# Løsningsforslag DAT200 eksamen høst 2018

## Oppgave 1

a) 2*n
b) 1,5^n

## Oppgave 2 – 8%

a) O-notasjon brukes til å angi hvor fort en funksjon vokser med verdien den får inn. Man sier at en funksjon f(x) er O(g(x)) hvis for alle x over en viss verdi og en gitt konstant c så er c*g(x) >= f(x). Man bruker ofte O-notasjon for å angi kjøretidene til algoritmer. For kjøretida til algoritmer så er verdien n lik størrelsen på datamengden algoritmen skal operere på. O-notasjon brukes siden den konkrete kjøretida avhenger av maskinvare, andre programmer, og andre ting som har lite med hvor god algoritmen er. Derimot hvor fort kjøretida vokser med datamengden er stort sett bare avhengig av hvor god algoritmen er. Derfor er O-notasjon god til å sammenlikne ulike algoritmer for samme problem.
b) Der O-notasjon gir en øvre grense for hvor fort funksjonen vokser, så gir omega-notasjon en nedre grense. Så f(x) er Ω(g(x)) hvis for alle x over en viss verdi og en gitt konstant c så er c*g(x) <= f(x). Theta-notasjon angir både øvre og nedre grense. Så f(x) er Θ(g(x)) hvis for alle x over en viss verdi og for to gitte konstanter c og d så er c*g(x) <= f(x) <= d*g(x).

## Oppgave 3

O(n)

## Oppgave 4

Rekurrens: T(n) = 2*T(n/2) + 1, T(1) = 1

Denne kan løses på flere måter.

Master metode: a=2, b=2, k=0. $b^k = 1$. Får første tilfelle, O(n^logb(a)) = O(n^log2(2)) = O(n)

Gjentatt substitusjon

T(n/2) = 2*T(n/4) + 1

Setter inn i første: T(n) = 2*(2*T(n/4) + 1) + 1 = 4*T(n/4) + 3

T(n/4) = 2*T(n/8) + 1

Setter inn i forrige: T(n) = 4*(2*T(n/8) + 1) + 3 = 8*T(n/8) + 7

Finner mønster: $T(n) = 2^k*T(n/2^k) + 2^k-1$

Får T(1) ved $n = 2^k$. Kan substituere n for $2^k$ i likningen

T(n) = n*T(1) + n-1 = n + n – 1. Dominant ledd n

Svar: O(n)

## Oppgave 5 – 8%

```
public static File leitEtterFil(File fila, String filnavn) {
	if (fila.isDirectory()) {
		File[] filene = fila.listFiles();
		for (File f: filene) {
			File sjekk = leitEtterFil(f, filnavn);
			if (sjekk != null) return sjekk;
		}
	} else {
		if (fila.getName().equals(filnavn)) return fila;
	}
	return null;
}
```

Kjøretid i O-notasjon: O(n) hvor n er antall filer som ligger i katalogene under f, siden denne må gå gjennom alle filene for å sjekke navnene opp mot det oppgitte navnet.

Prosent telling: O-notasjon teller 1%, algoritmen teller 7%

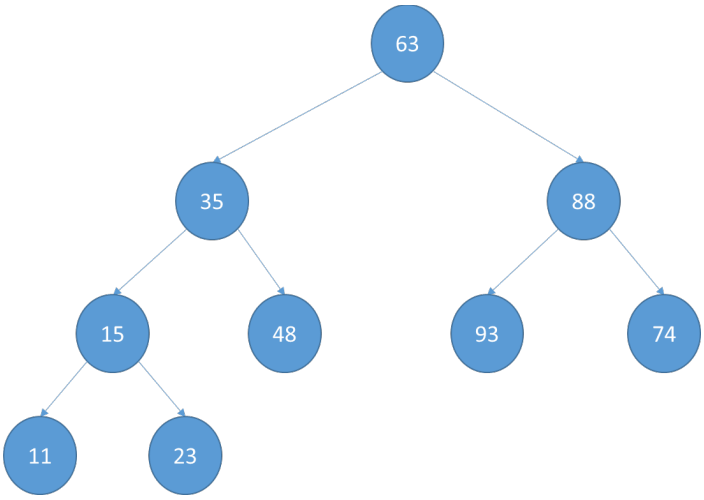
## Oppgave 6

Binærtre. Det hadde vært et AVL tre om nodene 93 og 74 kom i motsatt rekkefølge.

