Tietorakenteet, syksy 2004

Matti Luukkainen

- Kalvot perustuvat kirjaan *Introduction to Algorithms*, Cormen et. all., sekä
- Timo Karvin *Tietorakenteet*-monisteeseen
- Kalvot eivät ole tarkoitettu itseopiskelumateriaaliksi vaan luentojen tueksi.

1. Johdanto

- Kaikki epätriviaalit ohjelmat joutuvat tallettamaan ja käsittelemään tietoa suoritusaikanaan
- Esim. "puhelinluettelo":
  - numeron lisäys
  - numeron poisto
  - numeron muutos
  - numeron haku nimen perusteella
  - nimen haku numeron perusteella
  - nimi-numero -parien tulostaminen aakkosjärjestyksessä
  - ...
- suoritusaikana tiedot tallennetaan *tietorakenteeseen*

- tietorakenteella tarkoitetaan
  - tapaa miten tieto tallennetaan koneen muistiin, *ja*
  - operaatioita joiden avulla tietoa päästään käyttämään ja muokkaamaan

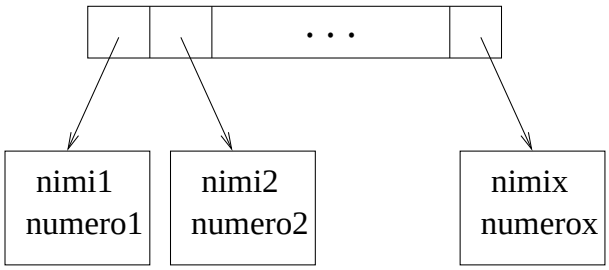
- joskus lähes samasta asiasta käytetään nimitystä *abstrakti tietotyyppi*
  - tietorakenteen sisäinen toteutus piilotetaan käyttäjältä, ja
  - abstrakti tietotyyppi näkyy käyttäjille ainoastaan operaatioina minkä avulla tietoa käytetään
  - abstrakti tietotyyppi ei siis ota kantaa siihen miten tieto on koneen muistiin varastoitu ...

- Puhelinluettelon tietorakenne voisi olla esim. seuraavanlainen
- Oleellisia operaatioita ovat (Abstrakti tietotyyppi puhelinluettelo):

- init	luodaan tyhjä luettelo
- add(n,p)	lisää luetteloon nimen n ja sille numeron p
- del(n)	poistaa nimen n luettelosta
- find(n)	palauttaa luettelosta henkilön n numeron
- update(n,p)	muuttaa n:lle numeroksi p:n
- list	listaa nimi-numero -parit aakkosjärjestyksessä

- tieto voisi olla talletettu taulukkoon suoraan

- tai taulukosta voi olla linkki varsinaisen nimi/numero-parin tallettavaan muistialueeseen



- Operaatioiden toteutus . . . laskareissa

- kuten tulemme kurssin edetessä näkemään on monia vaihtoehtoja valita tietorakenne jolla on puhelinluettelolle sopivat operaatiot
- voimme käyttää taulukon lisäksi esim. listaa, hajautusrakennetta tai puuta
- käsiteltävän tietomäärän ollessa suuri on kuitenkin oleellista että operaatiot voidaan suorittaa *tehokkaasti*
- sovelluksen tarvitsemien operaatioiden tehokkuus (tai vaativuus) oleellisesti määrää mikä tietorakenne valitaan

- kurssilla kiinnitämme siis erityisesti huomiota operaatioiden ja yleensäkin algoritmien vaativuuteen
- toinen huomioitava seikka on se toimiiko operaatio tai algoritmi oikein

- katsotaan seuraavaksi kurssin sisällysluetteloa
- tämän jälkeen "palataan asiaan" ja tarkastellaan miten oikeellisuutta ja vaativuutta voidaan arvioida

Kurssin sisältö

1. Johdanto (1 viikko)
   - mitä tarkoitetaan tietorakenteella
   - algoritmin oikeellisuuden osoittaminen
   - algoritmin vaativuuden analysointi
   - vaativuusfunktioiden kertaluokat
2. Perustietorakenteet (1 viikko)
   - lista, pino ja jono
   - periaate tietorakenteen toteuttamiseen linkitettynä rakenteena
3. hakupuut (3 viikkoa)
   - binäärihakupuut
   - tasapainoitetut puut (puna-musta-puu)
4. hajautus (1 viikko)
   - hajautusfunktion valinta
   - yhteentörmäysstrategiat
5. keko (1 viikko)
   - täydellinen binääripuu taulukkona
   - keko-operaatiot
   - prioriteettijono

