581378-4 Algoritmisen tietojenkäsittelyn perusteet (2 ov)

Luentorunko, syyslukukausi 2000

Matti Nykänen
Helsingin yliopisto
Tietojenkäsittelytieteen laitos

(Perustuu Pekka Kilpeläisen luentoihin syyslukukaudella 1999.)

Alustava sisältö

"kertokaa tulevat tapahtumat" [Jesaja 41:23]*

1. Johdanto: Kurssin sisältö ja hallinto; Laskentaongelmat ja algoritmit
2. Algoritmien suunnittelumenetelmistä
3. Algoritmien oikeellisuus
4. Tehokkuusanalyysin perusteita
5. Tehokkaasti ratkeavat ja laskennallisesti vaativat ongelmat
6. Ei-ratkeavat ja puoliratkeavat ongelmat
7. Laskennan perusmalleista
8. Rinnakkaisalgoritmit
9. Satunnaistetut algoritmit
10. Uusia laskennan malleja

*Raamatunlainaukset on plagioitu kurssin pohjana olevasta Harelin kirjasta

Käytännön järjestelyt

Valinnainen tietojenkäsittelytieteen cum laude -kurssi, 2 ov.

Suunnattu erityisesti

- tutkijalinjan
- opettajan suuntautumisvaihtoehdon

opiskelijoille.

vastuuhenkilö Matti Nykänen, vastaanotot C481 ma, ke 11:00-11:30 ja pe 9:30-10:00, puh. 191 44237,
sähköposti `matti.nykanen@cs.helsinki.fi`

Minimiesitiedot: perustietous ohjelmoinnista ja algoritmeista (esim. Ohjelmoinnin perusteet)

Luennot viikoilla 43–49 ke 12:15–14:00 ja pe 10:15–12:00 salissa A217. (25.10. ja 6.12. ei ole luentoa.)

Harjoitukset viikoilla 45–49 ma 12:15–14:00 salissa A217. (Ei vielä viikolla 44!)

Vapaaehtoisia mutta tehdyiksi merkitsemistäsi (= läsnä ja valmiina esittelemään ratkaisuehdotustasi) tehtävistä saat *lisäpisteitä*! (6p?)

Suorittaminen ja arvostelu

(Ainoa) välikoe pe 15.12.2000
klo 10:00–14:00 laitoksen Auditoriossa.
(30p?)

Keräämäsi laskuharjoituspisteet korvaavat menettämiäsi koepisteitä arvosanaa määrättäessä.

Kurssimateriaali

Päälähde David Harel: *Algorithmics – The spirit of computing*, toinen painos, Addison-Wesley 1992 + mahdollisesti lisäksi artikkelikopioita yms.

Kurssin kotisivulle
`http://www.cs.Helsinki.FI/u/mnykanen/ATPe/`
laitetaan

- luentokalvot (kunkin luennon jälkeen)
- harjoitustehtävät
- tiedotuksia
- ...

Ei-sähköisessä muodossa oleva materiaali ilmaantuu salin A413 kurssikansioon.

1. Johdanto (Harel luvut 1–2)

Kurssilla tutustutaan tietojenkäsittelytieteen keskeisiin *algoritmisiin* käsitteisiin, ideoihin, menetelmiin ja tuloksiin.

Oma motivaatio: tietojenkäsittelytieteen ”oma juttu” on nähdä ja tehdä (sen ulkopuolelta tulevien) informaation käsittelyn ongelmien *mekanisoituvat* osat, ja niihin tarvitaan algoritmeja.

David Harel:

”Anyone [associated with computers] ought to be aware of these topics ... the special ways of thinking that go with them ought to be available to [any] enquiring person.”

”The material [...], while not directly aimed at producing better programmers or system analysts, can aid [...] by providing an overall picture of some of the most fundamental issues relevant to their work.”

Laskentaongelmat ja algoritmit

Tietokoneet suorittavat *ohjelmien* kontrolloimina mitä moninaisimpia tehtäviä. Tarkastelemme pääasiassa tehtäviä, joiden suoritus päättyy äärellisessä ajassa tuottaen annetuista *syötteistä* halutun *tuloksen*. Kukin tietokoneohjelma perustuu johonkin *algoritmiin*.

Tarkennamme ja modifioimme algoritmin määritelmää tarvittaessa; alustavasti algoritmi on *täsmällinen, yksinkertaisista perusaskelista koostuva ratkaisuperiaate*.

Esim. Kakun paistaminen:

syötteet: kakun ainekset
tulos: valmis kakku
laitteisto: kokki, uuni, astiat
ohjelma: kakun resepti (keittokirjassa)
prosessi: kakun valmistustapahtuma
algoritmi: reseptin (abstrakti) idea

Huom: rajoitetut perusoperaatiot:

”Valmista suklaakakku” ei ole (ainakaan keskivertokokille) kelvollinen resepti.

Vastaavasti tietojenkäsittelyalgoritmeissa rajoitumme ohjelmointikielissä yleisiin operaatioihin kuten yksittäisten merkkien ja numeroiden lukeminen, vertailu ja käsittely sekä normaalit kontrollirakenteet:

- peräkkäisyys (A; B)
- ehdollisuus (**if** ... **then** ... [**else** ...])
- toisto (**while**, **for**, **repeat-until**)
- alirutiinien (mahd. rekursiivinen) kutsu

Muutoinkin edellytämme automaattisesti suoritettavilta algoritmeilta suurempaa täsmällisyyttä kuin ihmisille tarkoitetuilta resepteiltä.

Lyhyt algoritmistiikan historia

300–400 e.Kr: Eukleides: kahden kokonaisluvun suurin yhteinen tekijä (gcd)

800-luku: Mohammed al-Khwarism: kymmenjärjestelmän aritmeettiset operaatiot

(1600-luvulta mekaanisia laskukoneita.)

1800-luku: Babbagen ”analyyttinen kone” (mekaaninen, *ohjelmoitava*)

1930-luku: algoritmikäsitteen formalisointi, algoritmien mahdollisuudet ja rajat: Alan Turing, Kurt Gödel, Andrei Markov, Alonzo Church, Emil Post, Stephan Kleene

1940-luku: modernin (digitaalisen) tietokoneen synty

1960-luku ...: monipuolista ja syvällistä algoritmitutkimusta

Algoritmi vs. prosessi

Esim. laskettava maksettavien palkkojen summa, kun annettuna on lista työntekijätietueita

Algoritmi luonnollisella kielellä:

(1) Pane muistiin luku 0;
(2) Käy lista läpi lisäten kunkin
        työntekijän palkka muistiinpanoon;
(3) Tulosta muistiin merkitty luku;

Vastaavasti **pseudokielellä**:

(1) TotSal:= 0;
(2) **for** each record t in the list of records **do**
        TotSal:= TotSal + t.salary;
(3) **output** TotSal;

Huom: algoritmi on (lyhyt ja) kiinteä, mutta

- syöte,
- algoritmin suoritus ja
- tuloste

voivat olla mielivaltaisen pitkiä; tämä juuri tekee algoritmeista hyödyllisiä, kiinnostavia ja haastavia!

Edell. algoritmi on helppo nähdä oikeaksi. Osoitetaan kuitenkin sen oikeellisuus huolellisesti *matemaattisella induktiolla*, joka on keskeinen algoritmien suunnittelun ja analysoinnin apuväline.

Matemaattinen induktio: Ominaisuuden $P(n)$ osoitetaan pätevän kaikilla luonnollisilla luvuilla n näyttämällä että

1. $P(0)$ pätee ja että
2. $P(n) \Rightarrow P(n+1)$

Valitaan palkanlaskenta-algoritmissa $P(n) \equiv$ "TotSal $= \sum_{i=1}^{n} t_i$.salary, kun algoritmi on käsitellyt tietueet t_i missä $1 \leq i \leq n$".

Selvästi $P(0)$ pätee (TotSal=0).

Oletetaan sitten, että $P(n)$ pätee, eli TotSal on n ensin käsitellyn tietueen palkkojen summa $\sum_{i=1}^{n} t_i$.salary. Kun käsitellään t_{n+1}, TotSal kasvaa arvolla t_{n+1}.salary eli saa arvon $\sum_{i=1}^{n+1} t_i$.salary $\Rightarrow P(n+1)$ pätee.

Algoritmi siis laskee missä tahansa työntekijälistassa oikean palkkasumman mille tahansa n ensimmäiselle työntekijälle. Esityisesti algoritmin pysähtyessä n on listan tietueiden lukumäärä ja TotSal siis haluttu kokonaissumma.

Algoritmiset ongelmat

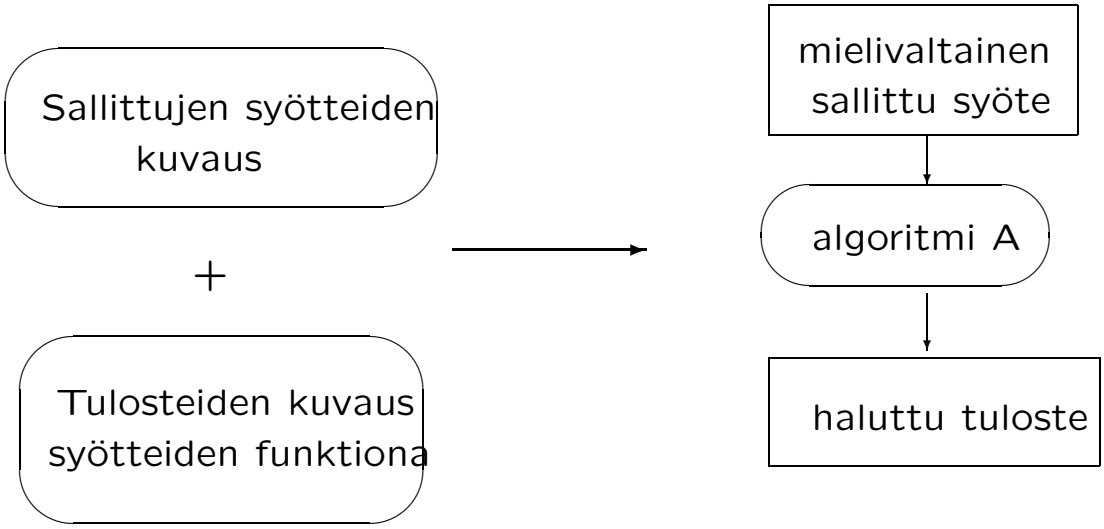
Algoritmisen ongelman muodostavat

1. sallittujen syötteiden kuvaus ja
2. haluttujen tulosten kuvaus syötteiden funktioina

Esimerkkejä algoritmisista ongelmista:

P_1: Laske annettujen (binääri)lukujen x ja y summa.
P_2: Annettuna nimi a ja aakkosjärjestetty lista nimiä $B = b_1, \ldots, b_n$, esiintyykö a listassa B?
P_3: Annettuna verkko G. Sisältääkö G Hamiltonin kehän (kunkin solmun kautta täsmälleen kertaalleen kulkevan syklin)?
P_4: Annettuna ohjelma M ja sen syöte X. Pysähtyykö $M(X)$?
P_5: Pysähtyykö annettu ohjelma M kaikilla syötteillään?

Onko algoritmia, joka ratkaisee ongelman P_i?



Algoritminen ongelma **ja sen ratkaisu**

Huom. Algoritmin on ratkaistava ongelma *kaikilla* sallituilla syötteillä; erityisesti suorituksen on *päätyttävä* niillä ja annettava *oikea* vastaus!

Proseduuri on muuten kuten algoritmi muttei välttämättä aina pääty.

Prosessi on se tekeminen jossa seurataan algoritmin/proseduurin askeleita.

Algoritmien suunnittelumenetelmät

(Harel luku 4)

"Herra neuvoo hänelle miten toimia" [Jesaja 28:26]
"sillä joka asialla on aikansa ja tapansa" [Saarn. 8:6]

- Miten algoritmeja keksitään?

Algoritmisten ongelmien ratkaiseminen on luovaa (ja monesti haastavaa!) toimintaa.

Käsiteltävien objektien (sanan laajassa merkityksessä) ominaisuuksien tunteminen usein keskeistä $\leadsto$ tilanteesta riippuen tarvitaan tietoja esim. reaalianalyysistä, lukuteoriasta, kombinatoriikasta, biologiasta, ... joihin otetaan laskennallinen näkökulma.

Pieni joukko vakiintuneita **algoritmisia menetelmiä**

- usein esiintyviä ratkaisuperiaatteita
- voivat auttaa algoritmin kehittämisessä

  - etsintä ja läpikäynti
  - ”hajoita-ja-hallitse”
  - ahneet algoritmit
  - dynaaminen ohjelmointi

Huom: näitä tarkastellaan perusteellisemmin *Algoritmien suunnittelu ja analyysi* -kurssilla

Etsintä ja läpikäynti

Monen algoritmisen ongelman ratkaisu perustuu jonkin eksplisiittisen tai implisiittisen rakenteen läpikäyntiin.

Esim. työntekijöiden palkkasumman laskenta

Taulukoiden ja lineaaristen rakenteiden kuten listojen läpikäynti johtaa tyypillisesti silmukoihin (tai sisäkkäisiin silmukoihin). Hierarkkisten rakenteiden läpikäynti on usein luontevaa rekursiivisesti.

Esim. (binäärinen etsintäpuu ja puulajittelu)

Puut ovat tietojenkäsittelyssä keskeisiä (sekä abstrakteja että konkreettisia) rakenteita. Mm. monet indeksi- eli hakemistorakenteet perustuvat puihin. Tarkastellaan tässä *binääripuita*.

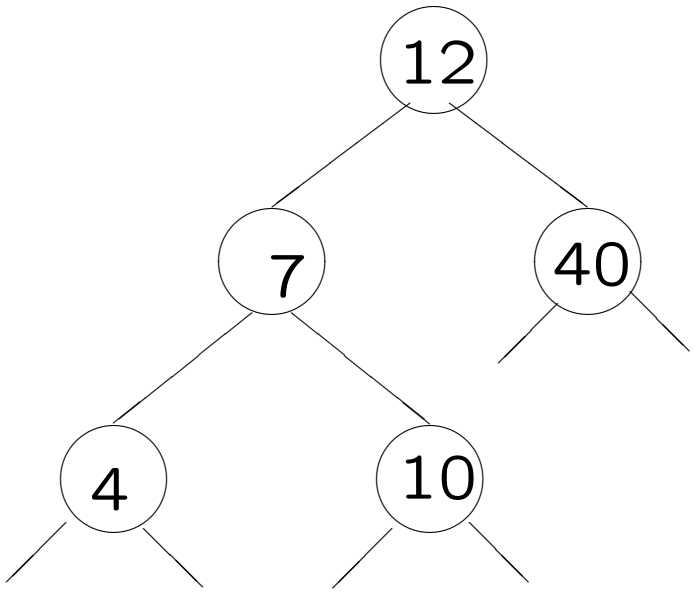
Binääripuu on rakenne, joka on joko
a) *tyhjä* tai
b) koostuu *solmusta*, jolla on *vasen* ja *oikea* alipuu; solmun alipuut ovat binääripuita, jotka eivät sisällä k.o. solmua.

Binäärinen etsintäpuu (binary search tree, BST) on binääripuu, jonka jokaisessa solmussa v voi olla talletettuna arvo v.key s.e. kaikki sen vasempaan alipuuhun v.left talletetut arvot ovat *pienempiä kuin* v.key, ja kaikki sen oikeaan alipuuhun v.right talletetut arvot ovat *suurempia kuin* v.key.

Huom: Binäärinen etsintäpuu sopii vain yksikäsitteisten arvojen tallettamiseen, koska mikään key-arvo ei voi (ilman modifiointia) esiintyä monta kertaa.

Tälläinen *assosiaatiomuisti* on kätevä sekä algoritmien suunnittelussa että ohjelmoinnissa!

Esimerkki binäärisestä etsintäpuusta:



(Ks. myös Harel kuva 2.13, sivu 44.)

Arvoa x voi hakea binäärisestä etsintäpuusta T rekursiivisella algoritmilla. Oletetaan, että solmuun v on talletettu myös avaimeen v.key liittyvää sisältöä v.data, jota haetaan avaimella x:

BSTSeach(x, T):
(1) Jos T on tyhjä, palauta ”x ei ole puussa”;
(2) muuten, jos T.key = x, palauta T.data;
(3) muuten, jos x $<$ T.key,
suorita BSTSeach(x, T.left);
(3) muuten suorita BSTSeach(x, T.right);

Etsintäpuuhun T talletetut avaimet voidaan tulostaa kasvassa järjestyksessä käymällä puu läpi sopivassa järjestyksessä, ns. *sisäjärjestyksessä*:

```
TulostaSisäjärjestys(T):
Jos T on tyhjä, älä tee mitään;
muuten
        TulostaSisäjärjestys(T.left);
        tulosta T.key;
        TulostaSisäjärjestys(T.right);
```

(Ks. myös Harel kuvat 2.14 ja 2.15, sivu 45.)

Etsintäpuun muodostaminen ja läpikäynti tarjoaa erään ratkaisumahdollisuuden lajitteluongelmaan.

Lajittelu (sorting)*: Annettuna arvot $a_1, \ldots, a_n$, aseta ne järjestykseen $b_1 \leq b_2 \leq \cdots \leq b_n$. ($\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$)

Yksinkertaisuuden vuoksi oletetaan, että mikään arvo ei toistu lajiteltavassa listassa.

Puulajittelu

(1) Muodosta syötelistasta $a_1, \ldots, a_n$ BST T;
(2) TulostaSisäjärjestys(T);

Vaiheen (1), binäärisen hakupuun muodostamisen voi tehdä lisäämällä kullekin a_i (alunperin tyhjään) puuhun solmun kohtaan, jossa sen kuuluu sijaita (BSTSearch-proseduurin nojalla).

*Parempi nimitys olisi järjestäminen (ordering)

Edellisen kuvan binäärinen hakupuu muodostuu esim. lisäämällä alunperin tyhjään puuhun järjestyksessä avaimet 12, 7, 10, 40 ja 4.

Huom: monet muut lajittelualgoritmit ovat käytännössä puulajittelua tehokkaampia.

Ongelman ”etsintäavaruuden” hahmottaminen ja sen tehokkaan läpikäyntijärjestyksen keksiminen ei välttämättä ole yksinkertaista.

(Ks. esim. etäisyyden maksimointi monikulmion sisällä, Harel kuvat 4.1 ja 4.2, sivut 80–82.)

”Hajoita-ja-hallitse”-tekniikka
(divide-and-conquer)

Usein ongelma voidaan ratkaista jakamalla se pienempiin samankaltaisiin osiin, ratkaisemalla nämä osaongelmat, ja yhdistämällä osaongelmien ratkaisut alkuperäisen ongelman ratkaisuksi. Jos osaongelmat ovat täsmälleen alkuperäisen ongelman pienempiä tai yksinkertaisempia tapauksia, ratkaisu voi perustua rekursioon.

Olemme jo edellä nähneet Hajoita-ja-hallitse-tekniikasta esimerkkejä, mm. TulostaSisäjärjestys(T).

Esim. (Hanoin tornit)

Pylväät A, B ja C; pylväässä A päällekkäin N kappaletta eri kokoisia kiekkoja ylöspäin pienenevässä järjestyksessä. Selvitettävä yksittäisten kiekkojen siirrot pylväitten A, B ja C välillä s.e. kiekkopino saadaan lopulta pylvääseen B eikä missään vaiheessa suurempi kiekko ole pienemmän päällä. (Ks. Harel kuva 2.7, sivu 31.)

Yksi kiekko (N=1) on yksinkertaista siirtää A:sta B:hen.

Entä kun kiekkoja on useampia (N>1)?

Siirretään N-1 päällimmäistä (rekursiivisesti) ”apupylvääseen” C, alimmainen A:sta B:hen ja lopuksi N-1 pienempää C:stä B:hen:

```
Moves(N,X,Y,Z):
// X on lähtö-, Y kohde- ja Z apupylväs
      if N=1 then
            output "siirrä" + X + "→" + Y;
      else
            Moves(N-1, X, Z, Y);
            output "siirrä" + X + "→" + Y;
            Moves(N-1, Z, Y, X);
      fi;
```

Esim. kolmikiekkoisen Hanoin tornit -ongelman ratkaisevat siirrot voidaan tuottaa kutsulla Moves(3, A, B, C);

Rekursiivisen algoritmin suoritusta voi tarkastella **kutsupuuna**, jossa kutakin kutsua vastaa solmu, jonka lapsisolmuina ovat k.o. kutsun tekemät rekursiiviset kutsut.

(Ks. Harel, kuva 2.8, sivu 34.)

Myös lajitteluongelma voidaan ratkaista tehokkaasti hajoita-ja-hallitse-tekniikalla. Eräs periaatteeltaan yksinkertainen tämän tekniikan sovellus on ns. **lomituslajittelu** (merge sort).

Periaate:

1) *(”hajoita”-vaihe)*: Jaa lajiteltava jono kahteen (likimain) yhtäsuureen osaan ja lajittele ne rekursiivisesti. Palauta yhden mittaiset jonot ssellaisenaan, koska ne ovat järjestyksessä *(”hallitse”)*.

2) *(yhdistämisvaihe)*: Lomita lajitellut alijonot yhdeksi järjestyksessä olevaksi jonoksi.

MergeSort($a_1, \ldots, a_n$):

    if n=1 **then return** a_1;
    else
        S1:= MergeSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$); (*)
        S2:= MergeSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$);
        return Merge(S1, S2);

(*) $\lfloor x \rfloor$ = suurin kokonaisluku, joka on $\leq x$;
"floor", "lattiafunktio".

Funktio Merge(S1,S2) lomittaa jonot S1 ja S2 yhdeksi järjestetyksi jonoksi.

(Ks. Harel, kuva 4.3, sivu 85.)

Ahneet algoritmit

Useissa ongelmissa on tarve valita jossain mielessä paras ratkaisu joukosta vaihtoehtoja. Ahne algoritmi (greedy algorithm) tekee tällaisia optimivalintoja ”lokaalisti”.

Esim. Maksettava 16 mk mahdollisimman pienellä määrällä 10, 5 ja 1 markan kolikoita.

Ahne periaate: Kunnes koko summa on katettu, valitse suurin mahdollinen kolikko, ja pienennä katettavaa summaa sen arvolla

$\rightsquigarrow$ 10 + 5 + 1 mk.

Huom: ahneus ei aina kannata. Kuvitellaan käyttöön 1, 5 ja 11 markan kolikot. Silloin ahne algoritmi maksaisi 15 mk kolikoilla 11 + 1 + 1 + 1 + 1, vaikka 5 + 5 + 5 mk olisi parempi valinta.

