1 Johdanto

Kun informaatiota säilötään tietokoneen

- keskusmuistiin
- oheislaitteille (kuten kovalevylle)

se pitää organisoida siten, että

- voidaan vastata kysymyksiin, jotka koskevat säilöttyä informaatiota
- säilöttyä informaatiota voidaan päivittää
- vastaamiseen ja päivittämiseen kuluvat resurssit (aika ja tila) ovat kohtuulliset
- varsinkin silloin, kun informaatiota on paljon.

Kurssi käsittelee tällaisia

- *fyysisiä* tallennusrakenteita
- rakenteiden kysely- ja
- päivitysmenetelmiä

kahdesta eri näkökulmasta:

- Kuinka tehokkaasti operaatiot (tallennus, kyselyt ja päivitykset) voidaan hoitaa?

  Erityisesti tallennettavan informaation määrän kasvaessa suureksi.

- Millaisilla suunnittelu- ja ohjelmointiperiaatteilla näitä tehokkaita operaatioita voidaan löytää ja toteuttaa?

1.1 Sisältö ja tavoitteet

Kurssi sisältää perus*tietorakenteita* (data structures) kuten

- listoja (list)
- jonoja (queue)
- pinoja (stack)
- puita (tree)
- verkkoja (graph)

sekä tavallisimpia *algoritmeja*, jotka käyttävät niitä, kuten

- lajittelualgoritmeja (sorting)
- verkkoalgoritmeja (graph algorithms).

Kurssin tavoitteena on pystyä

- tunnistamaan millaiset tietorakenteet sopivat annettuun algoritmiseen ongelmaan.

  Tämän vuoksi kurssin

  - luennot
  - laskuharjoitukset
  - kokeet

  sisältävät ongelmanratkaisutehtäviä.

- toteuttamaan sopivat tietorakenteet.

  Kurssi *ei* kerro yksityiskohtaisesti, miten se tehdään jollakin tietyllä ohjelmointikielellä (edes Javalla), vaan yleisiä periaatteita, jotka soveltuvat ohjelmointiin yleisesti.

1.2 Lähdemateriaali

- Timo Karvin luentomuistiinpanot syksyltä 2002.

  Osta monistemyynnistä tätä kurssia varten!

  Tämä luentokurssi käsittelee samat asiat mutta eri järjestyksessä ja eri näkökulmasta.

- T.H.Cormen, C.E.Leiserson, R.L.Rivest ja C.Stein: *Introduction to Algorithms. Second Edition.* The MIT Press, 2001.

  Yllämainittujen luentomuistiinpanojen päälähde. Laitoskirjaston kurssikirjahyllyssä.

  Voit ostaa kirjakaupasta, jos tarvitset (jo nyt) *laajaa ja perusteellista hakuteosta* algoritmeista ja tietorakenteista.

  Myös 1. painos (Cormen, Leiserson ja Rivest, 1990) käy.

- A.Levitin: *Introduction to the Design & Analysis of Algorithms.* Addison Wesley, 2003.

  Voit ostaa kirjakaupasta, jos kurssi saa kiinnostumaan siitä, *millaisia lähestymistapoja voi yrittää* kohdatessaan uuden algoritmisen ongelman.

  Ei kuitenkaan riitä (ainoaksi) hakuteokseksi algoritmeihin ja tietorakenteisiin.

- Kirjakaupoissa on myynnissä paljon kirjoja aiheesta "tietorakenteet Java-kielellä."

  Tälle kurssille ne *eivät* ole tarpeen.

  Ne saattavat olla hyödyllisiä *Tietorakenteiden harjoitustyössä:*

  – Oma erillinen ja pakollinen opintosuoritus.

  – Harjoittaa kurssin asioiden soveltamista käytäntöön kurssin laskuharjoituksia suuremmassa ohjelmointiongelmassa.

1.3 Lähialueita

Tietokantasuunnittelu: Hahmotetaan käyttäjän tietotarpeita *loogisella* tasolla menemättä fyysiseen toteutukseen.

Valitaan minkä sisältöisiä hakemistoja (indeksejä) kannattaa ylläpitää, jotta tietojärjestelmä toimisi tehokkaasti.

