Dr. U. Köthe *Heidelberg, 14. Dezember 2016*

Übungsblatt 8

Aufgabe 8.1 Arithmetik für natürliche und ganze Zahlen [11 Punkte]

Die Arithmetik der natürlichen Zahlen (Datentypen `uintx_t` mit `x` $\in \{8, 16, 32, 64\}$ Bits) wird auf den meisten Computern Modulo 2^x implementiert:

$$r = (u \text{ OP } v) \bmod 2^x$$

Für die ganzen Zahlen (Datentypen `intx_t`) wird die Zweierkomplementdarstellung gewählt, d.h. für alle a im zulässigen Bereich $-2^{x-1} \leq a < 2^{x-1}$ gilt

$$-a = (2^x - a) \bmod' 2^x = {\sim}a + 1$$

wobei die Modulo-Operation $\bmod'$ wieder auf den Bereich $-2^{x-1} \leq a < 2^{x-1}$ abbildet (anstatt wie $\bmod$ auf den Bereich $0 \leq r < 2^x$), und ${\sim}a$ die bitweise Negation in `x`-Bit-Darstellung ist. Benutzen Sie für die Aufgabe die Rechenregeln der Modulo-Operation[1].

(a) Man kann durch die Operationen `uintx_t(a)` bzw. `intx_t(u)` die Bitfolge einer ganzen Zahl `a` mit `x` Bits als natürliche Zahl uminterpretieren und umgekehrt (wichtig: die Binärdarstellungen ändern sich dabei nicht, nur deren Interpretation). Beschreiben Sie den Effekt dieser sogenannten *Typumwandlungen* durch mathematische Formeln.

(b) Ein Vorteil des Zweierkomplements ist, dass die Addition, Subtraktion und Multiplikation von ganzen Zahlen a, b vom Typ `intx_t` auf die entsprechenden Operationen der natürlichen Zahlen zurückgeführt werden können. Es gilt

$$c = (a \text{ OP } b) \bmod' 2^x \quad \Leftrightarrow \quad c = \texttt{intx\_t}\big((\texttt{uintx\_t(a)} \text{ OP } \texttt{uintx\_t(b)}) \bmod 2^x\big)$$

Beweisen Sie dies für die Addition (OP $= +$) und die Multiplikation (OP $= *$) für den Fall $a \geq 0$ und $b < 0$ (der Beweis für die anderen Fälle funktioniert analog).

(c) Beweisen Sie, dass für positive Zahlen u, v, w die Identität

$$((u + v) \cdot w) \bmod 2^x = (((u + v) \bmod 2^x) \cdot w) \bmod 2^x$$

gilt. Widerlegen Sie durch ein Gegenbeispiel die Vermutung, dass auch

$$((u + v)/w) \bmod 2^x = (((u + v) \bmod 2^x)/w) \bmod 2^x$$

gilt. In der Praxis bedeutet dies beispielsweise, dass Addition, Subtraktion und Multiplikation das gleiche liefern, wenn man durchweg mit `uint8_t` rechnet (rechte Seite der Identität mit $x = 8$), oder wenn man intern `uint32_t` benutzt und erst am Schluss auf `uint8_t` reduziert (also die höheren Bits wegwirft, linke Seite der Identität). Bei Division gilt das hingegen nicht.

[1] siehe z.B. `https://en.wikipedia.org/wiki/Modulo_operation#Equivalencies`

Aufgabe 8.2 Perfect Shuffle nach Position k [6 Punkte]

Wir greifen hier nochmals die Methode des Perfect Shuffle aus Übung 5 auf, um die Verwendung der bitweisen Operatoren zu üben. Ein Zauberkünstler kann durch geeignetes Abwechseln von Out- und In-Shuffle erreichen, dass die anfangs oberste Karte an eine beliebige Position k gebracht wird. Das Kartenspiel wird wieder durch ein Array vom Typ `std::vector<int>` mit den Zahlen $0...51$ repräsentiert (Karte 0 ist anfangs oben).

Um Karte 0 an Position k zu bringen, muss man k in Binärdarstellung bringen und dann die Bits von links nach rechts (also vom größten zum kleinsten) durchgehen. Für jedes 0-Bit führt man ein Out-Shuffle aus, für jedes 1-Bit ein In-Shuffle. Führende 0-Bits kann man dabei ignorieren, weil Karte 0 danach immer noch oben wäre. Für Interessierte: Das funktioniert, weil ein Out-Shuffle die Karten von Position i (mit $i < 26$) zur Position $2i$ bewegt, und ein In-Shuffle zur Position $2i + 1$.

