Datortentamen 2021-08-17

TDDE24 Funktionell och imperativ programmering del 2

Distanstentamen

Detta är en *distanstentamen* i TDDE24. Läs genom instruktionerna noga för att förstå hur tentan ska genomföras.

Distanstentamen: Tillåtna och otillåtna hjälpmedel

- Användning av resurser på Internet, inklusive kursens websidor, är uttryckligen *tillåten*. Som tidigare kan websidor under `https://docs.python.org/3/` vara användbara. Detta inkluderar både biblioteks- och språkreferenser.
- Man får trots detta **inte kopiera text eller lösningar direkt från andra källor.** Man måste skriva koden på egen hand och förstå och beskriva vad den gör.
- Man får **inte kommunicera med andra under tentan**, vare sig för att diskutera uppgifterna eller för andra syften – precis som under en vanlig tenta. Tentan genomförs alltså på egen hand, isolerat. Det gäller både kurskamrater och andra som kan finnas tillgängliga hemma, på telefon, på nätet, eller liknande.
- Man får **inte göra tentasvar eller relaterad information tillgängliga för andra** på något sätt under tentans gång, fram till **16:00** (för att ge marginaler då vissa har förlängd tid). Alla uppgifter ska genomföras helt individuellt.
- Vi påminner om att vi är **skyldiga att anmäla** möjligt fusk till disciplinnämnden, utan att själva försöka reda ut om det faktiskt var fusk.

Distanstentamen: Grundläggande skillnader i uppgifter

Tentauppgifternas natur och stil måste till viss del anpassas för att fungera i en distanstenta med tillgång till Internet. Var därför extra noga med att läsa genom alla delar av varje enskild uppgift, och tänk inte att allt ska fungera som i de flesta tidigare tentor!

- Då vi inte följer tidigare tentamensstruktur försöker vi inte heller ordna uppgifterna i svårighetsordning.
- **Poänggränser förutbestäms inte.** Det kan man bara göra när man har *mycket* erfarenhet av hur poängen motsvarar de faktiska kunskaperna som vi ska examinera. Examinatorns uppgift är alltid att bedöma uppfyllande av kursmål, och förutbestämda poänggränser är bara en av många olika metoder att göra detta.

Distanstentamen: Datormiljö

I och med distanstentan behöver du själv ha tillgång till en datormiljö där du kan arbeta med dina uppgifter. **Vi kan inte hjälpa till med att sätta upp den miljön.**

- Du ska använda **Python 3**, som vi har undervisat i.

  Detta kan finnas som kommandot `python3` på din dator. Det kan också finnas som kommandot `python` – testa då med `python --version` så att det verkligen är en version av Python 3 och **inte gamla Python 2**.

  Du är själv ansvarig för att installera Python 3 om det inte redan finns på din dator. Detta är ditt eget ansvar och ska ha förberetts i god tid inför tentan. För Windows kan du *antagligen* använda "installer"-filerna på slutet av `https://www.python.org/downloads/release/python-396`, "Windows installer (64-bit)".

  Versionen ska vara antingen **Python 3.8.x**, som var standardversion under kursen, eller den nyare **Python 3.9.x**. För att få tillgång till Python 3.8.x på universitetets datorer kan man behöva kommandot `module add courses/TDDE24`. Uppgifterna är **inte** testade med tidigare versioner av Python (3.7, 3.6, ...) och kanske inte fungerar på dessa! Samma gäller prerelease-versioner av Python 3.10. Vill du se vilken version du använder kör du `python --version` alternativt `python3 --version`.

- Du får använda vilka utvecklingsmiljöer eller editorer du vill, men är själv ansvarig för att de fungerar.

- Det är *möjligt* att Thinlinc eller tjänster som `ssh.edu.liu.se` är tillgängliga under tentan. I så fall är det tillåtet att använda dem. Det är också möjligt att dessa tjänster är otillgängliga, eller att de finns tillgängliga till en början men går ner under tentans gång, så att du inte längre kan komma åt ditt arbete. Detta är tyvärr inget som vi i kursledningen kan påverka: LiU:s policy är att tentorna *ska* genomföras på distans och att man är ansvarig för sin egen datormiljö.

Frågor under tentans gång

Tentarelaterade frågor skickas via *epost* direkt till `jonas.kvarnstrom@liu.se`.

