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

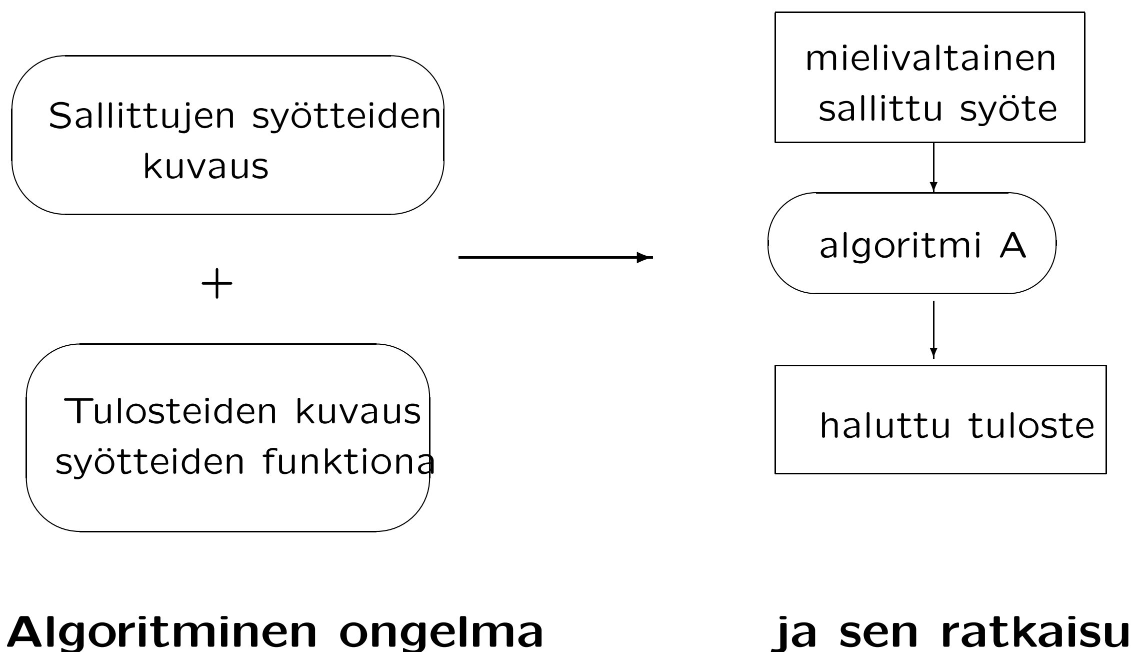
Algoritmisen ongelman muodostavat

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?



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 $\rightsquigarrow$ 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*.

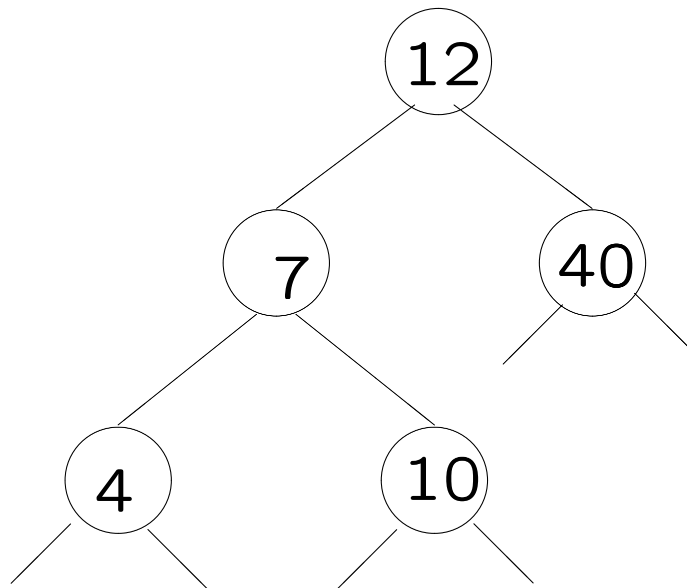
Binääripuu on rakenne, joka on joko
a) *tyhjä* tai
b) koostuu *solmusta*, jolla on *vasen* ja *oikea* alipuu; solmun alipuut ovat binääripuita, jotka eivät sisällä k.o. solmua.

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 $b_1 \le b_2 \le \cdots \le b_n$. $(\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\})$

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

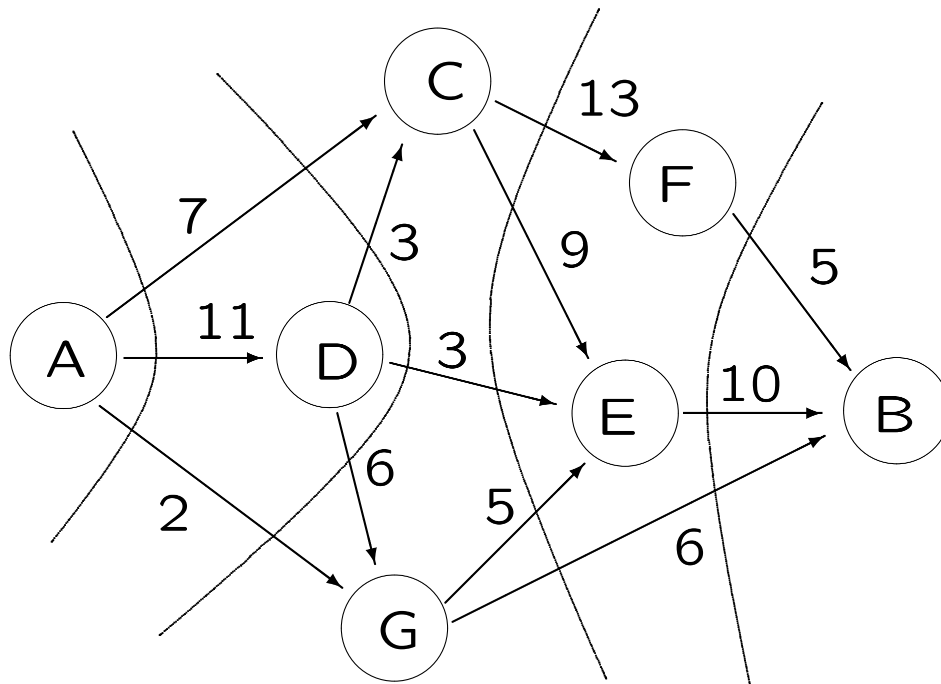
Joskus ahneet (lokaalisti optimaaliset) valinnat eivät johda oikeaan tulokseen.

(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 $b_1b_2b_3\ldots b_{n-1}$ – rekursio!

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

   Vain ne operaatiot joilla jono $a_1a_2a_3\ldots a_{m-1}$ muuttuu jonoksi $b_1b_2b_3\ldots b_{n-1}$.

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

   1 poisto + jonon $a_1a_2a_3\ldots a_{m-1}$ muutos jonoksi β.

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

$W_{i,j}$ 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 $b_1 b_2 b_3 \dots b_j$ j lisäyksellä.
- $W_{i,0} = i$ koska jono $a_1 a_2 a_3 \dots a_i$ tyhjenee i poistolla.

Ensimmäinen yritys: suora rekursio

```
function W(i, j: ℕ): 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(..., ...)
        korvattu taulukkoviitteillä V[..., ...];
        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:

		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						

(Column header above b…a: β)

Lisäongelmia (HT):

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