6. järjestäminen (2 viikkoa)
    - lomitusjärjestäminen
    - kekojärjestäminen
    - pikajärjestäminen
    - järjestelyn alaraja
    - laskemisjärjestäminen
    - lokerikkojärjestäminen
7. verkot (3 viikkoa)
    - verkon määritelmä ja toteutus
    - verkkojen läpikäyntialgoritmit
    - lyhimmät polut
    - virittävä puu

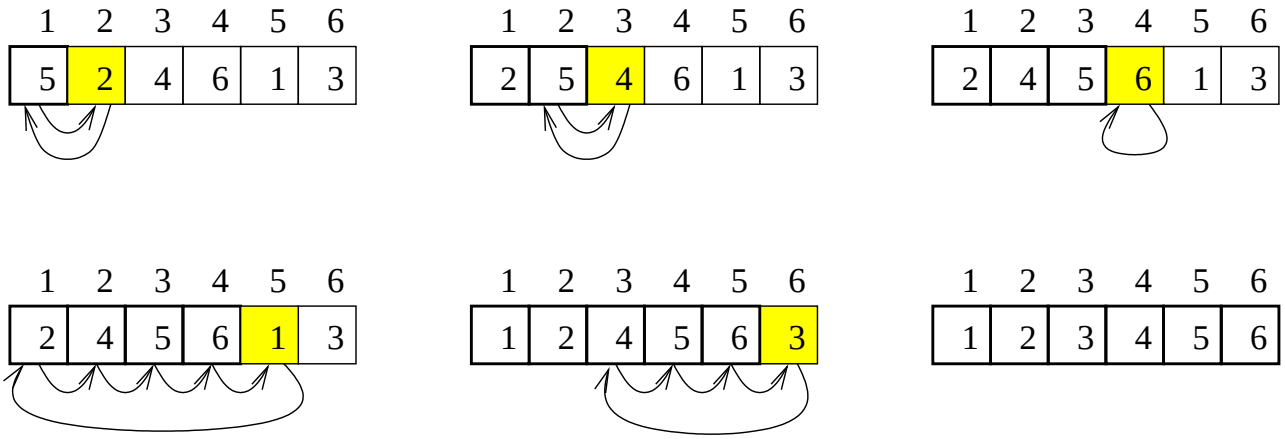
1.1 Oikeellisuus

- yksi mahdollisuus toteuttaa puhelinluettelon **add** operaatio (taulukkototeutuksessa) on lisätä uusi nimi-numero-pari aina taulukon ensimmäiseen tyhjään paikkaan
- tällöin operaation **list** yhteydessä nimet täytyy järjestää aakkosjärjestykseen ...
- järjestäminen on yleisemminkin tärkeässä roolissa tietojenkäsittelyssä ja myöhemmin kurssilla tarkastellaan muutamia tehokkaita järjestämisalgoritmeja

- tarkastellaan nyt esimerkkinä ehkä jo joillekin tuttua lisäysjärjestämistä (insertionsort)
- algoritmi *pseudokoodina*

```
1  for j ←2 to length[A] do
2      key ←A[j]
3      ▷A[j] paikalleen järjestettyyn osaan A[1,...,j-1]
4      i ←j–1
5      while i>0 and A[i]>key
6          A[i+1] ←A[i]
7          i ←i–1
8      A[i+1] ←key
```

- algoritmin toiminta esimerkkisyötteellä:



- algoritmin toimintaidea voidaan ilmaista ns. *invariantin* avulla:

  jokaisen **for***-lauseen alussa taulukon osa* $A[1, \ldots, j-1]$ *on järjestyksessä ja sisältää alunperin taulukon osassa* $A[1, \ldots, j-1]$ *olleet alkiot*

- invarianttia voidaan käyttää algoritmin oikeellisuustodistuksissa:

- näytetään että
  - invariantti on voimassa tultaessa ensimmäistä kertaa toistolauseeseen (joka on nyt **for**-lause),
  - invariantti pysyy totena jokaisen **for**-lauseen rungon suorituksen jälkeen

- **for**-lauseen suorituksen loputtua $j = n+1$ (huom for-lauseen semantiikka) ja koska invariantti on edelleen voimassa

  seuraa tästä että $A[0, \ldots, n]$ on järjestyksessä ja sisältää alunperin taulukossa olleet alkiot

  eli algoritmi toimii oikein.

