3 ALGORITMI KEERUKUS JA PÕHILISED ANDMESTRUKTUURID

Arvutiprogrammid lahendavad mitmesuguseid ülesandeid arvutuste, andmetöötluse ja automaatsete otsuste tegemise alal. Selleks on programmil vaja sooritada hulk tegevusi. Nende tegevuste järjestatud jada nimetatakse **algoritmiks**.

Sõna „algoritm“ ise pärineb Pärsia matemaatiku al-Khwārizmī (780-850) nimest. 825. aasta paiku kirjutas ta araabiakeelse töö, milles kirjeldas arvutamist kümnendsüsteemis. 12. sajandil tõlgiti see ladina keelde pealkirjaga *„Algoritmi de numero Indorum“* ehk „Algoritmi indialaste numbritest“. „Algoritmi“ oli siin lihtsalt al-Khwārizmī nime kohandus ning indialaste numbrid olid need, mida Euroopas hakati tundma araabia numbrite nime all – nulli sisaldavad kümnendsüsteemi numbrid. Edaspidi tähistaski algoritmi termin peamiselt kümnendsüsteemi ja seonduvaid arvutusreegleid.

Tolleaegne arvutusreeglistik (ehk algoritm!) oli esitatud värsivormis ja moodustas mitmesajarealise poeemi. Luulevorm pole aga täpsete ja keeruliste eeskirjade jaoks kõige efektiivsem, seetõttu on hilisemad teadlased uurinud meetodeid algoritmide kiiremaks ja kompaktsemaks esitamiseks.

19. sajandil leiutasid George Boole ja Giuseppe Peano aritmeetika kirjapanemiseks konkreetsete reeglite ja sümbolitega süsteemi, sellest ajast alates võib ka rääkida algoritmidest tänapäevases tähenduses. Arvutiteaduse seisukohast kõige olulisema aluse pani algoritmiteooriale 1930. aastatel Alan Turing, kes kirjeldas **Turingi masina** idee**.** Turingi masin on teoreetiline masin, mille eesmärkideks on:

1. võimalus täita kõiki võimalikke algoritme,
2. võimalikult väike masinasse ehitatud operatsioonide arv.

Tänapäeval ongi arvutussüsteemide hindamise oluliseks kriteeriumiks see, kas süsteemi abil on võimalik luua Turingi masinat – see näitab, et süsteem on üldotstarbeliselt kasutatav igasuguste algoritmide täitmiseks.

3.1 ALGORITMI KEERUKUS

Programmeerimisvõistlustel tuleb välja mõelda ja programmeerida algoritm, mis antud ülesande lahendab. Samas ainult korrektsest algoritmist ei piisa – programm peab vastuse leidma etteantud aja jooksul ning see ei tohi kasutada ka ülemääraselt palju mälu. Nii aja- kui ka mälulimiit on harilikult iga ülesande juures välja toodud. Seetõttu on kasulik enne koodi kirjutama asumist hinnata, kas väljamõeldud lahendus töötab piisavalt kiiresti ega kasuta liiga palju mälu.

Programmi töökiirus sõltub mitmetest teguritest. Mõned neist on vähemalt osaliselt programmeerija kontrolli all, teised mitte. Näiteks ei saa programmeerija üldjuhul muuta arvuti protsessori töökiirust, kuid ta saab valida ülesande lahendamiseks sobivama algoritmi. Seetõttu on programmeerija jaoks oluline, milline on tema valitud algoritmi **ajaline keerukus** (*time complexity*).

Algoritmi ajaline keerukus sõltub enamasti olulisel määral algandmete mahust – näiteks suure faili pakkimine võtab harilikult kauem aega kui väikese faili pakkimine ja miljonielemendilise jada sorteerimine on aeganõudvam kui kümnest elemendist koosneva jada sorteerimine. Algoritmi ajalist keerukust väljendatakse funktsiooniga f, mis seab igale selle algoritmi järgi lahendatavale

konkreetsele ülesandele andmemahuga n vastavusse selle lahendamiseks sooritatavate operatsioonide arvu $f(n)$.

Teades operatsioonide arvu $f(n)$ ja protsessori töökiirust, saab ligikaudu arvutada, kui palju aega konkreetse ülesande lahendamiseks kulub. Erinevate operatsioonide täitmine võib aga võtta erineva hulga aega, seetõttu kasutatakse **elementaaroperatsiooni** mõistet. Elementaaroperatsiooniks nimetatakse sellist operatsiooni, mille arvuti suudab täita konstantse ajaga, sõltumata argumendi väärtusest. Näiteks kuni 64-bitiste arvude liitmine, lahutamine ja korrutamine on kaasaegses arvutis elementaaroperatsioonid, neist pikemate arvudega opereerimine aga tõenäoliselt mitte. Küll aga võib algoritmide omavahelisel võrdlemisel operatsiooni mõistet veelgi lihtsustada või võrreldagi ainult mingit konkreetset tüüpi operatsioonide arvu. Sel juhul saame üldiselt hinnata, kumb algoritm on kiirem, kuid mitte arvutada ligikaudset tööaega.

3.1.1 Keskmine ja halvim keerukus

Mõnel juhul ei sõltu algoritmi ajaline keerukus ainult andmemahust, vaid ka sellest, milline on etteantud andmestik. Näiteks kui otsida jadast mingi konkreetse väärtusega elementi, siis üheks võimalikuks lahenduseks on käia kogu jada elementhaaval läbi. Parimal juhul on otsitava väärtusega element kohe jada esimesel kohal ja pole mingit põhjust teisi elemente enam läbi vaadata, seega piisas vaid ühest võrdluse operatsioonist. Kui aga otsitava väärtusega element on jadas viimane, tuleb kogu jada läbi vaadata ja n-elemendilises jadas tehakse sel juhul n võrdlemist. Viimane kirjeldatud juhtum on selgelt halvim võimalik, sest kunagi ei ole vaja teha ka rohkem kui n võrdlust. Kirjeldatud olukordi nimetatakse vastavalt algoritmi ajaliseks keerukuseks parimal juhul ja halvimal juhul. Praktikas on sageli kasulik teada ka algoritmi keerukust keskmisel juhul ehk operatsioonide arvu, mida tuleb keskmiselt sooritada konkreetse ülesande lahendamiseks andmemahu n korral. Ülalkirjeldatud jadast otsimise korral tuleb keskmiselt teha $n/2$ võrdlusoperatsiooni.

3.1.2 Asümptootiline hinnang

Kuna väikesed andmemahud enamasti niikuinii probleeme ei tekita, on algoritmi töökiiruse hindamisel kõige olulisemaks see, kuidas töökiirus muutub andmemahu piiramatul kasvul. Sellist hinnangut nimetatakse **asümptootiliseks hinnanguks** ning see keskendubki just keerukusfunktsiooni kasvukiiruse uurimisele. Algoritmi keerukusfunktsiooni kasvukiirus näitab, kui kiiresti kasvab selle algoritmi alusel koostatud programmi tööaeg töödeldavate andmete mahu kasvades. Programmeerimisvõistlustel annab see just aimu, kas väljamõeldud algoritm suudab töödelda vajaliku andmemahu etteantud aja jooksul.

Funktsioonide kasvukiiruse võrdlemisel kasutatakse tavaliselt järgmisi seoseid:

- O (tavaline suurtäht O) tähistab, et funktsioon ei kasva kiiremini võrdlusfunktsoonist.
- Ω (kreeka täht oomega) tähistab, et funktsioon ei kasva aeglasemini võrdlusfunkstioonist.
- Θ (kreeka täht teeta) tähistab seda, et funktsioonid on sama kasvuga.

Formaalselt väljendudes $f(n) \in O(g(n))$, kui funktsiooni $f(n)$ kasvukiirus ei ületa funktsiooni $g(n)$ kasvukiirust. See tähendab, et leiduvad positiivsed konstandid c ja n_0, mille korral kehtib

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

Sümbol ∀ tähendab „iga“. Kogu avaldis tähendab siis, et funktsioon f alates kohast n_0 ei kasva kiiremini kui võrdlusfunktsioon g. Piiri kasutamine on vajalik seetõttu, et väikeste väärtuste puhul võib algoritmi tööajas esineda moonutusi. Tegevused, mida tuleb teha vaid üks kord sõltumata andmemahust (näiteks muutujatele algväärtuse andmine), võivad võtta väikeste

andmemahtude korral lõviosa algoritmi tööajast. Suurte andmemahtude korral on ühekordsetele tegevustele kuluv aeg täpselt sama, kuid see on tühine, võrreldes kogu tööajaga.

Definitsioon ei piira funktsioonide $f(n)$ ja $g(n)$ enda väärtusi, vaid nende väärtuste suhet $f(n)\ /\ g(n)$. Võib öelda ka, et kasv on ülalt tõkestatud. Seetõttu aga võib anda O-hinnanguid ükskõik kui suure liiaga ehk kui $f(n) \in O(n^2)$, siis kehtib ka $f(n) \in O(n^3)$.

$f(n) \in \Omega\big(g(n)\big)$, kui leiduvad positiivsed konstandid c ja n_0, mille korral kehtib

$$\forall n \geq n_0 \colon f(n) \geq cg(n).$$

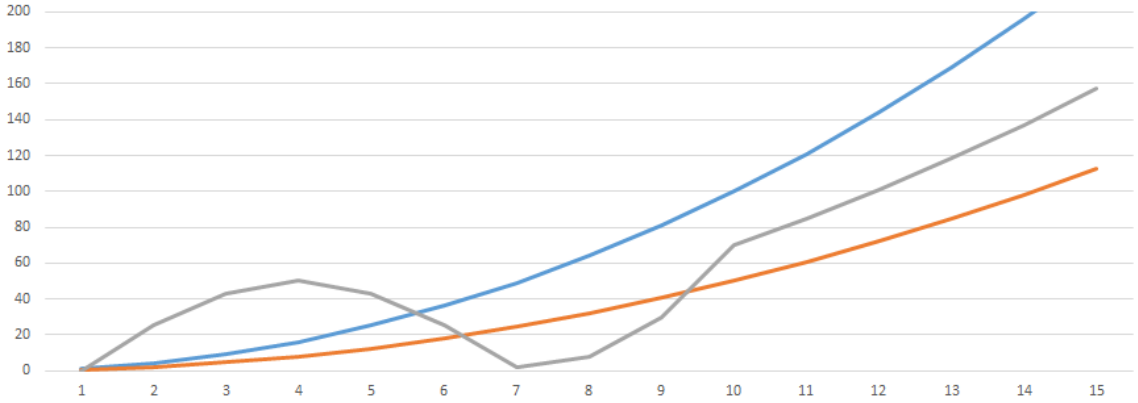
Seekord on funktsiooni f kasv alt tõkestatud funktsiooniga g. Väited $f(n) \in O\big(g(n)\big)$ ja $g(n) \in \Omega\big(f(n)\big)$ on samaväärsed.

$f(n) \in \Theta\big(g(n)\big)$, kui leiduvad positiivsed konstandid c_1, c_2 ja n_0, mille korral kehtib

$$\forall n \geq n_0 \colon c_1 g(n) < f(n)\ \leq c_2 g(n).$$

Sellisel juhul on funktsioon f tõkestatud funktsiooniga g nii ülalt kui alt, st kehtib nii $f(n) \in O(g(n))$ ja $f(n) \in \Omega\big(g(n)\big)$.

Järgneval graafikul on kujutatud sinisega funktsioon $g(n)\ =\ n^2$, oranžiga funktsioon $h(n)\ =\ 0{,}5 * n^2$ ja halliga uuritav funktsioon $f(n)$, mille keerukust proovime hinnata:



Kuigi funktsioon $f(n)$ kasvab väikeste n väärtuste korral kiiremini kui funktsioon $g(n)$ siis on jooniselt näha, et väärtuse $n = 6$ korral jääb $f(n)$ funktsioonist $g(n)$ maha (ja ei möödu sellest kunagi) ehk iga n väärtuse korral $n \geq 6$ kehtib $f(n) < g(n)$. Seega $f(n) \in O\big(g(n)\big)$ ehk $f(n) \in O(n^2)$. Analoogselt alates väärtusest $n = 10$ on funktsiooni $f(n)$ väärtus alati suurem kui funktsiooni $h(n) = 0{,}5 * g(n)$ väärtus, seega $f(n) \in \Omega\big(g(n)\big)$ ehk $f(n) \in\ \Omega(n^2)$. Seega, kui võtta $n_0 = 10$, on täidetud ka tingimus Θ-hinnanguks:

$$\forall n \geq 10 \colon 0{,}5n^2 < f(n)\ \leq n^2\ \rightarrow f(n) \in\ \Theta(n^2).$$

Praktikas kasutatakse sageli O-notatsiooni ehk antakse vaadeldavale algoritmile ülempiiri hinnang, millest rohkem aega ei tohiks selle töö võtta. Isegi juhtudel, kui tegelikult saaks kasutada ka Θ-hinnangut ehk tugevamat väidet, kasutatakse mitteformaalsetes algoritmialastes aruteludes pigem O-d. Seetõttu tuleb erinevate materjalidega töötades olla hoolikas.

3.1.3 Kasvuseoste omadused

Kasvuseostel kehtivad järgmised kasulikud omadused:

1. $\forall c > 0: cf(n) \in \Theta(f(n))$. See tähendab, et funktsiooni korrutamine konstandiga ei muuda selle kasvukiirust. Erijuhul $c = 1$ saame $f(n) \in \Theta(f(n))$, samuti $f(n) \in O\big(f(n)\big)$ ja $f(n) \in \Omega(f(n))$. Kui $f(n) = 1$, siis $c \in \Theta(1)$, mis tähendab, et kõigi konstantsete funktsioonide kasvukiirused võrdsed. See ei tähenda, et kaks programmi töötaksid sama kaua, aga nende tööaeg kasvab samal viisil.
2. Kui $f(n) \in O\big(g(n)\big)$ ja $g(n) \in O\big(h(n)\big)$, siis $f(n) \in O\big(h(n)\big)$. See tähendab, et seos O on transitiivne. Samuti on transitiivsed seosed Ω ja Θ.
3. Kui $f(n) \in \Theta\big(h(n)\big)$ ja $g(n) \in O\big(h(n)\big)$, siis $(f(n) + g(n)) \in \Theta\big(h(n)\big)$. See tähendab, et kahe funktsiooni summa kasvu määrab kiirema kasvuga liidetav. Näiteks kui $f(n) \in \Theta(n^3)$ ja $g(n) \in \Theta(n)$, siis $(f(n) + g(n)) \in \Theta(n^3)$.
4. Kui $f_1(n) \in \Theta\big(g_1(n)\big)$ ja $f_2(n) \in O\big(g_2(n)\big)$, siis $f_1(n)f_2(n) \in O\big(g_1(n)g_2(n)\big)$.
   Kui $f_1(n) \in \Theta\big(g_1(n)\big)$ ja $f_2(n) \in \Omega\big(g_2(n)\big)$, siis $f_1(n)f_2(n) \in \Omega\big(g_1(n)g_2(n)\big)$.
   See tähendab, et kahe funktsiooni korrutise kasv on tegurite kasvude korrutis.
5. Kui $p(n)$ on d-astme polünoom, siis $p(n) \in \Theta(n^d)$.
6. $\forall 0 \leq r \leq s: n^r \in O(n^s)$. See tähendab, et madalama astmega astmefunktsioon ei kasva kiiremini kõrgema astmega astmefunktsioonist. Kehtib ka tugevam väide: madalama astmega funktsioon kasvab alati rangelt aeglasemalt kui kõrgema astmega astmefunktsioon.
7. $\forall k \geq 0, b > 1: n^k \in O(b^n)$. See tähendab, et mitte ükski astmefunktsioon ei kasva kiiremini ühestki eksponentfunktsioonist. Tegelikult kasvab astmefunktsioon alati rangelt aeglasemalt.
8. $\forall k > 0, b > 1: \log_b n \in O(n^k)$. See tähendab, et ükski logaritmfunktsioon ei kasva kiiremini ühestki astmefunktsioonist. Tegelikult kasvab logaritmfunktsioon alati rangelt aeglasemalt.
9. $\forall a > 1, b > 1: \log_a n \in \Theta(\log_b n)$. See tähendab, et kõik logaritmfunktsioonid kasvavad sama kiirusega.
10. $\forall r \geq 0: \sum_{i=1}^{n} i^r \in \Theta(n^{r+1})$. See tähendab, et r. järku astmerea summa kasvab nagu $(r + 1)$. astme polünoom.

