UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF2220 — Algoritmer og datastrukturer
Eksamensdag:	14. desember 2012
Tid for eksamen:	14:30 – 18:30
Oppgavesettet er på 7 sider.	
Vedlegg:	Ingen
Tillatte hjelpemidler:	Alle trykte eller skrevne

Kontroller at oppgavesettet er komplett før
du begynner å besvare spørsmålene.

Innhold

Generelle **råd**:

- Skriv **leselig** (uleselige svar teller ikke . . .)
- Husk å **begrunne** svar.
- Skriv korte og presise kommentarer. Hvis du bruker kjente datastrukturer (list, set, map) trenger du ikke forklare hvordan de fungerer. Generelt: hvis du bruker abstrakte datatyper fra biblioteket kan du bruke disse uten å forklare hva de gjør.
- **Vekten** til en oppgave indikerer vanskelighetsgraden og tidsbruken du bør bruke på oppgaven, ut i fra våre estimat. Dette kan være greit å benytte for å disponere tiden best mulig, dvs. ikke bruk mye tid på en oppgave med liten prosentsats (gå videre).

Lykke til!
Dino Karabeg & Martin Steffen

(Fortsettes på side 2.)

Oppgave 1 Diverse oppgaver (vekt 12%)

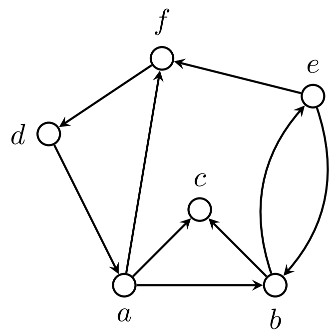
1a Tidskompleksitet (vekt 4%)

Hva er worst-case tidskompleksiteten til følgende programkode, avhengig av input-parameteren n?

```
int z = 0;
for (int i = n; i >= 1; i = i/2) {
  for (int j = 1; j <= n ; j++) {
      z = z + i + j ;
  }
}
```

1b SCCs i grafer (vekt 4%)

Anta gitt den rettete grafen G med noder $a, \ldots, f$ i Figur 1.



Figur 1: Rettet graf

1. Hvilke sterkt sammenhengende komponenter (SCCer, strongly-connected components på engelsk) har vi i G? Bare lag en liste.
2. Vis hvordan SCCs er bestemt algoritmisk. Gi trinnene i algoritmen. Ett trinn skal tilsvare å følge en kant i grafen mellom traversering, ikke mer detaljert enn det.
3. Anta nå en *u*rettet graf. Beskriv (ingen kode er nødvendig) hvordan man kan bestemme SCCs av urettede grafer på en måte som er enklere enn måten for rettet grafer?[1] Innebærer denne forenklingen også en forbedring med henblikk på worst-case tidskompleksitet? Forklar kort.

1c Boyer Moore (vekt 4%)

Beregn **good suffix shift** til nålen i form av en tabell:

ANANASBANAN

[1]Husk: *sterke* sammenhengskomponenter (SCCs) og bare *sammenhengskomponenter* faller sammen for urettete grafer.

(Fortsettes på side 3.)

Oppgave 2 Huffman-koding (vekt 12%)

2a Huffman tre (vekt 2%)

Gitt en fil med følgende frekvenstabell:

tegn	frekvens
`A`	2
`B`	6
`C`	10
`D`	7
`E`	1
`F`	1
`G`	3

Vis Huffman-treet for denne frekvenstabellen.

