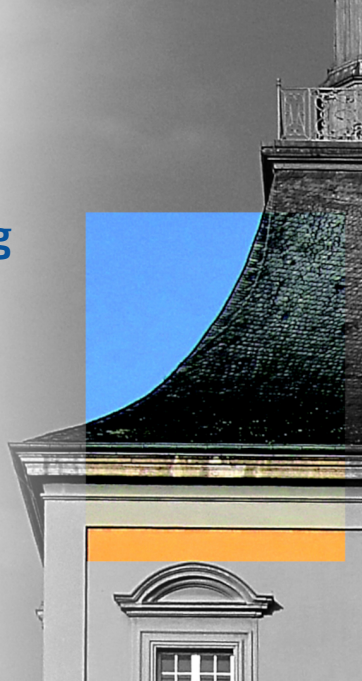
UNIVERSITÄT BONN

Algorithmen und Programmierung

Algorithmen I

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



UNIVERSITÄT BONN

Literaturempfehlungen

Es gibt sehr viele gute Bücher zu Algorithmen im Allgemeinen sowie zu Kategorien von Algorithmen im Speziellen. Als Einstieg in Algorithmen im Allgemeinen empfehlen wir sowohl für dieses Kapitel als auch für folgende Kapitel:

- Anfängerfreundliche Primärquelle: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. Introduction to Algorithms. MIT press, 2009.
- Für Interessierte: Udi Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley Longman Publishing Co., Inc., 1989.

Sortieren, Rekursion und Landausymbole

Überblick

Sortierverfahren sind grundlegende Algorithmen und die einfachsten, nicht trivialen Beispiele.

Wir beschreiben Sortierverfahren am Beispiel, als Algorithmus, betrachten deren Implementierung und schätzen die Kosten in Abhängigkeit der Größe des Inputs ab

Sie lernen in den folgenden Abschnitten Sortierverfahren, Rekursion und Landausymbole kennen

Sortieren, Rekursion und Landausymbole

Sortierverfahren

Offene Fragen

Wie sortiert man ein Array von Objekten (effizient)?

Wie notiert man einen Algorithmus?

Welche Schritte sind nötig um zu zeigen, dass ein Verfahren ein Algorithmus ist?

UNIVERSITÄT BONN

Einleitung

Ein wiederkehrendes Problem ist das Sortieren von Objekten. Zu diesem Zweck existieren viele Sortierverfahren. Beispiele sind **Selection Sort**, **Quick Sort** und **Merge Sort**.

In dieser Vorlesung lernen Sie **Min Sort** (eine Variante von Selection Sort) und **Merge Sort** kennen.

Didaktisch höchst wertvolle Visualisierungen weiterer Sortierverfahren finden Sie hier:

`https://www.youtube.com/user/AlgoRythmics/videos`

Sortieren, Rekursion und Landausymbole

Sortierverfahren - Min Sort

Wir sortieren ein Array indem wir aufsteigend jeweils ein kleinstes Element auswählen

Offene Fragen

Wie notiert man einen Algorithmus?

Welche Schritte sind nötig um zu zeigen, dass ein Verfahren ein Algorithmus ist?

UNIVERSITÄT BONN

Einleitung

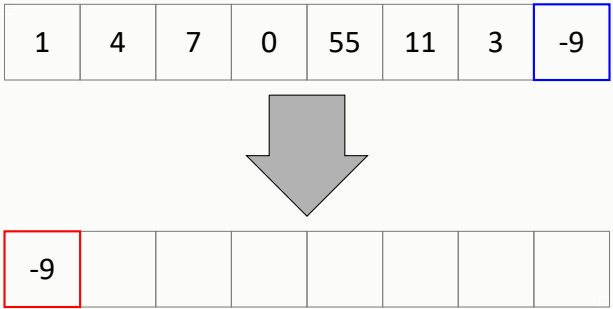
Erinnern Sie sich an das zuerst eingeführte Sortierverfahren dieser Vorlesung. Ähnlich zu diesem wollen wir nun ein Verfahren betrachten, bei dem sukzessive das aktuell kleinste Element eines Arrays gefunden wird, um anschließend nach vorn sortiert zu werden.

Hier gibt es die Unterscheidung ob man die Elemente des Arrays innerhalb des Arrays umsortieren soll, oder ob man dazu ein neues Array anlegen soll. Die erste Variante nennt man **in place**, die zweite Variante nennt man **out of place**. Wir betrachten hier beide Varianten.

UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

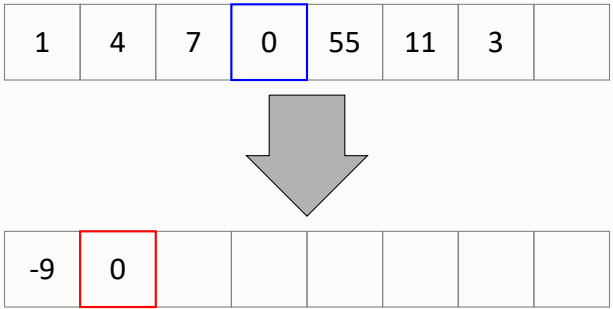
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

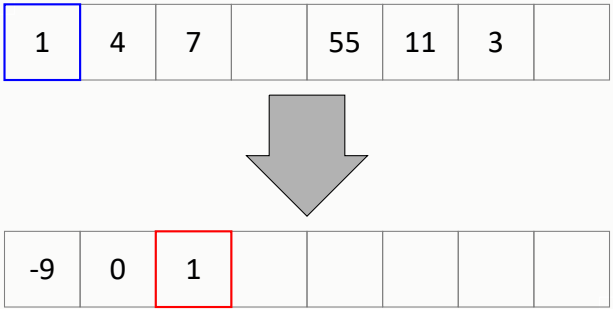
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

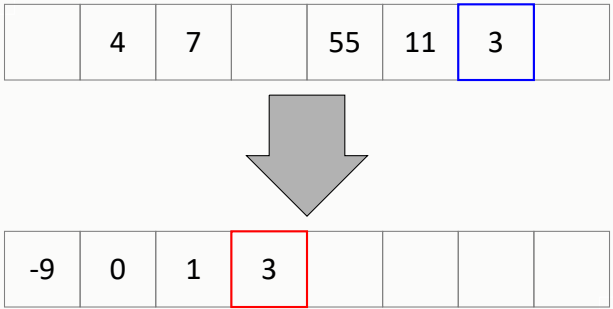
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

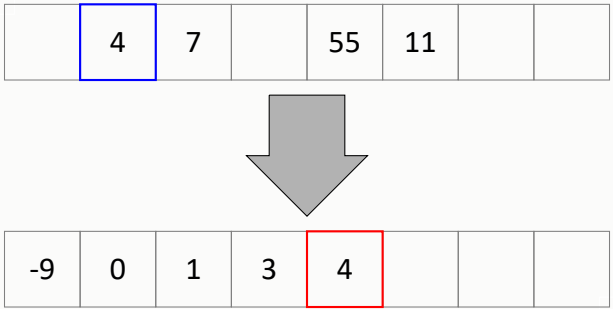
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

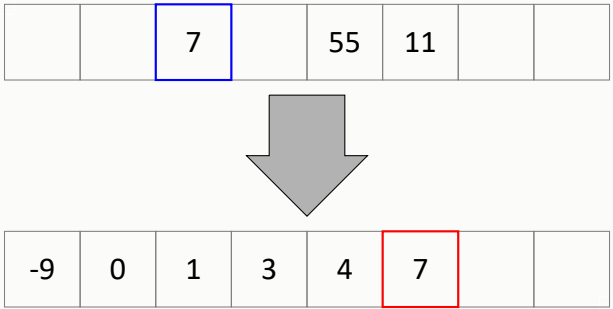
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

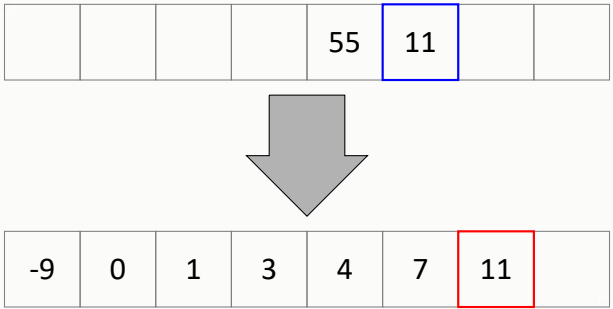
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

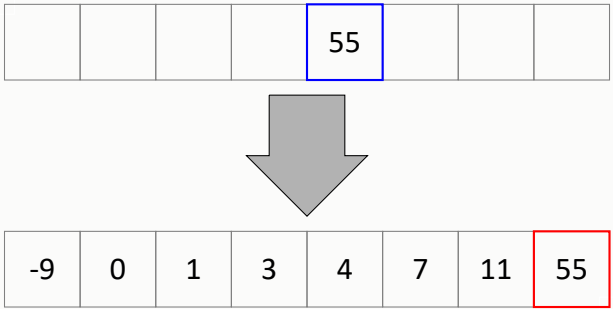
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place) Beispiel