Spara inte frågorna till slutet. På en vanlig tenta går man genom alla uppgifter så snart man får tillgång till dem, så man kan ställa alla eventuella frågor till examinatorn vid ett eller två korta besök i tentasalen. Under denna distanstenta kan du visserligen skicka frågor när som helst, men även här kommer frågorna att besvaras “då och då" och **det kan ta någon timme att få svar** – dels går det inte att spendera varje sekund klistrad framför epostklienten, dels kan det komma många frågor på samma gång.

Skickar du frågor för sent kan det hända att du inte hinner få svar i tid för att avsluta din lösning innan tentan avslutas!

För de som har förlängd skrivtid kommer jag att vara tillgänglig för frågor fram till klockan 15, men även under tiden 13-15 kan svaren dröja och även i detta fall behöver ni tänka på att ställa frågorna i tid.

Läs alltså genom uppgifterna i början och ställ frågor i god tid!

Informationsutskick under tentans gång

Information som är intressant för flera tentander kan skickas ut via kursens epostlista under tentans gång. Även de som omtentar ska finnas med på den listan. Detta kan gälla:

- Korrigeringar till tentan.
- Påminnelser om sådant som står i instruktionerna men som flera studenter har missat.

Håll alltså koll på din egen epost under tentan.

Inlämning av svar

Inlämning sker via Lisam. Vi använder “inlämningssystemet" där man laddar ner tentan i PDF-format och gör **en inlämning för hela tentan**.

I inlämningen kan man skriva ett textsvar, men den texten kan vara tom. Det viktiga är möjligheten att **bifoga filer**, vilket man ska använda för att bifoga **en separat fil per uppgift** (`ex1.py`, `ex2.py`, ...).

Man kan lämna in svar flera gånger, men måste då lämna HELA sitt svar (alla filer)! Utnyttja det gärna för att gradvis uppdatera dina svar med nya lösningar, ifall något t.ex. skulle gå fel med din uppkoppling till Internet mot slutet av tentatiden.

Varje svar ersätter helt det tidigare svaret, så om du först lämnar in `ex1.py`, ska samma fil `ex1.py` **skickas med även i nästa inlämning**!

Namnge filerna ex1.py, ex2.py, ex3.py, ...!
Annars kan det hända att vi missar dem i rättningen. Uppgifter som har a- och b-delar ska också lämnas in i samma fil (3a och 3b i ex3.py).

Deadlines

För alla som har tenta *den vanliga tiden 08–13* finns en slutlig hård **deadline 13:10** då ditt slutliga svar ska vara inlämnat. Detta är enbart till för att du inte ska missa inlämningen på grund av att klockan går fel med 1 minut eller för att uppkopplingen till Internet fick kortvarig hicka. Använd inte denna utökade tid till att arbeta vidare – om du då får problem med inlämningssystemet har du inga marginaler kvar.

Har du *utökad skrivtid* till 15:00 använder du en separat inlämning med en slutlig hård **deadline 15:10**.

Om inlämningssystemet av någon anledning skulle släppa genom en försenad uppdatering av en inlämning, räknas bara eventuella tidigare inlämningar som gjordes innan deadline.

Vid tekniska problem

Om du har tekniska problem relaterade till Lisam eller andra system på universitetet, kontakta då helpdesk på `helpdesk@liu.se` eller **013-285898**. Det finns inget som vi i kursledningen kan göra i fråga om teknisk support!

Om du inte skulle få kontakt med Lisam under dina sista 10 minuter går det också att lämna in via epost till adressen `jonas.kvarnstrom@liu.se`. Du har samma deadline som anges ovan, och epost ska skickas från din universitetsadress. **Ange då ditt anonyma nummer, på formen A-99999!**

Detta är bara för nödfall. Normalt ska inlämningar lämnas via inlämningssystemet! När du lämnar in via epost krävs en hel del extra arbete, och du missar möjligheten att vara anonym.

Viktiga tips om testning – missa inte poäng i onödan

Utöver rättningskriterierna (nästa sida) vill vi starkt uppmana er att tänka på detta:

