



Hochschule Karlsruhe
Technik und Wirtschaft

UNIVERSITY OF APPLIED SCIENCES

PROJEKTBERICHT

Optimierung von Programmen

Autor:

Adrian WEBER

Betreuer:

Prof. Dr. Christian PAPE

12. Januar 2017

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Systemumgebung | 3 |
| 2 | Hinweise zur Zeitmessung | 4 |
| 3 | Verwendete Algorithmen | 5 |
| 3.1 | Minimumsuche | 5 |
| 3.2 | Sortieren durch direkte Auswahl | 6 |
| 3.3 | Sortieren durch Einfügen | 8 |
| 3.4 | Bottom-Up Mergesort | 11 |
| 3.5 | Quicksort | 12 |
| 4 | Zeitmessung: Ergebnisse | 13 |
| 4.1 | Minimumsuche | 13 |
| 4.2 | Sortieren durch direkte Auswahl | 16 |
| 4.3 | Sortieren durch Einfügen | 18 |
| 4.4 | Bottom-Up Mergesort | 19 |
| 4.5 | Quicksort | 20 |

1 Systemumgebung

Die benötigten Komponenten für das Labor (Compiler, Debugger) für dieses Labor wurden mit *Cygwin* installiert. Die genauen Daten zur relevanten Hardware und den Versionen der Komponenten können aus Tabelle 1 entnommen werden.

| | |
|--------------------------|--|
| Gerätebezeichnung | Samsung RFC 730 SE07DE |
| Prozessor | Intel(R) Core(TM) i7-2670QM COU @2.20GHz |
| Arbeitsspeicher | 8GB (1333 Mhz) |
| Festplatte | Samsung SSD 850 EVO 250GB |
| Systemtyp | 64-Bit Betriebssystem |
| Betriebssystem | Windows 10 Home Version 1607 (Build 14393.351) |
| Compiler | gcc-g++ Version 5.4.0-1 |
| Debugger | gdb 7.11.1-2 |

Tabelle 1: Beschreibung des Testsystems.

2 Hinweise zur Zeitmessung

In diesem Projektbericht werden für verschiedene Algorithmen Zeitmessungen vorgenommen und tabellarisch dokumentiert. Alle Messungen finden unter den Bedingungen der in Kapitel 1 beschriebenen Systemumgebung statt. Zusätzlich gelten für die Zeitmessung folgende Vorgaben.

- Zeitmessungen werden mit dem Datentyp `double` durchgeführt.
- Für die Messungen wird die Bibliothek `std::chrono` in C++ verwendet.
- Die Feldgrößen entsprechen einer 2er-Potenz und werden solange verdoppelt, bis der Hauptspeicher nicht mehr ausreicht. Gestartet wird mit der Feldgröße $n := 65536$.
- Es gibt insgesamt drei Testszenarien für die Zeitmessung.
 - Der Algorithmus wird mit einer aufsteigen sortierten Folge getestet (In den Tabellen mit `ASC` bezeichnet).
 - Der Algorithmus wird mit einer absteigend sortierten Folge getestet (In den Tabellen mit `DESC` bezeichnet).
 - Der Algorithmus wird mit einer folge von generierten Zufallszahlen getestet (In den Tabellen mit `RAND` bezeichnet).

3 Verwendete Algorithmen

In diesem Abschnitt werden die verwendeten Algorithmen kurz vorgestellt, dabei wird Bezug auf Funktionsweise und die Komplexität jedes Algorithmus genommen. Zusätzlich wird die Ausführung in einem kleinen Beispiel gezeigt.

