

4 Praktikum 4: Sortierverfahren / Hashing

Ihre Vorgesetzten bei der Firma “Data Fuse Inc.” sind begeistert von Ihren Fähigkeiten! Da die Verarbeitungsgeschwindigkeit der enormen Datenmengen weiter optimiert werden muss wurden Sie beauftragt, ein Framework zur Messung von Laufzeiten zu entwickeln. Mit Hilfe dieses Programms sollen Sie anschließend die Ausführungsdauer verschiedener Sortieralgorithmen in Abhängigkeit einer Problemgröße n untersuchen und auswerten. Da exakte Zeitmessungen in C/C++ nicht trivial sind, brauchen Sie dies nicht selber zu implementieren. Stattdessen sollen Sie OpenMP nutzen, um die Zeiten zu messen (Vorteil: einheitlich und einfache Nutzung). Im zweiten Teil der Aufgabe soll eine einfache Hashtabelle implementiert werden.

4.1 Teilaufgabe 1

Für alle Aufgaben gilt, dass Sie hierzu die Vorlage aus ILIAS benutzen können, in der bereits der Programmruumpf sowie ein Benchmarkaufruf für einen Sortieralgorithmus exemplarisch implementiert ist. Sie dürfen aber auch gerne eine komplett eigene Lösung erstellen, bzw die Vorlage Ihren Wünschen gemäß anpassen.

1. Vervollständigen Sie die Sortieralgorithmenbibliothek, bestehend aus der Header-Datei *sorting.h* und implementieren Sie die folgenden Algorithmen in der zugehörigen cpp-Datei *sorting.cpp* sowie im eigenen Namespace *sorting*:

- Heapsort
- Mergesort

Verwenden Sie hier bitte für die Merge-Methode den Algorithmus aus der Vorlesung (s. Folie 12 von der Vorlesung ADS_K6.Sortieralgorithmen_TeilIII.pdf)

- Natürlicher Mergesort

Der Unterschied zwischen dem originalen Mergesort und dem natürlichen Mergesort besteht im ersten Schritt der rekursiven Aufteilung in verschiedene Teillisten. Der Original-Mergesort unterteilt die *gesamte* Zahlenmenge soweit bis Teillisten der Größe 1 entstehen und setzt diese dann rekursiv wieder zusammen während dabei 2 benachbarte Teilfolgen gemischt und dadurch sortiert werden.

Der natürliche Mergesort unterteilt nicht die gesamte Zahlenmenge in Teillisten der Größe 1, sondern erkennt bereits vorsortierte Teilfolgen (sogenannte *runs*) und unterteilt die Teillisten in diese *runs*. Sobald dies geschehen ist, setzt der natürliche Mergesort die Teillisten iterativ auf ähnliche Weise wieder zusammen wie der normale Mergesort. In den Abbildungen 1 und 2 ist der Unterschied zwischen diesen beiden Verfahren bei einer gleichen Zahlenmenge erkennbar. **Wichtiger Hinweis! Sie brauchen den Natural Mergesort nicht effizient für sehr große Zahlenmengen implementieren (Kein Benchmarktest). Allerdings muss Ihr Algorithmus korrekt arbeiten und die Unit-test erfolgreich bestehen (kleine Daten richtig sortieren).**

