TKT20001 Tietorakenteet ja algoritmit (kevät 2020)

Kurssikoe 1, malliratkaisut

Tehtävän 1 tarkasti Jyrki Kivinen, tehtävän 2 Jukka Rautaoja, tehtävän 3 Marcus Leivo ja tehtävän 4 Hannu Kärnä.

1. [*4 pistettä*] *Pikajärjestäminen.* Selitä lyhyesti pikajärjestämisen toimintaperiaate. Yksityiskohtaista pseudokoodia tms. ei tarvitse antaa; sopiva vastauksen tarkkuustaso on sanallinen selitys, jota on havainnollistettu parilla kuvalla tai kaaviolla. Mitä tiedetään pikajärjestämisen aika- ja tilavaativuudesta? (Näitä ei tarvistse perustella.)

   Hyvän vastauksen pitäisi normaalikäsialalla mahtua helposti yhdelle sivulle.

   Ratkaisu: Pikajärjestämisessa taulukon alkiot järjestetään pienimmästä suurimpaan seuraavasti:

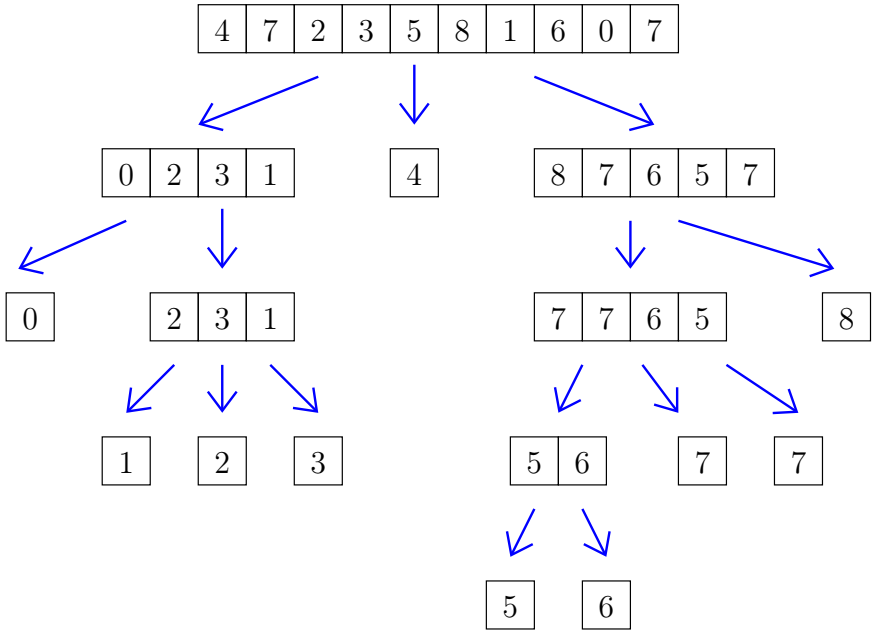
   (1) Jos taulukossa on korkeintaan yksi alkio, lopeta tekemättä mitään.

   (2) Valitse jokin taulukon alkio *jakoalkioksi*.

   (3) Järjestele taulukon alkiot niin, että jakoalkiota pienemmät alkiot tulevat jakoalkion vasemmalle puolelle ja suuremmat oikealle.

   (4) Järjestä jakoalkion vasemmalle puolelle jäävä osataulukko ja oikealle puolelle jäävä osataulukko soveltamalla kumpaankin rekursiivisesti pikajärjestämistä.

