EFFIZIENZ VON SORTIERALGORITHMEN

Eine vorwissenschaftliche Arbeit verfasst von Laurenz Weixlbaumer
und betreut durch Mag.[a] Anna-Maria Klaghofer BEd.

8. Klasse
September 2021

Abendgymnasium Linz
BG/BRG für Berufstätige
Spittelwiese 14
4020 Linz

Inhaltsverzeichnis

Einleitung

Ein hochgestelltes „[?]“ weist auf fehlende Quellen hin. In der „Release Version“ ist es, wie dieser Absatz und die obenstehenden Hinweise („Vorschau“), unsichtbar. Sind ganze Sätze (oder Absätze) kursiv gesetzt so sind sie ebenfalls nur in dieser Vorschauversion sichtbar.

Die zwei primären Leitfragen dieser Arbeit sind:

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

Die Ermittlung der theoretischen Effizienz wird in Kapitel 2, und die Ermittlung der praktischen Effizienz in Kapitel 3 behandelt.

Das Verhältnis zwischen theoretischer und praktischer Effizienz wird in Kapitel 5 behandelt.

Als Basis der Erarbeitung dieser Fragestellungen dient Kapitel 1.

Notizen Die Leitfragen im Erwartungshorizont sind sinngemäß

- Was ist ein Sortieralgorithmus?
- Was ist die theoretische und praktische Effizienz eines Algorithmus und wie kann sie ermittelt werden?
- Wie verhält sich die „theoretische Effizienz“ von ausgewählten Sortieralgorithmen zu der (zu ermittelnden) „praktischen Effizienz“ jener Algorithmen?
- Welcher praktische Nutzen lässt sich daraus ziehen?

wobei die erste Frage implizit aus den folgenden herausgeht und die letzte Frage ebenfalls implizit als Teil der (verpflichtenden) Konklusion angenommen wird. Dieser Umstand sollte nahch möglichkeit konkret im Zuge der Darstellung der primären Fragen deutlich gemacht werden.

Es folgt ein Vergleich der *ungefähren* Gliederung im Erwartungshorizont mit der angestrebten, konkreten Gliederung.

0. Einleitung *(Titel und Position ident)*
1. Definition und Funktion von Algorithmen *(Titel und Position ident)*
2. Beschreibung von Sortieralgorithmen *(in Kapitel 4, „Behandelte Algorithmen und ihre theoretische Effizienz“)*
3. Laufzeitermittlung als Effizienzangabe eines Algorithmus *(in Kapitel 3, „Laufzeitermittlung als Effizienzangabe“)*
4. „Funktionsermittlung“ eines Algorithmus *(in Kapitel 2, „Asymptotische Analyse als Effizienzangabe“)*

5. Vergleich der praktischen und theoretischen Effizienz *(Titel und relative Position ident)*
6. Ergebnisse für die Praxis *(Titel wird als Konklusion interpretiert, Position ident)*

Die einzige größere Veränderung ist folglich, dass die „Beschreibung von Sortieralgorithmen“ erst nach der Definition der Effizienzen erfolgt und die konkrete theoretische Effizienz der Algorithmen mitbeinhaltet.

Kapitel 1

Definition und Funktion von Algorithmen

Dieses Kapitel beschäftigt sich mit der Definition des Algorithmus- und, in diesem Kontext, Effizienzbegriffs sowie mit der grundlegenden Funktionsweise von Algorithmen am Beispiel von Sortieralgorithmen.

In Abschnitt 1.1 werden der Ursprung und vorhergehende Bedeutungen des Wortes „Algorithmus“ behandelt. Ebenso wird eine, zur kontemporären Bedeutung des Begriffs passende, Begriffsdefinition aufgestellt.

Ein Überblick über die verschiedenen Möglichkeiten zur Darstellung von Algorithmen wird in Abschnitt 1.2 gegeben. Die in der folgenden Arbeit verwendete Art der Beschreibung wird in Abschnitt 1.3 im Detail beschrieben.

Die Begrifflichkeit „Sortieralgorithmus“ wird in Abschnitt 1.4 definiert. Anhand eines Beispielalgorithmus, dem Insertion-Sort, wird die Arbeitsweise von Algorithmen, und insbesondere von Sortieralgorithmen, dargestellt.

In Abschnitt 1.5 wird eine Definition des Effizienzbegriffs im Kontext von Algorithmen vorgenommen.

1.1 Definition

Noch um 1957 war „*algorithm*“ nicht in *Webster's New World Dictionary* vertreten. Dort war nur der ältere Begriff „*algorism*“ zu finden, zu Deutsch das Rechnen mit der arabischen Zahlschrift. *Algorism* entstammt dem Namen eines persischen Autors, *Abū 'Abd Allāh Muhammad ibn Mūsā al-Khwārizmī* [1](circa 825 A. D., vgl. Knuth, 1997, S. 1–2). Al-Khwārizmī schrieb das Buch *Kitab al jabr wa'l-muqābala* („[...] Rechenverfahren durch Ergänzen und Ausgleichen“), aus dessem Titel ein anderes Wort, „Algebra“, entstammt (Gericke, 1984, S. 197–199). Die Wandlung des Begriffs *algorism* zu *algorithm* (und damit Algorithmus) ist in *Vollständiges mathematisches Lexicon* (Wolff, 1747, S. 38) dokumentiert, hier wird der Begriff wie folgt definiert: „Unter dieser Benennung werden zusammen begriffen die 4 Rechnungs-Arten in der Rechen-Kunst nemlich addiren, multipliciren, subtrahiren und dividiren. [...]“ (alte Rechtschreibung übernommen).

Vor 1950 wurde der Begriff Algorithmus am häufigsten mit dem euklidischen Algorithmus assoziiert, einem Verfahren zur Ermittlung des größten gemeinsamen Teilers zweier Zahlen (vgl. Knuth,

[1] Al-Khwārizmīs Name stellt in der Literatur eine Quelle der Verwirrung dar: Horowitz, Sahni und Rajasekaran, 1997, S. 1, Knuth, 1997, S. 1, Pickover, 2009, S. 84 und Gericke, 1984, S. 197–199 verwenden bei ihren Erwähnungen jeweils unterschiedliche Namen, sprechen jedoch von derselben Person.

1997, S. 2).

Die moderne Bedeutung des Wortes *Algorithmus* ist nicht unähnlich zu den Bedeutungen der Wörter *Rezept*, *Vorgang*, *Verfahren*, et cetera. Dennoch konnotiert das Wort etwas anderes – es ist nicht nur eine endliche Menge von Regeln, die eine Folge von Operationen für die Lösung einer bestimmten Aufgabe darstellen, ein Algorithmus hat nach Horowitz, Sahni und Rajasekaran, 1997, S. 1 zwingend folgende fünf Merkmale:

1. *Eingabe.* Ein Algorithmus hat keine, eine oder mehrere Eingangsmengen, welche entweder zu Beginn oder während der Laufzeit gegeben werden.
2. *Ausgabe.* Ein Algorithmus hat eine oder mehrere Ausgabemengen, welche eine wohldefinierte Beziehung zu den Eingangsmengen haben.
3. *Eindeutigkeit.* Jeder Arbeitsschritt eines Algorithmus ist eindeutig definiert.
4. *Endlichkeit.* Ein Algorithmus terminiert[2] nach einer endlichen Anzahl von Arbeitsschritten.
5. *Effektivität.* Die Arbeitsschritte müssen einfach genug sein um in endlicher Zeit von einer Person mit Stift und Papier ausgeführt werden zu können. *Mit Beispielen besser zu erklären.*

Etwas informeller kann ein Algorithmus als ein *wohldefiniertes* (vgl. Aluffi, 2009, S. 16) Berechnungsverfahren (engl. *computational procedure*) beschrieben werden, das einen Wert, oder eine Menge von Werten, als Eingabe erhält und einen Wert, oder eine Menge von Werten, als Ausgabe produziert (vgl. Cormen u. a., 2001, S. 5). Ein Algorithmus ist also eine Folge von Berechnungsschritten, welche Eingabe zu Ausgabe überführen.

1.2 Methoden zur Algorithmusbeschreibung

Algorithmen können auf vielfältige Weise beschrieben und dargestellt werden.

Knuth verwendet etwa in seinem Standardwerk „*The Art of Computer Programming, Volume 1: Fundamental Algorithms*“ (Knuth, 1997) eine Mischung aus natürlicher Sprache und mathematischen Ausdrücken.

Algorithmus E (*Euklids Algorithmus*). Gegeben seien zwei positive ganze Zahlen m und n. Der *größte gemeinsame Teiler*, also die größte positive ganze Zahl welche sowohl m als auch n gerade teilt, ist zu ermitteln.

E1 [Ermittle den Rest.] Dividiere m durch n, r ist der Rest. (Es gilt nun also $0 \leq r < n$.)
E2 [Ist er Null?] Gilt $r = 0$ so terminiert der Algorithmus, n ist die Lösung.
E3 [Verringere.] Setze $m \leftarrow n$, $n \leftarrow r$ und gehe zurück zu Schritt E1.

Abbildung 1.1: Algorithmusbeschreibung nach Knuth. Unter Anderem verwendet in Knuth, 1997 und Knuth, 1998. Beispiel entnommen aus Knuth, 1997, S. 2, Algorithm E (übersetzt aus dem Englischen).

Diese Art der Algorithmusbeschreibung, wie sie beispielhaft in Abbildung 1.1 anhand von Euklids Algorithmus dargestellt wird, gliedert sich gut in den Lesefluss ein, und ist am besten geeignet um die Funktionsweise eines Algorithmus verständlich darzustellen (Zobel, 2015, S. 147).

Eine andere Methode ist die Darstellung eines Algorithmus mithilfe von *Flowcharts*, wie sie in Abbildung 1.2 wieder am Beispiel von Euklids Algorithmus dargestellt wird.

Diese Art der Beschreibung eignet sich am besten für kleine bzw. kurze Algorithmen (Horowitz, Sahni und Rajasekaran, 1997, S. 5). Sie wird (wie etwa auch von Knuth) meistens in Kombination mit einer anderen Algorithmusbeschreibung verwendet um diese zu komplementieren.

[2] Abbruch mit Erfolg oder Misserfolg

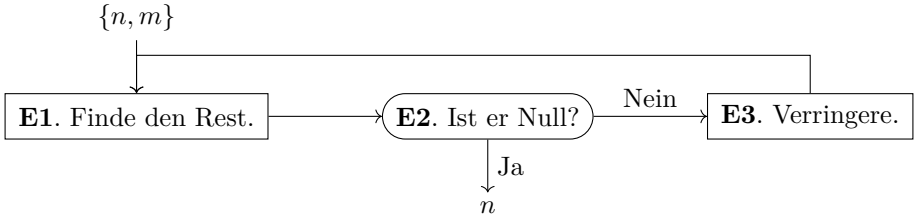


Abbildung 1.2: Algorithmusbeschreibung in Form eines Flowcharts. Beispiel in abgeänderter Form aus Knuth, 1997, S. 3, Fig. 1 entnommen.

```
EUKLID(n, m)
1  while (r = n  mod m) ≠ 0
2        m = n
3        n = r
4  return n
```

Abbildung 1.3: Beschreibung durch *Pseudocode* wie er auch in der folgenden Arbeit verwendet wird.