Mergesort

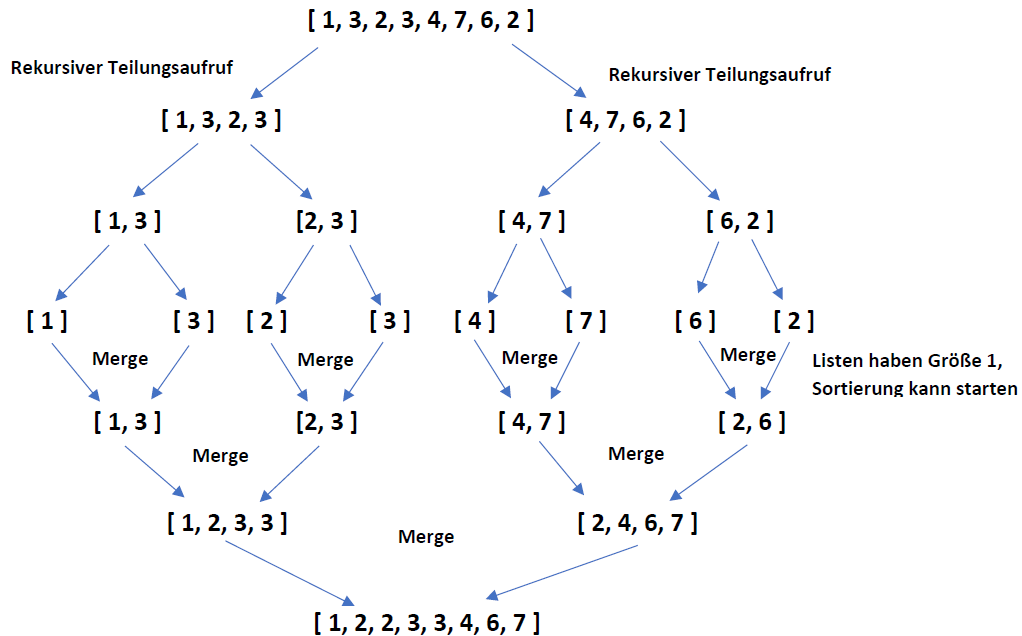


Abbildung 1: Mergesort

Natural Mergesort

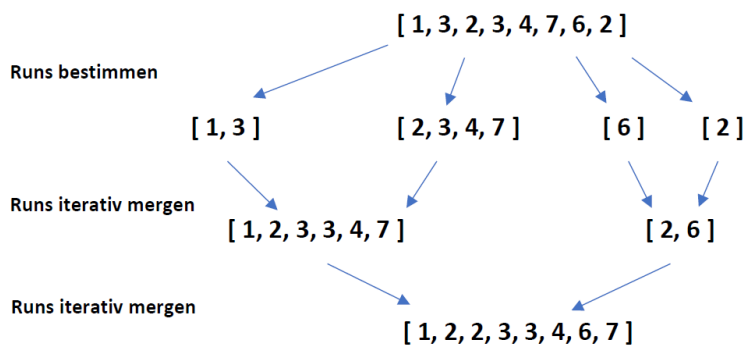


Abbildung 2: Natürlicher Mergesort

- Quicksort

Implementieren Sie das Quicksort-Verfahren mit dem Partition-Algorithmus, der in der Vorlesung vorgestellt wurde (s. Folien 20–24 von ADS_K6_Sortieralgorithmen_TeilII.pdf).

- Hybrider Quicksort

Evaluieren Sie den Cross Over Point ab welchem n_0 Insertion Sort schneller sortiert als Quicksort. Verwenden Sie dann für alle Teilfolgen, die weniger als n_0 Elemente haben Insertion Sort zum sortieren und für alle Teilfolge, die gleich oder mehr Elemente als n_0 haben

den Quicksort-Algorithmus.

Optimieren Sie den Quicksort, indem Sie das Pivot-Element nach dem vorgestellten Verfahren nach Folien 27 und 28 optimieren. Beachten Sie dabei, dass Sie am Ende noch ein $\text{swap}(a[\text{center}], a[\text{right}])$ durchführen müssen, damit das Pivot-Element das letzte Element der Teilfolge ist.

- Shellsort

Shellsort mit der **Hibbard Folge** ($H_i = 2H_{i-1} + 1$) im Vergleich zu

Shellsort mit der **Abstandsfolge** ($H_i = 3H_{i-1} + 1$)

2. Erstellen bzw. vervollständigen Sie das Hauptprogramm. Im Hauptprogramm sollen die zu messenden Sortieralgorithmen mit einer entsprechenden Problemgröße n aufgerufen und die Ergebnisse der Zeitenmessungen in Textdateien geschrieben werden.

3. Messen Sie anschließend die Ausführungszeiten in Abhängigkeit der Problemgröße n für:

- Heapsort, $n = 1000 : 1000 : 1000000$
- Mergesort, $n = 1000 : 1000 : 1000000$
- Quicksort, $n = 1000 : 1000 : 1000000$
- Hybrider Quicksort, $n = 1000 : 1000 : 1000000$
- Shellsort, Folge 1, $n = 1000 : 1000 : 1000000$
- Shellsort, Folge 2, $n = 1000 : 1000 : 1000000$

Wobei $n = 1000 : 1000 : 1000000$ z.B. bedeutet, dass Sie bei der Problemgröße $n = 1000$ beginnen sollen und die Problemgröße in jedem Schritt um 1000 erhöhen bis Sie bei 1000000 angekommen sind. Nach jedem Schritt wird die Ausführungszeit für diese Problemgröße in eine Textdatei geschrieben. Messen Sie **nicht** die Gesamtlaufzeit! Initialisieren Sie Ihre Datenstrukturen vor jeder Messung neu mit Zufallszahlen (Integer).