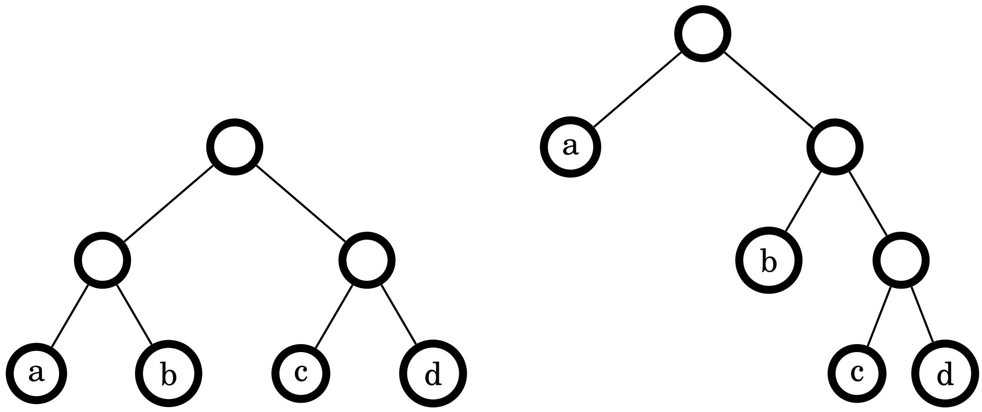
2b Huffman-koding (vekt 5%)

Gitt et alfabet (= reservoar av tegn) med størrelse 16 (la oss si `a`, `b`, ..., `p`), og en tekst med lengde 100 tegn. Anta en Huffman-koding hvor tegnet `a` er kodet med *én* bitt, for eksempel `0`. Hva kan du slutte om frekvensen av tegnet `a` i teksten?

2c Huffman-koding (vekt 5%)

Vanligvis kan en gitt tekst ha mer enn én Huffman-koding som er optimal som definert i forelesningen. Anta en tekst som bruker et alfabet med 4 tegn: `a`, `b`, `c`, og `d`.

> *Spørsmål:* Er det mulig at teksten har 2 forskjellige Huffman-kodinger som gitt i treet til høyre resp. til venstre?



Hvis ditt svar er *Ja*, vis dette med et eksempel, dvs. en frekvenstabell for de 4 tegnene. Hvis ditt svar er *Nei*, begrunn svaret.

(Fortsettes på side 4.)

Oppgave 3 Topologisk sortering (vekt 12%)

3a Sjekke kompatibel sortering (vekt 9%)

Anta en *rettet, asyklisk graf* ("directed, acyclic graph", DAG) $G = (V, E)$, og at alle nodene i V er i en gitt rekkefølge. Implementer en effektiv algoritme som *sjekker* at den gitte rekkefølgen er en gyldig *topologisk sortering* av G.

3b Kompleksitet (vekt 3%)

Gi tidskompleksiteten til løsningen din.

Oppgave 4 Beste vekslepenger (vekt 12%)

Implementer en algoritme som beregner *vekslepenger* på en optimal måte, dvs. på en måte som ikke bruker for mange mynter. Du kan anta at "valuta systemet" er gitt, dvs. utvalget av disponible mynter. Konkret er det følgende seks slags mynter (tenk "kronestykker") tilgjengelig:

$$1, 2, 5, 10, 20, 50.$$

Input og output av algoritmen er som følgende:

Input: en ikke-negativ integer som *pengesum*.

Output: en integer som gir det minste antall mynter nødvendig for å dekke pengesummen.

For å illustrere situasjonen: Anta pengesummen *72*. Følgende to kombinasjoner av mynter summerer til den korrekte summen 72, men kombinasjonen på den andre linjen bruker færre mynter:

$$\begin{array}{rcll} 72 & = & 1+1+10+10+10+10+10+10+10 & \Rightarrow \text{9 stykker} \\ 72 & = & 20+5+20+5+20+2 & \Rightarrow \text{6 stykker} \end{array}$$

Din algoritme må gi det *minimale* antall mynter for alle input. Utvalget av disponible *typer* mynter er <u>bestemt</u> som angitt ovenfor.

Oppgave 5 Binærtrær og binære søketrær (vekt 20%)

Disse oppgavene sjekker om et binærtre faktisk er et binært <u>søke</u>tre. Vi gjør det i en rekke trinn, som kan bli løst uavhengig av hverandre. For eksempel, problem **5b** kan du løse ved å *anta* at du har en løsning for problem **5a**, selv om du ikke (ennå) har implementert den, osv. Å følge den gitt rekkefølge kan likevel hjelpe.

Felles for alle deloppgavene: Gitt et *binær*tre med ikke-negative integer-verdier (nøkler/"keys") i nodene. Verdiene fyller ikke nødvendigvis kravet for binær <u>søke</u>trær.

(Fortsettes på side 5.)

5a Minimum & maximum nøkkel (vekt 5%)

Nå:

> Implementer en metode `min_key` som beregner *minimum* av alle nøkler i treet. Gjør det samme for å beregne *maksimum*; kall metoden `max_key`. Løsning skal ha en tidskompleksitet av $\mathcal{O}(n)$ (= lineær i antall noder).

Merk: Hvis du ønsker å bruke (eller din løsning må bruke) `min_key` på et *tomt* tre uten noder (og derfor uten nøkler), så kan du i dette tilfellet bruke `Integer.MAX_VALUE` (den største, disponible integer) som resultat. Tilsvarende kan du bruke 0 som resultat av `max_key` på et tomt tre (0 = den minste, ikke-negative integer).

5b Sjekking for søketre (vekt 5%)

Igjen anta gitt et binærtre som i den forrige deloppgave.

> Implementer en rekursiv algoritme `is_bst` som sjekker om treet er et binært *søke*tre. Bruke prosedyrene `max_key` og `min_key`,

som bestemmer maksimum, resp. minimum av alle nøkler i et tre.

5c Kompleksitet (vekt 5%)

Anta gitt et binærtre som er "fullstendig ubalansert" og dermed ligner en lineær liste, for eksempel: ingen node i treet har et barn til venstre. Med denne antagelsen

> hva er tidskompleksiteten av løsningen din (avhengig av antall noder i treet)? Forklar kort resultatet.

Du kan *anta* at prosedyrene `max_key` og `min_key` har kompleksitet $\mathcal{O}(n)$ avhengig av antall noder i treet.

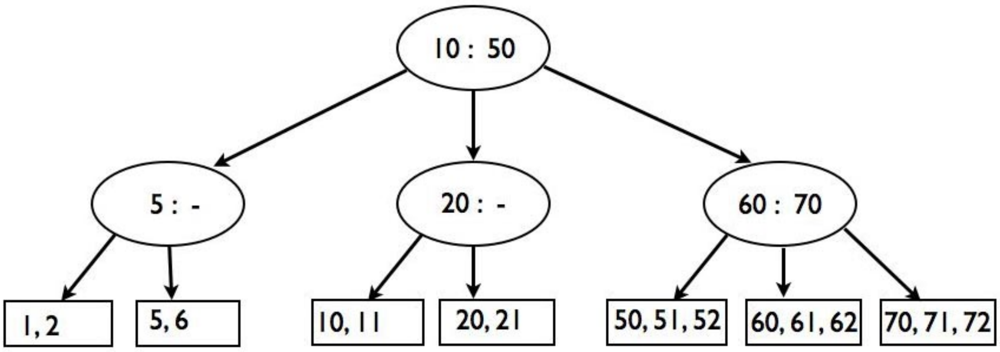
5d Forbedring av løpetid (vekt 5%)

Du skal forbedre implementasjonen med henblikk på tidskompleksitet, og *ikke bruke* `max_key` og `min_key`. Design en rekursiv prosedyre `is_bst_help` som tar 2 ekstra integer-verdier som argument, la oss si `low` og `high`. Igjen: `Integer.MAX_VALUE` er den største, disponible integer. Løsningen din skal ha $\mathcal{O}(n)$ som tidskompleksitet.

```
  boolean public is_bst () {

    return this.is_bst_help(Integer.MAX_VALUE,0);
  }

boolean public is_bst_help (int low, int high) {

      <   ... fill out ..... >
  }
```

(Fortsettes på side 6.)

Oppgave 6 B-Trær (vekt 12%)



Figur 2: B-tree

6a Innsetting (vekt 3%)

Anta at B-treet i Figur 2 er av *ordning* $M = L = 3$. Tegn B-treet etter innsetting av verdien 53.

6b Sletting (vekt 3%)

Gi B-treet etter å ha *slettet* 5 fra treet av Figur 2.

6c Kompleksitet (vekt 3%)

Anta nå et generelt B-tre av ordning M som lagrer n data elementer (for noen verdier M og n). Anta at hver indre node og hvert blad er lagret i en egen, separat "memory block". Estimer antall disk-tilganger ("disk accesses") som er nødvendig for å innsette en verdi i *worst-case*. Gi løsningen avhengig av M og n.

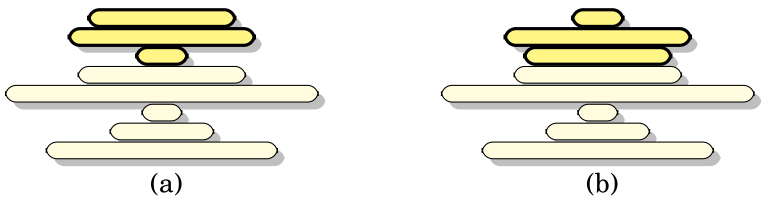
6d Amortisert kompleksitet (vekt 3%)

Vanligvis skjer ikke splitting av noder særlig ofte: etter at en node har vært splittet, vil det være flere innsettinger før den samme noden må bli splittet igjen.
Begrepet "amortisert kompleksitet" (engelsk *amortized cost*) tar hensyn til dette.
Den *amortiserte kompleksiteten* av innsettnings-operasjonen ble beregnet som følger: *del* antall disk-accesser som er nødvendig for en rekke `insert`-operasjoner på antall inserts ved å anta *worst-case* for sekvensen av insert-operasjoner.
Estimér amortisert kompleksitet av insert-operasjonen og avhengig av M og n.
Bruk de samme antagelser som i *deloppgave 6c*. Diskuter forskjellen mellom kompleksiteten av `insert` i deloppgavene 6c og 6d.

(Fortsettes på side 7.)

Oppgave 7 Sortering (vekt 20%)

Vi starter med å beskrive et abstrakt problem: "pannekake sortering". Gitt en haug med n pannekaker av forskjellige størrelse. Oppgaven er å sortere den slik at mindre pannekaker till slutt vil være på toppen av større pannekaker. Den eneste operasjonen for å forandre haugen er å stikke inn en "stekespade" under de øverste k pannekakene (for en verdi k) og "med ett kast, snu disse k pannekaker samtidig'.



Figur 3: Snu 3 pannekaker (fra venstre til høyre)

7a Pannekaker sortering (vekt 12%)

Implementer "pannekake sortering". Anta at dataene du må sortere er lagret i en array `p`; Index 0 refererer til den nederste pannekaken i "haugen" og toppen er representert på den høyeste indexen.
Anta at du har 3 metoder eller prosedyrer disponibelt:

`flip(k)`, `max(i,j)`, og `min(i,j)`.

Den første tilsvarer operasjonen beskrevet ovenfor. Med `max`-operasjonen kan du bestemme posisjonen av et maksimalt element i arrayen ("den største pannekaken") i elementene `p[i],...p[j-1]`. Operasjonen `min(i,j)` er tilsvarende for å lokalisere et minimalt element. Merk: `flip`, `max`, og `min` er de *eneste* operasjonene tillatt for å håndtere arrayen (men det er ikke obligatorisk å bruke alle tre). Merk at du *ikke* kan bruke `p[i]` til å lese fra eller skrive til enkelt-elementer i arrayen.

7b Kompleksitet (vekt 4%)

Anta at de nevnte operasjonene `flip`, `max`, og `min` har *konstant* tidskompleksitet, dvs., $\mathcal{O}(1)$ (for eksempel på grunn av spesialhardware som en "stekespade" og det menneskligе øyet for å bestemme størrelsen av pannekakene). Gi worst-case kompleksiteten av din løsning.

7c Kompleksitet (vekt 4%)

Til forskjell fra problem **7b**: Anta nå at operasjonene `flip`, `max`, og `min` er av *lineær* tidskompleksitet avhengig av input; for `flip`, lineær i antall pannekaker som ble snudd, for max/min, lineær i differansen mellom de to input parameterne. Hva er kompleksiteten av pannekakesorteringen nå?