Algoritmide koostamise strateegiad

> **Algoritmide koostamise strateegiad** *(algorithmic paradigmas)* on üldised põhimõtted sellest, kuidas konstrueerida tulemuslikke algoritme probleemide lahendamiseks.

Miks on kasulik selliste strateegiate väljatöötamine?
- Strateegiad annavad lahendusmudelid erinevate ülesandeklasside lahendamiseks. Kui on selge, millisesse klassi ülesanne kuulub, saab hakata sobiva mudeli suunas töötama.
- Strateegiad on sellised, et nendele tuginedes koostatud algoritmid on tõlgitavad tavalistesse (harjumuslike juhtstruktuuride ja andmetüüpidega) programmeerimiskeeltesse.
- Algoritmide ajutisi ja lõplikke nõudmisi nii tööaja kui ka vajatava mälu osas ning oodatavaid tulemusi on võimalik ligikaudselt hinnata.

Valik erinevaid strateegiad:
- Jõumeetod *(Brute-force)*
- Jaga ja valitse *(Divide and Conquer)*
- Dünaamiline programmeerimine *(Dynamic Programming)*
- Ahne algoritm *(Greedy Method)*
- Tagurdusmeetod *(Backtracking)*

Tavaliselt saab ühe probleemi lahendamiseks koostada algoritmi erinevaid meetode kasutades, siiski on üks lahendus efektiivsem kui teine. Õige paradigma valik on algoritmi koostamisel väga oluline. Reeglina on lihtsam lahendus, mis probleemi vaadates kohe pähe kargab, programmi lugedes kergemini äratuntav on ja mida ka kergem programmeerida tundub, ajaliselt keerukam. Sellegipoolest ei tule robustsemaid meetodeid aia taha visata, sest väikeste andmehulkade puhul on nad piisavalt head ja ära tasub algoritmi läbipaistvus ja selgus ning sellega kaasnev kiirem teostamise aeg ja eeldatavasti väiksem vigade hulk.

Lahendamine jõumeetodil (Brute-force)

Jõumeetodil algoritm leiab lahenduse ebaefektiivselt, tavaliselt vaadates selleks läbi kõikvõimalikud lahendused ja teed. Selline algoritm on hästi arusaadav ja kergesti välja mõeldav, kuid nõuab paljude sammude sooritamist. Sõltub lähteandmete iseloomust, hulgast ja sellest, mida otsitakse, kas selline meetod on sobiv. Proovi läbi kõik võimalused ja vali siis välja parim! Hea öelda, tihti võimatu täita, sest keerukusklass võib kerkida O(N!)-ni.

Ehkki on tegemist enamasti aeglase meetodiga, on tal ka mõned **eelised**:
Jõumeetodil lahenduse uurimine viib tavaliselt probleemist parema arusaamise juurde. Teda võib ka mõista kui mõtlemise strateegiat. Väikese algandmete hulga juures võib sellist lahendust paberil läbi mängida ja muutub probleem arusaadavamaks.

Programmeerimisprojekti kulud koosnevad paljudest osadest ja programmi efektiivsus on vaid üks nendest. Programmeerija aeg on masina omast tavaliselt kallim. See ei tähenda, et ajalisele keerukusel mõelda ei tule, kuid kohad kus programme kasutatakse, on väga erinevad. Jõumeetodil töötavad algoritmid on lihtsaimad, paremini arusaadavd, kergamini realiseeritavad ja veakindlamad. Kavalama algoritmi ülesleidmine ja programmi testimine võib võtta palju rohkem aega.

Need algoritmid on kergemini kohandatavad. Neid on lihtsam modifitseerida veidi teistsuguste olude jaoks kui elegantsemad ja keerukamad algoritmid.

Näide: Leida arvu 625 kõik tegurid. Kuidas lahendada? Alustades 1-st ja lõpetades 625 jagada arv läbi kõigi arvudega. Kui arv jagub (jääk on 0), siis on järgmine tegur leitud. Kas on võimalik rakendada mõnda teist algortimi, mis on ehk küll keerulisem kirjeldada, kuid täidetavate sammude arv on väiksem? Gaussi arvude 1..100 liitmine. Milline on lahendus jõumeetodil? Milline oli Gaussi lahendus? Millist lahendust on lihtsam modifitseerida näiteks selleks, et leida arvude 1..100 ruutue summa?

