Toinen kurssikoe

- Koe to 6.5 klo 9-12 salissa A111, **koeaika normaalista poiketen joko 2h 45 min tai 2h 50min**
- Kokeessa saa olla mukana A4:n kokoinen kaksipuoleinen kynällä tehty, itse kirjoitettu lunttilappu
- lunttilapun teossa ei siis saa käyttää kopiokonetta eikä tietokonetta
- kannattaa huomata, että lunttilapun takia kokeen kysymyksissä pääpaino on kurssilla käsiteltävien asioiden soveltaminen
- koekysymykset ovat tyyliltään esim. seuraavanlaisia
  - minkälainen on tietorakenne x
  - selitä miten algoritmi x toimii
  - kehitä algoritmi ongelmaan x
  - määritä algoritmin x aikavaativuus
  - todista, että ...
- eli koetehtävät ovat pääosin tyyliltään laskaritehtävien kaltaisia

Tärkeää ja vähemmäntärkeää

- koealue monisteen sivut 268-514 ja laskuharjoitukset 8-12
- Toisen periodin neljä ydinteemaa
  - hajautus, keko, järjestäminen ja verkot
  - Kokeessa on kysymyksiä kaikista ydinteemoista
- Edellisten lisäksi aika- ja tilavaativuusanalyysi on edelleen osattava
- Toisen kurssikokeen kannalta vähemmän tärkeää
  - Java-spesifiset asiat
  - Alaraja vertailuihin perustuvalle järjestämiselle (s 364-368)
  - järjestäminen lineaarisessa ajassa (s 364-375)
  - Union-find-tietorakenteen yksityiskohdat (s. 507-513)
- B+-puista tai A*-algoritmista ei tule kysymyksiä toiseen kurssikokeeseen
- seuraavilla kalvoilla on listattu tärkeimpä teemoja eri aihepiireistä
- teknistä asiaa kalvoilla ei juuri ole, eli ne toimivat vain lukuohjeena ja varsinainen asia löytyy luentokalvoista

Aika- ja tilavaativuusanalyysi

- Aikavaativuus: algoritmin käyttämä laskenta-"aika" syötteen koon funktiona
- Tilavativuus: algoritmin käyttämän aputilan määrä syötteen koon funktiona
- Tarkastellaan lähes koko kurssin ajan **pahimman tapauksen vaativuutta**, miten paljon aikaa/tilaa algoritmin suorituksessa kuluu enimmillään
  - poikkeuksena hajautus ja pikajärjestäminen, joissa kiinnostavaa on keskimääräisen tapauksen vaativuus
- Kiinnostavaa on mihin **vaativuusluokkaan** algoritmi kuuluu
  - $\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, ...
- usein analyysissä riittävät seuraavan kalvon nyrkkisäännöt
- joidenkin verkkoalgoritmien aikavaativuusanalyysissä nyrkkisääntöjen suoraviivainen soveltaminen ei riitä
- **rekursiivisten algoritmien tilavaativuuden analysoinnissa on otettava huomioon rekursion syvyys (ks. monisteen sivut 58 ja 61)**
  - koealueeseen kuuluu paljon rekursiivisia algoritmeja, joten muista kerrata tämä asia!

Algoritmien aikavaativuusanalyysin nyrkkisäännöt

1. yksinkertaisten käskyjen aikavaativuus on vakio eli $\mathcal{O}(1)$
2. peräkkäin suoritettavien käskyjen aikavaativuus on sama kuin käskyistä eniten aikaavievän aikavaativuus
3. ehtolauseen aikavaativuus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)
4. aliohjelman suorituksen aikavaativuus on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman käskyjen suoritusaika)
5. looppeja sisältävän algoritmin aikavaativuuden arvioiminen:
   - arvioi sisimmän loopin runko-osan aikavaativuus: $\mathcal{O}(runko)$
   - arvioi kuinka monta kertaa sisin looppi yhteensä suoritetaan: lkm
   - aikavaativuus on $\mathcal{O}(lkm \times runko)$
6. rekursiota käyttävien algoritmien aikavaativuuden arvioiminen:
   - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(runko)$
   - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
   - aikavaativuus on $\mathcal{O}(lkm \times runko)$

