# 2 Praktikum 2: 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.

## 2.1 Teilaufgabe 1

Für alle Aufgaben gilt, dass Sie hierzu die Vorlage aus ILIAS benutzen können, in der bereits der Programmrumpf sowie ein Benchmarkaufruf für einen Sortieralgorithmus exemplarisch implementiert sind. 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
   - Natürlicher Mergesort
     Der Unterschied zwischen Mergesort und natürlichem Mergesort besteht im ersten Schritt- bei der rekursiven Aufteilung in verschiedene Teillisten. Der normale Mergesort unterteilt die *gesamte* Zahlenmenge bis Teillisten der Größe 1 und setzt diese dann rekursiv wieder zusammen, während dabei sortierte 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 auf ähnliche Weise wieder zusammen wie der normale Mergesort. In 1 und 3 ist der Unterschied 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).**
   - Quicksort
     Evaluieren Sie ihren persönlichen Cross-Over Point ab welchem $n_0$ Insertion Sort schneller sortiert als Quicksort. Stellen Sie ihre Laufzeit-Messergebnisse zu Quicksort und Insertion

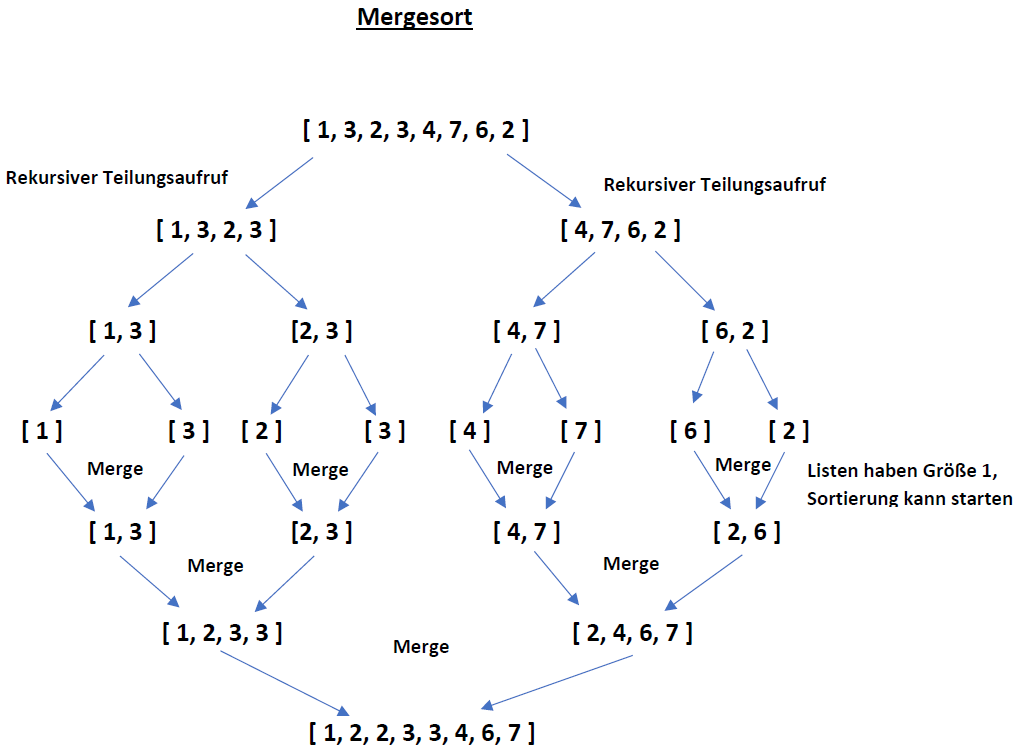


Abbildung 1: Mergesort

```
Function MERGE_SORT(A,B,left,right)
if left < right then
middle <- (left + right)/2
MERGE_SORT(A,B,left,middle)
MERGE_SORT(A,B,middle + 1,right)
MERGE(A,B,left,middle + 1, right)
endif
end
```

Abbildung 2: Pseudocode Mergesort

Sort in einer Grafik dar. Die x-Achse skalieren Sie bitte mit $n \in [1, 2 * n_0]$.
Schreiben Sie eine neue Methode, in der für alle Teilfolgen, die weniger als $n_0$ Elemente haben, Insertion Sort verwendet wird und für alle Teilfolgen, die gleich oder mehr Elemente als $n_0$ haben, der Quicksort-Algorithmus angewendet wird. Verwenden Sie für Quicksort die optimierte Version, dabei ist das Pivot-Element der Median vom 1., mittleren und letztem Element der Teilfolge.

