Univerzita sv. Cyrila a Metoda v Trnave

Fakulta prírodných vied
Katedra aplikovanej informatiky

Algoritmizácia a základy štruktúrovaného programovania v jazyku C

1. diel

Jana Jurinová

UNIVERZITA SV. CYRILA A METODA V TRNAVE

FAKULTA PRÍRODNÝCH VIED

Katedra aplikovanej informatiky

ALGORITMIZÁCIA A ZÁKLADY ŠTRUKTÚROVANÉHO PROGRAMOVANIA V JAZYKU C

1. diel

Jana Jurinová

Trnava, 2020

Autor:

Ing. Jana Jurinová, PhD.

Recenzenti:

doc. Ing. Michal Čerňanský, PhD.

Mgr. Ing. Roman Horváth, PhD.

Vysokoškolská učebnica bola schválená Edičnou radou Univerzity sv. Cyrila a Metoda v Trnave a vedením Fakulty prírodných vied UCM.

Za odbornú a jazykovú stránku tejto vysokoškolskej učebnice zodpovedá autorka.

Rukopis neprešiel redakčnou ani jazykovou úpravou.

Vydavateľ: Univerzita sv. Cyrila a Metoda v Trnave, 2020

1. vydanie

ISBN 978-80-8105-859-2

Predhovor

Táto vysokoškolská učebnica je určená predovšetkým študentom prvého ročníka študijného programu aplikovaná informatika, ale aj všetkým tým, ktorí chcú preniknúť do problematiky algoritmizácie a programovania. Učebnica ponúka ucelený pohľad na algoritmické riešenia rôznych problémov. Čitateľ v nej nájde materiál, ktorý poskytuje efektívny prístup k osvojeniu ilustrovaných konceptov. Tomu je podriadená štruktúra a radenie jednotlivých kapitol na ktoré nadväzuje druhý diel učebnice, ktorý dopĺňa a koncepčne rozvíja obsah prvej časti. Na lepšie vnímanie a chápanie obsahu sú objasňované koncepty ilustratívne podporené príkladmi pri zachovaní jednoduchosti so snahou poukázať na širšie súvislosti a alternatívne možnosti riešenia.

Na knižnom trhu sú dostupné viaceré publikácie zamerané na programovací jazyk *C*. Čitateľ si preto môže položiť otázku, prečo vznikla táto učebnica. Cieľom je využiť tento programovací jazyk strednej úrovne na implementáciu (otestovanie) základných algoritmických problémov a ich reprezentáciu. Ak chceme tieto problémy riešiť ukážkovo a nepoužívať na reprezentáciu iné formy zápisu algoritmov, ktorým sa tiež táto učebnica venuje, bolo nevyhnutné zvoliť konkrétny programovací jazyk, v ktorom sa dajú dané algoritmy a postupy implementovať. Výučbová prax ukázala, že jazyk *C*, ktorý nie je špecializovaný na jednu oblasť použitia, je na takéto účely vhodný. Z dôvodu existencie viacerých publikácii venovaných jazyku *C*, v tejto učebnici nenájdete viaceré informácie, ktoré sú na prácu s jazykom *C* a písanie programov v ňom veľmi dôležité. Napríklad, je samozrejmé, že pri vývoji reálneho robustného systému sa venuje pozornosť testovaniu všetkých chybových stavov, kód musí byť implementovaný tak, že môže byť rýchlo a jednoducho modifikovateľný, pre iných programátorov jednoducho čitateľný a pochopiteľný, navrhnutý tak, aby dobre spolupracoval s ďalšími komponentami systému, a v neposlednom rade bol prispôsobiteľný pre iné prostredie. Ešte raz zdôrazňujem skutočnosť, že implementácie riešených problémov používame v tejto učebnici ako jednu z foriem reprezentácie algoritmu, pri primeranej snahe dodržiavať všetky vyššie popísané hľadiská. Preto v prípade záujmu zvládnutia programovania v jazyku C odporúčam siahnuť aj po inej literatúre, napríklad po *Učebnici jazyka C* od Pavla Herouta. Tento diel učebnice tvoria dve kapitoly teoretického, ale najmä praktického charakteru, ktoré sú následne členené do menších kapitol. Prvá kapitola podrobne a postupne rozoberá algoritmické riešenie problémov, ktoré využívajú základné riadiace konštrukcie a jednoduché údajové typy. Druhá kapitola je venovaná časovej a priestorovej zložitosti algoritmov. Na konci sa nachádza zoznam použitej a odporúčanej literatúry, z ktorej sme vychádzali a môže rozšíriť poznatky čitateľa.

Druhý diel tejto učebnice je primárne zameraný na ilustráciu algoritmického spracovania problémov využívajúcich štruktúrované dátové typy, návrh funkcií, aby sme navrhovali algoritmy, ktoré spĺňajú vlastnosti modifikovateľnosti a štruktúrovanosti, ako aj prácu s dynamickými údajovými štruktúrami, resp. zdrojom dát z externého zdroja.

Učebnica vznikla zo súboru prednáškových materiálov, poznámok, skúsenosti a spätnej väzby z praktických cvičení pri výučbe predmetov „algoritmy a dátové štruktúry I“ a „programovanie I“ študentov aplikovanej informatiky na bakalárskom stupni na Fakulte prírodných vied Univerzity sv. Cyrila a Metoda v Trnave. Spôsob výučby týchto predmetov sa vyvíjal v mnohých smeroch, čiastočne s cieľom zamerať sa na vedomostné zázemie prichádzajúcich študentov (nerozvinuté formálne zručnosti v predmetnej oblasti). V dôsledku toho boli témy starostlivo vybrané a zoskupené. Snahou nebolo vytvoriť encyklopedickú publikáciu, čo podporilo zahrnutie tém tradične zdôrazňovaných alebo vynechaných vo väčšine kníh o algoritmoch.

Učebnica vychádza v elektronickej podobe na základe dvoch obmedzujúcich faktorov. Grafická reprezentácia niektorých stredne zložitých algoritmov je natoľko rozsiahla, že ani pri použitom formáte A4 nie je možné v printovej podobe dané diagramy prezentovať v dostatočnej veľkosti. Druhým dôvodom je snaha uľahčiť čitateľom prácu, preto ako súčasť učebnice poskytujem zdrojové kódy všetkých riešených príkladov, vďaka ktorým si môže čitateľ okamžite overiť ich funkčnosť a priamo s nimi pracovať. Odporúčam čitateľovi, aby s týmito príkladmi experimentoval, dotváral ich, rozširoval, modifikoval, pretože len aktívnym programovaním sa naučí a osvojí si požadované zručnosti. V každom kóde je vždy čo vylepšiť, prípadne zaujať iný postoj k riešeniu. Kódy programov je možné automaticky otvoriť v konkrétnom softvéri po kliknutí na link, ktorý predstavuje jeho označenie. Pri uvádzaní komentárov v programe sú tieto výhradne uvádzané bez diakritiky. Treba zdôrazniť aj zastúpenie anglického jazyka vo výrazových prostriedkoch používaných v programovaní. Pre programátora je znalosť anglického jazyka prakticky nevyhnutná, aspoň na úrovni čítania a porozumenia dokumentácií. Preto väčšina slovenských kľúčových termínov použitých v učebnici je na lepšie pochopenie interpretovaná aj v anglickom jazyku.

Pri čítaní jednotlivých kapitol v chronologickom poradí buďte, prosím, trpezliví. Miestami sú používané pojmy a informácie detailnejšie vysvetlené až nižšie. Je to tak z viacerých dôvodov, ale hlavným je vyhnúť sa opakovaniu poskytovaných informácií a nenavyšovať umelo rozsah tejto učebnice. Ak by ste však aj po preštudovaní celej učebnice, resp. jej druhého dielu, a pri mojej intenzívnej snahe opísať a poskladať informácie logicky, predsa len mali

otázky, prípadne ak by ste objavili nejakú chybu, alebo by vám nejaké informácie v tejto učebnici chýbali, neváhajte ma kontaktovať na jana.jurinova@ucm.sk.

Rada by som na tomto mieste poďakovala doc. Ing. Michalovi Čerňanskému, PhD. a Mgr. Ing. Romanovi Horváthovi, PhD. za ich prínosné odborné komentáre, poznámky a poznatky pri recenzovaní tejto učebnice. Ich skúsenosti bezpochyby výrazne prispeli k skvalitneniu informácií v tejto učebnici.

December 2020 Ing. Jana Jurinová, PhD., autorka učebnice

Obsah

1 Základné pojmy z algoritmizácie a programovania

Táto kapitola obsahuje stručný prehľad základných pojmov z algoritmizácie, ako aj programovania.

1.1 Algoritmický problém, algoritmus, algoritmizácia

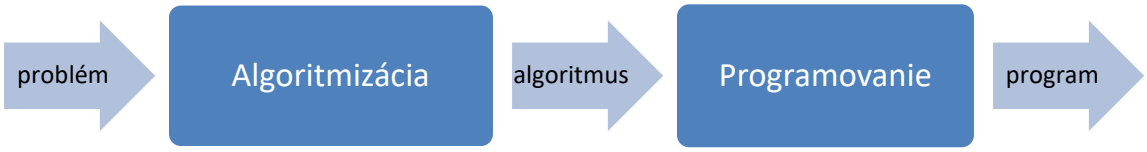
Hoci algoritmy sú oveľa zložitejšie, pojem algoritmus sa v literatúre často vysvetľuje na príklade receptu. Od začiatku 20. storočia sa termín *algoritmus* používa v zmysle univerzálneho návodu, či postupu na riešenie určitej triedy úloh, ktorá sa skladá z konečnej postupnosti jednoznačne definovaných krokov. Správny algoritmus vyjadruje postup práce na dosiahnutie stanoveného cieľa. Berie pritom do úvahy všetky detaily, možnosti, náhody alebo zriedkavé situácie, ktoré s konkrétnym algoritmom súvisia. Často je riešením jedného problému hneď niekoľko algoritmov s rôznymi postupnosťami krokov. Rôzne algoritmy sa tiež môžu líšiť v množstve času a pamäte, ktoré sú potrebné na ich kompletné vykonanie a riešenie úlohy. Algoritmus je možné patentovať. Jednoducho by sme mohli **algoritmický problém** zadefinovať ako problém, ktorý je charakterizovaný vstupnými a výstupnými hodnotami (údajmi, premennými, stavmi), vstupnou a výstupnou podmienkou.

Všeobecné pravidlá, ktoré definujú postupnú transformáciu vstupných údajov na výstupné pri splnení vstupných a výstupných podmienok, nazývame **algoritmus**. Vstupné údaje musia spĺňať definované vstupné podmienky na vstupe algoritmu, obdobne výstupné údaje musia spĺňať definované výstupné podmienky na výstupe algoritmu. Inými slovami, pod pojmom algoritmus rozumieme presne definovanú konečnú postupnosť pravidiel, ktorých realizácia nám pre vstupné dáta umožní po konečnom počte krokov získať zodpovedajúce výstupné dáta. Transformácia vstupných údajov na výstupné sa obvykle nerealizuje priamo, ale je kompozíciou vhodných operácií (elementárnych príkazov), ktoré zabezpečia spôsob transformácie.

Algoritmizácia je disciplína študujúca algoritmy, pričom pod algoritmizáciou problému rozumieme samotné vytvorenie algoritmu riešeného problému s cieľom splnenia všetkých vlastností algoritmu. Na rozdiel od teórie algoritmov, čo je matematická veda študujúca matematické modely algoritmov.

1.2 Programovanie, programovací jazyk, program

Programovanie môžeme definovať ako zápis algoritmov v jazyku, ktorému rozumie počítač, v tzv. programovacom jazyku. Je to inžinierska činnosť, nie je to veda ani umenie, na rozdiel od teórie algoritmov, čo je veda študujúca algoritmy. Vytvorenie algoritmu riešeného problému nazývame algoritmizáciou problému. Celý proces môžeme v jednoduchosti vyjadriť takto:



Obr. 1 Proces algoritmizácie problému.

Rozoznávame viaceré metódy (techniky, štýly) programovania (**programovacie paradigmy**), ktoré sa odlišujú najmä druhom a spôsobom rozmýšľania, logikou, ako aj sémantikou. Vývoj týchto paradigiem úzko súvisí s programovacími jazykmi, ktoré zase súvisia s vývojom hardvéru, a ten súvisí s riešenými problémami súčasnosti. Existujú programovacie jazyky ktoré sú multiparadigmové, to znamená, že podporujú viaceré štýly programovania. Napr. jazyk *C++* umožňuje procedurálne (procedúry, funkcie), objektovo orientované (triedy) ale aj generické programovanie (šablóny = *templates*). Na druhej strane existujú jazyky, ktoré nútia programátora dodržiavať určité typické konvencie danej programovacej paradigmy.

Medzi základné programovacie paradigmy patrí:

1. **Procedurálne programovanie –** je základnou metódou imperatívneho programovania. Program je kolekciou explicitne pomenovaných pamäťových miest (premenných), postupností príkazov a funkcií (podprogramov), ktoré reprezentujú parciálne algoritmy, resp. riešenie danej úlohy. Ak neurčí riadiaca konštrukcia inak, vykonávajú sa príkazy (inštrukcie) v takej postupnosti ako sú zapísané (zhora nadol). Úlohou programátora je navrhnúť celý výpočet ako postupnosť jednotlivých krokov, ako aj organizovať použitie pamäti počas výpočtu tak, aby príkazy postupne menili obsah pamäťových miest s cieľom dosiahnuť želaný výsledok. Môže uplatniť štruktúrovaný, alebo modulárny prístup k riešeniu problémov. Hlavným cieľom štruktúrovaného prístupu je dosiahnutie lepšej zrozumiteľnosti, vyššej kvality a rýchlejšie vytváranie programov skladaním implementovaného algoritmu z riadiacich štruktúr s jedným vstupným a jedným výstupným bodom, namiesto neobmedzeného použitia skokov. Program môže byť

členený na jednotlivé časti (moduly). Pri modulárnom prístupe môžeme povedať, že jeden modul programu vykonáva jednu akciu, zatiaľ čo druhý modul vykonáva inú akciu. Rovnaké akcie pritom môžu byť vykonávané nad rôznymi dátami. „Modulárne programovanie často využíva už existujúce, alebo dopredu napísané moduly. Telo programu sa tvorí s využitím týchto modulov. Ideálnou metódou sa zdá byť prepojenie oboch metód“ (Machová, Spišáková, 2011, str. 6). Táto paradigma je vhodná skôr pre malé projekty. Typickými predstaviteľmi sú *C*, *Pascal*, *Cobol*.

2. **Objektovo orientované programovanie (OOP) –** program je kolekciou diskrétnych objektov. Každý objekt je inštanciou nejakej triedy. Objekt zapúzdruje dáta (uchováva si stav vo vlastných premenných) a metódy (predstavujú správanie). Celá komunikácia je riadená posielaním si správ medzi sebou. Tieto správy spúšťajú jednotlivé metódy. Hlavná výhoda tohto prístupu je vyššia využiteľnosť kódu. OOP je vhodné pre rozsiahlejšie projekty. Typickými predstaviteľmi sú *C++*, *C#*, *Python*, *Java*.
3. **Funkcionálne programovanie –** základom tohto deklaratívneho prístupu programovania je abstraktná teória lambda kalkulu (*lambda calculus*). Tento štýl nepozná objekty ani triedy, nepozná priradenie do premennej, globálne premenné a ani cyklus, program je len kolekciou funkcií. Priebeh výpočtu je založený na postupnom aplikovaní funkcií. Funkcie bývajú aplikované na výsledky iných funkcií, t. j. funkcia je hodnotou. Široké zastúpenie tu nachádza rekurzia. Typickými predstaviteľmi sú *Haskell*, *Lisp*.
4. **Logické (relačné) programovanie** – základom tohto deklaratívneho prístupu k programovaniu je predikátová logika 1. rádu. Pri tomto štýle programovania sa nesústredíme ani tak na to, ako riešenie nájsť, ale na to, čo sa má riešiť, tzv. dopytujeme sa na výsledok. Definujeme relácie nad objektmi, pričom riešenie sa hľadá s pomocou backtrackingu (spätné reťazenie). To znamená, že programátor predloží systému nejaké tvrdenie (cieľovú hypotézu) a systém sa snaží dané tvrdenie dokázať na základe faktov a klauzúl programu. Namiesto funkcií používame predikáty. Typickým predstaviteľom je *Prolog*.

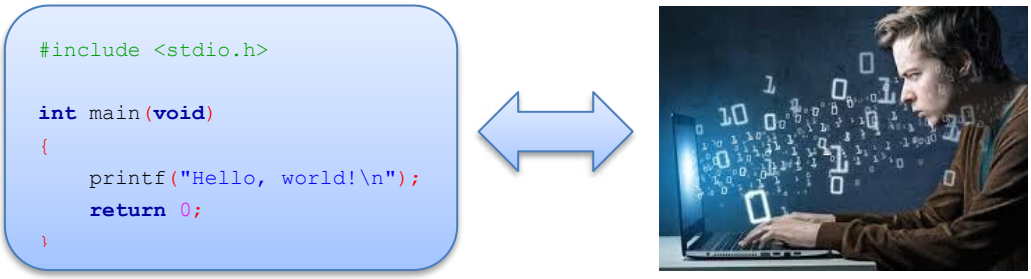
Existuje veľa definícií programovacieho jazyka a je ťažké dohľadať pôvodný zdroj. Najčastejšie sa **programovací jazyk** definuje ako nástroj na systematický opis postupu, výpočtu, správania, zmeny stavu, opis objektov, ich vlastností a štruktúr tak, aby mohol byť spracovaný počítačom. **Program** je skupina príkazov (inštrukcií) programovacieho jazyka,

ktorou opisujeme, ako má počítač riešiť určitú úlohu. Inými slovami, program je algoritmus vyjadrený programovacím jazykom. Program je chránený autorským zákonom.

Príkazy programovacieho jazyka môžeme skúmať z hľadiska ich syntaktického zápisu a sémantickej konštrukcie:

- **Syntax** – opisuje formálnu štruktúru príkazu, ako aj programu. Definuje kľúčové slová, identifikátory, určuje pravidlá, ako sa majú jednotlivé príkazy a časti programu zapisovať, alebo ako je ich možné kombinovať.
- **Sémantika** – vyjadruje logický význam príkazu, ako aj programu v danej syntaktickej podobe.

Zdrojový kód je syntakticky správne zapísaná postupnosť príkazov programovacieho jazyka. Zdrojový kód je na rozdiel od strojového kódu zrozumiteľný pre človeka (programátora).



Obr. 2 Zdrojový kód jazyka *C*.

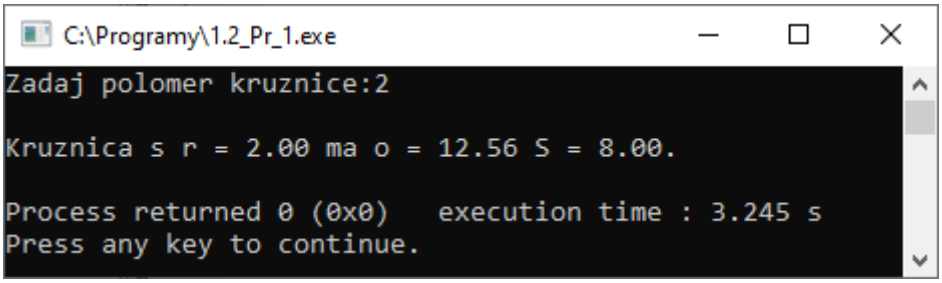
Štruktúra programu v jazyku *C*

Vychádzajúc napríklad z publikácie Horovčáka a Podlubného (1997) „základnou programovou jednotkou v jazyku *C* je funkcia. Veľké množstvo štandardných funkcií je vopred pripravených v knižniciach. Tieto funkcie vychádzajú na jednej strane z prislúchajúcej normy *ANSI*, na strane druhej priamo od autora kompilátora. Každá štandardná funkcia je spravidla sprevádzaná svojím tzv. *include* – súborom (jedným alebo viacerými). Tento súbor má príponu ***.h*** od slova *header* (hlavička) a obsahuje hlavne súhrn informácií o určitej skupine štandardných funkcií (ich deklarácie, globálne premenné a pod.). Do zdrojového textu programu sú tieto súbory zaradené s pomocou príkazu preprocesora *#include*. Rovnakým spôsobom môžu byť do zdrojového textu programu zaradené používateľom definované funkcie (podprogramy), ktoré je tiež možné združovať do knižnice. Ďalším prvkom, ktorý sa môže vyskytovať v štruktúre programu v jazyku C, sú globálne dátové prvky, ktoré sú dostupné vo všetkých funkciách daného programu. Program v jazyku *C* je teda súhrnom definícií funkcií a deklarácií globálnych prvkov. Jedna z funkcií sa musí volať ***main*** (označuje hlavný program). Jazyk *C* je

typický svojou blokovou štruktúrou (napr. telo každej funkcie tvorí jeden blok). Blokom je zdrojový text, uzavretý v dvojici zložených zátvoriek { }. V rámci bloku je možné pracovať s globálnymi i lokálnymi premennými, ktoré sú definované len v rámci bloku (lokálne môžu byť len dátové prvky, všetky funkcie sú vždy globálne)“.

Základná štruktúra programu v jazyku *C* je ilustrovaná na obr. 2. Jednoduchý program na výpočet obvodu a obsahu kružnice, na ktorom sú ilustrované viaceré spomenuté funkcionality ilustruje kód programu 1.2_Pr_1.c:

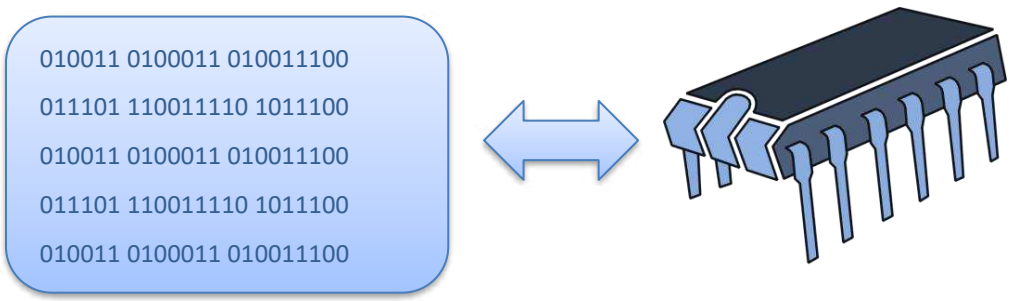
```c
#include <stdio.h>
#include <math.h> // kvoli vstavanej funkcii na vypocet druhej mocniny, pow()
#define PI 3.14   // definicia globalnej konstanty

double obvod_kruznice(double r); // deklaracia funkcie

int main(void)
{
    // const double PI = 3.14; // definicia lokalnej konstanty
    double r, obvod, obsah; // deklaracia premennych
    // osetrenie vstupnej premennej
    do
    {
        printf("Zadaj polomer kruznice:");
        scanf("%lf", &r);
    }
    while (r <= 0);
    // obvod = 2 * PI * r; // vypocet obvodu bez pouzitia funkcie
    // vypocet obsahu s vyuzitim vstavanej funkcie
    obsah = 2 * pow(r, 2);
    // vypis udajov, spolu s volanim vlastnej funkcie
   printf("\nKruznica s r = %.2lf ma o = %.2lf S = %.2lf. \n", r, obvod_kruznice(r), obsah);

    return 0;
}

// definicia funkcie
double obvod_kruznice(double r)
{
    return 2 * PI * r;
}
```



Obr. 3 Konzolový výstup programu 1.2._Pr_1.c.

Pozn. autora: Všetky uvedené funkcionality, ako aj ich rozšírenia sú v tejto učebnici, resp. v jej druhom dieli následne detailnejšie objasňované.

Programovací jazyk slúži na zápis programu. V skutočnosti však musí byť program preložený do jazyka, ktorému rozumie počítač, konkrétne procesor, teda do **strojového kódu**. Prevod z programovacieho jazyka do strojového kódu nazývame preklad (kompilácia). Prekladač je program, ktorý preloží nami vytvorený program do strojového kódu.



Obr. 4 Strojový kód.

História a charakteristika jazyka C

V čase písania učebnice je aktuálny štandard jazyka *C* tzv. *C18* (plné označenie *ISO/IEC 9899:2018*) vydaný v roku 2018, ktorý nahrádza štandard *C11* (plné označenie *ISO/IEC 9899:2011*) vydaný v roku 2011, ako aj štandard *C99* (plné označenie *ISO 9899:1999*) vydaný v roku 1999, ktorý bol prijatý za ANSI štandard v roku 2000. Jeho predchodcami boli tzv. *ISO C* (plné označenie *ISO/IEC 9899:1999* z roku 1990) a *ANSI C* z roku 1989, ktorý vychádza z *K&R* (prvý štandard z roku 1978).

Jazyk *C* je univerzálny programovací jazyk nízkej úrovne (*low level language*), avšak na strane druhej je možné ho zaradiť do skupiny vyšších programovacích jazykov. Tak ako uvádzajú aj Horovčák a Podlubný (1997) program napísaný v jazyku *C* je možné preložiť do veľmi efektívneho strojového kódu. Jazyk *C* má relatívne málo jednoduchých pravidiel, s pomocou ktorých sa dajú vytvárať a skladať jednotlivé časti programov do väčších a väčších celkov, to znamená, že má veľmi úsporné vyjadrovanie a je štruktúrovaný. Nie je špecializovaný na jednu oblasť používania. Jazyk *C* je implementovaný pre všetky typy procesorov. Vznikol celý rad rozšírení jazyka *C*, najperspektívnejším sa zdá byť jeho objektovo orientované rozšírenie, ktoré je známe pod označením jazyk *C++*. Toto sú hlavné dôvody prečo sme sa ho rozhodli využívať na reprezentáciu algoritmov v tejto učebnici.

Jazyk *C* patrí medzi tzv. **kompilačné jazyky** (*ahead–of–time compilation*). Zdrojový program musíme najskôr preložiť do podoby, ktorej rozumie procesor, a až potom môžeme program spustiť. Nejde teda o tzv. interpretačný jazyk, ktorý s pomocou interpretera interpretuje program zapísaný v nejakom programovacom jazyku riadok po riadku, alebo preloží program do nejakého medzikódu a tento medzikód potom vykonáva.

1.3 Etapy tvorby riešenia algoritmických problémov

Algoritmy môžu byť zapísané vo forme programov. Logická chyba v algoritme môže spôsobiť zlyhanie výsledného programu. Zostaviť správny algoritmus riešenia problému je často veľmi zložitý tvorivý a intuitívny proces. Má však zmysel. Pretože priamo implementovať rôzne algoritmy pre riešený problém a následne ich na základe empirickej analýzy vyhodnotiť, ktorý variant riešenia je vhodnejší, môže mať svoje výhody, ale aj nevýhody. Pri riešení jednoduchých problémov, ktorými sa v tejto učebnici zaoberám to môže byť relatívne správny postoj. Študent tak nadobudne skúsenosti s implementáciou rôznych možností riešenia problémov v konkrétnom programovacom jazyku. Následne si testovaním overí ich správnu funkčnosť, a na základe jednoduchých meraní vyhodnotí, ktorý z nich je vhodnejší. Ak študent v čase čítania tejto učebnice paralelne získava základné zručnosti z programovania, tento postup sa zdá byť aj vhodný. Avšak z pohľadu riešenia problémov v praxi je tento prístup viac ako nevhodný. Venovať čas a energiu implementácii, následnému testovaniu a čakaniu na výstupy programu, ktoré môžu v mnohých prípadoch trvať aj niekoľko dní, sa ukazuje ako značne neproduktívny prístup na to, aby sme zistili, že niečo je pomalé alebo nevhodné. O to viac ak je možné na základe algoritmickej, alebo matematickej analýzy predikovať tento výsledok, resp. predvídať ho na základe dostupných informácií z literatúry. Verím, že čitateľovi aj napriek tejto stručnej snahe o priblíženie týchto dvoch uplatňovaných prístupov k riešeniu problémov, ktoré skrývajú v skutočnosti oveľa viac detailov, je zrejmé aké úskalia tieto prístupy k riešeniu prinášajú.

Jednotlivé etapy môžeme teda charakterizovať:

1. **Zadanie úlohy a presné vymedzenie problému**

   Predstavuje rozbor problému, ktorý vedie k presnej formulácii zadania, určuje smerovanie riešenia a často aj definovanie požiadaviek kladených na program. Ak sa vyskytnú pri skúmaní viaceré možnosti riešenia, snažíme sa zistiť, ktorá z nich je najefektívnejšia. Pokúšame sa odhadnúť časovú a priestorovú zložitosť, ako aj prácnosť

naprogramovania každého z nich. Po zohľadnení týchto kritérií si vyberieme najvhodnejší spôsob. Je zrejmé, že tento bod zahŕňa analýzu problému a naznačenie riešenia, resp. riešení. Výsledkom rozboru je popis vstupov a výstupov, v prípade potreby aj pomocných premenných, ich podmienok a vzťahov medzi nimi.

2. **Návrh postupu riešenia**

   Vytvoríme algoritmus, čiže všeobecný zápis návodu riešenia. Výsledkom je zápis jednotlivých krokov (príkazov) algoritmu s využitím jednej z foriem zápisu algoritmov. Snahou je zvoliť tú najvhodnejšiu formu vzhľadom na riešený problém. Vhodné je overiť správnosť navrhnutého algoritmu, ako aj posúdiť či spĺňa všetky vlastnosti, ktoré sú naň kladené.

3. **Realizácia**

   Predstavuje prepis algoritmu do konkrétneho programovacieho jazyka, pri zohľadnení definovaných vstupných a výstupných podmienok. Každý programovací jazyk je tvorený súborom príkazov a pravidiel, ktoré majú svoju špecifickú syntax. Treba zvažovať aj skutočnosť, aby program spĺňal všetky podmienky naň kladené.

4. **Ladenie a testovanie**

   Ladenie programu predstavuje opravu syntaktických, aj logických chýb. Hlavnými cieľmi je overiť funkčnosť a správnosť programu. Medzi základné zásady testovania môžem zahrnúť:

   - poznať správny výsledok, čiže vedieť čo testujeme,
   - pri definovanej množine vstupných údajov by mal byť úplne otestovaný program vzhľadom na tieto vstupy,
   - testovanie všetkých ciest (vetiev),
   - testovanie hraničných a „kritických“ hodnôt.

   Do tejto kategórie však zahŕňame aj prispôsobenie softvéru požiadavkám používateľa.

5. **Dokumentácia a údržba**

   Ak je program dobrý a chceme ho ponúknuť aj iným ľuďom, je dobré, ak sa k nemu vytvorí minimálne pomocník (používateľská príručka). Vhodné je spracovať aj technickú dokumentáciu, ktorú môžu využiť iní programátori, kde opíšete, ako ste pristupovali k riešeniu jednotlivých problémov, atď. Táto dokumentácia často nadobúda dve podoby: komentáre v programe a písomná dokumentácia. Nesmieme zabúdať aj na údržbu

programu (podporu riešenia problémov) a ďalší vývoj (rôzne modifikácie, vylepšenia a pod.).

1.3.1 Metódy riešenia problémov

Samotné metódy riešenia algoritmických problémov môžeme rozdeliť do dvoch skupín:

1. **Všeobecné stratégie riešenia problémov**

   Ich podstata vychádza z predstavy tvorby tzv. **stavového priestoru riešenia**. Počiatočný stav je definovaný špecifikáciou problému, teda vstupnou podmienkou a vstupnými premennými. Realizátor je schopný vykonávať konečnú množinu operácií. Aplikácia týchto operácií na počiatočný stav zabezpečí generovanie nových stavov – stavov riešenia problému. V každom kroku je potrebné zisťovať, ako sa do daného stavu dostať (postupnosť operácií), a či daný stav nie je koncovým stavom riešenia, ktorý je definovaný výstupnou podmienkou a výstupnými premennými. Generovaním nových stavov riešenia sa vytvára stavový priestor riešenia problému. Tento priestor predstavuje **graf – strom**. Cesta v grafe od počiatočného stavu do konečného predstavuje postupnosť krokov riešenia daného problému, teda algoritmus.

   Všeobecné stratégie používajú metódy prehľadávania stavového priestoru do hĺbky (od koreňa k listu), alebo do šírky (po úrovniach stromu). Ich efektívnosť je malá, preto sa často pristupuje k tvorbe algoritmov priamo.

2. **Priamy postup riešenia problémov**

   Počiatočný stav zostáva rovnaký, iba stratégie sa menia. Základnými stratégiami pri priamej konštrukcii algoritmov sú abstrakcia a dekompozícia. **Abstrakcia** je proces, pri ktorom sa určité prvky, ktoré nie sú dôležité na danej úrovni, zanedbajú. **Dekompozícia** je proces, pri ktorom sa väčší problém rozkladá na menšie podproblémy. Týmto spôsobom sa pôvodný problém zjednodušuje. Takto možno problém rozložiť až na elementárne problémy, ktoré je realizátor schopný riešiť. Spätnou kompozíciou podproblémov sa potom dosahuje riešenie pôvodného problému.

Obdobne pristupujeme aj k návrhu programu, kde rozlišujeme prístup:

- **Zhora nadol** – ktorý sa začína na abstraktnej úrovni a postupne nastáva jeho zjemňovanie. Ide o tzv. prirodzený spôsob riešenia problémov – *rozdeľuj a panuj*. Medzi základné používané stratégie patrí analýza a dekompozícia.
- **Zdola nahor** – kde začíname program budovať z elementárnych jednotiek, ktoré postupne kombinujeme (skladáme) do vyšších zložitejších jednotiek.

Medzi hlavné vlastnosti programu (ukazovatele kvality) zaraďujeme: korektnosť (správnosť – schopnosť programu presne vykonávať svoju úlohu), spoľahlivosť (robustnosť – schopnosť programu reagovať na abnormálne podmienky), efektívnosť, rýchlosť, rozšíriteľnosť, kompatibilitu, dostupnosť, modifikovateľnosť, používateľsky prívetivé spracovanie (*user friendly*) a pod.

1.4 Skladba algoritmického jazyka a programovacieho jazyka C

Algoritmus sa často vytvára z dôvodu jeho následnej implementácie v konkrétnom programovacom jazyku, preto odlíšiť jednotlivé prvky používané v algoritme priamo od programovacieho jazyka je často zložité. Nami uvádzané príklady implementujeme s pomocou jazyka *C*. Ako vyplýva z názvu, táto kapitola poskytuje jednu spoločnú komplexnú východiskovú kapitolu. Je to tak najmä z dôvodu úzkeho prepojenia objasňovaných pojmov a skutočností v algoritmizácii a v programovaní, a vyhnutia sa redundancii objasňovaných informácií, ku ktorému by pri rozdelení tejto kapitoly určite prišlo. Pri príkladoch uvedených v kapitole 1.6 sa čitateľ stretne s konkrétnymi ukážkami, pri ktorých pochopí jednoznačný rozdiel medzi algoritmickým a programovacím jazykom. Väčšina informácií týkajúcich sa jazyka *C* v tejto kapitole sa opiera o viaceré odborné publikácie (Belan, 2013; Herout, 2010; Knuth, 2008; Horovčák, Podlubný, 1997).

1.4.1 Premenná

Premenná je objekt = miesto v pamäti počítača, ktorý obsahuje určitú hodnotu presne stanoveného typu (napr. celé číslo, reálne číslo, reťazec znakov a pod.). Každá premenná je označená identifikátorom (menom). Najčastejšie deklarácia pozostáva z uvedenia údajového typu a mena premennej. Konkrétny zápis deklarácie závisí na programovacom jazyku. V prípade jazyka *C*, po svojej deklarácii premenná nedisponuje žiadnym zmysluplným

obsahom (nachádza sa v nej to, čo je na príslúchajúcom pamäťovom mieste). Ak chceme aby disponovala zmysluplným obsahom hneď od jej vzniku hovoríme o tzv. inicializácii premennej hneď pri deklarácii, čomu sa hovorí definícia premennej. To znamená, že ide o priradenie počiatočnej hodnoty. Premenná vo všeobecnosti môže nadobúdať hodnoty priradením alebo načítaním.

Inicializácia premennej je prvé nastavenie konkrétnej hodnoty premennej (často býva súčasťou deklarácie ako sme uviedli vyššie). Ďalšie nastavenie hodnoty danej premennej označujeme ako priradenie hodnoty premennej.

Definícia premennej vyjadruje vyhradenie konkrétnej časti pamäte počítača, o čo sa plne stará kompilátor. Deklarácia premennej nevyhradzuje konkrétnu pamäť. Príklady:

```
int vek;                // deklaracia celociselnej premennej
vek = 18;               // priradenie hodnoty premennej
int vek = 18;           // definicia premennej
scanf("%d", &vek);      // nacitanie premennej od pouzivatela
unsigned int c, d;      // viacnasobna deklaracia premennych
```

Konštanta je objekt nadobúdajúci počas celej realizácie algoritmu jedinú konkrétnu hodnotu určitého údajového typu. V programe môžeme konštantou označiť premennú, ktorá má definovanú hodnotu už pri preklade programu a túto počas činnosti programu nemení. Prototyp na definovanie konštanty je:

const datovy_typ meno = hodnota;

```
#define PI 3.14                     // definovanie symbolickeho makra, zastarane
const double PI = 3.141598281;      // definovanie konstanty, odporucane
```

Identifikátor označuje jednotlivé objekty programovacieho jazyka. Tak ako je uvedené v publikácii Herouta (2010) je to názov objektu (premennej, konštanty, podprogramu (funkcie, procedúry) a pod.). Predstavuje postupnosť písmen a číslic a niektorých špeciálnych znakov, napr. '_', ktorá sa začína písmenom, alebo znakom '_'. Nesmie začínať číslicou. Vo väčšine programovacích jazykov nesmie identifikátor obsahovať medzery ani písmená, ktoré nie sú v anglickej abecede. Viacslovný identifikátor, ktorý by obsahoval medzeru nie je prípustný. Dĺžka identifikátora je daná typom kompilátora, *ANSI* norma odporúča maximálne 31 znakov. Jazyk *C* je „*case sensitive*“, čiže rozlišuje veľké a malé písmená, preto identifikátory *Suma* a *suma* sú rozdielne identifikátory. Je dobré voliť tzv. mnemotechnické identifikátory, teda také, ktoré približujú premennú, ktorá je v nich uložená. Napríklad premennú, do ktorej ukladáme výsledok výpočtu plochy obdĺžnika, môžeme nazvať *P1*, ale lepšie *PlochaObdlznika*.

Identifikátor sa nesmie zhodovať s niektorým z kľúčových slov programovacieho jazyka *C*. Odporúča sa na odlíšenie používať na konštanty a používateľom definované typy veľké písmená, na všetky ostatné objekty malé písmená. Takéto odporúčania závisia od použitého štýlu. Štýlov je viac, a závisia od programovacieho jazyka. Kľúčové slová jazyka *C* musia byť písané malými písmenami, inak budú vyhodnocované ako identifikátory.

