Instruktioner för datortentamen Torsdag 2019-04-24

TDDE24 Funktionell och imperativ programmering del 2

Hjälpmedel

Följande fysiska hjälpmedel är tillåtna:

- Exakt en valfri bok, t.ex. den rekommenderade kursboken. Boken får ha anteckningar, men inga lösa lappar som inte hör till bokens innehåll.
- 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. (Lösa tomma papper tillhandahålls av tentamensvakterna.)

Du har också tillgång till https://docs.python.org/3/library via startmenyn.

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

Genomförande

Under datortentan kommer du att arbeta i en begränsad och övervakad datormiljö. Med hjälp av en särskild **tentaklient** skickar du in dina svar. Varje uppgift kan **skickas in en enda gång**. Man kan alltså inte komplettera.

Vid denna omtenta kommer poäng *inte* att sättas under tentans gång. Poäng sätts i efterhand.

Du kan även använda tentaklienten för att **ställa frågor** till jourhavande, som vid behov kan vidarebefordra till examinator. **Ställ alla tentarelaterade frågor via klienten.** Besök i skrivsalen kommer endast att ske vid allvarligare tekniska problem.

För att underlätta rättningen:

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgifter, poäng och betyg

Datortentan består av totalt 6 uppgifter som vardera kan ge maximalt 5 poäng. Alla uppgifter är utvalda för att testa olika aspekter av kursinnehållet. De är också tänkta att ha olika svårighetsgrad och är ordnade så att svårigheten oftast ökar mot slutet. 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.

För betyg 3, 4 och 5 krävs minst 12, 19 respektive 25 poäng.

Rättningskriterier

Lösningar som bryter mot följande allmänna kriterier kommer att få poängavdrag.

- Programkod ska vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte gav felaktig kod! Poängavdrag ges vid icke körbar kod, eller *delvis* körbar kod (som kraschar för vissa fall).
- 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 under kursens laborationer. Detta inkluderar att skriva svenska eller engelska **docstrings** för **samtliga funktioner som definieras med `def` på "toppnivån"**, även hjälpfunktioner som man själv inför.

  Det är *inte* ett absolut krav att skriva docstrings för lambda-uttryck eller för nästlade funktioner som definieras inuti andra funktioner. Tänk dock på att koden ändå behöver vara tillräckligt väldokumenterad för att vara lätt att förstå.

  Specialtaggar som `:param:` krävs ej.
- Lösningen ska följa de **specifika regler och villkor** 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 för alla tänkbara indata som följer uppställningen i frågan.

  Man behöver dock normalt **inte felkontrollera** indata, utom när uppgiften särskilt anger det. Funktioner måste alltså ge *korrekta svar för korrekta indata* enligt uppgiften, men om inte annat anges får de *krascha eller ge felaktiga svar för felaktiga indata* (såsom när en funktion som bara ska hantera heltal ges en sträng som parameter).
- Den implementerade lösningen ska kunna köras inom **rimliga tidsramar**. Till exempel accepteras inte en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa. Eventuella undantag anges uttryckligen i frågan.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Eventuella testfall visar normalt returnerade värden, inte värden som funktionerna själva har skrivit ut.

De **testfall** vi ger i tentan är bara exempel, och är främst till för att förtydliga vad vi menar med beskrivningstexten. Det finns inga garantier för att en lösning som uppfyller de här testfallen är helt korrekt. Skapa gärna egna tester som kan täcka ytterligare fall.

Testfall *får* men *behöver inte* lämnas in.

Uppgift 1

Många mobilabonnemang tillåter en begränsad mängd nedladdning per månad. I den första uppgiften vill du gärna titta på din historiska dataanvändning för att se vilket abonnemang som blir bäst för dig. Din mobil är snäll nog att lämna ut dessa data som en Pythonlista med följande innehåll:

- Ett strikt positivt tal (`3` eller `324.25`) innebär att du laddade ner så många MB under en viss dag. Dagar då inget laddades ner *kan* finnas med i listan (som konstanten `0`), men kan också utelämnas helt.
- Konstanten `None` innebär att en ny månad påbörjas. Den förra månaden tar alltså slut och en ny påbörjas.

Som hjälp tittar vi särskilt på dessa randfall:

- Listan kan sakna `None`. Då gäller den alltså en enda månad. Om listan är helt tom gäller den en enda månad då inget laddades ned.
- Listan kan innehålla `None` som första element. Då börjar den alltså med en månad då inget laddades ned. Även sådana månader "räknas" som månader med konsumtion 0 och ska finnas med i resultatet. Motsvarande gäller sista elementet i listan.
- Listan kan innehålla `None` flera gånger i rad. Även detta innebär att det finns månader då inget laddades ner.

