Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 2
Sommersemester 2020
Übungen zur Vorlesung: Ausgabe am 16.06.2020

Aufgabe 7.1 (7 Punkte)

In dieser Aufgabe geht es um die Implementierung noch fehlender Teile eines Programms `sort_simple_mn.x`, das einen Teil der Funktionalität des Kommandozeilen-Werkzeugs `sort` implementiert.

Beim Aufruf erhält das Programm den Namen einer Datei sowie Optionen, durch die die Art der Sortierung der Zeilen und ihre Reihenfolge in der Ausgabe beeinflusst wird. Der Inhalt der übergebenen Datei wird natürlich nicht verändert.

Im Standardfall werden die Zeilen der Datei in aufsteigender Reihenfolge lexikographisch (also entsprechend der Ordnung wie im Telefonbuch) sortiert. Optional kann auch numerisch sortiert werden (Option `-n`), was bei Zeilen mit Zahlenwerten sinnvoll ist.

Für beide Formen der Sortierung kann die Reihenfolge der Ausgabe umgekehrt werden (Option `-r`), also absteigend sortiert werden.

Diese Aufgabe besteht aus mehreren Teilen.

Teil 1

Benennen Sie die Datei `pfn_line_store_template.c` um in `pfn_line_store.c`. Diese Datei erweitern Sie um eine Funktion

```
void pfn_line_store_sort(PfNLineStore *pfn_line_store,CompareFunc compar)
```

Diese soll mit Hilfe der C-Bibliotheksfunktion `qsort` die Zeilen sortieren, die in `pfn_line_store` gespeichert sind, und zwar unter Verwendung des Funktionszeigers `compar`, der auf die Vergleichsfunktion zeigt.

Dabei ist der Typ `CompareFunc` in `pfn_line_store.h` wie folgt deklariert:

```
typedef int (*CompareFunc)(const void *,const void *);
```

Teil 2

Der zweite Teil besteht darin, in der Datei `sort_simple.c` eine Funktion

```
void sort_simple(PfNLineStore *line_store,bool numerical_order,
                 bool reverse_order)
```

zu implementieren, die entsprechend der Parameter `numerical_order` und `reverse_order` die passende Vergleichsfunktion auswählt und diese dann beim Aufruf von `pfn_line_store_sort` verwendet, um die Zeilen aus `line_store` zu sortieren. Selbstverständlich müssen Sie vorher die vier Vergleichsfunktionen, nennen wir sie `lex_cmp`, `num_cmp`, `lex_reverse_cmp`, und `num_reverse_cmp`

implementieren. Dabei steht `lex` für die lexikographische Ordnung, `num` für die numerische Ordnung und `reverse` für die Umkehrung der Reihenfolge. Es ist sinnvoll, zunächst die ersten beiden Funktionen zu implementieren und sich dann zu überlegen, wie man diese für die beiden `_reverse`-Funktionen wiederverwenden kann.

Für die lexikographische Ordnung sollen Sie die Funktion `strcmp` verwenden. Da `PfNLine` der Basistyp des `lines`-Arrays in `PfNLineStore` ist, müssen Sie die **`void`**-Zeiger der `_cmp`-Funktion entsprechend casten, analog zur Funktion `double_cmp` aus der Vorlesung.

Für die numerische Sortierung müssen Sie testen, ob der entsprechende String einen numerischen Wert darstellt. Wenn das für beide Strings gilt, müssen Sie den Vergleich auf der Basis des numerischen Wertes durchführen. Verwenden Sie Variablen vom Typ **`double`** für die numerischen Werte. Es gelten die folgenden besonderen Regeln, wenn bei der numerischen Sortierung einer oder beide zu vergleichende Strings keine numerischen Werte darstellen:

- Falls beide Strings keine numerischen Werte darstellen, wird mit `strcmp` verglichen.
- Falls einer der beiden Strings einen numerischen Wert darstellt und der andere den leeren String, dann wird der leere String beim Vergleich wie der numerische Wert 0.0 behandelt.
- Ein String mit dem numerischen Wert 0.0 ist kleiner als jeder nicht leere String, der keinen numerischen Werte darstellt.
- Ein String mit einem numerischen Wert $\neq 0.0$ ist größer als jeder nicht leere String, der keinen numerischen Wert darstellt.

Teil 3

In den Materialien finden Sie eine Datei `sort_simple_mn_template.c`, die Sie bitte in `sort_simple_mn.c` umbenennen. Diese Datei enthält die `main`-Funktion. Sie müssen noch einen Optionsparser implementieren. Dieser besteht aus der Deklaration eines **`struct`**-Typs `Options` und den Funktionen `usage`, `options_new` und `option_delete`. Die auszugebenden `Usage`-Zeilen stehen in der Datei `usage.txt`. Die Bezeichner für die Komponenten der Struktur und die jeweiligen Typen ergeben sich aus der `main`-Funktion. Bei der Entwicklung des Optionsparsers orientieren Sie sich bitte am Optionsparser aus der Datei `pfn_line_store_mn.c` (siehe Aufgabe 5.1).

