Project II – *Multi-Precision Arithmetic*

Due date for the assignment: December 13, 2018 23:59

Rules for submission:

- All calculations have to be verifiable, e.g. if you have to compute a modular inverse, use the EEA or Fermat's little theorem.
- Individual work is required for all exercises and especially the programming assignments.
- You have to submit three files in Moodle:
  project2_*name_matrikelnummer*.pdf,
  project2_code_*name_matrikelnummer*.pdf,
  project2_code_*name_matrikelnummer*.zip
- project2_*name_matrikelnummer*.pdf contains your solution to exercises 2, 3, 5, 6, 7 and 8. You can either make photos of or scan your handwritten solution or you can typeset your solution in Latex. Make sure to provide readable solutions, otherwise we can't give you points.
- project2_code_*name_matrikelnummer*.pdf contains a copy of your solution to the programming assignments 1, 4 and 9 (results and your code). Use the Latex template project2_code_submission.tex to create the PDF file.
- project2_code_*name_matrikelnummer*.zip contains your solution to the programming assignments, i.e., all files of the C project, but in particular your `mpa_algs.c` file.

Exercises:

1. **Implementation Assignment:**
   In the following you should implement the schoolbook multiplication method. But instead of reconsidering the pseudo-code given in the lecture that efficiently "implements" a reorganization of the schoolbook method from scratch (using only low-level operations), we want to develop a more modular implementation that uses several auxiliary multi-precision operations. You should do the following steps:
   a) Implement the schoolbook addition algorithm introduced in the lecture (Alg. 14.7, p. 594, HAC).
   b) Implement a function that computes the product of a multi-precision integer and single-precision integer. (Do not use `mpz_mul_ui`!)
   c) Implement a function that computes the product of a multi-precision integer and B^i, for $i \in \mathbb{N}_0$. (Do not multiply!)
   d) Implement the schoolbook multiplication algorithm by using the above functions within a single for-loop.

   Remarks: Please use the GMP library (`struct mpz_t`) and the additional functions `mpz_setlimbn`, `mp_add_limb` and `mp_mul_limb` defined in `gmp_ext.h`. Furthermore, please use the template `mpa_algs.c` for your implementation and test the correctness of your code by applying the respective functions in `testsuite.h`.

2. Compute the product $6831 \cdot 8406$ using the Karatsuba algorithm with as many iterations as possible (i.e. only intermediate multiplications with both operands being single-precision (one digit) should be computed without a further recursion). Show each step of the computation.

3. Assume that we do arithmetic with 1024-bit numbers and that the word size is 32-bit (e.g., DEC Alpha case). How many single-precision multiplications are (approximately) needed if we multiply two numbers

   a) with the schoolbook method,

   b) with the Karatsuba algorithm with 1/2/3/4/5 iterations (these are five different cases with five distinct answers)?

   Provide a table that shows the numbers of single-precision multiplications in percent for each of the five cases relative to the schoolbook method?

   Why is it pointless to use more than five iterations of the Karatsuba algorithm in this case?

4. **Implementation Assignment:**
   Consider the following recursive multiplication algorithm for two n-digit numbers that uses the *divide-and-conquer* principle:

   - Given: Two positive n-digit numbers
     $a = (a_{n-1}...a_0)_B$ and $b = (b_{n-1}...b_0)_B$.
   - If $n = 1$: Compute the product of a and b using single-precision multiplication.
   - If $n > 1$: Split up a and b into two halves
     $a^{(1)} = (a_{n-1}...a_m)_B$ and $a^{(0)} = (a_{m-1}...a_0)_B$ respectively
     $b^{(1)} = (b_{n-1}...b_m)_B$ and $b^{(0)} = (b_{m-1}...b_0)_B$, where $m = \lceil \frac{n}{2} \rceil$.

