Q1. In an Affine Cipher where x, y, a, b ∈ Z26, the restriction on **both ‘a’ and ‘b’ is of course that both of them can only range from 0 to 25**. There is no additional restriction on the selection of b. On the other hand, there is a restriction on the selection of a which is that **‘a’ should be chosen in such a manner that the gcd (a, 26) = 1.** This means that the value of a can only be 1,3,5,7,9,11,15,17,19,21,23 and 25. Therefore, ‘a’ only has 12 possible values.

Q2. If we perform a brute force attack on the Affine cipher, then the maximum number of pairs of (a, b) that need to tested to break the cipher can be obtained by multiplying the possible values of ‘a’ with the possible values of ‘b’. The possible values of a are 1,3,5,7,9,11,15,17,19,21,23 and 25 which is a total of 12 possible values. On the other hand, since there is no restriction in the value of ‘b’, which means it’s value can be 0, 1, 2, 3, 4, 5, 6, 7, 8 ,9 ,10, 11 ,12, 13 ,14, 15 ,16, 17, 18, 19, 20, 21, 22, 23, 24, 25 (as b ∈ Z26). Thus ‘b’ can have a total of 26 values.

**The possible pairs of (a, b) hence are 12 x 26= 312.**

Q3. If we remove the constraint where a, b ∈ Z26, then the Affine Cipher will still work. This means that value of ‘a’ and ‘b’ can be greater than 26, the only restriction being that the gcd (a,26) = 1. If we consider the key to be (29,28), then if we try and encrypt 2(plaintext C), we get

But if we try to encrypt 2 (plaintext C) with the key (29%26,28%26), which is basically the key (3,2), we obtain,

which is similar to what is obtained with the original key. If we try to decrypt 8 (cipher text I), we will obtain 2 (plaintext C) using both the keys. This is because the inverse of 3 and 29 with respect to 26 is 9. Hence, a-1(y-b) % 26=9\*(8-28) % 26=2 (plaintext C). As well for the other key, 9\*(8-2) % 26=2 (plaintext C).

Hence multiple keys will give us the same results for encryption and decryption.

For numbers a, b 26, the key will give similar results as a key of the form (a%26,b%26). **This means that keyspace of the above Affine Cipher will increase infinitely as the acceptable values of ‘a’ and ‘b’ are infinitely many.**

From the evidence above, we can deduce that plain text that has been encrypted with a key (31, 30) can be decrypted using a key (5, 40). This means that a hacker would only have to check 312 basic cases to decrypt the cipher. Thus the security of the cipher remains the same.

Q4. According to my UID: 3035435462, this is the following information I was provided by the VPL.

Your plaintext: How vexingly quick daft zebras jump!

Your ciphertext: Spt mxaziluh dfzjn qvey oxckvr gfbw!

**We can find the key without using the Brute-force approach by solving two simultaneous linear equations**.

Let us assume the key to be (a, b).

If we take two letters in the plaintext and their cipher text characters we can form two equations.

j🡪g

u🡪f

Using the corresponding character codes, we form two equations that are

Subtracting the first equation from the second, we obtain:

This can be written as:

Now we need to know the inverse of 11 with respect to 26. The inverse of 11 is 19 with respect to 26, as 11\*19=1 mod 26. Therefore 11-1=19.

Multiplying the inverse of 11 on both sides

Hence the value of a is 7. Putting this value in one of the equations we can get the value of b.

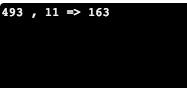
Hence the value of b is 21. **The key that has been found is therefore, (7, 21).**

Q5. In my implementation of rsahack(), where the given parameters are (n, e), firstly I have factorized ‘n’ into two prime factors. This has been done in the following way. A for-loop has been used which starts from 2 till n-1. Inside the loop, the divisibility of ‘n’ with ‘i’ is checked. If the remainder (% operator was used for this) of ‘n’ with ‘i’ is 0, this means that ‘n’ is divisible by ‘i’. After this ‘i’ is passed as a parameter for another function called isPrime(). This function checks if the given number is prime or not by testing its divisibility with other numbers from 2 till i-1. If it is prime then value True is returned, otherwise False is returned. Back in the original for loop, if ‘i’ is prime, the ‘p’ is assigned the value of ‘i’, while ‘q’ is assigned the value of n/i. Thus we have broken down n into a product of two prime numbers. After this ϕ(n) is calculated and its value is (p-1) x (q-1). Since we know that d \* e=1mod ϕ(n) , it means that d is inverse of e with respect to ϕ(n). Hence the value of the d can be calculated using the mInverse function already provided to us by passing (e, ϕ(n)) as parameters. The value returned by this function is returned as the value of d as calculated by the rsahack() function.

