UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	IN 115 — Algoritmer og datastrukturer
Eksamensdag:	19. mai 1999
Tid for eksamen:	9.00–15.00
Oppgavesettet er på 9 sider.	
Vedlegg:	Ingen
Tillatte hjelpemidler:	Alle trykte og skrevne

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

Merk:

Oppgavesettet består av 4 deler som kan løses uavhengig av hverandre. Oppgavene innen hver del er tenkt løst i den rekkefølge de står, men dette er selvsagt ikke noe krav til din besvarelse. Prosenten angitt på hver del antyder hvor mye vekt det vil bli lagt på denne delen under sensureringen.

Programmer skal skrives i Simula. Du behøver ikke gi noen fullstendig dokumentasjon av programmene, men du kan skrive noen få linjer som gir leseren nøkkelen til forståelse av programmet. Du kan anta at leseren kjenner problemstillingen i oppgaven meget godt.

Alle steder der det er spørsmål etter et program (eller en programbit) skal du skrive dette helt ut, og *ikke* bare henvise til liknende programmer, f.eks. i læreboka.

Les oppgavene nøye, og lykke til! Rune Djurhuus og Almira Karabeg

(Fortsettes på side 2.)

Oppgave 1 (10%)

1-a

Sorter sekvensen $9, 3, 1, 3, 4, 5, 1, 6, 4, 1, 2$ ved å bruke *Quick-sort* med median-av-tre partisjonering og "cutoff" lik 3. Du skal vise ved skisser hvordan sekvensen forandres under kjøringen av Quick-sort, men du skal ikke beskrive algoritmen, hverken i kode eller tekst. Du skal lage tilstrekkelig mange skisser (10 bør holde) til at mønsteret i algoritmen kommer tydelig fram.

1-b

Nå skal du velge ut det tredje minste elementet fra sekvensen ovenfor ved å bruke *Quick-select* med median-av-tre partisjonering og "cutoff" lik 3. Vis stadiene i kjøringen ved å lage skisser på samme måte som i punkt 1-a. Du skal heller ikke her beskrive algoritmen, hverken i kode eller tekst.

Oppgave 2 (15%)

IN105-studentene Per og Kari hadde tatt seg en pause i eksamenslesingen, og de sto utenfor Ifi og snakket om det fine mai-været:

Per: Det kom mer snø i Oslo i år enn på 50 år, men alt ble borte i løpet av noen uker.

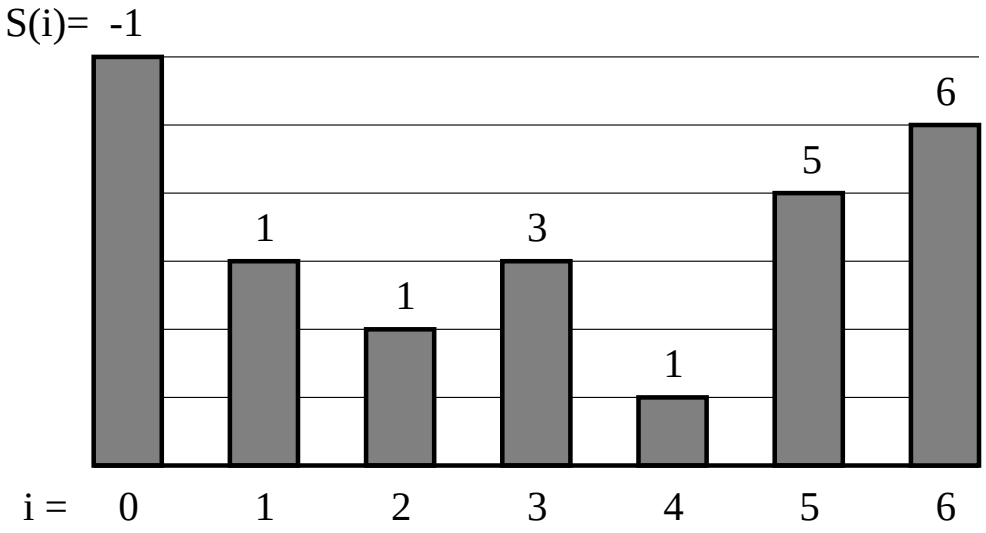
Kari: Hukommelsen din er ikke til å stole på. Du har rett i at det kom mye snø i vinter, men jeg husker godt vintrene på 80-tallet. Da kom det mye mer snø enn nå.