Jaga ja valitse (Divide and Conquer)

Meetodi selgitus on järgmine: probleem jagatakse mitmeks alamprobleemiks (mis on oma mõõdult väiksemad lähteprobleemist). Alamprobleemid lahendatakse üksteisest sõltumatult, seejärel ühendatakse alamprobleemide lahendused nö alt üles ja saadakse lahendus kogu probleemile. Kuid millise meetodiga lahendatakse alamprobleemid, on tegelikult oluline küsimus, sest sellest sõltub kogu lahenduse efektiivsus.
Jaga ja valitse algoritme iseloomustatakse järgmiste **omaduste** abil:
- **Minimaalne sisendi mõõt** n0 - kui probleemi suurus on alla selle ei hakata probleemi jagama.
- **Alamprobleemi suurus**, milleks kogu probleem jaotatakse – milline suurus on paras?
- Jagamisel saadavate **alamprobleemide arv** – liiga palju alamprobleeme pole ka hea.
- **Algoritm**, mida kasutatakse alamprobleemide **lahenduste ühendamiseks**.

Seega on mingi probleemi suurus `n0`, millest väiksemat probleemi jagada mõtet ei ole ja on mingi optimaalne suurus `n/k`, milleks probleem jaotada. Tavaliselt ehitatakse seda tüüpi algoritm üles rekursiivselt, sest kogu meetod on oma olemuselt rekursiivne (jaga ülesanne tükkideks ja need tükid omakorda tükkideks kuni saadakse sobiva suurusega tükid, mida on paras lahendada).

Jaga-ja-valitse tüüpi protseduur näeb reeglina välja järgmine:

```
procedure D-and-C (n : sisendi suurus);
   begin
      if n < = n0 then
         Lahenda probleem jaotamata
       else
          Jaota probleemi mõõt r tükiks, igaüks suurusega n/k;
          Iga tükiga (r korda) tee
            D-and-C (n/k);
       Ühenda r lahendust, et saada kogu lahendus;
       end if;
    end D-and-C;
```

Algoritmi hindamine
Aega võtavad nii rekursiivsed väljakutsed, kui ka iga osa töötlemine. Oletame, et probleem suurusega n jaotatakse b tükiks, millest igaühe suurus on n/k. Oletame, et jaotamine võtab aega D(n) ja saadud vastuste ühendamine C(n) ühikut. Sel juhul saame seose probleemi suurusega n lahendamiseks halvimal juhul:
$T(n)=rT(n/k)+D(n)+C(n)$

Sellist tüüpi algoritmi **näidetena** võib meelde tuletada kahte sorteerimisalgoritmi – need olid **kiirsorteerimine** *(quicksort)* ja **mestimisega sorteerimine** *(mergesort)*. Mõlemad algoritmid on rekursiivsed ja jaotavad mingi skeemi järgi kogu ülesannet tükkideks, et tükid sorteerida ja pärast ühendavad osad. Mestimisega sorteerimine tundub selle mõttes isegi tüüpilisem. Jaga ja valitse põhimõttest lähtub ka kahendotsimise algoritm ning otsimiskahendpuu peal töötavad algoritmid.

Dünaamiline programmeerimine *(Dynamic Programming)*

Programmide koostamisel on üsna tavaline eesmärk parima otsuse tegemine. Näiteks millist teed mööda minna punktist A punkti B, et läbitav distants oleks kõige lühem. Kuidas anda raha tagasi nii, et kasutataks võimalikult vähe rahatähti, kuid lõpmata palju neid võta ei ole jne. Parim lahendus võib olla nii miinimumi kui ka maksimumi leidmine.

Dünaamilist programmeerimist kasutatakse siis, kui otsitav vastus koosneb osadest, mis on omakorda on optimaalseteks lahendusteks alamprobleemile.

> **Dünaamilist programmeerimist** sobib kasutada siis, kui lahendamisel tuleb korduvalt ette sama või samasuguse alamülesande lahendamine. Leitud lahendused peetakse meeles, juhuks kui neid uuesti vaja läheb (pole vaja uuesti leidma hakata).