Tieto*kanta*suunnittelussa mallinnetaan *mitä* informaatiota kannattaa tallettaa.

Tieto*rakenne*suunnittelussa mallinnetaan *miten* se kannattaa tallettaa.

Matematiikka: Tietorakenteiden ja niiden päivitysalgoritmien

- oikean toiminnan todistaminen
- resurssitarpeiden arviointi.

Ohjelmointikielten teoria: Tyyppijärjestelmät joilla voisi ilmaista *tietorakenteiden yleis(käyttöis)iä määritelmiä*.

"Jos tyyppiä τ olevilla alkioilla on järjestysoperaatio, niin tyyppiä `array of` τ olevilla taulukoilla on lajitteluoperaatio."

Periytymisellä: "Aliluokka τ määrittelee yliluokan abstraktin metodin `int Comparable::compareTo(`x`:Object)`. Lajitteluun on `sort(Comparable[]` y`)`."

VÄÄRIN — merkitys muuttui:

Luvuilla on suuruusjärjestys.

Merkkijonoilla aakkosjärjestys.

Lukua saisi verrata merkkijonon kanssa!

Geneerisyydellä: "Jos annat proseduurin `compareTo`: $\tau \times \tau \to$ `int`, niin teen proseduurin `sort`: `array of` $\tau \to$ `array of` τ."

OIKEIN mutta yhä puutteellinen — proseduuri(e)n käyttäytyminen?

Ohjelmistosuunnittelu: Valmiiden tietorakenne*kirjastojen* tyyppihierarkioiden ja kutsurajapintojen suunnittelu.

"Miksi opiskella tietorakenteita, vaikka nykyaikaisissa ohjelmointikielissä on valmiit kirjastot?"

- Kirjaston ymmärtää paremmin, jos tuntee jonkin mahdollisen toteutustavan.
- Kirjastosta ei välttämättä löydy juuri sellaista tietorakennetta, jolla olisi kaikki juuri nyt tarvittavat ominaisuudet.
- Jonkunhan ne kirjastotkin on ohjelmoitava...
- Kuuluu ohjelmoinnin periaatteiden tuntemukseen.

2 Algoritmien oikeellisuudesta

- Aloitetaan esimerkillä, jossa ei vielä käytetä muita tietorakenteita kuin *taulukkoa* (array).
- Tutustutaan matemaattisiin työkaluihin algoritmien
  - kehittämiseen (etukäteen) tai
  - oikean toiminnan varmistamiseen (jälkikäteen).
- Esimerkkitehtävässä annetaan syötteenä
  - taulukko $A[0 \ldots N]$ lukuja
  - joka on järjestetty eli $A[i] < A[i+1]$ kaikilla indekseillä $0 \leq i < N$
  - luku b

  ja kysytään, esiintyykö b taulukossa A.

2.1 Hajoita ja hallitse

Hajoita ja hallitse (englanniksi divide and conquer, latinaksi divide et impera) on usein soveltuva menetelmä *algoritmin kehittämiseksi* annettuun laskentaongelmaan.

Vaatimukset:

Helpot perustapaukset voidaan tunnistaa ja ratkaista suoraan.

Vaikea tapaus voidaan aina *palauttaa* aidosti helpompiin osatapauksiin.

Osaratkaisut näille osatapauksille voidaan *yhdistellä* alkuperäisen vaikean tapauksen ratkaisuksi.

Silloin voidaan soveltaa yksinkertaista algoritmi*skeemaa:*

function hjh(x: **ongelmatapaus**): **sen ratkaisu**
  if x on helppo perustapaus **then**
    return tapausta x vastaava suora ratkaisu
  else
    Palauta x osatapauksiin $y_1, y_2, y_3, \ldots, y_m$;
    Laske *rekursiivisesti* näiden osatapausten y_i osaratkaisut $z_i = \text{hjh}(y_i)$ missä $1 \leq i \leq m$;
    Yhdistele nämä osaratkaisut $z_1, z_2, z_3, \ldots, z_m$ tapauksen x ratkaisuksi u;
    return u
  end if.