- koska kyseessä on **for**-lauseke, tiedämme varmasti että sen suoritus päättyy (sillä oletuksella että komento-osa ei jää silmukkaan!), sensijaan **while**-toistolause saattaisi jäädä ikuiseen silmukkaan
- yleisen toistolauseen tapauksessa on siis invariantin voimassa pysymisen lisäksi näytettävä että toistolause pysähtyy

- huomautuksia pseudokoodista:
  - lohkorakenne ilmaistaan sisennyksen avulla (javassa aaltosulut)
  - $\triangleright$-merkillä alkava rivi on kommentti
  - sijoitus $x \leftarrow 5$ (javassa $x = 5$)
  - taulukon alkioihin viittaus tyyliin $A[i]$
  - koosteinen data olioina/tietueina joilla kentät
    esim. jos C on tyyppiä jolla kentät key ja $next$, viitataan kenttiin $key[C]$ ja $next[C]$ (esim. javassa sama tapahtuu $C.key$ ja $C.next$)
  - **for**-lausekkeen indeksimuuttujan arvo:
    esim. lausekkeen **for** $i \leftarrow 1$ **to** $10\ldots$ suorituksen jälkeen $i = 11$

1.2 Vaativuus

- Algoritmin tehokkuudella tai vaativuudella tarkoitetaan sen suoritusaikana tarvitsemia resursseja
  - aika
  - tila
  - kommunikoinnin määrä
  - ...
- tällä kurssilla tarkastellaan lähinnä aikaa ja joskus tilaa
- resurssintarvetta tarkastellaan suhteessa syötteen kokoon, esim. järjestettäessä koko on taulukon A alkioiden lukumäärä
- voimme analysoida resurssien tarvetta
  - parhaassa tapauksessa (ei yleensä kiinnosta)
  - keskimääräisessä tapauksessa
  - pahimmassa tapauksessa tapauksessa
- keskitymme kurssilla pahimman tapauksen analyysiin
  - helpompaa kuin keskimääräisen tapauksen analyysi
  - tiedämme ainakin mihin on varauduttava
  - joskus keskimääräiset tapaukset suunnilleen yhtä vaikeita kuin pahimmatkin (lisäysjärjestäminen!)
  - monilla algoritmeilla pahimmat tapaukset yleisiä

- Tarkastellaan jälleen lisäysjärjestämistä
- oletetaan että rivin i suorittaminen kestää c_i aikayksikköä
- merkitään t_j:llä kuinka monta kertaa **while**-lausekkeen testi suoritetaan kullakin j:n arvolla

		kus	kert
1	**for** j ←2 **to** length[A] **do**	c_1	n
2	key ←A[j]	c_2	$n-1$
3	▷...	0	$n-1$
4	i ←j–1	c_4	$n-1$
5	**while** i>0 and A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] ←A[i]	c_6	$\sum_{j=2}^{n} t_j - 1$
7	i ←i–1	c_7	$\sum_{j=2}^{n} t_j - 1$
8	A[i+1] ←key	c_8	$n-1$

- nyt voimme laskea kuinka kauan suoritukseen kuluu
- käytetään merkintää $T(n)$ suoritusajasta syötteellä jonka pituus on n
- laskemalla kuvasta saamme:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- *parhaassa tapauksessa* (taulukko jo järjestyksessä) **while**-testi tehdään aina vaan kerran, eli $t_j = 1$, ja saamme

  $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

- koska vakiot $c_1, \ldots, c_8$ riippuvat käytettävästä laitteistosta, voimme merkitä $T(n) = an + b$ joillain vakioilla a ja b

  $T(n)$ on lineaarinen funktio syötteen pituuteen n nähden

- *Lisäysjärjestäminen toimii parhaassa tapauksessa lineaarisessa ajassa syötteen pituuteen nähden*

- *pahimmassa tapauksessa* (taulukko käänteisessä järjestyksessä) kohdan $A[i]$ alkio viedään taulukon alkuun, eli $t_j = j$, saamme

  $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

- Sovelletaan kaavaa $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$, ja $T(n)$ sievenee muotoon:

  $T(n) = \frac{n^2}{2}(c_5 + c_6 + c_7) + \frac{n}{2}(2c_1 + 2c_2 + 2c_4 + c_5 + c_6 + c_7 + 2c_8) - (c_2 + c_4 + c_5 + c_8)$,

  eli voimme merkitä $T(n) = an^2 + bn + c$ joillain vakioilla a, b ja c

- *Lisäysjärjestäminen toimii pahimmassa tapauksessa neliöisessä ajassa syötteen pituuteen nähden*

- koska yllä esiintyvät vakiot a, b ja $c, c_1, \ldots, c_8$ ovat konekohtaisia, emme jatkossa arvioi suoritusaikaa tällä tarkkuustasolla, vaan luonnehdimme mihin *kertaluokkaan* vaativuus kuuluu
- olemme siis kiinnostuneita onko algoritmin vaativuus esim. lineaarinen, neliöinen tai jotain muuta

1.3 Funktioiden kertaluokat

- Tarkastellaan funktioita $f : \mathbb{N} \to \mathbb{R}$ ja $g : \mathbb{N} \to \mathbb{R}$
- f kuuluu kertaluokkaan $\mathcal{O}(g)$ jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n > n_0$ ehto $0 \leq f(n) \leq dg(n)$ on voimassa.

- eli f on kertaluokassa $\mathcal{O}(g)$ jos ja vain jos voimme valita
  - jonkin kertoimen d
  - ja kohdan n_0 lukusuoralta

  siten että tämä kohdan jälkeen funktioin $f(n)$ kuvaaja pysyttelee funktion $dg(n)$ kuvaajan alapuolella
- tällöin merkitään $f = \mathcal{O}(g)$
- intuitiivinen tulkinta: jos f on ohjelman resurssitarvetta kuvaava funktio ja $f = \mathcal{O}(g)$, niin
  - tarpeeksi suurilla syötepituuksilla $n > n_0$
  - toteutuskohtaisia vakioihin d menemättä

  resurssitarpeiden yläraja on g

- Palataan lisäysjärjestämiseen:
  - parhaassa tapauksessa algoritmi kulutti aikaa $an + b \leq n(a + b) = \mathcal{O}(n)$, sillä vakioksi voidaan valita $d = a + b$.
  - pahimmassa tapauksessa $an^2 + bn + c \leq n^2(a + b + c) = \mathcal{O}(n^2)$ sillä vakioksi voidaan valita $d = a + b + c$.

- jatkossa tulemme analysoimaan algoritmeja $\mathcal{O}$-merkinnän tarkkuudella

- $\mathcal{O}$-merkintää käytetään *ylärajan* esittämiseen
  - jos jonkin algoritmin pahimman tapauksen vaativuus on esim. $\mathcal{O}(n^3)$ ei se välttämättä tarkoita että pahin tapaus toimii ajassa dn^3 jollekin d, vaan että algoritmi ei toimi ainakaan huonommin kuin ajassa dn^3
  - Esim: koska $n^2 = \mathcal{O}(n^5)$, voidaan (ja on ihan oikein) merkitä että lisäysjärjestäminen on pahimmassa tapauksessa $\mathcal{O}(n^5)$
  - mielekästä onkin etsiä pienintä mahdollista argumenttifunktioita f jolla algoritmin vaativuus on $\mathcal{O}(f)$, tai ...

- vaihtoehtoisesti voidaan esittää vaativuudelle alarajoja, ja tällöin käytössä merkintä Ω

- $f = \Omega(g)$ jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n > n_0$ ehto $0 \leq dg(n) \leq f(n)$ on voimassa.