Esim. (Verkon virittävä puu)

Mikä on halvin rautatieverkosto, joka kuitenkin tavoittaa kaikki kaupungit? (Arvioidaan rataverkon *kustannusta* sen ratojen kokonais*pituudella*.) Ongelman tapaus voidaan esittää **painotettuna verkkona**: kaupungit ovat verkon *solmuja*, kaupunkien väliset (mahdolliset) radat verkon *kaaria*, ja kaaren *paino* on sitä vastaavan radan pituus.

Huom: verkot ovat puiden yleistys; niissä voi olla kahden solmun välillä vaihtoehtoisia kaarista muodostuvia *polkuja* (puissa täsmälleen yksi), jotka muodostavat *syklin* (eli jostain solmusta itseensä johtavan, erillisistä kaarista muodostuvan polun).

Oletetaan, että mahdolliset ratayhteydet muodostavat *yhtenäisen* verkon, s.o. jokaisesta kaupungista on olemassa junayhteys muihin.

Ongelman ratkaisun muodostaa verkon
pienin virittävä puu:

yhtenäinen, syklitön aliverkko, joka
(i) kattaa kaikki solmut ja
(ii) on kokonaispituudeltaan lyhyin

Pienimmän virittävän puun voi muodostaa ahneella algoritmilla (Prim 1957):

Valitse puuhun verkosta mahd. lyhyt kaari;
Kunnes puu kattaa kaikki solmut, **toista**:
  Lisää puuhun seuraavaksi lyhyin
  verkon kaari, joka liittyy johonkin
  puun solmuun muttei luo puuhun sykliä;
Tulosta puu;

Esimerkki: Harel kuvat 4.4 ja 4.5, sivut 86–87.

Voidaan osoittaa, että Primin algoritmi tuottaa verkon pienimmän virittävän puun. (HT?)

Ahneet algoritmit ovat usein melko yksinkertaisia ja intuitiivisia – vaikeus on yleensä sen osoittamisessa, että ne tuottavat parhaan ratkaisun. Joskus voidaan tyytyä (tai joutua tyytymään) optimaalisen sijasta ”riittävän hyvään” ratkaisuun. Tällöin puhutaan ahneista **approksimointialgoritmeista**.

Dynaaminen ohjelmointi

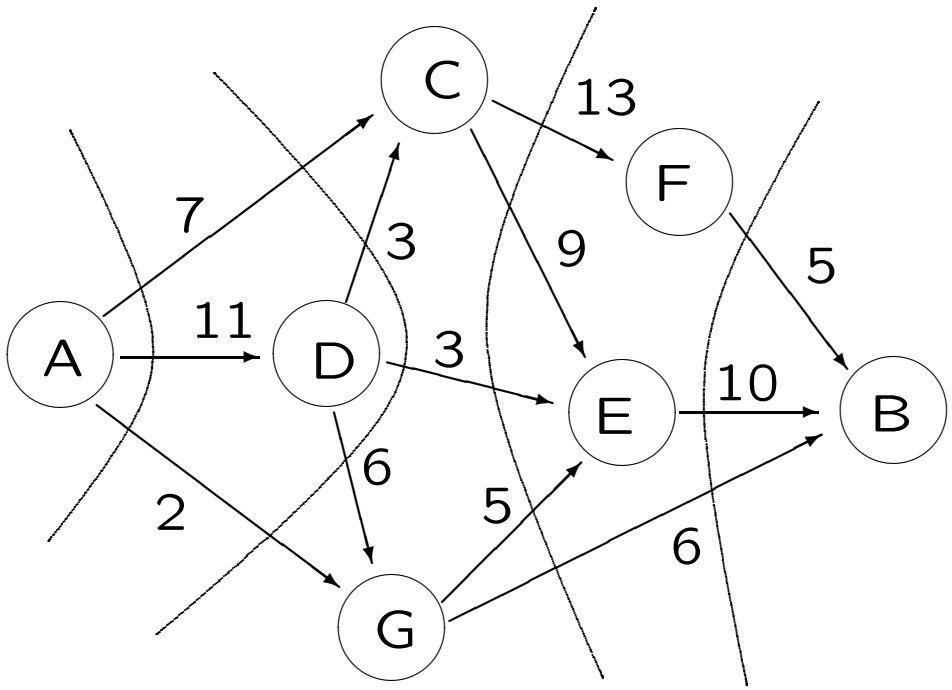
Joskus ahneet (lokaalisti optimaaliset) valinnat eivät johda oikeaan tulokseen.

(Ks. myös Harel, kuva 4.6, sivu 88.)

Esim. (”kriittisen polun pituus”)

Mallinnetaan projektin kulkua verkolla, jossa solmut vastaavat projektin välivaiheita. Verkossa on **suunnattu kaari** (u, v), jos vaiheessa u voidaan aloittaa työ, joka täytyy saada valmiiksi vaiheeseen v. Tämän kaaren (u, v) *paino* $w(u, v)$ on sitä vastaavan työn vaatima aika (esim. viikkoina).

Projektin kulkua kuvaava verkko
(A on projektin alku, B sen loppu):



Mikä on projektin vähimmäiskesto?
Sen määrää pisin A $\rightsquigarrow$ B -polku.

Ahne menetelmä, kulloinkin pisimmän kaaren valinta löytäisi reitin (A,D,G,B) pituudeltaan 11+6+6 = 23 (viikkoa). Projektin läpivienti vaatii kuitenkin vähintään 33 viikkoa:
(A,D,C,E,B) on ns. kriittinen polku.

Ongelma voidaan ratkaista soveltaen dynaamista ohjelmointia.

Dynaaminen ohjelmointi on hajoita-ja-hallitse-tekniikkaa täydentävä ratkaisuperiaate, jossa *kaikkien* osaongelmien *ratkaisut talletetaan* (esim. taulukkoon).

Tässä haettu optimiratkaisu löydetään tarkastelemalla lokaaleja valintoja ja niihin liittyvien aliongelmien optimiratkaisua:

Merkitään L(X) = pisimmän polun pituus solmusta X päätössolmuun B. Tässä
L(A) = max{7+L(C), 11+L(D), 2+L(G)},
L(D) = max{3+L(C), 3+L(E), 6+L(G)} jne.

Huom: samoja L()-arvoja evaluoidaan useasti, esim. L(C) sekä L(A):n että L(D):n selvittämiseksi. Niiden muistaminen säästää kokonaistyötä.

Laskenta saadaan tehokkaaksi laskemalla (ja tallettamalla) L()-arvot ”takaperoisessa” järjestyksessä; ensin käsitellään päätössolmu B, sitten sen välittömät edeltäjät jne:

$L(B)$ **:=** 0;
while $L(A)$ ei vielä ole tiedossa **do**
  Valitse mikä tahansa solmu v jonka $L(v)$ ei vielä ole tiedossa mutta jonka jokaisen lähtevän kaaren $v \to u$ päässä $L(u)$ on jo tiedossa;
  $L(v)$ **:=** $\max\{w(v,u) + L(u)\}$
end while

Ja sitten vain kehitetään sopivat *tietorakenteet* joilla valinta on nopeaa . . .

Algoritmiset metodit ovat hyödyllisiä varsinkin *tehokkaiden* algoritmien aikaansaamiseksi. Palaamme asiaan, kun tutustumme tehokkuusanalyysin perusteisiin.

Esimerkki algoritmin kehittämisestä

Syötteenä annetaan 2 merkkijonoa
$\alpha = a_1 a_2 a_3 \ldots a_m$ ja $\beta = b_1 b_2 b_3 \ldots b_n$. Jonoa α voidaan muokata (a) lisäämällä ja
(b) poistamalla yksittäisiä merkkejä.
Laskettava montako muokkausoperaatiota tarvitaan enintään jotta saataisiin β.

Ns. *editointietäisyysongelma* (edit distance problem) jolla esim. mitataan (bio)jonojen samankaltaisuutta.

Esimerkki biologisen ilmiön (mutaation) muuntamisesta algoritmiseksi laskentaongelmaksi.

(Lisätietoa kurssilla *Merkkijonomenetelmät*.)

Yritetään ensin *hajoittaa ja hallita*. Mistä jonon β *viimeinen* merkki b_n on tullut optimiratkaisussa?

1. Se voisi olla lisätty jonoon α.

   Silloin operaatioita kuluu: 1 lisäys + paras tapa muuttaa jono α
   alkuosaksi $b_1 b_2 b_3 \ldots b_{n-1}$ − rekursio!

2. Se voisi olla jonon α alkuperäinen viimeinen merkki a_m jos $a_m = b_n$.

   Vain ne operaatiot joilla
   jono $a_1 a_2 a_3 \ldots a_{m-1}$ muuttuu
   jonoksi $b_1 b_2 b_3 \ldots b_{n-1}$.

3. Se voisi olla jonon α jokin aikaisempi merkki a_{m-k} jolloin a_m on tiellä.

   1 poisto + jonon $a_1 a_2 a_3 \ldots a_{m-1}$ muutos jonoksi β.

Merkitään $W_{i,j}$ = "minimimäärä muutoksia joilla alkuosa $a_1 a_2 a_3 \ldots a_i$ muuttuu alkuosaksi $b_1 b_2 b_3 \ldots b_j$".

$W_{i,j}$ on siis *pienin* seuraavista luvuista:

1. $1 + W_{i,j-1}$

2. $W_{i-1,j-1}$ mutta vain jos $a_i = b_j$

3. $1 + W_{i-1,j}$

- $W_{0,j} = j$ koska tyhjästä jonosta saa jonon $b_1 b_2 b_3 \ldots b_j$ j lisäyksellä.

- $W_{i,0} = i$ koska jono $a_1 a_2 a_3 \ldots a_i$ tyhjenee i poistolla.

Ensimmäinen yritys: suora rekursio

function $W(i, j\colon \mathbb{N})\colon \mathbb{N}$

```
 1: if i = 0 then
 2:     u := j
 3: else if j = 0 then
 4:     u := i
 5: else
 6:     t := 1 + min(W(i − 1, j), W(i, j − 1));
 7:     if a_i = b_j then
 8:         u := min(t, W(i − 1, j − 1))
 9:     else
10:         u := t
11:     end if
12: end if
13: return u
```

Ongelma: kutsu $W(i, j)$ tekee saman kutsun $W(i-1, j-1)$ monta (2 tai 3) kertaa!

Toinen yritys: apumuistin käyttö välituloksille ("memoization")

Talletetaan jo lasketut arvot apumatriisiin $V[0 \ldots m, 0 \ldots n]$ ja katsotaan ensin sieltä:

function $W(i, j\colon \mathbb{N})\colon \mathbb{N}$
  if paikka $V[i, j]$ on jo täytetty **then**
    u **:=** $V[i, j]$
  else
    Ensimmäisen yrityksen rivit 1–12;
    $V[i, j]$ **:=** u
  end if
  return u

Ongelma: apumuistin täyttökirjanpito!

Kolmas yritys: dynaaminen ohjelmointi.

Täytetään V sellaisessa järjestyksessä ettei kirjanpitoa enää tarvita.

for all j **:=** 0 **to** n **do**
  $V[0, j]$ **:=** j
end for;
for all i **:=** 1 **to** m **do**
  $V[i, 0]$ **:=** i;
  for all j **:=** 1 **to** n **do**
    Ensimmäisen yrityksen rivit 6–11
    joissa rekursiokutsut $W(\ldots, \ldots)$
    korvattu taulukkoviitteillä $V[\ldots, \ldots]$;
    $V[i, j]$ **:=** u kuten kalvolla 41
  end for
end for

Vastaus ilmestyy lopuksi
taulukkopaikkaan $V[m, n] = W(m, n) = W_{m,n}$.

Esimerkiksi jonojen $\alpha =$ `ananas` ja $\beta =$ `banana` etäisyyden laskenta etenee seuraavasti:

			β					
		i	`b`	`a`	`n`	`a`	`n`	`a`
	j	0	1	2	3	4	5	6
	`a`	1	2	1	2	3	4	5
α	`n`	2	3	2	1	$\cdots$		
	`a`	3	$\vdots$					
	`n`	4						
	`a`	5						
	`s`	6						

Lisäongelmia (HT):

- Miten selviävät itse tarvittavat minimimuutokset?
- Voisiko $(m + 1) \times (n + 1)$ alkion aputaulukkoa kutistaa?
- ...

3. Algoritmien oikeellisuus

(Harel luku 5)

"Opettakaa minulle, missä olen mennyt harhaan"
[Job 6:24]

• Kuinka varmistua siitä, että algoritmi toimii oikein?

Ohjelmointia harjoittaneet tietävät, että oikein toimivien ohjelmien aikaansaaminen on haastavaa. Miljoonavahinkoja aiheuttaneista ja jopa ihmishenkiä vaatineista ohjelmistovirheistä kerrotaan tosia tarinoita.

Tietokoneohjelmien oikeellisuutta pyritään normaalisti osoittamaan **testaamalla** eli kokeilemalla toimintaa edustavalla ja kattavalla joukolla syötteitä.

Mahdollisia syötteitä usein ääretön määrä
$\leadsto$ testaamalla ei voi saada oikeellisuudesta *varmuutta*.

Meidän pitää voida varmistua siitä, että esittämämme yleinen ratkaisumenetelmä, algoritmi, toimii oikein.

Algoritmeja (ja kriittisiä ohjelmia) voidaan **verifioida** eli **todistaa oikeaksi**.

Ohjelmien verifiointia käsitellään perusteellisemmin kurssilla *Algoritmien oikeellisuus ja johtaminen*.

Mitä verifioinnissa on osoitettava?

Algoritmin pitää olla ongelman määrityksen mukainen ratkaisu: Sen on tuotettava kaikilla sallituilla syötteillä haluttu, spesifikaation mukainen tulos.

Algoritmi on *virheellinen*, *jos* jollain sallitulla syötteellä

- saadaan väärä vastaus tai
- suoritus voi joutua virheelliseen tai määrittelemättömään tilanteeseen (nollalla jako, yritys edetä puussa lehtisolmun lapseen, viittaus taulukon ulkopuolelle ym.) tai
- suoritus on päättymätön, ns. **ikuinen silmukka**.

Osittainen ja täysi oikeellisuus

Algoritmi tai ohjelma on **osittain oikeellinen**, jos se tuottaa sallituista syötteistä oikean tuloksen aina, *mikäli suoritus päättyy*.

Osittain oikeellinen algoritmi siis ei anna väärää vastausta, mutta voi joutua joskus ikuiseen silmukkaan.

Sanomme, että algoritmi/ohjelma **terminoi**, jos sen suoritus päättyy millä tahansa ongelman määrityksen mukaisella syötteellä.

Osittain oikeellinen ja lisäksi terminoiva algoritmi/ohjelma on **täysin oikeellinen**

Ks. Harel, kuva 5.3.

Osittaisen oikeellisuuden osoittaminen

Osittaisen oikeellisuuden osoittamisessa näytetään, että algoritmin suorituksessa *ei tapahdu vääriä asioita*:

• algoritmi ei tuota vääriä tuloksia ja

• muuttujilla ei ole virhetilanteeseen johtavia arvoja (esim. nolla jakajana tai indeksi taulukon rajojen ulkopuolella)

Argumentointi tapahtuu liittämällä algoritmissa sopiviin varmistuskohtiin (checkpoint) **väittämiä** (assertion). Väittämillä ilmaistaan asiantila, jonka halutaan osoittaa olevan k.o. kohdassa voimassa.

Yleensä väittämissä käytetään algoritmin muuttujia:

Esim. "taulukon alkiot 1, ..., I on järjestetty."

Algoritmin **alkuväittämä** (initial assertion) on yleensä sallitut syötteet kuvaava ehto. Vastaavasti **loppuväittämä** (final assertion) kuvaa sen, että tulosteen ja syötteen suhde on ongelman ratkaisulta haluttu.

Harel, kuva 5.5

Osittainen oikeellisuus perustellaan osoittamalla, että algoritmiin sijoitetut väittämät ovat **invariantteja** eli tosia aina kun suoritus saavuttaa niiden sijainnin.

Invarianttien verifiointi tapahtuu osoittamalla, että mikään siirtymä suorituksessa uuteen varmistuskohtaan ei tee siihen liittyvää väittämää epätodeksi.

Ns. Floydin menetelmä (Robert Floyd 1967)

Esim. Merkkijonon peilikuva

Toteutettava merkkijonon kääntävä funktio **reverse**, jolla reverse(" $c_1 \ldots c_n$ ") = " $c_n \ldots c_1$ ".

Esim. reverse("kalakauppias") = "saippuakalak"

Merkitään tyhjää merkkijonoa symbolilla λ. Oletetaan merkkijonojen käsittelyyn funktiot **head**(" $c_1 \ldots c_n$ ") = " c_1 "ja **tail**(" $c_1 \ldots c_n$ ") = " $c_2 \ldots c_n$ ".

Esim. head("kalakauppias") = "k"ja tail("kalakauppias") = "alakauppias".

Merkitään lisäksi merkkijonojen katenointia (Java-tyyliin) plus-merkillä:
"kala" + "kauppias" = "kalakauppias".

Funktio reverse(S) voidaan nyt toteuttaa allaolevalla algoritmilla (Harel kuva 5.6):

```
X:= S;
Y:= λ;
while X ≠ λ do
      Y:= head(X) + Y;
      X:= tail(X);
od;
output Y;
```

Intuitiivinen idea: Jonon S alusta siirretään yksitellen merkkejä alunperin tyhjän jonon Y alkuun; Y "kasvaa oikealta vasemmalle"

Osoitetaan ensin algoritmin osittainen oikeellisuus. Lisätään algoritmiin (kommentteina) kolme väittämää (Harel kuva 5.7):

// I: S on merkkijono
X:= S;
Y:= λ;
while X $\neq \lambda$ **do**// II: S = reverse(Y) + X
    Y:= head(X) + Y;
    X:= tail(X);
od;
// III: Y = reverse(S)
output Y;

Väittämä II kuuluu suorituksessa siihen kohtaan jossa silmukkaehtoa testataan.

Huom: väittämät I ja III muodostavat reverse-ongelman spesifikaation.

Osoitetaan väittämien invarianssi tarkistamalla kaikki mahdolliset varmistuskohtien väliset siirtymät (I $\rightarrow$ III, I $\rightarrow$ II, II $\rightarrow$ II ja II $\rightarrow$ III)

Mahdolliset suoritukset ovat I $\rightarrow$ III ja
I $\rightarrow$ II $\rightarrow$ [II $\rightarrow \cdots$ II $\rightarrow$] III (Harel kuva 5.8);
siten yksittäisten siirtymien verifiointi
osoittaa, että algoritmin lopussa sen
loppuväittämä on tosi.

Huom: varmistuskohdat valittu siten, että
niiden välisiin siirtymiin ei sisälly silmukoiden
toistoja

Hahmotellaan siirtymiin liittyvien väittämien
invarianssitodistus:

I $\rightarrow$ III: **while**-silmukkaa ei suoriteta lainkaan,
joten X=S=λ, ja siis pisteessä III
Y=λ=reverse(S)

I $\rightarrow$ II: väite II pätee, koska X=S ja
Y=λ=reverse(Y)

II $\rightarrow$ II: Merkitään muuttujien arvoja ennen silmukan rungon suoritusta X_1 ja Y_1 sekä sen jälkeen X_2 ja Y_2. Ennen siirtymää siis S = reverse(Y_1) + X_1. Jos S = $c_1 \ldots c_n$, arvot ovat siis siirtymän alkuehdon mukaan jollain $1 \leq k \leq n$ seuraavat:

$$\overbrace{c_{k-1} \cdots c_1}^{Y_1} \; \overbrace{c_k \cdots c_n}^{X_1}$$

Silmukan rungon suorituksen ansiosta Y_2 = head(X_1) + Y_1 ja X_2 = tail(X_1). Siirtymän jälkiehdon II, S = reverse(Y_2) + X_2, nähdään siis olevan voimassa:

$$\overbrace{c_k \cdots c_1}^{Y_2} \; \overbrace{c_{k+1} \cdots c_n}^{X_2}$$

Siirtymä II $\rightarrow$ III tapahtuu kun X=λ, jolloin siis S=reverse(Y) eli Y=reverse(S).

Olemme osoittaneet algoritmin osittaisen oikeellisuuden.

Terminoinnin osoittaminen

Algoritmin terminointi voidaan osoittaa valitsemalla sopiva suorituksen edetessä pienenevä suure, ns. **konvergentti**, joka kuitenkaan ei voi pienetä loputtomasti. Kuten väittämät, konvergentti riippuu usein algoritmin muuttujista tai tietorakenteista.

Esim. käsittelemättömien tietueiden lkm, lajittelualgoritmissa väärässä kohtaa jonoa olevien alkioiden lkm tms.

Jatketaan reverse-algoritmin parissa osoittamalla sen terminointi.

Reverse-algoritmin suoritus voi olla päättymätön vain jos varmistuskohta II ohitetaan äärettömän monesti. Osoitetaan tämä mahdottomaksi valitsemalla konvergentti, joka pienenee jokaisella II-kohdan ohituksella.

Merkkijonon X pituus selvästi pienenee jokaisessa silmukan toistossa. Toisaalta silmukan suoritus päättyy, kun X:n pituus $= 0$.

$\rightsquigarrow$ algoritmi terminoi.

Olemme osoittaneet algoritmin täyden oikeellisuuden.

Automaattinen verifiointi?

Algoritmien huolellinen verifiointi käsin on selvästikin työlästä.

Missä määrin tietokone voi auttaa verifioinnissa?

Voiko algoritmien verifioimisen suorittaa algoritmisesti?

Harel, kuva 5.4

Valitettavasti verifioinnin täydellinen automatisointi ei ole mahdollista.

Varmistuskohtien valinta siten, että jokainen toistorakenteen suoritus kulkee jonkin varmistuskohdan kautta voidaan automatisoida.

Väittämien ja konvergenttien valintaa ja niihin liittyviä todistuksia ei voida suorittaa algoritmisesti; sivuamme tätä myöhemmin.

Ohjelmien oikeaksitodistamista tukevia **todistusjärjestelmiä** tutkitaan ja kehitetään. Ne toimivat vuorovaikutuksessa käyttäjän kanssa ja pystyvät huolehtimaan suurelta osin väittämien verifiointiin tarvittavasta työläästä kaavamanipuloinnista.

Huom: väittämiä voi käyttää myös ohjelmien testauksen apuna: Esim. C-kielen **assert**-makro, joka tuottaa virheilmoituksen jos sen argumenttina oleva ehto on suorituksessa epätosi.

