Informasjon

Du måtte klare 10 av 22 oppgaver for å få øvingen godkjent.

Finner du feil, mangler eller forbedringer, ta gjerne kontakt!

Oppgave 2

4

Kommentar:
Fra side 152 i læreboka (CLRS) har vi at en node med indeks i har foreldrenode med indeks $\lfloor i/2 \rfloor = \lfloor 9/2 \rfloor = 4$.

Oppgave 3

20 og 21

Kommentar:
Fra side 152 i læreboka (CLRS) har vi at en node med indeks i har barnenoder med indeks $2i$ og $2i + 1$ som blir 20 og 21.

Oppgave 4

```
def search_tree(root, dna):
    node = root
    for letter in dna:
        if letter not in node.children:
            return 0
        node = node.children[letter]
    return node.count
```

Oppgave 5

```
def build_tree(dna_sequences):
    root = Node()
    for sequence in dna_sequences:
        node = root
        for letter in sequence:
            if letter not in node.children:
                node.children[letter] = Node()
            node = node.children[letter]
```

```
            node.count += 1
        return root
```

Oppgave 6

Ingen binære hauger er binære søketrær.

Kommentar:
I binære søktrær har noder i venstre deltre lavere verdi, og noder i høyre deltre har større verdi (og det kan ikke være likt siden vi antar at elementene er unike). Dette er ikke forenelig med hauger, der alle barnenoder må være *enten* større *eller* mindre. Merk også at for en haug med 3 eller fler noder, så må rotnoden ha to barnenoder. Eneste riktig svar er derfor at ingen binære hauger er binære søketrær.

Oppgave 7

Nei, ikke generelt.

Kommentar:
Det er lett å konstruere eksempler der et av de $\lceil n/2 \rceil$ største elementene er en intern node, f.eks. $\langle 1, 2, 7, 3, 4, 8, 9 \rangle$ der 3 og 4 er løvnoder mens 7 ikke er det selv om tallet er større. Det er også lett å konstruere eksempel der alle de $\lceil n/2 \rceil$ største elementene er løvnoder, f.eks. $\langle 1, 2, 3 \rangle$.

Oppgave 8

Ja, alltid

Kommentar:
En sortert tabell er alltid en haug. Dette kommer av at barnenoder alltid ligger lengre til høyre i tabellen enn foreldrenoden, og er dermed alltid enten større eller mindre (avhengig av stigende eller synkende sortering). Er tabellen sortert i stigende rekkefølge har vi en min-haug. Er tabellen sortert i synkende rekkefølge har vi en maks-haug.

Oppgave 9

$\langle 5, 6, 9, 11, 8, 11 \rangle$

Kommentar:
For å gå fra den grafiske fremstillingen til den underliggende tabellen leser man den grafiske fremstillingen fra venstre til høyre, linje for linje.

Oppgave 10

Riktige alternativer:

- Algoritmen har tidskompleksitet $O(n)$ om alle elementene har samme verdi.
- Algoritmen er *in-place*.

Kommentar:
Algoritmen har tidskompleksitet $O(n)$ om alle elementene har samme verdi. Å bygge heapen tar allerede $O(n)$ tid, og deretter er det $\Theta(n)$ kall til MAX-HEAPIFY. Hvis HEAPSORT skal kjøre i $O(n)$ tid, må hvert av disse kallene gå i $O(1)$ tid. Fra pseudokoden til MAX-HEAPIFY, ser vi at dersom det elementet som står på indeks i i tabellen allerede er minst like stort som barna sine, vil man ikke ha noen rekursive kall, og metoden vil kjøre i $O(1)$ tid. Det er nettopp dette som skjer siden alle elementene har samme verdi. Dermed tar HEAPSORT $O(1)$ tid i dette tilfellet.

Algoritmen er *in-place*. Dette stemmer, siden HEAPSORT ikke lagrer elementer utenfor den opprinnelige tabellen vi gir til algoritmen.

Det er ikke en stabil sorteringsalgoritme. Prøv f.eks. å sortere $\langle(1, 2), (1, 3)\rangle$ på x-koordinat med HEAPSORT.

Oppgave 11

Riktige alternativer:

- INSERT: legger til et element i prioritetskøen.
- MINIMUM: returnerer det minste elementet i prioritetskøen.
- DECREASE-KEY: senker verdien til et element i prioritetskøen.
- EXTRACT-MIN: returnerer og sletter det minste elementet i priortetskøen.

