IB111 Základy programování

N. Beneš, L. Korenčik, N. Kovářová, H. Lauko, P. Ročkai

Část A: Pravidla a organizace

Tento dokument je sbírkou cvičení a komentovaných příkladů zdrojového kódu. Každá kapitola odpovídá jednomu týdnu semestru a tedy jednomu cvičení. Cvičení v druhém týdnu semestru („nulté“) je určeno k seznámení se s výukovým prostředím, studijními materiály a základními nástroji ekosystému.

Každá část sbírky (zejména tedy všechny ukázky a příklady) jsou také k dispozici jako samostatné soubory, které můžete upravovat a spouštět. Této rozdělené verzi sbírky říkáme **zdrojový balík**. Aktuální verzi[1] (ve všech variantách) můžete získat dvěma způsoby:

1. Ve **studijních materiálech**[2] předmětu v ISu – soubory PDF ve složce text, zdrojový balík ve složkách 00 (organizační informace), 01 až 12 (jednotlivé kapitoly = týdny semestru), dále s1 až s3 (sady úloh) a konečně ve složce sol vzorová řešení. Doporučujeme soubory stahovat dávkově pomocí volby „stáhnout jako ZIP“.
2. Po přihlášení na studentský server aisa (buď za pomoci ssh nebo putty) zadáním příkazu ib111 update. Všechny výše uvedené složky pak naleznete ve složce ~/ib111.

Tato kapitola (složka) dále obsahuje **závazná** pravidla a organizační pokyny. Než budete pokračovat, pozorně si je prosím přečtěte.

Pro komunikaci s organizátory kurzu slouží **diskusní fórum** v ISu (více informací naleznete v části T.1). Nepište prosím organizátorům ani cvičícím maily ohledně předmětu, nejste-li k tomu specificky vyzváni. S žádostmi o výjimky ze studijních povinností, omluvenkami, atp., se obracejte vždy na studijní oddělení.

A.1: Přehled

Tento předmět sestává z cvičení, sad domácích úloh a závěrečného testu (zkoušky). Protože se jedná o „programovací“ předmět, většina práce v předmětu – a tedy i jeho hodnocení – se bude zaměřovat na praktické programování. Je důležité, abyste programovali co možná nejvíce, ideálně každý den, ale minimálně několikrát každý týden. K tomu Vám budou sloužit příklady v této sbírce a domácí úlohy, kterých budou za semestr 3 sady, a budou znatelně většího rozsahu (maximálně malé stovky řádků). V obou případech bude v průběhu semestru stoupat náročnost – je tedy důležité, abyste drželi krok a práci neodkládali na poslední chvíli.

Protože programování je těžké, bude i tento kurz těžký – je zcela nezbytné vložit do něj odpovídající úsilí. Doufáme, že kurz úspěšně absolvujete, a co je důležitější, že se v něm toho naučíte co nejvíce. Je ale nutno podotknout, že i přes svou náročnost je tento kurz jen malým krokem na dlouhé cestě.

A.1.1 Probíraná témata Předmět je rozdělen do 4 bloků (čtvrtý blok patří do zkouškového období). Do každého bloku v semestru patří 4 kapitoly (témata) a jim odpovídající 4 cvičení.

blok		téma
1	1.	if, cykly, proměnné, funkce
	2.	funkce, typy, ladění
	3.	seznamy, n-tice
	4.	typy, assert, korektnost
2	5.	datové struktury, složitost
	6.	proměnné, objekty, paměť
	7.	třídy, linked list
	8.	řazení
3	9.	rekurze 1
	10.	rekurze 2 – backtracking
	11.	práce s textem
	12.	interpret mini-pythonu
–	13.	bonusy, opakování

A.1.2 Organizace sbírky V následujících sekcích naleznete detailnější informace a **závazná** pravidla kurzu: doporučujeme Vám, abyste se s nimi důkladně seznámili. Zbytek sbírky je pak rozdělen na části, které odpovídají jednotlivým týdnům semestru. **Důležité:** během druhého týdne semestru už budete řešit přípravy z první kapitoly, přestože první cvičení je ve až v týdnu třetím. Nulté cvičení je volitelné a není nijak hodnoceno.

Kapitoly jsou číslovány podle témat z předchozí tabulky: ve třetím týdnu semestru se tedy **ve cvičení** budeme zabývat tématy, ke kterým jste v druhém týdnu vypracovali a odevzdali přípravy.

A.1.3 Plán semestru Tento kurz vyžaduje značnou aktivitu během semestru. V této sekci naleznete přehled důležitých událostí formou kalendáře. Jed-

[1] Než začnete pracovat na přípravách nebo příkladech ze sady, vždy se prosím ujistěte, že máte jejich aktuální verzi. Zadání příprav lze považovat za finální počínaje půlnocí na pondělí odpovídajícího týdne, sady podobně půlnocí na první pondělí odpovídajícího bloku. Bude-li nutné provést nějaké změny v zadání později, budete o nich informováni v diskusním fóru.

[2] https://is.muni.cz/auth/el/fi/podzim2025/IB111/um/

notlivé události jsou značeny takto (bližší informace ke každé naleznete v následujících odstavcích tohoto úvodu):

- „#X“ – číslo týdne v semestru,
- „cv0“ – tento týden běží „nulté“ cvičení (kapitola B),
- „cv1“ – tento týden probíhají cvičení ke kapitole 1,
- „X/v“ – mezivýsledek verity testů příprav ke kapitole X,
- „X/p“ – poslední termín odevzdání příprav ke kapitole X,
- „sX/Y“ – Yté kolo verity testů k sadě X.

Nejdůležitější události jsou zvýrazněny: termíny odevzdání příprav a poslední termín odevzdání úloh ze sad (obojí vždy o 23:59 uvedeného dne).

září

	Po	Út	St	Čt	Pá	So	Ne
#1	15	16	17	18	19	20	21
#2 cv 0	22	23	24	25 01/v	26	27 **01/p**	28 sv
#3 cv 1	29 s1/1	30					

říjen

	Po	Út	St	Čt	Pá	So	Ne
#3			1 s1/2	2 02/v	3 s1/3	4 **02/p**	5
#4 cv 2	6 s1/4	7	8 s1/5	9 03/v	10 s1/6	11 **03/p**	12
#5 cv 3	13 s1/7	14	15 s1/8	16 04/v	17 s1/9	18 **04/p**	19
#6 cv 4	20 s1/10	21	22 s1/11	23 05/v	24 **s1/12**	25 **05/p**	26
#7 cv 5	27 s2/1	28 sv	29 s2/2	30 06/v	31 s2/3		

listopad

	Po	Út	St	Čt	Pá	So	Ne
#7						1 **06/p**	2
#8 cv 6	3 s2/4	4	5 s2/5	6 07/v	7 s2/6	8 **07/p**	9
#9 cv 7	10 s2/7	11	12 s2/8	13 08/v	14 s2/9	15 **08/p**	16
#10 cv 8	17 sv s2/10	18	19 s2/11	20 09/v	21 **s2/12**	22 **09/p**	23
#11 cv 9	24 s3/1	25	26 s3/2	27 10/v	28 s3/3	29 **10/p**	30

prosinec

	Po	Út	St	Čt	Pá	So	Ne
#12 cv10	1 s3/4	2	3 s3/5	4 11/v	5 s3/6	6 **11/p**	7
#13 cv11	8 s3/7	9	10 s3/8	11 12/v	12 s3/9	13 **12/p**	14
#14 cv12	15 s3/10	16	17 s3/11	18	19 **s3/12**	20	21
	22	23	24	25	26	27	28
	29	30	31				

A.2: Hodnocení

Abyste předmět úspěšně ukončili, musíte v **každém bloku**[3] získat **50 bodů**. Žádné další požadavky nemáme.

Výsledná známka závisí na celkovém součtu bodů (splníte-li potřebných 4×50 bodů, automaticky získáte známku alespoň E). Hodnota ve sloupci „předběžné minimum“ danou známku zaručuje – na konci semestru se hranice ještě mohou posunout směrem dolů tak, aby výsledná stupnice přibližně odpovídala očekávané distribuci dle ECTS.[4]

známka	předběžné minimum	po vyhodnocení semestru
A	360	90. percentil + 75
B	320	65. percentil + 75
C	280	35. percentil + 75
D	240	10. percentil + 75
E	200	200

Body lze získat mnoha různými způsoby (přesnější podmínky naleznete v následujících sekcích této kapitoly). V blocích 1-3 (probíhají během semestru) jsou to:

- za každou úspěšně odevzdanou přípravu **1** bod (max. 6 bodů každý týden, nebo **24/blok**),
- za každou přípravu, která projde „verity“ testy navíc další **1** bod (max. 6 bodů každý týden, nebo **24/blok**),
- za účast[5] na cvičení získáte 3 body (max. tedy **12/blok**),
- za aktivitu ve cvičení 3 body (max. tedy **12/blok**).

Za přípravy a cvičení lze tedy získat teoretické maximum **72** bodů. Dále můžete získat:

- **7** bodů za úspěšně vyřešený příklad ze sady domácích úloh (maximálně 4 příklady, celkem tedy až **28/blok**).

Konečně blok 4, který patří do zkouškového období, nemá ani cvičení ani sadu domácích úloh. Body získáte účastí na závěrečném testu:

- **16** bodů za každý zkouškový příklad (5 příkladů, maximálně tedy celkem **80/blok**),
- -2 až +2 body za každou z 10 teoretických otázek (celkem až **20/blok**).

A.3: Přípravy

Jak již bylo zmíněno, chcete-li se naučit programovat, musíte programování věnovat nemalé množství času, a navíc musí být tento čas rozložen do delších období – semestr nelze v žádném případě doběhnout tím, že budete týden

[3] Máte-li předmět ukončen zápočtem, čtvrtý blok a tedy ani závěrečný test pro Vás není relevantní. Platí požadavek na 3×50 bodů z bloků v semestru.

[4] Percentil budeme počítat z bodů v semestru (první tři bloky) a bude brát do úvahy všechny studenty, bez ohledu na ukončení, kteří splnili tyto tři bloky (tzn. mají potřebné minimum 3×50 bodů).

[5] V případě, že jste **řádně omluveni** v ISu, nebo Vaše cvičení **odpadlo** (např. padlo na státní svátek), můžete body **za účast** získat buď náhradou v jiné skupině (pro státní svátky dostanete instrukce mailem, individuální případy si domluvte s cvičícími obou dotčených skupin). Nemůžete-li účast nahradit takto, **domluvte se** se svým cvičícím (v tomto případě lze i mailem) na vypracování 3 rozšířených příkladů ze sbírky (přesné detaily Vám sdělí cvičící podle konkrétní situace). Neomluvenou neúčast lze nahrazovat **pouze** v jiné skupině a to nejvýše jednou za semestr.

programovat 12 hodin denně, i když to možná pokryje potřebný počet hodin. Proto od Vás budeme chtít, abyste každý týden odevzdali několik vyřešených příkladů z této sbírky. Tento požadavek má ještě jeden důvod: chceme, abyste vždy v době cvičení už měli látku každý samostatně nastudovanou, abychom mohli řešit zajímavé problémy, nikoliv opakovat základní pojmy.

Také Vás prosíme, abyste příklady, které plánujete odevzdat, řešili vždy samostatně: případnou zakázanou spolupráci budeme trestat (viz také konec této kapitoly).

A.3.1 Odevzdání Každý příklad obsahuje základní sadu testů. To, že Vám tyto testy prochází, je jediné kritérium pro zisk základních bodů za odevzdání příprav. Poté, co příklady odevzdáte, budou **tytéž testy** na Vašem řešení automaticky spuštěny, a jejich výsledek Vám bude zapsán do poznámkového bloku. Smyslem tohoto opatření je zamezit případům, kdy omylem odevzdáte nesprávné, nebo jinak nevyhovující řešení, aniž byste o tom věděli. Velmi silně Vám proto doporučujeme odevzdávat s určitým předstihem, abyste případné nesrovnalosti měli ještě čas vyřešit. Krom základních („sanity“) testů pak ve čtvrtek o 23:59 a znovu v sobotu o 23:59 (těsně po konci odevzdávání) spustíme **rozšířenou** sadu testů („verity“).

Za každý odevzdaný příklad, který splnil **základní** („sanity“) testy získáváte jeden bod. Za příklad, který navíc splnil **rozšířené** testy získáte další bod (tzn. celkem 2 body). Výsledky testů naleznete v **poznámkovém bloku** v informačním systému.

Příklady můžete odevzdávat:

1. do **odevzdávárny** s názvem NN v ISu (např. 01),
2. příkazem ib111 submit ve složce ~/ib111/NN.

Podrobnější instrukce naleznete v kapitole T (technické informace, soubory 00/t*).

Termíny pro odevzdání příprav k jednotlivým kapitolám jsou shrnuty v přehledovém kalendáři v části A.1 takto:

- „01/v“ – předběžné (čtvrteční) verity testy pro příklady z první kapitoly,
- „01/p“ – poslední (sobotní) termín odevzdání příprav z 1. kapitoly,
- analogicky pro další kapitoly.

A.4: Cvičení

Těžiště tohoto předmětu je jednoznačně v samostatné domácí práci – učit se programovat znamená zejména hodně programovat. Společná cvičení sice nemohou tuto práci nahradit, mohou Vám ale přesto v lecčem pomoct. Smyslem cvičení je:

1. analyzovat problémy, na které jste při samostatné domácí práci narazili, a zejména prodiskutovat, jak je vyřešit,
2. řešit programátorské problémy společně (s cvičícím, ve dvojici, ve skupině) – nahlédnout, jak o programech a programování uvažují ostatní, a užitečné prvky si osvojit.

Cvičení je rozděleno na dva podobně dlouhé segmenty, které odpovídají těmto bodům. První část probíhá přibližně takto:

- cvičící vybere ty z Vámi odevzdaných příprav, které se mu zdají něčím zajímavé – ať už v pozitivním, nebo negativním smyslu,
  - řešení bude **anonymně** promítat na plátno a u každého otevře diskusi o tom, čím je zajímavé;
  - Vaším úkolem je aktivně se do této diskuse zapojit (můžete se například ptát, proč je daná věc dobře nebo špatně, a jak by se udělala lépe, vyjádřit svůj názor, odpovídat na dotazy cvičícího),
  - k promítnutému řešení se můžete přihlásit a ostatním přiblížit, proč je napsané tak, jak je, nebo klidně i rozporovat případnou kritiku (není to ale vůbec nutné),
- na Vaši žádost lze ve cvičení analogicky probrat **neúspěšná** řešení příkladů (a to jak příprav, tak příkladů z uzavřených sad).

Druhá část cvičení je variabilnější, ale bude se vždy točit kolem bodů za aktivitu (každý týden můžete za aktivitu získat maximálně 3 body).

Ve čtvrtém, osmém a dvanáctém týdnu proběhnou „vnitrosemestrálky“, kde budete řešit samostatně dva příklady ze sbírky, bez možnosti hledat na internetu – tak, jak to bude na závěrečném testu; každé úspěšné řešení (tzn. takové, které splní verity testy) získá 3 body za aktivitu pro daný týden (celkem tedy lze za příklady získat 6 bodů). Navíc dostanete 3 teoretické otázky, po jednom bodu, celkově lze tedy během vnitrosemestrálky získat až 9 bodů (počítají se jako aktivita, tzn. platí celkový limit 12/blok).

V ostatních týdnech budete ve druhém segmentu kombinovat různé aktivity, které budou postavené na příkladech typu r z aktuální kapitoly (které konkrétní příklady budete ve cvičení řešit, vybere cvičící, může ale samozřejmě vzít v potaz Vaše preference):

1. Můžete se přihlásit k řešení příkladu na plátně, kdy primárně vymýšlíte řešení Vy, ale zbytek třídy Vám bude podle potřeby radit, nebo se ptát co/jak/proč se v řešení děje. U jednodušších příkladů se od Vás bude také očekávat, že jako součást řešení doplníte testy.
2. Cvičící Vám může zadat práci ve dvojicích – první dvojice, která se dopracuje k funkčnímu řešení získá možnost své řešení předvést zbytku třídy – vysvětlit jak a proč funguje, odpovědět na případné dotazy, opravit chyby, které v řešení publikum najde, atp. – a získat tak body za aktivitu. Získané 3 body budou rozděleny rovným dílem mezi vítězné řešitele.
3. Příklad můžete také řešit společně jako skupina – takto vymyšlený kód bude zapisovat cvičící (body za aktivitu se v tomto případě neudělují).

A.5: Sady domácích úloh

Ke každému bloku patří sada 4–6 domácích úloh. Na úspěšné odevzdání každé domácí úlohy budete mít 12 pokusů rozložených do 4 týdnů odpovídajícího bloku cvičení. Odevzdávání bude otevřeno vždy v 0:00 prvního dne bloku (tzn. 24h před prvním spuštěním verity testů).

Termíny odevzdání (vyhodnocení verity testů) jsou vždy v pondělí, středu a pátek v 23:59 – vyznačeno jako s1/1–12, s2/1–12 a s3/1–12 v přehledovém kalendáři v části A.1.

A.5.1 Odevzdávání Součástí každého zadání je jeden zdrojový soubor (kostra), do kterého své řešení vepíšete. Vypracované příklady lze pak odevzdávat stejně jako přípravy:

1. do **odevzdávárny** s názvem sN_úkol v ISu (např. s1_a_queens),
2. příkazem ib111 submit sN_úkol ve složce ~/ib111/sN, např. ib111 submit s1_a_queens.

Podrobnější instrukce naleznete opět v kapitole T.

A.5.2 Vyhodnocení Vyhodnocení Vašich řešení probíhá ve třech fázích, a s každou z nich je spjata sada automatických testů. Tyto sady jsou:

- „syntax“ – kontroluje, že odevzdaný program je syntakticky správně, lze jej přeložit a prochází základními statickými kontrolami,
- „sanity“ – kontroluje, že odevzdaný program se chová „rozumně“ na jednoduchých případech vstupu; tyto testy jsou rozsahem a stylem podobné těm, které máte přiložené k příkladům ve cvičení,
- „verity“ – důkladně kontrolují správnost řešení, včetně složitých vstupů a okrajových případů.

Fáze na sebe navazují v tom smyslu, že nesplníte-li testy v některé fázi, žádná další se už (pro dané odevzdání) nespustí. Pro splnění domácí úlohy je klíčová fáze „verity“, za kterou jsou Vám uděleny body. Časový plán vyhodnocení fází je následovný:

- kontrola „syntax“ se provede obratem (do cca 5 minut od odevzdání),
- kontrola „sanity“ každých 6 hodin počínaje půlnocí (tzn. 0:00, 6:00, 12:00, 18:00),
- kontrola „verity“ se provede v pondělí, středu a pátek ve 23:59 (dle tabulky uvedené výše).

Vyhodnoceno je vždy pouze nejnovější odevzdání, a každé odevzdání je vyhodnoceno v každé fázi nejvýše jednou. Výsledky naleznete v poznámkových blocích v ISu (každá úloha v samostatném bloku), případně je získáte příkazem ib111 status.

Za každý domácí úkol, ve kterém Vaše odevzdání v příslušném termínu splní testy „verity“, získáte 7 bodů (strop bodů za úkoly je 28 za blok, počítají

se tedy maximálně čtyři úspěšně vyřešené úkoly).

A.5.3 Neúspěšná řešení Příklady, které se Vám nepodaří vyřešit kompletně (tzn. tak, aby na nich uspěla kontrola „verity“) nebudeme hodnotit. Nicméně může nastat situace, kdy byste potřebovali na „téměř hotové“ řešení zpětnou vazbu, např. proto, že se Vám nepodařilo zjistit, proč nefunguje.

Taková řešení můžou být předmětem společné analýzy ve cvičení, v podobném duchu jako probíhá rozprava kolem odevzdaných příprav (samozřejmě až poté, co pro danou sadu skončí odevzdávání). Máte-li zájem takto rozebrat své řešení, domluvte se, ideálně s předstihem, se svým cvičícím. To, že jste autorem, zůstává mezi cvičícím a Vámi – Vaši spolužáci to nemusí vědět (ke kódu se samozřejmě můžete v rámci debaty přihlásit, uznáte-li to za vhodné). Stejná pravidla platí také pro nedořešené přípravy (musíte je ale odevzdat).

Tento mechanismus je omezen prostorem ve cvičení – nemůžeme zaručit, že v případě velkého zájmu dojde na všechny (v takovém případě cvičící vybere ta řešení, která bude považovat za přínosnější pro skupinu – je tedy možné, že i když se na Vaše konkrétní řešení nedostane, budete ve cvičení analyzovat podobný problém v řešení někoho jiného).

A.6: Závěrečný test

Zkouškové období tvoří pomyslný 4. blok a platí zde stejné kritérium jako pro všechny ostatní bloky: musíte získat alespoň 50 bodů. Závěrečný test:

- proběhne v počítačové učebně bez přístupu k internetu nebo vlastním materiálům,
- k dispozici budete mít přehled jazyka (ib111.language.pdf a .html) a zabudovanou nápovědu dostupných programů (jiné materiály nejsou povoleny),
- budete moct používat textový editor, interpret jazyka Python a vývojová prostředí Thonny a VS Code.

Na vypracování testu budete mít 4 hodiny čistého času, a bude sestávat ze dvou částí (zadávají a odevzdávají se ovšem společně):

- pět programovacích příkladů, které budou hodnoceny automatickými testy; za každý příklad, který splní testy „verity“ získáte 16 bodů (za všechny ostatní 0), za celkem 0 až 80 bodů,
- deset teoretických otázek, přitom každá bude složená z 5 tvrzení, z toho dvou pravdivých a tří nepravdivých; hodnocení/otázka:
  - -2 body za 2 nepravdivé odpovědi,
  - -1 bod za 1 pravdivou + 1 nepravdivou, nebo za žádnou odpověď,
  - 0 bodů za 1 pravdivou a druhou nevybranou,
  - 2 body za 2 pravdivé odpovědi,

  celkem za -20 až +20 bodů.

Celkový maximální zisk je tedy 100 bodů (80+20). Základní možnosti, jak splnit minimální bodovou hranici, jsou 3 příklady + 2 body za teorii, nebo 2 příklady + 18 bodů za teorii. Nechcete-li se teorií vůbec zabývat, máte také možnost vyřešit 4 příklady (64 - 10 = 54 bodů).

Programovací příklady budou na stejné úrovni obtížnosti jako příklady typu p/r/v ze sbírky.

Během zkoušky můžete kdykoliv odevzdat (na počet odevzdání není žádný konkrétní limit) a vždy dostanete zpět výsledek testů syntaxe a sanity. Součástí zadání bude navíc soubor tokens.txt, kde naleznete 3 kódy. Každý z nich lze použít nejvýše jednou (vložením do komentáře do jednoho z příkladů), a každé použití kódu odhalí výsledek verity testu pro ten soubor, do kterého byl vložen. Toto se projeví pouze při prvním odevzdání s vloženým kódem, v dalších odevzdáních bude tento kód ignorován (bez ohledu na soubor, do kterého bude vložen).

A.6.1 Vnitrosemestrálky V posledním týdnu každého bloku, tedy

- cvičení 4 (20.–24. října),
- cvičení 8 (17.–21. listopadu),
- cvičení 12 (15.–19. prosince),

proběhne v rámci cvičení programovací test na 60 minut. Tyto testy budou probíhat za stejných podmínek, jako výše popsaný závěrečný test (slouží tedy mimo jiné jako příprava na něj). Řešit budete vždy ale pouze dva příklady, přitom za každý můžete získat 3 body (splní-li verity testy) a dále 3 teoretické otázky (hodnoceny jedním bodem za dvě pravdivá tvrzení, jinak nulou). Celkem tak můžete získat 0 až 9 bodů, které se počítají jako aktivita v příslušném bloku. Součástí zadání bude také 1 token pro odhalení výsledku verity testu.

A.7: Opisování

Na všech zadaných problémech pracujte prosím zcela samostatně (zejména tedy bez pomoci spolužáků, třetích stran, nebo jazykových modelů) – toto se týká jak příkladů ze sbírky, které budete odevzdávat, tak domácích úloh ze sad. To samozřejmě neznamená, že Vám zakazujeme společně studovat a vzájemně si pomáhat látku pochopit: k tomuto účelu můžete využít všechny zbývající příklady ve sbírce (tedy ty, které nebude ani jeden z Vás odevzdávat), a samozřejmě nepřeberné množství příkladů a cvičení, které jsou k dispozici online.

Příklady, které odevzdáváte, slouží ke kontrole, že látce skutečně rozumíte, a že dokážete nastudované principy prakticky aplikovat. Tato kontrola je pro Váš pokrok naprosto klíčová – je velice snadné získat pasivním studiem (čtením, poslouchánim přednášek, studiem již vypracovaných příkladů) pocit, že něčemu rozumíte. Dokud ale sami nenapíšete na dané téma několik programů, jedná se pravděpodobně skutečně pouze o pocit.

Abyste nebyli ve zbytečném pokušení kontroly obcházet, nedovolenou spolupráci budeme relativně přísně trestat. Za každý prohřešek Vám bude strženo **v každé instanci** (jeden týden příprav se počítá jako jedna instance, příklady ze sad se počítají každý samostatně):

- 1/2 bodů získaných (ze všech příprav v dotčeném týdnu, nebo za jednotlivý příklad ze sady) zaokrouhleno na celé body nahoru,
- navíc 10 bodů z hodnocení bloku, do kterého opsaný příklad patří,
- konečně 10 bodů (navíc k předchozím 10) z celkového hodnocení.

Opíšete-li tedy například 2 přípravy ve druhém týdnu a:

- Váš celkový zisk za přípravy v tomto týdnu je 5 bodů,
- Váš celkový zisk za první blok je 60 bodů,

jste **automaticky hodnoceni známkou X** (60 - 2,5 - 10 je méně než potřebných 50 bodů). Podobně s příkladem z první sady (60 - 3,5 - 10), atd. Máte-li v bloku bodů dostatek (např. 80 - 5 - 10 ≥ 50), ve studiu předmětu pokračujete, ale započte se Vám ještě navíc penalizace 10 bodů do celkové známky. Přestává pro Vás proto platit pravidlo, že 4 splněné bloky jsou automaticky E nebo lepší.

V situaci, kdy:

- za bloky máte před penalizací 66, 52, 51, 54,
- v prvním bloku jste opsali domácí úkol,

budete penalizováni:

- v prvním bloku 10 + 4, tzn. bodové zisky za bloky budou efektivně 52, 52, 51, 54,
- v celkovém hodnocení 10, tzn. celkový zisk 52 + 52 + 51 + 54 - 10 = 199, a budete tedy hodnoceni známkou **F**.

To, jestli jste příklad řešili společně, nebo jej někdo vyřešil samostatně, a poté poskytl své řešení někomu dalšímu, není pro účely kontroly opisování důležité. Všechny „verze“ řešení odvozené ze společného základu (včetně situace, kdy je tento základ odpovědí jazykového modelu) budou penalizovány stejně. Taktéž **zveřejnění řešení** budeme chápat jako pokus o podvod, a budeme jej trestat, bez ohledu na to, jestli někdo stejné řešení odevzdá, nebo nikoliv.

Podotýkáme ještě, že kontrola opisování **nespadá** do desetidenní lhůty pro hodnocení průběžných kontrol. Budeme se sice snažit opisování kontrolovat co nejdříve, ale odevzdáte-li opsaný příklad, můžete být bodově penalizováni kdykoliv (tedy i dodatečně, a to až do konce zkouškového období).

Část B: Želví grafika

Tato kapitola je náplní cvičení ve druhém týdnu semestru, a jejím smyslem je seznámit Vás s organizací cvičení, se studijními materiály (tedy zejména touto sbírkou), s programovacím prostředím Thonny a se základními elementy syntaxe jazyka Python. Zároveň Vám připomeneme (nebo ukážeme) základy algoritmizace pomocí tzv. želví grafiky.

B.1: Programovací jazyk

V tomto kurzu budeme používat jazyk Python, resp. jeho značně zjednodušenou podobu.[6] V této úvodní kapitole budeme programy zapisovat pouze na intuitivní úrovni: všechny konstrukce, které potřebujete, můžete odvodit z příkladů v ukázkových zdrojových kódech.

Každá další kapitola bude obsahovat sekci, která uvede syntaxi (zápis) a sémantiku (význam, chování) nových jazykových prostředků. Od chvíle, kdy bude nějaký nový prostředek takto uveden, jej můžete ve svých programech využívat.[7] Naopak, nic co nebylo tímto způsobem uvedeno, pro účely tohoto kurzu neexistuje, i když to třeba naleznete na internetu, nebo to znáte z předchozího programování v jazyce Python.

B.2: Přehled příkladů

Jednotlivé kapitoly sbírky obsahují 5 druhů příkladů: první sada jsou tzv. **ukázky** – jedná se o komentované řešení nějakého problému, které Vám ilustruje použití konstrukcí, které v daném týdnu budeme ve cvičení potřebovat. Tyto ukázky **nenahrazují** přednášku, přestože s ní mají určitý překryv – slouží k jejímu doplnění delšími, komentovanými ukázkami použití, které můžete využít jako inspiraci při řešení příkladů z ostatních částí. Tato kapitola obsahuje pět ukázek:

1. square – kreslení čtverce přímo a pomocí cyklu
2. hexagon – použití podprogramu
3. boxes – podprogramy s parametry
4. isosceles – použití proměnné
5. flower – podmíněné provádění kódu

Jak ukázky, tak příklady v dalších sekcích, mohou být označeny dýkou (†): jedná se o složitější příklady, které byste nicméně měli být schopni řešit (i bez dodatečných znalostí). Příklady označené dvojitou dýkou (‡) naopak předbíhají probranou látku, a neumíte-li je vyřešit, není to žádný problém.

Další část obsahuje „elementární“ příklady, které by měly sloužit k tomu, abyste si v rychlosti ověřili, že rozumíte základním konstrukcím a pojmům představeným v přednášce a ukázkách. Vypracovaná řešení této kategorie příkladů naleznete v kapitole R, resp. ve složce sol ve studijních materiálech. Do této kapitoly jsou zařazeny tyto elementární úlohy:

1. pentagon – pravidelný pětiúhelník
2. right – pravoúhlý trojúhelník (parametrický)
3. polygon – pravidelný n-úhelník

Další část tvoří **přípravy**: jsou to příklady, ze kterých si některé vyberete a **samostatně** vyřešíte v předstihu před samotným cvičením k danému tématu. Za tyto příklady dostáváte body, ale pouze pokud **odevzdáte** funkční řešení nejpozději v sobotu před příslušným cvičením.

Přípravy pro tento týden si můžete vyřešit dopředu také – je to ale výjimečně bez bodů:

1. trapezoid – rovnoramenný lichoběžník
2. fence – plot pomocí cyklu
3. spiral – spirála
4. heartbeat – stylizované EKG pomocí cyklu
5. diamond – kreslení stylizovaného diamantu
6. tunnel – soustředné čtverce (pohled do „tunelu“)

Předposlední část každé kapitoly tvoří **řešené** (rozšířené) příklady – tyto mají opět přiložená vzorová řešení. Část jich budete řešit ve cvičeních, část můžete použít pro další domácí přípravu (s možností samostatné kontroly svého řešení vůči tomu vzorovému) nebo také jako zdroj příkladů k procvičení před zkouškou. Tento týden do této kategorie spadají následující příklady:

1. circle – kružnice
2. pizza † – kruhová výseč
3. target – terč (soustředné kružnice)
4. arrow – obrys šipky
5. koch ‡ – Kochova vločka
6. hilbert ‡ – Hilbertova křivka

Kapitolu uzavírají příklady **volitelné**, které nejsou ve sbírce vyřešené, ale na kterých si můžete látku dále procvičovat.

1. house – domeček se střískou
2. star – parametrizovaná hvězda
3. flag – státní vlajka

B.d: Demonstrace (ukázky)

B.d.1 [square] Smyslem první ukázky je předvést základní „příkazy“ (procedury – tento pojem si přesněji vysvětlíme v dalších ukázkách) pro kreslení obrázků. Tyto procedury ovládají „želvu“, která se pohybuje po plátně a kreslí přitom čáru. Procedura forward želvě poručí, aby se posunula o danou vzdálenost vpřed (a nakreslila u toho úsečku ze své původní polohy do své nové polohy). Procedury left a right nic nekreslí, pouze želvou otočí o daný úhel (zadaný v stupních) doleva, resp. doprava.

Dovolíme-li želvě vracet se „po vlastních stopách“, stačí nám tyto 3 procedury na vykreslení libovolného spojitého obrazce. Pro začátek zkusíme nakreslit čtverec:

```
def square():
```

Čtverec lze nakreslit jednoduše jako 4 navazující úsečky stejné délky, přičemž každé dvě po sobě jdoucí svírají pravý úhel.

```
    forward(100)
    right(90)
    forward(100)
    right(90)
    forward(100)
    right(90)
    forward(100)
```

Předchozí definice square nás ale příliš neuspokojuje: k čemu máme počítač, když jsme museli každý krok explicitně popsat? Zejména je na první pohled vidět, že příkazy se opakují. Jistě by bylo dobré, abychom mohli počítači sdělit, že má nějakou akci provést 4×, místo abychom ji zapsali 4× pod sebe – to je v podstatě základní mechanismus, kterým nám počítač šetří práci.

```
def square_loop():
```

Základní formou tzv. **cyklu** (angl. **loop**) je příkaz „proveď akci n krát“, který se v Pythonu zapisuje jako for i in range(n) – v našem případě bude n = 4:

```
    for i in range(4):
```

Následuje tzv. tělo cyklu, které je tvořeno (odsazeným) seznamem příkazů, které se budou opakovat.

```
        forward(100)
        right(90)
```

Pozorný čtenář si jistě všiml, že definice square a square_loop nejsou

[6] Nicméně bude vždy platit, že programy, které v tomto kurzu naprogramujete, jsou plnohodnotné programy ve skutečném (neomezeném) jazyce Python. Nemusíte se tedy bát, že byste znalosti, které se tu naučíte, nevyužili v praxi.

[7] V sadách domácích úloh se budou objevovat zadání, která využívají jazyk ze začátku bloku – i v případě, když takovou úlohu začnete řešit později, platí omezení jazyka na týden uvedený v záhlaví zadání.

zcela ekvivalentní: ta druhá obsahuje jedno použití procedury right navíc. Pro tuto chvíli je nám to jedno, protože není-li volání right následováno žádným použitím forward, nebude mít na výsledný obrázek dopad. Nicméně obecně toto neplatí a je potřeba si na podobné **okrajové případy** dávat pozor.

Následuje definice main, smyslem které je demonstrovat funkčnost dříve definovaných square a square_loop.

```
def main():  # demo
```

Nejprve necháme želvu vykreslit čtverec „naivním“ způsobem, bez použití cyklu (první z definic výše).

```
    square()
```

Dále želvu požádáme, aby se přesunula na jiné místo plátna, aniž by nakreslila čáru: tento kus kódu pro nás není příliš podstatný, jeho smyslem je pouze vykreslit dva obrázky na jedno plátno, abychom je mohli lehce srovnat.

```
    penup()
    setheading(0)
    forward(200)
    pendown()
```

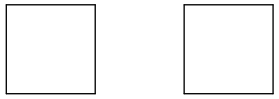
Na novém místě plátna požádáme želvu o vykreslení čtverce druhou metodou (cyklem). Jestli jsme se nespletli, budou oba obrázky identické.

```
    square_loop()
```

Příkazem (procedurou) done želvě oznámíme, že máme vše vykresleno a program má vyčkat na ukončení uživatelem.

```
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.d.2 [hexagon] V této ukázce sestrojíme „segmentovaný“ šestiúhelník složením z 6 pootočených rovnostranných trojúhelníků. Smyslem je ukázat, že část výpočtu si můžeme pojmenovat, a poté ji s výhodou využít jako stavební kámen něčeho složitějšího. V tomto případě se vybízí pojmenovat si právě vykreslení onoho rovnostranného trojúhelníku:

```
def triangle():
    for i in range(3):
        forward(100)
        left(120)
```

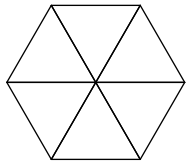
To, co jsme právě udělali, se obecně jmenuje **definice podprogramu**. V tomto případě se jedná konkrétně o **proceduru**, totiž podprogram, kterého smyslem je provést nějaké **akce** (vedlejší efekty). V našem případě je tedy triangle procedurou pro vykreslení rovnostranného trojúhelníku. Naše nově definovaná procedura triangle je k nerozeznání od těch zabudovaných (knihovních), které známe z předchozí ukázky: left, forward a pod.

```
def hexagon():
    for i in range(6):
        triangle()
        left(360.0 / 6)
```

Teď již víme, že main je také procedura, tedy podprogram, kterého smyslem je vykonat posloupnost akcí (typicky dalších procedur).

```
def main():  # demo
    speed(5)
    hexagon()
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.d.3 [boxes] Procedura, kterou jsme definovali v předchozí ukázce, totiž taková, která provede fixní (pokaždé stejnou) posloupnost akcí, není příliš zajímavá. Naštěstí lze procedury **parametrizovat**. Podobně jako u knihovních procedur forward nebo left si můžeme sami definovat proceduru, které pak při použití předáme nějaké číslo (obecněji **hodnotu**). Konkrétní předaná hodnota pak bude mít vliv na chování takto definované procedury.

Zde si definujeme proceduru square, která se nápadně podobá na proceduru square_loop z první ukázky, s jedním rozdílem: délka strany již není pevně daná, ale je nyní proceduře předána jako **parametr**.

```
def square(size):
    for i in range(4):
        left(90)
        forward(size)
```

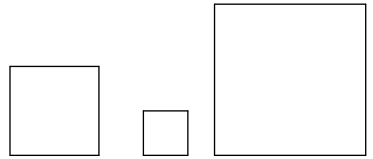
Takto definovanou proceduru můžeme opět používat zcela analogicky k těm zabudovaným – nyní včetně předání parametru, který diktuje, jak velký čtverec si přejeme vykreslit.

```
def main():  # demo
    speed(5)
    square(100)
```

Připomínáme, že následující tři příkazy slouží pouze k přesunu želvy na jinou pozici na plátně.

```
    penup()
    forward(100)
    pendown()

    square(50)

    penup()
    forward(200)
    pendown()

    square(170)

    done()
```

Výstup testů by měl vypadat přibližně takto:



B.d.4 [isosceles] Doposud jsme se nezabývali otázkou, odkud pochází definice procedur left, forward apod. Protože ale v této ukázce budeme potřebovat další knihovní podprogramy, je čas zmínit existenci příkazu import. Tím oznámíme interpretu Pythonu, že hodláme využívat podprogramy z externích **modulů**. V tomto kurzu se omezíme na moduly ze **standardní knihovny**, totiž takové, které jsou dodávány s každým interpretem jazyka Python.

Pro úplnost dodáme, že **modul** je sbírka vzájemně souvisejících, znovupoužitelných podprogramů (a případně i složitějších artefaktů, kterými se ale nebudeme v tomto kurzu příliš zabývat).

```
from turtle import forward, left, penup, pendown, done, \
    setheading, speed
```

Krom procedur pro práci se želvou budeme v tomto příkladu potřebovat několik matematických **funkcí**:

- odmocninu, realizovanou podprogramem sqrt,

- převod stupňů na radiány, realizovaný podprogramem radians,
- goniometrickou funkci **tangens**, realizovanou podprogramem tan.

Podprogramům, které realizují výpočet nějaké hodnoty na základě hodnot svých parametrů, budeme říkat **čisté funkce**, z důvodu jejich podobnosti s funkcemi z matematiky. Podprogramy sqrt, radians a tan jsou tedy v tomto smyslu (čistými) funkcemi.

```
from math import sqrt, radians, tan
```

Krom použití **funkcí** si v této ukázce předvedeme také použití **proměnných**. V nejjednodušším smyslu je proměnná pouze pojmenováním nějaké vypočtené hodnoty – takto je budeme nyní používat. Složitější případy použití proměnných (zejména **přiřazení**) si necháme na příští týden.

Obrázek, který budeme kreslit, je **rovnoramenný trojúhelník**, zadaný délkou základny a úhlem (v stupních) mezi základnou a ramenem.

```
def isosceles(base, angle):
```

První hodnotou, kterou si pojmenujeme (uložíme do proměnné) bude polovina základny: rovnoramenný trojúhelník si totiž pomyslně rozdělíme na dva stejné (pouze zrcadlově otočené) pravoúhlé trojúhelníky s odvěsnami height (výška) a half_base (polovina základny).

```
    half_base = float(base) / 2
```

Protože trojúhelník máme zadaný základnou a přilehlým úhlem, potřebujeme vypočítat délku ramene. To se nejsnadněji provede pomocí už zmíněného pomyslného pravoúhlého trojúhelníku. Na výpočet délky ramene použijeme Pythagorovu větu, ale nejprve potřebujeme znát výšku (druhou z odvěsen pomyslného trojúhelníku). Protože máme úhel zadaný v stupních, musíme ho nejprve převést na radiány, pak jednoduše použijeme funkci tangens, která udává poměr odvěsen v pravoúhlém trojúhelníku (protilehlá k přilehlé). Výšku získáme jednoduchou úpravou definičního výrazu.

```
    height = half_base * tan(radians(angle))
```

Konečně můžeme přistoupit k výpočtu délky ramene:

```
    side = sqrt(height ** 2 + half_base ** 2)
```

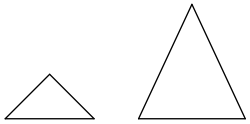
Nyní máme vše, co k vykreslení potřebujeme. Nejprve nakreslíme základnu, poté želvu otočíme o **vedlejší úhel** k angle (tak, aby úhel sevřený základnou a ramenem, které budeme kreslit jako další byl angle). Vrcholový úhel je daný vztahem 180 - 2 * angle, nicméně opět potřebujeme želvu otočit o příslušný vedlejší úhel (hodnotu 2 * angle dostaneme opět jednoduchou úpravou). Nakonec vykreslíme druhé rameno, a želva se tím vrátí do výchozí pozice.

```
    forward(base)
    left(180 - angle)
    forward(side)
    left(2 * angle)
    forward(side)
```

Abychom ověřili, že program pracuje správně, vykreslíme si dva různé trojúhelníky.

```
def main():  # demo
    speed(5)
    isosceles(100, 45)

    penup()
    setheading(0)
    forward(150)
    pendown()

    isosceles(120, 65)
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.d.5 **[flower]** Tato (pro tento týden poslední) ukázka předvede použití příkazu if, který slouží k podmíněnému vykonání nějaké akce. Nejprve si ale definujeme pomocnou proceduru triangle, která by nás již neměla překvapit: vykresluje tupoúhlý, rovnoramenný trojúhelník, který bude sloužit jako lupínek květiny. Důležitou vlastností této procedury je, že zachová pozici i orientaci želvy.

```
def triangle():
    forward(100)
    right(165)
    forward(52)
    right(30)
    forward(52)
    right(165)
```

Vykreslíme nyní stylizovanou květinu, které ale chybí některé lupínky: konkrétně ty, jejichž pořadové číslo je dělitelné třemi nebo pěti. Květinu budeme vykreslovat v cyklu, jak už je zvykem. To, čím se tato ukázka liší od předchozích, je, že samotná posloupnost akcí, které se v těle cyklu provedou, se bude iteraci od iterace lišit. Parametr nám zadává původní počet lupínků (kolik by jich bylo, kdyby žádný nechyběl).

```
def flower(petals):
    for i in range(petals):
```

Podmínku zapisujeme klíčovým slovem if, následovaným výrazem, který se vyhodnotí na booleovskou hodnotu (tzn. True nebo False) a za dvojtečkou seznamem příkazů, které se provedou **pouze**, vyhodnotil-li se předaný výraz na hodnotu True (tzn. byl pravdivý).

V tomto případě se dotazujeme, zda má indexová proměnná i nenulový zbytek po dělení jak číslem 3 tak číslem 5: znamená to, že ani jeden z nich není dělitelem. Všimněte si, že podmínku pro „chybějící“ lupínek jsme negovali: lupínek vykreslíme, je-li tato (negovaná) podmínka splněna, tedy bude chybět v případě, že byla splněna původní podmínka ze zadání.

Budete-li srovnávat zápis programu s obrázkem, který kreslí, je důležité si uvědomit, že první index je 0 (a je tedy dělitelný například i 3), nultý lupínek bude tedy chybět. Kdyby nechyběl, „ukazoval“ by směrem doprava.

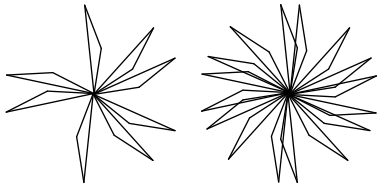
```
        if i % 3 != 0 and i % 5 != 0:
            triangle()
```

Bez ohledu na to, zda jsme lupínek vykreslili nebo nikoliv, musíme se pootočit k vykreslení (nebo přeskočení) dalšího lupínku: tento příkaz se provede v každé iteraci. Protože se pootočíme doprava, lupínky vykreslujeme ve směru hodinových ručiček (přičemž nultý by ukazoval 3 hodiny) – ve stejném směru, kterým ukazují vrcholy trojúhelníků, které lupínky reprezentují.

```
        right(360.0 / petals)
```

```
def main():  # demo
    speed(10)
    flower(15)

    penup()
    setheading(0)
    forward(220)
    pendown()

    flower(30)
    done()
```

Výstup testů by měl vypadat přibližně takto:



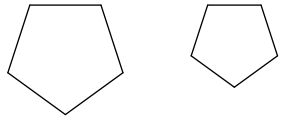
B.e: Elementární příklady

B.e.1 **[pentagon]** Implementujte proceduru pentagon, která vykreslí pra-

videlný pětiúhelník se stranami o délce side pixelů.

```
def pentagon(side):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.e.2 [right] Implementujte proceduru right_triangle, která vykreslí pravoúhlý trojúhelník s odvěsnami o délkách side_a a side_b. Můžou se vám hodit funkce z modulu math.

```
def right_triangle(side_a, side_b):
    pass
```

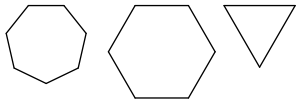
Výstup testů by měl vypadat přibližně takto:

B.e.3 [polygon] Zobecněte řešení z příkladu pentagon tak, abyste byli schopni vykreslit libovolný pravidelný mnohoúhelník. Toto obecné řešení implementujte jako proceduru polygon s parametry:

- sides je počet stran kresleného mnohoúhelníku, a
- length je délka každé z nich.

```
def polygon(sides, length):
    pass
```

Výstup testů by měl vypadat přibližně takto:

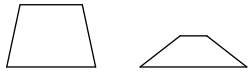


B.p: Přípravy

B.p.1 [trapezoid] Nakreslete rovnoramenný lichoběžník s délkami základen base_length a top_length a výškou height (lichoběžník je čtyřúhelník s jednou dvojicí rovnoběžných stran – základen – spojených rameny, které jsou obecně různoběžné).

```
def trapezoid(base_length, top_length, height):
    pass
```

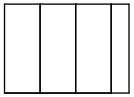
Výstup testů by měl vypadat přibližně takto:



B.p.2 [fence] Napište program, který nakreslí „plot“ o délce length pixelů, složený z prken (obdélníků) o šířce plank_width a výšce plank_height. Přesahuje-li poslední prkno požadovanou délku plotu, ořežte jej tak, aby měl plot přesně délku length. Zamyslete se nad rozdělením vykreslování do několika samostatných procedur. Při kreslení se vám také může hodit while cyklus.

```
def fence(length, plank_width, plank_height):
    pass
```

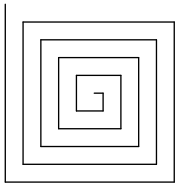
Výstup testů by měl vypadat přibližně takto:



B.p.3 [spiral] Implementujte proceduru spiral, která vykreslí čtyřhrannou spirálu s rounds otočeními (počet otočení říká, kolik hran musíme překročit, vydáme-li se ze středu spirály po přímce libovolným směrem). Parametr step pak udává počet pixelů, o který se hrany postupně prodlužují.

```
def spiral(rounds, step):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.p.4 [heartbeat] Implementujte proceduru heartbeat, která vykreslí stylizovanou křivku EKG. Parametr iterations udává počet tepů, které procedura vykreslí. Zbylé parametry zadávají amplitudu základního úderu a periodu slabšího úderu. Slabší úder má poloviční amplitudu. Například při periodě 3 bude mít sníženou amplitudu každý třetí úder, počínaje prvním.

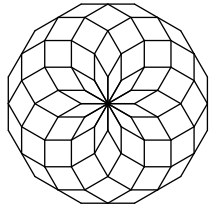
```
def heartbeat(amplitude, period, iterations):
    pass
```

Výstup testů by měl vypadat přibližně takto:

B.p.5 [diamond] Napište proceduru pro vykreslení stylizovaného diamantu. Tento se skládá z mnohoúhelníků, které jsou vůči sobě natočené o vhodně zvolený malý úhel (takový, aby byl výsledný obrazec pravidelný). Každý mnohoúhelník má sides stran o délce length pixelů.

```
def diamond(sides, length):
    pass
```

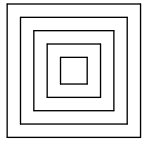
Výstup testů by měl vypadat přibližně takto:



B.p.6 [tunnel] Napište proceduru, která nakreslí „tunel“ – sekvenci soustředných čtverců, kde vnější má stranu délky size a každý další je o step jednotek menší.

```
def tunnel(size, step):
    pass
```

Výstup testů by měl vypadat přibližně takto:

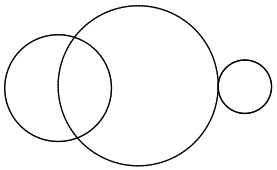


B.r: Řešené úlohy

B.r.1 [circle] Pomocí procedury pro mnohoúhelníky si nejprve zkuste vykreslit kružnici. Poté napište proceduru pro vykreslení kružnice o zadaném poloměru radius. (Nápověda: srovnejte obvod kružnice a pravidelného n-úhelníku). Kružnici nakreslete tak, aby její střed ležel v bodě, ve kterém byla želva před použitím procedury circle. Pro vypnutí a zapnutí kreslení použijte procedury penup a pendown. Po dokreslení kružnice vraťte želvu zpět do jejího středu.

```
def circle(radius):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.r.2 [pizza] † Nakreslete kruhovou výseč („dílek pizzy“) se středovým úhlem zadaným (v stupních) parametrem angle a délkou strany side.

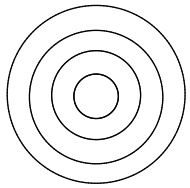
```
def pizza(side, angle):
    pass
```

Výstup testů by měl vypadat přibližně takto:

B.r.3 [target] Napište proceduru, která bude kreslit soustředné kružnice, a to tak, že první má poloměr radius a zbytek je rovnoměrně rozložen tak, aby bylo kružnic celkem count.

```
def target(radius, count):
    pass
```

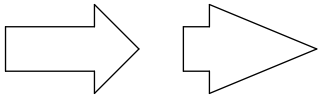
Výstup testů by měl vypadat přibližně takto:



B.r.4 [arrow] Nakreslete obrys šipky zadaných rozměrů (celková šířka width a celková výška height) a s úhlem špičky angle. Šipka by měla ukazovat v původním směru želvy. Želva nechť je po konci procedury ve stejné pozici a orientaci jako před jejím začátkem.

```
def arrow(width, height, angle):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.r.5 [koch] ‡ Pozor! Tento a následující příklad jsou založeny na rekurzi, kterou budeme probírat až na konci kurzu. Nemusíte si tedy lámat hlavu, pokud je neumíte vyřešit.

Nakreslete Kochovu vločku, která má stranu o délce size. Parametr depth udává kolikrát se má provést dělení strany vločky. Konstrukce začíná rovnostranným trojúhelníkem, přičemž vločka vzniká opakovanou aplikací následovného postupu na všechny úsečky, které v daném okamžiku tvoří obrazec:

1. vybranou stranu rozdělte na třetiny a prostřední část odstraňte,
2. nad prostřední částí sestrojte rovnostranný trojúhelník bez základny: danou stranu jste tak nahradili sekvencí 4 úseček: 2 zbývající krajní třetiny původní strany a 2 ramena přidaného trojúhelníku,

Daná iterace končí rozdělením poslední úsečky, která vznikla v iteraci předchozí. Proveďte celkem depth iterací. Testy vykreslují vločku hloubky dělení (počet iterací) 0 až 3.

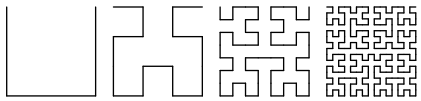
```
def koch_snowflake(size, depth):
    pass
```

Výstup testů by měl vypadat přibližně takto:

B.r.6 [hilbert] ‡ Nakreslete Hilbertovu křivku se stranou délky size a počtem dělení iterations. Hilbertova křivka vzniká, podobně jako Kochova vločka, opakovaným dělením stávajícího obrazce na zmenšené kopie sebe sama. Podrobnější návod, jak křivku nakreslit (na papír), naleznete na adrese https://is.muni.cz/go/9fh9k4.

```
def hilbert(size, iterations):
    pass
```

Výstup testů by měl vypadat přibližně takto:

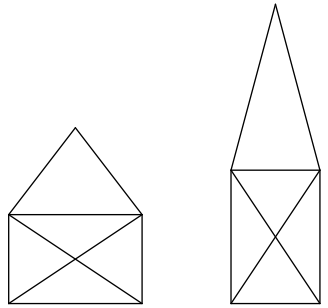


B.v: Volitelné úlohy

B.v.1 [house] Nakreslete domeček „jedním tahem“ (viz obrázky níže). Obdélníková část domečku má šířku width a výšku height (kladná reálná čísla), úhel špičky střechy je roof_angle stupňů (v rozsahu 1 až 179).

```
def house(width, height, roof_angle):
    pass
```

Výstup testů by měl vypadat přibližně takto:

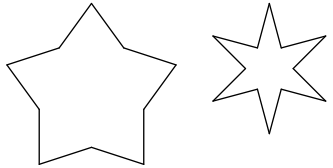


B.v.2 [star] Nakreslete hvězdu (viz obrázky níže) s points paprsky. (Počet paprsků je kladné celé číslo větší než 2). Paprsky hvězdy jsou tvořeny rovnoramennými trojúhelníky bez základny, jejichž výška je size (kladné číslo) a úhel svíraný rameny je angle (v rozsahu 1 až 179). Paprsky jsou rovnoměrně rozmístěny do kruhu. Jeden z paprsků vždy směřuje na sever.

Poznámka: S extrémními hodnotami parametrů může výsledná „hvězda“ spíše připomínat zakulacený mnohoúhelník nebo ozubené kolo.

```
def star(points, angle, size):
    pass
```

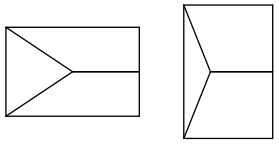
Výstup testů by měl vypadat přibližně takto:



B.v.3 [flag] Nakreslete obrys vlajky s klínem vlevo (viz obrázky níže). Parametry width a height (kladná reálná čísla) označují šířku, resp. výšku vlajky. Parametr triangle_ratio (reálné číslo mezi 0 a 1 včetně) označuje, do jaké části šířky vlajky má zasahovat její klín.

```
def flag(width, height, triangle_ratio):
    pass
```

Výstup testů by měl vypadat přibližně takto:



Část 1: If, cykly, proměnné

První kapitola sbírky slouží k procvičení látky z první přednášky – tento princip bude v platnosti celý semestr.

Připomínáme, že příklady ze sekce příprav jsou **bodované** a v každém čtyřtýdenním bloku **musíte** získat celkem alespoň 60 bodů (jakou část získáte za přípravy je už nicméně na Vás). Abyste získali za přípravy body, musíte je odevzdat vždy do **soboty 23:59** (tento týden 28.9.). Detailněji jsou pravidla popsána v části A.

Tento týden se budeme zabývat zejména tzv. **tokem řízení** (anglicky **control flow**) – téma, které jsme načali už v nultém týdnu. Jedná se zejména o konstrukci podmíněného vykonání kódu (příkaz `if`) a o konstrukce pro opakované spuštění sekvence příkazů (příkazy `for`, `while`). V menší míře se budeme zabývat také **proměnnými** – pojmenovanými **hodnotami**, vhodnými pro pozdější (případně vícenásobné) použití.

V ukázkách si na příkladech vysvětlíme již zmiňované základní konstrukce (teorii již znáte z přednášky). Ukázky označené znakem † jsou náročnější – pravděpodobně se u nich budete muset více soustředit. Nepovede-li se Vám takovou ukázku rozluštit napoprvé, zkuste ji na pár dnů odložit, a vrátit se k ní později (poté, co se Vám látka pro daný týden více rozleželа v hlavě a již jste si vyřešili pár příkladů).

1. `triangle` – návratové hodnoty podprogramů, funkce
2. `sum` – použití indexů v cyklech
3. `fibonacci` – přepis matematické posloupnosti do algoritmu
4. `cycle` – použití podmíněného příkazu
5. `converge` † – výběr podposloupnosti

Dále máte k dispozici několik elementárních příkladů, na kterých si můžete nové konstrukce rychle procvičit:

1. `divisors` – zjištění počtu dělitelů čísla použitím cyklu
2. `powers` – součet po sobě jdoucích k-tých mocnin
3. `multiples` – počítání násobků

Dalším krokem jsou samozřejmě již zmiňované přípravy. Ty, které hodláte odevzdat, vypracujte **zcela samostatně**, u těch zbývajících můžete pracovat způsobem, který Vám nejvíce vyhovuje: samostatně, probrat myšlenku se spolužáky, ale naprogramovat každý sám, dokonce si můžete vzájemně pomáhat i se samotným zápisem kódu. Ujistěte se ale, že **v žádném případě** neodevzdáváte příklad, se kterým Vám někdo pomáhal, a nepomáhejte spolužákům s příklady, které sami hodláte odevzdat!

1. `sequence` – n-té číslo posloupnosti s parametry
2. `nested` – vnořené posloupnosti
3. `triples` – největší pythagorejská trojice
4. `geometry` – predikáty trojúhelníkových vlastností
5. `fibsum` – suma sudých členů Fibonacciho posloupnosti
6. `next` – výpočet následujícího většího násobku

V předposlední sekci jsou rozšířené příklady: některé z nich si vyřešíte příští týden na cvičení, ostatní můžete řešit se spolužáky nebo samostatně jako přípravu na zkoušku. K těmto příkladům naleznete v kapitole K vzorová řešení: silně Vám ale doporučujeme na řešení se nedívat, dokud příklad nemáte vyřešený, nebo jste se u něj vysloveně nezasekli.

1. `even` – součet sudých mocnin
2. `prime` – kontrola prvočíselnosti
3. `coins` – minimální počet mincí pro hodnotu
4. `fibfibsum` † – použití posloupnosti k indexaci
5. `abundant` † – vlastnosti čísel a jejich dělitelů
6. `amicable` † – vlastnosti dvojic čísel

Poslední částí jsou tzv. volitelné příklady. Ty si můžete vypracovat dle libovůle samostatně nebo ve skupině, na rozdíl od příkladů typu `r` však k těmto příkladům řešení nepřikládáme.

1. `lvseq` – n-tý prvek jednoduché parametrické posloupnosti
2. `dnsum` – součet dvouciferných čísel s podmínkou
3. `path` – největší číslo v zadané posloupnosti

1.1: Programovací jazyk

Jak jsme již v předchozí kapitole zmínili, v tomto kurzu budeme programovat v omezené podmnožině jazyka Python. Každá kapitola v úvodní části představí všechny jazykové prostředky, které dosud neznáte.

1.1.1 Výrazy

Výrazy v Pythonu intuitivně odpovídají výrazům, které znáte z matematiky: skládají se z **konstant**, **proměnných**, **operátorů**, závorek a **volání funkcí** (o funkcích detailněji níže). Každý výraz má **hodnotu**, a smyslem výrazů je kompaktně popsat výpočet této hodnoty. Příklady:

```
a + 7
4
3 + 3 * 2
(3 + 3) * 2
a + 1 > 7
count ** 2 < 100
```

K dispozici máme tyto základní **binární** operátory (mají vždy dva operandy):

- aritmetické (odpovídají obvyklým matematickým operacím):
  - `a + b`, `a - b` – sčítání a odečítání,
  - `a * b` – násobení,
  - `a // b`, `a % b` – celočíselné dělení a zbytek po dělení (připouštíme pouze pro dva celočíselné operandy),
  - `a / b` – dělení s desetinným výsledkem (naopak připouštíme pouze v případě, kdy alespoň jedno z `a`, `b` je číslo s plovoucí desetinnou čárkou – `float`),
  - `a ** b` – mocnění a^b,
- relační (význam opět známe z matematiky):
  - `a == b` – rovnost,
  - `a != b` – různost / nerovnost,
  - `a > b`, `a < b` – ostré nerovnosti,
  - `a >= b`, `a <= b` – neostré nerovnosti,
- logické (odpovídají logickým spojkám):
  - `a and b` – logická konjunkce: platí `a` a `b` zároveň (vyhodnotí-li se `a` na `False`, podvýraz `b` **nebude vůbec vyhodnocen** protože již nemůže výsledek ovlivnit),
  - `a or b` – logická disjunkce: platí alespoň jedno z `a`, `b` (podobně, vyhodnotí-li se `a` na `True`, podvýraz `b` se **nevyhodnocuje**).

Navíc jsou k dispozici dva **unární** operátory (mají pouze jeden operand):

- `-a` – opačná hodnota,
- `not a` – logická negace.

Výrazem je také tzv. **ternární operátor**, který má podobu `x if cond else y` – vyhodnotí-li se podvýraz `cond` na pravdivou hodnotu, celý výraz se vyhodnotí na výsledek podvýrazu `x`, v opačném případě na výsledek `y` (nepoužitý podvýraz se **nevyhodnocuje**).

Několik dalších operátorů (resp. nových významů stejných operátorů) ještě přibude v příštích týdnech.

1.1.2 Příkazy

Dalším stavebním prvkem programu je **příkaz**, který odpovídá pokynu k provedení nějaké akce. Nejjednodušším příkazem je **libovolný výraz** (užitečnost takových příkazů úzce souvisí s funkcemi, které nejsou čisté, obzvláště pak s **procedurami**). Efektem takového příkazu je, že program vypočte jeho hodnotu a pak ji zapomene.

Druhým základním typem příkazu je **přiřazení**, které podobně jako v předchozím případě **vypočte hodnotu** výrazu, ale na rozdíl od předchozího si ji zároveň **zapamatuje** a **pojmenuje**. Takto pojmenovanou hodnotu – **proměnnou**

– pak můžeme s výhodou použít v pozdějších výrazech.[8] V obou případech platí, že 1 řádek = 1 příkaz.

Přiřazení zapisujeme jako jméno = výraz, například:

```
a = 2
b = a + 1
b = -b
average = (a + b) / 2
positive = a > 0
```

Krom obyčejného přiřazení můžeme použít ještě tzv. **složené přiřazení**, které umožňuje zápis některých častých operací zkrátit. Tato složená přiřazení zapisujeme (věnujte pozornost závorkám a rozdílu mezi / a //):

složené přiřazení	ekvivalentní zápis
a += 2	a = a + 2
x -= 2 * b	x = x - (2 * b)
a *= b + 2	a = a * (b + 2)
x /= a + b	x = x / (a + b)
x //= 3	x = x // 3

Pozor! Znak = v přiřazení **není operátor** a přiřazení **není výraz** – např. zápis (a = b) + 3 nepřipouštíme.

Posledním typem příkazu, který zde uvedeme, je tzv. **tvrzení**, které vyhodnotí zadaný výraz a je-li tento pravdivý, neudělá nic. V opačném případě ukončí program s chybou. Příklad:

```
assert x > 0
```

Tento příkaz budete prozatím potkávat zejména v přiložených testech.

1.1.3 Řízení toku Krom výpočtu a zapamatování si hodnot potřebujeme pro zápis algoritmů ještě **rozhodování** a **opakování**. K tomu slouží **příkazy toku řízení**, konkrétně if, for a while.

Příkaz if realizuje rozhodnutí na základě **pravdivostní hodnoty** (výrazu). Nejjednodušší forma je:

```
if podmínka₁:
    příkaz₁
    …
    příkazₙ
```

Význam tohoto zápisu je: vypočti hodnotu **výrazu** podmínka$_1$ a je-li výsledek pravdivý, proveď **příkazy** příkaz$_1$ až příkaz$_n$, jinak nedělej nic (výpočet pak pokračuje dalším příkazem v sekvenci). Příkaz if lze rozšířit o tzv. else větev:

```
if podmínka₁:
    příkazy₁
else:
    příkazy₂
```

který se chová stejně, ale v případě, že podmínka splněna nebyla, ještě vykoná příkazy z posloupnosti příkazy$_2$. Konečně nejobecnější podoba podmíněného příkazu je (vpravo ekvivalentní zápis pomocí výše uvedené formy):

```
if podmínka₁:          · if podmínka₁:
    příkazy₁           ·     příkazy₁
elif podmínka₂:        · else:
    příkazy₂           ·     if podmínka₂:
                       ·         příkazy₂
elif podmínka₃:        ·     else:
    příkazy₃           ·         if podmínka₃:
                       ·             příkazy₃
else:                  ·         else:
    příkazy₄           ·             příkazy₄
```

přičemž větví elif může být libovolný počet.

Pro **opakování** nějaké posloupnosti příkazů slouží **cykly**, které jsou dvojího typu: for a while. Cyklus for použijeme v případě, kdy předem známe počet **iterací** (opakování), které chceme provést:

```
for jméno in rozsah:
    příkazy
```

kde rozsah může být:

- range(počet) – vypočte hodnotu **výrazu** počet a provede sekvenci příkazy právě počet-krát (jméno je v i-té iteraci vázáno na hodnotu i),
- range(od, do) – vypočte hodnoty n_1, n_2 **výrazů** od, do a provede sekvenci příkazy pro hodnoty $i \in \langle n_1, n_2)$ (jméno je přitom opět vázáno na hodnotu i),
- range(od, do, krok) – podobně jako předchozí, ale provede sekvenci pro hodnoty $i \in I \cap \{n_1 + js \mid j \in \mathbb{N}_0\}$ kde:
  - s je výsledek vyhodnocení výrazu krok,
  - I je $\langle n_1, n_2)$ pro $n_1 \leq n_2$ nebo $(n_2, n_1\rangle$ jinak

  a jméno je vázáno na hodnoty i v **pořadí stoupajícího** j.

Naopak cyklus while použijeme v situaci, kdy umíme **výrazem** popsat, chceme-li **provést další iteraci**:

```
while podmínka:
    příkazy
```

nejprve vyhodnotí **výraz** podmínka. Je-li hodnota pravdivá, provede příkazy a výraz podmínka **opět vyhodnotí**. Cyklus je ukončen v okamžiku, kdy se podmínka vyhodnotí jako nepravdivá (v takovém případě už se příkazy neprovedou, může tedy nastat situace, kdy se příkazy neprovedou **ani jednou**).

Kdekoliv v **těle cyklu** (ale nikde jinde) se mohou objevit ještě příkazy break a continue (vztahují se k rozsahem nejmenšímu cyklu, v kterého těle jsou obsaženy – tzn. k „nejvnitřnějšímu“ aktivnímu cyklu) a mají následovný význam:

- continue okamžitě ukončí probíhající iteraci: program pokračuje další iterací (není-li to možné, cyklus je na tomto místě ukončen),
- break okamžitě ukončí vykonávání cyklu.

1.1.4 Podprogramy Podprogramy jsou základním stavebním prvkem složitějších programů. Podprogram (v Pythonu také zvaný **funkce**) zastřešuje ucelený úsek kódu, který má navíc název, parametry a návratovou hodnotu. Podprogram **definujeme** následujícím zápisem:

```
def podprogram(parametr₁, parametr₂, …, parametrₙ):
    příkaz₁
    …
    příkazₙ
```

kde podprogram je **jméno**, parametr$_1$ až parametr$_n$ jsou **jména** tzv. **formálních parametrů** a příkaz$_1$ až příkaz$_n$ jsou sekvencí příkazů, které tvoří tzv. **tělo podprogramu**.

V podprogramu se krom už známých příkazů může objevit příkaz return výsledek, který jeho vykonávání **ukončí** a určí **návratovou hodnotu** (výsledek), kterou získá vyhodnocením **výrazu** výsledek.

Chceme-li již definovaný podprogram (funkci) použít, slouží k tomu tzv. **volání funkce**. Volání je **výraz**, a zapisuje se následovně:

```
podprogram(výraz₁, výraz₂, …, výrazₙ)
```

Zde podprogram je **jméno** a výraz$_1$ až výraz$_n$ jsou tzv. **skutečné parametry**. Protože se jedná o výraz, má **hodnotu**, která odpovídá návratové hodnotě podprogramu (příkazu return, kterým byl ukončen). S touto hodnotou můžeme pracovat jako s libovolným jiným výrazem:

```
výsledek₁ = funkce(3, 4)
výsledek₂ = 1 + 2 * funkce(3, 4)
výsledek₃ = funkce(funkce(1, 2), 3)
```

Význam použití podprogramu (volání funkce) je následovný:

- jménům ze seznamu formálních parametrů jsou přiřazeny **hodnoty**, které vzniknou vyhodnocením výrazů výraz$_1$ až výraz$_n$,
- provede se **tělo** podprogramu (sekvence příkazů příkaz$_1$ až příkaz$_n$),

[8] Samotné přiřazení nijak s hodnotami nemanipuluje, zejména je nevytváří ani nekopíruje. Význam přiřazení je skutečně pouze **pojmenování** hodnoty, která už musí existovat (obvykle jako výsledek vyhodnocení výrazu). Prozatím tento rozdíl není příliš důležitý – na chování programů začne mít dopad až ve třetí kapitole, kdy do jazyka přidáme složené typy. **Pozor:** některé programovací jazyky dávají přiřazení úplně jiný význam!

- **návratová hodnota** se použije jako výsledek celého podvýrazu volání funkce a pokračuje se vyhodnocováním celého výrazu, ve kterém bylo volání obsaženo.

1.1.5 Zabudované podprogramy Krom podprogramů, které si sami definujete, můžete využívat několik takových, které jsou v jazyce **zabudované** (jsou součástí jazyka). Seznam těchto podprogramů budeme během semestru postupně rozšiřovat. Prozatím jsou to tyto (všechny zde uvedené podprogramy jsou zároveň **čisté funkce**):

- `min(a, b)` a `max(a, b)`: vybere nejmenší, resp. největší hodnotu mezi svými parametry,
- `abs(x)`: spočte absolutní hodnotu parametru `x`,
- `round(x)`: pro desetinné číslo `x` se vyhodnotí na nejbližší celé číslo (hodnoty přesně mezi se zaokrouhlí na nejbližší sudé číslo),
- `float(x)`: pro celé číslo `x` se vyhodnotí na odpovídající číslo s plovoucí desetinnou čárkou (v případě, že konverzi provést nelze, protože `x` příliš velké, je program ukončen s chybou).

Dále máte k dispozici **proceduru** `print`, kterou si můžete pomoct při programování, ale kterou jinak v tomto kurzu budeme potřebovat jen výjimečně.

1.1.6 Knihovny Pomocí příkazu (píšeme vždy na začátek programu)

`from module import name`$_1$, `name`$_2$, …
můžeme požádat o zpřístupnění podprogramů nebo konstant `name`$_1$, `name`$_2$ atd. z **knihovny** `module`. V této chvíli můžete používat pouze tyto **čisté funkce**, které realizují výpočet **funkcí** v matematickém smyslu, a konstanty z knihovny `math`:

- `pi` – číslo π (poměr obvodu a průměru kružnice),
- goniometrické a cyklometrické funkce:
  - `cos(x)`, `sin(x)`, `tan(x)` – známé goniometrické funkce (parametr `x` je zadán v **radiánech**),
  - `acos(x)`, `asin(x)` – cyklometrické (inverzní trigonometrické) funkce, vstupem je reálné číslo intervalu $\langle -1, 1\rangle$ a výsledkem je odpovídající úhel z intervalu $\langle 0, \pi\rangle$,
  - `atan(x)` – inverzní funkce k funkci `tan` (vstupem je libovolné reálné číslo, výsledkem úhel z intervalu $(-\pi/2, \pi/2)$,
  - `atan2(y, x)` – úhel svíraný x-ovou osou a polopřímkou z počátku, která prochází bodem (x, y), v rozsahu $(-\pi, \pi\rangle$,
- funkce pro převod úhlů:
  - `radians(x)` – stupně na radiány a
  - `degrees(x)` – radiány na stupně,
- funkce pro výpočet kořenů:
  - `sqrt(x)` – druhá odmocnina reálného čísla `x` a
  - `isqrt(x)` – největší celé číslo menší rovno odmocnině `x`,
- funkce pro převod reálných čísel na celá (viz též zabudovanou funkci `round` uvedenou výše):
  - `trunc(x)` – ořezání desetinné části,
  - `floor(x)` – největší celé číslo ≤ `x`,
  - `ceil(x)` – nejmenší celé číslo ≥ `x`,
- funkce `isclose(x, y)` která realizuje „přibližnou rovnost“ čísel s plovoucí desetinnou čárkou.

1.1.7 Shrnutí K dispozici tedy máme:

- výrazy:
  - konstanty a proměnné,
  - operátory pro aritmetiku, srovnání, logické spojky,
  - použití podprogramů (volání funkcí),
- příkazy:
  - přiřazení,
  - podmínku `if`, (`elif`, `else`),
  - cykly `for` a `while`,
  - tvrzení `assert`,
- definice vlastních podprogramů `def`,
- zabudované čisté funkce `min`, `max`, `abs`, `round`
- zabudovanou proceduru `print`,
- knihovnu `math` s konstantou `pi` a čistými funkcemi:
  - `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`,
  - `radians`, `degrees`,
  - `sqrt`, `isqrt`,
  - `trunc`, `floor`, `ceil`,
  - `isclose`.

1.d: Demonstrace (ukázky)

1.d.1 [triangle] Abychom demonstrovali zápis a použití (čistých) funkcí a tedy i návratových hodnot, zadefinujeme si jednoduchou funkci se třemi parametry: délkami stran, které můžou (ale nemusí) zadávat trojúhelník. Výsledkem je pravdivostní hodnota (`True` nebo `False`), která říká, zda zadaná trojice délek stran skutečně popisuje přípustný trojúhelník. Funkcím, které nemají vedlejší efekty (tj. čistým), a kterých výsledkem je pravdivostní hodnota, říkáme **predikáty**.

Funkce, stejně jako procedury, definujeme klíčovým slovem `def`, za kterým následuje název funkce. Názvy (a později v semestru i typové anotace) parametrů píšeme do závorek za název funkce a oddělujeme je čárkami. V tomto kontextu mluvíme o **formálních parametrech** – v těle funkce se chovají jako proměnné, do kterých jsou přiřazeny hodnoty tzv. **skutečných parametrů** – těch, které jsou funkci předány při jejím použití (viz také níže). Řádek ukončíme dvojtečkou a pokračujeme **tělem** funkce: seznamem příkazů, které se při jejím použití (zavolání) vykonají.

```
def is_triangle(a, b, c):
```

Vykonávání funkce je (korektně) ukončeno buď dojdou-li příkazy k vykonání (dojdeme „na konec“), nebo vykonáním příkazu `return`. Chceme-li, aby funkce poskytla svému volajícímu nějaký **výsledek**, musíme použít příkaz `return`, kterému tuto výslednou hodnotu předáme. Výsledek můžeme zapsat jako libovolný **výraz** (zejména tedy nemusí být uložen v proměnné).

Všimněte si, že v tomto případě je výsledkem funkce logická konjunkce (použití operátoru `and`) tří podvýrazů, kde každý popisuje jednu variantu tzv. trojúhelníkové nerovnosti. Za zmínku zde stojí i konkrétní zápis těchto variant – první konjunkt je zapsán v abecedním pořadí a každý další vznikl tzv. **cyklickou záměnou** předchozího, tzn. náhradami `a` → `b`, `b` → `c` a `c` → `a`.

```
    return (a + b > c) and (b + c > a) and (c + a > b)
```

Procedura `main` je součástí každého příkladu, a obsahuje jednoduché (základní) testy, které ověří, že jste naprogramovali zhruba to, co se očekávalo. Procházející testy **nezaručují**, že je Vaše řešení správné! U příkladů jsou testy pouze v kostrách (nachystaných zdrojových souborech `.py`): v HTML a PDF verzi sbírky je budeme zobrazovat jen v ukázkách jako je tato.

```
def main():  # demo
```

V tomto příkladu stojí za povšimnutí i samotný zápis testů (je důležité, abyste je uměli přečíst): příkaz `assert` ověří, že výraz, který mu předáváme, se vyhodnotí na hodnotu `True`, a pokud tomu tak není, program okamžitě ukončí s chybou.

Krom použití příkazu `assert` si všimněte i zápisu tzv. **volání funkce** (neboli jejího použití): volání funkce je **výraz**, který začíná **jménem** příslušné funkce, které je následováno závorkami, do kterých uvádíme (skutečné) hodnoty parametrů funkce. Závorky mohou být prázdné, ale nelze je vynechat.

```
    assert is_triangle(3, 4, 5)
    assert is_triangle(1, 1, 1)
    assert not is_triangle(1, 1, 3)
    assert not is_triangle(2, 3, 1)
```

1.d.2 [sum] Uvažme posloupnost

$$a_n = n^n$$

a posloupnost jejích částečných součtů

$$s_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i^i$$

Ujistěte se, že těmto definicím rozumíte: neznáte-li například definici operátoru $\sum$ (suma), můžete se s výhodou obrátit na Wikipedii. Pro jistotu uvádíme několik členů obou těchto posloupností:

$$a_1 = 1^1 = 1$$
$$a_2 = 2^2 = 4$$
$$a_3 = 3^3 = 27$$

$$s_1 = \sum_{i=1}^{1} i^i = 1^1 = 1 = 1$$
$$s_2 = \sum_{i=1}^{2} i^i = 1^1 + 2^2 = 1 + 4 = 5$$
$$s_3 = \sum_{i=1}^{3} i^i = 1^1 + 2^2 + 3^3 = 1 + 4 + 27 = 32$$

Naším úkolem bude nyní naprogramovat v Pythonu (čistou) funkci nth_element(n), která počítá příslušné a_n, a (opět čistou) funkci partial_sum(n), která počítá příslušné s_n. První funkce je přímočará, stačí nám znát zabudovaný operátor mocnění ** a zápis definice funkce:

```
def nth_element(n):
    return n ** n
```

Výpočet partial_sum(n) bude nicméně o něco složitější: operátor suma sčítá řadu čísel, jejichž počet je dán rozdílem mezi jeho horním a dolním indexem. Objeví-li se v některém indexu proměnná, počet sečtených členů bude typicky záviset na hodnotě této proměnné.

Jak již jistě víte z přednášky, v situaci, kdy potřebujeme opakovaně provádět příkazy (a zejména není-li počet opakování konstanta) použijeme **cyklus**. Nejjednodušší formou cyklu je příkaz „opakuj n-krát“, který v Pythonu zapisujeme for i in range(n).

Krom hodnoty n je zde důležitá ještě proměnná i: obecně se jedná o tzv. **proměnnou cyklu**. Tato proměnná má k tělu cyklu podobný vztah, jako má parametr funkce k tělu funkce: před každým provedením těla (tzv. **iterací**) se do i přiřadí nová hodnota (jaká přesně hodnota to bude záleží na konkrétní formě cyklu).

V tomto případě – cyklus tvaru for i in range(n) – se do i přiřadí **pořadové číslo iterace**, a samotnou proměnnou i pak nazýváme **indexovou proměnnou**. Ve většině programovacích jazyků (a Python není výjimkou) se **indexuje od 0**, tzn. v první iteraci je i = 0, ve druhé i = 1, atd., konečně v poslední iteraci je i = n - 1. Nyní můžeme konečně přistoupit k definici funkce partial_sum(n):

```
def partial_sum(n):
```

Jako první krok si zavedeme proměnnou, do které budeme postupně přičítat jednotlivé hodnoty a_i – takové proměnné říkáme **střadač** nebo **akumulátor** (angl. accumulator).

```
    result = 0
```

Následuje samotný cyklus, který v každé iteraci do akumulátoru result přičte příslušnou hodnotu a_i. Protože indexová proměnná i je číslována od 0, ale hodnoty a_i jsou číslovány od 1, vypočteme hodnotu a_i jako nth_element(i + 1):

```
    for i in range(n):
        result += nth_element(i + 1)
```

Po skončení cyklu je v akumulátoru požadovaná suma $s_n = \sum_{i=1}^{n} a_i$. Pro každé i v rozmezí 0 až n - 1 (včetně) bylo provedeno tělo cyklu, a v result je tedy uložen součet nth_element(0 + 1) + nth_element(1 + 1) + ... + nth_element(n - 1 + 1), neboli nth_element(1) + nth_element(2) + ... + nth_element(n).

```
    return result

def main():  # demo
    assert partial_sum(1) == 1
    assert partial_sum(2) == 5
    assert partial_sum(3) == 32
    assert partial_sum(4) == 288
    assert partial_sum(7) == 873612
    assert partial_sum(15) == 449317984130199828
```

1.d.3 [fibonacci] (Čistá) funkce fib počítá n-tý prvek tzv. Fibonacciho posloupnosti, dané předpisem: $f(1) = f(2) = 1, f(n) = f(n-1)+f(n-2)$ – každý prvek této posloupnosti je tedy součtem předchozích dvou (s výjimkou prvních dvou, které jsou pevně dané).

Zkusíte-li si posloupnost napsat na papír (1, 1, 2, 3, 5, ...), zřejmě zjistíte, že nejjednodušší způsob jak to udělat, je sečíst vždy poslední dvě už napsaná čísla a výsledek připsat na konec vznikajícího seznamu. Na dřívější čísla se už nemusíme znovu dívat: pro výpočet dalšího prvku potřebujeme vidět právě dva předchozí prvky. Můžete tedy vzít gumu, a po připsání jednoho čísla na konec smazat jedno číslo ze začátku – ani s tímto opatřením nebudete mít s výpočtem žádný problém. Na papíře budou v každém momentě 2 nebo 3 čísla, podle toho, kde se ve výpočtu nacházíte.

Tuto myšlenku využijeme pro zápis algoritmu: budeme potřebovat dvě **proměnné**, které budou reprezentovat ony dvě „naposled zapsaná“ čísla na konci posloupnosti (protože někdy máme ale na papíře čísla 3, budeme ve skutečnosti občas potřebovat ještě jednu – dočasnou – proměnnou).

Protože postup výpočtu sleduje fixní seznam kroků, který se dokola opakuje, použijeme navíc **cyklus**.

```
def fib(n):
```

Proměnná a reprezentuje předposlední a proměnná b poslední vypočtené Fibonacciho číslo. Na začátku jsme na papír napsali dvě jedničky – jedná se o ony pevně dané první dva prvky posloupnosti.

```
    a = 1
    b = 1
```

Zatím jsme „vypočítali“ první a druhé Fibonacciho číslo. Zajímá-li nás n-té číslo, musíme připsat dalších n - 2 čísel, aby platilo, že poslední číslo je to, které nás zajímá. V každé iteraci následujícího cyklu provedeme výpočet jednoho dalšího čísla (a umazání prvního čísla).

```
    for i in range(n - 2):
```

Do nové (dočasné) proměnné c si vypočteme další Fibonacciho číslo. Po tomto příkazu bude proměnná a obsahovat třetí číslo od konce aktuálně „zapsaného“ seznamu, proměnná b číslo předposlední a proměnná c číslo poslední. Jsme nyní v situaci, kdy si pamatujeme zároveň 3 čísla.

```
        c = a + b
```

„Zapomenutí“ prvního čísla realizujeme tak, že „nové“ poslední dvě čísla (nyní b a c) uložíme do proměnných a a b. Hodnotou uloženou v (dočasné) proměnné c se nebudeme dále zabývat – v další iteraci cyklu proměnnou c přepíšeme novou dočasnou hodnotou. Zamyslete se, zda je pořadí následujících dvou příkazů důležité, a proč.

```
        a = b
        b = c
```

Jak jsme zmínili na začátku, proměnná b reprezentuje poslední vypočtené Fibonacciho číslo (s výjimkou krátkého okamžiku uprostřed cyklu). Protože jsme vypočetli právě n čísel, poslední z vypočtených čísel je n-té, a tedy proměnná b obsahuje kýžený výsledek funkce fib.