Rekursiivisen algoritmin verifiointi

Esim. Hanoin tornit

```
Moves(N,X,Y,Z):
// X on lähtö-, Y kohde- ja Z apupylväs
    if N=1 then
        output "siirrä" + X + "→" + Y;
    else
        Moves(N-1, X, Z, Y);
        output "siirrä" + X + "→" + Y;
        Moves(N-1, Z, Y, X);
    fi;
```

Rekursiivisten algoritmien verifiointi voi olla suhteellisen helppoa.

Terminointi: Palautetaan mieliin Moves-proseduurin (binäärinen) kutsupuu. Suoritus on päättymätön vain jos kutsupuu on ääretön. Parametri N pienenee jokaisessa rekursiivisessa kutsussa yhdellä

$\leadsto$ Moves(N, X, Y, Z)-suorituksen kutsupuun syvyys on N, kun N on positiivinen kokonaisluku. Algoritmi siis terminoi.

Osittainen oikeellisuus voidaan osoittaa induktiolla. Osoitetaan, että kaikilla $N \geq 1$ pätee $P(N)$:

"Kutsun Moves(N, X, Y, Z) suoritus tuottaa jonon sallittuja siirtoja, joilla (sallitussa tilanteessa) N pienintä kiekkoa voidaan siirtää sauvasta X (mikäli ne ovat siinä) sauvaan Y eikä muihin kiekkoihin kosketa."

Tapaus $N = 1$ on selvä: "siirrä X $\rightarrow$ Y" on sallittu yhden kiekon siirto maalipylvääseen, joka koskee ainoastaan pienimpään kiekkoon.

Induktioaskel $N > 1$: ks. Harel kuva 5.10 selityksineen.

Terminointi + osittainen oikeellisuus $\rightsquigarrow$ täysi oikeellisuus.

Huom: usein algoritmin verifiointi jo rinnan sen suunnittelun kanssa on hyödyllistä.
(ns. **as-you-go** verification;
esimerkkejä mahdollisesti harjoitustehtävinä.)

Logiikka ja laskenta

(VAROITUS: Loput tästä luvusta ovat luennoijan omia päähänpinttymiä. . .)

Logiikka on jo Aristoteleen ajoista (300-luvulta e.a.a.) ollut ”oppi oikeasta päättelystä”: menetelmä jolla kaikkien tunnustamista tosiasioista lähtien voidaan epäilijä vakuuttaa johtopäätöksestä tekemällä sarja vain sellaisia *päättelyaskelia* jotka kaikki myöntävät muodollisesti oikeiksi.

1900-luvulle tultaessa kaivattiin *matematiikalle varmaa perustaa*. Päätettiin *formalisoida looginen päättely*.

Päättely (argumentointi) alun perin *kielellinen* toimitus, ja siksi matemaattinen logiikkakin kielen huomioiva matematiikan haara.

Ongelma: Miten formalisoida ”päättelyaskel jonka kaikki myöntävät muodollisesti oikeaksi”?

Ratkaisu: Formalisoidaan *(mekaanisen) laskettavuuden käsite* eli algoritmit 1930-luvulla – siis ennen tietokoneita annettiin matemaattinen vastine ikivanhalle intuitiiviselle käsitteelle.

Lisätietoja, tuloksia ja tekniikoita Matematiikan laitoksen kurssilla *Matemaattinen logiikka*.

Oma mielipide: logiikka nykyäänkin tärkeää tietojenkäsittelyssä, erityisesti tietokantateoriassa, tekoälyssä, ohjelmointikielten teoriassa ja ohjelmien oikeellisuustarkasteluissa.

Logiikka

Loogisen *objekti*kielen *merkitysoppi (semantiikka)* kertoo miten väitelauseiden *totuus* määritellään.

Tarski: perusväitteiden yhdistelyoperaatiot (”. . . ja. . . ”, ”kaikilla. . . ”,. . .) puretaan induktiivisella määritelmällä *meta*kieleen jossa perusväitteiden totuus ”nähdään helposti”.

”Yks’ vain nainen maailmassa on” kirjoitetaan tavallisella (ensimmäisen kertaluvun predikaatti)logiikalla

$$\exists x.\mathit{nainen}(x) \wedge \forall y.\mathit{nainen}(y) \rightarrow y = x$$

joka kääntyy metakielelle ”on alkio x joka on *nainen* ja jokaisella alkiolla y pätee, että jos y on *nainen*, niin y on sama alkio kuin x”.

Sitten vain katsotaan ulos ikkunasta!

(Logiikka, jatkoa)

Mekaaniset *päättelysäännöt*

$$\frac{\textit{premissi}_1 \ldots \textit{premissi}_k}{\textit{johtopäätös}}\text{[sivuehdot]}$$

valitaan totuuden säilyttäviksi: jos *premissi*t ovat tosia [ja sivuehdot voimassa] niin *johtopäätös*kin on tosi.

Olkoon jo todistettu "jos ϕ niin ψ". Siitä voidaan päätellä "jos on jokin x jolla ϕ niin ψ" mikäli ei aiemmin oletettu että väitteiden ϕ ja ψ alkiot x olisivat samat:

$$\frac{\phi \to \psi}{(\exists x \phi) \to \psi}[x \text{ ei } \psi\text{:ssä}]$$

Päättelystä enemmän Matematiikan laitoksen kurssilla *Logiikka I*.

Entä sääntöjoukon *täydellisyys* eli voidaanko kaikki tosiasiat myös todistaa? 1KL: kyllä.

(Logiikka, jatkoa)

Maailma kuvataan *teorialla*: joukolla tosina pidettyjä loogisia lauseita eli *aksioomia*.

Kysymykseen "onko Ψ totta?" vastataan

1. kääntämällä kysymys loogiseksi kaavaksi ψ,

2. yrittämällä todistaa käännös teoriasta,

3. ja vastaamalla "kyllä" jos todistus *löytyy*.

Tarpeeksi vahvoissa logiikoissa (kuten 1KL) ei mikään todistuksenetsintäalgoritmi voi aina tietää milloin etsintä on tuomittu epäonnistumaan ja voidaan vastata "ei".

Täydellisillä teorioilla ei ole tätä epätietoisuusongelmaa.

Laskenta

Lähtötiedot (kuten luvut) kirjoitetaan sovitulla syötesyntaksilla (kuten numerojonoina). [käännös logiikkaan]

Laskulaite tekee toimiessaan yksikäsitteisesti määriteltyjä perusaskeleita. [päättelysäännöt]

Perusaskelperhe on *universaali* jos niitä yhdistelemällä voi toteuttaa minkä tahansa laskentaprosessin. [sääntöjen täydellisyys]

Vaaditun kuvauksen toteuttavien perusaskeljonojen (ääretön) perhe kuvataan (äärellisellä) ohjelmalla. [teoria]

Annetuilla syötteiden kuvauksilla suoritetaan (ainoa) mahdollinen perusaskelten jono. [todistetaan teoriasta]

Jos jono aina päättyy, on kyseessä algoritmi. [teorian täydellisyys]

(Laskenta, jatkoa)

Jonon päättyessä luetaan vastaus ja käännetään sen syntaksi takaisin merkitykseensä. [on/ei]

Logiikka kuvailee epäsuorasti miten maailman asiat *ovat*.

Laskenta neuvoo miten tästä kuvauksesta voidaan *selvittää* se.

$$\text{algoritmi} = \text{logiikka} + \text{kontrolli}$$

Logiikkaa pidetään ohjelmointia *deklaratiivisempana* koska siinä esitetään vain *mitä* muttei *miten*.

Deklaratiivisiin ohjelmointikieliin on lisätty piirteitä tietämyksen esittämisestä: esim. oliokielissä voidaan ilmaista suoraan määritelmät muotoa "X on sellainen Y jolla. . ."

3 tietä loogisesti perusteltuihin algoritmeihin

1. tie: väittämät

Kalvoilla 48–54 ohjelmoija liitti sopivia väittämiä ohjelmansa – suorituspolkujen kannalta – "strategisiin" kohtiin.

Kalvoilla 55–56 ohjelmoija liitti sopivia konvergentteja ohjelmansa silmukoihin takaamaan niistä poistumisen.

Väittämä- ja konvergentti"nuolet" ovat väitelauseita muotoa "jos ennen. . . niin jälkeen. . . ". Esimerkiksi nuolessa $\mathrm{II} \rightarrow \mathrm{II}$ puhuttiin muuttujien (X, Y) arvoista ennen (X_1, Y_1) ja jälkeen (X_2, Y_2) tarkasteltavaa laskentapolun pätkää.

Nämä väitelauseet voidaan muotoilla "tavallisessa" logiikassa ja (yrittää) todistaa formaalisti, tai varmistua niiden totuudesta semanttisin järkeilyin.

Näin toiminee jokainen ohjelmoija – ellei muualla niin kommentoidessaan koodiaan.

Algoritmeja voi suunnitella ja ohjelmia kirjoittaa myös ”toisin päin”: ensin annetut alku- ja loppuväittämät, sitten niiden väliin sopiva puolivälin väittämä, ja ”induktiivisesti” samoin alku- ja loppupuoliskon – hajoita ja hallitse!

Itse laskentaoperaatiot kirjoitetaan toteuttamaan ohjelman tilan muutos kahden peräkkäisen väittämän välillä. Silmukoihin ja rekursioon liitetään vastaavat pysähtymiskonvergentit.

Lisätietoa kurssilla *Algoritmien oikeellisuus ja johtaminen*.

Kalvon 28 lomituslajitteluun voitaisiin päästä seuraavasti:

Alku: syötteenä on järjestämätön lista.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
Alkuväli: syöte on jaettu kahteen noin yhtä pitkään listaan.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
Loppuväli: listat on lomitettu.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
— Jaa syöte kahteen noin yhtä pitkään listaan.
Alkuväli: syöte on jaettu kahteen noin yhtä pitkään listaan.
— Järjestä kumpikin lista rekursiivisesti.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
— Lomita järjestetut listat keskenään.
Loppuväli: järjestetyt listat on lomitettu keskenään.
— Palauta lopputuloksena lomituksen tulos.
Loppu: tuloksena on sama lista järjestettynä.

Tämän tien (pää)ongelmana on, että laskentapolut jäävät loogisen tarkastelun ulkopuolelle.

2. tie: laskentapolkujen logiikka

Edellinen ongelma voidaan ratkaista määrittelemällä logiikka ei vain alkioille vaan myös poluille.

Tai kielellisesti, ottamalla logiikkaan *aikaa koskevia sanoja* kuten "seuraavassa silmänräpäyksessä" ($\bigcirc$), "aina tästä eteenpäin" ($\Box$),. . .

(Aikalogiikasta lisää Harel sivut 289–291.)

Silloin se, että väittämän II totuus säilyy, voidaan kirjoittaa vaikkapa

$$\Box(\mathrm{II} \rightarrow \bigcirc \mathrm{II})$$

eli "jokaisella suoritusaskeleella pätee, että jos II on nyt totta, niin se on totta myös tätä seuraavalla askeleella".

Silloin kalvojen 48–56 esimerkkialgoritmin osittaisen oikeellisuuden teoria voisi olla vaikka: edellinen väittämän II säilyvyys ja alustuksen aksiooma

$$\mathrm{I} \rightarrow \bigcirc \mathrm{II}.$$

Niistä voidaan (yrittää) todistaa aikalogiikan omilla päättelysäännöillä tai nähdä semanttisesti osittainen oikeellisuus

$$\mathrm{I} \rightarrow \bigcirc \Box (X = \lambda \rightarrow \Box \bigcirc \mathrm{III})$$

eli "jos lähtöväittämä I taataan, niin toisesta askeleesta alkaen pätee, että jos silmukkaehto joskus laukeaa, niin sen jälkeen III pätee".

Menestyksekäs *äärellisissä* järjestelmissä: mikroprosessoreissa, protokollissa,... Lisää kurssilla *Automaattinen verifiointi*.

Tämän tien (pää)ongelmana on että deklaratiivinen ohjelmoija ei halua kirjoittaa (juuri) mitään laskentapoluista.

3. tie: "loogiset" ohjelmointikielet

Tällä tiellä edellinen ongelma ratkaistaan määrittelemällä algoritmien kuvausnotaation – ohjelmointikielen – semantiikka laitetason sijasta päättelysääntöjen tapaan.

Yksinkertaisen perusaskeleen idea pois:

Logiikkaohjelmointikielissä kuten Prolog (Harel, sivut 70–71) askeleeksi tulee suoraan (rajoitettu) todistusaskel:

$$\frac{\neg a_1 \vee \ldots \vee \neg a_m \qquad b_1 \wedge \ldots \wedge b_n \rightarrow a_1}{\neg b_1 \vee \ldots \vee \neg b_n \vee \neg a_2 \vee \ldots \vee \neg a_m}$$

Funktionaalisissa kielissä kuten Lisp (Harel, sivut 68–70) askeleeksi tulee monimutkaisemman lausekkeen sievennys:

$$\frac{\texttt{if true then } e_1 \texttt{ else } e_2}{e_1}$$

Lisätietoja kurssilla *Symbolinen ohjelmointi*.

Tavoitteena on helpottaa loogisesti oikeiden ohjelmien laadintaa ja todistamista lyhentämällä välimatkaa ohjelmakoodin ja ajattelun välillä.

Kalvon 52 esimerkkialgoritmi voitaisiin tällä tiellä johtaa seuraavasti (vain intuitio):

Annetun merkkijonon S käännös reverse(S) voitaisiin laskea kutsulla rev(S, λ) jos meillä olisi apufunktio rev(X, Y) = reverse(X) + Y.

Johdetaan apufunktio induktiolla sen ensimmäisen (merkkijono)parametrin X suhteen.

Ei-rekursiivisessa perustapauksessa $X = \lambda$ vastaus on Y.

Rekursiivisessa tapauksessa $X \neq \lambda$ on olemassa lyhyempi jono tail(X) ja merkki head(X).

Meillä on siis lupa käyttää lauseketta

$$\text{rev}(\text{tail}(X), \text{head}(X) + Y)$$

koska se on määritelty induktiivisesti.

Se on myös oikea valinta, sillä sen arvo on induktiivisesti

$$\text{reverse}(\text{tail}(X)) + \text{head}(X) + Y.$$

Koska head(X) on merkki, tämä on

$$\text{reverse}(\text{head}(X) + \text{tail}(X)) + Y$$

eli haettu

$$\text{reverse}(X) + Y.$$

Yhteenvetona algoritmiksi saadaan siis

function reverse(S)=rev(S,λ)

missä

function rev(X,Y)=
  if $X = \lambda$ **then**
    Y
  else
    rev(tail(X),head(X)+Y)
  end if.

Algoritmin tekeminen ja oikeaksi todistaminen kulkivat siis käsi kädessä. Ohjelmoijan vastuulle jäi etenemisstrategian valinta.

Tulos on (syntaktisin muutoksin) valmis ohjelma funktionaalisella kielellä.

Lähestymme *konstruktiivista matematiikkaa* ja *intuitionistista logiikkaa*, joissa olemassaoloväitteen "on olemassa sellainen x jolla $P(x)$" saa todistaa vain *näyttämällä* jonkin sellaisen sopivan x.

(*Ei* siis esimerkiksi ristiriidalla "jos sellaista x ei olisi, niin siitä seuraisi $0 = 1$".)

Väitteen "jokaisella x on olemassa jokin y jolla $Q(x, y)$" todistus on silloin sellainen *funktio* f joka poimii kullekin x jonkin sopivan $y = f(x)$ jolla $Q(x, f(x))$.

Myös tämä f on näytettävä lausekkeena (eikä vain parijoukkona) – eli annettava sille algoritmi!

Silloin ongelman spesifikaatiota vastaavan olemassaoloväitteen todistus on sen mukainen algoritmi.

4. Algoritmien tehokkuus

(Harel luku 6)

"vastaa jo minulle!" [Psalmi 69:18]

• Kuinka paljon suoritusaikaa tai -tilaa algoritmin suoritus vaatii?

Keskitymme lähinnä **aikavaativuuden** tarkasteluun. Myös algoritmien **tilavaativuus** on tarkastelun arvoinen, mutta tietyssä mielessä ajantarpeelle alisteinen: jos algoritmi käyttää vähän aikaa, se ei voi varata tai käyttää paljon muistitilaa.

(Algoritmianalyysiä tarkastellaan perusteellisemmin kurssilla *Algoritmien suunnittelu ja analyysi*.)

Onko tarpeen analysoida tai parantaa algoritmien tehoa?
Tietokoneethan ovat noin 1,5 vuoden kuluttua kaksi kertaa nykyistä tehokkaampia (Mooren laki). **On**, sillä:

- Tehoton algoritmi (tai vaativa ongelma) voi vaatia kohtuullisillakin syötteillä ja nopeimmillakin koneilla jopa *miljoonien vuosien* suoritusajan.

- Tehon kasvua tarvitaan *suurempiin* ongelmiin, ei nykyisten nopeuttamiseen. On tärkeää millä vauhdilla ajan tarve *kasvaa*.

- Ns. tosiaikajärjestelmissä on ongelmat ratkaistava niin nopeasti, että vasteet voivat ohjata toiminnan (teollisuusprosessin, lentokoneen tai auton ohjausmekanismin ym.) kulkua.

Algoritmien optimointi

Tarkastellaan ensin valmiin algoritmin tehostamista.

Toistorakenteiden optimointi siirtämällä laskentaa mahdollisuuksien mukaan silmukoiden ulkopuolelle on eräs keskeinen optimointitekniikka.

Esim. Olkoot oppilaiden koepisteet taulukossa P[0 ... N-1] ja paras saavutettu pistemäärä MAX pistettä, $0 < \text{MAX} < 60$. Halutaan skaalata pisteet välille 0–60 siten, että parhaat saavat 60 pistettä ja kaikkien pisteitä kasvatetaan samassa suhteessa:

```
(1)      for (I = 0; I < N; I++)
(2)          P[I] = P[I]*60/MAX;
```

Luku 60 on vakio ja MAX ei muutu silmukassa, joten jakolaskua 60/MAX ei tarvitse suorittaa joka kerta erikseen:

```
(1)       Fact = 60/MAX;
(2)       for (I = 0; I < N; I++)
(3)             P[I] = P[I]*Fact;
```

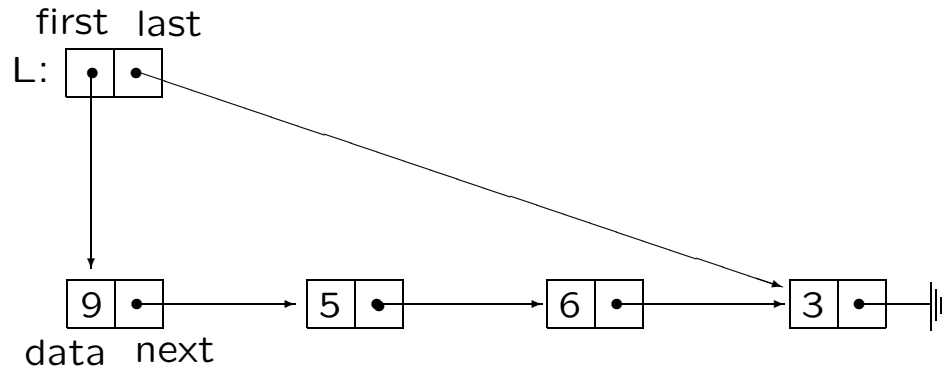
Modifioitu silmukka noin 50% tehokkaampi.

Optimoivat kääntäjät suorittavat tällaisia transformaatioita automaattisesti.

Vaan entä jos MAX onkin osoitin? Taulukon P sisään? Ns. aliasing-ongelma.

Toisinaan vastaavan optimoinnin aikaansaanti vaatii algoritmiin muutoksia, jollaisia kääntäjä tuskin osaa tehdä:

Esim. Etsitään annettua arvoa X järjestämättömästä linkitetystä listasta L:



Haku:

```
p:= L.first;
while (p ≠ null and then p.data ≠ X) do
      p:= p.next;
if p=null then
      return "not found";
else // tee jotain löytyneellä p
```

Jokaista listan alkiota kohden evaluoidaan *kaksi* ehtoa. Tämä voidaan välttää lisäämällä X ensin listan loppuun. Sen jälkeen etsinnässä ei tarvitse testata listan loppumista kesken:

```
p:= L.first;
while (p.data ≠ X) do
        p:= p.next;
if p=L.last then
        return "not found";
else // tee jotain löytyneellä p
```

$\rightsquigarrow$ noin 50% nopeutus

Algoritmien tehokkuudessa voi kuitenkin olla myös **kertaluokkaeroja**, t.s. tehokkaamman algoritmin suoritusaika ei ole missään vakiosuhteessa hitaampaan (esim. 50% tai edes 90% pienempi), vaan suoritusaikojen *suhde kasvaa rajatta* syötteiden koon mukana.

Vaativuuden kertaluokka-arviot

Algoritmien suoritusvaativuutta arvioidaan yleensä jonain syötteen koon n funktiona $f(n)$

n voi olla syötteen pituus bitteinä, usein kuitenkin jokin ”luonteva” syötteen kokoa kuvaava parametri kuten listan/taulukon alkioiden lkm tai verkon solmujen tai kaarten lkm

Yleensä (ellei muuta sanota), arvioidaan algoritmin **pahimman tapauksen** käyttäytymistä, jonka perusteella voidaan sanoa, ettei algoritmi *millään* n:n kokoisella syötteellä vaadi arvioitua enemmän aikaa tai muistia.

Arvioissa tarvittavat funktiot ovat luonnollisilla luvuilla määriteltyjä, ja ne saavat positiivisia (reaali)arvoja; rajoitumme siis tällaisiin.

Funktioiden kasvunopeutta ilmaistaan **asymptoottisilla notaatioilla**. Merkitsemme

$$f(n) = O(g(n))$$

(''f on (korkeintaan) kertaluokkaa g''), jos funktion f arvo (riittävän suurilla argumenteilla) on enintään verrannollinen funktion g arvoon.

Täsmällisemmin: $f(n) = O(g(n))$ jos on sellaiset vakiot c ja m että kaikilla $n \geq m$ pätee $f(n) \leq cg(n)$.

Huom: $f(n) = O(g(n))$ merkitsee siis, että g:n kasvunopeus on **asymptoottinen yläraja** f:n kasvunopeudelle. Usein käytetään merkintää $f(n) = O(g(n))$ (väärin) tiukemmassa merkityksessä

$$f(n) = O(g(n)) \text{ ja } g(n) = O(f(n))$$

Oikeampi merkintä tälle tiukemmalle suhteelle on

$$f(n) = \Theta(g(n)),$$

jonka voisi lukea "f ja g ovat samaa kertaluokkaa" tai "f on täsmälleen kertaluokkaa g".

