Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное   
учреждение высшего образования

«Новосибирский государственный технический университет»



Кафедра теоретической и прикладной информатики

### Лабораторная работа № 5 по дисциплине «Программные Средства Защиты Информации»

### Обмен ключами по схеме Диффи-Хеллмана

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Факультет: | ПМИ |  |  |
| Группа: | ПМИМ-01 |  |  |
| Студенты: | Ершов П. К.  Малышкина Е. Д.  Слободчикова А. Э. |  |  |
| Бригада: | 2 |  |  |
| Преподаватель: | Авдеенко Т. В. |  |  |

Новосибирск

2021

1. **Цель работы**

Освоить методы генерации больших простых чисел и методы проверки больших чисел на простоту. Познакомиться с теоремой Эйлера, научиться строить первообразные корни по модулю n. Изучить схему обмена ключами Диффи-Хеллмана.

Примечание: В данной лабораторной работе под большими числами будем понимать числа, превышающие 264.

1. **Задание**
2. Реализовать приложение для шифрования, позволяющее выполнять следующие действия:
   1. Генерировать большие простые числа:
   2. программа по заданным (количество проверок в тесте Рабина-Миллера) и (количество бит) должна генерировать простое n-битное число, отображая при этом, сколько итераций алгоритма генерации простого числа потребовалось выполнить для его генерации и сколько времени было затрачено на это;
   3. программа по заданным границам диапазона должна выводить все простые числа из этого диапазона, отображая время, затраченное на генерацию всех чисел;
   4. Определять для заданного числа первые 100 первообразных корней, отображая при этом суммарное время, затраченное программой на их поиск.
   5. Моделировать обмен ключами между абонентами по схеме Диффи-Хеллмана. Программа должна получать большие простые числа 𝑋A, 𝑋B и n случайным образом с помощью алгоритма генерации простого числа, а также предоставлять пользователю возможность задавать их.
3. С помощью реализованного приложения выполнить следующие задания:
   1. Протестировать правильность работы разработанного приложения.
   2. Сделать выводы о проделанной работе.
4. **Исследования**

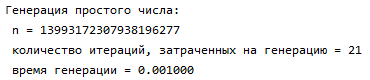


Рисунок 1. Генерация простого числа

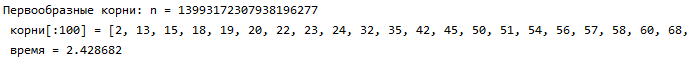


Рисунок 2. Первые сто первообразных корней числа

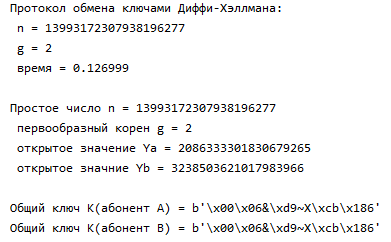


Рисунок 3. Работа протокола обмена ключами Диффи-Хеллмана

1. **Выводы**

Реализованные алгоритмы позволяют достаточно быстро получить результат, однако они сильно зависят от алгоритма факторизации, который в свою очередь выбирается в зависимости от рабочих диапазонов больших чисел.

В данной реализации, факторизация через перебор делителей часто приводит к зависанию программы т.к. при обходе всех делителей от 2 до √n последний делитель может быть ближе к √n чем к 2, нахождение которого через обход всех значений в порядке их возрастания выполняется довольно долго.

Алгоритм Диффи-Хеллмана хорошо выполняет задачу по получению общего секретного ключа для двух и более абонентов используя незащищенный от прослушивания, но, что важно, защищенный от модификации передаваемых данных, канал связи.

1. **Код программы**

**import** time  
**import** random  
**import** typing **as** ty  
**import** numpy **as** np  
  
*# get all prime numbers below n***def** primesfrom2to(n: int) -> ty.List[int]:  
 sieve = np.ones(n // 3 + (n % 6 == 2), dtype=np.bool\_)  
 **for** i **in** range(1, int(n \*\* 0.5) // 3 + 1):  
 **if** sieve[i]:  
 k = 3 \* i + 1 | 0x1  
 sieve[k \* k // 3::2 \* k] = **False** sieve[k \* (k - 2 \* (i & 0x1) + 4) // 3::2 \* k] = **False  
 return** np.r\_[2, 3, (3 \* np.nonzero(sieve)[0][1:] + 1 | 0x1)].tolist()  
  
dividers = primesfrom2to(2000)  
  
*# miller-rabin primality test***def** mrprimality(p: int, \*, t: int = 1) -> bool:  
 **if** p == 1 **or** p **in** dividers: *# fast return* **return True  
 if not** p & 0x1: *# if even* **return False** b = 0  
 m = p - 1  
  
 **while** m % 2 == 0: *# find pow of 2* m //= 2  
 b += 1  
  
 **for** \_ **in** range(t):  
 a = random.randrange(2, p)  
 z = pow(a, m, p)  
  
 **if** z **in** (1, p - 1):  
 **continue  
  
 for** j **in** range(b):  
 z = pow(z, 2, p)  
 **if** z == p - 1:  
 **break  
 else**:  
 **return False  
 return True***# some stupid factorization***def** factor(n: int) -> ty.Generator[int, **None**, **None**]:  
 **if** mrprimality(n):  
 **yield** n  
 **return  
  
 while** n % 2 == 0:  
 n //= 2  
 **yield** 2  
  
 **if** n == 1:  
 **return  
  
 for** x **in** range(3, int(np.sqrt(n)) + 1, 2):  
 **while** n % x == 0:  
 n //= x  
 **yield** x  
  
 **if** n == 1:  
 **return  
  
 if** mrprimality(n):  
 **break  
 yield** n  
  
*# squeeze same numbers in array***def** squeeze(arr: ty.List[int]) -> ty.List[int]:  
 **if** len(arr) < 2:  
 **return** arr  
 arr.sort()  
  
 cur = arr[0]  
 i = 0; j = 1  
 **while** j != len(arr):  
 **if** arr[j] == cur:  
 arr[i] \*= arr[j]  
 arr.pop(j)  
 **else**:  
 i += 1  
 j += 1  
 **return** arr  
  
*# euler's function***def** euler(n: int) -> int:  
 **if** n < 3:  
 **return** 1  
  
 *# simple check if n is prime* **if** n **in** dividers:  
 **return** n - 1  
  
 res = n *# type: float* factors = tuple(factor(n))  
 **for** x **in** set(factors):  
 **for** \_ **in** range(factors.count(x)):  
 n //= x  
 res -= res / x  
 **if** n > 1:  
 res -= res / n  
 **return** int(res)  
  
*# simple primary number generator***def** generate(n: int, \*, k: int = 5) -> ty.Tuple[int, int]:  
 *"""generate a prime number. returns (number, attempts)"""* fails = 0  
 **while True**:  
 *# generate n random bits and set first and last ones to 1 which will ensure to be n bits long and odd* num = random.getrandbits(n) | 0x1 << n - 1 | 0x1  
  
 **for** x **in** dividers: *# check num to be divided by any prime number below 2000* **if** num % x == 0 **and** num // x != 1:  
 **break  
 else**:  
 *# check miller-rabin primality test k times* **if** mrprimality(num, t=k):  
 **return** num, fails  
 fails += 1  
  
*# primitive root modulo n***def** primroots(n: int) -> ty.Generator[int, **None**, **None**]:  
 *# phi = euler(n)* phi = n - 1 *# n is prime* factors = set(factor(phi)) *# factorize phi* **for** i **in** range(2, phi + 1):  
 **if** all(pow(i, phi // x, n) != 1 **for** x **in** factors):  
 **yield** i  
  
*# diffie-hellman implementation***class** DiffieHellman:  
 **def** \_\_init\_\_(self, n: ty.Optional[int] = **None**, x: ty.Optional[int] = **None**) -> object:  
 *"""* **:rtype***: object  
 """* self.n = n **or** generate(64)[0]  
 self.g = next(primroots(self.n))  
  
 size = self.n.bit\_length()  
 self.\_size = size // 8 + 1  
  
 self.\_s = x **or** generate(size)[0]  
 **while** self.\_s >= self.n:  
 self.\_s, \_ = generate(size)  
  
 self.\_key = pow(self.g, self.\_s, self.n)  
  
 **def** \_\_call\_\_(self, key: int) -> bytes:  
 self.\_key = pow(key, self.\_s, self.n)  
 **return** self.key  
  
 @property  
 **def** key(self) -> bytes:  
 **return** self.\_key.to\_bytes(self.\_size, **"big"**)

Main.py

**from** prime **import** \*  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 t0 = time.time()  
 num, attempts = generate(64)  
 t1 = time.time()  
  
 print(**"Генерация простого числа:\n n = %d\n количество итераций, затраченных на генерацию = %d\n время генерации = %f\n"** % (num, attempts, t1 - t0))  
  
 gen = primroots(num)  
 roots = list(next(gen) **for** \_ **in** range(100))  
 t2 = time.time()  
  
 print(**"Первообразные корни: n = %d\n корни[:100] = %s\n время = %f\n"** % (num, roots, t2 - t1))  
  
 g = roots[0]  
 A = DiffieHellman(num); B = DiffieHellman(num)  
  
 yA = A.\_key; yB = B.\_key  
 k1 = A(yB)  
 k2 = B(yA)  
 t3 = time.time()  
 print(**"Протокол обмена ключами Диффи-Хэллмана:\n n = %d\n g = %d\n время = %f\n"** % (num, g, t3 - t2))  
 print(**"Простое число n = %d\n первообразный корен g = %d\n открытое значение Ya = %d\n открытое значние Yb = %d\n"** % (num, g, yA, yB))  
 print(**"Общий ключ K(абонент A) = %s\nОбщий ключ K(абонент B) = %s"** % (k1, k2))