Computer science problems.

About the problems. The theme of this year's problems is public key cryptography, i.e. a cryptographic approach based on a pair of matching keys, public and private, that have the property that what is encrypted by one can be decrypted only by the other one. Specifically, we will look at the RSA implementation of public key cryptography.

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. 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 1.7 or 1.8, 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. Your program may get

input from the user (i.e. it asks to enter some data and then reads it) or you may store the data in specific variables within your program. You need to clearly explain how to input or set the data.

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.

- 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.

Overview of public key cryptography and its RSA implementation. The most obvious approach to secure communications over an insecure channel is *symmetric cryptography*: the two communicating parties share a secret key. It is contrasted with *public key cryprography* (also known as *asymmetric cryptography*). In asymmetric cryptography every participant has a pair of a private and public keys: two numbers connected by a specific mathematical property. A participant's public key is publicly known, and their private key isn't known to anyone other than the participant themselves. Anything that is encrypted with their public key can be decrypted with the corresponding private key, and vice versa. Public key cryptography algorithms are based on problems that are believed to have no computationally feasible solutions: the only known approaches to these problems are by checking all possible cases out a very large set, and that takes a very long time. Here we will be exploring a widely used public key cryptography system known as RSA, by the names of its designers Ron Rivest, Adi Shamir, and Leonard Adleman.

The RSA scheme is based on computational difficulty of finding factors of a very large number: there is no known algorithm to do it other than by checking all potential factors, which becomes prohibitively slow for very large numbers (such as 50-100 decimal digits and even higher). See [**?**] or any other modern cryptography book for more details. Your goal for this problem set is to write your own implementation of RSA and its applications.

Note that you will be working with very large numbers, so you might want to choose a programming language that allows you to work with integers that are longer than 32 bits. Writing your own implementation is also fine.

Also, please review modular arithmetic **add a reference**

Problem 1, part 1. The need to transmit information between two parties secretly is quite common, especially on the Internet. Usually we assume that two parties that need to communicate secretly to each other via insecure medium (such as Internet) share a secret key and use it to encrypt information before sending it. Once the information is received by the other party, the same key is used to decrypt it. In this scenario every pair of participants share a key that is known just to the two of them.

Suppose that in an organization of 100 people every person may need to secretly communicate with any other person, so every two people come up with a secret key that only the two of them share. How many secret keys total would that organization need? What about an organization of 1000 people? Come up with the formula for an organization of N people and prove it.

Part 2. In addition to the need to communicate secretly between pairs of employees, there may be a need for three or more participants to have a secret conversation that others wouldn't be able to eavesdrop on what's being sent. Assuming that we also need a separate secret key for any group of $3 \leq k \leq N$ participants, how many keys total would we need? Notice that we do need a key for all N employees since we also need to protect information from outside observers.

Part 3. Go back to the beginning of the problem set an re-read the description of public key cryptography. Since in the case of public key cryptography each participant only needs one key set (that includes their public and private key), the total number of key sets for a group of N people is N. In addition to reducing the total number of keys, public key cryptography provides another very important benefit. What is it? Please be specific.

Problem 2. Write a function that determines if a given number is prime. Note that your numbers can be quite large. Test it on the following numbers to determine if they are prime:

1. 58148527
2. 665999921
3. 1558438050556301

Can your algorithm handle a larger prime than any of the **prime** numbers given above? If yes, give an example. Note that your algorithm must always give the correct answer (there is an area of CS known as probabilistic primality testing which determines whether a large number is prime with very high probability, but we are not considering it here).

Problem 3.

1. How long (approximately) does your algorithm run on these numbers?
2. If you consider numbers of a given length (say, 10 decimal digits), in what cases does the algorithm run the longest?
3. Approximately how does the worst-case time increase with addition of one decimal digit to the number length (for instance, going from 10 digit numbers to 11 digits)?

Please explain your answers. We are not looking for specific formulas, just for distinction between polynomial time increase and exponential increase.

