# Instruktioner - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

## Hjälpmedel

Följande hjälpmedel är tillåtna:

- Exakt en valfri bok, t.ex. den rekommenderade kursboken. Boken får ha anteckningar, men inga lösa lappar.
- Exakt ett A4-papper med egna anteckningar om precis vad som helst. Det kan vara hand- eller maskinskrivet, på ena eller båda sidorna.
- Pennor, radergummin och liknande för att kunna skissa lösningar på papper.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar utöver det som listats ovan.

Lösa tomma papper för att göra anteckningar tillhandahålls av tentamensvakterna.

## Genomförande

Under datortentan kommer ni att arbeta i särskilda datortentakonton där ni har tillgång till en starkt begränsad miljö. Ni har t.ex. ingen koppling till internet. I praktiken kommer miljön bestå av ett terminalfönster och Emacs.

Starta tentaklienten från menyn som visas när man högerklickar på skrivbordet. Tentaklienten används för att ställa frågor till examinator under tentan, samt för att skicka in lösningar.

Varje uppgift kan skickas in en enda gång och kommer därefter poängsättas. Det finns alltså inga möjligheter till komplettering.

## Betygsättning

Datortentan består av totalt sex uppgifter som vardera kan ge maximalt fem poäng. Uppgifterna är utvalda för att ha olika svårighetsgrad och är ordnade så att den lättaste oftast kommer först. Observera att vad som är lätt eller svårt naturligtvis skiljer sig från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet. Betygssättningen sker enligt följande tabell:

- För betyg 3 krävs minst 12 poäng.
- För betyg 4 krävs minst 19 poäng.
- För betyg 5 krävs minst 25 poäng.

## Rättningskriterier

Följande allmänna kriterier används vid poängsättning av uppgifter. Lösningar som bryter mot dessa kommer att få poängavdrag.

- Funktioner ska ha exakt samma namn som i uppgiften. Detta underlättar rättningen.
- Namn på parametrar, variabler och hjälpfunktioner som inte specificerats explicit i uppgiften ska vara beskrivande och följa namnstandarden.
- Lösningen ska följa de regler som uppgiften har satt upp. Det kan t.ex. röra sig om att vissa funktioner eller metoder ej får användas, eller att lösningen ska göras enligt en särskild modell.
- Lösningen ska fungera exakt som i körexemplen i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara generell, d.v.s. den ska inte enbart fungera för de körexempel som finns i uppgiften, utan även för alla andra möjliga indata på samma form. Felkontroller av indata behöver dock normalt inte göras, om inte uppgiften särskilt uttrycker det.

# Uppgifter - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Tisdag 14 januari kl 8-13

## Uppgift 1

### Deluppgift 1a (2p)

Det torde vara allmänt känt att toner, musikens atomer, kan betecknas med vanliga bokstäver. Vi talar t.ex. om tonen A eller om tonen G. De bokstäver som används för att beteckna toner återfinner vi också i vanlig löpande text. Vad skulle hända om man tog en vanlig text, plockade ut alla bokstäver som också betecknar toner och sedan spelade dem en i taget? Det skulle antagligen låta fruktansvärt illa, så det måste vi pröva, åtminstone i teorin.

Vi vill alltså ha en funktion `find_notes` som tar en sträng. Som utdata vill vi ha en sträng som enbart innehåller de bokstäver som betecknar toner, i samma ordning som de förekom i texten. De toner vi bryr oss om i den här uppgiften är (i ordning) C, D, E, F, G, A och H. Alla förekomster av dessa bokstäver, stora eller små, ska alltså samlas ihop till en sträng, som enbart innehåller små bokstäver.

```
>>> find_notes("Bananer")
'aae'
>>> find_notes("Andningshål")
'adgh'
>>> find_notes("Nu ska vi testa denna lilla funktion")
'aeadeaaf'
```

