TARTU ÜLIKOOL

TEADUSKOOL

PROGRAMMEERIMISE ALUSED

Õppevahend TK õpilastele

Ahto Truu

Tartu 2007

Käesolev õppevahend on mõeldud programmeerimise aluste süstemaatiliseks tutvustamiseks üldhariduskoolide õpilastele, eelkõige on silmas peetud Tartu Ülikooli juures tegutseva Teaduskooli informaatikaosakonna vajadusi. Lugejalt eeldatakse iseseisva töötamise oskust ja minimaalset mingis protseduurses keeles programmeerimise kogemust.

Saateks

Käesolev õppevahend on mõeldud programmeerimise aluste süstemaatiliseks tutvustamiseks üldhariduskoolide õpilastele.

Eelkõige olen silmas pidanud Tartu Ülikooli juures tegutseva Teaduskooli informaatikaosakonna vajadusi. Kuna Teaduskooli õppetöö toimub kaugõppe vormis, eeldan materjali kasutajalt iseseisva töötamise oskust ja minimaalset mingis protseduurses keeles programmeerimise kogemust. Aine esituses olen püüdnud keskenduda universaalsetele tõdedele, mis kehtivad kõigis levinud programmikeeltes, mitte õpetada ühe või teise keele süntaksit või mõne konkreetse programmeerimiskeskkonna iseärasusi.

Ajendatuna soovist laduda edasisteks õpinguteks kindel vundament, on käsitlus võrdlemisi matemaatiline ja range. Seetõttu ei tarvitse materjal olla sobilik noorematele õpilastele. Neil soovitaksin alustada "Programmeerimise algkursusest", mida on võimalik õppida nii Teaduskooli kaudu kui ka iseseisvalt aadressil `http://nrg.tartu.ee/algkursus/`.

Materjal on organiseeritud vastavalt Teaduskooli kaheaastase põhikursuse õppetöö korraldusele: esimesel aastal keskendume programmeermise üldistele ehituskividele — algoritm ja selle õigsus, valikulaused, korduslaused, alamprogrammid, algoritmide keerukus – teisel aastal kasutame neid vahendeid kombinatoorika ja klassikaliste andmestruktuuride standardalgoritmide ja nende koostamise võtete tundmaõppimiseks. Praeguses väljaandes on katmata kursuse kaks viimast teemat — sõnetöötlus ja arvutusgeomeetria.

Enne materjali sisu juurde asumist tahan tänada kõiki, kes selle valmimisele kaasa on aidanud: Tiigrihüppe Sihtasutust rahalise toetuse eest; Piret Truud, Indrek Jentsonit, Andrei Solntsevit ja Kristo Tammeojat paranduse-ettepanekute eest (kõik allesjäänud vead on muidugi minu vastutusel); Jüri Kihot, Rein Pranki ja Jaanus Pöialit, kelle varasematest materjalidest ma eeskuju võtnud ja ülesannete ideid laenanud olen; ja kindlasti ka Teaduskooli informaatikaosakonna õpilasi, kes õppematerjalide põlvkonnavahetuse käigus ebamugavusi kannatama pidid.

Ahto Truu, `ahto.truu@ut.ee`

Sisukord

Peatükk 1

Põhimõisted

Sisukord

Käesolevas peatükis tutvume kõige põhilisemate mõistetega, ilma milleta ei saa rääkida ei programmeerimisest ega programmidest.

1.1 Algoritm ja programm

1.1.1 Algoritm

Algoritmiks (ingl *algorithm*) nimetatakse samm-sammulist eeskirja mingi tegevuse sooritamiseks või eesmärgi saavutamiseks. Kõige sagedamini kasutatakse seda terminit just matemaatilise ülesande lahendamiseks mõeldud eeskirja kohta.

Sõna 'algoritm' tuleb IX sajandi araabia matemaatiku Abu Jafar Muhammad ibn Musa hüüdnimest al-Horaẓmi ('mees Horezmi linnast', tema sünnilinna, praeguse Usbekistani territooriumil asuva Hiiva tolleaegse nime järgi). Al-Horaẓmi XII sajandil ladina keelde tõlgitud tööde kaudu jõudsid Lääne-Euroopa matemaatikuteni muude Indiast ja Araabiast pärit ideede kõrval ka mitmed võrrandite lahendamise eeskirjad, mida hakatigi algoritmideks kutsuma.

Erinevate (mittematemaatiliste) algoritmidega puutume kokku iga päev: näiteks kokaraamatus olevad retseptid või sõbrale jäetud juhised kohtumispaika jõudmiseks. Algoritmid on ka koolis õpetatavad mitmekohaliste arvude kirjaliku liitmise-lahutamise-korrutamise-jagamise eeskirjad.

Eel- ja järeltingimus

Lisaks tegevuse sooritamiseks vajalike sammude loetelule on algoritmi kirjeldusel veel kaks tähtsat komponenti: eel- ja järeltingimus.

Algoritmi **eeltingimuseks** (ingl *precondition*) nimetatakse selle rakendamiseks vajalike eelduste loetelu.

Paljude algoritmide eeltingimused on nii keerulised, et nende täielik väljakirjutamine pole mõeldav. Sellisel juhul esitatakse ilmutatud kujul ainult spetsiifiliselt sellele algoritmile iseloomulikud eeldused ja jäetakse märkimata valdkonna üldisemad alused.

Näiteks kokaraamatus toodud retsepti alguses olev komponentide loetelu on selle retsepti eeltingimus — kui kokal vajalikke aineid pole, ei saa ta seda toitu valmistada. Lisaks ilmutatud kujul üles loetud toiduainetele on vaja ka töövahendeid (pliiti või ahju, potti või panni jne), aga neid tavaliselt eraldi ei nimetata. Töövahendite vajadus selgub alles valmistamisjuhendit lugedes. Kui tegemist on köögiga, kus eksootilisemaid vahendeid (näiteks *wok*-panni) ei ole, peab kokk lugema läbi ka valmistamisjuhendi ja saab alles selle põhjal otsustada, kas tal on kõik vajalikud eeldused täidetud.

Sõbrale kohtumispaika jõudmiseks jäetud juhised algavad tavaliselt mingist mõlemale poolele teadaolevast kohast — kohalejõudmise algoritmi eeltingimus on, et sõber läheb omal käel sellesse alguspunkti ja alustab juhiste järgi liikumist just sealt.

Kirjaliku jagamise algoritmi (õigupoolest mistahes jagamisalgoritmi) eeltingimus on, et jagaja ei ole null.

Algoritmi **järeltingimuseks** (ingl *postcondition*) nimetatakse lubadust, mis peab täituma, kui, alustades algoritmi eeltingimust rahuldavast olekust, sooritada kõik algoritmi sammud.

Igapäevase elu algoritmides pole järeltingimust enamasti välja toodud — see lepitakse osapoolte vahel kokku muul moel. Näiteks retsepti järeltingimus on lubatud roa valmimine, teejuhise järeltingimus soovitud sihtpunkti jõudmine ning kirjaliku jagamise eeskirja järeltingimus jagatise ja jäägi tekkimine kindlatesse kohtadesse arvutuse tulemuste hulgas.

Vahetingimus

Algoritmi uurimisel ja eriti selle õigsuse tõestamisel on oluline ka vahetingimuse mõiste.

Algoritmi **vahetingimuseks** (ingl *midcondition*) algoritmi järjestikuste sammude S_i ja S_{i+1} vahel nimetatakse tingimust, mis kehtib, kui algoritmi täitmine on jõudnud seisu, kus samm S_i on juba lõpetatud, kuid sammu S_{i+1} pole veel alustatud.

Kui vaadelda sammuga S_i lõppevat osa ühe algoritmina ja sammuga S_{i+1} algavat osa teise algoritmina, siis on nende sammude vahel kehtiv vahetingimus samaaegselt esimese osa järeltingimus ja teise osa eeltingimus.

Mitteformaalselt võib öelda, et eel- ja järeltingimus on vahetingimuse erijuhud — eeltingimus on vahetingimus, mis kehtib enne algoritmi esimese sammu täitmist, järeltingimus on vahetingimus, mis kehtib pärast algoritmi viimase sammu täitmist.

Näiteks kohtumispaika minemise algoritmis võib olla lõik: "... lähed kaks tänavavahet edasi ja pöörad paremale; seal on sild; lähed üle selle ja pöörad vasakule...". Keskmine fraas ei ole ilmselt algoritmi samm — sild kas on lubatud kohas või ei ole ja algoritmi täitja (antud juhiste järgi liikuja) ei saa seda fakti muuta. Tegemist on hoopis vahetingimusega.

Tõend

Algoritmi kirjelduses ilmutatud kujul välja toodud vahetingimust nimetatakse **tõendiks** (ingl *assertion*). Tõendid võimaldavad algoritmi täitjal kontrollida, kas algoritmi täitmine sinnamaani on olnud edukas.

Nagu eeltingimused, on ka vahetingimused sageli liiga suured, et neid täielikult välja kirjutada ja kontrollida. Mittetäielike vahetingimuste kasutamisel peab meeles pidama, et igasuguse tõendi mittekehtimine tähendab alati veaolukorda, kuid vigade puudumist tõestab ainult täieliku vahetingimuse kehtimine.

Silla puudumine eelmises näites tähendab kindlasti, et midagi on valesti läinud. Selle olemasolu ei tõesta veel, et orienteeruja on õiges kohas (ta võib olla ka mõne teise silla juures), kuid välistab siiski suure hulga valesid võimalusi (kõik need tänavad, kus pole üldse silda).

Mõnikord muudab vahetingimuse rikkumine algoritmi edasise täitmise võimatuks. Kui silda ei ole, ei saa sellest ka üle minna, seega tekiks selle algoritmi täitmisel tõrge ka ilma vahetingimust kontrollimata. Alati see siiski nii ei ole.

Näiteks juhiste "... pöörad paremale; seal on lehekiosk; lähed kaks tänavavahet edasi ja pöörad vasakule..." täitmist lehekioski puudumine otseselt ei sega, kuid tõenäoliselt ei jõua kord juba teelt eksinu ka õigesse kohta.

Just keerulisemates algoritmides on vahetingimuste väljatoomine väga kasulik. Vahetingimuste kontrollimine algoritmi täitmise ajal võimaldab kohe avastada, kui midagi on viltu läinud.

Diagnostikast veelgi olulisem on aga see, et vahetingimuste sõnastamine sunnib algoritmi koostajat oma loogika korralikult läbi mõtlema ja sageli jäävad vead üldse tegemata.

Algoritmi õigsus

Osutub, et vahetingimuste abil on võimalik ka algoritmi õigsust tõestada.

Olgu meil antud n-sammuline algoritm, mille sammud on $S_1, S_2, \ldots, S_n$, eeltingimus T_e ja järeltingimus T_j. Selle algoritmi **õigsuse tõestuseks** (ingl *proof of correctness*) nimetatakse tingimuste jada $T_0, T_1, \ldots, T_n$, mille korral kehtivad väited

- $T_0 = T_e$;
- iga $i \in 1, 2, \ldots, n$ korral: tingimuse T_{i-1} kehtivuse korral garanteerib sammu S_i täitmine tingimuse T_i kehtivuse;
- $T_n = T_j$.

Selle definitsiooni tähendus peaks olema ka intuitiivselt üsna hästi mõistetav. Algoritmi eeltingimusest alustades näitame algoritmi iga sammu juures, et selle sammu täitmine viib sammule eelneva vahetingimuse üle sammule järgnevaks vahetingimuseks. Kui viimase sammu järel saavutatav tingimus on algoritmi järeltingimus, siis ütlemegi, et algoritmi õigsus on tõestatud.

1.1.2 Programm

Programmiks (ingl *program*) nimetatakse algoritmi esitust mingis formaalses keeles, tavaliselt programmikeeles või masinakoodis.

Programmide näiteid on väljastpoolt arvutimaailma raske leida — tavaliselt kasutatakse igapäevases elus eeskirjade ja juhiste esitamiseks loomulikku keelt. Muidugi võib ka inimesele täitmiseks antud tegevuskava võrrelda programmiga, kuid üldjuhul ei täida inimene talle antud korraldusi mehaaniliselt ja mõtlemata.

Arvuti seevastu täidab programmi alati täpselt nii, nagu see on kirjas, juurdlemata selle üle, mida programmi autor seda kirjutades tegelikult tahtis. See (Greeri kolmanda seaduse nime all tuntud) põhimõte on nii oluline, et väärib eraldi esiletoomist:

> Arvutiprogramm teeb seda, mida te käsite tal teha,
> ja mitte seda, mida te tahate, et ta teeks.[1]

1.1.3 Programmikeel

Programmikeeleks (ingl *programming language*) nimetatakse algoritmide arvutile esitamiseks loodud formaalset keelt.

Kuigi programmikeeled on esimeste programmeeritavate arvutite ajast saadik arenenud tublisti inimsõbralikumaks, ei ole inimkeeles programmeeritavaid arvuteid siiski niipea loota. Näilisele inimsõbralikkusele vaatamata sarnanevad tänapäevased programmikeeled oma jäikuse poolest pigem masina kui inimese loomulikule keelele.

Loomulik keel ei sobi algoritmide arvutile esitamiseks peamiselt kahel põhjusel: esiteks on ta tänapäeva arvutite jaoks liiga keeruline ja teiseks on ta mitmetimõistetav. Isegi kui loomuliku keele keerukus õnnestuks ületada, jääb mitmetimõistetavus ikkagi takistama selle kasutamist algoritmide piisavalt täpseks esitamiseks.

Et vältida loomuliku keele mitmetimõistetavusest kerkivaid raskusi, kasutatakse formaalseid keeli ka väljaspool arvutimaailma. Lihtsaim praktiline näide on pangas maksekorralduse vormistamiseks täidetav plank — selle range struktuur ja piiratud väljendusvõimalused tagavad, et pangatöötaja saab üheselt aru, kellelt, kellele ja kui palju tuleb raha üle kanda. Vabas vormis kirjutatud maksekorralduse puhul ei tarvitse korralduse andja mõte üheselt arusaadav olla.

Nagu loomuliku keele puhul, tuleb ka formaalsest keelest rääkides eristada selle süntaksit ja semantikat.

[1] Arthur Bloch. *Murphy seaduste täielik kogu.* Ersen, 1999.

Keele **süntaksiks** (ingl *syntax*) nimetatakse keele lausetele esitatavaid vormilisi nõudeid. Teiste sõnadega, keele süntaks kirjeldab, millised on selles keeles vormiliselt korrektsed laused.

Keele **semantikaks** (ingl *semantics*) nimetatakse selle keele vormiliselt korrektsetele lausetele omistatavaid sisulisi tähendusi. Süntaktiliselt vigastele lausetele tavaliselt mingit tähendust ei anta.

Seejuures väärib eraldi märkimist, et süntaktiliselt korrektne lause võib semantiliselt ikkagi ebaõige olla. Näiteks on väide "Tartu on Eesti pealinn" vormiliselt küll igati korrektne (kõik sõnad on õigesti kirjutatud ja lauseehitusega sobivas käändes, pärisnimed algavad suurtähega jne), kuid sisuliselt vale — Eesti pealinn on Tallinn, mitte Tartu.

Nagu loomuliku keele lause, võib ka süntaktiliselt korrektne programm semantiliselt vigane olla. Paljud algajad programmeerijad keskendavad uue keele õppimisel kogu tähelepanu süntaksile ja unustavad, et tegelikult on peamine just semantika — keelekonstruktsioonide sisuline tähendus.

Ilmselt soodustab seda tendentsi ka asjaolu, et arvuti on võimeline automaatselt kontrollima ainult programmeerija kirjutatud teksti vormilist korrektsust, kuid mitte selle sisu vastavust programmeerija taotlusele (see ongi Greeri kolmanda seaduse kehtimise põhjus!). Sageli ei õnnestu süntaktiliselt vigast programmi üldse käima panna, kuid semantiliselt "logisev" programm läheb käima ja võib vahel isegi õigeid (või esmapilgul õigetena tunduvaid) tulemusi väljastada, jättes petliku mulje, nagu oleks ülesanne lahendatud.

1.1.4 Programmikeelte liigitus

Kuigi arvutite ja programmeerimise ajaloo jooksul on loodud palju erinevaid programmikeeli, on väga vähe selliseid, mis ei sarnane ühegi teisega. Sellest lähtuvalt on loomulik, et programmikeeli jagatakse nende ülesehituse ja eesmärkide järgi mitmetesse kategooriatesse.

Programmikeeli võib klassifitseerida nende otstarbe järgi — programmikeel võib olla kas üld- või eriotstarbeline. **Üldotstarbeline** (ingl *general-purpose*) programmikeel võimaldab kirjutada programme paljude erinevate valdkondade jaoks, **eriotstarbeline** (ingl *special-purpose*) programmikeel seevastu on orienteeritud ühe valdkonna ülesannete lahendamisele — "õige" valdkonna ülesannete lahendamine on spetsialiseeritud keeles mugavam kui üldotstarbelises keeles, kuid mõne teise valdkonna ülesannete lahendamine vastavalt ebamugavam või isegi võimatu.

Eriotstarbelised programmikeeled on näiteks andmebaaside töötlemise keel SQL ja dokumentide kirjeldamise keel PostScript (vastupidiselt laialt levinud arvamusele on PostScript täievoliline programmikeel, mitte pelgalt failivorming).

Teine võimalus on liigitada programmikeeli nende keelekonstruktsioonide järgi — keel võib olla kas imperatiivne või deklaratiivne. **Deklaratiivses** (ingl *declarative*) keeles programmeerides kirjeldab programmeerija, mida on vaja saavutada, **imperatiivses** (ingl *imperative*) keeles aga seda, kuidas soovitud tulemuseni jõuda.

Lõviosa tänapäeval kasutatavaid programmikeeli on imperatiivsed, kõige laiemalt levinud (kuigi mitte ainus praktikas kasutatav) deklaratiivne keel ongi andmebaaside töötlemise keel SQL.

Kolmas tähtsam võimalus on liigitada programmikeeli abstraktsioonitaseme järgi — mida abstraktsem on keel, seda vähem peab programmeerija selles keeles programmeerimisel teadma konkreetse arvuti tehnilistest detailidest. Sellest liigitusest tuleb lähemalt juttu järgnevates lõikudes.

Kuigi kolme eeltoodud kriteeriumi võib pidada peamisteks, ei ammenda nad kaugeltki kõiki võimalusi programmikeeli liigitada. Täiendavaid viiteid erinevat tüüpi keeltele ja neid kirjeldavatele allikatele on õppematerjali viimases peatükis.

Masinakood

Arvuti "loomulik keel" koosneb ainult arvudest. Kogu informatsioon, mida arvuti töötleb, esitatakse arvuti mälus arvudena kodeeritult. Kõik käsud, mida arvuti täita oskab, kodeeritakse samuti arvudena. Programmide ja andmete sellist esitust nimetatakse **masinakoodiks** (ingl *machine code*).

Tänapäevase arvuti masinakood on peaaegu kindlasti imperatiivne keel, mis koosneb tavaliselt mõnekümnest kuni mõnesajast käsust ja on üldiselt igal arvutitüübil erinev.

Kuna masinakood on igal arvutitüübil erinev, ei ole masinakoodis kirjutatud programme üldjuhul võimalik vähese vaevaga ühelt arvutilt teisele üle viia. Kui arvutid on piisavalt erinevad, on sageli otstarbekam programm uue arvuti jaoks lihtsalt uuesti kirjutada.

Masinakoodis programmeerimine on vaevarikas ja veaohtlik tegevus — näiteks tõenäosus, et mingit sõna esitavas arvujadas kogemata 97 asemel 98 kirjutanud inimesel oma viga märkamata jääb, on märksa suurem kui tõenäosus, et see juhtub selles sõnas 'a' asemel 'b' kirjutanud inimesel.

Kuigi masinakoodis programmeerimise järele pole tänapäeval enam praktilist vajadust (väga üksikud erandid välja arvatud), on arvuti töö paremaks mõistmiseks siiski kasulik tutvuda vähemalt mõne arvutitüübi masinakoodi ülesehitusega. Muude eelistuste puudumisel võiks selleks olla Donald E. Knuthi spetsiaalselt õppeotstarbeks loodud MIX[2] või MMIX[3].

[2] `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`

[3] `http://www-cs-faculty.stanford.edu/~knuth/mmix.html`

Sümbolkeel

Sümbolkeeleks (ingl *symbolic language*) nimetatakse programmikeelt, milles programmeerimisel kasutatakse operatsioonidele ja andmetele viitamiseks arvuliste koodide asemel sümbolkujul nimesid. Sümbolkeeli liigitatakse kõrg- ja madalkeelteks.

Madalkeele (ingl *low-level language*) käsud vastavad üldiselt üsna üksühele masinakoodi käskudele. Seega on madalkeeles kirjutatud programm endiselt seotud konkreetse arvutitüübi masinakoodiga, kuid sümbolesitus võimaldab programmeerijal kergemini vigu vältida ning juba tehtud vigu avastada ja parandada.

Kõrgkeel (ingl *high-level language*) on konkreetse arvuti ja opsüsteemi ülesehitusest sõltumatu, selle asemel on kõrgkeel orienteeritud programmi loogika ja töödeldavate andmete struktuuri paremale esitamisele. Kõrgkeeles kirjutatud programmi on võimalik ka ühelt arvutitüübilt teisele üle viia.

Kompilaator ja interpretaator

Kuna arvuti sümbolkeeles kirjutatud programme otse täita ei saa, tuleb sümbolkeelne programm enne täitmist masinakoodi tõlkida. Kõige parem on lasta see tüütu töö arvutil endal ära teha. Selleks vajalikku programmi nimetatakse **translaatoriks** (ingl *translator*).

Translaatorit, mis tõlgib ja täidab programmi ühe käsu kaupa, nimetatakse **interpretaatoriks** (ingl *interpreter*); translaatorit, mis tõlgib korraga terve programmi või mooduli ja salvestab tulemuse hilisemaks kasutamiseks, nimetatakse **kompilaatoriks** (ingl *compiler*).

Praktikas on levinud ka mitmesugused hübriidlahendused. Näiteks Java kompilaator tõlgib Java-programmi spetsiaalsesse vahekeelde (nn baitkoodi), mille täitmiseks kasutatav programm (nn Java virtuaalmasin) on sisuliselt baitkoodi interpretaator.

1.2 Programmi elemendid

Valdava enamiku tänapäevaste programmikeelte süntaksi elemendid jagunevad kolme liiki: primitiivid, juhtkonstruktsioonid ja deklaratsioonid.

1.2.1 Primitiivid

Primitiivideks (ingl *primitive*) nimetatakse neid korraldusi, mille täitmise tulemusena tegelikult soovitud tulemus saavutatakse. Näiteks retseptis on primitiivid otseselt toiduvalmistamise operatsioonid: “hakkida”, “koorida”,

"keeta" jne. Analoogiliselt on sõbrale kohtumispaika minekuks jäetud juhistes primitiivid otseselt tema liikumist suunavad fraasid: "keera vasakule", "keera paremale", "mine N tänavavahet edasi" jne.

Primitiivid võivad olla parameetriteta või parameetritega. Parameetriteta primitiiv koosneb ainult korraldusest midagi teha: "istu", "seisa" jne.

Parameetritega primitiiv sisaldab tegevust täpsustavaid lisandeid. Ilmselt on lihtne näha, et juhised "mine 2 tänavavahet edasi" ja "mine 3 tänavavahet edasi" on üsna sarnased, erineb ainult üks detail — tänavavahede arv. Seda muutuvat detaili nimetataksegi üldkujulise primitiivi "mine N tänavavahet edasi" **parameetriks** (ingl *parameter*).

Nii loomulikes kui ka formaalsetes keeltes on parameetritega primitiive rohkem kui parameetriteta primitiive, kuigi loomuliku keele puhul pole see vahetegemine alati selge — näiteks, kas "keera vasakule" ja "keera paremale" on kaks eraldi primitiivi või hoopis üldkujulise primitiivi "keera suunas X" kasutamine erinevate parameetritega? (Hiljem näeme, et selliste küsimuste lahendamisega tuleb tegeleda ka programmeerimise käigus.)

1.2.2 Juhtkonstruktsioonid

Juhtkonstruktsioonideks (ingl *control structure*) nimetatakse programmi elemente, mis määravad, milliseid primitiive, millises järjekorras ja kui palju kordi täidetakse.

Seni on näidetes esinenud ainult üks juhtkonstruktsioon: **järjend** (ingl *sequence*) — kõik primitiivid täidetakse juhistes antud järjekorras, ühtegi vahele jätmata või kordamata. Järjend on enam-vähem samal kujul olemas ka kõigis programmikeeltes — üldiselt täidetakse muude konstruktsioonide puudumisel primitiive just selles järjekorras, nagu nad programmi tekstis kirjas on.

Huvitavamate ja keerulisemate juhtkonstruktsioonide uurimisele pühendame kolm järgmist peatükki, sellepärast neil praegu pikemalt ei peatu.

1.2.3 Deklaratsioonid

Deklaratsioonid (ingl *declaration*) kirjeldavad objekte, millega programmis edaspidi tegemist tuleb. Deklaratsioonid ei tee otseselt midagi (selleks on primitiivid) ega suuna ka programmi käiku (selleks on juhtkonstruktsioonid).

Deklaratsioone programmis võib võrrelda tegelaste tutvustusega näidendi või filmistsenaariumi alguses või definitsioonidega matemaatikaõpikus, sest nende ülesandeks on panna paika "tegelased" ja fikseerida "mõisted", millega programm edaspidi opereerima hakkab.

1.3 Muutuja, väärtus, tüüp

1.3.1 Muutuja

Arvuti mälu koosneb nummerdatud mälupesadest. Igasse mälupessa võib salvestada ühe arvu[4], mis püsib seal seni, kuni arvuti välja lülitatakse või kuni samasse mälupessa mõni teine arv salvestatakse.

Programmi töö ajal mingi väärtuse hoidmiseks eraldatud mälupesa nimetataksegi **muutujaks** (ingl *variable*).

Masinakoodis kasutab programmeerija mälupesa poole pöördumiseks selle numbrit, mis pole kuigi mugav — on ju üsna tüütu meeles pidada, et parasjagu töödeldava arvujada minimaalne väärtus on mälupesas number 325452, maksimaalne väärtus mälupesas 325456 ja jada elemendid veel mingite muude numbritega mälupesades.

Sümbolkeeles annab programmeerija igale muutujale nime. Üldiselt peaks muutuja nimi kirjeldama selles hoitava väärtuse sisu. Näiteks võib jada minimaalse elemendi väärtuse hoidmiseks kasutatavat muutujat nimetada *min* ja maksimaalse elemendi väärtuse hoidmiseks kasutatavat *max*.

Muidugi on muutujale antava nime ja selles hoitavate andmete vastavus programmeerija asi. Arvuti sellest kinnipidamist kontrollida ei saa — kui arvuti suudaks inimese mõtteid lugeda, oleks programmeerimine kui amet juba kadunud.

Siiski on programmeerija enda huvides muutujatele mõistlikud nimed panna, vastasel korral ei saa ta ise ka varsti aru, millised andmed kuhu salvestatud on — rääkimata juba suurtest tarkvaraprojektidest, kus ühe programmi kirjutamisega tegeleb mitmete aastate jooksul kümneid või isegi sadu programmeerijaid.

1.3.2 Väärtus ja tüüp

Nagu eelpool öeldud, on arvuti riistvara seisukohalt mälus ainult arvud. See, et mõnda arvu tõlgendatakse arvuna, mõnda sümboli koodina ja mõnda veel mingil muul moel, on programmeerijate omavahelise kokkuleppe küsimus ja arvuti protsessor ei tea sellest üldiselt midagi.

Masinakoodis on nende vastavuste meelespidamine programmeerija mure, sümbolkeeles võtab (vähemalt osa) selle(st) vaeva(st) enda kanda translaator. Selleks, et translaator teaks, mismoodi ühe või teise mälupesa sisuga ümber käia, tuleb muutuja kirjeldamisel teatada selle **tüüp** (ingl *type*), see tähendab, öelda, millist liiki väärtusi selles muutujas hoidma hakatakse.

[4] Tegelikult on asi veidi keerulisem, aga täpsemaks selgituseks pole siinkohal ruumi.

Tavaliselt on programmikeeltes eraldi tüübid täis- ja reaalarvude, märkide (sümbolite), tekstide (sõnede) ja tõeväärtuste esitamiseks. Paljudes keeltes on lisaks neile olemas spetsiaalsed tüübid kuupäevade, kellaaegade jmt suuruste jaoks.

Abstraktne ja konkreetne tüüp

Abstraktne andmetüüp (ingl *abstract data type, ADT*) koosneb väärtusvarust ja operatsioonivarust.

Andmetüübi **väärtusvaru** (ingl *domain*) näitab, milliseid väärtusi on võimalik seda tüüpi muutujasse salvestada, ja **operatsioonivaru**, mida on võimalike nende väärtustega peale hakata.

Näiteks Java täisarvutüüp *int* võimaldab salvestada täisarvulisi väärtusi lõigus $-2\,147\,483\,648$ kuni $2\,147\,483\,647$ ning sooritada nende väärtustega aritmeetilisi tehteid, võrrelda neid omavahel jmt.

Konkreetne andmetüüp (ingl *concrete data type*) erineb abstraktsest selle poolest, et määrab kindlaks ka selle tüübi kõigi võimalike väärtuste esitamise arvuti mälus. Veidi lihtsustades võib öelda, et konkreetne tüüp seab abstraktse tüübi igale võimalikule väärtusele vastavusse arvu, millena see väärtus arvuti mällu salvestatakse.

Tavaliselt jätab kõrgkeele kirjeldus andmetüüpide väärtuste esituse translaatori kirjutaja otsustada. Sel juhul on kõrgkeele kirjelduses andmetüübid antud abstraktsete tüüpidena.

Liht- ja liittüüp

Andmetüüpi, mille võimalikud väärtused on programmikeele jaoks atomaarsed, jagamatud suurused, nimetatakse **lihttüübiks** (ingl *simple type*). Tüüpi, mille võimalikud väärtused koosnevad selgelt eristatavatest osadest, nimetatakse **liittüübiks** (ingl *composite type*).

Näiteks märgitüüp on lihttüüp — üheski üldotstarbelises keeles ei vaadelda tähemärke kui mingitest osadest koosnevaid moodustisi, vaid ikka kui terviklikke ja jagamatuid suurusi.

Tekstitüüp seevastu on liittüüp — kõigis üldotstarbelistes keeltes, kus tekstitüüp üldse olemas on, vaadeldakse teksti kui märgijada, millega võib manipuleerida ühe tervikuna, kuid mida saab töödelda ka üksikute märkide kaupa.

Arvutüübid on üldotstarbelistes keeltes enamasti lihttüübid, kuigi põhimõtteliselt poleks võimatu ka arvude käsitlemine numbrijadadena — samamoodi, nagu tekste vaadeldakse märgijadadena.

Lisaks keeles olemasolevatele standardsetele tüüpidele on programmeerijal enamasti võimalik defineerida ka uusi tüüpe konkreetse ülesande andmete mugavamaks esitamiseks. Kaks levinumat konstruktsiooni uute liittüüpide moodustamiseks on massiiv ja kirje.

Massiiv

Massiiv (ingl *array*) on nummerdatud elementide kogum. Nagu igal teisel muutujal, on ka massiivitüüpi muutujal üks nimi; massiivi elemendi poole pöördumiseks kasutatakse massiivi nime ja elemendi järjekorranumbrit ehk **indeksit** (ingl *index*). Paljudes programmikeeltes kehtib nõue, et ühe massiivi elemendid peavad kõik sama tüüpi olema.

Massiivid võivad olla ühe- või mitmemõõtmelised. Ühemõõtmelist massiivi võib kujutada jadana, mille igal elemendil on üks indeks — elemendi järjekorranumber jadas. Kahemõõtmelist massiivi võib kujutada tabelina, mille igal elemendil on kaks indeksit — rea- ja veerunumber. Kuigi praktikas seda eriti tihti vaja ei lähe, lubavad programmikeeled tavaliselt kasutada ka kolme- ja enamamõõtmelisi massiive.

Kirje

Kirje (ingl *record*) on mitmetüübiliste elementide kogum. Kirje elemente nimetatakse **väljadeks** (ingl *field*). Erinevalt massiivi elementidest on kirje väljadel tavaliselt nimed, mitte indeksid; kirje välja poole pöördumiseks kasutatakse koos muutuja ja välja nimesid.

Keerukamad tüübid

Paljud keeled lubavad liittüüpide moodustamise konstruktsioonide korduva kasutamisega kirjeldada kuitahes keerukaid tüüpe: võib defineerida kirje, mille üks väli on massiiv, mille elemendid on omakorda kirjed jne. Suurtes programmisüsteemides pole sugugi haruldased andmestruktuurid, kus sellisel moel on üksteise sees 4-5 kihti "organisatsiooni" enne kui tegelike andmeteni jõutakse.

Sellist hierarhilist moodustist on igati sobilik võrrelda alamkataloogide (kaustade) hierarhiaga arvuti kõvakettal — põhimõtteliselt võiks ju kõiki faile ühes kataloogis koos hoida, aga siis oleks seal segamini kõikvõimalikke faile, millel omavahel mingit pistmist pole ja vajaliku info leidmine võib päris tülikas olla. Hoopis mõistlikum on failid teemade kaupa kataloogidesse jagada ja igale kataloogile asjakohane nimi panna. Liittüüpidel on programmi andmete korrastamisel täpselt samasugune funktsioon.

1.3.3 Avaldis

Valdava enamiku programmikeelte **avaldised** (ingl *expression*) on oma põhiomadustelt üsna sarnased matemaatikatunnist tuttavate aritmeetiliste avaldistega: avaldistes võib kasutada konstante ja muutujaid, tehtemärke ja funktsioonide tähiseid ning sulge tehete järjekorra märkimiseks.

Avaldise **väärtustamiseks** (ingl *evaluation*) — selle avaldise väärtuse arvutamiseks — asendatakse muutujate nimed selles avaldises nende muutujate väärtustega — see tähendab, nende väärtustega, mis on parajasti salvestatud vastavatesse mälupesadesse — sooritatakse vajalikud tehted ja lõpptulemust nimetatakse selle avaldise **väärtuseks** (ingl *value*).

Füüsikatunnist peaks olema hästi teada, et erinevat tüüpi suurusi (näiteks massi ja aega) ei saa omavahel kokku liita. Sarnased nõuded kehtivad ka programmeerimisel: selleks, et avaldises nõutud tehteid oleks võimalik sooritada, tuleb igat tehtemärki (**operaatorit** (ingl *operator*)) rakendada sobivatele väärtustele (**operandidele** (ingl *operand*)).

Translaator kontrollib programmis olevate avaldiste korrektsust andmetüüpide operatsioonivarude järgi — iga väärtusega tohib sooritada ainult neid operatsioone, mis kuuluvad vastava tüübi operatsioonivarusse.

Ka iga operatsiooni tulemusel on oma tüüp, mida arvestatakse selle tulemuse kasutamisel järgmistes operatsioonides — näiteks kahe täisarvu liitmisel on tulemuseks uus täisarv ja seda võib ka edaspidi kasutada ainult operatsioonides, mis kuuluvad täisarvutüübi operatsioonivarusse.

Erinevate keelte tüübisüsteemid on erinevad. Näiteks on mõnes keeles kahe täisarvu jagamisel tulemuseks uus täisarv (sellised keeled jagavad jäägiga), mõnes teises keeles aga reaalarv (sellised keeled jagavad täpselt[5]). Selleks, et kirjutada korrektseid programme, peab programmeerija tundma kasutatava keele tüübisüsteemi.

1.3.4 Omistamine

Imperatiivse keele kõige tähtsam primitiiv on **omistamine** (ingl *assignment*). Omistamiskäsk on tavaliselt kujul

$$muutuja \leftarrow avaldis$$

ning selle tähendus (semantika) on: arvutada antud avaldise väärtus ja salvestada tulemus antud muutuja väärtuse hoidmiseks kasutatavasse mälupessa. Selles mälupesas varem olnud andmed (muutuja esialgne väärtus) lähevad seejuures kaotsi.

[5] Õigem oleks öelda, et jagavad täpsemalt, sest tegelikult on reaalarvud arvutis ligikaudsed suurused. Arvutustäpsusest tuleb veidi lähemalt juttu edaspidi.

Omistamiskäsu semantika juures on oluline, et avaldise väärtus arvutatakse välja enne muutuja esialgse väärtuse rikkumist. See võimaldab kasutada omistamiskäske kujul

$$muutuja \leftarrow muutuja + 1.$$

Selline omistamiskäsk arvutab välja avaldise $muutuja+1$ väärtuse, kasutades selleks muutuja esialgset väärtust, ja alles pärast seda salvestab tulemuse *muutuja* mälupessa, suurendades sellega muutuja väärtust 1 võrra.

Omistamiskäsust (eriti selle viimatimainitud kujul) rääkides torkab silma, et muutuja osaleb omistamiskäsus kahes erinevas tähenduses — avaldise väärtuse arvutamisel kasutatakse muutujat kui väärtust (oluline on tema mälupesa sisu, mitte selle asukoht), väärtuse salvestamisel aga kui kohta (oluline on tema mälupesa asukoht, mitte see, mida selles mälupesas parajasti hoitakse).

Muutuja esimeses tähenduses kasutamisel saadavat väärtust nimetatakse selle muutuja **paremväärtuseks** (ingl *rvalue*), sest selles tähenduses kasutatakse muutujat omistamismärgist paremal pool; tema teises tähenduses kasutamisel saadavat mälupesa aga **vasakväärtuseks** (ingl *lvalue*), sest selles tähenduses kasutatakse muutujat omistamismärgist vasakul pool.

Konstantidel ja avaldistel on üldiselt ainult paremväärtused — avaldis $2 + 3$ esitab väärtust 5, aga ei osuta mingit andmete salvestamiseks sobivat kohta arvuti mälus.

Muutujatel on tavaliselt olemas nii vasak- kui ka paremväärtus. Erandi moodustavad muutujad, millele pole veel midagi omistatud. Paljudes keeltes kasutatakse sellisel juhul muutuja paremväärtusena neid andmeid, mis muutuja hoidmiseks kasutatavas mälupesas parajasti on (enamasti on seal eelmiste programmide tööst mällu jäänud rämps, millega arvutamine mingit mõistlikku tulemust ei anna). Mõnes keeles loetakse, et sellisel muutujal paremväärtus puudub ja katse seda kasutada on viga. Mõnes keeles antakse igale muutujale kohe selle loomisel kindel algväärtus (arvulist tüüpi muutuja korral tavaliselt 0, muudel tüüpidel kindlat traditsiooni pole).

Neist kolmest variandist on levinuim esimene — efektiivsuse tõttu. Paraku on see variant ka ohtlikem, sest väärtustamata muutujat kasutav programm hakkab küll tööle, aga võib igal käivitamisel erinevalt käituda — vastavalt sellele, mis parasjagu mälus olema juhtub. Selliseid ebajärjekindlalt avalduvaid vigu on üldiselt üsna raske leida.

Üks võimalus algväärtustamata muutujatest tingitud vigade vältimiseks on igale muutujale kohe selle loomisel mingi väärtus omistada. Isegi kui see väärtus pole programmi järgneva loogika seisukohalt sobiv, on programmi käitumine vähemalt järjekindel ja vea leidmine seetõttu märksa lihtsam.

1.4 Pseudokeel

Edasises vaatleme algoritmide kirjeldusi poolformaalses pseudokeeles. Iga algoritmi kirjeldus koosneb pealkirjast, eel- ja järeltingimusest, kasutatavate abimuutujate kirjeldusest ning algoritmi sammude loendist.

Paljud algoritmid on lisaks sellele näha ka viies erinevas keeles programmidena õppematerjali lisades. Nende näiteprogrammide eesmärk on illustreerida tekstis kirjeldatud põhimõtete praktilist realiseerimist erinevates keeltes.

Algoritmi kirjelduses näitab eeltingimus tavaliselt seda, millistes muutujates ja millisel kujul peavad olema esitatud algandmed selle algoritmi tööks, ja järeltingimus seda, millistes muutujates ja millisel kujul on algoritmi töö lõppedes vastus.

Algoritmi sammudena kasutame järgmisi primitiive:

- omistamiskäsk kujul

  $$muutuja \leftarrow avaldis$$

  arvutab antud avaldise väärtuse ja salvestab selle antud muutuja uue väärtusena;

- sisestuskäsk kujul

  `sisesta` *muutuja*

  loeb sisendist (kasutajalt) väärtuse ja salvestab selle antud muutuja uue väärtusena; ühe muutuja asemel võib olla ka muutujate loetelu, sel juhul loeb see käsk uued väärtused kõigile muutujatele nende loetelus esinemise järjekorras;

- väljastuskäsk kujul

  `väljasta` *avaldis*

  arvutab antud avaldise väärtuse ja väljastab selle kasutajale; ühe avaldise asemel võib olla ka avaldiste loetelu;

- kommentaarid kujul

  — kommentaari tekst

  jäetakse algoritmi täitmisel vahele; need on mõeldud ainult selgituseks lugejale.

Juhtkonstruktsioone (peale järjendi) meil esialgu ei ole. Nendega tutvume järgmistes peatükkides.

1.4.1 Näited

Lihtne tervitus

Esimene näide väljastab kasutajale tervituse.

Sellise programmiga on hea kontrollida programmeerimissüsteemi korrasolekut — kui isegi nii lihtsa programmi täitmine ei õnnestu, on arvatavasti töövahendid rikkis; keerulisema programmi korral võib ebaõnnestumise põhjuseks olla viga programmis. Siiski võimaldab see programm teha läbi programmi koostamise tsükli: teksti sisestamine, transleerimine, käivitamine.

Algoritm 1.1 Lihtne tervitus

1. `väljasta 'Tere, kasutaja'`

Interaktiivne tervitus

Õige natuke keerulisem näide, mis kasutab ka üht abimuutujat.

Algoritm 1.2 Interaktiivne tervitus
Abimuutujad: *nimi* — kasutaja nimi, tekst

1. `väljasta 'Teie nimi: '`
2. `sisesta` *nimi*
3. `väljasta 'Tere, '`, *nimi*

Tasub tähele panna, et viimases väljastamiskäsus on esimene koma väljastatava teksti osa, teine aga kahe käsus esineva avaldise (tekstikonstandi ja muutuja) eraldaja. Samalaadsetele detailidele tuleb tähelepanu pöörata ka reaalsetes programmikeeltes.

Vahetus

Kahe antud muutuja (a ja b) väärtuste vahetamine.

Algoritm 1.3 Vahetab kahe muutuja väärtused
Sisend: a, b — vahetatavad väärtused, mõlemad sama tüüpi
Väljund: a, b — väärtused vahetatud
Abimuutujad: c, sama tüüpi kui a ja b

1. $c \leftarrow a$
2. $a \leftarrow b$
3. $b \leftarrow c$

Selle algoritmi näitel saame illustreerida ka vahetingimuste kasutamist algoritmi õigsuses veendumiseks.

Vastavalt algoritmi kirjelduses olevale eeltingimusele peavad vahetatavad väärtused olema muutujates a ja b. Tähistame neid väärtusi vastavalt a_0 ja b_0. Kui lisaks tähistame puuduva väärtuse sümboliga $\perp$, on algoritmi eeltingimus

$$a = a_0,\ b = b_0,\ c = \perp.$$

Esimene samm omistab muutujale c muutuja a esialge väärtuse, seega pärast esimese sammu täitmist peab kehtima vahetingimus

$$a = a_0,\ b = b_0,\ c = a_0.$$

Analoogiliselt peab pärast teist omistamist kehtima

$$a = b_0,\ b = b_0,\ c = a_0$$

ja pärast kolmandat

$$a = b_0,\ b = a_0,\ c = a_0,$$

mis, nagu näha, sisaldab endas ka programmi järeltingimust. Lisaks lubatud vahetusele on muutunud ka c väärtus, aga kuna c oli algusest peale kuulutatud abimuutujaks, on selle väärtuse rikkumine lubatud. Seega võime järeldada, et meie algoritm on korrektne.

Edaspidises lisame algoritmi olulisemad vahetingimused sageli kommentaaridena otse algoritmi sammude vahele.

Algoritm 1.3 Vahetab kahe muutuja väärtused
Sisend: a, b — vahetatavad väärtused, mõlemad sama tüüpi
Väljund: a, b — väärtused vahetatud
Abimuutujad: c, sama tüüpi kui a ja b

— vahetingimus: $a = a_0$, $b = b_0$, $c = \perp$

1. $c \leftarrow a$

   — vahetingimus: $a = a_0$, $b = b_0$, $c = a_0$

2. $a \leftarrow b$

   — vahetingimus: $a = b_0$, $b = b_0$, $c = a_0$

3. $b \leftarrow c$

   — vahetingimus: $a = b_0$, $b = a_0$, $c = a_0$

Õigete vahetingimuste leidmine on sageli lihtsam programmi või algoritmi tekstis tagantpoolt ettepoole liikudes.

Vaatleme näiteks ruutvõrrandi lahendamise algoritmi 1.4, mis kasutab üldtuntud valemit

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Muidugi on sama üldtuntud ka ruutvõrrandi lahendivalemi rakendamise eeltingimused, aga oletame hetkeks, et me neid ei tea. Sellisel juhul saame algoritmi täitmiseks vajalikud eeltingimused kõige lihtsamalt leida just tagantpoolt ettepoole liikudes.