Sovelletaan hajoita ja hallitse -periaatetta kalvojen 2 esimerkkiongelmaamme:

- Intuitio: pidetään yllä taulukon osaa A[vasen ... oikea] jonka ulkopuolella b ei ainakaan voi olla.
- Koko tehtävä ratkeaa alkuarvoilla vasen $= 0$ ja oikea $= N$.
- Helppo perustapaus: Jos vasen $>$ oikea, niin osa on tyhjä ja ratkaisu on **false**.
- Palautusperiaate: Valitaan tutkittava indeksi vasen $\leq$ keski $\leq$ oikea.
  - Jos A[keski] $= b$, niin ratkaisu on **true**.
  - Jos A[keski] $> b$, niin riittää tutkia alkuosa A[vasen ... keski -1].
  - Jos A[keski] $< b$, niin riittää tutkia loppuosa A[keski $+1$ ... oikea].

  Muita mahdollisuuksia ei ole.

- Palautusperiaate vie aidosti helpompiin tapauksiin, koska jäljelle jäävä (alku- tai loppu)osa on aidosti lyhyempi kuin alkuperäinen osa.

  Osan $A[p \ldots q]$ pituus on $\ell(p, q) = (q - p + 1)$.

Tarkennettu skeema hjh:

function hae(vasen: $\mathbb{N}$, oikea: $\mathbb{N}$): **boolean**

```
1:  if vasen > oikea then
2:      return false
3:  else
4:      Valitse jokin vasen ≤ keski ≤ oikea;
5:      if A[keski] = b then
6:          return true
7:      else if A[keski] > b then
8:          return hae(vasen, keski −1)
9:      else
10:         return hae(keski +1, oikea)
11:     end if
12: end if.
```

Riviä 4 tarkennetaan myöhemmin.

2.2 Rekursio ja induktio

- Kalvojen 2.1 algoritmiskeema hae on *rekursiivinen* eli kutsuu itseään.
- Rekursiivisten algoritmien ja tietorakenteiden ominaisuuksia osoitetaan usein matemaattisella *induktiolla*.

  Perustapaus tulee niistä ohjelman haaroista, joissa *ei tehdä* rekursiokutsuja.

  Induktioaskel tulee niistä ohjelman haaroista, joissa *tehdään* rekursiokutsuja.

  Kutsuparametrien pitää olla sellaisia, että saadaan vedota induktioperiaatteeseen.

  Siis *pienentää aidosti muttei rajatta* sitä suuretta, jonka suhteen induktio etenee.

  Kutsun palauttaman tuloksen ominaisuudet saadaan induktio-oletuksesta.

Lause 2.2.1. *Rekursiokutsu* hae(p, q)

1. *pysähtyy (ilman ajonaikaista virhettä)*
2. *kertoo löytyykö b taulukon osasta $A[p \ldots q]$*

kaikilla indekseillä $0 \leq p \leq N + 1$ *ja* $-1 \leq q \leq N$ *joilla pituus* $\ell(p, q) \geq 0$.

Todistus. Induktiolla yli osan pituuden $\ell(p, q)$.

Olkoon ensiksi $\ell(p, q) = 0$. Silloin $p > q$, ja kutsun suoritus päätyy riville 2. Väite 1 pätee, koska rivi 2 palauttaa vakion. Väite 2 pätee, koska tuo palautettava vakio on **false**.

Olkoon sitten $\ell(p, q) > 0$. Silloin $0 \leq p \leq q \leq N$, ja suoritus päätyy riville 4.

Silloin indeksointi pysyy taulukon sisällä, eli ei aiheuta ajonaikaista virhettä. (Laitteistoviat jätetään huomiotta.)

Jos suoritus etenee riville 6, niin väitteet 1 ja 2 pätevät kuten yllä (ilman rekursiota tai induktiota).

Jos suoritus eteneekin riville 8, niin tehdään uusi rekursiokutsu uusilla arvoilla $p' = p$ ja $q' =$ keski -1. Koska

- $\ell(p', q') < \ell(p, q)$
- p' ja q' toteuttavat indekseille asetetut ehdot

voimme vedota induktio-oletukseen: uusi rekursiokutsu

- pysähtyy (väite 1)
- osa $A[p' \ldots q']$ on tutkittu (väite 2).