Jaga-ja-valitse printsiibi puhul leitaks rekursiivsete pöördumiste korral samad suurused korduvalt. Tüüpiliste DP ülesannete puhul on võimalikke lahendusi palju, kuid leida on vaja, milline või millised on parim(ad). Saab leida kõik variandid ja seejärel valitakse välja parim variant.
Kuni komponente, millest vastuste kombinatsioonid moodustatakse, on vähe, saab hakkama kõigi võimaluste järjestikuse leidmisega. Kuid komponentide hulga kasvades muutub probleem kiiresti nii suureks, et nimetatud taktika ei sobi. Tegemist on eksponentsiaalse keerukusega ülesande lahendusega ja viimast tuli teatavasti vältida. Kõigi kombinatsioonide arvutamisel ei arvestata näiteks sellega, et ebasobivate variantide kujunemist on juba ette võimalik ära tunda või ei osata kasutada luba leitud alamprobleemide lahendusi.

Dünaamiliseks programmerimiseks sobivad probleemid saab ära tunda järgmiselt:

- Probleemi saab jagada **järkudeks** *(stages)*, kus igas järgus nõutakse otsuse tegemist. Otsus on näiteks: kuhu minna järgmisena (lühim tee).
- Igas järgus on mitu **olekut** *(state)*, näiteks punkt/koht, kuhu selleks hetkeks (järguks) jõutud on.
- Iga otsus antud järgus ja antud olekus viib süsteemi järgmise järgu mingisse olekusse. Otsus kuhu edasi kavatsetakse minna määrab, millistesse punktidesse järgmises olekus jõuda on võimalik.
- Antud olekus tehtud optimaalne otsus kõigi järgmiste olekute kohta ei sõltu eelmistest otsustest. St kui antud punktini on jõutud kõigist võimalikest variantidest optimaalseimat teed pidid, jätkatakse ajaloole tähelepanu pööramata (ajalugu loomulikult peetakse meeles, kuid see ei mõjuta järgmise otsuse tegemist).
- Viimane järk peab lahenduma iseenesest.

Kõige raskem kirjeldatud taktika juures ongi õigete järkude ja seisundite üles leimine. Edasine pole enam keeruline.

Dünaamiline programmeerimine on ka matemaatiline mõiste (ja pole tingimata seotud programmi kirjutamisega), kuid annab

sammud probleemi lahendamiseks ja selles mõttes on ta ikkagi lähedane arvutite programmeerimisele. Sellise mõtteviivi loojaks peetakse ameerika matemaatikut *Richard Bellman*'i, kes kirjeldas, kuidas lahendada probleemi, kus tuleb vastu võtta üks otsus teise järel. Aasta siis oli 1955.

Richard Bellmanni poolt on sõnastatud ka nn **optimaalsuse printsiip** *(Principle of Optimality*):
Optimaalsel tegevusel on omadus, et ükskõik, milline on algseisund ja alguses tehtud otsused, tuleb järgmised optimaalsed otsused teha arvestades hetkeseisundit.

Seega DP püüab ühe korraga meeles pidada erinevaid poolikuid lahendusvariante, millisteni on antud hetkeks jõutud. Tavaliselt kasutatakse vahetulemuste meelespidamiseks tabelit, millest lõpuks on vastus väljaloetav või kombineeritav. DP on **alt-üles tehnika** *(Bottom-Up Technique)*, kus lahendatakse kõige väiksemad alamprobleemid ja nendelt liigutakse samm haaval kõrgemale.

Tüüpiline **DP lahendus** probleemile **koosneb järgmistest osades**t:
1. **Moodustatakse tabel** kõigi alamülesannete võimalike vastuste jaoks
2. Algoritmi alguses **initsialiseeritakse tabelis** need elemendid, mis vastavad kõige väiksematele alamülesannetele või **tühjendatakse kogu tabel** (kasutades mingit arvu, millest saab järeldada, et antud väärtus on veel kasutamata).
3. **Elemendid tabelis täidetakse** kindlas järjekorras (vastavuses kasvava alamülesande mahuga) ja kasutades vaid neid elemente, mis on juba leitud. Kui antud element on juba täidetud, siis teda reeglina (on erandeid) muutma ei hakata, sest selle alamülesande jaoks on lahendus käes. Tabeli pidamist nimetatakse inglise keeles *memoization*.
4. Iga element arvutatakse vaid kord.
5. Viimane arv, mis leitakse on ka **optimaalne vastus** kogu ülesandele või peale antud arvuni jõudmist saab tabelit kasutades konstrueerida vastuse (kui vastus koosneb mitmest komponendist ehk on vaja näidata, kuidas optimaalne lahendus saadakse).
6. **Realisatsioon on** alati **iteratiivne**, mitte aga rekursiivne, ehkki ülesanne võib oma olemuselt rekursiivne tunduda.