Skriv en funktion `datause(seq)` som tar in indata på detta format och returnerar en lista med det totala antalet konsumerade MB för varje månad som representeras i indata, i tur och ordning, samt ett sista värde som är antalet konsumerade MB i den *månad* då du använde mobilen mest. Alla månader ska finnas med i listan i korrekt ordning, även de då inget laddades ned enligt randfallen ovan.

Exempel – skapa gärna egna tester med inspiration av villkoren ovan!

- `assert datause([12, None]) == [12, 0, 12]`
- `assert datause([1, 5, 17, 2]) == [25, 25]`
- `assert datause([None, 1, 12, 5, None, 22, 21, None, 2]) == [0, 18, 43, 2, 43]`

Bra att tänka på

- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både listbyggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Inskickade parametrar får inte modifieras.
- Dokumentera funktioner med docstrings! Detta är sista varningen.
- Testa efter varje ändring, även när du *tror* att ändringen var harmlös.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgift 2

MergeSort är en känd sorteringsalgoritm som baseras på principen *söndra och härska*: Man delar upp listor i delar, löser de resulterande delproblemen genom att sortera de mindre listorna var för sig, och slår sedan ihop de resulterande listorna. Det sista steget underlättas då kraftigt av att de listor som ska slås ihop redan är sorterade.

Du ska nu skriva en funktion som tar *två* sorterade listor `s1` och `s2`, och som steg för steg konstruerar en enda sorterad lista som innehåller alla element från dessa. Detta ska ske genom att man i varje steg plockar *ett* element från en lämplig lista: `s1` eller `s2`. Funktionen ska finnas i två varianter:

- En som arbetar enligt rekursiv lösningsmodell: `merge_r(s1,s2)`
- En som arbetar enligt iterativ lösningsmodell: `merge_i(s1,s2)`

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

Ytterligare villkor:

- Varje lista kan ha godtycklig längd.
- Listor kan innehålla godtyckliga element, inklusive identiska element. Vi garanterar att alla par av element kan jämföras med de vanliga jämförelseoperatorerna.
- Du ska inte använda existerande sorteringsalgoritmer och inte heller implementera egna sorteringsalgoritmer, utan plocka enskilda element i rätt ordning enligt ovan.
- Du får inte använda listbyggare (eng. list comprehensions) eller inbyggda funktioner som behandlar alla element i hela listor. Däremot är givetvis `len()` tillåten.
- Lösningarna får inte modifiera listor som skickas in.

Båda funktionerna ska bl.a. klara följande **tester**, där `merge_x` är antingen `merge_i` eller `merge_r`:

- `assert merge_x([], [1]) == [1]`
- `assert merge_x([1, 2, 8, 13], [3, 5, 21]) == [1, 2, 3, 5, 8, 13, 21]`
- `assert merge_x(['a', 'c'], ['b']) == ['a', 'b', 'c']`

Skapa gärna egna tester med inspiration av villkoren ovan!

Uppgift 3

Skriv en funktion `sum_nth(seq)` som tar en lista som kan innehålla godtyckligt nästlade listor samt ett heltal `n` > 0. Funktionen ska returnera summan av vart `n`:te heltal i listan med start i det *första* elementet. Med "vart `n`:te" menar vi i den ordning som elementen påträffas om man traverserar alla listans element i tur och ordning, så som visas i exemplen nedan.

- Man får anta att alla element i `seq` är antingen heltal eller listor (som också bara består av heltal eller listor (som också...)).
- Man får anta att det finns minst ett heltal någonstans i den nästlade listan.
- Funktionen ska vara icke-destruktiv.
- Det är tillåtet att lösa uppgiften rekursivt eller iterativt.

Exempel:

- `assert sum_nth([1, 2, 3, 4, 5], 1) == 1+2+3+4+5`
- `assert sum_nth([1, [2, 3], 4, [5]], 2) == 1+3+5`
- `assert sum_nth([1, [[2, [3]], 4], [5]], 3) == 1+4`

Uppgift 4

Deluppgift 4a (3p)

Curryfiering (engelska **currying**) är ibland användbart i t.ex. funktionell programmering. Något förenklat kan man se detta som att man har en funktion som tar flera argument, och att man nu vill "fixera" värdet på det första argumentet så att man slipper ange det värdet. Man vill med hjälp av curryfiering "ta bort" det första argumentet så att man bara behöver ange de övriga argumenten.

