

SWE3 - Übung 2

WS 2025/26

Aufwand in h: 12

Tim Peko

Inhaltsverzeichnis

1. Aufgabe: Merge Sort	2
1.1. Lösungsidee	2
1.1.1. Merge Sort	2
1.1.1.1. Eigenschaften	2
1.1.2. Implementierung In-Memory	3
1.1.2.1. Buffer Interfaces	3
1.1.2.2. Kernlogik	4
1.1.2.3. stream_reader Fixes	7
1.1.2.4. File Handling	8
1.1.2.5. In-Memory Buffers	8
1.1.2.6. Zusammenfügen	9
1.2. Testfälle	10
1.2.1. Liste der abgedeckten Testfälle	10
1.3. Ergebnisse	11
2. Aufgabe: On Disk	12
2.1. Anforderungen	12
2.2. Lösungsidee	12
2.2.1. Kleine Optimierung	12
2.2.2. Validierung und Fehlerbehandlung	12
2.2.3. On-Disk Buffers	13
2.3. Tests	14
2.4. Ergebnisse	14
2.4.1. Testergebnisse	14
2.4.2. Benchmark	15

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.

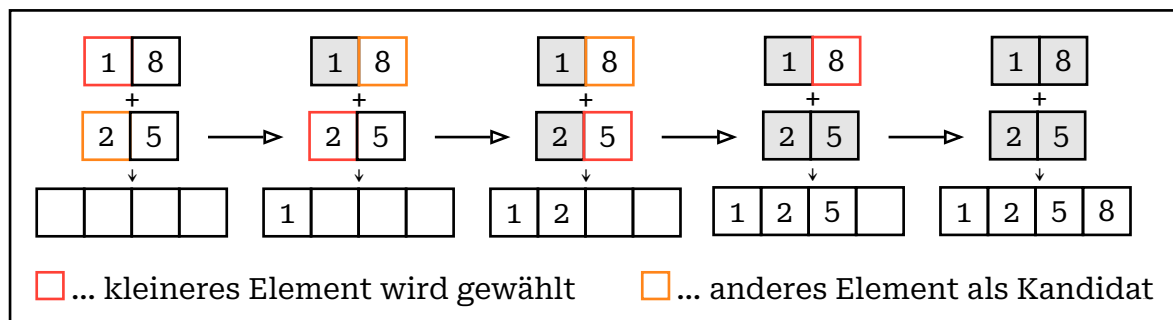


Abbildung 1: Visualisierung eines Merge Schrittes

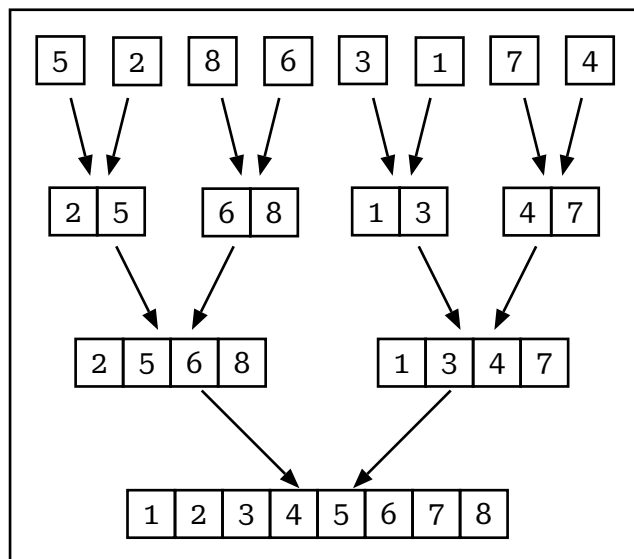


Abbildung 2: Visualisierung des Merge Sort Algorithmus

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.