Huom: Asymptoottisissa notaatioissa funktioitten vakiokertoimilla ei ole merkitystä. Esimerkiksi aikavaativuuden ilmauksina niiden voidaan ajatella osoittavan yksinkertaisten perusoperaatioitten lukumäärän riippuvuutta syötteen koosta, ei esim. niiden täsmällistä lukumäärää tai suoritusaikaa.

”Vakioiden unohtaminen” perustellaan sillä, että yhden abstraktin laskenta-askeleen ajatellaan vievän jonkin vakioajan, mutta tämän vakion tarkka arvo riippuu käytetystä tietokoneesta.

Mutta deklaratiivisissa ohjelmointikielissä tämä ”yksikköaskeloletus” ei enää pädekään...

Seuraava helposti todistettava lemma (HT?) osoittaa, että asymptoottisissa kertaluokkanotaatioissa vain dominoivilla termeillä on merkitystä:

Lemma Jos $f(n) = O(g(n))$, niin
$$f(n) + g(n) = O(g(n)).$$

Tällä periaatteella esim. $n^2 + n \log n = O(n^2)$ ja $8n^5 + 50n^3 + 100n^2 = O(n^5)$.

Esim. Binäärihaku

Jos haemme arvoa Y *järjestämättömästä* taulukosta A[1 ... N], joudumme pahimmassa tapauksessa tutkimaan kaikki N alkiota. Jos taulukko on *järjestetty* (kasvavaan järjestykseen), voimme toimia hajoita-ja-hallitse-tyyliin huomattavasti tehokkaammin.

Periaate: Verrataan arvoa Y taulukon *keskimmäiseen* alkioon A[M]. Jos ne ovat samat, etsintä päättyy onnistuneena. Muuten, jos $Y < A[M]$, se voi löytyä ainoastaan taulukon alkupuoliskosta, josta haku jatkuu samalla periaatteella. (Jos etsittävä alue supistuu tyhjäksi, haku päättyy epäonnistuneena.) Vastaavasti jos $Y > A[M]$, hakua jatketaan taulukon loppupuoliskosta.

Ks. Harel kuva 6.2

Binäärihaku (Harel kuva 6.1):

```
L:= 1; U:= N;
while L ≤ U do
      M := ⌊(L+U)/2⌋;
      case
            Y < A[M]: U:= M-1;
            Y = A[M]: return "Y paikassa M";
            Y > A[M]: L:= M+1;
      endcase
return "Y ei löydy";
```

(HT: tarkasta algoritmin oikeellisuus*)

Mikä on binäärihaun pahimman tapauksen vaativuus?

Jokaisella toistokerralla tarkasteltava taulukon osa A[L ... U] pienenee (vähintään) puoleen $\rightsquigarrow$ silmukka suoritetaan enintään $\log_2(N) + 1$ kertaa.

*Binäärihaku on erittäin hankala kirjoittaa oikein. Idea julkaistiin v. -46, mutta ensimmäinen virheetön versio vasta v. -62!

Binäärihaun vaatima logaritminen suoritusaskelten lukumäärä kasvaa huomattavan hitaasti verrattuna lineaarisen peräkkäishaun vaatimaan työhön:

N	$\log_2(N) + 1$
10	4
100	7
1000	10
10^6	20
10^9	30
10^{18}	60

Binäärihaun etsintäavaruuden puolittava periaate on erittäin keskeinen ja moneen tilanteeseen sopiva paradigma.

Esim. virheellisen syötetietueen paikantaminen.

Edellisen perusteella voimme sanoa, että binäärihaun aikavaativuus on $O(\log n)$, kun n on etsittävän taulukon alkioiden lkm.

Huom: Verifioinnissa riitti tarkastella tarkistuspisteitten (eli varmistuspisteitten) välisiä siirtymiä, kuten silmukan aloitusta, yksittäistä toistoa tai lopetusta. Vastaavasti suoritusajan kertaluokka-arvioissa riittää tarkastella tarkistuspisteiden välisten siirtymien lukumäärää.

Miksi?

Koska kukin tarkistuspisteiden välinen siirtymä tapahtuu suorittamalla jokin kiinteä määrä perusoperaatioita, sen aika on jokin t_i. Jos siirtymiä on $f(n)$ kappaletta ja t on yksittäisten siirtymien vaatimista ajoista suurin, suoritusaika on enintään

$$tf(n) = O(f(n))$$

Huom Suoritusajan asymptoottinen kertaluokka-arvio voi olla harhaanjohtava:

Esim. $O(n)$-algoritmi voi olla käytännössä tehokkaampi kuin $O(\log n)$-algoritmi, mikäli käsiteltävien syötteiden koko on pieni ja O-notaation kätkemät vakiokertoimet ovat logaritmisen algoritmin tapauksessa suuria.

Käytännön algoritmien tehokkuus on siis hyvä tarkistaa myös toteuttamalla algoritmi ja testaamalla sitä. Näitä asioita on esitelty kurssilla *Algoritmitekniikka*.

Tehokkuuslausekkeiden vakiotekijät ovat harvoin kovin suuria, joten asymptoottinen kertaluokka antaa usein varsin hyvin kuvan algoritmin tehokkuudesta.

Silmukoiden vaativuusanalyysi

Sisäkkäisten silmukoiden aikavaativuuden kertaluokka määräytyy yleensä siitä, kuinka monesti sisin silmukka suoritetaan:

```
for (i=1; i<=n; i++)
        ⋮
    for (j=1; j<=m; j++)
        ⋮
```

Tällaisen kiinteän silmukkarakenteen aikavaativuus on suoraviivaisesti $O(mn)$ ($= O(n^2)$, jos $m = O(n)$).

Monimutkaisemmat silmukat johtavat usein summalausekkeisiin. Tyypillinen tilanne:

```
for (i=1; i<=n; i++)
        ⋮
    for (j=1; j<i; j++)
        ⋮
```

Sisemmän silmukan runko suoritetaan

$$\sum_{i=1}^{n-1} i = 1+2+\cdots+(n-1) = n(n-1)/2 = O(n^2)$$

kertaa.

Rekursion suoritusaika-analyysi

Esim. Lomituslajittelu

MergeSort($a_1, \ldots, a_n$):
    if n=1 **then return** a_1;
    else
        S1:= MergeSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$);
        S2:= MergeSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$);
        return Merge(S1, S2);

Tarkastellaan yksinkertaisuuden vuoksi tilannetta, jossa n on kahden potenssi.

Yhden alkion mittaisen taulukon/listan lajittelu vaatii vakioajan.

Pitemmän listan tapauksessa tehdään ensin vakiomäärä työtä listan jakamiseksi ja sitten käsitellään rekursiivisesti kaksi puolta pienempää tapausta. Listojen lomittamisen on helppo nähdä vaativan yhdistetyn listan pituuteen verrannollisen määrän työtä.

Saadaan rekursiivisen algoritmin suoritusajalle tyypillinen **palautuskaava** eli **rekurrenssi**:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + n + 1 \end{aligned}$$

Palautuskaavojen ratkaisemiseksi on lukuisia menetelmiä. Emme kuitenkaan paneudu niihin nyt.

Monesti algoritmin vaativuutta on helpompi arvioida tarkastelemalla sen koodin rakenteen sijasta sen toimintaa korkeammalta tasolta: montako kertaa jokin asia tehdään, monestiko jotain tietorakennetta päivitetään jne.

Sovelletaan tätä lähestymistapaa lomituslajittelun analysointiin:

Tarkastellaan MergeSort-rutiinin kutsupuuta. Jokaisessa kutsussa (puun kaaressa) listan koko pienenee puoleen

$\rightsquigarrow$ puussa on $\log n$ (+ 1) tasoa.

Jokaisessa juuresta k kutsun päässä olevassa solmussa tuotetaan lomittamalla $n/2^k$-solmuista tuloslistaa. Toisaalta k kutsun päässä juurisolmusta on 2^k solmua

$\rightsquigarrow$ jokaisella kutsupuun tasolla suoritettavien Merge-rutiinien tulosten yhteispituus on n.

$\rightsquigarrow$ kokonaisaika on $O((\log n + 1)(1 + n)) = O(n \log n)$.

Keskimääräisen tapauksen analysointi

Pessimististä pahimman tapauksen analyysin tulosta hyödyllisempiä olisivat usein tulokset, jotka kertoisivat kuinka algoritmi toimii ”tyypillisesti” tai keskimäärin.

Esim. Peräkkäishaun keskimääräinen analysointi

Yksinkertainen peräkkäishaku listasta vaatii selvästi lineaarisen ajan: pahimmassa tapauksessa lista on käytävä kokonaan läpi.

Toisaalta haettu arvo voi löytyä heti listan alusta.

Oletetaan, että etsitty arvo löytyy listasta ja kaikki listan järjestykset ovat yhtä todennäköisiä. Tällöin etsitty arvo x voi olla kussakin listan paikassa $1, \ldots, n$ yhtä suurella todennäköisyydellä $1/n$. Tarvittavien vertailujen (eli suoritusajan) *odotusarvo* (matematiikan kurssilta *Todennäköisyyslaskenta I*) on eri vaihtoehtojen todennäköisyyksillä painotettu summa

$$\sum_{i=1}^{n} \frac{1}{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Siis keskimäärin noin puolet listan alkioista joudutaan tutkimaan, joten peräkkäishaulle myös keskimääräisen tapauksen vaativuus on $O(n)$.

Keskimääräisen tapauksen analysointi on usein huomattavasti hankalampaa kuin pahimman tapauksen analysointi.

Joillain algoritmeilla voidaan keskimääräisen tapauksen vaativuus osoittaa pahimman tapauksen vaativuutta pienemmäksi.

Eräs tärkeä tällainen algoritmi on yksi käytännössä parhaista lajittelumenetelmistä, **pikalajittelu** (quicksort).

(Ilman virittelyä) pikalajittelun pahimman tapauksen vaativuus on $O(n^2)$, mutta sen keskimääräinen aikavaativuus on $O(n \log n)$ (ja vakiokertoimet ovat pieniä).

Lisäksi on ns. **amortisoitu** analyysi, jolla arvioidaan *pahinta tapausta kun työ jakautuu epätasaisesti*. Esimerkiksi hajatuksessa (hashing) yksi operaatio voi viedä kauan – johtaa taulun kasvattamiseen – mutta nopeuttaa seuraavia operaatioita.

Ongelman vaativuuden rajat

Onko algoritmiselle ongelmalle löydetty ratkaisualgoritmi riittävän hyvä?

Olisiko mahdollista löytää asymptoottisesti tehokkaampi ratkaisu, vai onko algoritmi **optimaalinen**?

Kysymyksiin vastaamiseksi meidän on arvioitava **laskentaongelman vaativuutta**:

Kuinka paljon aikaa (tai tilaa) ongelman P tapausten ratkaiseminen vaatii syötteen koon funktiona?

Löydetyn algoritmin A vaativuus muodostaa **ylärajan** ratkaistavan ongelman vaativuudelle:

Ongelma P voidaan ratkaista tarvitsematta ainakaan enemmän resursseja mitä algoritmi A vaatii.

Toisaalta saatamme pystyä todistamaan ongelman vaativuuden **alarajoja**:

Kaikki (tietyntyyppiset, esim. kiinnitettyihin perusoperaatioihin perustuvat) ongelman P ratkaiset algoritmit vaativat *vähintään* tietyn määrän aikaa tai tilaa.

Ks. Harel kuva 6.6.

Useimmat tunnetut alarajatodistukset koskevat jotakin *rajoitettua laskentamallia* jossa sallitaan vain tietyt perusoperaatiot tietoalkioille tai kontrollivuolle.

Esim. Järjestetystä taulukosta etsinnän alaraja

Voimme osoittaa, että binäärihaku on optimaalinen menetelmä arvon etsimiseen järjestetystä taulukosta:

Rajoitumme algoritmeihin, jotka kohdistavat syötteisiin, etsittävään arvoon ja taulukon alkioihin, ainoastaan kahden operandin välisiä vertailuja $<$, $\leq$, $=$, $>$ ja $\geq$. Eli alkion rakennetta (esim. bittihahmoa) *ei* saa käyttää!

Osoitamme, että jokainen tällainen algoritmi A suorittaa pahimmassa tapauksessa vähintään $\log_2 n$ vertailua:

Oletetaan, että haettava arvo Y esiintyy kasvavassa järjestyksessä olevassa taulukossa L[$1 \ldots n$].

Algoritmin A suorituksia yhdellä taulukolla L ja eri arvoilla Y voi kuvata **päätöspuulla**.

Ks. Harel kuva 6.7.

Päätöspuun haarautumasolmut vastaavat algoritmin tekemiä syötearvoihin kohdistuvia vertailuja, lehtisolmut algoritmin tuottamia tuloksia.

Vertailujen välissä algoritmi voi suorittaa jonon muita operaatioita.

Suoritukset haarautuvat kahteen vaihtoehtoon kunkin vertailun tuloksen perusteella (jos algoritmi ei tee turhia vertailuja), joten päätöspuu on binääripuu.

Yksittäinen suoritus = polku juuresta lehteen.

Päätöspuussa on oltava ainakin n lehteä: algoritmin on löydettävä Y jokaisesta taulukon $L[1 \ldots n]$ paikasta.

Binääripuussa, jonka korkeus on h, on enintään 2^h lehteä (helppo induktio)
$\rightsquigarrow$ n-lehtisen binääripuun korkeus on vähintään $\log_2 n$.

$\rightsquigarrow$ Jokin algoritmin A suoritus vaatii vähintään $\log_2 n$ vertailua.

$\rightsquigarrow$ Binäärihaku on asymptoottisesti optimaalinen algoritmi etsintään järjestetystä taulukosta.

Vastaavalla päätöspuuhun perustuvalla argumentoinnilla voidaan osoittaa, että arvojen vertailuun perustuva n-alkioisen listan tai taulukon lajittelu vaatii vähintään $\Theta(n \log n)$ vertailua.

$\rightsquigarrow$ Lomituslajittelu on eräs lukuisista asymptoottisesti optimaalisista lajittelumenetelmistä.

Muita esim. pika- (quick-) ja kekolajittelut (heapsort).

Monille ongelmille tunnetaan tehokkaita algoritmeja, joista osa on optimaalisia (vaativuus vastaa ongelman vaativuuden alarajaa).

Jatkossa näemme, että lukuisten ongelmien tarkka vaativuus on avoin: emme osaa ratkaista niitä tehokkaasti, vaikka tehokkaan ratkaisun olemassaolo on periaatteessa mahdollista.

5. Työläät ongelmat

(Harel luku 7)

"Eikä tätä asiaa saa hoidetuksi päivässä tai parissa"
[Esra 10:13]

• Riittääkö algoritmien tehokkuus kaikkien laskentaongelmien ratkaisemiseen?

Esim. Hanoin tornien ratkaisemisen työläys

Montako siirtoa Move(N,X,Y,Z)-proseduuri tulostaa?

Tarkastellaan suorituksen Move(N,X,Y,Z) kutsupuuta.

Puun jokaisessa solmussa tulostetaan yksi siirto. Paljonko solmuja on?

Lasketaan solmut kussakin syvyydessä.

(Solmun **syvyys** puussa on juuresta siihen johtavan polun pituus.)

Syvyydessä 0 on vain juuri Move(N,A,B,C).

Syvyydessä 1 on 2 solmua (Move(N-1,A,C,B) ja Move(N-1,C,B,A).

...

Lehtitasolla, syvyydessä $N-1$, on 2^{N-1} solmua.

$\rightsquigarrow$ yhteensä $1+2+\cdots+2^{N-1}=2^N-1$ solmua.

Moves-algoritmin aikavaativuus on **eksponentiaalinen!**

Jos esim. sekunnissa voidaan suorittaa miljoona Moves-kutsua, 64 kiekon ongelman siirtojen laskenta vie *yli puoli miljoonaa vuotta*!

Algoritmin eksponentiaalisuus on väistämätön, sillä sen tulosteen pituus on eksponentiaalinen.

Ongelman ratkaisu voi vaatia eksponentiaalisen ajan, vaikka tuloste olisikin lyhyt, esim. ”kyllä” /”ei”.

(Ns. **päätösongelmat**.)

Esim. ”Apinapalapeli”

Järjestettävä $n = m \times m$ neliönmuotoista korttia m-sivuiseksi neliöksi s.e. korttien värilliset kuviot osuvat kohdakkain.

Ks. Harel kuva 7.1.

Oletetaan, että kortteja ei saa pyörittää (niissä on vaikka taustana maisemakuva, joka kiinnittää niiden ylös-alas-suunnan).

Apinapalapelin päätösongelma: Voidaanko kortit järjestää halutulla tavalla?

Suoraviivainen ratkaisu:

Generoi kaikki mahdolliset järjestykset (esim. riveittäin alkaen alueen vasemmasta yläkulmasta). Ilmoita ”kyllä”, jos löytyy järjestys, jossa kaikkien vierekkäisten korttien reunat sopivat toisiinsa. Muuten ilmoita ”ei”.

Pahimmassa tapauksessa käytävä kaikki $n!$ järjestystä läpi.

Esim. 5×5-palapelin tapauksessa järjestyksiä on $25! = 1*2* \cdots * 25 \approx 15 * 10^{24}$. Jos tarkastetaan sekunnissa miljoona järjestystä, aikaa menee yli $491 * 10^{12}$ vuotta!

Apinapeliongelmaan (ja satoihin samankaltaisiin) ei tunneta pahimmassa tapauksessa oleellisesti tehokkaammin toimivaa ratkaisua.

Polynominen vs. ylipolynominen aika

Algoritmisen tehokkuuden vedenjakajana pidetään sitä, onko algoritminen ongelma

• ratkaistavissa **polynomisella** algoritmilla, ajassa $O(n^k)$ jollain kiinteällä k, vai

• vaatiiko sen ratkaiseminen (pahimmassa tapauksessa) **eksponentiaalisen** määrän (c^n, $c > 1$, tai enemmän) aikaa tai tilaa.

Ongelmia, joille on olemassa *polynomisessa* ajassa toimiva ratkaisualgoritmi, sanotaan **käytännössä ratkeaviksi** (tractable).

Ongelmia, joille on olemassa vain *eksponentiaalisen* ajan vaativia ratkaisualgoritmeja, sanotaan **työläiksi** (intractable).

Polynomisuuteen/ylipolynomisuuteen perustuva jako erottaa käytännössä käyttökelpoiset ja käyttökelvottomat algoritmit selvästi, vaikka teoriassa polynomisen algoritmin vaativuus voisi olla esim. $O(n^{1000})$.

Ks. Harel kuva 7.4.

Tarkastelujen perusteella algoritmiset ongelmat voidaan siis jakaa kahteen luokkaan: työläisiin ja käytännössä ratkeaviin.

Ks. Harel kuva 7.6.

Väliin jää **rajavyöhyke** jonka ongelmille *tunnetaan* vain eksponentiaalinen algoritmi vaikka polynominenkin *voi* olla olemassa (ja odottaa löytäjäänsä).

NP-täydelliset ongelmat

Kukaan ei ole osannut todistaa ylipolynomista alarajaa apinapalapelin vaativuudelle.

Toisaalta yhtään polynomista ratkaisualgoritmia ei tunneta.

Muita vastaavia **NP-täydellisiä** ongelmia tunnetaan satoja, useilta eri aloilta (mm. verkkoteoria, suunnittelu ja skedulointi, pelit, logiikka, ohjelmien optimointi).

Useat NP-täydelliset ongelmat ovat käytännössä esiintyviä keskeisiä ongelmia.

Tutustutaan muutamiin tyypillisiin NP-täydellisiin ongelmiin.

Kauppamatkustajan ongelma
(TSP, travelling salesman problem)

Annettuna painotettu verkko
(”kaupungit ja niiden väliset etäisyydet”).

Mikä on kokonaispituudeltaan lyhin reitti,
joka käy kertaalleen kussakin kaupungissa
(eli lyhin ns. **Hamiltonin kehä**)?

Vastaava päätösongelma: annettuna verkon
lisäksi maksimipituus K, onko verkossa
kauppamatkustajan reittiä pituudeltaan $\leq K$?

Harel kuva 7.9.

Huom: Kauppamatkustajan ongelma ei ole
”leikkiongelma” – tärkeä mm.
piirisuunnittelussa

Kauppamatkustajan päätösongelma voidaan ratkaista suoraviivaisesti kuten apinapalapeli: Generoi kaikki $N!$ kaupunkien järjestystä ja tarkasta kustakin, muodostaako se kehän, jonka pituus $\leq K$.

Kuten apinapalapelin tapauksessa, pahimmassa tapauksessa toivotonta esim. kun $N \geq 25$.

Kauppamatkustajan ongelmalle läheistä sukua on toinen NP-täydellinen ongelma, **Hamiltonin polku** -ongelma:

Onko annetussa verkossa polkua, joka kulkee täsmälleen kerran kunkin solmun kautta?

Ks. Harel kuva 7.10.

Pinnallisesti samankaltaisen ongelmien vaativuus voi kuitenkin erota huomattavasti:

Eulerin polku -ongelmassa kysytään, onko annetussa verkossa polkua, joka kulkee täsmälleen kerran jokaista verkon *kaarta* pitkin.

Ks. Harel kuva 7.11.

Voisi vaikuttaa, että tämänkin ongelman ratkaisemiseksi pitäisi pahimmillaan tarkastella kaikkia mahdollisia polkuja.

Kuitenkin voidaan osoittaa, että verkossa G on Eulerin polku (Euler 1736) täsmälleen silloin, kun
(1) G on yhtenäinen ja
(2) jokaisesta solmusta (mahdollisesti lukuunottamatta polun alkua ja loppua) lähtee parillinen määrä kaaria.

$\rightsquigarrow$ Eulerin polku -ongelma on ratkaistavissa tehokkaasti (ajassa $O(|E|)$, missä $|E|$ on verkon kaarien lukumäärä).

Lauselogiikan toteutuvuusongelma

Kenties keskeisin NP-täydellinen ongelma liittyy yksinkertaisten loogisten kaavojen toteutuvuuteen.

NP-täydellisyyden käsite määriteltiin vuonna 1971 juuri tämän ongelman kautta Stephen Cookin artikkelissa ”The Complexity of Theorem Proving Procedures” joka jatkoi loogikko Kurt Gödelin (välillä unohdettua) työtä.

Samalla nähdään yksinkertainen esimerkki kalvoilla 64–66 mainitusta loogisesta kielestä muoto- ja merkitysoppeineen.

