Algoritm ja algoritmide keerukus

Algoritm on protsess, millega täidetakse mingi ülesande. Iga arvutiprogramm tugineb mingile algoritmile. Et algoritmist kasu oleks, peab ta lahendama piisavalt üldise ja hästi kirjeldatud probleemi. Algoritm peab määrama täpselt kõik sisendi isendid ning kirjeldama vastavad väljundid. Algoritmi moodustavad sammud, mis muudavad sisendi isendi väljundiks. Mis on sisendi isendid? Näiteks sorteerimisel võib ollsa isendiks nii konkreetne arvude jada kui ka konkreetne stringide jada.

> **Algoritm** on lõplik käskude (ingl *instruction*) hulk, mis annab täpse operatsioonide järjestuse mingi probleemide klassi lahendamiseks. Arvutiteaduses on algoritm mingi meetod probleemi lahendamiseks *(*ingl *problem solving)*, mida saab realiseerida arvutiprogrammi abil.

Algoritm on ülesande lahendamiseks täpselt määratletud reeglite lõplik korrastatud kogum (Infotehnoloogia sõnastik EVS-ISO/IEC 2382-1:1998).

Tavapärane näide probleemist on sorteerimine:

Sisend: võtmete jada n võtmest $a_1 \ldots a_n$

Väljund: Ümberjärjestatud võtmed, nii et $a_1 \le a_2 \le \ldots \le a_{n-1} \le a_n$

Algoritm annab sammud, kuidas sisendist soovitud väljund / tulemus (näiteks võtmed mittekahanevas järjestuses) saadakse. Sorteerimise jaoks on võimalikke algoritme mitu. Enamus on piisavalt üldsed, et sorteerida nii arvulisi kui ka tekstilisi väärtuseid. Samuti peab sorteerimisalgoritm andma õige tulemuse, kui andmed on juba järjestatud, kui esinevad korduvad väärtused jne.

Algoritmi saab täpselt kirjeldada erinevaid vahendeid kasutades – visuaalselt graafilises keeles, pseudokeeles, programmeerimiskeeles kui ka inimkeeles. Oluline on, et kirjeldus oleks täpne ja üheselt mõistetav.

Algoritm peab olema määratud nii täpselt, et seda suudaks täita isegi arvuti. Kuna algoritm peab olema praktiliselt täidetav, siis lisandub veel üks nõue - täidetavaid samme peab olema lõplik arv (aga see arv võib olla üsna suur). Algoritm peab lahendama ülesande õigesti kõigi lubatud sisendandmete korral.

Kõige olulisemad algoritmi omadused on **tõhusus** (ingl *efficiency*) ja **õigsus** (ingl *correctness*). Lisaks pea hea algoritm olema kergesti realiseeritav. Need hea algoritmi omadused on tihti omavahel vastuolus - tõhusat algoritmi ei pruugi olla lihtne realiseerida.

> Algoritm on **õige** *(*ingl *correct)*, kui kõigi sisendite korral, mis vastavalt algoritmi kirjeldusele on lubatud, lõpetab ta töö ja annab tulemuse, mis rahuldab ülesande tingimusi. Öeldakse, et algoritm lahendab arvutusülesande.

Algoritmi efektiivsusele ehk tõhususele tuleb samuti tähelepanu pöörata. Algoritmi tõhususe all mõistetakse tihti seda, et algoritm peab töötama teatud aja piires. Algoritmi optimeerimine pole alati kerge ja seetõttu tuleb programmist üles leida see koht, mis tegelikuks pudelikaelaks on (tihti ca 10% koodist). Just viimase kallal tasub vaeva näha, mitte ülejäänud 90% koodi kallal. Kui algoritmi töötamise kiirus aga oluline pole, siis ei tasu ka tema kallal liigselt vaeva näha. Et erinevate lahenduste erinev efektiivsus nähtavaks muutuks pole tihti probleemi mahtu (enamasti algandmete hulka) väga suureks vaja ajadagi.

