Computer science problems.

About the problems. This problem set explores RSA encryption, digital signatures, and their applications to electronic cash.

What you need to do. For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible).

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows (free trial versions do not qualify). It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

    1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
    2. The *programming language*, including the *version* (Java 1.15, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
    3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Please read the instructions for individual problems on the input and output data.
       Input/output files are assumed to be at the default location for your program's project. Make it clear in comments where that is.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.
7. Make sure that you clearly specify where input files are supposed to be located, provide an example input file and an example of how the file name would be specified in the input. Use relative paths (from the top of the project or from the executable), not absolute paths.

- Clear, understandable, and well-organized code. This includes:

  1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
  2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
  3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
  4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
  5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

Background and problems.

We provide key definitions and ideas. For more details and discussion see [2] or any other textbook that covers RSA digital signatures.

This problem set is based on *RSA encryption*, a *public key encryption* scheme. In public key cryptography each participant generates a pair of keys: a *public* key and a corresponding *private* key. The two keys act as inverses of each other: anything that's encrypted by one can only be decrypted by the other one. The participant, say Alice, makes her public key known to the world and keeps her private key secret. If someone wants to send Alice a secret message, they would encrypt it with Alice's public key (anyone can do it). The message can be safely sent via an insecure channel since no one, except Alice, would be able to decrypt it: she's the only one who knows her private key.

There are many public key encryption schemes, each based on some computation that's easy to compute from known inputs, but is *computationally* *infeasible* to invert, i.e. restore the inputs from the result. *Computationally infeasible* means that the fastest known solution is exponential in the size of the inputs.

We will be using *RSA* encryption which is based on the *prime factorization* problem: given the product n of two primes p and q, there is no algorithm to find p and q faster than trying all possible candidates. Numbers that are used for RSA encryption are very large: over 1000 bits. RSA public key encryption scheme was first publicly described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman It was later established that an equivalent system was developed in 1973 by the English mathematician Clifford Cocks, but its usage remained classified until the 90s.

Problem 1: modular arithmetic. Before we start on the RSA, we need to develop some tools. Many computations in cryptography involve modular arithmetic on integers, i.e. arithmetic in which the result of a computation is taken modulo some number m (called the *modulus*). The notation $a \equiv b \mod m$ means that a and b have the same remainder when divided by m. For instance, $16 \equiv 5 \mod 11$ and $-10 \equiv 1 \mod 11$.

In most cases we are looking for the result of a computation modulo m in the set $\{0, 1, 2, \ldots, m-1\}$. For example, when we compute 2×10 modulo 11, of all possible numbers that are equivalent to 20 modulo 11 we choose 9 since $0 \leq 9 \leq 10$.

Part 1. Using the above conventions, perform the following computations by hand:

1. $10 + 9$ modulo 13
2. $5 - 12$ modulo 13
3. 11×5 modulo 13

Part 2. Prove that for any integer a and b the following holds:

1. $(a + b) \equiv (a' + b') \mod m$, where $a \equiv a' \mod m$, $b \equiv b' \mod m$, and $a', b' \in \{0, 1, 2, \ldots, m-1\}$.
2. $(a \times b) \equiv (a' \times b') \mod m$, where $a \equiv a' \mod m$, $b \equiv b' \mod m$, and $a', b' \in \{0, 1, 2, \ldots, m-1\}$.

Part 3. Compute the following by hand, using the result in Part 2, show your computations:

1. $19 \times (28 + 35) \times (-12)$ modulo 13.
2. $(11 + 15)^8$ modulo 13 (think of how to apply Part 2).

While addition, subtraction, and multiplication are quite straightforward in modular arithmetic, division is very different. In real numbers when we divide by 2, we multiply by the reciprocal (also called the *multiplicative inverse*) of 2, which is $\frac{1}{2}$. We say that 2 and $\frac{1}{2}$ (also denoted 2^{-1}) are multiplicative inverses of each other because their product equals 1. We take this as a definition of a multiplicative inverse and apply it to operations modulo m.