Algoritm 1.4 Lahendab ruutvõrrandi
Sisend: a, b, c — võrrandi $ax^2 + bx + c = 0$ kordajad
Väljund: x_1, x_2 — võrrandi lahend
Abimuutujad: d — abimuutuja

1. $d \leftarrow b^2 - 4ac$
2. $x_1 \leftarrow (-b - \sqrt{d})/(2a)$
3. $x_2 \leftarrow (-b + \sqrt{d})/(2a)$

Omistamiskäsu $x_2 \leftarrow \ldots$ täitmiseks peab olema võimalik arvutada selle paremal poolel oleva avaldise väärtus. Selleks omakorda on vaja, et murrujoone all ei oleks null ($2a \neq 0$, ehk $a \neq 0$) ja juuremärgi all ei oleks negatiivne arv ($d \geq 0$). Samad tingimused peavad kehtima ka omistamise $x_1 \leftarrow \ldots$ eel.

Selleks, et omistamise $d \leftarrow \ldots$ järel kehtiks järgmiste omistamiste jaoks vajalik tingimus ($d \geq 0$), ei tohi muutujale d omistatava avaldise väärtus olla negatiivne ($b^2 - 4ac \geq 0$, ehk $b^2 \geq 4ac$).

Lisades need tingimused algoritmi kirjeldusse, saamegi tulemuse, mille õigsust on võimalik rangelt tõestada.

Algoritm 1.4 Lahendab ruutvõrrandi
Sisend: a, b, c — võrrandi $ax^2 + bx + c = 0$ kordajad, $a \neq 0$, $b^2 \geq 4ac$
Väljund: x_1, x_2 — võrrandi lahend
Abimuutujad: d — abimuutuja

— vahetingimus: $a \neq 0$, $b^2 \geq 4ac$

1. $d \leftarrow b^2 - 4ac$

— vahetingimus: $a \neq 0$, $d \geq 0$

2. $x_1 \leftarrow (-b - \sqrt{d})/(2a)$
3. $x_2 \leftarrow (-b + \sqrt{d})/(2a)$

Muidugi ei maksa eelnevast järeldada, et vahetingimuste hoolikas jälgimine on vajalik ainult matemaatiliste algoritmide õigsuses veendumiseks. Peaaegu kõigil primitiividel on mingid eeltingimused, mille rikkumise tagajärjeks on kas programmi töö katkemine veateatega või lihtsalt ebaõiged tulemused. Ja programmi autori mainele ei tule kasuks kumbki variant.

Ülesanded

Ülesanne 1.1 *Leida igapäevasest elust vähemalt kaks erinevat algoritmi. Tuua välja nende sammud ning eel- ja järeltingimused. Sõnastada ka mõni vahetingimus.*

Ülesanne 1.2 *Uurida oma programmeerimissüsteemi andmetüüpe.*

- *Millised tüübid on olemas?*
- *Milline on iga tüübi väärtus- ja milline operatsioonivaru?*
- *Kui palju igat tüüpi muutuja mälus ruumi võtab?*

Kontrolltööks loetleda kasutatava programmeerimissüsteemi täisarvutüübid ja nende väärtusvarud. Miks pole väärtusvarude otspunktid "ümmargused" arvud?

Ülesanne 1.3 *Kirjutada algoritm ja (vabalt valitud keeles) programm, mille täitmise käigus omistatakse muutujale s kolme muutuja x, y ja z väärtuste summa ning muutujale k nende väärtuste aritmeetiline keskmine. Millised peaks olema s ja k tüübid, kui x, y ja z on täisarvud? Miks?*

Ülesanne 1.4 *Kirjutada programm, mis sisestab täisnurkse kolmnurga kaatetite pikkused ning väljastab selle kolmnurga hüpotenuusi pikkuse ja pindala. Millised on hüpotenuusi pikkuse ja pindala tüübid, kui kaatetite pikkused on reaalarvud? Aga kui kaatetite pikkused on täisarvud? Miks?*

Ülesanne 1.5 *Kirjutada programm, mis deklareerib kahe täisarvulise väljaga kirjetüübi ja neist kirjetest koosneva kolmeelemendilise massiivi, omistab selle massiivi kõigi kolme kirje mõlemale väljale mingid fikseeritud väärtused ning väljastab iga kirje sisu eraldi reale.*

Kirjandus

Programmeerimise õppimiseks ei pea muidugi läbi lugema kõiki järgnevaid raamatuid, aga kindlasti soovitan tutvuda Pólya' ülesannete lahendamise metoodikaga ja lugeda vähemalt ühte ülejäänud raamatutest — neis on arvutite ehituse ja tööpõhimõtete kohta palju huvitavat materjali, mille tundmine tuleb kasuks nii järgnevate peatükkide omandamisel kui ka üldisemalt.

György (George) Pólya. *Kuidas seda lahendada.* Valgus, 2001.
Maailmakuulsa matemaatiku raamat kirjeldab üldisi ülesannete lahendamise tehnikaid, mis sobivad suurepäraselt ka algoritmide ja programmide koostamiseks. 200 lk.

Rein Jürgenson. *Programmeerimine.* Valgus, 1989.
Programmeerimise algõpetus Basicu, Pascali ja Fortrani näitel. 376 lk.

Ülo Kaasik, Jüri Kiho, Mare Koit. *Kuidas programmeerida.* Valgus, 1990.
Programmeerimise algõpetus Basicu ja E-skeemide näitel. 264 lk.

Jüri Kiho. *Elementaaralgoritmid.* Tartu Ülikool, 1991.
Programmeerimise algõpetus E-skeemide näitel. 144 lk.

Rein Jürgenson. *Programmeerimise algkursus.* Tallinna Tehnikaülikool, 1999.
Programmeerimise algõpetus Pascali näitel. 88 lk.

J. Glenn Brookshear. *Computer Science: An Overview.* Addison-Wesley, 1999.
Varasemate trükkidega klassikaks saanud programmeerimise ja arvutiteaduse algõpetuse õpik. 609 lk.

Дж. Гленн Брукшир. *Введение в компьютерные науки.* Вильямс, 2001.
Eelmise tõlge. 688 lk.

Peatükk 2

Valikulaused

Sisukord

Vaid väga väike osa algoritmidest on kirjeldatavad alati täpselt samas järjekorras täidetavate operatsioonide järjendina. Käesolevas peatükis vaatleme valikulauseid, millega saab kirjeldada tegevuse suunamist vastavalt algoritmi täitmise ajal selguvatele tingimustele.

Kasutades näitena jälle kokaraamatut, kohtame retseptides sageli juhiseid, kuidas mõnd puuduvat komponenti teisega asendada (näiteks "...kui veiniäädikat käepärast pole, võib selle asemel kasutada ka sidrunimahla...") või võimaluse korral rooga mittekohustuslike lisanditega huvitavamaks teha (näiteks "...kõige peale võib puistata veidi jahvatatud kaneeli...").

Nii algoritme käsitlevas kirjanduses kui ka programmikeeltes kasutatakse tingimuste esitamiseks matemaatilisest loogikast laenatud vahendeid, sestap alustamegi peatükki loogika põhimõistete vaatlemisest.

2.1 Loogika alused

Loogika (ingl *logic*) on filosoofia, lingvistika ja matemaatika piiril asuv teadus, mis tegeleb arutluste ja tõestuste uurimisega, see tähendab eelduste ja nendest tehtavate järelduste tõelevastavuse uurimisega.

Õigupoolest ei ole formaalse loogika seisukohalt arutlusel ja tõestusel mingit vahet — arutluseks nimetatakse üldiselt mõtlemist, järelduste tegemist, tõestuseks aga tehtud järelduste õigsuse demonstreerimist kellelegi teisele. Loogika seisukohalt pole oluline, kas järeldusi tehakse omaks tarbeks või teistele esitamiseks — nende paikapidavust see ei muuda.

Võib öelda, et loogika peaeesmärk on leida sellised järelduste tegemise reeglid, et tõestest eeldustest tehtaks ainult tõeseid järeldusi. (See on muidugi üsna jäme lihtsustus, tegelikult on loogika uurimisvaldkond palju laiem ja probleemid mitmekesisemad.)

Analoogiliselt algoritmidele on ka loogikas väidete (ja ennekõike nendevaheliste seoste) täpsemaks esitamiseks kasutusel formaalsed tähistused.

2.1.1 Tõeväärtus

Loogilise arutluse elementaarosad on väited, mis võivad olla kas tõesed või väärad, näiteks "$2 + 2 = 4$" või "Eesti pealinn on Tartu".

Väidet nimetatakse **tõeseks** (ingl *true*), kui see vastab tegelikkusele ja **vääraks** (ingl *false*), kui see ei vasta tegelikkusele. Eelpool toodud väidetest on esimene tõene, teine aga väär.

Laused, mis midagi ei väida, ei saa olla ei tõesed ega väärad ning pole seetõttu ka loogika uurimisobjektid. Sellised on näiteks enamik küsi- ja hüüdlauseid.

2.1.2 Lausearvutus

Lausearvutus (ingl *propositional calculus*) on loogika osa, mis uurib keerulisemate väidete konstrueerimist lihtsamatest.

Lausearvutuse valemite koostamisel asendatakse lausetes olevad väited neid tähistavate muutujatega ja lausete omavahelised seosed loogiliste tehetega. Selleks jagatakse uuritav arutlus või tõestus lauseteks, need omakorda osalauseteks jne, kuni jõutakse väideteni, mida enam osadeks jagada ei saa. Saadud väiteid tähistatakse loogikas harilikult suurte ladina tähtedega ja iga sellise muutuja tõeväärtuseks loetakse loomulikult tema poolt tähistatava väite tõeväärtus.

Loogilised tehted, mille abil väited uuesti lauseteks seotakse, on eitus, konjunktsioon, disjunktsioon, implikatsioon ja ekvivalents. Mitmest seotud väitest koosnevaid avaldisi tähistatakse harilikult väikeste ladina tähtedega ja nende tõeväärtused defineerime järgnevates lõikudes.

Nagu aritmeetikas, võib ka lausearvutuses avaldise väärtus sõltuda tehete sooritamise järjekorrast. Nagu aritmeetikas, kasutatakse ka lausearvutuses tehete järjekorra märkimiseks sulge — sulgudes olevad tehted sooritatakse alati enne.

Nagu aritmeetikas, oleks ka lausearvutuses tülikas kõigis avaldistes kõigi tehete järjekorda sulgudega näidata, seepärast määratakse tehetele prioriteedid: kõrgeima prioriteediga tehe on eitus (sarnane miinusmärgi kasutamisele aritmeetikas), järgmine konjunktsioon (sarnane korrutamisele), seejärel disjunktsioon (sarnane liitmisele), seejärel implikatsioon ja lõpuks ekvivalents (sarnane võrdusele).

Eitus

Eitus (ingl *negation*) on unaarne ehk üheoperandiline tehe ja avaldise $\neg p$ (loetakse "mitte p") tõeväärtus on vastupidine muutuja või avaldise p tõeväärtusele.[1]

Loogilise eituse abil esitatakse lausearvutuses konstruktsioone, mida loomulikus keeles väljendatakse sõnade "ei", "mitte" ja muude negatiivsete vormide abil. Lausearvutuse valemi $\neg p$ tehniliselt kõige täpsem tõlge eesti keelde on väide "pole õige, et p kehtib" ehk lühemalt "pole õige, et p".

Näiteks, kui A tähistab väidet "Kati läks eile koos Matiga välja", siis $\neg A$ tähistab väidet "pole õige, et Kati läks eile koos Matiga välja". Oluline on tähele panna, et eitav väide ei täpsusta, kas Kati ei läinud üldse välja või läks ta välja küll, aga ilma Matita; samuti ei ütle eitav väide midagi Mati tegemiste kohta.

[1] Eituse tähistamiseks kasutatakse $\neg p$ asemel sageli ka kirjaviise $\sim p$ või $\overline{p}$.

Loomuliku keele väidet "Kati ei läinud eile koos Matiga välja" mõistavad paljud inimesed selliselt, et Kati ei läinud üldse välja; sageli tehakse sellisest sõnastusest koguni järeldus, et Mati läks välja ja Kati jäi koju.

Juba sellest näitest selgub, et loomuliku keele väidete lausearvutuse valemiteks teisendamine ei tarvitse alati lihtne ülesanne olla. Eituse puhul on oluline selgeks teha, millisele esialgse väite osale eitus rakendub (seda osa nimetatakse eituse fookuseks) ja ka lausearvutuse valemis eitust just samale osale rakendada, vastasel korral muudame lause valemiks teisendamisel selle mõtet ja võime kergesti valedele järeldustele jõuda.

Konjunktsioon

Konjunktsioon (ingl *conjunction*) ehk **loogiline "ja"** (ingl *boolean and*) on binaarne ehk kaheoperandiline tehe. Avaldis $p \wedge q$ (loetakse "p ja q") on tõene, kui nii p kui ka q on mõlemad tõesed, ning väär igal muul juhul.[2]

Konjunktsiooni abil esitatakse lausearvutuses kahe väite samaaegset kehtimist. Loomulikus keeles vastab sellele enamasti sidesõna "ja" kasutamine või lihtsalt väidete järjest esitamine.

Näiteks, kui A tähistab väidet "Kati õpib" ja B tähistab väidet "Mati õpib", siis $A \wedge B$ võib eesti keeles üles kirjutada nii kujul "Kati õpib. Mati õpib." kui ka kujul "Kati õpib ja Mati õpib" kui ka kujul "Kati ja Mati õpivad".

Üldiselt on konjunktsiooni $p \wedge q$ abil võimalik esitada väiteid, mille saab sõnastada kujul "kehtivad nii p kui ka q" ehk lühemalt "nii p kui ka q". Näiteks väidet "1 ja 1 on kokku 2" ei saa konjunktsioonina esitada, kuigi ka selles väites esineb sidesõna "ja".

Disjunktsioon

Disjunktsioon (ingl *disjunction*) ehk **loogiline "või"** (ingl *boolean or*) on samuti binaarne tehe. Avaldis $p \vee q$ (loetakse "p või q") on tõene, kui vähemalt üks väidetest p ja q on tõene, ning väär, kui kumbki neist ei kehti.

Loomulikus keeles vastab disjunktsioonile enamasti sidesõnade "või" või "ehk" kasutamine.

Näiteks, kui A tähistab väidet "Kati koristab tuba" ja B tähistab väidet "Kati laulab", siis $A \vee B$ võib eesti keeles üles kirjutada kujul "Kati koristab tuba või laulab".

Loomulikus keeles kasutatakse väidet "p või q" sageli tähenduses "kehtib kas p või q, kuid mitte mõlemad korraga". Näiteks väidet "Kati koristab oma toa ära või saab karistada" mõistetakse enamasti tähendavat, et Kati ei saa

[2] Konjunktsiooni tähistamiseks kasutatakse $p \wedge q$ asemel sageli ka kirjaviisi $p\&q$.

karistada, kui ta oma toa korda teeb. Sõna "või" sellise tõlgenduse nimi on loogikas **välistav "või"** (ingl *exclusive or*).

Implikatsioon

Implikatsioon (ingl *implication*) on samuti binaarne tehe. Avaldis $p \supset q$ (loetakse "kui p, siis q") on väär ainult siis, kui p kehtib, aga q ei kehti.[3]

Implikatsiooni $p \supset q$ nimetatakse ka järeldusseoseks, selles sõnastuses on p implikatsiooni eeldus ja q selle järeldus. Kuna implikatsioonitehte tulemus sõltub ainult avaldiste p ja q tõeväärtustest, mitte aga sisulisest põhjuslikust seosest nende väidete kehtivuse vahel, ei saa seda tehet päriselt samastada arutluse ja tõestuse mõistetega, millest oli juttu peatüki alguses.[4]

Loomulikus keeles kasutatakse implikatsiooni $p \supset q$ esitamiseks tavaliselt väljendeid "q, sest p" või "kui p, siis q" (vahel ka lühemalt "kui p, q"), kuid siis peetakse silmas ikka põhjuslikku seost. Seetõttu nimetatakse loomulikus keeles sageli implikatsiooni eeldust põhjuseks ja järeldust tagajärjeks.

Näiteks, kui A tähistab väidet "Kati ei koorista oma tuba" ja B tähistab väidet "Kati saab karistada", siis $A \supset B$ võib eesti keeles esitada kujul "kui Kati ei koorista oma tuba, saab ta karistada" ja sellisel juhul eeldatakse ikka põhjuslikku seost — Kati saab karistada just oma kohustuste täitmatajätmise eest, mitte lihtsalt niisama.

Levinuim viga nii implikatsioonitehte kui ka põhjusliku seose kasutamisel on implikatsiooni eelduse mittekehtimisest selle järelduse mittekehtimise tuletamine. See tähendab, kui kehtib $p \supset q$ ja p on väär, tehakse sellest eksijäreldus, et ka q peab olema väär. See pole aga õige ei formaalselt (kui p on väär, siis $p \supset q$ on tõene sõltumata q väärtusest) ega ka sisuliselt (ühel tagajärjel võib olla mitu erinevat põhjust, näiteks Kati võib oma toa ära koristada, kuid saada karistuse väikevenna kiusamise eest).

Ekvivalents

Ekvivalents (ingl *equivalence*) on samuti binaarne tehe. Avaldis $p \equiv q$ (loetakse "p parajasti siis, kui q") on tõene, kui p ja q tõeväärtused on samad, ning väär, kui p ja q tõeväärtused on erinevad.[5]

---

[3] Implikatsiooni tähistamiseks kasutatakse $p \supset q$ asemel sageli ka kirjaviise $p \rightarrow q$ või $p \Rightarrow q$.

[4] Olukord on sarnane eelmises peatükis vaadeldud Greeri kolmanda seaduse mõttega — formaalne loogika on abiks arutluste täpsel üleskirjutamisel, kuid ei asenda mõtlemist.

[5] Ekvivalentsi tähistamiseks kasutatakse $p \equiv q$ asemel sageli ka kirjaviise $p \sim q$, $p \leftrightarrow q$ või $p \Leftrightarrow q$.

Loomulikus keeles kasutatakse ekvivalentsi harva, enamasti matemaatilistes tekstides, kus $p \equiv q$ sõnastatakse kujul "p parajasti siis, kui q" või "p siis ja ainult siis, kui q".

Koondtabel

Lõpetuseks lausearvutuse põhitehete tõeväärtused ühtse koondtabelina.

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \supset B$	$A \equiv B$
väär	väär	tõene	väär	väär	tõene	tõene
väär	tõene	tõene	väär	tõene	tõene	väär
tõene	väär	väär	väär	tõene	väär	väär
tõene	tõene	väär	tõene	tõene	tõene	tõene

Tabel 2.1: Lausearvutuse tehted

2.1.3 Predikaatarvutus

Predikaatarvutus (ingl *predicate calculus*) on lausearvutuse laiendus, kus lausemuutujate asemel kasutatakse predikaate. Predikaatarvutus võimaldab esitada mitmeid praktikas olulisi seoseid ja järeldusi, milleks lausearvutuse vahenditest ei piisa.

Predikaat

Esimeses peatükis programmi elementidest rääkides oli juttu sellest, et primitiividel võivad olla ka parameetrid. Osutub, et ka loogikas on sageli kasulik üksikute lausemuutujate asemel vaadelda parametriseeritud väiteid.

Vaadeldes väiteid $A =$ "Kati õpib" ja $B =$ "Mati õpib", märkame, et nad on sama struktuuriga. Tähistades nende väidete ühise osa "x õpib" sümboliga $Q(x)$, saamegi **predikaadi** (ingl *predicate*), millest x asendamisel konkreetsete väärtustega võime tekitada erinevaid väiteid. Kui tähistame $k =$ "Kati" ja $m =$ "Mati", saame väite A esitada kujul $Q(k)$ ja väite B kujul $Q(m)$.

Ka matemaatikast tuntud võrdus- ja võrratusmärgid ($=$, $<$, $>$, $\leq$, $\geq$) on predikaadid, kuigi "$< (2, 4)$" asemel kirjutatakse "$2 < 4$".[6]

Sageli seab predikaadi esitatava väite olemus piiranguid objektidele, millele seda predikaati rakendada võib (näiteks ei ole ju mõistlik rääkida õppimisest elutute esemete puhul). Hulka, mille elementidele võib predikaati rakendada, nimetatakse **universumiks** (ingl *universe*).

[6] Kirjaviisi "$< (2, 4)$" nimetatakse prefiks-, "$2 < 4$" aga infikskujuks. Põhimõtteliselt pole põhjust, miks ka muid predikaate ei võiks $P(x, y)$ asemel kirjutada kujul xPy.

Nagu primitiivide puhul, pole ka predikaatide defineerimisel alati selge, millised väite osad peaks jätma fikseerituks ja millised eraldama parameetritena. Näiteks väite $C =$ "Kati meeldib Matile" üldistusteks sobivad

$$\begin{aligned} R_1(x) &= \text{“}x \text{ meeldib Matile”;} \\ R_2(y) &= \text{“Kati meeldib } y\text{'le”;} \\ R(x,y) &= \text{“}x \text{ meeldib } y\text{'le”.} \end{aligned}$$

Esialgse väite saab siis esitada kujul $R_1(k)$ või $R_2(m)$ või $R(k,m)$.

Võrreldes parameetriteta lausemuutujate kasutamisega võimaldavad predikaadid esitada rohkem infot erinevate väidete omavaheliste seoste kohta. Näiteks väited "Kati ja Mati õpivad ning Kati meeldib Matile" saame lausearvutuse valemina esitada kujul

$$A \wedge B \wedge C,$$

predikaatarvutuse valemina aga kujul

$$Q(k) \wedge Q(m) \wedge R(k,m).$$

Predikaatarvutuse valemist on kohe näha, et meil on tegemist korduvate väidete ja korduvate isikutega, lausearvutuse seisukohalt on tegu lihtsalt kolme samaaegselt kehtiva väitega.

Üldisuskvantor

Üldisuskvantor (ingl *universal quantifier*) on tehe, mida võib rakendada predikaatloogika avaldisele. Avaldis $\forall x : p$ (loetakse "iga x korral kehtib p") on tõene, kui väide p kehtib iga universumisse kuuluva objekti x jaoks.

Sageli on vaja märkida, et väide p kehtib ainult teatud hulga H (mitte terve universumi) elementide kohta. Seda saaks kirjeldada valemiga

$$\forall x : x \in H \supset p$$

(loetakse "iga x korral kehtib: kui x on hulga H element, siis kehtib p"), kuid sagedase kasutamise tõttu on lepitud kokku lühemas ja ülevaatlikumas kirjaviisis

$$\forall x \in H : p$$

(loetakse "iga x korral hulgast H kehtib p"). Näiteks väite "naturaalarvude hulga $\mathbb{N}$ kõik elemendid on mittenegatiivsed" võib esitada kujul

$$\forall n \in \mathbb{N} : n \geq 0.$$

Olemasolukvantor

Olemasolu- ehk **eksistentsikvantor** (ingl *existential quantifier*) on teine levinud kvantor. Avaldis $\exists x : p$ (loetakse "leidub x, mille korral kehtib p") on tõene, kui väide p kehtib vähemalt ühe (aga võimalik, et ka enam kui ühe) universumisse kuuluva objekti x jaoks.

Analoogiliselt üldisuskvantoriga võib ka olemasolukvantori kasutamisel näidata, millise hulga elemente me vaatleme. Näiteks väite "täisarvude hulgas $\mathbb{Z}$ leidub negatiivseid elemente" võib esitada kujul:

$$\exists x \in \mathbb{Z} : x < 0.$$

2.1.4 Loogiline samaväärsus

Loogikas on tähtsal kohale valemite **loogilise samaväärsuse** (ingl *logically equivalent*) mõiste.

Lausearvutuse valemeid p ja q nimetatakse loogiliselt samaväärseteks ja kirjutatakse $p = q$, kui nende tõeväärtused on samad nendes esinevate lausemuutujate kõigi võimalike tõeväärtuste korral.

Loogilist samaväärsust ei tohi segamini ajada ekvivalentsitehtega. Näiteks valemite $p = A \wedge B \supset A$ ja $q = A \wedge (B \supset A)$ tõeväärtused on samad, kui A ja B on mõlemad tõesed, seega sellisel juhul väide $p \equiv q$ kehtib. Siiski pole need valemid loogiliselt samaväärsed, sest kui A ja B on mõlemad väärad, pole p ja q tõeväärtused samad, seega ei saa me väita, et $p = q$.

Kõige lihtsam (kuigi sageli töömahukas) viis lausearvutuse valemite loogilise samaväärsuse kontrollimiseks on koostada nende valemite tõeväärtustabelid. Valemi tõeväärtustabelis on üks rida iga võimaliku selles valemis esinevate muutujate väärtustuse kohta.[7] Näiteks võib tuua eelmises lõigus vaadeldud valemite tõeväärtustabelid.[8]

			1		2			2		1	
A	B	A	$\wedge$	B	$\supset$	A	A	$\wedge$	$(B$	$\supset$	$A)$
v	v		v		**t**			**v**		t	
v	t		v		**t**			**v**		v	
t	v		v		**t**			**t**		t	
t	t		t		**t**			**t**		t	

Predikaatarvutuse valemite jaoks üldjuhul selliseid tabeleid koostada ei saa, sest predikaatide võimalike tõeväärtuste komplektide leidmiseks tuleb enne

[7] Kokku on n erineva muutujaga valemi tõeväärtustabelis 2^n rida.

[8] Tõeväärtustabeli kirjalikul koostamisel kantakse sellesse iga tehte tulemus vastava tehtemärgi alla tehete sooritamise järjekorras, nagu näidatud valemite kohal olevate numbritega.

kokku leppida, millisest universumist pärit objektidega me tegeleme ja milline on iga predikaadi sisuline tähendus. Ilma neid kahte parameetrit fikseerimata pole võimalik antud predikaadi $P(x)$ jaoks kindlaks teha, kas leiduvad x väärtused, mille korral $P(x)$ on vastavalt tõene või väär, seega me ei tea, kas tabelisse tuleb lisada read vastavate juhtude jaoks.

Näiteks täisarvude hulgal $\mathbb{Z}$ võib predikaat $P(x) = (x < 0)$ olla (sõltuvalt x valikust) nii tõene kui ka väär, kuid naturaalarvude hulgal $\mathbb{N}$ saab ta olla ainult väär.

Siiski on olemas ka predikaatarvutuse valemite paare, mis on samaväärsed igas interpretatsioonis (universumihulga ja predikaatide tähenduste fikseerimist nimetatakse valemi interpretatsiooniks). Järgmised samaväärused kehtivad alati:

$$\begin{aligned}
\neg\neg p &= p; \\
\neg(p \wedge q) &= \neg p \vee \neg q; \\
\neg(p \vee q) &= \neg p \wedge \neg q; \\
p \supset q &= \neg p \vee q; \\
p \supset q &= \neg(p \wedge \neg q); \\
p \equiv q &= (p \wedge q) \vee (\neg p \wedge \neg q); \\
p \equiv q &= (p \supset q) \wedge (q \supset p); \\
\neg(\forall x : p) &= \exists x : \neg p; \\
\neg(\exists x : p) &= \forall x : \neg p.
\end{aligned}$$

2.2 Loogilised avaldised

Siiani rääkisime loogika teooriast, nüüd pöördume taas programmeerimise poole ja vaatame, kuidas seda teooriat algoritmide kirjeldamisel rakendada.

2.2.1 Predikaadid

Igas programmikeeles on olemas terve hulk predikaate kontrollimaks mitmesuguseid tingimusi, mida erinevad andmeobjektid võivad rahuldada või mitte rahuldada.

Arvud

Kõigis levinud programmikeeltes on olemas predikaadid arvutüüpi väärtuste võrdlemiseks: $=$, $<$, $>$, $\leq$, $\geq$. Kuna märke $\leq$ ja $\geq$ arvuti klaviatuuril ega

programmikeelte lubatud sümbolite hulgas tavaliselt pole, kasutatakse nende predikaatide esitamiseks mingeid muid kombinatsioone, levinuimad on "<=" ja "=>"; võrdusmärgi asemel kasutatakse mitmetes keeltes "==".

Ettevaatlik peab olema reaalarvude võrdlemisel. Reaalarve hoitakse arvuti mälus ligikaudsete suurustena ja ümardamisvigade tõttu võib sageli saada ootamatuid tulemusi.

Harjutuseks võiks uurida, kui palju nulle peab avaldises $1+0{,}00001$ teises arvus koma ja ühe vahele panema, et arvuti arvaks, et summa on ikka 1. Siis võiks võtta selle väikese arvu ja võrrelda teda nulliga. Kes varem pole selliste asjade peale mõelnud, võib üllatuse osaliseks saada... Vähimat arvu x, mille korral $1 + x$ pole veel 1, nimetatakse selle arvutüübi "epsiloniks".

Võimalikud on ka vastupidised vead. Näiteks võib juhtuda, et 0,44 ja $0{,}11 + 0{,}33$ võrdlemine annab negatiivse tulemuse. Seda tüüpi vigade vältimiseks võib absoluutselt täpse võrdumise asemel nõuda, et arvude erinevus oleks väiksem mingist piisavalt pisikesest suurusest või piisavalt palju väiksem võrreldavatest suurustest. (Kes on matemaatikatunnis õppinud absoluutse ja suhtelise vea mõisteid, peaks märkama, et eelnimetatud asendustest esimene kontrollib absoluutse ja teine suhtelise vea suurust.)

Lisaks võrdlemispredikaatidele võib programmikeel pakkuda ka muid, aga need on üldiselt igal keelel erinevad.

Tekstid

Samuti on kõigis (või vähemalt peaaegu kõigis) programmikeeltes olemas predikaadid sümboli- ja tekstitüüpi väärtuste võrdlemiseks.[9]

Tavaliselt võrdlevad programmikeelte standardsed predikaadid tekstitüüpi väärtusi **leksikograafiliselt** (ingl *lexicographical order*) — võrdlemisel jäetakse vahele need sümbolid kummagi teksti algusest, mis on omavahel võrdsed ja otsus langetatakse esimese erineva sümboli põhjal; kui üks tekstidest otsa saab, loetakse tema väiksemaks. Samasugust järjestust kasutavad sõnaraamatud ja entsüklopeediad — leksikograafilisel võrdlemisel loetakse väiksemaks tekst, mis oleks sõnaraamatus eespool.

Erinevus on selles, et sõnaraamatutes tavaliselt suuri ja väikesi tähti ei eristata, aga enamikus programmikeeltes on sümbolite võrdlemine **tõstutundlik** (ingl *case-sensitive*). Kas suuremaks loetakse suuri või väikseid tähti, pole üldiselt määratud ning sõltub kasutatavast programmeerimissüsteemist ja arvuti seadetest.

[9] Hoiatuseks C ja Java kasutajatele — kuigi tekstitüüpi muutujate võrdlemine tavalise võrdlusoperaatori == abil on mõlemas keeles süntaktiliselt korrektne, pole tulemus sageli sugugi ootuspärane. Tekstitüüpi väärtuste võrdlemiseks tuleb C's kasutada funktsioone `strcmp()` ja `strncmp()`, Javas aga klassi `String` meetodeid `equals()` ja `compareTo()`.

Eesti täpitähti ei pea enamiku programmikeelte võrdlemispredikaadid üldse tähtedeks ja need satuvad sorteerimisel kuhu juhtub (ühe süsteemi piires ikka ühte kohta, aga üldiselt mitte sinna, kuhu nad eesti tähestiku järgi käima peaks). Kui on vaja korralikku eestikeelset järjestamist, tuleb selleks eraldi vaeva näha.

Lisaks võrdlemispredikaatidele on programmikeeltes sageli olemas predikaadid, mis kontrollivad, kas antud sümbol on number, täht (nagu juba öeldud, "täpilisi" üldiselt tähtedeks ei peeta) või mõnd muud liiki sümbol.

Mõnes programmikeeles on ka eraldi predikaadid kontrollimaks, kas antud teksti on võimalik tõlgendada arvu, kuupäeva või kellaajana. Milliseid arvude, kuupäevade ja aegade vorminguid need funktsioonid "tunnistavad", tuleb igal konkreetsel juhul eraldi uurida.

2.2.2 Loogilised tehted

Tavaliselt on programmikeeltes lausearvutuse tehetest olemas eitus, loogiline "ja", nii loogiline kui ka välistav "või" ja ekvivalents. Implikatsiooni eraldi tehtena enamasti realiseeritud ei ole, sest seda ei lähe programmeerimisel kuigi sageli vaja.

Kvantoreid üldotstarbelistes programmikeeltes tavaliselt ei ole, kuigi mõnes süsteemis olemasolevaid primitiive massiivist või muust andmekogust antud tingimustele vastavate elementide leidmiseks ja loendamiseks võib mingil määral kvantoritega võrrelda, samuti saab vajaduse korral nende abil (või päris ise) kvantorid realiseerida.

Loogikatehete kasutamisel peab tähelepanu pöörama sellele, et lisaks tavalistele loogikatehetele, mis opereerivad tõeväärtustega, on paljudes keeltes olemas ka **bitiloogika** (ingl *bitwise logic*) tehted, mis opereerivad täisarvudega.

Bititehted tõlgendavad täisarvu kahendkuju[10] iga numbrit 0 väära ja numbrit 1 tõese tõeväärtusena, sooritavad loogilise tehte oma operandide samades positsioonides olevate bittide vahel ja tagastavad tulemusena saadud bittidest moodustatud arvu.

2.2.3 Loogilised muutujad

Peaaegu kõigis tänapäevastes programmikeeltes on olemas eraldi andmetüüp tõeväärtuste esitamiseks. Seda tüüpi muutujale on võimalik omistada loogilise avaldise väärtus ja seejärel võib muutujasse salvestatud väärtust kasutada kõikjal, kus muidu oleks tulnud kasutada seda avaldist ennast.

[10] Kahendsüsteemi kirjeldus on olemas peaaegu kõigis arvutiõpikutes ja ka paljudes matemaatikaõpikutes.

Muutuja kasutamise üks eelis on see, et talle saab panna korraliku nime, seega on pärast programmi lugedes kergem aru saada, millist tingimust see avaldis esitab. Keerukamate avaldiste puhul ei tarvitse see avaldise enda lugemisest kohe selguda.

Teine eelis on see, et muutujasse salvestatud väärtust võib korduvalt kasutada ilma, et arvuti peaks kulutama aega avaldise väärtuse uuesti arvutamisele. Sellest saadav võit on muidugi tühine, kui avaldis koosneb ainult paari arvu võrdlemisest, kuid keerukate predikaatide kasutamisel võib vahe olla märgatav.

2.3 Valikulausete liigid

2.3.1 Ühene valik

Ühene valik (ingl *conditional statement*) on lihtsaim võimalik valikulause, kus mingi tegevus kas sooritatakse või jäetakse sooritamata vastavalt antud tingimuse kehtivusele.

Peatüki alguses toodud kokaraamatu näide kaneeli kasutamisest oli just ühese valiku näide — kui kokal oli kaneel käepärast, lisas ta seda toidule, vastasel korral ei teinud midagi.

Ühese valikulause üldkuju on

```
kui t
   käsud
lõppkui
```

Kui tingimus t kehtib, täidetakse *käsud* ja jätkatakse algoritmi täitmist võtmesõnale `lõppkui` järgnevast lausest. Kui tingimus t ei kehti, jäetakse *käsud* vahele ja jätkatakse kohe võtmesõnale `lõppkui` järgnevast lausest.

Oluline on tähele panna, kuidas mõjub valikulause kasutamine algoritmi vahetingimustele. Oletame, et valikulause eel kehtis vahetingimus p.

Kui tingimus t kehtib, on valikulause sees oleva käsujada eeltingimus $p \wedge t$ — kehtib kõik, mida me teadsime enne tingimuse t kontrollimist, lisaks saime teada, et ka t kehtib. Kui me sellest seisust valikulause sees olevaid käske täites jõuame tingimuseni q, siis, kuna `lõppkui` pole tegelikult käsk, vaid ainult valikulause lõpu näitaja, ei muuda see enam midagi ja seega jääb valikulause täitmise lõpuks kehtima vahetingimus q.

Kui tingimus t valikulause alguses selle kontrollimisel ei kehti, tähendab see, et me liigume kohe valikulause lõppu, kusjuures nüüd me teame, et kehtib $p \wedge \neg t$.

Võttes kokku need kaks ainuvõimalikku varianti valikulause läbimiseks, saame, et valikulause järel peab kehtima vahetingimus $q \vee p \wedge \neg t$.

Näiteks, kasutades eelmise peatüki lõpus toodud algoritmi kahe muutuja väärtuste vahetamiseks, võime koostada algoritmi kahe arvu järjestamiseks suuruse järjekorda.

Algoritm 2.1 Järjestab kahe muutuja väärtused suuruse järjekorda
Sisend: a, b — järjestatavad väärtused
Väljund: a, b — väärtused vahetatud nii, et $a \leq b$
Abimuutujad: c

1. `kui` $a > b$
2. $\quad c \leftarrow a$
3. $\quad a \leftarrow b$
4. $\quad b \leftarrow c$
5. `lõppkui`

Algoritmi õigsuses veendumiseks tähistame muutujate a ja b algväärtused vastavalt a_0 ja b_0 ning c puuduva algväärtuse $\perp$. Siis saame algoritmi eeltingimuseks

$$a = a_0 \wedge b = b_0 \wedge c = \perp.$$

Kui valikulause tingimus kehtib (see tähendab, kui $a_0 > b_0$), vahetavad selle sees olevad käsud muutujate a ja b väärtused ning valikulause lõppu jõuame vahetingimusega

$$a_0 > b_0 \wedge a = b_0 \wedge b = a_0 \wedge c = a_0.$$

Kui valikulause tingimus ei kehti (see tähendab, kui $a_0 \leq b_0$), jõuame valikulause lõppu vahetingimusega

$$a_0 \leq b_0 \wedge a = a_0 \wedge b = b_0 \wedge c = \perp.$$

Võttes need tingimused kokku ja jättes kõrvale abimuutuja c, mille väärtusest me niikuinii ei hooli, saame

$$(a_0 > b_0 \wedge a = b_0 \wedge b = a_0) \vee (a_0 \leq b_0 \wedge a = a_0 \wedge b = b_0),$$

millest ongi näha, et muutuja a lõppväärtus on kahest algväärtusest just väiksem ja muutuja b lõppväärtus neist kahest just suurem.[11]

[11] Lisaks on näha, et täielikud vahetingimused kasvavad tõesti väga ruttu suurteks ja kohmakateks avaldisteks, mistõttu järgnevas kasutame enamasti osalisi vahetingimusi.

Paljude programmikeelte süntaks lubab valikulauses tingimuse kehtimisel täidetavate käskude jada asemel ainult üht käsku või lauset. Sellistes keeltes on tavaliselt olemas vahendid mitme lause ühendamiseks üheks liitlauseks — tavaliselt tuleb liitlause moodustamiseks selle osad grupeerida võtmesõnade `begin` ja `end` või mingit liiki sulgude vahele.

Kui keeles on liitlause moodustamise võimalus, siis võib selliselt moodustatud liitlauset harilikult kasutada igal pool, kus on lubatud lihtlause kasutamine — muuhulgas ka järgnevalt vaadeldavates valikulausetes.

2.3.2 Kahene valik

Kahene ehk **alternatiivne valik** (ingl *alternative statement*) erineb ühesest selle poolest, et algoritmis määratakse mingi tegevus ka juhtumiks, kui kontrollitav tingimus ei kehti. Seega kahese valikulause täitmisel sooritatakse üks tegevus, kui tingimus kehtib ja mingi teine tegevus, kui tingimus ei kehti.

Peatüki alguses toodud näide sidrunimahla kasutamisest veiniäädika asemel oli kahese valiku näide — kui kokal oli veiniäädikat, kasutas ta seda; kui ei olnud, kasutas selle asemel sidrunimahla. (Mida teha siis, kui kumbagi pole, see retsept ei kirjeldanud. Algoritmi õigsuse seisukohalt on selline ühe võimaliku variandi vaatluse alt välja jätmine viga, kui kahest komponendist vähemalt ühe olemasolu pole algoritmi eeltingimus.)

Kahese valikulause üldkuju on

```
kui t
   tõene-käsud
muidu
   väär-käsud
lõppkui
```

Kui tingimus t kehtib, täidetakse *tõene-käsud* ja jätkatakse algoritmi täitmist võtmesõnale `lõppkui` järgnevast lausest. Kui tingimus t ei kehti, täidetakse *väär-käsud* ja jätkatakse võtmesõnale `lõppkui` järgnevast lausest.

Selle valikulause vahetingimuste uurimiseks oletame jälle, et valikulause eel kehtis vahetingimus p.

Siis jõuame tingimuse t kehtimise korral käsujada *tõene-käsud* täitmisele eeltingimusega $p \wedge t$. Kui *tõene-käsud* täitmise järel kehtib vahetingimus q_1, jõuame sel juhul valikulause lõppu vahetingimusega q_1.

Tingimuse t mittekehtimise korral jõuame käsujada *väär-käsud* täitmisele eeltingimusega $p \wedge \neg t$. Kui *väär-käsud* täitmise järel kehtib vahetingimus q_2, jõuame sel juhul valikulause lõppu vahetingimusega q_2.

Kokkuvõttes peab valikulause täitmise järel kehtima vahetingimus $q_1 \vee q_2$.

Algoritm 2.2 Lahendab lineaarvõrrandi
Sisend: a, b — võrrandi $ax + b = 0$ kordajad
Väljund: x_0 — võrrandi lahend, kui see on üheselt määratud

1. `sisesta` a, b
2. `kui` $a = 0$
3. `väljasta 'Pole ühest lahendit'`
4. `muidu`
5. $x_0 \leftarrow -b/a$
6. `väljasta 'Lahend on '`, x_0
7. `lõppkui`

Kindlasti tasub tähele panna, et kahene valikulause ei tarvitse olla samaväärne kahest ühesest valikulausest koosneva järjendiga

```
kui t
   tõene-käsud
lõppkui
kui ¬t
   väär-käsud
lõppkui
```

Kui tingimus t kehtib, võib *tõene-käsud* täitmine muuta programmi olekut nii, et esimesest valikulausest väljumise järel tingimus t enam ei kehti. Siis aga on $\neg t$ tõene ja vastavalt täidetakse ka *väär-käsud*.

Kui mingil põhjusel on siiski vaja alternatiivse valikulause asemel just üheseid valikulauseid kasutada, võib eelmises lõigus kirjeldatud viga vältida tõeväärtusmuutuja t_0 kasutamise abil:

```
t0 ← t
kui t0
   tõene-käsud
lõppkui
kui ¬t0
   väär-käsud
lõppkui
```

Selles versioonis salvestatakse tingimuse t esialgne tõeväärtus muutujasse t_0, kus see (eeldusel, et *tõene-käsud* sellele muutujale uut väärtust ei omista) säilib ka teise valikulause jaoks.

Veidi keerulisema näitena vaatleme kahte erinevat algoritmi kolmest arvust maksimaalse leidmiseks.

Algoritm 2.3 Leiab kolmest arvust maksimaalse
Sisend: a, b, c — uuritavad arvud
Väljund: $x = max(a, b, c)$

1. `kui` $a \geq b \wedge b \geq c$
2. $\quad x \leftarrow a$
3. `lõppkui`
4. `kui` $a \geq c \wedge c \geq b$
5. $\quad x \leftarrow a$
6. `lõppkui`
7. `kui` $b \geq a \wedge a \geq c$
8. $\quad x \leftarrow b$
9. `lõppkui`
10. `kui` $b \geq c \wedge c \geq a$
11. $\quad x \leftarrow b$
12. `lõppkui`
13. `kui` $c \geq a \wedge a \geq b$
14. $\quad x \leftarrow c$
15. `lõppkui`
16. `kui` $c \geq b \wedge b \geq a$
17. $\quad x \leftarrow c$
18. `lõppkui`

Algoritm 2.4 Leiab kolmest arvust maksimaalse
Sisend: a, b, c — uuritavad arvud
Väljund: $x = max(a, b, c)$

1. $x \leftarrow a$
2. `kui` $b > x$
3. $\quad x \leftarrow b$
4. `lõppkui`
5. `kui` $c > x$
6. $\quad x \leftarrow c$
7. `lõppkui`

Kuigi mõlemad algoritmid saavutavad lõpuks sama tulemuse, kasutab algoritm 2.3 võrdluspredikaati 12, algoritm 2.4 aga vaid 2 korda, seega on nende algoritmide efektiivsuse vahe 6-kordne.

2.3.3 Mitmene valik

Paljudes programmikeeltes on olemas ka **mitmene valik** ehk **lüliti** (ingl *switch*), mis võimaldab mingi avaldise väärtuse järgi valida ühe paljudest tegevustest.