Kommentar:
Se side 162 i læreboka.

Oppgave 12

Riktige alternativer:

- $\langle 6, 4, 9, 3, 4, 7, 2\rangle$
- $\langle 6, 9, 4, 3, 2, 7, 4\rangle$
- $\langle 6, 4, 3, 9, 2, 7, 4\rangle$

Kommentar:
Et viktig poeng er at foreldrenoder settes inn før sine etterkommernoder. Man kan selvfølgelig også kjøre TREE-INSERT på alle tabellene, og se hvilke som er riktige.

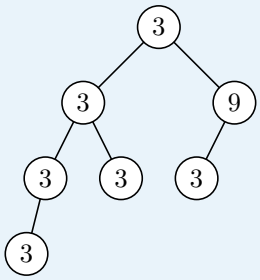
Oppgave 13

Minste er 27 og største er 57

Kommentar:
Her må vi følge det læreboka har som definisjon på et binært søketre. Bortsett fra at det skal være et binært tre, har læreboka følgende krav (side 287):

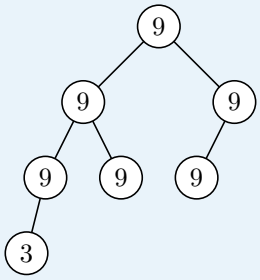
> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leqslant x.key$. If y is a node in the right subtree of x, then $y.key \geqslant x.key$.

Det minste treet som oppfyller dette er:



som har verdi $6 \cdot 3 + 9 = 27$.

Det største treet som oppfyller dette er:



som har verdi $6 \cdot 9 + 3 = 57$.

Oppgave 14

Både kjøretiden i beste tilfelle og den gjennomsnittlige kjøretiden er $\Theta(n \lg n)$, mens kjøretiden er $\Theta(n^2)$ i verste tilfelle.

Kommentar:
INORDER-TREE-WALK(T.*root*) tar $\Theta(n)$ tid uansett hvordan treet ser ut, så kjøretiden må

domineres av innsettingen i binærtreet. I verste tilfelle setter man inn tall i synkende eller stigende rekkefølge. I såfall vil høyden på treet bli $\Theta(n)$, siden binærtreet vil kun bli en lang kjede. Totaltiden blir dermed $\Theta(n^2)$ siden man må søke seg ned til bunnen av binærtreet for hver innsetting. Et binærtre har høyde $\Theta(\lg n)$ i forventning og TREE-INSERT tar $O(h)$ tid der h er høyden på treet, så innsetting av tallene vil ta $O(n \lg n)$ tid i forventning, og dermed vil det ta $O(n \lg n)$ tid også i beste tilfelle.

Kan dette ta $o(n \lg n)$ tid? For å se på det, så kan vi regne på hvor mange ganger algoritmen besøker en bestemt node i binærtreet. Det er ikke vanskelig å se at antallet ganger er lik antallet etterkommere noden har. Siden rotnoden har $n-1$ etterkommere, må den ha blitt besøkt $n-1$ ganger. Barnenodene til rotnoden har maksimalt $n-3$ etterkommere. Tilsvarende har nodene på dybde k maksimalt $n - 2^{k+1} + 1$ etterkommere. Vi vet også at minimumshøyden på treet er $\lg n$, så et minste antall nodebesøk b er gitt av:

$$b = \sum_{k=0}^{\lg n}(n - 2^{k+1} + 1) = \sum_{k=0}^{\lg n} n - \sum_{k=0}^{\lg(n)+1} 2^{k+1} + \sum_{k=0}^{\lg n} 1$$

$$= n(\lg n + 1) - 2^{\lg n + 2} - 2 + \lg n + 1 = n \lg n + n - 4n + \lg n - 1 = \Theta(n \lg n)$$

Så algoritmen gjør minst $\Theta(n \lg n)$ nodebesøk, noe som betyr at kjøretid i beste tilfelle og gjennomsnittlig kjøretid må være $\Theta(n \lg n)$.

Oppgave 15

Sorteringsalgoritmen er stabil dersom du setter inn tallene i søketreet i samme rekkefølge som de var i den opprinnelige tabellen.