Die wohl am weitreichendsten etablierte Art der Beschreibung von Algorithmen ist jedoch *Pseudocode* (vgl. Zobel, 2015, S. 147), wie er etwa in Abbildung 1.3 zu sehen ist.

Pseudocode ist näher zu tatsächlichen Programmiersprachen in denen ein Algorithmus implementiert werden könnte als die bisher erwähnten Alternativen (vgl. Oda u. a., 2015, S. 1). Er bleibt allerdings abstrakt genug um Algorithmen (im Gegensatz zu vielen Programmiersprachen) unabhängig von Hardware und im Kontext unwichtigen Formalismen diverser Programmiersprachen darstellen zu können (vgl. Bemer, 1958, S. 1). Die Darstellung durch Pseudocode macht allerdings dennoch ein erstes Verständnis des beschriebenen Algorithmus üblicherweise schwerer als andere Methoden, bleibt allerdings durch die Nähe zu möglichen konkreten Implementationen unmissverständlicher (vgl. Zobel, 2015, S. 147). Aus letzterem Grund wird in der folgenden Arbeit Pseudocode zur Algorithmusbeschreibung verwendet.

1.3 Pseudocode

Es gibt nicht *eine bestimmte* Pseudocode-Notation.

Programmiersprachen wie beispielsweise C++17 sind versioniert und standardisiert (vgl. ISO/IEC 14882, 2017), für Pseudocode gibt es keinen derart weit verbreiteten (oder wohldefinierten) Standard (vgl. Knill, 1996). Eine derartige Standardisierung wäre allerdings auch entgegen der Zielsetzung: Die Freiheit, Pseudocode an den gegebenen Kontext anzupassen ist eine seiner großen Stärken. So wird zwar in einem Gutteil der in dieser Arbeit zitierten Werke Pseudocode verwendet (sofern im Kontext erfordert), konkrete Ausprägungen variieren jedoch in Syntax und Semantik[3].

Wie schon in Abschnitt 1.2 erwähnt wird für Beschreibungen von Algorithmen in dieser Arbeit überwiegend Pseudocode zum Einsatz kommen. Genauer werden die *„pseudocode conventions“* aus Cormen u. a., 2001, 20ff verwendet, folgend kurz zusammengefasst.

- Namen von Algorithmen sind in KAPITÄLCHEN gesetzt. Namen von Variablen sind *kursiv* gesetzt und üblicherweise einzelne Buchstaben (i, n, ...).

[3] Vgl. beispielsweise die diversen Definitionen der verwendeten Pseudocode-Varianten in Horowitz, Sahni und Rajasekaran, 1997, S. 5, Cormen u. a., 2001, 20ff und Aho, Ullman und Hopcroft, 1974, 33ff, die jeweils an den Verwendungskontext angepasst sind.

- Die „Blockstruktur“ ist durch Einrückung vorgegeben. Beispielsweise beginnt der Körper der **while** Schleife aus Abbildung 1.3 auf Zeile 2 und endet auf Zeile 3.
- Die Konstrukte **while** und **for** sind ähnlich interpretierbar wie jene in Sprachen wie C oder Java, wobei eine Vereinfachung der in dieser Sprachgruppe üblichen Syntax der **for** Schleife vorgenommen wurde.

```
1  i = 2
2  while i < 100
3        // Körper der Schleife
4        i = i^2
5
```

Während i kleiner als 100 ist, wird der Körper der Schleife ausgeführt. Nach Zeile 5 gilt $i = 256$

```
1  for j = 0 to 10
2        // Körper der Schleife
```

Hier ist j der „Schleifenzähler“ und 10 der Endwert. Der Zähler wird um 1 erhöht, bis er *gleich* dem Endwert ist.

- Ebenso ähnlich zu Programmiersprachen wie C, ... ist die Verwendung von = mit der Bedeutung einer *Zuweisung* und == mit der Beudeutung eines *Vergleichs* (wie $<$ oder $\leq$).
- Wie aus den vorhergehenden Beispielen hervorgeht deutet das Symbol „// “ darauf hin, dass der Rest der Zeile als Kommentar zu verstehen ist.
- Elemente eines *Array* (Datenfeld) A werden durch $A[i]$ abgerufen, wobei i der Index des Elements ist. Arrays werden im Gegensatz zu vielen üblichen Programmiersprachen beginnend mit 1 indiziert und befüllt – $A[1]$ ist das erste Element in einem Array A.

  Die Schreibweise $A[i\,..\,j]$ wird verwendet um einen „Ausschnitt“ bzw. ein „*Subarray*“ eines Arrays darzustellen. So steht $A[1\,..\,j]$ für das Subarray von A welches (für $j > 2$) die j Elemente $A[1]$, $A[2]$, ... $A[j]$ beinhaltet.

  Die Länge eines Arrays kann durch $A.\mathit{length}$ abgerufen werden. (In C müsste hierfür eine separate Variable geführt werden.)
- Manchmal werden Schritte auch nur in natürlicher Sprache beschrieben, wenn dadurch die Verständlichkeit gefördert wird, und dennoch die Eindeutigkeit der Schritte gegeben ist.

  So wird beispielsweise wenn angebracht „tausche $A[i]$ und $A[j]$ aus“ anstelle von

  ```
  1  t = A[i]
  2  A[i] = A[j]
  3  A[j] = t
  ```

  geschrieben.
- Eine **return** Anweisung überträgt die Kontrolle und einen optionalen, auf die Anweisung folgenden, Wert zurück zur Aufrufstelle.

  Hier wäre ein Beispiel angebracht, vielleicht mit Rekursion?

1.4 Sortieralgorithmen

Ein Sortieralgorithmus ist ein Algorithmus der eine Eingabemenge A in eine sortierte Permutation dieser Menge überführt, und als Ausgabemenge zurückgibt. Genauer muss bei einer Eingabemenge mit den n Elementen $a_1, a_2, \ldots, a_n$ gelten, dass für alle Elemente $a'_1, a'_2, \ldots, a'_n$ der Ausgabemenge die Ordnungsrelation $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt (vgl. Knuth, 1998, S. 4).

Ein ikonischer Sortieralgorithmus ist der „insertion sort“ (vgl. Knuth, 1998, S. 74). Jedes Element wird einzeln betrachtet und in eine (wachsende) sortierte „Teilliste“ *eingefügt* – daher der Name *insertion* sort. Dieser Algorithmus wird in Abschnitt 4.1 (siehe insbesondere Abbildung 4.1)

genauer behandelt, für ein grundlegendes Verständnis der Arbeitsweise von Algorithmen wird es jedoch hilfreich sein ihn auch jetzt schon etwas genauer zu betrachten.

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
```

Bevor ein Element $A[j]$ betrachtet wird besteht die Annahme, dass die Elemente $A[1], \ldots, A[j-1]$ sortiert sind. (Aus diesem Grund beginnt die **for** Schleife in Zeile 1 mit $j = 2$ und nicht etwa mit $j = 1$; das Subarray $A[1..j-1]$ hat für $j = 2$ nur ein Element und ist somit immer „sortiert“.)

Jedes Element $A[2..A.length]$ wird nun einzeln betrachtet und in $A[1..j-1]$ „einsortiert“. Um ein Element derart in die Teilliste einzufügen zu können muss

(a) der neue Index des einzufügenden Elements gefunden werden.
(b) Platz für das einzufügende Element geschaffen werden.

Dies passiert in der **while** Schleife (beginnend mit Zeile 5). Sie iteriert vorerst über alle möglichen Indexe i, absteigend und beginnend mit dem höchstmöglichen ($i = j - 1$, Zeile 4) bis zum kleinstmöglichen $i = 1$. Für jedes i wird das Element $A[i]$ „nach oben“, zu $A[i+1]$ verschoben (Zeile 6) – somit wird Platz für das einzufügende Element geschaffen.

Im Zuge dieser Verschiebung wird anfangs das Element $A[j]$ überschrieben, nachdem bei der ersten Iteration gilt, dass $j = i + 1$. Aus diesem Grund wird der Wert von $A[j]$ zu Beginn der **for** -Schleife in der Variable *key* zwischengespeichert (Zeile 2).

Gilt nach einer Verschiebung $A[i] \leq key$ (bzw. gilt also $A[i] > key$, Zeile 5, *nicht*), so wurde der neue Index des einzufügenden Elements gefunden: Das Element ist nach $A[i]$, also am Index $i + 1$ einzufügen (Zeile 8).

Nach Beendigung der **for** Schleife wurde die Eingabemenge vollständig sortiert.

1.5 Effizienz

Die Effizienz eines Algorithmus ist bestimmt durch seinen Verbrauch von Ressourcen (vgl. Sedgewick und Flajolet, 2013). In der folgenden Arbeit wird zwischen der „praktischen Effizienz“ und der „theoretischen Effizienz“ eines Algorithmus unterschieden. Erstere ist empirisch zu ermitteln (siehe Kapitel 3), letztere wird durch mathematische Analyse bestimmt (siehe Kapitel 2).

Diese Unterscheidung geschieht in der Literatur üblicherweise implizit. So behandeln beispielsweise etwa McGeoch, 2012, Luo u. a., 2012, Sedgewick und Flajolet, 2013 und Horowitz, Sahni und Rajasekaran, 1997 allesamt in irgendeiner Form die „Effizienz von Algorithmen“. McGeoch, 2012 behandelt sowohl „theoretische“ als auch „praktische“ Effizienz unter dem selben Namen, abhängig vom Kontext. Luo u. a., 2012 behandelt ausschließlich praktische Effizienz, nennt diese allerdings nur Effizienz. Als Gegenbeispiel behandeln Sedgewick und Flajolet, 2013 und Horowitz, Sahni und Rajasekaran, 1997 nur theoretische Effizienz, nennen diese aber auch nur Effizienz.

Wie der Großteil vergleichbarer Analysen (vgl. bspw. Cormen u. a., 2001, 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.

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[4]) als Funktion T in Abhängigkeit der „Eingabegröße“ n dargestellt, also als $T(n)$.

Ist die Effizienz eines Algorithmus zu ermitteln, gilt es also die Zeit die dieser benötigt zumindest annähernd als Funktion in Abhängigkeit von n darzustellen.

[4] 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.

Kapitel 2

Asymptotische Analyse als Effizienzangabe

Dieses Kapitel beschäftigt sich mit der Frage, wie die theoretische Effizienz (lt. der Definition in Abschnitt 1.5) eines Algorithmus ermittelt werden kann.

Dafür werden in Abschnitt 2.1 und Abschnitt 2.2 die grundlegende Einheit der theoretischen Effizienz und einige untergeordnete Unterteilungen erläutert. Ein Wekzeug für die vereinfachende Darstellung der Effizienz wird in Abschnitt 2.3 gegeben.

In Abschnitt 2.4 werden einige oft vorkommende Wachstumsraten der Effizienz als Funktion $T(n)$ dargestellt.

Schlussendlich werden in Abschnitt 2.5 einige Pseudocode-Ausschnitte beispielhaft hinsichtlich ihrer Effizienz analysiert, um den Ermittlungsprozess der theoretischen Effizienz darzustellen.

2.1 Einfache Operationen

Die Werte von $T(n)$, also die theoretische Effizienz eines Algorithmus auf einer Eingabemenge der Größe n, sind im Kontext der theoretischen Analyse die Anzahl der „einfachen Operationen“ oder „Schritten“ welche für diese Eingabemenge ausgeführt werden müssen (vgl. Cormen u. a., 2001, S. 25 und Horowitz, Sahni und Rajasekaran, 1997, 18f). „Einfache Operationen“ sind hier jene Operationen deren benötigte Zeit unabhängig von den Operanden ist, beispielsweise arithmetische Grundrechenarten 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. Shaffer, 2011, S. 55).

2.2 Szenarien

Es wird zwischen der theoretischen Effizienz im „günstigsten“, „ungünstigsten“, und „durchschnittlichen Szenario bzw. *Fall*“ unterschieden, alle beziehen sich auf die Beschaffenheit der Eingangsmenge (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 28).

Zur Veranschaulichung dieser Unterscheidung ist ein einfacher Suchalgorithmus, Sequential-Search (vgl. Knuth, 1998, S. 396), zu betrachten 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
```

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).

