PB111 Nízkoúrovňové programování

P. Ročkai & A. Matoušek

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 prvním 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 pb111 update. Všechny výše uvedené složky pak naleznete ve složce ~/pb111.

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 praktického testu (kolokvia). 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 (typicky se bude jednat o velmi malé programy v rozsahu jednotek až desítek řádků, kterých byste měli být v průměru schopni vyřešit několik za hodinu) 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.

bl.		téma
1	1.	abstraktní stroj
	2.	lokální proměnné, řízení toku
	3.	podprogramy
	4.	adresy, pole, struktury
2	5.	paměť
	6.	zřetězený seznam
	7.	dynamická paměť
	8.	recyklace paměti
3	9.	dynamické pole
	10.	binární halda
	11.	hašovací tabulka
	12.	vyhledávací strom

A.1.2 Organizace sbírky V následujících sekcích naleznete detailnější informace a **závazná** pravidla kurzu: doporučujeme Vám, abyste se s nimi důkladně seznámili.[3] Zbytek sbírky je pak rozdělen na části, které odpovídají jednotlivým týdnům semestru. **Důležité:** během prvního týdne semestru už budete řešit přípravy z první kapitoly, přestože první cvičení je ve až v týdnu druhém. Nulté cvičení je volitelné a není nijak hodnoceno.

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

[1] Studijní materiály budeme tento semestr doplňovat průběžně, protože kurz v tomto semestru teprve vzniká. 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 úterý odpovídajícího týdne (den přednášky), sady podobně půlnocí na první pondělí odpovídajícího bloku. Pro první týden tedy 19.2.2023 23:59 a první sadu 26.2.2023 0:00.

[2] https://is.muni.cz/auth/el/fi/jaro2024/PB111/um/

[3] Pravidla jsou velmi podobná těm v kurzu IB111, ale přesto si je pozorně přečtěte.

A.1.3 Plán semestru Tento kurz vyžaduje značnou aktivitu během semestru. V této sekci naleznete přehled důležitých událostí formou kalendáře. Jednotlivé 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,
- „sX/z_1“ – první kolo známek za kvalitu kódu sady X,
- „sX/op“ – termín pro opravná odevzdání sady X,
- „sX/z_2“ – finální známky za kvalitu kódu sady X,
- „test“ – termín programovacího testu.

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

únor

	Po	Út	St	Čt	Pá	So	Ne
#1	19	20	21	22	23	24	25
cv 0				01/v		**01/p**	
#2	26	27	28	29			
cv 1	s1/1		s1/2	02/v			

březen

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

duben

	Po	Út	St	Čt	Pá	So	Ne
#7	1	2	3	4	5	6	7
cv 6	s2/4	$s1/z_1$	s2/5	07/v	s2/6	**07/p**	
#8	8	9	10	11	12	13	14
cv 7	s2/7	s1/op	s2/8	08/v	s2/9	**08/p**	
#9	15	16	17	18	19	20	21
cv 8	s2/10	$s1/z_2$	s2/11	09/v	**s2/12**	**09/p**	
#10	22	23	24	25	26	27	28
cv 9	s3/1		s3/2	10/v	s3/3	**10/p**	
#11	29	30					
cv10	s3/4	$s2/z_1$					

květen

	Po	Út	St	Čt	Pá	So	Ne
#11			1 sv	2	3	4	5
			s3/5	11/v	s3/6	**11/p**	
#12	6	7	8 sv	9	10	11	12
cv11	s3/7	s2/op	s3/8	12/v	s3/9	**12/p**	
#13	13	14	15	16	17	18	19
cv12	s3/10	$s2/z_2$	s3/11		**s3/12**		
	20	21	22	23	24	25	26
	27	28	29	30	31		
		$s3/z_1$					

červen

	Po	Út	St	Čt	Pá	So	Ne
						1	2
	3	4	5	6	7	8	9
		s3/op					
	10	11	12	13	14	15	16
		$s3/z_2$		test			
	17	18	19	20	21	22	23
				test			
	24	25	26	27	28	29	30
				test			

A.2: Hodnocení

Abyste předmět úspěšně ukončili, musíte v **každém bloku**[4] získat **60 bodů**. Žádné další požadavky nemáme.

Výsledná známka závisí na celkovém součtu bodů (splníte-li potřebných 4×60 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.[5]

známka	předběžné minimum	po vyhodnocení semestru
A	420	90. percentil + 75
B	360	65. percentil + 75
C	310	35. percentil + 75
D	270	10. percentil + 75
E	240	240

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 **0,5** bodu (max. 3 body každý týden, nebo **12/blok**),
- za účast[6] 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 **60** bodů. Dále můžete získat:

- **10** bodů za úspěšně vyřešený příklad ze sady domácích úloh (celkem vždy **60/blok**).

V blocích 2-4 navíc můžete získat body za kvalitu řešení příkladů ze sady

[4] Máte-li předmět ukončen zápočtem, čtvrtý blok a tedy ani závěrečný test pro Vás není relevantní. Platí požadavek na 3×60 bodů z bloků v semestru.

[5] 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 3×60 bodů).

[6] 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 max. 1–2× za semestr.

úloh předchozího bloku:

- za kvalitu kódu max. **5** bodů za příklad (celkem **30/blok**).

Konečně blok 4, který patří do zkouškového období, nemá ani cvičení ani sadu domácích úloh. Krom bodů za kvalitu kódu ze třetí sady lze získat:

- **15** bodů za každý zkouškový příklad (celkem **90/blok**).

Celkově tedy potřebujete:

- blok 1: **60/120** bodů,
- blok 2: **60/150** bodů,
- blok 3: **60/150** bodů,
- blok 4: **60/120** bodů (neplatí pro ukončení zápočtem).

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 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 dalšího 0,5 bodu (tzn. celkem 1,5 bodu). Výsledky testů naleznete v **poznámkovém bloku** v informačním systému.

Příklady můžete odevzdávat:

1. do **odevzdávárny** s názvem NN v ISu (např. 01),
2. příkazem pb111 submit ve složce ~/pb111/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),
- dále podobným způsobem vybere vzájemné (peer) recenze, které jste v předchozím týdnu psali, a stručně je s Vámi prodiskutuje (celkovou strukturu recenze, proč je který komentář dobrý nebo nikoliv, jestli nějaký komentář chybí, atp.) – opět se můžete (resp. byste se měli) zapojovat,
- 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 jeden příklad ze sbírky, bez možnosti hledat na internetu – tak, jak to bude na závěrečném testu; každé úspěšné řešení (tzn. takové, které splní verity testy) získá ony 3 body za aktivitu pro daný týden.

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 6 domácích úloh, které tvoří významnou část hodnocení předmětu. 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 pb111 submit sN_úkol ve složce ~/pb111/sN, např. pb111 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 a kontroly paměťových chyb.

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 pb111 status.

A.5.3 **Bodování** Za každý domácí úkol, ve kterém Vaše odevzdání v příslušném termínu splní testy „verity“, získáte 10 bodů.

Za stejný úkol máte dále možnost získat body za kvalitu kódu, a to vždy v hodnotě max. 5 bodů. Body za kvalitu se počítají v bloku, **ve kterém byly uděleny**, tzn. body za kvalitu ze **sady 1** se započtou do **bloku 2**.

Maximální bodový zisk za jednotlivé sady:

- sada 1: 60 za funkčnost v bloku 1 + 30 za kvalitu v bloku 2,
- sada 2: 60 za funkčnost v bloku 2 + 30 za kvalitu v bloku 3,
- sada 3: 60 za funkčnost v bloku 3 + 30 za kvalitu v bloku 4 (**zkouškovém**).

A.5.4 **Hodnocení kvality kódu** Automatické testy ověřují **správnost** vašich programů (do takové míry, jak je to praktické – ani nejpřísnější testy nemůžou zaručit, že máte program zcela správně). Správnost ale není jediné kritérium, podle kterého lze programy hodnotit: podobně důležité je, aby byl program **čitelný**. Programy totiž mimo jiné slouží ke komunikaci myšlenek lidem – dobře napsaný a správně okomentovaný kód by měl čtenáři sdělit, jaký řeší problém, jak toto řešení funguje a u obojího objasnit **proč**.

Je Vám asi jasné, že čitelnost programu člověkem může hodnotit pouze člověk: proto si každý Váš **úspěšně** vyřešený domácí úkol přečte opravující a své postřehy Vám sdělí. Přitom zároveň Váš kód oznámkuje podle kritérií podrobněji rozepsaných v kapitole Z. Tato kritéria aplikujeme při známkování takto:

- hodnocení A dostane takové řešení, které jasně popisuje řešení zadaného problému, je správně dekomponované na podproblémy, je zapsáno bez zbytečného opakování, a používá správné abstrakce, algoritmy a datové struktury,
- hodnocení B dostane program, který má výrazné nedostatky v jedné, nebo nezanedbatelné nedostatky ve dvou oblastech výše zmíněných, například:
  - je relativně dobře dekomponovaný a zbytečně se neopakuje, ale používá nevhodný algoritmus nebo datovou strukturu a není zapsán příliš přehledně,
  - používá optimální algoritmus a datové struktury a je dobře dekomponovaný, ale lokálně opakuje tentýž kód s drobnými obměnami, a občas používá zavádějící nebo jinak nevhodná jména podprogramů, proměnných atp.,
  - jinak dobrý program, který používá zcela nevhodný algoritmus, **nebo** velmi špatně pojmenované proměnné, **nebo** je zapsaný na dvě obrazovky úplně bez dekompozice,
- hodnocení X dostanou programy, u kterých jste se dobrovolně vzdali hodnocení (a to jasně formulovaným komentářem **na začátku souboru**, např. „Vzdávám se hodnocení.“),
- hodnocení C dostanou všechny ostatní programy, zejména ty, které kombinují dvě a více výrazné chyby zmiňované výše.

Známky Vám budou zapsány druhé úterý následujícího bloku. Dostanete-li známku B nebo C, budete mít možnost svoje řešení ještě zlepšit, odevzdat znovu, a známku si tak opravit:

- na opravu budete mít týden,
- na opraveném programu nesmí selhat verity testy,
- testy budou nadále probíhat se stejnou kadencí jako během řádné doby k vypracování (pondělí, středa, pátek o 23:59).

Bude-li opravující s vylepšeným programem spokojen, výslednou známku Vám upraví.

Termíny, které se vážou k hodnocení kvality, jsou vždy v úterý a jsou vyznačené v přehledovém kalendáři v části A.1 takto:

- „s1/z_1“ – obdržíte známky za první sadu,
- „s1/op“ – termín pro odevzdání opravených řešení 1. sady,
- „s1/z_2“ – výsledné známky za první sadu,
- analogicky pro s2 a s3.

Jednotlivé **výsledné** známky se promítnou do bodového hodnocení úkolu následovně:

- známka **A** Vám vynese **5 bodů**,
- známka **B** pak **2 body**,
- známka **X** žádné body neskýtá,
- známka **C** je hodnocena **-1 bodem**.

Samotné body za funkcionalitu se při opravě kvality již nijak nemění.

A.5.5 **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: Vzájemné recenze

Jednou z možností, jak získat body za aktivitu, jsou vzájemné (peer) recenze. Smyslem této aktivity je získat praxi ve čtení a hodnocení cizího kódu. Možnost psát tyto recenze se váže na vlastní úspěšné vypracování téhož příkladu.

Příklad: odevzdáte-li ve druhém týdnu 4 přípravy, z toho u třech splníte testy „verity“ (řekněme p1, p2, p5), ve třetím týdnu dostanete po jednom řešení těchto příkladů (tzn. budete mít možnost recenzovat po jedné instanci 02/p1, 02/p2 a 02/p5). Termín pro odevzdání recenzí na přípravy z druhé kapitoly je shodný s termínem pro odevzdání příprav třetí kapitoly (tzn. sobotní půlnoc).

Vypracování těchto recenzí je dobrovolné. Za každou vypracovanou recenzi získáte jeden bod za aktivitu, počítaný v týdnu, kdy jste recenze psali (v uvedeném příkladu by to tedy bylo ve třetím týdnu semestru, tedy do stejné „kolonky“ jako body za příklady 02/r).

Udělení bodů je podmíněno smysluplným obsahem – **nestačí** napsat „nemám co dodat“ nebo „není zde co komentovat“. Je-li řešení dobré, napište **proč** je dobré (viz též níže). Vámi odevzdané recenze si přečte Váš cvičící a některé z nich může vybrat k diskusi ve cvičení (v dalším týdnu), v podobném duchu jako přípravy samotné.

Pozor, v jednom týdnu lze získat maximálně **3 body** za aktivitu, bez ohledu na jejich zdroj (recenze, vypracování příkladu u tabule, atp.). Toto omezení není dotčeno ani v případě, kdy dostanete k vypracování více než 3 příklady (můžete si ale vybrat, které z nich chcete recenzovat).

A.6.1 **Jak recenze psát** Jak recenze vyzvednout a odevzdat je blíže popsáno v kapitole T. Své komentáře vkládejte přímo do vyzvednutých zdrojových souborů. Komentáře můžete psát česky (slovensky) nebo anglicky, volba je na Vás. Komentáře by měly být stručné, ale užitečné – Vaším hlavním cílem by mělo být pomoct adresátovi naučit se lépe programovat.

Snažte se aplikovat kritéria a doporučení z předchozí sekce (nejlépe na ně přímo odkázat, např. „tuto proměnnou by šlo jistě pojmenovat lépe (viz doporučení 2.b)“). Nebojte se ani vyzvednout pozitiva (můžete zde také odkázat doporučení, máte-li například za to, že je obzvlášť pěkně uplatněné) nebo poznamenat, když jste se při čtení kódu sami něco naučili.

Komentáře vkládejte vždy **před** komentovaný celek, a držte se podle možnosti tohoto vzoru (použití ** pomáhá odlišit původní komentáře autora od poznámek recenzenta):

```
/** A short, one-line remark. **/
```

U víceřádkových komentářů:

```
/** A longer comment, which should be wrapped to 80 columns or
 ** less, and where each line should start with the ** marker.
 ** It is okay to end the comment on the last line of text like
 ** this. **/
```

Při vkládání komentářů **neměňte** existující řádky (zejména se ujistěte, že máte vypnuté automatické formátování, editujete-li zdrojový kód v nějakém IDE). Jediné povolená operace jsou:

- vložení nových řádků (prázdných nebo s komentářem), nebo
- doplnění komentáře na stávající **prázdný** řádek.

A.7: Závěrečný programovací test

Zkouškové období tvoří pomyslný 4. blok a platí zde stejné kritérium jako pro všechny ostatní bloky: musíte získat alespoň 60 bodů. Závěrečný test:

- proběhne v počítačové učebně bez přístupu k internetu nebo vlastním materiálům,
- k dispozici bude tato sbírka (bez vzorových řešení příkladů typu e a r) a skripta,
- budete moct používat textový editor nebo vývojové prostředí VS Code, standardní překladače jazyka C a odpovídající nástroje, překladač tiny a odpovídající virtuální stroj.

Na vypracování praktické části budete mít 4 hodiny čistého času, a bude sestávat ze šesti příkladů, které budou hodnoceny automatickými testy, s maximálním ziskem 90 bodů. Příklady jsou hodnoceny binárně (tzn. příklad je uznán za plný počet bodů, nebo uznán není). Kvalita kódu hodnocena nebude. 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 4 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).

Zkouška proběhne až po vyhodnocení recenzí za třetí blok (tzn. ve druhé polovině zkouškového období). Plánované termíny[7] jsou:

- čtvrtek 13.6. 9:00–13:00, 14:00–18:00,
- čtvrtek 20.6. 9:00–13:00, 14:00–18:00,
- čtvrtek 27.6. 9:00–13:00, 14:00–18:00.

A.7.1 **Vnitrosemestrálky** V posledním týdnu každého bloku, tedy

- cvičení 4 (18.-22. března),
- cvičení 8 (15.-19. dubna),
- cvičení 12 (13.-17. května),

proběhne v rámci cvičení programovací test na 40 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 jeden příklad, za který můžete získat 3 body, které se počítají jako body za aktivitu v tomto cvičení.

A.8: Opisování

Na všech zadaných problémech pracujte prosím zcela samostatně – 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, posloucháním 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.

[7] Může se stát, že termíny budeme z technických nebo organizačních důvodů posunout na jiný den nebo hodinu. V takovém případě Vám samozřejmě změnu s dostatečným předstihem oznámíme.

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),
- 10 bodů z hodnocení bloku, do kterého opsaný příklad patří,
- 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 4,5 bodu,
- Váš celkový zisk za první blok je 65 bodů,

jste **automaticky hodnoceni známkou X** (65 - 2,25 - 10 je méně než potřebných 60 bodů). Podobně s příkladem z první sady (65 - 5 - 10), atd. Máte-li v bloku bodů dostatek (např. 80 - 5 - 10 > 60), 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 77, 62, 61, 64,
- v prvním bloku jste opsali domácí úkol,

budete penalizováni:

- v prvním bloku 10 + 5, tzn. bodové zisky za bloky budou efektivně 62, 62, 61, 64,
- v celkovém hodnocení 10, tzn. celkový zisk 62 + 62 + 61 + 64 - 10 = 239, 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 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: Úvod

Účelem tohoto kurzu je seznámit Vás s tím, jak probíhá výpočet na úrovni procesoru, a jaký je vztah mezi tímto nízkoúrovňovým výpočetním modelem a tzv. jazyky vyšší úrovně. Abychom mohli tento vztah zkoumat, musíme porozumět jak

1. onomu vyššímu jazyku (v tomto kurzu to bude ten nejnižší z nich – jazyk C – abychom co nejvíce zmenšili vzdálenost, kterou musíme překlenout) tak
2. výpočetnímu stroji (který odpovídá procesoru a paměti), který bude výpočty našich programů realizovat.

Z předchozího již znáte jiný vyšší programovací jazyk, Python – ten použijeme jako odrazový bod. Potřebovat budete samozřejmě pouze ty části jazyka, které znáte z kurzu IB111.

B.1: Programovací jazyk

Podobně jako v kurzu IB111, budeme používat omezenou podmnožinu „skutečného“ (prakticky běžně používaného) programovacího jazyka – v tomto kurzu to bude podmnožina jazyka C. Každá kapitola, počínaje tou druhou, bude obsahovat popis všech prvků jazyka C, které musíte v dané kapitole zvládnout.[8]

V nulté a první kapitole se budeme zabývat pouze strojovým kódem a jazykem symbolických adres – budeme tedy programovat přímo výpočetní stroj, se zcela minimální abstrakcí. (Omezený) jazyk C se objeví teprve ve 2. kapitole.

B.2: Výpočetní stroj

Stav výpočetního stroje, se kterým budeme v tomto předmětu pracovat, je velmi jednoduchý. Skládá se z:

1. šestnácti registrů, každý o šířce 16 bitů:
   - registr rv (return value),
   - registry l1 až l7 (local),
   - registry t1 až t6 (temporary),
   - registry bp a sp,
2. speciálního 16bitového registru pc (program counter),
3. 64 KiB paměti adresované po slabikách (bajtech) – adresa je tedy 16bitové celé číslo (bez znaménka), které přesně určuje právě jednu paměťovou buňku, přitom každá taková buňka obsahuje celé číslo v rozsahu 0 až 255.

Sémanticky speciální jsou pouze registry pc a sp – všechny ostatní jsou z pohledu stroje ekvivalentní a jejich jména nemají pro samotný výpočet žádný speciální význam – jedná se pouze o konvenci, která nám usnadní čtení (a psaní) programů.

Výpočet stroje probíhá takto:

1. z adresy uložené v registru pc se načtou dvě šestnáctibitová slova – hi z adresy pc a lo z adresy pc + 2 – která kódují jednu instrukci,
2. instrukce je strojem dekódována a provedena:
   - slovo hi kóduje operaci (vyšší slabika), cílový registr a první registrový operand,
   - slovo lo kóduje přímý (immediate) operand, nebo druhý registrový operand (v nejvyšší půlslabice),
   - provede se efekt instrukce (tento efekt samozřejmě závisí jak na operaci, tak na operandech) – obvykle je součástí tohoto efektu změna hodnoty uložené v registru pc,
3. nebyl-li výpočet zastaven, pokračuje bodem 1.

Registry jsou očíslovány v pořadí uvedeném výše, totiž rv je registr číslo 0 a sp je registr číslo 15. Je vidět, že číslo registru lze zakódovat do jedné půlslabiky (registr pc operandem být nemůže).

Následuje výčet všech operací, které umí stroj provést. Nebudeme všechny operace potřebovat hned, a nebudeme se tedy zatím ani podrobněji zabývat jejich sémantikou – tu si rozebereme vždy na začátku kapitoly, v níž začnou být tyto operace relevantní.

1. speciální operace:
   - práce se zásobníkem (push, pop),
   - nastavení registru na konstantu (put),
   - nastavení registru na hodnotu z jiného registru (copy),
   - znaménkové rozšíření bajtu (sext),
2. operace pro práci s pamětí:
   - kopírování dat z paměti do registru (ld, ldb),
   - kopírování z registru do paměti (st, stb),
3. aritmetické operace:
   - aditivní – bez rozlišení znaménkovosti (add, sub),
   - násobení mul,
   - dělení se znaménkem (sdiv, smod),
   - dělení bez znaménka (udiv a umod),
4. operace pro srovnání dvou hodnot:
   - rovnost (eq, ne),
   - znaménkové ostré nerovnosti (slt, sgt),
   - znaménkové neostré nerovnosti (sle, sge),
   - bezznaménkové ostré (ult, ugt), a konečně
   - bezznaménkové neostré (ule, uge),
5. bitové operace:
   - logické operace and, or a xor aplikované po bitech,
   - bitové posuvy shl (levý), shr (pravý) a aritmetický sar,
6. řízení toku:
   - nepodmíněný skok jmp,
   - podmíněné skoky jz (jump if zero) a jnz (if not zero),
   - volání a návrat z podprogramu (call, ret),
7. ovládaní stroje:
   - halt zastaví výpočet,
   - asrt zastaví výpočet s chybou, je-li operand nulový.

B.3: Jazyk symbolických adres

Stroj jako takový pracuje pouze s **číselnými** adresami – instrukce, která obsahuje adresu, ji vždy obsahuje jako číslo. To při programování představuje značný problém, protože adresy jednotlivých částí programu závisí na tom, kolik instrukcí se nachází v části předchozí. Uvažme třeba tento program (uložený v paměti od adresy nula):

```
put  0     → rv ; vynuluj registr rv
add  1, rv → rv ; do registru rv přičti 1
jnz  rv, 0x0004 ; je-li rv nenulové, skoč na adresu 4
```

Protože každá instrukce je kódována do 4 bajtů, adresa druhé instrukce (operace add) je 4 (její kódování je uloženo na adresách 4, 5, 6 a 7). Program jak je napsaný provede prázdný cyklus 65535× (v poslední iteraci je v registru rv hodnota ffff, přičtením jedničky se změní na nulu, podmíněný skok „není-li rv nula“ se neprovede a cyklus tak skončí).

Uvažme nyní situaci, kdy do programu potřebujeme (na začátek) zařadit další instrukci, např. nastavení registru l1:

```
put  0     → l1 ; vynuluj registr l1
put  0     → rv ; vynuluj registr rv
add  1, rv → rv ; do registru rv přičti 1
jnz  rv, 0x0004 ; je-li rv nenulové, skoč na adresu 4
```

Tím se ale posunuly všechny další instrukce v programu na jiné adresy – proto adresa skoku předaná operaci jnz neodpovídá původnímu programu – tento nový program bude cyklit donekonečna (rozmyslete si proč).

[8] Žádné jiné v kurzu nebudou k dispozici, máte tedy zaručeno, že dokážete přečíst každý program, který k dané kapitole patří – ať už z přednášky, z této sbírky, nebo kód svých spolužáků, který uvidíte ve cvičení.

Je asi zřejmé, že kdyby měla každá změna programu (přidání nebo odebrání instrukce) znamenat, že musíme opravit všechny adresy ve všech ostatních instrukcích, moc dobře by se nám neprogramovalo. Proto pro zápis strojového kódu používáme tzv. jazyk **symbolických adres**. Ten nám umožňuje místa v programu – adresy – pojmenovat **symbolem** – textovým názvem, podobně jako nazýváme třeba proměnné v jazyce Python. Symbol zavedeme tzv. **návěstím** a použijeme v zápisu instrukce[9] na místě adresy:

```
    put  0     → rv ; vynuluj registr rv
loop:               ; návěstí pro první instrukci cyklu
    add  1, rv → rv ; do registru rv přičti 1
    jnz  rv, loop   ; je-li rv nenulové, skoč na začátek cyklu
```

Když nyní přidáme na začátek programu instrukci, nic špatného se nestane – při sestavení (angl. **assembly**) programu se pak do podmíněného skoku místo adresy 4 doplní adresa 8 – totiž adresa instrukce, která bezprostředně následuje za návěstím.

B.d: Demonstrace (ukázky)

B.d.1 [tinyvm] Tento program implementuje kompletní sémantiku výpočetního stroje, který budeme v tomto kurzu používat. Je naprogramován v jazyce z 11. týdne kurzu IB111, měli byste tedy samotnému zápisu programu bez problémů rozumět. V komentářích je pak vysvětlena sémantika (jak stroj pracuje).

