INF 4130 Exercise set 3, 08/09-2015 w/proposed solutions

We start with a few short exercises on algorithm running times, and running time analysis. As you know we usually use *O*-notation (more correctly, asymptotic notation) for running times. A short note on the course web page describes four variants of asymptotic notation: *O*, Θ, Ω and *o*.

Exercise 1

a) Show that $n+3$ is $O(n)$.

For all $n > 3$ we have $n+n = 2n > n+3$, such that $n+3 = O(2n) = O(n)$.

b) Show that $2n \log n$ is $O(n^2)$.

For $n > 0$ we have $n > \log n$, such that $2n \log n = O(2n*n) = O(n^2)$.

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

For which constant c is $2^{n+1} \leq c\ 2^n$?

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

No, for alle constants c, there is an n such that $n^3 > c \cdot n^2$. (n^3 grows faster than n^2.)

Exercise 2

a) What do we know about the running time of an algorithm if it is $O(n!)$?

Not much, we only know that the running time is lower than $c \cdot n!$, for some constant c, but the running time may in principle lie anywhere in the interval $(0, c \cdot n!]$, so it doesn't tell us a whole lot. (Usually one would probably mean that the running time is close to $n!$, in some sense, but mathematically $O(n!)$ need not be a tight bound.)

b) What do we know about the running time of an algorithm if it is $\Omega(n)$?

Again, not much. We only know that the running time is larger than $c \cdot n$, for some constant c. The running time may in principle lie anywhere in the interval $(c \cdot n, \infty)$, not really telling us much.

c) What do we know about the running time of an algorithm if it is $\Theta(2^n)$?

Here we know a bit more, our analysis has probably been a bit more thorough than in the last two cases, but the situation is far from perfect: the running time of our algorithm grows as 2^n, an exponential growth of the running time as the size of the input grows.

d) What do we know about the running time of an algorithm if it is $O(n^2)$?

Here we know that the running time is lower than n^2, it can be constant, sub-linear (log n or similar), linear (n), or close to n^2. N real life n^2 is a fairly tight limit — there isn´t *that* much room between 0 and n^2, so we have a fairly good understanding of algorithm running time.

e) The statement “This algorithm has a running time of at least $O(n^2)$.” may seem odd. Does it make sense?

The running time of the algorithm is “larger than lower than $c \cdot n^2$”? If we want to indicate that the running time lies above n^2, we should rephrase.

We now continue with a few exercises on string search, partially from the textbook. Spend some time repeating/discussing why/how the different shift strategies of Knuth-Morris-Pratt and simplified Boyer-Moore (Horspool) work.

Exercise 3 (Exercise 20.3 in Berman & Paul)

Simulate CreateNext page 637-8, use the pattern “abracadabra”.

First, note that there is a typo in the CreateNext routine in the book. Line 8 from below should be "j <- Next[j]" (not "j <- Next[j-1]").
Otherwise Next for P = "abracadabra" is:

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

What about the pattern P’="abcdef"? It has no repeated letters (thus no overlap). How is the pattern moved now?

Exercise 4

Calculate the array Shift[a:z] for the patterns P = "announce" and P’ = "honolulu" - simulate CreateShift spage 639.

"Bad character shift" for P = "announce" is calculated like this:

```
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
```

The answer is therefore. All other symbols in the alphabet get a Shift value of 8.

```
Shift[a] = 7
Shift[n] = 2
Shift[o] = 4
Shift[u] = 3
Shift[c] = 1
Shift[*] = 8    (* indicates the rest of our alphabet, {a,…,z}\{a,n,o,u,c}.)
```

For P’ = "honolulu" the answer is:

```
Shift[h] = 7
Shift[o] = 4
Shift[n] = 5
Shift[l] = 1
Shift[u] = 2
Shift[*] = 8
```

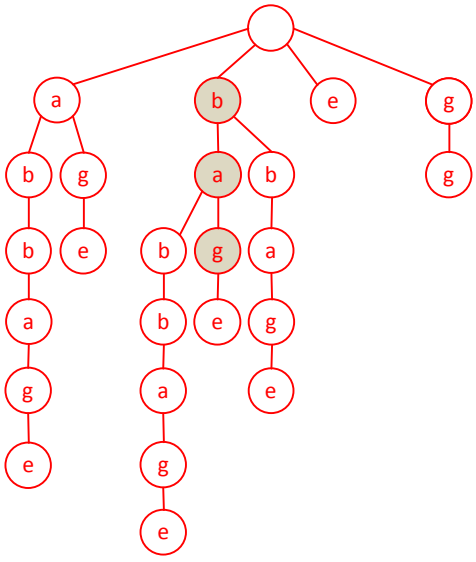
Exercise 5

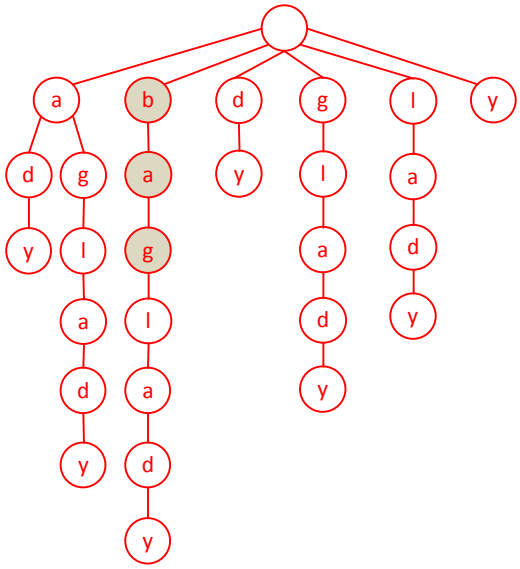
Draw uncompressed suffix trees for the strings "BABBAGE" and "BAGLADY". And check if "BAG" is a common substring. Can you make do with only one tree?

```
BABBAGE    E
           GE
           AGE
           BAGE
           BBAGE
           ABBAGE
           BABBAGE
```

```
BAGLADY    Y
           DY
           ADY
           LADY
           GLADY
           AGLADY
           BAGLADY
```

The two suffix trees are as follows. Checking if "bag" is a common substring is done bay checking for it in both trees. It can of course also be done with one tree, we just insert suffixes for both strings in the same tree, and mark their origins in some suitable way.





Extra

NB: some background knowledge on regular languages and NFAs and DFA is needed.

As a general problem setting we may want to search for a string matching a given regular expression *R* in a longer string *S*. We may then reformulate the problems as searching for a string matching the regular expression ".**R*" at the start of *S*. Here "." Means any symbol in our alphabet, and the asterisk means that what comes before it may be repeated zero or more times. So ".*" just means that anything can come before the string we really want (expressed by *R*), including the empty string.

We may solve the problem as follows: first construct an NFA (non-deterministic finite automaton) corresponding to ".**R*". This can be done intuitively, or by a so-called Thompson-construction. Then we transform this non-deterministic machine into a DFA (deterministic finite automaton) in the standard way.

This DFA is easily transformed into a normal computer program that reads *S* in linear time, and every time we arrive in a final state for the DFA, we know that we have read something that matches with *R.*

QUESTION: Why is this method not as fast as it might seem? What limits it´s running time? When will it be fast?

The interesting part here is that if we have made the DFA for ".*R", the algorithm for finding R-matches in S is quick. Designing the NFA for ".*R" is also straight forward and quick. What may make the algorithm slow is the size of the DFA (and therefore the time to construct it), which may be exponential relative to the size of the NFA. Worst case, our algorithm may be exponential in the size of *R*.

There are of course many optimizations one can do in special cases, and a lot of literature in the field. It is also possible to avoid constructing the DFA.

[end]