Hajautus

- Käyttötarkoitus:
  - abstrakti tietotyyppi joukko
  - insert, delete ja search *keskimäärin* nopeita eli $\mathcal{O}(1)$
  - pahimmassa tapauksessa kuitenkin $\mathcal{O}(n)$
  - muut joukon operaatiot $\mathcal{O}(n)$
  - oikea täyttösuhde n/m, missä n talletettujen alkioiden määrä ja m hajautusrakenteen koko, tärkeä!
- yhteentörmäys ja sen selvitysstrategiat
  - ketjutus
  - avoin hajautus
- hajautusfunktion valinta oleellinen tehokkuuden kannalta
  - jakojäännösmenetelmä
  - kertolaskumenetelmä
  - universaalihajautus

- millä perusteilla hajautusfunktion valinta tapahtuu?
  - saadaanko hajautustaulun koko valita vapaasti
- jos hajautettavat avaimet ovat muun tyyppisiä kuin lukuja, esim. merkkijonoja, miten tällöin toimitaan?
  - ensin muunnos avaimen tyypistä etumerkittömäksi kokonaisluvuksi ja sitten hajautus
  - tai suoraan kuvaus esim. merkkijonolta hajautusalueelle
- avoimen hajautuksen erilaiset kokeilustrategiat:
  - lineaarinen kokeilu, neliöinen kokeilu, kaksoishajautus
- entä jos hajautusalue tulee liian täyteen?
  - uudelleenhajautus, vie aikaa suoritettaessa mutta kannattaa pidemmän päälle
- missä tilanteessa hajautusta kannattaa käyttää ja missä ei? mikä on vaihtoehto?

Keko

- kaksi versiota: minimikeko ja maksimikeko
- muutama tehokkaasti eli ajassa $\mathcal{O}(\log n)$ toimiva operaatio
  - heap-insert
  - minimikeossa heap-del-min, heap-dec-key
  - maksimikeossa heap-del-max, heap-inc-key
  - kekoehdon korjaava apuoperaatio heapify
- Käyttötarkoitus:
  - prioriteettijono (minimikeko)
  - kekojärjestäminen (maksimikeko)

- keko ajatellaan binääripuuna
- kekoehdot
  - vanhempi suurempi kuin lapset (maksimikeossa)
  - "mahdollisimman" täysi, lehdet mahdoillisimman vasemmalla
  - katso tarkemmin sivulta 309
- keko on melkein täydellinen binääripuu (s. 135 ja 137)
- tästä seuraa, että keko on tasapainoinen ja sen korkeus solmumäärän n suhteen on vain noin $\log n$
- operaatiot kulkevat pahimmassa tapauksessa keon juuresta lehteen tai lehdestä juureen
  - operaatioiden aikavaativuus $\mathcal{O}(\log n)$
  - ks. esimerkiksi heapify s. 313 tai heap-insert s. 318
- vaikka keko ajatellaan puuna, se talletetaan taulukkoon
  - ei tarvita viitteitä: vie vähän muistia ja liikkuminen keossa nopeaa

Järjestäminen

- Kiinnostavimpia tietysti ajassa $\mathcal{O}(n \log n)$ toimivat, eli
  - kekojärjestäminen
  - lomitusjärjestäminen
  - pikajärjestäminen
- **Kekojärjestäminen**
  - tehdään taulukosta maksimikeko, toimenpide vaatii $n/2$ heapify-operaatiota
  - viedään alkiot taulukkoon paikalleen kutsumalla n kertaa heap-del-max
  - koska heapifyn ja heap-del-max:n aikavaativuus on $\mathcal{O}(\log n)$, on aikavaativuus kokonaisuudessaan $\mathcal{O}(n \log n)$
  - algoritmi ei vaadi kuin vakiomäärän aputilaa
- Lomitus- ja pikajärjestäminen perustuvat **hajoita-ja-hallitse** (engl. divide-and-conquer) -tekniikkaan:
  - *hajoitetaan* ongelma pienempiin osaongelmiin
  - *hallitaan*, eli ratkaistaan osaongelmat rekursiivisesti
  - *yhdistetään* osaratkaisut siten että saadaan ratkaisu koko ongelmalle