- Vi ger ofta flera testfall i varje uppgift, ibland i `assert`-form och ibland i den löpande texten. **Testa dem alla och skapa egna variationer!** Ofta hittar man fel som faktiskt är lätta att korrigera.
- De testfall vi ger är bara exempel och täcker definitivt inte allt – man måste också utgå från beskrivningen i texten. **Skapa egna tester** som täcker fler fall!
- Det går *delvis* att testa genom att **köra implementationen med många olika indata** även om du inte vet vad korrekt svar ska bli (`print` istället för `assert`). Kraschar koden? Skriver den ut uppenbart felaktiga svar? Då har du hittat ett fel.
- Tänk på att **testa specialfall** som *tomma listor*, *tomma tupler*, *negativa tal*, med mera. Testa också *långa* listor eller *djupt nästlade* listor.
- Var noga med att **läsa exakt vad som står**! En *godtycklig lista* är precis vilken lista som helst, så testa med olika listor, även sådana med nästlade listor eller tupler eller andra elementtyper. En *sekvens* behöver inte nödvändigtvis vara en lista, så testa även med tupler och kanske en sträng eller två. Ska det fungera för heltal $n \geq 0$ ska det också fungera för $n = 0$, så testa det. Ska funktionen *ta en sekvens och returnera en lista* får den inte returnera en tupel, även om inputsekvensen råkade vara en tupel.
- Tänk på att man ofta ska **klara godtycklig input**. Även om de angivna testfallen bara är heltal, kanske det står att man ska klara godtyckliga listor, och alltså vilka element som helst. **Testa då detta!** Fastna inte heller i att ett exempel bara råkar ange *positiva* tal, eller att en lista råkar vara *sorterad*. Återigen, läs vad uppgiften ska klara och skapa egna exempel som testar varierande input.
- För att misslyckade testfall inte ska få koden att krascha när den *importeras* av våra testscript är det bra att se till att testfallen inte alls körs vid importering av filen. Det gör man genom att lägga testfallen inom följande `if`-sats, så slipper man komma ihåg att kommentera bort testfallen på slutet:

```
if __name__ == '__main__':
    print("Running tests!")
    assert f(12) == 34
    assert ...
```

Testning är extremt viktig!

Vi ser ofta *många* fel som väldigt enkelt kunde ha upptäckts och fixats om man bara *testade* sin lösning lite mer. Vad är bäst, att få 4 poäng på 4 uppgifter (16 totalt) eller att hinna med en uppgift till men få 2 poäng per uppgift på grund av bristande testning (10 totalt)?

Men testning fångar inte allt!

Grunden i uppgifterna är alltid att *läsa* och *förstå* vad som egentligen menas.

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i poängavdrag.

- Lösningen ska givetvis vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte resulterade i felaktig kod och så att koden inte kraschar när filen importeras av våra granskningsscript, t.ex. i kvarlämnade `assert`-satser! Poängavdrag ges vid icke körbar kod, även för *delvis* körbar kod (som kraschar för vissa fall).
- Lösningen ska följa alla de **specifika regler och villkor** som står i uppgiften.

  Den ska också **fungera exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat – men tänk på att koden kan vara felaktig trots att körexemplen fungerar! Lösningen ska vara **generell** och ska fungera för *alla* indata som följer uppställningen i uppgiften. Att en lösning enbart fungerar för listor med begränsad längd eller för vissa storlekar på indata är exempel på signifikanta fel.
- Funktioner och källkodsfiler ska ha **exakt samma namn** som anges i uppgiften.
- Man ska kunna köra funktioner **flera gånger** med olika indata och korrekt resultat, utan att ladda om koden däremellan. Se upp med olika former av globalt tillstånd / globala variabler. Se upp med defaultargument och modifiera dem aldrig. Testa själv att köra många testfall i rad.
- Kod ska vara **lättförståelig**. Det innebär t.ex. att egna namn (på parametrar, variabler med mera) ska vara beskrivande och följa namnstandarden. Det innebär också att lösningen ska vara **välstrukturerad** och tillräckligt **väldokumenterad** för att en granskare **enkelt** ska förstå hur lösningen fungerar och varför den ser ut som den gör.

  Om *docstrings* krävs, beskrivs detta uttryckligen i uppgiftens instruktioner. Detta är en ändring från tidigare år.
- 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 uppgiften.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Eventuella testfall i tentatexten visar normalt returnerade värden, inte värden som funktionerna har skrivit ut.
- Man får använda Pythons standardbibliotek (upp till och med Python 3.9), men **inte bibliotek som måste installeras separat**. Se upp om du har installerat extra bibliotek på din dator. **Notera att NumPy är ett separat bibliotek som alltså inte får användas!** Du får inte heller skriva av andras kod.

Rättningskriterier (fortsättning)

Avdrag för felaktig och ofullständig kod ges **även om lösningen är uppenbar** för granskaren, och även om det syns att lösningen “kan ha varit på rätt väg". Det ingår i uppgiften att se till att koden uppfyller specifikationen, och att *själv* upptäcka och åtgärda problem under tentans gång. Den inlämnade koden är ditt slutgiltiga svar, inte ett mellansteg som granskaren ska arbeta vidare med.