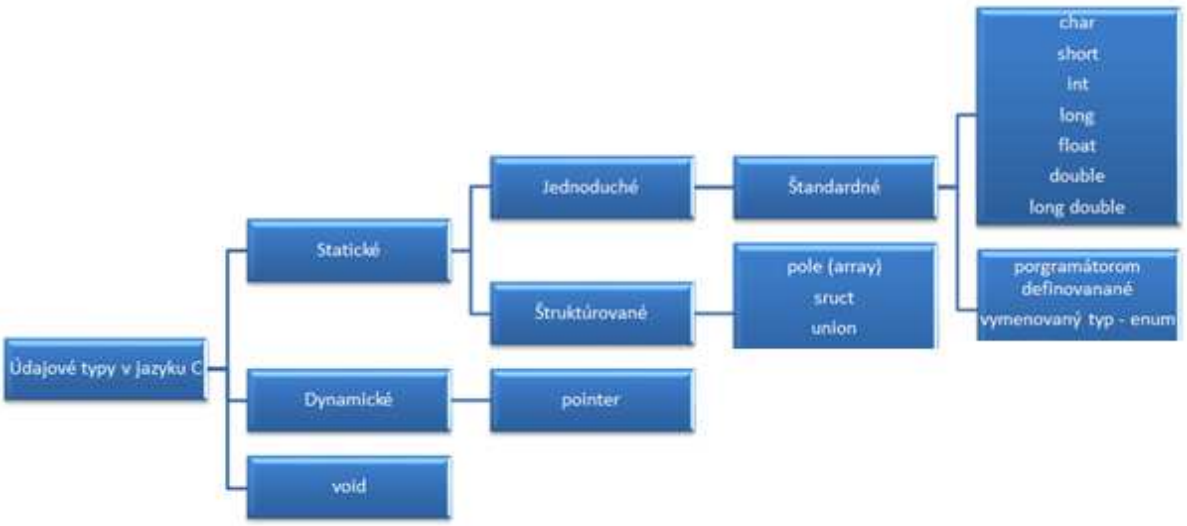
1.4.2 Dátový typ

Všetky premenné sú vždy určitého typu. To znamená, že je pre ne definovaná prípustná množina hodnôt a podľa ich typu s nimi môžeme pracovať. Medzi základné dátové typy, ktoré premenná môže nadobúdať patrí:

- **celočíselný typ** – predstavuje čísla definované na množine *Z*. Do takejto premennej potom nemôžeme vložiť výsledok delenia, pretože ten často nie je celé číslo, aj v prípade ak delíme celé čísla, ak však nie je našim cieľom pracovať s celočíselným delením,
- **reálny typ** – predstavuje číslo s desatinnou časťou, nesmieme zabúdať, že dátové typy jednotlivých programovacích jazykov môžu pracovať s rôznou presnosťou desatinných miest,
- **znakový typ** – slúži na uloženie znaku. Treba sa zamyslieť ako je prislúchajúci dátovy typ v konkrétnom programovacom jazyku reprezentovaný, pretože môže v prípade použitia takejto premennej vo výpočte vzniknúť chyba sprevádzaná výpisom chybového hlásenia,
- **typ textového reťazca** – na uloženie slov, t. j. viac ako jedného znaku,
- **logický typ** – hodnota vyjadruje stav *platí* alebo *neplatí*, *áno – nie*, *true – false*, *1 – 0*.

Na základe definovania prípustných množín hodnôt sa premennej priraďuje zodpovedajúci dátový typ v konkrétnom programovacom jazyku. Typov je samozrejme oveľa viac, sú závislé od konkrétneho programovacieho jazyka.

1.4.2.1 Typy premenných jazyka C



Obr. 5 Typy premenných jazyka C.

Zložitejšie dátové typy jazyka *C* detailnejšie rozoberáme v druhom diele tejto učebnice. Medzi jednoduché (atomické, t. j. ďalej už nedeliteľné) preddefinované údajové typy jazyka *C* patria:

- **Celočíselné ordinálne typy**

  Ordinálne typy opisujú konečnú usporiadanú množinu hodnôt. Pre každý prvok množiny je jasne určený predchodca a nasledovník okrem prvého, resp. posledného prvku množiny. Majú vždy minimálnu a maximálnu hodnotu (prvý, resp. posledný prvok množiny). Hodnoty ordinálnych dátových typov sú zobrazené do ordinálnych čísel 0, 1, 2 . . .

  Môžu byť neznamienkové = len kladné (*unsigned*, používaná prípona *u/U,* napr. 123*u*) alebo znamienkové (*signed*). Pri záporných číslach sa píše znamienko '–'.

  Celočíselné literály (**literál** je priamo zápis určitej hodnoty v programovacom jazyku) môžu byť zapísané v troch číselných sústavách:

  - desiatkovej (dekadická sústava) – postupnosť čísiel od 0 až 9, z ktorých prvé nesmie byť 0, napr. 92, 14, 0,
  - osmičkovej (oktálová sústava) – číslica 0 nasledovaná postupnosťou osmičkových číslic 0 až 7, napr. 0126, 035, 0, 05,
  - šestnásťkovej (hexadecimálna sústava) – číslica 0 nasledovaná znakom *x* (alebo *X*) a postupnosťou šestnástkových číslic 0 až 9, *a* až *f* (alebo *A* až *F*), napr. *0x58*, *0xCD*, *0x0*, *0x1*.

Odporúčanie: Pri literáloch zapísaných v šestnástkovej sústave je vhodné zvoliť si jednotný druh zápisu. Odporúčam spôsob s malým *x* a veľkými písmenami *A* až *F*, napr. *0x3B*, z dôvodu najväčšej prehľadnosti.

- **short int** (skrátene short):
  - celočíselná premenná, používa sa zvyčajne aspoň 16 bitov,
  - rozsah pre *unsigned short int* je zvyčajne 0 ÷ 65 535,
  - rozsah pre *signed short int* je zvyčajne –32 768 ÷ 32 767;
- **int**:
  - celočíselná premenná, používa sa aspoň 32 bitov,
  - rozsah pre *signed int* je zvyčajne –2 147 483 648 ÷ 2 147 483 647,
  - rozsah pre *unsigned int* je zvyčajne 0 ÷ 4 294 967 296,
  - presný rozsah je v jazyku *C* daný kompilátorom v tvare symbolických konštánt *INT_MIN*, *INT_MAX* definovaných v hlavičkovom súbore *limits.h*;
- **long int** (skrátene long):
  - celočíselná premenná, používa sa aspoň 32 bitov,
  - rozsah pre *signed long int* je zvyčajne –2 147 483 648 ÷ 2 147 483 647,
  - pre literály typu *long* používame príponu *L* alebo *l*, napr. *123l*, *123L*;
- **long long int** (skrátene long long):
  - celočíselná premenná, používa sa aspoň 64 bitov,
  - rozsah pre *signed long int* je zvyčajne –9 223 372 036 854 775 808 ÷ 9 223 372 036 854 775 807;
- **char**:
  - znaková premenná, používa sa 8 bitov = 1 B,
  - hodnota znakovej konštanty (ordinálne číslo) je odvodená z prislúchajúcej tabuľky znako - *ASCII*[1]. Veľkosť znakovej konštanty je *int*, nie *char*,

[1] *ASCII* je anglická skratka pre *American Standard Code for Information Interchange*. Ide o kódovací systém znakov anglickej abecedy, číslic, netlačiteľných znakov, iných špeciálnych znakov a riadiacich kódov slúžiacich na riadenie dátového prenosu, na formátovanie tlače, prípadne na iné účely (Encyclopedia Britannica, 2020).

- rozsah pre *signed char* je zvyčajne –128 ÷ 127,
- rozsah pre *unsigned char* je zvyčajne 0 ÷ 255,
- ohraničuje sa jednoduchými apostrofmi, napr.: 'a', '*', '\'' (na zobrazenie úvodzoviek sa používa \"),
- niektoré používané znaky majú svoje znakové vyjadrenie:

Používaný znak	Znakové vyjadrenie	Čo to znamená
\n	\0x0A	nový riadok
\r	\0x0D	návrat na začiatok riadku
\t	\0x0C	tabulátor
\\	\0x5C	zobrazí spätné lomítko
\‘	\0x2C	zobrazí apostrof
\0	\0x00	nulový=prázdny znak – na označenie ukončenia reťazca

- používa sa aj pre reťazce znakov - reťazcové literály sú ohraničené úvodzovkami, napr. "Toto je retazcova konstanta = literal" , je však pre ne potrebné deklarovať pole znakov v jazyku *C*.

- **Reálne neordinálne typy**
  - **float**:
    - reálne čísla s desatinnou bodkou, zvyčajne sa používa 4 B = 32 bitov,
    - rozsah je zvyčajne $–3{,}4 \cdot 10^{38} \div 3{,}4 \cdot 10^{38}$,
    - presný rozsah je daný kompilátorom v tvare symbolickej konštanty *FLT_MAX* definovanej v hlavičkovom súbore *float.h*,
    - literály typu *float* sa ukončujú písmenom *F* alebo *f*, napr. *2.14f*;
  - **double**:
    - reálne čísla s desatinnou bodkou, zvyčajne sa používa 8 B = 64 bitov,
    - rozsah je zvyčajne $–1{,}8 \cdot 10^{308} \div 1{,}8 \cdot 10^{308}$
    - reálne literály sú implicitne typu ***double*****,** presnosť cca 20 desatinných miest,
    - presný rozsah je daný kompilátorom v tvare symbolickej konštanty *DBL_MAX* definovanej v hlavičkovom súbore *float.h*;

- **long double**:
  - reálne čísla s desatinnou bodkou, zvyčajne sa používa 10 B ÷ 16 B,
  - rozsah je zvyčajne $-1{,}2 \cdot 10^{4932} \div 1{,}2 \cdot 10^{4932}$,
  - presný rozsah je daný kompilátorom v tvare symbolickej konštanty *LDBL_MAX* definovanej v hlavičkovom súbore *float.h*;

- **Logické hodnoty**:
  - jazyk *C* neposkytuje priamo typ *boolean*. Booleovské hodnoty sú zvyčajne reprezentované s pomocou celočíselných hodnôt, kde: nulová hodnota (0) reprezentuje hodnotu *false*, nenulová hodnota (najčastejšie však 1) hodnotu *true*.

Uvedené jednotlivé rozsahy dátových typov sú len orientačné, pretože v praxi závisia od prísluchajúceho prekladača. Ich presnú veľkosť je možné zistiť aplikáciou operátora ***sizeof.*** Napr. *sizeof(int),* ktorý vráti veľkosť v bajtoch.

Definícia nového dátového typu

Nový typ sa definuje s pomocou kľúčového slova ***typedef***, podľa prototypu:

typedef stary_typ novy_typ;

Predstavte si, že chceme definovať typ s názvom *RETAZEC*, ako pole 100 znakov. Urobíme to nasledovne:

```
typedef char RETAZEC[100];
```

Teraz už môžem typ *RETAZEC* používať v programe:

```
int main(void)
{
    RETAZEC a, b; // deklaracia premennych a, b typu RETAZEC
    ...
    return 0;
}
```

Vymenovaný (enumeračný) typ

Vymenovaný typ slúži na definovanie zoznamu symbolických konštánt. Položky zoznamu sú vlastne celočíselné konštanty. Konštanty možno použiť všade tam, kde štandardné celočíselné hodnoty. Hodnoty jednotlivých symbolických konštánt začínajú 0 a postupne sa zväčšujú o 1.

Je ich možné nastaviť aj na špecifické celočíselné hodnoty. Použitie takéhoto typu často sprehľadňuje program. Definícia sa robí s pomocou kľúčového slova **enum**:

enum MenoTypu {ZoznamKonstant} ZoznamPremennych;

Príklad použitia vymenovaného typu na dni v týždni spolu s definíciou premennej *den*. V hlavnom programe je možné priradiť hodnotu buď priamo s pomocou konštanty (pozor, neuvádzame do úvodzoviek), alebo jej číselnou hodnotou.

```
enum dni_v_tyzdni
{
    Pondelok,
    Utorok,
    Streda,
    Stvrtok,
    Piatok
} den;

int main(void)
{
    den = Pondelok;
    den = 0;            // je jedno, ktore priradenie sa pouzije, je to to iste
}
```

Ten istý príklad, ale definícia premennej sa deje v hlavnom programe. Pritom používame špecifické číslovanie konštánt, začíname od 1.

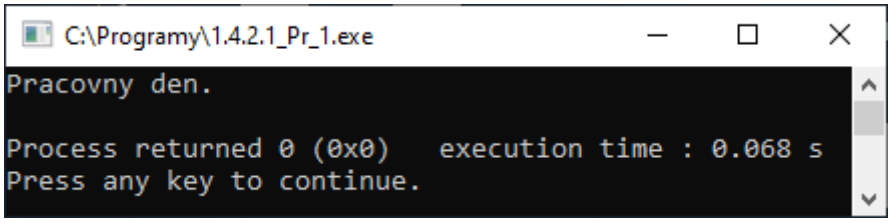
```
enum dni_v_tyzdni
{
    Pondelok = 1,
    Utorok,
    Streda,
    Stvrtok,
    Piatok
};

int main(void)
{
    enum dni_v_tyzdni den;
    den = Pondelok;
    den = 1;     // je jedno, ktore priradenie sa pouzije, je to to iste
}
```

Hodnotu premennej enumeračného typu je možné priradiť do celočíselnej premennej.

```
int i_den = den;
printf("%d\n", i_den); // vypise sa cislo 1
```

Ukážka praktickejšieho použitia enumeratívneho typu, *1.4.2.1_Pr_1.c*:

```c
#include <stdio.h>

typedef enum
{
    pondelok = 1,
    utorok,
    streda,
    stvrtok,
    piatok,
    sobota = 0,
    nedela = 0
} DEN;

int main(void)
{
    DEN dnes = utorok;
    switch (dnes)
    {
    case pondelok:
    case utorok:
    case streda:
    case stvrtok:
    case piatok:
        printf("Pracovny den.\n");
        break;
    case nedela:
        printf("Vikend.\n");
        break;
        //case sobota: sobota == nedela, v switch staci iba raz!
    }
    return 0;
}
```



Obr. 6 Konzolový výstup programu 1.4.2.1_Pr_1.c.

Štruktúrované dátové typy nie sú atomické, ale reprezentujú vždy niekoľko položiek buď rovnakého, alebo rôzneho typu. Prázdny dátový typ (***void***) sa zvyčajne využíva pri funkciách, resp. v súvislosti s pointrom. Viac v druhom diele tejto učebnice.

1.4.3 Operátory a štandardné funkcie

Takmer každý programovací jazyk ponúka základné matematické (+, –, ., /), relačné (=, <, >, <=, >=, <>) a logické operátory (*AND*, *OR*, *NOT* a pod.), ako aj množstvo štandardných

(tzv. vstavaných/knižničných) funkcií, ktoré môže programátor bez toho aby ich definoval využívať. V prípade ak chcete využívať vstavané funkcie, musíte vložiť (*include*) prislúchajúce tzv. hlavičkové súbory (*header file* *.*h*). Napríklad v prípade volania knižničných vstupno--výstupných funkcií vkladáme hlavičkový súbor *stdio.h* s pomocou príkazu:

```
#include <stdio.h>
```

V prípade reprezentácie algoritmu by sme tento nemali viazať na žiadny konkrétny programovací jazyk a využívať operátory a funkcie vychádzajúce z matematiky. Operátor je symbol indikujúci operáciu, ktorá má byť realizovaná na istom počte elementov. Tieto elementy sa nazývajú operandy. Často sa používajú v priraďovacích a podmieňovacích výrazoch. Podľa počtu operandov ich delíme na:

- unárne,
- binárne,
- ternárne.

Podľa **funkčnosti** ich delíme na:

1. **Aritmetické operátory**

   Používajú sa v bežných aritmetických výpočtoch v takmer rovnakej podobe, ako ich poznáme z matematiky. Tiež priorita operátorov násobenia a delenia je vyššia ako sčítania a odčítania. Podľa počtu operandov rozoznávame:

   Unárne operátory (jeden operand)

   - + a –, použitie `i = +1; k = -j;` `// pisu sa pred operand - prefixovy zapis`
   - **operátory inkrementácie a dekrementácie**

     Používajú sa pri inkrementácii (zväčšenie o 1) ++, resp. dekrementácii (zníženie o 1) − − danej premennej. Veľmi dôležité je vedieť rozdiel medzi prefixovým a postfixovým zápisom. Napr.:

     - **++x** premenná x je inkrementovaná pred použitím (prefixový zápis)
     - **x++** premenná x je inkrementovaná po použití (postfixový zápis)
     - **--x** premenná x je dekrementovaná pred použitím (prefixový zápis)
     - **x--** premenná x je dekrementovaná po použití (postfixový zápis)

Pozn. autora: Použitie je možné len s premennou, t. j. *l-hodnotou*[2]. Časté chybné použitie je napr. 45++ alebo *– –(i + j).*

Príklady:

```
int i = 5, j = 1, k;
i++;              // i=6
j = ++i;          // j=7, i=7
j = i++;          // j=7, i=8
k = --j + 2;      // k=8, j=6
```

Binárne operátory (dva operandy)

- **sčítanie** + c = a + b;
- **odčítanie** – x = 1 – y;
- **násobenie** * o = PI * d;
- **reálne delenie** / 3.0 / 2 // výsledok je 1.5

  Pozn.: O reálne delenie ide v prípade, keď sú delené reálne čísla, teda aspoň jeden z operandov nie je celočíselného typu.
- **celočíselné delenie (DIV)** / 3 / 2 // výsledok je 1
- **delenie modulo (MOD)** % 5 % 2 // výsledok je 1 (zvyšok po delení)

Pozn.1: To, či bude výsledok delenia celočíselný, alebo reálny závisí od typu operandov – nastáva tam implicitná typová konverzia. V prípade, že je aspoň jeden z operandov typu *float* alebo *double*, tak ide o reálne delenie.

Pozn.2: Pri aritmetických operáciách môže nastať pretečenie, resp. podtečenie, kedy dostaneme matematicky nesprávny výsledok, pričom kompilátor v takomto prípade žiadnu chybu neohlási. Táto chyba vzniká len pri celých číslach. Pri práci s reálnymi číslami vzniká chyba sprevádzaná chybovým hlásením. V prípade, že je nevyhnutné zachovať dátové typy premenných, tak je veľmi zložité sa proti takýmto chybám brániť. V opačnom prípade je riešením použitie dátového typu s väčším rozsahom. Jediný prípad, kedy pri aritmetických operáciách kompilátor ohlási chybu, je v prípade delenia 0. V tomto prípade vznikne výnimka.

---

[2] Pri programovaní v jazyku *C* sa často pracuje s pojmom *l-hodnota* (*l-value*). *L-hodnota* predstavuje adresu, t. j. napríklad premenná *i* je *l-hodnotou*, ale konštanta (123) alebo výraz (*i* + 3) nie je *l-hodnotou*. *L-hodnota* je to, čo môže byť na ľavej strane priradenia.

2. Logické operátory

Používajú sa v logických, ako aj v podmieňovacích výrazoch. V jazyku *C*, ktorý nemá implicitne definovaný logický typ *boolean* je výsledok typu *int*, kde 0 znamená *false* (nepravda) a nenulová hodnota (najčastejšie 1, ale nie je to podmienka) znamená *true* (pravda). Rozlišujeme:

- Unárne: **negácia** **NOT** **!**
- Binárne: **logický súčin** **AND** **&&**
  logický súčet **OR** **||**

Belan (2013) tento koncept objasňuje takto: „Niekedy je nevyhnutné v jednej podmienke otestovať viacero vecí naraz. Napríklad potrebujete zistiť, či je premenná *i* väčšia ako –10 a menšia ako 10. Každú vec zvlášť viete zistiť jednoducho $i > -10$, $i < 10$. Ak chcete zistiť či sú splnené obidve naraz, použije sa operátor &&. Takže podmienka, ktorá zistí, či je hodnota v premennej *i* súčasne väčšia ako −10 a menšia ako 10 vyzerá: $i > -10$ && $i < 10$. Ak potrebujeme zistiť, či je splnená aspoň jedna z podmienok, použijeme operáciu logického súčtu ||. Takže výraz $i > 0$ || $j > 0$ nadobúda hodnotu „nepravda“ jedine vtedy, ak sú hodnoty oboch premenných *i* a *j* menšie ako nula alebo rovné ako nula. Stačí, keď je hodnota v jednej z premenných kladná (pokojne môžu byť aj obidve), podmienka nadobudne hodnotu *pravdy“*.

Pri použití logických operátorov && a || môže mať výsledná hodnota logických výrazov určená na základe logickej hodnoty prvého operanda (ide o tzv. **skrátené vyhodnocovanie logických výrazov – *short circuit***). To znamená, že argumenty sú vyhodnocované zľava doprava a len čo je možné určiť konečný výsledok, vyhodnocovanie okamžite skončí. To znamená pri logickej konjunkcii: x && y, ak je hodnota prvého operanda 0 (nepravda), druhý operand nie je testovaný. Pri logickej disjunkcii: x || y, ak je hodnota prvého operanda 1 (pravda), druhý operand nie je testovaný. Inými slovami je možné povedať, že skrátené vyhodnocovanie znamená, že operandy výrazu sú vyhodnocované zľava doprava a len čo je možné určiť konečný výsledok, vyhodnocovanie je okamžite ukončené. Toto má význam najmä pri zložených logických výrazoch, kde vďaka tomu môže byť výpočet urýchlený. V prípade logického súčinu je potom vhodné ako prvý výraz uvádzať najpravdepodobnejšiu nepravdu, v prípade logického súčtu najpravdepodobnejšiu pravdu. Napríklad použitie výrazu v podmienke

if(*y* != 0 && *x* / *y* < *z*) je absolútne korektné, pretože delenie nulou nikdy nenastane (čo zabezpečí prvá časť podmienky).

Belan (2013) uvádza: „Ak chceme hodnotu logického výrazu zmeniť na opačnú, používa sa na to operátor *!*. Ak napríklad chceme napísať podmienku „v premennej znak nie je veľké písmeno“, teda podmienku, ktorá nadobudne hodnotu *pravda* ak v tej premennej veľké písmeno nie je a hodnotu *nepravda* inak, budeme postupovať takto: Najprv si napíšeme podmienku, ktorá nám zistí, či v premennej *znak* veľké písmeno je. Taká podmienka môže vyzerať napríklad takto: *znak* >= 'A' *&& znak* <= 'Z'. Táto podmienka však funguje presne naopak, ako potrebujeme. Ale vďaka operátoru *!* z nej vyrobíme podmienku, ktorú potrebujeme: !(*znak* >= 'A' *&& znak* <= 'Z')“.

3. Relačné operátory

Používajú sa v logických a podmieňovacích výrazoch, najčastejšie na porovnávanie číselných hodnôt. Dátovým typom návratovej hodnoty relačných operátorov je *int*.

- **rovnosť** = =
- **nerovnosť (rôznosť)** !=
- **menšie** <
- **menšie alebo rovné** <=
- **väčšie** >
- **väčšie alebo rovné** >=

Pozor na častú chybu pri porovnávaní: *i* = 5 a *i* = = 5 je rozdiel. V prvom prípade sa do premennej *i* uloží hodnota 5, v druhom prípade ide o testovanie rovnosti obsahu premennej *i* s hodnotou 5.

Príklad: Aký bude výstup programu? (Belan, 2013)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 3, j = 4;
6      printf("%d\n", i == 3);    // 1 = pravda, i je rovne 3
7      printf("%d\n", j >= 6);    // 0 = nepravda, j=4 co nie je vacsie ani rovne 6
8      printf("%d\n", j = 5);     // 5 ide o priradenie
9      printf("%d\n", j < 6);     // 1 = pravda, j=5 co je mensie ako 6
10     printf("%d\n", j == 4);    // 0 = nepravda, premenna j = 5, priradenie na riadku 8
```

```
11
12      return 0;
13  }
```

Pozn.: V jazyku *C* majú aritmetické operátory a operátor porovnania vyššiu prioritu ako logické operátory, takže výraz v úlohe podmienky:

- *if* (*c* >= 'A' && *c* <= 'Z')

  je správny. Ak si však nie ste istý prioritou vykonávania vždy používajte zátvorky

  if ((*c* >= 'A') && (*c* <= 'Z'))

4. Priraďovacie operátory

Základný priraďovací operátor je =, ktorý môžeme využívať aj pre viacnásobné priradenie, napr. $k = j = i = 2$; ktoré sa vyhodnocuje sprava doľava: $k = (j = (i = 2))$; Toto je možné vďaka tomu, že priradenie je výraz. V tejto súvislosti sa často stretáme s pojmom *l-hodnota* (*l-value*). Pod týmto pojmom si môžeme predstaviť niečo, čo má adresu v pamäti, najčastejšie je to teda premenná. *L-hodnota* je to, čo môže ležať na ľavej strane priradenia.

Medzi zložené priraďovacie operátory patria:

- **premenná += výraz** je to isté ako **premenná = premenná + výraz**
- to isté platí pre **odčítanie −=, násobenie *=, delenie /=, zvyšok po delení %=**

Používajú sa na zjednodušenie (zostručnenie) zápisu najbežnejších priraďovacích príkazov.

Príklad:

```
int i = 4, j = 3;
j += i;     // j = 7
j /= --i;  // i = 3, j = 2 - ide o celociselne delenie, pretoze oba operandy su typu int
j *= i - 2 // j = 2, j = j * (i - 2) = 2 nie j = j * i - 2 = 4
```

5. Bitové operátory

Slúžia na manipuláciu s jednotlivými bitmi. Tieto operácie nie sú príliš využívané, ale môžete sa s nimi stretnúť v počítačovej grafike, pri práci so zvukmi a v aplikáciách ktoré niečo riadia. Môžu byť aplikované iba na operandy integrálnych dátových typov *char*, *short*, *int* a *long* (so znamienkom i bez). Jazyk *C* rozoznáva tieto bitové operátory:

- **&** Uskutočňuje bitový súčin (konjunkciu) (*AND*) *x & y*

- | Uskutočňuje bitový súčet (disjunkciu) (*OR*) *x | y*
- ^ Uskutočňuje bitový exkluzívny súčet (nonekvivalenciu) (*XOR*) *x ^ y*
- ~ Uskutočňuje bitový doplnok *~x, ~y*
- << Uskutočňuje posun bitov v binárnych hodnotách doľava
- >> Uskutočňuje posun bitov v binárnych hodnotách doprava

x	*y*	*x & y*	*x | y*	*x ^ y*	*~x*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Príklad:

int i = 7 & 9; // *i* bude 1

0111
&1001
0001

int i = 7 | 9; // *i* bude 15

0111
&1001
1111

6. **Ternárny operátor –** ktorý by mal mať okolo seba tri operandy, ale keďže toto by bolo ťažké zapísať, pozostáva tento z dvoch znakov medzi ktorými je operand (bližšie vysvetlené v kapitole 1.5 Základné riadiace konštrukcie).

7. **Operátor čiarky**

Operátor čiarky je operátorom postupného vyhodnotenia. Obdobne ako operátor logický súčin (&&), logický súčet (||) a ternárny operátor (? :) zaručuje vyhodnotenie ľavého operanda pred vyhodnotením pravého operanda.

Syntax operátora čiarky: vyraz1, vyraz2

Ako uvádza Herout (2010) spracováva sa tak, že sa vyhodnotí *vyraz1* a výsledok tohto vyhodnotenia je zabudnutý, následne sa vyhodnotí *vyraz2* a to je záverečný výsledok po použití výrazu s operátorom čiarky. Výsledok nie je *l-hodnota*.

Príklad:

```
int i = 2, j = 4;       // toto nie je operator ciarky, ide o definiciu premennych
j = (i++, i - j);       // i bude 3 a j bude -1
```

Čiarkou môžeme oddeliť jednotlivé výrazy v mieste, kde je očakávaný jediný výraz. Najčastejšie použitie operátora čiarky je v riadiacich príkazoch *for* a *while*, napr.:

```
for (i = 0, j = 0; i < MAX; i++, j++)
```

1.4.4 Výraz, príkaz, priradenie, blok

Výraz (*expression*) je predpis obsahujúci konštanty, premenné, literály a spôsob ich spracovania s pomocou operátorov a funkcií, podobných napríklad tým, ktoré poznáme z matematiky. Výsledkom je hodnota prislúchajúceho typu, ktorá vznikne po vykonaní vo výraze naznačeného spracovania. Inými slovami je to niečo, čo sa dá vypočítať, napr. $i + 10$, alebo $f > 11{,}4$.

Príkaz (*statement*) je konkrétna operácia (akcia) zapísaná v prislúchajúcom programovacom jazyku, ktorá prikazuje procesoru vykonávať isté presne stanovené činnosti. To znamená, že sú to tie časti, ktoré počas vykonávania programu spracúvajú dáta (na rozdiel od deklarácií). Príkazy v jazyku C sú oddeľované bodkočiarkou "*;*". Základným príkazom je **príkaz priradenia (*assigment*)**, ktorý má tvar *l-hodnota = výraz;* (*l-value = expression;*). Ako výraz môže byť použitá konštanta, premenná, literál, funkcia a ich ľubovoľná aj viacnásobná kombinácia oddelená operátormi. Napr. ***p = v;***

- *l-value* je niečo čo má adresu v pamäti, a teda môže tomu byť priradená hodnota,
- kde *p* je meno premennej (identifikátor),
- *v* je výraz – predpis,
- vykonaním príkazu nadobudne hodnota premennej *p* hodnotu výrazu *v* umiestneného na pravej strane priradenia,
- predchádzajúca hodnota tejto premennej je nenávratne stratená.

Nesmieme zabúdať, že samotná bodkočiarka predstavuje prázdny príkaz (*null statement*). Tento sa často používa v súvislosti s cyklami – cyklus bez tela.

Príklady:

Čo je čo?

```
int i, j;        // deklaracia celociselnych premennych i a j
i = 10;          // priradenie - zakladne
j = i + 10;      // priradenie na zaklade vyrazu, j = 20
float f;         // deklaracia realnej premennej f
float f = 3.14; // definicia realnej premennej f, t. j. deklaracia + inicializacia
```

Akú hodnotu nadobudne premenná *i*?

```
int i = 10.5;    // i = 10, nastáva implicitná typová konverzia
```

Zložený príkaz verzus blok

Ak potrebujeme vykonať nie jeden, ale viac príkazov, tieto musíme uzatvoriť do jedného **zloženého príkazu** s pomocou zložených zátvoriek **{ }**. Príkazy v zložených zátvorkách sa vykonávajú v postupnosti, ako sú zapísané, ale navonok to vyzerá, akoby išlo o jeden príkaz. **Blok** je navyše od zloženého príkazu doplnený deklaráciou/definíciou premenných.

Toto je blok, lebo obsahuje deklaráciu premennej *i*.

```
{
    int i;
    i = 5;
    j = 2;
}
```

Toto je zložený príkaz, pretože neobsahuje žiadne definície premenných.

```
{
    i = 5;
    j = 2;
}
```

Zaujímavosťou je, že v blokoch môžu byť použité premenné s rovnakým identifikátorom (menom) ako v inom bloku. Dôvodom je, že na začiatku každého bloku môžeme definovať lokálne premenné. Napr.:

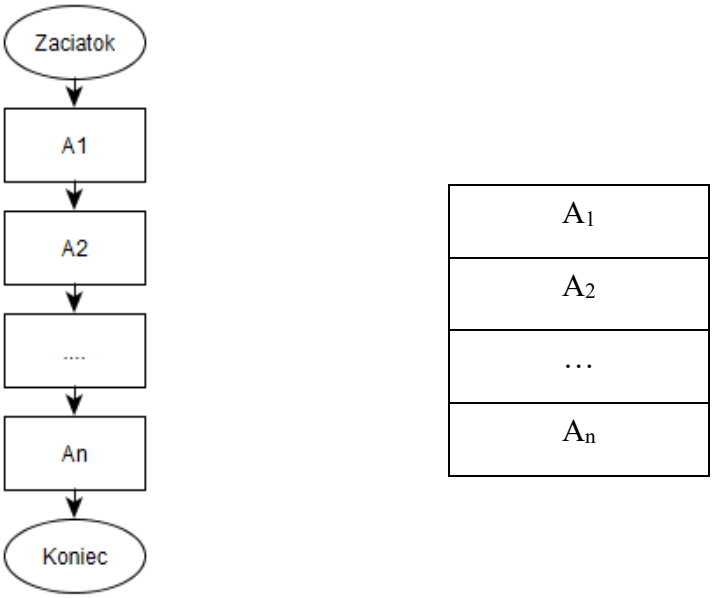
```
int main(void)
{
    int a;
    {
        int a; // ina premenna a, ako je deklarovana vyssie
        int b;
    }
    // tu uz nie je mozne pouzit premennu b
    return 0;
}
```

1.5 Základné riadiace konštrukcie

Kľúčovou otázkou kompozície riešenia algoritmu, aj programu zostáva spôsob riadenia jednotlivých akcií, ktoré ho tvoria. Riešenie je možné opísať obmedzeným počtom riadiacich konštrukcií resp. štruktúr. Medzi základné riadiace konštrukcie (príkazy) patrí sekvencia, podmienka a cyklus.

1.5.1 Sekvencia

Sekvencia predstavuje postupnosť akcií, ktoré sa vykonajú v poradí, v akom sú zapísané, ak explicitne nie je stanovené iné poradie. Na obrázku je postupnosť akcií, pričom akcia A_{i+1} sa vykoná bezprostredne po ukončení akcie A_i. Napr. priložte kartu k čítačke (jednoduchý príkaz), zadajte heslo a pod.



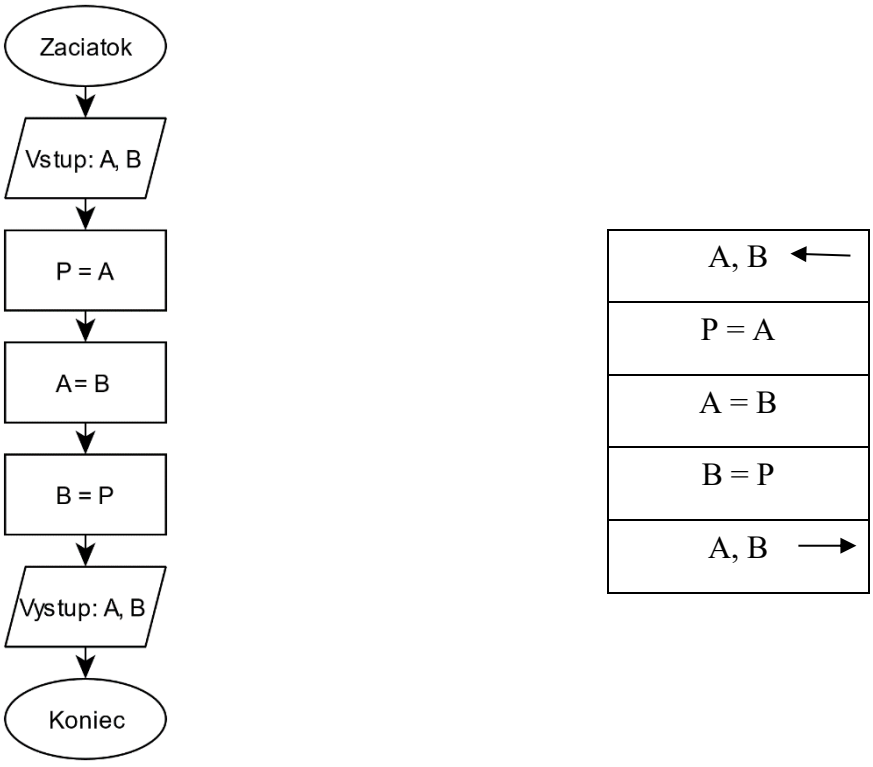
Obr. 7 Zakreslenie sekvencie s pomocou vývojového diagramu a NS diagramu.

Pozn. autora: Vývojový diagram je definovaný ako orientovaný graf, preto je nevyhnutné zakresľovať aj začiatok aj koniec. NS diagram pozostáva z blokov, a je absolútne zrejmé, kde algoritmus začína a kde končí. Je preto možné zakreslenie začiatku a konca vynechať. Viac o grafickej reprezentácie algoritmov sa čitateľ dozvie v kapitole 1.6.3.

Príklad:

Zostavte algoritmus, ktorý zabezpečí výmenu dvoch hodnôt uložených v premenných *A*, *B*. To znamená, že na výstupe chceme, aby premenná *A* nadobudla hodnotu premennej *B* a premenná *B* hodnotu premennej *A*. Napríklad ak na vstupe zadáme $A = 1$, $B = 2$, aby na výstupe bolo $A = 2$, $B = 1$.

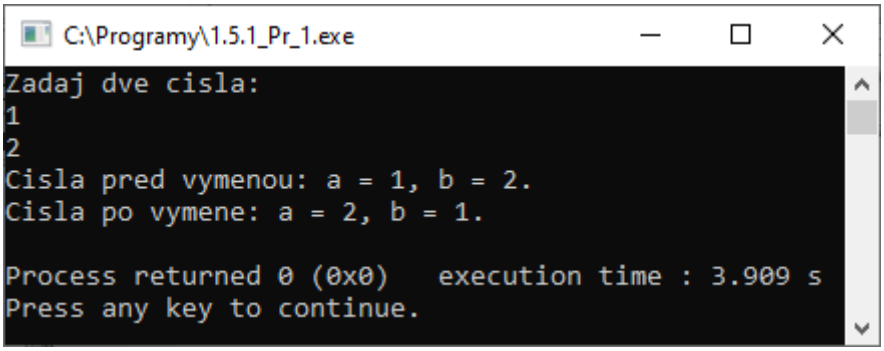
Postup: Ak na vstupe zadáme hodnoty 1 a 2, potom po prvom kroku sú hodnoty v premenných: $A = 1$, $B = 2$. Aby sa výmena hodnôt premenných korektne vykonala, použijeme pomocnú premennú *P,* do ktorej uložíme hodnotu premennej *A*. Po druhom kroku: $P = 1$, $A = 1$, $B = 2$. Teraz môžeme bez straty obsahu premennej *A*, do nej uložiť hodnotou premennej *B*, pretože túto máme zapamätanú v pomocnej premennej *P*, po treťom kroku dostávame: $P = 1$, $A = 2$, $B = 2$. Následne hodnotu z pomocnej premennej *P* uložíme do premennej *B*, po štvrtom kroku dostávame: $P = 1$, $A = 2$, $B = 1$. V piatom kroku môžeme vypísať obsah premenných *A* a *B*, $A = 2$, $B = 1$. Môžeme konštatovať, že je splnená výstupná podmienka algoritmu: *A* a *B* majú vymenené hodnoty.



Obr. 8 Zakreslenie sekvencie výmeny hodnôt dvoch premenných s pomocou vývojového a NS diagramu.

Jedna z možných implementácií *1.5.1_Pr_1.c*:

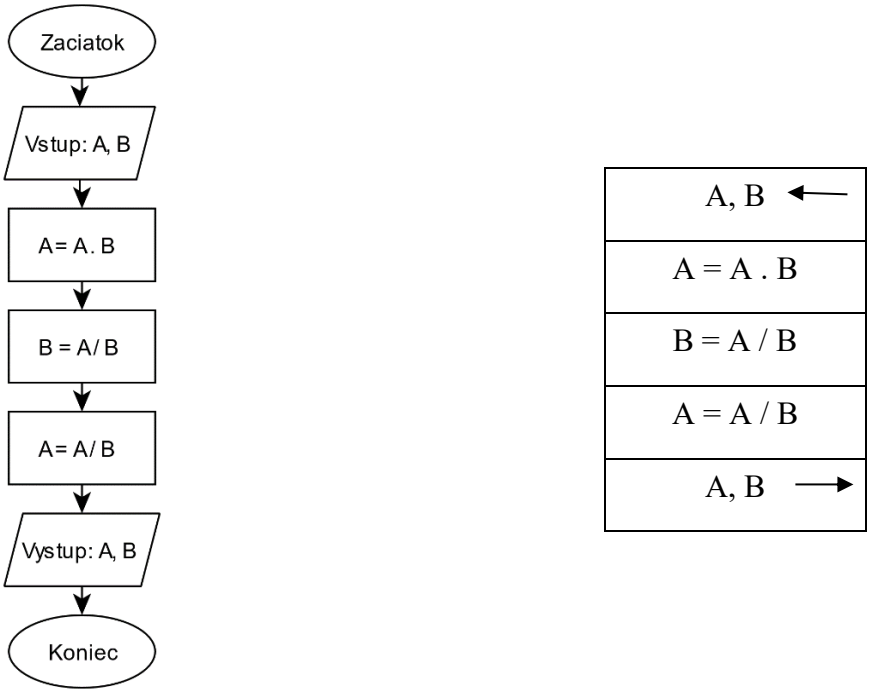
```c
#include <stdio.h>

int main(void)
{
    int a, b, p;

    printf("Zadaj dve cisla:\n");
    scanf("%d %d", &a, &b);
    printf("Cisla pred vymenou: a = %d, b = %d.\n", a, b);
    p = a;
    a = b;
    b = p;
    printf("Cisla po vymene: a = %d, b = %d.\n", a, b);

    return 0;
}
```



Obr. 9 Konzolový výstup programu 1.5.1_Pr_1.c.

Pozn. autora: Všimnite si, že pri reprezentácii algoritmu často používam na označenie premenných kapitálky a v programe malé písmená, alebo kombináciou malých a veľkých znakov pri viacslovných identifikátoroch. Je to preto, že medzi programátormi platí konvencia, ktorá hovorí, že na zápis identifikátorov premenných by sa mali používať malé znaky, alebo kombinácia malých a veľkých znakov. Veľké znaky sa využívajú na vlastné typy, alebo konštanty. Pri algoritmickom spracovaní žiadna takáto konvencia nie je známa, preto je to vždy na autorovi. Odporúčam však dodržiavať jednotný štýl.

Upravte algoritmus tak, aby zabezpečil výmenu dvoch hodnôt premenných A, B bez použitia pomocnej premennej. Tento algoritmus je postavený na myšlienke, že po načítaní hodnôt do premenných A a B, sa zrealizuje súčin týchto hodnôt, ktorý sa uloží do premennej A. Do premennej B sa následne uloží výsledok po reálnom delení premenných A a B a potom sa do premennej A uloží výsledok po reálnom delení premenných A a B, čím dosiahneme výmenu hodnôt dvoch premenných bez použitia pomocnej premennej.



Obr. 10 Zakreslenie sekvencie výmeny hodnôt dvoch premenných bez pomocnej premennej s pomocou vývojového diagramu a NS diagramu.

Jedna z možných implementácií *1.5.1_Pr_2.c*:

```c
#include <stdio.h>

int main(void)
{
    int a, b;

    printf("Zadaj dve cisla:\n");
    scanf("%d %d", &a, &b);
    printf("Cisla pred vymenou: a = %d, b = %d.\n", a, b);
    a = a * b;
    b = a / b;
    a = a / b;
    printf("Cisla po vymene: a = %d, b = %d.\n", a, b);

    return 0;
}
```

```
C:\Programy\1.5.1_Pr_2.exe
Zadaj dve cisla:
1
2
Cisla pred vymenou: a = 1, b = 2.
Cisla po vymene: a = 2, b = 1.