  Etsitty alkio b ei voi olla tutkimattomassa osassa $A[q' + 2 \ldots q]$, koska jo $A[q' + 1] > b$ ja taulukko A on järjestyksessä.

Silloin myös tarkasteltava rekursiokutsu

- pysähtyy (väite 1)
- tutkittuaan osan $A[p \ldots q]$ (väite 2).

Jos suoritus etenee aina riville 10 saakka, niin väitteet 1 ja 2 voidaan osoittaa symmetrisesti kuten edellä. □

Koko algoritmin oikea toiminta on erikoistapaus $p = 0$ ja $q = N$.

2.3 Toisto ja sen todistaminen

- Kalvojen 2.1 algoritmiskeemassa hae rekursio voidaan korvata *toisto*lla (iteraatiolla):

  Kutsun sijaan palataan silmukan alkuun päivitetyin parametrein.

  function hai(A: **array of** $\mathbb{N}, b$: $\mathbb{N}$): **boolean**

  ```
  1:  vasen := 0;
  2:  oikea := N;
  3:  while vasen ≤ oikea do
  4:      Valitse jokin vasen ≤ keski ≤ oikea;
  5:      if A[keski] = b then
  6:          return true
  7:      else if A[keski] > b then
  8:          oikea := keski −1
  9:      else
  10:         vasen := keski +1
  11:     end if
  12: end while;
  13: return false.
  ```

- Yleiseen muunnokseen palataan tarkemmin pinotietorakenteiden yhteydessä.

- Silmukoille käytetään (kalvojen 2.2 induktion sijaan) toiston tahdissa etenevää todistustapaa:

  Invariantti on ohjelman tilaa koskeva väite, joka

  – on totta silmukkaan tultaessa

  – pysyy totena silmukan kierroksesta toiseen, vaikka muuttujien arvot vaihtuvatkin.

  Vastaa induktiolla todistettavaa väitettä.

  Siis kertoo, mitä seuraava silmukkakierros tekee.

  Konvergentti on suure

  – joka pienenee aidosti silmukan joka kierroksella

  – jonka arvolla 0 silmukasta poistutaan.

  Vastaa suuretta, jonka suhteen induktiota tehdään.

  Siis takaa silmukan pysähtymisen.

- Algoritmiskeemassa hai

  invariantiksi voidaan taas valita

  "b ei voi olla osan A[vasen ... oikea] ulkopuolella":

  – Totta aluksi koska vasen $= 0$ ja oikea $= N$ eli koko taulukko.

  – Jos totta silmukan rungon alussa vanhoilla arvoilla vasen ja oikea, niin totta myös rungon lopussa uusilla arvoilla vasen′ ja oikea′:

  Vertailun A[keski] vs. b tulos ja taulukon A järjestys.

  konvergentiksi voidaan taas valita ℓ(vasen, oikea):

  – ℓ(vasen′, oikea′) $< \ell$(vasen, oikea).

  – **while**-ehto epätodeksi kun ℓ(vasen, oikea) ≤ 0.

- Päätellään: silmukan invariantti on totta myös niillä muuttujien arvoilla joilla konvergentti poistuu silmukasta.

- Algoritmiskeemassa hai päätellään:

  "Jos silmukan alalaidasta **end while** poistutaan, niin b ei voi olla taulukossa A."

- Invariantin ideaa käytetään myös tietorakenteissa:

  Ominaisuus jota pidetään yllä aina kun tietorakennetta päivitetään.

- *Opiskeluvihje:* Paina mieleesi tietorakenteiden invariantit.

  Päivitysmenetelmien periaatteet on (yleensä) helppo johtaa niistä!

2.4 Kaksi hakumenetelmää

- Johdetaan kalvojen 2.3 algoritmiskeemasta hai algoritmeja kiinnittämällä riville 4 valintasääntö.

- Eri säännöt hyödyntävät taulukon A järjestystä eri tavoin.

  Peräkkäishaku (sequential search) saadaan valitsemalla aina peräkkäisjärjestyksessä *seuraava* paikka taulukossa. (Kalvo 2.4.1.)

  Haku voidaan lopettaa heti kun A[keski] $\geq b$.