```
    return b

def main():  # demo
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(3) == 2
    assert fib(5) == 5
    assert fib(9) == 34
    assert fib(11) == 89
    assert fib(20) == 6765
    assert fib(40) == 102334155
    for i in range(3, 100):
        assert fib(i) - fib(i - 1) == fib(i - 2)
```

1.d.4 [cycle] Uvažujme posloupnost definovanou jako $a_1 = 1, a_{n+1} = a_n \diamond n$, kde $\diamond$ se cyklicky vybírá z $+, \cdot, -$. Prvních 5 prvků této posloupnosti (zařazené v OEIS jako A047908) je:

$$a_1 = 1$$
$$a_2 = a_1 + 1 = 2$$
$$a_3 = a_2 \cdot 2 = 4$$
$$a_4 = a_3 - 3 = 1$$
$$a_5 = a_4 + 4 = 5$$

Naším úkolem bude napsat (čistou) funkci, která vyčíslí n-tý prvek této posloupnosti:

```
def cycle(n):
```

Protože budeme chtít použít cyklus while, musíme si indexovou proměnnou explicitně zavést:

```
    i = 1
```

K výpočtu a_i potřebujeme znát hodnotu a_{i-1}, proto si aktuální hodnotu a_i uložíme do proměnné a_i (podobně jako jsme k výpočtu Fibonacciho posloupnosti potřebovali poslední dva prvky). V další iteraci (poté, co se zvýší indexová proměnná i) budeme mít v a_i chvíli hodnotu a_{i-1}, kterou využijeme pro výpočet (nové) hodnoty a_i.

```
    a_i = 1
```

Cyklus while, jak jistě víte z přednášky, provádí své tělo tak dlouho, dokud platí podmínka cyklu. V tomto případě tedy budeme cyklus opakovat dokud platí i < n:

```
    while i < n:
```

Nyní se musíme rozhodnout, který operátor použít pro výpočet další hodnoty a_i. Protože cyklicky vybíráme ze 3 možností, můžeme se rozhodnout dle zbytku po dělení indexu i třemi: v první, čtvrté, sedmé atd. iteraci použijeme operátor +, v druhé, páté, … operátor * a konečně ve třetí, šesté, … operátor -:

```
        if i % 3 == 1:
            a_i = a_i + i
        elif i % 3 == 2:
            a_i = a_i * i
        else:  # i % 3 == 0
            a_i = a_i - i

        i += 1
```

V každé iteraci cyklu zvyšujeme indexovou proměnnou i o jedna, a před cyklem platilo i ≤ n. Po cyklu musí tedy nutně platit i == n, a protože zároveň po každé iteraci platí, že a_i obsahuje hodnotu a_i, musí také platit, že po ukončení cyklu je v proměnné a_i uložena hodnota a_n.

```
    return a_i
```

```
def main():  # demo
    assert cycle(1) == 1
    assert cycle(2) == 2
    assert cycle(3) == 4
    assert cycle(4) == 1
    assert cycle(5) == 5
    assert cycle(6) == 25
    assert cycle(7) == 19
    assert cycle(8) == 26
```

1.d.5 [converge] † Každá **omezená** posloupnost – tedy taková, která nabývá hodnoty pouze z nějakého konečného intervalu – má tzv. konvergentní podposloupnost. Co tyto termíny přesně znamenají nás nemusí trápit (více se dozvíte v matematické analýze): nám bude stačit intuice.

Podposloupnost je posloupnost, která vznikne „přeskočením“ některých prvků původní posloupnosti (zde je B podposloupnost sestávající z lichých prvků posloupnosti A):

$$A \to 1, 2, 3, 4, 5, \ldots$$
$$B \to 1, 3, 5, \ldots$$

Konvergentní posloupnost je pak taková, že se její prvky postupně blíží nějaké konkrétní hodnotě (tzv. limitě L) – přibližně platí, že čím větší index i, tím je vzdálenost $|L - a_i|$ menší.

Naším úkolem bude nějakou takovou konvergentní podposloupnost najít: začneme omezenou posloupností $a_i = \sin(i)$ a budeme budovat konvergentní podposloupnost B s prvky b_j. Pozor: hledáme libovolnou podposloupnost s potřebnou vlastností, nikoliv nějakou konkrétní – máme tak při implementaci relativně velkou volnost. Jak tedy na to?

První pozorování je, že se stačí zabývat kladnými hodnotami a_i. Dále pak stačí zabezpečit, aby platilo $b_{j+1} \leq b_j$. Při výběru hodnoty b_1 máme mnoho možností, ale je výhodné zvolit $a_1 = b_1 = \sin(1)$. Zapišme nyní funkci convergent(n), které výsledkem bude hodnota b_n:

```
def convergent(n):
```

Pro samotný výpočet budeme potřebovat dva indexy: index i náleží posloupnosti A (čísluje tedy prvky a_i) zatímco index j náleží posloupnosti B (čísluje prvky b_j).

```
    i = 1
    j = 1
```

Navíc si potřebujeme pamatovat poslední nalezenou hodnotu b_j – proměnná last bude vždy (opět s výjimkou krátkého okamžiku mezi dvěma sousedními příkazy uvnitř cyklu) obsahovat j-tou hodnotu posloupnosti B (kde j značí hodnotu proměnné j zavedené výše). Vzpomeňte si také, že $a_1 = b_1$.

```
    last = sin(i)
```

Následuje samotný cyklus, který bude hledat hodnotu b_j. Tento bude postupně procházet prvky a_i posloupnosti A. Vždy, když nalezneme nové a_i, pro které platí $a_i \leq b_j$ – kde b_j je uloženo v proměnné last – můžeme toto a_i přidat do posloupnosti B, jako b_{j+1}, a odpovídajícím způsobem upravit proměnné j a last. V programu zapisujeme a_i jako sin(i).

```
    while j < n:
        i += 1
        if sin(i) > 0 and sin(i) <= last:
            j += 1
            last = sin(i)
```

Po ukončení cyklu platí j == n (před cyklem platilo j ≤ n, cyklus ukončíme jakmile přestane platit j < n a zároveň hodnotu j v každé iteraci zvýšíme nejvýše o 1). Protože v každém kroku platí, že proměnná last obsahuje prvek b_j a nyní zároveň platí j = n, celkem dostáváme, že po ukončení cyklu je v proměnné last uložena hodnota b_n.

```
    return last
```

```
def main():  # demo
    assert convergent(1) == sin(1)
    assert convergent(2) == sin(3)
    assert convergent(3) == sin(44)
```

Krom obvyklých konkrétních případů, které testujeme výše, můžeme ověřovat i **vlastnosti** námi implementovaných funkcí. Například níže kontrolujeme monotónnost (posloupnost je nestoupající) a omezenost zespodu (nulou). Tyto dvě vlastnosti dohromady zaručují, že posloupnost je konvergentní: samozřejmě, v konečném čase lze takto ověřit pouze konečný počet případů, a **testy** nám tedy ani jednu ze zmiňovaných tří vlastností **nemohou zaručit**.

```
    for i in range(5):
        assert convergent(i + 1) <= convergent(i)
        assert convergent(i) > 0
```

1.e: Elementární příklady

1.e.1 [divisors] Napište funkci, která vrátí počet různých kladných dělitelů kladného celého čísla number (např. číslo 12 je dělitelné 1, 2, 3, 4, 6 a 12 – výsledek divisors(12) bude tedy 6.

```
def divisors(number):
    pass
```

1.e.2 [powers] Napište funkci, která spočítá sumu prvních n k-tých mocnin kladných po sobě jdoucích čísel, tzn. sumu $s_n = \sum_{i=1}^{n} a_i$, kde i-tý člen $a_i = i^k$.

```
def powers(n, k):
    pass
```

1.e.3 [**multiples**] Napište funkci sum_of_multiples s parametrem n, která spočítá sumu kladných čísel a_i, kde $a_i \leq n$ a zároveň $3|a_i$ nebo $5|a_i$ (t.j. každé a_i je dělitelné třemi nebo pěti). Například pro n = 10 je očekávaný výsledek $33 = 3 + 5 + 6 + 9 + 10$.

```
def sum_of_multiples(n):
    pass
```

1.p: Přípravy

1.p.1 [**sequence**] Napište (čistou) funkci sequence, která spočítá hodnotu členu a_n níže popsané posloupnosti, kde n je první parametr této funkce.

První člen posloupnosti, a_0, je zadán parametrem initial, každý další člen je pak určen sumou $a_j = \sum_{i=1}^{k} (-1)^i \cdot i \cdot a_{j-1}$, kde k je druhým parametrem funkce sequence. Například pro parametry k = 3 a initial = 2 jsou první 3 členy posloupnosti:

$$a_0 = 2$$

$$a_1 = \sum_{i=1}^{3} (-1)^i \cdot i \cdot a_0 = -a_0 + 2a_0 - 3a_0 = -2 + 4 - 6 = -4$$

$$a_2 = \sum_{i=1}^{3} (-1)^i \cdot i \cdot a_1 = -a_1 + 2a_1 - 3a_1 = 4 - 8 + 12 = 8$$

Očekávaný výsledek pro volání sequence(2, 3, 2) je tedy 8.

```
def sequence(n, k, initial):
    pass
```

- -

1.p.2 [**nested**] Napište funkci nested, která spočítá n-tý člen posloupnosti (počítáno od 0), která vznikne napojením postupně se prodlužujících prefixů přirozených čísel.

Nechť A_i je posloupnost čísel 1 až i:

$$A_1 \to 1$$
$$A_2 \to 1, 2$$
$$A_3 \to 1, 2, 3$$
$$A_4 \to 1, 2, 3, 4$$
$$A_5 \to 1, 2, 3, 4, 5$$

Hledaná posloupnost B vznikne napojením posloupností A_1, A_2, A_3 ... (do nekonečna) za sebe:

$$B \to 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, \ldots$$

Vaším úkolem je najít n-tý prvek posloupnosti B.

```
def nested(n):
    pass
```

Dále napište funkci nested_sum, která spočítá sumu prvních n členů této posloupnosti.

```
def nested_sum(n):
    pass
```

- -

1.p.3 [**triples**] Napište funkci largest_triple, která najde pythagorejskou trojici (a, b, c) – totiž takovou, že a, b a c jsou přirozená čísla a platí $a^2 + b^2 = c^2$ (tzn. tvoří pravoúhlý trojúhelník). Hledáme trojici, která:

1. má největší možný součet $a + b + c$,
2. hodnoty a, b jsou menší než max_side.

Výsledkem funkce bude součet $a + b + c$, tedy největší možný obvod pravoúhlého trojúhelníku, jsou-li obě jeho odvěsny kratší než max_side. Předpokládejte, že max_side bude vždy alespoň 5.

```
def largest_triple(max_side):
    pass
```

1.p.4 [**geometry**] Napište predikát (tj. čistou funkci, která vrací pravdivostní hodnotu – boolean), který je pravdivý, je-li možno vytvořit pravoúhlý trojúhelník ze stran o délkách zadaných kladnými celými čísly a, b a c.

```
def is_right(a, b, c):
    pass
```

Dále napište predikát, který je pravdivý, popisují-li parametry a, b a c rovnostranný trojúhelník.

```
def is_equilateral(a, b, c):
    pass
```

Konečně napište predikát, který je pravdivý, popisují-li parametry a, b a c rovnoramenný trojúhelník.

```
def is_isosceles(a, b, c):
    pass
```

1.p.5 [**fibsum**] Napište funkci, která spočítá sumu prvních n **sudých** členů Fibonacciho posloupnosti (tj. členů, které jsou sudé, nikoliv těch, které mají sudé indexy). Například volání fibsum(3) = 44 = 2 + 8 + 34.

```
def fibsum(n):
    pass
```

1.p.6 [**next**] Napište funkci, která pro zadané celé číslo number najde nejbližší větší číslo, které je násobkem kladného celého čísla k.

```
def next_multiple(number, k):
    pass
```

Dále napište funkci, která pro zadané kladné celé číslo number najde nejbližší větší prvočíslo.

```
def next_prime(number):
    pass
```

1.r: Řešené úlohy

1.r.1 [**even**] Uvažujme posloupnost a_i druhých mocnin kladných sudých čísel $a_i = 4i^2$. Napište funkci, která vrátí sumu prvních n členů této posloupnosti $s_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} 4i^2$.

```
def even(n):
    pass
```

1.r.2 [**prime**] Napište funkci, která ověří, zda je číslo number prvočíslo.

```
def is_prime(number):
    pass
```

1.r.3 [**coins**] Uvažme, že chceme přesně zaplatit sumu value, přičemž máme k dispozici pouze mince denominací 1, 2 a 5 korun. Spočtěte, kolik nejméně mincí potřebujeme.

```
def coins(value):
    pass
```

1.r.4 [**fibfibsum**] † Nechť A je Fibonacciho posloupnost s členy a_n a B je posloupnost taková, že má na i-té pozici a_i-tý prvek posloupnosti A, tj. prvek s indexem a_i (nikoliv prvek s indexem i). Napište funkci, která sečte prvních count prvků posloupnosti B (t.j. ty prvky posloupnosti A, kterých **indexy** jsou po sobě jdoucí Fibonacciho čísla).

Například fibfibsum(6) se vypočte takto:

$$a_1 + a_1 + a_2 + a_3 + a_5 + a_8 = 1 + 1 + 1 + 2 + 5 + 21 = 31$$

```
def fibfibsum(count):
    pass
```

1.r.5 [**abundant**] † Napište predikát is_abundant, který je pravdivý, pokud je kladné celé číslo number abundantní, t.j. je menší, než součet jeho vlastních dělitelů.

Za vlastní dělitele čísla považujeme všechny jeho kladné dělitele s výjimkou čísla samotného; např. vlastní dělitelé čísla 12 jsou 1, 2, 3, 4, 6.

```
def is_abundant(number):
    pass
```

1.r.6 [**amicable**] † Napište predikát, který určí, jsou-li dvě kladná celá

čísla spřátelená (amicable). Spřátelená čísla jsou taková, že součet všech vlastních dělitelů jednoho čísla se rovná druhému číslu, a naopak – součet všech vlastních dělitelů druhého čísla se rovná prvnímu.

Za vlastní dělitele čísla považujeme všechny jeho kladné dělitele s výjimkou čísla samotného; např. vlastní dělitelé čísla 12 jsou 1, 2, 3, 4, 6.

```
def amicable(a, b):
    pass
```

1.v: Volitelné úlohy

1.v.1 **[lvseq]** Napište čistou funkci nth_element_lv která vrátí index-tý prvek posloupnosti, která vzniká takto:

$$x_0 = 2$$
$$x_1 = p$$
$$x_n = px_{n-1} - qx_{n-2}$$

Parametry p, q mohou být libovolná celá čísla, parametr index libovolné nezáporné celé číslo (v tomto příkladu indexujeme posloupnost od nuly).

```
def nth_element_lv(p, q, index):
    pass
```

1.v.2 **[dnsum]** Napište čistou funkci sum_elements_dn, která vrátí součet prvních count prvků vzestupně seřazené posloupnosti kladných celých čísel, která jsou dělitelná číslem div a zároveň nejsou dělitelná číslem nondiv. Předpokládejte, že všechny parametry jsou kladná celá čísla a že číslo div není dělitelné číslem nondiv. (Můžete zkusit přemýšlet, co by se stalo v takovém případě.)

```
def sum_elements_dn(div, nondiv, count):
    pass
```

1.v.3 **[path]** Napište čistou funkci largest_on_path která vrátí největší číslo, na které narazíme, půjdeme-li dle níže popsaných kroků od kladného celého čísla num po číslo 1. Povolené kroky jsou následující:

- je-li num sudé, vydělíme je dvěma,
- je-li num liché a větší než 1, vynásobíme je třemi a_přičteme 1,
- je-li num rovno jedné, skončili jsme.

```
def largest_on_path(num):
    pass
```

Část 2: Číselné algoritmy

Tento týden pokračujeme v programování s čísly (první setkání se složitějšími datovými typy nás čeká příští týden). Tentokrát si naprogramujeme řadu jednoduchých algoritmů, které si vystačí s konstrukcemi, které již známe: cykly `for` a `while`, podmíněnými příkazy `if`, proměnnými, a definicemi čistých funkcí. Významnější roli budou hrát i čísla s plovoucí desetinnou čárkou – typ `float`.

To, co bude tento týden nové je, že algoritmy, které budeme programovat, budou mít složitější strukturu, budou používat více proměnných a budou se typicky více větvit. V tomto týdnu byste si tedy z cvičení měli odnést základní dovednosti algoritmizace a v tomto kontextu si procvičit použití a zápis konstrukcí, které znáte z prvních dvou přednášek.

V neposlední řadě dojde tento týden i na základy dekompozice: některé algoritmy, které budeme programovat, bude vhodné rozložit na podprogramy. Podobně jako minulý týden, budeme tento týden pracovat pouze s čistými funkcemi: kdykoliv v příkladech pro tento týden zmíníme funkci, myslíme tím implicitně funkci čistou.

Ukázky:

1. `descending` – n-tá cifra čísla
2. `comb` – kombinační čísla, `for` a `while`
3. `triangle` – řešení trojúhelníků (desetinná čísla)

Elementární příklady:

1. `palindrome` – je číslo palindrom?
2. `gcd` – největší společný dělitel (naivně)
3. `digits` – počet cifer v posloupnosti

Přípravy:

1. `digit_sum` – variace na ciferný součet
2. `joined` – posloupnost čísel
3. `fraction` – převod na řetězcový zlomek
4. `maximum` – lokální maximum na intervalu
5. `credit` – ověření korektnosti čísla platební karty
6. `workdays` – počet pracovních dnů v roce

Rozšířené úlohy:

1. `savings` – úročení a inflace
2. `fridays` – počet pátků 13. v zadaném roce
3. `delete` – umazávání cifer z čísla
4. `cards` – visa, mastercard
5. `bisect` † – aproximace kořenů
6. `parasitic` – k-parazitní čísla

Volitelné úlohy:

1. `rivendell` – čísla z Groglinky
2. `palindrome` – elfí číselné palindromy
3. `zwelf` – cvelfí ciferné míchání

2.1: Programovací jazyk

Tato kapitola používá stejné jazykové prostředky a zabudované podprogramy jako kapitola první. Přibyla pouze jediná knihovní (čistá) funkce, a to `factorial(n)` z knihovny `math`, pro přímý výpočet faktoriálu přirozeného čísla `n`.

2.2: Poziční číselné soustavy

K zápisu čísel v západní civilizaci běžně používáme desítkovou soustavu. Desítková soustava je jednou z mnoha tzv. pozičních číselných soustav, při kterých se hodnota čísla odvíjí od toho, na jaké pozici stojí jaká číslice. Hodnotu čísla získáme tak, že pozice číslujeme od nuly zprava, hodnotu každé číslice násobíme základem umocněným na pozici a výsledky sečteme.

V desítkové soustavě tedy nejpravější číslici násobíme $10^0 = 1$, druhou číslici zprava násobíme $10^1 = 10$, třetí zprava $10^2 = 100$ atd.

Můžeme ovšem za základ vzít i jiné číslo než je desítka. Třeba ve trojkové soustavě násobíme číslice zprava hodnotami 1, 3, 9, 27, … v sedmičkové soustavě násobíme číslice zprava hodnotami 1, 7, 49, 343.

To, že daný zápis je myšlen v soustavě s jiným základem než 10, typicky v matematice značíme uzávorkováním a dolním indexem. Například $(321)_7$ je zápis čísla 162, protože $3 \cdot 49 + 2 \cdot 7 + 1 = 162$.

Důležité je si uvědomit, že čísla (jako abstraktní pojem pro počet) jsou úplně nezávislá na zvolené reprezentaci. Pokud bychom se vyvinuli jinak a neměli deset prstů, ale třeba jen osm, tak by nám desítková soustava připadala bizarní a osmičková jako zcela přirozená. (A mimochodem, v historii se taky používala soustava dvanáctková nebo šedesátková – zbytek té historie vidíme např. na současném systému pro měření času.)

Hlavní myšlenkou zde je to, že $(101)_2 = 5$, tedy jde o totéž číslo, jen jinak zapsané.

V Pythonu máme standardně možnost používat tyto soustavy:

- desítkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` a zápis čísel nezačíná žádným speciálním prefixem),
- dvojkou (používáme číslice `0`, `1` a zápis čísel začíná `0b`),
- osmičkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7` a zápis čísel začíná prefixem `0o`),
- šestnáctkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f` a zápis čísel začíná prefixem `0x`).

Tedy např. číslo `0o321` je číslo $(321)_8 = 209$. Totéž číslo se taky dá v Pythonu zapsat jako `209` nebo `0xd1` nebo `0b11010001`, ale pořád je to stejné číslo, jak dokládá i skutečnost, že výraz `0b11010001 == 209` se vyhodnotí na `True`.

2.d: Demonstrace (ukázky)

2.d.1 **[descending]** V této ukázce si naprogramujeme jednoduchý algoritmus, který pracuje s desítkovým rozvojem celého čísla: konkrétně se budeme ptát, zda jsou v desítkovém zápisu daného čísla jednotlivé cifry uspořádané sestupně (uvažujeme pořadí od nejvýznamnější, tzn. nejlevější, cifry).

Protože chceme pracovat s ciframi, jeví se jako rozumné zadefinovat si pomocnou funkci, která nám vrátí konkrétní cifru. Desítkový rozvoj přirozeného čísla n, které má m desítkových cifer, lze zapsat:

$$n = \sum_{i=0}^{m} a_i \cdot 10^i$$

kde pro každé a_i platí $0 \leq a_i \leq 9$. Za povšimnutí stojí i to, že dle zde použité definice má nejméně významná cifra („jednotky“) index 0.

Chceme-li nalézt k-tou cifru, můžeme postupovat následovně: nejprve n vydělíme číslem 10^k – pohled na pravou stranu výše uvedené rovnosti nám rychle napoví, že členy, u kterých je mocnina desítky menší než k ze sumy úplně zmizí a člen a_k se stane nejnižším (rozmyslete si, jak vypadá člen, kde $i = k$):

$$n/10^k = \sum_{i=k}^{m-k} a_i \cdot 10^{i-k}$$

Zbývá učinit následovné pozorování: protože nás zajímá hodnota a_k, a protože každé jiné a_i se v rozvoji $n/10^k$ objevuje vynásobeno nějakou kladnou mocninou desítky, můžeme s výhodou použít operaci zbytku po dělení (modulo, operátor `%`): tímto se zbavíme všech ostatních členů (formálněji: zbytek po dělení členu $a_i \cdot 10^{i-k}$ desíti je 0 pro každé $i > k$):

$$n/10^k \equiv a_k \pmod{10}$$

Tímto je vysvětlena na pohled velice jednoduchá funkce get_digit:

```
def get_digit(number, k):
    return (number // 10 ** k) % 10
```

Následující funkce pracuje na stejném principu: každé dělení desíti odstraní jednu cifru (jeden člen sumy, která definuje desítkový rozvoj). Počet provedených iterací si udržujeme v čítači count.

```
def count_digits(number):
    count = 0
    while number > 0:
        count += 1
        number = number // 10
    return count
```

Funkce get_digit a count_digits nám už umožní popsat náš původní problém přirozeným způsobem: pro každou dvojici cifer ověříme, že jsou ve správném pořadí. Protože cifry jsou při procházení zleva očíslovány sestupně, musíme si dát pozor, v jakém pořadí ony dvě srovnávané cifry následují.

```
def is_descending(number):
```

Dvojic cifer je o jednu méně, než cifer samotných: dvojciferné číslo má jednu dvojici cifer, trojciferné dvě, atd., proto musíme od výsledku count_digits odečíst jedničku.

```
    for k in range(count_digits(number) - 1):
```

Označme a_i cifry čísla number: volání funkce get_digit(number, i) tedy vrací hodnotu a_i. Cifra s indexem k + 1 je **nalevo** od cifry s indexem k: mají-li být tedy cifry uspořádány sestupně zleva doprava, musí pro každou dvojici platit $a_{k+1} \geq a_k$. Protože kontrolujeme, že tato podmínka platí pro každou dvojici, jakmile nalezneme nějakou, která ji porušuje (proto v podmínce níže naleznete negaci „chtěné“ vlastnosti), víme, že celkový výsledek je False, a vykonávání funkce ukončíme příkazem return (na ostatní dvojice se už není potřeba dívat).

```
        if get_digit(number, k + 1) < get_digit(number, k):
            return False
```

V cyklu výše jsme zkontrolovali každou dvojici cifer: kdyby některá porušila kýženou vlastnost (cifry jsou uspořádané sestupně), spustil by se příkaz return a funkce by byla ukončena. Proto, dojdeme-li až sem, víme, že vlastnost platila pro každou dvojici cifer, a tedy platí i pro číslo jako celek.

```
    return True
```

Zbývá pouze ověřit, že jsme v implementaci neudělali chybu.

```
def main():  # demo
    assert is_descending(7)
    assert is_descending(321)
    assert is_descending(33222111)
    assert is_descending(9999)
    assert is_descending(7741)
    assert not is_descending(123)
    assert not is_descending(332233)
    assert not is_descending(774101)
```

2.d.2 [comb] V této ukázce se zaměříme na ekvivalenci for a while cyklů. Podíváme se přitom na **kombinační čísla**, definovaná jako:

$$(n \mid k) = n!/(k! \cdot (n-k)!)$$

kde $k \leq n$. Samozřejmě, mohli bychom počítat kombinační čísla přímo z definice, navíc v modulu math je již k dispozici funkce factorial, takže bychom se v zápisu obešli úplně bez cyklů. Nicméně jednoduché pozorování nám (resp. programu, který bude výpočet provádět) může ušetřit významné množství práce. Jak jistě víte, faktoriál je definován takto:

$$n! = \prod_{i=1}^{n} i$$

A tedy:

$$n!/k! = \prod_{i=1}^{n} i / \prod_{i=1}^{k} i = \prod_{i=k+1}^{n} i$$

Navíc, abychom měli zaručeno, že skutečně práci ušetříme, můžeme tento trik aplikovat na větší z k nebo $n-k$.

```
def comb_for(n, k):
```

Nejprve zjistíme, které z k resp. $n-k$ je menší: vzhledem k symetrii definice vůči těmto dvěma hodnotám můžeme případně k nahradit hodnotou $n-k$, aniž bychom změnili výsledek: platí $(n \mid k) = (n \mid n-k)$.

```
    if k < n - k:
        k = n - k
```

Dále chceme vynásobit všechna čísla mezi k a n (nicméně k samotné chceme přeskočit, zatímco n chceme zahrnout):

```
    numerator = 1

    for i in range(k + 1, n + 1):
        numerator *= i

    return numerator // factorial(n - k)
```

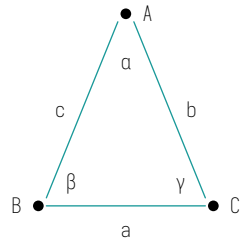
Nyní ekvivalentní definice pomocí cyklu while:

```
def comb_while(n, k):
    if k < n - k:
        k = n - k

    numerator = 1
    i = k + 1

    while i <= n:
        numerator *= i
        i += 1

    return numerator // factorial(n - k)
```

Kontrolu správnosti tentokrát provedeme trochu jinak: nebudeme kontrolovat předem vypočtené hodnoty, které bychom napsali do programu jako konstanty, jak jsme to většinou dělali doteď. Místo toho ověříme, že naše implementace dává stejný výsledek, jako výpočet přímo z definice. Díky tomu můžeme kontrolovat výrazně více případů, aniž bychom se takříkajíc upsali k smrti.

```
def main():  # demo
    for n in range(1, 50):
        for k in range(1, n):
            naive = factorial(n) // (factorial(k) * factorial(n - k))
            assert comb_for(n, k) == naive
            assert comb_while(n, k) == naive
```

2.d.3 [triangle] V této ukázce si napíšeme program, který bude počítat obvod trojúhelníka, který ale může být zadaný různými způsoby: tři strany, 2 strany a sevřený úhel, dva úhly a libovolná strana. Strany budeme značit a, b, c; úhel mezi a a b bude γ (gamma), mezi b a c bude α (alpha) a konečně mezi c a a je úhel β (beta):



Nejjednodušší je samozřejmě výpočet obvodu pro trojúhelník zadaný třemi stranami:

```
def perimeter_sss(a, b, c):
    return a + b + c
```

Následuje trojúhelník zadaný dvěma stranami a sevřeným úhlem, kdy získáme délku třetí strany použitím kosinové věty.

```
def perimeter_sas(a, gamma, b):
    c = sqrt(a ** 2 + b ** 2 - 2 * a * b * cos(radians(gamma)))
    return perimeter_sss(a, b, c)
```

Dále vyřešíme případ jedné strany a dvou jí přilehlých úhlů, kdy použijeme naopak větu sinovou.

```
def perimeter_asa(alpha, c, beta):
    gamma = radians(180 - alpha - beta)
    alpha = radians(alpha)
    beta = radians(beta)
    a = c * sin(alpha) / sin(gamma)
    b = c * sin(beta) / sin(gamma)
    return perimeter_sss(a, b, c)
```

Poslední případ, který budeme řešit, jsou dva úhly a strana přilehlá pouze druhému z nich. Tento případ lehce převedeme na předchozí.

```
def perimeter_aas(alpha, gamma, c):
    return perimeter_asa(alpha, c, 180 - alpha - gamma)
```

Tím končí samotná implementace, nyní přistoupíme k jejímu testování. Asi si uvědomujete, že v předchozím byl relativně velký prostor k překlepům a záměnám stran nebo úhlů. Proto budeme testovat důkladněji, než bylo dosud obvyklé – budeme postupovat podobně, jako v předchozí ukázce. Nejprve si implementujeme 2 pomocné funkce, které z popisu pomocí 3 délek stran vypočtou dva různé úhly:

```
def get_alpha(a, b, c):
    return acos(float(b ** 2 + c ** 2 - a ** 2) /
                (2 * b * c)) * 180.0 / pi

def get_beta(a, b, c):
    return acos(float(a ** 2 + c ** 2 - b ** 2) /
                (2 * a * c)) * 180.0 / pi
```

Pro samotnou kontrolu funkcí z rodiny perimeter_* si definujeme pomocnou proceduru, která pracuje s obecným trojúhelníkem, zadaným délkami stran.

```
def check_triangle(a, b, c):
    alpha = get_alpha(a, b, c)
    beta = get_beta(a, b, c)
    gamma = 180 - alpha - beta
```

Na tomto místě si všimněte, že na číslech s plovoucí desetinnou čárkou (typ float) **nepoužíváme** běžnou rovnost ==. Problém je, že výpočty tohoto typu mají **omezenou přesnost**: vypočteme-li stejnou hodnotu (v matematickém smyslu) dvěma různými postupy (označme výsledky jako x a y), může sice platit x == y, ale stejně dobře může také nastat x != y. To, co by mělo platit pokaždé je, že hodnoty x a y jsou si „blízko“ – tzn. že, až na chybu způsobenou nepřesností, jsou stejné. Žel, co přesně znamená „blízko“ není přesně definované a záleží od konkrétního výpočtu. Nám bude stačit výchozí definice funkce isclose z modulu math, která funguje dobře ve většině situací.

```
    assert isclose(perimeter_sss(a, b, c), a + b + c)
    assert isclose(perimeter_sas(a, gamma, b), a + b + c)
    assert isclose(perimeter_sas(b, alpha, c), a + b + c)
    assert isclose(perimeter_sas(c, beta, a), a + b + c)
    assert isclose(perimeter_asa(alpha, b, gamma), a + b + c)
    assert isclose(perimeter_asa(beta, a, gamma), a + b + c)
    assert isclose(perimeter_asa(alpha, c, beta), a + b + c)
```

Zbývá proceduru check_triangle zavolat na vhodně zvolené trojúhelníky. Strany a a b můžeme volit libovolně:

```
def main():  # demo
    for a in range(1, 6):
        for b in range(1, 6):
```

stranu c pak ale musíme zvolit tak, aby byla splněna trojúhelníková nerovnost (jinak budou funkce perimeter_* zcela oprávněně počítat nesmysly):

```
            for c in range(abs(a - b) + 1, a + b):
                check_triangle(a, b, c)
```

Na závěr si ještě demonstrujeme případ, kdy je řešení trojúhelníku skutečně nepřesné, totiž že výsledek, který obdržíme různými způsoby, může být skutečně různý.

```
    alpha = get_alpha(3, 4, 5)
    beta = get_beta(3, 4, 5)
    assert isclose(perimeter_asa(alpha, 5, beta), 12)
    assert perimeter_asa(alpha, 5, beta) != 12
    assert perimeter_sas(3, 90, 4) == 12
```

2.e: Elementární příklady

2.e.1 [palindrome] Napište predikát, který ověří, zda je číslo number palindrom, zapíšeme-li jej v desítkové soustavě. Palindrom se vyznačuje tím, že je stejný při čtení zleva i zprava.

```
def is_palindrome(number):
    pass
```

2.e.2 [gcd] Napište čistou funkci gcd, která pro zadaná kladná čísla nalezne jejich největšího společného dělitele. Použijte naivní algoritmus (tedy takový, který bude zkoušet všechny možnosti, počínaje největším vhodným kandidátem).

```
def gcd(x1, x2):
    pass
```

2.e.3 [digits] Napište funkci count_digit_in_sequence, která spočte kolikrát se cifra digit vyskytuje v číslech v rozmezí od čísla low po číslo high včetně. Například cifra 1 se na intervalu od 0 po 13 vyskytuje šestkrát, konkrétně v číslech: 1 10 11 12 13.

```
def count_digit_in_sequence(digit, low, high):
    pass
```

2.p: Přípravy

2.p.1 [digit_sum] Implementujte funkci power_digit_sum, která vrátí „speciální“ ciferný součet čísla number, který se od běžného ciferného součtu liší tím, že každou cifru před přičtením umocníme na číslo její pozice. Pozice číslujeme zleva, přičemž první má číslo 1. Vstupem funkce power_digit_sum bude libovolné nezáporné celé číslo, na výstupu se očekává celé číslo. Výpočet budeme provádět v číselné soustavě se základem 7.

Příklad: Číslo 1234 zapíšeme v sedmičkové soustavě jako $(3412)_7$ – skutečně, $3 \cdot 7^3 + 4 \cdot 7^2 + 1 \cdot 7^1 + 2 \cdot 7^0 = 1029 + 196 + 7 + 2 = 1234$. Proto power_digit_sum(1234) získáme jako $3^1 + 4^2 + 1^3 + 2^4 = 36$.

```
def power_digit_sum(number):
    pass
```

2.p.2 [joined] Napište funkci, která vytvoří číslo zřetězením count po sobě jdoucích kladných čísel počínaje zadaným číslem start. Tato čísla zřetězte vyjádřená v binární soustavě. Například volání joined(1, 3) zřetězí sekvenci $(1)_2 = 1$, $(10)_2 = 2$, $(11)_2 = 3$ a vrátí číslo $(11011)_2 = 27$. V Pythonu lze binární čísla přímo zapisovat v tomto tvaru: 0b11011 (podobně lze stejné číslo zapsat v šestnáctkové soustavě zápisem 0x1b nebo osmičkové jako 0o33).

```
def joined(start, count):
    pass
```

2.p.3 [fraction] V této úloze bude Vaším úkolem získat hodnotu indexté koeficientu řetězového zlomku pro racionální číslo s čitatelem nom a jmenovatelem denom.

Řetězový zlomek je forma reprezentace čísla jako součet celého čísla a_0 a převrácené hodnoty jiného čísla, které opět reprezentujeme součtem celého čísla a_1 a další převrácené hodnoty. Celá čísla a_n postupně tvoří řadu koeficientů řetězového zlomku.

Například řetězový zlomek $4 + (1/(2 + 1/(6 + (1/7))))$ reprezentuje číslo 415/93 a jeho koeficienty jsou 4, 2, 6 a 7.

Koeficienty řetězového zlomku pro číslo n můžete získat iterativním postupem:

1. Rozdělte číslo n na jeho celočíselnou část p a zlomkovou část q. Číslo p přímo udává první koeficient posloupnosti, tzn. a_0, zbytek koeficientů je

odvozen od q (viz další krok). Posloupnost má tedy tvar: $p; a_1, a_2, a_3, \ldots$.
2. Pro získání dalšího koeficientu opakujte 1. krok s převrácenou hodnotou zlomkové části ($1/q$).

```
def continued_fraction(nom, denom, index):
    pass
```

2.p.4 **[maximum]** Napište funkci, která najde celé číslo x, které leží mezi hodnotami low a high (včetně), a pro které vrátí funkce poly maximální hodnotu (tzn. libovolné x takové, že pro všechny x' platí $f(x) \geq f(x')$, kde f je funkce, kterou počítá podprogram poly).

```
def poly(x):
    return 10 + 30 * x - 15 * x ** 3 + x ** 5

def maximum(low, high):
    pass
```

2.p.5 **[credit]** Napište predikát, který ověří, zda je číslo korektní číslo platební karty. Číslo platební karty ověříte podle Luhnova algoritmu:

1. zdvojnásobte hodnotu každé druhé cifry zprava; je-li výsledek větší než 9, odečtěte od něj hodnotu 9,
2. sečtěte všechna takto získaná čísla a cifry na lichých pozicích zprava (kromě první cifry zprava, která slouží jako kontrolní součet),
3. číslo karty je platné právě tehdy, je-li po přičtení kontrolní cifry celkový součet dělitelný 10.

Například pro číslo 28316 je kontrolní cifra 6 a součet je: $2 + (2 \cdot 8 - 9) + 3 + 2 \cdot 1 = 2 + 7 + 3 + 2 = 14$. Po přičtení kontrolní cifry je celkový součet 20. Protože je beze zbytku dělitelný deseti, číslo karty je platné.

```
def is_valid_card(number):
    pass
```

2.p.6 **[workdays]** Napište funkci, která zjistí, kolik bude pracovních dnů v roce year. Dny v týdnu mají hodnoty 0–6 počínaje pondělím s hodnotou 0. Předpokládejte, že year je větší než 1600.

České státní svátky jsou:

datum	svátek
1.1.	Den obnovy samostatného českého státu
—	Velký pátek
—	Velikonoční pondělí
1.5.	Svátek práce
8.5.	Den vítězství
5.7.	Den slovanských věrozvěstů Cyrila a Metoděje
6.7.	Den upálení mistra Jana Husa
28.9.	Den české státnosti
28.10.	Den vzniku samostatného československého státu
17.11.	Den boje za svobodu a demokracii
24.12.	Štědrý den
25.12.	1. svátek vánoční
26.12.	2. svátek vánoční

Přestupné roky: v některých letech se na konec února přidává 29. den. Jsou to roky, které jsou dělitelné čtyřmi, s výjimkou těch, které jsou zároveň dělitelné 100 a nedělitelné 400.

Čistou funkci first_day můžete použít k tomu, abyste zjistili, na který den v týdnu padne 1. leden daného roku. Např. first_day(2001) vrátí nulu, protože rok 2001 začínal pondělím.

```
def first_day(year):
    assert 1601 <= year
    years = year - 1601
    offset = years + years // 4 - years // 100 + years // 400
    return offset % 7

def workdays(year):
    pass
```

2.r: Řešené úlohy

2.r.1 **[savings]** Vaším úkolem bude spočítat, kolik následujících let Vám vydrží úspory o hodnotě savings v bance. Na konci každého roku Vám banka úročí obnos na účtu úrokovou sazbou interest_rate (zadanou v procentech). Dále, abyste pokryli své životní náklady, na začátku každého roku vyberete z účtu obnos withdraw, který se každým rokem zvyšuje o inflaci inflation (opět zadanou v procentech). Vybíraný obnos se po započítání inflace zaokrouhluje dolů na celá čísla. Úroková sazba a inflace jsou konstantní a meziročně se nemění. Po zúročení banka celkovou částku zaokrouhluje dolů na celá čísla.

Příklad: při počátečním obnosu 100000 korun, ročních výdajích 42000 korun, úrokové sazbě 3,2% a inflaci 1,5% bude po prvním roce na účtu $(100000 - 42000) \cdot 1.032 = 59856$. Další rok se výdaje zvýší o inflaci na $42000 \cdot 1.015 = 42630$.

Budete-li mít hotovo, zkuste přemýšlet nad variantou, která by se vyhnula použití aritmetiky s plovoucí desetinnou čárkou (tedy s typem float). Budete si samozřejmě muset upravit zadání i přiložené testy – např. tak, že místo procent budou vstupem promile (desetiny procent), ovšem zadaná celočíselně (tedy např. 15 místo 1.5).

```
def savings_years(savings, interest_rate, inflation, withdraw):
    pass
```

2.r.2 **[fridays]** Napište funkci, která spočítá počet pátků 13. v daném roce year. Parametr day_of_week udává den v týdnu, na který v daném roce padne 1. leden. Dny v týdnu mají hodnoty 0–6, počínaje pondělím s hodnotou 0.

Přestupné roky: v některých letech se na konec února přidává 29. den. Jsou to roky, které jsou dělitelné čtyřmi, s výjimkou těch, které jsou zároveň dělitelné 100 a nedělitelné 400.

```
def fridays(year, day_of_week):
    pass
```

2.r.3 **[delete]** Napište funkci delete_to_maximal, která pro dané číslo number najde největší možné číslo, které lze získat smazáním jedné desítkové cifry.

```
def delete_to_maximal(number):
    pass
```

Napište funkci delete_k_to_maximal, která pro dané číslo number najde největší možné číslo, které lze získat smazáním (vynecháním) k desítkových cifer.

```
def delete_k_to_maximal(number, k):
    pass
```

2.r.4 **[cards]** Napište predikát is_visa, který je pravdivý, reprezentuje-li číslo number platné číslo platební karty VISA, tj. začíná cifrou 4, má 13, 16, nebo 19 cifer a zároveň je platným číslem platební karty (viz příklad credit).

```
def is_visa(number):
    pass
```

Dále napište predikát is_mastercard, který je pravdivý, reprezentuje-li číslo number platné číslo platební karty MasterCard, tj. začíná prefixem 50–55, nebo 22100–27209, má 16 cifer a zároveň je platným číslem platební karty.

```
def is_mastercard(number):
    pass
```

2.r.5 [bisect] † Napište funkci bisect, která aproximuje kořen spojité funkce f (předané parametrem fun) s chybou menší než epsilon na zadaném intervalu od low po high včetně. Algoritmus bisekce předpokládá, že v zadaném intervalu se nachází právě jedno řešení.

Při hledání řešení postupujte následovně:

1. spočtěte hodnotu funkce pro bod uprostřed intervalu, a je-li výsledek v rozsahu povolené chyby, vraťte tento bod,
2. jinak spočtěte hodnoty funkce v hraničních bodech intervalu a zjistěte, ve které polovině má funkce kořen,
3. opakujte výpočet s vybranou polovinou jako s novým intervalem.

Chybu e spočtete v bodě x jako $e = |f(x)|$.

Poznámka: funkci předanou parametrem můžete v Pythonu normálně volat jako libovolnou jinou funkci.

```
def bisect(fun, low, high, eps):
    pass

def fun_a(x):
    return x ** 2 - 3

def fun_b(x):
    return x ** 3 - x - 1

def fun_c(x):
    return sqrt(x) / x - x ** 3 + 5
```

2.r.6 [parasitic] Kladné celé číslo se nazývá k-parazitní v soustavě o základu b (kde b je celé číslo vetší než 1 a k je celé číslo v rozsahu 1 až $b - 1$), pokud jeho k-násobek vznikne tak, že jeho poslední (nejpravější) číslici v zápisu v soustavě o základu b přesuneme na první pozici. Například číslo 179487 je 4-parazitní v desítkové soustavě, protože platí $179487 \cdot 4 = 717948$; číslo 32 je 2-parazitní v trojkové soustavě, protože $32 = (1012)_3$, $32 \cdot 2 = 64$ a $64 = (2101)_3$.

Napište čistou funkci is_parasitic, která zjistí, zda je zadané číslo num k-parazitní v soustavě o základu base pro nějaké k – pokud ano, takové k vrátí; jinak vrátí None.

```
def is_parasitic(num, base):
    pass
```

2.v: Volitelné úlohy

2.v.1 [rivendell] Elfové z Groglinky používají k zápisu čísel jedenáctkovou soustavu, přičemž kromě nám známých číslic 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 používají ještě číslici δ reprezentující hodnotu minus jedna. (Tím se liší od ostatních elfů, kteří touto číslicí reprezentují hodnotu deset). Napište čistou funkci elf_digit_sum(num), která dostane na vstupu kladné celé číslo a vrátí součet hodnot jeho číslic v zápise elfů z Groglinky.

Například:

- elfí ciferný součet $1729 = (1332)_e$ je $1 + 3 + 3 + 2 = 9$,
- podobně $1234 = (1\delta 22)_e$ má součet $1 - 1 + 2 + 2 = 4$,
- a $999987 = (62334\delta)_e$ má součet $6 + 2 + 3 + 3 + 4 - 1 = 17$.

```
def elf_digit_sum(num):
    pass
```

2.v.2 [palindrome] Elfové používají k zápisu čísel jedenáctkovou soustavu, přičemž kromě nám známých číslic 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 používají ještě číslici δ reprezentující číslo deset.

O kladném celém čísle řekneme, že je elfím palindromem, pokud se jeho elfí (jedenáctkový) zápis čte stejně zleva i zprava poté, co **vynecháme** všechny číslice δ a následně odstraníme zbytečné levostranné nuly. (Za elfí palindromy považujeme i čísla, jejichž elfí zápis je tvořen pouze číslicemi δ.)

Napište predikát elf_palindrome(num), který vrátí True, je-li zadané číslo elfím palindromem; False jinak.

Například číslo 144 je elfím palindromem, protože jeho elfí zápis je $(121)_e$. Elfími palindromy jsou také čísla $2564 = (1\delta 21)_e$, $1211 = (\delta 01)_e$ a $33670 = (2332\delta)_e$. Elfími palindromy **nejsou** čísla $233 = (1\delta 2)_e$, $1729 = (1332)_e$.

```
def elf_palindrome(num):
    pass
```

2.v.3 [zwelf] Cvelfové používají k zápisu čísel dvanáctkovou soustavu, přičemž kromě nám známých číslic 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 používají ještě číslice δ (s hodnotou deset) a ε (s hodnotou jedenáct).

Cvelfí míchání je taková operace, kdy vezmeme kladné celé číslo v cvelfím zápisu a přeskládáme jeho číslice tak, aby všechny číslice δ stály vlevo a všechny číslice ε stály vpravo. Ostatní číslice zůstanou v původním pořadí. Výsledný zápis pak opět přečteme jako číslo v cvelfím zápisu. Napište čistou funkci zwelf_shuffle(num), která dostane na vstupu kladné celé číslo a vrátí výsledek po cvelfím míchání.

Například:

- číslo 3302 zapíše cvelf jako $(1\delta\varepsilon 2)_\beta$ a po cvelfím míchání z něj vznikne $(\delta 12\varepsilon)_\beta = 17459$,
- číslo 1587 zapíše cvelf jako $(\varepsilon 03)_\beta$ a po míchání z něj vznikne $(03\varepsilon)_\beta = 47$ (levostrannou nulu při čtení zápisu samozřejmě ignorujeme),
- číslo 1729 zapíše cvelf jako $(1001)_\beta$ a to se tedy mícháním nezmění.

```
def zwelf_shuffle(num):
    pass
```

Část 3: Seznamy a n-tice

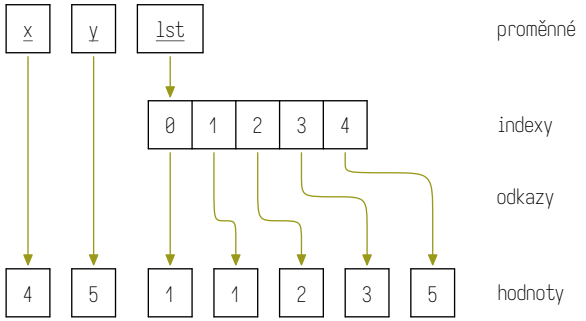
Tento týden se budeme poprvé zabývat složenými datovými typy, konkrétně těmi, které reprezentují sekvence: seznamy a uspořádanými n-ticemi. Prozatím jsme se setkali pouze s hodnotami tzv. **skalárních typů**: zejména `int`, `float`, `bool`. Použití těchto datových typů nám umožňovalo pamatovat si **fixní množství** dat: například při výpočtu n-tého prvku Fibonacciho posloupnosti jsme si potřebovali pamatovat tři čísla, které jsme měli uložené ve třech proměnných. To, co nám ale hodnoty tohoto charakteru neumožňovaly, bylo například zapamatovat si všechny dosud spočtené prvky. Zkuste se zamyslet, co by se stalo, kdybychom chtěli vyčíslit n-tý prvek posloupnosti zadané třeba takto (v OEIS nalezne pod číslem A165552):

$$a_1 = 1$$
$$a_n = \sum_{k=1}^{n-1} d(k,n) \cdot a_k$$

kde $d(k,n) = k$ když k dělí n a 0 jinak. Tady už nestačí pamatovat si poslední dva prvky – co je horší, nestačí nám **žádný** konstantní počet proměnných: potřebujeme jich tolik, kolikátý prvek chceme spočítat.

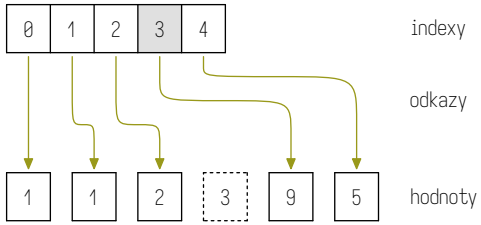
To je přesně situace, kdy lze použít **sekvenční datový typ**: hodnota sekvenčního typu se skládá z libovolného počtu jiných hodnot, očíslovaných po sobě jdoucími celými čísly. Číslu, které popisuje pozici „vnitřní" hodnoty, říkáme **index**, a podobně jak tomu bylo s indexovými proměnnými, první prvek má číslo (index) `0`.

V Pythonu existují dva základní sekvenční typy: první je **uspořádaná n-tice** (anglicky **tuple**, případně **n-tuple**), ten druhý pak **seznam** (anglicky **list**). Hodnoty obou těchto typů mají **vnitřní strukturu** – vzpomeňte si, že proměnné **váží** hodnoty ke jménům: sekvence obdobně **váže** hodnoty k **indexům** (celým číslům). Seznam a n-tice se tedy chovají podobně, jako bychom měli proměnné pojmenované `lst[0]`, `lst[1]`, `lst[2]`, atd. K těmto **pomyslným proměnným** můžeme navíc přistupovat **nepřímo**: jako index můžeme použít nejen konstantu, ale libovolné jiné číslo v programu – klidně třeba hodnotu proměnné, nebo i výraz, např. `lst[i]` nebo `lst[i + 1]`.



Obdoba **použití** proměnné (např. ve výrazu `x + 1`, který se vyhodnotí na `5`) je **indexace** seznamu, např. `lst[0]` se vyhodnotí na `1`, `lst[2] + 1` se vyhodnotí na `3`, atp. Výraz `lst[x]` se vyhodnotí na `5`.

Máme-li hodnotu typu seznam, můžeme navíc **měnit** na **kterou** hodnotu ten který index odkazuje, a tato **změna odkazu** je zcela analogická **přiřazení do proměnné**. Toto **vnitřní přiřazení** zapisujeme podobně jako to běžné, např. `lst[3] = 9`, a má obdobný efekt (na obrázku je již pouze hodnota typu seznam z proměnné `lst`):



Uspořádaná n-tice se pak od seznamu liší zejména tím, že **nemá vnitřní přiřazení**: přiřazení hodnot indexům je tedy pevně dané při vytvoření n-tice a nelze jej již dále v programu měnit. Zároveň do n-tice nelze po jejím vzniku přidávat nové indexy (těm by totiž bylo potřeba přiřadit hodnoty, a to v n-tici nelze).

Použití seznamů a n-tic si dále demonstrujeme na několika ukázkách:

1. `statistics` – iterace a indexace seznamů
2. `fibonacci` – konstrukce nového seznamu
3. `sequence` – výpočet výše uvedené posloupnosti
4. `points` – práce s n-ticemi a seznamy n-tic
5. `rotate` – mutace (vnitřní přiřazení) na seznamech

Elementární příklady:

1. `predicates` – predikáty na seznamech
2. `explosion` – filtrování seznamu podle kritéria
3. `cartesian` – výpočet kartézského součinu

Přípravy:

1. `numbers` – převod číselných soustav
2. `fraction` – vyhodnocení řetězového zlomku
3. `histogram` – četnost hodnot ve vstupním seznamu
4. `length` – délka lomené čáry
5. `merge` – sloučení dvou uspořádaných seznamů
6. `cellular` – jednoduché buněčné automaty

Rozšířené úlohy:

1. `quiz` – vyhodnocení multiple-choice testu
2. `rectangles` – překryv obdélníků v zadaném seznamu
3. `concat` – spojování vnořených seznamů
4. `rcellular` – buněčný automat in situ
5. `squares` – metoda nejmenších čtverců
6. `partition` † – přerozdělení seznamu podle velikosti

Volitelné úlohy:

1. `flats` – hledání rovin ve dvourozměrném terénu
2. `plateau` – náhorní plošiny v podobném duchu
3. `exponent` – výběr čísla podle prvočíselného rozkladu

3.1: Programovací jazyk

Tato kapitola přidává do našeho jazyka důležité prostředky pro popis a práci se složenými datovými typy (doteď jsme pracovali pouze s čísly a logickými hodnotami). Protože složená data jsou **hodnoty**, podobně jako čísla, většina změn se bude týkat **výrazů**. Mezi příkazy se objeví nová varianta cyklu `for` (pro procházení seznamu) a nové varianty **přiřazení**.

3.1.1 Literály Literály jsou typem **výrazů**. V této kapitole se objeví dva typy literálů: **seznamový literál** a **literál n-tice**.

Seznamový literál má tvar `[`*výraz*$_1$`, `*výraz*$_2$`, …, `*výraz*$_n$`]` (výrazy oddělené čárkami, uzavřené do hranatých závorek) a jeho významem je seznam, který má na indexu i hodnotu, která vznikla vyhodnocením výrazu *výraz*$_i$. Výrazů může být libovolný počet, včetně nuly (v takovém případě má výraz podobu `[]` a jeho hodnotou je prázdný seznam). Příklady:

```
[1]
[]
[1, 3, 2]
[[1, 2], [2, 3]]
[4, a + 1, f(3)]
[1, numbers[3]]
[(0, 1), (1, 1), (2, 1)]
```

Podobně, ale s kulatými závorkami, zapisujeme **literál n-tice**; ten má 3 možné podoby:

- () označuje prázdnou n-tici,
- (výraz,) označuje 1-tici (všimněte si koncové čárky),
- $(\text{výraz}_1, \text{výraz}_2, \ldots, \text{výraz}_n)$ pro $n \geq 2$.

Význam je analogický jako v případě seznamu. V některých případech lze kulaté závorky v zápisu n-tice vynechat, je-li takový zápis jednoznačný (podobně jako lze vynechat některé závorky v aritmetických výrazech). Můžeme tedy psát např. (vpravo ekvivalentní zápis s vypsanými závorkami):

```
return 1, 2          · return (1, 2)
x = 7, a + 1         · x = (7, a + 1)
a = x + 1, f(3), 7   · a = (x + 1, f(3), 7)
```

Ve všech uvedených případech jsou čárkami oddělené hodnoty interpretovány jako n-tice. Tuto zkratku ale nelze použít např. v parametru podprogramu nebo v seznamovém literálu.

3.1.2 **Rozbalení** Pro práci s n-ticemi budeme často používat tzv. **rozbalení**. Nejedná se ani o výraz ani o příkaz: je to speciální zápis, který se může objevit na **levé straně přiřazení**, v cyklu for a v intenzionálních seznamech. Zápisem se podobá na **literál n-tice**, ale místo výrazů obsahuje **jména**: $(\text{jméno}_1, \text{jméno}_2, \ldots, \text{jméno}_n)$. Podobně jako v literálu lze kulaté závorky vynechat. Můžeme tedy psát např.:

```
(x, y) = (1, 2)
x, y = (1, 2)
x, y = 1, 2
x, y = point_2d
x, y, z = point_3d
x, y = y, x
```

3.1.3 **Příkazy** Pro práci se seznamy se nám budou hodit dvě nové varianty cyklu for; první z nich (základní) zapisujeme:

```
for vazby in seznam:
    příkazy
```

kde se **výraz** seznam vyhodnotí na **seznam** a vazby je buď **jméno** nebo **rozbalení**. Tělo cyklu (příkazy) se pak provede jednou pro každý prvek seznamu seznam. V i-té iteraci odpovídají vazby i-tému prvku seznamu seznam. Je-li seznam prázdný, tělo se neprovede ani jednou.

Rozšířená verze

```
for index, vazby in enumerate(seznam):
    příkazy
```

má stejný význam jako v předchozím případě, s těmito změnami:

- index je **jméno**, které váže **index** právě iterovaného prvku v seznamu seznam (nebo ekvivalentně váže pořadové číslo právě prováděné iterace, počítáno od 0),
- v případě, kdy jsou vazby **rozbalení**, musí být uzavřeny v kulatých závorkách (jinými slovy, na tomto místě nelze závorky vynechat).

Dále přidáme dvě nové varianty příkazu **přiřazení**:

- na levé straně se může krom **jména** objevit také výše popsané **rozbalení**: $\text{jméno}_1, \ldots, \text{jméno}_n$ = výraz s významem analogickým běžnému přiřazení (pouze je dotčeno několik proměnných najednou),
- o něco komplikovanější je **přiřazení do prvku seznamu**, které zapisujeme jako seznam[index] = výraz kde seznam je **jméno** a index je **výraz** s celočíselnou hodnotou.

Přiřazení do prvku seznamu (nazýváme ho též **vnitřním** přiřazením) se ale svým významem od běžného přiřazení podstatě odlišuje: tento příkaz **upraví** stávající objekt, který je přiřazen jménu seznam.

3.1.4 **Výrazy** Krom literálů přibývá se složenými datovými typy ještě několik nových výrazů. Prvním z nich je **indexace**, která má tvar seznam[index], kde:

- seznam je **jméno** proměnné (typu seznam),
- index je aritmetický **výraz** (jeho hodnotou je celé číslo),
- výsledkem je hodnota, která je v seznamu jméno uložena na indexu i, kde i je hodnota, na kterou se vyhodnotil výraz index.

Například:

```
a[0]
numbers[i + 1]
names[compute_index(m, n)]
```

Dalším novým typem výrazu je **použití (volání) metody**, které má tvar $\text{objekt.metoda}(\text{výraz}_1, \ldots, \text{výraz}_n)$ a je obdobou **použití podprogramu** (volání funkce), který je ve speciálním vztahu s objektem vázaným ke **jménu** objekt:

- nejprve se vyhodnotí parametry $\text{výraz}_1, \ldots, \text{výraz}_n$,
- provede se volání samotné metody s názvem metoda,
- hodnotou výrazu je **návratová hodnota** volané metody.

Příklady:

```
numbers.append(a + 3)
4 + names.pop()
left.append(right.pop())
numbers.append(min(a, b))
```

Další dva nové typy výrazů nám umožní zapisovat hodnoty typu seznam:

- **seznamový literál**, který jsme již zavedli výše, nám umožňuje zapsat seznam o pevném počtu prvků, a
- **intenzionální seznam**, kterého délka může být proměnlivá, ale uložené hodnoty se řídí nějakým předpisem.

Intenzionální seznam má tyto tvary:

- [prvek for jméno in range(počet)], kde
  - počet je **výraz** s celočíselnou hodnotou,
  - výsledkem je seznam, který má počet prvků,
  - prvek i vznikne vyhodnocením výrazu prvek, přičemž jméno má pro dané vyhodnocení hodnotu i (počínaje nulou),
- $[\text{prvek for jméno in range}(n_1, n_2)]$, je analogický, ale hodnoty vázané na jméno jsou z intervalu $\langle n_1, n_2)$,
- [prvek for jméno in rozsah if podmínka], kde rozsah je range(počet) nebo range(od, do) a který má stejný význam jako předchozí, ale obsahuje pouze ty prvky, pro které se podmínka vyhodnotí jako pravdivá,
- [prvek for vazby in seznam], kde
  - seznam je **výraz** typu seznam,
  - vazby jsou **rozbalení** nebo **jméno**,
  - hodnotou je seznam, který má stejný počet prvků jako seznam a
  - na i-té pozici je hodnota, která vznikne vyhodnocením **výrazu** prvek, přičemž vazby v každém vyhodnocení odpovídají i-tému prvku seznamu seznam,
- [prvek for vazby in seznam if podmínka], který je opět ekvivalentní předchozímu, ale opět obsahuje pouze ty prvky, pro které se podmínka vyhodnotí jako pravdivá.

Výrazy podmínka se v obou případech vyhodnocují se stejnými vazbami, jako výraz prvek. Příklady:

```
[1 for i in range(5)]
[i + 1 for i in range(2 * count)]
[2 * i for i in range(7) if i != 3]
[2 * i for i in numbers]
[i ** 2 for i in numbers if i > 0]
```

Poslední nový typ výrazu je obměnou již známých relačních operátorů: výrazy x == y, x != y, x < y, x > y, x >= y, x <= y připouštíme i v případech, kdy se **oba** podvýrazy x, y vyhodnotí na seznamy, nebo se oba vyhodnotí na n-tice. Operátor < je v tomto případě dán **lexikografickým uspořádáním**:

- je-li x prefixem y nebo naopak, jako menší se vyhodnotí hodnota s menším počtem prvků,

- jinak nechť je i nejmenší index, na kterém se x a y liší a x_i a y_i jsou prvky na této pozici; výraz x < y se vyhodnotí na výsledek srovnání $x_i < y_i$.

Chování ostatních operátorů je již jednoznačně určeno rovností a operátorem <.

3.1.5 Zabudované podprogramy Pro práci se složenými datovými typy také přibudou tyto zabudované **čisté funkce**:

- len(x) – výsledkem je délka (počet prvků) seznamu x (nezáporné celé číslo),
- sum(x) – výsledkem je suma (součet) všech prvků seznamu x,
- min(x), max(x) – výsledkem je nejmenší (největší) ze všech prvků seznamu x (je-li seznam prázdný, program je ukončen s chybou).

Pro jednodušší práci s celými čísly přidáváme navíc čistou funkci

- divmod(x, y), které výsledkem je **dvojice** (x // y, x % y).

Nakonec máme nově k dispozici tyto zabudované **metody** pro hodnoty typu **seznam**:

- l.append(x) – přidá hodnotu x na konec seznamu l,
- l.pop() – odstraní ze seznamu l poslední prvek,
- l.copy() – vytvoří a vrátí kopii seznamu l.

Pozor, metody append a pop **nejsou čisté**: modifikují vstupní seznam l.

3.d: Demonstrace (ukázky)

3.d.1 [statistics] Tato ukázka demonstruje základní použití seznamů: zejména jejich **indexaci** a **iteraci**. Oba tyto koncepty si demonstrujeme na výpočtu jednoduchých statistik nad prvky předem daného seznamu: průměru, mediánu a směrodatné odchylky.

Jako první statistiku vypočteme **průměr**, který získáme jako podíl součtu všech prvků vstupního seznamu a jeho délky. Protože obě tyto operace jsou v Pythonu zabudované, je definice velice jednoduchá:

```
def average(data):
    return float(sum(data)) / len(data)
```

Protože indexace je v určitém smyslu jednodušší než iterace, budeme pokračovat výpočtem mediánu: medián je hodnota, která se objeví v **uspořádaném** souboru čísel uprostřed. Protože zatím neumíme posloupnosti řadit, budeme požadovat, by vstupem byla posloupnost již seřazená. Tuto posloupnost budeme reprezentovat **neprázdným** seznamem:

```
def median(data):
```

Zbývá tedy vypočítat index, na kterém nalezneme medián: tady nastávají dvě možnosti: buď je seznam liché, nebo sudé délky. Délku seznamu zjistíme vestavěnou (čistou) funkcí len:

```
    if len(data) % 2 == 1:
```

Případ liché délky je jednodušší, proto jej vyřešíme první. V tomto případě existuje skutečný prostřední prvek, a my pouze vrátíme jeho hodnotu. Celočíselné dělení dvěma nám dá právě ten správný index – přesvědčte se o tom!

```
        return data[len(data) // 2]
    else:
```

V opačném případě je seznam sudé délky (prázdný seznam neuvažujeme, nevyhovuje vstupní podmínce). Běžná definice mediánu v tomto případě říká, že výsledkem má být aritmetický průměr obou „prostředních“ hodnot (těch, které jsou nejblíže pomyslnému středu, který se nachází přesně mezi nimi).

```
        return float(data[len(data) // 2] +
                     data[len(data) // 2 - 1]) / 2
```

Poslední a nejsložitější statistikou je tzv. **směrodatná odchylka** s. Tuto spočítáme jako odmocninu tzv. **rozptylu** s^2, který je popsaný následovným vztahem (n je počet prvků, x_i jsou jednotlivé prvky a m je průměr):

$$s^2 = 1/(n-1)\sum_{i=1}^{n}(x_i - m)^2$$

```
def stddev(data):
```

Pro výpočet jednotlivých členů budeme potřebovat průměr, který již máme implementovaný výše. Dále si nachystáme **střadač** (akumulátor), do kterého sečteme jednotlivé kvadratické odchylky $(x_i - m)^2$:

```
    mean = average(data)
    square_error_sum = 0.0
```

Chceme-li pro každý prvek seznamu provést nějakou akci nebo výpočet, použijeme k tomu **cyklus**. Mohli bychom samozřejmě použít konstrukce, které již známe: indexovou proměnnou, cyklus tvaru for i in range(n), funkci len a indexaci seznamu data. V případě, že ale indexovou proměnnou nepotřebujeme k ničemu jinému, než indexaci jednoho seznamu, lze použít mnohem úspornější a čitelnější zápis:

```
    for x_i in data:
```

V těle takovéhoto cyklu máme v proměnné x_i uloženy přímo **hodnoty** ze seznamu data, nemusíme tedy vůbec indexovat.

```
        square_error_sum += (x_i - mean) ** 2
```

Protože rozptyl (variance) je vlastně střední (průměrná) kvadratická odchylka s drobnou korekcí, vypočteme…

```
    variance = square_error_sum / (len(data) - 1)
```

… a celkový výsledek získáme jako odmocninu rozptylu:

```
    return sqrt(variance)
```

Konečně funkčnost ověříme na několika jednoduchých příkladech.

3.d.2 [fibonacci] V této ukázce si demonstrujeme **vytváření** seznamu, který bude výstupem (čisté) funkce fib. Seznam bude obsahovat prvních n členů Fibonacciho posloupnosti, které vypočteme už známým postupem (viz též fibonacci.py z části 1).

```
def fib(n):
```

Seznam budeme budovat v cyklu. Proměnné a a b již nebudeme potřebovat, protože máme k dispozici celý seznam, bylo by tedy nehospodárné pamatovat si dva prvky ještě jednou, a to jak z pohledu využití paměti (i když v tomto případě by to nebyl velký prohřešek), ale zejména z pohledu čitelnosti programu. Většinou je nežádoucí uchovávat stejnou informaci na více místech, není potom často jasné, jsou-li obě „místa“ plně ekvivalentní, a pokud ano, tak že se chybou v programu nemůžou rozejít.

```
    out = [1, 1]
    for i in range(n - 2):
```

Pro výpočet dalšího Fibonacciho čísla využijeme zápis pro indexování seznamu od konce: je-li použitý index záporný, automaticky se k němu přičte délka indexovaného seznamu, tzn. out[-2] je totéž jako out[len(out) - 2]. Rozmyslete si, že tento výraz skutečně popisuje předposlední prvek seznamu out!

```
        value = out[-1] + out[-2]
```

Přidání na konec existujícího seznamu provedeme voláním **metody** append. Metody jsou podprogramy, které často leží někde mezi procedurou a čistou funkcí (nicméně i metody můžou být čisté, a naopak můžou mít i charakter procedury). Mají navíc ale jednu speciální vlastnost, v podobě význačného parametru, který píšeme při volání **před** jejich jméno. Následovné volání append má tedy dva parametry – out a value.

```
        out.append(value)
```

Nyní stojíme před drobným problémem: mohlo se stát, že volající si vyžádal méně než dva prvky posloupnosti, ale my jsme pro pohodlí výpočtu do seznamu vložili první dvě hodnoty. Jedna možnost řešení byla hned na začátku funkce ověřit, zda není n nula nebo jedna, a rovnou vrátit příslušný seznam ([] nebo [1]). My tento problém místo toho využijeme, abychom si ukázali, jak ze stávajícího seznamu hodnoty navíc odstranit. Rozmyslete si, že tělo cyklu se provede skutečně právě jednou, je-li n = 1 a dvakrát, je-li n = 0.

Metoda pop (bez dalších parametrů) odstraní ze seznamu poslední prvek.

```
    while len(out) > n:
        out.pop()

    return out
```

Jako obvykle, program zakončíme několika testy, abychom se ujistili, že námi implementovaná funkce pracuje (aspoň v některých případech) správně.

```
def main():  # demo
    assert fib(0) == []
    assert fib(1) == [1]
    assert fib(2) == [1, 1]
    assert fib(3) == [1, 1, 2]
    assert fib(5) == [1, 1, 2, 3, 5]
    assert fib(9) == [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

3.d.3 [sequence] V předchozích dvou ukázkách byl seznam vstupem nebo výstupem funkce. Nyní se podíváme na funkci, která má na vstupu i výstupu pouze jediné číslo, ale seznam využije pro svůj výpočet. Vrátíme se k výpočtu n-tého prvku posloupnosti, podobně jak tomu bylo v příkladech z první části. Vyčíslovat budeme posloupnost, se kterou jsme se setkali v úvodu:

$$a_1 = 1$$
$$a_n = \sum_{k=1}^{n-1} d(k,n) \cdot a_k$$

kde $d(k,n) = k$ když k dělí n a 0 jinak. Implementace bude formou čisté funkce.

```
def sequence(position):
```

Podobně jako při výpočtu fib v předchozí ukázce si vytvoříme proměnnou, ve které budeme mít uložen dosud vypočtený prefix posloupnosti. V tomto případě to ale není proto, abychom jej mohli použít jako návratovou hodnotu, ale čistě pro naše interní účely.

```
    seq = [1]
```

Do seznamu seq budeme v cyklu přidávat nové prvky posloupnosti, v každé iteraci jeden. Potřebujeme provést n - 1 iterací (jeden prvek už v seznamu máme). Nabízí se dvě možnosti: for cyklus, podobně jako v předchozím, nebo while cyklus. Protože potřebujeme indexovat od 1, není for cyklus příliš pohodlný, navíc u while cyklu je na pohled zřejmé, že má správný počet iterací, přikloníme se k této variantě:

```
    while len(seq) < position:
```

Do proměnné n si uložíme index právě počítaného prvku (číslováno od 1).

```
        n = len(seq) + 1
```

Nyní potřebujeme vypočítat hodnotu, kterou přidáme na konec seznamu. Nachystáme si střadač total, ve kterém budeme počítat definiční sumu, a indexovou proměnnou k (která bude indexovat už vypočtené hodnoty počínaje první s indexem 1).

```
        total = 0
        k = 1
```

Samotný výpočet sumy provedeme opět v cyklu.

```
        while k < n:
            if n % k == 0:
                total += k * seq[k - 1]
            k += 1
```

V proměnné total máme nyní další prvek posloupnosti, který si přidáme do seznamu seq a pokračujeme další iterací.

```
        seq.append(total)
```

Seznam seq byl čistě pomocný – umožnil nám provést výpočet. Výsledkem funkce je ale jediné číslo, totiž position-tý prvek posloupnosti. Ten nalezneme na indexu position - 1 (seznamy indexujeme od nuly, první prvek je tedy na indexu 0, atd.).

```
    return seq[position - 1]
```

Hodnoty pro testy pochází z databáze OEIS.

```
def main():  # demo
    from_oeis = [1, 1, 1, 3, 1, 6, 1, 15, 4, 8, 1, 54, 1, 10, 9,
                 135, 1, 78, 1, 100, 11, 14, 1, 822, 6, 16, 40]
    for i in range(len(from_oeis)):
        assert sequence(i + 1) == from_oeis[i]
```

3.d.4 [points] Uspořádané n-tice jsou v Pythonu velmi podobné seznamům: lze je indexovat a iterovat, ptát se na jejich délku funkcí len, ale také například vytvářet (n+m)-tice spojením n-tice s m-ticí. Jak jsme již zmiňovali v úvodu, zásadní rozdíl je, že n-tice nemá vnitřní přiřazení a nelze ji tedy po vytvoření měnit.

Ve skutečnosti ale n-tice používáme v programech výrazně jinak než seznamy, přestože mají velmi podobnou strukturu a operace. V typickém použití obsahuje seznam pouze jeden typ hodnot, ale počet hodnot je variabilní. N-tice se chovají opačně: je běžné, že obsahují hodnoty různých typů (ale všechny n-tice daného určení mají na stejném indexu stejný typ) a mají fixní počet položek.

Tento princip si demonstrujeme na příkladu, kde budeme pracovat s barevnými body v rovině. Body budeme reprezentovat jako trojice (souřadnice x, souřadnice y, barva). Každá n-tice, která reprezentuje bod, bude mít právě tuto strukturu, a bude mít vždy 3 složky (budeme tedy mluvit o trojicích). Navíc bude platit, že první dvě složky budou vždy čísla, a třetí složka bude vždy řetězec.

V principu můžeme k těmto složkám přistupovat indexací, ale existuje i mnohem lepší zápis – **rozbalení** n-tice do proměnných. Srovnejte si zápis x, y, colour = point, kde dále pracujeme se jmény x, y a colour, oproti point[0] a point[1] pro souřadnice a point[2] pro barvu. Pro srovnání si můžete v tomto příkladu přepsat všechny rozbalení trojic na indexaci a zvážit, co se Vám lépe čte.

Jako první si definujeme jednoduchou (čistou) funkci, která spočte Euklidovskou vzdálenost dvou bodů (která samozřejmě nezávisí na jejich barvě).

Poznámka: použití _ jako názvu proměnné není z pohledu Pythonu ničím zvláštním, jedná se o identifikátor jako kterýkoliv jiný. Nicméně jeho použitím indikujeme budoucím čtenářům, že hodnotu této proměnné nehodláme používat, a domluvou se tedy jedná o zástupný symbol.

```
def distance(a, b):
    a_x, a_y, _ = a
    b_x, b_y, _ = b
    return sqrt((a_x - b_x) ** 2 + (a_y - b_y) ** 2)
```

Dále si definujeme funkci, která v neprázdném seznamu najde barvu „nejlevějšího“ bodu (takového, který má nejmenší x-ovou souřadnici).

```
def leftmost_colour(points):
    x_min, _, result = points[0]

    for x, _, colour in points:
        if x < x_min:
            x_min = x
            result = colour

    return result
```

Dále si definujeme čistou funkci, která dostane jako parametry seznam bodů points a barvu colour, a jejím výsledkem bude bod, který se nachází v **těžišti** soustavy bodů dané barvy (a který bude stejné barvy). Vstupní podmínkou je, že points obsahuje aspoň jeden bod barvy colour.

```
def center_of_gravity(points, colour):
    total_x = 0.0
    total_y = 0.0
    count = 0
    for p_x, p_y, p_colour in points:
```

```
        if colour == p_colour:
            total_x += p_x
            total_y += p_y
            count += 1

    return (total_x / count, total_y / count, colour)
```

Jako poslední si definujeme (opět čistou) funkci, která spočítá průměrnou vzdálenost bodů různé barvy. Vstupní podmínkou je, že seznam points musí obsahovat aspoň dva různobarevné body.

```
def average_nonmatching_distance(points):
    total = 0.0
    pairs = 0

    for i in range(len(points)):
        for j in range(i):
            _, _, i_colour = points[i]
            _, _, j_colour = points[j]
            if i_colour != j_colour:
                total += distance(points[i], points[j])
                pairs += 1

    return total / pairs
```

Testy jsou tentokrát rozsáhlejší, protože jsme definovali větší počet funkcí. Pro úsporu horizontálního místa některé testy používají lokální aliasy pro funkce, např. dist = average_nonmatching_distance – takové přiřazení znamená, že dist je (lokální) synonymum pro average_nonmatching_distance.

```
def main():  # demo
    test_distance()
    test_leftmost_colour()
    test_center_of_gravity()
    test_average_nonmatching_distance()

def test_average_nonmatching_distance():
    r00 = (0, 0, "red")
    r10 = (1, 0, "red")
    b20 = (2, 0, "blue")
    b10 = (1, 0, "blue")
    g30 = (3, 0, "green")
    y20 = (2, 0, "yellow")
    w40 = (4, 0, "white")
    dist = average_nonmatching_distance

    assert dist([r00, b20]) == 2
    assert dist([b10, r00, b20]) == 1.5
    assert dist([r00, b20, b10, g30]) == 1.8
    assert dist([r00, b20, g30]) == 2
    assert dist([r00, b20, b10, r10]) == 1
    assert dist([r00, b10, g30, y20, w40]) == 2

def test_center_of_gravity():
    r00 = (0, 0, "red")
    r22 = (2, 2, "red")
    b20 = (2, 0, "blue")
    b02 = (0, 2, "blue")
    cog = center_of_gravity

    assert cog([r00], "red") == (0, 0, "red")
    assert cog([r00, r22], "red") == (1, 1, "red")
    assert cog([b20, b02], "blue") == (1, 1, "blue")
    assert cog([r00, b02, b20, r22], "red") == (1, 1, "red")
    assert cog([r00, b02, b20, r22], "blue") == (1, 1, "blue")

    g68 = (6, 8, "green")
    g00 = (0, 0, "green")
    g64 = (6, 4, "green")
    g86 = (8, 6, "green")
    green = [g68, g00, g64, g86]

    assert cog([g68, g00, g64], "green") == (4, 4, "green")
    assert cog(green, "green") == (5, 4.5, "green")
    green.append(r22)
    green.append(b20)
    assert cog(green, "green") == (5, 4.5, "green")

def test_leftmost_colour():
    p1 = (0, 0, "white")
    p2 = (-2, 15, "red")
    p3 = (13, -15, "yellow")
    p4 = (0, 1, "black")

    assert leftmost_colour([p1]) == "white"
    assert leftmost_colour([p3]) == "yellow"
    assert leftmost_colour([p1, p3]) == "white"
    assert leftmost_colour([p1, p3, p4, p2]) == "red"
    assert leftmost_colour([p1, p4]) == "white"
    assert leftmost_colour([p3, p4]) == "black"

def test_distance():
    p1 = (0, 0, "white")
    p2 = (1, 0, "red")

    assert distance(p1, (0, -1, "red")) == 1
    assert distance(p2, p1) == 1
    assert distance(p1, p2) == 1
    assert distance(p1, (2, 0, "black")) == 2
    assert distance(p1, (3, 4, "black")) == 5
    assert distance((-3, -4, "black"), p1) == 5
```

3.d.5 [rotate] V poslední ukázce pro tento týden se budeme zabývat vnitřním přiřazením, tzn. změnou samotné hodnoty typu seznam (změnou vnitřních vazeb indexů na hodnoty). Po delší době tedy budeme implementovat **proceduru** (podprogram, kterého hlavním smyslem je provést nějakou akci – v tomto případě pozměnit existující hodnotu). Tato procedura provede **rotaci** seznamu (na místě) o zadaný počet prvků. Např. rotací seznamu [1, 2, 3, 4]:

- o jedna doprava dostaneme seznam [4, 1, 2, 3],
- o dva doprava seznam [3, 4, 1, 2],
- o dva doleva tentýž seznam [3, 4, 1, 2] a konečně,
- o jedna doleva seznam [2, 3, 4, 1].

Směr rotace určíme dle znaménka: kladná čísla budou rotovat doprava, záporná doleva.

Možností, jak „in situ“ rotaci seznamu implementovat je několik, my si ukážeme dvě. První je konceptuálně nejjednodušší, ale nepříliš efektivní: jako základní operaci používá posuv o jedna doleva nebo doprava. Každá rotace o jedničku musí projít celý seznam, posuvy o větší počet prvků budou tedy procházet celý seznam mnohokrát – proto je tato implementace neefektivní.

```
def rotate_naive(lst, amount):
    while amount != 0:
        if amount < 0:
```

Posuv doleva implementujeme tak, že první prvek přesuneme na poslední místo a všechny ostatní o jedna doleva.

```
            backup = lst[0]
            for i in range(len(lst) - 1):
                lst[i] = lst[i + 1]
            lst[-1] = backup
            amount += 1
        else:
```

Posuv doprava je analogický, ale všechny přesuny budou opačným směrem.

```
            backup = lst[-1]
            for i in range(len(lst) - 1, 0, -1):
                lst[i] = lst[i - 1]
            lst[0] = backup
            amount -= 1
```

Jiná možnost je prvky rovnou posouvat na správné místo v seznamu (použitím vnitřního přiřazení), musíme si ale pamatovat prvky, které takto přepisujeme, a to až do doby, než je můžeme samotné přesunout na jejich cílovou pozici. Takových prvků může být najednou až tolik, jaká je velikost posuvu.

Každý prvek ale přesouváme nejvýše jednou (bez ohledu na velikost posuvu), celkový počet operací je tedy výrazně menší než v předchozí implementaci.

```
def rotate_smart(lst, amount):
```

Pro jednoduchost implementujeme pouze posuvy doprava – posuvy doleva by byly analogické. Díky tomu je tato implementace při rotacích doleva méně efektivní (malé otočení doleva je totéž jako velké otočení doprava). V proměnné backup si budeme pamatovat ty prvky, které budeme v nejbližší době ukládat na své cílové pozice (po prvních amount přesunech zde budou uloženy právě ty prvky, které aktuálně v lst dočasně chybí).

```
    amount = amount % len(lst)
    backup = []
    for i in range(0, amount):
        backup.append(lst[i])

    for i in range(len(lst)):
```

Do target spočteme cílové políčko pro další přesun, a prvek zde umístěný prohodíme s příslušným prvkem v seznamu backup. Na pozici i % amount seznamu backup se nachází prvek, který byl v původním seznamu na pozici i, a tedy je to ten prvek, který potřebujeme umístit do lst[target]. Jejich prohozením se do backup[i % amount] dostane prvek, který byl v původním seznamu na pozici target (tj. i + amount) a tedy se k němu vrátíme po dalších amount iteracích ((i + amount) % amount == i % amount).

```
        target = (i + amount) % len(lst)
        displaced = backup[i % amount]
        backup[i % amount] = lst[target]
        lst[target] = displaced
```

Protože máme dvě implementace stejné funkce, testy si parametrizujeme konkrétní implementací, aby nám stačilo napsat je jednou. Za parametr rotate se postupně doplní rotate_naive a rotate_smart.

```
def check_rotate(rotate):
    lst = [1, 2, 3, 4]
    rotate(lst, 1)
    assert lst == [4, 1, 2, 3]
    rotate(lst, -1)
    assert lst == [1, 2, 3, 4]
    rotate(lst, -2)
    assert lst == [3, 4, 1, 2]
    rotate(lst, -2)
    assert lst == [1, 2, 3, 4]
    lst.append(5)
    rotate(lst, 3)
    assert lst == [3, 4, 5, 1, 2]

def main():  # demo
    check_rotate(rotate_naive)
    check_rotate(rotate_smart)
```

3.e: Elementární příklady

3.e.1 [predicates] Napište predikát all_greater_than, který je pravdivý, právě když jsou všechna čísla v seznamu sequence větší než n.

```
def all_greater_than(sequence, n):
    pass
```

Dále napište predikát any_even, který je pravdivý, je-li v seznamu sequence aspoň jedno sudé číslo.

```
def any_even(sequence):
    pass
```

3.e.2 [explosion] Napište (čistou) funkci survivors, která ze vstupního seznamu objects spočítá nový seznam, který bude obsahovat všechny prvky z objects, které jsou dostatečně vzdálené (dále než radius) od bodu center.

Můžete si představit, že funkce implementuje herní mechaniku, kdy v bodě center nastala exploze tvaru koule, která zničila vše uvnitř poloměru radius, a funkce survivors vrátí všechny objekty, které explozi přežily.

