Inhaltsverzeichnis

Einleitung

Ein hochgestelltes „[?]“ weist auf fehlende Quellen hin. In der „Release Version“ ist es, wie dieser Absatz und die obenstehenden Hinweise, unsichtbar.

Vorerst ist es ausreichend, einen Algorithmus als eine schwarze Box (vgl. Bunge, 1963) zu betrachten, die auf eine deterministische[1] Art und Weise eine Eingangsmenge zu einer Ausgangsmenge überführt (Cormen u. a., 2001, S. 5). Ein Sortieralgorithmus überführt die Elemente seiner Eingangsmenge zu einer geordneten Ausgangsmenge[?]. (Die konkrete Ordnungsrelation ist hierbei irrelevant[?].)

Die Effizienz eines Algorithmus ist bestimmt durch seinen Verbrauch von Ressourcen[?]. Folgend wird zwischen „praktischer Effizienz“[?] und „theoretischer Effizienz“[?] eines Algorithmus unterschieden. Erstere ist empirisch zu ermitteln: Der Ressourcenverbrauch eines Algorithmus wird konkret gemessen. Letztere wird durch mathematische Analyse bestimmt.

Damit können bereits die zwei Forschungsfragen dieser Arbeit formuliert werden:

- Wie kann die „praktische Effizienz“ eines Algorithmus ermittelt werden?
- Wie verhält sich die „theoretische Effizienz“ von ausgewählten Sortieralgorithmen zu der (zu ermittelnden) „praktischen Effizienz“ jener Algorithmen?

Wie der Großteil vergleichbarer Analysen (vgl. ebd., S. 23 und Shaffer, 2011, S. 58 . . . [?]) beschäftigt sich diese Arbeit mit der Analyse der von einem Algorithmus als Ressource verbrauchte Zeit, im Folgenden auch als „Laufzeit“ bezeichnet. Wenn nicht näher angegeben beziehen sich die Begriffe „praktische“- und „theoretische Effizenz“ immer auf die Laufzeit.

Für konkrete Implementationen und Quellcode-Beispiele wird die Programmiersprache C++ (ISO/IEC 14882, 2017, „C++17“) verwendet.

[1] Alle auftretenden Zustände sind definiert und reproduzierbar, der „nächste Schritt“ ist immer eindeutig festgelegt (vgl. Bocchino u. a., 2009, S. 1).

Kapitel 1

Algorithmen

In diesem Kapitel werden die in der folgenden Arbeit behandelten Sortieralgorithmen mit ihrer theoretischen Effizienz dargelegt.

Üblicherweise ist die Effizienz eines Algorithmus proportional zur Größe der jeweiligen Eingangsmenge, deshalb wird die Effizienz im Folgenden (und im Allgemeinen auch in der Literatur[1]) als Funktion der „Eingabegröße“ $T(n)$ dargestellt.

Die Werte von $T(n)$, also die theoretische Effizienz eines Algorithmus auf einer Eingabemenge der Größe n, sind die Anzahl der „einfachen Operationen“ oder „Schritten“ welche für diese Eingabemenge ausgeführt werden müssen (vgl. Cormen u. a., 2001, S. 25 und Horowitz u. a., 1997, 18f). „Einfache Operationen“ sind hier jene Operationen deren benötigte Zeit unabhängig von den Operanden ist, beispielsweise arithmetische Grundrechenarten[2] und Vergleiche (vgl. Shaffer, 2011, S. 55). Durch diese Einschränkung kann die Anzahl der einfachen Operationen als stellvertretend zur tatsächlichen, sonst experimentell zu ermittelnden Laufzeit gesehen werden (vgl. ebd., S. 55).

Günstigster, ungünstigster und durchschnittlicher Fall Es wird zwischen der theoretischen Effizienz im „günstigsten“, „ungünstigsten“, und „durchschnittlichen Fall“ unterschieden, alle beziehen sich auf die Beschaffenheit der Eingangsmenge (vgl. Horowitz u. a., 1997, S. 28).

Zu betrachten ist ein einfacher Suchalgorithmus, Sequential-Search (vgl. Knuth, 1998, S. 396) der eine Liste nach einem Element durchsucht und terminiert nachdem er es gefunden hat.

```
SEQUENTIAL-SEARCH(A, value)
1  for i = 1 to A.length
2       if A[i] == value
3            return i
4  return 0
```

[1] Alle Werke die in dieser Arbeit zitiert werden und sich mit der Effizienz beziehungsweise Komplexität von Algorithmen beschäftigen stellen diese in Abhängigkeit von einer Eingabegröße (auch: Problemgröße) dar.

[2] Diese Aussage ist nur bedingt richtig nachdem die Zeit welche für Multiplikation und Division benötigt wird üblichwerweise von der Größe der Operanden abhängig ist (vgl. Horowitz u. a., 1997, S. 19).

Der günstigste Fall für einen solchen Algorithmus tritt auf wenn die Eingangsmenge eine Liste ist, in der das gesuchte Element an der ersten Position steht. In diesem Fall wird nur ein Vergleich ausgeführt, die Laufzeit ist kurz. Steht das gesuchte Element jedoch an der letzten Position so werden $A.length$ beziehungsweise n Vergleiche ausgeführt, eine solche Menge führt zum ungünstigste Fall. Unter der Annahme, dass die Elemente von A gleichmäßig verteilt sind, führt der Algorithmus durchschnittlich $n/2$ Vergleiche aus, dies ist der durchschnittliche Fall (vgl. Shaffer, 2011, S. 59).