3.2 ALGORITMI KEERUKUSE HINDAMINE

Kui algoritm koosneb ainult järjestikustest käskudest, siis kogu algoritmi täitmisaeg on nende järjestikuste käskude täitmisaegade summa. Formaalselt avaldub k järjestikuse käsuga algoritmi täitmisaeg kujul

$$T(n) = f_1(n) + f_2(n) + \cdots + f_k(n),$$

kus $f_i(n)$ on i. käsu täitmisaeg.

3.2.1 Hargnemiste keerukuse hindamine

Kui algoritmis esineb hargnemisi, siis keskmise täitmisaja hindamiseks tuleb iga hargnemise korral arvesse võtta tõenäosust, millega mingit haru täidetakse. Näiteks lihtsa tingimuslause korral kontrollitakse kõigepealt tingimust ja selle tulemusena täidetakse vastavalt kas esimest või teist haru. Sel juhul avaldub täitmisele kuluv keskmine koguaeg järgmiselt:

$$T(n) = f_o(n) + pf_1(n) + (1 - p)f_2(n),$$

kus $f_o(n)$ on tingimuse kontrollimise kulu, $f_1(n)$ ja $f_2(n)$ vastavalt esimese ja teise haru kulu ning p

tõenäosus, et täidetakse esimene haru.
Halvimaks juhuks on see, et täidetakse alati haru, mille täitmine võtab kauem aega. Sel juhul avaldub koguaeg

$$T(n) = f_o(n) + \max\big(f_1(n), f_2(n)\big).$$

3.2.2 Tsüklite keerukuse hindamine

Tsüklitega ehk iteratiivse algoritmi hindamisel tuleb arvestada, et tsükli sisus olevaid käske täidetakse korduvalt. Kui tsüklit täidetakse k korda ja tsükli sisu ühekordse täitmise ajaline keerukus on $f(n)$, siis tsükli täitmise ajaline keerukus on $kf(n)$.

Kui korduste arv k on konstant, siis $kf(n) \in \Theta(f(n))$, ehk algoritmi ajalist keerukust mõjutavad oluliselt ainult tsüklid, mille korduste arv on sõltuvuses andmemahust n. Näiteks kui vaatame tsüklit, mis käib läbi jada pikkusega n ja iga elemendi korral teeb ühe võrdlustehte, mis on konstantse keerukusega, see tähendab $f(n) \in \Theta(1)$, saame tsükli täitmise kuluks $nf(n) \in \Theta(n)$.

3.2.3 Mitmekordsete tsüklite keerukus

Mitmekordsete tsüklite korral saab neid vaadata seestpoolt väljapoole. Kui sisemine tsükkel käib andmed läbi n korda ja on seega keerukusega $t(n) \in \Theta(n)$ ning välimist tsüklit täidetakse samuti n korda, siis välimise tsükli täitmine on keerukusega $nt(n) \in \Theta(n \cdot n) = \Theta(n^2)$.

Päriselt ei koosne kõik tsüklid muidugi täpselt n sammust. Näiteks kui me tahame leida jadast pikkusega n kõikvõimalikud elementide paarid, siis välimine tsükkel käib läbi kõik elemendid, kuid sisemise puhul piisab, kui alustada tsüklit vaadeldavale elemendile järgnevast elemendist. Sellisel juhul käib sisemine tsükkel esimesel korral läbi $n-1$ elementi, teisel $n-2$ jne. Kogukeerukus avaldub siis kujul

$$\begin{aligned} T(n) &= (n-1)f(n) + (n-2)f(n) + \cdots + (n-n)f(n) \\ &= \big((n-1) + (n-2) + \cdots + (n-n)\big)f(n). \end{aligned}$$

Sulgudes on meil tegelikult jada $0, 1 \ldots n-1$. Esitame selle kujul $0 + \sum_{i=1}^{n} i - n$. Kuna $0 \in \Theta(1)$, $\sum_{i=1}^{n} i \in \Theta(n^2)$ (10. omadus) ja $n \in \Theta(n)$. Kui $f(n) \in \Theta(1)$, saame

$$(\Theta(1) + \Theta(n^2) - \Theta(n))\Theta(1) = \Theta(n^2).$$

Kui täpsed valemid pole meeles või ei anna ühtki neist konkreetsel konstruktsioonil kasutada, siis tuleb appi O-hinnang. Nagu eelnevalt mainitud, võib O-hinnanguid anda liiaga. Antud näites

$$\begin{aligned} T(n) &= (n-1)f(n) + (n-2)f(n) + \cdots + (n-n)f(n) \leq \\ &\leq nf(n) + nf(n) + \cdots + nf(n) = n^2 f(n). \end{aligned}$$

Kuna $n^2 f(n) \in \Theta(n^2)$, siis $T(n) \in O(n^2)$.

Rusikareeglina saabki kasutada O-hinnangut, et konstantse sisuga ühekordne tsükkel on keerukusega $O(n)$, kahekordne $O(n^2)$, kolmekordne $O(n^3)$ jne. See on mõistlik küll vaid juhul, kui igas tsüklis tsüklimuutujat muudetakse lineaarselt, see tähendab, et tsükli korduste arv on $k * n + c$, kus k ja c on konstandid. Nt tsüklid päistega

```
for (int i=0; i<2*n; i++)
for (int i=100; i<n+10; i++)
for (int i=0; i<n; i+=2)
```

on kõik keerukusega $O(n)$, kuigi esimene teeb $2n$, teine $n-90$ ja viimane $n/2$ kordust.

Samas tsükkel päisega `for (int i = 1; i <= n; i*= 2)` on hoopiski $O(logn)$ ($\log_2 n$ `kordust`) ning tsükkel päisega `for (int i=0; i<(n-2)*n; i++)` on ruutkeerukusega $O(n^2)$.

3.2.4 Rekursiooni keerukuse hindamine

Rekursiivse algoritmi täitmisele kuluvat aega hinnatakse rekurrentse võrrandiga. Näiteks „jaga ja valitse“ tüüpi ülesannetel, kus ülesanne jagatakse kaheks võrdseks (andmemahuga $n/2$) alamülesandeks ja mõlemad lahendatakse samal meetodil, siis juhul, kui alamülesannete lahenduste töötlemiseks kulub veel n sammu, avaldub kogu sammude arv järgmisel kujul:

$$T(n) = \begin{cases} 0, kui\ n = 1 \\ 2T\left(\frac{n}{2}\right) + n, kui\ n > 1 \end{cases}$$

Pealtnäha pole sellisest võrrandist suurt abi, kuna selle lahendamine pole just kõige lihtsam, kuid paljudel juhtudel saab sellisel kujul avaldatud lahendusaega hinnata järgmise seose abil:
Kui $T(n) = aT(n/b) + f(n),\ kus\ a \geq 1\ ja\ b > 1$, siis

$$T(n) = \begin{cases} \Theta\big(f(n)\big), f(n) \in \Omega\big(n^{\log_b a + e}\big)\ ja\ af\left(\frac{n}{b}\right) \leq cf(n), \text{mingi konstandi } c \leq 1 \text{ ja suure n korral} \\ \Theta\big(n^{\log_b a} \cdot \log n\big), \quad f(n) \in \Theta(n^{\log_b a}) \\ \Theta\big(n^{\log_b a}\big), \quad f(n) \in O\big(n^{\log_b a - e}\big) \text{ ja } e > 0 \end{cases}$$

Ülalpool toodud näite korral $a = b = 2$, $\log_b a = 1$ ja $f(n) \in \Theta(n) = \Theta(n^{log_b a})$. Nüüd saame teise rea põhjal, et $T(n) \in \Theta(n \log n)$.

Sagedamini esinevate rekursiivsete algoritmide võrrandid on toodud järgnevas tabelis:

T(n)	Keerukus	Näide algoritmist
$\boldsymbol{T(n/a) + \Theta(1)}$	$\Theta(\log n)$	Kahendotsing jadast (a=2)
$\boldsymbol{T(n-1) + \Theta(1)}$	$\Theta(n)$	Faktoriaali leidmine rekursiivselt
$\boldsymbol{bT(n/b) + \Theta(1)}$	$\Theta(n)$	“Jaga ja valitse” ilma valitsemiseta
$\boldsymbol{cT(n/c) + \Theta(n)}$	$\Theta(n \log n)$	“Jaga ja valitse”, nt põimemeetod
$\boldsymbol{T(n-1) + \Theta(n)}$	$\Theta(n^2)$	Kiirmeetodil sorteerimise halvim juht
$\boldsymbol{dT(n-1) + \Theta(1)}$	$\Theta(d^n)$	Hanoi tornid (d=2)

3.3 TAVALISED KEERUKUSKLASSID

Vaatame siin algoritmide tavalisemaid keerukusklasse ja nende esinemise juhtusid.

$O(1)$ - **konstantse** keerukusega on sellised ülesanded, mille lahendamise aeg ei sõltu oluliselt sisendi suurusest.

$O(log\ n)$ – **logaritmilise** keerukusega on harilikult algoritmid, mis jagavad sisendi igal sammul enam-vähem võrdseteks osadeks. Näiteks kahendotsingu algoritm on logaritmilise keerukusega.

$O(n)$ – **lineaarse** keerukusega ülesanded on enamasti sellised, kus sisend käiakse läbi konstantne arv kordi. Lineaarse keerukusega on näiteks suvalise jada kõikide elementide summa leidmine või mingi kindla väärtusega elemendi leidmine sorteerimata jadast.

$O(n\ log\ n)$ - Sellise keerukusega on tavaliselt parimad sortimisalgoritmid ja seega ei saa madalamasse keerukusklassi kuuluda ükski algoritm, mis kasutab võrdlustel põhinevat sortimist.

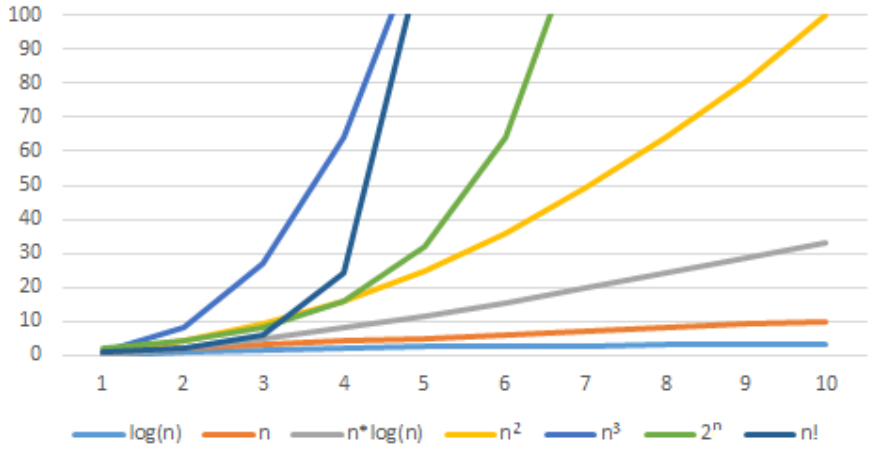
$O(n^2)$ – **ruutkeerukusega** algoritmid on tavaliselt kahekordse tsükliga algoritmid – niimoodi on näiteks võimalik läbi vaadata sisendi elementide kõikvõimalikud paarid.

$O(n^3)$ – **kuupkeerukusega** algoritmid on harilikult kolmekordse tsükliga algoritmid.

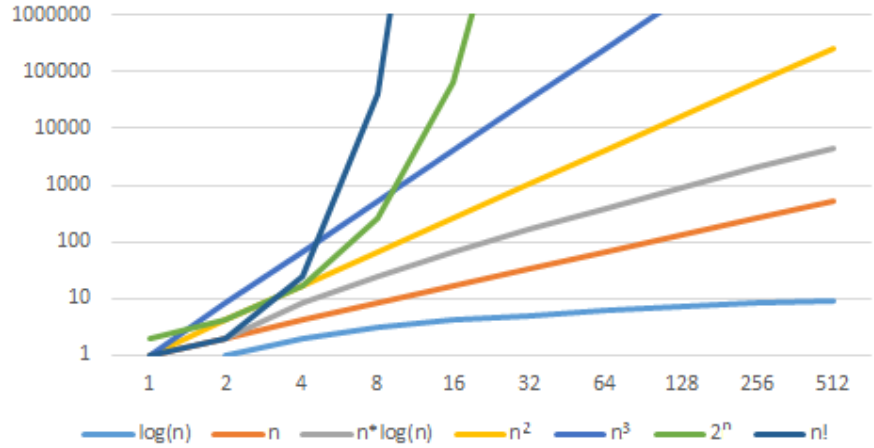
$O(2^n)$ – **eksponentsiaalse** keerukusega algoritmid käivad tavaliselt läbi sisendi kõikvõimalikud alamhulgad.

$O(n!)$ – **faktoriaalse** keerukusega algoritmid enamasti vaatavad läbi kõik sisendi permutatsioonid.

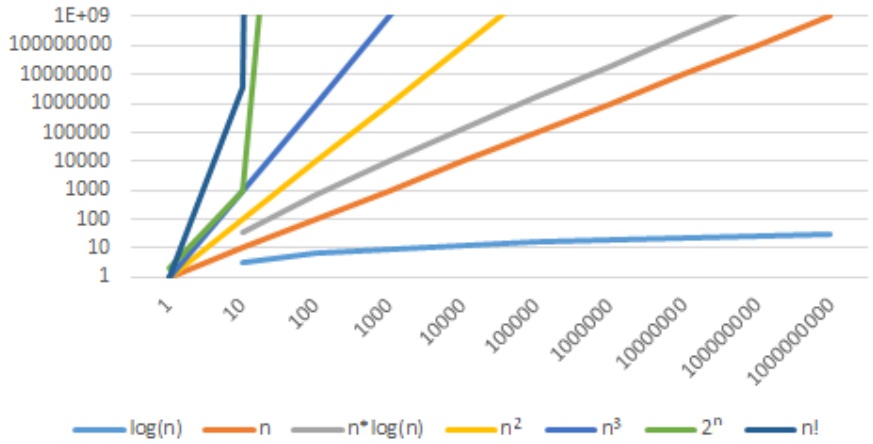
Nii näevad tavaliste keerukusklasside funktsioonid välja graafikul:



Järgmisena on esitatud samad funktsioonid logaritmilisel skaalal:



Veel üks graafik praktikas enam kasutatavate väärtustega:



3.3.1 Keerukusklass ja ajalimiit

Võistlustel on tavaliselt ette antud ajalimiit ning sisendandmete lubatud maht. Selle põhjal saab ennustada, millise keerukusklassiga algoritm lahenduseks sobib. Järgmises tabelis on toodud sisendi suurus ja algoritmi keerukusklass, mis sellise sisendi suudab töödelda ligikaudu sekundiga, kui protsessor suudab sooritada 100 miljonit operatsiooni sekundis:

n	Keerukusklass	Näide
10 – 11	$O(n!), O(n^6)$	Kõigi permutatsioonide leidmine
15 – 18	$O(2^n * n^2)$	Rändkaupmehe ülesanne dünaamilise planeerimisega
18 – 22	$O(2^n * n)$	Dünaamiline planeerimine bitimaskiga
100	$O(n^4)$	
400	$O(n^3)$	Floyd-Warshalli algoritm
2000	$O(n^2 \log n)$	
10000	$O(n^2)$	Aeglased sorteerimisalgoritmid
1000000	$O(n \log n)$	Kiiremad sorteerimisalgoritmid, mitmed “jaga ja valitse” tüüpi ülesanded.
100 000 000*	$O(n), O(\log n), O(1)$	

* enamasti jäävad võistlustel andmemahud miljoni piiresse, kuna suuremate sisendite töötlus võtab liiga palju aega.

3.4 ANDMESTRUKTUURID

Iga mittetriviaalne programm kasutab oma töös küllaltki suurt hulka andmeid. Ilmselgelt pole võimalik iga väärtuse jaoks luua eraldi muutujat, vaid andmed tuleb grupeerida ja korrastada. Juba enne arvutite leiutamist mõtlesid inimesed andmete korrastamiseks välja näiteks loendid, tabelid ja sõnastikud. Sarnased abistavad struktuurid on ehitatud ka enamikus programmeerimiskeeltes. Andmete korrastamine täidab kaht peamist eesmärki:

- Programmi on võimalik lihtsama vaevaga lugeda, mõista ja parandada.
- Õigesti valitud struktuuri korral on andmeid on võimalik kiiresti kätte saada ja muuta.

