Department of Computer Science
Ashoka University

Introduction to Computer Science: CS-1102-3

RSA Extra Credit Assignment

Collaborators: None **Name:** Vishnu Varadarajan

---

1 Honor Code

I pledge that all work in this assignment is my own, or cited appropriately. I worked on this project a month ago, when Suban mentioned it in class, and thus was able to get through all this code.

2 BigNum Package

The following code file contains the BigNum Class with basic arithmetic operations, like addition, subtraction, multiplication, division, and the comparisons. Invariants are in-code and time complexities are both in-code and mentioned afterwards
bignum.py.

```
from random import randint

class BigNum:
    def __init__(self, digits) -> None:
        if type(digits) == int:
            self.list = []
            while digits > 0:
                self.list.append(digits%10)
                digits = digits//10
        elif type(digits) == list:
            self.list = digits

    @property
    def length(self):
        self.remove_redundant_zeros()
        return len(self.list)

    def remove_redundant_zeros(self):
        if self.list == []:
            return self
        i = -1
        while self.list[i] == 0:
                self.list.pop()
                i = i-1
                if abs(i) >= len(self.list):
                    break
        return self

    def __add__(self, other):
        l1 = self.length
        l2 = other.length
        i,j = 0, 0
        sum = []
        carry = 0
        self.list.append(0)
        other.list.append(0)
        # assert: l1,l2, self, other, sum are established
        # INV: after k iterations, sum contaings the sum of self and other,
        #       considered till only k digits,
        #       carry = 1 if self[k-1] + other[k-1] + carry >= 10, 0 otherwise
        while (i < l1 or j < l2):
            s = self.list[i] + other.list[j] + carry
            sum.append(s%10)
            carry = s//10
            if i < l1:
                i = i+1
            if j < l2:
                j = j+1
        if carry == 1:
            sum.append(1)
        # assert: sum contaings the sum of self and other, carry = 0
        return BigNum(sum)
        # time complexity: O(N), N = max(l1, l2)
```

```python
        # space complexity: O(N), sum contains self + other ~ N digits

    def __sub__(self, other):
        l1 = self.length
        l2 = other.length
        if l1 < l2:
            raise ValueError("Operand 1 should be greater than Operand 2")
        i,j = 0, 0
        diff = []
        borrow = 0
        self.list.append(0)
        other.list.append(0)
        # assert: l1,l2, self, other, diff are established
        # INV: after k iterations, diff contains the difference of self and other,
        #      considered upto k digits, (0<= k <= min(l1,l2))
        #      d = self[k-1] - other[k-1] - borrow;
        #      if d < 0, d = 10 + d, borrow = 1, else borrow = 0
        while (i < l1 or j < l2):
            d = self.list[i] - other.list[j] - borrow
            if d < 0:
                d = 10 + d
                borrow = 1
            else:
                borrow = 0
            diff.append(d)
            if i < l1:
                i = i+1
            if j < l2:
                j = j+1
        # assert: diff contains the difference of self and other, carry = 0
        return BigNum(diff)
        # time complexity: O(N), N = max(l1, l2)
        # space complexity: O(N), diff contains self - other ~ N digits

    def __mul__(self, other):
        l1 = self.length
        l2 = other.length
        self.list.append(0)
        other.list.append(0)
        pdt = BigNum([0])
        i = 0
        # assert l1,l2, self, other, pdt are established
        # INV: after i iterations, pdt contains the product of self and the other number
        # considered only upto i digits.
        while (i < l2):
            j = 0
            carry = 0
            current_digit = other.list[i]
            c_sum = []
            while (j< l1):
                p = (self.list[j] * current_digit) + carry
                c_sum.append(p%10)
                carry = p//10
                j += 1
            if carry != 0:
                c_sum.append(carry)

            pdt = pdt+BigNum([0]*i + c_sum)
            i += 1
        # assert pdt contains the product of the two numbers.
        return pdt
        # time complexity: O(N^2), N = max(l1, l2). technically O(N*M), N = digits in self,
    M = digits in other

    def __eq__(self, other):
        l1 = self.length
        l2 = other.length
        if l1 != l2:
            return False
        i = 0
        # assert: l1, l2, self, other are established
        # INV: after i iterations, self[0..i-1] == other[0..i-1]
        while i < l1:
            if self.list[i] != other.list[i]:
                # assert: self != other
                return False
            i += 1
        # assert: self == other
        return True
```

```
        # Time complexity: O(N), where n is size of self and other(if equal)
        # Space complexity: O(1)

    def __lt__(self, other):
        l1 = self.length
        l2 = other.length
        if l1 < l2:
            return True
        if l1 > l2:
            return False
        i = 1
        # assert: l1, l2, self, other are established
        # INV: after i iterations, relation not established <=> self[l-i .. l-1] == other[l-
    i .. l-1]
        while i <= l1:
            if self.list[-i] < other.list[-i]:
                # assert: self < other
                return True
            if self.list[-i] > other.list[-i]:
                # assert: self > other
                return False
            i += 1
        # assert: self == other
        return False
        # Time complexity: O(N), where n is size of self and other(if equal)
        # Space complexity: O(1)

    def __le__(self, other):
        l1 = self.length
        l2 = other.length
        if l1 < l2:
            return True
        if l1 > l2:
            return False
        i = 1
        # assert: l1, l2, self, other are established
        # INV: after i iterations, relation not established <=> self[l-i..l-1] == other[l-i
    ..l-1]
        while i <= l1:
            if self.list[-i] < other.list[-i]:
                # assert: self < other
                return True
            if self.list[-i] > other.list[-i]:
                # assert: self > other
                return False
            i += 1
        # assert: self == other
        return True
        # Time complexity: O(N), where n is size of self and other(if equal)
        # Space complexity: O(1)

    def __gt__(self, other):
        return not self <= other

    def __ge__(self, other):
        return not self < other

    def __floordiv__(self, other):
        # need to implement long division
        return BigNum(int(self)//int(other))

    def __mod__(self, other):
        # need to implement modulo
        return BigNum(int(self)%int(other))

    def sqr(self):
        return self*self

    def __pow__(self, exp):
        if exp == BigNum([0]):
            return BigNum([1])
        if exp == BigNum([1]):
            return self
        if exp%BigNum([2]) == BigNum([0]):
            return (self**(exp//BigNum([2]))).sqr()
        else:
            return self*((self**(exp//BigNum([2]))).sqr())

    def __int__(self):
```

```python
        j = list(map(str, self.list.copy()))
        j.reverse()
        return int(''.join(j))

def randgen(l:int, r:int) -> BigNum:
    return BigNum(randint(l, r))
```