Asymptotische Notation Zur einleitenden Frage, was Asymptotik sei, schreibt Bruijn in seinem Buch *Asymptotic Methods in Analysis*:

> It often happens that we want to evaluate a certain number, defined in a certain way, and that the evaluation involves a very large number of operations so that the direct method is almost prohibitive. In such cases we should be very happy to have an entirely different method for finding information about the number, giving at least some useful approximation to it. [...] A situation like this is considered to belong to asymptotics.

(Bruijn, 1958, S. 1). Die Ermittlung der exakten Anzahl an einfachen Operationen die ein Algorithmus mit einer gewissen Eingabemenge benötigt ist üblicherweise nicht lohnenswert (vgl. Horowitz u. a., 1997, S. 28) oder sogar unnmöglich (vgl. Mehlhorn, 1984, S. 37). Diese „certain number“ die im obigen Zitat erwähnt wird ist im gegebenen Kontext demzufolge $T(n)$, die Information die es zu finden gilt ist die *Größenordnung des Wachstums* (auch *Wachstumsrate*) der Funktion (vgl. Shaffer, 2011, S. 63).

Kann die Effizienz eines Algorithmus als

$$T(n) = an^2 + bn + c \tag{1.1}$$

ausgedrückt werden (wobei a, b und c beliebige von n unabhängige Konstanten sind) so ist nur der Term n^2 von Interesse, nachdem Terme niederer Ordnung (bn und c) und konstante Faktoren (a) bei großen n relativ bedeutungslos werden (vgl. Cormen u. a., 2001, S. 28). (1.1) kann nun mithilfe der O-Notation zu

$$T(n) = O(n^2)$$

umformuliert werden. Das ist der Kern der asymptotischen Analyse in diesem Kontext: Die Untersuchung eines Algorithmus für große n und die damit einhergehende Möglichkeit der Vereinfachung von Algorithmen (vgl. Shaffer, 2011, S. 63). Genauer beschreibt $O(g(n))$ für eine gegebene Funktion $g(n)$ (im obigen Fall $g(x) = n^2$) die Menge von Funktionen

$$O(g(n)) = \{f(n) \ : \text{es existieren positive Konstanten } c \text{ und } n_0 \text{ derart,} \\ \text{dass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$

(vgl. Mehlhorn, 1984, S. 37). Die O-Notation definiert also eine „asymptotische obere Schranke“ (vgl. Shaffer, 2011, S. 64): Der Wert der Funktion $f(n)$ ist kleiner oder gleich dem Wert der Funktion $cg(n)$ für alle $n \geq n_0$.

Pseudocodenotation *Zu erweitern.*

Die Pseudocodenotation ist (wie alle Pseudocodedarstellungen der folgenden Algorithmen) aus Cormen u. a., 2001 übernommen.

Die Namen von Algorithmen sind in KAPITÄLCHEN und Variablen in *Kursivschrift* gesetzt.

Arrays (üblicherweise A) haben immer eine Eigenschaft *length* welche die Größe des Arrays darstellt und mit $A.\mathit{length}$ abgerufen wird. Im Fließtext werden $A.\mathit{length}$ und n oft synonym verwendet.

1.1 Insertion Sort

Der Insertion Sort baut auf der Annahme auf, dass vor der Begutachtung eines Elements $A[j]$ die Elemente $A[1\,..\,j-1]$ bereits sortiert wurden, $A[j]$ wird anschließend in das bereits sortierte „Subarray“ einsortiert (vgl. Knuth, 1998, S. 80). Siehe Abbildung 1.1 für eine Illustration der Vorgehensweise anhand eines Beispielarrays.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

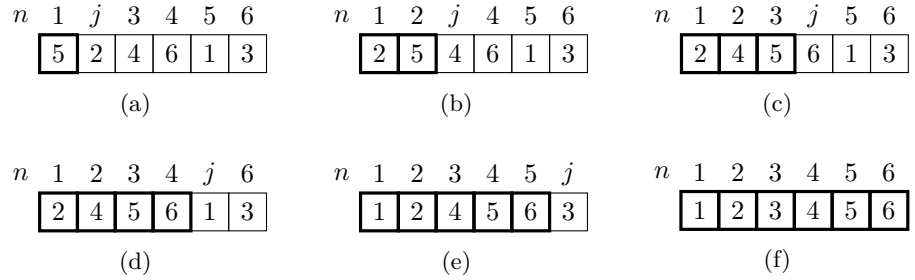


Abbildung 1.1: Illustration des Arrays $A = \{5, 2, 4, 6, 1, 3\}$ während es von INSERTION-SORT(A) bearbeitet wird. Über einem Rechteck steht sein Index in A, in den Rechtecken steht der jeweilige Wert von A an diesem Index. Fett gedruckte Rechtecke sind Teil des bereits sortierten Subarrays $A[1\,..\,j-1]$. (a) bis (e) zeigen die fünf Iterationen der **for**-Schleife auf Zeilen 1–8 beginnend mit $j = 2$ bis $j = A.\mathit{length}$ beziehungsweise $j = 6$. Das Element $A[j]$ ist mit einem Obenstehendem j gekennzeichnet, in der jeweils nächsten Iteration wurde es dann bereits an seine korrekte Position im sortierten Subarray $A[1\,..\,j-1]$ bewegt. (f) zeigt das sortierte Array. Abbildungen (a)–(e) sind maßgeblich inspiriert von Cormen u. a., 2001, S. 18, Figure 2.2.