Wiederholen Sie pro n die Messungen 10 mal pro Algorithmus und berechnen daraus den Mittelwert und **speichern Sie nur die mittlere Laufzeit als Messergebnis ab**.

4. Stellen Sie ihre Messergebnisse unter Zuhilfennahme von MATLAB, Octave oder GNU PLOT grafisch dar (Beispiele: siehe 4.3.5). Entsprechen die Messergebnisse den Erwartungen (z.B. bzgl. O-Notation)? Achten Sie bei den Plots auf aussagekräftige Achsenbeschriftungen und eine vernünftige Legende.

5. Beachten Sie unbedingt die Lösungshinweise (4.3) und **planen Sie genügend Zeit für die Messungen ein**.

6. Alle Beispiele (Textausgaben, Codevorlagen, Plots,...) dienen der Illustration und dürfen gerne entsprechend Ihren eigenen Vorstellungen angepasst werden.

4.2 Teilaufgabe 2

Implementieren Sie eine Hash-Tabelle als Array (ohne Re-Hashing), in dem Sie die Vorlage `hashtable.h` bzw. `hashtable.cpp` vervollständigen. Implementieren Sie folgende Features:

1. Konstruktor:

Beim Erzeugen einer Klasseninstanz soll ein entsprechend dem übergebenen Parameter dimensionierter `vector<int>` dynamisch auf dem Heap alloziert werden. Initial erhalten alle Werte des Vektors den Wert -1 . Die Größe der Hashtabelle wird mit dem übergebenen Wert initialisiert, der Kollisionszähler und die Anzahl der gespeicherten Elemente sollten mit 0 initialisiert werden. Der Konstruktor initialisiert auch den Belegungsfaktor mit 0.6 und die Hashmethode mit dem Verfahren der quadratischen Sondierung.

2. Destruktor:

Stellen Sie sicher, dass aller zur Laufzeit dynamisch allozierter Speicher bei Zerstörung des Objektes wieder freigegeben wird.

3. Berechnung des Hashindex:

Implementieren Sie die Methode `HashTable::hashValue(int item)`, die den Hash-Index $h_i(x)$ berechnet. Übergeben Sie der Methode den Schlüssel. Tritt eine Kollision auf, so soll der Kollisionszähler erhöht werden. Zur Kollisionsvermeidung sollen die folgenden Kollisionsstrategien verwendet werden, wobei M die Tabellengröße ist:

- Lineares Sondieren: $h_i(x) = (x + i) \% M$
- Quadratisches Sondieren: $h_i(x) = (x + i * i) \% M$
- Doppeltes Hashing: $h_i(x) = (x + i * (R - x \% R)) \% M$, wobei $R < M$ die nächst kleinere Primzahl zu M ist.

Ist der Hashtabellenplatz frei, wird das Element x in der Hashtabelle an der Stelle $h_i(x)$ gespeichert. Primär soll die Strategie fürs Quadratische Sondieren im Praktikum eingesetzt werden. (Unit-Tests). Die andere Sondierungs-Strategien sollen im Praktikum implementiert und gezeigt werden.

4. Rehashing:

Wenn der Belegungsfaktor der Hashtabelle einen Schwellenwert überschritten hat (hier: 0.6, dh. die Tabelle ist zu 60% gefüllt), soll ein automatisches Rehashing durchgeführt werden. Dazu soll eine neue Hashtabelle erzeugt werden mit der Größe $M_{neu} > 2 * M_{alt}$ und M_{neu} sei eine Primzahl. Es soll eine maximale Tabellengröße von 1000 angenommen werden. Sie können somit alle Primzahlen bis 1000 in einem Vektor vorab initialisieren. Alle Werte von der alten Hashtabelle müssen mit der neuen Hashfunktion (die ja von der Tabellengröße abhängt) übertragen werden. Der Speicher der alten Hashtabelle soll wieder frei gegeben werden.

5. Einfügen in die Hashtabelle:

Implementieren Sie die Funktion `HashTable::insert(int item)` zum Einfügen von Elementen.

Prüfen Sie vor dem Einfügen ob die Tabelle theoretisch zu voll wird. Ist dies der Fall soll vor dem Einfügen ein Rehashing durchgeführt werden. Berechnen Sie nun den Hashindex mit der gewählten Hashmethode (dies soll ein Attribut der Hashtabelle sein). Bei erfolgreichem Einfügen soll der Zähler für die Anzahl der Elemente erhöht werden.

