Instruktioner - Datortentamen TDDE24 och 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 du att arbeta i en begränsad och övervakad miljö. Med hjälp av en särskild tentaklient skickar du in dina svar. Du kan även använda tentaklienten för att ställa frågor till examinator. Besök i skrivsalen kommer endast ske vid allvarligare tekniska problem.

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 vara välstrukturerad och väldokumenterad på samma sätt som kursens laborationer.
- 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.
- Den implementerade lösningen ska kunna köras inom rimliga tidsramar. Till exempel är en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa inte acceptabel.

Uppgifter - Datortentamen TDDE24 och TDDD73 Funktionell och imperativ programmering (i Python)

Onsdag 22 augusti 2018 kl 14-19

Uppgift 1

Medianen av en samling värden är ett alternativ till medelvärdet framför allt för skeva distributioner.

- För samlingar med ett ojämnt antal tal är medianen det mittersta talet om de ordnas i storleksordning.
- För samlingar med ett jämnt antal tal är medianen medelvärdet av de två mittersta talen om de ordnas i storleksordning.

Skriv en funktion `median(seq)` som givet en icke-tom lista med värden räknar ut medianen enligt ovan. Exempel:

```
>>> median([1, 1, 2, 3, 5, 7, 7, 7, 8])
5

>>> median([1, 1, 2, 3, 4, 7, 7, 7, 8, 9])
5.5

>>> median([1, 7, 2, 3, 7, 5, 8, 3, 7, 1])
4
```

Uppgift 2

Skriv en funktion interval som tar en lista med tupler [(s_1, e_1), (s_2, e_2), ..., (s_N, e_N)] och beräknar mellanrummen mellan de angivna intervallen. Alla element s_i och e_i är heltal och de kommer i stigande ordning, d.v.s. det gäller alltid att $s_k < e_k$ och att $e_k <= s_{k+1}$. Exempel: interval([(1, 3), (5, 8), (10, 12)]) ger [(3, 5), (8, 10)]. I indatat har vi en lista med intervallen 1-3, 5-8 och 10-12. Som resultat får vi en lista med mellanrummen, d.v.s. 3-5 och 8-10. Ändpunkterna ingår i intervallen.

Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `interval_r`) och en som arbetar enligt en iterativ modell (kallad `interval_i`). Du får inte använda inbyggda funktioner eller metoder som behandlar hela listor. Du får inte heller använda listbyggare (eng. *list comprehensions*). Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera indata. Funktionerna ska fungera exakt likadant, bortsett från att de ska ha olika lösningsmodeller. Här är några exempel:

```
>>> interval_r([(1, 3), (5, 8), (10, 12)])
[(3, 5), (8, 10)]

>>> interval_i([(10, 13), (16, 19), (19, 20), (25, 33)])
[(13, 16), (20, 25)]
```

De båda funktionerna `interval_r` och `interval_i` ger vardera 2.5 poäng, och det är möjligt att få poäng på dem oberoende av varandra. Definiera båda funktionerna i samma fil.

Uppgift 3

Deluppgift 3a (3p)

Att derivera funktioner är en viktig del av den matematiska analysen. En funktion kan antingen deriveras symboliskt eller numeriskt. Du har huvudsakligen lärt dig att derivera funktioner symboliskt. Ett sätt att numeriskt uppskatta derivatan *f'(x)* av en funktion *f(x)* är att räkna ut lutningen på en linje mellan två punkter kring *x* enligt *(f(x+h) - f(x-h)) / 2h*. Skriv en högre ordningens funktion `derivate(f, h)` som

- tar en funktion *f(x)* och ett tal *h*, och
- returnerar en funktion som tar ett argument och beräknar derivatan *f'(x)* numeriskt enligt ovan.

Deluppgift 3b (2p)

Skriv **ett uttryck** som använder `derivate` för att räkna ut derivatan av $7x^3 + 4x^2$ för x=8 och h=0.001.

