НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
КАФЕДРА ІНФОРМАЦІЙНОЇ БЕЗПЕКИ

«Криптографія»

Комп’ютерний практикум №4

Студенти: Маврикін Едуард
Слобода Ірина
Група: ФБ-25
Варіант 2

Київ – 2025

Вивчення криптосистеми RSA та алгоритму електронного підпису; ознайомлення з методами генерації параметрів для асиметричних криптосистем

Мета та основні завдання роботи: ознайомлення з тестами перевірки чисел на простоту і методами генерації ключів для асиметричної криптосистеми типу RSA; практичне ознайомлення з системою захисту інформації на основі криптосхеми RSA, організація з використанням цієї системи засекреченого зв'язку й електронного підпису, вивчення протоколу розсилання ключів.

1. Написати функцію пошуку випадкового простого числа з заданого інтервалу або заданої довжини, використовуючи датчик випадкових чисел та тести перевірки на простоту. В якості датчика випадкових чисел використовуйте вбудований генератор псевдовипадкових чисел вашої мови програмування. В якості тесту перевірки на простоту рекомендовано використовувати тест Міллера-Рабіна із попередніми пробними діленнями. Тести необхідно реалізовувати власноруч, використання готових реалізацій тестів не дозволяється.

Будемо реалізовувати функцію пошуку випадкового простого числа з заданого інтервалу, у якості датчика випадкових чисел будемо використовувати метод random.randint(). Для тесту перевірки на простоту використовуємо тест Міллера-Рабіна із попереднім пробним діленням. Для цього спочатку треба реалізувати тест пробних ділень, який використовує ознаку Паскаля. Знаємо, що натуральне число N (число яке ми тестуємо на простоту) записане у системі числення з основою B (у нашому випадку десятична система численна => основа 10):

$$N = \overline{a_t a_{t-1} \ldots a_1 a_0} = a_t B^t + a_{t-1} B^{t-1} + \ldots + a_1 B + a_0,$$

Це реалізовано у частині кода, яка виділена блакитним. Ми записуємо залишки від ділення у num_parts =[]. Після чого обчислюємо послідовність

де $r_i = B^i \bmod m$. - виділено зеленим та наприкінці розраховуємо залишок за формулою:

$$N \equiv \sum_{i=0}^{t} a_i r_i \pmod{m},$$

```python
def pascal(N, m, B=10):
    num_parts = []
    while N > 0:
        num_parts.append(N % B)
        N //= B

    r = [1]
    for i in range(1, len(num_parts)):
        r.append((r[i-1] * B) % m)

    lishok = sum(d * r[i] for i, d in enumerate(num_parts)) % m
    return lishok == 0
```

Далі виконуються пробні ділення на малих числах з використанням функції pascal():

```python
def trial(N):
    small = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    for prime in small:
        if pascal(N, prime):
            return False
    return True
```

Тест Міллера-Рабіна реалізовано відповідно до трьох кроків з методички:

4.3. Імовірнісний тест Міллера-Рабіна

Оберемо деяке число $k \in N$.

Крок 0. Знаходимо розклад $p-1=d\cdot 2^s$ та встановлюємо лічильник у 0.

Крок 1. Вибираємо випадкове число $x \in N$ з інтервалу $1<x<p$ (незалежне від раніше обраних x). За допомогою алгоритму Евкліда знаходимо $\gcd(x,p)$. Якщо $\gcd(x,p)=1$, то переходимо до кроку 2. Якщо $\gcd(x,p)>1$, то p – складене число, алгоритм завершує свою роботу.

Крок 2. Перевіряємо, чи є p сильно псевдопростим за основою x:

2.1 Якщо $x^d \bmod p = \pm 1$, то p є сильно псевдопростим за основою x, інакше

2.2 Для всіх послідовних r від 1 до $s-1$ виконати такі дії:

Обчислити $x_r = x^{d\cdot 2^r} \bmod p$ (тобто $x_r = x_{r-1}^2 \bmod p$).

Якщо $x_r = -1$, то p є сильно псевдопростим за основою x,

інакше якщо $x_r = 1$, то p не є сильно псевдопростим за основою x,

інакше перейти до наступного значення r.

2.3 Якщо за кроки 2.1 та 2.2 не було встановлено, що p є сильно псевдопростим за основою x, то p не є сильно псевдопростим за основою x.

