Övning 3 - Tillämpad datalogi 2013

```
0 # coding: latin
```

Summering

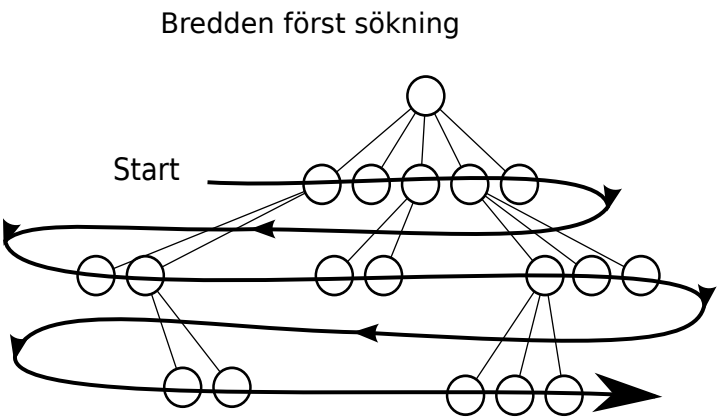
Vi gick igenom **problemträd**, sökning i problem träd med **bredden först**, och **djupet först**. Vi exemplifierade det genom att lösa två uppgifter **strykord**, med djupet först, och **sjuor i rad**, med bredden först.

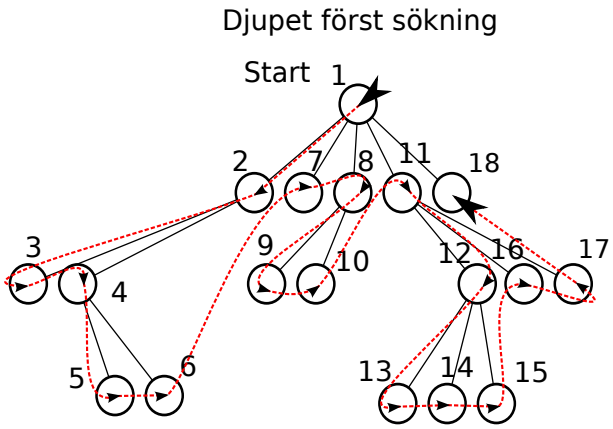
- Some imports, ignore

```
20 import sys, os
21 sys.path.append(os.getcwd())
```

Problemträd, breddenförst, djupetförst

- Ett problemräd kan ses som en enkelriktad graf där varje nod kan ha noll eller flera underordnade noder.
- Bredden först söker av problemträdet nivå för nivå
- Djupet först söker igenom problemträden genom att gå på djupet från vänster till höger
- Djup-först sökning är en strategi för att traversera ett träd där man följer varje gren i trädet till dess yttersta nod (dess löv) innan man backar upp i hierarkin och väljer nästa gren att undersöka.
- En implementaion av djupet först sökning påminner om trädutskrifen **Preordning:** (noden) (vänster träd) (höger träd), med skillnaden att vi nu kan ha fler kanter än två till en nod. Regeln blir då istället (noden) (träd längst till vänster),...,(träd längst till höger träd)





1. Strykord

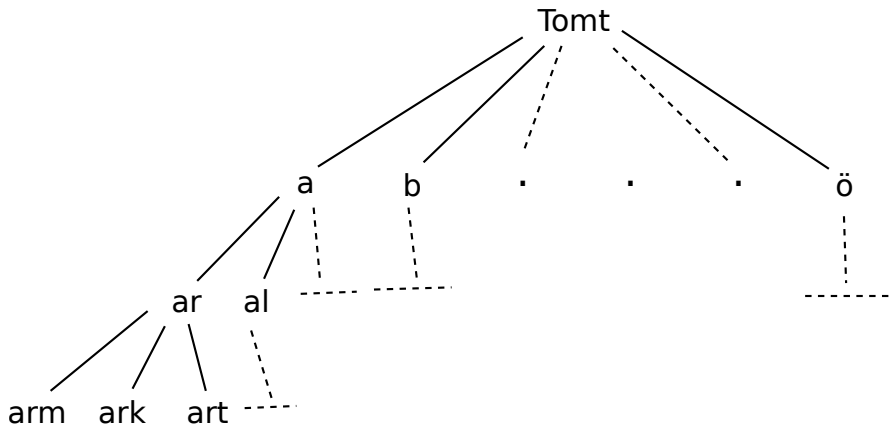
Ordet orkan är ett strykord eftersom man kan stryka sista bokstaven om och om igen och bara få riktiga ord.

orkan - orka - ork - or - o

Uppgiften är att finna det längsta strykordet i svenska språket. Ordlistan finns på fil. Rita problemträdet och jämför olika tänkbara algoritmer.