2.3 Asymptotische Analyse

Zur einleitenden Frage, wie die asymptotische Analyse zu beschreiben sei, schreibt Bruijn in *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 üblichweise nicht lohnenswert (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 28) oder sogar unmöglich (vgl. Mehlhorn, 1984, S. 37) – gemäß Bruijn ist in diesem Fall eine Ermittlung mithilfe asymptotischer Methoden ratsam.. 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 T (vgl. Shaffer, 2011, S. 63).

Diese Größenordnung des Wachstums bei steigendem bzw. großen Werten von n wird in der einschlägigen Literatur oft in der O-Notation angeschrieben (vgl. Aho, Ullman und Hopcroft, 1974, S. 2, Knuth, 1997, S. 107, Horowitz, Sahni und Rajasekaran, 1997, S. 29, ...).

Kann die Effizienz eines Algorithmus als

$$T(n) = an^2 + bn + c \tag{2.1}$$

ausgedrückt werden (wobei a, b und c beliebige von n unabhängige Konstanten sind), so kann (2.1) mithilfe dieser O-Notation zu

$$T(n) = O(n^2) \tag{2.2}$$

vereinfacht werden.

Diese Vereinfachung verleiht dem Umstand Ausdruck, dass hier nur der Term n^2 von Interesse ist. Terme niederer Ordnung wie bn und c, und konstante Faktoren wie a werden bei großem bzw. wachsendem n relativ bedeutungslos (vgl. Cormen u. a., 2001, S. 28), und können so „weggelassen werden“. Die O-Notation ersetzt also die Kenntnis einer Zahl durch das Wissen, dass eine solche Zahl existiert (vgl. Bruijn, 1958, S. 3).

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).

n	$f(n)$	$g(n)$	$\frac{f(n)}{g(n)}$
10^0	1	18	0.05555555
10^1	100	135	0.74074074
10^2	10000	10215	0.97895252
10^3	1000000	1002015	0.99798905
10^4	100000000	100020015	0.99979989
10^5	10000000000	10000200015	0.99997999
10^6	1000000000000	1000002000015	0.99999799
10^7	100000000000000	100000020000015	0.99999979
10^8	10000000000000000	10000000200000016	0.99999997
10^9	1000000000000000000	1000000002000000000	0.99999999

Tabelle 2.1: Tabelle der Funktionswerte zweier Funktionen $f(n) = n^2$ und $g(n) = n^2 + 2n + 15$. Bei steigendem n nähern sich die Funktionswerte von f und g einander an, der Quotient $\frac{f(n)}{g(n)}$ konvergiert gegen 1.

Tabelle 2.1 veranschaulicht diese Vernachlässigbarkeit von Termen niederer Ordnung und von konstante Faktoren bei großem n.

Komplexität Die Funktion $T(n)$, also die Laufzeit eines Algorithmus in Abhängigkeit der Eingabegröße n wird auch *Zeitkomplexität* des Algorithmus genannt. Das asymptotische Verhalten von $T(n)$ für große Werte von n wird *asymptotische Zeitkomplexität* genannt (vgl. Aho, Ullman und Hopcroft, 1974, S. 2). Im folgenden ist mit *Komplexität* eines Algorithmus die asymptotische Zeitkomplexität des Algorithmus gemeint.

Ein Algorithmus in $O(n^2)$ hätte demzufolge eine „Komplexität von n^2“ – für große n kann seine Laufzeit durch die Funktion $f(n) = n^2$ beschrieben werden.

Definition der O-Notation Genauer beschreibt $O(g(n))$ für eine gegebene Funktion $g(n)$ (im Fall von (2.2) also $g(x) = n^2$) die Menge von Funktionen (vgl. Mehlhorn, 1984, S. 37)

$$O(g(n)) = \left\{ f(n) \ : \begin{array}{l} \text{es existieren positive Konstanten } c \text{ und } n_0 \text{ derart, dass} \\ 0 \leq f(n) \leq c \cdot g(b) \text{ für alle } n \geq n_0 \end{array} \right\}$$

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

2.4 Praxisnahe Komplexitäten

Einige der in der in diesem Kontext am häufigsten vorkommenden Komplexitäten sind n^2, $\log n$ und $n \log n$[1](vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 38). (Anders gesagt sind also viele Sortieralgorithmen in $O(n^2)$, $O(\log n)$ oder $O(n \log n)$.) Tabelle 2.2 veranschaulicht diese Komplexitäten anhand von Funktionswerten für diverse Werte von n.

Der in Tabelle 2.2 ebenfalls dargestellte *Quotient* der jeweiligen Funktionswerte und n zeigt das Verhältnis eines Algorithmus dessen Laufzeit linear wächst, zu einem Algorithmus dessen Laufzeit durch eine der dargestellten Komplexitäten zu beschreiben ist. So ist ein Algorithmus in $O(\log n)$ für $n = 256$ fast 50-fach schneller als ein Algorithmus in $O(n)$.

Anzumerken ist jedoch, dass die O-Notation die *asymptotische* Wachstumsrate darstellt (bzw. die Komplexität eines Algorithmus als asymptotisches Verhalten zu sehen ist). So kann etwa ein Algorithmus in $O(n^2)$ für kleine n schneller als ein Algorithmus in $O(n)$ sein.

[1] Alle in dieser Arbeit behandelten Sortieralgorithmen haben eine dieser Komplexitäten.

In diesem Kontext anmerken, dass O-Notation die asymptotische Wachstumsrate *darstellt – $O(n^2)$ kann für kleine n schneller als $O(n)$ sein.*

n	$\frac{n}{n}$	n^2	$\frac{n}{n^2}$	$\log n$	$\frac{n}{\log n}$	$n \log n$	$\frac{n}{n \log n}$
1	*1*	1	*1*	0		0	
4	*1*	16	*0.25*	1.38629436	*2.88539008*	5.54517744	*0.72134752*
16	*1*	256	*0.0625*	2.77258872	*5.77078016*	44.3614195	*0.36067376*
64	*1*	4096	*0.015625*	4.15888308	*15.3887471*	266.168517	*0.24044917*
256	*1*	65536	*0.00390625*	5.54517744	*46.1662413*	1419.56542	*0.18033688*
1024	*1*	1048576	*0.00097656*	6.93147180	*147.731972*	7097.82712	*0.14426950*
4096	*1*	16777216	*0.00024414*	8.31776616	*492.439907*	34069.5702	*0.12022458*
16384	*1*	268435456	*0.00006103*	9.70406052	*1688.36539*	158991.327	*0.10304964*
65536	*1*	4294967296	*0.00001525*	11.0903548	*5909.27888*	726817.498	*0.09016844*

Tabelle 2.2: Werte der Funktionen $f(n) = n$, $f(n) = n^2$, $f(n) = \log n$ und $f(n) = n \log n$ mit nebengestelltem, *kursiv gesetztem*, Quotient aus dem jeweiligem Funktionswert und n. Leerstehende Felder repräsentieren ein undefiniertes Ergebnis. Es gilt $n = 2^0, 2^2, 2^4, \ldots, 2^16$.

2.5 Ermittlung

Im folgenden Abschnitt werden Komplexitätsanalysen diverser Pseudocode-Ausschnitte (vgl. Shaffer, 2011, 69ff) präsentiert.

```
1  a = 2 · A[i]
2  if a ≤ b/4
3        return a
```

Alle vorkommenden Operationen — die Zuweisung, Multiplikation und der Arrayzugriff in Zeile 1; der Vergleich und die Division in Zeile 2; die **return** Anweisung in Zeile 3 — sind einfache Operationen (siehe Abschnitt 2.1). Folglich ist der Ausschnitt in $O(1)$, seine Laufzeit ist konstant und unabhängig von den behandelten Werten.

```
1  s = 0
2  for j = 0 to n
3        s = s + 1
```

Zeile 1 ist in $O(1)$. Die **for** Schleife in Zeile 2 wird n Mal wiederholt. Nachdem der Körper der Schleife, Zeile 3, in $O(1)$ ist, ist die Schleife in $O(n)$. Demzufolge ist der gesamte Ausschnitt in $O(n)$.

```
1  s = 0
2  for j = 0 to n
3        for i = 0 to j
4              s = s + 1
5  for k = 0 to n
6        A[k] = s/k
```

Dieser Ausschnitt ist ein Beispiel eines etwas komplexeren Algorithmus mit drei **for** Schleifen.

Die erste Zeile ist in $O(1)$. Die letzte Schleife in Zeile 5 ist wie im vorhergehenden Beispiel in $O(n)$ nachdem ihr Körper in Zeile 6 in $O(1)$ ist (auch Zuweisungen in Arrays sind einfache Operationen).

Die erste Schleife in Zeile 2 läuft n Mal, bei jeder Iteration wächst ihr Schleifenzähler j um 1. Wie oft die „innere“ Schleife in Zeile 3 läuft ist abhängig von j: Für $j = 0$ läuft sie nicht, für

$j = 1$ läuft sie einmal, für $j = 2$ läuft sie zweimal, ..., für $j = n$ läuft sie n Mal. Nachdem gilt, dass (vgl. Y.-G. Chen und Fang, 2007)

$$\sum_{i=0}^{n} i = \frac{n(n-1)}{2} = O(n^2)$$

und die Komplexität des Körpers der inneren Schleife in $O(1)$ ist, ist der gesamte Schleifenkonstrukt in $O(n^2)$.

Damit ist der Ausschnitt also in $O(1) + O(n) + O(n^2)$ was zu $O(n^2)$ vereinfacht werden kann.

Andere Berechnungsmodelle Aho, Ullman und Hopcroft, 1974 (*The Design and Analysis of Computer Algorithms*) *Nur als Notiz, falls noch Zeit für nähere Ausführungen bleibt.*

Kapitel 3

Laufzeitermittlung als Effizienzangabe

In diesem Kapitel wird eine Methode zur Ermittlung der Laufzeit von Algorithmen dargestellt.

Abschnitt 3.1 beschäftigt sich, als Ausgangspunkt dieses Kapitels, mit der Aufstellung einer Frage und einer Konkretisierung des Ziels: Der Ermittlung einer „praktischen Effizienz“.

Um diese Frage zu konkretisieren und umsetzbar zu machen, werden in Abschnitt 3.2 Arten und Größen der Eingabemengen definiert. Abschnitt 3.3 behandelt die Bereitstellung einer Testumgebung.