3.1 Minimumsuche

Die Minimumsuche sucht das kleinste Element in einer Menge von vergleichbaren Elementen und gibt dieses zurück. Dabei müssen alle Elemente mindestens einmal betrachtet werden, was eine Laufzeit von $O(n)$ zur Folge hat. Der nachfolgende Pseudocode verdeutlicht die Vorgehensweise des Algorithmus.

```
1 A := array with comparable elements;
2 minimum = A[0];
3 for i = 1 to n do
4     if minimum > A[i] then
5         minimum = A[i];
6     end if
7     i := i + 1;
8 end for
```

Beispiel 3.1.1 - Ausführung der Minimumsuche

Für die Ausführung des Algorithmus wird das Array A , eine Variable $minimum$ und eine Zählvariable i wie folgt initialisiert.

$$A := \{7, 4, 17, 1, 6, 7\}, \text{ minimum} := A[0] = 7, i := 1 \quad (1)$$

Im ersten Schritt der Schleife wird nun die Stelle $A[i]$ (blau) mit dem aktuellen Minimum (rot) verglichen. Ist die Stelle $A[i]$ größer als das aktuelle Minimum, wird dieses aktualisiert. Das ganze wiederholt sich, bis jedes Element einmal betrachtet wurde. Nachfolgend der Zustand der Variablen in Zeile vier der Minimumsuche.

$$\begin{aligned} A &= \{7, 4, 17, 1, 6, 7\}, \text{ minimum} = 7, i = 1 \\ A &= \{7, 4, 17, 1, 6, 7\}, \text{ minimum} = 4, i = 2 \\ A &= \{7, 4, 17, 1, 6, 7\}, \text{ minimum} = 4, i = 3 \\ A &= \{7, 4, 17, 1, 6, 7\}, \text{ minimum} = 1, i = 4 \\ A &= \{7, 4, 17, 1, 6, 7\}, \text{ minimum} = 1, i = 5 \end{aligned} \quad (2)$$

3.2 Sortieren durch direkte Auswahl

Der Algorithmus *Sortieren durch direkte Auswahl* (englisch Selectionsort) sortiert ein Array von Elementen in aufsteigender Reihenfolge. Dabei sucht der Algorithmus das kleinste Element im Array und vertauscht es mit der ersten Stelle im Array. Anschließend wird im Array ohne die erste Stelle erneut das Minimum gesucht und an die zweite Stelle getauscht. So entsteht ein sortierter Teil S und ein unsortierter Teil U im Array. Der Algorithmus fährt solange fort, bis in U keine Elemente mehr vorhanden sind. Der Algorithmus liegt in der Komplexitätsklasse $O(n^2)$.

```
1  A := array with comparable elements;
2  for i = 0 to n - 1 do
3      minimum = searchMin(A, i);
4      switch(A[i], minimum);
5  end for
```

Beispiel 3.2.1 - Ausführung Sortieren durch direkte Auswahl

Für die Ausführung des Algorithmus wird das Array A und eine Zählvariable i initialisiert. Zur Verdeutlichung werden i blau und das aktuell gefundene Minimum rot markiert. Der bereits sortierte Teil des Arrays wird grün markiert

$$A := \{7, 4, 17, 1, 6, 7\}, i := 0 \quad (3)$$

Der Algorithmus startet bei $A[0] = 7$, dazu wird im Bereich $A[0] - A[5]$ das Minimum 1 gefunden.

$$A = \{\textcolor{red}{7}, 4, 17, \textcolor{blue}{1}, 6, 7\}, i = 0 \quad (4)$$

Anschließend wird das Minimum mit dem aktuellen Index vertauscht, da $7 > 1$.

$$A = \{\textcolor{green}{1}, 4, 17, 7, 6, 7\}, i = 0 \quad (5)$$

Nach dem vertauschen startet der nächste Schleifendurchlauf mit $A[1] = 4$.

$$A = \{\textcolor{green}{1}, \textcolor{red}{4}, 17, 7, 6, 7\}, i = 1 \quad (6)$$

In diesem Fall muss nicht getauscht werden da 4 bereits das Minimum ist.

$$A = \{\textcolor{green}{1}, \textcolor{green}{4}, 17, 7, 6, 7\}, i = 1 \quad (7)$$