   Jakoalkion valintaan ja alkioiden järjestelemiseen kohdassa (3) on erilaisia menetelmiä. Seuraava esimerkki havainnollistaa algoritmin toimintaa, kun jakoalkioksi valitaan aina taulukon esimmäinen alkio (kuva luentomateriaalin sivulta 125):



   Algoritmin aikavaativuus on pahimmassa tapauksessa $O(n^2)$ ja keskimäärin $O(n \log n)$, kun taulukon eri järjestykset oletetaan keskenään yhtä todennäköisiksi. (Käytännössä hyvin toteutettu pikajärjestäminen on yleensä yksi nopeimmista ellei

nopein järjestämisalgoritmi.) Tilavaativuus on naiivilla toteutuksella pahimmassa tapauksessa $O(n)$ ja keskimäärin $O(\log n)$; optimoidulla totetuksella tilavaativuudeksi saadaan myös pahimmassa tapauksessa $O(\log n)$.

Kommentteja vastauksista: Vastauksissa ilmeni hyvin paljon epäselviä tai suorastaan virheellisiä käsityksiä. Seuraavassa käydään läpi yleisimpiä niistä, vaikka niistä ei kaikista olekaan arvostelussa vähennetty pisteitä.

Kun algoritmin keskimääräinen aikavaativuus on $O(n \log n)$, se ei tarkoita, että aikavaativuus on $O(n \log n)$ "yleensä", "hyvin toteutettuna" tms. Kysymyksessä on täsmällinen matemaattinen väittämä: kun syöte valitaan satunnaisesti tiettyjen oletusten vallitessa, niin suoritusajan odotusarvo on $O(n \log n)$. Pikajärjestämisen (ja yleensä järjestämisalgoritmien) tapauksessa nämä oletukset ovat, että taulukon alkiot ovat eri suuria ja niiden jokainen järjestys esiintyy yhtä todennäköisesti.

Pikajärjestämistä voisi tarkemmin sanoa perheeksi algoritmeja, koska jakoalkion valintaan ja siihen liittyvään taulukon järjestelemiseen on erilaisia vaihtoehtoja. Perusversiossa jakoalkioksi valitaan taulukon ensimmäinen alkio, mutta todelliseen käyttöön tarkoitetussa toteutuksessa parempi (mutta silti yksinkertainen) valinta olisi taulukon ensimmäisen, viimeisen ja keskimmäisen alkion mediaani, tai yleisemmin esim. viiden tasavälisesti poimitun alkion mediaani. Kaikilla näillä jakoalkion valintamenettelyillä algoritmin aikavaativuus on $O(n \log n)$ keskimäärin ja $O(n^2)$ pahimmassa tapauksessa.

Miksi sitten ensimmäisen alkion valitsemista pidetään huonona, vaikka se johtaa samaan aikavaativuusluokkaan kuin parempina pidetyt? Pääasiallinen syy on, että ensimmäisen alkion valitsevan algoritmin (eräs) pahin tapaus on valmiiksi järjestetty taulukko, jollaisia (ainakin suunnilleen) voi helposti kuvitella esiintyvän käytännön sovelluksissa. Useamman alkion mediaaniin perustuville algoritmeille on niillekin mahdollista löytää tapauksia, joissa aikavaativuus on $O(n^2)$, mutta tämä vaatii huomattavasti keinotekoisempia konstruktioita, joiden esiintyminen käytännössä on vähemmän luultavaa.

(Teoriassa on mahdollista valita jakoalkioksi koko taulukon mediaani ajassa $O(n)$. Tämä ei kuitenkaan ole käytännöllistä, koska tarvittava algoritmi on melko monimutkainen ja sen käyttäminen johtaisi suuriin vakiokertoimiin, mikä poistaisi keskeisen syyn ylipäänsä käyttää pikajärjestämistä eikä muita $O(n \log n)$-algoritmeja.)

Monessa vastauksessa oli mainittu, että ensimmäisen alkion valitseminen johtaa aikavaativuuteen $O(n^2)$, jos syötetaulukko on laskevassa järjestyksessä (kun se halutaan nousevaan). Tämä on kyllä totta, mutta oleellisempaa ja helpommin nähtävää on, että valmiiksi oikeassa järjestyksessä oleva taulukko johtaa tähän aikavaativuuteen.