Andmestruktuur on vahend suure hulga samatüübiliste andmete hoidmiseks ja neile efektiivse juurdepääsu ja töötlemise korraldamiseks.

Andmestruktuurid otseselt ülesandeid ei lahenda, kuid erinevatel struktuuridel töötavad erinevad algoritmid ning seega aitab õige andmestruktuuri valik kaasa efektiivse lahenduse leidmisele ja kasutusele.

3.4.1 Andmestruktuuride klassifitseerimine

Andmestruktuure võib jagada **konkreetseteks** ja **abstraktseteks**. Konkreetsed andmestruktuurid kirjeldavad andmete tegelikku paigutust arvutis, sellised on näiteks massiiv ja ahel. Abstraktsed andmestruktuurid kirjeldavad pigem võimalusi, mida selle andmestruktuuriga teha saab, samas võib andmeid nende sees esitada mitmel viisil. Abstraktsed andmestruktuurid on näiteks pinu ja järjekord.

Sageli võib abstraktse andmestruktuuri luua nii, et ta kasutab sisemiselt mõnd lihtsamat konkreetset andmestruktuuri.

Andmestruktuurid võivad olla kas **staatilised** (s.t nende suurus pannakse alguses paika ja neisse ei saa uusi elemente lisada või neid eemaldada) või **dünaamilised** (saab elemente lisada ja eemaldada).

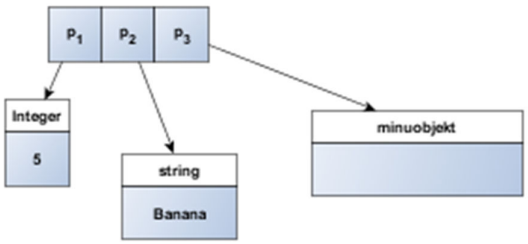
3.5 MASSIIV

Massiiv (*array*) on andmestruktuur, mis koosneb indekseeritud ühetüübilistest elementidest. See tähendab, et igal elemendil massiivis on kindel järjenumber ehk indeks. Elementide arvu massiivis nimetatakse massiivi pikkuseks. Lühidalt oli massiividest juttu juba esimeses peatükis, kuna enamasti on programmeerimiskeeltes massiivi jaoks olemas lausa omaette andmetüüp.

Esimeses peatükis oli ka juttu, kuidas massiive mälus hoitakse. Harilikult antakse massiivi deklareerimisel ette selle pikkus ja elementide andmetüüp. Selle info põhjal reserveeritakse massiivi jaoks vajaliku pikkusega mäluplokk. Teades esiteks seda, millisel mäluaadressil massiiv algab, ja teiseks elemendi andmetüübi pikkust, on lihtne arvutada, kust algab 2. element, kust 3. jne. Tänu sellele on indeksile vastava elemendi leidmine massiivist kiire ja lihtne operatsioon. Asjaolu, et terve massiiv hoitakse mälus koos, teeb massiivi järjestikuse elementhaaval läbikäimise samuti väga kiireks, kuna juba esimese elemendi lugemisel lisatakse vahetult järgnevad elemendid suure tõenäosusega protsessori vahemälusse.

3.5.1 Massiivid Pythonis

Pythonis kasutatakse massiivi asemel listi mõistet ning selle ülesehitus on veidi teistsugune. Nagu juba esimeses peatükis selgus, on Pythoni list sisuliselt viitade massiiv. Seetõttu võimaldab Python erinevalt tavapärasest massiivist hoida listis erinevat tüüpi elemente – see, kui palju mingi element mälus ruumi võtab, ei ole sellise ülesehituse seisukohast oluline:



3.5.2 Massiivi võimalused ja piirangud

Massiivi elementide väärtust saab lihtsasti muuta tavalise omistamistehtega, kuid massiivi elemente lisada ja neid sealt eemaldada üldjuhul ei saa. Kuna aga elementide väärtusi saab muuta, siis saab simuleerida ka elementide lisamist ja eemaldamist. Selleks võib programmeerija deklareerida piisavalt suure pikkusega massiivi ja pidada ise järge, kui mitu elementi tal tegelikult parasjagu massiivis on. Kui parasjagu on kasutusel m elementi, siis uue elemendi lisamine on selle väärtuse salvestamine kohale a[m] (juhul, kui indekseerimine algab 0-st, nagu siin raamatus käsitletud keeltes on). Sellise lisamise töömaht ei sõltu massiivi pikkusest ega elementide arvust selles ja on seega konstantse keerukusega. Kui aga sooviks on lisada element massiivi keskele kohale k, tuleb elemendid kohtadel $k \ldots m-1$ tõsta ühe koha võrra edasi. Selle keerukus on juba $O(n)$. Juhul, kui ei ole vaja säilitada elementide omavahelist järjekorda, võib muidugi salvestada a[k] asuva elemendi väärtuse massiivis viimaseks ja salvestada kohale a[k] uus, soovitud väärtus. Sama on eemaldamisega: kui soovitakse eemaldada element kusagilt kasutuses oleva osa keskelt kohalt k, siis tuleb kõik elemendid kohtadel $k+1..m-1$ tõsta üks koht ettepoole. Kiiresti saab eemaldada ainult viimast elementi. Kui elementide järjekorra säilitamine ei ole oluline, saab tegelikult salvestada eemaldatava väärtuse kohale a[k] viimase väärtuse a[m-1] ning tagada sellega alati konstantse keerukuse ka eemaldamisel. Sageli siiski on järjekorra säilitamine oluline.

3.5.3 Mitmemõõtmelised massiivid

Massiivid võivad olla ka mitmemõõtmelised. Mitmemõõtmelisi massiive on kõige lihtsam ette kujutada tabelina ja tavaliselt räägitaksegi kahemõõtmeliste massiivide puhul ridadest ja veergudest.

Joonisel on kujutatud kahemõõtmeline massiiv A elementide a_{11}-a_{34} hoidmiseks.

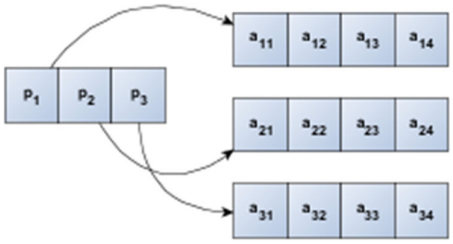
a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}

C/C++ keeles ei erine mitmemõõtmeline massiiv mälus tegelikult ühemõõtmelisest: elemendid on kas ridade või veergude kaupa järjest:

	a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	

Ka nii on elemendi leidmine lihtne, kuna andes ette rea ja veeru indeksi, saab kiiresti arvutada elemendi tegeliku asukoha ehk kauguse massiivi algusest. Kui indekseerimist alustada nullist, siis kaugus avaldub valemiga $k = (r-1) \cdot vn + (v-1)$, kus r on rea number, v on veeru number ja vn veergude arv. Kui hakata ridu ja veerge ka 0-st loendama, nagu programmeerimises tavaks, on valemi kuju veelgi lihtsam: $k = r \cdot vn + v$. Sarnaselt saab arvutada ka suuremamõõtmeliste massiivide elementide kaugusi massiivi algusest, näiteks kolmemõõtmelise massiivi puhul $k = i_3 + N_3 \cdot (i_2 + N_2 \cdot n_1)$, kus i tähistab vastava mõõtme indeksit ja N elementide arvu vastava mõõtme suunal.

Java mitmemõõtmelised massiivid on üles ehitatud veidi teistsugusel põhimõttel: need on pigem massiivid massiividest, mis siis omakorda võivad koosneda massiividest jne.



Selline lähenemine võimaldab luua „hambulisi“ (*jagged*) massiive, kus read ei ole võrdse pikkusega. Sarnane põhimõte on ka Pythonis: listi elemendiks võib olla teine list jne. Loomulikult on võimalik luua ka C++ massiive viitadest (pointeritest), mis viitavad massiividele.

3.5.4 Dünaamilised massiivid

Massiivi pikkus on tavaliselt fikseeritud ja programmi käitusajal seda muuta ei saa. See tekitab mitmeid ebameeldivusi, mistõttu on kasutusele võetud dünaamilised massiivid, mille pikkust saab muuta. Esialgu eraldatakse dünaamilisele massiivile mingi pikkusega osa mälust – olgu see n. Kuni meil ongi vaid kuni n elementi, töötab kõik nagu tavalises massiivis. Kui on vaja lisada $n + 1$. element, siis

1. eraldatakse pikem mäluosa, harilikult pikkusega $2n$,
2. kopeeritakse elemendid esialgsest massiivist uude massiivi,
3. esialgsele massiivile kuulunud mäluosa vabastatakse.

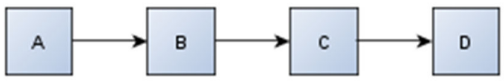
Muidugi võtab elementide kopeerimine uude massiivi aega, kuid suurendades iga kord vajadusel massiivi pikkust 2 korda, on see küllaltki harv olukord ning keskmine elemendi lisamise keerukus massiivi lõppu on enamikel juhtudel konstantne. Teisalt jälle võtab dünaamiline massiiv enamasti mälus rohkem ruumi, kui seal olevate elementide hoidmiseks tegelikult vaja on.

Pythoni `list` ongi dünaamiline massiiv. C++ standardteegis realiseerib dünaamilise massiivi klass `vector` ja Javas `ArrayList`. Tavaliselt on sellistel klassidel meetodid elemendi massiivi lõppu lisamiseks ja lõpust eemaldamiseks, mis on enamasti konstantse keerukusega. Samuti on realiseeritud elemendi lisamine ja eemaldamine suvaliselt positsioonilt, kuid kuna sel juhul tõstetakse tagapool asuvad elemendid ümber, on need operatsioonid lineaarse keerukusega.

Massiive kasutatakse väga palju nii iseseisva andmestruktuurina kui ka teiste andmestruktuuride, sealhulgas pinude, järjekordade ja kuhjade realiseerimiseks.

3.6 AHEL

Massiiviga seotud lisamise ja eemaldamise probleemidest on vaba **ahel**. Ahel on andmestruktuur, kus iga element alates esimesest teab, kus asub talle järgnev element:



Elementide arv ahelas ei ole fikseeritud, neid saab vabalt lisada ja eemaldada konstantse ajaga:

Samas, kuna elemendi asukoht ei ole kindlalt määratud, siis elemendi lugemine ahelast on keerukusega $O(n)$ – iga kord tuleb alustada esimesest elemendist ja käia ahelat läbi kuni soovitud elemendini. See kehtib ka lisamis- ja eemaldamiskoha otsimise jaoks.

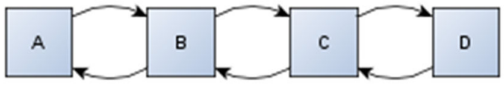
Ahelal on massiivi ees järgmised eelised:

- dünaamilisus – ahel kasvab ja kahaneb vastavalt vajadusele loomulikult;
- lisamis- ja eemaldamisoperatsioonide lihtsus ja kiirus;
- puudub vajadus määrata algsuurust;

Ahelal on ka negatiivsed küljed:

- Suurem mäluvajadus ühe elemendi hoidmiseks, kuna lisaks elemendi väärtusele tuleb meeles pidada ka viit järgmisele elemendile;
- Elemendid ei ole kergesti leitavad, lugemiseks tuleb alustada alati algusest;
- Kuna elemendid võivad paikneda mälus juhuslikult, siis mälust lugemine võib võtta kauem aega, sest ahelas lähestikku paiknevad elemendid ei pruugi olla mälus lähestikku ega jõua protsessori vahemällu.
- Ahelas tagurpidi liikumine on väga kohmakas –iga kord tuleb alustada algusest.

Viimase probleemi lahenduseks kasutatakse topeltseotud ahelaid, kus lisaks järgnevale elemendile teab iga element ka talle eelnevat elementi. See aga nõuab iga elemendi kohta juba kahe viida meelespidamist.



Kuigi ahela näol on tegemist väga tuntud andmestruktuuriga, mida tutvustavad enam-vähem kõik programmeerimisõpikud, tuleb võistlusülesannete lahendamisel tegelikult harva ette ülesandeid, kus on praktiline kasutada ahelat. Eelised dünaamilise massiivi ees tulevad praktikas küllaltki harva välja. Protsessori vahemälu kasvamisega on tegeliku töökiiruse seisukohalt järjest olulisemaks saanud sarnaste andmete füüsiline lähedus mälus.

C++ keeles realiseerib ahelat klass `foward_list` (päisfailis `<forward_list>`) ja topeltseotud ahelat klass `list` (päisfailis `<list>`) . Javas on topeltseotud ahela jaoks klass `LinkedList` (`java.util*`). Pythoni standardisse ahel ei kuulugi.

Järgmises tabelis on toodud dünaamilise massiivi ja ahela tavaliste operatsioonide jaoks kuluv sammude arv:

Operatsioon elemendiga	Dünaamiline massiiv	Topeltseotud ahel
Lugemine suvalisest kohast	1	n/4 keskmiselt, otstest 1
Lõppu lisamine	1, halvimal juhul n	1
Lisamine kohale i	n/2 keskmiselt	n/4 keskmiselt, algusesse lisamine 1
Eemaldamine kohalt i	n/2 keskmiselt	n/4 keskmiselt, otstest 1
Eemaldamine käesolevast kohast	n/2 keskmiselt	1
Lisamine käesolevasse kohta	n/2 keskmiselt	1

3.7 PINU

Pinu (*stack*) on abstraktne andmestruktuur, kus elementidele pääseb juurde kindlas järjekorras – elemendi lisamisel paigutatakse see kõige viimaseks ja elemendi eemaldamisel võetakse ära kõige viimasena lisatud element. Sellise juurdepääsusüsteemi kohta kasutatakse sageli inglisekeelset lühendit LIFO (*last in, first out* – viimasena sisse, esimesena välja).

Pinul on kaks olulist operatsiooni: elemendi lisamine ja elemendi lugemine ja eemaldamine. Suvalisest kohast elementi lugeda/eemaldada ei saa, selleks tuleb eemaldada kõik „pealpool“ olevad elemendid. Samuti ei saa elementi lisada suvalisse kohta pinus – jällegi, selleks tuleks eemaldada „pealpool“ olevad elemendid ja need pärast lisamist tagasi panna.

Pinuga tutvusime põgusalt juba eelmises peatükis alamprogrammide teema juures. Seal selgus, et alamprogrammide väljakutseid hoitakse pinumälus. Pinu põhimõttel töötab ka veebilehitsejates nupp „Tagasi“ – vaadatud leheküljed pannakse järjest pinusse ja „Tagasi“ nupu vajutusel võetakse sealt viimati lisatud lehekülg. Samuti töötab ka tekstiredakorites tagasivõtt (*undo*).

Pinu kasutatakse ka mitmest tehtest koosnevate arvutustehete vastuse leidmiseks. Lihtsam on selleks arvutustehe esitada postfiks-kujul ehk pööratud Poola kujul. See tähendab, et tehtemärk kirjutatakse operandide järele, mitte vahele, nagu tavakasutuses (infiks-kujul). Nii 3 + 4 on postfiks-kujul 3 4 + ja (3 + 4)*5 on 3 4 + 5 *. Postfiks-kuju eeliseks on see, et sulge ei ole vaja kasutada. Postfiks-kujul arvutamine pinu abil toimub järgmiselt:

1. Loe avaldisest järgmine operand või operaator.
2. Kui järgmist operandi/operaatorit pole, loe vastus pinust ja lõpeta töö.
3. Kui on operand (arv), pane see pinusse.
4. Kui on operaator (tehtemärk) loe pinust kaks operandi, soorita tehe ja pane vastus pinusse.
5. Korda esimesest punktits alates.

Pinu realiseeritakse enamasti (dünaamilise) massiivina. Massiivi korral tuleb valida piisavalt suur massiiv ning hoida meeles elementide arvu pinus. Tähistades massiivi A-ga ja pinus olevate elementide arvu n-ga, siis lisamisel suurendatakse n-i ühe võrra ja lisatakse uus väärtus kohale a_n. Eemaldamisel tagastatakse kohal $a_n$olev väärtus ja vähendatakse n-i ühe võrra.

Pinu on võimalik realiseerida ka ahelana. Sellisel juhul lisatakse uus element alati ahela algusesse ning eemaldamise korral tagastatakse ja eemaldatakse esimene element ahelast.

Mõlemal juhul on eemaldamise ja lisamise keerukus $O(1)$.

