Vilniaus Universitetas

Matematikos ir Informatikos fakultetas

3 kursas 1 grupė - Informatika

Marius Pozniakovas

Užduotis 23:

Pirminiai skaičiai
(Eratosteno rėčio ir Miller-Rabin algoritmai)

Užduotis: Duotas natūralusis skaičius n. Nustatyti, ar n yra pirminis skaičius.

Natūralusis skaičius – skaičius, kuris **tik** dalinasi *iš savęs* ir *iš vieneto*. Mokyklos matematikos kurse pasakyti ar skaičius yra pirminis atrodo nesunku, nes skaičiai, kurie yra tikrinami dažniausiai būna mažesni už 100. Toliau gilinantis, skaičiai didėja ir natūralu, kad dalinti iš 2,3,5 ir t.t. užtrunka tiesiog per daug laiko ir yra neoptimalu. Todėl matematikai atranda įvairių algoritmų, kurie gali pagelbėti šitai problemai. Su šiuo laboratoriniu darbu bus analizuojami **Eratosteno rėčio** ir **Miller-Rabin** algoritmai.

Algoritmų kintamieji ir apipavidalinimas programoje:

Algoritmo įgyvendinimui naudojamas python3.8.2. Failai ir aplankalai esantys projekto direktorijoje:

- *main.py* (naudojamas pagrindinei programos tėkmei)
- ***py_files*** aplankas:
    - *prime_test.py* (naudojamas saugoti pagrindinius pirminių skaičių testavimo algoritmus)
    - *file_operation.py* (naudojamas saugoti funkcijas skirtas darbui su failais)
    - *generate_graphs.py* (naudojamas generuoti algoritmo sudėtingumo grafikus)
    - *__init__.py* (tuščias failas, skirtas sujungti visus tris viršuje paminėtus failus)
- *task.png* (paveiksliukas su užduotimi)
- ***files*** *aplankas*
    - *numbers.txt* (tekstinis failas skirtas lengvesniam programos testavimui)
    - *eratosthenes.txt* (tekstinis failas sugeneruojamas ***test*** *režimu)*
    - *miller_rabin.txt* (tekstinis failas sugeneruojamas ***test*** *režimu)*

Eratosteno rėčio algoritmas yra faile *prime_test.py*, ***eratosthenes*** funkcijoje.

Miller-Rabin algoritmas yra faile *prime_test.py*, (***miller_rabin, compute, single_miller_test***) funkcijose.

Programos paleidimas: ***python main.py [režimas]***. Programa gali būti vykdoma dvejais būdais – ***test*** ir ***file***. Įrašius neteisingą režimą (sisteminį kintamajį) programa gražina pranešimą: ***usage main.py [test/file]***

Test režimu programa patikrins visus skaičius nuo 1 iki 10000 naudodama abu aukščiau minėtus algoritmus ir automatiškai sugeneruos grafikus. Taip pat, programa sukurs failus *eratosthenes.txt* ir *miller_rabin.txt*, kuriuose išsaugos informaciją apie algoritmo vykdymo laiką ir testų rezultatus.

File režimu programa patikrins skaičius užrašytus *numbers.txt* faile ir sugeneruos grafikus su skaičiais, kurie buvo patikrinti. Taigi atlikti pirminio skaičiaus testą tam tikram skaičiui galima papildžius failą *numbers.txt* (instrukcijos tame pačiame faile).

Eratosteno rėčio algoritmas

***Paaiškinimas*:**

Tarkime turime skaičių n ir norime patikrinti ar n yra pirminis. Atliekame Eratosteno rėčio algoritmą.

1. Susikuriame natūralių skaičių sąrašą [2;N] (žinome, kad jeigu n < 2, tada skaičius nėra pirminis)	1. [2, 3, 4, 5, 6, 7, ..., N]
2. Pradedame nuo 2. Iš skaičiaus sąrašo ištriname skaičius, kurie dalijasi iš 2 (be liekanos). Juos iš sąrašo pašaliname.	2. Daliname iš 2 ir šaliname: [2, 3, 5, 7, ..., N]
3. Taip einame per skaičius, kol pasiekiame skaičių, kuris lygus √(N)	3. Kartojame iki √(N) ir šaliname
4. Jeigu skaičius N dar yra skaičių sąraše, vadinasi jis yra *pirminis*, jeigu skaičiaus nėra – jis *sudėtinis.*	4. [19, 29....],

Pavyzdys:

Tarkime turime skaičių 13 ir norime patikrintis ar 13 yra pirminis skaičius. Atliekame Eratosteno rėčio algoritmą.

Susikuriame natūralių skaičių sąrašą:

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

Pradedame nuo 2, ir šaliname skaičius, kurie dalinasi iš jo be liekanos:

[2, 3, 5, 7, 9, 11, 13]

Ir t.t., iki tol, kol mūsų paimtas skaičius bus didesnis nei √(13) ~ 3.6 = 3:

[2, 3, 5, 7, 11, 13]

Tikriname ar skaičius yra numerių sąraše:

[2, 3, 5, 7, 11, 13]

Kadangi skaičius yra numerių sąraše, jis *pirminis* skaičius.

Kodo funkcija:

```
def eratosthenes(check_if_prime):
    '''function that returns true/false if check_if_prime is a prime number
    uses eratosthenes sieve algorithm'''

    # Algorithm Idea:
    ## 1. Make a list of all numbers from 2 to x
    ## 2. Starting from 2, delete all multiples (except itself)
    ## 3. Repeat the step 2 (until square root of n)
    ## 4. Check if our number is still in list

    #1. get the list ready
    number_list = []
    for i in range(2, check_if_prime + 1, 1):
        number_list.append(i)

    #2 - 3. start from 2
    cycling_number = 2
        #until square root of n
    while cycling_number <= int(sqrt(check_if_prime)):
        #if we find a the number we are on right now in list
        if cycling_number in number_list:
            #cycle through the list and find multiples
            for divising_number in number_list:
                if int(divising_number) % int(cycling_number) == 0 \
                and divising_number > cycling_number:
                    number_list.remove(divising_number)

        cycling_number = cycling_number + 1

    #after we finished
    if check_if_prime in number_list:
        return True
    else:
        return False
```

Kodo funkcionalumas:

Faile *numbers.txt* paliekame skaičius, kuriuos norime patikrinti. Taip pat užrašome 1 atskirtą kabliataškiu, norėdami panaudoti Eratosteno rėčio algoritmą:

```
18977;1
777;1
10007;1
64577;1
44444;1
```

Gautas atsakymas:

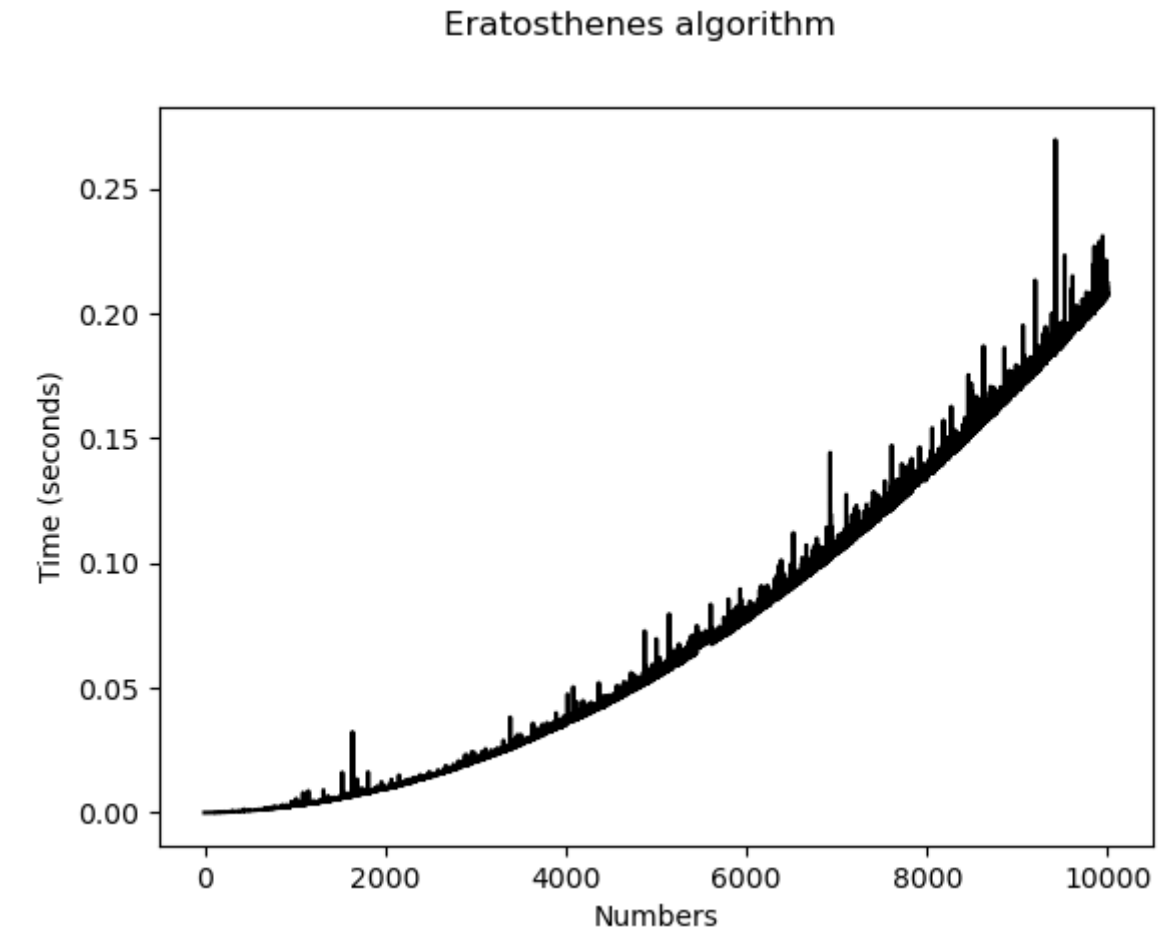
```
Erasosthenes sieve algorithm was working for 0.73 seconds
--- 18977 is NOT a prime number ---

Erasosthenes sieve algorithm was working for 0.0 seconds
--- 777 is NOT a prime number ---

Erasosthenes sieve algorithm was working for 0.21 seconds
--- 10007 is a PRIME number ---

Erasosthenes sieve algorithm was working for 8.14 seconds
--- 64577 is a PRIME number ---

Erasosthenes sieve algorithm was working for 3.84 seconds
--- 44444 is NOT a prime number ---

File 2020-05-14_01.01.17.txt saved information aswell.
```

Failas *2020-05-14_01.01.17.txt*:

```
18977 is COMPOSITE number. Algorithm used: Eratosthenes sieve. Time passed: 0.7331890999999999 secs
777 is COMPOSITE number. Algorithm used: Eratosthenes sieve. Time passed: 0.0017559000000000324 secs
10007 is PRIME number. Algorithm used: Eratosthenes sieve. Time passed: 0.20843299999999998 secs
64577 is PRIME number. Algorithm used: Eratosthenes sieve. Time passed: 8.1372707 secs
44444 is COMPOSITE number. Algorithm used: Eratosthenes sieve. Time passed: 3.844400799999999 secs
```

Algoritmo sudėtingumas:

Pabandžius algoritmą ***test*** režimu gautas grafikas:



Miller - Rabin algoritmas

***Paaiškinimas*:**

Tarkime turime skaičių n ir norime patikrinti ar n yra pirminis. Atliekame Miller-Rabin algoritmą. Šis algoritmas nėra tikslus, nes šis testas įrodo, kad skaičius nėra pirminis, vadinasi, jeigu per mažai kartų yra atliktas testas, jis nebegali įrodyti, kad skaičius nėra pirminis.

1. Jeigu skaičius yra lyginis (t.y. dalinasi iš 2 be liekanos) – skaičius nėra pirminis

2. (n – 1) dalindami iš dviejų iki tol, kol gausime liekaną suskaičiuojame $n - 1 = 2^k * m$ (k, m – sveikieji skaičiai)

3. Sugeneruojame *atsitiktinį* skaičių a, kuris $1 < a < n - 1$

4. Skaičiuojame $b_0 = a^m \bmod(n)$. Jeigu suskaičiuotas b_0 lygus ± 1, darome prielaidą, kad skaičius n yra *turbūt* pirminis

5. Jeigu suskaičiuotas b_0 nelygus ± 1, tada skaičiuojame b_1, o $b_1 = b_0^2 \bmod(n)$.
    a. Jeigu b_1 ir t.t. lygus +1, vadinasi n nėra pirminis
    b. Jei b_1 ir t.t. lygus -1, n yra *turbūt* pirminis
    c. Jeigu b_1 ir t.t. nelygus +1 ar -1, skaičiuojame b_2

Pavyzdys:

Tarkime turime skaičių 53 ir norime patikrintis ar 13 yra pirminis skaičius. Atliekame Miller-Rabin algoritmą.

1. Skaičius nėra lyginis. Tęsiame toliau.
2. Skaičiuojame $(n - 1) = 2^k * m$.
    a. $m = 13$
    b. $k = 2$
3. Pasirenkame *atsitiktinį* skaičių, kuris didesnis nei 1, bet mažesnis nei 52.
    a. Pasirenkame skaičių 2, $a = 2$
4. Skaičiuojame b_0. $b_0 = 2^{13} \bmod 53 = 30 \bmod 53$
5. Kadangi $30 != \pm 1$, skaičiuojame b_1.
6. $b_1 = 30^2 \bmod 53 = -1 \bmod 53$.
7. Kadangi gavome -1, skaičius yra *turbūt* pirminis.

***Kodo funkcijos(1) (miller_rabin)*:**

```
def miller_rabin(check_if_prime, iterations):
    '''function that returns true/false if check_if_prime is a prime number,
    uses miller_rabin algorithm'''

    #Algorithm Idea:
    ## 0. If the number is even we can return true instantly :), else:
    ## 1. Find n-1 = 2^k*m
    #### n = check_if_prime, n-1 = testing_number (before substraction),
    #### m = testing_number
    ## 2. Choose random number: 1 < a < n-2
    ## 3. Compute b(0) = a^m(modn) and so on

    ### Example with 53 below.

    #0 (53 isnt even, move on)
    if check_if_prime % 2 == 0:
        return False
    #1
    else:

        #get n-1 (53 - 1 = 52)
        testing_number = check_if_prime - 1

        #get lowest possible number that doesnt divide from 2 without reminder
        while (int(testing_number) % 2 == 0):
            testing_number = testing_number / 2

        #iterate and try to find
        while(iterations > 0):
            iterations = iterations - 1
            #if this test fails, the number is certainly not a prime number
            if single_miller_test(testing_number, check_if_prime) == False:
                return False

        #this number is possibly a prime number :)
        return True
```

***Kodo funkcijos(2) (single_miller_test)*:**

```
def single_miller_test(small_number, check_if_prime):
    '''miller test function
       if it returns true, that means check_if_prime is PROBABLY prime
       if it returns false, that means check_if_prime is NEVER prime'''

    #2, get random element (1 - 51, suppose we pick 2)
    rand = randint(1, check_if_prime - 2)

    ans = compute(rand, small_number, check_if_prime)

    if (ans == check_if_prime - 1 or ans == 1):
        return True

    while(small_number != check_if_prime - 1):
        ans = pow(ans, 2) % check_if_prime
        small_number = small_number * 2

        if ans == 1:
            return False

        elif ans == check_if_prime - 1:
            return True

    return False
```

***Kodo funkcijos(3) (compute)*:**

```
def compute(random_num, divided_number, check_if_prime):

    #suppose the answer is 1
    ans = 1

    #2 % 53 = 2
    random_num = random_num % check_if_prime

    #make sure its an int
    divided_number = int(divided_number)

    while (divided_number > 0):

        #if m is not even
        if (divided_number % 2 == 1):
            # 1) 1 * 2 % 53 = 2;
            # 3) 2 * 16 % 53 = 16;
            ans = (ans * random_num) % check_if_prime

        # 1) 13 / 2 = 6;
        # 2) 6 / 2 = 3;
        # 3) 3 / 2 = 1;
        divided_number = divided_number // 2

        # 1) 2 ^ 2 % 53 = 4 % 53 = 4;
        # 2) 4 ^ 2 % 53 = 16
        # 3) 16 ^ 16 % 53 = 256 % 53 = 44
        random_num = pow(random_num, 2) % check_if_prime
    return ans
```

Kodo funkcionalumas:

Faile *numbers.txt* paliekame tuos pačius skaičius, kuriuos tikrinome Eratosteno rėčio algoritmu. Taip pat užrašome 2 atskirtą kabliataškiu, nustatydami Miller – Rabin algoritmą:

```
18977;2
777;2
10007;2
64577;2
44444;2
```

Gautas atsakymas:

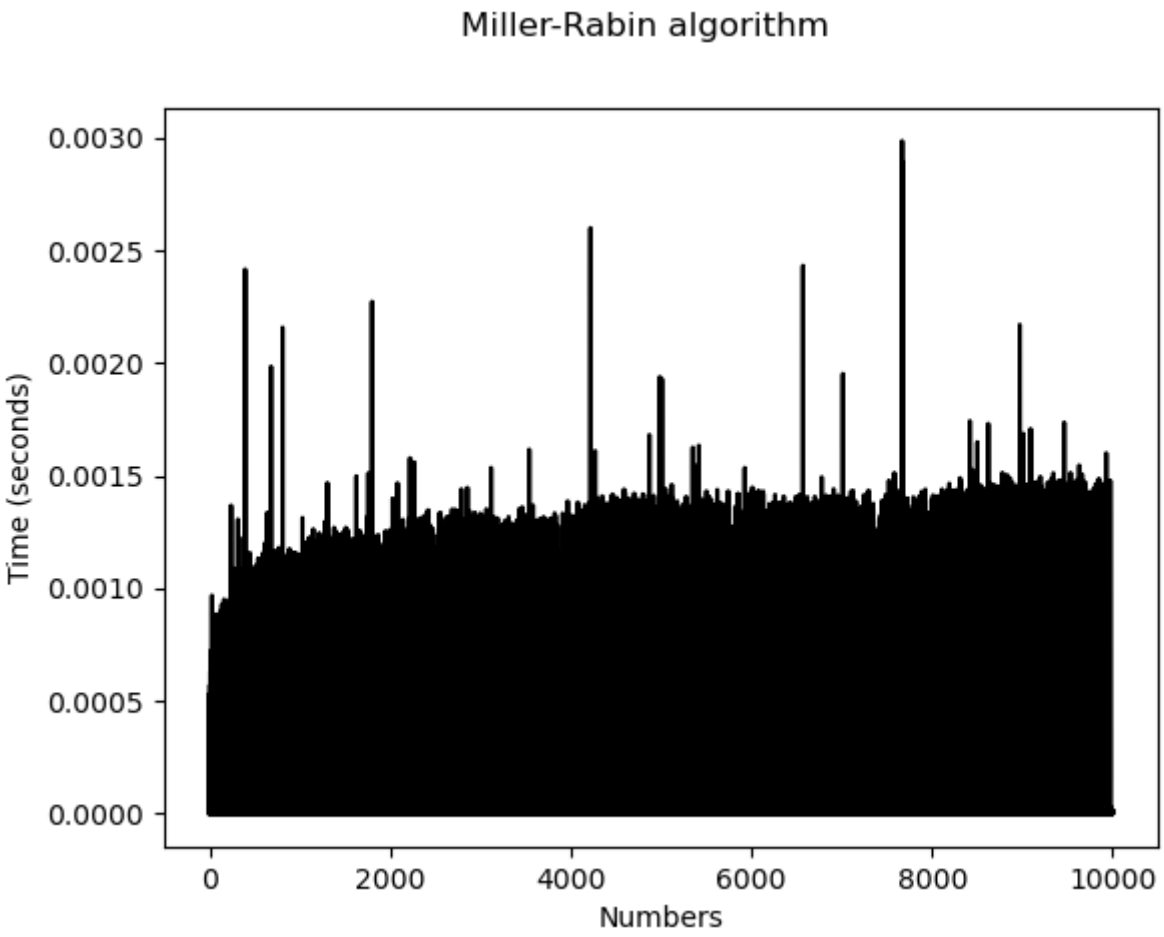
```
Miller-Rabin algorithm was working for 0.0 seconds
--- 18977 is NOT a prime number ---

Miller-Rabin algorithm was working for 0.0 seconds
--- 777 is NOT a prime number ---

Miller-Rabin algorithm was working for 0.0 seconds
--- 10007 is a PRIME number ---

Miller-Rabin algorithm was working for 0.0 seconds
--- 64577 is a PRIME number ---

Miller-Rabin algorithm was working for 0.0 seconds
--- 44444 is NOT a prime number ---

File 2020-05-14_05.17.50.txt saved information aswell.
```

Failas *2020-05-14_05.17.50.txt*:

```
18977 is COMPOSITE number. Algorithm used: Miller-Rabin. Iterations used None.  Time passed: 3.4000000000089514e-05 secs
777 is COMPOSITE number. Algorithm used: Miller-Rabin. Iterations used None.  Time passed: 1.7500000000003624e-05 secs
10007 is PRIME number. Algorithm used: Miller-Rabin. Iterations used None.  Time passed: 6.740000000005075e-05 secs
64577 is PRIME number. Algorithm used: Miller-Rabin. Iterations used None.  Time passed: 6.530000000004588e-05 secs
44444 is COMPOSITE number. Algorithm used: Miller-Rabin. Iterations used None.  Time passed: 1.6999999999933735e-06 secs
```

Algoritmo sudėtingumas:

Pabandžius algoritmą ***test*** režimu gautas grafikas:



Išvados

- Eratosteno rėčio algoritmas įrodo, kad skaičius yra pirminis visais atvejais, o Miller – Rabin algoritmas įrodo, kad skaičius nėra pirminis.
- Abiejų algoritmų skaičiaus testavimo ilgis yra tiesiogiai proporcingas skaičiaus ilgiui
- Eratosteno rėčio algoritmas veiks daug ilgiau su ilgesniais skaičiais.
- Didėjant skaičiams Miller – Rabin algoritmas labai minimaliai ilgina algoritmo veikimo ilgį.
- Palyginus to paties skaičiaus testo ilgį abiejuose algoritmuose Miller – Rabin algoritmas yra daug efektyvesnis.
- Ilgiausias trukęs skaičiavimas Eratosteno rėčio algoritme truko virš 0.25 sekundės, o Miller – Rabin – apie 0.0030 sekundės.

Literatūra:

https://research.cs.wisc.edu/techreports/1990/TR909.pdf

http://www.mat.uniroma2.it/~schoof/millerrabinpom.pdf

https://primes.utm.edu/prove/prove2_1.html

https://primes.utm.edu/prove/prove2_3.html#MillersERHTest