Algoritmi hindamine
Kasutades DP-st saadakse tavaliselt hakkama halvimal juhul ruutkeerukusega, paremal juhul on keerukus $O(n \log n)$. Samal ajal analoogiliste ülesannete lahendus kõigi variantide leidmisel viib eksponentsiaalse keerukuseni.

Ahne algoritm (Greedy Method)

Ka see algoritmitüüp on sobiv optimiseerimisülesannete lahendamiseks ja paljudel juhtudel on teda kergem koostada ja ta töötab kiiremini kui DP algoritm. Ahne algoritm ei anna alati vastuseks optimaalset tulemust ja kui tulemus on ka optimaalne, on seda väga raske tõestada. Kui tulemus pole parim, võib see olla aluseks edasisel töötlusel.

Milline on olukord, kui hakatakse optimiseerima ja peamised osalejad ning funktsioonid selles?
- On olemas **kandidaatide hulk** (graafi tipud, teede pikkused, rahatähtede suurused ...)
- On **valitute hulk**, mis või kes on juba **kasutatud** (sobivaks tunnistatud, tagasi antud rahatähed, läbitud graafi tipud, ...)
- Mingi **eeldatav lahendus** *(solution)*, otsitav summa vms, mille järgi saab otsustada, kas välja valitud kandidaadid moodustavad lahendused (ei pruugi olla optimaalne)
- **Jätkamise näitaja** *(feasible)*, mille järgi saab otsustada, kas kandidaatide hulka saab suurendada, et lahendust leida.
- **Valikufunktsioon** *(select)*, mille abil valitakse uusi kandidaate väljavalitute hulka
- **Vastusefunktsioon**, mis annab lõpliku väärtuse lahendusele

Optimeerimisülesanne otsib kõigi kandidaatide hulgast mingit alamhulka (valitute hulka), mis rahuldaks teatud tingimusi. Tingimuseks on enamasti mingi maksimaalse või minimaalse väärtuse leidmine ja sellele vastavalt on ka tehtud valikufunktsioon.

Ahne algoritmi alusel lahendatud ülesanded näevad üsna ühte tüüpi välja (NB! See EI OLE Pascalis vms keeles!):

```
{C – kõik kandidaadid
 S – väljavalitute seltskond
 X – üksik kandidaat}
function select (C : candidate_set) return candidate;
function solution (S : candidate_set) return
boolean; {Kas on lahendus?}
function feasible (S : candidate_set) return
boolean;{Kas saab veel lisada?}

function greedy (C : candidate_set) return candidate_set
x : candidate;
```

```
S : candidate_set;
begin
   S := {};
   while (not solution(S)) and C <> {} do
     x := select(C);
     C := C - x; {eemaldame X kandidaatide hulgast}
{Kui X sobib hulka S, siis paneme ta sinna.}
     if feasible(S <= x) then
        S := S + { x };
     end if;
   end while;
   if solution(S) then
     return S;
   else
     return no solution;
   end if;
end greedy;
```

Mõne ülesandeklassi jaoks annab ahne algoritm õige (optimaalse) vastuse, teiste ülesandeklasside juures ei anna (kuid võib siiski sobida, kui seda kõige optimaalsemat lahendust polegi tegelikult vaja leida).
Ahne algoritm sobib kindlasti siis, kui igal sammul tehtav lokaalselt optimaalne valik viib globaalselt optimaalse tulemuseni. Ehk teiste sõnadega: kui alamülesannete optimaalsed lahendused annavad tulemuseks kogu probleemi optimaalse lahenduse, on sobiv kasutada ahnet algoritmi.

Seljakoti probleem *(knapsack problem)*:
Tegemist on klassikalise probleemiga, mida tihti näitena kasutatakse. Varas on laos. Tal on seljakott, mis suudab kanda esemeid teatud raskuse piirides (või on varga enda võimed piiratud teatud kilodega). Igal esemel on hind ja loomulikult ka raskus. Varga eesmärk on võtta kaasa kraami võimalikult suure summa eest.

Ülesandel on kaks variatsiooni:
1. **Diskreetne e täisarvuline** *(0-1 knapsack problem)* – asju ei ole võimalik tükeldada, vaid nad on terved. Asjad kaaluvad täisarv kilosid ning seljakotiga saab varas ära viia ka täisarv kilodes kraami.
2. **Pidev e murdarvuline** *(fractional knapsack problem)* – kraami saab panna seljakotti osade kaupa, pole täisarvulist nõuet (ilmselt on tegemist kaalukaubaga, näiteks kullaliivaga).