- molemmat toteuttavat periaatetta hieman eri tavalla

- lomitusjärjestämisessä hajoittaminen tarkottaa taulukon puolittamista, yhdistämisessä taas tehdään lomitusoperaatio merge

```
merge-sort(A,p,r)
1   if p<r
2       q = ⌊(p + r)/2⌋
3       merge-sort(A,p,q)
4       merge-sort(A,q+1,r)
5       merge(A,p,q,r)
```

- pikajärjestämisessä hajoittamisessa siirrellään alkiota siten, että pienet menevät vasemmalle ja isot oikealle puolelle, yhdistämisvaihetta ei tarvita!

```
quick-sort(A,p,r)
1   if p<r
2       q = partition(A,p,r)
3       quick-sort(A,p,q)
4       quick-sort(A,q+1,r)
```

- hallintavaihe tarkoittaa (useimmiten) rekursiivista kutsua, jolla ratkaistaan alkuperäisen ongelman osaongelma

Lomitusjärjestäminen

- hajoittaminen vaan etsii taulukonosan puolen välin, eli operaatio on kevyt $\mathcal{O}(1)$
- yhdistämisvaihe eli merge käy molemmat puolikkaat läpi ja vie aikaa $\mathcal{O}(k)$ jos yhdistettävien puolikkaiden yhteenlaskettu koko on k
- yhdistämisvaihe tarvitsee aputaulukon eli vie tilaa $\mathcal{O}(k)$
- algoritmin aikavaativuuden selvittämisessä kannattaa miettiä rekursiopuuta (s. 344-346)
  - lasketaan rekursiivisen funktion rungon vievä aika (joka on käytännössä sama kuin mergeen menevä aika) yhteen tasoittain
  - jokaisen tason merget vievät yhteenlaskien aikaa $\mathcal{O}(n)$
  - huomioidaan että tasojen lukumäärä on $\log n + 1$ jos taulukon koko on n (s. 346)
  - tästä päädytään aikavaativuuteen $\mathcal{O}(n \log n)$
- tilavaativuus:
  - rekursion takia $\mathcal{O}(\log n)$
  - merge-operaation $\mathcal{O}(n)$ kuitenkin dominoi tilavaativuutta

Pikajärjestäminen

- hajoittaminen eli partition vie *jakoalkiota* pienemmät taulukon vasemmalle ja suuremmat oikealle puolelle (s 350-351)
  - jakoalkio ja sen kanssa samansuuruiset alkiot tulevat yleensä taulukon oikealle puolelle
- partition-operaatio käy koko taulukonosan läpi eli sen aikavaativuus on $\mathcal{O}(k)$ jos hajoitettavassa taulukonosassa k alkiota
- yhdistämisvaihetta ei siis tarvita koska rekursiivisten kutsujen jälkeen sekä vasen että oikea puoli taulukosta järjestyksessä ja pienet alkiot ovat kaikki vasemmalla ja isot oikealla puolella
- jakoalkion valinta
  - ratkaisee kuinka tasan jako menee
  - jaon tasaisuuden takia paras valinta olisi jaettavan taulukonosan alkioiden mediaani
  - mediaanin valinta ei kannata sillä se hidastaisi algoritmia liikaa
  - hyvä valinta "kolmen mediaani", ks s. 363

