

Programmierung für Naturwissenschaften 2
Sommersemester 2021
Übungen zur Vorlesung: Ausgabe am 28.04.2021

Bitte beachten Sie die Hinweise zu den für diese Übungsaufgaben relevanten Abschnitten der Vorlesung. Diese Hinweise finden Sie jeweils am Ende der Beschreibung der Übungsaufgaben.

Aufgabe 4.1 (2 Punkte)

Aus Zeitgründen können wir in diesem Semester in der Vorlesung das Thema Codon-Translation in C (siehe `C_slides.pdf`, Abschnitt 13, frame 1-9) nicht behandeln. Aus der Schule kennen Sie bereits die biologischen Grundlagen des Themas. Zudem wurde eine Implementierung der Codon-Translation schon einmal als Peer-Teaching Aufgabe in PfN1 behandelt. In jeder der Kleingruppen muss sich der Studierende mit dem lexikographisch kleinsten Nachnamen anhand der oben genannten Folien auf das Thema vorbereiten und das erworbene Wissen an die anderen Mitglieder der Kleingruppe weitergeben.

Falls jemand von diesen Studierenden nicht zur Übung erscheint, werden die übrigen Mitglieder der betroffenen Kleingruppe am Anfang der Übung auf die anderen Kleingruppen verteilt.

Diese Aufgabe soll am Anfang der Übung bearbeitet werden. Der oder die Studierende, die sich vorbereitet hat, kann ggf. bei Unklarheiten Fragen, zumindest zu den Grundbegriffen, beantworten oder Erläuterungen mündlich ergänzen. Sie müssen sich auch die vollständige Implementierung in der Datei `dna2aa.c` ansehen, sie mit der Python-Implementierung in `dna2aa.py` vergleichen und durch `make test` verifizieren, dass beide Implementierungen die gleichen Ergebnisse liefern.

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.

Dokumentieren Sie die Eigenschaften der Eingabedaten, die bei `make test` verwendet werden, ebenso wie die Laufzeiten der beiden Implementierungen. Beschreiben Sie in wenigen Sätzen die wesentlichen Unterschiede der C- und der Python-Implementierung bzgl. der Techniken zur Extraktion der Codons und zur Translation eines Codons in eine Aminosäure. Insgesamt sollen es maximal 15 Zeilen mit maximal 80 Zeichen pro Zeile sein.

Die Unterschiede bzgl. des Einlesens der Dateien und der Ausgabe der Ergebnisse sollen nicht beschrieben werden.

Falls sich Studierende aus anderen Kleingruppen an dieser Aufgabe beteiligt haben, geben Sie bitte deren Namen mit an.

Aufgabe 4.2 (6 Punkte) Wir definieren eine Menge $M \subset \mathbb{N}$ durch die folgenden Bedingungen:

- $1 \in M$
- Falls $i \in M$, dann ist auch $2i + 1 \in M$ und $3i + 1 \in M$.
- Keine andere Zahl ist Element von M .

Schreiben Sie ein C-Programm `enumM.c` mit genau einem Parameter, nämlich einer positiven ganzen Zahl n . Überprüfen Sie in Ihrem Programm, dass genau ein Parameter über die Kommandozeile übergeben wurde und dass der Parameter eine positive ganze Zahl ist. Falls das nicht zutrifft, soll Ihr Programm eine sinnvolle Fehlermeldung auf `stderr` ausgeben. Das Programm soll die Elemente $i \in M$ mit $i \leq n$ in aufsteigender Reihenfolge ausgeben.

Beispiel: für $n = 13$ soll die Ausgabe wie folgt aussehen:

```
1
3
4
7
9
10
13
```

Der Wert von n kann beliebig sein, d.h. wenn Ihre Lösung ein Array verwendet, dann müssen Sie das Array mit dynamischer Speicherverwaltung allokieren und natürlich am Ende wieder freigeben. Auch wenn die Definition der Menge rekursiv ist, dürfen Sie in Ihrer Implementierung keine rekursive Funktion verwenden.

In den Materialien finden Sie ein Makefile. Durch `make` wird Ihr Programm kompiliert und es entsteht, wenn alles gut geht, ein ausführbares Programm `enumM.x`. Durch `make test_Enum` verifizieren Sie, dass das Programm die erwartete Ausgabe (siehe oben) erzeugt.

Die Anzahl der Berechnungsschritte Ihrer Implementierung muss kleiner oder gleich $a + bn$ sein, wobei a und b Konstanten sind, die nicht von n abhängen. Man sagt auch, dass die Laufzeit des implementierten Algorithmus linear ist in n . Diesen Begriff kennen die Studierenden, die bereits das Modul *Algorithmen und Datenstrukturen* absolviert haben. Er ist aber für das Verständnis des zweiten Teils der Aufgabe nicht notwendig. Sie haben nun alle Informationen für den ersten Teil dieser Aufgabe und sollten diese zunächst lösen, bevor Sie mit dem Lesen des zweiten Teils beginnen.

Im zweiten Teil der Aufgabe sollen Sie die genannten Konstanten a und b so schätzen, dass sich dadurch eine möglichst kleine obere Schranke für die Anzahl der Berechnungsschritte ergibt. Dazu erweitern Sie Ihr Programm um eine Variable `steps`, mit der Sie die Anzahl der Berechnungsschritte Ihres Programms zählen. Wir nehmen vereinfachend an, dass die folgenden Operationen jeweils einen Berechnungsschritt (CPU-Zyklus) erfordern:

- Wertzuweisung,
- Arithmetische Operationen (z.B. `+`, `<=`),
- Logische Operationen (wie z.B. `&&`),
- Test in einer `if`-Anweisung oder einer `for` bzw. `while`-Schleife,
- Indexzugriff auf Array (lesend oder schreibend),
- Funktionsaufruf von `printf` mit einem Argument,
- Aufruf von `malloc` zum Allokieren eines Speicherbereichs,
- Aufruf von `free` zur Freigabe eines Speicherbereichs.