Lauselogiikan formaalikieli koostuu *muuttujasymboleista* $\mathcal{X} = \{X_i : i \in \mathbb{N}\}$ ja induktiivisista *kaavanmuodostussäännöistä:*

Perustapaus: Jokainen muuttujasymboli on kaava.

Konjunktio: Jos A ja B ovat kaavoja, niin myös $(A \wedge B)$ on kaava.

Disjunktio: Jos A ja B ovat kaavoja, niin myös $(A \vee B)$ on kaava.

Negaatio: Jos A on kaava, niin myös $\neg A$ on kaava.

Implikaatio: Jos A ja B ovat kaavoja, niin myös $(A \to B)$ on kaava.

Merkitys annetaan
totuusarvojakeluilla $f\colon \mathcal{X} \mapsto \{\textbf{tosi}, \textbf{epätosi}\}$
jotka yleistyvät induktiolla:

- $f(A \wedge B)$ on tosi jos ja vain jos sekä $f(A)$ että $f(B)$ ovat **tosi**a.

- $f(A \vee B)$ on **tosi** jos ja vain jos edes toinen arvoista $f(A)$ tai $f(B)$ on **tosi**.

- $f(\neg A)$ on **tosi** jos ja vain jos $f(A)$ on **epätosi**.

- $f(A \rightarrow B)$ on **epätosi** jos ja vain jos $f(A)$ on **tosi** vaikka $f(B)$ on **epätosi**.

  (Ns. *materiaalinen* implikaatio, ei aina sama kuin semanttinen syy-seuraus-suhde.)

Toteutuvuusongelmassa (satisfiability, SAT) kysytään, onko annettu lauselogiikan kaava toteutuva, s.o, tuleeko se todeksi jollain totuusarvojakelulla.

Esim.

$$\neg(E \to F) \wedge (F \vee (D \to \neg E))$$

on toteutuva sijoituksella
$\{E :=$ **tosi**$, F :=$ **epätosi**$, D :=$ **epätosi**$\}$.
Toisaalta

$$\neg((D \wedge E) \to F) \wedge (F \vee (D \to \neg E))$$

ei ole toteutuva. (Miksi?)

Jos kaavassa on n muuttujaa, sen toteutuvuus on suoraviivaista tarkastaa tutkimalla mahdolliset 2^n totuusarvosijoitusta.

Toisaalta pahimmassa tapauksessa oleellisesti tehokkaampaa menetelmää ei tunneta.

Useat NP-täydelliset ongelmat liittyvät skedulointiin tai sovittamiseen:

Esim. lukujärjestyksen laatiminen
$\approx$ ns. kolmiulotteinen paritus.

NP-täydellisten ongelmien tyypillisiä piirteitä:

- Osaongelmien ratkaisemiseksi tehtävä valintoja, jotka estävät muita jatkovalintoja
  $\rightsquigarrow$ ahne lähestymistapa ei toimi.
- Osaongelmia on ylipolynominen määrä, esim. kaikki järjestykset ($n!$) tai kaikki osajoukot (2^n)
  $\rightsquigarrow$ dynaaminen ohjelmointi (osaongelmien ratkaisujen tallettaminen) ei toimi.
- Ei tunneta keinoja karsia osaongelmia etukäteen tutkimatta niitä.

NP-täydellisen ongelman ratkeavilla (”kyllä”-) tapauksilla on kuitenkin yksinkertainen (polynomisen tai usein lineaarisen kokoinen) **todiste**, josta on helppo vakuuttua ogelman tapauksen ratkeavuudesta.

Esimerkkejä todisteista: apinakorttien järjestys, polku, totuusarvosijoitus.

Todisteen tarkastamisen helppous tarkoittaa, että se on tehtävissä **polynomisessa ajassa**.

A onkin NP-ongelma jos ja vain jos on polynomi p ja polynominen algoritmi B joilla kysymys ”onko tapauksella x ongelman A vastaus **kyllä**” voidaan kääntää kysymykseksi ”onko olemassa jokin y kokoa p(x:n koko) jolla $B(x, y) =$ **kyllä**”.

Luokka NP

NP-täydelliset ongelmat ovat tehokkaasti (s.o. polynomisessa ajassa) ratkaistavissa **epädeterministisillä algoritmeilla**.

Epädeterministisillä algoritmeilla on käytössään (hypoteettinen) perusoperaatio **choose** A **or** B, joka valitsee käskyistä A ja B suoritettavaksi sen "oikean" joka aikanaan johtaa lopputulokseen **kyllä** (jos mahdollista).

Sen avulla algoritmi voi tehdä suorituksen onnistumisen kannalta parhaita mahdollisia ratkaisuja

mutta

se tarvitsee "kokonaisnäkemystä" laskennan tulevaisuudesta joten se ei enää olekaan paikallinen vain algoritmin nykytilan määräämä konekäsky!

Eräitä tapoja ymmärtää choose:

Onnenlantti jota heitettäessä saadaan aina oikea vastaus (jos sellainen on).

(Oikean valinnan n vaihtoehdon joukosta voi tehdä heittämällä lanttia $\log n$ kertaa.)

Jakaudutaan laskennassa A- ja B-haaroihin, jotka etenevät toisistaan riippumatta ('eri prosesseina") ja riittää, että jokin haara huutaa "kyllä!"

(Saadaan *epädeterministisen laskentapuun* käsite.)

Etsitään rekursiivisesti vaihtoehdot A ja B mutta suoritusajassa laskutetaan vain siitä vaihtoehdosta joka vastaa "kyllä".

(Vertaa todiste-ajatus.)

NP-täydelliset ongelmat voidaan ratkaista epädeterministisillä algoritmeilla seuraavalla periaatteella:

1. Generoi epädeterministisesti todiste ongelman tapauksen ratkeamisesta;

2. Tarkista, osoittaako todiste tapauksen ratkeavan; jos kyllä, palauta "yes", muuten palauta "no".

Todisteen pituus polynominen ja se voidaan tarkastaa polynomisessa ajassa $\leadsto$ kumpikin askel voidaan suorittaa polynomisessa ajassa.

(Polynomi q polynomista $p(n)$, $q(p(n))$, on polynomi. Esim
$5(3n^3 + 15n)^5 + 20(3n^3 + 15n)^2$.)

Esim. toteutuvuusongelma voidaan y.o. periaatteella ratkaista epädeterministisesti polynomisessa (itse asiassa jopa lineaarisessa ajassa):

1. Generoi epädeterministisesti totuusarvosijoitus annetun kaavan F muuttujille.

2. Sovella sijoitusta kaavan F muuttujiin ja tarkista onko tulos tosi.
Jos on, tulosta "yes", muuten tulosta "no".

Epädeterministisellä polynomisessa ajassa toimivalla algoritmilla ratkaistavat ongelmat muodostavat **ongelmaluokan NP**.

Merkitsemme esim. SAT $\in$ NP.

NP-täydellisten ongelmien ”täydellisyys”

Luokassa NP on (luultavasti)* muitakin ogelmia kuin NP-täydelliset ongelmat.

NP-täydelliset ongelmat ovat sikäli erittäin mielenkiintoisia, että

1. ne ovat työläydeltään ekvivalentteja: joko kaikki työläitä tai kaikki polynomisessa ajassa ratkeavia

2. ne ovat työläimpiä luokan NP-ongelmista (”täydellisiä”): jos *yksikin* NP-täydellinen ongelma on ratkaistavissa deterministisellä (s.o. ”normaalilla”) polynomisessa ajassa toimivalla algoritmilla, niin *kaikki* luokan NP ongelmat ovat käytännössä ratkeavia.

*Vaativuusteoriassa on lukuisia konjektuureja, joita ei ole pystytty todistamaan. Asioita tarkastellaan perusteellisemmin erityisesti kurssilla *Laskennan vaativuusteoria*.

Edellisiin ominaisuuksiin liittyy vaativuusteorian tunnetuin ja keskeisin avoin ongelma: Päteekö $\mathbf{P} = \mathbf{NP}$, missä **P** on polynomisessa ajassa deterministisesti ratkeavien ongelmien luokka. Emme tiedä, onko maaginen epädeterminismi välttämätöntä luokan NP ongelmien tehokkaaseen ratkaisemiseen. Yleisesti uskotaan, että P$\neq$NP.

Kuinka *kaikkia* luokan NP ongelmia koskevia väitteitä (2. yllä) voidaan todistaa?

Cook osoitti v. 1971 SAT-ongelman NP-täydellisyyden: Jos SAT osataan ratkaista deterministisesti polynomisessa ajassa, niin jokainen luokan NP ongelma osataan ratkaista deterministisesti polynomisessa ajassa.

Sen jälkeen muita ongelmia on osoitettu NP-täydellisiksi käyttäen **polynomista palautusta**.

Ongelmien palauttaminen toisiin(sa)

Miten kahden ongelman A ja B vaikeutta voidaan vertailla?

Jos molempien tarkka vaativuus tunnetaan, niin helppoa.

Mutta entä jos ei tunneta (kuten juuri NP-ongelmille)?

Yleinen periaate: *palautetaan* ongelman A ratkaiseminen toisen ongelman B ratkaisemiseen, ja päätellään siitä ettei A ole ainakaan vaikeampi kuin B.

Toisin sanoen, *oletetaan* että B osattaisiin ratkaista, ja osoitetaan että silloin myös A osattaisiin ratkaista "pienellä lisätyöllä".

Lisätyötä sallitaan *vähemmän* kuin ongelmien oletettu vaikeustaso.

Esimerkiksi ongelmat

$A \equiv$ "Onko elämällä tarkoitus?" ja
$B \equiv$ "Onko Jumala olemassa?"

ovat sellaisia, joiden ratkaisemisen vaikeutta emme tunne.

Mutta ongelma A voidaan palauttaa ongelmaksi B järkeilemällä "Jos Jumala olisi olemassa, niin Hän ei olisi luonut elämää vain huvin vuoksi, joten silloin elämällä olisi tarkoitus."

Siis A ei ole ainakaan vaikeampi kuin B.

Toisaalta voisi myös järkeillä että "jos elämällä olisi tarkoitus, niin jonkin on pitänyt se tarkoitus asettaa".

Silloin ongelma B vuorostaan palautuisi ongelmaksi A, eli ne olisivat *yhtä vaikeita*.

(Lisävaivaa ei käytetty liikaa, koska järkeilyn suoritti ihminen. . .)

Polynomiset palautukset

Algoritmisen ongelman A palautuksella ongelmaan B tarkoitetaan konstruktioa, joka osoittaa, että A on ratkaistavissa käyttämällä apuna B:n ratkaisevaa algoritmia, mikäli sellainen on olemassa.

Polynominen palautus päätösongelmasta A päätösongelmaan B on polynomisessa ajassa laskettava muunnos $f()$, jolla syöte x on ongelman A "kyllä"-tapaus joss $f(x)$ on ongelman B "kyllä"-tapaus.

Polynominen palautus ongelmasta A ongelmaan B tarkoittaa, että A on ratkaistavissa polynomisessa ajassa *mikäli* B on.

Tarkastellaan esimerkiksi Hamiltonin polku -ongelman palauttamista kauppamatkustajan ongelmaan.

Olkoon verkko G käsiteltävä Hamiltonin polku -ongelman tapaus; ratkaistavana siis, onko verkossa jokaisen solmun kautta täsmälleen kerran kulkevaa polkua.

Muutetaan G painotetuksi verkoksi G', jossa on kaari kaikkien verkon solmuparien välillä; jos kaari kuuluu alkuperäiseen verkkoon G, sen paino verkossa G' on 1, muuten sen paino on 2.

Ks. Harel kuva 7.12.

Muunnos voidaan (”selvästi”) tehdä polynomisessa ajassa.

Muunnos on palautus ongelmien välillä: Olkoon n solmujen lkm verkossa G. Verkossa G on Hamiltonin polku joss verkossa G' on kauppamatkustajan reitti, jonka pituus on enintään $n + 1$.

Mikäli TSP-ongelmalla olisi tai sille löydetään polynomisessa ajassa toimiva algoritmi, palautus ja k.o. algoritmi yhdessä muodostavat Hamiltonin polku -ongelman polynomisessa ajassa ratkaisevan algoritmin.

Ks. Harel kuva 7.13.

Monet ongelmien väliset palautukset ovat monimutkaisempia kuin edellinen, mutta niiden periaate on sama.

Yhteenvetona: Tekniseltä kannalta ongelma A on NP-täydellinen, jos

1. A kuuluu luokkaan NP eli on ratkaistavissa epädeterministisellä polynomisessa ajassa toimivalla algoritmilla, ja

2. jokaisen luokan NP ongelman ratkaiseminen voidaan palauttaa polynomisessa ajassa ongelman A ratkaisemiseen.

Jälkimmäisestä ominaisuudesta sanotaan, että A on **NP-kova** (NP-hard).

NP-kovuus osoitetaan yleensä palautuksella jostain aiemmin NP-täydelliseksi (eli -kovaksi) tiedetystä ongelmasta.

NP-täydellisten ongelmien approksimointi

Useat NP-täydelliset päätösongelmat (esim. TSP) ovat kyllä/ei-versioita **kombinatorisista optimointiongelmista**.

Optimointiongelmat eivät ole niitä vastaavia päätösongelmia helpompia, joten niitäkin kutsutaan NP-täydellisiksi.

NP-täydellisten optimointiongelmien tarkkoja ratkaisuja pyritään **approksimoimaan**.

Esim. Geometrinen TSP.

Piirisuunnittelussa esiintyvä ongelma: Annettuna n pistettä tasossa, mikä on lyhin kunkin pisteen kautta kertaalleen kulkeva kehä? (Jokaisen pisteparin välillä mahdollinen yhteys, jonka pituus normaali geometrinen etäisyys.)

Ongelma voidaan osoittaa NP-täydelliseksi.

Polynomisessa ajassa voidaan (ns. Christofidesin algoritmilla) löytää kehä, joka on enintään 50% optimaalista pidempi

Enintään 2 $\times$ optimaalisen pituinen kehä C voidaan löytää seuraavalla periaatteella:

1. Muodosta annetun verkon pienin virittävä puu T.

2. Muodosta polku P kulkemalla kukin puun T kaari kahdesti.

3. Muuta polku P Hamiltonin kehäksi C oikaisemalla jo vierailtujen solmujen ohi.

Approksimoidun kehän pituus voidaan arvioida seuraavasti:

Jos C^* on lyhin mahdollinen Hamiltonin kehä, virittävän puun T kokonaispituus $L(T)$ on enintään sen kokonaispituus $L(C^*)$: T on halvin tapa yhdistää kaikki solmut puuksi, kukin ($C^* -$ yksi kaari) taas jokin tapa yhdistää ne poluksi.

Polun P pituus on $2L(T)$, ja koska polun P oikaiseminen kehäksi C ei voi pidentää polkua,

$$L(C) \leq L(P) = 2L(T) \leq 2L(C^*)$$

Tällä periaatteella löydetyn Hamiltonin kehän C pituus on siis enintään kaksi kertaa lyhimmän Hamiltonin kehän C^* pituus.

Todistettavasti työläät ongelmat

NP-täydellisten ongelmien täsmällinen vaativuus on avoin ongelma, vaikka niiden uskotaan melko yleisesti olevan aidosti työläitä.

Muita ongelmia on pystytty *todistamaan* työläiksi osoittamalla niillä olevan eksponentiaalisia aikavaativuuden alarajoja.

Tällaisia esimerkiksi yleistetyt (N$\times$N-laudalla pelattavat) lautapelit kuten shakki, tammi ja Go.

Myös ohjelmien verifiointiin liittyvä dynaaminen lauselogiikka (PDL, propositional dynamic logic).

Ongelmien vaativuusluokat

Ongelmat, jotka ovat ratkaistavissa deterministisesti tai epädeterministesti polynomisessa ajassa sekä NP-täydelliset ongelmat muodostavat vastaavat **ongelmaluokat** P (joskus PTIME), NP (joskus NPTIME) ja NPC.

Ongelmaluokkien välisten suhteiden selvittäminen muodostaa aktiivisen **laskennan vaativuusteorian** tutkimuskentän.

Tarkastelemalla logaritmisella (LOG), polynomisella (P) tai eksponentiaalisella (EXP) määrällä aika- (TIME) tai tilaresursseja (SPACE) ratkaistavissa olevia ongelmia saadaan erilaisia ongelmien vaativuusluokkia. Epädeterminismin salliminen (N) saattaa lisäksi muuttaa annetuilla resursseilla ratkaistavissa olevien ongelmien joukkoa.

Vaativuusluokkien yleisiä suhteita tiedetään, mutta monet kysymykset ovat avoimia.

Ks. Harel kuva 7.15.

Esimerkiksi on varsin selvää, että sallimalla epädeterminismi polynomisessa ajassa ratkeavien ongelmien joukko ei ainakaan suppene, ja että polynomisessa ajassa ratkaistavat ongelmat eivät voi vaatia ylipolynomista muistitilaa. Vaativuusluokkien suhteina ilmaistuna

$$\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE}.$$

Kuitenkaan ei tiedetä, onko jompikumpi tai kumpikaan y.o. sisältyvyyksistä aito.

Kalvolla 126 esiteltiin NP todisteiden avulla: $A \in$ NP jos ja vain jos on $B \in$ P ja polynomi p siten, että

$$x \in A \Leftrightarrow \underline{\exists \text{ bittijono } y.\, |y| \leq p(|x|)} \wedge \langle x, y \rangle \in B.$$

Negaatio eli komplementti synnyttää luokan co-NP: $\bar{A} \in$ co-NP jos ja vain jos on $B \in$ P ja polynomi p siten, että

$$\begin{aligned} x \in \bar{A} &\Leftrightarrow \neg\exists y.\, |y| \leq p(|x|) \wedge \langle x, y \rangle \in B \\ &\Leftrightarrow \forall y.\, |y| \leq p(|x|) \rightarrow \underbrace{\langle x, y \rangle \notin B}_{\bar{B} \in \mathsf{P}} \end{aligned}$$

Operaatio <u>"on polynominen todiste"</u> ja komplementointi synnyttävät *polynomisen hierarkian* luokkien P ja PSPACE välimaastoon.

Esimerkiksi

$$\exists z.\, |z| \leq q(|x|) \wedge \forall y.\, |y| \leq p(|x|) \rightarrow \langle x, y, z \rangle \notin B$$

antaa ongelman joka on tämän hierarkian *toisen tason positiivisella puolella* Σ_2^p:

- Taso 0 on $\mathsf{P} = \Sigma_0^p = \Pi_0^p$.

- Tason $n+1$ positiivinen puoli Σ_{n+1}^p saadaan edellisen tason n negatiivisesta puolesta Π_n^p lisäämällä uusi polynominen todiste. Siis $\mathsf{NP} = \Sigma_1^p$.

- Tason $n+1$ negatiivinen puoli Π_{n+1}^p saadaan saman tason $n+1$ positiivisesta puolesta Σ_{n+1}^p komplementoimalla. Siis co-$\mathsf{NP} = \Pi_1^p$

Kiinnostava esimerkiksi peleissä: ”minulla on siirto, jonka jälkeen kaikilla sinun siirroillasi minä olen voittanut”.

Vaativuusluokkien määrittelyt ja niitä koskevat tulokset (esim. Cookin teoreema) perustuvat algoritmien mahdollisimman yksinkertaisiin perusmalleihin, erityisesti ns. Turingin koneisiin. Tutustumme näihin myöhemmin.

NP-täydellisiä ongelmia ei siis *osata* ratkaista *tehokkaasti*, ja

todistettavasti vaativia ongelmia ei *pystytä* ratkaisemaan *tehokkaasti*.

Seuraavaksi kohtaamme hyvin määriteltyjä algoritmisia ongelmia, joita ei *pystytä lainkaan* ratkaisemaan, vaikka resursseja olisi käytössa mielivaltaisesti.

6. Ratkeamattomat ongelmat

(Harel luku 8)

"tämä työ on sinulle liian raskas" [2. Moos. 18:18]

• On myös ongelmia, joita ei *lainkaan* pystytä ratkaisemaan algoritmisesti!

Algoritmista rakeamattomuutta käsitellään perusteellisemmin matematiikassa *Laskettavuuden teorian* tai Rekursioteorian nimikkeellä.

Suoraviivaisella laskenta-argumentilla voi nähdä, että kaikilla laskentaongelmilla ei ole ratkaisualgoritmia:

Tarkastellaan päätösongelmia.

Sekä syötteet että ohjelmat (algoritmit) voidaan pohjimmiltaan esittää binäärijonoina.

Algoritmeja/ohjelmia ja syötteitä on siten *numeroituvasti* (”yhtä paljon kuin luonnollisia lukuja”).

Päätösongelmat ovat syötejonojoukkojen osajoukkoja

$$\{x \in \{0,1\}^n \mid x \text{ on ongelman ”yes-tapaus}\}.$$

Erilaisia päätösongelmia on siten *ylinumeroituvasti* (”yhtä paljon kuin reaalilukuja”).

Siis kaikilla päätösongelmilla ei voi olla ratkaisualgoritmia (vaikkei aika- tai tilaresursseja rajoitettaisi lainkaan).

$\rightsquigarrow$

Onko ihmisen ongelmanratkaisukyky rajallinen?

Onko ihminen algoritmi tai joukko algoritmeja?

Ainakin ihmisen aika- ja muistiresurssit ovat kovin rajalliset!

Sama päättely ”graafisesti”:

Laaditaan ääretön taulukko

	syöte 1	syöte 2	syöte 3	$\cdots$
ohjelma 1	$\pm$	$+$	$-$	$\cdots$
ohjelma 2	$-$	$\mp$	$+$	$\cdots$
ohjelma 3	$+$	$+$	$\mp$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Käännetään merkit sen *diagonaalilla* ympäri.

Löytyykö tämä käännetty diagonaali taulukon miltään riviltä r?

Ei, koska rivin r syöte r oli myös käännetty!

Algoritmisesti ratkeamattomia ongelmia siis on välttämättä olemassa.

Ratkeamattomat ongelmat eivät pelkästään teoreettisia kuriositeetteja, vaan osa myös käytännössä relevantteja.

Tutustutaan muutamiin ratkeamattomiin ongelmiin.

Tason kaakelointi

Laajennus apinapalapeliongelmasta.

Annettuna joukko (kiinteästi suunnattuja, s.o. ei pyöriteltäviä) kaakelilaattojen malleja.

Kaakeleissa kukin lävistäjien erottamista neljästä alueesta jonkin värinen (tai yhtä hyvin värillinen apinan puolikas, kuten aiemmin).

Harel kuvat 8.1 ja 8.2.

Voiko annetun tyyppisillä laatoilla peittää minkä tahansa äärellisen alueen siten, että toisiaan koskettavien reunojen väri on sama?

