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 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 lämpligt dokumenterad i enlighet med uppgiftsspecifikationen.
- 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 12 januari 2016 kl 8-13

Uppgift 1

Att beräkna π med många decimaler är ett nöje som många ägnat sig åt genom tiderna. 1789 beräknade en slovensk matematiker Jurij Vega 140 decimaler på pi, även om bara de första 126 visade sig vara korrekta. Detta var världsrekord i mer än 50 år. Den matematiska analysen har gett upphov till serier och iterationer för π:s exakta värde som i princip gör det möjligt att beräkna talet med önskad precision. Srinivasa Ramanujan upptäckte en mängd oändliga serier för π, exempelvis den snabbt konvergerande

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Din uppgift är att implementera en funktion `pi(k)` som beräknar π enligt Ramanujans formel för det givna värdet på `k`.

```
>>> pi(0)
3.1415927300133055

>>> pi(2)
3.1415926535897887
```

Uppgift 2

Skriv två varianter av en funktion som tar en lista med strängar och returnerar en lista med samma strängar fast omvända. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `reverseeach_r`) och en som arbetar enligt en iterativ modell (kallad `reverseeach_i`). Båda funktionerna ska behandla ett element i taget, vilket innebär att du inte får 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:

```
>>> reverseeach_r(["paris", "i", "sirap”])
["sirap", "i", "paris"]

>>> reverseeach_i(["hanna", "leo"])
["annah", "oel"]
```

De båda funktionerna `reverseeach_r` och `reverseeach_i` ger vardera 2.5 poäng.

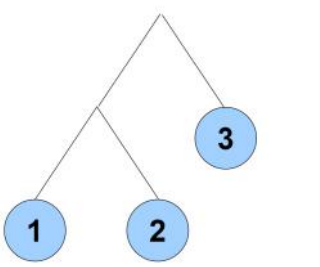
Uppgift 3

Låt oss definiera ett binärt träd enligt följande:

1. Den tomma listan `[]` är ett binärt träd (det tomma trädet).
2. Ett enda värde `x` är ett binärt träd (ett löv).
3. En lista bestående av två icke-tomma binära träd `[left, right]` är ett binärt träd.

Deluppgift 3a (3p)

Implementera en högre-ordningens funktion `tree_apply(fn, tree)` som tar en binär funktion `fn` och ett träd `tree` representerat enligt ovan och applicerar `fn` rekursivt på varje inre nod, från löven och uppåt, så att endast ett enda värde returneras. I exemplet nedan beräknas först `fn(1,2)` och sedan `fn(fn(1,2), 3)`.



För att få full poäng ska du skapa lämpliga funktioner för att manipulera träddatastrukturen.

Exempel:

```
>>> tree_apply(lambda x, y: x+y, [[1, 2], 3])
6
```

Deluppgift 3b (2p)

Använd `tree_apply` för att hitta det nästa minsta elementet i ett binärt träd med minst två löv.

Uppgift 4

Att hitta den längsta gemensamma delsekvensen (en. Longest Common Subsequence) givet två strängar är ofta användbart. Det används t.ex. av diff och Git. En delsekvens är en godtycklig delmängd av den ursprungliga sekvensen, det enda kravet är att elementen behåller sin inbördes ordning. Sekvensen ”abc” har 6 delsekvenser: ”ab”, ”ac”, ”bc”, ”a”, ”b” och ”c”.

Implementera en funktion `lcs(s1, s2)` som returnerar längden på den längsta gemensamma delsekvensen för strängarna `s1` och `s2`. Funktionen behöver inte vara minnes- eller tidseffektiv. Du får anta att strängarna har som mest 100 tecken om du så önskar.