2.1 Complexities

1. `__add__`
   - Time complexity: $O(N)$, where N is $\max(l_1, l_2)$
   - Space complexity: $O(N)$, sum contains `self` + `other` $\approx N$ digits
2. `__sub__`
   - Time complexity: $O(N)$, where N is $\max(l_1, l_2)$
   - Space complexity: $O(N)$, diff contains `self - other` $\approx N$ digits
3. `__mul__`
   - Time complexity: $O(N^2)$, where N is $\max(l_1, l_2)$; technically $O(N \times M)$, where N is the number of digits in `self` and M is the number of digits in `other`
   - Space complexity: $O(N^2)$
4. `__eq__`
   - Time complexity: $O(N)$, where N is the size of `self` and `other` if equal
   - Space complexity: $O(1)$
5. `__lt__`
   - Time complexity: $O(N)$, where N is the size of `self` and `other` if equal
   - Space complexity: $O(1)$
6. `__le__`
   - Time complexity: $O(N)$, where N is the size of `self` and `other` if equal
   - Space complexity: $O(1)$
7. `__gt__`
   - Defined as `not self <= other`, So time complexity is $O(N)$
   - Space complexity: $O(1)$
8. `__ge__`
   - Defined as `not self < other`, So time complexity is $O(N)$
   - Space complexity: $O(1)$
9. `__init__`
   - Time complexity: $O(N)$, where N is the number of digits in the integer or list
   - Space complexity: $O(N)$, to store the digits
10. `length`
    - Time complexity: $O(N)$, due to the call to `remove_redundant_zeros`
    - Space complexity: $O(1)$
11. `remove_redundant_zeros`
    - Time complexity: $O(N)$, where N is the number of trailing zeros
    - Space complexity: $O(1)$
12. `__floordiv__`
    - Placeholder implementation with Python's `int` conversion (not real complexity of `BigNum` division)

- Time complexity: $O(N)$, if using Python's `int`
- Space complexity: $O(N)$, for converting `BigNum` to `int` and back

13. `__mod__`
    - Placeholder implementation with Python's `int` conversion (not real complexity of `BigNum` modulo)
    - Time complexity: $O(N)$, if using Python's `int`
    - Space complexity: $O(N)$, for converting `BigNum` to `int` and back
14. `sqr`
    - Time complexity: $O(N^2)$, as it calls `__mul__`
    - Space complexity: $O(N^2)$
15. `__int__`
    - Time complexity: $O(N)$, for converting the list of digits to an integer
    - Space complexity: $O(N)$, for storing the string representation before conversion
16. `randgen`
    - Time complexity: $O(N)$, for generating a random number with N digits
    - Space complexity: $O(N)$, for storing the digits of the random number

3 `RSA` Implementation

The following code implements the RSA algorithm. I have written functions for each subpart separately, and called them in the `keygen` function. Below are the lines where the subpart's helper functions are written and called.

