SWE3 - Übung 2

WS 2025/26

Aufwand in h: 12

Tim Peko

Inhaltsverzeichnis

1. Aufgabe: Merge Sort

1.1. Lösungsidee

1.1.1. Merge Sort

Der Merge Sort Algorithmus funktioniert, indem wir immer sortierte Subarrays zu einem sortierten Superarray zusammenfügen. Dazu machen wir uns die sortierte Eigenschaft zu Nutze und fügen immer das kleinste Element des linken und rechten Subarrays zu dem Superarray hinzu. Dieser Vorgang wird in Abbildung 1 demonstriert. Wichtig ist hierbei, dass ein zusätzlicher Buffer benötigt wird, der das gemerged Superarray speichert.

Um die gesamte Collection zu sortieren, brechen wir die Collection auf die kleinste möglichen Subarrays, die bereits sortiert sind, auf. Das sind die Subarrays, die nur ein Element enthalten. Danach wird der Merge Schritt für die immer größer werdenden (gemerged) Super-/Subarrays wiederholt, bis die gesamte Collection sortiert ist. Das ist in Abbildung 2 visualisiert.

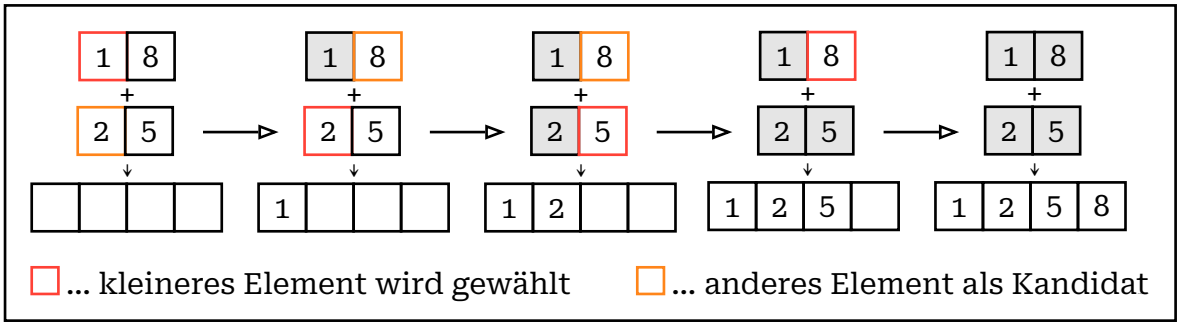


Abbildung 1: Visualisierung eines Merge Schrittes

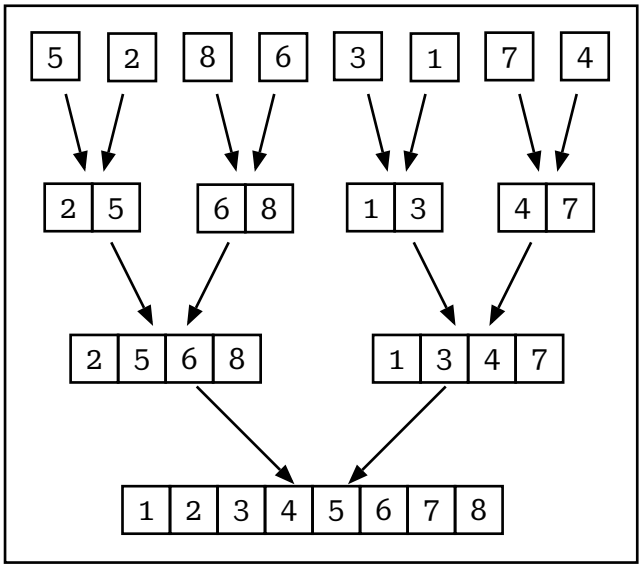


Abbildung 2: Visualisierung des Merge Sort Algorithmus

1.1.1.1. Eigenschaften

Der Merge Sort Algorithmus ist

- ein stabiler Sortieralgorithmus, da er die relative Reihenfolge der Elemente mit gleichem Wert beibehält.

- kein in-place Sortieralgorithmus, da er zusätzlichen Speicher benötigt.
- ein vergleichsbasierter Sortieralgorithmus, da er nur auf Vergleichsoperatoren basiert.
- gut in Dateien aufteilbar, da er die Daten in kleinere Chunks aufteilt, die separat sortiert werden können.
- mit einer Zeitkomplexität von $O(n \log n)$ im besten, durchschnittlichen und schlechtesten Fall.
- mit einer Platzkomplexität von $O(n)$ für den zusätzlichen Speicher.

