Datortentamen 2022-01-11

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **6 uppgifter** som vardera kan ge maximalt 5 poäng. För betyg 3, 4 och 5 krävs minst 15, 20 respektive 25 poäng. *Detta betyder inte att tentan är svårare än de år gränsen var 12 poäng, utan att poängsättningen har ändrats och vissa avdrag har minskats!*

Vad som är lätt eller svårt skiljer sig från person till person. Därför bör man inte förutsätta att de första uppgifterna är enklast, utan snabbt läsa genom alla uppgifter för att kunna prioritera arbetet. Vissa uppgifter kan också ge delpoäng för att man löser specifika delar, så om en uppgift verkar svår i sin helhet kanske du ändå vill arbeta med en del av den!

Tillåtna hjälpmedel och verktyg

Följande hjälpmedel och verktyg är tillåtna / tillgängliga:

- **Pennor, suddgummin och liknande** för att kunna skissa lösningar på papper. (Lösa tomma papper tillhandahålls av tentamensvakterna.)
- **Python 3.9**, som vi har använt under årets kursomgång. Detta finns som kommandot python3.9 på tentadatorerna.

  I terminalfönstret *bör* det också gå att använda Python 3.9 genom att skriva python eller python3, men dessa "alias" fungerar inte nödvändigtvis om du kör program inuti en editor som vscode. Om du får fel version av Python inuti en editor har våra nya svarsmallar en testfunktion som ska tala om det automatiskt vid testkörningen, och det är då upp till dig själv att lösa problemet: Peka ut korrekt version av Python för editorn, eller kör helt enkelt allt med python3.9 från kommandoraden.
- **Ett antal editorer**, inklusive vscode, atom, och gvim. Fler editorer kan finnas tillgängliga från menyer eller från kommandoradsprompten.

  Editorerna har inte alla plugins installerade. Meningen är att man ska få tillgång till de menyer och tangentbordsbindningar som man är van vid, inte att man ska ha en fullständig integrerad utvecklingsmiljö som hjälper till med allt man ska göra.
- Standardiserade **systemprogram** i kommandoradsprompten eller menyerna.
- **Websidor** under `https://docs.python.org/3.9/` (via "Web Access"-ikonen). Här finns både biblioteks- och språkreferenser.

Otillåtna hjälpmedel; fusk och vilseledande

Otillåtna hjälpmedel inkluderar alla former av elektronisk utrustning utöver tentadatorerna, samt böcker och anteckningar.

Under datortentan arbetar du i en begränsad och övervakad datormiljö där utvecklingsmiljöer som PyCharm och Thonny inte är tillåtna, och där du har begränsad tillgång till nätet.

Utöver detta gäller följande:

- Man får **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 (utom för att ställa frågor till tentavakter, så klart).
- Man får **inte göra tentasvar eller relaterad information tillgängliga för andra** på något sätt under tentans gång, fram till **21:15** (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 eller "försök till vilseledande" till disciplinnämnden, utan att själva försöka reda ut om det faktiskt var fusk.

Svarsmallar

För varje uppgift i tentan *skickar vi med* en “svarsmall", med filnamn från ex1.py till ex6.py. **Mallen ska alltid användas som grund till din inlämnade uppgift** och du ska varken döpa om den eller skapa egna filer. Kopiera mallfilen från `given_files` till den katalog där du arbetar med uppgifterna. Skriv din kod överst i filen och dina tester inuti run_tests(). Testkör med t.ex. “`python3.9 ex1.py`" från kommandoraden.

Mallen hjälper till med detta:

- Testar att man kör rätt version av Python, så att man inte får underliga buggar på grund av fel Pythonversion. Talar annars om vilken version som används.
- Inkluderar “tomma" funktionsdeklarationer för de funktioner du behöver skriva, så du slipper klippa och klistra och inte riskerar att skriva fel. Deklarationerna är bortkommenterade, så du får själv ta bort kommentarstecknen “#". Du får givetvis skapa egna hjälpfunktioner som tillägg!
- Inkluderar eventuella assert-tester som vi redan har angivit i uppgiften, så du slipper klippa och klistra. (Du behöver ändå tänka på att skapa egna tester.)