Kommentar:
Sorteringsalgoritmen er stabil dersom du setter inn tallene i søketreet i samme rekkefølge som de var i den opprinnelige tabellen. Dette kan man se ved å se på TREE-INSERT. Denne legger alltid en node med samme verdi som en tidligere node inn i det høyre deltreet til noden. Den vil da bli skrevet ut senere når vi kjører INORDER-TREE-WALK.

Sorteringsalgoritmen er ikke asymptotisk optimal i verste tilfelle for sammenligningsbasert sortering. Hvis den skulle vært det, måtte den vært $\Theta(n \lg n)$ i verste tilfelle, men den er $\Theta(n^2)$ (se oppgaven over).

Hvis alle elementene i tabellen du skal sortere er like, vil ikke algoritmen ta $O(n)$ tid. Dette ville stride i mot at algoritmen var $\Theta(n \lg n)$ i beste tilfelle, som vist i oppgaven over.

Oppgave 16

Hvis man klarer å bygge et binært søketre i lineær tid uansett input, kan man så kjøre INORDER-TREE-WALK som bruker lineær tid, og dermed få ut elementene i sortert rekkefølge. Dette strider i mot den nedre grensen for sammenligningsbasert sortering på $\Theta(n \lg n)$ i verste tilfelle.

Oppgave 17

Her kan det være nyttig å bemerke seg at antallet stier til en node er lik summen av antallet stier til foreldrenodene sine. Vi kan dermed starte å si at det finnes én sti til rotnoden, og deretter gå nedover og for hvert nivå finne antallet stier til nodene der, ved å summere opp antallet stier til foreldrene. Her er det viktig at man alltid lagrer antall stier til en node når man har funnet det ut, for å unngå å måtte gjøre mer arbeid enn nødvendig. Det å lagre delsvar underveis, og bruke dem til å regne ut nye svar er en type dynamisk programmering, som er pensum i forelesning 6, og denne treliknende strukturen er en type rettet asyklisk graf, som er pensum i forelesning 10.

Oppgave 18

```
def adapted_search(root, dna, index):
    counter = 0
    node = root
    while True:
        counter += node.count
        if index == len(dna) or dna[index] not in node.children:
            return counter
        node = node.children[dna[index]]
        index += 1


def string_match(dna, segments):
    root = build_tree(segments)
    counter = 0
    for i in range(len(dna)):
        counter += adapted_search(root, dna, i)
    return counter
```

Kommentar:
Lager et søketre som i tidligere oppgave. Gå deretter gjennom DNA-sekvensen og søker fra alle mulige startpunkter. Her er det viktig at når man søker i treet at man husker å legge sammen alle matchene man finner når man går nedover i treet.

Oppgave 19

$\Omega(\lg n)$ og $O(n)$ er riktige.

Kommentar:
Den største høyden et binærtre kan ha er dersom alle nodene henger i en kjede etter hverandre. Da får man en høyde på $\Theta(n)$. Dersom man har et komplett binærtre, derimot, er høyden $\Theta(\lg n)$, tilsvarende høyden på en haug. $\Omega(\lg n)$ og $O(n)$ var de eneste alternativene som tillot begge disse tilfellene.

Oppgave 20

$\Theta(n \lg n)$, $O(\lg(n!))$, $\Omega(\lg(n!))$ og $\Omega(\lg n)$

Kommentar:
Vi vet at en haug med n elementer har en høyde på $\Theta(\lg n)$. En haug med høyde $n!$ har dermed en høyde på $\Theta(\lg(n!))$. Fra beviset for nedre grense for sammenligningsbasert sortering vet vi at $\Theta(n \lg n) = \Theta(\lg(n!))$.

Oppgave 21

Riktige alternativer:

- Hvis antall interne noder med to barn er k og antall løvnoder er ℓ, så er $k + 1 = \ell$.
- Hvis antall løvnoder er $O(1)$ er høyden på treet $\Theta(n)$.
- Et komplett binærtre har $\lfloor n/2 \rfloor$ interne noder.
- Et komplett binærtre har høyde $\Theta(\lg n)$.