Prvky parametru objects a parametr center jsou uspořádané trojice, které reprezentují body v prostoru.

```
def distance(a, b):
    pass

def survivors(objects, center, radius):
    pass
```

3.e.3 [cartesian] Napište funkci, která vrátí kartézský součin seznamů x a y, jako nový seznam dvojic.

```
def cartesian(x, y):
    pass
```

3.p: Přípravy

3.p.1 [numbers] V této úloze naprogramujeme trojici (čistých) funkcí, které slouží pro práci s číselnými soustavami. Reprezentaci čísla v nějaké číselné soustavě budeme ukládat jako dvojici (base, digits), kde base je hodnota typu int, která reprezentuje základ soustavy, a digits je seznam cifer v této soustavě, kde každý prvek je hodnota typu int, která spadá do rozsahu [0, base - 1]. Index seznamu digits odpovídá příslušné mocnině base. Například:

- (10, [2, 9]) je zápis v desítkové soustavě a interpretujeme jej jako 2 * 1 + 9 * 10, co odpovídá číslu 92
- (7, [2, 1]) je zápis v sedmičkové soustavě a kóduje 2 * 1 + 1 * 7 = 9

První funkce implementuje převod čísla number do ciferné reprezentace v soustavě se základem base:

```
def to_digits(number, base):
    pass
```

Další funkce provádí převod opačným směrem, z ciferné reprezentace number vytvoří hodnotu typu int:

```
def from_digits(number):
    pass
```

Konečně funkce convert_digits převede ciferný zápis z jedné soustavy do jiné soustavy. Nápověda: tato funkce je velmi jednoduchá.

```
def convert_digits(number, base):
    pass
```

3.p.2 [fraction] Stejně jako v 02/fraction.py budete v této úloze pracovat s řetězovým zlomkem. Tentokrát implementujeme převod opačným směrem, na vstupu bude seznam koeficientů řetězového zlomku, a výstupem bude zlomek klasický.

Naprogramujte tedy čistou funkci continued_fraction, která dostane jako parametr seznam koeficientů a vrátí zlomek ve tvaru (numerator, denominator).

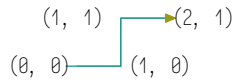
```
def continued_fraction(coefficients):
    pass
```

3.p.3 [histogram] Napište (čistou) funkci, která pro zadaný seznam nezáporných čísel data vrátí nový seznam obsahující dvojice – číslo a jeho četnost. Výstupní seznam musí být seřazený vzestupně dle první složky. Můžete předpokládat, že v data se nachází pouze celá čísla z rozsahu [0, 100] (včetně).

```
def histogram(data):
    pass
```

3.p.4 [length] Napište čistou funkci, která dostane na vstup seznam bodů v rovině (tj. seznam dvojic čísel) a vrátí délku lomené čáry, která těmito body prochází (tzn. takové, která vznikne spojením každých dvou sousedních bodů seznamu úsečkou). Souřadnice i délky reprezentujeme čísly s plovoucí desetinnou čárkou (typ float).

Například seznam [(0, 0), (1, 0), (1, 1), (2, 1)] definuje tuto lomenou čáru:



složenou ze tří segmentů (úseček) velikosti 1. Její délka je 3.

```
def length(points):
    pass
```

3.p.5 **[merge]** Naprogramujte (čistou) funkci, která ze dvou vzestupně seřazených seznamů čísel `a`, `b` vytvoří nový vzestupně seřazený seznam, který bude obsahovat všechny prvky z `a` i `b`. Nezapomeňte, že nesmíte modifikovat vstupní seznamy (jinak by funkce nebyla čistá). Pokuste se funkci naprogramovat **efektivně**.

```
def merge(a, b):
    pass
```

3.p.6 **[cellular]** Napište (čistou) funkci, která simuluje jeden krok výpočtu jednorozměrného buněčného automatu (cellular automaton). My se omezíme na **binární** (buňky nabývají hodnot 0 a 1) **jednorozměrný** automat s **konečným stavem**: stav takového automatu je seznam jedniček a nul, například:

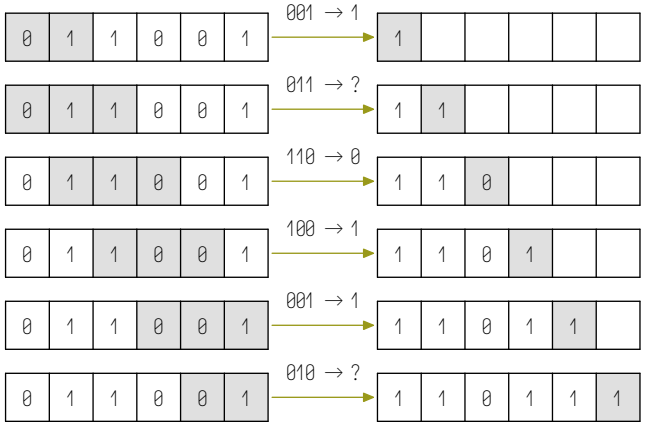
0	1	1	1	0	0	1

Protože obecný automat tohoto typu je stále relativně složitý, budeme implementovat automat s fixní sadou pravidel:

`old[i - 1]`	`old[i]`	`old[i + 1]`	`new[i]`
0	0	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Pravidla určují, jakou hodnotu bude mít buňka v následujícím stavu, v závislosti na několika okolních buňkách stavu nynějšího (konkrétní indexy viz tabulka). Neexistuje-li pro danou vstupní kombinaci pravidlo, do nového stavu přepíšeme stávající hodnotu buňky. Na krajích stavu interpretujeme chybějící políčko vždy jako nulu.

Výpočet s touto sadou pravidel tedy funguje takto:



Na vstupu dostanete stav (konfiguraci) `state`, výstupem funkce je nový seznam, který obsahuje stav vzniklý aplikací výše uvedených pravidel na `state`.

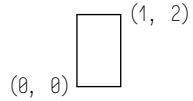
```
def cellular_step(state):
    pass
```

3.r: Řešené úlohy

3.r.1 **[quiz]** Naprogramujte funkci `mark_points`, která spočítá počet bodů, které student získal v multiple-choice testu. Vypracované řešení je reprezentováno parametrem `solution`, kterého prvky odpovídají možnostem, které student označil (tzn. je-li `solution[0]` rovno `2`, odpověď na první otázku byla `2`). Správné odpovědi jsou v parametru `answers` jako seznam dvojic, kde pozice v seznamu odpovídá číslu otázky, a dvojice je ve formě (správná odpověď, body).

```
def mark_points(answers, solution):
    pass
```

3.r.2 **[rectangles]** Napište (čistou) funkci, která jako parametr dostane seznam obdélníků a vrátí seznam obdélníků, které se překrývají s nějakým jiným. Obdélník samotný je reprezentovaný dvěma body (levým dolním a pravým horním rohem, a má nenulovou výšku i šířku). Obdélníky budeme zapisovat jako dvojice dvojic – `((0, 0), (1, 2))` například reprezentuje tento obdélník:



Mohl by se Vám hodit predikát, který je pravdivý, když se dva obdélníky překrývají:

```
def has_overlap(a, b):
    pass

def filter_overlapping(rectangles):
    pass
```

3.r.3 **[concat]** Napište funkci, která zploští seznam seznamů do jednoho nového seznamu tak, že vnořené seznamy pospojuje za sebe.

```
def concat(lists):
    pass
```

3.r.4 **[rcellular]** Podobně jako v `cellular` budeme v této úloze pracovat s 1D buněčným automatem. Místo výpočtu nové konfigurace do nového seznamu ale budeme **modifikovat** stávající seznam.

Toto samozřejmě nelze při použití stejných pravidel: v době vyhodnocování `i`-té buňky by již byla buňka s indexem `i - 1` přepsaná novou hodnotou. Proto použijeme pravidlo, které se dívá jen doprava:

`old[i]`	`old[i + 1]`	`old[i + 2]`	`new[i]`
1	0	0	0
0	1	0	1
0	1	1	1
1	0	1	0
1	1	1	0

Opět platí, že není-li nějaká konfigurace v tabulce uvedena, hodnota na indexu `i` se nemění.

Na rozdíl od předchozích příkladů, budeme v tomto implementovat **proceduru**: `cellular_in_situ` nebude hodnotu vracet, místo toho bude editovat seznam, který dostala jako parametr (viz též úvod k tomuto týdnu).

```
def cellular_in_situ(state):
    pass
```

3.r.5 **[squares]** Napište čistou funkci `least_squares`, která dostane na vstupu dva stejně dlouhé seznamy čísel. Hodnoty na odpovídajících pozicích v těchto seznamech udávají souřadnice jednoho vstupního bodu.

Výsledkem funkce nechť je trojice (α, β, r) kde $y = \alpha + \beta x$ udává přímku, která nejlépe aproximuje zadané body, a r je seznam tzv. residuí (vertikálních vzdáleností jednotlivých bodů od vypočtené přímky). Označíme-li souřadnice jednotlivých bodů (x_i, y_i) a $\bar{x}$, $\bar{y}$ aritmetické průměry příslušných seznamů, hledané koeficienty získáte použitím těchto vzorců:

$$\beta_s = \sum (x_i - \bar{x})(y_i - \bar{y})$$
$$\beta_x = \sum (x_i - \bar{x})^2$$
$$\beta = \beta_s / \beta_x$$
$$\alpha = \bar{y} - \beta \bar{x}$$

V případě, že body leží na vertikální přímce (a tedy β není definovaná), vraťte místo trojice hodnotu `None`.

```
def least_squares(x, y):
    pass
```

3.r.6 [partition] † Naprogramujte proceduru `partition`, která na vstup dostane seznam čísel `data` a platný index `idx`. Pro pohodlnost hodnotu `data[idx]` nazveme `pivot`.

Procedura přeuspořádá seznam tak, že přesune prvky menší než `pivot` před `pivot` a prvky větší než `pivot` za `pivot`.

Po transformaci bude tedy seznam **pomyslně** rozdělen na tři části:

- čísla menší než `pivot`
- pivot
- čísla větší než `pivot`

Relativní pořadí prvků v první a poslední části není definováno, takže oba následovné výsledky pro seznam `[3, 4, 1, 2, 0]` a index `0` jsou správné: `[1, 0, 2, 3, 4]` nebo `[1, 2, 0, 3, 4]`.

```
def partition(data, idx):
    pass
```

3.v: Volitelné úlohy

3.v.1 [flats] Mějme seznam nezáporných celých čísel reprezentující výšky ve 2D terénu. Plošinou v tomto seznamu nazveme maximální souvislý úsek stejné výšky délky alespoň 2.

Čistá funkce `flats` dostane na vstupu takový seznam a vrátí seznam, v němž je každá plošina reprezentovaná její výškou, a to ve stejném pořadí, v jakém jsou plošiny v původním seznamu.

```
def flats(heights):
    pass
```

Příklad: Volání `flats([2, 2, 4, 5, 4, 4, 3])` vrátí `[2, 4]`. Volání `flats([1, 2, 2, 10, 2, 9, 3, 3, 2, 2])` vrátí `[2, 3, 2]`.

3.v.2 [plateau] Pojmem „náhorní plošina“ označíme v seznamu celých čísel souvislou podposloupnost alespoň dvou stejných prvků, která ani z jedné strany nesousedí s vyšším prvkem.

Čistá funkce `rightmost_plateau` dostane na vstup neprázdný seznam celých čísel a pokud tento seznam obsahuje alespoň jednu náhorní plošinu, tak vrátí index prvního prvku nejpravější náhorní plošiny v seznamu; v opačném případě vrátí číslo `-1`.

```
def rightmost_plateau(heights):
    pass
```

Příklad: Volání `rightmost_plateau([2, 2, 4, 5, 5, 2])` vrátí `3`, protože seznam obsahuje jednu náhorní plošinu tvořenou čísly `5`, první prvek této plošiny je na indexu `3`. Volání `rightmost_plateau([3, 3, 2, 4, 4])` vrátí `3`, protože zadaný seznam obsahuje dvě náhorní plošiny, první prvek té nejpravější je na indexu 3. Volání `rightmost_plateau([2, 2, 3, 3, 4])` vrátí `-1`, protože zadaný seznam neobsahuje žádnou náhorní plošinu.

3.v.3 [exponent] Čistá funkce `largest_exponent` dostane na vstup neprázdný seznam kladných čísel `numbers` a prvočíslo `prime` a vrátí to ze zadaných čísel, které má v prvočíselném rozkladu největší mocninu zadaného prvočísla (pokud se tam zadané prvočíslo nevyskytuje, má mocninu `0`). Pokud je v seznamu více čísel se stejnou mocninou zadaného prvočísla v rozkladu, vrátí to nejmenší z nich.

```
def largest_exponent(numbers, prime):
    pass
```

Příklad: Volání `largest_exponent([24, 36, 54], 2)` vrátí `24`. Volání `largest_exponent([625, 1375, 1250], 5)` vrátí `625`.

Část 4: Testování a typy

Tento týden se zaměříme na **korektnost** (správnost) programů – zejména nás budou zajímat nástroje, které nám pomohou psát programy bez chyb. K dispozici máme dvě základní kategorie takových nástrojů:

1. **statické**, totiž takové, které analyzují program aniž by jej spouštěli – pracují podobně jako například edulint, který již znáte,
2. **dynamické**, které kontrolují, zda program pracuje správně během samotného provádění programu.

Tyto dva přístupy ke kontrole správnosti programu reprezentují určitým způsobem opačné kompromisy. Dynamické nástroje jsou velice **přesné** (umožňují kontrolovat prakticky libovolné, i velmi složité, vlastnosti), ale nemůžou nám zaručit, že program se bude za všech okolností chovat správně. Taková kontrola je často velmi časově náročná, protože abychom si ověřili správnost programu, musíme jej **testovat**: opakovaně spouštět s různými vstupy.

Statická kontrola je naopak méně přesná (umožňuje nám kontrolovat pouze jednoduché vlastnosti programu), ale je rychlá (program není potřeba spouštět) a **může** být **bezpečná** (tzn. některé statické kontroly můžou zaručit, že určitý typ chyby v programu nikdy za běhu nenastane).

V kategorii statických nástrojů jsou pro nás zajímavé zejména **typové anotace**, které lze kontrolovat programem mypy. V tomto předmětu máme již zkušenost s **dynamickou** typovou kontrolou, kdy pokus například o sečtení čísla a řetězce vede na běhovou chybu, tzn. program v momentě, kdy se takovou operaci pokusí provést, havaruje s výjimkou TypeError. Typové anotace a statická typová kontrola nám umožní většině podobných chyb předejít, aniž bychom museli program spustit (natož důkladně testovat).

Z těch dynamických jsou pro nás přístupná zejména dynamická **tvrzení**, která zapisujeme již známým klíčovým slovem assert. Dynamická tvrzení nám zejména umožňují formalizovat a automaticky při každém volání kontrolovat vstupní a výstupní podmínky funkcí (podprogramů). Můžeme je také použít k zápisu a ověření dalších podmínek, o kterých jsme přesvědčeni, že musí v daném místě programu za každých okolností platit.

V obou případech (typové anotace a dynamická tvrzení) musíme do programu přidat dodatečné informace, které netvoří přímo součást výpočetní části programu (tzn. nepopisují samotné kroky výpočtu). Mohlo by se na první pohled zdát, že přidávat tyto „přebytečné“ prvky do programu je práce navíc, která nás bude při programování leda zdržovat. Trochu hlubší analýza ale odhalí, že počáteční zápis programu tvoří jen zlomek celkového času, který programováním strávíme – ladění a údržba typicky zaberou času mnohem víc. Investice do anotací se většinou v těchto návazných fázích vývoje programu velmi rychle vrátí.

Anotace plní 3 základní funkce:

1. nutí nás hlouběji se zamyslet o chování programu – často si uvědomíme chybu už v čase, kdy uvažujeme jaké použít anotace,
2. umožňují použití automatických nástrojů pro kontrolu správnosti, čím detekují chyby, které nám v prvním bodě přeci jen proklouznou,
3. slouží jako dokumentace, jak pro programátory, kteří naše funkce chtějí použít, tak pro pozdější úpravy a opravy v samotném kódu.

Tento týden si práci s anotacemi (zejména těmi typovými) nacvičíme na příkladech. Nejprve ale jejich použití demonstrujeme v několika ukázkách:

1. shapes – typové anotace
2. barcode – vstupní a výstupní podmínky (1. část)
3. ean – vstupní a výstupní podmínky (2. část)

Elementární příklady:

1. typefun – základní typování funkcí
2. squares – složitější typování
3. fridays – typování

Přípravy:

1. database – typování
2. points – typování seznamů
3. course – kombinace typování
4. triangle – volitelné argumenty
5. doctor – zanořené seznamy
6. divisors – tabulace počtu společných dělitelů

Rozšířené úlohy:

1. squares – metoda nejmenších čtverců podruhé
2. life – hra života
3. predicates – příklady na pochopení kódu
4. poly † – derivace a integrace polynomů
5. – – (tbd)
6. precondition – určování vstupní podmínky

Volitelné úlohy:

1. fibnum – čísla ve Fibonacciho soustavě
2. – – (tbd)
3. gambling – výpočet bodů podle hodu kostkami

4.1: Programovací jazyk

Hlavní novinkou této kapitoly jsou **typové anotace**. Ty se dotknou zejména definice funkce a příkazu přiřazení. Rozšířený zápis definice funkce má následovný tvar:

def podprogram(p_1: typ_1, p_2: typ_2, …, p_n: typ_n) -> typ_r:

příkazy

Příkaz přiřazení dostane nový tvar, konkrétně:

jméno: typ = výraz

Význam všech anotací tvaru jméno: typ (tzn. jak v parametrech funkcí, tak v přiřazení) je „jméno **vždy** váže hodnotu typu typ“. Význam anotace -> typ v definici funkce má pak význam „návratová hodnota funkce je **vždy** typu typ“. Pravdivost těchto tvrzení pak (staticky) ověří program mypy, jak již bylo naznačeno v úvodě.

4.1.1 Typy Na místě typ se ve výše uvedených formách může objevit:

- jednoduchý typ:
  - bool – hodnota je True nebo False,
  - int – hodnota je celé číslo,
  - float – hodnota je číslo s plovoucí desetinnou čárkou,
  - str – hodnota je řetězec,
  - None – hodnota je None,
- složený typ, který vznikne použitím **typového konstruktoru** (tuple, list, atp.) a **typových parametrů** (píšeme v hranatých závorkách za konstruktor; v těchto závorkách typ představuje opět cokoliv z tohoto seznamu):
  - tuple[typ_1, typ_2, …, typ_n] – hodnota je n-tice a její i-tá složka je typu typ_i,
  - list[typ] – hodnota je seznam, kterého **každý** prvek je typu typ,
- tzv. volitelný typ, který vznikne zápisem typ | None, popisuje hodnotu, která může být typu typ, nebo může být None (ale nic jiného)[9],
- nebo tzv. **typový alias**, tedy jméno, které je přiřazením svázáno s konkrétním typem (jména typových aliasů začínají velkým písmenem):

  TypovýAlias = typ

[9] Zápis pomocí „svislítka“ | umožňuje i obecnější typy, v tuto chvíli se ale omezíme na tvar typ | None. Komplikovanější typy tohoto tvaru zavedeme v sedmé kapitole.

4.2: Časté typové chyby

V této části najdete popis některých častých typových chyb. Budeme ji postupně doplňovat, pokud vám nějaká typová chyba není jasná, můžete se zeptat v diskusním fóru. Nevkládejte tam však skutečný kód ze svých řešení domácích úkolů. Pokuste se problém s anotacemi izolovat do nějaké malé ukázky.

4.2.1 Cykly, seznamy a indexace Uvažme následující kód:

```
def longer_than_average_indices(data: list[str]) -> list[int]:
    total_length = 0
    for i in data:
        total_length += len(i)

    avg = total_length / len(data)
    out = []
    for i in range(len(data)):
        if len(data[i]) > avg:
            out.append(i)
    return out
```

Pro tento kód dostaneme následující výstup z mypy:

```
longer.py:11: error: Incompatible types in assignment
    (expression has type "int", variable has type "str")
longer.py:12: error: No overload variant of "__getitem__" of
    "list" matches argument type "str"
longer.py:12: note: Possible overload variants:
longer.py:12: note:     def __getitem__(self, int) -> str
longer.py:12: note:     def __getitem__(self, slice) -> List[str]
longer.py:14: error: Incompatible return value type
    (got "List[str]", expected "List[int]")
Found 3 errors in 1 file (checked 1 source file)
```

Obecně platí, že chyby je vhodné opravovat od začátku, protože další chyby mohou být způsobeny těmi předchozími a samy o sobě tak nemusí vždy dávat dobrý smysl.

1. První chyba se nachází na řádku s druhým `for` cyklem. Snažíme se tu přiřadit do proměnné typu `str` výraz typu `int`. V tomto případě se jedná o přiřazení do řídící proměnné cyklu a problém je způsoben tím, že jsme použili jméno proměnné, kterou jsme použili již v prvním cyklu, ale v tomto případě se ji snažíme použít pro iteraci přes položky jiného typu.
   - Chyba je mimo jiné důsledkem toho, že řídící proměnné cyklů (a obecně proměnné definované uvnitř cyklů) jsou v Pythonu (na rozdíl od mnohých dalších jazyků) dostupné i po skončení cyklu.
   - Chyby se zbavíme typicky tak, že použijeme jinou proměnnou.
2. Druhá chyba, ta na následujícím řádku, nám říká, že proměnná, kterou se snažíme indexovat je špatného typu.
   - Tato chyba je v tomto případě důsledkem té první, ale může samozřejmě nastat i samostatně. Mypy má již zapamatované, že `i` je typu `str` a tedy předpokládá, že se pokoušíme indexovat seznam řetězcem.
   - Poněkud neintuitivní je, že se v chybě neobjevuje indexace pomocí hranatých závorek, ale metoda `__getitem__`. To je dáno tím, že touto metodou je vnitřně indexace implementována.
   - Dva řádky „note“ říkají, že máme dvě možnosti, čím indexovat – buď pomocí `int` nebo `slice`. Typ `slice` v IB111 nepoužíváme, jako jediná možnost tedy zbývá indexování typem `int`.
3. Poslední chyba nám říká, že se snažíme vrátit hodnotu jiného typu, než jaká byla očekávána (díky anotaci funkce).
   - I tato chyba je v tomto případě následkem té první.

4.2.2 Operátor umocňování (`**`) Operátor `**` je specifický v tom, že v závislosti na svých argumentech může vracet různé typy, což komplikuje jeho použití v otypovaném kódu. Uvažme následující funkci, která počítá nezápornou mocninu čísla 2.

```
def power2(num: int) -> int:
    assert num >= 0
    return 2 ** num
```

Pro tento kód dostaneme následující výstup z `mypy --strict`:

```
pow.py:2: error: Returning Any from function declared to return "int"
Found 1 error in 1 file (checked 1 source file)
```

Problém je v tom, že výraz `2 ** num` pro celočíselné `num` vrací buď `int` (pokud je `num` ≥ `0`) nebo `float` (pokud je `num` < `0`). Řešení této situace je dvojí:

- pokud jste si jisti, že funkce `power2` vždycky dostane jen nezáporné parametry (jako v našem příkladě, kde je to vstupní podmínka funkce), pak je řešením výraz nejprve přiřadit do anotované proměnné, tedy např. nejprve provedeme `result: int = 2 ** num` a následně `return result`;
- pokud funkce `power2` může dostat i záporný parametr (tedy pokud bychom rozvolnili výše uvedenou vstupní podmínku), pak je nejlepší vynutit, aby výsledek byl vždy typu `float`, např. pomocí změny typu jednoho z operandů: `2.0 ** float`; samozřejmě je pak třeba rovněž změnit typovou anotaci návratové hodnoty funkce na `-> float`.

Typ `Any` se pak v chybové hlášce objevuje proto, že operátor `**` je v Pythonu otypovaný tak, že vrací `Any`. Lze si představit i jiná možná řešení, ale autoři mypy (resp. autoři typeshed, což je projekt, který se zabývá typovými anotacemi pro standardní knihovny a vestavěné funkce a operátory Pythonu) se (z dobrých důvodů) rozhodli, že tomuto výrazu raději žádný typ nepřidělí.

4.d: Demonstrace (ukázky)

4.d.1 [shapes] V tomto příkladu budeme počítat základní vlastnosti geometrických objektů, které budeme popisovat n-ticemi (zejména čísel). Příklad slouží k seznámení s typovou anotací parametrů a návratových hodnot podprogramů (funkcí).

Jak již víte z přednášky, anotace základních typů (`int`, `float`, `str`, atp.) se zapisuje přímo názvem typu, zatímco anotace složených typů mají trochu složitější zápis: seznamy zapisujeme jako `list[element]` (kde `element` je typová anotace platná pro každý prvek seznamu) a n-tice (zapisujeme jako `tuple[x, y, z]` – tento zápis značí trojici, kde `x`, `y` a `z` jsou postupně typové anotace pro první, druhou a třetí složku n-tice). Konečně případy, kdy potřebujeme otypovat hodnotu, která je typu `type`, ale nemusí nutně existovat (může být v některých případech `None`), použijeme anotaci `type | None`.

Jako první si definujeme čistou funkci pro výpočet obsahu kruhu (anglicky disc), která má jediný parametr typu `float` a jejíž výsledkem je opět číslo typu `float`. Tím, že tyto skutečnosti zapíšeme do programu jako anotace de-facto deklarujeme vstupní a výstupní podmínky funkce: vstupní podmínkou je, že skutečná hodnota předávaného parametru je typu `float`, zatímco výstupní je, že návratová hodnota je též typu `float`. Pro jistotu připomínáme, že za splnění **vstupní** podmínky zodpovídá **volající**, zatímco za splnění **výstupní** podmínky zodpovídá **volaná** funkce.

Program `mypy` nám pro takto anotovanou funkci zaručí dvě věci: jednak, že omylem funkci nezavoláme se špatným typem parametru (neporušíme vstupní podmínku na typy), třeba s hodnotou typu řetězec. Dále pak kontroluje, že v těle funkce neporušujeme výstupní podmínku – návratová hodnota je číslo typu `float` (nevrátíme omylem v žádném příkazu `return` ve funkci třeba řetězec, nebo `None`). K provedení této kontroly není potřeba program spouštět.

```
def disc_area(radius: float) -> float:
    return pi * radius ** 2
```

Zatímco pro popis kruhu nám stačí jediné číslo, pro popis obdélníku již potřebujeme čísla dvě, výšku a šířku. Máme dvě možnosti: můžeme potřebné hodnoty předat jako dva samostatné parametry, nebo můžeme obě hodnoty zabalit do n-tice (dvojice). Druhý přístup je lepší v případě, kdybychom potřeboval vytvořit třeba seznam obdélníků (to bude i náš případ). Proto zvolíme přístup s dvojicí čísel. Někdy má smysl složitější typy **pojmenovat**, a protože s obdélníky budeme pracovat na více místech, zavedeme si pro typ dvojice čísel jméno `Rectangle`:

```
Rectangle = tuple[float, float]
```

Nyní již můžeme přistoupit k samotné definici (opět čisté) funkce pro výpočet plochy obdélníku. Výsledkem bude opět číslo.

```
def rectangle_area(dimensions: Rectangle) -> float:
    width, height = dimensions
    return width * height
```

Elipsa reprezentuje podobný případ, kdy potřebujeme k jejímu popisu dvě čísla, tentokrát délky jejích dvou poloos. Všimněte si, že typ popisující elipsu je identický s typem pro obdélník. S tím jsou spojeny určité problémy, které si objasníme níže. Protože elipsami se nebudeme dále zabývat, nebudeme tentokrát typ pojmenovávat.

```
def ellipse_area(semiaxes: tuple[float, float]) -> float:
    major, minor = semiaxes
    return pi * major * minor
```

Abychom demonstrovali i nehomogenní n-tice (tj. takové, které mají složky různých typů), zadefinujeme si ještě pravidelný n-úhelník, který zadáme hlavním poloměrem (tzn. vzdáleností vrcholu od středu) a počtem vrcholů (který je na rozdíl od poloměru celočíselný).

```
def polygon_area(polygon: tuple[float, int]) -> float:
    radius, vertices = polygon
    half_angle = pi / vertices
    half_side = sin(half_angle) * radius
    minor_radius = cos(half_angle) * radius
    return vertices * minor_radius * half_side
```

Nyní si definujeme funkci, která budou pracovat s trochu složitějšími typy: vstupem bude seznam barevných obdélníků a jedna vybraná barva, výsledkem bude celková plocha dané barvy. Pro barvu (reprezentovanou řetězcem) si zavedeme typové synonymum: to je typicky vhodné v případech, kdy se příslušný typ objevuje jako složka n-tice. Uvažte rozdíl mezi čitelností typové anotace `tuple[tuple[int, int], str]` vs. `tuple[Rectangle, Colour]`.

```
Colour = str

def coloured_area(rectangles: list[tuple[Rectangle, Colour]],
                  selected_colour: Colour) -> float:
```

Na tomto místě musíme `mypy` trochu pomoct, protože literál `0` lze interpretovat jako celé i jako desetinné číslo, přičemž výchozí interpretace je celočíselná. V podstatě máme dvě možnosti: můžeme literál zapsat jako `0.0`, čím nejednoznačnost odstraníme, nebo přidáme typovou anotaci i proměnné (střadači) `area`. Taková anotace se zapisuje na levou stranu přiřazení a syntakticky je stejná jako anotace parametru.

```
    area: float = 0
```

Cyklus pro sečtení ploch se už od zápisu, na který jsme zvyklí, nijak neliší. Stojí nicméně za zmínku, že `mypy` za nás kontroluje krom správného volání funkce `rectangle_area` také to, že srovnáváme hodnoty stejných (obecněji kompatibilních) typů – kdybychom omylem srovnali třeba řetězec (barvu) a obdélník (třeba proto, že jsme zaměnili pořadí `rect` a `colour` při rozbalování hodnoty typu `tuple[Rectangle, Colour]`), `mypy` by nás na tuto chybu upozornilo.

```
    for rect, colour in rectangles:
        if colour == selected_colour:
            area += rectangle_area(rect)
    return area
```

Dále napíšeme funkci, která ze seznamu obdélníků vybere ten s největší plochou, existuje-li takový právě jeden. Je zde vidět, že návratový typ může být, podobně jako typy parametrů, složitější – připomínáme, že `type | None` znamená, že hodnota může být buď typu `type` nebo `None` (vzpomeňte si také, že `Rectangle` je synonymum pro `tuple[float, float]`).

```
def largest_rectangle(rectangles: list[Rectangle]) \
        -> Rectangle | None:

    if len(rectangles) == 0:
        return None

    largest = rectangles[0]
    count = 0

    for r in rectangles:
        if isclose(rectangle_area(r), rectangle_area(largest)):
            count += 1
        elif rectangle_area(r) > rectangle_area(largest):
            count = 1
            largest = r

    return largest if count == 1 else None
```

Konečně napíšeme funkci, která ze seznamu obdélníků vybere ty, které mají plochu stejnou nebo větší, než je průměrná plocha celého vstupního seznamu (který musí být neprázdný).

```
def large_rectangles(rectangles: list[Rectangle]) \
        -> list[Rectangle]:
    total = sum([rectangle_area(r) for r in rectangles])
    average = float(total) / len(rectangles)
    result = []
    for r in rectangles:
        if rectangle_area(r) >= average:
            result.append(r)
    return result
```

Nyní zbývá pouze popsané funkce otestovat:

```
def main() -> None:  # demo
    unit_rectangle = (1, 1)
    assert isclose(rectangle_area(unit_rectangle), 1)
    assert isclose(rectangle_area((2, 2)), 4)
    assert isclose(polygon_area((sqrt(2), 4)), 4)
    assert isclose(polygon_area((1, 6)), 2.5980762113533)
    assert isclose(ellipse_area((1, 1)), 3.1415926535898)
    assert isclose(ellipse_area((2, 6)), 37.699111843078)
    assert isclose(ellipse_area((12.532, 8.4444)), 332.4597362298)
```

Na začátku jsme zmiňovali, že elipsu a obdélník reprezentujeme stejným typem, a že by to mohlo vést k určitým problémům. Samozřejmě, nemůže se stát nic horšího, než co by se stalo, kdybychom anotace nepoužili vůbec, nicméně musíme si zároveň uvědomit, že typové anotace nejsou všemožné, a ani před něčím, co napohled vypadá jako typová chyba, nás nemusí ochránit. Uvažte následující (zakomentovaný) příkaz – protože `unit_rectangle` je typu `tuple[float, float]` a funkce `ellipse_area` očekává parametr téhož typu, je z pohledu `mypy` takové volání v pořádku. Přesto je zřejmé, že takovéto použití nebylo zamýšleno, a téměř s jistotou povede k chybě v programu. Tuto konkrétní situaci lze lépe řešit použitím **složených datových typů**, které si ukážeme přespříští týden.

```
    pass  # assert ellipse_area(unit_rectangle) == 1

    red, green, blue = "red", "green", "blue"
    red_1 = ((1, 1), red)
    red_2 = ((5, 6), red)
    green_1 = ((1, 1), green)
    green_2 = ((5, 6), green)
    blue_1 = ((2, 3), blue)
    assert isclose(coloured_area([red_1, green_1], red), 1)
    assert isclose(coloured_area([red_1, red_2], red), 31)
    assert isclose(coloured_area([red_1, green_2, blue_1], blue), 6)
    assert isclose(coloured_area([red_1, green_1], blue), 0)
    assert largest_rectangle([]) is None
    assert largest_rectangle([(1, 1), (4, 3), (6, 2)]) is None
    assert largest_rectangle([(5, 5), (4, 3), (1, 1)]) == (5, 5)
    assert largest_rectangle([(12, 2), (10.2, 1.5)]) == (12, 2)
    r_1, r_2, r_3 = (1, 3), (5, 5), (7, 2)
    assert large_rectangles([r_1, r_2, r_3]) == [r_2, r_3]
    assert large_rectangles([r_1, r_2]) == [r_2]
    assert large_rectangles([r_1, r_1]) == [r_1, r_1]
```

4.d.2 [barcode] Tato ukázka je první z dvojice, která demonstruje použití **tvrzení** (assertion) pro popis vstupních a výstupních podmínek. Nejprve si v rychlosti zopakujme trochu teorie.

Velmi důležitá vlastnost tvrzení je, že ve **správném** (korektním) programu **musí za všech okolností platit**. Dojde-li k porušení některého tvrzení, pro-

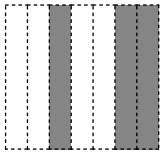
gram havaruje s chybou AssertionError a **vždy** se jedná o **chybu v programu**. Je-li tedy uživatel schopen programu předložit vstup, který způsobí, že program havaruje s chybou AssertionError, tento program je špatně.

Smyslem takovýchto tvrzení tedy není kontrola vstupu, nebo jiných okolností, které můžou selhat – naopak, slouží jako dokumentace a pomůcka k ladění: odhalit příčinu chybného chování programu je tím snazší, čím dříve si všimneme nějakou odchylku od chování očekávaného. Budeme-li důsledně kontrolovat vstupní a výstupní podmínky příkazy assert, je pravděpodobné, že chybu odchytíme brzo (program havaruje).

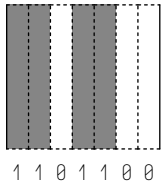
Naopak, budeme-li spoléhat na vlastní neomylnost (případně neomylnost kolegů), ale chyba se do programu přeci dostane, bude se pravděpodobně nekontrolovaně šířit – funkce, kterých vstupní podmínka nebyla splněna jednoduše vypočtou nesprávný výsledek, se kterým bude program nadále pracovat a produkovat další a další nesmyslné mezivýsledky. Výstup nebo chování programu bude nesprávné, ale bude velice obtížné a časově náročné poznat, ve kterém kroku výpočtu došlo k první chybě.

Nyní již můžeme přejít k ukázkovému programu: téma první části budou čárové kódy. V tomto modulu se budeme zabývat samotným kódováním sekvence černých a bílých pruhů, zatímco v části druhé (ean.py) se budeme zabývat již dekódovanými číselnými hodnotami.

Čárový kód sestává z řady **pruhů** (anglicky area), kde každý pruh může být černý nebo bílý. Pruhy zabírají celou výšku kódu a mají fixní šířku, přičemž na šířku se vždy dotýkají: dva sousední černé pruhy tvoří jednolitou plochu. Každá číslice je kódována do sedmi pruhů, třeba číslice 2 vypadá takto (v binárním zápisu 0010011; na obrázku je šířka jednoho pruhu přehnaná, skutečné pruhy jsou velmi úzké).



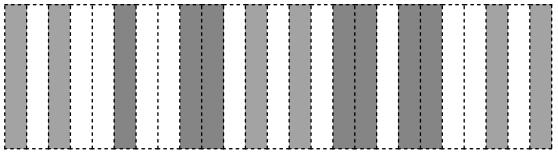
Každá číslice má 3 různá možná kódování, značená L, R a G, přičemž v kódech EAN-8, se kterými budeme pracovat, se objevují pouze kódování L a R, která jsou vzájemně inverzní: obrázek výše je v kódování L, odpovídající kódování R je následovné:



Čárové kódy standardu EAN mají 5 skupin pruhů:

- počáteční skupina, vždy 101,
- první polovina číslic (každá kódována do sedmi pruhů),
- středová dělící skupina, vždy 01010,
- druhá polovina číslic (opět po sedmi pruzích),
- koncová skupina, vždy 101.

Následuje kompletní příklad se dvěma číslicemi (2 a 2), první kódovanou v L a druhou v R. Pro odlišení jsou pruhy koncových a středové skupiny vybarveny světlejší barvou a místo 0 a 1 používají symboly _ a X:




```
def digit_count(num: int, base: int) -> int:
    result = 0
    while num > 0:
        num //= base
        result += 1
    return result

def digit_slice(num: int, base: int, low: int, size: int) -> int:
    return num // base ** low % base ** size

def bit_count(num: int) -> int:
    return digit_count(num, 2)

def bit_slice(num: int, low: int, size: int) -> int:
    return digit_slice(num, 2, low, size)
```

Jako první definujeme predikát barcode_valid, který bude kontrolovat platnost kódu (tzn. má-li požadovanou strukturu a správně zakódované číslice). Protože se jedná o relativně složitý predikát, některé kontroly oddělíme do samostatných funkcí (mnoho z nich navíc později využijeme při dekódování). Krom samotného čárového kódu má funkce parametry digit_count (počet očekávaných číslic kódu), l_coding je požadované kódování levé číselné části (L nebo R) a r_coding pravé číselné části.

```
def barcode_valid(barcode: int, digit_count: int,
                  l_coding: str, r_coding: str) -> bool:
```

Vstupní podmínky tohoto predikátu se dotýkají pouze pomocných parametrů. Zapíšeme je jako tvrzení na začátku těla:

```
    assert l_coding == 'L' or l_coding == 'R'
    assert r_coding == 'L' or r_coding == 'R'
    assert digit_count % 2 == 0
```

Pro lepší čitelnost kódu si pojmenujeme několik užitečných konstant.

```
    boundary_size = 3
    center_size = 5
    total_marker_size = 2 * boundary_size + center_size
```

Nejprve zkontrolujeme, má-li čárový kód správnou délku: musí obsahovat dvě krajové a jednu středovou skupinu a sudý počet pruhů, které kódují číslice.

```
    if bit_count(barcode) < total_marker_size:
        return False  # not enough space for all required markers
    if (bit_count(barcode) - total_marker_size) % 2 != 0:
        return False  # does not evenly split into halves

    half_width = barcode_half_width(barcode)
    center_start = boundary_size + half_width
    center_end = center_start + center_size
```

Dále prověříme, že krajové a středová značka mají správné hodnoty.

```
    if bit_slice(barcode, 0, boundary_size) != 0b101:
        return False  # bad start marker
    if bit_slice(barcode, center_end + half_width, 3) != 0b101:
        return False  # bad end marker
    if bit_slice(barcode, center_start, center_size) != 0b01010:
        return False
```

Nakonec zkontrolujeme, že má správně zakódované číslice. Zde uplatníme několik pomocných funkcí, kterých definice uvidíme později: (čistá) funkce barcode_digits z čárového kódu extrahuje dvě číslice-kódující oblasti, predikát barcode_valid_digits ověří, že vstupní číselná oblast správně kóduje číslice.

```
    if half_width % 7 != 0:
        return False
    if 2 * half_width // 7 != digit_count:
        return False

    left, right = barcode_digits(barcode)
```

```
    if not barcode_valid_digits(left, l_coding):
        return False
    if not barcode_valid_digits(right, r_coding):
        return False

    return True
```

Pomocná funkce pro výpočet délky jedné ze dvou číslicových oblastí čárového kódu, v počtu pruhů. Vstupní podmínkou je správná délka kódu (taková, aby se dal správně rozdělit na příslušné oblasti). Vstupní podmínku opět zapíšeme pomocí příkazů assert.

```
def barcode_half_width(barcode: int) -> int:
    bits = bit_count(barcode)
    assert bits >= 11
    assert (bits - 11) % 2 == 0
    return (bits - 11) // 2
```

Jak již bylo zmíněno, funkce barcode_digits extrahuje z čárového kódu dvě číselné oblasti. Potřebné vstupní podmínky již kontroluje pomocná funkce barcode_half_width kterou hned na začátku voláme, nebudeme je tedy ve funkci barcode_digits opakovat.

```
def barcode_digits(barcode: int) -> tuple[int, int]:
    half_width = barcode_half_width(barcode)
    left = bit_slice(barcode, 8 + half_width, half_width)
    right = bit_slice(barcode, 3, half_width)
    return (left, right)
```

Dále potřebujeme být schopni kódovat a dekódovat jednotlivé číslice, k čemu nám poslouží následující dvojice funkcí. V druhém parametru zadáváme, které kódování číslic požadujeme (L nebo R). V kódovací funkci je vstupní podmínkou jednak správnost druhého parametru, ale také to, že digit je skutečně jediná číslice.

```
def barcode_encode_digit(digit: int, coding: str) -> int:
    assert 0 <= digit <= 9
    assert coding == 'L' or coding == 'R'

    codes = [0b0001101, 0b0011001, 0b0010011, 0b0111101, 0b0100011,
             0b0110001, 0b0101111, 0b0111011, 0b0110111, 0b0001011]

    code = 0
    shift = 1
    bits = codes[digit]

    for _ in range(7):
        area = bits % 2
        bits //= 2
        if coding == 'L':
            code += area * shift
        if coding == 'R':
            code += (1 - area) * shift
        shift *= 2

    return code
```

Dekódování číslic provedeme „hrubou silou“ (lze to i lépe, ale pro tuto chvíli k tomu úplně nemáme ty správné jazykové prostředky). Vstupní podmínkou je, že code je nezáporné číslo. Nepovede-li se číslici v zadaném kódování přečíst, funkce vrátí None.

```
def barcode_decode_digit(code: int, coding: str) -> int | None:
    assert code >= 0
    for digit in range(10):
        if barcode_encode_digit(digit, coding) == code:
            return digit
    return None
```

Nyní jsme již připraveni definovat predikát, který bude kontrolovat správné kódování dané číselné oblasti. Jednak musí ověřit správnou délku. Jestli délka vyhovuje, opakovaným použitím funkce barcode_decode_digit se pokusíme jednotlivé číslice přečíst – selže-li tato funkce na některé skupině sedmi pruhů, je kód neplatný.

```
def barcode_valid_digits(areas: int, coding: str) -> bool:
    base = 2 ** 7
    while areas > 0:
        if barcode_decode_digit(areas % base, coding) is None:
            return False
        areas //= base
    return True
```

Konečně můžeme přistoupit k samotnému kódování a dekódování číselných oblastí čárového kódu. Dekódovat lze pouze platnou číselnou oblast, vstupní podmínkou je tedy pravdivost predikátu barcode_valid_digits. Je tedy odpovědnost volajícího špatné čárové kódy zamítnout před pokusem o jejich dekódování (lze k tomu využít třeba právě predikátu barcode_valid_digits, není-li platnost zaručena jinak).

```
def barcode_decode(areas: int, coding: str) -> int:
    assert barcode_valid_digits(areas, coding)
    result = 0
    base = 2 ** 7
    shift = 1

    while areas > 0:
        digit = barcode_decode_digit(areas % base, coding)
        areas //= base
        assert digit is not None
        result += digit * shift
        shift *= 10
    return result
```

Zbývá poslední funkce, která ze zadaných číslic vytvoří číselnou oblast čárového kódu. Vstupní podmínkou je zde pouze to, že vstupní číslo je nezáporné.

Výstupní podmínkou je, že jsme vytvořili platnou číselnou oblast. Vzpomeňte si, že výstupní podmínka je (v případě čisté funkce) vlastnost návratové hodnoty, kterou funkce sama zaručuje. Výstupní podmínku zapisujeme jako tvrzení (assert) před návratem z funkce.

```
def barcode_encode(digits: int, coding: str) -> int:
    assert digits >= 0
    result = 0
    base = 2 ** 7
    shift = 1
    while digits > 0:
        result += barcode_encode_digit(digits % 10, coding) * shift
        shift *= base
        digits //= 10
    assert barcode_valid_digits(result, coding)
    return result

def main() -> None:  # demo
    assert not barcode_valid(0b111, 0, 'L', 'L')
    assert barcode_valid(0b10101010101, 0, 'L', 'L')
    code_27_ok = 0b101_0010011_01010_0111011_101
    code_27_bad1 = 0b101_0010011_01010_0101011_101
    code_27_bad2 = 0b101_0010111_01010_0111011_101
    code_1337_ok = 0b101_0011001_0111101_01010_1000010_1000100_101
    assert barcode_valid(code_27_ok, 2, 'L', 'L')
    assert barcode_valid(code_1337_ok, 4, 'L', 'R')
    assert not barcode_valid(code_27_bad1, 2, 'L', 'L')
    assert not barcode_valid(code_27_bad2, 2, 'L', 'L')
    code_27_l, code_27_r = barcode_digits(code_27_ok)
    assert code_27_l == 0b0010011
    assert code_27_r == 0b0111011
    assert barcode_decode(code_27_l, 'L') == 2
    assert barcode_decode(code_27_r, 'L') == 7
    assert barcode_encode(13, 'L') == 0b0011001010111101
    assert barcode_encode(37, 'R') == 0b1000010100000100
```

4.d.3 [ean] European Article Number (EAN) je systém číslování výrobků, který pravděpodobně znáte z čárových kódů v supermarketech. EAN funguje podobně jako ISBN, se kterým jste minulý týden pracovali v příkladu

04/isbn.py, nicméně neomezuje se na knihy. V této ukázce budeme pokračovat v používání **tvrzení** (assert) pro popis vstupních a výstupních podmínek funkcí. Protože budeme chtít převádět číselné kódy na čárové a obráceně, využijeme funkce pro práci s čárovými kódy, které jsme definovali v předchozí ukázce.

```
from d2_barcode import \
    barcode_valid, barcode_decode, barcode_encode, barcode_digits, \
    digit_count, digit_slice

def digit_compose(left: int, right: int, base: int,
                  r_size: int) -> int:
    assert digit_count(right, base) <= r_size
    return left * (base ** r_size) + right

def decimal_count(num: int) -> int:
    return digit_count(num, 10)

def decimal_slice(num: int, low: int, digits: int) -> int:
    return digit_slice(num, 10, low, digits)

def bit_compose(left: int, right: int, r_bits: int) -> int:
    return digit_compose(left, right, 2, r_bits)

def decimal_compose(left: int, right: int, r_digits: int) -> int:
    return digit_compose(left, right, 10, r_digits)
```

Podobně jako v případě ISBN budeme EAN reprezentovat jako číslo. Jako první si zadefinujeme predikát, který bude rozhodovat, jedná-li se o platný EAN: postup je podobný jako pro ISBN, poslední cifra je kontrolní. EAN existuje v několika délkách, ale algoritmus pro jejich kontrolu je vždy stejný: proto dostane náš predikát krom samotného EAN jako parametr i očekávanou délku kódu. Tento predikát samotný nemá žádné vstupní podmínky.

```
def ean_valid(ean: int, length: int) -> bool:
    checksum = 0
    odd = True
    digits = 0

    while ean > 0:
        digits += 1
        checksum += ean % 10 * ean_digit_weight(odd)
        odd = not odd
        ean //= 10

    return digits <= length and checksum % 10 == 0
```

Pomocná funkce, která popisuje váhy jednotlivých číslic v EAN kódu (pro účely výpočtu kontrolní číslice).

```
def ean_digit_weight(odd: bool) -> int:
    return 1 if odd else 3
```

Další funkce, kterou budeme definovat, slouží k vytvoření platného EAN-13 kódu z jednotlivých komponent: prefixu GS1 (zjednodušeně odpovídá zemi výrobce), kódu výrobce (který je minimálně pěticiferný) a kódu samotného výrobku. Vstupní podmínky odpovídají omezením na jednotlivé komponenty. Celková délka kódu bez kontrolního součtu musí být 12 cifer. Funkce komponenty zkombinuje a přidá kontrolní cifru. Výstupní podmínkou je, že jsme vytvořili platný třináctimístný EAN kód (kontrolujeme ji těsně před návratem z funkce).

```
def generate_ean(gs1: int, manufacturer: int, product: int,
                 product_digits: int) -> int:
    assert 0 <= gs1 < 1000
    assert manufacturer >= 0
    assert decimal_count(product) <= product_digits
    assert decimal_count(manufacturer) + product_digits <= 10
    manufacturer_digits = 12 - product_digits - 3

    odd = False
    check = 0

    for part in [product, manufacturer, gs1]:
        while part > 0:
            check += part % 10 * ean_digit_weight(odd)
            part //= 10
            odd = not odd

    check = 10 - check % 10

    ean = decimal_compose(gs1, manufacturer, manufacturer_digits)
    ean = decimal_compose(ean, product, product_digits)
    ean = decimal_compose(ean, check, 1)

    assert ean_valid(ean, 13)
    return ean
```

Následují dvě funkce pro konverzi mezi číselným a čárovým kódem. První dostane na vstupu platnou číselnou reprezentaci EAN-8 (tuto vstupní podmínku kontroluje první příkaz assert). Výstupní podmínkou naopak je, že funkce vytvoří platný čárový kód – tuto kontrolujeme, jak je obvyklé, těsně před návratem.

```
def ean8_to_barcode(ean: int) -> int:
    assert ean_valid(ean, 8)
    left = barcode_encode(decimal_slice(ean, 4, 4), 'L')
    right = barcode_encode(decimal_slice(ean, 0, 4), 'R')

    barcode = 0
    barcode = bit_compose(barcode, 0b101, 3)
    barcode = bit_compose(barcode, left, 7 * 4)
    barcode = bit_compose(barcode, 0b01010, 5)
    barcode = bit_compose(barcode, right, 7 * 4)
    barcode = bit_compose(barcode, 0b101, 3)

    assert barcode_valid(barcode, 8, 'L', 'R')
    return barcode
```

Poslední funkce v tomto souboru slouží pro opačnou konverzi: z čárového kódu vytvoří číselnou reprezentaci. Vstupní podmínkou je, že čárový kód je platný a kóduje 8 číslic; toto díky predikátu barcode_valid lehce ověříme. Nicméně si musíme dát pozor na **výstupní** podmínku: mohlo by se zdát, že analogicky k předchozímu případu by bylo rozumné požadovat platnost číselného EAN.

Není tomu tak: byla-li splněna vstupní podmínka (čárový kód barcode je platný), funkce musí svoji výstupní podmínku **vždy splnit**. Musíme si ale uvědomit, že existují platné osmičíslicové čárové kódy, které **nekódují** platný EAN-8. Proto je výstupní podmínka platnosti EAN kódu příliš silná – nedokážeme ji zabezpečit.

Jako vhodné řešení se jeví v případě, kdy na vstupu dostaneme čárový kód reprezentující neplatný EAN, vrátit hodnotu None: výstupní podmínku tak zeslabíme jen minimálně. Bude vždy platit, že výstupem je buď platný EAN-8 (a to vždy, když je to možné), nebo hodnota None (pouze v případech, kdy vstup reprezentoval neplatný EAN-8). Ze zápisu návratové hodnoty je zřejmé, že tato výstupní podmínka je splněna, nemá tedy smysl ji dodatečně kontrolovat příkazem assert.

```
def barcode_to_ean8(barcode: int) -> int | None:
    assert barcode_valid(barcode, 8, 'L', 'R')
    left, right = barcode_digits(barcode)
    ean = decimal_compose(barcode_decode(left, 'L'),
                          barcode_decode(right, 'R'), 4)
    if not ean_valid(ean, 8):
        return None
    return ean

def main() -> None:  # demo
    week_04

    assert ean_valid(12345670, 8)
    assert ean_valid(1122334455666, 13)
    assert not ean_valid(12345674, 8)
    assert not ean_valid(1122334455664, 13)
    assert generate_ean(123, 123212, 123, 3) == 1231232121235
    assert generate_ean(444, 12345, 1111, 4) == 4441234511119
    assert ean8_to_barcode(12345670) == 0x5324dea354ea11395
    assert ean8_to_barcode(11112228) == 0x53264c9956cd9b245
    assert barcode_to_ean8(0x5324dea354ea11395) == 12345670
    assert barcode_to_ean8(0x53264c9956cd9b245) == 11112228
    assert barcode_to_ean8(0x53264c9956cd9b395) is None
```

4.e: Elementární příklady

4.e.1 [typefun] Otypujte následující funkce tak, aby prošla typová kontrola s přiloženými testy.

Funkce degrees konvertuje radiány na stupně.

```
def degrees(radians):
    return (radians * 180) / pi
```

Funkce to_list rozdělí číslo na číslice o daném základu.

```
def to_list(num, base):
    digits = []
    result = []

    while num > 0:
        digits.append(num % base)
        num //= base

    for i in range(len(digits)):
        result.append(digits[-i - 1])

    return result
```

Funkce diagonal vytvoří seznam obsahující prvky na diagonále matice matrix.

```
def diagonal(matrix):
    diag = []
    for i in range(len(matrix)):
        diag.append(matrix[i][i])
    return diag
```

Funkci with_id je v parametru elements předán seznam dvojic (celočíselný klíč, řetězec). Funkce najde prvek s klíčem id_ a vrátí odpovídající řetězec.

```
def with_id(elements, id_):
    for element_id, val in elements:
        if id_ == element_id:
            return val
    return None
```

Funkce update_students v seznamu studentů, zadaných trojicemi (učo, jméno a volitelně rok ukončení studia) všem studentům, kteří ještě nemají studium ukončené, nastaví rok ukončení studia na zadaný.

```
def update_students(students, end):

    result = []

    for uco, name, graduated in students:
        if graduated is None:
            graduated = end
        result.append((uco, name, graduated))

    return result
```

Predikát is_increasing je pravdivý, pokud je seznam celých čísel seq rostoucí.

```
def is_increasing(seq):
    for i in range(1, len(seq)):
        if seq[i - 1] >= seq[i]:
            return False
    return True
```

4.e.2 [squares] Otypujte následující funkce tak, aby prošla typová kontrola s přiloženými testy.

```
def slope(x, y, average_x, average_y):
    dividend = 0
    divisor = 0

    for i in range(len(x)):
        dividend += ((x[i] - average_x) * (y[i] - average_y))
        divisor += (x[i] - average_x) ** 2

    if divisor == 0:
        return None

    return dividend / divisor


def deviations(x, y, alpha, beta):
    res = []
    for i in range(len(x)):
        res.append(abs(y[i] - beta * x[i] - alpha))
    return res


def least_squares(x, y):
    average_x = float(sum(x)) / len(x)
    average_y = float(sum(y)) / len(y)

    beta = slope(x, y, average_x, average_y)
    if beta is None:
        return None

    alpha = average_y - beta * average_x

    return (alpha, beta, deviations(x, y, alpha, beta))


def main() -> None:
    assert check([1, 2], [3, 4], (2, 1, [0, 0]))
    assert check([1, 2, 3], [3, 4, 5], (2, 1, [0, 0, 0]))
    assert least_squares([1, 1, 1], [3, 4, 5]) is None
    assert check([1, 2, 3], [2, 2, 2], (2, 0, [0, 0, 0]))
    assert check([1, 2, 3], [1, 4, 1], (2, 0, [1, 2, 1]))
    assert check([1, 2, 3], [1, 2, 4],
                 (-2.0 / 3.0, 3.0 / 2.0,
                  [1.0 / 6.0, 1.0 / 3.0, 1.0 / 6.0]))


def check(x: list[float], y: list[float],
          expect: tuple[float, float, list[float]]) -> bool:
    result = least_squares(x, y)
    if result is None:
        return False
    (alpha1, beta1, r1) = result
    (alpha2, beta2, r2) = expect
    if not isclose(alpha1, alpha2) or not isclose(beta1, beta2):
        return False
    for i in range(len(r1)):
        if not isclose(r1[i], r2[i]):
            return False
    return True
```

4.e.3 [fridays] Otypujte následující implementaci příkladu 02/fridays.py.

```
def is_leap(year):
    if year % 400 == 0:
        return True
    if year % 4 == 0 and year % 100 != 0:
        return True
    return False

def days_per_month(year, month):
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week):
    return day_of_week == 4

def fridays(year, day_of_week):
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
```

```
        if is_friday(day_of_week) and day == 13:
            count += 1
        day_of_week = (day_of_week + 1) % 7
    return count
```

4.p: Přípravy

4.p.1 [database] V této úloze budete pracovat s databázovou tabulkou. Tabulka je dvojice složená z **hlavičky** a seznamu **záznamů**. **Hlavička** obsahuje seznam názvů sloupců. Jeden záznam je tvořen seznamem hodnot pro jednotlivé sloupce tabulky (pro jednoduchost uvažujeme jenom hodnoty typu řetězec). Ne všechny hodnoty v záznamech musí být vyplněny – v tom případě mají hodnotu None.

Vaším úkolem bude nyní otypovat a implementovat následující funkce. Funkce get_header vrátí hlavičku tabulky table.

```
def get_header(table):
    pass
```

Funkce get_records vrátí seznam záznamů z tabulky table.

```
def get_records(table):
    pass
```

Procedura add_record přidá záznam record na konec tabulky table. Můžete předpokládat, že záznam record bude mít stejný počet sloupců jako tabulka.

```
def add_record(record, table):
    pass
```

Predikát is_complete je pravdivý, neobsahuje-li tabulka table žádnou hodnotu None.

```
def is_complete(table):
    pass
```

Funkce index_of_column vrátí index sloupce se jménem name. Můžete předpokládat, že sloupec s jménem name se v tabulce nachází. První sloupec má index 0.

```
def index_of_column(name, header):
    pass
```

Funkce values vrátí seznam platných hodnot (tzn. takových, které nejsou None) v sloupci se jménem name. Můžete předpokládat, že sloupec se jménem name se v tabulce nachází.

```
def values(name, table):
    pass
```

Procedura drop_column smaže sloupec se jménem name z tabulky table. Můžete předpokládat, že sloupec se jménem name se v tabulce nachází.

```
def drop_column(name, table):
    pass
```

Konečně otypujte následující dvě testovací funkce (jejich implementaci neměňte, pouze přidejte typové anotace).

```
def make_empty():
    return ["A", "B", "C", "D"], []

def make_table():
    return (["A", "B", "C"],
            [["a1", "b1", None],
             ["a2", "b2", "c2"],
             ["a3", None, "c3"]])
```

4.p.2 [points] Vraťme se k ukázkovému příkladu 03/points.py, kde Vám byly představeny n-tice. Při takto komplikovaných typech je vhodné funkce otypovat, jak pro čitelnost, tak pro jednodušší hledání chyb. Vaším úkolem bude nyní otypovat funkce i testovací procedury a případně proměnné, tak, aby Vám prošla typová kontrola. Doporučujeme si zavést typové aliasy pro opakující se jednoznačně pojmenovatelné typy.

Funkce distance spočte Euklidovskou vzdálenost dvou bodů a a b.

```
def distance(a, b):
    a_x, a_y, _ = a
    b_x, b_y, _ = b
    return sqrt((a_x - b_x) ** 2 + (a_y - b_y) ** 2)
```

Funkce leftmost_colour v neprázdném seznamu bodů najde barvu „nejlevějšího“ bodu (takového, který má nejmenší x-ovou souřadnici).

```
def leftmost_colour(points):
    x_min, _, result = points[0]

    for x, _, colour in points:
        if x < x_min:
            x_min = x
            result = colour

    return result
```

Dále funkce center_of_gravity dostane jako parametry seznam bodů points a barvu colour; jejím výsledkem bude bod, který se nachází v **těžišti** soustavy bodů dané barvy (a který bude stejné barvy). Vstupní podmínkou je, že points obsahuje alespoň jeden bod barvy colour.

```
def center_of_gravity(points, colour):
    total_x = 0.0
    total_y = 0.0
    count = 0
    for p_x, p_y, p_colour in points:
        if colour == p_colour:
            total_x += p_x
            total_y += p_y
            count += 1

    return (total_x / count, total_y / count, colour)
```

Jako poslední si definujeme funkci average_nonmatching_distance, která spočítá průměrnou vzdálenost bodů různé barvy. Vstupní podmínkou je, že seznam points musí obsahovat alespoň dva různobarevné body.

```
def average_nonmatching_distance(points):
    total = 0.0
    pairs = 0

    for i in range(len(points)):
        for j in range(i):
            _, _, i_colour = points[i]
            _, _, j_colour = points[j]
            if i_colour != j_colour:
                total += distance(points[i], points[j])
                pairs += 1

    return total / pairs
```

4.p.3 [course] V této úloze bude Vaším úkolem implementovat a otypovat následující funkce, které implementují dotazy na školní kurzy. Kurz je reprezentován seznamem dvojic (student, známka), přičemž student je trojice (učo, jméno, semestr) a známka je řetězec z rozsahu A až F.

Funkce failed vrátí seznam studentů kurzu course, kteří z něj mají známku F.

```
def failed(course):
    pass
```

Funkce count_passed vrátí počet studentů, kteří úspěšně ukončili kurz course, tedy z něj nemají známku F. Parametr semester je volitelný: je-li specifikován (není None), funkce vrátí počet úspěšných studentů v daném semestru, jinak vrátí počet všech úspěšných studentů.

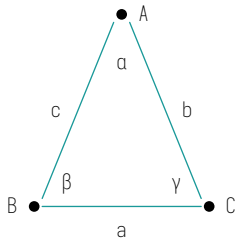
```
def count_passed(course, semester):
    pass
```

Funkce student_grade vrátí známku studenta s učem uco. Pokud takový student v kurzu course není, vrací None.

```
def student_grade(uco, course):
    pass
```

4.p.4 [triangle] V této úloze bude Vaším úkolem rozšířit a otypovat implementaci z ukázky 02/triangle.py.

Strany trojúhelníku značíme a, b, c. Úhel mezi a a b je γ (gamma), mezi b a c je α (alpha) a mezi c a a je úhel β (beta):



1. Prvním úkolem bude implementovat obecnou funkci perimeter, která má volitelné parametry tří stran a tří úhlů trojúhelníku. Je-li to možné z předaných parametrů, funkce spočítá obvod trojúhelníku jednou z metod **SSS**, **ASA**, **SAS**, jinak vrátí None.
2. Druhým úkolem bude otypovat zbytek pomocných funkcí tak, aby Vám prošla typová kontrola. Typ funkce perimeter neměňte.

```
def perimeter(a: float | None,
              b: float | None,
              c: float | None,
              alpha: float | None,
              beta: float | None,
              gamma: float | None) -> float | None:
    pass
```

Funkce perimeter_sss spočte obvod trojúhelníku zadaného třemi stranami.

```
def perimeter_sss(a, b, c):
    return a + b + c
```

Funkce perimeter_sas spočte obvod trojúhelníku zadaného dvěma stranami a nimi sevřeným úhlem.

```
def perimeter_sas(a, angle, b):
    c = sqrt(a ** 2 + b ** 2 - 2 * a * b * cos(radians(angle)))
    return perimeter_sss(a, b, c)
```

Funkce perimeter_asa spočte obvod trojúhelníku zadaného stranou a jí přilehlých úhlů.

```
def perimeter_asa(alpha, c, beta):
    gamma = radians(180 - alpha - beta)
    alpha = radians(alpha)
    beta = radians(beta)
    a = c * sin(alpha) / sin(gamma)
    b = c * sin(beta) / sin(gamma)
    return perimeter_sss(a, b, c)
```

4.p.5 [doctor] V této úloze bude Vaším úkolem implementovat funkce pracující se seznamem pacientů patients u lékaře. Každý pacient má záznam (dvojici), který obsahuje jeho unikátní identifikátor a seznam návštěv s výsledky. Návštěva je reprezentovaná čtveřicí – rokem, kdy pacient navštívil lékaře, a naměřenými hodnotami: pulz, systolický a diastolický tlak. Seznam návštěv pacienta je uspořádaný vzestupně od nejstarší. Můžete předpokládat, že každý pacient má alespoň jeden záznam.

Vaším prvním úkolem bude implementovat a otypovat funkci missing_visits, která zjistí, kteří pacienti nebyli na prohlídce od roku year. Jako výsledek vraťte seznam identifikátorů pacientů.

```
def missing_visits(year, patients):
    pass
```

Dále napište a otypujte funkci patient_reports, která vrátí seznam zpráv o pacientech. Zpráva o pacientovi je čtveřice, která obsahuje záznam o jeho nejvyšším doposud naměřeném pulzu a pro každou měřenou hodnotu informaci, zda se měření dané hodnoty v jednotlivých letech konzistentně zvyšují (True nebo False).

Například zpráva o pacientovi (1, [(2015, 91, 120, 80), (2018, 89, 125, 82), (2020, 93, 120, 88)]) je (93, False, False, True).

```
def patient_reports(patients):
    pass
```

4.p.6 [divisors] Napište čistou funkci, která na vstupu dostane dvě celá kladná čísla rows a cols a vrátí tabulku (dvourozměrný seznam) o rows řádcích a cols sloupcích. V buňce v řádku y a sloupci x bude počet společných dělitelů čísel x a y. Levý horní roh má souřadnice x = y = 1.

Například pro vstup rows = 4, cols = 2 dostaneme tabulku [[1, 1], [1, 2], [1, 1], [1, 2]].

```
def common_divisors(rows, cols):
    pass
```

4.r: Řešené úlohy

4.r.1 [squares] Do programu (který si možná pamatujete z druhého týdne) doplňte typové anotace tak, aby prošel kontrolou nástrojem mypy bez chyb.

Pomocné funkce.

```
def find_slope(points, avg_x, avg_y):
    dividend = 0.0
    divisor = 0

    for i, (x, y) in enumerate(points):
        dividend += ((x - avg_x) * (y - avg_y))
        divisor += (x - avg_x) ** 2

    if divisor == 0:
        return None

    return dividend / divisor

def find_intercept(avg_x, avg_y, beta):
    return avg_y - beta * avg_x
```

První verze má jako vstup dva vektory (seznamy), jeden se souřadnicemi x a druhý se souřadnicemi y. Výsledkem nechť je dvojice (α, β).

```
def regress_vectors(x, y):
    return regress_points([(x[i], y[i]) for i in range(len(x))])
```

Druhá verze má jako parametr seznam dvojic, kde každá dvojice popisuje jeden bod.

```
def regress_points(points):
    avg_x = sum([x for x, _ in points]) / len(points)
    avg_y = sum([y for _, y in points]) / len(points)

    slope = find_slope(points, avg_x, avg_y)

    if slope is None:
        return None

    intercept = find_intercept(avg_x, avg_y, slope)
    return (intercept, slope)
```

Výpočet residuí z dvojice seznamů.

```
def residuals_vectors(x, y, alpha, beta):
    points = [(x[i], y[i]) for i in range(len(x))]
    return residuals_points(points, alpha, beta)
```

Výpočet residuí ze seznamu dvojic.

```
def residuals_points(points, alpha, beta):
    res = []
    for i, (x, y) in enumerate(points):
        res.append(abs(y - beta * x - alpha))
    return res
```

4.r.2 [life] Opět je Vaším úkolem do již hotového programu doplnit typové anotace tak, aby prošel kontrolou nástrojem mypy. Zároveň si zde můžete procvičit porozumění kódu (budete-li vědět, co která funkce dělá, typové

anotace se Vám budou vymýšlet lépe).

```
def cell_value(grid, x, y):
    if 0 <= x < len(grid) and 0 <= y < len(grid):
        return grid[x][y]
    return 0

def live_neighbour_count(grid, x, y):
    assert x < len(grid) and y < len(grid)

    res = 0
    for row in range(x - 1, x + 2):
        for col in range(y - 1, y + 2):
            res += cell_value(grid, row, col)
    return res - grid[x][y]

def next_value(grid, x, y):
    assert x < len(grid) and y < len(grid)

    live_neighbours = live_neighbour_count(grid, x, y)

    if grid[x][y] == 0:
        return 1 if live_neighbours == 3 else 0

    if live_neighbours == 2 or live_neighbours == 3:
        return 1
    return 0

def step(grid):
    assert len(grid) > 0

    res = []
    for i in range(len(grid)):
        res.append([])
        for j in range(len(grid[0])):
            res[i].append(next_value(grid, i, j))
    return res

def life(grid, count):
    assert len(grid) > 0
    assert count >= 0

    world = [curr.copy() for curr in grid]

    for _ in range(count):
        next_step = step(world)
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                world[i][j] = next_step[i][j]

    return world
```

4.r.3 **[predicates]** Tento příklad bude mírně nekonvenční v tom, že nebudete programovat nové funkce. Vaším úkolem bude naopak poznat, co zadaná funkce počítá a napsat testy, které Vaši hypotézu ověří. Každá funkce zde zadaná je predikát a většina má nějakou vstupní podmínku. Samotné funkce i proměnné v nich jsou záměrně pojmenované tak, aby Vám názvy nic neřekly.

```
def f_1(x: int, y: int) -> bool:
    assert y >= 1
    assert x >= 1
    a = 0
    b = 1
    while x > 1:
        c = a + b
        a = b
        b = c
        x -= 1
    return b == y

def test_f_1() -> None:
    pass

def f_2(x: int, y: int) -> bool:
    assert x > 0
    b = 1
    a = x // 2
    while a >= b:
        if x % b == 0:
            y -= 1
        b += 1
    return y <= 1

def test_f_2() -> None:
    pass

def f_3(x: int, y: int) -> bool:
    assert x > 0 and y > 0
    a = 1
    b = 0
    while a <= max(x, y):
        if x % a == 0:
            b += 1
        if y % a == 0:
            b -= 1
        a += 1
    return b > 0

def test_f_3():
    pass

def f_4(x: int, y: int) -> bool:
    for z in range(1, x):
        b = True
        for i in range(2, floor(sqrt(z)) + 1):
            if z % i == 0:
                b = False
        if b:
            y -= 1
    return y == 0

def test_f_4():
    pass

def f_5(x: int) -> bool:
    assert x >= 0
    y = 0
    z = x
    while z > 0:
        y = y * 7 + z % 7
        z = z // 7
    return x == y

def test_f_5():
    pass

def f_6(x: int, y: int) -> bool:
    assert x >= 0
    z = 0
    while x > 0:
        z = z * 2 + (x % 2)
        x = x // 2
    return y == z

def test_f_6() -> None:
    pass

def f_7(x: int, y: int) -> bool:
    assert x >= 0
    z = 2
    while x > 1:
        if x % z == 0:
            y -= 1
        while x % z == 0:
            x = x // z
        z += 1
    return y == 0

def test_f_7() -> None:
    pass

def f_8(x: int, y: int, z: int) -> bool:
    assert x > 0 and y > 0
```

```
    d = 2
    r = 0
    while x > 1 and y > 1:
        if x % d == 0 and y % d == 0:
            x = x // d
            y = y // d
            r += 1
        while x % d == 0:
            x = x // d
        while y % d == 0:
            y = y // d
        d += 1
    return r == z

def test_f_8() -> None:
    pass
```

4.r.4 [poly] † V tomto příkladu se budeme zabývat **polynomy**, které pravděpodobně znáte ze střední školy. Jestli ne, stačí Vám v tuto chvíli vědět, že se jedná o výrazy tvaru

$$P(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0 = \sum_0^n a_i x^i$$

Hodnotám a_i říkáme koeficienty. Koeficienty budeme reprezentovat pomocí zlomků (zlomky proto, že je chceme dělit a násobit, aniž bychom se dopouštěli nepřesnosti spojené s hodnotami typu float). V Pythonu k tomu můžeme použít typ Fraction, který je součástí standardní knihovny.

Polynom jako celek budeme reprezentovat jako seznam koeficientů: na $(n-i)$-tém indexu bude uložena hodnota a_i. Z tohoto indexu je také zřejmé, k jaké mocnině x se koeficient váže (je to x^i).

```
Polynomial = list[Fraction]
```

Vaším úkolem bude implementovat 2 operace: derivaci (angl. differentiation) a integraci. Derivací polynomu $P(x) = \sum_0^n a_i x^i$ je polynom $P'(x) = \sum_0^{n-1} b_i x^i$ kde koeficienty b_i získáme ze vztahu $b_i = (i+1)a_{i+1}$ (pomyslný nulový koeficient b_n do seznamu ukládat nebudeme).

```
def differentiate(poly: Polynomial) -> Polynomial:
    pass
```

Integrace je opačná operace k derivaci: opět uvažujme $P(x) = \sum_0^n a_{n-i} x^i$, pak integrál $\int P(x) = \sum_0^{n+1} c_i x^i$ bude mít koeficienty $c_0 = C$, $c_i = a_{i-1}/i$) kde C je libovolná konstanta. Pro jednoduchost budeme uvažovat $C = 0$.

```
def integrate(poly: Polynomial) -> Polynomial:
    pass
```

Příklad:

$$\begin{aligned} \int P(x) &= 2x^4 + x^3 + 2x^2 + x + C \\ P(x) &= 0 + 8x^3 + 3x^2 + 4x + 1 \\ P'(x) &= 0 + 0 + 24x^2 + 6x + 4 \end{aligned}$$

Totéž se symbolickými koeficienty:

$$\begin{aligned} \int P(x) &= c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \\ P(x) &= 0 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \\ P'(x) &= 0 + 0 + b_2 x^2 + b_1 x + b_0 \end{aligned}$$

Poslední úlohou je ověřit, že operace jsou skutečně vzájemně inverzní. Napište funkci, která toto ověří. Protože derivace „zapomíná“ hodnotu a_0 (při výpočtu nových koeficientů se vůbec nepoužije), ověřit můžeme pouze jedno pořadí složení obou operací. Rozmyslete si které to je. Opačný směr ověřte tak dobře, jak to lze.

```
def check_inverse(poly: Polynomial) -> bool:
    pass
```

4.r.5 [mystery]

1. Popište, co dělá funkce mystery_function(nums).
2. Přepište funkci tak, aby dosáhla stejného výstupu pouze pomocí manipulace prvků ve stávajícím seznamu (tedy bez vytváření nového seznamu)
3. Formulujte vstupní podmínku funkce mystery_function

```
# python
def mystery_function(nums):
    result = [0] * len(nums)
    i = 0
    for num in nums:
        if num % 2 == 0:
            result[i] = num // 2
            i += 1
    for num in nums:
        if num % 2 != 0:
            result[i] = num * 2
            i += 1
    return result
```

Odhalte, co dělá následující funkce a zjednodušte ji.

```
def mysterious_shift(arr):
    result = []
    secret_code = 123456
    cipher_key = 654321

    for essential_index in range(len(arr)):
        data_point = arr[essential_index] + essential_index
        code_combination = data_point + secret_code
        decoded_element = code_combination - secret_code
        key_interaction = decoded_element * cipher_key
        final_element = key_interaction / cipher_key

        distraction_1 = secret_code * cipher_key
        distraction_2 = distraction_1 / cipher_key
        distraction_3 = distraction_2 - secret_code

        final_element += distraction_3 - distraction_3

        for _ in result:
            final_element = final_element * 1

        result.append(final_element)

    return result

def main() -> None:
    pass
```

4.r.6 [precondition] Opět netradiční úloha: tentokrát budete doplňovat vstupní podmínky, opět k funkcím, které jsou zapsané bez jakýchkoliv užitečných názvů nebo komentářů. Vstupní podmínky doplňujte do samostatných funkcí (predikátů) k tomuto účelu nachystaných. Vstupní podmínka musí zaručit, že funkce skončí a splní výstupní podmínku. Zároveň by měla co nejméně omezit použitelnost funkce (tzn. měla by povolit co nejvíce vstupů).

```
def f_1(x_0: int, y: int) -> int:
    assert precondition_1(x_0, y)
    x = x_0
    z = 0
    s = -1 if (x < 0) != (y < 0) else 1
    while abs(x) > 0:
        x -= s * y
        z += s
    assert x_0 // y == z
    return z

def precondition_1(x_0: int, y: int) -> bool:
    return False
```

```
def f_2(x_0: int, y_0: int) -> int:
    assert precondition_2(x_0, y_0)
    x = x_0
    y = y_0
    z = 0
    while x != y:
        x += 1
        y -= 1
        z += 2
    assert x_0 + z == y_0
    return z

def precondition_2(x_0: int, y_0: int) -> bool:
    return x_0 <= y_0 and False

def f_3(x: int, y: int) -> int:
    assert precondition_3(x, y)
    i = 2
    j = 1
    while i <= min(x, -y):
        if x % i == 0 and y % i == 0:
            j = i
        i += 1
    assert j == gcd(x, y)
    return j

def precondition_3(x: int, y: int) -> bool:
    return True

def f_4(x_0: int, y: int) -> tuple[int, int]:
    assert precondition_4(x_0, y)
    x = x_0
    z = 0
    while x >= y:
        x -= y
        z += 1
    assert z * y + x == x_0
    assert z >= 0 and x >= 0
    return (z, x)

def precondition_4(x_0: int, y: int) -> bool:
    return False
```

4.v: Volitelné úlohy

4.v.1 [fibnum] Fibonácci používají k zápisu kladných celých čísel Fibonacciho soustavu. Ta používá jen dvě číslice 0 a 1; řády čísel ovšem nejsou mocniny dvou jako v klasické dvojkové soustavě, ale jsou postupně zprava 1, 2, 3, 5, 8, 13, … (Jde tedy o Fibonacciho čísla bez úvodních 0 a 1.) Některá čísla je takto možno zapsat dvěma různými způsoby, např. číslo 17 se zapíše buď jako $(100101)_\phi$ nebo jako $(11101)_\phi$. Platí totiž $17 = 13+3+1 = 8+5+3+1$. Proto se zavádí tzv. **kanonický zápis** čísla ve Fibonacciho soustavě, kdy se zakazuje mít vedle sebe dvě jedničky.

Čistá funkce fib_ones spočítá, kolik jedniček je v kanonickém Fibonacciho zápisu nezáporného celého čísla num.

Příklady: V kanonickém Fibonacciho zápisu čísla 17 jsou tři jedničky, viz výše. V kanonickém Fibonacciho zápisu čísla 34 je jedna jednička (je to totiž přímo Fibonacciho číslo). V kanonickém Fibonacciho zápisu čísla 101 jsou čtyři jedničky, protože platí $101 = 89 + 8 + 3 + 1$.

```
def fib_ones(num):
    pass
```

4.v.2 [magic] Magický čtverec je dvourozměrná matice vzájemně různých kladných celých čísel, pro niž platí, že součty čísel v každém řádku, každém sloupci a obou hlavních úhlopříčkách jsou stejné. Klasickým příkladem je magický čtverec 3x3: 8 1 6 3 5 7 4 9 2 v němž se součty všech řádků, všech sloupců a obou diagonál rovnají 15.

Napište predikát is_magic_square, který na vstupu dostane dvourozměrné pole celých čísel a ověří, že se jedná o magický čtverec.

```
def is_magic_square(square: list[list[int]]) -> bool:
    pass


def main() -> None:
    assert is_magic_square([[8, 1, 6], [3, 5, 7], [4, 9, 2]])
    assert is_magic_square([])
    assert is_magic_square([[1]])
    assert not is_magic_square([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    assert not is_magic_square([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
    assert is_magic_square([[16, 2, 3, 13], [5, 11, 10, 8],
                            [9, 7, 6, 12], [4, 14, 15, 1]])
```

4.v.3 [gambling] Čistá funkce gambling_score ohodnotí výsledek hozený na kostkách (neprázdný seznam celých čísel od 1 do 6 včetně) takto:

Trojice stejných čísel se boduje jako 100× hozené číslo, kromě trojice jedniček, která je za 1000. Čtveřice stejných čísel se počítá za dvojnásobek hodnoty trojice, pětice se počítá za dvojnásobek hodnoty čtveřice atd. Pokud po započítání všech trojic, čtveřic, pětic atd. zbudou nějaké (dosud nezapočítané) jedničky a pětky, počítá se každá jednička za sto bodů, každá pětka za padesát bodů. Získané body se sečtou.

Příklad: Pro vstup [1, 1, 1, 1, 5, 3, 3, 3, 4] funkce vrátí 2350 (čtveřice jedniček za 2000 bodů, trojice trojek za 300 bodů, jedna pětka za 50). Pro vstup [2, 2, 5, 2, 2, 5, 2, 2] funkce vrátí 1700 (šestice dvojek za 1600 bodů, dvě pětky za 100). Pro vstup [2, 2, 3, 4, 6, 6] funkce vrátí 0 (není zde žádná trojice ani lepší skupina stejných čísel, žádné jedničky, žádné pětky).

Všimněte si zejména, že na pořadí čísel v seznamu nezáleží a že počítáme vždy maximální množství výskytů daného čísla (tedy poté, co jsme v prvním příkladu započítali čtveřici jedniček za 2000 bodů, už neuvažujeme o tom, kolik trojic jedniček v seznamu je).

```
def gambling_score(dice):
    pass
```

Část S.1: Sada úloh k prvnímu bloku

V prvním bloku jsou následující domácí úkoly:

- a_clock – vykreslení ciferníku hodin pomocí želví grafiky,
- b_primes – rozklad na prvočinitele,
- c_race – hra ve stylu „Člověče, nezlob se“ s číselnou reprezentací,
- d_mancala – mankalová hra,
- e_2048 – jednorozměrná varianta hry 2048,
- f_freecell – zhodnocení stavu hry FreeCell.

První tři úkoly vyžadují pouze znalosti základních příkazů a celočíselné aritmetiky; zbývající tři úkoly používají seznamy.

S.1.a: primes

Napište čistou funkci nth_smallest_prime_divisor, která vrátí index-té nejmenší prvočíslo vyskytující se v prvočíselném rozkladu čísla num. Pokud se v rozkladu vyskytuje některé prvočíslo vícekrát, počítáme všechny jeho výskyty, tedy např. v čísle $2 \cdot 2 \cdot 3 \cdot 3 \cdot 3 \cdot 5 = 540$ je třetím nejmenším prvočíslem číslo 3. Pokud má num méně než index prvočísel v rozkladu, funkce vrátí None.

Předpokládejte, že num i index jsou kladná celá čísla. Zde indexujeme od 1, tedy první prvočíslo v rozkladu má index 1.

Je potřeba, aby vaše funkce fungovala rozumně rychle i pro velmi velká čísla, u nichž je hledané prvočíslo malé. (Není třeba vymýšlet zvláště chytrá řešení, jen je třeba nedělat zbytečnou práci navíc.)

```
def nth_smallest_prime_divisor(num, index):
    pass
```

S.1.b: race

Uvažujme hru čtyř hráčů s následujícími pravidly:

- herní plán je jednorozměrný, s neomezenou délkou a vyznačeným startovním políčkem;
- každý hráč má jednu figurku, na začátku umístěnou na startovním políčku;
- hráči střídavě hází kostkou a posunují své figurky o hozené číslo;
- pokud by hráčova figurka měla vstoupit na políčko obsazené figurkou jiného hráče, tato figurka je „vykopnuta“ (jako v Člověče, nezlob se) zpět na start.

Situaci na herním plánu budeme reprezentovat pomocí nezáporného celého čísla tak, že jeho zápis v pětkové soustavě reprezentuje obsazenost jednotlivých políček bez startovního políčka. Číslice 0 reprezentuje prázdné políčko, číslice 1–4 pak reprezentují obsazenost figurkou konkrétního hráče. Pohyb figurek přitom v pětkovém zápisu probíhá „zprava doleva“, tedy směrem od nižších řádů k vyšším.

Příklady:

Všechny figurky jsou na startu – stav reprezentovaný číslem 0.

Figurky hráčů 1 a 3 jsou na startu, figurka hráče 2 je dvě políčka od startu, figurka hráče 4 je šest políček od startu. Tento stav je reprezentovaný číslem $(400020)_5 = 4 \cdot 5^5 + 2 \cdot 5^1 = 12510$.

Napište čistou funkci play, která na plánu reprezentovaném číslem arena provede jeden tah hráče player o zadaný hod kostkou throw a vrátí číslo reprezentující nový stav hry.