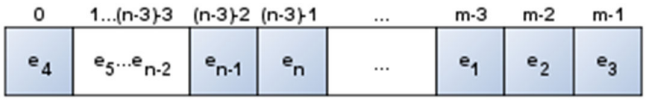
Ajalooliselt on pinu kohta eesti keeles kasutatud ka sõna „magasin“. Folkloori kohaselt kasutati Tartu Ülikoolis üht terminit ja Tallinnas teist, aga tänapäeval on „pinu“ rohkem levinud. Kõige paremini saab magasini mõiste selgeks aga siis, kui sul on kell 9 algoritmide ja andmestruktuuride eksam, aga ärkad kell 9:20. Siis lööd käega vastu otsmikku: „Oh, magasin!“

3.8 JÄRJEKORD

Ka järjekord on selline andmestruktuur, kus elementidele pääseb ligi kindlas järjekorras. Järjekorra põhimõte on nagu tavalises poesabas: uus element lisatakse alati lõppu ja järgmine element loetakse ja eemaldatakse alati algusest. Sellist põhimõtet tähistab lühend FIFO (*first in, first out* – esimesena sisse, esimesena välja).

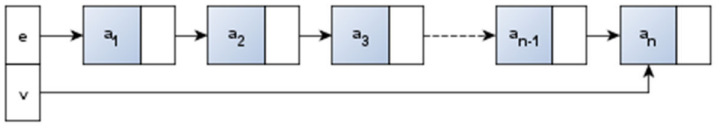
Sarnaselt pinuga on järjekorral operatsioonid elemendi lisamiseks ja eemaldamiseks (koos lugemisega).

Järjekord realiseeritakse tavaliselt samuti massiivina või ahelana, kuid mõlemad on veidi keerulisemad kui pinu korral. Massiivi kasutades on probleemiks see, et järjekorra korral ei püsi kumbki ots muutumatuna. Esimeseks võimaluseks on muidugi kas lisamisel või eemaldamisel nihutada kõiki järjekorras olevaid elemente massiivis, kuid sellisel juhul on lisamise või eemaldamise protseduuri keerukuseks juba $O(n)$. Teiseks võimaluseks on meeles pidada lisaks elementide arvule ka esimese elemendi indeks, mis siis elementi eemaldades nihkub ühe võrra edasi. Sellisel juhul aga on vaja päris korraliku pikkusvaruga massiivi ning ületäitumise oht on väga suur. Selle lahendusena käiakse massiivis ringiratast – massiivi viimasele elemendile järgneb järjekorras jälle esimene. Kuna on teada, et algus on alati lõpust eespool, siis selline lahendus toimib päris hästi.



Järjekord massiivina. m - massiivi pikkus, n - elementide arv järjekorras, e_1…e_n – elemendid järjekorras.

Vajadus elementidele mõlemast otsast ligi pääseda, teeb ka lihtahela kasutamise järjekorra jaoks mõnevõrra keerulisemaks kui pinu realiseerimisel. Topeltseotud ahelat siiski vaja ei ole, piisab, kui peame eraldi meeles viita viimasele elemendile – sel moel saab lisada uut elementi kiiresti järjekorra lõppu. Tähelepanelik tuleb aga olla olukorras, kus järjekorras ongi vaid üks element – sellisel juhul on see korraga nii esimene kui ka viimane ning selle eemaldamisel tuleb nullida ahela viit nii esimesele kui ka viimasele elemendile.



Järjekord lihtahelana. e - viit esimesele elemendile, v – viit viimasele elemendile, $a_1 \ldots a_n$ – elemendid järjekorras.

Ka järjekorras on mõlemal juhul eemaldamise ja lisamise keerukus $O(1)$.

3.8.1 Kaheotsaline järjekord

Kaheotsaline järjekord (*double ended queue, deque*) on selline andmestruktuur, kus elemendi lisamine ja võtmine on lubatud mõlemast otsast. Praktikas on see andmestruktuur huvitav sellepärast, et seda saab omakorda kasutada nii järjekorra kui ka pinu realiseerimiseks.

3.8.2 Eelistusjärjekord

Eelistusjärjekord (*priority queue*) on selline järjekord, kus igal elemendil on prioriteet ning elemente võetakse järjekorrast vastavalt elemendi prioriteedile. Päris elus kasutatakse eelistusjärjekorda näiteks erakorralise meditsiini osakonnas (EMO) - kõigepealt hinnatakse patsiendi seisundi tõsidust ja kõrgema prioriteediga patsiente teenindatakse enne. Veelgi keerulisemaks teeb olukorra see, et mõnel juhul – nagu ka EMOs – võib elemendi prioriteet muutuda ning siis on vaja seda elementi järjekorras „nihutada“. Seetõttu peab eelistusjärjekorra realiseerimisel arvestama sellega, et elemendi võtmisel eelistusjärjekorrast tuleb otsida pidevalt, milline on suurima prioriteediga element, või tuleb hoida elemente pidevalt prioriteedi järgi sorteerituna, mis teeb jällegi lisamise ja prioriteedi muutmise keerulisemaks. Seetõttu on eelistusjärjekorra realiseerimiseks kõige parem andmestruktuur hoopiski kuhi, millest tuleb juttu alampeatükis 3.9.

3.9 PINU JA JÄRJEKORRA KASUTAMINE

Enamasti on keelte standardteekides pinu juba valmiskujul olemas. C++ on päisfailis `<stack>` klass `stack`, Javas on parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Deque`. Pythonis eraldi klassi pinu jaoks ei ole, kuid Pythoni `list`il on meetodid `append()` järjendi lõppu elemendi lisamiseks ja `pop()` elemendi eemaldamiseks lõpust.

Järgmine tabel illustreerib pinu kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Pinu p loomine (täisarvude jaoks)	stack<int> p	Deque<Integer> p = new ArrayDeque<Integer> ()	p = []
Elemendi 'a' lisamine	p.push(a)	p.push(a)	p.append(a)
Elemendi eemaldamine	p.pop()	a = p.pop()	a = p.pop()
Pealmise elemendi vaatamine	a = p.top()	a = p.peek()	a= p[-1]
Kas pinu on tühi?	p.empty()	p.isEmpty()	not p

Järjekorra leiab C++ keeles päisfailist `<queue>` (klass queue); Javas on taas üldjuhul parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Queue`. Pythonis saab kasutada struktuuri `collections.deque`.

Järgmine tabel illustreerib järjekorra kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine (täisarvude jaoks)	queue<int> s	Queue<Integer> s = new ArrayDeque<Integer> (); Queue<Integer> s = new LinkedList<Integer> ();	s = deque([])
Elemendi 'a' lisamine	s.push(a)	s.add(a)	s.append(a)
Elemendi eemaldamine	s.pop()	a = s.remove(); a = s.poll()	a = s.popleft()
Esimese elemendi vaatamine	a = s.front()	a = s.element(); a = s.peek()	a= s[0]
Kas järjekord on tühi?	s.empty()	s.isEmpty()	not s

Vaatame vahepeal ühe ülesande näitel, kuidas neid andmestruktuure kasutada:

Ühes pisikeses linnakeses elavad lemmingud.
Nende linnas käib ringiratast trammiliin, millel on n peatust.
Trammil on ainult üks uks ja tramm ise on kitsas.
Seetõttu sisenevad lemmingud trammi ükshaaval, nii et esimesena sisenenud lemming liigub trammi tahaosse, tema järel sisenenud jääb tema ette jne.
Väljumine trammist toimub sisenemisele vastupidises järjekorras: kõige viimasena sisenenud lemming väljub esimesena jne.
Igas peatuses on platvorm pikkusega p, millele mahub ootama täpselt p lemmingut. Platvormilt trammi sisenevad lemmingud täpselt ootejärjekorras: kes oli järjekorras enne, siseneb enne.
Liiklemine on korraldatud järgmiselt:
1. Tramm peatub peatuses platvormi lõpus ning lemmingud väljuvad järjest trammist.
2. Kui järgmisena väljuv lemming on jõudnud oma soovitud peatusesse, siis ta lahkub, kui mitte, siis läheb ta platvormil olevasse ootejärjekorda trammile sisenemist ootama.
3. Kui järgmine väljumist ootav lemming ei ole oma peatuses ning platvormil enam ruumi ei ole, siis rohkem lemminguid selles peatuses ei välju, isegi, kui see on nende sihtpeatus.
4. Keegi kellestki mööda ei trügi.
5. Lõpuks sõidab tramm platvormi etteotsa ning võtab järjekorrast peale nii palju lemminguid, kui trammi mahub.
Seejärel sõidab tramm järgmisesse peatusesse ning seal kordub sama protseduur.
Õhtul lahkuvad kõik lemmingid töölt samal ajal ja moodustavad peatustesse järjekorrad. Leia, kui palju aega kulub trammil kõigi lemmingute kodupeatustesse toimetamiseks aega, kui iga sisenemine ja väljumine võtab 1 sekundi, platvormi ühest otsast teise sõit võtab trammil 5 sekundit ja peatuste-vaheline sõit kestab täpselt 1 minuti. Tramm alustab oma teekonda esimesest peatusest.
Sisendi esimesel real on kolm arvu: peatuste arv $n (2 \le n \le 100)$, kohtade arv trammis $k (1 \le k \le 100)$, platvormi pikkus $p (1 \le p \le 100)$. Peatuste platvormid on kõik sama pikkusega.
Järgneval n real on iga peatuse kohta `1...n` seal ootavate lemmingute arv $l (0 \le l \le p)$ ja selle järel iga lemmingu l_i $(0 \le l_i \le l)$ kohta peatuse number n_j $(1 \le n_j \le n)$, kuhu ta sõita soovib. Järjekorras esimene lemmingu peatus on antud esimesena, teise peatus teisena jne.

NÄIDE:

```
5 2 3
3 4 5 2
2 1 3
0
3 3 4 1
1 3
```

Vastus:

```
1220
```

Kui nüüd pinu ja järjekorra mõisted värskelt meeles on, siis on kerge taibata, et trammi sisenemine ja väljumine käib pinu põhimõttel ning peatustes olevad trammisabad on järjekorrad. Muud polegi vaja teha, kui sisendandmed sisse lugeda ja seejärel kogu situatsioon „läbi mängida.“ Iga peatuse jaoks teeme eraldi järjekorra, mida hoiame ühes järjekordade massiivis.

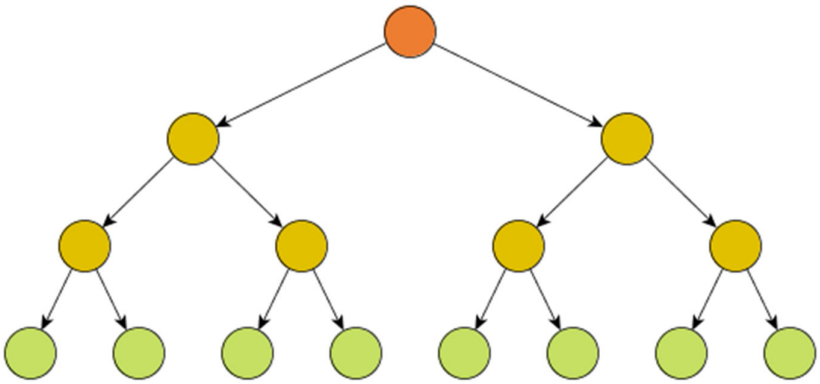
```cpp
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main() {
    int peatuste_arv, kohtade_arv, jk_pikkus;

    //pinu trammis olevate reisijate hoidmiseks, viimasena sisenenu väljub esimesena
    stack<int> tramm;
    //Massiiv trammisabadest. Iga saba on järjekord, kes enne sabas, läheb enne peale
    queue<int> trammisabad[100];

    cin >> peatuste_arv >> kohtade_arv >> jk_pikkus;

    for (int i = 0; i < peatuste_arv; i++) { // iga peatuse kohta
        int ootajate_arv;
        cin >> ootajate_arv; // loeme ootajate arvu peatuses
        for (int j = 0; j < ootajate_arv; j++) { // ja iga ootaja kohta
            int sihtkoht;
            cin >> sihtkoht; // loeme tema sihtkoha
            trammisabad[i].push(sihtkoht - 1); // ning paneme selles peatuses sabasse
        }
    }

    int peatus = 0, kulunud_aeg = 0; // alustame esimesest peatusest
    while (true) {
        // mahalaadimine
        while (!tramm.empty() && // kuni trammis on reisijaid ja
            ((trammisabad[peatus].size() < jk_pikkus) || // peatuses on ruumi või
                tramm.top() == peatus)) { // on järgmine mahamineja jõudnud kohale
            if (tramm.top() != peatus) { // kui järgmine väljuja ei ole oma peatuses
                trammisabad[peatus].push(tramm.top()); // läheb ta selle peatuse sappa
            }
            tramm.pop(); // eemaldame väljuja trammist
            kulunud_aeg++; // lisame väljumiseks kulunud aja
        }

        bool valmis = tramm.empty(); // kas tramm on tühi?
        for (int i = 0; i < peatuste_arv; i++) {
            valmis &= trammisabad[i].empty(); // kas kõigi peatuste sabad on tühjad?
        }
        if (valmis)
            break; // kõik reisijad on kohal
        kulunud_aeg += 5; // liidame peatuses sõitmiseks kulunud aja
        // pealelaadimine
        while ((tramm.size() < kohtade_arv) && // kuni trammis on ruumi
                !trammisabad[peatus].empty()) { // ja peatuses on ootajaid
            tramm.push(trammisabad[peatus].front()); // paneme trammi esimese ootaja
            trammisabad[peatus].pop(); // ja eemaldame ta trammisabast
            kulunud_aeg++; // lisame sisenemiseks kulunud aja
        }

        peatus = (peatus + 1) % peatuste_arv; // järgmine peatus
        kulunud_aeg += 60; // liidame peatuste vahel sõitmiseks kulunud aja
    }

    cout << kulunud_aeg << endl;

    return 0;
}
```

3.10 KAHENDPUU

Puu on dünaamiline andmestruktuur, mille elemente nimetatakse **tippudeks**. Tipud on korraldatud range hierarhiana, see tähendab, et igal tipul võivad olla **alluvad**, aga neil saab olla ainult üks vahetu **ülemus**. **Juurtipp** kõige kõrgema taseme ülemus, sellel ülemust enam pole. Tippu, millel ei ole alluvaid, nimetatakse **leheks,** alluvatega tippu nimetatakse **vahetipuks** ehk **sõlmeks**.

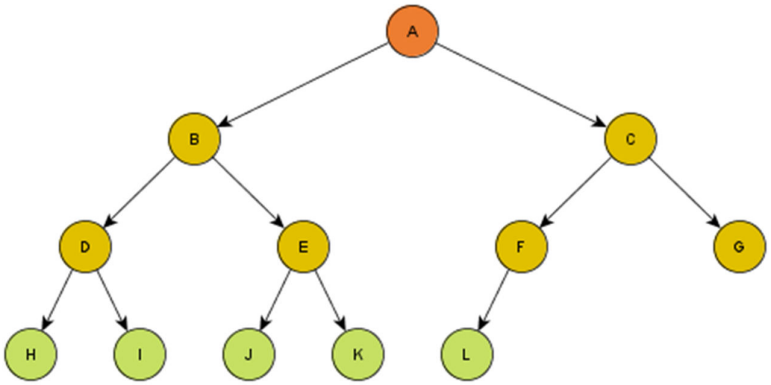
Juurtipp asub esimesel tasemel, kõik tema vahetud alluvad 2. tasemel, nende vahetud alluvad omakorda 3. tasemel jne. Tasemete arvu puus nimetatakse puu **kõrguseks.**

Puud, mille igal sõlmel on kuni n alluvat, nimetatakse n-puuks. Kõige levinum n-puu on **kahendpuu**. Puud nimetatakse **täielikuks**, kui tema igal vahetipul on sama palju alluvaid ning kõik lehed asuvad samal tasemel. n tipuga täieliku kahendpuu kõrgus on $\log_2 n$.



Täielik kahendpuu. Juurtipp on punane, lehed rohelised.

Kui täielikust kahendpuust on viimaselt tasemelt paremalt vasakule eemaldatud 0 või rohkem lehte, siis nimetatakse sellist kahendpuud **kompaktseks**.



Kompaktne kahendpuu.

Kahendpuus on sageli kasulik eristada, kas tipp on oma ülemuse vasak või parem alluv, seda ka juhul, kui ülemusel ongi vaid üks alluv.