Kaakelointiongelma (monine variaatioineen) on algoritmisesti **ratkeamaton** (noncomputable, undecidable):

Sen ratkaisevaa algoritmia ei ole (*eikä voikaan olla olemassa*).

Palaamme kaakelointiongelmaan Turingin koneiden yhteydessä.

Kuvamme algoritmisten ongelmien kentästä on laajentunut, ks. Harel kuva 8.3.

Rajoittamattomuus ja ratkeamattomuus

Ratkeamattoman ongelman ratkaisemisyritykset edellyttävät rajoittamattoman monien tapausten tutkimista: jos tapauksia olisi rajoitettu määrä, ratkaisualgoritmi voisi tutkia ne äärellisessä ajassa.

Toisaalta rajoittamattomuudesta ei välttämättä seuraa ratkeamattomuus.

Esim. Dominopolkuongelma.

Annettuna joukko kaakelityyppejä ja kaksi tason pistettä V ja W, voidaanko muodostaa V:stä W:hen "dominopolku", jossa peräkkäisten laattojen toisiaan koskettavat reunat samanväriset?

Ks. Harel, kuva 8.4.

Jos käytettävissä äärellinen alue, ongelma on selvästi ratkeava. (Miksi?)

Voidaan osoittaa, että ongelma on ratkeamaton, jos käytettävissä on puolikas taso, mutta *ratkeava, jos ”dominopolku” saa käyttää hyväkseen koko tasoa!*

Kontekstittomien kielten ekvivalenssi

Ohjelmointikielten syntaksi eli muotosäännöt määritellään usein **kontekstittomilla kieliopeilla** (tai jollain niiden variaatioilla)

Esim.

$$
\begin{array}{lll}
\text{Statement} & \rightarrow & \text{if (Expr) Statement} \\
& | & \text{while (Expr) Statement} \\
& & \ldots \\
\text{Expr} & \rightarrow & \text{(Expr)} \\
& | & \text{Expr BinOp Expr} \\
& & \ldots
\end{array}
$$

Annettuna kaksi kontekstitonta kielioppia, kuvaavatko ne saman kielen?

Ongelma on algoritmisesti ratkeamaton.

Pysähtymisongelma

Pysähtyykö annettu algoritmi/ohjelma annetulla syötteellä?

Keskeisin ratkeamaton ongelma – monien muiden ongelmien ratkeamattomuus palautuu pysähtymisongelman ratkeamattomuuteen.

Joskus terminoivuus on helppo nähdä:

Esim. Algoritmi A:

(1) **while** X$\neq$1 **do** X:= X-2;

Pysähtyy jos ja vain joss syöte X on pariton positiivinen kokonaisluku.

Toisinaan erittäin vaikeaa:

Esim. Algoritmi B:

(1) **while** X≠1 **do**
(1.1) **if** X parillinen **then** X:= X/2;
(1.2) **else** X:= 3*X+1;
endwhile

Algoritmi B päättyy kaikilla positiivisilla kokonaisluvuilla X, *joilla sitä on kokeiltu*.

Kukaan ei ole pystynyt todistamaan sen päättymistä *kaikilla* pos. kokonaisluvuilla X.

Osoitetaan seuraavaksi, että yleinen pysähtymisongelma on algoritmisesti ratkeamaton.

Tyypilliseen tapaan jonkin asian mahdottomuus osoitetaan epäsuorasti, osoittamalla että asian mahdollisuudesta seuraa looginen ristiriita.

Tehdään vastaoletus:
On olemassa algoritmi Pysähtyy(R,X), joka

- saa syötteenään algoritmin R ja sen syötteen X
- palauttaa arvon **true**, jos R(X) pysähtyy, ja muuten arvon **false**.

Muodostetaan algoritmi S(W):

if Pysähtyy(W,W) **then while true do**
{ /* ei mitään, pysytään silmukassa! */ }
return false;

Päättyykö evaluointi S(S)?

S(S) pysähtyy täsmälleen, kun
Pysähtyy(S,S)=**false**, eli kun
S(S) *ei pysähdy!*

Ristiriita $\leadsto$ vastaoletus oli väärä.

Algoritmia Pysähtyy(R,X) ei voi olla olemassa.

Pysähtymisongelman vaihtoehtoinen ratkeamattomuustodistus: Cantorin **diagonalisointiargumentti**, ks. kalvo 151 tai Harel sivut 210–211.

Ongelman Q NP-kovuus osoitetaan *polynomisella* palautuksella jostain toisesta NP-kovasta ongelmasta P. (Palautus osoittaa, että *jos* Q osattaisiin ratkaista polynomisessa ajassa, niin sama pätisi myös ongelmaan P.)

Vastaavasti ongelman Q ratkeamattomuus osoitetaan *rajoittamattomalla algoritmisella* palautuksella jostain ratkeamattomaksi tiedetystä ongelmasta P

(usein juuri pysähtymisongelmasta).

Tässä palautus perustuu oletukseen, että meillä olisi (hypoteettinen) ongelman Q ratkaiseva algoritmi A_Q, jota voisi käyttää ns. **oraakkelina** eli aliohjelmana, joka muodostaa osan ongelman P ratkaisevasta algoritmista.

(Polynominen palautushan oli käännös kielestä toiseen, ei aliohjelma jota voi kutsua useasti ja jonka jälkeen voi jatkaa edelleen pääohjelmassa.)

Ongelmalle P hahmotellaan ratkaisualgoritmi käyttäen algoritmia A_Q aliohjelmana.

Jos P on ratkeamaton, yllä hahmoteltu konstruktio osoittaa, että myöskään ongelmalla Q ei voi olla ratkaisualgoritmia A_Q.

Verifiointiongelman ratkamattomuus

Aiemmin väitimme, että algoritmien verifiointiongelma on ratkeamaton.

Perustellaan asia palautuksella pysähtymisongelmasta:

Vastaoletus: On olemassa algoritmi (oraakkeli) A, joka saa syötteenä ongelman spesifikaation P ja ohjelman R. Suoritus A(P,R) päättyy, ja sen paluuarvo on tosi joss R on spesifikaafion P mukainen täysin oikeellinen ohjelma.

Nyt pysähtymisongelman tapaus (R, X) voidaan ratkaista verifiointioraakkelin A avulla:

(1) C:= A(”Input=X; Output=anything”, R);
(2) **if** C = **true return** ”R(X) pysähtyy”;
(3) **else return** ”R(X) ei pysähdy”;

Rivillä (1) käytetty spesifikaatio ei vaadi osittain oikeellisuudelta mitään, joten oraakkeli A vain tarkistaa, pysähtyykö R syötteellä X.

Koska pysähtymisongelma on ratkeamaton, ylläoleva ”algoritmi”on mahdoton.

$\rightsquigarrow$ verifiointiongelmalla ei ole ratkaisualgoritmia.

Osittain ratkeavat ongelmat

Pysähtymisongelma ei tunnu täysin ratkeamattomalta: Annettua algoritmia R voidaan simuloida annetulla syötteellä X ja katsoa, pysähtyykö se. Jos suoritus päättyy, vastaus on ”kyllä”.

Ongelmallista on tietää, kannattaako simulointia vielä jatkaa vai onko ohjelma R esim. ikuisessa silmukassa.

Pysähtymisogelman ”kyllä”-tapauksilla on siis äärellinen ja äärellisessä ajassa tarkastettava **todiste** (certificate), nimittäin päättyvän suorituksen jäljitys.

Ongelmat, joiden ”kyllä-tapauksilla on tällainen todiste, ovat **osittain ratkeavia** (partially decidable).*

*Rekursioteoriassa tällaisia ongelmia kutsutaan **rekursiivisesti lueteltaviksi** (recursively enumerable).

Osittain ratkeavat ongelmat voidaa osoittaa laskettavuuden kannalta ekvivalenteiksi: jokainen niistä on palautettavissa muihin osittain ratkeaviin ongelmiin.

Huom: Verifiointiongelma on ”voimakkaammin ratkeamaton” ongelma kuin pysähtymisongelma:

Pysähtymisongelma voidaan palauttaa verifiointiongelmaan, mutta ei päinvastoin.

Verifiointiin sisältyy pysähtymisongelmaa vaikeampi terminointiongelma (eli **totaalisuusongelma**): Pysähtyykö R *kaikilla* syötteillään?

Ratkeamattomat ongelmat muodostavat vastaavan (mutta täsmällisemmin tunnetun) monitasoisen hierarkian kuin työläät ongelmat.

7. Laskennan perusmallit

(Harel luku 9)

"Puhun vielä sen jäsenistä, sen ihmeellisestä rakenteesta." [Job 41:4]

- Mikä on yksinkertaisin mutta riittävän voimakas malli algoritmien esittämiseen?

Algoritmien yksinkertaistettuja malleja tarkastellaan perusteellisemmin kursseilla *Ohjelmoinnin ja laskennan perusmallit* sekä *Laskennan teoria*.

Käsiteltävän datan malli?

Ohjelmien tietokonetoteutuksissa kaikki data (merkit, muuttujien arvot, tietorakenteet) on pohjimmallaan peräkkäiseen muistiin sijoitettuja bittien jonoja.

$\rightsquigarrow$ Riittää pystyä käsittelemään binäärijonoja.

Mukavuussyistä käytetään laajempia **aakkostoja** (esim. ASCII-merkkejä tai kymmenjärjestelmän numeroita 0,...,9).

Algoritmien käsittelemät arvot ja tietorakenteet voivat kasvaa.

$\rightsquigarrow$ käsitellään dataa *rajoittamattoman* pituisella **nauhalla** (”potentiaalinen”, ei ”aktuaalinen” äärettömyys).

Nauhalla voi kussakin positiossa olla yksi aakkoston merkki.

HUOM! Laskennan *tehokkuuteen* vaikuttaa montako ”datamerkkiä” on käytössä tarvittavien ”välimerkkien” lisäksi: unaarilukujärjestelmä (”tukkimiehen kirjanpito”) on eksponentiaalisesti pidempää kuin binääri- ja muut.

Minimaalinen kotrollirakenne?

Tarkastelemissamme algoritmeissa (ja normaaleissa tietokoneohjelmissa) implisiittinen oletus, että algoritmia suorittava **prosessori** on kullakin hetkellä jossain *kohdassa* algoritmia.

Voidaan mallintaa **tiloina**: algoritmi on kussakin vaiheessa täsmälleen yhdessä tilassa.

Kontrollin eteneminen riippuu muuttujien ja tietorakenteiden arvoista: (**if** N=0 **then** ... **else**..., **while** I $<$ N **do** jne.)

Mallinnetaan kontrollin ohjaus minimaalisella tavalla: Algoritmeilla on käytössään yhtä nauhapositioa kerrallaan katsova **lukupää**.

Algoritmin seuraava tila määräytyy lukupään kohdalla olevan nauhamerkin perusteella, samoin se, siirtyykö lukupää seuraavaan vai edelliseen merkkiin.

Harel kuva 9.3.

Lisäksi algoritmit muuttavat tietorakenteitaan. Toteutetaan tämä sallimalla lukupään ennen siirtymistään kirjoittaa kohdalla olevan merkin tilalle uusi merkki (eli se on **luku- ja kirjoituspää**).

Kyseinen minimaalinen algoritmien esitysmalli on **Turingin kone** (Alan Turing, 1936).

Turingin kone

Hieman tarkemmin: **Turingin kone** (**TM**) koostuu

- äärellisestä joukosta **tiloja**,
- äärellisestä **aakkostosta**,*
- äärettömästä **nauhasta** ja
- nauhaa merkkipositio kerrallaan käsittelevästä **nauhapäästä**

+ **siirtymäfunktiosta**, joka tilan ja nauhapään kohdalla olevan merkin perusteella määrää 1) seuraavan tilan, 2) merkkipositioon kirjoitettavan uuden merkin ja 3) nauhapään siirron oikealle tai vasemmalle.

≈ ”Turingin koneen ohjelma”

Siirtymäfunktio voidaan esittää tilasiirtymäkaaviona (Ks. Harel kuva 9.4).

*Käytetään merkkiä # esittämään puuttuvaa tai poistettua merkkiä

Tilat, joista ei siirtymiä eteenpäin, ovat **lopputiloja**.

Suorituksen alku: kone nimetyssä **alkutilassa** ja nauhapää syötteen ensimmäisen merkin kohdalla. Suoritus etenee siirtymäfunktion ohjaamana, kunnes saavuttaa lopputilan. Kone hyväksyy tai hylkää syötteen sen mukaan, onko lopputila YES vai NO.

Esim. Palindromien tunnistus
(Harel sivut 229–230).

Turingin kone ei ole rajoittunut päätösongelmien ratkaisemiseen. Voidaan sopia, että suorituksen päätyttyä nauhalla (esim. !-merkkien välissä) on sen laskema tuloste.

Esim. Kymmenjärjestelmän lukujen yhteenlasku (Harel sivu 232).

Nauhalla syötteenä X'+'Y

Siirtymäfunktion periaate erittäin karkeasti:

Kirjoita X:n vasemmalle puolelle merkki '!'

Kunnes jompikumpi luku loppuu, toista:

Paikanna ja poista luvun Y viimeinen numero y (siirry sen arvoa vastaavaan tilaan *nro*y, $y \in \{0, \ldots, 9\}$).

Paikanna ja poista luvun X viimeinen numero x sekä siirry arvoa $x + y$ vastaavaan tilaan *sum0*, *sum0+muistinro*, ... tai *sum9+muistinro*.

Kirjoita summan numero syötettä lähinnä edeltävään tyhjään positioon.

Toista samaan tapaan, muistaen (tiloissa) onko muistinumero otettava huomioon.

Jos toinen luku loppuu, toisen jäljelle jääneet numerot siirrettävä yksitellen (tarpeen vaatiessa muistinumero lisäten) keskeneräisen tulosteen alkuun.

Lopuksi lisää '!'-merkki tulostetta edeltävään nauhapositioon.

Esimerkkisuoritus (tila nauhan sisällön vieressä):

```
... # # # # # 8 9 + 1 2 3 # # ...
... # # # # ! 8 9 + 1 2 3 # # ...
... # # # # ! 8 9 + 1 2 # # # ... (nro3)
... # # # # ! 8 # + 1 2 # # # ... (sum2+muistinro)
... # # # 2 ! 8 # + 1 2 # # # ...
... # # # 2 ! 8 # + 1 # # # # ... (nro3; 2+muistinro)
... # # # 2 ! # # + 1 # # # # ... (sum1+muistinro)
... # # 1 2 ! # # + 1 # # # # ...
... # # 1 2 ! # # + # # # # # ... (nro2; 1+muistinro)
... # 2 1 2 ! # # + # # # # # ... (luvut-loppu)
... ! 2 1 2 ! # # + # # # # # ... (YES)
```

Mahdollista, mutta ohjelmointina *erittäin hankalaa!*

Miksi siis tarkastellaan Turingin koneita (tai muita primitiivisiä malleja)?.

Yksinkertaisissa malleissa mahdollista keskittyä algoritmien oleellisiin piirteisiin.

Laskettavuuden ja formaalikielten teorian sekä vaativuusteorian keskeiset tulokset perustuvat Turingin koneiden tarkasteluun.

Esim. Cookin teoreema (SAT on NP-täydellinen).

Vaativuusluokkien (P, NP, EXP jne.) täsmällinen määrittely perustuu Turingin koneisiin:

aikavaativuus = montako tilasiirtymää suoritettava

tilavaativuus = montako nauhapositiota käytettävä.

Church-Turingin teesi

Onko primitiivinen Turingin kone sitten riittävän voimakas laskennan malli?

Algoritmien esittämiseen on esitetty lukuisia malleja: mm. Churchin lambda-kalkyyli, Postin produktiosysteemit, Kleenen rekursiiviset funktiot ja nykyaikaiset korkean tason ohjelmointikielet.

Voidaan osoittaa, että että kaikki algoritmien esitystavat ovat yhtä voimakkaita. (Tätä varten ohjelmointikieliä idealisoitava siten, että muuttujien arvoina saa olla rajoittamattoman suuria arvoja.)

Perustuu havaintoon, että missä tahansa mallissa esitetyn algoritmin toimintaa voidaan **simuloida** jotain toista mallia käyttäen.

$\rightsquigarrow$ **Church-Turingin teesi** (CT-teesi):
Intuitiivinen algoritminen laskettavuus =
laskettavuus Turingin koneella.

"Intuitiivinen algoritminen laskettavuus" ei ole täsmällinen käsite, mutta täysin erilaisten algoritmisten mallien havaittu ekvivalenssi tukee CT-teesiä.

Huom: CT-teesin nojalla esimerkiksi korkean tason kielellä johtamamme pysähtymättömyyongelman ratkeamattomuustulos siirtyy kaikkiin muihin (riittävän voimakkaisiin) laskennan malleihin.

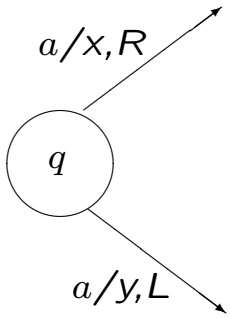
Laskettavuuden käsite on **robusti**, käytetystä mallista riippumaton.

Turingin koneen variaatiot

Turingin koneista on useita variaatioita, jotka voidaan nähdä laskentavoimaltaan yhteneviksi: ainoastaan toiseen suuntaan ääretön nauha, useampia kuin yksi nauha, kaksiulotteinen nauha, jne.

Harel, kuva 9.6.

Erityisen kiinnostava on **epädeterminstinen Turingin kone** (NTM), jolla on yksikäsitteistä siirtymäfunktioita yleisempi **siirtymärelaatio**: Yhdestä tilasta q voi lähteä samalla nauhamerkillä a useampia siirtymiä:



Epädeterministisen Turingin koneen voi jälleen ajatella valintatilanteissa ”maagisesti” arvaavan YES-lopputilaan lopulta johtavat siirtymät, mikäli sellaiset on olemassa.

Epädeterminismi ei kuitenkaan laajenna periaatteellista laskettavuutta: Deterministisellä TM:llä voi (vaikkakin työläästi) *simuloida* epädeterministisen TM:n vaihtoehtoisia laskentoja ja tutkia, johtaako joku niistä YES-lopputilaan.

Jos kiinnitetään (aakkosto sekä) esitystapa Turingin koneiden (a) siirtymäfunktioille ja (b) konfiguraatioille (tilalle, pään paikalle ja käsiteltyjen nauhapaikkojen sisällöille), niin voidaan tehdä Turingin kone, joka tuottaa syötteistä (a) ja (b) joko (i) seuraavan konfiguraation tai (ii) tiedon ettei sellaista ole. Saadaan **universaalikone** tulkkaamaan muita(kin) Turingin koneita!

Mallien väliset simuloinnit

Perustellaan hieman korkean tason ohjelmointikielten ja Turingin koneiden laskentavoiman yhtenevyyttä.

Kohtalaisen helppo uskoa, että korkean tason ohjelmointikielellä voi simuloida Turingin koneiden toimintaa – opetuskäyttöön on ohjelmoitukin Turingin koneiden tulkkeja. (Täyttä simulointia varten täytyy tietysti olettaa idealisoitu ohjelmointikieli, jossa muistia on käytössä rajatta.)

Perustellaan hieman päinvastaista suuntaa, tietokoneohjelmien simulointia Turingin koneella.

Tietokoneohjelman suorituksen pohjana on konekielinen ohjelma, joko jotain prosessoria varten käännetty versio ohjelmasta tai sitä tulkitsevan ohjelman (esim. Java-virtuaalikoneen) koodi.

Konekieliohjelmat koostuvat yksinkertaisista konekäskyistä, jotka voivat käsitellä koneen muistipaikkojen sisältöä.

Laskennan ns. **RAM**-malli. (Tarkemmin ks. esim. Aho, Hopcroft & Ullman: Design and analysis of computer algorithms, luku 1. Addison-Wesley 1974.)

Yksityiskohdat ohittaen:

RAM-koneen muistipaikkojen sisältöjä voidaan ylläpitää TM:n nauhalla.

RAM-koneen konekieliohjelmaa vastaava siirtymäfunktio simuloi yksittäisten käskyjen suoritusta ja kontrollin kulkua ohjelmassa.

Muistipaikkojen sisällön paikantaminen ja muuttaminen nauhalla on hieman hankalaa, mutta ei liian työlästä: voidaan osoittaa, että simulointi vie enintään *polynomisen ajan* suhteessa RAM-koneen käyttämään aikaan.

Vastaava pätee (tietyin oletuksin) muidenkin algoritmisten mallien välisiin simulointeihin: Jos algoritmi toimii jossain laskennan perusmallissa polynomisessa ajassa, sitä voidaan simuloida myös muiden mallien mukaisilla algoritmeilla polynomisessa ajassa.

Siis myös ongelmien jako työläisiin (eksponentiaalisen ajan vaativiin) ja käytännössä (eli polynomisessa ajassa) ratkeaviin on robusti algoritmien esitystavan suhteen.

Huom: ”Alipolynomisissa” aikavaativuuksissa voi olla eroja. Korkean tason kielellä lineaarisessa ajassa suoritettavan algoritmin simulointi Turingin koneella voi vaatia esim. ajan $\Theta(n^2)$.

Cookin teoreema

Turingin koneiden pääasiallinen merkitys on ongelmien alarajatarkasteluissa (työläyden tai ratkeamattomuuden osoittamisessa).

Kuinka esimerkiksi voidaan sanoa, että jos lauselogiikan toteutuvuusongelma SAT voidaan ratkaista deterministisesti polynomisessa ajassa niin *mikä tahansa* epädeterministisesti polynomisessa ajassa ratkeava päätösongelma A voidaan ratkaista deterministisesti polynomisessa ajassa?

(Ts. SAT-ongelman NP-kovuus; Cook 1971)

Miten lausekalkyylin kaavan toteutuvuus liittyy sen ratkaisemiseen, onko syöte x ongelman A "kyllä"-tapaus?

Jos $A \in$ NP niin ongelma "$x \in A$" on ratkaistavissa jollain epädeterministisellä Turingin koneella M polynomisessa ajassa $p(|x|)$.

Turingin koneen yksinkertaisuuden takia on mahdollista (deterministisesti ja polynomisessa ajassa) muodostaa kaava F_x, joka kuvaa koneen M mahdollisia laskentoja syötteellä x; suoritus päättyy tilaan "kyllä" jos ja vain jos F_x on toteutuva.

Suoritusajan (ja -tilan) ylärajan $p(|x|)$ tunteminen "etukäteen" sallii *lause*logiikan käytön; vastaava "ei-äärellinen" tekniikka palauttaa pysähtymisongelman *1KL:n* toteutuvuuteen – ks. esim. G.S. Boolos & R.C. Jeffrey: Computability and Logic (3. painos), luku 10 (Cambridge University Press 1989).