Scores for svar som er nesten riktige:

- AVL tre og binært søketre får tre poeng
- DAG får to poeng
- Graf får ett poeng

Figur for ekvivalent tre:




## Oppgave 7

En lenket liste er en kjede med noder av følgende form:

Public class LinkedList <E> {

Private class Node {

E element;

Node neste;

```
    }


    Private Node forste; // Referanse til starten av lista

    Private Node siste; // referanse tilslutten av lista
}
```

Hver node inneholder et element og en referanse til neste node. Den lenkete lista inneholder en referanse til første og siste node.

Løsningsforslaget beskriver en enkeltlenket liste. Andre varianter som dobbeltlenket liste og sirkulær liste får også full score.

Operasjoner:

AddLast: Antar at du har referanse til siste element. Krever at du lager en ny node for elementet, setter neste-referansen til siste element til å referere til den nye node, og at du oppdaterer referansen til siste element i lista til å referere til den nye noden. Kjøretid O(1)

getFirst: Bruk referansen til første node for å finne første node. Hent ut verdien i den. Kjøretid O(1)

get(index): Bruk referansen til første node for å finne første node. Følg neste-referansene for å komme videre. Gå index antall skritt (følg en neste-referanse for hvert skritt). Hent ut verdien i noden du står i etterpå. Kjøretid O(index), som ofte regnes som O(n) siden index like gjerne kan være langt bak i lista.

Remove(index): Gjør get(index) men behold referansen til elementet før det man står i. Oppdater neste pekeren til elementet foran til å referere til elementet etter i stedet for slik å «hekte av» elementet som skal fjernes. O(n) av samme grunn som get(index).

## Oppgave 8

Insertion Sort, best case: O(n)

Insertion Sort, worst case $O(n^2)$

Merge Sort, best case O(n*log(n))

Merge Sort, Worst Case O(n*log(n))

Quicksort, best case O(n*log(n))

Quicksort, worst case $O(n^2)$

## Oppgave 9

En in-place sorteringsalgoritme sorterer ved å bytte om på elementer i den opprinnelige lista. Andre sorteringsalgoritmer sorterer inn i en eller flere nye lister. Fordelen med in-place sortering er at den ikke trenger å sette av plass til nye lister og derfor bruker mindre minne. In-place sorteringsalgoritmer er også ofte er raskere siden de ikke trenger å kopiere data. Ulempene er at de

er vanskeligere å parallellisere samt at du mister den opprinnelige lista. Det regnes som regel som en fordel å være in-place så ulempene teller lite her.

Insertion sort, Shell sort og quicksort er in-place. Mergesort og counting sort er ikke in-place.

## Oppgave 10

| | |
|---|---|
| **0** | 45 |
| **1** | 1 |
| **2** | |
| **3** | |
| **4** | 27 |
| **5** | 23 |
| **6** | 5 |
| **7** | 7 |
| **8** | 35 |

Forklaring:

45%9 = 0, settes inn på posisjon 0

23%9 = 5, settes inn på posisjon 5

35%9 = 8, settes inn på posisjon 8

7%9 = 7, settes inn på posisjon 7

1%9 = 1, settes inn på posisjon 1

5%9 = 5. Der ligger allerede tallet 23. Prøver $5 + 1^2 = 6$, som er ledig.

27%9 = 0. Der ligger tallet 45. Prøver $0 + 1^2 = 1$. Der ligger allerede tallet 1. Prøver deretter $0 + 2^2 = 4$. Den posisjonen er ledig.

## Oppgave 11

En god hashfunksjon skal unngå kollisjoner så langt det lar seg gjøre. En kollisjon er når ulike nøkler havner på samme sted i hashtabellen. Hvis flere elementer havner på samme sted i tabellen så går ytelsen til hashtabellen ned siden man må leite gjennom alle elementene som havner på samme sted. Å finne elementer i en hashtabell er O(1) best og average case, men O(n) worst case. Worst case skjer ved mange kollisjoner.

## Oppgave 12

1: HashMap<String, Kunde>: Brukernavn -> Kunde. Hashmap siden dette vil være et likhetssøk

2: TreeMap<Tidsfrist, Ordre>: Denne skal holdes sortert og gir O(n) på denne operasjonen.

> **Alternativ, noe redusert uttelling:** List<Ordre> som holdes sortert på tidsfrist. Denne har samme kjøretid som TreeMap på oppgave 2, O(n), men er betydelig tregere på innsetting, noe en nettbutikk også må håndtere. Innsetting O(n) mot O(log(n)) for TreeMap. Noe redusert score men gir uttelling.

For denne og punkt 4-5: Merk at ordre ikke nødvendigvis kommer inn i systemet i riktig rekkefølge for disse spørsmålene. En ordre kan komme i september og ha tidsfrist i desember, en annen kan komme 15. oktober og ha tidsfrist 20. oktober.

**Vanlig feil svar:** En minimumhaug gir O(n*log(n)) ved gjentatte kall til removeMin samt at du ødelegger haugen. Dette er ikke bedre enn usorterte datastrukturer som du sorterer når det trengs. Merk at selv om både denne oppgaven og oppgave 5 spør om tidsfrist så er det ulike spørsmål. Denne oppgaven spør etter alle ordre sortert etter tidsfrist. Oppgave 5 spør etter å finne og fjerne den med lavest tidsfrist. Ulike datastrukturer egner seg til å svare på de to spørsmålene. Søketrær holder dataene sortert, hauger gjør ikke det, de garanterer bare et rota er minste (evt. største) element.

3: HashMap<Kategori, Liste av varer>: Dette er et likhetssøk på kategori, derfor hashmap. Siden det kan være flere varer i en kategori må hver verdi i HashMap være ei liste.

**Alternativ, noe redusert uttelling:** Binært søketre som tolererer duplikater, med Kategori som nøkkel og Vare som verdi. Denne vil holde varene sortert på kategori men har høyere kjøretid enn HashMap versjonen.

4 og 5: Heap<Ordre, sortert på tidsfrist>: Heap er den beste strukturen her, etterfulgt av TreeMap. HashMap er uegnet siden den ikke kan gi ordre med lavest tidsfrist. Ei liste vil enten være treg på innsetting – spørsmål 4 (hvis den holdes sortert), eller være treg på uthenting – spørsmål 5 (hvis den ikke holdes sortert).

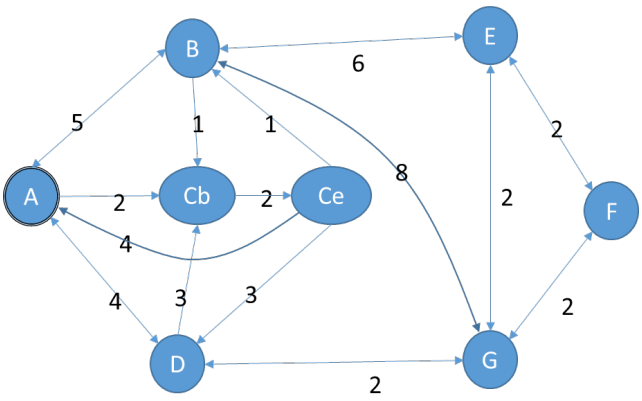
6: TreeMap<Navn, Vare>: Skal kunne søke på delvise navn, noe TreeMap støtter men HashMap ikke gjør. En ArrayList som holdes sortert vil gi noe uttelling men er dårligere enn TreeMap på dette. Binærsøk på en ArrayList har samme kjøretid som ulikhetssøket på et TreeMap, men er dårligere på andre operasjoner som for eksempel å legge til nye varer.

## Oppgave 13

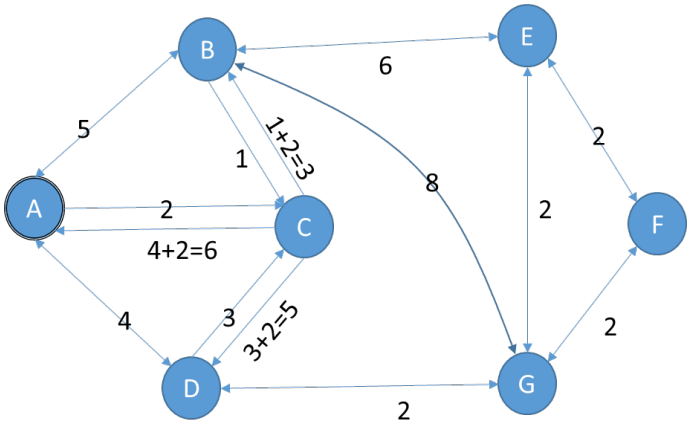
Rekkefølge: C, B, D, G, E, F

## Oppgave 14

**Løsningsforslag:** Splitt hver node med nodekostnad i to noder. I eksemplet splitt node C i nodene Cb og Ce. Alle kanter som kommer inn til C skal inn til Cb. Alle kanter som går ut fra C skal gå ut fra Ce. Lag en kant fra Cb til Ce med kostnaden i noden. Gjør tilsvarende med eventuelle andre noder med nodekostnad.



**Alternativ løsning, full score:** Legg kostnaden for noden til kostnaden for alle kantene som går ut av noden. Dette vil i eksemplet føre til at kanten C -> B får kostnad 3 og kanten C -> D får kostnad 5, mens kostnadene den andre veien for disse kantene blir uforandret. Se figur under. Merk at i alle grafrepresentasjonene som er gjennomgått i faget så er en toveis kant representert som to kanter, en hver vei.




## Oppgave 15

Løsning variant 1:

- Lag en klasse «Node» med innholdet «E element, int kostnad, int forrigeNode»
- Endre lista «elementer» fra E til Node
- I addNode, lag et nytt Node objekt og legg det til i lista med kostnad Integer.MAX_VALUE og forrigenode KortesteVeiGraf.HAR_IKKE_FORRIGE.
- getKostnad henter ut noden og henter ut kostnaden dens
- setKostnad henter ut noden og endrer kostnaden dens
- getForrigePaaVeien henter ut noden og henter ut forrige på veien
- setForrigePaaVeien henter ut noden og setter forrige på veien

Løsning variant 2, får lettere feil

- Lag to nye lister, «nodekostnader» og «forrigePaaVeienListe»
- I addNode, legg til nye elementer i «nodekostnader» og «forrigePaaVeienListe»: kostnad Integer.MAX_VALUE og forrigenode KortesteVeiGraf.HAR_IKKE_FORRIGE. Det er viktig at listene «nodekostnader» og «forrigePaaVeienListe» er like lange som «elementer» og settes inn i samme rekkefølge! Dette er en bug-kilde i denne løsningen
- getKostnad henter ut kostnaden på oppgitt indeks i «nodekostnader»
- setKostnad setter den oppgitte indeksen i «nodekostnader»
- getForrigePaaVeien henter ut verdien i oppgitt indeks i «forrigePaaVeienListe»
- setForrigePaaVeien endrer verdien i oppgitt indeks i «forrigePaaVeienListe»