### Deluppgift 1b (3p)

Om man inte är jättevan att hantera toner kan man tycka att strängen `'ccceddddf'` bara är konstig, men den som kan läsa noter ser omedelbart att det är starten på *Gubben Noak*. För att göra det lite lättare för oss ska vi fixa en funktion som kan visualisera tonerna lite tydligare. Vi önskar en funktion print_notes som tar en sträng som enbart innehåller någon av bokstäverna i strängen `"cdefgah"` och som skriver ut dem som ett slags förenklat notsystem. Exempel:

```
>>> print_notes("ccceddddfcdefgah")
..............h
.............a.
............g..
.......f...f...
...e......e....
....ddd..d.....
ccc.....c......
```

Av exemplet framgår tonernas ordning nerifrån och upp, samt att vi fyller ut hela figuren med punkter. Nu kan vi lite tydligare se starten på *Gubben Noak*, plus en enkel skala.

# Uppgift 2

Skriv en funktion som samsorterar två sorterade listor. Utgå från att elementen alltid är sorterade i stigande ordning. Exempel:

```
>>> merge([1, 3, 6, 8], [2, 4, 5, 9])
[1, 2, 3, 4, 5, 6, 8, 9]
>>> merge(['a', 'j', 's'], ['f', 'm'])
['a', 'f', 'j', 'm', 's']
```

Funktionen ska finnas i två varianter: `merge_r` som arbetar rekursivt och `merge_i` som arbetar iterativt. Listorna ska i huvudsak behandlas ett element i taget och lösningarna får inte inbegripa omvandling till andra sammansatta datatyper.

# Uppgift 3

## Deluppgift 3a (3p)

Skriv en funktion `insert` som tar en sorterad lista och placerar ett element på rätt plats. Listan kan vara sorterad på olika sätt, och för att ange var elementet ska placeras skickas även en ordningsfunktion med. Exempel:

```
>>> insert(2, [1, 3, 4], lambda x, y: x < y)
[1, 2, 3, 4]
>>> insert(5, [10, 7, 4, 2], lambda x, y: x > y)
[10, 7, 5, 4, 2]
```

## Deluppgift 3b (2p)

Med hjälp av insert vill vi sedan ha två ytterligare funktioner som ska fungera enligt följande exempel:

```
>>> insert_abs(3, [1, -2, -4, 5])
[1, -2, 3, -4, 5]
>>> insert_abs(-3, [1, -2, -4, 5])
[1, -2, -3, -4, 5]
>>> insert_seq([1, 2], [[7, 7, 7], [45]])
[[7, 7, 7], [1, 2], [45]]
```

Funktionen `insert_abs` ska sortera in ett tal i en lista där talen kommer i stigande ordning enligt absolutvärdet. Funktionen `insert_seq` ska sortera in en lista i en lista av listor som är sorterad i fallande ordning efter listans längd. Lösningarna ska använda den tidigare definierade funktionen `insert`.

## Uppgift 4

Skriv en funktion `longest_sequence` som tar en rak lista som indata. Funktionen ska gå igenom elementen och hitta den längsta delsekvensen som består av samma element, och returnera en tupel bestående av detta element samt längden på denna längsta delsekvens. Om det finns flera delsekvenser av samma längd ska det först förekommande elementet returneras. Exempel:

```
>>> longest_sequence(['a','a','b','b','c','c','c'])
('c', 3)
>>> longest_sequence([1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0])
(1, 3)
```

## Uppgift 5

Vi har lagrat information om några personers barn i en dictionary. Värdena i vår dictionary är tupler (vilket förklarar varför det finns extra kommatecken efter de barn som är ensamma). Vi förutsätter för enkelhetens skull att alla namn är unika och önskar nu en funktion `descendants` som finner alla avkomlingar till en person. Ordningen mellan avkomlingarna är utan betydelse, men resultatet ska returneras i form av en lista (inte en tupel). Vår exempellista:

```
children = {'Linus': ('Eva', 'Per'),
            'Linnea': ('Per', ),
            'Eva': ('Emilia', 'Emil'),
            'Per': ('Stina', ),
            'Stina': ('Lillan', )}
```