Schreiben Sie eine Funktion

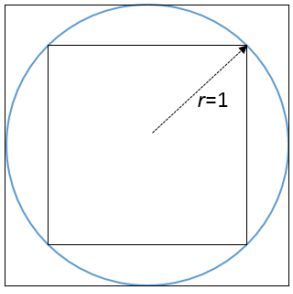
```
std::vector<int> shuffle_top_to(std::vector<int> deck, uint8_t k)
```

die diese Methode realisiert. Verwenden Sie die bitweisen Operatoren von C++, um über die richtige Reihenfolge von Out- und In-Shuffle zu entscheiden. Testen Sie mittels `assert()` in einer Schleife über alle möglichen k, dass sich die Karte 0 danach tatsächlich an Position k befindet. Geben Sie Ihre Lösung im File `perfect_shuffle_to_k.cpp` ab.

Aufgabe 8.3 Algorithmus von Archimedes zur Bestimmung von π [11 Punkte]

Numerische Algorithmen können wegen der begrenzten Genauigkeit von Computerberechnungen auch dann falsche Resultate liefern, wenn sie eigentlich mathematisch korrekt implementiert sind. Glücklicherweise erlebt man das nur noch selten, weil der Datentyp `double` und seine arithmetischen und algebraischen Operationen mittlerweile sehr ausgereift sind, aber diese Übungsaufgabe demonstriert, dass Fehler auch bei einfachen Aufgaben immer noch auftreten können. Mit dem Typ `float` wird der Effekt noch deutlicher. Geben Sie die Lösung im File `archimedes.cpp` ab.

Gute Schätzwerte für π wurden bereits in der Antike benötigt (z.B. um die Größe und damit den Preis eines Grundstücks mit gekrümmten Grenzen zu bestimmen). Archimedes (ca. 287 – 212 v. Chr.) hat dafür eine geniale Methode erfunden: Man zeichne zunächst einen Einheitskreis (Radius $r = 1$), dessen Umfang nach Definition 2π ist. Dann konstruiere man ein regelmäßiges n-Eck, das den Kreis genau von innen berührt, und eines, das den Kreis genau von außen berührt. Die Skizze zeigt dies für Quadrate ($n = 4$):



Das innere n-Eck hat stets einen kleineren Umfang als der Kreis, das äußere einen größeren. Wenn s_n und t_n die Seitenlängen des inneren und des äußeren n-Ecks sind, gilt somit

$$ns_n < 2\pi < nt_n \quad \text{bzw.} \quad \frac{n}{2}s_n < \pi < \frac{n}{2}t_n$$

Im Falle des Quadrats findet man leicht $s_4 = \sqrt{2}$ und $t_4 = 2$, also $2.82 < \pi < 4$. Das sind natürlich sehr ungenaue Schätzungen. Kennt man aber die Seitenlängen für ein bestimmtes n, kann man die Seitenlängen für $2n$ einfach nach folgenden Formeln berechnen:

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \quad \text{und} \quad t_{2n} = \frac{2}{t_n}\left(\sqrt{4 + t_n^2} - 2\right)$$

Durch wiederholtes Verdoppeln von n bekommt man immer bessere Schätzungen, weil die n-Ecke den Kreis immer besser approximieren. Archimedes hat mit dem Sechseck begonnen und gelangte nach vier Verdoppelungen zum 96-Eck. Sein Ergebnis $\frac{223}{71} < \pi < \frac{22}{7} \approx 3.142$ (mit nur 0.04% Abweichung) war viele Jahrhunderte lang die beste bekannte Approximation.

(a) Implementieren Sie den Algorithmus unter Verwendung des Datentyps `float` (beachten Sie, dass `float`-Literale mit dem Suffix `f` geschrieben werden, z.B. `4.0f`). Beginnen Sie mit einem Quadrat und führen Sie 13 Verdoppelungen aus (wir sind dann beim 32768-Eck). Geben Sie in jeder Iteration den unteren und den oberen Schätzwert sowie deren Abweichung vom genauen Wert für π aus (erhöhen Sie die Anzahl der angezeigten Dezimalstellen mittels `std::cout << std::setprecision(16)`). Was beobachten Sie, und woher rührt dieses Verhalten?

(b) Wiederholen Sie das Experiment mit dem Datentyp `double` (wenn nötig mit mehr Iterationen). Was beobachten Sie jetzt?

(c) Numerikexperten empfehlen, die Gleichungen für s_{2n} und t_{2n} durch folgende Formeln zu ersetzen

$$s_{2n} = \frac{s_n}{\sqrt{2 + \sqrt{4 - s_n^2}}} \quad \text{und} \quad t_{2n} = \frac{2t_n}{\sqrt{4 + t_n^2} + 2}$$

Zeigen Sie mit Hilfe der binomischen Formel $(a+b)(a-b) = a^2 - b^2$, dass die neuen Formeln algebraisch äquivalent zu den alten sind. Wiederholen Sie das Experiment mit den neuen Formeln (jeweils für `float` und `double`) und überzeugen Sie sich, dass das Verfahren jetzt funktioniert. Warum sind die neuen Formeln besser? Wie viele zusätzliche Dezimalstellen von π bekommt man in etwa pro Iteration?