Якщо p виявилось не псевдопростим за основою x, то p – складене число, алгоритм завершує свою роботу; інакше лічильник збільшується на 1.

Крок 3. Якщо лічильник менший за k, то переходимо до кроку 1, інакше алгоритм завершує роботу.

```python
def miller_rabin(p, k=10):
    if p < 2 or p % 2 == 0:
        return p == 2
    s = 0
    d = p - 1
    while d % 2 == 0:
        d //= 2
        s += 1
    def check(x):
        x = pow(x, d, p)
        if x == 1 or x == p - 1:
            return False
        for _ in range(s - 1):
            x = pow(x, 2, p)
            if x == p - 1:
                return False
        return True
    counter = 0
    for _ in range(k):
        x = random.randint(2, p - 2)

        if gcd(x, p) != 1:
            return False
        if check(x):
            return False
        counter += 1
    if counter < k:
        return False
    return True
```

Далі вже спробуємо знайти просте число на заданому інтервалі. Почнемо з малих значень, нехай це буде інтервалі 18-169:

```
lab4/Mavrykin_FB_25_Sloboda_FB-25_cp4/lab4.py5-26>
Початок діапазона: 18
Кінець діапазона: 169
Випадкове просте число у заданому діапазоні:  37
```

Тепер перевіримо на більшому інтервалі:

```
lab4/Mavrykin_FB_25_Sloboda_FB-25_cp4/lab4.py
Початок діапазона: 2
Кінець діапазона: 2567
Випадкове просте число у заданому діапазоні:  157
```

2. За допомогою цієї функції згенерувати дві пари простих чисел p, q і 1 1 p , q довжини щонайменше 256 біт. При цьому пари чисел беруться так, щоб pq ≤ p1q1 ; p і q – прості числа для побудови ключів абонента A, 1 p і q1 – абонента B.

Генеруємо пари на основі попередніх функцій та встановлюємо зазначені початок та кінець інтервалу на якому будуть обиратись прості числа (для того щоб вони були не менше 256 біт).

```
70
71  def gen_pairs(bit_length=256):
72      start = 2**(bit_length - 1)
73      finish = 2**bit_length - 1
74
75      p = find(start, finish)
76      q = find(start, finish)
77      p1 = find(start, finish)
78      q1 = find(start, finish)
79
80      while p * q > p1 * q1:
81          p1 = find(start, finish)
82          q1 = find(start, finish)
83
84      """print("\033[36m" + "~~~" * 50 + "\033[0m")
85      print("Перша пара (абонент A):")
86      print(f"p = {p}, q = {q}")
87      print("\n")
88      print("\nДруга пара (абонент B):")
89      print(f"p1 = {p1}, q1 = {q1}")"""
90      return (p, q), (p1, q1)
```

Перевіримо виконання:

```
Перша пара (абонент A):
p = 971652596552487664067927234091042834621896767319672045737330295826831757570 63, q = 641115598646838139378744366904272872080319119063445416466101247688664612446 43

Друга пара (абонент B):
p1 = 109065886204146938785345579311481385468433347626290125070149666341918550930653, q1 = 850124855526537867193923713651173204989923898624497934174118914003754211180 27
```

3. Написати функцію генерації ключових пар для RSA. Після генерування функція повинна повертати та/або зберігати секретний ключ (d, p,q) та відкритий ключ (n,e) . За допомогою цієї функції побудувати схеми RSA для абонентів A і B – тобто, створити та зберегти для подальшого використання відкриті ключі (e,n) , (,) 1 n1 e та секретні d і d1 .

Для реалізації генерації ключових пар, згідно до теоретичних вимог, нам необхідні деякі математичні функції – пошук оберненого елемента за модулем, який в свою чергу потребує розширеного алгоритму Евкліда та функція Ойлера. Реалізацію пошуку оберненого елемента та розширеного алгоритму Евкліда візьмемо з комп'ютерного практикуму 4. Функцію Ойлера реалізовано відповідно до формули

$$\varphi(n) = (p-1)(q-1).$$

```
91  def gcd_evc(a, b):
92      if b == 0:
93          return a, 1, 0
94      gcd, x1, y1 = gcd_evc(b, a % b)
95      x = y1
96      y = x1 - (a // b) * y1
97      return gcd, x, y
98
99  def obratn(a, m):
100     gcd, x, _ = gcd_evc(a, m)
101     if gcd != 1:
102         return None
103     return x % m
104
105 def oiler(p, q):
106     return (p - 1) * (q - 1)
107
```

