Sorteerimine

Kui töödeldavad andmed on järjestatud, on võimalik kiiremini ja mugavamalt midagi üles leida. 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.

Tavaliselt järjestatakse kirjed võtmeväärtuse kasvavas järjekorras (väiksem kõige ees), kuid võib ka järjestada kahanevas järjekorras (suurem kõige ees).
Tavaliselt tuukse asorteerimisalgoritmide 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 (näit. kirjed Pascali mõttes) 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 ilma kirjeteta, kuid vahet pole, see võiks olla ka massiiv kirjetest vms.

Sorteerimist võib 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 kasvab see aeg, kui kirjete arv näiteks kahekordistub? Teisisõnu hinnatakse algortimi ajalist keerukust.
2) **Mälu** - kas saab sorteerida nn. 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.
4) **Programmeerimiskeel** - pole enam eriti aktuaalne, aga osa algoritme nõuab rekursiooni kasutamist ja mõned vanemad keeled ei võimalda seda meetodit kasutada.

Sorteerimismeetodid jagatakse **klassideks**:
1) vahelepanekuga sorteerimised;
2) vahetussorteerimised;
3) loendussorteerimised;
4) mestimissorteerimised;
5) valik- ja puusorteerimised.

> Sorteerimisalgoritm on **stabiilne** *(stable)*, kui kaks sama võtmeväärtusega kirjet jäävad ka peale sorteerimist samasse järjekorda. Stabiilsus võib olla oluliseks omaduseks, kui järjestatakse keerulisemat andmekomplekti. Sorteerimisalgoritmide puhul pööratakse stabiilsusele ka tähelepanu.

Protseduurne abstraktsioon

Tarkvaraarenduses ei ole alati ette teada, millised algoritmid kuhu kõige paremini sobivad. Nagu me peatselt näeme, on üks sorteerimisalgoritm sobivam kasutada ühtedes tingimustes ja teine teistes tingimustes. Seega tuleks programme ülesehitada selliselt, et vajaduse korral algoritmi kerge muuta oleks.

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

Enamus sorteerimisprotseduure on esitatud selliselt, et protseduuri:
a) eesmärk,
b) sisend ja väljund,
c) eeltingimused *(precondiotions)*
d) järeltingimused *(postconditions)*
e) väljakutsumine,
oleksid ühesugused, kuid algoritm, mille alusel ta oma eesmärgi saavutab (andmed ära sorteerib) on erinev.

Lepime kokku järgnevas:
a) **Eesmärk**: sisendandmete massiiv sorteerida mittekahanevasse järjekorda
b) **Sisend ja väljund**
sisendandmeteks on
a) täisarvude massiiv a
b) täisarvud alg ja lopp, mis näitavad, mitmendast massiivi elemendist sorteerimine algab ja mitmenda elemendiga lõpeb
väljundandmeteks on sama massiiv sorteeritult
c) **eeltingimused** - massiiv on arvudega täidetud ja on teada ning korrektselt esitatud alg ja lopp
d) **järeltingimused** - massiivi muudetakse ja temast saab mittekahanevasse järjekorda sorteeritud massiiv

Lisaks kasutatakse alamprogrammides alamprogrammi vaheta, mille ülesandeks on kahs muutujas väärtuste ringivahetamine.

```
def vaheta(a,b):
    "protseduur vahetab väärtused kahes parameetris"
    temp = a
    a = b
    b = temp
    return a,b
```

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: $O(n^2)$ nii keskmisel kui ka halvimal juhul

Eripärad: 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.

Protseduur:
```
def valiksort(a,alg,lopp):
    "Protseduur 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] = vaheta(a[i],a[mini])
```

Mullsorteerimine e mullimeetod (bubblesort)

Klass: vahetussorteerimised

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

1. Alustades massiivi algusest võrdle omavahel kahte järjestikust kirjet.
2. Kui suurem kirje on eespool, siis vaheta kirjed omavahel ja jäta meelde, et vahetus toimus.
3. Korda tegevust seni, kuni oled jõudnud massiivi viimaste kirjeteni.
4. Kui vahetusi oli toimunud, siis korda kõike algusest peale, vastasel juhul on massiiv sorteeritud.