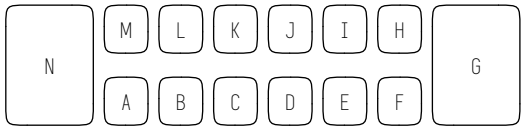
Předpokládejte, že arena je validní stav hry (tj. nezáporné celé číslo, v jehož pětkovém zápisu se objevuje každá z číslic 1–4 nejvýše jednou), že player je jedno z čísel 1, 2, 3, 4 a že throw je kladné celé číslo. (Nemusí být nijak shora omezené; předpokládejte, že máme kostky s různě velkými čísly.)

```
def play(arena, player, throw):
    pass
```

S.1.c: mancala

Mankala[10] je souborné označení deskových her pro dva hráče, jejichž společným znakem je přemisťování kuliček (kamínků, pecek, apod.) mezi důlky. V tomto domácím úkolu si naprogramujete jednoduchou variantu takové hry – pravidla jsou inspirována hrou **Kalaha**[11], resp. jednou z jejích obměn.

Hrací deska sestává z dvou řad menších důlků (jejich počet je parametrem hry, viz níže) a dvou větších důlků vlevo a vpravo. Vypadá tedy např. takto (počet menších důlků v každé řadě je zde šest):



Hru hrají dva hráči, kteří sedí proti sobě. Každému hráči patří menší důlky na jeho straně a větší důlek vpravo – tento větší důlek nazýváme hráčovou **bankou**. Na začátku hry je v každém menším důlku předem určený počet kuliček (toto je druhý parametr hry), banky jsou prázdné. Hra probíhá po kolech, přičemž se hráči střídají. Průběh každého kola je následující:

- Hráč si vybere jeden ze svých menších důlků, který obsahuje nějaké kuličky. Pokud jsou všechny důlky hráče prázdné, hra končí (viz níže).
- Hráč vezme všechny kuličky z vybraného důlku a začne je po jedné rozdělovat do následujících důlků proti směru hodinových ručiček, včetně svého banku, ale **ne do banku soupeře**. Pokud tedy např. spodní hráč vzal kuličky z důlku C, pak je bude postupně rozdělovat do důlků D, E, F, G, H, I, J, K, L, M, A, B, C, atd., dokud mu nějaké kuličky budou zbývat.
- Pokud při rozdělování padla poslední kulička do prázdného menšího důlku na straně aktuálního hráče **a jeho oponent má v protějším důlku nějaké kuličky**, sebere hráč svou poslední kuličku a **všechny** kuličky v protějším důlku a přesune je do své banky.
- Pokud při rozdělování padla poslední kulička do hráčovy banky, v dalším kole hraje tentýž hráč znovu; v opačném případě se hráči vystřídají.

Hra končí, když má hráč, který je na tahu, všechny menší důlky prázdné. Jeho protivník si pak přesune všechny kuličky ze svých menších důlků do své banky. Vyhrává ten hráč, který má v bance více kuliček.

Hrací desku reprezentujeme pomocí dvou seznamů nezáporných celých čísel. Každý seznam představuje důlky jednoho z hráčů (postupně zleva doprava z hráčova pohledu), přičemž počet kuliček v bance hráče je posledním prvkem seznamu. Desce naznačené výše tedy odpovídají seznamy [A, B, C, D, E, F, G] a [H, I, J, K, L, M, N].

Abyste si hru mohli vyzkoušet (poté, co úlohu vyřešíte), je vám k dispozici soubor game_mancala.py, který vložte do stejného adresáře, jako je soubor s vaším řešením, případně jej upravte dle komentářů na jeho začátku a spusťte. Kliknutím na jeden z důlků se provede tah, klávesa R hru resetuje a Q ukončí.

Implementujte nejprve čistou funkci init, která vrátí dvojici seznamů reprezentujících hrací desku se size menšími důlky, v nichž je na začátku start kuliček. Banky obou hráčů jsou prázdné. Předpokládejte, že size

[10] https://en.wikipedia.org/wiki/Mancala
[11] https://en.wikipedia.org/wiki/Kalah

i start jsou kladná celá čísla.

```
def init(size, start):
    pass
```

Dále napište proceduru play, která odehraje jedno kolo hry. Parametr our je seznam reprezentující stranu aktuálního hráče, parametr their je seznam reprezentující stranu protivníka. Předpokládejte, že tyto seznamy mají stejnou délku větší než 1 a že obsahují pouze nezáporná celá čísla. Parametr position (celé číslo) určuje, který důlek se má vybrat (0 je důlek nejvíce vlevo z pohledu hráče).

Pokud je position mimo platný rozsah, procedura nic nemodifikuje a vrátí konstantu INVALID_POSITION. Pokud je position indexem prázdného důlku, procedura nic nemodifikuje a vrátí konstantu EMPTY_POSITION. Jinak procedura modifikuje seznamy dle pravidel hry a vrátí buď konstantu PLAY_AGAIN nebo ROUND_OVER, podle toho, jestli má aktuální hráč hrát znovu nebo už skončil. Tyto konstanty jsou už definovány; nijak je neměňte.

```
INVALID_POSITION = 0
EMPTY_POSITION = 1
ROUND_OVER = 2
PLAY_AGAIN = 3

def play(our, their, position):
    pass
```

S.1.d: 2048

V tomto domácím úkolu si naprogramujete zjednodušenou variantu hry **2048**[12]. Na rozdíl od původní hry budeme uvažovat jen jednorozměrný hrací plán, tj. jeden řádek.

Hrací plán budeme reprezentovat pomocí seznamu nezáporných celých čísel; nuly budou představovat prázdná místa. Například seznam [2, 0, 0, 2, 4, 8, 0] reprezentuje následující situaci:

2			2	4	8	

Základním krokem hry je posun doleva nebo doprava. Při posunu se všechna čísla „sesypou“ v zadaném směru, přičemž dvojice stejných číslic se sečtou. Posunem doleva se tedy uvedený seznam změní na [4, 4, 8, 0, 0, 0, 0].

Abyste si hru mohli vyzkoušet (poté, co úlohu vyřešíte), je vám k dispozici soubor game_2048.py, který vložte do stejného adresáře, jako je soubor s vaším řešením, případně jej upravte dle komentářů na jeho začátku a spusťte. Hra se ovládá šipkami doleva a doprava, R hru resetuje a Q ukončí.

Napište proceduru slide, která provede posun řádku reprezentovaného seznamem row, a to buď doleva (pokud má parametr to_left hodnotu True) nebo doprava (pokud má parametr to_left hodnotu False). Procedura přímo modifikuje parametr row a vrací True, pokud posunem došlo k nějaké změně; v opačném případě vrací False.

```
def slide(row, to_left):
    pass
```

S.1.e: freecell

FreeCell[13] je pasiánsová karetní hra, kterou možná znáte jako součást operačních systémů jisté společnosti se sídlem v Redmondu. Ve hře se používá klasický balíček 52 karet se čtyřmi barvami (suits) a třinácti hodnotami (ranks) od esa po krále. Hrací pole obsahuje:

- volná pole (cells) – typicky čtyři, ve variantách jedno až deset,
- domácí pole (foundations) – vždy přesně čtyři, do každého z nich se odkládají karty ve stejné barvě, postupně od esa po krále,
- sloupce (cascades) – typicky osm, ve variantách čtyři až deset; do sloupců se na začátku rozdají všechny karty.

Povolené přesuny karet jsou následující:

- je možno přesouvat karty z volných polí a spodní karty sloupců;
- na prázdné volné pole a do prázdného sloupce je možno položit libovolnou kartu;
- na prázdné domácí pole je možno položit eso libovolné barvy;
- na kartu v domácím poli je možno položit kartu stejné barvy s hodnotou přesně o jednu vyšší;
- na spodní kartu sloupce je možno položit další kartu, pokud je její hodnota přesně o jednu nižší a pokud se její barva liší (ve smyslu červená / černá).

Karty budeme reprezentovat jako dvojice (rank, suit), kde rank je jedno z čísel 1 až 13 (pro karty s hodnotami 1, 11, 12, 13 máme níže zavedeny konstanty) a suit je jedno z čísel 0 až 3 (postupně reprezentující srdce, kára, piky a kříže; níže opět reprezentované konstantami). Zde uvedené konstanty nijak neměňte.

```
ACE, JACK, QUEEN, KING = 1, 11, 12, 13
HEARTS, DIAMONDS, CLUBS, SPADES = 0, 1, 2, 3
```

Implementujte predikát can_move, tj. jestli je v zadané situaci možné provést přesun nějaké karty. Situace je reprezentována třemi seznamy, jejichž prvky jsou buď karty nebo None.

- cascades je seznam spodních karet sloupců (None je prázdný sloupec),
- cells je seznam karet na volných polích (None je prázdné pole),
- foundation je seznam horních karet na domácích polích (None je opět prázdné pole).

Předpokládejte, že vstupní situace je skutečnou situací ve hře (např. není možné, aby se někde objevila stejná karta dvakrát).

```
def can_move(cascades, cells, foundation):
    pass
```

[12] https://play2048.co/

[13] https://en.wikipedia.org/wiki/FreeCell

Část 5: Datové struktury I

V této kapitole se budeme opět zabývat zabudovanými datovými strukturami: z třetí kapitoly již známe **seznam** a **n-tici**, tento týden přibudou **zásobník** (stack), **slovník** (dictionary) a **množina** (set).

Přípravy:

1. attendance – práce s množinou
2. worktime – práce se slovníkem
3. sublist – algoritmus nad seznamy čísel
4. sum – hledání součtu ve dvojici seznamů
5. course – práce se slovníkem známek
6. colours – práce se slovníkem barev v reprezentaci RGB

5.1: Programovací jazyk

Tato kapitola přidává dva nové typy složených hodnot:[14]

- **množina** – set – podobně jako seznam obsahuje vnitřní hodnoty, s tím rozdílem, že v množině nemají hodnoty pevně určené pořadí, a každá se v dané množině může objevit nejvýše jednou,
- **slovník** – dict – obsahuje **klíče** (podobně jako v množině se daný klíč může objevit nejvýše jednou) a ke každému klíči právě jednu **přidruženou hodnotu** (obvykle nazýváme prostě **hodnota**, a mluvíme o dvojicích klíč – hodnota).

Pro hodnoty, které vkládáme do množin, nebo je používáme jako klíče ve slovníku, platí důležité omezení: taková hodnota **nesmí** mít vnitřní přiřazení, ani jiné operace, které mohou vnitřně danou hodnotu změnit. Zejména tedy nelze takto používat **seznamy**, ale ani slovníky nebo množiny. Přípustná jsou naopak zejména celá čísla, řetězce a n-tice z nich složené.

S novými typy hodnot přidáváme i nové tvary výrazů (literály, přístup k přidruženým hodnotám, množinové operace) a příkazů (přiřazení, for cyklus) a nové zabudované podprogramy.

5.1.1 Literály Jak jsme již zvyklí, hodnoty typu **množina** a **slovník** můžeme do programu zapsat pomocí speciálních výrazů – literálů (podobně jako tomu bylo u seznamů, n-tic a řetězců). Tyto literály mají tvar:

- {} je **prázdný slovník** (pozor, nikoliv množina!),
- {$klíč_1$: $hodnota_1$, $klíč_2$: $hodnota_2$, …} je **slovník**, kde $klíč_i$ jsou **výrazy**, kterých vyhodnocením vzniknou **klíče**, přičemž vyhodnocením výrazu $hodnota_i$ vznikne vždy hodnota přidružená odpovídajícímu klíči (vyhodnotí-li se dva různé výrazy $klíč_i$ na stejný výsledek, použije se dvojice více vpravo),
- {$hodnota_1$, $hodnota_2$, …} reprezentuje **množinu** s prvky, které vzniknou vyhodnocením **výrazů** $hodnota_i$.

Prázdná množina literál nemá. Chceme-li vytvořit prázdnou množinu, použijeme k tomu zabudovanou funkci set() bez parametrů.

5.1.2 Výrazy Přístup k přidružené hodnotě uložené ve slovníku[15] zapisujeme výrazem tvaru slovník[klíč], kde:

- slovník je **výraz** který se vyhodnotí na hodnotu typu slovník a
- klíč je **výraz**, který je nejprve vyhodnocen, poté je výsledná hodnota ve slovníku vyhledána,
- výraz slovník[klíč] jako celek se pak vyhodnotí na odpovídající přidruženou hodnotu byl-li klíč ve slovníku nalezen, v opačném případě je program ukončen s chybou.

Oproti seznamům jsou jak množiny tak slovníky vybaveny **efektivním** dotazem na přítomnost prvku (u slovníku klíče), a to výrazy tvaru:

```
hodnota in množina
klíč in slovník
```

kde hodnota, množina, klíč a slovník jsou **podvýrazy** a výsledkem je **pravdivostní hodnota**.

5.1.3 Zabudované podprogramy Objekty typu **slovník** mají tyto zabudované metody:

- d.keys() – výsledkem je speciální hodnota, kterou lze pouze iterovat nebo převést na seznam (viz níže), a která obsahuje pouze klíče ve slovníku přítomné (bez přidružených hodnot),
- d.values() – analogicky, ale pro přidružené hodnoty,
- d.items() – taktéž, ale obsahuje dvojice (klíč, hodnota),
- d.get(k) nebo d.get(k, fallback) – vyhledá klíč k v slovníku, a vyhodnotí se na odpovídající hodnotu, je-li tato přítomna, jinak na None (první tvar) nebo na fallback (druhý tvar),
- d.pop(k) – odstraní ze slovníku klíč k (včetně přidružené hodnoty),
- d.copy() – vytvoří kopii slovníku.

Objekty typu **množina** pak mají tyto zabudované metody:

- s.add(v) – vloží do množiny hodnotu v (byla-li již přítomna, nestane se nic),
- s.remove(v) – odstraní hodnotu v (není-li hodnota přítomna, program je ukončen s chybou),

Pro vytváření hodnot přidáváme několik zabudovaných **čistých funkcí**:

- list(x) – převede hodnotu x na seznam, kde x může být:
  - množina,
  - výsledek volání d.keys(), d.values() nebo d.items() na slovníku d,
- set() – vytvoří prázdnou množinu,
- set(l) – převede seznam l na množinu,
- dict(l) – převede seznam dvojic l na slovník.

5.1.4 Příkazy Pro práci s prvky množin a s klíči, hodnotami a dvojicemi (klíč, hodnota) ve slovníku lze použít for cykly těchto tvarů:

```
for vazby in množina:
    příkazy

for vazby in slovník.keys():
    příkazy

for vazby in slovník.items():
    příkazy

for vazby1, vazby2 in slovník.items():
    příkazy
```

Kde vazby je vždy buď **jméno** nebo **rozbalení** a množina a slovník jsou **výrazy**. V posledním uvedeném případě je nutné případné rozbalení uzávorkovat, například:

```
for shape, (x, y) in centers.items():
    pass
```

Posledním novým prvkem je vnitřní přiřazení do slovníku:

```
slovník[klíč] = hodnota
```

kde slovník, klíč i hodnota jsou **výrazy**. Byl-li klíč již ve slovníku přítomen, jeho přidružená hodnota se změní na výsledek vyhodnocení výrazu hodnota. V opačném případě je klíč do slovníku přidán (pozor, v tomto se slovníky liší od seznamů).

5.p: Přípravy

5.p.1 [attendance] V tomto příkladu budeme pracovat se systémem docházky jedné fiktivní firmy. Při příchodu do práce si musí každý zaměstnanec

[14] Výše zmíněný **zásobník** nemá samostatný datový typ: lze jej přímo reprezentovat pomocí seznamu.

[15] Zápis je analogický k indexaci seznamů a řetězců. Oproti těmto již známým typům ale slovníky „indexujeme“ **klíčem**, který **nemusí** být celé číslo, a i v případě, kdy jím celé číslo je, **nemusí** klíče tvořit spojitou řadu začínající nulou. Množinu indexovat nelze.

pípnout kartičkou u vchodu a zaznamenat tak svůj příchod. Při odchodu zase stejně musí zaznamenat, že z práce odešel.

Čidlo u dveří pak do firemního systému zaznamená data o docházce zaměstnance. Každý záznam je trojice obsahující kód zaměstnance, časovou známku a typ záznamu - příchod nebo odchod.

```
EmployeeId = str  # kód zaměstnance
TimeStamp = int  # počet sekund od nějakého pevného bodu
RecordType = bool  # typ záznamu

ENTRY = True
LEAVE = False

MachineRecord = tuple[EmployeeId, TimeStamp, RecordType]
```

Bohužel, někteří zaměstnanci zapomínají zaznamenávat svou docházku. Vaším úkolem je napsat čistou funkci employees_with_missing_records, která projde seznam záznamů, a vrátí množinu obsahující kódy těch zaměstnanců, pro které existuje v seznamu nějaká nesrovnalost – buď z práce odešli, aniž by do ní přišli, nebo přišli do práce vícekrát bez záznamu o odchodu. Seznam záznamů začíná v situaci, kdy žádný zaměstnanec v práci není. Můžete počítat s tím, že seznam je seřazený podle času od nejstaršího záznamu po nejnovější.

```
def employees_with_missing_records(
        records: list[MachineRecord]) -> set[EmployeeId]:
    pass
```

5.p.2 [worktime] V tomto příkladu budeme opět pracovat se systémem docházky (data mají stejný formát i význam).

```
EmployeeId = str  # kód zaměstnance
TimeStamp = int  # počet sekund od nějakého pevného bodu
RecordType = bool  # typ záznamu

ENTRY = True
LEAVE = False

MachineRecord = tuple[EmployeeId, TimeStamp, RecordType]
```

Na základě odpracovaných hodin za jeden měsíc firma počítá mzdu pro zaměstnance. Napište čistou funkci seconds_spent_working, která zjistí, kolik sekund každý zaměstnanec odpracoval. Můžete počítat s tím, že vstupní seznam je seřazený podle časových známek od nejstaršího záznamu po nejnovější, že se v něm nevyskytují žádné nesrovnalosti, že záznamy začínají v situaci, kdy žádný zaměstnanec v práci není a že každý zaměstnanec, který do práce přišel, z ní také později odešel.

Nápověda: odečtením dvou časových známek zjistíte, kolik sekund uplynulo mezi nimi.

```
def seconds_spent_working(
        records: list[MachineRecord]) -> dict[EmployeeId, int]:
    pass
```

5.p.3 [sublist] V tomto příkladu dostanete dva seznamy obsahující celá čísla. Vaším úkolem je napsat čistou funkci largest_common_sublist_sum, která najde takový společný podseznam seznamů left a right, který má největší celkový součet, a tento součet vrátí.

Podseznamem seznamu S myslíme takový seznam T, pro který existuje číslo k takové, že platí S[k + i] == T[i] pro všechna i taková, že $0 \leq i < len(T)$. Například seznam [1, 2] je podseznamem seznamu [0, 1, 2, 3], kde k = 1.

Složitost smí být v nejhorším případě až kubická vzhledem k délce delšího vstupního seznamu.

```
def largest_common_sublist_sum(left: list[int], right: list[int]) -> int:
    pass
```

5.p.4 [sum] Vaším prvním úkolem je napsat predikát sum_to_exactly, který rozhodne, zda se v seznamu left nachází nějaký prvek x a v seznamu right nějaký prvek y tak, že platí x + y == to.

Řešení, kde bude počet kroků výpočtu úměrný součinu délek obou seznamů, je vyhovující.[16]

```
def sum_to_exactly(left: list[int], right: list[int], to: int) -> bool:
    pass
```

Dále napište predikát sum_to_at_least, který rozhodne, zda se v seznamu left nachází nějaký prvek x a v seznamu right nějaký prvek y tak, že platí x + y >= at_least. V tomto případě vyžadujeme složitost lineární vzhledem k délce delšího seznamu.

```
def sum_to_at_least(left: list[int], right: list[int], at_least: int)
-> bool:
    pass
```

5.p.5 [course] Známky studentů z jednoho předmětu jsou uloženy ve slovníku, kde klíčem je UČO studenta a hodnotou je známka zadaná jako písmeno. Možná hodnocení jsou 'A' až 'F', dále, 'N', 'P', 'X', 'Z' a '-'.

Napište čistou funkci modus, jejímž vstupem bude slovník známek a výstupem bude jejich modus, tedy nejčastější hodnota. Předpokládejte, že známek se stejnou četností může být více, takže funkce bude vždy vracet množinu známek, a to i v případě, že je nejčastější hodnota určena jednoznačně. V případě, že je vstupní slovník prázdný, bude výstupem prázdná množina.

[16] Existuje lepší řešení tohoto příkladu se složitostí $n \cdot \log n$ vzhledem k délce většího seznamu. Toto řešení ale vyžaduje seřazení seznamů.

```
def modus(marks: dict[int, str]) -> set[str]:
    pass
```

Dále napište predikát check, který ověří, že známky jsou smysluplné, tedy že odpovídají buďto předmětu ukončenému zkouškou (známky 'A' - 'F', nebo 'X'), kolokviem (známky 'P' nebo 'N'), anebo zápočtem (známky 'Z' nebo 'N'). Hodnocení '-' je možné u jakéhokoliv způsobu hodnocení. Klasifikované zápočty neuvažujeme.

```
def check(marks: dict[int, str]) -> bool:
    pass
```

5.p.6 [colours] V tomto příkladu budeme pracovat s RGB kódy různých barev. Tyto kódy jsou uloženy ve slovníku, kde klíčem je řetězec - název barvy, a hodnota je trojice celých čísel, které představují hodnoty červené, zelené a modré složky.

Vaším úkolem je napsat čistou funkci, která na vstupu dostane slovník barev a trojici celých čísel z rozsahu 0-255 a vrátí množinu názvů, které jsou zadané trojici nejblíže (množina bude obsahovat více prvků pouze v případě, že několik různých barev je od té zadané stejně daleko).

Blízkost barev budeme měřit pomocí tzv. Manhattanské vzdálenosti, která je dána součtem absolutních hodnot rozdílů na jednotlivých souřadnicích. Například pro trojice

```
A = (150, 0, 65)
B = (120, 30, 100)
```

je Manhattanská vzdálenost rovna

```
|150 - 120| + |0 - 30| + |65 - 100| = 30 + 30 + 35 = 95
```

```
Colour = tuple[int, int, int]

def nearest_colour(names: dict[str, Colour],
                   colour: Colour) -> set[str]:
    pass
```

5.r: Řešené úlohy

5.r.1 [transitive] Binární relací nad danou množinou je množina dvojic prvků z této množiny. Daná relace se pak nazývá tranzitivní, platí-li pro všechny dvojice $(a, b), (b, c)$ z této relace, že se v relaci nachází i dvojice (a, c). V této úloze budeme pracovat s relacemi nad celými čísly.

Napište predikát, který rozhodne, je-li zadaná relace tranzitivní.

```
def is_transitive(relation: set[tuple[int, int]]) -> bool:
    pass
```

5.r.2 **[setops]** Vaším úkolem bude naprogramovat základní množinové operace (zatím máme k dispozici pouze operace, které pracují vždy s jedním prvkem). U každé operace si rozmyslete, kolik kroků provede vzhledem k velikostem obou vstupních množin.

První a v nějakém smyslu nejjednodušší operací je sjednocení. Nejprve implementujte sjednocení jako čistou funkci, poté jako proceduru, která rozšíří stávající množinu o prvky nějaké další (a implementuje tedy sjednocení „in situ“). Srovnejte jejich složitost.

```
def set_union(a: set[int], b: set[int]) -> set[int]:
    pass
```

```
def set_update(to_extend: set[int], other: set[int]) -> None:
    pass
```

Druhou standardní operací je průnik. Ten je o něco složitější a také je na místě zvážit rozdíl mezi čistou verzí, která sestrojí novou množinu, a procedurou, která zmenší množinu stávající. Dejte pozor na to, že tu stejnou množinu není dovoleno zároveň jak měnit tak procházet.

```
def set_intersect(a: set[int], b: set[int]) -> set[int]:
    pass
```

```
def set_keep(to_reduce: set[int], other: set[int]) -> None:
    pass
```

5.r.3 **[setdiff]** Uvažme nyní operaci rozdílu – opět v čisté i procedurální verzi. Opět srovnejte efektivitu obou implementací vzhledem k velikosti obou parametrů.

```
def set_difference(a: set[int], b: set[int]) -> set[int]:
    pass
```

```
def set_remove(to_reduce: set[int], other: set[int]) -> None:
    pass
```

Množinový rozdíl má jednu zajímavou variaci – tzv. symetrický rozdíl, kdy konstruujeme množinu, která obsahuje prvky, které náleží do právě jedné vstupní množiny. Opět implementujte obě verze. Symetrický rozdíl je možné složit z ostatních množinových operací mnoha způsoby – rozmyslete si, které fungují lépe a které hůře.

```
def set_symmetric_diff(a: set[int], b: set[int]) -> set[int]:
    pass
```

```
def set_symmetric_inplace(to_change: set[int],
                          other: set[int]) -> None:
    pass
```

5.r.4 **[maps]** V tomto příkladu budeme pracovat se slovníky. Slovník může mimo jiné reprezentovat zobrazení: klíč se zobrazí na příslušnou hodnotu. Naprogramujte čistou funkci image, které předáme slovník f, který reprezentuje zobrazení, a množinu values. Výsledkem bude obraz množiny values – tedy množina hodnot, na které se hodnoty z množiny values zobrazí.

```
def image(f: dict[int, int], values: set[int]) -> set[int]:
    pass
```

Podobně funkce preimage spočítá vzor zadané množiny values (množinu hodnot, které f zobrazí na některý prvek množiny values):

```
def preimage(f: dict[int, int], values: set[int]) -> set[int]:
    pass
```

Dále naprogramujte čistou funkci compose, které vstupem budou dvě zobrazení (slovníky) f a g a výsledkem bude slovník, který reprezentuje zobrazení f ∘ g. Vstupní podmínkou je, že f je definováno pro každou hodnotu z obrazu g.

```
def compose(f: dict[int, int], g: dict[int, int]) -> dict[int, int]:
    pass
```

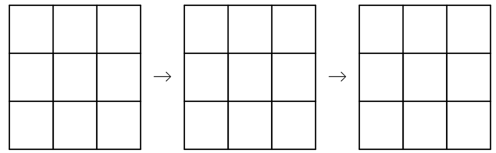
Konečně naprogramujte čistou funkci kernel, které vstupem bude zobrazení (slovník) f a výsledkem bude relace ekvivalence R (množina dvojic) taková, že $(x, y) \in R$ právě když $f(x) = f(y)$.

```
def kernel(f: dict[int, int]) -> set[tuple[int, int]]:
    pass
```

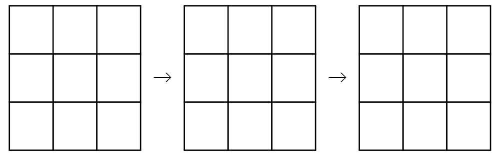
5.r.5 **[life]** Vaším úkolem je naprogramovat tzv. „hru života“ – jednoduchý dvourozměrný celulární automat. Simulace běží na čtvercové síti, kde každá buňka je mrtvá (hodnota 0) nebo živá (hodnota 1). V každém kroku se přepočte hodnota všech buněk, a to podle toho, zda byly v předchozím kroku živé a kolik měly živých sousedů (z celkem osmi, tzn. včetně úhlopříčných):

stav	živí sousedé	výsledek
živá	0–1	mrtvá
živá	2–3	živá
živá	4–8	mrtvá
mrtvá	0–2	mrtvá
mrtvá	3	živá
mrtvá	4–8	mrtvá

Příklad krátkého výpočtu:



Jiný (periodický) výpočet je například:



Napište čistou funkci, která dostane jako parametry počáteční stav hry (jako množinu dvojic, která reprezentuje souřadnice živých buněk) a počet kroků, a vrátí stav hry po odpovídajícím počtu kroků.

```
def life(cells: set[tuple[int, int]],
         n: int) -> set[tuple[int, int]]:
    pass
```

5.v: Volitelné úlohy

5.v.1 **[bugs]** Budeme zkoumat řadu vedle sebe sedících světlušek. Každá světluška má energii, která se vyjadřuje nezáporným celým číslem. Bude nás zajímat vývoj této energie v čase, přičemž v každém kroku dojde k následujícímu:

- Energie všech světlušek se zvětší o 1.
- Světlušky, které mají energii větší než 3, se rozsvítí. To způsobí, že se energie jejich sousedních světlušek zvýší o další 1. To může způsobit jejich rozsvícení (pokud dosud nebyly rozsvícené) atd.
- Energie všech světlušek, které se v tomto kroku rozsvítily, se sníží na 0. Všechny rozsvícené světlušky zhasnou.

Máme-li tedy na začátku světlušky ve stavu [0, 2, 0, 2, 0], v následujícím kroku budou ve stavu [1, 3, 1, 3, 1] a dále pak [3, 0, 0, 0, 3].

Čistá funkce light_bugs vrátí seznam seznamů reprezentujících prvních time kroků pozorování světlušek, jejichž počáteční energie je daná parametrem start. Předpokládejte, že se start skládá jen z čísel od 0 do 3 včetně, má délku alespoň dvě a že time je kladné celé číslo.

```
def light_bugs(start, time):
    pass
```

Příklad: pro vstup ([0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0], 7) funkce vrátí následující seznam:

```
example = [[0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0],
           [1, 1, 1, 1, 2, 0, 2, 1, 1, 1, 1],
           [2, 2, 2, 2, 3, 1, 3, 2, 2, 2, 2],
           [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
           [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
           [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]]
```

Část 6: Datové struktury II

V této kapitole budeme pokračovat v práci s datovými strukturami.

Demonstrační příklady:

1. hills – použití zásobníku k sledování nadmořské výšky
2. closure – práce s množinami čísel

Elementární příklady:

1. symmetric – kontrola symetričnosti relace

Přípravy:

1. rpn – vyhodnocení výrazů v postfixovém zápisu
2. b_happy – rozhodování iterativně zadané vlastnosti čísel
3. flood – vyplňování jednobarevné plochy v rastrovém obrázku
4. histogram – statistické zpracování jednorozměrného signálu
5. alchemy – výroba substancí podle sady pravidel
6. stack – kontrola posloupnosti operací se zásobníkem

Rozšířené úlohy:

1. transitive – tranzitivní relace
2. fixpoint – hledání pevného bodu množinové funkce
3. breadth – nejdelší řádek stromu
4. variables – vyhodnocení výrazu zadaného slovníkem
5. connected † – spojitost sítě MHD
6. lakes † – jezírka v krajině

6.1: Programovací jazyk

Tato kapitola přidává několik odvozených operací na seznamech a množinách. Pozor, tyto operace mají **lineární** složitost.

6.1.1 Výrazy Z minulé kapitoly známe operace:

`hodnota in množina`
`klíč in slovník`

Nyní přidáme analogické dotazy tohoto tvaru na přítomnost hodnoty v seznamu: `hodnota in seznam` (zde `seznam` je opět podvýraz), ale musíme si pamatovat, že pro **seznam** tento dotaz **není efektivní**: obsahuje skrytou iteraci potenciálně všemi prvky seznamu.

Pro **množiny** připouštíme nově tyto tvary výrazů (kde $množina_1$ a $množina_2$ jsou vždy **podvýrazy**, které se musí vyhodnotit na hodnoty typu množina):

- $množina_1$ `|` $množina_2$ se vyhodnotí na **sjednocení**,
- $množina_1$ `&` $množina_2$ se vyhodnotí na **průnik** a
- $množina_1$ `-` $množina_2$ se vyhodnotí na **rozdíl** příslušných množin.

Konečně pro **seznamy** přidáváme výraz tvaru $seznam_1$ `+` $seznam_2$ (kde $seznam_1$ a $seznam_2$ jsou opět podvýrazy), který se vyhodnotí na **nový seznam** s prvky z prvního i druhého seznamu (nejprve všechny prvky levého operandu, pak všechny prvky pravého, vždy v původním pořadí).

6.1.2 Zabudované podprogramy Objekty typu **množina** získají tyto nové zabudované metody:

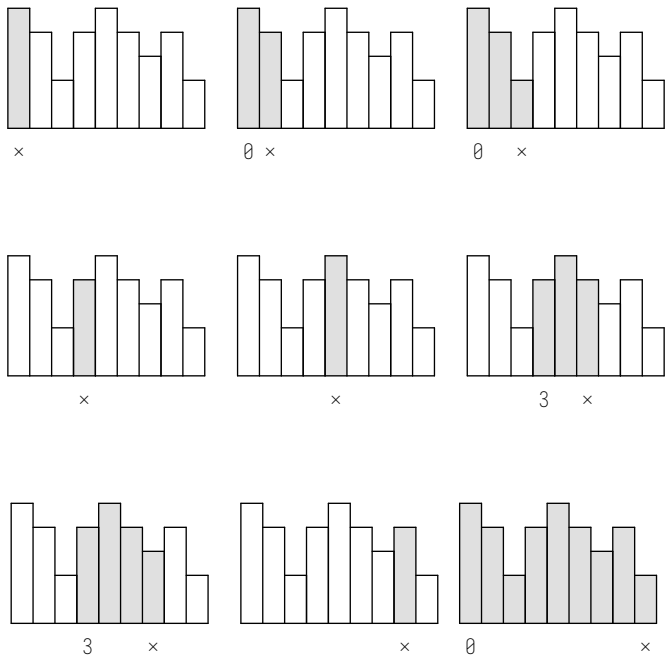
- s_1`.update(`s_2`)` – **přidá** do množiny s_1 všechny prvky, které se **nachází** v s_2 (v s_1 tak bude po provedení operace sjednocení obou množin),[17]
- s_1`.intersection_update(`s_2`)` – **odebere** z množiny s_1 všechny prvky, které se **nenachází** v s_2 (v s_1 tedy bude po provedení průnik),
- s_1`.difference_update(`s_2`)` – **odebere** z množiny s_1 všechny prvky, které se **nachází** v s_2 (v s_1 tedy bude po provedení rozdíl).

Přidáme také několik zabudovaných metod pro práci se seznamy. **Pozor** všechny tyto metody jsou **ekvivalentní iteraci** – nelze tedy jejich použitím ušetřit výpočetní čas, jsou jen syntaktickou zkratkou pro obšírnější `for` cyklus:

- `l.reverse()` – otočí pořadí prvků v seznamu,
- `l.index(v)` – vyhodnotí se na index, na kterém se nachází hodnota `v` (je-li takových více, výsledkem je ten nejmenší; není-li takový žádný, program je ukončen s chybou),
- l_1`.extend(`l_2`)` – přidá na konec seznamu l_1 všechny prvky ze seznamu l_2 (ve stejném pořadí),
- `l.insert(i, v)` – vloží **před** index `i` hodnotu `v` (tedy hodnoty na indexech $j \geq i$ přesune o jednu pozici doprava a na index `i` uloží hodnotu `v`),
- `l.pop(i)` – odstraní hodnotu z indexu `i` (a tedy všechny hodnoty na vyšších indexech přesune o jednu pozici doleva).

6.d: Demonstrace (ukázky)

6.d.1 [hills] Uvažme následovný problém: na vstupu máme výškový profil trasy, a zajímá nás, jak dlouho jsme se pohybovali ve výšce aspoň takové, v jaké jsme teď. Zajímavé hodnoty budeme samozřejmě dostávat pouze na sestupu. Například (aktuální pozici budeme značit symbolem × a odpovídající úsek vyšší nadmořské výšky vybarvíme):



Definujeme tedy čistou funkci `hills`, která dostane na vstupu seznam výšek (celých čísel) a které výsledkem bude stejně dlouhý seznam indexů, které odpovídají vždy prvnímu vybarvenému sloupci v ilustraci výše.

`def hills(heights: list[int]) -> list[int]:`

V proměnné `stack` budeme udržovat zásobník, který bude obsahovat indexy všech předchozích vrcholů, které jsou nižší než ten aktuální. Do proměnné `indices` budeme počítat výsledný seznam indexů.

```
    stack: list[int] = []
    indices: list[int] = []
    for i in range(len(heights)):
        while len(stack) > 0 and heights[stack[-1]] >= heights[i]:
            stack.pop()
        if len(stack) == 0:
            indices.append(0)
        else:
            indices.append(stack[-1] + 1)
        stack.append(i)
    return indices
```

[17] Pozor, s_1`.update(`s_2`)` **není** totéž, jako s_1 `=` s_1 `|` s_2 – první operace vnitřně změní existující hodnotu s_1, ta druhá vytvoří **novou množinu** a výsledek sváže se jménem s_1.

Funkčnost ověříme na několika příkladech (seznam example odpovídá obrázku výše).

```
def main() -> None:  # demo
    assert hills([1, 2, 3]) == [0, 1, 2]
    assert hills([3, 2, 1]) == [0, 0, 0]
    assert hills([1, 2, 1]) == [0, 1, 0]
    assert hills([2, 2, 2]) == [0, 0, 0]
    assert hills([1, 2, 3, 2]) == [0, 1, 2, 1]
    assert hills([1, 3, 2, 3]) == [0, 1, 1, 3]
    assert hills([3, 1, 3, 2]) == [0, 0, 2, 2]
    example = [4, 3, 1, 3, 4, 3, 2, 3, 1]
    assert hills(example) == [0, 0, 0, 3, 4, 3, 3, 7, 0]
```

6.d.3 [closure] V této ukázce se budeme zabývat datovým typem **množina**. Stejně jako u seznamů, slovníků a podobně se jedná o složený typ, který má prvky. Množina má některé vlastnosti společné jak se seznamem – obsahuje pouze prvky, ale nikoliv klíče, tak se slovníkem – podobně jako klíče ve slovníku, hodnoty v množině můžou být přítomny nejvýše jednou. Od seznamu se liší mimo jiné tím, že množinu nelze indexovat (pouze iterovat).

Krom omezení na výskyt každého prvku nejvýše jednou poskytuje množina **efektivní** test na přítomnost prvku (podobně, jako slovník poskytuje efektivní test na přítomnost klíče). Chceme-li zjistit, objevuje-li se nějaká hodnota v běžném seznamu, strávíme tím čas, který je přímo úměrný počtu prvků tohoto seznamu. Naopak v množině lze očekávat, že čas potřebný pro zjištění přítomnosti na počtu prvků v množině vůbec nezávisí: trvá přibližně stejně dlouho nalézt prvek v množině o deseti prvcích i v množině o deseti milionech prvků (takto to funguje v Pythonu – tato operace má očekávanou **konstantní** složitost; některé jiné jazyky poskytují datový typ množina, kde čas potřebný k zjištění přítomnosti prvku závisí na tom, kolik **řádů** má číslo popisující její velikost – mluvíme pak o tzv. **logaritmické** složitosti).

Uvažme zobrazení $f : A \times A \to A$ kde $A \subseteq \mathbb{Z}$ a f je zadané tabulkou (slovníkem, kde klíč je dvojice čísel a hodnota je číslo – rozmyslete si, že takový slovník skutečně reprezentuje tabulku, budou-li ve slovníku přítomny všechny potřebné dvojice). Například logickou spojku and lze podobnou tabulkou reprezentovat takto (budeme-li reprezentovat True číslem 1 a False číslem 0):

	0	1
0	0	0
1	0	1

Jako slovník bychom stejnou tabulku zapsali takto:

```
{(0, 0): 0, (0, 1): 0,
 (1, 0): 0, (1, 1): 1}.
```

Zobrazení f budeme říkat **operace** a budeme jej popisovat následujícím typem:

```
Operation = dict[tuple[int, int], int]
```

Na vstupu tedy dostaneme tabulku, která reprezentuje f a množinu čísel $B \subseteq A$. Naším úkolem bude nalézt nejmenší množinu čísel C takovou, že:

- $B \subseteq C$, tedy C obsahuje všechny zadané prvky,
- pro každé $(x, y) \in C \times C$ platí $f(x, y) \in C$ – říkáme, že množina C je **uzavřena** na operaci f.

```
def closure(set_b: set[int], operation_f: Operation) -> set[int]:
```

Jak budeme postupovat? Množinu C budeme budovat postupně: začneme tím, že do C vložíme všechny prvky z B:

```
    set_c = set_b.copy()
```

Dále budeme procházet všechny dvojice ze součinu $C \times C$, a nalezneme-li takovou, že její obraz ještě v množině C není, přidáme jej tam. Toto ale nemůžeme udělat přímo: přidat prvek do množiny, kterou právě iterujeme, je zakázáno (protože by bylo těžké zaručit, aby byla iterace konzistentní – tzn. aby se nestalo, že v iteraci uvidíme některé, ale ne všechny, nové prvky).

Proto si napíšeme pomocnou funkci find_missing, která najde chybějící prvky a vrátí je jako množinu. Stojíme před dvěma problémy: po přidání nových prvků musíme celou proceduru opakovat, protože vznikly nové dvojice. Tento problém vyřešíme tak, že budeme funkci find_missing volat opakovaně, tak dlouho, dokud bude nalézat nové prvky.

Druhý problém je, že tento postup není příliš efektivní: rádi bychom se vyhnuli procházení dvojic, které jsme již kontrolovali. To sice samozřejmě lze, ale značně by nám to zkomplikovalo kód, proto tentokrát ušetříme práci sobě (a nějakou tím přiděláme počítači).

```
    to_add = find_missing(set_c, operation_f)

    while len(to_add) != 0:
        set_c.update(to_add)
        to_add = find_missing(set_c, operation_f)

    return set_c
```

Pomocná (čistá) funkce find_missing je velmi jednoduchá: projde všechny dvojice z $C \times C$ (tedy součinu množiny set_c se sebou samou), a zobrazí-li se tato dvojice na prvek, který v set_c zatím není, přidá ho do své návratové hodnoty.

```
def find_missing(set_c: set[int], operation_f: Operation) \
        -> set[int]:
    result: set[int] = set()

    for x in set_c:
        for y in set_c:
            to_add = operation_f[(x, y)]
            if to_add not in set_c:
                result.add(to_add)

    return result
```

Zbývá otestovat, že funkce closure se chová, jak čekáme.

```
def main() -> None:  # demo
    op_and = {(0, 0): 0, (0, 1): 0, (1, 0): 0, (1, 1): 1}
    op_xor = {(0, 0): 0, (1, 0): 1, (0, 1): 1, (1, 1): 0}
    set_false = set([0])
    set_true = set([1])
    set_both = set([0, 1])

    assert closure(set_false, op_and) == set_false
    assert closure(set_true, op_and) == set_true
    assert closure(set_both, op_and) == set_both
    assert closure(set_false, op_xor) == set_false
    assert closure(set_true, op_xor) == set_both

    add_mod4 = {(0, 0): 0, (0, 1): 1, (0, 2): 2, (0, 3): 3,
                (1, 0): 1, (1, 1): 2, (1, 2): 3, (1, 3): 0,
                (2, 0): 2, (2, 1): 3, (2, 2): 0, (2, 3): 1,
                (3, 0): 3, (3, 1): 0, (3, 2): 1, (3, 3): 2}

    assert closure(set([0]), add_mod4) == set([0])
    assert closure(set([1]), add_mod4) == set([0, 1, 2, 3])
    assert closure(set([2]), add_mod4) == set([0, 2])
    assert closure(set([3]), add_mod4) == set([0, 1, 2, 3])
    assert closure(set([0, 2]), add_mod4) == set([0, 2])
```

6.e: Elementární příklady

6.e.1 [symmetric] Jak jistě víte, binární relací nad danou množinou A je každá množina dvojic prvků z množiny A, tzn. relace nad A je podmnožina kartézského součinu $A \times A$. Daná relace se pak nazývá symetrická, platí-li pro všechny dvojice (a, b) z této relace, že se v relaci zároveň nachází i dvojice (b, a). V této úloze budeme pracovat s relacemi nad celými čísly.

Napište predikát, kterého hodnota bude True dostane-li v parametru symetrickou relaci, False jinak.

```
def is_symmetric(relation: set[tuple[int, int]]) -> bool:
    pass
```

6.p: Přípravy

6.p.1 **[rpn]** Napište (čistou) funkci, která na vstupu dostane:

- neprázdný výraz `expr` složený z proměnných a z aritmetických operátorů, zapsaný v postfixové notaci, a
- slovník, přiřazující proměnným číselnou hodnotu (můžete se spolehnout, že všechny proměnné použité v daném výrazu jsou v tomto slovníku obsaženy),

a vrátí číslo, na které se daný výraz vyhodnotí. Každý operátor nebo proměnná je samostatný řetězec, celý výraz je pak tvořen posloupností těchto řetězců. Povolené operátory jsou pouze `+` a `*`.

Postfixová notace funguje následovným způsobem:

- výraz čteme zleva doprava, přitom si každou hodnotu zapíšeme,
- narazíme-li na operátor, např. `+`:
  - v hlavě sečteme poslední dvě hodnoty které jsme napsali,
  - tyto hodnoty smažeme,
  - zapíšeme místo nich součet, který jsme si zapamatovali.

Tento postup opakujeme, až dokud nepřečteme celý výraz. Je-li výraz správně utvořený, na konci tohoto procesu máme zapsané jediné číslo. Toto číslo je výsledkem vyhodnocení zadaného výrazu.

```
def rpn_eval(expr: list[str], variables: dict[str, int]) -> int:
    pass
```

6.p.2 **[b_happy]** Dané přirozené číslo je **b-šťastné** platí-li, že nahradíme-li jej součtem druhých mocnin jeho cifer, vyjádřených v poziční soustavě se základem `b`, a tento postup budeme dále opakovat na takto vzniklém čísle, po konečném počtu kroků dostaneme číslo 1.

Například číslo 3 je 4-šťastné, protože:

- $3 = (3)_4$
- $3^2 = 9 = (21)_4$
- $2^2 + 1^2 = 5 = (11)_4$
- $1^2 + 1^2 = 2 = (2)_4$
- $2^2 = 4 = (10)_4$
- $1^2 + 0^2 = 1$.

Číslo 2 není 5-šťastné:

- $2 = (2)_5$
- $2^2 = 4 = (4)_5$
- $4^2 = 16 = (31)_5$
- $3^2 + 1^2 = 10 = (20)_5$
- $2^2 + 0^2 = 4$

a protože se nám ve výpočtu číslo 4 zopakovalo, nemůžeme již dojít k výsledku 1.

Napište predikát, který o číslu `number` rozhodne, je-li `base`-šťastné.

```
def is_b_happy(number: int, base: int) -> bool:
    pass
```

6.p.3 **[flood]** **Flood fill** je algoritmus z oblasti rastrové grafiky, který vyplní souvislou jednobarevnou plochu novou barvou. Postupuje tak, že nejdříve na novou barvu obarví pozici, na které začíná, dále se pokusí obarvit její sousedy (pozice jiné než cílové barvy se neobarvují), a podobně pokračuje se sousedy těchto sousedů, atd. Zastaví se, dojde-li na okraj obrázku, nebo narazí na pixel, který nemá žádné nové stejnobarevné sousedy.

Sousední pixely uvažujeme pouze ve čtyřech směrech, tj. ne diagonálně.

Napište proceduru, která na vstupu dostane plochu reprezentovanou obdélníkovým seznamem seznamů (délky všech vnitřních seznamů jsou stejné), počáteční pozici (je zaručeno, že se bude jednat o platné souřadnice), a cílovou barvu, na kterou mají být vybrané pozice přebarveny.

```
Position = tuple[int, int]
Area = list[list[int]]

def flood_fill(area: Area, start: Position, colour: int) -> None:
    pass
```

6.p.4 **[histogram]** Napište (čistou) funkci, která na vstupu dostane signál `data` reprezentovaný seznamem celočíselných amplitud (vzorků). Výsledkem bude statistika tohoto signálu, kterou vytvoří následujícím způsobem:

1. funkce signál nejdříve očistí od všech vzorků s amplitudou větší než `max_amplitude` a menších než `min_amplitude`,
2. následně jej převzorkuje tak, že sloučí každých `bucket` vzorků (poslední vzorek může být nekompletní) do jednoho vypočtením jejich průměru a jeho následným zaokrouhlením (pomocí vestavěné funkce `round`),
3. nakonec spočítá, kolikrát se v upraveném signálu objevují jednotlivé amplitudy, a vrátí slovník, kde klíč bude amplituda a hodnota bude počet jejích výskytů.

```
def histogram(data: list[int], max_amplitude: int,
              min_amplitude: int, bucket: int) -> dict[int, int]:
    pass
```

6.p.5 **[alchemy]** V této úloze budete zjišťovat, je-li možné pomocí alchymie vyrobit požadovanou substanci. Vstupem je:

- množina substancí, které již máte k dispozici (máte-li už nějakou substanci, máte ji k dispozici v neomezeném množství),
- slovník, který určuje, jak lze existující substance transmutovat: klíčem je substance kterou můžeme vytvořit a hodnotou je seznam „vstupních“ substancí, které k výrobě potřebujeme,
- cílová substance, kterou se pokoušíme vyrobit.

Napište predikát, kterého hodnota bude `True`, lze-li z daných substancí podle daných pravidel vytvořit substanci požadovanou, `False` jinak.

```
def is_creatable(owned_substances: set[str],
                 rules: dict[str, set[str]], wanted: str) -> bool:
    pass
```

6.p.6 **[stack]** Čistá funkce `valid_stack_ops` dostane na vstupu dva seznamy `pushed`, `popped` a rozhodne, jestli tyto seznamy mohly být výsledkem posloupnosti operací **push** a **pop** nad zásobníkem, který je na začátku prázdný. (Seznam `pushed` má odpovídat pořadí, v němž byly prvky vkládány operací **push**; seznam `popped` pořadí, v němž byly prvky odebírány operací **pop**.) Předpokládejte, že se ani v jednom vstupním seznamu neopakují stejné prvky.

Příklady:

Pro vstup `([1, 2, 3, 4, 5], [4, 5, 3, 2, 1])` má být výsledkem `True`, protože existuje posloupnost operací **push 1**, **push 2**, **push 3**, **push 4**, **pop** (vrátí 4), **push 5**, **pop** (vrátí 5), **pop** (vrátí 3), **pop** (vrátí 2), **pop** (vrátí 1).

Pro vstup `([1, 2, 3, 4, 5], [4, 3, 5, 1, 2])` má být výsledkem `False`, protože neexistuje žádná posloupnost operací **push** a **pop**, která by odpovídala těmto seznamům.

```
def valid_stack_ops(pushed: list[int], popped: list[int]) -> bool:
    pass
```

6.r: Řešené úlohy

6.r.2 **[fixpoint]** Mějme funkci `f`, která pro dané celé číslo `a` vrátí množinu obsahující `a`, `a // 2` a `a // 7`. Použitím této funkce na množině pak míníme její použití na každém prvku dané množiny a následné sjednocení všech obdržených výsledků.

Napište (čistou) funkci, která na množinu ze svého argumentu použije `f`, dále použije `f` na obdržený výsledek a takto bude pokračovat až dojde do bodu, kdy se dalším použitím `f` daná množina už nezmění. Výsledkem bude počet aplikací `f` na množinu, které bylo potřeba provést, než se proces zastavil.

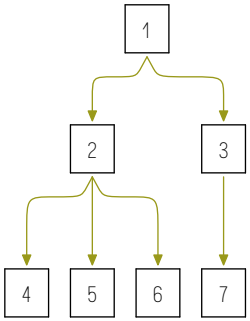
Například z množiny `{1, 5, 6}` vznikne první aplikací popsané funkce množina `{0, 1, 2, 3, 5, 6}`:

- hodnota `1` se zobrazila na `{1, 1 // 2 = 0, 1 // 7 = 0}`,
- hodnota `5` na `{5, 5 // 2 = 2, 5 // 7 = 0}`, a konečně
- hodnota `6` na `{6, 6 // 2 = 3, 6 // 7 = 0}`.

Po další aplikaci se už množina nijak nezmění, proto je výsledkem číslo jedna.

```
def fixpoint(starting_set: set[int]) -> int:
    pass
```

6.r.3 [breadth] Uvažujme neprázdný strom s očíslovanými vrcholy (kořen má vždy číslo 1), např.:



Tento strom zakódujeme do slovníku takto:

```
Tree = dict[int, list[int]]

def example_tree() -> Tree:
    return {1: [2, 3],
            2: [4, 5, 6],
            3: [7],
            4: [], 5: [], 6: [], 7: []}
```

Tedy klíče jsou čísla vrcholů a hodnoty jsou seznamy jejich (přímých) potomků. Napište čistou funkci, která najde „nejdelší řádek“ v obrázku takovéhoto stromu a vrátí jeho délku. Řádek je vždy tvořen uzly, které mají stejnou vzdálenost od kořene.

Pomůcka: máte-li uložený nějaký řádek v seznamu, lehce získáte řádek následující (o jedna vzdálenější od kořene). Pak už stačí nalézt nejdelší takový seznam.

```
def breadth(tree: Tree) -> int:
    pass
```

6.r.4 [variables] Uvažujme jednoduché aritmetické výrazy se sčítáním a násobením. Budeme je ukládat do dvojice slovníků (expr a const), a to následovně:

- klíč je vždy jméno proměnné (řetězec),
- hodnota ve slovníku expr je trojice:
  - první složka je operátor '*' nebo '+',
  - druhá a třetí složka jsou operandy – názvy proměnných,
- hodnota ve slovníku const je číslo.

Každá proměnná se objeví v nejvýše jednom slovníku. Proměnné, které se nenachází v žádném z nich jsou rovny nule.

Napište čistou funkci, která dostane jako parametry slovníky expr a const a název proměnné. Výsledkem bude hodnota této proměnné. Při vyhodnocování se Vám bude hodit zásobník a pomocný slovník.

```
def evaluate(expr: dict[str, tuple[str, str, str]],
             const: dict[str, int], var: str) -> int:
    pass
```

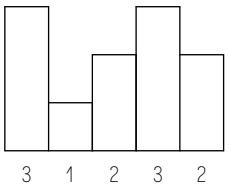
6.r.5 [connected] † Uvažme městskou hromadnou dopravu, která má pojmenované zastávky, mezi kterými jezdí (pro nás anonymní) spoje. Spoje mají daný směr: není zaručeno, že jede-li spoj z A do B, jede i spoj z B do A. Dopravní síť budeme reprezentovat slovníkem, kde klíčem je nějaká zastávka A, a jemu příslušnou hodnotou je seznam zastávek, do kterých se lze z A dopravit bez dalšího zastavení.

Napište predikát, který rozhodne, je-li možné dostat se z libovolné zastávky na libovolnou jinou zastávku pouze použitím spojů ze zadaného slovníku.

```
def all_connected(stops: dict[str, list[str]]) -> bool:
    pass
```

6.r.6 [lakes] † Napište (čistou) funkci, která na vstupu dostane průřez krajiny a spočte, kolik vody se v dané krajině udrží, bude-li na ni neomezeně pršet. Krajina je reprezentována sekvencí celých nezáporných čísel, kde každé reprezentuje výšku jednoho úseku. Všechny úseky jsou stejně široké a mimo popsaný úsek krajiny je všude výška 0.

Například krajina [3, 1, 2, 3, 2] dokáže udržet 3 jednotky vody (mezi prvním a čtvrtým segmentem):




```
def lakes(land: list[int]) -> int:
    pass
```

6.v: Volitelné úlohy

6.v.2 [robot] Představte si robota, který se umí pohybovat dopředu a dozadu a otáčet se o 90° v obou směrech. Pozici robota reprezentujeme dvojicí celých čísel; první souřadnice je x-ová (záporná čísla jsou na západ od počátku, kladná na východ), druhá souřadnice je y-ová (záporná čísla jsou na sever, kladná na jih).

Čistá funkce simulate_robot dostane seznam instrukcí pro robota, vykoná je a vrátí finální pozici robota. Na začátku je robot na souřadnicích (0, 0) a je otočen směrem k severu. Jednotlivé instrukce jsou dvojice v tomto formátu:

- ("rotate", n) – robot se otočí o $n \cdot 90°$ doprava (pro záporná n doleva);
- ("forward", n) – robot se posune o n kroků dopředu;
- ("backward", n) – robot se posune o n kroků dozadu;
- ("undo", n) – robot zruší efekt posledních n provedených instrukcí.

U příkazů jiných než rotate je n vždy nezáporné celé číslo. Instrukce undo může být použita vícekrát a je tak možno rušit efekt více instrukcí, např. posloupnost instrukcí forward 3, backward 7, undo 1, undo 1 způsobí, že robot bude stát na své počáteční pozici. Smíte předpokládat, že k instrukci undo n nedojde ve chvíli, kdy zbývá méně než n předchozích instrukcí. Zejména tedy undo 1 nemůže stát na začátku souboru (ale undo 0 ano).

```
def simulate_robot(instructions: list[tuple[str, int]]) \
        -> tuple[int, int]:
    pass
```

6.v.3 [frogbot] Představte si robotickou žábu, která umí skákat rovně dopředu o zadanou celočíselnou délku a otáčet se o 90° v obou směrech.

Čistá funkce simulate_frogbot dostane seznam instrukcí pro robožábu, vykoná je a vrátí počet různých pozic, na kterých se žába během vykonávání instrukcí nacházela (včetně počáteční a poslední pozice). Pozor na to, že na některou pozici se v průběhu vykonávání instrukcí může žába dostat vícekrát – tuto pozici pořád započítáváme jen jednou.

Jednotlivé instrukce jsou dvojice v tomto formátu:

- ("rotate", n) – robožába se otočí o $n \cdot 90°$ (kladný úhel doprava, záporný doleva);
- ("jump", n) – robožába poskočí o n jednotek dopředu.

Zde n může být libovolné kladné celé číslo (funkce musí bez problémů fungovat i pro obrovská čísla).

Poznámka: Všimněte si, že na počáteční pozici ani natočení žáby odpověď

vůbec nezáleží.

```
def simulate_frogbot(instructions: list[tuple[str, int]]) -> int:
    pass
```

Část 7: Vlastní datové typy, třídy

Ukázky:

1. shapes – složené datové typy
2. hospital – jednoduché objekty
3. stack – zásobník jako zřetězený seznam

Elementární příklady:

1. warriors – složené datové typy
2. sorted – variace na zřetězený seznam
3. maximum – hledání v zřetězeném seznamu

Přípravy:

1. duration – datová struktura pro práci s časem
2. tortoise – želví grafika bez grafiky
3. filter – výběr ze zřetězeného seznamu dle kritéria
4. ring – kruhový buffer pevné velikosti
5. hash – hashovací tabulka pomocí zřetězených seznamů
6. doubly † – obousměrně zřetězený seznam

Rozšířené úlohy:

1. circular – seznam zřetězený do kruhu
2. shuffle – přeuspořádání zřetězeného seznamu
3. books – jednoduchá databáze knih
4. select – výběr ze zřetězeného seznamu dle indexů
5. zipper † – seznam s posuvným význačným prvkem
6. poly † – reprezentace polynomů

7.1: Programovací jazyk

Tato kapitola přináší možnost definovat vlastní (uživatelské) datové typy. K tomuto účelu zavedeme nový typ **definice**. Definice datového typu musí stát vně jakékoliv jiné definice (tedy na stejné úrovni jako definice funkcí, které jsme doteď znali).

Definice typu má následovný tvar:

```
class Třída:
    def __init__(self, param₁: typ₁, …, paramₙ: typₙ) -> None:
        tělo
    def metoda₁(self, param₁: typ₁, …, paramₙ: typₙ) -> typ:
        tělo
    …
```

Uvnitř **definice typu** se tedy může objevit definice **inicializační funkce** a **definice metod** (a nic jiného). Tyto definice se v obou případech velmi podobají na definice funkcí – základním rozdílem (krom toho, kde stojí) je povinný první parametr s názvem self.

7.1.1 Vytváření hodnot V případě inicializační funkce (povinně nazvané __init__) reprezentuje parametr self nový objekt, který je potřeba inicializovat (zejména nastavit počáteční hodnoty atributů).

Nové **hodnoty** uživatelského typu Třída se vytvoří následovným **výrazem**:

Třída(výraz₁, …, výrazₙ)

Protože se jedná o výraz, lze jej použít jako podvýraz v jiných výrazech, nebo třeba v přiřazovacím příkazu na pravé straně takto:

objekt = Třída(výraz₁, …, výrazₙ)

Tento výraz krom samotného vytvoření objektu zavolá inicializační funkce __init__, s následovnými vazbami formálních parametrů:

- self se váže na **nově vznikající objekt**,
- param_1 se váže na hodnotu výrazu výraz_1, atd.,
- param_n se váže na hodnotu výrazu výraz_n.

7.1.2 Atributy Hlavním úkolem inicializační funkce je nastavit počáteční hodnoty **atributů** nového objektu. Atributy se velmi podobají proměnným, nejsou ale svázané s aktuálně vykonávanou funkcí, ale s objektem. Přístup k atributům objektu je **výraz**, který se podobá na použití metody. Např.:

```
person.weight
bmi = person.weight / person.height ** 2
d = sqrt(point.x ** 2 + point.y ** 2)
```

Objekty mají určitou podobnost s n-ticemi, které již dobře známe: sdružují několik hodnot (potenciálně různých typů) do jedné. Mají ale i dvě zásadní odlišnosti:

- atributy objektů jsou **pojmenované** (jsou určeny jmény, nikoliv pořadím),
- objekty mají **vnitřní přiřazení** – vazbu atributu na hodnotu lze měnit (použitím přiřazovacího příkazu).

Přiřazení do atributu je příkaz, který se podobá na ostatní druhy přiřazení, které známe (zejména na vnitřní přiřazení do seznamu nebo slovníku):

objekt.atribut = výraz

kde objekt a atribut jsou **jména**. Významem je změna vazby atributu (na hodnotu, která vznikne vyhodnocením výrazu výraz).

7.1.3 Metody V metodách parametr self reprezentuje objekt, na kterém byla metoda použita. Tedy při použití metody (druh **výrazu**, který již známe u zabudovaných typů):

objekt.metoda₁(výraz₁, …, výrazₙ)
se vážou formální parametry na skutečné parametry takto:

- self se váže na hodnotu objekt,
- param_1 se váže na hodnotu výrazu výraz_1, atd.,
- param_n se váže na hodnotu výrazu výraz_n.

Jinak jsou metody stejné jako obyčejné funkce.

7.d: Demonstrace (ukázky)

7.d.1 [shapes] V této ukázce demonstrujeme základní použití složených datových typů. Srovnejte 05/shapes.py – budeme nyní řešit stejné problémy, ale místo n-tic (kde jsou jednotlivé složky číslované ale jinak anonymní) budeme používat složené typy, které mají jednotlivé složky pojmenované.

```
from math import isclose, pi, sqrt, cos, sin
```

Jako první si definujeme typ pro kruh (anglicky disc), který má jediný atribut, totiž poloměr typu float.

```
class Disc:
    def __init__(self, radius: float) -> None:
        self.radius = radius
```

Dále definujeme čistou funkci disc_area, která má jediný parametr typu Disc a jejíž výsledkem je číslo typu float.

```
def disc_area(disc: Disc) -> float:
    return pi * disc.radius ** 2
```

Dalším typem bude obdélník, Rectangle, který má atributy dva, šířku a výšku.

```
class Rectangle:
    def __init__(self, width: float, height: float) -> None:
        self.width = width
        self.height = height
```

Podobně jako u kruhu, definujeme čistou funkci pro výpočet plochy:

```
def rectangle_area(rect: Rectangle) -> float:
    return rect.width * rect.height
```

Elipsa reprezentuje podobný případ, kdy potřebujeme k jejímu popisu dvě čísla, tentokrát délky jejích dvou poloos. Všimněte si, že na rozdíl od reprezentace v ukázce 05/shapes.py (kde jsme používali n-tice) nám tu záměna elipsy a obdélníku v žádném případě nehrozí.

```
class Ellipse:
```

```
    def __init__(self, major: float, minor: float) -> None:
        assert major >= minor
        self.major = major
        self.minor = minor

def ellipse_area(ellipse: Ellipse) -> float:
    return pi * ellipse.major * ellipse.minor
```

Atributy složeného typu samozřejmě nemusí být všechny stejného typu (jako tomu bylo v této ukázce dosud). Zadefinujeme si tedy ještě pravidelný n-úhelník, který zadáme hlavním poloměrem (tzn. vzdáleností vrcholu od středu) a počtem vrcholů (který je na rozdíl od poloměru celočíselný).

```
class Polygon:
    def __init__(self, radius: float, vertices: int) -> None:
        self.radius = radius
        self.vertices = vertices

def polygon_area(polygon: Polygon) -> float:
    half_angle = pi / polygon.vertices
    half_side = sin(half_angle) * polygon.radius
    minor_radius = cos(half_angle) * polygon.radius
    return polygon.vertices * minor_radius * half_side
```

Dále napíšeme funkci, která ze seznamu obdélníků vybere ten s největší plochou, existuje-li takový právě jeden. Je zde vidět, že se složenými typy pracujeme velmi obdobně jako s těmi zabudovanými. Tím, že používáme pouze abstraktní operace (které jsou „schované“ do funkcí) je dokonce tělo oproti implementaci z ukázky 05/shapes.py zcela nezměněné.

```
def largest_rectangle(rectangles: list[Rectangle]) \
        -> Rectangle | None:

    if len(rectangles) == 0:
        return None

    largest = rectangles[0]
    count = 0

    for r in rectangles:
        if isclose(rectangle_area(r), rectangle_area(largest)):
            count += 1
        elif rectangle_area(r) > rectangle_area(largest):
            count = 1
            largest = r

    return largest if count == 1 else None
```

Nyní zbývá pouze popsané funkce otestovat:

```
def main() -> None:  # demo
    unit_rectangle = Rectangle(1, 1)
    assert isclose(rectangle_area(Rectangle(2, 2)), 4)
    assert isclose(rectangle_area(unit_rectangle), 1)
    assert isclose(polygon_area(Polygon(sqrt(2), 4)), 4)
    assert isclose(polygon_area(Polygon(1, 6)), 2.5980762113533)
    assert isclose(ellipse_area(Ellipse(1, 1)), 3.1415926535898)
    assert isclose(ellipse_area(Ellipse(6, 2)), 37.699111843078)
    assert isclose(ellipse_area(Ellipse(12.532, 8.4444)),
                   332.4597362298)
```

Jak již bylo naznačeno, problém, který se nám objevil s elipsou a obdélníkem před dvěma týdny nás už nyní nemusí trápit. Odkomentujete-li následovné tvrzení, mypy Vám v programu ohlásí chybu.

```
    pass  # assert ellipse_area(unit_rectangle) == 1

    assert largest_rectangle([]) is None
    r_11 = Rectangle(1, 1)
    r_43 = Rectangle(4, 3)
    r_55 = Rectangle(5, 5)
    r_62 = Rectangle(6, 2)
    r_c2 = Rectangle(12, 2)
    r_xy = Rectangle(10.2, 1.5)
    assert largest_rectangle([r_11, r_43, r_62]) is None
    assert largest_rectangle([r_55, r_43, r_11]) == r_55
    assert largest_rectangle([r_c2, r_xy]) == r_c2
```

7.d.2 [hospital] V této ukázce se budeme zabývat jednoduchými objekty, které můžeme chápat jako rozšíření složených typů o **metody**. Metoda je podprogram, který je svázán se svým složeným typem (objektem): metoda má vždy parametr, který reprezentuje instanci objektu se kterou bude pracovat. V Pythonu tento parametr explicitně uvádíme v hlavičce metody (tzn. v seznamu formálních parametrů), a to vždy jako první a vždy se jménem self.

Při **volání** metod používáme tečkovou notaci, stejně jako u zabudovaných typů: máme-li hodnotu items typu list, můžeme napsat třeba items.append(1) a víme, že toto volání provede nějakou akci nad hodnotou items. Naše metody se budou chovat stejně (ve skutečnosti je totiž append metoda třídy list).

Máme-li hodnotu hospital typu Hospital (u objektů také mluvíme o instanci hospital třídy Hospital), můžeme napsat třeba hospital.add_doctor('dept', doc). Metodě definované jako add_doctor(self, department, doctor) bude hodnota hospital předána právě parametrem self, hodnoty uvedené při volání v závorkách pak v dalších parametrech. Přesněji:

- metoda add_doctor má 3 formální parametry,
- uvažujeme volání hospital.add_doctor('dept', doc).

Parametry se předají takto:

- hodnota hospital bude předána prvním parametrem (self),
- druhý parametr, department, bude mít hodnotu 'dept',
- třetí parametr, doctor, bude mít hodnotu doc.

Třída Doctor je obyčejný složený typ bez metod, jaké známe z předchozí ukázky. Bude mít atributy name (jméno lékaře) a night_shift (lze-li tomuto lékaři plánovat noční směny).

```
class Doctor:
    def __init__(self, name: str, night_shift: bool) -> None:
        self.name = name
        self.night_shift = night_shift
```

Třída Hospital reprezentuje samotnou nemocnici. Nemocnice má lékaře a oddělení, na kterých jednotliví lékaři pracují. Data budeme ukládat do slovníku, ve kterém jako klíče použijeme názvy jednotlivých oddělení a hodnoty budou seznamy lékařů.

```
class Hospital:
```

Inicializační funkce __init__ inicializuje novou nemocnici. Krom objektu, který bude inicializovat (parametr self) jí předáme seznam názvů oddělení (parametr departments). Metoda inicializuje atribut departments.

```
    def __init__(self, departments: list[str]) -> None:
        self.departments: dict[str, list[Doctor]] = {}
        for name in departments:
            self.departments[name] = []
```

Metoda add_doctor zařadí lékaře doctor na oddělení department. Vstupní podmínkou je, že toto oddělení v nemocnici existuje.

```
    def add_doctor(self, department: str, doctor: Doctor) -> None:
        self.departments[department].append(doctor)
```

Protože krom zvláštního zápisu volání je metoda podprogram jako každý jiný, lze metody stejně tak klasifikovat na čisté funkce, predikáty a podobně. Není ale obvyklé mluvit v tomto kontextu o procedurách: metody velmi často mění předaný objekt (parametr self) – na rozdíl od funkcí budeme tedy předpokládat, není-li uvedeno jinak, že metoda mění objekt self.

Budeme nicméně nadále explicitně uvádět, má-li mít metoda nějaké **jiné** vedlejší efekty. Není-li tedy uvedeno jinak, metoda může měnit **pouze** objekt předaný parametrem self. Metoda, která je označená jako **čistá** (a tedy i metoda, která je označená jako **predikát**) **nemění** ani tento.

Metoda (predikát) night_coverage zkontroluje, že je na každém oddělení aspoň jeden lékař, který může být zařazen na noční směnu.

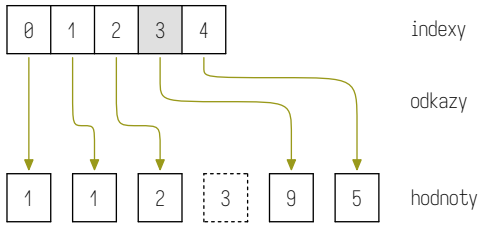
```
    def night_coverage(self) -> bool:
```

```
    for department, doctor_list in self.departments.items():
        found = False
        for doctor in doctor_list:
            if doctor.night_shift:
                found = True
                break

        if not found:
            return False
    return True
```

7.d.3 [stack] V této ukázce se zaměříme na datové struktury. Jednoduše zřetězený seznam jste již viděli v přednášce, zde si ukážeme velice jednoduchou obměnu téhož. Seznamy tohoto typu nejsou sice v praxi až tak oblíbené (s možnou výjimkou Linuxového jádra, kde se používají často) ale velmi dobře ilustrují klíčové znalosti práce s pamětí. Proto je velmi důležité, abyste jim rozuměli.

Zřetězený seznam je složený z **uzlů**. Každý uzel je samostatná hodnota uložená v paměti (to, kde přesně je uložená a jak se o tom rozhodne, nás prozatím nebude příliš zajímat, stejně jako jsme to dosud neřešili u jiných typů hodnot). Každý uzel si bude pamatovat jedno z čísel, které bylo do seznamu uloženo. Co je ale mnohem zajímavější je, že si zároveň bude pamatovat svého následovníka: další uzel v seznamu.

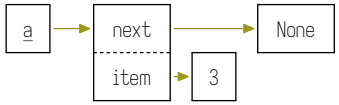
Zde je na místě připomenout, jak v Pythonu fungují proměnné, konkrétně **atributy** složených typů. Ze třetí kapitoly si jistě pamatujete, že zabudovaný typ list přiřazuje (váže) hodnoty k jednotlivým indexům. Má navíc tzv. **vnitřní přiřazení**: vazbu indexu a hodnoty lze změnit. Vnitřní přiřazení zapisujeme třeba items[3] = 9, jeho efekt jsme si ukazovali na obrázku, který si zde připomeneme:



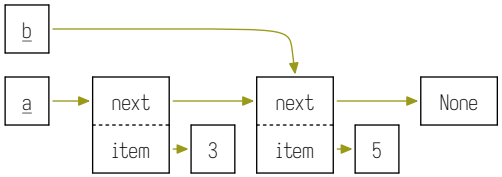
Složené typy mají stejný koncept vnitřního přiřazení, místo (proměnného) počtu indexů mají ale (pevnou) množinu jmen. Zadefinujme si složený typ Node, kterým budeme reprezentovat jednotlivé uzly zřetězeného seznamu:

```
class Node:
    def __init__(self, item: int) -> None:
        self.next: Node | None = None
        self.item = item
```

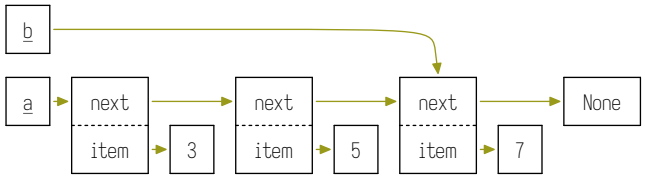
Vytvoříme-li novou hodnotu typu Node, například voláním a = Node(3), bude výsledek vypadat takto:



Vytvořme nyní novou hodnotu, b = Node(5) a použijme vnitřní přiřazení a.next = b. Výsledek bude:



Pro jistotu vytvoříme ještě jeden uzel, tentokrát dvojicí příkazů b.next = Node(7) a b = b.next. Výsledná situace bude vypadat takto:



Na tomto posledním obrázku je také vidět, že k uzlu s hodnotou 5 již sice nemáme přímý přístup (není přímo uložen v žádné proměnné), dostaneme se k němu ale skrz atribut next uzlu a.

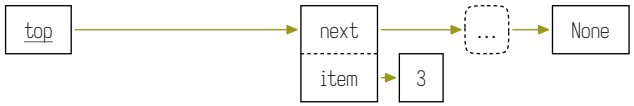
Nyní již můžeme přistoupit k implementaci samotného zásobníku. Tento bude mít pouze 2 metody, push a pop. Metoda push vloží novou hodnotu na vrchol zásobníku. Pro tuto hodnotu vytvoří nový uzel a přidá ho na začátek seznamu. Metoda pop naopak uzel odstraní a hodnotu v něm uloženou vrátí. Je-li seznam prázdný, vrátí None.

class Stack:

Inicializační funkce __init__ inicializuje prázdný zásobník. Vrchol zásobníku bude uzel (hodnota typu Node), je-li zásobník neprázdný, jinak bude None.

```
    def __init__(self) -> None:
        self.top: Node | None = None
```

Následuje metoda push. Ta vytvoří nový uzel a nastaví jeho následníka na stávající vrchol (ať už je to uzel nebo None). Parametrem metody push je hodnota, kterou chceme do zásobníku vložit. Uvažme následující situaci před voláním stack.push(7):



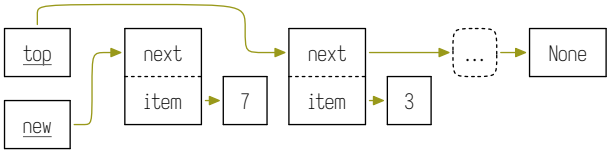

```
    def push(self, item: int) -> None:
```

Metoda push má pouze tři příkazy. Proto si na ní detailně ilustrujeme, jak se bude vnitřní struktura (tvořená zejména atributy next jednotlivých uzlů) postupně měnit.

Atribut top prozatím obsahuje uzel, který byl doteď (tzn. těsně před voláním metody push) vrcholem zásobníku. První příkaz vytvoří nový uzel (voláním Node(item)) a přiřadí jej do lokální proměnné new.

```
        new = Node(item)
```

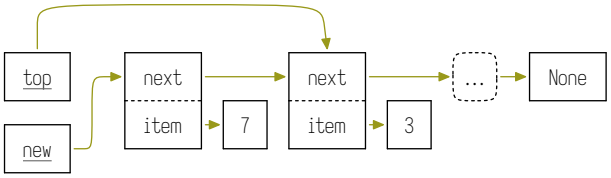
Tento uzel zatím není nijak svázaný se zbytkem seznamu:



V dalším kroku provážeme uzel new se zbytkem seznamu. Atribut top ovšem stále odkazuje předchozí vrchol zásobníku.

```
        new.next = self.top
```

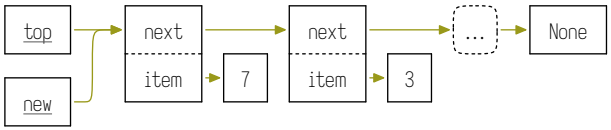
Nová situace:



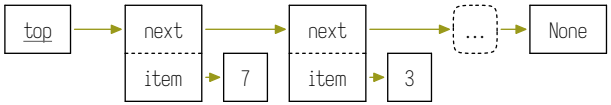
V posledním krok změníme odkaz (atribut) top tak, aby ukazoval na nový vrchol.

```
        self.top = new
```

Atribut top a lokální proměnná new tak sdílí tutéž hodnotu:



Návratem z metody push lokální proměnná new zanikne, a atribut top zůstane jediným odkazem na (teď již nový) vrchol zásobníku. K předchozímu vrcholu se dostaneme skrz atribut next nového vrcholu:



Druhou metodou je pop, která odstraní prvek (a odpovídající uzel) ze zásobníku. V obecném případě můžeme samozřejmě metodu pop volat v libovolném stavu zásobníku. Pro ilustraci ale předpokládejme, že byla zavolána těsně po ukončení výše vyobrazeného push(7).

```
def pop(self) -> int | None:
```

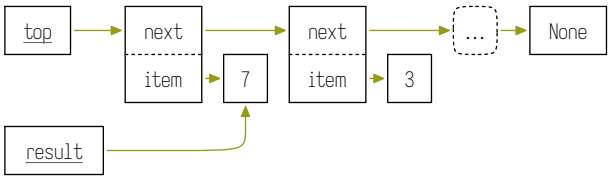
Nejprve vyřešíme případ, kdy byl zásobník prázdný. To poznáme tak, že atribut top je nastavený na None. V takovém případě stav nijak neměníme, a pouze vrátíme None, čím indikujeme volajícímu, že nebylo ze zásobníku co odstranit.

```
    if self.top is None:
        return None
```

Na tomto místě již víme, že zásobník je neprázdný, a tedy atribut top obsahuje nějaký vrchol. Nejprve si poznačíme hodnotu, která je v tomto uzlu uložena:

```
    result = self.top.item
```

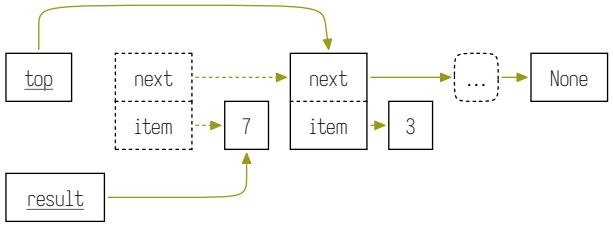
Po vykonání tohoto příkazu bude lokální proměnná result sdílet hodnotu s atributem top.item:



Dále přesměrujeme atribut top na nový vrchol. Uvědomte si, že je-li stav zásobníku X, po provedení dvojice operací push a pop se tento vrátí do stejného stavu X. Zejména bude mít tentýž vrchol jako před provedením obou operací.

```
    self.top = self.top.next
```

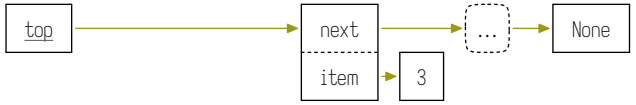
Srovnejte následující situaci se situací vyobrazenou před voláním push výše.



Všimněte si také, že na původní vrchol zásobníku již neexistuje žádný odkaz (není uložen v žádné proměnné ani atributu). V jazyce Python taková hodnota automaticky zanikne. Zbývá už jen vrátit požadovanou hodnotu:

```
    return result
```

Po provedení dvojice volání push a pop se tedy dostaneme do původního stavu. Ještě jednou zdůrazňujeme, že volání push a pop nemusí být takto provázána vždy. Lze třeba volat vícekrát za sebou push, nebo pop. Na vyobrazených situacích to ve skutečnosti nic nemění, s výjimkou konkrétních čísel uložených v zásobníku.



7.d.4 [fifo] V této ukázce budeme implementovat (tentokrát neomezenou) frontu pomocí zřetězeného seznamu. Třída Node bude sloužit jako jeden uzel fronty:

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None
```

Třída Queue bude implementovat běžné rozhraní fronty (push, pop) a data bude ukládat do jednoho spojitého řetězu uzlů (instancí třídy Node).

Hlavu tohoto řetězu (tzn. takový uzel, z kterého lze dojít do všech ostatních uzlů) uložíme do atributu chain. Řetěz bude mít právě tolik prvků, kolik jich je uloženo ve frontě a bude ukončen uzlem, který má next nastavený na None. Výjimku tvoří případ, kdy je fronta prázdná, kdy není hodnota chain vůbec určena.

```
class Queue:
    def __init__(self) -> None:
        self.chain: Node | None = None
        self.insert: Node | None = None

    def push(self, value: int) -> None:
        if self.insert is None:
            self.chain = self.insert = Node(value)
        else:
            self.insert.next = Node(value)
            self.insert = self.insert.next

    def pop(self) -> int | None:
        if self.chain is None:
            return None

        value = self.chain.value
        self.chain = self.chain.next
        if self.chain is None:
            self.insert = None
        return value
```

Všimněte si, že správně implementovaná fronta při žádné operaci **neprochází** zřetězený seznam, kterým je reprezentovaná. V přiložených testech si demonstrujeme zejména to, že fronta bude funkční i v situaci, kdy ji uměle uprostřed „rozpojíme“ – samozřejmě jen do chvíle, než by se takové rozpojení dostalo do hlavy fronty.

```
def main() -> None:  # demo
    queue = Queue()
    queue.push(1)
    check_count(queue, 1)
    check_value(queue.pop(), 1)
    assert queue.pop() is None
    queue.push(3)
    queue.push(5)
    queue.push(7)
    check_count(queue, 3)
    assert queue.chain is not None
    assert queue.chain.value == 3

    assert queue.chain is not None
    broken = queue.chain.next
    assert broken is not None
    lost = broken.next
    assert lost is not None
    broken.next = None

    queue.push(8)
```

```
    queue.push(9)
    check_value(queue.pop(), 3)
    broken.next = lost
    check_value(queue.pop(), 5)
    check_value(queue.pop(), 7)
    check_value(queue.pop(), 8)
    check_value(queue.pop(), 9)
    assert queue.pop() is None

def check_count(queue: Queue, count: int) -> None:
    node = queue.chain
    while node:
        node = node.next
        count -= 1
    assert count == 0

def check_value(value: int | None, expect: int) -> None:
    assert value is not None
    assert value == expect
```

7.e: Elementární příklady

7.e.1 [warriors] Třída Warrior reprezentuje válečníka, který má jméno a sílu. Tyto jeho vlastnosti bude třída reprezentovat atributy name a strength. Tato třída obsahuje pouze inicializační funkci __init__.

```
class Warrior:
    def __init__(self, name: str, strength: int) -> None:
        self.name = name
        self.strength = strength
```

Velké množství válečníků tvoří hordu, kterou reprezentujeme třídou Horde. Horda má interní strukturu – je rozdělena do pojmenovaných klanů, které reprezentujeme slovníkem (jméno klanu, seznam válečníků).

```
class Horde:
```

Vytvoří hordu se zadanými klany.

```
    def __init__(self, clans: dict[str, list[Warrior]]) -> None:
        pass
```

Metoda vrátí aktuální stav hordy, t.j. slovník všech klanů.

```
    def clans(self) -> dict[str, list[Warrior]]:
        pass
```

Metoda přidá válečníka do klanu. Neexistuje-li klan daného jména, metoda jej vytvoří.

```
    def add_warrior(self, clan: str, warrior: Warrior) -> None:
        pass
```

Metoda (a zároveň predikát) zkontroluje, má-li každý klan dostatečnou sílu, která je rovna součtu sil všech jeho válečníků. Měl by vám stačit nanejvýš jeden průchod seznamy válečníků.