Algoritmi tööaeg ja keerukus

Tihti on sama tulemuse saavutamiseks võimalik rakendada mitut erinevat algoritmi ja tekib kindlasti küsimus, millist algoritmi eelistada. Kuni probleem on väike, ei ole otsuse tegemise alused olulised. Kui aga programm peab lahendama suure probleemi (või hästi palju keskmises mahus probleeme), siis tuleb hoolikalt mõtelda algoritmi peale, täpsemalt sellele, kui palju ta aega kulutab ja kui palju mälu vajab.

Probleemi suuruse all tuleks mõista ennekõike töödeldavate andmete hulka.

Juhul, kui algoritm/programm on ise väljatöötatud, tuleks seda suuta kriitilise pilguga hinnata – kas probleemi mahu (töödeldavate andmete hulga) suurenedes kasvab programmi tööaeg mõistlikes piirides. Otsuse langetamiseks tuleks algoritme mingil viisil hinnata. Parameetritest, mille järgi hinnata algoritmi headust (kasutatava ressursi suurust), on olulisemad kaks:

- vajatava mälu hulk;
- töötamise kiirus (tööks kuluv aeg).

Hoolimata arvutite arenemisest ja kiiremaks muutumisest võib tõsisemate majanduslike, teaduslike jm probleemide lahendamine neile endiselt üle jõu käia. Investeering uude arvutisse võib töökiirust tõsta 10 .. 100 korda, samal ajal kui efektiivsema algoritmi rakendamine suurte andmehulkade korral vähendab tööaega tuhandeid kordi. Igal juhul tuleb kasutada arvutiressurssi targalt ja kokkuhoidlikult.

Algoritmide hindamiseks tuleb uurida nende tööd ja luua selleks sobilik abstrakstsioon. Selleks kasutatakse **hüpoteetilist arvutimudelit** (ingl *hypothetical computer, Random Access Machine (RAM)*), mille omadused on järgmised:

- Kõik lihtoperatsioonid (+, -,*, if, =) loetakse võrdseteks ja tähendavad ühte sammu.
- Tsükkel ja funktsiooni töö ei ole lihtoperatsioonid, vaid need koosnevad mitmest sammust - 1000 arvu sortreerimine ei saa mitte kuidagi olla üks lihtoperatsioon ehk samm. Tsükli sammude arv tuleb määrata arvestades tsükli korduste arvu.
- Iga pöördumine mälu poole on samuti üks samm. Hüpoteetiline arvuti ei tee vahet, kas infoüksus on mälus või kettal.

Algoritmi hindamiseks arvutatakse välja tehtavate sammude arv. Ja ei lasta ennast häirida sellest, et ka tavalised tehted kulutavad aega erinevalt. Kui mudel ajada liiga keeruliseks, muutub tülikaks tema kasutamine.

Algoritmide analüüs *(Analysis of algorithms)* on informaatika haru, mille aineks on algoritmi oluliste karakteristikute määramine. Analüüsiga leitakse algoritmi arvutuslik keerukus, st leitakse tema aja-, mälu ja muu ressursivajadus, mis on tarvilik tema käivitamiseks.

Algoritmi tööaega ei saa lihtsalt sekundites fikseerida, sest:

a) arvutid töötavad erineva kiirusega;
b) töödeldavate andmete hulk on erinev;
c) tegelik tööaeg sõltub ka programmeerimiskeelest ja kompilaatori poolt tekitatud masinkoodi käskudest.

Mida rohkem on andmeid, seda rohkem tööd e. samme tuleb nende töötlemiseks teha.

Algoritmide keerukusprobleemidega tegeleb vastav teadus - **arvutuslik keerukusteooria** *(*ingl *computational complexity theory)*, mille loojateks on Juris Hartmanis ja Richard Stearns.