INSERTION-SORT ist aus Cormen u. a., 2001, S. 18 entnommen, eine Implementation ist in Listing A.2 zu finden.

Im günstigsten Fall ist A bereits sortiert, für alle $j = 2, 3, \ldots A.length$ gilt nun $A[j-1] \leq key$: der Körper der **while** Schleife auf Zeile 5 wird also nie ausgeführt. Der Algorithmus ist in diesem Fall in $O(n)$ (vgl. ebd., S. 28).

Der ungünstigste Fall ist eine umgekehrt sortierte Liste. Jedes Element $A[j]$ muss mit jedem Element des sortierten Subarrays $A[1 \ldots i]$ verglichen werden, der Algorithmus ist im ungünstigsten Fall also in $O(n^2)$ (vgl. ebd., 28f).

1.2 Quicksort

Quicksort ist ein *divide-and-conquer* Algorithmus, der rekursiv ein Array in zwei Subarrays teilt und diese sortiert (vgl. Knuth, 1998, 113f). Ein *divide-and-conquer* Algorithmus, auch „Teile-und-Herrsche-Algorithmus", besteht nach Cormen u. a., 2001, S. 65 immer aus drei Schritten:

Divide *Teile die Aufgabe in mehrere Subaufgaben.* PARTITION teilt ein Array $A[p\,..\,r]$ in zwei Subarrays $A[p\,..\,q-1]$ und $A[q+1\,..\,r]$ derart, dass alle Elemente von $A[p\,..\,q-1]$ kleiner oder gleich $A[q]$ sind, was wiederum kleiner oder gleich allen Elementen von $A[q+1\,..\,r]$ ist. Der index q ist als Teil der Prozedur zu ermitteln.

Conquer *„Erobere" die Subaufgaben durch rekursive Auflösung.* In QUICKSORT werden die zwei Subarrays $A[p\,..\,q-1]$ und $A[q+1\,..\,r]$ durch rekursive Aufrufe an QUICKSORT (und damit PARTITION) sortiert.

Combine *Füge die Lösungen der Subaufgaben zur Lösung des Originalproblems zusammen.* Nachdem die Subarrays bereits sortiert sind müssen sie nicht kombiniert werden, das ganze Array $A[p\,..\,q]$ ist sortiert.

Die obige Prozessbeschreibung von PARTITION und QUICKSORT ist entnommen aus ebd., S. 170.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

PARTITION wählt immer ein Element $x = A[r]$ als „Drehpunkt", um den das Subarray $A[p\,..\,r]$ geteilt wird. Im Laufe der Prozedur wird das Subarray in vier Regionen geteilt die gewisse Eigenschaften erfüllten, zu sehen in Abbildung 1.2.

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

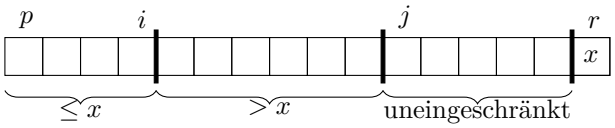


Abbildung 1.2: Die vier Regionen die von PARTITION auf einem Subarray $A[p\,..\,r]$ behandelt werden. Die Werte in $A[p\,..\,i]$ sind alle $\leq x$, die Werte in $A[i+1\,..\,j-1]$ sind alle $> x$, und $A[r] = x$. Das Subarray $A[j\,..\,r-1]$ kann jegliche Werte beinhalten. Die Abbildung und Beschreibung wurden übernommen aus Cormen u. a., 2001, S. 173, Abbildung 7.2.

QUICKSORT und PARTITION sind aus Cormen u. a., 2001, S. 171 entnommen. Eine Implementation des ersteren ist in Listing A.3 zu finden, letzterer Algorithmus ist äquivalent zu std :: partition (vgl. ISO/IEC 14882, 2017, S. 927).

Die Laufzeit von Quicksort hängt von der Aufteilung der Eingabemenge, welche in PARTITION geschieht, ab. Im ungünstigsten Fall produziert PARTITION ein Subarray mit $n-1$ Elementen, und eines mit 0 Elementen. Die Rekursionsgleichung für die Laufzeit ist in diesem Fall

$$\begin{aligned} T(n) &= T(n-1) + T(0) + O(n) \\ &= T(n-1) + O(n) \end{aligned} \tag{1.2}$$

nachdem PARTITION $O(n)$ ist und ein Aufruf mit einer Menge der Größe 0 eine Laufzeit von $O(1)$ hat (vgl. Cormen u. a., 2001, S. 175). Daraus folgt eine Effizienz von $O(n^2)$ im schlechtesten Fall (vgl. ebd., S. 1146). Diese Laufzeit tritt auch auf, wenn die Eingabemenge bereits sortiert ist (ebd., S. 175).

Im günstigsten Fall produziert PARTITION ein Subarray der Größe $\lfloor n/2 \rfloor$ und eines der Größe $\lceil n/2 \rceil - 1$. In diesem Fall ist die (vereinfachte) Rekursionsgleichung

$$T(n) = 2T(n/2) + O(n),$$

mit der Lösung $T(n) = O(n \lg n)$ nach ebd., S. 94.