Siis mikä tahansa NP-ongelma voidaan palauttaa polynomisesti ongelmaan SAT (eli SAT on NP-kova).

Kaakelointiongelman ratkeamattomuus

Tason kaakeloimisen ja Turingin koneen suorittamisen välillä on yllättävä yhteys, jonka avulla voidaan osoittaa kaakelointiongelman ratkeamattomuus palautuksella pysähtymisongelmasta.

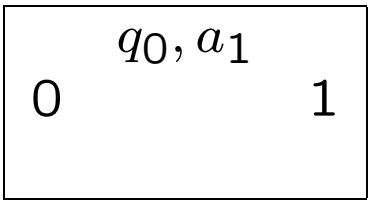
Rajoitutaan kaakelointiongelman versioon, jossa kysytään, voiko koko äärettömän puolitason kaakeloida annetuilla laattatyypeillä siten, että kiinnitetty laatta t esiintyy alimmassa rivissä.

Palautus on seuraavanlainen:

Kukin kaakeloinnin vaakarivi saadaan sopivalla laattavalikoimalla vastaamaan yhtä Turingin koneen M suorituksen vaihetta.

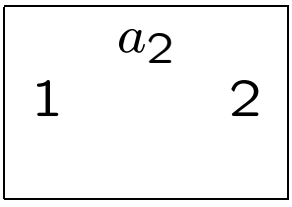
Laattojen värejä käytetään koodaamaan koneen nauha-aakkoston merkkejä sekä tilojen ja merkkien pareja. (Molempia on äärellisesti!) (Tässä värit esitetään suoraan niiden koodaamina symboleina tai symbolipareina.)

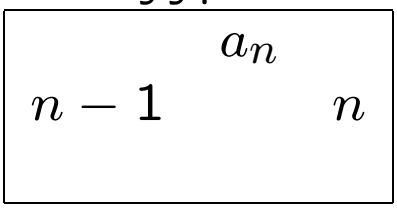
Alin kaakelirivi saadaan esittämään laskennan alkutilannetta syötteellä $a_1, \ldots, a_n$ seuraavasti:

Laatta t: 

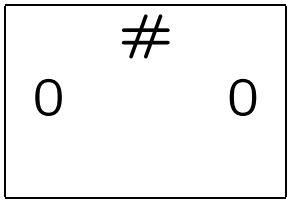
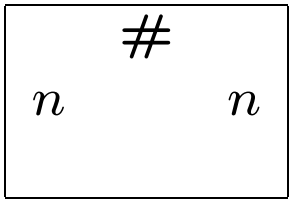
kuvaa alkutilan q_0 ja tiedon, että nauhapää on ensimmäisen syötemerkin a_1 kohdalla.

Loppuja syötteen merkkejä $a_2, \ldots, a_n$ esittävät laatat saadaan pakotettua oikeille paikoilleen muodostamalla laattatyypit

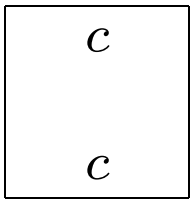


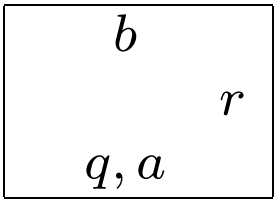
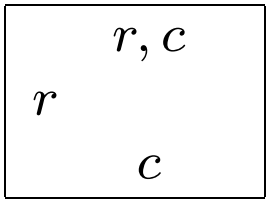
, ..., 

.

Syötettä edeltäviä ja seuraavia tyhjämerkkejä esitetään laatoilla  ja 

Ks. Harel kuva 9.10.

Seuraavia kaakelirivejä (eli suorituksen tilanteita) varten suurin osa nauhamerkeistä säilyy ennallaan. Tämä saadaan aikaan laattatyypillä  kutakin aakkoston merkkiä c kohden.

Koneen tilan ja nauhapään position esitys saadaan siirrettyä seuraavaan kaakeliriviin laatoilla  ja  jokaiselle nauhamerkille c ja siirtymälle $q \xrightarrow{a/b,R} r$.

Nauhapään oikealle siirtävä siirtymä esitetään oleellisesti yllä olevan peilikuvana.

Muodostetuilla laattatyypeillä voidaan kaakeloida koko puolitaso täsmälleen silloin, kun Turingin koneen M suoritus syötteellä $a_1, \ldots, a_n$ ei pääty.

Konstruktio siis palauttaa kaakelointiongelman pysähtymisongelmaan (Harel, kuva 9.9) – tarkemmin sanoen sen komplementtiin.

$\rightsquigarrow$ Kaakelointiongelma on ratkeamaton.

(Jos nimittäin sekä ongelma itse että sen komplementti ovat osittain ratkeavia, niin se onkin kokonaan ratkeava – HT.)

Äärelliset automaatit

Turingin koneilla on myöskin laskentavoimaltaan rajoittuneempia versioita, jotka ovat silti hyödyllisiä .

Jos rajoitetaan Turingin koneen nauhapää etenemään vain oikealle päin, sen kirjoittamilla merkeillä ei ole laskennan etenemisen kannalta merkitystä. (Miksi?)

Saatu yksinkertainen malli, **äärellinen automaatti** siis vain käy syötemerkit kertaalleen läpi ja suorittaa niiden ohjaamana siirtymiä tilasta toiseen.

Yksinkertaisuudestaan huolimatta äärellisillä automaateilla on tietojenkäsittelyssä huomattava merkitys esim. merkkijonojen tunnistusmenetelmänä.

Esim. Desimaalilukujen tunnistaminen

Äärellisiä automaatteja voi käyttää myös reaktiivisten järjestelmien esittämiseen ja analysointiin; syötejonona tällöin järjestelmään vaikuttavat tapahtumat.

Ks. Harel kuva 9.14.

Turingin koneitten, äärellisten automaattien sekä laskentavoimaltaan näiden väliin sijoittuvien mallien tarkastelu luo perustan **automaattien ja formaalikielten** tutkimukselle.

8. Rinnakkaisalgoritmit

(Harel, luvun 10 sivut 265-281.)

"Monet harhailevat, mutta tieto lisääntyy"
[Daniel 12:4]

"Kaksin on parempi kuin yksin, sillä kumpikin saa vaivoistaan hyvän palkan" [Saarnaaja 4:9]

- Kuinka prosessorien lisääminen vaikuttaa algoritmiseen laskentavoimaan?

Rinnakkaisuuden edut ja rajat

Rinnakkaisuudesta eli samanaikaisesta yhteistoiminnasta on selvästi etua.

Esim. talon rakennus 4 tunnissa 200 työntekijän voimin.

Algoritmisen toiminnan rinnakkaistaminen: Vrt

X:= 3; Y:= 4;

ja

X:= 3; Y:= X;

Ylempi rinnakkaistuu, jälkimmäinen ei.

Esim. kymmenen metriä pitkän ojan vs. kymmenen metriä syvän kuopan kaivaminen .

Esim. Onko (1. luennolla käsitelty) palkkalistan summaus **väistämättä peräkkäinen** (inherently sequential) tehtävä?

V: Ei; N luvun summaus voidaan tehdä $O(\log N)$ askeleessa käyttämällä $N/2$ prosessoria: Ensimmäisessä vaiheessa prosessorit summaavat yhtäaikaisesti lukupareja (ensimmäinen, toinen), (kolmas, neljäs), ..., ($N-1$:nen, N:s). Toisessa vaiheessa summataan $N/4$ paria edellisen vaiheen summia jne kunnes jäljellä kokonaissumma.

Harel kuva 10.1.

Vrt. N joukkueen turnaus, jossa ottelujen häviäjät karsiutuvat, voidaan järjestää $N/2$ kentällä $O(\log N)$ ottelupäivässä

Kiinteä ja laajeneva rinnakkaisuus

Keskitytään toistaiseksi ongelman rinnakkaisessa ratkaisemisessa tarvittavien prosessorien lukumäärään.*

Edellä saimme aikaiseksi kertaluokkanopeutuksen ($O(N) \rightsquigarrow O(\log N)$) käyttämällä syötteen koon mukaan lineaarisen määrän ($N/2$) prosessoreita.

ns. **laajeneva rinnakkaisuus** (expanding parallelism).

Jos käytettävissä kiinteä määrä prosessoreja, vain vakiokertoimella nopeuttaminen mahdollista.

*Tiedonsiirto myös tärkeä ongelma: miten syötteet jaetaan prosessoreille, ja miten prosessorit välittävät tuloksensa toisilleen?

Rinnakkaisalgoritmien tutkimuksessa tarkastellaan suoritusajan ja muistitilan lisäksi kolmatta suuretta ja sen tarvetta syötteen koon funktiona:

prosessorien määrää ("size complexity").

Esim. N kokonaislukua voidaan laskea yhteen ajassa $O(\sqrt{N})$ käyttämällä $\sqrt{N}$ prosessoria.

(HT?)

Rinnakkainen lomituslajittelu

Palautetaan mieliin rekursiivinen lomituslajittelualgoritmi:

MSort($a_1, \ldots, a_n$):
1. **if** n=1 **then return** a_1;
2. **else**
3. S1:= MSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$);
4. S2:= MSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$);
5. **return** Merge(S1, S2);

Hajota-ja-hallitse-tyyppinen algoritmi rinnakkaistuu helposti, jos aliongelmien ratkaisujärjestyksellä ei ole väliä.

Merkitään toimenpiteiden A ja B rinnakkaista käynnistämistä ($A||B$). Lomituslajittelu voidaan toteuttaa rinnakkaisena:

ParMSort($a_1, \ldots, a_n$):
1. **if** n=1 **then return** a_1;
2. **else**
3. (S1:= ParMSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$) ||
4. S2:= ParMSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$))
5. **return** Merge(S1, S2);

Suoritusta ParMSort($a_1, \ldots, a_n$) kuvaa binäärinen kutsupuu, jossa kutakin solmua suorittaa oma prosessori.

Puussa lehtiä n ja siten tasoja $O(\log n)$.

Tehtävänsä (syvemmillä tasoilla) suorittaneita prosessoreja voidaan käyttää uudestaan $\rightsquigarrow n$ prosessoria riittää.

Rinnakkaisuus $\rightsquigarrow$ yhden tason suoritusaika = yhden k.o. tasolla olevan kutsun suoritusaika.

Suoritusaikaa dominoi listojen S1 ja S2 lomittaminen.

Oletetaan, että n on kahden potenssi, ja lasketaan lomittamisessa tehtävien vertailujen lukumäärät. Yhden mittaisten listojen lomittamiseksi tarvitaan yksi vertailu, kahden mittaisten lomittamiseksi enintään 3, kahdeksan mittaisille enintään 7 jne:

$$\begin{aligned}
& 1 + 3 + 7 + 15 + \cdots + (n-1) \\
= \; & (2^1 - 1) + (2^2 - 1) + \cdots + (2^{\log n} - 1) \\
= \; & \sum_{i=1}^{\log n} (2^i - 1) = 2 \sum_{i=0}^{\log n - 1} 2^i - \log n \\
= \; & 2(2^{\log n} - 1) - \log n = 2n - \log n - 2 < 2n.
\end{aligned}$$

Siis lajittelun voi tehdä ajassa $O(n)$ käyttäen $O(n)$ prosessoria.

Aiemmin näimme, että alkioiden vertailuun perustuvan lajittelun optimaalinen peräkkäistoteutus vaatii $\Theta(n \log n)$ vertailua.

Tulovaativuus ja työoptimaalisuus

Merkitään rinnakkaisalgoritmin tarvitsemaa prosessorien määrää $p(n)$ ja aikaa $t(n)$.

Mikä on hyvä/tehokas rinnakkaisalgoritmi? Nopein? Vähän prosessoreja tarvitseva?

Rinnakkaisalgoritmin toimintaa voidaan simuloida yhdellä prosessorilla

- suorittamalla rinnakkaiset osuudet jossain peräkkäisjärjestyksessä, ja siten
- ajassa $O(p(n) \times t(n))$

Yleensä tavoitellaan mahdollisimman nopeaa rinnakkaisalgoritmia, joka on **työoptimaalinen**:

Sen **tulovaativuus** $p(n) \times t(n)$ on samaa kertaluokkaa kuin tehokkaimman peräkkäisalgoritmin aikavaativuus.

Esim. rinnakkaisella lomituslajittelulla $p(n) = O(n)$ ja $t(n) = O(n)$, joten se ei ole työoptimaalinen:
$p(n) \times t(n) = O(n^2) \neq O(n \log n)$.

Ks. Harel kuva 10.2.

Kiinteät laskentaverkot

Rinnakkaisalgoritmien suunnittelussa käytetyt prosessorimallit voivat olla vaikeita toteuttaa käytännössä.

Jos jokainen prosessori voi käsitellä jokaista muistipaikkaa, kuinka kytkennät toteutetaan? Kuinka muistipaikan samanaikaisen lukemisen tai kirjoittamisen konfliktit ratkaistaan?

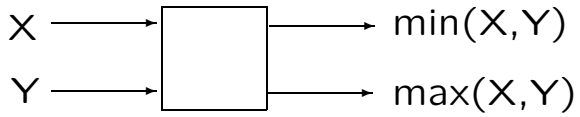
Käytännössä suoraviivaisesti toteutettavia ovat yksinkertaisista laskentaelementeistä koostuvat **laskentaverkot** (networks). Kukin laskentaelementti/prosessori kytketty kiinteästi muutamiin naapuriprosessoreihin.

Esim. Normaalin prosessorin perustoiminnot toteutettu yksinkertaisista AND-, OR- ja NOT-**porteista** (gate) koostuvina logiikkapiireinä.

Lajitteluverkot

Lajitteluverkot laskentaverkkoja, joilla voidaan järjestää n syötelukua kasvavaan järjestykseen.

Ne koostuvat yksinkertaisista vertailuprosessoreista (comparator):



Ns. **odd-even-lajitteluverkko** toteuttaa lomituslajittelun idean kiinteästi kytketyillä vertailuprosessoreilla.

Harel kuva 10.4.

Verkon alkuosa: kaksi rinnakkaista aliverkkoa $n/2$ alkion lajittelemiseksi.

Verkon loppuosa: aliverkko järjestettyjen $n/2$-alkioisten jonojen lomittamiseksi.

Lajitteluverkon suoritusaika $t(n)$ = verkon *syvyys*: monenko prosessorin läpi lukujen on pahimmillaan kuljettava.

Voidaan osoittaa: odd-even-lajitteluverkolla $t(n) = O((\log n)^2)$ ja $p(n) = O(n(\log n)^2)$.

$\rightsquigarrow$ Odd-even-lajittelu ei ole työoptimaalinen.

On olemassa myös työoptimaalinen lajitteluverkko, jolla $p(n) = O(n)$ ja $t(n) = O(\log n)$.

Se on kuitenkin monimutkainen ja käytännössä tehoton (vakiokertoimet suuria).

Systoliset laskentaverkot

Numeerisessa tieteellisessä laskennassa käsitellään tyypillisesti kookkaita lukuvektoreita ja -matriiseja.

laskutoimitukset usein tehokkaita ns. **systolisilla laskentaverkoilla**, joissa laskenta etenee jaksollisesti synkronoituina vaiheina (vrt. veren pumppaus elimistössä).

Esim. matriisin ja vektorin kertolasku.

Kurssilla N oppilasta, kukin osallistunut pisteytettyihin osasuorituksiin, joita M kpl $\rightarrow$ $N \times M$-matriisi.

Oppilaan arvosana määräytyy osasuoritusten painokertoimilla $(w_1, \ldots, w_M)$ painotettuna summana.

Peräkkäisratkaisu:

for opp:= 1 **to** N **do**
    Arv:= 0.0;
    for suor:= 1 **to** M **do**
        Arv:= Arv + Suor[opp][suor] * w[suor];
    output "Oppilaan", opp, "arvosana:", Arv;

Aikavaativuus selvästi $O(NM)$.

Ratkaistavissa ajassa $O(N + M)$ systolisella laskentaverkolla, jossa M liukuhihnaksi kytkettyä prosessoria.

Harel kuvat 10.5 ja 10.6.

Rinnakkaisuuden voima?

Käyttämällä tapauksen koon mukaan kasvava määrä prosessoreita ogelmille saadaan siis kertaluokaltaan nopeampia ratkaisualgoritmeja.

Auttaako rinnakkaisuus ratkaisemaan ongelmia, jotka ovat peräkkäisalgoritmeille (i) ratkeamattomia tai (ii) työläitä ongelmia?

Vastaus kysymykseen (i) on ”ei”:

Ongelman ratkaisua rinnakkaisalgoritmilla voi simuloida yhdellä prosessorilla, joka tekee jossain järjestyksessä kaikkien prosessorien työt.

Siis Church-Turingin teesi pätee myös rinnakkaisalgoritmeihin.

Vastaus kysymykseen (ii) on ”emme tiedä”:

Todistettavasti työläille ongelmille ei tunneta polynomisessa ajassa toimivia rinnakkaisalgoritmeja.

Toisaalta (työläiksi *epäillyt*) NP-täydelliset ongelmat ovat rinnakkaisalgoritmeilla polynomisessa ajassa ratkeavia:

Luokan NP päätösongelma A voidaan ratkaista epädeterministisellä algoritmilla Q_A jonkin polynomin $p(n)$ rajoittamassa määrässä askelia, tekemällä valintatilanteissa ”maagisia oikeita arvauksia”.

Rinnakkaistoteutus voi simuloida y.o. suoritusta deterministisesti allokoimalla valintatilanteissa uuden prosessorin jatkamaan kunkin vaihtoehtoisen valinnan mukaista laskentaa.

Jonkun prosessorin suoritus päättyy lopputilaan "kyllä" joss algoritmi Q_A hyväksyy syötteen. Toisaalta jos mikään prosessori ei päädy hyväksyvään lopputilaan $p(n)$ askeleen sisällä, tiedetään, että algoritmi Q_A hylkäisi syötteen.

Huom: ylläoleva NP-laskennan simulointi voi vaatia *eksponentiaalisen* määrän prosessoreita!

Tämä on kalvon 128 "jakaudutaan"-idea: NP on polynomiaikaista mutta muuten rajoittamatonta rinnakkaisuutta jossa eri prosessit kommunikoivat vain siten, että ensimmäinen "kyllä"-prosessi tappaa kaikki muut, ja vastaukseen "ei" päädytään vasta jos kaikki prosessit kuolevat itsekseen.

Rinnakkaislaskennan vaativuusluokat

Edellisen perusteella rinnakkaisalgoritmeilla polynomisessa ajassa ratkeavien ongelmien luokka **Parallel-P** (tai Parallel-PTIME) tuntuu epäkäytännöllisen laajalta: mahdollisesti eksponentiaalisesti kasvava prosessorien lukumäärä ei ole realistinen.

Käytännöllisesti mielekkääksi rinnakkaisvaativuuden luokaksi nimeltä **NC** (Nick's Class*) on esitetty "erittäin nopeasti" polynomisella määrällä prosessoreja ratkaistavissa olevien ongelmien luokkaa.

"Erittäin nopeasti": **polylogaritmisessa** ajassa $O((\log n)^k)$ jollain kiinteällä k.

Edes syötettä ei ole aikaa lukea annetussa järjestyksessä...

*Nicholas Pippengerin mukaan.

Monet käytännölliset ongelmat (esim. tarkasteltu palkkasummaus ja lajittelu) kuuluvat luokkaan NC.

Voidaan osoittaa, että luokan NC ongelmat ovat ratkaistavissa deterministisillä peräkkäisalgoritmeilla polynomisessa ajassa.

Siten tiedetään seuraavat ogelmaluokkien suhteet:

$$\mathrm{NC} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$$

Voidaan lisäksi osoittaa, että
Parallel-P = PSPACE.

Edellisten vaativuusluokkien sisältyvyyksien uskotaan (muttei tiedetä) olevan aitoja

Tämä tarkoittaisi (järjestyksessä vasemmalta oikealle), että

• joitain käytännössä ratkeavia ongelmia ($\in$ P) ei voida ratkaista erittäin nopeasti käytännöllisellä määrällä rinnakkaisprosessoreja
(esim. suurimman yhteisen tekijän laskenta?)

• joitain epädeterminismin avulla tehokkaasti ratkaistavissa olevia ongelmia ei voida ratkaista deterministisesti käytännöllisessa ajassa

• joitain rajoittamattomalla määrällä rinnakkaisuutta ratkaistavissa olevia ongelmia ei voida ratkaista peräkkäisalgoritmeilla käytännöllisessä ajassa edes epädeterministisesti.

NP-täydellisyys: ”Uskomme että epädeterministinen polynominen aika on aidosti determinististä vahvempaa. Silloin ainakin nämä ongelmat olisivat työläitä.”

P-täydellisyys: ”Uskomme että luokassa P on 'luonteeltaan sarjallisia' ongelmia jotka eivät rinnakkaistu merkittävästi. Näitä lienevät ainakin sen 'vaikeimmat' ongelmat.”

Luokan P sisällä *polynomiset* palautukset ovat liian karkea väline ongelmien luokitteluun (HT); niiden tilalla käytetään rajoittuneempia kuten *logaritmitilaisia* palautuksia.

Looginen esimerkki P-täydellisestä ongelmasta on **samastus (unification)** (C. Dwork, P.C. Kanellakis ja J.C. Mitchell: On the sequential nature of unification, *Journal of Logic Programming* 1 (1984), sivut 35–50):

Annettu kaksi lauseketta ϕ ja ψ jotka sisältävät (yhteisiä) muuttujasymboleita. Voidaanko näille muuttujasymboleille valita sellaiset arvot, että lausekkeista tulee samat?

Jos esimerkiksi $\phi = f(x, x)$ ja $\psi = f(g(y), x)$, niin sijoitus $x = g(y)$ tuottaisi yhteisen muodon $f(g(y), g(y))$.

Jos taas $\phi = f(h(x), x)$, niin yhteistä muotoa ei ole koska $h \neq g$.

Tuntuu luontevalta, että "patologisilla" ϕ ja ψ tarvittavat muuttujasijoitusten valinnat vaikuttavat toisiinsa niin, ettei niitä voi tehdä toisistaan riippumatta eli rinnakkain.

9. Satunnaistetut algoritmit

(Harel, luvun 11 sivut 309-310, 313-319 ja 322-331.)

"Heitetään arpaa, että saamme tietää" [Joona 1:7]

- Hyödymmekö jotain sallimalla algoritmien tehdä satunnaisia valintoja?