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

Algoritmi tööaega *(*ingl *running time)* mõõdetakse tehtavate sammude arvuga (hüpoteetilisel arvutil). Algoritmi analüüsimisel uuritakse algoritmi tööaega mingi algandmete hulga puhul. Leitakse funktsioon, mis iseloomustab algandmete hulga seotust sammude arvuga. Algoritmi tööaja hindamist nimetatakse ka algoritmi **keerukuse** (ingl *complexity*) hindamiseks. Keerukus määratakse funktsioonina algandmete hulgast N.

Sõltuvalt algandmete iseloomust saab määrata:

- **tööaja ehk keerukuse halvimal juhul** *(*ingl *worst-case running time / complexity)* – maksimaalne võimalik sammude arv, mis on vajalik tulemuse saavutamiseks N algandme korral. Selle teadmine annab kindluse, et enam hullemaks minna ei saa (näiteks otsimisel tehakse kõige rohkem samme siis, kui otsitav puudub).
- **tööaja ehk keerukuse parimal juhul** *(*ingl *best-case running time / complexity)* – minimaalne võimalik sammude arv tulemuse saavutamiseks N algandme korral. Selle näitajaga ei ole üldiselt midagi kasulikku pihta hakata, sest hinnang on liialt optimistlik (näiteks otsmisel on otsitav massiivis esimene).
- **tööaja ehk keerukuse keskmisel juhul** *(*ingl *average-case running time / complexity)* – keskmine sammude arv N algandme korral.

Ehkki halvima ja parima keerukuse vahe võib olla väga suur, arvestatakse siiski enim halvima juhuga. See garanteerib,

et hullemaks minna ei saa ning halbu või halbadele lähedasi juhte esineb kõige sagedamini.

Järgnevad koodiõigud lahendavad otsimisülesande: kas otsitav väärtus paikneb massiivis, kasutades selleks jada- ehk järjestikotsimist.

```
leitud = 0;
for (i = 0; i < MaxN; i++) {
    if (arvud[i] == otsitav) {
        leitud = 1;
    }
}
```

Koodinäide 1 Otsimine for-tsükliga

```
leitud = 0;
i = 0;
while (leitud != 1 && i < N) {
    if (arvud[i] == otsitav) {
        leitud = 1;
    }
    i++;
}
```

Koodinäide 2 Otsimine while-tsükliga

- Milles seisneb on algoritmide erinevus?
- Kas erinevate algandmete puhul on nende tööaeg erinev? Kui jah, siis kuidas see avaldub?
- Milline algandmete komplekt on halvim variant kummagi algoritmi jaoks ning milline on parim?

Algoritmi keerukuse hindamiseks tuleb loendada sammud, mida algoritm oma töötamise ajal teeb. Elementaaroperatsiooni ehk sammu määramisest oli juttu eespool hüpoteetilise arvuti kirjelduses.

Vaatame näitena lisamissorteerimist ehk pistemeetodit. Täpsemat selgitust algoritmi kohta saab lugeda sorteerimise materjalidest, kuid pseudokoodi uurides saab siiski üldise pildi. Iga lause kõrvale korduste arv ehk mitu korda lauset täidetakse. Mõnikord määratakse ka hüpoteetlised konstandid (c_1, c_2 jne), mis peaksid kirjeldama iga käsu erinevat aega. Kuid hüpoteetilises arvutis on üldistatud kõik sammud samale ajale. Eeldame, et sorteeritava massiivi suurus (sisendandmete hulk) on `n`. `While`-tsükli keha korduste arvu (mitu kohta arv massiivis ettepoole tõstetakse) on raske määrata, sest see sõltub otseselt andmetest. Vastavaks operatsioonide arvuks on kirjutatud `tj`.

```
Insertion_Sort(A)              mitu korda
for j = 2 to Length(A)         n
    k = A[j]                   n-1
    i = j - 1                  n-1
    while i > 0 and A[i] > k   j=2 to n sum(tj)
        A[i+1] = A[i]          j=2 to n sum(tj-1)
        i = i - 1              j=2 to n sum(tj-1)
    A[i+1] = k                 n-1
```