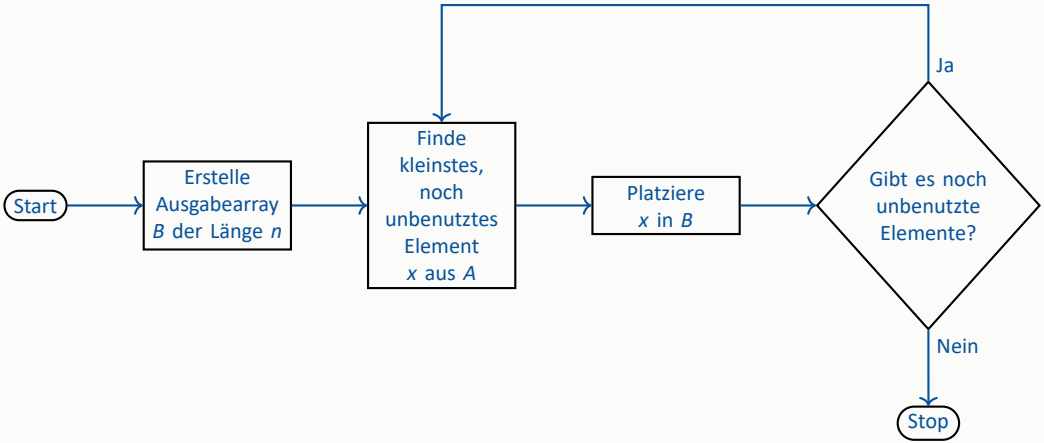
Bei **Min Sort (out of place)** platziert man das aktuell kleinste Element an die nächste freie Stelle in dem neuen Array.



UNIVERSITÄT BONN

Min Sort (out of place)
Vorüberlegung

Schlussendloch wollen wir den Algorithmus textuell notieren. Zur Hilfestellung beschreiben wir das Verfahren zuerst durch ein Ablaufdiagramm.



UNIVERSITÄT BONN

Min Sort (out of place)
Algorithmus I

Der Algorithmus wird wie folgt beschrieben.

Algorithm: MinSort (out of place)

Eingabe : Array $\mathbb{A} = [A_0, \ldots, A_{n-1}]$ von untereinander vergleichbaren Objekten
Ausgabe : Array $\mathbb{B} = [B_0 \leq B_1 \leq \ldots \leq B_{n-1}]$ welches die Elemente des Arrays $\mathbb{A}$ (als Kopie) enthält

1 Erstelle neues Array $\mathbb{B}$ der Länge n
2 Def. (Multi)menge $\mathbb{S} = \{A_0, \ldots, A_{n-1}\}$
3 **for** $i = 0, \ldots, n-1$ **do**
4 | Finde $A_k \in \mathbb{S}$ minimal
5 | Entferne A_k aus $\mathbb{S}$ und platziere es in B_i.
6 **return** $\mathbb{B}$

Wir müssen noch zeigen, dass das beschriebene Verfahren das gestellte Problem löst.

UNIVERSITÄT BONN

Min Sort (out of place) Algorithmus II

Zunächst überlegen wir uns, ob alle Schritte zulässig sind.

Der Schleifenkörper wird n mal durchlaufen.

- In jedem Schleifendurchlauf wird genau ein Element A_k der Multimenge gewählt und entfernt. Die Multimenge hat zu Beginn genau n Elemente. Damit ist diese Operation zulässig.
- In jedem Schleifendurchlauf $0 \leq i < n$ wird ein Element A_k in dem Array $\mathbb{B}$ an der Stelle i platziert. Das Array $\mathbb{B}$ hat Platz für n Elemente. Damit ist diese Operation zulässig.

Damit sind alle Schritte zulässig.

UNIVERSITÄT BONN

Min Sort (out of place) Algorithmus III

Wir zeigen nun, dass das gestellte Problem gelöst wird. Wir betrachten dazu jeden Schleifendurchlauf.

- Jedes Element $A_0, \ldots, A_{n-1}$ wird in Zeile 4 genau einmal ausgewählt, aus $\mathbb{A}$ entfernt und an eine vorher unbenutzte Stelle in $\mathbb{B}$ platziert. Also enthält $\mathbb{B}$ am Ende des Verfahrens die Elemente des Array $\mathbb{A}$.
- Bei jedem Schleifendurchlauf $0 \leq i < n$ wird ein Element aus $\mathbb{A}$ ausgewählt welches mindestens so groß ist wie die Elemente $B_0, \ldots, B_{i-1}$. (Denn falls das nicht richtig ist, hätte es schon zu einem früheren Zeitpunkt in Zeile 4 ausgewählt werden müssen). Damit gilt zum Ende des Schleifendurchlaufs i schon $B_0 \leq \ldots \leq B_i$.