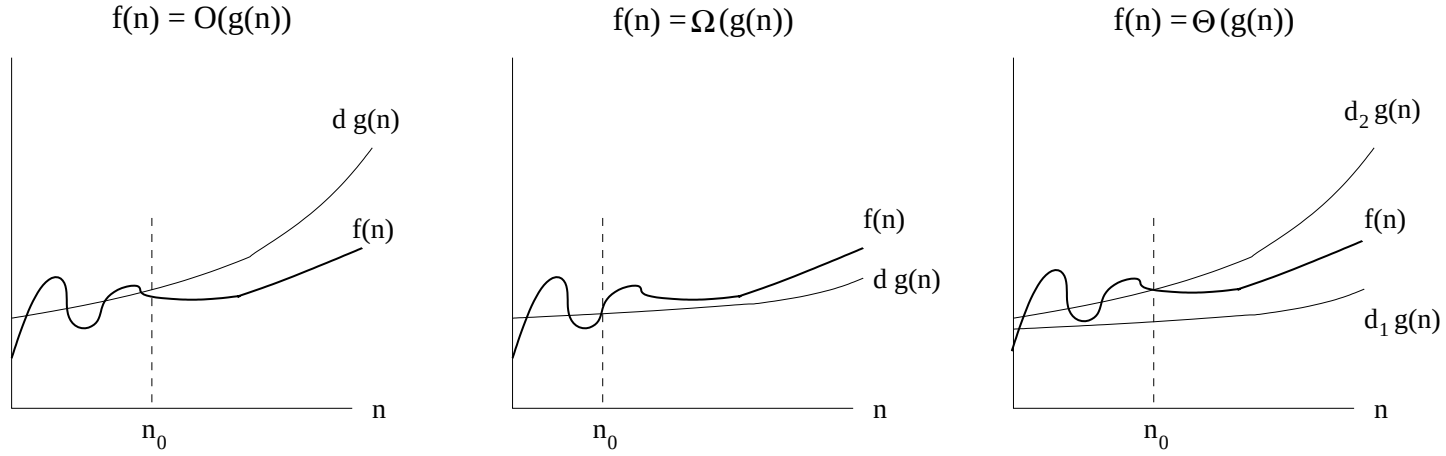
- Esim: valitsemalla $d \leq a$ huomaamme että $dn^2 \leq an^2 \leq an^2 + bn + c$ eli lisäysjärjestäminen on myös pahimmassa tapauksessa $\Omega(n^2)$

  siis ylä ja alaraja vaativuudelle on sama, eli kyseessä on tarkka arvio, myös tälle tapaukselle on olemassa oma merkintätapa

- g on sekä ylä- että alaraja f:lle, merkitään $f = \Theta(g)$, jos ja vain jos $f = \mathcal{O}(g)$ $f = \Omega(g)$
  - nyt olemassa vakiot d_1 ja d_2 siten että funktio $f(n)$ jää kuvaajien $d_1 g(n)$ ja $d_2 g(n)$ väliin
  - f käyttäytyy oleellisesti samoin kuin g toisin sanoen f:llä sama *asymptoottinen* kasvunopeus kuin g:llä

- Esim: koska lisäysjärjestämisen pahimman tapauksen yläraja $\mathcal{O}(n^2)$ sekä alaraja $\Omega(n^2)$, voimme käyttää merkintää $\Theta(n^2)$

- seuraavat kuvat valaisevat asymptoottisten merkintöjen eroja



- tärkeimmät vaativuusluokat (kasvavassa järjestyksessä) syötteen kookoon n nähden ovat
  1. $\mathcal{O}(1)$ (vakio)
  2. $\mathcal{O}(log\ n)$ (logaritminen)
  3. $\mathcal{O}(n)$ (lineaarinen)
  4. $\mathcal{O}(n\ log\ n)$
  5. $\mathcal{O}(n^2), \mathcal{O}(n^3), \ldots$ (polynomiaalinen)
  6. $\mathcal{O}(2^n)$ (eksponentiaalinen)
- sääntöjä algoritmien analysointiin:
  - yksinkertaisia muuttujia käsittelevät sijoitus-, syöttö, tulostuskäskyjen sekä niitä käsittelevien yksinkertaisten vertailujen ja aritmeettisten operaatioiden aikavaativuus on $\mathcal{O}(1)$
  - käskyjonon $S_1, S_2, \ldots, S_k$ aikavaativuus on $\mathcal{O}(f_1 + \ldots + f_k) = max(\mathcal{O}(f_1), \ldots, \mathcal{O}(f_k))$, missä $\mathcal{O}(f_i)$ käskyn S_i vaativuus
  - ehtolauseen aikavaatimus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaatimus)
  - silmukan aikavaatimus on $\mathcal{O}(k(f_1 + f_2))$ missä k toistojen määrä f_1 lopetusehdon testausaika ja f_2 silmukan lauseiden suoritusaika
  - aliohjelman/metodin aikavaativuus on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman/metodin käskyjen suoritusaika)

Esimerkki: Kuplajärjestäminen

```
1     for i ←n downto 2 do
2         for j ←1 to i–1 do
3             if A[j] > A[j+1] then
4                 ▷vaihdetaan A[j]:n ja A[j+1]:n sisältöä
5                 apu ←A[j]
6                 A[j] ←A[j+1]
7                 A[j+1] ←apu
```