Några körexempel:

```
>>> descendants('Linus', children)
['Eva', 'Per', 'Emilia', 'Emil', 'Stina', 'Lillan']
>>> descendants('Linnea', children)
['Per', 'Stina', 'Lillan']
>>> descendants('Torsten', children)
[]
```

## Uppgift 6

En metod att lagra fakta är att strukturera dem i ett så kallat diskrimineringsträd. Vi talar här om enkla faktapåståenden som "Karl är far till Lisa", "Nils är en pojke" eller "Avståendet mellan Linköping och Norrköping är 40". Vi kan representera dessa, och ytterligare ett par fakta, som listor i Python så här:

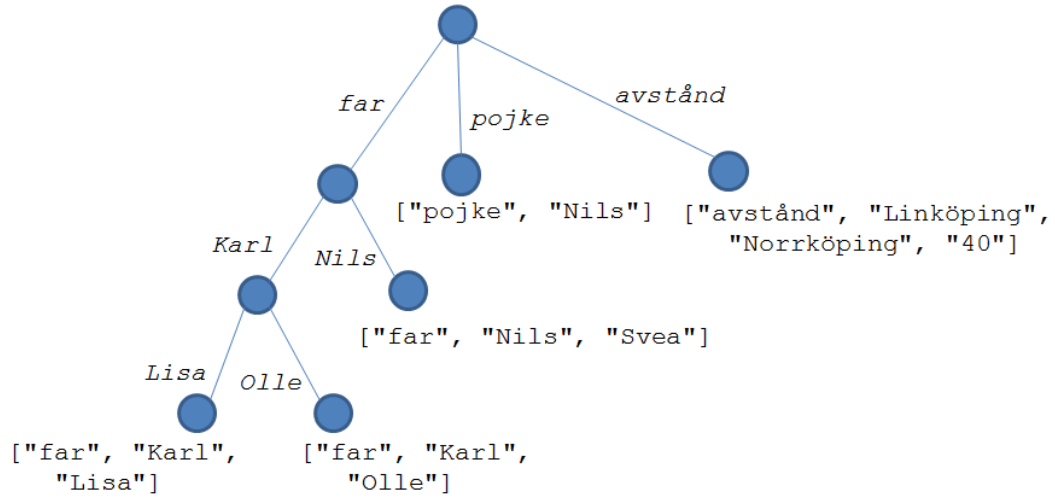
```
["far", "Karl", "Lisa"]
["pojke", "Nils"]
["avstånd", "Linköping", "Norrköping", "40"]
["far", "Nils", "Svea"]
["far", "Karl", "Olle"]
```

Genom att lagra dessa i ett diskrimingeringsträd möjliggör man en ganska snabb uppslagning av fakta. Vi kan alltså relativt enkelt tala om huruvida ett faktum finns i trädet eller ej. Vi grenar upp (diskriminerar) fakta genom att alla fakta vars första element är lika läggs under en och samma gren.

De tre fakta som inleds med "far" i exemplet ovan hamnar under en gren. Dessa fakta måste sedan i sin tur grenas upp med avseende på andra elementet och så vidare.

Om bara ett faktum finns under en gren görs ingen ytterligare förgrening. När man lägger in ett nytt faktum startar man i rotnoden. Finns en förgrening med första elementet följer man den, annars lägger man till en ny gren.

