Tartu Ülikool
Matemaatika-informaatikateaduskond
Arvutiteaduse instituut

Erkki Alamäe

Kiirmeetodi ning biti- ja baidikaupa järjestamise võrdlemine standardi IEEE 754 ujukomaarvude sorteerimiseks

Praktiline töö

Juhendaja Ain Isotamm

Tartu 2006

Sisukord

Retsensioon...3
Kiirmeetod...4
Bitikaupa järjestamine...4
1. Sissejuhatus...7
2. Kiirmeetod...8
2.1 Algoritmi kirjeldus...8
2.2 Kiirmeetodi analüüs...9
3. Biti- ja baidikaupa järjestamine ehk positsioonimeetod.......................................11
3.1 Algoritmi kirjeldus...11
3.2 Positsioonimeetodi analüüs...12
4. Testid...13
Kokkuvõte...15
Abstract...16
Kirjandus...17
Lisad...18
Lisa 1. Testitulemuste tabelid...18
Lisa 2. Standardile IEEE 754 vastavate ujukomaarvude bitikaupa sorteerimisest 20
Lisa 3. Programmi kood...25

Retsensioon

Mait Raagi poolt 2005. a tehtud töö „Kiirmeetodi ja bitikaupa järjestamise võrdlemine" uuris kahte sorteerimisalgoritmi (stabiilne randomiseeritud kiirmeetod ning bitikaupa järjestamine) ning võrdles nende kiirust erinevate andmehulkade korral. Jõuti järelduse-le, et suuremate andmehulkade korral töötab bitikaupa järjestamine kiiremini. Järeldust kinnitasid ka Java-keeles realiseeritud katseprogrammide tulemused.

Algoritmide paremaks võrdlemiseks sorteeriti mõlema algoritmiga mittenegatiivseid täisarve. Samuti olid mõlemad uuritavad algoritmid realiseeritud stabiilselt (omavahel võrdsed elemendid ei vaheta sorteerimise käigus järjekorda) ning kummagi meetodi pu-hul ei tuntud huvi mälukasutuse vastu.

Töö headeks külgedeks võib lugeda ühtlast vormistust ning mõlema uuritava meetodi detailset kirjeldust ja analüüsi. Kasutatud kirjandusele oli viidatud. Lisaks oli kaasasolev programmikood põhjalikult kommenteeritud.

Töö puudusteks võib lugeda mõlema uuritava algoritmi ebaefektiivset implementatsi-ooni, mistõttu katsetulemused ei vastanud kummagi kirjeldatud meetodi keerukusklassi-le. Kiirmeetodi puhul oli kirjeldatud keerukusklassiks $O(n \times \log n)$, kus n on sorteerita-va järjendi pikkus, ning bitikaupa järjestamise puhul $O(k \times n)$, kus k on järkude arv, mil-le järgi sorteeriti. Siin ja edaspidi tähistus $O(f(n))$ tähendab suvalise funktsiooni $f(n)$ kor-ral, et mingi algoritmi tööaeg suureneva sisendandmete hulga n korral ei kasva kiiremini kui funktsioon $f(n)$, s.t tegu on algoritmi tööaja asümptootilise hinnanguga.

Järgnevalt uurime, millised olid baastöös realiseeritud meetodite keerukusklassid.

Kiirmeetod

On teada, et randomiseeritud kiirmeetodi keerukusklass on üldjuhul ning ka halvimal juhul $O(n \times \log n)$ (Wikipedia: Quicksort). Baastöös oli kiirmeetodi puhul ebaefektiivselt realiseeritud andmestruktuuri elementide lisamine (Raag 2005: Lisa 2, koodiread 275-286). Igakordsel massiivi elemendi lisamisel eraldati tervele massiivile ühe elemendi võrra suurem mälupiirkond, kopeeriti kõik elemendid ümber uude massiivi ning viimasele kohale lisati uus element. Kuna aga Java-keeles toimub int-tüüpi massiivi jaoks mälu eraldamine lineaarses ajas (Lindholm *et al* 1999: Initial Values of Variables; Array Variables), siis iga elemendi lisamisel uue jada loomine nõuab aega suurusjärgus

$$\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2) \quad ,$$

kus n on lõpliku järjendi suurus. Arvestame, et randomiseeritud kiirmeetod moodustab sel viisil kaks $n / 2$ elemendilist enam-vähem võrdse pikkusega massiivi („veelahkmest" väiksemad ja suuremad) ning loeme „veelahkmega" võrdsete arvude hulga tühiseks (vt 2. Kiirmeetod). Lisades hinnangusse ka massiivide ühendamiseks kuluva aja n, on töös (Raag 2005) realiseeritud kiirmeetodi tööaeg seega arvutatav võrrandist

$$T(n) = 2T(n/2) + 2(n/2)^2 + n = 2T(n/2) + O(n^2) \quad ,$$

kus $T(n)$ tähistab meetodi tööaega n-elemendilise massiivi korral ning liige n on keerukushinnangus loetud liikme $O(n^2)$ sisse. Põhiteoreemist (Kiho 2003: 20) saame, et töös realiseeritud kiirmeetodi keerukusklass on $O(n^2)$.

Bitikaupa järjestamine

Baastöös realiseeritud bitikaupa järjestamise meetod kasutab bitijärkude sorteerimiseks võrdlustel põninevat algoritmi (Raag 2005: Lisa 2, koodiread 376-396).

Algoritm alustab sorteerimist suurimast järgust, otsib mingit järku sorteerides üles esimese arvu, mille vaadeldavat järku bitt on 1, ja jätab sellele eelnevad („vasakpoolsed")

bitid meelde. Olgu see arv a_i. Seejärel hakkab algoritm alates positsioonist i otsima arvu, millel oleks eelnevad bitid samad, kuid vaadeldava bitijärgu kohal asuv bitt 0. Olgu see arv a_j. Seejärel vahetab algoritm omavahel arvud a_i ning a_j ja jätkab vaadeldavas järgus biti 1 otsimist positsioonist $i + 1$. Arvude a_j ja a_j otsimine katkestatakse, kui indeksid i või j on ületanud massiivi pikkuse n, ning alustatakse järgmise bitijärgu sorteerimist.

Et kirjeldatud algoritm kasutab võrdlemist, siis on tema keerukuseks halvimal juhul vähemalt $O(n \times \log n)$ (Kiho 2003: 71). Konstrueerime ka näite, kus algoritmi keerukus on $O(n^2)$. Olgu järjestikustel arvudel $a_1, a_2, ..., a_n$ mingis vaadeldavas bitijärgus kõik eelnevad bitid võrdsed ning olgu uuritavat järku bitid

$$\underbrace{11...1100...00}_{n} \quad .$$

Algoritm valib arvuks a_i esimese arvu ning arvuks a_j arvu positsioonis $a_{[n/2]}$ ning vahetab need, kusjuures $[n / 2]$ tähistab täisosa võtmist suurusest $n / 2$. Seejärel töötab algoritm järjekorras

$$\begin{array}{l} i:=2, \ \ j:=[(n/2)+1], \ \ a_i:=:a_j \\ i:=3, \ \ j:=[(n/2)+2], \ \ a_i:=:a_j \\ \qquad\qquad \vdots \\ i:=[(n/2)-1], \ \ j:=n, \ \ a_i:=:a_j \end{array} ,$$

kus märk := tähistab omistamist ning a_i :=: a_j muutujate a_i ning a_j väärtuste vahetamist. Ülaltoodud sammude järel on vaadeldavad arvud antud bitijärgu järgi sorteeritud. On lihtne näha, et selliseid samme sooritatakse $n / 2$ korda ning iga kord kulub arvu a_j leidmiseks $n / 2$ sammu, mistõttu töös realiseeritud meetod kulutab halvimal juhul n arvu sorteerimiseks ühe järgu järgi aega suurusjärgus

$$(n/2)^2 = n^2/4 = O(n^2) \quad .$$

Oletame, et algoritm saab ette rangelt kahaneva 2^k elemendist koosneva jada, kus arvudel on kahendesituses k järku (s.t iga arv on esindatud korra). Oletame, et arvud on märgita, s.t suurimat järku bitt on osa arvust, mitte ei tähista arvu märki. Esimese järgu puhul on arvude vastavad bitid sellises järjekorras, et eespool on kõik bitid 1 ning tagapool kõik bitid 0, teise järgu puhul kehtib sarnane järjestus järjendi kahes pooles pikkusega

n / 2, kolmanda järgu puhul igas n / 4 pikkuses osas jne, kusjuures i-nda järgu puhul on „lõikusid" kokku 2^{i-1}. Seega töös (Raag 2005) realiseeritud bitikaupa järjestamise puhul kulub n-elemendilise täisarvumassivi k järgu järgi sorteerimiseks aega halvimal juhul suurusjärgus

$$\sum_{i=1}^{k} 2^{i-1}\left(\frac{n}{2^i}\right)^2 = \frac{1}{2}n^2\sum_{i=1}^{k}\frac{2^i}{2^{2i}} = \frac{1}{2}n^2\sum_{i=1}^{k}\frac{1}{2^i} \approx \frac{1}{2}n^2 = O(n^2) \quad .$$