Halvimal juhul (massiiv on järjestatud kahanevalt) tähendab `tj` elemendi `j` jaoks tema võrdlemist elementidega `A[1]` kuni `A[j-1]`, millest saab leida sammude arvu aritmeetilise jada summa valemeid kasutades:

`j` elemendi töötlemiseks: `n*(n+1)/2-1`

`j-1` elemendi töötlemiseks: `n*(n-1)/2`

Valem algoritmi tööaja arvutamiseks halvimal juhul sõltub sorteeritavate elementide arvust `n` ja on järgmine:

`T(n)=`

`=n+(n-1)+(n-1)+(n(n+1)/2-1)+(n(n-1)/2)+(n(n-1)/2)+(n-1)=`

$=C_1*\mathbf{n^2} + C_2*\mathbf{n} - C_3$

Teisendussamme on vahel loomulikult tunduvalt rohkem, aga tulemuseks saadud funktsioon `T(n)` on ruutfunktsioon, nagu $T(n)=an^2+bn+c$,

kus `a`, `b` ja `c` on määratud erinevate sammude korduste arvudega.

Toodud hinnang on ligikaudne (me ei tea midagi täpsemat `ci`-de kohta), et ei tee paha lihtsustamiseks seda veelgi ligikaudsemaks muuta. Sel juhul jääb alles vaid `ni` kõige kõrgem aste (antud juhul n^2), mis määrabki algoritmi järgu. Arvesse ei võeta ka kordajaid, mis konkreetsest arvutist sõltuvad.

Asümptootiline hinnang algoritmi efektiivsusele / keerukusele

Me võime rääkida algorimi töökiirusest ning selle seosest probleemi mõõduga `N` (andmete hulgaga). St on erinevad funktsioonid `T(N)`, mis algoritmi töökiirust iseloomustavad (`N`, N^2, `log N` jne).

Probleemid programmi tööajaga tekivad tavaliselt siis, kui töödelda tuleb suuri andmehulki. Seega on vaja uurida justnimelt seda, kuidas algoritm (programm) hakkab käituma siis, kui andmete hulk on suur ja kasvab veelgi. Sellist tegevust nimetatakse algoritmi **asümptootiline keerukuse** *(*ingl *asymptotic complexity)* hindamiseks. Uuritakse ja analüüsitakse algoritmi **keerukusfunktsiooni kasvukiirust** *(*ingl *growth)*. Sisuliselt näitab programmi aluseks oleva algoritmi keerukusfunktsiooni kasvukiirus, kuidas muutub (kasvab) programmi tööaeg (või ka mõne muu ressursi vajadus), kui töödeldavate andmete hulk kasvab

Funktsiooni käitumise asümptootiline hinnang on tegelikult matemaatikasse kuuluv mõiste ja sellega väljendatakse funktsiooni väärtuse muutumise üldist trendi funktsiooni argumendi lähenemisel mingile konkreetsele väärtusele või lõpmatusele, ja sellega kaasnevaid funktsiooni väärtuse piire. Hinnanguid ja vastavaid tähistusi on mitu. Meid huvitab funktsiooni kasvamise **asümptootiline ülemine piir** *(*ingl *asymptotic upper bound).* Seda tähistatakse suure O-tähisega (nn *Big O notation, order-of magnitude*). Hinnang on ligikaudne ja seotakse funktsioonis esineva kõige kõrgema astmega. Hinnang kehtib `N`-i piisavalt suurte väärtuste korral.

Ametlik definitsioon asümptootilisele ülemisele piirile (Big Oh) on järgmine:

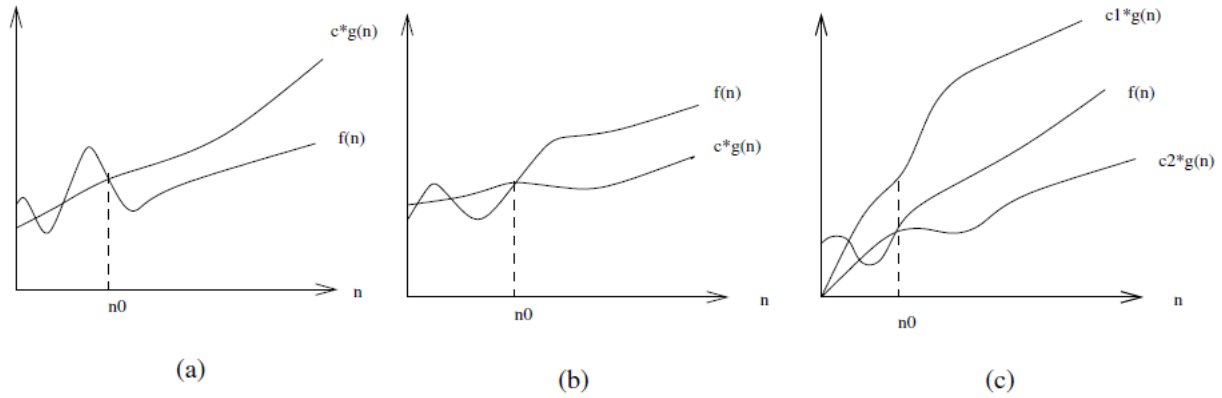
Funktsioon `f(N) = O(g(N))` tähendab, et `c*g(N)` on `f(N)` väärtuste ülemine piir. On olemas konstandid `C` ja N_0, nii et `f(N) < c*g(N)` kõigi `N` jaoks, kui `N` > N_0

St mängu võetakse veel üks konstant `C`, mille abil

- püütakse kaotada sammude väljaarvutamisel tekkivad vead (näiteks `N`-i madalamate astmete kaotamise läbi)
- püütakse likvideerida vead, mis tekivad algoritmi analüüsides ebaoluliste lausete vahelejätmise tõttu
- võimaldatakse klassifitseerida algoritmid tööaja ülemise piiri järgi keerukusklassidesse.

N_0 väljendab seda, et algoritmi analüüsimisel ei paku huvi olukorrad, kus andmete hulk on väike: meid ei huvita, kuidas läheb algoritmil 10 arvu sorteerimisega, kui oluline on 10 000 arvu sorteerimine.

Lisaks kohtab kirjanduses funktsiooni `Ω(N)`, mis kirjeldab funktsiooni väärtuse alumist piiri. Ning `Θ(N)`, mis kirjeldab nii ülemist kui alumist piiri.



Joonis 1: Funktsioonid Big O (a), Ω (b) ja Θ (c) (S. Skiena, Algorithm Design Manual)

Algoritmide põhilised keerukusklassid

Et algoritmi tööaega hinnata, on vaja peamist tähelepanu pöörata kasutatavatele keelekonstruktsioonidele – st algoritmi või programmi struktuurile. Paljude nö tunnustatud algoritmide kohta on nende keerukusklassid määratud ja nendest lähtudes saab hinnata algoritmi sobivust konkreetsetes olukordades.

Järgnevas loetelus on peamised ja üksteisest rohkem eristuvad keerukusklassid, kuid toodud loetelu ei ole lõplik. Kokkuleppeliselt peetakse logaritmi all silmas kahendlogaritmi ehkki suurusjärgu mõttes ei ole olulist vahet, millisel alusel logaritmi leida. Tähistustes võib kohata kirjaviisi `log` kui ka `lg`.

`O(1)` – **konstantne keerukus**. Algoritmi tööaeg ei sõltu andmete hulgast. Samamoodi ei sõltu tööaja kasv ka andmehulga kasvust. Kõiki programmi lauseid täidetakse üks kord. Lahendamiseks on tihti olemas valem.

`O(log n)` – **logaritmiline keerukus.** Algoritmi tööaeg suureneb andmehulga kasvades üpris aeglaselt. Näiteks kui andmehulk `N` kasvab `N` korda (N^2), siis `lg N` kasvab vaid 2 korda. Ülesande lahendamisel jõutakse tulemuseni samm haaval liikudes samal ajal andmehulka pidevalt vähendades ehk probleemi mõõt väheneb pidevalt kordades. Tüüpiliseks näiteks on kahendotsimine, kus iga järgmise võrdlemisega (sammuga) väheneb läbivaatamist vajav andmehulk kaks korda.