1.3 Heapsort

Unter dem Begriff „Heap“ wird im Folgenden eine in einem Array abgebildete Datenstruktur verstanden, welche als nahezu vollständiger Binärbaum betrachtet werden kann (vgl. Aho u. a., 1974, 87f, siehe Abbildung 1.3). Stellt ein Array A einen Heap dar, so hat es neben dem bekannten Attribut $A.\mathit{length}$ ein Attribut $A.\mathit{heap\text{-}size}$, was der Anzahl der Arrayelemente entspricht, die einen Knoten darstellen. Nur die Elemente in $A[1\,..\,A.\mathit{heapsize}]$ sind gültige Elemente des Heaps, $A[A.\mathit{heap\text{-}size}+1\,..\,A.\mathit{length}]$ kann beliebige Elemente enthalten. Diese Notation und Eigenschaften sind äquivalent zu jenen in Cormen u. a., 2001.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

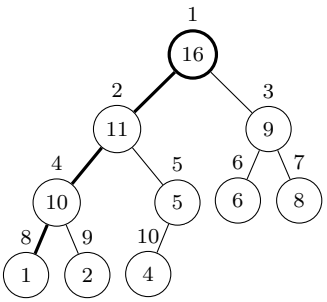
LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

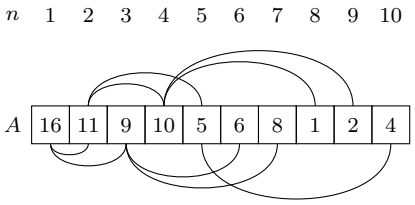
PARENT, LEFT und RIGHT sind aus Cormen u. a., 2001, S. 152 entnommen.

Es gibt zwei Arten von binären Heaps: Max-Heaps und Min-Heaps. Beide Arten erfüllen eine „Heap-Eigenschaft“ die von der Art des Heaps abhängig ist. In einem Max-Heap ist dies die *Max-Heap-Eigenschaft* die aussagt, dass für jeden Knoten i der nicht der Wurzelknoten ist $A[\text{PARENT}(i)] \leq A[i]$ gelten muss (vgl. Horowitz u. a., 1997, S. 92). Das heißt der Wert eines Knotens ist höchstens der seines Elternknotens das größte Element eines Max-Heaps ist der Wurzelknoten. Min-Heaps werden im Folgenden nicht verwendet und deshalb nicht näher behandelt.

Die *Höhe* eines Knotens in einem Heap ist definiert durch die Anzahl der Kanten entlang des längsten, einfachen Weges zu einem Blattknoten (vgl. Cormen u. a., 2001, S. 153). Die Höhe eines Heaps ist die Höhe des Wurzelknotens (siehe Abbildung 1.3(b)).



(a) Der Wurzelknoten und die Kanten entlang eines der längsten, einfachen Wege zu einem Blattknoten sind fett gedruckt. Die Anzahl der hervorgehobenen Kanten entspricht der Höhe $h = 3$ des Baums.



(b) Die Linien über und unter dem Array stellen die Eltern-Kind-Beziehungen zwischen den Knoten dar, Eltern sind immer links von ihren Kindern.

Abbildung 1.3: Ein binärer Max-Heap, dargestellt (a) als Binärbaum und (b) als Array (wobei $A.\textit{heap-size}$ gleich $A.\textit{length}$ ist). Die Zahlen in den Kreisen und Rechtecken ist der Wert dieses Knotens beziehungsweise dieses Elements, die Zahlen darüber sind der Index des jeweiligen Knotens/Elements im Array. Angeleht an Cormen u. a., 2001, S. 152 und Aho u. a., 1974, S. 88.

Aufrechterhaltung eines Heaps MAX-HEAPIFY erhält ein Array A und einen index i im Array. Es nimmt an, dass

- die Binärbäume links und rechts von $A[i]$ bereits Heaps sind aber dass,
- $A[i]$ kleiner als seine Kinder sein könnte, und damit die Heap-Eigenschaft verletzen würde.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Im Laufe der rekursiven Aufrufe an MAX-HEAPIFY „gleitet“ das Element $A[i]$ den Heap hinunter bis die Heap-Eigenschaft gegeben ist: Zeilen 1 bis 7 ermitteln das größte Element $A[largest]$ aus $\{\text{LEFT}(i), \text{RIGHT}(i), A[i]\}$. Ist $\text{LEFT}(i)$ oder $\text{RIGHT}(i)$ nicht Teil des Heaps — gilt also $\text{LEFT}(i)$ oder $\text{RIGHT}(i) \leq A.heap\text{-}size$ — so ist $A[i]$ garantiert das größte Element. Ist $A[largest]$ gleich $A[i]$ dann ist der Baum ab $A[i]$ ein Max-Heap und MAX-HEAPIFY terminiert. Andernfalls werden das größte Element und $A[i]$ vertauscht wodurch die Heap-Eigenschaft wieder gilt (Zeile 9). Der Baum ab $A[largest]$ könnte nun jedoch gegen die Heap-Eigenschaft verstoßen (nachdem $A[largest]$ und $A[i]$ vertauscht wurden), also wird MAX-HEAPIFY rekursiv auf diesem „Unterbaum“ aufgerufen (Zeile 10).