Consider a set $\{0, 1, 2, \ldots, m-1\}$ and operations modulo m. Then a multiplicative inverse of a is a number a^{-1} in that same set such that $a \times a^{-1} \equiv 1 \mod m$. For instance, $2^{-1} \equiv 7 \mod 13$ and $3^{-1} \equiv 9 \mod 13$.

You may observe that every number in $\{1, 2, \ldots, 12\}$ has a multiplicative inverse modulo 13. In general if m is prime then every non-zero natural number less than m has a multiplicative inverse modulo m. If m is composite, only numbers k such that $\gcd(m, k) = 1$ have multiplicative inverses, where $\gcd(m, k)$ is the greatest common divisor of m and k.

Part 4. Find multiplicative inverses of the following numbers modulo 11:

1. 3
2. 10
3. 7

Part 5. Using the definition that division by a is multiplying by a^{-1}, find the following modulo 11 (note that the result has to be between 0 and $m-1$ inclusive, so reduce the result modulo 11 if needed):

1. Divide 3 by 5
2. Divide 2 by 10
3. Divide 10 by 2

Part 6, programming. Multiplicative inverses for large numbers are found using Extended Euclidean Algorithm. See [2] or `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm` for its description.

Implement this algorithm as a program that prompts the user for two integer numbers: the modulus m and a number k you are trying to invert. k must be positive, and less than m. The program computes and prints k^{-1} modulo m if it exists. If it doesn't exist, i.e. if $\gcd(m, k) \neq 1$, the function should print "The inverse doesn't exist".

Use your algorithm to find the multiplicative inverse of:

1. 7 modulo 120
2. 100 modulo 22391
3. 10799 modulo 4699463680

Your result must always be in the set $\{0, 1, \ldots, m-1\}$.

Note: many programming languages, such as C, C++, and Java, have an operation % that acts like $\mod$ on positive numbers. However, it gives an incorrect result on negative numbers. See the following discussion of the issue and how to fix it: `https://www.delftstack.com/howto/java/mod-of-negative-numbers-in-java/` If you are using a different programming language, look up its documentation.

Part 7. What is the efficiency of the Extended Euclidean algorithm in the worst case? Your answer may be somewhat approximate, you don't need to prove it. However, provide your reasoning and explanations. If you need to review the definition of algorithm efficiency, see [1] or any other book on algorithms' efficiency.

Part 8. In cryptography it is quite common to raise a large number to a large power modulo some number m. As you might guess, it's helpful to reduce intermediate results modulo m to working with very large numbers.

Consider computing 7^{33} modulo 11. How would you perform the computation? Write out intermediate steps.

Part 9, programming. Generalize your approach to arbitrary $a^k \mod m$ and implement it as a program that prompts the user for a, k, and m (in that order) and computes and prints $a^k \mod m$. Note that a, k, m are all positive integers. Your implementation should handle 6 digit powers without a noticeable delay.

Please note that you may not use built-in libraries for handling large integers (such as BigInteger in Java) for this problem.

Problem 2: RSA encryption. This problem introduces the RSA encryption scheme. We will be using many of its foundational properties without a proof. If you are interested, please refer to books on cryptography and/or number theory. Here we will be focusing on implementation and on some properties that we will be using later. Also please note that our examples use small numbers, whereas the actual RSA numbers are very large.

As mentioned in the introduction, RSA is based on prime factorization problem: given two large primes p and q, it's easy to compute their product $n = p \times q$, but knowing n there is no better way to find p and q without trying all possible options.

To generate a public/private key pair, Alice picks two primes p and q and computes $n = pq$. Then she computes $\phi(n) = (p-1)(q-1)$ (this is known as *Euler's totient function* `https://en.wikipedia.org/wiki/Euler%27s_totient_function`). Note that someone who doesn't know p and q cannot compute $\phi(n)$.

