IN3130 Exercise set 3

We start with a few short exercises on algorithm running times and running time analysis. This math is central to complexity theory, and important to have a good understanding of. As you know we usually use *O*-notation (more correctly, *asymptotic notation*) to indicate running times of algorithms. A short note on the course web page describes four variants of asymptotic notation: *O*, Θ, Ω and *o.*

Exercise 1

a) Show that $n \cdot 3$, $n+3$ both are $O(n)$.

$n \cdot 3 = O(3n) = O(n)$. For $C = 3$, we always have $3n \leq C \cdot n$.

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 $2n \log n = O(2n \cdot n) = O(n^2)$.

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 all 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. In 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 continue with a few exercises on string search, partially from the textbook.

Exercise 3

Spend some time repeating/discussing why/how the different shift strategies of Knuth-Morris-Pratt and simplified Boyer-Moore (Horspool) work. Pay attention to what parts of the pattern P and T overlap with what, and why that is necessary for a match to be possible.

Answer 3

Left to the student

Exercise 4

Find the overlapping prefixes and suffixes (as defined in the Knuth-Morris-Pratt-algorithm) for the string "ababc"

Answer 4

j = 0: no substrings to overlap
j = 1: no substrings to overlap
j = 2: (suffix of P[1..1] = b that is prefix of P[0..0] = a) no overlap
j = 3: (suffix of P[1..2] = ba that is prefix of P[0..1] = ab) overlap: a
j = 4: (suffix of P[1..3] = bab that is prefix of P[0..2] = aba) overlap: ab

Exercise 5 (Knuth-Morris-Pratt, Exercise 20.3 in Berman & Paul)

Simulate CreateNext pages 637-8 in Berman & Paul – calculate the Next[]-array for the pattern "abracadabra".

Answer 5

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 5 (Horspool)

Simulate CreateShift page 639 in Berman & Paul – calculate the array Shift[a:z] for the patterns P_1 = "announce", and P_2 = "honolulu".

Answer 5

"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
```

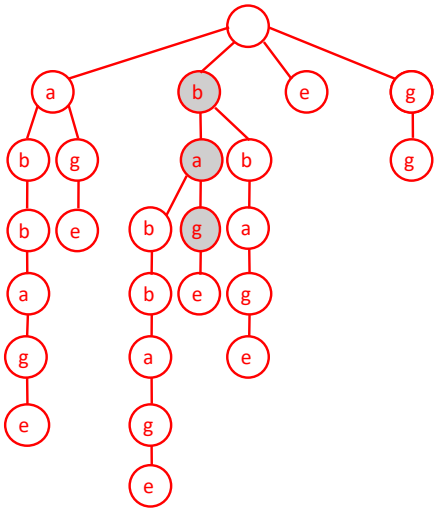
Exercise 6

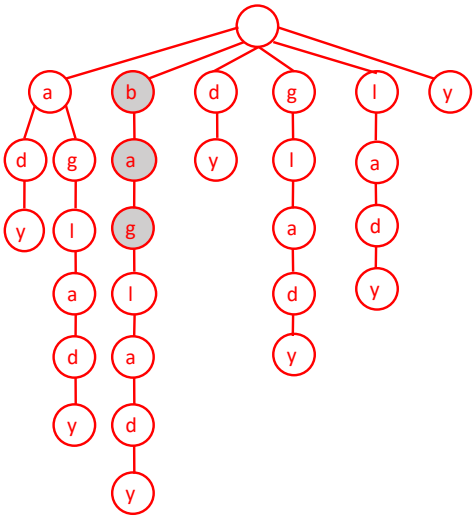
Draw uncompressed suffix trees for the strings "BABBAGE" and "BAGLADY". And check if "BAG" is a common substring. Can you make do with only one tree?

Answer 6

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


a
b
e
g
b
g
a
b
g
b
e
b
g
a
a
b
e
g
g
a
e
b
a
g
e


a
b
d
g
l
y
d
g
a
y
l
a
d
y
y
l
g
a
d
y
a
l
a
d
y
d
a
d
y

[end]