Далі реалізовано саму функцію генерації RSA відповідно до логіки, яка розписана у теоретичних відомостях:

```
108 def rsa_pairs(bit_length=256):
109     (p, q), _ = gen_pairs(bit_length=256)
110     n = p * q
111     phi = oiler(p, q)
112     e = 2 ** 16 + 1
113     if gcd(e, phi) != 1:
114         raise ValueError("e повинно бути взаємнопростим з функцією Ойлера")
115     d = obratn(e, phi)
116     return (d, p, q), (n, e)
117
```

Перевіримо як воно працює:

```
Абонент A:
Секретний ключ: d = 4166790134183025974843296610848008719477998712640271663695187192092087175724738518555588273032078673549919846897155457384196673716354386346684934642798877
Відкритий ключ: n = 6229416361155028248130691652830849452481456341225573922111264069350984721479098893229803224085313985964787284065112752112809614648990090582384730778163509, e = 65537
p = 971652596552487664067927234091042834621896767319672045737330295826831757570 63, q = 641115598646838139378744366904272872080319119063445416466101247688664612446 43

Абонент B:
Секретний ключ: d1 = 1466209359470409209492333421503550160045346686410982640814453584388159383515888852742453912122306911106531117739911414226729531280344313254295391804655357
Відкритий ключ: n1 = 7982966087198820998795302437906302802931950302177998615191230751852355364084319932456755588270721771185179981961732597162908918659419512915830727688311941, e1 = 65537
p1 = 90936855309195565244845209614136081868697826339489742310912276052495012170059, q1 = 87785816433346368380636768227650040321292858445868933730245755109536763397999
```

4. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів А і В. Кожна з операцій (шифрування,

розшифрування, створення цифрового підпису, перевірка цифрового підпису) повинна бути реалізована окремою процедурою, на вхід до якої повинні подаватись лише ті ключові дані, які необхідні для її виконання. За допомогою датчика випадкових чисел вибрати відкрите повідомлення M і знайти криптограму для абонентів A и B, перевірити правильність розшифрування. Скласти для A і B повідомлення з цифровим підписом і перевірити його.

Операцію шифрування реалізовано за логікою відповідною до формули:

```
134    def encrypt(message, public):
135        n, e = public
136        return pow(message, e, n)
```

$$C = M^e \bmod n \ ,$$

Операція розшифрування також реалізована відповідно до формули:

```
138    def decrypt(cipher, secret):
139        d, p, q = secret
140        n = p*q
141        return pow(cipher, d, n)
```

$$M = C^d \bmod n \ .$$

Створення цифрового підпису

```
143    def sign(message, secret):
144        d, p, q = secret
145        n = p*q
146        return pow(message, d, n)
```

$$S = M^d \bmod n$$

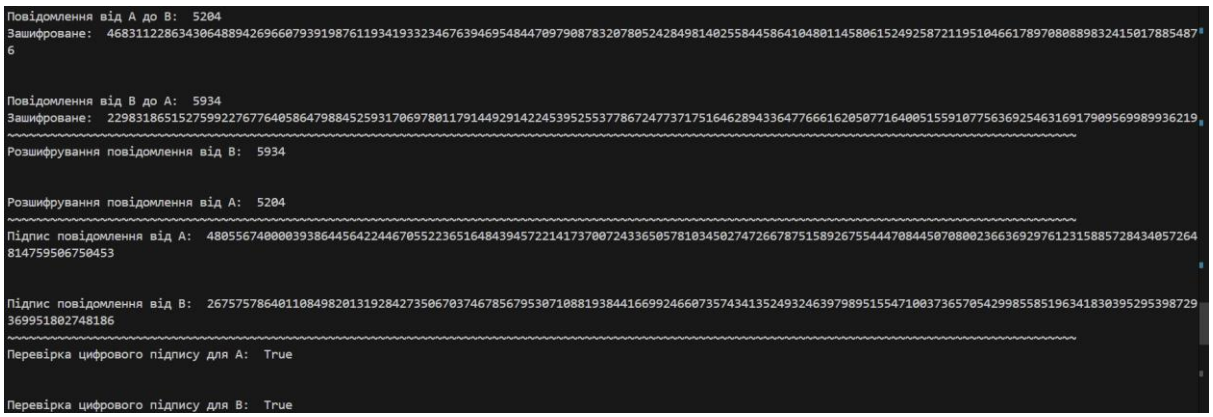
Перевірка цифрового підпису також відбувається за формулою:

```
148    def verify(message, sign, public):
149        n, e = public
150        return pow(sign, e, n) == message
```

$$M = S^e \bmod n,$$

Тобто, якщо ця рівність виконується, то повідомлення не було змінено і відповідає початковому стану.

