Obligatorisk oppgave 2 i INF 4130, høsten 2008

Innleveringsfrist er 24. oktober

Generelt for alle oppgavene

Samme reglement gjelder som for obligatorisk oppgave 1. Det kan komme presiseringer og forandringer i oppgaveteksten underveis, så følg med på beskjeder.

Husk at det ikke bare er programkode som skal leveres! Det er like viktig å svare skikkelig på drøftingsspørsmål som å levere god kode.

Vi legger opp til at programmene deres vil testes automatisk mot en rekke datasett når de mottas. Det er derfor viktig at output-formatet er identisk (tegn for tegn, også blanke) med formatet som er gitt i eksempelet på hver oppgave. Vi anbefaler derfor sterkt at en sjekker svaret programmet lager nøye med testdataene som følger med hver oppgave. Spør gjerne gruppelærer dersom en er usikker på dette.

Det er lagt ut en side med ytterligere leveringsanvisning for obliger. Den ligger under ”oppgaver” på kurssiden. Det blir typisk utvidet etter hvert, så følg med!

Det skal her i oblig 2 leveres to programmer (ett i oppgave 1 og ett i oppgave 2) og disse skal ha navn nøyaktig “Oppgave1” og “Oppgave2” (på den måten det blir mest naturlig innenfor språket en bruker, for eksempel skal klassen med main()-metoden hete “OppgaveX” i Java og skriptfila skal hete “OppgaveX.py” i Python).

Alle programmer skal ta input-fil som sin første parameter og output-fil som sin andre parameter. Under vil vi bare referere til input og output, da er det disse vi mener. Regler for programmeringsspråk er som på forrige oblig. De som fikk godkjent et annet språk for oblig 1 kan bruke dette på denne obligen også.

Se oppgave 1 i oblig 1 angående tips til I/O-håndtering i Java.

Oppgave 1 (Splay-trær)

Vi skal lage en enkel implementasjon av splay-trær. Den skal kun støtte en kombinert søke- og innsettingsoperasjon:

- Dersom noden eksisterer skal en følge standard algoritme for aksessering i splay-trær, som blant annet involverer at den aksesserte noden blir splayet til roten.
- Dersom noden ikke eksisterer skal den settes inn. En skal da følge standard algoritme for innsetting i splay-trær, som blant annet involverer at foreldernoden en skal sette inn under blir splayet, for så å sette inn den nye noden i roten.

En skal også rapportere hvorvidt noden eksisterte før innsetting eller ikke.

Implementasjonen skal bygge på å gå rekursivt ned i treet for å sette inn den nye noden. Splay-operasjonen som bringer en node opp til roten skal så gjøres mens rekursjonen trekker seg tilbake mot roten (altså, splay-operasjonen gjøres *bottom up*). Det som skal programmeres er altså en variant av det som diskuteres først i annet avsnitt av kap. 12.1. i Weiss (altså spesifikt skal det ikke gjøres slik koden i Weiss gjør det).

Verdiene i treet skal være heltall, og nodene i treet skal være standard søketre-noder: Hver node skal inneholde verdi og venstre og høyre barn, men *ikke foreldrepekere*. Om man ønsker kan man bruke en eksplisitt stakk som lagrer veien fra roten og ned til den aktuelle noden.

Dataformat

Input: En linje med heltall med mellomrom mellom som skal gis til innsettings-/ søkemetoden i den rekkefølgen de står.

Output: *For hver operasjon* skal programmet skrive ut en linje: Først enten "inserted" eller "found", så nodeverdien, og så alle nodene i treet i infiks rekkefølge med (dybde,verdi) for hver node. (Dette gjøres greiest med en rekursiv prosedyre).

Eksempel 1

Input:

```
23 93 54 93
```

Output:

```
inserted 23 (0,23)
inserted 93 (1,23) (0,93)
inserted 54 (1,23) (0,54) (1,93)
found 93 (2,23) (1,54) (0,93)
```

Eksempel 2

Et eksempel med utgangspunkt i figur 4.47 og utover i Weiss, side 135—137 i utgaven med copyright fra 1999, som det også kopieres fra (… er her forkorting av eksempelet). Vi setter først inn 1 til 32, og aksesserer så 1 og 2.

Input:

```
1 2 3 4 … 32 1 2
```

Output:

```
inserted 1 (0,1)
…
inserted 32 (31,1) (30,2) … (0, 32)
found 1 (0,1) (16,2) (17,3) … (1,32)
found 2 (1,1) (0,2) (9,3) (8,4) … (1,32)
```

Hele dette eksempelet ligger på fila `~inf3130/Oblig2/Opg1-test1-input.txt` og korrekt output på `~inf3130/Oblig2/Opg1-test1-svar.txt`. Det er i denne oppgaven bare ett gyldig svar for hver input.

Oppgave 2 (Flyt i nettverk)

Denne oppgaven går ut på å implementere FordFulkerson-algoritmen, med bruk av kortest mulig forbedringsveier i hvert skritt (Edmonds og Karps variant). Gitt en graf med kapasiteter, skal programmet skrive ut beste flyt, hvor mange forbedringsveier en måtte bruke med denne algoritmen, samt et kutt (det som algoritmen gir) som beviser at dette er optimal flyt.

Grafen er rettet, så mellom to noder u og v kan kapasiteten fra u til v være forskjellig fra den fra v til u. Alle kapasiteter er heltallige.

Dataformat

Input:

- Først en linje med antall noder m.
- Så m linjer med m tall i hver (en matrise/tabell) som angir kapasitetene mellom hvert par av noder. Dersom i er linjen og j er kolonnen til et tall, så angir tallet kapasiteten som kan gå fra node i og til node j, altså kapasiteten til kanten (i, j) i grafen.

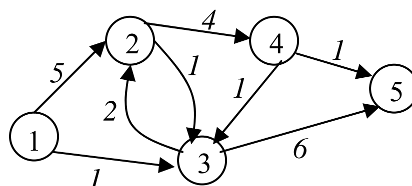
Vi kan tenke oss at nodene er nummerert fra 1 til m, og da er node 1 kilde og node m sluk.
Merk at det altså kan være positiv kapasitet både fra en node u til en node v, og fra v til u. Langs diagonalen angis tallet null. Det vil aldri gå kanter inn i kilden eller ut av sluket (og dermed vil første kolonne og siste linje alltid ha bare nuller).

Output:

- Først en linje først med et enkelt tall som angir antall forbedringsveier algoritmen brukte.
- Så m linjer med m tall i hver der flyten på kantene skrives ut, etter samme format som kapasitetene ble angitt på i input (vertikal indeks er *fra* og horisontal indeks er *til*).
- Til slutt en egen linje med nummeret på de nodene som er på kildesiden av et kutt med kapasitet lik flyten, i sortert rekkefølge. Kildenoden skal taes med og nodene skal indekseres fra 1 og oppover.

Eksempel
Under gir vi først et typisk input og så en tegning av det tilsvarende nettverket, med kapasiteter på kantene:

```
5
0 5 1 0 0
0 0 1 4 0
0 2 0 0 6
0 0 1 0 1
0 0 0 0 0
```



Dette skal gi som svar:

```
4
0 3 1 0 0
0 0 1 2 0
0 0 0 0 3
0 0 1 0 1
0 0 0 0 0
1 2 4
```

Tips til implementasjon

Vi er ikke kritiske til minnebruk, en kan fint unne seg tre-fire $m \times m$-arrayer her uten at det teller negativt. For eksempel kan en bruke én for det opprinnelige problemet (N), én for den flyten man i øyeblikket har på hver kant (f), og én med de flytforandringene som er mulig (Nf), som i figur 14.8.

Ellers er det jo bare å gjennomføre et bredde-først søk med en FIFO-kø, og det er kanskje lurt å ha en boolsk array som angir om noden er sett i dette søket..

Oppgave 3 (Algoritmer og avgjørbarhet)

1. Beskriv kort hvordan du ville argumentert for at en hvilken som helst algoritme kan implementeres i Java. Er det mulig å bevise en slik påstand formelt?

2. Hvilke av problemene under er uavgjørbare? Beskriv en Turing-maskin (inklusive tilstander og transisjoner) for å vise at et problem er avgjørbart, eller vis en reduksjon fra stoppeproblemet for å vise at et problem er uavgjørbart.)

a. $L_1 = \Sigma^*$ (der mengden Σ^* altså består av alle mulige strenger).

b. $L_2 = \{M \mid M \text{ gjenkjenner } L_1\}$

c. $L_3 = \{M \mid M \text{ gjenkjenner } L_2\}$

- - - o o o - - -