Damit erzeugt das Verfahren ein Array $\mathbb{B} = [B_0 \leq B_1 \leq \ldots \leq B_{n-1}]$ welches die Elemente des Arrays $\mathbb{A}$ enthält. Also löst das Verfahren das beschriebene Problem und ist somit ein Algorithmus.

Zur Beweisstrategie

Auf der vorhergehenden Folie haben wir eine oft verwendete **Beweisstrategie** angewandt. Die Idee ist es eine **Invariante** zu finden, die zum Ende des Verfahrens (in)direkt zum gewünschten (Teil)ergebnis führt.

Die **Invariante** in der vorhergehenden Folie ist das Teilarray $[B_0 \leq \ldots \leq B_{i-1}]$. Dies ist ein **invariantes Teilergebnis** dass bei jedem Schleifendurchlauf **vergrößert** wird und am Ende das **gewünschte Ergebnis** liefert.

UNIVERSITÄT BONN

Min Sort (out of place) Implementierung

Wir betrachten nun eine Implementierung von Min Sort. Diese ist stark vereinfacht, da sie nur für `int`-Arrays zur Verfügung steht. Wie Sie diese Implementierung für beliebige Datentypen verallgemeinern, lernen Sie später.

Außerdem wissen wir noch nicht, wie welche Datenstruktur für (Multi)mengen verwenden. Wir verwenden hier eine Behelfslösung. In den Vorlesungen zu abstrakten Datentypen und Datenstrukturen lernen Sie einen methodisch korrekten Ansatz.

UNIVERSITÄT BONN

Min Sort (out of place) Implementierung

```
#include <vector>
// Min Sort out of place
std::vector<int> int_array_min_sort_out_of_place(const std::vector<int>& input) {
  const int laenge = input.size();
  // Array der als Lösung erzeugt wird
  std::vector<int> loesung (laenge);
  // Bereits verwendete Elemente werden markiert
  std::vector<bool> verwendet (laenge, false); // Alle Einträge sind mit false initalisiert

  // Iteriert durch das Array
  for (int i = 0; i < laenge; i++) {
        // Finde den Index des ersten, noch nicht verwendeten Element
        int min_idx;
        for (min_idx = 0; verwendet[min_idx] == true; min_idx++) {
        }
```

UNIVERSITÄT BONN

Min Sort (out of place) Implementierung

```
    // Finde ab dort den Index des kleinsten, unverwendeten Element
    for (int j = min_idx + 1; j < laenge; j++) {
      if (verwendet[j] == false and input[min_idx] > input[j]) {
          min_idx = j; // Element bei j ist unverwendet und kleiner
        }
      }
    }

    // Schreibe das kleinste, unverwendete Element in die Lösung
    loesung[i] = input[min_idx];
    // Markiere das kleinste, unverwendete Element als verwendet
    verwendet[min_idx] = true;
  }

  return loesung;
}
```

Disclaimer

Wir werden in der Vorlesung nun nicht mehr in dieser hohen Detaildichte argumentieren

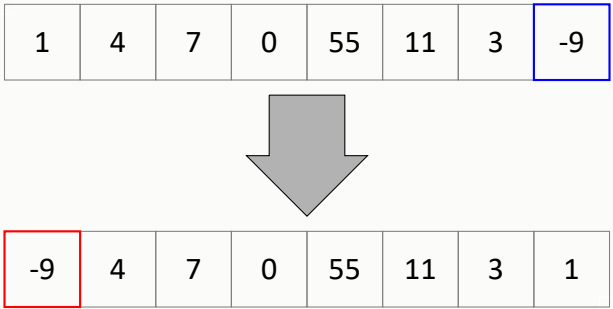
Allerdings ist es nötig, dass Sie sich selbst immer überzeugen, dass alle Schritte zulässig sind und das Verfahren wirklich das jeweils genannte Problem löst

Überzeugen Sie sich ebenfalls, dass Sie die besprochenen Algorithmen implementieren können

UNIVERSITÄT BONN

Min Sort (in place) Beispiel

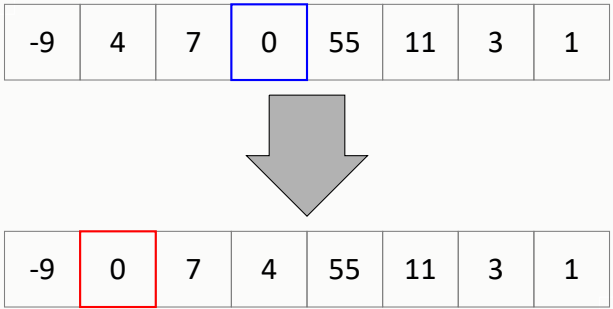
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place) Beispiel

