**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**ALGORITMŲ SUDARYMAS IR ANALIZĖ**

1 laboratorinis darbas

(Varianto Nr.1)

Atliko: Povilas Pučkovas

**KAUNAS, 2011**

**1. Rūšiavimas „CountingSort“ algoritmu**

**1.1 Rūšiavimo „CountingSort“ algoritmu teorinis įvertinimas**

Tai stabilus algoritmas, kurio laikas [Θ](http://en.wikipedia.org/wiki/Big_O_notation)(*n+k*) kur n ir k yra masyvų ilgiai A(duomenų masyvo) ir C(Skaičiavimo masyvo). Kad šis algoritmas būtų našus k neturi būti daug didesnis už n.

Jeigu minimali ir maksimali reikšmės nėra žinomos pirminis duomenų perėjimas bus reikalingas kad surasti tą ilgą. Jo sudėtingumas bus Θ(*n*).

Semantika:

1. Suskaičiuojama kiek kokių elementų yra masyve A
2. Elementų kiekiai saugomi atitinkamose C masyvo vietose (pvz ketvertu skaičius bus saugomas 4toje masyvo vietoje)
3. Iš eilės pereinama per C masyvą ir spausdiname einamojo elemento numeri tiek kartu kokia yra jo reikšmė.

**1.2 Rūšiavimo „CountingSort“ algoritmu išeities tekstai**

**3.2.1 CountingSort.java**

package countingsort;

import java.io.File;

import java.io.IOException;

public class CountingSort {

Failas failas;

double minRange;

double maxRange;

int operacijuSk=0;

private int generateResult(int numValues, Failas counter) throws IOException {

int opSk = 4;

int outputPos = 0;

for (int i = 0; i < numValues; i++) {

opSk +=9;

for (int j = 0; j < (int)counter.gautiSkaiciu(i); j++) {

failas.irasytiSkaiciu(i+minRange, outputPos);

outputPos++;

opSk += 6;

}

}

return opSk;

}

}

Failas failas = new Failas("failas");

failas.generuotiDuomenisBeLiekanos(1, 10, 3);

System.out.println("Pries CountingSort:");

failas.spausdintiSkaicius();

System.out.println("Po CountingSort:");

long start = System.currentTimeMillis();

CountingSort surikiuoti = new CountingSort("failas");

long time = System.currentTimeMillis() - start;

failas.spausdintiSkaicius();

failas.gautiFaila().setLength(0);

failas.gautiFaila().close();

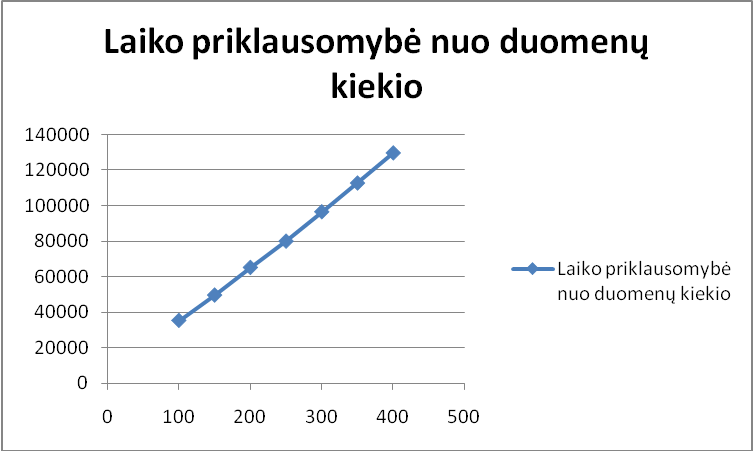
File trinti = new File("failas");

trinti.delete();

}

**1.3 Rūšiavimo „CountingSort“ algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (sekundėmis) | Operacijų kiekis |
| 100 | 53 | 3620 |
| 200 | 66 | 7117 |
| 300 | 137 | 11448 |
| 400 | 167 | 14936 |
| 500 | 247 | 18442 |
| 600 | 329 | 21927 |
| 700 | 381 | 25439 |