Vi kan åskådliggöra ovanstående fakta så här:




För att kunna hantera diskrimineringsträd vill vi införa ett antal abstrakta datatyper:

- **item** är en primitiv datatyp som beskriver alla grundläggande element, t.ex. "Karl", "far", "40"
- **fact** är en sekvens av **item** som används för att beskriva ett faktum, d.v.s. ett löv i trädet
- **d_tree** är unionen av **fact** och **branch_seq**, d.v.s. ett löv eller en sekvens med grenar
- **branch_seq** är en sekvens av **branch** som används för att beskriva grenarna
- **branch** är en tupel av **item** och **d_tree** som används för att beskriva det diskriminerande elementet och en nod i trädet

I filen `discrim_s.py` finns ett program för att bygga upp och lägga till fakta till ett diskrimineringsträd. Där används följande representation:

- **item** = en sträng
- **fact** = en lista `["FACT", item1, item2, ...]`
- **branch_seq** = en lista `[branch1, branch2, ...]`
- **branch** = en lista `[item, d_tree]`

Huvudfunktionen `insert_dt` lägger in ett nytt faktum i trädet.

## Deluppgift 6a (1p)

I programkoden i filen `discrim_s.py` saknas definitioner av funktionerna `is_fact` och `item`. Definiera dessa i linje med resten av koden.

## Deluppgift 6b (4p)

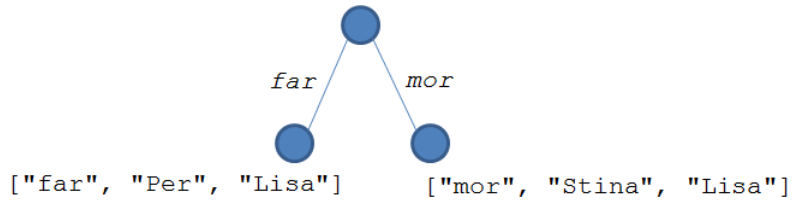
Den centrala funktionen `discriminate` saknas också. Den tar två fakta, diskriminerar (gör skillnad mellan) dessa och bygger upp ett diskrimineringsträd. Dessutom tar den en parameter som anger vilket element vi skall börja diskrimineringen med. Definiera funktionen `discriminate` abstrakt. Nya primitiver får införas om det är motiverat. Följande körexempel illustrerar hur discriminate ska fungera.

Först definierar vi tre fakta att arbeta med:

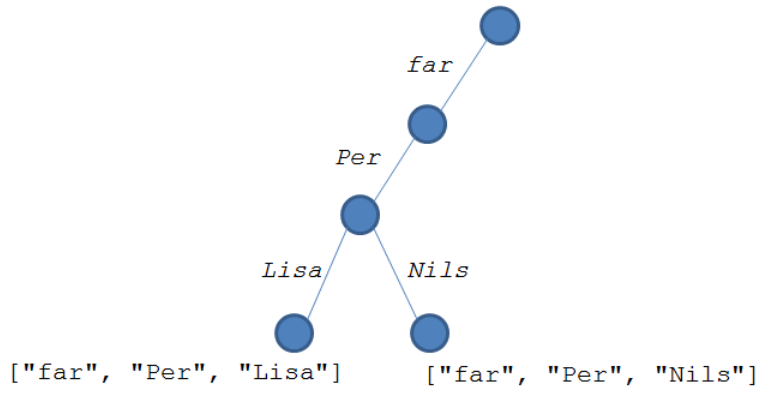
```
fact1 = build_fact(["far", "Per", "Lisa"])
fact2 = build_fact(["mor", "Stina", "Lisa"])
fact3 = build_fact(["far", "Per", "Nils"])
```

Därefter gör vi följande anrop till discriminate. Varje svar illustreras även grafiskt.

```
>>> discriminate(fact1, fact2, 1)
[['far', ['FACT', 'far', 'Per', 'Lisa']], ['mor', ['FACT', 'mor',
'Stina', 'Lisa']]]
```




```
>>> discriminate(fact1, fact3, 1)
[['far', [['Per', [['Lisa', ['FACT', 'far', 'Per', 'Lisa']], ['Nils',
['FACT', 'far', 'Per', 'Nils']]]]]]]
```




```
>>> discriminate(fact1, fact3, 2)
[['Per', [['Lisa', ['FACT', 'far', 'Per', 'Lisa']], ['Nils', ['FACT',
'far', 'Per', 'Nils']]]]]
```