Lösningar som misslyckas med alltför många fall räknas normalt inte som lösningar och får 0 poäng. Det går oftast att få delpoäng för lösningar som *misslyckas* med vissa specifika fall, men inte för lösningar som *bara lyckas* med vissa specifika fall.

Rättningskriterier som många missar

Det är relativt vanligt att studenter missar eller ignorerar följande kriterier. Vi har varit snälla vid årets första tenta men kommer att vara hårdare denna gång.

- Lösningen **SKA** vara körbar. Vi **SKA INTE** behöva ändra på koden för att den omedelbart kraschar, t.ex. i testfall som misslyckas.
- Filer och funktioner **SKA** ha korrekta namn.
- Funktioner **SKA** ge korrekta resultat även när man kör dem flera gånger.

Tillåtet / icke-krav

Vi får ofta frågor om vad som är tillåtet i en lösning. **Om inget annat anges** i en uppgift, är följande uttryckligen **tillåtet**:

- Att lösa uppgiften **rekursivt eller iterativt, eller med en kombination** av dessa lösningsmodeller (t.ex. hybrider med rekursiva anrop och defaultargument). Med andra ord: Om inget annat sägs kräver vi inte någon specifik form av rekursion, och du kan använda en lösningsform du känner dig bekväm med.
- Att importera och använda **alla vanliga "inbyggda" funktioner** från Pythons standardbibliotek (upp till och med Python 3.9), t.ex. matematiska funktioner från `math`, högre ordningens funktioner som `map()`, och så vidare.
- Att använda **listbyggare** (list comprehensions), generatoruttryck (generator expressions), slicing (delsekvenser) och andra funktionaliteter i språket.
- Att skapa **hjälpfunktioner**, nästlade eller icke nästlade. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven.
- Att **addera defaultargument till funktioner**, så länge som funktionerna fortfarande går att anropa på sådant sätt som visas i uppgiften. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven. Defaultargument kan ibland användas på sätt som bryter mot en rekursiv lösningsmodell. Se även varningar för defaultargument under rättningskriterier.
- Att **anta att indata följer specifikationen i uppgiften**, utan några egna felkontroller. Står det t.ex. att funktionen ska ta en sekvens, behöver man inte själv kontrollera att man faktiskt får en sekvens som parameter (om inte uppgiften särskilt anger detta).

  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).
- Att **bryta mot "ytliga" kodningsstandarder** i fråga om t.ex. mellanrum, indentering, radlängd och antal blankrader, så länge som koden fortfarande är *lättläst och lättförståelig*.

Detta gäller inte om uppgiften gör specifika undantag!

Uppgift 1: Listor och tupler (3p)

Översikt: I den första uppgiften ska vi plocka element från båda ändarna av en lista och para ihop dem till tupler.

Skapa filen `ex1.py` för hela denna uppgift!

Implementera funktionen `tuplify(seq: list)`, som:

- Tar som argument en lista `seq` med minst ett element, där alla element kan ha godtyckliga typer.
- Returnerar en lista med tupler, där:
  - Första tupeln innehåller första och sista elementet från listan, i den ordningen
  - Andra tupeln innehåller näst första och näst sista elementet från listan
  - ...
  - Sista tupeln innehåller de två "mittersta" elementen i listan (om längden var jämn) eller det enda mittersta elementet i listan (om längden var ojämn)

  Se även exemplen nedan för att förstå tanken!

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Allmänna tips och ledtrådar:

- Tänk på att det finns en speciell syntax för att skapa en tupel med bara 1 element. Uttryck som `(x)` ses bara som vanliga parenteser och skapar inte tupler.

Exempel:

- `assert tuplify([1]) == [(1,)]`
- `assert tuplify(['hello', 'hello', 'world']) == [('hello', 'world'), ('hello',)]`
- `assert tuplify([1, 2, 3, 4]) == [(1, 4), (2, 3)]`
- `assert tuplify([1, 2, 3, 4, 5]) == [(1, 5), (2, 4), (3,)]`
- `assert tuplify([1, 2, 3, 4, 5, 6]) == [(1, 6), (2, 5), (3, 4)]`

Skapa egna testfall av olika storlek, med olika elementtyper och olika kombinationer av värden. Testfallen får gärna lämnas in, men det är inget krav.