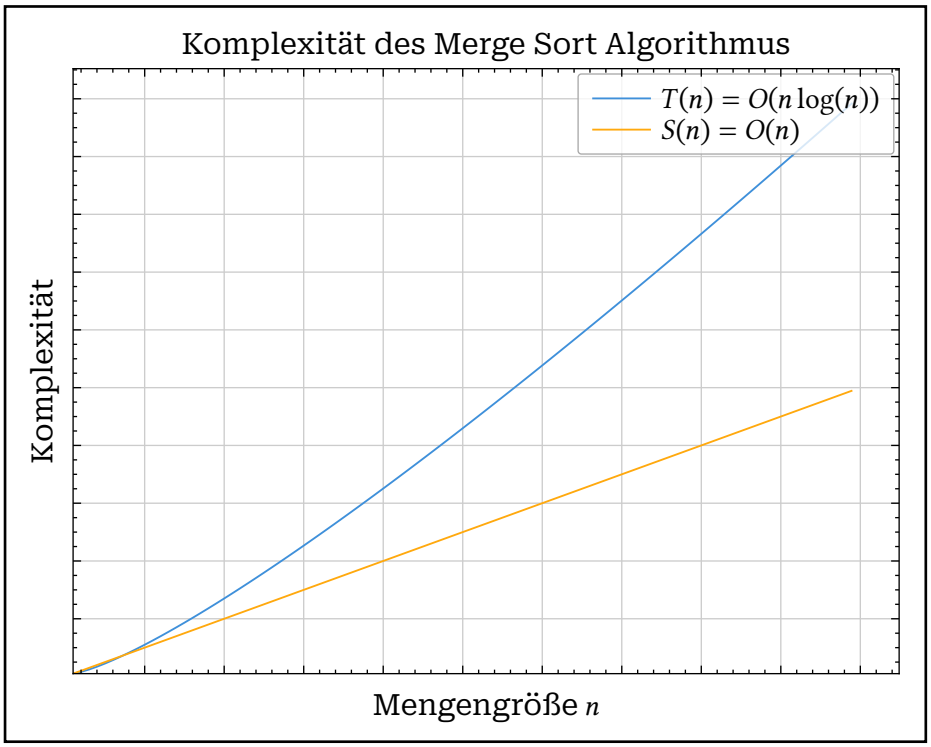


Abbildung 3: Komplexität des Merge Sort Algorithmus

1.1.2. Implementierung In-Memory

Es existiert eine `main.cpp` Datei, die allerdings nur eine Dummy Main Funktion enthält. Sie dient lediglich dazu, das Projekt kompilieren zu können. Alternativ könnte in Visual Studio auch der Projekttyp auf `Static Library` gesetzt werden. Das wird hier zur leichteren Kompatibilität nicht gemacht.

1.1.2.1. Buffer Interfaces

Da in Abschnitt 2 auf die On-Disk Sortierung eingegangen wird, habe ich einen Merge Sort implementiert, der es uns erlaubt, einfach per Interface die Buffer Typen (In-Memory oder On-Disk) zu wechseln:

- IMergeReader: liefert das aktuelle Element (`get()`), rückt vor (`advance()`), meldet Ende (`is_exhausted()`), kann in einen Writer umgewandelt werden (`into_writer()`).
- IMergeWriter: hängt Elemente an (`append(value)`), kann in einen Reader umgewandelt werden (`into_reader()`).

Diese Trennung macht die Kernlogik unabhängig vom Speicherort der Daten und ermöglicht identische Sortierlogik für In-Memory und On-Disk.

1.1.2.2. Kernlogik

Der Kern der Merge Sort Implementierung ist die `merge_step` Methode, die zwei Subarrays merged und in einen Zielbuffer schreibt. Das ist in Abbildung 1 visualisiert. Chunksize ist dabei die Größe der bereits gemergeden und damit sortierten Subarrays.

- Vergleiche jeweils das aktuelle Element beider Leser.
- Schreibe das kleinere in den Writer und rücke am entsprechenden Leser vor.
- Beende den Chunk, sobald je Seite `chunk_size` Elemente übertragen wurden oder eine Seite erschöpft ist.
- Schreibe verbleibende Elemente der nicht erschöpften Seite für diesen Chunk.