Im ungünstigsten Fall muss der ganze Teilbaum welcher den Knoten i als Wurzel hat durchlaufen werden. Ist n die Größe dieses Teilbaums kann die Höhe des Baums mit $h = \lg n$ ermittelt werden, somit ist die Effizienz im ungünstigsten Fall in $O(\lg n)$ (vgl. Cormen u. a., 2001, S. 155). Der Einfachheit halber wird diese Effizienz auch für den günstigsten Fall übernommen, eine asymptotisch engere (aber deswegen nicht richtigere) Effizienz wird in Bollobás u. a., 1996 angeführt.

Bauen eines Heaps Stellt das Array A einen Heap dar so sind die Elemente $A[\lfloor n/2 \rfloor + 1 \,..\, n]$ Blattknoten des Baumes[3]. BUILD-MAX-HEAP führt für die Knoten $A[1 \,..\, \lfloor n/2 \rfloor]$ MAX-HEAPIFY aus und baut so einen vollständigen Heap aus einem beliebigen Array.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

Die Effizienz von BUILD-MAX-HEAP ist immer in $O(n)$, Knuth, 1998, S. 155 zeigt die Nontrivialität dieser Aussage.

[3] Die Kinder eines Knotens an der Stelle i sind an den Stellen $2i$ und $2i + 1$ (siehe LEFT und RIGHT). Alle Knoten die keine Blattknoten sind (d. h., die Eltern sind) müssen sich demzufolge an den Stellen $A[1 \,..\, \lfloor n/2 \rfloor]$ befinden, da ihre Kinder sonst außerhalb des Heaps wären: Ist $i > \lfloor n/2 \rfloor$ dann wäre $2i > n$ also muss der Knoten an der Stelle i ein Blattknoten sein. (Q. e. d., damit ist Aufgabe 6.1-7 aus Cormen u. a., 2001, S. 154 gelöst.)

Heapsort Die Eingangsmenge $A[1\,..\,n]$ wird mithilfe von Build-Max-Heap in einen Heap verwandelt. $A[1]$ beinhaltet nun das größte Element der Liste, durch austauschen von $A[1]$ und $A[n]$ befindet sich das Element bereits an seiner finalen Position. Das bereits einsortierte Element kann nun durch Verkleinerung des Heaps um 1 von diesem entfernt werden. Das neue Wurzelement könnte jetzt jedoch die Heap-Eigenschaft verletzen, was durch einen Aufruf von Max-Heapify am Wurzelelement unmöglich gemacht wird.

```
Heapsort(A)
1  Build-Max-Heap(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      Max-Heapify(A, 1)
```

Max-Heapify, Build-Max-Heap und Heapsort sind aus Cormen u. a., 2001, S. 154, 157 entnommen. Die Standardbibliothek von C++ deckt die obigen Algorithmen sehr gut ab: Build-Max-Heap ist äquivalent zu std :: make_heap (vgl. ISO/IEC 14882, 2017, S. 933), Heapsort (ohne dem Aufruf von Build-Max-Heap) ist äquivalent zu std :: sort_heap (vgl. ebd., S. 933.). Listing A.4 ist eine „Implementation“ der obigen Algorithmen — sie ruft lediglich die eben genannten Funktionen der Standardbibliothek in Folge auf.

Die Effizienz von Heapsort ist in jedem Fall in $O(n \lg n)$ nachdem jeder der $n-1$ Aufrufe an Max-Heapify in $O(\lg n)$ ist (vgl. Cormen u. a., 2001, S. 160).

1.4 Merge Sort

Merge Sort ist, wie der Quicksort ein divide-and-conquer Algorithmus: ein Problem wird in kleinere Subprobleme aufgeteilt, die Lösungen der Subprobleme werden (oft rekursiv) ermittelt und zusammengesetzt.

Die entscheidende Handlung ist beim Merge sort das Kombinieren (nicht wie beispielsweise beim Quicksort das Aufteilen) welches in Merge durchgeführt wird.