Process returned 0 (0x0)   execution time : 3.385 s
Press any key to continue.
```

Obr. 11 Konzolový výstup programu 1.5.1_Pr_2.c.

Pozn. autora: Chápte tento problém ako ilustračný. Pri programovaní na výmenu hodnôt používame vždy pomocnú premennú, ako je uvedené vyššie.

1.5.2 Podmieňovací príkaz

Príkazy vetvenia umožňujú programátorovi riadiť spracovanie programu na základe vyhodnotenia podmienky alebo hodnoty premennej. Napríklad ak máte menej ako tri položky tovaru, ktorý si chcete zakúpiť, choďte do pokladne vľavo, inak vpravo (vetvenie). Zápis **úplnej podmienky** v jazyku *C*:

```
if (podmienka)
  prikaz1;
else
  prikaz2;
```

if a *else* sú kľúčové slová jazyka *C*. Podmienka sa musí vždy uzatvoriť do okrúhlych zátvoriek. Ako podmienka môže byť použitý ľubovoľný výraz. V prípade, že je výsledkom podmienky pravda (v jazyku *C* je pravda chápaná ako hodnota 1 alebo iná nenulová hodnota, všeobecne *true*), vykoná sa *prikaz1*. Ak je výsledkom podmienky nepravda (v jazyku *C* hodnota nula, všeobecne *false*), vykoná sa *prikaz2*. V prípade, že je nevyhnutné pri splnení resp. nesplnení podmienky vykonať viac ako jeden príkaz, tak je nevyhnutné uzatvoriť ich do zložených zátvoriek, čiže vytvoriť zložený príkaz.

```
if (podmienka)
{
   prikaz1;
   prikaz2;
}     // tu nie je bodkociarka
else
{
   prikaz3;
   prikaz4;
}
```

V prípade potreby môžeme využívať aj **neúplnú podmienku**, ktorá v prípade nesplnenia podmienky iba preskočí vykonanie *prikazu1*, to znamená, že za kľúčovým slovom *if* nasleduje podmienka uzavretá v zátvorkách. Potom nasleduje kus kódu v zložených zátvorkách (zložené zátvorky treba uviesť v prípade, ak sa má vykonať viac než jeden príkaz, odporúča sa však zložené zátvorky uvádzať vždy), ktorý sa vykoná len vtedy, keď je podmienka splnená. Opačné nastavenie podmienky (t. j. vykonanie časti kódu len pri nesplnení podmienky) nie je možné. Všeobecný tvar:

```
if (podmienka)
  prikaz1;
```

alebo

```
if (podmienka)
{
   prikaz1;
   prikaz2;
};
```

Viacnásobná (zložená) podmienka umožňuje vetvenie na základe vyhodnotenia viacerých podmienok. Opätovne platí, že v prípade ak sa má pri splnení alebo nesplnení podmienky vykonať viacero príkazov, tak tieto je nevyhnutné uviesť ako zložený príkaz. Všeobecný tvar:

```
if (podmienka)
 prikaz1;
else if (podmienka)
 prikaz2;
else if (podmienka)
 prikaz3;
     . . .
else
 prikazN;
```

Príklad:

Určte hodnotenie v rozmedzí A – E na základe získaného počtu bodov, ak hodnotenie A je udelené v prípade dosiahnutia 94 bodov a viac, hodnotenie B v prípade ak študent získal 84 bodov a viac, ale menej ako 94, hodnotenie C v prípade ak študent získal 72 bodov a viac, ale menej ako 84, hodnotenie D v prípade ak študent získal 62 bodov a viac, ale menej ako 72, hodnotenie E v prípade ak študent získal 56 bodov a viac, ale menej ako 62. Hodnotenie sa neudelí v prípade, že študent získa menší počet ako 56 bodov.

Vstup (vstupné premenné):

N – počet bodov

Výstup (výstupné premenné):

ZNAMKA – hodnotenie študenta na základe získaného počtu bodov

Vstupné podmienky:

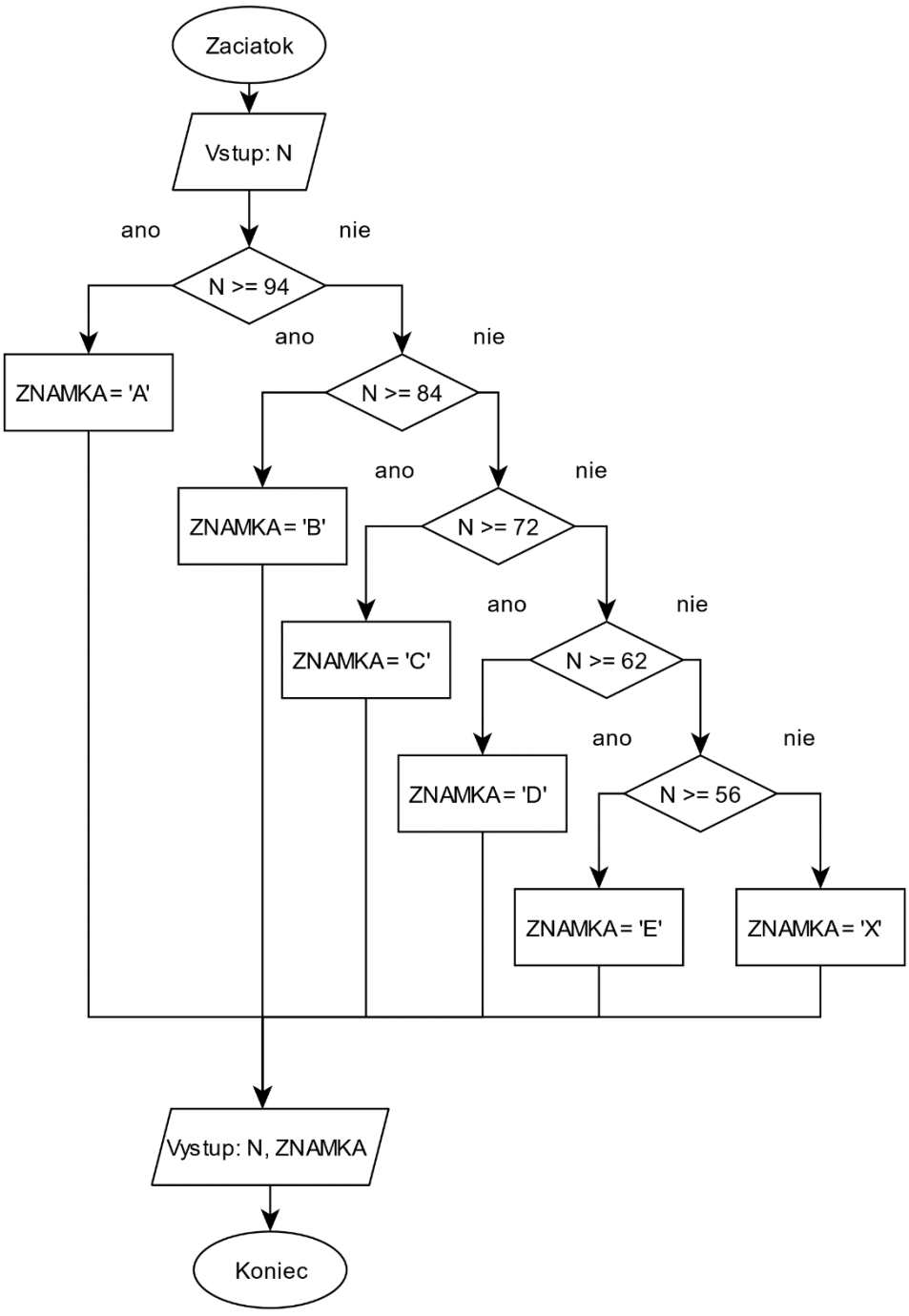
$N \in Z_0^+$

Výstupné podmienky:

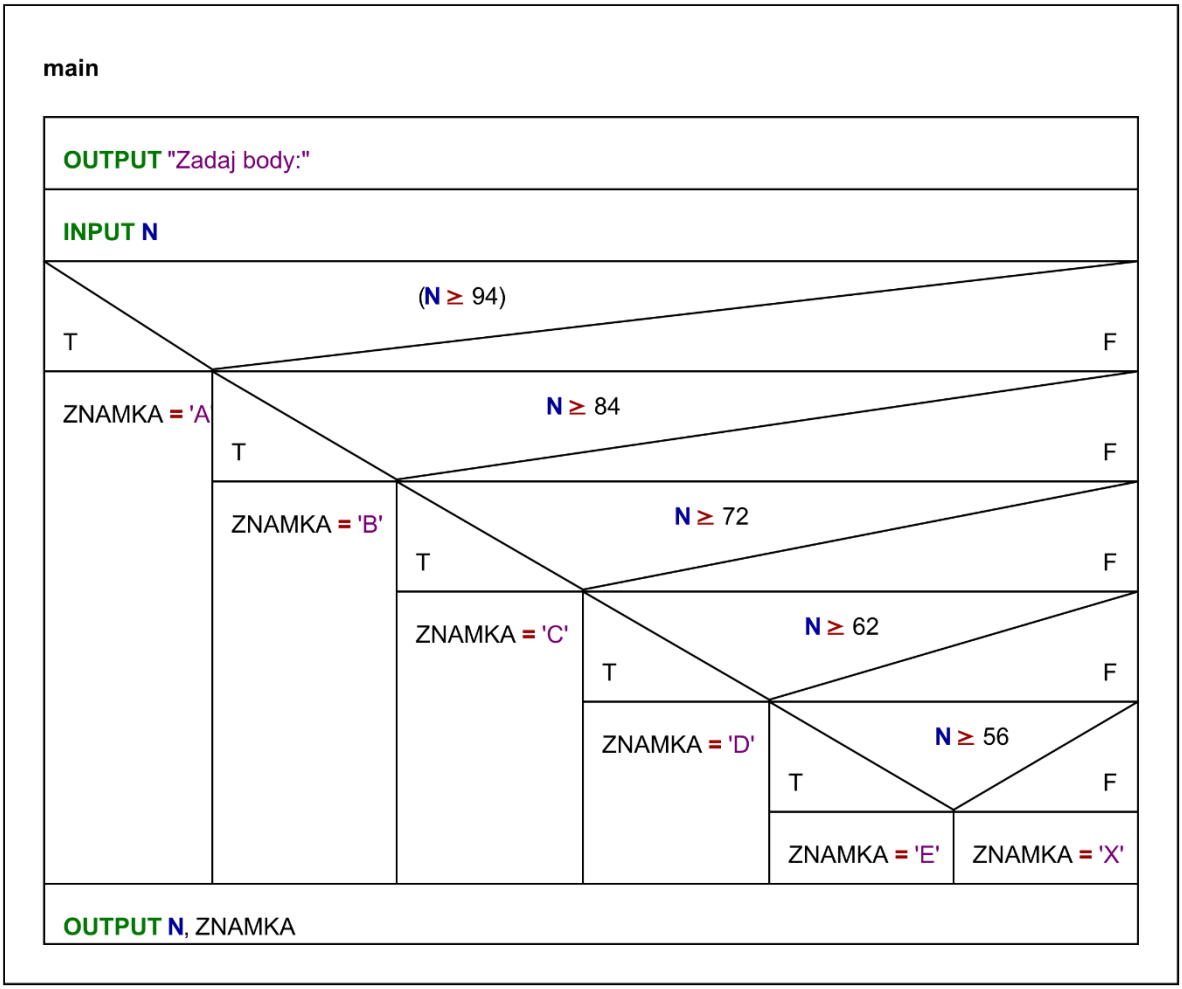
$ZNAMKA = \{A, B, C, D, E, X\} \wedge ZNAMKA = A$ ak $N \in \langle 100, 94\rangle \wedge ZNAMKA = B$ ak

$N \in (94, 84\rangle \wedge ZNAMKA = \text{C}$ ak $N \in (84, 72\rangle \wedge ZNAMKA = \text{D}$ ak $N \in (72, 62\rangle \wedge$

$\wedge ZNAMKA = \text{E}$ ak $N \in (62, 56\rangle \wedge ZNAMKA = \text{X}$ ak $N \in (56, 0\rangle$



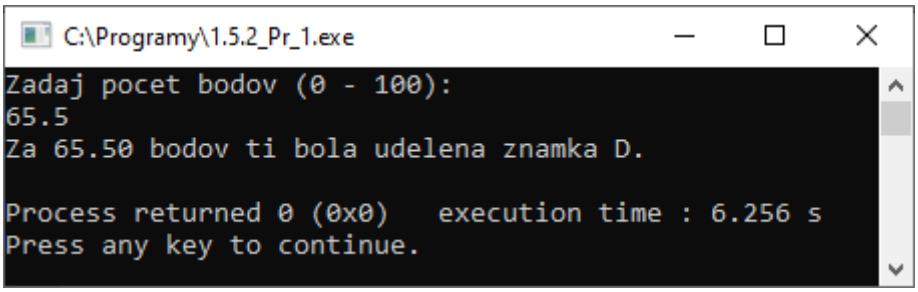
Obr. 12 Zakreslenie definovaného problému s pomocou vývojového diagramu.



Obr. 13 Zakreslenie definovaného problému s pomocou NS diagramu.

Jedna z možných implementácií 1.5.2_Pr._1.c:

```c
#include <stdio.h>

int main(void)
{
    float n;
    char znamka;

    do
    {
        printf("Zadaj pocet bodov (0 - 100):\n");
        scanf("%f", &n);
    }
    while (n < 0 || n > 100);

    if (n >= 94)
        znamka = 'A';
    else if (n >= 84)
        znamka = 'B';
    else if (n >= 72)
        znamka = 'C';
    else if (n >= 62)
        znamka = 'D';
    else if (n >= 56)
        znamka = 'E';
    else
        znamka = 'X';

    printf("Za %.2f bodov ti bola udelena znamka %c.\n", n, znamka);

    return 0;
}
```



Obr. 14 Konzolový výstup programu 1.5.2_Pr_1.c.

Na viacnásobné vetvenie sa vo vhodnom prípade v jazyku *C* používa príkaz ***switch*** **(prepínač)**, ktorý testuje, či sa rozhodovací výraz (musí byť celočíselného typu v jazyku *C*) rovná jednej z niekoľkých celočíselných konštánt, a podľa toho je tok programu ďalej riadený. Počet vetiev (návestí, začínajú kľúčovým slovom *case*) nie je obmedzený. V každej vetve môže byť viac príkazov, ktoré nie je nevyhnutné uzatvárať do zložených zátvoriek. Je možné použiť vetvu *default*, ktorá sa realizuje v prípade, ak žiadna z vetiev *case* nevyhovuje. Všeobecný tvar:

```
switch (výraz)
{
   case hodnota1 :
      prikaz1;
      break;
   case hodnota2 :
      prikaz2;
      break;
   case hodnota3 :
      prikaz3;
      break;
   case hodnota4 :
      prikaz4;
      break;
   default :
      prikaz5;
      break;
}
```

Príkaz ***break*** je veľmi dôležitý, často je ním ukončená každá vetva. Jeho neuvedenie znamená, že sa robia všetky príkazy od vetvy s hľadanou hodnotu, až po prvý *break* (resp. *return*), alebo po koniec *switchu*. Inými slovami, tok riadenia neopúšťa *switch*, ale spracováva nasledujúcu vetvu v poradí, pričom nezáleží na tom, či za kľúčovým slovom *case* sú uvedené príkazy alebo nie. Ak chceme pre viacero hodnôt rozhodujúceho výrazu realizovať jeden a ten istý príkaz (príkazy), musíme definovať pre každú vyhovujúcu hodnotu rozhodujúceho výrazu samostatnú vetvu. V prípade väčšieho počtu vyhovujúcich hodnôt sa *switch* (z dôvodu jeho formy zápisu) stáva nepoužiteľný a vhodnejšie je použiť podmienku. Forma zápisu v tomto prípade je napr.:

```
switch (výraz)
{
   case hodnota1 :
   case hodnota2 :
   case hodnota3 :
      prikaz1;
      break;
   case hodnota4 :
      prikaz2;
      break;
   default :
      break;
}
```

Pozn.: V prípade hodnoty výrazu *hodnota1*, *hodnota2* alebo *hodnota3* sa vykoná *prikaz1*, v prípade hodnoty 4 *prikaz2* a v ostatných prípadoch sa nevykoná žiadny príkaz.

Vetva *default* sa nemusí nevyhnutne uvádzať (nie je povinná). Realizuje sa vtedy, keď sa žiadna z ostatných vetiev nevyhodnotí ako vyhovujúca. Je ale dobrým zvykom uvádzať ju aj v prípade,

ak je prázdna. Vetva *default* nemusí byť vždy uvedená na konci, toto vyplýva len zo zaužívanej konvencie. Avšak v prípade ak nie je uvedená ako posledná, ukončujúci *break* je nevyhnutný.

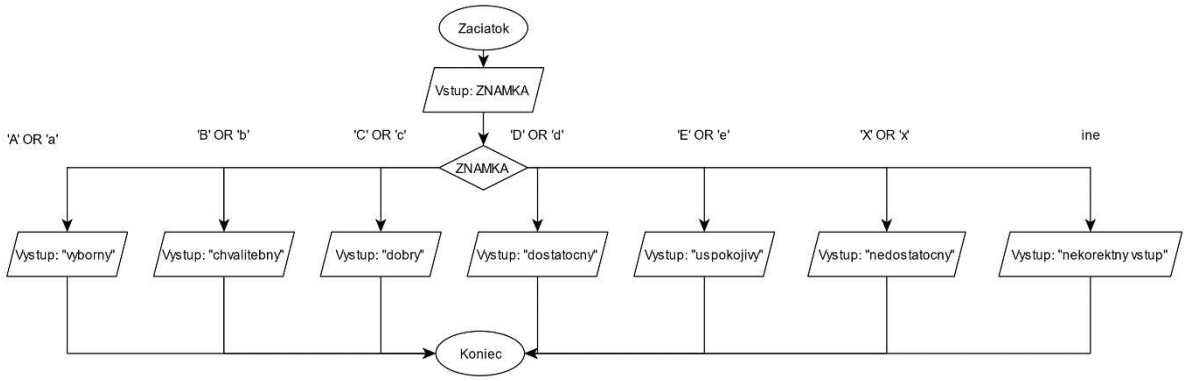
Príkaz *break* ukončuje prislúchajúce návestie *switchu*, v ktorom je uvedený. Na túto skutočnosť netreba zabúdať.

Otázka na zamyslenie: Ako by ste riešili vyššie uvedený problém pridelenia hodnotenia podľa počtu získaných bodov cez *switch*?

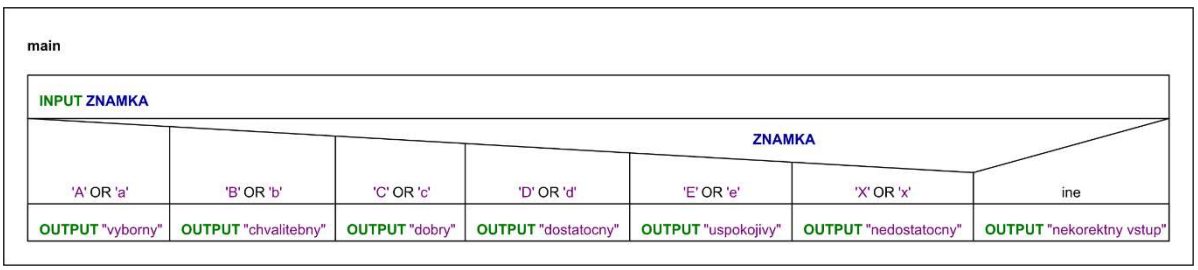
Odpoveď: Takýto problém nie je vhodný na riešenie s pomocou *switchu*, pretože nevieme exaktne určiť hodnotu jednotlivých vetiev. Udelenie hodnotenia je závislé od získaného počtu bodov, ktorý je vymedzený intervalom. To by znamenalo, že pre každú hodnotu, ktorú môže študent získať, musí sa zadefinovať samostatná vetva. Okrem toho by sme museli uvažovať, že študent môže získať len celočíselný počet bodov, inak by sme nesplnili základný predpoklad, že rozhodovací výraz musí byť v jazyku *C* celočíselného typu. V takomto prípade by sme týchto vetiev mali 100, čo by viedlo rozhodne k neprehľadnejšiemu zápisu, než pri použití zloženej podmienky s uvedomením vzájomne vylúčiteľných možností.

Príklad:

Na základe získaného hodnotenia vypíšte slovne používateľovi jeho hodnotenie.



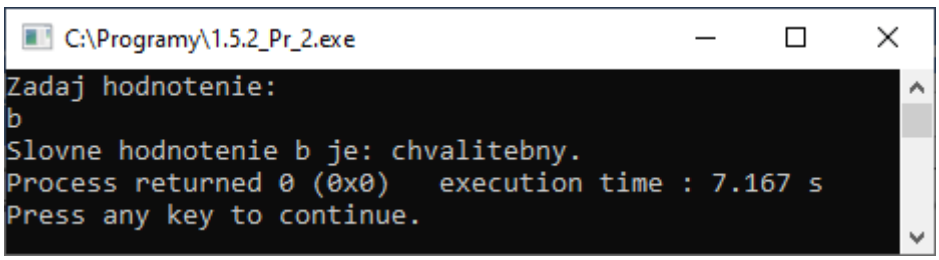
Obr. 15 Zakreslenie definovaného problému s pomocou vývojového diagramu.



Obr. 16 Zakreslenie definovaného problému s pomocou NS diagramu.

Jedna z možných implementácií 1.5.2_Pr._2.c:

```c
#include <stdio.h>

int main(void)
{
    char mark;

    printf("Zadaj hodnotenie:\n");
    scanf("%c", &mark);
    printf("Slovne hodnotenie %c je: ", mark);
    switch (mark)
    {
    case 'A':
    case 'a':
        printf("vyborny.");
        break;
    case 'B':
    case 'b':
        printf("chvalitebny.");
        break;
    case 'C':
    case 'c':
        printf("dobry.");
        break;
    case 'D':
    case 'd':
        printf("dostatocny.");
        break;
    case 'E':
    case 'e':
        printf("uspokojivy.");
        break;
    case 'X':
    case 'x':
        printf("nedostatocny.");
        break;
    default:
        printf("ziadne, pretote taketo hodnotenie nepoznam.");
        break;
    }

    return 0;
}
```



Obr. 17 Konzolový výstup programu 1.5.2_Pr_2.c.

V tomto prípade ide o typické použitie *switchu*, kde počet jednotlivých vetiev nie je rozsiahly a vieme ich exaktne špecifikovať.

Pozn. autora: Keďže hodnotenie *a* resp. *A* má pre nás ten istý logický význam, viackrát sme využili uvedenie vetvy bez *break*, napr. riadok 12, 16 a pod., čím sme zabezpečili požadovanú funkcionalitu bez nevyhnutnosti opakovania kódu. Vstup sme neošetrovali, z dôvodu, že sme využili vetvu *default*, ktorá reaguje v prípade, ak používateľ zadá iné hodnotenie než je korektné.

1.5.3 Podmienený výraz – ternárny operátor

Ternárny operátor funguje v istých prípadoch ako úplná podmienka *if – else*. Otázkou teda môže byť, prečo existuje a používa sa? Pretože *if – else* nie je výraz, ale príkaz, takže úplnú podmienku nie je možné použiť tam, kde je potrebný výraz. V niektorých prípadoch je jedno, čo použijete, ale existujú miesta, kde je použitie podmieneného výrazu veľmi vhodné. Syntax podmieneného výrazu:

```
booleovsky_vyraz ? vyraz1 : vyraz2
```

t. j. výsledná hodnota podmieneného výrazu je buď výsledok vyhodnotenia *vyraz1*, ak booleovský výraz nadobudne hodnotu *true*, alebo v opačnom prípade bude mať hodnotu vyhodnotenia výrazu *vyraz2*.

Napr.: $x = (a > b)\ ?\ a : b;$

1.5.4 Cykly = iteračné príkazy

Cyklus slúži na opakovanie určitého príkazu, alebo skupiny príkazov na základe podmienky, čiže pozostáva z podmienky cyklu a postupnosti príkazov nazývanými telom cyklu. Napríklad stlačte tlačidlo trikrát (opakovanie), kým svieti červená alebo oranžová (opakovanie s podmienkou). V tele cyklu musí nastať stav, ktorý spôsobí neplatnosť podmienky, a tým bude cyklus ukončený. V opačnom prípade by šlo o nežiaduci stav, teda zacyklenie. Každý prechod cyklom sa nazýva iterácia cyklu (jedno opakovanie cyklu). V rámci jednej iterácie cyklu sú vykonané všetky príkazy, ktoré sa nachádzajú v tele cyklu.

Telo cyklu môže byť aj prázdne. V takom prípade hovoríme o cykloch bez tela. Odporúča sa v takýchto prípadoch bodkočiarku uviesť odsadenú na novom riadku. Príkladom takéhoto cyklu môže byť napr. preskočenie všetkých znakov až po zadanie znaku konca riadku:

```
while (getchar() != "\n")
    ;
```

Špeciálnym prípadom sú **„nekonečné“ cykly**. Nekonečné cykly majú podmienku nastavenú vždy ako pravdivú (*true*). Ukončenie nekonečného cyklu sa dosiahne príkazom *break*, prerušenie tela cyklu príkazom *continue* po splnení určenej podmienky v tele cyklu.

- Nekonečný cyklus, napr. **while**(1) {…}

Vo všeobecnosti počet opakovaní cyklu môže byť definovaný implicitne alebo explicitne. Ak je opakovanie definované implicitne, tak je počet opakovaní viazaný na splnenie alebo nesplnenie nejakej podmienky. Pre explicitne definované opakovanie je počet opakovaní jednoznačne určený. Z takejto definície cyklu vyplývajú tri spôsoby zápisu cyklu:

- počet opakovaní je zadaný implicitne, pričom testovanie podmienky sa realizuje pred časťou, ktorá sa má opakovať **while** (vyraz) {prikazy;}
- počet opakovaní je zadaný implicitne a testovanie je na konci časti, ktorá sa má opakovať **do** {prikazy;} **while** (vyraz);
- počet opakovaní vieme jednoznačne (explicitne) určiť, tzv. cyklus for

  for (vyraz_start; vyraz_stop; vyraz_iter) {prikazy;};

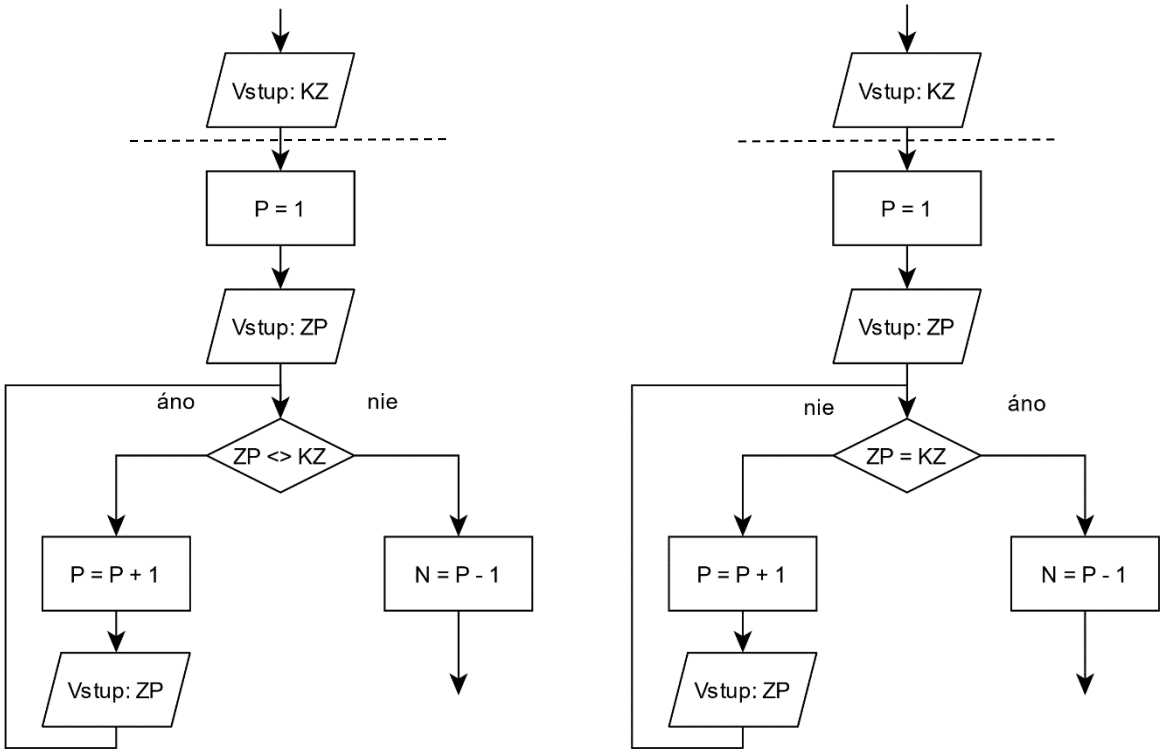
1.5.4.1 Cyklus s podmienkou na začiatku

Pri použití tohto cyklu nie je dopredu známe, koľkokrát sa má cyklus opakovať. Telo cyklu sa vykoná, ak je podmienka pravdivá. To znamená, že telo cyklu sa nemusí vykonať ani raz. Všeobecný zápis:

```
while (podmienka)
{
  TeloCyklu;
}
```

Na príklade ilustrujeme načítavanie prvkov postupnosti do premennej *ZP*, kým nebude zadaný ukončovací znak *KZ*. Koncový znak nesmie mať hodnotu, ktorá sa vyskytuje v čítanej postupnosti, v opačnom prípade by bol cyklus predčasne ukončený (čo by bolo nežiaduce). Pokiaľ bude koncový znak definovaný v programe, potom prvá značka načítania *KZ* bude vynechaná a v rozhodovacom bloku môže byť namiesto premennej *KZ* zapísaná priamo táto

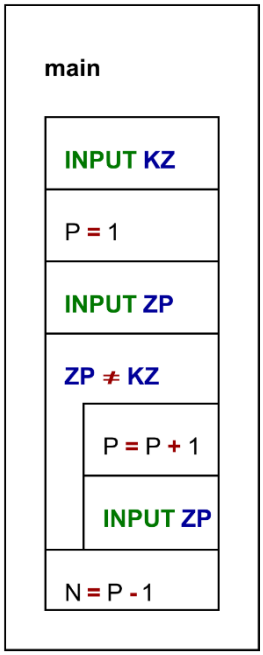
hodnota. Počet zadaných prvkov počítame s pomocou počítadla *P*. Prvá hodnota je načítaná ešte pred začiatkom cyklu. Preto sa javí ako logické, nastaviť počítadlo na začiatku na hodnotu 1. Avšak ukončovací znak nie je súčasťou načítavaných znakov postupnosti a slúži na končenie načítavania. Práve z tohto dôvodu musíme počítadlo dekrementovať o 1 po ukončení tela cyklu, ak chceme dosiahnuť reálny počet načítaných prvkov. V prípade ak by sme sa tejto dekrementácii chceli vyhnúť, počítadlo treba inicializovať na 0.



Legenda: KZ – koncový znak, ZP – znak (prvok) postupnosti, P – počítadlo, N – počet načítaných prvkov

Obr. 18 Fragment algoritmu ilustrujúci načítanie prvkov až po zadaný ukončovací znak s pomocou vývojového diagramu.

Pozn. autora: V prípade zakreslenia cyklov s podmienkou na začiatku/na konci, je možné podmienku, na základe ktorej sa vykoná alebo nevykoná telo cyklu zapísať vždy dvomi spôsobmi. Je to z toho dôvodu, že je možné označiť hrany splnenia/nesplnenia podmienky podľa svojho uváženia. Avšak, výrazne odporúčam zakresľovať cykly tak, aby nastalo vykonanie tela cyklu vždy, keď je podmienka splnená (fragment algoritmu na obrázku 18 vľavo). Toto odporúčanie vychádza zo skutočnosti, že väčšina programovacích jazykov, ako aj jazyk *C* dodržuje túto konvenciu, t. j. telo cyklu sa vykonáva, ak podmienka platí.



Obr. 19 Fragment algoritmu ilustrujúci načítanie prvkov až po zadaný ukončovací znak s pomocou NS diagramu.

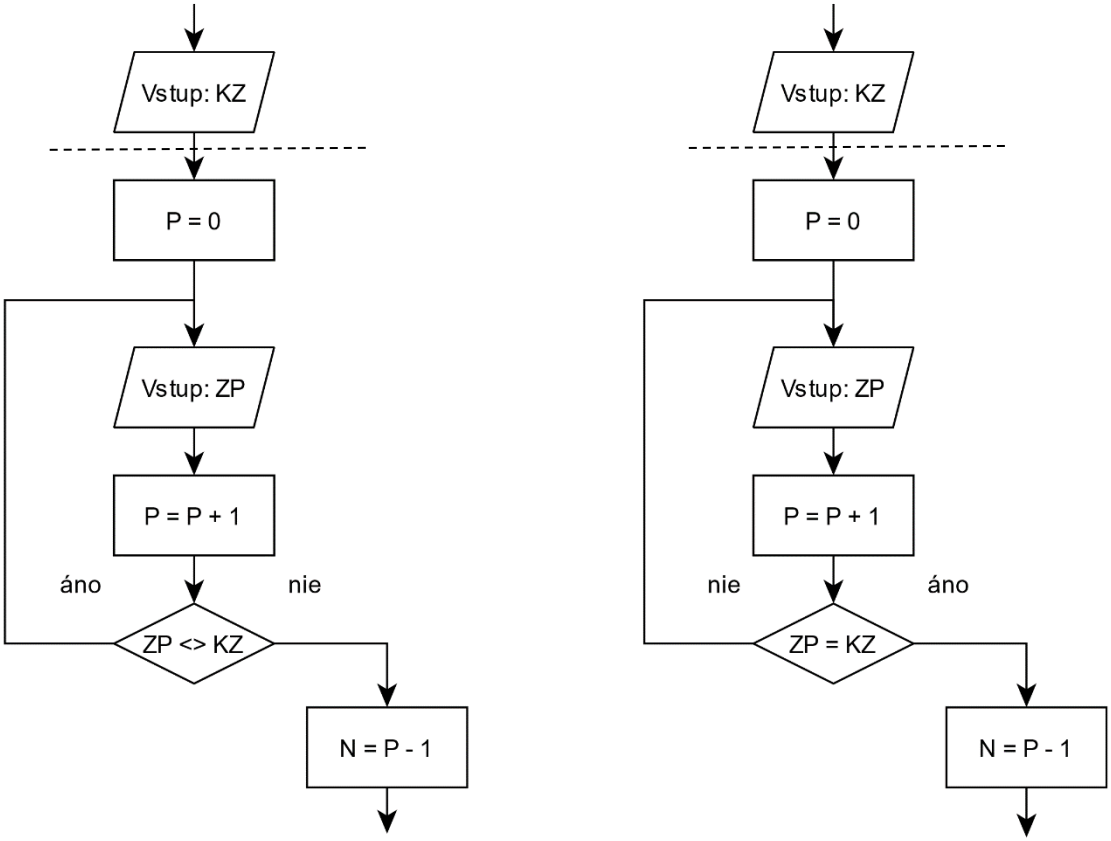
Pozn. autora: V prípade zakreslenia cyklov s podmienkou na začiatku/na konci s pomocou NS diagramu už nemáme pri určení podmienky možnosť voľby ako to bolo pri vývojovom diagrame. Tu sa nijako neoznačuje, kedy sa vykonáva tela cyklu, preto je veľmi dôležité vychádzať zo skutočnosti, že telo cyklu sa opakuje, kým podmienka platí.

1.5.4.2 Cyklus s podmienkou na konci

Pri použití tohto cyklu tiež nie je vopred známe, koľkokrát sa má cyklus opakovať. Telo cyklu sa tiež vykonáva, kým je podmienka pravdivá. V tomto prípade sa však telo cyklu vykoná aspoň jedenkrát, čiže aj pri neplatnej podmienke, na rozdiel od cyklu s podmienkou na začiatku. Všeobecný zápis:

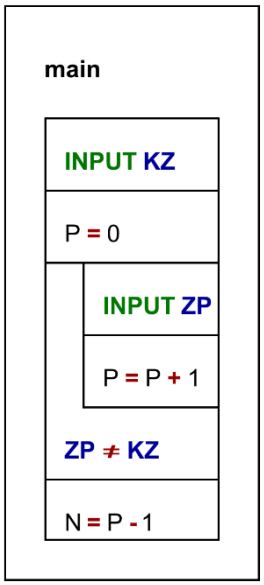
```
do
{
  TeloCyklu;
}
while (podmienka);
```

Príklad uvedený v kapitole 1.5.4.1 zakreslený s pomocou cyklu s podmienkou na konci.



Obr. 20 Fragment algoritmu ilustrujúci načítanie prvkov až po zadaný ukončovací znak s pomocou vývojového diagramu.

Pozn. autora: Z rovnakých dôvodov ako je uvedené vyššie odporúčame používať zakreslenie s pomocou algoritmu vľavo.



Obr. 21 Fragment algoritmu ilustrujúci načítanie prvkov až po zadaný ukončovací znak s pomocou NS diagramu.

1.5.4.3 Cyklus FOR

Cyklus sa väčšinou používa ak vieme dopredu určiť počet opakovaní cyklu. Všeobecný zápis:

```
for (vyraz_start; vyraz_stop; vyraz_iter)
{
  TeloCyklu;
}
```

- *vyraz_start* – vykoná sa iba raz pred spustením tela cyklu. Slúži väčšinou na inicializáciu počítadla.
- *vyraz_stop* – predstavuje ukončovaciu podmienku. Musí platiť, aby sa vykonalo telo cyklu. Kontroluje sa pred každým vykonávaním tela cyklu.
- *vyraz_iter* – vykonáva sa po každom priebehu tela cyklu. Väčšinou sa používa na zvyšovanie hodnoty počítadla.

Cyklus *for* prebieha tak, že sa na začiatku vyhodnotí *vyraz_start*, otestuje sa, či je *vyraz_stop* pravdivý, urobí sa príkaz (telo cyklu) a nakoniec sa urobí *vyraz_iter*. Následne sa vykonáva ďalšia iterácia cyklu.

Výrazy: *vyraz_start*, *vyraz_stop* a *vyraz_iter* spolu nemusia vôbec súvisieť a tiež nemusia byť vôbec uvedené. Ak však nie je niektorý z týchto výrazov uvedený, je nevyhnutné uviesť znak bodkočiarky "*;* ", ktorou je tento chýbajúci výraz oddelený od ostatných. Avšak takéto použitie zápisu *for* nie je to najvhodnejšie.

Príklad: Všetky príklady vytlačia čísla od 1 po 10, inšpirované Heroutom (2010).

1. Klasický *for*.