Then Alice picks a number $e < n$ such that $gcd(e, \phi(n)) = 1$. This means that e has a multiplicative inverse modulo $\phi(n)$, and it can be found by Extended Euclidean algorithm. Let $d = e^{-1} \mod \phi(n)$. e is known as *encryption exponent* (or *public exponent*), and d as a *decryption exponent* (or *private exponent*).

Alice's public key is a pair (n, e), and her private key is a triple (p, q, d).

Part 1. Using the following pairs of p and q, generate RSA keys for at least two different e for each pair.

1. $p = 13, q = 7$
2. $p = 23, q = 19$

Part 2. Can you always take a prime e to guarantee that $\gcd(\phi(n), e) = 1$? If yes, why? If not, given an example showing when it doesn't work and briefly explain why it doesn't work.

Part 3. In RSA system the message to be encrypted is an integer number less than n, let's denote it m. Then the encryption of m is $c = m^e \mod n$. In order to decrypt c Alice computes $c^d = m^{ed} \equiv m \mod n$ (see [2] or any other cryptography textbook for a proof).

Given $p = 23$, $q = 19$, and $e = 5$, find d, and then:

1. Encrypt the message 33

2. Decrypt the message 264.

You might want to use the function for raising a number to a large power modulo another number.

Note that in real life one has to be careful not to send an obvious message, such as $m = 0$ or $m = 1$. Typically a padding scheme is employed which makes the message into a larger number. Padding also prevents certain attacks based on the multiplicative properties of RSA encryption that we will discuss later.

Part 4. Knowing that there is no way to factor a product of two primes except by trial and error and that there is no way of finding d from e and n without knowing $\phi(n)$, please compute the efficiency of Alice's decryption computations in terms of $O(n)$ and the number of computations someone would need to do to decrypt the message without knowing p, q, and d. What can you say about security of RSA encryption based on your results?

Part 5. Implement RSA the following functions:

1. RSA signature generation: given two primes and an encryption exponent, compute the public key and the private key. Print out both. If the encryption exponent is not relatively prime to $\phi(p \times q)$, print an error message.

2. RSA encryption: given a public key, i.e. a pair (n, e), and a message $0 \leq m \leq n$, return the encryption of m with this key.

3. RSA decryption: given a private key (p, q, d) and an encrypted message c, return the decryption of c. Use the function that you wrote in Problem 1 for raising a number to a large power modulo n. Note that there is a way to speed up your computation: `https://en.wikipedia.org/wiki/Chinese_remainder_theorem`. You are welcome to use it, but you don't have to.

4. Try your decryption on the following: $p = 8783, q = 9133, e = 5, m = 34367293$. Since you know p and q, you can also compute d and decrypt, to check.

Problem 3: digital signatures. We have observed that the public and private keys are inverses of each other: what one of them encrypts, the other one decrypts. Encrypting with one's public key is an operation that anyone can perform, and it guarantees secrecy: only the person who possesses the corresponding private key can decrypt and read the message. Now let's suppose Alice encrypts the message with her private key, i.e. she computes $m^d \mod n$. Anyone can decrypt it: $(m^d)^e \equiv m^{de} \equiv m \mod n$. Therefore there is no secrecy in the message. However, no one except Alice could've constrcted $m^d \mod n$ since she is the only one who knows d that matches her public key (n, e). Thus this process is equivalent to signing the message m.

Note that Alice would need to send the message m in plain text along with the signature $m^d \mod n$, this way a reader can verify the signature on the message.

Part 1. Assuming Alice's public key $n = 80215139, e = 5$, check if the message m and its signature $s(m) = m^d \mod n$ match. If verification fails, show exactly which parts of the computation don't match.

1. $m = 123, s(m) = 49259120$
2. $m = 555, s(m) = 59131983$
3. $m = 1234567, s(m) = 58520412$

Part 2. While one needs to know d in order to sign a message, it is actually very easy to create a pair $(m, m^d \mod n)$ that passes the verification. How can one do it? Is there anything they can gain from it? Clearly explain your answers.