Im nächsten Schritt wird $A[2] = 17$ mit dem gefundenen Minimum 6 aus $A[2] - A[5]$ behandelt.

$$A = \{1, 4, 17, 7, 6, 7\}, i = 2 \quad (8)$$

Wieder wird vertauscht, da $17 > 6$.

$$A = \{1, 4, 6, 7, 17, 7\}, i = 2 \quad (9)$$

Für $A[3] = 7$ wird kein Minimum gefunden, da 7 bereits das kleinste Element ist.

$$A = \{1, 4, 6, 7, 17, 7\}, i = 3 \quad (10)$$

Anschließend werden noch $A[4] = 17$ mit dem letzten Minimum 7 aus $A[4] - A[5]$ betrachtet.

$$A = \{1, 4, 6, 7, 17, 7\}, i = 4 \quad (11)$$

Die Elemente werden vertauscht, da $17 > 7$.

$$A = \{1, 4, 6, 7, 7, 17\}, i = 4 \quad (12)$$

Anschließend terminiert der Algorithmus, dass letzte Element muss nicht mehr geprüft werden, da es bereits im letzte Schritt behandelt wurde.

3.3 Sortieren durch Einfügen

Der Algorithmus *Sortieren durch Einfügen* (englisch Insertionsort) sortiert ein Array von Elementen in aufsteigender Reihenfolge. Dabei iteriert der Algorithmus in einer äußeren Schleife über das gesamte Array. Jedes Element wird anschließend in einer inneren Schleife mit jedem Vorgänger verglichen und vertauscht, solange dieser größer ist wie das aktuelle Element. Am Ende entsteht so ein vollständig sortiertes Array. Der Algorithmus liegt in der Komplexitätsklasse $O(n^2)$.

```
1  A := array with comparable elements;
2  for i = 1 to n do
3      for j = i to j > 0 and A[j - 1] > A[j] do
4          switch(A[j], A[j - 1]);
5      end for
6  end for
```

Beispiel 3.3.1 - Ausführung Sortieren durch Einfügen

Für die Ausführung des Algorithmus wird das Array A und zwei Zählvariablen i und j initialisiert. Zur Verdeutlichung wird i blau, j und $j - 1$ rot und der bereits sortierte Teil des Arrays grün dargestellt.

$$A := \{7, 4, 17, 1, 6, 7\}, i := 1, j := 0 \quad (13)$$

Der Algorithmus startet mit $i = 1$ bei $A[1] = 4$ in der äußeren Schleife

$$A = \{7, 4, 17, 1, 6, 7\}, i = 1, j = 0 \quad (14)$$

In der inneren Schleife wird $j = i = 1$ gesetzt und $A[j]$ mit $A[j - 1]$ verglichen.

$$A = \{7, 4, 17, 1, 6, 7\}, i = 1, j = 1 \quad (15)$$

Da $7 > 4$ werden die beiden Elemente vertauscht und j anschließend dekrementiert.

$$A = \{4, 7, 17, 1, 6, 7\}, i = 1, j = 0 \quad (16)$$

Da $j = 0$ wird die innere Schleife abgebrochen und mit der äußeren fortgefahren.

$$A = \{4, 7, 17, 1, 6, 7\}, i = 2, j = 0 \quad (17)$$

Anschließend wird wieder die innere Schleife mit $j = i = 2$ ausgeführt.

$$A = \{4, \textcolor{red}{7}, \textcolor{red}{17}, 1, 6, 7\}, i = 2, j = 2 \quad (18)$$

Da bereits $7 < 17$ muss nicht getauscht werden und die innere Schleife wird abgebrochen.

$$A = \{4, 7, 17, \textcolor{blue}{1}, 6, 7\}, i = 3, j = 2 \quad (19)$$

Die innere Schleife wird jetzt mit $j = i = 3$ ausgeführt. Da $17 > 1$ werden die Elemente $A[j]$ mit $A[j - 1]$ vertauscht. Das wiederholt sich bis $j = 0$, da außerdem $7 > 1$ und $4 > 1$. Anschließend terminiert die innere Schleife wegen $j = 0$.