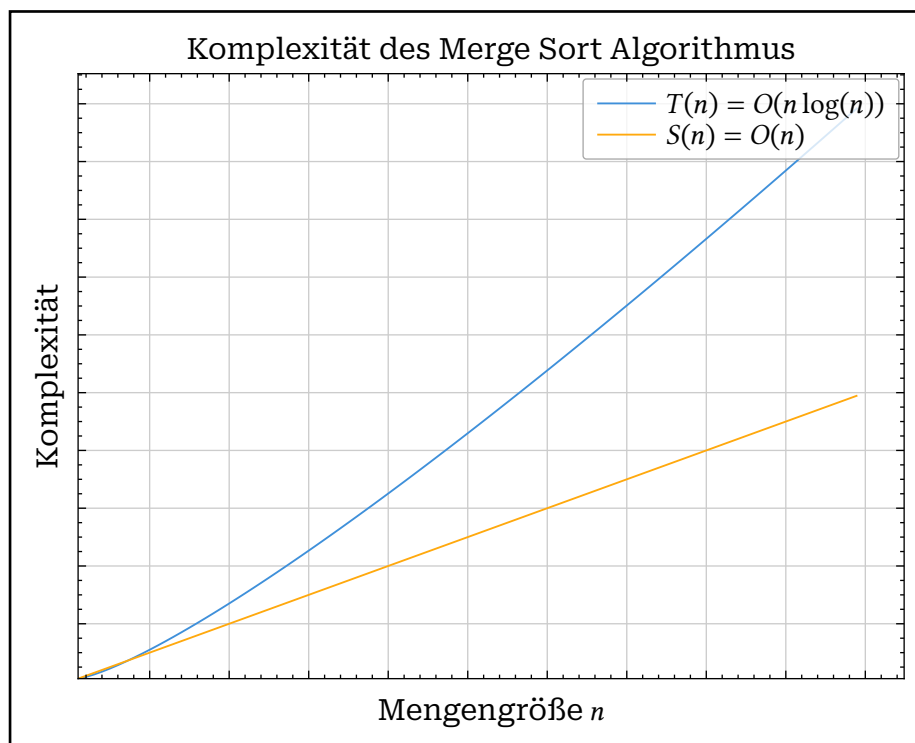


Abbildung 3: Komplexität des Merge Sort Algorithmus

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:

```
1 // merge_sort.hpp
2
3 /// Generic interface for reading elements in merge operations.
4 template<typename T>
5 class IMergeReader {
6 public:
7     virtual ~IMergeReader() = default;
8
9     /// Returns the current value at the reader's position.
10    virtual T get() = 0;
```

cpp

```

11
12     /// Advances the reader to the next value.
13     virtual bool advance() = 0;
14
15     /// Checks if the reader is advanced to exhaustion.
16     virtual bool is_exhausted() = 0;
17
18     /// Converts this reader into an IMergeWriter, allowing the buffer to be reused
    for writing.
19     virtual std::unique_ptr<IMergeWriter<T>> into_writer() = 0;
20 };
21
22     /// Generic interface for writing (appending) elements in merge operations.
23     template<typename T>
24     class IMergeWriter {
25     public:
26         virtual ~IMergeWriter() = default;
27
28         /// Appends a value at the writer's position.
29         virtual bool append(const T& value) = 0;
30
31         /// Converts this writer into an IMergeReader, allowing the written data to be
    read.
32         virtual std::unique_ptr<IMergeReader<T>> into_reader() = 0;
33     };

```

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 gemerged und damit sortierten Subarrays.

```

1  template <typename T>
2  bool merge_sorter::merge_step(IMergeReader<T> &reader_l, IMergeReader<T> &reader_r,
3  IMergeWriter<T> &writer, size_t chunk_size_per_reader)
4  {
5      size_t l_merged_count = 0; // Increment until chunk_size is reached
6      size_t r_merged_count = 0; // Increment until chunk_size is reached
7      bool l_exhausted = reader_l.is_exhausted();
8      bool r_exhausted = reader_r.is_exhausted();
9
10     // Zip and merge one chunk of size chunk_size_per_reader
11     while (l_merged_count < chunk_size_per_reader && r_merged_count <
12     chunk_size_per_reader && !l_exhausted && !r_exhausted)
13     {
14         T left_val = reader_l.get();
15         T right_val = reader_r.get();
16
17         if (left_val <= right_val)
18         {
19             writer.append(left_val);
20             l_exhausted = !reader_l.advance();
21             l_merged_count++;
22         }
23         else
24         {
25             writer.append(right_val);
26             r_exhausted = !reader_r.advance();
27             r_merged_count++;
28         }
29     }
30
31     // Finish remaining elements from left reader for this chunk
32     while (l_merged_count < chunk_size_per_reader && !l_exhausted)

