

Programmierung für Naturwissenschaften 2
Sommersemester 2021
Übungen zur Vorlesung: Ausgabe am 17.06.2021

Falls Sie auf einem Rechner mit macOS und einem Intel-Prozessor arbeiten, vereinfacht sich die Fehlersuche bei Speicherfehlern, wenn Sie den AdressSanitizer verwenden, siehe <https://en.wikipedia.org/wiki/AddressSanitizer>. Sie müssen dazu beim Kompilieren und Linken mit `gcc` oder `g++` zwei Optionen zusätzlich verwenden. Um die Nutzung zu vereinfachen, wurde das Makefile für Aufgabe 10.2 und 10.3 um eine `debug` Option erweitert. Wenn Sie `make debug=yes` aufrufen, dann werden die AdressSanitizer-Optionen verwendet. Wenn die AdressSanitizer-Bibliothek installiert wird, entsteht ein ausführbares Programm, das bei Speicherfehlern Hinweise auf die Programmzeilen liefert, an denen der Fehler aufgetreten ist.

Aufgabe 10.1 (2 Punkte)

Aus Zeitgründen können wir in diesem Semester in der Vorlesung das Thema *Case study in run time optimization* (siehe `C_slides.pdf`, 473-492) nicht behandeln. Da in diesen Folien Aspekte der Programmierung behandelt werden, die sonst nicht vorkommen, wird das Thema in dieser Peer Teaching Aufgabe behandelt.

In jeder Kleingruppe muss sich ein Studierender auf der Basis der oben genannten Folien auf das Thema vorbereiten und das erworbene Wissen an die anderen Mitglieder der Kleingruppe weitergeben. Bitte sprechen Sie sich in Ihrer Kleingruppe ab, wer diese Aufgabe übernimmt.

Die Dateien mit den entsprechenden Programmcodes finden Sie in den Materialien.

Nach der Übung dokumentiert jede Kleingruppe in der Datei `bearbeitung.txt` das Vorgehen bei der Erarbeitung des Themas und ggf. noch bestehende Verständnisfragen oder Hinweise zu Unklarheiten in den Folien. Willkommen sind natürlich auch Bemerkungen zur Lehrform selbst.

Aufgabe 10.2 (5 Punkte)

In den Materialien zu dieser Aufgabe finden Sie im Verzeichnis `Csources` in den Dateien `multiseq.c` und `multiseq.h` die C-Implementierung einer Klasse `Multiseq` zum Parsen von Dateien im Multi-Fasta Format und zur Verwaltung der darin enthaltenen Informationen. Dieses Format wird für biologische Sequenzen, insbesondere DNA- und Proteinsequenzen, verwendet. Für jede einzelne Sequenz gibt es eine eigene Kopfzeile, die mit dem Zeichen `>` beginnt. Darauf folgen dann beliebig viele Zeilen mit Zeichen aus dem DNA- oder Aminosäurealphabet. In den Materialien finden Sie mehrere Beispieldateien mit dem Suffix `.fna`. Einige Dateien sind allerdings nicht korrekt formatiert, was im C-Programm auch erkannt und mit einer entsprechenden Fehlermeldung quittiert wird.

Ihre Aufgabe besteht darin, in der Datei `multiseq.cpp` die Methoden der Klasse `Multiseq` in C++ zu implementieren. Dabei sollen Sie den C-Programmcode übertragen und an die Syntax einer C++-Klasse anpassen. Dabei können Sie einen großen Anteil des C-Codes übernehmen und müssen im Wesentlichen die Zugriffe auf den Zeiger `multiseq` durch entsprechende Syntax mit dem Schlüsselwort `this` ersetzen. Die Methodennamen der Klasse finden Sie in `multiseq.hpp`.