```
for (i = 1; i <= 10; i++)
{
    printf("%d ", i);
}
```

2. Využitie inicializácie riadiacej premennej *i* pri jej deklarácii. Toto nie je veľmi vhodné riešenie z dôvodu, že nie je všetko uvedené priamo v príkaze *for*. Toto môže viesť k tomu, že inicializáciu riadiacej premennej neúmyselne zmodifikujete ešte pred jej použitím v cykle.

```
int i = 1;
for (; i <= 10; i++)
{
    printf("%d ", i);
}
```

3. Riadiaca premenná je menená v tele cyklu. Toto nie je tiež vhodné riešenie z dôvodu, že môže byť zdrojom častých chýb z nepozornosti, a riadiacu premennú zmodifikujete neúmyselne viackrát.

```
int i = 1;
for (; i <= 10;)
{
    printf("%d ", i++);
}
```

4. V iteračnej časti nemusí byť žiadny iteračný príkaz. Toto je ale veľmi neprehľadné riešenie, ktoré bez akéhokoľvek logického dôvodu prehadzuje výkonný príkaz cyklu (tlač hodnoty) s iteračným príkazom. Dôvodom že je to vôbec funkčné je, že funkcia *printf()* vracia s pomocou návratovej hodnoty počet úspešne vypísaných premenných, v tomto prípade 1.

```
int i = 1;
for (; i <= 10; printf("%d ", (i - 1)))
{
    i++;
}
```

5. Telo cyklu môže byť prázdne – tiež neprehľadné riešenie.

```
int i = 1;
for (; i <= 10; printf("%d ", (i++)))
    ;
```

6. Využitie nekonečného cyklu s pomocou *for* a príkazu *break*. Absolútne zbytočne komplikované riešenie.

```
int i = 1;
for (;;)
{
    if (i > 10)
        break;
    printf("%d ", i++);
}
```

7. Využitie operátora čiarky – nie veľmi vhodné použitie, pretože tu nastáva presunutie výkonného príkazu do iteračnej časti.

```
int i = 1;
for (; i <= 10; printf("%d ", (i)), i++)
    ;
```

8. Využitie operátora čiarky pri inicializácii ($i = 1$, $sum = 0$) je vhodné, pri výpočte ($sum \mathrel{+}= i$, $i{+}{+}$) je nevhodné.

```
int i, sum;
for (i = 1, sum = 0; i <= 10; sum += i, i++)
    ;
```

Premenná *sum* musí byť deklarovaná mimo cyklu *for*. V prípade, že by sme premennú *sum* deklarovali priamo v cykle *for* (čo je možné od verzie štandardu *C99*), tak by premenná

mala oblasť platnosti len v rámci cyklu *for*, takže po opustení cyklu by s ňou nebolo možné pracovať, t. j. výpočet v cykle by bol zbytočný.

```
for (int i = 1, sum = 0; i <= 10; i++)
    sum += i;
// snaha o takyto vypis obsahu premennej sum by bol zbytocny
// printf("Sucet je %d. \n", sum);
```

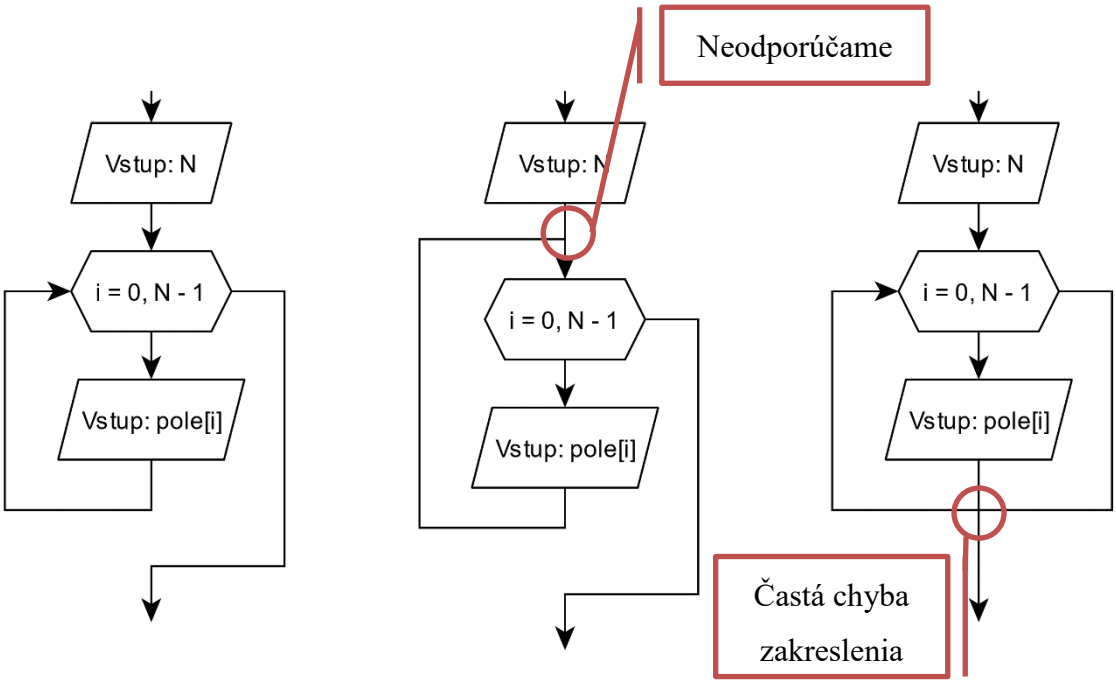
Každý cyklus *for* je možné prepísať s pomocou cyklu s podmienkou na začiatku. Z pohľadu programovania je irelevantné, ktorý zápis programátor použije, avšak z pohľadu reprezentácie algoritmu sa *for* považuje za prehľadnejšiu formu zápisu. V prípade zápisu algoritmu s pomocou diagramu musíme vždy prepísať cyklus *for* s pomocou cyklu s podmienku na začiatku vždy, keď sa riadiaca premenná modifikuje iným spôsobom ako je jej inkrementácia, resp. dekrementácia o 1. Je to z dôvodu, že pri zakreslení cyklu *for* nevidíme, ako sa riadiaca premenná modifikuje.

Prepis cyklu *for* s pomocou cyklu s podmienkou na začiatku:

```
vyraz_start;
while (vyraz_stop)
{
   TeloCyklu;
   vyraz_iter;
}
```

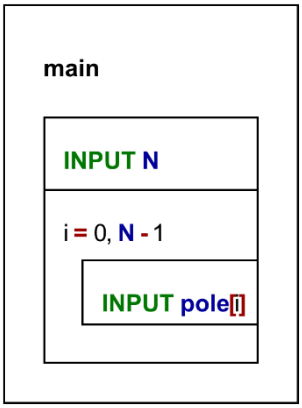
Typické použitie cyklu *for* je v prípade načítavania postupnosti čísiel. Počet prvkov postupnosti je vopred známy. V uvedenom príklade je uložený v premennej *n*. Parameter cyklu (riadiaca premenná) *i* je zároveň využitý ako index prvku poľa pri ukladaní prvkov – čo predstavuje načítanie prvkov do dátovej štruktúry poľa.

```
scanf("%d", &n);
for (i = 0; i < n; i++)
{
    scanf("%d", &pole[i]);
}
```



Obr. 22 Možnosti zakreslenie načítania prvkov do poľa s pomocou vývojového diagramu.

Pozn. autora: Pri zakresľovaní algoritmov s pomocou vývojových diagramov by sa hrany (ktoré reprezentujú tok riadenia) mali vždy spájať na hranách a nie na značkách. Jedinú výnimku tvorí cyklus *for*. V tomto prípade odporúčame cykliť na značku a nie na hranu, pretože v prípade zacyklenia nad značku *foru* to často vyvoláva dojem, akoby sa opätovne inicializovala riadiaca premenná *i* na počiatočnú hodnotu, t. j. v tomto prípade 0, čo však nie je pravdou. Častou chybou študentov býva aj zakreslenie, ako ilustrujem na obrázku 22 úplne vpravo. Toto spôsobuje porušenie vlastnosti determinovanosti.



Obr. 23 Možnosti zakreslenia načítania prvkov do poľa s pomocou NS diagramu.

Pozn. autora: Vzhľadom na skutočnosť že NS diagram pozostáva z jednotlivých blokov, nemusíme riešiť také detaily a ani chyby, ako boli opísané vyššie. Z tohto uhla pohľadu môžem konštatovať, že študenti robia pri práci s NS diagramom menej chýb. Dôvodom toho je určite aj skutočnosť, že vývojové diagramy sa neobmedzujú len na štruktúrované programovanie, narozdiel od NS diagramov.

1.5.4.4 Príkaz goto

Podľa Herouta (2010) príkaz ***goto*** sa v štruktúrovano koncipovaných programoch používa len málokedy, pretože v štruktúrovanom jazyku sa mu je možné vždy vyhnúť. V prípade, že je použitý, musí byť na to dôvod. Použitie skokového príkazu *goto* je charakteristické pre ukončenie vnorených (zahniezdených = *nested*) cyklov.

Príklad:

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        for (int k = 0; k < 10; k++)
        {
            if (x[k] == 0)
                goto chyba;
            a[i] = a[i] + b[j] / x[k];
        }
    }
}
printf("Dobre.\n"); // + napr. tlac vysledkov

chyba : printf("Nulovy delitel.\n");
```

1.5.4.5 Skokové príkazy break, continue a return

Všetky cykly je možné ukončiť alebo prerušiť s pomocou skokových príkazov ***break*** a ***continue.*** *Break* ukončuje najvnútornejší cyklus = predčasné ukončenie tela cyklu a pokračuje vykonávaním za cyklom. *Continue* prerušuje telo cyklu, ale vracia sa späť k podmienke cyklu, a až podľa jej vyhodnotenia sa rozhodne, či bude cyklus pokračovať alebo nie.

Ďalším skokovým príkazom je ***return***, ktorý ukončí vykonávanie danej funkcie, v ktorej je zapísaný. V súvislosti s funkciami sa používa na vrátenie návratovej hodnoty. Zatiaľ sme ho použili pri funkcii *int main(void)* v podobe *return 0;*, čo reprezentuje bezchybné ukončenie

programu. V prípade chyby sa väčšinou vráti hodnota 1. Bližšie sa problematike funkcií budem venovať v druhom diele tejto učebnice.

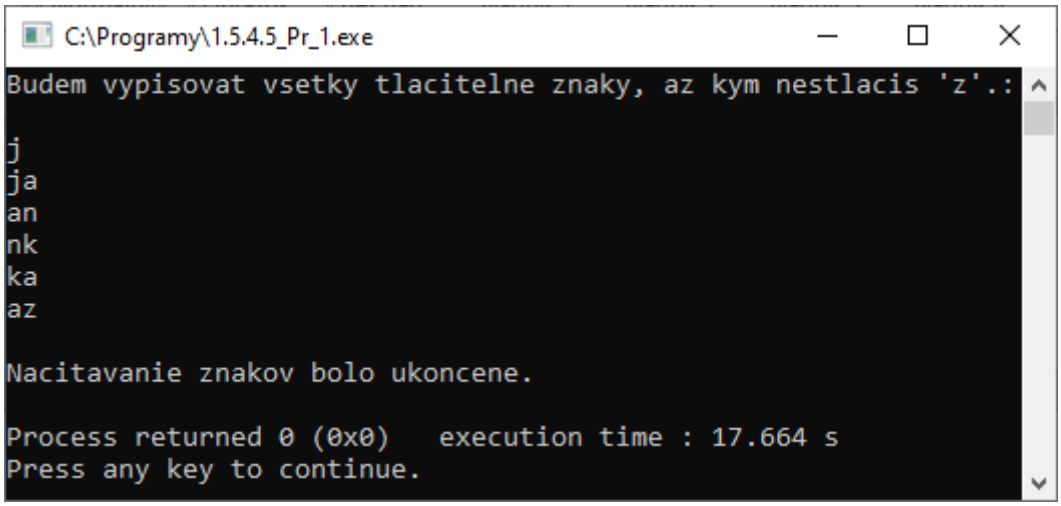
V prípade, ak by bola operácia z príkladu pre príkaz *goto* realizovaná cez funkciu, je vhodnejšie namiesto príkazu *goto* použiť *return*. Vtedy na získanie informácie o tom, či funkcia zrealizovala úlohu správne alebo nie, môžeme využiť návratovú hodnotu. Ak je hodnota 0, bude to znamenať, že funkcia skončila bez chýb a hodnota 1 s chybou.

```
int funkcia(void)
{
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            for (int k = 0; k < 10; k++)
            {
                if (x[k] == 0)
                    return 1; //neuspech
                a[i] = a[i] + b[j] / x[k];
            }
        }
    }
    return 0; //uspech
}
```

V niektorých prípadoch sa môžeme stretnúť aj s použitím funkcie *exit()*, ktorá má podobný význam ako príkaz *return*. Rozdiel je v tom, že bez ohľadu na akúkoľvek funkciu, z ktorej je vyvolaná, spôsobí ukončenie programu.

Príklad ilustrujúci použitie skokových príkazov *break* a *continue* 1.5.4.5_Pr_1.c:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int c; // deklaracia premennej
6
7    printf("Budem vypisovat vsetky tlacitelne znaky, az kym nestlacis 'z'.: \n");
8    while (1)
9    {
10     if ((c = getchar()) < ' ')
11       continue; //zahodenie neviditelneho znaku
12     if (c == 'z')
13       break; // zastavenie po nacitani znaku z
14     putchar(c);
15   }
16   printf("\nNacitavanie znakov bolo ukoncene.\n");
17
18   return 0;
19 }
```



Obr. 24 Konzolový výstup programu 1.5.4.5_Pr_1.c.

1.5.5 Komentáre

Aj keď ich študenti často nepoužívajú, komentáre predstavujú dôležitú časť zdrojového programu, pretože sprehľadňujú niekedy na prvý pohľad značne nepochopiteľný program. Ich primárnou úlohou je aby sa v programe zorientoval niekto cudzí, ale aj sám autor, keď sa k programu vráti za nejaký čas a už nemá v čerstvej pamäti všetky skutočnosti. Odporúčam komentáre uvádzať pred každou logicky ucelenou časťou kódu. Pri funkciách sa často uvádza popis jej funkcionality pred definíciou funkcie. Jazyk *C* rozlišuje dva typy komentárov:

- Jednoriadkový

```
// toto je jednoriadkovy komentar
```

- Viacriadkový – je možné ho využiť aj na jednoriadkový komentár

```
/*  toto
    je viacriadkovy
    komentar
*/
```

ANSI C nedovoľuje používať vnorené komentáre (*nested comments*), napr.:

```
/* prva cast komentara  /* druha cast komentara */ */
```

1.5.6 Typová konverzia v jazyku C

Ako uvádza Herout (2010), pod pojmom typová konverzia rozumieme prevod hodnoty premennej určitého typu na iný typ. Jazyk *C* rozoznáva dva druhy typovej konverzie:

- Implicitná (automatická) typová konverzia.
- Explicitná (vynútená, požadovaná) typová konverzia.

Implicitná typová konverzia má tieto základné pravidlá:

1. Pred vykonaním operácie sa operandy základných dátových typov konvertujú takto:
   - Typ *char* alebo *short int* sa konvertuje na typ *int*. Typ *float* sa konvertuje na typ *double*.
   - Všetky operandy typu *unsigned char* a *unsigned short* sa konvertujú na *int*, avšak len v prípade, ak typ *int* dokáže reprezentovať ich hodnotu (nepretečie). Inak sa konvertujú na *unsigned int*.
   - Napr.: Ak je deklarácia: `char c;` potom:

```
c = 65;          // 65 je konvertovane na char, t. j. c obsahuje znak "A"
c++;             // c obsahuje znak "B"
c = c + '1';     // je ako c = ORD(c) + ORD('1')
```

2. Ak majú dva operandy jednej operácie rôzny typ, tak je typ operandu s nižšou prioritou konvertovaný na typ s prioritou vyššou podľa nasledujúcej hierarchie (*int* má najnižšiu prioritu):

   int → *unsigned int*

   unsigned int → *long*

   long → *unsigned long*

   unsigned long → *double*

   double → *long double*

   Napr.: Ak je deklarácia: `int i;` potom:

```
i = 'A';         // je ako i = ORD('A'); (ordinarne cislo 'A' je 65)
i = 'A' + 2;     // je ako i = ORD('A') + 2;
i = 3.8;         // i bude 3 (0.8 sa odreze)
```

   Ak je deklarácia: `double g;` potom:

```
g = 5;           // g bude 5.0
```

3. V priraďovacích výrazoch je typ na pravej strane konvertovaný na typ z ľavej strany, čo je tiež typ výsledku.

   Napr.: `i = g * c;`

Najskôr sa skonvertuje *c* na *int* (pravidlo 1), potom sa *c* skonvertuje na *double* (pravidlo 2). Výsledok výrazu *g* * *c* je *double*, ale podľa pravidla 3 sa skonvertuje na *int* a priradí do *i*.

Explicitnú typovú konverziu na rozdiel od implicitnej typovej konverzie, ktorú nie sme schopní ovplyvňovať, môžeme využívať podľa našich potrieb, avšak s tým rizikom, že nevhodné použitie môže spôsobiť značné problémy. Explicitná typová konverzia sa tiež nazýva pretypovanie (*casting* alebo *typecasting*) a má formu:

(typ) (vyraz)

ktorá znamená, že *vyraz* (alebo premenná) je v čase prekladu konvertovaná na požadovaný *typ*. Operátor pretypovania sa zapisuje vo forme okrúhlych zátvoriek, vo vnútri ktorých je meno dátového typu na ktorý chceme pretypovať. Pretypovanie má najvyššiu prioritu, preto v prípade, ak chceme pretypovať výraz, je nevyhnutné tento uzatvoriť do zátvoriek, inak bude pretypovaný iba prvý člen výrazu.

Napr.:

```
int i = 10;
double d;
d = (double)i;   // d bude 10.0
/* bez pretypovania by bol vysledok chybny, petoze funkcia na vypocet odmocniny (sqrt())
potrebuje ako parameter typ double*/
d = sqrt((double) i);
```

Príklad a význam často používaných konverzií:

(*int*) char_vyraz – prevod znaku na ordinálne číslo

(*char*) int_vyraz – prevod ordinálneho čísla na zodpovedajúci znak

(*int*) double_vyraz – odrezanie desatinnej časti

(*double*) int_vyraz – prevod celého čísla na reálne

Explicitná typová konverzia je vhodná vo viacerých prípadoch. Najčastejšie pri používaní pointerov (bližšie vysvetlené v druhom dieli učebnice), alebo pri funkciách pri nesúlade skutočných parametrov s formálnymi.

1.6 Formálny zápis algoritmov a overenie správnosti algoritmu

Forma zápisu všeobecného algoritmu môže byť rôzna, závisí predovšetkým od účelu použitia algoritmu. Podmienkou je však jednoznačnosť a zrozumiteľnosť zápisu tak, aby realizátor vedel jednoznačne vykonávať v každom kroku definované akcie.

Jednotlivé spôsoby zápisu algoritmov môžeme kategorizovať na:

- slovný zápis (v prirodzenom jazyku) – vyznačuje sa nejednoznačnosťou výkladu a rozsiahlosťou,
- grafický zápis – s využitím vývojových diagramov (štandardizovaný spôsob), štruktúrogramov = Nassi-Schneidermanove diagramy, obrázkového návodu, a pod.
- zápis s pomocou rozhodovacích tabuliek,
- matematický zápis – algoritmus je vyjadrený popisom vzťahov medzi veličinami, sústavami rovníc, maticami a pod.,
- využitie pseudojazyka,
- alebo zápis priamo v programovacom jazyku.

Na overenie správnosti algoritmu existuje tiež viacero prístupov:

- dôkazové metódy (matematika),
- testovanie, krokovanie (ručne na papieri najčastejšie v tabuľkovej forme, alebo na počítači, ak je algoritmus implementovaný).

1.6.1 Slovný zápis algoritmu

Jedna z najnevhodnejších foriem zápisu najmä z dôvodu častej nepresnosti vyjadrení. Oproti iným spôsobom môžeme ako negatívum vnímať aj rozsah a neprehľadnosť zápisu.

Príklad:

Zadanie: Zistite či majiteľovi pozemku obdĺžnikového tvaru daných rozmerov postačí na oplatenie *100 €*, ak *1 m* pletiva stojí *5 €*?

Slovný zápis: Načítanie rozmerov daného pozemku od používateľa: šírka a dĺžka. Výpočet obvodu daného pozemku na základe vzťahu $obvod = 2 \cdot šírka + 2 \cdot dĺžka$. Následne výpočet ceny pletiva, ktorú bude potrebné zaplatiť za pletivo podľa vzťahu $cena = obvod \cdot 5$. Zistenie, či cena je vyššia ako *100 €*. Ak áno, tak vypísať používateľovi, že nemá dostatok peňazí, poprípade ho informovať aké navýšenie predpokladaného rozpočtu potrebuje na základe

vzťahu *doplatiť* = *cena* − 100. V opačnom prípade ho informovať, že má dostatok finančných prostriedkov na oplotenie pozemku.

1.6.2 Zápis s pomocou pseudojazyka

Pseudojazyk (*pseudocode*) je neformálny jazyk, ktorý je ľahko čitateľný, neuvádza na pochopenie algoritmu nepodstatné detaily. Využíva konštrukcie štruktúrovaného programovania. Existujú rôzne formy zápisu. Často sa v zápise, v závislosti od historického vývoja programovacích jazykov, využivajú kľúčové pojmy inšpirované programovacím jazykom *Pascal,* napr.:

- **Začiatok a koniec bloku/algoritmu**
  - begin/end = začiatok/koniec

    begin

    akcia1;

    akcia2;

    akcia3;

    end
- **Podmienka (neúplná, úplná, zložená)**
  - if, then, else = ak, potom, inak

    if test **then** akcia;

    if test **then** $akcia_1$; **else** $akcia_2$;

    if test **then** {$akcia_1$; $akcia_2$; … $akcia_N$;} **else** $akcia_1$;

    if test **then** {$akcia_1$; $akcia_2$; … $akcia_N$;} **else if** test $akcia_1$; **else** $akcia_1$;
- **Cykly**
  - cyklus s podmienkou na začiatku – while, do = pokiaľ, opakuj
  - cyklus s podmienkou na konci – repeat, until = opakuj, kým
  - cyklus s explicitne definovaným počtom opakovaní – for, from, to = pre, od, do

    while test **do** akcia;

    repeat akcia **while** test;

    for i **from** a **to** z akcia;

Príklad:

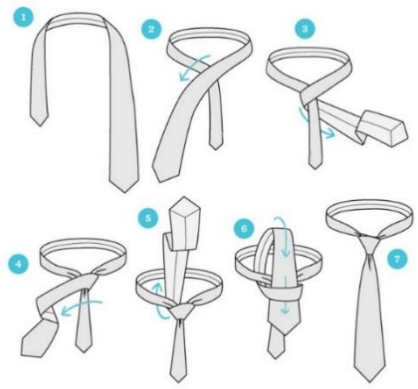
Zadanie: Zistite či majiteľovi pozemku obdĺžnikového tvaru daných rozmerov postačí na oplatenie *100 €*, ak *1 m* pletiva stojí *5 €*?

Pseudojazyk:

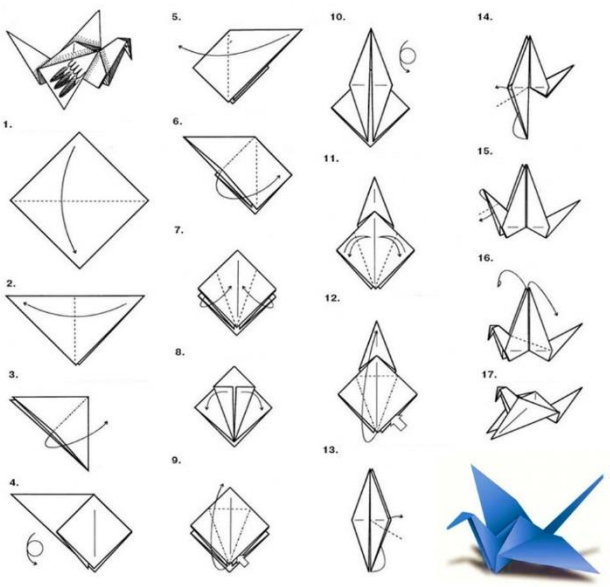
```
začiatok
vstup Sirka;
vstup Dlzka;
obvod = 2*Sirka + 2*Dlzka;
cena = obvod*5;
ak cena <= 100, tak výstup „Máš dosť peňazí.“;
    inak {
            doplatit = cena – 100;
            výstup „Nemáš dostatok peňazí, chýba ti“ doplatit „peňazí“.
          }
koniec
```

1.6.3 Grafický zápis

Medzi grafické zápisy algoritmov patria aj rôzne obrázkové návody ako príprava rôznych jedál – tzv. fotoreceptov, ako si uviazať kravatu, ako poskladať origami a pod. Vzhľadom na charakter mňou riešených algoritmov sa v tejto učebnici týmto formám nevenujem. Pozornosť som zamerala na vývojové diagramy a štruktúrogramy.



Obr. 25 Obrázkový návod na uviazanie kravaty (https://www.stevula.sk/n/ako-uviazat-kravatu).



Obr. 26 Obrázkový návod origami – japonský žeriav (https://handmadebase.com/sk/how-do-origami-crane-boom-of/).

1.6.3.1 Vývojové diagramy a štruktúrogramy

Vývojový diagram (*flowchart*) je orientovaný graf, kde sú hrany orientované od jedného kroku k druhému (najčastejšie zhora nadol, resp. zľava do prava). Predstavuje teda grafické znázornenie algoritmu riešenia problému postupnosťou normalizovaných značiek, ktorými zachytáva tok riadenia a spracovania dát. Značky sú normalizované podľa ISO 5807:1985, a aj preto môžeme povedať, že patrí medzi najpoužívanejšie prostriedky grafického formálneho zápisu algoritmov. Jeho uplatnenie nájdeme aj v UML[3]. Vývojový diagram je vhodný najmä na vizualizáciu jednoduchých algoritmov.

Najčastejšie používané symboly vývojového diagramu:

- **Úsečka** (spojnica) – určuje smer spracovania algoritmu. Môže, ale nemusí byť ukončená šípkou. Smer dole a zľava do prava je prioritný, v tomto prípade nie je nevyhnutné použiť šípky. Šípky sa používajú väčšinou v prípade, že tento smer je iný, alebo ak je treba smer toku spracovania zvýrazniť, napríklad pri znázornení iterácie. Úsečky sú zvislé alebo

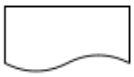
[3]Unified Modeling Language (UML) je v softvérovom inžinierstve grafický jazyk slúžiaci na vizualizáciu, špecifikáciu, navrhovanie a dokumentáciu programových systémov.

vodorovné čiary, v prípade potreby sa môžu križovať alebo spájať, ale tak, aby bol vždy jasný tok riadenia.

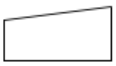
- **Obdĺžnik** s popisom – definuje čiastkový krok spracovania algoritmu, tzv. akciu, príkaz.
- **Obdĺžnik so skosenými hranami –** vstup/výstup. Vzhľadom na neexaktnosť je nevyhnutné dodatočne špecifikovať či ide o vstup, resp. výstup, napr. slovne, alebo symbolikou.
- Obdĺžnik so zaoblenými rohami (**elipsa**) – začiatok alebo ukončenie algoritmu.
- **Kosoštvorec** – vetvenie postupu v algoritme v závislosti na splnení/nesplnení podmienky.
- **Šesťuholník –** cyklus s explicitne definovaným počtom opakovaní – cyklus *for*.
- **Kruh –** spojka.

Symboly vývojového diagramu – základné rozšírenia:

- **Dokument** býva často reprezentovaný obdĺžnikom s vlnitou spodnou hranou.



- **Ručný vstup** býva často zobrazený s pomocou štvoruholníku zľava doprava so stúpajúcou hornou hranou. Príklad ručného vstupu: napr. vstup dát z formulára.



- **Manuálna operácia** býva reprezentovaná lichobežníkom. Používa sa na znázornenie operácie alebo úpravy, ktorá môže byť urobená iba ručne.

- **Dátový súbor (databáza)** býva znázornený s pomocou valca.

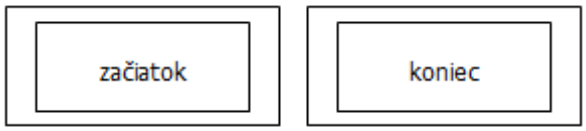
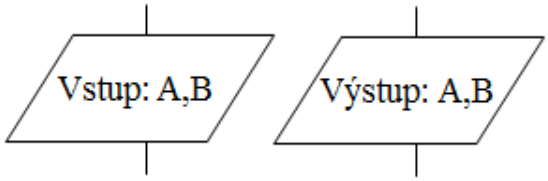
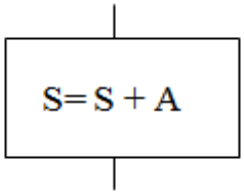
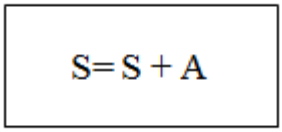
- **Dátové úložisko (*cloud*)** býva znázornený s pomocou obláčika.

Nassi a Schneiderman navrhli ešte v roku 1972 dokumentačnú symboliku, ktorá zodpovedá zásadám štruktúrovaného programovania. Nenájdeme v nej napríklad zastúpenie pre skokový

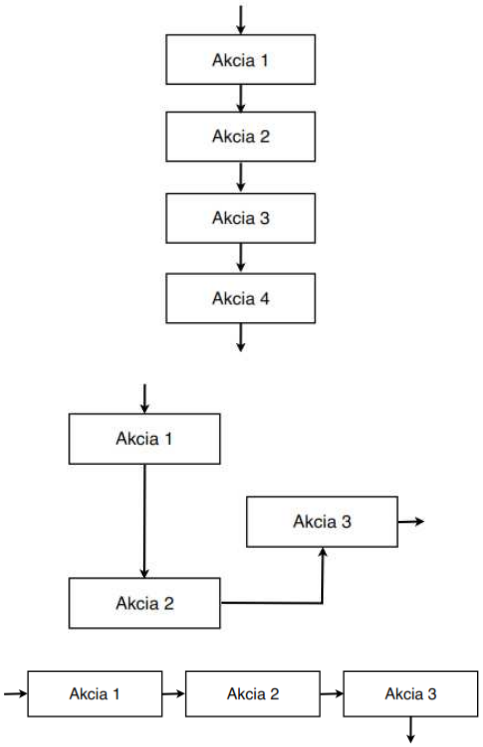
príkaz *goto*. Vychádzajúc z metódy riešenia problému zhora nadol, problém je rozkladaný na menšie a menšie podproblémy, až kým nedosiahneme jednoduché príkazy a základné riadiace konštrukcie. Každá jednotlivá akcia programu sa uvedie v zodpovedajúcom štruktúrnom bloku. Štruktúrne bloky môžu byť do seba ľubovoľne vnorené. Táto postupnosť štruktúrnych blokov tvorí **štruktúrogram (NS diagram)**.

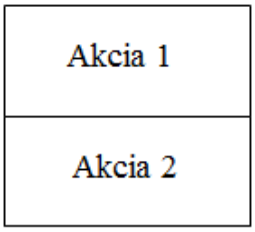
Takmer všetko čo je možné zakresliť s pomocou vývojového diagramu, je možné zakresliť aj s pomocou štruktúrogramu. Rozdiel však nájdeme pri reprezentácii rôznych vstupov dát, ako aj zakreslenia skokových príkazov. NS diagram poskytuje len jednu možnosť vstupu, na rozdiel od vývojových diagramov, avšak doplňujúcim popisom sa dá ilustrovať aj táto skutočnosť. Inými slovami by sme mohli povedať, že použitie NS diagramov je to len iná forma reprezentácie algoritmu, s pomocou vývojového diagramu. Avšak pri konkrétnych riešeniach na istom stupni detailnosti si uvedieme isté výhody a nevýhody týchto návrhov.

Značky pre jednotlivé príkazy a základné riadiace konštrukcie:

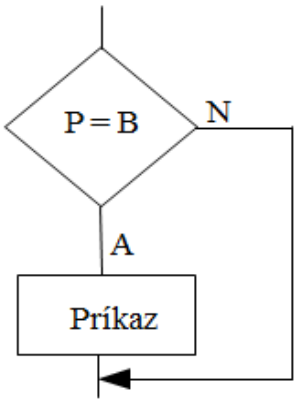
<table>
<tr><th>Vývojový diagram (flowchart)</th><th>NS diagram (structogram)</th></tr>
<tr><td colspan="2">Začiatok/koniec</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td colspan="2">Vstup / Výstup – načítanie premenných A, B, resp. výstup premenných A, B</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td colspan="2">Spracovanie – príkaz</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

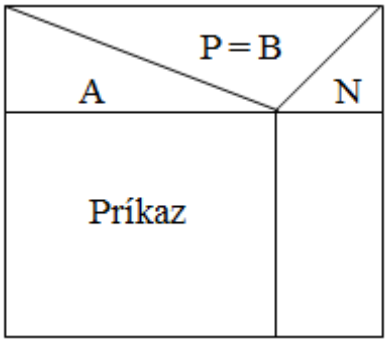
Postupnosť/sekvencia – príkaz1, príkaz2…



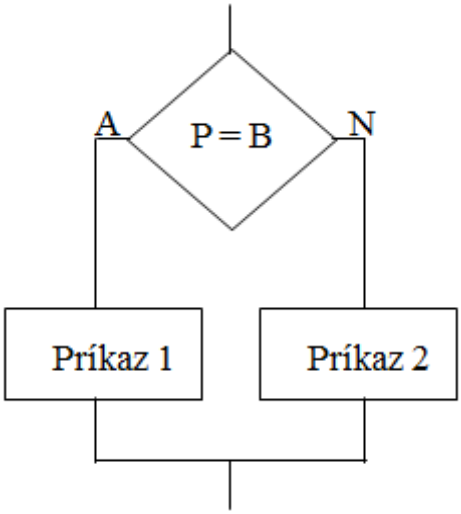


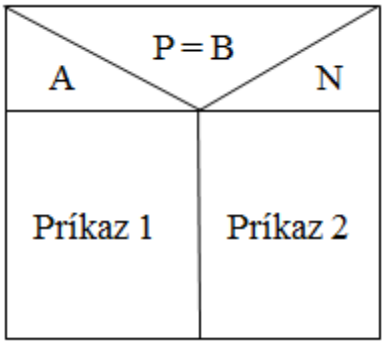
Podmienené spracovanie (neúplná podmienka) – ak P = B, tak príkaz



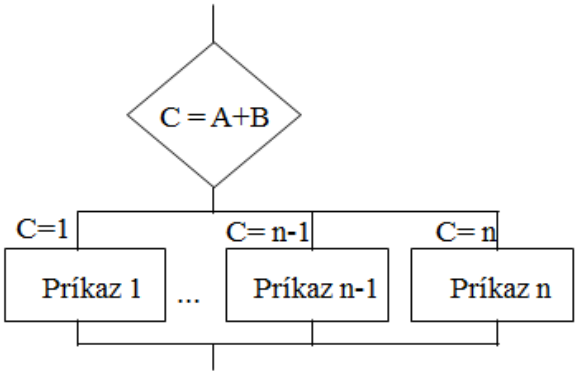


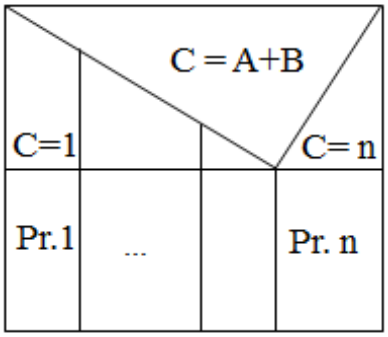
Jednoduché vetvenie (úplná podmienka) - ak P = B, tak príkaz1, inak príkaz2



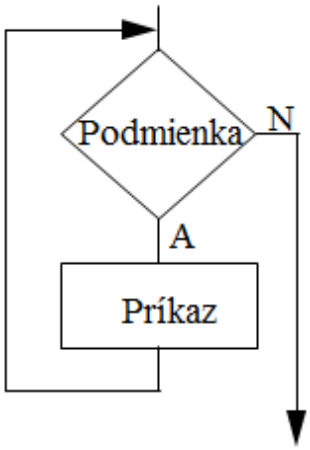


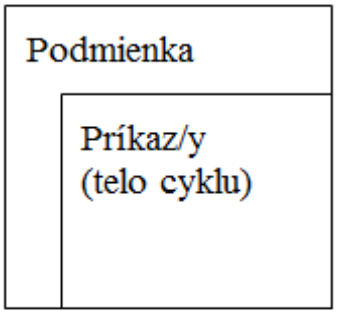
Viacnásobné vetvenie (prepínač)



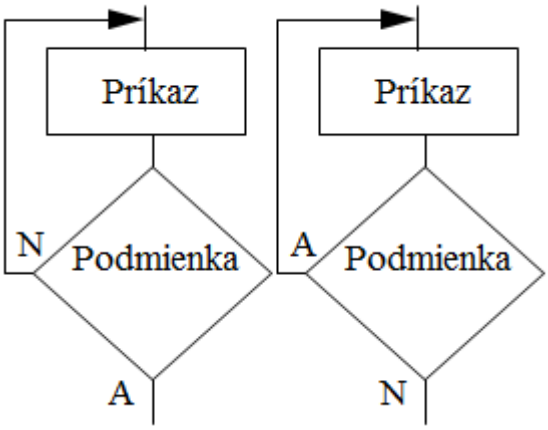


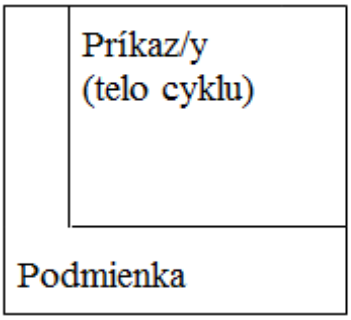
Cyklus (opakovanie) s testom podmienky na začiatku – pokiaľ platí podmienka opakuj príkaz (príkazy)



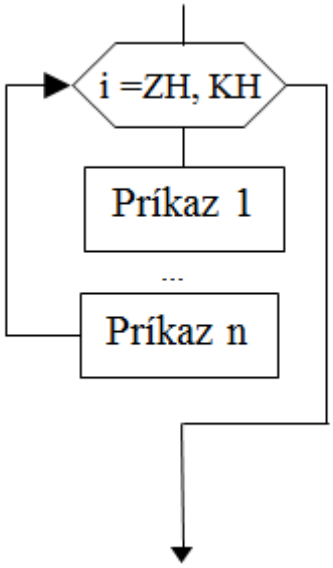


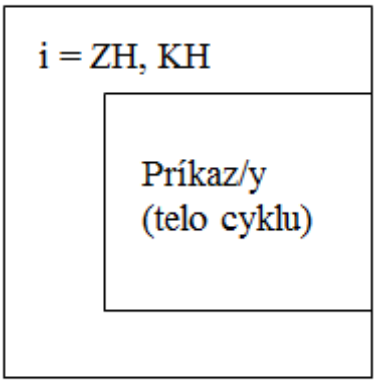
Cyklus (opakovanie) s testom podmienky na konci – opakuj príkaz (príkazy) kým je splnená podmienka, pri vývojovom diagrame sa dá aj opakuj príkaz (príkazy) kým nie je splnená podmienka

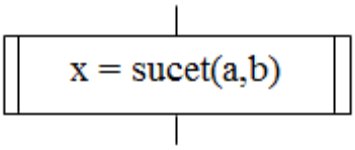
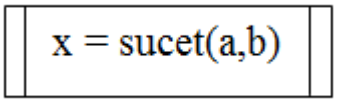
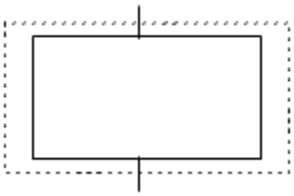
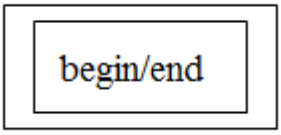
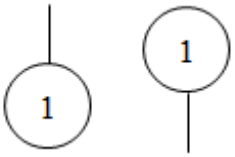
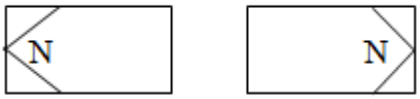




Cyklus s explicitne stanoveným počtom opakovaní – pri *i = ZH, KH* opakuj príkaz, kde *ZH* – začiatočná hodnota, *KH* – koncová hodnota, ktorú riadiaca premenná *i* ešte nadobúda v rámci cyklu, iná alternatíva zápisu (neodporúčame): pre *i = od ZH po KH vrátane* opakuj príkaz





<table>
<tr><td colspan="2">Volanie podprogramu (funkcie)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td colspan="2">Blok (podprogram) – začiatok (begin) {príkazy} koniec (end)</td></tr>
<tr><td>  </td><td> 
</td></tr>
<tr><td colspan="2">Spojka (párová značka)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

Pozn. autora: V tejto učebnici na reprezentáciu algoritmu s pomocou vývojového diagramu používam nástroj yEd Graph Editor (https://www.yworks.com/products/yed) a s pomocou NS diagramu Structorizer (https://structorizer.fisch.lu/). Najmä v prípade použitia Structorizera autori tohto nástroja zaviedli menšie zmeny v súvislosti s reprezentáciu zaužívaných značiek uvedených vyššie. Napríklad v prípade vstupu a výstupu nepoužívajú šípku, ale kľúčové slovo. A tiež nepoužívajú značku pre začiatok a koniec. Z tohto dôvodu nájdete v NS diagramoch uvedených v tejto učebnici mále grafické odlišnosti.

1.6.4 Rozhodovacie tabuľky

Rozhodovacie tabuľky sú vhodné na znázornenie komplexných rozhodovacích procesov v tabuľkovej forme. Umožňujú vyjadrovať a algoritmizovať rozhodovacie procesy, opísať a analyzovať zložité systémy a ich činnosti do formy vhodnej na programovanie. Boli zavedené v roku 1958 a používali sa ako vývojový nástroj softvérového inžinierstva.

Rozhodovaciu tabuľku definujeme podľa Chvalovského (1984) ako dvojrozmernú, tabuľkovo usporiadanú informáciu, ktorá využíva základné vzťahy matematickej logiky na stanovenie variantov rozhodnutí alebo činností (*závislých výrokov*), ktoré logicky vyplývajú zo splnenia alebo nesplnenia kombinácie daných podmienok (*nezávislých výrokov*).

Rozhodovacie tabuľky ponúkajú možnosť:

- zhrnúť a vyjadriť komplexnú logiku problému prehľadnou a jednoznačnou formou, ktorá nie je závislá na organizačnej alebo výpočtovej technike,
- znázorniť logické alternatívne smery činností pri kombinácii všetkých možných podmienok, ktoré v súvislosti s riešením problému môžu nastať,
- uľahčiť analýzu problému a zjednodušiť jeho dokumentáciu a efektívne vyjadrenie umožňujúce ľahké vykonanie zmien a úprav,
- usporiadať logiku systému takým spôsobom, ktorý umožní jeho ľahké pochopenie a nevyžaduje špeciálne znalosti,
- v zvláštnych prípadoch môže byť rozhodovacia tabuľka podkladom pre priame vytvorenie programu.

Každá rozhodovacia situácia je opísateľná s pomocou pojmov:

- *podmienky;*
- *akcie;*
- *predpisy;*
- *výsledky.*

Základná štruktúra rozhodovacej tabuľky je určená štyrmi kvadrantmi:

- V I. kvadrante je uvedený zoznam podmienok ($p_1, p_2, p_3, \ldots, p_n$), ktoré treba testovať pri riešení problému.
- II. kvadrant obsahuje zoznam činností ($a_1, a_2, a_3, \ldots, a_m$), ktorých vykonanie alebo nevykonanie je predmetom rozhodovania.
- V III. kvadrante sa uvádza v stĺpcoch kombinácia možných vstupov podmienok vo forme pravdivostných hodnôt (*A – áno, N – nie*), ktoré nadobúdajú podmienky, uvedené v I. kvadrante.
- Vo IV. kvadrante sa uvádzajú v stĺpcoch vstupy činností. Znak *X* vyjadruje, ktoré činnosti sa pri každom stave podmienok majú vykonať. Prázdne pole znamená, že nie je potrebné

činnosť realizovať. Každá činnosť v stĺpci tohto kvadrantu zodpovedá kombinácii stavu podmienok v prislúchajúcom stĺpci III. kvadrantu. Činnosti je možné rozpracovať do rôznych stupňov podrobnosti vrátane prechodu k ďalším rozhodovacím tabuľkám.

Identifikácia		**Číslo pravidla**	
Časť podmienok	I.	Časť indikácie podmienky	III.
Časť akcií	II.	Časť indikácie akcie	IV.

Príklad:

Rozhodnite o udelení zápočtu študentovi, ak kritériami na udelenie je splnená účasť študenta na výučbe a spracovanie dvoch zadaní. V prípade, ak študentovi chýba spracovanie jedného zo zadaní, poskytnúť mu opravný termín.

	Zápočet	1.	2.	3.	4.	5.	6.	7.	8.
P1	Účasť	A	A	A	A	N	N	N	N
P2	1. zadanie	A	A	N	N	A	A	N	N
P3	2. zadanie	A	N	A	N	A	N	A	N
A1	Udeľ zápočet	X							
A2	Daj nový termín		X	X					
A3	Neudeľ zápočet				X	X	X	X	X

Obr. 27 Rozhodovacia tabuľka ilustrujúca udelenie/neudelenie zápočtu.

Pozn.: Ak nie je niektorá kombinácia možná, môžeme ju eliminovať, alebo priamo eliminovať pravidlo s cieľom zníženia možných kombinácií zapísaných v rozhodovacej tabuľke.

Možnosti konštrukcie rozhodovacej tabuľky:

- **Úplná rozhodovacia tabuľka** zabezpečuje riešenie každej reálnej situácie, ktorá môže nastať v dôsledku splnenia alebo nesplnenia uvažovaných podmienok.
- **Rozšírená rozhodovacia tabuľka** vyjadruje rozšírenie podmienok, alebo činností slovami alebo inou symbolikou. V rozšírenej rozhodovacej tabuľke je časť každej

podmienky zapísaná v I. a časť v III. kvadrante, napr. v I. kvadrante je *stav účtu menší ako* a v jednotlivých stĺpcoch III. kvadrantu sú v tom istom riadku peňažné sumy.

- **Zmiešaná rozhodovacia tabuľka** obsahuje kombináciu binárneho a rozšíreného zápisu podmienok, alebo činností.
- **Redukovaná rozhodovacia tabuľka** vzniká odstraňovaním duplicitných párov pravidiel ich zlučovaním do zložených pravidiel. *Duplicitný pár* vytvárajú tie dve pravidlá, ktorých stavy podmienok vedú k tej istej kombinácii volieb činností, pričom sa líšia len pri jednej podmienke. Ako duplicitný pár môžeme chápať možnosti 6, 7 a 8 uvedeného príkladu.

1.6.5 Vlastnosti algoritmu

Pri tvorbe algoritmu sú naň kladené isté nároky, ktoré môžeme charakterizovať ako vlastnosti algoritmu. Tieto môžeme rozdeliť na dve skupiny: nutné a odporúčané (očakávané) vlastnosti. Principiálne vychádzam z diela Niklausa Wirtha (1989).

Nutné vlastnosti algoritmu:

- **Determinovanosť (deterministickosť)**

  Táto vlastnosť algoritmu vyjadruje, že každý krok algoritmu musí byť jednoznačne a presne definovaný. V každej situácii musí byť úplne zrejmé, čo a ako sa má vykonať – ako postupuje tok riadenia algoritmu a ako má vykonávanie algoritmu pokračovať.

- **Konečnosť**

  Každý algoritmus musí skončiť po vykonaní konečného počtu krokov. Tento počet krokov môže byť ľubovoľne veľký (podľa rozsahu problému a hodnôt vstupných údajov), ale pre každý jednotlivý vstup musí byť konečný v čase a v priestore.

- **Rezultatívnosť**

  Ak algoritmus dáva po konečnom počte krokov správny výsledok, ktorý je pre rovnaké vstupné údaje vždy rovnaký (ak algoritmus skončí), hovoríme, že algoritmus je rezultatívny. Táto vlastnosť algoritmu sa vzťahuje na výsledok a súvisí s konečnosťou.

Očakávané vlastnosti algoritmu:

- **Hromadnosť**

  Algoritmus, ktorý spĺňa túto vlastnosť je použiteľný nielen pre konkrétne vstupné hodnoty, ale je použiteľný pre celú prípustnú množinu vstupných hodnôt daného problému.

- **Elementárnosť**

  Aby algoritmus splnil túto vlastnosť, je dôležité, aby bol zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, ako aj programovacieho jazyka, v ktorom sa algoritmus bude realizovať (implementovať).

- **Efektívnosť**

  Znamená s čo možno s najmenším počtom krokov získať riešenie súčasne za najkratší čas (časová zložitosť algoritmu) a s využitím minimálneho množstva výpočtových prostriedkov (priestorová zložitosť algoritmu).

  Pozn. autora: Z pohľadu algoritmizácie by bolo postačujúce pri posudzovaní priestorových nárokov vyčísliť napríklad počet premenných potrebných na riešenie daného problému a posúdiť či tieto sú závislé od vstupov alebo nie, resp. či sa priestorové nároky počas činnosti algoritmu budú meniť. V istých momentoch, ako napríklad pri analýze viacerých vhodných riešení jedného problému však toto nemusí byť dostačujúce, resp. dostatočne presné. Z tohto dôvodu, v tejto učebnici sa budem snažiť aj vyčíslovať priestorové nároky odvíjajúce sa od použitých premenných a prislúchajúcich dátových typov. V prípade použitia viacerých funkcií, budem vyčíslovať priestorové nároky aj pre každú funkciu samostatne, aby bolo zrejmé, ktoré priestorové nároky sú trvalé a ktoré sú len dočasné. Plne si uvedomujem, že pri určovaní priestorovej zložitosti algoritmu sa na túto pozeráme trochu z iného pohľadu, ako je zvykom. Tejto problematike sa zodpovedne venujem v poslednej kapitole, vzhľadom na jej rozsiahlosť a náročnosť. Je však dôležité aby si študenti uvedomovali základné skutočnosti v tejto súvislosti hneď od začiatku návrhu prvého algoritmu.

Pri posudzovaní efektívnosti by sme sa mali pozrieť aj na dve ďalšie vlastnosti:

- **Modifikovateľnosť**

  Každý algoritmus by mal poskytnúť možnosť jednoduchej zmeny (úpravy) toku údajov, ako aj samotného algoritmu.

- **Štruktúrovanosť**

  Algoritmus by mal pozostávať zo samostatných, ale logicky previazaných celkov.

1.7 Ako pristupovať k návrhu algoritmu zadaného problému

V tejto kapitole na niekoľko ilustratívnych príkladoch vysvetlím ako postupovať pri návrhu algoritmu zadaného problému a jeho následnej implementácii, čiže prakticky využijeme všetky vedomosti, ktoré sme si osvojili v predchádzajúcich častiach.

1.7.1 Príklad 1 – kvadratická rovnica

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte s pomocou vývojového diagramu (VD) a NS diagramu riešenie. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém výpočtu koreňov kvadratickej rovnice.

Rozbor úlohy: Pri priamych metódach riešenia problémov sme sa stretli s pojmom abstrakcia, ktorý predstavuje proces, pri ktorom sa zanedbajú určité prvky, ktoré nie sú dôležité na danej úrovni. Podobne môžeme abstrakciu použiť aj v súvislosti so zadaným problémom. Ak sa na problém pozrieme komplexne, tak v závislosti od koeficientov kvadratickej rovnice môže v prípade ak $a = 0$ nastať riešenie lineárnej rovnice. Avšak ak abstrahujeme, a cieľom je riešiť len korene kvadratickej rovnice, tak toto nemusíme uvažovať a stačí iba doplniť vstupnú podmienku pre koeficient a, t. j. $a \neq 0$.

Kvadratická rovnica s jednou neznámou je najbežnejší prípad kvadratickej rovnice. Má všeobecný vzorec:

$$ax^2 + bx + c = 0$$

Člen a nazývame kvadratický člen, b lineárny člen a c absolútny člen, pričom $a \neq 0$. Ak $a = 0$, výsledkom je lineárna rovnica, ako sme už uviedli. Tak ako uvádza aj Zrebný (2010a) diskriminant kvadratickej rovnice je možné určiť podľa vzorca:

$$D = b^2 - 4ac$$

Z toho vyplývajú tri typy výsledku:

- Ak je diskriminant väčší ako 0 a všetky členy patria do množiny reálnych čísel, rovnica má dva reálne korene.
- Ak sa diskriminant rovná 0, rovnica má jeden dvojnásobný koreň.
- Ak je diskriminant menší ako 0, rovnica nemá v množine reálnych čísel riešenie, má však riešenie v množine komplexných čísel.

Riešenie kvadratickej rovnice s pomocou diskriminantu udáva vzorec:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

Iný pohľad na riešenie:

Ako uvádza Zrebný (2010b) kvadratická rovnica v normovanom tvare je rovnica v tvare:

$ax^2 + bx + c = 0$, kde $a = 1$.

Ak je $a \neq 1$, tak celú rovnicu vydelíme číslom a. Získame tak rovnicu:

$x^2 + px + q = 0$, kde $p = b / a$, $q = c / a$.

Tento typ rovnice môžeme okrem „riešenia s pomocou diskriminantu“ riešiť rozkladom na súčin s využitím vzťahov medzi koreňmi rovnice a koeficientmi p, q.

Platí:

$$x_1 \cdot x_2 = q$$

$$x_1 + x_2 = -p$$

Riešiť kvadratickú rovnicu týmto spôsobom je však pre strojové spracovanie výrazne zložitejšie, než riešenie s pomocou diskrimantu. Z tohto dôvodu ho zavrhujem, pretože by som nesplnila vlastnosť efektívnosti, ktorá okrem iného hovorí o tom, že by sme mali hľadať riešenie s čo možno najmenším počtom krokov.

Vstup (vstupné premenné):

a, b, c – koeficienty kvadratickej rovnice

Výstup (výstupné premenné):

$x1$, $x2$ – korene rovnice

Výpis: Nekonečne veľa riešení.

Výpis: Nemá riešenie.

Výpis: Lineárna rovnica s koreňom x.

Výpis: Reálny dvojnásobný koreň.

Výpis: 2 reálne rôzne korene.

Výpis: Riešenie v množine komplexných čísiel.

Vstupné podmienky:

$a, b, c \in R$

Výstupné podmienky:

$x1, x2 \in R$

Výpis: Nekonečne veľa riešení – ak $a = 0 \wedge b = 0 \wedge c = 0$.

Výpis: Nemá riešenie – ak $a = 0 \wedge b = 0 \wedge c \neq 0$.

Výpis: Lineárna rovnica s koreňom x – ak $a = 0 \wedge b \neq 0$ tak $x1 = -c / b$.

Výpis: Reálny dvojnásobný koreň – ak $d = 0$, tak $x1 = x2 = -b / (2a)$.

Výpis: 2 reálne rôzne korene – ak $d > 0$, tak $ax1^2 + bx1 + c = 0 \wedge ax2^2 + bx2 + c = 0$.

Výpis: Riešenie v množine komplexných čísiel – ak $d < 0$.

Pomocné premenné:

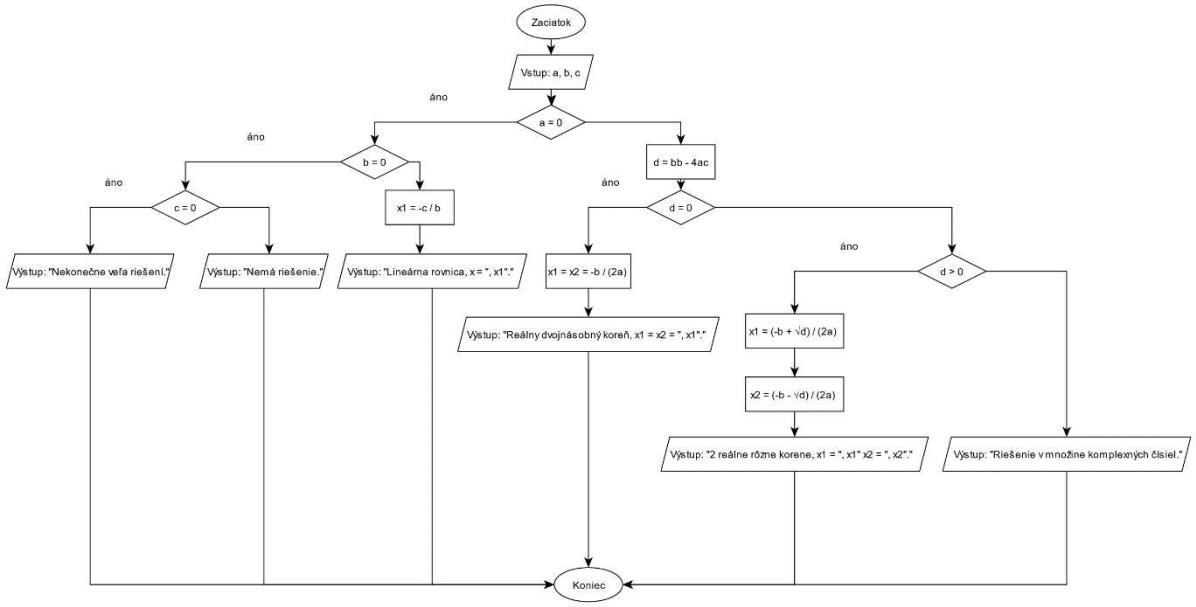
d – diskriminant

Podmienky pre pomocné premenné:

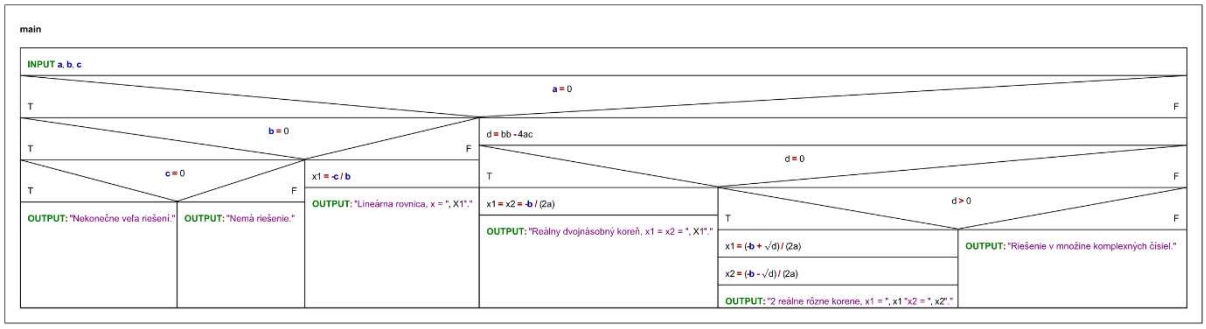
$d \in R \wedge d = b^2 - 4ac$

Pozn. autora: Všimnite si, že pri definícii vstupov a výstupov, nie vždy tieto musia byť definované s pomocou premennej. Výstupom môže byť aj nejaká informácia, napr. výpis „Nekonečne veľa riešení“, alebo kombinácia výpisu a obsahu premennej, napr. „Lineárna rovnica s koreňom x.“

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 28 Vývojový diagram na riešenie kvadratickej rovnice komplexne.



Obr. 29 NS diagram na riešenie kvadratickej rovnice komplexne.

Nutné vlastnosti algoritmu:

- **Determinovanosť –** táto vlastnosť algoritmu vyjadruje, že každý krok algoritmu musí byť jednoznačne a presne definovaný. To znamená, že v každej situácii musí byť úplne zrejmé, čo a ako sa má vykonať – ako postupuje tok riadenia algoritmu a ako má vykonávanie algoritmu pokračovať.

  V prvom kroku algoritmu sa načítajú hodnoty *a, b* a *c*. V závislosti od ich hodnôt sa počítajú korene, alebo vypisujú jednotlivé možnosti riešenia, pričom pri každom rozhodovacom procese je jednoznačne určené, kade bude algoritmus pokračovať pri splnení/nesplnení podmienky (sú označené vetvy rozhodovacieho procesu, T = true = pravda = áno, F = false = nepravda = nie). Je teda zrejmé, že realizátor má jednoznačne definované činnosti v každom kroku a tieto činnosti sú aj korektne

zakreslené. Častá chyba študentov je, že zabúdajú označiť pri rozhodovacom procese kladnú a zápornú vetvu rozhodovacieho procesu a tým je determinovanosť porušená.

- **Konečnosť –** každý algoritmus musí skončiť po vykonaní konečného počtu krokov. Riešenie pozostáva len zo sekvencie (ktorá nemení tok riadenia – primárne je postup vykonávania zhora nadol, resp. zprava doľava) a zloženej podmienky. Preto, v tomto prípade je konečnosť zabezpečená správne navrhnutými podmienkami pri zloženej podmienke.
- **Rezultatívnosť** – hovorí o tom, že algoritmus dáva po konečnom počte krokov správny výsledok, ktorý je pre rovnaké vstupné údaje vždy rovnaký. Ak algoritmus zabezpečí splnenie vstupnej a výstupnej podmienky, hovoríme, že je správny.

  Pozn. **autora:** Ak by sme nepoužili správne zátvorky pri výpočte koreňov *x1* a *x2*, porušili by sme túto vlastnosť. Napríklad vyhodnotenie s pomocou výrazu: $x1 = (-b + \sqrt{d}) / 2a$, nie je to isté ako: $x1 = (-b + \sqrt{d}) / (2a)$. V prvom prípade by sa vypočítal čitateľ, ten by sa vydelil dvoma a následne vynásobil hodnotou *a*.

  Vstupná podmienka $a, b, c \in R$ v prípade tvorby programu bude zabezpečená deklaráciou premenných požadovaného dátového typu: *float* alebo *double*. Ošetrenie pri zadávaní vstupov v tomto prípade nie je nevyhnutné, pretože pracujeme na celej množine *R*.

  Výstupná podmienka $x1, x2 \in R$ bude zabezpečená deklarovaním premenných požadovaného dátového typu a ostatné časti výstupných podmienok budú zabezpečené správne navrhnutým a zrealizovaným algoritmom.

Odporúčané/očakávané vlastnosti algoritmu:

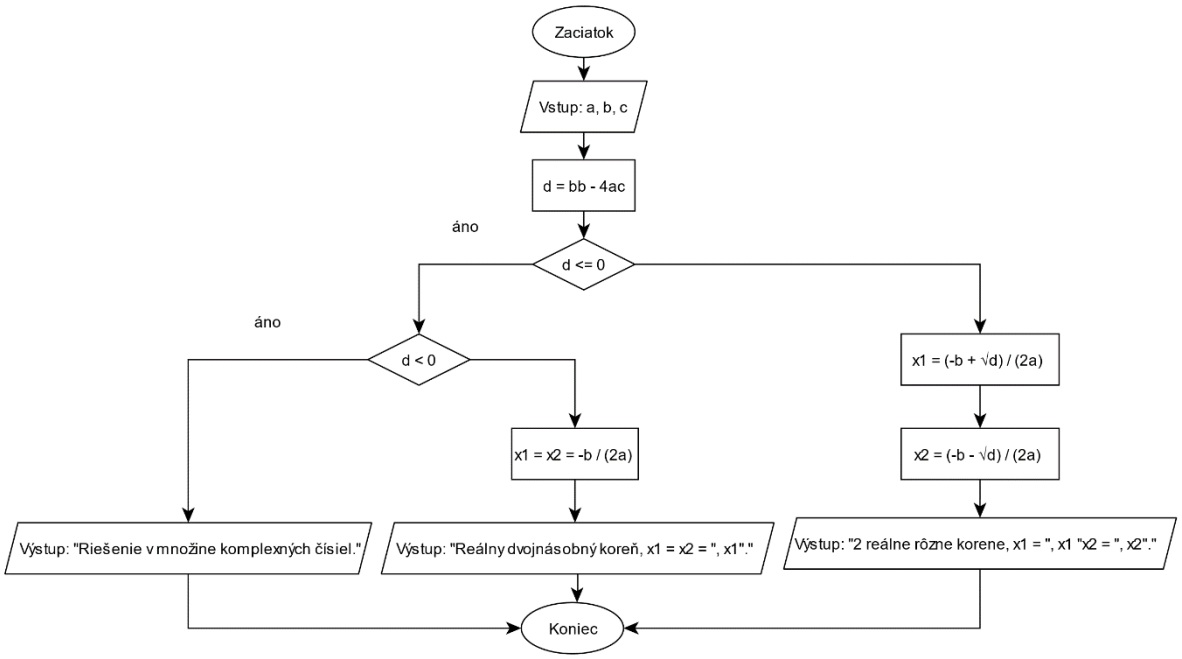
- **Hromadnosť** – môžeme konštatovať že dané riešenie spĺňa túto vlastnosť, t. j. algoritmus je platný nielen pre konkrétne hodnoty *a*, *b* a *c*, ale pre celú prípustnú množinu hodnôt riešeného problému, čo je zadefinované vstupnou podmienkou (všetky reálne čísla).
- **Elementárnosť** – vlastnosť je splnená, algoritmus je zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, v ktorom sa algoritmus bude realizovať (implementovať).
- **Efektívnosť** – znamená s čo možno s najmenším počtom krokov získať riešenie za najkratší čas (časová zložitosť algoritmu) a s využitím minimálneho množstva výpočtových prostriedkov (priestorová zložitosť algoritmu). Môžeme konštatovať, že aj

táto vlastnosť je splnená. V prípade, ak by sme na výpočet koreňa lineárnej rovnice definovali ďalšiu premennú, algoritmus by bol síce plne funkčný, avšak zbytočne by sme navýšili priestorovú náročnosť. Aktuálne algoritmus vyžaduje použitie šiestych premenných. Priestorové nároky v závislosti od hodnôt vstupných premenných sú počas činnosti algoritmu nemenné, z čoho vyplýva, že priestorová zložitosť je konštantná → S(1).

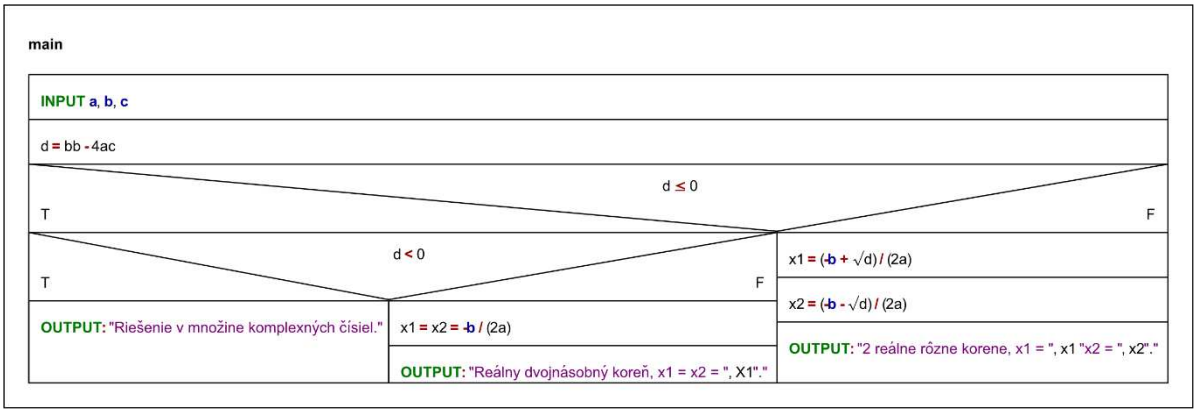
Časová zložitosť je konštantná → O(1), pretože počet vykonaní príkazov sa „výrazne“ nemení v závislosti od hodnôt vstupných koeficientov.

Pozn. autora: Slovo „výrazne“ si treba predstaviť tak, že či už sa vykoná jedna alebo desať operácií, nijako to výrazne neovplyvní rád zložitosti algoritmu. Bližšie sa čitateľ o časovej a priestorovej zložitosti dočíta v kapitole 2.

Ak by sme doplnili vstupnú podmienku o $a \neq 0$ (uplatnili abstrakciu), čiže sústredili by sme sa len čisto na riešenie kvadratickej rovnice, tak sa celé riešenie značne zjednoduší.



Obr. 30 Vývojový diagram na riešenie kvadratickej rovnice pri obmedzení $a \neq 0$.

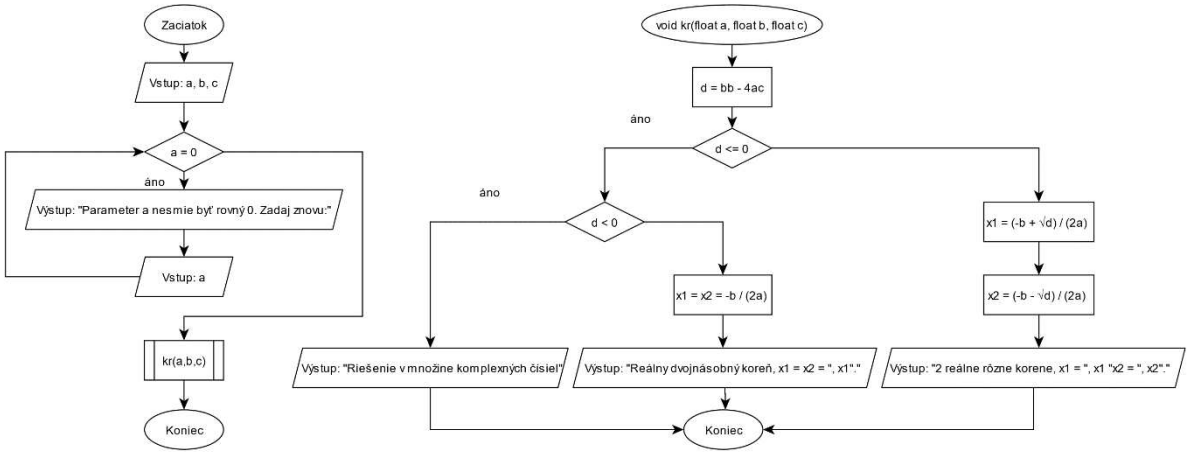


Obr. 31 NS diagram na riešenie kvadratickej rovnice pri obmedzení $a \neq 0$.

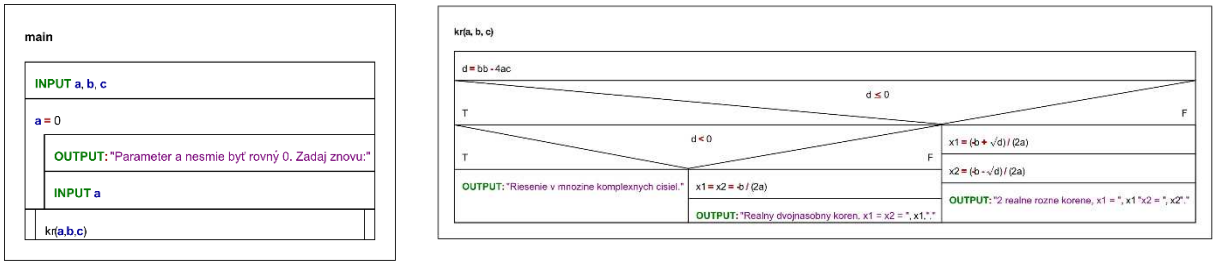
Pri posudzovaní efektívnosti by sme sa mali pozrieť aj na vlastnosti modifikovateľnosti a štruktúrovanosti, ktoré hovoria o tom, že algoritmus má byť navrhnutý tak, aby poskytol možnosť jednoduchej zmeny (úpravy) toku údajov, ako aj samotného algoritmu, a aby pozostával zo samostatných ale logicky previazaných celkov. Aby bola táto vlastnosť úplne splnená, mali by sme kľúčové činnosti realizovať cez funkciu.

Ak budeme výpočet realizovať s pomocou funkcie, tak by sme mali priestorovú aj časovú zložitosť (viac o časovej a priestorovej zložitosti sa čitateľ dozvie v kapitole 2) vyjadriť trochu inak:

Priestorová zložitosť	main()	S(1) = konštantná Používajú sa tri premenné: *a*, *b* a *c*.
	kr()	S(1) = konštantná Funkcia má tri parametre *a*, *b* a *c* a jednu premennú *d*.
Časová zložitosť	main()	O(1) = konštantná
	kr()	O(1) = konštantná



Obr. 32 Vývojový diagram na riešenie kvadratickej rovnice s pomocou procedúry.



Obr. 33 NS diagram na riešenie kvadratickej rovnice s pomocou procedúry.

V takomto prípade by sme však mohli namietať, že funkcia nie je navrhnutá dostatočne univerzálne. Vždy totižto poskytne prislúchajúce výpisy výsledkov. Čo ak ale potrebujeme len korene rovnice vynásobiť, či ich nejako inak spracovať a nepotrebujeme poznať ich hodnoty? V takom prípade by bolo vhodné, aby táto funkcia bola spracovaná inak – viac o funkciách v druhom diele učebnice.

Pozn. autora: Ak čitateľ cíti miernu frustráciu z problematiky spojenej s funkciami, môže sa k týmto častiam opätovne vrátiť po preštudovaní druhého diela učebnice, kde sú funkcie detailnejšie vysvetlené.

Overenie správnosti algoritmu:

Ako sme uviedli v kapitole 1.6, na overenie správnosti algoritmu existuje viacero prístupov. Tento krok je veľmi dôležitý, avšak mnohí študenti ho často ignorujú. Neuvedomujú si skutočnosť, že v prípade návrhu algoritmu a jeho reprezentácie graficky je väčšinou jediná možnosť overenia správnosti algoritmu krokovanie, resp. iná vhodná forma. Pri krokovaní si do pomocnej tabuľky najčastejšie zapisujeme hodnoty používaných premenných spolu s popisom toho čo sa deje v jednotlivých krokoch algoritmu. Ak algoritmus nie je správny, ak nerieši zadanú úlohu, tak bola celá naša činnosť zbytočná. Každý jeden algoritmus po jeho

návrhu je možné otestovať krokovaním, mení sa len reprezentácia zápisu. Aj keď študenti častejšie využívajú možnosť jeho implementácie priamo v programovacom jazyku a následnom otestovaní na počítači. Netreba zabúdať, že je možné mnohé riešenia „krokovať na papieri“ a overiť tak ich správnosť.

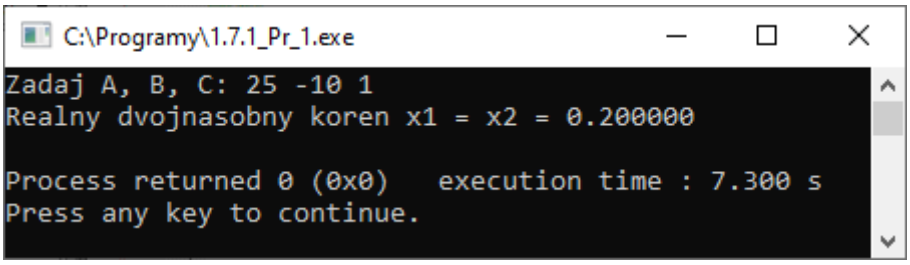
Ďalšou častou chybou študentov býva, že sa obmedzia na otestovanie jednej, prípadne dvoch situácií. Je však dôležité otestovať všetky možné kombinácie prípustných, ako aj neprípustných vstupov, pretože ak algoritmus dáva správne výsledky pre jednu konkrétnu vstupnú hodnotu alebo kombináciu vstupov, to ešte neznamená, že je správny. Na to potrebujeme ukázať, že dáva správne výsledky pre všetky prípustné kombinácie vstupných údajov spĺňajúcich vstupnú podmienku.

V prípade výpočtu koreňov kvadratickej rovnice podľa prvého návrhu môže nastať šesť rôznych situácií:

1. Ak $a = 0 \wedge b = 0 \wedge c = 0$ v takom prípade sa vypíše informácia: „Nekonečne veľa riešení.“
2. Ak $a = 0 \wedge b = 0 \wedge c \neq 0$ v takom prípade sa vypíše informácia: „Nemá riešenie.“
3. Ak $a = 0 \wedge b \neq 0 \wedge c \neq 0$ v takom prípade sa najprv výpočíta koreň lineárnej rovnice a následne sa vypíše informácia: „Lineárna rovnica s koreňom x = …“
   Napr. ak $a = 0$, $b = 2$, $c = 24$ tak $x1 = -c / b = -24 / 2 = -12$.
4. Ak $d = 0$, tak riešením je jeden reálny, tzv. dvojnásobný koreň, ktorý sa určí na základe vzťahu $x1 = x2 = -b / (2a)$.
   Napr. ak $a = 25$, $b = -10$, $c = 1$ tak $x1 = x2 = -b / (2a) = 10 / (2 \,.\, 25) = 0{,}2$.
5. Ak $d > 0$, tak riešením sú dva reálne rôzne korene, ktoré počítame na základe vzťahu
   $$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$
   Napr. ak $a = 1$, $b = 2$, $c = -63$ tak
   $$x1 = \frac{-b + \sqrt{D}}{2a} = \frac{-2 + \sqrt{2^2 - 4 \,.\, 1 \,.\, (-63)}}{2 \,.\, 1} = \frac{-2 + \sqrt{256}}{2} = \frac{-2 + 16}{2} = 7$$
   $$x2 = \frac{-b - \sqrt{D}}{2a} = \frac{-2 - \sqrt{2^2 - 4 \,.\, 1 \,.\, (-63)}}{2 \,.\, 1} = \frac{-2 - \sqrt{256}}{2} = \frac{-2 - 16}{2} = -9$$
6. Ak $d < 0$ tak riešenie na množine reálnych čísiel neexistuje a vypíše sa „Riešenie v množine komplexných čísiel.“

Jedna z možných implementácií prvého návrhu 1.7.1_Pr_1.c:

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    float a, b, c, d, x1, x2;
    printf("Zadaj A, B, C: ");
    scanf("%f %f %f", &a, &b, &c);

    if (a == 0)
    {
        if (b == 0)
        {
            if (c == 0)
                printf("Nekonecne vela rieseni.\n");
            else
                printf("Nema riesenie.\n");
        }
        else
        {
            x1 = -c / b;
            printf("Linearna rovnica, x = ", x1, ".\n");
        }
    }
    else
    {
        d = b * b - 4 * a * c;
        if (d == 0)
        {
            x1 = x2 = -b / (2 * a);
            printf("Realny dvojnasobny koren x1 = x2 = %f\n", x1);
        }
        else
        {
            if (d > 0)
            {
                x1 = (-b + sqrt(d)) / (2 * a);
                x2 = (-b - sqrt(d)) / (2 * a);
                printf("2 realne rozne korene x1 = %f, x2 = %f\n", x1, x2);
            }
            else
                printf("Riesenie v mnozine komplexnych cisiel.\n");
        }
    }

    return 0;
}
```



Obr. 34 Konzolový výstup programu 1.7.1_Pr_1.c.