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5       L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7       R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13       if L[i] ≤ R[j]
14            A[k] = L[i]
15       else A[k] = R[j]
16            j = j + 1
```

MERGE mag auf den ersten Blick komplex wirken, es kann jedoch in einfache Phasen heruntergebrochen werden:

0. p, q und r sind Indices des Eingabearrays A, es gilt $p \leq q < r$. Es wird angenommen, dass die Subarrays $A[r\,..\,q]$ und $A[q+1\,..\,r]$ sortiert sind.

1. Erstelle ein Array L mit den Werten $A[r\,..\,q]$ und ein Array R mit den Werten $A[q+1\,..\,r]$. Hänge ∞ an beide Arrays an. *(Zeilen 1 bis 9)*

2. Assoziiere mit dem Array L einen Index i und mit dem Array R einen Index j. Beide beginnen bei 1, jegliche Zugriffe auf L und R geschiehen von nun an nur über diese Indices. Durch Erhöhen eines der Indices können nun Elemente de facto aus dem Array entfernt werden. *(Zeilen 8 und 9)*

3. Iteriere über alle Elemente des Arrays A:

   Ist $L[i] \leq R[j]$ so wird das aktuelle Element aus A mit $L[i]$ überschrieben. $L[i]$ ist nun einsortiert, i wird um 1 erhöht. Andernfalls wird das aktuelle Element aus A mit $R[j]$ überschrieben und j um eins erhöht. *(Zeilen 12 bis 16)*

   Wird ein Array „erschöpft“, wurden also alle seine ursprünglich aus A stammenden Elemente einsortiert, so zeigt der jeweilige Index auf ∞. Dieses Element kann niemals einsortiert werden nachdem kein Element von L oder R jemals $< \infty$ sein kann, zeigt ein Index eines Subarrays darauf wird dieses also de facto ignoriert. Durch dieses „Scheinelement“ muss nicht bei jeder Iteration überprüft werden, ob eines der Arrays leer ist.

Die Effizienz der MERGE Prozedur ist immer in $O(n)$, nachdem die Anweisungen auf Zeilen 1–3 und 8–11 konstante Zeit benötigen und die **for**-Schleifen auf Zeilen 4–7 und 12–16 alle anschaulicherweise in $O(n)$ sind (vgl. Cormen u. a., 2001, S. 34).

Um nun ein Array A zu sortieren wird MERGE-SORT$(A, 1, A.\mathit{length})$ aufgerufen. MERGE-SORT halbiert das Eingangsarray durch rekursive Aufrufe bis nur mehr Subarrays der Größe 1 übrig bleiben. Diese Subarrays werden anschließlich

durch den auf die rekursiven Aufrufe folgenden Aufruf von MERGE in sukzessiv größere Subarrays zusammengefügt bis ein sortiertes Array als Resultat des letzten Aufrufs hervorgeht.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

MERGE und MERGE-SORT sind aus Cormen u. a., 2001, 33f entnommen. MERGE ist äquivalent zu std :: inplace_merge (vgl. ISO/IEC 14882, 2017, S. 929), eine Implementation von MERGE-SORT ist in Listing A.5 zu finden.

Unter der Annahme, dass n eine Zweierpotenz ist liefert jeder Teilungsschritt (Zeilen 2–4) zwei Subarrays der Größe $n/2$. Jeder Rekursionsschritt (Zeilen 3–4) trägt also $2T(n/2)$ zur Effizienz bei (Cormen u. a., 2001, S. 36).

Wie auch schon bei QUICKSORT muss die Effizienz des MERGE-SORT durch eine Rekursionsgleichung

$$T(n) = \begin{cases} O(1), & \text{wenn } n = 1, \\ 2T(n/2) + O(n), & \text{wenn } n > 1. \end{cases} \tag{1.3}$$

beschrieben werden (vgl. ebd., S. 36). Der Summand $O(n)$ ist hier die Effizienz von MERGE. (1.3) deckt sich mit (1.2) sowohl im günstigsten als auch im den ungünstigsten Fall, damit ist die Effizienz von QUICKSORT in $O(n \lg n)$ (vgl. ebd., S. 36).

Kapitel 2

Herangehensweise

Um die praktische Effizienz eines Algorithmus zu ermitteln, gilt es die Laufzeit mit verschiedenen Eingabegrössen zu messen, um sie als Funktion beschreiben zu können. Das Fundament hierfür ist in Abschnitt 2.1 gegeben, eine konkrete Methode zur Ermittlung der praktischen Effizienz wird in Abschnitt 2.3 dargelegt.

Die Beschaffenheit der Eingabemenge — im Falle von Sortieralgorithmen beispielsweise der Grad der Sortiertheit — hat oftmals großen Einfluss auf die Effizenz (siehe Kapitel 3 und ...[?]). In Abschnitt 2.2 werden einige Eingabemengen beschrieben.

2.1 Laufzeitermittlung

Ein Algorithmus wird ausgeführt, die Zeit unmittelbar vor (t_{vor}) und nach (t_{nach}) der Ausführung wird gemessen. Die Laufzeit des Algorithmus ist nun gleich $t_{nach} - t_{vor}$.

Eine konkrete Implementation einer Klasse zur Messung der Laufzeit eines Algorithmus ist in Listing 2.1 gegeben. *Referenzen zu empirischen Analysen die einen ähnlichen Mechanismus zur Laufzeitermittlung verwenden (kann nicht so schwer sein, was sollen sie sonst verwenden) sind beizufügen.*

```
1  class experiment {
2    std::function<void()> algorithm;
3
4  public:
5    using time_t = double;
6
7    explicit experiment(std::function<void()> algorithm)
8      : algorithm(std::move(algorithm)) { };
9
10   auto run() const {
11     const auto start = std::chrono::steady_clock::now();
12
13     algorithm();
14
15     const auto end = std::chrono::steady_clock::now();
```

```
16
17      return std::chrono::duration<time_t, std::micro>{ end − start };
18    }
19  };
```

Listing 2.1: Implementation einer Klasse zur Ermittlung der Laufzeit eines Algorithmus.

Die Laufzeit eines einfachen Algorithmus kann mit ihrer Hilfe durch

```
void a1() {  ...  }
const auto time = experiment(a1).run();
```

ermittelt werden, wobei die Zeitspanne in Mikrosekunden ($1\mu s = 10^{-6}s$) angegeben ist.

Algorithmen mit Eingabewerten Die im Folgenden behandelten Algorithmen haben üblicherweise einen oder mehrere Eingabewerte, sie erfüllen also nicht die Form wie sie vom Konstruktor in Listing 2.1, Zeile 7 erwartet wird.

Diese Einschränkun kann umgangen werden, indem eine „umhüllende Funktion“ der Form **void**() erstellt wird. Diese „Hülle“ ruft in Folge den tatsächlichen Algorithmus mit all seinen Eingabewerten auf.