$$\begin{aligned} A &= \{4, 7, \textcolor{red}{17}, \textcolor{red}{1}, 6, 7\}, i = 3, j = 3 \\ A &= \{4, \textcolor{red}{7}, \textcolor{red}{1}, 17, 6, 7\}, i = 3, j = 2 \\ A &= \{\textcolor{red}{4}, \textcolor{red}{1}, 7, 17, 6, 7\}, i = 3, j = 1 \\ A &= \{\textcolor{red}{1}, 4, 7, 17, 6, 7\}, i = 3, j = 0 \end{aligned} \quad (20)$$

Die äußere Schleife wird jetzt mit $i = 4$ fortgesetzt.

$$A = \{1, 4, 7, 17, \textcolor{blue}{6}, 7\}, i = 4, j = 0 \quad (21)$$

Die innere Schleife wird dann mit $j = i = 4$ ausgeführt. Da $17 > 6$ werden die Elemente $A[j]$ mit $A[j - 1]$ vertauscht. Das wiederholt sich bis $j = 2$, da außerdem $7 > 6$. Anschließend terminiert die innere Schleife wegen $A[j - 1] < A[j]$.

$$\begin{aligned} A &= \{1, 4, 7, \textcolor{red}{17}, \textcolor{red}{6}, 7\}, i = 4, j = 4 \\ A &= \{1, 4, \textcolor{red}{7}, \textcolor{red}{6}, 17, 7\}, i = 4, j = 3 \\ A &= \{1, \textcolor{red}{4}, \textcolor{red}{6}, 7, 17, 7\}, i = 4, j = 2 \end{aligned} \quad (22)$$

Die äußere Schleife wird mit dem letzten Element $i = 5$ fortgesetzt.

$$A = \{1, 4, 6, 7, 17, \textcolor{blue}{7}\}, i = 4, j = 2 \quad (23)$$

Die innere Schleife startet bei $j = i = 5$ und vergleicht $A[j - 1]$ mit $A[j]$. Da $17 > 7$ werden die Elemente vertauscht, danach bricht die Schleife ab da $A[j - 1] = A[j]$.

$$\begin{aligned} A &= \{1, 4, 6, 7, \textcolor{red}{17}, \textcolor{red}{7}\}, i = 4, j = 3 \\ A &= \{1, 4, 6, \textcolor{red}{7}, \textcolor{red}{7}, 17\}, i = 4, j = 3 \end{aligned} \tag{24}$$

Anschließend Terminiert die äußere Schleife wegen $i = N$ und der Algorithmus ist fertig.

$$A = \{\textcolor{green}{1}, \textcolor{green}{4}, \textcolor{green}{6}, \textcolor{green}{7}, \textcolor{green}{7}, \textcolor{green}{17}\}, i = 4, j = 2 \tag{25}$$

3.4 Bottum-Up Mergesort

3.5 Quicksort

4 Zeitmessung: Ergebnisse

4.1 Minimumsuche

Folgende Zeitmessungen zeigen die Laufzeiten der Minimumsuche mit drei verschiedenen Implementierungen: Einmal mit einfacher Schleife (Normal), mit ausgerollten Schleifen und mit einem Prefetch-Befehl.