Problem 4: hash functions and digital signatures. Typically in real life one wants to digitally sign documents, not relatively small numbers. However, RSA allows only messages that are smaller than n. The problem is solved by a *hash function.* Hash functions, also referred to as one-way functions, are functions that take an input of any size (such as a binary representation of a long document) and produce an output of a *fixed length.* For digital signatures this length must be smaller than the RSA modulus n.

Assuming that h is a hash function that all participants have agreed upon beforehand, Alice would take the following steps to produce a digital signature of a document X:

1. Compute $h(X)$
2. Compute $s(X) = h(X)^d$
3. Send $(X, s(X))$

Part 1. List steps that another participant, Bob, needs to take to verify Alice's digital signature of X knowing Alice's public key (n, e). Indicate when the signature is rejected as invalid. What is the computational efficiency of Bob's computations?

Part 2. You may've used hash functions for constructing hash tables `https://en.wikipedia.org/wiki/Hash_table`. In that case a typical function to use would be $h(x) = x \mod k$, where k is the size of the hash table. However, using hash functions for digital signatures requires using *cryptographic hash functions*, not just any hash functions. A cryptographic hash function has the following properties:

1. it is quick to compute the hash value for any given message

2. it is infeasible to generate a message that yields a given hash value (i.e. to reverse the process that generated the given hash value)

3. it is infeasible to find two different messages with the same hash value (such two messages form a *collision*)

4. a small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value

Explain which of these properties hold and which don't for $h(x) = x \mod k$.

Part 3. Explain how one can forge a digital signature if the hash function used in the scheme fails requirement 3 above.

Part 4. Explain a strategy to forge a digital signature if the hash function used in the scheme satisfies requirements 1,2, and 3, but fails the requirement 4 above.

Part 5. Assuming that you can perform 1 billion of hash computations per second, what is the largest length of the output of a hash function makes finding a collision feasible within a week with the probability of at least 1/2? What about finding a collision to a given X?

Problem 5: multiplicative property of RSA, blind signatures.

Part 1. Prove that for an RSA encryption $E(m)$ the following holds: $E(m_1 \times m_2) = E(m_1) \times E(m_2)$.

The result of part 1 is known as the multiplicative property of RSA. It allows an interesting use of digital signatures - so-called *blind signatures.* Blind signatures allow obtaining a digital signature on a message without revealing the message that's being signed. It's an electronic equivalent to signing a sealed envelope without knowing what's inside. This can be used in any situation when the digital signature certifies someone's real-life credentials (such that they are who they say they are), but the signed document doesn't have their identity. An application is described in Problem 5.

Part 2. In order to generate a digital signature, Bob sends Alice a message $r \times m$, where r is what's known as the blinding factor. Alice sends Bob $(r \times m)^d \mod n$. Explain what Bob needs to send as r so that he would be able to recover $m^d \mod n$ (the message m signed by Alice) from what Alice sent back. Note that all that Bob knows is Alice's public key (n, e).

Part 3. A digital signature service typically would have separate public keys (with their corresponding private keys): one for signatures, and one for secret communications, i.e. decrypting messages sent to them. Why is it a necessary security precaution, especially when the digital signatures in question are blind signatures? In other words, what can an attacker obtain if the same key is used for a digital signature service and for secret communication? Clearly describe the attack.

Part 4. Implement a blind signature process: a function that takes a message, "blinds" it by generating a blinding factor (make it based on a random number) and multiplying, calls the function that produces a digital signature for it, and then "unblinds" it to obtain the digital signature on the original message. The function may also take the public key for the signature or read the key from global variables.

Problem 6: Digital cash. In this problem you will use blind signatures to implement *digital cash:* electronic equivalent of cash that provides anonymity of transactions while preventing double-spending and other potential issues. The idea is due to David Chaum, although an attempted implementation never took off.

The system assumes that there is a *bank* that holds all of the participants' accounts. A *customer*, let's say Alice, withdraws digital cash from her account. Then she pays a *merchant*, say Bob. Bob verifies that the cash is valid and deposits it at the bank. It's credited to his account. While the bank can verify that the cash that Bob's depositing is valid, it has no way of tracking which customer it has come from since multiple customers have withdrawn cash.