Om du inte orkar räkna ut korrekt svar för hand: Skriv åtminstone ut resultatet av `tuplify()` för många olika parametrar. Ser det rimligt ut? Kraschar det?

Uppgift 2: Dela upp i dellistor (6p)

Skapa filen `ex2.py` för hela denna uppgift!

Uppgift 2a: Dela upp med godtycklig lösningsmodell (4p)

Skriv funktionen `split_lists(seq, sizes)`, där:

- Parametern `seq` är en sekvens (lista eller tupel) av $n \geq 1$ godtyckliga element.
- Parametern `sizes` är en *sträng* bestående av siffror (“0" till “9").

Funktionen ska returnera en *lista av* `len(sizes)` *listor*, där varje dellista innehåller så många element som angavs av motsvarande siffra i `sizes`, i tur och ordning:

- `assert split_lists([1, 2, 0, 4, 7, 6], "132") == [[1], [2, 0, 4], [7, 6]]`
  # "132" ==> ska finnas 1+3+2==6 element i listan, vilket det gör
  # "132" ==> plocka i tur och ordning 1, 3 och 2 element ur listan

Om summan av siffrorna i `sizes` skiljer sig från antalet element i `seq`, är problemet ovan inte lösbart. Då ska funktionen istället signalera detta genom att returnera `(None,None)`:

- `assert split_lists([1, 2, 3, "x"], "12") == (None, None)`
- `assert split_lists([1, 2], "12") == (None, None)`

Parametern `sizes` kan innehålla godtyckliga siffror, även 0, på godtycklig position:

- `assert split_lists([1, 2, 3], "102") == [[1], [], [2, 3]]`

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.
- Glöm inte att skapa egna testfall.

Uppgift 2b: Dela upp rekursivt (2p)

För ytterligare 2 poäng, skriv funktionen `split_lists_rec(seq, sizes)` som tar samma parametrar och returnerar samma resultat, men är implementerad med en rekursiv lösningsmodell. Det innebär alltså att ett rekursivt anrop ska beräkna svaret på ett äkta delproblem av exakt samma typ, som vi har diskuterat under kursen.

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.
- Det går utmärkt att lämna in samma rekursiva lösning även för deluppgift 2a, men du måste då ändå definiera (`def`) båda funktionerna (funktionsnamnen). Finns bara en av funktionerna `split_lists(seq, sizes)` eller `split_lists_rec(seq, sizes)` definierad, räknas det som att bara en av dem är inlämnad!

Exempel:

- Använd samma testfall som du skapade för 2a, men med `split_lists_rec` istället.

Uppgift 3: Alternerande hopp (5p)

Översikt: Givet två sekvenser av heltal, använd ett värde från ena sekvensen för att hoppa till ny position i den andra sekvensen, och vice versa. Upprepa till ett slutvillkor är uppnått.

Obs: Denna funktion ser mer komplicerad ut än den egentligen är! En anledning till att beskrivningen är lång är att vi vill illustrera ett exempel.

Skapa filen `ex3.py` för hela denna uppgift!

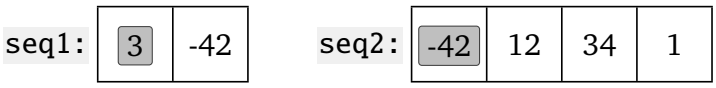
Implementera funktionen `alternating_jumps(seq1, seq2, maxjumps: int)`, som:

- Tar två godtyckliga heltalssekvenser `seq1` och `seq2` samt ett heltal `maxjumps` som anger ett maximalt antal "hopp".

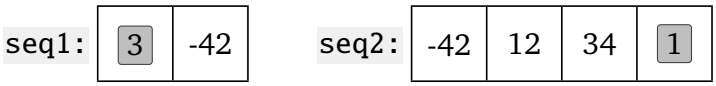
  Vi garanterar att `len(seq1)` ≥ 2, att `len(seq2)` ≥ 2, och att `maxjumps` > 0.

  Sekvenserna kan innehålla godtyckliga heltal, även talet `0`.

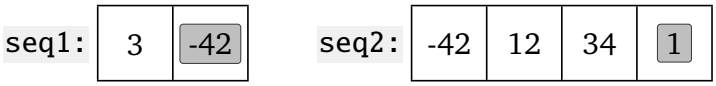
- Börjar på första positionen i både `seq1` och `seq2`.