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

Eripärad: 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 algortimi on võimalik kohandada selliselt, et kõigepealt saavad paika väiksemad kirjed, st massiivi algus.

Protseduur:

```
def mullsort(a,alg,lopp):
    "protseduur 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 vahetatake"
            if a[pos] > a[pos+1]:
                veel = True
                a[pos],a[pos+1] = vaheta(a[pos],a[pos+1])
        jarg += 1
```

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. Esimeseks arvuks tuleb massiivi lisada väga väike arv, millest väiksemat sorteeritavate kirjete hulgas ei leidu. Seda on vaja, et töö käigus mitte üle minna massiivi algusest.
Alustades 2. elemendist:

1. Võta kirje.
2. Leia talle sobiv kohta temast vasakul olevate kirjete hulgas (selleks tuleb teda võrrelda vasakule poole jäävate kirjetega, kuni õigekoht on leitud).
3. Nihuta kirjed eest ära, et paigutada vaatlusalune kirje oma kohale.
4. Korda tegevust kõigi kirjetega kuni massiivi lõpuni.

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

Eripärad: Massiivi sorteerimisel tekitab rohkem raskusi vahelepanekuks ruumi tegemine - kui arv lisatakse rea algusesse, tuleb nihutada kõiki ülejäänud kirjeid. Sobib paremini juhul, kui üksik uus kirje on vaja õigesse kohta lisada. Või dünaamilise nimistu sorteerimiseks, kus kirjeid ei ole vaja füüsiliselt ümber paigutada.

Protseduur:
```
def lisamissort(a,alg,lopp):
    "protseduur sorteerib listi lisamismeetodil"
    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
```

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

Klass: vahetussorteerimine

Põhiidee: Autor on Donald L. Shell (1959). Idee 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 poolitamine - esialgne samm on `n div 2` jne. On leitud, et sammud ei tohiks olla teineteise kordsed, või soovitatakse samme, mis on 1 võrra väiksemad 2 astmetest (31, 15, 7, 3, 1).

1. Järgnevas algoritmis võetakse sammuks `n div 2`.
2. Võrreldakse omavahel kirjeid, mis on üksteisest sammu kaugusel.
3. Kui eespool olev kirje on tagapool olevast suurem, siis kirjed vahetatakse.
4. Tegevust korratakse seni, kuni ühtegi vahetust enam antud sammuga teha pole vaja.
5. Seejärel vähendatakse sammu (näiteks jagatakse samm 2) ja kõik kordub 2. reast alates uuesti.

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

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

Protseduur:
```
def shell_sorteerimine(a,alg,lopp):
    "protseduur sorteerib listi Shelli meetodil"
    samm = lopp-alg
    while samm > 1:
    "Uue sammu arvutamine (uus samm on pool eelmisest)"
        samm = samm / 2
        veel = True
        while veel:
            print samm
            veel = False
            for i in range(alg,lopp-samm+1):
                k = i + samm
    "Kui kirjed on vaja vahetada, siis nad vahetatakse"
                if a[k] < a[i]:
                    a[k],a[i] = vaheta(a[k],a[i])
                    veel = True
```

Sorteerimine mestimisega e ühildusmeetod (merge sort)

Klass: mestimissorteerimised

Põhiidee: (John von Neumann 1945) Algoritm koosneb kahest osast – massiivi jagamisest ning kahe osa ühendamisest. Sisuliselt viimase läbi kirjed lõpuks järjestatuks saavadki.

1. Jaga algandmed kaheks enam vähem võrdseks osaks.
2. Sorteeri kumbki osa eraldi.

3. Kombineeri mõlemad osad kokku üheks sorteeritud massiiviks.

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

Keerukus: O(n $\log_2$n) nii halvimal kui ka keskmisel juhul.

Eripärad: vajab täiendavat mälu ajutiste massiivide tegemiseks (reaalselt sama palju kui on sorteeritavaid andmeid), rekursioon võib minna liiga sügavaks ja tekitada täitmisaegse vea. Ajutisse massiivi pannakse kirjed mestimise käigus. Meetod võiks sobida ka suurte andmehulkade sorteerimiseks välise sorteerimise käigus – sorteeritud failides olevad kirjed kirjutatakse kokku kolmandasse faili.