Protože se jedná o spustitelný program, popisuje sémantiku výpočetního stroje velmi přesně – můžete jej tedy použít jako referenční příručku strojového kódu, který budeme používat.

Doporučujeme Vám program si pozorně přečíst už nyní, na začátku semestru, ale je zcela v pořádku, pokud neporozumíte ihned všemu. Očekáváme, že se budete k programu minimálně několik následujících týdnů pravidelně vracet. Studium sémantiky nových operací na začátku několika příštích kapitol je k tomu ideální příležitostí.

Jádro celého stroje tvoří procedura step, která načte, dekóduje a provede jednu instrukci. Vstupními parametry jsou:

1. pc je aktuální hodnota programového čítače,
2. regs je seznam 16 celých čísel, každé v rozsahu 0 až 65535, jenž reprezentují hodnoty uložené v registrech,
3. mem je seznam 65536 celých čísel, každé v rozsahu 0 až 255, přitom číslo uložené na indexu i reprezentuje paměťovou buňku s adresou i.

[9] Striktně vzato se v takové chvíli nejedná o zápis instrukce, pouze o předpis, jak konkrétní instrukci dopočítat – protože je to ale výpočet velmi jednoduchý, nebudeme obvykle tyto případy rozlišovat (tzn. návěstí budeme přímo interpretovat jako adresu, kterou reprezentuje v daném programu).

Návratovou hodnotou je dvojice celých čísel (nová hodnota programového čítače, příznak má-li výpočet pokračovat).

```
def step( pc: int, regs: list[ int ],
          mem: list[ int ] ) -> tuple[ int, int ]:
```

Následující tvrzení popisují základní vstupní podmínky.

```
    assert 0 <= pc < 65536
    assert len( regs ) == 16
    assert len( mem ) == 65536
```

Abychom mohli instrukci co nejsnadněji provést, dekódujeme ji na několik pojmenovaných hodnot. Hodnota opcode je číslo operace, složené ze dvou půlslabik – kategorie cat a konkrétní operace z dané kategorie op. V šestnáctkovém zápisu tedy opcode = 0x12 značí operaci 2 z kategorie 1.

```
    opcode  = mem[pc]
    cat     = opcode // 16
    op      = opcode % 16
```

Druhá slabika popisuje vstupní a výstupní registr – tyto se uplatní u většiny operací (výjimky tvoří zejména operace z kategorie 0 – speciální instrukce, a kategorie 15 – řízení toku). Je-li šestnáctkový zápis druhé slabiky 0x82, je výstupním registrem ten s číslem 8 (t1) a (prvním) vstupním registrem je registr číslo 2, totiž 12.

```
    r_out   = mem[ pc + 1 ] // 16
    r_1     = mem[ pc + 1 ] %  16
```

Třetí a čtvrtá slabika pak popisují buď tzv. přímý (immediate) operand (číselnou hodnotu, která je přímo součástí instrukce) nebo druhý vstupní registr (pro binární operace nad registry, např. ty dobře známé aritmetické). Nemá-li instrukce přímý operand, je poslední slabika nevyužitá.

```
    imm     = mem[ pc + 3 ] + mem[ pc + 2 ] * 256
    r_2     = mem[ pc + 2 ] // 16
    addr    = imm
```

Než instrukci vykonáme, vypíšeme dekódovanou instrukci a aktuální stav stroje – procedura print_state nemá na výpočet stroje žádný vliv, není tedy nutné ji blíže zkoumat. Můžete si ji ale přizpůsobit dle vlastního vkusu nebo potřeby.

```
    print_state( pc, regs, cat, op, imm, r_out, r_1, r_2 )
```

Nyní již následuje samotné vykonání instrukce. První dvě operace jsou z kategorie 14 – řízení stroje. Instrukce asrt ukončí výpočet s chybou, je-li ve vstupním registru hodnota nula, jinak pokračuje ve výpočtu další instrukcí. Instrukce halt výpočet zastaví vždy (nikoliv ale chybou).

```
    if opcode == 0xee and regs[ r_1 ] == 0: # asrt
        return pc, ERROR
    if opcode == 0xef: # halt
        return pc, HALT
```

Následují speciální operace z kategorie 0. Operace copy uloží do výstupního registru hodnotu registru vstupního, operace put uloží přímý operand do výstupního registru a konečně sext provede znaménkové rozšíření spodní slabiky vstupního registru a výsledek uloží do toho výstupního.

```
    if opcode == 0x0c: # copy
        regs[ r_out ] = regs[ r_1 ]
    if opcode == 0x0d: # put
        regs[ r_out ] = imm
    if opcode == 0x0e: # sext
        regs[ r_out ] = as_signed( regs[ r_1 ], 8 ) % 65536
```

Dále do kategorie 0 patří operace pro obecnou práci s pamětí.

Operace st (store):

- uloží slovo ze vstupního registru
- na adresu, která vznikne jako součet přímého operandu a hodnoty ve **výstupním** registru (jedná se zde o výjimečné použití výstupního registru jako vstupní hodnoty).

Varianta stb zapíše pouze spodní slabiku vstupního registru a přepíše tedy jedinou buňku paměti.

```
    if opcode in [ 0x03, 0x04 ]: # st, stb
        addr = ( addr + regs[ r_out ] ) % 65536

    if opcode == 0x03: # stb
        mem[ addr ] = regs[ r_1 ]  % 256

    if opcode == 0x04: # st
        mem[ addr + 0 ] = regs[ r_1 ] // 256
        mem[ addr + 1 ] = regs[ r_1 ]  % 256
```

Operace ld analogicky:

- vypočte adresu jako součet přímého operandu a hodnoty ve **vstupním** registru,
- z vypočtené adresy načte slovo a uloží ho do **výstupního** registru.

Varianta ldb přečte z paměti pouze jednu slabiku a na celé slovo ji doplní levostrannými nulami.

```
    if opcode in [ 0x01, 0x02 ]: # ld, ldb
        addr = ( addr + regs[ r_1 ] ) % 65536

    if opcode == 0x01: # ldb
        regs[ r_out ] = mem[ addr ]

    if opcode == 0x02: # ld
```

```
        regs[ r_out ] = mem[ addr ] * 256 + mem[ addr + 1 ]
```

Konečně jsou v kategorii 0 operace push a pop pro práci se zásobníkem. Jejich efekt je implementován pomocnými procedurami push a pop níže, protože stejný efekt budeme potřebovat i při implementaci některých dalších operací.

Operace push sníží hodnotu uloženou v registru sp o dvě a na výslednou adresu uloží slovo ze vstupního registru.

Operace pop analogicky nejprve přečte slovo uložené na adrese dané registrem sp, uloží ho do výstupního registru a konečně hodnotu registru sp o dvě zvýší.

```
    if opcode == 0x0a: # push
        push( regs, mem, regs[ r_1 ] )
    if opcode == 0x0b: # pop
        regs[ r_out ] = pop( regs, mem )
```

Tím je kategorie 0 vyřešena. Dále pokračujeme kategorií 15, která obsahuje operace pro řízení toku. Operace call uloží hodnotu programového čítače na zásobník (podobně jako operace push) – jedná se o tzv. návratovou adresu. Dále nastaví pc na hodnotu přímého operandu, čím předá řízení podprogramu na této adrese uloženému.

```
    if opcode == 0xfe: # call
        push( regs, mem, pc )
        return imm, CONT
```

Operace ret ukončí vykonávání podprogramu a řízení vrátí volajícímu – návratovou adresu načte ze zásobníku podobně jako operace pop. Tuto adresu nezapomeneme zvýšit, protože adresa na uložená zásobníku ukazuje na instrukci call, která volání způsobila.

```
    if opcode == 0xff: # ret
        return pop( regs, mem ) + 4, CONT
```

Konečně operace skoků – nepodmíněné jmp a podmíněné jz a jnz – pouze nastaví programový čítač na hodnotu přímého operandu. Podmíněný skok se provede v případě, že je hodnota vstupního registru nulová (jz) nebo naopak nenulová (jnz). Není-li podmínka splněna, tyto operace nemají žádný efekt a výpočet pokračuje další instrukcí.

```
    if cat == 0xf and ( op == 0 or # jmp
                        op == 1 and regs[ r_1 ] == 0 or # jz
                        op == 2 and regs[ r_1 ] != 0 ): # jnz
        return imm, CONT
```

Kategorie 1 až 3 obsahují binární aritmetické operace v několika variantách:

- operace z kategorie 1 použije přímý operand jako levý a vstupní registr jako pravý,
- v kategorii 2 je tomu naopak, levý operand je vstupní registr a pravý operand je přímý,
- konečně kategorie 3 pracuje se dvěma vstupními registry (přímý operand nemá).

Implementace aritmetických operací naleznete v čisté funkci arith definované níže.

```
    if cat == 1:
        regs[ r_out ] = arith( op, imm, regs[ r_1 ] )
    if cat == 2:
        regs[ r_out ] = arith( op, regs[ r_1 ], imm )
    if cat == 3:
        regs[ r_out ] = arith( op, regs[ r_1 ], regs[ r_2 ] )
```

Konečně kategorie 10 a 11 provádí aritmetické srovnání dvou hodnot – buď ve variantě se dvěma registry, nebo srovnání vstupního registru s přímým operandem.

```
    if cat == 0xa:
        regs[ r_out ] = compare( op, regs[ r_1 ], regs[ r_2 ] )
    if cat == 0xb:
        regs[ r_out ] = compare( op, regs[ r_1 ], imm )
```

Tím je implementace kompletní. S výjimkou několika málo operací pokračuje výpočet další instrukcí, tzn. té, která je uložena na adrese o 4 vyšší, než byla ta aktuální (každá instrukce je kódována čtyřmi slabikami).

```
    return pc + 4, CONT
```

Následující dva podprogramy realizují operace se zásobníkem – adresa vrcholu zásobníku je uložena v registru sp (registr číslo 15).

```
def push( regs: list[ int ], mem: list[ int ], val: int ) -> None:
    regs[ 15 ] = ( regs[ 15 ] - 2 ) % 65536
    mem[ regs[ 15 ] ] = val

def pop( regs: list[ int ], mem: list[ int ] ) -> int:
    rv = mem[ regs[ 15 ] ]
    regs[ 15 ] = ( regs[ 15 ] + 2 ) % 65536
    return rv
```

Čistá funkce as_signed bijektivně zobrazí celé číslo n z rozsahu $\langle 0, 2^b)$ na číslo v rozsahu $\langle -2^{b-1}, 2^{b-1})$, metodou známou jako dvojkový doplňkový kód. Opačné zobrazení lze v Pythonu provést velmi jednoduše, jako `m % 2 ** b`. Protože stejný výraz popisuje zkrácení výsledku, které se používá při bezznaménkové aritmetice s pevnou šířkou slova, budeme jej níže zapisovat přímo, bez použití pomocné funkce. Zobrazení realizované funkcí as_signed budeme níže značit $t(n)$.

```
def as_signed( num: int, bits: int ) -> int:
    mod = 2 ** bits
    num = num % mod
    return num if num < mod // 2 else num - mod
```

Čistá funkce arith realizuje základní aritmetické a logické operace – sčítání, odečítání, násobení a dělení se zbytkem, bitové logické operace a bitové posuvy. Vstupem jsou dvě celá čísla op_1 a op_2 v rozsahu $\langle 0, 2^{16})$, výsledek je v témže rozsahu.

```
def arith( op: int, op_1: int, op_2: int ) -> int:
```

Užitečnou vlastností dvojkového doplňkového kódu je, že operace sčítání (odečítání) a násobení se provádí zcela stejně, jako jejich bezznaménkové verze – platí:

$$t(a) + t(b) = t(a + b)$$
$$t(a) - t(b) = t(a - b)$$
$$t(a) \cdot t(b) = t(a \cdot b)$$

Pro dělení podobná rovnost žel neplatí, proto musíme rozlišovat operace sdiv/udiv a smod/umod.

```
    if op == 0x1: return ( op_1 + op_2 ) % 65536
    if op == 0x2: return ( op_1 - op_2 ) % 65536
    if op == 0x3: return ( op_1 * op_2 ) % 65536
    if op == 0x4: return op_1 // op_2
    if op == 0x6: return op_1 %  op_2
```

Pro jednoduchost budeme při dělení dodržovat znaménkovou konvenci, která se používá v jazyce C, a která je žel odlišná od té, která se používá v jazyce Python.

```
    if op == 0x5 or op == 0x7: # signed div/rem
        dividend = as_signed( op_1, 16 )
        divisor = as_signed( op_2, 16 )
        quot, rem = divmod( dividend, divisor )

        if ( dividend > 0 ) != ( divisor > 0 ) and rem != 0:
            rem -= divisor
        if quot < 0 and rem != 0:
            quot += 1

        return ( quot if op == 0x5 else rem ) % 65536
```

Bitové logické operace se provádí po jednotlivých bitech (číslicích ve dvojkovém zápisu). Každá operace provede na odpovídajících bitech v operandech příslušnou logickou operaci (and, or nebo xor), čím získá odpovídající bit výsledku. Operandy jsou vždy 16bitové.

```
    if op in [ 0xa, 0xb, 0xc ]:
        result = 0
        for idx in range(16):
```

```
        bit_1 = op_1 // 2 ** idx % 2
        bit_2 = op_2 // 2 ** idx % 2

        if op == 0xa:
            bit_r = bit_1 == 1 and bit_2 == 1
        if op == 0xb:
            bit_r = bit_1 == 1 or  bit_2 == 1
        if op == 0xc:
            bit_r = ( bit_1 == 1 ) != ( bit_2 == 1 )

        result += bit_r * 2 ** idx

    return result
```

Bitové posuvy jsou jednoduché – posuv doleva odpovídá násobení a posuv doprava dělení příslušnou mocninou dvojky. Při pravých posuvech (dělení) musíme opět rozlišit bezznaménkovou (shr) a znaménkovou (sar) verzi. Znaménkový (tzv. aritmetický) posuv pak lze chápat i jako operaci, která posouvá jednotlivé bity doprava, ale zleva doplňuje místo nul kopie znaménkového bitu.

```
    if op == 0xd:
        return ( op_1 * 2 ** op_2 ) % 65536
    if op == 0xe:
        return ( op_1 // 2 ** op_2 ) % 65536
    if op == 0xf:
        return ( as_signed( op_1, 16 ) // 2 ** op_2 ) % 65536

    assert False
```

Čistá funkce compare realizuje aritmetická srovnání. Krom rovnosti ($=$, $\neq$) jsou všechny operace závislé na tom, pracujeme-li se znaménkovou reprezentací – v dvojkovém doplňkovém kódu jsou kódy záporných čísel větší, než kódy čísel kladných, např. 16bitové kódování -1 je 0xffff, přitom 16bitové kódování +1 je 0x0001, výsledek 0xffff < 0x0001 ale jistě v tomto kontextu nedává smysl.

```
def compare( op: int, arg_1: int, arg_2: int ) -> int:

    sig_1 = as_signed( arg_1, 16 )
    sig_2 = as_signed( arg_2, 16 )

    if op == 0x0: result = arg_1 == arg_2
    if op == 0xf: result = arg_1 != arg_2

    if op == 0x1: result = arg_1 <  arg_2
    if op == 0x2: result = arg_1 <= arg_2
    if op == 0x3: result = arg_1 >  arg_2
    if op == 0x4: result = arg_1 >= arg_2

    if op == 0xa: result = sig_1 <  sig_2
    if op == 0xb: result = sig_1 <= sig_2
    if op == 0xc: result = sig_1 >  sig_2
    if op == 0xd: result = sig_1 >= sig_2

    return 1 if result else 0
```

Tím je implementace step a jejích pomocných funkcí hotova. Za pomoci step je již velmi jednoduché implementovat podprogram run, který provede celý výpočet. Program je uložen od adresy 0.

```
CONT = 0
HALT = 1
ERROR = 2

def run( regs: list[ int ], mem: list[ int ] ) -> bool:
    print( "  pc inst op_1 op_2  out | " +
           "  rv   l1   l2   l3   l4   l5   l6   l7 " +
           "  t1   t2   t3   t4   t5   t6   sp   bp" )
    status = CONT
    pc = 0

    while status == CONT:
        pc, status = step( pc, regs, mem )

    return status == HALT
```

Další částí je podprogram read_program, která ze vstupního souboru přečte počáteční stav paměti (kterému můžeme také říkat **program**).

```
def write_nibble( mem: list[ int ], index: int, nibble: int ) -> None:
    mem[ index // 2 ] += nibble * 16 if index % 2 == 0 else nibble

def read_program( filename: str ) -> list[ int ]:
    mem = [ 0 for i in range( 65536 ) ]
    index = 0

    with open( filename, 'r' ) as file:
        for line in file.readlines():
            if line[ 0 ] == ';':
                break
            for word in line.rstrip().replace( "'", '' ).split( ' ' ):
                for hex_nibble in word:
                    write_nibble( mem, index, FROM_HEX[ hex_nibble ] )
                    index += 1

    return mem
```

Ukázky:

1. xxx

Přípravy:

1. fib – n-té fibonacciho číslo (mod 2^{16})
2. adder – dvouslovná sčítačka (32b)
3. gcd – euklidův algoritmus
4. prime – rozhodování prvočíselnosti
5. popcnt – spočítat jedničky ve slově
6. perfect – součet dělitelů

Řešené příklady:

1. reverse – otočení bitů ve slově
2. hamming – hammingova vzdálenost slov
3. packed – sečtení dvou dvojic po složkách
4. bitswap – prohození dvou bitů ve slově
5. collatz – počítání kroků iterovaného výpočtu
6. xxx

1.1: Strojový kód

V této kapitole budeme potřebovat 2 typy instrukcí – výpočetní (aritmetické, logické, atp.) a instrukce pro řízení toku (nepodmíněné a podmíněné skoky). Zejména prozatím nebudeme potřebovat pracovat s adresami, pamětí obecně, ani zásobníkem.

1.1.1 Kopírování hodnot Nejzákladnější operací, kterou můžeme v programu potřebovat, je nastavení registru, a to buď na předem známou konstantu, nebo na hodnotu aktuálně uloženou v některém jiném registru.

K nastavení registru na konstantu můžeme použít operaci put, která nastaví výstupní registr na hodnotu přímého operandu. Zápis této instrukce bude vypadat např. takto:

```
put 13    → rv
put 0x70 → l1
halt
```

Tento program nastaví registr rv na hodnotu 13 a registr l1 na hodnotu 112.

Pro kopírování hodnot mezi registry použijeme operaci copy – ta nastaví výstupní registr na tutéž hodnotu, jakou má registr vstupní. Například:

```
put  17 → l1
copy rv → l2 ; sets l2 = 13
copy l1 → rv ; sets rv = 17
halt
```

Po provedení tohoto programu budou hodnoty registrů rv = 17, l1 = 17 a l2 = 13.

1.1.2 Aritmetika Další důležitou kategorií jsou aritmetické instrukce. Následující tabulka shrnuje operace, které máte k dispozici. Registr l1 odpovídá proměnné a, registr l2 proměnné b, registr rv pak proměnné x.

název	python	tiny
sčítání	x = a + b	add l1, l2 → rv
odečítání	x = a - b	sub l1, l2 → rv
násobení	x = a * b	mul l1, l2 → rv
dělení	x = a // b	sdiv l1, l2 → rv
		udiv l1, l2 → rv
zbytek	x = a % b	smod l1, l2 → rv
		umod l1, l2 → rv

Všimněte si, že operaci celočíselného dělení a zbytku po dělení odpovídají dvě různé instrukce. Je to proto, že fyzicky jsou registry realizované jako sekvence binárních přepínačů – každý přepínač reprezentuje jeden bit. Tyto binární sekvence lze interpretovat různými způsoby, nicméně b-bitový registr obvykle chápeme jako:

1. celé číslo n bez znaménka v rozsahu $\langle 0, 2^b)$ – pak sekvence bitů přímo odpovídá binárnímu zápisu tohoto čísla,
2. jako celé číslo s se znaménkem v rozsahu $\langle -2^{b-1}, 2^{b-1})$, a to tak, že:
   a. je-li nejvyšší bit nastaven na 1, $s = n - 2^b$,
   b. jinak $s = n$

Podmínku z bodu (a) můžeme také chápat jako $[\![n \geq 2^{b-1}]\!]$.

Pro 16bitová čísla, která budeme v tomto předmětu používat zdaleka nejčastěji, to jsou tyto rozsahy:

- $\langle 0, 65535\rangle$ (nebo 0-ffff v šestnáctkovém zápisu) pro reprezentaci bez znaménka,
- $\langle -32768, 32767\rangle$ (nebo -8000 až 7fff šestnáctkově) pro reprezentaci se znaménkem.

Tato reprezentace má tu vlastnost, že sčítání, odečítání a násobení používá na úrovni bitů stejný algoritmus v obou případech – proto operace add funguje stejně dobře bez ohledu na to, chápeme-li operandy jako znaménkové nebo bezznaménkové.

To ale neplatí pro dělení (a nebude to platit ani pro srovnání, jak uvidíme za chvíli) – výsledek se bude lišit v závislosti na tom, je-li operace znaménková (sdiv, smod) nebo nikoliv (udiv, umod).

1.1.3 Srovnání Prakticky každý vyšší programovací jazyk má nějakou formu **podmíněného příkazu**. Aby byla tato konstrukce užitečná, potřebujeme mít k dispozici **predikáty** – operace, kterých výsledkem je pravdivostní hodnota. Ty nejběžnější již dobře znáte – jsou to celočíselné srovnávací operátory. V Pythonu je zapisujeme jako a == b, a < b, atp.

Náš výpočetní stroj má pro tento účel sadu operací – jsou shrnuty v tabulce níže. Jak již bylo výše naznačeno, s výjimkou rovnosti musíme rozlišovat znaménkovou a bezznaménkovou verzi. Na rozdíl od Pythonu (nebo jazyka C) nemá strojový kód složené výrazy, proto musíme výsledek srovnání vždy uložit do registru (analogem v Pythonu je booleovská proměnná – budeme ji zde opět značit x).

python	tiny		
x = a == b	eq l1, l2 → rv		**eq**ual
x = a != b	ne l1, l2 → rv		**n**ot **e**qual
x = a < b	slt l1, l2 → rv	**s**igned	**l**ess **t**han
	ult l1, l2 → rv	**u**nsigned	**l**ess **t**han
x = a > b	sgt l1, l2 → rv	**s**igned	**g**reater **t**han
	ugt l1, l2 → rv	**u**nsigned	**g**reater **t**han
x = a <= b	sle l1, l2 → rv	**s**igned	**l**ess or **e**qual
	ule l1, l2 → rv	**u**nsigned	**l**ess or **e**qual
x = a >= b	sge l1, l2 → rv	**s**igned	**g**reater or **e**qual
	uge l1, l2 → rv	**u**nsigned	**g**reater or **e**qual

Výsledek uložený do výstupního registru (v příkladech výše rv) je u instrukcí z této rodiny vždy 1 (pravda) nebo 0 (nepravda). To zejména znamená, že je možné tyto výsledky kombinovat operacemi and, or a xor a výsledek bude vždy opět 0 nebo 1, v souladu s definicí příslušné logické operace (k těmto se vrátíme níže).

1.1.4 Řízení toku Abychom mohli realizovat podmíněné příkazy a cykly, budeme k tomu potřebovat speciální operace – podobně jako příslušným příkazům ve vyšším jazyce jim budeme říkat **řízení toku**.

Výpočetní stroj tiny obsahuje 3 operace tohoto typu:

- jmp addr způsobí, že výpočet bude pokračovat od adresy addr – bez ohledu na aktuální stav registrů; adresu můžeme (a typicky budeme)

zadávat jako **symbol** (jméno **návěstí** – viz též část B.3),

- jz reg, addr (jump if zero) nejprve ověří, je-li hodnota registru reg nulová – pokud ano, provede skok stejně jako jmp addr, v případě opačném pokračuje na další instrukci bez jakéhokoliv dalšího efektu,
- jnz reg, addr (jump if not zero) se chová stejně, ale skok provede pouze je-li hodnota uložená v reg nenulová.

V kombinaci s aritmetickými a srovnávacími operacemi popsanými výše dokážeme zapsat jednoduchou podmínku např. takto (odpovídající program v Python-u je uveden v komentářích):

```
    put 1     → l1 ; a = 1
    slt l1, 3 → t1 ; t = a < 3
    jz  t1, else   ; if t:
then:
    put 2     → l2 ;     b = 2
    jmp endif      ; else:
else:
    put 3     → l2 ;     b = 3
endif:
    halt
```

Zkuste si program spustit pomocí tinyvm.py z kapitoly B, a také upravit první instrukci na put 5 → l1 a srovnejte výsledek. Podobně můžeme zapsat také while cyklus (cykly for do strojového kódu přímo přepsat nemůžeme, ale jak jistě víte, je vždy možné nejprve je přepsat na cykly while). Uvažme tento velmi jednoduchý program v Pythonu:

```
a = 1
while a < 3:
    a += 1
```

Přepis do strojového kódu bude opět vyžadovat určitou kreativitu, protože máme pouze instrukce skoku, nikoliv instrukce cyklu. Stačí si ale uvědomit, že while True se realizuje snadno: pomocí nepodmíněného skoku zpět (na nižší adresu).

```
    put 1     → l1   ; a = 1
loop:                ; while True:
    slt l1, 3 → t1   ;     t = a < 3
    jz  t1, end      ;     if not t: break
    add l1, 1 → l1   ;     a += 1
    jmp loop
end:
    halt
```

Cyklus while podmínka jsme přepsali na while True a podmíněný break – ekvivalenci těchto dvou zápisů si rozmyslete.