- Shellsort mit der **Hibbard Folge** ($H_i = 2H_{i-1} + 1$)
- Shellsort mit der **Abstandsfolge** ($H_i = 3H_{i-1} + 1$)
- Freiwillig: Eigene Ideen zu Sortieralgorithmen als Vergleich.

2. Erstellen bzw. vervollständigen Sie das Hauptprogramm. Im Hauptprogramm sollen die zu messenden Sortieralgorithmen mit einer entsprechenden Problemgröße $n$ aufgerufen und die Ergeb-

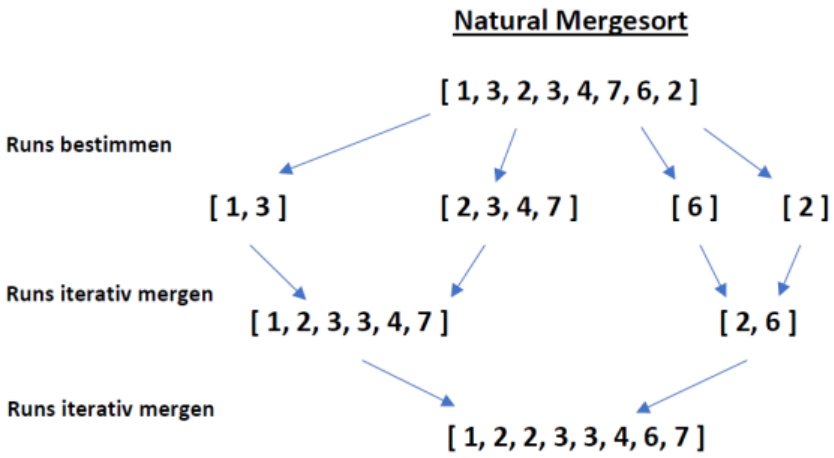


Abbildung 3: Natürlicher Mergesort

nisse der Zeitmessungen 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, $n = 1000 : 1000 : 1000000$
   - Shellsort, $n = 1000 : 1000 : 1000000$

   wobei $n = 1000 : 1000 : 1000000$ bedeutet, dass Sie bei der Problemgröße $n = 1000$ beginnen 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 ihre Messung 10-mal für das gleiche $n$ und berechnen Sie den Mittelwert aus den 10 Laufzeitmessungen.

4. Stellen Sie ihre Messergebnisse unter Zuhilfennahme von MATLAB, Octave oder GNUPLOT grafisch dar (Beispiele: siehe Abschnitt 2.3.4). Entsprechen die Messergebnisse den Erwartungen (z.B. bzgl. O-Notation)? Achten Sie bei den Plots auf aussagekräftige Achsenbeschriftungen und eine vernünftige Legende. Integrieren Sie ggf. eine mat. Fkt., die ihre Laufzeit möglichst gut approximiert.

5. Beachten Sie unbedingt die Lösungshinweise (s. Abschnitt 2.3) 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.

## 2.2 Teilaufgabe 2

Implementieren Sie eine Hash-Tabelle als Array, 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 allokiert und mit dem Wert $-1$" initialisiert werden. Die Größe der Hashtabelle wird mit dem übergebenen Wert initialisiert, der Kollisionszähler und die Anzahl der gespeicherten Elemente sollten mit 0 initialisiert werden.

2. Destruktor:
   Stellen Sie sicher, dass jeglicher zur Laufzeit dynamisch allokierter Speicher bei Löschen des Objektes wieder freigegeben wird.

3. Einfügen in die Hashtabelle:
   Implementieren Sie die Funktion `HashTable::insert(int item)` zum Einfügen eines neuen Elementes mit dem Wert item. Prüfen Sie *vor dem Einfügen*,ob die Tabelle theoretisch zu voll wird d.h. der Belegungsfaktor erreicht wird. Ist dies der Fall soll vor dem Einfügen ein Rehashing durchgeführt werden. Bei erfolgreichem Einfügen soll der Zähler für die Anzahl der Elemente erhöht werden.

4. 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 soll zwischen linearem und quadratischem Sondieren, sowie doppeltem Hashing als Strategie gewählt werden können. Die auszuführende Methode wird im Konstruktor bei der Erstellung der Hashtabelle mit der Variable `m_sondierMethode` vorgegeben. ($M$ ist die Größe der Hashtabelle):

   Lineares Sondieren:
   $$h_i(x) = (x + i)\%M \tag{1}$$
   Quadratisches Sondieren:
   $$h_i(x) = (x + i^2)\%M \tag{2}$$
   Doppeltes Hashing:
   $$h_i(x) = (x + i(R - x\%R))\%M \tag{3}$$

5. Rehashing:
   Falls die Hash-Tabelle einen Belegungsfaktor ß $> 0.6$ hat, soll **vor dem Einfügen** ein automatisches Rehashing durchgeführt werden. Dazu soll eine neue Hashtabelle erzeugt werden mit der Größe $M_{neu} > 2 * M_{alt}$ und $M_{neu}$ sei die nächst größere 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 bzw. sequenziell von der alten Hashtabelle ausgelesen und in die neue Hashtabelle eingefügt werden. Der Speicher der alten Hashtabelle soll wieder frei gegeben werden.