Protseduur:

```
def mestimisega_sorteerimine(a, alg, lopp):
    ""Massiiv a sorteeritakse kasvavasse järjekorda mestimismeetodil,
      protseduuris poolitatakse massiiv ning seejärel tehakse sama kummagi
      massiivi poolega.""
    if alg < lopp:
        kesk = (alg + lopp) / 2
        mestimisega_sorteerimine(a, alg, kesk)
        mestimisega_sorteerimine(a, kesk + 1, lopp)
        merge(a, alg, kesk, lopp);
```

Vajalik on lisada algoritm protseduurile `Merge`. Protseduur vajab täiendavalt mälu ajutise massiivi hoidmiseks.

Algoritm:

1. Võrreldakse kummagi poole esimest elementi omavahel.
2. Väiksem element pannakse ajutisse massiivi.
3. Ära pandud element jäetakse järgneva vaatluse alt välja

Eelnevaid tegevusi korratakse seni kuni üks massiivipooltest tühjaks saab.

4. Osast, mis tühjaks ei saanud, kirjutatakse elemendid ajutise massiivi lõppu.
5. Ajutine massiiv kirjutatakse ümber originaalmassiivi.

```
def merge(a, alg, kesk, lopp):
    "Ühendab kaks osa A[alg] kuni A[kesk] ja A[kesk+1] kuni A[lopp] omavahel"
    jvasak = alg
    jparem = kesk + 1
    temp =[]
    "Ühendame, kuni üks massiivipooltest otsa saab"
    while (jvasak<=kesk) and (jparem<=lopp):
        if a[jvasak] < a[jparem]:
            temp.append(a[jvasak])
            jvasak += 1
        else:
            temp.append(a[jparem])
            jparem += 1
    "Kuni on elemente, kopeeri nad tempi"
    while jvasak <= kesk:
        temp.append(a[jvasak])
        jvasak += 1
    "Kuni on elemente, kopeeri nad tempi"
    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. Jada elemendid paigutatakse ringi selliselt, et kõik, mis on poolitavast elemendist väiksemad, asuksid massiivis temast vasakul ja kõik elemendid, mis on suuremad, asuksid massiivis temast paremal (vasak ja parem pool ei ole sorteeritud).
3. Tegevust korratakse rekursiivselt iga osaga, kuni osa suuruseks saab 1 element. Siis peaks ka massiiv sorteeritud olema.

Poolitusalgoritmi seletus:
Kõik elemendid, mis on poolitajast väiksemad viiakse poolitajast vasakule ja kõik elemendid, mis poolitajast suuremad poolitajast paremale. Selleks

1. Määratakse poolitavaks väärtuseks massiivi[osa] esimene element
2. Liikudes algusest lõpu poole leitakse esimene element, mis on poolitajast suurem
3. Liikudes lõpust alguse poole leitakse esimene element, mis on poolitajast väiksem.
4. Leitud 2 väärtust vahetatakse omavahel, selliselt satuvad nad 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 rekursiivse protseduuri väljakutsega minnakse kummaski massiivi osas sügavusse. Rekursiooni lõpetamise tingimuseks on sorteeritava jada pikkus üks element.

Keerukus: O(n $\log_2$n) keskmisel juhul ja O(n^2) halvimal juhul

Eripärad: 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" tekivad pooled on suurustega `1` ja `n-1`.

Protseduur:

```
def kiir_sorteerimine(a, l, r):
    "Sorteerib massiivi a kiirsorteerimise meetodil."
    if l < r:
        piv = a[l]
        i = l+1
        j = r
        while i < j:
            while a[i] <= piv and i < r:
                i += 1
            while a[j] > piv and j > l:
                j -= 1
            if i < j:
                abi=a[i]
                a[i]=a[j]
                a[j]=abi
        if piv > a[j]:
            a[l] = a[j]
            a[j] = piv
        kiir_sorteerimine(a,l,j-1)
        kiir_sorteerimine(a,i,r)
```

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)*

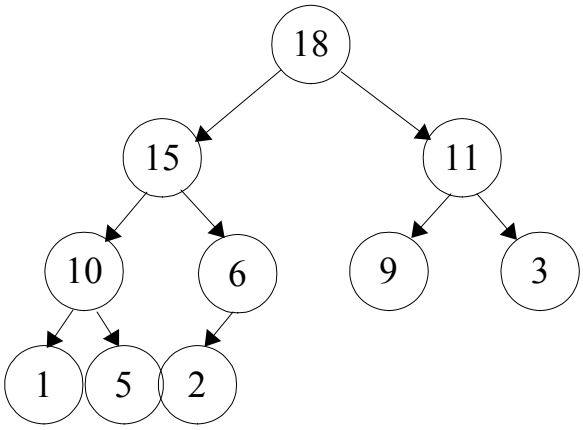
Kahendkuhi on kahendpuu kujuline andmestruktuur, mida tavaliselt realiseeritakse massiivi abil. Kuhjas olevad elemendid on teatud reegli alusel järjestatud.
Kuhjas hoitavad elemendid peavad olema seal vastavalt **kuhja omadusele** *(heap property)*: vanema väärtus ei ole väiksem järglaste väärtustest. See omadus kehtib iga kahendpuud moodustava alampuu kohta ja puu juur on seega kõige suurem element.

Massiivi abil realiseeritud kahendpuu iga tipp moodustab massiivi ühe elemendi. Puu juureks on element indeksiga 1, edasi tulevad juure järglased on 2. ja 3. element, nende järglased 4., 5., 6. ja 7. element jne. Puu kõik tasemed, va viimane, on täidetud.
Iga tipu I jaoks saab arvutada nii vanema kui laste indeksid:

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

Järgnevates algoritmides kasutatakse protseduure `Parent(i)`, `Left(i)` ja `Right(i)`.

Kuhja 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

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 protseduuri `Heapify`).

Sorteerimine

Klass: puusorteerimine

Põhiidee: Suuremat vajadust lisamälu järele ei ole (vaid paari muutuja jagu). Sorteerimine toimub kahes etapis:
1. Arvudemassiivist moodustatakse väärtuste ümberpaigutamise teel kuhi.
2. Kuhi sorteeritakse vastava algoritmiga.

Kõige keerulisema osa kogu tööst moodustab ühe kuhja tipu paigutamine tema õigele kohale oma järglaste suhtes. Protseduuri on vaja nii kuhja moodustamisel kui ka hilisemal sorteerimisel. Tipu jaoks kontrollitakse, kas tema kumbki järglane pole tipust suurem. Vajaduse korral tehase tippude väärtustega vahetus ja sama protseduuri korratakse rekursiivselt tipu kohta, mis eelmises operatsioonis kõige suuremaga ära vahetati (massiivi lahtri jaoks, kus **oli** suurim). Rekursioon võib vajaduse korral kuni leheni välja minna. Selleks kasutatakse protseduuri `Heapify`**.**

Kui massiivist on sel viisil kuhi moodustatud, järgneb sorteerimine:
1. Tipmine element võetakse kuhjast ära (tema on kõige suurem), selleks vahetatakse 1. ja viimane element ning kuhja suurust vähendatakse ühe võrra.
2. Kuhi moodustatakse uuesti sel teel, et tippu sattunud väike element viiakse vastava protseduuriga oma õigele kohale.

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 O(n log n).

Eripärad: Sorteeritud massiiv hakkab tekkima massiivi lõpust.

```
def heapify(a, n, i):
    "Abiprotseduur elemendi veeretamiseks kuhjas ülalt poolt õigele kohale."
    "a - kuhi"
    "n - kuhja suurus"
    "Parameeter i näitab, millist elementi on vaja õigesse kohta veeretada."
    "Protseduur on rekursiivne."
# Leiame tipu i vasaku (l) ja parema järglase (r) indeksi
    l = 2 * i
    r = 2 * i + 1
# Tuvastame, milline tipu i ja tema järglaste väärtustest on suurim
    if (l<= n-1) and (a[l] > a[i]):
        maxi = l
    else:
        maxi = i
    if (r <= n-1) and (a[r] > a[maxi]):
        maxi = r
# Kui tipp i ise ei ole suurim, vahetatakse tippude sisud ja kutsutakse
protseduur uuesti välja.
    if maxi != i:
        a[i],a[maxi] = vaheta(a[i],a[maxi])
        a = heapify(a, n, maxi)
    return a
```

Kuhja ehitamise olulise osa moodustabki kirjeldatud protseduur. Kogu ehitustööd alustatakse kuhja eelviimasest tasemest (viimast taset pole mõtet uurida, sest lehtedel pole järglasi) ja kõigile tippudele sealt ülespoole rakendatakse `Heapify`-protseduuri:

```
def BuildHeap(a):
    "Abiprotseduur kuhja ehitamiseks."
    "Sisendiks on massiiv a, väljundiks sama massiiv, mis on kuhjaks konedatud."
    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 - sisestatud massiiv sorteeritud kujul}"
    a = BuildHeap(a)
    kuhjasuurus = n
    for i in range(n-1 ,0, -1):
        a[0],a[i] = vaheta(a[0],a[i])
        kuhjasuurus -= 1
        a = heapify(a,kuhjasuurus,0)
    return a
```

Kirjeldatud meetodist võib leida ka nö tagurpidi variandi – kuhja põhiomaduseks on see, et vanem pole kunagi suurem oma järglastest, sellest tulenevalt on puu juur kõige väiksem ja sorteeritud massiiv kujuneb alates esimesest (kõige väiksemast) elemendist.

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 `O(n log n)`. Seega sorteerimine kuhjaga ja sorteerimine mestimisega, mis omavad ajalist keerukust `O(n log n)` halvimal juhul ning kiirsorteerimine, mis omab sama keerukust keskmisel juhul on need meetodid, mida võimaluse korral kasutada tasuks.

Sorteerimine loendamisega e loendamismeetod *(counting sort)*

Klass: loendussorteerimised

Põhiidee: (1954 a. H. H. Seward)
Loendamisega sorteerimise idee seisneb selles, et iga arvu kohta massiivis loetakse üle, 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. Loendurmassiivi algväärtustamine.
2. Erinevate massiivis olevate väärtuste loendamine.
3. Igale arvule eelnevate arvude kokku lugemine.
4. Arvude paigutamine uude massiivi vastavalt leitud kohale.

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

Eripärad: Algoritmi kasutusvaldkond on piiratud - tema abil sobib sorteerida positiivseid täisarve, mis on kindlates piirides. Huvitav, miks siiski ainult positiivseid?
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.

Protseduur:

```
def countsort(a,algus,lopp):
    "Sorteerib massiivi a loendamissorteerimise meetodil. "
    "Eelduseks on, massiivis on täisarvud vahemikus 0 .. 1000"
    maksarv = 1000
    loend = []
    b = []
    for i in range(maksarv+1):
        loend += [0]
    for i in range(algus,lopp+1):
        b += [0]
    "Loeme kokku, mitu tükki on mingit arvu 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]
    "Arvude paigutamine massiivi b sorteeritult ja b ümberkirjutamine a-sse."
    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 järjekorda peale hunnikutesse tõstmist hunniku sees muuta ei tohi. Kirjalikul kujul on vastavalt D. Knuthile see algoritm selgitatud 1929. a. sorteerimismasina juhendis, autor 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 *radix sort'i* puhul.

Keerukus: Algoritmi ajaline keerukus sõltub sellest, millist algoritmi ühe järgu järgi järjestamiseks kasutada. Kindlasti pole sobivad O(N2) 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.

Eripärad: 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.

Sorteerimine kastidega, 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 (vähe arve).
Kõigi kastide sisud sorteeritakse eraldi. Kuna väärtusi on igas kastis vähe, peaks sobima selleks üsna suvaline algoritm, ka ruutkeerukusega.
Seejärel kastid läbitakse kasvamise järjekorras 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).
Realiseerimiseks on vaja veel teist massiivi, mille suurus on sama sorteeritava massiiviga ja mida kasutatakse kastide haldamiseks. Iga kasti sisu jaoks moodustatakse lineaarne nimistu.

Sama ideed võiks kasutada ka täisarvude jaoks, kui arve ei ole väga palju. Näiteks jaotatakse arvud sajalise järgi osadeks, sorteeritakse osad ja ühendatakse osad uuesti üheks massiiviks.