Ahne algoritm osutub sobivaks teise variandi puhul. Saab leida eseme hinna kaaluühiku kohta ja vastavalt sellele lisada kõige kallimat ja vähem kaaluvat kaupa kõigepealt ja edasi veidi vähem kasulikke kaupu just nii palju, kui mahub.
Esimene variant vajab dünaamilist programmeerimist. Ei ole raske konstrueerida lähteandmeid, kus kõige kergema ja kallima kauba maksimaalne ärakasutamine ei vii tegelikult optimaalse lahenduseni, sest seljakott jääb osaliselt täitmata ja varga jõud kasutamata.

Sama probleemi saab kohandada arvude hulgale. On arvude hulk A, milles igal arvul on väärtus ja kogus. On suurus K, mis tuleb hulga A arvude väärtustest tekitada (arvude summa ei tohi ületada K-d, kuid ei leidu enam ühtegi vaba arvu, mida valitud arvude hulka lisada saaks, ilma et K lõhki läheks). Arvude kogus selles summas peab olema maksimaalne (või ka minimaalne, sõltuvalt kehtestatud kriteeriumist) – nii palju (või nii vähe) arve kui võimalik.
Sidudes ahne algoritmi optimeerimise tunnustega (vt eespool) saame:
- Hulka A kuuluvad arvud on kandidaadid
- A alamhulk B on lahendus, kui kogu B suurus mahub antud mõõtmesse K ja pole rohkem ühtegi arvu, mida võiks hulka B lisada.
- Objektiivne funktsioon, mida tuleb maksimeerida (või minimeerida), on arvude koguarv.

Juba tuntud algoritmidest, mis kvalifitseerub ka ahneks ja mille lahendus on täpne, mitte ligikaudne, on valiksorteerimise algoritm (igal sammul otsitakse vähimat arvu, mida sorteerimata massiiviosa algusesse tõsta).

Tagurdusmeetod (Backtracking)

Tagurdusmeetodit sobib kasutada siis, kui oodatavaks vastuseks on hulk (vektor) X1 ... Xn ja kus Xi kuulub mingisse lõplikku hulka S. Vektorisse valitakse liikmed selliselt, et teatud kriteeriumiks olev funktsioon maksimeeritakse, minimeeritakse (sisuliselt optimeerimine) või rahuldab mõnda muud kavalat tingimust. On ka võimalik, et tuleb leida kõik vektorid, mis kriteeriumile vastavad.

Klassikaline näide selle meetodi selgitamiseks on **8 lipu ülesanne** *(eight queen problem)*. Kuidas paigutada malelauale 8 lippu selliselt, et nad ühte lubatud käiku kasutades üksteist maha lüüa ei saaks.

Tagurdusmeetodile vastav lahendus on selline, kus tehakse niikaua samme, kuni saab, kui edasi enam minna ei saa, liigutakse tagasi sellisesse positsiooni, kust saab valida uue liikumistee või komplekteerimisvariandi ja proovitakse siis teist teed pidi liikuda. Lippude puhul paigutatakse lippe lauale mingi skeemi järgi seni, kuni võimalik. Kui järgmist lippu enam paika panna ei saa, võetakse viimane lipp oma kohalt ära, leitakse talle uus koht ja proovitakse edasi.

Vektorisse valitakse hulgast S väärtuseid seni, kuni nad sobivad. Kui järgmise elemendi lisamisel vektorisse enam vajalik kriteerium täidetud ei ole, võetakse sealt viimane element välja ja proovitakse teisi lisada. Selline tegevus võib kesta seni, kuni kõik variandid on läbi vaadatud. Halvim olukord on ilmselt juhul, kui vastust ei leita.

Näideteks kvalifitseeruvad ka labürintide läbimised, kui liigutakse nii sügavale ühe skeemi järgi kui saab ja siis hakatakse tagasi tulle teisi võimalusi proovima. Sellist strateegiat kutsutakse ka **sügavuti minevaks algoritmiks** *(depth first).* Sisuliselt on tegemist variantide täisläbivaatusega ja seda sorti algoritmid on kõige edukamalt realiseeritavad rekursiooni kasutades.