INF 4130 Oppgavesett 2, 13/09-2011 m/løsningsforslag

Vi skal starte litt mykt med noen korte oppgaver omkring algoritmers kjøretid og analyse av dette. Som dere vet bruker vi oftest *O*-notasjon (eller riktigere *asymptotisk notasjon*) til å angi kjøretiden. I notatet dere fikk på forelesningen (som også ligger på kurssidene) er det beskrevet fire ulike varianter av slik notasjon: *O*, Θ, Ω og *o*.

Oppgave 1

a) Vis at $n+1$ er $O(n)$.

For alle $n > 1$ er $n+n = 2n > n+1$, slik at vi altså har $n+1 = O(2n) = O(n)$.

b) Vis at $n \log n$ er $O(n^2)$.

For $n > 0$ er $n > \log n$, slik at vi altså har $n \log n = O(nn) = O(n^2)$.

c) Er $2^{n+1} = O(2^n)$?

For hvilken konstant c er $2^{n+1} \leq c\, 2^n$?

d) Er $\frac{10n+16n^3}{2} = O(n^2)$?

Nei, for alle konstanter c finnes en n slik at $n^3 > c \cdot n^2$. (n^3 vokser raskere enn n^2.)

Oppgave 2

a) Hva vet vi om en algoritmes kjøretid om denne er $O(n!)$?

Ikke mye, vi vet bare at kjørtiden er lavere enn $c \cdot n!$, for en konstant c, men kjøre-tiden kan i prinsippet ligge i intervallet $(0, c \cdot n!]$, som jo ikke sier oss så mye. (Som regel vil man nok med et sånt utsagn mene at kjøretiden ligger oppunder $n!$, i en eller annen forstand, men matematisk trenger ikke $O(n!)$ angi noen *tight* grense.)

b) Hva vet vi om en algoritmes kjøretid om denne er $\Omega(n)$?

Dette sier oss heller ikke mye, vi vet bare at kjøretiden er større enn $c \cdot n$, for en kons-tant c. Kjøretiden kan i prinsippet ligge i intervallet $(c \cdot n, \infty)$, som jo ikke sier oss så mye.

c) Hva vet vi om en algoritmes kjøretid om denne er $\Theta(2^n)$?

Her vet vi litt mer, analysen vår har nok vært litt mer nøyaktig enn i de to foregående tilfellene, men situasjonen er fortsatt ikke helt lys: kjøretiden til algoritmen vår vokser som 2^n, og er altså eksponensiell.

d) Hva vet vi om en algoritmes kjøretid om denne er $O(n^2)$?

Her vet vi at kjøretiden er lavere enn n^2, den kan være konstant, sub-lineær (log n eller liknende), lineær (n), eller ligge opptil n^2. I praksis er jo n^2 en såpass *tight* grense—det er ikke plass til **så**

mye mellom 0 og n^2, så vi kan vel si at vi at vi har nogenlunde god kunnskap om denne algoritmens kjøretid.

e) Utsagnet «Denne algoritmen har kjøretid minst O(n^2).» kan virke litt pussig. Har det mening?

Algoritmen har altså kjøretid «minst ikke mer enn en $c{\cdot}n^2$»? Mener man at kjøretiden ligger over n^2, bør man formulere seg på en annen måte.

Så fortsetter vi litt med oppgaver om strengsøk, delvis hentet fra læreboka. Bruk gjerne litt tid på gruppa til å repetere/diskutere hvorfor/hvordan de ulike skift-strategiene til Knuth-Morris-Pratt og forenklet Boyer-Moore (Horspool) virker.

Oppgave 3 (Oppgave 20.3 i Berman & Paul)

Simuler CreateNext side 637-8, bruk patternet “abracadabra”.

Merk for det første at det er en trykkfeil i CreateNext-prosedyren i boka. I linje 8 bakfra skal det stå "j <- Next[j]" (ikke "j <- Next[j-1]").
Ellers er Next for P = "abracadabra" slik:

```
a b r a c a d a b r a
0 0 0 0 1 0 1 0 1 2 3
```

Hva om vi beregner for patternet P’="abcdef"? Her er det jo ingen like bokstaver. Hvordan flyttes patternet da?

Oppgave 4

Beregn arrayet Shift[a:z] for patternet P = "announce" - simuler CreateShift side 639.

"Bad character shift" blir bergnet etter følgende mønster:

```
P[0] = a       Shift[P[0]] = 8 - 0 - 1 = 7
P[1] = n       Shift[P[1]] = 8 - 1 - 1 = 6
P[2] = n       Shift[P[2]] = 8 - 2 - 1 = 5
P[3] = o       Shift[P[3]] = 8 - 3 - 1 = 4
P[4] = u       Shift[P[4]] = 8 - 4 - 1 = 3
P[5] = n       Shift[P[5]] = 8 - 5 - 1 = 2
P[6] = c       Shift[P[6]] = 8 - 6 - 1 = 1
```

Svaret blir dermed som under. Andre tegn i alfabetet får Shift-verdi 8.

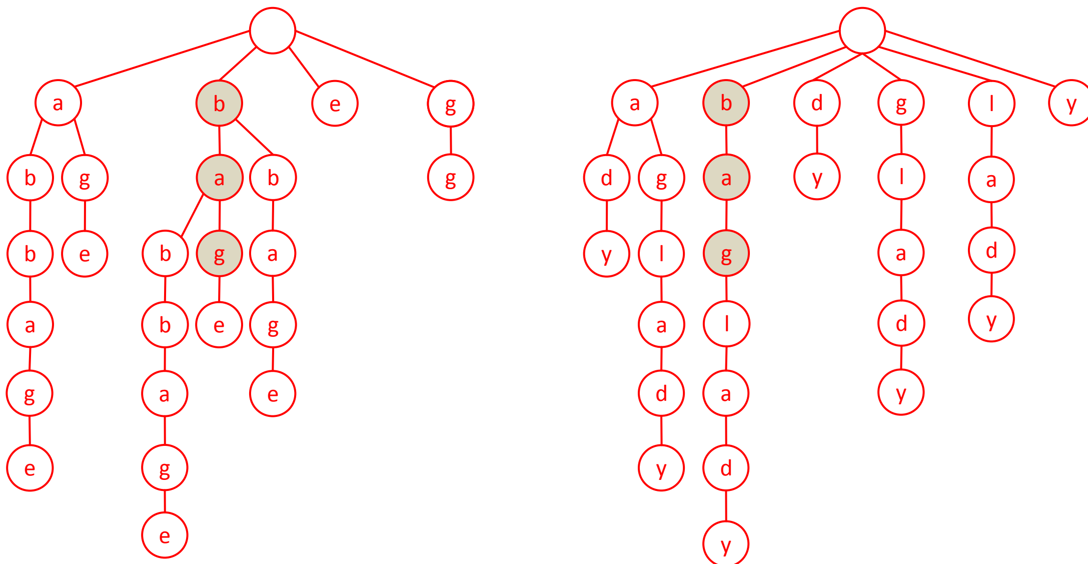
```
Shift[]
Shift[a] = 7
Shift[n] = 2
Shift[o] = 4
Shift[u] = 3
Shift[c] = 1
```

Oppgave 5

Tegn ukomprimerte suffikstrær for strengene "BABBAGE" og "BAGLADY". Sjekk så om "BAG" er en felles substreng. Kan du klare deg med bare ett tre?

```
BABBAGE    E                     BAGLADY    Y
           GE                               DY
           AGE                              ADY
           BAGE                             LADY
           BBAGE                            GLADY
           ABBAGE                           AGLADY
           BABBAGE                          BAGLADY
```

De to suffiks-trærne blir altså som følger. Å sjekke om "bag" er en felles substreng gjøres nå enkelt ved å sjekke om den finnes i begge trær. Dette kan selvsagt også gjøres med bare ett tre, suffiksene for begge strenger legges inn i samme tre, men vi markerer på passelig måte hvilken streng den aktuelle bokstaven kommer fra.



Ekstraoppgave

NB: Denne oppgaven krever litt kunnskap om regulære språk og NFA-er/DFA-er.

Som en generell problemstilling kunne vi tenke oss å søke etter en streng som passer med et gitt regulært uttrykk R, i en lengere streng S. Vi kan da i stedet si at vi forsøker å finne en delstreng fra starten av S som stemmer med det regulære uttrykket ".* R"

Her betyr "." et hviket som helst tegn i alfabetet, og stjernen betyr at det som står foran kan gjentas null eller flere ganger. Dermed betyr ".*" alle mulige strenger, inklusive den tomme.

Vi kan da løse søkeproblemet som følger: Vi lager først en NFA (ikkedeterministisk endelig automat) som tilsvarer ".*R" (dette er lett, enten intuitivt eller med en såkalt Thompson-konstruksjon), og lager denne om til en DFA (deterministisk endelig automat) med standard-algoritmen for NFA->DFA.

Denne DFA-en er svært lett å lage om til et program som leser S i lineær tid, og hver gang vi kommer til en sluttilstand i DFA-en vet vi at vi akkurat har lest noe som passer med R.

Spørsmål: Hvorfor er dette ikke en så rask algoritme som det i første omgang kan synes? Hva er det som begrenser hastigheten på denne algoritmen? Når vil den være rask?

Det interessante her er altså at om man har fått laget DFA'en for ".*R" (f.eks som en tabell), så blir algoritmen for å lete etter R-treff i T meget rask og grei. Det er også raskt og greit å sette opp NFA'en for ".*R". Det som gjør at denne algoritmen dog kan bli sen er at størrelsen av DFA'en (og dermed tiden til å lage den fra NFA'en) kan bli eksponensiell i forhold til størrelsen av NFA'en. Dermed er algoritmen i verste fall eksponensiell i lengden av R.

Dog kan det gjøres mye optimalisering her for spesielle tilfeller, og det er masse å lese omkring dette. Man kan også unngå å lage DFA'en.

[slutt]