Puudel tavaliselt realiseeritavatest operatsioonidest ja nende kasutamises tuleb põhjalikumalt juttu peatükis 6, Graafiteooria.

3.10.1 Kahendpuu esitus

Kahendpuud võib realiseerida nii, et iga tipp on objekt, millel on väljadena viidad tema vasakule ja paremale alluvale.

Sageli on efektiivsem ja lihtsam realiseerida kahendpuu massiivi peale. Puu juur on massiivi esimene element ehk a[0]. Selle vasakpoolne alluv on kohal a[1] ja parempoolne kohal a[2]. a[1] vasakpoolne alluv on kohal a[3] ja parempoolne alluv a[4] jne. Selline esitus sobib hästi kompaktsete kahendpuude hoidmiseks, kuna n-tipuline puu täidab täpselt massiivi pikkusega n.

Eelmisel joonisel kujutatud kompaktse kahendpuu esitus massiivina

Tipp	Asukoht massiivis
Puu juurelement	a[0]
Elemendi a[k] vasak alluv	a[2k+1]
parem alluv	a[2k+2]
ülemus	a[(k-1)/2]

3.10.2 Kahendotsingu puu

Olgu puu igas tipus võtmega kirje. Kahendotsingu puuks (*binary search tree*) nimetatakse sellist puud, mis on tühi või mille vasaku alampuu kõigi tippude võtmed on väiksemad kui juurtipu võti ning parema alampuu tippude võtmed on kõik suuremad kui juurtipu võti ning mõlemad alampuud on kahendotsingu puud.

Kahendotsingu puul kasutatavad operatsioonid on võtme otsimine, uue kirje lisamine ja kirje eemaldamine, samuti vähima ja suurima võtmega kirje leidmine.

Otsimine otsingupuust on lihtne: võrdleme kõigepealt otsitavat väärtust juurtipu võtmega – kui otsitav väärtus on suurem, peab see asuma parempoolses alampuus, kui aga väiksem, siis vasakpoolses. Sama reegel kehtib ka alampuudest otsimisel. Kuna igal sammul liigutakse puus ühe taseme võrra alla, siis sellise otsimise korral on halvima juhu keerukuseks puu kõrgus. Seetõttu on oluline hoida kahendotsingu puu kõrgus võimalikult väike ehk hoida puu tasakaalus.

Tasakaalustatud kahendotsingu puuks nimetatakse sellist kahendotsingu puud, mille iga tipu vasak ja parem alampuu on enam-vähem sama kõrgusega. Kui ühegi tipu alampuude kõrgused ei erine rohkem kui ühe võrra, siis sellist kahendotsingu puud nimetatakse **AVL-puuks.** Nimi tuleb selle puu leiutajate G. Adelson-Velski ja E. Landise nimetähtedest.

Ühtlase kõrguse hoidmine on oluline selleks, et sellises puus on tavapärased operatsioonid teostatavad keerukusega $O(log\ n)$. Loomulikult võib nii tipu lisamine kui ka eemaldamine viia sellise puu tasakaalust välja, seetõttu on oluline puu vajadusel pärast lisamist/eemaldamist tasakaalustada.

3.11 KUJUTIS

Kujutis (*map*) on andmestruktuur, mille elementideks on kaheosalised kirjed – võtme ja väärtuse paarid. Võtmed on kordumatud, st ühe ja sama võtmega elemente võib olla vaid üks. Peamisteks operatsioonideks on kirje lisamine ja eemaldamine ning võtme järgi kirje leidmine.

Kui võtmete võimalik väärtusvahemik on teada ning see ei ole kuigi ulatuslik, saab kasutada massiivi koos **vahetu indekseerimisega** – võtmele v vastava element kirjutatakse massiivi kohale $a[v]$. Puuduvate võtmete kohale kirjutatakse mingi spetsiaalne väärtus, et neid tegelikest võtmetest eristada. Sellisel moel on nii lisamine, eemaldamine kui ka võtme järgi elemendi leidmine keerukusega $O(1)$.

Kui aga võtmeid on rohkem, kasutatakse kujutise realiseerimiseks **paisktabelit** *(hash table)*. Paisktabel on samuti massiiv, kuid elementi võtmega v ei kirjutata otse kohale a[v], vaid kasutatakse **paiskfunktsiooni**, mis seab igale võtmele vastavusse mingi indeksi. See tähendab, et võtmele v vastav kirje kirjutatakse paisktabelisse kohale a[$f(v)$], kus f on paiskfunktsioon.

Paiskfunktsioon peab olema kiiresti arvutatav. Harilikult valitakse paiskfunktsiooniks $f(v) = v \bmod m$, kus m on paisktabeli ridade arv.

Paisktabeli realiseerimisel võetakse tavaliselt mugavalt palju ruumi, et andmed paisktabelisse vabalt ära mahuvad. Sellegipoolest võib paisktabelis esineda **põrkeid** ehk **kollisioone**. Põrge on selline olukord, kus paiskfunktsioon f annab kahe erineva võtme v_1 ja v_2 jaoks sama väärtuse, st $f(v_1) = f(v_2)$, kui $v_1 \neq v_1$.. Sisuliselt tähendab see, et kaks kirjet peaks asuma paisktabelis samal kohal.

Kokkupõrgete lahendamiseks kasutatakse peamiselt kahte erinevat võimalust. **Välisahelate** meetodi korral on igal real kaks lisavälja: üks selle jaoks, et märkida, kas rida on juba hõivatud ja teine on viit lihtahelale. Kui objekti võtmega v salvestatakse reale $f(v)$, kuid see rida on juba hõivatud, salvestatakse uus objekt realt $f(v)$ algavasse lihtahelasse.

Lahtise adresseerimise meetodi korral lahendatakse põrge järgmiselt: kui võtmele v vastav rida $f(v)$ on juba hõivatud, otsitakse eelmine vaba rida ning lisatav kirje salvestatakse sinna. Lugemisel tuleb siis samuti otsida võtit igalt eelnevalt realt, kuni võti leitakse või jõutakse tühja reani, mis tähendab, et antud võtit tabelis ei esine. Veelgi parem on kasutada topeltpaiskamisega lahtist adresseerimist, mille korral ei otsita vaba kohta järjest, vaid kasutatakse mingit teist funktsiooni $f_2(v)$, mis tagastab sammu pikkuse, mitu kohta tuleb esialgsest asukohast edasi minna, et leida võtmele vastav uus asukoht. Sellise meetodiga leitakse kirje otsimisel üldiselt kiiremini üles.

3.11.1 Kujutis programmeerimiskeeltes

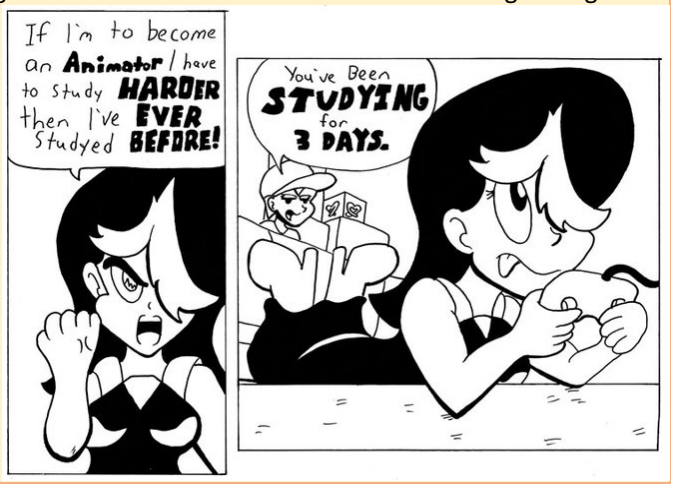
C++ pakub standardteegis kahte klassi `map` ja `unordered_map`. Mõlemad on mõeldud võti-väärtus paaride hoidmiseks, kuid `map` on tavaliselt realiseeritud kahendotsingupuuna, mistõttu on sealt elemendi leidmine aeglasem operatsioon, see-eest võimaldab mugavamat elementide läbikäimist (elemendid on võtme järgi sorteeritud). `unordered_map` on enamasti realiseeritud paisktabelina ja võimaldab võtme järgi ligipääsu konstantse keerukusega.

Java valik pole halvem: seal on klassid `TreeMap` ja `HashMap` ning nendel liides `Map`.

Pythonis on olemas andmetüüp `dict` (*dictionary*) võti-väärtus paaride hoidmiseks. Realiseeritud on see topeltpaiskamisega lahtise adresseerimisega paisktabelina.

3.12 PRAKTILINE ÜLESANNE

Usina linna avatud ülikoolis on igaühel võimalik käia kuulamas erinevaid kursusi. Usinas linnas elavad muidugi usinad õppijad, kes enamasti osalevad aastas viiel kursusel. Seetõttu on ülikoolil plaanis pakkuda 5-kursuselisi valmiskomplekte. Esialgu otsustatakse katsetada kombinatsiooni kõige populaarsematest kursustest, selleks aga on vaja teada, millise viie kursuse kombinatsioon esineb kõige sagedamini. Erinevaid kursusi on palju ja need on nummerdatud arvudega 100…499.
Sisendi esimesel real on arv $n (1 \le n \le 10000)$ mis näitab, kui palju on olnud eelneval aastal viie kursuse valijaid, ning järgneval n real igaühel ühe õppija valitud kursuste numbrid valimise järjekorras, tühikutega eraldatuna. Väljastada kõige sagedasem kursuse kombinatsioon. Kui sama sagedusega on mitu kombinatsiooni, väljastada kõik erinevad suurima sagedusega kombinatsioonid, iga kombinatsioon eraldi real.



NÄIDE:

```
5
101 102 103 104 105
217 301 101 105 400
101 301 400 217 105
101 301 103 104 105
105 104 103 102 101
```

Vastus

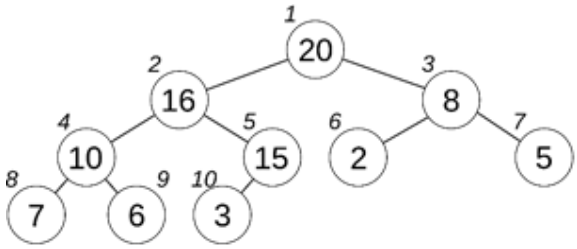
```
101 102 103 104 105
101 105 217 301 400
```

Siin on peamiseks küsimuseks, kuidas leida korduvad kombinatsioonid ja neid loendada. Erinevaid võimalikke kombinatsioone on $\binom{400}{5}$, mis on ligikaudu 83 miljardit. See ei aita meid kuidagi edasi. Üheks võimaluseks oleks välja mõelda funktsioon, mis igale kombinatsioonile seab vastavusse ühe arvu – nii saaks leida iga kombinatsiooni jaoks sisendis sellele vastava arvu ning loendada, kui mitu korda mingit arvu esineb. Antud juhul on loomulik meetod kasutada kursusenumbreid stringidena ja need kokku kleepida üheks pikaks stringiks. Kui enne kursused mingis järjekorras sorteerida, siis samale kombinatsioonile vastab sama string (ja vastupidi – ühele stringile vastab üks kindel kombinatsioon) ja seda stringi saame kasutada võtmena:

```
#include <map>
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    map<string, int> logi;

    // siin hoiame seni leitud suurimat esinemissagedust. Kuna ülesande tingimustes on
    //vähemalt 1 kombinatsioon olemas, võib selle panna üheks.
    int maxN = 1;
    string kursused[5]; //siin hoiame ühe inimese võetud kursusi
```

```
    for (int i = 0; i < n; i++) {
        cin>> kursused[0] >> kursused[1] >> kursused[2] >> kursused[3] >> kursused[4];
        sort(kursused, kursused + 5); // sorteerime kursused kasvavas järjekorras

        string voti;
        for (int j = 0; j < 5; j++) {
            voti += kursused[j]; // ühendame kursuste numbrid üheks stringiks
        }
        // kui sellist võtit pole (sellist kombinatsiooni ei esine)
        if (!logi.count(voti))
{           logi[voti] = 1; // lisame selle väärtusega 1.
        }
        else {  // kui võti on
            int m = logi[voti] + 1; // suurendame kombinatsiooni korduste arvu
            logi[voti] = m; // ja muudame vastavaks
            // kui on suurem korduste arv, kui senine maksimum, muudame maksimumi.
            maxN = max(maxN, m);
        }
    }

    map<string, int>::iterator it;
    for (it = logi.begin(); it != logi.end(); it++) { // käime mapi läbi
        if (it->second == maxN) // kui on suurim korduste arv
            for (int i = 0; i < 13; i += 3) {
                // eraldame võtmest kursuste numbrid ja väljastame need
                cout << it->first.substr(i, 3) << " ";
            }
        cout << endl;
    }

    return 0;
}
```

3.13 KUHI



Kuhi (*heap*) on selline puu, millel kehtib **kuhjaomadus**: ühegi tipu võti ei ole suurem kui tema ülemuse võti. Kui kehtib vastupidine omadus – ühegi tipu võti ei ole väiksem kui tema ülemuse võti – siis on tegemist **pöördkuhjaga**.

Kahendkuhi on kuhi, mis on ühtlasi ka kompaktne kahendpuu. Sageli mõeldaksegi kuhja all justnimelt kahendkuhja ning siingi peatükis vaatame edasi vaid kahendkuhja.

Kuna kuhi on kompaktne kahendpuu, siis sobib selle esituseks väga hästi massiiv.

Kuhjal on enamasti realiseeritud järgmised operatsioonid:

- Suurima (või vähima) elemendi (võtme)väärtuse leidmine, keerukus $\Theta(1)$;
- Lisamine, keerukus $O(log(n))$;
- Suurima (vähima) elemendi kustutamine, keerukus $\Theta(log(n))$
- Kuhjastamine ehk etteantud elementide kuhjaks paigutamine;

Operatsioone kuhjal võib realiseerida mitmel moel, lisamisel ja eemaldamisel tuleb aga hoolega silmas pidada, et kuhja omadus säiliks. Selleks on kuhjal enamasti meetodid elementide nihutamiseks üles- ja allapoole. Lisamine on harilikult realiseeritud nii, et uus element lisatakse kuhja lõppu ning

nihutatakse seda kuhjas ülespoole, kuni kuhja omadus on taastatud. Eemaldamisel asendatakse juurelement viimase elemendiga kuhjas, seejärel kukutatakse see element kuhjas õigesse kohta ja kuhjaomadus saab taastatud.

Kuhjasid kasutatakse:

- kuhjameetodil sorteerimiseks ($O(n \log n)$),
- valikualgorimide (*selection algorithms*) realiseerimiseks, kuna miinimum ja maksimumelemendi leidmine on kuhjas konstantse ajaga ning mediaani ja k. elemendi leidmine toimuvad vähem kui lineaarse ajaga,
- mitmete graafitöötlusalgoritmide realiseerimiseks,
- eelistusjärjekorra realiseerimiseks.

C++ keeles on päisfailis `<queue>` klass `priority_queue`. Selle esimene, eemaldatav element on alati suurim element. Kuna kuhja on mugav hoida massiivis, siis `priority_queue` kasutab vaikimisi `vector` klassi tegeliku konteinerina ning rakendab sellel operatsioone kuhjastruktuuri säilitamiseks.

Javas on klass `PriorityQueue`, mis on oma olemuselt pöördkuhi, nii et Javas võetakse eelistusjärjekorrast järgmisena kõige väiksem element.

Pythonis on hea kasutada `Lib/heapq.py`. Nagu Javagi, kasutab Python pöördkuhja ehk esimene element on kõige väiksem.

Järgmine tabel illustreerib kuhja kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine	priority_queue<int> e	PriorityQueue<Integer> e = new PriorityQueue<Integer> ();	e = []; heapify(e)
Elemendi 'a' lisamine	e.push(a)	e.add(a)	heappush(e, a)
Elemendi eemaldamine	e.pop()	a = e.remove(); a = e.poll()	a = heappop(e)
Esimese elemendi vaatamine	a = e.top()	a = e.element(); a = e.peek()	a= e[0]
Kas järjekord on tühi?	e.empty()	e.isEmpty()	not e

3.14 SORTIMINE

Sortimine on etteantud hulga elementide mingisse kindlasse järjestusse seadmine. Enamasti kasutatakse numbrilist või leksikograafilist järjestust. Elemendid antakse harilikult ette massiivina ning sortimise tulemuseks on massiiv, kus

- ükski eelnev element ei ole suurem järgnevast,
- väljund on sisendi permutatsioon.