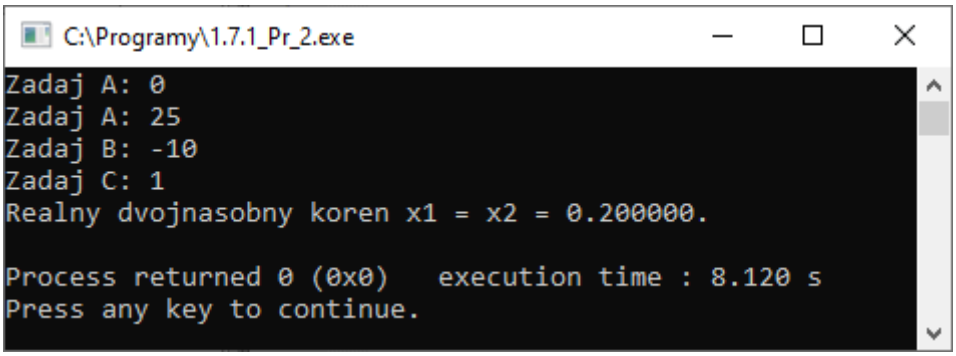
Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 24 B/48 B = *a* (*float*/*double* = 4 B/8 B) + *b* (*float*/*double* = 4 B/8 B) + *c* (*float*/*double* = 4 B/8 B) + *d* (*float*/*double* = 4 B/8 B) + *x1* (*float*/*double* = 4 B/8 B) + *x2* (*float*/*double* = 4 B/8 B) = 24 B/48 B.

Pri zjednodušenom riešení na základe doplnenej vstupnej podmienky $a \neq 0$ je vhodné pri realizovaní programu doplniť ošetrenie vstupnej premennej na základe definovanej podmienky. Je to tak z dvoch dôvodov: ak by *a* bolo rovné 0, tak by išlo o riešenie lineárnej rovnice, ktoré tu neriešime, v opačnom prípade by nastalo delenie 0.

Jedna z možných implementácií druhého návrhu 1.7.1_Pr_2.c:

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    float a, b, c, d, x1, x2;
    do
    {
        printf("Zadaj A: ");
        scanf("%f", &a);
    }
    while (a == 0);

    printf("Zadaj B: ");
    scanf("%f", &b);
    printf("Zadaj C: ");
    scanf("%f", &c);
    d = b * b - 4 * a * c;
    if (d <= 0)
        if (d < 0)
            printf("Riesenie v mnozine komplexnych cisiel.\n");
        else
        {
            x1 = x2 = -b / (2 * a);
            printf("Realny dvojnasobny koren x1 = x2 = %f.\n", x1);
        }
    else
    {
        x1 = (-b + sqrt(d)) / (2 * a);
```

```
30          x2 = (-b - sqrt(d)) / (2 * a);
31          printf("2 realne rozne korene x1 = %f, x2 = %f.\n", x1, x2);
32      }
33
34      return 0;
35  }
```



Obr. 35 Konzolový výstup programu 1.7.1_Pr_2.c.

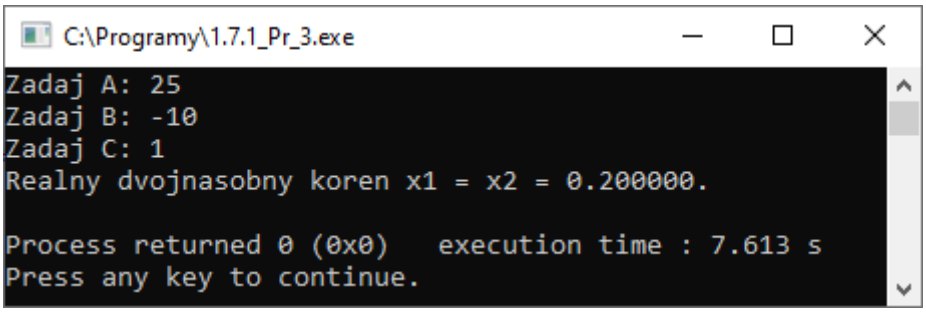
Pozn. autora: Ako môžete vidieť z kódu programu, načítanie koeficientov *a*, *b* a *c*, sme oddelili. Je to z dôvodu, že koeficient *a* má obmedzenie, ktoré je nevyhnutné testovať. Ak by sme načítanie robili s pomocou jedného príkazu, nastalo by opätovne vyžiadanie hodnôt všetkých troch koeficientov, čo by nebolo žiaduce.

```
do
{
    printf("Zadaj A, B, C: ");
    scanf("%f %f %f", &a, &b, &c);
}
while (a == 0);
```

Jedna z možných implementácií tretieho návrhu 1.7.1_Pr_3.c:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  // deklaracia procedury
5  void kr(float a, float b, float c);
6
7  int main(void)
8  {
9      float a, b, c;
10     do
11     {
12         printf("Zadaj A: ");
13         scanf("%f", &a);
14     }
15     while (a == 0);
16
17     printf("Zadaj B: ");
18     scanf("%f", &b);
19     printf("Zadaj C: ");
20     scanf("%f", &c);
```

```
21      kr(a, b, c);
22
23      return 0;
24  }
25
26  // definicia procedury
27  void kr(float a, float b, float c)
28  {
29      float x1, x2, d = b * b - 4 * a * c;
30      if (d <= 0)
31          if (d < 0)
32              printf("Riesenie v mnozine komplexnych cisiel.\n");
33          else
34          {
35              x1 = x2 = -b / (2 * a);
36              printf("Realny dvojnasobny koren x1 = x2 = %f.\n", x1);
37          }
38      else
39      {
40          x1 = (-b + sqrt(d)) / (2 * a);
41          x2 = (-b - sqrt(d)) / (2 * a);
42          printf("2 realne rozne korene x1 = %f, x2 = %f.\n", x1, x2);
43      }
44  }
```



Obr. 36 Konzolový výstup programu 1.7.1_Pr_3.c.

Ak výpočet realizujeme s pomocou funkcie, tak by sme mali priestorovú aj časovú zložitosť vyjadriť trochu inak:

Priestorové nároky	main()	*a* (*float*/*double* = 4 B/8 B) + *b* (*float*/*double* = 4 B/8 B) + *c* (*float*/*double* = 4 B/8 B) = **12 B/24 B**
	kr()	*a* (*float*/*double* = 4 B/8 B) + *b* (*float*/*double* = 4 B/8 B) + *c* (*float*/*double* = 4 B/8 B) + *d* (*float*/*double* = 4 B/8 B) + *x1* (*float*/*double* = 4 B/8 B) + *x2* (*float*/*double* = 4 B/8 B) = **24 B/48 B**

Treba si uvedomiť, že priestorové nároky funkcie *kr()* vznikajú, keď nastáva jej vykonávanie a zanikajú v momente, keď nastane jej ukončenie. Priestorové nároky funkcie *main()* sú trvalé od začiatku až po koniec programu.

1.7.2 Príklad 2 – súčin dvoch čísiel s pomocou sčítania

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte s pomocou vývojového diagramu (VD) a NS diagramu riešenie. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém násobenia dvoch celých kladných čísel *a*, *b* s pomocou sčítania.

Rozbor úlohy: Vzhľadom na zadanú podmienku, že je nevyhnutné súčin čísiel realizovať s pomocou sčítania, je potrebné túto úlohu realizovať s pomocou cyklu, v ktorom sa bude hodnota jedného čísla navyšovať o hodnotu samého seba toľkokrát, koľko je hodnota druhého čísla. V takomto prípade je možné použiť na riešenie cyklus s pevne definovaným počtom opakovaní. Z pohľadu zakreslenia je to najprehľadnejšie riešenie. Na základe faktu, že každý *for* cyklus je možné realizovať s pomocou cyklu s podmienkou na začiatku, za istých okolností aj s pomocou cyklu s podmienkou na konci, je na riešenie však možné použiť všetky tri cykly.

Riešenie 1 – s pomocou cyklu s pevným počtom opakovaní

Vstup (vstupné premenné):

a – číslo 1 zadané používateľom

b – číslo 2 zadané používateľom

Výstup (výstupné premenné):

sucin – súčin čísiel *a* a *b*

Vstupné podmienky:

$a, b \in Z^+ \wedge sucin = 0$

Výstupné podmienky:

$$sucin \in Z^+ \wedge sucin = a \,.\, b = \underbrace{a + a + \ldots + a}_{b\text{-krát}}$$

Pomocné premenné:

i – riadiaca premenná cyklu

Podmienky pre pomocné premenné:

$i \in \langle a, b \rangle$

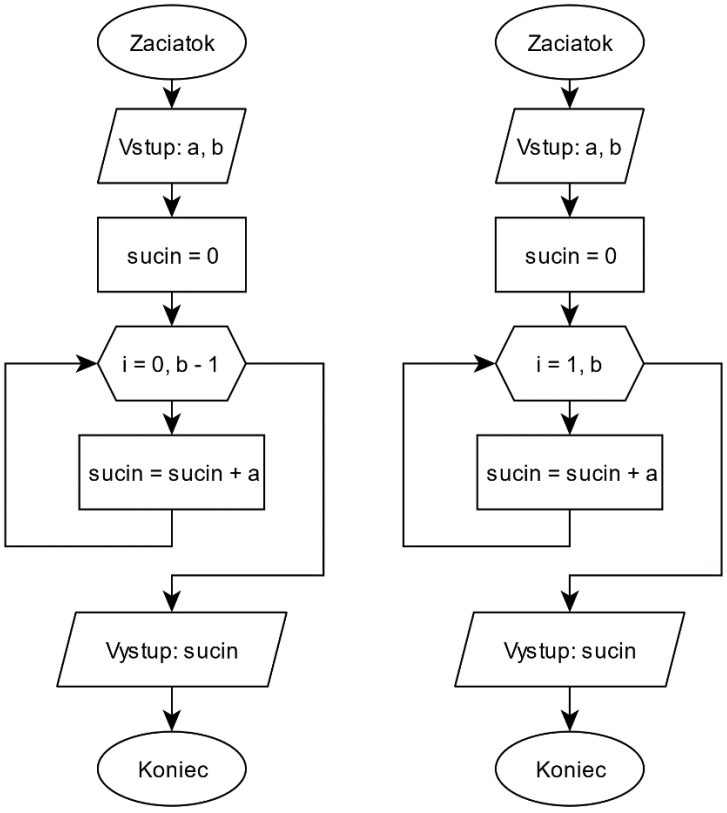
Algoritmus riešenia - slovné riešenie:

- Zavedieme pomocnú riadiacu premennú *i*, ktorá bude reprezentovať počet vykonávaní tela cyklu. V tele cyklu výstupnú premennú *sucin* využijeme ako pomocnú premennú, ktorá bude reprezentovať čiastočný súčin.
- Na začiatku majú *i* a *sucin* nulové hodnoty, potom sa bude k *sucin* *b*-krát pripočítavať hodnota *a*, súčasne sa hodnota *i* bude zvyšovať o jednotku.

Algoritmus riešenia s prvkami pseodojazyka:

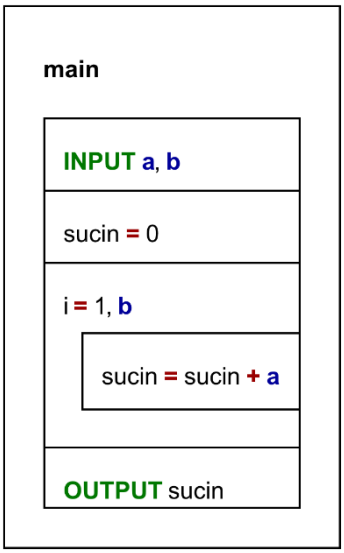
1. Polož $i = 0$, $sucin = 0$.
2. Pripočítaj *b*-krát k *sucin* hodnotu *a*, k *i* pripočítaj 1.
3. Následne vypíš hodnotu v premennej *sucin*.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 37 Vývojový diagram na riešenie súčinu dvoch čísiel s pomocou sčítania.

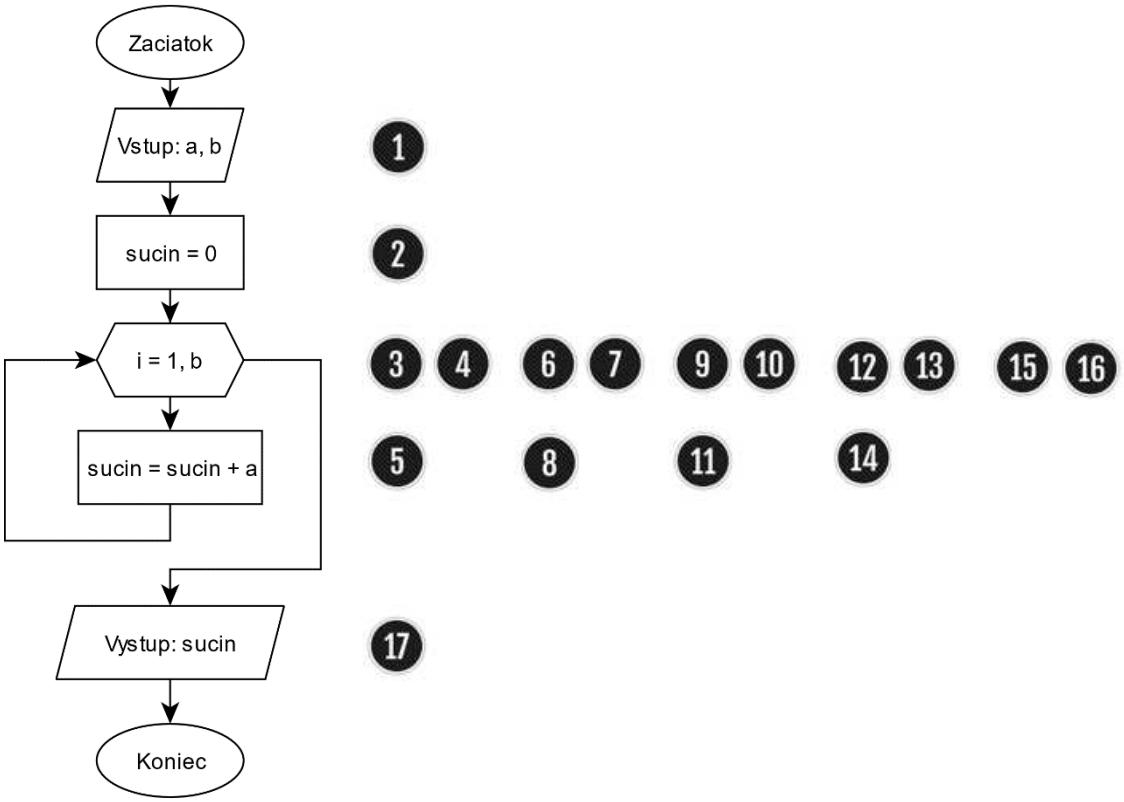
Pozn. autora: Riešenie na obr. 37 vpravo predpokladá, že riadiaca premenná *i* bude inicializovaná na 1, pričom telo cyklu sa bude vykonávať až kým riadiaca premenná neprekoná hodnotu *b*. Takéto zakreslenie je na prvý pohľad prehľadnejšie než v prípade riešenia vľavo. Samozrejme, že obidve reprezentácie sú správne, počet opakovaní tela cyklu je *b*-krát.



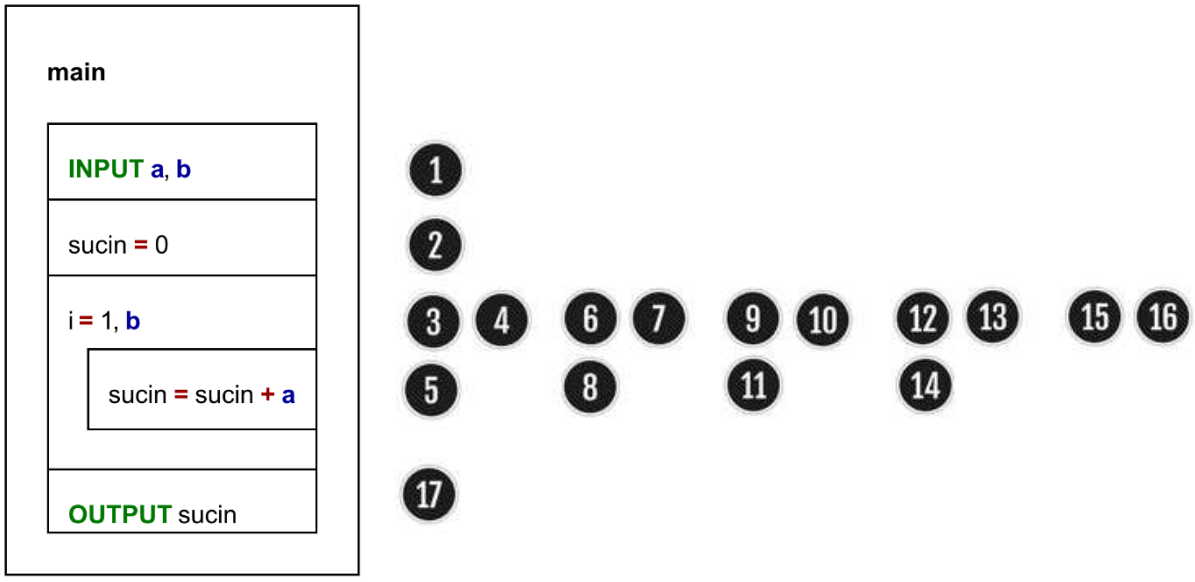
Obr. 38 NS diagram na riešenie súčinu dvoch čísiel s pomocou sčítania.

Overenie správnosti algoritmu:

Pri jednotlivých akciách algoritmu reprezentovaného diagramom sú uvedené čísla krokov, ktoré korešpondujú s krokmi ilustrovanými v tabuľke.



Obr. 39 Vývojový diagram na riešenie súčinu dvoch čísiel s pomocou sčítania – krokovanie.

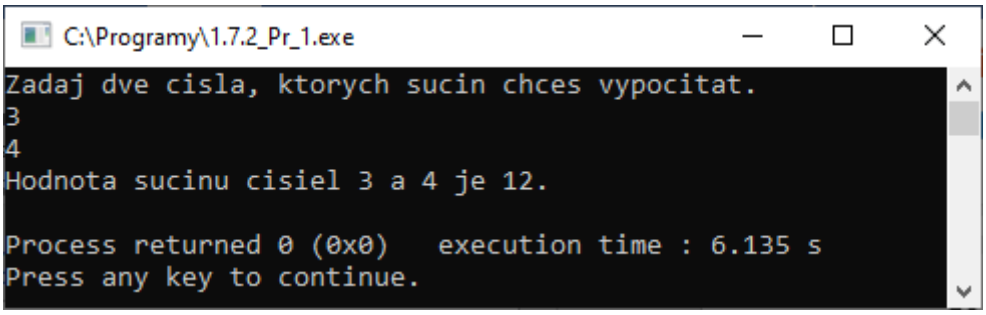


Obr. 40 NS diagram na riešenie súčinu dvoch čísiel s pomocou sčítania – krokovanie.

Krok	Hodnoty sledovaných premenných				Popis jednotlivých krokov
	a	b	i	sucin	
❶	3	4			Načítanie vstupných premenných *a*, *b*
❷				0	Inicializácia premennej *sucin*
❸			1		Inicializácia riadiacej premennej
❹					Testovanie $i \leq b$, 1 je menšie rovné ako 4
❺				3	Navýšenie premennej *sucin* o hodnotu *a*
❻			2		Inkrementácia riadiacej premennej
❼					Testovanie $i \leq b$, 2 je menšie rovné ako 4
❽				6	Navýšenie premennej *sucin* o hodnotu *a*
❾			3		Inkrementácia riadiacej premennej
❿					Testovanie $i \leq b$, 3 je menšie rovné ako 4
⓫				9	Navýšenie premennej *sucin* o hodnotu *a*
⓬			4		Inkrementácia riadiacej premennej
⓭					Testovanie $i \leq b$, 4 je rovné ako 4
⓮				12	Navýšenie premennej *sucin* o hodnotu *a*
⓯			5		Inkrementácia riadiacej premennej
⓰					Testovanie $i \leq b$, 5 nie je menšie rovné ako 4 → koniec cyklu
⓱					Výstup premennej *sucin*

Jedna z možných implementácií 1.7.2_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    unsigned int a, b, sucin = 0, i;
    do
    {
        printf("Zadaj dve cisla, ktorych sucin chces vypocitat.\n");
        scanf("%d %d", &a, &b);
    }
    while (a <= 0 || b <= 0);
    for (i = 0; i < b; i++)
        sucin += a;
    // for(i = 1; i <= b; i++) sucin += a;      // druha alternativa pouzitia
    printf("Hodnota sucinu cisiel %d a %d je %d.\n", a, b, sucin);

    return 0;
}
```



Obr. 41 Konzolový výstup programu 1.7.2_Pr_1.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 16 B = *a* (*unsigned int* = 4 B) + *b* (*unsigned int* = 4 B) + *sucin* (*unsigned int* = 4 B) + *i* (*unsigned int* = 4 B).

Riešenie 2 – s pomocou cyklu s podmienkou na začiatku/konci

Vstupné premenné:

a – číslo 1 zadané používateľom

b – číslo 2 zadané používateľom

Výstupné premenné:

sucin – súčin čísiel *a* a *b*

Vstupné podmienky:

$a, b \in Z^+ \wedge sucin = 0 \wedge p = 0$

Výstupné podmienky:

$sucin \in Z^+ \wedge sucin = a \,.\, b = \underbrace{a + a + \ldots + a}_{b\text{-krát}}$

Pomocné premenné:

p – počet sčítancov

Podmienky pre pomocné premenné:

$p \in \langle a, b \rangle$

Algoritmus riešenia – slovné riešenie:

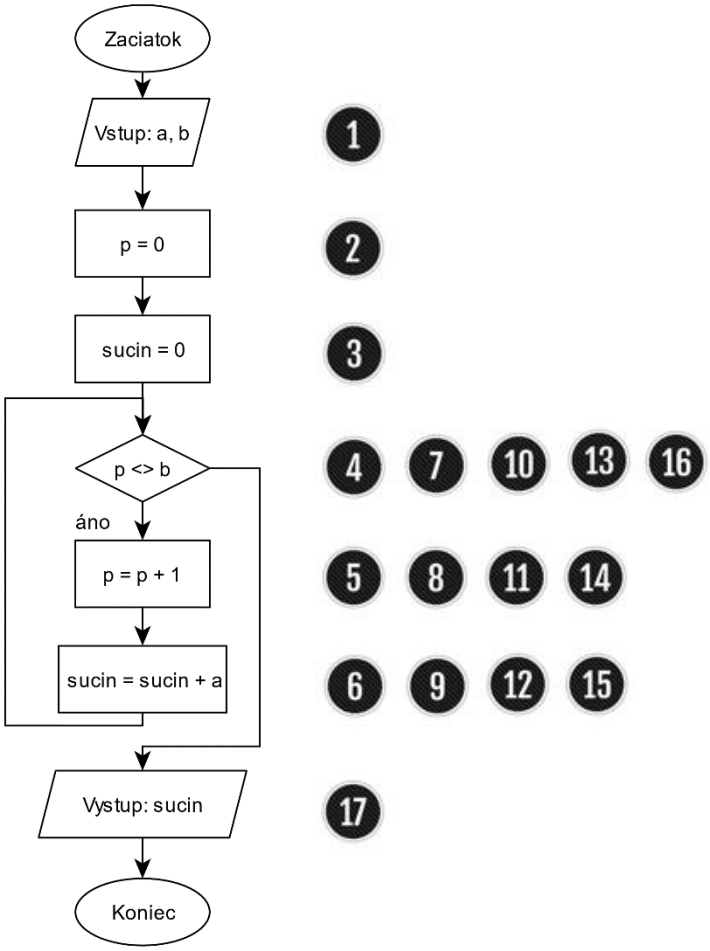
- Zavedieme pomocnú premennú *p*, ktorá bude reprezentovať počet sčítancov a výstupnú premennú *sucin* využijeme počas realizácie algoritmu ako pomocnú premennú, ktorá bude reprezentovať čiastočný súčin.

- Na začiatku majú *p* a *sucin* nulové hodnoty, potom sa bude k *sucin* postupne pripočítavať hodnota *a*, súčasne sa hodnota *p* bude zvyšovať o jednotku.
- Tento postup sa bude opakovať dovtedy, pokiaľ počet sčítancov *p* nedosiahne hodnotu *b*.

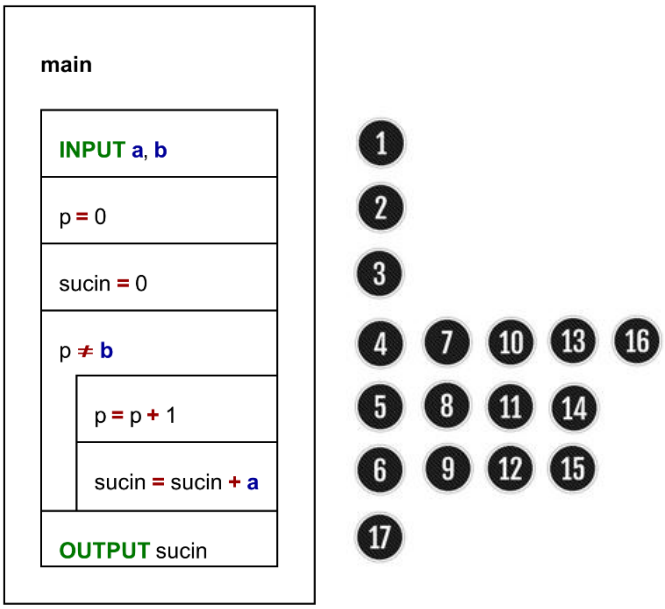
Algoritmus riešenia – pseodojazyk:

1. Polož $p = 0$, $sucin = 0$.
2. Pripočítaj k *sucin* hodnotu *a*, k *p* pripočítaj 1.
3. Ak $p = b$, tak ukonči cyklus a vypíš hodnotu v premennej *sucin*, inak pokračuj bodom 2.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 42 Vývojový diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s cyklom s podmienkou na začiatku.



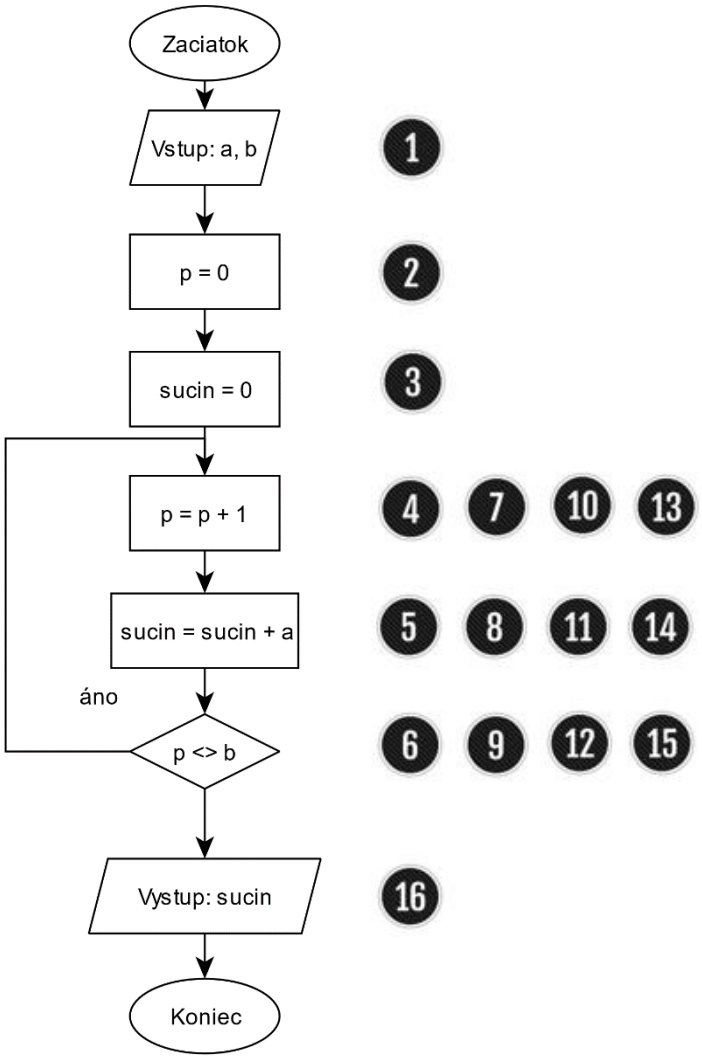
Obr. 43 NS diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s cyklom s podmienkou na začiatku.

Overenie správnosti algoritmu:

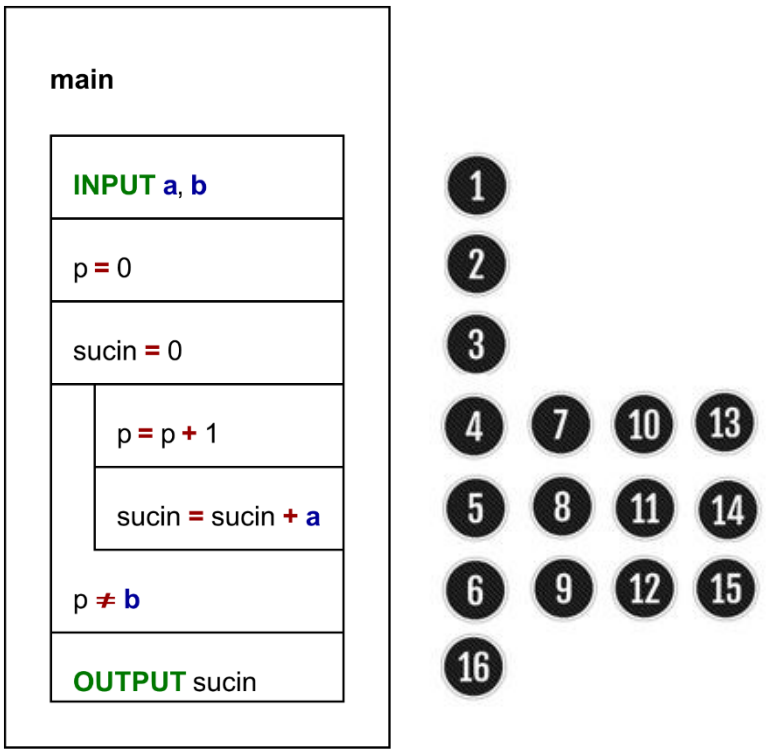
Pri jednotlivých akciách algoritmu reprezentovaného diagramom sú uvedené čísla krokov, ktoré korešpondujú s krokmi ilustrovanými v tabuľke.

Krok	Hodnoty sledovaných premenných				Popis jednotlivých krokov
	a	b	p	sucin	
1	3	4			Načítanie vstupných premenných *a*, *b*
2			0		Inicializácia premennej *p*
3				0	Inicializácia premennej *sucin*
4					Testovanie $p \neq b$, 0 je rôzne od 4
5			1		Inkrementácia premennej *p*
6				3	Navýšenie premennej *sucin* o hodnotu *a*
7					Testovanie $p \neq b$, 1 je rôzne od 4
8			2		Inkrementácia premennej *p*
9				6	Navýšenie premennej *sucin* o hodnotu *a*
10					Testovanie $p \neq b$, 2 je rôzne od 4
11			3		Inkrementácia premennej *p*
12				9	Navýšenie premennej *sucin* o hodnotu *a*
13					Testovanie $p \neq b$, 3 je rôzne od 4

⑭			4		Inkrementácia premennej *p*
⑮				12	Navýšenie premennej *sucin* o hodnotu *a*
⑯					Testovanie $p \neq b$, 4 nie je rôzne od 4 → koniec cyklu
⑰					Výstup premennej *sucin*



Obr. 44 Vývojový diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s cyklom s podmienkou na konci.



Obr. 45 NS diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s cyklom s podmienkou na konci.

Overenie správnosti algoritmu:

Pri jednotlivých akciách algoritmu reprezentovaného diagramom sú uvedené čísla krokov, ktoré korešpondujú s krokmi ilustrovanými v tabuľke.

Krok	Hodnoty sledovaných premenných				Popis jednotlivých krokov
	a	b	p	sucin	
❶	3	4			Načítanie vstupných premenných *a*, *b*
❷			0		Inicializácia premennej *p*
❸				0	Inicializácia premennej *sucin*
❹			1		Inkrementácia premennej *p*
❺				3	Navýšenie premennej *sucin* o hodnotu *a*
❻					Testovanie $p \neq b$, 1 je rôzne od 4
❼			2		Inkrementácia premennej *p*
❽				6	Navýšenie premennej *sucin* o hodnotu *a*
❾					Testovanie $p \neq b$, 2 je rôzne od 4
❿			3		Inkrementácia premennej *p*

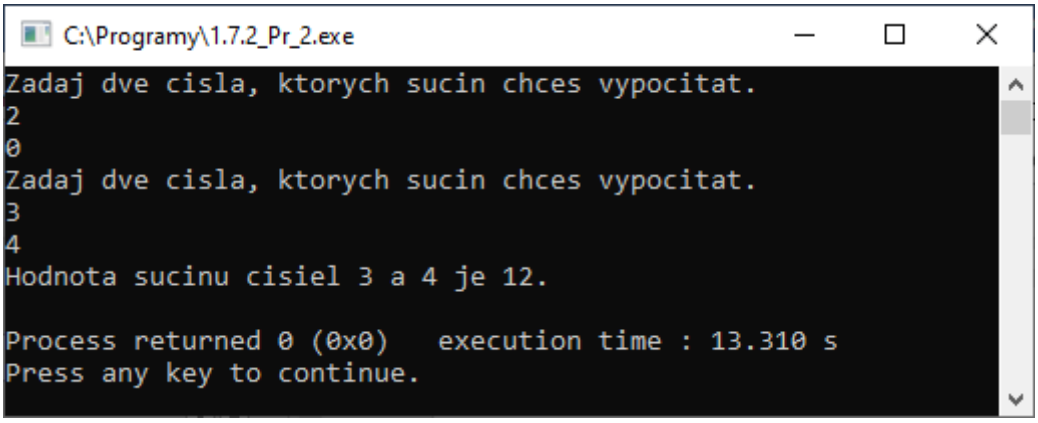
⓫				9	Navýšenie premennej *sucin* o hodnotu *a*
⓬					Testovanie $p \neq b$, 3 je rôzne od 4
⓭			4		Inkrementácia premennej *p*
⓮				12	Navýšenie premennej *sucin* o hodnotu *a*
⓯					Testovanie $p \neq b$, 4 nie je rôzne od 4 → koniec cyklu
⓰					Výstup premennej *sucin*

Pozn. autora: Na prvý pohľad sa by mohlo zdať že riešenie s pomocou cyklu s podmienkou na začiatku, resp. na konci je ekvivalentné. Dané riešenie pre konkrétne hodnoty bude vždy poskytovať korektné výsledky, ak však zoberieme do úvahy skutočnosť, že v prípade cyklu s podmienkou na konci sa telo cyklu vykoná vždy minimálne jedenkrát, je riešenie s pomocou cyklu s podmienkou na začiatku vhodnejšie. Teda napr. pri vstupoch $a = 0$ a $b = 0$, je výsledok 0. Pri riešení, kde je použitý cyklus s podmienkou na začiatku, nastane vyhodnotenie podmienky cyklu a telo sa nevykoná ani raz, oproti riešeniu s pomocou cyklu s podmienkou na konci. Tam sa vykonajú dva príkazy úplne zbytočne. Na druhej strane, ak sa spätne pozrieme, ako sme zadefinovali vstupnú podmienku: $a, b \in Z^+$, je zrejmé, že 0 predstavuje nekorektný vstup. Preto v tomto prípade celá vyššie uvedená myšlienka nemá opodstatnenie, avšak v iných prípadoch môže byť veľmi dôležitá.

Jedna z možných implementácií 1.7.2_Pr_2.c:

```c
#include <stdio.h>

int main(void)
{
    unsigned int a, b, sucin = 0, p = 0;
    do
    {
        printf("Zadaj dve cisla, ktorych sucin chces vypocitat.\n");
        scanf("%d %d", &a, &b);
    }
    while (a <= 0 || b <= 0);
    while (p < b)
    {
        p++;
        sucin += a;
    }

    // druha alternativa s pouzitym cyklu s podmienkou na konci
    /*
    do{
        p++;
        sucin += a;
    }while (p < b);
```