Problem 4. Another common algorithm in RSA cryptography is Extended Euclidean Algorithm that is used to find the gcd (the greatest common divisor) of two integer numbers. Any cryptography book describes it, you can also look it up on wikipedia. Your task it to write your own implementation for large numbers. Test your implementation on the following pairs of numbers (output their gcd):

1. 1530864174490193 and 1324693829967097
2. 3478169421466939 and 18237769603176901

Problem 5. Extended Euclidean Algorithm is often used to find a *multiplicative inverse* of a number a modulo some number m. A *multiplicative inverse* of a modulo m is a number x such that $ax = 1 \mod m$. Equivalently this means that $ax - qm = 1$ for some q. The latter equation can be solved by Extended Euclidean Algorithms.

A multiplicative inverse of a modulo m exists if and only if $\gcd(a, m) = 1$. In this case a and m are called *coprime* numbers (also known as *relatively prime*).

Implement multiplicative inverse algorithm (you should use your Extended Euclidean Algorithm implementation as a function). If multiplicative inverse of a given number modulo a given m doesn't exists, print a message that says that. You may exit the program or return a specific value (e.g. -1), in addition to printing the message.

Test your algorithm on inputs of your choosing, and also show its results on the following pairs:

1. $a = 7, m = 12$
2. $a = 3, m = 9988426849099$
3. $a = 65537, m = 9988426849099$

Approximately, how does the running time of the algorithm scale with the number of digits of m?

Problem 6. In your program you will perform computations modulo a large number in order to encrypt and decrypt a message. Without using any libraries and functions of your language that provide modular arithmetics, please implement addition, multiplication, and raising to a power modulo m. Try to make your computations efficient in time and memory.

Please be aware that the modulus operation in several programming languages does not correspond to the mathematical definition of a remainder on negative numbers. An example of this is % operator in C, C++, and Java. If you are using such languages, you might need to implement your own modulus function.

What are the results of the following operations?

1. $100000000000000000 + 2000000000000000 \mod 9988426849099$
2. $100000000000000000 \times 2000000000000000 \mod 9988426849099$
3. $100000000000000000^{2000000000000000} \mod 9988426849099$

Again, estimate the rate at which the running time increases with each extra decimal digit of m.

Problem 7. RSA encryption keys are formed as following:

- Two distinct large primes p and q are chosen at random and are kept secret.
- Their product $n = pq$ is computed.
- An integer e is chosen so that it is coprime with $\phi(n) = (p-1)(q-1)$. ϕ is Euler's totient function, the general definition of it is given here: https://en.wikipedia.org/wiki/Euler%27s_totient_function. We refer to e as the public key exponent.
- The multiplicative inverse of e modulo $\phi(n)$ is computed, it is referred to as d, the private key exponent.

The public part of the key (known to everyone) is n and e. The private part of the key (known only to the person receiving the messages) is p, q, and d. The encryption and decryption process is described in problem 9.

Part 1. Please write a function that computes a pair of private/public RSA keys given prime numbers p, q and e. Use it to generate a key pair for

1. $p = 10040238757, q = 12604798513, e = 23,$
2. $p = 10040238757, q = 12604798513, e = 65537$

Note that a small value of e is not a problem, as long as p and q are sufficiently large and randomly chosen. In practice p and q must be much larger than in this example.

Part 2. The public exponent e is chosen to be coprime with $\phi(n) = (p-1)(q-1)$. Please explain why choosing e to be prime may not be sufficient, give an example when this choice wouldn't work. You may use small numbers for your example.

Problem 8.

1. What parts of the key generation are fast for long numbers? What parts are slower?

2. In real life RSA key generation probabilistic primality testing is used to check that p, q are indeed prime. Probabilistic primality testing is polynomial in the length of the number being checked, and gives a correct answer with a very large probability (although not equal to 1). Why is it being used, instead of the method you used in Problem 2?

Problem 9. If Alice wants to send a secret message to Bob, she encrypts it with his public key n, e. The message has to be a number m such that $1 \leq m \leq n$ and $\gcd(m, n) = 1$. In order to encrypt the message, she computes $c = m^e \mod n$.

In order to decrypt the message, Bob uses his secret key to compute c^d. The fact that $ed = 1 \mod \phi(n)$ implies that $m^{ed} = m \mod n$. For the proof see [**?**] or any other modern cryptography book.

Your goal is to implement encryption and decryption functions for large integers.

The encryption function takes the message m and the public key parameters n, e and returns c (the encrypted text). If m does not satisfy the required conditions, an error is signaled and the program stops.