```
void a2(size_t s) {  ...  }
const auto wrapper = std::bind(a2, 1337);
const auto time = experiment(wrapper).run();
```

2.2 Eingabengenerierung

Referenzen anderer Analysen mit gleichen oder überlappenden Eingabearten sind beizufügen.

Folgende Eingabemengen werden für die Ermittlung der praktischen Effizienz verwendet:

1. sortiert
2. invertiert
3. zufällig geordnet

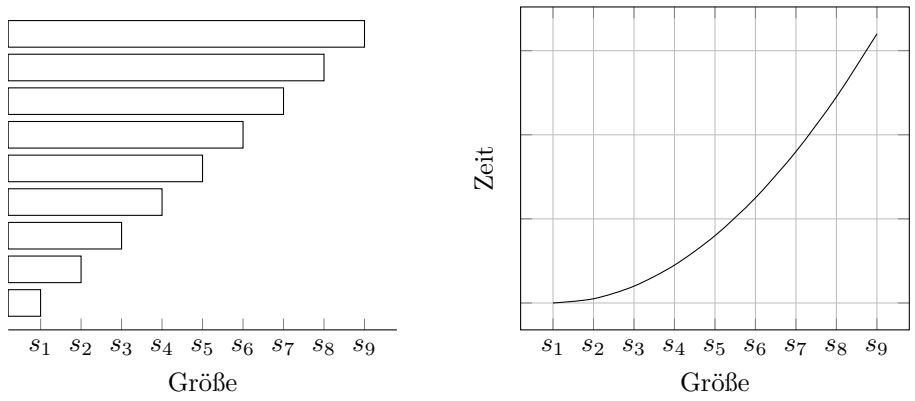
Siehe Listing A.1 für eine beispielhafte Implementation der obigen Eingabemengearten.

Eine ausführlichere Auswahl an Eingangsmengen welche an einen sie bearbeitenden Algorithmen angepasst ist — wie sie für eine eingehende Analyse eines einzelnen Algorithmus angebracht wäre — würde das allgemeiner gesetzte Ziel dieser Arbeit verfehlen.

2.3 Funktionsermittlung

Im gegebenen Kontext ist es das Ziel der „Funktionsermittlung“ eine Funktion in Abhängigkeit der Eingabegröße aufzustellen, welche die Laufzeit darstellt.

Die x-Werte dieser Funktion stellen also die Größe der Eingabemenge, und die y-Werte die Zeit die der Algorithmus bei Eingabe einer Menge dieser Größe benötigt hat, dar (siehe Abbildung 2.1).



(a) Größe der Eingangsmengen, dargestellt als Balkendiagramm.

(b) $f(s)$

Abbildung 2.1: Beispielhafter Funktionsgraph der praktischen Effizienz eines hypothetischen Algorithmus mit Illustration der Eingabemengengrößen.

Diese Funktion der praktischen Effizienz kann mithile der Funktion benchmark::run in Listing 2.2 ermittelt werden.

```
1  namespace benchmark {
2    using timings_t = std::map<size_t, experiment::time_t>;
3
4    template<class A, class S>
5    timings_t run(A algorithm, S set, int total_chunks) {
6      timings_t timings;
7
8      const size_t set_size = set.size();
9
10     std::fesetround(FE_TONEAREST);
11
12     const size_t chunk_size = set_size > total_chunks ?
13       std::nearbyint(set_size / total_chunks) : 1;
14
15     for (size_t i = chunk_size; i <= set_size; i += chunk_size) {
16       auto subset = S(set.begin(), set.begin() + i);
17       const auto time = experiment(std::bind(algorithm,
18         subset.begin(), subset.end(), std::less<>())
19       ).run();
20
21       timings.emplace(i, time.count());
22     }
23
24     return timings;
25   }
```

```
26
27      ...
28  }
```

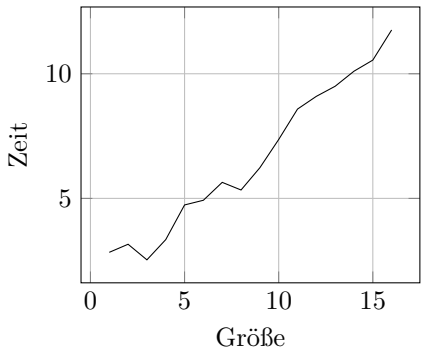
Listing 2.2: Implementation einer Funktion zur Ermittlung der praktischen Effizienz eines Algorotihmus mit einer bestimmenten Eingabemenge.

Der Parameter total_chunks definiert die Anzahl der Untermengen (am Beispiel von Abbildung 2.1 also 9). In Kombination mit der Größe der Eingabemenge set_size kann damit die konstante *Schrittgrösse* chunk_size ($s_1 - s_2$ beziehungsweise $s_n - s_{n+1}$ für $0 \leq n \leq 9$) ermittelt werden.

Die Funktion liefert Wertepaare als Inhalt eines Kontainers vom Typ std :: map<size_t, experiment::time_t>. Dieser assoziative Kontainer enthält eine geordnete Menge von Schlüssel-Wert-Paaren wobei die Schlüssel die Größe der jeweils betrachteten Untermenge, und die Werte die benötigten Zeiten darstellen. Die Werte können nun wie in Abbildung 2.2(a) ausgegeben, oder wie in Abbildung 2.2(b) als Graph dargestellt werden.

Größe	Zeit		
1	2.836	9	6.23
2	3.16	10	7.363
3	2.535	11	8.583
4	3.346	12	9.094
5	4.737	13	9.499
6	4.925	14	10.103
7	5.643	15	10.548
9	5.337	16	11.757

(a) *Quick sort* auf sehr kleinen, sortierten Eingabemengen; links ist die Größe der Eingabemenge, rechts die Zeit in Mikrosekunden.



(b) Graph der Daten in (a).

Abbildung 2.2: Demonstration des Ausgabeformats aus Listing 2.2 mit daraus generiertem Graphen.

Kapitel 3

Ergebnisse

Viele Graphen, unterteilung nach Art der Eingabe?

Konklusion

42.

Anhang A

Implementationen

Mögl. alternativer Titel: Algorithmusimplementationen. Listings sollten floating sein und labels und descriptions haben.