1. Generate two large prime numbers p and q, and using them, generate $N = p \cdot q$.
    - Helper function `primegen` on line: 75.
    - Called on line: 85,86
2. Generate $\Phi(N) = (p-1)(q-1)$.
    - Calculated on line: 88.
3. Generate a value e which is co-prime to $\Phi(N)$, $2 < e < \Phi(N)$. This must be done by creating a function that accepts $\Phi(N)$ as an argument and returns e. e is the public key.
    - Calculated on line: 89-90.
4. Generate a value d which is the modular multiplicative inverse of $e \mod \Phi(N)$. d is the private key.
    - Helper function `mod_inv` on line: 26.
    - Called on line: 91.
5. Create an encryption function that accepts the public key, N, and a message m and outputs a ciphertext c.
    - Helper function `encrypt` on line: 97.
    - Called on line: 127.
6. Create a decryption function that accepts the private key, N, and a ciphertext c and outputs the message m.
    - Helper function `decrypt` on line: 106.
    - Called on line: 136.
7. Hashing
    - Helper function for hashing on line: 116.
    - Helper function for signature verification on line: 120.
    - Called on line: 134-135.

```python
from bignum import BigNum, randgen
from random import randint
import hashlib

def gcd(a, b): # euclid gcd
    # INV: gcd(a,b) = gcd(b, a%b)
    while b != 0:
        a, b = b, a%b
    return a

def eegcd(a, b): #extended euclid gcd
    r, r1 = a, b
    s, s1 = 1, 0
    t, t1 = 0, 1
    # assert: r_0, r_1, s_0, s_1, t_0, t_1 are established
    # INV: after i iterations, q_i = r_(i-1)//r_i; s_i = s_(i-1) - q_i*s_(i-2); t_i = t_(i-1) - q_i*t_(i-2); r_i = s_i*a + t_i*b
    #      gcd(a,b) = gcd(r_(i-1), r_i)
    while r1 != 0:
        q = r//r1
        s, s1 = s1, s - q*s1
        t, t1 = t1, t - q*t1
        r, r1 = r1, r - q*r1
    # assert: r_0 = gcd(a,b), s_0*a + t_0*b = r_0
    return r, s, t

def mod_inv(a, m): # modular inverse
    # bezout coefficients are the modular inverses with respect to the other number.
    x = eegcd(a, m)[1]
    return x%m

def expmod(b, e, m): # modular exponentiation
    def sqr(x):
        return x*x
    if (e == 0):
        return 1
    elif (e%2 == 0):
        return sqr(expmod(b,e//2,m)) % m
    else:
        return b*sqr(expmod(b,e//2,m)) % m
    # proved by induction.

def miller_rabin(n, k): # miller rabin primality test
    if n == 2:
        return True
    if n%2 == 0:
        return False

    s, q = 0, n-1
    # INV: (n-1) = q*2^s
    while q%2 == 0:
        s += 1
        q //= 2
    # assert: (n-1) = q*2^s, q is odd

    # INV: after i iterations, n is prime with probability p^i
    for i in range(k):
        a = randint(1, n-1)
        b = expmod(a, q, n)
        if b == 1 or b == n-1:
            # assert: n is probably prime
            continue

        for j in range(s-1):
            b = (b**2) % n
            if b == 1:
                # assert: n is composite
                return False
            elif b == n-1:
                break
        else:
            # assert: n is composite
            return False
    return True

def primegen(bits):
    # INV: after i iterations, loop ongoing <=> num is composite
    while True:
        num = randint(2**(bits-1), 2**bits-1)
```

```python
        if miller_rabin(num, 40):
            break
    # assert: num is prime
    return num

def keygen(bits=32): # RSA key generation
    p = primegen(bits)
    q = primegen(bits)
    n = int(BigNum(p)*BigNum(q))
    phi_n = int((BigNum(p)-BigNum(1))*(BigNum(q)-BigNum(1)))
    e = randint(2, phi_n)
    # INV: loop ongoing <=> gcd(e, phi_n) != 1
    while gcd(e, phi_n) != 1:
        e = randint(2, phi_n)
    # assert: gcd(e, phi_n) = 1
    d = mod_inv(e, phi_n)
    return p, q, n, phi_n, e, d

def encrypt(e, N, m):
    cipher = ""
    # INV: after i iterations, cipher = enctrypted(m[0..i-1]) where each character is encrypted using raising the asscii representation of the character to power e mod N
    for x in m:
        letter = ord(x)
        cipher += str(pow(letter, e, N)) + " "
    # assert: cipher = encrypted(m)
    return cipher

def decrypt(d, N, cipher):
    m = ""
    parts = cipher.split()

    for part in parts:
        if part:
            charecter = int(part)
            m += chr(pow(charecter, d, N))

    return m
def hash(message, n):
    # hash is essentially a many to one function. getting the hash from message is easy but getting the message from hash is hard.
    # the %n is used to make the hash is within the range of n, which makes things easier
    return str(int(hashlib.sha256(message.encode()).hexdigest(), 16)%n)
def verify_signature(message, signature, e, n):
    if hash(message, n) == decrypt(e, n, signature):
        print("Signature verified!!!!\n")

p, q, n, phi_n, e, d = keygen(32)
print(f'p: {p}\n q: {q}\n n: {n}\n phi_n: {phi_n}\n e: {e}\n d: {d}\n')
message = "yo what thw fcuk is this shit bro"
cipher_text = encrypt(e, n, message)

# idea is to encrypt the hash using the private key, so it can be decrypted using the public key. Upon decrypting the hash, we can verify by applying sha256 on the original message, an confirming if the same hash is generated. Since only I own the private key, and know the message, only I can hash, encode, and send the a valid signature.
signature = encrypt(d, n, hash(message, n))

print("Message:", message, '\n')
print("Cipher text:", cipher_text, '\n')
print("Signature: ", signature)
verify_signature(message, signature, e, n)
print("Decrypted message:", decrypt(d, n, cipher_text))
```