Ein konkreter Weg zur „praktischen Effizienz“ wird in Abschnitt 3.4 gegeben, Abschnitt 3.5 behandelt einen Konstrukt zur Orchestrierung dieser Vorgehensweise.

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

Hier sind „A theoretician's guide to the experimental analysis of algorithms“ (Johnson, 2002) und A Guide to Experimental Algorithmics *(McGeoch, 2012) wichtig. Shaffer, 2011, S. 83 kann als Argumentation für das ausgeprägte Verwenden der* Standard Library *ausgelegt werden nachdem hier potentielle Vorurteile bzw. Ungleichheiten bei der Programmierung de facto wegfallen.*

Der Werdegang der Empirie in den Computerwissenschaften ist nicht unspannend: Zu Zeiten von Johnson, 2002 war die empirische Analyse als Feld offenbar noch bei weitem nicht so weit fortgeschritten und verbreitet wie im Erscheinungsjahr von McGeoch, 2012. Beide beinhalten weitgehend äquivalente Kernaussagen, aber erstere Publikation ist noch bei weitem unausgereifter als letztere.

3.1 Formulierung einer Frage

Es gilt die Laufzeit eines Algorithmus zu ermitteln und diese annähernd als Funktion $T(n)$ (wobei $T(n)$ die Zeit und n die Eingabegröße ist) darzustellen. Das Ziel deckt sich also mit jenem der asymptotischen Analyse aus Kapitel 2. Genauer gilt es, ebenfalls wie in der asymptotischen Analyse, eine solche Funktion für diverse Arten von Eingabemengen zu ermitteln (McGeoch, 2012, S. 27).

Nach McGeoch, 2012, S. 10 kann „the experimental process“ im Kontext der empirischen Algorithmusanalyse im Wesentlichen grob in die Formulierung einer Frage, die Bereitstellung einer, die Frage behandelnden, Testumgebung bzw. eines Testprogramms und die Ausführung des Testprogramms aufgegliedert werden.

Eine grobe Formulierung der Frage ergibt sich schon aus der vorhergehenden Einleitung dieses Kapitels:

„Wie viel Zeit benötigt ein Sortieralgorithmus um Eingabemengen verschiedener Art und Größe in eine sortierte Ausgabemenge zu überführen?“

3.2 Eingabemengen

Für die Ermittlung der praktischen Effizienz werden sortierte, invertierte und zufällig geordnete Eingabemengen verwendet. Um diese Mengen zu generieren werden die Funktionen sets :: sorted, sets :: inverted und sets :: random verwendet.

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

Listing 3.1: Funktionen welche Mengen der verwendeten Eingabemengearten generieren.

(Die Funktion utils :: random_shuffle, welche für das Generieren der zufällig geordneten Menge verwendet wird, verwendet den Mersenne-Twister Algorithmus um hochwertige Pseudozufallszahlen zu generieren und damit die Eingabemenge „durchzumischen“. Siehe Listing A.7 für die verwendete Implementation der Funktion.)

Eine ausführlichere Auswahl an Eingangsmengen welche an einen sie bearbeitenden Algorithmen angepasst ist — wie sie für eine eingehende Analyse eines einzelnen Algorithmus angebracht wäre (vgl. McGeoch, 2012, 27ff) — würde das allgemeiner gesetzte Ziel dieser Arbeit verfehlen.

3.3 Testumgebung

Es gilt die Zeit, welche eine Funktion für die vollständige Ausführung benötigt, zu messen. Die Funktion wird ausgeführt, die Zeit unmittelbar vor (*start*) und nach (*end*) der Ausführung wird gemessen. Die Laufzeit des Algorithmus ist nun gleich $end - start$.

Eine konkrete Implementation einer Klasse zur Messung der Laufzeit eines Algorithmus ist in Listing 3.2 gegeben.

```
1  class experiment {
2      std::function<void()> function;
3
4      using clock_t = std::chrono::high_resolution_clock;
5
6  public:
7      using duration_t = clock_t::duration;
8
9      explicit experiment(std::function<void()> function)
10         : function(std::move(function)) {};
11
12     [[nodiscard]] duration_t run() const;
13 };
14
15
16 experiment::duration_t experiment::run() const {
17     const auto start = clock_t::now();
18
19     function();
20
21     const auto end = clock_t::now();
22
23     return duration_t{end - start};
24 }
```

Listing 3.2: 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 der für das System kleinstmöglichen Einheit angegeben ist (vgl. ISO/IEC 14882, 2017, S. 652). Auf den verwendeten Testsystem wird die Zeit in Nanosekunden gemessen, $1ns = 10^{-9}s = 0,000000001s$.

Algorithmen mit Eingabewerten Der Konstruktor in der Klasse aus Listing 3.2 erwartet als einzige Eingabe eine Variable des Typs std::function<**void**()>, also eine Funktion ohne Eingabe- und Rückgabewerte. Dies ist der Fall um größtmögliche Flexibilität in der Verwendung der Testumgebung zu gewährleisten.

Die in dieser Arbeit behandelten Algorithmen haben alle einen oder mehrere Eingabewerte, sie erfüllen also nicht die Form wie sie vom Konstruktor erwartet wird. Um ein Experiment mit einer Funktion eines anderen Typs als **void**() zu initialisieren wird ein „*wrapper*“ verwendet:

```
const size_t size = 1337;
void func(const size_t s) {  ...  }
const auto wrapper = [&]() { func(size);  };
```

```
const auto time = experiment(wrapper).run();
```

Im obenstehenden Beispielcode wird ein Lambda-Ausdruck verwendet um eine „umwickelnde Funktion“, einen sogenannten *wrapper*, zu erstellen. Ein Aufruf von *wrapper* führt zum Aufruf der Funktion *func* mit dem Eingabeparameter size bzw. 1337.

3.4 Ermittlung

Das Ziel ist es, eine Funktion aufzustellen, bzw. zu approximieren, welche die Laufzeit eines Algorithmus in Abhängigkeit der Eingabegröße darstellt (vgl. McGeoch, 2012, S. 37). Um diesem Ziel nachzukommen gilt es die Eingabemenge in wachsende Untermengen aufzuteilen, diese Untermengen zu sortieren und die dafür benötigte Zeit (in Kombination mit der Größe der jeweiligen Untermenge) auszugeben.

Hierfür ist eine Klasse **class** benchmark gegeben, die mit einer Eingabemenge, einem Sortieralgorithmus, einer Angabe zur Art der Ermittlung der Untermengen und der Anzahl der zu erstellenden Untermengen konstruiert wird.

```
1  class benchmark {
2  public:
3      using algorithm_t = sorters::sorter_t<sets::iterator_t>;
4
5      struct result : public std::map<size_t, experiment::duration_t> {
6          // ...
7      }
8
9      // ...
10
11     benchmark(sets::set_t set, algorithm_t algorithm, benchmark::step_type_t step_type,
           size_t total_chunks);
12
13     [[nodiscard]] benchmark::result run() const;
14 };
```

Listing 3.3: Klasse zur Approximation einer Funktion der Laufzeit eines Algorithmus in Abhängigkeit der Eingabegröße.

In Listing 3.3 wird ein vereinfachter Einblick in eine Schnittstelle zu einem, diese Anforderungen erfüllenden, Mechanismus gegeben[1].

Die Funktion benchmark::run retourniert eine Instanz der Klasse **class** benchmark::result die wiederum eine Menge von Größenangaben und damit assoziierten Zeitangaben ist. Eine von dieser Funktion zurückgegebene Menge von Paaren hat beispielsweise die Form

$$\{(1, 0.0768125), (2, 0.104375), (3, 0.129938), (4, 0.170375), \ldots, (n, t)\}$$

Sie ist so interpretieren, dass der gegebene Algorithmus für das Sortieren einer Untermenge der Größe n der gegebenen Menge t Zeiteinheiten benötigt hat.

Art der Schrittgröße In benchmark.run erfolgt also eine Teilung der ursprünglichen Eingabemenge in eine gewisse Anzahl von Untermengen. Die Art der Wachstumsrate der Größe dieser Untermengen wird im Konstruktor der Klasse festgelegt, wo entweder ein lineares oder ein quadratisches Wachstum gewählt werden kann (siehe Abbildung 3.1).

[1] Für eine vollständige Implementation siehe src/benchmark.h und src/benchmark.cpp.

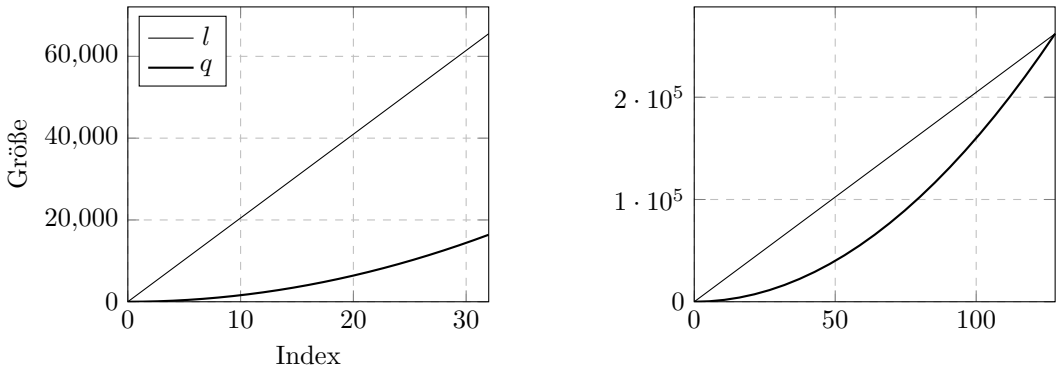


Abbildung 3.1: Graphen der Untermengengrößen bei einer gesamten Mengengröße von 2^{18}, geteilt auf 128 Untermengen mit linearer (l) und quadratischer (q) Schrittgröße.

3.5 Orchestrierung

In den vorhergehenden Abschnitten wurde

- eine Definition für einige Eingabemengen mit ihren korrespondierenden Generatoren gegeben (Abschnitt 3.2).
- eine Testumgebung (**class** experiment) zur Ermittlung der Laufzeit eines Algorithmus mit einer bestimmten Eingabemenge und Größe (Abschnitt 3.3).
- einen Mechanismus zur Laufzeitermittlung für steigende bzw. variierende Eingabegrößen (**class** benchmark, Abschnitt 3.4).

Noch abgehend ist ein Mechanismus zur *Orchestrierung* bzw. *Instrumentalisierung* der bestehenden Teile. Genauer gilt es ein ausführbares Programm bereitzustellen, welches mithilfe der benchmark Klasse für alle möglichen Kombinationen aus den Algorithmen und Eingabearten Benchmarkresultate ermittelt und diese ausgibt. Ebenfalls gilt es einen Mechanismus zur einfachen Kontrolle des Umfangs und der Präzision dieser Messungen bereitzustellen.

```
out/
├─ heap_inverted
├─ heap_random
├─ heap_sorted
├─ insertion_inverted
└─ ...
```

(a) Struktur des Ausgabeordners, die Endung „/“ repräsentiert einen Ordner.

n	t in μs
2048	1425.91
4096	3702.23
6144	4790.03
8192	6419.75
10240	8436.86
⋮	⋮

(b) Inhalt der ersten 5 Zeilen einer beispielhaften Ausgabedatei.

Abbildung 3.2: Beispielhafte Ordner- und Dateistruktur nach Ausführung des Programms zur Benchmark–Orchestrierung.