  Binäärihaku (binary search) saadaan valitsemalla aina (jompi kumpi) *keskimmäinen* paikka jäljellä olevassa taulukossa. (Kalvo 2.4.2.)

  Haku voidaan lopettaa heti kun jäljellä on korkeintaan 1 paikka.

2.4.1 Peräkkäishaku

function phai(A: **array of** $\mathbb{N}, b$: $\mathbb{N}$): **boolean**

1: keski := 0;
2: **while** (keski $< N$) **and** (A[keski] $< b$) **do**
3: keski := keski $+1$
4: **end while;**
5: **return** (A[keski] $= b$).

2.4.2 Binäärihaku

function hab(A: **array of** $\mathbb{N}, b$: $\mathbb{N}$): **boolean**

```
 1: vasen := 0;
 2: oikea := N;
 3: while vasen ≤ oikea do
 4:    keski := ⌊(vasen + oikea)/2⌋;
 5:    if A[keski] = b then
 6:       return true
 7:    else if A[keski] > b then
 8:       oikea := keski − 1
 9:    else
10:       vasen := keski + 1
11:    end if
12: end while;
13: return false.
```


3 Algoritmien analyysistä

- Algoritmin vaatimina resursseina voi tarkastella

  suoritusaikaa

  muistinkulutusta suorituksen aikana.

  "muisti $\leq$ aika": muistin kuluttamiseen kuluu välttämättä myös aikaa, mutta ei päinvastoin.

  Tällä(kin) kurssilla mitataan ensisijaisesti aikaa.

- Voi tarkastella resurssintarvetta

  pahimmillaan eli algoritmille *hankalimmilla* syötteillä.

  Silloin saadaan tietää mihin täytyy varautua.

  Tällä kurssilla analysoidaan (yleensä) pahimman tapauksen käyttäytymistä.

  "Helppoa" koska saamme valita syötteet itse...

  tyypillisesti eli *tyypillisillä* syötteillä.

  Silloin täytyisi tietää *syötteiden todennäköisyysjakauma*, jonka pohjalta voi sitten laskea resurssitarpeen *odotusarvo* (yms.).

  Resurssintarve **parhaimmillaan** ei (yleensä) kiinnosta:

  Esimerkiksi kalvojen 2.4 algoritmeilla se tarkoittaa, että b sattuu löytymään onnekkaasti heti ensimmäisestä tutkitusta taulukkopaikasta — so what?

- Periaatteina on tarkastella resurssia

  1. syötteen pituuden kasvaessa yhä suuremmaksi.

     Kuten syötetaulukon A yläraja N kalvoilla 2.4.

  2. laitteistosta tai ohjelmointikielestä riippumatta.

     "Lähdekoodin tasolla."

     Formaalisti pitäisi määritellä etukäteen jokin *abstrakti laskentamalli:*

     Hajasaantikone (Random Access Machine, RAM) eli "tavallisen" yksiprosessorisen tietokoneen keskeiset piirteet.

     Osoitinkone (pointer machine) jossa ei ole indeksoitavia taulukoita, vaan kaikki muistihaku on tehtävä seuraamalla osoitinviiteketjuja.

     Turingin kone (Turing machine) jonka avulla voi määritellä laskentaongelmille *vaativuusluokkia*.

     ⋮

- Periaate 1 $\Rightarrow$ kannattaa seurata *montako kertaa algoritmi päättää suorittaa uudelleen jonkin koodinpätkän.*

  Tällaisia "väijyntäpaikkoja" ovat

  rekursiivisen aliohjelman rivi 1 koska sillä lasketaan montako sisäkkäistä kutsua syntyy

  toistorakenteen alkurivi koska sehän on nimettömän rekursiivisen aliohjelman rivi 1 kalvojen 2.3 mukaan.

- Periaate 2 $\Rightarrow$ ei tarvitse seurata kauanko kuluu kahden peräkkäisen väijyntäpaikan välissä.

- Lasketaan siis montako kertaa ohjelman suoritus ohittaa jonkin väijyntäpaikan.

- Ohjelman vaatima kokonaisaika on kaikkien eri ohitusten yhteinen lukumäärä.

- Profiloija-nimiset ohjelmointityökalut (profiler) keräävät juuri tällaisia tilastoja eri väijyntäpaikkojen ohituksista.