Mallen kan se ut *ungefär* så här:

```
# Här i början lägger du din egen kod för uppgift 1.

#def the_function_you_should_write(seq: list):
#    pass

def check_python_version():
    ...färdig kod som kollar att du kör rätt version av Python...

def run_tests():
    # De här testerna står uttryckligen som assertions på tentan.
    print("Kör uppgiftens tester...")
    assert the_function_you_should_write(10) == 42

    # Här lägger du dina egna tester.  Du kan till exempel skapa egna
    # assertions, eller lägga till andra tester såsom enkla utskrifter
    # av resultatet av en körning.
    print("Kör egna tester...")

    # Här kan du lägga tester där du inte vet korrekta svar men
    # ändå kan skriva ut resultatet.  Kanske det kraschar, kanske
    # det är uppenbart fel...
    print("Kör utskriftstester...")
    print("Resultat 1:", the_function_you_should_write(4711))

    print("Har kört alla tester")

if __name__ == '__main__':
    check_python_version()
    run_tests()
```

Under tentan: Lämna in uppgifter

Med hjälp av **tentaklienten** (som du har fått information om i förväg och som även beskrivs i `tentaklient.pdf`) skickar du in dina svar, med en separat inlämning per uppgift. Alla uppgifter måste lämnas in innan tentans slut, då tentasystemet automatiskt stängs!

När du gör en inlämning i tentaklienten anger du också *vilken* uppgift du lämnar in (nummer 1–6). I uppdelade uppgifter lämnas del (a) och del (b) in på samma gång, i samma fil. Var försiktig så att du lämnar in rätt uppgiftsnummer!

Det går bra att skicka in en lösning på samma uppgift flera gånger, och den nya inlämningen ersätter då alltid den tidigare. Utnyttja det: **Testa att lämna in en lösning tidigt**, så du garanterat vet hur det fungerar, och **riskera inte att missa sluttid / deadline**, utan lämna in din nuvarande minst 5 minuter före sluttiden även om du tror du kan vilja utnyttja resten av tiden för att polera svaret mera.

Precis som vid vanliga tentor kommer inga svar att granskas förrän efter tentans slut.

Under tentan: Ställa frågor

Om du har **tekniska problem** (kan inte logga in, har problem med tangentbordet, kan inte skicka in svaret på en fråga ...) kontaktar du tentavakterna som vid behov kan kontakta teknisk jour.

Frågor om tentauppgifterna ställs istället via tentaklienten. Frågorna går då till examinatorn. Tentavakter och teknisk jour ska *inte* svara på tentafrågor.

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 datortenta 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 tentaklienten, dels kan det komma många frågor på samma gång.

Ofta kommer många frågor på slutet, så skickar du frågor för sent kan det hända att du inte hinner få svar i tid för att avsluta din lösning innan tentans slut!

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

Under tentan: Informationsutskick

Information som är intressant för flera tentander kan skickas ut via tentaklienten under tentans gång. Detta brukar oftast handla om påminnelser om sådant som redan står i instruktionerna, men som flera studenter har missat eller missförstått.

Håll alltså koll på tentaklienten och dess meddelandesystem under tentan.

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 är de skrivna direkt i `assert`-form och ibland beskrivs de indirekt som del av 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 med sådana där elementen består av nästlade listor eller tupler eller andra datastrukturer. 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.

Att testa är extremt viktigt!

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! Poängavdrag ges vid icke körbar kod. (Misslyckade tester i `run_tests()` är OK, eftersom vi inte kör dessa när vi importerar filerna.)
- 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. Vi hjälper numera till med detta genom svarsmallarna.
- 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.
- 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. Testfall i tentatexten visar normalt returnerade värden, inte värden som har skrivits ut.

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 i 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.

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). Står det att funktionen ska ta en lista av heltal, får man anta att det är en lista och att den bara innehåller heltal.

  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, radbrytningar, radlängd och antal blankrader, så länge som koden fortfarande är *lättläst och lättförståelig*. Samma gäller stil på identifierare (`snake_case`, `camelCase` med mera).

Detta gäller inte om uppgiften gör specifika undantag!

Uppgift 1: Gapful numbers (5p)

Översikt: I den första uppgiften ska vi ta reda på om heltal uppfyller en viss egenskap.