6. Programmieren Sie ein Hauptprogramm main, in dem man die Kollisionsstrategie sowie den Belegungsfaktor auswählen kann und anschließend Elemente in die Hash-Tabelle einfügen kann. Geben Sie den Hash-Index für das eingefügte Element aus.
7. Nachdem die Unittests erfolgreich durchgelaufen sind, erzeugen Sie in Ihrem Hauptprogramm eine Hashtabelle der Größe 1000 und fügen Sie automatisch 200 Zufallszahlen ein, die im Wertebereich von 1000 bis 1500 liegen und geben Sie die Anzahl der Kollisionen auf der Konsole aus.

4.3 Lösungshinweise

4.3.1 Allgemeine Hinweise zur Zeitmessung

- Kompilieren Sie Ihr Projekt vor der Messung unbedingt im *RELEASE* Modus und verwenden Sie das Compilerflag */Ox* in Visual Studio oder *-O3* bei Verwendung des gcc, um eine maximale Performance zu erhalten.
- Deaktivieren Sie alle unnötigen Konsolenausgaben für die Messungen, da diese sehr viel Zeit kosten.
- Beenden Sie alle anderen Anwendungen (Browser, E-Mail-Client, Antivirus, etc.....), da diese das Ergebnis ebenfalls drastisch verfälschen können!
- Achten Sie ebenfalls darauf, dass Sie nur die reine Sortier-/Ausführungszeit messen, und nicht zB das Erzeugen der Zufallszahlen mitmessen

4.3.2 OpenMP Compiler-Einstellungen

Hinweis: Wenn Sie eine aktuelle Version von Visual Studio verwenden, sollte es im Normalfall nicht nötig sein, dass Sie OpenMP in den Compilereinstellungen aktivieren müssen.

Damit Sie OpenMP zur Zeitmessung nutzen können sind entsprechende Compilereinstellungen erforderlich. Falls Ihre IDE nicht hier gelistet ist müssen Sie die für Sie erforderlichen Einstellungen per Selbstrecherche herausfinden.

Linux g++ Aktivierung von OpenMP unter Linux durch Compilerflag **-fopenmp** und linken gegen OpenMP Library **-lgomp**. Beispielaufruf für g++:

```
1 g++ main.cpp sorting.cpp hashtable.cpp unit_tests.cpp -o praktikum4 -fopenmp -lgomp
```

Windows Visual Studio Aktivierung von OpenMP unter Visual Studio:

<https://msdn.microsoft.com/de-de/library/fw509c3b.aspx>

- 1 Öffnen Sie das Dialogfeld Eigenschaftenseiten des Projekts.
- 2 Erweitern Sie den Knoten Konfigurationseigenschaften.
- 3 Erweitern Sie den Knoten C/C++.
- 4 Wählen Sie die Eigenschaftenseite Sprache aus.
- 5 Ändern Sie die Eigenschaft OpenMP-Unterstützung.

4.3.3 Format der TXT-Dateien

Erzeugen Sie für jeden gemessenen Algorithmus eine eigene Textdatei. Die Messungen sollten tabulatorgetrennt spaltenweise abgespeichert werden, damit sie möglichst einfach geplottet werden können. Ein Auszug aus der Datei "quicksort.txt" könnte dann beispielsweise wie folgt aussehen:

- 1.Spalte: Problemgröße n
- 2.Spalte: Berechnungsdauer in s

Beispiel:

```
1 ...  
2 986000 6.3498632997e-02  
3 987000 6.3852430001e-02  
4 988000 6.3209023996e-02  
5 ...
```

4.3.4 Beispiele zum Plotten mit MATLAB / GNU PLOT / Octave

- MATLAB

Erzeugen Sie im selben Ordner, indem sich Ihre Messungen befinden, eine M-Skript-Datei mit einem beliebigen Namen, z.B. *make_plots.m*:

```
1 clear;clc;close all;  
2  
3 fid=fopen('quicksort.txt');  
4 data=textscan(fid,'%d %f');  
5 fclose(fid);  
6 x=data{1};  
7 quicksort_y=data{2};  
8  
9 fid=fopen('mergesort.txt');  
10 data=textscan(fid,'%d %f');  
11 fclose(fid);  
12 mergesort_y=data{2};  
13  
14 fid=fopen('heapsort.txt');  
15 data=textscan(fid,'%d %f');  
16 fclose(fid);  
17 heapsort_y=data{2};  
18  
19 fid=fopen('shellsort.txt');
```

```
20 data=textscan(fid,'%d %f');
21 fclose(fid);
22 shellsort_y=data{2};
23
24
25 figure;
26 title('sorting algorithms');
27 xlabel('n [-]');
28 ylabel('t [s]');
29 hold on;
30 plot(x,quicksort_y);
31 plot(x,mergesort_y);
32 plot(x,heapsort_y);
33 plot(x,shellsort_y);
34 legend('quicksort','mergesort','heapsort','shellsort','Location','northwest');
35 hold off;
```