Aufgabe 8.4 Text dechiffrieren mit Umlauten [12 Punkte]

In dieser Aufgabe wenden wir uns nochmals der Entschlüsselung zu, aber die Aufgabe wird gegenüber Übung 6 leicht modifiziert:

- Es handelt sich diesmal um einen deutschen Text, wir müssen also Umlaute und 'ß' korrekt behandeln. Wir benutzen daher `wchar_t` und `std::wstring`.
- Wir verwenden diesmal einen eigenen Datentyp `Character` mit folgender Definition

```
struct Character
{
    wchar_t clear;
    wchar_t encrypted;
    int count;
};
```

Auf Moodle finden Sie die Datei `verschluesselt_utf8.txt`, die einen mit einer Substitutionschiffre verschlüsselten deutschen Text enthält. Ein Grundgerüst der Lösung finden

Sie auf Moodle in der Datei `decrypt_utf8.cpp` (sie enthält Code zum Einlesen und Schreiben von UTF8-Dateien). Geben Sie die vervollständigte Datei ab.

Leider bieten noch nicht alle Compiler die volle Unicode-Unterstützung, z.B. brauchen Sie mindestens Version 5 von g++ und Version 2015 von Visual Studio. Um Ihren Compiler zu testen, kompilieren Sie `decrypt_utf8.cpp` unverändert und führen es aus. Die Ausgabedatei `entschluesselt.txt` muss dann identisch zur Eingabedatei `verschluesselt_utf8.txt` sein. Funktioniert das nicht, fragen Sie bitte Ihren Tutor.

(a) Schreiben Sie eine Funktion `int biggest_code(std::wstring const & text)`, die den größten in `text` vorkommenden Zeichencode bestimmt. Lesen Sie den Text ein, wenden Sie die Funktion darauf an und geben Sie das Ergebnis aus.

(b) Legen Sie ein Array `std::vector<Character> characters` an, das groß genug ist, um alle im Text vorkommenden Zeichencodes als Index verwenden zu können. Initialisieren Sie die Elemente des Arrays so, dass der Datenmember `count` auf 0 gesetzt wird und `clear` sowie `encrypted` dem jeweiligen Index entsprechen.

(c) Bestimmen Sie die Häufigkeit der Zeichen im Text, indem Sie `count` beim jeweiligen Index von `characters` hochzählen. Wandeln Sie dafür Großbuchstaben zunächst in Kleinbuchstaben um, so dass Groß- und Kleinschreibung ignoriert wird. Damit dies auch mit Umlauten funktioniert, muss der Funktion `std::tolower()` ein zweites Argument für die verwendete Sprache übergeben werden (`german` ist zu diesem Zweck in `decrypt_utf8.cpp` bereits definiert):

```
std::tolower(zeichen, german)
```

`std::tolower()` ohne Sprache kennt nur das englische Alphabet, keine Umlaute.

(d) Rufen Sie `std::sort()` mit einer geeigneten Lambda-Funktion auf, um das Array `characters` aufsteigend nach `count` zu sortieren.

(e) Lesen Sie die Datei `buchstaben_haeufigkeit_utf8.txt` (ebenfalls auf Moodle) ein, die die Zeichen des Alphabets aufsteigend nach Häufigkeit enthält. (Die Zeichen ‘x’ und ‘q’ fehlen allerdings, weil sie in unserem Text nicht vorkommen. Außerdem unterscheidet sich die Reihenfolge leicht von Angaben im Internet, unter anderem weil dort die Umlaute nicht separat, sondern als ‘ae’, ‘oe’ und ‘ue’ gezählt werden.) Iterieren Sie jetzt über das sortierte `characters`-Array und identifizieren Sie alle Kleinbuchstaben, deren `count` ungleich Null ist (die Funktion `std::islower()` unterstützt ebenfalls ein zweites Argument für die Sprache). Weisen Sie dem Datenmember `clear` entsprechend der Häufigkeit den passenden Klarbuchstaben zu.

(f) Rufen Sie `std::sort()` mit einer geeigneten Lambda-Funktion auf, um das Array `characters` wieder in die ursprüngliche Reihenfolge zu bringen.

(g) Iterieren Sie über den Text und ersetzen Sie mit Hilfe von `characters` die verschlüsselten Buchstaben durch die jeweiligen Klarbuchstaben. Beachten Sie dabei, dass die Entschlüsselung der Großbuchstaben mit Hilfe der korrespondierenden Kleinbuchstaben erfolgen muss. Schreiben Sie das Ergebnis in die Datei `entschluesselt.txt` und geben Sie diese Datei mit ab.

Bitte laden Sie Ihre Lösung spätestens bis 21. Dezember 2016, 9:00 Uhr in Moodle hoch.