Använd den givna filen `ex1.py` för hela denna uppgift!
Filen innehåller bland annat testfall.

Ett heltal $n \geq 100$ är *gapful* om det är jämnt delbart med talet som formas av dess första och sista siffra.

I den här uppgiften antar vi att vi (precis som vanligt) arbetar i bas 10, med siffror 0 till 9. Då kan vi se att 105 är *gapful*, för första och sista siffran bildar talet 15, och 105 är jämnt delbart med 15. Fler exempel ges längre ner.

Din uppgift är att skriva funktionen `gapful(n: int)`, som returnerar `True` om `n` är gapful, och returnerar `False` annars.

Ytterligare villkor:

- Funktionen ska klara godtyckliga heltal $n \geq 100$. (Man får anta att n uppfyller villkoret och behöver inte testa detta.)

Allmänna tips och ledtrådar:

- För att hitta siffror i ett tal kan det vara bra att först konvertera till en sträng. När man har satt ihop första och sista siffran kan man konvertera tillbaka till ett heltal.

Exempel:

- `assert gapful(100) == True`
- `assert gapful(101) == False`
- `assert gapful(105) == True`
- `assert gapful(106) == False`
- `assert gapful(54288) == True`

Uppgift 2: Kasta om efterföljande element (5p)

Använd den givna filen `ex2.py` för hela denna uppgift!

Vi är nu intresserade av funktionen `reverse_pairs(seq: list)`, som tar en lista `seq` av $n \geq 0$ godtyckliga element och returnerar en *ny* lista där varje element på en jämn position (0, 2, 4, ...) har bytt plats med det efterföljande elementet. Om listans längd är udda ska det sista elementet behålla sin plats, eftersom det inte finns något efterföljande element att byta plats med.

Att göra: Skriv två implementationer av `reverse_pairs`. Båda ska fungera enligt beskrivningen ovan, men:

- `reverse_pairs_i(seq)` ska arbeta enligt **iterativ** lösningsmodell.
- `reverse_pairs_r(seq)` ska arbeta enligt **rekursiv** lösningsmodell.

Kom ihåg: En rekursiv *lösningsmodell* innebär inte bara att funktionen anropar sig själv, utan att varje rekursivt anrop ska beräkna och returnera *det korrekta svaret för ett delproblem*. I detta fall ska alltså `reverse_pairs_r(seq)` anropa sig själv med en *kortare lista* (hur mycket kortare?) och få tillbaka ett korrekt resultat för denna kortare lista.

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

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Detta gäller inte för eventuella egna hjälpfunktioner.

Exempel: Båda funktionerna ska bl.a. klara följande **tester**, där `reverse_pairs` är antingen `reverse_pairs_i` eller `reverse_pairs_r`. Skapa gärna fler tester med inspiration av villkoren ovan!

- `assert reverse_pairs([]) == []`
- `assert reverse_pairs([1, 2, 'x', 4]) == [2, 1, 4, 'x']`
- `assert reverse_pairs([1, 2, 3, 4, 5]) == [2, 1, 4, 3, 5]`

Uppgift 3: Dubblera udda tal i nästlad lista (5p)

Använd den givna filen `ex3.py` för hela denna uppgift!

Av någon anledning tycker vi inte om udda tal. Funktionen `doubled_odds(seq: list)` ska därför ta en godtycklig nästlad lista `seq` och returnera en ny lista som har samma nästlade struktur, men där alla udda heltal n (av typ `int`) är "ersatta" med dubbla värdet $2 * n$.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Detta gäller inte för eventuella egna hjälpfunktioner.
- Listor kan nästlas i godtyckligt antal nivåer. En lösning behöver klara av detta, vilket oftast är enklast om man gör den rekursiv. Det krävs dock inte att man använder en rekursiv *lösningsmodell*, utan lösningen får gärna kombinera iteration och rekursion. Den kan till och med vara helt iterativ, men det kräver tekniker som vi ännu inte har lärt oss.
- Som synes i exemplen nedan kan listan innehålla annat än bara heltal och andra nästlade listor.