1. Sissejuhatus

Käesolevas töös uuritakse kolme algoritmi: vähimast bitijärgust alustavat bitikaupa sorteerimist (ingl k *LSD base two radix sort* või *least significant digit base 2 radix sort*), vähimast bitijärgust alustavat baidikaupa sorteerimist (ingl k *LSD base 256 radix sort*) ning stabiilset randomiseeritud kiirmeetodit (ingl k *stable randomized quicksort*). Baidikaupa sorteerimist uuritakse juhendaja soovitusel, ülejäänud algoritme (mõningate erinevustega) vaadeldi ka baastöös. Kõik meetodid on stabiilsed, s.t ei vaheta omavahel võrdsete elementide järjekorda.

Keskendutakse algoritmide ajalise keerukuse uurimisele ning ei huvituta mälu kasutusest. Erinevalt baastööst, kus uuriti mittenegatiivseid täisarve, vaadeldakse antud töös IEEE 754 standardile vastavate märgiga 32-bitiste ujukomaarvude sorteerimist.

Algoritmide keerukuse võrdlemiseks on Java-keeles koostatud katseprogramm.

Antud töö ning kirjutatud programmikood on elektrooniliselt kättesaadavad ka Internetist aadressil http://www.ut.ee/~erkkia/ajaa/.

2. Kiirmeetod

Kiirmeetod on C.A.R Hoare'i leiutatud „jaga-ja-valitse” sorteerimisalgoritm, mille keerukus on keskmiselt $O(n \times \log n)$ ning halvimal juhul $O(n^2)$. Kiirmeetodi tööpõhimõtteks on andmestikust mingi kindla elemendi valimine, mida käsitletakse „veelahkmena”. Sorteeritav andmestik jaotatakse kaheks osaks: ühte ossa pannakse veelahkmest väiksemad ja teise veelahkmest suuremad elemendid. Moodustatud osad sorteeritakse rekursiivselt, ühendatakse kokku, ning tagastatakse tulemus (Kiho 2003: 62).

Randomiseeritud kiirmeetod on kiirmeetodi modifikatsioon, kus veelahkmeks valitakse iga kord suvaline element jadast. Randomiseeritud kiirmeetodi halvima tööaja hinnang on $O(n \times \log n)$ (Wikipedia: Quicksort). Antud töös on realiseeritud stabiilne randomiseeritud kiirmeetod.

2.1 Algoritmi kirjeldus

Kiirmeetod (Lisa 3, read 113-137) saab töötlemiseks juhuslike ujukomaarvude massiivi. Et tegu on randomiseeritud kiirmeetodiga, valitakse etteantud massiivist veelahkmeks suvaline element (rida 119). Seejärel luuakse kolm abimassiivi: üks veelahkmest väiksemate arvude jaoks, teine veelahkmega võrdsete arvude jaoks ning kolmas veelahkmest suuremate arvude tarbeks (read 123-125). Etteantud järjend jaotatakse, võrreldes neid veelahkmega, vastavatesse abimassiividesse (read 128 – 132). Seejärel sorteeritakse rekursiivselt veelahkmest väiksemate ning suuremate arvude järjendid (veelahkmega võrdseid arve ei ole tarvis sorteerida, kuna nad on kõik omavahel võrdsed), ühendatakse kolm abimassiivi (järjekorras väiksemad, võrdsed ja suuremad) kokku ning tulemusena tagastatakse ühendatud massiiv (rida 136). Baasjuhul, kui etteantud järjendi pikkus on väiksem kui 1, ei sorteerita seda ning tagastatakse kohe etteantud massiiv (rida 115).

2.2 Kiirmeetodi analüüs

Näitame, et antud töös realiseeritud kiirmeetod on stabiilne. Omavahel võrdsed elemendid ei saa omavahel järjekorda vahetada, kuna etteantud järjend läbitakse järjest ning kõik vaadeldud elemendid lisatakse abimassiividesse samas järjekorras nagu neid vaadeldakse. Seega ei muuda üks kiirmeetodi samm omavahel võrdsete elementide järjekorda. Induktsiooni põhjal ei muuda seda ka kiirmeetodi rekursiivsed väljakutsed veelahkmest väiksemate ning suuremate arvude sorteerimiseks.

Näitame, et randomiseeritud kiirmeetodi keskmine kiirushinnang on $O(n \times \log n)$, kus n on sorteeritava massiivi elementide arv.

Elementide jaotamine kolme abimassiivi on käesolevas töös realiseeritud implementatsioonis keerukusega $O(n)$, kuna kasutatakse Java-keele dünaamiliselt suurendatavaid massiive võimaldavat andmestruktuuri *LinkedList,* mille korral massiivi suurendamine toimub konstantses ajas. Kolme abimassiivi ühendamine üheks järjendiks on samuti keerukusega $O(n)$. Olgu „veelahkmega" võrdsete elementide arv m. Kiirmeetodi esimesele kahele rekursiivsele väljakutsele – esimesele rekursioonitasemele - antakse töödelda kokku $n - m$ (ning kunagi mitte rohkem kui n) elementi. Sama hinnang kehtib teise, kolmanda, jne rekursioonitaseme kohta, kus i-inda rekursioonitaseme all mõistame kõiki meetodi väljakutseid, mida tegi suvaline rekursioonitaseme $i - 1$ meetod. Nagu mainitud, töödeldakse igal rekursioonitasemel ülimalt n elementi, seega tehakse igal tasemel tööd mahus $O(n)$.

Et veelahkmeks valitakse juhuslik element, võime lugeda, et keskmiselt on rekursiivselt sorteeritavad järjendid enam-vähem võrdse suurusega. Näitame, et kui nad ei ole võrdse suurusega, erineb rekursioonitasemete arv võrdse jaotamisega juhuga võrreldes sel juhul ainult mingi konstant c korda.

Oletame, et veelahe valitakse igal rekursioonitasemel sellisel viisil, et temast mõlemale poole jäävate elementide arvud on suhtes a / b, kus $a > b$ ning $1 < (a + b) / a < 2$. Tähistame $m = (a + b) / a$ ning uurime sellise jaotuse korral rekursioonitasemete arvu. Saame

$$\log_m n = \frac{\log_2 n}{\log_2 m} = c \log_2 n, \qquad c = \frac{1}{\log_2 m} \quad .$$

Näeme, et rekursioonitasemete arv erineb andmestiku alati pooleks jaotamisel ning alati suhtes a / b jaotamisel mingi konstant c korda. Rekursioonitasemeid on eelneva põhjal kokku $O(c \times \log_2 n) = O(\log n)$.

Seega, arvestades tasemete arvu ning igal tasemel tehtavat tööd, on randomiseeritud kiirmeetodi keskmiseks keerukushinnanguks $O(n \times \log n)$.

3. Biti- ja baidikaupa järjestamine ehk positsioonimeetod

Biti- ja baidikaupa sorteerimine on mõlemad positsioonimeetodi erijuhud. Positsioonimeetodi tööpõhimõtteks on arvude sorteerimine järkude kaupa, s.t sorteeritavaid elemente vaadeldakse kui „liitvõtmeid", millel on k erinevat järku (Kiho 2003: 72). Leidub kaks põhimõtteliselt erinevat positsioonimeetodi varianti: üks alustab sorteerimist kõige väiksemast järgust, liikudes suuremate poole, ning teine kõige suuremast, liikudes väiksemate poole. Antud töös oleme realiseerinud kõige väiksemast järgust alustava positsioonimeetodi.

Positsioonimeetod kasutab järkude sorteerimisel abimeetodina tavaliselt loendamismeetodit, millisel juhul tema keerukus on $O(k \times n)$, kus n on etteantud järjendi suurus (Kiho 2003:72).

Antud töös sorteerime 32-bitiseid arve bitikaupa ($k = 32$) ning baidikaupa ($k = 4$).

3.1 Algoritmi kirjeldus

Algoritm saab töötlemiseks ette suvaliste ujukomaarvude järjendi, mida hakatakse vaatlema bittide kaupa. Enne sorteerimist rakendatakse kõigile arvudele teatud teisendust (read 21-30, teisenduse kohta vt Lisa 2). Arvude töötlemist alustatakse kõige väiksemast ehk „parempoolseimast" järgust. Järkude vaatlemiseks arvust eraldi defineeritakse bitimask (rida 32). Loendamismeetodi abil sorteeritakse kõik arvud vaadeldavate bittide järgi mittekahanevalt (read 50–80). Kui arvud massiivis on ühe järgu põhjal sorteeritud, siis nihutatakse bitimaski järgmise järgu peale („vasakule") ning sorteeritakse massiiv uue järgu järgi mittekahanevalt. Kui bitimaski kõige kõrgemast järgust edasi nihutada, „kaovad" tema bitid 1 ära ning bitimask võrdub nulliga (vastav kontroll on real 50). Siis on ka järjend võtmete kasvamise järjekorras sorteeritud. Peale loendamismeetodi töö lõppu rakendame kõigile arvudele eelpoolmainitud teisenduse pöördteisendust (read 95-104, pöördteisenduse kohta vt Lisa 2). Peale viimast tegevust on ujukomaarvude järjend mittekahanevalt sorteeritud.

3.2 Positsioonimeetodi analüüs

Näitame, et positsioonimeetodi alamalgoritmina kasutatav loendamismeetod töötab ajas $O(n)$, kus n on sorteeritava massiivi elementide arv. Loendamismeetod läbib järjendi, lugedes kokku erinevate võtmete esinemissageduse, see on keerukusega $O(n)$. Seejärel summeerib loendamismeetod võtmete esinemissagedused nii, et alustades vähimast võtmest liidetakse igale järgnevale võtme esinemissagedusele eelneva võtme esinemissagedus. Ka see tegevus töötab ajas $O(n)$. Võtmete esinemissageduste massiivis on pärast viimast tegevust elementide arv, mis on antud võtmest väiksemad. Arvestades elemendist väiksemate elementide esinemiste arvu, kopeeritakse abimassiivi sorteeritavad elemendid. Ka see on keerukusega $O(n)$, kuna iga elemendi korral on kohe teada tema koht massiivis. Et kõik kirjeldatud tegevused on keerukusega $O(n)$, on loendamismeetodi kogukeerukus $O(n)$.

Positsioonimeetodi keerukus on $O(k \times n)$, kus k on järkude arv, kuna loendamismeetodit rakendatakse igale järgule.

Näitame, et järkude sorteerimiseks kasutatav loendamismeetod on stabiilne. Kui mingi järgu k_i korral on iga võtme korral temast väiksemate võtmete arv leitud, hakatakse sorteeritavaid elemente abimassiivi lisama, alustades jada lõpust. Igal elemendi a_j lisamisel abimassiivi vähendatakse mingist võtmest d väiksemate elementide esinemiste arvu $s(d)$ ning element a_j kopeeritakse abimassiivi kohale $s(d)$. Kui algoritm leiab järgmise elemendi $a_{j'}$, kusjuures järjendi läbimise järjekorra tõttu $j' < j$, millel on võti $d' = d$, lisatakse ta abimassiivis elemendi a_j ette, sest $s(d)$ on vahepeal vähenenud. Et selline elementide a_j ning $a_{j'}$ vaheline järjekord kehtis ka enne abimassiivi kopeerimist, on loendamismeetod stabiilne.

Vähimast järgust alustav positsioonimeetod on stabiilne, kuna sorteeritavate arvude iga järgu järjestamiseks kasutatav loendamismeetod on stabiilne.

4. Testid

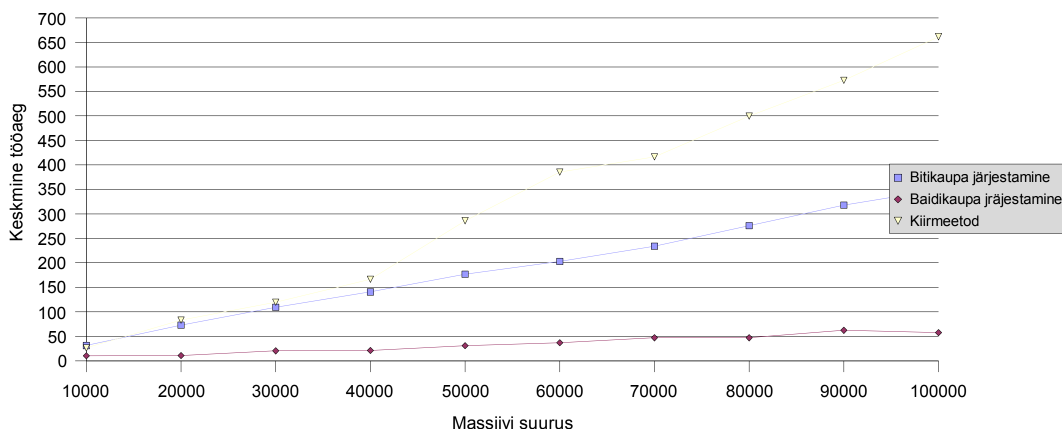
Uuritavate meetodite ajalise keerukuse uurimiseks koostati testimisprogramm (Lisa 3, read 208-362). Seadeldis testib kõigi kirjeldatud algoritmide tööaega suvaliste ujukoma-arvude massiividel, mis sisaldavad võrdselt negatiivseid ja positiivseid ujukomaarve. Seejuures viiakse testid läbi kolmes massiivide suuruse vahemikus: kõigepealt 100, 200, ..., 1000; seejärel 10 000, 20 000, ..., 100 000 ning viimases testis vahemikus 100 000, 200 000, ..., 1 000 000.

Testid viidi läbi kolm korda ning uuriti keskmisi tulemusi. Testimiseks kasutati Pentium M 1,73 GHz protsessori ja 1024 MB DDR-II SDRAM mäluga arvutit.

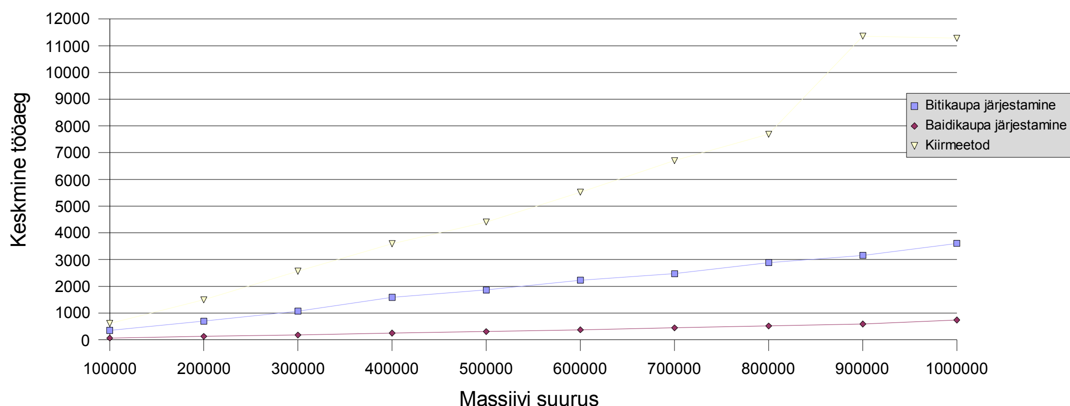
Et vahemikus 100, 200, ..., 1000 olid sorteerimismeetodite tööajad üldjuhul alla 1 ms (vt Lisa 2), siis ei pea me vajalikuks siinkohal tööaegade graafikut ära tuua. Mõningatel juhtudel esinesid sorteerimisaegades „hälbed”. Näiteks saadi bitikaupa sorteerimise puhul 100-elemendilise massiivi sorteerimise mõõdetud ajaks 2. katsel 16 ms, samal ajal kui 1. katsel kulus 1000-elemendilise massiivi massiivi sorteerimiseks alla 1 ms. Võib oletada, et sorteerimismeetodi töö ajal käivitus Java virtuaalmasina automaatne mälu vabastamise protseduur, tegi oma töö ära, ning lasi sorteerimismeetodil edasi töötada, mistõttu pikenes mõõdetud sorteerimise aeg.

Vahemikus 10 000, 20 000, ..., 100 000 tulid välja erinevused erinevate algoritmide tööaegade vahel. Jooniselt 1 on näha, et nii biti- kui ka baidikaupa järjestamise tööajad kasvavad lineaarselt. Tuletame siinkohal meelde, et positsioonimeetodi keerukushinnanguks on $O(n \times k)$, kus n on sorteeritava järjendi suurus ning k järkude arv, mille järgi sorteeritakse. Antud töös bitikaupa meetodi puhul $k_{bitt} = 32$, baidikaupa järjestamise puhul $k_{bait} = 4$ ning $k_{bitt} / k_{bait} = 8$. Ka jooniselt on näha, et biti- ning baidikaupa järjestamise tööaeg erineb suurusjärgus 8 korda. Jooniselt 1 on ka näha, et kiirmeetodi tööaja kasv suurenevate andmemahtude korral kiireneb, mis on kooskõlas meie tulemusega, et kiirmeetodi keerukus on $O(n \times \log n)$.

Joonis 1. Algoritmide tööaja sõltuvus massiivi elementide arvust.



Veelgi suuremate andmemahtude (vt joonis 2) korral suurenesid veelgi vahed positsioonimeetodite ning kiirmeetodi tööaegade vahel. Kui 100 000 elemendi puhul oli bitikaupa järjestamise ning kiirmeetodi tööaja suhe 662 ms / 349 ms ≈ 2, siis 1 000 000 elemendi puhul on vastav suhe 11 281 ms / 3604 ms ≈ 3, mis illustreerib suurenevate andmehulkade korral kiirmeetodi tööaja kiiremat kasvu võrreldes positsioonimeetoditega.



Joonis 2. Algoritmide tööaja sõltuvus massiivi elementide arvust.

Kokkuvõte

Käesolevas töös võrreldakse kolme sorteerimisalgoritmi: stabiilset randomiseeritud kiirmeetodit (*stable randomized quicksort*), vähimast järgust alustavat bitikaupa järjestamist (*least significant digit base two radix sort*) ja baidikaupa järjestamist (*least significant digit base 256 radix sort*). Kõigi kolme algoritmiga sorteeritakse standardile IEEE 754 vastavaid 32-bitiseid ujukomaarve. Töös näidatakse, et biti- ja baidikaupa järjestamisega saab mittekahanevalt sorteerida positiivseid ja negatiivseid ujukomaarve. Nähakse, et kiirmeetodi keskmine ja samuti halvim tööaeg on keerukusega $O(n \times \log n)$, kus n on sorteeritava massiivi elementide arv. Samuti nähakse, et biti- ja baidikaupa järjestamine on keerukusega $O(n \times k)$, kus n on sorteeritava massiivi elementide arv ning k järkude arv, mille järgi sorteeritakse. Algoritmide keerukusklasse kontrolliti testidega, mis näitasid, et biti- ja baidikaupa järjestamine on eriti suuremate andmehulkade korral kordades kiiremad kiirmeetodist.

Abstract

Comparison of quicksort, base two radix sort and base 256 radix sort for sorting IEEE 754 floating-point numbers
Practical work

Erkki Alamäe

Current work compares three sorting algorithms (stable randomized quicksort, least significant digit base two radix sort, and least significant digit base 256 radix sort) for sorting IEEE 754 32-bit floating point numbers. It is shown that the average and and also worst-case complexity of quicksort is $O(n \times \log n)$, where n is the length of the array to be sorted. Complexity of radix sorts is shown to be $O(n \times k)$, where n is the length of the array to be sorted and k is the number of radices in a number. It is also shown that IEEE 754 32-bit floating poing numbers can be sorted with radix sorts. Complexities of algorithms are tested with experiments with different sizes of n and k.

Kirjandus

Raag, M. 2005. Kiirmeetodi ja bitikaupa järjestamise võrdlemine. Käsikiri Tartu Ülikooli arvutiteaduse instituudis

Lindholm et al = T.Lindholm, F.Yellin, 1999. The JavaTM Virtual Machine Specification. Sun Microsystems, Inc. (http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html). Viimati külastatud 27.11.2006.

Kiho, J. 2003. Algoritmid ja andmestruktuurid. Tartu: Tartu Ülikooli Kirjastus.

Wikipedia (http://en.wikipedia.org). Viimati külastatud 27.11.2006

Lisad

Lisa 1. Testitulemuste tabelid

Bitikaupa sorteerimine

Massiivi suurus	100	200	300	400	500	600	700	800	900	1000
1. katse	0	0	0	0	0	0	0	15	0	0
2. katse	16	0	0	0	15	0	0	16	0	16
3. katse	0	15	0	0	0	16	0	0	0	15
Keskmine	5	5	0	0	5	5	0	10	0	10

Massiivi suurus	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1. katse	31	63	109	141	187	203	234	281	328	360
2. katse	32	78	109	141	172	203	234	266	313	344
3. katse	31	78	110	140	172	203	234	281	312	344
Keskmine	31	73	109	141	177	203	234	276	318	349

Massiivi suurus	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1. katse	360	688	1046	1688	1844	2235	2484	2812	3203	3859
2. katse	344	703	1109	1625	1907	2265	2500	3047	3141	3468
3. katse	344	703	1063	1453	1844	2188	2437	2797	3125	3485
Keskmine	349	698	1073	1589	1865	2229	2474	2885	3156	3604

Tabel 1. Bitikaupa järjestamisel kulunud aeg (ms).

Baidikaupa sorteerimine

Massiivi suurus	100	200	300	400	500	600	700	800	900	1000
1. katse	0	0	0	0	0	0	0	0	0	0
2. katse	0	0	0	0	16	0	0	0	0	0
3. katse	0	0	0	0	0	0	0	0	0	0
Keskmine	0	0	0	0	5	0	0	0	0	0

Massiivi suurus	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1. katse	16	16	15	15	31	32	47	47	62	63
2. katse	0	16	31	16	31	32	47	47	62	63
3. katse	15	0	15	32	31	47	47	47	63	47
Keskmine	10	11	20	21	31	37	47	47	62	58

Massiivi suurus	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1. katse	63	125	172	250	313	375	438	500	562	750
2. katse	63	125	187	250	296	375	468	547	641	812
3. katse	62	125	172	250	313	375	438	500	562	657
Keskmine	63	125	177	250	307	375	448	516	588	740

Tabel 2. Baidikaupa järjestamisel kulunud aeg (ms).

Kiirmeetod

Massiivi suurus	100	200	300	400	500	600	700	800	900	1000
1. katse	0	15	0	0	0	0	16	0	0	0
2. katse	0	15	0	0	0	0	0	0	0	0
3. katse	15	0	0	0	0	0	0	0	0	0
Keskmine	5	10	0	0	0	0	5	0	0	0

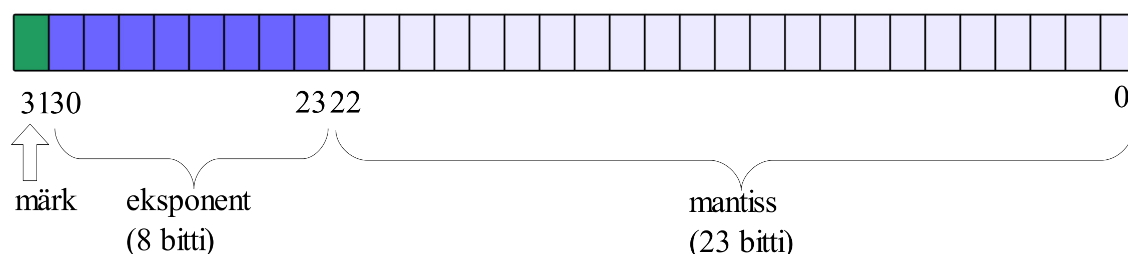
Massiivi suurus	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1. katse	32	125	125	187	296	375	453	515	547	688
2. katse	31	63	109	156	281	453	406	468	562	672
3. katse	16	62	125	157	281	328	390	516	609	625
Keskmine	26	83	120	167	286	385	416	500	573	662

Massiivi suurus	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1. katse	641	1453	2938	3407	4594	5484	6532	7969	15468	9421
2. katse	594	1547	2406	4093	4547	5672	7109	7656	9234	12703
3. katse	594	1484	2344	3281	4063	5407	6469	7437	9375	11718
Keskmine	610	1495	2563	3594	4401	5521	6703	7687	11359	11281

Tabel 3. Kiirmeetodil kulunud aeg (ms).

Lisa 2. Standardile IEEE 754 vastavate ujukomaarvude bitikaupa sorteerimisest

Standardile IEEE 754 vastavate 32-bitiste ujukomaarvude (edaspidi lihtsalt ujukomaarvude) esitust arvuti mälus illustreerib järgnev joonis.



Joonis 1. Ujukomaarvu (32 bitti) esitus standardi IEEE 754 järgi

Arvu eksponenti kujutatakse märgita kaheksabitilise täisarvuna, kusjuures kõik eksponendid on nihkes arvu -127 võrra. Mantissi kujutatakse arvu 2 astmetena kahanevas järjekorras, kusjuures normaliseeritult. Mantissi järjekorras „nullis" bitt, mida normaliseeritud kuju tõttu välja ei kirjutata, tähistab arvu $2^{eksponent}$, järgmine bitt arvu $2^{eksponent-1}$ jne.

Näide. Kui eksponendiosas on bitid 10000000, mida tõlgendatakse kümnendsüsteemis kui arvu 128 - 127 = 1, ning mantissi osas esimesed bitid 1101 ning ülejäänud nullid, siis see esitus tähistab ujukomaarvu

$$1\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+1\cdot 2^{-2}=2+1+0+\frac{1}{4}=3,25 \quad .$$

Ujukomaarv on miinusmärgiga, kui tema märgibitt on 1, ning positiivne, kui märgibitt on 0.

Järgnevalt näitame, et leiduvad funktsioonid Φ ja Φ^{-1}, mis teisendavad ujukomaarvu bitikaupa järjestatavaks arvuks ja tagasi.

Definitsioon. Täielik bitimask on bitimask, mis koosneb ainult bittidest 1.

Lemma 1. Rakendades mingile bitijärjendile *b* XOR-tehet täieliku bitimaskiga, pööra-me selle bitijärjendi bitid ümber (bitist 1 saab 0, bitist 0 saab 1).

Tõestus. Loogikatehe XOR tagastab 1, kui parajasti üks kahest kohakuti olevast bitist on 1, ning 0 muul juhul. Et täielikus bitimaskis on kõik bitid ühed, siis *b* bitid, mis olid enne nullid, saavad XOR-tehte tulemusena ühtedeks ning kõik bitid, mis olid ühed, saa-vad nullideks.

Definitsioon. Ütleme, et vaatleme mingi fikseeritud eksponendiosa korral mantissiosa *a* väärtust arvu 2 astmete summana, kui kirjutame $V_m(a)$.

Definitsioon. Maksimaalse mantissiosaga arv on selline arv, mis on kõigist omavahel võrdse eksponendiosaga arvudest on kõige suurem. Selle arvu mantissiosa kutsume maksimaalseks mantissiosaks.

Lemma 2. Mingi fikseeritud eksponendiosa korral koosneb maksimaalne mantissiosa *m* ainult bittidest 1.

Tõestus. Mantissiosa *a* esitab arvu 2 astmete summat eksponendi kahanevas järjekorras. Et $2^i > 0$ iga täisarvulise *i* korral, siis iga liidetav suurendab mantissi väärtust $V_m(a)$. Järelikult on summa maksimaalne, kui kõik võimalikud liidetavad on esitatud, ehk kõik bitid mantissiosas on väärtusega 1.

Lemma 3. Olgu arvul mingi fikseeritud eksponendi korral mantissisosa *a* ning tema väärtus $V_m(a)$. Olgu sellesama eksponendi korral maksimaalne mantiss *m* ja tema väärtus $V_m(m)$. Rakendades XOR-tehet täieliku bitimaskiga mantissiosale *a*, saame tehte tulemusena arvu mantissiossa väärtuse $V_m(m) - V_m(a)$.

Tõestus. Olgu meil mantissiosa *b*, mille saame, kui rakendame mantissiosale *a* täieliku bitimaskiga XOR-tehet. Meil on kaks arvu

$$\begin{array}{ll} a: & a_{22}a_{21}a_{20}\ldots a_0 \\ b: & b_{22}b_{21}b_{20}\ldots b_0 \end{array},$$

kus a_i ja b_i ($0 \le i \le 22$) tähistavad üksikuid bitte arvude *a* ja *b* mantissiosades. Lemma (1) põhjal $a_i \neq b_i$ iga *i* korral. On ilmne, et liites mantissiosadele *a* ja *b* vastavad arvu 2 astmed, saame maksimaalsele mantissiosale *m* vastava väärtuse $V_m(m)$. Seega võime kir-

jutada $V_m(m) = V_m(a) + V_m(b)$. Saame $V_m(b) = V_m(m) - V_m(a)$, mida oligi tarvis tõestada.

Definitsioon. Defineerime sarnaselt mantissiosale eksponendiosa a, maksimaalsele eksponendiosale vastavad bitid m ning temale vastava väärtuse täisarvuna $V_e(m)$.

Lemma 4. Rakendades täieliku bitimaskiga XOR-tehet eksponendiosale a, saame tulemuseks eksponendiossa väärtuse $V_e(m) - V_e(a)$.

Tõestus. Analoogiline lemma (3) tõestusega, kuna ujukomaarvu eksponenti esitatakse märgita täisarvuna.

Definitsioon. Defineerime funktsiooni $\Phi(x)$, kus x tähistab kõiki ujukomaarvule vastavaid bitte. Funktsioon Φ rakendab argumendile järgnevaid loogikatehteid:

$$\Phi(x) := \begin{cases} x\ XOR\ \mathrm{0x80000000}, & x\ on\ positiivne \\ x\ XOR\ \mathrm{0xFFFFFFFF}, & x\ on\ negatiivne \end{cases}.$$

Defineerime ka funktsiooni $\Phi^{-1}(x)$:

$$\Phi^{-1}(x) = \begin{cases} x\ XOR\ \mathrm{0x80000000}, & x\ on\ negatiivne \\ x\ XOR\ \mathrm{0xFFFFFFFF}, & x\ on\ positiivne \end{cases}.$$

Siin 0x80000000 ning 0xFFFFFFFF tähistavad bitimaske kuueteistkümnendsüsteemis.

Lemma 5. Funktsioonid $\Phi(x)$ ning $\Phi^{-1}(x)$ on bijektiivsed.

Tõestus. Jaotame tõestuse kahte ossa.

(1) Injektiivsus. Igal IEEE 754 standardi järgi esitataval ujukomaarvul on normaliseerituse tingimuse tõttu vaid üks võimalik bitiesitus. Oletame, et leiduvad mingid arvud a ja b, et $a \neq b$ ja $\Phi(a) = \Phi(b)$. Funktsioonide Φ ja Φ^{-1} definitsioonidest saame, et sel juhul peavad a ja b olema samamärgilised. Järelikult teisendatakse neid samade bitimaskidega. On lihtne näha, et XOR-tehe ei saa kaht erinevat bitijada võrdsete bitimaskide korral teisendada võrdseks bitijadaks.

(2) Sürjektiivsus. Sarnane injektiivsuse tõestusega.

Lemma 6. Näitame, et $\Phi^{-1}(\Phi(x)) = I(x)$, kus I on samasusfunktsioon.

Tõestus. Jaotame tõestuse kahte ossa: juhul, kui arv x on positiivne ning juhul kui ta on negatiivne.

(1) Oletame, et ujukomaarv x on positiivne. Sel juhul Φ vahetab tema märgibiti, misjärel ta on negatiivne. Rakendades seejärel funktsiooni Φ^{-1} arvule x, pööratakse nüüd uuesti arvu x märgibitt, misjärel oleme tagasi saanud algse arvu.

(2) Kui x on negatiivne, pöörab Φ kõik tema bitid ümber. Kuna pööratakse ka tema märgibitt, on x nüüd positiivne arv. Funktsioon Φ^{-1} saab ette positiivse arvu ning pöörab taas kõik tema bitid ümber. Oleme tagasi saanud algse arvu x.

Definitsioon. Defineerime sarnaselt funktsioonidele V_m ja V_e funktsiooni V, mis seab igale bitijärjendile vastavusse järjekorranumbri. Vaatlemata täpsemat järjekorranumbrit, võime mingi kahe ujukomaarvu a ja b puhul kirjutada $\mathrm{V}(a) < \mathrm{V}(b)$, mis tähendab „arv a satub positsioonimeetodiga sorteeritud järjendis ettepoole arvust b” ning analoogiliselt $V(a) > V(b)$.

Teoreem 1. Olgu meil kaks negatiivset ujukomaarvu a ja b, kusjuures $a > b$. Negatiivsete ujukomaarvude esituse tõttu kehtib $V(a) < V(b)$. Funktsioon Φ teisendab neid nii, et $V(\Phi(a)) > V(\Phi(b))$.

Tõestus. Jaotame tõestuse kaheks osaks.

(1) Olgu arvude a ja b eksponendiosad võrdsed, s.t $V_e(a) = V_e(b)$. Rakendame arvudele a ja b funktsiooni Φ ning tähistame saadud arve a' ja b'. Lemma (3) põhjal

$$\begin{array}{l} V_m(a') = V_m(m) - V_m(a) \\ V_m(b') = V_m(m) - V_m(b) \end{array} .$$

Kuna $a > b$ ning vastavate arvude eksponendiosad on võrdsed, siis negatiivsete ujukomaarvude esituse tõttu $V_m(a) < V_m(b)$. Kasutades viimast võrratust, saame

$$V_m(m) - V_m(a) > V_m(m) - V_m(b) \quad ,$$

millest

$$V_m(a') > V_m(b') \quad .$$

Et funktsioon Φ pöörab negatiivsete ujukomaarvude märgibitid ümber, on a' ja b' positiivsed arvud. Et arvude a ja b eksponendiosad olid võrdsed enne Φ rakendamist, on nad võrdsed ka pärast Φ rakendamist. Seega erinevad arvud a' ja b' vaid mantissiosade poolest. Kuna $V_\mathrm{m}(a') > V_m(b')$, saame, et $V(a') > V(b')$, mida oligi tarvis tõestada.

(2) Oletame, et arvude a ja b eksponendiosad on erinevad. Tõestus on sarnase osaga (1), aga rakendame lemmat (4).

Teoreem 2. Kui rakendada funktsiooni $\Phi(x)$ kõigile ujukomaarvudele sorteeritavas massiivis, sorteerida massiiv positsioonimeetodiga ning seejärel rakendada kõigile arvudele funktsiooni $\Phi^{-1}(x)$, on massiivis esialgselt olnud arvud mittekahanevalt sorteeritud.

Tõestus. Oletame, et rakendame positsioonimeetodit ilma teisendavate funktsioonideta ujukomaarvude järjendile. Positiivsed arvud sorteeritakse omavahel mittekahanevasse järjekorda, kuna vähimast mantissiosa bitist alustades on iga temale järgnev bitt suurema „tähtsusega”, v.a märgibitt. Seetõttu on positiivsed arvud juba vajalikus järjestuses.

Kui massiivis on nii negatiivseid kui positiivseid arve, loetakse kõik negatiivsed arvud suuremaks positiivsetest arvudest, kuna nende „suurim järk” ehk märki tähistav bitt on 1. Lisaks, iga kahe negatiivse arvu a ja b korral, kus $a < b$, sorteerib positsioonimeetod nad nii, et b asub järjendis eespool kui a, kuna arv a on bittesituses järkude kaupa vaadeldes suurem.

Rakendame kõigile arvudele järjendis funktsiooni Φ. Funktsiooni definitsioonist on näha, et märgibitt pööratakse nii negatiivsetel kui ka positiivsetel arvudel ümber. Seega satuvad negatiivsed arvud sorteeritud massiivi positiivsete arvude ette.

Positiivsete arvude korral enne järjestamist ülejäänud bitte ei muudeta, seega säilib arvude omavaheline mittekahanev järjestus. Negatiivsete arvude puhul pööratakse kõik bitid ümber. Teoreemist (1) saame, et funktsioon Φ teisendab negatiivsed arvud a ja b (seejuures endiselt $a < b$) kujudele a' ja b', et enne positsioonimeetodi rakendamist kehtib $V(a') < V(b')$. Sorteerides jada positsioonimeetodil, saame, et a' asub järjendis eespool kui b'. Rakendades nüüd kõigile elementidele massiivis funktsiooni Φ^{-1}, saame lemma (6) põhjal tagasi esialgsed arvud a ja b, ning nüüd asub a sorteeritud järjendis eespool kui b. Et öeldu kehtib iga võimaliku negatiivse arvupaari a ja b kohta, on kõik negatiivsed arvud pärast kirjeldatud samme mittekahanevas järjekorras sorteeritud. Et negatiivsed arvud asuvad sorteeritud järjendis eespool kui positiivsed arvud, oleme teoreemi tõestanud.

Lisa 3. Programmi kood

Sorteerimine.java:

```
001 import java.io.FileWriter;
002 import java.io.IOException;
003 import java.util.LinkedList;
004
005 public class Sorteerimine {
006   /**
007    * Bitt- ja baithaaval järjestamise meetod IEEE 754 standardile
vastavate ujukomaarvude
008    * massiivi sorteerimiseks.
009    * Nn "arvujärkude" sorteerimiseks kasutatakse
loendamismeetodit.
010    * Kasutab lisamälu mahus O(n), kus n on etteantud järjendi
pikkus.
011    * @param A sorteeritav täisarvujada
012    * @param radix kas 2 või 256 - vastavalt bitt- ja baithaaval
järjestamisele
013    */
014   public static void radix_sort_count(float[] A, int radix){
015
016     // Järgnev realiseerib bijektsiooni, mille abil saame
sorteerida negatiivseid
017     // ja positiivseid ujukomaarve loendamismeetodil.
018
019     // Iga arvu korral vahetame tema märgi ja kui märgibitt on 1,
020     // siis rakendame XOR-tehet kogu arvule täisbitimaskiga
(0xFFFFFFFF).
021     for(int i = 0; i < A.length; i++) {
022       // kui arv on negatiivne
023       if((Float.floatToIntBits(A[i]) & 0x80000000) == 0x80000000)
024         // rakendame XOR-tehet kogu arvule
025         A[i] = Float.intBitsToFloat(Float.floatToIntBits(A[i]) ^
0xFFFFFFFF);
026       // kui arv on positiivne
027       else
028         // vahetame arvu märki
029         A[i] = Float.intBitsToFloat(Float.floatToIntBits(A[i]) ^
0x80000000);
030     }
031
032     int positsioon = 0xFFFFFFFF & (radix - 1); // esimese
positsiooni bittmask
033     int nihe = (radix == 2) ? 1 : 8;           // bitt- või
baithaaval
034
035     float[] B = new float[A.length]; // abimassiiv tulemuse
salvestamiseks
036     int[] c = new int[radix];        // loendurid positsioonide
```

```
väärtuste jaoks
037
038     // Mälu kokkuhoiu jaoks kasutame vaheldumisi massiive A ja B,
039     // selleks defineerime viidad
040     //   from - millisest massiivist võtame sorteeritavad
väärtused
041     //   to   - millisesse massiivi salvestame tulemuse
042     float[] from = A;
043     float[] to   = B;
044
045     // Järjestame alustades vähima järguga "positsioonist".
046     // Positsioon, mille järgi järjestatakse, "nihkub" järjest
vasakule,
047     // suuremate järkudega positsioonide poole.
048     // Tsükkel lõpeb, kui ta on bitinihetega läinud kaugemale kui
vaadeldavate
049     // täisarvude suurima järguga number.
050     for (int tsykkel = 0; positsioon != 0; tsykkel++){
051
052       // nullime loendurid
053       for(int i = 0; i < c.length; i++) c[i] = 0;
054
055       // loendame "positsioonide" järgi
056       for(int i = 0; i < from.length;  i++)
057         c[((Float.floatToIntBits(from[i]) & positsioon) >>>
nihe*tsykkel) ]++;
058
059       // summeerime loendid
060       for(int i = 1; i < c.length; i++) c[i] += c[i-1];
061
062       // kopeerime jadast from jadasse to
063       for(int i = from.length - 1, j; i >= 0; i--){
064         j = --c[(Float.floatToIntBits(from[i]) & positsioon) >>>
nihe*tsykkel];
065         to[j] = from[i];
066       }
067
068       // vahetame massiivid from ja to ära, et taaskasutada juba
eraldatud mälu
069       if(A == from){
070         from = B;
071         to = A;
072       } else {
073         from = A;
074         to = B;
075       }
076
077       // Vaadeldav positsioon "nihkub" edasi suuremate arvujärkude
poole
078       // ühe järgu võrra
079       positsioon <<= nihe;
080     }
081
082     // Kui praegu A == from, siis enne viimasest tsüklist
väljumist kehtis A == to.
083     // See tähendab, et sorteeritud jada asub massiivis B.
```

```
084     if(A == to){
085       // Kopeerime massivi B sisu massiivi A.
086       for(int i = 0; i < from.length; i++) from[i] = to[i];
087     }
088
089     // Rakendame taas eelnevalt mainitud bijektsiooni igale
arvule.
090     // Kui arvu märgibitt on 1 ehk arv on negatiivne, oli ta enne
bijektsiooni rakendamist
091     // positiivne. Järelikult vahetame tema märgi tagasi.
092     // Kui arvu märgibitt on 0, siis oli ta enne bijektsiooni
rakendamist negatiivne ning
093     // rakendasime temale XOR-tehet täisbitimaskiga.
094     // Seega tuleb temale taas täisbitimaskiga XOR-tehet
rakendada.
095     for(int i = 0; i < A.length; i++){
096       // kui arv oli positiivne
097       if((Float.floatToIntBits(A[i]) & 0x80000000) == 0x80000000)
098         // vahetame märgi tagasi
099         A[i] = Float.intBitsToFloat(Float.floatToIntBits(A[i]) ^
0x80000000);
100       // kui arv oli negatiivne
101       else
102         // rakendame talle XOR-tehet täisbitimaskiga
103         A[i] = Float.intBitsToFloat(Float.floatToIntBits(A[i]) ^
0xFFFFFFFF);
104     }
105   }
106
107   /**
108    * Stabiilselt realiseeritud kiirmeetod ujukomaarvude järjendi
sorteerimiseks.
109    * Kasutab lisamälu mahus O(n * log n), kus n on etteantud
järjendi suurus.
110    * @param A etteantud ujukomaarvude järjend
111    * @return sorteeritud ujukomaarvude järjend
112    */
113   public static LinkedList<Float> quicksort(LinkedList<Float> A){
114     // rekursiooni baasjuhtum
115     if(A.size() <= 1) return A;
116
117     // "Veelahe", millest ühele poole jäävad temast väiksemad
118     // ning teisele poole temast suuremad elemendid.
119     float veelahe = A.get((int)(Math.random() * A.size()));
120
121     // Abimälu veelahkmega võrdsete ning veelahkmest väiksemate
122     // ja suuremate elementide hoidmiseks.
123     LinkedList<Float> vaiksemad = new LinkedList<Float>();
124     LinkedList<Float> vordsed   = new LinkedList<Float>();
125     LinkedList<Float> suuremad  = new LinkedList<Float>();
126
127     // Itereerime üle etteantud järjendi ning kogume elemendid
abimassiividesse.
128     for(float a : A){
129       if(a < veelahe) vaiksemad.addLast(a);
130       else if (a == veelahe) vordsed.addLast(a);
```

```
131       else suuremad.addLast(a);
132     }
133
134     // Sorteerime rekursiivselt veelahkmest väiksemad ning
suuremad elemendid
135     // ning tagastame väiksemad, võrdsed ning suuremad elemendid
ühendatuna ühte massiivi.
136     return yhenda_float(quicksort(vaiksemad), vordsed,
quicksort(suuremad));
137   }
138
139   /**
140    * Meetod kolme ujukomajärjendi ühendamiseks.
141    * @param vaiksemad kiirmeetodi mõistes veelahkmest väiksemad
arvud
142    * @param vordsed kiirmeetodi mõistes veelahkmega võrdsed arvud
143    * @param suuremad kiirmeetodi mõistes veelahkmest suuremad
arvud
144    * @return kolmest parameetrite järjekorras ühendatud
ujukomaarvude massiiv
145    */
146   public static LinkedList<Float> yhenda_float(
147       LinkedList<Float> vaiksemad,
148       LinkedList<Float> vordsed,
149       LinkedList<Float> suuremad){
150
151     // Eraldame eraldi mälu ühendatud massiivi jaoks.
152     LinkedList<Float> yhendatud = new LinkedList<Float>();
153
154     yhendatud.addAll(vaiksemad);
155     yhendatud.addAll(vordsed);
156     yhendatud.addAll(suuremad);
157
158     return yhendatud;
159   }
160   /**
161    * Abimeetod etteantud pikkusega suvalistest ujukomaarvudest
koosneva
162    * jada loomiseks. Tagastab java süsteemse andmestruktuuri
float[].
163    * @param pikkus soovitava jada pikkus
164    * @param suurim suurim element, mida jadasse soovime
165    * @return suvaliste ujukomaarvude massiiv
166    */
167   private static float[] loo_jada_float(int pikkus, int suurim){
168     // Eraldame mälu massiivi jaoks.
169     float[] jada = new float[pikkus];
170
171     // Lisame massiivi järjest suvalisi ujukomaarve.
172     for(int i = 0; i < pikkus; i++){
173       // Pooled lisatavatest arvudest on negatiivsed ja pooled
174       // positiivsed arvud.
175       if(i % 2 == 0)
176         jada[i] = (float)(suurim * Math.random());
177       else
178         jada[i] = -(float)(suurim * Math.random());
```

```
179     }
180     // Tagastame massiivi.
181     return jada;
182   }
183
184   /**
185    * Abimeetod etteantud pikkusega suvalistest ujukomaarvudest
koosneva
186    * jada loomiseks. Tagastab andmestruktuuri tüübist
LinkedList<Float>.
187    * @param pikkus soovitava jada pikkus
188    * @param suurim suurim element, mida jadasse soovime
189    * @return suvaliste ujukomaarvude massiiv
190    */
191   private static LinkedList<Float> loo_jada_linked_float(int
pikkus, int suurim){
192     // Salvestame viida ujukomaarvude massiivile.
193     LinkedList<Float> jada = new LinkedList<Float>();
194
195     // Lisame massiiiv järjest suvalisi ujukomaarve.
196     for(int i = 0; i < pikkus; i++){
197       // Pooled lisatavatest arvudest on negatiivsed ja
198       // pooled positiivsed arvud.
199       if(i % 2 == 0)
200         jada.addLast((float)(suurim * Math.random()));
201       else
202         jada.addLast(-(float)(suurim * Math.random()));
203     }
204     // Tagastame massiivi.
205     return jada;
206   }
207
208   /**
209    * Peameetod. Teostab biti- ja baidikaupa järjestamise ning
kiirmeetodi
210    * tööaja testid. Töötab massiividel suurusjärgus 100...1000,
10000...100000
211    * ja 100000...1000000.
212    * @param args
213    */
214   public static void main(String args[]){
215
216     // Massiivid suurusvahemikus 100...1000.
217     int[][] testi_tulemused_vaike = new int[10][2];
218     testi_tulemused_vaike[0][0] = 100;
219     testi_tulemused_vaike[1][0] = 200;
220     testi_tulemused_vaike[2][0] = 300;
221     testi_tulemused_vaike[3][0] = 400;
222     testi_tulemused_vaike[4][0] = 500;
223     testi_tulemused_vaike[5][0] = 600;
224     testi_tulemused_vaike[6][0] = 700;
225     testi_tulemused_vaike[7][0] = 800;
226     testi_tulemused_vaike[8][0] = 900;
227     testi_tulemused_vaike[9][0] = 1000;
228
229     // Massiivid suurusega 10000...100000.
```

```
230     int[][] testi_tulemused_keskmine = new int[10][2];
231     testi_tulemused_keskmine[0][0] = 10000;
232     testi_tulemused_keskmine[1][0] = 20000;
233     testi_tulemused_keskmine[2][0] = 30000;
234     testi_tulemused_keskmine[3][0] = 40000;
235     testi_tulemused_keskmine[4][0] = 50000;
236     testi_tulemused_keskmine[5][0] = 60000;
237     testi_tulemused_keskmine[6][0] = 70000;
238     testi_tulemused_keskmine[7][0] = 80000;
239     testi_tulemused_keskmine[8][0] = 90000;
240     testi_tulemused_keskmine[9][0] = 100000;
241
242     // Massiivid suurusega 100000...1000000.
243     int[][] testi_tulemused_suur = new int[10][2];
244     testi_tulemused_suur[0][0] = 100000;
245     testi_tulemused_suur[1][0] = 200000;
246     testi_tulemused_suur[2][0] = 300000;
247     testi_tulemused_suur[3][0] = 400000;
248     testi_tulemused_suur[4][0] = 500000;
249     testi_tulemused_suur[5][0] = 600000;
250     testi_tulemused_suur[6][0] = 700000;
251     testi_tulemused_suur[7][0] = 800000;
252     testi_tulemused_suur[8][0] = 900000;
253     testi_tulemused_suur[9][0] = 1000000;
254
255     // Testime bitikaupa sorteerimist ning kirjutame tulemused
failidesse.
256     testi_positsioonimeetod(testi_tulemused_vaike, 2);
257     kirjuta_faili(testi_tulemused_vaike, "bitsort_vaike.txt");
258     testi_positsioonimeetod(testi_tulemused_keskmine, 2);
259     kirjuta_faili(testi_tulemused_keskmine,
"bitsort_keskmine.txt");
260     testi_positsioonimeetod(testi_tulemused_suur, 2);
261     kirjuta_faili(testi_tulemused_suur, "bitsort_suur.txt");
262
263     // Testime baidikaupa sorteerimist ning kirjutame tulemused
failidesse.
264     testi_positsioonimeetod(testi_tulemused_vaike, 256);
265     kirjuta_faili(testi_tulemused_vaike, "bytesort_vaike.txt");
266     testi_positsioonimeetod(testi_tulemused_keskmine, 256);
267     kirjuta_faili(testi_tulemused_keskmine,
"bytesort_keskmine.txt");
268     testi_positsioonimeetod(testi_tulemused_suur, 256);
269     kirjuta_faili(testi_tulemused_suur, "bytesort_suur.txt");
270
271     // Testime kiirmeetodit ning kirjutame tulemused failidesse.
272     testi_kiirmeetod(testi_tulemused_vaike);
273     kirjuta_faili(testi_tulemused_vaike, "kiirmeetod_vaike.txt");
274     testi_kiirmeetod(testi_tulemused_keskmine);
275     kirjuta_faili(testi_tulemused_keskmine,
"kiirmeetod_keskmine.txt");
276     testi_kiirmeetod(testi_tulemused_suur);
277     kirjuta_faili(testi_tulemused_suur, "kiirmeetod_suur.txt");
278   }
279
280   /**
```

```
281    * Meetod positsioonimeetodi testimiseks. Võimaldab testida nii
biti-
282    * kui ka baidikaupa sorteerimist.
283    * @param testi_tulemused kahedimensionaalne massiiv, milles on
etteantud sorteeritava
284    *                        järjendite suurused ja millesse
salvestatakse ka testide tulemused
285    * @param radix Arvujärk, mille järgi sorteerime, antud juhul
kas 2 või 256.
286    */
287   static void testi_positsioonimeetod(int[][] testi_tulemused, int
radix){
288     // Käime läbi massiivi testi_tulemused
289     for(int i = 0; i < testi_tulemused.length; i++){
290       // Loome nõutud pikkusega massiivi suvaliste
ujukomaarvudega.
291       float[] test_jada = loo_jada_float(testi_tulemused[i][0],
1000000);
292       // Mõõdame meetodi töö algusaja.
293       long algus = System.currentTimeMillis();
294       // Sorteerime järjendi.
295       radix_sort_count(test_jada, radix);
296       // Mõõdame meetodi töö lõpuaja.
297       long l6pp = System.currentTimeMillis();
298       // Salvestame meetodi tööaja.
299       testi_tulemused[i][1] = (int)(l6pp-algus);
300     }
301   }
302
303   /**
304    * Protseduur kiirmeetodi testimiseks.
305    * @param testi_tulemused kahedimanesionaalne massiiv, milles on
etteantud
306    *                        sorteeritavate järjendite suurused
ning millesse
307    *                        salvestatakse ka testide tulemused
308    */
309   static void testi_kiirmeetod(int[][] testi_tulemused){
310     // Käime läbi massiivi testi_tulemused
311     for(int i = 0; i < testi_tulemused.length; i++){
312       // Loome nõutud pikkusega massiivi suvaliste
ujukomaarvudega.
313       LinkedList<Float> test_jada =
loo_jada_linked_float(testi_tulemused[i][0], 10000000);
314       // Mõõdame meetodi töö algusaja.
315       long algus = System.currentTimeMillis();
316       // Sorteerime järjendi.
317       quicksort(test_jada);
318       // Mõõdame meetodi töö lõpuaja.
319       long l6pp = System.currentTimeMillis();
320       // Salvestame meetodi tööaja.
321       testi_tulemused[i][1] = (int)(l6pp-algus);
322     }
323   }
324
325   /**
```

```
326    * Abimeetod testitulemuste faili kirjutamiseks.
327    * @param testi_tulemused kahedimensionaalne massiiv, milles on
salvestatud sorteeritava
328    *                        järjendi pikkus ning selle
sorteerimiseks kulunud tööaeg.
329    * @param faili_nimi failinimi, kuhu salvestame tulemused
330    */
331   static void kirjuta_faili(int[][] testi_tulemused, String
faili_nimi){
332     // Defineerime viida FileWriter-tüüpi objektile.
333     FileWriter fileWriter = null;
334     // Järgnev võib tekitada erindeid, seega ümbritseme ta try-
blokiga.
335     try{
336       // Loome kirjutamiseks uue faili.
337       fileWriter = new FileWriter(faili_nimi);
338       // Kirjutame faili taanetega eraldatult sorteeritavate
massiivide pikkused.
339       for(int i = 0; i < testi_tulemused.length; i++){
340         fileWriter.append(testi_tulemused[i][0] + "\t");
341       }
342       // Kirjutame faili reavahetuse märgi.
343       fileWriter.append("\n");
344       // Kirjutame faili massiivide sorteerimiseks kulunud
tööajad,
345       // kusjuures eraldi tulemused eraldame taanetega.
346       for(int i = 0; i < testi_tulemused.length; i++){
347         fileWriter.append(testi_tulemused[i][1] + "\t");
348       }
349     } catch (IOException e){
350       System.out.println("Viga faili " + faili_nimi + "
avamisel.");
351     } finally {
352       // Kirjutame faili kindlasti kõvakettale välja.
353       try {
354         fileWriter.flush();
355         fileWriter.close();
356       } catch (IOException e){
357         System.out.println("Viga faili " + faili_nimi + "
kirjutamisel.");
358       }
359     }
360
361   }
362 }
```