- aikavaativuus:
  - pahin tapaus: jos partition jakaa alkiotalähes aina huonosti (toiselle puolelle vain yksi tai muutama alkio), toimii algoritmi ajassa $\mathcal{O}(n^2)$ s. 356
  - keskimäärin algoritmi toimii ajassa $\mathcal{O}(n \log n)$, keskimääräisen tapauksen aikavaativuusanalyysi ei kuulu kurssille
  - algoritmin pahin tapaus on erittäin harvinainen
- tilavaativuuden ratkaisee rekursion syvyys
  - keskimäärin $\mathcal{O}(\log n)$
  - pahimmassa tapauksessa algoritmin perusversiossa $\mathcal{O}(n)$
- käytännössä pikajärjestämistä kannattaa viritellä, s. 361-363

Järjestäminen algoritmien esiprosessointivaiheena

- järjestä ensin käsiteltävä data niin algoritmin jatkotoimet helpottuvat, esim.
  - lh10 tehtävät 1 ja 2
  - Kruskalin algoritmi
  - ...

Verkot

- solmujen joukko V, kaarien joukko E
  - kaarien lukumäärästä käytetään merkintää $|V|$ ja solmujen lukumäärästä $|E|$
- Käsitteet:
  - suuntaamaton verkko, suunnattu verkko
  - kaaripainot vai ei
  - vierussolmu
  - polku, sykli
  - saavutettavuus
  - syklitön verkko
  - vahvasti yhtenäinen verkko
  - yhtenäinen verkko
- esitystavat
  - vieruslista
  - vierusmatriisi

Verkon läpikäynti

- idea siis käydä kaikissa verkon solmuissa tai kaikissa lähtösolmusta s saavutettavissa olevissa solmuissa
- verkoilla ei kaaripainoja tai kaaripainoja ei huomioida!
- algoritmit toimivat sekä suunnatuilla että suuntaamattomilla verkoilla
- **leveyssuuntainen läpikäynti** (breath first search eli bfs)
  - edetään taso kerrallaan, ensin vieraillaan lähtösolmun vierussolmuissa, sitten vierussolmujen vierussolmuissa, . . .
  - merkataan solmut vierailluiksi (musta väri) jotta algoritmi ei jää looppiin
  - selvittää lyhimmät polut lähtösolmusta muihin solmuihin
  - polun pituudella tarkoitetaan kaarien määrää, *kaaripainoja ei siis huomioida algoritmissa!*

- **syvyyssuuntainen läpikäynti** (depth first search eli dfs)
  - edetään polku kerrallaan
  - merkataan solmut ensin löydetyksi (harmaa) ja lopulta käsitellyksi (musta)
  - harmaat solmut ovat niitä joista saavutettavien solmujen tutkiminen on vielä kesken
- syvyyssuuntaisen läpikäynnin sovellukset:
  - verkon syklittömyyden tarkastus: verkossa sykli jos ja vain jos etsinnän aikana kohdataan harmaa solmu
  - syklittömän verkon solmut voidaan järjestää dfs:n avulla topologisesti: kun solmu muuttuu mustaksi, laitetaan se pinoon
  - vahvasti yhtenäisten komponenttien selvittäminen
- leveys- ja syvyyssuuntaista algoritmia voidaan siis soveltaa suuntaamattomiin ja suunnattuihin verkkoihin
- topologinen järjestäminen ja vahvasti yhtenäisten komponenttien selvittäminen mielekästä vain suunnatuille verkoille

Verkkoalgoritmien aikavaativuus

- algoritmien aikavaativuus riippuu yleensä *sekä solmujen lukumäärästä* $|V|$ *että kaarien lukumäärästä* $|E|$

- tarkastellaan syvyyssuuntaista läpikäyntiä

```
DFS(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   DFS-visit(G,s)

DFS-visit(G,u)
4   color[u] = gray
5   for jokaiselle solmulle v ∈ Adj[u]      // kaikille u:n vierussolmuille v
6       if color[v]==white                 // solmua v ei vielä löydetty
7           DFS-visit(G,v)
8   color[u] = black
```

