsortiert, int linkeGrenze, int rechteGrenze) {
    int i, j, k, mitte;
    int[] tmp = new int[unsortiert.length];
    //Wenn mehr als ein Element eingegrenzt
    if (rechteGrenze > linkeGrenze) {
        mitte = (linkeGrenze + rechteGrenze) / 2;
        mergeSort(unsortiert, linkeGrenze, mitte);
        mergeSort(unsortiert, linkeGrenze: mitte+1, rechteGrenze);
        // alle Werte ins Feld b kopieren:
        // tmp = u[l],...,u[mitte],u[r],...u[mitte+1]
        for (i = mitte; i >= linkeGrenze; i--)
            tmp[i] = unsortiert[i];
        for (j = mitte+1; j <= rechteGrenze; j++)
            tmp[rechteGrenze + mitte + 1 - j] = unsortiert[j];
        // dann die größten Elemente der Teilliste absteigend
        // miteinander vergleichen. Durch die inverse Anordnung der rechten Hälfte
        // entfällt die Überprüfung ob i und j die Grenzen der Teilarrays überschreiten.
        i = linkeGrenze; j = rechteGrenze;
        for (k = linkeGrenze; k <= rechteGrenze; k++) {
            if (tmp[i] < tmp[j]) {
                unsortiert[k] = tmp[i];
                i++;
            } else {
                unsortiert[k] = tmp[j];
                j--;
            }
        }
    }
}
```

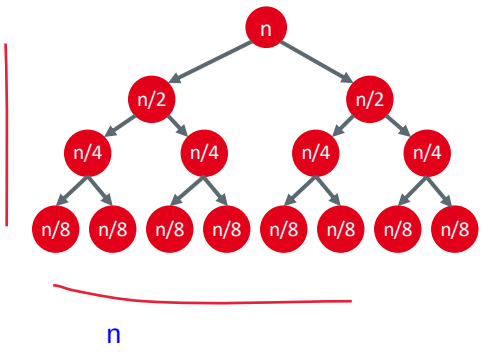

Betrachten wir im Folgenden die Implementierung von Mergesort. Die Grundannahme hierbei ist, dass in der zu sortierenden Liste mindestens ein Element vorhanden ist, d.h.: mindestens ein Element befindet sich zwischen linker und rechter Grenze. Ist dies der Fall, wird die mittlere Position des Arrays anhand der Ganzzahldivision durch 2 identifiziert, wobei für die beide Teilhälften ein weiterer Rekursionsaufruf ausgeführt wird. Sobald nur noch ein Element vorliegt, werden die einzelnen Elemente durch paarweisen Vergleich miteinander in eine sortierte Liste überführt. Zur Durchführung des Algorithmus wird pro Rekursionsschritt ein weiterer Array als Zwischenspeicherort benötigt.

Rekursive Sortieralgorithmen

Mergesort: Laufzeit

- Durch **Verwendung** des **mittleren Elements** als Trennelement, entsteht ein balancierter Binärbaum.
- In der **Combine-Phase** werden pro Ebene des Binärbaums **n-Vergleiche für die Zusammensetzung der Listen** benötigt.
- Ein **Binärbaum** hat eine Tiefe von **log(n+1) -1** sodass eine **Gesamtkomplexität von O(n* log(n))** entsteht
- Die **Gesamtkomplexität O(n log n)** besteht für den **Best-, Avg.- und Worst-Case**.

log(n)+1

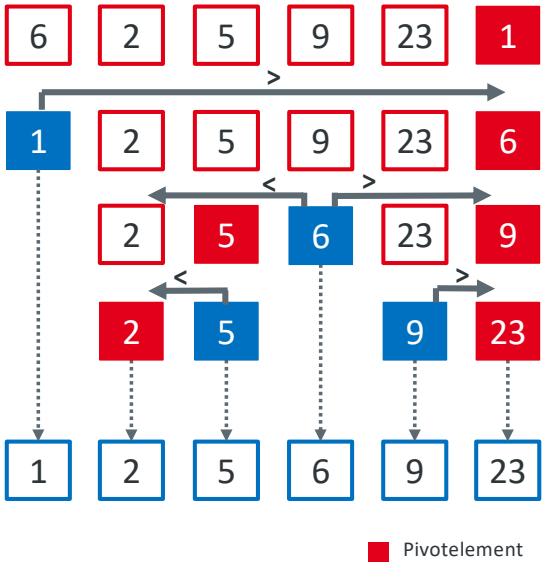


Dadurch, dass die Aufteilung der Ausgangsliste stets am mittleren Element stattfindet ist der dadurch entstehende Aufrufbaum ein balancierter Binärbaum. Dies ist insofern relevant, da die Zerteilung der Gesamtliste in Teilprobleme nur der erste Schritt ist. Sobald nur noch Blätter vorliegen werden diese dann in der Combine-Phase durch paarweisen Vergleich miteinander wieder zusammengesetzt. Hierzu sind pro Ebene des Aufrufbaums n-Vergleiche notwendig. Wie wir bereits aus dem ersten Kapitel der Vorlesung wissen, hat ein Binärbaum eine Tiefe von log(n) +1 sodass eine Gesamtkomplexität der notwendigen Vergleiche von O(n log n) folgt. Diese Gesamtkomplexität gilt sowohl für Best- Avg. und Worst-Case da sie unabhängig von der Sortierung der Ausgangsliste ist.