     Then the product of a and b is computed using the following formula:

     $$\begin{aligned} a \cdot b = & \ (a^{(1)} \cdot B^m + a^{(0)}) \cdot (b^{(1)} \cdot B^m + b^{(0)}) \\ = & \ a^{(1)} \cdot b^{(1)} \cdot B^{2m} + (a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}) \cdot B^m + a^{(0)} \cdot b^{(0)} \ (\#) \end{aligned}$$

     The inner products occurring in this formula are thereby computed by recursively applying the algorithm.

   a) Implement the above algorithm (use function-recursion!).

   b) Estimate the number of single-precision multiplications required by this algorithm in the case of two $n = 2^k$ $(k \geq 0)$ digit numbers (justify your estimation). Is it more efficient than schoolbook multiplication?

   c) Optimize your implementation by applying the ideas of Karatsuba, i.e.,

   $$\begin{aligned} (\#) = & \ a^{(1)} \cdot b^{(1)} \cdot B^{2m} + \\ & ((a^{(1)} + a^{(0)}) \cdot (b^{(1)} + b^{(0)}) - a^{(1)} \cdot b^{(1)} - a^{(0)} \cdot b^{(0)}) \cdot B^m + \\ & a^{(0)} \cdot b^{(0)} \end{aligned}$$

   You can make use of `mpz_sub`.

   Remark: Again, please use the template `mpa_algs.c` for your implementation and test the correctness of your code by applying the respective functions in `testsuite.h`.

5. Which of the following statements about squaring, schoolbook and Karatsuba multiplication are true?

   a) Karatsuba multiplication is a divide-and-conquer algorithm.

b) Concerning the efficiency of recursive Karatsuba as seen in Assignment 4, it is the best choice not to stop recursion before an operand length equal to one is reached.

c) When squaring an integer one can save approximately one half of the single-precision multiplications compared to using schoolbook multiplication.

d) Karatsuba is based on the Fast Fourier Transformation.

e) In each iteration of Karatsuba one trades two multiplications for extra additions and subtractions (compared to schoolbook multiplication).

f) Currently, there is no multiplication algorithm exhibiting a better asymptotic time complexity than Karatsuba.

6. Since all computers are binary, it is natural to choose $R = B^n = 2^{sn}$, where s is the word length of the computer, for the Montgomery reduction algorithm. Is it possible that R and the modulus of the RSA cryptosystem are *not* relatively prime (which is a necessary condition for the Montgomery reduction)?

7. Verify that `MRed`$(\tilde{a} \cdot \tilde{b}) = \tilde{c}$, where $c = a \cdot b \bmod m$, for the set of values $a = 9$, $b = 33$, $m = 71$, and $R = 100$. First compute $\tilde{c}$ directly from c, and secondly compute $\tilde{c}$ using Montgomery reduction (i.e., using the MRed algorithm presented in the lecture). Note down all steps (including, e.g., inverse calculations) in your solution.

8. Let the RSA cryptosystem with public parameters $m = 527$ and $e = 5$ be given. Choose an appropriate Montgomery radix R (for decimal representation) and encrypt the message $x = 100$ using square-and-multiply with Montgomery reduction.

9. **Implementation Assignment:**
As a final step you should implement the montgomery reduction `MRed`. Since we want to make use of the previously implemented Karatsuba multiplication, we do not consider the word-level montgomery algorithm here (it already includes an interleaved multiplication). Instead, you should implement the montgomery reduction in three steps:

   a) Implement the transformation function `Trans`.

   b) Implement a function that takes the modulus m as input and gives the parameter m' as output.

   c) Implement the montgomery reduction function `MRed`.

   Since we are developing a cross-platform library with GMP, you will need to choose the correct radix for the reduction on the fly, based on the word-size of the processor and the length of the input parameters. Furthermore you should keep in mind that the reduction has to be the most efficient function out of the three (i.e. about the complexity of two multiplications). While you are allowed to use any `mpz_...` function for the first two steps a) and b), you should by all means avoid any kind of complex computation (like division, reduction, exponentiation, ...) in the third step c).

   Remark: Again, please use the template `mpa_algs.c` for your implementation and test the correctness of your code by applying the respective functions in `testsuite.h`.