Vi skapar problemträdet genom att börja med ett tomt ord, och sedan skapar vi barn genom att lägga på en bokstav och kolla ifall det ordet finns. *Vilken sökningsmetod passar för detta problem?* Eftersom vi behöver gå igenom hela trädet för att vara säkra på att vi kollat alla möjligheter, dvs göra en uttömmande sökning, spelar det ingen roll. Låt oss välja djupet förstsökning eftersom den kan elegant programeras med rekursion. Vi använder därför en *rekursiv djupetförst*-sökning. Orden läggs i en *dict* - kan använda binärträd eller något smartare som sparar minne.

Problemträdet vi får ser ut på följande sätt:




```
67
68 # coding:iso-8859-1
69
70 def langsta_strykord(ord, ordlista):
71     """ Rekursiv djupetförst som hittar längsta strykordet givet orden
72     i ordlista """
73
74     alfabet="abcdefghijklmnopqrstuvwxyzåäö"
75     global rekord
76     if len(ord) > len(rekord):
77        rekord = ord
78     for tkn in alfabet:
79       if ordlista.has_key(ord+tkn):
80           langsta_strykord(ord+tkn, ordlista)
```

Vi behöver en ordlista

```
83 rekord=""
84 svenska={} #Dictionary
85
86 svenskfil = open("ss100.txt")
87 for rad in svenskfil.readlines():
88     svenska[rad.strip().lower()]=True #Ta bort returtecknet och gör om till
       upper
89
90
91 langsta_strykord("", svenska)
92 print "Längsta strykord: None"
```

```
Längsta strykord: None
```

```
91 s=''
92 for b in rekord:
93     s+=b
94     print '->', s,
```

```
-> s -> sk -> ska -> skar -> skarp -> skarpa -> skarpas -> skarpast ->
    skarpaste -> skarpastes
```

2. Sjuor till hundra

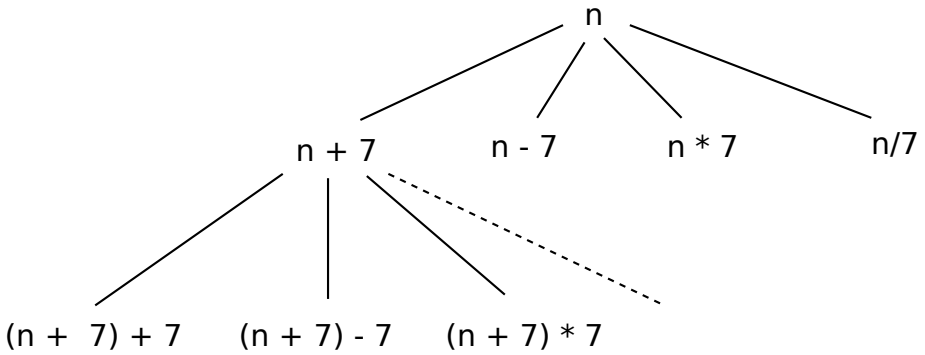
Det gäller att skriva talet 100 med enbart sjuor och dom fyra räknesätten, till exempel så här:

*7 +7 /7 *7 *7 =98*

Det var ett gott försök som inte nådde ända fram. För att man ska få använda / måste divisio- nen gå jämnt ut. Rita problemträdet och diskutera bästa algoritm för att avgöra OM problemet är lösbart.

Om man dessutom vill veta hur lösningen ser ut krävs en mer komplicerad datastruktur. Beskriv den och skissa ett program.

Vi vill hitta kortaste sättet att komma fram till hundra. Som gjort för breddenförst eftersom
vi då går igenom trädet nivå för nivå och OM vi påträffar en lösning kommer det garanterat vara den kortaste.
Låt oss använda en kö där vi lagrar resultaten från beräningarna vi gör när vi travesterar problemträdet nivå för nivå. OBS vi utför endast division om sju är delbart med n.



Så här skulle vi kunna göra:

Kön

Stoppa in n i kön

n_0

Ta ut n från kön.
$_0$
Är det en lösning? Om inte gå vidare. Annars stanna.

Tom

Skapa barnen till n och
$_0$
lägg in i kön. Där:

n_{01} n_{02} n_{03} n_{04}

n_{01} = n +7

n_{02} = n -7

n_{03} = n *7

n_{04} = n /7 (om n är delbart med 7)

Ta ut n från kön.
$_1$
Är det en lösning? Om inte gå vidare. Annars stanna.

n_{02} n_{03} n_{04}

Skapa barnen till n och
$_{01}$
lägg in i kön. Där:

n_{02} n_{03} n_{04} n_{011} n_{012} n_{013} n_{014}

Ta ut n från kön.
$_2$
Är det en lösning? Om inte gå vidare. Annars stanna.

n_{03} n_{04} n_{011} n_{012} n_{013} n_{014}

OSV...

Våra algoritm blir:
0. Stoppa in roten i kön
1. Ta ut första värdet i köm
2. Om det inte är en lösning skapa baren och sätt in i kön. Annars stanna
3. Gå till 1.

```
126
127 from queue import Queue
128
129 q = Queue()
130 def makesons(num, q):
```

```
131     if num==100:
132         print "Hundra!",
133         return False
134     q.put(num+7)
135     q.put(num-7)
136     q.put(num*7)
137     if(tal%7==0):
138         q.put(num/7)
139     return True
140 
141 m=True
142 q.put(0)
143 while (not q.isempty()) and m:
144     m=makesons(q.get(), q)
```

Error:

```
Traceback (most recent call last):
  File "/usr/local/lib/python2.7/dist-packages/pyreport-0.3.4c-py2.7.egg/pyreport/main.py"
    exec block_text in self.namespace
  File "<string>", line 4, in <module>
  File "<string>", line 10, in makesons
NameError: global name 'tal' is not defined
```

Notera dock att vi kommer att testa samma tal många ggr:

0 ⇒ [7, -7, 0, 0]
7 ⇒ [14, 0, 49, 1]
-7 ⇒ [0, -14, -49, ...]

Notera att vi får ett stort span av tal med många dubletter:

Totalt antal: ⇒ 19568
Antal unika: ⇒ 809
Med många stora och små tal ⇒
[−823543. − 235298. − 134456.]...[−1.0.1.2.]...[252105.352947.823543.]

För att få ut kedjan måste varje nod minnas sin "far", dvs varifrån den kom- mer. Det lättaste är att returnera den noden istf. False från makesons (istf. True returnerar man None), men det går även att använda ett Exception.

```
167 
168 class Node:
169     def __init__(self, num = 0, father = None, op='?'):
170         self.num = num
171         self.father = father
172         self.op = op
173 
174 from queue import Queue
175 from sys import exit
176 
177 q=Queue()
178 
179 def makesons(father, q):
180     num = father.num
181     if num == 100:
182         raise StopIteration(father)
183     q.put(Node(num+7, father,'+'))
184     q.put(Node(num-7, father,'-'))
185     q.put(Node(num*7, father,'*'))
186     if(num%7==0):
187         q.put(Node(num/7, father,'/'))
188 
```

```
189 def writechain(father):
190     if father != None:
191         writechain(father.father)
192         print father.op, 7, '->', father.num
193 
194 q.put(Node())
195 
196 try:
197     while not q.isempty():
198         makesons(q.get(), q)
199 except StopIteration as si:
200     writechain(si.args[0])
```

```
? 7 -> 0
+ 7 -> 7
+ 7 -> 14
* 7 -> 98
* 7 -> 686
+ 7 -> 693
+ 7 -> 700
/ 7 -> 100
```

```
196 
197 # StopIteration: Raised by an iterator's next() method to signal that there are
198 # no further values. This is derived from Exception rather than
199 # StandardError, since this is not considered an error in its
200 #normal application.
```

Misc

Graph

- A "graph" in this this context is a collection of "vertices" or "nodes" and a collection of edges that connect pairs of vertices.
- Complexity bredden först, O(V)+O(E)=O(V+E) där V är antalet noder(vertex) och E antalet kopplingar (edges).
- Complexity djupet först, O(V)+O(E)=O(V+E) där V är antalet noder(vertex) och E antalet kopplingar (edges).

Searching

Even though a binary search is generally better than a sequential search, it is important to note that for small values of n, the additional cost of sorting is probably not worth it. In fact, we should always consider whether it is cost effective to take on the extra work of sorting to gain searching benefits. If we can sort once and then search many times, the cost of the sort is not so significant. However, for large lists, sorting even once can be so expensive that simply performing a sequential search from the start may be the best choice. [ref http://interactivepython.org/courselib/static/pythonds/SortSearch/searching.html#lst-binarysearchpy]

Hash table

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty.

- Load factor lambda=numberofitems/tablesize

Creating hashfunctions

- Folding, divide and add. For 436-555-4601, divide digits into groups of 2 (43,65,55,46,01) and add 43+65+55+46+01, giving the hash number 210 Collision resolution
- linear probing,

  - rehash(pos)= (pos+1)% sizeoftable
  - rehash(pos)= (pos+3)% sizeoftable
  - rehash(pos)= (pos+skip)% sizeoftable

- quadratic probing
  - rehash(pos)= (pos+skip)% sizeoftable, skip =1,3,5,7