Außerdem sollen Sie in der Datei `multiseq_test.cpp` ein Testprogramm in C++ analog zu `Csources/multiseq_test.c` entwickeln. Ihre Implementierung soll die folgenden Bedingungen erfüllen:

- Die Fehlerbehandlung soll durch eine Ausnahmebehandlung erfolgen. D.h. es müssen jeweils die Aufrufe der Funktionen `error_msg_empty_sequence` und `error_msg_missing_header` einschließlich der folgenden `return NULL`-Anweisung (siehe `multiseq.c`) durch Aufrufe der entsprechenden Funktion mit dem Präfix `throw_` ersetzt werden. Diese Funktionen sind in der Datei `multiseq_template.cpp` implementiert. Diese Datei benennen Sie in `multiseq.cpp` um. Im Hauptprogramm müssen Sie eine mögliche Ausnahme, initiiert durch `throw`, mit einem `try`-Block und einem `catch (std::invalid_argument &msg)`-Block behandeln. Im letzteren Block wird auf die Fehlermeldung aus dem String-Objekt `msg` mit `msg.what()` zugegriffen. In den Vorlesungsfolien finden Sie ein Beispiel mit einer `bad_alloc`-Ausnahmebehandlung. Sie müssen in beiden genannten `.cpp`-Dateien eine Anweisung `#include <stdexcept>` einfügen.
- Die Fehlermeldungen müssen identisch sein mit den in `files2msg` aus `check_err.py` spezifizierten Fehlermeldungen. Das ist gewährleistet, wenn Sie die beiden Funktionen `throw_empty_sequence` und `throw_missing_header` verwenden.
- Die Fehlermeldungen werden über den Fehlerstream `std::cerr` ausgegeben. Die Ausgabe erfolgt ausschließlich in `multiseq_test.cpp`. Daher ist eine Anweisung `#include <iostream>` notwendig.
- Sie dürfen in Ihrer eigenen Implementierung Speicher nicht mit `malloc`, `calloc` oder `realloc` allokieren.
- Das Einlesen des Dateiinhaltes soll wie im C-Programm mit Hilfe der Klasse `PfNFileInfo` erfolgen. Dafür finden Sie die entsprechenden Dateien in den Materialien. Diese können unverändert benutzt werden, d.h. eine Konvertierung nach C++ ist nicht erforderlich.
- Während im C-Programm die Zeiger auf die Anfänge der Kopfzeilen bzw. der Sequenz jeweils in einem dynamischen Array gespeichert werden, sollen Sie in der C++-Implementierung Instanzen `header_vector` und `sequence_vector` der Klasse `std::vector` verwenden, die bereits in `multiseq.hpp` deklariert sind. Sie brauchen dazu die Methoden `push_back()`, `back()`, `size()` und einen indexbasierten Zugriff.
- Zur Ausgabe von Sequenzen in der Methode `show` können Sie wie bei der C-Implementierung `fwrite` verwenden.

In den Materialien finden Sie ein Makefile zum Kompilieren Ihrer Quelldateien. Durch `make test` verifizieren Sie, dass Ihre Implementierung korrekt funktioniert. Wenn der Test nicht erfolgreich ist, dann liefert das Shell-Skript `multiseq_test.sh` entsprechende Fehlermeldungen. Sie sollten sich den ersten fehlgeschlagenen Test ansehen, den Fehler in Ihrem Programm beseitigen, und dann weiter fortfahren. Bzgl. des Python-Skripts `check_err.py`, das die korrekte Fehlerbehandlung verifiziert, gehen Sie analog vor. Falls Sie nicht unter macOS arbeiten und das Programm `valgrind` verfügbar ist, wird dieses in einem weiteren Test verwendet.

Punkteverteilung:

- 2 Punkte für den Konstruktor `Multiseq`.
- 1 Punkt insgesamt für die anderen Funktionen der Klasse `Multiseq`.

- 1 Punkt für die Implementierung der Funktion `main()` in `multiseq_test.cpp`.
- 1 Punkt für bestandene Tests.