  > Om `seq1=[3,-42]` och `seq2=[-42,12,34,1]` kan vi till exempel illustrera sekvenserna så här, med en mörkare markering på "nuvarande position":
  >
  > 
  >

- Tittar på värdet på nuvarande position i `seq1`, och hoppar så många steg framåt (eller bakåt för negativa tal) i `seq2`.

  > I exemplet ovan har vi värdet **3** i `seq1` och går alltså tre steg framåt i `seq2`:
  >
  > 
  >

- Tittar på värdet på nuvarande position i `seq2`, och hoppar så många steg framåt (eller bakåt för negativa tal) i `seq1`.

  > I exemplet ovan har vi värdet **1** i `seq2` och går alltså `1` steg framåt i `seq1`:
  >
  > 
  >

- Ständigt håller reda på de hopp som har gjorts hittills. Vi kan kalla detta för `jumps`.

  > Efter det första hoppet i exemplet hade vi `jumps=[3]`. Efter de två hoppen som gjorts ovan har vi `jumps=[3,1]`, eftersom vi hoppade först 3 steg (i `seq2`) och sedan 1 steg (i `seq1`).

- Fortsätter på detta sätt, med byten mellan `seq1` och `seq2`, ända till ett av följande slutvillkor är uppfyllt. Samtliga slutvillkor exemplifieras också i testfallen. Tänk på att slutvillkoren måste testas vid *varje* hopp, inte bara efter att *två* hopp har gjorts!
  - Om vi just har hoppat, och resultatet är att vi pekar ut *sista* elementet i *båda* sekvenserna, returneras `(-4711, jumps)`.

    > I exemplet ovan har vi nått till sista elementet i båda sekvenserna, och därmed returneras `(-4711, [3,1])`.

  - Om vi just har hoppat *utan* nå fram till sista elementet i båda sekvenserna, men vi har uppnått det maximala antalet hopp `maxjumps`, får vi inte hoppa fler gånger. Då returneras `(-49152, jumps)`.
  - Om vi är på väg att hoppa, men hoppet skulle resultera i en ny position som är utanför gränserna för en sekvens (före det första elementet eller efter det sista elementet), kan detta hopp inte genomföras; det är "ogiltigt". Då returneras `(-42, jumps)` där `jumps` innehåller de hopp som faktiskt genomfördes.

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Exempel (skapa gärna egna testfall):

- `assert alternating_jumps([1, -42], [-42, 1], 10) == (-4711, [1, 1])`
- `assert alternating_jumps([3, -42], [-42, 12, 34, 1], 10) == (-4711, [3, 1])`
- `assert alternating_jumps([1, -1, 14], [-1, 1, 14], 10) == (-49152, [1, 1, -1, -1, 1, 1, -1, -1, 1, 1])`
- `assert alternating_jumps([2], [1, 1, -10, 5], 10) == (-42, [2])`
  # Hoppar 2 steg framåt i seq2; försöker sedan hoppa 10 steg bakåt i seq1; kan ej genomföras

Uppgift 4: Beräkna värden i ett träd (5p)

Skapa filen `ex4.py` för hela denna uppgift!

I denna uppgift ska vi beräkna en funktion av ett *träd*, som representeras/lagras så här:

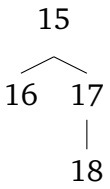
- En **nod** representeras som en tupel med exakt 2 element.
- Första elementet i en nod är dess **nodvärde**, som kan vara ett godtyckligt Python-värde av godtycklig typ.
- Andra elementet i en nod är en lista som innehåller nodens **barn**, som också är noder. Det kan finnas noll eller flera barn.

Som vanligt representeras ett **träd** av dess **rotnod**, och noder utan barn kallas för **löv**. Några exempel på träd är:

- `(15, []) # En rotnod med nodvärde 15, utan barn`
- `(15, [(16, []), (17, [(18, [])])]) # En rotnod med nodvärde 15, med två barn`

Det senare trädet kan vi också *skriva ut* eller *visualisera* så här, så syns strukturen bättre:

```
(15, [
  (16, []),
  (17, [
    (18, [])
  ])
])
```



Vi vet alltså redan vad ett *nodvärde* är. Nu vill vi kunna använda detta för att beräkna ett värde för ett helt träd, vilket vi kallar **trädvärde**.

Trädvärdet för ett träd `t` är dess nodvärde *minus* trädvärdet för första barnet, *plus* trädvärdet för andra barnet, *gånger* trädvärdet för tredje barnet, *minus* trädvärdet för fjärde barnet, *plus* trädvärdet för femte barnet, och så vidare. Vi använder alltså subtraktion, addition och multiplikation för de första tre barnen och börjar sedan om med subtraktion igen, och fortsätter så ända till noden inte har fler barn.

