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. Ein Sortieralgorithmus überführt die Elemente seiner Eingangsmenge zu einer geordneten Ausgangsmenge[?]. (Die konkrete Ordnungsrelation ist hierbei irrelevant[?].)

Die Effizienz eines solchen 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[?] beschränkt sich diese Arbeit auf Zeit als von einem Algorithmus verbrauchte Ressource, im Folgenden auch als „Laufzeit“ bezeichnet. Folgend bezieht sich die 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

Hier werden die zu behandelden Algorithmen mit ihrer theoretischen Effizienz dargelegt — bestenfalls mit Begründung der Auswahl. Ein erster Vergleich der Effizienz ist hier angebracht, ebenfalls könnten bereits Hypothesen bzgl. der Ergebnisse in Kapitel 3 aufgestellt werden.

In diesem Kapitel werden die in der folgenden Arbeit behandelten Sortieralgorithmen dargelegt. Die Beschreibung eines jeden Algorithmus umfasst eine beispielhafte Implementation und eine grobe Herleitung der theoretischen Effizienz.

1.1 *Insertion*

```
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  }
```

1.2 *Exchanging*

```
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    // TODO: These two calls to std::partition are
         suboptimal.
10   auto const middle1 = std::partition(first, last, [=](
         auto const &elem) {
```

```
11      return cmp(elem, pivot); });
12    auto const middle2 = std::partition(middle1, last, [=](
          auto const &elem) {
13      return !cmp(pivot, elem); });
14
15    quick(first, middle1, cmp);
16    quick(middle2, last, cmp);
17  }
```

1.3 *Selection*

```
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  }
```

1.4 *Merging*

```
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 }
```

Kapitel 2

Herangehensweise

Die theoretische Effizienz eines Algorithmus wird üblicherweise als Funktion in Abhängigkeit der Eingabegrösse beschrieben, wobei die jeweiligen Funktionswerte die theoretische Laufzeit darstellen[?]. Um die praktische Effizienz eines Algorithmus zu ermitteln, gilt es also 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 2.2 für eine beispielhafte Implementation der obigen Eingabemengearten.

```
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 2.2: Implementation von „Generatoren“ für diverse Arten von Eingabemengen.

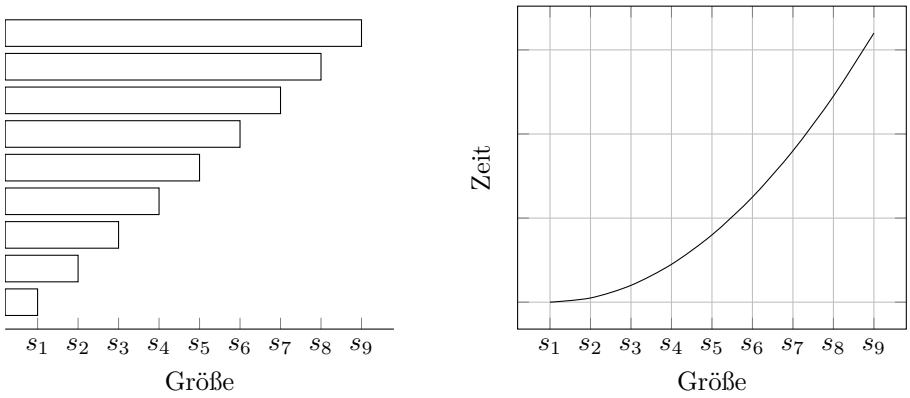
Eine ausführlichere Auswahl an Eingangsmengen welche an einen sie bearbeitenden Algorithmen angepasst ist — wie sie für eine eingehende Analyse eines einzelnen Algorithmus angebracht wäre — würde das allgemeiner gesetzte Ziel dieser Arbeit verfehlen.

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).

Diese Funktion der praktischen Effizienz kann mithile der Funktion benchmark ::run in Listing 2.3 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
```



(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.

```
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 2.3: Implementation einer Funktion zur Ermittlung der praktischen Effizienz eines Algorotihmus mit einer bestimmenten Eingabemenge.

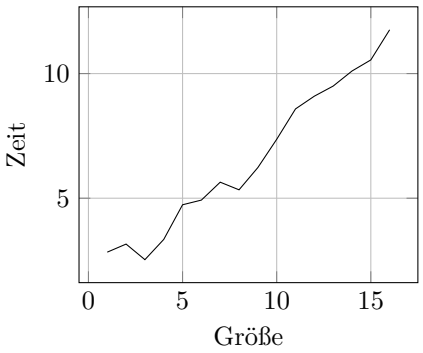
Der Parameter total_chunks definiert die Anzahl der Untermengen (am Beispiel von Abbildung 2.1 also 9). In Kombination mit der Größe der Eingabemenge set_size kann damit die konstante *Schrittgrösse* chunk_size ($s_1 - s_2$ beziehungsweise $s_n - s_{n+1}$ für $0 \leq n \leq 9$) ermittelt werden.

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 darstel-

len. 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 2.3 mit daraus generiertem Graphen.

Kapitel 3

Ergebnisse

Viele Graphen, unterteilung nach Art der Eingabe?

Konklusion

42.

Literatur

Bocchino, Robert L. u. a. (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.

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.

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

Abbildungsverzeichnis

Tabellenverzeichnis

Listings