Diese Anforderungen erfüllt das *benchmark*-Programm bzw. der Code in src/main.cpp. Standardmäßig erstellt es einen Ordner *out* und befüllt diesen mit gesammelten Daten. Für jede mögliche Kombination aus Algorithmus und Eingabeart wird eine Tabelle mit Untermengengrößen und Zeitangaben erstellt, siehe Abbildung 3.2.

Eine Steuerung der Vorgänge zur Datensammlung und -aufbereitung im Programm kann durch Parameter auf der Kommandozeile erfolgen. So sagt etwa der Befehl „benchmark −o data −s 512“ aus, dass ein Ordner *data* (anstelle von out) erstellt werden soll. Ebenfalls soll die größte zu sortierende Liste genau 512 Elemente haben. Alle möglichen Befehle sind Abbildung 3.3 zu entnehmen.

```
Usage: benchmark [options]
Options:
  −h, −−help          Display this information and exit.
  −o, −−output        Sets the output path. (default: ./out)
  −s, −−size          Sets the sample size. (default: 262144 i.e. 2^18)
  −c, −−chunks        Sets the number of chunks that the set will be devided into.
                      (default: 128)
  −t, −−step−type     Specifies the step type to use, one of 'linear' or
                      'quadratic'. (default: linear)
  −a, −−average       Repeat the benchmarking a given number of times and output
                      the average results of all runs. (default: 0)
  −m, −−median        Like −a, except it outputs the median. (default: 0)
  −r, −−randomize     Randomize the order in which the different combinations
                      between sorters and sets are benchmarked. (default: false)
```

Abbildung 3.3: Anwendungsbeschreibung des „benchmark“-Programms, ausgegeben nach Ausführung von benchmark −h.

Kapitel 4

Behandelte Algorithmen und ihre theoretische Effizienz

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

Konkrete Implementation der folgenden Algorithmen sind Anhang A zu entnehmen.