Detta kan man göra med en högre ordningens funktion som vi kan kalla `curry`.

Anta till exempel att du redan har funktionen `sum_nth(seq, n)` från förra uppgiften. Denna funktion tar som synes två parametrar, en sekvens och ett heltal. Genom att anropa `curry(sum_nth, [1, 2, 3, 4, 5])` ska man istället få fram en funktion som bara tar ett heltal:

- `f = curry(sum_nth, [1, 2, 3, 4, 5])`
- `assert f(1) == 1+2+3+4+5`
- `assert f(2) == 1+3+5`

Din uppgift är att skriva den högre ordningens funktionen `curry(f, v)` som binder första argumentet i funktionen `f`, som har två argument, till värdet `v`. Detta görs alltså genom att returnera en ny funktion med *ett* argument, som vid anrop ger samma resultat som att räkna ut `f(v, x)`.

Deluppgift 4b (2p)

I Pythons standardbibliotek finns funktionen `math.pow(x,y)` som returnerar x^y där `x` och `y` är flyttal.

Importera denna funktion och använd sedan `curry` för att definiera funktionen `pow2(z)` som returnerar 2^z. Detta ska definieras med hjälp av ett enda uttryck och utan att använda `def`:

- `pow2 = ...`

Du ska inte definiera egna hjälpfunktioner utöver `curry`.

Exempel:

- `assert pow2(3) == 8`
- `assert pow2(8) == 256`

Uppgift 5

I denna uppgift ska du hitta längden av en *längsta växande delsekvens* i en sekvens som innehåller godtyckliga tal.

- Med "delsekvens" menar vi ett urval av talen i den ursprungliga sekvensen, där talen står i samma ordning men vissa tal tal (noll eller flera) kan ha "hoppats över". Givet `[1,2,3,4,5]` är `[1,2,4]` och `[]` delsekvenser, men inte `[1,4,2]` eller `[1,4,8]`.
- Med "växande" menar vi att varje tal är minst lika stort som det föregående. Givet `[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]` är `[0,8,4]` en delsekvens men inte växande. Däremot är `[1,3,7]` och `[0,8,14,15]` växande delsekvenser.
- Givet förra sekvensen är $[0, 2, 6, 9, 11, 15]$ och $[0, 4, 6, 9, 11, 15]$ exempel på längsta växande delsekvenser: Båda har 6 tal, och det finns ingen växande delsekvens som är längre.

Skapa funktionen `lvd(seq)` (börjar med bokstaven L, inte siffran 1) som tar en sekvens av tal och returnerar längden hos de längsta växande delsekvenserna.

Att tänka på: Man kan inte se på ett visst tal om det kommer att vara med i någon *längsta* växande delsekvens eller inte. Poängen med denna uppgift är att man behöver testa olika alternativ.

Ytterligare villkor:

- Funktionen behöver *inte* vara minnes- eller tidseffektiv. Det är till exempel acceptabelt att lösningen provar alla alternativ.
- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både list-byggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Funktionen ska vara funktionell i bemärkelsen att parametrar inte får ändras.

Exempel:

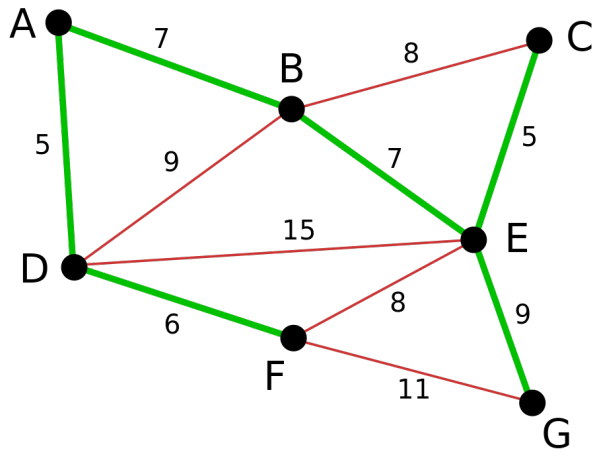
- `assert lvd([0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]) == 6`
- `assert lvd([1, 2, 4, 100, 1, 2, 3, 4, 5, 6]) == 7`

Uppgift 6

Under första halvan av 1980-talet började konstruktionen av SUNET, Swedish University computer NETwork, som skulle koppla ihop Sveriges universitet med hjälp av IP-nätverk över ett antal inhyrda telelinjer på 64 kbps. Så småningom började man behöva betydligt mer bandbredd, och nya nätverk började byggas.