3.1 Peräkkäishaun analyysi

Analysoidaan kalvojen 2.4.1 peräkkäishakualgoritmin pahinta suoritusaikaa.

- Syötteenä saadaan $N + 2$ lukua.

  Annetaan siis suureen N kasvaa.

- Pahin tapaus on se, jossa suoritus etenee loppuriville 5 saakka, eli b ei löydy taulukon alkuosasta $A[0 \ldots N-1]$.

- Ainoa väijyntäkohta on rivi 2.

Lause 3.1.1. *Rivi 2 suoritetaan enintään $N + 1$ kertaa.*

Todistus. Lause 3.1.2 rivin 1 antamalla alkuarvolla keski $= 0$. □

Lause 3.1.2. *Rivi 2 suoritetaan pahimmillaan vielä $\ell(\text{keski}, N)$ kertaa.*

Todistus. Aletaan etsiä apufunktiota $T(m)$ = "montako kertaa rivi 2 vielä suoritetaan, kun $\ell(\text{keski}, N) = m$".

(Toisin sanoen, suoritetaan muuttujan vaihto.)

Voidaan kirjoittaa *palautuskaava* (recurrence)

$$T(m) = \begin{cases} 1 & \text{kun } m = 1 \\ 1 + T(m-1) & \text{kun } m > 1 \end{cases} \qquad (1)$$

joka antaa apufunktiolta T vaaditut ominaisuudet:

- Jos $m = 1$, niin rivin 2 silmukkaehdon ensimmäinen osa keski $< N$ testataan epätodeksi. Silmukka päättyy.

- Jos $m > 1$, niin koko rivin 2 silmukkaehto testataan todeksi, koska b ei löydy vielä. Silmukka jatkuu arvolla $m - 1$.

Arvaus $T(m) = m$ täyttää palautuskaavan (1) ehdot, mikä voidaan havaita sijoittamalla arvaus kaavaan ja tarkistamalla. □

3.2 Binäärihaun analyysi

Analysoidaan kalvojen 2.4.2 binäärihakualgoritmin pahinta suoritusaikaa.

- Pahin tapaus on sellainen, jossa joka kierroksella

  - alkupuoli $A[\text{vasen} \ldots \text{keski} - 1]$ jää (1 paikan verran) lyhyemmäksi kuin loppupuoli $A[\text{keski} + 1 \ldots \text{oikea}]$

  - b täytyy hakea loppupuolesta.

  Siis sellainen, jossa $N = 2^k - 1$ ja $b > A[N]$.

Lause 3.2.1. *Rivi 3 suoritetaan pahimmassa tapauksessa* $\log_2(N+1)+2$ *kertaa.*

Todistus. Etsitään apufunktiota $T(m)$ = "rivin 3 suoritusten lukumäärä, kun $m = \ell$(vasen, oikea)". Algoritmin aikavaatimus on silloin $T(N+1)$.

Lähdekoodista saa palautuskaavan

$$T(m) = \begin{cases} 1 & \text{jos } m = 0 \\ 1 + T(\frac{m}{2}) & \text{jos } m > 0 \text{ on parillinen} \\ 1 + T(\frac{m-1}{2}) & \text{jos } m > 0 \text{ on pariton.} \end{cases}$$

Koska tarkastelemme pahinta tapausta, voidaan pariton haara jättää pois.

Muuttujanvaihdolla $m = 2^k$ saadaan uusi palautuskaava

$$T'(k) = \begin{cases} 2 & \text{jos } k = 0 \\ 1 + T'(k-1) & \text{jos } k > 0 \end{cases}$$

joka ratkaisu on $T'(k) = k + 2$ palautuskaavan (1) tyyliin.