**2.Rikiavimas piramide**

**2.1 Rūšiavimo piramide algoritmo teorinis įvertinimas**

HeapSort algoritmas – kai rikiuojama duomenis sukeliant į krūvos(piramidinę) struktūrą

1. Sudaroma Krūvos duomenų struktūra
2. Iteratiškai iš sudarytos krūvos pašalinamos šaknys. Jas išsaugome mums reikiama tvarka: pvz., realizuojant masyvu, galime ją įrašyti į masyvo gale atsilaisvinusią vietą (pašalinus šaknį ji yra pakeičiama mažiausiu krūvos lapu).

Krūva sudaroma iš masyvo {a1, a2, ..., an}. sudaroma taip kad:

{a1', a2', ..., an'), kur ai' >= a2i' ir ai' >= a2i+1'

Antroji algoritmo dalis yra pats rikiavimas. Eilinėje iteracijoje imama krūvos viršūnė ir sukeičiama vietomis su paskutiniu masyvo elementu. Kadangi krūvos viršūnėje visada bus didžiausias elementas, tai jį sukeitę vietomis su paskutiniu masyvo elementu, žinosime, kad maksimalus masyvo elementas yra jo paskutinėje pozicijoje. Taip sukeitę elementus vietomis, turime atkurti krūvą. Tai padaryti paprasta, nes vietą pakeitęs būna tik vienas elementas (jis atsiduria viršūnėje). Šį elementą, jei jis nėra didžiausias, reikia perstumti keletu lygių žemiau, kol vėl gausime krūvą. Tai atlikę, pradedame naująją iteraciją, tik šį kartą jau nagrinėsime vienetu trumpesnį masyvą, nes didžiausias elementas, kuris anksčiau buvo viršūnėje, jau atsidūrė savo vietoje ir jo nagrinėti nereikia.

**2.2 Rūšiavimo piramide algoritmo išeities tekstai**

Algoritmas.java

package heapsort;

import java.io.FileNotFoundException;

import java.io.IOException;

public class rikiavimasPiramide {

private Failas failas;

rikiavimasPiramide(String failoVardas) throws FileNotFoundException, IOException {

failas = new Failas(failoVardas);

}

public void keistiFaila(String naujofailoVardas) throws IOException {

failas.gautiFaila().close();

failas.keistiFaila(naujofailoVardas);

}

private int perstatyti(int i, int j) throws IOException {

int operacijuSk = 3;

if (i <= j / 2) {

int k1 = 2 \* i;

int k2 = 2 \* i + 1;

operacijuSk += 6;

if (k2 > j) {

k2 = k1;

operacijuSk +=1;

}

int k;

operacijuSk+= 9;

if (failas.gautiSkaiciu(k1 - 1) > failas.gautiSkaiciu(k2 - 1)) {

k = k1;

} else {

k = k2;

}

operacijuSk+=7;

if (failas.gautiSkaiciu(i - 1) < failas.gautiSkaiciu(k - 1)) {

double laikinas = failas.gautiSkaiciu(i - 1);

failas.irasytiSkaiciu(failas.gautiSkaiciu(k - 1), (i - 1));

failas.irasytiSkaiciu(laikinas, (k - 1));

operacijuSk +=14;

operacijuSk += perstatyti(k, j);

}

}

return operacijuSk;

}

public int sudarytiRusiavimoMedi() throws IOException {

int operacijuSk = 0;

for (int i = (int) (failas.gautiSkaiciuKieki() / 2); i >= 1; i--) {

operacijuSk+=perstatyti(i, (int) failas.gautiSkaiciuKieki());

}

operacijuSk+=6\*(int)(failas.gautiSkaiciuKieki());

return operacijuSk;

}

public int rikiuoti() throws IOException {

int operacijuSk = 1;

operacijuSk += sudarytiRusiavimoMedi();

for (int i = (int) failas.gautiSkaiciuKieki(); i >= 2; i--) {

double laikinas = failas.gautiSkaiciu(0);

failas.irasytiSkaiciu(failas.gautiSkaiciu(i - 1), 0);

failas.irasytiSkaiciu(laikinas, i - 1);

operacijuSk += perstatyti(1, i - 1)+10;

}

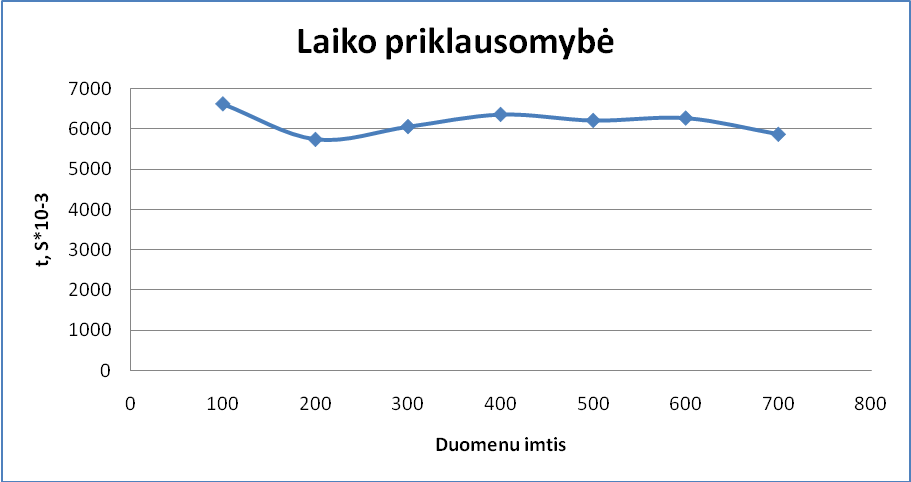
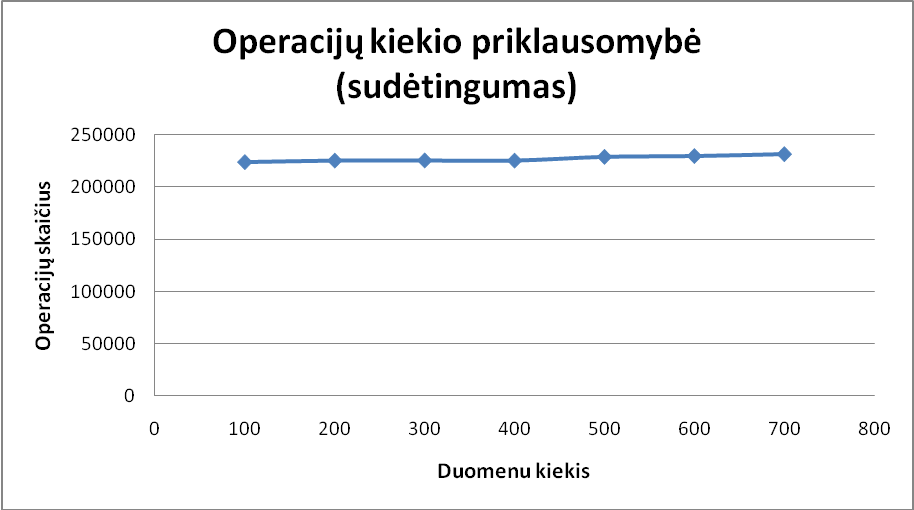
operacijuSk += 5\*((int) failas.gautiSkaiciuKieki()-2);

return operacijuSk;

}

**2.3 Rūšiavimo piramide algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (mili sekundėmis) | Operacijų kiekis |
| 100 | 6623 | 223897 |
| 200 | 5733 | 225329 |
| 300 | 6053 | 225456 |
| 400 | 6359 | 225314 |
| 500 | 6208 | 228811 |
| 600 | 6268 | 229525 |
| 700 | 5861 | 231315 |



**3. Rūšiavimas „InsertionSort“ algoritmu**

**3.1 Rūšiavimo „InsertionSort“ algoritmu teorinis įvertinimas**

Kiekvienos iteracijos metu elementas yra įdedamas į jau surikiuotą sąrašo dalį. Tuomet pereinamą per surikiuotąją dalį iš galo į priekį kol randama tinkama vieta tam elementui. Elementas tuomet įdedamas į tą vietą, o visi einantieji už jo yra perslenkami į kairę, būtent dėl šios priežasties, jeigu sąrašas jau yra surikiuotas, rikiavimas naudojant šį algoritmą būtų lygus O(*n*), kadangi kiekvienas elementas jau yra surikiuotas. Kitu atveju - jeigu sąrašas surikiuotas atvirkštine tvarka, rikiavimas trukų O(*n*2), kadangi jis ieškotų per visą surikiuotąją dalį, ir kiekvieną kartą įstatydamas elementą sukeistų toliau esančius.

**3.2 Rūšiavimo „InsertionSort“ algoritmu išeities tekstai**

**RikiavimoAlgoritmai.java**

package algoritmuanalize;

import java.io.FileNotFoundException;

import java.io.IOException;

public class RikiavimoAlgoritmai {

private Failas failas;

RikiavimoAlgoritmai(String failoVardas) throws FileNotFoundException, IOException{

failas = new Failas(failoVardas);

}

public int insertionSort() throws IOException{

int operacijuSk = 5;

int i, j;

double newValue;

for (i = 1; i < failas.gautiSkaiciuKieki(); i++) {

newValue = failas.gautiSkaiciu(i);

j = i;

operacijuSk +=11;

while (j > 0 && failas.gautiSkaiciu(j - 1) > newValue) {

failas.irasytiSkaiciu(failas.gautiSkaiciu(j-1), j);

j--;

operacijuSk+=12;

}

failas.irasytiSkaiciu(newValue, j);

}

return operacijuSk;

}

}

**Main.java**

package algoritmuanalize;

import java.io.FileNotFoundException;

import java.io.IOException;

diskoDuomenys duomenysDiske = new diskoDuomenys("duomenuFailas");

duomenysDiske.uzpildytiSveikaisSkaiciai(1, 100, 700);

long start = System.currentTimeMillis();

CountingSort surikiuoti = new CountingSort("duomenuFaila");

long time = System.currentTimeMillis() - start;

duomenysDiske.spausdintiSkaicius();

duomenysDiske.gautiFaila().setLength(0);

duomenysDiske.gautiFaila().close();

File trintiFaila = new File("duomenuFaila");

trintiFaila.delete();

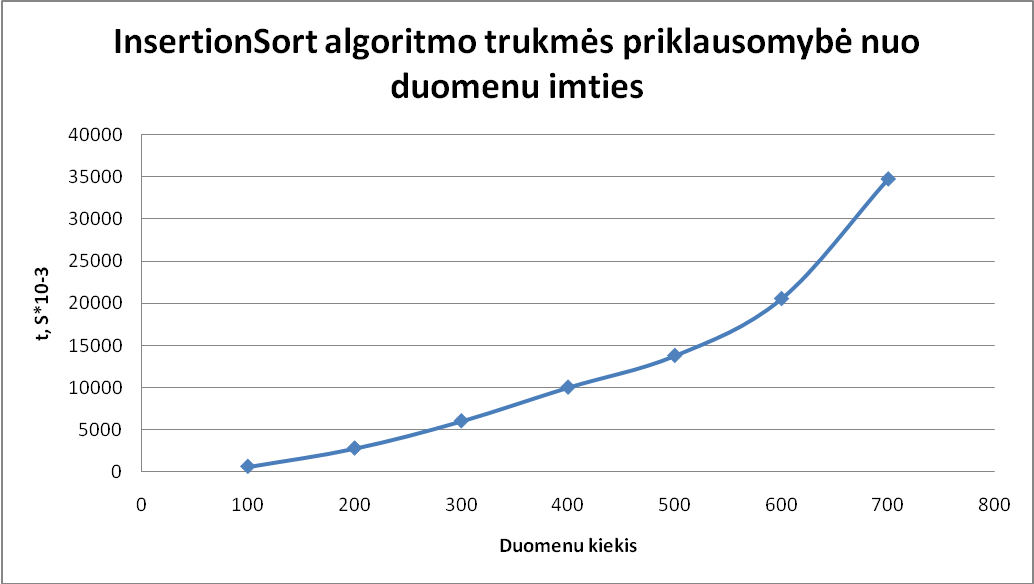
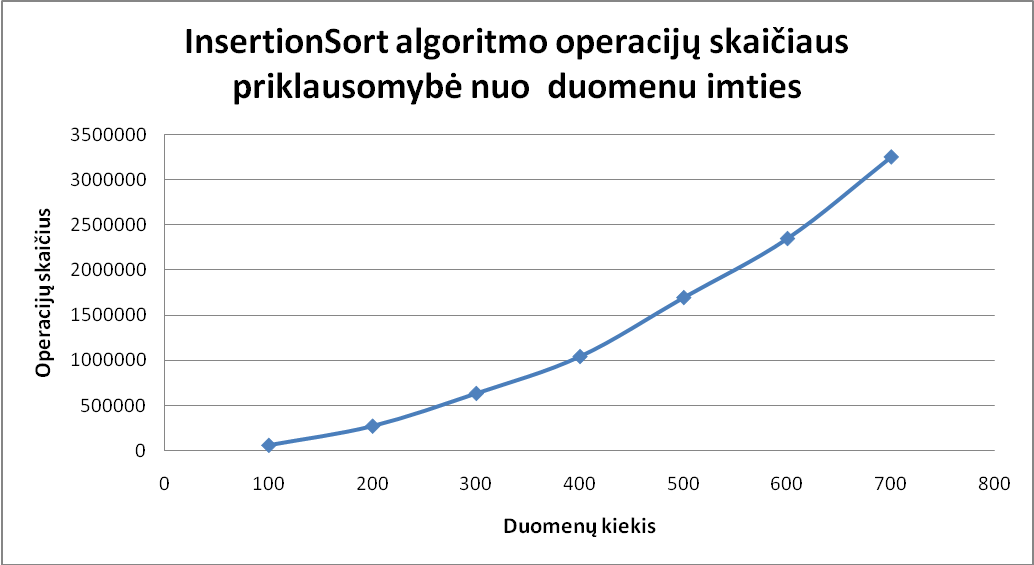
System.out.println(time);

System.out.println(surikiuoti.operacijuSk);

}

**3.3 Rūšiavimo „InsertionSort“ algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (sekundėmis) | Operacijų kiekis |
| 100 | 627 | 57307 |
| 200 | 2801 | 269185 |
| 300 | 6028 | 631105 |
| 400 | 10019 | 1039522 |
| 500 | 13770 | 1692100 |
| 600 | 20514 | 2344642 |
| 700 | 34686 | 3247219 |



**4. Išvados:**

InsertionSort algoritmas nors ir yra greitas ir labiausia taikytinas nedideliam duomenų kiekiui, tačiau iš grafikų matyti, kad nerekomenduotinas taikyti didelių imčių netvarkingiems duomenims rikiuoti. Labiausia priklauso nuo geriausio ir blogiausio atvejo, tačiau vidutiniškai dirba netoli geriausio atvejo. Rūšiavimas piramide labiausiai priklauso nuo struktūros ir yra nestabilus. Didžiausias trūkumas, kad duomenys turi atitikti piramidės struktūrą. Counting Sort

labai stabilus algoritmas. Didžiausias trūkumas jeigu duomenų imtis yra žymiai mažesnė už skaičiavimo imtį – tada našumas algoritmo mažas.