In order for this to work we need blind signatures. The bank's public key (n, e) is assumed to be well-known. For simplicity we assume that there is only one denomination of cash, say only \$20 bills. It's easy to generalize this to more denominations. We also assume that the bank uses a specific cryptographic hash function f and all customers can easily compute this function on any argument.

First we explain a simple version of the system that doesn't prevent double-spending. Then we add more machinery to prevent double-spending.

In the *simplified* system in order to withdraw cash, Alice creates a bill number x for each bill that she would like get. Then she computes the hash $f(x)$ for each bill. After that she creates a random number r for each bill, generates a blinding factor for it (the way you did it in the previous problem), and sends it to the bank to sign. The bank can verify the request comes from Alice, it signs the values she sent and debits her account by the required amount.

Alice then unblinds the signature and combines it with x to obtain $(x, f(x)^d \mod n)$, where d is the bank's private exponent. Then Alice takes the money to pay Bob. Bob can easily verify that the money is real because of the bank's signature. He then checks with the bank to make sure the bill with the number x hasn't been used before, and then accepts the payment and deposits the bill. The bank also verifies that the bill is real and credits Bob's account. However, the bank doesn't know who paid Bob since Alice's money was signed blindly.

Part 1. Please answer the following questions about this system, clearly explain your answer:

1. Why does Alice gets the signature on $f(x)$, and not on x itself? What kind of cheating would the cash of the form $(x, x^d \mod n)$ allow?
2. $f(x)$ is a cryptographic hash function, so no one (including the bank) can invert it to get x. Why can't Alice send $f(x)$ to the bank to sign it as is, not blinded?
3. Why does Bob need to call the bank before the accepts the cash, to make sure that no one has used this bill number already?

If you answered the questions above correctly, you observed that the system is good at discovering a double-spending, but not particularly good about doing anything about it. The need for Bob to check with the bank during every transaction is inconvenient and also lets the bank to know the exact time for every transaction, interfering with anonymity.

To deal with these issues, Chaum proposed a *modification* to the system that allows detection of double-spending in a way that exposes double-spenders.

The idea is that one bill of digital cash is made up for multiple chunks, each of which embeds Alice's information (her account number and the bill number). When paying Bob, she opens up a part of each chunk - there are two options of opening it, and Bob gives her a string of 0s and 1s to indicate which ones he wants.

The trick is that opening a chunk in only one of the two ways doesn't reveal anything. However, if Alice spends the same bill at another merchant's, say at Carol's, this merchant will choose a different string of 0s and 1s, so there would be chunks for which Bob and Carol selected the opposite values. But opening the same chunk in two different ways provides enough information that, when combined, produces Alice's bank account!

Let's see the math behind it. In this setup the bank uses two cryptographic hash functions f and g, each taking two arguments.

When withdrawing the money, Alice needs to prepare k chunks to sign for each bill. She takes her bank account number, concatenates it with the bill number. Let's call the result I; its length is the same for all customers. Then she randomly generates a different binary number a_i for each chunk of the same length as I. She will be using a_i in one part of the cash, and $I \oplus a_i$ in the other, where $\oplus$ is the bitwise XOR operation.

She also generates two random numbers c_i and d_i for each chunk, of some predefined length that matches the length of the second argument of the hash function g. The purpose of these numbers is to add randomness before hashing.

Then she computes the following for each chunk i:

1. $x_i = g(a_i, c_i)$
2. $y_i = g(I \oplus a_i, d_i)$
3. $f(x_i, y_i)$

Then she obtains digital signature on each of $f(x_i, y_i)$ by sending their blinded copies to the bank.

Verifying the digital cash becomes a bit tricky. Alice would need to provide enough information to Bob for each chunk to allow him to verify the signature, but not give him both a_i and $I \oplus a_i$ for the same chunk. If Bob chooses 0, Alice gives him $I \oplus a_i$. If he chooses 1, she gives him a_i.