The decryption function may just take c, d, and n and compute m. However, it would be more efficient to compute m using p and q to speed up the computation. This approach uses the mathematical result known as the Chinese remained theorem (see https://en.wikipedia.org/wiki/Chinese_remainder_theorem or [**?**] or any other modern cryptography book).

You can implement a computation using just n for a partial credit, or use Chinese remainder theorem for full credit.

Test your functions on encrypting a message and then decrypting it with the corresponding key. As always, submit all your test cases.

After that test your functions on the following examples and submit the results:

1. Encrypt a message $m = 123456789$ with $n = 178076698764316482877$ and $e = 65537$ (this value of e is frequently used in RSA cryptography).

2. Decrypt a message $c = 75332313748983829041 74$ with the private key $p = 694420975411, q = 12583867141, d = 72721952570797077781473$. Also compute e.

3. Decrypt a message $c = 47921891174796141377 08$ with the private key $p = 694420975411, q = 12583867141, d = 102805897573984082675 3$. Also compute e.

Problem 10. Please explain why it is fast to encrypt a message m using n and e (even when n is very large, such as 50-100 decimal digits) and it is fast to compute m from its encryption c and the private key d, but an attempt to compute m from c without knowing d would take a very long time (and would not be feasible for very large values of n).

Problem 11. Suppose Alice is a bank client, and Bob represents the bank. Since RSA scheme can only encode numbers (and they have to be less than n), Alice decides to develop a system of codes in which numbers refer to specific instructions. For instance, 2 means "transfer", 55 means "deposit", etc. Each of the instructions is sent encrypted (with Bob's public key), followed by an encrypted account number and an encrypted amount. More specifically, assuming that Bob's public key is n, e, Alice sends to Bob the following sequence of numbers:

- $i^e \mod n$,
- $a^e \mod n$,
- $k^e \mod n$,

where i is the code for one of the instructions, a is an account number, and k is the amount.

Oscar, a malicious attacker, records all messages that Alice sends to Bob. He knows that Alice has just bought a very expensive car and instructed Bob to transfer the cost of the car to the car dealer. He also tricks Alice to buy a coffee mug from his own company for $5 and got a transfer from Alice's account at Bob's bank. Oscar can send messages to Bob pretending to be Alice, and Bob wouldn't know the difference.

Is there anything Oscar can do to have a large amount of money be deposited into his account? If yes, please be very specific about what Oscar needs to do and how he knows this information. If he can't, please explain why.

Problem 12. Alice decides that perhaps she would be safer if she combines all of the message into one number and encrypt that number. She appends the instruction (such as 2 for "transfer"), the account (always a 10-digit number), and the amount to form one large decimal number $m = i \circ a \circ k$, where $\circ$ stands for string concatenation (appending one string to another). The result is still smaller than n because n is at least 50 decimal digits long. Alice sends $m^e \mod n$ to Bob.

Is there anything Oscar can do now to have money transferred to his account? Once again, Oscar has access to all encrypted messages that Alice sends to Bob and he can get Alice to make a purchase from him and to transfer the payment into his account via Bob's bank, and he can also send messages to Bob pretending to be Alice.

As before, if Oscar can do something problematic to Alice, please clearly state what it is, and if this is impossible, please explain why.

If you think that Oscar still can obtain money from Alice without her authorization, or can harm Alice's business in some other way, please explain what would you change in the use of RSA in order to prevent this from happening.

Problem 13. Public key encryption can also be used for *digital signatures*: an equivalent of real-life signatures that prove the authorship of a message. This is done by using the private key to encrypt the message. If Bob signs his message with his private key, anyone can decrypt the message using his public key, so the message is not secret. However, it is guaranteed that Bob is the author of the message since he is the only one who knows the private key (assuming that no one stole it from him, but this is always an assumption in public key encryption scheme).

Write two functions: one to digitally sign a message m with the private key p, q, d (assume that $1 \leq m \leq n$, $\gcd(m, n) = 1$), and the other one to verify that the digital signature c is indeed that of a plain-text message m and was indeed signed by the owner of the private key corresponding to n, e.

Please test your functions to check if the following examples have correct digital signatures for the given messages:

1. The signature is 111889714168449968140, the message is "12345", $n = 178076698764316482877$ and $e = 65537$.
2. The signature is 68295009307484396257, the message is "54321", $n = 178076698764316482877$ and $e = 65537$.