Per måtte til slutt motvillig innrømme at han ikke var sikker på når det sist kom mer snø enn i vinter, men han bestemte seg for å finne det ut en gang for alle. Han ringte derfor til Metereologisk Institutt, og de sendte han en datafil med statistikk over total snømengde på Blindern de siste N vintrene.

For hver vinter i ønsker Per å regne ut hvor mange år det er siden sist det kom mer snø enn i vinter i. Han har ikke tid til å lage et program som beregner det fordi han skal lese på IN105-eksamen, men han kjenner en dyktig IN115-student. Du sier selvsagt ja til å hjelpe Per!

Du skal anta at vintrene er nummerert fra 0 til $N-1$ og at snømengden for vintrene er lagret i en heltallstabell `Mengde(0:N-1)`. Antall år siden siste vinter med mer snø lagrer vi i tabellen `S(0:N-1)`. Vi definerer at $S(i) = -1$ dersom det ikke finnes noen tidligere vinter med mer snø enn i vinter i. I eksemplet i figur 1 er snømengdene framstilt i et histogram med `S`-verdiene angitt på toppen av hver søyle.

(Fortsettes på side 3.)



Figur 1:

2-a

Skriv en enkel, rett-fram prosedyre

```
PROCEDYRE BeregnS1;
```

som beregner `S`-verdiene for alle vintrer, gitt snømengdetabellen `Mengde(0:N-1)`. Angi tidsforbruket til algoritmen med store-O notasjon.

2-b

Vi ønsker nå å beregne `S`-verdiene mer effektivt. Vi observerer at `S(i)` lett kan beregnes hvis vi kjenner den nærmeste (tidligere) vinteren da det kom mer snø enn i vinter i. Vi bruker tabellen $H(0:N-1)$ til å ta vare på denne informasjonen. Hvis en slik vinter j finnes, setter vi $H(i) = j$, hvis ikke setter vi $H(i) = i + 1$. Vi får da ligningen $S(i) = i - H(i)$.

Skriv en ny prosedyre

```
PROCEDYRE BeregnS2;
```

som løser problemet i lineær tid ved å benytte seg av en stakk-datastruktur til å lagre passende `H`-verdier som den bruker i beregningen. `H`-tabellen er initsiert med 0-verdier når `BeregnS2` kalles.

Vis at din `BeregnS2`-prosedyre ikke bruker mer enn lineær tid.

Oppgave 3 35%

Mange trealgoritmer liker å besøke hver node to ganger i stedet for en gang, ved å bruke en kombinasjon av *preorder* og *inorder* som vi kan kalle *dobbeltorder*. Dobbeltorder-traversering av et tre er definert på følgende måte:

(Fortsettes på side 4.)

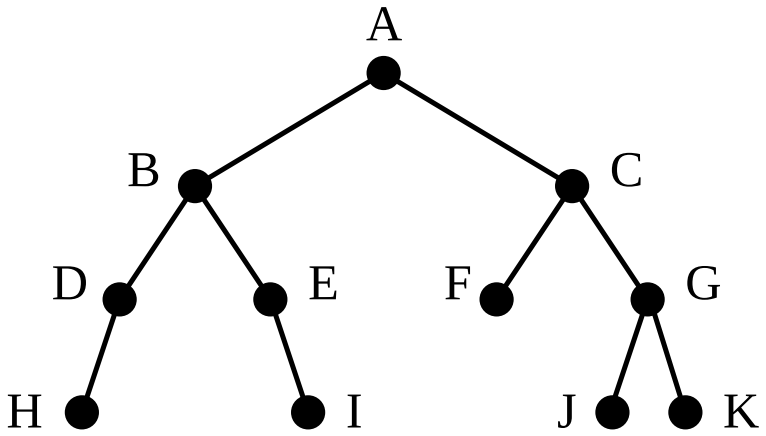
Hvis det binære treet er tomt, gjør ingenting. Ellers

1. *besøk roten for første gang*
2. *traverser venstre subtre på dobbeltorder vis*
3. *besøk roten for andre gang*
4. *traverser høyre subtre på dobbeltorder vis*

For eksempel vil starten av en dobbeltorder-traversing av treet i figur 2 være sekvensen:

$$A_1 B_1 D_1 H_1 H_2 D_2 B_2 E_1 E_2 I_1 I_2 \ldots$$

der A_1 betyr at node A er besøkt for første gang, E_2 betyr at node E er besøkt for andre gang, osv.



Figur 2:

3-a

Fullfør sekvensen som tilsvarer dobbeltorder-traverseringen av treet i figur 2. Hva kalles ordningen av nodene du får hvis du fjerner alle nodene med indeks 2 fra sekvensen? Hva kalles ordningen du får hvis du i stedet fjerner alle nodene med indeks 1?

3-b

Vi skal representere nodene i binærtreet vårt på vanlig måte med følgende Simula-klasse:

(Fortsettes på side 5.)

```
CLASS node;
BEGIN
  text navn;
  REF(node) vsub, hsub;
  BOOLEAN hsref;
END;
```

der `vsub` og `hsub` er pekere til henholdsvis venstre og høyre subtre (eventuelt NONE, dersom subtreet er tomt), og `hsref` er en hjelpevariabel som skal brukes senere i oppgaven.

Skriv en rekursiv prosedyre

```
PROCEDURE DobbeltOrder(T); REF(node) T;
```

som traverserer treet T på dobbeltorder vis, og skriver ut navnet til nodene (med et 1-tall eller et 2-tall etter seg) etter hvert som man besøker nodene. Utskriften skal altså tilsvare sekvensen du lagde i punkt 3-a.

3-c

Du skal i denne deloppgaven skrive en prosedyre som, gitt en peker P til en node i treet og en indeks d (som kan ha verdiene 1 eller 2), returnerer med P besøkt for d'te gang sin etterfølger i dobbeltorder-sekvensen. Prosedyren skal ha signatur

```
REF(Node) PROCEDURE Neste(P,d); REF(Node) P, INTEGER d;
```

Du kan anta at `P =/= NONE`. Vi definerer at `Neste(P,d)` skal returnere med NONE dersom node P besøkt for d'te gang er det siste steget i dobbeltorder-traverseringen.

Eksempel: Hvis man har treet i figur 2 og får node B og d=1 som innparametere, skal prosedyren returnere node D.

Forklar først kort hvorfor det er umulig å implementere `Neste`-prosedyren korrekt ved å bruke representasjonen av binætreet angitt i deloppgave 3-b. For å kunne skrive `Neste`-prosedyren, skal vi anta at det er gjort følgende modifikasjon av trestrukturen:

- Variabelen `hsref` har verdien TRUE for alle noder i treet som ikke har noe høyre subtre, og verdien FALSE for alle andre noder.
- For alle noder Q som ikke har noe høyre subtre, skal `Q.hsub` peke på den neste noden vi skal gå til etter å ha besøkt Q for andre gang i dobbeltorder-traverseringen.
- `hsub` skal ha verdien NONE for den siste noden i dobbeltorder-traverseringen.

Implementer nå `Neste`-prosedyren på den modifiserte trestrukturen.

(Fortsettes på side 6.)

3-d Kan puffes (Dårligste karakter er 4.0)

Implementer prosedyren

```
PROCEDURE SettDobbeltOrder(rot); REF(Node) rot;
```

som tar roten til et vanlig binætre (slik det er definert i oppgave 3-b) som innparameter og utfører modifikasjonene angitt i oppgave 3-c på treet.

Oppgave 4 (40%)

Sjakk er et spill som har fascinert programmererer helt siden datamaskinens barndom. De beste sjakkprogrammene er nå på nivå med de beste menneskene i verden, og i visse faser av spillet er de langt foran.

Et område hvor sjakkprogrammene har gjort store framskritt, er i stillinger med få brikker igjen på brettet. I dag finnes det dataprogrammer som spiller alle sjakkstillinger med 5 eller færre brikker helt perfekt. Det skjer ved at programmet slår opp stillingen i en tabell hvor det står om stillingen er tapt, vunnet eller uavgjort, og hvilket trekk som er best.

I denne oppgaven skal vi se på hvordan en slik tabell kan konstrueres. Vi skal se på en svært forenklet form for sjakk der det bare finnes 1 brikke på brettet, men prinsippene vi bruker, kan også anvendes på "ekte" sjakk.

Vi skal anta at vi har et sjakkbrett med $N \times N$ felter. Sjakkbrikken vi skal bruke er springeren (hesten). I et trekk kan springeren enten bevege seg to felter horisontalt og et felt vertikalt, eller et felt horisontalt og to felter vertikalt. Vi sier at springeren beveger seg som en 'L'. Figur 3 på neste side viser at en springer på midten av et 8 x 8 brett kan gå til 8 felter i et trekk. Hvis springeren står på kanten av sjakkbrettet, blir antall mulige trekk færre.

Problemet vi skal se på er følgende: *Gitt et felt på sjakkbrettet, hvor mange trekk må en springer bruke for å komme til dette feltet?*

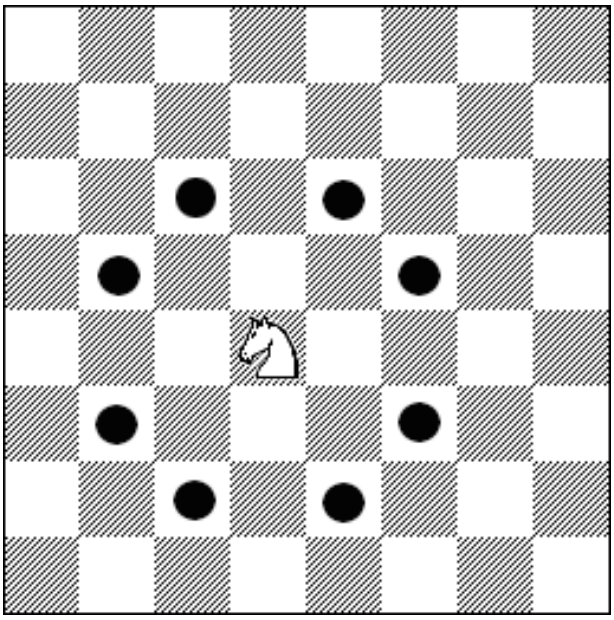
Et eksempel: En springer må bruke 3 trekk på å komme til feltet rett til høyre for seg, se figur 4 neste side.

Vi skal representere hver mulig stilling med en node i en graf. Vi skal bruke følgende Simuladeklarasjoner:

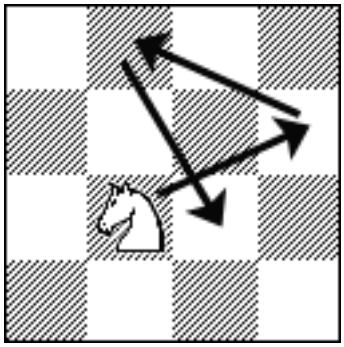
```
CLASS Sjakknode;
BEGIN
  REF(Felt) springer;
  INTEGER merke;
  REF(Sjakknode) ARRAY(1:8) utkanter;
END;

CLASS Felt(xPos,yPos); INTEGER xPos,yPos;
```

(Fortsettes på side 7.)



Figur 3:



Figur 4:

(Fortsettes på side 8.)

Siden det bare er 1 brikke på brettet, vil en stilling være entydig gitt ved posisjonen til springeren. Vi skal angi springerposisjonen med en peker `springer` som er av klassen `Felt`. Variablene `springer.xPos` og `springer.yPos` representerer indeksene til henholdsvis raden (vannrett) og kolonnen (loddrett) til feltet som springeren står i. Stillingen med springeren på feltet øverst til venstre på sjakkbrettet har `springer.xPos` lik 1 og `springer.yPos` lik 1.

Tabellen `utkanter` representerer utkantene fra noden. Den inneholder pekere til de stillinger hvert lovlige springertrekk kan føre til. Dersom springeren har færre enn 8 felter å gå til, vil den siste delen av tabellen inneholde NONE-pekere.

`Merke` er en hjelpevariabel som dere kan bruke ved behov.

4-a

Tegn grafen som representerer tilfellet $N = 3$, dvs. at vi har et sjakkbrett med 3×3 felter. La hver node ha navn (springer.xPos , springer.yPos).

4-b

```
INTEGER ARRAY avstand(1:N, 1:N);
REF (Felt) ARRAY besteTrekk (1:N, 1:N);
```

Gitt et felt (u,v) på sjakkbrettet, så skal den todimensjonale tabellen `avstand` inneholde hvor mange trekk en springer må bruke fra hvert felt på brettet til feltet (u,v). Den todimensjonale tabellen `besteTrekk` skal for hver stilling (springerplassering) angi feltet som springeren skal gå til for å komme raskest fram til feltet (u,v). (Hvis det er flere veier som er like raske, skal `besteTrekk` angi det første feltet på en av veiene.)

Du skal nå skrive prosedyren

```
PROCEDURE GenererTabeller(S); REF(Sjakknode) S;
```

Innparameteren `S` gir en peker til sjakknoden som representerer sluttstillingen, dvs. feltet (u,v) som springeren skal til. Når prosedyren har kjørt ferdig, skal tabellene `avstand` og `besteTrekk` være korrekt utfylt. Vi definerer at `avstand(S.springer.xPos,S.springer.yPos)` skal ha verdien 0 og `besteTrekk(S.springer.xPos, S.springer.yPos)` skal ha verdien NONE. Du skal anta at grafen der `S` er en av nodene, er generert ferdig på forhånd. Du skal også anta at `besteTrekk`-tabellen er initsiert med NONE-pekere, og at `avstand`-tabellen er initsiert med MAXINT-verdier, der MAXINT er det største tallet som får plass i en heltallsvariabel.

4-c

Gitt et sjakkbrett av størrelse $N \times N$. Hva er tidsforbruket til tabellgenereringsprosedyren du lagde i oppgave 4-b, angitt i store-O notasjon? Begrunn svaret kort.

(Fortsettes på side 9.)

4-d

På grunn av springerens veldefinerte bevegelsesmønster, behøver vi ikke representere grafen eksplisitt ved `Sjakknode`-objekter for å generere tabellene. Skriv en ny versjon av prosedyren

```
PROCEDURE GenererTabeller(U,V); INTEGER U,V;
```

som genererer tabellene ved bare å få indeksene til feltet springeren skal til, som innparameter.

4-e

For et virkelig sjakksluttspill er antall mulige stillinger mye større, slik at tabellene lett blir veldig store. Det er dermed viktig å spare plass. Forklar hvorfor `besteTrekk`-tabellen er overflødig, og vis at et program som skal bruke tabellene for å finne den raskeste veien fra et gitt felt til sluttfeltet, kun øker tidsforbruket sitt med en konstant faktor (uansett hvor stor N er) dersom programmet bare har tilgang til `avstand`-tabellen.