```
    def validate_clan_strength(self, required: int) -> bool:
        pass
```

7.e.2 [sorted] V této úloze budete implementovat jednoduchý zřetězený seznam s dodatečnou vlastností, že jeho prvky jsou vždy vzestupně seřazené.

Třída Node reprezentuje jeden uzel seznamu, a má dva atributy: hodnotu typu int a odkaz na další uzel next. Tuto třídu nijak nemodifikujte.

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None
```

Následující třída reprezentuje seřazený, zřetězený seznam. Implementujte naznačené metody insert a get_greatest_in.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

```
class SortedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Metoda insert vloží do seznamu nový prvek. Nezapomeňte, že seznam musí být vždy seřazený. Metoda by měla projít celý seznam nejvíce jednou.

```
    def insert(self, value: int) -> None:
        pass
```

Následující metoda vrátí největší prvek seznamu, jehož hodnoty spadají do oboustranně uzavřeného intervalu [value, value + dist]. Pokud žádný takový prvek není, vrátí None. V případech, kdy se tomu lze vyhnout, neprocházejte seznam zbytečně celý.

```
    def get_greatest_in(self, value: int, dist: int) -> int | None:
        pass
```

7.e.3 [maximum]

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Napište čistou funkci, která najde největší hodnotu uloženou ve vstupním zřetězeném seznamu, případně None je-li vstupní seznam prázdný.

```
def maximum(num_list: LinkedList) -> int | None:
    pass
```

7.p: Přípravy

7.p.1 [duration] Naprogramujte třídu TimeInterval, která bude reprezentovat časový interval. Vstupní podmínkou inicializační funkce je, že všechny parametry jsou nezáporná čísla a minuty a sekundy jsou nejvýše 59.

```
class TimeInterval:
    def __init__(self, hours: int, minutes: int, seconds: int) -> None:
        pass
```

Metoda zkrátí interval o čas reprezentovaný parametrem interval.

```
    def shorten(self, interval: 'TimeInterval') -> None:
        pass
```

Metoda prodlouží interval o čas reprezentovaný parametrem interval.

```
    def extend(self, interval: 'TimeInterval') -> None:
        pass
```

Metoda vrátí reprezentovaný interval jako n-tici ve formátu (hodiny, minuty, sekundy), kde minuty a sekundy nabývají hodnoty z uzavřeného intervalu [0, 59].

```
    def format(self) -> tuple[int, int, int]:
        pass
```

7.p.2 [tortoise] V této úloze budete programovat třídu Tortoise, která se chová podobně jako želva, kterou jsme používali v kapitole B. Rozdílem bude, že naše želva nebude kreslit na obrazovku, ale pouze počítat své aktuální souřadnice. Souřadnice želvy jsou po každém kroku celočíselné, ale výpočty provádějte na hodnotách typu float, které po každém kroku zaokrouhlíte zabudovanou funkcí round.

Všechny kreslící metody želvy budou vracet odkaz na vlastní instanci, aby bylo lze volání pohodlně řetězit (viz použití v testech).

```
Point = tuple[int, int]

class Tortoise:
```

Želva je po vytvoření otočena v kladném směru osy y t.j. „na sever“ a nachází se v bodě initial_point.

```
    def __init__(self, initial_point: Point) -> None:
        pass
```

Metoda forward posune želvu vpřed o vzdálenost distance.

```
    def forward(self, distance: int) -> 'Tortoise':
        pass
```

Metoda backward ji posune naopak vzad, opět o vzdálenost distance.

```
    def backward(self, distance: int) -> 'Tortoise':
        pass
```

Metody left a right želvu otočí o počet stupňů daný parametrem angle. Metoda left proti, a metoda right po směru hodinových ručiček.

```
    def left(self, angle: int) -> 'Tortoise':
        pass

    def right(self, angle: int) -> 'Tortoise':
        pass
```

Konečně (čistá) metoda position vrátí aktuální pozici želvy.

```
    def position(self) -> Point:
        pass
```

7.p.3 [filter] Třídy Node a LinkedList pro reprezentaci zřetězeného seznamu máte již připraveny. Nijak je nemodifikujte.

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Napište čistou funkci filter_linked, která vytvoří nový zřetězený seznam, který vznikne z toho vstupního (num_list) vynecháním všech uzlů s hodnotou menší než lower_bound. Měl by Vám stačit jeden průchod vstupním seznamem.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

```
def filter_linked(lower_bound: int,
                  num_list: LinkedList) -> LinkedList:
    pass
```

7.p.4 [ring] Naprogramujte třídu RingBuffer která se bude chovat jako fronta, ale bude mít shora omezenou velikost. Pro ukládání dat bude využívat jinou třídu, SimpleList (tuto třídu nesmíte měnit, ani přistupovat k jejím atributům), která poskytuje toto rozhraní (sl je instance SimpleList):

- sl.append(x) vloží na konec seznamu prvek x,
- sl.get(i) vrátí hodnotu na indexu i,
- sl.size() vrátí aktuální velikost seznamu,
- sl.set(i, x) nastaví index i na hodnotu x.

Pozor: V žádné metodě neprocházejte celý seznam.

```
class RingBuffer:
```

Při inicializaci se nastaví velikost kruhové fronty na size. Pro ukládání dat bude použita instance třídy SimpleList předaná parametrem storage.

```
    def __init__(self, size: int, storage: 'SimpleList') -> None:
        pass
```

Metoda push se pokusí přidat prvek na konec fronty. Je-li fronta plná, metoda vrátí False a nic neudělá. V opačném případě prvek vloží na konec fronty a vrátí True.

```
    def push(self, value: int) -> bool:
        pass
```

Metoda pop odstraní prvek ze začátku fronty a vrátí jej. Je-li fronta prázdná, metoda nic neudělá a vrátí None.

```
    def pop(self) -> int | None:
        pass
```

7.p.5 [hash] Hashovací tabulka je datová struktura, která umožňuje rychlé ukládání a vyhledávání hodnot. Základem je hashovací funkce, která určí přihrádku, do níž hodnota patří. V každé přihrádce je pak jednosměrně zřetězený seznam obsahující hodnoty v dané přihrádce.

V našem příkladu budeme používat hashovací funkci modulo, konkrétní hodnota modulu bude stanovena při vytváření hashovací tabulky jako parametr inicializační funkce.

Vaším úkolem bude implementovat třídu HashTable:

- Inicializační funkce __init__ vytvoří seznam (typ list) o m přihrádkách. Každá přihrádka je na začátku tvořená prázdným zřetězeným seznamem.
- Metodu insert, která vloží hodnotu do správné přihrádky. Vstupní podmínkou je, že hodnota v tabulce není přítomna. Tuto metodu implementujte co nejefektivněji.
- Metodu contains, která zjistí, zda se daná hodnota v tabulce vyskytuje, či nikoliv.
- Metodu remove, která zadanou hodnotu z tabulky odebere.
- Metodu bucket, která pro zadaný klíč vrátí hlavu zřetězeného seznamu, který tvoří klíči příslušnou přihrádku (bez ohledu na přítomnost klíče v tabulce), nebo None je-li tato prázdná.

Třídu Node nijak neměňte. Tabulka musí fungovat i v případě, že je seznam vrácený metodou bucket nějak upraven.

```
class Node:
    def __init__(self, key: int) -> None:
        self.key = key
        self.next: Node | None = None

class HashTable:
    def __init__(self, m: int) -> None:
        pass

    def insert(self, key: int) -> None:
        pass

    def contains(self, key: int) -> bool:
        pass

    def remove(self, key: int) -> None:
        pass

    def bucket(self, key: int) -> Node | None:
        pass
```

7.p.6 [doubly] † V této úloze budeme programovat dvojitě zřetězený seznam, který se podobá jednoduše zřetězenému seznamu, který již dobře znáte. Jak napovídá už název, každý uzel bude připojen do řetězu na obě strany, tzn. krom následovníka si bude pamatovat i svého předchůdce.

Oproti seznamu zřetězenému jednoduše se v tom dvojitém lépe odebírají prvky: z libovolného místa seznamu (tedy zejména na obou koncích) lze totiž odebrat prvek bez toho, abychom museli seznam jakkoliv procházet. A proto i Vaše implementace uvedených metod (kromě search) by měla fungovat bez jakéhokoliv procházení seznamu.

```
class Node:
    def __init__(self, init_val: int) -> None:
        self.value = init_val
        self.next: Node | None = None
        self.prev: Node | None = None

class DoubleLinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
        self.tail: Node | None = None
```

Metoda append přidá novou hodnotu na konec seznamu.

```
    def append(self, value: int) -> None:
        pass
```

Metoda prepend naopak vloží novou hodnotu na začátek. Na rozdíl od zabudovaného typu list je toto v principu levná operace.

```
def prepend(self, value: int) -> None:
    pass
```

Metoda remove odstraní ze seznamu libovolný uzel.

```
def remove(self, node: Node) -> None:
    pass
```

Konečně metoda search najde první uzel s danou hodnotu. Když takový uzel neexistuje, vrátí None.

```
def search(self, value: int) -> Node | None:
    pass
```

7.r: Řešené úlohy

7.r.1 [circular] V této úloze naprogramujeme lehce modifikovaný jednosměrně zřetězený seznam (ten standardní znáte z přednášky a z řešeného příkladu sorted_list.py). Rozdíl bude spočívat v tom, že poslední odkaz v seznamu nebude None jako dříve, ale bude ukazovat na hlavu, čím seznam uzavře do kruhu. Třída Node reprezentuje jeden uzel. Zvažte, jakého typu by měl být její atribut next.

```
class Node:
    def __init__(self, value: int) -> None:
        self.next = None
        self.value = value
```

Následuje třída CircularList, která má jediný povinný atribut, head, který ukazuje na hlavu seznamu. V prázdném seznamu by měla být v head uložena hodnota None. Hned po vytvoření reprezentuje instance třídy CircularList právě prázdný seznam. Naznačené metody nechť se chovají následovně:

- insert vloží novou hodnotu na začátek seznamu
- last vrátí poslední **uzel** (nikoliv hodnotu)

Tyto metody nepotřebují nijak procházet seznam hodnot.

Metody split_by_value a split_by_node rozdělí stávající seznam na dva kratší seznamy, a to tak, že uzly od hlavy až k uzlu popsaného parametrem (včetně) ponechá ve stávajícím seznamu, a ze zbytku vytvoří nový seznam, který vrátí. Pořadí uzlů (a tedy i hodnot) musí zůstat zachováno. Metoda split_by_value seznam rozdělí na prvním výskytu zadané hodnoty. Vstupní podmínky:

- hodnota předaná metodě split_by_value musí být v seznamu aspoň jednou přítomna,
- uzel předaný metodě split_by_node patří tomuto seznamu.

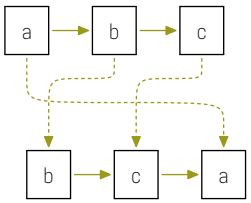
Příklad: uvažme hodnotu lst typu CircularList, která obsahuje prvky 4, 5, 1, 2, 3 a 7. Po provedení příkazu new = lst.split(5) zbudou v seznamu lst pouze hodnoty 4 a 5, zatímco seznam new bude mít prvky 1, 2, 3 a 7.

```
class CircularList:

    def __init__(self) -> None:
        self.head = None

    def insert(self, value: int) -> None:
        pass

    def last(self) -> Node | None:
        pass

    def split_by_value(self, value: int) -> 'CircularList':
        pass

    def split_by_node(self, node: Node) -> 'CircularList':
        pass
```

7.r.2 [shuffle] Na vstupu dostanete (standardní Pythonovský) seznam čísel z rozsahu $\langle 0, n-1 \rangle$ takový, že každé číslo se v něm vyskytuje právě jednou, a který tedy popisuje permutaci. Na každém indexu tohoto seznamu najdete číslo, na které se má daný index permutací zobrazit. Vaším úkolem je ve funkci shuffle tuto permutaci aplikovat na vstupní zřetězený seznam (t.j. upravit odpovídajícím způsobem pořadí jeho uzlů). Předpokládejte, že má právě n uzlů.

Nevytvářejte při řešení nové uzly ani nemodifikujte hodnoty (atribut value) těch existujících. Funkce rovněž nesmí modifikovat vstupní Pythonovský seznam permutation.

Příklad: Je-li zadaná permutace $2, 0, 1$, přesune se prvek z pozice 0 na pozici 2, z pozice 1 na pozici 0 a ten z pozice 2 na pozici 1:



Zadané třídy nijak nemodifikujte. Zamyslete se nad tím, jak to udělat efektivně. Pro správné řešení vám postačují dva přechody vstupním zřetězeným seznamem.

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None

def shuffle(permutation: list[int], linked: LinkedList) -> None:
    pass
```

7.r.3 [books] V této úloze naprogramujeme jednoduchou knihovnu (kolekci knížek). Nejprve implementujte třídu Book s atributy name a author.

```
class Book:
    def __init__(self, name: str, author: str) -> None:
        self.name = name
        self.author = author
```

Dále naprogramujte třídu Bookshelf, která reprezentuje knihovnu, do které lze přidávat knížky a případně je pak odebírat.

```
class Bookshelf:

    def __init__(self, books: list[Book]) -> None:
        pass

    def add_book(self, book: Book) -> None:
        pass
```

Metoda books vrátí seznam knih v pořadí, v jakém byly do knihovny přidány.

```
    def books(self) -> list[Book]:
        pass
```

Metoda group_by_author vrátí slovník, který přiřadí každému autorovi seznam knih, které napsal. K implementaci této metody Vám stačí jeden průchod seznamem knih.

```
    def group_by_author(self) -> dict[str, list[Book]]:
        pass
```

7.r.4 [select]

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Napište čistou funkci, která sestaví zřetězený seznam, který bude obsahovat

hodnoty, které se nachází ve vstupním seznamu na zadaných indexech. Pořadí hodnot zachovejte. Předpokládejte, že indexy v seznamu indices jsou platné a vzestupně seřazené. K implementaci této funkce Vám stačí jeden průchod seznamy indices a linked.

```
def select(indices: list[int], linked: LinkedList) -> LinkedList:
    pass
```

7.r.5 [zipper] † Naprogramujte datovou strukturu 'zipper': jedná se o strukturu podobnou zřetězenému seznamu, s jedním důležitým rozdílem: přesto, že používá jednoduché zřetězení (nikoliv dvojité), lze se v něm efektivně pohybovat oběma směry. Nicméně na rozdíl od dvojitě zřetězeného seznamu nám zipper umožňuje udržovat pouze jediný kurzor.

Jak zipper funguje? Používá následující strukturu:

Jak efektivně kurzor posunout o jednu pozici doleva nebo doprava si pravděpodobně dovedete představit. Pro jednoduchost budeme uvažovat pouze neprázdný zipper.

Pro zajímavost: zipper lze implementovat také pomocí dvojice zásobníků, a tato implementace je typicky efektivnější. V tomto cvičení ale preferujeme použití zřetězených struktur.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

```
class Zipper:
    def __init__(self, num: int) -> None:
        pass
```

Vrátí aktuální hodnotu kurzoru.

```
    def cursor(self) -> int:
        pass
```

Vloží prvek nalevo od kurzoru.

```
    def insert_left(self, num: int) -> None:
        pass
```

Smaže prvek nalevo od kurzoru, existuje-li takový, a vrátí jeho hodnotu. Jinak vrátí None.

```
    def delete_left(self) -> int | None:
        pass
```

Posune kurzor o jednu pozici doleva. Není-li se kam posunout, metoda neudělá nic.

```
    def shift_left(self) -> None:
        pass
```

Posune kurzor o jednu pozici doprava. Není-li se kam posunout, metoda opět neudělá nic.

```
    def shift_right(self) -> None:
        pass
```

7.r.6 [poly] † Polynomy jste již potkali v příkladu r4_poly z páté kapitoly. Připomeňme si, že polynom je výraz tvaru:

$$P(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0 = \sum_0^n a_i x^i$$

Tentokrát budeme polynomy sčítat, odečítat a násobit. Polynom si pro účely tohoto příkladu zavedeme jako datovou strukturu s operacemi popsanými níže. Polynomy se sčítají a násobí dle běžných pravidel – součet $ax^k + bx^k$ se do výsledného polynomu promítne jako $(a+b)x^k$, zatímco výraz $ax^k \cdot bx^l$ povede na člen abx^{k+l}. Nezapomeňte, že při násobení dvou polynomů lze stejnou mocninu x dostat různými způsoby, třeba $x \cdot x^3$ je totéž jako $x^2 \cdot x^2$. Potřebné algoritmy pro výpočet koeficientů výsledného polynomu si jistě již zvládnete z uvedeného odvodit.

```
class Polynomial:
```

Vytvoří nový polynom. Koeficienty ve vstupním seznamu jsou uloženy v pořadí $a_n, a_{n-1}, \ldots, a_1, a_0$ a tento seznam smí obsahovat vedoucí nuly. Vnitřní reprezentaci si ovšem můžete zvolit libovolnou.

```
    def __init__(self, coefs: list[int]) -> None:
        pass
```

Vrátí koeficienty polynomu jako seznam, opět v pořadí $a_n, a_{n-1}, \ldots, a_1, a_0$. Výsledný seznam nesmí obsahovat vedoucí nuly (tzn. pro nenulový polynom platí $a_n \neq 0$).

```
    def get_coefs(self) -> list[int]:
        pass
```

Čistá funkce, které výsledkem je součet vstupních polynomů self + other.

```
    def add(self, other: 'Polynomial') -> 'Polynomial':
        pass
```

Čistá funkce, které výsledkem je rozdíl vstupních polynomů self - other.

```
    def subtract(self, other: 'Polynomial') -> 'Polynomial':
        pass
```

Čistá funkce, které výsledkem je součin vstupních polynomů self * other.

```
    def multiply(self, other: 'Polynomial') -> 'Polynomial':
        pass
```

Demonstrační příklady:

1. countsort – řazení počítáním menších prvků
2. insertsort – řazení zřetězeného seznamu
3. binsearch – hledání půlením intervalu

Elementární příklady:

1. sorted – kontrola seřazenosti seznamu
2. selectsort – řazení výběrem
3. uniqbound – dolní mez v seřazeném seznamu bez opakování

Přípravy:

1. count – počítání frekvence hodnoty v seřazeném seznamu
2. extremes – lokální minima a maxima v seznamu hodnot
3. rotated – kontrola seřazenosti až na rotaci
4. frequency – řazení podle frekvence výskytu
5. merge † – sloučení seřazených zřetězených seznamů
6. unique † – hledání unikátních prvků v seřazeném seznamu

Rozšířené úlohy:

1. bound – hledání v seřazeném seznamu
2. nested – řazení se zachováním struktury
3. flipped – seřazenost až na jedno prohození
4. greater † – permutace cifer
5. heapsort † – řazení haldou
6. radix † – řazení po cifrách

8.1: Programovací jazyk

Tato kapitola přináší pouze dva nové prvky (oba souvisí s řazením).

1. Zabudovanou čistou funkci sorted(x), které výsledkem je nový seznam, který je vzestupně uspořádaný (pro l = sorted(x) a i <= j platí l[i] <= l[j]), a zároveň obsahuje stejné prvky jako x. Parametr x může být:
   - seznam (list),
   - množina (set),
   - d.items(), d.keys() nebo d.values() je-li d hodnota typu slovník (dict).
2. Zabudovanou metodu-proceduru l.sort(), která přeuspořádá seznam l tak, aby byl vzestupně seřazený (samotné prvky se při tom opět nijak nemění).

8.d: Demonstrace (ukázky)

8.d.1 [countsort] V této ukázce se budeme zabývat dvěma velmi jednoduchými řadicími algoritmy založenými na počítání.

První algoritmus funguje pro seznamy, ve kterých se žádná hodnota neopakuje. Pracuje na velmi jednoduchém principu:

- uvažme libovolný prvek R_i vstupního seznamu,
- spočítejme kolik se v seznamu nachází celkem prvků, které jsou menší než R_i; tuto hodnotu označme c_i,
- v seřazeném seznamu se musí R_i objevit na indexu c_i: index (je-li počítán od nuly) je právě počet prvků, které dané hodnotě v seznamu předchází.

Spočítáme-li tedy hodnotu c_i pro každý vstupní prvek, můžeme již přímočaře sestavit výstupní seznam: ke každému prvku známe index, na který ho chceme uložit. Čistá funkce count_sort tuto myšlenku realizuje:

```
def count_sort(records: list[int]) -> list[int]:
```

Protože budeme často iterovat sekvencí indexů seznamu records, uložíme si tuto sekvenci do pomocné proměnné.

```
    indices = [i for i in range(len(records))]
```

Dále si nachystáme dva seznamy: v jednom budeme počítat hodnoty c_i, do toho druhého potom vstupní prvky uložíme vzestupně seřazené.

```
    counts = [0 for _ in indices]
    result = [0 for _ in indices]
```

Hlavní cyklus vypočte do seznamu counts jednotlivé hodnoty c_i. Nejjednodušeji získáme c_i tak, že spočítáme všechna R_j taková, že platí $R_j < R_i$.

Abychom si ale ušetřili práci, uvědomíme si, že není potřeba nejprve při výpočtu c_i vyhodnotit $R_j < R_i$ a později při výpočtu c_j vyhodnotit $R_i < R_j$.

Protože beztak předpokládáme, že se prvky neopakují, platí pro $i \neq j$ právě jedna z těchto dvou možností. Platí-li tedy $R_j < R_i$, můžeme srovnání započítat do c_i (našli jsme prvek menší než R_i) a naopak, platí-li $R_i < R_j$, srovnání rovnou započteme do c_j.

```
    for i in indices:
        for j in range(i):
            if records[j] < records[i]:
                counts[i] += 1
            else:
                counts[j] += 1
```

Zbývá tedy už jen sestavit výsledný seznam. Připomínáme, že hodnota R_i je v programu k dispozici jako records[i] a odpovídající hodnotu c_i máme uloženou v counts[i].

```
    for i in indices:
        result[counts[i]] = records[i]
```

Protože hodnoty se na vstupu neopakují, je v counts uložena permutace indexů seznamu records: máme tedy zaručeno, že zapíšeme na každý index seznamu result, a zároveň, že žádnou hodnotu ze seznamu records neztratíme (nepřepíšeme). To, že výsledný seznam result bude vzestupně seřazený, je pak již zřejmé z předchozího.

```
    return result
```

Druhý algoritmus je v jistém smyslu „opačný“ než ten první: bude pracovat se seznamy, které obsahují pouze hodnoty z předem daného, nepříliš velkého rozsahu $\langle l, h\rangle$. Protože hodnot je málo, budou se v delších seznamech často opakovat. Algoritmus je také velmi jednoduchý:

1. pro každou hodnotu z rozsahu $v \in \langle l, h\rangle$ spočítáme, kolikrát se ve vstupním seznamu nachází; tento počet označíme c_i kde $i = v - l$,
2. s použitím této informace sestavíme výsledný seznam tak, že pro každou hodnotu $v \in \langle l, h\rangle$ do něj vložíme c_i kopií hodnoty v (zde opět $i = v - l$).

Tento algoritmus je realizován čistou funkcí distribution_sort:

```
def distribution_sort(records: list[int], low: int,
                      high: int) -> list[int]:
```

Sekvenci všech hodnot, které se na vstupu mohou objevit si, ve vzestupném pořadí, uložíme do proměnné values. Zároveň si nachystáme seznam counts, ve kterém budeme počítat hodnoty c_i.

```
    values = [i for i in range(low, high)]
    counts = [0 for _ in values]
```

Nyní zjistíme počet výskytů každé hodnoty z values ve vstupním seznamu records:

```
    for record in records:
        counts[record - low] += 1
```

A sestavíme výsledný seznam.

```
    result = []
    for value in values:
        for _ in range(counts[value - low]):
```

```
            result.append(value)

    return result
```

Přestože řadicí algoritmy, které jsme implementovali, jsou velmi jednoduché, není těžké v nich udělat chybu. A to navíc třeba takovou, že se bude projevovat jen vzácně. Proto tyto algoritmy otestujeme obzvlášť důkladně. Funkce test_parameters definovaná níže popisuje parametry seznamů, které budeme testovat: rozsah hodnot (hodnoty budou spadat do rozsahu low <= value < high) a počet prvků. Pro danou sadu parametrů vygenerujeme všechny možné seznamy tak, aby splnily vstupní podmínky (v případě funkce count_sort se hodnoty nesmí opakovat) a ověříme dvě definující vlastnosti řazení:

1. výstup je permutací vstupu,
2. výstup je seřazený.

```
def main() -> None:  # demo

    for low, high, count in test_parameters():
        for records in all_lists(low, high, count, False, []):
            result = count_sort(records)
            assert is_permutation(result, records)
            assert is_sorted(result)

    for low, high, count in test_parameters():
        for records in all_lists(low, high, count, True, []):
            result = distribution_sort(records, low, high)
            assert is_permutation(result, records)
            assert is_sorted(result)

def is_permutation(a: list[int], b: list[int]) -> bool:
    result = [0 for _ in range(max(a + b) + 1)]
    for item in a:
        result[item] += 1
    for item in b:
        result[item] -= 1
    for diff in result:
        if diff != 0:
            return False
    return True

def is_sorted(records: list[int]) -> bool:
    for i in range(len(records) - 1):
        if records[i] > records[i + 1]:
            return False
    return True

def all_lists(low: int, high: int, count: int, repeats: bool,
              prefix: list[int]) -> list[list[int]]:
    if count == 0:
        return [prefix]

    result = []
    for x in range(low, high):
        if repeats or x not in prefix:
            result.extend(all_lists(low, high, count - 1, repeats,
                                    prefix + [x]))
    return result

def test_parameters() -> list[tuple[int, int, int]]:
    result = []
    for high in range(10):
        for low in range(high):
            for count in range(1, 5):
                result.append((low, high, count))
    return result
```

8.d.2 [insertsort] V této ukázce si ukážeme další řadicí algoritmus, tentokrát budeme ale řadit **zřetězené** seznamy, které nelze efektivně indexovat. Jejich výhodou je naopak možnost levně vkládat hodnoty doprostřed: proto si na nich demonstrujeme tzv. **insertion sort**, neboli řazení vkládáním. Myšlenka tohoto algoritmu je také velmi jednoduchá:

1. vytvoříme prázdný **výstupní seznam**,
2. prvky postupně odebíráme ze začátku vstupního seznamu,
3. pro každý odebraný vstupní prvek najdeme ve vznikajícím výstupním seznamu správné místo a tam ho vložíme.

Nejprve si definujeme složený datový typ, kterým budeme reprezentovat zřetězené seznamy:

```
class Node:
    def __init__(self, value: int):
        self.next: Node | None = None
        self.value = value

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Nyní již můžeme přistoupit k samotnému zápisu algoritmu pro řazení vkládáním. Oproti předchozí ukázce bude algoritmus realizovat **procedura**. Nazveme ji insert_sort, a bude přesně kopírovat postup z úvodního odstavce. Abychom zachovali jednoduchou a jasnou strukturu hlavního výpočtu, veškeré pomocné výpočty oddělíme do pomocných procedur.

```
def insert_sort(records: LinkedList) -> None:
    out = LinkedList()
    while records.head is not None:
        to_insert = remove_head(records)
        insert_sorted(out, to_insert)
```

V seznamu out máme nyní seřazený výsledek, naším úkolem ale bylo přeuspořádat stávající seznam records, který je nyní prázdný. Proto do něj „převěsíme“ celý seznam out.

```
    records.head = out.head
```

První pomocná **procedura**, remove_head, oddělí hlavu neprázdného seznamu, a vrátí ji jako samostatný (izolovaný) uzel.

```
def remove_head(lst: LinkedList) -> Node:
    assert lst.head is not None
    result = lst.head
    lst.head = lst.head.next
    result.next = None
    return result
```

Další pomocná **procedura**, insert_sorted, vloží uzel do seřazeného seznamu, a to tak, že výsledný seznam zůstane seřazený (jeho délka se přitom zvýší o jedna). Více explicitně, procedura insert_sorted má tyto vstupní podmínky:

- out je seřazený zřetězený seznam (může být prázdný),
- node je uzel, který není součástí out.

Výstupní podmínkou je:

- out je seřazený seznam a node je součástí out.

Samotné vložení uzlu je jednoduché: „těžká“ část této procedury je nalézt vhodné místo, kam uzel vložit. Tuto část oddělíme do pomocné **čisté funkce**, find_position, která vrátí dvojici uzlů, **mezi které** budeme uzel vkládat. Jeden, nebo i oba vrácené uzly mohou být None.

```
def insert_sorted(out: LinkedList, node: Node) -> None:
    before, after = find_position(out, node.value)
    if before is None:
        out.head = node
    else:
        before.next = node
    node.next = after
```

Zbývá nám definovat poslední, a v podstatě i nejsložitější, podprogram. Na rozdíl od těch předchozích se bude jednat o **čistou funkci**: vstupní seznam nebudeme nijak měnit. Tato funkce má následující vstupní podmínku:

- items je seřazený zřetězený seznam,
- value může ale nemusí být v seznamu přítomna.

Nazveme-li složky návratové hodnoty before a after, výstupní podmínku můžeme popsat takto:

- `before` i `after` jsou `None` a `items` je prázdný, **nebo**
- `before` je `None` a `value <= after.value`, **nebo**
- `after` je `None` a `before.value <= value`, **nebo**
- `before.value <= value <= after.value`.

```
def find_position(items: LinkedList, value: int) \
        -> tuple[Node | None, Node | None]:
    before = None
    after = items.head

    while after is not None and value >= after.value:
        before = after
        after = after.next

    return (before, after)
```

Tím je definice procedury `insert_sort` a jejích pomocných podprogramů hotova. Zbývá nám proceduru otestovat: na to budeme potřebovat další dvě pomocné funkce (obě budou čisté): `to_linked_list` která z klasického Pythonovského seznamu vytvoří seznam zřetězený, a funkce `to_python_list` která provede konverzi opačnou.

```
def to_linked_list(items: list[int]) -> LinkedList:
    out = LinkedList()
    for i in range(len(items) - 1, -1, -1):
        node = Node(items[i])
        node.next = out.head
        out.head = node
    return out

def to_python_list(items: LinkedList) -> list[int]:
    ptr = items.head
    out = []
    while ptr is not None:
        out.append(ptr.value)
        ptr = ptr.next
    return out
```

Stejně jako v předchozí ukázce budeme proceduru `insert_sort` testovat pro všechny seznamy z parametrické rodiny. Přípustné kombinace parametrů nám bude generovat funkce `test_parameters`, jako seznam trojic: nejmenší a největší číslo, které se objeví, a celková délka seznamu.

```
def test_parameters() -> list[tuple[int, int, int]]:
    result = []
    for high in range(10):
        for low in range(high):
            for count in range(1, 5):
                result.append((low, high, count))
    return result
```

Funkce `main` podle parametrů z `test_parameters` vygeneruje všechny odpovídající **seřazené** seznamy, a pro každý seřazený seznam ověří, že procedura `insert_sort` korektně seřadí všechny jeho permutace.

```
def main() -> None:  # demo
    for low, high, count in test_parameters():
        for records in all_lists(low, high, count, True, []):
            linked = to_linked_list(records)
            insert_sort(linked)
            result = to_python_list(linked)
            assert is_sorted(result)
            assert is_permutation(result, records)
```

8.d.3 [binsearch] V poslední ukázce pro tento týden se budeme zabývat **hledáním** v seřazeném seznamu. V krátkých seznamech si můžeme dovolit hledat „naivně“: srovnáme hledanou hodnotu postupně s každým prvkem. Je zřejmé, že v nejhorším případě musíme provést tolik srovnání, kolik prvků je v prohledávaném seznamu.

Je-li ale seznam seřazený, můžeme hledání velmi výrazně urychlit. Technika, kterou k tomu použijeme se jmenuje **půlení intervalu**. Ač to nemusí být na první pohled zřejmé, je velmi důležité, abyste princip této techniky pochopili, protože na ní staví řada fundamentálních výsledků, které budete v dalších semestrech studovat.

Základní myšlenkou algoritmu je rozdělit si vstupní seznam na dvě přibližně stejně dlouhé poloviny. Je-li hodnota v seznamu přítomna, musí se nacházet v jedné z těchto dvou částí. Protože celý seznam je seřazený, platí to i o každém jeho podseznamu, zejména to tedy platí o našich přibližných polovinách.

Je-li nějaký hodnota `value` přítomná v seznamu `list`, musí nutně platit `min(list) <= value <= max(list)`. Je-li `list` navíc vzestupně seřazený, platí `min(list) == list[0]` a `max(list) == list[-1]`. Celkem tedy `list[0] <= value <= list[-1]`.

Protože se jedná o podmínku **nutnou**, není-li splněna, můžeme s jistotou říci, že se hledaná hodnota v daném (pod)seznamu nenachází. Zjistíme-li tedy, že tuto nutnou podmínku některý z našich podseznamů porušuje, nemusíme se tímto nadále vůbec zabývat: stačí nám vyřešit problém pouze pro zbývající polovinu.

```
def bin_search(records: list[int], value: int) -> bool:
```

Zbývá nám vyřešit konkrétní zápis této myšlenky. Zejména se chceme vyhnout vytváření nových seznamů: tato operace je drahá, a ve výsledku bychom pak oproti naivnímu hledání nic neušetřili. Můžeme si ale pamatovat **rozsah indexů** ve kterém aktuálně hledáme. Indexy si nazveme l (`low`) a h (`high`), a budeme je chápat jako **polouzavřený** interval $\langle l, h)$: index l (`low`) do rozsahu patří, index h (`high`) už nikoliv. Zejména to znamená, že interval je prázdný právě když `low == high`.

Na začátku výpočtu prohledáváme celý seznam, proměnné `low` a `high` tedy nastavíme na příslušné hodnoty:

```
    low, high = 0, len(records)
```

Hledání pokračuje dokud je prohledávaný (pod)seznam neprázdný. Najdeme-li hledanou hodnotu, cyklus ukončíme dříve: skončí-li tedy cyklus pro nesplnění podmínky, hledaná hodnota v seznamu nebyla přítomna (za předpokladu, že hodnota v seznamu byla přítomna, musí být přítomna v prázdném seznamu → spor).

```
    while low < high:
```

Stávající seznam si rozdělíme na ony avizované „přibližně stejně velké“ části (jejich délka se může lišit o jedničku, byl-li seznam liché délky). Dělení provedeme na indexu m (`mid`). První podseznam je tedy $\langle l, m)$. Ten druhý by pak měl být $\langle m, h)$, nicméně je praktičtější použít $\langle m + 1, h)$.

Proč jsme vypustili samotné m (`mid`)? Jedná se o právě jeden prvek, se kterým se bude tedy dobře pracovat (nemusíme si hlídat existenci). Navíc nám jeho vyloučení z dalšího hledání zaručuje, že se prohledávaný seznam v každé iteraci zkrátí aspoň o jedničku. Nehrozí nám tak, že se program „zacyklí“ na nějakém okrajovém případu, který jsme neošetřili.

```
        mid = low + (high - low) // 2
```

Jako první ověříme, zda na indexu m není uložena hledaná hodnota: pokud ano, hledání ukončíme. V opačném případě víme, že index m můžeme z dalších úvah vypustit.

Navíc musíme zdůvodnit, proč musí nutně index m v některé iteraci ukazovat na hledanou hodnotu, byla-li v seznamu přítomna. Uvědomme si, že struktura algoritmu je taková, že je-li prvek přítomen, je nutně přítomen v rozsahu $\langle l, h)$. Zároveň se v každé iteraci interval striktně zmenšuje, a m vždy leží v tomto intervalu.

Konečně nejmenší neprázdný interval vede na $m = l = h - 1$, jediný prvek v tomto intervalu je tudíž na indexu m, a hledanou hodnotu tedy zaručeně najdeme nejpozději ve chvíli, kdy $l = h - 1$.

```
        if records[mid] == value:
            return True
```

Dále tedy zkontrolujeme podseznam $\langle l, m)$: je-li `value` v této části seznamu, platí již zmiňovaná nutná podmínka: `records[low] <= value <= records[mid]`, zejména pak její druhá část: `value <= records[mid]`.

Tuto znegujeme na `records[mid] < value`: platí-li tato negace, nutná podmínka je porušena a hodnota `value` se v této části seznamu nenachází. Proto prohledávaný interval zúžíme na $\langle m + 1, h)$ a pokračujeme další iterací.

```
        if records[mid] < value:
            low = mid + 1
```

Zbývá provést analogickou kontrolu pro rozsah $\langle m, h)$. Můžeme-li přítomnost value v této části vyloučit, budeme se v další iteraci zabývat už pouze podseznamem $\langle l, m)$.

```
        if records[mid] > value:
            high = mid
```

Jak již bylo zmíněno dříve, dojde-li k ukončení cyklu proto, že nám k prohledání zbyl prázdný podseznam, víme, že hledaný prvek v seznamu nebyl přítomen. Vrátíme tedy False.

```
    return False
```

Tím je implementace hotova. Podobně jako u řadicích algoritmů budeme hledání půlením intervalu testovat velmi pečlivě: nejprve vygenerujeme každý seřazený seznam v daném rozsahu parametrů. Pro každý z nich pak ověříme, že výsledek hledání je správný, a to jak pro hodnoty, které jsou v seznamu přítomny, tak i hodnoty, které v něm nejsou (buď chybí, nebo jsou mimo rozsah hodnot).

```
def main() -> None:  # demo
    for low, high, count in test_parameters():
        for records in sorted_lists(low, high, count, []):
            for v in range(low - 1, high + 1):
                assert bin_search(records, v) == (v in records)

def sorted_lists(low: int, high: int, count: int,
                 prefix: list[int]) -> list[list[int]]:
    if count == 0:
        return [prefix]

    result = []
    for x in range(low, high):
        result.extend(sorted_lists(x, high, count - 1, prefix + [x]))
    return result

def test_parameters() -> list[tuple[int, int, int]]:
    result = []
    for high in range(10):
        for low in range(high):
            for count in range(0, 8):
                result.append((low, high, count))
    return result
```

8.e: Elementární příklady

8.e.1 [sorted] Napište predikát is_sorted, který rozhodne, je-li vstupní seznam vzestupně seřazený. Existuje řešení, jehož složitost je lineární.

```
def is_sorted(num_list: list[int]) -> bool:
    pass
```

8.e.2 [selectsort] Naprogramujte algoritmus řazení výběrem (formou procedury, která upraví vstupní seznam).

```
def selectsort(num_list: list[int]) -> None:
    pass
```

8.e.3 [uniqbound] Napište čistou funkci, která najde v zadaném uspořádaném seznamu numbers největší číslo, které není větší než parametr value. Neexistuje-li takové, vraťte None.

V ostatních případech je tedy výsledkem vždy číslo, které se nachází v numbers a vždy platí lower_bound(numbers, x) ≤ x.

Předpokládejte, že v seznamu numbers se čísla neopakují. Očekávaná složitost řešení je logaritmická.

```
def lower_bound(numbers: list[int], value: int) -> int | None:
    pass
```

8.p: Přípravy

8.p.1 [count] Implementujte čistou funkci count_in_sorted, která ve vzestupně seřazeném seznamu records co nejefektivněji spočte počet výskytů hodnoty value. K hodnotám v records přistupujte použitím metody get: např. records.get(7) vrátí hodnotu na indexu 7. Délku seznamu získáte voláním records.size(). Dobré řešení úlohy je logaritmické časové složitosti.

```
def count_in_sorted(records: 'CountingList', value: int) -> int:
    pass
```

8.p.2 [extremes] Napište čistou funkci local_extremes, která dostane na vstupu seznam values čísel a vrátí dvojici seznamů min_indices, max_indices. Každý prvek seznamu values je unikátní. Seznam min_indices (max_indices) bude obsahovat indexy lokálních minim (maxim) seznamu values. Oba tyto seznamy budou vzestupně seřazené. Řešení očekáváme v lineární časové složitosti.

```
Minima = list[int]
Maxima = list[int]

def local_extremes(values: list[int]) -> tuple[Minima, Maxima]:
    pass
```

8.p.3 [rotated] Implementujte predikát is_cyclically_sorted, který je pravdivý, je-li seznam cyklicky seřazený. Seznam je cyklicky seřazený, existuje-li rotace, po které bude seřazený vzestupně. Měli byste být schopni napsat řešení, jehož složitost je lineární.

```
def is_cyclically_sorted(records: list[int]) -> bool:
    pass
```

8.p.4 [frequency] Implementujte čistou funkci frequency_sort, která podle frekvencí výskytu seřadí hodnoty v seznamu values. Hodnoty se stejnou frekvencí výskytu nechť jsou seřazeny vzestupně podle hodnoty samotné. Výsledný seznam bude obsahovat každou hodnotu právě jednou.

```
def frequency_sort(values: list[int]) -> list[int]:
    pass
```

8.p.5 [merge] † Třída LinkedList reprezentuje zřetězený seznam, se kterým budete pracovat. Uzly tohoto seznamu mají atribut next, ke kterému můžete libovolně přistupovat a měnit ho a metodu compare, která srovná hodnoty uložené ve dvou uzlech. K samotným hodnotám přímo přistupovat nesmíte. Volání a.compare(b) vrátí (-1, 0, 1) je-li hodnota v uzlu a (menší, stejná, větší) než hodnota v uzlu b. První uzel je uložen v atributu head. Třídy LinkedList a Node nijak nemodifikujte.

```
class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Napište funkci merge, která spojí 2 vzestupně seřazené zřetězené seznamy do jediného seřazeného seznamu. Funkce nevytváří nové uzly, pouze přepojuje ukazatele next stávajících uzlů z obou seznamů. Seznamy lze spojit v lineárním čase.

```
def merge(left: LinkedList, right: LinkedList) -> LinkedList:
    pass
```

8.p.6 [unique] † Implementujte co nejefektivněji čistou funkci unique, která vrátí seznam unikátních prvků ze vzestupně seřazeného seznamu values. Vstupní seznam je reprezentován třídou, která poskytuje pouze metody get(i) (vrátí i-tý prvek) a size (vrátí počet prvků). Výsledný seznam je běžný seznam typu list a bude také vzestupně seřazený. Funkci je možné napsat efektivněji než s lineární složitostí.

```
def unique(values: 'CountingList') -> list[int]:
    pass
```

8.r: Řešené úlohy

8.r.1 [bound] Implementujte funkci left_bound, která ve vzestupně seřazeném seznamu values co nejefektivněji najde index prvního výskytu hodnoty target. Pokud se hodnota v seznamu nenachází, vrátí None. V této úloze je lineární řešení neefektivní.

```
def left_bound(values: list[int], target: int) -> int | None:
    pass
```

8.r.2 [nested] Implementujte čistou funkci sort_nested, která vzestupně uspořádá prvky v seznamu seznamů čísel lists, a to tak, že přeuspořádá jenom čísla ve vnitřních seznamech, aniž by měnil jejich délku. Výstupní

seznam bude tedy obsahovat stejný počet stejně dlouhých seznamů jako ten vstupní, ale v obecném případě budou tyto vnořené seznamy obsahovat jiná čísla.

```
def sort_nested(lists: list[list[int]]) -> list[list[int]]:
    pass
```

8.r.3 [flipped] Implementujte predikát is_almost_sorted, který je pravdivý, je-li v seznamu items potřeba prohodit právě jednu dvojici různých čísel, aby se stal vzestupně seřazeným. Existuje řešení, jehož časová složitost je lineární.

```
def is_almost_sorted(items: list[int]) -> bool:
    pass
```

8.r.4 [greater] † Napište funkci next_greater, která vrátí nejmenší větší číslo se stejnými ciframi jaké má číslo number. Pokud větší číslo neexistuje, funkce vrací None. Nezkoušejte všechny permutace cifer, existuje efektivnější řešení.

```
def next_greater(number: int) -> int | None:
    pass
```

8.r.5 [heapsort] † Implementujte algoritmus řazení haldou. Základní myšlenka algoritmu je podobná algoritmu řazení výběrem:

- vstupní seznam rozdělíme na dvě pomyslné části, neseřazenou (na začátku) a seřazenou (na konci);
- v každé iteraci najdeme největší prvek v neseřazené části, přesuneme ho na její konec, a pomyslnou dělící čáru posuneme o jeden prvek doleva.

To, čím se algoritmus od řazení výběrem liší je metoda „hledání“ onoho největšího prvku. Seznam totiž před samotným začátkem řazení přeuspořádáme do formy tzv. haldy, která má tyto vlastnosti:

- největší prvek je na nulté pozici,
- pro prvek na pozici i platí, že je větší než prvky na pozicích $2i + 1$ a $2i + 2$ (existují-li).

Je zřejmé, že nahrazením největšího prvku tuto vlastnost můžeme lehce pokazit. Klíčové pozorování je, že její obnovení je snadné (a zejména rychlé). Začneme od indexu `i = 0` a opakovaně (tak dlouho, dokud index `i` ukazuje dovnitř neuspořádané části pole):

- vybereme index největšího prvku z možností `i`, `2*i + 1` nebo `2*i + 2` – pokud jsme vybrali `i`, jsme hotovi;
- v opačném případě vyměníme vybraný prvek s tím na indexu `i` a `i` nastavíme na index vybraný v předchozím kroku.

Mělo by být vidět, že za předpokladu, že před výměnou největšího prvku měl seznam vlastnosti haldy, uvedenou procedurou je opět získá (její obvyklý název je sift_down). Zbývá tedy zajistit, aby mělo vstupní pole tyto vlastnosti i před samotným začátkem řazení.

Toho dosáhneme například tak, že budeme opakovaně spouštět proceduru sift_down s počáteční hodnotou `i` nastavenou postupně na hodnoty $n/2, n/2-1, \ldots, 0$ kde n je délka vstupního seznamu. Proč tato procedura funguje se dozvíte například v článku „Heapsort“ v anglické wikipedii.

```
def heapsort(records: list[int]) -> None:
    pass
```

```
def test_parameters() -> list[tuple[int, int, int]]:
    result = []
    for high in range(10):
        for low in range(high):
            for count in range(1, 5):
                result.append((low, high, count))
    return result
```

8.r.6 [radix] † Posledním řadicím algoritmem, který v této kapitole prozkoumáme, je řazení po číslicích: obvyklé jméno pro tento algoritmus je „radix sort“, případně „bucket sort“. Algoritmy, které jsme viděli dosud, pracují všechny (krom distribution_sort) na principu srovnávání dvojic prvků. Tento princip je velmi obecný, ale často také omezující.

V této úloze se vrátíme k myšlence funkce distribution_sort a místo porovnávání prvků je budeme počítat, zvolíme si ale jiné kritérium. Naším cílem bude seřadit seznam čísel, a využijeme k tomu skutečnosti, že čísla lze rozložit na jednotlivé cifry (v nějaké poziční soustavě). Pro jednoduchost si zvolme soustavu desítkovou (algoritmus ve skutečnosti ale na konkrétní volbě soustavy nezávisí).

Základním stavebním kamenem bude procedura sort_by_digit, která:

1. přeuspořádá vstupní seznam tak, aby byl uspořádaný podle i-té číslice,
2. a to tak, aby přitom nezměnila relativní pořadí prvků, které mají na i-té pozici stejnou číslici.

Protože číslic je málo, ale hodnot v seznamu potenciálně hodně, hodí se na toto přeuspořádání právě funkce distribution_sort:

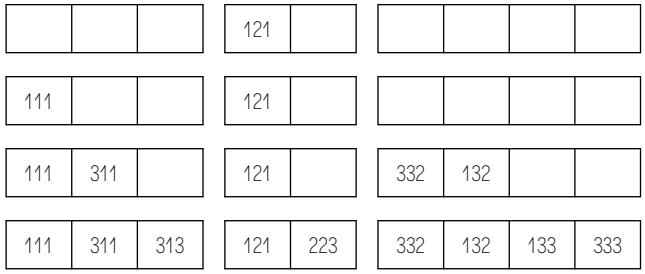
1. spočítáme, kolik vstupních čísel padne do kterého „kyblíčku“ (rozsahu prvků se stejnou i-tou cifrou),
2. pro každý kyblíček spočítáme, na jakých indexech se bude ve výsledném seznamu nacházet,
3. vstupní seznam v jednom průchodu do takto nachystaných kyblíčků rozřadíme (kyblíčky zaplňujeme ve stejném pořadí, v jakém iterujeme vstupní seznam).

Vyzbrojeni procedurou sort_by_digit už lehce seznam seřadíme: začneme od poslední cifry, a postupujeme doleva. Lehce se o správnosti tohoto postupu přesvědčíme indukcí:

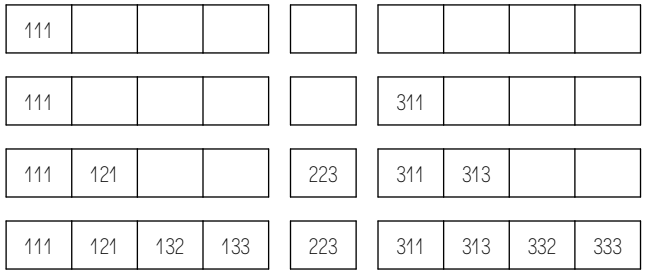
1. po první iteraci je seznam seřazen podle první (nejpravější) cifry,
2. předpokládejme, že po i-té iteraci je seznam seřazen podle cifer $i, i-1, \ldots, 0$; v iteraci $i+1$ bude procedurou sort_by_digit seřazen podle cifry $i+1$, ale ta nezměnila pořadí prvků, které jsou na pozici $i+1$ stejné: proto je po iteraci $i+1$ seznam seřazen podle cifer $i+1, i, i-1, \ldots, 0$.

Následující seznam je již seřazen podle nejnižší cifry. Ukažme si na něm zbytek algoritmu:

Spočítáme počty cifer na prostřední pozici a dostaneme: 3× 1, 2× 2, 4× 3. Nachystáme si příslušné kyblíčky a vyplňujeme je (například) zleva doprava:



Postup opakujeme na nejlevější pozici: 4× 1, 1× 2, 4× 3




```
def radixsort(to_sort: list[int]) -> list[int]:
    pass
```

8.v: Volitelné úlohy

8.v.1 [llsort] V tomto příkladu budeme pracovat se zřetězenými seznamy. Třídy Node a LinkedList jsou připraveny; nijak je nemodifikujte.

```
class Node:
    def __init__(self, value: int):
        self.value = value
```

```
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Implementujte proceduru sort_linked_list, která vzestupně seřadí zadaný zřetězený seznam. Nevytvářejte přitom žádné nové uzly ani nemodifikujte hodnoty (atributy value) těch existujících. Seřazení je třeba provést pouze pomocí změn atributů next (a head).

Není třeba vymýšlet nějaké optimalizace, kvadratické řešení je zde v pořádku.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

```
def sort_linked_list(llist: LinkedList) -> None:
    pass
```

8.v.2 [duplicates] V tomto příkladu budeme pracovat se zřetězenými seznamy. Třídy Node a LinkedList jsou připraveny; nijak je nemodifikujte.

```
class Node:
    def __init__(self, value: int):
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Implementujte proceduru, která dostane na vstup vzestupně seřazený jednosměrně zřetězený seznam, z tohoto seznamu odstraní všechny duplikáty (uzly se stejnými hodnotami) tak, že v něm nechá vždy pouze první výskyt. Odstraněné uzly funkce spojí do nového zřetězeného seznamu (se zachováním jejich pořadí) a ten vrátí.

Při řešení neměňte hodnoty atributu value ani nevytvářejte nové uzly typu Node, tj. jediné, co můžete s uzly dělat, je měnit odkazy na následující uzel.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

Příklad: Je-li zřetězený seznamu tvaru 1 → 2 → 2 → 2 → 7 → 7 → 10, pak procedura modifikuje tento seznam do tvaru 1 → 2 → 7 → 10 a vrátí zřetězený seznam tvaru 2 → 2 → 7.

```
def remove_duplicates(llist: LinkedList) -> LinkedList:
    pass
```

8.v.3 [diff] V tomto příkladu budeme pracovat se zřetězenými seznamy. Třídy Node a LinkedList jsou připraveny; nijak je nemodifikujte.

```
class Node:
    def __init__(self, value: int):
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None
```

Implementujte proceduru, která dostane na vstup dva vzestupně seřazené jednosměrně zřetězené seznamy a z prvního z těchto seznamů odstraní uzly s hodnotami, které se vyskytují ve druhém seznamu. Druhý zřetězený seznam musí zůstat nezměněn.

Při řešení neměňte hodnoty atributu value ani nevytvářejte nové uzly typu Node, tj. jediné, co můžete s uzly dělat, je měnit odkazy na následující uzel.

Očekávané řešení má složitost lineární vůči součtu délek vstupních seznamů.

V tomto příkladu je zakázáno použití Pythonovských datových struktur seznam, množina, slovník.

Příklad: Je-li první zřetězený seznamu tvaru 1 → 3 → 5 → 5 → 7 → 10 a druhý zřetězený seznam tvaru 1 → 1 → 2 → 5 → 12, pak procedura upraví první seznam do tvaru 3 → 7 → 10 (a druhý seznam nechá v původní podobě).

```
def list_diff(left: LinkedList, right: LinkedList) -> None:
    pass
```

Část S.2: Sada úloh k druhému bloku

V druhém bloku jsou následující domácí úkoly:

- a_connect_four – hra Connect Four,
- b_warehouse – práce se zbožím ve skladu,
- c_robot – simulace pohybu robota,
- d_life – celulární automat ve stylu „Game of Life“,
- e_stock_exchange – vypořádání pokynů na burze,
- f_tetris – hra Tetris.

První dva úkoly vyžadují pouze základní použití seznamů (z prvního bloku), další dva úkoly k tomu přidávají datové struktury z páté kapitoly a poslední dva úkoly navíc využívají uživatelsky definované datové typy (třídy).

S.2.a: warehouse

V tomto úkolu se budeme zabývat skladem zboží. Zboží je ve skladu uloženo po balících, které reprezentujeme trojicemi hodnot: množství (počet jednotek) zboží, jednotková cena zboží a datum exspirace. Všechny tři hodnoty budou vždy kladná celá čísla, přičemž datum exspirace bude vždy zadáno tak, aby jeho zápis v desítkové soustavě byl ve formátu YYYYMMDD dle ISO 8601.

```
Package = tuple[int, int, int]  # amount, price, expiration date
```

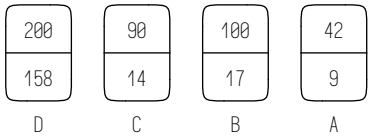
Obsah skladu budeme reprezentovat seznamem balíků, přičemž tento seznam bude vždy seřazen sestupně dle data exspirace. (Je zájmem společnosti, které sklad patří, aby se jako první prodaly balíky, jejichž konec trvanlivosti se blíží; přitom balíky budeme prodávat od konce seznamu.)

Nejprve implementujte funkci remove_expired, která ze skladu odstraní všechny balíky s prošlou trvanlivostí (tj. ty, jejichž datum exspirace předchází dnešnímu datu today, které je zadáno stejně jak je popsáno výše). Funkce vrátí seznam odstraněných balíků v opačném pořadí, než byly umístěny ve skladu.

```
def remove_expired(warehouse: list[Package],
                   today: int) -> list[Package]:
    pass
```

Dále pak implementujte funkci try_sell, která uskuteční prodej při zadaném maximálním množství max_amount a zadané maximální průměrné jednotkové ceně max_price. Přitom je cílem prodat co nejvíce zboží (v rámci respektování zadaných limitů). Prodávat je možno jak celé balíky, tak i jen jejich části; je tedy dovoleno existující balík rozbalit a odebrat z něj jen několik jednotek zboží (tím vlastně z jednoho balíku vzniknou dva – jeden zůstane ve skladu, druhý se dostane ke kupci). Je ovšem třeba postupovat tak, že se balíky odebírají pouze z konce seznamu reprezentujícího sklad – tj. není možno prodat balík (nebo jeho část), aniž by předtím byly prodány všechny balíky nacházející se v seznamu za ním. Funkce vrátí seznam balíků, které se dostaly ke kupci, a to v tom pořadí, jak se postupně ze skladu odebíraly.

Pro příklad uvažujme sklad s následujícími balíky (datum exspirace zde neuvádíme, horní číslo je množství, spodní cena; pořadí balíků odpovídá seřazení seznamu, prodáváme tedy „zprava“):



- Pokud by přišel požadavek na prodej s maximálním množstvím 500 a maximální průměrnou jednotkovou cenou 9, pak se prodá pouze celý balík A.
- Pokud by místo toho byla požadovaná maximální průměrná cena 12, pak se prodá celý balík A a 25 jednotek zboží z balíku B. (Balík B se tedy rozdělí: ve skladu zůstane balík s množstvím 75, ke kupci se dostane balík s množstvím 25.)
- Pokud by byla požadovaná maximální průměrná cena 14, pak se prodá celý balík A a 70 jednotek zboží z balíku B.
- Pokud by byla požadovaná maximální průměrná cena 15, pak se prodají celé balíky A, B a C.
- Pokud by byla požadovaná maximální průměrná cena 16, pak se prodají celé balíky A, B, C a dvě jednotky zboží z balíku D.
- Konečně pro maximální průměrnou cenu 81 se prodají všechny balíky.

```
def try_sell(warehouse: list[Package],
             max_amount: int, max_price: int) -> list[Package]:
    pass
```

S.2.b: robot

Představte si, že máme plán ve tvaru neomezené čtvercové sítě, na níž jsou položeny čtvercové dílky s nákresy ulic či křižovatek (něco jako kartičky ve hře Carcassone). Tyto dílky budeme reprezentovat jako množiny směrů, kterými je možné dílek opustit. Tedy např. dílek {NORTH, SOUTH} je ulice, která vede severojižním směrem, dílek {EAST, SOUTH, WEST} je křižovatka ve tvaru T, dílek {EAST} je slepá ulice (z toho dílku je možné se posunout pouze na východ, ale nikam jinam). Dovolujeme i prázdnou množinu, což je dílek, z nějž se nedá pohnout nikam.

```
Heading = int
NORTH, EAST, SOUTH, WEST = 0, 1, 2, 3
Tile = set[Heading]
```

Situaci na čtvercové síti popisujeme pomocí slovníku, jehož klíči jsou souřadnice a hodnotami dílky. Na souřadnicích, které ve slovníku nejsou, se žádný dílek nenachází. Souřadnice jsou ve formátu (x, y), přičemž x se zvyšuje směrem na východ a y směrem na jih.

```
Position = tuple[int, int]
Plan = dict[Position, Tile]
```

Napište nejprve predikát is_correct, který vrátí True právě tehdy, pokud na sebe všechny položené dílky správně navazují. Tedy je-li možno dílek nějakým směrem opustit, pak v tomto směru o jednu pozici vedle leží další dílek, a navíc je z tohoto dílku možné se zase vrátit.

```
def is_correct(plan: Plan) -> bool:
    pass
```

Dále implementujte čistou funkci run, která bude simulovat pohyb robota po plánu a vrátí jeho poslední pozici. Předpokládejte přitom, že plán je korektní (ve smyslu predikátu is_correct výše) a že robotova počáteční pozice je na některém z položených dílků. Robot se pohybuje podle následujících pravidel:

- Na počáteční pozici si robot vybere první ze směrů, kterým je možné se pohnout z počátečního dílku, a to v pořadí sever, východ, jih, západ. Pokud se z počáteční pozice není možné pohnout vůbec, funkce končí.
- V dalších krocích robot preferuje setrvat v původním směru (tj. pokud může jít rovně, půjde rovně). Není-li to možné, pohne se robot jiným ze směrů na aktuálním dílku – nikdy se ovšem nevrací směrem, kterým přišel (pokud dojde do slepé ulice, zastaví) a má-li více možností, vybere si tu, která pro něj znamená otočení doprava.
- Pokud robot přijde na dílek, kde už někdy v minulosti byl, zastaví.

```
def run(plan: Plan, start: Position) -> Position:
    pass
```

S.2.c: life

Hru **Life**[18] už jste si možná zkusili implementovat v rámci rozšířených příkladů ve čtvrté kapitole. V tomto úkolu budete implementovat její trochu složitější verzi. Místo jednoho života budeme simulovat souboj dvou různých organismů (modré a oranžové buňky), pozice po úmrtí buňky bude

[18] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

po několik kol neobyvatelná a budeme mít trochu jiná pravidla pro to, kdy buňky vznikají a zanikají. Kromě toho bude náš „svět“ neomezený a bude obsahovat „otrávené“ oblasti, kde žádné buňky nepřežijí.

Stav „světa“ je dán slovníkem, jehož klíči jsou 2D souřadnice a hodnotami čísla od jedné do šesti:

- číslo 1 reprezentuje živou modrou buňku,
- číslo 2 reprezentuje živou oranžovou buňku,
- čísla 3 až 6 reprezentují pozici, kde dříve zemřela buňka (čím větší číslo, tím víc času od úmrtí buňky uplynulo).

Pozice, které nejsou obsaženy ve slovníku, jsou prázdné.

```
Position = tuple[int, int]
State = dict[Position, int]
```

Stejně jako ve hře Life, za **okolí** pozice považujeme sousední pozice ve všech osmi směrech, tj. včetně diagonál. Základní pravidla vývoje světa jsou následující:

- Pokud jsou v okolí prázdné pozice přesně tři živé buňky, vznikne zde v dalším kole buňka nová. Barva nové buňky odpovídá většinové barvě živých buněk v okolí. Jinak zůstává prázdná pozice prázdnou.
- Pokud je v okolí živé buňky tři až pět živých buněk (na barvě nezáleží), buňka zůstane živou i v dalším kole (a ponechá si svou barvu). V opačném případě buňka umře a stav této pozice v dalším kole bude číslo 3.
- Má-li pozice stav 3 až 5, pak v dalším kole bude mít stav o jedna větší.
- Má-li pozice stav 6, v dalším kole bude prázdná.

„Otrávené“ pozice jsou zadány extra (jako množina) a mění základní pravidla tak, že živé buňky na otrávených pozicích **a v jejich okolí** vždy zemřou a na těchto pozicích (otrávených a jejich okolí) nikdy nevzniknou nové buňky.

Napište čistou funkci evolve, která dostane počáteční stav světa initial, množinu „otrávených“ pozic poison a počet kol generations a vrátí stav světa po zadaném počtu kol.

```
def evolve(initial: State, poison: set[Position],
           generations: int) -> State:
    pass
```

Pro vizualizaci je vám k dispozici soubor game_life.py, který vložte do stejného adresáře, jako je soubor s vaším řešením. Na začátku tohoto souboru jsou parametry vizualizace (velikost buněk, rychlost vývoje), popis iniciálního stavu světa a „otrávených“ pozic. Vizualizace volá vaši funkci evolve s parametrem generations vždy nastaveným na 1.

S.2.d: tetris

Jistě už jste někdy slyšeli o hře Tetris. Pokud ne, vítejte v civilizaci! Hledat můžete začít například tady: https://duckduckgo.com/?q=tetris. V tomto domácím úkolu si klon této hry naprogramujete.

Abyste si hru mohli vyzkoušet (poté, co implementujete všechny níže uvedené metody), je vám k dispozici soubor game_tetris.py, který vložte do stejného adresáře, jako je soubor s vaším řešením, případně jej upravte dle komentářů na jeho začátku a spusťte. Hra se ovládá těmito klávesami:

- pohyb doleva: šipka doleva nebo A,
- pohyb doprava: šipka doprava nebo D,
- pohyb dolů: šipka dolů nebo S (děje se také automaticky s nastavenou prodlevou),
- rychlý pád dolů: mezerník,
- otočení proti směru hodinových ručiček: Q nebo Page Up,
- otočení po směru hodinových ručiček: E nebo Page Down,
- ukončení hry: X,
- restart: R.

Třída Tetris, kterou máte implementovat, reprezentuje stav hry, tj. obsah herní oblasti (již spadlé kostky), aktuálně padající blok, jeho pozici a aktuální skóre. Způsob reprezentace je na vás. Testy i grafické rozhraní používají ke komunikaci s vaší třídou pouze zde popsané metody.

Rozměry herní oblasti budou zadány při inicializaci (funkci __init__). Všechny pozice mimo zadané rozměry považujeme za neprostupnou zeď. Souřadnice zde používáme ve tvaru (sloupec, řádek), přičemž pozice (0, 0) je v levém horním rohu herní oblasti. Čísla sloupců rostou zleva doprava, čísla řádků shora dolů.

Padající bloky reprezentujeme seznamem relativních souřadnic, přičemž (0, 0) je střed otáčení. Tedy např. [(-1, 0), (0, 0), (1, 0), (0, 1)] je tetromino tvaru T otočené směrem dolů, které se bude otáčet kolem své prostřední kostky. Blok [(-1, -1), (0, -1), (1, -1), (0, 0)] má stejný tvar, ale otáčí se kolem své „spodní nožičky“. Střed otáčení nemusí být nutně součástí bloku, např. [(-1, -1), (-1, 0), (-1, 1), (0, 1)] je tetromino tvaru L, které se otáčí kolem prázdného místa ve svém rohu.

Přestože se v grafickém rozhraní používají pouze tetromina (tedy klasické tetrisové bloky), vaše řešení musí být obecné a fungovat s libovolnými tvary bloků.

Poznámka: Protože za zeď považujeme i prostor „nad“ herní oblastí, může se v mnoha případech stát, že blok, který se nově objevil, nebude možné otočit, dokud se neposune o něco níže. Ačkoli reálné implementace tuto možnost většinou nějak ošetřují, zde pro zjednodušení nic takového neděláme a považujeme to za očekávané chování.

```
Position = tuple[int, int]

class Tetris:
```

Po inicializaci by měla být herní oblast prázdná, o zadaných rozměrech. Není žádný padající blok a skóre je nastaveno na 0.

```
    def __init__(self, cols: int, rows: int):
        pass
```

Čistá metoda get_score vrátí aktuální skóre.

```
    def get_score(self) -> int:
        pass
```

Metoda-predikát has_block vrátí True právě tehdy, existuje-li padající blok.

```
    def has_block(self) -> bool:
        pass
```

Metoda add_block přidá do hry padající blok na zadaných souřadnicích. Pokud přidání bloku není možné (překrýval by se s již položenými kostkami), metoda situaci nezmění a vrátí False; jinak vrátí True. Metoda bude volána pouze tehdy, neexistuje-li žádný padající blok. Seznam block nijak nemodifikujte. Pokud si ho hodláte někam uložit, tak buďto zařiďte, aby se ani později nemodifikoval, nebo si vytvořte jeho kopii.

```
    def add_block(self, block: list[Position],
                  col: int, row: int) -> bool:
        pass
```

Metoda left posune padající blok o jednu pozici doleva, je-li to možné. Tato metoda, stejně jako všechny následující metody pohybu, bude volána jen tehdy, existuje-li padající blok.

```
    def left(self) -> None:
        pass
```

Metoda right posune padající blok o jednu pozici doprava, je-li to možné.

```
    def right(self) -> None:
        pass
```

Metoda rotate_cw otočí padající blok po směru hodinových ručiček o 90 stupňů, je-li to možné.

```
    def rotate_cw(self) -> None:
        pass
```

Metoda rotate_ccw otočí padající blok proti směru hodinových ručiček o 90 stupňů, je-li to možné.

```
    def rotate_ccw(self) -> None:
        pass
```

Metoda down posune padající blok o jednu pozici směrem dolů. Pokud takový

posun není možný, kostky z padajícího bloku se napevno umístí do herní oblasti; zcela zaplněné řádky se pak z oblasti vymažou a skóre se zvýší o druhou mocninu počtu vymazaných řádků.

```
def down(self) -> None:
    pass
```

Metoda drop shodí padající blok směrem dolů (o tolik pozic, o kolik je to možné). Kostky z padajícího bloku se pak napevno umístí do herní oblasti; zcela zaplněné řádky se pak z oblasti vymažou a skóre se zvýší o druhou mocninu počtu vymazaných řádků.

```
def drop(self) -> None:
    pass
```

Čistá metoda tiles vrátí seznam všech pozic, na nichž má být vykreslena kostka – tedy jednak všechny položené kostky v herní oblasti, jednak všechny kostky tvořící padající blok. Na pořadí pozic v seznamu nezáleží. Tuto metodu používají jak testy pro ověření správnosti implementace, tak grafické rozhraní pro vykreslení hry.

```
def tiles(self) -> list[Position]:
    pass
```

Demonstrační příklady:

1. bsearch – binární vyhledávání podruhé
2. tsearch – hledání v obyčejných binárních stromech
3. lookup – hledání v binárních **vyhledávacích** stromech
4. minmax – vyhodnocení „min/max“ stromu
5. cycle – použití koncové rekurze

Elementární příklady:

1. count – počet uzlů ve stromě
2. leafsum – součet listů n-árního stromu
3. depth – délka nejdelší větve ve stromě

Přípravy:

1. evaluate – vyhodnocení aritmetického výrazu zadaného stromem
2. rpn – od stromu k postfixovému zápisu
3. children – úprava atributu v každém uzlu stromu
4. treezip – od dvojice stromů ke stromu dvojic
5. build † – převod závorkovaného řetězce na strom
6. prune – prořezávaní binárních stromů

Rozšířené úlohy:

1. treesum – součet všech uzlů ternárního stromu
2. brackets – závorkování výrazu zadaného stromem
3. heap – kontrola haldové vlastnosti binárního stromu
4. avl – kontrola tzv. AVL vlastnosti stromu
5. bdd – vyhodnocení binárního rozhodovacího diagramu
6. average – průměrná délka větve stromu

9.1: Programovací jazyk

Tato kapitola přináší do jazyka dva nové prvky, které oba souvisí s typy:

1. Typovou anotaci typ_1 | typ_2 | … | typ_n, která realizuje tzv. **součtové typy**, kdy o nějaké hodnotě umíme říct, že je určitě některého z vyjmenovaných typů, ale který konkrétně to bude se rozhodne až za běhu programu.
2. Zabudovaný predikát isinstance(value, type), který rozhodne, je-li hodnota value typu type. Tento predikát lze s výhodou použít v kombinaci se součtovými typy, kdy se v programu potřebujeme rozhodnout podle skutečného typu hodnoty value.
   V těle podmíněného příkazu if isinstance(value, type) pak platí, že hodnota value má i staticky (tzn. pro účely typové kontroly programem mypy) přiřazen typ type.

9.d: Demonstrace (ukázky)

9.d.1 [bsearch] Minulý týden jsme si, mimo jiné, ukázali algoritmus pro efektivní hledání hodnoty v seřazeném seznamu, a to metodou **půlení intervalu**. Dnes si ukážeme jinou implementaci téhož algoritmu: místo cyklu použijeme koncovou rekurzi. Takto zapsaný algoritmus nám poskytne trochu jinou perspektivu na známý problém a zároveň připomene základní myšlenku rekurze, kterou již znáte z přednášky. Při studiu této ukázky Vám doporučujeme otevřít si také ukázku 08/bin_tree.py a oba přístupy (iterativní z minulého týdne a rekurzivní v tomto souboru) průběžně srovnávat.

Protože rekurzivní implementace bude potřebovat dodatečné parametry, rozdělíme si ji na dva predikáty: bin_search_rec, která provede samotné rekurzivní hledání, a bin_search, která rekurzi pouze nastartuje (a slouží tak zejména jako příjemnější rozhraní pro volání funkce bin_search_rec).

Chceme-li použít rekurzi, musíme problém formulovat tak, aby měl jasně určené podproblémy (nebo podproblém), který je v nějakém smyslu menší, než původní problém. Dále pak budeme chtít, aby bylo jednoduché odpovědi na podproblémy zkombinovat tak, abychom dostali odpověď na původní problém. V případě, kdy je podproblém pouze jeden, je často možné použít navíc **koncovou rekurzi**: výsledek (vhodně zvoleného) podproblému je přímo i výsledkem celého problému. Koncová rekurze má proti té obecné dvě základní výhody:

- takto zapsaný výpočet lze provádět **efektivně** (bez použití dodatečné paměti),
- o koncové rekurzi se lépe uvažuje, protože má zvlášť jednoduchou strukturu.

Na to, abychom „objevili“ v algoritmu vhodné podproblémy, trochu si jej zobecníme: místo hledání v seznamu si jej zadefinujeme, jako hledání v nějakém souvislém úseku daného seznamu: konkrétně v polouzavřeném intervalu $\langle l, h)$ kde l je dané parametrem low a h je dané parametrem high. Toto by nám již mělo nápadně připomínat implementaci z minulého týdne.

Pro úplnost, predikát bin_search_rec odpovídá na otázku „je hodnota value přítomna v seznamu records na některém indexu z intervalu $\langle l, h)$?“

```
def bin_search_rec(records: list[int], value: int,
                   low: int, high: int) -> bool:
```

V řešení jednotlivých případů začneme od toho nejjednoduššího: je-li vstupní interval prázdný, hodnota value se v něm jistě nenachází. Tato podmínka je analogická ukončovací podmínce cyklu while z iterativní verze. Vrátíme tedy hodnotu False a jsme hotovi.

```
    if low == high:
        return False
```

Řešení ostatních případů záleží na tom, ve které části seznamu se musí hodnota nacházet (je-li přítomna). Tyto případy jsou analogické k případům, které iterativní verze ošetřovala v těle cyklu. Nejprve si vybereme vhodný dělící bod m (zhruba uprostřed intervalu). Zejména platí, že m (v programu reprezentované proměnnou mid) vždy spadá do intervalu $\langle l, h)$.

```
    mid = low + (high - low) // 2
```

Je-li tedy hledaná hodnota přímo na indexu mid, je určitě v intervalu $\langle l, h)$ a tedy můžeme odpovědět True. Argument proč to stačí je analogický k iterativní verzi.

```
    if records[mid] == value:
        return True
```

Jednoduché případy máme vyřešeny, nyní zbývají ty složitější: totiž ty, které vedou na nějaký podproblém. Je-li hodnota na indexu m („uprostřed“ seznamu) menší než value, znamená to, že je-li hodnota value v seznamu někde přítomna, musí to být v horní části.

Kýžený podproblém je tedy „je hodnota value přítomna v seznamu records na indexech z intervalu $\langle m+1, h)$?“ Je zde dobře vidět i struktura koncové rekurze: odpověď na novou otázku je zároveň odpovědí na tu původní (totiž „je value přítomno v intervalu indexů $\langle l, h)$“). Výsledek řešení podproblému můžeme přímo, bez jakýchkoliv dalších úprav, vrátit.

```
    if records[mid] < value:
        return bin_search_rec(records, value, mid + 1, high)
```

Zbývá poslední možnost: hodnota musí být v spodní části prohledávaného intervalu, a tedy podproblém, který musíme vyřešit, je „je hodnota value přítomna v intervalu indexů $\langle l, m)$?“

```
    if records[mid] > value:
        return bin_search_rec(records, value, low, mid)
```

Protože jsme pokryli všechny možnosti, do tohoto místa se již program nemůže dostat. Toto naznačíme tvrzením False.

```
    assert False
```

Samotný predikát bin_search se již pomocí bin_search_rec vyjádří velice snadno: stačí zvolit interval $\langle l, h)$ tak, že pokrývá právě všechny platné indexy seznamu records.

```
def bin_search(records: list[int], value: int) -> bool:
    return bin_search_rec(records, value, 0, len(records))
```

Protože řešený problém je identický jako minulý týden, budou i testy identické.

```
def main() -> None:  # demo
    for low, high, count in test_parameters():
        for records in sorted_lists(low, high, count, []):
            for v in range(low - 1, high + 1):
                assert bin_search(records, v) == (v in records)

def sorted_lists(low: int, high: int, count: int,
                 prefix: list[int]) -> list[list[int]]:
    if count == 0:
        return [prefix]

    result = []
    for x in range(low, high):
        result.extend(sorted_lists(x, high, count - 1, prefix + [x]))
    return result

def test_parameters() -> list[tuple[int, int, int]]:
    result = []
    for high in range(10):
        for low in range(high):
            for count in range(0, 8):
                result.append((low, high, count))
    return result
```

9.d.2 [tsearch] V této ukázce budeme pracovat se **stromy**. Strom je datová struktura, která se podobá zřetězenému seznamu, s jedním zásadním rozdílem: uzly nemají následníka jednoho, ale několik. Podle toho, kolik, dělíme stromy na binární (2 následníci), ternární (3 následníci), atd. Lze také uvažovat stromy s proměnným počtem následníků (takovým se většinou říká n-ární). Počátečnímu uzlu (tomu, který nemá ve stromě žádné předchůdce) často říkáme kořen.

Stromy sdílí se zřetězenými seznamy krom podobné struktury i jednu velmi důležitou vlastnost: jsou to **rekurzivní datové struktury**. Co to znamená? U seznamu to, že následník uzlu seznamu tvoří opět seznam (navíc striktně menší seznam). A u stromu zase platí, že každý následník je podstrom (striktně menší strom).

Tato struktura velmi dobře koresponduje s naší představou o rekurzi: problém rozdělíme na podproblémy (pro každý podstrom vznikne jeden) a dílčí výsledky nějak zkombinujeme na výsledek celkový. Elementární (bázové) podproblémy pak tvoří stromy o jediném uzlu (takové, které nemají žádné podstromy, známé též jako **listy**), případně stromy prázdné (je-li to výhodné).

Strom budeme reprezentovat analogicky k uzlu zřetězeného seznamu. Prázdný strom budeme reprezentovat hodnotou None.

```
class Tree:
    def __init__(self, value: int, left: 'Tree | None',
                 right: 'Tree | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: int) -> Tree:
    return Tree(value, None, None)
```

Jako první příklad na práci se stromy si naprogramujeme test na přítomnost hodnoty ve stromě. Vstupem je (potenciálně prázdný) strom a hledaná hodnota.

```
def search(tree: Tree | None, value: int) -> bool:
```

Aplikujeme nyní již snad dobře známý postup: nejprve vyřešíme bázové (jednoduché) případy: je-li strom prázdný, hledaná hodnota se v něm jistě nenachází (vracíme False).

```
    if tree is None:
        return False
```

Naopak, je-li hledaná hodnota uložena v aktuálním uzlu, můžeme rovnou vrátit True.

```
    if value == tree.value:
        return True
```

Zbývají případy, které neumíme řešit přímo: víme ale, že je-li hodnota ve stromě přítomna, musí to být v levém nebo v pravém podstromě. Protože podstromy jsou menší (jednodušší) než celý strom, jedná se o podproblémy, které můžeme řešit rekurzí. Aplikujeme tedy predikát search na oba podstromy: hodnota je ve stromě přítomna, je-li přítomna alespoň v jednom z jeho podstromů.

```
    return search(tree.left, value) or search(tree.right, value)
```

Nezbývá, než predikát search otestovat na několika jednoduchých vstupech.

```
def main() -> None:  # demo
    t1 = Tree(7, leaf(2), Tree(1, leaf(5), leaf(6)))
    assert search(t1, 7)
    assert search(t1, 2)
    assert search(t1, 1)
    assert search(t1, 5)
    assert search(t1, 6)
    assert not search(t1, 4)
    t2 = Tree(8, t1, leaf(10))
    assert not search(t2, 4)
    assert search(t2, 8)
    assert search(t2, 10)
    assert search(t2, 1)
    assert search(t2, 5)
```

9.d.3 [lookup] V této ukázce budeme pokračovat v práci s binárními stromy. Definice stromu tedy zůstává od předchozí ukázky nezměněna.

```
class Tree:
    def __init__(self, value: int, left: 'Tree | None',
                 right: 'Tree | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: int) -> Tree:
    return Tree(value, None, None)
```

Analogií k seřazenému seznamu je takzvaný **vyhledávací strom**. Tento má tu vlastnost, že všechny hodnoty uložené v levém podstromě jsou menší nebo rovny hodnotě uložené v zkoumaném uzlu, a naopak, hodnoty v pravém podstromě jsou větší nebo rovny. Podobně jako v uspořádaném seznamu, lze ve vyhledávacím stromě test na přítomnost hodnoty provést výrazně rychleji, než ve stromě obecném.

```
def lookup(tree: Tree | None, value: int) -> bool:
```

Jednoduché případy jsou zcela stejné, jako při hledání v obecném stromě.

```
    if tree is None:
        return False
    if value == tree.value:
        return True
```

Zajímavá změna se objeví v rekurzivním případě: podobně jako při hledání půlením intervalu můžeme srovnáním hledané hodnoty a hodnoty v aktuálním uzlu rozhodnout, ve kterém podstromě se hledaná hodnota musí nacházet (je-li přítomna). Je-li hledaná hodnota menší, než ta v aktuálním uzlu, víme jistě, že se v pravém podstromě určitě nemůže objevit. Stačí nám tedy vyřešit jediný podproblém, a to test na přítomnost hodnoty v levém podstromě. Protože máme jediný podproblém, nabízí se možnost použít koncovou rekurzi: musí ale navíc platit, že řešení podproblému je přímo i řešením problému. Rozmyslete si, že tomu tak skutečně je!

```
    if value < tree.value:
        return lookup(tree.left, value)
```

Opačný případ je zcela analogický: můžeme-li vyloučit přítomnost hodnoty v levém podstromě, zbývá jediný podproblém, který je navíc menší než ten aktuální (podstrom je jednodušší než celý strom). Opět postupujeme koncovou rekurzí.

```
    if value > tree.value:
        return lookup(tree.right, value)
```

Mělo by být zřejmé, že jsme vyčerpali všechny možnosti, program se do tohoto místa tedy nemůže dostat. Tuto skutečnost opět deklarujeme tvrzením False.

```
    assert False
```

Krom predikátu lookup zadefinujeme ještě jeden predikát: takový, který zjistí, je-li nějaký strom korektním vyhledávacím stromem. Predikát ale pro rozklad na podproblémy stačit nebude: lze sestavit strom ze dvou korektních vyhledávacích stromů takový, že výsledek nebude korektním vyhledávacím stromem, ale lokálně (jen z jednoho vrcholu a jeho přímých následníků) to nebude lze poznat. Třeba tento: Tree(5, Tree(2, leaf(1), leaf(10)), leaf(8)).

Musíme vyřešit **silnější problém**: takový, který nám umožní složit správné řešení z vyřešených podproblémů. Jaké jsou lokální vlastnosti korektního vyhledávacího stromu? Jsou to:

- maximum levého podstromu je ≤ hodnota aktuálního uzlu,
- minimum pravého podstromu je ≥ hodnota aktuálního uzlu,
- levý i pravý podstrom jsou korektní.

Potřebujeme tedy funkci, která zjistí korektnost, minimum a maximum daného (pod)stromu: víme už, že z těchto informací umíme zjistit korektnost celého stromu. Na to, abychom mohli použít rekurzi, musíme ještě zjistit minimum a maximum: za předpokladu, že je strom korektní, platí:

- minimum levého podstromu je zároveň minimum celého stromu,
- maximum pravého podstromu je zároveň maximum stromu.

Všechny informace tedy umíme spočítat lokálně, z informacích získaných řešením podproblémů. Můžeme tedy přistoupit k rekurzivnímu řešení problému.

Abychom si trochu zjednodušili život, přidáme si umělý parametr: příhodnou mez, kterou použijeme jako minimum i maximum, je-li zadaný strom prázdný (takový strom totiž žádné přirozené meze nemá). Tento postup nám oproti variantě s None ušetří spoustu psaní.

```
def is_correct_rec(tree: Tree | None, bound: int) \
        -> tuple[bool, int, int]:
```

Jako vždy, nejprve vyřešíme jednoduché případy: prázdný strom je korektní (splňuje všechny požadavky). Zároveň nemá žádné přirozené meze, proto použijeme tu, kterou nám volající předal jako výchozí.

```
    if tree is None:
        return (True, bound, bound)
```

Je-li strom neprázdný, získáme vlastnosti levého i pravého podstromu rekurzivním voláním.

```
    l_ok, l_min, l_max = is_correct_rec(tree.left, tree.value)
    r_ok, r_min, r_max = is_correct_rec(tree.right, tree.value)
```

Podle kritérií uvedených výše vypočteme, je-li strom jako celek korektní.

```
    this_ok = l_ok and r_ok and l_max <= tree.value <= r_min
```

Nyní nám stačí sestavit návratovou hodnotu. Není-li strom korektní, nemusíme se správností mezí zabývat: žádný strom, který má nekorektní podstrom, nemůže být korektní, bez ohledu na meze svých podstromů.

```
    return (this_ok, l_min, r_max)
```

Protože jsme potřebovali formulovat silnější problém, má funkce is_correct_rec nesprávné rozhraní: zejména to není predikát (výsledkem je n-tice, nikoliv bool), navíc má nežádoucí parametr bound. Původně zamýšlený predikát ale už pomocí is_correct_rec lehce zapíšeme:

```
def is_correct(tree: Tree) -> bool:
    ok, _, _ = is_correct_rec(tree, 0)
    return ok

def main() -> None:  # demo
    t1 = Tree(7, Tree(4, leaf(1), leaf(5)), leaf(8))
    assert is_correct(t1)
    assert lookup(t1, 7)
    assert lookup(t1, 5)
    assert lookup(t1, 1)
    assert lookup(t1, 4)
    assert lookup(t1, 8)
    assert not lookup(t1, 9)
    assert not lookup(t1, 2)
    assert not lookup(t1, 6)

    t2 = Tree(5, Tree(2, leaf(1), leaf(10)), leaf(8))
    assert not is_correct(t2)
```

9.d.4 [minmax] Tato ukázka přinese oproti předchozím dvě rozšíření:

- n-ární stromy (tedy takové, kde počet potomků jednoho uzlu není předem omezen – potomky budeme ukládat do seznamu),
- nepřímá (nebo vzájemná – mutual) rekurze, tedy situaci, kdy nějaká funkce *f* ve svém řešení používá k řešení menších podproblémů funkci *g* a naopak, *g* využívá pro menší podproblémy funkci *f*.