- Invariantti:

  Taulukon osa $A[i{+}1, \ldots, n]$ järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin osan $A[1, \ldots, i]$ alkiot

  – alussa $i = n$ eli voimassa itsestään (loppuosa tyhjä)

  – invariantti pysyy totena jokaisen ulomman **for**-lauseen rungon suorituksen jälkeen: ensin kohtaan $A[i]$ haetaan alkuosan suurin alkio, sitten i:n arvo vähenee yhdellä

  – lopuksi $i = 2$ eli invariantista seuraa
  taulukon osa $A[2, \ldots, n]$ järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin paikan $A[1]$ alkio
  siis taulukko järjestyksessä

- Usein algoritmista voidaan eristää yksi tai useampia perusoperaatioita joiden suorituskertojen määrä kuvaa algoritmin aikavaativuutta
- nyt sopiva perusoperaatio on rivin 3 **if** vertailu
  - ensin j saa arvot väliltä $1 \ldots n-1$, eli vertailu suoritetaan $n-1$ -kertaa
  - sitten j saa arvot väliltä $1 \ldots n-2$, eli vertailu suoritetaan $n-2$ -kertaa
  - $\ldots$
  - vertailu suoritetaan siis $1+2+\ldots+n-1 = \sum_{i=1}^{n} i - n = \frac{1}{2}n(n+1) - n$ kertaa, eli aikavaativuus $\mathcal{O}(n^2)$
- huom: toisin kuin lisäysjärjestämisellä aikavaativuus on nyt sama *kaikissa tapauksissa*

Esimerkki: Binäärihaku

- Annettuna järjestyksessä oleva taulukko $A[1, \ldots, n]$ ja luku x. Onko luku taulukossa?

```
1  vasen ←1
2  oikea ←n
3  found ←false
4  while vasen ≤ oikea and not (found) do
5      keski ←(vasen+oikea)/2
6      if A[keski]=x then found ←true
7      if A[keski]>x then oikea ←keski–1
8      else vasen = keski+1
```

- Invariantti:

  jos x on taulukossa niin se on välillä $A[vasen, oikea]$ ja $found = tosi$ jos ja vain jos x löytyi jo

  - tosi alussa sillä $vasen = 1, oikea = n$ ja $found = false$
  - pysyy selvästi totena toistolauseen suorituksessa:
    jos x löytyy,asetetaan $found = true$
    jos $A[keski] > x$ niin myös kaikilla $j > keski$ on $A[j] > x$
    eli ei tarvitse etsiä kohdan $keski$ takaa ja voidaan asettaa $oikea \leftarrow keski - 1$
    else -haara vastaavasti
  - lopuksi joko $vasen > oikea$ ja $found = false$ eli etsittyä ei löytynyt, tai $x = A[keski]$

- toistolause terminoi koska jokaisella toistolla joko hakualue pienenee tai etsitty alkio löytyy

- Merkitään $T(k)$:llä binäärihaun pahimman tapauksen aikavaativuutta silloin kuin taulukosta on vielä tutkimatta k paikkaa.
- T on määritelty seuraavasti *rekursioyhtälönä*:

$$\begin{aligned} T(k) &= T(k/2) + \mathcal{O}(1), \text{kun } k > 1 \\ T(1) &= \mathcal{O}(1) \end{aligned}$$

- Oletetaan yksinkertaisuuden vuoksi että taulukon A pituus n on jokin kahden potenssi, eli $n = 2^j$ jollain kokonaisluvulla j ottamalla tästä kaksikantainen logaritmi molemmilta puolilta, saadaan $j = log_2\ n$
- Olkoon c joku tarpeeksi suuri vakio. Binäärihaun aikavaatimus saadaan laskemalla rekursiokaava auki

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + c + c + c = T(n/2^3) + 3c \\ &\cdots \\ &= T(n/2^j) + jc = (j+1)c = (log_2\ n + 1)c \\ &= \mathcal{O}(log_2\ n) \end{aligned}$$

- yksinkertaistimme analyysiä olettamalla että $n = 2^j$, toinen yksinkertaistus mitä teimme oli rekursioyhtälössä missä emme ottaneet huomioon että oikeasti taulukko ei jakaudu aina täsmälleen puoliksi
- tekemämme yksinkertaistukset eivät kuitenkaan vaikuta vaativuusanalyysin oikeellisuuteen