`O(N)` – **lineaarne keerukus**. Algoritmis on vaja iga elementi natuke töödelda (tüüpiliselt ühe korra, kuid korduste arv võib olla ka suurem). Sammude arv on N. Andmehulga suurenemisel kasvab lineaarselt ka algoritmi tööaeg: kui andmehulk kasvab 2 korda, kasvab ka tööaeg 2 korda. Näited: lineaarne otsimine, maksimumi ja miinimumi leidmine, keskmise arvutamine.

`O(n*log n)` – **linearitmiline keerukus**. Kui see õnnestub saavutada ruutkeerukuse asemel `O(N²)` asemel, on võit juba väga suur. Silmaga peale vaadates pole alati lihtne tuvastada, kas selline keerukus on saavutatud. Tavaliselt tähendab seda, et lineaarses algoritmis `O(N)` on `O(log n)` algoritm. Algoritm lahendab probleemi seda kõigepealt väiksemateks osadeks tükeldades (nagu logaritmilise keerukuse korral), kõik nn alamprobleemid ära lahendades ning pärast kogu lahenduse kokku saamiseks alamprobleemide lahendused ühendades. Näited: sorteerimine kuhjaga (ingl *heapsort*), kiirsorteerimine (ingl *quicksort*) ja mestimisega sorteerimine (ingl *merge sort*).

`O(N²)` – **ruutkeerukus**. Sellises algoritmis vaadatakse tihti läbi kõik paarid, mis saavad N elemendist moodustuda. Materjalis varem käsitletus lisamissorteerimine kuulub nimelt sellesse keerukusklassi. Võib arvestada nii, et andmehulga kasvamisel N korda suureneb tööaeg N^2 korda. Või teistpidi vaadatuna – N elemendi töötlemiseks on vaja teha N^2 sammu. Enamasti on selles algoritmis 2 tsüklit üksteise sees ja mõlemad sõltuvad algandmete hulgast. Lisaks nimetatud lisamissorteerimisele kuuluvad siia veel mitmed lihtsamad sorteerimisalgoritmid, aga ka mitmed graafialgoritmid.

`O(N³)` – **kuupkeerukus**. Tüüpiliselt on sellises algoritmis 3 tsüklit üksteise sees, mis kõik sõltuvad algandmete hulgast. Sobilik vaid väikeste andmehulkade korral. Näiteks maatriksite korrutamine.

`O(2^N)` – **eksponentsiaalne keerukus**. Väike arvutus ütleb, et sellise klassi algoritmis tehakse N elemendi töötlemiseks palju tööd ning andmete hulga suurenemisel on aja kasv märgatav. Näiteks kui N=10 on töötlussammude arv suurusjärgus 1000, andmehulga kasvamisel 2 korda kasvab töötlussammude arv 1 000 000-ni. Sama juhtub algoritmi tööajaga. Ebapraktiline algoritm. Sellised on tihti jõumeetodil lahendused, parima tulemuse leidmiseks kõigi variantide täisläbivaatused. Kasutatav harva ja väikese andmemahu korral.

`O(N!)` – **faktoriaalne keerukus**. Selline funktsioon tekib näiteks kõigi permutatsioonide leidmisel

Tee vahet järgneval

Algoritmi alusel koostatud programmi tegelik tööaeg ja efektiivsus sõltub andmete hulgast, protsessori kiirusest, arvutist, kompilaatorist, ...

Algorimi keerukus on hinnang sellele, kuidas algorimi poolt esitatavad nõudmised ajale muutuvad näiteks siis, kui probleemi mõõt kasvab.

Keerukus mõjutab jõudlust, mõjutab tööaega.

Kui andmemaht kasvab, jääb algoritmi keerukus ikka samaks, ehkki tööaeg kasvab.