1.1.5 Bitové logické operace XXX

1.1.6 Bitové posuvy XXX

1.2: Programovací jazyk

Tato kapitola jazyk C nepoužívá.

1.d: Demonstrace (ukázky)

1.d.1 [triangle] V této ukázce napíšeme jednoduchý program, který rozhodne zadává-li trojice vstupních čísel trojúhelník (tzn. určí, zda vstup splňuje potřebné trojúhelníkové nerovnosti).

V Pythonu tento problém řeší výraz:

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

Nejprve si nachystáme testovací kód a testovací data (tuto část můžete při čtení přeskočit – to bude platit i v dalších ukázkách). Testovací kód načte testovací data z paměti – jak tyto instrukce fungují si blíže ukážeme v dalších kapitolách.

Vstup budeme očekávat v registrech l1, l2 a l3 a výsledek (0 nebo 1) zapíšeme do registru rv.

```
test: ; driver
    ld   l7, 0x24   → l1
    add  l7, 2      → l7
    ld   l7, 0x24   → l2
    add  l7, 2      → l7
    ld   l7, 0x24   → l3
    add  l7, 2      → l7
    eq   l1, 0xffff → t1 ; test-end marker
    jz   t1, solution
    halt

data:   ;  l1  l2  l3  →  rv
    .word  3,  4,  5,     1
    .word  1,  1,  1,     1
    .word  1,  1,  3,     0
    .word  2,  3,  1,     0
    .word -1,  0

check:
    ld   l7, 0x24   → t1
    eq   rv, t1     → t1
    asrt t1
    add  l7, 2      → l7
    jmp  test
.trigger set _tc_expect_ 4
.trigger inc _tc_

solution: ; zde začíná řešení
```

Protože potřebujeme implementovat konjunkci, nastavíme do registru rv její neutrální hodnotu – true, tzn. 1. Každý ze tří testů pak bude implementovaný stejně – sečte dvě strany, srovná tento součet se stranou třetí a výsledek tohoto srovnání přidá do registru rv bitovou operací and.

```
    put 1      → rv
```

Každý z následovných tří bloků realizuje jednu nerovnost. Mezivýsledky ukládáme do registrů t1 (součet stran) a t2 (výsledek srovnání součtu se zbývající stranou).

```
    add l1, l2 → t1
    sgt t1, l3 → t2
    and rv, t2 → rv

    add l1, l3 → t1
    sgt t1, l2 → t2
    and rv, t2 → rv

    add l2, l3 → t1
    sgt t1, l1 → t2
    and rv, t2 → rv
```

Tím je výpočet ukončen a protože je již výsledek uložen ve správném registru, nezbývá než předat řízení zpátky do testovacího kódu.

```
    jmp check
```

1.d.2 [factorial] V této ukázce naprogramujeme jeden z nejjednodušších číselných algoritmů vůbec – iterativní výpočet faktoriálu. Novým prvkem bude tedy **cyklus** – opakované spuštění stejného segmentu kódu.

Testovací kód můžete opět přeskočit, řešení začíná návěstím solution. Vstupní hodnota bude v registru l6, výsledek pak v registru rv.

```
test: ; driver
    ld   l7, 0x10   → l6
    eq   l6, 0xffff → t1 ; test-end marker
    jz   t1, solution
    halt

data:   ; input, expect
    .word    1,      1
    .word    2,      2
    .word    3,      6
    .word    4,     24
    .word   -1,      0
```

```
check:
    add  l7, 2      → l7
    ld   l7, 0x10   → t1
    eq   rv, t1     → t1
    asrt t1
    add  l7, 2      → l7
    jmp  test

.trigger set _tc_expect_ 4
.trigger inc _tc_

solution: ; zde začíná řešení
```

Registr rv budeme používat jako střadač (akumulátor) – před vstupem do cyklu jej nastavíme na neutrální hodnotu 1 a v každé iteraci jej vynásobíme počítadlem iterací.

Jako řídící proměnnou použijeme přímo l6 – protože na pořadí násobení nezáleží, můžeme hodnoty násobit sestupně v pořadí $n \cdot (n-1) \cdot (n-2) \ldots 1$. Jakmile řídící proměnná dosáhne hodnoty nula, cyklus ukončíme (musíme si pouze dát pozor, abychom nulou již nenásobili).

```
    put 1 → rv
loop:
    mul rv, l6 → rv
    sub l6, 1  → l6
    jnz l6, loop

    jmp check
```

1.p: Přípravy

1.p.1 **[fib]** Vaším úkolem je naprogramovat iterativní výpočet n-tého Fibonacciho čísla. Vstupní hodnotu n naleznete v registru l6, výsledek uložte do registru rv. Po skončení výpočtu proveďte skok na návěstí check. Hodnotu registru l7 zachovejte.

1.p.2 **[adder]** Vaším úkolem je tentokrát naprogramovat 32bitovou sčítačku. Vstupem jsou 4 16bitové hodnoty uložené v registrech l1 až l4, kde l1 a l3 jsou nižší a l2 a l4 jsou vyšší slova sčítanců. Nižší slovo výsledku uložte do rv, to vyšší pak do l6. Hodnotu v registru l7 neměňte.

1.p.3 **[gcd]** Vaším úkolem je naprogramovat Euklidův algoritmus pro nalezení největšího společného dělitele hodnot uložených v registrech l1 a l2 – obě hodnoty interpretujte jako 16bitová celá čísla bez znaménka.

Výsledek uložte do registru rv. Po ukončení výpočtu skočte na návěstí check. Hodnotu registru l7 neměňte.

1.p.4 **[prime]** Napište program, který rozhodne, je-li hodnota uložená v registru l6 prvočíslem (hodnotu interpretujte jako číslo bez znaménka). Výsledek (1 pokud prvočíslem je, 0 jinak) uložte do registru rv a proveďte skok na návěstí check. Hodnotu registru l7 nijak neměňte.

1.p.5 **[popcnt]** Napište program, který určí, kolik je jedniček v binárním zápisu čísla uloženého v registru l6. Výsledek uložte do registru rv a poté proveďte skok na návěstí check. Hodnotu registru l7 nijak neměňte.

1.p.6 **[abundant]** Napište program, který rozhodne, je-li hodnota uložená v registru l6 abundantním číslem (číslo n je abundantní, je-li součet všech jeho dělitelů $d > 2n$). Vstup interpretujte jako číslo bez znaménka.

Výsledek (1 pokud abundantní je, 0 jinak) uložte do registru rv a proveďte skok na návěstí check. Hodnotu registru l7 nijak neměňte.

1.r: Řešené úlohy

1.r.1 **[reverse]** Vaším úkolem bude otočit pořadí bitů ve slově. Vstupní slovo bude uloženo v registru l1, výsledek uložte do rv a skočte na návěstí check. Hodnotu v registru l7 neměňte.

1.r.2 **[hamming]** Spočtěte Hammingovu vzdálenost slov uložených v l1 a l2. Hammingovou vzdáleností je počet pozic, v nichž se vstupní slova liší hodnotou bitu. Výsledek uložte do rv a skočte na návěstí check. Hodnotu v registru l7 neměňte.

1.r.3 **[packed]** Mnohé procesory nabízí tzv. vektorové instrukce, které se k jednomu velkému registru chovají, jako by obsahoval několik menších čísel uložených vedle sebe. Vaším úkolem bude emulovat jednu z těchto instrukcí; konkrétně sčítání, které 16bitové registry považuje za dvojice dvou osmibitových čísel a dvě takové dvojice sečte po složkách. Vstup je v registrech l1 a l2, výsledek uložte do rv a skočte na návěstí check. Hodnotu v registru l7 neměňte.

1.r.4 **[bitswap]** Prohoďte dva zadané bity ve slově v registru l1. Indexy bitů jsou v l2 a l3. Můžete předpokládat, že budou v rozsahu 0–15, nula označuje nejméně významný bit slova. Výsledek uložte do rv a skočte na návěstí check. Hodnotu v registru l7 neměňte.

1.r.5 **[collatz]** Uvažme následující funkci f na kladných celých číslech:

```
f(n) = n / 2      je-li n sudé
f(n) = 3n +1      je-li n liché
```

Collatzova domněnka říká, že budeme-li na libovolné kladné celé číslo tuto funkci opakovaně aplikovat, dostaneme se nakonec k výsledku 1.

Vaším úkolem je tento výpočet provést a

1. spočítat, po kolika aplikacích funkce f na vstup je poprvé výsledkem jednička a
2. zjistit nejvyšší mezivýsledek, který při výpočtu vznikl.

Můžete předpokládat, že pro číslo na vstupu domněnka skutečně platí a že v průběhu výpočtu nevznikne mezivýsledek, který by se nevlezl do šestnáctibitového registru.

Počáteční číslo naleznete v registru l1. Počet aplikací funkce uložte do rv, nalezené maximum do l6 a skočte na návěstí check. Hodnotu v registru l7 neměňte.

Ukázky:

1. fib – lokální proměnné, cyklus
2. prime – aritmetika a rozsah číselných hodnot

Přípravy:

1. gcd – euklid podruhé
2. rand – generování pseudonáhodných čísel
3. collatz – počítání kroků iterovaného výpočtu
4. packed – součet n-tic po složkách
5. popcnt – počet nenulových cifer při daném základu
6. hamming – hammingova vzdálenost při daném základu

Řešené příklady:

1. palindrome – binární palindrom
2. largest – nejvyšší číslice při daném základu
3. factors – rozklad na prvočinitele
4. primes – rozklad na prvočinitele podruhé
5. transpose – překlopení bitové matice
6. balanced – vyvážená trojková soustava

Volitelné příklady:

1. digits – ciferný součet při daném základu
2. rotate – bitová rotace slova

2.1: Strojový kód

V této kapitole žádné nové operace potřebovat nebudeme – budeme se soustředit na jazyk C a jak se jeho základní konstrukce přeloží na operace, které známe z předchozí kapitoly.

2.2: Programovací jazyk

Počínaje touto kapitolou budeme většinu programů psát ve zjednodušené verzi jazyka C. V tomto kurzu budeme psát programy do jednoho souboru, který bude sestávat z definic typů (uvidíme později) a podprogramů. Na diskusi o sémantice podprogramů zatím nejsme připraveni, proto je budeme chápat jako syntaktickou obálku pro kód, který budeme psát.

Program bude typicky vypadat takto:

```
int podprogram( int parametr1, int parametr2 )
{
    …
}

int main()
{
    assert( podprogram( 1, 2 ) == 3 );
    …
}
```

Podprogram s názvem main bude v tomto kurzu vždy obsahovat testy, které ověřují základní funkcionalitu ostatních podprogramů. Můžete si do něj vždy přidat svoje vlastní testy. Zápis podprogram(1, 2) je volání (použití) podprogramu – prozatím jej nebudeme mimo testy potřebovat, protože jediné podprogramy, které budeme moct v tomto předmětu použít, jsou ty, které si sami napíšeme.

2.2.1 Hodnoty, objekty a proměnné Proměnné znáte již z kurzu IB111 – proměnné v jazyce C mají s těmi v Pythonu mnoho společného, ale mají také důležité odlišnosti. Prvním, v zásadě syntaktickým, rozdílem je, že v jazyce C musíme každou proměnnou **deklarovat** – to provedeme zápisem typ jméno; případně typ jméno = výraz;. První forma proměnnou pouze deklaruje, ale její počáteční hodnotu ponechá neurčenu – tuto hodnotu **není dovoleno použít**.

Typ proměnné určuje, jakých hodnot může nabývat – k dispozici máme prozatím tyto zabudované typy:

- unsigned – celé číslo v rozsahu 0 až 65535,[10]
- int – celé číslo v rozsahu -32768 do 32767,[11]
- bool – celé číslo, 0 nebo 1, které typicky reprezentuje pravdivostní hodnotu – 0 pro false, 1 pro true,
- signed char – celé číslo v rozsahu -128 až 127,
- unsigned char – celé číslo v rozsahu 0 až 255,
- char – typ se stejným rozsahem jako jeden z předchozích dvou (který z nich je určeno implementací), ale přesto z pohledu kontroly typů od obou odlišný.

Proměnná je v jazyce C pevně svázaná[12] s **objektem**. Objekt je **abstrakce paměti** – reprezentuje entitu, která je schopna pamatovat si **hodnotu**, jíž můžeme z objektu **přečíst** nebo do objektu **uložit** novou (a tím tu předchozí přepsat). Objekt tak můžeme chápat jako dvojí zobecnění paměťové buňky:

- místo jednoho bajtu si pamatuje **hodnotu** (která může mít potenciálně složitou vnitřní strukturu, i když takové zatím neumíme v jazyce C sestrojit),
- místo adresy má **identitu** – objekt můžeme „uchopit“ a pracovat s ním – obvykle tak, že tento objekt svážeme s proměnnou.

Realizace objektů je důležitým prvkem implementace programovacího jazyka a může se případ od případu lišit. Zejména není pravda, že by byl objekt pevně svázán s nějakou adresou nebo registrem – překladač může objekt transparentně přesouvat dle potřeby výpočtu.[13]

2.2.2 Život a rozsah platnosti Objekt, který je s proměnnou svázaný, vznikne právě deklarací, a zanikne opuštěním rozsahu platnosti této proměnné. Čtení objektu je implicitní – provede se kdykoliv proměnnou použijeme jako hodnotu ve výrazu, zápis do objektu pak provedeme operátorem přiřazení (viz také další sekce).

Podobně jméno proměnné je platné počínaje deklarací, a konče pravou složenou závorkou, která ukončuje nejbližší uzavírající blok (složený příkaz nebo tělo funkce – podrobněji rozebereme dále). Například:

```
{
    // zde x ještě není deklarováno
    int x;
    {
        int y;
        … // zde můžeme použít jak x tak y
    } // zde končí rozsah platnosti y
    … // zde již y není lze použít
} // zde končí rozsah platnosti x
```

U proměnných je tak syntakticky zaručeno, že jsou svázány s živým objektem – kdykoliv můžeme jméno proměnné použít, objekt, který tato proměnná pojmenovává, existuje.

2.2.3 Výrazy Na úrovni jazyka C je základní jednotkou výpočtu **výraz** – podobně jako v jazyce Python můžeme výrazy tvořit induktivně. Jsou-li:

- e_1, e_2 … e_n výrazy,
- var jméno proměnné,
- lit číselný literál (konstanta),

existují také výrazy tvaru:[14]

[10] Pro typy int a unsigned je konkrétní rozsah přípustných hodnot daný implementací – na mnoha systémech jsou tyto typy 32bitové.

[11] Starší standardy jazyka C neurčují, jaké kódování se použije pro znaménkové typy, novější již požadují dvojkový doplňkový kód (viz také předchozí kapitola).

[12] Na rozdíl od jazyka Python, kde je možné vazbu proměnné na objekt změnit přiřazením. To v jazyce C možné není.

[13] Překladače jazyka C například běžně přesouvají objekty mezi registry a zásobníkem podle aktuální situace. Tentýž objekt může být tedy v různých fázích výpočtu fyzicky uložen na různých místech.

[14] S dalšími operátory se setkáme v pozdějších kapitolách.

1. lit (konstanta) je výraz,
2. var (jméno proměnné) je výraz,
3. použití aritmetického operátoru (binární v infixovém zápisu, unární v prefixovém):
   - e_1 + e_2, e_1 - e_2,
   - e_1 * e_2, e_1 / e_2, e_1 % e_2 (modulo)
   - unární mínus -e_1,
4. relační operátory:
   - e_1 == e_2 (rovnost), e_1 != e_2 (nerovnost)
   - e_1 <= e_2, e_1 >= e_2, e_1 < e_2, e_1 > e_2
5. bitové logické operace a posuvy:
   - binární e_1 & e_2 (and), e_1 | e_2 (or), e_1 ^ e_2 (xor),
   - unární ~e_1 – bitová negace,
   - bitové posuvy zapisujeme e_1 >> e_2, e_1 << e_2,
6. **operátory** přiřazení (pozor na změnu oproti jazyku Python – v jazyce C je přiřazení výraz, nikoliv příkaz):
   - jednoduché var = e_1,
   - složené var += e_1, var -= e_1,
   - dále var *= e_1, var /= e_1, var %= e_1,
   - s bitovým posuvem var <<= e_1, var >>= e_2,
   - s bitovou operací var &= e_1, var ^= e_1, var |= e_1,
7. operátory zvýšení a snížení proměnné o jedničku:
   - prefixové ++var, --var,
   - postfixové var++, var--,
8. operátor čárka, e_1, e_2,
9. booleovské logické operace:
   - binární e_1 && e_2 (and), e_1 || e_2 (or),
   - unární !e_1,
   - ternární e_1 ? e_2 : e_3,

2.2.4 **Vyhodnocení výrazu** Nyní víme, jak výrazy vypadají (jakou mají **syntaxi**), můžeme tedy přistoupit k otázce, co takové výrazy **znamenají** (jakou mají **sémantiku**). Všechny zde uvedené výrazy[15] popisují nějakou **hodnotu** a výraz samotný je návodem, jak tuto hodnotu získat.

Vyhodnocení výrazu (provedení výpočtu tímto výrazem popsaného) budeme samozřejmě realizovat pomocí již zavedeného výpočetního stroje tiny. Abychom mohli výpočet skutečně provést, musíme určit registr, do kterého má být výsledek zapsán – budeme mluvit o **vyhodnocení výrazu E do registru R**.[16]

1. Výraz lit se vyhodnotí přímo na číselnou hodnotu zapsanou ve zdrojovém kódu. Například vyhodnocení výrazu 7 do registru rv se realizuje instrukcí put 7 → rv.
2. Výraz var se vyhodnotí na hodnotu, která je v momentě vyhodnocení tohoto výrazu uložena v objektu svázaném s proměnnou var. Prozatím uvažujeme pouze situace, kdy je objekt svázaný s var uložen přímo v registru. Je-li např. var uloženo v l1, vyhodnocení výrazu var do registru rv realizujeme instrukcí copy l1 → rv.
3. Uvažme nyní výraz tvaru e_1 + e_2. Víme, že e_1 a e_2 popisují nějaké hodnoty. Abychom mohli vyčíslit hodnotu e_1 + e_2, budeme nejprve potřebovat tyto hodnoty. Na to použijeme **dočasné registry** – vyhodnocení e_1 + e_2 do rv bude vypadat takto:
   a. vyhodnoť e_1 do registru t_1,
   b. vyhodnoť e_2 do registru t_2,
   c. proveď add t_1, t_2 → rv.

   Musíme samozřejmě zabezpečit, že výpočet e_2 nepřepíše registr t_1 – jak přesně se toho dosáhne budeme zkoumat později.[17]

   Analogicky se vypočtou ostatní aritmetické, bitové, atd. operátory (k hodnotám s/bez znaménka a operacím dělení se ještě vrátíme).
4. Výrazy tvaru var = e_1 mají krom hodnoty také **vedlejší efekt** – zápis do objektu svázaného s proměnnou var. Jejich realizace vypadá takto – vyhodnocujeme do registru rv, objekt svázaný s var nechť žije v l1:
   a. vyhodnoť e_1 do registru rv
   b. proveď copy rv → l1.

   Všimněte si, že hodnota e_1 je zároveň hodnotou celého výrazu, a zůstává uložená v registru rv, jak bylo požadováno.
5. Složené přiřazení var += e_1 je analogické, pouze je operaci copy předřazena příslušná aritmetická nebo logická operace:[18]
   a. vyhodnoť e_1 do registru t_1
   b. proveď add t1, l1 → rv,
   c. proveď copy rv → l1.

   Výrazy zvýšení a snížení o jedničku jsou analogické, liší se pouze ve výsledné hodnotě. Prefixové verze, ++var, --var jsou pouze syntaktické zkratky pro var += 1 resp. var -= 1, ale postfixové se liší – vyhodnocení var++ do rv proběhne takto (var je svázáno s l1):
   a. proveď copy l1 → rv,
   b. proveď add l1, 1 → l1.

   Hodnota výrazu var++ je tedy **původní** hodnota var, předtím, než bylo provedeno zvýšení proměnné o jedničku.
6. Výraz e_1, e_2 představuje „zapomenutí hodnoty“ výrazu e_1 – výraz e_1 je proveden pouze pro svoje vedlejší efekty (např. výše uvedené přiřazení). Vyhodnocení e_1, e_2 do registru rv lze realizovat např. takto:
   a. vyhodnoť e_1 do rv,
   b. vyhodnoť e_2 do rv.
7. Zbývá zatím nejsložitější typ výrazů, a to jsou booleovské logické operace. XXX

Uvažme nyní několik konkrétních příkladů:

1. var + 1 se vypočte XXX

2.2.5 **Příkazy**

- výraz + středník
- složený příkaz
- if, else
- for
- while
- break
- continue

2.d: Demonstrace (ukázky)

2.d.1 **[fib]** Toto je první ukázka v jazyce C. Struktura souboru se bude od programů v jazyce symbolických adres poněkud lišit. Přesto, že zatím nevíme jak podprogramy pracují, budou součástí kostry a řešení tedy budeme vkládat do nachystaného podprogramu.

Práci s lokálními proměnnými a základní konstrukce řízení toku – podmíněný příkaz a cyklus – si demonstrujeme na klasickém iterativním algoritmu pro výpočet n-tého Fibonacciho čísla.

```
int fib( int n )
{
```

První dvě Fibonacciho čísla jsou 1, 1 – uložíme je do proměnných a, b. V jazyce C jsou proměnné pevně svázány s **objekty** – jakmile proměnná (jméno) vznikne, její vazbu na objekt již není možné změnit. Operátor přiřazení způsobí **změnu hodnoty objektu**. Více o proměnných a přiřazení naleznete v sekci 2.2.

Zejména si dejte pozor na to, že sémantika proměnných v jazyce C se výrazně liší od té, kterou znáte z jazyka Python!

Deklarace lokálních proměnných zapisujeme takto:

1. jména typu, v tomto případě znaménkového celočíselného typu int,
2. neprázdného seznamu deklarovaných jmen (oddělených čárkou), které mohou být doplněny tzv. deklarátory (označují např. ukazatele: uvidíme

[15] Toto tvrzení v jazyce C neplatí obecně pro všechny výrazy – existují i takové, které hodnotu nemají.

[16] Tato konstrukce skutečně tvoří základ překladu výrazů v překladači jazyka C. Rozdílem je, že překladač pracuje s dočasnými registry mnohem hospodárněji, než naivní překlad zde popsaný – tím šetří nejen volné registry, ale i instrukce, které by hodnoty mezi registry zbytečně přesouvaly. Toto platí i pro velmi jednoduché překladače (např. také tinycc).

[17] Prozatím si vystačíme s představou, že při překladu udržujeme množinu volných dočasných registrů (takových, které jsme zatím nepoužili, nebo kterých hodnotu už jsme upotřebili, a nebudeme ji v dalším výpočtu potřebovat). Je asi jasné, že ať začneme s jakkoliv velkou konečnou množinou dočasných registrů, při výpočtu dostatečně složitého výrazu nám musí dojít – jak se s tímto problémem vypořádat si ukážeme v příští kapitole.

[18] Pro výrazy tvarů, které jsme zatím zavedli, je var += e_1 ekvivalentní výrazu var = var + e_1. V obecném případě, kdy je na levé straně složeného přiřazení složitější výraz (a nikoliv pouze název proměnné), to už ale neplatí!

je v pozdější ukázce),
3. volitelného inicializátoru, který popisuje počáteční hodnotu proměnné.

```
int a = 1, b = 1;
```

Jazyk C má 3 typy cyklů. Cyklus for používáme zejména v situaci, kdy předem známe potřebný počet iterací. Má následující strukturu:

1. klíčové slovo for,
2. hlavička cyklu, uzavřená v kulatých závorkách,
   a. inicializační příkaz (výraz, deklarace proměnné, nebo prázdný příkaz) je vždy ukončen středníkem a provede se jednou před začátkem cyklu; deklaruje-li proměnné, tyto jsou platné právě po dobu vykonávání cyklu,
   b. podmínka cyklu (**výraz** nebo prázdný příkaz) je opět vždy ukončena středníkem a určuje, zda se má provést další iterace cyklu (vyhodnotí-li se na true),
   c. **výraz** iterace (výraz, který **není** ukončen středníkem), který je vyhodnocen **vždy** na konci těla (před dalším vyhodnocením podmínky cyklu),
3. tělo cyklu (libovolný příkaz, často složený).

Všimněte si také, že podmínka cyklu i < n - 2 je složený výraz – jedná se o použití operátoru < (menší než), přitom pravý operand – n - 2 je opět použití operátoru (-, odečítání). Je velmi důležité si uvědomit, že náš výpočetní stroj takové výrazy neumí přímo zpracovat – dekomponovat takovéto výrazy na sekvence instrukcí je jednou z hlavních úloh překladače jazyka C.