Exempel: Trädvärdet för `(15, [(16, []), (17, [(18, [])])])` är `15 - (16) + (17 - (18))`, där vi på högsta nivån börjar med `-` och fortsätter med `+`. Delträdet `(17, [(18, [])])` behandlas som ett eget träd och dess nodvärde är `17 - (18)`, med subtraktion för det första barnet i delträdet.

Se även exemplen nedan, som visar konkreta resultat för olika träd.

Din uppgift är att skriva funktionen `treeval(tree)`, som beräknar och returnerar trädvärdet för ett träd (där parametern `tree` alltså är trädets rotnod).

Fortsätter på nästa sida

Allmänna tips och ledtrådar:

- Ett träd är en nästlad struktur, och därmed kan rekursion vara användbart. Det finns dock inget krav på detta, och även med rekursion behöver man inte använda en strikt rekursiv lösningsmodell.

Exempel:

- `assert treeval((1, [])) == 1`
- `assert treeval((1, [(2, [])])) == -1`
- `assert treeval((1, [(-3, []), (0, []), (-4, [])])) == -16`
  `# (((1 - -3) + 0) * -4)`
- `assert treeval((1, [(-3, [(7, []), (3, []), (5, [])]), (0, []), (-4, []), [4, []]])) == -148`
- `assert treeval((1, [(-3, [(7, [(42, []), (14, [])]), (123, []), (456, [])]), (75, []), (-44, []), [4, []]])) == 2825676`

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Uppgift 5: Högre ordningens funktioner (4p)

Skapa filen `ex5.py` för hela denna uppgift!

Denna uppgift handlar om högre ordningens funktioner.

Del 1

I del 1 ska du definiera funktionen `apply_to_both(seq1, seq2, f)`, som:

- Tar två sekvenser (listor/tupler) `seq1` och `seq2`, var och en med godtyckligt antal element av godtyckliga typer.
- Även tar en funktion `f` med två argument.
- Applicerar `f` på två element tagna från `seq1` respektive `seq2` i tur och ordning (se exemplen), så länge som det finns element kvar i båda.
- Ignorerar "överblivna" element från den längre sekvensen, om `seq1` och `seq2` inte är lika långa.
- Returnerar en lista som innehåller resultaten av att applicera `f`, i tur och ordning.

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Exempel:

- `assert apply_to_both([1,2,"x"],[float,int,str], isinstance) == [False, True, True]`
  `# Motsvarar [isinstance(1,float), isinstance(2,int), isinstance("x",str)]`
- `assert apply_to_both([1,2,3],[4,5,6], int.__mul__) == [4, 10, 18]`
  `# Där int.__mul__ är funktionen som multiplicerar heltal,`
  `# så returvärdet är [1*4, 2*5, 3*6]`
- `assert apply_to_both([1,2,3],[4,5,6,7], int.__mul__) == [4, 10, 18]`
  `# Samma svar eftersom överskjutande värdet 7 ignoreras`

Del 2

I del 2 ska du definiera funktionen `flip(f)`, som:

- Tar en funktion `f` med två argument.
- Returnerar en ny funktion som beräknar samma värde, men tar sina argument i omvänd ordning.

Exempelvis har Python den inbyggda funktionen `isinstance(obj,class)`, och vi vet att `isinstance(12,int)` är sant. Ett anrop till `flip(isinstance)` ska då returnera en funktion där man kan ge klassen `class` som *första* parameter och objektet `obj` som *andra* parameter, och `flip(isinstance)(int,12)` ska vara sant.

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Exempel:

- `assert flip(isinstance)(int, 12)`
- `assert int.__sub__(10, 4) == 6 # Subtraktion, 10 - 4 == 6`
- `assert flip(int.__sub__)(10, 4) == -6 # Baklänges, 4 - 10 == -6`

Uppgift 6: Kombinatorik (6p)

Skapa filen `ex6.py` för hela denna uppgift!

Obs: Även denna funktion kan se mer komplicerad ut än den egentligen är!

Nu ska vi hitta alla alternativ, och sedan se om något passar!

Tänk dig att du har en lista med tal, till exempel `[1,2,3,4,5]`. Nu vill du veta om det finns något sätt att stoppa in operationerna `+`, `-` och `*` mellan dessa tal så att resultatet blir ett visst värde, till exempel `12`. Det finns många olika varianter som kan provas – en är `1+2+3+4+5` (men det blir 15 och inte 12), en annan är `1+2-3+4-5`.