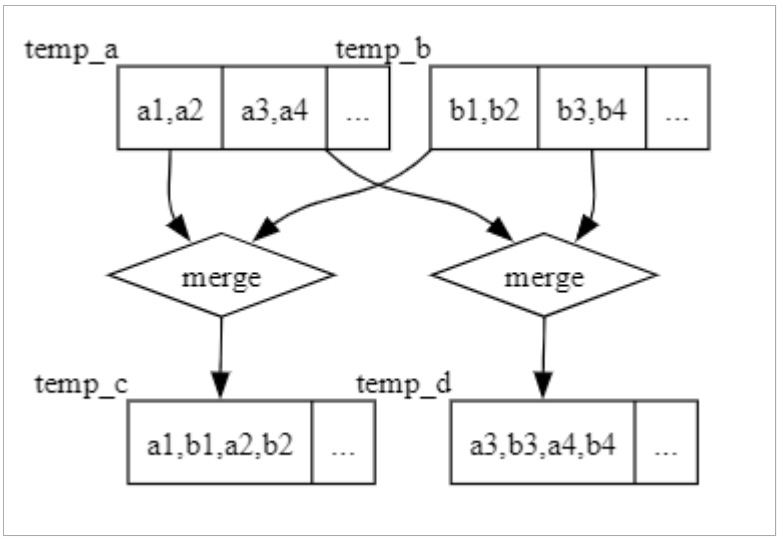
In Vorbereitung auf On-Disk Sortierung, alterniert die `merge` Methode für jeden Teilchunk zwischen zwei Zielbuffern. Abbildung 4 visualisiert diesen Vorgang. Das alternierende Schreiben erlaubt es, im nächsten Durchlauf ohne zusätzliche Kopien wieder einzulesen.

```cpp
void merge_sorter::merge(IMergeReader<T> &sorted_l, IMergeReader<T> &sorted_r,
IMergeWriter<T> &writer_l, IMergeWriter<T> &writer_r, size_t chunk_size)
{
    bool write_to_left = true; // Start with writer_l for first chunk

    while (true)
    {
        IMergeWriter<T>& writer = write_to_left ? writer_l : writer_r;

        if (!merge_step(sorted_l, sorted_r, writer, chunk_size)) {
            break;
        }

        // Switch to the other writer for the next chunk
        write_to_left = !write_to_left;
    }
}
```



— Übungsangabe

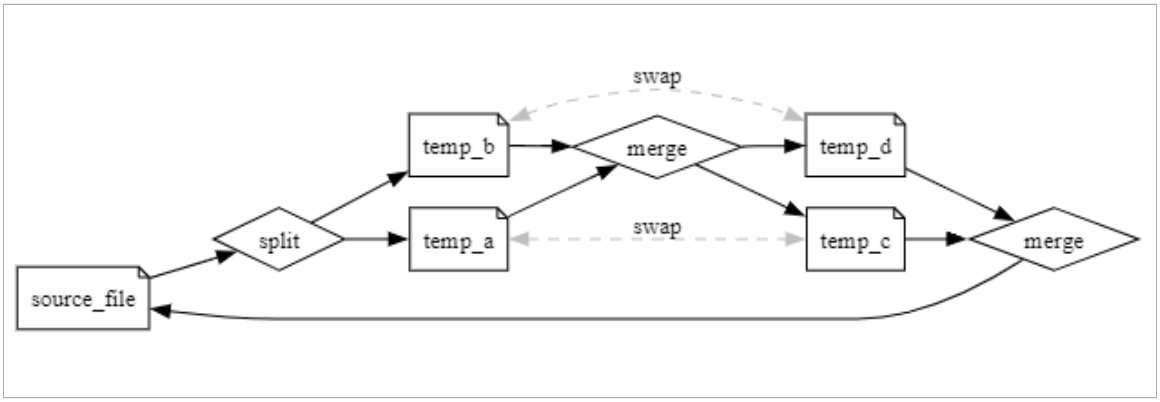
Abbildung 4: Jeweils Chunks der Größe n von A und B, werden zu einem Chunk der Größe $2n$ zusammengeführt. Die neuen Chunks werden abgechselnd in C und D geschrieben.

`sort` führt wiederholt Merges mit verdoppelnder Chunkgröße aus und tauscht nach jedem Durchlauf die Rollen von Readern/Writern (Re-Seating). Abgebrochen wird, wenn die Writer die jeweils vollständigen sortierten Hälften enthalten.

Zusätzlich gibt es noch die Hilfsmethode `split`, die alternierend die Elemente der unsortierten Collection in zwei Hälften schreibt.

Die `complete_sort` Methode
1. Splittet die unsortierte Collection in zwei Hälften
2. Führt den finalen Merge Schritt aus
3. Setzt den Quellbuffer auf den neuen sortierten Buffer

Visualisierung in Abbildung 5.



— Übungsangabe

Abbildung 5: Flowchart der `complete_sort` Methode: Aufteilung und wiederholtes zusammenführen von immer größeren Chunks.

Hinweis: Der finale Zusammenführungsschritt schreibt die sortierten Hälften zurück in die Quelle und setzt deren Cursor auf den Anfang zurück (Re-Seating).