6. Nachdem die Unittests erfolgreich durchgelaufen sind, erzeugen Sie in Ihrem Hauptprogramm eine Hashtabelle der Größe 1000 mit Kollisionsstrategie ihrer Wahl. 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.

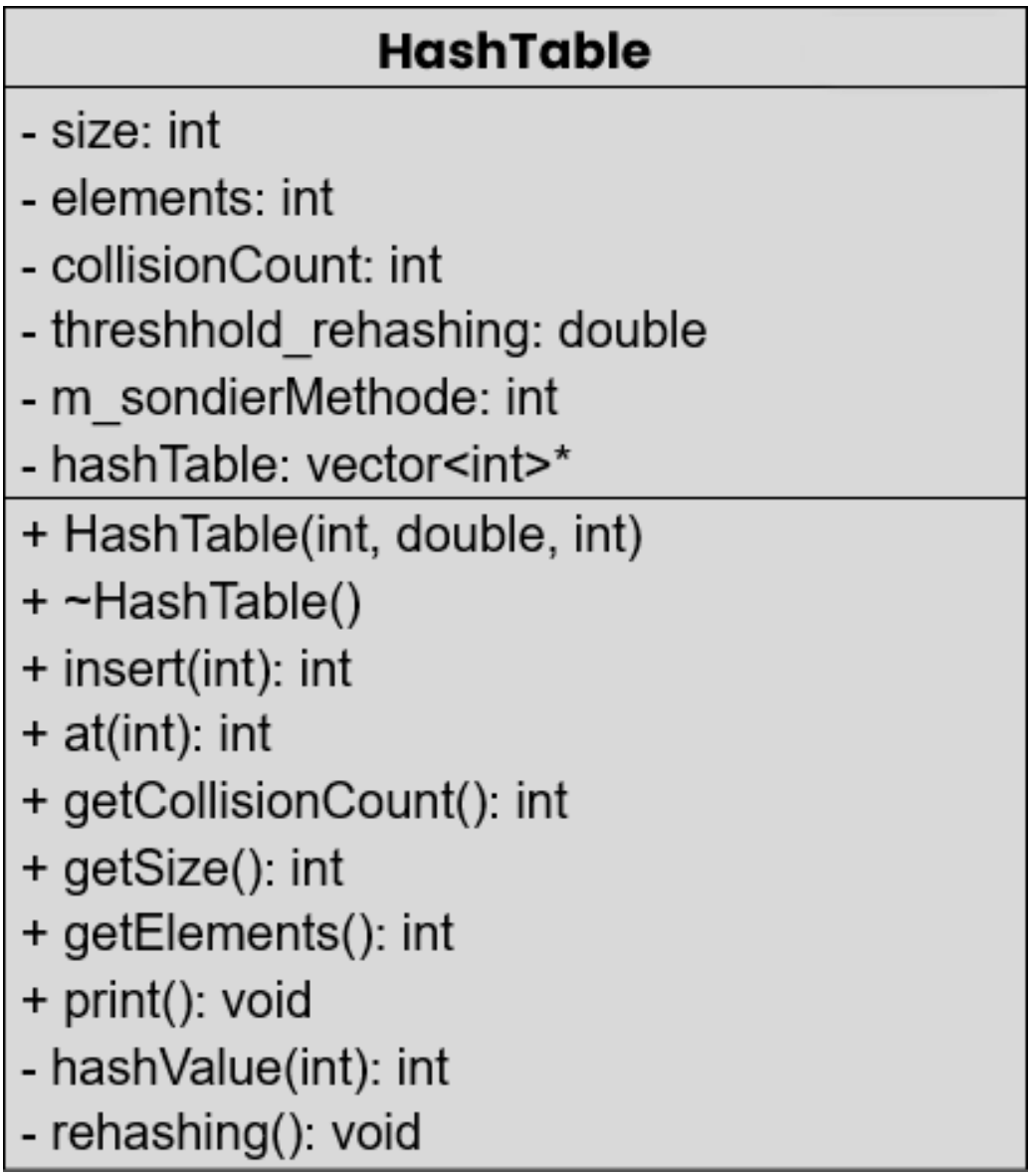


Abbildung 4: UML-Klassendiagramm hashtable.h

## 2.3 Lösungshinweise

### 2.3.1 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.

### 2.3.2 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:

```
...
986000   6.3498632997e-02
987000   6.3852430001e-02
988000   6.3209023996e-02
...
```