För att uppgiften ska vara väldefinierad måste vi också diskutera beräkningsordningen. När vi skriver `1+2-3+4-5` menar vi `((((1+2)-3)+4)-5)`, så vi börjar med att räkna ut 1+2 och subtraherar sedan 3, precis som när man skriver in ett tal i en gammaldags miniräknare. Om vi börjar med talen `[12,5,32,1]` och beräknar `12+5*32-1` menar vi `(((12+5)*32)-1)`.

Den här uppgiften *kan* göras i ett enda steg, men för att underlätta vissa delar (inte minst felsökningen) delar vi upp det i två steg där vi kan se resultat i ett mellanläge.

Steg 1: Hitta kombinationer

Det första steget är att hitta *alla* olika sätt att stoppa in de tre operationerna `+`, `-` och `*` mellan värdena i en given heltalslista `numbers`. Med andra ord, vi vill hitta alla möjliga *kombinationer* av operationer.

Varje sådan möjlig kombination ska representeras som en *lista* `operations` där varje operation som används, i tur och ordning, anges av strängen `"+"`, `"-"` eller `"*"`. Om talen är `[12,5,32,1]` representerar vi alltså beräkningen `12+5*32-1` som `["+","*","-"]`.

Vi vill också gärna se *resultaten* av varje uträkning, så detta ska läggas till och ge en tupel på formen `(operations, result)` exempelvis `(["+","*","-"],543)`. Detta representerar alltså att "en möjlig beräkning är `12+5*32-1` och resultatet av denna beräkning är 543".

Uppgiften är nu att skapa funktionen `find_variants(numbers)`, som:

- Tar en lista `numbers` med 2 eller flera godtyckliga heltal.
- Returnerar en lista som innehåller *alla* de möjliga tuplerna på formen `(operations, result)` enligt ovan. Varje kombination av operationer ska finnas med exakt en gång i denna lista.

Exempel: Eftersom man får skapa elementen i vilken ordning som helst definierar vi en hjälpfunktion som låter oss testa att rätt element *finns med* utan att bry sig om ordningen. Mängder är bra till detta.

```
def same_elements(elems1, elems2):
    return set(elems1) == set(elems2)
```

Sedan får vi till slut vårt första testfall:

- `assert same_elements(find_variants([3, 5, 11]), [(["+", "+"], 19), (["+", "-"], -3), (["+", "*"], 88), (["-", "+"], 9), (["-", "-"], -13), (["-", "*"], -22), (["*", "+"], 26), (["*", "-"], 4), (["*", "*"], 165)])`

Med 3 värden i listan finns $3^{3-1} = 9$ olika sätt att applicera operationer. Vi kan till exempel räkna ut `(3+5)+11==19` vilket representeras av tupeln `(["+", "+"], 19)`. På en annan plats i listan ser vi `(["-", "*"], -22)` vilket innebär att `(3-5)*11==-22`. Ytterligare ett testfall följer:

- `assert find_variants([12, 34]) == [(["+"], 46), (["-"], -22), (["*"], 408)]`

Tips:

- Det finns alltid 3^{n-1} kombinationer, där n är antalet element i `numbers`.
- Rekursion är ofta ett lämpligt verktyg, som kan hjälpa till när man måste välja (potentiellt olika) operationer för ett godtyckligt stort antal beräkningssteg. Man *måste* dock inte ha en *rekursiv lösningsmodell*. Det *går* också att lösa detta iterativt, men då genom tekniker vi inte har lärt ut i kursen.

Steg 2: Ger någon kombination rätt värde?

I steg 2 ska du skriva funktionen `find_matching(numbers, wanted)`, som i princip ska filtrera resultatet från `find_variants(numbers)` och returnera en lista med exakt de tupler som anger det önskade värdet `wanted` som resultat.

För full poäng ska denna funktion skrivas med hjälp av en icke nästlad *list comprehension*, på följande form, utan extra hjälpfunktioner eller annan överflödig kod:

```
def find_operations(numbers, wanted: int):
    return [... for ... ... ... ]
```

Exempel:

- `assert find_matching([3, 5, 11], 165) == [(["*", "*"], 165)]`
  `# Lista med den enda tupel i find_variants([3, 5, 11]) som anger resultatet 165!`
- `assert find_matching([1, 1, 1], 1) == [(["+", "-"], 1), (["-", "+"], 1), (["*", "*"], 1)]`