In der Funktion `main` wird der Optionsparser aufgerufen, die durch den Benutzer spezifizierten Dateien werden eingelesen und die entsprechende `PfNLineStore`-Struktur aufgebaut. Schließlich folgt der Aufruf von `sort_simple` und die Freigabe des Speichers.

Durch `make` können Sie Ihr Programm kompilieren. Durch `make test` verifizieren Sie die Korrektheit für einige Test-Dateien.

Nachdem Sie verifiziert haben, dass `make test` funktioniert, rufen Sie bitte `make test_large` auf. Beschreiben Sie in einem Satz, auf welche Art von Daten die Programme `sort_simple_mn.x` und `sort` angewendet werden. Notieren Sie die Laufzeiten der Programme (in Sekunden) und beschreiben Sie mögliche Gründe für die unterschiedlichen Laufzeiten. Es geht hier nicht um eine aysmptotische Analyse, sondern um konkrete Laufzeiten bei Ausführung des Programms auf Ihrem Rechner.

Punkte-Verteilung:

- 1 Punkt für `pfn_line_store_sort`,

- 1 Punkt insgesamt für die beiden Funktionen zur lexikographischen Sortierung,
- 2 Punkte für die Funktionen zur numerischen Sortierung
- 1 Punkt für den Optionsparser
- 1 Punkt für funktionierende Tests
- 1 Punkt für eine nachvollziehbare Auswertung und Erklärungen.

Zur Bearbeitung dieser Aufgabe ist es hilfreich, die Abschnitte 1-6, 8,9, 11, 14, 16, 21, 22 der Vorlesung (siehe Spalte *Nummer* in der Tabelle in `pfn2_vorlesung_2020.html`), zu kennen.

Aufgabe 7.2 (5 Punkte) In dieser Aufgabe geht es um die Lösung des Teilmengen-Summen-Problems. Dieses besteht darin, für eine nicht-leere Menge A von positiven ganzen Zahl und eine ganze Zahl s zu entscheiden, ob es eine Teilmenge $A' \subseteq A$ gibt, so dass die Summe aller Zahlen aus A' gleich s ist. Falls es eine solche Teilmenge A' gibt, sollen ihre Elemente in aufsteigender Reihenfolge ausgegeben werden. Jede Zahl aus A darf höchstens einmal in A' vorkommen.

Beispiel: Sei $A = \{3, 5, 7, 12\}$. Hier sind Lösungen des Teilmengen-Summen-Problems für einige Werte von s zwischen 15 und 23:

s	A'
$15 = 3 + 5 + 7$	
$16 = 5 + 11$	
$18 = 7 + 11$	
$19 = 3 + 5 + 11$	
$21 = 3 + 7 + 11$	
$23 = 5 + 7 + 11$	

Für $s \in \{17, 20, 22, 24, 25\}$ gibt es keine Lösung bzgl. A.

Nehmen wir an, dass die Menge A durch ein aufsteigend sortiertes Array a repräsentiert wird. Dann gibt es genau dann eine Lösung des Teilmengen-Summen-Problems für A und s, wenn $hs(a, |a|, s)$ den Wert true zurückliefert, wobei hs eine rekursive Funktion ist, die wie folgt definiert ist:

$$hs(a, i, t) = \begin{cases} \text{true} & \text{if } t = 0 \\ \text{false} & \text{else if } i = 0 \\ \text{true} & \text{else if } t \geq a[i-1] \text{ and } hs(a, i-1, t-a[i-1]) \\ hs(a, i-1, t) & \text{otherwise} \end{cases}$$

Hierbei steht hs für *has solutions*. Für die Implementierungsaufgaben ist es sicher hilfreich, sich zu überlegen, warum diese rekursive Definition sinnvoll ist.

In den Materialien zu dieser Aufgabe finden Sie u.a. die folgenden Dateien:

- Eine Headerdatei `subsetsum.h` mit den Vorwärtsdeklarationen der zu implementierenden Funktionen.
- Ein Optionsparser und das Hauptprogramm.
- Textdateien mit ganzen Zahlen und Dateien mit den erwarteten Ergebnissen für vorgegebene Werte von s.