1.1.2.3. `stream_reader` Fixes

Die per Moodle offiziell bereitgestellte `stream_reader` Klasse zeigt unerwartetes Verhalten bei der Verwendung: Die interne Methode `stream_reader::next()` gibt am Ende des Filestreams einen leeren String `""` zurück. Das liegt an der Implementierung der `stream_reader::has_next()` Methode:

```cpp
template<typename T>
inline bool stream_reader<T>::has_next() {
  return buffer.has_value() || (m_in && m_in.good() && !m_in.eof());
}
```

Diese spiegelt nämlich die Implementierung des `std::istream::operator >>` wider und daher wird bei `stream_reader::next()` versucht, ein neues `T` auszugeben, was am Ende des Streams fehlschlägt und einen leer initialisierten String zurückgibt.

Die Lösung wird durch einen Buffer implementiert:

```cpp
template<typename T>
inline bool stream_reader<T>::has_next() {
    if (buffer) return true;
    T tmp;
    if (m_in >> tmp) {           // only true when a token was actually read
        buffer = std::move(tmp); // stash it for next()
        return true;
    }
    return false;
}

template<typename T>
inline std::optional<T> stream_reader<T>::next() {
    if (buffer) {
        auto v = std::move(*buffer);
        buffer.reset();
        return v;
    }
    T tmp;
    if (m_in >> tmp) return tmp; // succeed → return token
    return std::nullopt;         // fail → no token
}
```

1.1.2.4. File Handling

Dateien werden ausschließlich über `>>`/`<<` verarbeitet (Anforderung). Zum Einlesen wird `stream_reader<T>` benutzt, der tokenweise liest; zum Schreiben werden die Tokens mit Trennzeichen in einen `std::ofstream` ausgegeben. Fehler beim Öffnen werden abgefangen und führen zu aussagekräftigen Exceptions.

1.1.2.5. In-Memory Buffers

Die `InMemoryReader` und `InMemoryWriter` Klassen werden verwendet, um In-Memory Datenfolgen zu lesen und zu schreiben. Sie verwenden einen `std::vector<T>` als Datenquelle und schreiben in diesen.

Die Reader haben einen Cursor, der auf das aktuelle Element zeigt und bei jedem Aufruf von `advance()` um eins erhöht wird. Bei einer Konvertierung zu einem

Writer, wird der backing Vector geleert und per `shared_ptr` an den entstehenden Writer übergeben.

Wird der Writer zurück in einen Reader konvertiert, wird der Cursor auf 0 gesetzt und der backing Vector ebenfalls per `shared_ptr` an den entstehenden Reader übergeben.

1.1.2.6. Zusammenfügen

Die komplette In-Memory Sortierung wird in der `sort_vec_in_memory` Methode zusammengeführt. Die `sort_file_in_memory` Methode liest mit den in Abschnitt 1.1.2.4 gezeigten Methoden die Datei in einen `std::vector<std::string>` und führt dann die Sortierung mit `sort_vec_in_memory` durch, bevor sie die sortierte Datenfolge wieder in die Datei schreibt.

Die Implementierung konvertiert die Eingabe in einen Reader, orchestriert `complete_sort` mit vier Puffern und schreibt das Ergebnis zurück in die Quelle. Dadurch bleibt die Kernlogik identisch zu On-Disk.

1.2. Testfälle

Der Standard-Testfall

1. **Arrange** - Legt eine Datei mit zufälligen Strings an

```cpp
std::string filename = "test_file.txt";
file_manipulator::fill_randomly(filename, array_length, string_length);
```

2. **Act** - Sortiert diese per `merge_sorter::sort_file_in_memory(...)`

```cpp
merge_sorter sorter;
sorter.sort_file_in_memory(filename);
```

3. **Assert** - Verifiziert das Ergebnis hinsichtlich der Sortierung.

```cpp
std::ifstream file(filename);
stream_reader<std::string> reader(file);

std::string prev = reader.get();
while (reader.has_next()) {
    std::string current = reader.get();
    ASSERT_LE(prev, current) << "Elements are not in sorted order";
    prev = current;
}
```

1.2.1. Liste der abgedeckten Testfälle

Die folgenden Testfälle überprüfen die `merge_sorter` Implementierung:

- Default Testfall - *Parametrisiert*
  - String Länge 2, 10, 20
  - Array Länge 10, 200, 5000, 100000
- Leere Datei
- Datei mit nicht alphanumerischen Zeichen
- Verkehrt sortierte Datei
- Datei mit Duplikaten
- Datei mit unterschiedlich langen Strings

Zusätzlich gibt es kleine Testfälle, die sowohl die `random.h` als auch die `stream_reader.h` Implementierung testen.