Strojový kód k této ukázce si zobrazíte příkazem

```
$ tinycc -S d1_fib.c
```

Realizaci podmínky cyklu naleznete ve strojovém kódu pod návěstím .cond.1 – jedná se o instrukce sub l1, 2 → t1 a slt l4, t1 → t1. Překladač zde použil registr t1 pro uložení **mezivýsledku**, který je na úrovni jazyka C implicitní a nemá žádné jméno.

```
for ( int i = 0; i < n - 2; ++i )
{
```

Platnost proměnných má v jazyce C, opět na rozdíl od jazyka Python, blokovou strukturu – složené příkazy (bloky) zároveň tvoří rozsah platnosti jmen. Proměnná, která je deklarovaná uvnitř bloku, existuje pouze během vykonávání tohoto bloku. Po ukončení bloku (uzavírací složenou závorkou) tato proměnná, příslušný objekt a tedy i uložená hodnota přestanou existovat.

Cyklus v tomto ohledu není nijak speciální – před další iterací cyklu je blok ukončen. Je-li tedy **v těle** cyklu deklarovaná proměnná, jedná se v každé iteraci o zcela novou proměnnou. Srovnejte situaci s řídící proměnnou (v tomto případě i).

```
    int c = a + b;
```

Posun výpočetního „okna“ realizujeme přiřazením. Počet existujících objektů (a tedy využitá paměť nebo registry) se přiřazením nemění, pouze se provede kopie hodnoty z jedné strany na druhou.

```
    a = b;
    b = c;
}
```

Podprogramy jsme se zatím nezabývali, nicméně příkaz return ukončí vykonávání podprogramu a volajícímu předá návratovou hodnotu, podobně jako v jazyce Python.

```
    return b;
}

int main() /* demo */
{
```

Na tomto místě opět trochu předběhneme látku – použití (volání) podprogramu je **výraz** a jeho vyhodnocení odpovídá naší intuitivní představě: skutečné parametry (uvedené v kulatých závorkách za jménem) se použijí jako pomyslné inicializátory formálních parametrů a s takto inicializovanými parametry se vykoná tělo podprogramu. Po jeho ukončení se výraz volání podprogramu vyhodnotí na návratovou hodnotu. Jak se tento mechanismus realizuje na úrovni výpočetního stroje si povíme v příští kapitole.

Speciální výraz assert (tvrzení) vyhodnotí svůj parametr a je-li výsledek false, program ukončí s chybou.

```
    assert( fib( 1 ) == 1 );
    assert( fib( 2 ) == 1 );
    assert( fib( 7 ) == 13 );
    assert( fib( 20 ) == 6765 );

    return 0;
}
```

2.d.2 **[prime]** V této ukázce vyřešíme klasický úkol rozhodování prvočíselnosti, a to metodou pokusného dělení. Hlavička podprogramu deklaruje number jako hodnotu typu unsigned – celé číslo bez znaménka. Protože náš cílový stroj – tiny – má slova velikosti 16 bitů, bude i typ unsigned 16bitový.[19]

```
bool is_prime( unsigned number )
{
```

Protože nejmenší prvočíslo je 2, a protože zbytek algoritmu pro hodnoty 1 ani 0 nebude pracovat správně, tyto hodnoty rovnou zamítneme.

```
    if ( number < 2 )
        return false;
```

V lokální proměnné divisor budeme udržovat aktuální pokusný dělitel.

```
    unsigned divisor = 2;
```

Nejjednodušší podmínka cyklu, která by zde fungovala, je divisor < number. To je ale relativně neefektivní, protože je-li n prvočíslo, provedeme mnoho zbytečných iterací.

Je-li totiž d dělitelem n, existuje nějaké c takové, že $cd = n$. Je-li $c \leq d$, potom $c^2 \leq cd = n$ a tedy $c^2 \leq n$ (je-li $d \leq c$, platí analogicky $d^2 \leq n$).

Proto existuje-li libovolný dělitel v intervalu $\langle 2, n)$, existuje i takový, pro který $d^2 \leq n$. Tuto podmínku umíme jednoduše zapsat s použitím základní aritmetiky a srovnání (operace, které jsou k dispozici tak v stroji tiny, tak v jazyce C). Přesto ale narazíme na problém: nepracujeme totiž se skutečnými celými čísly, ale s 16bitovými slovy.

A zde potřebné nerovnosti mohou selhat, je-li výsledek násobení mimo rozsah daného typu – v našem případě mimo rozsah $\langle 0, 2^{16})$. Konkrétně např. $261^2 = 68121$, šestnáctkově 0x10a19 – pokusíme-li se toto číslo zapsat do 16 bitů, dostaneme 0x0a19, desítkově 2585.

Jaké má tento jev důsledky pro náš výpočet? Uvažme např. test prvočíselnosti 65521. Pro divisor = 255 se divisor * divisor vyhodnotí na 65025, což je méně než 65521 a proto se vykoná tělo cyklu. V další iteraci dostaneme divisor = 256, ale divisor * divisor se na 16 bitech vyhodnotí na nulu a do cyklu tedy opět vstoupíme – a to i přesto, že při standardním celočíselném násobení bychom dostali výsledek $256^2 = 65536 > 65025$. Tento problém bude pokračovat po mnoho iterací – takto zapsaný cyklus se zastaví až pro hodnotu divisor = 16203, kdy dostáváme:

- $16203^2 = 262537209$, šestnáctkově 0xfa5fff9,
- omezeno na 16 bitů 0xfff9, desítkově tedy 65529.

To ale jistě není počet iterací, který jsme očekávali – náš plán byl, že se cyklus provede pouze 256krát.

```
    unsigned sqrt_uint_max = 1 << sizeof( unsigned ) * 4;

    while ( divisor <= sqrt_uint_max &&
            divisor * divisor <= number )
    {
```

Zjistíme-li, že divisor dělí hodnotu number beze zbytku, jistě to znamená, že number není prvočíslo (divisor nemůže být ani 1 ani rovný number).

```
        if ( number % divisor == 0 )
            return false;
```

[19] Standard jazyka C neurčuje přesné velikosti základních typů. Na většině současných systémů je typ unsigned 32bitový (a to i na 64bitových architekturách), ale není to nijak zaručeno.

```
        ++ divisor;
    }

    return true;
}

int main() /* demo */
{
    assert( is_prime( 2 ) );
    assert( is_prime( 3 ) );
    assert( is_prime( 5 ) );
    assert( is_prime( 13 ) );
    assert( is_prime( 29 ) );
    assert( is_prime( 97 ) );
    assert( is_prime( 619 ) );

    assert( !is_prime( 1 ) );
    assert( !is_prime( 4 ) );
    assert( !is_prime( 6 ) );
    assert( !is_prime( 8 ) );
    assert( !is_prime( 68 ) );
    assert( !is_prime( 77 ) );
    assert( !is_prime( 81 ) );
    assert( !is_prime( 323 ) );
    assert( !is_prime( 36863u ) );

    assert( is_prime( 65521u ) );
}
```

2.p: Přípravy

2.p.1 **[gcd]** Implementujte podprogram gcd, který Euklidovým algoritmem spočte největší společný dělitel dvou přirozených čísel, která obdrží jako argumenty.

```
unsigned gcd( unsigned x, unsigned y )
{
```

Řešení pište sem. x a y jsou proměnné obsahující vstupní čísla. Výsledek vraťte příkazem return.

```
    return x;
}
```

2.p.2 **[rand]** V této přípravě budete generovat pseudonáhodná 32bitová čísla. Protože náš stroj ani jazyk 32bitovou aritmetiku nepodporuje, budeme každé takové číslo representovat dvěma 16bitovými hodnotami hi (horních 16 bitů) a lo (spodních 16 bitů). Protože zatím neumíme z podprogramu vrátit dvě hodnoty zároveň, napíšete podprogramy dva, každý vracející jednu polovinu 32bitového čísla.

Vzorec pro generování dalšího čísla v řadě na základě předchozího vypadá takto:

$$\mathrm{rand}(\mathrm{prev}) = \mathrm{prev} \cdot 1103515245 + 12345$$

Nápověda: Vynásobte si na papír pod sebe dvě čtyřciferná čísla. Každá cifra je jeden bajt. Vynásobení dvou bajtů a přičtení třetího se do jednoho slova vejde: 0xff · 0xff + 0xff = 0xff00

rand_hi vrací horních 16 bitů výsledku.

```
unsigned rand_hi( unsigned hi, unsigned lo )
{
    return 0;
}
```

rand_lo vrací spodních 16 bitů výsledku.

```
unsigned rand_lo( unsigned hi, unsigned lo )
{
    return 0;
}
```

2.p.3 **[collatz]** Uvažme následující funkci[20] f na kladných celých číslech:

```
f(n) = n / 2      je-li n sudé
f(n) = 3n +1      je-li n liché
```

Collatzova domněnka říká, že budeme-li na libovolné kladné celé číslo tuto funkci opakovaně aplikovat, dostaneme se nakonec k výsledku 1.

Implementujte podprogram collatz_lds, který na svůj vstup bude opakovaně aplikovat funkci f než dojde k jedničce. Výsledkem podprogramu bude délka nejdelší klesající posloupnosti mezivýsledků, tj. kolikrát nejvíce po sobě prováděl půlení čísla.

Můžete předpokládat, že pro číslo na vstupu domněnka skutečně platí a že v průběhu výpočtu nevznikne mezivýsledek, který by se nevlezl do šestnáctibitového registru.

```
unsigned collatz_lds( unsigned n )
{
    return 0;
}
```

2.p.4 **[packed]** Uvažme slovo, které nereprezentuje jedno šestnáctibitové číslo, ale n-tici několika kratších čísel uložených vedle sebe (například 2 osmibitová nebo 4 čtyřbitová). Vaším úkolem bude čísla ve dvou takovýchto n-ticích sečíst po složkách a výsledek opět vrátit jako n-tici stejného tvaru.

Implementujte podprogram packed_add, který toto sčítání provede. Jeho první dva argumenty jsou slova reprezentující n-tice, třetím je šířka čísla v n-tici. V případě, že tato šířka nedělí šířku slova, je číslo uložené v nejvyšších bitech kratší. Můžete předpokládat, že šířka nebude nulová.

```
unsigned packed_add( unsigned v1, unsigned v2, unsigned width )
{
```

Řešení pište sem. v1 a v2 jsou proměnné obsahující vstupní n-tice, width obsahuje šířku čísla v n-tici. Výsledek vraťte příkazem return

```
    return 0;
}
```

2.p.5 **[popcnt]** Implementujte podprogram popcnt, který spočítá počet nenulových číslic representace zadaného čísla při zadaném základu. Můžete předpokládat, že základ je alespoň dva.

```
unsigned popcnt( unsigned n, unsigned base )
{
```

Řešení pište sem. n je proměnná obsahující vstupní číslo, base obsahuje základ. Výsledek vraťte příkazem return

```
    return 0;
}
```

2.p.6 **[hamming]** Implementujte podprogram hamming, který spočítá Hammingovu vzdálenost mezi reprezentacemi dvou zadaných čísel při daném číselném základu, tj. počet pozic, v nichž se reprezentace čísel liší cifrou. Je-li jedna z reprezentací kratší, doplňte ji zleva nulami do délky delší reprezentace. Můžete předpokládat, že základ je alespoň dva.

```
unsigned hamming( unsigned x, unsigned y, unsigned base )
{
```

Řešení pište sem. x a y jsou proměnné obsahující vstupní čísla, base obsahuje základ. Výsledek vraťte příkazem return

```
    return 0;
}
```

2.r: Řešené úlohy

2.r.1 **[palindrome]** Implementujte predikát is_binary_palindrome, který o zadaném čísle rozhodne, zda je jeho binární reprezentace palindromem.

```
bool is_binary_palindrome( unsigned n )
{
```

[20] Ač mluvíme o matematické **funkci** f, neočekáváme, že tato bude implementována samostatným podprogramem (tzv. funkcí) jazyka C. O těch zatím v kurzu řeč nebyla.

Řešení pište sem.

```
    return false;
}
```

2.r.2 **[largest]** Implementujte podprogram largest_digit, který pro zadané číslo n vrátí jeho nejvyšší číslici při základu base.

```
unsigned largest_digit( unsigned n, unsigned base )
{
```

Řešení pište sem.

```
    return 0;
}
```

2.r.3 **[factors]** Implementujte podprogram factor_count, který vrátí počet všech činitelů v prvočíselném rozkladu zadaného kladného čísla. Opakující se prvočinitele započítejte opakovaně.

```
unsigned factor_count( unsigned n )
{
    assert( n > 0 );
```

Řešení pište sem.

```
    return 0;
}
```

2.r.4 **[primes]** Implementujte podprogram prime_count, který vrátí počet unikátních činitelů v prvočíselném rozkladu zadaného kladného čísla. Opakující se prvočinitele započítejte pouze jednou.

```
unsigned prime_count( unsigned n )
{
    assert( n > 0 );
```

Řešení pište sem.

```
    return 0;
}
```

2.r.5 **[transpose]** Označíme-li postupně bity ve slově tak, že nejméně významný je 0 a nejvýznamnější je F, pak slovo reprezentuje následující bitové čtvercové matice o velikostech 4×4 až 1×1:

```
F E D C   8 7 6   3 2   0
B A 9 8   5 4 3   1 0
7 6 5 4   2 1 0
3 2 1 0
```

Implementujte podprogram transpose, který zadanou matici transponuje, tj. vrátí slovo reprezentující jednu z matic:

```
F B 7 3   8 5 2   3 1   0
E A 6 2   7 4 1   2 0
D 9 5 1   6 3 0
C 8 4 0
```

Velikost jedné strany matice je také argumentem podprogramu. Můžete předpokládat, že se bude jednat o číslo mezi 1 a 4. Pro velikosti menší než 4 nevyužité bity na vstupu ignorujte a ve výsledku nechť jsou nulové.

```
unsigned transpose( unsigned m, int size )
{
```

Řešení pište sem.

```
    return 0;
}
```

2.r.6 **[balanced]** Vyvážená trojková soustava nepoužívá číslice s hodnotami 0, 1 a 2, nýbrž 0, 1 a -1 (zapisované zde 0, + a −). Například číslo dvě se v ní representuje jako +−: $1 \cdot 3^1 + (-1) \cdot 3^0$.

Implementujte podprogram balanced_digits, který vrátí dvě 8bitová čísla zabalená (jako v p4_packed) do jednoho slova: spodní slabika bude obsahovat počet číslic + a horní počet číslic − ve vyváženětrojkovém zápisu zadaného **celého** čísla.

```
unsigned balanced_digits( int n )
{
```

Řešení pište sem.

```
    return 0;
}
```

2.v: Volitelné úlohy

2.v.1 **[digits]** Implementujte podprogram digit_sum, který spočítá ciferný součet čísla při zadaném základu. Můžete předpokládat, že základ je alespoň dva.

```
unsigned digit_sum( unsigned n, unsigned base )
{
```

Řešení pište sem.

```
    return 0;
}
```

2.v.2 **[rotate]** Implementujte podprogram rotate, který provede bitovou rotaci zadaného slova o zadaný počet pozic. Kladný počet provádí rotaci vlevo, záporný vpravo.

```
unsigned rotate( unsigned word, int amount )
{
```

Řešení pište sem.

```
    return 0;
}
```

Část 3: Podprogramy

Tento týden se budeme (konečně) zabývat podprogramy. Tyto nám zejména umožní stavět **abstrakce**, dekomponovat problémy a organizovat programy. Důležitou formou využití podprogramů je také **rekurze**, při které uplatníme jak dekompozici tak abstrakci, a která nám dává jednoduchý mechanismus k řešení řady náročných problémů.

Ukázky:

1. cycle – hledání délky cyklu v permutaci
2. pow – algoritmus pro celočíselné mocnění

Přípravy:

1. cellular – buněčný automat nad slovem
2. permute – jsou dvě slova vzájemnou permutací v base 3
3. rotate – jsou dvě slova vzájemnou rotací
4. fractal – n-tý člen fraktální posloupnosti
5. palindrome – nejmenší počet palindromových segmentů
6. sat – splnitelnost malé výrokové formule

Řešené příklady:

1. power – modulární mocnění
2. squares – součty druhých mocnin
3. knight – skákání koněm
4. bezout – rozšířený Euklidův algoritmus
5. xxx
6. xxx

3.1: Strojový kód

V této kapitole přidáváme operace pro práci s podprogramy – zejména call a ret, a zásobníkem – push a pop.

XXX

3.2: Programovací jazyk

Tato kapitola zavádí důležitý nový koncept, totiž **podprogram** a s ním související nový typ výrazu – volání (použití) podprogramu.

3.2.1 **Definice** Syntaxi **definice podprogramu** již zběžně známe:

```
typ₀ jméno₀( typ₁ jméno₁, … ,typₙ jménoₙ )
{
    příkaz₁
    příkaz₂
    …
    příkazₘ
}
```

Jednotlivé prvky zápisu mají tento význam:

- typ_0 je tzv. **návratový typ** – může být prozatím pouze jeden z již známých typů (int, unsigned, …),
- $jméno_0$ je název zaváděného podprogramu,
- typ_1 … typ_n jsou typy jednotlivých **parametrů**,
- $jméno_1$ … $jméno_n$ jsou **jména** parametrů,
- $příkaz_1$ … $příkaz_m$ tvoří **tělo** podprogramu.

3.2.2 **Výrazy** Hlavní nový typ výrazů je v této kapitole **použití podprogramu** nebo také **volání podprogramu** (možná znáte také jako „volání funkce“). Tento typ výrazu má následovný tvar:

funcall ≡ jméno($expr_1$, $expr_2$, …, $expr_n$)

Předpokládejme, že jméno odpovídá definici z předchozí sekce. Výraz je pak typově správný v případě, že:

- typ výrazu $expr_1$ je možné implicitně převést na typ_1,
- typ $expr_2$ na typ typ_2, atd., až $expr_n$ na typ_n.

Typ výrazu funcall jako celku je pak typ_0 z definice výše.

Vyhodnocení tohoto výrazu probíhá následovně:

1. Pro každý formální parametr je vytvořen objekt odpovídajícího typu; uvnitř těla podprogramu pak jména formálních parametrů pojmenovávají právě tyto objekty.
2. Výrazy $expr_1$ až $expr_n$ jsou vyhodnoceny na hodnoty (v blíže neurčeném pořadí!) a každá takto získaná hodnota je zapsána do příslušného objektu z předchozího bodu.
3. Řízení je předáno tělu podprogramu (vzniká při tom mimo jiné nový rozsah platnosti jmen).
4. Hodnota celého výrazu funcall je pak určena prvním spuštěným příkazem return uvnitř těla. Tento příkaz zároveň předá řízení zpátky volajícímu podprogramu.

3.2.3 **Příkazy** Nový příkaz return expr; má dva efekty:

1. vyhodnotí výraz expr a jeho hodnotu **vrátí** volajícímu (viz předchozí podsekce),
2. ukončí vykonávání podprogramu a předá řízení volajícímu.

3.d: Demonstrace (ukázky)

3.d.1 **[cycle]** Předmětem této ukázky bude algoritmus pro hledání délky cyklu na kterém leží zadaný prvek v zadané permutaci. Permutace σ na množině M je bijektivní zobrazení $M \to M$. Cyklus je pak posloupnost prvků, které dostaneme opakovanou aplikací σ.

Permutace bude zadaná podprogramem (čistou funkcí) permute, která pro zadanou hodnotu vrátí její obraz. Protože pracujeme s 16bitovými hodnotami, je zaručeno, že hledaný cyklus bude mít délku nejvýše 2^{16}.

Pro permute uvedeme tzv. prototyp – deklaraci podprogramu, která zavede jeho jméno, návratový typ a typy a počet parametrů. Definici naleznete na konci souboru.

```
unsigned permute( unsigned n );
```

Poznámka: různé počáteční hodnoty mohou vést k různým posloupnostem hodnot, přitom každá z nich bude periodická a tyto posloupnosti budou po dvou disjunktní (rozmyslete si, proč tomu tak je – v obecné rovině, bez ohledu na konkrétní definici permute – vhodné zdůvodnění bude fungovat pro libovolnou permutaci).

Algoritmus implementujeme jako samostatný podprogram (opět čistou funkci). Podprogram budeme tentokrát rovnou definovat – definice podprogramu sestává z **prototypu** (ve stejném tvaru jako výše) a **těla**, které obsahuje příkazy, které se při aktivaci (volání) podprogramu provedou. Parametr initial určí počáteční hodnotu, pro kterou budeme délku cyklu určovat.

```
unsigned find_period( unsigned initial )
{
```

Samotný algoritmus je velmi jednoduchý – stačí zjistit, kolikrát musíme funkci permute zavolat, abychom podruhé dostali stejnou hodnotu. Zejména si tedy nemusíme pamatovat všechny prvky takto vygenerované posloupnosti, ani je mezi sebou srovnávat.

```
    unsigned last = initial;
    unsigned length = 0;
```

Použijeme zde cyklus typu do-while, který asi nejlépe vystihuje podstatu problému – zejména nám umožní bez větších komplikací zapsat podmínku cyklu. Jistě je ale možné použít řadu jiných zápisů a konkrétní volba je vesměs otázkou osobních preferencí.

```
    do
```

```
    {
        ++ length;
        last = permute( last );
    }
    while ( last != initial );
```

Protože cyklus skončil, platí last == initial (negace podmínky cyklu) a v proměnné length je počet provedených iterací.

```
    return length;
}

int main() /* demo */
{
    assert( find_period( 1 ) == 3 );
    assert( find_period( 5 ) == 3 );
    assert( find_period( 14 ) == 13 );

    return 0;
}

unsigned permute( unsigned n )
{
    if ( n < 13 )
        return n / 3 * 3 + ( n + 1 ) % 3;
    else
        return n / 13 * 13 + ( n + 1 ) % 13;
}
```

3.d.2 [pow] Jazyk C neposkytuje zabudovanou operaci mocnění pro celočíselné typy. V této ukázce naprogramujeme známý algoritmus binárního umocňování (anglicky známého popisněji jako „exponentiation by squaring“).[21] Klíčovou vlastností tohoto algoritmu je, že jeho složitost je lineární k **počtu bitů exponentu** – naivní algoritmus opakovaným násobením má naproti tomu složitost **exponenciální** (složitost je v tomto případě přímo úměrná **hodnotě** exponentu, nikoliv délce jeho zápisu).

Pro srovnání implementujeme obě standardní verze, jak rekurzivní (přímočará, ale méně efektivní) tak iterativní (efektivnější, ale méně průhledná a náchylnější na chyby implementace).

```
int pow_rec( int n, int exp )
{
```

Operaci budeme definovat pouze pro kladné exponenty. Vyhneme se tak mimo jiné nutnosti definovat hodnotu pro 0^0.

```
    assert( exp >= 1 );
    if ( exp == 1 )
        return n;
```

Výpočet je založený na pozorování, že pro sudý exponent k platí $n^k = n^{2l} = (n^2)^l$ kde $l = k/2$. Za cenu výpočtu jedné druhé mocniny – n^2 – tak můžeme exponent snížit na polovinu (rekurzivní zanoření se provede právě tolikrát, kolik bitů je v zápisu hodnoty exp).

Je-li exponent lichý, snížíme jej o jedničku. Všimněte si, že tento případ nemůže nastat dvakrát po sobě – díky tomu celkovou složitost nijak neohrozí (exponent se sníží na polovinu přinejhorším v každém druhém kroku).

Všimněte si také, že koncová rekurze se objevuje pouze v sudé větvi výpočtu – to přirozeně povede k drobným komplikacím při zápisu iterativní verze.

```
    if ( exp % 2 == 0 )
        return pow_rec( n * n, exp / 2 );
    else
        return n * pow_rec( n, exp - 1 );
}

int pow_iter( int n, int exp )
{
    assert( exp >= 1 );
```

Pomocná proměnná, která bude udržovat „liché“ mocniny. Její význam je přesněji vysvětlen níže.

```
    int odd = 1;
```

Oproti předchozí verzi nahradíme rekurzi cyklem – podmínka ukončení je stejná jako bázový případ rekurze.

```
    while ( exp > 1 )
    {
```

Musíme nyní zejména vyřešit situaci, kdy je exponent lichý. Zde je potřebný vztah trochu složitější: $n^k = n \cdot (n^{2l})$ pro $l = \lfloor k/2 \rfloor$.

Nyní nastane zmiňovaný drobný problém: faktor n před závorkou **nevstupuje** do výpočtu druhé mocniny v další iteraci (to je také důvod, proč tento případ nebyl v pow_rec koncovou rekurzí).