Definice stromu se od předchozích liší pouze reprezentací následníků. Protože se jedná o seznam, tento může být přirozeně prázdný a není tedy potřeba pro neexistující následníky používat None. Protože ale budeme chtít reprezentovat stromy, které nemají hodnoty ve všech uzlech, objeví se None tentokrát jako možná hodnota uzlu.

```
class Tree:
    def __init__(self, value: int | None, children: list['Tree']):
        self.value = value
        self.children = children
```

Jaký problém tedy budeme řešit? Uvažme strom, který má dva typy vnitřních uzlů (vnitřní uzly jsou ty, které mají nějaké následníky): uzly typu „min“ a uzly typu „max“. Tyto jsou ve stromě navíc rozvrženy tak, že uzel „max“ má následníky pouze typu „min“ a naopak, uzel „min“ má následníky pouze typu „max“.

Bude výhodné o situaci uvažovat tak, že to, které uzly budou „min“ a které „max“ bude záviset od jejich vzdálenosti od kořene, a od toho, je-li kořen typu „min“ nebo typu „max“. Krom vnitřních uzlů má strom **listy**: to jsou právě ty uzly, které již žádné následníky nemají. Náš „minmax“ strom bude v listech obsahovat celá čísla. Hodnotu vnitřního uzlu pak spočítáme jako minimum (je-li to uzel typu „min“) nebo maximum (je-li typu „max“) hodnot všech jeho následníků.

Funkce nazveme tree_minmax (kořen je typu „min“) a tree_maxmin (kořen je typu „max“). Z popisu výše je zřejmé, že je-li kořen stromu typu „min“, budou kořeny všech podstromů typu „max“: rekurzivní volání proto bude vždy používat opačnou funkci.

```
def tree_minmax(tree: Tree) -> int:
```

Jako vždy, nejprve vyřešíme jednoduché případy: konkrétně zde případ, kdy je uzel listem (má hodnotu nastavenu přímo).

```
    if tree.value is not None:
        return tree.value
```

Ze seznamu potomků (podstromů) vytvoříme seznam jejich hodnot použitím funkce tree_maxmin. Z tohoto seznamu již lehce získáme výsledek: protože kořen je typu „min“, bude to minimum z hodnot všech následníků.

```
    return min([tree_maxmin(child) for child in tree.children])
```

Funkce tree_maxmin je vůči tree_minmax zcela symetrická:

```
def tree_maxmin(tree: Tree) -> int:
    if tree.value is not None:
        return tree.value
    return max([tree_minmax(child) for child in tree.children])
```

Funkce již zbývá pouze otestovat.

```
def internal(children: list[Tree]) -> Tree:
    return Tree(None, children)

def leaf(value: int) -> Tree:
    return Tree(value, [])
```

```
def main() -> None:  # demo
    t1 = internal([internal([leaf(1), leaf(2)]),
                   internal([leaf(3), leaf(4)])])
    t2 = internal([t1, internal([leaf(4), leaf(5), leaf(6)])])
    assert tree_minmax(t1) == 2
    assert tree_maxmin(t1) == 3
    assert tree_minmax(t2) == 3
    assert tree_maxmin(t2) == 4
```

9.d.5 [cycle] Mějme následující problém: na vstupu je zadaný seznam čísel a počáteční index. V každém kroku k aktuálnímu indexu přičteme hodnotu na tomto indexu uloženou. Mohou nastat tyto možnosti:

1. index po konečném počtu iterací „vypadne“ z rozsahu seznamu,
2. výpočet se zacyklí a bude navštěvovat nějakou množinu indexů „donekonečna“.

Zajímá nás která možnost nastane, a v případě 2 také délka cyklu, který se bude opakovat (t.j. velikost množiny indexů, které budou v cyklu navštěvovány).

V této ukázce naprogramujeme čistou funkci cycle, která tento problém řeší. Problém rozdělíme na dvě části: nejprve zjistíme, která z možností nastala. Poté, je-li to možnost 2, zjistíme délku cyklu. Jako cvičení si můžete zkusit implementovat verzi, která problém vyřeší na jeden průchod, za cenu uložení dodatečné informace.

Použijeme koncovou rekurzi, ale tato bude mít trochu jiný charakter, než v předchozích ukázkách: problém, který řešíme, nemá žádnou jasnou (statickou) strukturu podproblémů, a nemůžeme tedy použít jednoduchou strukturální rekurzi.

Hlavní myšlenka rekurze nicméně zůstane zachována: nejprve vyřešíme elementární případy, kdy je odpověď na první pohled jasná. Ty zbývající musíme nějakým vhodným způsobem převést na jednodušší instance: to, v čem se tento příklad liší od těch předchozích je, že nemáme k dispozici jasného kandidáta na vhodnou jednodušší instanci (chybí nám již zmiňovaná struktura podproblémů).

Jak tedy měřit jednoduchost? Neexistuje žel žádná univerzální odpověď ani univerzální postup, a „uvidět“ vhodné řešení vyžaduje určitý cvik.

Zaměřme se tedy na funkci cycle_detect, která bude zjišťovat, jestli se výpočet zacyklí nebo nikoliv. V tomto případě se jako vhodné měřítko jednoduchosti jeví kritérium „kolik indexů jsme ještě během výpočtu nenavštívili?“. Jednou z indicií je i to, že když je tento počet 0, stojíme před elementárním případem – index je buď platný (a tedy navštívený: našli jsme cyklus) nebo neplatný. Pro žádný složitější případ nezbývá prostor. Máme tedy jakousi záruku, že dokážeme-li postupně toto číslo snižovat, dříve nebo později narazíme na elementární problém. To je dobře.

Z praktického hlediska je ale lepší pamatovat si množinu použitých indexů, nikoliv těch nepoužitých: to ale není problém, protože tyto množiny jsou ve velmi jednoduchém vztahu (jsou vzájemnými doplňky v množině všech platných indexů). Přidáme-li index do množiny navštívených indexů, je to totéž, jako bychom jej odebrali z množiny indexů nenavštívených.

Funkce cycle_detect tedy bude mít 3 parametry: samotný seznam čísel, aktuální index a množinu již navštívených indexů. Výsledkem pak bude libovolný index, který se během výpočtu zopakoval (existuje-li, jinak None).

```
def cycle_detect(numbers: list[int], index: int,
                 visited: set[int]) -> int | None:
```

Podobně jako v předchozím, nejprve vyřešíme jednoduché případy: je-li index mimo meze seznamu numbers, není co řešit: vracíme None.

```
    if index < 0 or index >= len(numbers):
        return None
```

Naopak, je-li index přítomen v množině visited, víme, že se během výpočtu zopakoval a můžeme jej tedy vrátit.

```
    if index in visited:
        return index
```

Ve zbývajících případech nemůžeme přímo rozhodnout. Můžeme ale aktuální index označit za navštívený, provést krok výpočtu, a novou instanci problému prohlásit za jednodušší: díky tomu můžeme zbytek práce bezpečně delegovat na rekurzivní volání cycle_detect.

Vzhledem k předchozímu víme, že index dosud nebyl navštívený, tedy jeho přidáním se množina visited zvětší o 1, a tedy počet nenavštívených indexů o 1 klesne. Víme tedy, že takto formulovaná nová instance je blíže elementárnímu případu než ta stávající.

```
    jump_to = index + numbers[index]
    return cycle_detect(numbers, jump_to, visited | {index})
```

Funkce cycle_length je ještě o něco zapeklitější. Nejlepší míra „jednoduchosti“ je zde počet kroků, které musíme provést, abychom se z indexu index dostali na index start. Tato informace ale není vůbec nikde ve funkci přítomna, a není ani jasné, že je tento počet konečný. Skutečně, vhodnou volbou parametrů můžeme způsobit, že funkce cycle_length nikdy neskončí (například numbers = [1, 0], start = 0, index = 1).

Z pátého týdne ale víme, že funkce mohou mít **vstupní podmínku**: toho zde s výhodou využijeme. Aby funkce cycle_length smysluplně fungovala, musí platit, že index start je z indexu index dosažitelný konečným počtem kroků výpočtu – toto kritérium tedy zvolíme jako vstupní podmínku.

```
def cycle_length(numbers: list[int], index: int,
                 start: int, count: int) -> int:
```

Protože budeme začínat v situaci, kdy platí index == start, ale ještě jsme žádný krok výpočtu neprovedli (count je 0), musíme si elementární případ pohlídat: ten totiž nastane pouze je-li count alespoň 1.

```
    if count and index == start:
        return count
```

Nyní zbývá vyřešit rekurzivní volání. Ze vstupní podmínky víme, že z index do start se dostaneme konečným počtem kroků výpočtu. Provedeme-li tedy krok výpočtu z indexu index, tato vzdálenost se o jedna zmenší. Protože byla na začátku konečná (byla splněna vstupní podmínka), bude jistě konečná i po provedení kroku výpočtu: vstupní podmínka funkce cycle_length je i v nové situaci splněna (toto je velmi důležité ověřit!) a můžeme tedy provést rekurzivní volání. Zároveň víme, že se jedná o „jednodušší“ instanci (vzdálenost se nutně zmenšila).

```
    jump_to = index + numbers[index]
    return cycle_length(numbers, jump_to, start, count + 1)
```

Nyní už je jednoduché funkce zkombinovat do funkce cycle. Všimněte si, že výstupní podmínka funkce cycle_detect nám zaručuje splnění vstupní podmínky funkce cycle_length.

```
def cycle(numbers: list[int], start: int) -> int:
    cycle_start = cycle_detect(numbers, start, set())

    if cycle_start is None:
        return 0

    return cycle_length(numbers, cycle_start, cycle_start, 0)
```

Na závěr pár jednoduchých testů:

```
def main() -> None:  # demo
    assert cycle([0], 0) == 1
    assert cycle([1], 0) == 0
    assert cycle([1, -1], 0) == 2
    assert cycle([2, 0, -2], 0) == 2
    assert cycle([2, 0, -2], 1) == 1
    assert cycle([2, 0, -2], 2) == 2
    assert cycle([1, 1, 1], 0) == 0
    assert cycle([1, 1, -1], 0) == 2
```

9.e: Elementární příklady

9.e.1 [count] Třída Tree reprezentuje (neohodnocený) binární strom. Prázdný strom je reprezentován hodnotou None.

```
class Tree:
```

```
    def __init__(self, left: 'Tree | None',
                 right: 'Tree | None'):
        self.left = left
        self.right = right

def leaf() -> Tree:
    return Tree(None, None)
```

Napište čistou funkci, která vrátí počet uzlů v zadaném stromě.

```
def count(tree: Tree | None) -> int:
    pass
```

9.e.2 [leafsum] Třída Tree bude tentokrát reprezentovat n-ární strom, který má v uzlech uloženy celočíselné hodnoty.

```
class Tree:
    def __init__(self, value: int, children: list['Tree']):
        self.value = value
        self.children = children
```

Napište (čistou) funkci, která na vstupu dostane instanci výše popsaného stromu a vrátí součet čísel ve všech jeho listech (uzlech bez potomků).

```
def sum_leaves(tree: Tree) -> int:
    pass
```

9.e.3 [depth] Třída Tree reprezentuje (neohodnocený) binární strom. Prázdný strom je reprezentován hodnotou None.

```
class Tree:
    def __init__(self, left: 'Tree | None',
                 right: 'Tree | None'):
        self.left = left
        self.right = right

def leaf() -> Tree:
    return Tree(None, None)
```

Napište čistou funkci, která vrátí hloubku zadaného stromu, tzn. délku jeho nejdelší větve (posloupnosti uzlů od kořene k listu).

```
def depth(tree: Tree | None) -> int:
    pass
```

9.p: Přípravy

9.p.1 [evaluate] V tomto příkladu budeme pracovat se stromy, které reprezentují aritmetické výrazy. Tyto mají následující strukturu:

- konstantu reprezentuje strom, který má oba podstromy prázdné,
- složený výraz je reprezentován stromem, který má v kořenu uložen operátor (+ nebo *) a jeho neprázdné podstromy reprezentují operandy.

Žádné jiné uzly ve stromě přítomny nebudou.

```
class Tree:
    def __init__(self, value: str | int,
                 left: 'Tree | None',
                 right: 'Tree | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: int) -> Tree:
    return Tree(value, None, None)
```

Napište čistou funkci, která na vstupu dostane instanci výše popsaného stromu a vrátí výsledek vyhodnocení výrazu, který tento strom reprezentuje.

```
def evaluate(tree) -> int:
    pass
```

9.p.2 [rpn] V tomto příkladě budeme opět pracovat s aritmetickými výrazy. Tyto mají následující strukturu:

- konstantu reprezentuje strom, který má oba podstromy prázdné,
- složený výraz je reprezentován stromem, který má v kořenu uložen operátor (+ nebo *) a jeho neprázdné podstromy reprezentují operandy.

Žádné jiné uzly ve stromě přítomny nebudou.

```
class Tree:
    def __init__(self, value: str,
                 left: 'Tree | None',
                 right: 'Tree | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: str) -> Tree:
    return Tree(value, None, None)
```

Napište čistou funkci, která dostane jako parametr instanci výše uvedeného stromu reprezentující nějaký aritmetický výraz, a vrátí seznam řetězců, ve kterém je tento výraz zapsán v postfixové (rpn) notaci. Každý prvek bude odpovídat právě jednomu uzlu vstupního stromu.

```
def to_rpn(tree) -> list[str]:
    pass
```

9.p.3 [children] Uvažme n-ární strom, který má v uzlech uloženu volitelnou hodnotu typu int.

```
class Tree:
    def __init__(self, children: list["Tree"]):
        self.value: int | None = None
        self.children = children
```

Napište proceduru, která obdrží instanci výše popsaného stromu, a vyplní atributy value všech jeho uzlů tak, aby byl v každém uzlu uložen celkový počet jeho potomků (tedy včetně nepřímých). Správné řešení má složitost lineární vůči počtu uzlů stromu.

```
def count_children(tree) -> None:
    pass
```

9.p.4 [treezip] Třídy IntTree, StrTree a TupleTree reprezentují postupně stromy, které mají v uzlech uložená celá čísla (int), řetězce (str) a dvojice číslo + řetězec.

```
class IntTree:
    def __init__(self, value: int):
        self.value = value
        self.left: IntTree | None = None
        self.right: IntTree | None = None

class StrTree:
    def __init__(self, value: str):
        self.value = value
        self.left: StrTree | None = None
        self.right: StrTree | None = None

class TupleTree:
    def __init__(self, value: tuple[int, str]):
        self.value = value
        self.left: TupleTree | None = None
        self.right: TupleTree | None = None
```

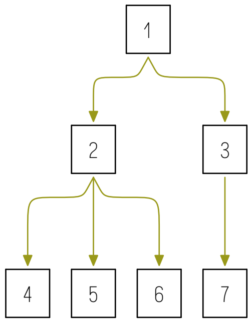
Napište (čistou) funkci, která obdrží jednu instanci IntTree a jednu instanci StrTree a vrátí nový strom typu TupleTree, který vznikne takto:

- uzel ve výstupním stromě bude přítomen, existuje-li odpovídající uzel v obou vstupních stromech,
- hodnota uzlu vznikne jako dvojice hodnot uložených v odpovídajících uzlech vstupních stromů.

Očekávaná složitost řešení je lineární vůči součtu počtu uzlů v obou stromech.

```
def treezip(it, st):
    pass
```

9.p.5 [mktree] Uvažujme neprázdný strom s očíslovanými vrcholy (kořen má vždy číslo 1):



Tento strom zakódujeme do slovníku takto:

```
TreeDict = dict[int, list[int]]

def example_tree() -> TreeDict:
    return {1: [2, 3],
            2: [4, 5, 6],
            3: [7],
            4: [], 5: [], 6: [], 7: []}
```

Klíče tohoto slovníku jsou čísla vrcholů a hodnoty jsou seznamy čísel jejich přímých potomků. Nejprve napište predikát, který ověří, že se jedná o korektně zadaný strom, tedy:

1. obsahuje kořen (uzel číslo 1),
2. každý vrchol se v seznamech potomků objevuje právě jednou, s výjimkou kořene, který se zde neobjevuje vůbec,
3. žádný uzel není svým vlastním (přímým) potomkem.

```
def is_tree(tree: TreeDict) -> bool:
    pass
```

Dále napište čistou funkci make_tree, která ze zadaného „slovníkového“ stromu tree vytvoří instanci třídy Tree tak, aby reprezentovala stejný strom. Vstupní podmínkou je, že tree je korektní strom, tzn. platí is_tree(tree).

```
class Tree:
    def __init__(self, value: int, children: list["Tree"]):
        self.value: int = value
        self.children = children

def make_tree(tree: TreeDict) -> Tree:
    pass
```

9.p.6 [prune] Pro účely tohoto cvičení musíme trochu pozměnit zápis stromu do tříd. Protože budeme strom měnit „na místě“, musí být prázdný i neprázdný strom reprezentován stejným typem. Proto si jej rozdělíme na třídy Node a Tree, které budou hrát podobnou roli jako jejich protějšky v zřetězeném seznamu. Tyto třídy nijak nemodifikujte.

```
class Node:
    def __init__(self, value: int,
                 left: 'Node | None',
                 right: 'Node | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: int) -> Node:
    return Node(value, None, None)

class Tree:
    def __init__(self, root: Node | None):
        self.root = root
```

Napište proceduru, která na vstupu dostane instanci výše popsaného stromu tree a množinu celých čísel keep a ze stromu tree odstraní všechny vrcholy (uzly), kterých hodnota v množině keep chybí. Spolu s vrcholem odstraňte i celý podstrom, který v něm začíná. Správné řešení má složitost lineární vůči počtu uzlů původního stromu.

```
def prune(tree: Tree, keep: set[int]) -> None:
    pass
```

9.r: Řešené úlohy

9.r.1 [treesum] Uvažujme ternární stromy, které mají v uzlech uložena celá čísla:

```
class Tree:
    def __init__(self, value: int,
                 first: 'Tree | None',
                 second: 'Tree | None',
                 third: 'Tree | None'):
        self.value = value
        self.first = first
        self.second = second
        self.third = third

def leaf(value: int) -> Tree:
    return Tree(value, None, None, None)
```

Napište čistou funkci, která na vstupu dostane instanci výše popsaného stromu a vrátí součet všech hodnot ve všech jeho uzlech.

```
def sum_tree(tree) -> int:
    pass
```

9.r.3 [heap]

```
class Tree:
    def __init__(self, key: int,
                 left: 'Tree | None',
                 right: 'Tree | None'):
        self.key = key
        self.left = left
        self.right = right

def leaf(key: int) -> Tree:
    return Tree(key, None, None)
```

Binární halda je binární strom, který má dvě speciální vlastnosti uvedené níže. V tomto příkladu budeme kontrolovat pouze tu druhou, totiž vlastnost haldy:

1. každé patro je plné (s možnou výjimkou posledního),
2. hodnota každého uzlu je větší nebo rovna hodnotě libovolného jeho potomka.

Predikát is_heap rozhodne, splňuje-li vstupní strom tuto druhou vlastnost.

```
def is_heap(tree) -> bool:
    pass
```

9.r.4 [avl]

```
class Tree:
    def __init__(self, left: 'Tree | None',
                 right: 'Tree | None') -> None:
        self.left = left
        self.right = right

def leaf() -> Tree:
    return Tree(None, None)
```

AVL strom je binární strom, který:

1. je vyhledávací, tzn. splňuje vlastnost popsanou v ukázce d3_lookup, a zároveň
2. pro každý jeho uzel platí abs(l_height - r_height) ≤ 1, kde l_height a r_height jsou výšky levého a pravého podstromu daného uzlu.

Napište predikát, který ověří, že vstupní strom má tuto druhou vlastnost (je-li zároveň stromem vyhledávacím ověřovat nemusíte). Pokuste se vlastnost ověřit jediným průchodem stromu (tedy každý uzel navštivte pouze jednou – naivní řešení, kdy opakovaně počítáte výšky průchodem podstromů není příliš uspokojivé).

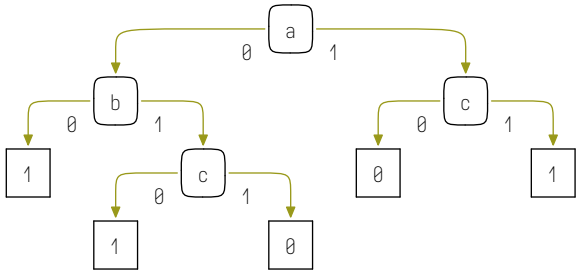
```
def is_avl(tree) -> bool:
    pass
```

9.r.5 [bdd] Binární rozhodovací diagram (anglicky „binary decision di-

agram“, zkráceně „BDD“) je datová struktura, která umožňuje efektivně kódovat formule výrokové logiky, například:

$$\phi = a \vee (b \wedge c) \Rightarrow (a \wedge c)$$

Protože budeme takto zapsané funkce pouze vyhodnocovat, můžeme se na BDD dívat jako na binární strom,[19] který má ve vnitřních uzlech názvy proměnných a v listech pravdivostní hodnoty (budeme je reprezentovat hodnotami 0 a 1). BDD pro výše uvedenou formuli může vypadat například takto:



BDD vyhodnotíme tak, že začneme v kořenu, a v každém uzlu se rozhodneme podle pravdivosti proměnné, kterou je tento uzel označený: je-li pravdivá, pokračujeme doprava, jinak doleva. Výsledkem je hodnota, kterou najdeme v takto nalezeném listu. Srovnejte tabulku pravdivostních hodnot:

a	b	c	b ∧ c	a ∨ (b ∧ c)	a ∧ c	φ
0	0	0	0	0	0	1
0	0	1	0	0	0	1
0	1	0	0	0	0	1
0	1	1	1	1	0	0
1	0	0	0	1	0	0
1	0	1	0	1	1	1
1	1	0	0	1	0	0
1	1	1	1	1	1	1

```
class BDD:
    def __init__(self, val: str, left: 'BDD | None',
                 right: 'BDD | None') -> None:
        self.val = val
        self.left = left
        self.right = right
```

Naprogramujte čistou funkci, která vyhodnotí zadané BDD pro dané ohodnocení proměnných. Předpokládejte, že každý vnitřní uzel má oba potomky. Hodnoty proměnných jsou zadané množinou `true_vars`: je-li název proměnné v této množině, proměnná je pravdivá, jinak nikoliv. V listech jsou uloženy řetězce `"0"` (výsledek je `False`) nebo `"1"` (výsledek je `True`).

```
def evaluate_bdd(bdd, true_vars: set[str]) -> bool:
    pass
```

[19] V praxi se obvykle používají tzv. redukované BDD, kde jsou některé podstromy vhodně sloučeny, a to tak, aby se nezměnil výsledek vyhodnocení. Na samotný proces vyhodnocování tato úprava nemá žádný vliv.

9.r.6 [average]

```
class Tree:
    def __init__(self, left: 'Tree | None',
                 right: 'Tree | None') -> None:
        self.left = left
        self.right = right

def leaf() -> Tree:
    return Tree(None, None)
```

Napište čistou funkci, která pro vstupní binární strom spočítá průměrnou délku větve (cesty od kořene k listu). K řešení úlohy je postačující projít strom jen jednou.

```
def average_branch_len(tree) -> float:
    pass
```

Demonstrační příklady:

1. cycle – hledání cyklu s rozhodováním
2. minmax – jak vyhrát tic-tac-toe
3. sat – splnitelnost formulí výrokové logiky
4. 8puzzle † – puzzle s osmi kameny a devíti políčky

Elementární příklady:

1. subsets – generování všech podmnožin
2. flatten – rekurze na vnořených seznamech

Přípravy:

1. squaresum – rozklad čísla na součet čtverců
2. permutations – výpočet všech permutací seznamu
3. chain – elfí číselné řetězy
4. digits – generování čísel s daným ciferným součtem
5. partitions – generování všech rozkladů zadané množiny
6. circle – nejdelší šestnáctkový kruh

Rozšířené úlohy:

1. subseq – neklesající podposloupnosti
2. ipv4fix – oprava rozbité IPv4 adresy
3. weighted – hledání slov splňujících podmínku
4. nested † – řazení vnořeného seznamu bez změny struktury
5. subsetsum – známý NP-těžký problém
6. dnfsat – splnitelnost formulí v DNF

10.1: Programovací jazyk

V této kapitole se jazyk nemění.

10.d: Demonstrace (ukázky)

10.d.1 [cycle] V tomto příkladu se vrátíme k problému 09/cycle.py z minulého týdne. Připomeňme si základní strukturu:

- vstupem je seznam čísel, a počáteční index,
- v každém kroku výpočtu se číslo na aktuálním indexu k tomuto indexu přičte, čím vznikne nový index.

Tento proces se může, ale nemusí, zacyklit. Ve verzi z minulého týdne jsme pouze rozhodovali, která možnost nastane. Tentokrát bude problém postaven trochu jinak: všechna čísla v seznamu budou kladná, a v každém kroku máme možnost rozhodnout se, budeme-li číslo přičítat nebo odečítat.

Naším cílem bude zjistit, nejen existuje-li nějaký cyklus (sekvence rozhodnutí vlevo/vpravo taková, že ji lze donekonečna opakovat), ale navíc existuje-li takový, že navštíví všechny platné indexy. Není těžké si domyslet, že na počátečním indexu vůbec nezáleží, protože hledaný cyklus prochází každým indexem, a tedy jej můžeme z formulace problému vypustit.

Problém budeme řešit jak jinak než rekurzí. Hlavní část řešení zastřešuje predikát solve_rec, s následovnými parametry:

- numbers je zadaná „hrací plocha“,
- index je současně zkoumaný index,
- goal je index, ke kterému chceme dojít, a konečně
- to_visit je množina dosud nenavštívených indexů.

Predikát odpovídá na otázku: lze se z indexu index dostat na index goal tak, že každý index z to_visit se použije právě jednou? Zřejmě si dovedete představit, že jakmile vyřešíme tento problém, dokážeme již původní otázku na přítomnost cyklu lehce vyjádřit jako jeho instanci (chceme se dostat z nějakého indexu na tentýž index a použít k tomu právě všechny platné indexy).

```
def solve_rec(numbers: list[int], index: int, goal: int,
              to_visit: set[int]) -> bool:
```

Vyřešíme nejprve jednoduché případy. Vypadneme-li z rozsahu indexů, jistě se nám už k indexu goal nepodaří dojít a odpovídáme zamítavě.

```
    if index < 0 or index >= len(numbers):
        return False
```

V případě, že jsme na indexu goal a množina to_visit je prázdná, je zřejmé, že odpověď je True (jsme tam, kde máme být, a máme se tam dostat bez použití jakéhokoliv jiného indexu).

```
    if index == goal and not to_visit:
        return True
```

Konečně případ, kdy se nacházíme na indexu, který není cílem, a zároveň jej již nelze použít (není přítomen v to_visit): zamítáme. Speciálním případem této podmínky je i stav, kdy je množina to_visit prázdná.

```
    if index not in to_visit:
        return False
```

V ostatních případech nelze přímo rozhodnout. Jednodušší instance sestavíme tak, že aktuální index odebereme z to_visit a posuneme se buď doleva (index_left) nebo doprava (index_right). Do goal vede přípustná cesta tehdy, když taková existuje v alespoň jedné z takto sestrojených instancí. Instance jsou jednodušší, protože množina to_visit se zmenšila, a případ, kdy je prázdná je vždy jednoduchý (viz výše).

```
    remaining = to_visit - {index}
    index_left = index - numbers[index]
    index_right = index + numbers[index]

    return (solve_rec(numbers, index_left, goal, remaining) or
            solve_rec(numbers, index_right, goal, remaining))
```

Jak již bylo naznačeno, původní problém již lehce zapíšeme jako instanci problému, který řeší predikát solve_rec.

```
def solve(numbers: list[int]) -> bool:
    indices = [i for i in range(len(numbers))]
    return solve_rec(numbers, 0, 0, set(indices))
```

Řešení jako obvykle otestujeme na jednoduchých příkladech.

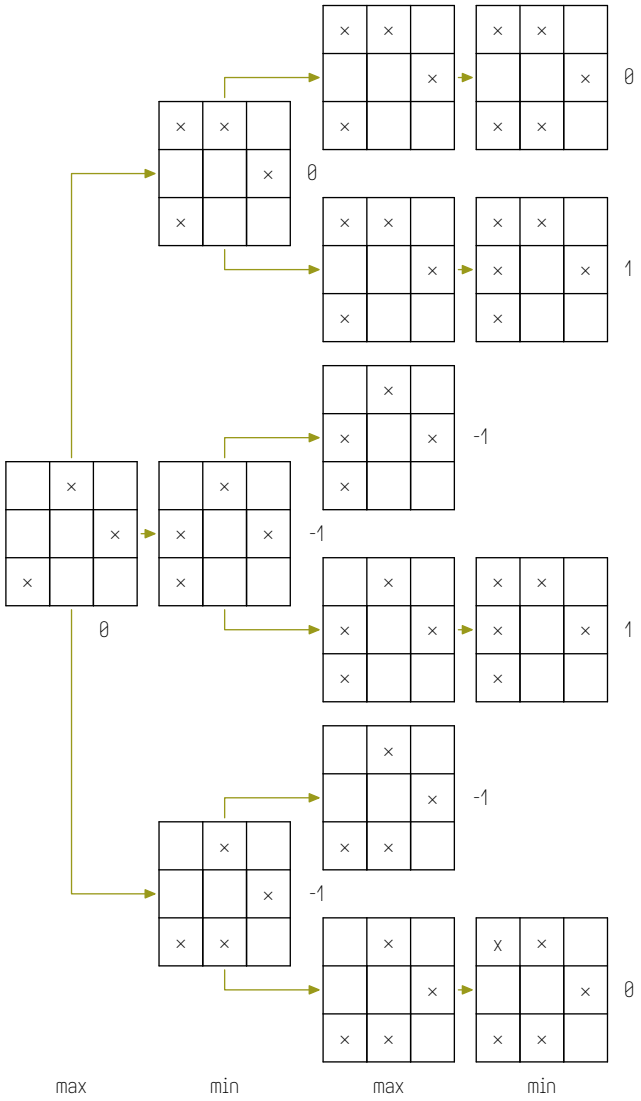
```
def main() -> None:  # demo
    assert not solve([1])
    assert solve([1, 1])
    assert not solve([1, 0, 1])
    assert not solve([1, 1, 1])
    assert solve([1, 1, 2])
    assert not solve([1, 2, 1])
    assert solve([1, 1, 1, 3])
    assert solve([3, 1, 1, 1])
    assert solve([2, 1, 2, 2, 1])
    assert not solve([2, 2, 2, 2, 2])
    assert not solve([2, 2, 1, 2])
```

10.d.2 [minmax] V tomto příkladu se vrátíme k „minmax“ stromům z 09/minmax.py a zejména k jejich praktické aplikaci. Strom už nicméně nebudeme reprezentovat explicitně jako datovou strukturu, budeme jej vždy konstruovat „podle potřeby“ lokálně, v rámci rekurzivního řešení nějakého problému.

Problém, který budeme řešit je jak vyhrát (nebo aspoň neprohrát) piškvorky na ploše 3×3 (v angličtině známé jako „tic-tac-toe“). Tato hra je dost jednoduchá na to, abychom dokázali řešení najít i celkem naivně.

Jak si jistě pamatujete z minula, v „minmax“ stromu se střídají „min“ uzly a „max“ uzly: my teď do každého uzlu umístíme hrací plán: do uzlů typu „max“ takový, kde jsme na tahu my (hráč s křížky) a do uzlů typu „min“ pak ty, kde je na tahu hráč s kolečky (náš protivník). Zbývá nám ještě ohodnotit listy, které budou reprezentovat ukončené hry (některý hráč vyhrál, nebo je plocha již zaplněná a došlo tedy k remíze). To provedeme tak,

že remízu ohodnotíme nulou (neutrální výsledek), výhru křížků ohodnotíme $+1$ (pozitivní) a výhru koleček -1 (negativní) výsledek. Takový strom můžeme zřejmě nakreslit z libovolné herní pozice. Například (poslední tah je vždy vybarven, u vnitřních uzlů uvádíme jejich vypočtené hodnoty):



Jak nám takový strom pomůže vyhrát? Střídající se minima a maxima v jednotlivých patrech stromu odpovídají nejlepším možným tahům příslušného hráče: dojdeme-li do listu s hodnotou -1, znamená to, že kolečka vyhrála (tomuto hráči budeme odteď říkat „min“). Cílem hráče „min“ je tedy dostat se do listu ohodnoceného -1. Naopak, hráč s křížky (bude se jmenovat „max“) se pokouší dostat do listu ohodnoceného +1. Toto odpovídá elementárním případům rekurze.

Stojí-li hráč před posledním rozhodnutím (uvažme třeba nejspodnější případ z druhého sloupce obrázku, kde se hráč „min“ rozhoduje mezi dvěma políčky), vybere si tu z nich, která povede k výhře (je-li to možné), případně k remíze. Je vidět, že to odpovídá právě následníkovi s nejmenší hodnotou (pro hráče „min“, u hráče „max“ je tomu přesně naopak). Totéž samozřejmě platí i o patro výš, a tak dále, až ke kořeni.

Abychom mohli takový pomyslný „minmax“ strom prohledat, musíme umět reprezentovat jeho jednotlivé vrcholy: ty neobsahují nic jiného, než herní pozice. Ty budeme reprezentovat dvourozměrným seznamem čísel. Prázdná políčka budou mít hodnotu 0, hráči pak budou používat „svoji“ hodnotu: hráč „min“ dostane -1 a hráč „max“ +1. Jednotlivý tah pak budeme reprezentovat jako dvojici (x, y) souřadnic, každou z rozsahu $\langle 0, 2 \rangle$.

```
Plan = list[list[int]]
Move = tuple[int, int]
```

První pomocnou funkci, kterou si zadefinujeme, bude čistá funkce put, která dostane plán, souřadnice tahu, a hráče, a vytvoří nový plán takový, kde zadaný hráč obsadil zadané políčko. Vstupní podmínkou je, že políčko bylo prázdné.

```
def put(plan: Plan, where: Move, player: int) -> Plan:
    x, y = where
    assert plan[y][x] == 0
    plan = [row.copy() for row in plan]
    plan[y][x] = player
    return plan
```

Čistá funkce list_empty vytvoří seznam všech přípustných tahů (tzn. souřadnice všech prázdných políček v předané hrací ploše).

```
def list_empty(plan: Plan) -> list[Move]:
    return [(x, y)
            for x in range(3)
            for y in range(3)
            if not plan[y][x]]
```

Další (opět čistá) funkce bude line, která na vstupu dostane počáteční souřadnice (parametry x a y) a „směr“ (parametry dx a dy, které udávají požadovaný přírůstek na dané souřadnici). Z těchto spočítá, je-li celá takto popsaná „čára“ obsazena týmž hráčem. Pokud ano, vrátí identifikátor hráče, jinak nulu. Tato situace zřejmě odpovídá (nějaké) vítězné pozici.

```
def line(plan: Plan, x: int, y: int, dx: int, dy: int) -> int:
    player = plan[y][x]
    for n in range(1, 3):
        if plan[y + dy * n][x + dx * n] != player:
            return 0
    return player
```

Následuje pomocná funkce, která vrátí svůj první nenulový parametr,[20] existuje-li takový (jinak vrátí nulu).

```
def either(a: int, b: int) -> int:
    return a if a else b
```

Poslední pomocnou funkcí je winner, která rozhodne, zda některý hráč již vyhrál, a pokud ano, který. Určitě to není nejkrásnější funkce v historii funkcí, ale účel plní a je relativně kompaktní (a to je občas také žádoucí).

```
def winner(plan: Plan) -> int:
    player = 0
    for v in range(3):
        player = either(player, line(plan, v, 0, 0, 1))
        player = either(player, line(plan, 0, v, 1, 0))
    player = either(player, line(plan, 0, 0, 1, 1))
    player = either(player, line(plan, 2, 0, -1, 1))
    return player
```

Tím jsme vybaveni k implementaci samotného rekurzivního prohledávání „minmax“ stromu hry tic-tac-toe.

```
def decide(plan: Plan, player: int) -> tuple[int, Move | None]:
```

Jak jsme již zvyklí, vyřešíme nejprve jednoduché případy, totiž ty, kdy se nacházíme v listu. Listy jsou dvojího typu: některý hráč vyhrál, nebo je pole již plné a nastala remíza.

```
    won = winner(plan)
    empty = list_empty(plan)
    moves = []
    if won or len(empty) == 0:
        return (won, None)
```

Nejsme-li v listu, musíme prohledat následníky. Následník se od aktuálního vrcholu odlišuje tím, že hráč, který je na tahu, do některého volného pole umístí svůj symbol. Následníků je právě tolik, kolik je volných políček. Nesmíme zapomenout, že na tahu bude v rekurzivním volání opačný hráč, než je ten současný. Krom skóre, které danému uzlu přisoudí rekurzivní volání si zapamatujeme i tah, který do tohoto uzlu vedl.

```
    for move in empty:
        score, _ = decide(put(plan, move, player), -player)
        moves.append((score, move))
```

[20] V Pythonu by bylo lze stejného efektu docílit použitím operátoru or, nicméně se jedná o docela atypickou vlastnost jazyka, proto se zde takovému použití raději vyhneme.

Nyní již máme výsledky pro všechny následníky: vybereme ten nejlepší možný – hráč „max“ ten maximální, zatímco hráč „min“ ten minimální. Všimněte si, že vybíráme ze seznamu, který obsahuje dvojice (skóre, tah). Je-li několik ekvivalentních možností (mají stejné skóre), hráč „min“ vybere ten s nejmenšími a hráč „max“ ten s největšími souřadnicemi. Protože na konkrétní volbě nezáleží, můžeme si tuto zápisovou zkratku na tomto místě dovolit.

```
    return max(moves) if player > 0 else min(moves)
```

Tím je hra tic-tac-toe vyřešena: máme algoritmus, který hraje „nejlépe, jak je to možné“ – může-li v nějaké pozici vynutit výhru, nebo alespoň remízu, decide vybere právě takové tahy, aby ji skutečně vynutil.

Výjimečně si krom jednoduchých automatických testů přidáme i možnost hry vypisovat na obrazovku. Pomocná procedura draw přidá do rozpracovaného obrázku hry další tah.

```
def draw(plan: Plan, game_rows: list[list[str]]) -> None:
    for i in range(min(len(plan), len(game_rows))):
        game_row = game_rows[i]
        game_row.append(' │ ' if game_row else ' ')
        for cell in plan[i]:
            game_row.append('×' if cell > 0 else
                            '○' if cell < 0 else '_')
```

A konečně procedura play nechá hrát strategii decide samu proti sobě a výsledek nakreslí. Parametry jsou počáteční pozice a hráč, který je na tahu. V parametru game si funkce udržuje „obrázek“ hry, který na konci vypíše. Všimněte si, že tato funkce s výhodou využívá koncové rekurze.

```
def play(plan: Plan, player: int, game: list[list[str]]) -> None:
    draw(plan, game)
    _, move = decide(plan, player)

    if move is None:
        for row in game:
            for seg in row:
                print(end=seg)
            print()
        print()
    else:
        plan = put(plan, move, player)
        play(plan, -player, game)

def main() -> None:  # demo
```

Nejprve si vykreslíme několik jednoduchých her. Zkuste si hry upravit a rozmyslete si, proč decide hraje zrovna takto.

```
    play([[-1, +0, +0],
          [+0, +1, +0],
          [+0, +1, -1]], 1,
         [[], [], []])
    play([[-1, +0, +1],
          [+0, +0, +0],
          [+0, +1, -1]], 1,
         [[], [], []])
    play([[-1, +0, +1],
          [+0, +0, +0],
          [-1, +1, +0]], 1,
         [[], [], []])
```

První dva testy odpovídají obrázku ze začátku příkladu. Ty zbývající nejsou příliš intuitivní (proto jsme si nechali hry vykreslovat), nicméně odpovídají konkrétním volbám, které algoritmus provede.

```
    assert decide([[+0, +1, -1],
                   [+0, -1, +1],
                   [+1, +1, -1]], -1) == (-1, (0, 0))

    assert decide([[+0, +1, -1],
                   [+0, -1, +1],
                   [+1, +0, -1]], +1) == (0, (0, 0))

    assert decide([[-1, +0, +1],
                   [+0, +0, +0],
                   [+0, +1, -1]], 1) == (1, (1, 1))

    assert decide([[-1, +0, +1],
                   [+0, +0, +0],
                   [-1, +1, +0]], 1) == (0, (0, 1))
```

10.d.3 **[sat]** Výrokovou logiku jistě znáte, například z předmětu MZI. To co možná nevíte je, že každou formuli výrokové logiky lze přepsat do obzvláště jednoduchého tvaru: takzvané **konjunktivní normální formy**. V této formě se formule skládá ze závorek (klauzulí), které jsou spojeny konjunkcí. V každé závorce je pak disjunkce **literálů**: proměnných, nebo jejich negací. Například:

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee c)$$

To, jak se formule do této podoby převede nás teď nemusí zajímat (někdy později v průběhu studia to nejspíš ještě zjistíte), co je důležité je, že nám stačí pracovat s formulemi tohoto tvaru.

Jak takové formule reprezentovat v programu? Vybudujeme si vhodné typy odspodu, tzn. od samotných proměnných, které budeme reprezentovat písmeny:

```
Variable = str
```

Literál budeme reprezentovat dvojicí: krom proměnné si potřebujeme pamatovat, je-li literál **pozitivní** (pozitivní je, když proměnné nepředchází negace): na toto nám stačí hodnota typu bool.

```
Literal = tuple[Variable, bool]
```

Dalším útvarem je klauzule, tedy disjunkce nějakého počtu literálů:

```
Clause = list[Literal]
```

A konečně samotná formule, která je konjunkcí klauzulí:

```
Formula = list[Clause]
```

Zbývá poslední typ, který budeme potřebovat, a tím je **valuace**: přiřazení pravdivostních hodnot jednotlivým proměnným.

```
Valuation = dict[str, bool]
```

Problém, který budeme řešit se jmenuje **splnitelnost**: bude nás zajímat, existuje-li valuace taková, že se zadaná formule vyhodnotí na True. Nejprve si ale naprogramujeme jednodušší funkci: **vyhodnocení** formule, kterého vstupem je nějaká formule a valuace proměnných, a výsledkem je pravdivostní hodnota. Budeme navíc ale uvažovat i případ, kdy valuace není úplná, tzn. některé proměnné nemají pravdivostní hodnotu určenu. V takovém případě můžou nastat tři případy:

1. formule je pravdivá bez ohledu na nepřiřazené proměnné (v každé klauzuli je alespoň jeden splněný literál),
2. formule je nepravdivá: existuje klauzule, která obsahuje pouze přiřazené proměnné a zároveň není splněna,
3. o pravdivosti nelze rozhodnout: některou klauzuli se nepovedlo splnit, ale tato klauzule obsahuje nerozhodnutou proměnnou.

Funkce evaluate bude v těchto situacích vracet postupně True (určitě splněno), False (určitě nesplněno) a None (nevíme).

```
def evaluate(phi: Formula, valuation: Valuation) -> bool | None:

    undecided_clause = False
```

Formuli budeme vyhodnocovat po jednotlivých klauzulích. Výsledek pro každou z nich může být, podobně jako pro celou formuli, „splněna“, „nesplněna“ nebo „nelze říct“.

```
    for clause in phi:
        satisfied = False
        undecided_literal = False

        for variable, positive in clause:
            if variable not in valuation:
                undecided_literal = True
            elif valuation[variable] == positive:
```

```
            satisfied = True
            break
```

V případě, že se klauzuli nepovedlo splnit, musíme rozlišit dva případy: jestli tato obsahovala nerozhodnutý literál (příslušná proměnná nemá přiřazenu pravdivostní hodnotu), výsledek pro klauzuli je „nelze říct“ a pokračujeme ve vyhodnocování (může se totiž ještě objevit klauzule, která formuli rozhodne v záporu). Jsou-li ale všechny proměnné v klauzuli přiřazené, víme, že formule jako celek se vyhodnotí na False a tento výsledek můžeme rovnou vrátit.

```
        if not satisfied:
            if undecided_literal:
                undecided_clause = True
            else:
                return False
```

Žádná klauzule se nevyhodnotila na False, pro formuli jako celek zbývají tedy pouze možnosti „splněna“ nebo „nelze říct“. Druhá možnost nastane v případě, kdy se nám některou klauzuli nepodařilo rozhodnout.

```
    return None if undecided_clause else True
```

Dále budeme potřebovat (čistou) funkci, která nám z formule získá množinu všech proměnných, které se ve formuli objevují.

```
def variables(phi: Formula) -> set[str]:
    var_set: set[str] = set()
    for clause in phi:
        for var, _ in clause:
            var_set.add(var)
    return var_set
```

Poslední pomocnou funkcí (opět čistou) bude extend, která do valuace přidá novou proměnnou. Vstupní podmínkou je, že tato proměnná ještě ve valuaci hodnotu přiřazenou nemá.

```
def extend(val: Valuation, var: str, value: bool) -> Valuation:
    assert var not in val
    new = val.copy()
    new[var] = value
    return new
```

Nyní již můžeme přistoupit k samotnému řešení problému: možná si pamatujete **pravdivostní tabulky** – jejich konstrukcí lze jednoduše zjistit, je-li formule splnitelná. K tomu nám totiž stačí nalézt splňující přiřazení (tedy takové, při kterém se formule vyhodnotí na True). Pro $\phi = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee c)$ vypadá pravdivostní tabulka takto:

a	b	c	ϕ
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Potřebujeme tedy algoritmus, který takovou tabulku sestrojí a najde první řádek, kde formuli jako celku náleží hodnota 1 (neboli True). Jak již jistě tušíte, použijeme rekurzi. Budeme si přitom předávat dvě pomocné hodnoty: seznam proměnných, jejichž pravdivost ještě potřebujeme rozhodnout, a částečnou valuaci, kterou budeme postupně budovat. Význam predikátu satisfiable_rec je „lze přiřazení valuation doplnit tak, aby formuli splnilo?“

```
def satisfiable_rec(phi: Formula, to_decide: list[str],
                    valuation: Valuation) -> bool:
```

Jako obvykle, nejprve vyřešíme jednoduchý případ, totiž ten, kdy již formuli dokážeme rozhodnout. Tento případ zejména nastane, je-li již přiřazení valuation kompletní a tedy seznam to_decide prázdný.

Může se ale stát, že formuli dokážeme rozhodnout i přesto, že jsme dosud nepřiřadili pravdivostní hodnoty všem proměnným. Toto odpovídá třeba hned první dvojici řádků tabulky výše: na hodnotě c vůbec nezáleží, a při vyhodnocování druhého sloupce prvního řádku zjistíme, že „tudy cesta nevede“: můžeme rovnou skočit na řádek třetí.

```
    result = evaluate(phi, valuation)
    if result is not None:
        return result
```

V případě, že zatím rozhodnout nelze, z to_decide vybereme proměnnou, které následně přisoudíme pravdivostní hodnotu.

```
    var = to_decide.pop()
```

Vybrané proměnné můžeme přisoudit hodnotu True nebo False, čím dostaneme dvě (striktně úplnější) valuace: nazveme je val_true a val_false.

```
    val_true = extend(valuation, var, True)
    val_false = extend(valuation, var, False)
```

Konečně přiřazení valuation lze na splňující přiřazení doplnit právě tehdy, když lze takto doplnit alespoň jedno z rozšířených přiřazení val_true nebo val_false. Zároveň je zřejmé, že instance, které řešíme rekurzí jsou jednodušší: zbývá o jednu nerozhodnutou proměnnou méně.

```
    return (satisfiable_rec(phi, to_decide.copy(), val_true) or
            satisfiable_rec(phi, to_decide, val_false))
```

Není již těžké si uvědomit, že formule je splnitelná právě když lze prázdnou valuaci rozšířit na valuaci splňující:

```
def satisfiable(phi: Formula) -> bool:
    return satisfiable_rec(phi, list(variables(phi)), {})
```

Tím jsme hotovi, implementaci si ještě na několika formulích otestujeme. Aby se nám formule trochu lépe četly, zadefinujeme si pro jejich vytváření dvě jednoduché pomocné funkce (positive a negative).

```
def positive(var: str) -> Literal:
    return (var, True)

def negative(var: str) -> Literal:
    return (var, False)

def main() -> None:  # demo
    phi_1 = [[positive('a'), positive('b')],
             [negative('a'), positive('c')],
             [negative('a'), negative('b'), positive('c')]]
    assert satisfiable(phi_1)

    phi_2 = [[positive('a')], [negative('a')]]
    assert not satisfiable(phi_2)

    phi_3 = [[positive('a'), positive('b')],
             [negative('a'), positive('b')],
             [positive('a'), negative('b')],
             [negative('a'), negative('b')]]
    assert not satisfiable(phi_3)

    phi_4 = [[positive('a'), positive('b'), positive('c')],
             [negative('a'), positive('b'), negative('c')],
             [positive('a'), negative('b'), negative('c')],
             [positive('a'), negative('b'), positive('c')],
             [negative('a'), negative('b'), negative('c')],
             [negative('a'), positive('c')],
             [positive('a'), negative('c')]]
    assert not satisfiable(phi_4)

    phi_5 = [[positive('a'), positive('b'), positive('c')],
             [negative('a'), positive('b'), negative('c')],
             [positive('a'), negative('b'), positive('c')],
             [negative('a'), negative('b'), negative('c')]]
    assert satisfiable(phi_5)
```

10.d.4 [8puzzle] † V této ukázce přidáme oproti předchozím několik novinek. Nejprve si ale představme problém, který budeme řešit. Možná znáte hru „15 puzzle“ – hraje se s 15 posuvnými kameny v rámu o rozměru 4×4 – jedno místo tedy zůstává volné a umožňuje kameny posouvat. My budeme řešit o něco menší variantu této hry: 8 kamenů v rámečku 3×3. Na kamenech může být třeba obrázek, ale tradiční varianta, kterou budeme používat i my, má kameny očíslované od 1 do 8. Vyřešený rébus má tedy tuto podobu:

	1	2
3	4	5
6	7	8

Hra se hraje tak, že dostaneme pole nějak pomíchané a snažíme se sestavit jej do podoby nakreslené výše. K dispozici máme vždy několik tahů – můžeme si vybrat, který sousední kámen do prázdného políčka přemístit. Protože hra je ve své klasické podobě realizovaná fyzicky, přesouvat můžeme kameny pouze ve 4 směrech: nahoru, dolů, doleva a doprava. Příklad krátké hry:

1	4	2
3	5	
6	7	8

1	4	2
3		5
6	7	8

1		2
3	4	5
6	7	8

	1	2
3	4	5
6	7	8

Každý přípustný počáteční stav hry (konfigurace) má mnoho řešení: my budeme odpovídat na otázku, jak dlouhé je to nejkratší[21] (s nejmenším počtem kroků). Nejprve si zadefinujeme několik užitečných typů a pomocných funkcí. Uspořádání rámečku (krabičky) budeme reprezentovat lineárním seznamem, a to tak, že vyřešená hra bude mít tvar `[0, 1, 2, 3, 4, 5, 6, 7, 8]`: hrací pole budeme odečítat ze seznamu po řádcích, vždy zleva doprava, prázdné políčko reprezentujeme nulou. Souřadnice políčka budou dvojice čísel z rozsahu $\langle 0, 2\rangle$, přičemž $(0, 0)$ je levý horní roh.

```
Box = list[int]
Position = tuple[int, int]
```

Tahy budeme reprezentovat jako **pohyb volného políčka** (rozmyslete si, že se jedná o ekvivalentní, ale úspornější popis, než si pamatovat který kámen tahal kterým směrem). Tento pohyb budeme zapisovat jako `(dx, dy)` – posuv ve směru x a ve směru y samostatně. Po směru hodinových ručiček jsou to postupně dvojice $(1, 0), (0, 1), (-1, 0), (0, -1)$.

```
Move = tuple[int, int]
```

Dále budeme potřebovat převádět mezi indexem v seznamu `Box` a souřadnicemi daného políčka. K tomu slouží následující dvě (čisté) funkce.

```
def to_index(position: Position) -> int:
    x, y = position
    return y * 3 + x

def to_position(index: int) -> Position:
    return (index % 3, index // 3)
```

Dále si zadefinujeme (opět čistou) funkci, která nám pro daný tah vrátí ten opačný (když provedeme tah `m` a poté `opposite(m)`, nestane se nic – ujistěte se, že rozumíte, proč tomu tak je).

```
def opposite(move: Move) -> Move:
    shift_x, shift_y = move
    return (-shift_x, -shift_y)
```

Základ herní mechaniky realizuje procedura `move_blank`, která v daném rozložení kamenů posune prázdné místo ve směru daném parametrem `move`. Pohyb realizuje výměnou hodnot na odpovídajících pozicích v seznamu, který hru reprezentuje.

```
def move_blank(box: Box, move: Move) -> None:
    shift_x, shift_y = move
    blank_idx = box.index(0)
    blank_pos = to_position(blank_idx)
    blank_x, blank_y = blank_pos
    other_pos = (blank_x + shift_x, blank_y + shift_y)
    other_idx = to_index(other_pos)
    box[blank_idx], box[other_idx] = box[other_idx], box[blank_idx]
```

Dále nás bude zajímat, je-li nějaký tah při daném rozložení kamenů přípustný, tzn. nepokusíme se přesunout neexistující kámen (umístěný mimo hrací plochu) do prázdného místa, které je zrovna na některém kraji. Tuto kontrolu realizuje predikát `admissible`.

```
def admissible(box: Box, move: Move) -> bool:
    move_x, move_y = move
    blank_x, blank_y = to_position(box.index(0))
    return (0 <= move_x + blank_x < 3 and
            0 <= move_y + blank_y < 3)
```

Předposlední pomocná čistá funkce je `distance`, která nám řekne, kolikrát se daný kámen musí určitě posunout, aby se dostal na své správné místo. Protože kameny lze posouvat pouze v pravých úhlech, záleží pouze na počtu horizontálních a počtu vertikálních posunů samostatně. Uvažme například posuv ze souřadnic $(2, 2)$ na souřadnice $(1, 0)$ (šipky reprezentují směr pohybu). Je vidět, že určitě potřebujeme aspoň tři posuvy, co odpovídá naznačenému vzorci $|x_1 - x_2| + |y_1 - y_2|$ – v našem příkladě tedy $|2 - 1| + |2 - 0| = 1 + 2 = 3$. Přesun lze jistě realizovat i více kroky, nás ale bude zajímat minimum.

	×	←
		↑
		↑

	×	
	↑	←
		↑

	×	
	↑	
	↑	←

Toto číslo odpovídá tzv. Manhattanské metrice[22] (vzdálenosti) mezi současnou a koncovou pozicí daného kamene.

```
def distance(box: Box, tile: int) -> int:
    want_x, want_y = to_position(tile)
    now_x, now_y = to_position(box.index(tile))
    return abs(want_x - now_x) + abs(want_y - now_y)
```

Vyzbrojeni minimálním počtem kroků, které potřebujeme k přesunu daného kamene na své místo, se pokusíme odhadnout, kolik nejméně kroků potřebujeme k vyřešení celého rébusu. Tento odhad je naštěstí velmi jednoduchý: stačí si uvědomit, že přesunem jednoho kamene se ke své koncové pozici přiblíží **pouze tento kámen** a žádný jiný. Jistě se nám často stane, že kroků bude potřeba víc: to nám ale nebude vadit, důležité je pouze to, abychom měli dobrý spodní odhad.

```
def need_steps(box: Box) -> int:
    total = 0
    for tile in range(1, 9):
        total += distance(box, tile)
    return total
```

Tím jsou pomocné funkce vyřešeny a můžeme se pustit do samotného hledání nejkratšího řešení. Stejně jako v předchozích ukázkách, budeme používat rekurzi a backtracking, ale objeví se zde i slibované novinky.

1. Dosud jsme všechny prohledávací algoritmy realizovali jako čisté funkce. Prohledávací algoritmus pro „8 puzzle“ má ale **sdílený stav**: efektivní řešení tohoto rébusu vyžaduje, abychom sdíleli informace mezi jednotlivými podvýpočty. To nám umožní ty, o kterých z předchozího prohledávání víme, že nevedou k cíli, rychle zamítnout.
2. Protože beztak je výpočet realizován procedurou, nebudeme pro každý tah vytvářet novou (upravenou) kopii stavu hry: místo toho si budeme pamatovat pouze **sekvenci tahů** jako explicitní zásobník a hrací plochu budeme upravovat **in situ** (na místě). Ušetříme tak značné množství práce.

Sdílený stav zapouzdříme do **třídy**, která bude mít následovné atributy:

[21] V mnoha případech existuje víc než jedno nejkratší řešení, to na náš úkol ale nemá zásadní vliv, protože nás zajímá pouze jejich délka, kterou mají samozřejmě všechny společnou.

[22] Můžete si ji prostudovat online, pro pochopení řešení hry si ale vystačíte s informacemi zde uvedenými.

- best: délka dosud nalezeného nejlepšího řešení (k vyřešení hry s nejdelším optimálním řešením je potřeba 31 tahů[23] – toto číslo tedy použijeme jako počáteční horní odhad pro délku),
- found: nastavíme na True jakmile nalezneme libovolné řešení,
- moves: zmiňovaný zásobník tahů, které jsme provedli z počáteční konfigurace, a který nám umožní efektivně se ve výpočtu vracet,
- box: aktuálně zkoumaná herní pozice,
- visited: slovník,[24] ve kterém si budeme pamatovat již objevené herní pozice (konfigurace hrací plochy) a v kolika krocích jsme k nim z té počáteční došli (tento slovník nám umožní přeskočit velkou část redundantních podstromů).

```
class Solver:
    def __init__(self, initial: Box):
        self.best = 31
        self.found = False
        self.moves: list[Move] = []
        self.box = initial.copy()
        self.visited: dict[tuple[int, ...], int] = {}
```

Následující dvě metody realizují provedení jednoho tahu (apply) resp. jeho vrácení (backtrack). Všimněte si, že jsou to jediné dvě metody, které přímo modifikují jak aktuální hrací pole, tak zásobník tahů.

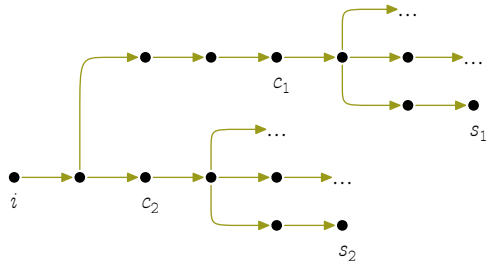
```
    def apply(self, move: Move) -> None:
        self.moves.append(move)
        move_blank(self.box, move)

    def backtrack(self) -> None:
        move_blank(self.box, opposite(self.moves.pop()))
```

Samotné rekurzivní hledání realizuje metoda-procedura search.

```
    def search(self) -> None:
```

Struktura rekurzivního řešení je stále zachována. Nejprve jednoduché (přímo řešitelné nebo nezajímavé) případy. Ten první jednoduchý případ je ale nového typu: nacházíme-li se v konfiguraci, kterou jsme již někdy v minulosti (v jiném podstromě) navštívili, zjistíme, kolik kroků jsme na to v minulosti potřebovali (jak hluboko ve stromě se nacházela).

Podstrom, který je na dané konfiguraci „zavěšen“ je totiž vždy stejný: má smysl jej prohledávat pouze v případě, že jsme tuto konfiguraci ještě nikdy nepotkali, nebo ji potkali pouze ve větší hloubce. V tom druhém případě si totiž celkovou délku cesty k řešení zkrátíme. Uvažme například tuto situaci (i je počáteční konfigurace, c je současná konfigurace, která se ve stromě opakuje, s je vyřešený rébus):



Je vidět, že navštívíme-li uzel c_1 jako první, má smysl uzel c_2 prohledat, protože cesta z i do s_2 je kratší, než cesta z i do s_1 kterou jsme již našli. Naopak, dostaneme-li se do uzlu c_1 poté, co jsme již c_2 navštívili, nemůžeme touto cestou žádné lepší řešení než s_2 nalézt a tento podstrom můžeme celý zamítnout.

Není-li konfigurace rovnou zamítnuta, nezapomeneme si pro pozdější výpočet poznačit její hloubku do atributu self.visited.

```
key = tuple(self.box)

if key in self.visited:
    if self.visited[key] <= len(self.moves):
        return

self.visited[key] = len(self.moves)
```

Druhý jednoduchý případ je již dobře známého typu: nalezli jsme řešení. Zároveň si poznačíme jeho hloubku v případě, že se jedná o řešení zatím nejlepší (nejkratší).

```
if self.box == list(range(0, 9)):
    self.best = min(self.best, len(self.moves))
    self.found = True
    return
```

Poslední jednoduchý případ je ten, kdy již víme, že nejkratší možná cesta ze současného stavu k řešení je delší, než ta zatím nejlepší nalezená. K tomu s výhodou použijeme pomocnou funkci need_steps, kterou jsme si dříve definovali. Připomeňme si, že tato nám dává **spodní odhad** na délku cesty k řešení: je-li tento příliš dlouhý, skutečná délka bude jistě také.

```
if len(self.moves) + need_steps(self.box) > self.best:
    return
```

Zbývá vyřešit případy, o kterých nelze přímo říct nic. Rekurzivně tedy prohledáme podstromy, do kterých vedou jednotlivé přípustné tahy. Najdeme-li v některé větvi nové nejlepší řešení, rekurzivní volání tuto skutečnost poznačí do atributů best a found.

```
for move in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
    if admissible(self.box, move):
        self.apply(move)
        self.search()
        self.backtrack()
```

Pomocná metoda-procedura, která spustí hledání, a vrátí jeho celkový výsledek: None v případě, kdy řešení neexistuje, jinak délku toho nejlepšího možného.

```
def solve(self) -> int | None:
    self.search()
    return self.best if self.found else None
```

Hotové řešení jako obvykle otestujeme na několika příkladech.

```
def main() -> None:  # demo
    assert Solver([0, 1, 2,
                   3, 4, 5,
                   6, 7, 8]).solve() == 0
    assert Solver([1, 0, 2, 3, 4, 5, 6, 7, 8]).solve() == 1
    assert Solver([1, 2, 0, 3, 4, 5, 6, 7, 8]).solve() == 2
    assert Solver([1, 2, 5, 3, 4, 0, 6, 7, 8]).solve() == 3
    assert Solver([1, 2, 5, 3, 0, 4, 6, 7, 8]).solve() == 4
    assert Solver([1, 0, 5, 3, 2, 4, 6, 7, 8]).solve() == 5
    assert Solver([0, 1, 5, 3, 2, 4, 6, 7, 8]).solve() == 6
    assert Solver([0, 8, 6, 5, 4, 7, 2, 3, 1]).solve() == 30
    assert Solver([8, 0, 6, 5, 4, 7, 2, 3, 1]).solve() == 31
```

10.e: Elementární příklady

10.e.1 **[subsets]** Napište čistou funkci, která dostane na vstup množinu čísel a vrátí seznam všech jejích podmnožin (v libovolném pořadí).

```
def subsets(input_set: set[int]) -> list[set[int]]:
    pass
```

10.e.3 **[flatten]** Typ pro libovolně zanořený seznam znáte z přednášky:

```
NestedList = list['int | NestedList']
```

Vaším úkolem je napsat čistou funkci, která na vstupu dostane NestedList (vnořený seznam celých čísel) a vrátí obyčejný seznam, který zachovává pořadí čísel na vstupu, ale „zapomene“ strukturu vnoření.

```
def flatten(to_flatten: NestedList) -> list[int]:
    pass
```

[23] Počet tahů není vůbec jednoduché odvodit teoreticky. Horní mez 31 tahů byla určena výpočetně, vyhledáním optimálního řešení z každého přípustného herního stavu. Pro hru „15 puzzle“ je tato mez 80 tahů (opět získána výpočetně). Znalost dobrého horního odhadu na délku řešení je pro efektivitu našeho algoritmu klíčová: pro zobecnění hry na $n \times n$ políček, kdy podobně dobrý odhad nemáme, je potřeba použít mírně sofistikovanější algoritmus. Jeho základní myšlenkou je nějakou mez zvolit, a nenajdeme-li v této mezi žádné řešení, postupně ji zvyšovat. To, jestli **nějaké** řešení existuje lze zjistit snadno z počáteční konfigurace, bez prohledávání.

[24] Tento slovník má trochu zvláštní typ. Je to proto, že seznam nelze použít jako klíč: seznam (typ Box) tedy musíme převést na n-tici, kterou již můžeme použít jako klíč. Zápis s třemi tečkami říká, že n-tice obsahuje nějaký počet celých čísel, který není blíže určený.

10.p: Přípravy

10.p.1 **[squaresum]** Napište predikát, který rozhodne, zda lze dané číslo num napsat jako součet $\sum_{i=1}^{n} a_i^2$, kde n je zadáno parametrem count a a_i jsou po dvou různá kladná čísla. Jinými slovy, lze num zapsat jako součet count druhých mocnin různých kladných čísel?

```
def is_sum_of_squares(num: int, count: int) -> bool:
    pass
```

10.p.2 **[permutations]** Napište čistou funkci, která ze vstupního seznamu vytvoří seznam všech jeho permutací (tedy seznamů takových, že jsou tvořena stejnými hodnotami v libovolném pořadí). Výsledný seznam permutací nechť je uspořádán lexikograficky.

Nápověda: řešení se znatelně zjednoduší, budete-li celou dobu pracovat se seřazenou verzí vstupního seznamu (seřazení je nakonec také jen permutace). Dobré řešení pak vytvoří každou permutaci pouze jednou a také je vytvoří rovnou ve správném pořadí.

```
def permutations(word: list[int]) -> list[list[int]]:
    pass
```

10.p.3 **[chain]** Napište predikát, který dostane na vstupu množinu čísel M a délku n a rozhodne, existuje-li navazující posloupnost čísel délky právě n. Navazující posloupnost je taková, kde každé další číslo začíná v jedenáctkovém zápisu stejnou číslicí, jakou končí předchozí. Čísla se v posloupnosti nesmí opakovat.

```
def elven_chain(numbers: set[int], length: int) -> bool:
    pass
```

10.p.4 **[digits]** Napište čistou funkci, která vrátí množinu všech čísel, kterých ciferný součet v desítkové soustavě je právě digit_sum a zároveň jejich počet cifer není větší než max_length (rozmyslete si, že bez tohoto omezení by byla hledaná množina nekonečná).

```
def digits(digit_sum: int, max_length: int) -> set[int]:
    pass
```

10.p.5 **[partitions]** † Rozkladem množiny M je množina neprázdných množin P_1, P_2, … P_n, které jsou vzájemně disjunktní a jejichž sjednocením je celá množina M.

Máme-li například množinu M = {1, 2, 3}, pak všechny její rozklady jsou:

```
{{1}, {2}, {3}}
{{1}, {2, 3}}
{{2}, {1, 3}}
{{3}, {1, 2}}
{{1, 2, 3}}
```

Vaším úkolem bude napsat čistou funkci, která vygeneruje všechny rozklady dané množiny celých čísel. Pro zjednodušení nebudeme pracovat s datovým typem množina, ale všechny množiny budeme reprezentovat pomocí seznamů. Můžete předpokládat, že jednotlivé prvky vstupního seznamu jsou unikátní.

```
NumSet = list[int]
Partitions = list[list[list[int]]]

def partitions(nums: NumSet) -> Partitions:
    pass
```

10.p.6 **[circle]** Napište (čistou) funkci, která dostane na vstupu množinu čísel a vrátí délku nejdelšího šestnáctkového kruhu, který se z nich dá vytvořit. Pokud se žádný kruh vytvořit nedá, vrátí 0.

Šestnáctkový kruh je posloupnost čísel (bez opakování) taková, že každé další číslo začíná v šestnáctkovém zápisu stejnou cifrou, jakou končí číslo předchozí. Navíc první číslo v posloupnosti začíná stejnou číslicí, jakou končí poslední číslo.

```
def hex_circle(numbers: set[int]) -> int:
    pass
```

10.r: Řešené úlohy

10.r.1 **[subseq]** Na vstupu dostanete neprázdný seznam čísel. Vaším úkolem je vrátit množinu všech seznamů, které:

1. jsou vlastními podposloupnostmi vstupního seznamu, tzn. vzniknou ze vstupního seznamu vynecháním alespoň jednoho prvku,
2. jsou neklesající, tzn. pro seznam out a celá čísla i, j platí i < j → out[i] ≤ out[j].

Protože datový typ set neumožňuje ukládat seznamy jako prvky, výstup uložte do seznamu (na pořadí jednotlivých posloupností v tomto seznamu nezáleží).

```
def subseq(seq: list[int]) -> list[list[int]]:
    pass
```

10.r.2 **[equivalence]** Z předmětu IB000 Matematické základy informatiky víme, že každá relace ekvivalence na nějaké množině M jednoznačně určuje rozklad množiny M, tedy množinu vzájemně disjunktních podmnožin, jejichž sjednocením je celá množina M. Platí to i naopak, každý rozklad jednoznačně určuje relaci ekvivalence.

Například na množině M = {1, 2, 3} můžeme definovat relaci ekvivalence {(1, 1), (2, 2), (2, 3), (3, 2), (3, 3)}, které odpovídá rozklad [{1}, {2, 3}]

Napište funkci partition2pairs, která jako parametr dostane rozklad množiny (tedy seznam podmnožin) a vrátí množinu uspořádaných dvojic, které představují odpovídající relaci ekvivalence. Dále napište funkci pairs2partitions, která z relace zadané jako množina uspořádaných dvojic vytvoří odpovídající rozklad (seznam podmnožin). V obou případech můžete předpokládat, že vstup je korektní.

```
Pair = tuple[int, int]

def partition2pairs(partition: list[set[int]]) -> set[Pair]:
    pass

def pairs2partition(pairs: set[Pair]) -> list[set[int]]:
    pass
```

10.r.4 **[nested]** † Z přednášky již znáte vnořený seznam čísel NestedList:

```
NestedList = list['int | NestedList']
```

Napište proceduru, která na vstupu dostane NestedList celých čísel a upraví ho tak, aby v něm byla čísla seřazená vzestupně napříč všemi vnitřními seznamy. Například seznam [[4, 7, 1], [], [8], [0, 5]] se použitím této procedury změní na [[0, 1, 4], [], [5], [7, 8]].

```
def sort_nested(list_of_lists: NestedList) -> None:
    pass
```

10.r.5 **[subsetsum]** Napište čistou funkci, která najde libovolnou podmnožinu zadané množiny kladných celých čísel nums, součet jejíchž prvků je přesně total. Pokud taková podmnožina neexistuje, funkce vrátí None.

Při řešení přemýšlejte, jestli některé výpočty neprovádíte opakovaně a jak byste se tomu mohli vyhnout.

```
def subset_sum(nums: set[int], total: int) -> set[int] | None:
    pass
```

10.r.6 **[dnfsat]** Ve třetí ukázce této kapitoly jsme řešili problém splnitelnosti výrokové formule. Tato formule byla ve speciálním tvaru, takzvané konjunktivní normální formě.

Nyní se podíváme na stejný problém pro formule v jiném speciálním tvaru – v tzv. **disjunktivní** normální formě. V tomto tvaru se formule skládá opět z klauzulí, tentokrát je ale jejich disjunkcí. Uvnitř závorek se pak objevuje konjunkce literálů. Například:

$$(a \wedge b) \vee (\neg a \wedge b \wedge a) \vee (\neg a \wedge c \wedge b \wedge \neg c)$$

Napište čistou funkci satisfiable, která rozhodne, je-li takto zadaná formule splnitelná. Než se pustíte do řešení, dobře si rozmyslete, co splnitelnost znamená a v jakých přesně případech je formule v tomto tvaru (ne)splnitelná. Typy, kterými formuli reprezentujeme jsou stejné, jako ty v ukázce.

```
Variable = str
```

```
Literal = tuple[Variable, bool]
Clause = list[Literal]
Formula = list[Clause]

def satisfiable(phi: Formula) -> bool:
    pass
```

10.v: Volitelné úlohy

10.v.1 [powers] Napište čistou funkci sum_different_powers, která pro zadané kladné celé číslo num a celé číslo k ≥ 2 rozhodne, zda se dá num napsat jako součet druhé, třetí, … až k-té mocniny **různých** kladných celých čísel.

Funkce musí rozumně fungovat pro num v řádech milionů a pro k do 10.

Příklad: Volání {fun}(17, 3) vrátí True, protože $17 = 4^2 + 1^3$. Volání {fun}(80, 3) vrátí False, protože není žádný způsob, jak číslo 80 zapsat jako součet druhé a třetí mocniny různých kladných celých čísel. Volání {fun}(365, 5) vrátí True, protože $365 = 10^2 + 2^3 + 4^4 + 1^5$. Volání {fun}(1000, 4) vrátí True, protože $1000 = 24^2 + 7^3 + 3^4$. Volání {fun}(1002, 4) vrátí False, protože 1002 se nedá zapsat jako součet druhé, třetí a čtvrté mocniny různých kladných celých čísel.

```
def sum_different_powers(num: int, k: int) -> bool:
    pass
```

10.v.3 [lowest] V tomto příkladu máme na vstupu neprázdný řetězec desítkových číslic (tj. znaků '0' až '9'), který **nezačíná znakem '0'**, a chceme je rozsekat na části tak, aby tvořily rostoucí posloupnost čísel zapsaných v desítkové soustavě, přitom žádná část nesmí začínat znakem '0'. Ze všech takových posloupností pak chceme vybrat tu, která má co nejnižší své poslední číslo. Vaším úkolem je napsat čistou funkci, která spočítá toto číslo. Funkce by měla fungovat na vstupech o řádově desítkách znaků.

Příklad: Řetězec "23245" můžeme rozsekat na rostoucí posloupnosti následujícími způsoby: 2, 3, 245 nebo 2, 32, 45 nebo 23, 245 nebo 23245. Nejnižší poslední číslo je 45; volání lowest_increasing_sequence_end("23245") tedy vrátí 45.

```
def lowest_increasing_sequence_end(digits: list[int]) -> int:
    pass
```

Část 11: Rekurze III, práce s textem

Demonstrační příklady:

1. spellcheck – jednoduchá kontrola pravopisu
2. printlist – výpis vnořených odrážkových seznamů
3. printdict – výpis vnořených slovníků bez rekurze

Elementární příklady:

1. names – čtení a zápis csv
2. xxx
3. wordfreq – frekvence slov v textovém souboru

Přípravy:

1. config – formátování strukturovaných dat do souboru
2. rewrite – přepisovací pravidla
3. ipv4 – kontrola a konverze adres protokolu IPv4
4. parser – čtení seznamů ze souboru
5. build – převod vnořených závorek na strom
6. template – rozepisováni závorek

Rozšířené úlohy:

1. brackets – převod stromu na uzávorkovaný řetězec
2. ipv4fix – oprava poškozené IPv4 adresy
3. trailing – odstranění přebytečných mezer ze souboru
4. correct – automatická oprava překlepů
5. language – identifikace jazyka
6. weighted – generování všech řetězců se zadanými vlastnostmi

11.1: Programovací jazyk

Tato kapitola přidává operace práci s **řetězci**. Krom nových **výrazů** se drobná rozšíření dotknou i příkazu for (který můžeme použít k procházení řetězce po znacích). Na rozdíl od seznamů ale pro řetězce neexistuje vnitřní přiřazení.

Tato kapitola přináší také prostředky pro jednoduchou práci se soubory a další interakci s prostředím (zejména operačním systémem).

11.1.1 Literály Podobně jako tomu bylo v případě seznamů a n-tic, řetězce můžeme do programu zapsat pomocí **řetězcových literálů**. Ty mají jeden z těchto tvarů: 'znaky', "znaky", """znaky""", '''znaky'''. Významově jsou všechny tyto tvary ekvivalentní: vytvoří hodnotu typu řetězec, která obsahuje znaky.

Pro většinu znaků je obsah vzniklého řetězce totožný se zápisem literálu, až na dva druhy výjimek:

- některé znaky nebo sekvence znaků se v literálech nesmí mimo speciální sekvence objevit:
  - znak konce řádku v literálech s jednoduchým oddělovačem ('znaky' a "znaky"),
  - samotný oddělovač ('', "", ''', """) použitý pro zápis daného literálu – nebylo by zřejmé, zda se jedná o konec literálu nebo nikoliv,
- některé sekvence znaků, které začínají znakem \ (zpětné lomítko) se **přeloží** na jeden znak:
  - \', \" se přeloží na samotné znaky ' a ",
  - \\ se přeloží na znak \,
  - \n se přeloží na znak konce řádku,
  - \a, \b, \f, \r, \t, \v se přeloží na různé speciální znaky, které v tomto kurzu nebudou důležité,
  - \NNN a \xNN, \uNNNN, \UNNNNNNNN, kde N… je třiciferný osmičkový nebo dvou-, čtyř- nebo osmiciferný šestnáctkový zápis nějakého čísla n, se přeloží na znak x který má v tabulce znaků Unicode pozici n.

Snadno se přesvědčíte, že „zakázané“ znaky resp. sekvence znaků lze vždy zapsat nějakým alternativním způsobem pomocí \-sekvencí.

11.1.2 Výrazy Podobně jako seznamy, řetězce lze **indexovat**: zápis je stejný jako u seznamů: řetězec[index], kde řetězec je **jméno** a index je celočíselný výraz. Na rozdíl od seznamů, výsledkem indexace je v případě řetězce **opět řetězec**, který ale obsahuje pouze jediný znak.

Dále nově připouštíme relační operátory x == y, x != y, x < y, x > y, x <= y, x >= y i v případě, kdy se podvýrazy x a y oba vyhodnotí na řetězce. Uspořádání je dáno **lexikograficky**.

11.1.3 Příkazy Jediný nový příkaz, který souvisí s řetězci, je

```
for ch in řetězec:
    příkazy
```

kde ch je **jméno** a řetězec je **výraz**, který se vyhodnotí na hodnotu typu řetězec. Podobně jako ostatní varianty příkazu for, tento provede sekvenci příkazy jednou pro každý znak uložený v řetězci řetězec. Jméno ch je přitom v i-té iteraci vázáno na jednopísmenný řetězec odpovídající znaku na i-té pozici hodnoty řetězec.

Pro práci se soubory (a dalšími zdroji, o kterých ale v tomto předmětu nebude řeč) budeme krom zabudovaného podprogramu open (vysvětleno níže) slouží také příkaz with – je obvyklé je používat vždy společně, a to ve tvaru:

```
with open(cesta, režim) as název:
    příkazy
```

Tato konstrukce nám umožní se souborem pracovat v těle příkazu with pomocí jména název (stejně, jako kdybychom přiřadili výsledek volání open do proměnné), ale navíc máme zaručeno, že po opuštění tohoto bloku je práce se souborem korektně ukončena.

Takto otevřený a pojmenovaný soubor můžeme **iterovat** již dobře známým příkazem for:

```
for řádek in soubor:
    příkazy
```

kde řádek je jméno a soubor je výsledek volání open (obvykle vázaný příkazem with). Ke jménu řádek budou postupně vázány hodnoty typu str, které obsahují vždy jeden řádek souboru (včetně ukončovacího znaku '\n'). Cyklus je ukončen po přečtení posledního řádku.

11.1.4 Zabudované podprogramy Objekty typu řetězec navíc poskytují tyto **zabudované metody** (ve všech případech jsou zároveň **čistými funkcemi** – vstupní řetězec nikdy nemodifikují):

- s.isupper(), s.islower() – predikáty, vyhodnotí se na True v případě, že všechny abecední znaky v řetězci s jsou velká (resp. malá) písmena,
- s.isalpha(), s.isdecimal() – predikáty, které se vyhodnotí na True sestává-li s pouze z abecedních znaků (isalpha) resp. desítkových číslic (isdecimal),
- s.upper(), s.lower() – vyhodnotí se na řetězec, který vznikne ze s nahrazením všech abecedních znaků na odpovídající velká (upper) resp. malá (lower) písmena,
- s.split(delim) – vyhodnotí se na **seznam**, který vznikne rozdělením s na podřetězce oddělovačem delim (oddělovače nejsou součástí výsledných řetězců),
- s.join(parts) – vyhodnotí se na řetězec, který vznikne vložením řetězce s mezi každé dva řetězce uložené v **seznamu** parts,
- s.replace(from, to) – vyhodnotí se na řetězec, který vznikne ze s substitucí všech výskytů podřetězce from za podřetězec to,
- s.rstrip() – vyhodnotí se na řetězec, který vznikne odstraněním všech pravostranných bílých znaků (zejména mezer a znaků konce řádku).

Jak bylo naznačeno výše, práci se soubory nám umožňuje zabudovaný podprogram open(cesta, režim)[25]. Parametr cesta (typu řetězec) určuje kde v souborovém systému se má hledat soubor, se kterým chceme pracovat, řetězec režim pak určuje jakým způsobem hodláme soubor používat. Základní možnosti jsou tyto:

[25] Nejedná se v tomto případě ani o čistou funkci, ale ani o klasickou proceduru.

- 'r' – režim pouze pro čtení nám umožní ze souboru číst textová data, ale nic dalšího,
- 'w' – režim pro zápis textu, kdy je soubor při otevření zkrácen na nulovou délku (z takto otevřeného souboru nelze číst),
- 'x' – jako 'w', ale soubor je prvně vytvořen (v případě, že již existuje, je program ukončen s chybou),
- 'a' – jako 'w', ale soubor není zkrácen, nová data jsou zapisována na konec souboru.

Tyto základní možnosti lze kombinovat se specifikátorem 't' nebo 'b', který určí, chceme-li se souborem pracovat v **textovém** nebo **binárním** režimu. Neuvedeme-li ani jedno z nich, implicitní je textový režim. V tomto předmětu se omezíme na textový režim.

S hodnotou f, které vznikne voláním podprogramu open v textovém režimu, můžeme použít také několik zabudovaných metod:

- f.close() – ukončí práci se souborem (obvykle **nepoužíváme**, ukončení provedeme místo toho správným použitím příkazu with),
- f.read(n) – přečte nejvýše n znaků a vrátí je jako hodnotu typu str,
- f.readline() – přečte znaky od aktuální pozice až do konce řádku a vrátí je jako hodnotu typu str,
- f.readlines() – přečte celý zbytek souboru po řádcích, výsledkem je hodnota typu list, která obsahuje pro každý přečtený řádek jednu položku typu str,
- f.write(s) – zapíše řetězec s (t.j. hodnotu typu str) do souboru.

11.1.5 Knihovny Většina funkcionality pro interakci s vnějším světem je k dispozici formou knihoven (obdoba knihovny math, kterou známe z první kapitoly). Zde uvádíme pouze stručný přehled, bližší informace k použití jednotlivých knihoven získáte v 11. přednášce. Použití knihovny je potřeba vždy na začátku souboru deklarovat řádkem

from knihovna import $jméno_1$, $jméno_2$, …
K dispozici máme tyto knihovny:

- gzip – práce s komprimovanými soubory *.gz,
  - open – otevře komprimovaný soubor (dále s ním lze pracovat jako s obyčejným souborem, liší se ale implicitním použitím binárního režimu) – voláme pomocí příkazu with,
- csv – práce s textovými soubory, které obsahují tabulky hodnot oddělené čárkou (nebo jiným oddělovačem),
- sys – obecná interakce se systémem:
  - argv – seznam hodnot typu str, které byly programu předány při spuštění na příkazové řádce,
- os – další podprogramy (zejména procedury) pro práci se systémem (cesta je hodnota typu str):
  - remove(cesta) – odstraní (smaže) soubor,

11.d: Demonstrace (ukázky)

11.d.1 [spellcheck] V této ukázce načteme seznam slov uložených v komprimovaném souboru (ve formátu gzip) a použijeme jej k implementaci (velmi zjednodušené) kontroly pravopisu. K načtení souboru použijeme standardní modul gzip.

import gzip
Načtení slovníku realizujeme jednoduchým podprogramem read_dictionary, který soubor dekomprimuje a slova uloží do množiny (množina proto, abychom dokázali slova rychle vyhledávat). Výstup dekompresního algoritmu budeme **číst** (písmenko r v parametru mode) v **textovém režimu** (písmenko t). Dekomprimovaná data pak již čteme stejně jako libovolný jiný soubor, třeba iterací, která postupně vrací jednotlivé řádky. Protože slova jsou v souboru uložena ve formátu 1 řádek = 1 slovo, bude nám právě tento režim vyhovovat. K odstranění znaků konce řádku použijeme metodu strip.

```
def read_dictionary(path: str) -> set[str]:
    out: set[str] = set()
    with gzip.open(path, 'rt') as data:
        for word in data:
            out.add(word.strip())
    return out
```

Samotnou kontrolu provede čistá funkce spellcheck. Vstupem je množina přípustných slov (obsah seznamu slov načteného výše) a text, který chceme zkontrolovat. Výstupem je pak krom samotných neznámých slov také seznam čísel řádků, na kterých se ve vstupu objevují. K reprezentaci použijeme slovník, kde klíčem je špatně napsané slovo a hodnotou zmiňovaný seznam.