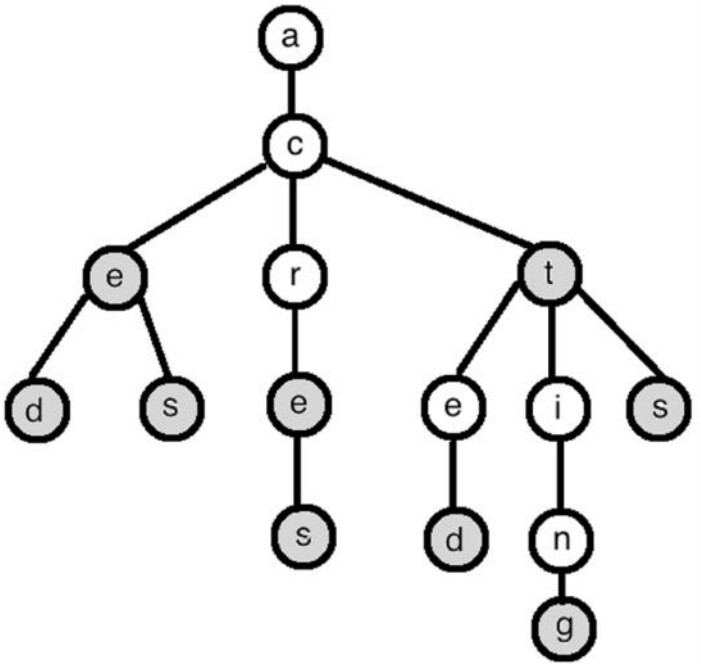
```
>>> lcs("abc", "cba")
1

>>> lcs("abc", "aco")
2

>>> lcs("lovxxelyxxxxx", "xxxxxxxlovely")
7
```

Uppgift 5

En Trie är en effektiv datastruktur för att lagrar en mängd strängar som bland annat används för att snabbt hitta alla möjliga fortsättningar på en sträng. En Trie är ett träd där varje nod motsvarar en bokstav och en väg från roten motsvarar ett ord eller en del av ett ord. Effektiviteten uppnås genom att gemensamma prefix endast lagras en gång. En Trie som lagrar orden ace, aced, aces, acre, acres, act, acted, acting och acts ser ut så här (grå noder representerar att ett ord slutar där):



Din uppgift är att föreslå och implementera en datastruktur för att representera en Trie. Datastrukturen ska klara följande:

1. Skapa en tom Trie, `create_trie()`.
2. Addera ett ord till en Trie, `add_word(trie, word)`. Du får själv välja om du vill returnera en ny modifierad Trie eller att direkt modifera `trie`.
3. Avgöra om ett ord finns med i en Trie, `word_in_trie(trie, word)`. Funktionen ska returnera `True` om och endast om ordet `word` finns i `trie`. Givet exemplet ovan skulle `word_in_trie` returnera `True` för ace, aced, aces, acre, acres, act, acted, acting och acts.
4. Hitta alla ord som matchar ett prefix i en Trie, `find_all_matches(trie, prefix)`. Funktionen ska returnera en lista med alla ord in `trie` som börjar på `prefix`. Givet exemplet ovan skulle `find_all_matches` för `"ace"` returnera `["ace", "aced", "aces"]`.

Uppgiften ger 3 poäng för `add_word` och 1 poäng vardera för `word_in_trie` och `find_all_matches`.

Uppgift 6

Uppgift 6a (2p)

Implementera en funktion `permutations(seq)` som returnerar en lista med alla permutationer av listan `seq`. En permutation är en ordning av elementen i listan. Ordningen mellan permutationerna är godtycklig och behöver inte vara exakt som nedan. Inga inbyggda funktioner är tillåtna. Listbyggare är tillåtna.

```
>>> permutations([1, 2, 3])
[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

Uppgift 6b (3p)

Generalisera `permutations(seq)` till att även klara listor som innehåller listor i godtyckligt antal nivåer. Det betyder att funktionen ska permutera varje underlista och se varje permutation som ett eget element i den övre listan. Ordningen mellan permutationerna är godtycklig och behöver inte vara exakt som nedan. Inga inbyggda funktioner är tillåtna. Listbyggare är tillåtna.

```
>>> permutations([1, [2], 3])
[[1, [2], 3], [[2], 1, 3], [[2], 3, 1], [1, 3, [2]], [3, 1, [2]], [3, [2],
1]]
```

```
>> permutations([[1, 2], 3])
[[[1, 2], 3], [3, [1, 2]], [[2, 1], 3], [3, [2, 1]]]
```