Asi nejjednodušším řešením je použití pomocného střadače, který bude udržovat tyto „přebývající“ faktory. Je-li exp liché, přinásobíme tedy faktor n do proměnné odd. Na konci ovšem nesmíme zapomenout, že ve výsledném n tyto faktory chybí.

```
        if ( exp % 2 == 1 )
            odd *= n;
```

Dále je již výpočet pro sudé i liché exponenty stejný: hodnotu proměnné n umocníme na druhou a exponent vydělíme dvěma.

```
        n *= n;
        exp /= 2;
    }
```

Na závěr si vzpomeneme, že některé faktory celkového výsledku jsme si „odložili“ do proměnné odd.

```
    return n * odd;
```

Pro ilustraci uvažme výpočet 3^{10}:

iterace	n	exp	odd
0.	3	10	1
1.	3·3	5	1
2.	(3·3)·(3·3)	2	3·3
3.	(3·3)·(3·3)·(3·3)·(3·3)	1	3·3

V proměnné n jsme sesbírali 8 faktorů, zatímco proměnná odd získala 2, celkem jich je tedy potřebných 10. Rekurzivní výpočet naproti tomu probíhá takto:

$$(3 \cdot (3 \cdot 3) \cdot (3 \cdot 3)) \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3))$$

Uvažme ještě výpočet 3^{11}. Je zejména důležité si uvědomit, že faktor, který na daném řádku přidáváme do odd (je-li exp na předchozím řádku liché) je právě hodnota n z tohoto předchozího řádku.

iterace	n	exp	odd
0.	3	11	1
1.	3·3	5	3
2.	(3·3)·(3·3)	2	3·(3·3)
3.	(3·3)·(3·3)·(3·3)·(3·3)	1	3·(3·3)

Stejný výpočet rekurzivně:

$$3 \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3)) \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3))$$

```
}

int main() /* demo */
{
    assert( pow_rec( 1, 2 ) == 1 );
    assert( pow_rec( 2, 1 ) == 2 );
    assert( pow_rec( 2, 2 ) == 4 );
    assert( pow_rec( 2, 3 ) == 8 );
    assert( pow_rec( 3, 2 ) == 9 );
```

[21] Popis algoritmu na české wikipedii je v době psaní tohoto textu zcela nepoužitelný. Podívejte se raději do té anglické.

```
    for ( int i = 0; i < 32; ++i )
        for ( int j = 1; j < ( i < 8 ? 5 : 4 ); ++j )
            assert( pow_rec( i, j ) == pow_iter( i, j ) );
}
```

3.p: Přípravy

3.p.1 **[cellular]** Napište čistou funkci, která simuluje jeden krok výpočtu jednorozměrného buněčného automatu (cellular automaton). Stav budeme reprezentovat celým číslem bez znaménka – jednotlivé buňky budou bity tohoto čísla. Stav osmibitového automatu by mohl vypadat například takto:

0	1	1	0	1	0	0	1
7	6	5	4	3	2	1	0

Pro zjednodušení použijeme pevnou sadu pravidel (+1, 0, -1 jsou relativní pozice bitů vůči tomu, který právě počítáme):

+1	0	-1	nová
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Pravidla určují, jakou hodnotu bude mít buňka v následujícím stavu, v závislosti na okolních buňkách stavu nynějšího. Na krajích stavu interpretujeme chybějící políčko jako nulu.

Výpočet s touto sadou pravidel tedy funguje takto:

1

0	1	1	0	0	0	1	0
0							

001 → 0

2

0	1	1	0	0	0	1	0
0	0						

011 → 0

3

0	1	1	0	0	0	1	0
0	0	1					

110 → 1

atd.

7

0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	

010 → 0

8

0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	1

100 → 1

Výpočet budeme provádět na 16bitových číslech (bez znaménka).

```
unsigned cellular_step( unsigned w );
```

3.p.2 **[permute]** Napište predikát is_permutation, který pro dvě celá čísla bez znaménka rozhodne, je-li možné jedno z čísel obdržet jako permutaci trojkového zápisu čísla druhého, jsou-li obě čísla zapsaná použitím jedenácti trojkových číslic (zleva doplníme nuly dle potřeby).

```
bool is_permutation( unsigned a, unsigned b );
```

3.p.3 **[rotate]** Binární rotace slova o jeden bit přesune nejvýznamnější bit na místo nejméně významného a všechny ostatní posune o jednu pozici doleva (směrem k významnějším).

Napište predikát, který rozhodne, je-li možné získat jedno z čísel jako binární rotaci druhého (o libovolný počet bitů).

```
bool is_rotation( unsigned a, unsigned b );
```

3.p.4 **[fractal]** Význačnou vlastností fraktální posloupnosti je, že má samu sebe jako vlastní podposloupnost. Jednoduchým příkladem takové posloupnosti je tzv. „trojúhelníková“ posloupnost

1,

1, 2,

1, 2, 3,

1, 2, 3, 4,

...

čteno po řádcích zleva doprava. Odstraníme-li poslední prvek každého „řádku“ (t.j. první výskyt daného čísla v posloupnosti), výsledkem bude opět tatáž posloupnost.

Podobnou fraktální posloupností je tzv. přirozená Fibonacciho fraktální posloupnost, kterou lze také uspořádat do řádků:

1,

1,

1, 2,

1, 2, 3,

1, 2, 3, 4, 5,

1, 2, 3, 4, 5, 6, 7, 8,

...

V tomto případě je délka řádku příslušné Fibonacciho číslo a n-tý řádek je posloupnost po sobě jdoucích celých čísel od 1 do fib(n).

Vaším úkolem je naprogramovat čistou funkci ffib, která vrátí n-tý prvek této druhé fraktální posloupnosti (číslováno od 1).

```
int ffib( int n );
```

3.p.5 **[palindrome]** Uvažme binární zápis celého čísla bez znaménka. Palindromem budeme nazývat takové číslo, které má při čtení zleva doprava stejné cifry, jako při čtení zprava doleva (nejsou-li levý a pravý zápis stejné délky, ten kratší doplníme levostrannými nulami tak, aby byl zápis stejně dlouhý).

Vaším úkolem je naprogramovat čistou funkci, která pro zadané celé číslo bez znaménka určí, jaký je nejmenší počet segmentů takových, že jejich připojením za sebe vznikne zadané číslo, a přitom každý segment je palindromem. Tyto segmenty nemusí být stejně dlouhé.

Poznámka: každé 16bitové číslo lze zapsat jako sekvenci 16 palindromů, každý délky jeden bit.

```
int palindrome_segments( unsigned number );
```

3.p.6 **[sat]** V tomto příkladu budeme pracovat s výrokovou logikou: Vaším úkolem bude rozhodnout, existuje-li pro danou formuli splňující přiřazení.

Formule bude zadaná čistou funkcí evaluate; tato vyhodnotí formuli číslo n pro valuaci zadanou hodnotou val – každý bit slova val odpovídá jedné volné proměnné – formule jich tak bude obsahovat vždy 16.

Navíc umožní evaluate určit pravdivost formule i v situaci, kdy nejsou určené všechny proměnné – bude totiž brát do úvahy pouze ty bity ve val, které jsou ve slově mask nastavené na jedna.

Výsledkem evaluate bude:

- 0 v případě, že formuli při daném val a mask není možné splnit,
- 1 v případě, že již splněná je (a na proměnných, které jsou v mask zakázané, již nezáleží),
- nebo -1 v případě, že výsledek závisí na bitech, které jsou v mask nulové.

```
int evaluate( unsigned n, unsigned mask, unsigned val );
```

Predikát sat rozhodne, je-li formule číslo n splnitelná (n zadává číslo formule podle číslování používaného funkcí evaluate). Vaše řešení musí fungovat i v případě, že funkce evaluate bude definovaná jinak.

```
bool sat( unsigned n );
```

3.r: Řešené úlohy

3.r.1 [power] Napište podprogram power, který provede modulární mocnění. Využijte rekurzivního binárního mocnění, které se opírá o vztahy:

$$a^{2n} = (a^2)^n$$

$$a^{2n+1} = a \cdot a^{2n}$$

Můžete předpokládat, že zadaný modul bude nanejvýš 256.

```
unsigned char power( unsigned char base, unsigned char exponent,
                     unsigned modulus );
```

3.r.2 [squares] Napište podprogram squares, který o zadaném čísle n rozhodne, kolika způsoby je toto možné zapsat jako součet nejvýše m druhých mocnin kladných čísel. Stejné sčítance v různém pořadí se přitom počítají pouze jednou.

```
unsigned squares( unsigned n, unsigned m );
```

3.r.3 [knight] Napište podprogram knight_hops, který spočítá jaký nejmenší počet skoků musí udělat kůň na standardní šachovnici 8×8, aby se dostal ze souřadnic (x1, y1) na souřadnice (x2, y2). Všechny souřadnice nabývají hodnot 0 až 7.

Nápověda: nejprve vyřešte problém „lze se ze startu do cíle dostat na nejvýše n skoků?“

```
unsigned knight_hops( char x1, char y1, char x2, char y2 );
```

3.r.4 [bezout] Naprogramujte podprogram bezout, který obdrží dvě celá čísla a a b a s využitím rozšířeného Euklidova algoritmu nalezne jejich koeficienty do Bézoutovy rovnosti, tj. celá čísla α a β taková, že

$$\alpha \cdot a + \beta \cdot b = gcd(a, b)$$

Dle obvyklé volací konvence očekávejte a v l1 a b v l2. Protože podprogram vrací dva výsledky, je potřeba volací konvenci ad-hoc rozšířit: α nechť je v rv, β v l1.

```
main:
    put  0 → t1
    jmp .main.loop
```

Řešení včetně návěští bezout můžete psát třeba sem.

Testovací data

```
data:
      a,    b  →  gcd( a, b )
    .word     1,    1,     1
    .word     2,    2,     2
    .word    18,   15,     3
    .word    20,   30,    10
    .word    -1
```

Zbytek souboru je implementace testů a můžete jej ignorovat.

```
.main.loop:
    ld   t1, data     → l1
    eq   l1, 0xffff   → t2
    jnz  t2, .main.leave
    ld   t1, data + 2 → l2

    push t1
    call bezout
    pop  t1

    .trigger set _tc_expect_ 4
    .trigger inc _tc_

    ld   t1, data     → l3
    ld   t1, data + 2 → l4
    ld   t1, data + 4 → l5

    mul  l3, rv → rv
    mul  l4, l1 → l1
    add  l1, rv → rv
    eq   rv, l5 → rv
    asrt rv

    add  t1, 6 → t1
    jmp  .main.loop
.main.leave:
    halt
```

Část 4: Adresy a ukazatele

Předmětem této kapitoly je zejména práce s ukazateli. Vztah mezi ukazatelem a adresou je podobný, jako mezi objektem a pamětí:

- podobně jako adresa je (strojové) **slovo**, ukazatel je **hodnota**,
- platný ukazatel **musí** „ukazovat“ na platný objekt,
- ukazatel je reprezentován, stejně jako adresa, celým číslem bez znaménka (o pevné bitové šířce dané architekturou stroje).

Podobně jako na většině architektur platí, že ne každé číslo je platná adresa, prakticky vždy platí, že existují platné adresy, které nejsou platnými ukazateli.

Ukázky:

1. divmod – výstupní parametry
2. xxx

Přípravy:

1. rand – náhodná čísla podruhé
2. zero – počet nul a index té nejlevější
3. xxx
4. xxx
5. xxx
6. xxx

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

4.1: Strojový kód

V této kapitole přidáváme operace pro práci s pamětí, konkrétně ld, ldb, st a stb. Tím popis strojového kódu uzavřeme, protože jsme již pokryli všechny existující instrukce – zbývající dva bloky na úrovni strojového kódu nic nového nepřinesou. Veškeré nové konstrukce jazyka C bude možné přeložit na již známou sadu instrukcí.

XXX

4.2: Programovací jazyk

Tato kapitola přidává ukazatele a (TODO!) operátor přetypování.

4.2.1 Definice V definici podprogramu umožníme, aby se na místě návratového typu objevilo klíčové slovo void, které značí, že výsledek vyhodnocení podprogramu **není hodnota**.

4.2.2 Hodnoty a typy V této kapitole přidáváme velmi důležitou novou třídu hodnot, a společně s ní příslušné typy. Je-li T libovolný typ, pak T * je typ „ukazatele na objekt typu T“. Hodnota typu T * je reprezentovaná celým číslem bez znaménka, které odpovídá **adrese**, na které je uložen příslušný objekt.[22]

Klíčové slovo void označuje (pseudo)typ void. Nejedná se o typ v běžném smyslu, protože neexistuje žádná hodnota typu void. Abychom odlišili „běžné“ typy (s existujícími hodnotami), zavedeme pojem **hodnotový typ**. Typ void je tak prvním typem, který hodnotový není.

Přesto, že nemůže existovat hodnota typu void, je možné napsat **výraz** tohoto typu. Takový výraz ale není možné ve většině případů použít jako podvýraz. Jediná místa, kde se **podvýraz** typu void objevit smí, jsou:

- libovolný operand operátoru čárka (nezávisle),
- e_1 a e_2 ve výrazu ternary = e_0 ? e_1 : e_2 – pak je typ výrazu ternary jako celku také void,[23]
- e_1 ve výrazu přetypování (void) e_1.

Je-li expr výraz typu void, může se také objevit v příkazu tvaru expr; (příkaz tvořený výrazem – v tomto případě se ovšem nejedná o podvýraz).

4.2.3 Výrazy Tato kapitola zavádí tři nové tvary výrazů. Abychom ale mohli tyto výrazy správně popsat, musíme nejprve upravit způsob, jakým uvažujeme o vyhodnocení výrazů a zavést několik nových pojmů.

Některé výrazy je možné **vyhodnotit na objekt** – prozatím se jedná pouze o výrazy tvaru jméno kde jméno pojmenovává nějakou proměnnou.[24] Vyhodnotí-li se nějaký výraz na objekt, další postup závisí na **kontextu** v jakém se objevuje:

1. Většina podvýrazů tvoří **r-kontexty**, tzn. takové, které očekávají hodnotu – např. operandy aritmetických nebo relačních operátorů, parametry předávané podprogramu, pravý operand operátoru přiřazení, atd. Je-li nějaký podvýraz vyhodnocen na objekt v **r-kontextu**, je z tohoto objektu navíc **přečtena hodnota** a výsledkem výrazu je až tato hodnota.
2. Některé podvýrazy jsou ale **l-kontextem** a v takovém případě se vyhodnocení zastaví určením **objektu** a čtení hodnoty ze získaného objektu se **neprovede**. Jedná se zejména o levý operand přiřazovacího operátoru (odtud také název **l-kontext**).

Objektu získanému vyhodnocením výrazu se říká také **l-hodnota** (chceme-li pak zdůraznit, že mluvíme o „normálních“ **hodnotách** a nikoliv **objektech**/**l-hodnotách**, můžeme použít také pojem **r-hodnota**).

Nyní můžeme konečně přistoupit k popisu nových tvarů výrazů:

1. Výraz *$expr_1$, tzv. **dereferenci**, lze použít pouze vyhodnotí-li se $expr_1$ na **platný ukazatel** (jedná-li se o ukazatel neplatný, program je chybný a jeho chování není určeno), a vyhodnotí se na **objekt** (l-hodnotu), který je tímto ukazatelem popsán. Je-li výraz $expr_1$ typu T *, je výraz *$expr_1$ typu T.
2. Výraz &$expr_1$ lze použít jen tehdy, vyhodnotí-li se $expr_1$ na **objekt** (je l-hodnotou) a výsledkem je **ukazatel** na tento objekt. Unární operátor & se anglicky nazývá „address of“, nicméně jeho výsledkem není striktně vzato adresa.[25] Je-li výraz $expr_1$ typu T, je výraz &$expr_1$ typu T *.
3. $expr_1$ = $expr_2$ (jedná se o zobecnění již zavedeného var = $expr_2$) lze použít, vyhodnotí-li se $expr_1$ na objekt (tzn. $expr_1$ popisuje l-hodnotu). Efektem tohoto výrazu je, že hodnota uložená v takto popsaném objektu se přepíše na hodnotu, kterou získáme vyhodnocením výrazu $expr_2$.

4.2.4 Příkazy Příkaz return v definici podprogramu bez návratové hodnoty (návratový typ je void) píšeme return; – nesmí se zde zejména objevit výraz.

4.d: Demonstrace (ukázky)

4.d.1 [divmod] Předmětem této ukázky je operátor získání adresy (unární &). Jedná se o konstrukci, která nám umožní pomyslně zafixovat adresu objektu – získáme-li tímto způsobem adresu nějakého objektu, překladač je nucen zabezpečit, aby program při čtení z této adresy vždy získal aktuální

[22] To neznamená, že objekt je s touto adresou pevně svázán – pouze to, že kdykoliv může být objekt **sekvenčně pozorován**, bude k nalezení na této adrese.

[23] Pravidla pro typy e_1 a e_2 zároveň vynucují, že musí být void oba nebo ani jeden.

[24] Na rozdíl od jazyka C++ výrazy tvaru var = e_1 ani tvaru e_0 ? e_1 : e_2 nikdy objekt nepopisují (nejsou l-hodnotami).

[25] Tuto zkratku si můžeme dovolit pouze proto, že použití operátoru & donutí překladač „zafixovat“ pozorovatelnou adresu objektu – každé použití & na objekt musí vrátit stejnou hodnotu a každá dereference takto získaného ukazatele musí, po dobu jeho platnosti, popisovat tento objekt.

hodnotu uloženou v adresovaném objektu.[26]

Jsme-li schopni získat adresu (vzpomeňte si, že se z pohledu výpočtu jedná o obyčejné číslo), můžeme s ní pracovat jako s každou jinou hodnotou. Zejména ji tedy můžeme předat jako parametr podprogramu – schopnost, kterou využijeme pro realizaci **výstupních parametrů**.

Adresy mají v jazyce C speciální typy – ukazatele. Hodnota typu ukazatel je sice číselná hodnota, ale sémanticky pro ni řada operací nedává smysl (nemá třeba smysl adresy násobit nebo sčítat mezi sebou). Naopak existují operace, které jsou smysluplné pouze pro adresy – zejména načtení hodnoty z adresy a zápis hodnoty na adresu.

```
int main() /* demo */
{
    // ...
}
```

4.p: Přípravy

4.p.1 [**rand**] V této přípravě budete opět generovat 32bitová pseudonáhodná čísla stejně jako ve druhém týdnu. Tentokrát však poloviny 32bitového výsledku nebudete vracet dvěma podprogramy, nýbrž s využitím vstupně-výstupních parametrů podprogramu jediného.

Vzorec pro generování dalšího čísla v řadě na základě předchozího vypadá stále takto:

$$\mathrm{rand}(\mathrm{prev}) = \mathrm{prev} \cdot 1103515245 + 12345$$

Napište podprogram rand, jehož dva parametry hi a lo ukazují na vyšší a nižší slovo počátečního čísla. Do těchto slov zapište vyšší a nižší slovo čísla následujícího.

```
void rand( unsigned *hi, unsigned *lo );
```

4.p.2 [**zero**] Napište podprogram zero_count, který spočte nulové bity v zadaném slově word. Druhý argument je výstupní – je-li předán nenulový ukazatel a slovo obsahuje nějaký nulový bit, zapište do čísla, na nějž ukazatel ukazuje, index nejlevější (tj. nejvýznamnější) nuly. Nejméně významný bit má jako obvykle index 0.

```
unsigned zero_count( unsigned word, unsigned *leftmost );
```

4.p.3 [**mode**]

4.p.4 [**digits**]

4.p.5 [**copy**]

4.p.6 [**collapse**]

[26] Toto omezení na překladač se netýká **souběžného** přístupu – platí jen pro sekvenční programy. Zejména tedy nadále platí, že kdykoliv není možné hodnotu objektu **sekvenčně** pozorovat, může být uložena jinde, než na takto získané adrese.

Část S.1: Model výpočtu

Tato sada obsahuje příklady zaměřené na práci s celými čísly a na zápis jednoduchých algoritmů a podprogramů (zejména čistých funkcí).

1. a_float – aritmetika s plovoucí desetinnou čárkou,
2. b_depth – práce s implicitním stromem,
3. c_tree – predikáty na stromech,
4. d_enum – číslování množin celých čísel,
5. e_xxx – ???
6. f_xxx – ???

Úlohu a byste měli zvládnout vyřešit hned po první přednášce. Příklady b, c a d vyžadují znalosti z nejvýše třetí přednášky (využívají podprogramy, případně i rekurzi).

S.1.a: float

Vaším úkolem je napsat program, který sečte dvě 16bitová čísla s plovoucí desetinnou čárkou ve formátu podobnému IEEE 754 binary16. Oproti skutečnému IEEE si práci v několika ohledech zjednodušíme:

- mantisa nebude používat implicitní bit, tzn. v nenulovém normalizovaném čísle bude mít vždy na nejvyšší pozici jedničku,
- nebudeme uvažovat subnormální čísla, nekonečna ani hodnoty NaN (vstupem jsou tedy vždy normalizovaná konečná čísla),
- budeme předpokládat, že nedojde k přetečení (tzn. výsledek je vždy možné reprezentovat),
- zaokrouhlovat budeme vždy pouze ořezáním, tzn. směrem k nule,
- prohodíme mantisu a exponent,[27] tzn. od nejvyššího bitu je nejprve znaménkový bit (1 = záporné, 0 = kladné), následuje 10 bitů mantisy a 5 bitů exponentu.

Výsledek sčítání nechť je také normalizovaný a je-li nulový, exponent i znaménkový bit budou také nulové.

Exponent je v aditivním kódování, ale protože jeho absolutní hodnota pro sčítání nehraje roli (důležitý je pouze rozdíl dvou exponentů), můžeme jej interpretovat jako pětibitové celé číslo bez znaménka.

Vstupní hodnoty jsou uloženy v registrech l1 a l2. Výsledek zapište do registru rv a skočte na návěstí check. Hodnotu v registru l7 neměňte.

V implementaci můžete použít následovný algoritmus:

1. vstupní hodnoty rozeberte na znaménkový bit, mantisu a exponent,
2. podle rozdílu exponentů zarovnejte mantisy „pod sebe“, tzn. tak, aby k-tý bit jedné i druhé mantisy odpovídal n-tému řádu; posuvy mantis provádějte tak, abyste zachovali 15 bitů přesnosti (přesto se v tomto kroku může stát, že některé bity vstupu ztratíme, např. při výpočtu $2^{280} + 2^{-8}$),
3. srovnejte znaménkové bity a zarovnané mantisy a podle potřeby proveďte součet nebo rozdíl a nastavte výsledný znaménkový bit (při sčítání nezapomeňte na případný přenos),
4. výsledek normalizujte (nejvyšší bit výsledné mantisy musí být jedna) – v tomto kroku dostanete výsledný exponent,
5. složte výsledek z vypočtené mantisy, exponentu a znaménkového bitu.

Příklad: uvažme vstupy x_1 = 0xc010, x_2 = 0x4011, hledáme $x_0 = x_1 + x_2$:

1. získáme znaménkové bity $s_1 = 1$, $s_2 = 0$,
2. mantisa m_1 = 0x200, m_2 stejně tak,
3. abychom dosáhli potřebné přesnosti, mantisy posuneme o 5 bitů doleva, tzn. dostaneme v obou případech 0x4000,
4. exponenty jsou kódovány jako e_1 = 0x10 a e_2 = 0x11 (tyto hodnoty značí „absolutní“ exponenty 1 a 2), jejich rozdíl je tedy 1 ve prospěch e_2 a jako prozatímní exponent výsledku si proto poznačíme $e_0 = e_2$,
5. mantisu m_1 posuneme o jeden bit doprava: $m_1' = 0x2000$,
6. protože znaménkové bity jsou různé, budeme odečítat – abychom předešli přetečení, odečítáme vždy menší číslo od většího, tzn. $m_0 = m_2 - m_1' = 0x4000 - 0x2000 = 0x2000$,
7. znaménkový bit výsledku bude jistě $s_0 = 0$,
8. první nenulový bit výsledné mantisy m_0 je na druhé nejvýznamnější pozici, ale normalizace vyžaduje, aby byl nenulový nejvyšší bit – proto m_0 posuneme o jednu pozici doleva a e_0 o jedničku snížíme,
9. mantisu ořízneme na výsledných 11 bitů přesnosti a z takto získaných s_0, m_0 a e_0 složíme hledané x_0.

Dobře si rozmyslete vnitřní reprezentaci jednotlivých částí (zejména mantisy). Formát čísel je navržen tak, aby se minimalizoval potřebný počet bitových posuvů.

S.1.b: depth

Uvažme implicitně zadaný binární strom, jehož každý uzel je popsaný 16bitovým číslem bez znaménka. Je-li v registru rv číslo nějakého uzlu, podprogram get_children do registrů l1 a l2 uloží číslo levého a pravého potomka. Hodnota nula znamená, že příslušný potomek neexistuje.

Vaším úkolem je napsat podprogram find_depth, který v registru rv obdrží číslo kořenového uzlu, a který zjistí maximální hloubku takto popsaného stromu. Výsledek uložte do registru l6 a řízení vraťte volající funkci.

Vaše řešení musí fungovat i v případě, že se konkrétní definice get_children změní (bude reprezentovat jiné stromy). Krom registrů l1 a l2 nejsou po volání get_children určeny hodnoty žádných dalších registrů.

S.1.c: tree

Opět budeme pracovat s implicitním stromem, tentokrát zadaným dvojicí čistých funkcí, tree_child_count a tree_get_child.[29] Stromy jsou popsané celočíselným identifikátorem (parametr tree_id) a obsahují uzly, které jsou popsané kladnými celými čísly (parametr node_id).

Funkce tree_get_child má navíc tuto vstupní podmínku:

- volání tree_get_child(t, n, i) je přípustné pouze
- platí-li i < tree_child_count(t, n).

Stromy jsou vždy neprázdné a kořen každého stromu má identifikátor 1.

```
int tree_child_count( int tree_id, int node_id );
int tree_get_child( int tree_id, int node_id, int index );
```

Vaším úkolem je naprogramovat predikát tree_is_binary, který rozhodne, je-li zadaný strom binární (každý uzel má 0, 1 nebo 2 potomky).

```
bool tree_is_binary( int tree_id );
```

Dále naprogramujte predikát tree_is_full, který rozhodne, je-li zadaný strom plný, tzn. existuje n takové, že každý uzel má buď n potomků nebo žádné potomky.

```
bool tree_is_full( int tree_id );
```

Konečně naprogramujte predikát tree_is_balanced, který rozhodne, je-li zadaný strom vyvážený, tzn. pro každý jeho vrchol platí, že rozdíl výšek nejvyššího a nejnižšího podstromu je nejvýše 1.

```
bool tree_is_balanced( int tree_id );
```

[27] Ve formátu IEEE je exponent ve vyšších bitech zejména kvůli jednoduchosti srovnání, to ale v tomto příkladu implementovat nebudeme.