Abychom se alespoň trochu přiblížili realitě, budeme se chtít vypořádat s některými problémy:

- slova nejsou vždy oddělena mezerami: často se objevují čárky, tečky, uvozovky, závorky a podobně,
- na velikosti písmen občas záleží, ale ne vždy:
  - slovo, které ve slovníku obsahuje velká písmena, je, napíšeme-li jej malými písmeny, typicky chybou (třeba „Jean-Pierre“)
  - naopak, slovo, které je ve slovníku malými písmeny, může v textu stát na začátku věty, nebo obsahovat velká písmena z jiného důvodu, a typicky to chyba není.

Skutečné programy pro kontrolu pravopisu jsou obvykle mnohem složitější, nám ale bude tato úroveň realizmu stačit. Metody, které neznáte, si dohledejte v dokumentaci: i to je důležitá součást programování.

```
def spellcheck(dictionary: set[str], text: str) -> dict[str, list[int]]:
    problems: dict[str, list[int]] = {}
    to_erase = {',', '.', '!', '?', '(', ')', '"'}
    for lineno, line in enumerate(text.split('\n')):
        processed = ''
        for char in line:
            processed += ' ' if char in to_erase else char
        for word in processed.split():
            if word not in dictionary and \
               word.lower() not in dictionary:
                if word not in problems:
                    problems[word] = []
                problems[word].append(lineno + 1)
    return problems
```

Celý program otestujeme na několika jednoduchých vstupech. Slovník naleznete v souboru zz.words.gz (na stroji aisa si jej můžete prohlédnout třeba příkazem zless).

```
def main() -> None:  # demo
    dictionary = read_dictionary('zz.words.gz')
    assert len(spellcheck(dictionary, 'hello world')) == 0
    assert len(spellcheck(dictionary, 'hello, world!')) == 0
    assert len(spellcheck(dictionary, 'hello, borld!')) == 1
    bad = spellcheck(dictionary, 'Hello, borld!\nErr, I mean'
                     '"world". Truely.')
    assert bad == {'borld': [1], 'Truely': [2]}, str(bad)
```

11.d.2 [printlist] V této ukázce se zaměříme na rekurzivní procedury pro práci s výstupem. Konkrétně se budeme zabývat vnořenými odrážkovými seznamy, které budeme v programu reprezentovat jako seznam objektů typu Item. Každá odrážka (instance Item) v takovém seznamu má nějaký vlastní text (atribut text) a případně seznam pododrážek (atribut sublists).

```
class Item:
    def __init__(self, text: str):
        self.text: str = text
        self.sublists: list[Item] = []
```

V parametru itemize budeme proceduře print_itemize_rec předávat relevantní odrážkový seznam, v parametru prefix budeme uchovávat řetězec, který vypíšeme před každou jednotlivou odrážkou: tím budeme realizovat zanoření, které by mělo ve výstupu vypadat takto:

```
- odrážka 1
 - odrážka druhé úrovně
 - další odrážka druhé úrovně
- odrážka 2
 - zanořená odrážka
  - ještě zanořenější odrážka
```

Na této proceduře je zajímavé také to, že bázový případ není zmíněn explicitně: pozorný čtenář si ale jistě všimne, že odrážka, která již žádné pododrážky nemá, bude mít seznam sublists prázdný. Na prázdném seznamu ale procedura print_itemize_rec neudělá vůbec nic: cyklus v jejím těle se ani jednou neprovede.

Výstup postupně sestavujeme v seznamu lines, který si předáváme pomocným parametrem.

```
def format_itemize(itemize: list[Item], prefix: str,
                   lines: list[str]) -> None:
    for i in itemize:
        lines.append(prefix + '- ' + i.text + "\n")
        format_itemize(i.sublists, prefix + ' ', lines)
```

Procedura print_itemize pomocí procedury format_itemize vytvoří seznam řádků a tyto uloží do souboru: krom otevření souboru pro zápis se stará také o nastartování rekurze.

```
def print_itemize(itemize: list[Item], path: str) -> None:
    lines: list[str] = []
    format_itemize(itemize, '', lines)
    with open(path, 'w') as out:
        for line in lines:
            out.write(line)
```

Tím je ukázka kompletní. Program jako obvykle otestujeme na jednoduchém vstupu.

```
def main() -> None:  # demo
    path = 'zt.print_itemize.txt'
    itemize = [Item('foo'), Item('bar'), Item('wibble')]
    itemize[1].sublists.extend([Item('baz'), Item('quux')])
    itemize[1].sublists[0].sublists.append(Item('baz 2'))
    itemize[2].sublists.extend([Item('quuux')])
    print_itemize(itemize, path)
    assert open(path).read() == ('- foo\n'
                                 '- bar\n'
                                 ' - baz\n'
                                 '   - baz 2\n'
                                 ' - quux\n'
                                 '- wibble\n'
                                 ' - quuux\n')
```

11.d.3 [printdict] Tato ukázka je variací na předchozí: budeme opět zapisovat rekurzivní datovou strukturu do souboru, tentokrát na to ale použijeme zápis bez rekurze. Nejprve si zadefinujeme potřebné typy, zejména třídu NestedDict. Tato reprezentuje zanořený slovník, kde klíče jsou řetězce a hodnoty jsou buď řetězce, nebo vnořené slovníky.

```
NestedDict = dict[str, 'str | NestedDict']
```

Výpis slovníku provede procedura print_nested. Formát výpisu bude následovný:

- je-li ke klíči asociovaná hodnota typu řetězec, klíč a hodnota se vypíšou na jeden řádek, oddělené dvojtečkou, patřičně odsazené dle úrovně zanoření,
- je-li hodnota zanořený slovník, klíč se vypíše na samostatný řádek ukončený dvojtečkou a obsah slovníku se vypíše pod něj, odsazený o jednu mezeru navíc.

Klíče seznamu budou seřazeny abecedně. Příklad:

```
klíč 1:
 abecedně první klíč vnořeného slovníku: řetězec
 další klíč vnořeného slovníku: jiný řetězec
 třetí klíč:
  více zanořený klíč: další řetězec
klíč 2: řetězec v hlavním slovníku
```

Z kapitoly 6 si jistě pamatujete základní datové struktury: k procházení rekurzivní struktury bez použití rekurze se bude hodit zásobník, který budeme realizovat seznamem a jeho metodami append (vloží prvek na vrchol zásobníku) a pop (odebere prvek z vrcholu).

```
def print_nested(records: NestedDict, path: str) -> None:
```

Začneme tím, že si otevřeme soubor path pro zápis a výsledek si poznačíme do proměnné out.

```
    with open(path, 'w') as out:
```

Dále si nachystáme zásobník, ve kterém budeme uchovávat rozpracované podúlohy. Tyto budeme reprezentovat jako dvojice:

- jednak si musíme pamatovat, který zanořený slovník na dané úrovni zanoření právě zpracováváme (toto bude první složka),
- dále pak u každého rozpracovaného slovníku potřebujeme vědět, které klíče je ještě potřeba zpracovat (resp. které jsme již vypsali).

Pro začátek na zásobník vložíme „hlavní“ slovník (ten, který jsme dostali jako parametr) a poznačíme si, že musíme zpracovat všechny jeho klíče. Protože klíče ke zpracování budeme odebírat z konce seznamu (kvůli efektivitě), vložíme je do seznamu v opačném abecedním pořadí.

```
        stack = []
        todo = list(records.keys())
        todo.sort()
        todo.reverse()
        stack.append((records, todo))
```

Tím máme nachystaný počáteční stav a dále budeme zpracovávat jednotlivé podúlohy, a každou, kterou dokončíme ze zásobníku odstraníme. Podúlohy budeme zpracovávat až do chvíle, kdy se zásobník zcela vyprázdní. Narazíme-li během zpracování některé podúlohy na další (vnořený slovník), podobně je vložíme do zásobníku.

```
        while stack:
```

Pracujeme vždy s podúlohou na vrcholu zásobníku, tzn. tou „nejnovější“ (vzpomeňte si, že zásobník je „last in, first out“).

```
            items, keys = stack[-1]
```

Dojdou-li nám v daném slovníku (podúloze) klíče ke zpracování, jsme hotovi: podúlohu odstraníme ze zásobníku a pokračujeme ve výpočtu s další podúlohou (která se tímto dostala na vrchol).

```
            if not keys:
                stack.pop()
                continue
```

Množina klíčů ke zpracování nebyla prázdná – stojíme tedy před nedokončenou podúlohou. Ze seznamu nezpracovaných klíčů jeden vybereme a zpracujeme (k tomu budeme potřebovat i odpovídající hodnotu).

```
            key = keys.pop()
            value = items[key]
```

Pro účely výpisu si spočteme řetězec s mezerami, které je potřeba umístit na začátek řádku – protože „hlavní“ slovník je odsazen o 0 mezer, počet mezer je o jedna menší než současná hloubka zásobníku.

```
            prefix = ''.join([' ' for _ in range(len(stack) - 1)])
```

Nyní se musíme rozhodnout, jakého typu je hodnota, kterou máme zpracovat: je-li to řetězec, vypíšeme jej přímo ke klíči. Naopak, je-li to zanořený slovník, vypíšeme pouze klíč a podslovník zařadíme mezi podúkoly, které je potřeba zpracovat, a to tak, že jej (opět se všemi klíči) vložíme na vrchol zásobníku.

```
            if isinstance(value, str):
                print(prefix + key + ': ' + value, file=out)
            else:
                print(prefix + key + ':', file=out)
                todo = sorted(value.keys())
                todo.reverse()
                stack.append((value, todo))
```

Proceduru print_nested si na jednoduchém vstupu ještě otestujeme.

```
def main() -> None:  # demo
    path = 'zt.print_dict.txt'
    d1: NestedDict = {'y': 'foo', 'x': 'bar'}
    d11: NestedDict = {'x': 'baz'}
```

```
    d2: NestedDict = {'dictionary 1.1': d11, 'string': 'quux'}
    d: NestedDict = {'dictionary 1': d1, 'dictionary 2': d2,
                     'string 1': 'str'}
    print_nested(d, path)
    assert open(path).read() == ('dictionary 1:\n'
                                 ' x: bar\n'
                                 ' y: foo\n'
                                 'dictionary 2:\n'
                                 ' dictionary 1.1:\n'
                                 '  x: baz\n'
                                 ' string: quux\n'
                                 'string 1: str\n')
```

11.e: Elementární příklady

11.e.3 [wordfreq] Napište funkci, která ve vstupním souboru najde 3 nejčastější slova. Obsahuje-li soubor méně než 3 různá slova, výsledný seznam bude kratší. V případě, kdy mají dvě slova stejnou frekvenci výskytu, upřednostněte to, které je lexikograficky menší.

```
def most_common(path: str) -> list[str]:
    pass
```

11.p: Přípravy

11.p.1 [config] Napište proceduru write_config, která do souboru zadaného cestou filename zapíše konfiguraci ze slovníku config. (Pokud už takový soubor existuje, přepište jej.) Struktura slovníku je taková, že klíč je název sekce a hodnotou další slovník, který již obsahuje dvojice klíč-hodnota typu řetězec.

Formát výstupního souboru nechť je následující:

- prázdné sekce (takové, kterým je přiřazený prázdný slovník) ignorujeme,
- pro každou neprázdnou sekci zapíšeme řádek [jméno sekce] a na další řádky postupně vypíšeme obsah příslušného slovníku ve formátu klíč = "hodnota".
- sekce i jednotlivé klíče v každé sekci uspořádejte na výstupu podle abecedy.

Příklad: pro vstupní slovník

```
{ 'main': { 'code': 'IB111',
            'name': 'Základy programování' },
  'empty': {},
  'exams': { 'hard': 'no' } }
```

se do zadaného souboru zapíše toto:

```
[exams]
hard = "no"
[main]
code = "IB111"
name = "Základy programování"
```

Pro slovník s konfigurací si zavedeme typové synonymum Config:

```
Config = dict[str, dict[str, str]]

def write_config(filename: str, config: Config) -> None:
    pass
```

11.p.2 [rewrite] Napište predikát, jehož hodnota bude True pokud lze požadované slovo wanted utvořit z iniciálního slova initial pomocí přepisovacích pravidel rules a False jinak. Slova vytváříme tak, že kterékoli písmeno z již vytvořených slov nacházející se mezi klíči slovníku pravidel rules můžeme nahradit za kterékoli písmeno z příslušné hodnoty. (Pro zjednodušení možnost zacyklení procesu vytváření slov nemusíte vůbec řešit.)

```
def is_creatable(wanted: str, initial: str,
                 rules: dict[str, list[str]]) -> bool:
    pass
```

11.p.3 [ipv4] V této úloze se budeme zabývat adresami protokolu IP verze 4, které sestávají ze 4 čísel oddělených tečkami, například 192.0.2.0 (více informací o IPv4 naleznete například na Wikipedii). Adresy budeme reprezentovat řetězci.

Napište predikát, kterého hodnota bude True, představuje-li jeho parametr validní IPv4 adresu. Daná IPv4 adresa je validní právě tehdy, když je tvořená čtyřmi dekadickými čísly od 0 až 255 (včetně) oddělenými tečkou (pro jednoduchost v této úloze připouštíme pouze kanonický tvar IPv4 adres).

```
def ipv4_validate(address):
    pass
```

Dále napište čistou funkci, která vypočte číselnou hodnotu dané adresy. Konverze IPv4 adresy na její číselnou hodnotu je podobná konverzi binárního zápisu čísla na dekadický s tím rozdílem, že u IPv4 adresy pracujeme se základem 256. Hodnota adresy 192.0.2.0 je tedy $192 \cdot 256^3 + 0 \cdot 256^2 + 2 \cdot 256^1 + 0 \cdot 256^0 = 3\,221\,225\,984$. Můžete počítat s tím, že vstupem bude vždy validní IPv4 adresa ve výše popsaném kanonickém tvaru.

```
def ipv4_value(address):
    pass
```

11.p.4 [parser] V tomto úkolu budeme ze zadaného souboru číst vnořené odrážkové seznamy:

- každý seznam je uvozený jménem na samostatném řádku,
- po jméně následuje samotný seznam, přičemž každá odrážka je opět na samostatném řádku,
- zanoření odrážky lze rozeznat podle počtu mezer před odrážkou (znakem -): 1 mezera značí odrážku první úrovně, 2 mezery odrážku druhé úrovně, atd.,
- mezi sousedními řádky se může úroveň zanoření zvýšit nejvýše o jedna, snížit se ale může libovolně.

Příklad zanořeného seznamu (v souboru je takových několik, oddělených prázdným řádkem):

```
List 1

- Item 1
 - Item 1.1
 - Item 1.2
  - Item 1.2.1
   - Item 1.2.1.1
 - Item 1.3
  - Item 1.3.1
- Item 2
```

Seznam budeme na výstupu reprezentovat dvěma třídami:

- Item reprezentuje odrážku s textem v atributu text a případným podseznamem v atributu sublists,
- Itemize pak reprezentuje seznam jako celek, se jménem name a odrážkami první úrovně v seznamu items.

Tyto třídy nijak nemodifikujte.

```
class Item:
    def __init__(self, text: str):
        self.text: str = text
        self.sublists: list[Item] = []

class Itemize:
    def __init__(self, name: str):
        self.name: str = name
        self.items: list[Item] = []
```

Implementujte podprogram parse_lists, který vrátí seznam instancí třídy Itemize, které přečte ze souboru s názvem filename. Můžete předpokládat, že soubor obsahuje pouze správně formátované seznamy a mezi každými dvěma seznamy je jeden prázdný řádek.

```
def parse_lists(filename: str) -> list[Itemize]:
    pass
```

11.p.5 [build] † V tomto příkladu budeme pracovat s n-árními stromy, které nemají v uzlech žádné hodnoty (mají pouze stromovou strukturu). Třídu Tree nijak nemodifikujte.

```
class Tree:
    def __init__(self) -> None:
        self.children: list[Tree] = []
```

Napište (čistou) funkci, které na základě dobře uzávorkovaného řetězce tvořeného pouze znaky (a) vybuduje instanci výše popsaného stromu, a to tak, že každý pár závorek reprezentuje jeden uzel, a jejich obsah reprezentuje podstrom, který v tomto uzlu začíná. Ve vstupním řetězci bude vždy alespoň jeden pár závorek.

```
def build_tree(brackets: str) -> Tree:
    pass
```

11.p.6 [template] Napište čistou funkci, která na základě daného vzoru vytvoří množinu všech odpovídajících řetězců. Vzor je tvořený alfanumerickými znaky a navíc může obsahovat hranaté závorky – znaky [a]. Mezi těmito závorkami může stát libovolný počet přípustných znaků (krom samotných hranatých závorek) a na daném místě se ve výsledném řetězci může nacházet libovolný z těchto znaků. Například vzor a[bc]d reprezentuje řetězce abd a acd.

```
def resolve_template(template: str) -> set[str]:
    pass
```

11.r: Řešené úlohy

11.r.1 [brackets] V tomto příkladu budeme pracovat se stromy, které mají v jednotlivých uzlech uloženy řetězce. Tyto stromy budeme používat k reprezentaci aritmetických výrazů složených z konstant a binárních operátorů:

- konstantu reprezentuje strom, který má oba podstromy prázdné,
- složený výraz je reprezentován stromem, který má v kořenu uložen operátor a jeho neprázdné podstromy reprezentují operandy.

Žádné jiné uzly ve stromě přítomny nebudou.

```
class Tree:
    def __init__(self, value: str,
                 left: 'Tree | None',
                 right: 'Tree | None'):
        self.value = value
        self.left = left
        self.right = right

def leaf(value: str) -> Tree:
    return Tree(value, None, None)
```

Napište čistou funkci, která dostane výše popsaný strom jako parametr a vrátí odpovídající plně uzávorkovaný aritmetický výraz, formou řetězce. Plným uzávorkováním myslíme, že každému aritmetickému operátoru přísluší jedna dvojice kulatých závorek.

```
def tree_to_expr(tree) -> str:
    pass
```

11.r.2 [ipv4fix] Napište (čistou) funkci, která dostane na vstup řetězec složený pouze z číslic od 1 do 9 včetně a vrátí množinu všech možných IPv4 adres, z nichž tento řetězec mohl vzniknout vynecháním teček. Za IPv4 adresu považujeme řetězec tvořený čtyřmi čísly v rozsahu od 0 po 255 včetně oddělenými tečkami. Například řetězec 25525511135 mohl vzniknout výše popsaným způsobem z adres 255.255.11.135 a 255.255.111.35.

```
def ipv4_restore(digits: str) -> set[str]:
    pass
```

11.r.3 [trailing] Někdy se stane, že při programování v Pythonu omylem necháte na konci řádku mezery, nebo jiné bílé znaky (např. tabulátor). Při kontrole programem edulint je toto označeno za chybu. Vaším úkolem je napsat jednoduchý program, který tento typ chyby v zadaných souborech opraví. Seznam souborů k opravě dostanete jako argumenty na příkazové řádce (v Pythonu je naleznete v seznamu sys.argv počínaje indexem 1). Soubor, se kterým právě pracujete, můžete načíst celý do paměti.

Poznámka: tento program lze testovat dvěma způsoby. Spustíte-li jej bez dalších parametrů, spustí se přiložené testy. Předáte-li naopak programu nějaké parametry, spustí se přímo procedura trailing, která tyto zpracuje obvyklým způsobem. Například:

```
python r3_trailing.py soubor1.txt soubor2.py
```

```
def trailing() -> None:
    pass
```

11.r.4 [correct] V první ukázce jsme viděli jednoduchý program na kontrolu pravopisu. Tento úkol bude podobný, ale místo vyznačení nalezených chyb je budeme rovnou opravovat.

Ze 4. kapitoly si možná pamatujete tzv. Hammingovu vzdálenost: jednalo se o funkci, která dvojici slov stejné délky přidělí nezáporné celé číslo: počet znaků, ve kterých se liší. Náš „autocorrect“ bude pro jednoduchost používat právě tuto metriku.

Pro každé slovo ze vstupu, které se nenachází ve slovníku, tedy:

1. nalezněte všechna slova stejné délky,
2. vyberte ta, která mají minimální Hammingovu vzdálenost od toho vstupního,
3. obsahuje-li seznam slova, která se se vstupem shodují na první pozici, ponechte pouze tato,
4. obdobně na poslední pozici, pak na druhé, předposlední, atd.,
5. ze zbytku vyberte první slovo dle abecedy a toto použijte jako opravu.

Procedura autocorrect má 3 parametry: název souboru s komprimovaným slovníkem (ve formátu gzip), název vstupního souboru a název výstupního souboru, do kterého zapíše opravený text. Níže máte nachystaných několik čistých funkcí, které Vám řešení můžou usnadnit – rozmyslete si, co dělají, a jak je použít.

```
def autocorrect(dict_file: str, input_file: str,
                output_file: str) -> None:
    pass

def hamming(s1: str, s2: str) -> int:
    assert len(s1) == len(s2)

    distance = 0
    s1 = s1.upper()
    s2 = s2.upper()

    for i in range(len(s1)):
        if s1[i] != s2[i]:
            distance += 1

    return distance

def closest_by_hamming(word: str, words: set[str]) -> set[str]:
    res: set[str] = set()
    best: int | None = None

    for curr_word in words:
        distance = hamming(word, curr_word)

        if best is None or distance < best:
            res = set()
            best = distance
        if distance == best:
            res.add(curr_word)

    return res

def closest_by_ends(word: str, candidates: set[str]) -> set[str]:
    for offset in range(len(word) // 2):
        for direction in [-1, 1]:
            idx = direction * offset
            filtered = set()

            for curr_word in candidates:
                if word[idx] == curr_word[idx]:
                    filtered.add(curr_word)
```

```
            if filtered:
                candidates = filtered

    return candidates
```

11.r.5 [language] Jednou z možností, jak poznat v jakém (přirozeném) jazyce je nějaký dokument napsaný, je jednoduchá statistická analýza. Napište funkci, která dostane jako parametr slovník lang_freq a název souboru text_file:

1. lang_freq bude pro každý jazyk obsahovat slovník tvaru { 'a': 357907, 'b': 113756, … } kde hodnota u každého písmene je počet jeho výskytů v nějakém reprezentativním dokumentu,
2. soubor text_file je textový soubor, kterého jazyk chceme určit.

Jazyk určujte tak, že spočítáte frekvence jednotlivých písmen v souboru text_file a srovnáte je s těmi uloženými ve slovníku lang_freq.

Jak nalezneme nejlepší shodu? Informace o frekvenci písmen v nějakém dokumentu lze chápat jako vektory v 26-rozměrném prostoru (resp. více-rozměrném, uvažujeme-li písmena s diakritikou, ale přesná dimenze není podstatná). Za nejpodobnější budeme považovat vektory, které svírají nejmenší úhel. Tento získáte ze vztahu $a \cdot b = |a| \cdot |b| \cdot \cos\theta$ (kde na levé straně je běžný skalární součin, „absolutní hodnoty“ na straně pravé jsou pak délky, které zjistíte ze vztahu $|a|^2 = a \cdot a$).

```
def recognize_language(lang_freq: dict[str, dict[str, int]],
                       text_file: str) -> str:
    pass
```

11.r.6 [weighted] Napište čistou funkci, která vrátí množinu všech slov, tvořených znaky {"0", "1", "2"} s danou délkou length a váhou weight. Váhou myslíme počet nenulových číslic v daném slově.

```
def weighted_words(length: int, weight: int) -> set[str]:
    pass
```

11.v: Volitelné úlohy

11.v.1 [enclosed] V tomto příkladu budeme pracovat s textovými soubory, v nichž nás budou zajímat kulaté, hranaté a složené závorky. Napište funkci count_fully_enclosed, která v případě, že je obsah souboru korektně uzávorkován, vrátí počet nezávorkových znaků, které jsou uzavřeny do všech tří typů závorek. Znaky konce řádku přitom nepočítáme. Není-li obsah souboru korektně uzávorkován, funkce vrátí None.

Příklad: Je-li na vstupu soubor s tímto obsahem:

```
a + (((
b - c) + d)
[{{(x, y)}}])
```

(písmeno a stojí na začátku řádku), pak má funkce vrátit číslo 4, protože jsou zde celkem čtyři nezávorkové znaky, které jsou uzavřeny do všech tří typů závorek (jsou to znaky x, y – za čárkou je mezera).

```
def count_fully_enclosed(filename: str) -> int | None:
    pass
```

11.v.2 [edit] V tomto příkladu budeme pracovat s textovými soubory, které budou obsahovat následující editační značky (dvouznakové; první znak je vždy symbol stříšky ^):

- ^H znamená „smazat předchozí znak“ (pokud žádný předchozí znak není, nestane se nic);
- ^U má význam podle toho, kde se nachází; pokud se nachází na začátku řádku, znamená „vrátit se na konec předchozího řádku“, pokud se nachází jinde, znamená „smazat vše od začátku řádku“;`
- ^W znamená „smazat předchozí slovo na tomto řádku“ (včetně případných mezer, které stojí mezi posledním slovem a značkou ^W; pokud žádné předchozí slovo na aktuálním řádku není, chová se jako ^U).

Slovo zde definujeme jako libovolnou posloupnost nemezerových znaků (tedy např. řetězec "␣␣␣Hello,␣world!␣␣" obsahuje dvě slova – mezery zde zdůrazňujeme znakem ␣). Smíte předpokládat, že se v souboru nevyskytují jiné bílé znaky než mezery a konce řádků.

Napište funkci apply_edit_marks, která přečte soubor s editačními značkami a vrátí řetězec, který vznikne tak, že se všechny úpravy naznačené editačními značkami provedou. Úpravy se provádějí postupně od prvního řádku a zleva doprava, tedy se např. značka ^U může dostat na začátek řádku předchozími úpravami a pak se chová tak, jak se má chovat na začátku řádku. Smíte předpokládat, že se symbol stříšky ^ v souboru nevyskytuje jinde než ve výše uvedených značkách.

Příklad: Je-li na vstupu soubor s tímto obsahem:

```
Hello, world^W^H^H!
How are you tonight?
^U^Wtoday?
Everything is
awesome^U good ^W^H^U okay, i^HI guess.  ^Whope.
```

(první písmeno H stojí na začátku řádku), pak funkce vrátí řetězec:

"Hello!\nHow are you today?\nEverything is okay, I hope.\n"

(\n zde reprezentuje znak konce řádku, jak je nejen v Pythonu obvyklé).

```
def apply_edit_marks(filename: str) -> str:
    pass
```

11.v.3 [paintbot] Představte si robota, který se umí pohybovat rovně dopředu o zadanou celočíselnou délku, otáčet se o 90° v obou směrech a případně za sebou nechávat stopu (tj. označovat místa, přes která jde).

Pozici robota reprezentujeme dvojicí celých čísel; první souřadnice je x-ová (záporná čísla jsou na západ od počátku, kladná na východ), druhá souřadnice je y-ová (záporná čísla jsou na sever, kladná na jih). Na začátku je na souřadnicích (0, 0), je otočen k východu a je ve stavu, že za sebou nezanechává stopu.

Funkce simulate_paintbot přečte ze zadaného souboru seznam instrukcí pro robota a bude je vykonávat do chvíle, než robot při pohybu narazí na vlastní stopu, tj. **vejde** na již označené místo.

Funkce vrátí robotovu poslední pozici (tedy tu, na které narazil na vlastní stopu, nebo tu, kde skončil s vykonáváním poslední instrukce). Předpokládejte, že zadaný textový soubor není prázdný a obsahuje následující typy instrukcí (vždy jedna instrukce na řádku, žádné extra mezery na začátku ani na konci řádku):

- rotate left – robot se otočí o 90° doleva;
- rotate right – robot se otočí o 90° doprava;
- walk k – robot popojde o k jednotek dopředu, kde k je právě jedna římská číslice (tabulka níže; pokud je robot ve stavu, že za sebou zanechává stopu, tak označí všechna místa, kterými projde, včetně toho posledního, kam došel);
- toggle – pokud za sebou robot zanechával stopu, tak odteď nebude; v opačném případě stopu zanechávat začne (počínaje aktuální pozicí).

Zde k může být jedno z:

- I = 1 krok,
- V = 5 kroků,
- X = 10 kroků,
- L = 50 kroků,
- C = 100 kroků,
- D = 500 kroků,
- M = 1000 kroků.

Smíte předpokládat, že celkový počet polí, které robot v průběhu vykonávání instrukcí projde, je menší než milion.

```
def simulate_paintbot(filename: str) -> tuple[int, int]:
    pass
```

Část 12: Opakování

Toto je poslední kapitola hlavní části sbírky. Příklady této kapitoly slouží k procvičení učiva z celého semestru, neobjevují se zde již žádné nové koncepty ani konstrukce.

Elementární příklady:

1. wormhole – práce s ciframi
2. wordwrap – zalomení dlouhých řádků
3. bounds – minimum a maximum ve stromě

Přípravy:

1. lists – práce se seznamem seznamů
2. bowling – výpočet bodování kuželek
3. count – variace na ciferný součet
4. spreadsheet – zpracování souboru s tabulkou čísel
5. wordmask – maskování písmen ve slově
6. composite – hledání vysoce složených čísel

Rozšířené úlohy:

1. walk – procházka čtvercovou mřížkou
2. arraylist – zřetězený seznam polí
3. cycle † – nekonečné proudy
4. stream † – obecné proudy
5. disjoint – nejbližší číslo s úplně jinými ciframi
6. poly – zápis a čtení polynomu z řetězce

12.e: Elementární příklady

12.e.1 [wormhole] Do červí díry spadne seznam kladných celých čísel nums a množina cifer (celá čísla od 0 po 9) allowed. Na druhém konci vypadnou pouze ta čísla, jejichž všechny cifry jsou v množině allowed.

Napište čistou funkci wormhole, která vrátí seznam všech čísel ze seznamu nums, která projdou červí dírou (pořadí zachovejte podle vstupního seznamu).

```
def wormhole(nums: list[int], allowed: set[int]) -> list[int]:
    pass
```

12.e.2 [wordwrap] Napište čistou funkci word_wrap která podle potřeby nahradí mezery ve vstupním řetězci orig za znaky nového řádku, a to tak, aby pro každý řádek platilo, že je buď dlouhý nejvýše max_line_len znaků, nebo neobsahuje žádné mezery.

```
def word_wrap(orig: str, max_line_len: int) -> str:
    pass
```

12.e.3 [bounds]

```
class Tree:
    def __init__(self, value: int, left: 'Tree | None',
                 right: 'Tree | None') -> None:
        self.value = value
        self.left = left
        self.right = right

def leaf(value: int) -> Tree:
    return Tree(value, None, None)
```

Napište čistou funkci get_bounds, která nalezne minimální a maximální hodnotu v zadaném neprázdném stromě.

```
def get_bounds(tree: Tree) -> tuple[int, int]:
    pass
```

12.p: Přípravy

12.p.1 [lists] Napište čistou funkci filter_out_odd, která jako parametr dostane seznam seznamů čísel a vrátí nový seznam seznamů čísel, který vytvoří takto:

- z vnitřních seznamů odstraní lichá čísla, a
- z vnějšího seznamu odstraní (případně i nově vzniklé) prázdné seznamy.

Ostatní prvky v seznamech zůstanou v původním pořadí. Pro vstup `[[1, 5], [1, 2, 3], [], [4, 5, 6]]` tedy funkce vrátí `[[2], [4, 6]]`.

```
def filter_out_odd(list_of_lists: list[list[int]]) -> list[list[int]]:
    pass
```

Dále napište čistou funkci without_middle_occurrence, která dostane jako parametr seznam čísel values a hledané číslo value a vrátí seznam bez prostředního výskytu hledaného čísla. Vyskytuje-li se hledané číslo v zadaném seznamu sudý počet krát, bereme jako prostřední ten blíže začátku, tedy např. pro vstup `([2, 2, 3, 2, 2], 2)` funkce vrátí `[2, 3, 2, 2]`. (Pokud seznam hledané číslo neobsahuje, vraťte původní seznam nebo jeho kopii.)

```
def without_middle_occurrence(values: list[int], value: int) -> list[int]:
    pass
```

12.p.2 [bowling] Napište funkci bowling_score, která spočítá celkové skóre bowlingové hry, přičemž počty shozených kuželek jsou v seznamu rolls (předpokládejte, že tento seznam obsahuje validní hody a že je dostatečně dlouhý). Skóre v bowlingu se počítá takto: Hraje se na 10 kol, v každém kole se háže až dvakrát, kromě posledního, kde se za určitých okolností háže třikrát. Pokud hned prvním hodem kola dosáhne hráč 10 bodů (**strike**), podruhé už neháže a do skóre se mu započítá 10 plus hodnoty dvou dalších **hodů**. Pokud v součtu obou hodů dosáhne hráč 10 bodů (**spare**), do skóre se mu započítá 10 plus hodnota jednoho dalšího **hodu**. V ostatních případech se do skóre započítá součet obou hodů kola. Pokud hráč zahrál strike v posledním kole, háže ještě dvakrát. Pokud hráč zahrál spare v posledním kole, háže ještě jednou.

Příklad: Pro vstup `[10, 10, 3, 6, 4, 5, 9, 1, 7, 3, 10, 0, 1, 10, 3, 7, 10]` funkce vrátí **149**; pro vstupní seznam obsahující dvanáctkrát **10** funkce vrátí **300**.

Vysvětlení prvního příkladu:

1. kolo: **strike**, počítá se $10 + 10 + 3 = 23$ bodů
2. kolo: **strike**, počítá se $10 + 3 + 6 = 19$ bodů
3. kolo: $3 + 6 = 9$ bodů
4. kolo: $4 + 5 = 9$ bodů
5. kolo: **spare**, počítá se $9 + 1 + 7 = 17$ bodů
6. kolo: **spare**, počítá se $7 + 3 + 10 = 20$ bodů
7. kolo: **strike**, počítá se $10 + 0 + 1 = 11$ bodů
8. kolo: $0 + 1 = 1$ bod
9. kolo: **strike**, počítá se $10 + 3 + 7 = 20$ bodů
10. kolo: **spare**, háže se tedy ještě jednou a počítá se $3 + 7 + 10 = 20$ bodů.

Celkem **149** bodů.

Rozdělení hodů do jednotlivých kol pro názornost:

10	10	3 6	4 5	9 1	7 3	10	0 1	10	3 7 10

Vysvětlení druhého příkladu:

V každém kole padne **strike**, počítá se tedy $10 + 10 + 10 = 30$ bodů. V posledním kole rovněž padne **strike**, háže se tedy ještě dvakrát a počítá se opět $10 + 10 + 10 = 30$ bodů. Dohromady tedy **10** kol po **30** bodech, což je **300** bodů. Rozdělení hodů do jednotlivých kol pro názornost:

10	10	10	10	10	10	10	10	10	10 10 10

```
def bowling_score(rolls: list[int]) -> int:
    pass
```

12.p.3 [count] Napište funkci count_seq, která nad desítkovou reprezen-

tací nezáporného celého čísla num provede následující výpočet:

1. vybere všechny cifry, po kterých následuje alespoň seq stejných cifer; pro účely této kontroly chápeme num cyklicky, tzn. po poslední cifře následuje opět první,
2. vybrané cifry sečte a součet vrátí.

Cykličnost v bodě 1 můžeme chápat jako nekonečné opakování num, např. v čísle 123 následují po cifře 2 cifry 3, 1, 2, 3, 1, atd.

Příklady výpočtu:

- pro num=111222 a seq=2 je výsledkem 3 (1 + 2), protože po první (1) a čtvrté (2) cifře následují 2 stejné cifry,
- pro num=1111 a seq=1 je výsledkem 4, protože po každé cifře následuje alespoň jedna stejná cifra,
- pro num=1234 a seq=0 je výsledkem součet všech číslic, totiž 10.

```
def count_seq(num: int, seq: int) -> int:
    pass
```

12.p.4 [restore] Napište čistou funkci restore_sequence, která dostane neprázdný řetězec složený pouze z číslic 0 a 1 a vrátí množinu všech možných řetězců, které vzniknou doplněním znaků čárky ',' do původního řetězce tak, aby části jimi oddělené byly dvojkové zápisy čísel v intervalu od low do high včetně. Hodnota low bude vždy alespoň 1. Rozdělení musí být takové, že žádný zápis neobsahuje levostranné nuly.

```
def restore_sequence(digits: str, low: int, high: int) -> set[str]:
    pass
```

12.p.5 [wordmask] Napište čistou funkci wordmask, která vypočte všechny možnosti zamaskování slova word. Slovo zamaskujete aplikováním masky mask, tj. na každý znak slova se aplikuje korespondující znak masky. Je-li maska kratší než slovo, aplikuje se cyklicky.

Například pro slovo **abababa** a masku **XX?** je situace následovná (odpovídající písmena jsou pod sebou):

```
abababa
XX?XX?X
```

Maska je složena ze 2 znaků, X a ?:

1. obsahuje-li maska na dané pozici znak X, odpovídající znak slova se nemění,
2. naopak, je-li na dané pozici znak ?, odpovídající znak ve slově se zamaskuje některým znakem ze seznamu alternatives.

Funkce wordmask pak vrátí seznam všech slov (v libovolném pořadí), které mohou tímto postupem vzniknout.

Například pro slovo abababa, masku XX? a seznam alternativ ['x', 'y'] bude výsledkem maskování některá permutace seznamu ['abxbaxa', 'abybaxa', 'abxbaya', 'abybaya'].

```
def wordmask(word: str, mask: str, alternatives: list[str]) -> list[str]:
    pass
```

12.p.6 [composite] Napište čistou funkci highly_composite, která dostane na vstupu množinu přirozených čísel a vrátí množinu těch z nich, která jsou vysoce složená relativně k původní množině. Přirozené číslo je vysoce složené, má-li striktně víc dělitelů (a to včetně těch, které v zadané množině nejsou), než libovolné menší číslo ze zadané množiny.

```
def highly_composite(numbers: set[int]) -> set[int]:
    pass
```

12.r: Řešené úlohy

12.r.1 [walk] V této úloze budeme implementovat simulaci procházky po 2D mřížce. Pro reprezentaci pozice v mřížce budeme používat uspořádanou dvojici (x, y).

```
Position = tuple[int, int]
```

Cesta procházky je zadaná jako řetězec path, který se skládá z příkazů ← / → pro pohyb doleva a doprava (po ose x) a ↑ / ↓ pro pohyb nahoru a dolů (po ose y). Souřadnice rostou ve směru doprava na x-ové ose a nahoru na y-ové ose.

Napište čistou funkci walk, která vrátí finální pozici pro procházku path z počáteční pozice start.

```
def walk(path: str, start: Position) -> Position:
    pass
```

Dále napište čistou funkci meet, která vrátí pro dvojici cest path_1, path_2 a počátků start_1 a start_2, první pozici na které se procházky potkají. Procházky se provádí synchronně, tj. kroky se vykonávají najednou pro obě procházky. Pokud se procházky nepotkají, funkce vrátí None.

```
def meet(path_1: str, path_2: str, start_1: Position,
         start_2: Position) -> Position | None:
    pass
```

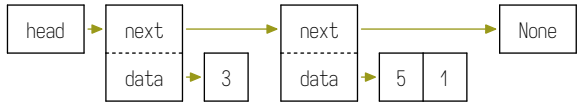
12.r.2 [arraylist] V této úloze budeme programovat jednoduše zřetězený seznam, který si v každém uzlu udržuje seznam hodnot data maximální délky capacity. Jinak je zřetězený seznam definován tak, jak jej už znáte:

```
class Node:
    def __init__(self) -> None:
        self.data: list[int] = []
        self.next: 'Node | None' = None

class ArrayList:
    def __init__(self, capacity: int) -> None:
        self.capacity = capacity
        self.head: Node | None = None
        self.tail: Node | None = None
```

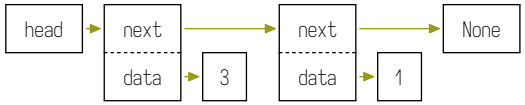
Napište metodu append, která vloží hodnotu value na konec posledního uzlu, není-li plný, jinak vytvoří nový uzel na konci seznamu.

```
def append(self, value: int) -> None:
    pass
```

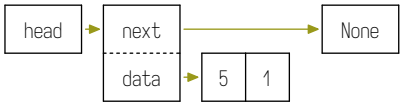
Napište metodu delete, která smaže první výskyt hodnoty value ze seznamu. Pokud by po smazání nastalo, že zůstane v seznamu prázdny uzel, smaže se i ten. Například mějme následující seznam:



Po smazání hodnoty 5 bude výsledný seznam vypadat následovně:

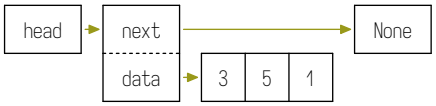


Naproti tomu smazáním hodnoty 3 z původního seznamu vznikne prázdny uzel, který se smaže:

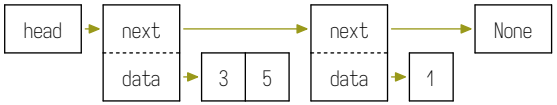



```
def delete(self, value: int) -> None:
    pass
```

Konečně napište metodu compact, která maximalizuje využití kapacity uzlů: přesune prvky v seznamu tak, aby se uzly v seznamu odpředu zaplnily. Přebytečné prázdné uzly metoda smaže. Ve výsledném seznamu zachovejte vzájemné pořadí prvků. Například kompaktní reprezentace pro seznam z předchozího příkladu a kapacitu 3 je:



Výsledek pro seznam s kapacitou 2 je:




```
def compact(self) -> None:
    pass
```

12.r.3 [cycle] † Obecný proud je datová struktura podobná seznamu, která je potenciálně nekonečná, ale funguje přitom i v programovacích jazycích se striktním vyhodnocováním. V tomto příkladu se omezíme na nekonečné cyklické proudy. Do třídy Stream si doplňte potřebné atributy. Metoda get z proudu vybere další prvek (tzn. odstraní první prvek a vrátí jej).

```
class Stream:
    def __init__(self, data: list[int]) -> None:
        pass

    def get(self) -> int:
        pass
```

Čistá funkce cycle ze seznamu (který je konečný) vytvoří proud (který je nekonečný), a to tak, že pomyslně zřetězí nekonečně mnoho kopií tohoto seznamu za sebe.

```
def cycle(data: list[int]) -> Stream:
    pass
```

Čistá funkce drop odstraní ze vstupního proudu n počátečních prvků a vrátí výsledný proud.

```
def drop(n: int, original: Stream) -> Stream:
    pass
```

Čistá funkce take dostane na vstupu (nekonečný) proud a vytvoří z něj konečný seznam, a to tak, že vybere prvních n prvků.

```
def take(n: int, original: Stream) -> list[int]:
    pass
```

Čistá funkce every_nth vytvoří proud, který vznikne z toho vstupního tak, že vždy jeden prvek zachová a pak $n - 1$ prvků přeskočí. Jinými slovy, vyberete ze vstupního proudu ty prvky, které jsou na pozicích dělitelných n.

```
def every_nth(n: int, original: Stream) -> Stream:
    pass
```

12.r.4 [stream] † V tomto příkladě pokračujeme proudy. Tentokrát budou proudy obecné: mohou být jak konečné tak nekonečné, a nemusí být cyklické. Protože v obecném případě nelze proud uložit celý, musíme datovou strukturu naprogramovat tak, aby potřebný výpočet proběhl až ve chvíli, kdy se pokusíme z proudu vybrat další prvek.

To zabezpečíme tak, že každá transformace proudu bude samostatná třída, která si bude pamatovat odkaz na vnitřní proud (t.j. ten, který transformuje) a podle potřeby z něj bude vybírat prvky.

Protože všechny tyto třídy mají metodu take_head, obecný proud lze reprezentovat jako instanci libovolné z těchto tříd.

Definici typu 'Stream' naleznete níže.

Třída FinStream bude reprezentovat konečný proud, který vznikl ze seznamu konverzní funkcí to_stream. Ostatní třídy reprezentují transformace popsané níže u příslušných funkcí.

```
class FinStream:
    def __init__(self, data: list[int]) -> None:
        pass
```

Metoda take_head vrátí dvojici, kde první složka je první prvek proudu (existuje-li) a druhá složka reprezentuje proud, který vznikne odstraněním prvního prvku.

```
    def take_head(self) -> tuple[int | None, 'Stream']:
        pass

class Cycle:
    def __init__(self, inner: 'Stream') -> None:
        pass

    def take_head(self) -> tuple[int | None, 'Stream']:
        pass

class Drop:
    def __init__(self, n: int, inner: 'Stream') -> None:
        pass

    def take_head(self) -> tuple[int | None, 'Stream']:
        pass

class Take:
    def __init__(self, n: int, inner: 'Stream') -> None:
        pass

    def take_head(self) -> tuple[int | None, 'Stream']:
        pass

class Skip:
    def __init__(self, inner: 'Stream') -> None:
        pass

    def take_head(self) -> tuple[int | None, 'Stream']:
        pass

Stream = FinStream | Cycle | Drop | Take | Skip
```

Čistá funkce, která vytvoří konečný proud z dat zadaných v seznamu.

```
def to_stream(data: list[int]) -> Stream:
    pass
```

Čistá funkce, která vytvoří nekonečný proud, a to tak, že bude vybírat prvky z vnitřního proudu, dokud to lze. V případě, že prvky dojdou (vstupní proud byl konečný), výstupní proud se vrátí na začátek toho vstupního a toto bude dále opakovat (libovolně dlouho).

```
def cycle(inner: Stream) -> Stream:
    pass
```

Čistá funkce, která vytvoří nový proud tím, že zahodí prvních n prvků toho vstupního.

```
def drop(n: int, original: Stream) -> Stream:
    pass
```

Čistá funkce, která z libovolně dlouhého vstupního proudu vytvoří konečný proud o nejvýše n prvcích.

```
def take(n: int, original: Stream) -> Stream:
    pass
```

Čistá funkce, která vytvoří proud, který se bude chovat následovně: první prvek vybere z proudu data, pak dalších n prvků přeskočí, kde n je hodnota vybraná z proudu skips. Toto bude opakovat, dokud budou v data nějaké prvky. Dojdou-li v skips hodnoty, výsledný proud nebude dále nic přeskakovat.

```
def skip(data: Stream, skips: Stream) -> Stream:
    pass
```

12.r.5 [disjoint] Napište čistou funkci nearest_disjoint, která pro vstup n nalezne číslo m takové, že:

- množiny cifer použitých v m a n jsou disjunktní,
- $|m - n|$ je nejmenší možné.

```
def nearest_disjoint(n: int) -> int | None:
    pass
```

12.r.6 [poly] S polynomy jsme se už setkali dvakrát, v kapitolách 5 a 7. Ještě jednou si připomeňme, jak polynomy vypadají:

$$P(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0 = \sum_0^n a_i x^i$$

Tentokrát budeme pracovat s řetězcovou reprezentací polynomů, která

vypadá jako výše uvedený zápis, pouze místo a_i bude obsahovat konkrétní koeficienty. Pro lepší čitelnost budeme navíc požadovat, aby byly záporné koeficienty v řetězci zapsané jako $5x^2 - 7x$, nikoliv jako $5x^2 + -7x$. Vaším úkolem je napsat dvojici funkcí: poly_to_str, která převede seznam koeficientů na řetězec a str_to_poly která realizuje opačnou konverzi. Koeficienty budou v seznamech v pořadí a_i na indexu $n - i$.

```
def poly_to_str(poly: list[int]) -> str:
    pass

def str_to_poly(string: str) -> list[int]:
    pass
```

12.v: Volitelné úlohy

12.v.1 [ast] Máte připraveny třídy, které budou tvořit AST (abstraktní syntaktický strom) velmi jednoduchého programu:

- Arithmetic reprezentuje binární aritmetickou operaci; její objekty mají atributy op (jeden z řetězců '+', '-', '*', '/'), left (levý operand), right (pravý operand).
- Assignment reprezentuje přiřazení; její objekty mají atributy var (řetězec, jméno proměnné na levé straně přiřazení) a rhs (pravá strana přiřazení).

Dále je připraven typový alias Expression, který reprezentuje uzel stromu výrazu – buď číslo typu int nebo řetězec (reprezentuje proměnnou) nebo objekt typu Arithmetic. Výše uvedené atributy left, right a rhs jsou typu Expression.

Tyto třídy ani typový alias Expression nijak nemodifikujte.

```
class Arithmetic:
    def __init__(self, op: str, left: 'Expression',
                 right: 'Expression'):
        self.op = op
        self.left = left
        self.right = right

Expression = Arithmetic | str | int

class Assignment:
    def __init__(self, var: str, rhs: Expression):
        self.var = var
        self.rhs = rhs
```

Napište čistou funkci, která dostane na vstupu jednoduchý program ve formě seznamu přiřazení a vrátí slovník reprezentující hodnoty proměnných na konci programu. Pokud během vykonávání programu dojde k chybě (dělení nulou nebo použití proměnné, které předtím nebyla přiřazena hodnota), funkce vrátí None. Dělení je vždy celočíselné (i když je reprezentováno znakem /).

```
def execute(program: list[Assignment]) -> dict[str, int] | None:
    pass
```

12.v.2 [mem] Mějme jednoduchý programovací jazyk, jehož (jednoznakové) instrukce se vyhodnocují nad neomezenou pamětí. Paměť indexujeme celými čísly, přičemž každá paměťová buňka drží jedno celé číslo; na začátku obsahují všechny buňky v paměti číslo 0. V průběhu vykonávání programu si pamatujeme **index aktuální buňky**; na začátku je to 0. Instrukce jazyka jsou následující:

- '<' – snížíme *index aktuální buňky* o 1;
- '>' – zvýšíme *index aktuální buňky* o 1;
- '+' – zvýšíme hodnotu aktuální buňky o 1;
- '-' – snížíme hodnotu aktuální buňky o 1;
- '[' – je-li hodnota aktuální buňky rovna nule, skočíme **za** odpovídající znak ']';
- ']' – skočíme **na** odpovídající znak '['.

O programu předpokládáme, že je vzhledem ke znakům '[' a ']' dobře uzávorkovaný. Není-li výše řečeno jinak, po provedení instrukce se přesuneme na instrukci následující. Program končí ve chvíli, kdy by další provedená instrukce měla ležet za jeho koncem.

Provedení každé jednotlivé instrukce by nemělo trvat příliš dlouho (ideálně by mělo být skoro konstantní; zejména by nemělo záviset na délce programu). Je v pořádku si něco předpočítat, než začnete provádět instrukce programu.

Napište čistou funkci execute, která vyhodnotí zadaný program a vrátí obsah paměťových buněk jako slovník. Při testování ignorujeme paměťové buňky, které obsahují hodnotu 0, tedy např. slovníky {1: 0, 2: 3} a {2: 3} jsou z hlediska testů ekvivalentní.

```
def execute(program: str) -> dict[int, int]:
    pass
```

12.v.3 [column] Tabulkové procesory často pro označení sloupců používají znaky anglické abecedy, přičemž po vyčerpání 26 možností A až Z se pokračuje AA, AB, ..., ZZ, AAA, AAB, ...

Čistá funkce spreadsheet_column dostane jako parametr index sloupce (nezáporné celé číslo, indexujeme od 0) a vrátí řetězec příslušný danému sloupci. Indexu 2 tedy odpovídá řetězec "C", indexu 27 řetězec "AB", indexu 16383 řetězec "XFD".

Funkce musí rozumně rychle fungovat pro libovolně velká čísla.

```
LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def spreadsheet_column(index):
    pass
```

Část S.3: Sada úloh k třetímu bloku

Ve třetím bloku jsou následující domácí úkoly:

- a_minesweeper – hra Minesweeper (hledání min),
- b_family_tree – práce se stromovou strukturou rodokmene,
- c_navigate – opět simulace pohybu robota (ovšem tentokrát s možností volby cesty na křižovatkách),
- d_alphametics – řešení slovně-početních hádanek,
- e_nonogram – řešení malovaných křížovek,
- f_numberlink – řešení hry Numberlink.

Ve všech těchto úkolech se vám může hodit rekurze, i když některé z nich mohou být rozumně řešitelné i bez ní.

S.3.a: minesweeper

I v této sadě si naprogramujete jednu hru, a bude jí **Minesweeper**[26]. Naše verze bude trochu modifikovaná, zejména kliknutí na minu nebude nutně znamenat konec hry, ale způsobí výbuch, který poškodí část herní plochy. (Každá mina bude mít přiřazenu tzv. „sílu“ určující, kolik okolních políček bude zasaženo.)

Abyste si hru mohli vyzkoušet (poté, co implementujete všechny níže uvedené metody), máte opět k dispozici soubor game_minesweeper.py, který spusťte ze stejného adresáře, jako je soubor s vaším řešením. Na začátku souboru jsou konstanty, jejichž úpravou můžete změnit velikost herní plochy, počet min a vzhled hry.

Třída Minesweeper, kterou máte implementovat, reprezentuje stav hry, tj. obsah herní plochy, pozici min a aktuální skóre. Interní detaily jsou na vás, nicméně očekáváme, že objekty této třídy budou mít alespoň tyto dva atributy:

- status – 2D seznam (seznam seznamů – řádků) reprezentující stav hry; prvky vnitřních seznamů jsou těchto hodnot (UNKNOWN, EXPLODED, DESTROYED jsou celočíselné konstanty definované níže):
  - UNKNOWN představuje dosud neodkryté (a nezničené) políčko,
  - EXPLODED představuje vybuchlou minu,
  - DESTROYED představuje políčko zničené výbuchem,
  - 0 až 8 představují odkryté políčko s informací o počtu sousedících min.
- score – počet bodů (celé číslo); body se udělují takto:
  - +1 bod za každé odkrytí políčka bez miny,
  - -10 bodů za každou vybuchlou minu.

[26] https://en.wikipedia.org/wiki/Minesweeper_(video_game)

Kliknutí na některé políčko herní plochy bude zpracováno metodou uncover (viz níže). Je-li již políčko odkryté nebo zničené výbuchem, tato metoda nemá žádný efekt. V opačném případě se políčko odkryje a nastane jeden z těchto případů:

- Je-li na tomto políčku mina, vybuchne a všechna políčka ve vzdálenosti menší nebo rovné síle miny budou zničena. Pokud na některém z těchto políček byla dosud nevybuchlá mina, rovněž vybuchne. To může zničit další políčka a tento proces se může opakovat (i vícekrát). Políčka, kde vybuchla mina, se označí stavem EXPLODED, ostatní zničená políčka se označí stavem DESTROYED. (Stav EXPLODED na herní ploše zůstává a nemění se na DESTROYED ani při dalším výbuchu.)
- V opačném případě se stav políčka nastaví na 0 až 8 podle počtu min (i těch už vybuchlých) v bezprostředním okolí. Je-li stav 0, odkryjí se všechna okolní políčka, což se opět může vícekrát opakovat.

Pojmy „okolí“ a „vzdálenost“ zde chápeme ve všech osmi směrech (tedy i diagonálně). Vybuchlá mina se silou 1 tedy zničí až osm políček, vybuchlá mina se silou 2 zničí až 24 políček atd.

Souřadnice zde používáme opět ve tvaru (sloupec, řádek), přičemž sloupce číslujeme od 0 zleva a řádky od 0 shora.

Hodnoty níže uvedených konstant neměňte.

```
Position = tuple[int, int]

UNKNOWN = -1
EXPLODED = -2
DESTROYED = -3

class Minesweeper:
```

Po inicializaci mají být všechna pole herní plochy neodkrytá, herní plocha má mít rozměry zadané parametry width a height a skóre má být 0. Parametr mines určuje pozici min (klíče slovníku) a jejich sílu (hodnoty slovníku). Slovník mines nijak nemodifikujte. Pokud si ho hodláte někam uložit, tak buďto zařiďte, aby se ani později nemodifikoval, nebo si vytvořte jeho kopii.

```
    def __init__(self, width: int, height: int,
                 mines: dict[Position, int]):
        pass
```

Metoda uncover provede odkrytí políčka dle popisu výše a případně upraví skóre. Předpokládejte, že souřadnice jsou validní (tj. v rozsahu herní plochy).

```
    def uncover(self, x: int, y: int) -> None:
        pass
```

S.3.b: navigate

Vrátíme se k robotovi, jehož pohyb jsme simulovali ve druhé sadě úkolů (b_robot). Budeme mít opět stejný plán ve tvaru neomezené čtvercové sítě s čtvercovými dílky s nákresy ulic či křižovatek. Tentokrát ovšem dáme robotovi možnost se pohybovat libovolným směrem podle možností na aktuálním dílku.

```
Heading = int
NORTH, EAST, SOUTH, WEST = 0, 1, 2, 3
Tile = set[Heading]
Position = tuple[int, int]
Plan = dict[Position, Tile]
```

Implementujte čistou funkci navigate, která vrátí cestu, kterou se robot dostane ze zadané startovní do zadané cílové pozice na zadaném plánu. Pokud žádná taková cesta neexistuje, funkce vrátí None. Vrácená cesta je ve formě seznamu všech pozic, kterými robot projde od startovní do cílové pozice včetně. Předpokládejte, že plán je korektní, tj. splňuje predikát is_correct z úlohy s2/b_robot, a že zadané pozice jsou na některém z položených dílků.

Doporučení: Použijte princip backtrackingu. Budete muset nějak zařídit, aby robot neběhal v kruzích (pak by vaše funkce nemusela skončit).

```
def navigate(plan: Plan, start: Position, goal: Position) \
        -> list[Position] | None:
    pass
```

S.3.c: alphametics

Slovní aritmetika[27] (někdy též cryptarithm nebo algebrogram) je matematický hlavolam zadaný jako rovnice se slovy, např. „SEND + MORE = MONEY“. Cílem je přiřadit každému písmenu unikátní[28] číslici tak, aby po jejich nahrazení rovnost platila. Přitom zápis žádného z čísel nesmí začínat nulou. V tomto konkrétním případě (a v desítkové soustavě) je jediné možné řešení, a to $S \to 9$, $E \to 5$, $N \to 6$, $D \to 7$, $M \to 1$, $O \to 0$, $R \to 8$, $Y \to 2$. Po tomto nahrazení číslicemi skutečně platí $9567 + 1085 = 10652$.

Cílem této úlohy je napsat čistou funkci, která podobné hlavolamy řeší,

[27] https://en.wikipedia.org/wiki/Verbal_arithmetic

[28] „Unikátní“ znamená, že dvě různá písmena nemohou mít přiřazenu stejnou číslici.

a to v zadané poziční soustavě (základem bude vždy celé číslo mezi 2 a 26 včetně). Omezíme se přitom pouze na sčítání, jiné aritmetické operace neuvažujeme. Rovnice na vstupu je zadána dvěma parametry. Levá strana rovnice lhs je seznam (alespoň dvou) slov, přičemž každé slovo je dáno jako seznam písmen (jednoznakových řetězců). Pravá strana rovnice je pak je vždy právě jedno slovo (seznam písmen).

Funkce vrátí slovník, který každému písmenu hlavolamu přiřazuje unikátní hodnotu číslice. Pokud existuje více řešení, funkce vrátí libovolné z nich. Pokud neexistuje žádné řešení, funkce vrátí None.

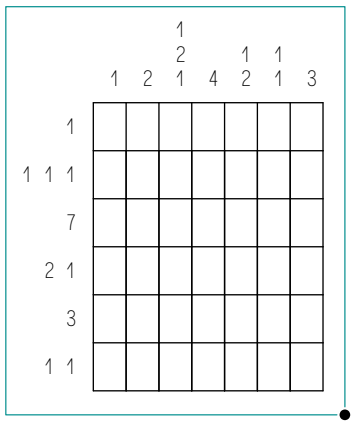
Nápověda: Použijte techniku backtrackingu. Vzpomeňte si na svá základoškolská léta – zejména na sčítání pod sebou, které začíná vždy zprava. I zde se k řešení hodí postupně zkoušet přiřazovat hodnoty číslicím, které jsou u jednotlivých sčítanců co nejvíce vpravo, a rekurzi včas ukončit, když už je jasné, že výsledku není možno dosáhnout.

```
def solve(lhs: list[list[str]], rhs: list[str], base: int) \
        -> dict[str, int] | None:
    pass
```

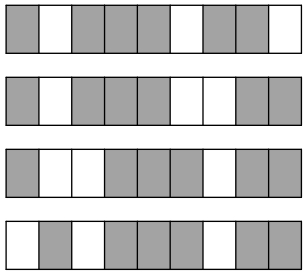
S.3.d: nonogram

Malované křížovky[29] (nonogramy) jsou logické hlavolamy, u kterých je cílem vybarvit některá políčka čtvercové sítě podle zadané číselné legendy. Výsledkem je typicky jednoduchý obrázek. Existují různé druhy malovaných křížovek, v této úloze nás budou zajímat pouze ty základní černobílé.

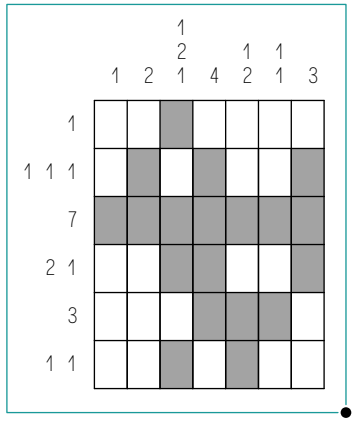
Zadání malované křížovky vypadá např. takto:



Číselná legenda u řádků a sloupců ukazuje, kolik políček máme v dané řadě (řádku nebo sloupci) vybarvit a jak mají být vybarvená políčka seskupena. Pokud bychom tedy například měli legendu **1 3 2** a řádek délky 9 políček, pak jej můžeme vyplnit jedním z těchto způsobů:



Řešením malované křížovky je vybarvení políček takové, že každý řádek a každý sloupec odpovídá zadané legendě. Výše uvedený příklad má tedy následující (jediné) řešení:



V této úloze si zkusíte napsat program, který bude schopen některé jednodušší malované křížovky řešit pomocí techniky backtrackingu. Jednotlivá políčka křížovky budeme reprezentovat typem Pixel, což je zde typový alias pro int použitý pouze pro lepší čitelnost anotací.

```
Pixel = int
EMPTY, FULL, UNKNOWN = 0, 1, 2
```

Máme zde připravené globální konstanty EMPTY (reprezentuje prázdné políčko), FULL (reprezentuje vybarvené políčko), UNKNOWN (reprezentuje neznámý stav políčka). Počet různých druhů políček si můžete pro účely implementace případně rozšířit, ale tyto tři konstanty zachovejte.

Dále máme připraven typový alias pro číselnou legendu. Legenda pro řádky bude v seznamu uložená zleva doprava, legenda pro sloupce shora dolů.

```
Clue = list[int]
```

Nakonec je připravena třída Picture, která bude reprezentovat výsledný obrázek. Tuto třídu můžete libovolně upravovat (přidávat vlastní atributy a metody), ale zachovejte parametry metody __init__ i způsob inicializace atributů height, width a pixels.

```
class Picture:
    def __init__(self, height: int, width: int):
        self.height = height
        self.width = width
        self.pixels = [[UNKNOWN for _ in range(width)]
                       for _ in range(height)]
```

Nejprve implementujte čistou funkci gen_lines_with_prefix, která vrátí seznam všech řad délky size, které odpovídají zadané legendě (clue) a zároveň začínají zadaným prefixem (prefix). Předpokládejte, že prefix má délku nejvýše size a obsahuje pouze hodnoty EMPTY a FULL. Na pořadí seznamů uvnitř vnějšího seznamu nezáleží.

Nápověda: Využijte backtracking. Zkuste začít implementací pro situace, kdy je prefix prázdný, a tuto implementaci pak rozšiřte.

```
def gen_lines_with_prefix(clue: Clue, size: int,
                          prefix: list[Pixel]) -> list[list[Pixel]]:
    pass
```

Dále implementujte čistou funkci solve, která najde řešení malované křížovky se zadanou legendou. Pokud žádné řešení neexistuje, vrátí None. Pokud existuje více než jedno řešení, vrátí libovolné z nich.

Nápověda: Využijte backtracking. Použijte funkci gen_lines_with_prefix. Začněte v levém horním rohu. Střídejte řádky a sloupce. V testech budeme používat jen takové vstupy, které se tímto přístupem dají dostatečně rychle vyřešit.

```
def solve(rows: list[Clue], cols: list[Clue]) -> Picture | None:
    pass
```

S.3.e: numberlink

Numberlink[30] je logický hlavolam, v němž je zadána čtvercová síť s několika dvojicemi čísel a cílem je spojit všechny dvojice stejných čísel lomenou čarou, přičemž každým políčkem čtvercové sítě musí procházet právě jedna čára. V naší implementaci místo kreslení čar do čtvercové sítě vepíšeme čísla všude tam, kudy by spojnice zadaných čísel prošla.

[29] https://en.wikipedia.org/wiki/Nonogram

[30] https://en.wikipedia.org/wiki/Numberlink

Příklad vstupu:

			4			
	3			2	5	
			3	1		
			5			
		1				
2				4		

a řešení:

2	2	2	4	4	4	4
2	3	2	2	2	5	4
2	3	3	3	1	5	4
2	5	5	5	1	5	4
2	5	1	1	1	5	4
2	5	1	5	5	5	4
2	5	5	5	4	4	4

			4	∘	∘	∘
	3			2	5	∘
	∘	∘	3	1		∘
			5	∘		∘
		∘	∘	∘		∘
		1				∘
2				4	∘	∘

(Srovnejte s obrázkem na Wikipedii.)

Máme připravené typové aliasy Grid pro 2D seznamy, Position pro dvojice souřadnic (sloupec, řádek; číslujeme jako obvykle od nuly zleva a shora) a Ends, jehož význam je vysvětlen níže.

```
Grid = list[list[int]]
Position = tuple[int, int]
Ends = dict[tuple[int, bool], Position]
```

Nejprve implementujte čistou funkci get_ends, která dostane na vstup zadání hlavolamu jako 2D seznam, který obsahuje pouze nezáporná celá čísla, přičemž nuly reprezentují prázdná políčka a ostatní čísla ve vstupu jsou vždy přesně dvakrát. Funkce vrátí slovník typu Ends, v němž jsou pro každé kladné číslo n ze vstupu dvě položky: (n, True): (x_1, y_1) a (n, False): (x_2, y_2), kde (x_1, y_1) a (x_2, y_2) jsou souřadnice výskytu daného čísla. Na tom, které souřadnice jsou u položky s True a s False, nezáleží. True, False zde používáme jenom proto, abychom mohli mít dvě různé položky pro každé číslo. (Proč volíme zrovna takovou reprezentaci, je vysvětleno níže.)

```
def get_ends(grid: Grid) -> Ends:
    pass
```

Dále implementujte čistou funkci solve, která najde řešení pro zadaný vstup. Pokud žádné řešení neexistuje, vrátí None. Pokud existuje více než jedno řešení, vrátí libovolné z nich.

Nápověda: Využijte backtracking. Spočítejte si nejdříve pozice čísel pomocí funkce get_ends. Na tyto pozice se můžete dívat jako na dva konce provázku, které se snažíte dostat k sobě a spojit. V každém kroku backtrackingu si zvolte jeden z „konců“ a pokuste se jej posunout – možné směry posunutí jsou právě ty lokální volby, které při backtrackingu provedete. Přitom je vhodné volit z možných konců takový, který má co nejméně těchto možných směrů. Kromě posouvání konců si zároveň chcete zaznamenat, která políčka už jsou obsazena.

```
def solve(grid: Grid) -> Grid | None:
    pass
```

Poznámka k volbě typu Ends pro reprezentaci „konců provázků“: Mnozí by jistě mohli navrhnout, že mít ve dvojicích klíčů arbitrární hodnoty True a False je zbytečné a že by se slovník „konců“ dal napsat jinak (např. s typem dict[int, tuple[Position, Position]]). Zde zvolený typ má ale jistou symetrii, která je výhodná pro implementaci funkce solve. Ke všem „koncům“ se totiž chováme stejně, a tedy kód pro nalezení jednoho konkrétního (toho s nejméně možnostmi pohybu) stejně jako kód pro jeho posunutí můžeme napsat obecně a nemusíme u toho rozebírat více různých případů.

K.1: Týden 1

K.1.e.1 [divisors]

```
def divisors(number):
    count = 1  # number always divides itself
    divisor = 1
    maximal = number // 2
    while divisor <= maximal:
        if number % divisor == 0:
            count += 1
        divisor += 1
    return count

def verity(student):
    from quick import assert_eq
    for num in range(1, 3000):
        assert_eq(divisors, student, num)
```

K.1.e.2 [powers]

```
import run_tests  # noqa
def powers(n, k):
    result = 0
    for i in range(n):
        result += (i + 1) ** k
    return result

def verity(student):
    from qold import assert_eq, fixed, mixed, nat, nat1
    for n, k in mixed().stream(nat1, nat).take(10000):
        assert_eq(powers, student, fixed(n, k))
```

K.1.e.3 [multiples]

```
import run_tests  # noqa
def sum_of_multiples(n):
    result = 0
    for i in range(n + 1):
        if i % 3 == 0 or i % 5 == 0:
            result += i
    return result

def verity(student):
    from quick import assert_eq
    for num in range(1, 3000):
        assert_eq(sum_of_multiples, student, num)
```

K.1.r.1 [even]

```
import run_tests  # noqa
def even(n):
    result = 0
    for i in range(n):
        number = 2 * (i + 1)
        result += number ** 2
    return result
```

K.1.r.2 [prime]

```
import run_tests  # noqa
def is_prime(number):
    if number < 2:
        return False
    divisor = 2
    while divisor ** 2 <= number:
        if number % divisor == 0:
            return False
        divisor += 1
    return True
```

K.1.r.3 [coins]

```
import run_tests  # noqa
def coins(value):
    result = 0

    result += value // 5
    value %= 5

    result += value // 2
    value %= 2

    return result + value
```

K.1.r.4 [fibfibsum]

```
import run_tests  # noqa
def fibfibsum(count):
    result = 0

    index_a = 1
    index_b = 1

    a = 1
    b = 1
    i = 1

    for _ in range(count):
        while i < index_a:
            c = a + b
            a = b
            b = c
            i += 1
        result += a

        index_c = index_a + index_b
        index_a = index_b
        index_b = index_c
    return result
```

K.1.r.5 [abundant]

```
import run_tests  # noqa
def sum_divisors(number):
    result = 0
    for i in range(1, number // 2 + 1):
        if number % i == 0:
            result += i
    return result

def is_abundant(number):
    return sum_divisors(number) > number
```

K.1.r.6 [amicable]

```
import run_tests  # noqa
def sum_divisors(number):
    result = 0
    for i in range(1, number // 2 + 1):
        if number % i == 0:
            result += i
    return result

def amicable(a, b):
    return sum_divisors(a) == b and sum_divisors(b) == a
```

K.2: Týden 2

K.2.e.1 [palindrome]

```
def reverse(number):
```

```
    result = 0
    while number > 0:
        digit = number % 10
        result = result * 10 + digit
        number = number // 10
    return result

def is_palindrome(number):
    return number == reverse(number)

def verity(student):
    from quick import assert_eq
    for num in range(10000):
        assert_eq(is_palindrome, student, num)

import run_tests  # noqa
```

K.2.e.2 [gcd]

```
def gcd(x1, x2):
    if x1 == 0 or x2 == 0:
        return max(abs(x1), abs(x2))

    curr_divisor = min(abs(x1), abs(x2))
    while curr_divisor > 0:
        if x1 % curr_divisor == 0 and x2 % curr_divisor == 0:
            return curr_divisor
        curr_divisor -= 1

def verity(student):
    from quick import assert_eq, nat1, anno_enum
    for i in range(3000):
        x, y = anno_enum(tuple[nat1, nat1], i)
        assert_eq(gcd, student, x, y)

import run_tests  # noqa
```

K.2.e.3 [digits]

```
def count_digit_in_sequence(digit, low, high):
    count = 0
    if low == 0 and digit == 0:
        count += 1
    for number in range(low, high + 1):
        while number > 0:
            if digit == number % 10:
                count += 1
            number = number // 10
    return count

def verity(student):
    from quick import assert_eq
    for digit in range(10):
        for low in range(100):
            for count in range(20):
                assert_eq(count_digit_in_sequence, student,
                          digit, low, low + count)
        assert_eq(count_digit_in_sequence, student,
                  digit, 10**20, 10**20 + digit)

import run_tests  # noqa
```

K.2.r.1 [savings]

```
def apply_interest(amount, rate):
    return floor(amount + amount * rate / 100.0)

def savings_years(savings, interest_rate, inflation, withdraw):
    years = 0
    while savings >= withdraw:
        savings -= withdraw
        savings = apply_interest(savings, interest_rate)
        withdraw = apply_interest(withdraw, inflation)
        years += 1
    return years

def apply_interest_alt(amount, rate):
    return amount * (1000 + rate) // 1000

import run_tests  # noqa
```

K.2.r.2 [fridays]

```
def is_leap(year):
    if year % 400 == 0:
        return True
    if year % 4 == 0 and year % 100 != 0:
        return True
    return False

def days_per_month(year, month):
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week):
    return day_of_week == 4

def fridays(year, day_of_week):
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
            if is_friday(day_of_week) and day == 13:
                count += 1
            day_of_week = (day_of_week + 1) % 7
    return count

import run_tests  # noqa
```

K.2.r.3 [delete]

```
def delete_to_maximal(number):
    result = 0
    power = 1
    while number // power > 0:
        candidate = number // (power * 10) * power + number % power
        power *= 10
        if result < candidate:
            result = candidate
    return result

def delete_k_to_maximal(number, k):
    for i in range(k):
        number = delete_to_maximal(number)
    return number

import run_tests  # noqa
```

K.2.r.4 [cards]

```
def digits(number):
    counter = 0
    while number >= 10 ** counter:
        counter += 1
    return counter

def first_n_digits(number, n):
    return number // (10 ** (digits(number) - n))

def is_visa(number):
    if not is_valid_card(number):
        return False
    digs = digits(number)
    if digs == 13 or digs == 16 or digs == 19:
        return first_n_digits(number, 1) == 4
    return False

def is_mastercard(number):
    if not is_valid_card(number) or digits(number) != 16:
        return False

    if 50 <= first_n_digits(number, 2) <= 55:
        return True
```

```
    if 22100 <= first_n_digits(number, 5) <= 27209:
        return True

    return False
```

K.2.r.5 [bisect]

```
def bisect(fun, low, high, eps):
    while True:
        mid = (low + high) / 2
        x = fun(mid)
        if abs(x) < eps:
            return mid

        if fun(low) * x < 0:
            high = mid
        else:
            low = mid

import run_tests  # noqa
```

K.2.r.6 [parasitic]

```
def is_parasitic(num: nat1, base: int) -> 'int | None':
    orig = num
    last = num % base
    power = 1
    while num >= base:
        power *= base
        num //= base
    new = orig // base + last * power
    return new // orig if new % orig == 0 else None

def verity(student: to_test) -> None:
    Base = Annotated[int, oneof(2, 5, 9, 10, 11, 101)]
    is_parasitic.__annotations__['base'] = Base
    assert_eq(is_parasitic, student)

    inputs = {}
    inputs[10] = 102564, 128205, 142857, 153846, 179487, 205128, 230769
    inputs[11] = 15, 20, 30, 40, 140, 175, 190, 380, 570, 1830, \
        2440, 2745, 3660, 4575, 4880, 163170, 168720, 186480, \
        196840, 209790, 221445, 233100, 253080, 295260, 337440, \
        421800, 442890, 506160, 590520, 674880, 759240, 843600, \
        1812760, 2265950, 2719140, 3172330, 3625520, 4078710, \
        4531900, 20096145, 26794860, 33493575, 35726480, 40192290, \
        46891005, 53589720, 60288435, 66987150, 71452960, 248205020, \
        310256275, 336849670, 673699340
    inputs[13] = 21, 24, 244, 2240, 2800, 2856, 3360, 3570, 3808, \
        3920, 4080, 4480, 5040, 5355, 5600, 5712, 6160, 6720, 11424, \
        536312, 603351, 689544, 63978880, 76474755, 79973600, \
        81573072, 95968320, 101966340, 108764096, 111963040, \
        116532960, 127457925, 127957760, 143952480, 152949510, \
        159947200, 163146144, 175941920, 191936640, 326292288, \
        1178277708
    inputs[14] = 39, 65, 7683, 12805, 48893, 97786, 836615, 1505907, \
        1673230, 2509845, 3346460, 107984565, 143979420, 179974275, \
        215969130, 251963985, 287958840, 295157811, 323953695, \
        359948550, 395943405, 431938260, 467933115, 491929685
    inputs[15] = 28, 56, 6328, 12656, 69034, 103551, 138068, 172585, \
        207102, 241619, 1423828, 2847656, 172295168, 193832064, \
        215368960, 236905856, 258442752, 279979648, 301516544, \
        320361328, 640722656
    inputs[16] = 260, 315, 325, 351, 390, 455, 520, 585, 650, 715, \
        780, 845, 910, 975, 67650, 83886, 95325, 101475, 135300, \
        169125, 202950, 236775, 270600, 304425, 338250, 372075, \
        405900, 439725, 473550, 507375, 1065220, 1290555, 1331525, \
        1438047, 1597830, 1864135, 2130440, 2396745, 2663050, \
        2929355, 3195660, 3461965, 3728270, 3994575, 16909320, \
        19022985, 21136650, 23250315, 25363980, 27477645, 29591310, \
        31704975
    inputs[17] = 24, 32, 48, 96, 5568, 5760, 6960, 7424, 8352, 9280, \
        13920, 16704, 25056, 27840, 1436760, 1724112, 1787968, \
        1856736, 2011464, 2298816, 2586168, 2681952, 2873520, \
        3160872, 3448224, 3735576, 4022928, 4310280, 4597632, \
        8045856, 465050496, 481086720, 581313120, 620067328, \
        697575744, 775084160, 1162626240, 1395151488
    inputs[18] = 833, 1666, 2499, 8075, 8398, 12597, 20995, 41990, \
        4858889, 9717778, 14576667, 847689275, 881596846, 1322395269
    inputs[19] = 24, 30, 36, 45, 48, 72, 90, 508, 8145, 8688, 10860, \
        13032, 16290, 17376, 26064, 32580, 131184, 147582, 163980, \
        180378, 196776, 213174, 229572, 245970, 262368, 278766, \
        295164, 2489200, 2520315, 2738120, 2987040, 3136392, \
        3235960, 3360420, 3484880, 3733800, 3920490, 3982720, \
        4200525, 4231640, 4480560, 4704588, 5040630, 5880735, \
        6272784, 6720840, 7560945, 8401050, 9241155, 9409176, \
        10081260, 10921365, 11761470, 12601575, 13441680, 14281785, \
        15121890, 899129808, 999033120, 1061472690, 1098936432, \
        1132237536, 1198839744, 1298743056, 1398646368, 1415296920, \
        1498549680, 1598452992, 1698356304, 1798259616, 2122945380

    for base, numbers in inputs.items():
        for n in numbers:
            assert_eq(is_parasitic, student, n, base)

    for n in list(range(1, 100, 10)) + list(range(100, 1001, 100)):
        for base in list(range(2, 11)) + [13, 20, 21, 23, 101]:
            for d in range(1, base):
                repeat = sum(d * base ** k for k in range(n))
                assert_eq(is_parasitic, student, repeat, base)

import run_tests  # noqa
```

K.3: Týden 3

K.3.e.1 [predicates]

```
def all_greater_than(sequence, n):
    for x in sequence:
        if x <= n:
            return False
    return True

def any_even(sequence):
    for x in sequence:
        if x % 2 == 0:
            return True
    return False

import run_tests  # noqa
```

K.3.e.2 [explosion]

```
def distance(a, b):
    x1, y1, z1 = a
    x2, y2, z2 = b
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2 + (z1 - z2) ** 2)

def survivors(objects, center, radius):
    out = []
    for obj in objects:
        if distance(obj, center) > radius:
            out.append(obj)
    return out

import run_tests  # noqa
```

K.3.e.3 [cartesian]

```
def cartesian(a, b):
    out = []
    for x in a:
        for y in b:
            out.append((x, y))
    return out

import run_tests  # noqa
```

K.3.r.1 [quiz]

```
def mark_points(answers, solution):
```

```
    scored_points = 0
    for i in range(len(solution)):
        correct_answer, points = answers[i]
        if solution[i] == correct_answer:
            scored_points += points
    return scored_points

import run_tests  # noqa
```

K.3.r.2 [rectangles]

```
def has_overlap(a, b):
    (ax1, ay1), (ax2, ay2) = a
    (bx1, by1), (bx2, by2) = b
    return ax1 <= bx2 and ax2 >= bx1 and ay1 <= by2 and ay2 >= by1

def filter_overlapping(rectangles):
    out = []
    count = len(rectangles)
    for i in range(count):
        for j in range(count):
            if i != j and has_overlap(rectangles[i], rectangles[j]):
                out.append(rectangles[i])
                break
    return out

import run_tests  # noqa
```

K.3.r.3 [concat]

```
def concat(lists):
    out = []
    for entry in lists:
        for x in entry:
            out.append(x)
    return out

import run_tests  # noqa
```

K.3.r.4 [rcellular]

```
def right(state, idx, i):
    return 0 if idx + i >= len(state) else state[idx + i]

def local(state, idx):
    return (state[idx], right(state, idx, 1), right(state, idx, 2))

def cellular_in_situ(state: list[int]):
    for i in range(len(state)):
        config = local(state, i)
        if config == (1, 0, 0):
            state[i] = 0
        elif config == (0, 1, 0):
            state[i] = 1
        elif config == (0, 1, 1):
            state[i] = 1
        elif config == (1, 0, 1):
            state[i] = 0
        elif config == (1, 1, 1):
            state[i] = 0

def verity(student):
    from quick import assert_eq
    for n in range(40000):
        state = []
        while n > 0:
            state.append(n % 2)
            n //= 2
        assert_eq(cellular_in_situ, student, state)

import run_tests  # noqa
```

K.3.r.5 [squares]

```
def slope(x, y, average_x, average_y):
    dividend = 0
    divisor = 0

    for i in range(len(x)):
        dividend += ((x[i] - average_x) * (y[i] - average_y))
        divisor += (x[i] - average_x) ** 2

    if divisor == 0:
        return None

    return dividend / divisor

def deviations(x, y, alpha, beta):
    res = []
    for i in range(len(x)):
        res.append(abs(y[i] - beta * x[i] - alpha))
    return res

def least_squares(x, y):
    average_x = float(sum(x)) / len(x)
    average_y = float(sum(y)) / len(y)

    beta = slope(x, y, average_x, average_y)
    if beta is None:
        return None

    alpha = average_y - beta * average_x

    return (alpha, beta, deviations(x, y, alpha, beta))

import run_tests  # noqa
```

K.3.r.6 [partition]

```
def partition(data, idx):
    pivot = data[idx]
    low, high = 0, len(data) - 1
    while True:
        while data[low] < pivot:
            low += 1

        while data[high] > pivot:
            high -= 1

        if low >= high:
            return
        data[low], data[high] = data[high], data[low]

import run_tests  # noqa
```

