Løsningsforslag eksamen DAT200 kont vinter 2020

Oppgave 1

Svar: O(n)

Oppgave 2

Svar: $O(n^2)$

Oppgave 3

Svar: O(n*k) hvor n er antall elementer i vekt_liste og verdi_liste (de to skal alltid være like lange) og k er kapasiteten til ryggsekken.

Oppgave 4

Løsningsforslag i Python-lik pseudokode:

```
def konverter_til_arrayliste(lenket_liste):
        array_liste = ArrayList()
        for element in lenket_liste:
                array_liste.append(element)
        return array_liste
```

Kjøretid O(n). Å hente ut neste element fra en iterator i en lenket liste er O(1), totalt O(n) siden dette gjøres n ganger. Append i en array-liste kjører på O(1) amortized tid, og gjøres n ganger, totalt O(n) tid. Dette er de to dominerende operasjonene. Total kjøretid blir derfor O(n)

Oppgave 5

Teknikken splitt og hersk går ut på å dele opp et stort problem i mindre delproblemer, løse hvert delproblem for seg, og så kombinere løsningen på delproblemene til en løsning på hele problemet. Teknikken krever at problemet kan splittes opp i uavhengige delproblem. Teknikken brukes i de tilfellene hvor det å splitte et problem i delproblemer og det å kombinere løsninger på delproblemer til en løsning på hele problemet er raskere enn å løse problemet.

Et eksempel er flettesortering / MergeSort algoritmen for sortering. Å sortere med de enkle algoritmene er $O(n^2)$. Å splitte ei liste i to kan gjøres i O(1) tid. Å kombinere to sorterte lister til en sortert liste kan gjøres gjennom å iterere i parallell gjennom de to listene i O(n) tid.

```
Def merge_sort(liste):
        If len(liste) < 10:
                Return Insertion_sort(liste)
        Else:
```

Start = merge_sort(første halvdel av lista)

Slutt = merge_sort(andre halvdel av lista)

Return merge(start, slutt)

Def merge(liste1, liste2):

Nv_1 = første element i liste1

Nv_2 = første element i liste2

Resultat = tom liste

While det fortsatt er elementer igjen:

If nv_1 <= nv_2:

Resultat.append(nv_1)

Nv_1 = neste element i liste1

Else:

Resultat.append(nv_2)

Nv_2 = neste element i liste2

Return resultat

Av de sorteringsalgoritmene som er listet opp er MergeSort og QuickSort splitt-og-hersk, de andre er det ikke.

Oppgave 6

Python kode:

```
def palindrome(streng):
    if len(streng) <= 1:
        return True
    if streng[0] == streng[-1]:
        return palindrome(streng[1:len(streng)-1])
    else:
        return False
```

Merk at oppgaven eksplisitt ber om en rekursiv funksjon, så iterative funksjoner får lav score selv om de virker.

Oppgave 7

Plass	Innhold
0	9
1	18
2	Tom
3	Tom

4	27
5	Tom
6	15
7	7
8	17

Oppgave 8

Fyllingsgrad: Hvor stor andel av cellene i tabellen som har en verdi. Hashtabellen fra oppgave 7 har 9 elementer hvorav 6 har en verdi, så fyllingsgraden er 6/9 = 2/3 = 0.6667.

Hashfunksjon: En hashfunksjon er en funksjon som konverterer fra et stort verdiområde til et mindre. Typisk konverterer den fra store tall, strenger eller objekter til heltallsindekser i en tabell.

Kollisjon: To verdier kolliderer hvis de er ulike, men hashfunksjonen gir samme verdi for dem. Verdiene havner derfor i utgangspunktet i samme celle i hashtabellen.

Oppgave 9

Et tre kan defineres som enten en enkelt node (bladnode) eller ei rot og ett eller flere undertrær. Undertrærne er selv trær, og rota til hvert undertre er barn av rota til dette treet.

Et tre kan defineres som en rettet asyklisk graf der du har en node, rota, som ikke har noen kanter inn til seg, og hvor alle andre noder har en og bare en kant inn til seg.

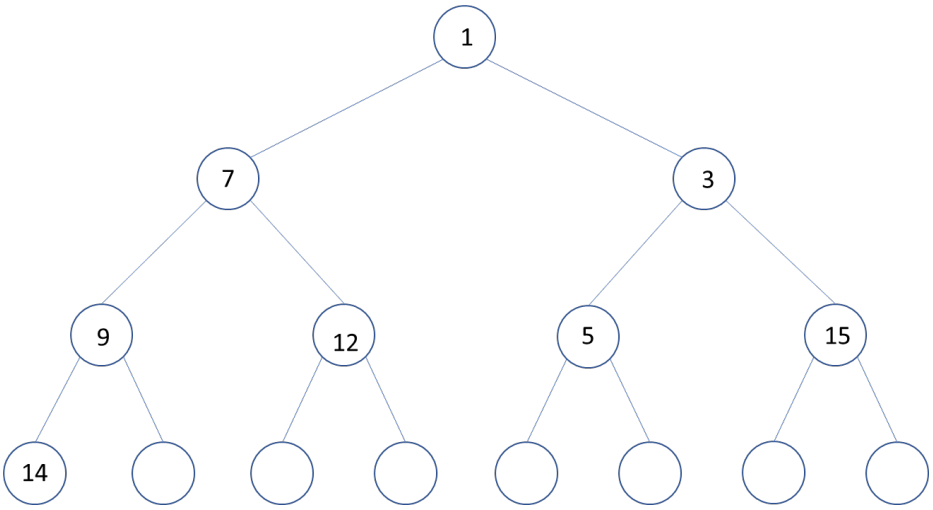
Et tre kan defineres som en mengde noder hvor hver node unntatt en (rota) har en og bare en forelder, og alle noder har null eller flere barn.

Bare en av disse definisjonene trengs for å svare på oppgaven.

Oppgave 10

- Bil: HashMap på registreringsnummer, gjør spørring 1 på O(1) tid.
- Kunde: TreeMap med etternavn+fornavn som nøkkel gjør spørring 2 på O(log(n)) tid. HashMap er ikke aktuelt her da oppgaven ber om en ulikhetsspørring (delvis navn).
- Verksted_reservasjon: TreeMap med tidspunkt som nøkkel gjør spørring 3 og 4 på O(log(n)) tid. Et HashMap gjør spørring 3 på O(1) tid, men støtter ikke spørring 4. Hvis man skal velge én datastruktur blir det TreeMap. Velger man to må man ta hensyn til at begge må holdes oppdatert til enhver tid.
- Historikk: Sortert liste, sortert på tid. I dette tilfellet vil det du setter inn alltid være nyere enn de tidligere innslagene, så du kan holde lista sortert ved å sette inn på slutten. Å sette inn på slutten tar O(1) amortized tid. For å finne siste verkstedbesøk henter du bare ut det siste elementet, som tar O(1) tid. Siden dette på mange måter vil oppføre seg som en stabel, har jeg gitt full score på «stabel» / «stack» som svar her.

Oppgave 11



Oppgave 12

Idé: Lag den transponerte grafen, og kjør deretter bredde først søk eller dybde først søk på den, hvor man starter i noen til emnet man er interessert i. Den transponerte grafen er en graf hvor alle kantene har motsatt retning. For en matrise-graf vil dette være den transponerte matrisen, derav navnet transponert graf. Siden en «noder med naboliste» graf er oppgitt som implementasjon i oppgaven, må man lage en egen algoritme for å transponere slike grafer.

Def transponer_graf(graf):

- Lag en ny graf G'. Sett inn alle nodene fra «graf» i G' men ikke kantene
- For hver node N i «emnegraf»
    - For hver nabo
        - Finn naboen i G' og sett inn N i nabolista
- Returner G'

Def finn_forutsetninger(emne, emnegraf)

- G' = transponert_graf(emnegraf)
- Lag ei tom liste med emner R
- Kjør bredde-først-søk eller dybde-først-søk på G' hvor man starter i noden «emne»
    - Sett nodene inn i R etter hvert som man støter på dem
- Returner R, som er svaret på spørsmålet

Kjøretid O(V + E). Både transponer_graf og resten av finn_forutsetninger, hvor kjøretida domioneres av BFS eller DFS, har denne kjøretida. O(E) vil gi nesten full score da E som regel dominerer over V her.

Merknad om graf-metoden get_forrige (Python) / getForrigePaaVeien (Java), som noen studenter har brukt for å finne forrige node: Denne henter ut egenskapen «forrige» fra en oppgitt node. Denne egenskapen ble i forelesningene hvor denne eksempelkoden ble brukt satt av grafalgoritmer som for eksempel Dijkstra's Algoritme for å notere veien som den finner. Man kan se fra den oppgitte koden at «forrige» egenskapen til nodene ikke settes når man bygger grafen (add_node og add_edge

metodene i Python, addNode og addEdge i Java), og derfor godt kan være None (Python)/udefinert (Java) for alle nodene i det man starter med å skulle finne forutsetningene. Man kan derfor ikke bruke get_forrige/getForrigePaaVeien i denne oppgaven. Man kan også se at «forrige» egenskapen er en enkeltverdi mens det er flere noder i eksempel-grafen som har flere kanter som går inn til noden, eksempel DAT230, DAT320 og DAT310.

Oppgave 13

Dijkstra's Algoritme:

- Sett kostnaden til startnoden til 0, og kostnaden til alle andre noder til uendelig.
- Legg startnoden i en prioritetskø med prioritet 0. Denne køen bruker kostnaden som prioritet og henter alltid ut noden med lavest kostnad.
- For hver iterasjon, så lenge det er noder i prioritetskøen
    - Ta ut en node fra prioritetskøen
    - Sjekk alle naboene til noden.
        - Hvis gammel kostnad > kostnad for denne noden + kostnad til kanten fra denne noden, endre kostnad for naboen. Hvis den allerede ligger i prioritetskøen, bruk decrease_key. Ellers sett den inn i prioritetskøen med kostnaden som prioritet.
        - Hvis du ønsker å finne veien og ikke bare kostnaden, sett «forrige» egenskapen til naboen til å referere til denne noden hvis du oppdaterte kostnaden i forrige punkt
- Hvis du ønsker veien, start i målnoden og bygg opp veien som ei liste. Sett inn målnoden, følg «forrige» referansene og sett inn nodene i lista helt til du kommer tilbake til start. Da har du veien.
- Kjøretid avhenger av hva som brukes som prioritetskø.
    - Basis kjøretid O(V*<tid for å ta ut en node i prioritetskøa> + V*<tid for å legge til en node i prioritetskøa> + E*<tida det tar å senke en nøkkel i prioritetskøa>)
    - ArrayList prioritetskø: Da er å ta ut en node O(V) mens å senke prioriteten er O(1). Da blir kjøretida til algoritmen $O(V^2 + E)$
    - Binærhaug: Da er å ta ut en node O(log(V)) og å senke prioriteten er O(log(V)). Da blir kjøretida O(V*log(V) + E*log(V))
    - Mer avanserte hauger kan brukes som reduserer kjøretida til O(V*log(V) + E). Eksempler på slike hauger er pairing heap og fibonacci heap, som ikke er pensum i dette faget