4.1 Insertion Sort

Wie schon in Abschnitt 1.4 ausgeführt, baut der Insertion Sort 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 4.1 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
```

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 $j = 2, 3, \ldots A.\mathit{length}$ gilt nun $A[j-1] \leq \mathit{key}$: der Körper der **while** Schleife auf Zeile 5 wird also nie ausgeführt. Der Algorithmus ist in diesem Fall in $O(n)$ (vgl. Cormen u. a., 2001, 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 in diesem Fall also in $O(n^2)$ (vgl. Cormen u. a., 2001, 28f).

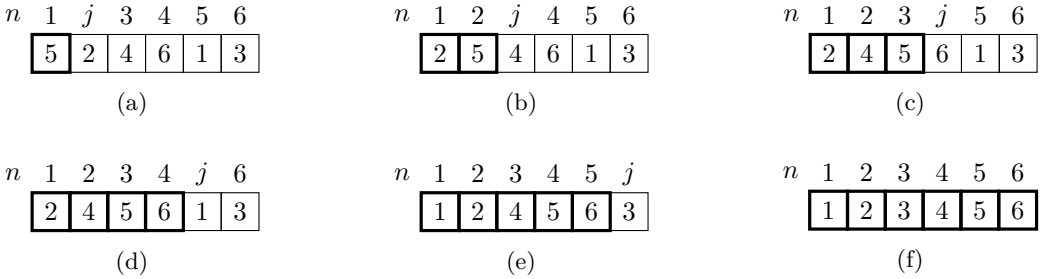


Abbildung 4.1: Illustration des Arrays $A = \{5, 2, 4, 6, 1, 3\}$ während es von INSERTION-SORT(A) bearbeitet wird. Über einem Rechteck steht sein Index in A, in den Rechtecken steht der jeweilige Wert von A an diesem Index. Fett gedruckte Rechtecke sind Teil des bereits sortierten Subarrays $A[1\,..\,j-1]$. (a) bis (e) zeigen die fünf Iterationen der **for**-Schleife auf Zeilen 1–8 beginnend mit $j = 2$ bis $j = A.\,length$ 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.

4.2 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 Cormen u. a., 2001, S. 170.

PARTITION(A, p, r)
1 $x = A[r]$
2 $i = p - 1$
3 **for** $j = p$ **to** $r - 1$
4 **if** $A[j] \leq x$
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 4.2.

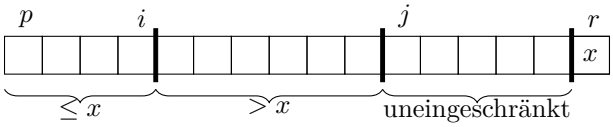


Abbildung 4.2: Die vier Regionen die von PARTITION auf einem Subarray $A[p\,..\,r]$ behandelt werden. Die Werte in $A[p\,..\,i]$ sind alle $\leq x$, die Werte in $A[i+1\,..\,j-1]$ sind alle $> x$, und $A[r] = x$. Das Subarray $A[j\,..\,r-1]$ kann jegliche Werte beinhalten. Die Abbildung und Beschreibung wurden aus Cormen u. a., 2001, S. 173, Abbildung 7.2 übernommen.

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $\quad q =$ PARTITION(A, p, r)
3 $\quad$ QUICKSORT$(A, p, q-1)$
4 $\quad$ QUICKSORT$(A, q+1, r)$

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

$$\begin{aligned} T(n) &= T(n-1) + T(0) + O(n) \\ &= T(n-1) + O(n) \end{aligned} \tag{4.1}$$

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. Sedgewick und Flajolet, 2013, S. 49). Diese Laufzeit tritt auch auf, wenn die Eingabemenge bereits sortiert ist (vgl. Cormen u. a., 2001, 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 $T(n) = O(n \log n)$ nach Cormen u. a., 2001, S. 94.

4.3 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, Ullman und Hopcroft, 1974, 87f, siehe Abbildung 4.3). 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 $A[1\,..\,A.\mathit{heap\text{-}size}]$ 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** $\lfloor i/2 \rfloor$

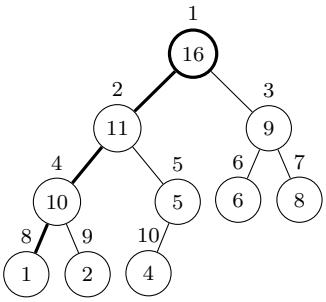
LEFT(i)
1 **return** $2i$

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

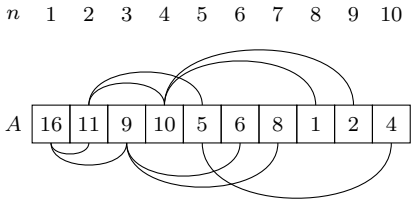
PARENT, LEFT und RIGHT sind aus Cormen u. a., 2001, S. 152 entnommen.

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, Sahni und Rajasekaran, 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 4.3 (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 4.3: 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. Angelehnt an Cormen u. a., 2001, S. 152 und Aho, Ullman und Hopcroft, 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 $h = \log n$ ermittelt werden, somit ist die Effizienz im ungünstigsten Fall in $O(\log n)$ (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, Fenner und Frieze, 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[1]. 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.

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.

[1] 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(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. ISO/IEC 14882, 2017, 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 $O(n \log n)$ nachdem jeder der $n - 1$ Aufrufe an MAX-HEAPIFY in $O(\log n)$ ist (vgl. Cormen u. a., 2001, S. 160).

4.4 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 kann die Effizienz des MERGE-SORT durch eine Rekursionsgleichung

$$T(n) = \begin{cases} O(1), & \text{wenn } n = 1, \\ 2T(n/2) + O(n), & \text{wenn } n > 1. \end{cases} \tag{4.2}$$

beschrieben werden (vgl. Cormen u. a., 2001, S. 36). Der Summand $O(n)$ ist hier die Effizienz von MERGE. (4.2) deckt sich mit (4.1) sowohl im günstigsten als auch im ungünstigsten Fall, damit ist die Effizienz von QUICKSORT in $O(n \log n)$ (vgl. Cormen u. a., 2001, S. 36).

Kapitel 5

Vergleich der theoretischen und der praktischen Effizienzen

Dieses Kapitel beschäftigt sich mit der finalen Forschungsfrage dieser Arbeit, namentlich wie sich die (nun bereits ermittelte) theoretische Effizienz zu der praktischen Effizienz verhält.

Dies setzt eine Ermittlung der praktischen Effizienz voraus. Basierend auf den Methoden in Kapitel 3 erläutert Abschnitt 5.1 die Art und Weise auf der der verwendete Datenstamm generiert wird.

Abschnitt 5.2 beantwortet die angeführte Frage durch Ermittlung der Approximierbarkeit der praktischen aus der theoretischen Effizienz.

Abschnitt 5.3 behandelt die Idee des „besten“ Algorithmus.

5.1 Konkrete Ermittlung der Daten

Um den verwendeten Datenstamm zu generieren wird ein einzelnes Skript verwendet, welches Aufrufe an das benchmark-Programm und die Ablage und Weiterverarbeitung der Daten automatisiert. Somit wird eine Reproduktion der Daten erleichtert, es gilt nur einen einzelnen Befehl auszuführen. Das Skript ist in scripts/data/generate.js abgelegt[1].

Zur Generierung der Daten wurde durchgehend ein ThinkPad T480s (Modellnummer 20L8S02E00) verwendet, auf dem ein Intel Core i7-8550U (8 Threads × 1.80 GHz Base bzw. 4.00 GHz Turbo) und 16 GB DDR4 RAM (2400 MT/s mit NVMe SSD als Swap) verbaut sind.

Alle ermittelten Daten sind im Ordner project/data abgelegt. Eine Darstellung aller Daten in project/data/canon (dem Haupt-Datenstamm) ist in Anhang B gegeben.

5.2 Approximierbarkeit der Effizienzen

Ein Ziel dieser Arbeit ist es, zu ermitteln wie sich die theoretische Effizienz zur praktischen Effizienz verhält. Zu diesem Zweck wird folgend überprüft, wie gut die praktische aus der theoretischen Effizienz „vorhersagbar“ ist.

Für jede Kombination aus Algorithmus und Eingabemengenart[2] wird basierend auf den Ergebnissen aus Kapitel 4 eine Funktion der erwarteten Effizienz aufgestellt.

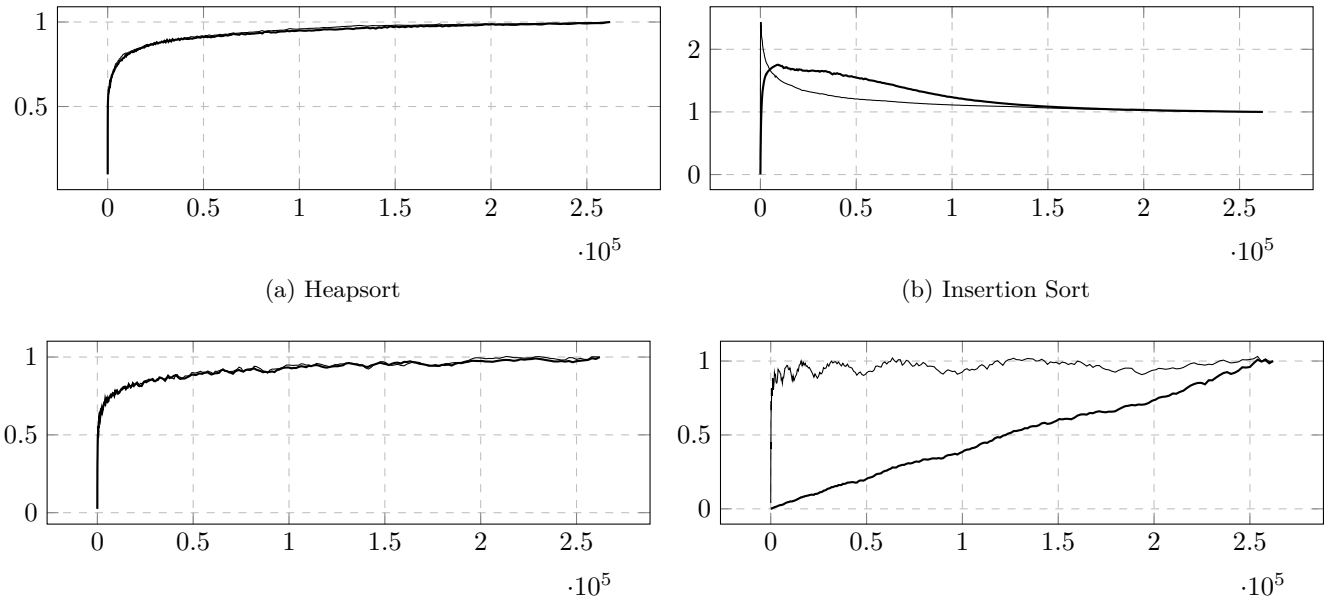
[1] In selbigem Ordner ist auch eine README-Datei zu finden welche die Bedienung des Skripts näher erläutert.
[2] Mit Ausnahme der zufällig sortierten Art, welche folgend nicht verwendet wird.

Diese Funktion der erwarteten Effizienz basiert nun allerdings ausschließlich auf der theoretischen Effizienz, die wiederum auf der O-Notation aufbaut. Werte dieser Funktion sind nur für große n gültig — tatsächlich beschreibt sie nur die Wachstumsrate, nicht aber konkrete Messwerte. Um anhand dieser Funktion sinnvoll Rückschlüsse auf die praktische Effizienz ziehen zu können ist also eine Skalierung notwendig, die wiederum einen konkreten Messwert vorraussetzt.

Ist also die Zeit t_s die ein Algorithmus für das Sortieren einer Menge der Größe n_s bekannt (als Wertepaar folgend als *Skalierungsvektor* bezeichnet), kann die unskalierte Funktion f der erwarteten Effizienz anhand dieser (einzelnen) Messung zu einer skalierten Version

$$f_s(n) = f(n) \cdot \frac{t_s}{f(n_s)}$$

transformiert werden[3].



(a) Heapsort

(b) Insertion Sort

(c) Merge Sort

(d) Quicksort

Abbildung 5.1: Quotient zwischen erwarteter und tatsächlicher Zeit, die der jeweilige Algorithmus für das Bearbeiten der jeweiligen Eingabemenge benötigt hat. Werte < 1 sind langsamer, Werte > 1 schneller als die Projektion. Normale Graphenstärke repräsentiert eine sortierte Liste kombiniert mit der schnellsten Projektion, fette Graphenstärke eine invers sortierte Liste mit der langsamsten Projektion. Die erwartete Zeit wurde mit jener der größten Untermenge skaliert.

In Abbildung 5.1 ist eine Aufstellung des Größenverhältnis zwischen der skalierten erwarteten Effizienz und tatsächlichen Messwerten gegeben. Die erwartete Effizienz wurde jeweils mithilfe des letzten Messwertes (also des Messwertes mit der größten Untermengengröße) skaliert.

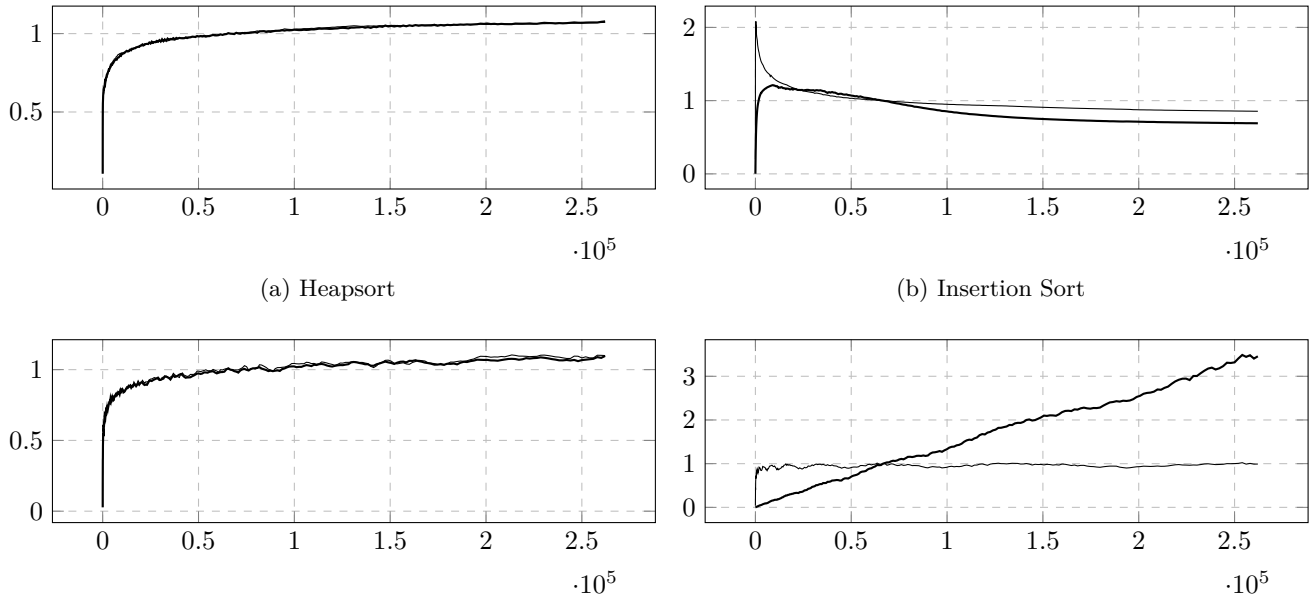
Daraus lässt sich schließen, dass für Heap- und Insertion Sort ab einer Untermengengröße von etwa 150000 (das sind etwa 57% der größten Untermenge) die Messwerte mit einer Präzision ≈ 1, de facto perfekt, vorhergesagt wurden. Beim Merge Sort nimmt die Präzision ebenfalls mit der Annäherung zum Messwert zu, die Präzision der Messung schwankt allerdings stärker.

Die Präzision beim Quicksort ist im besten Fall, mit sortieren Listen, ähnlich oder besser jener bei Heap-, Insertion und Merge Sort. Im schlechtesten Fall ist sie allerdings bedeutend schlechter als jene für andere, er ist langsamer als projeziert. (Daraus zu schließen, dass der Quicksort langsamer

[3]Der Effizienz halber würde in einer tatsächlichen Implementation dieser Methode der Faktor $t_s/f(n_s)$ nur einmal berechnet, und dann wiederverwendet werden.

als andere Algorithmen wäre allerdings falsch: Tatsächlich ist er deutlich schneller, siehe Anhang B.)

Auffallend ist, dass alle Algorithmen für Eingabemengen die deutlich kleiner als die Mengengröße im Skalierungsvektor sind, eine niedrige Approximationspräzision haben. Beim Insertion Sort ist dies besonders auffällig, er ist in diesem Fall bedeutend schneller als die Approximation. Das ist unter anderem auch bemerkenswert, weil dies bei keinem anderen Algorithmus der Fall ist — andere Algorithmen sind hier durchwegs deutlich langsamer als vorhergesagt.



(a) Heapsort

(b) Insertion Sort

(c) Merge Sort

(d) Quicksort

Abbildung 5.2: Analog zu Abbildung 5.1. Die erwartete Zeit wurde mit jener der mittleren Untermenge skaliert.

Abbildung 5.2 zeigt, dass auch bei einer Veränderung des Skalierungsvektors — in diesem Fall zur Mitte der ermittelten Werte — die vorhergehenden Observationen noch immer halten. Für Untermengen die größer als der Skalierungsvektor sind, wird der Quicksort im schlechtesten Fall zunehmend schneller als die Projektion, ein Umstand der aufgrund der Wahl des Skalierungsvektors in Abbildung 5.1 nicht ersichtlich war.

Daraus lässt sich schließen, dass die für den Quicksort ermittelte Effizienz in der Praxis nicht besonders repräsentativ ist. Tatsächlich zeigt Abbildung 5.3, dass n und $n \cdot \log n$ bedeutend besser für eine Approximation der praktischen Effizienz geeignet sind. Das ist eines der Defizite der O-Notation in diesem Kontext — sie beschreibt nur eine *obere Schranke* für sehr große n, macht jedoch keine Aussage über die Praktikabilität dieser Schranke, oder ob sie für relativ kleine n (hier hächstens 262144 bzw. 2^18) überhaupt gültig ist (siehe Abschnitt 2.3).

Reproduktion Die in diesem Abschnitt ermittelten Werte sind in Rohform in project/data/canon/comparison abgelegt. Sie können mithilfe von scripts/data/generate.js unter Eingabe des Tasknamens „supplementary/comparison“ explizit generiert werden, der konkrete zur Generierung verwendete Code liegt in scripts/data/reciprocal-approx.js.

5.3 Der „beste Algorithmus“

Die Bestimmung eines „besten“ Algorithmus setzt messbare Kriterien voraus, anhand welcher Kandidaten bewertet werden können. Im gegebenen Kontext ist das die Zeit, die ein Algorithmus

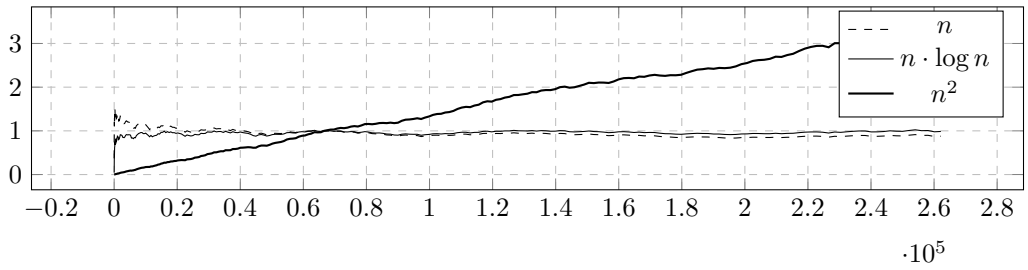
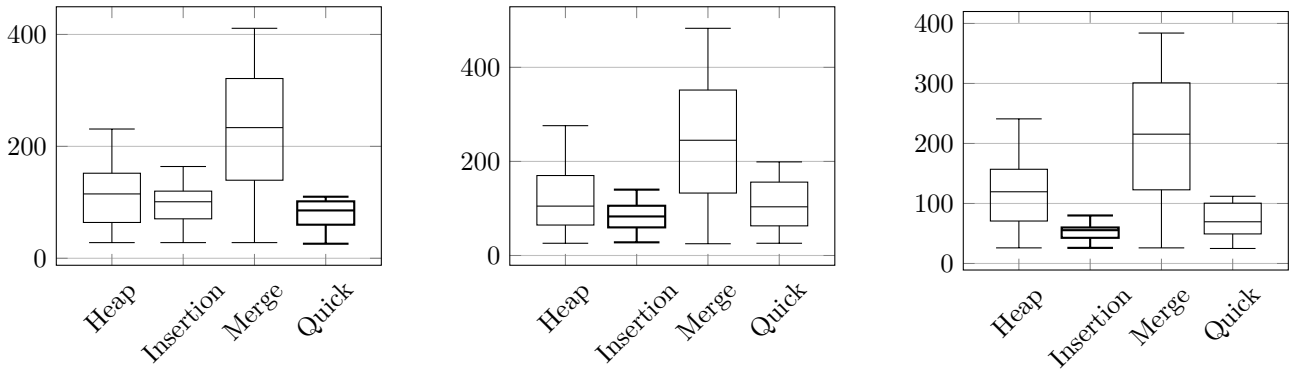


Abbildung 5.3: Quotient zwischen erwarteter und tatsächlicher Zeit die der Quicksort für das bearbeiten einer invers sortierten Liste benötigt. Verschiedene Graphen repräsentieren verschiedene Vergleichsfunktionen, skaliert mit der mittleren Untermenge.

für das Sortieren einer Liste benötigt.

Wie in Kapitel 2 ausgeführt wird, und wie aus den Abbildungen in Anhang B hervorgeht, bestehen zwischen den Algorithmen große Unterschiede in dieser benötigten Zeit, die unter anderem auf die Größe und Beschaffenheit der zu sortierenden Liste zurückzuführend sind. Aus diesem Grund gilt es für alle Kombinationen diesen Faktoren den besten *der untersuchten* Algorithmen zu ermitteln.

Erstens ist hier also wichtig festzuhalten, dass durch eine solche Analyse keine allgemeine Aussage wie „Sortieralgorithmus X ist besser als *alle* anderen Algorithmen.“ getroffen werden können — es kann nur der beste aus den Untersuchten ermittelt werden. Zweitens kann in diesem Schritt ein besserer Algorithmus nur einzeln für jede Kombination aus Parametern ermittelt werden. Der beste Algorithmus ist dann jener, der in den Kombinationen die dem Einsatzgebiet am besten Entsprechen der Beste ist.



(a) Umgekehrte Liste (b) Zufällige Liste (c) Sortierte Liste

Abbildung 5.4: Boxplots der benchmark-Ergebnisse für sehr kleine Mengen ($n \leq 16$).

Konklusion

Kapitel 1 legt mit einer Definition des Algorithmusbegriffs, Methoden zur Algorithmusbeschreibung und einer einleitenden Definition des Effizienzbegriffs im Kontext die Basis für die darauffolgenden Kapitel.

Mit der O-Notation und der dahinführenden asymptotischen Analyse wird in Kapitel 2 ein Weg gegeben, die theoretische Effizienz eines Algorithmus zu ermitteln und darzustellen. Eine solche Ermittlung erfolgt in Kapitel 4 neben der Beschreibung der im Folgenden verwendeten Algorithmen.

Eine konkrete Möglichkeit zur Bestimmung der praktischen Effizienz eines Sortieralgorithmus wird in Kapitel 3 erarbeitet. Aufbauend auf einer einfachen Testumgebung und einer Strategie zur Funktionsermittlung kann mithilfe des dargelegten „benchmark“-Programms automatisiert die praktische Effizienz von diversen Algorithmen auf diversen Eingabemengearten ermittelt werden.

Kapitel 5 beschreibt das Verhältnis zwischen praktischer und theoretischer Effizienz anhand der Approximierbarkeit der praktischen aus der theoretischen Effizienz. In Abschnitt 5.2 wird dargelegt, dass für die meisten Algorithmen ein einziger Datenpunkt und Kenntnis der theoretischen Effizienz für eine praktikable Approximation der praktischen Effizienz ausreicht. Abschnitt 5.3 beschreibt die Schwierigkeit der Ermittlung eines „besten“ Algorithmus im Allgemeinen, und legt einen Weg zur Ermittlung eines *guten* Algorithmus für einen gewissen Einsatzbereich dar.

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.

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

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