Mitmese valikulause üldkuju on

```
valik a
  variant v_1
    käsud-1
  variant v_2
    käsud-2
  ...
  variant v_n
    käsud-n
  muidu
    muidu-käsud
lõppvalik
```

Kui avaldise a väärtus on v_1, täidetakse *käsud-1* ja jätkatakse algoritmi täitmist võtmesõnale `lõppvalik` järgnevast lausest, kui avaldise a väärtus on v_2, täidetakse *käsud-2* ja jätkatakse võtmesõnale `lõppvalik` järgnevast lausest jne. Kui avaldise a väärtus ei ole ükski v_i väärtustest, täidetakse *muidu-käsud* ja jätkatakse siis valikukäsu järelt.

Sageli seavad programmikeeled realisatsiooni efektiivsuse huvides piiranguid sellele, millist tüüpi avaldisi ja väärtusi võib mitmeses valikus kasutada. Kui on vaja teha mitmene valik mingit muud tüüpi avaldise põhjal või keeles, kus sellist konstruktsiooni üldse pole, võib selle asemel kasutada korduvaid kaheseid valikuid. Programmi parema loetavuse huvides tuleks võimalusel siiski kasutada mitmest valikut, mitte seda muude vahenditega imiteerida.

2.4 Lausete pesastamine

Struktuursete programmikeelte oluline ja praktikas väga kasulik omadus on võimalus juhtkonstruktsioone üksteise sisse asetada ehk **pesastada** (ingl *nest*). See tähendab seda, et igale poole, kuhu võib kirjutada primitiivi, võib selle asemel panna ka keerulisema konstruktsiooni — näiteks liitlause või valikulause.

Tegelikult oleme eelnevates näidetes pesastamist juba kasutanud, pannes valikulause sisse mitmest käsust koosneva järjendi (mis mäletatavasti oli ka üks juhtkonstruktsioonidest), samuti koostades järjendeid mitmetest valikulausetest.

Osutub, et järjend pole selles suhtes mingi erand, täpselt sama hästi võib ühe valikulause sisse panna ka teise valikulause, nagu näiteks järgnevas algoritmis, mis kontrollib, kas antud kolm arvu võivad olla mingi kolmnurga küljepikkused.

Algoritm 2.5 Kontrollib, kas antud arvud on kolmnurga küljepikkused
Sisend: a, b, c — uuritavad arvud
Väljund: diagnoos tekstina ekraanil

1. `kui` $a > 0 \wedge b > 0 \wedge c > 0$
   — a, b, c sobivad pikkusteks
2. `kui` $a + b > c \wedge a + c > b \wedge b + c > a$
   — sobivad kolmnurga küljepikkusteks
3. `väljasta 'Jah'`
4. `muidu`
   — ei sobi kolmnurga küljepikkusteks
5. `väljasta 'Ei'`
6. `lõppkui`
7. `muidu`
   — ei sobi üldse pikkusteks
8. `väljasta 'Ei'`
9. `lõppkui`

Lõigus 2.3.3 märgitud mitmese valiku realiseerimine kaheste valikute abil tuleb üsna loomulikult välja just pesastamise teel:

```
kui a = v_1
   käsud-1
muidu
   kui a = v_2
      käsud-2
   muidu
   ...
      kui a = v_n
         käsud-n
      muidu
         muidu-käsud
      lõppkui
   ...
   lõppkui
lõppkui
```

Ülesanded

Ülesanne 2.1 *Kasutades tähistusi A = "sajab vihma", B = "sajab lund", C = "on suvi", D = "on talv", esitada lausearvutuse valemitena väited*

kui sajab vihma, siis on suvi;
kui on talv, sajab lund;
suvel lund ei saja;
pole suvi ega talv.

Millised neist väidetest on tõesed?

Ülesanne 2.2 *Kasutades eelmise ülesande tähistusi, sõnastada eesti keeles väited, mida esitavad valemid*

$A \wedge B$;
$B \supset \neg C$;
$\neg(C \wedge D)$.

Millised neist väidetest on tõesed?

Ülesanne 2.3 *Kasutades tavalisi matemaatilisi tähistusi, esitada predikaatarvutuse valemitena väited*

täisarvude hulgas leidub nullist suuremaid;
kõik naturaalarvud on negatiivsed;
kõik naturaalarvud kuuluvad täisarvude hulka.

Millised neist väidetest on tõesed?

Ülesanne 2.4 *Sõnastada eesti keeles väited, mida esitavad valemid*

$\forall x \in \mathbb{Z} : x \in \mathbb{R}$;
$(\exists x \in \mathbb{Z} : x < 0) \wedge (\exists x \in \mathbb{Z} : x > 0)$;
$\exists x \in \mathbb{Z} : (x < 0 \wedge x > 0)$.

Millised neist väidetest on tõesed?

Ülesanne 2.5 *Koostada tehte välistav "või" ja valemi*

$$(p \vee q) \wedge \neg(p \wedge q)$$

tõeväärtustabelid ning veenduda, et nad on loogiliselt samaväärsed.

Ülesanne 2.6 *Shefferi kriips on binaarne loogiline tehe, mille tulemus $p|q$ on väär parajasti siis, kui p ja q on mõlemad tõesed. Shefferi kriips on universaalne tehe — tema abil on võimalik avaldada kõik teised tehted; näiteks eituse võib avaldada kujul*

$\neg p = p|p$.

Avaldada Shefferi kriipsu abil kõik lõigus 2.1.2 vaadeldud tehted.

Ülesanne 2.7 *Kirjutada programm, mis sisestab kolm arvu a, b ja c ning väljastab need mittekahanevas järjekorras. Kontrollida, et programm töötab õigesti ka siis, kui mõned arvudest on omavahel võrdsed.*

Ülesanne 2.8 *Kirjutada programm, mis sisestab neli arvu a, b, c ja d ning väljastab need mittekahanevas järjekorras.*

Ülesanne 2.9 *Kirjutada programm, mis sisestab ruutvõrrandi*

$$ax^2 + bx + c = 0$$

kordajad ja leiab selle reaalarvulised lahendid (kui need on olemas) või väljastab korrektse teate (kui võrrandil reaalarvulised lahendid puuduvad).

Ülesanne 2.10 *Antud kahe kolmnurga küljepikkused a_1, b_1, c_1 ja a_2, b_2, c_2. Kirjutada programm, mis kontrollib, kas need kolmnurgad on kongruentsed, sarnased või erinevad.*

Ülesanne 2.11 *Antud kolm naturaalarvu i, j, k, mis rahuldavad tingimust $1 \leq i < j < k \leq 9$. Kirjutada programm, mis kontrollib, kas nende arvudega märgitud ruudud asuvad allolevas tabelis samas reas, veerus või diagonaalis.*

1	*2*	*3*
4	*5*	*6*
7	*8*	*9*

Ülesanne 2.12 *Kirjutada programm, mis kontrollib, kas antud positiivne täisarv on liig- või lihtaasta number. (Aasta on liigaasta, kui tema number jagub neljaga, välja arvatud need aastad, mille number jagub sajaga, kuid ei jagu neljasajaga.)*

Kirjandus

Tõnu Tamme, Tanel Tammet, Rein Prank. *Loogika. Mõtlemisest tõestamiseni.* Tartu Ülikool, 1997.
Kõige mahukam seni eesti keeles ilmunud formaalse loogika õpik, mis sobib nii lause- ja predikaatarvutuse põhjalikumaks tundmaõppimiseks kui ka muude loogikavaldkondadega tutvumiseks. 424 lk.

Ene Grauberg. *Loogika ja vaidluskunst.* Olion, 1989.
Põhiliselt loogikale kui väitluse ja veenmise aparaadile keskenduv õpik, matemaatilisi formalisme on vähem. 95 lk.

A. Kolman, O. Zich. *Huvitav loogika.* Valgus, 1970.
Populaarne sissejuhatus formaalse loogika alustesse. 118 lk.

David Gries. *The Science of Programming.* Springer, 1981.
Programmeerimise põhitõdede ja -meetodite formaalne käsitlus. Väga sobiv neile, kes tahaks rohkem ja põhjalikumaid näiteid loogika rakendamisest programmeerimisel. 382 lk.

Peatükk 3

Korduslaused

Sisukord

Arvutite võimest täita miljoneid käske sekundis oleks üsna vähe kasu, kui iga käsk tuleks eraldi välja kirjutada — sellise lähenemise korral jätkuks kogu maailma programmeerijatest vaevalt mõne tänapäevase protsessori jõudluse ärakasutamiseks. Käesolevas peatükis vaatleme korduslauseid, mille abil on võimalik üht programmilõiku täita kuitahes palju kordi ja seega lühikese programmi abil väga suur hulk tööd ära teha.

3.1 Korduslause mõiste

Kordus- ehk **iteratsiooni-** ehk **silmuselause** (ingl *loop statement*) kõige levinum kuju on

```
senikui t
  käsud
lõppsenikui
```

Rida `senikui` t nimetatakse selle korduse päiseks ja lõiku *käsud* korduse kehaks. Kui päises olev tingimus t kehtib, täidetakse korduse kehas olevad käsud ja pöördutakse seejärel tagasi päise juurde. Kui tingimus ikka veel kehtib, täidetakse uuesti *käsud* jne, kuni ükskord saabub hetk, kui t enam ei kehti. Siis jätkub algoritmi täitmine `lõppsenikui` järel olevast lausest.

Edasise jaoks on kasulik tähele panna, et vastavalt definitsioonile on eeltoodud korduslause samaväärne (lõpmatu) konstruktsiooniga

```
kui t
  käsud
  kui t
    käsud
    kui t
      käsud
      ...
    lõppkui
  lõppkui
lõppkui
```

3.1.1 Korduse invariant

Korduslauset sisaldava algoritmi vahetingimuste leidmiseks pöörame kõigepealt tähelepanu sellele, et korduse kehas olevate käskude täitmise järel võib juhtuda, et kohe hakatakse neidsamu käske uuesti täitma. Seega on korduse keha ühe täitmise järeltingimus ühtlasi ka sellesama korduse keha järgmise täitmise eeltingimus (täpsemalt küll selle osa).

Seda korduse keha eel- ja järeltingimuse ühisosa nimetataksegi **korduse invariandiks** (ingl *loop invariant*), vahel ka **tsükli invariandiks**.

Selleks, et vajalik eeltingimus oleks täidetud ka korduse keha esimesel läbimisel, peab invariant kehtima ka vahetult enne korduslauset. Algoritmi samme, mis selle tagavad, nimetatakse **korduse algatamiseks** (ingl *loop initiation*).

Seega on korduslausega seotud vahetingimused

korduse algatamine
— vahetingimus: p
`senikui` t
— vahetingimus: $p \wedge t$
korduse keha
— vahetingimus: p
`lõppsenikui`
— vahetingimus: $p \wedge \neg t$

(kus p on muidugi selle korduse invariant).

Korduslause koostamist on kasulik alustada just selle invariandi leidmisest. Vaatleme näitena n-elemendilise arvumassiivi $a_{1\ldots n}$ elementide summeerimist nii, et korduse lõpuks oleks summa muutujas s.

Milline võiks olla selle korduse invariant? Ilmselt on meil summeerimise ajal osa elemente juba kokku liidetud ja osa veel liitmata. Kõige lihtsam on alustada summeerimist massiivi ühest otsast, siis saame juba liidetud ja veel liitmata elementide üle pidada arvet üheainsa abimuutuja i abil, mis osutab viimasena summasse liidetud elemendi positsiooni ehk järgmisena liidetavale elemendile eelnevat positsiooni.

Korduse invariant oleks seega $s = a_1 + a_2 + \ldots + a_i$ ja selle säilitamiseks peame iga järjekordse elemendi summasse liitmisel edasi nihutama ka järjehoidjat, see tähendab suurendama i väärtust.

Millised tegevused sobivad sellise korduse algatamiseks? Loomulik on alustada seisust, kus ükski element pole veel summasse liidetud; ka i peab siis olema 0 — järgmisena liidame summasse jada esimese elemendi a_1.

Kui kaua tuleks sellist kordust täita? Ilmselt seni, kuni on veel liitmata elemente, ehk siis seni, kuni kehtib $i < n$.

Seega otsitav kordus (koos algatamisega) võiks olla

$i \leftarrow 0$
$s \leftarrow 0$
`senikui` $i < n$
$i \leftarrow i + 1$
$s \leftarrow s + a_i$
`lõppsenikui`

3.1.2 Korduse osaline õigsus

Korduse **osalise õigsuse** (ingl *partial correctness*) tõestamiseks on vaja tõestada

- korduse algatamise õigsus: on vaja näidata, et korduse algatamise käsud tagavad korduse invariandi p kehtivuse vahetult korduslause täitmise eel;
- korduse invariandi säilimine: on vaja näidata, et eeltingimuse $p \wedge t$ kehtimisest korduse keha täitmise eel järeldub invariandi p kehtimine korduse keha täitmise järel;
- korduse järeltingimuse kehtimine: on vaja näidata, et korduslause töö lõppedes kehtivast vahetingimusest $p \wedge \neg t$ järeldub algoritmi järgmise sammu tööks vajalik eeltingimus.

Massiivi summeerimise näite jaoks järeldub kõigi kolme nõutud tingimuse täidetus vahetult eelmise lõigu arutelust, millega me invariandi ja korduse tingimuse valisime.

3.1.3 Korduse täielik õigsus

Erinevalt teistest senivaadeldud juhtkonstruktsioonidest (järjend ja valik) pole korduse kasutamisel alati garanteeritud selle töö lõppemine.

Näiteks massiivi summeerimiseks leitud korduslause kehast mõlema omistamise eemaldamisel (korduse kehaks jääb tühi käsk) saame korduse, mis rahuldab küll kõiki osalise õigsuse nõudeid, kuid ei lõpeta tööd, kui $n > 0$.

Algoritmi osalise õigsuse mõiste tähendabki seda, et algoritm annab töö lõppedes alati õige vastuse, kuid võib mõnede sisendandmete korral lõpmatuseni tööle jääda.

Algoritmi **täieliku õigsuse** (ingl *full correctness*) tõestamiseks on vaja tõestada

- algoritmi osaline õigsus: on vaja näidata, et kui see algoritm oma töö lõpetab, on ta saavutanud nõutud tulemuse (formaalselt väljendub see järeltingimuse kehtimises);
- algoritmi peatumine: on vaja näidata, et algoritm lõpetab oma töö mistahes korrektsete (eeltingimust rahuldavate) sisendandmete korral.

Tasub märkida, et esimeses peatükis toodud üldkujuga algoritmi õigsuse tõestus vastab algoritmi osalisele õigsusele. Algoritmi täieliku õigsuse tõestamiseks on lisaks vaja näidata, et algoritmi iga samm lõpetab oma töö.

3.1.4 Korduse variant

Muidugi võib kordusega algoritmi täieliku õigsuse tõestamisel korduse keha täitmise arvu lõplikkust näidata ükskõik millise vettpidava arutluse abil. Aga et tegemist on sageli lahendatava ülesandega, siis on selle lahendamiseks välja mõeldud tüüptehnika.

Korduse variandiks (ingl *loop variant*) nimetatakse mittenegatiivset täisarvulist suurust τ, millel on järgmised omadused

- τ väheneb korduse keha igal täitmisel vähemalt 1 võrra;
- kui τ väärtus on 0, siis korduse täitmine lõpetatakse.

Need kaks omadust koos tagavad, et τ on korduse keha täitmiste arvu ülemine tõke: see tähendab, et τ väärtus korduse keha täitmise eel on alati vähemalt sama suur kui selle korduse keha veel teha jäänud täitmiste arv; seetõttu nimetatakse teda ka **tõkkefunktsiooniks** (ingl *bound function*).

Sellise τ olemasolu tagab alati, et korduse täitmine lõpeb teatava arvu sammude järel, sest pole võimalik koostada lõpmatut kahanevat positiivsete täisarvuliste liikmetega jada.

Näiteks massiivi summeerimise korduse variandiks sobib $\tau = n - i$:

- kuna n väärtus korduse täitmisel ei muutu ja i väärtus suureneb korduse keha igal täitmisel 1 võrra, peab τ väärtus samal ajal 1 võrra vähenema;
- kui $\tau = 0$, siis $i = n$ ja korduse täitmine lõpetataksegi.

Mitte alati ei ole võimalik korduse varianti leida. Mõnikord tähendab see, et kordus on tõesti vigane (ja selle algoritmi järgi kirjutatud programm võib "rippuma" jääda); mõnikord tähendab see, et kordus on lihtsalt variandi leidmiseks sobimatu kujuga ja selle täieliku õigsuse tõestamiseks tuleb kasutada muid vahendeid.

Mõnikord võib üsna lihtsa korduse täieliku õigsuse tõestamine üllatavalt raskeks pähkliks osutuda. Näiteks pole mitmete väljapaistvate matemaatikute katsed korduse

```
— eeltingimus n > 0
senikui n > 1
  kui n/2 ∈ ℕ
    n ← n/2
  muidu
    n ← 3n + 1
  lõppkui
lõppsenikui
```

täielikku õigsust tõestada või ümber lükata siiani vilja kandnud. Kordus on n kõigi seni uuritud väärtuste korral oma töö lõpetanud, kuid pole kindel, et ta seda alati teeb.

Matemaatilises mõttes sobiks selle ülesande (enim tuntud $3n+1$ ülesande ja Ulami ülesande nime all) lahenduseks nii tõestus, et see kordus lõpetab oma töö n mistahes positiivse algväärtuse korral kui ka tõestus, et n mingi väärtuse korral jääb see kordus tõesti lõpmatuseni tööle, programmeerimise mõttes näeks me muidugi parema meelega, et lahendus oleks positiivne: et õnnestuks tõestada algoritmi täielik õigsus.

3.1.5 Terviklik näide

Naturaalarvu n faktoriaal (mida tähistatakse $n!$) defineeritakse järgmiselt:

$$n! = \begin{cases} 1, & \text{kui } n = 0, \\ 1 \cdot 2 \cdot \ldots \cdot n, & \text{kui } n > 0. \end{cases}$$

Antud arvu faktoriaali arvutamisel pole muidugi erilist vahet, kas korrutada tegureid $n!$ avaldises kokku vasakult paremale või paremalt vasakule.

Otsustame näiteks paremalt vasakule korrutamise kasuks ja tähistame abimuutuja i abil järgmisena korrutisse lisatava teguri.

Selliste tähistuste korral on meil korduse iga läbimise alguses korrutisse lisamata tegurid $1 \ldots i$. Pärast selle tähelepaneku tegemist on juba lihtne välja kirjutada kogu kordus koos kõigi oluliste vahetingimustega:

— eeltingimus $n \geq 0$
$f \leftarrow 1$
$i \leftarrow n$
— vahetingimus: $f = n!/i!$
`senikui` $i > 0$
  — vahetingimus: $i > 0 \wedge f = n!/i!$
  $f \leftarrow i * f$
  — vahetingimus: $f = n!/i! \cdot i = n!/(i-1)!$
  $i \leftarrow i - 1$
  — vahetingimus: $f = n!/i!$
`lõppsenikui`
— vahetingimus: $i = 0 \wedge f = n!/i!$

Algoritmi osalises õigsuses veendumiseks paneme tähele, et

- korduse algatamine on korrektne: kui $i = n$, siis $n!/i! = n!/n! = 1 = f$;

- korduse samm säilitab invarianti vastavalt selles välja toodud vahetingimustele;
- korduse lõpetamine on korrektne: kui $i = 0$, siis $n!/i! = n!/1 = n!$ ja seega viimasest vahetingimusest järeldubki, et $f = n!$.

Algoritmi täielikus õigsuses veendumiseks paneme lisaks tähele, et tõkkefunktsiooniks sobib väga hästi i väärtus ise.

3.2 Kordusskeemide liigid

Programmikeeltes on olemas mitmeid veidi erineva ülesehitusega korduslauseid, mille sobivuse ühes või teises olukorras määrab üldiselt mugavus.

3.2.1 Loendajaga kordus

Loendajaga korduse üldkuju on

```
korda i ← a...b
  käsud
lõppkorda
```

ja see on samaväärne kordusega

```
i ← a
senikui i ≤ b
  käsud
  i ← i + 1
lõppsenikui
```

Loendajaga korduse õigsuse tõestamiseks on üks võimalus esitada see just eeltoodud kujul ja rakendada siis lõikudes 3.1.2 ja 3.1.3 vaadeldud tehnikaid.

Loendajaga kordust on mugav kasutada, kui korduse keha täitmiste arv on teada kohe korduslauseni jõudmisel. Näiteks on arvumassiivi summeerimise algoritmi loendajaga kordust kasutav versioon veidi ülevaatlikum:

```
s ← 0
korda i ← 1...n
  — vahetingimus: s = a_1 + ... + a_{i-1}
  s ← s + a_i
  — vahetingimus: s = a_1 + ... + a_i
lõppkorda
```

Mõnes keeles (näiteks Pascalis) ei ole lubatud korduse juhtmuutuja i väärtuse muutmine korduslause kehas — sellise lubamatu operatsiooni tagajärjel võib programmi käitumine muutuda ettearvamatuks. Mõnes keeles (näiteks Basicus) on loendajaga korduse süntaksis olemas võimalus näidata loendaja suurendamise samm.

3.2.2 Eelkontrolliga kordus

Eelkontrolliga korduslause üldkuju on

```
senikui t
  käsud
lõppsenikui
```

Lõigus 3.1 vaatlesime korduslauseid just eelkontrolliga korduslause näitel, seega pole selle skeemi tähendust enam vaja selgitada.

Eelkontrolliga lauset on hea kasutada, kui korduse keha täitmise kordade arv pole ette teada, vaid lõpetamise vajadus selgub alles siis, kui korduse keha on juba piisavalt täidetud.

Praktikas väga levinud näide on failist tekstiridade lugemine — tekstifaili avamisel pole üldiselt teada, mitu rida failis on, seega pole praktiliselt muud võimalust kui iga rea lugemisel kontrollida, kas fail sai läbi või ei saanud.[1]

3.2.3 Järelkontrolliga kordus

Nagu nimestki arvata võib, erineb järelkontrolliga korduslause eelkontrolliga lausest selle poolest, et tingimuse täidetust kontrollitakse mitte enne, vaid pärast korduse keha täitmist.

Järelkontrolliga korduslause esineb levinud programmikeeltes kahel erineval kujul. Neist kahest kujust esimene,

```
korda
  käsud
senikui t
```

on samaväärne eelkontrolli kasutava konstruktsiooniga

[1] Väärib eraldi märkimist, et ka faili lõppemise kontroll on erinevates süsteemides põhimõtteliselt erinevalt lahendatud. Näiteks Pascali ja Basicu funktsioonid annavad ette teada, kui järgmine katse failist midagi lugeda ebaõnnestuks faili lõppemise tõttu; C ja C++ funktsioonid seevastu annavad alles tagantjärele teada, kui viimane juba tehtud katse failist lugeda sel põhjusel ebaõnnestus.

käsud
`senikui` *t*
käsud
`lõppsenikui`

ning teine,

`korda`
käsud
`kuni` *t*

on samaväärne konstruktsiooniga

käsud
`senikui` ¬*t*
käsud
`lõppsenikui`

Nagu näha, täidetakse järelkontrolliga korduslauses korduse kehaks olevaid käske alati vähemalt üks kord ja selle korduslause kahe variandi ainuke erinevus on see, et ühel juhul näidatakse korduslause tingimusena korduse täitmise jätkamis- teisel juhul aga lõpetamistingimus.

Kahe vastandliku variandi kasutamine on tingitud inglise keele eripärast[2], kummagi olemasolu või puudumine on põhiliselt konkreetse programmikeele autori(te) maitse küsimus.

Üldiselt kasutatakse seda liiki kordust praktikas harvemini kui loendajaga või eelkontrolliga kordust.

3.2.4 Muud kordusskeemid

Tingimuseta kordus, mille üldkuju on

`korda`
käsud
`lõppkorda`

töötab alati lõpmatuseni ja pole sellisel kujul ilmselt eriti praktiline.

Seda liiki kordust kasutatakse (keeltes, kus see üldse olemas on) alati koos katkestusega kujul

[2]Erinevalt eesti keelest, kus tsüklilise tegevuse märkimiseks kasutatakse põhiliselt ühte lausekonstruktsiooni "korda..., kuni...", on inglise keeles neid kaks: "repeat... while..." ja "repeat... until...", kusjuures esimeses neist on "while" järel jätkamistingimus, teises aga "until" järel lõpetamistingimus.

```
korda
   eel-käsud
   kui t
      katkesta
   lõppkui
   järel-käsud
lõppkorda
```

Keeltes, kus katkestus olemas on, saab seda tavaliselt kasutada ka teiste korduste täitmise "ennetähtaegseks" lõpetamiseks. Keeltes, kus katkestus puudub, saab selle asendada abimuutujaga ja eel- või järelkontrolliga. Näiteks eeltoodud kordus on samaväärne skeemiga

```
l ← väär
senikui ¬l
   eel-käsud
   l ← t
   kui ¬l
      järel-käsud
   lõppkui
lõppkorda
```

3.3 Korduste pesastamine

Kuigi teoreetiliselt on iga algoritm võimalik esitada ülimalt ühe korduslause abil, on praktikas sageli mugavam kasutada mitut kordust.

Vaatleme näiteks järjestamisülesannet: antud on n-elemendiline arvumassiiv $a_{1\ldots n}$; järjestada selle elemendid mittekahanevalt (see tähendab nii, et $a_1 \leq a_2 \leq \ldots \leq a_n$).

Tegemist on klassikalise ülesandega, mille lahendamiseks on väga palju erinevaid algoritme, siinkohal vaatleme neist üht lihtsamat, mida tuntakse **pistemeetodi** (ingl *insertion sort*) nime all.

Meetodi idee on hoida massiivi algusosa elemente omavahel järjestatult ja kasvatada seda järjestatud osa, pistes veel järjestamata elemente nende õigetele kohtadele juba järjestatud elementide vahele.

Algoritmi põhiosaks on seega kordus, mille igal sammul kehtib invariant $a_1 \leq a_2 \leq \ldots \leq a_{i-1}$ mingi i jaoks. Kuna tühi jada on alati järjestatud, kehtib invariant $i = 1$ jaoks triviaalselt ja korduse algatamiseks polegi vaja midagi teha. Kui invariant kehtib $i = n + 1$ jaoks, on kogu massiiv sobivalt järjestatud ja korduse võib lõpetada.

Otsitava algoritmi üldkuju on seega

```
korda i ← 1...n
    — invariant: a_1 ≤ a_2 ≤ ... ≤ a_{i−1}
    korduse keha
lõppkorda
```

ja jääb vaid üle kirjutada korduse keha nii, et selle igal täitmisel suureneb i väärtus, mille jaoks invariant kehtib.

Selleks kasutame korduse kehana omakorda kordust. Vaatleme esimest veel järjestamata elementi a_i ja võrdleme teda oma vasakpoolse naabriga: kui see on suurem kui a_i, vahetame nad omavahel ja jätkame vaadeldava elemendi (nüüd juba a_{i-1}) võrdlemist oma uue vasakpoolse naabriga jne.

Seega on sisemise korduse invariant $a_j \leq a_{j+1} \leq \ldots \leq a_i$. See tingimus kehtib $j = i$ jaoks triviaalselt järelikult pole ka selle korduse algatamiseks peale j algväärtustamise midagi muud teha. Kui invariant kehtib $j = 1$ jaoks, olemegi saavutanud välimise korduse invariandi senisest ühe võrra suurema i jaoks ja võime lõpetada.

Järgneva algoritmi täielikus õigsuses veendumiseks paneme veel tähele, et mõlemad kordused lõpetavad kindlasti oma töö. Välimise korduse variandiks sobib $\tau_1 = (n + 1) - i$ ja sisemise omaks $\tau_2 = j - 1$.

Algoritm 3.1 Järjestab antud massiivi elemendid mittekahanevalt
Sisend: $a_{1\ldots n}$ — sorteeritav massiiv
Väljund: $a_{1\ldots n}$ — väärtused vahetatud nii, et $\forall i < n : a_i \leq a_{i+1}$
Abimuutujad: i, j

```
1.  korda i ← 1...n
        — invariant: a_1 ≤ a_2 ≤ ... ≤ a_{i−1}
2.      j ← i
3.      senikui j > 1
            — invariant: a_j ≤ a_{j+1} ≤ ... ≤ a_i
4.          kui a_{j−1} > a_j
5.              a_{j−1} ↔ a_j
6.              j ← j − 1
7.          muidu
8.              j ← 1
9.          lõppkui
10.     lõppsenikui
11. lõppkorda
```

(siin ja edaspidi kasutame kahe muutuja väärtuste vahetamiseks lühiduse huvides süntaksit $a \leftrightarrow b$).

Ülesanded

Ülesanne 3.1 *Kirjutada kordus, mis leiab summa*

$$1 + 1/2 + 1/3 + \ldots + 1/n.$$

Kirjutada välja selle korduse eel- ja järeltingimus, invariant ning variant. Tõestada selle osaline ja täielik õigsus.

Ülesanne 3.2 *Kirjutada kordus, mis leiab antud arvumassiivi maksimaalse väärtuse. Tõestada selle õigsus.*

Ülesanne 3.3 *Kirjutada kordus, mis leiab antud arvumassiivi maksimaalse väärtusega elemendi indeksi. Tõestada selle õigsus. Millise indeksi väljastab see kordus, kui massiivis on mitu maksimaalse väärtusega elementi?*

Ülesanne 3.4 *Kirjutada kordus, mis leiab antud arvumassiivist suuruselt teise väärtuse. Tõestada selle õigsus. Kontrollida, et programm töötab õigesti ka siis, kui massiivis on mitu maksimaalse väärtusega elementi.*

Ülesanne 3.5 *Kirjutada kordus, mis summeerib antud arvumassiivi need elemendid, millele eelnev element on positiivne. Tõestada selle õigsus.*

Ülesanne 3.6 *Kirjutada kordus, mis summeerib antud arvumassiivi positiivsed elemendid, millele eelnev ja järgnev element on negatiivsed. Tõestada selle õigsus.*

Ülesanne 3.7 *Kirjutada kordus, mis leiab summa*

$$1 + 1/2! + 1/3! + \ldots + 1/n!.$$

Tõestada selle õigsus. (Mitte kasutada kahte pesastatud kordust.)

Ülesanne 3.8 *Vaatleme lõigus 3.2.4 toodud skeemi, mis kasutab katkestusega korduse simuleerimiseks eelkontrolliga kordust ja abimuutujat. Koostada analoogiline skeem, mis kasutab katkestusega korduse simuleerimiseks järelkontrolliga kordust ja abimuutujat. Põhjendada, et teie skeem on samaväärne lõigus 3.2.4 toodud skeemiga.*

Ülesanne 3.9 *Koostada arvumassiivi elementide järjestamiseks algoritm, mis suurendab massiivi alguses olevat järjestatud osa sel teel, et leiab veel järjestamata elementide hulgast minimaalse ja paigutab selle järjestatud osa lõppu. Tõestada selle algoritmi õigsus.*

Peatükk 4

Alamprogrammid

Sisukord

Põhimõistete peatükis vaatlesime liittüüpe kui suurte andmemahtude korrastamise vahendit. Käesolevas peatükis tutvume alamprogrammidega, mida võib analoogiliselt vaadelda kui suurte koodimahtude struktureerimise ja organiseerimise vahendit.

4.1 Alamprogrammi mõiste

Alamprogrammiks (ingl *subroutine*) nimetatakse programmilõiku, mis on mõeldud korduvaks kasutamiseks programmi teiste osade poolt. Tavaliselt sooritab alamprogramm ühte suhteliselt terviklikku tegevust, mille määravad alamprogrammi eel- ja järeltingimus.

Alamprogrammi kasutamine koosneb kaht liiki konstruktsioonidest:

- ühekordsest alamprogrammi kirjeldusest;
- ühe- või mitmekordsest alamprogrammi väljakutsest.

Alamprogrammi kirjeldus seob alamprogrammi nime (madalkeeles aadressi) ja selle sisuks olevad tegevused, kuid ei soorita neid tegevusi. Alamprogrammi kirjeldus kuulub seega deklaratsioonide hulka.

Alamprogrammi väljakutse on juhtkonstruktsioon, mis nõuab, et väljakutse kohas tuleb sooritada eelnevas kirjelduses selle alamprogrammi sisuks märgitud tegevused.

Esimeses lähenduses (mis tegelikult pole päris õige) võib kujutleda, et translaator asendab alamprogrammi väljakutse lause selle alamprogrammi sisuks olevate lausetega.

Alamprogrammid pole sugugi programmeerijate leiutatud. Näiteks võib kokaraamatus esineda tegevusjuhis kujul "...piruka jaoks tuleb kõigepealt valmistada pärmitaigen (vt põhiretsept lk X)...". Tegemist on tüüpilise alamprogrammi väljakutsega ja leheküljel X olev pärmitaigna retsept on selle alamprogrammi kirjeldus.

4.1.1 Voorused

Alamprogrammide kasutamisel on mitmeid voorusi:

- programm on loetavam — kui igal alamprogrammil on selle sisule vastav nimi, on põhiprogrammist palju lihtsam aru saada, sest programmi üldise ehitusega tutvumisel ei pea kohe kõiki selle detaile haarama;

- programm on testitavam — kui igal alamprogrammil on selge ülesanne ning eel- ja järeltingimus, on võimalik iga alamprogramm eraldi tõestada või testida enne kui neid omavahel ühtsesse süsteemi ühendama ja siis sellest süsteemist kui tervikust vigu otsima hakata;

- programm on hallatavam — kui sama tegevuse korduvaks sooritamiseks tuleks vastavat programmiosa tekstina mitmesse kohta kopeerida, peaks hiljem sellest osast vigade avastamisel neid vigu igas koopias eraldi parandama; see on asjata lisatöö, pealegi tekib oht, et mõni koopia jääb parandamisel vahele.

4.1.2 Metoodika

Alamprogramme sisaldava programmi kirjutamisele võib läheneda vähemalt kahel põhimõtteliselt erineval viisil.

Ülalt alla arendus

Ülalt alla (ingl *top-down*) arenduse korral jagatakse lahendatav ülesanne alamülesanneteks ja kirjeldatakse tervikülesande lahendus alamlahenduste kombinatsioonina.

Sellise lähenemise korral algab struktuurse programmi kirjutamine põhiprogrammist, milles alamülesannete lahendamise sammud vormistatakse (sel hetkel veel puuduvate) alamprogrammide väljakutsetena ja hiljem lisatakse vajalike alamprogrammide kirjeldused (võimalik, et need tekitavad omakorda vajaduse uute veel madalama taseme alamprogrammide järele).

Sellist järjest detailsema taseme alamprogrammide kirjutamist jätkatakse seni, kuni kõigi alamülesannete lahendused on välja kirjutatud kasutatava programmikeele primitiivide täpsuseni.

Ülalt alla arenduse üks olulisemaid puudujääke on see, et programmi kõiki osi ei saa kohe nende kirjutamise järel testida — kui põhiprogrammi kasutatavad alamprogrammid on veel kirjutamata, ei saa põhiprogrammi käima panna.

Alt üles arendus

Alt üles (ingl *bottom-up*) arenduse korral vaadeldakse alamprogrammide kirjutamist kui kasutatava programmikeele primitiivide hulga laiendamist ülesande lahendamiseks sobivas suunas. Sellist lähenemist õigustab ka asjaolu, et sageli on programmikeele standardsed primitiivid realiseeritud just alamprogrammidena.

Alt üles arenduse kõige suurem raskus on vajadus prognoosida, milliseid uusi primitiive ülesande lahendamiseks tegelikult vaja läheb. Ilma sellise prognoosita on oht kulutada palju aega sellele, et kirjutada alamprogramme, mida lõpuks üldse vaja ei lähe ja unustada esimesel lähenemisel kirjutada mõned, mis hiljem vajalikuks osutuvad.

Kombineeritud arendus

Kahe eelpool kirjeldatud "puhta" arendusmetoodika puudujääkide leevendamiseks jagatakse programmi kirjutamine tavaliselt kaheks etapiks.

Projekteerimise etapil lähenetakse ülesande lahendamisele ülalt alla meetodil ja jagatakse selle lahendus ilma programmi kirjutamata (ja sageli isegi konkreetset programmikeelt valimata) alamülesanneteks eesmärgiga saada teada, milliseid alamprogramme on selle ülesande lahendamiseks vaja ja kuidas need alamprogrammid omavahel seotud olema peaks.

Realiseerimise etapil hakatakse vajalikke alamprogramme kirjutama alt üles meetodil, sest nii on iga alamprogrammi valmimise ajaks olemas ka kõik need, mida ta oma töös kasutab ja seetõttu on võimalik iga alamprogrammi kohe pärast tema kirjutamist testida.

Illustratsiooniks koostame lihtsa algoritmi kõigi algarvude leidmiseks etteantud lõigust $L = a \ldots b$.[1]

Projekteerimise etapil paneme tähele, et kõige ilmsem lahendus on kontrollida iga lõiku L jääva arvu kohta, kas ta on algarv, ja väljastada need arvud, mille korral kontrollimine annab positiivse tulemuse.

Kuna üldotstarbelistes programmikeeltes enamasti algarvulisuse kontrollimise primitiivi pole, peame selle ise realiseerima. Kõige lihtsam lahendus on proovida kontrollitavat arvu kõigi temast väiksemate ja ühest suuremate täisarvudega jagada; täpse jagumise korral on tegemist kordarvuga, kui aga jagamine ühegi arvuga ei õnnestu, on tegemist algarvuga.

Realiseerimise etapil alustame altpoolt, algarvulisuse kontrollimise alamprogrammist. Kohe pärast selle alamprogrammi kirjutamist saame tõestada tema aluseks oleva algoritmi õigsuse ja testida selle realiseerimise korrektsust. Alles siis, kui algarvulisuse kontrollimise alamprogramm on valmis ja kontrollitud, liigume edasi põhiprogrammi juurde.

[1] Olgu kohe märgitud, et see algoritm on oma lihtsuse tõttu ka silmatorkavalt ebaefektiivne, algarve saab leida märksa väiksema hulga arvutustega. Kuna efektiivsemad algoritmid on ka keerukamad, pole nende uurimine siinkohal põhjendatud, näite eesmärk on illustreerida alamprogrammide kasutamist.

4.1.3 Protseduur ja funktsioon

Enamik tänapäevaseid programmikeeli eristavad protseduure ja funktsioone.

Funktsiooniks (ingl *function*) nimetatakse alamprogrammi, mis tagastab oma töö tulemusena mingi väärtuse. Funktsioonil on tüüp — funktsioon tagastab ainult sellesse tüüpi kuuluvaid väärtusi — ja funktsiooni väljakutset võib kasutada avaldises seda tüüpi operandina.

Protseduuriks (ingl *procedure*) nimetatakse alamprogrammi, mis ei ole funktsioon. Protseduur ei tagasta otseselt mingit väärtust ja seetõttu ei saa protseduuri väljakutset operandina kasutada.

Selle erinevuse illustreerimiseks vaatleme näiteks paljudes programmikeeltes protseduurina realiseeritud primitiivi antud sõne väljastamiseks ja peaaegu kõigis keeltes funktsioonina realiseeritud primitiivi antud reaalarvu siinuse arvutamiseks.

Kuna sõne väljastamise primitiiv on protseduur, võib selle väljakutset kasutada lihtlausena, näiteks

```
väljasta 'Tere, kasutaja'
```

Kuna siinuse arvutamine on funktsioon, võib selle väljakutset kasutada operandina avaldises, näiteks

$$z \leftarrow sin(x) + sin(y) + 1$$

Edaspidises kasutame alamprogrammide kirjeldamiseks esimeses peatükis kasutusele võetud pseudokeelt, lisades igale algoritmile nime, mille abil on võimalik sellele hiljem mujalt viidata.

Näiteks protseduuri 1 ja 100 vahele jäävate algarvude väljastamiseks võiks esitada algoritmis 4.1 toodud kujul, kus OnAlgarv tähistab (veel kirjutamata) funktsiooni muutuja a väärtuse algarvulisuse kontrollimiseks.

Algoritm 4.1 AlgarvudSajani: Väljastab algarvud lõigust $1 \ldots 100$
Abimuutujad: $a \in \mathbb{N}$ — jooksev algarvukandidaat

1. `korda` $a \leftarrow 1 \ldots 100$
2. `kui` OnAlgarv
3. `väljasta` a
4. `lõppkui`
5. `lõppkorda`

Funktsiooni väärtuse põhiprogrammile tagastamise käsu süntaks on erinevates programmikeeltes erinev, oma pseudokeeles kasutame edaspidi lauset kujul

`tagasta` v

kus v on funktsiooni väljakutse väärtus põhiprogrammi avaldises.

Funktsiooni muutuja a jooksva väärtuse algarvulisuse kontrollimiseks võime seega esitada nii, nagu näha algoritmis 4.2.

Algoritm 4.2 ONALGARV: Kontrollib, kas muutuja a väärtus on algarv
Sisend: $a \in \mathbb{N}$ — uuritav väärtus
Väljund: *tõene*, kui a on algarv, *väär* vastasel juhul
Abimuutujad: $j \in \mathbb{N}$ — jooksev jagajakandidaat

1. `kui` $a < 2$
   — vähim algarv on 2
2. `tagasta` *väär*
3. `lõppkui`
4. `korda` $j \leftarrow 2 \ldots a - 1$
5. `kui` $a/j \in \mathbb{N}$
   — leidsime jagaja, järelikult kordarv
6. `tagasta` *väär*
7. `lõppkui`
8. `lõppkorda`
   — ei leidnud jagajat, järelikult algarv
9. `tagasta` *tõene*

4.1.4 Deklaratsioon ja definitsioon

Paljudes keeltes eristatakse alamprogrammi deklaratsiooni ja definitsiooni.

Alamprogrammi **deklaratsioon** (ingl *declaration*) fikseerib selle välise liidese — nime, parameetrite arvu ja nende tüübid ning funktsiooni korral ka tagastatava väärtuse tüübi —, kuid ei sisalda alamprogrammi sisuks olevaid käske; **definitsioon** (ingl *definition*) sisaldab ka alamprogrammi sisu, kuid mõnes keeles pole deklaratsiooni olemasolu korral parameetrite kirjelduse kordamine enam vajalik.

Neid mõisteid eristatakse tavaliselt üsna tehnilistel (ja erinevates keeltes veidi erinevatel) põhjustel, seetõttu me neil pikemalt ei peatu.[2]

[2]Pascalis vormistatakse deklaratsioonid `forward`-lause abil; QBasicu keskkonnas varjab keskkond need programmeerija eest ära; C's ja C++'is nimetatakse neid **prototüüpideks** (ingl *prototype*); Javas pole neid üldse.

4.2 Alamprogrammi lokaalsed muutujad

Suures programmis, mis koosneb paljudest alamprogrammidest, võib kergesti juhtuda, et mitme alamprogrammi autorid tahavad kasutada samade nimedega abimuutujaid. See on aga ohtlik, sest sellisel juhul võib ühe alamprogrammi kasutamine rikkuda teise tööks vajalikud andmed.[3]

Ohu vähendamiseks (ja ka muudel põhjustel) võimaldavad tänapäevased programmikeeled kasutada mitme erineva skoobi ja elueaga muutujaid.

4.2.1 Muutuja eluiga

Muutuja **elueaks** (ingl *lifetime*) nimetatakse programmi osa, mille täitmise ajal see muutuja oma väärtust säilitab.

Kuna muutuja väärtust hoitakse mälus, siis peab ta oma eluea vältel mingit mäluosa oma valduses hoidma.

Eluea järgi võib muutujad jagada kolme liiki:

- **staatilise** (ingl *static*) muutuja eluiga on kogu programmi täitmise aeg; talle eraldatakse mälu programmi käivitamise hetkel ja see mälu jääb tema kasutusse kuni programmi täitmise lõpuni; muutujale kord omistatud väärtus püsib alles kuni järgmise omistamiseni;

- **automaatse** (ingl *automatic*) muutuja eluiga on tavaliselt mingi alamprogrammi ühe täitmise aeg; automaatne muutuja luuakse (talle eraldatakse mälu) alamprogrammi täitmise alguses ja ta hävitatakse (talle eraldatud mälu vabastatakse) selle täitmise lõppedes; alamprogrammi järgmisel täitmisel võidakse selle muutuja hoidmiseks kasutada hoopis teist mälupesa, seega automaatse muutuja väärtus alamprogrammi kahe täitmiskorra vahel ei säili;

- **dünaamiline** (ingl *dynamic*) muutuja luuakse ja hävitatakse programmeerija otsese käsu peale, seega on dünaamilise muutuja eluiga täielikult programmeerija määrata.[4]

4.2.2 Muutuja skoop

Muutuja **skoobiks** (ingl *scope*) nimetatakse programmi osa, milles selle muutuja nimi nähtav on.

[3] Kui muutujate väärtused programmeerija selja taga iseenesest muutuma hakkavad, ei kehti enam ka algoritmide õigsuse tõestused!

[4] Mõnes keeles on ka dünaamiliste muutujate hävitamine automaatne — süsteem hävitab ise need muutujad, mille väärtust ei ole enam võimalik kasutada. Seda nimetatakse **prahikoristusega** (ingl *garbage collection*) mäluhalduseks.

Muutuja nimega pole suurt midagi peale hakata, kui sellele ei vasta mälupesa, milles muutuja väärtust hoida. Sellepärast ongi muutuja skoop tavaliselt tema eluea alamhulk.