4.1.1 Variante 1: Normal

| Feldgröße [n] | Laufzeit [μs] | | |
|---------------|----------------------|--------|--------|
| | ASC | DESC | RAND |
| 16384 | 30 | 29 | 29 |
| 32768 | 54 | 61 | 58 |
| 65536 | 111 | 123 | 108 |
| 131072 | 233 | 232 | 213 |
| 262144 | 421 | 479 | 425 |
| 524288 | 880 | 886 | 903 |
| 1048576 | 1740 | 1737 | 1743 |
| 2097152 | 3764 | 3751 | 3566 |
| 4194304 | 6926 | 6951 | 7049 |
| 8388608 | 13348 | 13548 | 13558 |
| 16777216 | 27043 | 27975 | 28325 |
| 33554432 | 55425 | 53431 | 54039 |
| 67108864 | 109768 | 106431 | 108913 |
| 134217728 | 202659 | 206078 | 204265 |
| 268435456 | 409796 | 400175 | 444034 |
| 536870912 | 1077360 | 961671 | 955176 |

Tabelle 2: Normale Minimumsuche.

4.1.2 Variante 2: Mit Schleifen Ausrollen

| Feldgröße [n] | Laufzeit [μs] | | |
|---------------|----------------------|--------|--------|
| | ASC | DESC | RAND |
| 16384 | 31 | 30 | 28 |
| 32768 | 58 | 64 | 59 |
| 65536 | 106 | 108 | 109 |
| 131072 | 216 | 233 | 230 |
| 262144 | 426 | 441 | 443 |
| 524288 | 901 | 914 | 889 |
| 1048576 | 1746 | 1758 | 1773 |
| 2097152 | 3530 | 3518 | 3513 |
| 4194304 | 7045 | 7041 | 7035 |
| 8388608 | 13597 | 13635 | 14096 |
| 16777216 | 27933 | 28018 | 27096 |
| 33554432 | 53879 | 55001 | 52187 |
| 67108864 | 104367 | 101978 | 104117 |
| 134217728 | 216644 | 216945 | 215795 |
| 268435456 | 444691 | 460349 | 466158 |
| 536870912 | 840530 | 813142 | 899364 |

Tabelle 3: Minimumsuche mit Schleifen ausrollen

4.1.3 Variante 3: Mit Prefetch

| Feldgröße [n] | Laufzeit [μs] | | |
|---------------|----------------------|--------|---------|
| | ASC | DESC | RAND |
| 16384 | 26 | 27 | 26 |
| 32768 | 57 | 54 | 53 |
| 65536 | 98 | 99 | 97 |
| 131072 | 197 | 196 | 198 |
| 262144 | 395 | 397 | 395 |
| 524288 | 841 | 831 | 831 |
| 1048576 | 1617 | 1614 | 1617 |
| 2097152 | 3286 | 3280 | 3286 |
| 4194304 | 6532 | 6412 | 6376 |
| 8388608 | 12936 | 12353 | 12426 |
| 16777216 | 25763 | 26332 | 25283 |
| 33554432 | 50038 | 49740 | 52096 |
| 67108864 | 98898 | 106837 | 103934 |
| 134217728 | 203489 | 198460 | 199950 |
| 268435456 | 408708 | 381526 | 407902 |
| 536870912 | 834184 | 831809 | 1117660 |

Tabelle 4: Minimumsuche mit Prefetch.

4.1.4 Interpretation

Bei genauerer Betrachtung der Tabellen 2, 3 und 4 sind zunächst keine großen Unterschiede zu erkennen. Die Laufzeit wächst, wie erwartet, konstant mit der Feldgröße. Auch zwischen den unterschiedlichen Varianten der Array-Befüllung sind keine nennenswerten Auswirkungen auf die Laufzeit festzustellen, da in jedem Fall immer alle Elemente untersucht werden müssen.

Zwischen der normalen Minimumsuche und der Variante mit Schleifen ausrollen sind können keine Unterschiede in der Laufzeit festgestellt werden; allein die Variante mit Prefetch zeigt einen kleinen Gewinn bei der Laufzeit.

4.2 Sortieren durch direkte Auswahl

Die nachfolgenden Tabellen zeigen die Laufzeiten von drei verschiedenen Varianten von *Sortieren durch direkte Auswahl*. Dabei wird bei jeder Variante eine andere Implementierung der Minimumsuche (siehe Abschnitt oben) verwendet.