```
1  namespace sets {
2    using set_t = std::vector<int>;
3    using iterator_t = set_t::iterator;
4
5    set_t sorted(const size_t size) {
6      auto set = set_t(size);
7
8      std::iota(set.begin(), set.end(), 1);
9
10     return set;
11   }
12
13   set_t inverted(const size_t size) {
14     auto set = set_t(size);
15
16     std::iota(std::rbegin(set), std::rend(set), 1);
17
18     return set;
19   }
20
21   set_t random(const size_t size) {
22     auto set = sorted(size);
23
24     utils::random_shuffle(set.begin(), set.end());
25
26     return set;
27   }
28
29   ...
30 }
```

Listing A.1: Implementation von „Generatoren“ für diverse Arten von Eingabemengen.

```
1  template <class I, class P = std::less<>>
2  void insertion(I  first ,  I  last ,  P cmp = P{}) {
3    for (auto it =  first ;  it  != last ;  ++it) {
4      auto const insertion = std::upper_bound(first, it,  *it ,  cmp);
5      std :: rotate( insertion ,  it ,  std :: next(it ));
6    }
7  }
```

Listing A.2: Implementation des *insertion sort*.

```
1  template <class I, class P = std::less<>>
2  void quick(I  first ,  I  last ,  P cmp = P{}) {
3    auto const N = std::distance(first ,  last );
4    if  (N <= 1)
5      return;
6
7    auto const pivot = *std::next(first ,  N / 2);
8
9    auto const middle1 = std::partition(first ,  last ,  [=](auto const &elem) {
10     return cmp(elem, pivot); });
11   auto const middle2 = std::partition(middle1, last,  [=](auto const &elem)
         {
12     return !cmp(pivot, elem); });
13
14   quick( first ,  middle1, cmp);
15   quick(middle2,  last ,  cmp);
16 }
```

Listing A.3: Implementation des *quicksort*.

```
1  template<class RI, class P = std::less<>>
2  void heap(RI first ,  RI last ,  P cmp = P{}) {
3      std :: make_heap(first, last ,  cmp);
4      std :: sort_heap(first ,  last ,  cmp);
5  }
```

Listing A.4: Implementation des *heapsort*.

```
1  template <class BI, class P = std::less<>>
2  void merge(BI first, BI last, P cmp = P{}) {
3    auto const N = std::distance(first, last);
4    if (N <= 1)
5      return;
6
7    auto const middle = std::next(first, N / 2);
8
9    merge(first, middle, cmp);
10   merge(middle, last, cmp);
11
12   std::inplace_merge(first, middle, last, cmp);
13 }
```

Listing A.5: Implementation des *merge sort*.

Literatur

Aho, A.V., J.E. Hopcraft und J.D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000296.

Bocchino, Robert L., Vikram S. Adve, Sarita V. Adve und Marc Snir (2009). „Parallel Programming Must Be Deterministic by Default“. In: *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*. HotPar'09. Berkeley, California: USENIX Association, S. 4.

Bollobás, B., T.I. Fenner und A.M. Frieze (März 1996). „On the Best Case of Heapsort“. In: *J. Algorithms* 20.2, S. 205–217. ISSN: 0196-6774. DOI: 10.1006/jagm.1996.0011.

Bruijn, N.G. de (1958). *Asymptotic Methods in Analysis*. Bibliotheca Mathematica: A Series of Monographs on Pure and Applied Mathematics. Amsterdam, NL: North-Holland Publishing Company.

Bunge, Mario (1963). „A General Black Box Theory“. In: *Philosophy of Science* 30.4, S. 346–358. ISSN: 00318248, 1539767X. URL: http://www.jstor.org/stable/186066.

Cormen, Thomas H., Charles F. Leiserson, Ronald L. Rivest und Clifford Stein (2001). *Introduction to Algorithms*. 2nd. Cambridge, Massachusetts: The MIT Press.

Horowitz, Ellis, Sartaj Sahni und Sanguthevar Rajasekaran (1997). *Computer Algorithms*. New York: Computer Science Press.

ISO/IEC 14882 (Dez. 2017). *Programming Language C++*. Standard. Geneva, Switzerland: International Organization for Standardization.

Knuth, Donald E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0201896850.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Monographs in Theoretical Computer Science. An EATCS Series. Berlin, DE: Springer Berlin Heidelberg. ISBN: 9780387133027.

Shaffer, C.A. (2011). *Data Structures & Algorithm Analysis in Java*. Dover Books on Computer Science Series. New York: Dover Publications. ISBN: 9780486485812.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings