Sorteerimine

Kui töödeldavad andmed on mingil viisil organiseeritud, on võimalik nende hulgas kiiremini otsida (erinevad otsimismeetodid!). Andmete järjekorda seadmine ei ole tavaliselt eesmärk omaette, vaid ta on vahesamm teiste tegevuste juurde jõudmiseks. Seetõttu kuuluvad ka arvutiteaduse huviorbiiti mitmesugused algoritmid, mille abil andmeid järjestada saab.

Sorteerimine on suvalises järjekorras olevate kirjete ümberjärjestamine seni, kuni nad paiknevad mittekahanevalt vastavalt mingi võtmevälja väärtusele.

Sorteerimisprobleemi võib formaalsemalt sõnastada ka järgmiselt:

```
Sisendiks on n arvu (a_1, a_2, ... a_n)
```

Väljundiks on samad arvud ümberjärjestatult $(a'_1, a'_2, \dots a'_n)$ nii, et $a'_1 \le a'_2 \le \dots a'_n$

Tavaliselt järjestatakse kirjed võtmeväärtuse kasvavas järjekorras (väiksem esimene), kuid võib ka järjestada kahanevas järjekorras (suurem esimene).

Tavaliselt tuuakse sorteerimisalgoritmide kohta näited, mis paigutavad kirjed ümber mittekahanevasse järjekorda. On vaja natuke mõtelda, et algoritm muuta vastupidiseks.

Kirje (record) (seda mõistet kasutatakse sorteerimisest rääkides ühe elemendi mõttes, st sorteeritakse kirjeid) võib olla üksik märk, täisarv, reaalarv, string - sel juhul on ta ise võtmeks. Keerulisemate andmeagregaatide puhul fikseeritakse võtmeks üks kirje väli (näiteks inimese puhul pikkus, vanus vms) ja temale vastavalt järjestatakse kõik andmed (koos võtmega tõstetakse ringi kõik temaga seotud andmed).

Võtmeid võib olla ka mitu, st esmane, teisane jne. võti.

Kui kirjes on mitu andmeelementi ja ühe järgi sorteeritakse, siis teised kirje osad tulevad loomulikult võtmega kaasa ja seetõttu pole õige väita, et üks kirje on teisest väiksem. Korrektsem on öelda, et üks kirje eelneb või järgneb teisele. Algoritmide lihtsamaks mõistmiseks vaatame neid täisarvude massiivi näitel.

Sorteerimisalgoritme võib toimimise järgi jagada mitmeti.

Esimene võimalik jaotus on

- **sisemine sorteerimine** (*internal sort*) kõik kirjed mahutatakse korraga põhimällu ja kogu tegevus toimub seal. Andmed on mälus massiivina või mõne dünaamilise andmestruktuurina;
- **väline sorteerimine** (external sort) sorteeritav andmekogum ei mahu korraga põhimällu, osa on kettal ja kuidagi tuleb andmed järjestada.

Vaadeldavad näited kuuluvad kõik sisemise sorteerimise alla.

Sisemise sorteerimise algoritme on palju. Konkreetses olukorras tuleb teha vailk. Valikul lähtutakse järgmistest **kriteeriumidest**:

- 1) **Tööaeg** kui kaua läheb N kirje sorteerimiseks ja kuidas aeg kasvab, kui kirjete arv näiteks kahekordistub? Teisisõnu hinnatakse algoritmi ajalist keerukust.
- 2) **Mälu** kas saab sorteerida koha peal (vaja on lisaks ruumi vaid mõnele kirjele) või on ruumi küllalt ja võib arvestada kasvõi N täiendava kirjega ja kirjutada kogu andmemassiiv ümber teise kohta.
- 3) **Esialgne järjestatus** kui esialgu on kirjed juhuslikus järjekorras (täitsa segamini), on sobivamad ühed algoritmid. Kui kirjed on peaaegu järjestatud, siis on sobivamad teised algoritmid.

Inga Petuhhov TLÜ 1/13

4) **Programmeerimiskeel** - pole enam eriti aktuaalne, aga osa algoritme nõuab rekursiooni kasutamist ja mõned vanemad keeled ei võimalda seda meetodit kasutada.

Sorteerimismeetodid võib vastavalt nende tööloogikale jagada klassideks:

- 1) vahelepanekuga sorteerimised;
- 2) vahetussorteerimised;
- 3) loendussorteerimised;
- 4) mestimissorteerimised;
- 5) valik- ja puusorteerimised.

Järgnevas materjalis on näiteid kõigist nimetatud klassidest.

Sorteerimisalgoritm on **stabiilne** (*stable*), kui kaks sama võtmeväärtusega kirjet jäävad ka peale sorteerimist samasse järjekorda. Stabiilsus võib olla oluliseks omaduseks, kui järjestatakse keerulisemat andmekomplekti. Näiteks juhul kui on vaja sorteerida mitme tunnuse järgi. Inimesi perekonnanime (esmane tunnus) ja eesnime (teine tunnus) järgi sorteerides tuleb esmalt sorteerida eesnimede järgi ja seejärel stabiilse meetodiga perekonnanimede järgi. Niimoodi saavutatakse tulemus, kus sama perekonnanimega inimesed on järjestatud omakorda eesnime järgi.

Protseduurne abstraktsioon

Tarkvaraarenduses ei ole alati ette teada, milline algoritm kõige paremini konkreetses olukorras sobib. Nagu me peatselt näeme, on üks sorteerimisalgoritm sobivam kasutada ühtedes tingimustes ja teine teistes tingimustes. Seega tuleks programme ülesehitada selliselt, et vajaduse korral algoritmi lihtsalt muuta saab.

Suuremate projektide kallal ei tööta ainult üks inimene, vaid töö tuleb jagada mitme töötaja vahel. Nii võib näiteks Mari kirjutada valmis funktsiooni andmete sorteerimiseks ja Jaan seda funktsiooni oma programmis kasutada. Et Jaan ei peaks süvenema Mari kirjutatud funktsiooni sisusse, tuleb alluda reeglitele, mida seab **protseduurse abstraktsiooni** ideoloogia. Nimelt peab selle kohaselt iga protseduuri / funktsiooni jaoks andma sellise kirjelduse, et funktsiooni võiks julgelt ilma koodi süvenemata kasutada. Teisalt – kui tekib vajadus andmete iseloomu tõttu võtta kasutusele teine sorteerimisalgoritm, siis peaks piisama sellest, et Mari funktsiooni ümber kirjutab ja Jaan sellest teadmagi ei pea.

Järgnevas materjalis on enamus sorteerimisfunktsioone esitatud nii, et funktsiooni:

- a) eesmärk,
- b) sisend ja väljund,
- c) eeltingimused (precondiotions),
- d) järeltingimused (postconditions),
- e) väljakutsumine,

on ühesugused, kuid algoritm, mille alusel ta oma eesmärgi saavutab (andmed ära sorteerib) on erinev. Funktsioonide nimed on küll järgnevas materjalis erinevad, kuid need võiks ka ära muuta.

Lepime kokku järgnevas:

Eesmärk: sisendandmete massiiv sorteerida mittekahanevasse järjekorda

Sisend ja väljund: sisendandmeteks on

a) täisarvude massiiv a

Inga Petuhhov TLÜ 2/13

b) täisarvud alg ja lopp, mis näitavad, mitmendast massiivi elemendist sorteerimine algab ja mitmenda elemendiga lõpeb

väljundandmeteks on sama massiiv sorteeritult

eeltingimused - massiiv on arvudega täidetud ja on teada ning korrektselt esitatud alg ja lopp

järeltingimused - massiivi muudetakse ja temast saab mittekahanevasse järjekorda sorteeritud massiiv

Kahes muutujas väärtuste vahetamiseks (mida on sorteerimisel vaja teha mitmel pool), kasutatakse koodi lühendamise eesmärgil Pythonile iseloomulikku võimalust:

```
a,b = b,a
```

Valiksorteerimine e valikmeetod (selection sort)

Klass: vahetussorteerimised

Põhiidee: Mööda massiivi liigutakse vasakult paremale. Vasakule tekib sorteeritud massiiv. Igal sammul otsitakse järgmise elemendi kohale õiget (suuruselt sobivat) arvu, mis ühtlasi on sorteerimata osa kõige väiksem arv.

- 1. Alusta massiivi esimesest elemendist.
- 2. Sorteerimata osast otsi väikseim arv.
- 3. Vaheta leitud arv sorteeritud osale järgneva arvuga. Nii satub nimetatud arv massiivis oma õigele kohale.
- 4. Vii järg massiivis edasi ja mine sammule 2.
- 5. Tegevust korda seni, kuni kogu massiiv on läbitud (ja sorteeritud).

Keerukus: O(n²) nii keskmisel kui ka halvimal juhul

Eriomadused: Selle algoritmi erisuseks on, et tulemused võib juba sorteerimise käigus väljastada, sest massiivi algusse hakkavad kogunema sorteeritud arvud. Sorteerimise kiirus ei sõltu sellest, kui sorteeritud massiiv eelnevalt on. Tööd tehakse sorteerimiseks alati ühepalju.

```
def valiksorteerimine(a,alg,lopp):
    "Funktsioon sorteerib listi valiksorteerimise meetodil"
    for i in range(alg,lopp+1):
        mini = i
        "Leitakse sorteerimata osast vähim suurus"
        for j in range(i, lopp+1):
            if a[j] < a[mini]:
                 mini = j
        "Vahetatakse vähim oma õigele kohale"
        a[i],a[mini] = a[mini], a[i]
        return a</pre>
```

Mullsorteerimine e mullimeetod (bubblesort)

Klass: vahetussorteerimised

Põhiidee: Omavahel võrreldakse paarikaupa arve. Kui kaks kõrvuti asetsevat arvu on vales järjekorras, tuleb nad vahetada. Peale ühekordset massiivi läbimist ei pruugi veel kõik arvud paigas olla, kuid suurim on kindlasti sattunud viimasele positsioonile, st oma õigele kohale.

- 1. Alustades massiivi algusest võrdle omavahel kahte järjestikust arvu.
- 2. Kui suurem arv on eespool, siis vaheta arvud omavahel ja jäta meelde, et vahetus toimus.

Inga Petuhhov TLÜ 3/13

- 3. Korda tegevust seni, kuni oled jõudnud massiivi viimaste arvuni.
- 4. Kui vahetusi tehti, siis korda kõike algusest peale, vastasel juhul on massiiv sorteeritud.

Keerukus: Halvimal juhul $O(n^2)$, parimal juhul O(n), keskmiseks loetakse siiski $O(n^2)$.

Eriomadused: Kui massiiv on juba peaaegu sorteeritud, siis võib saavutada ajalist efekti. Muidu on aga meetod aeglane tänu suurele vahetuste arvule. Massiiv saab sorteerituks lõpust alates, kuid sama algoritmi on võimalik kohandada selliselt, et kõigepealt saavad paika väiksemad kirjed, st massiivi algus.

```
def mullsorteerimine(a,alg,lopp):
    "Funktsioon sorteerib listi mullimeetodil"
    jarg = alg
    veel = True
    while jarg<lopp and veel:
        "Muutuja 'veel' näitab, kas vahtusi toimus."
        veel = False
        for pos in range(alg,lopp-jarg):
            "Kui kõrvuti olevad arvud on valepidi, siis nad vahetatakse"
        if a[pos] > a[pos+1]:
            veel = True
            a[pos+1],a[pos] = a[pos],a[pos+1]
        jarg += 1
    return a
```

Mullimeetodi variatsiooniks on *shakersort*, kus kirjete võrdlemist alustatakse vaheldumisi massiivi algusest ja lõpust. See meetod osutub kasulikuks, kui väike element on massiivi lõpus. Miks?

Lisamissorteerimine e pistemeetod (insertion sort)

Klass: vahelepanekuga sorteerimised

Põhiidee: Sorteeritud on vasakpoolne massiivi osa, kuid mitte lõplikult. Paremalt poolt võetakse järgmine element ja sobitatakse ta sorteeritud poolele õigesse kohta vahele. Algoritm meenutab käes hoitavate kaartide järjestamist.

Alustades 2. elemendist:

- 1. Võta massiivist arv ja pane ta eraldi muutujasse.
- 2. Võrdle seda arvu järjest massiivis temast vasakule poole jäävate arvudega ja tõsta koha võrra paremale kõik arvud, mis on sellest arvust suuremad.
- 3. Kui jõuad sama suure või väiksema arvuni, oled leidnud talle leidnud õige koha. Et arve on eest ära paremale nihutatud, on õige massiivi lahter vaba. Pane arv omale kohale.
- 4. Korda tegevust kõigi arvudega kuni massiivi lõpuni.

Keerukus: Halvimal ja keskmisel juhul $O(n^2)$ ning parimal juhul O(n) (sõltuvalt massiivi eelnevast sorteeritusest).

Eriomadused: Kõige rohkem tööd põhjustab selles algoritmis arvu vahelepanekuks ruumi tegemine, ehk ükshaaval arvude paremale poole nihutamine. Algoritmist saab iaga idee selle jaoks, kui on vaja juba sorteeritud massiivi uus element lisada nii, et ta oma õigele kohale satuks. Meetodit sobib kasutada ahela sorteerimiseks, kus ahela elemente ei ole vaja füüsiliselt ümber paigutada. On vaid vaja elementide suuruseid võrreldes leida sobiv koht. Ja loomulikult saab sel viisil lisada elemente juba sorteeritud ahelasse.

```
def lisamissorteerimine(a, alg, lopp):
```

Inga Petuhhov TLÜ 4/13

```
"Funktsioon sorteerib listi pistemeetodil"
for uuspos in range(alg+1,lopp+1):
    uusarv = a[uuspos]
    pos = uuspos
"Kirjeid nihutatakse vasakule seni kuni on leitud sobiv koht uuele
kirjele."
    while a[pos-1] > uusarv and pos > 0:
        a[pos] = a[pos-1]
        pos -= 1
    a[pos] = uusarv
    return a
```

Shell'i sorteerimine e väheneva sammu meetod (Shell sort)

Klass: vahetussorteerimine

Põhiidee: Autor on Donald L. Shell (1959). Algoritm sarnaneb mulli meetodile, kuid uuendusena käiakse massiiv läbi algul suuremate ja hiljem väiksemate sammudega, et kirjed rutem oma õigele paigale vahetada. Probleemiks on sammu valik. Rusikareegliks võib olla pidev sammu poolitamine.

- 1. Järgnevas algoritmis võetakse esimeseks sammuks n // 2.
- 2. Võrreldakse omavahel arve, mis on üksteisest välja arvutatud sammu kaugusel (esimesel korral on selleks pool massiivi pikkusest). Kui eespool olev arv on tagapool olevast suurem, siis arvud vahetatakse.
- 3. Tegevust korratakse seni, kuni ühtegi vahetust enam antud sammuga teha pole vaja.
- 4. Seejärel vähendatakse sammu (näiteks jagatakse kahega) ja kõik kordub.
- 5. Sammu vähendamist jätkatakse, kuni see saab üheks millega ollakse jõudnud mullimeetodini.

Keerukus: on tõestatud, et sobiva sammuvaliku korral on $O(n^{3/2})$, kuid halva sammuvaliku puhul võib olla ka $O(n^2)$.

Eriomadused: Hoolimata mitte just eeskujulikust ajalisest keerukusest, on tegelik tööaeg suvaliste andmete korral tunduvalt parem kõigist eelnevatest. Kindlasti töötab ta kiiremini juba osaliselt sorteeritud massiivil.

Sorteerimine mestimisega e ühildusmeetod (merge sort)

Klass: mestimissorteerimised

Inga Petuhhov TLÜ 5/13

Põhiidee: (Autor John von Neumann 1945) Algoritm koosneb kahest osast – massiivi jagamisest ning kahe osa ühendamisest. Sisuliselt viimase läbi kirjed lõpuks järjestatuks saavadki. Algoritm oli esialgu kavandatud välismälu (kettal vms failides olevate andmete) sorteerimiseks.

- 1. Jaga algandmed kaheks enam vähem võrdseks osaks.
- 2. Sorteeri kumbki osa eraldi.
- 3. Kombineeri mõlemad osad kokku üheks sorteeritud massiiviks. Vt allpool funktsiooni merge ().

Olemuselt on see algortim rekursiivne ja toetub otseselt lähenemisele "Jaga ja valitse" (divide et impera). Kõigepealt minnakse rekursiivse funktsiooniiväljakutsega nö sügavusse ja rekursiooni lõpetamise tingimuseks on, et sorteeritava jada pikkus on üks element. Rekursioonist väljudes ühendatakse massiivi osi järjest omavahel, saades nii üha pikemad sorteeritud lõigud.

Keerukus: O(n log₂n) nii halvimal kui ka keskmisel juhul.

Eriomadused: Vajab täiendavat mälu ajutiste massiivide tegemiseks (reaalselt sama palju kui on sorteeritavaid andmeid), sest kahe sorteeritud osa ühemdamisel kirjutatakse arvud esialgu ringi teise massiivi. Meetodi idee sobib ka kasutamiseks, kui on juba olemas kaks sorteeritud massiivi ning need on vaja ühendada üheks.

Meetodi idee sobib ka väga edukalt välismälu sorteerimiseks, kui andmed on failis ja neid on liiga palju, et kõiki korraga mällu lugeda. Sel juhul võetakse sorteeritavalt failist sobiliku pi8kkusega jupid, sorteeritakse need suvalise meetodiga ja kirjutatakse tulemused ajutistesse failidesse. Sellele järgneb sorteeritud ajutiste failide paari kaupa ühendamine suuremateks failideks, kuni lõpuks kõik üheks failiks ühendatud saab. Sellisel põhimõttel töötab näiteks Unixi sort-käsk.

Järgnev rekursiivne funktsioon moodustab tervest massiivist jupid, mida hiljem sorteeritult ühendama hakatakse. Jupid tekivad tänu rekursioonile, millele antakse ette juppide piirid ideksitena. Viimasena väljakutsutav funktsioon merge() peab etteantud massiivi piirkonna sorteeritult kokku ühendama.

```
def mestimisega_sorteerimine(a, alg, lopp):
    if alg < lopp:
        kesk = (alg + lopp) // 2
        mestimisega_sorteerimine(a, alg, kesk)
        mestimisega_sorteerimine(a, kesk + 1, lopp)
        merge(a, alg, kesk, lopp)
    return a</pre>
```

Funktsiooni merge () algoritm on järgmine:

- 1. Võrdle kummagi poole esimest elementi omavahel.
- 2. Väiksem element pane ajutisse massiivi.
- 3. Tõsta selles pooles indeksit edasi
- 4. Korda esimest kolme sammu seni kuni üks massiivipooltest tühjaks saab.
- 5. Kirjuta teises massiivi pooles viimased arvud (need, mis veel ringi kirjutamata) ringi ajutisse massiivi.
- 6. Kirjuta ajutine massiiv ringi originaalmassiivi.

```
def merge(a, alg, kesk, lopp):
    "Ühendab massiivist a kaks osa a[alg] kuni a[kesk]
    ja a[kesk+1] kuni a[lopp] omavahel"
    jvasak = alg
```

Inga Petuhhov TLÜ 6/13

```
jparem = kesk + 1
temp = []
"Ühendame, kuni üks massiivipooltest otsa saab"
while (jvasak<=kesk) and (jparem<=lopp):</pre>
    if a[jvasak] < a[jparem]:</pre>
        temp.append(a[jvasak])
        jvasak += 1
    else:
        temp.append(a[jparem])
        jparem += 1
"Üks pooltest sai tühjaks, kirjutame teisest ka arvu lõpuni ringi."
while jvasak <= kesk:</pre>
    temp.append(a[jvasak])
    jvasak += 1
while jparem <= lopp:
    temp.append(a[jparem])
    jparem += 1
for i in range(alg, lopp+1):
    a[i] = temp[i-alg]
```

Kiirsorteerimine e kiirmeetod (quicksort)

Klass: vahetussorteerimised

Põhiidee: (Cambridge'i Ülikooli professor Anthony Richard Hoare 1962 a):

- 1. Võetakse mingi poolitava väärtusega element.
- 2. Arvud paigutatakse massiivis ringi selliselt, et kõik, mis on poolitavast arvust väiksemad, asuksid massiivis temast vasakul ja kõik suuremad paremal (vasak ja parem pool ei ole sorteeritud). Poolitava arvu koht on nende kahe poole vahel.
- 3. Tegevust korratakse rekursiivselt iga osaga, kuni osa suuruseks saab 1 element. Siis peaks ka massiiv sorteeritud olema.

Poolitusalgoritmi seletus:

- 1. Määratakse poolitavaks väärtuseks PIV massiivi[osa] esimene arv (vastavalt kokkuleppele võib määrata ka mõne muu arvu või valida selleks lausa juhuslikult arv vastavast massiivi osast).
- 2. Liikudes massiivi osa algusest lõpu poole (I) leitakse esimene arv, mis on poolitajast suurem
- 3. Liikudes massiivi osa lõpust alguse poole (J) leitakse esimene arv, mis on poolitajast väiksem.
- 4. Leitud kaks ar vu vahetatakse omavahel, ja nad satuvad nö õigele massiivi poolele. Kirjeldatud tegevust korratakse seni kuni järjehoidjad I ja J teineteisest mööduvad.
- 5. Poolitav väärtus PIV viiakse oma õigele kohale suurema ja väiksema poole vahel. Selleks vahetatakse massiivi[osa] esimene ja J-s element omavahel.

Algoritm on rekursiivne. St esialgu jaotatakse elemendid ümber poolitava väärtuse ja seejärel funktsiooni rekursiivse väljakutsega minnakse kummaski massiivi osas sügavusse. Rekursiooni lõpetamise tingimuseks on sorteeritava jada pikkus üks element.

Keerukus: O(n log₂n) keskmisel juhul ja O(n²) halvimal juhul

Eriomadused: võtab kõige rohkem aega, kui massiiv on juba peaaegu sorteeritud (aga see sõltub siiski rohkem poolitusväärtuse valikust), rekursioon võib minna liiga sügavaks ja tekitada täitmisaegse vea. Halvim juhus on see, kui poolitav element valitakse "ühte serva" ja tekivad pooled on suurustega 1 ja n−1.

Inga Petuhhov TLÜ 7/13

```
def kiir sorteerimine(a, left, right):
    "Sorteerib massiivi a kiirsorteerimise meetodil."
    if left < right:
        piv = a[left]
        i = left + 1
        j = right
        while i < j:
            while a[i] <= piv and i < right:
                i += 1
            while a[j] > piv and j > left:
                j -= 1
            if i < j:
                a[i], a[j] = a[j], a[i]
        if piv > a[j]:
            a[left] = a[j]
            a[j] = piv
    # Järgnevate väljakutsetega antakse funktsioonile ette
    # massiivi esimene pool ja teine pool
        kiir sorteerimine(a,left,j-1)
        kiir sorteerimine(a,i,right)
```

Meetodist on mitmeid modifikatsioone, suurem osa nendest tegeleb probleemiga, kuidas valida sobivat poolitajat. Hoare originaalvariandis valitakse poolitavaks väärtuseks massiiviosa 1. element, mis sobib piisavalt hästi siis, kui massiiv on segamini. Kui massiiv on osaliselt järjestatud, pole see taktika parim.

Sorteerimine kuhjaga e kuhjameetod (heapsort)

Meetod kasutab erilist andmestruktuuri – kahendkuhja.

Kahendkuhi (binary heap)

Kahendkuhi on kahendpuu kujuline andmestruktuur, mida tavaliselt realiseeritakse massiivi abil. Kuhjas olevad elemendid on teatud reegli alusel järjestatud.

Kahendkuhjas hoitavad elemendid paiknevad vastavalt **kuhja omadusele** (heap property): vanema väärtus ei ole väiksem järglaste väärtustest. See omadus kehtib iga kahendpuud moodustava alampuu kohta. Seega puu juureks on kõige suurem element.

Massiivi abil realiseeritud kahendpuu iga tipp moodustab massiivi ühe elemendi. Puu juureks on element indeksiga 1, juure lapsed on 2. ja 3. element, nende lapsed omakorda 4., 5., 6. ja 7. element ine. Kahendpuu peab olema peaaegu täielik, st puu kõik tasemed, väljaarvatud viimane, on täidetud.

Iga tipu I jaoks saab arvutada nii vanema kui laste indeksid (kui indekseerimine algab1-ga):

- vanema indeks (Parent(i)) on i // 2
- vasaku järglase indeks (Left(i)) on 2 * i
- parema järglase indeks (Right(i)) on 2 * i + 1

Kui aga indekseerimine algab 0-ga, on arvutused järgmised:

- vanema indeks (Parent(i)) on (i + 1) // 2 1
- vasaku järglase indeks (Left(i)) on 2 * i + 1
- parema järglase indeks (Right(i)) on 2 * i + 2

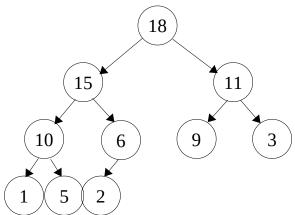
Eristatakse niinimetatud *max-heap'i* ja *min-heap'i*. Eelnevalt kirjeldatud kahendkuhja omadus on *max-heap'i* omadus. Kui vanem on alati väiksem oma lastest, siis saame *min-heap'i*. Viimasel juhul on kuhja esimene element (puu juur) kõige väiksem.

Inga Petuhhov TLÜ 8/13

Järgneva sorteerimise jaoks võib lisaks koostada funktsioonid Parent (i), Left (i) ja Right (i).

Kahendkuhja omadus avaldub järgmise võrdusena:

A[Parent(I)] >= A[I]



Joonis 1: Kahendkuhi puuna

1	2	3	4	5	6	7	8	9	10
18	15	11	10	6	9	3	1	5	2

Joonis 2 Kahendkuhi massiivina, kus indekseerimine algab 1-st

Kahendkuhja kasutatakse ka prioriteetidega järjekorra realiseerimiseks. Sel juhul paikneb kõrgeima prioriteediga element kuhja tipus (massiivi 1. lahtris) ja peale tema eemaldamist tuleb kuhi ringi ehitada (vt järgnevas lõigus funktsiooni Heapify). Sõltuvalt sellest, kuid as prioriteedid on määratud, saab kasutada kas *min-heap'i* või *max-heap'i*.

Sorteerimine

Klass: puusorteerimine

Põhiidee: Sorteerimine toimub kahes etapis:

- 1. Arvudemassiivist moodustatakse väärtuste ümberpaigutamise teel kahendkuhi.
- 2. Kahendkuhi sorteeritakse vastava algoritmiga.

Kõige keerulisema osa kogu tööst moodustab kahendkuhja tipu paigutamine õigele kohale oma järglaste suhtes. Funktsiooni (heapify()) on vaja nii kahendkuhja moodustamisel kui ka hilisemal sorteerimisel. Algoritm toimib järgmiselt: elemendi väärtust võrreldakse tema laste väärtustega. Kui vähemalt üks laps on suurem, valitakse lastest suurem ja vahetatakse elemendiga. Sama tegevust korratakse tipuga, mis eelmises operatsioonis kõige suuremaga ära vahetati. Kontroll võib vajaduse korral kuni leheni välja minna. Allpoolt toodud funktsioon on rekursiivne, kuid hakkama saab edukalt ka tavalise tsükliga

Kui massiivist on sel viisil kuhi moodustatud, järgneb sorteerimine:

- 1. Juurelement eemaldatakse kuhjast, vahetades ta viimase elemendiga ning kuhja suurust vähendatakse ühe võrra.
- 2. Kuhja omadus taastatakse kuhja juureks saanud väikese elemendi vahetamisel suurema lapse vastu, kasutades heapify() funktsiooni.

Inga Petuhhov TLÜ 9/13

Keerukus: Kahendpuu maksimaalne kõrgus on log n. Seega ka kõigi kuhjas tehtavate operatsioonide ajaline keerukus ei saa ületada log n. Siit tulenevalt saame kuhja abil sorteerimise keerukuseks 0 (n log n). Selle keerukusega on nii kuhja moodustamine kui ka edasine sorteerimine

Eriomadused: Sorteeritud massiiv hakkab tekkima massiivi lõpust.

```
def heapify(a, n, i):
    "Funktsioon elemendi viimiseks kuhjas ülalt poolt õigele kohale."
    "a - kuhi"
    "n - kuhja suurus"
    "i - õigesse kohta viidava elemendi indeks"
    left = 2 * i + 1
    right = 2 * i + 2
    maxi = i
    if left < n and a[left] > a[maxi]:
        maxi = left
    if right < n and a[right] > a[maxi]:
        maxi = right
    if maxi != i:
        a[maxi], a[i] = a[i], a[maxi]
        a = heapify(a, n, maxi)
    return a
```

Kuhja ehitamise oluliseks osaks on eelnev funktsioon. Tööd alustatakse kuhja eelviimasest tasemest, sest lehtedel pole järglasi. St läbi on vaja vaadata ca pooled massiivi elemendid. Kõigile tippudele sealt ülespoole rakendatakse heapify () -funktsiooni:

```
def buildHeap(a):
    "Funktsioon kuhja moodustamiseks."
    "Sisend: on massiiv a, väljund: sama massiiv, mis vastab kuhja reeglile."
    for i in range(len(a)/2, -1, -1):
       a = heapify(a, len(a), i);
    return a
Sorteerimine:
def HeapSort(a, n):
     "Sisend: a - massiiv, mis vastab kuhja tingimustele"
     "Väljund: a - massiiv sorteeritud kujul"
    a = buildHeap(a)
    kuhjasuurus = n
    for i in range (n-1, 0, -1):
         a[0], a[i] = a[i], a[0]
         kuhjasuurus -= 1
         a = heapify(a,kuhjasuurus,0)
```

Kõiki eespool kirjeldatud meetodeid kutsutakse ka **võrdlemistega sorteerimisteks** (comparison sort), sest sorteerimise aluseks on kirjete kahekaupa võrdlemine. On tõestatud, et sellist tüüpi algoritmide korral pole lootustki saada paremat ajalist keerukust kui 0 (n log n). Seega sorteerimine kuhjaga ja sorteerimine mestimisega, mis omavad ajalist keerukust 0 (n log n) halvimal juhul ning kiirsorteerimine, mis omab sama keerukust keskmisel juhul on need meetodid, mida võimaluse korral kasutada tasub.

Sorteerimine loendamisega e loendamismeetod (counting sort)

Klass: loendussorteerimised

return a

Inga Petuhhov TLÜ 10/13

Põhiidee: (1954 a. H. H. Seward). Idee seisneb selles, et iga arvu jaoks loendatakse, mitu arvu on temast väiksemad ja selle alusel saab leida arvule õige koha sorteeritud massiivis. Kui arvust on M väiksemat arvu, siis arv ise tuleb paigutada massiivis M+1 kohale. Kui massiivis on sama suurusega arve, tuleb algoritmi veidi muuta, et arve üle ei kirjutataks – vähendada talle eelnevate arvude hulka ühe võrra, kui üks neist arvudest on oma kohale kirjutatud.

Algoritmi sammud on järgmised:

- 1. Kõigi massiivis olevate väärtuste loendamine loendurmassiivi abil.
- 2. Igale arvule eelnevate arvude koguse leidmine loendurmassiivi abil (massiivis arvude esinemiste liitmine).
- 3. Arvude paigutamine uude massiivi vastavalt leitud kohale ja ümberkirjutamine algsesse massiivi.

Keerukus: Kirjeldatud algoritm on ajaliselt keerukuselt lineaarne O(N).

Eriomadused Algoritmi kasutusvaldkond on piiratud - tema abil sobib sorteerida positiivseid täisarve, mis on kindlates piirides. Tundub, et võiks ka sorteerida negatiivseid arve. Aga sel juhul peab ette teada olema, millisesse vahemikku arvud jäävad, et sellest lähtudes loendurmassiivi õigesti kasutada.

Arvestama peab kasutada oleva mälumahuga. Algoritm nõuab täiendavalt mälu kahe massiivi jagu: loendamiseks (nii mitu elementi, kui loendatavas massiivis võib olla erinevaid arve) ja uue sorteeritud massiivi moodustamiseks. Seega hea ajalise keerukusega kaasneb suur mäluline keerukus.

def countsort(a,algus,lopp):

```
"Sorteerib massiivi a loendamissorteerimise meetodil. "
"Eelduseks on, massiivis on täisarvud vahemikus 0 .. 1000"
maksarv = 1000
loend = (maksarv+1) * [0]
b = (lopp+1-algus)*[0]
"Loendame erinevate arvude esinemise jadas."
for i in range(algus,lopp+1):
    loend[a[i]] = loend[a[i]]+1
"Mitu arvu antud arvule eelneb?"
for i in range(1, maksarv+1):
    loend[i] = loend[i-1] + loend[i]
"Kirjutame arvu massiivi b õigele kohale."
for i in range(lopp, algus-1, -1):
    b[loend[a[i]]-1] = a[i]
    loend[a[i]] = loend[a[i]]-1
for i in range(algus,lopp+1):
    a[i] = b[i]
```

Loendamisega sorteerimise algoritm on stabiilne.

Sorteerimine järkudega e positsioonimeetod (radix sort)

Põhiidee: Meetod pärineb ajast, kui andmete hoidmiseks kasutati perfokaarte ja neid oli vaja mingil alusel järjestada. Järjestamistööd tegi edukalt spetsiaalne masin.

Kõigepealt järjestati kaardid ringi kõige "noorema" veeru järgi - vastavalt tulbas perforeeritud väärtusele 0 .. 9 moodustati 10 erinevat hunnikut ja seejärel tõsteti hunnikud uuesti üksteise peale üheks hunnikuks. Järgmisel sammul tehti sama eelviimase tulbaga jne kuni kõigi tulpade järgi oli kogu hunnik korra ringi tõstetud. Selliselt saab kogu pakk sorteeritud. Oluline on, et kaartide

Inga Petuhhov TLÜ 11/13

järjekorda peale hunnikutesse tõstmist hunniku sees muuta ei tohi. Kirjalikul kujul on vastavalt D. Knuthile see algoritm selgitatud 1929. a. sorteerimismasina juhendis, autoriks L. J. Comrie.

Asendades perfokaardid arvudega saame sorteerida sama kohtade arvuga arve (3-kohalisi, 4-kohalisi jne).

Algoritm:

- 1. Arvud jagatakse 10sse ossa vastavalt üheliste järgus olevale väärtusele
- 2. Osad tõstetakse järjekorras kokku.
- 3. Arvud jaotatakse 10 ossa kümneliste väärtuse järgi
- 4. Osad tõstetakse järjekorras kokku.
- 5. Sama kordub sajaliste, tuhandeliste jne kohta.

Kui arvud on üheliste järgi ära jagatud, peavad kümneliste järgi tekkivates hunnikutes ühelised omavahelise järjekorra säilitama.

Algseis	Ühelised	Kümnelised	Sajalised
123	880	123	123
880	282	147	147
276	562	562	276
147	123	276	282
282	276	282	562
562	147	880	880

Joonis 3 Sammud positsioonimeetodi puhul.

Keerukus: Algoritmi ajaline keerukus sõltub sellest, millist algoritmi ühe järgu järgi järjestamiseks kasutada. Kindlasti pole sobivad $O(N^2)$ algoritmid, kuid sobida võib hoopis eelmisena kirjeldatud loendamisega sorteerimine, kuivõrd sorteeritavate erinevate arvude hulk on piiratud (0...9). Arvudest järkude kättesaamiseks tuleb uurida konkreetse keele võimalusi.

Ajalise keerukuse osas on saavutatav O (N), mälu on aga vaja veel teise massiivi jaoks vahetulemuste hoidmiseks.

Eriomadused: Sorteerida saab vaid teatud tingimustele vastavaid suuruseid, näiteks arvud peavad olema ühesuguse kohtade arvuga (või tuleb arvu ette nullid lisada). Kirjeldatud algoritmil on ka teistlaadseid kasutusalasid: arvude asemel võib sorteerida ühesuguse pikkusega sõnu. Hästi sobib ta kasutamiseks kuupäevade järjestamisel (vms juhul, kui sorteerida on vaja mitme erineva väärtuse järgi): kõigepealt sorteeritakse päeva järgi (loendamisega sorteerimise põhimõttel), siis kuu järgi ja lõpuks aasta järgi. Meetod on lihtsam ja kiirem kuupäevade paarikaupa võrdlemisest ja vahetamisest. Ehkki kuupäevi enamasti hoitakse tavaliste arvudena ja siis langeb osade kaupa järjestamise vajadus ära.

Sorteerimine kastiga, pangega või ämbriga e kimbumeetod (bucket sort)

Põhiidee: vahemik, kuhu sorteeritavad väärtused kuuluvad, jagatakse n võrdseks osaks, igale osale eraldatakse oma **kast** (*bucket*). Iga arv pannakse temale sobivasse vahemikku ehk kasti. Kui arvud on ühtlaselt jaotunud võib oodata, et igasse kasti satub enam-vähem ühepalju arve (<u>vähe arve</u>).

Kõigi kastide sisud sorteeritakse eraldi. Kuna väärtusi on igas kastis vähe, peaks sobima selleks üsna suvaline algoritm, ka ruutkeerukusega.

Inga Petuhhov TLÜ 12/13

Seejärel võetakse kastidest kasvamise järjekorras arvud ja saadaksegi sorteeritud massiiv.

Keerukus: algoritm töötab keskmiselt lineaarse ajaga. Igasugune sorteeritav materjal siiski ei anna lubatud keskmist lineaarset tulemust, vaid sisendiks peaksid olema sõltumatud juhuslikud arvud vahemikus [0,1). Arvud peavad olema ühtlaselt jaotunud, ms tagab kastide ühesuguse täituvuse.

Realiseerimiseks on vaja veel teist massiivi, mille suurus on sama sorteeritava massiiviga ja mida kasutatakse kastide haldamiseks. Iga kasti sisu jaoks moodustatakse ahel.

Meetod sobib kasutamiseks ka tavaelus. Näiteks saab kasutatud sellist meetodit sisseastujate tööde tähestiku järgi järjekorda panemiseks – kõigepealt jaotus nime esitähe järgi hunnikutesse, seejärel iga hunniks sorteerida ja lõpuks hunnikud ühendada. Lubab kenasti kasutada ka paralleeltöötlust.

Inga Petuhhov TLÜ 13/13