Ein Aufruf von `calloc` zum Allokieren eines Speicherbereichs mit ℓ Bytes benötigt ℓ Schritte.

Fügen Sie jeweils zu einer Inkrementierungs-Anweisung für `steps` einen Kommentar analog zum folgenden Beispiel hinzu:

```
steps += 4; /* 2 Arithmetik, eine logische Operation, ein Test */
if (x * y + 1 <= z)
{
    steps += 4; /* 2 Arithmetik, Indexzugriff, Wertzuweisung */
    values[2 * x + 1] = 13;
}
steps++; /* Wertzuweisung am Schleifenanfang */
for (i = 0; i < 100; i++)
{
    steps += 3; /* logische Operation, Test, Update */
    steps += 2; /* Indexzugriff, Wertzuweisung */
    values[i] = 14;
}
```

Der Kommentar muss also angeben, welche Berechnungsschritte bzgl. der darauf folgenden Anweisung gezählt werden. Für den Test der Schleifenabbruchbedingung und der Update-Anweisung in einer `for`-Schleife soll die Erhöhung von `steps` am Anfang des Blocks, der zur Schleife gehört, erfolgen (siehe oben). Die auf `steps` angewendeten Operationen selbst sollen nicht gezählt werden.

Am Ende des Programms geben Sie den Wert von `n` und `steps` durch eine Anweisung der Form `printf("#%lu\t%lu\n", n, steps);` aus. Dabei ist `n` die Variable, die den Wert `n`, also die Obergrenze der ausgegeben Werte von `M`, speichert. Durch den Aufruf von `make steps.tsv` wird Ihr Programm für alle `n` von 1 bis 500 aufgerufen und die Werte von `n` und die jeweilige Anzahl $s(n)$ der Berechnungsschritte werden ausgegeben.

Mit Hilfe des Programms `fit_linear.py` (dessen Implementierung Sie nicht verstehen müssen) sollen Sie nun möglichst kleine ganzzahlige Werte für a und b finden, so dass $s(n) \leq a + bn$ für alle Werte aus `steps.tsv` gilt.

Dazu brauchen Sie eine Installation von Python3 inklusive Numpy und SciPy. Dann liefert der Aufruf `./fit_linear.py -i steps.tsv` Ihnen gut an die Daten angepasste reelle Werte für a und b .

Dokumentieren Sie Ihre Ergebnisse, d.h. die Werte ganzzahliger a und b , indem Sie sie in das Makefile an der passenden Stelle eintragen. Aktuell stehen dort die Werte, die für die Musterlösung gelten.

Durch den Aufruf von `make test_fit` wird verifiziert, dass die Werte zu einer oberen Schranke für $s(n)$ führen.

Punkteverteilung:

- 3 Punkte für die Implementierung der Methode zur Ausgabe von M
- 1 Punkt für den bestandenen Test
- 2 Punkte für die Bestimmung der Parameter von a und b

Zur Bearbeitung dieser Aufgabe sollten Sie die Abschnitte der Vorlesung von *Syntactic Basics of C* bis einschliesslich *Dynamic Allocation of Memory*, (außer *Case Study on Classification*) kennen.

Aufgabe 43 (3 Punkte) Betrachten Sie folgenden C-Code:

```
char c, *string, string2[3] = {0}, **strings;
int num, *nump, a, *b;
```

```

string = "Hallo Welt";
c = string[6];

string2[0] = 'A';
string2[1] = 'B';

strings = malloc(sizeof (*strings) * 3);
strings[0] = &c;
strings[1] = string2;
strings[2] = "third string";

a = 7;
nump = &a;
*nump = 5;
b = &a;
num = (int) (*(strings + 1))[1];

```

Dabei ist eine Wertzuweisung der Form **char *s = "abc";** eine Abkürzung für

```
char *s = {'a', 'b', 'c', '\0'};
```

- Benennen Sie jeweils den Typ der 8 deklarierten Variablen.
- Geben Sie jeweils den Wert der folgenden Ausdrücke nach Ausführung des Programmcodes an. Falls ein Wert eine Speicheradresse ist, schreiben Sie „Adresse“. Sie müssen natürlich keinen konkreten Wert angeben.

- c	- *strings
- string2	- strings[2][2]
- string2[0]	- num

- Geben Sie für jeden der folgenden Ausdrücke an, ob sie wahr oder falsch sind.

- *strings[0] == *(string + 6)	- *b == *nump
- b == nump	- string == strings[1]

Diese Lösungen schreiben Sie bitte als Kommentar in die C-Datei.

Zur Bearbeitung dieser Aufgabe sollten Sie sich die Folien aus `C_slides.pdf` zum Abschnitt „Pointers“ (frame 77-82) und zum Abschnitt „Arrays“ (frame 139-142, 169-174) angesehen haben. Die Zusammenfassung zu Zeigern (frame 200-202) ist ggf. ebenso hilfreich.

Bitte die Lösungen zu diesen Aufgaben bis zum 04.05.2021 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.