Part 2. What other information for a chunk i would Alice give to Bob if he chooses 0? If he chooses 1? Clearly explain how Bob would be able to verify the bank's signature on $f(x_i, y_i)$ in each case and why he wouldn't know both a_i and $I \oplus a_i$ for any chunk i.

After a merchant verifies the signature on all chunks, they send all of the information obtained from Alice to the bank in the request to deposit the cash.

The bank repeats all the verifications, deposits the cash, and saves all the information in case the bill is used again.

As we mentioned earlier, if Alice tries to spend the same bill at, say, Carol's, Bob and Carol are very likely to choose different binary values for at least one chunk, so one of them will have a_i, and the other one $a_i \oplus I$.

Part 3. How would the bank be able to determine I by combining a_i and $a_i \oplus I$ that it got from Carol and Bob?

Part 4. Explain why there is no longer a need for Bob to call the bank before accepting cash. Note that the bank knows their customers' real identity, address, etc., and also can freeze their account if they are caught cheating.

However, if this is the entire system then Alice will actually be able to cheat very easily: since the bank signs the chunks blindly, nothing forces Alice to embed her real account number! If she doesn't, opening up a chunk produces some nonsense or even implicates someone else.

Therefore there is an additional step: if Alice needs to produce k chunks, she sends $k + k'$ blinded chunks to the bank (for instance, twice as many). The bank randomly selects k' chunks and asks Alice to "unblind" them. Since the blinding factors are different for all chunks, this doesn't reveal anything about the rest.

The bank then "opens" the ones that it chose, verifies that they have the right information, signs the rest blindly, and sends them back to Alice. Of course, if any of them are incorrect, Alice is in a big trouble with the bank - or maybe even with the law.

Part 4. It is important to have both k and k' large enough to avoid cheating, but that increases the number of computations. Compute the following, clearly explain your answer (note that Alice has two possibilities of getting caught: when the bank opens the chunks and when two merchants have chosen different binary values for the same chunk that has wrong information):

1. If $k = k' = 10$ and Alice embeds 5 chunks with a wrong account number, what are her chances to get away with double-spending?

2. If $k = 10$, $k' = 5$, and Alice embeds 5 chunks with a wrong account number, what are her chances to get away with double-spending?

3. If $k = k' = 10$, and Alice embeds 1 chunk with a wrong account number, what are her chances to get away with double-spending?

Part 5. Are there any other possibilities of cheating in the system? For example, can participants collaborate to defraud the bank? If yes, how? If not, what scenarios have you considered and how is the cheating prevented? Assume that the bank is always an honest participant in all the protocols. However, the bank may be interested in tracking transactions, so all protocols should guarantee that it cannot learn who paid whom, except when someone is cheating.

Part 6. Your task is to implement this system. Since we haven't discussed generating really large prime numbers, the RSA modulus for the bank's digital signatures would consist of two 6-digit primes. You can use 103141 and 197261 for testing, but keep in mind that we might use other ones for testing. The prime numbers and the public exponent e will be taken as inputs. The public exponent to test is 65537.

Since real-life hash functions have a larger range than the product of two 6-digit primes, you will use some digits of MD5 (neither MD5 itself, nor this way of using a hash function, are secure for any real purposes, but will be fine for our demo). A typical implementation of MD5 returns the result as a hexadecimal number or in the form that can be easily converted to a hexadecimal number, so I am assuming this in the write-up below. You will need to do the following:

1. For the hash function f you will be taking the first 8 hexadecimal digits (i.e. the first 32 binary digits) of the result of MD5 on the concatenation of the two inputs, x_i and y_i, in this order. For example, if MD5 returns `E2FC714C4727EE9395F324CD2E7F331F` in hexadecimal, you will take the `E2FC714C` for your hash value (and would need to convert it to decimal before digitally signing).

2. The range of g doesn't matter as much since its output is passed to f anyway, but for simplicity we will use the last 8 hexadecimal value of MD5 on the concatenation of the inputs, so if the result of MD5 is as in the example above, the hash value will be `2E7F331F`.