K.4: Týden 4

K.4.e.1 [typefun]

```
def degrees(radians: float) -> float:
    return (radians * 180) / pi

def diagonal(lst: list[list[int]]) -> list[int]:
    diag = []
    for i in range(len(lst)):
        diag.append(lst[i][i])
    return diag

def to_list(num: int, base: int) -> list[int]:
    digits = []
    result = []

    while num > 0:
        digits.append(num % base)
        num //= base

    for i in range(len(digits)):
        result.append(digits[-i - 1])

    return result

Element = tuple[int, str]

def with_id(elements: list[Element], id_: int) -> str | None:
    for element_id, val in elements:
        if id_ == element_id:
            return val
```

```
    return None

Student = tuple[int, str, int | None]

def update_students(students: list[Student],
                    end: int) -> list[Student]:

    result: list[Student] = []

    for uco, name, graduated in students:
        if graduated is None:
            graduated = end
        result.append((uco, name, graduated))

    return result

def is_increasing(seq: list[int]) -> bool:
    for i in range(1, len(seq)):
        if seq[i - 1] >= seq[i]:
            return False
    return True

import run_tests  # noqa
```

K.4.e.2 [squares]

```
def slope(x: list[float], y: list[float], average_x: float, average_y:
float) \

        -> float | None:
    dividend: float = 0
    divisor: float = 0

    for i in range(len(x)):
        dividend += ((x[i] - average_x) * (y[i] - average_y))
        divisor += (x[i] - average_x) ** 2

    if divisor == 0:
        return None

    return dividend / divisor

def deviations(x: list[float], y: list[float], alpha: float, beta: float)
\
        -> list[float]:
    res: list[float] = []
    for i in range(len(x)):
        res.append(abs(y[i] - beta * x[i] - alpha))
    return res

def least_squares(x: list[float], y: list[float]) \
        -> tuple[float, float, list[float]] | None:
    average_x: float = float(sum(x)) / len(x)
    average_y: float = float(sum(y)) / len(y)

    beta: float | None = slope(x, y, average_x, average_y)
    if beta is None:
        return None

    alpha: float = average_y - beta * average_x

    return (alpha, beta, deviations(x, y, alpha, beta))

import run_tests  # noqa
```

K.4.e.3 [fridays]

```
Day = int
Year = int
Month = int

def is_leap(year: Year) -> bool:
    if year % 400 == 0:
        return True
    if year % 4 == 0 and year % 100 != 0:
        return True
    return False

def days_per_month(year: Year, month: Month) -> int:
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week: Day) -> bool:
    return day_of_week == 4

def fridays(year: Year, day_of_week: Day) -> int:
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
            if is_friday(day_of_week) and day == 13:
                count += 1
            day_of_week = (day_of_week + 1) % 7
    return count

import run_tests  # noqa
```

K.4.r.1 [squares]

```
def find_slope(points: list[tuple[float, float]],
               avg_x: float, avg_y: float) -> float | None:
    dividend: float = 0
    divisor: float = 0

    for i, (x, y) in enumerate(points):
        dividend += ((x - avg_x) * (y - avg_y))
        divisor += (x - avg_x) ** 2

    if divisor == 0:
        return None

    return dividend / divisor

def find_intercept(avg_x: float, avg_y: float, beta: float) -> float:
    return avg_y - beta * avg_x

def regress_vectors(x: list[float], y: list[float]) \
        -> tuple[float, float] | None:
    return regress_points([(x[i], y[i]) for i in range(len(x))])

def regress_points(points: list[tuple[float, float]]) \
        -> tuple[float, float] | None:
    avg_x = sum([x for x, _ in points]) / len(points)
    avg_y = sum([y for _, y in points]) / len(points)

    slope = find_slope(points, avg_x, avg_y)

    if slope is None:
        return None

    intercept = find_intercept(avg_x, avg_y, slope)
    return (intercept, slope)

def residuals_vectors(x: list[float], y: list[float],
                      alpha: float, beta: float) -> list[float]:
    points = [(x[i], y[i]) for i in range(len(x))]
    return residuals_points(points, alpha, beta)

def residuals_points(points: list[tuple[float, float]],
                     alpha: float, beta: float) -> list[float]:
    res = []
    for i, (x, y) in enumerate(points):
        res.append(abs(y - beta * x - alpha))
    return res

import run_tests  # noqa
```

K.4.r.2 [life]

```
Grid = list[list[int]]

def cell_value(grid: Grid, x: int, y: int) -> int:
    if 0 <= x < len(grid) and 0 <= y < len(grid):
        return grid[x][y]
    return 0

def live_neighbour_count(grid: Grid, x: int, y: int) -> int:
    assert x < len(grid) and y < len(grid)

    res = 0
    for row in range(x - 1, x + 2):
        for col in range(y - 1, y + 2):
            res += cell_value(grid, row, col)
    return res - grid[x][y]

def next_value(grid: Grid, x: int, y: int) -> int:
    assert x < len(grid) and y < len(grid)

    live_neighbours = live_neighbour_count(grid, x, y)

    if grid[x][y] == 0:
        return 1 if live_neighbours == 3 else 0

    if live_neighbours == 2 or live_neighbours == 3:
        return 1
    return 0

def step(grid: Grid) -> Grid:
    assert len(grid) > 0

    res: Grid = []
    for i in range(len(grid)):
        res.append([])
        for j in range(len(grid[0])):
            res[i].append(next_value(grid, i, j))
    return res

def life(grid: Grid, count: int) -> Grid:
    assert len(grid) > 0
    assert count >= 0

    world = [curr[:] for curr in grid]

    for _ in range(count):
        next_step = step(world)
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                world[i][j] = next_step[i][j]

    return world
```

K.4.r.3 [predicates]

```
def test_f_1() -> None:
```

f_1(x, y) právě když fib(x) == y

```
    assert f_1(1, 1)
    assert f_1(2, 1)
    assert f_1(3, 2)
    assert f_1(4, 3)
    assert f_1(5, 5)
    assert f_1(6, 8)
    assert not f_1(4, 2)

def test_f_2() -> None:
```

f_2(x, y) právě když divisors(x) ≥ y

```
    assert f_2(1, 1)
    assert f_2(2, 2)
    assert f_2(3, 2)
    assert f_2(12, 6)
    assert not f_2(12, 7)
    assert f_2(12, 5)

def test_f_3() -> None:
```

f_3(x, y) právě když divisors(x) > divisors(y)

```
    assert not f_3(1, 1)
    assert f_3(12, 13)
    assert not f_3(3, 2)
    assert f_3(12, 6)

def test_f_4() -> None:
```

f_4(x, y) právě když y - 1 je počet prvočísel < x

```
    assert f_4(3, 2)
    assert f_4(5, 3)
    assert f_4(6, 4)
    assert f_4(7, 4)
    assert f_4(11, 5)
    assert f_4(12, 6)
    assert f_4(17, 7)

def test_f_5() -> None:
```

f_5(x) právě když je x base-7 palindrom

```
    assert f_5(6)
    assert f_5(1 * 7**3 + 2 * 7**2 + 2 * 7 + 1)
    assert f_5(1 * 7**4 + 2 * 7**3 + 7 ** 2 + 2 * 7 + 1)
    assert not f_5(1 * 7**4 + 3 * 7**3 + 7 ** 2 + 2 * 7 + 1)

def test_f_6() -> None:
```

f_6(x, y) právě když jsou x, y v binárním zápisu zrcadlové obrazy

```
    assert f_6(0b1101, 0b1011)
    assert f_6(0b110001, 0b100011)
    assert f_6(0b1101001, 0b1001011)
    assert not f_6(0b101001, 0b1001011)
    assert not f_6(0b101, 0b111)

def test_f_7() -> None:
```

f_7(x, y) právě když je y počet různých prvočísel v rozkladu x

```
    assert f_7(7, 1)
    assert f_7(14, 2)
    assert f_7(15, 2)
    assert f_7(30, 3)

def test_f_8() -> None:
```

f_8(x, y, z) právě když je z počet různých prvočísel dělících x a zároveň y

```
    assert f_8(1, 1, 0)
    assert f_8(2, 4, 1)
    assert f_8(4, 2, 1)
    assert f_8(21, 14, 1)
    assert f_8(14, 28, 2)
    assert f_8(28, 28, 2)
    assert f_8(9, 12, 1)
    assert f_8(16, 12, 1)
    assert f_8(24, 12, 2)
    assert f_8(120, 60, 3)
    assert f_8(180, 60, 3)
    assert f_8(180, 120, 3)
```

K.4.r.4 [poly]

```
def differentiate(poly: Polynomial) -> Polynomial:
    res = poly.copy()
    res.pop()

    if res == []:
        return [Fraction(0)]

    for i in range(len(res)):
        res[i] *= len(res) - i

    return res

def integrate(poly: Polynomial) -> Polynomial:
    res = poly.copy()

    if res == [Fraction(0)]:
        return res
```

```
    for i in range(len(res)):
        res[i] = Fraction(res[i], len(res) - i)

    res.append(Fraction(0))

    return res

def check_inverse(poly: Polynomial) -> bool:
    dif_int = differentiate(integrate(poly))
    int_dif = integrate(differentiate(poly))

    if len(dif_int) != len(int_dif) != len(poly):
        return False

    for i in range(0, len(poly) - 1):
        if dif_int[i] != poly[i] or int_dif[i] != poly[i]:
            return False

    if dif_int[-1] != poly[-1]:
        return False

    return True
```

K.4.r.5 [**mystery**]

```
def mystery_function(nums: list[int]) -> list[int]:
```

Přeskládá a přepočítá prvky pole tak, že nejprve budou poloviny sudých prvků a poté dvojnásobky lichých prvků.

```
    result = [0] * len(nums)

    i = 0
    for num in nums:
        if num % 2 == 0:
            result[i] = num // 2
            i += 1
    for num in nums:
        if num % 2 != 0:
            result[i] = num * 2
            i += 1
    return result


def mysterious_shift(arr: list[float]) -> list[float]:
```

Funkce ke každému prvku pole přičte jeho index.

```
    result: list[float] = []

    secret_code = 123456
    cipher_key = 654321

    for essential_index in range(len(arr)):
        data_point = arr[essential_index] + essential_index
        code_combination = data_point + secret_code
        decoded_element = code_combination - secret_code
        key_interaction = decoded_element * cipher_key
        final_element = key_interaction / cipher_key

        distraction_1 = secret_code * cipher_key
        distraction_2 = distraction_1 / cipher_key
        distraction_3 = distraction_2 - secret_code

        final_element += distraction_3 - distraction_3

        for _ in result:
            final_element = final_element * 1

        result.append(final_element)

    return result

import run_tests  # noqa
```

K.4.r.6 [**precondition**]

```
def precondition_1(x_0: int, y: int) -> bool:
    return y != 0 and x_0 % y == 0

def precondition_2(x_0: int, y_0: int) -> bool:
    return x_0 <= y_0 and (x_0 - y_0) % 2 == 0

def precondition_3(x: int, y: int) -> bool:
    return x > 0 and y < 0

def precondition_4(x_0: int, y: int) -> bool:
    return x_0 >= 0 and y > 0

import run_tests  # noqa
```

K.5: Týden 5

K.5.r.1 [**transitive**]

```
def is_transitive(relation: set[tuple[int, int]]) -> bool:
    for a, b in relation:
        for b_prime, c in relation:
            if b == b_prime and (a, c) not in relation:
                return False
    return True

import run_tests  # noqa
```

K.5.r.2 [**setops**]

```
def set_union(a: set[int], b: set[int]) -> set[int]:
    result = set()

    for x in a:
        result.add(x)
    for x in b:
        result.add(x)

    return result

def set_update(to_extend: set[int], other: set[int]) -> None:
    for x in other:
        to_extend.add(x)

def set_intersect(a: set[int], b: set[int]) -> set[int]:
    if len(b) < len(a):
        a, b = b, a

    result = set()

    for x in a:
        if x in b:
            result.add(x)

    return result

def set_keep(to_reduce: set[int], other: set[int]) -> None:
    for x in to_reduce.copy():
        if x not in other:
            to_reduce.remove(x)

import run_tests  # noqa
```

K.5.r.3 [**setdiff**]

```
def set_difference(a: set[int], b: set[int]) -> set[int]:
    result = set()
    for x in a:
        if x not in b:
            result.add(x)
    return result

def set_remove(to_reduce: set[int], other: set[int]) -> None:
    for x in other:
        if x in to_reduce:
            to_reduce.remove(x)

def set_symmetric_diff(a: set[int], b: set[int]) -> set[int]:
    result = set()
    for x in a:
```

```
        if x not in b:
            result.add(x)
    for x in b:
        if x not in a:
            result.add(x)
    return result

def set_symmetric_inplace(to_change: set[int],
                          other: set[int]) -> None:
    for x in other:
        if x in to_change:
            to_change.remove(x)
        else:
            to_change.add(x)

import run_tests  # noqa
```

K.5.r.4 [maps]

```
def image(f: dict[int, int], values: set[int]) -> set[int]:
    result = set()
    for x in values:
        if x in f:
            result.add(f[x])
    return result

def preimage(f: dict[int, int], values: set[int]) -> set[int]:
    result = set()
    for x in f.keys():
        if f[x] in values:
            result.add(x)
    return result

def compose(f: dict[int, int], g: dict[int, int]) -> dict[int, int]:
    result = {}
    for x in g.keys():
        result[x] = f[g[x]]
    return result

def kernel(f: dict[int, int]) -> set[tuple[int, int]]:
    result = set()
    for x in f.keys():
        for y in f.keys():
            if f[x] == f[y]:
                result.add((x, y))
    return result

import run_tests  # noqa
```

K.5.r.5 [life]

```
def updated(x: int, y: int, cells: set[tuple[int, int]]) -> bool:
    count = 0
    alive = (x, y) in cells

    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx or dy:
                count += (x + dx, y + dy) in cells

    return count in {2, 3} if alive else count == 3

def life(cells: set[tuple[int, int]], n: int) \
        -> set[tuple[int, int]]:
    if n == 0:
        return cells

    todo = set()
    ngen = set()

    for x, y in cells:
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                todo.add((x + dx, y + dy))

    for x, y in todo:
        if updated(x, y, cells):
            ngen.add((x, y))

    return life(ngen, n - 1)

import run_tests  # noqa
```

K.6: Týden 6

K.6.e.1 [symmetric]

```
def is_symmetric(relation: set[tuple[int, int]]) -> bool:
    for a, b in relation:
        if (b, a) not in relation:
            return False
    return True

import run_tests  # noqa
```

K.6.r.2 [fixpoint]

```
def apply_f_on_num(num: int) -> set[int]:
    return {num, num // 2, num // 7}

def fixpoint(starting_set: set[int]) -> int:
    next_set: set[int] = set()
    prev_set = starting_set.copy()
    result = 0
    while True:
        for num in prev_set:
            next_set.update(apply_f_on_num(num))
        if len(prev_set) == len(next_set):
            return result
        result += 1
        prev_set.update(next_set)

import run_tests  # noqa
```

K.6.r.3 [breadth]

```
def breadth(tree: Tree) -> int:
    maximal = 1
    row = [1]

    while row:
        next_row = []
        for node in row:
            for succ in tree[node]:
                next_row.append(succ)
        if len(next_row) > maximal:
            maximal = len(next_row)
        row = next_row

    return maximal
```

K.6.r.4 [variables]

```
def operation(operator: str, left: int, right: int) -> int:
    if operator == "+":
        return left + right
    return left * right

def evaluate(expr: dict[str, tuple[str, str, str]],
             const: dict[str, int], var: str) -> int:
    results = {}
    stack = [var]

    while stack:
        top = stack[-1]
        if top in const:
            results[top] = const[top]
        elif top in expr:
            op, left, right = expr[top]
            if left in results and right in results:
                results[top] = operation(op, results[left],
                                         results[right])
            else:
                stack.append(left)
                stack.append(right)
```

```
                continue  # do not pop
        else:
            results[top] = 0
        stack.pop()

    return results[var]

import run_tests  # noqa
```

K.6.r.5 [connected]

```
def all_connected(stops: dict[str, list[str]]) -> bool:
    for stop in stops.keys():
        stack = [stop]
        reachable = {stop}
        while stack:
            for current in stops[stack.pop()]:
                if current not in reachable and current != stop:
                    reachable.add(current)
                    stack.append(current)
        if stops.keys() != reachable:
            return False
    return True

import run_tests  # noqa
```

K.6.r.6 [lakes]

```
def lakes(land: list[nat]) -> int:
    water = 0
the stack holds the left edges of currently "open" basins
    stack: list[int] = []

    for i in range(len(land)):
        height = land[i]

        bottom = 0

closing the basins

        while stack and height >= land[stack[-1]]:
            water += (i - stack[-1] - 1) * (land[stack[-1]] - bottom)
            bottom = land[stack[-1]]
            stack.pop()

        if stack:
            water += (i - stack[-1] - 1) * (height - bottom)

        stack.append(i)

    return water

def verity(student: to_test) -> None:
    assert_eq(lakes, student)
    assert_eq(lakes, student, strategy=s_exp, count=2000)

    for n in range(500, 10000, 500):
        seq = list(range(n))
        basin = [abs(h) for h in range(-n, n)]
        assert_eq(lakes, student, seq)
        assert_eq(lakes, student, basin)
        assert_eq(lakes, student, basin + basin)

import run_tests  # noqa
```

K.7: Týden 7

K.7.e.1 [warriors]

```
class Warrior:
    def __init__(self, name: str, strength: int) -> None:
        self.name = name
        self.strength = strength

class Horde:
    def __init__(self, clans: dict[str, list[Warrior]]) -> None:
        self._clans = clans

    def clans(self) -> dict[str, list[Warrior]]:
        return self._clans

    def add_warrior(self, clan: str, warrior: Warrior) -> None:
        if clan not in self._clans:
            self._clans[clan] = [warrior]
        else:
            self._clans[clan].append(warrior)

    def validate_clan_strength(self, required: int) -> bool:
        for d, ws in self._clans.items():
            total = 0
            for w in ws:
                total += w.strength
            if total <= required:
                return False
        return True

import run_tests  # noqa
```

K.7.e.2 [sorted]

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None
```

XXX the visited set is 'any' (masqueraded as to_test) because for reasons unknown, python explodes on set[int] here (though it works elsewhere)

```
    def to_str(self, visited: to_test) -> str:
        out = str(self.value)
        if id(self) in visited:
            out += " (loop)"
        elif self.next is not None:
            out += ' → ' + self.next.to_str(visited | {id(self)})
        return out

    def __repr__(self) -> str:
        return self.to_str(set())

    def __eq__(self, other: object) -> bool:
        if isinstance(other, Node):
            return self.value == other.value and self.next == other.next
        else:
            return NotImplemented

class SortedList:
    def __init__(self) -> None:
        self.head: Node | None = None

    def insert(self, value: int) -> None:
        node = Node(value)

        it: Node | None = self.head
        prev = None
        while it is not None and it.value < value:
            prev = it
            it = it.next

        node.next = it
        if prev is not None:
            prev.next = node
        else:
            self.head = node

    def get_greatest_in(self, value: int, dist: int) -> int | None:
        out = None
        it = self.head
        while it is not None and it.value < value:
            it = it.next
        while it is not None and it.value <= value + dist:
            out = it.value
            it = it.next
        return out
```

```
    def __eq__(self, other: object) -> bool:
        if hasattr(other, 'head'):
            return self.head == getattr(other, 'head')
        else:
            return NotImplemented

    def __repr__(self) -> str:
        return '(head) → ' + repr(self.head)

def make_tests(list_type: to_test) -> to_test:
    def construct_linked(values: list[int]) -> to_test:
        result = list_type()
        for v in values:
            result.insert(v)
        return result

    def construct_and_get(values: list[int], value: int,
                          dist: nat) -> int | None:
        lst = construct_linked(values)
        return lst.get_greatest_in(value, dist)

    return construct_linked, construct_and_get

def verity(student: to_test) -> None:
    student.Node = Node
    setattr(student.SortedList, '__eq__', SortedList.__eq__)
    setattr(student.SortedList, '__repr__', SortedList.__repr__)

    ref_c, ref_g = make_tests(SortedList)
    stu_c, stu_g = make_tests(student.SortedList)

    assert_eq(ref_c, stu_c)
    assert_eq(ref_g, stu_g)

    for n in range(10, 1000, 10):
        assert_eq(ref_g, stu_g, list(range(n)), 0, 3)

def sanity(student: to_test, replace: to_test) -> None:
    replace(Node)
    student.main()

import run_tests  # noqa
```

K.7.e.3 [maximum]

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next: Node | None = None

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None

def maximum(num_list: LinkedList) -> int | None:
    node = num_list.head

    if node is None:
        return None

    max_val = node.value

    while node is not None:
        if node.value > max_val:
            max_val = node.value
        node = node.next
    return max_val

import run_tests  # noqa
```

K.7.r.1 [circular]

```
class Node:
    def __init__(self, value: int) -> None:
        self.value = value
        self.next = self

class CircularList:
    def __init__(self) -> None:
        self.head: Node | None = None
        self.end: Node | None = None

    def insert(self, value: int) -> None:
        new_head = Node(value)
        if self.head is None:
            self.end = new_head
        else:
            assert self.end is not None
            new_head.next = self.head
            self.end.next = new_head
        self.head = new_head

    def last(self) -> Node | None:
        return self.end

    def split_by_value(self, value: int) -> 'CircularList':
        assert self.head is not None
        it = self.head
        while it.value != value:
            it = it.next
        return self.split_by_node(it)

    def split_by_node(self, node: Node) -> 'CircularList':
        assert self.head is not None
        assert self.end is not None

        if node == self.end:
            return CircularList()

        new_list = CircularList()
        new_list.head = node.next
        new_list.end = self.end
        new_list.end.next = new_list.head

        node.next = self.head
        self.end = node

        return new_list

import run_tests  # noqa
```

K.7.r.2 [shuffle]

```
class Node:
    def __init__(self, value: int):
        self.value = value
        self.next: Node | None = None
        self.id = 0

    def idstr(self) -> str:
        return ('0123456789'[self.id // 10] +
                '0123456789'[self.id % 10])

class LinkedList:
    def __init__(self) -> None:
        self.head: Node | None = None

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, LinkedList):
            return NotImplemented
        a = self.head
        b = other.head
        while a or b:
            if a is None or b is None:
                return False
            if a.id != b.id:
                return False
            if a.value != b.value:
                return False
            a = a.next
            b = b.next
        return True

    def __repr__(self) -> str:
        out = '(head)'
```

```
        ptr = self.head

        while ptr is not None:
            out += ' → ' + str(ptr.value) + ptr.idstr()
            ptr = ptr.next
        return out

def build_linked(nums: list[int]) -> LinkedList:
    head = Node(0)
    tail = head

    for i, v in enumerate(nums):
        tail.next = Node(v)
        tail = tail.next
        tail.id = i % 100

    result = LinkedList()
    result.head = head.next
    return result

def shuffle(permutation: list[int], linked: LinkedList) -> None:
    if permutation == []:
        return

    nodes_in_order: list[Node | None] = \
        [None for _ in permutation]

    curr_idx = 0
    curr_node = linked.head

    while curr_node is not None:
        nodes_in_order[permutation[curr_idx]] = curr_node
        curr_node = curr_node.next
        curr_idx += 1

    linked.head = nodes_in_order[0]
    last_added = linked.head

    for i in range(1, len(nodes_in_order)):
        if last_added is None:
            break

        last_added.next = nodes_in_order[i]
        last_added = last_added.next

    if last_added is not None:
        last_added.next = None

def verity(student: to_test) -> None:
    for index in list(s_gamma(10000)) + list(s_exp(5000)):
        p = list(permute_enum(index))
        for v in list(range(len(p))), p:
            assert_eq(shuffle, student,
                      get_arg=lambda: (p.copy(), build_linked(v)))

    for n in range(100, 3000, 100):
        p = list(reversed(range(n)))
        assert_eq(shuffle, student,
                  get_arg=lambda: (p.copy(), build_linked(p)))

def sanity(student: to_test, replace: to_test) -> None:
    replace(Node)
    replace(LinkedList)
    student.main()
```

K.7.r.3 [books]

```
class Book:
    def __init__(self, name: str, author: str) -> None:
        self.name = name
        self.author = author

class Bookshelf:
    def __init__(self, books: list[Book]) -> None:
        self._books = books

    def add_book(self, book: Book) -> None:
        self._books.append(book)

    def group_by_author(self) -> dict[str, list[Book]]:
        result: dict[str, list[Book]] = {}
        for book in self._books:
            if book.author not in result:
                result[book.author] = []
            result[book.author].append(book)
        return result

    def books(self) -> list[Book]:
        return self._books

import run_tests  # noqa
```

K.7.r.4 [select]

```
def select(indices: list[int], linked: LinkedList) -> LinkedList:
    current = linked.head
    index = 0
    result = LinkedList()
    last = None

    for pick in indices:
        while index < pick:
            index += 1
            assert current is not None
            current = current.next
        assert current is not None

        node = Node(current.value)

        if last is None:
            result.head = node
        else:
            last.next = node

        last = node

    return result
```

K.7.r.5 [zipper]

```
class Node:
    def __init__(self, value: int) -> None:
        self.next: Node | None = None
        self.value = value

class Zipper:
    def __init__(self, value: int) -> None:
        self.left: Node | None = None
        self.right: Node | None = None
        self._cursor = value

    def cursor(self) -> int:
        return self._cursor

    def insert_left(self, num: int) -> None:
        node = Node(num)
        node.next = self.left
        self.left = node

    def delete_left(self) -> int | None:
        value: int | None = None
        if self.left:
            value = self.left.value
            self.left = self.left.next
        return value

    def shift_left(self) -> None:
        if self.left is None:
            return

        node = Node(self._cursor)
        node.next = self.right
        self.right = node
        self._cursor = self.left.value
        self.left = self.left.next

    def shift_right(self) -> None:
```

```
        if self.right is None:
            return

        node = Node(self._cursor)
        node.next = self.left
        self.left = node
        self._cursor = self.right.value
        self.right = self.right.next

def ops_enum(index: int) -> list[str]:
    ops: list[str] = []
    while index:
        index, kind = divmod(index, 4)
        if kind == 0:
            ops.append('shift_left')
        elif kind == 1:
            ops.append('shift_right')
        elif kind == 2:
            ops.append('delete_left')
        elif kind == 3:
            index, value = divmod(index, 7)
            ops.append('insert_left ' + str(value + 1))
    ops.extend(['shift_left' for _ in range(5)])
    ops.extend(['shift_right' for _ in range(5)])
    return ops

Ops = Annotated[list[str], ops_enum]

def run(Z: to_test) -> to_test:
    def cursor_after_each_op(ops: Ops) -> list[int]:
        zipper = Z(0)
        out: list[int] = []
        for op in ops:
            parts = op.split(' ')
            if parts == ['shift_left']:
                zipper.shift_left()
            elif parts == ['shift_right']:
                zipper.shift_right()
            elif parts == ['delete_left']:
                zipper.delete_left()
            else:
                cmd, value = parts
                assert cmd == 'insert_left'
                zipper.insert_left(int(value))
            out.append(zipper.cursor())
        return out

    return cursor_after_each_op
```

```
def verity(student: to_test) -> None:
    ref = run(Zipper)
    chk = run(student.Zipper)
    assert_eq(ref, chk, count=20000)
    assert_eq(ref, chk, strategy=s_exp, count=3000)
    for n in range(200, 4000, 200):
        prune, ins, rewind = [], [], []

        for i in range(n):
            ins.extend(['insert_left ' + str(i), 'shift_left'])
            rewind.append('shift_right')
            prune.extend(['shift_left', 'delete_left', 'shift_left'])

        assert_eq(ref, chk, ins + rewind + prune + rewind)

import run_tests  # noqa
```

K.7.r.6 [poly]

```
class Polynomial:
    def __init__(self, coefs: list[int]) -> None:
        self.coefs = coefs.copy()
        self.coefs.reverse()
        self.normalize()

    def normalize(self) -> None:
        while len(self.coefs) > 1 and not self.coefs[-1]:
            self.coefs.pop()

    def add(self, other: 'Polynomial') -> 'Polynomial':
        result = Polynomial([])
        result.coefs = [0 for _ in range(max(len(self.coefs),
                                             len(other.coefs)))]

        for i in range(len(self.coefs)):
            result.coefs[i] += self.coefs[i]

        for i in range(len(other.coefs)):
            result.coefs[i] += other.coefs[i]

        return result

    def invert(self) -> 'Polynomial':
        result = Polynomial([])
        result.coefs = [-x for x in self.coefs]
        return result

    def subtract(self, other: 'Polynomial') -> 'Polynomial':
        return self.add(other.invert())

    def multiply(self, other: 'Polynomial') -> 'Polynomial':
        result = Polynomial([])
        result.coefs = [0 for i in range(len(self.coefs) *
                                         len(other.coefs))]

        for i in range(len(self.coefs)):
            for j in range(len(other.coefs)):
                result.coefs[i + j] += self.coefs[i] * other.coefs[j]

        result.normalize()
        return result

    def get_coefs(self) -> list[int]:
        coefs = self.coefs.copy()
        coefs.reverse()
        return coefs

import run_tests  # noqa
```

K.8: Týden 8

K.8.e.1 [sorted]

```
def is_sorted(num_list: list[int]) -> bool:
    if len(num_list) <= 1:
        return True

    for i in range(len(num_list) - 1):
        if num_list[i] > num_list[i + 1]:
            return False

    return True

import run_tests  # noqa
```

K.8.e.2 [selectsort]

```
def selectsort(num_list: list[int]) -> None:
    for i in range(len(num_list)):
        min_idx = i
        for j in range(i + 1, len(num_list)):
            if num_list[min_idx] > num_list[j]:
                min_idx = j
        num_list[i], num_list[min_idx] \
            = num_list[min_idx], num_list[i]

import run_tests  # noqa
```

K.8.e.3 [uniqbound]

```
def lower_bound(num_list: list[int], num: int) -> int | None:
    if len(num_list) == 0 or num < num_list[0]:
        return None

    left = 0
    right = len(num_list) - 1
```

```
    while left != right:
        mid = (left + right + 1) // 2

        if num_list[mid] > num:
            right = mid - 1
        else:
            left = mid

    return num_list[left]

import run_tests  # noqa
```

K.8.r.1 [bound]

```
def left_bound(array: list[int], target: int) -> int | None:
    lower, upper = 0, len(array)
    while lower < upper:
        mid = (lower + upper) // 2
        if target <= array[mid]:
            upper = mid
        else:
            lower = mid + 1

    assert lower == upper

    if lower < len(array) and array[lower] == target:
        return lower
    return None

import run_tests  # noqa
```

K.8.r.2 [nested]

```
def flatten(arr: list[list[int]]) -> list[int]:
    result = []
    for sublist in arr:
        for elem in sublist:
            result.append(elem)
    return result

def sort_nested(arr: list[list[int]]) -> list[list[int]]:
    flattened = flatten(arr)
    flattened.sort()

    result = []
    index = 0
    for nested in arr:
        sublist = []
        for _ in range(len(nested)):
            sublist.append(flattened[index])
            index += 1
        result.append(sublist)
    return result

def verity(student: to_test) -> None:
    assert_eq(sort_nested, student)
    assert_eq(sort_nested, student, strategy=s_exp, count=2000)

import run_tests  # noqa
```

K.8.r.3 [flipped]

```
def last_index(arr: list[int], idx: int) -> int:
    first = arr[idx]
    while idx < len(arr) - 1 and first == arr[idx + 1]:
        idx += 1
    return idx

def skip_run(arr: list[int], idx: int, step: int) -> int:
    first = arr[idx]
    while (idx + step < len(arr) and
           idx + step >= 0 and
           first == arr[idx + step]):
        idx += step
    return idx

def misplaced(arr: list[int]) -> list[int]:
    indices = []
    for i in range(1, len(arr)):
        if arr[i] < arr[i - 1]:
            indices.append(i)
    return indices

def is_almost_sorted_bad(arr: list[int]) -> bool:
    indices = misplaced(arr)

    if len(indices) == 0 or len(indices) > 2:
        return False

    if len(indices) == 2:
        first, second = indices[0] - 1, indices[1]
    else:
        first, second = indices[0] - 1, last_index(arr, indices[0])

    copy = arr.copy()
    copy[first], copy[second] = copy[second], copy[first]

    return len(misplaced(copy)) == 0

def is_almost_sorted(arr: list[int]) -> bool:
    indices = misplaced(arr)

    if len(indices) == 0 or len(indices) > 2:
        return False

    if len(indices) == 2:
        first, second = indices[0] - 1, indices[1]
    else:
        first, second = (skip_run(arr, indices[0] - 1, -1),
                         skip_run(arr, indices[0], 1))

    copy = arr.copy()
    copy[first], copy[second] = copy[second], copy[first]

    result = len(misplaced(copy)) == 0
# reject inputs that trigger a bug in the previous reference solution; TODO remove later

    if result:
        assert is_almost_sorted_bad(arr)
    return result

def swap(items: list[int], i: int, j: int) -> list[int]:
    items = items.copy()
    items[i], items[j] = items[j], items[i]
    return items

def verity(student: to_test) -> None:
    assert_eq(is_almost_sorted, student)

    for n in range(100, 1001, 100):
        for d in 1, 2:
            items = [i // d for i in range(n)]
            for i in range(0, n, 33):
                for j in range(0, n, 77):
                    assert_eq(is_almost_sorted, student,
                              swap(items, i, j))

import run_tests  # noqa
```

K.8.r.4 [greater]

```
def get_digit(num: int, power: int, base: int) -> int:
    return (num // base ** power) % base

def next_greater(num: int) -> int | None:
    base = 10
    swap_digit = 0
    swap_power = 0
    min_power = 0
    min_digit = base
    last = 0

    while base ** swap_power <= num:
        swap_digit = get_digit(num, swap_power, base)
        if swap_digit < last:
            break
```

```
        swap_power += 1
        last = swap_digit

    if base ** swap_power > num:
        return None

    for i in range(swap_power - 1, -1, -1):
        digit = get_digit(num, i, base)
        if digit < min_digit and digit > swap_digit:
            min_power = i
            min_digit = digit

    num += (min_digit - swap_digit) * base ** swap_power
    num += (swap_digit - min_digit) * base ** min_power

    low_order = [get_digit(num, i, base)
                 for i in range(0, min_power + 1)]

    num //= base ** (min_power + 1)
    for digit in sorted(low_order):
        num *= base
        num += digit

    return num

import run_tests  # noqa
```

K.8.r.5 [heapsort]

```
def heapsort(to_sort: list[int]) -> None:
    heapify(to_sort)

    for i in range(len(to_sort) - 1, 0, -1):
        to_sort[i], to_sort[0] = to_sort[0], to_sort[i]
        sift_down(to_sort, 0, i)

def heapify(to_heap: list[int]) -> None:
    for i in range((len(to_heap) - 1) // 2, -1, -1):
        sift_down(to_heap, i, len(to_heap))

def sift_down(heap: list[int], idx: int, heap_end: int) -> None:
    while idx < heap_end:
        left_idx = 2 * idx + 1
        right_idx = 2 * idx + 2
        largest = idx

        if left_idx < heap_end and heap[left_idx] > heap[largest]:
            largest = left_idx
        if right_idx < heap_end and heap[right_idx] > heap[largest]:
            largest = right_idx

        if largest == idx:
            break
        else:
            heap[largest], heap[idx] = heap[idx], heap[largest]
            idx = largest

import run_tests  # noqa
```

K.8.r.6 [radix]

```
def radixsort(to_sort: list[int]) -> list[int]:
    if to_sort == []:
        return []

    max_digits = digit_count(max(to_sort))
    res = to_sort
    for i in range(max_digits):
        res = counting_sort_by_digit(res, i)
    return res

def counting_sort_by_digit(to_sort: list[int], curr_digit: int) -> \
        list[int]:
    bucket_size = [0 for i in range(10)]
    bucket_start = [0 for i in range(10)]
    bucket_index = [0 for i in range(10)]
    res = [0 for i in range(len(to_sort))]

    for num in to_sort:
        bucket_size[digit(num, curr_digit)] += 1

    for i in range(1, len(bucket_size)):
        bucket_start[i] = bucket_start[i - 1] + bucket_size[i - 1]

    for num in to_sort:
        d = digit(num, curr_digit)
        res[bucket_start[d] + bucket_index[d]] = num
        bucket_index[d] += 1

    return res

def digit(num: int, pos: int) -> int:
    return (num // (10 ** pos)) % 10

def digit_count(num: int) -> int:
    result = 0
    while num > 0:
        result += 1
        num //= 10
    return result

import run_tests  # noqa
```

K.9: Týden 9

K.9.e.1 [count]

```
def count(tree: Tree | None) -> int:
    if tree is None:
        return 0

    return 1 + count(tree.left) + count(tree.right)
```

K.9.e.2 [leafsum]

```
def sum_leaves(node: Tree) -> int:
    if len(node.children) == 0:
        return node.value
    return sum([sum_leaves(child) for child in node.children])
```

K.9.e.3 [depth]

```
def depth(tree: Tree | None) -> int:
    if tree is None:
        return 0

    return 1 + max(depth(tree.left), depth(tree.right))
```

K.9.r.1 [treesum]

```
def sum_tree(node: Tree | None) -> int:
    if node is None:
        return 0
    return (node.value +
            sum_tree(node.first) +
            sum_tree(node.second) +
            sum_tree(node.third))
```

K.9.r.3 [heap]

```
def is_heap(tree: Tree | None) -> bool:
    if tree is None:
        return True

    if not heap_property_check(tree):
        return False

    return is_heap(tree.left) and is_heap(tree.right)

def heap_property_check(node: Tree) -> bool:
    if node.left is not None and node.left.key > node.key:
        return False
    if node.right is not None and node.right.key > node.key:
        return False
    return True
```

K.9.r.4 [avl]

```
def is_avl(tree: Tree | None) -> bool:
    ok, _ = is_avl_rec(tree)
    return ok

def is_avl_rec(tree: Tree | None) -> tuple[bool, int]:
    if tree is None:
        return (True, 0)

    l_avl, l_depth = is_avl_rec(tree.left)
    r_avl, r_depth = is_avl_rec(tree.right)

    return (l_avl and r_avl and abs(l_depth - r_depth) <= 1,
            max(l_depth, r_depth) + 1)
```

K.9.r.5 [bdd]

```
def evaluate_bdd(bdd: BDD, true_vars: set[str]) -> bool:
    if bdd.val == "1":
        return True
    if bdd.val == "0":
        return False

    assert bdd.left is not None and bdd.right is not None

    next_bdd = bdd.right if bdd.val in true_vars else bdd.left
    return evaluate_bdd(next_bdd, true_vars)
```

K.9.r.6 [average]

```
def average_branch_len(tree: Tree | None) -> float:
    if tree is None:
        return 0
    branch_lens = all_branch_lens(tree)
    return float(sum(branch_lens)) / len(branch_lens)

def all_branch_lens(tree: Tree) -> list[int]:
    res: list[int] = []
    all_branch_lens_rec(tree, 1, res)
    return res

def all_branch_lens_rec(tree: Tree,
                        curr_depth: int, lens: list[int]) -> None:
    if tree.left is None and tree.right is None:
        lens.append(curr_depth)
        return

    for child in [tree.left, tree.right]:
        if child is not None:
            all_branch_lens_rec(child, curr_depth + 1, lens)
```

K.10: Týden 10

K.10.e.1 [subsets]

```
def subsets(original: set[int]) -> list[set[int]]:
    result: list[set[int]] = [set()]
    subsets_rec(list(original), result)
    return result

def subsets_rec(original: list[int],
                result: list[set[int]]) -> None:
    if not original:
        return
    curr_num = original.pop()
    to_add: list[set[int]] = []
    for curr_set in result:
        to_add.append(curr_set | {curr_num})
    result.extend(to_add)
    subsets_rec(original, result)

import run_tests  # noqa
```

K.10.e.3 [flatten]

```
def flatten(to_flatten: NestedList) -> list[int]:
    flattened: list[int] = []
    for item in to_flatten:
        if isinstance(item, list):
            flattened.extend(flatten(item))
        else:
            flattened.append(item)
    return flattened
```

K.10.r.1 [subseq]

```
def subseq(seq: list[int]) -> list[list[int]]:
    res: list[list[int]] = []
    candidates: list[list[int]] = [[]]
    subseq_rec(seq, 0, candidates)

    for candidate in sorted(candidates):
        if not res or res[-1] != candidate:
            res.append(candidate)

    return res

def subseq_rec(seq: list[int],
               curr_pos: int,
               res: list[list[int]]) -> None:
    assert curr_pos >= 0 and curr_pos <= len(seq)

    if curr_pos == len(seq):
        return

    to_add = []
    for curr_seq in res:
        if len(curr_seq) == 0 or curr_seq[-1] <= seq[curr_pos]:
            if len(curr_seq) + 1 != len(seq):
                new_seq = curr_seq.copy()
                new_seq.append(seq[curr_pos])
                to_add.append(new_seq)

    res.extend(to_add)
    subseq_rec(seq, curr_pos + 1, res)

import run_tests  # noqa
```

K.10.r.2 [equivalence]

```
Pair = tuple[int, int]

def partition2pairs(partition: list[set[int]]) -> set[Pair]:
    result = set()
    for subset in partition:
        for elem1 in subset:
            for elem2 in subset:
                result.add((elem1, elem2))
    return result

def pairs2partition(pairs: set[Pair]) -> list[set[int]]:
    partitions: dict[int, set[int]] = {}
    for (a, b) in pairs:
        partitions[a] = partitions.get(a, {a}) | {b}

    all_elements = set(partitions.keys())
    result: list[set[int]] = []
    for element, partition in partitions.items():
        if element not in all_elements:
            continue
        result.append(partition)
        for elem in partition:
            all_elements.remove(elem)

    return result

import run_tests  # noqa
```

K.10.r.4 [nested]

```
NestedList = list['int | NestedList']

def copy(self: int | NestedList) -> int | NestedList:
    if isinstance(self, int):
        return self
    else:
        return [copy(i) for i in self]

def nested_enum(idx: int) -> NestedList:
```

```
    items: NestedList = []
    for sub in list_enum(idx):
        sub, nest = divmod(sub, 2)
        if nest:
            items.append(nested_enum(sub))
        else:
            items.append(sub)
    return items

NestedListGen = Annotated[NestedList, nested_enum]

def flatten(to_flatten: NestedList, result: list[int]) -> list[int]:
    for item in to_flatten:
        if isinstance(item, int):
            result.append(item)
        else:
            flatten(item, result)
    return result

def fill(nested: NestedList, values: list[int], index: int) -> int:
    for i, item in enumerate(nested):
        if isinstance(item, int):
            nested[i] = values[index]
            index += 1
        else:
            index = fill(item, values, index)
    return index

def sort_nested(nested: NestedListGen) -> None:
    flat = flatten(nested, [])
    flat.sort()
    fill(nested, flat, 0)

def verity(student: to_test) -> None:
    assert_eq(sort_nested, student)
    assert_eq(sort_nested, student, strategy=s_exp, count=2000)

    nest_1: NestedList = [1, 3, 2]
    nest_2 = copy(nest_1)
    nest_5 = copy(nest_1)

    for n in range(12):
        nest_2 = [copy(nest_2), copy(nest_2)]
        nest_5 = [5, copy(nest_5), 1, copy(nest_5), 7]
        assert_eq(sort_nested, student, copy(nest_2))
        assert_eq(sort_nested, student, copy(nest_5))

    for n in range(120):  # ± deepcopy recursion limit
        nest_1 = [copy(nest_1)]
        assert_eq(sort_nested, student, nest_1.copy())

import run_tests  # noqa
```

K.10.r.5 [**subsetsum**]

```
def subset_sum_rec(nums: list[int], total: int,
                   start: int) -> None | set[int]:
    if total == 0:
        return set()

    for i in range(start, len(nums)):
        num = nums[i]
        if num > total:
            return None

        result = subset_sum_rec(nums, total - num, i + 1)
        if result is not None:
            result.add(num)
            return result

    return None

def subset_sum(nums: set[nat1], total: nat1) -> None | set[int]:
    return subset_sum_rec(sorted(nums), total, 0)

def validate_sum(i_arg: to_test, f_result: to_test,
                 g_result: to_test) -> bool:
    if f_result is None or g_result is None:
        return f_result == g_result

    numbers, total = i_arg
    return (isinstance(g_result, set) and
            sum(f_result) == sum(g_result) and g_result <= numbers)

def verity(student: to_test) -> None:
    assert_eq(subset_sum, student, count=10000,
              validate_result=validate_sum)
    for index in list(s_gamma(2500)) + list(s_mult()(800)):
        numbers = set_enum(index)
        for total in (sum(i for i in numbers if i % 2 == 0),
                      sum(i for i in numbers if i % 2 == 1),
                      sum(sorted(numbers)[-5:-1]),
                      sum(sorted(numbers)[::2]),
                      sum(sorted(numbers)[::3])):
            assert_eq(subset_sum, student, numbers, total,
                      validate_result=validate_sum)
            assert_eq(subset_sum, student,
                      {n + 101 for n in numbers},
                      101 * len(numbers) // 2 + total,
                      validate_result=validate_sum)

import run_tests  # noqa
```

K.10.r.6 [**dnfsat**]

```
def satisfiable(phi: Formula) -> bool:
    for clause in phi:
        curr_vars: dict[str, bool] = {}
        contradiction_found = False

        for variable, value in clause:
            if curr_vars.get(variable, value) != value:
                contradiction_found = True
            curr_vars[variable] = value

        if not contradiction_found:
            return True

    return False
```

K.11: Týden 11

K.11.e.1 [**names**]

```
def names_sorted_tuples(filename: str) -> list[tuple[str, str]]:
    names = []
    with open(filename) as file:
        for line in file.readlines():
            items = line.split(',')
            if len(items) == 2:
                names.append((items[1], items[0]))
    names.sort()
    return names

def names_sorted(filename: str) -> list[str]:
    return [first + ' ' + last for last, first in
            names_sorted_tuples(filename)]

def format_names(source: str, dest: str) -> None:
    with open(dest, 'w') as formatted:
        for last, first in names_sorted_tuples(source):
            formatted.write(first + ',' + last + "\n")

import run_tests  # noqa
```

K.11.e.3 [**wordfreq**]

```
def most_common(path: str) -> list[str]:

    with open(path) as file:
        all_words = file.read().split()

        word_freq: dict[str, int] = {}
        for word in all_words:
            word = "".join([char for char in word if char.isalpha()])
```

```
            word = word.lower()
            word_freq[word] = word_freq.get(word, 0) + 1

        items = [(-freq, word) for word, freq in word_freq.items()]
        result = []
        for i, (_, word) in enumerate(sorted(items)):
            if i == 3:
                break
            result.append(word)
        return result

import run_tests  # noqa
```

K.11.r.1 [brackets]

```
def tree_to_expr(node: Tree) -> str:
    if node.left is None or node.right is None:
        return node.value

    return "".join(["(", tree_to_expr(node.left),
                    " ", node.value,
                    " ", tree_to_expr(node.right),
                    ")"])
```

K.11.r.2 [ipv4fix]

```
def split(orig: str, index: int) -> tuple[str, str]:
    left = right = ''
    for i in range(index):
        left += orig[i]
    for i in range(index, len(orig)):
        right += orig[i]
    return left, right

def decode_decimal(digits: str) -> int:
    result = 0
    table = {"0": 0, "1": 1, "2": 2, "3": 3, "4": 4,
             "5": 5, "6": 6, "7": 7, "8": 8, "9": 9}

    for digit in digits:
        result *= 10
        result += table[digit]

    return result

def ipv4_restore_rec(digits: str, count: int, current: list[str],
                     result: set[str]) -> set[str]:
    if count == 0:
        if digits == "":
            result.add(".".join(current))
        return result

    for i in range(1, len(digits) + 1):
        left, right = split(digits, i)
        if decode_decimal(left) >= 256:
            break
        current.append(left)
        ipv4_restore_rec(right, count - 1, current, result)
        current.pop()

    return result

def ipv4_restore(digits: str) -> set[str]:
    return ipv4_restore_rec(digits, 4, [], set())

import run_tests  # noqa
```

K.11.r.3 [trailing]

```
def trailing() -> None:
argv[0] is the name of the program

    for i in range(1, len(sys.argv)):
        print("working on", sys.argv[i])
        trailing_from_file(sys.argv[i])

def trailing_from_file(filename: str) -> None:
    lines: list[str]

    with open(filename, "r") as trail_file:
        lines = trail_file.readlines()
        for i in range(len(lines)):
            lines[i] = lines[i].rstrip()

    with open(filename, "w") as trail_file:
        for line in lines:
            trail_file.write(line + "\n")

import run_tests  # noqa
```

K.11.r.4 [correct]

```
def autocorrect(dict_file: str, input_file: str,
                output_file: str) -> None:
    dictionary = read_dictionary(dict_file)

    with open(input_file) as file:
        text = file.read()
        word = ""

        with open(output_file, "w") as out:
            for char in text:
                if char.isalpha():
                    word += char
                else:
                    out.write(corrected_word(word, dictionary))
                    word = ""
                    out.write(char)
            out.write(corrected_word(word, dictionary))

def corrected_word(word: str,
                   dictionary: dict[int, set[str]]) -> str:
    words = dictionary.get(len(word), set())
    word = word.lower()
    if not words or word in words:
        return word
    return best_correction(word, words)

def read_dictionary(path: str) -> dict[int, set[str]]:
    res: dict[int, set[str]] = {}
    with gzip.open(path, 'rt') as data:
        for word in data:
            word = word.strip()
            key = len(word)
            if key not in res:
                res[key] = set()
            res[key].add(word)
    return res

def best_correction(word: str, matches: set[str]) -> str:
    candidates = closest_by_hamming(word, matches)
    return min(closest_by_ends(word, candidates))

import run_tests  # noqa
```

K.11.r.5 [language]

```
def recognize_language(lang_data: dict[str, dict[str, int]],
                       text_file: str) -> str:
    lang_freqs = lang_vectors(lang_data)
    file_freq = letter_freq_vector(text_file)

    min_angle = pi
    min_lang = ''

    for lang, lang_freq in lang_freqs.items():
        a = vector_angle(file_freq, lang_freq)
        if a < min_angle:
            min_angle = a
            min_lang = lang

    return min_lang

def letter_freq_vector(filename: str) -> list[int]:
```

```
    freqs = [0 for i in range(26)]
    indices = enumerate(list("abcdefghijklmnopqrstuvwxyz"))
    letters = dict([(letter, idx) for idx, letter in indices])

    with open(filename) as file:
        text = file.read()

        for char in text:
            if "a" <= char <= "z" or "A" <= char <= "Z":
                freqs[letters[char.lower()]] += 1

    return freqs

def lang_vectors(languages: dict[str, dict[str, int]]) \
        -> dict[str, list[int]]:
    res: dict[str, list[int]] = {}
    for language, freqs in languages.items():
        res[language] = [y for x, y in freqs.items()]

    return res

def vector_angle(v1: list[int], v2: list[int]) -> float:
    assert len(v1) == len(v2)

    dot_product = sum([v1[i] * v2[i] for i in range(len(v1))])
    len_v1 = sqrt(sum([x ** 2 for x in v1]))
    len_v2 = sqrt(sum([x ** 2 for x in v2]))
    return acos(dot_product / (len_v1 * len_v2))

import run_tests  # noqa
```

K.11.r.6 [weighted]

```
def add_char_to_words(words: set[str], char: str) -> set[str]:
    result = set()
    for word in words:
        result.add(char + word)
    return result

def weighted_words(length: nat, weight: nat) -> set[str]:
    if weight == 0 and length == 0:
        return {""}

    if weight > length or length == 0:
        return set()

    to_add_zero = weighted_words(length - 1, weight)
    to_add_nonzero = weighted_words(length - 1, weight - 1)
    result = add_char_to_words(to_add_zero, "0")
    result.update(add_char_to_words(to_add_nonzero, "1"))
    result.update(add_char_to_words(to_add_nonzero, "2"))

    return result

def verity(student: to_test) -> None:
    for length in range(15):
        for weight in range(8):
            assert_eq(weighted_words, student, length, weight)

import run_tests  # noqa
```

K.12: Týden 12

K.12.e.1 [wormhole]

```
def is_allowed(num: int, allowed: set[int]) -> bool:
    while num > 0:
        if num % 10 not in allowed:
            return False
        num //= 10
    return True

def wormhole(nums: list[int], allowed: set[int]) -> list[int]:
    return [num for num in nums if is_allowed(num, allowed)]

import run_tests  # noqa
```

K.12.e.2 [wordwrap]

```
def word_wrap(orig: str, max_line_len: int) -> str:
    chars = list(orig)
    cur_line_len = 0
    last_space = None

    for index in range(len(chars)):
        cur_line_len += 1

        if chars[index] == "\n":
            cur_line_len = 0

        if chars[index] == " ":
            last_space = index

        if cur_line_len > max_line_len:
            if last_space is not None:
                chars[last_space] = "\n"
                cur_line_len = index - last_space

    return "".join(chars)

import run_tests  # noqa
```

K.12.e.3 [bounds]

```
def get_bounds(tree: Tree) -> tuple[int, int]:
    return get_bounds_rec(tree, tree.value, tree.value)

def get_bounds_rec(tree: Tree | None,
                   low: int, high: int) -> tuple[int, int]:
    if tree is None:
        return (low, high)

    low = min(tree.value, low)
    high = max(tree.value, high)
    low, high = get_bounds_rec(tree.left, low, high)
    return get_bounds_rec(tree.right, low, high)
```

K.12.r.1 [walk]

```
DIRS = {
    '←': (-1, 0),
    '→': (+1, 0),
    '↑': (0, +1),
    '↓': (0, -1),
}

def step(direction: str, pos: Position) -> Position:
    x, y = pos
    dx, dy = DIRS[direction]
    return (x + dx, y + dy)

def walk(path: str, pos: Position) -> Position:
    for direction in path:
        pos = step(direction, pos)
    return pos

def meet(path_1: str, path_2: str, pos_1: Position,
         pos_2: Position) -> Position | None:
    if pos_1 == pos_2:
        return pos_1

    for i in range(max(len(path_1), len(path_2))):
        if i < len(path_1):
            pos_1 = step(path_1[i], pos_1)
        if i < len(path_2):
            pos_2 = step(path_2[i], pos_2)
        if pos_1 == pos_2:
            return pos_1
    return None
```

K.12.r.2 [arraylist]

```
class Node:
    def __init__(self) -> None:
        self.data: list[int] = []
        self.next: 'Node | None' = None

class ArrayList:
    def __init__(self, capacity: int) -> None:
```

```
    self.capacity = capacity
    self.head: Node | None = None
    self.tail: Node | None = None

def append(self, value: int) -> None:
    if self.head is None:
        self.head = Node()
        self.tail = self.head

    assert self.tail is not None

    if len(self.tail.data) == self.capacity:
        node = Node()
        self.tail.next = node
        self.tail = node

    self.tail.data.append(value)

def delete(self, value: int) -> None:
    node = self.head
    prev = None
    while node is not None:
        if value in node.data:
            if len(node.data) == 1:
                self.unlink(prev, node)
            else:
                node.data.pop(node.data.index(value))
            return
        prev = node
        node = node.next

def unlink(self, prev: Node | None, node: Node) -> None:
    if prev is None:
        self.head = node.next
    else:
        prev.next = node.next
    if node == self.tail:
        self.tail = prev

def compact(self) -> None:
    node = self.head
    while node is not None:
        self.move_to(node)  # fill current node
        node = node.next

def move_to(self, node: Node) -> None:
    fit = self.capacity - len(node.data)
    if node.next is None or fit == 0:
        return

    if len(node.next.data) <= fit:
        node.data.extend(node.next.data)
        self.unlink(node, node.next)
        self.move_to(node)  # need more data → tail-recurse
    else:
        trimmed = []
        for idx, val in enumerate(node.next.data):
            if idx < fit:
                node.data.append(val)
            else:
                trimmed.append(val)
        node.next.data = trimmed

import run_tests  # noqa
```

K.12.r.3 [cycle]

```
class Stream:
    def __init__(self, data: list[int]) -> None:
        self.data = data
        self.pos = 0
        self.step = 1

    def get(self) -> int:
        elem = self.data[self.pos]
        self.pos = (self.pos + self.step) % len(self.data)
        return elem

def cycle(data: list[int]) -> Stream:
    return Stream(data.copy())

def drop(n: int, original: Stream) -> Stream:
    stream = Stream(original.data.copy())
    stream.step = original.step
    stream.pos = (original.pos + n * stream.step) % len(stream.data)
    return stream

def take(n: int, original: Stream) -> list[int]:
    result = []
    for _ in range(n):
        result.append(original.get())
    return result

def every_nth(n: int, original: Stream) -> Stream:
    stream = Stream(original.data.copy())
    stream.pos = original.pos
    stream.step = (original.step * n) % len(original.data)
    return stream

import run_tests  # noqa
```

K.12.r.4 [stream]

```
class FinStream:
    def __init__(self, data: list[int]) -> None:
        self.data = data
        self.pos = 0

    def take_head(self) -> tuple[int | None, 'Stream']:
        if self.pos >= len(self.data):
            return (None, self)

        tail = FinStream(self.data)
        tail.pos = self.pos + 1
        return (self.data[self.pos], tail)

class Cycle:
    def __init__(self, inner: 'Stream') -> None:
        self.inner = inner
        self.orig = inner

    def take_head(self) -> tuple[int | None, 'Stream']:
        tail = Cycle(self.orig)
        head, tail.inner = self.inner.take_head()
        if head is None:
            head, tail.inner = self.orig.take_head()
        return (head, tail)

class Drop:
    def __init__(self, n: int, inner: 'Stream') -> None:
        self.inner = inner
        for _ in range(n):
            _, self.inner = self.inner.take_head()

    def take_head(self) -> tuple[int | None, 'Stream']:
        return self.inner.take_head()

class Take:
    def __init__(self, n: int, inner: 'Stream') -> None:
        self.n = n
        self.inner = inner

    def take_head(self) -> tuple[int | None, 'Stream']:
        if self.n == 0:
            return None, self

        tail = Take(self.n - 1, self.inner)
        head, tail.inner = self.inner.take_head()
        return (head, tail)

class Skip:
    def __init__(self, inner: 'Stream', skips: 'Stream') -> None:
        self.inner = inner
```

```
        self.skips = skips

    def take_head(self) -> tuple[int | None, 'Stream']:
        head, inner_tail = self.inner.take_head()
        skip, skips_tail = self.skips.take_head()

        if skip is not None:
            for _ in range(skip):
                head_skipped, inner_tail = inner_tail.take_head()

        return (head, Skip(inner_tail, skips_tail))

Stream = FinStream | Cycle | Drop | Take | Skip

def to_stream(data: list[int]) -> Stream:
    return FinStream(data.copy())

def cycle(stream: Stream) -> Stream:
    return Cycle(stream)

def drop(n: int, original: Stream) -> Stream:
    return Drop(n, original)

def take(n: int, original: Stream) -> Stream:
    return Take(n, original)

def skip(inner: Stream, skips: Stream) -> Stream:
    return Skip(inner, skips)
```

K.12.r.5 [disjoint]

```
def to_digits(n: int) -> list[int]:
    if n == 0:
        return [0]
    out = []
    while n > 0:
        out.append(n % 10)
        n //= 10
    return out

def from_digits(digits: list[int]) -> int | None:
    if not digits:
        return None
    out = 0
    for d in digits:
        out *= 10
        out += d
    return out

def nearest_disjoint(n: int) -> int | None:
    digits = to_digits(n)
    available = set(range(10)) - set(digits)
    tail_len = len(digits) - 1

    if not available:
        return None

    first = digits[-1]
    big_digit = max(available)
    small_digit = min(available)
    small_nonzero = 0 if available == {0} else min(available - {0})

    first_small = [x for x in available if x < first]
    first_big = [x for x in available if x > first]

    lead_small = [max(first_small)] if first_small else []
    lead_big = [min(first_big)] if first_big else [small_nonzero, small_digit]

    tail_big = [big_digit for i in range(tail_len)]
    tail_small = [small_digit for i in range(tail_len)]
    smaller = from_digits(lead_small + tail_big)
    bigger = from_digits(lead_big + tail_small)

    if smaller is not None and bigger is not None and n - smaller < bigger
- n:
        return smaller
    return bigger

import run_tests  # noqa
```

K.12.r.6 [poly]

```
def poly_to_str(coefs: list[int]) -> str:
    result = ""

    for i in range(len(coefs)):
        curr_term = term_to_string(coefs[i], len(coefs) - i - 1, i !=
0)
        if curr_term != "":
            result += curr_term + " "

    if result == "":
        return "0"

    return result.rstrip()

def digit_to_int(digits: str, table: dict[str, int]) -> int:
    number = 0

    for curr in digits:
        number *= 10
        number += table[curr]

    return number

def int_to_digits(num: int, digits: str) -> str:
    out = ''
    while num > 0:
        out = digits[num % 10] + out
        num //= 10
    return out

def upper_index_to_int(idx: str) -> int:
    return digit_to_int(idx,
                        {"⁰": 0, "¹": 1, "²": 2, "³": 3, "⁴": 4,
                         "⁵": 5, "⁶": 6, "⁷": 7, "⁸": 8, "⁹": 9})

def coef_to_int(coef: str) -> int:
    table = {"+": 0, "-": 0,
             "0": 0, "1": 1, "2": 2, "3": 3, "4": 4,
             "5": 5, "6": 6, "7": 7, "8": 8, "9": 9}
    sign = -1 if coef[0] == '-' else 1
    return sign * digit_to_int(coef, table)

def coef_exp_from_term(term: str) -> tuple[int, int]:
    parts = term.split("x")
    if len(parts) == 1:
        return (coef_to_int(term), 0)

    coef, power = parts
    if coef == "+" or coef == "-":
        coef += "1"

    if power == "":
        return (coef_to_int(coef), 1)

    return (coef_to_int(coef), upper_index_to_int(power))

def get_terms_from_str_poly(poly: str) -> list[str]:
    tokens = poly.split()
    terms = []

    start = 0
    if len(tokens) % 2 == 1:
        terms.append(tokens[0])
        start = 1

    for i in range(start, len(tokens), 2):
        terms.append(tokens[i] + tokens[i + 1])

    return terms

def str_to_poly(poly: str) -> list[int]:
    terms = get_terms_from_str_poly(poly)

    coefs = []
    last_exp = -1
```

```
    for term in terms:
        coef, exp = coef_exp_from_term(term)

        while last_exp > exp + 1:
            coefs.append(0)
            last_exp -= 1

        coefs.append(coef)
        last_exp = exp

    while last_exp != 0:
        coefs.append(0)
        last_exp -= 1

    return coefs

def int_to_upper_index(num: int) -> str:
    return int_to_digits(num, "⁰¹²³⁴⁵⁶⁷⁸⁹")

def term_to_string(coef: int, power: int, sign: bool) -> str:
    if coef == 0:
        return ""

    term = "- " if coef < 0 else ("+ " if sign else "")
    coef = abs(coef)

    if coef != 1:
        term += int_to_digits(coef, "0123456789")

    if power >= 1:
        term += "x"

    if power > 1:
        term += int_to_upper_index(power)

    return term

import run_tests  # noqa
```

Tato kapitola obsahuje informace o technické realizaci předmětu, a to zejména:

- jak se pracuje s kostrami úloh,
- jak sdílet obrazovku (terminál) ve cvičení,
- jak se odevzdávají úkoly,
- kde najdete výsledky testů a jak je přečtete,
- kde najdete hodnocení kvality kódu (učitelské recenze),
- jak získáte kód pro vzájemné recenze.

T.1: Informační systém

Informační systém tvoří primární „rozhraní“ pro stahování studijních materiálů, odevzdávání řešení, získání výsledků vyhodnocení a čtení recenzí. Zároveň slouží jako hlavní komunikační kanál mezi studenty a učiteli, prostřednictvím diskusního fóra.

T.1.1 Diskusní fórum Máte-li dotazy k úlohám, organizaci, atp., využijte k jejich položení prosím vždy přednostně diskusní fórum.[31] Ke každé kapitole a ke každému příkladu ze sady vytvoříme samostatné vlákno, kam patří dotazy specifické pro tuto kapitolu nebo tento příklad. Pro řešení obecných organizačních záležitostí a technických problémů jsou podobně v diskusním fóru nachystaná vlákna.

Než položíte libovolný dotaz, přečtěte si relevantní část dosavadní diskuse – je možné, že na stejný problém už někdo narazil. Máte-li ve fóru dotaz, na který se Vám nedostalo do druhého pracovního dne reakce, připomeňte se prosím tím, že na tento svůj příspěvek odpovíte.

Máte-li dotaz k výsledku testu, nikdy tento výsledek nevkládejte do příspěvku (podobně nikdy nevkládejte části řešení příkladu). Učitelé mají přístup k obsahu Vašich poznámkových bloků, i k Vámi odevzdaným souborům. Je-li to pro pochopení kontextu ostatními čtenáři potřeba, odpovídající učitel chybějící informace doplní dle uvážení.

T.1.2 Stažení koster Kostry naleznete ve **studijních materiálech** v ISu: Student → IB111 → Studijní materály → Učební materiály. Každá kapitola má vlastní složku, pojmenovanou 00 (tento úvod a materiály k nultému cvičení), 01 (první běžná kapitola), 02, …, 12. Veškeré soubory stáhnete jednoduše tak, že na složku kliknete pravým tlačítkem a vyberete možnost Stáhnout jako ZIP. Stažený soubor rozbalte a můžete řešit.

T.1.3 Odevzdání řešení Vypracované příklady můžete odevzdat do **odevzdárny** v ISu: Student → IB111 → Odevzdávárny. Pro přípravy používejte odpovídající složky s názvy 01, …, 12. Pro příklady ze sad pak s1_a_csv, atp. (složky začínající s1 pro první, s2 pro druhou a s3 pro třetí sadu).

Soubor vložíte výběrem možnosti Soubor – nahrát (první ikonka na liště nad seznamem souborů). Tímto způsobem můžete najednou nahrát souborů několik (například všechny přípravy z dané kapitoly). Vždy se ujistěte, že vkládáte správnou verzi souboru (a že nemáte v textovém editoru neuložené změny). **Pozor!** Všechny vložené soubory se musí jmenovat stejně jako v kostrách, jinak nebudou rozeznány (IS při vkládání automaticky předřadí Vaše UČO – to je v pořádku, název souboru po vložení do ISu **neměňte**) .

O každém odevzdaném souboru (i nerozeznaném) se Vám v poznámkovém bloku log objeví záznam. Tento záznam i výsledky testu syntaxe by se měl objevit do několika minut od odevzdání (nemáte-li ani po 15 minutách výsledky, napište prosím do diskusního fóra).

Archiv všech souborů, které jste úspěšně odevzdali, naleznete ve složce Private ve studijních materiálech (Student → IB111 → Studijní materiály → Private).

T.1.4 Výsledky automatických testů Automatickou zpětnou vazbu k odevzdaným úlohám budete dostávat prostřednictvím tzv. **poznámkových bloků** v ISu. Ke každé odevzdávárně existuje odpovídající poznámkový blok, ve kterém naleznete aktuální výsledky testů. Pro přípravy bude blok vypadat přibližně takto:

```
testing verity of submission from 2025-09-17 22:43 CEST
subtest p1_foo passed    [  1]
subtest p2_bar failed
subtest p3_baz failed
subtest p4_quux passed   [  1]
subtest p5_wibble passed [  1]
subtest p6_xyzzy failed
   {bližší popis chyby}
verity test failed

testing syntax of submission from 2025-09-17 22:43 CEST
subtest p1_foo passed
subtest p2_bar failed
  {bližší popis chyby}
subtest p3_baz failed
  {bližší popis chyby}
subtest p4_quux passed
subtest p5_wibble passed
subtest p6_xyzzy passed
syntax test failed

testing sanity of submission from 2025-09-17 22:43 CEST
subtest p1_foo passed    [  1]
subtest p2_bar failed
subtest p3_baz failed
subtest p4_quux passed   [  1]
subtest p5_wibble passed [  1]
subtest p6_xyzzy passed  [  1]
sanity test failed

best submission: 2025-09-17 22:43 CEST worth *7 point(s)
```

Jednak si všimněte, že každý odstavec má **vlastní časové razítko**, které určuje, ke kterému odevzdání daný výstup patří. Tato časová razítka nemusí být stejná. V hranatých závorkách jsou uvedeny dílčí body, za hvězdičkou na posledním řádku pak celkový bodový zisk za tuto kapitolu.

Také si všimněte, že best submission se vztahuje na jedno konkrétní odevzdání jako celek: v situaci, kdy odstavec „verity“ a odstavec „sanity“ nemají stejné časové razítko, **nemusí** být celkový bodový zisk součtem všech dílčích bodů. O konečném zisku rozhoduje vždy poslední odevzdání před příslušným termínem (opět jako jeden celek).[32]

Výstup pro příklady ze sad je podobný, uvažme například:

```
testing verity of submission from 2025-10-11 21:14 CEST
subtest foo-small passed
subtest foo-large passed
verity test passed          [  7]

testing syntax of submission from 2025-10-14 23:54 CEST
subtest build passed
syntax test passed

testing sanity of submission from 2025-10-14 23:54 CEST
subtest foo passed
sanity test passed

best submission: 2025-10-11 21:14 CEST worth *7 point(s)
```

[31] Nebojte se do fóra napsat – když si s něčím nevíte rady a/nebo nemůžete najít v materiálech, rádi Vám pomůžeme nebo Vás nasměrujeme na místo, kde odpověď naleznete.

[32] Můžete si tak odevzdáním nefunkčních řešení na poslední chvíli snížit výsledný bodový zisk. Uvažte situaci, kdy máte v pátek 4 body za sanity příkladů p1, p2, p3, p6 a 1 bod za verity p1, p2. V sobotu odevzdáte řešení, kde p1 neprochází sanity testem, ale p4 ano a navíc projdou verity testy příklady p4 a p6. Váš výsledný zisk bude 5.5 bodu. Tento mechanismus Vám nikdy nesníží výsledný bodový zisk pod již jednou dosaženou hranici „best submission“.

Opět si všimněte, že časová razítka se mohou lišit (a v případě příkladů ze sady bude k této situaci docházet poměrně často, vždy tedy nejprve ověřte, ke kterému odevzdání se který odstavec vztahuje a pak až jej dále interpretujte).

T.1.5 Další poznámkové bloky Blok corr obsahuje záznamy o manuálních bodových korekcích (např. v situaci, kdy byl Váš bodový zisk ovlivněn chybou v testech). Podobně se zde objeví záznamy o penalizaci za opisování.

Blok log obsahuje záznam o všech odevzdaných souborech, včetně těch, které nebyly rozeznány. Nedostanete-li po odevzdání příkladu výsledek testů, ověřte si v tomto poznámkovém bloku, že soubor byl správně rozeznán.

Blok misc obsahuje záznamy o Vaší aktivitě ve cvičení (netýká se bodů za vzájemné recenze ani vnitrosemestrální testy). Nemáte-li před koncem cvičení, ve kterém jste řešili příklad u tabule, záznam v tomto bloku, připomeňte se svému cvičícímu.

Konečně blok sum obsahuje souhrn bodů, které jste dosud získali, a které ještě získat můžete. Dostanete-li se do situace, kdy Vám ani zisk všech zbývajících bodů nebude stačit pro splnění podmínek předmětu, tento blok Vás o tom bude informovat. Tento blok má navíc přístupnou statistiku bodů – můžete tak srovnat svůj dosavadní bodový zisk se svými spolužáky.

Je-li blok sum v rozporu s pravidly uvedenými v tomto dokumentu, přednost mají pravidla zde uvedená. Podobně mají v případě nesrovnalosti přednost dílčí poznámkové bloky. Dojde-li k takovéto neshodě, informujte nás o tom prosím v diskusním fóru. Případná známka uvedená v poznámkovém bloku sum je podobně pouze informativní – rozhoduje vždy známka zapsaná v hodnocení předmětu.

T.2: Studentský server aisa

Použití serveru aisa pro odevzdávání příkladů je zcela volitelné a vše potřebné můžete vždy udělat i prostřednictvím ISu. Nevíte-li si s něčím z níže uvedeného rady, použijte IS.

Na server aisa se přihlásíte programem ssh, který je k dispozici v prakticky každém moderním operačním systému (v OS Windows skrze WSL[33] – Windows Subsystem for Linux). Konkrétní příkaz (za xlogin doplňte ten svůj):

```
$ ssh xlogin@aisa.fi.muni.cz
```

Program se zeptá na heslo: použijte to fakultní (to stejné, které používáte k přihlášení na ostatní fakultní počítače, nebo např. ve fadmin-u nebo fakultním gitlab-u).

T.2.1 Pracovní stanice Veškeré instrukce, které zde uvádíme pro použití na stroji aisa platí beze změn také na libovolné školní UNIX-ové pracovní stanici (tzn. z fakultních počítačů není potřeba se hlásit na stroj aisa, navíc mají sdílený domovský adresář, takže svoje soubory z tohoto serveru přímo vidíte, jako by byly uloženy na pracovní stanici).

T.2.2 Stažení koster Aktuální zdrojový balík stáhnete příkazem:

```
$ ib111 update
```

Stažené soubory pak naleznete ve složce ~/ib111. Je bezpečné tento příkaz použít i v případě, že ve své kopii již máte rozpracovaná řešení – systém je při aktualizaci nepřepisuje. Došlo-li ke změně kostry u příkladu, který máte lokálně modifikovaný, aktualizovanou kostru naleznete v souboru s dodatečnou příponou .pristine, např. 01/e2_concat.cpp.pristine. V takovém případě si můžete obě verze srovnat příkazem diff:

```
$ diff -u e2_concat.cpp e2_concat.cpp.pristine
```

Případné relevantní změny si pak již lehce přenesete do svého řešení.

Krom samotného zdrojového balíku Vám příkaz ib111 update stáhne i veškeré recenze (jak od učitelů, tak od spolužáků). To, že máte k dispozici nové recenze, uvidíte ve výpisu. Recenze najdete ve složce ~/ib111/reviews.

T.2.3 Odevzdání řešení Odevzdat vypracované (nebo i rozpracované) řešení můžete ze složky s relevantními soubory takto:

```
$ cd ~/ib111/01
$ ib111 submit
```

Přidáte-li přepínač --wait, příkaz vyčká na vyhodnocení testů fáze „syntax“ a jakmile je výsledek k dispozici, vypíše obsah příslušného poznámkového bloku. Chcete-li si ověřit co a kdy jste odevzdali, můžete použít příkaz

```
$ ib111 status
```

nebo se podívat do informačního systému (blíže popsáno v sekci T.1).

Pozor! Odevzdáváte-li stejnou sadu příprav jak v ISu tak prostřednictvím příkazu ib111, ujistěte se, že odevzdáváte vždy všechny příklady.

T.2.4 Sdílení terminálu Řešíte-li příklad typu r ve cvičení, bude se Vám pravděpodobně hodit režim sdílení terminálu s cvičícím (který tak bude moct promítat Váš zdrojový kód na plátno, případně do něj jednoduše zasáhnout).

Protože se sdílí pouze terminál, budete se muset spokojit s negrafickým textovým editorem (doporučujeme použít micro, případně vim umíte-li ho ovládat). Spojení navážete příkazem:

```
$ ib111 beamer
```

Protože příkaz vytvoří nové sezení, nezapomeňte se přesunout do správné složky příkazem cd ~/ib111/NN.

T.3: Kostry úloh

Pracujete-li na studentském serveru aisa, můžete pro překlad jednotlivých příkladů použít přiložený soubor makefile, a to zadáním příkazu

```
$ make příklad
```

kde příklad je název souboru bez přípony (např. tedy make e1_factorial). Tento příkaz postupně:

XXX

Selže-li některý krok, další už se provádět nebude. Povede-li se překlad v prvním kroku, v pracovním adresáři naleznete spustitelný soubor s názvem příklad.bin, se kterým můžete dále pracovat.

Existující přeložené soubory můžete smazat příkazem make clean (vynutíte tak jejich opětovný překlad a spuštění všech kontrol).

T.3.1 Textový editor Na stroji aisa je k dispozici jednoduchý editor micro, který má podobné ovládání jako klasické textové editory, které pracují v grafickém režimu, a který má slušnou podporu pro práci se zdrojovým kódem. Doporučujeme zejména méně pokročilým. Další možností jsou samozřejmě pokročilé editory vim a emacs.

Mimo lokálně dostupné editory si můžete ve svém oblíbeném editoru, který máte nainstalovaný u sebe, nastavit režim vzdálené editace (použitím protokolu ssh). Minimálně ve VS Code je takový režim k dispozici a je uspokojivě funkční.

T.3.2 Vlastní prostředí XXX

[33] Jako alternativu, nechcete-li z nějakého důvodu WSL instalovat, lze použít program putty.

Část U: Doporučení k zápisu kódu

Tato sekce rozvádí obecné principy zápisu kódu s důrazem na čitelnost a korektnost. Samozřejmě žádná sada pravidel nemůže zaručit, že napíšete dobrý (korektní a čitelný) program, o nic více, než může zaručit, že napíšete dobrou povídku nebo namalujete dobrý obraz. Přesto ve všech těchto případech pravidla existují a jejich dodržování má obvykle na výsledek pozitivní dopad.

Každé pravidlo má samozřejmě nějaké výjimky. Tyto jsou ale výjimkami proto, že nastávají **výjimečně**. Některá pravidla připouští výjimky častěji než jiná:

1 **Dekompozice** Vůbec nejdůležitější úlohou programátora je rozdělit problém tak, aby byl schopen každou část správně vyřešit a dílčí výsledky pak poskládat do korektního celku.

A. Kód musí být rozdělen do ucelených jednotek (kde jednotkou rozumíme funkci, typ, modul, atd.) přiměřené velikosti, které lze studovat a používat nezávisle na sobě.
B. Jednotky musí být od sebe odděleny jasným **rozhraním**, které by mělo být jednodušší a uchopitelnější, než kdybychom použití jednotky nahradili její definicí.
C. Každá jednotka by měla mít **jeden** dobře definovaný účel, který je zachycený především v jejím pojmenování a případně rozvedený v komentáři.
D. Máte-li problém jednotku dobře pojmenovat, může to být známka toho, že dělá příliš mnoho věcí.
E. Jednotka by měla realizovat vhodnou **abstrakci**, tzn. měla by být **obecná** – zkuste si představit, že dostanete k řešení nějaký jiný (ale dostatečně příbuzný) problém: bude Vám tato konkrétní jednotka k něčemu dobrá, aniž byste ji museli (výrazně) upravovat?
F. Má-li jednotka parametr, který fakticky identifikuje místo ve kterém ji používáte (bez ohledu na to, je-li to z jeho názvu patrné), je to často známka špatně zvolené abstrakce. Máte-li parametr, který by bylo lze pojmenovat called_from_bar, je to jasná známka tohoto problému.
G. Daný podproblém by měl být vyřešen v programu pouze jednou – nedaří-li se Vám sjednotit různé varianty stejného nebo velmi podobného kódu (aniž byste se uchýlili k taktice z bodu F), může to být známka nesprávně zvolené dekompozice. Zkuste se zamyslet, není-li možné problém rozložit na podproblémy jinak.

2 **Jména** Dobře zvolená jména velmi ulehčují čtení kódu, ale jsou i dobrým vodítkem při dekompozici a výstavbě abstrakcí.

A. Všechny entity ve zdrojovém kódu nesou **anglická** jména. Angličtina je univerzální jazyk programátorů.
B. Jméno musí být **výstižné** a **popisné**: v místě použití je obvykle jméno náš hlavní (a často jediný) **zdroj informací** o jmenované entitě. Nutnost hledat deklaraci nebo definici (protože ze jména není jasné, co volaná funkce dělá, nebo jaký má použitá proměnná význam) čtenáře nesmírně zdržuje.[34]
C. Jména **lokálního** významu mohou být méně informativní: je mnohem větší šance, že význam jmenované entity si pamatujeme, protože byla definována před chvílí (např. lokální proměnná v krátké funkci).
D. Obecněji, informační obsah jména by měl být přímo úměrný jeho rozsahu platnosti a nepřímo úměrný frekvenci použití: globální jméno musí být informativní, protože jeho definice je „daleko“ (takže si ji už nepamatujeme) a zároveň se nepoužívá příliš často (takže si nepamatujeme ani to, co jsme se dozvěděli, když jsme ho potkali naposled).
E. Jméno parametru má dvojí funkci: krom toho, že ho používáme v těle funkce (kde se z pohledu pojmenování chová podobně jako lokální proměnná), slouží jako dokumentace funkce jako celku. Pro parametry volíme popisnější jména, než by zaručovalo jejich použití ve funkci samotné – mají totiž dodatečný globální význam.
F. Některé entity mají ustálené názvy – je rozumné se jich držet, protože čtenář automaticky rozumí jejich významu, i přes obvyklou stručnost. Zároveň je potřeba se vyvarovat použití takovýchto ustálených jmen pro nesouvisející entity. Typickým příkladem jsou iterační proměnné i a j.
G. Jména s velkým rozsahem platnosti by měla být také **zapamatovatelná**. Je vždy lepší si přímo vzpomenout na jméno funkce, kterou právě potřebuji, než ho vyhledávat (podobně jako je lepší znát slovo, než ho jít hledat ve slovníku).
H. Použitý slovní druh by měl odpovídat druhu entity, kterou pojmenovává. Proměnné a typy pojmenováváme přednostně podstatnými jmény, funkce přednostně slovesy.
I. Rodiny příbuzných nebo souvisejících entit pojmenováváme podle společného schématu:
   - table_name, table_size, table_items – nikoliv např. items_in_table;
   - list_parser, string_parser, set_parser;
   - find_min, find_max, erase_max – nikoliv např. erase_maximum nebo erase_greatest nebo max_remove.
J. Jména by měla brát do úvahy kontext, ve kterém jsou platná. Neopakujte typ proměnné v jejím názvu (cars, nikoliv list_of_cars ani set_of_cars) nemá-li tento typ speciální význam. Podobně jméno nadřazeného typu nepatří do jmen jeho metod (třída list by měla mít metodu length, nikoliv list_length).
K. Dávejte si pozor na překlepy a pravopisné chyby. Zbytečně znesnadňují pochopení a (zejména v kombinaci s našeptávačem) lehce vedou na skutečné chyby způsobené záměnou podobných ale jinak napsaných jmen. Navíc kód s překlepy v názvech působí značně neprofesionálně.

3 **Stav a data** Udržet si přehled o tom, co se v programu děje, jaké jsou vztahy mezi různými stavovými proměnnými, co může a co nemůže nastat, je jedna z nejtěžších částí programování.

TBD: Vstupní podmínky, invarianty, …

4 **Řízení toku** Přehledný, logický a co nejvíce lineární sled kroků nám ulehčuje pochopení algoritmu. Časté, komplikované větvení je naopak těžké sledovat a odvádí pozornost od pochopení důležitých myšlenek.

TBD.

5 **Volba algoritmů a datových struktur** TBD.

6 **Komentáře** Nejde-li myšlenku předat jinak, vysvětlíme ji doprovodným komentářem. Čím těžší myšlenka, tím větší je potřeba komentovat.

A. Podobně jako jména entit, komentáře které jsou součástí kódu píšeme anglicky.[35]
B. Případný komentář jednotky kódu by měl vysvětlit především „co“ a „proč“ (tzn. jaký plní tato jednotka účel a za jakých okolností ji lze použít).
C. Komentář by také neměl zbytečně duplikovat informace, které jsou k nalezení v hlavičce nebo jiné „nekomentářové“ části kódu – jestli máte například potřebu komentovat parametr funkce, zvažte, jestli by nešlo tento parametr lépe pojmenovat nebo otypovat.
D. Komentář by **neměl** zbytečně duplikovat samotný spustitelný kód (tzn. neměl by se zdlouhavě zabývat tím „jak“ jednotka vnitřně pracuje). Zejména jsou nevhodné komentáře typu „zvýšíme proměnnou i o jedna“ – komentář lze použít k vysvětlení **proč** je tato operace potřebná – co daná operace dělá si může každý přečíst v samotném kódu.

7 **Formální úprava** TBD.

[34] Nejde zde pouze o samotný fakt, že je potřeba něco vyhledat. Mohlo by se zdát, že tento problém řeší IDE, které nás umí „poslat“ na příslušnou definici samo. Hlavní zdržení ve skutečnosti spočívá v tom, že musíme přerušit čtení předchozího celku. Na rozdíl od počítače je pro člověka „zanořování“ a zejména pak „vynořování“ na pomyslném zásobníku docela drahou operací.

[35] Tato sbírka samotná představuje ústupek z tohoto pravidla: smyslem našich komentářů je naučit Vás poměrně těžké a často nové koncepty, a její cirkulace je omezená. Zkušenost z dřívějších let ukazuje, že pro studenty je anglický výklad značnou bariérou pochopení. Přesto se snažte vlastní kód komentovat anglicky – výjimku lze udělat pouze pro rozsáhlejší komentáře, které byste jinak nedokázali srozumitelně formulovat. V praxi je angličtina zcela běžně bezpodmínečně vyžadovaná.