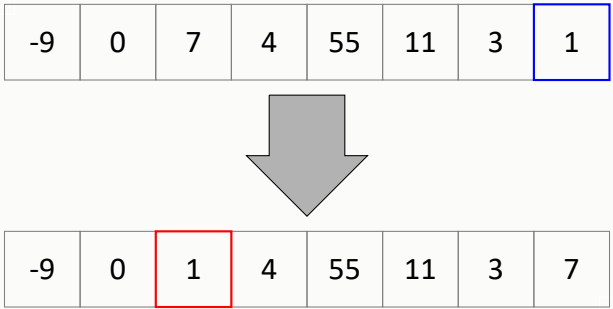
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place)
Beispiel

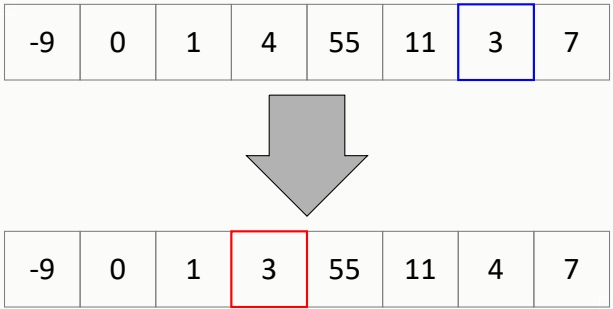
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place)
Beispiel

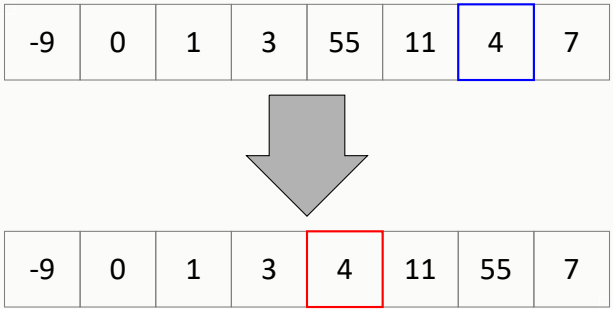
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place)
Beispiel

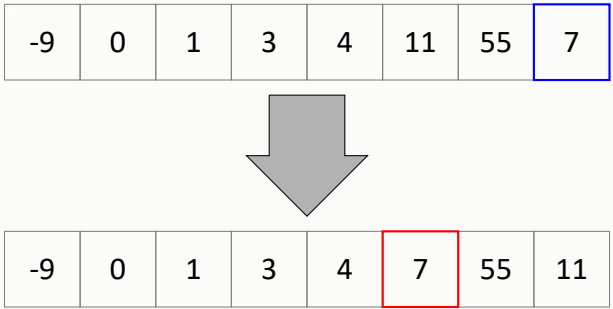
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place) Beispiel

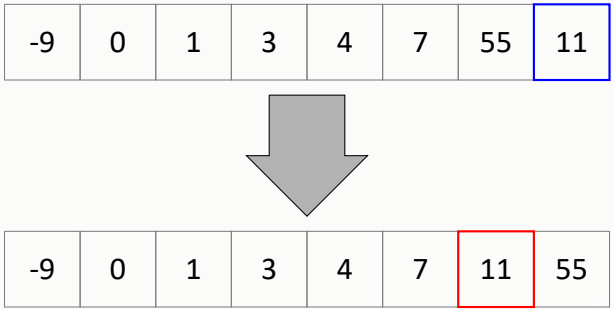
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place) Beispiel

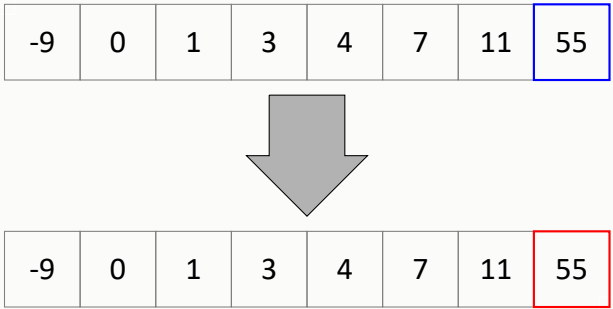
Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place) Beispiel

Bei **Min Sort (in place)** tauschen das aktuell kleinste Element und das aktuell zu besetzende Element in demselben Array die Plätze.



UNIVERSITÄT BONN

Min Sort (in place)
Algorithmus und Implementierung

Üben Sie den Umgang mit Algorithmen ein indem Sie

- das zugehörige Verfahren beschreiben (und dabei zuerst ein Ablaufdiagram verwenden falls es Ihnen hilft)
- nachweisen dass Ihr Verfahren der beschriebene Algorithmus ist und
- den Algorithmus implementieren.

Haben Sie Fragen?