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

- `assert doubled_odds([1, 2, 3]) == [2, 2, 6]`
- `assert doubled_odds([-1, [2, 3], ['Hi', 4, [7]]]) == [-2, [2, 6], ['Hi', 4, [14]]]`
- `assert doubled_odds([-1, [2, 3], ('Hi', 4, [7])]) == [-2, [2, 6], ('Hi', 4, [7])]`

Uppgift 4: Högre ordningens funktioner (5p)

Använd den givna filen `ex4.py` för hela denna uppgift!

Innan vi kommer till själva uppgiften ska vi diskutera i vilket sammanhang den kan vara intressant. Tänk dig därför – *men skapa inte* – följande funktioner:

- `f2(x)`, som beräknar `sqrt(sqrt(x))`
- `f3(x)`, som beräknar `sqrt(sqrt(sqrt(x)))`
- `f4(x)`, som beräknar `sqrt(sqrt(sqrt(sqrt(x))))`
- `f4b(x)`, som beräknar `ln(ln(ln(ln(x))))`

Vi ser ett mönster, där vi applicerar en funktion flera gånger på ett värde – men det kan vara olika antal gånger, och det kan vara olika funktioner som appliceras. För att slippa skriva en separat funktionsdefinition (`def`) för varje sådan funktion vill vi istället generalisera detta, med hjälp av en högre ordningens funktion `multiple_apply(fn, times: int)` som låter oss skriva så här för att definiera funktionerna:

- `f2 = multiple_apply(sqrt, 2)`
- `f3 = multiple_apply(sqrt, 3)`
- `f4 = multiple_apply(sqrt, 4)`
- `f4b = multiple_apply(ln, 4)`

Dessa anrop ska alltså *definiera* funktionerna, så vi sedan kan *anropa* dem som t.ex. `f3(x)`.

Deluppgift 4a (3p)

Skapa nu denna högre ordningens funktion, alltså `multiple_apply(fn, times: int)`. Funktionen ska **ta** en funktion `fn` och ett icke-negativt heltal `times` som argument och **returnera** en ny funktion. När den *returnerade* funktionen anropas ska den i sin tur:

- Ta ett tal `x` som argument
- Returnera värdet av `fn(fn(fn(...(x))))`, där `fn` har applicerats totalt `times` gånger (lämpligen genom en loop).

  Om `times==0` returneras alltså helt enkelt `x`.

Exempel:

- Anta att vi sätter `newfun = multiple_apply(foo, 3)`, där `foo` är någon godtycklig funktion som tar exakt 1 parameter.
- Då ska `newfun` vara en funktion, och `newfun(10)` ska returnera `foo(foo(foo(10)))`.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Detta gäller inte för eventuella egna hjälpfunktioner.

Deluppgift 4b (2p)

Definiera funktionen `pow2mult(n: int, c: int)`, som använder sig av `multiple_apply` för att beräkna $2^n \cdot c$. Funktionen `pow2mult` får definieras med `def`, som vanligt, eller med hjälp av tilldelning på det sätt som exemplifierades i uppgift 4a.

Du ska inte definiera några ytterligare hjälpfunktioner, utan ska använda en lambdafunktion som första argument till `multiple_apply`.

Tips: $2^n \cdot c$ kan beräknas genom att börja med c och dubblera detta värde `n` gånger. Använd `multiple_apply` för att få en funktion som åstadkommer själva dubbleringarna, och anropa denna funktion.

Exempel:

- `assert pow2mult(3, 1) == 8`
- `assert pow2mult(3, 3) == 24`
- `assert pow2mult(0, 3) == 3`
- `assert pow2mult(15, 0) == 0`

Uppgift 5: Attraktiva tal (5p)

Använd den givna filen `ex5.py` för hela denna uppgift!

I denna uppgift ska du avgöra om ett *tal* är *attraktivt*. För att göra detta ska du skriva tre funktioner. Funktionerna kan ge delpoäng oberoende av varandra, men poängsumman bedöms utifrån uppgiften i sin helhet.

Steg 1: Primtal

Om ett tal är attraktivt eller inte beror på vissa egenskaper hos primtal. Vi behöver därför börja med att skriva följande funktion.

- `is_prime(n: int)` ska ta ett heltal `n` ≥ 1 och returnera `True` om heltalet `n` är ett primtal. Annars ska funktionen returnera `False`.