```
template <class I, class P = std::less<>>
void quick(I first, I last, P cmp = P{}) {
    auto const N = std::distance(first, last);
    if (N <= 1)
        return;

    auto const pivot = *std::next(first, N / 2);

    auto const middle1 = std::partition(first, last, [=](auto const &elem) {
        return cmp(elem, pivot); });
    auto const middle2 = std::partition(middle1, last, [=](auto const &elem) {
        return !cmp(pivot, elem); });

    quick(first, middle1, cmp);
    quick(middle2, last, cmp);
}
```

Listing A.3: Implementation des *quicksort*.

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

Listing A.4: Implementation des *heapsort*.

```
template <class BI, class P = std::less<>>
void merge(BI first, BI last, P cmp = P{}) {
    auto const N = std::distance(first, last);
    if (N <= 1)
        return;

    auto const middle = std::next(first, N / 2);

    merge(first, middle, cmp);
    merge(middle, last, cmp);

    std::inplace_merge(first, middle, last, cmp);
}
```

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

```
1  void write(std::string sub_path, const char *algo_name, const char *set_name,
       timings_t timings) {
2      std::filesystem::path file_path;
3
4      if (sub_path.size())
5          file_path += sub_path;
6
7      file_path += "/";
8
9      std::filesystem::create_directories(file_path);
10
11     file_path += algo_name;
12     file_path += "_";
13     file_path += set_name;
14
15     printf("writing %s\n", file_path.c_str());
16
17     if (std::filesystem::exists(file_path)) {
18         std::filesystem::remove(file_path);
19     }
20
21     std::ofstream os(file_path.c_str());
22
23     for (const auto [n, t] : timings) {
24         os << n << " " << t << "\n";
25     }
26 }
```

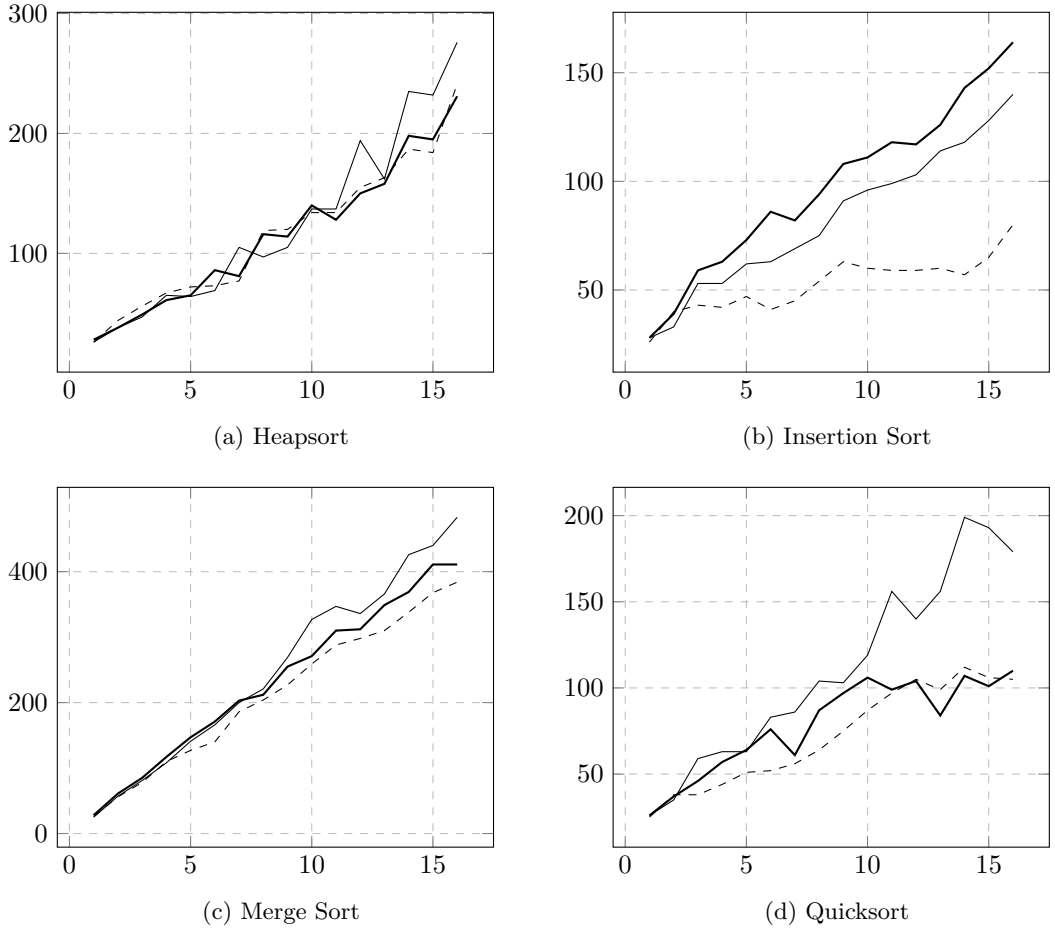
Listing A.6: Implementation einer Funktion zum Speichern des Rückgabewerts von benchmark::run aus Listing **??**.