Anta nu att några universitet ska kopplas ihop med nya fysiska kommunikationsförbindelser. För att forskare på två universitet ska kunna kommunicera krävs antingen att de två universiteten har en *direkt* förbindelse eller att de kan prata *indirekt* via ett annat universitet.

Då man dessutom vill minimera kostnaden att bygga nätet vill man koppla samman alla universitet så att den totala kostnaden minimeras. Som underlag kan vi då skapa en graf som innehåller kostnaderna för ett antal olika tänkbara förbindelser. Om vi t.ex. väljer att skapa en direkt förbindelse mellan D och E nedan, har detta en kostnad av 15.



Givet en sådan graf vill vi hitta ett **minsta uppspännande träd** (eng. *minimum spanning tree*) i grafen av alla möjliga förbindelser, alltså ett träd som når alla noder samtidigt som summan av bågkostnaderna är minimal. De tjocka gröna bågarna i grafen ovan ingår i ett sådant träd. Här pratar t.ex. D med E via vägen D-A-B-E.

Kruskals algoritm

En vanlig algoritm för att hitta ett minsta uppspännande träd är Kruskals algoritm, som utgår från en oriktad graf `G` representerad som en lista med bågar och deras vikter (kostnader).

Grafen ovan kan t.ex. representeras som `[('A', 'B', 7), ('A', 'D', 5), ('B', 'C', 8), ('B', 'D', 9), ('B','E', 7), ('C', 'E', 5), ('D', 'E', 15), ('D', 'F', 6), ('E', 'F', 8), ('E','G', 9), ('F', 'G', 11)]`.

(Fortsättning på nästa sida)

Kruskals algoritm i pseudokod:

- Låt `T` vara en tom graf (som så småningom ska bli ett minsta uppspännande träd).
- Upprepa så länge minst en båge i `G` inte är markerad som "förbrukad":
  - Låt `v` vara den *billigaste* bågen av de bågar i `G` som inte märkts som "förbrukade"
  - Märk `v` som "förbrukad"
  - Om man kan addera `v` till trädet `T` utan att det bildas en cykel, så gör detta. Annars (om en cykel skulle ha bildats) var bågen onödig, och `v` ska inte finnas med i trädet.

Att göra

Din uppgift är att implementera en funktion `mst(edges)` som givet en lista med bågar använder Kruskals algoritm för att hitta ett minsta uppspännande träd. Som exemplifierat ovan är varje båge en tupel `(from, to, weight)` där `from` och `to` är namn på noder medan `weight` är kostnaden (vikten) för bågen.

Funktionen ska returnera en tupel vars första element är totalkostnaden för det genererade trädet och vars andra element är en mängd (`set`) innehållande de bågar som ingår i trädet.

Funktionen ska inte modifiera sina argument. Den ska vara väl dokumenterad så att du förklarar vad delar av din kod gör.

Exempel (klipp och klistra):

- `assert mst([('A', 'B', 7), ('A', 'D', 5), ('B', 'C', 8), ('B', 'D', 9), ('B','E', 7), ('C', 'E', 5), ('D', 'E', 15), ('D', 'F', 6), ('E', 'F', 8), ('E','G', 9), ('F', 'G', 11)]) == (39, {('A', 'D', 5), ('C', 'E', 5), ('D', 'F', 6), ('A', 'B', 7), ('B','E', 7), ('E', 'G', 9)}`

Tips

Ett sätt att se om det blir en cykel om man adderar `v` till `T`, är att hela tiden hålla reda på exakt vilken mängd av noder som just nu går att nå från varje annan nod.

- Till en början kan man från noden `v` bara nå just `v`, då `T` inte innehåller några bågar som vi kan använda för att nå andra noder.
- När man vill veta om man *borde* addera bågen `(from, to, weight)` till `T` kan man enkelt testa om det redan går att nå `from` från `to`. I så fall skulle en cykel skapas.
- Om man faktiskt *adderar* bågen `(from, to, weight)` till `T`, kommer detta inte bara att koppla ihop två noder utan potentiellt två delgrafer:
  - Från *varje* nod som redan kunde nås från `from` kan man nu *också* nå *alla* noder som redan kunde nås från `to`.
  - Från *varje* nod som redan kunde nås från `to` kan man nu *också* nå *alla* noder som redan kunde nås från `from`.