KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

ALGORITMŲ SUDARYMAS IR ANALIZĖ

PIRMAS NAMŲ DARBAS

Karolis Ryselis, IFF-1

1. Užduotis

Rikiavimo uždavinys: realizuoti ir palyginti tris rūšiavimo algoritmus, kai rūšiuojami duomenys saugomi kietajame diske masyvo ir sąrašo būdais. Lyginami algoritmai

- Rūšiavimas „suliejimu“
- Greitasis rūšiavimas „Quick sort“
- Rūšiavimo algoritmas „Bucket sort“

2. Realizacija

2.1. Realizacijos aplinka

Laboratorinio darbo užduotys atliekamos python programavimo kalba Aptana Studio 3 aplinkoje.

2.2. Rūšiuojami elementai

Rūšiuojami elementai- sveikieji neneigiami skaičiai, kurių reikšmė yra iki 4 baitų sveikojo skaičiaus be ženklo dydžio (2^{32}-1). Rikiuosime kiekvienu būdu bent po 60 minučių. Rikiavimo pradinė duomenų imtis- 10000 skaičių, kiekviena sekanti imtis didesnė už prieš tai buvusią 10000:
$elementų\ skaičius = 10000 \times eilės\ numeris,\ eilės\ numeris = 1,2,3, \ldots$

2.3. Elementų rūšiavimas kietajame diske

Rūšiuojami elementai saugomi kietajame diske. Skaitant elementą bus nueita iki jo fizinės vietos diske ir perskaitomas atitinkamas kiekis baitų (4 baitai). Masyvo atveju elementai pasiekiami žinant jų adresą faile (4*elemento indeksas), sąrašo atveju iteruojama per elementus, kol pasiekiamas norimas elementas. Duomenų failas atrodo taip (realiai dvejetainis):

146410 82154523 424101 724142100 644210411 82241201 421412310 2246349 66564554 14635467

Darbui su failais realizuotos dvi klasės: BinaryFile (darbui su masyvais) ir BinaryFileList (darbui su sąrašais).

Klasė BinaryFile:

```
class BinaryFile:
    file_name = FILENAME
    ops = 0

    def __init__(self):
        self.open_file()
        self.file_size = os.path.getsize(self.file_name) / 4

    def open_file(self):
        self.handle = open(self.file_name, 'r+b')

    def get_byte_at_index(self, index):
        self.handle.seek(4 * index)
        bytes = self.handle.read(4)
        self.ops += 3
        return bytes_to_integer(bytes)

    def swap_bytes(self, index1, index2):
        self.handle.seek(4 * index1)
        bytes1 = self.handle.read(4)
        self.handle.seek(4 * index2)
        bytes2 = self.handle.read(4)
        self.handle.seek(4 * index2)
        self.handle.write(bytes1)
        self.handle.seek(4 * index1)
        self.handle.write(bytes2)
        self.ops += 8
```

```
    def __str__(self):
        res = ''
        for i in range(10):
            res += '%10s\n' % self.get_byte_at_index(i)
        return res
```

Klasė BinaryFileList:

```
class BinaryFileList(BinaryFile):
    current_position = 0
    ops = 0

    def seek(self, index):
        diff = index - self.current_position
        seek_dir = 4 if diff >= 0 else -4
        self.handle.seek(4 * diff, os.SEEK_CUR)
        self.current_position = index
        self.ops += 4

    def get_byte_at_index(self, index):
        self.seek(index)
        bytes = self.read()
        self.ops += 3
        return bytes_to_integer(bytes)

    def read(self):
        self.current_position += 1
        self.ops += 2
        return self.handle.read(4)

    def write(self, val):
        self.current_position += 1
        self.handle.write(val)
        self.ops += 2

    def swap_bytes(self, index1, index2):
        self.seek(index1)
        bytes1 = self.read()
        self.seek(index2)
        bytes2 = self.read()
        self.seek(index2)
        self.write(bytes1)
        self.seek(index1)
        self.write(bytes2)
        self.ops += 8
```

3. Rūšiavimo algoritmų tyrimas

3.1. Rūšiavimas “suliejimu”

3.1.1. Algoritmo teorinis įvertinimas

Rūšiavimas suliejimu remiasi principu “skaldyk ir valdyk”. Duomenų imtis dalijama į dvi lygias dalis, kiekviena dalis vėl dalijama, kol lieka tik vienas elementas. Tuomet jis jau yra surikiuotas. Duomenų imtis, kuri buvo padalinta, suliejama taip, kad būtų jau surikiuota: skaitomi elementai iš abiejų dalių ir rašomi į bendrą masyvą tokia tvarka, kad būtų surikiuoti. Kadangi masyvai viduje jau yra surikiuoti, reikia tik iteruoti per juos abu ir lyginti mažiausius neperskaitytus elementus. Mažesnis iš jų dedamas į bendrą masyvą. Kai vienas masyvas pasibaigia, kito masyvo elementai surašomi iš eilės nelyginant jų su niekuo. Teorinis algoritmo įvertis yra $O(n \log_2 n)$, n- rikiuojamų elementų skaičius.

3.1.2. Algoritmo kodas ir sudėtingumo skaičiavimas

Algoritmui realizuoti specialiai papildomais metodais papildytos duomenų valdymo klasės, kadangi reikia kurti naujus failus (algoritmą, dirbantį masyvo viduje, labai sunku realizuoti), failų papildymo metodas.

Klasė BinaryFile:

```
class BinaryFile(quick_sort.BinaryFile):
    def __init__(self, filename):
        self.file_name = filename
        self.open_file()
        self.file_size = os.path.getsize(self.file_name) / 4

    def make_new_file(self, type, pos1, pos2):
        if type == 0:
            name = '../FILE_LEFT%s' % self.file_count
        else:
            name = '../FILE_RIGHT%s' % self.file_count
        self.file_count += 1
        self.handle.seek(0)
        content = self.handle.read()
        handle = open(name, 'w')
        handle.write(content[pos1 * 4:pos2 * 4])
        handle.close()
        new_left = BinaryFile(name)
        self.ops += 9
        return new_left

    def set_byte(self, index, byte):
        self.handle.seek(4 * index)
        self.handle.write(byte)
        self.ops += 2

    def del_file(self):
        self.handle.close()
        os.remove(self.file_name)
        self.ops += 2

    def get_raw_byte_at_index(self, index):
        self.handle.seek(4 * index)
        bytes = self.handle.read(4)
        self.ops += 3
        return bytes

    def append(self, val):
        self.handle.seek(0, os.SEEK_END)
        self.handle.write(val)
        self.file_size = os.path.getsize(self.file_name) / 4
        self.ops += 3
```

Klasė BinaryFileList:

```
class BinaryFileList(quick_sort.BinaryFileList):
    def __init__(self, filename):
        self.file_name = filename
        self.open_file()
        self.file_size = os.path.getsize(self.file_name) / 4

    def make_new_file(self, type, pos1, pos2):
```

```
        if type == 0:
            name = '../FILE_LEFT%s' % self.file_count
        else:
            name = '../FILE_RIGHT%s' % self.file_count
        self.file_count += 1
        self.handle.seek(0)
        content = self.handle.read()
        self.handle.seek(4 * self.current_position)
        handle = open(name, 'w')
        handle.write(content[pos1 * 4:pos2 * 4])
        handle.close()
        new_left = BinaryFile(name)
        self.ops + 11
        return new_left

    def set_byte(self, index, byte):
        self.seek(index)
        self.write(byte)
        self.ops += 2

    def del_file(self):
        self.handle.close()
        os.remove(self.file_name)
        self.ops += 2

    def get_raw_byte_at_index(self, index):
        self.seek(index)
        bytes = self.read()
        self.ops += 3
        return bytes

    def append(self, val):
        self.seek(self.file_size)
        self.write(val)
        self.file_size = os.path.getsize(self.file_name) / 4
        self.ops += 3
```

Algoritmas realizuotas funkcijomis merge (atlieka masyvų suliejimą) ir mergeSort (atlieka masyvų dalijimą ir suliejimo iškvietimą).

```
def merge(A, start, mid, end):
    L = A.make_new_file(0, start, mid)                  #c1
    R = A.make_new_file(1, mid, end)                    #c1
    i = 0                                               #c2
    j = 0                                               #c2
    k = start                                           #c2

    for l in range(k, end):                             #(end-k+1)*c3
        if j >= R.file_size or (i < L.file_size and L.get_byte_at_index(i) <
R.get_byte_at_index(j)):#c7
            A.set_byte(l, L.get_raw_byte_at_index(i))   #c4
            i = i + 1                                   #c5
        else:
            A.set_byte(l, R.get_raw_byte_at_index(j))   #c4
            j = j + 1                                   #c5
    L.del_file()                                        #c6
    R.del_file()                                        #c6
```

```
    A.ops += 8 + (end - k) * 3 + 1 + L.ops + R.ops

def merge_sort(A, p, r):
    if r - p > 1:                                   #c8
        mid = int((p + r) / 2)                      #c9
        merge_sort(A, p, mid)                       #T(mid-p)
        merge_sort(A, mid, r)                       #T(r-mid)
        merge(A, p, mid, r)                         #T1(p,mid,r)
        A.ops += 5
    else:
        A.ops += 1
```

Skaičiuojame algoritmo įvertį pagal užrašytą kodą.

$T_{merge}(start, mid, end) = 2c_1 + 3c_2 + (end - k + 1)c_3 + c_7 + c_4 + c_5 + c_6.$

Kadangi $k = start$ ir konstantas galime apjungti į vieną,

$T_{merge}(start, mid, end) = (end - start + 1)c_3 + C.$

$T_{mergeSort}(p, r) = c_8 + c_9 + T_{mergeSort}(mid - p) + T_{mergeSort}(r - mid) + T_{merge}(p.mid, r).$

Kadangi $mid = \frac{p+r}{2}$, o merge funkcijos vykdymas nepriklauso nuo parametro mid, tai

$T_{mergeSort}(p, r) = c_8 + c_9 + T_{mergeSort}\left(\frac{r-p}{2}\right) + T_{mergeSort}\left(\frac{r-p}{2}\right) + T_{merge}(p, r).$

Kadangi funkcijos merge_sort veikimo greitis priklauso nuo $r - p$, tai pažymėję $n = r - p$ gauname

$T_{mergeSort}(n) = 2T_{mergeSort}\left(\frac{n}{2}\right) + T_{merge}(p, r) + C.$

$T_{merge}(n) = (n + 1)c_3 + C.$

$T_{mergeSort}(n) = 2T_{mergeSort}\left(\frac{n}{2}\right) + T_{merge}(p, r) + C.$

$T_{mergeSort}(n) = 2T_{mergeSort}\left(\frac{n}{2}\right) + (n + 1)c_3 + C.$

Gavome rekursinę lygtį, kuriai galima taikyti pagrindinę teoremą.

$f(n) = (n + 1)c_3 + C, g(n) = n^{\log_2 2} = n$

Funkcijų laipsniai vienodi, todėl galioja antrasis teoremos atvejis ir gauname

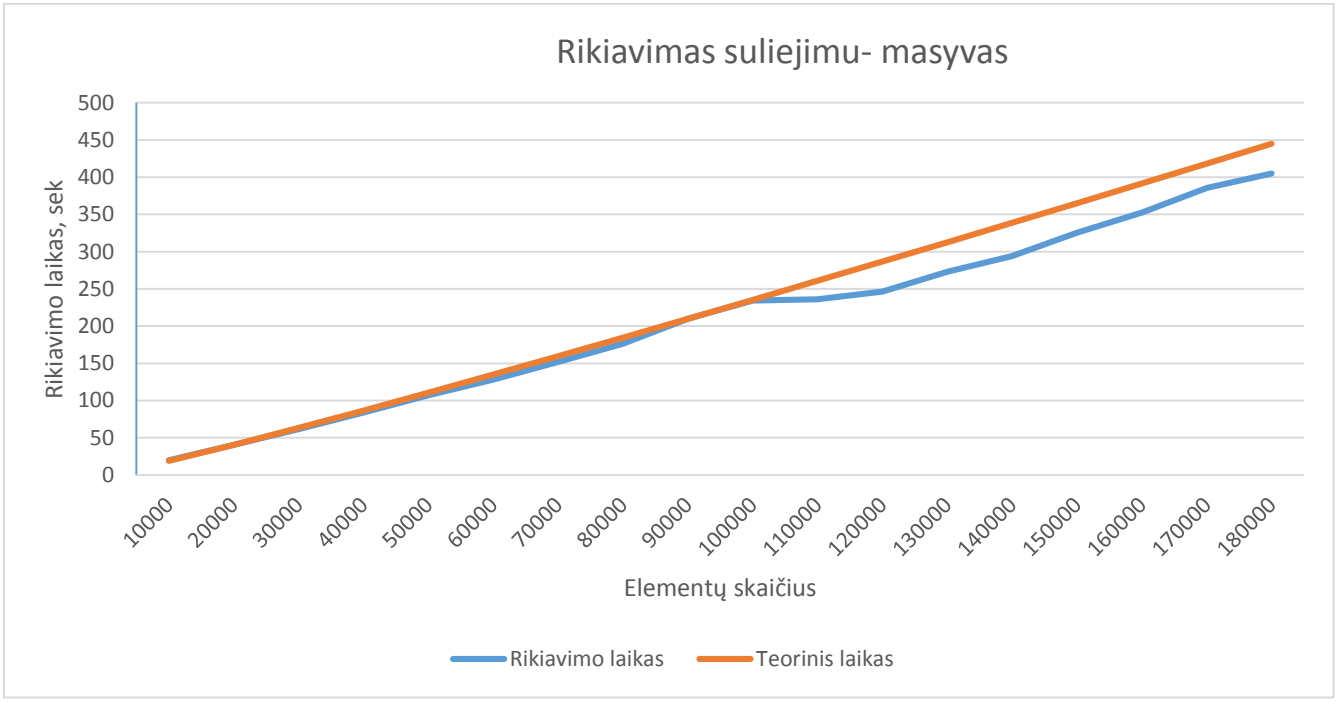
$T_{mergeSort}(n) = O(n \log_2 n)$

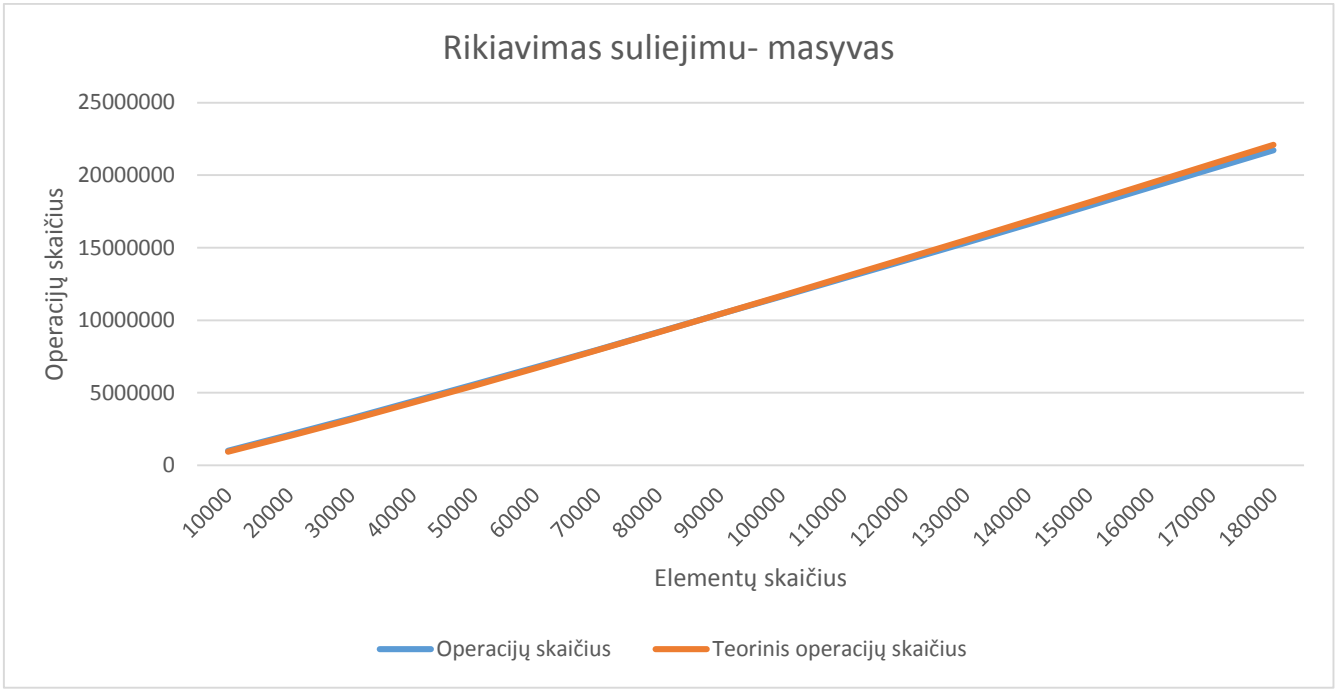
3.1.3. Algoritmo eksperimentinis tyrimas

Masyvo rikiavimas

Elementų skaičius	Laikas, s	Operacijų skaičius
10000	19,69	998.048
20000	40,382	2.096.128
30000	61,431	3.226.128
40000	83,951	4.392.288

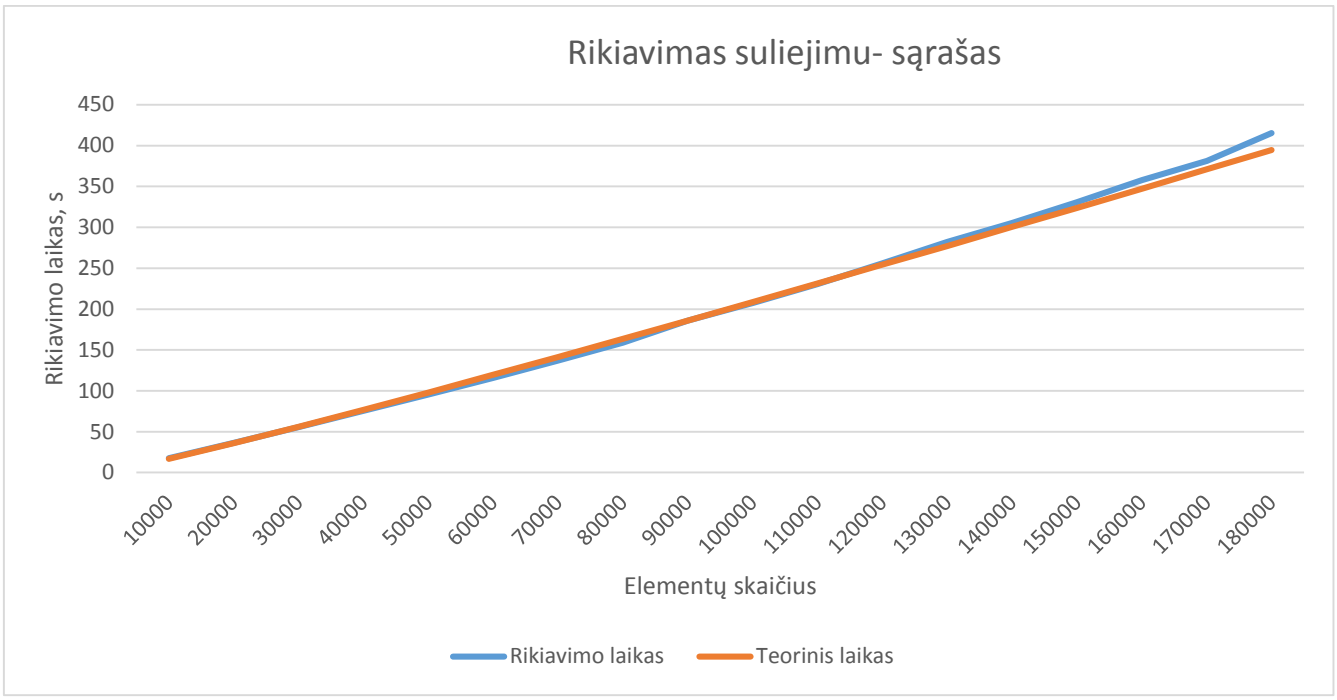
50000	107,054	5.572.288
60000	128,209	6.752.288
70000	151,707	7.954.608
80000	176,085	9.184.608
90000	209,691	10.414.608
100000	234,479	11.644.608
110000	236,002	12.874.608
120000	246,253	14.104.608
130000	273,051	15.334.608
140000	293,937	16.609.248
150000	325,292	17.889.248
160000	352,665	19.169.248
170000	385,43	20.449.248
180000	404,846	21.729.248

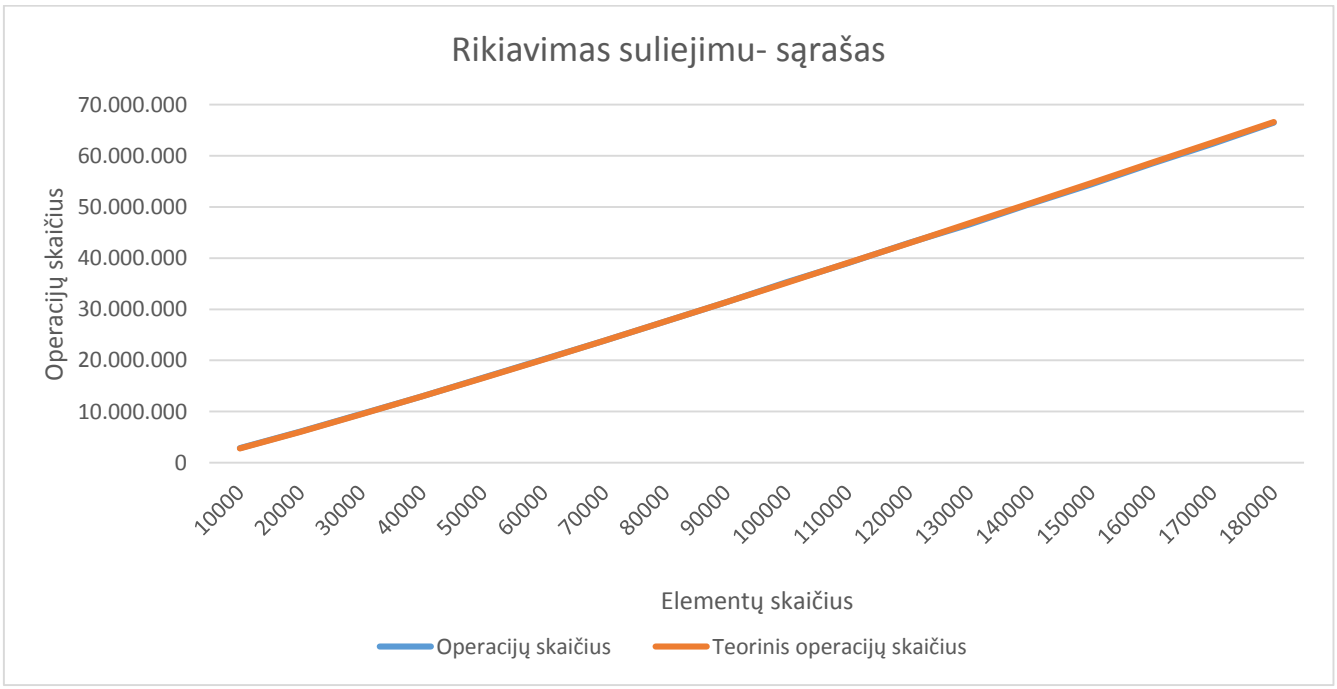




Sąrašo rikiavimas

Elementų skaičius	Laikas, s	Operacijų skaičius
10000	17,509	2.844.970
20000	36,261	6.109.052
30000	55,554	9.545.860
40000	75,66	13.006.378
50000	95,671	16.622.926
60000	115,939	20.244.812
70000	137,09	23.870.484
80000	158,938	27.652.706
90000	185,97	31.382.380
100000	207,485	35.302.298
110000	230,478	39.035.454
120000	256,155	42.935.380
130000	282,038	46.593.968
140000	305,136	50.593.668
150000	330,683	54.385.340
160000	357,937	58.457.788
170000	381,355	62.340.708
180000	415,134	66.474.932





Matome, jog teoriniai paskaičiavimai ir gauti rezultatai praktiškai identiški.

3.2. Greitas rūšiavimas „Quick sort“

3.2.1. Algoritmo teorinis įvertinimas

Greito rūšiavimo algoritmas yra rekursinis algoritmas, paremtas principu „skaldyk ir valdyk“. Pasirenkamas indeksas, kuris tampa atraminiu elementu. Jis yra sukeičiamas su paskutiniu elementu. Tuomet visi elementai, mažesni už atraminį, sukeliami į masyvo pradžią, o didesni lieka gale. Tuomet atraminis elementas, kuris yra paskutinėje pozicijoje, perkeliamas į tą vietą, kurioje baigiasi už jį mažesni skaičiai. Tokiu būdu kairėje pusėje

lieka visi mažesni elementai, dešinėje- visi didesni. Abi pusės rikiuojamos analogiškai, kol lieka vienas elementas, kuris jau yra surikiuotas. Atliekant pakeitimus masyvo viduje, nereikalingas masyvų sujungimas, masyvas jau surikiuotas. Teorinis algoritmo įvertis $O(n \log_2 n)$ vidutiniu atveju, $O(n^2)$ blogiausiu atveju.

3.2.2. Algoritmo kodas ir sudėtingumo skaičiavimas

Algoritmui realizuoti naudojamos funkcijos sort (rekursinė funkcija) ir partition (padalija masyvus).

```
def sort(array_to_sort, start_index, end_index):
    if start_index < end_index:                                    #c1
        pivot_index = random.randint(start_index, end_index - 1)#c2
        new_pivot_index = partition(array_to_sort, start_index, end_index,
pivot_index)#T1(start_index,end_index,pivot_index)
        sort(array_to_sort, start_index, new_pivot_index - 1)
#T(start_index,new_pivot_index-1)
        sort(array_to_sort, new_pivot_index + 1, end_index)
#T(new_pivot_index+1,end_index)
        array_to_sort.ops += 5
    else:
        array_to_sort.ops += 1

def partition(array_to_sort, start_index, end_index, pivot_index):
    pivot = array_to_sort.get_byte_at_index(pivot_index)           #c3
    array_to_sort.swap_bytes(pivot_index, end_index)               #c4
    current_index = start_index                                    #c5
    for i in range(start_index, end_index):                        #(end_index-
start_index+1)*c6
        if array_to_sort.get_byte_at_index(i) < pivot:             #c7
            array_to_sort.swap_bytes(i, current_index)             #c8
            current_index += 1                                     #c9
            array_to_sort.ops += 2
        array_to_sort.ops += 1
    array_to_sort.swap_bytes(end_index, current_index)             #c4
    array_to_sort.ops += 6 + end_index - start_index
    return current_index                                           #c10
```

Skaičiuojame algoritmo įvertį pagal užrašytą kodą.

$$T_{sort}(start, end) = c_1 + c_2 + T_{partition}(start, end, pivot) + T_{sort}(start, newPivot - 1) + T_{sort}(newPivot + 1, end)$$

$$T_{partition}(start, end, pivot) = c_3 + 2c_4 + c_5 + (end - start + 1)c_6 + (end - start)c_7 + k(end - start)(c_8 + c_9) + c_{10}$$

k- tikimybė, kad bus išpildyta sąlyga, kad i-tasis elementas mažesnis už atraminį. Kadangi skaičiai yra pasiskirstę tolygiai ir atraminis skaičius pasirenkamas atsitiktinai, tai tikimybė lygi 0,5, todėl $k = 0.5$. Padarius tokią prielaidą gauname, kad funkcijos darbo laikas nepriklauso nuo $pivot$ reikšmės.

$$T_{partition}(start, end) = c_3 + 2c_4 + c_5 + (end - start + 1)c_6 + (end - start)c_7 + 0.5(end - start)(c_8 + c_9) + c_{10}$$

Išplaukia, kad atraminio indekso reikšmė sort funkcijoje vidutiniškai bus lygi ribinių reikšmių vidurkiui, nes dydis yra atsiktinis, o tokiu atveju ir atraminio indekso reikšmė po perdalijimo bus lygi ribinių reikšmių

vidurkiui, nes atraminės reikšmės skaitinė vertė bus tolygiai pasiskirsčiusi su visomis interval reikšmėmis, todėl galime užrašyti:

$$T_{sort}(start, end) = c_1 + c_2 + T_{partition}(start, end) + T_{sort}\left(start, \frac{end - start}{2} - 1\right) + T_{sort}(\frac{end - start}{2} + 1, end)$$

Analogiškai, kaip ir suliejimo algoritmui, gauname

$$T_{sort}(n) = T_{partition}(n) + T_{sort}\left(\frac{n}{2} - 1\right) + T_{sort}\left(\frac{n}{2} - 1\right) + C$$

$$T_{sort}(n) = T_{partition}(n) + 2T_{sort}\left(\frac{n}{2} - 1\right) + C$$

$$T_{partition}(n) = (n+1)c_6 + (n)c_7 + 0{,}5(n)(c_8 + c_9) + C$$

$$T_{partition}(n) = n(c_6 + c_7 + 0{,}5(c_8 + c_9)) + C$$

$$T_{partition}(n) = nD + C$$

Įstatę į T_{sort} gauname

$$T_{sort}(n) = nD + 2T_{sort}\left(\frac{n}{2} - 1\right) + C$$

Kai n dideli, galime aproksimuoti

$$T_{sort}(n) = nD + 2T_{sort}\left(\frac{n}{2}\right) + C$$

Gavome lygtį, pagrindinės teoremos atžvilgiu analogišką suliejimo algoritmo lygčiai. Ją išsprendę gauname

$$T_{sort}(n) = O(n \log_2 n)$$

Tačiau, kai realizuojame sąrašu, turime turėti omeny, kad faile operacijos atliekamos ne su greta esančiais elementais, o su toliau vienas nuo kito esančiais, todėl priėjimas prie element tampa nebe $O(1)$, bet $O(n)$ ir algoritmo veikimas blogėja iki $O(n^2 \log_2 n)$, nes partition funkcijoje visos sukeitimo operacijos tampa $O(n)$ ir jos sudėtingumas didėja n kartų.

3.2.3. Algoritmo eksperimentinis tyrimas

Masyvo rikiavimas

Elementų skaičius	Rikiavimo laikas, s	Operacijų skaičius
10000	2,926	1.799.128
20000	6,241	3.859.816
30000	9,266	5.745.682
40000	13,319	8.285.446
50000	16,99	10.436.170
60000	21,39	13.160.900
70000	25,004	14.628.369
80000	28,895	17.836.704

90000	30,464	19.008.005
100000	37,074	22.976.894
110000	40,578	25.173.305
120000	43,328	26.906.395
130000	45,97	28.521.136
140000	51,377	32.113.356
150000	52,866	32.876.898
160000	63,726	39.227.072
170000	61,615	38.574.064
180000	64,842	40.502.371
190000	71,601	44.797.602
200000	75,08	46.869.256
210000	77,736	48.326.085
220000	82,28	51.230.699
230000	86,052	53.202.767
240000	96,045	59.882.721
250000	96,371	59.819.291
260000	102,25	63.544.809
270000	106,613	66.227.042
280000	108,999	68.024.663
290000	109,614	68.155.081
300000	115,658	71.746.207
310000	116,759	72.865.399
320000	123,243	76.872.229
330000	135,47	83.207.675
340000	132,862	82.525.574
350000	150,626	93.380.226
360000	150,97	93.942.811
370000	148,442	91.929.473
380000	153,554	94.696.249
390000	161,688	99.586.318
400000	161,375	99.346.156
410000	177,003	109.614.751
420000	164,776	102.036.261
430000	176,883	110.125.886

Sąrašo rikiavimas

Elementų skaičius	Rikiavimo laikas	Operacijų skaičius
10000	6,282	54.816.021
20000	17,282	212.214.427
30000	32,62	466.901.017
40000	52,58	829.806.955
50000	77,404	1.286.534.176

60000	116,947	1.842.992.987
70000	138,379	2.494.589.037
80000	171,907	3.251.201.779
90000	212,065	4.100.786.709
100000	258,662	5.065.290.069
110000	307,932	6.105.825.525
120000	363	7.280.574.535
130000	426,052	8.549.200.557
140000	465,698	9.877.812.081
150000	537,282	11.333.359.053
160000	594,7	12.897.585.695

Greitasis rikiavimas su masyvu elgiasi kaip tikėtasi, bet su sąrašu veikia lėčiau, nes dalijimo sudėtingumas dar padidėja n kartų.

3.3. Rikiavimo algoritmas „Bucket sort“

3.3.1. Algoritmo teorinis įvertinimas

Kibirinis rikiavimas veikia tokiu principu: sukuriama tiek „kibirų“, kiek masyve yra elementų, kiekvienam „kibirui“ priskiriamas intervalas, ir į „kibirus“ sudedami elementai. „Kibirai“ surikiuojami įterpimo rikiavimo algoritmu, iš eilės perskaitomi ir sudedami į bendrą masyvą, kuris yra surikiuotas. Kadangi elementų yra tiek, kiek kibirų, kiekvienam kibirui tenka vidutiniškai po vieną elementą. Kai kibirų intervalai vienodi, o elementai tolygiai pasiskirstę intervale, nukrypimai nuo vidurkio labai nedideli. Tokiu atveju algoritmo veikimo greitis yra tiesinis. Blogiausiu atveju, kai viskas sukrenta į vieną kibirą, turime rikiavimą įterpimu, kurio greitis kvadratinis.

3.3.2. Algoritmo kodas ir sudėtingumo skaičiavimas

```
def bucket_sort(lst):
    buckets = []                                                    #c1
    for i in range(lst.file_size):                                  #(n+1)*c2
        open('../bucket%s' % i, 'w').close()                        #n*c3
        buckets.append(merge_sort.BinaryFile('../bucket%s' % i))#n*c4
    max_val = 4294967296                                            #c5
    for i in range(lst.file_size):                                  #(n+1)*c2
        val = lst.get_byte_at_index(i)                              #n*c6
        buckets[int((val*lst.file_size)/max_val)-1].append(integer_to_bytes(val))#n*c7
    for i in range(lst.file_size):                                  #(n+1)*c2
        insertion_sort(buckets[i])                                  #n*c3
    tot = 0                                                         #c5
    for bucket in buckets:                                          #n*c8
        for i in range(bucket.file_size):                           #n*c9
            lst.set_byte(tot, bucket.get_raw_byte_at_index(i))      #n*c10
            tot += 1                                                #n*c11
    for bucket in buckets:
        lst.ops += bucket.ops
    lst.ops += lst.file_size * 2 + 1

def insertion_sort(l):
```

```
for i in xrange(1, l.file_size):
    j = i - 1
    key = l.get_byte_at_index(i)
    orig_val = l.get_raw_byte_at_index(i)
    while (j >= 0) and (l.get_byte_at_index(j) > key):
        l.set_byte(j + 1, l.get_raw_byte_at_index(j))
        j -= 1
    l.set_byte(j + 1, orig_val)
```

Jei skaičiai intervale pasiskirstę taip, kad į vieną kibirą įkris vienas skaičius, tai galime laikyti, kad kibiro rikiavimas yra pastovaus sudėtingumo operacija ($O(1)$). Susumavę operacijų kainas gauname

$T_{insertionSort} = C$

$T_{bucketSort}(n) = c_1 + 3(n+1)c_2 + 2nc_3 + nc_4 + 2c_5 + nc_6 + nc_7 + nc_8 + nc_9 + nc_{10} + nc_{11}$

$T_{bucketSort}(n) = n(3c_2 + 2c_3 + c_4 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11}) + c_1 + c_2 + 2c_5$

$T_{bucketSort}(n) = nC + D$

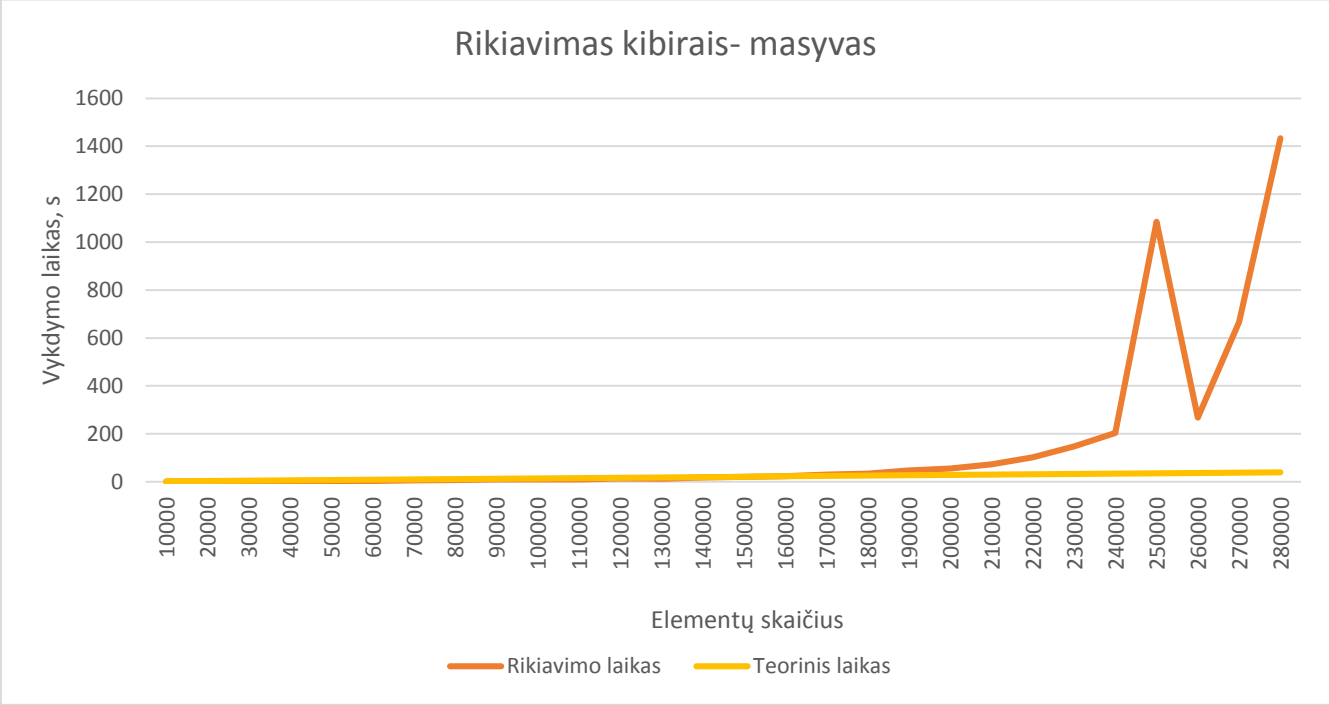
$T_{bucketSort}(n) = O(n)$

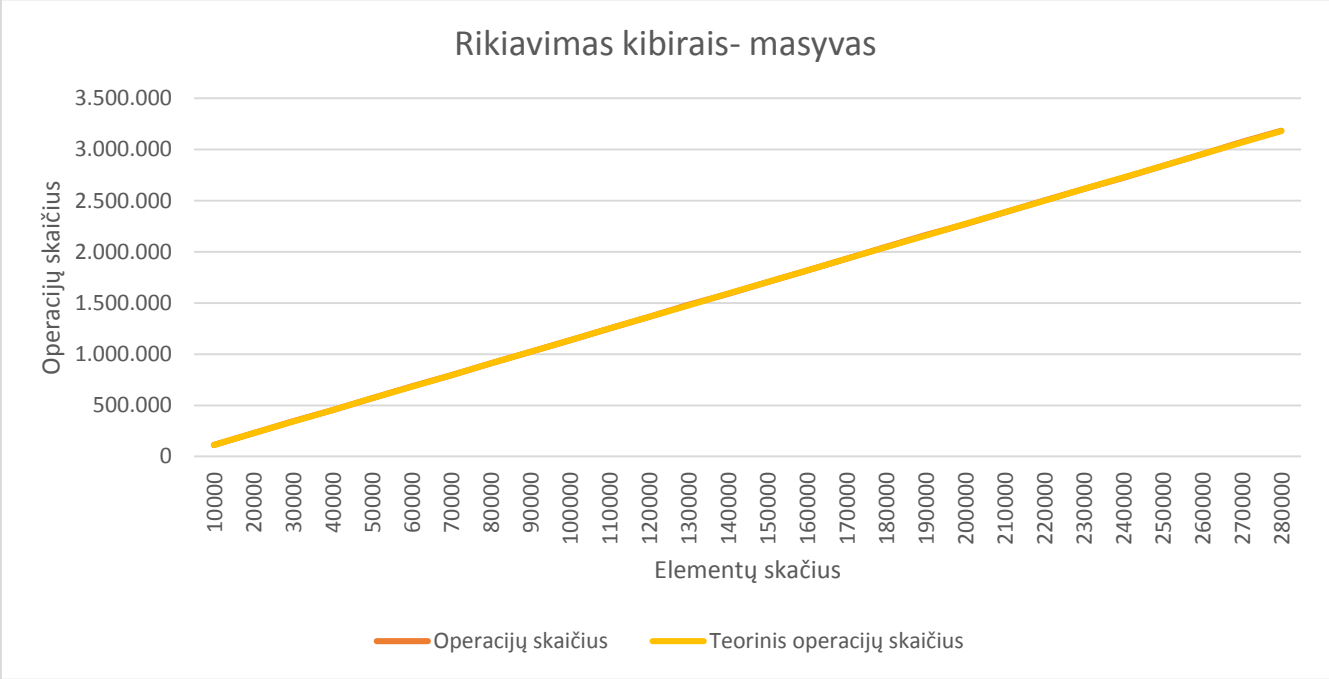
3.3.3. Algoritmo eksperimentinis tyrimas

Masyvo rikiavimas

Elementų skaičius	Rikiavimo laikas	Operacijų skaičius
10000	0,536589	113.799
20000	1,46219	227.653
30000	2,564288	342.625
40000	3,652695	453.629
50000	3,737394	568.672
60000	3,563442	683.307
70000	5,52515	795.173
80000	7,487865	911.076
90000	9,460135	1.023.040
100000	10,3311	1.135.461
110000	10,54669	1.250.960
120000	14,0756	1.364.549
130000	13,33672	1.480.444
140000	18,11189	1.589.254
150000	20,25268	1.703.713
160000	23,58647	1.816.637
170000	30,48125	1.932.346
180000	34,01655	2.046.946
190000	47,46778	2.164.850
200000	55,64844	2.269.198
210000	72,88033	2.385.625
220000	101,9843	2.500.219
230000	147,5749	2.614.169
240000	204,3626	2.723.586

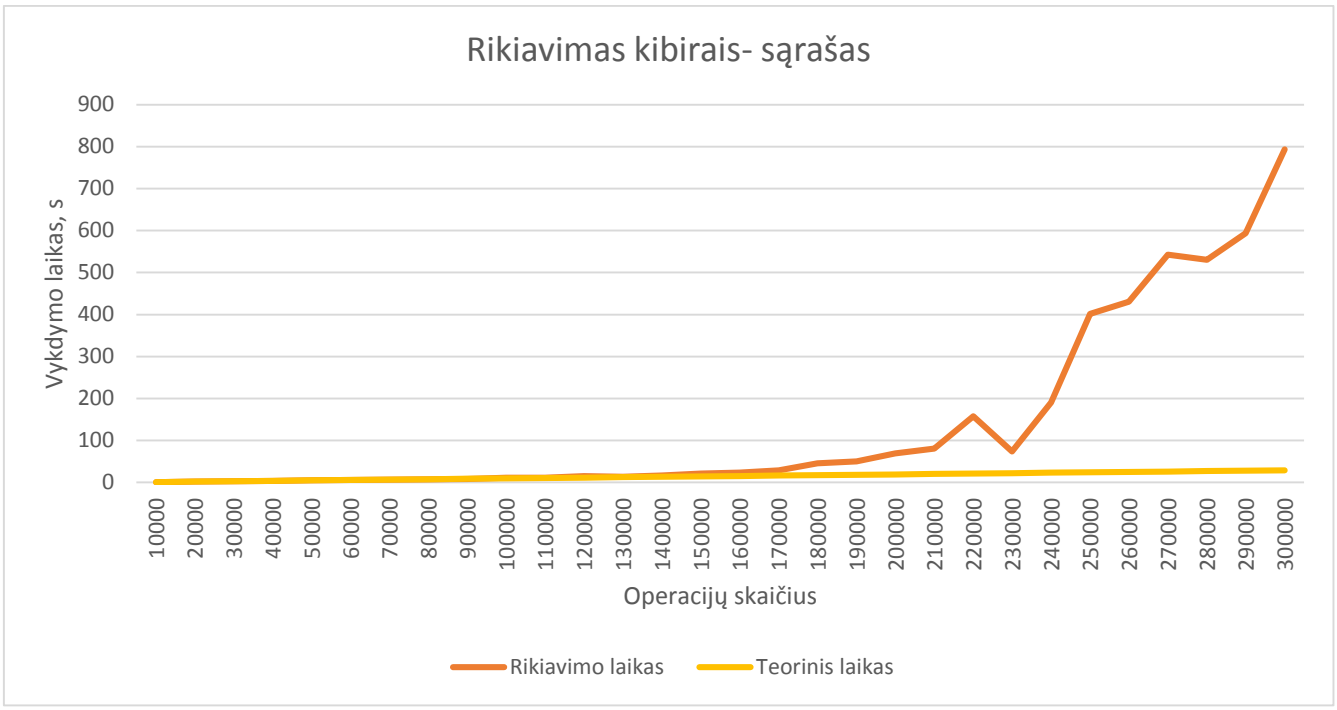
250000	1084,966	2.840.593
260000	268,345	2.955.424
270000	667,6368	3.072.924
280000	1433,356	3.181.847

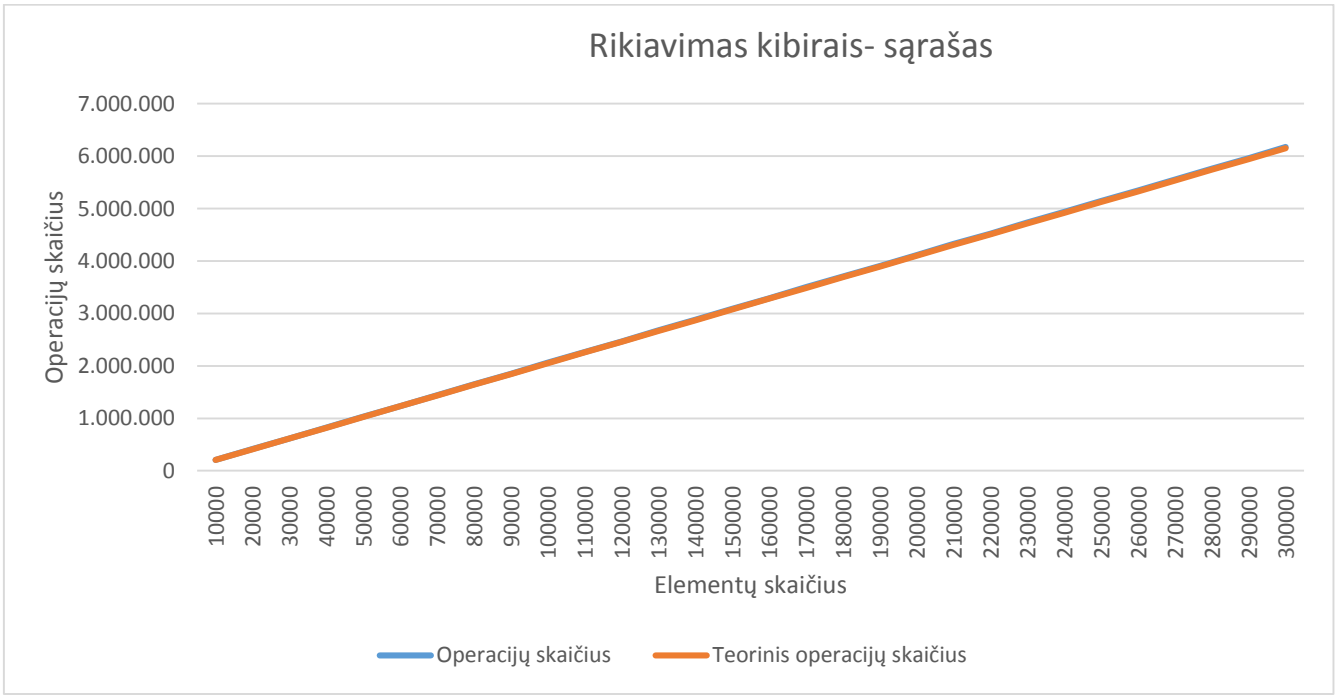




Sąrašo rikiavimas

Elementų skaičius	Rikiavimo laikas, s	Operacijų skaičius
10000	0,624527	205.888
20000	1,837687	412.165
30000	2,892835	617.827
40000	3,760724	825.117
50000	4,993715	1.029.808
60000	5,974612	1.233.952
70000	6,756028	1.440.996
80000	7,725545	1.645.247
90000	8,475076	1.851.828
100000	11,14049	2.060.922
110000	11,60961	2.265.450
120000	15,36238	2.466.120
130000	13,85761	2.676.054
140000	16,8663	2.878.079
150000	21,08524	3.085.322
160000	23,31932	3.288.407
170000	28,96221	3.500.068
180000	45,57485	3.702.871
190000	50,04542	3.907.945
200000	68,85936	4.110.959
210000	80,8772	4.324.845
220000	157,3409	4.521.946
230000	73,96379	4.737.541
240000	190,0844	4.935.673
250000	401,3497	5.141.599
260000	430,2922	5.348.277
270000	542,6417	5.552.734
280000	530,2178	5.759.113
290000	593,9525	5.963.553
300000	793,2806	6.173.924





Nukrypimai laiko grafikuose atsirado dėl labai didelio atidarytų failų skaičiaus, bet operacijų skaičiaus grafikai atitinka teorinius paskaičiavimus.

4. Išvados

Darbe realizuoti trys rikiavimo algoritmai- suliejimo, greitojo rikiavimo ir rikiavimo kibirais. Jiems leista vykdyti savo darbą po valandą su masyvais ir sąrašais didinant duomenų apimtis 10000 elementų.

Apskaičiuoti ir patikrinti algoritmų sudėtingumai:

- Suliejimo ir greitasis- $O(n \log_2 n)$
- Greitasis sąrašui- $O(n^2 \log_2 n)$
- Rikiavimas kibirais- $O(n)$

Geriausius laikus gavome su rikiavimu kibirais, bet jam reikia sukurti daug tarpinių failų. Greitasis algortimas dirba gana greitai su masyvu ir veikia tik duomenų failo viduje.

Tam tikros paklaidos atsirado dėl darbo diske ir kitų procesų, veikiančių kompiuteryje.

Operacijų skaičiaus priklausomybės nuo rikiuojamų elementų kiekio yra tokios, kaip tikėtasi, o laiko priklausomybės turi nukrypimų.