3.1 Complexities

1. gcd
   - Time complexity: $O(\log(\min(a, b)))$
   - Space complexity: $O(1)$
2. eegcd
   - Time complexity: $O(\log(n))$ where n is $\min(a, b)$
   - Space complexity: $O(1)$
3. mod_inv

- Time complexity: $O(\log(n))$, where n is $\min(a, b)$
- Space complexity: $O(1)$

4. `expmod`
   - Time complexity: $O(\log(e))$, where e is the exponent
   - Space complexity: $O(\log(e))$

5. `miller_rabin`
   - Time complexity: $O(k \log^3(n))$, where k is the number of iterations and n is the number to be tested
   - Space complexity: $O(\log(n))$

6. `primegen`
   - Time complexity: Depends on the density of prime numbers; on average $O(k \log^3(n))$ for each candidate (dominated by `miller_rabin`). also assumes randint is $O(1)$
   - Space complexity: $O(1)$

7. `encrypt`
   - Time complexity: $O(N \log(e))$, where N is the length of the message and e is the exponent
   - Space complexity: $O(N)$, where N is the length of the message

8. `decrypt`
   - Time complexity: $O(N \log(d))$, where N is the number of parts in the cipher and d is the exponent
   - Space complexity: $O(N)$, where N is the number of parts in the cipher

9. `hash`
   - Time complexity: depends on complexity of hashing function. SHA-256 is $O(N)$, where N is the length of the message
   - Space complexity: $O(1)$

10. `verify_signature`
    - Time complexity: $O(M + N \log(e))$, where M is the length of the message, N is the length of the signature, and e is the exponent
    - Space complexity: $O(1)$

4 Crack

By factoring N, we can find p and q, and then calculate $\Phi(N)$. With $\Phi(N)$, we can find d by calculating the modular multiplicative inverse of e. With d, we can decrypt the message. I also wrote a code to do this, using Pollards Rho Algorithm to factorize N. The code is below.

```
import random
from rsa_functions import gcd, mod_inv, decrypt

# p, q, n, phi_n, e, d
    =,24436991318977,34055572380119,832215726615760893022218263,832215726615702400458519168,76060994689140



def crack(e, n, Cipher):# take d n and cipher later
    p = pollards_rho(n)
    q = n//p
    phi_N = (p-1)*(q-1)
    d = mod_inv(e, phi_N)
    print('priv_key =', (d, n))
    print('Decrypted text:', decrypt(d, n, Cipher))



def pollards_rho(n):
    x = random.randint(1, n-1) # atarting value
    y = x
    c = random.randint(1, n-1) # constant
```

```
    d = 1
    g = lambda x: (x**2 + c) % n
    # assert: x, y, c, d, g are established
    # INV: after i iterations, x = g^i(x), y = g^(2i)(y), d = gcd(|x-y|, n)
    while d == 1:
        x = g(x)
        y = g(g(y))
        d = gcd(abs(x-y), n)
    if d == n:
        # assert: faliure, change constant and try again
        return pollards_rho(n)
    # assert: d is a non-trivial factor of n
    return d

e = 76060994689140911975842253
n = 83221572661576089302218263
# print(pollards_rho(n))

message = 'Hello World'
Cipher = '77323734602721437154790179 47435119572565254817974l842 48141802633665840964860680б
     48141802633665840964860680б 74559439893545716311495б154 69952486900290850938741400l
    81706876278521878658614073 8 74559439893545716311495б154 30608090992908671813267027l
    48141802633665840964860680б 2061025539421803230014424 82'

crack(e, n, Cipher)
```