```

```

31 {
32     writer.append(reader_l.get());
33     l_exhausted = !reader_l.advance();
34     l_merged_count++;
35 }
36
37 // Finish remaining elements from right reader for this chunk
38 ... // Same but for other side
39
40 return !l_exhausted || !r_exhausted;
41 }

```

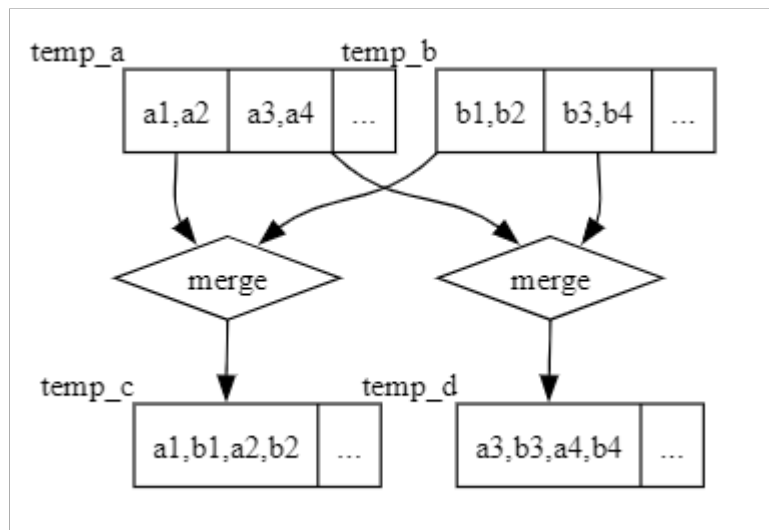
In Vorbereitung auf On-Disk Sortierung, alterniert die `merge` Methode für jeden Teilchunk zwischen zwei Zielbuffern. Abbildung 4 visualisiert diesen Vorgang.

```

1  template <typename T>
2  void merge_sorter::merge(IMergeReader<T> &sorted_l, IMergeReader<T> &sorted_r,
3  IMergeWriter<T> &writer_l, IMergeWriter<T> &writer_r, size_t chunk_size)
4  {
5      bool write_to_left = true; // Start with writer_l for first chunk
6      while (true)
7      {
8          IMergeWriter<T>& writer = write_to_left ? writer_l : writer_r;
9
10         if (!merge_step(sorted_l, sorted_r, writer, chunk_size)) {
11             break;
12         }
13
14         // Switch to the other writer for the next chunk
15         write_to_left = !write_to_left;
16     }
17 }