Tips: Det lägsta primtalet är 2. Om ett tal $n > 2$ är *jämnt delbart* med något heltal från och med 2 upp till och med `int(math.sqrt(n))`, är n inte ett primtal (det gick ju att dela upp det). Annars *är* det ett primtal.

Notera alltså att man *inte* behöver försöka dela med alla tal upp till n. Att avsluta testningen vid `int(math.sqrt(n))` är väldigt mycket snabbare då n är stort.

Exempel:

- `assert is_prime(1) == False`
- `assert is_prime(2) == True`
- `assert is_prime(10) == False`
- `assert is_prime(11) == True`

Steg 2: Primtalsfaktorisering

Vi behöver också kunna dela upp ett positivt heltal n i sina *primfaktorer*. I detta fall ska vi hitta en *lista* som enbart innehåller *primtal*, så att produkten av alla tal i listan är lika med det ursprungliga talet n. Exempelvis har talet 20 primfaktorerna `[2,2,5]`, eftersom 2 och 5 är primtal och $2 \cdot 2 \cdot 5 = 20$. Implementera därför följande funktion.

- `prime_factors(n: int)` ska ta ett heltal `n` ≥ 2 och returnera en *lista av heltal* som innehåller samtliga primfaktorer i `n`. Som ovan ska även “dubblerade" primfaktorer vara med: Produkten av alla tal i returvärdet ska vara lika med `n`.

  Returvärdet får ange primfaktorerna i godtycklig ordning: `prime_factors(20)` får t.ex. returnera `[2,2,5]` eller `[2,5,2]` eller `[5,2,2]`.

Exempel: Vi sorterar returvärdet för att ordningen inte ska spela någon roll.

- `assert sorted(prime_factors(2)) == [2]`
- `assert sorted(prime_factors(10)) == [2, 5]`
- `assert sorted(prime_factors(20)) == [2, 2, 5]`
- `assert sorted(prime_factors(55)) == [5, 11]`

Steg 3: Attraktiva tal

Ett tal är ett **attraktivt tal** (*attractive number*) om antalet primfaktorer i talet också är ett primtal. Här räknas återigen alla primfaktorer, även de som upprepas.

Som vi har sagt tidigare har talet 20 primfaktorerna `[2,2,5]`. Det finns alltså 3 primfaktorer, och eftersom 3 är ett primtal är 20 ett attraktivt tal. Talet 40 har istället 4 primfaktorer, `[2,2,2,5]`, och eftersom 4 inte är ett primtal är 40 *inte* ett attraktivt tal.

Med hjälp av de tidigare funktionerna kan du nu till slut skriva följande funktion:

- `is_attractive(n: int)` ska ta ett heltal `n` ≥ 2 och returnera `True` om `n` är attraktivt. Annars ska funktionen returnera `False`.

Exempel:

- `assert is_attractive(16) == False`
- `assert is_attractive(20) == True`
- `assert is_attractive(21) == True`
- `assert is_attractive(22) == True`
- `assert is_attractive(23) == False`
- `assert is_attractive(24) == False`
- `assert is_attractive(55) == True`

Uppgift 6: Dela upp i delmängder (5p)

Använd den givna filen `ex6.py` för hela denna uppgift!

Tänk dig att du har ett antal godispåsar, där var och en innehåller ett visst antal godisbitar. Du ska nu fördela godispåsarna mellan två barn. Det är väldigt viktigt att antalet godisbitar blir så lika som möjligt, så att ingen blir alltför besviken, men du kan inte öppna påsarna och dela ut enskilda bitar. Istället måste du försöka dela ut hela påsar på ett sätt som minimerar skillnaden i antal bitar.

Exempelvis har du 4 påsar med `[1,2,3,5]` bitar i varje. Då kan du ge ena barnet påsar med `[1,5]` bitar och andra barnet påsar med `[2,3]` bitar. Skillnaden blir 1 bit, och vi kan inte få en mindre skillnad om vi ska dela ut just de här påsarna.

Din uppgift är att definera funktionen `minimize_differences(seq: list[int])`, som ska hitta ett sätt att minimera denna skillnad. Funktionen tar som parameter en lista `seq` innehållande $n \geq 0$ godtyckliga heltal, och ska returnera en tupel `(seq1,seq2)` så att:

- **Varje element i `seq` har placerats i antingen `seq1` eller `seq2`.** Med andra ord, vi har delat ut varje godispåse till första eller andra barnet, och inte glömt bort någon påse, hittat på nya påsar eller ändrat antalet bitar.