1.3. Ergebnisse

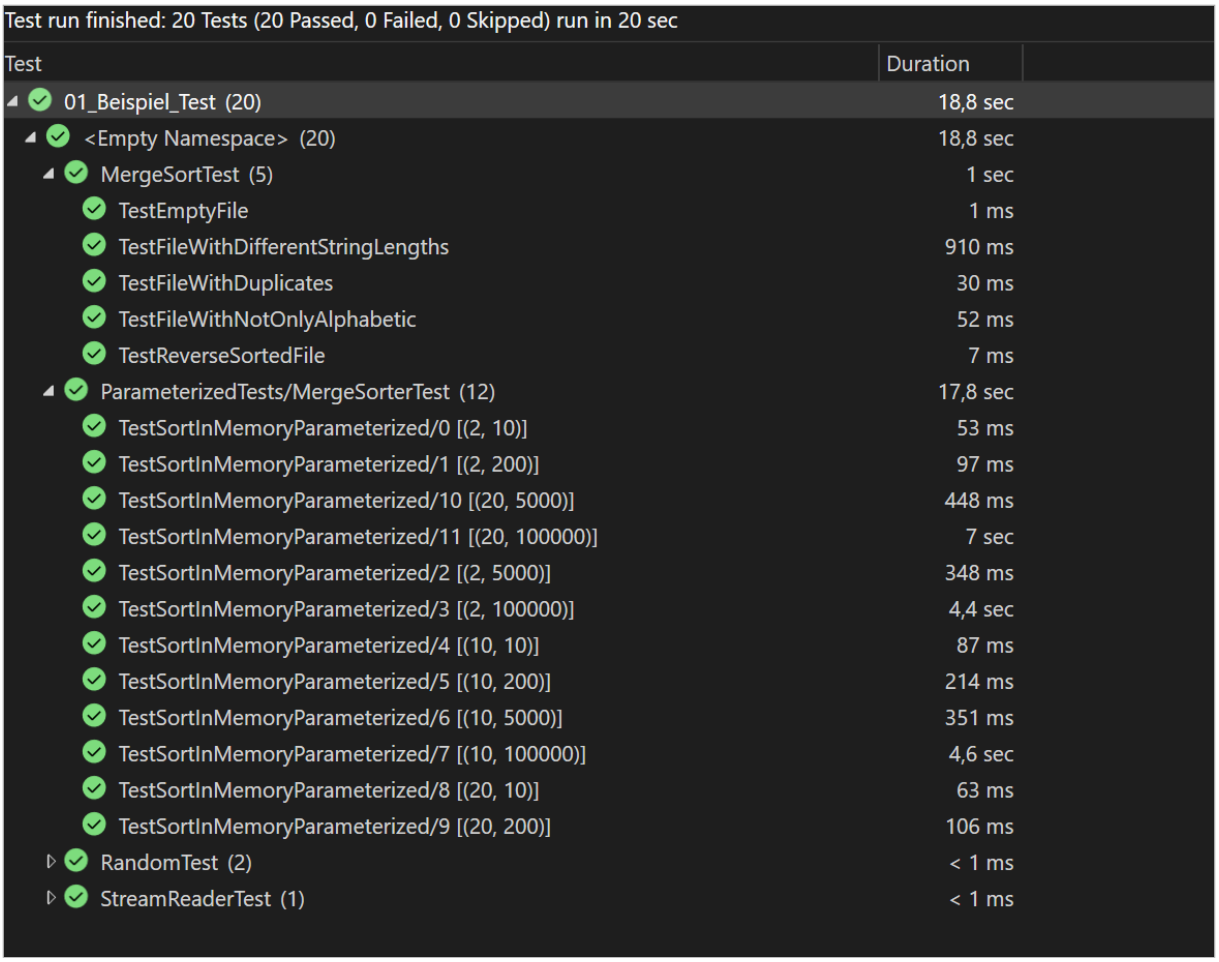


Abbildung 6: Ergebnisse der Testfälle für Beispiel 1

Alle Testfälle bestehen erfolgreich, wie in Abbildung 6 zu sehen ist. Es fällt auf, dass die Laufzeit bei den parametrisierten Testfällen stark unterschiedlich ausfällt. Das liegt an der stark unterschiedlichen Größe der zu sortierenden Datei.

