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 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.

Uppgifter - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Onsdag 17 augusti 2016 kl 14-19

Uppgift 1

Pekare är vanliga inom programmering för att hänvisa till data som finns på en annan plats, ofta i datorns minne. Pekare kan användas till mycket, t.ex. för att komprimera en text. Istället för att lagra varje tecken kan texten innehålla pekare till textstycken. Din uppgift är att expandera en text som innehåller pekare. Skriv en funktion `expand(mem, msg)` som tar en lista med strängar (`mem`) och en lista med strängar och heltal (`msg`) och returnerar en sträng där varje heltal i `msg` har bytts ut mot motsvarande sträng i `mem`. Du kan anta att alla pekare är giltiga. Nedan finns några exempel.

```
>>> mem = [' ', 'att', 'lycka', 'tenta', 'till', 'på', 'är']
>>> expand(mem, [2, 0, 6, 0, 1, 0, 3])
'lycka är att tenta'
>>> expand(mem, [2, 0, 4, 0, 5, 0, 3, 'n'])
'lycka till på tentan'
```

Uppgift 2

Skriv en funktion som tar två listor och skapar en ny lista genom att växelvis ta element från de två listorna. Börja alltid med ett element från den första listan om möjligt. Alla element från båda listorna ska vara med. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `interleave_r`) och en som arbetar enligt en iterativ modell (kallad `interleave_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:

```
>>> interleave_r([1, 3, 5], [2, 4, 6])
[1, 2, 3, 4, 5, 6]
>>> interleave_i(['a'], [1, 2, 3])
['a', 1, 2, 3]
>>> interleave_i(['a', 'b'], [1, 2, 3])
['a', 1, 'b', 2, 3]
```

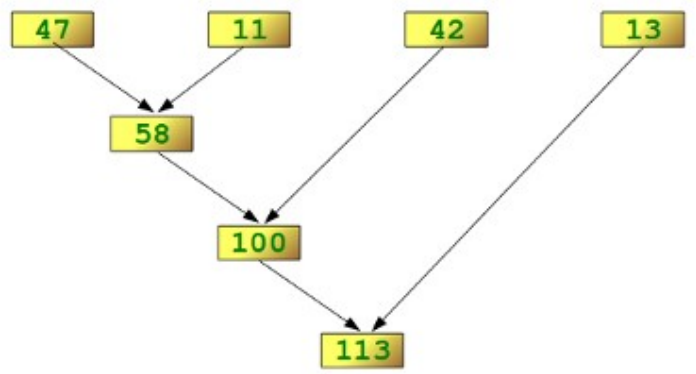
De båda funktionerna `interleave_r` och `interleave_i` ger vardera 2.5 poäng, och det är möjligt att få poäng på dem oberoende av varandra.

Uppgift 3

Deluppgift 3a (3p)

Att summera alla elementen i en sekvens är ett sätt att reducera en sekvens till ett värde. Att multiplicera alla elementen är ett annat. Skriv en funktion `reduce(f, seq)` som tar en binär funktion `f` och en sekvens `seq` och reducerar alla elementen i sekvensen genom att successivt anropa `f`. Du kan anta att `seq` innehåller minst ett element.

Till exempel ger `reduce(lambda x,y: x+y, [47,11,42,13])` följande beräkningar:




```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```

Deluppgift 3b (2p)

Skriv en högre ordningens funktion `reduce_if(f, p)` som tar en binär funktion `f` och en unär funktion `p` (en funktion som tar ett argument) och returnerar en funktion som tar en sekvens som indata och reducerar alla elementen i sekvensen som uppfyller `p` enligt `f`. Exemplet nedan adderar alla udda element i en sekvens, i det här fallet sekvensen `[0, 1, 2, 3, 4]`.

```
>>> reduce_if(lambda x, y: x+y, lambda x: x%2==1)(range(5))
4
```

Uppgift 4

Skriv en funktion `sum_all` som tar en lista som kan innehålla godtyckligt nästlade listor och returnerar summan av alla tal i listorna. Funktionen ska vara icke-destruktiv. Det går bra att lösa uppgiften rekursivt eller iterativt. Tänk som vanligt på att dokumentera funktionen på lämpligt sätt.

