Instructions: Problems 3,4 and 7 are to be handed in by next Monday (theoretical ones on Moodle, programming ones on CoCalc). For SAGE homework, register on CoCalc at `www.cocalc.com` *with your EPFL e-mail. Gauthier (Gauthier.Leterrier@epfl.ch) will add you to the course and you can work on your assignments online. You have to submit your code via the SAGE worksheet* `hw1.sagews` *on your CoCalc account. (Optional: you can install SAGE locally via* `www.sagemath.org/download.html`*)*

1. Find an upper bound for the number of bit operations required to compute $n!$ and indicate your answer in big-O notation. Then do the same for $\binom{n}{m}$ and for multiplying two polynomials in $\mathbb{Z}[x]$ of degrees n_1, n_2 and coefficients bounded by m.

2. a) Suppose you want to test if a large odd number n is prime by trial division by all odd numbers less than $\sqrt{n}$. Estimate the number of bit operations that will take.

   b) Now suppose you had a list of all primes up to $\sqrt{n}$ at your disposal and proceeded to only test division by those. What would the estimate become now? *(Use the prime number theorem, which estimates the number of primes $\leq n$ by $n/\log n$.)*

3. Let K denote any field (assume $K = \mathbb{R}, \mathbb{C}$ if unfamiliar with fields). Prove that polynomials $K[x]$ form a Euclidean domain by exhibiting a Euclidean function. Then give a Euclidean algorithm on $K[x]$ and prove it computes the gcd of monic polynomials.

4. The purpose of this problem is to improve on the estimate for the number of divisions required in the Euclidean algorithm. Recall that the Fibonacci numbers are defined recursively via $f_1 = f_2 = 1$ and $f_{n+1} = f_n + f_{n-1}$ for $n \geq 2$.

   (a) Suppose that in the Euclidean algorithm given in class it takes k divisions to find $\gcd(a, b)$ for $a > b > 0$. Show that $a \geq f_{k+2}$. Is this inequality always strict?

   (b) Prove that $f_n = \frac{1}{\sqrt{5}}(\alpha^n - \overline{\alpha}^n)$, where $\alpha = \frac{(1+\sqrt{5})}{2}$ and $\overline{\alpha} = \frac{(1-\sqrt{5})}{2}$. Use this to give an upper bound for k in terms of a that beats $2\log_2(a)$. *(Hint: Use the matrix identity* $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ *and consider eigenvalues for the first part).*

5. Familiarize yourself with SAGE, for instance by going through the tutorial at `http://doc.sagemath.org/pdf/en/tutorial/SageTutorial.pdf`. Pay special attention to the sections on linear algebra, group theory, basic rings and number theory.

6. Familiarize yourself with lists in Python/SAGE and write a function *myDivisors(N)* that takes an integer N as input and outputs a list of all the divisors, including $1, N$.

7. Code a function in SAGE which performs the extended Euclidean algorithm, i.e. for two integers $a > b$ outputs integers $\gcd(a, b), u, v$ satisfying the decomposition

$$u \cdot a + v \cdot b = \gcd(a, b).$$

8. Recall that the Euler phi-function $\varphi(n)$ counts the positive integers coprime to n:

$$\varphi(n) := \sharp\{0 \leq b < n \mid \gcd(b, n) = 1\}.$$

Find a formula for $\varphi(p^\alpha)$ for p a prime and $\alpha \geq 1$ an integer. Compute $\varphi(45)$. Now try to guess a formula for $\varphi(n)$ based on how n factors into prime powers.

9. *(Optional exercise for those curious or familiar with the above)* Suppose that you wanted to multiply two polynomials

$$p(X) = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1} \text{ and } q(X) = b_0 + b_1 X + \cdots + b_{n-1} X^{n-1}$$

by computing the coefficients of $r(X) := p(X)q(X) = \sum_{i=0}^{2n-2} c_k X^k$ given by $c_k = \sum_{i=0}^{k} a_i b_{k-i}$. A straightforward approach would require n^2 multiplications and be rather slow. This can be improved on via an approach based on Fast Fourier Transform (FFT). The point is that a polynomial of degree $< n$ is determined by its values $p(\omega_n^k)$ for $0 \leq k \leq n-1$ where $\omega_n = \exp(2\pi i/n)$ is a primitive n-th root of unity. The vector of these values is essentially the Discrete Fourier Transform (DFT) of the vector of coefficients of p. Writing $\widehat{a_t} = \sum_{j=0}^{n-1} a_j \omega_n^{jt} = p(\omega_n^t)$, this transformation is given by

$$\text{DFT: } a = (a_0, \ldots, a_{n-1}) \mapsto \widehat{a} = (\widehat{a_0}, \ldots, \widehat{a_{n-1}}).$$

Key is that this can be done faster than $O(n^2)$ steps (algorithm given below for $n = 2^k$). Then it is easy to compute the products $r(\omega_{2n}^t) = p(\omega_{2n}^t)q(\omega_{2n}^t)$ for $0 \leq t \leq 2n$ via DFT and recover $r(X)$ from $\{r(\omega_{2n}^t) | 0 \leq t \leq 2n\}$ via the inverse of the DFT transformation (similar algorithm, just replace ω_{2n} with ω_{2n}^{-1} and divide by $2n$ at the end).

Algorithm 1 RECURSIVE-DFT

Require: An integer $n = 2^k$ and a vector $a = (a_0, \ldots, a_{n-1})$.
Ensure: The DFT $\widehat{a} = (\widehat{a_0}, \ldots, \widehat{a_{n-1}})$ of a.
1: **if** $n == 1$ **then**
2: $\quad \widehat{a_0} := a_0$
3: **else**
4: $\quad a^{even} := (a_0, a_2, \ldots, a_{n-2})$
5: $\quad a^{odd} := (a_1, a_3, \ldots, a_{n-1})$
6: $\quad \widehat{a^{even}} :=$ RECURSIVE-DFT$(n/2, a^{even})$
7: $\quad \widehat{a^{odd}} :=$ RECURSIVE-DFT$(n/2, a^{odd})$
8: $\quad \omega_n = \exp(2\pi i/n)$
9: $\quad \omega = 1$
10: $\quad$ **for** $i = 0, \ldots, 2^{k-1} - 1$ **do**
11: $\qquad \widehat{a_i} = \widehat{a_i^{even}} + \omega \cdot \widehat{a_i^{odd}}$
12: $\qquad \widehat{a_{i+2^{k-1}}} = \widehat{a_i^{even}} - \omega \cdot \widehat{a_i^{odd}}$
13: $\qquad \omega := \omega \cdot \omega_n$
14: $\quad$ **end for**
15: **end if**
16: **return** $\widehat{a} = (\widehat{a_0}, \ldots, \widehat{a_{n-1}})$

(a) Prove the correctness of the RECURSIVE-DFT algorithm for $n = 2^k$.

(b) Let $T(n)$ be a function denoting the number of elementary steps (multiplication/addition) of the algorithm. Observe that steps 6. and 7. take time $2T(n/2)$ and steps 10-14. take linear time. We get therefore the recurrence relation:

$$T(n) = 2T(n/2) + g(n) \ \forall n \geq 1$$

for some function $g(n) = O(n)$. Show that this implies $T(n) = O(n \log n)$.

(c) Performing DFT requires roots of unity (**Z** does not have many !). Still, you can now read up on the Schönhage-Strassen algorithm to multiply integers, which uses DFT for integers modulo appropriate N, and $\mathbb{Z}/N$ has many roots of unity.