Skoobi järgi võib muutujad jagada kahte liiki:

- **globaalseks** (ingl *global*) nimetatakse muutujat, mis on kirjeldatud väljaspool kõiki alamprogramme;

  Selline muutuja on üldiselt nähtav kõigile programmi osadele, seega on alati oht, et keegi teine võib selle väärtust muuta. Sellepärast tuleks globaalseid muutujaid kasutada nii vähe kui võimalik.

  Globaalsed muutujad on tavaliselt staatilised.

- **lokaalseks** (ingl *local*) nimetatakse muutujat, mis on kirjeldatud mingi alamprogrammi sees;

  Selline muutuja ei ole sama programmi teistele osadele otse kättesaadav ja seega ei saa keegi kogemata seda muutujat kasutades tema väärtust rikkuda.

  Tavaliselt on lokaalsed muutujad automaatsed, aga mõned programmikeeled võimaldavad luua ka staatilisi lokaalseid muutujaid.[5]

  Staatiliste lokaalsete muutujate abil saab kirjutada alamprogramme, mis uuel kasutamisel "mäletavad", mida nad eelmisel korral tegid. See võib olla kasulik näiteks siis, kui alamprogrammi esimesel käivitamisel arvutatakse välja mingid abitulemused, mida läheb vaja ka järgmistel käivitamiskordadel.

4.3 Alamprogrammi parameetrid

Alamprogrammide kasutamine oleks üsna tüütu, kui vastaks tõele peatüki alguses antud lihtsustatud kujutlus, mille kohaselt translaator asendab alamprogrammi väljakutse lause mehhaaniliselt selle alamprogrammi kirjelduses olevate lausetega.

Oletame, et meil on vaja kirjutada alamprogramm, mis väljastab mistahes arvu ruudu. Kui alamprogrammi väljakutse oleks tõesti vaid mehhaaniline tekstiasendus, oleks meil vaja kirjutada eraldi alamprogrammid kõigi võimalike arvude ruutude väljastamiseks või luua ruutu tõstetava arvu edastamiseks globaalne muutuja ja omistada enne alamprogrammi väljakutsumist

[5] Õppematerjalile lisatud näiteprogrammid kasutavad ainult automaatseid lokaalseid muutujaid.

sellele muutujale vajalik väärtus (sisuliselt sama võtet kasutasime muutuja a väärtuse edastamisel algarvude arvutamise programmis).

Et kumbki variant pole kasutamiseks kuigi mugav — esimene neist on ilmselt võimatu, teise puhul hakkaks suuremates programmides (milles on palju alamprogramme ja neil omakorda palju parameetreid) tekkima massiliselt globaalseid muutujaid —, lubavad kõik tänapäevased programmikeeled kasutada alamprogrammides parameetreid.

4.3.1 Formaalsed ja tegelikud parameetrid

Parameetrite alamprogrammile edastamise aparaat koosneb formaalsetest ja tegelikest parameetritest.

Formaalsed parameetrid (ingl *formal parameter*) on alamprogrammi definitsioonis kasutusel selleks, et anda parameetritele nimed, millega on võimalik alamprogrammi sisuks olevates käskudes osutada, millal ja kuidas alamprogramm talle parameetritena antavaid väärtusi kasutab. Alamprogrammi väljakutsel sellele töötlemiseks antavaid väärtusi nimetatakse **tegelikeks parameetriteks** (ingl *actual parameter*).

Formaalse parameetri mõiste illustreerimiseks oletame, et üks sõber kingib teisele loteriipileti ja küsib "mida Sa teed, kui võidad?". Kui kingi saaja vastab midagi stiilis "kui võit on väike, ostan maiustusi, aga kui suur, siis peab mõtlema", on ta sellega defineerinud parameetriga algoritmi, mis kirjeldab tema käitumist võidu korral vastavalt võidu suurusele. Võidu suurus on selle algoritmi formaalne parameeter — sest tegelikult selle plaani järgi tegutsema hakata ei saa muidugi enne kui võit tõesti käes on.

Olukord on veelgi sarnasem matemaatikatunnist tuntud funktsioonidega. Funktsiooni f kirjelduses kujul $f(x) = 2x+1$ on x formaalne parameeter, mida kasutatakse selleks, et avaldises $2x+1$ näidata, kuidas f oma parameetrit kasutab. Avaldises $f(3) + f(4)$ on arvud 3 ja 4 aga funktsiooni f tegelikud parameetrid selle kahel kasutamisel, kusjuures $f(3) + f(4)$ tähistab muidugi avaldist $(2 \cdot 3 + 1) + (2 \cdot 4 + 1)$.

Mitme parameetriga alamprogrammides seatakse tegelikke ja formaalseid parameetreid tavaliselt vastavusse samamoodi kui mitme muutuja funktsioonides matemaatikas — alamprogrammi väljakutsel esitatakse kõik tegelikud parameetrid järjest ja selles järjestuses esimene tegelik parameeter loetakse alamprogrammi esimese formaalse parameetri väärtuseks, teine tegelik parameeter teise formaalse parameetri väärtuseks jne.[6]

[6]Kuigi selline parameetrite sidumise viis (kohtomistus) on levinuim, kasutatakse vahel ka muid skeeme. Näiteks nimiomistuse korral näidatakse alamprogrammi väljakutses iga tegeliku parameetri juures ka sellele vastava formaalse parameetri nimi. Käesoleva õppematerjali lisades esinevad keeled kasutavad kohtomistust.

Lisaks formaalsete parameetrite nimedele nõuavad paljud keeled ka nende tüüpide kirjeldamist. See võimaldab translaatoril kontrollida, et parameetritena edastatavate väärtuste kasutamine vastab nende väärtuste tüüpidele.[7] Programmi transleerimisel toimub parameetrite tüübikorrektsuse kontroll tavaliselt kahe sammuna: alamprogrammi definitsiooni transleerimisel kontrollib translaator, et parameetrite kasutamine alamprogrammi sisuks olevates käskudes on kooskõlas formaalsete parameetrite deklareeritud tüüpidega — iga parameetriga sooritatavad operatsioonid peavad kõik kuuluma selle parameetri tüübi operatsioonivarusse; alamprogrammi väljakutse transleerimisel kontrollib translaator, et tegelike parameetritena antud väärtuste tüübid vastavad formaalsete parameetrite deklareeritud tüüpidele.

4.3.2 Sisend- ja väljundparameetrid

Alamprogrammide parameetrid võib nende kasutuseesmärgi alusel jagada sisend- ja väljundparameetriteks.

Sisendparameetrid (ingl *input parameter*), nagu nimestki oletada võib, on mõeldud lähteandmete edastamiseks alamprogrammi väljakutsujalt sellele alamprogrammile.

Näiteks siinusfunktsioonile antav nurga suurus on sisendparameeter. Ka väljastamisprimitiivile (mis on paljudes programmeerimissüsteemides realiseeritud alamprogrammina) väljastamiseks antav suurus on selle primitiivi seisukohalt sisendparameeter — selle kaudu saab primitiiv andmed, mida ta töötlema (antud juhul väljastama) peab.

Väljundparameetrid (ingl *output parameter*) seevastu on mõeldud tulemuste tagastamiseks alamprogrammilt selle väljakutsujale.

Näiteks sisestamisprimitiivile parameetrina antav muutuja on väljundparameeter — primitiiv ei vaja selle muutuja esialgset väärtust (muutuja võib olla ka algväärtustamata), vaid kasutab teda ainult kohana kasutajalt või failist saadud andmete salvestamiseks.

Ka funktsiooni tagastatavat väärtust võib vaadelda väljundparameetrina, kujutledes, et translaator tekitab ajutise muutuja ja kasutab funktsiooni poolt sellesse muutujasse salvestatud tulemust avaldises funktsiooni väljakutse väärtusena.

Sageli on üks parameeter kasutusel nii sisend- kui väljundparameetrina. Näiteks kui arvumassiivi sorteerimise alamprogramm saab töödeldava massiivi ette parameetrina, siis on see kasutusel nii sisendparameetrina (sel-

[7] Kuigi sellest üldhariduskooli matemaatikatunnis ei räägita, on ka matemaatikas funktsioonidel ja nende parameetritel tüübid. Näiteks täisosa arvutamise funktsioon $\lfloor\rfloor$ teisendab reaalarve täisarvudeks, mida matemaatikud tähistavad "$\lfloor\rfloor : \mathbb{R} \to \mathbb{Z}$" ja loevad "täisosa on funktsioon reaalarvude hulgast täisarvude hulka".

le kaudu edastatakse alamprogrammile järjestamist vajavad elemendid) kui ka väljundparameetrina (selle kaudu tagastatakse alamprogrammist massiivi uus sisu, milleks on needsamad elemendid vajalikus järjekorras).

4.3.3 Väärtus- ja muutujaparameetrid

Kuigi eelmises jaotises kirjeldatud parameetrite liigitus on mugav algoritmi koostaja mõtte väljendamiseks, kasutatakse programmikeeltes realiseerimise efektiivsuse huvides veidi teistsugust klassifikatsiooni.

Väärtusparameetri (ingl *call by value*) kasutamisel leiab süsteem alamprogrammi käivitamisel tegeliku parameetrina antud avaldise paremväärtuse ja edastab alamprogrammile selle. Muidugi saab niimoodi kasutada ainult avaldist, millel on paremväärtus olemas.

Alamprogramm võib saadud parameetrit kasutada nagu algväärtustatud lokaalset muutujat. See tähenab, et kui alamprogramm sellele "lokaalsele muutujale" uue väärtuse omistab, kehtib uus väärtus ainult alamprogrammi sees ja ei mõjuta põhiprogrammi avaldist.

Eelnevast järeldub, et väärtusparameetrite mehhanismi on võimalik kasutada ainult alamprogrammide sisendparameetrite realiseerimiseks — kui alamprogrammis olev väärtus on põhiprogrammi avaldisest täielikult lahutatud, pole selle kaudu kuidagi võimalik andmeid alamprogrammist põhiprogrammile tagastada.

Muutujaparameetri (ingl *call by reference*) kasutamisel edastatakse alamprogrammile parameetrina antud avaldise vasakväärtus — tavaliselt on sellise parameetrina kasutusel põhiprogrammi muutuja (millest tuleb ka parameetrite edastamise viisi eestikeelne nimetus), kuigi võib kasutada ka kõiki muid avaldisi, millel on vasakväärtus olemas.

Muutujaparameetrile alamprogrammis omistatud uus väärtus muudab ka põhiprogrammi muutuja väärtuse, seega on muutujaparameetrite mehhanismi võimalik kasutada ka väljundparameetrite realiseerimiseks.

Kuna väärtusparameetrist alamprogrammi jaoks koopia tegemine kulutab nii aega kui mälu, edastatakse praktikas suuremahulisi väärtusi muutujaparameetritena ka siis, kui algoritmi loogika seisukohalt on tegemist alamprogrammi sisendparameetritega. Mõnes keeles on olemas isegi eraldi konstruktsioon väljendamaks, et alamprogramm talle antud muutujaparameetri väärtust ei muuda (see tähendabki, et sisuliselt kasutab alamprogramm seda sisendparameetrina).

Nimiparameetri (ingl *call by name*) kasutamisel edastab süsteem alamprogrammile parameetrina antud avaldise enda, ilma selle väärtust arvutamata. Kui alamprogrammil on avaldise väärtust (ükskõik, kas vasak- või paremväärtust) vaja, peab ta selle ise leidma.

Nimiparameetreid on hea kasutada näiteks siis, kui alamprogramm ei vaja igal käivitamisel kõigi oma parameetrite väärtusi ja mõne väärtuse leidmine võib olla väga töömahukas — sellisel juhul on otstarbekam arvutada parameetri väärtus välja alles siis, kui on kindel, et seda tõesti vaja läheb.

Parameetrite sellist kohtlemist nimetatakse **laisaks väärtustamiseks** (ingl *lazy evaluation*), vastandina **agarale väärtustamisele** (ingl *eager evaluation*). Mõned programmikeeled väärtustavad laisalt kõiki avaldisi, mitte ainult alamprogrammide parameetreid.[8]

4.4 Rekursioon

4.4.1 Rekursiooni mõiste

Alamprogrammide tähtis rakendus on rekursiivsete algoritmide programmeerimine. Algoritmi nimetatakse **rekursiivseks** (ingl *recursive*), kui selles kasutatakse ühe (või ka mitme) sammuna sama algoritmi ennast. Selle määratluse põhjal võib rekursiooni olemus jääda üsna arusaamatuks, sestap vaatleme lihtsa näitena naturaalarvu faktoriaali arvutamist.

Eelmises peatükis defineerisime arvu n faktoriaali $n!$ valemiga

$$n! = \begin{cases} 1, & \text{kui } n = 0, \\ 1 \cdot 2 \cdot \ldots \cdot n, & \text{kui } n > 0. \end{cases}$$

Selle definitsiooni järgimisel on kõige loomulikum tulemus järgmine korduslauset kasutav algoritm:

Algoritm 4.3 FAKT1: Leiab antud naturaalarvu faktoriaali
Sisend: $n \in \mathbb{N}$
Väljund: tagastab $n!$
Abimuutujad: i, f

1. $f \leftarrow 1$
2. `korda` $i \leftarrow 1 \ldots n$
   — vahetingimus: $f = (i-1)!$
3. $f \leftarrow f * i$
   — vahetingimus: $f = i!$
4. `lõppkorda`
5. `tagasta` f

[8] Käesoleva õppematerjali lisades kasutatud keeled üldiselt ei toeta ei nimiparameetreid ega laiska väärtustamist. Mingil määral saab neid teiste vahenditega imiteerida, aga see on tehniliselt üsna keeruline, seetõttu me neid võtteid siinkohal ei käsitle.

Faktoriaali võib lisaks eelpool vaadeldud valemile defineerida ka seosega

$$n! = \begin{cases} 1, & \text{kui } n = 0, \\ n \cdot (n-1)!, & \text{kui } n > 0. \end{cases}$$

Selle definitsiooni järgimine annab meile rekursiivse algoritmi:

Algoritm 4.4 FAKT2: Leiab antud naturaalarvu faktoriaali
Sisend: $n \in \mathbb{N}$
Väljund: tagastab $n!$

1. `kui` $n = 0$
2. `tagasta` 1
3. `muidu`
4. `tagasta` $n *$ FAKT2$(n-1)$
5. `lõppkui`

Selle algoritmi täitmine näiteks 3! arvutamiseks toimub järgmiselt:
alamprogrammi kasutaja (tavaliselt programmi mõni teine osa) käivitab FAKT2(3);
süsteem loob alamprogrammi FAKT2 esimese eksemplari ja hakkab seda täitma; kuna $n = 3 \neq 0$, asub süsteem alamprogrammi kehas täitma valikulause `muidu`-haru; avaldise 3∗FAKT2(2) arvutamine käivitab FAKT2(2);
süsteem loob alamprogrammi FAKT2 teise eksemplari ja hakkab seda täitma; kuna $n = 2 \neq 0$, asub süsteem alamprogrammi kehas täitma valikulause `muidu`-haru; avaldise $2 *$ FAKT2(1) arvutamine käivitab FAKT2(1);
süsteem loob alamprogrammi FAKT2 kolmanda eksemplari ja hakkab seda täitma; kuna $n = 1 \neq 0$, asub süsteem alamprogrammi kehas täitma valikulause `muidu`-haru; avaldise $1 *$ FAKT2(0) arvutamine käivitab FAKT2(0);
süsteem loob alamprogrammi FAKT2 neljanda eksemplari ja hakkab seda täitma; kuna $n = 0$, tagastab süsteem FAKT2(0) väärtusena 1;
süsteem lõpetab $1 *$ FAKT2(0) arvutamise ja tagastab FAKT2(1) väärtusena $1 \cdot 1 = 1$;
süsteem lõpetab $2 *$ FAKT2(1) arvutamise ja tagastab FAKT2(2) väärtusena $2 \cdot 1 = 2$;
süsteem lõpetab 3∗FAKT2(2) arvutamise ja tagastab FAKT2(3) väärtusena $3 \cdot 2 = 6$.

4.4.2 Programmi magasin

Rekursiooni mõistmiseks peame kõigepealt loobuma peatüki alguses kasutusele võetud lihtsustatud kujutlusest alamprogrammide käivitamise mehhaanika kohta ja tegema endale selgeks, kuidas asjad tegelikult käivad.

Alamprogrammi käivitamisel luuakse **aktiveerimiskirje** (ingl *activation record*). Iga alamprogrammi iga eksemplari kohta luuakse uus kirje, mis iseloomustab alamprogrammi just seda käivitamist. Tavaliselt on selles kirjes kolme liiki andmed:

- alamprogrammile edastatavad tegelikud parameetrid;
- ruum alamprogrammi lokaalsete muutujate jaoks;
- **naasmisaadress** (ingl *return address*) — informatsioon selle kohta, millisest käsust tuleb jätkata pärast alamprogrammi täitmise lõppu.

Eelmises lõigus toodud näites oleks väljakutse FAKT2(2) aktiveerimiskirje sisu selline:

- tegelikud parameetrid: $n = 2$;
- lokaalsed muutujad: pole;
- naasmisaadress: korrutamistehe avaldises $n * \text{FAKT2}(n - 1)$.

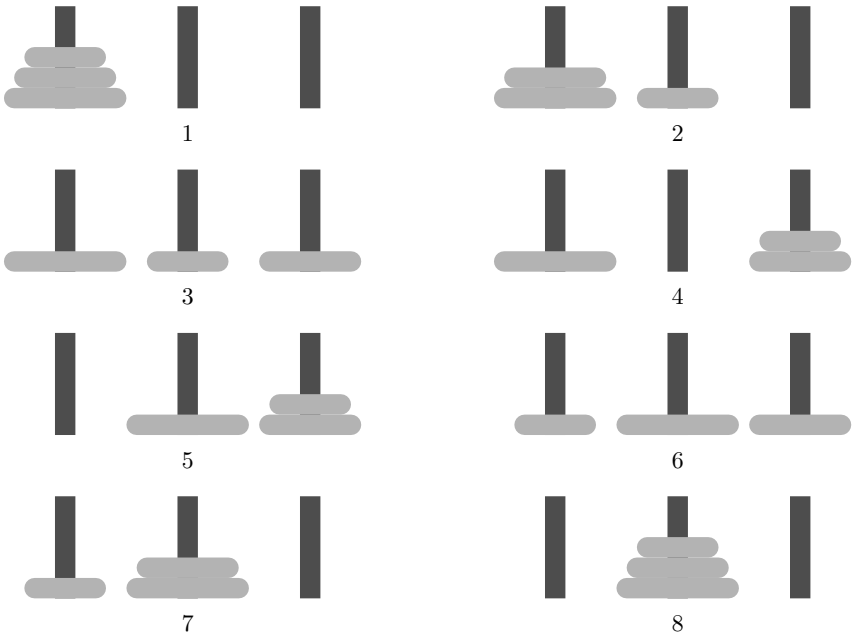
Rekursiooni toimimise jaoks on oluline, et neid aktiveerimiskirjeid võib ühe alamprogrammi jaoks olla mitu — kui iga alamprogrammi iga eksemplari parameetrid ja lokaalsed muutujad on omaette kirjes, võib korraga olla pooleli ühe alamprogrammi mitme eksemplari täitmine ilma et need üksteist segaks. Parasjagu pooleliolevate alamprogrammide aktiveerimiskirjeid hoitakse programmi **kutsemagasinis** (ingl *call stack*).[9]

Alamprogrammi ühe eksemplari töö lõppedes kustutatakse selle eksemplari aktiveerimiskirje ära ja vabanenud mälu võib süsteem kasutada nii selle kui ka teiste alamprogrammide uute eksemplaride loomiseks.

Magasini töö illustreerimiseks vaatleme veidi keerulisemat rekursiivset algoritmi, nimelt Hanoi tornide ülesande lahendust.

Ülesanne on järgmine: on kolm varrast, neist ühel n erineva suurusega ketast suuruse järjekorras (suurim all); vaja on kõik need kettad viia teisele vardale, kusjuures tõsta tohib ainult ühte ketast korraga; kolmandat varrast võib kasutada ajutise hoiukohana, kuid ühelegi vardale ei tohi panna suuremat ketast väiksema peale. Selle ülesande (üks võimalik) lahendus $n = 3$ jaoks on toodud joonisel 4.1.

[9] Peaaegu kõigis programmeerimiskeskkondades on võimalik programmi töö ajal magasini seisu vaadata. Rekursiivsete programmide silumisel on see väga vajalik vahend, seega tasub enne rekursiooni puudutavate koduste ülesannete lahendama hakkamist kindlasti välja uurida, kuidas teie kasutatavas süsteemis magasinile ligi pääseb.



Joonis 4.1: Hanoi tornid, $n = 3$

Hanoi tornide ülesannet on võimalik lahendada ka rekursiooni kasutamata, kuid rekursiivse algoritmi leidmine on palju lihtsam: selleks, et me saaks tõsta suurima ketta esimeselt vardalt teisele, peame enne kõik väiksemad kettad kolmandale vardale viima — kui mõned neist oleks esimesel vardal, ei saaks me suurimat ketast nende alt kätte; kui mõned neist oleks teisel vardal, et tohiks me suurimat ketast nende peale panna. Pärast suurima ketta teisele vardale tõstmist peame ka väiksemad kettad kolmandalt vardalt teisele viima. Osutub, et ülesande lahendus ongi täpselt nii lihtne:

Algoritm 4.5 HANOI: Lahendab Hanoi tornide ülesande
Sisend: $n \in \mathbb{N}$ — ketaste arv; $kust, kuhu, abi \in \{A, B, C\}$

1. `kui` $n > 0$
2. HANOI$(n-1, kust, abi, kuhu)$ — pealmised kettad lähtekohast abivardale
3. `väljasta` $kust$, `'→'`, $kuhu$ — alumine ketas lähtekohast sihtkohta
4. HANOI$(n-1, abi, kuhu, kust)$ — pealmised kettad abivardalt sihtkohta
5. `lõppkui`

Selle algoritmi täitmine näiteks 2 ketta viimiseks vardalt A vardale B toimub järgmiselt (märkide ⟨ ja ⟩ vahel on toodud programmi magasini seis):

alamprogrammi kasutaja käivitab HANOI$(2, A, B, C)$;
süsteem loob HANOI esimese eksemplari;
$\langle$HANOI$(2, A, B, C)\rangle$;
kuna $n = 2 > 0$, käivitatakse HANOI$(1, A, C, B)$;
süsteem loob HANOI teise eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, A, C, B)\rangle$;
kuna $n = 1 > 0$, käivitatakse HANOI$(0, A, B, C)$;
süsteem loob HANOI kolmanda eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, A, C, B)$, HANOI$(0, A, B, C)\rangle$;
kuna $n = 0$, lõpetab HANOI kolmas eksemplar oma töö kohe;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, A, C, B)\rangle$;
jätkub HANOI teise eksemplari täitmine: tõstetakse üks ketas vardalt A vardale C ja käivitatakse HANOI$(0, B, C, A)$;
süsteem loob HANOI uue kolmanda eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, A, C, B)$, HANOI$(0, B, C, A)\rangle$;
kuna $n = 0$, lõpetab HANOI kolmas eksemplar oma töö kohe;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, A, C, B)\rangle$;
HANOI teine eksemplar lõpetab oma töö;
$\langle$HANOI$(2, A, B, C)\rangle$;
jätkub HANOI esimese eksemplari täitmine: tõstetakse üks ketas vardalt A vardale B ja käivitatakse HANOI$(1, C, B, A)$;
süsteem loob HANOI uue teise eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, C, B, A)\rangle$;
kuna $n = 1 > 0$, käivitatakse HANOI$(0, C, A, B)$;
süsteem loob HANOI kolmanda eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, C, B, A)$, HANOI$(0, C, A, B)\rangle$;
kuna $n = 0$, lõpetab HANOI kolmas eksemplar oma töö kohe;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, C, B, A)\rangle$;
jätkub HANOI teise eksemplari täitmine: tõstetakse üks ketas vardalt C vardale B ja käivitatakse HANOI$(0, A, B, C)$;
süsteem loob HANOI uue kolmanda eksemplari;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, C, B, A)$, HANOI$(0, A, B, C)\rangle$;
kuna $n = 0$, lõpetab HANOI kolmas eksemplar oma töö kohe;
$\langle$HANOI$(2, A, B, C)$, HANOI$(1, C, B, A)\rangle$;
HANOI teine eksemplar lõpetab oma töö;
$\langle$HANOI$(2, A, B, C)\rangle$;
HANOI esimene eksemplar lõpetab oma töö;
$\langle\rangle$.

4.4.3 Rekursiooni õigsus

Eeltoodud näidete põhjal peaks juba silma torkama rekursiivsete algoritmide üldskeem: ülesande mingid lihtsad erijuhud lahendatakse otseselt, keerulisemad aga taandatakse kas ühele või mitmele mingis mõttes lihtsamale juhule ja otsitav tulemus saadakse nende lihtsamate juhtude lahendustest.

Neid lihtsaid erijuhte, mille algoritm otseselt lahendab, nimetatakse rekursiooni **baasiks** (ingl *base*) — nii faktoriaali kui ka Hanoi tornide ülesandes on baasiks juhtum, kui $n = 0$. Keerulisema juhu lahendamist lihtsamate juhtude kaudu nimetatakse rekursiooni **sammuks** (ingl *step*). Nagu eelneva põhjal oodata võibki, tuleb rekursiivse algoritmi õigsuses veendumiseks kontrollida nii baasi kui ka sammu õigsust.

Kuna rekursiooni baasi käsitlemine on "tavaline", rekursioonita algoritm, pole selle õigsuses veendumiseks vaja mingeid täiendavaid vahendeid — kasutada tuleb eelmistes peatükkides tutvustatud tehnikaid. Näiteks Hanoi tornide ülesande lahenduses on ilmne, et baasjuhtum lahendatakse õigesti — kui $n = 0$, tuleks ühelt vardalt teisele viia 0 ketast; kuna selleks pole vaja midagi teha, on algoritm, mis ei teegi midagi, ilmselt õige.

Rekursiooni sammu õigsuse tõestamisel eeldame, et alamprogrammi aktiivsest eksemplarist tehtavad väljakutsed selle algoritmi enda poole töötavad õigesti. Rekursiooni sammu õigsuse tõestamiseks on vaja veenduda, et selle eelduse kehtimisel töötab algoritm õigesti. Näiteks Hanoi tornide ülesande lahenduse korral eeldame, et mõlemad rekursiivsed väljakutsed kujul HANOI$(n-1,\ldots)$ toimetavad $n-1$ ketast nõutud viisil ühelt vardalt teisele. Sammu õigsuse tõestamiseks peame esiteks tähele panema, et kui me viime $n-1$ ketast lähtevardalt abivardale, tõstame viimase (suurima) ketta lähtevardale ja siis viime ka varem kõrvale pandud $n-1$ ketast abivardalt sihtvardale, on nende operatsioonide tulemusena tõesti kõik n ketast sihtvardal. Kui meie eeldused rekursiivsete väljakutsete õigsuse kohta kehtivad, on need kettad ka õiges järjekorras.

Aga sellest ei järeldu veel meie algoritmi õigsus. Nimelt tuleb meil lisaks kontrollida, et iga rekursiivse väljakutse eel kehtib meie algoritmi eeltingimus. Seda on muidugi raske teha, kui eeltingimus pole selgelt välja kirjutatud. Seda tingimust pole aga algoritmi 4.5 kirjelduses välja toodud sellepärast, et esimesena pähetulev sõnastus — eeltingimus: n ketast vardal *kust*, ülejäänud kaks varrast tühjad — on liiga range. Sellest eeltingimusest lähtudes ei võiks me rekursiivseid pöördumisi teha, sest meil on süsteemis üks liigne ketas.

Algoritmi õigsuses veendumiseks tuleb tähele panna, et tegelikult pole meil vaja nii ranget eeltingimust. Piisab sellest, kui me eeldame, et kõik need kettad, mida me oma algoritmi aktiivses eksemplaris ei puutu, on suuremad kui need, mida me liigutame. Tõepoolest, mõnel vardal meie ketastest allpool

olev ja neist kõigist suurem ketas ei erine ülesande tingimuste seisukohalt maapinnast — me võime kõiki teisi kettaid ilma mingite piiranguteta nende peale asetada.

Eelnevast lähtudes saame sõnastada oma lahenduse tegeliku eel- ja järeltingimuse — eeltingimus: n pealmist ketast vardal *kust*, kõik ülejäänud kettad kõigil kolmel vardal on suuremad kõigist neist n kettast; järeltingimus: n pealmist ketast vardalt *kust* viidud nende omavahelist järjekorda säilitades vardale *kuhu*, kõik muud kettad oma esialgsetel kohtadel. Sellise eel- ja järeltingimuse korral on rekursiooni sammu tõestamine juba lihtne.

Arvatavasti pole isegi selle ühe näite põhjal raske märgata, et rekursiivse algoritmi eeltingimus on natuke sarnane korduse invariandiga. Osutub, et see sarnasus pole juhuslik — nimelt on mistahes kordust kasutav algoritm võimalik üles kirjutada rekursiooni abil ja tavaliselt on seda teisendust kõige lihtsam teha just korduse invarianti rekursiooni eeltingimusena kasutades.

Rekursioonil ja kordusel on veel üks sarnasus — mõlemad võimaldavad koostada algoritme, mis jäävad lõpmatuseni tööle. Nagu korduse, nii ka rekursiooni puhul tõestab eel- ja järeltingimuste vaatlemine ainult algoritmi osalise õigsuse — kui see algoritm oma töö lõpetab, saame nõutud tulemuse. Tegelikult huvitab meid muidugi täielik õigsus — me tahame olla kindlad, et see tore hetk mingi lõpliku aja jooksul kätte jõuab.

Rekursiivse algoritmi puhul tähendab see, et me peame näitama, et mistahes lubatud algandmetest alustades jõuame mingi lõpliku arvu sammude järel välja rekursiooni baasini. Näiteks algoritmide 4.4 ja 4.5 puhul on selles veendumine triviaalne: kuna rekursiooni baas on $n = 0$ ja igal rekursiivsel pöördumisel vähendame n väärtust 1 võrra, jõuame n mistahes naturaalarvulisest väärtusest alustades baasjuhuni täpselt n sammu järel.

Kokkuvõtteks, rekursiivse algritmi koostamisel peame

- valima selle eel- ja järeltingimused nii, et need võimaldaks kasutada sama algoritmi ülesande mingite väiksemate osade lahendamiseks;
- valima mingid (soovitavalt lihtsad) baasjuhud, mille algoritm lahendab otseselt ja tõestama nende juhtude lahenduse õigsuse;
- taandama kõik ülejäänud juhud lihtsamate juhtude lahendamisele ja näitama, et taandamisel saadud lihtsamad juhud rahuldavad algoritmi eeltingimusi ja nende lihtsamate juhtude lahendustest koostatud lahendus rahuldab algoritmi järeltingimust;
- veenduma, et iga lubatud juht oleks kas baasjuht või taanduks baasjuhule mingi lõpliku arvu sammude järel.

Algoritmide 4.4 ja 4.5 jaoks oleme praeguseks kõik eeltoodud nõuded rahuldanud, seega võime nende õigsuse tõestatuks lugeda.

Ülesanded

Ülesanne 4.1 *Kirjutada õppematerjali lisas toodud algarvude arvutamise programmis protseduur* AlgarvudSajani *ümber funktsiooniks, mis algarvude ekraanile väljastamise asemel tagastab nende summa.*

Ülesanne 4.2 *Muuta sama programmi nii, et funktsioon* OnAlgarv *saab uuritava väärtuse parameetrina ja seejärel kaotada programmist globaalne muutuja a.*

Ülesanne 4.3 *Asendada samas programmis* AlgarvudSajani *funktsiooniga* AlgarvudAB, *mis saab parameetritena kaks täisarvu a ja b ning summeerib algarvud lõigus a . . . b. Uuritava lõigu otspunktid pärida kasutajalt.*

Ülesanne 4.4 *Naturaalarvu n poolfaktoriaal* $n!!$ *defineeritakse järgmiselt:*

$$n!! = \begin{cases} 1, & \text{kui } n = 0, \\ 1 \cdot 3 \cdot \ldots \cdot n, & \text{kui } n > 0 \text{ ja } n \text{ on paaritu,} \\ 2 \cdot 4 \cdot \ldots \cdot n, & \text{kui } n > 0 \text{ ja } n \text{ on paaris.} \end{cases}$$

Kirjutada poolfaktoriaalide arvutamiseks kaks funktsiooni: üks korduse ja teine rekursiooni abil. Tõestada mõlema õigsus. Kontrollida mõlema tööd ka piirjuhtudel ($n \in \{0, 1, 2\}$).

Ülesanne 4.5 *Algoritmi 4.5 järgi töötav programm teeb massiliselt alamprogrammi* Hanoi *väljakutseid, kus* $n = 0$. *Kuna alamprogramm sellisel juhul mingit kasulikku tööd ei tee, kulutab süsteem ilmaasjata aega selle alamprogrammi uute eksemplaride loomisele ja kustutamisele. Muuta algoritmi nii, et selliseid asjatuid väljakutseid ei oleks — see tähendab, kirjutada algoritm, milles baasjuht oleks* $n = 1$. *Tõestada uue versiooni õigsus.*

Ülesanne 4.6 *Fibonacci jada defineeritakse järgmise seosega:*

$$F_i = \begin{cases} 1, & \text{kui } i = 0 \text{ või } i = 1, \\ F_{i-1} + F_{i-2}, & \text{kui } i > 1. \end{cases}$$

Kirjutada antud järjekorranumbrile vastava Fibonacci jada liikme arvutamiseks kaks funktsiooni: üks rekursiooni ja teine korduse abil. Tõestada mõlema õigsus.

Võrrelda (katseliselt) nende funktsioonide töökiirusi parameetri i erinevatel väärtustel. Kas oskate kirjutada rekursiivse funktsiooni, mis töötab sama kiiresti kui kordust kasutav? (Viimane pole koduse töö kohtususlik osa.)

Peatükk 5

Algoritmide keerukus

Sisukord

Käesolevas peatükis tutvume algoritmi keerukuse mõistega ja vaatleme põhilisi meetodeid algoritmide keerukuse hindamiseks.

5.1 Algoritmi keerukuse mõiste

Algoritmi õigsuse kõrval on tähtis ka selle efektiivsus — võib juhtuda, et algoritm on küll teoreetiliselt korrektne, aga praktikas ei jätku meil selle alusel koostatud programmi täitmiseks ressursse.

Algoritmi efektiivsust võib hinnata mitme erineva ressursi kasutamise seisukohalt. Kõige levinum on algoritmi alusel koostatud programmi tööaja hindamine — seda tüüpi analüüsi nimetatakse algoritmi **ajalise keerukuse** (ingl *time complexity*) uurimiseks. Sageli on vaja hinnata ka programmi tööks kasutatava mälu mahtu — seda nimetatakse algoritmi **mahulise keerukuse** (ingl *space complexity*) uurimiseks. Muude ressursside hindamise vajadust tuleb praktikas oluliselt harvem ette.

Esmapilgul võib tunduda, et analüüsi asemel võib teha lihtsa katse — kirjutame programmi, paneme selle käima ja mõõdame meid huvitava ressursi kulu vahetult. Siiski pole selline lahendus alati kasutatav.

Esiteks võib juhtuda, et ülesanne on nii suur ja keeruline, et otsene mõõtmine pole praktiliselt võimalik — näiteks võib mõne ülesande lahendamine ebaefektiivse algoritmiga isegi parimatel superarvutitel võtta sajandeid. On selge, et nii kaua lahendust oodata ei saa. Sageli ongi algoritmi keerukuse analüüsimise eesmärgiks hinnata, kas ülesanne on olemasolevate ressurssidega lahendatav.

Teiseks võib algoritmi tööaeg erinevate algandmete korral olla erinev, mis tähendab, et ühest katsest ei piisa, neid tuleks teha mitmeid. Ja selleks, et otsustada, kui palju ja milliste andmetega katseid teha, on ikkagi vaja mingit eelnevat teoreetilist tööd.

Selleks, et arvestada algoritmi tööaja (või muu ressursikulu) sõltuvust algandmetest, väljendatakse algandmete "raskusaste" mingi arvulise suurusena ja avaldatakse tööaja hinnang funktsioonina, mille parameetriks on see algandmete raskusaste.

Analüüsi lihtsustamiseks püütakse andmete raskust väljendada võimalikult väikese arvu parameetrite abil. Kõige sagedamini kasutatakse parameetrina just algandmete mahtu: faili töötlemise algoritmi puhul faili suurust, arvujada töötlemisel selle elementide arvu jne.

Näiteks arvujadast maksimaalse väärtuse leidmiseks tuleb üldjuhul jada elemendid kõik ükshaaval läbi vaadata. Kui me loeme algoritmi sammuks jada ühe elemendi uurimise, siis kulutab selline järjestikune läbivaatus n-elemendilise jada töötlemiseks täpselt n sammu.

Sageli toob algandmete raskusastme lihtsustamine üheks parameetriks kaasa mõningase ebatäpsuse.

Näiteks selleks, et kontrollida, kas antud arv antud n-elemendilises jadas esineb või mitte, tuleb halvimal juhul läbi vaadata kõik n elementi — enne ei saa me kindlalt väita, et otsitavat nende hulgas pole. Seevastu parimal juhul on otsitav element jadas esimene ja me saame vaid ühe võrdluse järel kinnitada, et see arv jadas esineb. Edaspidises näeme, et keskmiselt kulub n-elemendilises jadas esineva elemendi leidmiseks $n/2$ võrdlust.

Jadast väärtuse leidmine pole selles mõttes erandlik ülesanne. Paljude algoritmide puhul on võimalik rääkida eraldi nende keerukusest **halvimal juhul** (ingl *worst-case complexity*), **parimal juhul** (ingl *best-case*) või **keskmiselt** (ingl *average-case*).

Lauaarvuti rakendusprogrammides huvitab meid tavaliselt keskmine keerukus. Näiteks otsimisprogrammi kasutaja jaoks on üldjuhul piisav teada, et programm suudab sekundis läbi vaadata keskmiselt 500 kB andmeid ja pole kuigi oluline, kas programm kulutab faili esimeses pooles iga kilobaidi läbivaatamiseks 1 ms ja teises pooles 3 ms või vastupidi.

Hoopis teine on lugu nn sardsüsteemide puhul, kus arvuti juhib mingit reaalajas töötavat seadet. Näiteks auto pidureid juhtiva pardaarvuti puhul on kindlasti oluline just selle reaktsiooniaeg halvimal juhul. Vaevalt lepib pidurisüsteemi viivituse tõttu haiglasse sattunud sõitja tehase lohutusega, et keskmiselt reageerivad pidurid piisavalt kiiresti ja teised sama marki auto omanikud pole veel avariid teinud.

5.1.1 Asümptootiline keerukus

Sageli on ühe ülesande lahendamiseks võimalik kasutada mitut erinevat algoritmi. Tavaliselt on programmi kasutajad sellisel juhul huvitatud võimalikult efektiivsest lahendusest. Algoritmide keerukuse analüüsi üks oluline rakendus ongi erinevate algoritmide keerukuse võrdlemine.

Kuna programmide ressursivajadused on enamasti suuremad just suuremahuliste andmete töötlemisel, on ka algoritmide efektiivsuse analüüsimisel loomulik pöörata tähelepanu põhiliselt sellele juhule.

Algoritmi sellist uurimist nimetatakse **asümptootilise keerukuse** (ingl *asymptotic complexity*) hindamiseks ja seda tüüpi analüüsis keskendutakse just algoritmi keerukusfunktsiooni **kasvukiiruse** (ingl *growth*) uurimisele.

Algoritmi keerukusfunktsiooni kasvukiirus näitab, kui kiiresti kasvab selle algoritmi alusel koostatud programmi ressursivajadus töödeldavate andmete mahu kasvades.

Veidi lihtsustades võib eeldada, et kui mingi algoritmi ajaline keerukus on $f(n)$, siis selle alusel koostatud programmi tööaeg avaldub kujul $cf(n)$,

kus c on kasutatava arvuti kiirust iseloomustav konstant. Kui c on ühe arvuti piires muutumatu konstant, siis võime algoritmide efektiivsuse võrdlemisel piirduda ainult $f(n)$ uurimisega.

Tabelist 5.1 on näha, et sisendandmete mahu suurenemisel võib programmi tööaja kasv, sõltuvalt kasutatava algoritmi keerukusest, olla väga erinev. Seejuures sõltub kasv algoritmi keerukust iseloomustavast funktsioonist $f(n)$, kuid ei sõltu arvutit iseloomustavast konstandist c. Samast on näha ka, et programmi tööaja kasv on lausa plahvatuslik, kui algoritmi keerukus avaldub eksponentfunktsioonina.[1]

Funktsioon $cf(n)$	Suhe $f(25)/f(5)$
c_1	1
$c_2 \log n$	2
$c_3\ n$	5
$c_4\ n \log n$	10
$c_5\ n^2$	25
$c_6\ n^3$	125
$c_7\ 2^n$	1 048 576
$c_8\ 3^n$	3 486 784 401

Tabel 5.1: Programmi tööaja kasv andmete mahu kasvades

Polünoomide[2] ja eksponentfunktsioonide kasvu põhimõttelist erinevust illustreerib veel paremini tabel 5.2. Selle tabeli esimeses veerus on programmi ajalise keerukuse hinnang, kolmes järgmises veerus programmi poolt sekundis töödeldavate andmete maht vastavalt 1, 10 ja 100 MOPS (miljonit operatsiooni sekundis) jõudlusega arvutil ja viimases veerus töödeldavate andmete mahu kasv arvuti jõudluse 10-kordsel kasvul.

Keerukus	1 MOPS	10 MOPS	100 MOPS	Vahe
n	1 000 000	10 000 000	100 000 000	10 **korda**
n^2	1 000	$\sim$ 3 162	10 000	$\sim$ 3 **korda**
n^3	100	$\sim$ 215	$\sim$ 464	$\sim$ 2 **korda**
2^n	$\sim$ 20	$\sim$ 23	$\sim$ 26	$\sim$ 3 **võrra**
3^n	$\sim$ 13	$\sim$ 15	$\sim$ 17	$\sim$ 2 **võrra**

Tabel 5.2: Töödeldavate andmete mahu kasv arvuti kiiruse kasvades

[1] Eksponentfunktsiooniks nimetatakse funktsiooni kujul a^x, kus a on konstant ja x funktsiooni argument. Mitte segi ajada astmefunktsiooniga, mis avaldub kujul x^a.

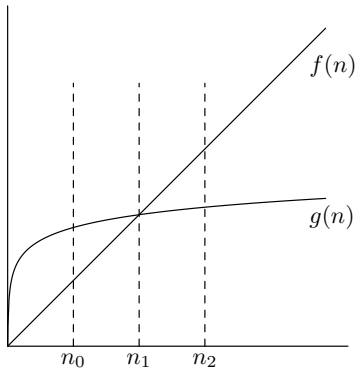
[2] Astmefunktsioonid on polünoomide lihtsamad erijuhud.

Tabelist on näha, et kui algoritmi keerukusfunktsioon on polünoom, siis arvuti jõudluse kasvamisel kordades kasvab kordades ka töödeldavate andmete maht. Kui aga algoritmi keerukus avaldub eksponentfunktsioonina, kasvab töödeldavate andmete maht ainult mingi konstandi võrra.

5.1.2 Funktsioonide kasv

Nagu eelpool märgitud, avaldub algoritmi keerukus algandmete raskusastme funktsioonina. Seega on meil algoritmide keerukuse võrdlemiseks vaja osata võrrelda funktsioone.

Osutub, et see polegi nii lihtne. Vaatleme näiteks joonist 5.1. Kui me võrdleme $f(n)$ ja $g(n)$ väärtusi kohal n_0, saame tulemuseks $f(n_0) < g(n_0)$; kohal n_1 saame $f(n_1) = g(n_1)$; kohal n_2 aga hoopis $f(n_2) > g(n_2)$.



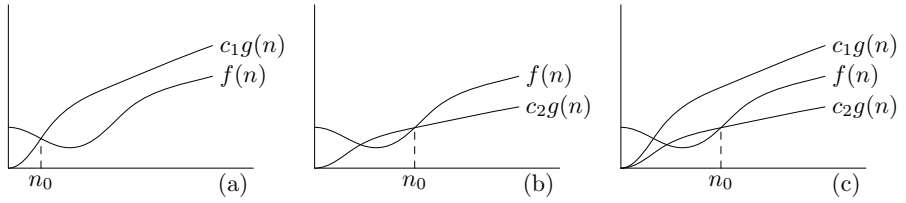
Joonis 5.1: Funktsioonide võrdlemine

Selle "vastuolu" põhjus on muidugi asjaolus, et funktsioonide väärtused sõltuvad nende argumentide väärtustest[3] ja selle lahendamiseks on leitud mitmesuguseid funktsioonide võrdlemise viise. Algoritmide analüüsimisel on neist kõige kasulikumaks osutunud funktsioonide võrdlemine nende kasvukiiruse järgi.

Öeldakse, et funktsiooni $f(n)$ kasvukiirus ei ületa funktsiooni $g(n)$ kasvukiirust, ja kirjutatakse $f(n) = \mathrm{O}(g(n))$, kui leiduvad positiivsed arvud c_1 ja n_0, mille korral kehtib

$$\forall n \geq n_0 : f(n) \leq c_1 g(n).$$

[3] See ju ongi funktsioonide mõte!