Rekursive Sortieralgorithmen

Quicksort: Grundidee

- **Quicksort zerteilt und ordnet** in der **Divide-Phase** die Ausgangsliste in zwei Teillisten in Bezug auf das **Pivotelement**.
- In der **Conquer-Phase** wird der Algorithmus **rekursiv auf die Teillisten angewendet** bis jene nur noch aus einem Element bestehen.
- In der **Combine-Phase** werden die Elemente **ohne Vergleich zusammengesetzt**.



Das zweite rekursive Sortierverfahren das wir uns anschauen ist Quicksort. Im Gegensatz zu Mergesort sortiert dieser Algorithmus die Teillisten in der Divide-Phase, sodass die einzelnen Elemente in der Combine-Phase nur noch zusammengesetzt werden müssen. Konkret funktioniert der Algorithmus wie folgt:

- In der Divide-Phase wird ein Pivotelement ausgewählt, das anschließend mit allen Elementen des Arrays verglichen wird. Jene Elemente die kleiner als das Pivotelement sind werden in die erste Teilliste eingefügt und jene Elemente die größer als das Pivotelement sind kommen in die zweite Teilliste.
- In der Conquer-Phase wird der Algorithmus rekursiv so lange auf die Teillisten angewendet, bis sie nur noch aus einem Element bestehen.
- Abschließend werden die Elemente in der Combine-Phase ohne Vergleich zusammengesetzt um eine sortierte Liste zu erhalten.

Der Clou von Quicksort ist der Wegfall des paarweisen Vergleichs mit jedem Element beim Zusammensetzten der sortierten Liste. Obwohl die Teillisten je Rekursionsschritt nicht komplett sortiert werden funktioniert das einfache Zusammensetzten in der Combine-Phase, da die Elemente in Relation zum Pivotelement in der richtigen Reihenfolge sind.

Rekursive Sortieralgorithmen

Quicksort: Implementierung

- Solange die **rechte Grenze größer als die linke Grenze ist** wird der Algorithmus ausgeführt.
- Das Array wird **inplace so sortiert** dass die Werte im **linken Teilbereich kleiner** und im **rechten Teilbereich größer** als das Pivotelement sind.
- Der Algorithmus benötigt **pro Rekursions-schritt nur 3 Variablen.**
- Der Algorithmus ist **nicht stabil.**

```java
public static void quickSort (int[] unsorted, int linkeGrenze, int rechteGrenze) {
    int l, r, pivot;
    //Bearbeite den Array nur wenn mehr als ein Element zwischen linker
    //under rechter Grenze liegt
    if (linkeGrenze < rechteGrenze) {
        pivot = unsorted[rechteGrenze];
        l = linkeGrenze;
        r = rechteGrenze-1;
        do {
            //geht die linke Seite durch und bricht ab sobald ein Element
            // größer als das Pivotelement ist
            while (unsorted[l] < pivot && l < rechteGrenze) l++;
            //geht die rechte Seite durch und bricht ab sobald ein Element
            // kleiner als das Pivotelement ist
            while (unsorted[r] > pivot && r > linkeGrenze) r--;
            //Vertausche die Position der Elemente
            if (l < r)
                vertausche(unsorted, l, r);
        } while (l < r);
        //Pivotelement an die richtige Position
        vertausche(unsorted, l, rechteGrenze);
        //Wende Quicksort auf die linke und rechten Teil an
        quickSort(unsorted, linkeGrenze, rechteGrenze: l-1);
        quickSort(unsorted, linkeGrenze: l+1, rechteGrenze);
    }
}
```


Im Folgenden schauen wir uns genauer die Java Implementierung von Quicksort an. Die Startbedingung für den Algorithmus ist, dass die Spannweite zwischen linker und rechter Grenze mehr als ein Element umfasst. Ist dies der Fall sucht die erste while-Schleife nach einem Element auf der linken Seite das größer als das Pivotelement ist und in der zweiten while Schleife nach einem Element auf der rechten-Seite das kleiner als das Pivotelement ist. Anschließend wird die Position dieser beiden Elemente miteinander getauscht. Dieser Schritt wird solange wiederholt bis alle Elemente der Teillisten mit dem Pivotelement verglichen wurden, sodass sich die zwei Laufindexe (l und r treffen). Im Anschluss daran wird Quicksort auf die zwei gebildeten Teillisten angewendet.

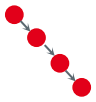
Quicksort benötigt pro Rekursionsschritt nur 3 Variablen, wodurch der Speicherbedarf minimal ist. Es gilt jedoch zu beachten, dass Quicksort kein stabiles Sortierverfahren ist.

Rekursive Sortieralgorithmen

Quicksort

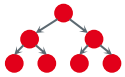
Die Anzahl der Rekursionsaufrufe, hängt von der Sortierung der Inputdaten ab.