Erinevaid sortimisalgoritme on päris palju. Sortimisalgoritmi nimetatakse **stabiilseks**, kui võrdsete elementide korral säilib nende omavaheline järjestus algses jadas. Näiteks, kui on antud mängukaardid järjestuses ♣7 ♥2 ♥7 ♣8, siis numbrite järgi sorteerides on korrektsed nii ♥2 ♣7 ♥7 ♣8 kui ka ♥2 ♥7 ♣7 ♣8, kuid ainult esimeses on säilinud seitsmete omavaheline algne järjestus: risti seitse on eespool ärtu seitsmest.

Öeldakse, et sortimisalgoritm töötab **kohapeal** *(in-place)*, kui algoritm ei vaja algandmete sorteerimiseks rohkem kui konstantset hulka lisamälu.

Sõnastame sorteerimisülesande ja vaatame selle erinevaid lahendusi:

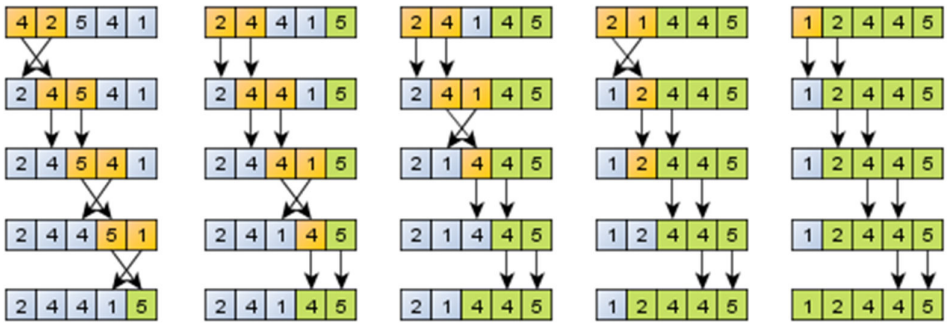
On antud n-elemendiline täisarvudest koosnev jada. Paiguta elemendid ümber nii, et $a_1 \leq a_2 \leq \cdots \leq a_n$.

3.14.1 Mullimeetod

Mullimeetod (*bubble sort*) on üks vanemaid ja tuntumaid sortimisalgoritme. Algoritm on lihtsasti mõistetav ning ka väga lihtne programmeerida, mistõttu tutvustatakse seda sageli programmeerimise algkursustel:

```
int mullimeetod(int* jada, int pikkus)
{
    bool vahetus = true;
    while (vahetus) { //kui on olnud vahetusi
        vahetus = false; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                vahetus = true; //ja peame meeles, et vahetus toimus
            }
        }
    }
}
```

Järgnev skeem kujutab selle algoritmi tööd:



Skeemilt on lihtne märgata, et esimesel läbikäimisel satub kõige suurem element jada lõppu, teisel läbikäimisel pannakse eelviimane element oma kohale jne, mis tähendab, et tegelikult ei ole vaja igal sammul kogu jada läbi käia. Veelgi enam – pole raske veenduda, et kõik elemendid, mis asuvad viimasest vahetusest tagapool, on juba sorteeritud. Siin on optimeeritud lahendus:

```
int optmullimeetod(int* jada, int pikkus)
{
    int viimane = pikkus;
    while (viimane > 0) { //kui on olnud vahetusi
        viimane = 0; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                viimane = i; //ja peame meeles viimase vahetuse koha
            }
        }
        pikkus = viimane;
    }
}
```

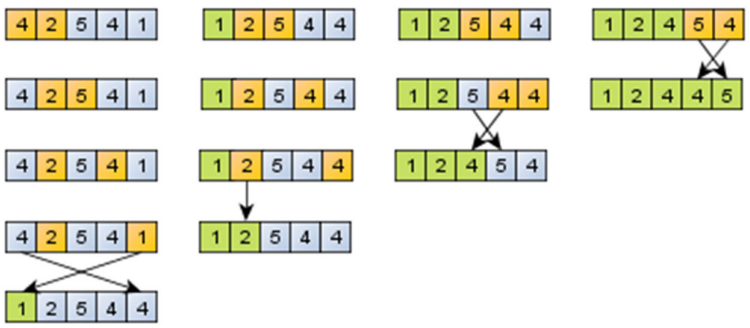
Mullimeetodi keerukus on $O(n^2)$ ning isegi teiste sama keerukusklassi sortimisalgoritmidega võrreldes (nt pistemeetod) on mullimeetod üldjuhul aeglasem, kuna keskmiselt tehakse rohkem elementide vahetusi.

3.14.2 Valikmeetod

Valikmeetod (*selection sort*) on samuti väga lihtne sortimisalgoritm: kõigepealt leitakse jadas kõige väiksem element ning vahetatakse see kõige esimese elemendiga. Seega pärast vahetust on kõige väiksem element oma kohal ning protseduuri korratakse ülejäänud jada elementidel, kuni kõik elemendid on õige koha leidnud:

```
void valikmeetod(int* jada, int pikkus)
{
    for (int j = 0; j < pikkus - 1; j++) { //üksik element on vähim, piisab
    // eelviimaseni vaatamisest
        int min = j; //paneme esimese õige asukohata elemendi minimaalseks
        for (int i = j + 1; i < pikkus; i++) { //käime jada lõpuni läbi
            if (jada[i] < jada[min]) { //leidime väiksema elemendi
                min = i; //jätame asukoha meelde
            }
        }
        if (min != j) { //kui vähim element ei ole juba õigel kohal
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[min];
            jada[min] = temp;
        }
    }
}
```

Siin on skeem, mis kujutab selle algoritmi tööd:



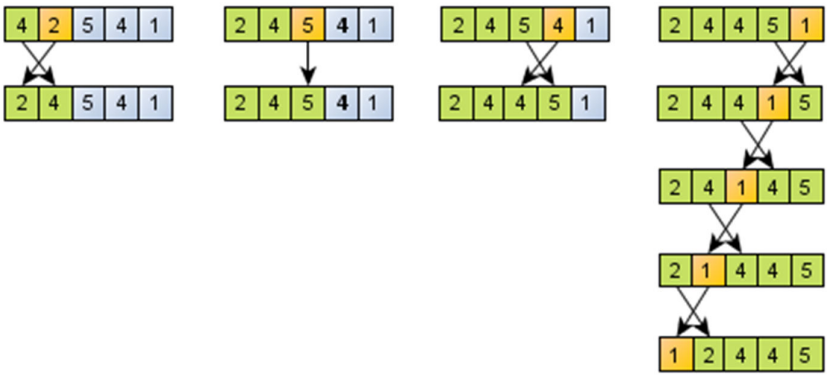
Valikmeetodi keerukus on samuti on $O(n^2)$. Samas on skeemilt kohe näha, et tegelikke vahetusi tehakse oluliselt vähem kui mullimeetodit kasutades – vahetuste arv ei ole kunagi suurem kui elementide arv.

3.14.3 Pistemeetod

Kolmas $O(n^2)$ keerukusega sortimisalgoritm on pistemeetod (*insertion sort*). Pistemeetod on sageli kasutuses väiksematel andmemahtudel ning on keskmiselt kiirem kui mulli- või valikmeetod ning sedagi on lihtne implementeerida:

```
void pistemeetod(int* jada, int pikkus)
{
    for (int i = 1; i < pikkus; i++) { //vaatame iga elementi alates teisest
    //kuni vaadeldava elemendini jada[i] on jada sorteeritud. Otsime õige koha jadas
        for (int j = i; j > 0 && jada[j - 1] > jada[j]; j--) {
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[j - 1];
            jada[j - 1] = temp;
        }
    }
}
```

Pistemeetod on kujutatud järgmisel skeemil:



Pistemeetodi tegelik keerukus oleneb algandmetest. Nagu mainitud, siis halvimal juhul – kui jada on kahanevas järjestuses – tuleb teha n^2 võrdlust ja vahetust, samas, kui andmed on juba õiges järjekorras, tehakse vaid n võrdlust. Väikeste jadade korral ($n < 20$) on pistemeetod kiirem kui keerulisemad sortimisalgoritmid ja seda kombineeritakse sageli kiirmeetodiga.

3.14.4 Põimemeetod

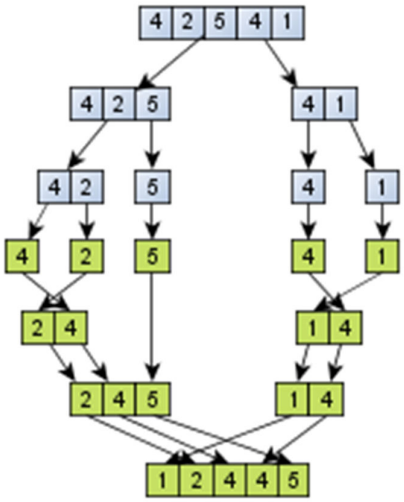
Põimemeetod (*merge sort*) on juba mõnevõrra keerulisem sortimisalgoritm. Selle mõtles 1945. aastal välja arvutiteaduse üks „isasid“ John von Neumann. Tegemist on „jaga ja valitse“ tüüpi algoritmiga, kus jada jagatakse pooleks ja sorteeritakse kumbki pool eraldi, hiljem sorteeritud pooled põimitakse. Loomulikult ei piirduta vaid ühekordse poolitamise ja põimimisega, vaid seda tehakse seni, kuni alamjadad on soovitud pikkusega.

Algoritmi põhiosa on lihtne:

```
void poimemeetod(int* jada, int v, int p)
{
    if (v < p) {
        int keskkoht = (v + p) / 2;
        poimemeetod(jada, v, keskkoht);
        poimemeetod(jada, keskkoht + 1, p);
        poimi(jada, v, keskkoht, p);
    }
}
```

Põimimise osa nõuab rohkem näputööd:

```
void poimi(int* jada, int v, int keskkoht, int p)
{
    int i, j, k;
    int vpikkus = keskkoht - v + 1; // esimese lõigu pikkus
    int ppikkus = p - keskkoht; // teise lõigu pikkus

    int *V = new int[vpikkus]; // põimimiseks teeme vasaku ja
    int *P = new int[ppikkus]; // parema lõigu jaoks eraldi massiivid

    for (i = 0; i < vpikkus; i++)
        V[i] = jada[v + i];
    for (j = 0; j < ppikkus; j++)
        P[j] = jada[keskkoht + 1 + j];

    i = 0; // vasakpoolse jada esimese elemendi indeks
    j = 0; // parempoolse jada esimese elemendi indeks
    k = v; // põimitava koha algusindeks jadas
    while (i < vpikkus && j < ppikkus) { // kuni mõlemas jadas on vaatamata elemente
        if (V[i] <= P[j]) { // vasakul on mittesuurem paigutamata element
            jada[k++] = V[i++]; // paneme selle jadasse
        }
        else { // paremal on väiksem element
            jada[k++] = P[j++]; // paigutame selle jadasse
        }
    }

    while (i < vpikkus) {//kui vasakus jadas on veel elemente
        jada[k++] = V[i++]; //kopeerime need järjest jadasse
    }

    while (j < ppikkus) { // kui paremas jadas on veel elemente
        jada[k++] = P[j++]; // kopeerime need järjest jadasse
    }
}
```



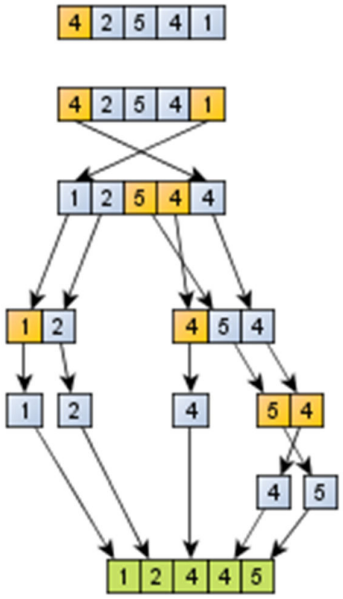
Põimemeetodi keerukusklass (ka halvimal juhul) on $O(n \log n)$ ja see sobib eelkõige suuremate jadade sorteerimiseks. Põimemeetodi miinuseks on, et see meetod nõuab lisamälu mahus $\Omega(n)$. Eksisteerib ka selline variant põimimisest, mis teeb seda kohapeal, ilma märkimisväärselt lisamälu kasutamata, kuid see algoritm on juba päris keeruline. Huvilistele: https://github.com/liuxinyu95/AlgoXY/blob/algoxy/sorting/merge-sort/src/mergesort.c
Põimemeetod sobib väga hästi ahelate sorteerimiseks. Sellisel juhul puudub ka vajadus lisamälu jaoks.

3.14.5 Kiirmeetod

Praegu tuntud algoritmidest on üldjuhul kõige kiirem meetod suurte jadade sortimiseks kiirmeetod (*quicksort*). Ka kiirmeetod on „jaga ja valitse“ tüüpi algoritm. Algoritmi idee on järgmine: sorteerimata jadast võetakse üks element, nn veelahe (*pivot*) ning võrreldakse teisi elemente sellega. Kõik elemendid, mis on veelahkmest väiksemad või võrdsed sellega, paigutatakse valitud elemendist vasakule ehk ettepoole, need aga, mis on suuremad, paigutatakse paremale ehk tahapoole. Selle tulemusena on veelahkmest vasakul ainult sellest mittesuuremad elemendid ja paremal ainult suuremad elemendid. Seejärel sooritatakse sama protseduur valitud elemendist vasakul ja paremal pool eraldi. Kui vasakul või paremal on vähem kui kaks elementi, on see osa juba sorteeritud ja rohkem ei olegi vaja midagi teha.

```
int jaota(int* jada, int v, int p)
{
    int e = jada[v];
    int i = v - 1;
    int j = p + 1;
    while (true) {
        while (jada[++i] < e) {}
        while (jada[--j] > e) {}
        if (i >= j) return j;
        int temp = jada[i];
        jada[i] = jada[j];
        jada[j] = temp;
    }
}
```

```
void kiirmeetod(int* jada, int v, int p)
{
    if (v < p) {
        int e = jaota(jada, v, p);
        kiirmeetod(jada, v, e);
        kiirmeetod(jada, e + 1, p);
    }
}
```



Kiirmeetodi keskmine keerukus on $O(nlog\ n)$. Halvimal juhul on kiirmeetodi keerukus küll $O(n^2)$, kuid praktikas on tegemist siiski kõige kiirema sorteerimisalgoritmiga. Sageli kombineeritakse kiirmeetodit pistemeetodiga – kui sorteeritava alamjada pikkus on väike, näiteks kuni 20 elementi, sorteeritakse see pistemeetodil, vastasel juhul vastavalt kiirmeetodi algoritmile.

Kiirmeetodi probleemseks kohaks võib osutuda selle meetodi ebastabiilsus.

3.14.6 Võrdlustel põhineva sortimise keerukuse alampiir

Eelnevalt vaadeldud sortimisalgoritmid põhinesid kõik elementide võrdlemisel. Selliste algoritmide keerukuse alampiir on $O(n \log n)$. Miks?

Iga võrdlustehe annab meile informatsiooni jada järjestuse kohta. Kokku on n-elemendilisel jadal $n!$ erinevat võimalikku järjestust. Saame konstrueerida otsustuste puu, kus lehtedeks on kõik võimalikud n permutatsioonid ja sõlmedeks võrdlustehted.

Võrdlustehtega sõlmel on

- vasakuks alluvaks on võrdlustehe, mille algoritm sooritab siis, kui võrdluse tulemus on negatiivne,
- paremaks alluvaks on võrdlustehe, mis sooritatakse siis, kui tulemus oli positiivne.

Iga permutatsioonini viib täpselt üks otsustuste tee. Kõige lühem selline tee ehk lehe vähim võimalik kaugus puu juurest vastab parimale juhule – kõige pikem tee – ehk puu kõrgus vastab halvimale võimalikule juhule. $n!$ lehega kahendpuu minimaalseks kõrguseks on $\log_2(n!) = \log_2 1 + \log_2 2$

+…+ $\log_2 n$. Kuna hetkel huvitab meid ainult alampiir, siis pole vaja leida täpset tulemust. Selle summa viimased $n/2$ liiget $\log_2 \frac{n}{2} \ldots \log_2 n$ ei ole ükski suurem kui $\log_2 \frac{n}{2}$. Seega

$$\log_2(n!) \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 n - \frac{n}{2}.$$

Järelikult on sellise puu minimaalne kõrgus $O(n \log n)$ ja seega ei saa võrdlemistel põhineva sorteerimisalgoritmi keerukus halvimal juhul olla parem kui $\Omega(n \log n)$.