Joonis 5.2: (a) $f(n) = \mathrm{O}(g(n))$; (b) $f(n) = \Omega(g(n))$; (c) $f(n) = \Theta(g(n))$

Paneme tähele, et $f(n) = \mathrm{O}(g(n))$ definitsioon ei piira mitte $f(n)$ või $g(n)$ väärtusi endid, vaid nende väärtuste suhet $f(n)/g(n)$, mis ei tohi lõpmatult kasvada.

Kuna konstant c_1 piirab suhet $f(n)/g(n)$ "ülalt", võib funktsioonide kasvule O-hinnanguid anda suvalise "liiaga" — $g(n)$ võib kasvada kuitahes palju kiiremini kui $f(n)$.

Näiteks $f(n) = 3n^2 + 1$ korral kehtivad nii $f(n) = \mathrm{O}(n^2)$ kui ka $f(n) = \mathrm{O}(n^3)$, $f(n) = \mathrm{O}(n^4)$ ja isegi $f(n) = \mathrm{O}(2^n)$. Nende kõigi puhul on O definitsiooni tingimused rahuldatud, kui valime $c_1 = 4$ ja $n_0 = 1$.

Küll aga ei kehti $f(n) = \mathrm{O}(n)$ (ehk teisiti öeldes, kehtib $f(n) \neq \mathrm{O}(n)$), sest pole võimalik valida O definitsiooni nõudeid rahuldavaid c_1 ja n_0. Tõepoolest, mistahes c_1 korral $f(n) > c_1 n$, kui $n \geq c_1/3$, seega pole ühegi c_1 jaoks võimalik leida O definitsioonis nõutud konstanti n_0.

Veel kirjutatakse $f(n) = \Omega(g(n))$, kui leiduvad positiivsed c_2 ja n_0, mille korral kehtib

$$\forall n \geq n_0 : f(n) \geq c_2 g(n).$$

Väited $f(n) = \Omega(g(n))$ ja $g(n) = \mathrm{O}(f(n))$ on samaväärsed. Tõepoolest, oletame, et $f(n) = \Omega(g(n))$. Vastavalt Ω definitsioonile peavad leiduma sellised positiivsed n_0 ja c_2, et iga $n \geq n_0$ korral kehtib $f(n) \geq c_2 g(n)$. Kui nüüd võtame $c_1 = 1/c_2$, siis peab iga $n \geq n_0$ korral kehtima ka $g(n) \leq c_1 f(n)$. See aga on täpselt $g(n) = \mathrm{O}(f(n))$ definitsioonis nõutud tingimus. Vastupidise järelduse võime tõestada analoogiliselt.

Veel kirjutatakse $f(n) = \Theta(g(n))$, kui leiduvad positiivsed c_1, c_2 ja n_0, mille korral kehtib

$$\forall n > n_0 : c_2 g(n) \leq f(n) \leq c_1 g(n).$$

Jällegi on lihtne veenduda, et $f(n) = \Theta(g(n))$ parajasti siis, kui $f(n) = \mathrm{O}(g(n))$ ja $f(n) = \Omega(g(n))$.

Erinevalt seosest O piirab seose Θ definitsioon funktsioonide $f(n)$ ja $g(n)$ väärtuste suhet mitte ainult "ülalt", vaid ka "alt". Seega $f(n) = \Theta(g(n))$

tähendab, et $f(n)$ ei kasva küll kiiremini kui $g(n)$, kuid samal ajal ei jää kasvus ka oluliselt maha.

Ülaltoodud näites vaadeldud $f(n) = 3n^2 + 1$ korral $f(n) = \Theta(n^2)$, kuid $f(n) \neq \Theta(n^3)$, $f(n) \neq \Theta(n^4)$ ja $f(n) \neq \Theta(2^n)$. Väite $f(n) = \Theta(n^2)$ kehtivuses veendumiseks valime $c_1 = 4$, $c_2 = 3$ ja $n_0 = 1$. Ülejäänud juhtudel on mistahes positiivse c_2 korral võimalik näidata, et n piisavalt suurte väärtuste juures jäävad $f(n)$ väärtused teistega võrreldes liiga väikesteks.

Avaldisi $f(n) = \Omega(g(n))$ ja $f(n) = \Theta(g(n))$ loetakse vastavalt "eff-enn on oomega-gee-enn" ja "eff-enn on teeta-gee-enn". Avaldist $f(n) = \mathrm{O}(g(n))$ loetakse "eff-enn on oo-gee-enn", kuigi rangelt võttes peaks selles avaldises võrdusmärgist paremal olema kreeka täht omikron.

Edasises kasutame mõnikord seoseid O, Ω ja Θ ka mitme muutuja funktsioonidel. Need üldistused defineeritakse loomulikul viisil: näiteks $f(n, m) = \mathrm{O}(g(n, m))$, kui leiduvad c_1, n_0 ja m_0, mille korral kehtib

$$\forall n \geq n_0, m > m_0 : f(n, m) \leq c_1 g(n, m).$$

5.1.3 Kasvuseoste omadused

Nagu eelnevatest lõikudest näha, võib tõmmata paralleele funktsioonide vahel kehtivate seoste O, Ω ja Θ ning arvude vahel kehtivate seoste $\leq$, $\geq$ ja $=$ vahele. See ei tohiks olla eriline üllatus, sest funktsioonide kasvu asusimegi uurima just eesmärgiga leida vahend nende võrdlemiseks.

Siiski pole kõik arvude võrdlemise seoste omadused funktsioonidele üle kantavad. Nimelt tekitavad funktsioonide kasvukiiruste võrdlemise seosed funktsioonide vahel **osalise järjestuse** (ingl *partial order*), kuid arvude võrdlemise seosed arvude vahel **lineaarse järjestuse** (ingl *total order*).

Nende erinevus seisneb selles, et mistahes kaks arvu on alati võrreldavad — suvaliste arvude a ja b puhul kehtib alati vähemalt üks väidetest $a \leq b$ või $a \geq b$ —, aga kahe funktsiooni $f(n)$ ja $g(n)$ puhul ei tarvitse kehtida ei $f(n) = \mathrm{O}(g(n))$ ega $f(n) = \Omega(g(n))$. Üks võrreldamatute kasvukiirustega funktsioonide paar on näiteks $f(n) = n$ ja $g(n) = n^{1+\sin n}$.

Algoritmide keerukuse analüüsimisel on sageli kasu seoste O, Ω ja Θ järgmistest omadustest:

1. $\forall c > 0 : cf(n) = \Theta(f(n))$
   see tähendab, et funktsiooni korrutamine konstandiga ei muuda selle kasvukiirust; erijuhul $c = 1$ saame $f(n) = \Theta(f(n))$, järelikult on seos Θ refleksiivne, täpselt nagu seos $=$ arvudel; samast järeldub, et ka seosed O ja Ω on refleksiivsed, täpselt nagu seosed $\leq$ ja $\geq$ arvudel; erijuhul $f(n) = 1$ saame $c = \Theta(1)$, järelikult on kõigi konstantsete funktsioonide kasvukiirused võrdsed;

2. $f(n) = \mathrm{O}(g(n)) \wedge g(n) = \mathrm{O}(h(n)) \supset f(n) = \mathrm{O}(h(n))$
   $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \supset f(n) = \Omega(h(n))$
   see tähendab, et seosed O ja Ω on transitiivsed, täpselt nagu seosed $\leq$ ja $\geq$ arvudel; sellest järeldub, et ka seos Θ on transitiivne, täpselt nagu seos $=$ arvudel;

3. $f(n) = \Theta(h(n)) \wedge g(n) = \mathrm{O}(h(n)) \supset (f(n) + g(n)) = \Theta(h(n))$
   kahe funktsiooni summa kasvu määrab kiirema kasvuga liidetav;

4. $f_1(n) = \Theta(g_1(n)) \wedge f_2(n) = \mathrm{O}(g_2(n)) \supset (f_1(n)f_2(n)) = \mathrm{O}(g_1(n)g_2(n))$
   $f_1(n) = \Theta(g_1(n)) \wedge f_2(n) = \Omega(g_2(n)) \supset (f_1(n)f_2(n)) = \Omega(g_1(n)g_2(n))$
   seda omadust võib tõlgendada nii, et kahe funktsiooni korrutise kasv on tegurite kasvude korrutis;

5. kui $p(n)$ on k. astme polünoom, siis $p(n) = \Theta(n^k)$
   see tähendab, et polünoomi kasvu määrab selle kõrgeima astmega liige; väga kasulik omadus, mille põhjal võime mistahes polünoomi uurimise asendada üksliikme uurimisega;

6. $\forall 0 \leq r \leq s : n^r = \mathrm{O}(n^s)$
   see tähendab, et madalama astmega astmefunktsioon ei kasva kiiremini kõrgema astmega astmefunktsioonist; kehtib ka tugevam väide: madalama astmega funktsioon kasvab alati rangelt aeglasemalt;

7. $\forall k \geq 0, b > 1 : n^k = \mathrm{O}(b^n)$
   see tähendab, et mitte ükski astmefunktsioon ei kasva kiiremini ühestki eksponentfunktsioonist; tegelikult kasvab astmefunktsioon alati rangelt aeglasemalt;

8. $\forall k > 0, b > 1 : \log_b n = \mathrm{O}(n^k)$
   see tähendab, et ükski logaritmfunktsioon ei kasva kiiremini ühestki astmefunktsioonist; tegelikult kasvab logaritmfunktsioon alati rangelt aeglasemalt;

9. $\forall b_1 > 1, b_2 > 1 : \log_{b_1} n = \Theta(\log_{b_2} n)$
   see tähendab, et kõik logaritmfunktsioonid kasvavad sama kiirusega;

10. $\forall r \geq 0 : \sum_{i=1}^{n} i^r = \Theta(n^{r+1})$
    see tähendab, et r. järku astmerea summa kasvab nagu $(r + 1)$. astme polünoom.

Vaatleme näiteks funktsioone $f(n) = (n^2 - n)/2$ ja $g(n) = 6n$ ning uurime nende kasvukiirusi. Selliste lihtsate funktsioonide puhul pole muidugi raske

vahetult tuletada, et $f(n) > cg(n)$, kui $n > 12c - 1$ ja sellest järeldada, et $f(n) \neq \mathrm{O}(g(n))$, $f(n) = \Omega(g(n))$ ja $f(n) \neq \Theta(g(n))$.

Pannes tähele, et $f(n) = (n^2 - n)/2 = 0{,}5n^2 - 0{,}5n$ on 2. astme polünoom, saame omaduse 5 põhjal $f(n) = \Theta(n^2)$. Kuna $g(n)$ on ilmselt 1. astme polünoom, kehtib sama omaduse põhjal $g(n) = \Theta(n)$. Edasi saamegi omaduse 6 alusel $f(n) \neq \mathrm{O}(g(n))$, $f(n) = \Omega(g(n))$ ja $f(n) \neq \Theta(g(n))$, mis muidugi ei erine eelmises lõigus otseselt saadud tulemustest.

Erinevalt eelmise näite lihtsatest funktsioonidest oleks $f(n) = n\sqrt{n}$ ja $g(n) = n + \log n$ kasvukiiruste võrdlemine otseselt seoste O, Ω ja Θ definitsioonide põhjal üsna tülikas. Neid funktsioone on oluliselt mugavam võrrelda seoste O, Ω ja Θ eeltoodud omadusi kasutades.

Tõepoolest, elementaaralgebrast on teada, et $f(n) = n\sqrt{n} = n^{1{,}5}$ ning omaduste 8 ja 3 põhjal saame $g(n) = n + \log n = \mathrm{O}(n)$. Edasi on omaduse 6 nõrgema väite põhjal näha, et $g(n) = \mathrm{O}(f(n))$ ehk $f(n) = \Omega(g(n))$; sama omaduse tugevama väite põhjal saame $f(n) \neq \mathrm{O}(g(n))$, millest omakorda järeldub $f(n) \neq \Theta(g(n))$.

Avaldiste teisendamisel kirjutatakse sageli näiteks $f(n) = n + \log n = \Theta(n) + \mathrm{O}(n) = \Theta(n)$. Selle (rangelt võttes veidi ebakorrektse) kirjutise tähendus on, et $f(n)$ koosneb kahest liidetavast, milles üks kasvab täpselt sama kiiresti kui n ja teine mitte kiiremini kui n, seega peab ka summa kasvama täpselt sama kiiresti kui n.

Ebakorrektsus seisneb selles, et $\mathrm{O}(n)$ pole konkreetne funktsioon, mida oleks võimalik mõne teise funktsiooniga liita. Siiski on selline notatsioon just mugavuse tõttu laialt levinud. Tuleb ainult olla ettevaatlik, et mitte teha ebaõigeid teisendusi. Näiteks "taandamine" $f(n) = \log n / \log \log n = \mathrm{O}(n)/\mathrm{O}(n) = \mathrm{O}(1)$ on ebaõige, sest kuigi $\log n$ ja $\log \log n$ on mõlemad $\mathrm{O}(n)$, pole nende kasvukiirused sugugi võrdsed — nad ainult kasvavad mõlemad aeglasemalt kui n.

5.2 Algoritmide keerukuse hindamine

Järgnevates lõikudes vaatleme tähtsamaid võtteid algoritmide ajalise keerukuse hindamiseks. Kuna just juhtkonstruktsioonid määravad, milliseid primitiive ja kui palju kordi täidetakse, ei tohiks olla eriline üllatus, et nii algoritmi ajaline keerukus kui ka selle hindamise võtted on otseselt seotud algoritmi struktuuriga.

Mahulise keerukusega tegeleme järgmistes peatükkides andmestruktuure vaadeldes, sest programmi mäluvajadus sõltub otseselt just andmete, mitte algoritmi struktuurist.

5.2.1 Maksumuse mudel

Selleks, et hinnata algoritmi kõigi primitiivide täitmiseks kuluvat koguaega, on muidugi vaja teada iga primitiivi täitmiseks kuluvat aega. Seda infot kõigi primitiivide kohta kokku nimetatakse süsteemi **maksumuse mudeliks** (ingl *cost model*). Erinevate operatsioonide maksumused võivad sõltuda nii arvuti riistvarast, operatsioonisüsteemist kui ka kasutatavatest programmeerimisvahenditest.

Lihtsaima mudeli puhul eeldame, et kõigi primitiivide maksumused on võrdsed. Sellisel juhul saame algoritmi ajaliseks keerukuseks primitiivi maksumuse ja täidetud primitiivide arvu korrutise. Kuigi selline mudel pole peaaegu kunagi päris täpne, on see sageli siiski piisav.

Näiteks võime arvutusalgoritmide hindamisel võrdsustada kõik aritmeetilised tehted kõigi standardsete arvutüüpide vahel. Kuigi tehte sooritamiseks kuluv aeg sõltub tavaliselt nii sooritatavast tehtest kui ka operandide tüüpidest (ja veel paljudest muudest asjadest), on need erinevused suhteliselt väikesed ja, mis peamine, konstantsed. Seega, kui meid huvitab ainult tööaja kasvu sõltuvus algandmetest, pole üksikute tehete vahelistel erinevustel meie jaoks erilist tähtust.

Selline lihtne mudel pole aga piisav, kui programmeerimissüsteemi primitiivide hulgas on ka keerulisemaid operatsioone. Näiteks mingi andmehulga sorteerimine või sellest hulgast vajalike andmete leidmine on operatsioonid, mille ajakulu ei sõltu mitte ainult andmetüüpidest, vaid ka andmemahtudest. Selliseid sõltuvusi enam eirata ei või.

5.2.2 Lineaarne algoritm

Algoritmi, milles ainsa juhtkonstruktsioonina on kasutusel järjend, nimetatakse **lineaarseks** (ingl *linear*). Kuna sellises algoritmis täidetakse iga primitiivi täpselt üks kord, on terve algoritmi täitmiseks kuluv aeg primitiivide täitmisaegade summa.

Täpsemalt, kui algoritm on kujul

käsk-1
käsk-2
...
käsk-k

ning *käsk-i* täitmiseks kuluv aeg on $f_i(n)$, siis avaldub algoritmi täitmise koguaeg $T(n)$ kujul

$$T(n) = f_1(n) + f_2(n) + \ldots + f_k(n).$$

Erijuhul, kui kõigi primitiivide täitmisajad on konstandid, on seda ilmselt ka terve lineaarse algoritmi täitmise aeg.

Kui primitiivide täitmise ajad on keerukamad funktsioonid, võib kasvukiiruse avaldise lihtsustamiseks kasutada lõigus 5.1.3 vaadeldud omadusi.

5.2.3 Hargnemistega algoritm

Algoritmi, milles juhtkonstruktsioonidena on kasutusel järjend ja valik, nimetatakse **hargnemistega** (ingl *branching*) algoritmiks. Sellises algoritmis täidetakse iga primitiivi ülimalt üks kord, seega ei saa algoritmi täitmiseks kuluv aeg mingil juhul ületada kõigi primitiivide täitmisaegade summat, küll aga võib olla sellest väiksem.

Täpsemalt, kui algoritm on kujul

```
kui tingimus
   tõene-käsud
muidu
   väär-käsud
lõppkui
```

ning *tõene-käsud* ajaline keerukus on $f_1(n)$, *väär-käsud* ajaline keerukus $f_2(n)$ ja tingimuse tõeväärtuse arvutamise keerukus $f_0(n)$, siis terve algoritmi ajaline keerukus on parimal juhul

$$T_{min}(n) = f_0(n) + min(f_1(n), f_2(n))$$

ja halvimal juhul

$$T_{max}(n) = f_0(n) + max(f_1(n), f_2(n)).$$

Kui *tingimus* kehtib tõenäosusega p[4], on algoritmi keskmine keerukus

$$T(n) = f_0(n) + pf_1(n) + (1-p)f_2(n).$$

Selle valemi põhjenduseks paneme tähele, et igal juhul tuleb väärtustada *tingimus*, kulutades selleks $f_0(n)$. Edasi tuleb tõenäosusega p täita *tõene-käsud*, kulutades selleks $f_1(n)$, või tõenäosusega $1-p$ täita *väär-käsud*, kulutades selleks $f_2(n)$. Kui me käivitame seda algoritmi M korda, siis keskmiselt pM juhul on *tingimus* tõene, ülejäänud $M - pM = (1-p)M$ juhul aga on *tingimus* väär. Kokku kulutame algoritmi M-kordsel käivitamisel

[4]Siin ja edaspidi märgime tõenäosusi mitte protsentides, vaid reaalarvudega $0 \ldots 1$. Arvestades, et protsent tähendab sajandikku, saame $20\% = 20/100 = 0{,}2$. Vahe on selles, et viimast kuju kasutades ei pea valemites tõenäosusi sajaga jagama, mis muudab valemid veidi lihtsamaks ja ülevaatlikumaks.

$Mf_0(n) + pMf_1(n) + (1-p)Mf_2(n)$, mis annabki ühe käivitamiskorra keskmiseks hinnaks $f_0(n) + pf_1(n) + (1-p)f_2(n)$.

Vaatleme näiteks ühest valikulausest koosnevat algoritmi:

— $n, k \in \mathbb{N}$; $n \geq k > 0$
`kui` $n/k \in \mathbb{N}$
  — midagi, mis on $\Theta(n)$
`muidu`
  — midagi, mis on $\Theta(1)$
`lõppkui`

Kui n ja k on programmeerimissüsteemi standardsed täisarvud, võime eeldada, et nende jaguvus on kontrollitav konstantse ajaga. Seega on selle algoritmi ajalise keerukuse hinnangud vastavalt eelpool toodud seostele

- parimal juhul:
  $T_{min}(N) = \Theta(1) + min(\Theta(n), \Theta(1)) = \Theta(1) + \Theta(1) = \Theta(1)$;
- halvimal juhul:
  $T_{max}(n) = \Theta(1) + max(\Theta(n), \Theta(1)) = \Theta(1) + \Theta(n) = \Theta(n)$;
- keskmiselt:
  kui arvudele n ja k pole seatud muid piiranguid, on jaguvuse tõenäosus $1/k$, mis annab keskmiseks ajakuluks
  $T(n) = \Theta(1) + (1/k)\Theta(n) + (1 - 1/k)\Theta(1) = \Theta(n/k)$,
  sest pole raske näha, et $n \geq k$ korral on $T(n)$ avaldises domineeriv keskmine liidetav.

5.2.4 Iteratiivne algoritm

Algoritmi, milles juhtkonstruktsioonina on kasutusel kordus, nimetatakse **iteratiivseks** (ingl *iterative*). Sellises algoritmis täidetakse korduslause kehas olevaid primitiive korduvalt, seega võib selle keerukus olla üsna suur.

Kui iteratiivne algoritm on kujul

`korda` $i \leftarrow 1 \ldots k$
  käsud
`lõppkorda`

ning *käsud* ajaline keerukus on $f(n)$, siis on kogu korduslause ajaline keerukus muidugi $kf(n)$.

Vaatleme näiteks juba tuttavat algoritmi arvujada liikmete summa leidmiseks:

```
— a_{1...n} — summeeritav jada
s ← 0
korda i ← 1...n
   s ← s + a_i
lõppkorda
väljasta s
```

Korduse kehaks on siin üks liitmis- ja üks omistamistehe, mille täitmiseks kulub ilmselt konstantne hulk aega, seega antud juhul $f(n) = \Theta(1)$. Seda korduse keha täidetakse täpselt n korda, seega on korduse keerukuseks $n\Theta(1)$. Lisaks sellele täidetakse enne kordust veel üks omistamine (keerukus $\Theta(1)$) ja pärast seda üks väljastamine (jälle $\Theta(1)$). Seega on algoritmi kogukeerukus

$$T(n) = \Theta(1) + n\Theta(1) + \Theta(1) = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n).$$

Ülesanne muutub raskemaks, kui korduse keha täitmise hind pole korduse kõigil läbimistel sama. Sellisel juhul peame *käsud* keerukuse avaldama kahe muutuja funktsioonina $f(n, i)$ ja kogu korduslause ajaline keerukus avaldub summana

$$T(n) = \sum_{i=1}^{k} f(n, i) = f(n, 1) + f(n, 2) + \ldots + f(n, k).$$

Väliselt on summa sarnane eelpool vaadeldud lineaarse algoritmi keerukuse avaldisega, kuid sellest ei tohi ennast petta lasta. Oluline erinevus võrreldes lineaarse algoritmiga seisneb asjaolus, et lineaarses algoritmis on summas esinevate liidetavate arv määratud programmi struktuuriga, kordusega algoritmis aga võib sõltuda sisendandmetest.

Vaatleme näiteks kahest pesastatud kordusest koosnevat algoritmi, mis loendab antud arvujadas kõik inversioonid (paarid, kus suurem arv on jadas väiksemast eespool):

```
— a_{1...n} — uuritav jada
k ← 0
korda i ← 1...n
  korda j ← i + 1...n
    kui a_i > a_j
      k ← k + 1
    lõppkui
  lõppkorda
lõppkorda
väljasta k
```

Selliste pesastatud juhtkonstruktsioonidega algoritmide keerukust on sageli lihtsam arvutada, liikudes algoritmi struktuuris seestpoolt väljapoole.

Valikulauses on nii arvude võrdlemine kui ka tingimuslikult täidetav omistamine konstantse ajaga operatsioonid, seega on kogu valikulause ajaline keerukus $\Theta(1)$.

Valikulause ise on sisemise korduslause keha, teda täidetakse j väärtustel $i+1$ kuni n, see tähendab täpselt $n-i$ korda. Kuna selle korduse keha täitmise maksumus on kõigil läbimistel sama, on meil tegemist lihtsama juhuga ja korduse kogukeerukus $f(n,i) = (n-i)\Theta(1)$.

Sisemine korduslause omakorda on välimise korduse keha, teda täidetakse i väärtustel 1 kuni n. Kuna selle korduse keha täitmise maksumus on erinevatel läbimistel erinev, on meil seekord tegemist keerukama juhuga ja korduse kogukeerukus

$$\begin{aligned} T(n) &= (n-1)\Theta(1) + (n-2)\Theta(1) + \ldots + (n-n)\Theta(1) \\ &= ((n-1) + (n-2) + \ldots + (n-n))\Theta(1) \\ &= \frac{n(n-1)}{2}\,\Theta(1) = \Theta(n^2)\Theta(1) = \Theta(n^2). \end{aligned}$$

Toodud näites on üleminek teiselt realt kolmandale tehtud aritmeetilise jada summa valemi põhjal. Kahjuks pole selliste summade lihtsustamiseks üldist eeskirja — neisse tuleb suhtuda loominguliselt.

Muidugi tasub võimalusel alati kasutada seoste O ja Θ lõigus 5.1.3 vaadeldud omadusi, korduslausete puhul on sageli abiks just omadus 10.

Osutub, et ka käesoleval juhul saaks me kasutada just seda omadust: $T(n)$ avaldises

$$T(n) = ((n-1) + (n-2) + \ldots + (n-n))\Theta(1)$$

võime sulgudes olevat summat S teisendada järgmiselt:

$$\begin{aligned} S &= (n-1) + (n-2) + \ldots + (n-n) \\ &= 0 + 1 + \ldots + (n-1) \\ &= 0 + \sum_{i=1}^{n} i - n \\ &= \Theta(1) + \Theta(n^2) - \Theta(n) = \Theta(n^2). \end{aligned}$$

Veel võib olla kasulik meeles pidada, et O-hinnanguid võib, erinevalt Θ-hinnanguist, anda suvalise liiaga. See tähendab, et kui korduse keerukuse täpse avaldise lihtsustamine ei õnnestu, võime Θ-hinnangu asemel anda mõningase liiaga O-hinnangu, "ümardades" kõik liidetavad ülespoole mingiks lihtsustamiseks soodsama kujuga avaldiseks.

Näiteks inversioonide loendamise programmi keerukuse avaldise lihtsustamisel võime kasutada ära asjaolu, et kõigis liidetavates on esimene korrutis n-st väiksem arv, seega saame

$$\begin{aligned} T(n) &= (n-1)\Theta(1) + (n-2)\Theta(1) + \ldots + (n-n)\Theta(1) \\ &< n\Theta(1) + n\Theta(1) + \ldots + n\Theta(1) \\ &= n^2\Theta(1) = \Theta(n^2) \\ T(n) &= \mathrm{O}(n^2). \end{aligned}$$

Selle näite puhul juhtus, et ka "ülespoole ümardatud" hinnang on täpne, aga alati ei pruugi see nii olla. Sellepärast ei või me liiaga antud hinnangu avaldamisel enam kasutada seost Θ.

5.2.5 Rekursiivne algoritm

Rekursiivse alamprogrammina avaldatavat algoritmi nimetatakse samuti **rekursiivseks** (ingl *recursive*). Nagu rekursiivne alamprogramm ise defineeritakse iseenda kaudu, tehakse seda ka tema keerukusfunktsiooniga.

Vaatleme näiteks juba varasemast tuttavat rekursiivset algoritmi antud naturaalarvu n faktoriaali leidmiseks:

```
— n ∈ ℕ
kui n = 0
  tagasta 1
muidu
  — rekursioon
  tagasta n * (n − 1)!
lõppkui
```

Selle algoritmi tööaja hinnangu $T(n)$ võib avaldada kujul

$$T(n) = \begin{cases} \Theta(1), & \text{kui } n = 0, \\ T(n-1) + \Theta(1), & \text{kui } n > 0, \end{cases}$$

kus juhul $n > 0$ väljendab $T(n-1)$ rekursiivsele väljakutsele kuluvat aega ja $\Theta(1)$ selle väärtuse kasutamist $n!$ arvutamisel.

Sellised **rekurrentsed** (ingl *recurrent*) võrrandid ja võrrandisüsteemid pole matemaatikas midagi haruldast. Kõige üldisem meetod nende lahendamiseks on: tuleb rekurrentseid pöördumisi mõned korrad avaldisse sisse asendada ja püüda vastuse üldkuju ära arvata. Tekkinud hüpoteeside tõestamiseks on tavaliselt kõige parem matemaatilise induktsiooni meetod.

Näiteks faktoriaali leidmise algoritmi puhul saame

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ &= T(n-2) + \Theta(1) + \Theta(1) \\ &= T(n-3) + \Theta(1) + \Theta(1) + \Theta(1), \end{aligned}$$

millest on juba küllalt lihtne pakkuda $T(n) = (n+1)\Theta(1) = \Theta(n)$.

Tabelis 5.3 on ära toodud mõnede rekursiivsete algoritmide analüüsimisel sageli esinevate rekurrentsete võrrandite lahendid. Tabeli vasakpoolses veerus on antud $T(n)$ avaldis $n > 0$ jaoks, lisaks eeldatakse kõigil juhtudel rajatingimust $T(0) = \Theta(1)$.

$T(n)$ avaldis	lahend
$T(n/c_1) + \Theta(1)$	$\Theta(\log n)$
$T(n-1) + \Theta(1)$	$\Theta(n)$
$c_2T(n/c_2) + \Theta(1)$	$\Theta(n)$
$c_3T(n/c_3) + \Theta(n)$	$\Theta(n \log n)$
$T(n-1) + \Theta(n)$	$\Theta(n^2)$
$c_4T(n-1) + \Theta(1)$	$\Theta({c_4}^n)$

Tabel 5.3: Mõnede sageli esinevate rekurrentsete võrrandite lahendid

5.3 Praktiline nõuanne

Algoritmide efektiivsuse hindamisele pühendatud peatüki lõpetuseks tasub märkida, et efektiivsus pole siiski algoritmi kõige tähtsam omadus. Õigsus on alati efektiivsusest olulisem. Tõepoolest, millist kasu võiks loota programmist, mis töötab küll välkkiirelt, kuid annab valesid vastuseid?

Ülesanded

Ülesanne 5.1 *Kontrollida otseselt seoste* O, Ω *ja* Θ *definitsioonidest lähtudes, kas kehtivad väited*

(a) $2^{n+1} = \mathrm{O}(2^n)$; *(d)* $2^{2n} = \mathrm{O}(2^n)$;
(b) $2^{n+1} = \Omega(2^n)$; *(e)* $2^{2n} = \Omega(2^n)$;
(c) $2^{n+1} = \Theta(2^n)$; *(f)* $2^{2n} = \Theta(2^n)$.

Põhjendada oma vastuseid ka lõigus 5.1.3 toodud omaduste abil.

Ülesanne 5.2 *Vaatleme järgmisi funktsioone:*

$\sqrt{n}$	n	2^n	$n \log n$
$n - n^3 + 7n^5$	$n^2 + \log n$	n^2	n^3
$\log n$	$n^{1/3} + \log n$	$(\log n)^2$	$\log_2 n$
$\log \log n$	$(1/3)^n$	$(3/2)^n$	6

Grupeerida need funktsioonid nii, et ühes grupis oleks koos sama kasvukiirusega funktsioonid. Järjestada grupid omavahel funktsioonide kasvukiiruste kasvamise järjekorras. (Kõigi nende funktsioonide kasvukiirused on omavahel võrreldavad.)

Ülesanne 5.3 *Vaatleme järgmist algoritmi:*

```
– n ∈ ℕ
korda i ← 1 . . . n
   korda j ← 1 . . . i
      väljasta i, j
   lõppkorda
lõppkorda
```

Milline on selle algoritmi ajaline keerukus parimal, halvimal ja keskmisel juhul? Kas teie antud hinnangud on täpsed või liiaga? Põhjendada oma vastuseid.

Ülesanne 5.4 *Vaatleme algoritmi* BITTE *naturaalarvu kahendkujus esinevate 1-bittide loendamiseks (kus* div *ja* mod *tähendavad täisarvulise jagamise jagatist ja jääki):*

```
– n ∈ ℕ
kui n = 0
   tagasta 0
muidu
   tagasta BITTE(n div 2) + (n mod 2)
lõppkui
```

Milline on selle algoritmi ajaline keerukus?

Kirjandus

Jüri Kiho. *Algoritmid ja andmestruktuurid.* Tartu Ülikool, 2003.
Andmestruktuuride realiseerimist ja klassikalisi algoritme käsitlev õpik, milles leidub ka keerukuse põhimõistetele pühendatud peatükk. 148 lk.

Valdo Praust. *Keerukusteooria alused.* Ülikoolide Informaatikakeskus, 1996.
Algoritmide ja ülesannete keerukuse teoreetilisemale poolele keskenduv õpik. 212 lk.

Thomas H. Cormen, Clifford Stein, Charles Leiserson, Ronald Rivest. *Introduction to Algorithms.* MIT, 2001.
Eelmise trükiga klassikaks saanud algoritmide ja andmestruktuuride põhiõpik, mis annab üsna hea ülevaate ka algoritmide keerukuse hindamisest ja selleks vajalikust matemaatilisest aparatuurist. 1180 lk.

Томас Кормен, Чарльз Лейзерсон, Рональд Ривест. *Алгоритмы: построение и анализ.* МЦНМО, 2001.
Eelmise tõlge. 960 lk.

Herbert S. Wilf. *Algorithms and Complexity.* Prentice Hall, 1986.
Ülikooli keskmiste kursuste tasemele orienteeritud konspekt. 240 lk.
`http://www.cis.upenn.edu/~wilf/AlgComp2.html`

Peatükk 6

Kombinatoorika

Sisukord

Kombinatoorika on matemaatika haru, mis uurib etteantud hulga elementide kombineerimise võimalusi. Kuna selliste võimaluste arv on sageli väga suur, tuleb kõigi variantide töötlemist nõudvad ülesanded arvutile usaldada. Käesolevas peatükis vaatlemegi põhilisi kombinatoorseid objekte ja nendega seotud algoritme.

6.1 Kombinatoorika põhimõisted

Kombinatoorseid algoritme võib liigitada mitme tunnuse alusel. Kaks loomulikumat liigitust on töödeldavate objektide liikide ja nende objektidega sooritatavate tegevuste alusel.

6.1.1 Kombinatoorsed objektid

Kombineeritavate elementide hulka nimetatakse **põhihulgaks** (ingl *base set*). Üldiselt võivad selle elementideks olla mistahes objektid, aga arvutis on muidugi kõige mugavam kasutada arve ja seepärast eeldame edaspidi, et m-elemendiline põhihulk on alati $M = \{1, \ldots, m\}$. See pole kuigi oluline kitsendus, sest kui mõnes praktilises ülesandes on vaja kombineerida muid objekte, võime tegelikke objekte hoida eraldi massiivis ja kasutada hulga M elemente indeksitena objektide sellest massiivist leidmisel.

Lihtsaim kombinatoorne objekt on **järjend** ehk **ennik** (ingl *tuple*), mis koosneb fikseeritud arvust põhihulga elemendidest. Kaht järjendit loetakse võrdseks, kui nende pikkused ja vastavatel positsioonidel olevad elemendid on samad.

Näiteks põhihulga $M = \{1, 2, 3\}$ korral saame moodustada 9 erinevat 2-elemendilist järjendit (ehk lühemalt 2-järjendit): $(1, 1)$, $(1, 2)$, $(1, 3)$, $(2, 1)$, $(2, 2)$, $(2, 3)$, $(3, 1)$, $(3, 2)$ ja $(3, 3)$.

Permutatsiooni (ingl *permutation*) ehk **ümberjärjestuse** puhul nõutakse lisaks, et põhihulga iga element võib permutatsioonis esineda maksimaalselt ühe korra.

Eelmise näite põhihulga korral on erinevaid 2-elemendilisi permutatsioone (ehk 2-permutatsioone) kokku 6: $(1, 2)$, $(1, 3)$, $(2, 1)$, $(2, 3)$, $(3, 1)$ ja $(3, 2)$.[1]

Ka **kombinatsioon** (ingl *combination*) võib, sarnaselt permutatsiooniga, sisaldada põhihulga iga elementi maksimaalselt ühes eksemplaris. Erinevus on selles, et kaks kombinatsiooni loetakse erinevateks ainult juhul, kui nende elementide hulgad (järjestusest hoolimata) on erinevad.

[1] Mõnedes vanemates raamatutes nimetatakse m-elemendilise põhihulga korral permutatsioonideks ainult m-permutatsioone ja $k < m$ korral räägitakse k-variatsioonidest.

Kahes eelmises näites vaadeldud põhihulga korral on erinevaid 2-kombinatsioone vaid 3: $\{1,2\}$, $\{1,3\}$ ja $\{2,3\}$.[2]

Hulga **tükelduseks** (ingl *partition*) nimetatakse selle jagamist mittetühjadeks alamhulkadeks nii, et põhihulga iga element kuulub täpselt ühte alamhulka. Kaht tükeldust loetakse erinevateks, kui leiduvad kaks elementi, mis ühes tükelduses on samas alamhulgas, aga teises erinevates.

Näiteks 3-elemendilisel põhihulgal on 5 erinevat tükeldust: $\{\{1,2,3\}\}$ ("tükeldus" koosneb ühest tükist); $\{\{1,2\},\{3\}\}$, $\{\{1,3\},\{2\}\}$, $\{\{2,3\},\{1\}\}$ (tükeldused kaheks tükiks) ja $\{\{1\},\{2\},\{3\}\}$ (tükeldus kolmeks tükiks).

Kuigi sellega pole kombinatoorsete objektide liigid kaugeltki ammendatud, on meil nüüdseks juba piisavalt näiteid, et vaadelda lähemalt, millised on põhilised kombinatoorikaülesanded.

6.1.2 Kombinatoorsed ülesanded

Üks ilmsemaid kombinatoorikaülesandeid on kõigi vastavat liiki objektide **genereerimine** (ingl *generation*) ehk **loetlemine** (ingl *enumeration*)[3], see tähendab kõigi objektide nimekirja koostamine.

Vahel pole meil aga vaja mitte objekte endid, vaid ainult nende arvu. Muidugi võib **loendamise** (ingl *counting*) ülesande (vähemalt teoreetiliselt) alati taandada genereerimisele — kui meil on olemas meid huvitavate objektide nimekiri, võime nende arvu teadasaamiseks nad vahetult üle lugeda. Aga sageli on objektide arvu võimalik leida ka oluliselt efektiivsemalt.

Kombinatoorse **otsimise** (ingl *searching*) ülesandes nõutakse kõigi objektide hulgast ainult teatud tingimusele vastava(te) objekti(de) leidmist. Kui tingimuseks on mingi hinnangufunktsiooni minimeerimine või maksimeerimine, nimetatakse seda ülesannet ka kombinatoorse **optimeerimise** (ingl *optimization*) ülesandeks.

[2]Kuna kombinatsioonid on põhihulga alamhulgad, märgitaksegi neid tavaliselt hulkadena, andes elementide loetelu ümarsulgude asemel looksulgudes.

[3]NB! Inglise keeles kasutatakse sõna 'enumerate' nii 'loetlema' kui ka 'loendama' tähenduses.

6.2 Loendamine

Ajalooliselt olid loendamisülesanded kombinatoorika üks esimesi praktilise väärtusega rakendusi — loendamisel põhineb tõenäosustooria, ja sellel omakorda põhinevad hasartmängud.

Lihtsamate ja korrapärasemate objektide korral on loendamisülesanded sageli lahendatavad ka arvuti abita. Arvutusvahendist sõltumata on loendamisel väga kasulikud kaks põhireeglit: korrutamisreegel ja liitmisreegel.

6.2.1 Korrutamisreegel

Korrutamisreegel ütleb, et kui meil on vaja teha järjest kaks valikut ning esimese valiku tegemiseks on v_1 võimalust ja iga esimese valiku järel on teise valiku tegemiseks v_2 võimalust, siis nende kahe valiku tegemiseks on kokku $v_1 \cdot v_2$ võimalust.

Selle reegli põhjal on lihtne leida m-elemendilise põhihulga n-järjendite arv: järjendi iga elemendi valimiseks on meil täpselt m võimalust, seega on kõigi n elemendi valimiseks kokku

$$T(m,n) = \underbrace{m \cdot m \cdot \ldots \cdot m}_{n} = m^n$$

võimalust.

Ka permutatsioonide arv on leitav korrutamisreegli abil: permutatsiooni esimese elemendi valimiseks on meil kogu põhihulk, seega m võimalust; teiseks elemendiks ei tohi me valida juba valitud elementi, seega jääb $m-1$ võimalust; kolmandaks ei tohi me valida kumbagi juba valitud elementi, seega jääb $m-2$ võimalust. Samal moel jätkates saame, et m-elemendilise põhihulga n-permutatsioonide arv on

$$P(m,n) = m \cdot (m-1) \cdot \ldots \cdot (m-(n-1)) = \frac{m!}{(m-n)!}, \tag{6.1}$$

millest erijuhul $n = m$ saame m-permutatsioonide arvuks

$$P(m) = P(m,m) = m!. \tag{6.2}$$

Korrutamisreeglit võib kasutada ka tagurpidi, jagamisreeglina: kui on teada, et kahe järjestikuse valiku tegemise võimaluste koguarv on v ja et mistahes esimese valiku järel on teist valikut võimalik teha v_2 erineval viisil, saame sellest avaldada esimese valiku tegemise võimaluste arvu $v_1 = v/v_2$.

Näiteks kombinatsioonide arvu leidmine on jagamisreegli abil tunduvalt mugavam kui korrutamisreegli abil. Korrutamisreegli kasutamist raskendab

asjaolu, et sama kombinatsiooni korduva loendamise vältimiseks tuleks elemente kombinatsiooni lisada mingis kindlas järjekorras (näiteks kasvavas). Siis aga sõltub teise elemendi valimise võimaluste arv sellest, millise elemendi me esimeseks valisime ja korrutamisreeglit ei saa enam rakendada.

Hoopis lihtsam on lähtuda tähelepanekust, et igast n-kombinatsioonist (mis on lihtsalt põhihulga n-elemendiline alamhulk) saab valemi 6.2 kohaselt moodustada $P(n) = n!$ erinevat n-permutatsiooni. Pole raske veenduda, et nii saame koostada kõik võimalikud n-permutatsioonid, ja seejuures igaühe täpselt ühe korra. Valemi 6.1 ja jagamisreegli põhjal saame seega n-kombinatsioonide arvuks

$$C(m,n) = \frac{P(m,n)}{n!} = \frac{m!}{n! \cdot (n-m)!} = \binom{m}{n}. \tag{6.3}$$

6.2.2 Liitmisreegel

Liitmisreegel ütleb, et kui meil on vaja teha üks kahest valikust ning esimese valiku tegemiseks on v_1 võimalust ja teise tegemiseks v_2 võimalust, siis neist kahest valikust ühe tegemiseks on kokku $v_1 + v_2$ võimalust. Seejuures on oluline, et nende kahe valiku võimaluste hulgad peavad olema ühisosata.

Selle reegli abil on lihtne leida rekurrentne valem m-elemendilise põhihulga n-kombinatsioonide arvu loendamiseks: kõik n-kombinatsioonid võime jagada kaheks selle põhjal, kas nad sisaldavad või ei sisalda elementi m.

Igast elementi m sisaldavast n-kombinatsioonist saame selle eemaldamisega $(m-1)$-elemendilise põhihulga $(n-1)$-kombinatsiooni. Iga elementi m mittesisaldav n-kombinatsioon on automaatselt ka $(m-1)$-elemendilise põhihulga n-kombinatsioon.

Kuna need variandid on teineteist välistavad (ei ole võimalik, et mingi kombinatsioon kuulub mõlemasse alamhulka) ja katavad kõik võimalused (ei ole võimalik, et mingi kombinatsioon ei kuulu kumbagi alamhulka), siis saamegi liitmisreegli põhjal, et $C(m,n) = C(m-1,n-1) + C(m-1,n)$.

Lisades sellele valemile veel ääretingimused $C(m,0) = 1$ (ainus 0-kombinatsioon on tühi hulk) ja $C(m,m) = 1$ (ainus m-kombinatsioon on põhihulk ise), ongi käes rekursiivse programmi koostamiseks piisav tulemus

$$C(m,n) = \begin{cases} 1, & \text{kui } n = 0, \\ 1, & \text{kui } n = m, \\ C(m-1,n-1) + C(m-1,n), & \text{kui } 0 < n < m. \end{cases} \tag{6.4}$$

Kuna valemid 6.3 ja 6.4 loendavad samasid objekte, siis on igati ootuspärane, et nende alusel kirjutatud programmid annavad ka ühesuguseid tulemusi.

Hulga tükelduste loendamine on eelmistest veidi tülikam ülesanne. Selle lahendamiseks tuleb liitmisreeglit rakendada kaks korda.

Esmalt paneme tähele, et m-elemendilise hulga kõigi tükelduste arvud $B(m)$ (ehk Belli arvud) avalduvad n tükiks tükelduste arvude $S(m,n)$ (ehk Stirlingi teist liiki arvude) kaudu:

$$B(m) = \sum_{n=1}^{m} S(m,n).$$

Liitmisreegli veelkordse kasutamisega saame rekurrentse võrrandi Stirlingi arvude avaldamiseks, arvestades, et m-hulga tükeldused on saadavad $(m-1)$-hulga tükeldustest järgmiselt: igast $(m-1)$-hulga $(n-1)$-tükeldusest saame m-hulga n-tükelduse, kui lisame sellele eraldi tükina elemendi m; igast $(m-1)$-hulga n-tükeldusest saame n erinevat m-hulga n-tükeldust, kui lisame elemendi m kordamööda igaühele olemasolevast n tükist. Pole raske veenduda, et saadud variandid on kõik erinevad ja katavad ühtlasi ka kõik võimalikud m-hulga n-tükeldused. Seega saame nende koguarvuks $S(m,n) = n \cdot S(m-1,n) + S(m-1,n-1)$.

