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, $0 < MAX < 60$. 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;
```

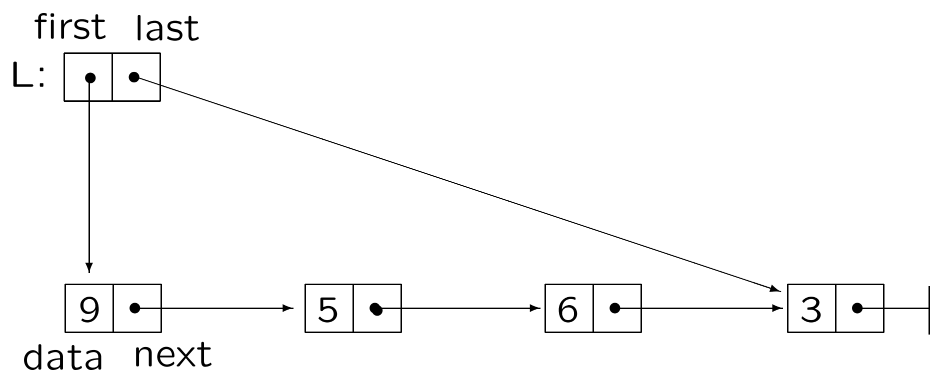
Modifioitu silmukka noin 50% tehokkaampi.

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 $\neq$ null **and then** p.data $\neq$ 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
```

$\leadsto$ 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

$\leadsto$ 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

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

$\leadsto$ 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.