- riveillä 1-2 suoritetaan for-looppi jokaiselle solmulle, eli $|V|$ kertaa, forin runko-osan vaativuus $\mathcal{O}(1)$, eli nyrkkisäännön 5 (ks. kalvo 4) perusteella rivien 1-2 aikavaativuus on $\mathcal{O}(|V|)$

- rivin 3 aikavaativuus on nyrkkisäännön 4 perusteella sama kuin rekursiivisen kutsun DFS-visit aikavaativuus

- mikä on rekursiivisen kutsun DFS-visit aikavaativuus?
  - nyrkkisääntö 6 neuvoo laskemaan rekursiivisten kutsujen lukumäärän ja arvioimaan runko-osan suorittamiseen menevän ajan
  - rekursiivinen kutsu suoritetaan enimmillään kerran jokaiselle solmulle, eli kutsuja on $|V|$ kappaletta
  - kutsu nimittäin suoritetaan vain jos solmu on valkoinen ja jokainen valkoinen solmu muuttaa heti kutsun tapahduttua värinsä
  - entä rekursiivisen metodin runko-osan vaativuus?
  - runko-osassa on kaksi vakioaikaista käskyä (rivit 4 ja 8) sekä for-looppi, jossa tutkitaan käsiteltävän solmun u vierussolmut
  - for-loopin suorituskerrat vaihtelevat jokaisen rekursiivisen kutsun osalta
  - yhteenlaskettuna kaikkien rekursiivisten DFS-visit-kutsujen aikana käydään läpi jokaisen solmun jokainen vierussolmu
  - toisin sanoen for:eissa käydään kaikki rekursiiviset kutsut huomioiden läpi kaikki verkon kaaret
- jos verkko on suuntaamaton, käydään jokainen kaari läpi molempiin suuntiin

- kaikkien rekursiivisten DFS-visit-kutsujen runko-osien yhteenlaskettu aikavaativuus on siis $\mathcal{O}(|V| + |E|)$
  - $|V|$ tulee rekursiivisten kutsujen lukumäärästä
  - $|E|$ taas tulee kaikkien rekursiivisten kutsujen for-lauseiden runkojen yhteenlasketusta suoritusmäärästä
  - nyrkkisääntöä 6 ei voitu soveltaa suoraan, sillä rekursiivisten kutsujen vaativuus vaihtelee kutsuittain

- algoritmin rivit 1-2 siis vievät aikaa $\mathcal{O}(|V|)$ ja rivi 3, joka sisältää rekursiivisen kutsun vie aikaa $\mathcal{O}(|V| + |E|)$

- nyrkkisäännön 2 perusteella tämä on yhteensä $\mathcal{O}(|V| + |E|)$

- jos verkon jokaisen solmun välillä on kaari, niin $|E| = |V|^2$
  - tälläisessä tilanteessa leveyssuuntaisen läpikäynnin aikavaativuus voidaan ilmaista pelkkien solmujen lukumäärän suhteen ja se on $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + |V|^2) = \mathcal{O}(|V|^2)$
- tilanne on sama, jos verkossa on esim. jokaisesta solmusta kaari kolmasosaan muita solmuja:
  - nyt $|E| = \frac{1}{3}|V|^2$
  - ja läpikäynnin aikavaativuus solmujen määrän suhteen on $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + \frac{1}{3}|V|^2) = \mathcal{O}(|V|^2)$
- jos taas kaaria on vähän, esim. korkeintaan 3 kaarta jokaisesta solmusta, on $|E| \leq 3 \cdot |V|$ eli $|E| = \mathcal{O}(|V|)$
  - läpikäynnin aikavaativuus voidaan ilmaista pelkkien solmujen määrän suhteen ja se on $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + 3 \cdot |V|) = \mathcal{O}(|V|)$
- usein kaarten ja solmujen lukumäärän suhdetta ei tiedetä ja sen takia aikavaativuus ilmaistaan molemmista riippuvalla ilmaisulla $\mathcal{O}(|V| + |E|)$
- lähes kaikkien kurssin verkkoalgoritmien aikavaativuusanalyysi etenee samaan tyyliin, eli rekursion tai loopin analysoinnissa joudutaan huomioimaan kaikkien kaarten läpikäynti