2. Aufgabe: On Disk

2.1. Anforderungen

Wir müssen folgende Anforderungen aus der Angabe erfüllen:

> Ein paar Implementierungshinweise:
> 1. Die Klasse merge_sorter soll die in der Übung besprochene Klasse file_manipulator für alle Dateioperationen verwenden. Diese Dateioperationen könnten sein: eine Datei kopieren, eine Datei mit Zufallswerten füllen, eine Datei in mehrere Dateien aufsplitten, den Inhalt einer Datei ausgeben.
> 2. Die Klasse file_manipulator operiert auf ifstreams und ofstreams. Die einzigen erlaubten Operationen auf diese Streams sind nur << und >>.
> 3. Die Klasse `stream_reader<value_type>` **...**
>
> — Übungsangabe

2.2. Lösungsidee

Prinzipiell müssen nur noch die `IMergeReader` und `IMergeWriter` Interfaces für On-Disk implementiert werden. Um auch die Anforderungen zu erfüllen, nutzen wir dazu eine `file_manipulator` Klasse, die folgende statische Methoden bereitstellt:

1. `fill_randomly(std::string const& file_name, size_type n = 100, size_type len = 4)`: Füllt eine Datei mit Zufallswerten.
2. `append(std::ofstream& file, std::string const& str)`: Fügt eine Zeichenkette an die Datei an.
3. `print(std::string const& src_file_name, std::ostream& out = std::cout)`: Gibt den Inhalt der Datei auf die Konsole aus.
4. `delete_file(std::string const& file_name)`: Löscht eine Datei.

2.2.1. Kleine Optimierung

In der `merge_sorter::sort(...)` Methode können wir die Iteration früher abbrechen, als wir in Abschnitt 1 implementiert haben. Dazu verwenden wir die `chunk_size > total_size / 2` Bedingung statt `chunk_size ⩾ total_size`. Das liegt daran, dass wir nur den Inhalt der beiden Writer, die jeweils die Hälfte der Daten entsprechen, sortieren müssen. Der letzte Merge Schritt wird dann über den finalen `merge_sorter::merge(...)` Aufruf in `merge_sorter::complete_sort(...)` durchgeführt.

2.2.2. Validierung und Fehlerbehandlung

- Öffnen einer Datei wird geprüft; bei Fehlern werden aussagekräftige Exceptions geworfen (z. B. in `FileMergeReader/-Writer`, `merge_sorter::sort_file_in_memory`).
- Lesen/Writing geschieht nur über <</>> (Anforderung 2); Formatfehler führen zu `has_next()==false` und werden nicht stillschweigend als leere Tokens interpretiert.

2.2.3. On-Disk Buffers

Die On-Disk Implementierung erfolgt über zwei Klassen in `file_merge_buffer.cpp`:

- `FileMergeReader<T>`: Implementiert `IMergeReader<T>` für das Lesen aus Dateien. Nutzt intern `stream_reader<T>` und `std::ifstream`. Beim Öffnen wird geprüft, ob die Datei existiert - andernfalls wird eine `std::runtime_error` Exception geworfen.
- `FileMergeWriter<T>`: Implementiert `IMergeWriter<T>` für das Schreiben in Dateien. Nutzt `std::ofstream` und die `file_manipulator::append()` Methode. Beim Erstellen wird die Zieldatei geleert, um saubere Ausgangsbedingungen zu schaffen.

Beide Klassen unterstützen die `into_reader()/into_writer()` Konvertierung, die für das Wechseln zwischen Lese- und Schreibmodus auf derselben Datei erforderlich ist. Dies ermöglicht es, dass die Buffer-Dateien sowohl als Eingabe als auch als Ausgabe in verschiedenen Phasen des Merge-Sort-Algorithmus verwendet werden können.

Wesentliche Verantwortung:
- FileMergeReader: Öffnet Datei, liest tokenweise über `stream_reader<T>`, validiert Verfügbarkeit (`has_next()`), liefert `peek()/get()`-Semantik.
- FileMergeWriter: Trunkiert/erstellt Zieldatei, hängt Tokens über `file_manipulator::append()` an.

2.3. Tests

Wir setzen auch in diesem Beispiel auf den etablierten Standard-Testfall aus Abschnitt 1.2. Dabei werden die Testfälle für On-Disk Sortierung hinzugefügt. Wir erzeugen beim Testfall $\text{string\_length} = 100$ und $\text{array\_length} = 1000000$ eine Datei mit knapp unter $100\ \text{MB}$ Daten. Mit dem zusätzlichen Speichervorbauch der vier Buffer sind das insgesamt $\sim 300\ \text{MB}$ Speicher.

Zusätzlich testen wir die Fehlerbehandlung für nicht vorhandene Dateien (Erwartung: `std::runtime_error`).

2.4. Ergebnisse

2.4.1. Testergebnisse

Die Tests verifizieren die korrekte Sortierung und robuste Fehlerbehandlung. Auch hier zeigt Abbildung 7, dass die Laufzeit entsprechend mit der Größe der zu sortierenden Datenfolge wächst. Der Testlauf, der die $100\ \text{MB}$ Datei sortiert, dauert ~10 Minuten. Die Struktur mit `IMergeReader/IMergeWriter` ermöglicht identische Kernlogik für In-Memory und On-Disk, was die Wartbarkeit verbessert.