[28] Pozor! Vaše řešení musí pracovat správně i v situaci, kdy se změní definice funkcí tree_child_count a tree_get_child (budou-li konzistentní se zadáním).

[29] Pro účely lexikografického srovnání dvou množin jejich prvky zapíšeme vzestupně, od nejmenšího prvku k největšímu.

S.1.d: enum

V tomto příkladu budeme pracovat s množinami přirozených čísel, tzn. množinami $M \subseteq \mathbb{N} = \{1, 2, 3, \ldots\}$. Tyto množiny lze souvisle očíslovat, např. tak, že je uspořádáme do tabulky a budeme je číslovat zleva doprava, shora dolů.

Tabulka bude v k-tém řádku obsahovat množiny, které obsahují k a případně čísla ostře menší, přitom nultý řádek obsahuje právě prázdnou množinu. Na řádku pak množiny uspořádáme vzestupně nejprve podle počtu prvků a pak lexikograficky.[30] Prvních několik řádků:

k	množiny
0	$\{\}$
1	$\{1\}$
2	$\{2\}, \{1,2\}$
3	$\{3\}, \{1,3\}, \{2,3\}, \{1,2,3\}$

Všimněte si, že žádná množina se v tabulce nebude opakovat. Podle algoritmu výše odečteme z tabulky množiny podle jejich indexů takto: $1 \to \{\}, 2 \to \{1\}, 3 \to \{2\}, 4 \to \{1,2\}, 5 \to \{3\}, 6 \to \{1,3\}, 7 \to \{2,3\}, 8 \to \{1,2,3\}$ atd.

Vaším úkolem je naprogramovat predikát is_member, který pro zadaný index i a přirozené číslo n rozhodne, zda je n prvkem i-té množiny.

```
bool is_member( unsigned i, unsigned n );
```

S.1.e: tiles

Uvažme obdélníkovou hrací plochu $m \times n$, kde jak m tak n jsou lichá, a ve které má každé pole přiřazenu bodovou hodnotu, a to takto:

- sudá pole v lichých řádcích a lichá pole v sudých řádcích mají nulovou hodnotu,
- rohová pole mají hodnotu 1,
- pole diagonálně sousední s rohovými mají hodnotu 2,
- každé další nenulové pole v první polovině řádku nebo sloupce má hodnotu o jedna vyšší než předchozí,
- zbytek se doplní symetricky.

Příklady hracích polí:

1	0	1
0	2	0
1	0	1

1	0	2	0	1
0	2	0	2	0
1	0	2	0	1

1
0
1

1	0	2	0	1
0	2	0	2	0
2	0	3	0	2
0	2	0	2	0
1	0	2	0	1

1	0	2	0	2	0	1
0	2	0	3	0	2	0
2	0	3	0	3	0	2
0	2	0	3	0	2	0
1	0	2	0	2	0	1

1	0	2	0	3	0	2	0	1

Na tomto poli budeme hrát jednoduchou hru: hráč dostane k kamenů rozměru $s \times 1$ a musí je umístit tak, aby se nepřekrývaly a zároveň součet hodnot zakrytých polí byl co největší. Hráč nemusí nutně použít všechny kameny, ale za každý nepoužitý je mu odečten bod. Kameny musí být položeny horizontálně.

Napište čistou funkci solve_tiles, které vstupem budou:

- rozměr hracího pole m (sloupce) a n (řádky),
- délka kamene s ,
- počet kamenů k,

a které výsledkem bude nejvyšší dosažitelný počet bodů.

```
int solve_tiles( int cols, int rows, int stone, int count );
```

S.1.f: mul

Napište podprogram mul, který vynásobí dvě base-256 čísla bez znaménka o libovolném počtu cifer. Čísla jsou v paměti zapsaná v pořadí MSB-first (na nejnižší adrese je nejvýznamnější slabika) od adresy a (resp. b) a jsou a_digits (resp. b_digits) slabik dlouhá.

Výsledek zapište do paměti od adresy result ve stejném formátu a počet cifer výsledku do uložte výstupního parametru result_digits. Výsledek nesmí obsahovat přebytečné levostranné nuly (samotná hodnota 0 má jednu cifru).

V podprogramu mul můžete využít a_digits + b_digits slabik paměti počínaje adresou result a samozřejmě lokální proměnné dle potřeby.

```
void mul( unsigned char *a, int a_digits,
unsigned char *b, int b_digits,
unsigned char *result, int *result_digits );
```

[30] Přesněji z pole, na jehož začátek ukazuje src.

Část 5: Pole

Předmětem této kapitoly je práce s poli a adresovatelnou pamětí obecně.

Ukázky:

1.

Elementární příklady:

1. copy – kopírování dat z pole do pole

Přípravy:

1. longest - nejdelší nepřerušený běh
2. memmem – výskyt zadané posloupnosti(?) bajtů
3. mode – nejčastější bajt
4. digits – rozklad/sklad na/z cifry/er
5. collapse – zkrácení opakovaných výskytů slova
6. erase – in-situ mazání z pole

Řešené příklady:

1. bsearch – binární vyhledávání
2. qsort – řazení rozdělováním
3. cyclic – hledání kružnic v grafu
4. closure – tranzitivní uzávěr
5. xxx
6. xxx

5.1: Programovací jazyk

Tato kapitola přináší **pole** – první složený typ, se kterým budeme v tomto kurzu pracovat. Speciální vlastností pole v jazyce C je, že neexistují **hodnoty** typu pole, pouze **objekty**. Protože se jedná o složené objekty, budou sestávat z **podobjektů** – jednotlivých položek. Všechny podobjekty (položky) daného pole jsou stejného typu.

5.1.1 Deklarace Pole vytvoříme podobně jako jiné objekty **deklarací proměnné**. Krom objektu tak vznikne jméno, kterým můžeme takto vytvořený objekt odkázat. Deklarace pole má tento tvar:

typ jméno[výraz];

Přitom typ může být libovolný dosud zavedený hodnotový typ (nemůže to tedy být pole, protože pole není hodnotový typ).

5.e: Elementární příklady

5.e.1 [copy] Napište podprogram copy_until, který zkopíruje až n slov z pole[31] src do pole dst. Kopírování přitom skončí dřív, pokud by se mělo kopírovat slovo s hodnotou delim; to se už nezkopíruje. Můžete předpokládat, že zdrojový a cílový kus paměti se nepřekrývají. Návratovou hodnotou je ukazatel bezprostředně **za** poslední zkopírované slovo (tj. do nebo za pole dst).

```
unsigned *copy_until( const unsigned *src, unsigned *dst,
                      unsigned n, unsigned delim );
```

5.p: Přípravy

5.p.1 [longest] Napište podprogram longest_run, který v zadaném poli čísel nalezne nejdelší nepřerušený běh opakujícího se čísla. Existuje-li nejdelších běhů více, podprogram vrátí ten s nejvyšší číselnou hodnotou.

Vstupem je dvojice ukazatelů haystack_begin a haystack_end které vymezují pole, v němž hledání proběhne (první ukazuje na začátek (pod)pole, ten druhý pak bezprostředně **za** jeho poslední prvek). Analogicky do výstupních parametrů needle_begin a needle_end zapište meze nalezené nejdelší posloupnosti.

```
void longest_run( int *haystack_begin, int *haystack_end,
                  int **needle_begin, int **needle_end );
```

5.p.2 [memmem] Napište podprogram memmem, který v poli haystack nalezne první výskyt podřetězce needle a vrátí ukazatel na jeho začátek. Pokud se jehla v kupce sena nenachází, vrátí NULL

```
char *memmem( char *haystack, int haystack_size,
              char *needle, int needle_size );
```

5.p.3 [mode] Implementujte podprogram mode, který spočítá modus (nejčastěji se vyskytující hodnotu) zadaného pole slabik. Je-li takových více, vrátí nejmenší z nich. Pole je podprogramu předáno jako ukazatel na jeho začátek a délka.

```
unsigned char mode( const unsigned char *array, unsigned length );
```

5.p.4 [digits] Napište dvojici podprogramů to_digits a from_digits, které zabezpečí rozklad čísla na číslice o daném základu nebo naopak sklad číslic na číslo. Můžete předpokládat, že základ bude nejméně dva.

Argument digits ukazuje na nejlevější z celkem size číslic.

```
unsigned from_digits( unsigned char base, unsigned char *digits,
                      unsigned size );
```

Výsledkem to_digits je počet číslic zapsaných do pole, na jehož začátek ukazuje digits. Číslice zapisujte do pole zleva. Můžete předpokládat, že pole bude dostatečně velké, aby se do něj vešly všechny číslice bez zbytečných levostranných nul. Je-li n nula, zapište právě jednu nulovou číslici.

```
unsigned to_digits( unsigned n, unsigned char base,
                    unsigned char *digits );
```

5.p.5 [collapse] Napište podprogram collapse, který obdrží ukazatel na začátek pole a jeho délku a toto pole na místě upraví tak, že všechny bezprostředně po sobě následující výskyty téhož slova nahradí výskytem jediným. Návratovou hodnotou bude ukazatel ukazující bezprostředně **za** poslední prvek takto upraveného pole.

```
unsigned *collapse( unsigned *array, unsigned length );
```

5.p.6 [erase] Napište podprogram erase, který z pole odstraní všechny výskyty nežádoucích vzorů – podřetězců bajtů. Jak vstupní pole, tak každý vzor bude zadán ukazatelem na svůj začátek. Jejich délka sice není předem známa, ale konec bude vždy rozpoznatelný tím, že poslední bajt bude mít nulovou hodnotu. Přirozeně pak tento nulový bajt nepovažujeme při srovnávání za součást dat ani vzorů. Obdobně není předem znám ani počet ukazatelů na vzory v poli non_grata a jeho konec bude signalizován nulovým ukazatelem.

Podprogram za nový konec zkráceného pole zapíše nulový bajt. Návratovou hodnotou je délka pole po promazání (bez závěrečného nulového bajtu).

V případě, že je některý nežádoucí vzor podřetězcem jiného, má při mazání přednost delší z nich. Vznikne-li se smazáním části pole podřetězec, který by opět vyhovoval některému nežádoucímu vzoru, k druhotnému mazání **nedojde**, ale pokračuje se s prohledáváním pole až za smazaným úsekem.

Vstupní pole i některý ze vzorů mohou být prázdné (v kterémžto případě ukazatel není nulový, nýbrž ukazuje přímo na nulový bajt). Seznam vzorů může být taktéž prázdný (tj. non_grata je ukazatel ukazující na nulový ukazatel). Ve vhodně navrženém řešení však tyto případy není zapotřebí ošetřovat zvlášť.

```
int erase( char *data, const char * const *non_grata );
```

5.r: Řešené úlohy

5.r.1 [bsearch] Napište podprogram bsearch, který binárním vyhledáváním nalezne první výskyt hodnoty value v zadaném vzestupně seřazeném poli.

[31] Složky můžeme sdružovat podle typu, nicméně pozor na ukazatele – stejně jako u lokálních proměnných, deklarace položek int *x, y; zavádí položku x typu int * a položku typu y typu int. Dvě položky typu int * zapisujeme jako int *x, *y;.

Pakliže se hodnota v poli nevyskytuje, vrátí podprogram nulový ukazatel.

```
int *bsearch( int value, int *array, int size );
```

5.r.2 [qsort] Napište podprogram sort, který na místě seřadí zadané pole čísel. Použijte algoritmus řazení rozdělováním (známý jako quicksort).

```
void sort( int *array, int size );
```

5.r.3 [cyclic] Napište predikát is_cyclic, který o zadaném orientovaném grafu rozhodne, zda obsahuje kružnici. Zadaný graf nemusí být souvislý.

Vrcholy grafu jsou reprezentovány přirozenými čísly. Ke každému vrcholu bude zadán seznam následníků – čísla vrcholů, do nichž ze zadaného vrcholu vede hrana. Predikát obdrží ukazatel na pole ukazatelů na seznamy následníků. Jejich délky ani počet vrcholů není předem znám, za posledním seznamem následníků ale bude nulový ukazatel a v každém seznamu následníků bude konec označen záporným číslem (které nemůže označovat žádný vrchol).

Kromě popisu grafu predikát obdrží i ukazatel na bajtové pole o délce odpovídající počtu vrcholů grafu. Tuto paměť můžete použít libovolně jako pracovní.

```
bool is_cyclic( const int * const *neighs, char *scratch );
```

5.r.4 [closure] Napište podprogram make_transitive, který na místě doplní zadanou relaci na její tranzitivní uzávěr.

Podprogram obdrží velikost n nosné množiny a n^2 bitů reprezentujících relaci: prvek i je v relaci s j právě tehdy, když je bit číslo $n \cdot i + j$ nastaven na jedničku. Bity jsou v předaném poli bajtů indexovány od nejméně významného a v pořadí little endian; například tedy nejvýznamnější bit druhého bajtu nese index 15.

```
void make_transitive( int n, unsigned char *r );
```

Část 6: Struktury, zřetězený seznam

V této kapitole se budeme zabývat dalším složeným objektem – **strukturou** (záznamovým typem). Na rozdíl od pole jsou struktury navíc **složenými hodnotami**. Struktury a pole jsou příbuzné, nicméně jsou mezi nimi také klíčové rozdíly:

- složky (podobjekty, podhodnoty) struktury není možné indexovat čísly (nejsou adresovatelné) ale je možné je pojmenovat,
- jednotlivé složky struktury nemusí mít stejný typ (struktura je typově heterogenní, zatímco pole je homogenní – každý prvek pole je stejného typu).

Hodnoty (a objekty) s pojmenovanými složkami potenciálně různých typů jsou v programování velice užitečné, protože umožňují stavbu **datových abstrakcí**. Hrají tak podobnou roli při práci s daty, jako podprogramy při stavbě **výpočetních abstrakcí**.

Ukázky:

1. insert – vložení prvku do zřetězeného seznamu
2. dup – duplikace zřetězeného seznamu

Přípravy:

1. erase – odstranění prvků ze zřetězeného seznamu
2. link – vytvoření zřetězeného seznamu z pole
3. select – výběr podseznamu podle indexů
4. recycle – znovupoužití odstraněných uzlů
5. isort – řazení zřetězeného seznamu vkládáním
6. vsort – řazení pole klíčů proměnné délky

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

6.1: Programovací jazyk

Tato kapitola přináší novou třídu složených typů – struktury, neboli záznamové typy.

6.1.1 Definice Definice struktury má tento tvar:

```
struct jméno0
{
    typ1 jméno1;
    typ2 jméno2;
    …
    typn jménon;
};
```

Každé jméno uvnitř definice ($jméno_1$ až $jméno_n$) definuje **složku** struktury odpovídajícího typu.[32]

Konečně $jméno_0$ je jménem struktury (pozor, není totéž jako jméno typu!). Definujeme-li strukturu foo, na místech, kde je očekáván typ, můžeme psát struct foo.

6.1.2 Výrazy Se strukturami souvisí také dva nové tvary výrazů:

- $expr_1$.jméno – přístup ke složce; typ výrazu $expr_1$ musí být struktura, která má složku pojmenovanou jméno. Vyhodnotí-li se $expr_1$ na objekt, výraz jako celek pojmenovává příslušný podobjekt (a může tedy stát na levé straně přiřazení).
- $expr_1$->jméno – nepřímý přístup ke složce skrze ukazatel – podobně jako dereference, vstupní podmínkou je, že $expr_1$ je **platný** ukazatel. Výsledkem je vždy objekt (l-hodnota).

6.p: Přípravy

6.p.1 [erase] Napište podprogram erase, který ze zadaného zřetězeného seznamu odstraní zadaný uzel. Seznam je předán ukazatelem na první uzel; tento může být i nulový. Návratovou hodnotou je ukazatel na první uzel promazaného seznamu, případně nulový ukazatel, je-li po smazání uzlu seznam prázdný. Není-li uzel součástí seznamu, podprogram erase žádnou změnu neprovede.

```
struct node
{
    int value;
    struct node *next;
};

struct node *erase( struct node *head, struct node *to_erase );
```

6.p.2 [link] Napište proceduru link, která ze vstupního pole celých čísel array vytvoří v zadané paměti zřetězený seznam. Uzly alokujte v paměti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

Výsledkem procedury link bude ukazatel na hlavu vytvořeného seznamu, nebo bude nulový je-li výsledný seznam prázdný. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit při tvorbě seznamu.

```
struct node
{
    int value;
    struct node *next;
};

struct heap
{
    char *start, *end;
};

struct node *link( struct heap *heap, int *array, int array_len );
```

6.p.3 [select] Napište proceduru select, která rozdělí stávající zřetězený seznam na dva, a to tak, že:

- ze vstupního zřetězeného seznamu linked vyřadí všechny prvky krom vybraných,
- z těchto vyřazených prvků vytvoří nový zřetězený seznam rejected.

Výběr prvků provede podle vzestupně seřazeného pole indexů indices o indices_count prvcích. Jinak řečeno, je-li idx prvek pole indices, idx-tý prvek vstupního seznamu linked bude zachován, jinak bude odstraněn.

Procedura nechť provede pouze jeden průchod vstupním seznamem linked i vstupním polem indices. Návratová hodnota nechť je ukazatel na hlavu seznamu rejected, nebo nulový ukazatel je-li tento seznam prázdný.

```
struct node
{
    int value;
    struct node *next;
};

struct node *select( struct node *linked,
                     int *indices, int indices_count );
```

6.p.4 [recycle] Napište proceduru link, která ze vstupního pole celých čísel array vytvoří v zadané paměti zřetězený seznam. Pro nové uzly využijte paměť určenou strukturou heap:

- přednostně využijte uzly ze seznamu recycle,
- až když je tento seznam prázdný, alokujte nové uzly ze začátku oblasti vymezené ukazateli start a end (přitom end ukazuje těsně za konec

[32] Hodnoty v uzlech se nesmí změnit, ukazatele na následníka však ano.

dostupné paměti).

Výsledkem procedury link bude ukazatel na hlavu vytvořeného seznamu, nebo bude nulový ukazatel je-li výsledný seznam prázdný. Procedura zároveň upraví strukturu heap tak, aby:

- ukazatel start ukazoval na první bajt paměti, který nebyl využit při tvorbě seznamu a
- ukazatel recycle tak, aby ukazoval na první nepoužitý uzel určený k recyklaci.

Poznámka: je-li po skončení link v recycle nenulový ukazatel, ukazatel start se nemění.

```
struct node
{
    int value;
    struct node *next;
};

struct heap
{
    char *start, *end;
    struct node *recycle;
};

struct node *link( struct heap *heap, int *array, int array_len );
```

6.p.5 **[isort]** Napište podprogram sort, který vzestupně seřadí uzly v zadaném zřetězeném seznamu a vrátí ukazatel na uzel s nejmenší hodnotou.

Použijte řazení vkládáním: každý prvek je vložen na správné místo do již seřazené části seznamu nalevo od své původní pozice. Řaďte změnou ukazatelů next, nikoli hodnot value.

```
struct node
{
    const int value;
    struct node *next;
};

struct node *sort( struct node *head );
```

6.p.6 **[vsort]** Vaším úkolem je naprogramovat proceduru vsort, která na místě seřadí zadané pole. Prvky tohoto pole jsou řetězce bajtů reprezentované strukturou key definovanou takto:

```
struct key
{
    unsigned char *data;
    int size;
};
```

Řazení proveďte lexikograficky. Pole je proceduře vsort předáno jako ukazatel na začátek array a počet prvků size. Použijte vhodný algoritmus (může být kvadratický, ale nějaký příčetný).

```
void vsort( struct key *array, int size );
```

6.r: Řešené úlohy

6.r.1 **[rle]** Run-length encoding je způsob kódování dat, který každou sekvenci opakující se hodnoty reprezentuje touto hodnotou doplněnou o informaci o počtu opakování. Jednu takovou dvojici hodnoty a počtu budeme reprezentovat strukturou rle_item:

```
struct rle_item
{
    int value;
    unsigned count;
};
```

Vaším úkolem bude vstupní posloupnost již zakódovaných dat překódovat do **kanonického tvaru**. V tomto tvaru se nevyskytují

- dvě položky se stejnou hodnotou vedle sebe a
- položky s nulovým počtem opakování.

Podprogram rle_canonize tuto konverzi provede na místě. Vstupním argumentem je pole kódovacích položek zadané ukazatelem na začátek a délkou (tj. počtem položek), návratovou hodnotou je nový počet položek po překódování.

```
int rle_canonize( struct rle_item *begin, int length );
```

6.r.2 **[msort]** Napište podprogram sort, který vzestupně seřadí uzly v zadaném zřetězeném seznamu a vrátí ukazatel na uzel s nejmenší hodnotou.

Použijte řazení sléváním (merge sort): seznam se rozdělí vedví, jeho poloviny se rekurzivně seřadí a následně se slijí tak, aby uzly byly ve správném pořadí.

```
struct node
{
    const int value;
    struct node *next;
};

struct node *sort( struct node *head );
```

6.r.3 **[lasso]** Zřetězený seznam nazveme **lasem**, pokud žádný ukazatel na následníka není nulový a část uzlů tak tvoří cyklus. Napište predikát is_lasso, který o zadaném zřetězeném seznamu rozhodne, zda se jedná o laso. Smíte přitom použít jen konstantní množství paměti.

```
struct node
{
    int value;
    struct node *next;
};

bool is_lasso( struct node *head );
```

6.r.4 **[length]** Napište podprogram length, který zjistí, kolik je ve zřetězeném seznamu uzlů, a to i tehdy, je-li seznam lasem (tj. jeho poslední uzel má jako následníka nějaký předchozí uzel, který se již v seznamu vyskytl). Smíte přitom použít jen konstantní množství paměti.

Tip: využijte algoritmu „želva a zajíc“ – dva ukazatele procházejí seznam různou rychlostí a nutně se musí potkat někde na cyklu. Délka cyklu se zjistí snadno, délka necyklického prefixu (do prvního opakujícího se uzlu) je těžší: když se spustí stejně rychlý průchod ze začátku seznamu i z místa setkání želvy a zajíce, střetnou se tyto dva ukazatele přesně v prvním opakujícím se uzlu.

```
struct node
{
    int value;
    struct node *next;
};

int length( struct node *head );
```

6.r.5 **[out]** Kruhovým seznamem je takový zřetězený seznam, jehož poslední uzel obsahuje místo nulového ukazatele odkaz zpět na první uzel.

Napište podprogram count_out, který ze zadaného kruhového seznamu destruktivně[33] vybere jeden uzel pomocí „rozpočitadla“: vybere se n-tý uzel (počítáno od nuly), ten se ze seznamu odstraní a stejným způsobem se s vyřazováním pokračuje od uzlu, který byl následníkem toho právě odstraněného. Až bude seznam tvořen jediným uzlem (který má za následníka sám sebe), podprogram tento uzel vrátí.

Podprogram by měl efektivně fungovat i pro n mnohem větší než je počet uzlů v seznamu.

```
struct node
{
    const int value;
    struct node *next;
};

struct node *count_out( struct node *head, unsigned n );
```

[33] 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.

Ukázky:

1. xxx

Přípravy:

1. rldec – dekodér RLE (run-length encoding)
2. rlenc – výpočet RLE
3. join – spojení několika polí do jednoho
4. split – dělení pole podle oddělovače
5. tree – konstrukce binárního stromu
6. sparse – převod matic do řídké reprezentace

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

7.p: Přípravy

7.p.1 **[rldec]** Napište podprogram rle_decode, který rozbalí vstupní pole dvojic ⟨hodnota, počet výskytů⟩ tak, že každou hodnotu nahradí příslušně mnohokrát opakovanou hodnotou. Každá taková dvojice je reprezentována hodnotou typu rle_item:

```
struct rle_item
{
    int value;
    unsigned count;
};
```

Výsledné pole hodnot umístěte na začátek paměti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je dvojice reprezentující pole hodnot: ukazatel na první a jejich počet. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole.

Nestačí-li zadaná paměť na uložení celých dat, bude ukazatel na výsledek nulový a ukazatel start se nezmění. Je-li výsledek prázdný, ukazatel na výsledek nulový nebude, délka však ano.

```
struct values
{
    int *data;
    int length;
};

struct values rle_decode( struct heap *heap, struct rle_item *items,
                          int length );
```

7.p.2 **[rlenc]** Napište podprogram rle_encode, který zakóduje vstupní pole dat tak, že každý běh opakující se hodnoty nahradí dvojicí ⟨hodnota, počet výskytů⟩. Každá taková dvojice je reprezentována hodnotou typu rle_item:

```
struct rle_item
{
    int value;
    unsigned count;
};
```

Výsledné pole kódovacích položek rle_item umístěte na začátek paměti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je dvojice reprezentující pole kódovacích položek: ukazatel na první a jejich počet. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole.