- **Worst-Case**: Invers geordneter Array. Vollkommen unausgeglichener Aufrufbaum



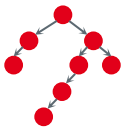
$\mathcal{O}$(n-1 + n-2 + n-3 ...) = $\mathcal{O}(n * \frac{n-1}{2}) = \mathcal{O}(n^2)$

- **Best-Case**: Die Größe der Teillisten sind perfekt ausgeglichen -> Balancierter Binärbaum



$\mathcal{O}(n\ (log(n) + 1)) = \mathcal{O}(n\ log\ n)$

- **Avg-Case**: Unbalancierter Binärbaum, der aber nicht vollkommen unausgeglichen ist



$\mathcal{O}(n\ log\ n)$

Betrachten wir abschließend das Laufzeitverhalten von Quicksort. Im Gegensatz zu Mergesort ist das Laufzeitverhalten von Quicksort abhängig von der Sortierung der Ausgangsdaten. Im Worst Case liegt ein invers geordneter Array vor, sodass das Pivotelement entweder das größte oder das kleinste Element in der Liste ist. In Folge dessen entsteht als Aufrufstruktur kein balancierter Baum sondern eine verkette Liste, sodass insgesamt $O(n^2)$ Vergleiche notwendig sind.

Im Best-Case wird das Pivotelement so gewählt, dass ein balancierter Binärbaum mit Tiefe log(n) +1 entsteht. Wie schon bei Mergesort sind hier pro Ebene n Vergleiche notwendig, sodass hieraus ein Laufzeitverhalten von O(n log n) resultiert.

Abschließend kann gezeigt werden, dass das Laufzeitverhalten von O(n log n) auch im Falle von unbalancierten Binärbaumen gilt, also wenn das Pivotelement nicht stets gleichgroße Teillisten produziert.

Rekursive Sortieralgorithmen

Überblick

-funktionalität von arrays copy of range angucken
anpassung des quicksorts aufgabe 6d -> Baustein
rechtegrenze war nicht includiert
-Elementare sortierverfahren
- Formel für interpolare
- doppelte hasching angucken, i bei kollisionen um 1 erhöhen
- quicksort, pivotelemt immer rechteste element

Laufzeit:

- **Mergesort** erzieht **durchgängig** eine Laufzeit von **O(n log n)** Vergleichen.
- **Quicksort** operiert **abhängig** von der Struktur der Elemente und hat im **Worst-Case** eine Laufzeit von **$O(n^2)$**

Speicher:

- **Mergesort** benötigt **O(log n) zusätzliche Arrays. Quicksort** dagegen nur **einzelne Felder**

Stabilität:

- **Mergesort ist stabil** während **Quicksort nicht stabil** ist.

Algorithmus	Best-Case	Avg.-Case	Worst-Case
Quicksort	O(n*log(n))	O(n*log(n))	$O(n^2)$
Mergesort	O(n*log(n))	O(n*log(n))	O(n*log(n))

38

Stellen wir zum Abschluss die zwei besprochenen rekursiven Sortieralgorithmen gegenüber. Bezüglich der Laufzeit fällt auf, dass Mergesort unabhängig von der Struktur und Sortierung der Ausgangsdaten operiert und so stets eine Laufzeit von O(n log n) Vergleichen erzielt. Die Laufzeit von Quicksort ist dagegen abhängig von der Struktur der Ausgangsdaten und im Worst-Case werden sogar $O(n^2)$ Vergleiche benötigt.

Hinsichtlich des benötigten Speicherbedarfs fällt auf, dass Quicksort deutlich effizienter operiert. Während bei Mergesort O(log n) zusätzliche Arrays für das Zusammensetzen der einzelnen Elemente benötigt werden, werden bei Quicksort nur drei weitere Felder pro Rekursionsschritt benötigt.

Abschließend handelt es sich bei Mergesort um ein stabiles Sortierverfahren während Quicksort kein stabiles Sortierverfahren darstellt.

Aufgabe 6

Elementare Sortieralgorithmen

a) Sortieren Sie die nebenstehende Zahlenfolge mittels **Insertionsort** und **Selectionsort** in aufsteigender Reihenfolge. Geben Sie den Array nach jeder Swap-Operation an.

2	2	9	1	5

b) Welcher der vorgestellten elementaren Sortieralgorithmen ist am besten für die Datenstruktur einer verketteten Liste geeignet?

c) Sortieren Sie die nebenstehende Zahlenfolge in aufsteigender Reihenfolge mittels **Mergesort** und **Quicksort (**Pivotelement ist **linkestes Element)**. Geben Sie die Zerlegung und Zusammensetzung des Arrays samt aller Zwischenschritte an

Aufgabe 6

Rekursive Sortieralgorithmen

d) Ändern Sie den **Quicksort** Algorithmus in der Form ab, dass dieser nicht mehr inplace (auf dem gleichen Array) operiert, sondern für jeden Rekursionsschritt eine sortierte Teilliste zurückgibt.

Hierzu soll die Methodensignatur wie folgt lauten:

```
public static int[] quickSort2 (int[] unsorted)
```