```

cpp



— Übungsangabe

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

Diese Schritte werden in der sort Methode so oft wiederholt und zwischen Quell- und Zielbuffern alterniert, bis die gesamte Collection sortiert ist. Dabei ist hier der finale merge Schritt noch ausständig.

```

1  template <typename T>
2  void merge_sorter::sort(
3      std::unique_ptr<IMergeReader<T>> &reader_l, ... &reader_r,
4      std::unique_ptr<IMergeWriter<T>> &writer_l, ... &writer_r,
5      size_t total_size)
6  {
7      size_t chunk_size = 1;
8      while (true)
9      {
10         merge(*reader_l, *reader_r, *writer_l, *writer_r, chunk_size);
11         chunk_size *= 2;
12         if (chunk_size >= total_size)
13         {
14             // We have finished the last iteration and writers contain sorted data
15             break;
16         }
17
18         // Prepare next chunk sized merge run
19         // Swap reader and writer buffer roles
20         ...
21     }
22 }

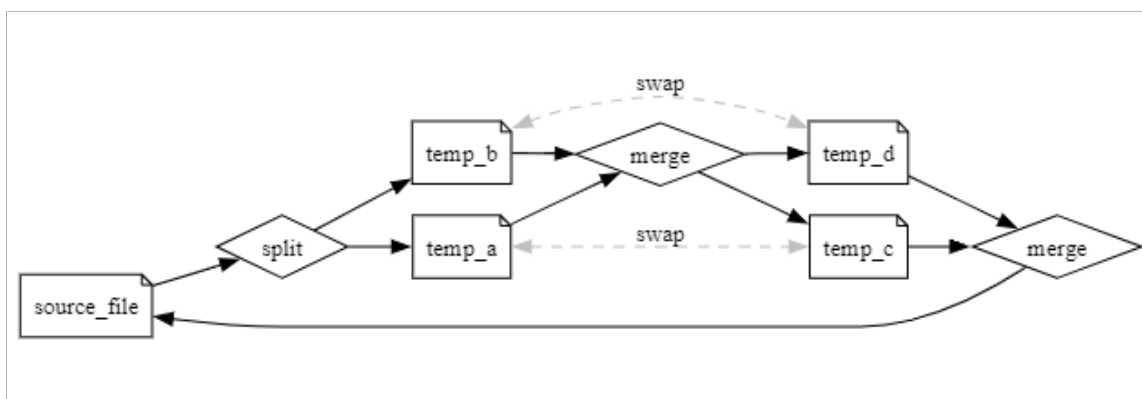
```

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.

```

1  template <typename T>
2  void merge_sorter::complete_sort(
3      std::unique_ptr<IMergeReader<T>> &unsorted_source,
4      std::unique_ptr<IMergeWriter<T>> buffer1, ... buffer4)

```

```

5 {
6     // Split the unsorted source into two halves
7     auto total_size = split(*unsorted_source, *buffer1, *buffer2);
8
9     // Sort the two halves
10    auto reader_l = buffer1->into_reader();
11    auto reader_r = buffer2->into_reader();
12    sort<T>(reader_l, reader_r, buffer3, buffer4, total_size);
13
14    // Merge the two sorted halves into the source
15    auto sorted_l = buffer3->into_reader();
16    auto sorted_r = buffer4->into_reader();
17    auto sorted_full = unsorted_source->into_writer();
18    merge_step<T>(*sorted_l, *sorted_r, *sorted_full, total_size);
19
20    // re-seat the source => reset the cursor to 0
21    unsorted_source = sorted_full->into_reader();
22 }

```

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:

```

1 template<typename T>
2 inline bool stream_reader<T>::has_next() {
3     return buffer.has_value() || (m_in && m_in.good() && !m_in.eof());
4 }

```

cpp

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:

```

1 template<typename T>
2 inline bool stream_reader<T>::has_next() {
3     if (buffer) return true;
4     T tmp;
5     if (m_in >> tmp) {           // only true when a token was actually read
6         buffer = std::move(tmp); // stash it for next()
7         return true;
8     }
9     return false;
10 }
11
12 template<typename T>
13 inline std::optional<T> stream_reader<T>::next() {
14     if (buffer) {
15         auto v = std::move(*buffer);
16         buffer.reset();
17         return v;
18     }
19     T tmp;
20     if (m_in >> tmp) return tmp; // succeed → return token
21     return std::nullopt;        // fail → no token
22 }

```

cpp

1.1.2.4. File Handling

Dateien werden mittels der `stream_reader` Klasse gelesen. Folgender Code wird dazu im Projekt verwendet:

```
1 std::ifstream read_file(file_name);
2 stream_reader<std::string> reader(read_file);
3 std::vector<std::string> data;
4 while (reader.has_next())
5 {
6     data.push_back(reader.get());
7 }
8 read_file.close();
```

cpp

Dateien werden folgendermaßen mit neuer Datenfolge beschrieben:

```
1 std::ofstream write_file(file_name);
2 for (const auto &entry : data)
3 {
4     write_file << entry << " ";
5 }
6 write_file.close();
```

cpp

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 template<typename T>
2 class InMemoryReader : public IMergeReader<T> {
3 private:
4     std::shared_ptr<std::vector<T>> _data;
5     size_t _cursor;
6
7 public:
8     explicit InMemoryReader(std::shared_ptr<std::vector<T>> data, size_t cursor = 0)
9         : _data(data), _cursor(cursor) {}
10
11     T get() override {
12         if (is_exhausted()) {
13             throw std::underflow_error("No more elements to read");
14         }
15         return (*_data)[_cursor];
16     }
17
18     bool advance() override {
19         if (is_exhausted()) {
20             return false;
21         }
22         _cursor++;
```

cpp


```

23     // Return true if we can further advance once more
24     return _cursor < _data->size();
25 }
26
27 bool is_exhausted() override {
28     return _cursor ≥ _data->size();
29 }
30
31 std::unique_ptr<IMergeWriter<T>> into_writer() override {
32     _data->clear();
33     return std::make_unique<InMemoryWriter<T>>(_data);
34 }
35 };
36
37 template<typename T>
38 class InMemoryWriter : public IMergeWriter<T> {
39 private:
40     std::shared_ptr<std::vector<T>> _data;
41
42 public:
43     explicit InMemoryWriter(std::shared_ptr<std::vector<T>> data = nullptr)
44         : _data(data ? data : std::make_shared<std::vector<T>>()) {}
45
46     bool append(const T& value) override {
47         _data->push_back(value);
48         return true;
49     }
50
51     std::unique_ptr<IMergeReader<T>> into_reader() override {
52         return std::make_unique<InMemoryReader<T>>(_data, 0);
53     }
54 };

```

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.

```

1  void merge_sorter::sort_vec_in_memory(std::vector<value_t> &data)
2  {
3      std::unique_ptr<IMergeReader<value_t>> input_reader(
4          std::make_unique<InMemoryReader<value_t>>(
5              std::make_shared<std::vector<value_t>>(data)
6          )
7      );
8
9      complete_sort<value_t>(
10         input_reader,
11         std::make_unique<InMemoryWriter<value_t>>(),
12         ...);
13
14     // Write the data from input_reader back to data vector
15     data.clear();
16     while (!input_reader->is_exhausted())
17     {
18         data.push_back(input_reader->get());
19         input_reader->advance();
20     }
21 }

```

cpp

1.2. Testfälle

Der Standard-Testfall

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

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

cpp

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

```
1 merge_sorter sorter;
2 sorter.sort_file_in_memory(filename);
```

cpp

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

```
1 std::ifstream file(filename);
2 stream_reader<std::string> reader(file);
3
4 std::string prev = reader.get();
5 while (reader.has_next()) {
6     std::string current = reader.get();
7     ASSERT_LE(prev, current) << "Elements are not in sorted order";
8     prev = current;
9 }
```

cpp

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

Test run finished: 20 Tests (20 Passed, 0 Failed, 0 Skipped) run in 20 sec

Test	Duration
01_Beispiel_Test (20)	18,8 sec
<Empty Namespace> (20)	18,8 sec
MergeSortTest (5)	1 sec
TestEmptyFile	1 ms
TestFileWithDifferentStringLengths	910 ms
TestFileWithDuplicates	30 ms
TestFileWithNotOnlyAlphabetic	52 ms
TestReverseSortedFile	7 ms
ParameterizedTests/MergeSorterTest (12)	17,8 sec
TestSortInMemoryParameterized/0 [(2, 10)]	53 ms
TestSortInMemoryParameterized/1 [(2, 200)]	97 ms
TestSortInMemoryParameterized/10 [(20, 5000)]	448 ms
TestSortInMemoryParameterized/11 [(20, 100000)]	7 sec
TestSortInMemoryParameterized/2 [(2, 5000)]	348 ms
TestSortInMemoryParameterized/3 [(2, 100000)]	4,4 sec
TestSortInMemoryParameterized/4 [(10, 10)]	87 ms
TestSortInMemoryParameterized/5 [(10, 200)]	214 ms
TestSortInMemoryParameterized/6 [(10, 5000)]	351 ms
TestSortInMemoryParameterized/7 [(10, 100000)]	4,6 sec
TestSortInMemoryParameterized/8 [(20, 10)]	63 ms
TestSortInMemoryParameterized/9 [(20, 200)]	106 ms
RandomTest (2)	< 1 ms
StreamReaderTest (1)	< 1 ms

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 `ifstream`s und `ofstream`s. 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 Zielfeile 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.

```

1  template<typename T>
2  class FileMergeReader : public IMergeReader<T> {
3  private:
4      std::string _filename;
5      std::unique_ptr<stream_reader<T>> _reader;
6      std::unique_ptr<std::ifstream> _file;
7
8  public:
9      T get() override {
10         if (is_exhausted()) {
11             throw std::underflow_error("No more elements to read");
12         }
13         return _reader->peek();
14     }
15
16     bool advance() override {
17         if (is_exhausted()) {
18             return false;
19         }
20         _reader->get(); // consume current element
21         return _reader->has_next();
22     }
23
24     bool is_exhausted() override {
25         return !_reader->has_next();
26     }
27 };
28
29 template<typename T>
30 class FileMergeWriter : public IMergeWriter<T> {
31 private:
32     std::string _filename;
33     std::unique_ptr<std::ofstream> _file;
34 public:
35     bool append(const T& value) override {
36         if (!_file->is_open()) {
37             return false;
38         }
39         file_manipulator::append(*_file, value);
40         return true;
41     }
42 };

```

cpp

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 `string_length = 100` und `array_length = 1000000` eine Datei mit knapp unter 100 MB Daten. Mit dem zusätzlichen Speichervorbauch der vier Buffer sind das insgesamt ~300 MB Speicher.

Zusätzlich testen wir die Fehlerbehandlung für nicht vorhandene Dateien:

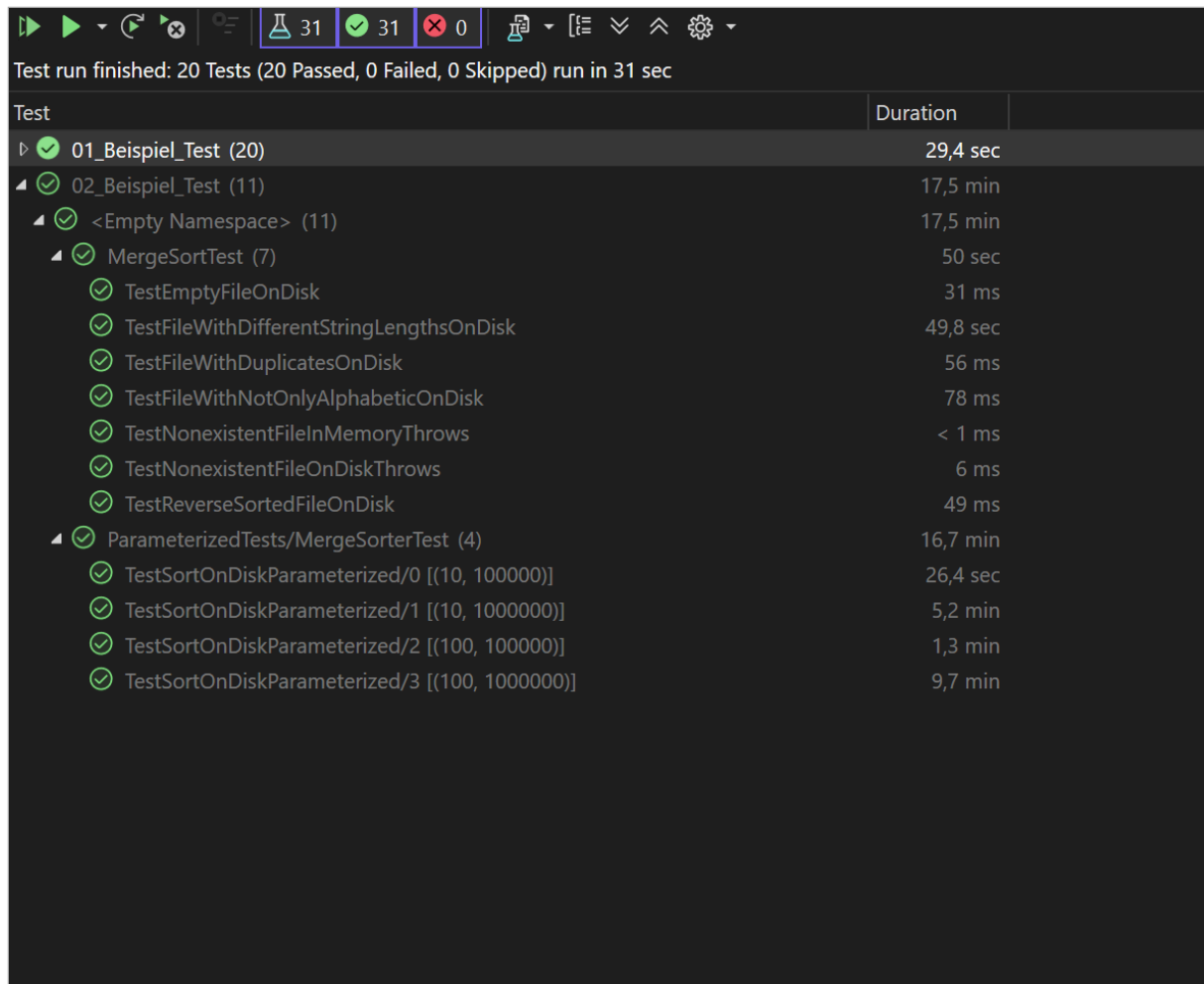
```
1 TEST(MergeSortTest, TestNonexistentFileOnDiskThrows) {  
2     // Arrange  
3     std::string filename = "__no_such_file_exists__.txt";  
4     remove(filename.c_str());  
5  
6     // Act + Assert  
7     merge_sorter sorter;  
8     ASSERT_THROW(sorter.sort_file_on_disk(filename), std::runtime_error);  
9 }
```

cpp

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 MB Datei sortiert, dauert ~10 Minuten. Die Struktur mit `IMergeReader/IMergeWriter` ermöglicht identische Kernlogik für In-Memory und On-Disk, was die Wartbarkeit verbessert.



Test	Duration
01_Beispiel_Test (20)	29,4 sec
02_Beispiel_Test (11)	17,5 min
<Empty Namespace> (11)	17,5 min
MergeSortTest (7)	50 sec
TestEmptyFileOnDisk	31 ms
TestFileWithDifferentStringLengthsOnDisk	49,8 sec
TestFileWithDuplicatesOnDisk	56 ms