Jakoalkion valinnan lisäksi algoritmin tehokkuuteen vaikuttaa, miten alkioita siirrellään halutun jaon aikaansaamiseksi. Esim. jos taulukon kaikki alkiot ovat yhtä suuria, on tietysti samantekevää, mikä niistä valitaan jakoalkioksi. Kurssimateriaalissa esitetyllä jako-proseduurilla on sekin huono puoli, että tässä tilanteessa se taas johtaa aikavaativuuteen $O(n^2)$. Tämä voitaisiin välttää toisenlaisella alkioiden siirtelystrategialla, jota emme kuitenkaan tässä rupea selvittämään.

Tilavaativuuden muodostumisessa tuntui olevan jonkin verran epäselvyyttä. Pikajärjestäminen ei käytä aputaulukoita. Tilavaativuus ei silti ole $O(1)$, sillä rekursion synnyttämä aktivaatiotietuepino vaatii tilan $O(\log n)$.

Hyvin monessa vastauksessa algoritmin toiminnan kuvauksessa todettiin, että kun alku- ja loppuosa on erikseen järjestetty rekursiivisesti, osataulukot yhdistetään. Ei niitä yhdistetä. Ne ovat valmiiksi yhdessä. Toisin kuin lomitusjärjestäminen, pikajärjestäminen tekee kaikki alkioiden siirtelyt taulukon alkuperäisen talletusalueen sisällä. Kun alku- ja loppuosa on kumpikin erikseen järjestetty, lopputulos on samalla hetkellä valmis ilman lisätoimia. Tämä osaltaan selittää, miksi kurssimateriaalissa esitetty jako-proseduuri (ja muut kirjallisuudesta löytyvät) on niin epäintuitiivinen: vaatii hieman temppuilua, että haluttu jaottelu saadaan syntymään ilman aputaulukon käyttämistä.

Arvosteluperusteet: Kolme pistettä tuli algoritmin toiminnan selityksestä ja yksi piste aika- ja tilavaativuudesta.

Toiminnan selityksen osalta

- yhteen pisteeseen riitti jonkinlainen kohtuullisen oikeansuuntainen selitys
- kahteen pisteeseen vaadittiin, että selityksen perusteella opiskelija vaikuttaa ymmärtäneen algoritmin jokseenkin oikein
- kolmeen pisteeseen vaadittiin, että toiminta oli kuvattu selvästi ja ymmärrettävästi.

Vaikka tehtävänannossa vihjattiin, että kuva olisi hyödyllinen, sellaista ei kuitenkaan ehdottomasti vaadittu, jos selitys oli muuten hyvin selvä. Tältä kannalta jako-proseduurin pseudokoodin kopioiminen vastauspaperiin ei ole hyvä havainnollistuskeino. Erilaisten jakoalkion valintamenettelyjen esittelyä ei ole vaadittu, mutta jos vastauksesta ei mitenkään käy ilmi, että ensimmäisen alkion valitseminen ei ole ainoa mahdollisuus, on vähennetty 0,5 pistettä.

Aikavaativuudesta sai 0,5 pistettä toteamalla, että keskimääräisen tapauksen aikavaativuus on $O(n \log n)$ ja pahimman tapauksen $O(n^2)$. Nämä kumpikin vaadittiin. Pisteitä ei ole vähennetty siitä hyvin yleisestä puutteesta, että aikavaativuuteen $O(n \log n)$ liittyvä ”keskimääräinen tapaus” on selitetty hyvin epämääräisesti (katso kommentteja tuonnempana). Tilavaativuuden osalta 0,5 pistettä on saanut toteamalla joko, että se on keskimäärin $O(\log n)$ ja pahimmassa tapauksessa $O(n)$ (mikä pätee kurssimateriaalissa esitetylle yksinkertaiselle toteutukselle) tai että se on aina $O(\log n)$ (mikä pätee optimoidulle toteutukselle). Toteamalla tilavaativuudeksi $O(n)$ ilman mitään täsmennystä ei ole saanut pisteitä.