3.15 SORTIMISE ERIMEETODID

Kui sisendandmed vastavad teatud täiendavatele piirangutele, siis mõningatel juhtudel on võimalik kasutada ka efektiivsemaid sortimisalgoritme, mille keerukusklass on $O(n)$. Tuntumad neist on loendamismeetod (*count sort*), positsioonimeetod (*radix sort*) ja kimbumeetod (*bucket sort*). Kõik need meetodid kasutavad lisamälu mahus $\Omega(n)$.

3.15.1 Loendamismeetod

Loendamismeetodit saab kasutada, kui andmed on teadaolevas ja küllaltki kitsas vahemikus. Loendamismeetodi idee seisneb selles, et luuakse eraldi loendusmassiiv, kus igale indeksile vastab üks elemendi võimalik väärtus. Kui suurim jadas esinev väärtus on max ja vähim min, siis on loodava jada pikkus $max - min + 1$. Seejärel käiakse esialgne jada elementhaaval läbi ning suurendatakse loendusmassiivis elementi kohal a_{i-min}. Sellisel moel loendatakse iga element.

0	1	2	3	4	indeks
1	1	0	2	1	
1	**2**	**3**	**4**	**5**	**väärtus, mida loendame**

Massiivile 4,5,2,4,1 vastav loendusmassiiv

Kui loendasime arve, pole edasi vaja teha muud, kui vastavalt loendusmassiivile kirjutada igale indeksile vastavat väärtust esialgsesse massiivi nii mitu korda, kui seda esines. Kui tegemist on aga objektidega, siis saab arvutada, mitmendal kohal peaks sorteeritud massiivis olema antud võtmeväärtusega element. Selleks arvutatakse loendusmassiivis iga elemendi m[i] jaoks alates teisest m[i] + m[i-1], s.t leitakse selle võtmega elemendi maksimaalne positsioon sorteeritud jadas. Loendusmassiiv on siis kujul [1, 2, 2, 4, 5].

Käime esialgse jada tagurpidi läbi ja iga elemendi jaoks leiame selle positsiooni sorteeritud jadas: j = m[a[i]- v] – 1. Kirjutamegi elemendi sellele positsioonile uude vastusjadasse ning vähendame loendusmassiivis selle elemendi võtmele vastavat väärtust. Näiteks, pannud viimase elemendi (1) vastusjadasse `[1,x,x,x,x]`, vähendame loendusmassiivis väärtust kohal 1-1 ja saame `[0,2,2,4,5]`. Järgmiseks paigutame massiivi eelviimase elemendi 4, ning saame vastusmassiivi `[1,x,x,4,x]` ning loendusmassiivi `[0,2,2,3,5]` – järgmine „4“, mis algses jadas ette tuleb, paigutatakse nüüd kohale 3 - 1 = 2. Sellisel moel elemente paigutades on tagatud ka stabiilsus.

See meetod vajab lisamälu vähemalt loendusmassiivi pikkuse k jagu. Objektide korral on lisaks sellele vaja veel ühe n-elemendilise jada jagu mälu sorteeritud elementide hoidmiseks, mis teeb kokku lisamälu vajaduseks $\Omega(n + k)$. Ka loendusmassiiv tuleb korra läbi käia, mistõttu omab selle algoritmi kasutamine mõtet siis, kui k on väike.

3.15.2 Positsioonimeetod

Positsioonimeetodiga sorteeritakse harilikult positiivseid, mitte väga pikki täisarve. Meetod seisneb selles, et arve ei vaadelda arvudena, vaid numbritest koosnevate liitvõtmetena. Seega kui näiteks kolmekohalisi arve sorteerides sorteerime need esmalt üheliste järgi, seejärel kümneliste järgi ja siis sajaliste järgi, saamegi tulemuseks sorteeritud jada.

Alamalgoritmina kasutatakse sorteerimiseks loendamismeetodit, seetõttu peavad võtmed olema kõik sama pikkusega. Kasutus ei piirdu siiski ainult arvudega, sel moel saab sorteerida ka sõnu.

3.15.3 Kimbumeetod

Kimbumeetod seisneb selles, et väärtuste järgi jaotatakse elemendid eraldi kimpudesse või ämbritesse, mida siis sorteeritakse edasi kas kimbumeetodil või mõnel muul viisil. Näiteks sorteerides jada, kus on arvud 0…99, eraldame ühte kimpu kõik arvud 0..9, teise 10…19 jne. Sorteerides need kimbud eraldi, saame tulemustest kokku panna sorteeritud jada.

3.16 MITME KRITEERIUMI JÄRGI SORTIMINE

Mõnikord võib olla vaja sorteerida mitme kriteeriumi järgi, näiteks kaarte masti ja väärtuse järgi. Üheks võimaluseks on kasutada mõnda stabiilset sortimismeetodit ning sortida iga kriteeriumi järgi eraldi, alustades kõige vähemtähtsamast. Kui kriteeriume on k, tuleb jada sorteerida k korda ehk sellise lähenemise keerukuseks on $O(kn \log n)$.

Tavalisemaks lahenduseks on koostada oma spetsiaalne võrdlemisfunktsioon, mis suudab objektid etteantud järjestuses reastada, võrreldes kõigepealt kõige olulisemat kriteeriumi ja kui selle järgi on objektid võrdsed, siis tähtsuselt järgmist jne. Halvimal juhul tuleb iga võrdlusoperatsiooni korrata k korda – iga kriteeriumi jaoks – ja seegi lähenemine annab halvimal juhul keerukuseks $O(kn \log n)$. Päriselt töötab see lahendus aga kiiremini kui eelmine, sest enamasti ei ole vaja siiski võrrelda kõiki kriteeriume ning teiseks päästab see lähenemine vajadusest kasutada stabiilset sortimisalgoritmi.

3.17 PROGRAMMEERIMISKEELTE STANDARDTEEGID

Enamasti pole siiski vaja jalgratast leiutada, kuna mõistlik üldine sorteerimine on keelte standardteekides olemas. Eriti oluline on see võistlustel, kus lahendamisaeg on piiratud ning oma sortimisalgoritmi kirjutamine ja testimine on liigne ajakulu. Seetõttu peaks iga võistleja end kurssi viima oma keele pakutavate võimalustega.

Keele spetsifikatsioon määrab enamasti ära, mis tingimused sortimismeetodile kehtima peavad. Praegu nõuavad nii C++, Java kui ka Python, et sortimisfunktsiooni halvima juhu keerukus on $O(n \log n)$. Java ja Pythoni sortimisfunktsioonid on stabiilsed, C++ seda nõuet ei esita, rõhutakse pigem nn kohapeal sortimisele. Millist algoritmi sortimiseks konkreetselt kasutatakse, sõltub juba konkreetselt kasutatavast teegist. Näiteks GNU C++ standardteek kasutab *introsort* algoritmi kombineerituna pistemeetodiga. *Introsort* on kiirmeetodi ja kuhjameetodi hübriid. Pythoniga tuli kasutusse *Timsort* algoritm, mis on tuletatud põime- ja pistemeetodist. Ka uuemad Java versioonid kasutavad *Timsort*i objektide sortimiseks, arvutüüpidel kasutatakse kahe lahkmega kiirmeetodi varianti.

3.17.1 C++

C++ on päisfailis <algorithm> defineeritud funktsioon sort, millele antakse ette mingi järjendi algus ja lõpp. Näiteks massiivi korral:

```
int n = 5
array size int t[] = {4,2,5,3,5}
sort(t, t+n);
```

vektori (`vector`) korral:

```
sort(v.begin(), v.end());
```

Vaikimisi sorteeritakse elemendid kasvavas järjekorras ja võrdlusoperaatori järgi. Paaride (`pair`) ja ennikute (`tuple`) korral sorteeritakse need vaikimisi esimese elemendi järgi, kui esimesed elemendid on võrdsed, siis teise järgi jne. Enda defineeritud struktuuridel ja objektidel tuleb ise defineerida ka võrdlusoperaator '<'. Samuti saab sort funktsioonile parameetrina ette anda võrdlusfunktsiooni, mis saab ette kaks argumenti ja tagastab tõeväärtuse: tõese, kui esimene argument on väiksem kui teine ja väära muul juhul:

```
bool cmp(string a, string b)
{
    if (a.size() != b.size())
        return a.size() < b.size();
    return a < b;
}
sort(v.begin(), v.end(), cmp);
```

On väga oluline meeles pidada, et sort ei ole stabiilne. Kui on vaja tagada stabiilsus, tuleb kasutada funktsiooni stable_sort, mis enamasti põhineb põimemeetodil.

3.17.2 Java

Javas on päises `Java.Util.Arrays` meetod `Arrays.sort`, millele antakse ette jada elementidest. Kui on vaja sorteerida lõiku jadast, saab ette anda ka alguse ja lõpu indeksid. Tasub tähele panna, et algus kuulub lõiku, lõpp aga mitte.

3.17.3 Python

Pythonis saab sortimiseks kasutada kaht peamist viisi:

- funktsioon `sorted` saab ette järjendi ja tagastab teise, sorteeritud järjendi,
- järjendil endal on meetod `sort`.

Vaikimisi sorteeritakse elemendid kasvavas järjekorras:

```
>>> sorted([4,5,2,4,1])
[1, 2, 4, 4, 5]
```

Mõlemale funktsioonile saab ette anda lipu reverse, mille tõese väärtuse korral jada sorteeritakse kahanevalt.

Lisaks on neil funktsioonidel ka parameeter `key`, mis määrab, mille järgi tegelikult sorteeritakse. `key` on üheargumendiline funktsioon, mille argumendiks on element ja tagastatavaks väärtuseks selle elemendi võti, mida kasutatakse sorteerimiseks.

Kindlasti tasub tutvuda ka mooduliga `operator`, kus on funktsioonid `itemgetter`, `attrgetter` ja `methodcaller`, mis teevad võtme etteandmise oluliselt mugavamaks ja võimaldavad ka mitme parameetri järgi sorteerimist:

```
>>> from operator import itemgetter
>>> opilased = [('Andres', 14 , 8),('Kati', 10, 3),('Mati', 9, 3)]

>>> sorted(opilased, key=itemgetter(2, 0, 1))
[('Kati', 10, 3), ('Mati', 9, 3), ('Andres', 14, 8)]
```

Muidugi võib võtme leidmise funktsiooni alati ise kirjutada. Võtme leidmise funktsioon ei pea olema rangelt objekti juures, vaid saab kasutada ka muid funktsioone:

```
>>> opilased = ['Andres', 'Kati', 'Mati']
>>> hinded = {'Mati': '3', 'Kati':'4', 'Andres': '5'}
>>> sorted(opilased, key=hinded.__getitem__, reverse=True)
['Andres', 'Kati', 'Mati']
```

Python3 versioonides kasutab sort objektide `__lt__()` funktsiooni. Kui oma objektil see meetod ise defineerida, saab määrata ka objektide sorteerimise järjestust. Python2 kasutades on soovitatav defineerida kõik võrdlusfunktsioonid.

Üks näiteülesanne:

Pärtlil on `n` mängukaarti. Ta soovib sorteerida kaardid nii, et kõige peal on potid, siis ärtud, seejärel ruutud ning kõige lõpuks ristid. Lisaks soovib ta, et iga mast on sorteeritud nii, et kõige peal on äss, seejärel kuningas, emand, soldat ja siis numbrid kahanevas järjekorras (10...2). Sisendi esimesel real on kaartide arv `n(1≤n≤1000000).` Järgmisel real on n kahemärgilist stringi, millest igaüks vastab mingile kaardile kaardipakist. Esimene tähemärk tähistab masti (C – risti, D – ruutu, H – ärtu ja S – poti), teine märk väärtust (2...9, T – 10, J – soldat, Q – emand, K – kuningas ja A – äss). Väljundiks on sorteeritud kaardipakk, nii et esimene on pealmine kaart, 2. pealmisest järgmine kaart jne.

NÄIDE:

```
5
C8 D8 SK DQ SK
```

Vastus:

```
SK SK DQ D8 C8
```

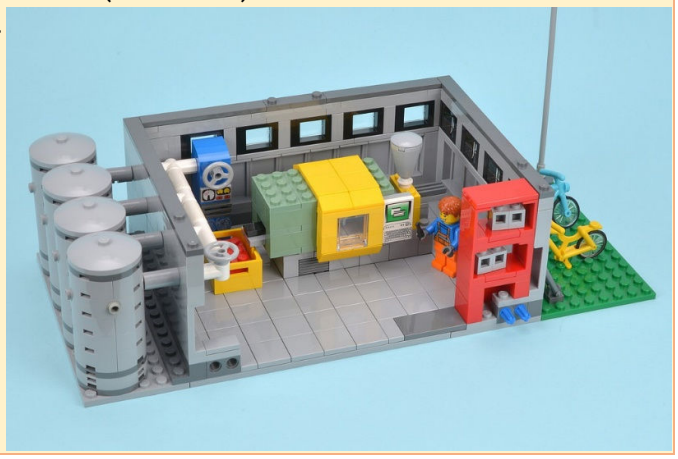
```cpp
#include <iostream>
#include <string>
#include <algorithm> //siin on sort meetod
using namespace std;

string mudel = "AKQJT98765432";

bool on_pealpool(string a, string b) // võrdlusfunktsioon kaardistringide võrdlemiseks
{
    if (a[0] != b[0])
        return (a[0] > b[0]); // mastid on tähestiku järjekorras tagurpidi
    else {
       // võrdleme teise tähemärgi positsioone mudelstringis: see, mis on eespool, on
       // väiksem (ehk pakis pealpool)
        return mudel.find(a[1]) < mudel.find(b[1]);
   }
}
```

```
int main()
{
    int n;
    cin >> n;
    string* pakk = new string[n];

    for (int i = 0; i < n; i++)
        cin >> pakk[i];

    // sorteerime paki, kasutades oma võrdlusfunktsiooni
    sort(pakk, pakk + n, on_pealpool);
    for (int i = 0; i < n; i++)
        cout << pakk[i] + ' ';
    return 0;
}
```

3.18 KONTROLLÜLESANDED

3.18.1 Vabrikud

Päikeselinnas on N vabrikut (1≤N≤100), millest igaüks valmistab oma graafiku alusel kaupa. Täpsemalt kulub igal vabrikul uue kaubapartii valmistamiseks P_i päeva, kus 1≤i≤N. Kui kaup on valmis, tullakse sellele veoautoga järele. Igal päeval, mil mõnes vabrikus on kaupa, tuleb veoautojuht ja toob kõigist vabrikutest valminud kaubad ära. Juht töötab aga ainult tööpäevadel, laupäeviti ja pühapäeviti tuleb valminud kaubad lihtsalt kohapeal ära müüa.
Antud on periood, mida uurida ning vabrikute andmed. Leida, mitu reisi peab veoautojuht selle aja jooksul tegema. Aja lugemine algab alati esmaspäevast.
Sisendi esimesel real on uuritava perioodi pikkus T (1≤T≤10000). Teisel real on vabrikute arv N. Järgmisel N real on arvud P_i (1≤Pi≤1000).



NÄIDE:
14
3
3
4
8
Vastus:
5
(esimene vabrik väljastab kaupa 3., 6., 9. ja 12. päeval, teine vabrik 4., 8. ja 12. päeval, kolmas ainult 8. päeval. Kuues päev jääb lugemata, sest see on laupäev).

3.18.2 Jalgpalliturniir

Jalgpalliturniiril saadakse 3 punkti võidu, 1 punkt viigi ning 0 punkti kaotuse eest. Meeskonnad järjestatakse järgmiste kriteeriumide alusel (kui kõigi eelmiste kriteeriumide tulemus on võrdne, võetakse järgmine)

1. Punktide arv (rohkem on parem)
2. Võitude arv (rohkem on parem)
3. Väravate vahe (suurem on parem)
4. Löödud väravate arv (rohkem on parem)
5. Mängitud mängude arv (vähem on parem)
6. Tähestikuline järjekord

Antud on jalgpalliturniiri mängude tulemused. Leida selle põhjal meeskondade järjestus.

Sisendi esimesel real on meeskondade arv `N(1≤N≤1000)`. Järgmisel `N` real on igaühel ühe meeskonna nimi. Nimed koosnevad kuni 32 märgist ja ainult ladina tähtedest. Meeskondadele järgneval real on mängude arv `M (1≤M≤1000)`. Järgmisel `M` real on igaühel ühe mängu tulemus järgmises formaadis: <Nimi1>-<Nimi2> <skoor1>:<skoor2>, näiteks `Eesti-Soome 1:2` Väljundisse kirjutada meeskondade järjestus ülaltoodud reeglite alusel.