Alkulukujen tunnistus ja generointi

Alkuluvuilla on keskeinen asema paitsi perinteisessä lukuteoriassa myös moderneissa salausmenetelmissä. (Ks. myöh.)

Positiivinen kokonaisluku n on **alkuluku** (prime), jos se on jaollinen positiivisista luvuista ainoastaan ykkösellä ja itsellään.

Muuten n on **yhdistetty luku** (eli koosteinen luku; composite number).

Ykköstä ei yleensä pidetä alkulukuna, joten alkulukuja ovat

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots$$

Alkulukuja on kaikkien lukujen joukossa äärettömästi, epätasaisesti jakautuneina.

Lukua n pienempien alkulukujen lukumäärää merkitään $\pi(n)$; Tiedetään $\pi(n) \approx n/\ln n$.

Lukuteoreettisissa salausmenetelmissä tarvitaan suuria, esim. 150-numeroisia alkulukuja.

Kuinka näitä voidaan tuottaa?

Alkulukujen lukumäärästä $\pi(n)$ voidaan arvioida, että satunnaisesti valittu luku n on alkuluku todennäköisyydella $1/\ln n$.
$\rightsquigarrow$ 150-numeroisen alkuluvun löytämiseksi riittää testata keskimäärin.

$$\ln 10^{150} = 150 \ln 10 \approx 150 \times 2,3 = 345$$

satunnaista 150-numeroista lukua.*

Alkulukujen *generointi* siis palautuu alkulukujen *tunnistamiseen*.

Kuinka ratkaistaan, onko n alkuluku?

*Puolet vähemmän kokeilemalla vain parittomia.

Suoraviivainen menetelmä: Kokeillaan, onko n jaollinen millään luvuista $2, 3, \ldots, n-1$. Jos ei, n on alkuluku, muuten yhdistetty.

Ongelma: Eksponentiaalista luvun n *"tekstiesityksen" pituuden suhteen*; toimii, jos n pieni, mutta mahdotonta, jos $n \approx 10^{150}$.

Lievää parannusta: Testataan jakajina kakkosen lisäksi vain alkulukuja $\leq \sqrt{n}$

– näitäkin eksponentiaalisen paljon. (Esim. 150-numeroisella luvulla **paljon** enemmän kuin mikrosekunteja alkuräjähdyksestä!)

Alkulukujen tunnistamiseen ei tunneta todistetusti oikein toimivaa tehokasta (eli polynomisessa ajassa toimivaa) determinististä algoritmia.

Toisaalta ongelma on suhteellisen helposti ratkaistavissa *satunnaistetulla* algoritmilla.

Satunnaistettu alkulukujen testaus

Ensimmäisiä vaikeitten laskentaongelmien satunnaistettuja ratkaisumenetelmiä (Solovay & Strassen, 1977).

algoritmityyppi	pysähtyy nopeasti	vastaa oikein
Monte Carlo	aina	todennäköisesti
Las Vegas	todennäköisesti	aina

Menetelmän periaate; sivuutamme lukuteoreettiset yksityiskohdat.

Perustuu sen osoittamiseen, että testattava luku N *suurella todennäköisyydellä* ei ole yhdistetty (eli on alkuluku).

Koosteisuuden **todiste**: luku $K \in \{1, \ldots, N-1\}$, joka osoittaa, että N on yhdistetty luku.

Jos

$$\text{syt}(K, N)^* > 1, \tag{1}$$

N on selvästi yhdistetty luku. Syt voidaan laskea tehokkaasti Eukleideen algoritmilla.

Muuten tutkitaan, päteekö

$$\mathit{Js}(K, N) \not\equiv K^{(N-1)/2} (\text{mod } N), \tag{2}$$

missä $\mathit{Js}(K, N) \in \{-1, 0, 1\}$ on ns. Jacobin symboli. Sekä Jacobin symbolien että modulaaristen potenssien laskenta voidaan tehdä tehokkaasti, $O(\log N)$ aritmeettisella operaatiolla.

Voidaan osoittaa, että jos (2) pätee, N ei ole alkuluku. Toisaalta, jos N on yhdistetty luku, ainakin puolella mahdollisista $K \in \{1, \ldots, N-1\}$ epäyhtälö (2) pätee.

*Eli gcd(K, N); lukujen suurin yhteinen tekijä.

Sen todennäköisyys, että N on yhdistetty luku, jolle yksi K ei toteuta testejä (1) ja (2), on siis $< 1/2$. Kokeilemalla kaksi satunnaista K saadaan tn $< 1/4$, ... kun on kokeiltu k lukua, N on yhdistetty todennäköisyydellä $< 1/2^k$, eli N on alkuluku todennäköisyydellä $> 1 - (1/2)^k$.

Harel, kuva 11.1.

50–200 todistajaa läpäissyt luku N on erittäin suurella todennäköisyydellä alkuluku.

Alkulukuja ja yhdistettyjä lukuja voidaan siis tunnistaa tehokkaasti satunnaistetulla algoritmilla.

Toisaalta yhdistetyn luvun tekijöihin jakoa ei osata tehdä tehokkaasti edes satunnaisuutta hyväksikäyttäen, vaikka ongelmaa on tutkittu matematiikassa noin kolmesataa vuotta!

Satunnaistetut vaativuusluokat

Satunnaisalgoritmien formaali malli: **probabilistinen Turingin kone**, joka voi tehdä satunnaisia valintoja (”heittää kolikkoa”).

Vaativuusluokka **RP** (randomized polynomial time): Ne päätösongelmat, jotka voidaan ratkaista probabilistisellä Turingin koneella, joka
1) sanoo varmasti ”ei”, jos syöte on ongelman ”ei”-tapaus, ja
2) sanoo todennäköisyydellä $> 1/2$ ”kyllä”, jos syöte on ongelman ”kyllä”-tapaus.

$\rightsquigarrow$ Virheellisen vastauksen todennäköisyys saatavissa mielivaltaisen pieneksi iteroimalla suorituksia.

Esim. Sen testaaminen, onko syöteluku yhdistetty, on luokan RP ongelma.

Probabilistinen Turingin kone on deterministisen yleistys. Toisaalta epädeterminististen Turingin koneiden ”maagisesti” oikeaan johtava ”kolikko” on voimakkaampi kuin ”normaali” jollain todennäköisyydellä oikeaan johtava.

Siten tiedämme ongelmaluokkien suhteista:

$$\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}.$$

Taaskaan ei tiedetä, ovatko sisältyvyydet aitoja.

Huom. Koska epädeterminismi ei laajentanut algoritmisesti ratkeavien ongelmien joukkoa, sama pätee sitä heikompaan satunnaisuuteen, eli Church-Turing-teesi pätee myös satunnaisalgoritmeihin.

Julkisen avaimen salaus

Alkulukujen generoinnin helppoudella ja lukujen tekijöinnin (arvellulla) työläydellä on tärkeitä sovelluksia verkossa tapahtuvassa tiedonsiirrossa.

Tarvitaan menetelmiä, joilla verkossa kulkeva tietoliikenne voidaan **salata** asiattomilta.

Lisäksi tarvitaan keinoja **digitaalisten allekirjoitusten** toteuttamiseen, joilla voidaan varmistaa viestin lähettäjä ja viestin muuttumattomuus.

Menetelmillä ilmeisiä sovelluksia sähköisen rahan ja kaupankäynnin toteuttamisessa.

Perinteiset menetelmät ovat **symmetrisiä**: Viestin salaus ja salauksen purku tapahtuvat (oleellisesti) samalla **avaimella**.

Ongelmana salassa pidettävän avaimen turvattu välittäminen kumppaneiden kesken.

Julkisen avaimen salausmenetelmät ovat **epäsymmetrisiä**.

Kullakin on **kaksi** avainta: **julkinen salausavain** *Encr* ja **henkilökohtainen purkuavain** *Decr*.

Viestejä käsitellään jonoina kokonaislukuja M (katkomalla ja tulkitsemalla niiden bittejä sopivasti).

Henkilön A purkuavain (tai -funktio) $Decr_A$ avaa hänen salausavaimellaan (-funktiollaan) $Encr_A$ salatun viestin M:

$$Decr_A(Encr_A(M)) = M$$

Funktioiden oltava tehokkaasti laskettavia.

Lisäksi salainen purkufunktio $Decr_A$ ei saa olla helposti (eli polynomisessa ajassa) laskettavissa julkisesta salausfunktiosta $Encr_A$.

Perinteinen menetelmä on kuin yhteinen lipas, jonka avaimesta on kopio kummallakin kumppanilla – mutta toivottavasti ei kenelläkään muulla. . .

Lähettäjä laittaa viestinsä lippaaseen, lukitsee sen ja lähettää postipakettina vastaanottajalle.

Julkisen avaimen menetelmässä vastaanottaja antaakin lähettäjälle *avoimen* lippaan, mutta pitääkin itsellään sen *ainoan* avaimen!

Avoin lipas on vastaanottajan *Encr*, sen avain taas vastaava *Decr*.

Jos salaus- ja purkufunktiot ovat bijektiivisiä, alkuperäinen viesti voidaan ensin *salata purkufunktiolla* $Decr_A$ ja sitten *purkaa salausfunktiolla*:

$$Encr_A(Decr_A(M)) = M.$$

Koska vain A tuntee purkufunktionsa $Decr_A$, hänen julkisellä salausfunktiollaan $Encr_A$ aukeavat viestit ovat välttämättä A:n lähettämiä
$\rightarrow$ digitaalinen allekirjoitus.

Esim. Harel kuva 11.4.

RSA-menetelmä

Tunnetuin julkisen avaimen salausmenetelmä.

Rivest, Shamir & Adleman, 1978.

Alice (A) valitsee kaksi satunnaista (esim. 150-numeroista) alkulukua P ja Q, kuten edellä käsiteltiin.

Lasketaan $N = P \times Q$ ja $R = (P-1) \times (Q-1)$.

Seuraavaksi A valitsee suuren (esim. 300-numeroisen) luvun K, jolla syt$(K, R) = 1$. Esim. molempia P ja Q suurempi alkuluku kelpaa.

Lopulta A laskee luvun G, jolla

$$G \times K = 1 (\text{mod } R).$$

Voidaan osoittaa, että tällainen G on olemassa ja yksikäsitteinen.

Julkinen avain on pari (G, N),
salainen avain on pari (K, N).

P ja Q voidaan hävittää, mutta ne on pidettävä salassa. (Miksi?)

Kuinka lähetetään viesti A:lle, jonka julkinen avain (G_A, N_A) tunnetaan?

Käsitellään viestiä M numeroina väliltä $0 \ldots N_A - 1$.

Salatun viestin $E_A(M)$ tuottaminen:

$$E_A(M) = M^{G_A} (\text{mod } N_A).$$

A purkaa vastaanottamansa salatun viestin:

$$D_A(E_A(M)) = E_A(M)^{K_A} (\text{mod } N_A).$$

Voidaan osoittaa, että tämä on sama kuin alkuperäinen viesti M.

Helpompi nähdä, että

$$\begin{aligned} D_A(E_A(M)) &= E_A(M)^{K_A} (\text{mod } N_A) \\ &= (M^{G_A} (\text{mod } N_A))^{K_A} (\text{mod } N_A) \\ &= M^{G_A \times K_A} (\text{mod } N_A) \\ &= E_A(D_A(M)) \end{aligned}$$

$\rightarrow$ menetelmä sopii digitaaliseen allekirjoittamiseen.

Menetelmän soveltamisessa tarvitut laskutoimitukset ovat polynomisessa ajassa ja kohtalaisen tehokkaasti laskettavia. (Sivuutetaan.)

RSA-menetelmää pidetään turvallisena: kaikki sen murtamiseen kehitetyt mahdollisuudet ovat vähintään yhtä työläitä kuin lukujen tekijöihin jako, johon ei ole löydetty polynomista ratkaisumenetelmää.

10. Algoritminen älykkyys

(Pohjana Harel, luku 12. Lisäksi materiaalia kirjasta S.J. Russell & P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall 1995.)

"What a piece of work is a man! how noble in reason! how infinite in faculty! in form and moving, how express and admirable! in action, how like an angel! in apprehension, how like a god! the beauty of the world! the paragon of animals!

And yet, to me, what is this quintessence of dust?"

[William Shakespeare: Hamlet (2. näytös, 2. kohtaus)]

Mitä on **teko-** tai **keinoäly**?

Voiko mekaaninen laskenta tuottaa älykkyyttä?

Lisää kurssilla *Tekoäly*.

”Spesisimin” synti

”Kysymys ’osaako tietokone ajatella?’ on kuin kysymys ’osaako sukellusvene uida?’.”

Olisi epäreilua *määritellä* ”älykkyys” tyyliin ”se erityinen kyky jolla ihmiset ratkovat menestyksekkäästi vastaantulevia ongelmia”.

Entä (omissa puuhissaan varsin menestyksekkäät) eläimet? Tai Maapalloa valloittamaan laskeutuvat avaruusoliot?

Älykkyyden määritelmä ei siis saa perustua biologiaan, vaan yleiseen kykyyn *käyttää informaatiota ongelmien ratkaisussa* tms.

Mutta sitähän tietokoneetkin tekevät! Voisivatko nekin siis olla älykkäitä?

Turingin testi

”Jos se näyttää leijonalta, kuulostaa leijonalta ja tuntuu leijonalta, niin se *on* leijona” – tai minun on ainakin syytä toimia ikään kuin se olisi leijona. . .

Alan M. Turing (taas!) sovelsi tätä periaatetta ehdottaessaan vuonna 1950 nimeään kantavaa määritelmää sille, milloin tietokone olisi älykäs.

Taustalla on silloin psykologiassa vallinnut *behavioristinen* ajattelu: se on älykäs jos se *käyttäytyy* älykkäästi.

Tentaattori A olkoon ihminen. Ihminen lienee kyvykäs arvioimaan muiden älykkyyttä...

Valitaan tietokonekandidaatille ihmisopponentti, ja annetaan niille nimet B ja C tentaattorin A tietämättä kumpi on kumpi.

Liitetään B ja C tentaattoriin A pelkkää tekstiä välittävillä tietoliikennelinjoilla. Näin sokaistaan tentaattorin A biologiset erityistaidot lajitoveriensa tunnistamiseen kuuloaistilla, näköaistilla,...

Harel, kuva 12.1.

A tekee tenttijöille B ja C kirjallisia kysymyksiä, ja yrittää vastausten perusteella päätellä kumpi on kumpi.

Tietokone on tunnustettava älykkääksi jos A ei tähän pysty. (Tai jos toistettaessa testiä eri tentaattoreilla A oikean vastauksen todennäköisyys lähestyy lantinheittoa $\frac{1}{2}$.)

Kiinalainen huone

Filosofi John Searle on kääntänyt tilanteen toisin päin:

Istun ääni- ym. eristetyssä huoneessa jossa on kaksi luukkua, ”syöte” ja ’tulos”.

Huoneessa on myös kirja, jonka jokaisella sivulla on yksi kiinalainen kirjoitusmerkki (enkä minä osaa kiinaa).

Aina kun syöteluukkuun ilmestyy kirjoitusmerkki, etsin sitä kirjani aukeamien vasemman puoleisilta sivuilta. Kun löydän, laitan tulosluukkuun oikean puoleisella sivulla olevan merkin.

Huone kuulemma keskustelee ulkomaailman kanssa älykkäästi kiinaksi – mutta missä huoneen älykkyys on?

Huone on korkeintaan yksinkertainen Turingin kone! Missä sen älykkyys sitten on?

Neuroottista älykkyyttä

Searle siis kieltää Turingin perusidean että älykkyys voitaisiin määritellä ”älykkään näköisenä” *makro*tason syöte-tulos-vastaavuutena. Entä *mikro*tasolla?

Jos hermosoluverkon kytkentöjen voimakkuudet eivät muutu – eli aivot eivät muista – voidaan verkon kokonaistoiminta (= yksittäisten hermosolujen syöte-tulos-operaatioiden yhdistelmä) kuvata loogisena operaationa (McCulloch & Pitts 1943) tai jopa äärellisenä automaattina. . .

. . . ja Turingin kone taas voidaan nähdä äärellisenä automaattina jolla *on* muisti!

Eli muistimme malli ei saisi toisaalta olla ”Turing-tyyppinen” eikä toisaalta liian sidoksissa biologiamme yksityiskohtiin.

Pelit

Kysymys *täyden* älykkyyden *periaatteellisesta* ohjelmoitavuudesta on enemmänkin filosofien, kognitiotieteilijöiden, psykologien yms. alaa.

Tietojenkäsittelijä taas yrittää kehittää laskennallisia mentelmiä jotka antavat *käytännössä* ”älykkäitä” vastauksia *rajoitetuilla* sovellusalueilla.

Yksi tälläinen alue ovat **pelit**: maailmasta tarvittava *tieto mahtuu laudalle* ja mahdollisia *siirtoja on rajallisesti*, ja silti menestyksekästä pelaamista pidetään älykkyyttä vaativana.

Pelejä kuvaillaan *pelipuilla* joiden solmut ovat pelitilanteita ja solmusta lähtevät kaaret mahdollisia siirtoja.

Harel, kuva 12.2.

Lehdet ovat lopputilanteita joissa ratkeaa voitto ja häviö.

3×3-jätkänshakin pelipuun juuressa on tyhjä ruudukko, ja sillä on 9 lasta, eli eri mahdollisuudet laittaa ensimmäinen ristimerkki. (R&N, kuva 5.1.)

Kussakin näin syntyneessä ruudukossa on 8 mahdollisuutta laittaa ensimmäinen ympyrämerkki. Puuhun saadaan korkeintaan $9 \cdot 8 \cdot 7 \cdot \cdots \cdot 1 = 362\,880$ solmua. Käsiteltävissä helposti tietokoneella.

Tavallisen shakin haarautumisaste (=lasten lukumäärä) on noin 35 ja peli voi edetä 40-50 siirtoparia. Saadaan suunnilleen kokoa 35^{100} oleva puu – suurempi kuin tunnetun maailmankaikkeuden protonien määrä tai mikrosekuntien määrä sitten alkuräjähdyksen!

Vaikka *periaatteessa* käsiteltävissä, käytännössä puusta karsitaan (= lapsia ei tuoteta eikä tutkita) mielenkiinnottomia solmuja.

Heuristiikat

(Kreikan ”heurisko”, ”keksiä”; vrt.
”heureka”, ”olen keksinyt”.)

Jos meillä olisi *varmaa* tietoa siitä mikä lapsisolmuista on koko loppupelin kannalta paras, niin emme tarvitsisi koko puuta: laskisimme nykyisen solmun lapset, valitsisimme niistä parhaan, unohtaisimme muut, ja jatkaisimme peliä sillä siirrolla. Saisimme klassisen algoritmin!

Vertaa operaatio **choose**, joka tietää. Tämä on kalvon 128 etsintäajatus.

Vaan emme tiedä, vaan joudumme tyytymään *arviointifunktioon* joka kertoo annetusta solmusta
• pitäisikö sen lapset tuottaa ja tutkia
• ja jos ei niin kuinka hyvä se kannaltamme on.

Shakissa voisi heuristinen arviointifunktio olla vaikkapa:

”Jos nykytilanteessa kukaan ei uhkaa ketään, niin tuota sen lapset tutkittaviksi – tilanne on epäselvä ja vaatii lisätietoa.

Muuten laske kummalle jäisi materiaalietu, jos kaikki uhkaukset pelattaisiin nyt heti tyyliin ’minä syön tämän – sinä syöt tuon – minä sen – . . . ’.”

Esimerkki *ahneesta* heuristiikasta: ajatellaan että kannattaa syödä heti kun vain pystyy eikä mietiä millainen tulevaisuus sen jälkeen on.

Altistaa tietenkin *uhrauksille* jossa tarjoamalla syötävää houkutellaan pelaaja lyhytnäköisen hyödyn toivossa lopulta tappioon johtavalle tielle.

Horisonttiongelma: kuinka kauaksi tulevaisuuteen pitää kurkistaa?
(R&N, kuva 5.5.)

Hyvässä arviointifunktiossa onkin aina sekä **ahne** että **konservatiivinen** komponentti sopivassa suhteessa.

Jos ahneeseen luotetaan liikaa, niin hävitään hurja/kaistapäisyyden vuoksi, koska puusta tutkitaan liian *vähän* vaihtoehtoja.

Jos konservatiiviseen, niin hävitään aikapulan vuoksi, koska tutkitaan liian *paljon*.

Arviointifunktio siis leikkaa pelipuun poikki (paljon) ennen kuin päästään lehtiin, ja liittää kuhunkin katkaisukohtaan luvun, joka kertoo (toivottavasti) kuinka *lupaavaa* olisi kulkea ensin siihen, kun lopullisena tavoitteena on päätyä voittolehteen.

Samat periaatteet sopivat myös *yksinpeleihin* kuten pasianssiin ja **ongelmanratkaisuun**.

Ohjattu haku ongelmanratkaisussa

Haetaan esimerkiksi lyhintä reittiä Aradista Bukarestiin (R&N, kuva 4.2.) – joskin tähän tehtäväänhän on "oikeakin" algoritmi. . .

Hakualgoritmi A^* (R&N, luku 4.2.) antaa kullekin vielä tutkimattomalle solmulle lupaavuusluvuksi "*vaadittu työ siirryttäessä alusta solmuun + heuristinen arvio jäljellä olevasta työstä jatkettaessa solmusta loppuun*", ja tutkii aina lupaavimman solmun.

Vaadittu työ = löydetty reitti teitä pitkin Aradista tähän kaupunkiin.
Arvio = etäisyys linnuntietä tästä kaupungista Bukarestiin.
(R&N, kuva 4.4.)

Kun arvio ei ole pessimistinen, löydetään paras reitti.

Jos arvio on liian optimistinen (esim. 0), etsintään kuluu liikaa aikaa.

Minimax-periaate

Miten katkaisuluvut nostetaan sisäsolmuihin?

Omalla vuorollani teen minulle lupaavimman siirron. Valitsen siis *maksimin* nykyisen (juuri)solmuni lasten luvuista.

Koska vastustajakin haluaa voittaa, niin hän valitsee omalla vuorollaan hänelle itselleen lupaavimman eli minulle vähiten lupaavan siirron. Hän siis valitsee *minimin*.

Harel, kuva 12.3.

Vertaa kalvojen 145-146 polynomiseen hierarkiaan: ”Minullapa onkin sellainen (siirto ja) lupaavuusluku, ettei mikään sinun seuraava (siirtosi ja) lukusi pysty sitä pienentämään!”

Minimax-periaate sallii puun pienentämisen edelleen (esim.) ns. α-β-karsinnalla.

Harel, kuva 12.4.