TU Ilmenau, Fakultät für Informatik und Automatisierung
FG Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. M. Dietzfelbinger, Philipp Schlag, Stefan Walzer, Alexander Jeuk
https://www.tu-ilmenau.de/iti/lehre/lehre-ss-2020/aud/

TECHNISCHE UNIVERSITÄT
**ILMENAU**

# Algorithmen und Datenstrukturen SS 2020 – Praktikum 4

Zur Bearbeitung zwischen 29. Juni - 3. Juli (KW 27)

Ziel dieses Praktikums ist die Implementierung von Quicksort. Wiederholen Sie daher sorgfältig Kapitel 6 der Vorlesung, insbesondere die Algorithmen `partition` und `r_qsort` auf Folien 30 und 33 (Druckfolien).
Die Eingabedatei enthält eine oder mehrere Zeilen. Jede Zeile enthält eine Zahl $n \in \{1, \ldots, 10^6\}$ sowie $n$ Strings über dem Alphabet $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$. Diese Strings sind lexikographisch aufsteigend sortiert auszugeben.
**Bemerkung:** Die entsprechende Ordnungsrelation auf dem Datentyp `string` ist in C++ schon als Operator „<“ verfügbar. Es gibt insofern keinen wesentlichen Unterschied zur Sortierung von Zahlen.

Listing 1: `sample.in`

```
10 im soldat project hin scheu es die dein recht ins
10 kommt meiner meinen seis und schon goetter hundert erhalten mephistopheles
10 bunten so project donnert heitrer satan bessre entfaltet fort tun
10 ueber schlaf wir den ein knabe reiche mit freuden lebt
10 kraeften helm by bueckt explanation wo und zu vernunft auch
```

Listing 2: `sample.exp`

```
dein die es hin im ins project recht scheu soldat
erhalten goetter hundert kommt meinen meiner mephistopheles schon seis und
bessre bunten donnert entfaltet fort heitrer project satan so tun
den ein freuden knabe lebt mit reiche schlaf ueber wir
auch bueckt by explanation helm kraeften und vernunft wo zu
```

## **Aufgabe 1** (Einfacher Quicksort)

Implementieren Sie den Quicksort-Algorithmus nach Vorbild der Vorlesung und reichen Sie Ihre Lösung für die Aufgabe `p4-a1-qs` ein. Verwenden Sie stets das erste Element der zu sortierenden Teilfolge als Pivotelement.
**Hinweise:**

- Die Vorlage `sort.cpp` suggeriert, dass `r_qsort(l,r)` das Teilarray `A[l..r-1]` sortieren soll. In der Vorlesung war dagegen die rechte Grenze $r$ inklusive. Entscheiden Sie sich für eine Variante und verwenden Sie diese konsequent.
- Für diese Aufgabe ist es völlig in Ordnung, wenn Ihr Array (vom Typ `vector<string>`) eine globale Variable ist, die dann nicht als Parameter an die Quicksort-Methode übergeben werden muss. Wenn Ihnen das unelegant erscheint, können Sie als Signatur Ihrer Methode folgendes wählen:

  ```
  void r_qsort(vector<string> &A, int l, int r);
  ```

  Das `&` bewirkt, dass `A` nicht eine *Kopie* des Arguments des Aufrufers ist, sondern dass das Array des Aufrufers als *Referenz* übergeben wird. Das Argument auf Seiten des Aufrufers und der Parameter `A` innerhalb des `r_qsort` Aufrufs sind dann *dasselbe Array*. Nur so kann `r_qsort` das Array des Aufrufers verändern.

- Um die Inhalte zweier Variablen `s` und `t` zu vertauschen, können Sie die Funktion `swap(s,t)` verwenden (`#include<utility>`). Das ist nicht nur übersichtlicher sondern auch effizienter als eine Befehlsfolge wie „`string tmp; tmp = s, s = t, t = tmp;`“.

## **Aufgabe 2** (Robuster Quicksort)

Verbessern Sie Ihren Algorithmus aus Aufgabe 1 im Sinne der folgenden beiden Teilaufgaben. Reichen Sie Ihr fertiges Programm dann als `p4-a2-rqs` ein.

**(a)** Wenn stets das erste Element der zu sortierenden Teilfolge als Pivot gewählt wird, hat Quicksort quadratische Laufzeit bei aufsteigend sortierten Eingabefolgen (`ascending.in`).

Randomisieren Sie daher Ihren Algorithmus, indem Sie das Pivotelement zufällig aus allen Elementen der zu sortierenden Teilfolge auswählen. Verwenden Sie dazu die Klassen `mt19937` (Pseudozufallszahlengenerator) sowie `uniform_int_distribution` aus der Standardbibliothek (`#include<random>`). Betrachten und adaptieren Sie dazu den Code aus folgendem Beispiel:

```
1  #include<random>
2  /* Pseudozufallszahlengenerator. Muss globale Variable sein */
3  mt19937 Zufallsquelle;
4  /* Gibt eine gleichverteilt zufaellige Zahl zwischen 1 und 6 zurueck */
5  int wuerfel() {
6      /* Definiere Uniforme Verteilung auf {1,2,3,4,5,6} */
7      uniform_int_distribution<int> verteilung(1,6);
8      /* Ziehe Zufallszahl gemaess der Verteilung */
9      return verteilung(Zufallsquelle);
10 }
```

**(b)** Bei Eingabefolgen, die viele Kopien des selben Werts enthalten hilft auch Randomisierung nicht quadratische Laufzeiten zu vermeiden (siehe `spam.in` und `large.in`[1]).

Wiederholen Sie Aufgabe 2 von Übungsblatt 7 (Dutch National Flag Problem) und ggf. die ausführliche Musterlösung als Video und PDF. Implementieren Sie die Partitionsvariante „3-Wege-Partition“, die drei Bereiche erzeugt, mit Elementen die kleiner, gleich bzw. größer als das Pivotelement sind.

[1] Die Testdatei `large.in` ist durch 1000000-maliges Ziehen eines zufälligen Wortes aus Goethes „Faust” entstanden. Wörter wie „der“ oder „und“ kommen entsprechend sehr häufig vor.