Vaihto takaisin antaa ratkaisun $T(m) = \log_2(m) + 2$, mistä tulos seuraa. □

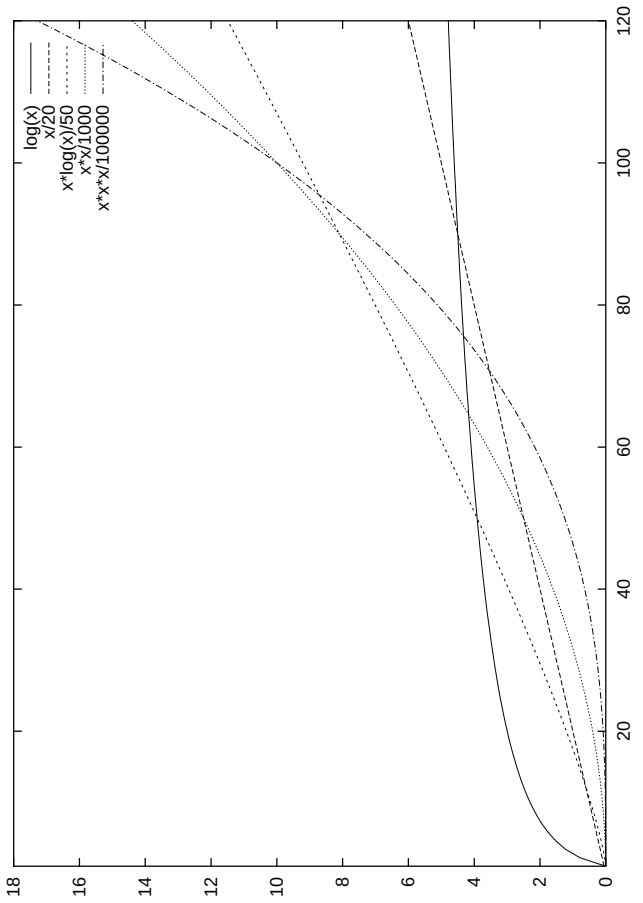
Pyöristetään vielä muut pituudet ylöspäin lähimpään pahimpaan:

Lause 3.2.2. *Rivi 3 suoritetaan korkeintaan* $\lceil \log_2(N+1) \rceil + 2$ *kertaa.*

Todistus. Annettua syötepituutta N lähin suurempi pahinta muotoa oleva syötepituus on $2^{\lceil \log_2(N+1) \rceil} - 1$. Tulos saadaan sijoittamalla se lauseeseen 3.2.1. □

- Lauseita 3.1.1 ja 3.2.2 vertaamalla voi sanoa, että *kalvojen 2.4.2 binäärihaku on parempi algoritmi kuin kalvojen 2.4.1 peräkkäishaku:*
  - Peräkkäishaku kiertää suurilla N silmukkaansa useammin kuin binäärihaku.
  - Vaikka peräkkäishaun silmukka onkin sisältä yksinkertaisempi, niin aikanaan se hyöty on syöty.
- Funktio $\log_2$ ilmestyy usein käytettäessä kalvojen 2.1 hajoita ja hallitse -periaatetta:

  Syöte on saatu *jaettua* kahteen (suunnilleen) yhtä suureen osaan, jotka on voitu käsitellä erikseen.



3.3 Funktioiden kertaluokat

- Aina emme voi (tai viitsi) analysoida algoritmin resurssitarpeita tunnetuksi lausekkeeksi.
- Funktion $g\colon \mathbb{N} \to \mathbb{R}$ *kertaluokka* $\mathcal{O}(g)$ koostuu niistä funktioista f, joilla on sellaiset vakiot $c > 0$ ja $n_0 \in \mathbb{N}$, että kaikilla $n \geq n_0$ ehto $0 \leq f(n) \leq c \cdot g(n)$ pätee (ja on siis määritelty).
- Eli: f on kertaluokassa $\mathcal{O}(g)$ jos ja vain jos voimme valita jonkin
  - kertoimen c ja
  - kohdan n_0 lukusuoralta siten,

  että sen *kohdan oikealla puolella funktion* $f(n)$ *kuvaaja pysyttelee funktion* $c \cdot g(n)$ *kuvaajan alapuolella.*

- Merkitään $f = \mathcal{O}(g)$ (eikä $f \in \mathcal{O}(g)$ kuten luulisi).

- Jos f on algoritmin resurssitarvetta kuvaava funktio, niin $f = \mathcal{O}(g)$ kertoo, että
  - tarpeeksi suurilla syötepituuksilla $n > n_0$ (kalvojen 3 periaate 1)
  - toteutuskohtaisiin vakioihin c menemättä (kalvojen 3 periaate 2)

  resurssitarpeen yläraja on $g(n)$.