Implementieren Sie in der Datei `subsetsum.c` eine Funktion

```
bool *subsetsum(const unsigned long *arr,unsigned long r,
                unsigned long s)
```

die selbst nicht rekursiv ist, aber durch Aufruf einer rekursiven Funktion nach der obigen rekursiven Definition das Teilmengen-Summen-Problems für A und s löst. Dabei wird A durch ein Array repräsentiert, auf das der Zeiger `arr` verweist und `r` ist die Anzahl der Elemente dieses Arrays. `subsetsum` soll `NULL` zurückliefern, wenn das Teilmengen-Summen-Problem für A und s nicht lösbar ist. Wenn es lösbar ist, soll ein Zeiger auf ein Array `mark` der Länge `r` vom Basistyp `bool` zurückgeliefert werden, so dass `mark[i]` genau dann `true` ist, wenn das i-te Element aus `arr` zur Lösung gehört. Um in der rekursiven Funktion die Werte in `mark` korrekt zu berechnen, muss die implizite Auswahl bzw. Nicht-Auswahl eines Elementes in der Rekursion durch Hinzufügen einer Anweisung `mark[i] = true` bzw. `mark[i] = false` protokolliert werden.

Beachten Sie, dass es i.A. mehrere Lösungen des Teilmengen-Summen-Problems für A und s gibt. Die für `subsetsum` verwendete Rekursion führt zu einer Lösung mit Elementen maximaler Größe (falls eine Lösung existiert).

Um den ersten Test durchzuführen, müssen Sie eine später zu entwickelnde Funktion zunächst wie folgt implementieren, damit es beim Linken keine Probleme gibt:

```
bool *subsetsum_memo(__attribute__ ((unused)) const unsigned long *arr,
                     __attribute__ ((unused)) unsigned long r,
                     __attribute__ ((unused)) unsigned long s)
{
  return NULL;
}
```

Durch `make` kompilieren Sie Ihr Programm zusammen mit dem Hauptprogramm. Durch `make test_r_small` und `make test_r_large` verifizieren Sie die Korrektheit für zwei Mengen von Zahlen und verschiedene Werte von s.

Es gibt noch einen weiteren Test `test_difficult` für $s = 10930$ und die 40 Zahlen aus der Datei `numbers40.txt`. Die Implementierung von `subsetsum` aus der Musterlösung benötigt 2 199 023 255 438 Funktionsaufrufe und 76 Minuten Laufzeit, um zu ermitteln, dass es keine Lösung gibt. Wenn Sie den einfachen rekursiven Algorithmus verwenden, dann wird Ihre Implementierung wahrscheinlich nicht viel schneller sein. Es ist daher sinnvoll, den rekursiven Algorithmus zu optimieren.

Implementieren Sie in der Datei `subsetsum.c` die Funktion `sumsetsum_memo`. In der bisherigen Version dieser Funktion soll im Funktionskopf jeweils `__attribute__ ((unused))` gestrichen werden. Diese Funktion kombiniert den rekursiven Algorithmus mit der Memoization-Technik, die Sie in der Vorlesung im Kontext der Berechnung von Binomialkoeffizienten kennen gelernt haben. Die Funktion selbst ist nicht rekursiv. Sie ruft aber eine rekursive Funktion auf.

Deklarieren Sie dazu einen Typ `DefinedValues`, der zwei boolsche Variablen `defined` und `has_sol` enthält. Letztere wird benutzt um den **`return`**-Wert eines rekursiven Aufrufs zu speichern. Die wesentlichen Parameter eines rekursiven Aufrufs sind i und t (siehe Definition von hs), so dass das Ergebnis eines rekursiven Aufrufs für die Parameter i und t in einer Matrix in Zeile i und Spalte t gespeichert wird. Der Maximalwert von i ist r und der Maximalwert von t ist s. Sie benötigen daher eine $(r+1) \times (s+1)$-Matrix M mit dem Basistyp `DefinedValues`.

Zum Allokieren der Matrix verwenden Sie die Makros aus `array2dim.h`. Stellen Sie sicher, dass alle `defined`-Werte in der Matrix mit `false` initialisiert sind. Wenn ein Wert beim Aufruf der

rekursiven Funktion vorher noch nicht berechnet wurde (`defined` ist `false`), wird er berechnet und das Ergebnis in der Matrix in `has_sol` gespeichert. Wenn ein Wert bereits einmal berechnet wurde, wird er aus der Matrix (d.h. dem `has_sol`-Wert) gelesen.

Durch `make test_m` verifizieren Sie, dass die Funktion `sumsetsum_memo` korrekt funktioniert und in sehr kurzer Zeit die richtigen Ergebnisse (auch für $s = 10930$) liefert.

Punkteverteilung:

- 1 Punkt für die Implementierung von `subsetsum`
- 3 Punkte für die Implementierung von `subsetsum_memo`
- 1 Punkt für funktionierende Tests

Zur Bearbeitung dieser Aufgabe ist es hilfreich, die Abschnitte 1-6, 8-11, 14, 16, 29 der Vorlesung (siehe Spalte *Nummer* in der Tabelle in `pfn2_vorlesung_2020.html`), zu kennen.

Bitte die Lösungen zu diesen Aufgaben bis zum 23.06.2020 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.