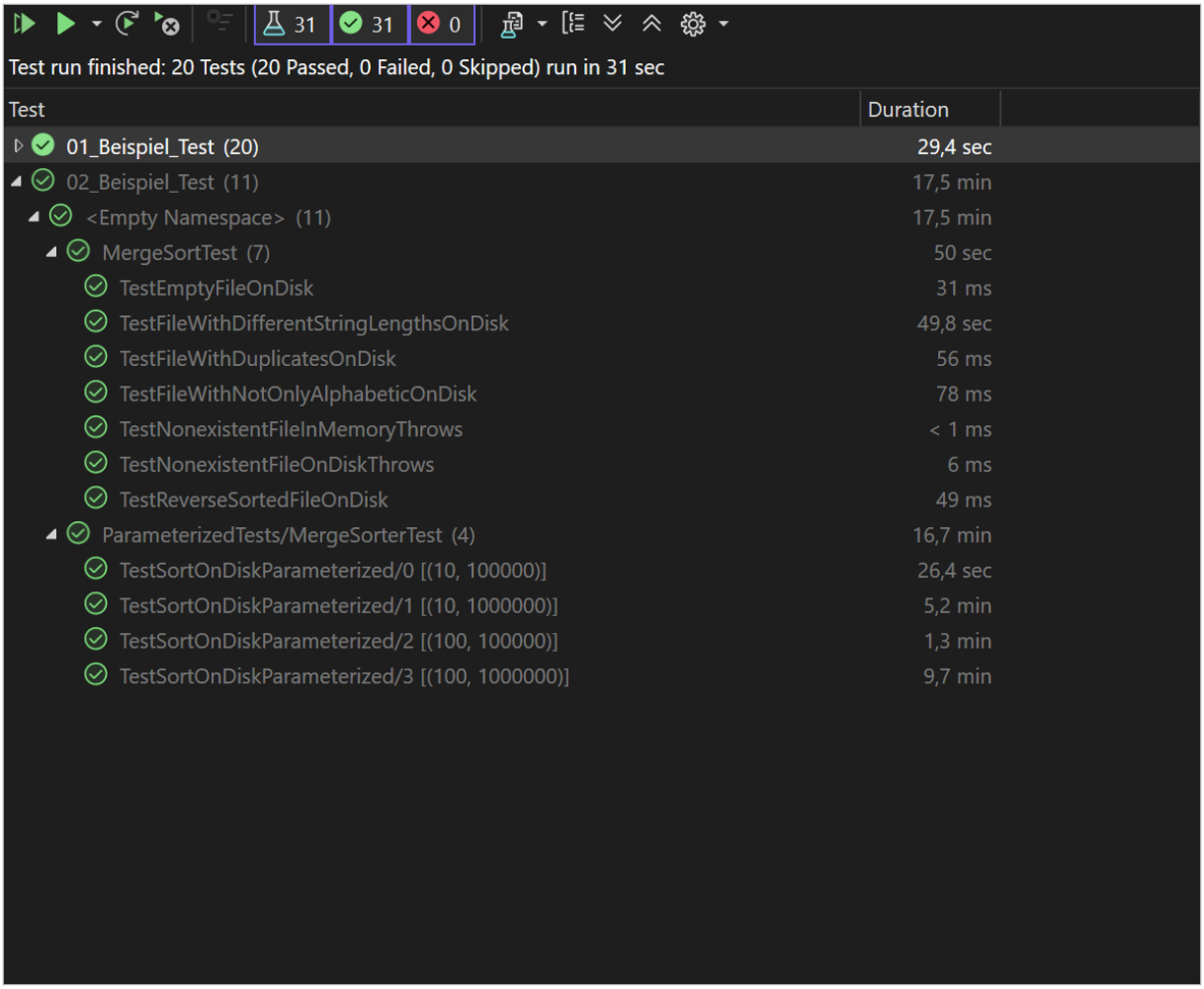


Abbildung 7: Ergebnisse der Testfälle für Beispiel 2

2.4.2. Benchmark

Final wurde ein gegenüberstellender Benchmark durchgeführt, der die Laufzeit $T(n)$ und den Speicherbedarf $S(n)$ des Merge Sort Algorithmus für In-Memory und On-Disk vergleicht. Die Implementierung dazu wurde in Rust geschrieben, da ich hierfür mit dem Benchmarking Ecosystem vertrauter bin. Die Ergebnisse sind in Abbildung 8 zu sehen. Sie zeigen, dass die Laufzeit für On-Disk bei langen Zeichenketten deutlich höher ist, da der Algorithmus auf der Festplatte arbeiten muss. Der Prozessspeicherbedarf bleibt bei On-Disk konstant, bei In-Memory nimmt er mit der Größe der Datenfolge zu.