4.2.1 Variante 1: Normal

| Feldgröße [n] | Laufzeit [ms] | | |
|---------------|---------------|---------|---------|
| | ASC | DESC | RAND |
| 16384 | 242 | 248 | 250 |
| 32768 | 977 | 970 | 968 |
| 65536 | 3761 | 3768 | 3765 |
| 131072 | 15548 | 15534 | 15539 |
| 262144 | 62171 | 62314 | 62207 |
| 524288 | 252007 | 258822 | 251515 |
| 1048576 | 1052480 | 1027300 | 1028070 |

Tabelle 5: Sortieren durch direkte Auswahl mit normaler Minimumsuche.

4.2.2 Variante 2: Mit Schleifen Ausrollen

| Feldgröße [n] | Laufzeit [ms] | | |
|---------------|---------------|--------|--------|
| | ASC | DESC | RAND |
| 16384 | 158 | 157 | 156 |
| 32768 | 638 | 638 | 636 |
| 65536 | 2871 | 4280 | 2551 |
| 131072 | 10134 | 10148 | 10152 |
| 262144 | 40565 | 40554 | 40590 |
| 524288 | 162909 | 162888 | 165314 |
| 1048576 | 698178 | 678409 | 693544 |

Tabelle 6: Sortieren mit Minimumsuche und Schleifen ausrollen.

4.2.3 Variante 3: Mit Prefetch

| Feldgröße [n] | Laufzeit [ms] | | |
|---------------|---------------|--------|--------|
| | ASC | DESC | RAND |
| 16384 | 166 | 164 | 163 |
| 32768 | 632 | 633 | 630 |
| 65536 | 2533 | 2531 | 2538 |
| 131072 | 10128 | 10127 | 10150 |
| 262144 | 40515 | 40519 | 40549 |
| 524288 | 162415 | 162706 | 162673 |
| 1048576 | 713851 | 696861 | 697996 |

Tabelle 7: Sortieren mit Minimumsuche und Prefetch

4.2.4 Interpretation

Die Laufzeit aller drei Varianten von *Sortieren durch direkte Auswahl* wächst exponentiell, also mit $O(n^2)$. Ein deutlicher Unterschied zeigt sich insbesondere zwischen der Variante mit normaler Minimumsuche und den beiden optimierten Varianten. Die Minimumsuche mit Schleifen ausrollen beziehungsweise mit Prefetch bringen hier einen Zeitvorteil von ca. 34%.

Vergleicht man dagegen Die Laufzeiten der beiden optimierten Varianten, lässt sich hier keinen Unterschied feststellen. Im vorherigen Kapitel wurde bei der Variante mit Prefetch ein kleiner Vorteil festgestellt, dieser lässt sich bei der Verwendung im Sortierverfahren nicht wieder finden.

4.3 Sortieren durch Einfügen

Die nachfolgenden Laufzeiten vergleichen zwei unterschiedliche Implementierungen des Algorithmus *Sortieren durch Einfügen*. In der ersten Variante wird mit zwei Schleifen sortiert, in der zweiten Abwandlung wird die innere Schleife ausgerollt und ein Prefetch-Befehl verwendet.

4.3.1 Variante 1: Normal

4.3.2 Variante 2: Mit Prefetch

4.3.3 Interpretation

4.4 Bottom-Up Mergesort

Nachfolgende Tabelle zeigt die Laufzeit des *Bottom-Up Mergesort* Algorithmus.

4.4.1 Interpretation

4.5 Quicksort

Nachfolgend werden die Laufzeiten von zwei Variante des *Quicksort* Algorithmus verglichen. Einmal wird der Quicksort mit Three-Way-Partitioning, das zweite mal Hybrid implementiert.

4.5.1 Variante 1: Three-Way-Partitioning

4.5.2 Variante 2: Hybrid

4.5.3 Interpretation