Lisades sellele valemile ääretingimused $S(m,1) = 1$ (ainus 1-elemendiline tükeldus on põhihulk ise) ja $S(m,m) = 1$ (ainus m-tükeldus on selline, kus iga element on omaette alamhulk), ongi käes programmeerimiseks kõlblik

$$S(m,n) = \begin{cases} 1, & \text{kui } n = 1, \\ 1, & \text{kui } n = m, \\ n \cdot S(m-1,n) + S(m-1,n-1), & \text{kui } 1 < n < m. \end{cases}$$

Ka liitmisreeglit on võimalik pöörata lahutamisreegliks.

Näiteks selleks, et leida 1 ja 100 vahele jäävate kolmega mittejaguvate täisarvude arv, on lihtsam leida kõigepealt kolmega jaguvate täisarvude arv (mis on loomulikult 33) ja siis järeldada, et kõik ülejäänud $100 - 33 = 67$ arvu peavad olema kolmega mittejaguvad.

6.2.3 Elimineerimismeetod

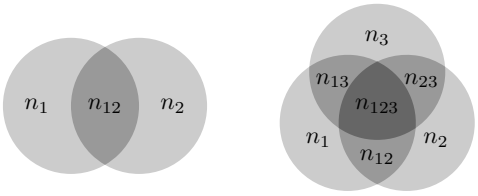
Elimineerimismeetod (ingl *inclusion-exclusion principle*) aitab leevendada liitmisreegli (praktikas üsnagi ahistavat) nõuet, et valikuvariantide hulgad peavad olema ühisosata.

Näiteks 1 ja 100 vahele jäävate kolme või viiega jaguvate arvude loendamiseks liitmisreeglit kasutada ei saa, sest need variandid pole teineteist välistavad — leidub ka arve, mis jaguvad nii kolme kui viiega.

Alamhulkade lõikumatuse nõude ignoreerimisel saame tõesti vale vastuse: 1 ja 100 vahel on kolmega jaguvaid arve 33 ja viiega jaguvaid 20, kuid kolme või viiega jaguvaid 47, mitte $33 + 20 = 53$.

Vea põhjuseks on muidugi asjaolu, et arve, mis jaguvad nii kolme kui viiega, loeme topelt — ühe korra kui kolmega, teise korra kui viiega jaguvaid. Topelt loeme seega kõiki 15-ga jaguvaid arve, mida uuritaval lõigul on 6. Õige vastuse saamiseks tuleb topelt loetud variantide arv lõppsummast maha lahutada: $33 + 20 - 6 = 47$.

Sama kehtib mistahes kahe tunnusega objektide loendamisel (joonisel 6.1 vasakul): kui n_1 objektil on esimene tunnus (ja võib-olla samaaegselt ka teine) ja n_2 objektil on teine tunnus (ja võib-olla samaaegselt ka esimene) ning n_{12} objektil on mõlemad tunnused, siis vähemalt üks neist kahest tunnusest on kokku $n = n_1 + n_2 - n_{12}$ objektil.



Joonis 6.1: Elimineerimismeetod

Kolme erineva tunnuse korral (joonisel 6.1 paremal) oleme ühe tunnusega objektide arvude liitmisel ja kahe tunnusega objektide arvude lahutamisel kõigi kolme tunnusega objektide arvu lisanud summasse kolm korda (n_1, n_2 ja n_3 hulgas) ja ka lahutanud kolm korda (n_{12}, n_{13} ja n_{23} hulgas), seega on need objektid sisuliselt loendamata ja õige tulemuse saamiseks tuleb neid eraldi arvestada (n_{123}).

Näiteks 1 ja 100 vahele jäävate kahe, kolme või viiega jaguvate arvude loendamiseks peame arvestama, et: kahega jaguvaid arve on 50, kolmega jaguvaid 33, viiega jaguvaid 20; kahe ja kolmega jaguvaid 16, kahe ja viiega jaguvaid 10, kolme ja viiega jaguvaid 6; kõigi kolmega jaguvaid 3. Seega on kahe, kolme või viiega jaguvaid arve kokku $50+33+20-16-10-6+3 = 74$.

Üldiselt, t erineva tunnuse korral on vähemalt ühe tunnusega objektide koguarv

$$N = -\sum_{i=1}^{t}(-1)^i \cdot N_i,$$

kus N_i tähistab vähemalt i tunnusega objektide arvude summat, kusjuures iga i korral tuleb vaadelda kõiki võimalikke i erineva tunnuse kombinatsioone.

6.3 Genereerimine

Genereerimisel, erinevalt loendamisest, peame mõtlema ka sellele, millises järjekorras me tulemust saada tahame.

Elementide loeteluna esitatavate kombinatoorsete objektide (järjendid, permutatsioonid, kombinatsioonid) puhul peetakse üldiselt standardseimaks **leksikograafilist** (ingl *lexicographical*) järjestust — kahe objekti võrdlemisel jäetakse vahele need elemendid kummagi objekti algusest, mis on omavahel võrdsed ja otsus langetatakse esimese erineva elemendi põhjal.

Mõnikord on kasulikum objekte genereerida **minimaalse muutuse** (ingl *minimal change*) järjestuses — tulemuste hulgas järjest olevate objektide erinevus peab olemas minimaalne. Minimaalsuse definitsioon sõltub muidugi konkreetsest objektist ja vahel ka sellest, milleks neid objekte kasutatakse.

Näiteks permutatsioone on võimalik järjestada nii, et iga järgmine on saadav eelmisest kahe kõrvutioleva elemendi vahetamise teel, järjendeid aga nii, et iga järgmine erineb eelmisest ainult ühes positsioonis ühe võrra.

Leksikograafiline	Minimaalse muutuse
$(1,2,3)$	$(1,2,3)$
$(1,3,2)$	$(1,3,2)$
$(2,1,3)$	$(3,1,2)$
$(2,3,1)$	$(3,2,1)$
$(3,1,2)$	$(2,3,1)$
$(3,2,1)$	$(2,1,3)$

Tabel 6.1: Hulga $\{1,2,3\}$ permutatsioonide järjestusi

Leksikograafiline	Minimaalse muutuse
$(1,1,1)$	$(1,1,1)$
$(1,1,2)$	$(1,1,2)$
$(1,2,1)$	$(1,2,2)$
$(1,2,2)$	$(1,2,1)$
$(2,1,1)$	$(2,2,1)$
$(2,1,2)$	$(2,2,2)$
$(2,2,1)$	$(2,1,2)$
$(2,2,2)$	$(2,1,1)$

Tabel 6.2: Hulga $\{1,2\}$ 3-järjendite järjestusi

Edasi tegeleme objektide genereerimisega leksikograafilises järjekorras.

6.3.1 Tagurdusmeetod

Väga üldine meetod igasugusteks variantide genereerimiseks on **tagurdusmeetod** (ingl *backtracking*). Selle põhiideed illustreerib näide 6.1: tegemist on rekursiivse algoritmiga, mis saab osaliselt valmis genereeritud variandi (parameeter v), leiab kõik võimalikud sammud selle täiendamiseks (loetelu w_1 kuni w_n) ja proovib jätkata genereerimist iga võimaliku täiendatud variandiga (abimuutuja v'). Kui mõni täiendatud variantidest osutub täielikuks (tingimus real 1), kasutame selle ära (näiteks väljastame) ja jätkame uute variantide genereerimist.

Tagurdusmeetodi kasutamiseks tuleb selle esimesel väljakutsel (kas põhiprogrammist või mõnest teisest alamprogrammist) anda kaasa **seeme** (ingl *seed*) — lähtepunkt, millest kõigi teiste konstrueerimist alustada. Tavaliselt on seemneks mingis mõttes tühi variant.

Algoritm 6.1 GenRek: Variantide genereerimine rekursiivselt
Sisend: v — osaline variant
Väljund: leiab kõik võimalused v täiendamiseks

1. `kui` v on täielik variant
   — kasutame v
2. `muidu`
   — vatleme kõiki v täiendamise võimalusi
3. `korda` $w \in w_{1\ldots n}$
4. $v' \leftarrow v + w$
   — invariant: v' on suurem osaline variant
5. GenRek(v')
   — tagurdus: proovime v järgmist täiendusvõimalust
6. `lõppkorda`
7. `lõppkui`

Väga lihtne on tagurdusmeetodil genereerida näiteks järjendeid (algoritm 6.2): n-järjendi genereerimisesl on osaliseks variandiks mingi k-järjend (kus $k \leq n$) ja täiendamise üks samm on elemendi a_{k+1} lisamine antud k-järjendi lõppu.

Nagu järjendite loendamisel juba mainitud, on selle sammu tegemiseks m võimalust — uueks elemendiks võime ilma piiranguteta valida ükskõik millise põhihulga elemendi.

Seemneks on tühi (0-elemendiline) järjend.

Algoritm 6.2 JarjRek: Järjendite genereerimine rekursiivselt
Sisend: $a_{1\ldots k}$ on k-järjend

1. `kui` $k = n$
   — $a_{1\ldots n}$ on leitud n-järjend
2. `muidu`
3. `korda` $a_{k+1} \leftarrow 1 \ldots m$
   — invariant: $a_{1\ldots k+1}$ on $(k+1)$-järjend
4. JarjRek($a_{1\ldots k+1}$)
5. `lõppkorda`
6. `lõppkui`

Permutatsioonide genereerimine (algoritm 6.3) erineb järjendite genereerimisest ainult permutatsiooni definitsioonist lähtuva kitsenduse võrra: etteantud k-permutatsiooni lõppu lisatava elemendi valimisel peame kontrollima, et see element permutatsioonis eespool juba kasutusel ei ole.

Selle kontrolli tõhusaks realiseerimiseks kasutatakse tavaliselt globaalset massiivi $v_{1\ldots m}$, mille element v_i näitab, kas põhihulga element i on veel vaba. Siis taandub real 4 oleva tingimuse kontrollimine selle massiivi ühe elemendi väärtuse uurimisele, kuid selle eest peame enne rekursiivset väljakutset real 5 elemendi a_{k+1} jooksva väärtuse kasutatuks märkima ja pärast rekursioonist naasmist selle elemendi jälle vabastama.

Algoritm 6.3 PermRek: Permutatsioonide genereerimine rekursiivselt
Sisend: $a_{1\ldots k}$ on k-permutatsioon

1. `kui` $k = n$
   — $a_{1\ldots n}$ on leitud n-permutatsioon
2. `muidu`
3. `korda` $a_{k+1} \leftarrow 1 \ldots m$
4. `kui` $a_{k+1} \notin a_{1\ldots k}$
   — invariant: $a_{1\ldots k+1}$ on $(k+1)$-permutatsioon
5. PermRek($a_{1\ldots k+1}$)
6. `lõppkui`
7. `lõppkorda`
8. `lõppkui`

Kombinatsioonide genereerimisel (algoritm 6.4) on kasulik uusi elemente kombinatsiooni lisada kasvavas järjekorras — nii on lihtne hoiduda sama elemendi korduvast lisamisest ja on välistatud ka samade elementide ümberjärjestuste eksikombel erinevateks kombinatsioonideks lugemine.

Algoritm 6.4 KombRek: Kombinatsioonide genereerimine rekursiivselt
Sisend: $a_{1\ldots k}$ on k-kombinatsioon
Abimuutujad: x — vähim vaba element

1. `kui` $k = n$
   — $a_{1\ldots n}$ on leitud n-kombinatsioon
2. `muidu`
3. `kui` $k = 0$
4. $x \leftarrow 1$
5. `muidu`
6. $x \leftarrow a_k + 1$
7. `lõppkui`
8. `korda` $a_{k+1} \leftarrow x \ldots m$
   — invariant: $a_{1\ldots k+1}$ on $(k+1)$-kombinatsioon
9. KombRek($a_{1\ldots k+1}$)
10. `lõppkorda`
11. `lõppkui`

6.3.2 Iteratsioonimeetod

Teine üldisem võimalus on genereerida objekte iterativselt.

Iteratiivse generaatori struktuur on toodud näites 6.5: kõigepealt luuakse esimene otsitav objekt vahetult ja edasi leitakse igal sammul jooksva objekti põhjal järgmine.

Algoritm 6.5 GenIte: Variantide genereerimine iteratiivselt
Abimuutujad: v — jooksev variant

1. $v \leftarrow$ EsimeneVariant
2. `senikui` $v \neq \emptyset$
   — kasutame v
3. $v \leftarrow$ JargmineVariant(v)
4. `lõppsenikui`

Nagu üldiselt iteratiivsete generaatorite puhul, nii on ka järjendite genereerimisel põhiline jooksva järjendi põhjal uue leidmine (algoritm 6.6): suurendada tuleb kõige parempoolsemat positsiooni, mille suurendamine on võimalik; kõik sellest positsioonist paremal pool olevad peavad olema maksimaalse võimaliku väärtusega (muidu oleks ju võimalik suurendada mõnda neist) ja järgmises järjendis peavad nende väärtused jälle vähimast alustama.

(Sisuliselt on tegemist ühe liitmisega n-kohalisele m-süsteemi arvule. Tõenäoliselt oleks seda kergem märgata, kui põhihulk oleks $\{1 \ldots m\}$ asemel $\{0 \ldots m-1\}$.)

Algoritm 6.6 JRGMJARJ
Sisend: $a_{1\ldots n}$ on n-järjend
Väljund: leksikograafiliselt järgmine n-järjend
Abimuutujad: $u \in \{0,1\}$ — jooksev ülekanne

1. $u \leftarrow 1$
2. `korda` $i \leftarrow n \ldots 1$
3. $\quad a_i \leftarrow a_i + u$
4. $\quad$ `kui` $a_i > m$
5. $\quad\quad a_i \leftarrow 1$
6. $\quad$ `muidu`
7. $\quad\quad u \leftarrow 0$
8. $\quad$ `lõppkui`
9. `lõppkorda`
10. `kui` $u = 0$
11. $\quad$ `tagasta` v
12. `muidu`
13. $\quad$ `tagasta` $\emptyset$
14. `lõppkui`

6.4 Otsimine

Üldiselt on otsimis- ja optimeerimisülesanded kombinatoorikas kõige loomingulisemad. Põhiline lahendusmeetod on variantide genereerimine rekursiivselt, kusjuures algoritmi töö kiirendamiseks püütakse võimalikult vara aru saada, kui parasjagu käsilolevat osalist varianti ei ole võimalik kõiki ülesande tingimusi rahuldavaks täielikuks variandiks välja arendada.

Otsimisülesande näitena vaatleme ülesannet paigutada N lippu $N \times N$ malelauale nii, et mitte ükski neist poleks ühegi teise tule all.

Esiteks paneme kohe tähele, et iga lipp peab laual olema eraldi real — samal real olevad lipud tulistaks üksteist kindlasti. See aga tähendab, et me võime kõik potentsiaalsed lahendused märkida üles kujul $a_{1\ldots N}$, kus a_k näitab k. real oleva lipu veerunumbrit.

Edasi võime hakata lippude paigutuse variante järjest läbi proovima. Seda on kasulik teha tagurdusmeetodil, sest nii saame k. lipu paigutamisel kohe kontrollida, et see ei jää juba varem laual olevate lippude $(1 \ldots k-1)$ tule alla ja mitte kulutada aega ülejäänud lippude $(k+1 \ldots N)$ paigutamisele, kuna seisu muudab kõlbmatuks ka üksainus tule alla jäänud lipp. Lahenduse töökiiruse seisukohalt on oluline, kuidas me hoiame infot tule all olevate väljade kohta.

Üks võimalus on seda infot eraldi üldse mitte hoida ja võrrelda iga uue lipu paigutamisel selle asukohta laual kõigi varem paigutatud lippude omadega — iga lipu korral tuleb kontrollida, kas uus on temaga samas veerus või samal diagonaalil, seega kulutame k. lipu iga positsiooni kontrollimiseks $O(k)$ operatsiooni.

Teine võimalus on iga lipu lauale paigutamisel märkida ära kõik väljad, mida see lipp tule all hoiab. Siis on küll võimalik uuele lipule koha valimisel selle koha kõlblikkuse või kõlbmatuse üle väga kiiresti otsustada, aga lipu lauale paigutamine ja selle sealt eemaldamine muutuvad ebaefektiivseks — $N \times N$ laual võib lipp tulistada maksimaalselt $4 \cdot N - 3$ välja.

Kõige parema lahenduse saame, pannes tähele, et laual võib olla ülimalt üks lipp igas veerus ning igal tõusval ja igal langeval diagonaalil. Jääb veel leida efektiivne viis neid kolme liiki objekte hõivatuks ja vabastatuks märkida (nagu põhihulga elementide haldamisel permutatsioonide genereerimisel).

Veergude haldamine on muidugi lihtne — selleks võime kasutada massiivi $v_{1\ldots N}$. Aga osutub, et ka diagonaalidega pole palju rohkem muret. Nimelt on igal tõusval diagonaalil rea- ja veerunumbri vahe ühe diagonaali piires sama ja erinevatel diagonaalidel erinev — seega võib nende olekute hoidmiseks kasutada massiivi $td_{1-N\ldots N-1}$. Langeva diagonaali identifitseerimiseks sobib rea- ja veerunumbri summa — seega võib nende olekute hoidmiseks kasutada massiivi $ld_{2\ldots 2\cdot N}$.

Algoritm 6.7 LIPUD: Lippude paigutamine malelauale
Sisend: r — rida, millele lippu paigutame

1. `kui` $r > n$
   — on leitud n lipu paigutus
2. `muidu`
3. `korda` $x \leftarrow 1 \ldots N$
4. `kui` $v_x = vaba \wedge td_{r-x} = vaba \wedge tl_{r+x} = vaba$
   — invariant: positsioon (r, x) on vaba
5. $v_x \leftarrow kinni$; $td_{r-x} \leftarrow kinni$; $td_{r+x} \leftarrow kinni$
6. LIPUD$(r + 1)$
7. $v_x \leftarrow vaba$; $td_{r-x} \leftarrow vaba$; $td_{r+x} \leftarrow vaba$
8. `lõppkui`
9. `lõppkorda`
10. `lõppkui`

Optimeerimisülesannete puhul on sageli kasulik hoida eraldi infot parima seni leitud lahenduse kohta ja hinnata iga variandi töötlemisel, kas on üldse võimalik, et selle edasiarendamisel võiks saada parema lahenduse. Kui see pole võimalik, pole ka mõtet seda varianti edasi töödelda.

Ülesanded

Ülesanne 6.1 *Kui palju on standardses 52-kaardilises pakis (kaks punast ja kaks musta masti, igas mastis 9 numbri- ja 4 pildilehte) kaarte, mis on:*

(a) punased?
(b) numbrid?
(c) punased ja numbrid?
(d) punased ja mitte numbrid?
(e) punased või numbrid?
(f) punased või numbrid, kuid mitte mõlemat korraga?

Vastata ilma kaarte vahetult loendamata. Põhjendada kõiki oma vastuseid.

Ülesanne 6.2 *Tavalist 6-tahulist täringut visatakse viis korda. Mitmel erineval viisil on võimalik saada viie viske summas paarisarv silmi? Põhjendada oma vastust.*

Ülesanne 6.3 *Kirjutada iteratiivne programm m-hulga m-permutatsioonide genereerimiseks leksikograafilises järjekorras.*

Ülesanne 6.4 *Kirjutada iteratiivne programm m-hulga n-kombinatsioonide genereerimiseks leksikograafilises järjekorras.*

Ülesanne 6.5 *Põhihulga $M = \{1, \ldots, m\}$* ***korratuseks*** *(ingl derangement) nimetatakse m-permutatsiooni $P = (p_1, p_2, \ldots, p_m)$, milles ükski element ei asu "oma õigel kohal", s.t. $\forall i \in M : p_i \neq i$. Tõestada korratuste arvude $D(m)$ võrrand $D(m) = (m-1) \cdot (D(m-1) + D(m-2))$. Määrata vajalikud ääretingimused ja kirjutada programm korratuste loendamiseks.*

Ülesanne 6.6 *Kirjutada rekursiivne programm korratuste genereerimiseks leksikograafilises järjekorras.*

Ülesanne 6.7 *Kirjutada iteratiivne programm korratuste genereerimiseks leksikograafilises järjekorras.*

Ülesanne 6.8 *Kirjutada programm, mis paigutab $N \times N$ lauale maksimaalse hulga maleratsusid nii, et mitte ükski neist poleks ühegi teise tule all.*

Kirjandus

Reimo Palm. *Diskreetse matemaatika elemendid.* Tartu Ülikool, 2003.
Sissejuhatus diskreetsesse matemaatikasse, sealhulgas ka kombinatoorika alustesse. Matemaatilisema kallakuga. 160 lk.

Donald L. Kreher, Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, 1999.
Kombinatoorika algoritmide õpik. 330 lk.
Aadressil `http://www.math.mtu.edu/~kreher/cages.html` on muu hulgas ka algoritmide realisatsioonid C's, kuid ilma raamatu endata on neid väga raske kasutada.

Витольд Липский. *Комбинаторика для программистов.* Мир, 1988.
Eelmisest vanem ja veidi õhem, põhimõtteliselt üsna sarnane. 213 lk.

Ülo Kaasik, Uno Kaljulaid. *Kombinatoorika analüütilisi ja algebralisi meetodeid.* Tartu Ülikool, 1993.
Kombinatoorika loengukonspekt. Üsna tõsine matemaatika. 159 lk.

Dany Breslauer, Devdatt P. Dubhasi. *Combinatorics for Computer Scientists.* University of Aarhus, 1995.
Kombinatoorika loengukonspekt. Tõsine matemaatika. 330 lk.
`http://www.brics.dk/BRICS/LS/95/4/BRICS-LS-95-4.ps.gz`

Peatükk 7

Lineaarsed andmestruktuurid

Sisukord

Käesolevas peatükis tutvume andmestruktuuri mõistega ning vaatleme tähtsamaid lineaarseid andmestruktuure ja nendega seotud algoritme.

7.1 Andmestruktuuri mõiste

Andmestruktuur (ingl *data structure*) on vahend suure hulga (tavaliselt samatüübiliste) andmete hoidmiseks ja neile efektiivse juurdepääsu korraldamiseks.

Käesolevas peatükis vaadeldavaid struktuure ühendab asjaolu, et nende sisu saab esitada lineaarse elementide loendina. Seetõttu kasutataksegi nende struktuuride kohta ühist nimetajat "lineaarne".

7.2 Madalama taseme struktuurid

Esimeste struktuuridena võtame vaatluse alla massiivi ja lihtahela. Neid võib pidada madalama taseme struktuurideks, kuna neid kasutatakse sageli abivahendina keerulisemate ja abstraktsemate struktuuride realiseerimisel.

Muidugi ei maksa sellest järeldada, et pole rakendusi, kus just massiiv või lihtahel ongi sobivaimad struktuurid. Neid on piisavalt.

7.2.1 Massiiv

Nagu juba varem märgitud, on massiiv ühetüübiliste elementide kogum, kus iga elementi identifitseerib täisarvuline indeks (mitmemõõtmelise massiivi korral vastavalt üks indeks iga mõõtme kohta).

Massiivi kui andmestruktuuri iseloomustavad järgmised omadused:

- Massiivi suurus on tavaliselt fikseeritud. Keeltes, kus massiivi suurust pärast massiivi loomist üldse muuta saab, on see üsna ajamahukas operatsioon (oluliselt ei peta ettekujutus, et selleks tehakse uus massiiv, kopeeritakse vana massiivi kõik elemendid uude ümber ja kustutatakse vana massiiv mälust ära). Paljudes keeltes pole massiivi suuruse muutmine pärast selle loomist enam võimalik.

  Kui on siiski vaja massiivi omadustega andmestruktuuri, mille elementide arv pole kohe teada, on üks võimalus luua maksimaalse vaja minna võiva suurusega massiiv ja pidada eraldi muutuja abil arvet selle üle, kui suur osa massiivist tegelikult kasutusel on.

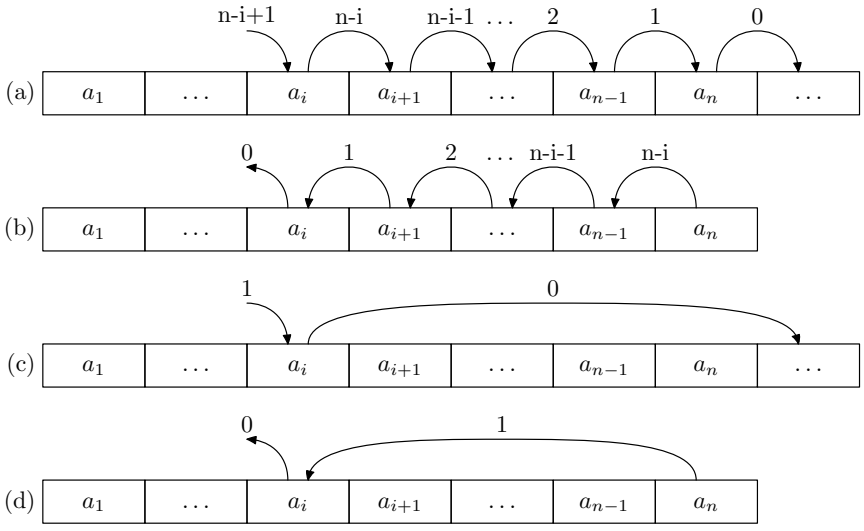
- Massiivi elemendi leidmine tema indeksi järgi on kiire operatsioon ja selleks tehtava töö maht ei sõltu massiivi suurusest. Selle operatsiooni

ajakulu sõltub tavaliselt massiivi mõõtmete arvust, aga on igal juhul tühine võrreldes kõigi keerulisemate struktuuridega.

- Massiivi elemendi väärtuse väljavahetamine (massiivi elemendile uue väärtuse omistamine) on tavaline omistamistehe ja selleks tehtava töö maht (kui element on indeksi järgi juba leitud) ei sõltu massiivi suurusest ega selle mõõtmete arvust.

- Vaadeldes muutuva suurusega massiive, on võimalik rääkida ka massiivi elementide lisamisest ja eemaldamisest.

  Kui kõigi olemasolevate elementide järjekord peab säilima, on massiivist elemendi eemaldamine või uue lisamine üsna kallis operatsioon. Elemendi lisamisel tuleb uuele elemendile teiste vahele ruumi tegemiseks kõiki temast tahapoole jäävaid elemente nihutada ühe koha võrra massiivi lõpu suunas (joonis 7.1 (a)). Olemasoleva elemendi eemaldamisel tekkinud tühimiku täitmiseks tuleb kõiki temast tahapoole jäävaid elemente nihutada massiivi alguse suunas (joonis 7.1 (b)). Nagu jooniselt näha, on n-elemendilise massiivi i. elemendi lisamisel või eemaldamisel selleks tehtava töö maht $\Theta(n - i)$.



Joonis 7.1: Elemendi massiivi lisamine ja massiivist eemaldamine

Kui olemasolevate elementide järjekorra säilimine pole oluline, saab nii uue elemendi antud positsioonile lisamise kui ka elemendi eemaldamise realiseerida konstantse töömahuga (joonis 7.1 (c) ja (d)).

Kui indeksi järgi elemendi leidmine on massiivis väga efektiivne standardoperatsioon, siis elemendi järgi indeksi leidmine (ehk antud väärtuse otsimine massiivi elementide hulgast) pole üldiselt kumbagi: üldjuhul ei saa seda teha efektiivselt ja seetõttu pole see operatsioon sageli ka programmikeele standardvahendite hulgas.

Ilma massiivi elementidele täiendavaid tingimusi kehtestamata on ainus võimalus antud väärtuse leidmiseks (või veendumiseks, et seda väärtust massiivi elementide hulgas üldse ei leidu) kõik massiivi elemendid järjest läbi vaadata. Sellist meetodit nimetatakse **lineaarseks otsinguks** (ingl *linear search*).

Algoritm 7.1 LinOtsV: Lineaarne otsing massiivis
Sisend: $a_{1\ldots n}$ — antud massiiv; x — otsitav väärtus
Väljund: tagastab leitud elemendi indeksi, või $n+1$, kui x massiivis ei esine
Abimuutujad: $i \in \mathbb{N}$ — uuritava elemendi indeks

1. `korda` $i \leftarrow 1 \ldots n$
   — invariant: $a_{1\ldots i-1}$ otsitavat väärtust ei sisalda
2. `kui` $a_i = x$
   — otsitav väärtus leitud, tagastame selle indeksi
3. `tagasta` i
4. `lõppkui`
5. `lõppkorda`
   — vahetingimus: $a_{1\ldots n}$ otsitavat väärtust ei sisalda
6. `tagasta` $n+1$

Muidugi pole raske näha, et selle algoritmi keerukus on parimal juhul $\Theta(1)$ (kui otsitav väärtus on a_1) ja halvimal juhul $\Theta(n)$ (kui otsitav väärtus on a_n või ei esine massiivis üldse).

Veidi keerukam on hinnata keskmist keerukust. Selleks peame kõigepealt tegema mingi eelduse erinevate väärtuste jaotumise kohta massiivis. Muude andmete puudumisel on kõige loomulikum eeldada, et otsitav väärtus võib võrdse tõenäosusega asuda ükskõik millises massiivi elemendis.

Seega, tõenäosusega $1/n$ leiame otsitava väärtuse elemendist a_1 juba pärast 1. võrdlust, tõenäosusega $1/n$ elemendist a_2 pärast 2. võrdlust, ..., tõenäosusega $1/n$ elemendist a_n pärast n. võrdlust. Selle eelduse kohaselt kulub M otsingu peale kokku

$$\frac{M}{n}1 + \frac{M}{n}2 + \ldots + \frac{M}{n}n = \frac{M}{n}(1 + 2 + \ldots + n) = \frac{M}{n}\frac{n(n+1)}{2} = M\frac{n+1}{2}$$

võrdlust, mis annab ühe otsingu keerukuseks keskmiselt

$$\frac{n+1}{2} = \Theta(n).$$

Järelikult kasvab lineaarseks otsinguks keskmiselt kuluv aeg võrdeliselt andmete mahuga.

Lineaarse otsingu suur ajakulu on tingitud eeskätt sellest, et iga võrdluse alusel saame edasise vaatluse alt välja jätta ainult ühe elemendi — selle, mida me just otsitava väärtusega võrdlesime.

Kui on teada, et massiivi elemendid on järjestatud, saame elemente kõrvale heita suuremate gruppidena. (Edasises eeldame, et massiivi elemendid on järjestatud mittekahanevalt: $\forall i < n : a_i \leq a_{i+1}$. Muidugi oleks arutelu sarnane ka vastupidiselt järjestatud massiivi korral.)

Sellises massiivis saame elemendi a_k otsitava väärtusega võrdlemise tulemusest teha järeldusi ka teiste elementide kohta. Tõepoolest, kui $a_k < x$, võime edasise vaatluse alt välja jätta mitte ainult a_k, vaid ka kõik a_i, kus $i < k$, sest ükski neist ei saa olla suurem kui a_k (joonis 7.2, vasakpoolne looksulg). Analoogiliselt, kui $a_k > x$, võime edasise vaatluse alt välja jätta ka kõik a_i, kus $i > k$ (joonis 7.2, parempoolne looksulg).

$$\overbrace{a_1 \leq \ldots \leq a_{k-1}}^{\leq a_k} \leq a_k \leq \overbrace{a_{k+1} \leq \ldots \leq a_n}^{\geq a_k}$$

Joonis 7.2: Kahendotsimise põhiidee

Nüüd jääb veel valida k väärtus, mis annaks maksimaalse võidu. Osutub, et kasulik on igal sammul valida veel vaatluse all olevatest elementidest keskmine — sellisel juhul saame iga võrdluse järel heita kõrvale pooled elemendid. Seda algoritmi nimetatakse **kahendotsinguks** (ingl *binary search*).

Kahendotsingu ajalise keerukuse hindamiseks võime selle sõnastada rekursiivselt: igal sammul jätame konstantse ajaga operatsiooni (ühe võrdlustehte) põhjal vaatluse alt välja pooled andmed ja jätkame sama algoritmi abil otsimist ülejäänud poolest. Selline sõnastus annab kahendotsingu ajakulu hinnanguks rekurrentse võrrandi

$$T(n) = \Theta(1) + T(n/2),$$

millest (eelmises peatükis toodud tabeli põhjal) saame

$$T(n) = \Theta(\log n).$$

See tulemus näitab küll, et kahendotsing on lineaarsest oluliselt efektiivsem, kuid ei selgita, miks on kasulik just keskmine element valida.

Asi on selles, et mõne muu elemendi valimisel parandame ajahinnangut parima võimaliku juhu jaoks (näiteks, valides massiivi alguse poole jääva elemendi, saame $a_k > x$ korral kõrvale heita rohkem kui pooled elemendid), aga samal ajal halvendame seda halvima võimaliku juhu jaoks (tulemuse $a_k < x$ korral saame kõrvale heita vähem elemente). Samas, jagades massiivi ebavõrdse pikkusega osadeks, on suurem tõenäosus, et otsitav element jääb suuremasse poolde — mis aga on meie algoritmi seisukohalt halvem võimalus.

Massiivi jagamisel võrdseteks osadeks on seega kaks kasulikku omadust: see vähendab esiteks halvema võimaluse tõenäosust ja teiseks ka halvema võimaluse halbust (tõsi küll, seda viimast parema võimaluse headuse arvelt).

Muide, massiivi ebavõrdse jagamise äärmuslikul juhul — kui valime alati kõige vasakpoolsema elemendi — jõuame tagasi lineaarse otsingu juurde.

Enne algoritmi detailset koostamist on kasulik mõelda ka, mida peaks tegema, kui otsitav väärtus esineb massiivis korduvalt, ja mida siis, kui seda massiivis üldse pole. Järgnev algoritm lahendab need küsimused nii: ta leiab otsitava väärtuse vasakpoolseima võimaliku **pistekoha** (ingl *insertion point*) — see tähendab vasakpoolseima koha, kuhu me võiks selle väärtuse massiivi olemasolevate elementide vahele pista nii, et tulemuseks oleks ikkagi järjestatud massiiv.

Algoritm 7.2 KahOtsV: Kahendotsing massiivis
Sisend: $a_{1\ldots n}$ — antud massiiv, $\forall i < n : a_i \leq a_{i+1}$; x — otsitav väärtus
Väljund: tagastab x vasakpoolseima pistekoha
Abimuutujad: $v, p \in \mathbb{N}$ — uuritava lõigu otspunktid;
$k \in \mathbb{N}$ — lõigu keskpunkt

1. $v \leftarrow 1$; $p \leftarrow v + n$
2. `senikui` $v < p$
   — invariant: otsitav pistekoht on $v \ldots p$
3. $k \leftarrow \lfloor (v + p)/2 \rfloor$
4. `kui` $a_k < x$
5. $v \leftarrow k + 1$
6. `muidu`
7. $p \leftarrow k$
8. `lõppkui`
9. `lõppsenikui`
   — vahetingimus: otsitav pistekoht on $v \ldots p$ ja $v = p$
10. `tagasta` v

Väärib märkimist, et algoritmi real 3 on oluline ümardada allapoole. (Miks? Mõelge, mis juhtuks ülespoole ümardades, kui $x > a_n$.)

7.2.2 Mäluhaldus

Enne järgmise struktuuri — lihtahela — juurde asumist peame korraks pöörduma tagasi juba käsitletud teema juurde ja uurima dünaamiliste muutujate loomise ja kasutamise tehnikaid.

Nagu juba varem öeldud, luuakse dünaamiline muutuja programmi töö ajal vastava käsu peale. Kuna sellist muutujat ei deklareerita programmi kirjutamise ajal, pole sellel ka nime, vaid ainult aadress. Selleks, et neid muutujaid oleks siiski võimalik kasutada, on olemas spetsiaalsed andmetüübid aadresside hoidmiseks, mida nimetatakse **viidatüübiks** (ingl *pointer*) või **viitetüübiks** (ingl *reference*).

Viida- või viitetüüpi muutuja on üldiselt muutuja nagu iga teinegi, ainult tema väärtus on mäluaadress ja põhiline operatsioon selle aadressi järgi vastava mälupesa poole pöördumine.

Oletame, et meil on programmis täisarvmuutuja a ja viitmuutuja v. Programmi käivitamisel eraldatakse kummalegi neist üks mälupesa, mille sisuks on üldiselt mingi juhuslik väärtus (joonis 7.3 (a)). "Tavalisele" muutujale väärtuse omistamisega oleme me juba tuttavad, näiteks

$$a \leftarrow 3$$

salvestatab muutuja a mälupesa väärtuse 3 (joonis 7.3 (b), üleval).

Kuna viitmuutuja väärtus on aadress, siis on viitmuutujaga seotud kaks võimalikku vasakväärtust: esiteks viitmuutuja enda aadress (tema väärtuse hoidmiseks kasutatava mälupesa number) ja teiseks tema väärtus (aadress, mille hoidmiseks seda muutujat kasutatakse). Seega on viitmuutujale omistamisel vaja eristada, kumba kahest võimalikust vasakväärtusest me kasutame.

Enamasti tähendab tavaline omistamine viitmuutuja väärtuseks uue aadressi määramist, näiteks

$$v \leftarrow \texttt{tee-uus}\ \mathbb{Z}$$

eraldab täisarvu hoidmiseks sobiva mälupesa (`tee-uus` $\mathbb{Z}$) ja salvestatab selle mälupesa aadressi muutujasse v (joonis 7.3 (b), all). Kuna uuel muutujal ei ole nime, on seda võimalik kasutada ainult tema aadressi kaudu. Selleks on viitmuutujale omistamisel tavaliselt olemas eriline süntaks, mis näitab, et me soovime salvestada uut väärtust mitte muutuja enda mälupessa (joonisel tähistatud "v:"), vaid tema poolt viidatavasse mälupessa (joonisel tähistatud "?:"). Näiteks

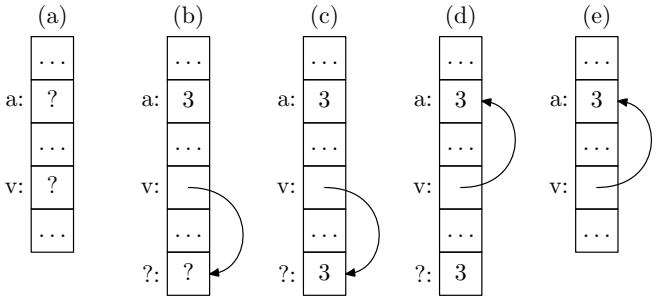
$$[\![v]\!] \leftarrow 3$$

salvestatab v poolt viidatavasse mälupessa väärtuse 3 (joonis 7.3 (c)).

Vahel on kasulik viitmuutujasse salvestada ka mõne "tavalise" muutuja aadress. Selle võimaldamiseks on paljudes keeltes olemas spetsiaalne operatsioon mistahes muutuja mäluaadressi leidmiseks. Näiteks

$$v \leftarrow]\!]a[\!]$$

salvestatab muutuja a aadressi muutuja v mälupessa (joonis 7.3 (d)).



Joonis 7.3: Arv- ja viitmuutuja

Viitmuutujate kasutamisel peab silmas pidama, et me ei kaotaks ära ainukest võimalust oma dünaamilise muutuja poole pöördumiseks. Näiteks eelpool vaadeldud omistamistest koosneva programmilõigu

— vahetingimus: joonis 7.3 (a)
$a \leftarrow 3$
$v \leftarrow$ `tee-uus` $\mathbb{Z}$
— vahetingimus: joonis 7.3 (b)
$]\!]v[\!] \leftarrow 3$
— vahetingimus: joonis 7.3 (c)
$v \leftarrow]\!]a[\!]$
— vahetingimus: joonis 7.3 (d)

täitmise järel on meil mälus dünaamiliselt loodud täisarvmuutuja, mille poole me ei saa pöörduda, sest meil pole kuskil alles selle muutuja aadressi. Samas on see mälu hõivatud ja süsteem ei saa seda kasutada ka muude andmete hoidmiseks. Tegemist on sageli esineva programmiveaga, mida nimetatakse **mälulekkeks** (ingl *memory leak*).

Mälulekete vältimiseks kasutatakse paljudes kõrgkeeltes **prahikoristust** (ingl *garbage collection*) — süsteem kustutab automaatselt kõik dünaamilised muutujad, millele ei osuta enam ükski viit.

Keeltes, kus prahikoristust ei ole, tuleb mittevajalikud dünaamilised muutujad kustutada spetsiaalse käsu abil. Näiteks programmilõik

— vahetingimus: joonis 7.3 (a)
$a \leftarrow 3$
$v \leftarrow$ `tee-uus` $\mathbb{Z}$
— vahetingimus: joonis 7.3 (b)
$[\![v]\!] \leftarrow 3$
— vahetingimus: joonis 7.3 (c)
`kustuta` v
$v \leftarrow]\!]a[\![$
— vahetingimus: joonis 7.3 (e)

mälu ei lekita, kuna dünaamiline muutuja kustutatakse mälust enne selle aadressi "unustamist".

Muutujate mälust kustutamisel kehtivad järgmised reeglid:

- Kustutada tohib ainult dünaamilisi muutujaid. Selle reegli vastu on kerge eksida, kui programmis kasutatakse viitu, mis osutavad staatilistele või automaatsetele muutujatele.
- Iga muutuja tuleb kustutada täpselt üks kord. Eelpool juba oli näide mälulekke kohta, mis tekib, kui dünaamilist muutujat üldse ei kustutata. Muutuja korduv kustutamine on isegi raskem viga, sest võib ajada segadusse süsteemi arvepidamise vaba ja kasutatud mälu üle. Selle reegli vastu on kerge eksida, kui ühele muutujale osutab mitu viita.
- Kustutatud muutujat enam kasutada ei tohi. Ka selle reegli vastu on eriti kerge eksida, kui ühele muutujale osutab mitu viita.

Mitut viita, mis osutavad samale muutujale, nimetatakse **teisikuteks** (ingl *alias*). Teisikviitade ja nendega seotud probleemide illustratsiooniks vaatleme joonist 7.4 ja paneme programmilõigu

— vahetingimus: joonis 7.4 (a)
$v \leftarrow$ `tee-uus` $\mathbb{Z}$
$[\![v]\!] \leftarrow 3$
— vahetingimus: joonis 7.4 (b)
$w \leftarrow v$
$[\![w]\!] \leftarrow 5$
— vahetingimus: joonis 7.4 (c)
`kustuta` v
— vahetingimus: joonis 7.4 (d)

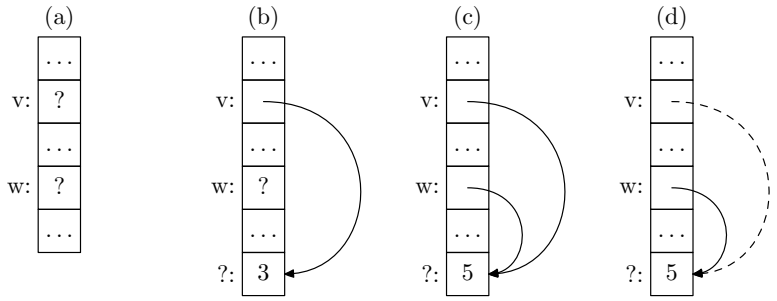
kohta tähele järgmist:

- Omistamine $[\![w]\!] \leftarrow \ldots$ muudab lisaks $[\![w]\!]$ väärtusele ka $[\![v]\!]$ väärtust. See on muidugi ootuspärane, arvestades, et $[\![w]\!]$ ja $[\![v]\!]$ on tegelikult üks muutuja. Küll aga tuleb see üllatusena, kui programmeerija ei märka, et v ja w on teisikud.

- Samal põhjusel ei tarvitse pärast dünaamilise muutuja kustutamist v kaudu olla ilmne, et kadunud on ka w poolt viidatav muutuja. Kui süsteem pole vabanenud mäluosa millegi muu jaoks kasutusele võtnud, võib $[\![w]\!]$ kasutamine pärast muutuja kustutamist isegi õnneks minna. See on eriti ohtlik, sest sellisel juhul võib viga programmi testimisel märkamata jääda ja avalduda alles selle reaalsel kasutamisel.

- Paljudes keeltes pole eelmise vea tegemiseks teisikuid vajagi, sest ka v säilitab oma väärtuse ja ka selle kaudu on võimalik püüda kustutatud muutuja poole pöörduda. Tagajärjed on muidugi samad, ainus erinevus on selles, et niisugust viga on kergem märgata (ja seega tegemata jätta). Ka aitab vea seda juhtu ära hoida muutujale v tühiviida omistamine kohe pärast käsku `kustuta` v.

  Tühiviit on spetsiaalne viidatüüpi väärtus, millele ei vasta reaalset mäluaadressi. Enamikus süsteemides toob iga katse tühiviida poolt osutatava mälupesa poole pöörduda kaasa veaolukorra, mistõttu see ei jää programmi testimisel märkamata. Tühiviita tähistatakse erinevates keeltes erinevalt[1], meie kasutame edaspidi sümbolit $\perp$.