2. [2 + 3 *pistettä*]

(a) Määrittele, millainen on *binäärihakupuu*. Mitä ehtoja binäärihakupuun pitää täyttää, että se olisi *AVL-puu*?

Ratkaisu: Binäärihakupuussa jokaisen solmun avain on suurempi kuin mikään avain sen vasemmassa alipuussa ja pienempi kuin mikään avain sen oikeassa alipuussa. Binäärihakupuu on AVL-puu, jos jokaisessa solmussa vasemman ja oikean alipuun korkeuksien erotus on -1, 0 tai 1.

Pisteytys: kummastakin seuraavasta erikseen yksi piste:

- On kerrottu, että binäärihakupuu koostuu solmuista, jotka on järjestetty niin, että puussa on helppo löytää suurin ja pienin alkio.
- AVL-puun tasapainoehto kerrottu. Jokaisen solmun vasemman ja oikean alipuun korkeusero on enintään 1.

(b) Esitä yksityiskohtaisena pseudokoodina algoritmi, joka saa parametrina osoittimen binääripuun juureen ja palauttaa puun lehtien lukumäärän.

Algoritmin aikavaativuuden tulee olla $O(n)$, missä n on puun solmujen lukumäärä. Aikavaativuutta ei tarvitse erikseen perustella.

Ratkaisu:

```
LEHDET(solmu)
    if solmu == null
        then return 0
    elseif solmu . vasen == null and solmu . oikea == null
        then return 1
    else return LEHDET(solmu . vasen) + LEHDET(solmu . oikea)
```

Pisteytys: kustkin seuraavasta erikseen yksi piste:

- Algoritmi laskee oikein lehtien eli lapsettomien solmujen lukumäärän.
- Algoritmi palauttaa oikein lehtien lukumäärän.
- Algoritmi ei aiheuta virhettä yrittämällä seurata null-osoitinta eikä ajaudu ikuiseen rekursioon/toistolauseeseen.

Kommentteja:

- Pisteet eivät ole riippuvaisia toisistaan, toki b-kohdan 1. ja 2. piste ovat lähellä toisiaan.
- Moni laski solmujen lukumäärän tai ei käsitellyt null-osoittimia oikein.

3. [*6 pistettä*] Mitkä ovat seuraavien algoritmien aikavaativuudet?

   Jokaisessa kohdassa anna aikavaativuudelle mahdollisimman tarkka arvio parametrin n funktiona käyttäen iso-O-merkintää ja selitä lyhyesti, miten päättelit aikavaativuuden. Selitykseksi riittää virke tai pari, joissa mainitset käyttämäsi yleiset periaatteet tms.; tarkkoja matemaattisia todistuksia ei tarvita.

   Algoritmit eivät tee mitään erityisen hyödyllistä. Kolme pistettä "..." tarkoittaa jotain vakioajassa tapahtuvaa laskentaa. Kohdissa (a)–(e) ilmoita annetun koodinpätkän aikavaativuus, kohdassa (f) kutsun `f(n)` aikavaativuus.

   (a)
   ```
   for i=1 to n
       ...
   for i=1 to n
       for j=1 to n
           ...
   ```

   Ratkaisu: Ensimmäinen silmukka suoritetaan n kertaa, joten sen aikavaativuus on $O(n)$. Toisessa silmukassa sisin osuus suoritetaan $n \cdot n$ kertaa, joten sen aikavaativuus on $O(n^2)$. Laitettaessa nämä kaksi osuutta peräkkäin suurempi aikavaativuus määrää kokonaisuuden aikavaativuuden, joten koko algoritmin aikavaativuudeksi tulee $O(n^2)$.

   (b)
   ```
   for i=1 to n
       for j=1 to i
           ...
   ```

   Ratkaisu: Sisin osuus suoritetaan i kertaa arvoilla $i = 1, \ldots, n$ eli yhteensä $1 + \ldots + n = n(n+1)/2 = O(n^2)$ kertaa. Siis aikavaativuus on $O(n^2)$.

   (c)
   ```
   j = 0
   while j < n
       ...
       j = j + 3
   ```

   Ratkaisu: Silmukka suoritetaan $\lceil n/3 \rceil$ kertaa, joten aikavaativuus on $O(n)$.

   (d)
   ```
   j = 0
   while j < n
       ...
       j = j + n/3 + 1
   ```

   Ratkaisu: Jokainen silmukan suorituskerralla laskuri kasvaa ainakin $n/3$, joten suorituskertojen lukumäärä on korkeintaan 3 eli vakio. Aikavaativuus on $O(1)$.

   (e)
   ```
   j = 1
   while j < n*n
       ...
       j = j*2
   ```

   Ratkaisu: Silmukkamuuttujan arvo kaksinkertaistuu joka suorituskerralla. Ylärajan n^2 saavuttamiseen tarvitaan noin $\log_2 n^2 = 2 \log_2 n$ kaksinkertaistusta, joten aikavaativuus on $O(\log n)$.

(f)
```
procedure f(n)
    if n == 0
        return
    f(n-1)
    f(n-1)
```

Ratkaisu: Parametrin n kasvattaminen yhdellä kaksinkertaistaa tehtävän laskennan, joten aikavaativuus on $O(2^n)$.

Kommentteja vastauksista: Todella moni oli yrittänyt perustella kohdan (a) aikavaativuutta sillä, että koodissa oli sisäkkäiset for-silmukat, mutta yleisesti ottaen tämä ei riitä todistamaan että aikavaativuus olisi $O(n^2)$. Kohdassa (b) monet olivat vastanneet aikavaativuudeksi $O(ni)$, mikä ei ole lainkaan mielekäs vastaus. Muuten tehtävä oltiin osattu suhteellisen hyvin.

Pisteytys: 0,5 pistettä oikeasta aikavaativuudesta ja 0,5 pistettä oikeasta perustelusta. Perusteluiden tuli olla tarkkuustasoltaan suunnilleen samaa luokkaa mallivastauksien kanssa.

4. [*5 pistettä*] Syötteenä on taulukko, jossa on n mielivaltaisen suurta positiivista kokonaislukua. Tehtävänä on etsiä taulukosta kahta lukua, joista pienempi on tasan puolet suuremmasta; siis luvut x ja y, joilla $y = 2x$. Jos tällaiset kaksi lukua löytyy, vastauksena tulee palauttaa niistä pienempi. Jos ratkaisuja on useita, mikä tahansa niistä kelpaa. Jos tällaisia kahta lukua ei ole taulukossa, vastauksena tulee palauttaa -1.

   Suunnittele ongelmaan ratkaisualgoritmi, jonka pahimman tapauksen aikavaativuus on $O(n \log n)$. Ratkaisussasi esitä algoritmi tehtäväpaperin alussa selitettyjen periaatteiden mukaisesti. Selitä myös lyhyesti ratkaisusi toimintaperiaate, erityisesti sen mahdollisesti käyttämät tietorakenteet. Perustele algoritmisi aikavaativuus.

   Esimerkki Jos syötteenä on taulukko $[3, 22, 7, 5, 11, 5]$, algoritmi palauttaa arvon 11, sillä $22 = 2 \cdot 11$. Jos taas syötteenä on taulukko $[8, 5, 14, 13, 6, 1]$, algoritmi palauttaa -1, koska mikään luku ei ole tasan puolet jostain toisesta.

   Ratkaisu 1: Käydään taulukko kerran läpi ja talletetaan sen alkiot aluksi tyhjään tasapainotettuun binäärihakupuuhun. Sen jälkeen käydään taulukko toisen kerran läpi ja jokaisen alkion x kohdalla tarkastetaan, löytyykö $2x$ hakupuusta. Jos löytyy, palautetaan x. Jos taulukon läpikäynti ei johda tulokseen, palautetaan -1.

   Algoritmissa käydään n-alkioinen taulukko kaksi kertaa läpi, ja kummallakin läpikäynnillä tehdään jokaista taulukon alkiota kohti yksi tasapainotetun hakupuun operaatio. Hakupuussa on enimmillään n avainta. Siis lisäys- ja hakuoperaatio menevät ajassa $O(\log n)$. Kokonaisaikavaativuus on $O(n) \cdot O(\log n) + O(n) \cdot O(\log n) = O(n \log n)$.

   Ratkaisu 2: Järjestetään ensin taulukko lomitusjärjestämisellä. Sen jälkeen käydään taulukko alkio kerrallaan läpi, ja alkion x kohdalla tarkastetaan binäärihaulla, onko alkio $2x$ taulukossa. Jos $2x$ löytyi, palautetaan x. Jos taulukon läpikäynti ei johda tulokseen, palautetaan -1.

   Lomitusjärjestäminen vie ajan $O(n \log n)$. Koska läpikäynnissä tehdään jokaista taulukon alkiota kohti yksi binäärihaku ajassa $O(\log n)$ ja lisäksi vakioaikaista laskentaa, läpikäynti menee ajassa $O(n \log n)$, joka siis on myös algoritmin kokonaisaikavaativuus.

   Ratkaisu 3:

```
etsiTupla(A[0...n-1])
    mergeSort(A)
    vasen = 0
    oikea = 1
    while oikea < n
            if A[oikea] == 2 · A[vasen]
               then return A[vasen]
            if A[oikea] < 2 · A[vasen]
               then oikea = oikea +1
            else vasen = vasen +1
    return -1
```

   Algoritmin aikavaativuus on $O(n \log n)$, joka tulee lomitusjärjestämisestä. While-silmukassa aina $vasen \leq oikea < n$ ja muuttujista *vasen* ja *oikea* toinen kasvaa

yhdellä, joten silmukka suoritetaan korkeintaan $2n$ kertaa. Koska laskenta silmukan sisällä on vakioaikaista, silmukan aikavaativuus on $O(n)$.

Selvästi jos algoritmi palauttaa jonkin muun arvon kuin -1, se on kelvollinen ratkaisu. Perustellaan vielä, että jos kelvollinen ratkaisu on olemassa, algoritmi aina löytää sen. Oletetaan tätä varten, että joillain p ja q pätee $A[q] = 2 \cdot A[p]$.

Niin kauan kuin $vasen \leq p$, muuttuja *oikea* ei voi saada arvoa $q+1$. Tällöin nimittäin taulukon järjestyksen takia $A[q] = 2 \cdot A[p] \geq 2 \cdot A[vasen]$, joten tilanteessa $oikea = q$ ehto *oikea*-muuttujan kasvattamiselle ei toteudu.

Vastaavasti kun $oikea \leq q$, muuttuja *vasen* ei voi saada arvoa $p+1$: pätee $A[oikea] \leq A[q] = 2 \cdot A[p]$, joten tilanteessa $vasen = p$ ehto *vasen*-muuttujan kasvattamiselle ei toteudu.

Siis niin kauan kuin $vasen \leq p$, pätee myös $oikea \leq q$; ja niin kauan kuin $oikea \leq q$, pätee myös $vasen \leq p$. Koska aluksi kumpikin ehto pätee, ne pysyvät kumpikin voimassa loppuun asti. Jos algoritmi ei palauta jotain toista ratkaisua, niin joka askelella joko *vasen* tai *oikea* kasvaa. Siis lopulta saavutetaan tilanne, jossa $vasen = p$ ja $oikea = q$, ja algoritmi palauttaa arvon $A[p]$.

Kommentteja vastauksista: Binäärihaku ei toimi järjestämättömässä taulukossa. Pikajärjestäminen voi pahimmassa tapauksessa vaatia ajan $O(n^2)$ ja yksi hajautustaulun operaatio ajan $O(n)$.

Koska luvut olivat mielivaltaisen suuria, ei tehtävää voinut ratkaista oikein esim. kirjaamalla luvut boolean-taulukkoon, jossa indeksiä k vastaava arvo kertoo, onko luku k syötteessä. Sitä varten pitäisi voida tallettaa mielivaltaisen suuri taulukko. Siten ei myöskään ole olemassa hajautusfunktiota, joka pahimmassa tapauksessa jakaisi talletettavat arvot tasaisesti hajautustauluun. Oli talletustila miten suuri vain, voidaan riittävän suuria lukuja valitsemalla muodostaa syöte, jonka jokaisella luvulla on sama hajautusarvo.

Jotkin vastaukset perustuivat väärään oletukseen, että $y = 2x$ tulee syötteessä aina myöhemmin kuin x, että x ja y ovat järjestämisen jälkeen taulukossa peräkkäin, x on aina pariton tai y on taulukon suurin arvo.

Joskus etsittiin nousevaan järjestykseen järjestetystä taulukosta x:ää vain y:n oikealta puolelta tai y:tä x:n vasemmalta puolelta. Vastaava tapahtui hajautustaulua käyttäen, jos sinne lisättiin taulukon alkioita pienimmästä alkaen ja samalle etsittiin alkion arvoa kahdella kerrottuna. Tupla-arvo ei vielä sillä hetkellä voi olla talletettuna. Ratkaisu toimi oikein – oli taulukko järjestetty tai ei – jos kunkin lisäyksen jälkeen etsittiin sekä kaksinkertaista arvoa että puolta pienempää.

Jos kuitenkin esim. lisättiin hajautustauluun sekä taulukon alkion arvo että arvo kerrottuna kahdella, ja sitten etsittiin arvon puolikasta, päädyttiin hyväksymään myös $y = 4x$. Esim. syöte on $\{1, 4\}$. Talletetaan 1 ja $2 \cdot 1 = 2$. Etsitään arvon 1 puolikasta, ei löydy. Talletetaan 4 ja $2 \cdot 4 = 8$. Etsitään arvon 4 puolikasta, löydetään ja palautetaan virheellisesti luku 2, jota ei syötteessä ollut. Tämän tapaisesta virheestä esiintyi muutamaa muunnelmaa.

Seuraavan silmukan suoritusaika ei ole $O(n)$:

```
i = 1, j = 2
while i < n
```

```
            if t[j] == 2*t[i] return t[i]
            if j == n
                    i++
                    j = i+1
            else j++
    return -1
```

Silmukan lopetus perustuu muuttujaan `i`, mutta ensisijaisesti silmukassa kasvatetaankin vain muuttujaa `j`. Muuttujan `i` arvo kasvaa vasta, kun `j` on pääsyt arvoon `n`, jolloin `j` palaa alkuun. Sama algoritmi tutummin kirjoitettuna:

```
for i = 1 to n-1
        for j = i+1 to n
                if t[j] == 2*t[i] return t[i]
return -1
```

Vaikka `j` käy i:n kasvaessa yhä pienemmän alueen läpi, ei suoritusaika ole myöskään $O(n \log n)$, kuten useassa vastauksessa arvioitiin, vaan $O(n^2)$. Keskimäärin sisempi silmukka suoritetaan noin $n/2$ kertaa, ei $\log n$ kertaa. Muita toistuvia virheitä aikavaativuuden arvioissa oli, että koko syötteen talletus tasapainotettuun binäärihakupuuhun tapahtuisi ajassa $O(n)$.

Järjestetyn taulukon pystyy tarkastamaan lineaarisessa ajassa kahta osoitinta käyttäen mallivastauksen Ratkaisun 3 tapaan. Jos kuitenkin toista osoitinta askelletaan alusta loppuun päin ja toista lopusta alkuun päin tai muuten väärin, jää joko joitakin mahdollisia lukupareja tutkimatta tai sitten algoritmi muuttuu aikavaativuudeltaa neliölliseksi, kun toinen osoitin joudutaan välillä palauttamaan alkupisteeseensä. Läpikäynnin optimoinnit eivät pudota aikavaativuutta luokkaan $O(n)$, jos edelleen kutakin arvoa verrataan useaan muuhun arvoon, joiden määrä kasvaa suorassa suhteessa syötteen kokoon. Joissakin ratkaisuissa oli käytetty apuna listaa tai toista taulukkoa, joka oli joko suora kopio järjestetystä taulukosta tai sen arvot kaksinkertaisena / puolitettuna. Myös näin oli mahdollista päätyä löytämään lukupari tavoitellussa ajassa $O(n)$, ohittamaan mahdollisia lukupareja tai käyttämään aikaa $O(n^2)$.

Arvosteluperusteet:

5 pistettä: kattava kuvaus ja aikavaativuusarvio algoritmista, jonka pahimman tapauksen aikavaativuus on $O(n \log n)$

4,5 pistettä: pieni virhe yllä mainitussa

4 pistettä:

- kattava kuvaus ja aikavaativuusarvio algoritmista, jonka keskimääräinen aikavaativuus on $O(n \log n)$: käytettiin hajautukseen perustuvaa tietorakennetta tai pikajärjestämistä tai määrittelemätöntä järjestämisalgoritmia *tai*
- puutteellinen / virheellinen kuvaus ja aikavaativuusarvio algoritmista, jonka pahimman tapauksen aikavaativuus on $O(n \log n)$

3,5 pistettä:

- pienen virheen sisältävä kuvaus ja aikavaativuusarvio algoritmista, jonka keskimääräinen aikavaativuus on O(nlog n) *tai*

- muuten virheetön algoritmi ja aikavaativuusarvio, jonka toimivuus perustuu siihen, että voidaan käyttää taulukkoa, jonka pituus on yhtä suuri kuin syötteen arvoalue, vaikka arvoalue oli tehtävänannossa määritelty mielivaltaisen suureksi

3 pistettä: puutteellinen / virheellinen kuvaus ja aikavaativuusarvio algoritmista, jonka keskimääräinen aikavaativuus on $O(n \log n)$, esim. vaativuutta ei ole arvioitu lainkaan

2,5 pistettä: algoritmi koostuu peräkkäisistä osista, joiden aikavaativuus on $O(n \log n)$, mikä on analysoitu oikein, mutta ratkaisussa on vakava ajatusvirhe tai puute

2 pistettä: pyritty $O(n \log n)$-ratkaisuun käyttämällä tehokkaaksi tunnettuja algoritmeja ja tietorakenteita, mutta oleellisesti puutteellinen / virheellinen vastaus, vähän enemmän oikeaa kuin 1,5 pisteen vastauksessa

1,5 pistettä:

- täsmällisesti kuvattu $O(n^2)$ aikavaativuuden ratkaisu *tai*
- pyritty $O(n \log n)$-ratkaisuun käyttämällä tehokkaaksi tunnettuja algoritmeja ja tietorakenteita, mutta oleellisesti puutteellinen / virheellinen vastaus; ei ole ilmeistä tapaa korjata ratkaisu toimivaksi, vaan tarvitaan oleellinen muutos, johon ei löydy viitteitä vastauksesta

1 piste:

- aikavaativuuden $O(n^2)$ ratkaisu, jossa puutteita *tai*
- oleellisesti virheellinen ratkaisu, jossa kuitenkin oikeaa ajatusta ja pyritty tehokkuuteen.