Nestačí-li zadaná paměť na uložení všech položek, bude ukazatel na výsledek nulový a ukazatel start se nezmění. Je-li výsledek prázdný, ukazatel na výsledek nulový nebude, délka však ano.

```
struct rle
{
    struct rle_item *items;
    int length;
};

struct rle rle_encode( struct heap *heap, int *data, int length );
```

7.p.3 **[join]** Napište podprogram join, který z několika vstupních polí vytvoří jedno souvislé pole. Vstupní pole budou zadána jako pole struktur span definovaných takto:

```
struct span
{
    unsigned *start, *end;
};
```

Paměť potřebnou pro spojené pole odeberte ze začátku oblasti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je opět struktura span, která popisuje nově vytvořené souvislé pole. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole. Nestačí-li dodaná paměť na uložení celého pole, budou oba ukazatele v návratové hodnotě nulové a ukazatel start v předané struktuře heap se nezmění.

```
struct span join( struct heap *heap,
                  struct span *arrays, int array_count );
```

7.p.4 **[split]** Napište podprogram split, který ve vstupním poli nalezne hranice úseků oddělených specifickou hodnotou (oddělovačem). Vstupem je pole slov a hodnota oddělovače, výstupem je pole úseků reprezentovaných strukturou span:

```
struct span
{
    unsigned *start, *end;
};
```

Paměť potřebnou pro pole úseků odeberte ze začátku oblasti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je struktura split_result. Ve výsledném poli spans nechť je vždy $n+1$ položek, kde n je počet výskytů oddělovače ve vstupním poli (některé rozsahy mohou být prázdné – rozmyslete si, v jakých případech tomu tak bude).

```
struct split_result
{
    int count;
    struct span *spans;
};
```

Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole. Nestačí-li dodaná paměť na uložení celého pole, bude ukazatel spans v návratové hodnotě nulový a ukazatel start se nezmění.

```
struct split_result split( struct heap *heap,
                           unsigned *array, int array_size,
                           unsigned delimiter );
```

7.p.5 **[tree]** Napište podprogram make_tree, který převede seřazené pole celých čísel na binární vyhledávací strom. Výstupní strom bude reprezentován takto:

- vnitřní uzly obsahují:
  - dva ukazatele na potomky,
  - jednobajtový příznak type, který určuje zda jsou tito potomci opět vnitřní uzly, nebo listy,
- listy obsahují pouze hodnotu (pro uložení listu jsou potřeba pouze dva bajty).

Vnitřní uzel bude reprezentován touto strukturou:

```
struct node
{
    unsigned char type;
    void *children[ 2 ];
};
```

Ukazatele children nemají konkrétní typ, protože tento v době překladu není znám – může to být int * je-li příslušný potomek listem, nebo struct node * je-li vnitřním uzlem.

Skutečný typ ukazatele se zjistí podle hodnoty type:

- hodnota 0 znamená, že jsou oba typu int *,
- hodnota 1 znamená, že první je typu struct node * a druhý typu int *,
- hodnota 2 naopak, a konečně,
- hodnota 3 znamená, že jsou oba typu struct node *.

Strom konstruujte následovně:

1. pole rozdělíme na stejně velké části (je-li liché délky, prostřední prvek umístíme do levé části),
2. alokujeme vnitřní uzel a jeho potomky sestrojíme rekurzivně:
   - má-li příslušné pole jediný prvek, potomkem je list a tento bude uložen přímo v původním poli (tzn. pro listy nealokujeme žádnou novou paměť),
   - je-li příslušné pole prázdné, potomek bude nulový ukazatel na list,
   - jinak alokujeme a sestrojíme vnitřní uzel podle bodu 1.

Veškerou paměť pro vnitřní uzly odeberte ze začátku oblasti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je ukazatel na kořen stromu. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit.

Nestačí-li zadaná paměť na uložení celé reprezentace, make_tree vrátí nulový ukazatel a ukazatel start se nezmění.

```
struct node *make_tree( struct heap *heap,
                        int *array, int array_size );
```

7.p.6 **[sparse]** Napište podprogram make_sparse, který převede běžnou reprezentaci matice (pole čísel o $n \cdot m$ prvcích) na řídkou (sparse) reprezentaci. Tato reprezentace bude složená z:

- hodnot n a m (počet sloupců a řádků),
- pole rows, které obsahuje m dvojic (ukazatel, délka m_k),
- pro každý řádek pole o m_k položkách:
  - položka může reprezentovat blok nul, nebo
  - jednu nenulovou hodnotu.

Vstupní matice obsahuje celá čísla bez znaménka v rozsahu $\langle 0, 65520)$. Řídká reprezentace bude používat čísla z rozsahu $\langle 65520, 65536)$ pro kódování bloků nul – počet nul získáme odečtením hodnoty 65519 (prázdné bloky nul reprezentovat nepotřebujeme). Je-li blok nul delší než 16, bude reprezentován dvěma nebo více položkami.

Vstupní matice je reprezentovaná touto strukturou:

```
struct dense_matrix
{
    unsigned cols, rows;
    unsigned *data;
};
```

Výstupní matice bude reprezentovaná takto:

```
struct sparse_row
{
    unsigned *data;
    unsigned size;
};

struct sparse_matrix
{
    unsigned cols, rows;
    struct sparse_row *row_data;
};
```

Veškerou paměť potřebnou pro výslednou řídkou matici odeberte ze začátku oblasti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je výsledná řídká reprezentace. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole.

Nestačí-li zadaná paměť na uložení celé reprezentace, ukazatel row_data ve výsledné sparse_matrix bude nulový a ukazatel start se nezmění.

```
struct sparse_matrix make_sparse( struct heap *heap,
                                  struct dense_matrix matrix );
```

7.r: Řešené úlohy

7.r.1 **[msort]** Napište podprogram sort, který vzestupně seřadí hodnoty v zadaném poli. K dispozici má pomocnou paměť o velikosti dvojnásobku vstupního pole.

Použijte řazení sléváním (merge sort).

```
void sort( int *data, int length, int *scratch );
```

7.r.2 **[inorder]** Napište podprogram inorder, který zkopíruje hodnoty ze stromu (zadaného kořenem) do nového pole v pořadí od nejlevější po nejpravější.

Strom bude mít hodnotu v každém uzlu. Uzel bude reprezentován následující strukturou:

```
struct node
{
```

```
    int value;
    struct node *left,
                *right;
};
```

Ukazatele left a right mohou být pochopitelně nulové. Také ukazatel na kořen může být nulový; ten reprezentuje prázdný strom. Nic jiného o struktuře stromu nepředpokládejte.

Výsledné pole hodnot umístěte na začátek paměti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    char *start, *end;
};
```

Návratovou hodnotou je dvojice reprezentující pole hodnot: ukazatel na první a jejich počet. Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole.

Nestačí-li zadaná paměť na uložení celých dat, bude ukazatel na výsledek nulový a ukazatel start se nezmění. Je-li výsledek prázdný, ukazatel na výsledek nulový nebude, délka však ano.

```
struct values
{
    int *data;
    int length;
};

struct values inorder( struct heap *heap, struct node *root );
```

Část 8: Recyklace paměti

Objekty jsou často užitečné jen po omezenou dobu.

Ukázky:

1. xxx

Přípravy:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

8.p: Přípravy

8.p.1 **[join]** Napište podprogram join, který ze zřetězeného seznamu polí vytvoří jedno souvislé pole. Vstupní pole budou zadána jako seznam struktur node definovaných takto:

```
struct node
{
    struct node *next;
    int size;
};
```

Pole hodnot typu unsigned char velikosti size bude uloženo bezprostředně za koncem struktury node.

Paměť potřebnou pro spojené pole odeberte ze začátku oblasti určené strukturou heap (tzn. vymezené ukazateli start a end, přitom end ukazuje těsně za konec dostupné paměti).

```
struct heap
{
    unsigned char *start, *end;
};
```

```
struct span
{
    unsigned char *start, *end;
};
```

Návratovou hodnotou je struktura span, která popisuje nově vytvořené souvislé pole. Má-li výsledné pole nulovou délku, alokujte pro něj jeden bajt paměti.

Procedura zároveň upraví ukazatel start v předané struktuře heap tak, aby ukazoval na první bajt paměti, který nebyl využit pro uložení pole. Nestačí-li dodaná paměť na uložení celého pole, budou oba ukazatele v návratové hodnotě nulové a ukazatel start v předané struktuře heap se nezmění.

```
struct span join( struct heap *heap, struct node *list );
```

8.p.2 **[recycle]** Napište podprogram recycle, který ověří, je-li zadaný zřetězený seznam alokován souvisle a těsně před začátkem dané oblasti paměti, a pokud ano, tuto paměť s ním spojenou vrátí předané (upravením struktury heap). V opačném případě žádnou změnu neprovede a vrátí false.

Bez ohledu na výsledek můžete seznam určený k recyklaci libovolně měnit. Z paměti určené předanou strukturou heap můžete navíc **dočasně** využít až n bitů paměti, kde n je délka předaného seznamu. Zamyslete se ale, zda (a za jakou cenu) lze úlohu řešit bez této dodatečné paměti.

```
struct node
{
    struct node *next;
    int value;
};

struct heap
{
    unsigned char *start, *end;
};

bool recycle( struct heap *heap, struct node *list );
```

8.p.3 **[coalesce]** Napište podprogram coalesce, který v zřetězeném seznamu bloků paměti (freelist) sloučí všechny těsně sousedící bloky do jediného uzlu. Blok začíná adresou samotného uzlu, který je definovaný takto:

```
struct node
{
    struct node *next;
    int size; /* including the node itself */
};
```

Složka size je celková velikost bloku (včetně struktury node která je uložena vždy na začátku bloku). Návratová hodnota je nová hlava seznamu. Uzly nechť jsou ve výstupním seznamu seřazeny vzestupně podle adresy a pro každou dvojici uzlů bude platit (char *) n->next > (char *) n + n->size. Složitost může být nejvýše kvadratická, ale není těžké problém vyřešit ani v čase $O(n \cdot log n)$.

Rozmyslete si také, co se bude dít s uzly, které byly ze seznamu vyřazeny.

```
struct node *coalesce( struct node *list );
```

8.p.4 **[fit]** Uvažme situaci, kdy máme k dispozici zřetězený seznam volných bloků paměti, přitom tento seznam je vzestupně seřazený podle velikosti bloku. Navíc disponujeme polem 13 ukazatelů, kde i-tý prvek odkazuje první uzel o velikosti 2^{i+2} nebo větší. Není-li tak velký blok k dispozici, ukazatel je nulový.

Vaším úkolem je naprogramovat čistou funkci first_fit, která co nejefektivněji najde nejmenší blok takový, že pojme alokaci o zadaném počtu bajtů. Není-li takový k dispozici, vrátí nulový ukazatel.

```
struct node
{
    struct node *next;
    int size;
};

struct node *first_fit( struct node *list, struct node **jumps,
                        unsigned size );
```

8.p.5 **[small]** Představte si situaci, kdy potřebujeme efektivně dynamicky alokovat mnoho malých objektů – velikosti 4 až 16 bajtů. Paměť budeme organizovat takto:

- na nejvyšší úrovni je paměť rozdělena do bloků o 512 bajtech, přitom každý blok disponuje 20bajtovou hlavičkou a řadou „přihrádek“ stejné velikosti,
- hlavička je tvořena ukazatelem na další blok stejné velikosti a informacemi o volných přihrádkách (bitmapou a počtem).

Hlavička je popsaná následující strukturou, přitom nejnižší bit první slabiky atributu bitmap popisuje přihrádku na nejnižší adrese (začátek bloku + 20 bajtů). Jednička v bitmapě znamená, že přihrádka je volná. Atribut free_slots udržuje celkový počet volných přihrádek v bloku.

```
struct header
{
```

```
    struct header *next;
    unsigned free_slots;
    unsigned char bitmap[ 16 ];
};
```

Vaším úkolem je naprogramovat čistou funkci find_free, která vrátí adresu volného místa v paměti o velikosti alespoň size bajtů. Není-li k dispozici přihrádka o nejmenší použitelné velikosti, výsledkem bude nulový ukazatel.

Parametr heads je pole 7 ukazatelů na hlavy seznamů pro příslušné velikosti – i-tý prvek je hlava seznamu bloků s přihrádkami o velikosti $2i + 4$ bajtů.

```
unsigned char *find_free( struct header **heads, unsigned size );
```

8.p.6 [free] Uvažme organizaci paměti stejnou jako v předchozí úloze – alokujeme malé objekty velikosti 4 až 16 bajtů, přitom:

- na nejvyšší úrovni je paměť rozdělena do bloků o 512 bajtech, kde každý blok disponuje 20bajtovou hlavičkou a řadou „přihrádek“ stejné velikosti,
- hlavička je tvořena ukazatelem na další blok stejné velikosti a informacemi o volných přihrádkách (bitmapou a počtem).

Hlavička je popsaná následující strukturou, přitom nejnižší bit první slabiky atributu bitmap popisuje přihrádku na nejnižší adrese (začátek bloku + 20 bajtů). Jednička v bitmapě znamená, že přihrádka je volná. Atribut free_slots udržuje celkový počet volných přihrádek v bloku.

```
struct header
{
    struct header *next;
    unsigned free_slots;
    unsigned char bitmap[ 16 ];
};
```

Vaším úkolem je naprogramovat proceduru mark_free, která obdrží adresu address nějaké přihrádky a tuto přihrádku označí jako volnou. Vstupní podmínkou je, že adresa patří alokované přihrádce správné velikosti.

Velikost size je skutečná velikost přihrádky, bez ohledu na to, pro jak velký objekt byla tato alokovaná. Adresa base určuje začátek paměti, ve které jsou bloky uložené. Bloky jsou od base vzdálené vždy o celočíselné násobky 512 (tzn. v rámci nějaké větší oblasti jsou jejich relativní adresy dělitelné 512).

```
void mark_free( unsigned char *base,
                unsigned char *address,
                unsigned size );
```

1. a_race – jednoduchá hra s figurkami a kostkou
2. b_permute – hledání a aplikace permutací
3. c_sched – jednoduchý plánovač vláken
4. d_heap – dynamická alokace paměti
5. e_tinier – zmenšená verze stroje tiny
6. f_eval – velmi malý interpret výrazů

S.2.a: race

Uvažme jednoduchou hru, ve které má každý hráč jednu figurku. Figurky se budou pohybovat po hracím plánu složeném z políček, přitom každé políčko je popsané následující strukturou:

```
struct location
{
    struct location *next;
    struct location *branch;
};
```

Je-li next nastaveno na nulový ukazatel, jedná se o cílové políčko (může být takových v plánku víc). Dorazí-li figurka na cílové políčko, hra končí a její vlastník vyhrává.

Je-li branch nenulový ukazatel, z políčka je možné odbočit. Stojí-li při hodu kostkou figurka na políčku s odbočkou, hráč se může rozhodnout, zda bude pokračovat rovně (směrem na next) nebo odbočí. Figurka poté provede příslušný počet kroků (první podle hráčova rozhodnutí, zbytek vždy na políčko next).

Průběh hry popisuje pole struktur step definovaných takto:

```
struct step
{
    int die;      /* the value that came up on the die */
    bool branch; /* the player chose to follow the branch */
};
```

Je-li branch nastaveno na true, v daném kroku se hráč rozhodl použít odbočku (není-li to v dané situaci možné, taková hra jistě proběhnout nemohla). Podobně není dovoleno provádět žádné kroky poté, co některý hráč vyhrál.

Hráči se v házení kostkou střídají, začíná hráč číslo 1. Padne-li na kostce číslo, které hráči neumožní pohyb, figura se nehýbe a kostkou hází další hráč. Je-li pohyb možný jen po jedné větvi (odbočce nebo rovně), hráč musí vybrat tento směr.

Je-li už na poli, kde se figurka zastaví, nějaká jiná figurka, tato se vrátí na startovní políčko (je „vyhozena“). Na začátku hry stojí všechny figurky na startovním poli. Na startovní pole navíc není možné dojít pohybem po herním plánu.

Čistá funkce race_winner určí, zda hra popsaná polem game o počtu prvků step_count mohla proběhnout na hracím plánu zadaném parametrem plan, byl-li počet hráčů player_count (může být nejvýše 12).

Výsledkem bude:

- hodnota -1 je-li hra neplatná (nemohla proběhnout),
- hodnota 0 je-li hra neukončená (dosud nikdo nevyhrál),
- kladné číslo – pořadové číslo hráče, který vyhrál.

```
int race_winner( struct location *plan, struct step *game,
                 int step_count, int player_count );
```

S.2.b: permute

V této úloze budeme pracovat s permutacemi (bijekcemi). Permutace bude zadaná jako pole n celých čísel v rozsahu od 0 do $n-1$, přitom každé se v poli objeví právě jednou.

Podprogram apply_permutation obdrží ukazatel na pole slov array, permutaci permutation a jejich společnou velikost size. Pole array pak na místě a v lineárním čase přeuspořádá podle dané permutace. Nechť:

- σ je zadaná permutace ($\sigma(i)$ = permutation[i]),
- B (before) je původní stav $B(i)$ = array[i] před voláním apply_permutation a,
- A (after) je nový stav $A(i)$ = array[i] po volání apply_permutation.

Pak pro každé $i < n$ platí A(σ(i)) = B(i), tj. prvek se přesune z pozice i na pozici $\sigma(i)$.

Nápověda: každou permutaci je možné rozložit na disjunktní cykly, přitom cyklus je posloupnost p_1, p_2, …, p_n taková, že $p_{i+1} = \sigma(p_i)$ a $p_1 = \sigma(p_n)$.

Pole array nebude mít víc než 8192 prvků (16KiB). Můžete také předpokládat, že na zásobníku je před voláním apply_permutation alespoň 2KiB volného místa.

```
void apply_permutation( unsigned *array, const int *permutation,
                        int size );
```

Podprogram find_permutation obdrží dvě pole slov, from a to, obě velikosti size. Je-li to možné, do pole permutation vyplní permutaci takovou, aby volání apply_permutation(from, permutation, size) způsobilo, že from a to budou identická pole, a vrátí true. V opačném případě vrátí false a obsah pole permutation není určen.

Složitost může být nejvýše kvadratická a platí stejná omezení na velikost vstupu a místo na zásobníku jako u apply_permutation.

```
bool find_permutation( unsigned *from, unsigned *to, int size,
                       int *permutation );
```

S.2.c: sched

Vaším úkolem je naprogramovat jednoduchý plánovač vláken typu round robin s dynamickou prioritou. Priorita bude v rozsahu 1 (nejvyšší) až 4 (nejnižší).

Vlákna budou reprezentované identifikátory – celými čísly bez znaménka. Můžete předpokládat, že nejvyšší identifikátor vlákna bude 63. Pro strukturu sched_data můžete využít až 4096 bajtů paměti.

```
struct sched_data;
```

Podprogramu sched_init bude předána prázdná struktura sched (všechny položky budou vynulované) – jejím úkolem je nastavit vše potřebné pro zbytek plánovače.

```
void sched_init( struct sched_data *info );
```

Podprogram sched_add zaregistruje nové vlákno. Vlákno bude spuštěno jako poslední v prioritní třídě zadané parametrem prio (číslo 1–4). Vstupní podmínkou je, že vlákno tid dosud zaregistrováno nebylo.

```
void sched_add( struct sched_data *info, unsigned tid, int prio );
```

Podprogram sched_up zvýší prioritu zadaného vlákna o jednu úroveň a novou prioritu vrátí. Je-li již vlákno na prioritě 1, jeho priorita se nezmění. Dojde-li ke změně priority, vlákno bude v nové prioritní třídě spuštěno jako poslední.

```
int sched_up( struct sched_data *info, unsigned tid );
```

Podprogram sched_down sníží prioritu zadaného vlákna o jednu úroveň a novou prioritu vrátí. Je-li již vlákno na prioritě 4, jeho priorita se nezmění. Dojde-li ke změně priority, vlákno bude v nové prioritní třídě spuštěno jako poslední.

```
int sched_down( struct sched_data *info, unsigned tid );
```

Podprogram sched vybere vlákno, které bude spuštěno jako další a vrátí jeho identifikátor. Zároveň upraví datové struktury tak, aby další volání

sched vrátilo další vlákno v pořadí. Vstupní podmínkou je existence alespoň jednoho vlákna.

Vlákna s vyšší prioritou mají vždy přednost před vlákny s nižší prioritou. Má-li několik vláken stejnou prioritu, vybere se to, které čeká na spuštění nejdéle.

```
unsigned sched( struct sched_data *info );
```

S.2.d: heap

Vaším úkolem bude naprogramovat jednoduchý dynamický alokátor založený na principu „first fit“ – paměť bude alokovat vždy na nejnižší možné adrese. Alokace i dealokace paměti může mít až lineární složitost.

```
struct mem_arena;
```

Podprogram mem_init připraví paměť pro použití podprogramy mem_alloc a mem_free. Oblast paměti, kterou bude využívat, je zadaná počáteční adresou memory a počtem bajtů bytes.

Proceduru mem_init nechť je možné ve stejném programu používat opakovaně (s nepřekrývajícími se oblastmi paměti).

Návratovou hodnotou je ukazatel, který bude předán podprogramům mem_alloc a mem_free. Veškerá metadata musí být uložena v předané oblasti paměti memory. Maximální povolená režie je 128 bajtů pro pevná metadata + 4 bajty na každou alokaci.

```
struct mem_arena *mem_init( unsigned char *memory, int bytes );
```

Podprogram mem_alloc nalezne vhodnou nevyužitou oblast paměti velikosti alespoň bytes, tuto označí jako využitou a ukazatel na tuto paměť vrátí. Není-li potřebná paměť k dispozici, vrátí nulový ukazatel.

```
unsigned char *mem_alloc( struct mem_arena *arena, int bytes );
```

Podprogram mem_free označí paměť jako nevyužitou. Vstupní podmínkou je, že mem je návratová hodnota předchozího volání mem_alloc se stejným parametrem arena a dosud nebyla voláním mem_free uvolněna.

```
void mem_free( struct mem_arena *arena, unsigned char *mem );
```

Část 9: Dynamické pole

Ukázky:

1. xxx

Přípravy:

1. resize – zvětšení 2D pole
2. deque – obousměrně dynamické pole
3. xxx
4. index – indexace n-rozměrného pole
5. stack – zásobník pomocí dynamického pole
6. queue – fronta ze dvou zásobníků

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

9.p: Přípravy

9.p.1 **[resize]** Napište podprogram resize_2d, který provede kopii dat z původního dvourozměrného pole do nového, většího. Nově vzniklé pozice nastavte na hodnotu value.

```
void resize_2d( int *old_data, int old_w, int old_h,
                int *new_data, int new_w, int new_h,
                int value );
```

9.p.2 **[deque]** Vaším úkolem bude tentokrát implementovat oboustranné dynamické pole – tzn. takové, které umožňuje přidávat prvky jak na začátek, tak na konec. Paměť budeme frontě předávat jako zřetězený seznam bloků typu struct mem_area:

```
struct mem_area
{
    unsigned char *start, *end;
    struct mem_area *next;
};
```

Dynamické pole samotné bude popsané strukturou deque. Oblast paměti, ve které je pole alokované, je určeno hlavou seznamu memory. Tento může navíc obsahovat další položky, v takovém případě jsou tyto určeny pro realokaci pole – seznam je seřazen vzestupně podle velikosti bloku.

```
struct deque
{
    int *start, *end;
    struct mem_area *memory;
};
```

Podprogramy push_front a push_back vloží prvek s hodnotou value na začátek resp. konec dynamického pole. Je-li potřeba provést realokaci, použije se paměť ze seznamu memory. Není-li k dispozici potřebná paměť, operace neproběhne a výsledkem bude false. Již nepotřebné bloky paměti vložte na začátek seznamu určeného vstupně-výstupním parametrem recycle.

Za předpokladu, že velikost oblastí v seznamu memory roste exponenciálně, musí mít obě operace amortizovaně konstantní složitost, a to bez ohledu na to, jakou posloupnost operací push_front a push_back uvažujeme (tzn. celková složitost vložení n prvků, libovolně rozložených mezi konec a začátek, bude $O(n)$).

```
bool push_front( struct deque *deque, int value,
                 struct mem_area **recycle );
bool push_back( struct deque *deque, int value,
                struct mem_area **recycle );
```

9.p.3 **[xxx]**

9.p.4 **[index]** Nejjednodušší reprezentace vícerozměrného pole (matice, tenzoru) je pomocí pole jednorozměrného a mapování indexů. Vaším úkolem bude naprogramovat dvojici funkcí, které budou tento přepočet provádět pro obecné n-rozměrné pole. Počet dimenzí a jejich velikost je zadaná parametry dimensions (počet) a dimension_bounds (pole rozměrů). Pole je uspořádáno tak, že první index roste nejpomaleji a poslední nejrychleji.

Funkce from_compound provede konverzi složeného indexu na jednoduchý, funkce to_compound pak konverzi opačným směrem. Funkce to_compound vyplní vypočtené indexy do nachystaného pole (výstupního parametru) compound_index.

```
int from_compound( int dimensions, int *dimension_bounds,
                   int *compound_index );
void to_compound( int index, int dimensions, int *dimension_bounds,
                  int *compound_index );
```

9.p.5 **[stack]** Vaším úkolem je implementovat zásobník pomocí dynamického pole (tzn. procedury push a pop). V dalším příkladu pak s pomocí dvou takových zásobníků implementujete frontu.

Zásobník budeme reprezentovat strukturou stack. Zásobník, který má všechny ukazatele nulové, považujeme za prázdný.