```
1  template <class I>
2  void random_shuffle(I &&first, I &&last) {
3      std::random_device rd;
4      std::mt19937 g(rd());
5
6      std::shuffle(first, last, g);
7  }
```

Listing A.7: Implementation eines Algorithmus zum zufälligen Durchmischen einer Eingabemenge unter verwendung des Mersenne-Twister Algorithmus.

Anhang B

Daten



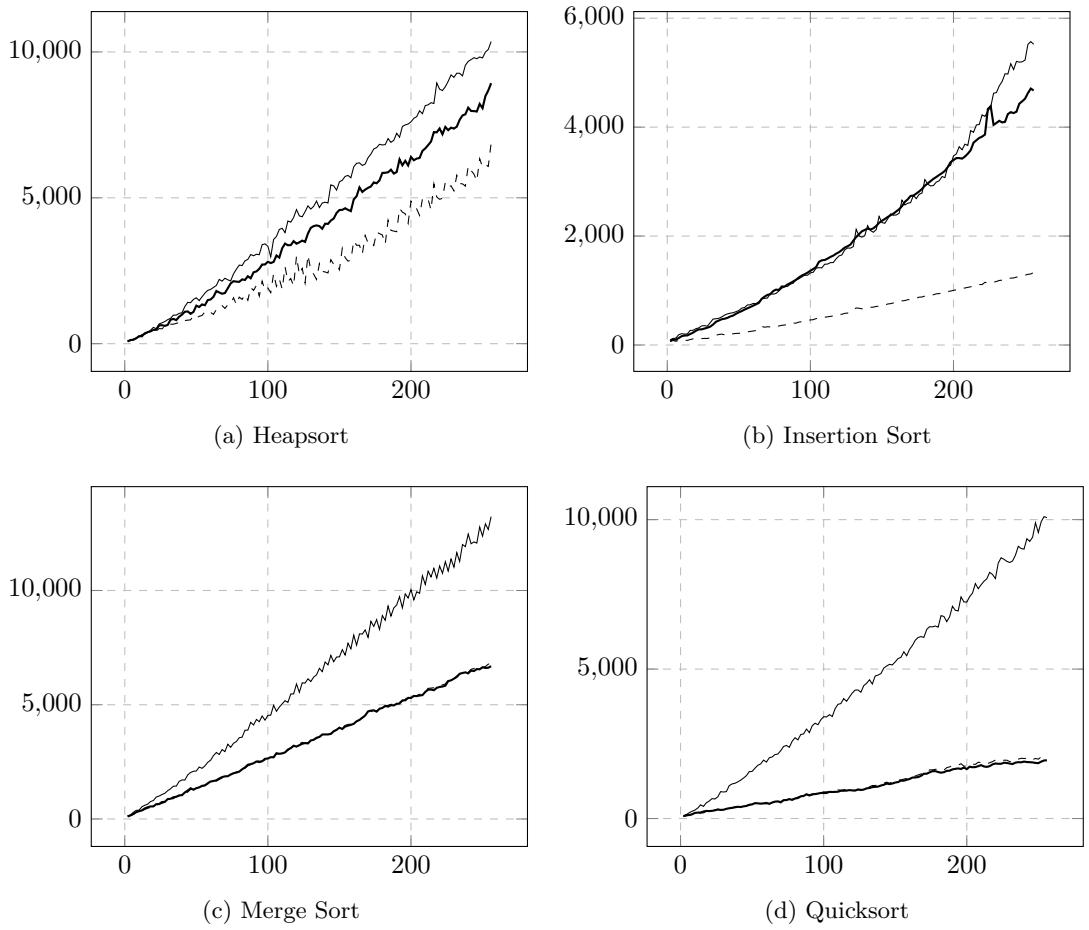
(a) Heapsort

(b) Insertion Sort

(c) Merge Sort

(d) Quicksort

Abbildung B.1: Laufzeit von Sortieralgorithmen auf sehr kleinen sortierten (normale Graphstärke), umgekehrten (fette Graphstärke) und zufällig geordneten (strichlierter Graph) Eingabemengen.



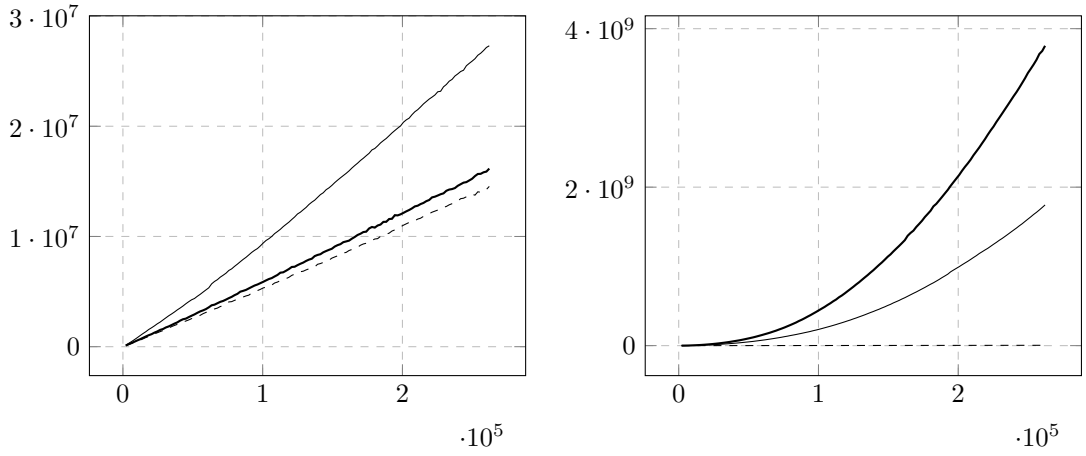
(a) Heapsort

(b) Insertion Sort

(c) Merge Sort

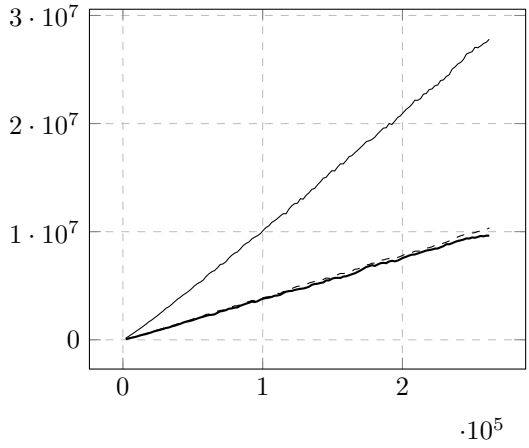
(d) Quicksort

Abbildung B.2: Laufzeit von Sortieralgorithmen auf kleinen sortierten (normale Graphstärke), umgekehrten (fette Graphstärke) und zufällig geordneten (strichlierter Graph) Eingabemengen.

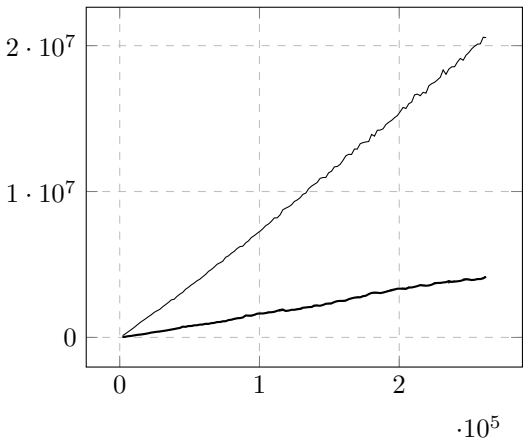


(a) Heapsort

(b) Insertion Sort

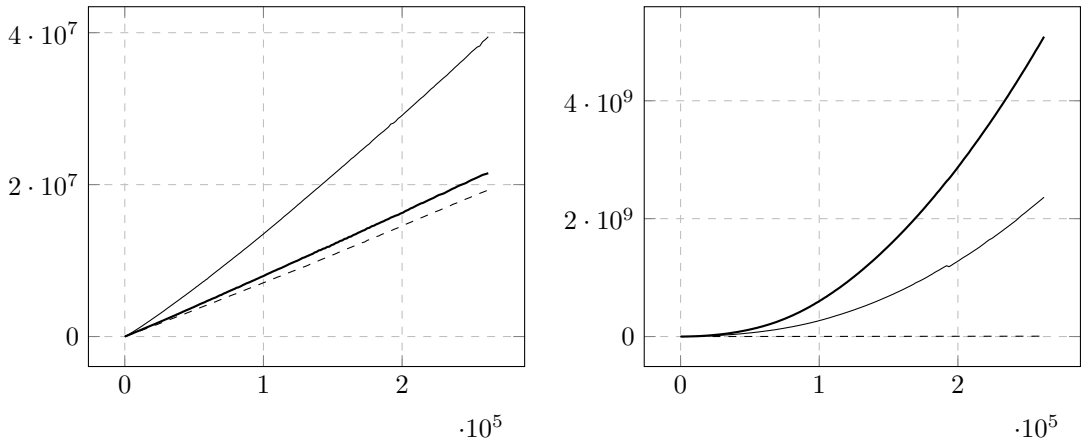


(c) Merge Sort



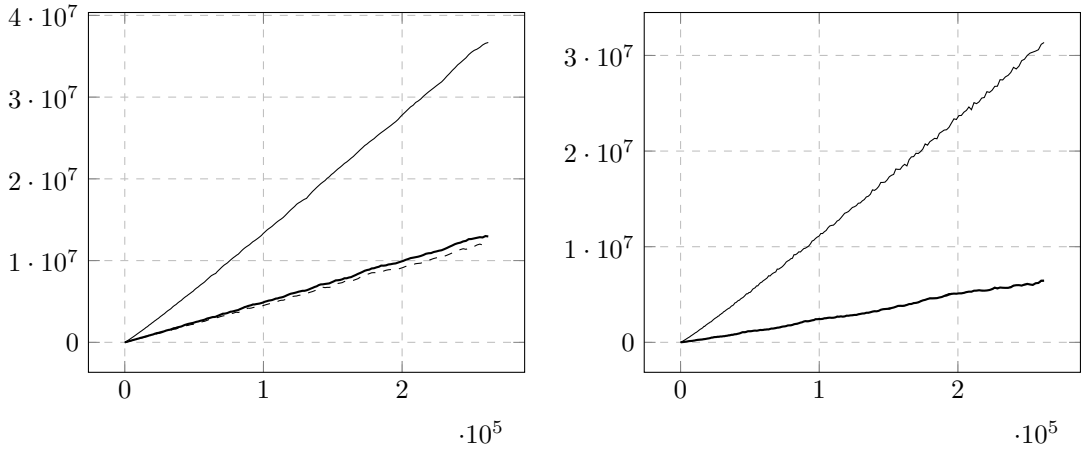
(d) Quicksort

Abbildung B.3: Laufzeit von Sortieralgorithmen auf großen sortierten (normale Graphstärke), umgekehrten (fette Graphstärke) und zufällig geordneten (strichlierter Graph) Eingabemengen.



(a) Heapsort

(b) Insertion Sort



(c) Merge Sort

(d) Quicksort

Abbildung B.4: Laufzeit von Sortieralgorithmen auf großen sortierten (normale Graphstärke), umgekehrten (fette Graphstärke) und zufällig geordneten (strichlierter Graph) Eingabemengen, gemessen mit quadratisch steigender Schrittweite.

Literatur

Aho, Alfred, Jeffrey Ullman und John Hopcroft (1974). *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc.

Aluffi, Paolo (2009). *Algebra: Chapter 0*. Graduate studies in mathematics. American Mathematical Society.

Bemer, Robert W. (Jan. 1958). „A Machine Method for Square-Root Computation“. In: *Commun. ACM* 1.1, S. 6–7.

Bollobás, Béla, Trevor I. Fenner und Alan M. Frieze (März 1996). „On the Best Case of Heapsort“. In: *J. Algorithms* 20.2, S. 205–217.

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

Chen, Yong-Gao und Jin-Hui Fang (Jan. 2007). „Triangular Numbers in Geometric Progression“. In: *Integers* 7.

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

Gericke, Helmuth (1984). *Mathematik in Antike und Orient*. Springer-Verlag Berlin Heidelberg.

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.

Johnson, David S. (2002). „A theoretician's guide to the experimental analysis of algorithms“. In: *Data Structures and Near Neighbor Searches and and Methodology: Fifth and Sixth DIMACS Implementation Challenges*. American Mathematical Society, S. 215–251.

Knill, Emmanuel (1996). *Conventions for quantum pseudocode*. Techn. Ber. Los Alamos National Lab., NM (United States).

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Third. USA: Addison-Wesley.

– (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Second. USA: Addison Wesley.

Luo, Ruibang, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu u. a. (2012). „SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler“. In: *Gigascience* 1.1, S. 2047–217X.

McGeoch, Catherine C. (2012). *A Guide to Experimental Algorithmics*. 1st. USA: Cambridge University Press.

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

Oda, Yusuke, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda und Satoshi Nakamura (2015). „Learning to generate pseudo-code from source code using statistical machine translation“. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, S. 574–584.

Pickover, Clifford A. (2009). *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling Milestones Series. Sterling.

Sedgewick, Robert und Phillipe Flajolet (2013). *Introduction to the Analysis of Algorithms*. 2nd. USA: Addison-Wesley Professional.

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

Wolff, Christian von (1747). *Vollständisches Mathematisches Lexicon*. Leipzig: Gleditsch.

Zobel, Justin (2015). *Writing for Computer Science*. 3rd. Springer Publishing Company, Incorporated.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings