SWE3 - Übung 1
WS 2025/26

Tim Peko

Inhaltsverzeichnis

1. Heapsort

1.1. Lösungsansatz

Heapsort sortiert ein Feld in-place, indem zuerst ein Max-Heap aufgebaut und anschließend das Maximum wiederholt an das Ende des aktiven Bereichs getauscht wird. Danach wird die Heap-Eigenschaft im verkleinerten Bereich per „Sinken" (shift down) wiederhergestellt.

- Phase 1 – Heapaufbau (Heapify): Beginnend beim letzten Elternknoten wird für jeden Knoten rückwärts die Heap-Eigenschaft durch `shift_down` hergestellt. Das ist linear in der Feldlänge $O(n)$.
- Phase 2 – Sortieren: Solange der aktive Bereich nicht leer ist, wird das Wurzelelement (Maximum) mit dem letzten Element des aktiven Bereichs getauscht und der aktive Bereich um eins verkleinert. Anschließend wird die Wurzel per `shift_down` wieder abgesunken. Diese Phase kostet $O(n \log(n))$.

Eigenschaften:
- Laufzeit: $O(n)$ für den Aufbau, $O(n \log(n))$ insgesamt im Worst-/Average-Case.
- Speicher: in-place, $O(1)$ zusätzlicher Speicher.
- Stabilität: nicht stabil.

1.1.1. Heap-Eigenschaft

Ein Heap ist ein binärer Baum, der die Heap-Eigenschaft erfüllt. Diese besagt, dass der Wert jedes Knotens größer oder gleich dem Wert seiner Kinder ist. Für einen Max-Heap bedeutet das, dass der Wert des Wurzelknotens größer oder gleich dem Wert seiner Kinder ist.

1.1.2. Heap in-place

Heapsort ist ein in-place-Algorithmus, d.h. er benötigt keinen zusätzlichen Speicher. Alle Operationen werden direkt auf dem zu sortierenden Array durchgeführt.

1.1.3. Index-Berechnung für Heap-Navigation

In einem als Array repräsentierten binären Heap können die Indizes von Eltern- und Kindknoten durch einfache arithmetische Operationen berechnet werden:

- Elternknoten: Für einen Knoten bei Index `i` befindet sich der Elternknoten bei Index $\left\lfloor \frac{i-1}{2} \right\rfloor$
- Linkes Kind: Für einen Knoten bei Index `i` befindet sich das linke Kind bei Index $2i + 1$
- Rechtes Kind: Für einen Knoten bei Index `i` befindet sich das rechte Kind bei Index $2i + 2$

Diese Beziehungen ergeben sich aus der vollständigen binären Baumstruktur, bei der die Knoten ebenenweise von links nach rechts nummeriert werden (beginnend bei Index 0).

Beispiel für Array `[9, 8, 5, 2, 2]`:
- Index 0 (Wert 9): Kinder bei Index 1 und 2
- Index 1 (Wert 8): Eltern bei Index 0, Kinder bei Index 3 und 4
- Index 2 (Wert 5): Eltern bei Index 0, keine Kinder
- Index 3 (Wert 2): Eltern bei Index 1, keine Kinder

- Index 4 (Wert 2): Eltern bei Index 1, keine Kinder

1.1.4. Heapsort-Ablauf am Beispiel

Zur Veranschaulichung des Heapsort-Algorithmus wird der Sortiervorgang am Beispiel des Arrays `[5, 2, 2, 8, 9]` schrittweise dargestellt:

1. Ausgangszustand (unsortiertes Array):

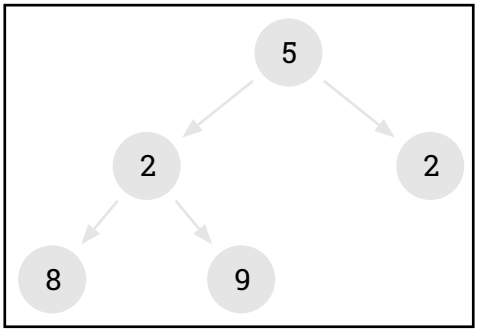

Abbildung 1: Unsortiertes Array als Baum

2. Nach Heapaufbau (Max-Heap):

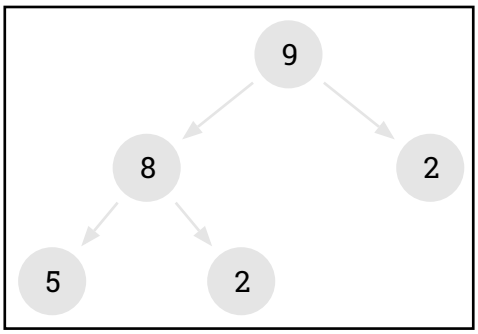

Abbildung 2: Max-Heap nach build_heap

3. Nach erstem Tausch (9 ans Ende):

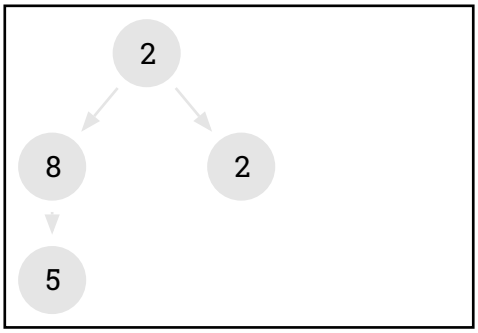

Abbildung 3: Nach Tausch von 9 mit letztem Element

4. Nach shift_down der Wurzel:

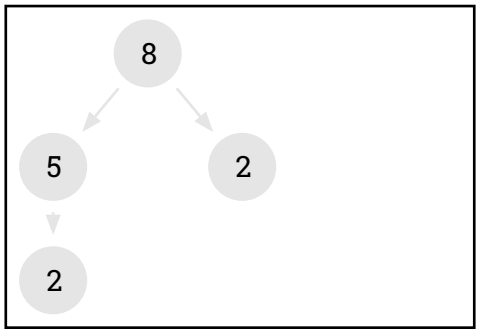

Abbildung 4: Nach Wiederherstellung der Heap-Eigenschaft

5. Sortiertes Ergebnis:
   Das Verfahren wird fortgesetzt, bis alle Elemente sortiert sind: `[2, 2, 5, 8, 9]`

1.2. Source Code

Im Folgenden sind die relevanten Quelltext-Snippets eingebunden, auf deren Basis der Algorithmus erklärt wird.

1.2.1. Signaturen (heapsort.hpp)

```cpp
#pragma once

#include <vector>
#include <iostream>
#include <string>

class heap_sorter
{
public:
  using content_t = std::vector<int>;
  using index_t = content_t::size_type;
  using size_t = content_t::size_type;
  using value_t = content_t::value_type;

  static void sort(content_t &c);
  static void print_content(const content_t &c);
  static void print_as_tree(const content_t &c);

private:
  static void build_heap(content_t &c);
  static void shift_down(content_t &c, index_t start, size_t len);
  static void print_as_tree(
    const content_t &c,
    const index_t i,
    const size_t len,
    const size_t depth,
    std::ostream &out = std::cout);
};
```

1.2.2. Hilfsfunktionen und Absenken (heapsort.cpp)

```cpp
using index_t = heap_sorter::index_t;
static constexpr index_t left(index_t i)
```

```
{
  return (i * 2) + 1;
}
static constexpr index_t right(index_t i)
{
  return (i * 2) + 2;
}
static constexpr index_t parent(index_t i)
{
  return (i - 1) / 2;
}

void heap_sorter::shift_down(content_t &c, index_t start, size_t len)
{
  index_t i = start;
  while (left(i) < len)
  {
    index_t largest = i;
    index_t l = left(i);
    index_t r = right(i);

    if (l < len && c[l] > c[largest])
    {
      largest = l;
    }
    if (r < len && c[r] > c[largest])
    {
      largest = r;
    }
    if (largest == i)
    {
      break;
    }
    std::swap(c[i], c[largest]);
    i = largest;
  }
}
```

1.2.3. Heapaufbau (heapsort.cpp)

```cpp
void heap_sorter::build_heap(content_t &c)
{
  if (c.empty())
    return;
  // Start from the last parent node and move upwards
  for (index_t i = c.size() / 2; i-- > 0;)
  {
    shift_down(c, i, c.size());
    print_as_tree(c);
    std::cout << std::endl;
  }
}
```

1.2.4. Sortierphase (heapsort.cpp)

```cpp
void heap_sorter::sort(content_t &c)
{
  build_heap(c);
  std::cout << "built heap, sorting..." << std::endl;
  for (index_t i = c.size(); i-- > 0;)
  {
    shift_down(c, 0, i + 1);
    std::cout << "shifted down" << std::endl;
    print_as_tree(c, 0, i + 1, 0);
    std::swap(c[0], c[i]);
    std::cout << "swapped" << std::endl;
    print_content(c);
  }
}
```

1.2.5. Baum-Ausgabe (heapsort.cpp)

```cpp
void heap_sorter::print_as_tree(const content_t &c, const index_t i, const
size_t len, const size_t depth, std::ostream &out)
{
  if (i >= len)
  {
    return;
  }

  print_as_tree(c, right(i), len, depth + 2, out);
  for (size_t d = 0; d < depth; ++d)
  {
    out << "  ";
  }
  out << c[i] << std::endl;
  print_as_tree(c, left(i), len, depth + 2, out);
}
```

1.3. Testfälle

Nachfolgend sind die vorhandenen Testfälle aus `main.cpp` beschrieben. Jeder Testfall gibt vor und nach dem Sortieren den Zustand aus und verifiziert das Ergebnis per Vergleich mit dem erwarteten Vektor.

1.3.1. Leeres Array

- Ziel: Prüfen, dass ein leeres Feld unverändert bleibt und keine Fehler auftreten.
- Relevanz: Randfall für korrekte Abbruchbedingungen im Algorithmus (`build_heap` kehrt sofort zurück).

```
Sorting empty array...
{}
built heap, sorting...
TEST PASSED: Empty array remains empty after sorting
```

Abbildung 5: Leeres Array

1.3.2. Negative Elemente

- Ziel: Sicherstellen, dass auch negative Werte korrekt sortiert werden.

- Relevanz: Vergleichsoperatoren funktionieren unabhängig vom Vorzeichen, Heapsort ist wertunabhängig.

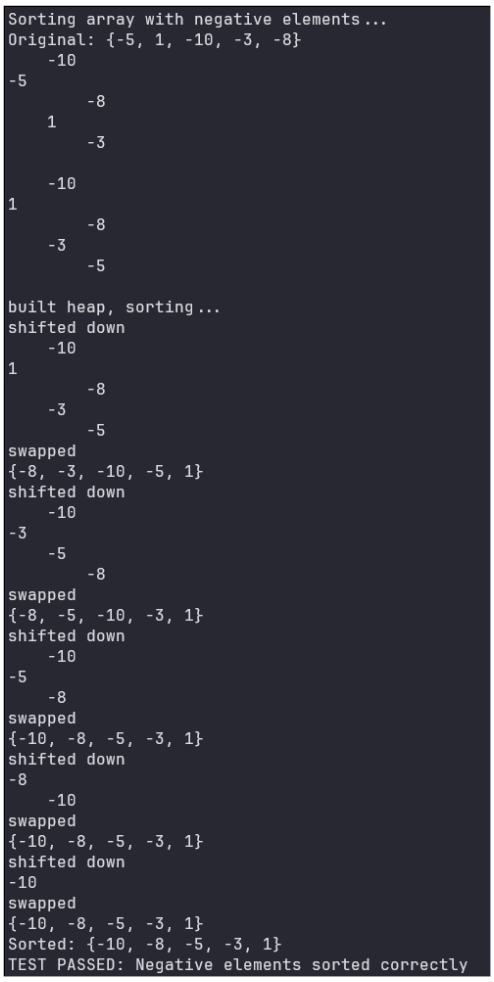

Abbildung 6: Negative Elemente

1.3.3. Absteigend sortiertes Array

- Ziel: Worst-Case-nahe Ordnung prüfen; das Ergebnis muss vollständig aufsteigend sein.
- Relevanz: Belastet die `shift_down`-Operationen besonders stark.

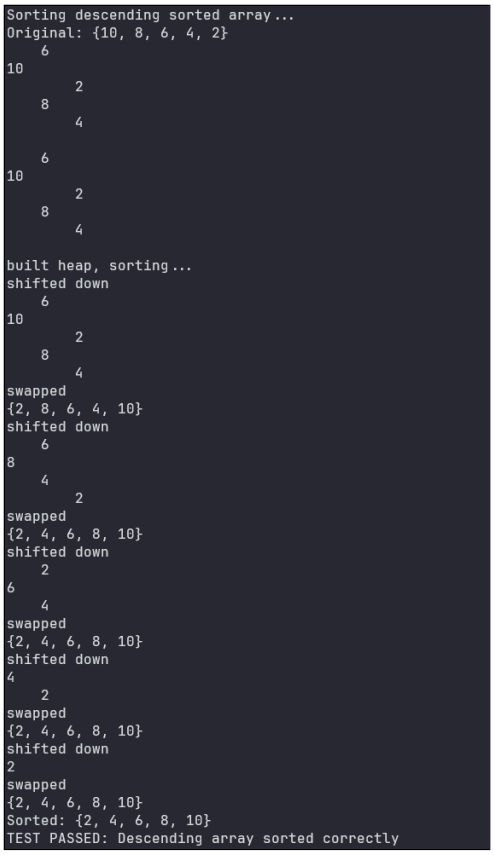

Abbildung 7: Absteigend sortiertes Array

1.3.4. Bereits sortiertes Array

- Ziel: Prüfen, dass ein bereits aufsteigend sortiertes Feld unverändert bleibt.
- Relevanz: Keine unnötigen Änderungen, Korrektheit der Swap-/Heap-Grenzenlogik.

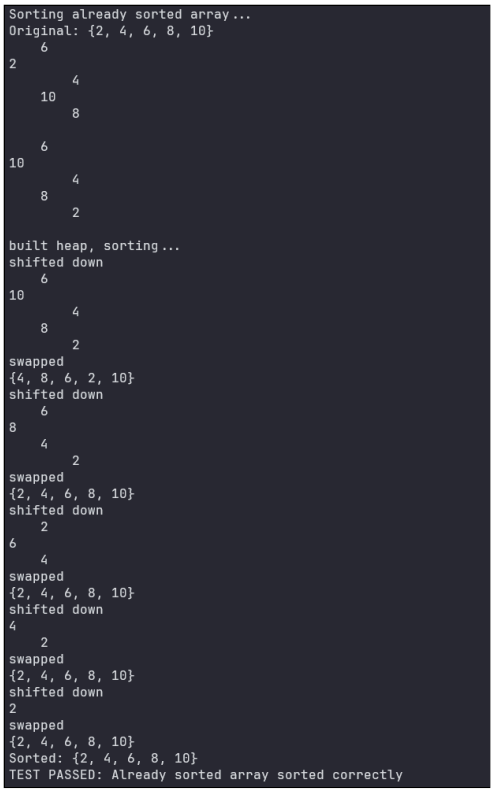

Abbildung 8: Bereits sortiertes Array

1.3.5. Duplikate

- Ziel: Sicherstellen, dass gleiche Werte korrekt gruppiert werden und die Gesamtordnung stimmt.
- Relevanz: Heapsort ist nicht stabil, aber die Sortierung nach Wert muss korrekt sein.

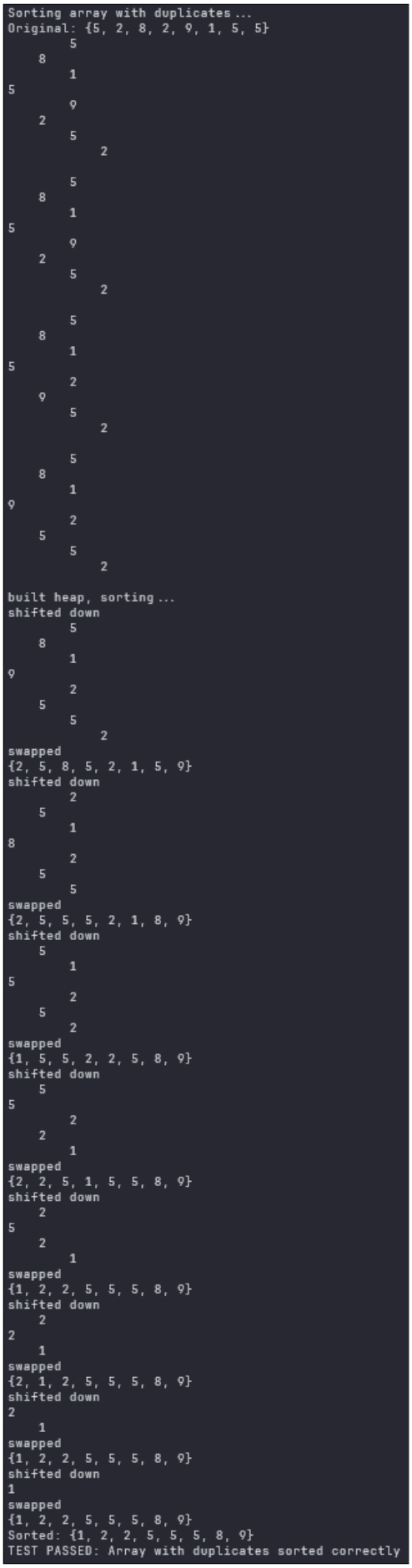

Abbildung 9: Duplikate

2. Heapsort Komplexität

2.1. Lösungsansatz

Für diese Aufgabe wird auf die Lösung der Aufgabe aus Abschnitt 1 zurückgegriffen. In der Umsetzung machen wir uns globale Variablen für die Vergleichs- und Tauschanzahl zunutze und erstellen Hilfsfunktionen, die diese Variablen entsprechend inkrementieren. Diese verwenden wir in der Heapsort Implementierung.

Außerdem wird in der `main.cpp` Funktion `run_for_size` implementiert, die für eine gegebene Array-Größe und Anzahl von Iterationen die durchschnittliche Vergleichs- und Tauschanzahl ermittelt und ausgibt. Diese Funktion wird für verschiedene Array-Größen aufgerufen und die Ergebnisse in Vektoren gespeichert. Die Ergebnisse sind in Abschnitt 2.3 zu sehen.

2.2. Source Code

Counter inkrementierende Hilfsfunktionen:

```cpp
// heapsort.hpp

extern long long count_cmp;
extern long long count_swap;

class heap_sorter
{
private:
  static bool less(const content_t &c, index_t i, index_t j) {
    count_cmp++;
    return c[i] < c[j];
  }

  static void swap(content_t &c, index_t i, index_t j) {
    count_swap++;
    std::swap(c[i], c[j]);
  }
};
```

`run_for_size` und `main` Funktion:

```cpp
// main.cpp

long long count_cmp = 0;
long long count_swap = 0;

std::pair<long long, long long> run_for_size(int size, int iterations)
{
    // Reset counters
    count_cmp = 0;
    count_swap = 0;

    // Take iterations testsamples
    for (int i = 0; i < iterations; i++)
    {
```

```
        std::vector<int> array = generate_random_array(size);
        heap_sorter::sort(array);
    }

    // Calculate average
    auto average_comparisons = count_cmp / iterations;
    auto average_swaps = count_swap / iterations;

    return std::make_pair(average_comparisons, average_swaps);
}

int main()
{
    int sizes[] = {
      100, 200, 500, 1000, 2000, 5000, 10000,
      15000, 20000, 30000, 40000, 60000, 80000, 100000
    };
    int iterations = 25;

    for (int size : sizes)
    {
        auto result = run_for_size(size, iterations);
        // Do something with the result, like printing it
    }

    return 0;
}
```

Geänderte Funktionen in der Heapsort Implementierung:

```cpp
// heapsort.cpp

void heap_sorter::sort(content_t &c)
{
  build_heap(c);
  for (index_t i = c.size(); i-- > 0;)
  {
    shift_down(c, 0, i + 1);
    swap(c, 0, i);
  }
}

void heap_sorter::shift_down(content_t &c, index_t start, size_t len)
{
  index_t i = start;
  while (left(i) < len)
  {
    index_t largest = i;
    index_t l = left(i);
    index_t r = right(i);

    if (l < len && less(c, largest, l))
    {
      largest = l;
```

```
    }
    if (r < len && less(c, largest, r))
    {
      largest = r;
    }
    if (largest == i)
    {
      break;
    }
    swap(c, i, largest);
    i = largest;
  }
}
```

2.3. Analyse

2.3.1. Komplexität

Das ermitteln der Vergleichs- & Tauschanzahl für die verschiedenen Array-Größen liefert die folgenden Ergebnisse:

```
Size: 20000
Average comparisons: 510720
Average swaps: 268385

Size: 30000
Average comparisons: 800704
Average swaps: 419914

Size: 40000
Average comparisons: 1101442
Average swaps: 576753

Size: 60000
Average comparisons: 1721334
Average swaps: 899780

Size: 80000
Average comparisons: 2363025
Average swaps: 1233633

Size: 100000
Average comparisons: 3019617
Average swaps: 1574933

Compares: [1030, 2455, 7427, 16854, 37710, 107713, 235329, 370337, 510720, 800704, 1101442, 1721334, 2363025, 3019617]
Swaps: [582, 1357, 4039, 9077, 20160, 57119, 124174, 194942, 268385, 419914, 576753, 899780, 1233633, 1574933]
```

Abbildung 10: Ausgabe der Vergleichs- & Tauschanzahl

Grafisch dargestellt in Abbildung 11 lässt sich deutlich eine Komplexität von $O(n \log(n))$ erkennen. Zum Vergleich wird noch die Skalierungsformen $O(n)$ als Referenz in der Grafik abgebildet. Dabei wird hier nur die Zeitkomplexität betrachtet. Die Speicherkomplexität ist $O(1)$, weil der Algorithmus in-place arbeitet. Das bedeutet, er benötigt keinen zusätzlichen Speicherplatz. Also Konstant im Bezug auf die Array-Größe n.

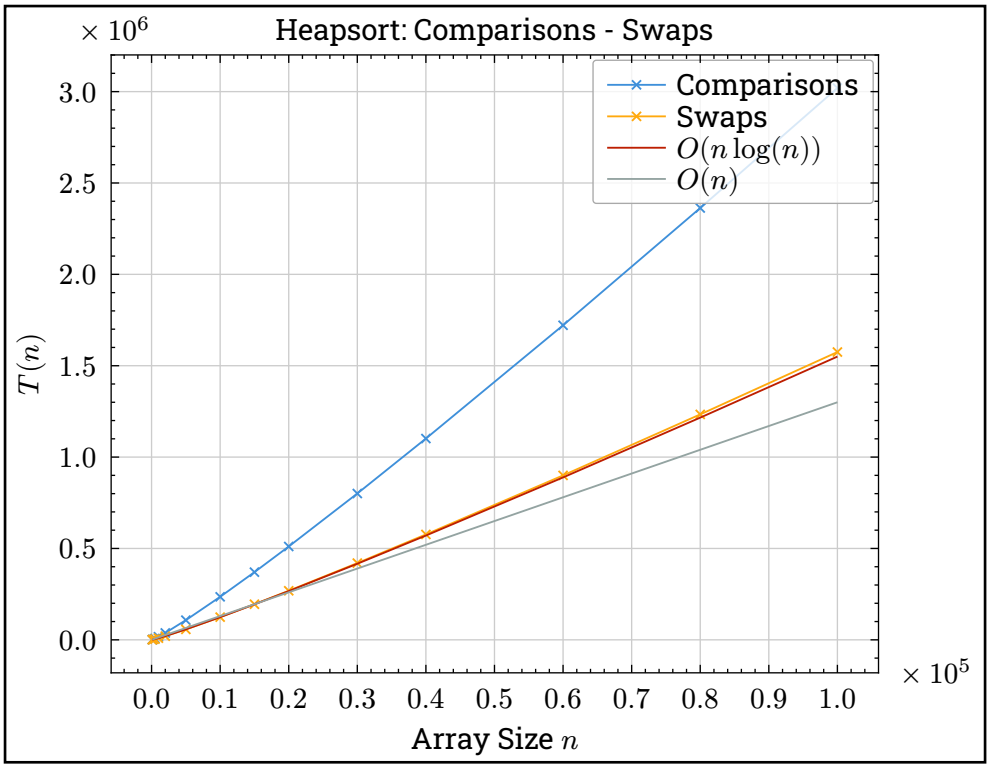

Abbildung 11: Komplexität von Comparisons und Swaps

Array Size n	Comparisons	Swaps
100	1030	582
200	2455	1357
500	7427	4039
1000	16854	9077
2000	37710	20160
5000	107713	57119
10000	235329	124174
15000	370337	194942
20000	510720	268385
30000	800704	419914
40000	1101442	576753
60000	1721334	899780
80000	2363025	1233633
100000	3019617	1574933

Tabelle 1: Comparisons und Swaps abhängig von Array Size n

Tabelle 1 zeigt die in der Abbildung 11 dargestellten Werte.

Aufwand in h: 6