```
24      */
25      printf("Hodnota sucinu cisiel %d a %d je %d.\n", a, b, sucin);
26
27      return 0;
28  }
```



Obr. 46 Konzolový výstup programu 1.7.2_Pr_2.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 16 B = *a* (*unsigned int* = 4 B) + *b* (*unsigned int* = 4 B) + *sucin* (*unsigned int* = 4 B) + *p* (*unsigned int* = 4 B).

Nutné vlastnosti algoritmu:

- **Determinovanosť** – táto vlastnosť algoritmu vyjadruje, že každý krok algoritmu musí byť jednoznačne a presne definovaný, t. j. v každej situácii musí byť úplne zrejmé, čo a ako sa má vykonať – ako postupuje tok riadenia algoritmu a ako má vykonávanie algoritmu pokračovať. Po načítaní vstupných premenných *a* a *b*, sa nastavia počiatočné hodnoty premenných *p* / *i* a *sucin*, čo reprezentujú vstupné podmienky. Ak by sme premenné neinicializovali, len ťažko by sme mohli očakávať korektné výsledky, resp. len ťažko by bolo možné algoritmus jednoznačne interpretovať. Následne musí realizátor vedieť vykonať jednoduchý súčet čísel. Ďalší krok – ukončenie algoritmu je viazané na splnenie podmienky. V prípade splnenia podmienky sa musí zopakovať telo cyklu. Je teda zrejmé, že realizátor má jednoznačne definované činnosti v každom kroku a tieto činnosti sú aj korektne zakreslené. Častá chyba študentov je, že zabúdajú označiť pri rozhodovacom procese cyklu s podmienkou na začiatku/konci kladnú a zápornú vetvu a tým je determinovanosť porušená.
- **Konečnosť** – každý algoritmus musí skončiť po vykonaní konečného počtu krokov. Riešenie pozostáva len zo sekvencie a cyklu. Pri použití prvého algoritmu riešenia je

konečnosť zabezpečená s pomocou cyklu s presne definovaným počtom opakovaní. Pri riešeniach s využitím cyklu s podmienkou na začiatku/konci je konečnosť zabezpečená správne navrhnutou podmienkou cyklu $p \neq b$, pri nastavení počiatočnej hodnoty p na 0, a pri zadaní vstupnej premennej b väčšej ako p, čo je ošetrené vo vstupných podmienkach. V každej iterácii cyklu sa p zvyšuje o 1. Čo v konečnom dôsledku zabezpečí konečnosť algoritmu.

- **Rezultatívnosť** – hovorí o tom, že algoritmus dáva po konečnom počte krokov správny výsledok, ktorý je pre rovnaké vstupné údaje vždy rovnaký. Ak algoritmus zabezpečí splnenie vstupnej a výstupnej podmienky, hovoríme, že je správny.

  Pozn. **autora:** Ak by sme neinicializovali korektne pomocnú premennú *p / i* a výstupnú premennú *sucin* na začiatku na 0, tak by sme túto vlastnosť nesplnili. Nie je teda žiadnou zvláštnosťou definovať vstupnú podmienku pre výstupnú premennú. Pri definovaní výstupnej podmienky študenti často zabúdajú na jej druhú časť, ktorá hovorí o tzv. korektnosti (správnosti) hodnoty. Ak by sme ju vynechali, tak by akékoľvek číslo z množiny Z^+ bolo riešením, čo nie je pravdou. V tomto príklade sa pripočítavanie začne od nuly, a keď počet sčítancov bude rovný b, dosiahne sa splnenie výstupnej podmienky.

  Vstupná podmienka $a, b \in Z^+$ v prípade tvorby programu bude zabezpečená deklaráciou premenných požadovaného dátového typu: *unsigned int* a ošetrením pri načítavaní:

  - Využitie podmienky pri zadávaní vstupu:
    - *if* (a <= 0 OR b <= 0) Zadaj hodnoty znovu
    - Toto riešenie umožní používateľovi opraviť zadané hodnoty len raz, čo nie je príliš používateľsky prívetivé (*user friendly*).
  - Vhodnejšie je využitie cyklu pri zadávaní vstupu:
    - *do* Zadaj hodnoty znovu *while* (a <= 0 OR b <= 0)
    - *while* (a <= 0 OR b <= 0) Zadaj hodnoty znovu
    - Použitie cyklu s podmienkou na konci eliminuje duplicitu kódu. V prípade použitia cyklu s podmienkou na začiatku musia byť najprv načítané hodnoty vstupných premenných a a b, a až následne môžu byť testované. V prípade, že by tieto hodnoty nezodpovedali vstupným podmienkam, nastane ich opätovné načítanie vo vnútri cyklu.

Toto opätovné načítavanie odpadá pri použití cyklu s podmienkou na konci.

Vstupná podmienka $p = 0$ a $sucin = 0$ bude zrealizovaná s pomocou operátora priradenia. Výstupná podmienka $sucin \in Z^+$ bude zabezpečená deklarovaním premennej požadovaného dátového typu a druhá časť podmienky bude zabezpečená správne navrhnutým a zrealizovaným algoritmom.

Odporúčané/očakávané vlastnosti algoritmu:

- **Hromadnosť** – môžeme konštatovať že dané riešenie spĺňa túto vlastnosť vzhľadom na zadanie problému, t. j. algoritmus je platný nielen pre konkrétne hodnoty *a*, *b*, ale pre celú prípustnú množinu hodnôt, čo je zadefinované vstupnou podmienkou (všetky kladné celé čísla).

  Pozn. autora: Ak by však zadanie bolo formulované inak, napr. riešte problém násobenia dvoch čísel *a*, *b.* Pre splnenie vlastnosti hromadnosti, by bolo potrebné pracovať na celej množine reálnych čísiel, aby algoritmus pracoval v celom rozsahu prípustných hodnôt. Obmedzenie práce nad množinou kladných celých čísiel nastalo z dôvodu podmienky realizovať súčin s pomocou súčtu.

- **Elementárnosť** – vlastnosť je splnená, algoritmus je zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, v ktorom sa algoritmus bude realizovať (implementovať).

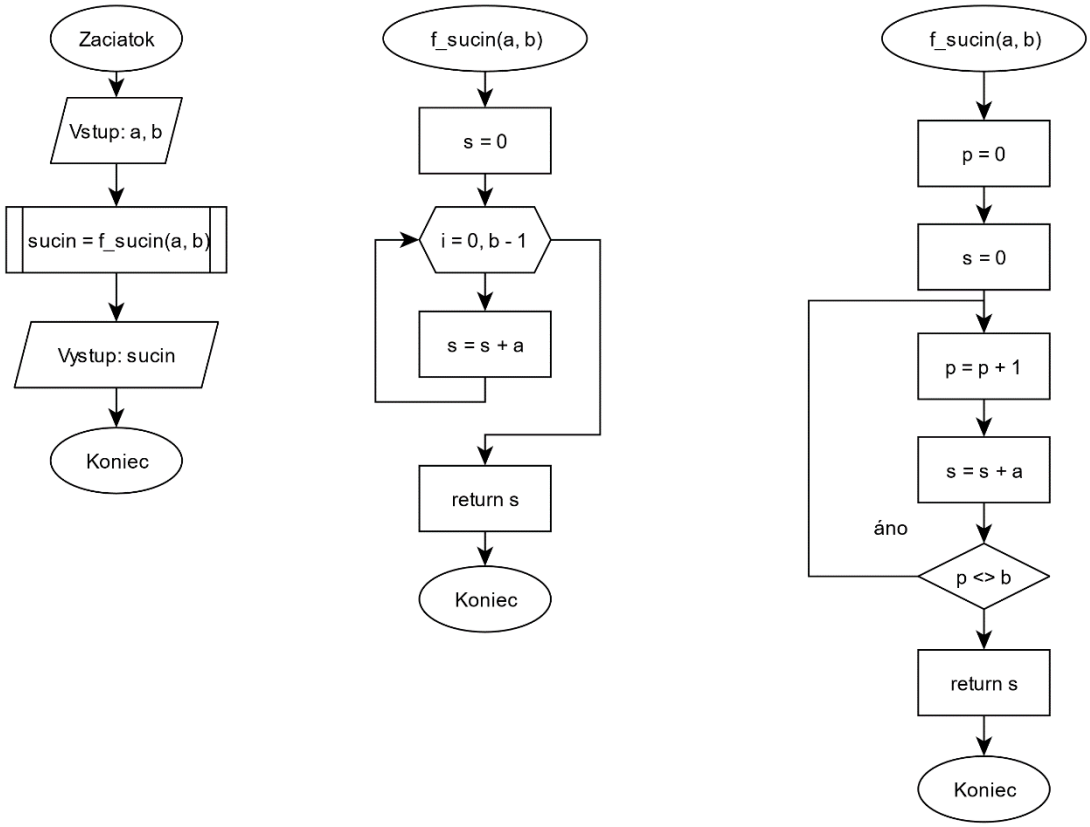
- **Efektívnosť** – znamená, s čo možno najmenším počtom krokov získať riešenie za najkratší čas (časová zložitosť algoritmu) a s využitím minimálneho množstva výpočtových prostriedkov (priestorová zložitosť algoritmu). Môžeme konštatovať, že aj táto vlastnosť je splnená. V prípade, že by sme pre čiastočný súčin definovali pomocnú premennú, algoritmus by bol síce plne funkčný, avšak zbytočne by sme navýšili priestorovú náročnosť. Momentálne algoritmus vyžaduje použitie štyroch premenných. Priestorové nároky v závislosti od hodnôt vstupných premenných sú počas činnosti algoritmu nemenné, z čoho vyplýva, že priestorová zložitosť je konštantná → S(1).

  Časová zložitosť je lineárna → O(n), pretože počet vykonaní príkazov tela cyklu je závislý od hodnoty vstupu *b*. Počet iterácií cyklu je teda *b*-krát. Ak teda určujeme časovú zložitosť pre veľké hodnoty vstupov, ktoré by mohli byť označené kľudne *n*, je zrejmý vzťah medzi nami definovaným vstupom a určenou časovou zložitosťou.

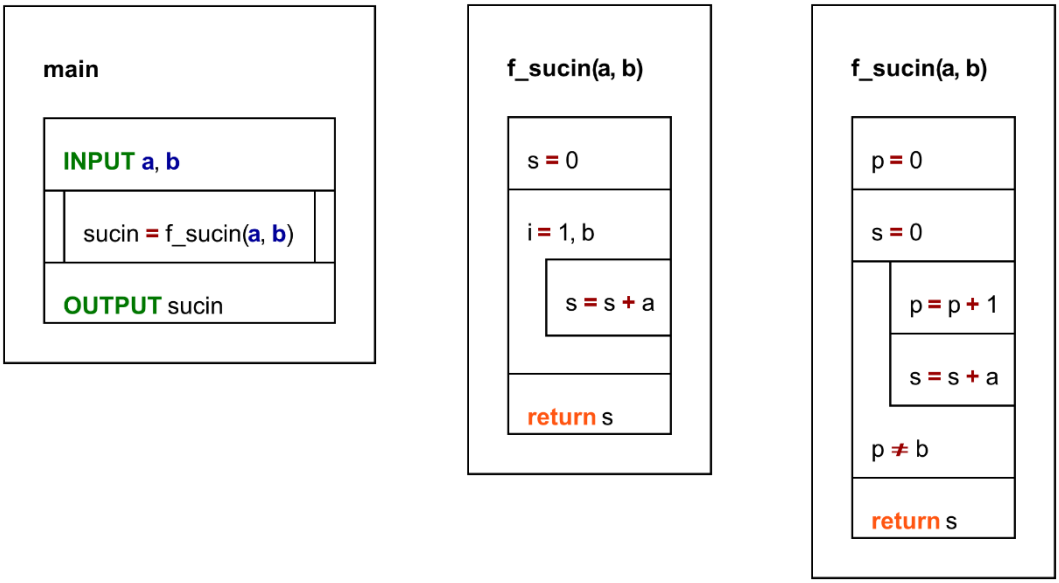
Pri posudzovaní efektívnosti by sme sa mali pozrieť aj na vlastnosti modifikovateľnosti a štruktúrovanosti, ktoré hovoria o tom, aby bol algoritmus navrhnutý tak, aby poskytol možnosť jednoduchej zmeny (úpravy) toku údajov, aj samotného algoritmu a aby pozostával zo samostatných, ale logicky previazaných celkov. Aby sme úplne splnili túto vlastnosť, mali by sme kľúčové činnosti realizovať cez funkciu. Avšak pri takomto elementárnom probléme to nie je príliš žiaduce. Opodstatnenie využívania funkcií je zrejmé pri návrhu komplexnejších algoritmov.

Ak výpočet súčinu čísiel *a* a *b* realizujeme s pomocou funkcie, tak by sme mali priestorovú aj časovú zložitosť vyjadriť trochu inak:

Priestorová zložitosť	main()	S(1) = konštantná Používajú sa tri premenné: *a*, *b* a *sucin*.
	f_sucin()	S(1) = konštantná Funkcia má dva parametre *a* a *b* a dve premenné *i / p* a *s*.
Časová zložitosť	main()	O(1) = konštantná
	f_sucin()	O(n) = lineárna



Obr. 47 Vývojový diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s funkciou.



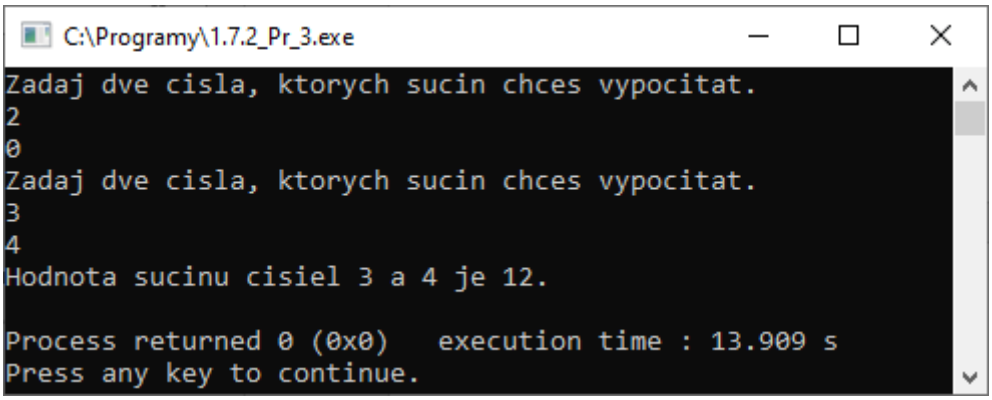
Obr. 48 NS diagram na riešenie súčinu dvoch čísiel s pomocou sčítania s funkciou.

Pozn. autora: Niekto by mohol namietať, že posledný príkaz vo funkcii reprezentovaný kľúčovým slovom *return*, porušuje vlastnosť algoritmu elementárnosti (vyvoláva dojem, že algoritmus viažeme na vývojové prostredie). Nie je však tak. Stačí sa na dané slovo pozrieť ako na anglický ekvivalent slovenského *vráť hodnotu*. Na dodržanie tejto vlastnosti je vo všeobecnosti vhodné vyhýbať sa použitiu operátorov jazyka *C* (napr. pri testovaní rovnosti nepoužívať ==, alebo pri inkrementovaní premennej o 1, nepoužívať iteračný operátor ++), ako aj vstavaných funkcií a pod.

Jedna z možných implementácií 1.7.2_Pr_3.c:

```c
#include <stdio.h>

// deklaracia funkcie
unsigned int f_sucin(unsigned int a, unsigned int b);

int main(void)
{
    unsigned int a, b, sucin;
    do
    {
        printf("Zadaj dve cisla, ktorych sucin chces vypocitat.\n");
        scanf("%d %d", &a, &b);
    }
    while (a <= 0 || b <= 0);

    // osetrenie vstupov s pomocou cyklu s podmienkou na zaciatku - podporuje redundantnost
    /*
    printf("Zadaj dve cisla, ktorych sucin chces vypocitat.\n");
    scanf("%d %d", &a, &b);
```

```
20      while (a <= 0 || b <= 0)
21      {
22              printf("Zadaj dve kladne cisla, ktorych sucin chces vypocitat.\n");
23              scanf("%d %d", &a, &b);
24      }
25      */
26
27      sucin = f_sucin(a, b); //volanie funkcie
28      printf("Hodnota sucinu cisiel %d a %d je %d.\n", a, b, sucin);
29
30      return 0;
31  }
32
33  // definicia funkcie
34  unsigned int f_sucin(unsigned int a, unsigned int b)
35  {
36      unsigned int s = 0, i;
37      for (i = 0; i < b; i++)
38          s += a;
39      return s;
40  }
41
42  // druha alternativa funkcie
43  /*
44  unsigned int f_sucin(unsigned int a, unsigned int b)
45  {
46      unsigned int s = 0, p = 0;
47      while (p < b)
48      {
49          p++;
50          s += a;
51      }
52      return s;
53  }
54  */
55  // tretia alternativa funkcie
56  /*
57  unsigned int f_sucin(unsigned int a, unsigned int b)
58  {
59      unsigned int s = 0, p = 0;
60      do
61      {
62          p++;
63          s += a;
64      }
65      while (p < b);
66      return s;
67  }
68  */
```



Obr. 49 Konzolový výstup programu 1.7.2_Pr_3.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, a vziať do úvahy aj použitie funkcie, tak by sme tieto mali vyjadriť trochu inak:

Priestorové nároky	main()	*a* (*unsigned int* = 4 B) + *b* (*unsigned int* = 4 B) + *sucin* (*unsigned int* = 4 B) = **12 B**
	f_sucin()	*a* (*unsigned int* = 4 B) + *b* (*unsigned int* = 4 B) + *p* / *i* (*unsigned int* = 4 B) + *s* (*unsigned int* = 4 B) = **16 B**

Treba si uvedomiť, že priestorové nároky funkcie *f_sucin()* vznikajú, keď nastáva jej vykonávanie a zanikajú v momente, keď nastáva jej ukončenie. Priestorové nároky funkcie *main()* sú trvalé od začiatku až po koniec programu.

1.7.3 Príklad 3 – najväčší spoločný deliteľ – Euklidov algoritmus

Euklidov algoritmus je v teórii čísel algoritmus na určenie najväčšieho spoločného deliteľa dvoch prirodzených čísel, často označovaný ako *NSD(a,b)*. Na jeho riešenie existuje viacero algoritmov, ktoré si rozoberieme.

Vstup (vstupné premenné):

a – prirodzené číslo zadané používateľom

b – prirodzené číslo zadané používateľom

Výstup (výstupné premenné):

nsd – najväčší spoločný deliteľ čísiel *a* a *b*

Vstupné podmienky:

$a, b \in N = \{1,2\ldots\infty\}$

Výstupné podmienky:

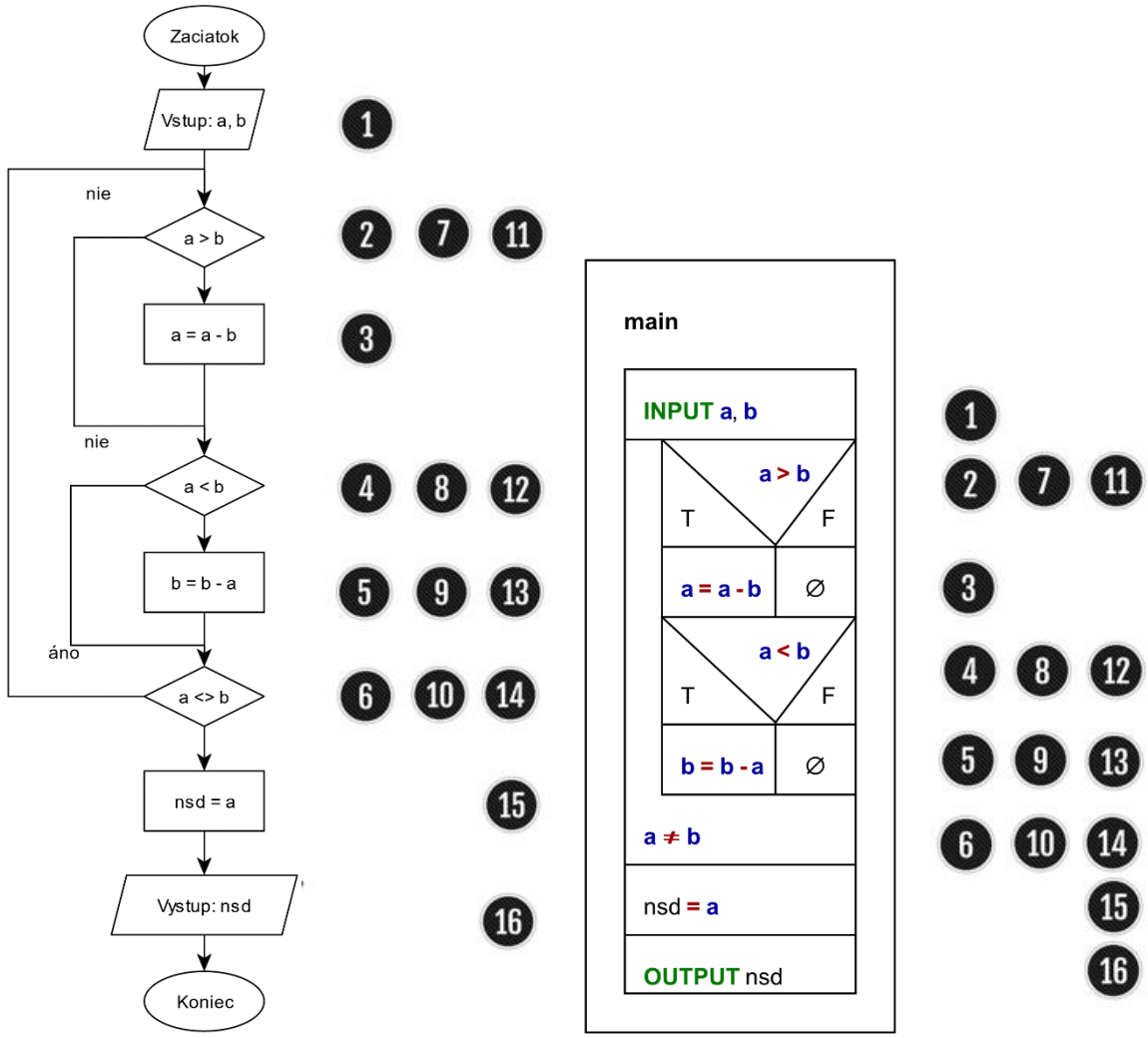
$nsd \in N = \{1,2\ldots \infty\} \wedge nsd$ je najväčší spoločný deliteľ čísiel a a b

Rozbor úlohy:

Algoritmus 1:

1. Majme dve prirodzené čísla *a* a *b*.
2. Ak sú obe rovnaké, tak skonči a *nsd* je toto číslo.
3. Inak odpočítaj od väčšieho menšie číslo, tento výsledok ulož do väčšieho čísla, menšie nechaj bezo zmeny a pokračuj predchádzajúcim bodom.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 50 Vývojový a NS diagram na nájdenie *NSD* – algoritmus 1.

Overenie správnosti algoritmu 1:

Realizácia algoritmu na príklade $a = 30$ a $b = 24$:

Krok	Hodnoty sledovaných premenných			Popis jednotlivých krokov
	a	b	nsd	
❶	30	24		Načítanie vstupných premenných *a*, *b*
❷				Testovanie $a > b$, 30 je viac ako 24
❸	6			Na základe splnenia podmienky $a = a - b = 30 - 24 = 6$
❹				Testovanie $a < b$, 6 je menej ako 24
❺		18		Na základe splnenia podmienky $b = b - a = 24 - 6 = 18$
❻				Testovanie $a \neq b$, 6 je rôzne od 18 → opakovanie cyklu
❼				Testovanie $a > b$, 6 nie je viac ako 18, nevykoná sa nič a pokračujeme ďalším testovaním
❽				Testovanie $a < b$, 6 je menej ako 18
❾		12		Na základe splnenia podmienky $b = b - a = 18 - 6 = 12$
❿				Testovanie $a \neq b$, 6 je rôzne od 12 → opakovanie cyklu
⓫				Testovanie $a > b$, 6 nie je viac ako 12, nevykoná sa nič a pokračujeme ďalším testovaním
⓬				Testovanie $a < b$, 6 je menej ako 12
⓭		6		Na základe splnenia podmienky $b = b - a = 12 - 6 = 6$
⓮				Testovanie $a \neq b$, 6 nie je rôzne od 6 → koniec cyklu
⓯			6	Priradenie hodnoty premennej *a* do premennej *nsd*
⓰				Výstup premennej *nsd*

Algoritmus 2:

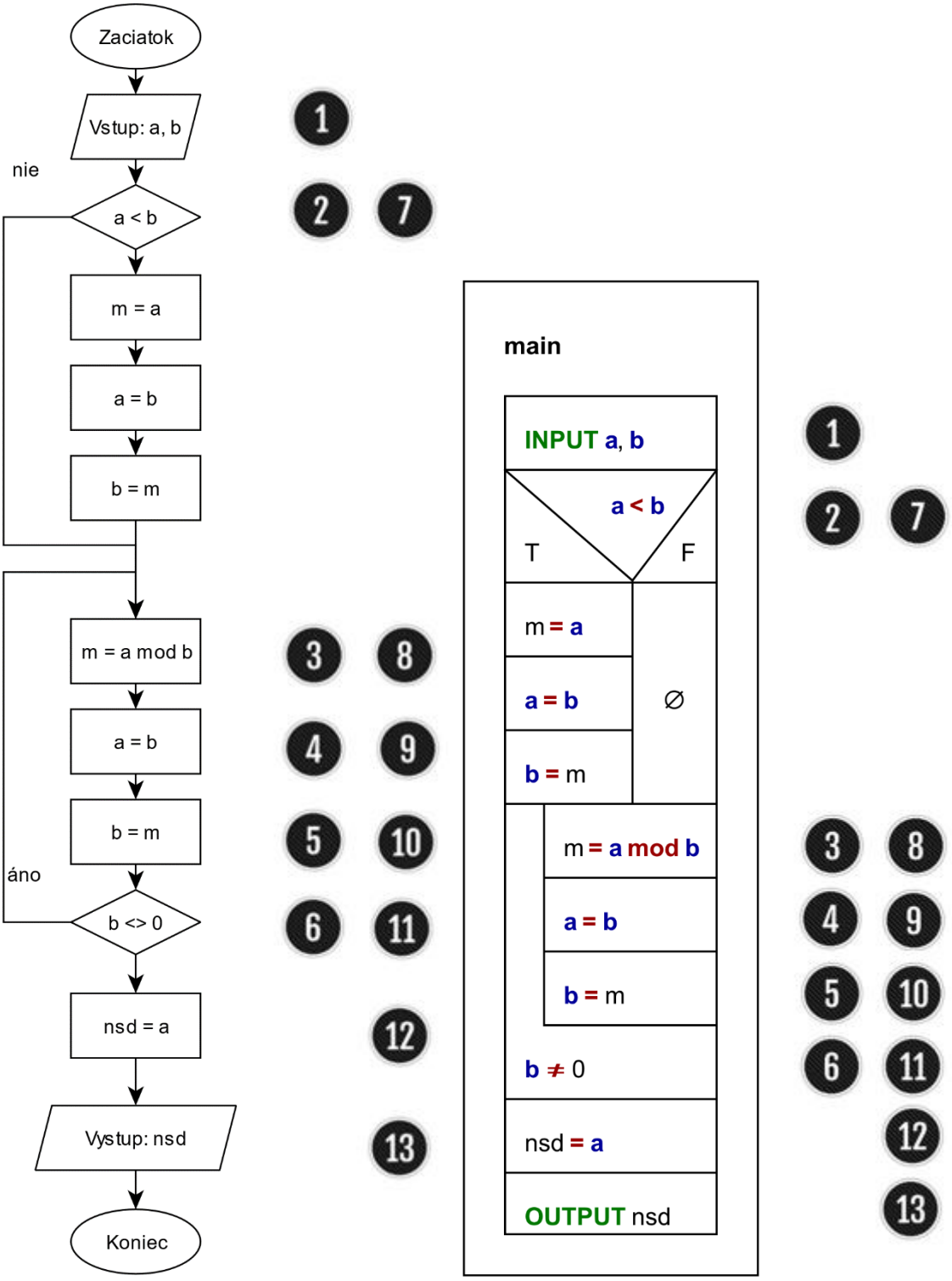
1. Majme dve prirodzené čísla *a*, *b*. Ak *a* je menšie ako *b*, vymeň ich hodnoty.
2. Pokiaľ *b* nie je nulové, opakuj:

   $m = a \bmod b$

   $a = b$

   $b = m$
3. Koniec algoritmu: v *a* je uložený najväčší spoločný deliteľ čísel *a* a *b*.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 51 Vývojový a NS diagram na nájdenie *NSD* – algoritmus 2.

Overenie správnosti algoritmu 2:

Realizácia algoritmu na príklade $a = 30$ a $b = 24$:

Krok	Hodnoty sledovaných premenných				Popis jednotlivých krokov
	a	b	m	nsd	
❶	30	24			Načítanie vstupných premenných *a*, *b*
❷					Testovanie $a < b$, 30 nie je menej ako 24, nevykoná sa nič a pokračujeme ďalším krokom algoritmu
❸			6		Zmena hodnoty premennej $m = 30 \bmod 24 = 6$
❹	24				Zmena hodnoty premennej $a = b = 24$
❺		6			Zmena hodnoty premennej $b = m = 6$
❻					Testovanie $b \neq 0$, 6 je rôzne od 0 → opakovanie cyklu
❼					Testovanie $a < b$, 24 nie je menej ako 6, nevykoná sa nič a pokračujeme ďalším krokom algoritmu
❽			0		Zmena hodnoty premennej $m = 24 \bmod 6 = 0$
❾	6				Zmena hodnoty premennej $a = b = 6$
❿		0			Zmena hodnoty premennej $b = m = 0$
⓫					Testovanie $b \neq 0$, 0 nie je rôzna od 0 → koniec cyklu
⓬				6	Priradenie hodnoty premennej *a* do premennej *nsd*
⓭					Výstup premennej *nsd*

Algoritmus 3:

Algoritmus riešenia – slovné riešenie: Budeme deliť vždy väčšie z dvoch čísiel *a*, *b* tým menším. Ak je zvyšok po delení 0, tak to menšie číslo je najväčší spoločný deliteľ, inak to väčšie číslo nahradíme zvyškom po delení a pokračujeme rovnakým spôsobom.

Algoritmus riešenia – slovné riešenie štruktúrovaná forma:

1. Ak *a* je menšie ako *b*, vymeň ich hodnoty.
2. Do *a* daj hodnotu zvyšku po delení *a* mod *b* (operácia modulo).
3. Keď sa *a* nerovná 0, choď na krok 1 a pracuj s novými hodnotami *a*, *b*.
4. Vráť *b* ako výsledok.

Algoritmus riešenia – pseodojazyk:

begin

Načítaj *a* a *b*;

repeat

if *a* < *b* **then**

Vymeň *a* a *b*;

a = *a* mod *b*;

until *a* <> 0;

nsd = *b;*

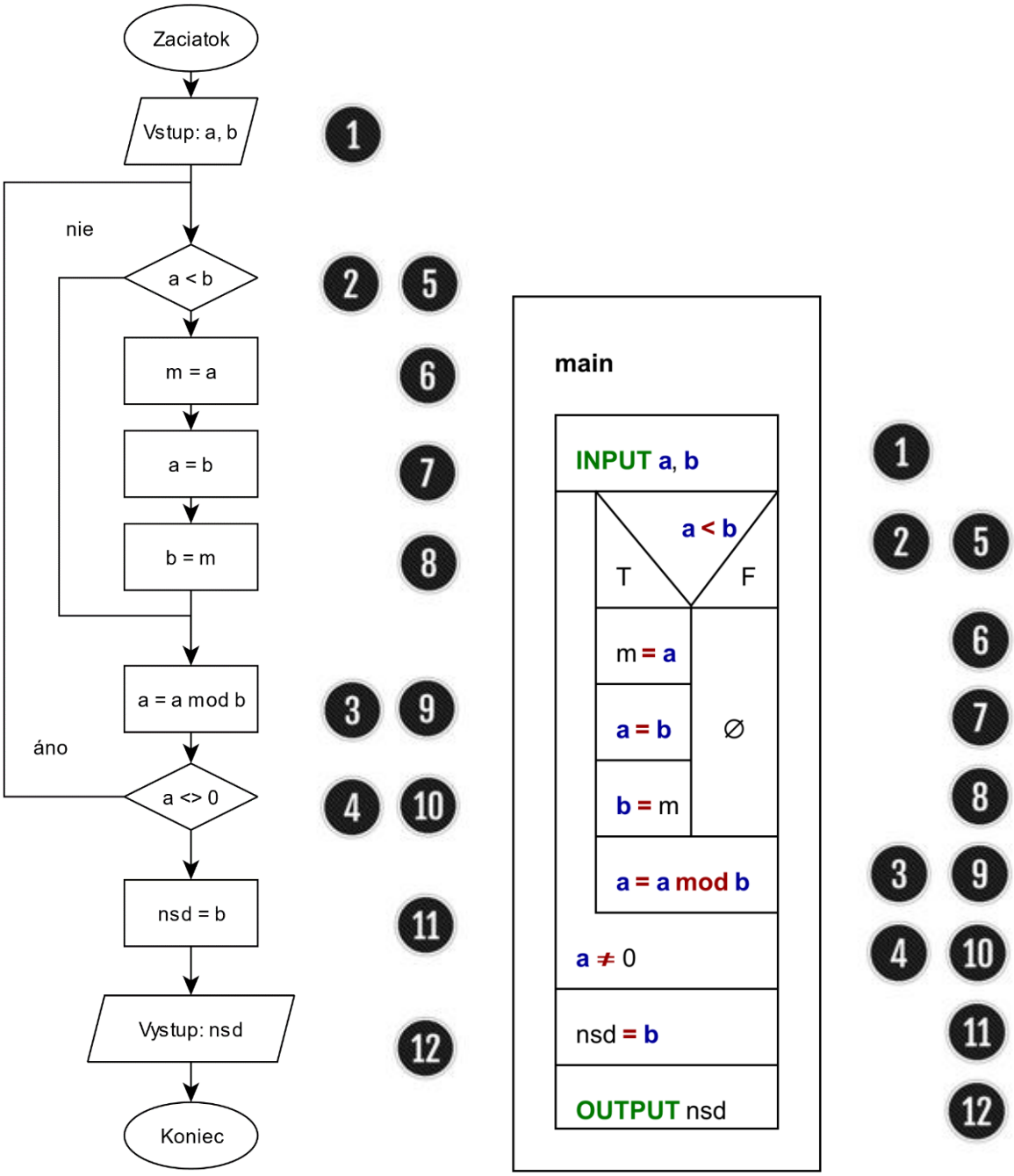
Vypíš *nsd*;

end

Pozn. autora: Podľa definície výstupu, by sme mali na základe použitého algoritmu buď hodnotu premennej *a* alebo *b* uložiť do výstupnej premennej *nsd*, a následne vypísať až jej obsah. Je zrejmé, že takto navyšujeme priestorové nároky algoritmu, avšak z pohľadu jednoznačnej reprezentácie vstupov a výstupov sa toto často deje. Samozrejme, že pri programovej realizácii, toto už nie je nevyhnutné, t. j. v prípade algoritmizácie sú často navyšované priestorové nároky z dôvodu prehľadnosti a jednoznačnosti reprezentácie, voči implementácii. Všimnite si, že v prípade grafickej reprezentácie riešenia s pomocou funkcie, je už táto skutočnosť reprezentovaná absolútne korektne aj bez tohto zbytočného priradenia.

Na strane druhej je možné sa na algoritmus pozerať len ako na funkčnú časť spracovania daných vstupov, a v takom prípade načítanie vstupov a výpis výstupov nepovažujeme za súčasť algoritmu.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 52 Vývojový a NS diagram na nájdenie *NSD* – algoritmus 3.

Overenie správnosti algoritmu 3:

Realizácia algoritmu na príklade $a = 30$ a $b = 24$:

Krok	Hodnoty sledovaných premenných				Popis jednotlivých krokov
	a	b	m	nsd	
❶	30	24			Načítanie vstupných premenných *a*, *b*
❷					Testovanie $a < b$, 30 nie je menej ako 24, nevykoná sa nič a pokračujeme ďalším krokom algoritmu
❸	6				Zmena hodnoty premennej $a = a \bmod b = 30 \bmod 24 = 6$
❹					Testovanie $a \neq 0$, 6 je rôzne od 0 → opakovanie cyklu
❺					Testovanie $a < b$, 6 je menej ako 24, nastane výmena hodnôt
❻			6		Zmena hodnoty premennej $m = a = 6$
❼	24				Zmena hodnoty premennej $a = b = 24$
❽		6			Zmena hodnoty premennej $b = m = 6$
❾	0				Zmena hodnoty premennej $a = a \bmod b = 24 \bmod 6 = 0$
❿					Testovanie $a \neq 0$, 0 nie je rôzna od 0 → koniec cyklu
⓫				6	Priradenie hodnoty premennej *b* do premennej *nsd*
⓬					Výstup premennej *nsd*

Posúdenie analyzovaných algoritmov na základe priestorovej a časovej zložitosti:

	Algoritmus 1	Algoritmus 2	Algoritmus 3
Priestorová zložitosť	2 premenné → S(1)	3 premenné → S(1)	3 premenné → S(1)
Časová zložitosť	O(n)	O(n)	O(n)
Záver	optimálne riešenie		

Pozn. autora: Keďže časová a priestorová zložitosť často stoja proti sebe, tak sa v súčasnosti skôr kladie dôraz na rýchlosť výpočtu voči priestorovým nárokom. Z pohľadu časovej zložitosti sú dané algoritmy ekvivalentné. Pri priestorovej zložitosti sú rozdiely pri prvom algoritme, tieto sú však zanedbateľné, avšak na istej úrovni detailnosti preukázateľné. Dokonca aj z pohľadu náročnosti implementácie sú dané algoritmy rovnocenné. Mohli by sme teda povedať, že všetky

analyzované riešenia sú vyhovujúce a rozdiely medzi nimi sú v tomto prípade skutočne zanedbateľné.

Nutné vlastnosti algoritmu:

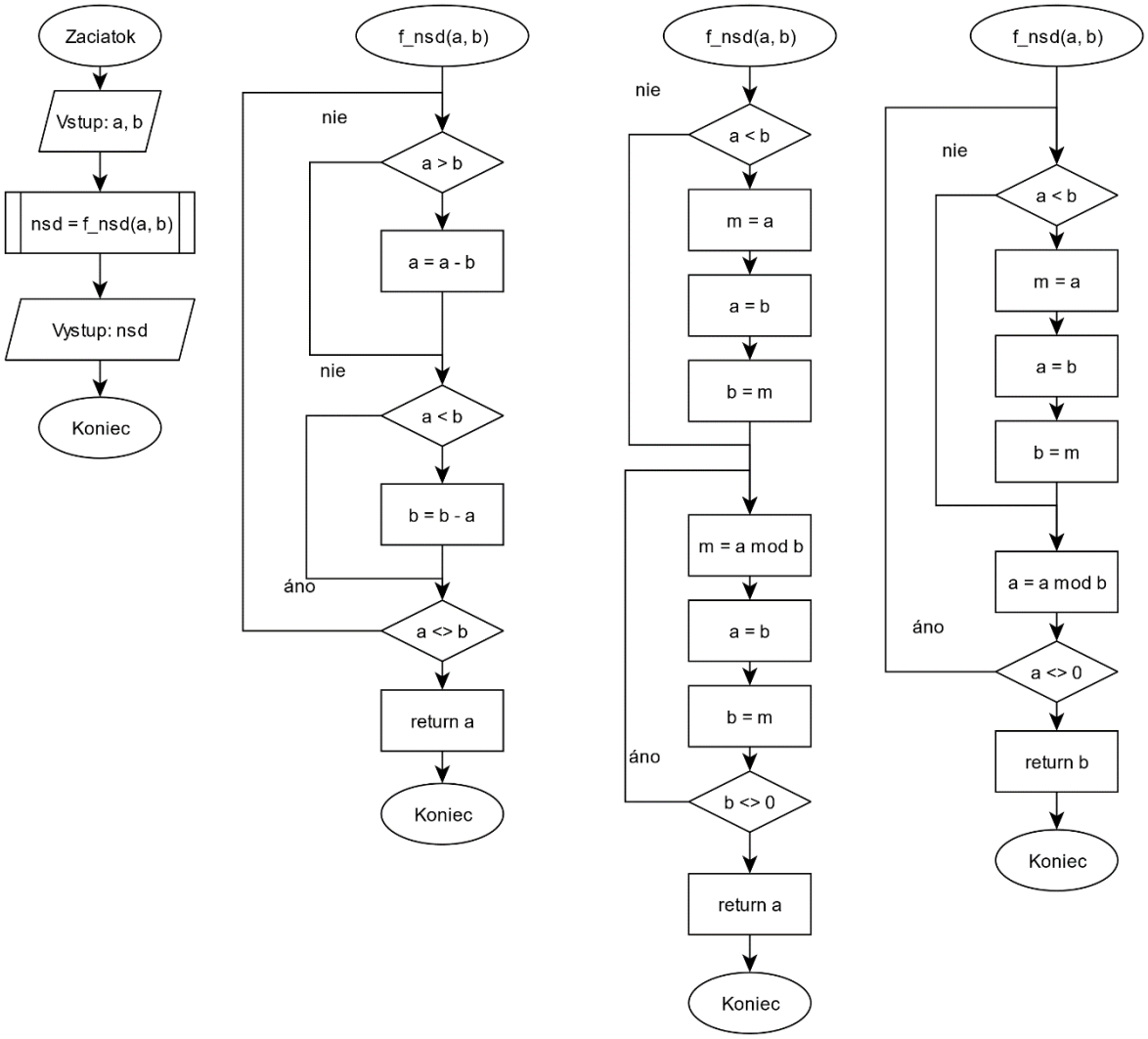
- **Determinovanosť –** táto vlastnosť algoritmu je splnená. Po načítaní vstupov, prípadnej výmeny ich hodnôt sa algoritmus realizuje predpísaným spôsobom, pričom v každom kroku je zrejmé, čo sa ma realizovať a kadiaľ pokračuje tok riadenia.
- **Konečnosť –** každý algoritmus musí skončiť po vykonaní konečného počtu krokov. V riešeniach je použitá sekvencia, neúplná podmienka a cyklus s podmienkou na konci. Pri algoritme 1 je konečnosť zabezpečená správne navrhnutými podmienkami v dôsledku ktorých nastane modifikácia premenných *a* a *b* tak, aby sa cyklus s podmienkou na konci ($a \neq b$) ukončil. V prípade použitia algoritmu 2 nastane v prípade potreby výmena hodnôt premenných *a* a *b*, následne sa vykonáva telo cyklu kde nastane modifikácia premenných *a*, *b* a *m* takým spôsobom, až nastane nesplnenie podmienky cyklu $b \neq 0$, čo spôsobí ukončenie algoritmu. Pri algoritme 3 je situácia riešená obdobne ako pri algoritme 2, avšak s rozdielnou modifikáciou premennej *a*, v dôsledku čoho nastáva aj modifikácia podmienky cyklu ($a \neq 0$), ktorá vedie k ukončeniu.
- **Rezultatívnosť** – aj táto vlastnosť je splnená. Vstupná podmienka $a, b \in N$ v prípade tvorby programu bude zabezpečená deklaráciou premenných požadovaného dátového typu: *unsigned int* a ošetrením pri načítavaní. Výstupná podmienka $nsd \in N$ bude zabezpečená deklarovaním premennej požadovaného dátového typu a druhá časť výstupnej podmienky bude zabezpečená správne navrhnutým a zrealizovaným algoritmom.

Odporúčané/očakávané vlastnosti algoritmu:

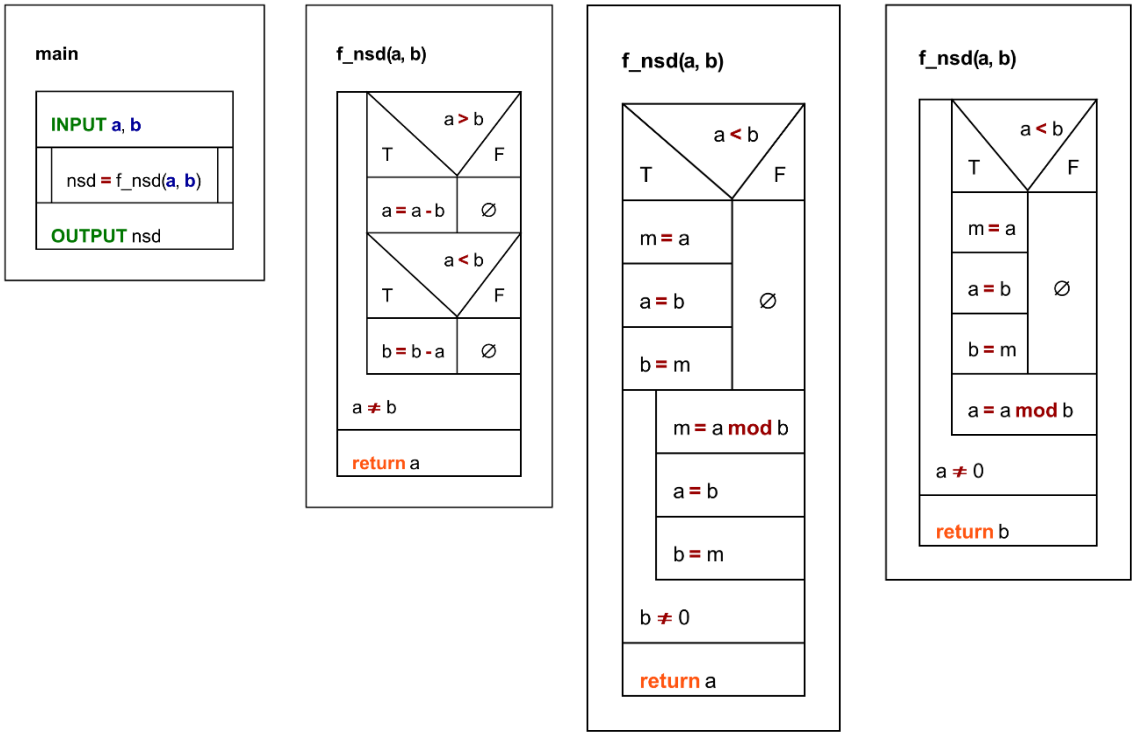
- **Hromadnosť** – môžeme konštatovať že dané riešenie spĺňa túto vlastnosť vzhľadom na charakter riešeného problému, t. j. algoritmus je platný nielen pre konkrétne hodnoty *a*, *b*, ale pre celú prípustnú množinu hodnôt (všetky prirodzené čísla), čo je zadefinované aj vstupnou podmienkou.
- **Elementárnosť** – vlastnosť je splnená, algoritmus je zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, v ktorom sa algoritmus bude realizovať (implementovať).

- **Efektívnosť** – môžeme konštatovať, že aj táto vlastnosť je splnená. V prípade, ak by sme na výmenu hodnôt pri algoritmoch 2 a 3 používali ďalšiu pomocnú premennú, algoritmus by síce bol plne funkčný, avšak zbytočne by sme navýšili priestorovú náročnosť riešenia.

  Pri posudzovaní efektívnosti by sme sa mali pozrieť aj na vlastnosti modifikovateľnosti a štruktúrovanosti, ktoré hovoria o tom, aby bol algoritmus navrhnutý tak, aby poskytol možnosť jednoduchej zmeny (úpravy) toku údajov, ako aj samotného algoritmu, a aby pozostával zo samostatných, ale logicky previazaných celkov. Aby sme úplne splnili túto vlastnosť, mali by sme kľúčové činnosti realizovať cez funkciu. Avšak pri takomto elementárnom probléme to nie je príliš žiaduce. Pre úplnosť však uvádzame aj tento spôsob návrhu algoritmu.



Obr. 53 Vývojový diagram na nájdenie najväčšieho spoločného deliteľa dvoch čísiel s pomocou funkcie.

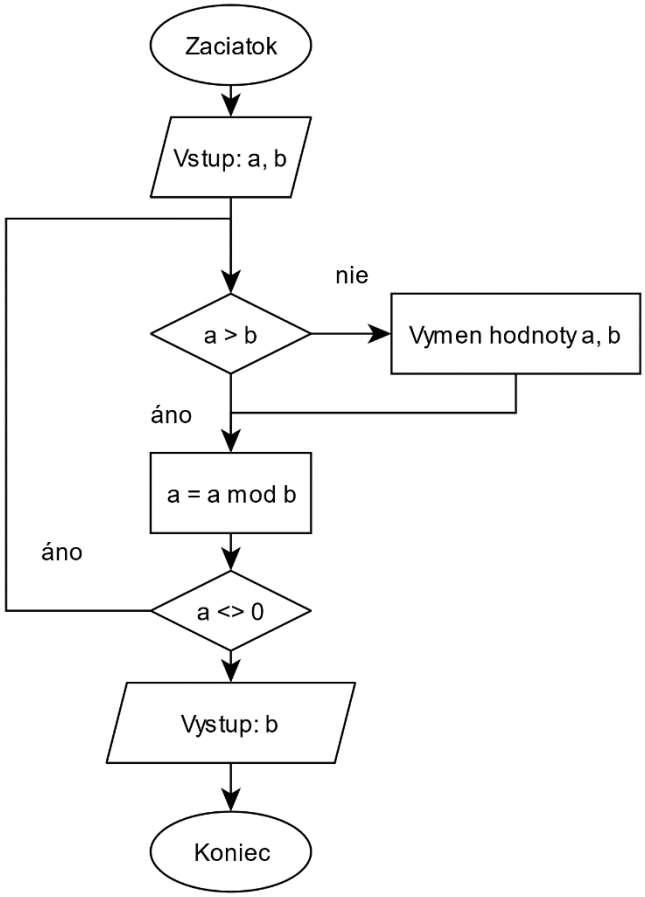


Obr. 54 NS diagram na nájdenie najväčšieho spoločného deliteľa dvoch čísiel s pomocou funkcie.

V prípade, ak sa bude realizovať hľadanie najväčšieho spoločného deliteľa čísiel *a* a *b* s pomocou funkcie, tak priestorové aj časové nároky by sme mali vyjadriť trochu inak:

Priestorová zložitosť	main()	S(1) = konštantná Používajú sa tri premenné: *a*, *b* a *nsd*.
	f_nsd()	S(1) = konštantná Funkcia má dva parametre *a*, *b*.
	f_nsd()	S(1) = konštantná Funkcia má dva parametre *a*, *b* a jednu premennú *m*.
	f_nsd()	S(1) = konštantná Funkcia má dva parametre *a*, *b* a jednu premennú *m*.
Časová zložitosť	main()	O(1) = konštantná
	f_ nsd()	O(n) = lineárna

Pozn. autora: Algoritmus na nájdenie *NSD* dvoch čísiel by mohol vyzerať aj takto:



Obr. 55 Vývojový diagram na nájdenie NSD – porušená vlastnosť elementárnosti.

V takomto prípade by sme mohli tvrdiť, že nie je splnená vlastnosť elementárnosti. Príkaz „Vymen hodnoty *a*, *b*“ je síce zrozumiteľný pre každého, ale nie každý musí vedieť ako zabezpečiť aby sa hodnoty premenných *a* a *b* vymenili.

Možné implementácie navrhnutých algoritmov:

Algoritmus 1 1.7.3_Pr_1.c:

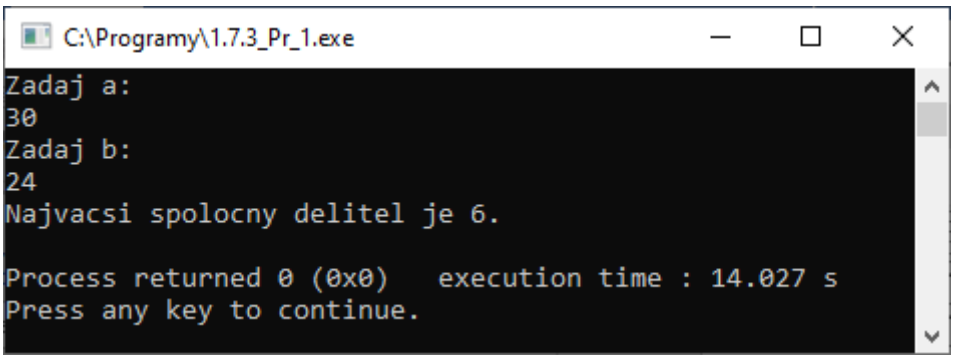
```c
#include <stdio.h>