Lyhimmät polut

- Dijkstran algoritmi
  - algoritmi tarkoitettu kaaripainollisille verkoille
  - laskee lyhimmän etäisyyden ja selvittää lyhimmän polun lähtösolmusta s jokaiseen muuhun solmuun
  - polun pituudella tarkoitetaan kaaripainojen summaa
  - ei toimi jos kaaripainot negatiivisia
  - toimii sekä suunnatuilla että suuntaamattomilla verkoilla
  - nodattaa ahnetta periaatetta: etenee aina lähimpään solmuun
  - toteutuksesta saadaan tehokas käyttämällä minimikekoa
  - aikavaativuusanalyyssä tärkeä muistaa, että keko-operaatioiden vaativuus on $\mathcal{O}(\log |V|)$ jos keossa pahimmillaan $|V|$ solmua

- Floyd-Warshallin algoritmi
  - algoritmi tarkoitettu kaaripainollisille verkoille
  - laskee lyhimmän etäisyyden ja selvittää lyhimmän polun kaikkien solmujen välillä
  - verkossa saa olla myös negatiivisia kaaripainoja
  - verkossa ei kuitenkaan saa olla negatiivisia syklejä
  - toimii sekä suunnatuilla että suuntaamattomilla verkoilla
  - toimintaperiaate eroaa radikaalisti muista verkkoalgoritmeista
  - kolme sisäkkäistä for-lausetta solmujoukon yli, eli aikavaativuus ilmiselvästi $\mathcal{O}(|V|^3)$
  - käyttää aputilaa $\mathcal{O}(|V|^2)$

Minimaaliset virittävät puut

- Mitä tarkoitetaan virittävällä puulla? Entä minimaalisella virittävällä puulla?
- Käsite minimaalinen virittävä puu on mielekäs vain yhtenäisille suuntaamattomille kaaripainollisille verkoille
- Primin algoritmi
  - aloitetaan jostain solmusta
  - lisätään virittävään puuhun solmu kerrallaan toimien niin, että aina lähimpänä puun ulkopuolella oleva solmu ja sen puuhun yhdistävä kaari lisätään seuraavaksi virittävään puuhun
  - aputietorakenteena minimikeko
- Kruskalin algoritmi
  - järjestetään kaaret painonmukaiseen järjestykseen, kevein ensin
  - otetaan kaaria mukaan järjestyksessä siten, että kaari hylätään jos se muodostaa syklin (eli päätepisteet eivät ole eri palassa)
  - aputietorakenteena union-find (joka ei kovin tärkeä kokeen kannalta)
- aikavaativuus molemmissa suunilleen sama, eli Prim $\mathcal{O}(|E| \log |V|)$ ja Kruskal $\mathcal{O}(|E| \log |E|)$

Verkkoalgoritmit ongelmanratkaisussa

- Monet verkkoalgoritmeista ovat aika kompakteja ja niiden toimintaperiaate on suhteellisen helppo ymmärtää
- aina ei ole kuitenkaan täysin selvää, että algoritmi toimii kuten halutaan ja tarvitaan matemaattista analyysiä algoritmien oikeellisuuden osoittamiseen
- toinen haastava asia on se, kuinka verkkoalgoritmeja sovelletaan ongelmanratkaisuun
- välillä soveltaminen on melko suoraviivaista (esim. laskareiden labyrintti- ja huoneiden laskemistehtävä)
- joskus ongelman muuttaminen verkko-ongelmaksi vaatii jonkin verran miettimistä (esim. vankilapako monisteen sivuilla 462-464, viikon 12 laskarien tehtävä 5)
- ja välillä vaaditaan epätriviaalia matematiikkaa (esim. viikon 12 laskarien tehtävä 4)