Aufgabe 10.3 (6 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 $A = \{3, 5, 7, 11\}$. 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 $|a|$ für die Länge des Arrays a und hs ist die Abkürzung von *has solution*. 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.hpp` mit den Vorwärtsdeklarationen der zu implementierenden Funktionen.
- Textdateien mit ganzen Zahlen und Dateien mit den erwarteten Ergebnissen für vorgegebene Werte von s .

Teil 1

Implementieren Sie in der Datei `subsetsum.cpp` eine C++-Funktion

```
bool *subsetsum(const size_t *arr, size_t r, size_t s)
```

Diese Funktion ruft eine rekursive Funktion `subsetsum_rec` auf, die nach der obigen rekursiven Definition das Teilmengen-Summen-Problem 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 den Wert `nullptr` zurückliefern, wenn das Teilmengen-Summen-Problem für A und s keine Lösung hat. Wenn es eine Lösung gibt, 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 des Elementes `a[i]` in der Rekursion durch eine 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 Fehler gibt:

```
bool *subsetsum_memo(__attribute__((unused)) const size_t *arr,
                    __attribute__((unused)) size_t r,
                    __attribute__((unused)) size_t s)
{
    return nullptr;
}
```

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 lohnt sich also nicht zu warten, bis `make test_difficult` beendet ist.

Es ist daher sinnvoll, den rekursiven Algorithmus zu optimieren. Dieser Schritt erfolgt im zweiten Teil der Aufgabe.

Teil 2

Implementieren Sie in der Datei `subsetsum.cpp` die Funktion `subsetsum_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 Peer-Teaching Aufgabe über Berechnung von Binomialkoeffizienten kennen gelernt haben.

Diese Funktion ruft eine rekursive Funktion `subsetsum_rec_memo` auf mit den gleichen Fallunterscheidungen wie oben, aber unter der Verwendung einer Matrix als Gedächtnis. Die Funktion hat daher einen weiteren Parameter, nämlich einen Zeiger auf eine Instanz der Klasse `DefineValueMatrix` (siehe `defined_value_matrix.hpp`). In einer Instanz dieser Klasse werden Elemente des Typs `DefinedValue` gespeichert, der jeweils zwei boolsche Werte `defined` und `has_solution` repräsentiert. Dabei ist `defined` genau dann `true`, wenn der Wert von `has_solution` definiert ist. Der

`has_solution`-Wert in Zeile i und Spalte t speichert den **return**-Wert des ersten Aufrufs von `subsetsum_rec_memo(..., i, t, ...)`.

Da $i \leq r$ und $t \leq s$ gilt, benötigt man eine $(r + 1) \times (s + 1)$ -Matrix, die Sie durch

```
DefinedValuesMatrix sol_tab(r+1,s+1);
```

erzeugen. Die Einträge in der Matrix sind anfangs alle undefiniert. Der lesende Zugriff erfolgt über die Methoden `defined_is_set` und `has_solution_is_set`.

Der Ausdruck `sol_tab_ptr->defined_is_set(row,col)` liefert den `defined`-Wert in Zeile `row` und Spalte `col` der Matrix, auf die durch `sol_tab_ptr` verwiesen wird. Entsprechendes gilt für `has_solution_is_set`. Zum Überschreiben des `has_solution`-Wertes in einem Matrix-Eintrag wird die Methode `has_solution_set` verwendet.

Wenn ein Wert beim Aufruf der rekursiven Funktion vorher noch nicht berechnet wurde (`defined` ist nicht `true`), wird er berechnet und das Ergebnis in der Matrix in `has_solution` gespeichert. Wenn ein Wert bereits einmal berechnet wurde, wird er aus der Matrix (d.h. dem `has_solution`-Wert) gelesen. Beachten Sie, dass auch in dieser Funktion das Array `mark` berechnet werden muss. Das erfolgt in analoger Weise wie bei der Funktion `subsetsum_rec`.

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

Punkteverteilung:

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

Bitte die Lösungen zu diesen Aufgaben bis zum 22.06.2021 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.