Переглянемо, чи працює:



5. За допомогою раніше написаних на попередніх етапах програм організувати роботу протоколу конфіденційного розсилання ключів з підтвердженням справжності по відкритому каналу за допомогою алгоритму RSA. Протоколи роботи кожного учасника (відправника та приймаючого) повинні бути реалізовані у вигляді окремих процедур, на вхід до яких повинні подаватись лише ті ключові дані, які необхідні для виконання. Перевірити роботу програм для випадково обраного ключа 0 < k < n.

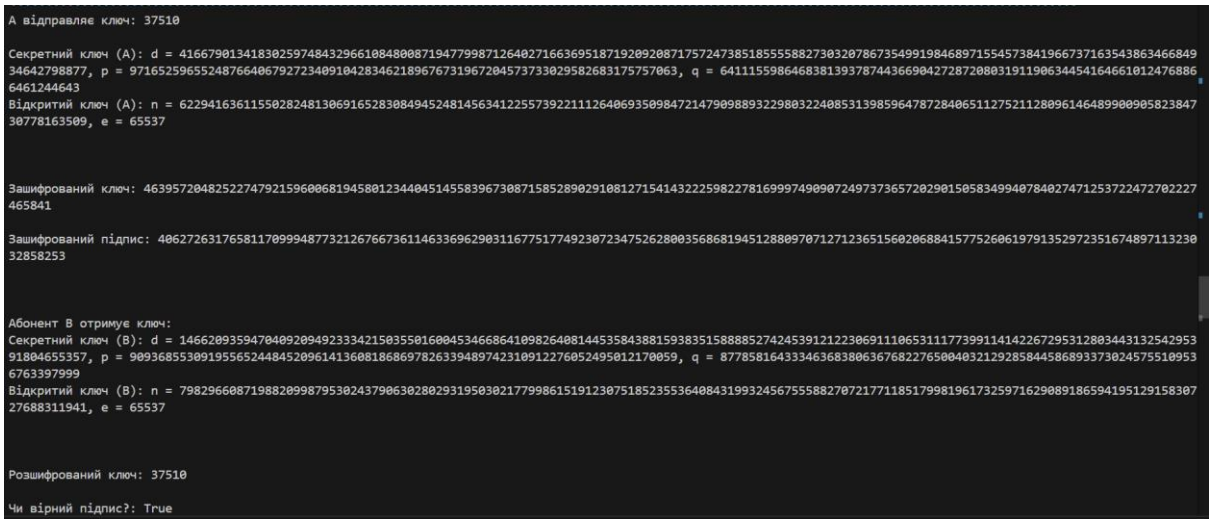
```
178
179 def send_key(k, sender_se, sender_pub, receiver_pub):
180     en_k = encrypt(k, receiver_pub)
181     signa = sign(k, sender_se)
182     en_s = encrypt(signa, receiver_pub)
183     return en_k, en_s
184
185 def receive_key(en_key, en_signa, receiver_se, sender_pub):
186     de_k = decrypt(en_key, receiver_se)
187     de_s = decrypt(en_signa, receiver_se)
188     valid_s = verify(de_k, de_s, sender_pub)
189     return de_k, valid_s
190
191 k = random.randint(1, public_b[1] - 1)
192 en_k, en_s = send_key(k, secret_a, public_a, public_b)
193 de_k, valid_s = receive_key(en_k, en_s, secret_b, public_a)
```

Тут ми генеруємо значення k відповідно до вимог – 0 < k < n (тобто менше за відкритий ключ, тому у коді граничний розмір генерації – public_b[1]-1). Також створюємо функції send_key та receive_key, які будуть відповідати за отримання та надсилання ключа.

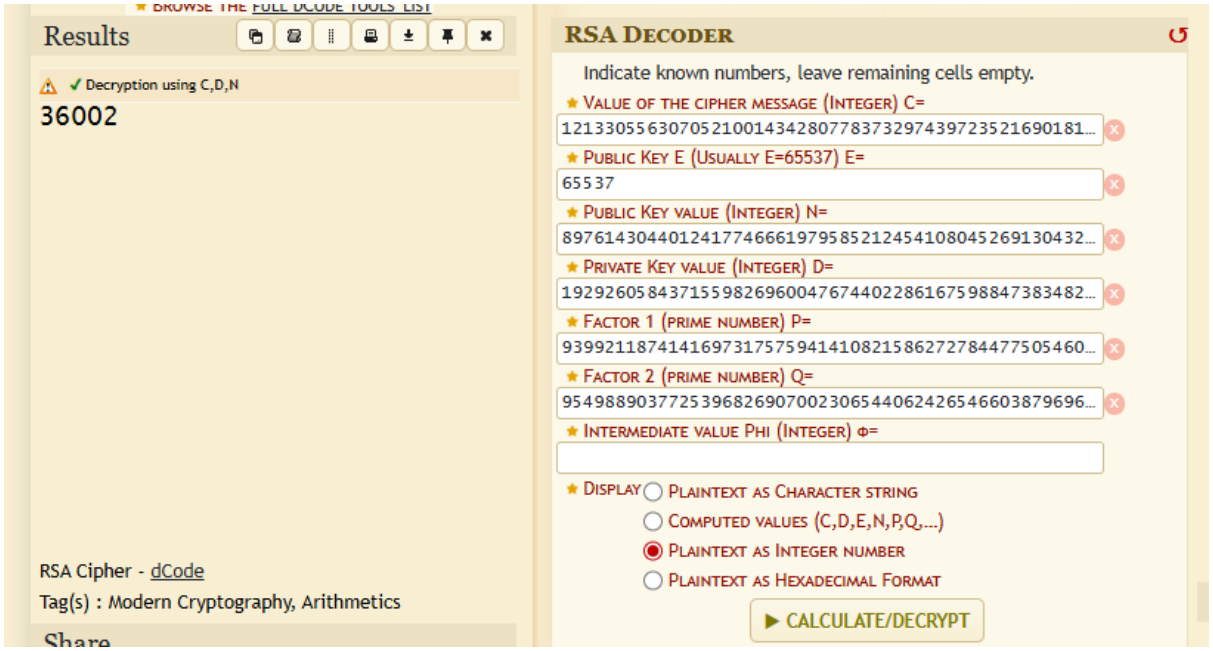
Тут для відправки ключа необхідно спочатку зашифрувати k, створити підпис та зашифрувати підпис.

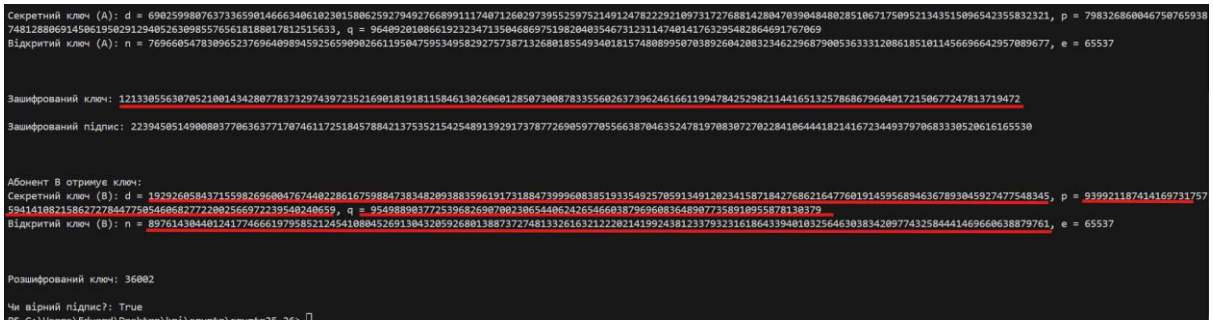
А для отримання ключа необхідно спочатку розшифрувати ключ, потім розшифрувати підпис та перевірити його.

Переглянемо як працює:



Тепер перевіримо це з використання сервісу https://www.dcode.fr/rsa-cipher





Бачимо, що все коректно працює.

Висновок: під час комп’ютерного практикуму ознайомились з механікою шифрування RSA. На основі цього ми реалізували відповідні функції (шифрування, дешифрування, створення цифрового підпису, його перевірку, надсилання та отримання ключа). Також набуті навички були протестовані на сторонньому ресурсі