Zusammenfassung

Wir sortieren ein Array indem wir aufsteigend jeweils ein kleinstes Element auswählen

Sie haben gesehen wie man Algorithmen notiert

Sie haben das Konzept der Invariante kennen gelernt

Sie wissen nun welche Schritte nötig sind um zu zeigen, dass ein Verfahren ein Algorithmus ist

Das Notieren und Argumentieren muss eingeübt werden

Sortieren, Rekursion und Landausymbole

Rekursion und Divide-And-Conquer-Verfahren

Ziel

Es gibt Probleme die sich rekursiv in selbstähnliche, kleinere Teilprobleme zerlegen lassen

Rekursive Lösungsstrategien machen sich diesem Umstand zunutze

Sie lernen hier promimente Beispiele kennen

UNIVERSITÄT BONN

Rekursion als allgemeines Konzept

Als **Rekursion** bezeichnet man die Tatsache, dass ein Objekte sich selbst als (verkleinerten) Bestandteil enthält oder dass ein Verfahren mithilfe seiner selbst definiert ist.

Beispiel:

UNIVERSITÄT BONN

Rekursive Funktionen

Wir nennen eine Funktion **rekursiv** wenn sie sich selbst beim Ausführen des Funktionskörpers aufrufen kann.

Ein triviales Beispiel für eine rekursiven Funktion ist es, die Fakultät einer natürlichen Zahl wie folgt zu definieren.

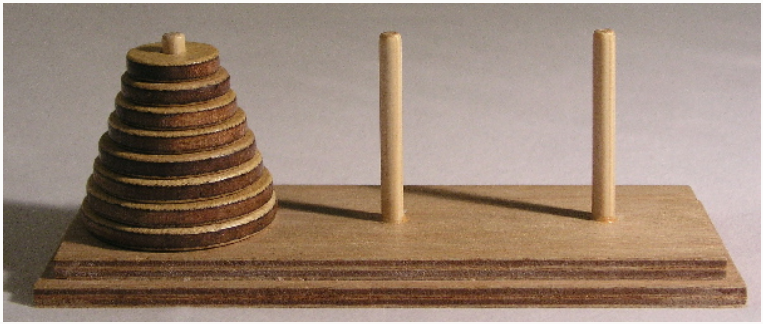
$$\begin{aligned} \mathsf{fak}(n+1) &= (n+1) \cdot \mathsf{fak}(n) \\ \mathsf{fak}(1) &= 1 \end{aligned}$$

Wir betrachten hier zwei einfache Probleme, die rekursiv gelöst werden können.

UNIVERSITÄT BONN

Türme von Hanoi I

Ein Beispiel für Rekursion ist die Lösungsstrategie bei den **Türmen von Hanoi**.



Ziel des Spiels ist es, den Turm bestehend aus *n* Scheiben von einem Pin auf einen anderen Pin in möglichst wenig Zügen zu bringen. Dabei darf immer nur eine Scheibe bewegt werden und kleinere Scheiben dürfen nicht unter größeren Scheiben gelegt werden.

UNIVERSITÄT BONN

Türme von Hanoi II

Es ist trivial das Spiel für $n = 1$ Scheiben zu lösen.

Wenn wir das Problem für eine feste Höhe n lösen können, dann können wir es auch für $n + 1$ lösen. Dazu lässt man die größte Scheibe zunächst am Platz und sortiert die restlichen n Scheiben in die Mitte. (Das ist möglich, da wir das Problem ja für die Höhe n lösen können.) Nun legt man die größte Scheibe nach rechts. Jetzt lässt man die größte Scheibe wieder fest und sortiert die restlichen n Scheiben von der Mitte nach rechts.

Stellen Sie fest, dass wir so eine **rekursive Lösung des Problems** beschrieben haben.

UNIVERSITÄT BONN

Permutationen I

Gegeben das spezielle Array $I_n = [1, 2, \ldots, n]$ der Länge n. Wir wollen alle **Permutationen** dieses Arrays berechnen. In anderen Worten wollen wir alle Möglichkeiten berechnen, die Elemente des Arrays neu zu ordnen. Die Menge der Permutationen von I_n nennen wir S_n.

Wir betrachten drei Beispiele.

$$S_1 = \{[1]\}$$
$$S_2 = \{[1, 2], [2, 1]\}$$
$$S_3 = \{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$$

UNIVERSITÄT BONN

Permutationen II

Es ist trivial das Problem für $n = 1$ lösen.