Kommentar:
Hvis antall interne noder med to barn er k og antall løvnoder er ℓ, så er $k + 1 = \ell$. Dette kan vises med induksjon:
Grunntilfelle: Vi har kun rotnoden. Den er en løvnode, og vi har ingen interne noder med to barn, så vi får $k = 0$ og $\ell = 1$, så $0 + 1 = 1$, som stemmer.
Induksjonssteg: Vi har et binærtre med n noder der $k + 1 = \ell$. Vi setter på en ny node. Den kan enten settes på en løvnode, eller på en internnode med ett barn. Hvis vi setter den på en løvnode, vil den noden vi setter den på slutte å være en løvnode, mens den nye noden vil bli en løvnode. Siden antallet noder med to barn ikke endrer seg vil k og ℓ fortsatt være det samme, så likheten holder fortsatt. Dersom vi setter den nye noden på en intern node med ett barn, vil den bli en intern node med to barn, så k øker med en, mens den nye noden blir en ny løvnode, så ℓ øker med en, så likheten holder fortsatt.

Hvis antall løvnoder er $O(1)$ er høyden på treet $\Theta(n)$. Siden antall løvnoder er $O(1)$ følger det av påstanden over at antall noder med to barn er $O(1)$, som igjen betyr at antall noder med ett barn er $n - O(1) - O(1) = \Theta(n)$. Disse vil bli delt opp i kjeder etter hverandre (med potensielt bare en slik node). Slike kjeder må jo enten starte ved rotnoden, eller ved en node med to barn. Dermed er det bare $\Theta(1)$ steder en slik kjede kan starte, noe som betyr at det kun kan være $O(1)$ slike kjeder. Det betyr at de må i gjennomsnitt ha en lengde på $\Theta(n)$, som igjen betyr at høyden på treet må være $\Omega(n)$. Siden det er n noder i treet, må høyden være $O(n)$, og dermed kan vi konkludere med at høyden er $\Theta(n)$.

Et komplett binærtre har $\lfloor n/2 \rfloor$ interne noder. I et komplett binærtre har alle interne noder to barn. Hvis vi kombinerer det med den første påstanden her får vi at $k+1 = l$, $k+\ell = n$ og antallet

interne noder er k. Ved å sette sammen ligningene får vi

$$k + (k+1) = n \implies 2k + 1 = n \implies k = \frac{n-1}{2}$$

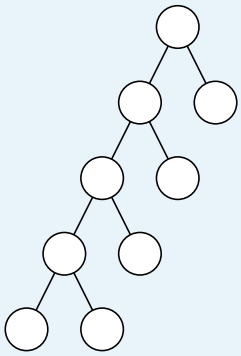
Siden k er et heltall må da $k = \lfloor n/2 \rfloor$.

Et komplett binærtre har ikke $\lfloor n/2 \rfloor$ løvnoder. Fra påstanden over er $\ell = k+1 = \lfloor n/2 \rfloor + 1 \neq \lfloor n/2 \rfloor$.

Et komplett binærtre har høyde $\Theta(\lg n)$. En haug med $2^k - 1$ elementer vil være et komplett binærtre, og hauger har høyde $\Theta(\lg n)$.

Hvis antall interne noder med kun én barnenode er $\Theta(n)$ er ikke høyden på treet nødvendigvis $\Theta(n)$. Fra over har et komplett binærtre har høyde $\Theta(\lg n)$ og antall kanter er $\Theta(n)$ siden alle noder bortsett fra rotnoden har en kant til foreldrenoden sin. Hvis vi endrer dette slik at man bytter ut alle kantene i dette treet, med en kant inn i en node og ut av noden igjen, vil disse nye nodene kun ha én barnenode, og det vil være $\Theta(n)$ av disse nodene. Videre vil høyden ikke mer enn dobles, så høyden er fortsatt $\Theta(\lg n)$. Merk at forskjellen fra påstand nummer 2 er at i påstand nummer 2, så begrenset vi oss også til at det kun var $O(1)$ noder som *ikke* var noder som hadde akkurat én barnenode.

Hvis det er flere løvnoder enn det er interne noder i et binærtre er høyden på treet ikke nødvendigvis $\Theta(\lg n)$. Et moteksempel er trær der noder med to barn henger i kjede nedover, der alle er knyttet neste node i kjeden, samt en løvnode, bortsett fra siste node i kjeden som er knyttet to løvnoder. Et slikt tre vil ha høyde $\Theta(n)$.
En illustrasjon:



Oppgave 22

$\Theta(\lg n)$ er riktig

Kommentar:
Her må vi prøve å begrense høyden på treet for å finne er asymptotisk uttrykk. Den minste mulige høyden på et slikt tre, er jo når treet er komplett. Et uttrykk for n, gitt av høyden h, blir da:

$$n = \sum_{i=0}^{h} 2048^i = \frac{2048^{h+1} - 1}{2048 - 1}$$

som gir

$$2048^{h+1} = 2047n + 1$$
$$h = \log_{2048}(2047n + 1) - 1 = \Theta(\lg n)$$

Oppgave 23

$\Theta(\lg \lg n)$

Kommentar:
For at treet skal bli minst mulig, bør elementene spre seg jevnest mulig over barnenodene i hvert lag av treet. For å løse denne er det nok enklest å sette opp en rekursiv funksjon for høyden, $\mathrm{T}(n)$, som er høyden til et tre med n noder. Siden rotnoden i dette treet vil ha $\lfloor \sqrt{n} \rfloor$ barnenoder, og hver av disse barnenodene får like mange etterfølgere, får hver av barnenodene $\frac{n-\sqrt{n}-1}{\sqrt{n}}$ etterfølgere. Hvis vi ser på hver av barnenodene som roten på sitt eget tre, vil dette treet dermed ha $\frac{n-\sqrt{n}-1}{\sqrt{n}} + 1 = \frac{n-1}{\sqrt{n}} = \sqrt{n} - \frac{1}{\sqrt{n}}$ noder. Rekursjonen blir dermed:

$$\mathrm{T}(n) = \mathrm{T}(\sqrt{n} - \frac{1}{\sqrt{n}}) + 1$$

$\frac{1}{\sqrt{n}} \ll 1$ for store n, så vi kan se bort i fra det leddet i argumentet. Spørsmålet blir hva høyden på rekursjonstreet gitt av $\mathrm{T}(n)$ er, og fra oppgave 19 i øving 3 vet vi at høyden her er $\Theta(\lg \lg n)$

Oppgave 24

Riktige alternativer:

- Algoritme 1, 3 og 4 har alle O(1) som kjøretid i beste tilfelle ved innsetting.
- Algoritme 1, 2, 3 og 5 støtter alle søk med $\mathrm{O}(\lg n)$ som gjennomsnittlig kjøretid.

Kommentar:
Med algoritme 5 vil hver *ikke* innsetting garantert ta $\Omega(n \lg n)$ tid i verste tilfelle dersom man bruker en sammenligningsbasert sorteringsalgoritme. Dette er siden tabellen allerede er sortert før du setter inn det nye elementet. Insertion-Sort vil bare flyttet det innsatte elementet forover i tabellen, og vil dermed ta $\Theta(n)$ tid i verste tilfelle.

Algoritme 1, 3 og 4 har alle O(1) som kjøretid i beste tilfelle ved innsetting. Ja, algoritme 1 har det siden det er alltid en mulighet at rotnoden kun har et barn, og at noden du setter inn ender som det andre barnet til rotnoden. Ja til algoritme 3, siden innsetting i hashtabeller har O(1) som kjøretid i beste tilfelle. Ja til algoritme 4, siden dynamiske tabeller kan legge til et element i

konstant tid i beste tilfelle.

Algoritme 1, 2, 3 og 5 støtter alle søk med $O(\lg n)$ som gjennomsnittlig kjøretid. Siden forventet høyde i et binærtre er $\Theta(\lg n)$ vil et søk i et binærtre som er generert med denne algoritmen ha gjennomsnittlig kjøretid $\Theta(\lg n)$. Samme med algoritme 2, men her er man garantert en høyde på $\Theta(\lg n)$. Hashtabeller har gjennomsnittlig oppslagstid på $O(1)$. Med algoritme 5, kan man binærsøke i tabellen i $\Theta(\lg n)$ tid.

Det er ikke 3 av algoritmene som støtter $o(n)$ som kjøretid i verste tilfelle ved søk. Med algoritme 1 kan binærtrærne bli $\Theta(n)$ høye i verste tilfelle, og da vil søking også ta $\Theta(n)$. Hashtabeller har en kjøretid på $\Theta(n)$ i verste tilfelle. Med algoritme 4 må du søke gjennom tabellen lineært som tar $\Theta(n)$ tid i verste tilfelle.

Oppgave 25

Riktige alternativer:

- Hvis $k = \Theta(n^3)$ er bruker algoritme 3 og 4 like lang tid (asymptotisk, i verste tilfelle) på en oppdatering.
- Algoritme 4 bruker like lang tid (asymptotisk, i verste tilfelle) på en oppdatering enten $k = n!$ eller $k = n^n$.

Kommentar:
Merge-Sort passer ikke bedre inn enn Insertion-Sort i denne situasjonen. Siden det bare et element som vil flyttes fremover i tabellen med Insertion-Sort, vil kjøretiden bli $\Theta(n)$, mens Merge-Sort har $\Theta(n \lg n)$ som beste kjøretid.

Hvis $k = \Theta(n^3)$ er bruker algoritme 3 og 4 like lang tid (asymptotisk, i verste tilfelle) på en oppdatering. Heap-Increase-Key i algoritme 3 bruker $\Theta(\lg n)$ tid i verste tilfelle, siden verste tilfelle er at du kjører Heap-Increase-Key på en node nederst i haugen, og noden skal fraktes til toppen av haugen. Max-Heap-Insert i algoritme 4 bruker $\Theta(\lg(nk))$ tid i verste tilfelle, siden høyden på haugen er $\Theta(\lg(nk))$ og også her må noden fraktes fra bunnen til toppen i verste tilfelle. Videre er $\Theta(\lg(nk)) = \Theta(\lg(n \cdot n^3) = \Theta(\lg(n^4)) = \Theta(4 \cdot \lg n) = \Theta(\lg n)$.

Algoritme 3 er ikke like rask som algoritme 4 på en oppdatering (asymptotisk, i verste tilfelle) uansett hva k er. Vi fant ut over at kjøretiden i verste tilfelle var $\Theta(\lg n)$ for algoritme 3 og $\Theta(\lg(nk))$ for algoritme 4. Hvis $k = n^n$ er verste tilfelle for algoritme 4 $\Theta(\lg(n \cdot n^n)) = \Theta(\lg(n^{n+1}) = \Theta((n+1)\lg n) = \Theta(n \lg n)$ og dette er ikke det samme som $\Theta(\lg n)$.

Algoritme 4 bruker like lang tid (asymptotisk, i verste tilfelle) på en oppdatering enten $k = n!$ eller $k = n^n$. Dette er riktig, siden $\Theta(\lg(nk)) = \Theta(n \lg n) = \Theta(\lg(n \cdot n!)) = \Theta(\lg(n \cdot n^n))$.

Ikke alle algoritmene bruker $O(n)$ minne uansett hva k er. Dette er fordi algoritme 4 vil bruke $\Theta(nk)$ minne, så om for eksempel $k = \omega(n)$ vil minnebruken til algoritmen være $\omega(n)$.

Ikke algoritmene bruker $O(n)$ tid på en oppdatering hvis $k = O(n)$. Dette er fordi algoritme 1 bruker MERGE-SORT som er $\Theta(n \lg n)$.

Det er ikke tilfelle at tre av de fire algoritmene bruker $O(1)$ tid på en oppdatering i beste tilfelle. Det er fordi MERGE-SORT og INSERTION-SORT er begge $\Omega(n)$ i beste tilfelle. (Man kunne selvfølgelig brukt en modifisert INSERTION-SORT som kun flytter på elementet vi oppdaterte. I så fall ville vi fått $O(1)$ tid i beste tilfelle, men etter oppgaven bruker vi INSERTION-SORT slik den er definert i læreboka.)

Kjøretiden til en oppdatering med algoritme 4 er ikke $\Theta(\lg(n^k))$ i verste tilfelle. Vi har allerede vist at kjøretiden i verste tilfelle er $\Theta(\lg(nk))$ som ikke er lik $\Theta(\lg(n^k))$ generelt.

Oppgave 27

For eksamensoppgavene, se løsningsforslaget til den gitte eksamenen.

For oppgaver i CLRS, så finnes det mange ressurser på nettet som har fullstendige eller nesten fullstendige løsningsforslag på alle oppgavene i boken. Det er ikke garantert at disse er 100% korrekte, men de kan gi en god indikasjon på om du har fått til oppgaven. Det er selvfølgelig også mulig å spørre studassene om hjelp med disse oppgavene.

Et eksempel på et ganske greit løsningsforslag på CLRS, laget av en tidligere doktorgradsstudent ved Rutgers, finnes her.