### 2.3.3 Beispiele zum Plotten mit MATLAB / GNUPLOT / 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*:

```
clear;clc;close all;

fid=fopen('quicksort.txt');
```

```
data=textscan(fid,'%d %f');
fclose(fid);
x=data{1};
quicksort_y=data{2};

fid=fopen('mergesort.txt');
data=textscan(fid,'%d %f')
fclose(fid);
mergesort_y=data{2};

fid=fopen('heapsort.txt');
data=textscan(fid,'%d %f');
fclose(fid);
heapsort_y=data{2};

fid=fopen('shellsort.txt');
data=textscan(fid,'%d %f');
fclose(fid);
shellsort_y=data{2};


figure;
title('sorting algorithms');
xlabel('n [-]');
ylabel('t [s]');
hold on;
plot(x,quicksort_y);
plot(x,mergesort_y);
plot(x,heapsort_y);
plot(x,shellsort_y);
legend('quicksort','mergesort','heapsort','shellsort','Location','northwest');
hold off;
```

Führen Sie das Skript anschließend aus:

```
>> make_plots
```

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

```
reset
set autoscale x
```

```
set autoscale y
set xlabel "n [-]"
set ylabel "t [s]"
set key top left

plot \
"quicksort.txt" with linespoints title 'Quicksort',\
"mergesort.txt" with linespoints title 'Mergesort',\
"shellsort.txt" with linespoints title 'Shellsort',\
"heapsort.txt" with linespoints title 'Heapsort',\
```

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

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

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

### 2.3.4 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):

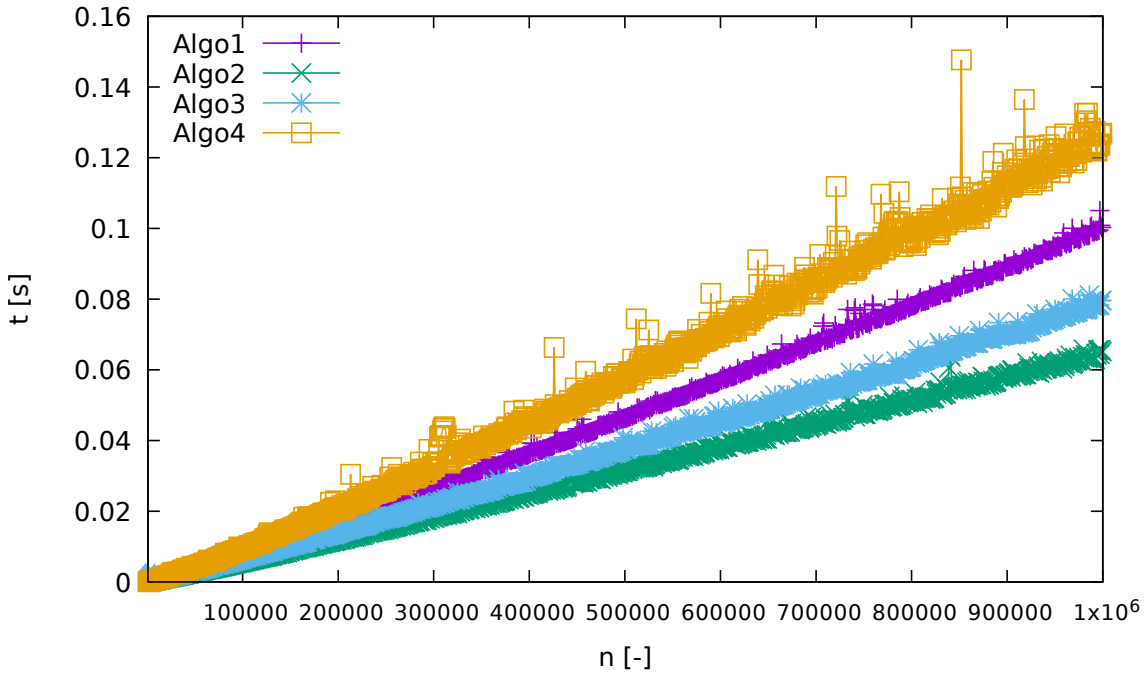


Abbildung 5: Laufzeitvergleich der Sortieralgorithmen