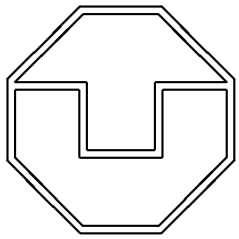
Technische Universität Dresden
Institut für Wissenschaftliches Rechnen

Wolfgang V. Walter



Programmieren 2

Aufgabenblatt 11 — Abgabetermin: 8.6.2018

Aufgabe 20: Zusammenmischen (Merge) mehrerer vorsortierter Dateien [16]

Es seien 99 bereits vorsortierte Textdateien `erfass1.dat`, `erfass2.dat`, ..., `erfass99.dat` gegeben. Diese bestehen aus in aufsteigender Reihenfolge sortierten, durch mindestens ein Leerzeichen oder ein Zeilenende voneinander getrennten ganzen Zahlen und sind i.a. unterschiedlich lang. Jede ganze Zahl kann sowohl mehrmals in derselben Datei als auch in mehreren Dateien vorkommen. Sie können solche Dateien selbst erzeugen, indem Sie das Programm `erfass.f95` übersetzen und ausführen.

Die Aufgabe besteht darin, die Inhalte der ersten n dieser Dateien insgesamt in aufsteigender Reihenfolge sortiert in eine Textdatei `ziel.dat` auszugeben (n ist eine einzulesende natürliche Zahl mit $1 \leq n \leq 99$). Dabei darf vorausgesetzt werden, dass keine der Eingabedateien leer ist und dass das Betriebssystem das gleichzeitige Öffnen von mindestens 100 Dateien erlaubt.

Schreiben Sie einen Fortran 95–Modul mit folgenden Bestandteilen:

a) Definition eines strukturierten Datentyps `filecomp`, bestehend aus einer ganzzahligen Komponente für die der jeweiligen Datei zugeordnete eindeutige Unitnummer (im Bereich $[21, 500]$) und einer ganzzahligen *Inhaltskomponente* für die zuletzt aus dieser Datei gelesene Zahl,

b) Überladung des Vergleichsoperators $\leq$ für Operanden des Typs `filecomp`, welcher einen Größenvergleich der *Inhaltskomponenten* der Operanden vornimmt,

c) Definition einer Subroutine `sort`, die mittels eines beliebigen Sortieralgorithmus (Empfehlung: Sortieren durch Einfügen = Insertion Sort) das als Argument übergebene eindimensionale Feld mit Elementen vom Typ `filecomp` (ein Teilfeld des im Hauptprogramm definierten *Arbeitsfelds*) in aufsteigender Reihenfolge (nach *Inhaltskomponente*) sortiert. Das Argument muss als Feld übernommener Gestalt vereinbart werden.

d) Definition einer Subroutine `insert`, die in das als erstes Argument übergebene eindimensionale Feld mit Elementen vom Typ `filecomp` (ein Teilfeld des im Hauptprogramm definierten *Arbeitsfelds*) ein als zweites Argument übergebenes Element vom Typ `filecomp` an der richtigen Stelle einfügt. Hierbei ist anzunehmen, dass das erste Element des übergebenen Felds nicht besetzt ist. Das Einfügen entspricht im wesentlichen einem Makroschritt des Insertion-Sort-Algorithmus, wobei natürlich der Wert der *Inhaltskomponente* des einzufügenden Elements die Position bestimmt. Die Einfügestelle soll mittels binärer Suche (Halbierungsverfahren) ermittelt werden.

Im Fortran 95–Hauptprogramm ist das *Arbeitsfeld*, auf dem alle Sortiervorgänge stattfinden, als eindimensionales, dynamisches Feld mit Elementen vom Typ `filecomp` zu vereinbaren. Es wird zunächst die Anzahl n (im Intervall $[1, 99]$) der zu berücksichtigenden Dateien eingelesen und dem *Arbeitsfeld* entsprechender Speicherplatz zugewiesen.

Der weitere Programmablauf gliedert sich dann in drei Phasen, die als Subroutinen in einem Modul definiert und im Hauptprogramm aufgerufen werden können.

`Phase 1`: Die Namen der n zu mischenden Dateien werden z.B. mittels *interner Ein/Ausgabe* (= *internal I/O*, d.h. mit einer Zeichenkettenvariable als Datei) erzeugt; sodann werden diese Dateien geöffnet. Den Dateien wird beim Öffnen eine eindeutige Unitnummer (im Bereich $[21, 500]$) zugeordnet. Die Zieldatei `ziel.dat` wird ebenfalls geöffnet.

`Phase 2`: Das *Arbeitsfeld* wird erstmalig mit Inhalten gefüllt. Dabei werden von jeder der ersten n Erfassungsdateien deren Unitnummer sowie aus der jeweiligen Datei das erste Element in das Feld eingetragen. Anschließend wird der so gefüllte Teil des *Arbeitsfelds* einmalig in aufsteigender Reihenfolge (nach *Inhaltskomponenten*) sortiert (mittels `sort`).

`Phase 3`: Der *aktive Indexbereich* des Arbeitsfelds ist anfänglich $(1 : n)$. Jedesmal, wenn eine Datei erschöpft ist, d.h. wenn alle ihre Elemente gelesen wurden, reduziert sich der *aktive Indexbereich* um einen Index, zunächst auf $(2 : n)$, dann auf $(3 : n)$, etc.

Der Algorithmus schreibt nun immer die *Inhaltskomponente* des ersten (kleinsten) Elements aus dem *aktiven Indexbereich* des Arbeitsfelds (immer an der Position mit kleinstem aktiven Index) in die Zieldatei `ziel.dat` und fügt sodann das nächste Element aus derselben Datei, aus der das gerade geschriebene Element stammte, mit Hilfe der Subroutine `insert` in den aktiven Bereich des Arbeitsfelds ein. So bleibt das Arbeitsfeld im *aktiven Indexbereich* immer in aufsteigender Reihenfolge sortiert. Falls eine Datei erschöpft ist und folglich kein Element mehr nachgeladen werden kann, erfolgt die oben beschriebene Reduktion des *aktiven Indexbereichs*.

Das Arbeitsfeld enthält in seinem *aktiven Indexbereich* grundsätzlich aus jeder der noch aktiven (noch nicht erschöpften) Dateien genau ein Element. Wenn die letzte Datei erschöpft ist, wird der *aktive Indexbereich* leer und das Programm wird ordnungsgemäß beendet.

Vergewissern Sie sich, dass die generierte Datei `ziel.dat` tatsächlich eine monoton (aber nicht streng monoton) wachsende Folge von Zahlen enthält — am besten durch nochmaliges Lesen und Testen am Ende des Programms.

Aufgabe F11: Priorisierte Warteschlange (fakultativ)

Bearbeiten Sie die Teilaufgaben c) und d) der vorhergehenden Aufgabe mit Hilfe einer Priority-Queue (als Heap implementierte Warteschlange), um die Effizienz des Algorithmus zu steigern. Hierzu sind in c) mittels `Build-Heap` ein Heap zu erzeugen und in d) durch Setzen des ersten Elements (Wurzel des Heaps) auf den einzufügenden Wert und durch `Heapify` an der Wurzel jeweils ein neues Element einzufügen.

Wichtig: Beachten Sie, dass hier der Heap entgegen der üblichen Ordnung an der **Wurzel** das **kleinste** Element enthält!

Entspricht der Laufzeitgewinn in etwa Ihren Erwartungen?