NÄIDE (unistamine on ju lubatud, eks):

```
10
Eesti
Lati
Leedu
Poola
Taani
Norra
Rootsi
Soome
Prantsusmaa
Inglismaa
12
Eesti-Lati 2:1
Lati-Leedu 0:1
Poola-Eesti 1:1
Poola-Eesti 0:0
Poola-Eesti 2:2
Taani-Eesti 3:2
Norra-Lati 2:0
Rootsi-Eesti 3:1
Soome-Lati 2:0
Prantsusmaa-Eesti 0:1
Inglismaa-Eesti 0:1
Prantsusmaa-Lati 0:1
```

Vastus:

```
Eesti
Rootsi
Norra
Soome
Taani
Leedu
Lati
Poola
Inglismaa
Prantsusmaa
```

3.18.3 Erdöse arvud

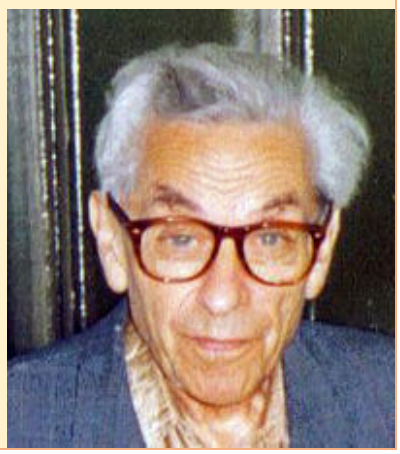
Ungari matemaatik Paul Erdös (pildil) oli kuulus selle poolest, et ta kirjutas oma elu jooksul palju teadusartikleid (üle 1500), tehes samas koostööd paljude teiste teadlastega. Selle põhjal on defineeritud Erdöse arvu mõiste. Nende teadlaste Erdöse arv, kes Erdösega koos midagi kirjutasid, on 1. Nende Erdöse arv, kes kirjutasid midagi koos Erdöse kaasautoritega, on 2 jne.
On antud hulk teadustöid koos autorite ninedega. Leida iga autori Erdöse arv. Kui autor pole Erdösega seotud, väljastada tema kohta -1.
Sisendi esimesel real on teadustööde arv `N (1<=N<=1000)`. Järgmisel `N` real on igaühel vastava teadustöö info: kõigepealt semikoolonitega eraldatud autorite nimed, siis töö pealkiri.
Väljundisse kirjutada tähestiku järjekorras autorid ja nende Erdöse arvud. Vältimaks kodeeringust tulenevaid probleeme, on Paul Erdös on alati esitatud kui Erdos, Paul. Paul Erdöst ennast pole vaja vastusesse kirjutada.
NB! Vastuses oodatakse, et nimed on harilikus leksikograafilises järjestuses, see tähendab näiteks, et Macaulay, Francis Sowerby on enne kui MacRobert, Thomas Murray.

NÄIDE:

```
10
Brown, Tom C.; Erdos, Paul; Freedman, A. R.; Quasi-progressions and descending
waves.
Li, Yong; Lipmaa, Helger; Pei, Dingyi; On delegatability of four designated
verifier signatures.
Mielikainen, Taneli; Ukkonen, Esko; The complexity of maximum matroid-greedoid
intersection and weighted greedoid maximization.
Brown, Tom C.; Pei, Dingyi; Shiue, Peter; Irrational sums.
Brazma, Alvis; Jonassen, Inge; Vilo, Jaak; Ukkonen, Esko; Pattern discovery in
biosequences
Targo Tennisberg; Katrin Gabrel; Võistlusprogrammeerimine
Bollobas, Bela; Erdos, Paul; Graphs of extremal weights.
Lipmaa, Helger; Mielikainen, Taneli; On private scalar product computation for
privacy-preserving data mining.
Bollobas, Bela; Das, Gautam; Gunopulos, Dimitrios; Mannila, Heikki; Time-series
similarity problems and well-separated geometric sets.
Mannila, Heikki; Ukkonen, Esko; A simple linear-time algorithm for in situ
merging.
```

Vastus:

```
Bollobas, Bela 1
Brazma, Alvis 4
Brown, Tom C. 1
Das, Gautam 2
Freedman, A. R. 1
Gunopulos, Dimitrios 2
Jonassen, Inge 4
Katrin Gabrel -1
Li, Yong 3
Lipmaa, Helger 3
Mannila, Heikki 2
Mielikainen, Taneli 4
Pei, Dingyi 2
Shiue, Peter 2
Targo Tennisberg -1
Ukkonen, Esko 3
Vilo, Jaak 4
```



3.18.4 Programmeerimisvõistlus

Programmeerimisvõistlusel on 1-100 osalejat ning 1-9 ülesannet. Osalejad saavad esitada lahendusi, mida loetakse kas õigeteks või valedeks. Tulemustes on osalejad järjestatud esmalt õigesti lahendatud ülesannete arvu järgi ja kui see on võrdne, siis kulunud aja järgi. Kui aeg on samuti võrdne, on osalejad oma numbrite järjekorras. Kulunud aega mõõdetakse vastavalt sellele, mitmendal võistluse minutil iga õige lahendus esitati. Kui võistleja esitas enne õiget lahendust samale ülesandele ka valesid lahendusi, liidetakse ajale iga vale esituse eest 20 minutit. Kui võistleja esitab valesid lahendusi pärast esimest õiget lahendust, siis nende eest midagi juurde ei liideta.
Sisendis on esimesel real esitatud lahenduste arv `N` ja järgmistel `N` real lahenduste logi ajalises järjestuses. Igal logi real on võistleja number, ülesande number, võistluse algusest kulunud aeg ning tulemus: õige (`C`) või vale (`I`).
Väljundisse kirjutada võistlejate paremusjärjestus vastavalt eespool kirjeldatud reeglitele. Iga võistleja kohta kirjutada tema number, lahendatud ülesannete arv ning kokku kulunud aeg.

NÄIDE:

```
1 2 9 I
3 1 11 C
1 3 19 I
1 2 21 C
1 1 25 C
```

Vastus:

```
1 2 66
3 1 11
```

Esimese võistleja lahendatud kolmas ülesanne ei mõjuta skoori, sest ta ei esitanud sellele ühtki õiget lahendust.

3.18.5 Pannkoogid

Vanaema teeb pannkooke ja asetab need taldrikule kuhja. Pannkoogid on natuke erineva suurusega, tähistame seda positiivse täisarvuga 1-100. Pannkoogikuhja on võimalik pannilabida abil osaliselt või täielikult ümber pöörata.
Olgu meil näiteks kuhi 8 4 6 7 5 2. Kui panna pannilabidas alt kolmanda koogi alla ja selle peale jäävad koogid ümber pöörata, saame kuhja 7 6 4 8 5 2. Kui järgmiseks teha pööre alates esimesest koogist, on kuhi pärast seda 2 5 8 4 6 7.
Sisendi esimesel real on pannkookide arv `N (1<=N<=30)`. Teisel real on pannkookide suurusi tähistavad N arvu. Eesmärgiks on sorteerida pannkoogid nii, et suuremad oleks allpool ja väiksemad peal. Väljundisse kirjutada selleks vajalikud pööramised.

NÄIDE:

```
5
5 4 3 2 1
```

Vastus:

```
1
```

NÄIDE:

```
5
5 1 2 4 3
```

Vastus:

```
1 4 2
```

3.18.6 Kaartide segamine

Tavalises kaardipakis on 52 mängukaarti. 19. sajandi teisel poolel elanud ja vesternitest tuntud kaardimängija Doc Holliday (pildil Val Kilmeri kehastatuna) oskas kaarte segada nii, et nad muutsid oma järjekorda spetsiifilisel viisil. Samas on tema käte liikumise järgi võimalik taibata, millist viisi ta parajasti kasutab. Leida, mis on kaartide lõplik järjekord eeldusel, et Doc Holliday alustab segamist uue pakiga, kus kaardid on sorteeritud.

Sisendi esimesel real on kaks täisarvu: võimalike segamisviiside arv `N (1<=N<=100)` ja segamiste arv `M (1<=M<=1000)`. Järgmisel `N` real on igaühel mingis arvud 1-52, mis näitavad, millisesse järjestusse esialgne pakk seda segamisviisi kasutades segatakse. Lõpuks tuleb `M` rida, millest igaühel on üks täisarv, mis näitab, mitmendat segamisviisi Doc Holliday järjest kasutab.

Väljundisse kirjutada arvud 1-52 sellises järjestuses nagu kaardid lõpuks on.

NÄIDE:

```
2 2
2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 51
52 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 1
1
2
```

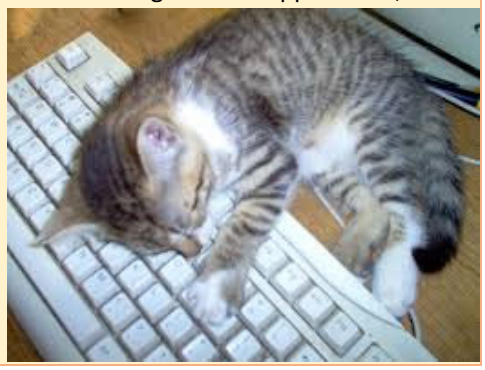
Vastus:

```
51 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 2
```

Esimene segamine vahetab omavahel esimesed kaks ja viimased kaks kaarti. Teine segamine vahetab esimese ja viimase kaardi.

3.18.7 Kass klaviatuuril

Juhan kirjutab arvutis teksti, aga ei vaata samal ajal ekraanile. Samal ajal istub laual kass, kes läheb aeg-ajalt `Home` ja `End` klahvide pihta, viies Juhani kursori vastavalt teksti algusse või lõppu. Leida, milline tekst niiviisi lõpuks välja tuleb.



Sisendis on täpselt üks tekstirida, mis koosneb tavalistest tähtedest ja tühikutest, see vastab Juhani kirjutatavale tekstile. Kohad, kus kass vajutas `Home`, on tähistatud märgiga `[` ja kohad, kus kass vajutas `End`, on tähistatud märgiga `]`. Väljundisse kirjutada tegelik tekst.

NÄIDE:

```
tere [maa]ilm
```

Vastus:

```
maatere ilm
```

3.18.8 Pinugrammid

Pinu andmestruktuuri saab kasutada anagrammide koostamiseks. Tähistame pinu operatsiooni *push* tähega `i` ning *pop* tähega o. Esialgsest sõnast saab anagrammi järgmiselt:
1. Lükkame kõik selle sõna tähed pinusse.
2. Pinust välja tõmmatud tähtedest koostame uue sõna.
Vastavalt operatsioonide järjekorrale võib saada erinevaid anagramme.
Näiteks esialgsest sõnast TROT võib saada sõna TORT kahel viisil:

```
i i i i o o o o
i o i i o o i o.
```

Sisendi kahel real on antud sõnade paar, mis moodustavad anagrammi. Leida kõik võimalused, kuidas kirjeldatud pinuoperatsioonide abil on võimalik esimene sõna teiseks muuta. Võimalused esitada leksikograafilises järjekorras.

NÄIDE:

```
madam
adamm
```

Vastus:

```
i i i i o o o i o o
i i i i o o o o i o
i i o i o i o i o o
i i o i o i o o i o
```

NÄIDE:

```
bahama
Bahama
```

Vastus:

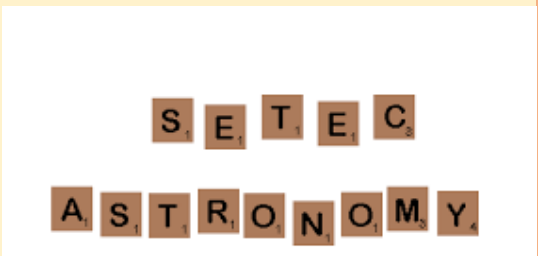
```
i o i i i o o i i o o o
i o i i i o o o i o i o
i o i o i o i i i o o o
i o i o i o i o i o i o
```

NÄIDE:

```
eric
rice
```

Vastus:

```
i i o i o i o o
```



3.18.9 Parv

Enamasti ületavad autod jõgesid sildade kaudu. Mõnes kohas pole siiski sildu ehitatud ja nende asemel kasutatakse parvesid. Kui ühtki autot ei ole, siis parv lihtsalt ootab. Kui autosid saabub, läheb parv vajadusel õigele kaldale, võtab auto(d) peale ning toimetab nad üle. Kui teisel pool on samal ajal autosid ootamas, võtab parv nad peale ja toob üle. Parv mahutab korraga N autot ja jõe ületamine võtab T minutit. Eeldame lihtsuse mõttes, et parve peale ja parvelt maha sõit aega ei võta. Alguses on parv jõe vasakul kaldal.

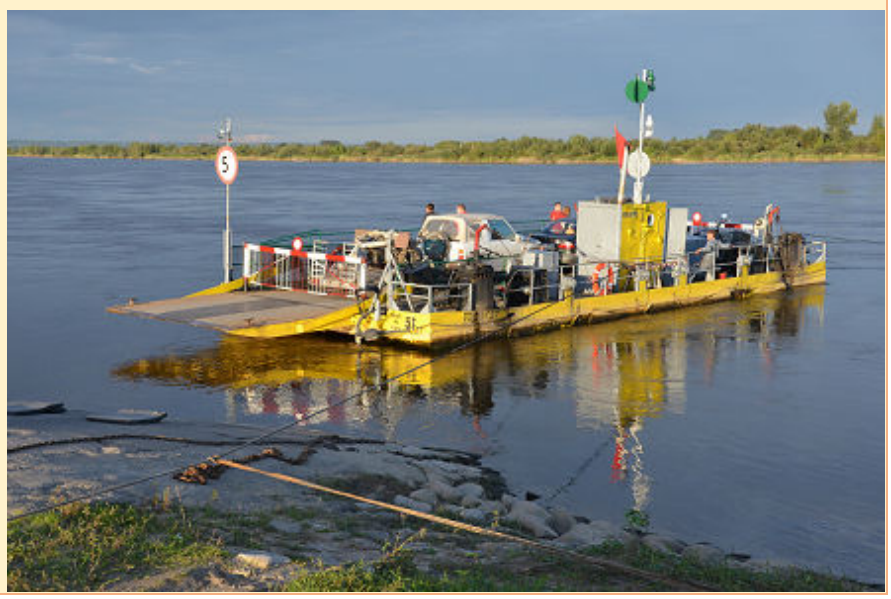
Sisendi esimesel real on kolm täisarvu: parvele korraga mahtuvate autode arv `N`, jõe ületamiseks kuluv aeg `T` ja autode koguarv `M`. Järgmistel `M` real on iga auto saabumise aeg ja kallas, kuhu ta saabub (`L` või `R`). Väljundisse kirjutada iga auto jaoks, millisel ajal ta vastaskaldale jõuab.

NÄIDE:

```
2 10 10
0 L
10 L
20 L
30 L
40 L
50 L
60 L
70 L
80 L
90 L
```

Vastus:

```
10
30
30
50
50
70
70
90
90
110
```



NÄIDE:

```
10 R
25 L
40 L
```

Vastus:

```
30
40
60
```

3.18.10 Ahelmurrud

Ahelmurruks nimetatakse avaldist kujul

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

Ahelmurdu saab lühendatult kirja panna kujul [a_0, a_1, a_2,…,a_n]. Näiteks järjend [2, 3, 1, 4] on võrdne murruga

$$2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{4}}} = \frac{43}{19}$$

Iga lihtmurdu on võimalik teisendada selliseks ahelmurruks. Antud on kaks positiivset täisarvu: lihtmurru lugeja ja nimetaja, leida sellele vastav ahelmurd ja väljastada see järjendiesituses.

NÄIDE:

```
43 19
```

Vastus:

```
2 3 1 4
```

NÄIDE 2:

```
1 2
```

Vastus:

```
0 2
```

3.19 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk3 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Lemming.cpp, Lemming.java, Lemming.py	Lemmingute ülesanne (pinu ja järjekord)
Kursused.cpp, Kursused.java, Kursused.py	Usina ülikooli ülesanne (kujutis)
Sortimine.cpp, Sortimine.java, Sortimine.py	Erinevad sorteerimismeetodid
Kaardid.cpp, Kaardid.java, Kaardid.py	Kaardipaki ülesanne (oma võrdlemine)