```
>>> sum_all([1, 1, 1])
3
>>> sum_all([[1], [1], [1]])
3
>>> sum_all([[2, 1, 2], [1], [1, 1, 1, 1, 2]])
12
>>> sum_all([[2, 1, [2]], [1], [1, 1, [2, 2, 2, 2], 1, 1, 2]])
20
```

Uppgift 5

Vi vill ha en abstrakt datatyp för att representera dubbelriktade köer. En dubbelriktad kö består av ett godtyckligt antal *element* där det är möjligt att plocka ut det första eller det sista elementet samt att lägga till nya element först eller sist i kön. Välj en lämplig representation och definiera nedanstående primitiva funktioner. Alla funktioner ska vara väl dokumenterade.

`make_deque()`: Returnerar en ny tom kö.

`length(deque)`: Returnerar längden på kön `deque`.

`front(deque)`: Returnerar första elementet i kön `deque`.

`back(deque)`: Returnerar sista elementet i kön `deque`.

`push_front_d(deque, elt)`: Lägg till `elt` först i kön `deque` destruktivt och returnera kön.

`pop_front_d(deque)`: Ta bort första elementet i kön `deque` destruktivt och returnera kön.

`push_back_d(deque, elt)`: Lägg till `elt` sist i kön `deque` destruktivt och returnera kön.

`pop_back_d(deque)`: Ta bort sista elementet i kön `deque` destruktivt och returnera kön.

`push_front_f(deque, elt)`: Lägg till `elt` först i kön `deque` funktionellt och returnera kön.

`pop_front_f(deque)`: Ta bort första elementet i kön `deque` funktionellt och returnera kön.

`push_back_f(deque, elt)`: Lägg till `elt` sist i kön `deque` funktionellt och returnera kön.

`pop_back_f(deque)`: Ta bort sista elementet i kön `deque` funktionellt och returnera kön.

De fyra första funktionerna ger tillsammans 1p. De destruktiva respektive funktionella funktionerna ger 2p vardera.

Här följer några exempel på hur primitiverna ska fungera. Tänk på att utdata kan skilja sig något beroende på ditt val av representation och implementation.

```
>>> q = make_deque()
>>> q1 = push_front_d(q, 1)
>>> length(q)
1
>>> length(q1)
1
>>> push_back_d(q, 2)
[1, 2]
>>> front(q)
1
>>> back(q)
2
>>> pop_front_d(q)
[2]
>>> front(q)
2
>>> pop_back_d(q)
[]
```

```
>>> q = make_deque()
>>> q1 = push_front_f(q, 1)
>>> length(q)
0
>>> length(q1)
1
>>> q2 = push_back_f(q1, 2)
>>> front(q2)
1
>>> back(q2)
2
>>> q3 = pop_front_f(q2)
>>> front(q3)
2
>>> pop_back_d(q3)
[]
```

Uppgift 6

Att hitta den kortaste vägen i en graf är ett vanligt förekommande problem inom datavetenskap. Beroende på hur grafen ser ut finns det olika sätt att räkna ut den kortaste vägen. Ett av de enklaste fallen är när alla bågar i grafen är lika långa, vilket till exempel är fallet i ett regelbundet rutnät där man bara får gå, ner, höger eller vänster. Den enklaste algoritmen för att hitta den kortaste vägen i det fallet är bredden först-sökning. Algoritmen ser ut så här:

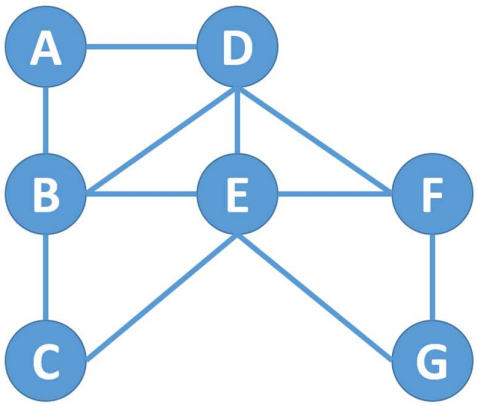
Bredden först-sökning:

1. Låt Q vara en kö.
2. Lägg in paret (startnod, 0) i kön där 0 är avståndet från startnoden.
3. Så länge det finns minst en nod kvar i kön gör:
4. Ta ut den första noden N och dess avstånd D från kön.
5. Markera N som besökt.
6. Om N är målnoden returnera D.
7. För varje barn B till N som inte redan besökts:
8. Lägg till paret (B, D+1) sist i kön.

Din uppgift är att implementera en funktion `bfs(graph, start, goal)` som använder bredden först-sökning för att hitta den kortaste vägen från `start` till `goal` och returnerar dess längd. För ett exempel se nedan. Till din hjälp har du färdig kod för att skapa och manipulera grafer (`graph.py`). Funktionen ska inte vara destruktiv och alla funktioner ska vara väl dokumenterade.

Om man kör algoritmen med A som start och C som mål på grafen nedan kommer följande hända:

1. (A, 0) läggs till kön.
2. (A, 0) plockas ur kön, markeras som besökt och dess obesökta barn (B, 1) och (D, 1) läggs till.
3. (B, 1) plockas ur kön, markeras som besökt och dess obesökta barn (C, 2), (D, 2) och (E, 2) läggs till.
4. (D, 1) plockas ur kön, markeras som besökt och dess obesökta barn (E, 2) och (F, 2) läggs till.
5. (C, 2) plockas ur kön, markeras som besökt och avståndet 2 returneras då C är målet.



Ytterligare exempel: Den kortaste vägen från A till B är 1 steg, från A till E 2 steg (via B eller D) och från A till G 3 steg (t.ex. via D och F).

```
>>> g = make_undirected_graph(['A', 'B', 'C', 'D', 'E', 'F', 'G'], [('A',
'B'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'E'), ('D',
'E'), ('D', 'F'), ('E', 'F'), ('E', 'G'), ('F', 'G')])
```

```
>>> bfs(g, 'A', 'A')
0
```

```
>>> bfs(g, 'A', 'C')
2
```

```
>>> bfs(g, 'A', 'G')
3
```