  Notera att `seq` kan innehålla dubletter (två godispåsar med samma storlek). I det fallet hanterar vi fortfarande varje element för sig. Om `seq==[1,2,2,1]` kan vi t.ex. ha `seq1=[1,2]` och `seq2=[1,2]`, eller `seq1=[1,2,2]` och `seq2=[1]`.

- **Skillnaden mellan summan av talen i `seq1` och summan av talen i `seq2` är minimal.**

  Det är *absolutvärdet* av skillnaden som är relevant. Om `seq1=[1,2,2]` och `seq2=[1]` är deras summor 5 respektive 1, och absoluta skillnaden är 4: Någon fick 4 godisbitar mer än den andra. Om `seq1=[1]` och `seq2=[1,2,2]` är absoluta skillnaden fortfarande 4. Det är denna skillnad som ska minimeras.

  Alltså kan `minimize_differences([1,2,3,1])` returnera t.ex. `([1,2],[1,3])` eller `([3],[1,2,1])` eller `([3],[1,1,2])` – det kan finnas olika kombinationer som ger samma skillnad, och den inbördes ordningen i respektive lista spelar ingen roll för slutsumman. (I detta fall var det omöjligt att få en mindre skillnad än 1.)

  Däremot kan `minimize_differences([1,2,2,1])` inte returnera `([1,2,2],[1])`. I det svaret hade vi visserligen placerat ut alla element från `seq` i `seq1` eller `seq2`, men det fanns alternativ med mindre skillnad.

Tips på nästa sida!

Allmänna tips och ledtrådar:

- Att hitta alla möjliga uppdelningar av elementen är en **kombinatorisk** uppgift där man behöver testa “alla alternativ" – alla sätt att dela upp godispåsarna mellan barnen. Fastna inte i fällan att på en gång försöka avgöra genom smarta tumregler om ett visst element borde läggas i `seq1` eller `seq2`. Prova alltid båda alternativen.
- Det kan vara enklast att i ett första steg hitta *alla* möjliga uppdelningar av elementen, oavsett om de är minimala eller inte. Då kan man alltså ha en hjälpfunktion som skapar en lång *lista* med alternativa tupler `(seq1,seq2)`. I nästa steg kan man använda denna lista för att se vilket alternativ som ger den minsta skillnaden.

  Fördel 1: Divide and conquer. Lös två mindre delproblem som kan testas var för sig.

  Fördel 2: Det blir lättare att strukturera lösningen då man inte behöver *jämföra* alternativ på samma gång som man *hittar* alternativen.
- Rekursion är ofta ett lämpligt verktyg i kombinatoriska uppgifter. 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.
- Det kan tänkas att det inte finns någon påse: `len(seq)==0`.
- Det går att ge 0 påsar till något barn, även om man har flera påsar.
- Det kan som sagt finnas flera minimala lösningar till samma problem.

Exempel: I den här uppgiften kan det ofta finnas många korrekta svar som alla ger minimal skillnad. Dels spelar ordningen på elementen i `seq1` respektive `seq2` ingen roll, eftersom det är *summornas* differens vi är ute efter. Dels kan det finnas flera olika sätt att kombinera olika tal till samma summor.

Därför anger vi inga exakta returvärden, utan skickar med (i `ex6.py`) en hjälpfunktion `check(fun, seq, correct_diff)` som används för testningen. Funktionen testar att de korrekta talen finns med i returvärdet och att differensen mellan `seq1` och `seq2` är den förväntade.

Som alltid kan man inte garantera att alla fel hittas med hjälp av testning!

- `check(minimize_differences, [1], 1)`
- `check(minimize_differences, [1, 1], 0)`
- `check(minimize_differences, [12, 34, 56], 10)`
- `check(minimize_differences, [11, 34, 11, 99, 99, 99, 99], 12)`
- `check(minimize_differences, [3, 4, 5, -3, 100, 1, 89, 54, 23, 20], 0)`
- `check(minimize_differences, [23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4], 1)`