Führen Sie das Skript anschließend aus:

```
1 >> make_plots
```

- GNUPLOT

Erzeugen Sie im selben Ordner, indem sich Ihre Messungen befinden, eine Datei mit einem beliebigen Namen, z.B. *plots.gnu*:

```
1 reset
2 set autoscale x
3 set autoscale y
4 set xlabel "n [-]"
5 set ylabel "t [s]"
6 set key top left
7
8 plot \
9 "quicksort.txt" with linespoints title 'Quicksort',\
10 "mergesort.txt" with linespoints title 'Mergesort',\
11 "shellsort.txt" with linespoints title 'Shellsort',\
12 "heapsort.txt" with linespoints title 'Heapsort',\
```

Starten Sie nun Gnuplot, wechseln Sie in das korrekte Verzeichnis, und führen Sie das Skript wie folgt aus:

```
1 $ cd 'pfad-zum-gnuplot-skript'
2 $ load "plots.gnu"
```


Weiterführende Befehle zu GNUPLOT findet man z.B. hier:
http://gnuplot.sourceforge.net/docs_4.0/gpcard.pdf

4.3.5 Beispielplots

Die Plots sollten, natürlich in Abhängigkeit der verwendeten CPU, in etwa so aussehen (in den Abbildungen wurden die Legenden anonymisiert um die Ergebnisse nicht vorweg zu nehmen):

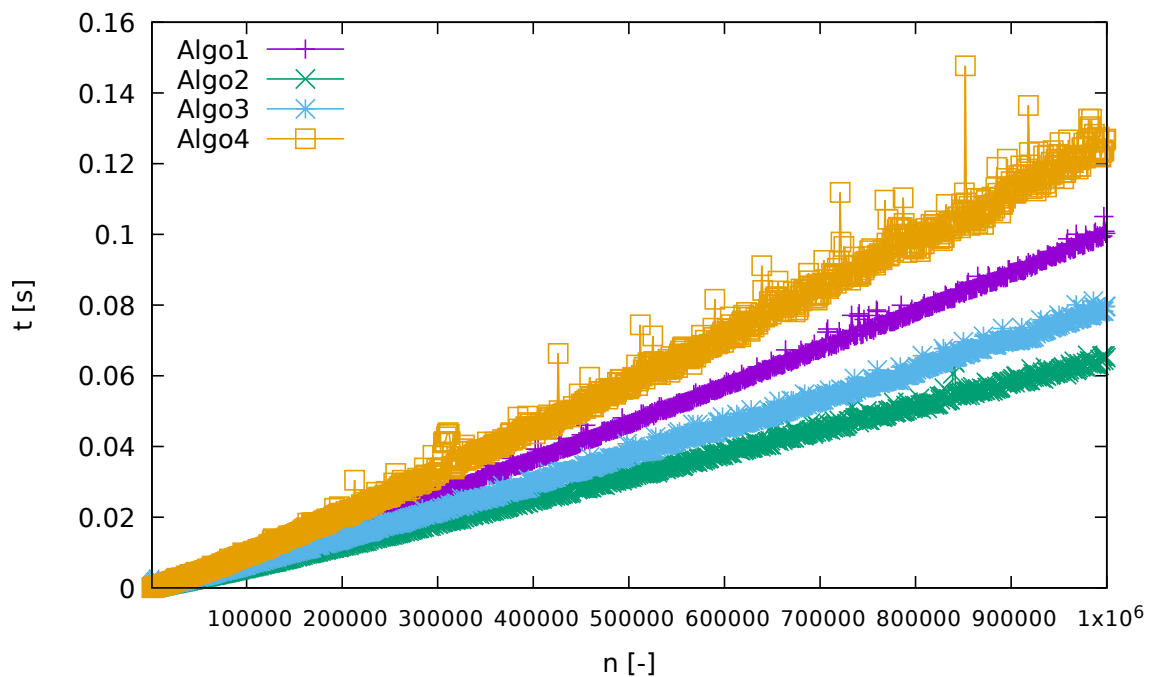


Abbildung 3: Laufzeitvergleich der Sortieralgorithmen