Wenn wir das Problem für ein festes n lösen können, dann können wir es auch für $n+1$ lösen. Beispiel:

$$S_3 \ni [2,1,3] \mapsto [4,2,1,3], [2,4,1,3], [2,1,4,3], [2,1,3,4] \in S_4$$

Zuerst berechnen wir S_n. Aus jeder Permutation $s \in S_n$ von I_n erstellen wir $n+1$ Permutationen von I_{n+1} wie folgt. Wir fügen in s das Symbol $n+1$ vor allen anderen Einträgen ein. Analog fügen wir in s das Symbol $n+1$ zwischen den ersten beiden Einträge ein. Diesen Vorgang wiederholen wir für alle $n+1$ möglichen Stellen.

Stellen Sie fest, dass wir so eine **rekursive Lösung des Problems** beschrieben haben.

UNIVERSITÄT BONN

Divide-And-Conquer-Verfahren I

Es gibt Probleme die sich in selbstähnliche, kleinere Teilprobleme zerlegen lassen. Da die Teilprobleme ähnlich zum eigentlichen Problem sind, können wir diese immer weiter in kleinere, selbstähnliche Teilprobleme zerlegen. Schlussendlich erreichen wir trivial lösbare Teilprobleme. Deshalb lässt sich das ursprüngliche Problem ebenfalls lösen.

Diese Lösungsstrategie nennen wir **Divide-And-Conquer**. Algorithmen die sich diese Stategie zunutze machen, nennen wir **Divide-And-Conquer-Verfahren**.

UNIVERSITÄT BONN

Divide-And-Conquer-Verfahren II

Divide-And-Conquer-Verfahren bestehen aus den folgenden drei (miteinander verwobenen) Schritten.

- **Divide** teilt das ursprüngliche Problem in kleinere Instanzen desselben Problems auf.
- **Conquer** löst die Teilprobleme rekursiv bis sie klein genug sind und trivial gelöst werden können.
- **Combine** führt die Lösungen der kleineren Probleme zusammen.

Haben Sie Fragen?

Zusammenfassung

Es gibt Probleme die sich in selbstähnliche, kleinere Teilprobleme zerlegen lassen

Rekursive Lösungsstrategien machen sich diesem Umstand zunutze

Sie kennen nun zwei promimente Beispiele

Sortieren, Rekursion und Landausymbole

Sortierverfahren - Merge Sort

Übersicht

Wir sortieren ein Array mithilfe eines Divide-And-Conquer-Verfahren

Dazu fällt uns auf, dass zwei sortierte Arrays sehr leicht zu einem sortierten Array zusammengefügt werden können.

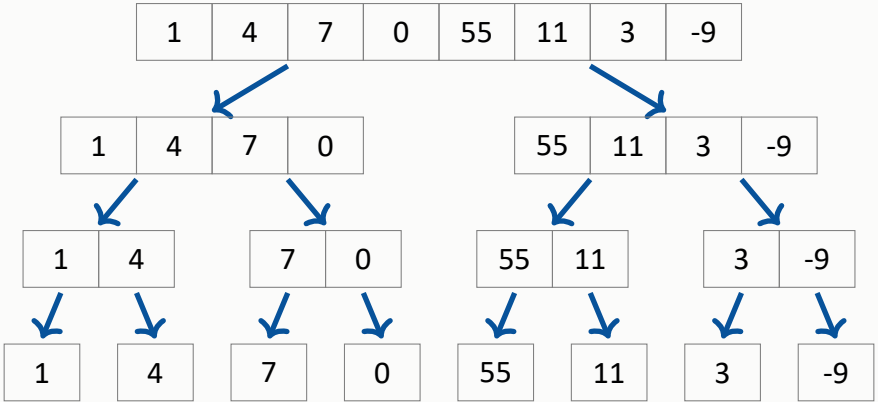
Merge Sort
Motivation

Wir beginnen mit folgender Beobachtung. Gegeben zwei **sortierte** Arrays der Länge n bzw. m. Dann lässt sich daraus schnell ein sortiertes Array der Länge $n + m$ erstellen.

Um diese Beobachtung zu nutzen fragen wir uns: Wie hilft uns das bei einem gegebenen, unsortierten Array? Wir halbieren das Array. Und halbieren sukzessive alle so entstehenden Arrays bis wir nur noch Arrays der Größe 1 haben. Diese sortieren wir wie oben beschrieben um so das gegebene Array zu sortieren.