Problem 14. Please explain what computations in the digital signature scheme are fast, and why it is fast to sign a message knowing the private key and to verify a signature on a message m knowing only the public key, but it is prohibitively slow to forge a digital signature on a desired message m knowing only the public key.

Also explain why it is fast to obtain a valid pair of a message m and its digital signature c knowing only the signer's public key if it doesn't matter to the forger what m is. Can this be used to obtain any unauthorized information or forge a signature on any meaningful message? If yes, what can be gained? If no, why not?

Problem 15. Sometimes it is useful to obtain a digital signature on a specific message in a way that the signer doesn't know what the message is. This is known as a *blind signature*. Note that this situation is different from the one discussed in the previous problem since the message is known to the person obtaining the signature, just not to the person providing the signature. Let us assume that Alice would like Bob to sign a message m in such a way that Bob doesn't know what it is. Alice would then do the following:

1. Choose a random number r (it is called the *blinding factor*).
2. Compute $m' = mr^e \mod n$, where n, e are the public key of the signer.
3. Bob signs m' by computing $s' = (m')^d \mod n$, where d is the private exponent.
4. In order to obtain the signature on just m, Alice then computes $s'r^{-1} \mod n$, where r^{-1} is the multiplicative inverse of r modulo n. This works because $r^{ed} = r \mod n$, and r and r^{-1} cancel out.

Write and test a function for computing a message m' and another function for removing the blinding factor from the signature s'.

Show step-by-step how Alice can obtain a blind signature from Bob on the message 33333333333 using only Bob's public key, where Bob signs the message without knowing what it is. Bob's private key is $p = 694420975411, q = 12583867141, d = 7272195257079707781473$, you can compute his public key from his private key. After removing the blinding factor, verify Bob's signature on the message.

Problem 16. Blind signatures can be used for anonymous financial transactions, often referred to as *digital cash.* Suppose customers want to be able to pay vendors from their accounts at Bob's bank, but they don't want the bank to know who paid whom. Each customer can then generate \$100 "bills", each with a random number chosen from a very large space so that the probability of two numbers being the same is insignificantly small. Then each customer brings their "bills" to the bank to sign. The bank should not know the random numbers, so it would sign the bills blindly. Note that we assume that all bills have the same denominations.

The bank needs to verify that the bills are well-formed and the random numbers are all distinct. This is impossible to do if the bank cannot see what it is signing. Therefore when Alice wants to have the bank sign her bills, she prepares twice as many as she would actually need. The bank would randomly choose a half of them and open them by asking Alice to provide the blinding factors. If they are all correctly formed, the bank would sign the rest blindly since the probability of Alice cheating at that point is very small. If Alice is caught cheating she pays a large fine and is banned from using Bob's bank digital cash services. As the bank signs the bills, it also reduces the amount in Alice's account by the corresponding amount and transfers the money into a "digital cash" pool.

Now Alice can "unblind" the signed bills and use them to pay any vendors she wants. The vendors can verify that the bills are indeed guaranteed by Bob's bank which they trust, so they accept them from Alice after verifying the bank's signature. They also verify that all the bills they get from Alice are distinct since they all have different random numbers (thus Alice cannot use 5 copies of the same \$100 bill to pay \$500 to a vendor). Then the vendor brings the bills back to Bob's bank, and the bank credits their accounts accordingly. The bank records and check the random number on the bill to make sure the vendor is depositing different bills, and not copies of the same one. It also verifies that the bills haven't been used prior. However, the bank cannot tell who paid the vendor: Alice or some other customer. Thus the transaction is anonymous on the part of the buyer.

Is the digital signature scheme described above completely safe from double-spending, i.e. using the same bill twice? If yes, why? If not, how can it be cheated, and by whom? Assume that the bank is always trustworthy and follows all the rules. Other participants, however, can cheat. If you think that there is a possibility of cheating, please explain it in detail. You don't need to suggest a fix.

Problem 17. It is recommended that one uses different public/private key pairs for digitally signing messages and for encryption, especially when providing blind signatures. Please explain how a blind signatures service can be maliciously misused if this rule isn't followed. Be specific (what the attacker's goal is, what do they send to the blind signature service, what do they obtain).