TestFileWithNotOnlyAlphabeticOnDisk	78 ms
TestNonexistentFileInMemoryThrows	< 1 ms
TestNonexistentFileOnDiskThrows	6 ms
TestReverseSortedFileOnDisk	49 ms
ParameterizedTests/MergeSorterTest (4)	16,7 min
TestSortOnDiskParameterized/0 [(10, 100000)]	26,4 sec
TestSortOnDiskParameterized/1 [(10, 1000000)]	5,2 min
TestSortOnDiskParameterized/2 [(100, 100000)]	1,3 min
TestSortOnDiskParameterized/3 [(100, 1000000)]	9,7 min

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 höher aber vergleichbar mit der In-Memory Implementierung ist, da der Algorithmus auf der Festplatte arbeiten muss. Der Prozessspeicherbedarf bleibt bei On-Disk annähernd konstant, bei In-Memory nimmt er mit der Größe der Datenfolge zu.

Der Benchmark wurde auf folgender Hardware durchgeführt:

CPU	13th Gen Intel(R) Core(TM) i5-1335U @ 2.5 GHz
Logical Cores	12
RAM	7.62 GB
Arch	x86_64
Kernel	6.6.87.2-microsoft-standard-WSL2
OS	Linux (openSUSE Tumbleweed 20251020)

Tabelle 1: Hardware-Spezifikationen des Benchmark-Systems

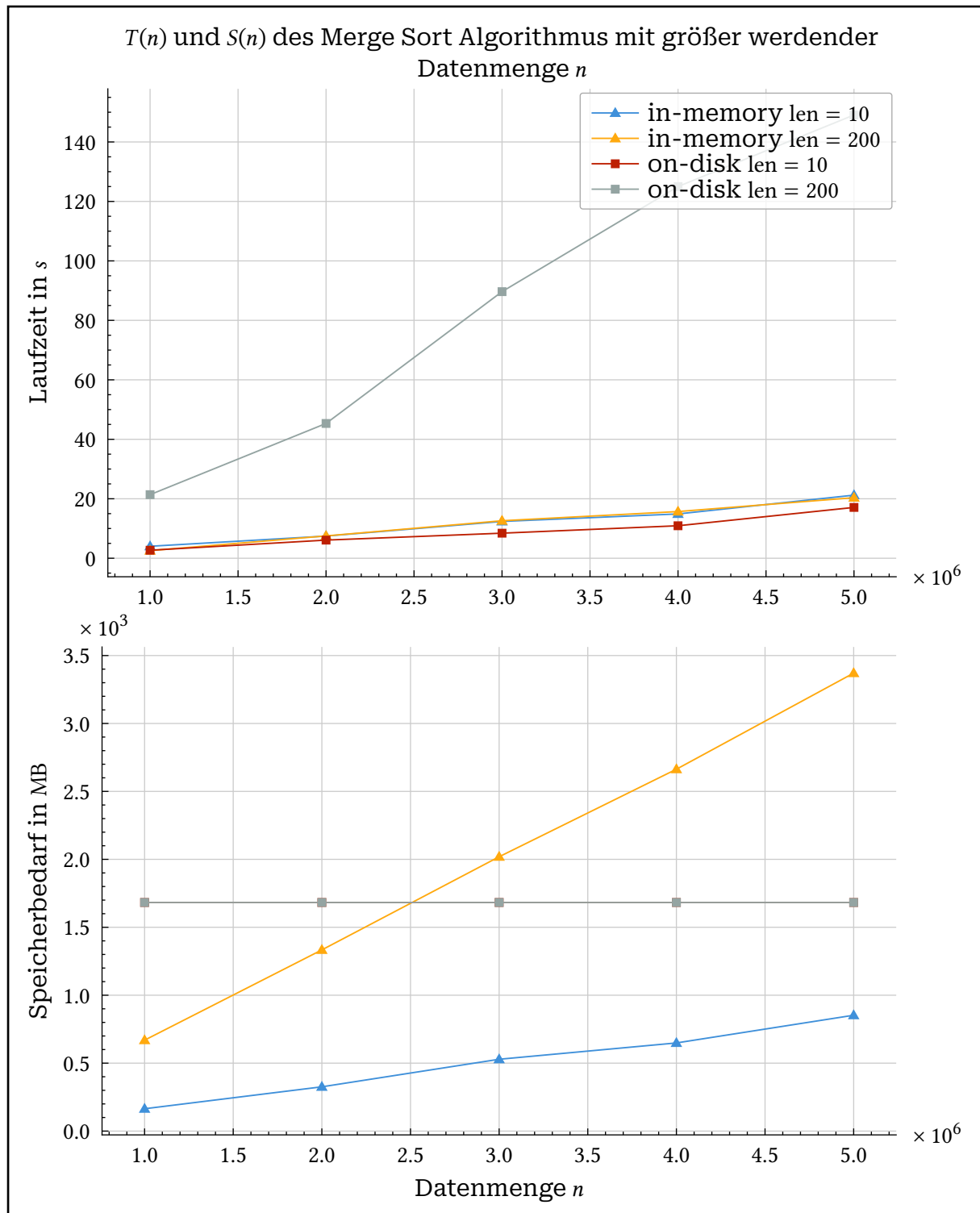


Abbildung 8: Benchmark Ergebnisse des Merge Sort Algorithmus auf einer 13th Gen Intel(R) Core(TM) i5-1335U CPU