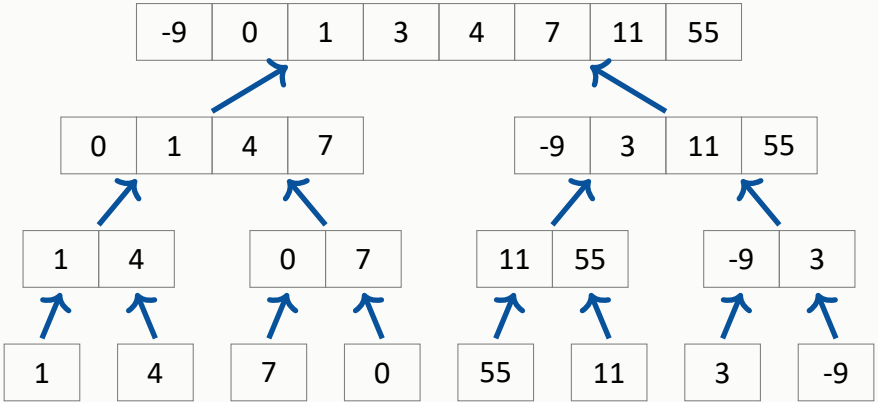
UNIVERSITÄT BONN

Merge Sort
Beispiel I



UNIVERSITÄT BONN

Merge Sort
Beispiel II



UNIVERSITÄT BONN

Merge Sort
Algorithmus (unoptimiert)

Der Algorithmus besteht aus zwei Funktionen. Die Funktion `mergesort` die das Problem rekursiv (conquer) in kleinere Teilprobleme zerlegt (divide) und die Funktion `merge` die zwei gegebene, sortierte Arrays in ein Ziel Array schreibt (combine).

Algorithm: mergesort

Eingabe : Array $\mathbb{A} = [A_0, \ldots, A_{k-1}]$ der Länge k

Ausgabe : Sortierte Kopie von $\mathbb{A}$

```
1  return mergesort(𝔸)
2  Funktion mergesort(𝕏 = [X_0, ..., X_{k-1}])
3      if k > 1 then
4          Erstelle Ganzzahl t = ⌈k/2⌉
5          𝕃 = [X_0, ..., X_{t-1}]
6          ℝ = [X_t, ..., X_{k-1}]                    // Divide-Schritt
7          𝕃_sort = mergesort(𝕃)
8          ℝ_sort = mergesort(ℝ)                      // Conquer-Schritt
9          return merge(𝕃_sort, ℝ_sort)               // Combine-Schritt
10     else
11         return 𝕏
```

Algorithm: mergesort (Fortsetzung)

Eingabe : Sortierte Arrays $\mathbb{L}$ und $\mathbb{R}$ der Länge l und r

Ausgabe : Sortiertes Array $\mathbb{Y}$ der Länge $l + r$ bestehend aus $\mathbb{L}$ und $\mathbb{R}$

```
1  Funktion merge(𝕃 = [L_0, ..., L_{l-1}], ℝ = [R_0, ..., R_{r-1}])
2      Erstelle neues Array 𝕐 der Länge l + r
3      Erstelle Gannzahlen i = 0 und j = 0
4      while i < l und j < r do
5          if L_i ≤ R_j then
6              Y_{i+j} = L_i und i++
7          else
8              Y_{i+j} = R_j und j++
9      while i < l do
10         Y_{i+j} = L_i und i++ // kopiert restl. Elemente aus L nach Y
11     while j < r do
12         Y_{i+j} = R_j und j++ // kopiert restl. Elemente aus R nach Y
13     return 𝕐
```

UNIVERSITÄT BONN

Merge Sort
Algorithmus (unoptimiert)

Üben Sie den Umgang mit Algorithmen ein indem Sie nachweisen dass alle Schritte zulässig sind und dass das beschriebene Verfahren das gewünschte Problem löst.

UNIVERSITÄT BONN

Merge Sort
Zur Implementierung

Die Implementierung des hier vorgestellten, unoptimierten Merge Sort Algorithmus ist an Sich nicht kompliziert. Aber Sie stoßen wahrscheinlich auf das folgende Problem. Beim Aufruf von der Funktion `merge` wird ein neues Array angelegt und zurückgegeben. Das verbraucht mehr Speicher als uns lieb ist.

Es gibt eine optimierte Variante von Merge Sort. In dieser wird nur zu Beginn ein Array anlegt. In diesem wird dann temporär gearbeitet. Die optimierte Variante von Merge Sort ist auf den ersten Anhieb herausfordernd zu implementieren.

Haben Sie Fragen?

Zusammenfassung

Mit Merge Sort sortieren wir ein Array indem wir das Problem mithilfe eines Divide-And-Conquer-Verfahren lösen

Dazu fällt uns auf, dass zwei sortierte Arrays sehr leicht zu einem sortierten Array zusammengefügt werden können.

Bonner Dialog für Cybersicherheit

Kommender Dienstag 17:30 - 20:30 Uhr im Hörsaal 2

`https://www.fkie.fraunhofer.de/de/Veranstaltungen/19-bdcs.html`

Community CTF für Anfänger:innen

Donnerstag 17.11. ab 18:15 Uhr in Raum 0.016

Discord Server: `https://discord.gg/cRSevKgwY4`