Joonis 7.4: Teisikviidad

Muidugi oleks kõiki neid vigu võimalik vältida, kui süsteem kontrolliks iga kustutamiskäsu juures vabastatava aadressi olekut ja nulliks kõik sellele muutujale osutavad viidad. Paraku oleks see väga ressursikulukas, mistõttu seda ei tehta. Seega peab programmeerija ise hoolikas olema.

[1] Näiteks Pascalis `nil`, Visual Basicus `Nothing`, C's `NULL`, C++'s `0`, Javas `null`.

7.2.3 Ahel

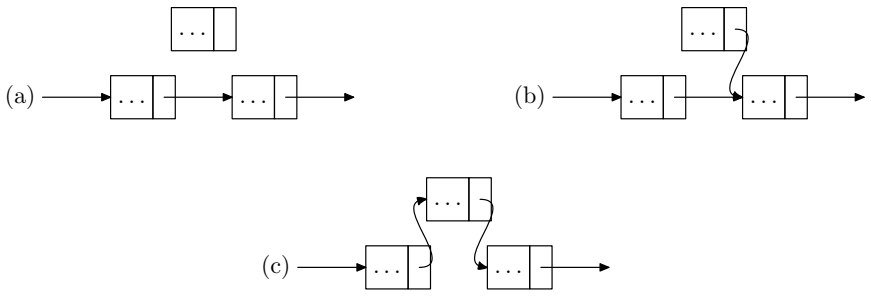
Nagu lubatud, vaatleme järgmise andmestruktuurina lihtahelat.

Lihtahel (ingl *singly linked list*) koosneb elementidest, millest igaühes on lisaks rakenduse seisukohalt vajalikele andmetele ka järgmise elemendi aadress. Nii on ahela töötlemisel võimalik liikuda esimeselt elemendilt teisele, teiselt kolmandale ja nii edasi kuni ahela lõpuni. Ahela viimases elemendis on tühiviit, mis annabki märku ahela lõpust.

Tavaliselt kasutatakse ahela hoidmiseks viita selle esimesele elemendile. Rohkem pole vaja, sest esimeselt elemendilt saame liikuda kõigi teisteni.

Ahelat kui andmestruktuuri iseloomustavad järgmised omadused:

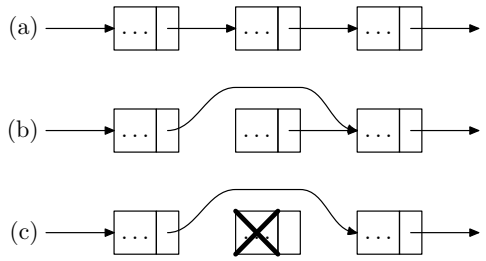
- Ahela pikkus ei ole fikseeritud. See võimaldab ahela abil realiseerida muutuva suurusega andmestruktuure ka keeltes, kus massiivi pikkus tuleb määrata juba programmi kirjutamise ajal.
- Ahela elemendi leidmine tema indeksi järgi on üsna kulukas. Üldjuhul tuleb selleks alustada ahela algusest ja liikuda vastav arv kordi ühelt elemendilt teisele. Seega on i. elemendi leidmise ajakulu $\Theta(i)$.
- Ahela elemendi väärtuse väljavahetamine (ahela elemendi andmeväljale uue väärtuse omistamine) on tavaline omistamistehe ja selleks tehtava töö maht (kui element on juba leitud) ei sõltu ei ahela pikkusest ega elemendi positsioonist ahelas.
- Uue elemendi lisamine ahelasse on üldiselt efektiivne operatsioon. Kui meil on olemas lisamispositsioonile eelneva elemendi aadress, kulub lisamiseks ainult kaks viitmuutuja omistamist (joonis 7.5).



Joonis 7.5: Elemendi lisamine ahelasse

Täpselt sama efektiivne on ühe elemendi asemel terve teise ahela lisamine ahelasse, kui meil on (lisaks lisamispositsioonile eelneva elemendi aadressile) olemas lisatava ahela esimese ja viimase elemendi aadressid.

- Ahelast elemendi eemaldamine on üldiselt efektiivne operatsioon. Kui meil on olemas eemaldatavale elemendile eelneva elemendi aadress, kulub eemaldamiseks ainult üks viitmuutuja omistamine (joonis 7.6).



Joonis 7.6: Elemendi eemaldamine ahelast

  Sama efektiivne on ühe elemendi asemel terve lõigu eemaldamine ahelast, kui meil on (lisaks eemaldamispositsioonile eelneva elemendi aadressile) olemas eemaldatava lõigu viimase elemendi aadress.

  Muidugi tuleb mõlemal juhul arvestada lisaks ka eemaldatud elementide mälust kustutamisega.

Nagu eelnevast näha, on ahela kasutamisel sageli vaja jooksvale elemendile eelneva elemendi aadressi. See tekitab raskusi ahela esimese elemendi töötlemisel, sest sellele eelnev element puudub.

Tihti on mugavaim lahendus lisada ahela algusse fiktiivne element, mille andmeväljad on tühjad ja mille ainus ülesanne on olla esimesele tegelikule elemendile eelnev element. Sellist ahelat nimetatakse **päisega lihtahelaks** ja lisatud elementi **päiseelemendiks** ehk **päiseks** (ingl *header*).

Teine komplikatsioon lihtahela kasutamisel on, et antud elemendile eelneva elemendi leidmine on üsna kulukas operatsioon. Üldjuhul tuleb selleks alustada ahela algusest ja liikuda edasi, kuni jõuame otsitava elemendini.

Paljudes algoritmides on võimalik ahela läbimiseks kasutada kaht muutujat, millest üks osutab eelmisele ja teine jooksvale elemendile. Sellist paari nimetatakse **tandemiks**.

Kui algoritmi pole siiski võimalik (või efektiivne) üles ehitada tandemile, võib lihtahela asemel kasutada **topeltahelat** (ingl *doubly linked list*), mille igas elemendis on lisaks andmeväljadele kaks viita: üks eelmisele, teine järgmisele elemendile.

Topeltahela peamine halvemus lihtahelaga võrreldes on keerukam viitade süsteem ja sellest tulenevalt ka keerukamad lisamis- ja eemaldamisoperatsioonid. Muidugi võib ka topeltahelast teha päisega variandi.

Kuna kõigis neis ahelates on indeksi järgi elemendi leidmine ikkagi kulukas, on väärtuse järgi elemendi leidmiseks mõeldav ainult lineaarne otsimine. Ahelas, erinevalt massiivist, oleks kahendotsimine lineaarsest aeglasem.

7.3 Järjestamine

Massiiv ja ahel on esitatavad elementide loendina ja sellest tekib tihti vajadus loetleda nende elemente nende väärtuste järjekorras. Andmeid võib **järjestada** (ingl *order*) ehk **sortida** (ingl *sort*) paljude erinevate algoritmide abil.

Korduslausete peatükis vaatlesime ühe näitealgoritmina **pistemeetodit** (ingl *insertion sort*):

Algoritm 7.3 PisteSort
Sisend: $a_{1\ldots n}$ — töödeldav massiiv
Väljund: $a_{1\ldots n}$ — väärtused vahetatud nii, et $\forall i < n : a_i \leq a_{i+1}$
Abimuutujad: i, j

1. `korda` $i \leftarrow 1 \ldots n$
   — invariant: $a_1 \leq a_2 \leq \ldots \leq a_{i-1}$
2. $j \leftarrow i$
3. `senikui` $j > 1$
   — invariant: $a_j \leq a_{j+1} \leq \ldots \leq a_i$
4. `kui` $a_{j-1} > a_j$
5. $a_{j-1} \leftrightarrow a_j$
6. $j \leftarrow j - 1$
7. `muidu`
8. $j \leftarrow 1$
9. `lõppkui`
10. `lõppsenikui`
11. `lõppkorda`

Olles vahepeal läbinud keerukuse peatüki võime nüüd hinnata ka selle algoritmi efektiivsust. Selleks paneme tähele, et sisemise korduslause kehaks on meil tingimuslause, milles on üks võrdlemine ja (sõltuvalt võrdluse tulemusest) üks või neli omistamist. Igal juhul on selle korduse keha täitmiseks kuluv aega tõkestatud mingi massiivi suurusest sõltumatu konstandiga. Seega on sisemise korduse keha keerukus $\Theta(1)$.

Seda kordust võidakse läbida minimaalselt 1 kord (kui a_i on $a_{1..i}$ hulgas maksimaalne) ja maksimaalselt i korda (kui a_i on $a_{1..i}$ hulgas vähim). Kui massiiv on alguses järjestatud juhuslikult, võiks eeldada, et elemendi a_i

lõpliku asukohana on võrdtõenäolised kõik i erinevat positsiooni, mis annab sisemise korduse täitmiste keskmiseks arvuks $i/2$.

Välimist kordust läbitakse täpselt n korda, mis annab kogu algoritmi keerukuseks parimal, halvimal ja keskmisel juhul vastvalt

$$\begin{aligned} T_{min}(n) &= \sum_{i=1}^{n} \Theta(1) = n\Theta(1) = \Theta(n), \\ T_{max}(n) &= \sum_{i=1}^{n} i\Theta(1) = \frac{n(n+1)}{2}\Theta(1) = \Theta(n^2), \\ T(n) &= \sum_{i=1}^{n} \frac{i}{2}\Theta(1) = \frac{n(n+1)}{4}\Theta(1) = \Theta(n^2). \end{aligned}$$

Pistemeetodit on võimalik kohandada lihtahela sortimiseks. Kuna lihtahelas on võimalik liikuda ainult ühes suunas, tuleb sel juhul ka sisemises korduses liikuda ahela alguse poolt lõpu poole. See muudab küll algoritmi vahetingimusi, kuid mitte selle efektiivsust.

Korduslausete peatükist on tuttav ka teine suhteliselt lihtne sortimisalgoritm, **valikumeetod** (ingl *selection sort*):

Algoritm 7.4 ValikuSort
Sisend: $a_{1\ldots n}$ — töödeldav massiiv
Väljund: $a_{1\ldots n}$ — väärtused vahetatud nii, et $\forall i < n : a_i \leq a_{i+1}$
Abimuutujad: i, j, k

1. `korda` $i \leftarrow 1 \ldots n$
   — invariant: $a_1 \leq a_2 \leq \ldots \leq a_{i-1}$
   — invariant: $\forall j \geq i : a_j \geq a_{i-1}$
2. $k \leftarrow i$
3. `korda` $j \leftarrow i+1 \ldots n$
   — invariant: $a_k = \min(a_{i\ldots j-1})$
4. `kui` $a_j < a_k$
5. $k \leftarrow j$
6. `lõppkui`
   — vahetingimus: $a_k = \min(a_{i\ldots j})$
7. `lõppkorda`
   — vahetingimus: $a_k = \min(a_{i\ldots n})$
8. $a_i \leftrightarrow a_k$
   — vahetingimus: $\forall j \geq i : a_j \geq a_i$
   — vahetingimus: $a_1 \leq a_2 \leq \ldots \leq a_i$
9. `lõppkorda`

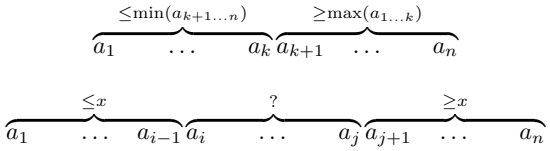
Ajalise keerukuse hindamine on seekord isegi lihtsam kui pistemeetodi puhul: sisemise korduse keha on $\Theta(1)$; sisemist kordust läbitakse $n-i$ korda ja välimist n korda, mis kokku annab selle meetodi keerukuseks igal juhul

$$\begin{aligned} T(n) &= \sum_{i=1}^{n}(n-i)\Theta(1) \\ &= (n^2 - \frac{n(n-1)}{2})\Theta(1) = \frac{n(n+1)}{2}\Theta(1) = \Theta(n^2). \end{aligned}$$

Valikumeetodi puhul on $\Theta(n^2)$ ainult elementide võrdlemiste arv, järjestatava massiivi elementide omistamiste arv on $\Theta(n)$, mis võib olla oluline, kui elemendid on suured kirjed või kui nende liigutamisega kaasneb mingeid muid kulutusi (näiteks võivad mingid teised andmestruktuurid hoida elementide indekseid, mida tuleks elemendi liigutamisel uuendada).

Valikumeetod vajab töödeldavate andmete hulgas ainult ühesuunalist liikumist (mõlemad kordused liiguvad andmete alguse poolt lõpu poole), seega on see peaaegu muutusteta kasutatav ka lihtahela sortimiseks.

Praktikas on massiivide sortimisel üldiselt efektiivseim **kiirmeetod** (ingl *quicksort*). Selle meetodi põhiidee jagada massiiv $a_{1\ldots n}$ kaheks osaks $a_{1\ldots k}$ ja $a_{k+1\ldots n}$ nii, et ükski esimese osa element pole suurem ühestki teise osa elemendist (joonis 7.7, üleval). Pärast seda võime kummagi osa eraldi sortida ja saamegi tulemuseks korrektselt järjestatud massiivi.



Joonis 7.7: Kiirmeetodi põhiidee

Jaotamiseks valime mingi massiivis esineva väärtuse x ning hakkame sellest väiksemaid elemente koguma massiivi esimesse ja suuremaid teise poolde (joonis 7.7, all). Alustame seisust $i=1$, $j=n$ ning liigume edasi $i \leftarrow i+1$, kui $a_i < x$ (see tähendab, kui a_i kuulub esimesse poolde) ja $j \leftarrow j-1$, kui $a_j > x$ (kui a_j kuulub teise poolde). Kui oleme mõlemas pooles avastanud elemendi, mis võiks kuuluda vastaspoolde, vahetame need elemendid omavahel ja jätkame võrdlemist kuni vaatlemata osa $a_{i\ldots j}$ on tühi.

Kui kasutada kummagi jaotamisel saadud massiiviosa sortimiseks sama meetodit, on tulemuseks järgmine rekursiivne algoritm:

Algoritm 7.5 KIIRSORT
Sisend: $a_{v\ldots p}$ — töödeldav massiiviosa
Väljund: $a_{v\ldots p}$ — väärtused vahetatud nii, et $a_v \leq \ldots \leq a_p$
Abimuutujad: i, j, x

1. `kui` $v < p$
2. $\quad i \leftarrow v$; $j \leftarrow p$
3. $\quad x \leftarrow a_{\lfloor (i+j)/2 \rfloor}$
4. $\quad$ `senikui` $i < j$
   — invariant: $\forall i' < i : a_{i'} \leq x$ ja $\forall j' > j : a_{j'} \geq x$
5. $\quad\quad$ `senikui` $a_i < x$
   — invariant: $\forall i' < i : a_{i'} \leq x$
6. $\quad\quad\quad i \leftarrow i + 1$
7. $\quad\quad$ `lõppsenikui`
8. $\quad\quad$ `senikui` $a_j > x$
   — invariant: $\forall j' > j : a_{j'} \geq x$
9. $\quad\quad\quad j \leftarrow j - 1$
10. $\quad\quad$ `lõppsenikui`
11. $\quad\quad$ `kui` $i < j$
12. $\quad\quad\quad a_i \leftrightarrow a_j$
13. $\quad\quad\quad i \leftarrow i + 1$; $j \leftarrow j - 1$
14. $\quad\quad$ `lõppkui`
15. $\quad$ `lõppsenikui`
16. $\quad$ KIIRSORT$(a_{v\ldots j})$
17. $\quad$ KIIRSORT$(a_{j+1\ldots p})$
18. `muidu`
    — ühest elemendist koosnev osa on alati sorditud
19. `lõppkui`

Kiirmeetodi ajalise keerukuse hindamiseks paneme tähele, et n-elemendilise massiivi jagamiseks kulub igal juhul $\Theta(n)$ sammu, millele järgneb kaks rekursiivset väljakutset.

Parimal juhul õnnestub massiiv igal sammul jagada kaheks võrdseks osaks. Siis on kiirmeetodi keerukus

$$T_{min}(n) = \Theta(n) + 2T_{min}(n/2) = \Theta(n \log n).$$

Halvimal juhul valime "veelahkmeks" alati kas maksimaalse või minimaalse väärtusega elemendi ja saame tulemuseks jaotuse 1- ja (n-1)-elemendiliseks osaks. Siis on kiirmeetodi keerukus

$$T_{max}(n) = \Theta(n) + T_{max}(n-1) = \Theta(n^2).$$

Õnneks realiseerub halvim stsenaarium üliharva ja keskmiselt on kiirmeetodi ajaline keerukus optimistlik $T(n) = \Theta(n \log n)$. Selle tulemuse tõestamine pole aga sugugi triviaalne ja jääb siinkohal tegemata.

Kuna kiirmeetod vajab kindlasti kahesuunalist liikumist andmete hulgas, ei sobi ta lihtahela töötlemiseks. Küll aga on võimalik seda algoritmi kohandada kasutamiseks topeltahelal.

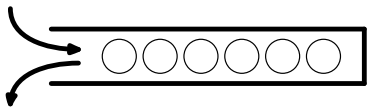
Lisaks senivaadelduile on olemas veel palju huvitavaid sortimisalgoritme, millega tasuks kindlasti tutvuda peatüki lõpus soovitatud raamatute abil.

7.4 Abstraktsemad struktuurid

7.4.1 Magasin

Magasin (ingl *stack*) ehk **pinu** on lineaarne andmestruktuur, mis võimaldab juurdepääsu oma elementidele kindlas järjekorras — magasinist elemendi väljavõtmisel saame esimesena selle elemendi, mille me sinna viimasena panime, järgmisena selle, mille panime eelviimasena jne.

Sellise juurdepääsusüsteemi kohta kasutatakse sageli lühendit **LIFO** (ingl *last-in-first-out*) ja seda võib kujutleda ühest otsast kinnise toruna (joonis 7.8) — mistahes elemendi kättesaamiseks peame kõigepealt eest ära võtma need, mis on tema ja toru lahtise otsa vahel.



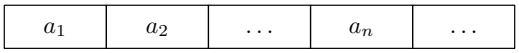
Joonis 7.8: Magasini tööpõhimõte

Põgusalt puutusime magasini mõistega kokku juba alamprogramme käsitledes. Seda meenutades on selge, et alamprogrammide aktiveerimiskirjete hoidmiseks on kõige õigem just magasin — kui üks alamprogramm kutsub välja teise ja teine omakorda kolmanda, tahame loomulikult, et kolmanda alamprogrammi töö lõppedes jätkatakse teise alamprogrammi täitmist ning esimese juurde pöördutakse tagasi alles pärast teise lõpetamist.

Kuna LIFO-tüüpi andmehoidla on kasulik ka mitmetel muudel juhtudel, tuleb keeltes, kus magasinitüüpi standardselt olemas ei ole, see realiseerida mõne teise andmestruktuuri abil.

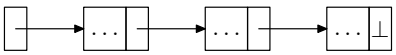
Üks võimalus on kasutada magasini baasina massiivi. See on üsna lihtne: hoiame magasini elementide arvu muutujas n ja magasini sisu massiivi a elementides a_1 kuni a_n nii, et a_n on kõige uuem element (joonis 7.9). Andmete

magasini panemisel suurendame n väärtust ja salvestame uued andmed massiivi elementi a_n. Andmete magasinist võtmisel tagastame a_n sisu ja seejärel vähendame n väärtust.



Joonis 7.9: Magasin massiivi baasil

Alternatiiv on kasutada lihtahelat (joonis 7.10). Ka see pole palju keerulisem, kui hoiame magasini sisu ahelas nii, et uuemad elemendid on ahelas vanematest eespool. Andmete magasini panemisel lisame uusi kirjeid ahela algusse, andmete võtmisel tagastame ahela esimese elemendi sisu. Kõik lisamise ja eemaldamise operatsioonid toimuvad alati ahela alguses, mis on juurdepääsu efektiivsuse seisukohalt parim variant.

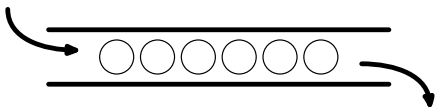


Joonis 7.10: Magasin lihtahela baasil

7.4.2 Järjekord

Teine lihtne, aga kasulik andmestruktuur on **järjekord** (ingl *queue*). Ka järjekord võimaldab juurdepääsu oma elementidele kindlas järjekorras, kuid erinevalt magasinist tagastab järjekord esimesena oma vanima elemendi.

Sellise juurdepääsusüsteemi kohta öeldakse **FIFO** (ingl *first-in-first-out*) ja seda võib kujutleda toruna, mille sisu liigub alati ühes suunas (joonisel 7.11 vasakult paremale).



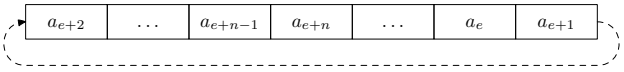
Joonis 7.11: Järjekorra tööpõhimõte

Järjekord on kasulik sellistes algoritmides, kus mingi protsess toodab andmeid ja mingi teine protsess tarbib neid, aga pole garanteeritud, et tarbija

kõige toodetuga piisavalt kiiresti hakkama saab. Väga levinud on sellised "torud" multitegumsüsteemides, aga peagi näeme, et neist võib olla kasu isegi siis, kui andmete tootja on ise ka nende tarbija.

Ka järjekorra võib realiseerida massiivi baasil, kuid efektiivse tulemuse saamine on sel juhul natuke keerulisem. Magasinis toimus andmete lisamine ja eemaldamine elementide jada samas otsas, jada teine ots "püsis paigal". Järjekorra puhul see aga nii ei ole. Püüd hoida järjekorra elemente alati massiivi ühes otsas tähendaks seda, et üks kahest operatsioonist (lisamine või eemaldamine) nõuaks kõigi elementide nihutamist.

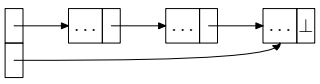
Selle vältmiseks on kasulikum hoida eraldi järjekorras olevate elementide arvu n ja esimese elemendi indeksit e. Järjekorra elemendid on sellisel juhul massiivi a elementides a_e, a_{e+1}, ..., a_{e+n-1}. Esimene vaba element on a_{e+n}. Selleks, et indeksid järjekorra kasutamisel tõkestamatult kasvama ei hakkaks, kasutame massiivi ringpuhvrina, kus massiivi viimasele elemendile järgneb esimene (joonis 7.12). Selline lahendus võimaldab realiseerida nii lisamise kui eemaldamise massiivi suurusest või järjekorras olevate elementide arvust sõltumatu väikese ajakuluga.



Joonis 7.12: Järjekord massiivi baasil

Vajadus pääseda kiiresti ligi järjekorra mõlemale otsale komplitseerib ka selle realiseerimist lihtahela baasil. Siin on lahenduseks hoida eraldi viita järjekorda kujutava ahela viimasele elemendile (joonis 7.13). Erinevalt igapäevase elu elavast järjekorrast, kus inimene jätab meelde tema ees seisja, on arvutis kasulikum hoida igas elemendis talle järgneva aadressi. Seega on lihtahela alguses järjekorra vanim ja lõpus uusim element.

Järjekorra viimase elemendi eemaldamisel tuleb panna tähele, et meil on tegemist teisikviitadega: kuna ainus element on ahelas üheaegselt nii esimene kui ka viimane, osutavad mõlemad viidad talle ja elemendi eemaldamisel tuleks ka mõlemad nullida.



Joonis 7.13: Järjekord lihtahela baasil

Ülesanded

Ülesanne 7.1 *Kui otsitav väärtus esineb massiivi elementide hulgas korduvalt, leiab* LinOtsV *selle vasakpoolseima (vähima indeksiga) esinemiskoha. Kirjutada* LinOtsP*, mis leiab otsitava väärtuse parempoolseima (suurima indeksiga) esinemiskoha. Tuua välja selle korduse invariant.*

Ülesanne 7.2 KahOtsV *leiab antud väärtuse jaoks vasakpoolseima võimaliku pistekoha. Kirjutada* KahOtsP*, mis leiab parempoolseima võimaliku pistekoha. Tuua välja selle korduse invariant ja põhjendada selle säilimist korduse keha täitmisel.*

Ülesanne 7.3 *Realiseerida lihtahela jaoks lineaarse otsimise mõlemad variandid (nii* LinOtsV *kui ka* LinOtsP *analoog).*

Ülesanne 7.4 *Koostada algoritmid päisega topeltahela elementide lisamiseks ja eemaldamiseks.*

Lisamisoperatsiooni parameetrid on lisamispositsioonile eelneva elemendi aadress ja lisatava elemendi aadress. Eemaldamisoperatsiooni ainus parameeter on eemaldatava elemendi aadress.

Kontrollida, et mõlemad operatsioonid töötavad õigesti ka ahela esimese ja viimase elemendi lisamisel ja eemaldamisel.

Ülesanne 7.5 *Realiseerida algoritm* PisteSort *lihtahela jaoks.*

Kirjandus

Rein Jürgenson. *Andmestruktuuride lühikursus.* Tallinna Tehnikaülikool, 2000.
Põhilisi andmestruktuure ja nende realiseerimist käsitlev õpik algajatele. 96 lk.

Jüri Kiho. *Algoritmid ja andmestruktuurid.* Tartu Ülikool, 2003.
Andmestruktuuride ja põhialgoritmide õpik. 148 lk.

Robert Sedgewick. *Algorithms.* Addison-Wesley, 1988.
Klassikaline andmestruktuuride ja algoritmide õpik. 650 lk. Osutatud väljaandes on programminäited toodud Pascalis, hiljem on ilmunud ka *Algorithms in C*, *Algorithms in C++* ja *Algorithms in Java*.

Thomas H. Cormen, Clifford Stein, Charles Leiserson, Ronald Rivest. *Introduction to Algorithms.* MIT, 2001.
Eelmise trükiga klassikaks saanud algoritmide ja andmestruktuuride põhiõpik, *de facto* standard paljudes ülikoolides üle maailma. 1180 lk.

Томас Кормен, Чарльз Лейзерсон, Рональд Ривест. *Алгоритмы: построение и анализ.* МЦНМО, 2001.
Eelmise tõlge. 960 lk.

Sanjoy Dasgupta, Christos H. Papadimitriou, Umesh V. Vazirani. *Algorithms.* McGraw-Hill, 2006.
Berkeley ja UCSD algoritmide ja andmestruktuuride kursuste põhjal koostatud õpik, sobib väga hästi ka iseseisvaks õppimiseks. 336 lk.
Aadressil `http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/` on väljas ka täistekst.

Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1997.
Monumentaalne teos, mille algselt planeeritud 7 köidet pidid koondama kõik olulisemad teadmised programmeerimise ja algoritmide kohta. Väga põhjalik, aga algajale üsna raske lugemine. Seni ilmunud 3 köidet: *Volume 1: Fundamental Algorithms*, 670 lk; *Volume 2: Seminumerical Algorithms*, 776 lk; *Volume 3: Sorting and Searching*, 794 lk.
`http://www-cs-faculty.stanford.edu/~knuth/taocp.html`

Дональд Э. Кнут. *Искусство программирования.* Вильямс, 2000.
Eelmise tõlge. 720+832+844 lk.

Peatükk 8

Mittelineaarsed andmestruktuurid

Sisukord

Käesolevas peatükis jätkame tähtsamate andmestruktuuride ja nendega seotud algoritmide uurimist. Seekord võtame vaatluse alla objektid, mille sisemine struktuur on keerulisem kui lihtsalt nende elementide loend, ja mida seetõttu nimetatakse "mittelineaarseteks".

8.1 Põhilised mittelineaarsed struktuurid

Tähtsaim mittelineaarne andmestruktuur on graaf. Arvutiteaduses on väga vähe ülesandeid, mis poleks ühel või teisel moel esitatavad graafiülesannetena. Graafi tähtsaim erijuht on puu. Nende kahega järgnevalt tutvumegi.

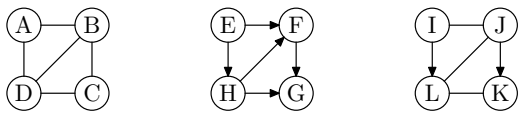
8.1.1 Graaf

Graaf (ingl *graph*) on matemaatiline objekt, mis koosneb mingist hulgast **tippudest** (ingl *vertex*) ja neid tippe paarikaupa ühendavatest **servadest** (ingl *edge*). Asjaolu, et graafi G tippude hulk on V ja servade hulk E, märgitakse tavaliselt $G = (V, E)$.

Kui graafi tippe u ja v ühendaval serval ei ole suunda määratud (seda serva pidi võib liikuda nii tipust u tippu v kui ka vastupidi), siis nimetatakse seda orienteerimata ehk suunamata servaks ehk lühemalt lihtsalt servaks ja tähistatakse $e = \{u, v\}$.

Kui aga serval on suund määratud, nimetatakse seda orienteeritud ehk suunatud servaks ehk **kaareks** (ingl *arc*) ja tähistatakse $e = (u, v)$.

Joonisel kujutatakse graafi tippe tavaliselt punktide, sõõride või kastikestena. Suunamata servi kujutatakse vastavaid tippe ühendavate joontena ja suunatud servi nooltena.



Joonis 8.1: Graafide näiteid

Graafi, mille kõik servad on suunamata, nimetatakse **suunamata graafiks** (ingl *undirected graph*) ehk lühemalt lihtsalt graafiks. Graafi, mille kõik servad on suunatud, nimetatakse **suunatud graafiks** (ingl *directed graph*). Graafi, mille osa servi on suunatud ja osa suunamata, nimetatakse **segagraafiks**. Näiteks joonisel 8.1 toodud graafidest on vasakpoolne suunamata, keskmine suunatud ja parempoolne segagraaf.

Teeks (ingl *path*) graafis $G = (V, E)$ nimetatakse "järjestikuste" servade jada $e_1 = \{v_0, v_1\}$, $e_2 = \{v_1, v_2\}$, ..., $e_n = \{v_{n-1}, v_n\}$, milles iga serva lõpptipp on talle järgneva serva algustipp.

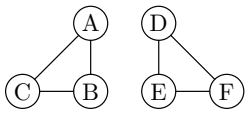
Ahelaks (ingl *chain*) nimetatakse teed, mis ei läbi ühtki serva korduvalt ja **lihtahelaks** (ingl *simple chain*) ahelat, mis ei läbi korduvalt ühtki tippu. **Tsükliks** (ingl *cycle*) nimetatakse ahelat ja **lihttsükliks** (ingl *simple cycle*) lihtahelat, mille algus- ja lõpptipp langevad kokku ($v_0 = v_n$). **Silmuseks** (ingl *loop*) nimetatakse ühest servast koosnevat tsüklit.[1]

Graafi $G = (V, E)$ **alamgraafiks** (ingl *subgraph*) nimetatakse mistahes graafi $G' = (V', E')$, mille korral kehtib $V' \subseteq V$ ja $E' \subseteq E$. See tähendab, et alamgraaf on saadav ülemgraafist mingi hulga tippude ja servade eemaldamise teel. Asjaolu, et G' on G alamgraaf, tähistatakse $G' \subseteq G$.

Graafi G alamgraafi G', mis ei ole võrdne graafi G endaga, nimetatakse tema **pärisalamgraafiks** (ingl *proper subgraph*) ja seda tähistatakse $G' \subset G$.

Tähtsal kohal on graafiteoorias sidususe mõiste. Suunamata graafi nimetatakse **sidusaks** (ingl *connected*), kui selle mistahes tipupaari v_1, v_2 jaoks leidub tee tipust v_1 tippu v_2.

Graafi G alamgraafi G' nimetatakse graafi G **sidususkomponendiks** (ingl *connected component*), kui G' on sidus ja graafis G ei leidu sidusat alamgraafi G'', mis sisaldaks graafi G' pärisalamgraafina. See tähendab, et sidususkomponent on maksimaalne sidus alamgraaf, seda ei saa kasvatada talle uute servade või tippude lisamisega ilma sidusust rikkumata.

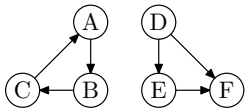


Joonis 8.2: Suunamata graafi sidususkomponendid

Näiteks joonisel 8.2 toodud graafis on kaks sidususkomponenti: tipud A, B, C ja kõik nendega seotud servad ning tipud D, E, F ja kõik nendega seotud servad.

[1] Teede, ahelate ja tsüklitega seonduv terminoloogia ei ole graafiteoorias väga hästi standardiseeritud, seetõttu tuleb enne põhjalikumate järelduste tegemist uurida, kuidas konkreetse raamatu autor need terminid enda jaoks defineerinud on. Näiteks kasutatakse mõnes ingliskeelses raamatus sõna 'path' meie 'ahela' tähenduses ja meie 'tee' jaoks sel juhul otsest vastet polegi. Mingil määral kehtib sama ka graafi enda mõiste kohta. Näiteks nimetatakse paljudes raamatutes graafideks ainult selliseid, milles mistahes kahe tipu vahel võib olla ülimalt üks serv ja silmused pole üldse lubatud. Programmeerimises tavaliselt graafidele selliseid piiranguid ei seata.

Suunatud või segagraafi G nimetatakse sidusaks, kui graafi G kõigi kaarte (v_1, v_2) suunamata servadega $\{v_1, v_2\}$ asendamisel saadud suunamata graaf G' on sidus, ja **tugevalt sidusaks** (ingl *strongly connected*), kui G mistahes tipupaari v_1 ja v_2 jaoks leidub nii tee tipust v_1 tippu v_2 kui ka tee tipust v_2 tippu v_1. Analoogiliselt võime eristada ka sidususkomponente ja **tugevalt sidusaid komponente** (ingl *strongly connected component*).[2]



Joonis 8.3: Suunatud graafi sidususkomponendid

Näiteks joonisel 8.3 toodud suunatud graafis on kaks sidususkomponenti: tipud A, B, C ja kõik nendega seotud kaared ning tipud D, E, F ja kõik nendega seotud kaared. (Kaarte asendamisel servadega saame sama graafi, mis on toodud joonisel 8.2.) Tugevalt sidusaid komponente on selles graafis aga tervelt neli: tipud A, B, C ja kõik nendega seotud kaared ning tipud D, E ja F igaüks eraldi. Tippe D, E ja F ühendavad kaared ei kuulu ühegi tugevalt sidusa komponendi koosseisu.

8.1.2 Puu

Puu (ingl *tree*) on sidus tsükliteta graaf.

Tegemist on mõnes mõttes piirjuhtumiga: puu on ühelt poolt minimaalne sidus graaf (ükskõik millise serva eemaldamine muudab puu mittesidusaks) ja teiselt poolt maksimaalne tsükliteta graaf (ükskõik millise uue serva lisamine tekitab tsükli).

Esimese väite kehtivuses veendumiseks vaatleme puu serva $e = \{u, v\}$. Oletame, et pärast selle serva eemaldamist on graaf endiselt sidus. See aga tähendab, et leidub mingi tee tipust u tippu v, mis ei läbi serva e. Lisades nüüd sellele teele serva e, saame tsükli. See on muidugi vastuolus eeldusega, et esialgne graaf oli tsükliteta, mistõttu peame järeldama, et sellist teed tipust u tippu v ei saa leiduda. Kuna arutelu ei sõltu serva e valikust, tähendab see, et puu muutub mittesidusaks ükskõik millise serva eemaldamisel.

Teise väite põhjendamine on sarnane: vaatleme puu tippe u ja v. Kuna puu on sidus, siis peab leiduma tee tipust u tippu v. See aga tähendab, et

[2]Peaks olema ilmne, et suunamata graafis langevad sidususe ja tugeva sidususe mõisted kokku, mistõttu sel juhul pole tugeva sidususe ja tugevalt sidusa komponendi mõiste järele vajadust.

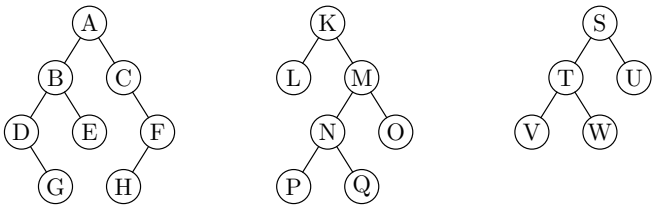
tippude u ja v vahele uue serva lisamine tekitaks tsükli. Kuna arutelu ei sõltu tippude u ja v valikust, tähendab see, et puusse tekib tsükkel mistahes serva lisamisel.

Programmeerimise seisukohalt pakuvad rohkem huvi **juurega puud** (ingl *rooted tree*). Juurega puu erineb tavalisest selle poolest, et üht selle tippu nimetatakse **juureks** (ingl *root*). Joonisel märgitakse juurtipp mõnikord rasvaselt, aga sagedamini paigutatakse tipud nii, et juurtipp on teistest kõrgemal, nagu näiteks tipud A, K ja S joonisel 8.4 toodud puudes.

Juurtipu esiletõstmine muudab puu hierarhiaks. Iga serva otstippudest on üks juurele lähemal kui teine. Juurele lähemat tippu nimetatakse teise **ülemuseks** (ingl *parent*), kaugemat tippu omakorda tema **alluvaks** (ingl *child*). Näiteks joonisel 8.4 vasakul toodud puus on tipp B tipu A alluv ja samal ajal tippude D ja E ülemus.

Puu tippu, millel on mõni alluv, nimetatakse **sõlmeks** (ingl *inner node*), alluvateta tippu aga **leheks** (ingl *leaf*). Näiteks joonisel 8.4 keskel kujutatud puus on K, M ja N sõlmed, aga L, O, P ja Q lehed.

Puu tippu koos tema kõigi (nii vahetute kui ka kaugemate) alluvatega nimetatakse **alampuuks** (ingl *subtree*). Näiteks joonisel 8.4 parempoolses puus moodustavad alampuu tipud T, V ja W. Muidugi on alampuu ka iga leht eraldi ja kogu puu tervikuna.



Joonis 8.4: Kahendpuude näiteid

Puud, mille igal sõlmel on maksimaalselt n alluvat, nimetatakse n-puuks. Näiteks kõik joonisel 8.4 kujutatud puud on **kahendpuud** (ingl *binary tree*). Puud, mille kõigil sõlmedel on sama arv alluvaid, nimetatakse **homogeenseks**. Näiteks joonisel 8.4 vasakul toodud puu ei ole homogeenne (tippudel C, D ja F on igaühel ainult üks alluv), aga keskmine ja parempoolne on.

Homogeenset puud, mille kõik lehed on juurest võrdsel kaugusel, nimetatakse **täielikuks**. Joonisel 8.4 kujutatud puudest pole ükski täielik, kuid parempoolset nimetatakse **kompaktseks** — tal puuduvad täielikkusest vaid mõned lehed alumise tippuderea lõpust.

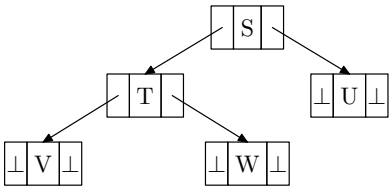
Kahendpuus on sageli kasulik eristatada, kas alluv on oma ülemuse vasak

või parem alluv, isegi juhul kui tegu on ainsa alluvaga. Kahendpuu joonistamisel paigutatakse vasak alluv oma ülemusest madalamale ja vasakule, parem alluv aga vastavalt madalamale ja paremale. Seega on joonisel 8.4 vasakul kujutatud puus tipp F tipu C parem ja tipp H omakorda tipu F vasak alluv.

8.2 Puualgoritmid

8.2.1 Kahendpuu esitamine arvuti mälus

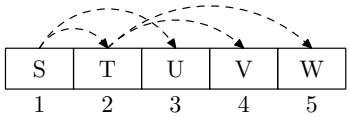
Kahendpuud on loomulik hoida rekursiivse struktuurina, kus iga tippu esitab kirje, milles on väljad selle tipu andmetega ja lisaks viidad selle tipu vasakule ja paremale alluvale. Kui tipul üht või teist (või ka mõlemat) alluvat pole, siis on vastavas väljas tühiviit. Kahendpuu loomuliku esituse näide on toodud joonisel 8.5.



Joonis 8.5: Kahendpuu loomulik esitus

Teine võimalus on hoida kahendpuud massiivis, nagu näidatud joonisel 8.6: puu juur on massiivi elemendis a_1; elemendis a_i hoitava tipu alluvad on elementides a_{2i} ja a_{2i+1} ning tema ülemus elemendis $a_{\lfloor i/2 \rfloor}$.[3]

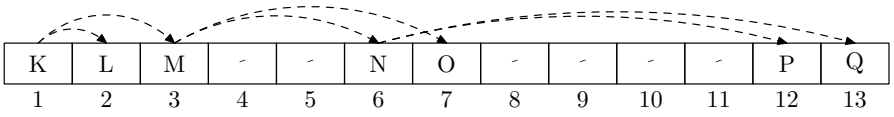
Massiiviesitus sobib hästi täielike ja kompaktsete kahendpuude hoidmiseks, sest siis täidab n-tipuline puu parajasti massiivi elemendid $a_{1\ldots n}$.



Joonis 8.6: Kompaktse kahendpuu massiiviesitus

Nagu näha ka jooniselt 8.7, jääks mittetäieliku puu sellisel esitamisel osa massiivi elemente kasutamata. Esiteks oleks see muidugi asjatu mälukulu.

[3]Kirjutis $\lfloor i/2 \rfloor$ tähendab, et jagamisel tuleb võtta jagatise täisosa.



Joonis 8.7: Mittetäieliku kahendpuu massiiviesitus

Teiseks peaks me siis leidma mingi võimaluse neid tühje elemente tähistada, et me neid hiljem puu töötlemisel kogemata kasutada ei püüaks. Kuna joonisel näidatud nooled on vaid mõttelised seosed, mitte viidad, ei saa me nende puhul tühiviidaga võrdlemist kasutada!

8.2.2 Kahendpuu läbimine

Paljude puuülesannete lahendamisel on vaja puu tippe läbida mingis kindlas järjekorras. Vaatleme näiteks sellist ülesannet: antud puu, mille lehtedes on arvud; vaja on saada igasse sõlmtippu sellest algavas alampuus olevate lehtede arvude summa.

Muidugi saame igas sõlmes vajaliku summa leida tema vahetutes alluvates olevate arvude summana, aga seda ainult eeldusel, et alluvates on vajalikud summad juba välja arvutatud. See tähendab, et nõutud summade efektiivseks arvutamiseks tuleb iga tipu alluvad töödelda enne seda tippu ennast.

Puu niisugust läbimist nimetatakse **lõppjärjestuses läbimiseks** (ingl *post-order traversal*) ja selle üldkuju kahendpuude jaoks kirjeldab algoritm 8.1, mille täiendamine summaülesande lahenduseks oleks muidugi triviaalne.

Algoritm 8.1 LabiPuuLJ: Kahendpuu läbimine lõppjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

1. `kui` $t \neq \perp$ — tühja puu korral pole midagi läbida
2. LabiPuuLJ($[\![t]\!].v$) — töötleme vasaku alampuu
3. LabiPuuLJ($[\![t]\!].p$) — töötleme parema alampuu
4. — siin töötleme tipu $[\![t]\!]$
5. `lõppkui`

Kui summaülesandes liikus informatsioon puu läbimisel "alt üles" ja selle lahendamiseks oli vaja alluvad töödelda enne ülemust, siis paljudes teistes ülesannetes liigub informatsioon vastupidi, "ülalt alla".

Seda tüüpi ülesannete efektiivseks lahendamiseks on sageli mugavaim viis algoritm 8.2, puu läbimine **eesjärjestuses** (ingl *pre-order*).

Algoritm 8.2 LABIPUUEJ: Kahendpuu läbimine eesjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

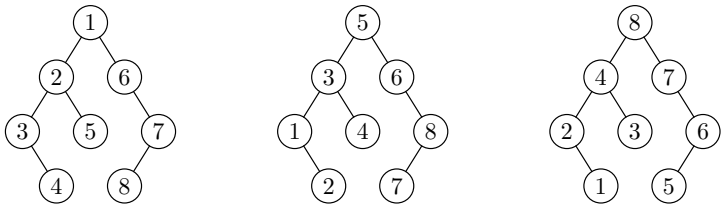
1. `kui` $t \neq \bot$ — tühja puu korral pole midagi läbida
2. — siin töötleme tipu $[\![t]\!]$
3. LABIPUUEJ($[\![t]\!].v$) — töötleme vasaku alampuu
4. LABIPUUEJ($[\![t]\!].p$) — töötleme parema alampuu
5. `lõppkui`

Kolmandat klassikalist tehnikat kujutav algoritm 8.3 läbib kahendpuu tipud **keskjärjestus** (ingl *in-order*): kõigepealt vasaku alampuu, siis juurtipu ja lõpuks parema alampuu. Erinevalt kahest eelmisest on see meetod loomulikul viisil kasutatav ainult kahendpuude korral, sest üldisemal juhul pole selge, millise kahe alluva vahel peaks töötlema alampuu juurtipu.

Algoritm 8.3 LABIPUUKJ: Kahendpuu läbimine keskjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

1. `kui` $t \neq \bot$ — tühja puu korral pole midagi läbida
2. LABIPUUKJ($[\![t]\!].v$) — töötleme vasaku alampuu
3. — siin töötleme tipu $[\![t]\!]$
4. LABIPUUKJ($[\![t]\!].p$) — töötleme parema alampuu
5. `lõppkui`

Joonis 8.8 illustreerib ühe kahendpuu läbimist nende kolme algoritmi poolt: joonisel vasakul, keskel ja paremal on puu tipud nummerdatud vastavalt nende ees-, kesk- ja lõppjärjestuses läbimise järjekorras.