```
struct stack
{
    int *start, /* start address */
        *top,   /* one past the top */
        *end;   /* one past the allocated area */
};
```

Pro alokaci paměti budeme používat následující dvě struktury. Struktura bucket popisuje blok paměti – tento blok začíná těsně za koncem příslušné struktury bucket a budeme jej vždy používat jako celek. Struktura arena pak obsahuje zřetězené seznamy volných bloků – na indexu i to budou právě bloky velikosti 2^{i+3} bajtů.

```
struct bucket
{
    struct bucket *next;
};

struct arena
{
    struct bucket *free[ 8 ];
};
```

Procedura push vloží prvek na vrchol zásobníku. Potřebnou paměť bude alokovat z předané struktury arena. Není-li to možné, vrátí false a operaci neprovede. Již nepotřebnou paměť vrátí předané aréně.

```
bool push( struct stack *stack, int value, struct arena *mem );
```

Procedura pop odstraní prvek z vrcholu zásobníku a jeho hodnotu vrátí. Alokovanou paměť zmenšovat nebudeme. Vstupní podmínkou je, že zásobník není prázdný.

```
int pop( struct stack *stack );
```

Konečně čistá funkce empty vrátí true právě když je zásobník prázdný.

```
bool empty( struct stack *stack );
```

9.p.6 **[queue]** Máme-li dvojici zásobníků, můžeme z nich relativně jednoduše vytvořit frontu:

- je potřeba vhodně zvolit kam budeme prvky vkládat,
- odkud je budeme odebírat,
- v příhodných chvílích vhodným způsobem prvky přesunout.

Za předpokladu, že operace push a pop jsou amortizovaně konstantní, bude totéž platit pro enqueue a dequeue.

V předchozí přípravě jste implementovali zásobník pomocí dynamického pole. Tuto implementaci zde můžete přímo použít; zásobník budeme reprezentovat strukturou stack. Zásobník, který má všechny ukazatele nulové, považujeme za prázdný.

```
struct stack
{
    int *start, /* start address */
        *top,   /* one past the top */
        *end;   /* one past the allocated area */
};
```

Pro alokaci paměti budeme používat následující dvě struktury. Struktura bucket popisuje blok paměti – tento blok začíná těsně za koncem příslušné struktury bucket a budeme jej vždy používat jako celek. Struktura arena pak obsahuje zřetězené seznamy volných bloků – na indexu i to budou právě bloky velikosti 2^{i+3} bajtů.

```
struct bucket
{
    struct bucket *next;
};

struct arena
{
    struct bucket *free[ 8 ];
};
```

Procedura push vloží prvek na vrchol zásobníku. Potřebnou paměť bude alokovat z předané struktury arena. Není-li to možné, vrátí false a operaci neprovede. Již nepotřebnou paměť vrátí předané aréně.

```
bool push( struct stack *stack, int value, struct arena *mem );
```

Procedura pop odstraní prvek z vrcholu zásobníku a jeho hodnotu vrátí. Alokovanou paměť zmenšovat nebudeme. Vstupní podmínkou je, že zásobník není prázdný.

```
int pop( struct stack *stack );
```

Konečně čistá funkce empty vrátí true právě když je zásobník prázdný.

```
bool empty( struct stack *stack );
```

Nyní je Vaším úkolem implementovat frontu (procedury enqueue a dequeue) použitím dvou takových zásobníků.

```
struct queue
{
    struct stack *front, *back;
};
```

Procedura enqueue vloží prvek do fronty (umožní-li to dostupná paměť) a vrátí true. V případě nezdaru vrátí false a frontu nijak nezmění. Procedura dequeue odstraní prvek z fronty a jeho hodnotu uloží do výstupního parametru value. Vstupní podmínkou je, že fronta nebyla prázdná.

Dojde-li paměť během operace dequeue, fronta bude v blíže neurčeném ale konzistentním stavu (zejména nebude docházet k únikům paměti a frontu bude možné nadále používat, pouze se nějaké prvky mohou ztratit nebo i objevit).

```
bool enqueue( struct queue *q, int  value, struct arena *mem );
bool dequeue( struct queue *q, int *value, struct arena *mem );
```

Část 10: Datové struktury v poli

Ukázky:

1. xxx

Přípravy:

1. shortest – nalezení nejkratší větve stromu
2. ancestor – nejbližší společný předek ve stromě
3. conn – existence cesty v orientovaném grafu
4. usssp – nejkratší cesty do všech vrcholů grafu
5. wsssp – totéž s ohodnocenými hranami
6. spanning – minimální kostra

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

10.p: Přípravy

10.p.1 [xxx]

10.p.2 [xxx]

10.p.3 [xxx]

10.p.4 [xxx]

10.p.5 [xxx]

10.p.6 [xxx]

Část 11: Hašovací tabulka

Ukázky:

1. xxx

Přípravy:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

11.p: Přípravy

11.p.1 [xxx]

11.p.2 [xxx]

11.p.3 [xxx]

11.p.4 [xxx]

11.p.5 [xxx]

11.p.6 [xxx]

Část 12: Vyhledávací strom

Ukázky:

1. xxx

Přípravy:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

Řešené příklady:

1. xxx
2. xxx
3. xxx
4. xxx
5. xxx
6. xxx

12.p: Přípravy

12.p.1 [xxx]

12.p.2 [xxx]

12.p.3 [xxx]

12.p.4 [xxx]

12.p.5 [xxx]

12.p.6 [xxx]

Část S.3: Datové struktury

1. a_union – struktura Union-Find v dynamickém poli
2. b_xxx
3. c_xxx
4. d_scapegoat – implementace vyhledávacího stromu
5. e_xxx
6. f_xxx

V příkladech d, e, f máte krom základního jazyka k dispozici také „zabudované“ podprogramy malloc a free.

S.3.a: union

Implementujte datovou strukturu union-find (známou také jako disjoint-set) pomocí dynamického pole.

Část K: Vzorová řešení

K.1: Týden 1

K.1.01.1 [reverse]

```
    put  0x0001 → t1    ; maska zdrojového bitu
    put  0x8000 → t2    ; maska cílového bitu
    put  0      → rv
loop:
    and  l1, t1 → t3    ; zjištění zdrojového bitu
    jz   t3, zero
    or   rv, t2 → rv    ; nastavení cílového bitu
zero:
    shl  t1, 1  → t1    ; posuv obou masek
    shr  t2, 1  → t2
    jz   t2, check      ; skončit, když cílový bit vypadl z registru
    jmp  loop
```

K.1.01.2 [hamming]

```
    put      0 → rv
    put 0x8000 → t3    ; maska srovnávaného bitu
loop:
    and  t3, l1 → t1    ; hodnoty srovnávaných bitů
    and  t3, l2 → t2
    ne   t1, t2 → t4    ; zvýšit čítač, pokud se bity nerovnají
    add  rv, t4 → rv
    shr  t3,  1 → t3    ; posunout masku srovnávaného bitu
    jz   t3, check      ; konec, pokud maskovaný bit vypadl z registru
    jmp  loop
```

K.1.01.3 [packed]

```
    add  l1, l2     → rv ; součet spodní slabiky
    and  rv, 0x00ff → rv
    and  l1, 0xff00 → l1 ; vynulovat spodní slabiky vstupů
    and  l2, 0xff00 → l2
    add  l1, l2     → t1 ; součet horních slabik
    or   t1, rv     → rv ; promítnout horní součet do výsledku
    jmp  check
```

K.1.01.4 [bitswap]

```
    put  0x0000 → rv
    shl   1, l2 → t2 ; masky zadaných bitů
    shl   1, l3 → t3
    and  l1, t2 → t4 ; extrakce zadaných bitů
    and  l1, t3 → t5
    xor  t2, -1 → t6 ; vynulování zadaných bitů
    and  l1, t6 → rv
    xor  t3, -1 → t6
    and  rv, t6 → rv
    jz   t4, zero    ; nastavení bitů
    or   rv, t3 → rv
zero:
    jz   t5, check
    or   rv, t2 → rv
    jmp  check
```

K.1.01.5 [collatz]

```
    put   0 → rv        ; vynulovat výstupní registry
    put   0 → l6
check_max:
    ugt  l1, l6 → t1    ; ověřit a nastavit nové maximum
    jz   t1, loop
    copy l1 → l6
loop:
    eq   l1,  1 → t1    ; ukončit při jedničce
    jnz  t1, check
    add  rv,  1 → rv    ; jinak zvýšit čítač
    and  l1,  1 → t1    ; test sudosti
    jz   t1, even
    mul  l1,  3 → l1    ; n = 3n + 1
    add  l1,  1 → l1
    jmp  check_max      ; možná jsme našli nové maximum
even:
    udiv l1,  2 → l1    ; n = n / 2
    jmp  loop           ; při dělení jsme jistě maximum nenašli
```

K.2: Týden 2

K.2.r.1 [palindrome]

```
bool is_binary_palindrome( unsigned n ) __override
{
    for ( int i = 0; i < 8; ++i )
        if ( !( n & ( 1u << i ) ) != !( n & ( 0x8000u >> i ) ) )
            return false;
    return true;
}
```

K.2.r.2 [largest]

```
unsigned largest_digit( unsigned n, unsigned base ) __override
{
    unsigned max = 0;

    while ( n > 0 )
    {
        unsigned d = n % base;
        n /= base;
        max = d > max ? d : max;
    }

    return max;
}
```

K.2.r.3 [factors]

```
unsigned factor_count( unsigned n ) __override
{
    assert( n > 0 );

    unsigned count = 0;
    unsigned d = 2;

    while ( n > 1 )
    {
        if ( n % d )
            ++ d;
        else
        {
            n /= d;
            ++ count;
        }
    }

    return count;
}
```

K.2.r.4 [primes]

```
unsigned prime_count( unsigned n ) __override
{
    assert( n > 0 );
```

```
    unsigned count = 0;
    unsigned d = 2;

    while ( n > 1 )
    {
        if ( n % d == 0 )
            ++ count;

        while ( n % d == 0 )
            n /= d;

        ++ d;
    }

    return count;
}
```

K.2.r.5 [transpose]

```
unsigned transpose( unsigned m, int size ) __override
{
    unsigned res = 0;

    for ( int y = 0; y < size; ++y )
    {
        for ( int x = 0; x < size; ++x )
        {
            int src = y * size + x;
            int dst = x * size + y;

            if ( m & ( 1u << src ) )
                res |= 1u << dst;
        }
    }

    return res;
}
```

K.2.r.6 [balanced]

```
unsigned balanced_digits( int n ) __override
{
    unsigned counts = 0;

    while ( n )
    {
        int d = n % 3;
        n /= 3;

        if ( d == 2 || d == -2 )
        {
            d /= -2;
            n -= d;
        }

        if ( d == 1 )
            counts += 0x0001u;
        else if ( d == -1 )
            counts += 0x0100u;
    }

    return counts;
}
```

K.3: Týden 3

K.3.r.1 [power]

```
unsigned char power( unsigned char base, unsigned char exponent,
                    unsigned modulus )
{
    if ( exponent == 0 )
        return 1;
```

Pozor, unsigned char se v aritmetických operacích implicitně povyšuje na (znaménkový) int a násobení pak může přetéci. Přetypujeme-li jeden z činitelů explicitně na unsigned, bude na unsigned implicitně přetypován i druhý činitel; součin bude také typu unsigned a k přetečení nedojde.

```
    unsigned squared = ( unsigned ) base * base % modulus;

    unsigned x = power( squared, exponent / 2, modulus );
    return exponent % 2 ? base * x % modulus : x;
}
```

K.3.r.2 [squares]

```
unsigned squares_above( unsigned n, unsigned m, unsigned min )
{
    if ( n == 0 )
        return 1;
    if ( m == 0 )
        return 0;

    unsigned c = 0;
    for ( int i = min; i * i <= n; ++i )
    {
        c += squares_above( n - i * i, m - 1, i );
    }
    return c;
}

unsigned squares( unsigned n, unsigned m )
{
    return squares_above( n, m, 1 );
}
```

K.3.r.3 [knight]

```
bool reachable_in( unsigned n, char x1, char y1, char x2, char y2 )
{
    if ( x1 == x2 && y1 == y2 )
        return true;
    if ( n == 0 || x1 >= 8 || x1 < 0 || y1 >= 8 || y1 < 0 )
        return false;

    for ( int dx = 1; dx <= 2; ++dx )
    {
        int dy = 3 - dx;
        for ( int sx = -1; sx < 2; sx += 2 )
        {
            for ( int sy = -1; sy < 2; sy += 2 )
            {
                if ( reachable_in( n - 1, x1 + sx * dx, y1 + sy * dy,
x2, y2 ) )
                    return true;
            }
        }
    }
    return false;
}

unsigned knight_hops( char x1, char y1, char x2, char y2 )
{
    int n = 0;

    while( !reachable_in( n, x1, y1, x2, y2 ) )
        ++ n;

    return n;
}
```

K.3.03.4 [bezout] Z https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode

```
bezout:                 ; (old_r, r) := (a, b)  # l1, l2
    put  1 -> l3        ; (old_s, s) := (1, 0)  # l3, l4
    put  0 -> l4
    put  0 -> l5        ; (old_t, t) := (0, 1)  # l5, l6
    put  1 -> l6
loop:
    jz   l2, done       ; while r ≠ 0 do
```

```
    sdiv l1, l2 -> l7   ;   quotient := old_r div r # l7
    copy l2     -> t1   ;   (old_r, r) := (r, old_r - quotient × r)
    mul  l2, l7 -> l2
    sub  l1, l2 -> l2
    copy t1     -> l1
    copy l4     -> t1   ;   (old_s, s) := (s, old_s - quotient × s)
    mul  l4, l7 -> l4
    sub  l3, l4 -> l4
    copy t1     -> l3
    copy l6     -> t1   ;   (old_t, t) := (t, old_t - quotient × t)
    mul  l6, l7 -> l6
    sub  l5, l6 -> l6
    copy t1     -> l5
    jmp  loop
done:
    copy l3 -> rv
    copy l5 -> l1
    ret
```

K.5: Týden 5

K.5.e.1 [copy]

```
unsigned *copy_until( const unsigned *src, unsigned *dst,
                      unsigned n, unsigned delim )
{
    while ( n-- )
    {
        if ( *src == delim )
            break;
        *dst++ = *src++;
    }
    return dst;
}
```

K.5.r.1 [bsearch]

```
int *bsearch( int value, int *array, int size )
{
    assert( size >= 0 );

    if ( size == 0 )
        return NULL;

    if ( size == 1 )
        return *array == value ? array : NULL;

    int half = ( size + 1 ) / 2;
    return array[ half - 1 ] < value
        ? bsearch( value, array + half, size - half )
        : bsearch( value, array, half );
}
```

K.5.r.2 [qsort]

```
void swap( int *a, int *b )
{
    int tmp = *b;
    *b = *a;
    *a = tmp;
}

int partition( int *array, int last_ix )
{
    int pivot = array[ last_ix ];
    int i = -1;

    for ( int j = 0; j < last_ix; ++ j )
    {
        if ( array[ j ] <= pivot )
        {
            ++ i;
            swap( array + i, array + j );
        }
    }

    ++ i;
    swap( array + i, array + last_ix );

    return i;
}

void sort( int *array, int size )
{
    if ( size < 2 )
        return;

    int pix = partition( array, size - 1 );
    sort( array, pix );
    sort( array + pix + 1, size - pix - 1 );
}
```

K.5.r.3 [cyclic]

```
enum
{
    white,
    gray,
    black,
};

bool is_cyclic_rec( int v, const int * const *neighs, char *vis )
{
    if ( vis[ v ] == black )
        return false;
    if ( vis[ v ] == gray )
        return true;

    vis[ v ] = gray;

    for( const int *succ = neighs[ v ]; *succ >= 0; ++ succ )
        if ( is_cyclic_rec( *succ, neighs, vis ) )
            return true;

    vis[ v ] = black;
    return false;
}

bool is_cyclic( const int * const *neighs, char *scratch )
{
    for ( int v = 0; neighs[ v ]; ++ v )
        scratch[ v ] = white;

    for ( int v = 0; neighs[ v ]; ++ v )
    {
        if ( is_cyclic_rec( v, neighs, scratch ) )
            return true;
    }

    return false;
}
```

K.5.r.4 [closure]

```
bool get( int n, unsigned char *r, int ix )
{
    return ( r[ ix / 8 ] >> ( ix % 8 ) ) & 0x01;
}

void set( int n, unsigned char *r, int ix )
{
    r[ ix / 8 ] |= 0x01 << ( ix % 8 );
}

int index( int n, int i, int j )
{
    return n * i + j;
}

bool add_transitivity( int n, unsigned char *r )
{
```

```
    bool changed = false;

    for ( int i = 0; i < n; ++ i )
    {
        for ( int j = 0; j < n; ++ j )
        {
            if ( !get( n, r, index( n, i, j ) ) )
                continue;

            for ( int k = 0; k < n; ++ k )
            {
                if ( !get( n, r, index( n, i, k ) ) &&
                        get( n, r, index( n, j, k ) ) )
                {
                    changed = true;
                    set( n, r, index( n, i, k ) );
                }
            }
        }
    }

    return changed;
}

void make_transitive( int n, unsigned char *r )
{
    while ( add_transitivity( n, r ) )
        ;
}
```

K.6: Týden 6

K.6.r.1 [rle]

```
int rle_canonize( struct rle_item *begin, int length )
{
    const struct rle_item *read = begin;
    struct rle_item *write = begin;

    while ( length > 0 )
    {
        struct rle_item collapsed = *read;
        while ( --length > 0 )
        {
            ++ read;
            if ( read->value == collapsed.value )
                collapsed.count += read->count;
            else if ( read->count )
                break;
        }

        if ( collapsed.count )
            *write++ = collapsed;
    }

    return write - begin;
}
```

K.6.r.2 [msort]

```
struct node *merge( struct node *xs, struct node *ys )
{
    struct node *head = NULL,
                *last = NULL;

    while ( xs || ys )
    {
        struct node **l =
            ( xs && ys && xs->value < ys->value ) || ( xs && !ys ) ? &xs
: &ys;

        if ( last )
            last->next = *l;

        last = *l;
        *l = ( *l )->next;

        if ( !head )
            head = last;
    }

    return head;
}

struct node *sort( struct node *head )
{
    if ( !head || !head->next )
        return head;

    struct node *mid  = head,
                *prev = NULL,
                *end  = head;

    while ( end && end->next )
    {
        prev = mid;
        mid = mid->next;
        end = end->next->next;
    }

    prev->next = NULL;

    return merge( sort( head ), sort( mid ) );
}
```

K.6.r.3 [lasso]

```
bool is_lasso( struct node *fast )
{
    struct node *slow = fast;

    while ( fast && fast->next )
    {
        fast = fast->next->next;
        slow = slow->next;

        if ( fast == slow )
            return true;
    }

    return false;
}
```

K.6.r.4 [length]

```
int length( struct node *head )
{
    struct node *fast = head,
                *slow = head;

    int fast_steps = 0;

    do
    {
        if ( !fast )
            return fast_steps;
        if ( !fast->next )
            return fast_steps + 1;

        fast = fast->next->next;
        slow = slow->next;

        fast_steps += 2;
    }
    while ( fast != slow );

    int length = 0;
    do
    {
        ++ length;
        fast = fast->next;
    }
    while ( fast != slow );
```

```
    while ( head != fast )
    {
        ++ length;
        head = head->next;
        fast = fast->next;
    }

    return length;
}
```

K.6.r.5 [out]

```
struct node *count_out( struct node *head, unsigned n )
{
    assert( head );

    unsigned length = 1;
    struct node *prev = head,
                *next = head->next;

    while ( next != head )
    {
        prev = next;
        next = next->next;
        ++ length;
    }

    while ( length > 1 )
    {
        for ( unsigned m = n % length; m > 0; --m )
        {
            prev = next;
            next = next->next;
        }

        next = next->next;
        prev->next = next;
        -- length;
    }

    assert( next->next == next );

    return next;
}
```

K.7: Týden 7

K.7.r.1 [msort]

```
void merge_to( int *src1, int len1, int *src2, int len2, int *dst )
{
    while ( len1 || len2 )
    {
        if ( ( len1 && len2 && *src1 < *src2 ) || ( len1 && !len2 ) )
        {
            *dst++ = *src1++;
            -- len1;
        }
        else
        {
            *dst++ = *src2++;
            -- len2;
        }
    }
}

void sort_to( int *data, int *dst, int *scratch, int length, int *scratch_end
)
{
    assert( length > 0 );

    if ( length == 1 )
    {
        *dst = *data;
        return;
    }

    int half = length / 2;
    int *new_scratch = scratch + ( length - half );

    assert( new_scratch <= scratch_end );

    sort_to( data + half, scratch, new_scratch, length - half, scratch_end
);
    sort_to( data, data + ( length - half ), new_scratch, half, scratch_end
);
    merge_to( data + ( length - half ), half, scratch, length - half,
dst );
}

void sort( int *data, int length, int *scratch )
{
    if ( length < 2 )
        return;

    sort_to( data, data, scratch, length, scratch + 2 * length );
}
```

K.7.r.2 [inorder]

```
int *inorder_rec( struct node *root, int *start, int *end )
{
    if ( !root )
        return start;

    start = inorder_rec( root->left, start, end );

    if ( !start || start + 1 > end )
        return NULL;

    *start++ = root->value;

    assert( start <= end );

    return inorder_rec( root->right, start, end );
}

struct values inorder( struct heap *heap, struct node *root )
{
    struct values res = { .data = ( int * ) heap->start };
    int *end = inorder_rec( root, res.data, ( int * ) heap->end );
    if ( end )
    {
        res.length = end - res.data;
        heap->start = ( char * ) end;
    }
    else
        res.data = NULL;

    return res;
}
```

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.[34] 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 → PB111 → 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 **odevzdávárny** v ISu: Student → PB111 → 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 → PB111 → 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 2022-09-17 22:43 CEST
subtest p1_foo passed    [0.5]
subtest p2_bar failed
subtest p3_baz failed
subtest p4_quux passed   [0.5]
subtest p5_wibble passed [0.5]
subtest p6_xyzzy failed
   {bližší popis chyby}
verity test failed

testing syntax of submission from 2022-09-17 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 2022-09-17 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: 2022-09-17 22:43 CEST worth *5.5 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).[35]

Výstup pro příklady ze sad je podobný, uvažme například:

```
testing verity of submission from 2022-10-11 21:14 CEST
subtest foo-small passed
subtest foo-large passed
verity test passed          [ 10]

testing syntax of submission from 2022-10-14 23:54 CEST
subtest build passed
syntax test passed

testing sanity of submission from 2022-10-14 23:54 CEST
subtest foo passed
sanity test passed

best submission: 2022-10-11 21:14 CEST worth *10 point(s)
```

[34] 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“.

[35] Jako alternativu, nechcete-li z nějakého důvodu WSL instalovat, lze použít program putty.

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 Recenze Vám adresované recenze, podobně jako archiv odevzdaných souborů, naleznete ve složce Private ve studijních materiálech (Student → PB111 → Studijní materiály → Private). Shrnutí bodového zisku za tyto recenze pak naleznete v poznámkovém bloku reviews.

T.1.6 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[36] – 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:

```
$ pb111 update
```

Stažené soubory pak naleznete ve složce ~/pb111. 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 pb111 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 ~/pb111/reviews.

T.2.3 Odevzdání řešení Odevzdat vypracované (nebo i rozpracované) řešení můžete ze složky s relevantními soubory takto:

```
$ cd ~/pb111/01
$ pb111 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

```
$ pb111 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 pb111, 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:

```
$ pb111 beamer
```

Protože příkaz vytvoří nové sezení, nezapomeňte se přesunout do správné složky příkazem cd ~/pb111/NN.

T.2.5 Recenze Příkaz pb111 update krom zdrojového balíku stahuje také:

1. zdrojové kódy, které máte možnost recenzovat, a to do složky ~/pb111/to_review,
2. recenze, které jste na svůj kód obdrželi (jak od spolužáků, tak od vyučujících), a to do stávajících složek zdrojového balíku (tzn. recenze na příklady z první kapitoly se Vám objeví ve složce ~/pb111/01 – že se jedná o recenzi poznáte podle jména souboru, který bude začínat uživatelským jménem autora recenze, např. xrockai.00123.p1_nhamming.cpp).

Chcete-li vypracované recenze odeslat:

1. přesuňte se do složky ~/pb111/to_review a
2. zadejte příkaz pb111 submit, případně doplněný o seznam souborů, které hodláte odeslat (jinak se odešlou všechny, které obsahují jakýkoliv přidaný komentář).

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.bin
```

kde příklad je název souboru bez přípony (např. tedy make p1_fib.bin). Tento příkaz přeloží program překladačem tinycc a rovnou jej také spustí, tzn. selže-li nějaký test, tuto informaci rovnou uvidíte na obrazovce.

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 (např. jej načíst do virtuálního stroje tinyvm.py z kapitoly B).

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.

[36] 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.

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 Prozatím je překladač tinycc k dispozici pouze na stroji aisa. Jakmile to bude možné, zveřejníme zdrojové kódy a/nebo již přeložené binárky tak, abyste si je mohli spustit i lokálně.

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 d), 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.[1]
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.[37]
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.

[37] 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.