Übungsblatt 1

1. Im Moodle finden Sie eine Datei `largeInt.zip`, die die C-Quelldateien `largeInt.h` und `largeInt.c` enthält. In ersterer ist eine Struktur `LargeInt` vorgegeben, mit deren Hilfe ganze Zahlen beliebiger Größe gespeichert werden können. Die `largeInt.h` Datei deklariert außerdem vier externe Funktionen, von denen aber nur eine in `largeInt.c` tatsächlich definiert werden. Implementieren Sie die restlichen Funktionen, indem Sie `largeInt.c` entsprechend erweitern und achten Sie darauf effizienten Code zu produzieren! Achten Sie auf eventuelle Kommentare in den Vorlagen und implementieren Sie die Funktionen im Sinne dieser Kommentare!

   Erläuterungen:

   Bekanntlicherweise kann die Hardware von Standardrechnern bestenfalls Zahlen in der Größenordnung von 64 Bits verarbeiten. Für wesentliche Kryptosysteme (allen voran RSA) brauchen wir aber eine Arithmetik mit sehr viel größeren Zahlen (etwa 2048 bis 4096) Bits.

   Diese muss folglich per Software implementiert werden. Im gegebenen Ansatz wird eine große Zahl einfach als ein Array von 32-Bit-Zahlen gespeichert. Arithmetik könnte nun direkt auf Bit-Ebene durchgeführt werden, man will sich aber die auf der Hardware ja vorhandenen arithmetischen Fähigkeiten für "kleine" Zahlen zunutze machen. Z.B. kann die Addition einer sehr großen Zahl auf eine Reihe von Additionen von kleineren Zahlen heruntergebrochen werden.

   Beispiel: Nehmen wir an, wir hätten einen Rechner zur Verfügung, der mit 4-Bit Zahlen rechnen kann. Wir brauchen aber 12-Bit Zahlen. Wir können wir eine Addition von zwei 12-Bit Zahlen simulieren?

   ```
      011110100110
   +  010111110111
   ---------------
      110110011101
   ```

   Wir könnten einfach nach Schulmethode Bitweise von rechts nach links addieren und den Übertrag jeweils mitführen. Das ist aber nicht effizient, weil uns die Hardware ja Arithmetik mit 4-Bit Zahlen erlaubt, d.h. wir könnten auch je zwei 4-Bit Zahlen addieren. Man könnte also die 12-Bit-Zahlen in je drei 4-Bit Zahlen unterteilen.

   ```
      0111 1010 0110
   +  0101 1111 0111
   -----------------
      1101 1001 1101
   ```

   Wie man am obigen Beispiel im mittleren Block aber sieht, entsteht bei der Addition von `1010` und `1111` ein Übertrag, der in den linken Block mit eingehen muss. Wenn wir auf einer Hardware mit 4-Bit-Zahlen rechnen, können bei der Addition Zahlen mit 5 Bits entstehen (z.B.: `1010 + 1111 = 11001`), was auf einer Hardware mit nur 4 Bits zu einem Überlauf führen würde. Daher können wir nur weniger als 4 Bit tatsächlich nutzen, so dass Platz für Überträge bleibt. Zum Beispiel könnten wir unsere 12 Bits in 6 Blöcke mit je 2 Bits aufteilen:

   ```
      01 11 10 10 01 10   ← 1. Summand
      01 01 11 11 01 11   ← 2. Summand
   +  01 01 01 00 01 00   ← Übertrag
   ---------------------
      11 01 10 01 11 01
   ```

   Vor dem Hintergrund dieser Ideen erklären sich die Konstanten aus `largeInt.h`:

- `BITSPERWORD`: Anzahl der Bits, die in einem 32-Bit-Wort tatsächlich zur Informationspeicherung genutzt werden.
- `STANDARD_USEBIT_MASK`: Zahl zur Maskierung der tatsächlich verwendeten Speicherbits.
- `STANDARD_CALCBIT_MASK`: Zahl zur Maskierung von evtl. auftretenden Überläufen.
- `WORD_RADIX`: Die Basis die wir zur Speicherung unserer Zahlen verwenden.
  Beispiel: Dezimalzahlen sind ja Zahlen zur Basis 10, Binärzahlen sind Zahlen zur Basis 2 und in unserem obigen Beispiel, in dem wir eine Zahl aus einer Reihe von 2-Bit-Zahlen zusammengesetzt haben, hatten wir es eigentlich mit Zahlen zur Basis 4 zu tun. Ist k die Basis unserer Zahlen und sind $a_{n-1}, \ldots, a_0$ die n Ziffern einer Zahl, so ergibt sich ihr Wert bekanntlich als $\sum_{i=0}^{n-1} a_i \cdot k^i$.

In der Vorlage ist `BITSPERWORD` auf 5 gesetzt. Ist das ein sinnvoller Wert? Falls nein, welcher Wert wäre für unsere Zwecke am besten geeignet?