- Tehdään analyysit

  aluksi systemaattisesti ja tarkasti

  myöhemmin intuitiivisesti ja lyhyesti.

Lause 3.3.1. *Jos $g(n) = \mathcal{O}(f(n))$, niin myös $f(n) + g(n) = \mathcal{O}(f(n))$.*

Todistus. Oletuksen mukaan on kohta n_0 ja vakio c siten, että kaikilla $n \geq n_0$ pätee $0 \leq g(n) \leq c \cdot f(n)$.

Silloin

$$\begin{aligned} f(n) + g(n) &\leq f(n) + c \cdot f(n) \\ &= (c+1) \cdot f(n) \end{aligned}$$

kaikilla $n \geq n_0$.

$\mathcal{O}$-määritelmä siis pätee, kun valitaan kohdaksi n_0 ja vakioksi $c + 1$. □

- Ohjelmoijan intuitio: Kauanko vie aliohjelma

  procedure per(x: **syöte jonka pituus on** n)
  Koodinpätkä joka vie $f(n)$ askelta;
  Koodinpätkä joka vie $g(n)$ askelta.

  kun toinen pätkä ei ole oleellisesti hitaampi kuin ensimmäinen?

- Funktio $f : \mathbb{N} \to \mathbb{R}$ on *asymptoottisesti positiivinen* (asymptotically positive), jos on kohta n_+ siten, että $f(n) > 0$ kaikilla $n \geq n_+$.

- Järkevät resurssifunktiot ovat tällaisia.

Lause 3.3.2. *Jos $f(n)$ on asymptoottisesti positiivinen, niin kertaluokka $\mathcal{O}(f(n))$ sisältää kaikki vakiofunktiot.*

Todistus. Mielivaltaiselle vakiofunktiolle $n \mapsto d$ voidaan valita $\mathcal{O}$-määritelmässä $c = \max(d, 1)$ ja $n_0 = n_+$. □

Lause 3.3.3. *Jos $f(n)$ on asymptoottisesti positiivinen ja d vakio, niin $f(n) + d = \mathcal{O}(f(n))$.*

Todistus. Suoraan lauseista 3.3.1 ja 3.3.2 □

Lause 3.3.4. *Jos*

1. *$f(n) - 1$ ja $g(n) - 1$ ovat asymptoottisesti positiivisia*

2. *$f(n) = \mathcal{O}(g(n))$*

niin myös $\log(f(n)) = \mathcal{O}(\log(g(n)))$.

Todistus.

- Oletuksen 1 mukaan on kohta n_a josta alkaen $f(n) > 1$.

  Samoin kohta n_b josta alkaen $g(n) > 1$.

- Oletuksen 2 mukaan on vakio $c > 0$ ja kohta n_c josta alkaen $f(n) \leq c \cdot g(n)$.

- Siis

  $$1 < f(n) \leq c \cdot g(n)$$

  pätee, kunhan $n \geq n_d = \max(n_a, n_b, n_c)$.

- Muistetaan: funktio $h(n)$ on *kasvava*, jos aina pätee $h(n) \leq h(n+1)$.

- Koska funktio $n \mapsto \log(n)$ on kasvava, saadaan

$$\begin{aligned} 0 \leq \log(f(n)) &\leq \log(c \cdot g(n)) \\ &= \log(c) + \log(g(n)). \end{aligned}$$

- Oletuksen 1 nojalla $\log(g(n))$ on asymptoottisesti positiivinen, joten lauseen 3.3.2 nojalla $\log(c) = \mathcal{O}(\log(g(n)))$.

- Siis lauseen 3.3.1 nojalla $\log(c) + \log(g(n)) = \mathcal{O}(\log(g(n)))$.

- Siis on vakio $c' > 0$ ja kohta n_e siten, että $\log(c) + \log(g(n)) \leq c' \cdot \log(g(n))$ kunhan $n \geq n_e$.

- Siis on $c' > 0$ jolla

$$0 \leq \log(f(n)) \leq c' \cdot \log(g(n))$$

kunhan $n \geq n_0 = \max(n_d, n_e)$. □