Joonis 8.8: Kahendpuu kolm läbimist

8.2.3 Kahendotsingu puu

Kahendpuud T nimetatakse **kahendotsingu puuks** (ingl *binary search tree*), ehk lühemalt otsingupuuks, kui (a) T on tühi või (b.1) vasaku alampuu kõigi tippude väärtused on juurtipu omast väiksemad ja (b.2) parema alampuu kõigi tippude väärtused juurtipu omast suuremad ja (b.3) kumbki alampuu on ka ise otsingupuu.[4]

Mingi väärtuse otsimine otsingupuus (algoritm 8.4) on sarnane kahendotsinguga järjestatud massiivis: igal sammul võrdleme otsitavat väärtust puu juurtipus olevaga; kui puu tipus olev väärtus on otsitavast väiksem, võime lisaks juurtipule edasise vaatluse alt välja jätta ka kogu vasaku alampuu, see tähendab jätkata otsimist paremas alampuus; kui puu juurtipus olev väärtus on otsitavast suurem, võime otsimist jätkata vasakus alampuus.

Algoritm 8.4 OtsiPuus: Otsimine otsingupuus
Sisend: t — viit otsingupuu juurele; x — otsitav väärtus
Väljund: Tagastab leitud kirje, või $\perp$, kui x puus ei esine

1. `kui` $t = \perp$
2. `  tagasta` $\perp$ — tühjas puus ei leia midagi
3. `muidukui` $x < [\![t]\!].x$
4. `  tagasta` OtsiPuus($[\![t]\!].v, x$)
5. `muidukui` $x > [\![t]\!].x$
6. `  tagasta` OtsiPuus($[\![t]\!].p, x$)
7. `muidu`
8. `  tagasta` t — leidsime otsitava
9. `lõppkui`

Elemendi lisamine otsingupuusse (algoritm 8.4) on otsimisega üsna sarnane. See pole ka kuigi üllatav, sest algoritmi põhiline iva on uuele elemendile sobiva koha leidmine. Selle käigus saame ilma lisatööta tuvastada ka juhu, kui lisatav väärtus puus juba esineb.

[4] See definitsioon eeldab, et puus ei ole korduvaid väärtusi. Sellest eeldusest loobumisel peame täpsustama, kuidas me korduvaid väärtusi hoida tahame, ja muidugi ka järgnevaid algoritme vastavalt täiendama.

Algoritm 8.5 LisaPuusse: Lisamine otsingupuusse
Sisend: t — viit otsingupuu juurele; x — lisatav väärtus
Väljund: Lisab väärtuse x puusse, kui see seal juba ei esine

1. `kui` $t = \bot$
2. `tee-uus` t — tühi puu, lisame uue lehe
3. $[\![t]\!].x \leftarrow x$
4. $[\![t]\!].v \leftarrow \bot$; $[\![t]\!].p \leftarrow \bot$
5. `muidukui` $x < [\![t]\!].x$
6. LisaPuusse($[\![t]\!]$.v, x)
7. `muidukui` $x > [\![t]\!].x$
8. LisaPuusse($[\![t]\!]$.p, x)
9. `muidu`
   — väärtus juba on puus, pole midagi teha
10. `lõppkui`

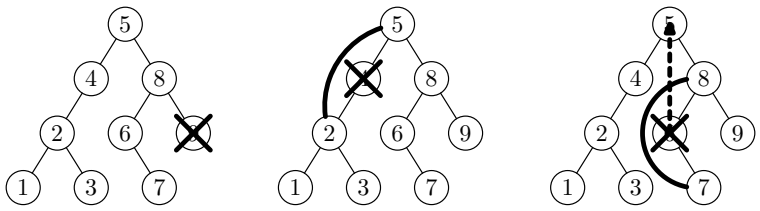
Pole raske näha, et mõlema algoritmi keerukus on võrdeline leitud või lisatud tipu kaugusega puu juurtipust — just nii palju kordi tuleb kummaski algoritmis teha rekursiivseid pöördumisi iseenda poole. Parimal juhul, kui töödeldav puu on täielik kahendpuu, on k-kihilises puus kokku $n = 2^k - 1$ tippu ja n-elemendilise andmehulga töötlemiseks kuluvad $\mathrm{O}(k) = \mathrm{O}(\log n)$ sammu on sel juhul kooskõlas juba varem viidatud kahendotsimise analoogiaga. Paraku kehtib see analoogia ainult selliste puude korral, mille iga tipu vasak ja parem alampuu on ligikaudu sama suured (sellisel juhul ütleme, et puu on tasakaalus).

Ilma spetsiaalseid pingutusi tegemata ei saa me üldjuhul eeldada, et meie otsingupuud tasakaalus püsivad. Näiteks elementide kasvavas või kahanevas järjekorras lisamisel saame tasakaalustatud puu asemel hoopis ahela, milles nii otsimine kui ka lisamine taanduvad lineaarseks ja võtavad $\mathrm{O}(\log n)$ asemel $\mathrm{O}(n)$ operatsiooni.

Selleks, et otsingupuud töö käigus jooksvalt tasakaalus hoida (ja seda piisavalt efektiivselt teha), on välja mõeldud mitmeid kavalaid võtteid. Tuntumad neist on **AVL-puud** (ingl *AVL-tree*) ja **puna-mustad puud** (ingl *red-black tree*). Nende algoritmide detailne kirjeldus on aga käesoleva materjali jaoks natuke liiga pikk.

Antud väärtuse x eemaldamine otsingupuust on veidi keerulisem kui otsimine või lisamine. Kui me oleme eemaldatavat väärtust sisaldava tipu puust üles leidnud, on põhimõtteliselt kolm võimalust:

- x asub lehes (näiteks $x = 9$, joonisel 8.9 vasakul); siis võime selle lehe lihtsalt puust eemaldada;
- x asub ühe alluvaga sõlmes ($x = 4$, joonisel keskel); siis võime sõlme puust eemaldada ja tema ainsa alluva tema asemele panna (on lihtne veenduda, et puu jääb endiselt otsingupuuks);
- x asub kahe alluvaga sõlmes ($x = 5$, joonisel paremal); siis ei saa me seda sõlme puust lihtsalt eemaldada, sest kahe vabaks jääva alluva hoidmiseks pole eemaldatava tipu ülemusel vabu viitasid; selle asemel leiame puust minimaalse eemaldatavast väärtusest suurema väärtuse x' (näites $x' = 6$), asendame eemaldatava väärtuse x tema asukohas väärtusega x' ja eemaldame selle asemel x' tema endisest asukohast (on lihtne veenduda, et puu jääb endiselt otsingupuuks; lisaks on lihtne veenduda, et x' ei saa asuda kahe alluvaga sõlmes, seega tema kustutamiseks pole enam vaja suuruselt järgmist elementi otsida; algoritmi õigsuse seisukohalt pole viimane ajaolu oluline).



Joonis 8.9: Elemendi eemaldamine otsingupuust

Algoritm 8.6 EemaldaPuust: Eemaldamine otsingupuust
Sisend: t — viit otsingupuu juurele; x — eemaldatav väärtus
Väljund: Eemaldab väärtuse x puust, kui see seal esineb

```
1.  kui t = ⊥
        — tühi puu, pole midagi teha
2.  muidukui x < [[t]].x
3.      EemaldaPuust([[t]].v, x)
4.  muidukui x > [[t]].x
5.      EemaldaPuust([[t]].p, x)
6.  muidu   — vaja on eemaldada tipp t
7.      kui [[t]].p = ⊥
8.          t' ← t; t ← [[t]].v   — haagime t puust lahti
9.          kustuta t'   — ja kustutame mälust
10.     muidu
11.         [[t]].x ← Minimaalne([[t]].p)   — asendame naaberväärtusega
12.         EemaldaPuust([[t]].p, [[t]].x)   — ja eemaldame vana eksemplari
13.     lõppkui
14. lõppkui
```

Algoritm 8.7 Minimaalne: Leiab otsingupuust minimaalse väärtuse
Sisend: t — viit otsingupuu juurele, ei või olla $\perp$

```
1. kui [[t]].v = ⊥
2.     tagasta [[t]].x
3. muidu
4.     tagasta Minimaalne([[t]].v)
5. lõppkui
```

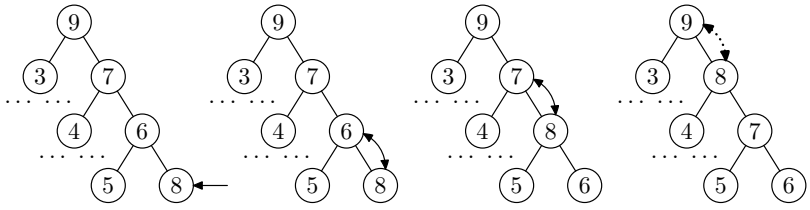
Teema lõpetuseks võiks veel märkida, et otsingupuu läbimisel keskjärjestuses külastame selle tippe nende väärtuste kasvamise järjekorras. See omadus järeldub üsna lihtsalt otsingupuu definitsioonist.

8.2.4 Kahendkuhi

Puud nimetatakse **kuhjaks** (ingl *heap*), kui tema ühegi tipu väärtus ei ületa selle tipu vahetu ülemuse väärtust. Kahendpuud, mis on samal ajal ka kuhi, nimetatakse muidugi kahendkuhjaks.

Vahetult kuhja definitsioonist järeldub, et mistahes kuhjas peab maksimaalse väärtusega element asuma juurtipus. See tähelepanek ongi aluseks, et konstrueerida kahendkuhjal põhinev andmestruktuur, millesse saab efektiivselt andmeid juurde lisada ja suvalisel hetkel kiiresti leida parasjagu kuhjas olevate elementide maksimumi.

Kuhja selliseks kasutamiseks on otstarbekas hoida kuhja elemente kompaktse kahendpuu massiiviesitusena. Uue elemendi lisamisel kuhja lisame selle kõigepealt massiivi lõppu (mis selle massiivi puuna tõlgendamise terminites tähendab, et me lisame puule uue lehe; joonisel 8.10 vasakul). Muidugi võib juhtuda, et uus leht ei rahulda kuhja tingimust.



Joonis 8.10: Elemendi lisamine kahendkuhja

Kuhja taastamiseks hakkame lisatud elementi puus ülespoole nihutama. Kui uus leht on oma ülemusest suurem, siis võime need kaks lelementi lihtsalt omavahel vahetada — siis on kuhja omadus vähemalt selles paaris taas rahuldatud. Kui vastlisatud element on suurem ka oma uuest ülemusest, vahetame ta veelkord ülespoole — nii jätkame, kuni ühel hetkel jõuame piisavalt suure ülemuseni või kuni uus element jõuab puu juurtippu.

Ei tohiks olla raske veenduda, et kui esialgne puu oli kuhi, siis on seda ka uue elemendi üles viimisel saadud puu.

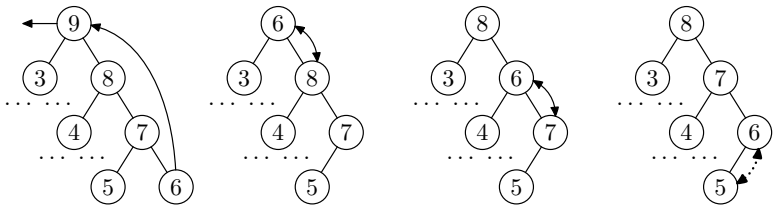
Algoritm 8.8 ViiUles: Viib kahendkuhja lisatud lehe selle õigele kohale
Sisend: $a_{1\ldots n}$ — puu massiiviesitus, kus $a_{1\ldots n-1}$ moodustavad kuhja
Väljund: a_n viidud kuhjas oma õigele kohale

1. $i \leftarrow n$
2. `senikui` $i > 1$
   — invariant: $a_{1..n}$ miinus a_i alampuu on kuhi $\wedge$ a_i alampuu on kuhi
3. $k \leftarrow \lfloor i/2 \rfloor$
4. `kui` $a_k < a_i$
5. $a_i \leftrightarrow a_k$ — a_i on oma ülemusest suurem, tõrjub selle allapoole
6. `lõppkui`
7. $i \leftarrow k$
8. `lõppsenikui`

Kuhjast maksimaalse elemendi väljavõtmisel tekib samasugune probleem nagu otsingupuust elemendi eemaldamisel — me ei saa eemaldatud elemendi kohta tühjaks jätta. Esimene mõte — tuua juurtipu uueks väärtuseks tema

kahest alluvast suurem — ei ole pikas perspektiivis hea lahendus, sest siis võib uus vaba koht puu lehtede hulgas tekkida ükskõik kuhu ja järgmisel lisamisel on raske seda leida.

Selleks, et vaba koht jääks kindlasti puu massiiviesituse lõppu, toome puu juurtipu uueks väärtuseks justnimelt massiivi viimase elemendi (ja selle koht omakorda jääb vabaks; joonisel 8.11 vasakul). Kuna see on puu leht ja seega üsna väikese väärtusega, on tõenäoline, et sellega saab jälle puu kuhjaomadus rikutud.



Joonis 8.11: Elemendi eemaldamine kahendkuhjast

Kuhja taastamiseks hakkame nüüd vastset juurtipu kuhjas allapoole viima: igal sammul valime uue elemendi ja tema alluvate hulgast maksimaalse ning tõstame selle lokaalseks ülemuseks. Nagu elemendi üles viimisel, järgneme ka seekord vahetuse korral uuele elemendile ja jätkame võrdlemist ja vahetamist, kuni puu on taas kuhi.

Algoritm 8.9 VIIALLA: Viib kahendkuhja lisatud juure selle õigele kohale
Sisend: $a_{1\ldots n}$ — puu massiiviesitus, kus $a_{2\ldots n}$ moodustavad kaks kuhja
Väljund: a_1 viidud kuhjas oma õigele kohale

1. $i \leftarrow 1$
2. `senikui` $i < n$
   — invariant: $a_{1\ldots i}$ on kuhi $\wedge$ a_i alampuu oleks kuhi, kui a_i oleks ∞
3. $\quad k \leftarrow i$
4. $\quad$ `kui` $2 \cdot i \leq n \wedge a_{2 \cdot i} > a_k$
5. $\quad\quad k \leftarrow 2 \cdot i$
6. $\quad$ `lõppkui`
7. $\quad$ `kui` $2 \cdot i + 1 \leq n \wedge a_{2 \cdot i + 1} > a_k$
8. $\quad\quad k \leftarrow 2 \cdot i + 1$
9. $\quad$ `lõppkui`
10. $\quad$ `kui` $k > i$
11. $\quad\quad a_i \leftrightarrow a_k$ — a_i on oma alluvast väiksem, see tõrjub ta allapoole
12. $\quad$ `lõppkui`

13. $i \leftarrow k$
14. `lõppsenikui`

Kahe eeltoodud võtte kombineerimisel saame efektiivse algoritmi massiivide sortimiseks: kõigepealt muudame massiivi kuhjaks, "lisades" selle elemente järjest kuhja, mida hoiame sama massiivi algusosas; seejärel võtame kuhjast maksimaalse elemendi, taastame kuhja, võtame allesjäänutest maksimaalse, taastame uuesti kuhja, ja nii edasi, kuni olemegi kõik elemendid järjestanud.

Algoritm 8.10 KUHJASORT
Sisend: $a_{1\ldots n}$ — töödeldav massiiv
Väljund: $a_{1\ldots n}$ — väärtused vahetatud nii, et $\forall i < n : a_i \leq a_{i+1}$

— muudame massiivi kuhjaks
1. `korda` $i \leftarrow 1 \ldots n$
   — invariant: $a_{1\ldots i-1}$ juba on kuhi
2. VIIULES($a_{1\ldots i}$)
3. `lõppkorda`
   — muudame kuhja järjestatud massiiviks
4. `korda` $i \leftarrow n \ldots 1$
   — invariant: $a_{1\ldots i}$ on kuhi
5. $a_i \leftrightarrow a_1$
6. VIIALLA($a_{1\ldots i-1}$)
   — vahetingimus: $a_i \leq a_{i+1} \leq \ldots \leq a_n$
7. `lõppkorda`

Kuna n-tipulises kompaktses puus võib element nii üles kui alla liikuda ülimalt $O(\log n)$ sammu, saame kummagi korduse ajalise keerukuse hinnanguks $O(n \log n)$.

8.3 Graafialgoritmid

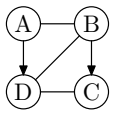
8.3.1 Graafi esitamine arvuti mälus

Kaks tähtsamat võtet graafide esitamiseks arvuti mälus on naabrusmaatriks ja kaareloendid. Mõlemad võtted sobivad paremini orienteeritud graafide esitamiseks. Orienteerimata servi esitatakse tavaliselt kahe vastassuunalise kaarena. See tähendab, et iga serva $\{v_1, v_2\}$ asemel pannakse graafi kaks kaart (v_1, v_2) ja (v_2, v_1). Kui pidada silmas ainult graafi ühelt tipult teisele liikumise võimalikkust, siis ei muuda selline asendus midagi. Kui aga mõnes ülesandes on vaja servi graafist eemaldada või vältida sama serva korduvat läbimist, tuleb sellised paariskaared kuidagi eraldi tähistada.

Graafi G **naabrusmaatriks** (ingl *adjacency matrix*) on ruutmaatriks, milles on üks rida ja üks veerg graafi G iga tipu kohta. Tipule v_1 vastava rea ja tipule v_2 vastava veeru ristumiskohas on info kaare (v_1, v_2) kohta.

Sellist maatriksit on kõige mugavam hoida 2-mõõtmelise massiivina. Lihtsaimal juhul on massiivi elemendid tõeväärtused või täisarvud, ning kaare olemasolu tähistab siis väärtus 1 ja selle puudumist väärtus 0 massiivi vastavas elemendis (nagu näha joonisel 8.12 toodud näites). Vajadusel võib luua ka massiivi, mille elemendid on keerukamad kirjed, ja hoida iga kaare kohta rohkem andmeid.

Täielikult orienteerimata graafi puhul kasutatakse naabrusmaatriksi asemel mõnikord "naabruskolmnurka" — maatriksist on kasutusel ainult selle peadiagonaali kohal olev osa. Kuna orienteerimata graafi naabrusmaatriks on alati peadiagonaali suhtes sümmeetriline, siis pole sel juhul diagonaali alla jääva osa andmetega täitmine vajalik.

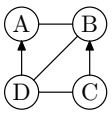


	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	0	1	1	0

Joonis 8.12: Segagraaf ja tema naabrusmaatriks

Graafi **esitus kaareloenditena** (ingl *adjacency-lists representation*) annab iga tipu kohta sellest tipust väljuvate servade loetelu.

Selliseid loetelusid on tavaliselt mugav hoida lihtahelates, ja kogu graafi esitus koosneb siis massiivist või ahelast, milles on üks element graafi iga tipu kohta. Lisaks otseselt tipu enda andmetele sisaldab see element ka viita tipust väljuvate servade loetelule (nagu näha joonisel 8.13 toodud näites), ja iga serva element omakorda viitab mingil viisil selle serva lõpptipule.



$$
\begin{array}{lcl}
A & \rightarrow & (B) \\
B & \rightarrow & (A,\ D) \\
C & \rightarrow & (B,\ D) \\
D & \rightarrow & (A,\ B,\ C)
\end{array}
$$

Joonis 8.13: Segagraaf ja tema kaareloendid

Kaareloendeid on parem kasutada sellistes algoritmides, kus on vaja kiiresti leida kõik antud tipust väljuvad kaared (või kõik tipud, kuhu antud tipust

edasi liikuda saab). Naabrusmaatriksid on tõhusamad algoritmides, kus on vaja kiiresti kontrollida, kas kahe antud tipu vahel on kaar või mitte.

Mälukasutuse seisukohalt on naabrusmaatriksid efektiivsemad tihedate ja kaareloendid hõredate graafide esitamisel.[5]

8.3.2 Graafi läbimine laiuti

Nagu puude korral, nii on ka paljude graafiülesannete lahenduse võti graafi tippude (ja servade) õiges järjekorras läbimises. Sageli osutub selleks õigeks järjekorraks graafi **läbimine laiuti** (ingl *breadth-first search*).

Laiuti läbimine (algoritm 8.11) algab mingist lähtetipust (joonisel 8.14 tipp A) ja vaatleb juba leitud tippudest väljuvaid servi või kaari, et leida uusi, seni avastamata tippe.[6] Uusi tippe töödeldakse nende avastamise järjekorras, seetõttu moodustavad avastatud, aga veel töötlemata tipud (algoritmis järjekorra Q sisu, joonisel tähistatud helehalliga) lähtetipu ümber omamoodi "laineharja" või "rindejoone", mis sellest igas suunas võrdse kiirusega eemaldub. Viimase asjaolu järgi ongi laiuti läbimine oma nime saanud.

Algoritm 8.11 LabiGraafL: Läbib graafi laiuti, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$; tippude järjekord Q

1. `korda` $v \in V(G)$ — kõik V tipud
2. $\quad v.olek \leftarrow valge$
3. `lõppkorda`
4. $Q \leftarrow v_0$; $v_0.olek \leftarrow hall$
5. `senikui` $Q \neq \emptyset$ — järjekord pole tühi
6. $\quad v \leftarrow Q$ — võtame järjekorra esimese elemendi
7. $\quad$ `korda` $w \in V(v)$ — kõik v naabertipud
8. $\quad\quad$ `kui` $w.olek = valge$ — uus tipp
9. $\quad\quad\quad Q \leftarrow w$; $w.olek \leftarrow hall$
10. $\quad\quad$ `lõppkui`
11. $\quad$ `lõppkorda`
12. $\quad v.olek \leftarrow must$ — v töödeldud
13. `lõppsenikui`

[5] **Tihedaks** (ingl *dense*) nimetatakse graafi, mille mille tipupaaridest enamik on omavahel ühendatud. **Hõredaks** (ingl *sparse*) nimetatakse graafi, mille tipupaaridest enamik on omavahel ühendamata.

[6] Siin ja ka järgnevates näidetes eeldame, et igast tipust väljuvaid servi vaadeldakse naabertippude nimede tähestikulises järjekorras. Algoritmid töötavad õigesti ka ilma selle eelduseta, aga näited tuleks teise läbivaatusjärjekorra puhul veidi erinevad.

8.3.3 Lühima tee leidmine

Sidusa graafi laiuti läbimisel tippude avastamiseks kasutatud servad (joonisel 8.14 märgitud paksema joonega) moodustavad selle graafi **toesepuu** (ingl *spanning tree*) — puu, mille tippude hulk langeb kokku graafi G tippude hulgaga ja mille servade hulk on graafi servade hulga alamhulk.

Graafil võib olla mitmeid erinevaid toesepuid, kuid just laiuti läbimisel saadud toesepuu kasulik omadus on, et igast tipust mööda sellesse puusse kuuluvaid servi lähtetippu viiv tee ongi minimaalse võimaliku pikkusega, kui me loeme tee pikkuseks sellele jäävate servade arvu. Seega on graafis lühimate teede leidmiseks vaja laiuti läbimist vaid minimaalselt modifitseerida.

Algoritm 8.12 LyhTeed: Leiab lühimad teed, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutujad $v.kaug$ ja $v.eelm$;
tippude järjekord Q

1. `korda` $v \in V(G)$ — kõik V tipud
2. $v.kaug \leftarrow \infty$
3. `lõppkorda`
4. $Q \leftarrow v_0$; $v_0.kaug \leftarrow 0$; $v_0.eelm \leftarrow \perp$
5. `senikui` $Q \neq \emptyset$ — järjekord pole tühi
6. $v \leftarrow Q$ — võtame järjekorra esimese elemendi
7. `korda` $w \in V(v)$ — kõik v naabertipud
8. `kui` $w.kaug = \infty$ — uus tipp
9. $Q \leftarrow w$; $w.kaug \leftarrow v.kaug + 1$; $w.eelm \leftarrow v$
10. `lõppkui`
11. `lõppkorda`
12. `lõppsenikui`

Algoritmi 8.12 õigsuse põhjendamiseks paneme tähele, et

- läbimise algatamine real 4 on ilmselt õige, sest lähtetipu v_0 kaugus iseendast on 0;
- real 5 algav kordus töötleb graafi tippe nende lähtetipust kaugenemise järjekorras; täpsemalt, kõik tipud, mille kaugus lähtetipust on k töödeldakse enne kui ükski tipp, mille kaugus on $k+1$;
- viitade $v_k.eelm = v_{k-1}$, $v_{k-1}.eelm = v_{k-2}$, $\ldots$, $v_1.eelm = v_0$ poolt näidatav tee mistahes tipust v_k lähtetippu v_0 on minimaalse sammude arvuga; kui see nii ei oleks, siis peaks sellel teel leiduma tipp v_i, mille viit $v_i.eelm$ osutab tipule v_{i-1} ja millel on olemas ka naaber v_j, mille kaugus lähtetipust on väiksem kui $i-1$; siis aga oleks v_j töödeldud

enne kui v_{i-1} ja tipp v_i avastatud serva $\{v_j, v_i\}$ kaudu; kuna seda ei juhtunud, siis järelikult sellist tippu v_j ei ole, ja leitud tee ongi lühim võimalik.

Kuna algoritm töötleb graafi G iga tippu täpselt ühe korra ja vaatleb iga serva täpselt kaks korda, siis kulub N tipu ja M servaga graafi töötlemiseks kokku $O(N + M)$ operatsiooni.

Selle algoritmi tööd illustreerib joonis 8.15, kus tippudele on märgitud nende kaugused lähtetipust ja nooled osutavad igast tipust tema ülemusele toesepuus.

Teema lõpetuseks võiks veel märkida, et algoritmi 8.12 on võimalik üldistada ka juhule, kui graafi servadele on antud mittenegatiivsed kaalud ja tee pikkuseks loetakse mitte teel olevate servade arvu, vaid nende kaalude summat. Seda üldistust tuntakse autori järgi **Dijkstra algoritmi** nime all, aga selle täpsemaks kirjeldamiseks ja põhjendamiseks ei jätku siinkohal ruumi.

8.3.4 Graafi läbimine sügavuti

Laiuti läbimise kõrval on teine tähtsam viis graafi töötlemiseks selle **läbimine sügavuti** (ingl *depth-first search*). Sarnaselt laiuti läbimisega algab ka sügavuti läbimine (algoritmid 8.13 ja 8.14) mingist lähtetipust ja vaatleb uute tippude leidmiseks juba avastatud tippudest väljuvaid servi või kaari. Erinevus seisneb selles, et iga uue tipu avastamisel töödeldakse sellest tipust algav "haru" enne järgmise juurde asumist lõpuni.

Algoritm 8.13 LabiGraafS: Läbib graafi sügavuti, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$, $v.alust$, $v.lopet$;
globaalne loendur t

1. `korda` $v \in V(G)$ — kõik V tipud
2. $v.olek \leftarrow valge$
3. `lõppkorda`
4. $t \leftarrow 0$
5. TootleTippS(v_0)

Algoritm 8.14 TootleTippS: Läbib sügavuti antud tipu "alampuu"
Sisend: $v \in V(G)$

1. $v.olek \leftarrow hall$; $t \leftarrow t + 1$; $v.alust \leftarrow t$ — v töötlemise algus
2. `korda` $w \in V(v)$ — kõik v naabertipud
3. `kui` $w.olek = valge$ — uus tipp
4. TootleTippS(w)

5. `lõppkui`
6. `lõppkorda`
7. $v.olek \leftarrow must$; $t \leftarrow t+1$; $v.lopet \leftarrow t$ — v töötlemise lõpp

Sarnaselt laiuti läbimisega eraldab ka sügavuti läbimine töödeldavast graafist toesepuu, aga see puu hoopis teise kujuga (vrdl jooniseid 8.14 ja 8.16). Sarnaselt laiuti läbimisega on ka sügavuti läbimise keerukus $O(N+M)$.

8.3.5 Topoloogiline sorteerimine

Paljudes sügavuti läbimisel põhinevates algoritmides on kasulik tähele panna iga tipu töötlemise alguse (algoritmi 8.14 rida 1) ja lõpu (rida 7) aega. Nende aegade omavahelised suhted erinevate tippude vahel on tihedalt seotud graafi läbimise käigus eraldatava toesepuu struktuuriga.

Üks ülesandeid, mille lahendamisel tippude läbimise aegade võrdlemine kasuks tuleb, on suunatud graafi tippude sorteerimine **topoloogilisse järjekorda** (ingl *topological order*). Topoloogiliseks nimetatakse graafi tippude järjestust, mille korral kõik kaared osutavad järjekorras eespool olevatelt tippudelt tagapool olevatele, aga mitte kunagi vastupidi.

Topoloogilist sorteerimist on vaja näiteks üksteisest sõltuvate tööde järjekorra planeerimisel: kui me esitame iga töö graafi tipuna ja tõmbame igasse tippu kaared kõigist neist, mis peavad enne selle töö alustamist tehtud olema, siis saame töid topoloogilises järjekorras ette võttes kõik õigesti tehtud.

Algoritm 8.15 TopoSort: Leiab suunatud graafis topoloogilise järjestuse
Sisend: graaf G
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$; tippude magasin S

1. `korda` $v \in V(G)$ — kõik V tipud
2. $v.olek \leftarrow valge$
3. `lõppkorda`
4. $S \leftarrow \emptyset$
5. `korda` $v \in V(G)$ — kõik V tipud
6. `kui` $v.olek = valge$ — uus juurtipp
7. TopoTipp(w)
8. `lõppkui`
9. `lõppkorda`
10. `senikui` $S \neq \emptyset$ — magasin pole tühi
11. $v \leftarrow S$ — võtame magasini tipmise elemendi
12. `väljasta` v
13. `lõppsenikui`

Algoritm 8.16 TopoTipp: Läbib sügavuti antud tipu "alampuu"
Sisend: $v \in V(G)$

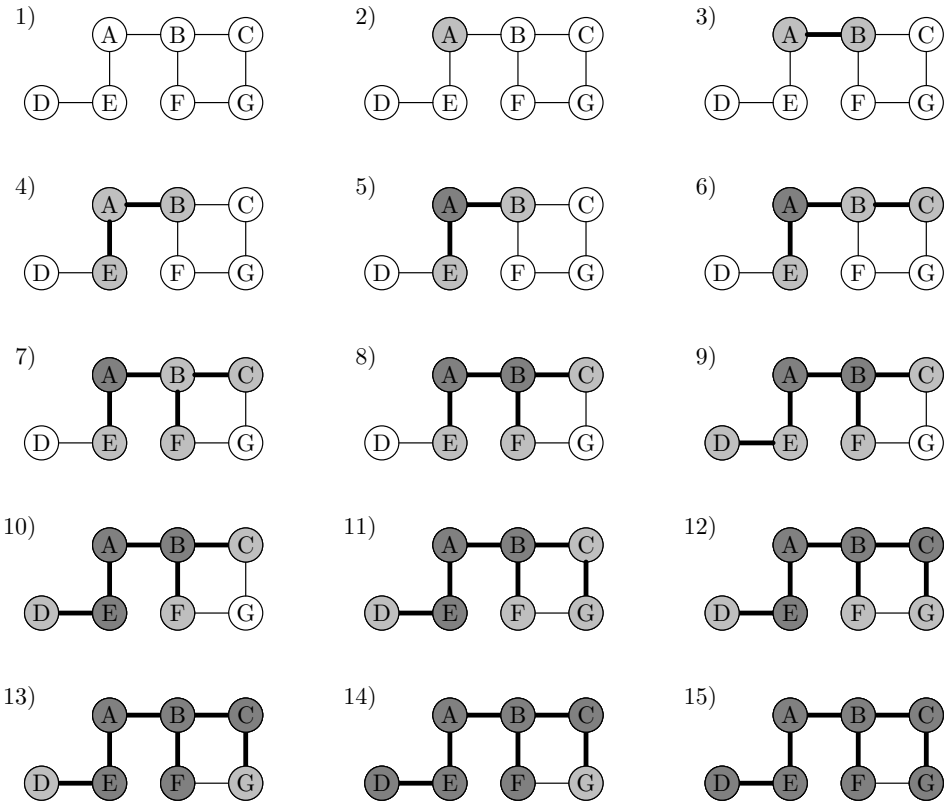
1. $v.olek \leftarrow hall$
2. `korda` $w \in V(v)$ — kõik v naabertipud
3. `kui` $w.olek = valge$ — uus tipp
4. TopoTipp(w)
5. `lõppkui`
6. `lõppkorda`
7. $v.olek \leftarrow must$; $S \leftarrow v$ — töödeldud tipp magasini

Algoritmi 8.15 õigsuse põhjendamiseks tuleb tähele panna, et

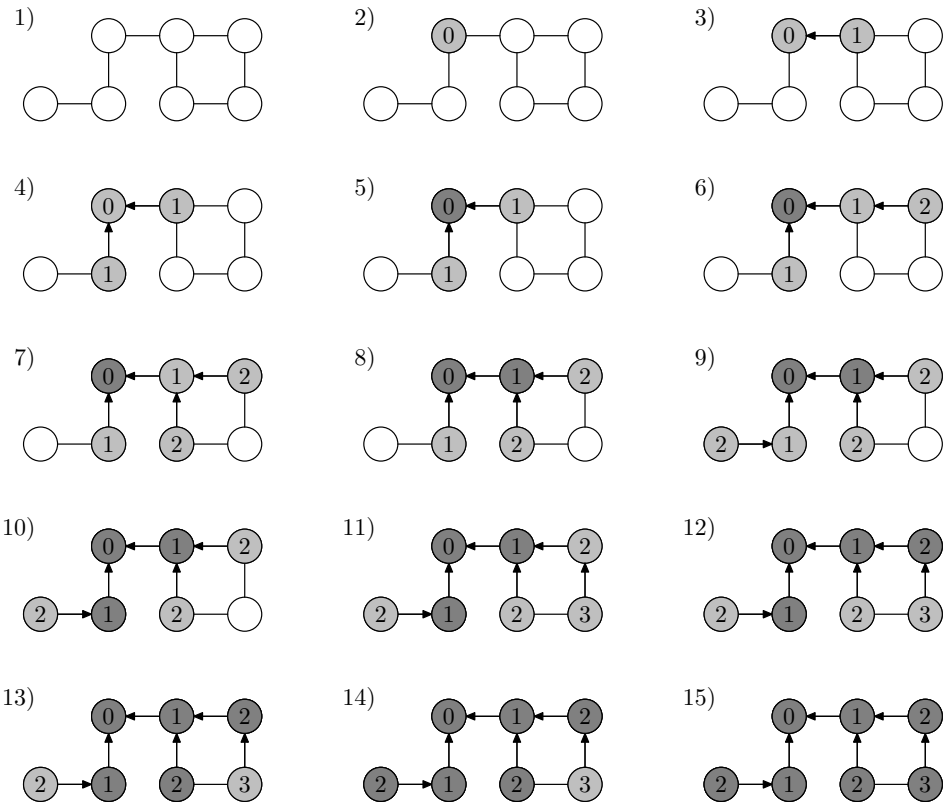
- väljakutse TopoTipp(v) töötleb kõik need tipud, mis on tipust v graafi servi (või kaari) pidi liikudes saavutatavad ja mis veel ei ole töödeldud;
- seega on väljakutse TopoTipp(v) täitmise lõpuks töödeldud kõik need tipud, mis ei tohi topoloogilises järjestuses tipust v eespool olla;
- seega tuleks topoloogilise järjestuse saamiseks tipud väljastada nende töötlemise lõpuaegade kahanemise järjekorras;
- seda aga TopoSort teebki, kuna TopoTipp väljakutsed panevad tipud nende töötlemise lõppedes magasini ja TopoSort väljastab nad magasinist väljavõtmise järjekorras (mis vastavalt magasini definitsioonile on vastupidine nende magasini panemise järjekorrale.

Algoritmi 8.15 tööd illustreerib joonis 8.17.

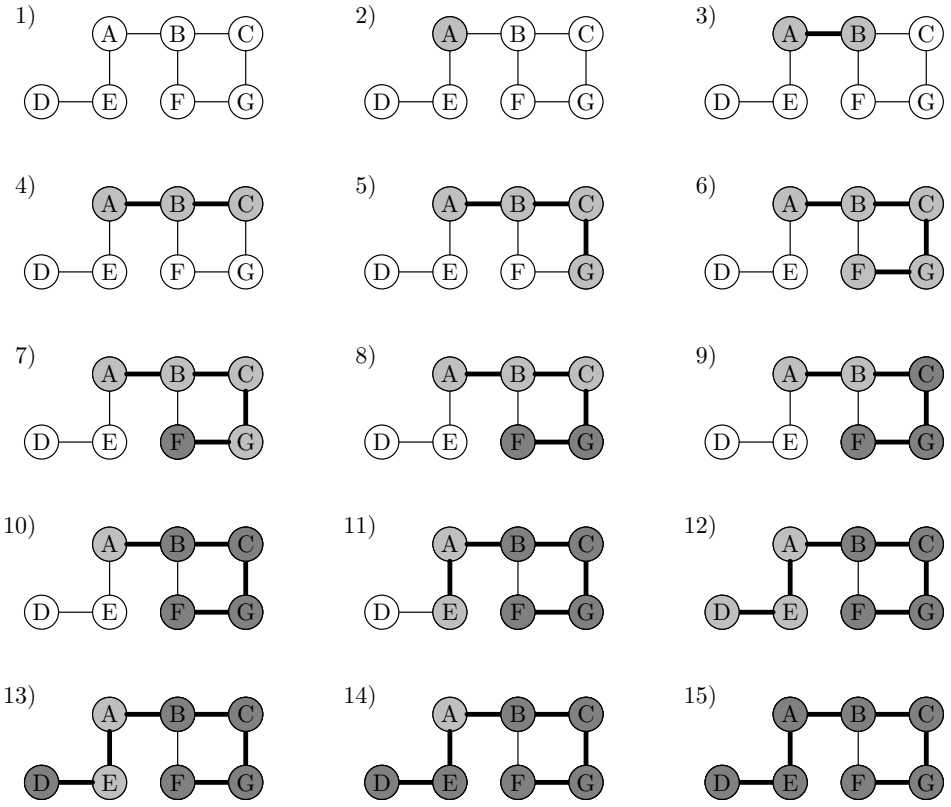
Muidugi ei saa topoloogiliselt järjestada graafi, milles on tsüklid. Näiteks graafis $G = (V, E)$, kus $V = \{A, B, C\}$ ja $E = \{(A, B), (B, C), (C, A)\}$, ei saa ükski tipp ei saa olla topoloogilises järjekorras esimene. Osutub, et efektiivne viis graafi tsüklilisuse kontrollimiseks ongi püüda teda topoloogiliselt sorteerida ja kontrollida, et selle käigus vastuolu ei teki.



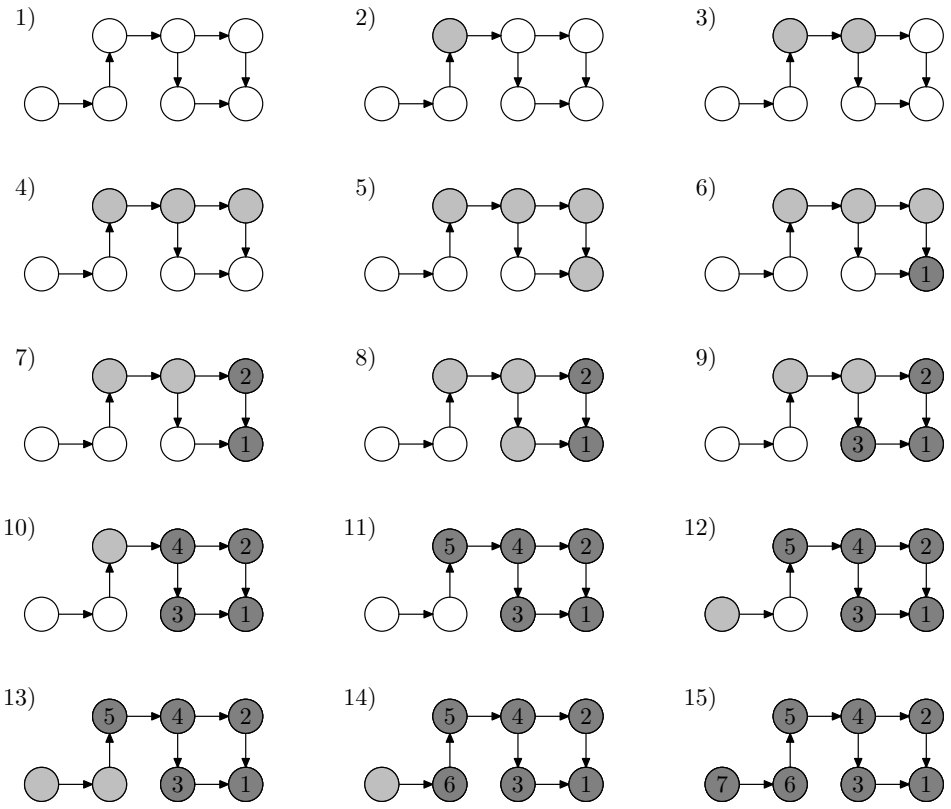
Joonis 8.14: Graafi läbimine laiuti



Joonis 8.15: Lühimate teede leidmine graafis



Joonis 8.16: Graafi läbimine sügavuti



Joonis 8.17: Graafi topoloogiline sorteerimine

Ülesanded

Ülesanne 8.1 *Kirjutada alamprogramm* TrykiPuu, *mis saab kahendpuu loomuliku esituse ja väljastab suluavaldise, mis sobib selle puu uuesti sisselugemiseks näidete hulgas toodud alamprogrammi* LoePuu *abil. (Vihje: millises järjekorras peavad puu tipud selles suluavaldises olema?)*

Ülesanne 8.2 *Koostada algoritm ja kirjutada programm loomulikus esituses antud kahendpuu tippude läbimiseks "sügavuse järjekorras": kõigepealt juurtipp, siis selle vahetud alluvad, siis nende vahetud alluvad jne. Näiteks kõigis kolmes joonisel 8.4 toodud puus tuleks tipud läbida nende märgendite tähestikulises järjekorras.*

Ülesanne 8.3 *Koostada algoritm, mis leiab naabrusmaatriksina antud suunatud graafi iga tipu jaoks temast väljuvate ja temasse sisenevate kaarte arvu. Milline on selle algoritmi tööaeg N tipu ja M servaga graafi töötlemisel?*

Kohandada see algoritm kaareloenditena antud graafile. Milline on tema tööaeg nüüd?

Ülesanne 8.4 *Koostada laiuti läbimisel põhinev algoritm ja kirjutada selle alusel programm, mis leiab naabrusmaatriksina antud suunamata graafi sidususkomponendid. Vastusena väljastada komponentide arv ja igasse komponenti kuuluvate tippude loetelu.*

Ülesanne 8.5 *Millise tulemuse annab algoritmi 8.15 rakendamine suunatud graafile, milles on tsükkel? Täiendada õppematerjali lisas toodud programmi* TopoSort *nii, et tsükleid sisaldava graafi töötlemisel oleks tulemuseks asjakohane veateade.*

Kirjandus

Kõik eelmises peatükis viidatud materjalid käsitlevad vähemalt mingil määral ka mittelineaarseid struktuure. Samuti võib puude ja graafidega seotud peatükke leida paljudest kombinatoorikaõpikutest, sealhulgas ka käesoleva õppevahendi kombinatoorikateemas viidatavatest. Seetõttu on allpool välja toodud vaid mõned konkreetselt graafiteooriale keskenduvad õpikud.

Peeter Puusemp. *Graafiteooria elemente.* Tallinna Tehnikaülikool, 2000.
Üldine sissejuhatus graafiteooriasse. 96 lk.

Ahto Buldas, Peeter Laud, Jan Villemson. *Graafid.* Tartu Ülikool, 2003.
Üldine sissejuhatus graafiteooriasse. 92 lk.

Oystein Ore. *Graafid ja nende kasutamine.* Valgus, 1976.
Keskendub rohkem graafiteooria matemaatilisele poolele, programmeerimisele eriti tähelepanu ei pööra. 139 lk.

Frank Harary. *Graph Theory.* Addison-Wesley, 1969.
Klassikaline graafiteooria õpik, samuti matemaatilise kallakuga. 274 lk.

Reinhard Diestel. *Graph Theory.* Springer, 2000.
Üsna uus klassikalise graafiteooria õpik. 315 lk.
`http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/`

Kokkuvõtteks

Õppematerjaliga lõpule jõudnud lugejale tahaksin öelda kahte asja.

Esiteks, ära on tehtud päris suur hulk tööd; muidugi väärivad kiitust eriti need, kes on lisaks teksti lugemisele ka kõik ülesanded ära lahendanud.

Ja teiseks, kindlasti ei maksa arvata, et sellega ongi kogu programmeerimine selge. Informaatika on väga lai valdkond ja areneb meeletu kiirusega. Kindlasti ei jõua keegi seda täielikult omandada, aga sellele vaatamata peab vormis püsimiseks ennast pidavalt arendama.

Selleks soovin kõigile jõudu, püsivust ja vastuvõtlikku meelt.