3. a_i, c_i, d_i are all between 0 and $2^{32} - 1$ (inclusive) that are padded in the front with zeros to 10 decimal digits as needed when converted to strings for MD5. After concatenation this makes 20 decimal digits as an input to MD5.

4. All other inputs to the hash functions are converted to decimal (that includes the binary strings, such as a_i and $a_i \oplus I$, as well as x_i and y_i) before being passed to MD5, and they are passed as a string of decimal digits, padded with zeros if needed, just like above.

5. The customer account number is 5 decimal digits and it's set for each customer. The bill number is also 5 decimal digits and is generated by the customer for every bill they want to create. It is the same for all $k + k'$ chunks sent to the bank for a signature. Note that the bill number in this case is known to the bank since it's a part of I, but the bank will never see it again if the cash is not double-spent.

6. You can use $k = k' = 10$ for testing, but make sure it's clear how to change it if needed.

7. Your program needs to represent the bank that keeps the accounts, digitally signs cash (after performing all the verification), and then deposits the cash when sent by a merchant. It also should be checking for double-spending.

8. The program needs to represent a customer who creates digital cash and sends all of the information to the bank for signing. The customer needs to respond to the bank with the information needed to check unblinded chunks. The customer also needs to respond with the appropriate numbers to a string of 0/1 from a merchant.

9. The program needs to represent the merchant who receives the cash, performs all of the necessary checks, and then sends all the needed information to the bank (note that all of the information that the customer reveals to prove that the cash is valid is also sent to the bank).

10. The blinded chunks that are sent to the bank for signature need to be printed to a file `to_sign.txt` as one value per line. When the bank responds with the randomly chosen chunks to unblind (this could be done by a function call or also via a file), the customer writes to a file `bank_verify.txt` the blinding factors r_i for each chunk that's open and with the other information needed to verify the cash. Information for each chunk in the file is one line and includes the following, in this order: r_i, a_i, c_i, d_i. The order of chunks is the same as before, only skipping the ones that don't need to be opened. The bank reads the file, checks the values, and if it checks out writes the signed chunks to the file `signed.txt`.

11. The merchant reads the signed chunks from the file `signed.txt` and generates a string of 0/1 that tells the customer what to reveal for each chunk. The string can be sent via a file or via a function call.

12. The customer responds to the merchant's string by writing all necessary information for each chunk (in the same order as they were sent to the merchant) to a file `merch_verify.txt`. Each line starts with 0 or 1, depending on what the merchant chose to see in that chunk.

13. After verifying the cash the merchant sends all the necessary info, again with the 0/1 starting each line, to the bank in a file `deposit.txt`.

14. The bank keeps previously sent information so that they can check for double spending (it may be kept in files as well - it's up to you).

15. If at any point cheating is detected, a message explaining who cheated and the proof of cheating is displayed.

16. Your program should come with a menu asking which of the steps should be performed next. We assume that there is only one customer and one merchant (so double-spending would mean paying the same merchant twice).

Problem 7. This is an open-ended non-programming problem. You are asked to extend the ideas in Problem 6 to be used for electronic voting. There are proposed systems that use blockchain, but those approaches are quite different.

Your answer (and discussion) should use digital signatures and blind signatures. **You don't need to implement your solutions.**

Your proposal may not be perfect, that's ok. Below are some properties expected of a digital voting system, for details see [3]:

1. Only eligible voters are able to vote.
2. No voter is permitted to vote more than once.
3. No one should be able to determine the value of anyone else's vote.
4. No one can duplicate a vote.
5. No one can alter another person's vote without being detected.
6. Voters can verify that their vote has been counted.

Coming up with a complete solution is very difficult. Discuss which properties your system provides and which ones it doesn't, discuss the challenges. However, focus just on the technical aspects of the system, not its usability or social aspects (such as accessibility, trust, etc.).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[2] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners.* Springer, 2010.

[3] Bruce Schneier. *Applied Cryptography.* Wiley, 2nd edition, 1996.