Der Benchmark wurde auf folgender Hardware durchgeführt:

CPU	AMD Ryzen 7 2700X Eight-Core Processor @ 4.07 GHz
Logical Cores	16
RAM	31.25 GB
Arch	x86_64
Kernel	6.17.0-2-default
OS	Linux (openSUSE Tumbleweed 20251007)

Tabelle 1: Hardware Spezifikationen des Benchmark Systems

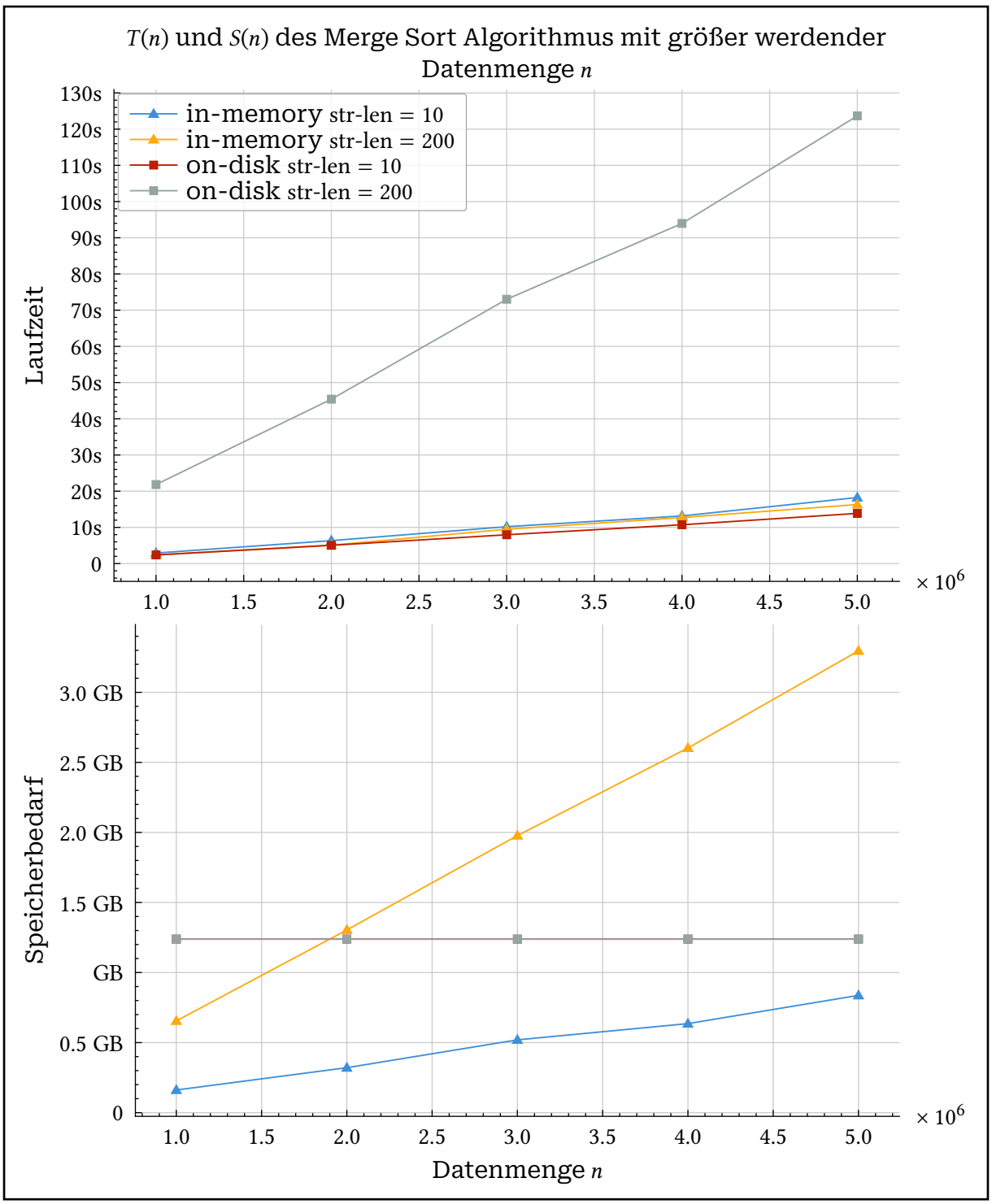


Abbildung 8: Benchmark Ergebnisse des Merge Sort Algorithmus auf einer AMD Ryzen 7 2700X Eight-Core Processor CPU