Uppgift 4

Skriv en funktion `filter(seq, pred)` som tar (1) en lista som kan innehålla godtyckligt nästlade listor och (2) ett predikat (dvs. en funktion som tar ett argument och returnerar ett sanningsvärde). Funktionen ska returnera en lista med samma liststruktur men som endast innehåller de enkla element som predikatet returnerar ett sant värde för. Tänk på att ett element kan vara en godtycklig lista av listor. Funktionen ska vara icke-destruktiv. Det går bra att lösa uppgiften rekursivt eller iterativt. Tänk som vanligt på att dokumentera funktionen på lämpligt sätt.

```
>>> filter([1, 2, 3], lambda x: x > 1)
[2, 3]

>>> filter([[1], [2], [3]], lambda x: x > 1)
[[], [2], [3]]

>>> filter([[1, [2, [3]], 2, [7], [2, [3, 2, 1]]]], lambda x: x % 2 == 0)
[[[2, []], 2, [], [2, [2]]]]
```

Uppgift 5

En *delsekvens* innehåller noll eller flera element från en annan sekvens, i samma ordning men eventuellt med hopp. Sekvensen ”ABC”, har till exempel delsekvenserna ””, ”A”, ”B”, ”C”, ”AB”, ”AC”, ”BC” och ”ABC”.

Längsta gemensamma delsekvensen (en. longest common subsequence) är ett klassiskt problem inom datavetenskapen: Givet två sekvenser, hitta den längsta sekvens som är delsekvens av *båda* sekvenserna. Sekvenserna ”GAC” och ”AGCAT” har t.ex. tre gemensamma delsekvenser av längd 2 (”GA”, ”GC” och ”AC”), och inga som är längre.

Skriv en funktion `llcs(seq1, seq2)` som tar två sekvenser och returnerar *längden* på den längsta gemensamma delsekvensen. Funktionen behöver inte vara minnes- eller tidseffektiv. Det är till exempel tillåtet att prova alla alternativ.

Ett sätt är följande.

- Om någon av sekvenserna är tom, har de inga gemensamma element och *llcs* är 0.
- Om sekvenserna annars börjar med *samma* element, kan detta element tas med i längsta LCS utan att detta kan förhindra några senare matchningar mellan sekvenserna. Det elementet ska alltså tas med i beräkningen, och därefter måste man också beräkna längsta LCS för *resten* av båda sekvenserna.
- Annars måste man ta hänsyn till två möjligheter: Det bästa kan vara att hoppa över ett element i `seq1` (och sedan beräkna *llcs* för de resulterande sekvenserna) eller att hoppa över ett element i `seq2`. Båda alternativen behöver i så fall testas och det bästa (längsta) resultatet returneras.

```
>>> llcs('GAC', 'AGCAT')
2
>>> llcs('GAAAC', 'AGACATA')
4
>>> llcs('JAVA', 'PYTHON')
0
```

Uppgift 6

Att skapa abstrakta datatyper är en vanlig uppgift för programmerare. Den här uppgiften handlar om att implementera en abstrakt datatyp som vi kallar ring. Ett ringobjekt kan innehålla ett godtyckligt antal element som är cykliskt ordnade, där ett av dem benämns toppelement. Vi vill ha följande primitiva funktioner (primitiver):

- `make_ring(elements)` : skapa och returnera en ny ring från en lista med element där första elementet i `elements` är toppelement i ringen.
- `top(ring)` : returnera toppelementet i ringen `ring`.
- `left_rotate(ring)`: returnera en ny ring som motsvarar den existerande ringen roterad ett steg åt vänster, utan att modifiera den existerande ringen.
- `right_rotate(ring)`: returnera en ny ring som motsvarar den existerande ringen roterad ett steg åt vänster, utan att modifiera den existerande ringen.
- `left_rotate_in(ring)`: rotera elementen i `ring` ett steg åt vänster. Inga nya objekt ska skapas eller returneras, utan ringen ska modifieras på plats.
- `right_rotate_in(ring)`: rotera elementen i `ring` ett steg åt höger. Inga nya objekt ska skapas eller returneras, utan ringen ska modifieras på plats.

Här följer några exempel på hur primitiverna ska fungera:

```
>>> ring1 = make_ring([1, 2, 3])
>>> top(ring1)
1

>>> top(left_rotate(ring1))
2

>>> top(right_rotate(ring1))
3

>>> top(left_rotate(left_rotate(left_rotate(ring1))))
1

>>> ring2 = make_ring(['a', 'b', 'c'])
>>> top(ring2)
'a'

>>> left_rotate_in(ring2)
>>> top(ring2)
'b'

>>> right_rotate_in(ring2)
>>> right_rotate_in(ring2)
>>> top(ring2)
'c'
```

Välj en lämplig representation för objekt av typen ring och definiera primitiverna. Din lösning ska stödja följande operationer och ska implementeras från grunden med listor och/eller dictionaries samt operationer på dessa. Lösningen ska vara funktionell om inte annat är angivet och dokumentation är viktigt. För fulla poäng ska datatypens representation vara dokumenterad.

`make_ring` och `top` ger 0,5 poäng, övriga primitiver ger 1 poäng vardera.