int main(void)
{
    // deklaracia premennych
    int a, b;
    // nacitanie vstupov
    do
    {
        printf("Zadaj a:\n");
        scanf("%d", &a);
    }
    while (a <= 0);

    do
    {
        printf("Zadaj b:\n");
        scanf("%d", &b);
    }
    while (b <= 0);

    // cyklus na hladanie nsd
    do
    {
        if (a > b)
            a = a - b;
        if (a < b)
            b = b - a;
    }
    while (a != b);

    // vypisanie vysledku
    printf("Najvacsi spolocny delitel je %d. \n", a);

    // program skoncil uspesne
    return 0;
}
```

Algoritmus 2 1.7.3_Pr_2.c:

```c
#include <stdio.h>

int main(void)
{
    // deklaracia premennych
    int a, b, m;

    // nacitanie vstupov
    do
    {
        printf("Zadaj a:\n");
        scanf("%d", &a);
    }
    while (a <= 0);

    do
    {
        printf("Zadaj b:\n");
        scanf("%d", &b);
    }
    while (b <= 0);

    if (a < b)
    {
        m = a;
        a = b;
        b = m;
    }
    // cyklus na hladanie nsd
    do
    {
        m = a % b;
        a = b;
        b = m;
    }
    while (b != 0);

    // vypisanie vysledku
    printf("Najvacsi spolocny delitel je %d. \n", a);

    // program skoncil uspesne
    return 0;
}
```

Algoritmus 3 1.7.3_Pr_3.c:

```c
#include <stdio.h>

int main(void)
{
    // deklaracia premennych
    int a, b, m;

    // nacitanie vstupov
    do
    {
        printf("Zadaj a:\n");
        scanf("%d", &a);
    }
    while (a <= 0);

    do
    {
        printf("Zadaj b:\n");
        scanf("%d", &b);
    }
    while (b <= 0);

    // cyklus na hladanie nsd
    do
    {
        if (a < b)
        {
            m = a;
            a = b;
            b = m;
        }
        a = a % b;
    }
    while (a != 0);

    // vypisanie vysledku
    printf("Najvacsi spolocny delitel je %d. \n", b);

    // program skoncil uspesne
    return 0;
}
```



Obr. 56 Konzolový výstup programu 1.7.3_Pr_1.c, 1.7.3_Pr_2.c a 1.7.3_Pr_3.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú na riešenie Euklidovho algoritmu s pomocou algoritmu 1 8 B = *a* (*int* = 4 B) + *b* (*int* = 4 B). V prípade použitia algoritmu 2 a 3 sú priestorové nároky o 4 B vyššie, z dôvodu použitia premennej *m* (*int* = 4 B), t. j. celkovo 12 B.

1.8 Príklady na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problém, ktorý načíta štyri čísla a vypíše číslo, ktoré sa rovná súčinu súčtu prvých dvoch čísiel a rozdielu druhých dvoch čísel.
2. Algoritmicky riešte problém výpočtu obvodu a obsahu štvorca.
3. Algoritmicky riešte problém výpočtu priemeru z dvoch čísel, v prípade zadania záporného čísla uvažujte a pracujte s jeho absolútnou hodnotou.
4. Algoritmicky riešte problém, ktorý načíta tri čísla a vypíše najväčšie z nich.
5. Algoritmicky riešte problém, ktorý načíta prirodzené číslo a rozhodne, či je jednociferné, dvojciferné alebo viacciferné.
6. Algoritmicky riešte problém, ktorý pozdraví podľa zadaného času pošas dňa vyjadreného v hodinách. Dopoludnia do 12 hodín pozdraví „dobré dopoludnie“, do 18 hodín „dobré popoludnie“, neskôr „dobrý večer“.
7. Algoritmicky riešte problém, ktorý vypíše, koľko minút uplynulo medzi dvoma časovými údajmi, napr. medzi 9:30 a 11:15 uplynulo 105 minút.
8. Algoritmicky riešte problém, ktorý vypočíta druhú mocninu ľubovoľného nezáporného čísla s pomocou sčítania.
9. Algoritmicky riešte problém, ktorý načíta tri čísla a vypíše ich na obrazovku zostupne (od najväčšieho po najmenšie).
10. Algoritmicky riešte problém, ktorý po zadaní dĺžok troch strán trojuholníka zistí či platí trojuholníková nerovnosť alebo nie, a následne určí o aký typ trojuholníka ide.
11. Algoritmicky riešte problém, ktorý vypočíta súčet čísiel od načítanej dolnej po načítanú hornú hranicu. Výsledok vypíšte. Riešte situáciu, ak používateľ zadá dolnú hranicu väčšiu ako hornú (aj tak realizovať výpočet).
12. Algoritmicky riešte problém, ktorý zistí od používateľa výšku jeho peňazí, aby vedel rozhodnúť, či má na vybraný nápoj alebo nie. Nápoj stojí 1€. V prípade ak na nápoj má,

tak mu o tom zobraziť výpis a vypočítať koľko mu ostane peňazí, v opačnom prípade mu oznámiť, že na nápoj nemá.

13. Algoritmicky riešte problém, ktorý v postupnosti čísiel zistí počet čísiel deliteľných dvomi a počet čísiel deliteľných tromi. Postupnosť čísiel načítavajte sekvenčne a vkladanie ukončite vložením hodnoty 0.
14. Algoritmicky riešte problém, ktorý vypíše všetky párne čísla pre zadaný interval ⟨a,b⟩.
15. Algoritmicky riešte problém, ktorý zistí koľkokrát sa celé číslo *k* (zadané používateľom) nachádza v postupnosti čísiel. Postupnosť čísiel načítavajte sekvenčne a vkladanie ukončite vložením hodnoty 0.
16. Algoritmicky riešte problém, ktorý pre zadané číslo vypíše všetky jeho násobky menšie alebo rovné ako určená hranica.
17. Algoritmicky riešte problém, ktorý vypočíta ciferný súčet zadaného prirodzeného čísla.
18. Algoritmicky riešte problém, ktorý bude porovnávať dvojice po sebe nasledujúcich čísiel v postupnosti čísiel a označí vzťah medzi nimi (<, >, =). Postupnosť čísiel načítavajte sekvenčne a vkladanie ukončite vložením hodnoty 0.
19. Algoritmicky riešte problém, ktorý bude zisťovať, či zadané čísla sú z intervalu ⟨1, 10⟩. Postupnosť čísiel načítavajte sekvenčne a vkladanie ukončite vložením hodnoty 0.
20. Algoritmicky riešte problém, ktorý načíta reálne číslo, a ak je toto číslo väčšie ako jedna, vypíše jeho dvojnásobok, inak vypíše jeho polovicu.
21. Algoritmicky riešte problém, ktorý pre zadané *a* a *b* vypíše tabuľku násobenia (všetky súčiny od *0 . 0* až *a . b*).
22. Algoritmicky riešte problém, ktorý pred zadané čísla vypíše, do ktorého z intervalov $(-\infty, 0\rangle$, $(0, 1\rangle$, $(1, 9\rangle$, $(9, 10\rangle$, $(10, \infty)$ patria. Postupnosť čísiel načítavajte sekvenčne a vkladanie ukončite vložením hodnoty 0.
23. Algoritmicky riešte problém, ktorý pre zadané hodnotenie predmetu (A, B, C, D, E, X) vypíše, či sa študentovi predmet môže na inej škole uznať (hodnotenia A až D), alebo nemôže (hodnotenia E a X).
24. Algoritmicky riešte problém, ktorý bude načítavať čísla tak dlho, kým nebude zadaná nula. Zadané čísla spočíta a vypíše výsledok.
25. Algoritmicky riešte problém, ktorý bude načítavať čísla tak dlho, kým nebude zadaná nula. Následne vypíše, koľko je v nej nepárnych čísiel deliteľných tromi.

26. Algoritmicky riešte problém, ktorý vypočíta faktoriál zadaného prirodzeného čísla.
27. Algoritmicky riešte problém, ktorý určí, či je zadané prirodzené číslo prvočíslo.
28. Algoritmicky riešte problém, ktorý vypíše všetky delitele zadaného prirodzeného čísla.
29. Algoritmicky riešte problém, ktorý spočíta čísla od 1 do n. Čislo n bude zadané používateľom.
30. Algoritmicky riešte problém, ktorý načíta n čísel a vypočíta ich aritmetický priemer.
31. Algoritmicky riešte problém, ktorý načíta n čísel a zistí najväčšie z nich.
32. Algoritmicky riešte problém, ktorý vypočíta n-tú mocninu čísla x.
33. Algoritmicky riešte problém, ktorý umožní zadať n čísiel. Následne vypíše číslo s najväčšou absolútnou hodnotou.
34. Algoritmicky riešte problém, ktorý číslo zadané v dvojkovej sústave prevedie do sústavy desiatkovej.

2 Časová a priestorová zložitosť

V prvom rade chcem upriamiť pozornosť čitateľa na fakt, že schopnosť presne opísať výkonnosť zložitých algoritmov s pomocou matematickej analýzy často vyžaduje vedomosti, ktoré prekračujú rozsah/záber tejto učebnice. Je dôležité uvedomiť si, že pokiaľ chceme porovnávať rôzne algoritmy, musíme stáť na pevnom vedeckom základe. Tiež je možné dohľadať detailné informácie o bežne používaných a známych algoritmoch v literatúre. Mojim primárnym cieľom bolo poukázať na súvislosti a možnosti, ktoré potrebujeme, aby sme mohli inteligentne a logicky, bez potreby nejakých odborných vedomostí pracovať s algoritmami.

Existujú mnohé algoritmy, ktoré nemajú jednoznačne vhodné riešenie, a je možné ich riešiť viacerými spôsobmi. V takomto prípade je nevyhnutné zvážiť každé jedno možné riešenie a podrobiť jednotlivé varianty riešenia detailnejšej analýze. Pri analýze algoritmov je nevyhnutné sa sústrediť na ich efektivitu, pri ktorej posudzujeme priestorovú a časovú zložitosť samotného algoritmu, ako aj jednoduchosť a efektívnosť samotnej implementácie, t. j. vytvorenie programu priamo v konkrétnom programovacom jazyku. Efektívnosť algoritmu priamo nesúvisí s náročnosťou samotného algoritmu z ľudského pohľadu. Algoritmus môže byť veľmi jednoduchý na pochopenie i implementáciu, napriek tomu môže byť jeho výpočtová zložitosť veľká a naopak.

Efektívnosť algoritmu môže byť analyzovaná na dvoch úrovniach: pred samotnou implementáciu, alebo po nej. Pred implementáciou ide viac menej o teoretický rozbor algoritmu. Posudzovanie efektívnosti algoritmu po implementácii, t. j. skutočný čas vykonania programu závisí od mnohých faktorov. Často sa posudzuje za predpokladu, že všetky ostatné faktory, ako napr. rýchlosť procesora, sú konštantné a nemajú žiadny vplyv na implementáciu. Čas vykonávania jednej operácie algoritmu by sa mal určiť v závislosti od vykonávania počtu inštrukcií, ktoré je stroj schopný vykonať za konkrétnu časovú jednotku, napr. jednu sekundu. To vzhľadom na variabilitu technického vybavenia v súčasnosti nie je vôbec ľahké. Aj z tohto dôvodu nebudem exaktne posudzovať čas vykonania jednotlivej operácie algoritmu, ale sledovať len rád rastu danej funkcie podľa ktorej budem posudzovať časovú zložitosť.

Časová zložitosť = **Time Complexity** je čas vykonávania jednotlivých operácií v závislosti od veľkosti riešeného problému, alebo veľkosti vstupných údajov. Neudávame ju v bežných časových jednotkách, pretože skutočný čas výpočtu programu realizujúceho navrhnutý algoritmus závisí od technických parametrov, ako je napríklad rýchlosť procesora použitého počítača, zatiaľ čo my chceme charakterizovať rýchlosť práce algoritmu samého. Presnejšie je

časový faktor teda definovaný počtom všetkých elementárnych operácií (aritmetických, logických, operácie porovnania, priradenia a pod.), ktoré algoritmus vykoná pri výpočte nad danými vstupnými údajmi. Musí byť čo najmenší. „Časová zložitosť algoritmu *A* je teda funkcia *tA*, ktorá každej hodnote *n*, charakterizujúcej veľkosť spracovávaných údajov, priraďuje hodnotu *tA(n),* čo je počet operácií, ktoré algoritmus *A* vykoná pri spracovaní údajov veľkosti *n*“ (Gunišová a Guniš, 2019).

Časovú zložitosť algoritmu je nevyhnutné posudzovať vzhľadom na vstupné dáta pre najhorší, priemerný a najlepší prípad.

- **Najhorší prípad = Worst Case** – udáva ako najdlhšie môže trvať výpočet podľa algoritmu pre vstupné dáta s veľkosťou *n*, t. j. je to teda akési horné ohraničenie, ktoré algoritmus neprekročí. Napríklad, ak sú v programe dva vnorené cykly, z ktorých každý sa vykoná najviac *n*-krát, potom taký algoritmus má horný odhad časovej zložitosti n^2.
- **Priemerný prípad = Average Case**
- **Najlepší prípad = Best Case** – udáva, ako najkratšie môže trvať výpočet podľa algoritmu pre vstupné dáta o veľkosti *n*.

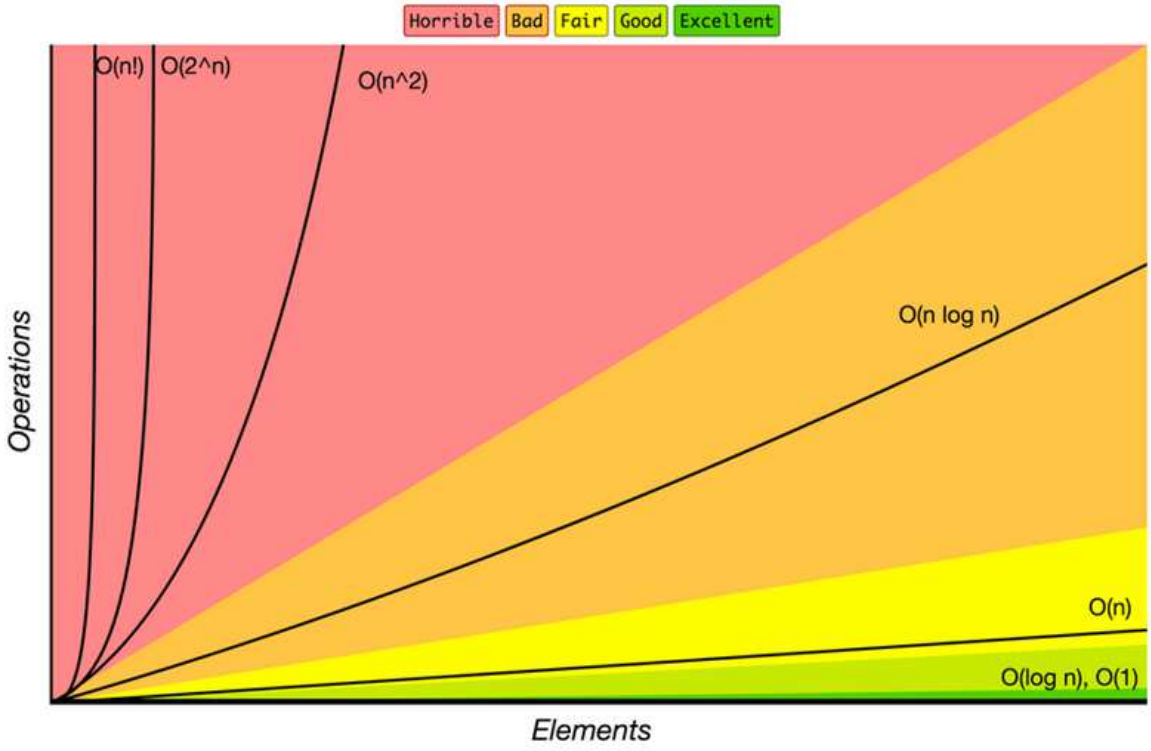
V prípade, ak chceme rozhodnúť, ktorý z algoritmov s rôznou časovou zložitosťou je lepší, je pre nás spravidla podstatné, ktorý algoritmus je rýchlejší pre veľké hodnoty *n*, pretože malé údaje bude takmer vždy možné spracovať v rozumnom čase. Podstatné je, ako sa správa funkcia vyjadrujúca časovú zložitosť algoritmov pre rastúce hodnoty *n*. Hovoríme potom o **asymptotickej časovej zložitosti** a z dvoch algoritmov považujeme za lepší ten, ktorý má asymptotickú zložitosť menšiu. Je to teda ten, ktorého funkcia časovej zložitosti rastie pomalšie s rastúcim *n*. Pri stanovení asymptotickej zložitosti algoritmov je podstatná iba rýchlosť rastu prislúchajúcej funkcie zložitosti. Napríklad o algoritme, ktorého časovú zložitosť charakterizuje predpis $3n^2 + 2n + 1$ stačí zistiť, že s rastúcim *n* sú jeho časové nároky úmerné n^2, čiže prislúchajúca funkcia časovej zložitosti je kvadratická, čo zapisujeme v tvare $O(n^2)$. Môžeme si všimnúť, že ak má presné vyjadrenie časovej zložitosti tvar súčtu, potom rýchlosť rastu tejto funkcie je určená rýchlosťou rastu najrýchlejšie rastúceho člena. Napríklad u mnohočlena (polynómu) je to člen s najvyššou mocninou *n*.

Pritom je ale možné a bežne sa stáva, že pre niektoré malé vstupné údaje pracuje algoritmus s väčšou asymptotickou časovou zložitosťou rýchlejšie. Keby mal napríklad algoritmus *A1* časovú zložitosť $tA1 = n^2$ a algoritmus *A2* s časovou zložitosťou $tA2 = 10 \cdot n$, má určite *A1* väčšiu asymptotickú časovú zložitosť ako *A2* (funkcia *tA1* rastie rýchlejšie než *tA2* pre

rastúce n), ale pre $n < 10$ je $tA1(n) < tA2(n)$, a teda *A1* pracuje nad malými vstupnými údajmi rýchlejšie ako *A2*. Pri praktickom riešení úloh na počítači je preto potrebné pri voľbe vhodných algoritmov zvažovať aj to, aké veľké vstupné údaje bude program spracovávať. Ak na veľkosť očakávaných vstupných údajov nie je vopred dané žiadne obmedzenie, zvolíme samozrejme algoritmus s menšou asymptotickou časovou zložitosťou (Gunišová a Guniš, 2019).

V praxi používané algoritmy majú väčšinou niektorú z nasledujúcich časových zložitostí:

- konštantnú *O(1)* – počet operácií je pre ľubovoľne veľké vstupné dáta zhruba rovnaký, t. j. algoritmus nezávisí od veľkosti vstupu. Typicky ide o jednoduché matematické výpočty, napr. hľadanie koreňov kvadratickej rovnice, pridanie prvku do neutriedeného poľa a pod.,
- logaritmickú *O(log n)* – algoritmus začína pracovať pomalšie pri zvýšení veľkosti problému. Takúto zložitosť majú často problémy, ktoré sa riešia dekompozíciou na menšie podroblémy – technika „rozdeľuj a panuj (*divide and conquer*)“. Napríklad hľadanie prvku v utriedenom poli (binárne vyhľadávanie) a pod.,
- lineárnu *O(n)* – náročnosť algoritmu sa zvyšuje podobne ako veľkosť vstupných dát. Typickým príkladom je spracovanie prvkov poľa, hľadanie extrémov v poli, sekvenčné vyhľadávanie a pod.
- *O(n . log n)* – touto zložitosťou sa vyznačujú napr. vylepšené metódy triedenia poľa,
- kvadratickú $O(n^2)$ – takouto zložitosťou sa vyznačujú algoritmy, v ktorých sú použité dva vnorené cykly,
- kubickú $O(n^3)$ – takouto zložitosťou sa vyznačujú algoritmy, v ktorých sú použité tri vnorené cykly, napr. násobenie matíc,
- polynomiálnu $O(n^k)$, $k \in \mathrm{R}$
- exponenciálnu $O(k^n)$, $k \in \mathrm{R}$ – tiež označovaná ako algoritmus hrubej sily, napr. algoritmus riešenia hanojských veží,
- faktoriálovú *O(n!)*.



Obr. 57 Typické rády časovej zložitosti algoritmov (https://www.bigocheatsheet.com/).

Typické rády časovej zložitosti algoritmov v číslach podľa Java Concept Of The Day (2020):

Časová zložitosť	Počet operácií pre 10 prvkov	Počet operácií pre 100 prvkov	Počet operácií pre 1 000 prvkov
O(1)	1	1	1
O(log n)	3	6	9
O(n)	10	100	1 000
O(n log n)	30	600	9 000
O(n^2)	100	10 000	1 000 000
O(2^n)	1 024	$1{,}26 \cdot 10^{29}$	$1{,}07 \cdot 10^{301}$
O(n!)	3 628 800	$9{,}3 \cdot 10^{157}$	$4{,}02 \cdot 10^{2\,567}$

Inými slovami, asymptotickú zložitosť je možné predstaviť si ako rýchlosť rastu danej funkcie a na jej základe zaradenie algoritmu do nejakej kategórie (lineárna, exponenciálna, logaritmická…), ktoré umožňujú porovnávať efektivitu algoritmov medzi sebou, a to nezávisle na implementácii a použitej platforme. Zatiaľ čo hodnota polynómu (lineárneho, kvadratického, kubického…) hoci aj vyššieho stupňa býva pre bežné *n* ešte prijateľná

a prislúchajúci algoritmus s takouto časovou zložitosťou býva z hľadiska trvania výpočtu použiteľný, rýchlosť rastu exponenciálnej funkcie je taká veľká, že exponenciálne algoritmy možno použiť len pre veľmi malé *n*. Pri návrhu algoritmu sa preto snažíme vyhýbať exponenciálnym algoritmom všade, kde je to možné (Gunišová a Guniš, 2019).

V prípade, ak dva posudzované algoritmy majú približne rovnakú časovú zložitosť, vždy je lepšie vybrať ten lepší, najmä ak pracujeme s veľkým počtom údajov. Typickým príkladom je triedenie údajov. Jednoduchšie triediace algoritmy majú časovú zložitosť kvadratickú $O(n^2)$, kým tie lepšie $O(n \cdot log\ n)$, Čiže prvý z nich urobí pri utriedení n čísel n^2 operácií porovnania a druhý $n \cdot log\ n$ porovnaní . Pokiaľ by na istom počítači trvalo jedno porovnanie 0,1 ms, potom utriedenie 100 čísel s pomocou prvého algoritmu by trvalo 1 sekundu a s pomocou druhého algoritmu 0,07 s. Toto nie je veľký rozdiel, avšak ak by sme potrebovali utriediť 100 000 čísel, rozdiel medzi oboma algoritmami bude zásadný: program s kvadratickou zložitosťou by počítal viac ako 11 dní, zatiaľ čo druhý algoritmus by urobil prácu za necelé tri minúty.

Analýza efektívnosti algoritmu po implementácii predstavuje empirickú analýzu algoritmu. Daný algoritmus je realizovaný s pomocou konkrétneho programovacieho jazyka. Následne je tento opakovane spustený na konkrétnom počítači. Získané dáta je možné podrobiť základnej štatistike a analyzovať čas potrebný na činnosť programu, aj pamäť potrebnú na ukladanie dát pri rôznych veľkostiach vstupných dát/postupností.

Tak ako uvádza Forišek (2018) tento prístup môže byť veľmi nepraktický, najmä ak s nim začneme ako s prvým. Prináša nasledovné úskalia:

1. Nechceme programovať všetky riešenia, ktoré nás napadnú pre riešenie daného problému, ale práve naopak. Chceme identifikovať to najlepšie a až to programovať.
2. Riešenie nemusí byť použiteľné, pretože sa môže stať, že pre veľké dáta toto riešenie a jeho činnosť môže trvať neúmerne dlho – najmä ak algoritmus riešenia disponuje väčšou asymptotickou časovou zložitosťou.
3. Výsledky môžu byť nepresné. Rýchlosť činnosti programu závisí od viacerých faktorov (napr. momentálna záťaž procesora inými úlohami) a otestovaním na konkrétnych vstupných dátach sa dozvieme len to, ako sa správajú pre tento konkrétny vstup. Tieto nepresnosti sa však dajú minimalizovať.

Priestorová zložitosť = **Space Complexity** je vyjadrená množstvom pamäťového priestoru požadovaného samotným algoritmom počas jeho životného cyklu. Často krát ju vyjadrujeme

v bajtoch B, resp. v bitoch b. Pamäťový priestor je daný súčtom fixnej pamäte potrebnej na uloženie dát a premenných, ktoré sú nezávislé od veľkosti riešeného problému (napr. premenné a konštanty, pre ktoré vieme jednoznačne vyčísliť veľkosť potrebnej pamäte, samotná veľkosť programu daná zdrojovým kódom a pod.), a premennej časti, ktorú vyžadujú premenné závislé od veľkosti riešeného problému, napr. v prípade dynamickej alokácie pamäte pre dáta, ktorých veľkosť dopredu nepoznáme, potreby pamäte pre zásobník v prípade použitia rekurzie (nevieme dopredu aká hlboká rekurzia bude) a pod. Cieľom je aby bola čo najmenšia. Priestorová (pamäťová) zložitosť „algoritmu *A* je teda opäť istou funkciou *mA*, ktorá veľkosti vstupných údajov *n* priraďuje potrebný počet pamäťových miest *mA*(*n*), ktoré bude algoritmus *A* pri výpočte s týmito údajmi využívať“ (Gunišová a Guniš, 2019).

Aj keď sa pri posudzovaní efektívnosti algoritmu obmedzíme iba na časovú a priestorovú zložitosť algoritmov pri hľadaní toho najlepšieho, môžeme sa dostať do ťažkej situácie, pretože tieto dve kritériá často stoja proti sebe. Stáva sa totiž, že na zrýchlenie výpočtu musíme použiť nejakú pomocnú dátovú štruktúru slúžiacu na uchovávanie vopred spočítaných a pripravených hodnôt. Najrýchlejší možný algoritmus riešiaci úlohu potom nie je optimálny z hľadiska pamäťových nárokov a naopak, algoritmus s najmenšou pamäťovou zložitosťou zase nemusí byť najrýchlejší. V súčasnosti sa obyčajne dáva prednosť časovému hľadisku a hľadajú sa čo najrýchlejšie algoritmy, a to i za cenu vyššej spotreby pamäte (Gunišová a Guniš, 2019).

2.1 Časová zložitosť algoritmov - príklady

Nájdenie maximálneho prvku v poli *n* prvkov:

```
max = a[0];
for (i = 1; i < n; i++)
    if (max < a[i])
        max = a[i];
```

Ak je elementárnou operáciou porovnanie, tak:

- $T(n) = n - 1$ *operácií*

Ak je elementárnou operáciou priradenie, tak:

- $T(n) = 1$ – najlepší prípad
- $T(n) = n$ – najhorší prípad
- $T(n) = (1 + n) / 2$ *priradení* – priemer

Nájdenie maximálneho prvku v štvorcovej matici s rozmerom *n*:

```
max = a[0][0];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (max < a[i][j])
            max = a[i][j];
```

Ak je elementárnou operáciou porovnanie, tak:

- $T(n) = n \,.\, n = n^2$ *operácií*

Ak je elementárnou operáciou priradenie, tak:

- $T(n) = 1$ – najlepší prípad
- $T(n) = n^2$ – najhorší prípad
- $T(n) = (1 + n^2) / 2$ *priradení – priemer*

3 Zoznam použitej literatúry a informačných zdrojov

BELAN, Anino, 2011. Kurz Jazyk C. Učebný text pre kvartu a kvintu osemročného gymnázia. 2. vydanie. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://docplayer.gr/24355101-Kurz--jazyka-c-ucebny-text-pre-kvartu-a-kvintu-osemrocneho-gymnazia.html⟩.

Encyclopedia Britannica, 2020. ASCII communications. [online]. [cit. 16.8.2020]. Dostupné na: ⟨https://www.britannica.com/topic/ASCII⟩.

FORIŠEK, Michal. Časová zložitosť. [online]. [cit. 14.11.2020]. Dostupné na: ⟨http://foja.dcs.fmph.uniba.sk/ads/materialy.php⟩.

GUNIŠOVÁ, Valentína a Ján GUNIŠ. Niekoľko slov o efektivite algoritmov. Oddelenie didaktiky informatiky a podporných technológií. Univerzita P. J. Šafárika. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://di.ics.upjs.sk/⟩.

HÁLA, Tomáš, 2002. Učebnice Pascalu. Praha : Computer Press. Vydaní druhé. ISBN 80-226-733-7.

HEROUT, Pavel, 2010. Učebnice jazyka C 1. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-383-8.

HEROUT, Pavel, 2010. Učebnice jazyka C 2. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-367-8.

HOROVČÁK, Pavel a Igor PODLUBNÝ, 1997. Úvod do programovania v jazyku C. [online]. [cit. 26.7.2019]. Dostupné na: ⟨http://people.tuke.sk/igor.podlubny/C/⟩.

CHVALOVSKÝ, Václav, 1984. Rozhodovací tabulky. SNTL, Praha.

JANOUŠEK, Jaroslav, ČEKANOVÁ, Adriana a Viera PALMÁROVÁ. Učebnica jazyka C. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://spseke.sk/tutor/projekt/⟩.

JAVA CONCEPT OF THE DAY, 2020. A Beginner's Guide To Big O Notations. . [online]. [cit. 3.8.2020]. Dostupné na: ⟨https://javaconceptoftheday.com/big-o-notations-tutorials/⟩.

KERNIGHAN, W. Brian a Dennis M. RITCHIE, 2019. Programovací jazyk C. Brno : Computer Press. ISBN 978-80-251-4965-2.

Know The Complexities! [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://www.bigocheatsheet.com/⟩.

KNUTH, Donald Ervin, 2008. Umění programování 1. díl, Základní algoritmy. Brno : Computer Press, ISBN 978-80-251-2025-5.

MACHOVÁ, Jana a Mária SPIŠÁKOVÁ, 2011. Procedúry a funkcie v Pascale (Zbierka úloh) Dostupné na: ⟨https://encyklopediapoznania.sk/data/eknihy/informatika/procedury_a_funkcie_v_pascale2_-_zbierka_uloh.pdf⟩.

SEDGEWICK, Robert, 2003. Algoritmy v C, Části 1–4, Základy datové struktury, třídení, vyhledávaní. Praha : SoftPress s. r. o. ISBN 80-86497-56-9.

WIRTH, Niklaus, 1989. Algoritmy a štruktúry údajov. Bratislava, Alfa. 2. vydanie. ISBN 80-05-00153-3.

WRÓBLEVSKI, Piotr, 2004. Algoritmy. Datové struktury a programovací techniky. Brno : Computer Press. ISBN 80-251-0343-9.

ZREBNÝ, Rudolf, 2010a. Riešenie kvadratickej rovnice. Pohodová matematika. © 2019. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://pohodovamatematika.sk/riesenie-kvadratickej--rovnice.html⟩.

ZREBNÝ, Rudolf, 2010b. Riešenie kvadratickej rovnice v normovanom tvare. Pohodová matematika. © 2019. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://pohodovamatematika.sk/riesenie-kvadratickej-rovnice-v-normovanom-tvare.html⟩.

Názov:	Algoritmizácia a základy štruktúrovaného programovania v jazyku C 1. diel (vysokoškolská učebnica)
Autor:	Ing. Jana Jurinová, PhD.
Recenzenti:	doc. Ing. Michal Čerňanský, PhD. Mgr. Ing. Roman Horváth, PhD.
Vydavateľ:	Univerzita sv. Cyrila a Metoda v Trnave
Rok:	2020
Rozsah:	130 strán/ 7,39 AH
Náklad:	50 ks
Grafická úprava obálky:	Ing. Jana Jurinová, PhD.
Tlač:	elektronická verzia