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

Programmierung für Naturwissenschaften 2
Sommersemester 2020
Übungen zur Vorlesung: Ausgabe am 19.05.2020

Bitte beachten Sie, dass Fehlermeldungen mit `fprint(stderr,...)` ausgegeben werden und (wenn es nicht anders angeben wird), das Programm mit `exit(EXIT_FAILURE);` abgebrochen wird (in `main` kann auch **`return`** `EXIT_FAILURE` stehen). Beim erfolgreichen Durchlauf des Programms ist **`return`** `EXIT_SUCCESS` die letzte Anweidung des Programms (die in der `main`-Funktion steht. Damit diese beiden Konstanten bekannt sind, müssen Sie mit einer **`#include`**-Anweisung `stdio.h` einfügen.

Aufgabe 4.1 (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 einen 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 += 3; /* 2 Arithmetik, 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 += 2; /* Schleifenabbruchbedingung, 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.

Falls Sie auf Ihrem Rechner eine Installation von Python3 inklusive SciPy zur Verfügung haben, dann liefert der Aufruf `./fit_linear.py` Ihnen bereits gut an die Daten angepasste reelle Werte für a und b.

Falls Sie keine Installation von Python3 inklusive SciPy zur Verfügung haben, dann schätzen Sie die Werte von a und b aus den Daten selbst zunächst grob ab oder benutzen eine andere Methode zur Abschätzung.

Ganzzahlige Werte für a und b können Sie nun verifizieren, indem Sie `./fit_linear.py` mit den Optionen `-a` und `-b` aufrufen. Diese erwarten als Argument die Werte von a bzw. b.

Sie werden wahrscheinlich feststellen, dass es Werte von n mit $s(n) > a + bn$ gibt. Entsprechende Fälle werden Ihnen angezeigt. Variieren Sie nun a und b in den Aufrufen von `./fit_linear.py`, bis keine Ausgabe der Form `no upper bound ...` erscheint. Der letzte Wert am Ende der Ausgabe ist $\sum_n (a + bn - s(n))$. Sie sollten in 10–20 Versuchen diesen Wert minimieren.

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 *Dynamic Allocation of Memory*, (außer *Case Study on Classification*) kennen.

Aufgabe 4.2 (4 Punkte) Die Türme von Hanoi bestehen aus drei Stäben, auf die n Scheiben verteilt sind. Die Scheiben sind alle unterschiedlich groß und daher auch unterschiedlich schwer. Anfangs befinden sich alle Scheiben auf Stab 1, wie etwa in der folgenden Abbildung dargestellt:

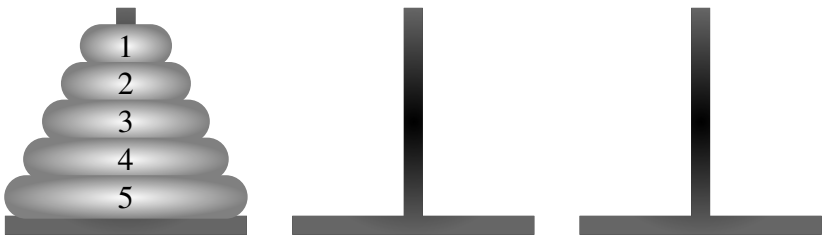

Abbildung 1: Ausgangslage für $n = 5$

Die Aufgabe besteht nun darin, alle Scheiben nach Stab 2 zu verschieben, wobei immer nur eine Scheibe bewegt werden darf. Stab 3 darf dabei zur Hilfe genommen werden. Aufgrund des Gewichtes der Scheiben darf niemals eine größere Scheibe auf einer kleineren Scheibe liegen. Die Lösung soll keine unnötigen Züge enthalten. Das Ergebnis der Bewegungen der Scheiben sieht dann so aus:

Implementieren Sie in der Datei `hanoi.c` eine C-Funktion **`void`** `hanoimoves(`**`int`** `n)`, so dass `hanoimoves(n)` die Folge der notwendigen Schritte für die Bewegung von n Scheiben von Stab 1 nach Stab 2 auf `stdout` (also mit `printf`) ausgibt. Z.B. soll `hanoimoves(3)` die Liste

```
(1,2) (1,3) (2,3) (1,2) (3,1) (3,2) (1,2)
```

liefern. Dabei folgt auf jedes Zahlenpaar ein Leerzeichen. Jedes Paar (i, j) mit $i, j \in \{1, 2, 3\}$ in dieser Liste bedeutet, dass die oberste Scheibe von Stab i nach Stab j bewegt wird. Um selbständig die Lösung zu entwickeln, schauen Sie sich im Material zu dieser Übung die drei Dateien `hanoi-moves`n`.txt` für $n \in \{3, 4, 5\}$ an, in denen jeweils die Folge der Bewegungsschritte für die Scheiben zusammen mit dem aktuellen Zustand der drei Stäbe angegeben wird.

Hinweis: Versuchen Sie das Problem in Teilprobleme zu zerlegen und lösen Sie dann die Teilprobleme. Durch Hinzufügen eines zusätzlichen Schritts, der genau eine Scheibe bewegt, erhalten Sie dann die komplette Lösung.

Durch `make` können Sie Ihr Programm kompilieren. Durch `make test` verifizieren Sie die Korrektheit für $n \in \{3, 4, \ldots, 7, 8\}$.

Zur Bearbeitung dieser Aufgabe sollten Sie die Abschnitte der Vorlesung von *Syntactic Basics of C* bis *Recursion and Sorting*, ausgenommen die folgenden Abschnitte kennen:

- *Case Study on Classification*,
- *2Dim Arrays*,
- *Structures*,
- *Synopsis Pointer Notation*,
- *Valgrind*,
- *Codon translations*,
- *Functions on Arrays*.

Im Abschnitt *Functions* sind für diese Aufgaben nur die Seiten 208–221 aus `C_slides.pdf` relevant. Im Abschnitt *Recursion and Sorting* sind für diese Aufgabe nur die Seiten 286–299 aus `C_slides.pdf` relevant.

Bitte die Lösungen zu diesen Aufgaben bis zum 26.05.2020 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.

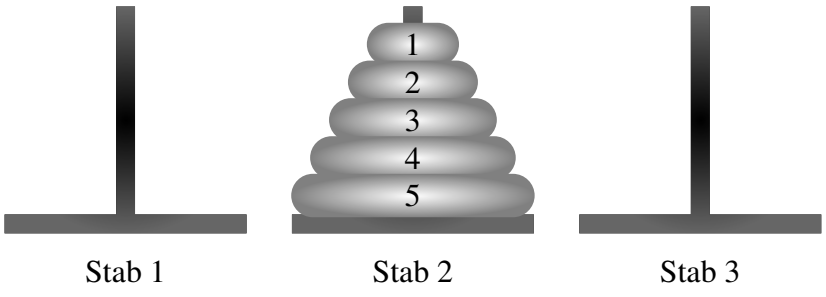

Abbildung 2: Endlage für $n = 5$