Q6**. The value of d that is returned by rsahack() can be checked by using the rsaencrypt()and rsadecrypt() functions.** Since we already have (n,e) given to us, we can use that to encrypt a number**. The encrypted number can then be decrypted by using the value of d calculated by the rsahack() function that we have created. If the decrypted value turns out to be original value, then the value of d that we have calculated is correct, otherwise it’s wrong.**

For example:

If the public key (493,11) is provided to us, then we can find the value of d by using the rsahack() we have created.

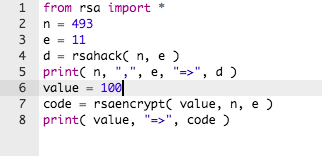
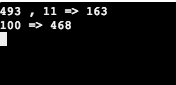
 

The value of d is found out to be 163.

The value of 100 is then encrypted.

y=xe mod n= (100)11 mod 493=468

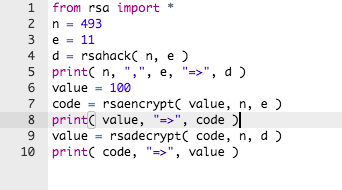
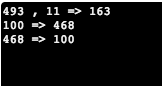
Hence the encrypted value of 100 is 468.

The encrypted value is then decrypted using the rsadecrypt() function.

x = (y)d mod n = (xe mod n)d mod n = (xed) mod n

x = (468)163 mod 493 =100

**Since the original number is obtained again, the value of d calculated by the rsahack() function is correct.**

Q7. The drawbacks of using trial division method in finding prime numbers is that we have to check the divisibility of a particular number with 2 and from there onwards all the odd numbers beginning with 3 till the square root of that number. This method is good for finding small prime numbers, but when large prime numbers are needed to be found, it’s a bad method, since a large number of divisibility tests need to be checked. This makes the trial division algorithm very inefficient, laborious and time consuming for large numbers. On the other hand, probabilistic tests like Fermat’s little theorem can find large prime numbers in a very short amount of time.

But they are not 100% accurate, as there maybe pseuodprimes (composite numbers that maybe declared as being prime by these tests), but the chances of these numbers occurring is so less, that these cases can be neglected. Also, we can reduce the probability of error by testing with different parameters. Testing again and again reduced the probability of a composite number being passed off as a prime number. Hence they are flexible.

If a composite number is tested using Trial division, we get the added bonus of finding one factor of the number, which proves that, that number is composite. But if a prime number is checked, no such proof is given. The only way we can prove it again in front of a third party is by running that process again, which is costly and time consuming. This gives us another reason why Probabilistic tests are better because Pratt certificates are certificates for primality of a number that use probabilistic tests like Lucas test which in turn uses Fermat’s little theorem to give them a certificate of primality.

Q8. Using the RSA encryption/ decryption methods taught to us, the result of encrypting 0 and 1 is undesirable as we obtain 0 and 1 again, respectively.

**Similarly, for 1, we obtain 1 again after encrypting it using the procedure given above. This makes the result undesirable for us, as a hacker would be easily able to decrypt messages that contain a lot of 0s and 1s, for example messages written in binary.** This makes the RSA semantically insecure as some information about the cipher text can be easily figured out, as the encryptions of 0 and 1 are quite evident to be 0 and 1 itself, respectively.

**One way we can overcome this problem is by adding a number ‘x’ to the number that needs to be encrypted.** After the addition has been done, we can then proceed with the encryption technique. Let us assume we add a number x=2 to all the numbers before encrypting them. So when we encrypt 1 we are actually encrypting 3. We will be using public key (493,11) and private key (493,163) for this task.

Now while decrypting 160, the following procedure is followed.

Finally, 2 is subtracted from 3 to obtain the original 1.

The problem with this method is with the encryption of 491.

Since RSA only works for inputs belong to Zn, the n here being 493. If we try to decrypt 0, we will obtain 0 again. We will encounter a similar problem with encrypting 492 as well.

Here we have selected 2 to be added to the input before being encrypted using RSA, but if we choose a larger number to be added, then all numbers equal to or greater than 493-x, will be impossible to encrypt using RSA.

In real life, a process called Padding (example: Optimal Asymmetric Encryption Padding) is used before actually encrypting with RSA. This removes the problems faced with encrypting the values of 0 and 1 and makes RSA safer as compared to earlier.